1.1 What are "FIR filters"?

FIR filters are one of two primary types of digital filters used in Digital Signal Processing (DSP) applications (the other type being IIR).

1.2 What does "FIR" mean?

"FIR" means "Finite Impulse Response".

1.3 Why is the impulse response "finite"?

The impulse response is "finite" because there is no feedback in the filter; if you put in an impulse (that is, a single "1" sample followed by many "0" samples), zeroes will eventually come out after the "1" sample has made its way in the delay line past all the coefficients.

1.4 How do I pronounce "FIR"?

Some people say the letters F-I-R; other people pronounce as if it were a type of tree. We prefer the tree. (The difference is whether you talk about *an* F-I-R filter or *a* FIR filter.)

1.5 What is the alternative to FIR filters?

DSP filters can also be "Infinite Impulse Response" (IIR). (See dspGuru's IIR FAQ.) IIR filters use feedback, so when you input an impulse the output theoretically rings indefinitely.

1.6 How do FIR filters compare to IIR filters?

Each has advantages and disadvantages. Overall, though, the advantages of FIR filters outweigh the disadvantages, so they are used much more than IIRs.

1.6.1 What are the *advantages* of FIR Filters (compared to IIR filters)?

Compared to IIR filters, FIR filters offer the following advantages:

- They can easily be designed to be "linear phase" (and usually are). Put simply, linear-phase filters delay the input signal, but don't distort its phase.
- They are simple to implement. On most DSP microprocessors, the FIR calculation can be done by looping a single instruction.
- They are suited to multi-rate applications. By multi-rate, we mean either "decimation" (reducing the sampling rate), "interpolation" (increasing the sampling rate), or both. Whether decimating or interpolating, the use of

FIR filters allows some of the calculations to be omitted, thus providing an important computational efficiency. In contrast, if IIR filters are used, each output must be individually calculated, even if it that output will discarded (so the feedback will be incorporated into the filter).

- They have desireable numeric properties. In practice, all DSP filters must be implemented using "finite-precision" arithmetic, that is, a limited number of bits. The use of finite-precision arithmetic in IIR filters can cause significant problems due to the use of feedback, but FIR filters have no feedback, so they can usually be implemented using fewer bits, and the designer has fewer practical problems to solve related to non-ideal arithmetic.
- They can be implemented using fractional arithmetic. Unlike IIR filters, it is always possible to implement a FIR filter using coefficients with magnitude of less than 1.0. (The overall gain of the FIR filter can be adjusted at its output, if desired.) This is an important considertaion when using fixed-point DSP's, because it makes the implementation much simpler.

1.6.2 What are the *disadvantages* of FIR Filters (compared to IIR filters)?

Compared to IIR filters, FIR filters sometimes have the disadvantage that they require more memory and/or calculation to achieve a given filter response characteristic. Also, certain responses are not practical to implement with FIR filters.

1.7 What terms are used in describing FIR filters?

- *Impulse Response* - The "impulse response" of a FIR filter is actually just the set of FIR coefficients. (If you put an "impluse" into a FIR filter which consists of a "1" sample followed by many "0" samples, the output of the filter will be the set of coefficients, as the 1 sample moves past each coefficient in turn to form the output.)
- *Tap* - A FIR "tap" is simply a coefficient/delay pair. The number of FIR taps, (often designated as "N") is an indication of 1) the amount of memory required to implement the filter, 2) the number of calculations required, and 3) the amount of "filtering" the filter can do; in effect, more taps means more stopband attenuation, less ripple, narrower filters, etc.)
- *Multiply-Accumulate (MAC)* - In a FIR context, a "MAC" is the operation of multiplying a coefficient by the corresponding delayed data sample and accumulating the result. FIRs usually require one MAC per tap. Most DSP microprocessors implement the MAC operation in a single instruction cycle.
- *Transition Band* - The band of frequencies between passband and stopband edges. The narrower the transition band, the more taps are required to implement the filter. (A "small" transition band results in a "sharp" filter.)

- *Delay Line* - The set of memory elements that implement the "Z^-1" delay elements of the FIR calculation.
- *Circular Buffer* - A special buffer which is "circular" because *incrementing* at the end causes it to wrap around to the beginning, or because *decrementing* from the beginning causes it to wrap around to the end. Circular buffers are often provided by DSP microprocessors to implement the "movement" of the samples through the FIR delay-line without having to literally move the data in memory. When a new sample is added to the buffer, it automatically replaces the oldest one.

4.1 What is the basic algorithm for implementing FIR filters?

Structurally, FIR filters consist of just two things: a sample delay line and a set of coefficients. To implement the filter:

1. Put the input sample into the delay line.
2. Multiply each sample in the delay line by the corresponding coefficient and accumulate the result.
3. Shift the delay line by one sample to make room for the next input sample.

4.2 How do I implement FIR filters in C?

There are lots of possibilities, including a few tricks. To illustrate, we have provided a set of FIR filter algorithms implemented in C called "FirAlgs.c" in ScopeFIR's distribution file. FirAlgs.c includes the following functions:

1. **fir_basic**: Illustrates the basic FIR calculation described above by implementing it very literally.
2. **fir_circular**: Illustrates how circular buffers are used to implement FIRs.
3. **fir_shuffle**: Illustrates the "shuffle down" technique used by some of Texas Instruments' processors.
4. **fir_split**: Splits the FIR calculation into two flat (non-circular) pieces to avoid the use circular buffer logic and shuffling.
5. **fir_double_z**: Uses a double-sized delay line so that the FIR calculation can be done using a flat buffer.
6. **fir_double_h**: Similar to fir_double_z, this uses a double-sized coefficient so that the FIR calculation can be done using a flat buffer.

4.3 How do I implement FIR filters in assembly?

FIR assembly algorithms are quite processor-specific, but the most common system uses a circular buffer mechanism provided by the DSP processor. The basic steps are:

1. Configure the circular buffer; load the coefficient and delay-line pointers. Then, for each input sample:
2. Store the incoming data in the delay line; increment the delay-line pointer.

3. Clear the multiplier-accumulator.
4. Loop over all coefficients/delays; accumulate the values obtained by multiplying the coefficients by the delayed samples.
5. Round or truncate the result as the FIR output.

Alternatively, a "shuffle down" method is used in Texas Instruments' older fixed-point processors to implement circular buffers. The processor literally moves each sample delay values by one slot during each multiply-accumulate (via the "MACD" instruction).

Each DSP microprocessor manufacturer provides example FIR assembly code in its data books or its application handbooks, so be sure to look at those before you "reinvent the circular buffer".

4.4 How do I test my FIR implementation?

Here are a few methods:

- **Impulse Test:** A very simple and effective test is to put an impulse into it (which is just a "1" sample followed by at lest N - 1 zeroes.) You can also put in an "impulse train", with the "1" samples spaced at least N samples apart. If all the coefficients of the filter come out in the proper order, there is a good chance your filter is working correctly. (You might want to test with non-linear phase coefficients so you can see the order they come out.) We recommend you do this test whenever you write a new FIR filter routine.
- **Step Test:** Input N or more "1" samples. The output after N samples, should be the sum (DC gain) of the FIR filter.
- **Sine Test:** Input a sine wave at one or more frequencies and see if the output sine has the expected amplitude.
- **Swept FM Test:** *From Eric Jacobsen:* "My favorite test after an impulse train is to take two identical instances of the filter under test, use them as I and Q filters and put a complex FM linear sweep through them from DC to Fs/2. You can do an FFT on the result and see the complete frequency response of the filter, make sure the phase is nice and continuous everywhere, and match the response to what you'd expect from the coefficient set, the precision, etc."

4.5 What tricks are useful in implementing FIR filters?

FIR tricks center on two things 1) not calculating things that don't need to be calculated, and 2) "faking" circular buffers in software.

4.5.1 How do I skip needless calculations?

First, if your filter has zero-valued coefficients, you don't actually have to calculate those taps; you can leave them out. A common case of this is "half-band" filter, which have the property that every-other coefficient is zero.

Second, if your filter is "symmetric" (linear phase), you can "pre-add" the samples which will be multiplied by the same coefficient value, prior to doing the multiply. Since this technique essentially trades an add for a multiply, it isn't really useful in DSP microprocessors which can do a multiply in a single instruction cycle. However, it is useful in ASIC implementations (in which addition is usually much less expensive than multiplication); also, some newer DSP processors now offer special hardware and instructions to make use of this trick.

4.5.2 How do I fake circular buffers in software?

When hardware support for circular buffers isn't available, you have to "fake" them. Also, since ANSI C has no construct to describe circular buffers, most C compilers can't generate code to use them, even if the target processor has them.

You can always implement a circular buffer by duplicating the logic of a circular buffer in software (and many have), but the overhead can be prohibitive; the circular-fake might take several instructions to implement, compared to just a single instruction to do the multiply-accumulate operation. Therefore you need to fake it.

Here are several basic techniques to fake circular buffers:

1. **Split the calculation:** You can split any FIR calculation into its "pre-wrap" and "post-wrap" parts. By splitting the calculation into these two parts, you essentially can do the circular logic only once, rather than once per tap. (See fir_double_z in FirAlgs.c above.)
2. **Duplicate the delay line:** For a FIR with N taps, use a delay line of size 2N. Copy each sample to its proper location, as well as at location-plus-N. Therefore, the FIR calculation's MAC loop can be done on a flat buffer of N points, starting anywhere within the first set of N points. The second set of N delayed samples provides the "wrap around" comparable to a true circular buffer. (See fir_double_z in FirAlgs.c above.)
3. **Duplicate the coefficients:** This is similar to the above, except that the duplication occurs in terms of the coefficients, not the delay line. Compared to the previous method, this has a calculation advantage of not having to store each incoming sample twice, and it also has a memory advantage when the same coefficient set will be used on multiple delay lines. (See fir_double_h in FirAlgs.c above.)
4. **Use block processing:** In block processing, you use a delay line which is a multiple of the number of taps. You therefore only have to move the data *once* per block to implement the delay-line mechanism. When the block size becomes "large", the overhead of a moving the delay line once per block becomes negligible.

1.1 What are "FIR filters"?

FIR filters are one of two primary types of digital filters used in Digital Signal Processing (DSP) applications (the other type being IIR).

1.2 What does "FIR" mean?

"FIR" means "Finite Impulse Response".

1.3 Why is the impulse response "finite"?

The impulse response is "finite" because there is no feedback in the filter; if you put in an impulse (that is, a single "1" sample followed by many "0" samples), zeroes will eventually come out after the "1" sample has made its way in the delay line past all the coefficients.

1.4 How do I pronounce "FIR"?

Some people say the letters F-I-R; other people pronounce as if it were a type of tree. We prefer the tree. (The difference is whether you talk about *an* F-I-R filter or *a* FIR filter.)

1.5 What is the alternative to FIR filters?

DSP filters can also be "Infinite Impulse Response" (IIR). (See dspGuru's IIR FAQ.) IIR filters use feedback, so when you input an impulse the output theoretically rings indefinitely.

1.6 How do FIR filters compare to IIR filters?

Each has advantages and disadvantages. Overall, though, the advantages of FIR filters outweigh the disadvantages, so they are used much more than IIRs.

1.6.1 What are the *advantages* of FIR Filters (compared to IIR filters)?

Compared to IIR filters, FIR filters offer the following advantages:

- They can easily be designed to be "linear phase" (and usually are). Put simply, linear-phase filters delay the input signal, but don't distort its phase.
- They are simple to implement. On most DSP microprocessors, the FIR calculation can be done by looping a single instruction.
- They are suited to multi-rate applications. By multi-rate, we mean either "decimation" (reducing the sampling rate), "interpolation" (increasing the sampling rate), or both. Whether decimating or interpolating, the use of FIR filters allows some of the calculations to be omitted, thus providing an important computational efficiency. In contrast, if IIR filters are used,

each output must be individually calculated, even if it that output will discarded (so the feedback will be incorporated into the filter).
- They have desireable numeric properties. In practice, all DSP filters must be implemented using "finite-precision" arithmetic, that is, a limited number of bits. The use of finite-precision arithmetic in IIR filters can cause significant problems due to the use of feedback, but FIR filters have no feedback, so they can usually be implemented using fewer bits, and the designer has fewer practical problems to solve related to non-ideal arithmetic.
- They can be implemented using fractional arithmetic. Unlike IIR filters, it is always possible to implement a FIR filter using coefficients with magnitude of less than 1.0. (The overall gain of the FIR filter can be adjusted at its output, if desired.) This is an important considertaion when using fixed-point DSP's, because it makes the implementation much simpler.

1.6.2 What are the *disadvantages* of FIR Filters (compared to IIR filters)?

Compared to IIR filters, FIR filters sometimes have the disadvantage that they require more memory and/or calculation to achieve a given filter response characteristic. Also, certain responses are not practical to implement with FIR filters.

1.7 What terms are used in describing FIR filters?

- *Impulse Response* - The "impulse response" of a FIR filter is actually just the set of FIR coefficients. (If you put an "impluse" into a FIR filter which consists of a "1" sample followed by many "0" samples, the output of the filter will be the set of coefficients, as the 1 sample moves past each coefficient in turn to form the output.)
- *Tap* - A FIR "tap" is simply a coefficient/delay pair. The number of FIR taps, (often designated as "N") is an indication of 1) the amount of memory required to implement the filter, 2) the number of calculations required, and 3) the amount of "filtering" the filter can do; in effect, more taps means more stopband attenuation, less ripple, narrower filters, etc.)
- *Multiply-Accumulate (MAC)* - In a FIR context, a "MAC" is the operation of multiplying a coefficient by the corresponding delayed data sample and accumulating the result. FIRs usually require one MAC per tap. Most DSP microprocessors implement the MAC operation in a single instruction cycle.
- *Transition Band* - The band of frequencies between passband and stopband edges. The narrower the transition band, the more taps are required to implement the filter. (A "small" transition band results in a "sharp" filter.)
- *Delay Line* - The set of memory elements that implement the "Z^-1" delay elements of the FIR calculation.

- *Circular Buffer* - A special buffer which is "circular" because *incrementing* at the end causes it to wrap around to the beginning, or because *decrementing* from the beginning causes it to wrap around to the end. Circular buffers are often provided by DSP microprocessors to implement the "movement" of the samples through the FIR delay-line without having to literally move the data in memory. When a new sample is added to the buffer, it automatically replaces the oldest one.

1. IIR Basics

1.1 What are IIR filters? What does "IIR" mean?

IIR filters are one of two primary types of digital filters used in Digital Signal Processing (DSP) applications (the other type being FIR). "IIR" means "Infinite Impulse Response".

1.2 Why is the impulse response "infinite"?

The impulse response is "infinite" because there is feedback in the filter; if you put in an impulse (a single "1" sample followed by many "0" samples), an infinite number of non-zero values will come out (theoretically).

1.3 What is the alternative to IIR filters?

DSP filters can also be "Finite Impulse Response" (FIR). FIR filters do not use feedback, so for a FIR filter with *N* coefficients, the output always becomes zero after putting in *N* samples of an impulse response.

1.4 What are the advantages of IIR filters (compared to FIR filters)?

IIR filters can achieve a given filtering characteristic using less memory and calculations than a similar FIR filter.

1.5 What are the disadvantages of IIR filters (compared to FIR filters)?

- They are more susceptable to problems of finite-length arithmetic, such as noise generated by calculations, and limit cycles. (This is a direct consequence of feedback: when the output isn't computed perfectly and is fed back, the imperfection can compound.)
- They are harder (slower) to implement using fixed-point arithmetic.
- They don't offer the computational advantages of FIR filters for multirate (decimation and interpolation) applications.

2.1 Linear Phase

2.1.1 What is the association between FIR filters and "linear-phase"?

Most FIRs are linear-phase filters; when a linear-phase filter is desired, a FIR is usually used.

2.1.2 What is a *linear phase* filter?

"Linear Phase" refers to the condition where the phase response of the filter is a linear (straight-line) function of frequency (excluding phase wraps at 180 degrees). This results in the *delay* through the filter being the same at all frequencies. Therefore, the filter does not cause "phase distortion" or "delay distortion". The lack of phase/delay distortion can be a critical advantage of FIR filters over IIR and analog filters in certain systems, for example, in digital data modems.

2.1.3 What is the condition for linear phase?

FIR filters are usually designed to be linear-phase (but they don't have to be.) A FIR filter is linear-phase if (and only if) its coefficients are symmetrical around the center coefficient, that is, the first coefficient is the same as the last; the second is the same as the next-to-last, etc. (A linear-phase FIR filter having an odd number of coefficients will have a single coefficient in the center which has no mate.)

2.1.4 What is the delay of a linear-phase FIR?

The formula is simple: given a FIR filter which has N taps, the delay is: (N - 1) / (2 * Fs), where Fs is the sampling frequency. So, for example, a 21 tap linear-phase FIR filter operating at a 1 kHz rate has delay: (21 - 1) / (2 * 1 kHz)=10 milliseconds.

2.1.4 What is the alternative to linear phase?

Non-linear phase, of course. ;-) Actually, the most popular alternative is "minimum phase". Minimum-phase filters (which might better be called "minimum delay" filters) have less delay than linear-phase filters with the same amplitude response, at the cost of a non-linear phase characteristic, a.k.a. "phase distortion".

A lowpass FIR filter has its largest-magnitude coefficients in the center of the impulse response. In comparison, the largest-magnitude coefficients of a minimum-phase filter are nearer to the beginning . (See dspGuru's tutorial How To Design Minimum-Phase FIR Filters for more details.)

2.2 Frequency Response

2.2.1 What is the Z transform of a FIR filter?

For an N-tap FIR filter with coefficients h(k), whose output is described by:

$$y(n)=h(0)x(n) \; h(1)x(n-1) \; h(2)x(n-2) \; \ldots \; h(N-1)x(n-N-1),$$

the filter's Z transform is:

$H(z)=h(0)z-0\ h(1)z-1\ h(2)z-2\ ...\ h(N-1)z-(N-1)$, or

$$H(z)=\sum_{n=0}^{N-1}h(n)z^{-n}$$

2.2.2 What is the frequency response formula for a FIR filter?

The variable z in H(z) is a continuous complex variable, and we can describe it as: $z=r\cdot e^{j\omega}$, where r is a magnitude and ω is the angle of z. If we let r=1, then H(z) around the unit circle becomes the filter's frequency response H(jω). This means that substituting $e^{j\omega}$ for z in H(z) gives us an expression for the filter's frequency response H(ω), which is:

$H(j\omega)=h(0)e^{-j0\omega}\ h(1)e^{-j1\omega}\ h(2)e^{-j2\omega}\ ...\ h(N-1)e^{-j(N-1)\omega}$, or

Using Euler's identity, $e^{-ja}=\cos(a) - j\sin(a)$, we can write H(ω) in rectangular form as:

$H(j\omega)=h(0)[\cos(0\omega) - j\sin(0\omega)]\ h(1)[\cos(1\omega) - j\sin(1\omega)]\ ...\ h(N-1)[\cos((N-1)\omega) - j\sin((N-1)\omega)]$, or

$$H(j\omega)=\sum_{n=0}^{N-1}h(n)[\cos(n\omega) - j\sin(n\omega)]$$

2.2.3 Can I calculate the frequency response of a FIR using the Discrete Fourier Transform (DFT)?

Yes. For an N-tap FIR, you can get N evenly-spaced points of the frequency response by doing a DFT on the filter coefficients. However, to get the frequency response of the filter at any *arbitrary* frequency (that is, at frequencies *between* the DFT outputs), you will need to use the formula above.

2.2.4 What is the DC gain of a FIR filter?

Consider a DC (zero Hz) input signal consisting of samples which each have value 1.0. After the FIR's delay line had filled with the 1.0 samples, the output would be the sum of the coefficients. Therefore, the gain of a FIR filter at DC is simply the sum of the coefficients.

This intuitive result can be checked against the formula above. If we set ω to zero, the cosine term is always 1, and the sine term is always zero, so the frequency response becomes:

$$H(j\omega) = \sum_{n=0}^{N-1} h(n)$$

2.2.5 How do I scale the gain of a FIR filter?

Simply multiply all coefficients by the scale factor.

2.3 Numeric Properties

2.3.1 Are FIR filters inherently stable?

Yes. Since they have no feedback elements, any bounded input results in a bounded output.

2.3.2 What makes the numerical properties of FIR filters "good"?

Again, the key is the lack of feedback. The numeric errors that occur when implementing FIR filters in computer arithmetic occur separately with each calculation; the FIR doesn't "remember" its past numeric errors. In contrast, the feedback aspect of IIR filters can cause numeric errors to compound with each calculation, as numeric errors are fed back.

The practical impact of this is that FIRs can generally be implemented using fewer bits of precision than IIRs. For example, FIRs can usually be implemented with 16 bits, but IIRs generally require 32 bits, or even more.

2.4 Why are FIR filters generally preferred over IIR filters in multirate (decimating and interpolating) systems?

Because only a fraction of the calculations that would be required to implement a decimating or interpolating FIR in a literal way actually needs to be done.

Since FIR filters do not use feedback, only those outputs which are actually going to be used have to be calculated. Therefore, in the case of decimating FIRs (in which only 1 of N outputs will be used), the other N-1 outputs don't have to be calculated. Similarly, for interpolating filters (in which zeroes are inserted between the input samples to raise the sampling rate) you don't actually have to multiply the inserted zeroes with their corresponding FIR coefficients and sum the result; you just omit the multiplication-additions that are associated with the zeroes (because they don't change the result anyway.)

In contrast, since IIR filters use feedback, every input must be used, and every input must be calculated because all inputs and outputs contribute to the feedback in the filter.

2.5 What special types of FIR filters are there?

Aside from "regular" and "extra crispy" there are:

- *Boxcar* - Boxcar FIR filters are simply filters in which each coefficient is 1.0. Therefore, for an N-tap boxcar, the output is just the sum of the past N samples. Because boxcar FIRs can be implemented using only adders, they are of interest primarily in hardware implementations, where multipliers are expensive to implement.
- *Hilbert Transformer* - Hilbert Transformers shift the phase of a signal by 90 degrees. They are used primarily for creating the imaginary part of a complex signal, given its real part.
- *Differentiator* - Differentiators have an amplitude response which is a linear function of frequency. They are not very popular nowadays, but are sometimes used for FM demodulators.
- *Lth-Band* - Also called "Nyquist" filters, these filters are a special class of filters used primarily in multirate applications. Their key selling point is that one of every *L* coefficients is zero--a fact which can be exploited to reduce the number of multiply-accumulate operations required to implement the filter. (The famous "half-band" filter is actually an Lth-band filter, with L=2.)
- *Raised-Cosine* - This is a special kind of filter that is sometimes used for digital data applications. (The frequency response in the passband is a cosine shape which has been "raised" by a constant.) See dspGuru's Raised-Cosine FAQ for more information.
- *Lots of others.*

3.1 What are the *methods* of designing FIR filters?

The three most popular design methods are (in order):

- **Parks-McClellan:** The Parks-McClellan method (inaccurately called "Remez" by Matlab) is probably the most widely used FIR filter design method. It is an iteration algorithm that accepts filter specifications in terms of passband and stopband frequencies, passband ripple, and stopband attenuation. The fact that you can directly specify all the important filter parameters is what makes this method so popular. The PM method can design not only FIR "filters" but also FIR "differentiators" and FIR "Hilbert transformers".
- **Windowing:**. In the windowing method, an initial impulse response is derived by taking the Inverse Discrete Fourier Transform (IDFT) of the desired frequency response. Then, the impulse response is refined by applying a data window to it.
- **Direct Calculation:** The impulse responses of certain types of FIR filters (e.g. Raised Cosine and Windowed Sinc) can be calculated directly from formulas.

3.2 How do I actually *design* FIR filters?

With a FIR filter design program, of course. ;-) Although it's possible to design FIR filters using manual methods, it is a whole lot easier just to use a FIR filter design program.

3.3 What FIR filter design programs are available?

FIR filter design programs come in three broad categories:

- **Filter Design Applications:** See dspGuru's Digital Filter Design Software page for a list of filter design programs.
  Near and dear to us here at dspGuru is Iowegian's own ScopeFIR product. We believe ScopeFIR offers an excellent combination of professional features, smooth user interface, and affordable price. We sell it for just $199, with a generous 60 day trial period. Even if you already use Matlab, ScopeFIR's "point and shoot" capabilities can improve your FIR filter design productivity.
- **Math Programs:** Matlab and its Free Clones offer built-in FIR filter design functions.
- **Source code:** One of the best places on the net to find source code to design FIR filters is Charles Poynton's Filter Design Software page.

The Fast Fourier Transform is one of the most important topics in Digital Signal Processing but it is a confusing subject which frequently raises questions. Here, we answer Frequently Asked Questions (FAQ's) about the FFT.

1. FFT Basics

1.1 What is the FFT?

The Fast Fourier Transform (FFT) is simply a fast (computationally efficient) way to calculate the Discrete Fourier Transform (DFT).

1.2 How does the FFT work?

By making use of periodicities in the sines that are multiplied to do the transforms, the FFT greatly reduces the amount of calculation required. Here's a little overview.

Functionally, the FFT decomposes the set of data to be transformed into a series of smaller data sets to be transformed. Then, it decomposes *those* smaller sets into even *smaller* sets. At each stage of processing, the results of the previous stage are combined in special way. Finally, it calculates the DFT of each small data set. For example, an FFT of size 32 is broken into 2 FFT's of size 16, which are broken into 4 FFT's of size 8, which are broken into 8 FFT's of size 4, which are broken into 16 FFT's of size 2. Calculating a DFT of size 2 is trivial.

Here's a slightly more rigorous explanation: It turns out that it is possible to take the DFT of the first N/2 points and combine them in a special way with the DFT of the second N/2 points to produce a single N-point DFT. Each of these N/2-point DFTs can be calculated using smaller DFTs in the same way. One (radix-2) FFT begins, therefore, by calculating N/2 2-point DFTs. These are combined to form N/4 4-point DFTs. The next stage produces N/8 8-point DFTs, and so on, until a single N-point DFT is produced.

1.3 How efficient is the FFT?

The DFT takes N^2 operations for N points. Since at any stage the computation required to combine smaller DFTs into larger DFTs is proportional to N, and there are log2(N) stages (for radix 2), the total computation is proportional to N * log2(N). Therefore, the ratio between a DFT computation and an FFT computation for the same N is proportional to N / log2(n). In cases where N is small this ratio is not very significant, but when N becomes large, this ratio gets very large. (Every time you double N, the numerator doubles, but the denominator only increases by 1.)

1.6 Are FFT's limited to sizes that are powers of 2?

No. The most common and familiar FFT's are "radix 2". However, other radices are sometimes used, which are usually small numbers less than 10. For example, radix-4 is especially attractive because the "twiddle factors" are all 1, -1, j, or -j, which can be applied without any multiplications at all.

Also, "mixed radix" FFT's also can be done on "composite" sizes. In this case, you break a non-prime size down into its prime factors, and do an FFT whose stages use those factors. For example, an FFT of size 1000 might be done in six stages using radices of 2 and 5, since 1000 = 2 * 2 * 2 * 5 * 5 * 5. It might also be done in three stages using radix 10, since 1000 = 10 * 10 * 10.

1.7 Can FFT's be done on *prime* sizes?

Yes, although these are less efficient than single-radix or mixed-radix FFT's. It is almost always possible to avoid using prime sizes.

2. FFT Terminology

2.1 What is an FFT "radix"?

The "radix" is the size of an FFT decomposition. In the example above, the radix was 2. For single-radix FFT's, the transform size must be a power of the radix. In the example above, the size was 32, which is 2 to the 5th power.

2.2 What are "twiddle factors"?

"Twiddle factors" are the coefficients used to combine results from a previous stage to form inputs to the next stage.

2.3 What is an "in place" FFT?

An "in place" FFT is simply an FFT that is calculated entirely inside its original sample memory. In other words, calculating an "in place" FFT does not require additional buffer memory (as some FFT's do.)

2.4 What is "bit reversal"?

"Bit reversal" is just what it sounds like: reversing the bits in a binary word from left to write. Therefore the MSB's become LSB's and the LSB's become MSB's. But what does that have to do with FFT's? Well, the data ordering required by radix-2 FFT's turns out to be in "bit reversed" order, so bit-reversed indexes are used to combine FFT stages. It is possible (but *slow*) to calculate these bit-reversed indices in software; however, bit reversals are trivial when implemented in hardware. Therefore, almost all DSP processors include a hardware bit-reversal indexing capability (which is one of the things that distinguishes them from other microprocessors.)

2.5 What is "decimation in time" versus "decimation in frequency"?

FFT's can be decomposed using DFT's of even and odd points, which is called a Decimation-In-Time (DIT) FFT, or they can be decomposed using a first-half/second-half approach, which is called a "Decimation-In-Frequency" (DIF) FFT. Generally, the user does not need to worry which type is being used.

3. FFT Implementation

3.1 How do I implement an FFT?

Except as a learning exercise, you generally will never have to. Many good FFT implementations are available in C, Fortran and other languages, and microprocessor manufacturers generally provide free optimized FFT implementations in their processors' assembly code, Therefore, it is not so important to understand *how* the FFT really works, as it is to understand how to *use* it.

3.2 I'm not convinced. I want to really learn how the FFT works.

(Gosh you're difficult!) Well, virtually every DSP book on the planet covers the FFT in detail. Also, the following *online* books explain the FFT: Numerical Recipies, Scientists and Engineer's Guide to DSP.

3.3 OK, I read that stuff. Let's cut to the chase. Where do I get an FFT implementation?

1. If you want an assembly language implementation, check out the web site of the manufacturer of your chosen DSP microprocessor. They generally provide highly optimized assembly implementations in their user's guides and application manuals, and also as part of the library of their C compilers.
2. Here are a couple of the best C implemenations:
    - FFTW - The "Fastest Fourier Transform in the West".
    - MixFFT - This Mixed-radix FFT implemenation by Jens Joergen Nielsen is fast, flexible, and easy to use.
3. There are several great FFT hotlists on the net. One of the best is Jörgs useful and ugly FXT page, and FFTW also has a very good FFT hotlist. (You can find other FFT hotlists on our DSP hotlists page.)

1
CHAPTER

1 The Breadth and Depth of DSP

Digital Signal Processing is one of the most powerful technologies that will shape science and engineering in the twenty-first century. Revolutionary changes have already been made in a *broad*
range of fields: communications, medical imaging, radar & sonar, high fidelity music reproduction, and oil prospecting, to name just a few. Each of these areas has developed a *deep* DSP technology, with its own algorithms, mathematics, and specialized techniques. This combination of breath and depth makes it impossible for any one individual to master all of the DSP technology that has been developed. DSP education involves two tasks: learning general concepts that apply to the field as a whole, and learning specialized techniques for your particular area of interest. This chapter starts our journey into the world of Digital Signal Processing by describing the dramatic effect that DSP has made in several diverse fields. The revolution has begun.

The Roots of DSP

Digital Signal Processing is distinguished from other areas in computer science by the unique type of data it uses: *signals*. In most cases, these signals originate as sensory data from the real world: seismic vibrations, visual images, sound waves, etc. DSP is the mathematics, the algorithms, and the techniques used to manipulate these signals after they have been converted into a digital form. This includes a wide variety of goals, such as: enhancement of visual images, recognition and generation of speech, compression of data for storage and transmission, etc. Suppose we attach an analog-to-digital converter to a computer and use it to acquire a chunk of real world data. DSP answers the question: *What next*?

The roots of DSP are in the 1960s and 1970s when digital computers first became available. Computers were expensive during this era, and DSP was limited to only a few critical applications. Pioneering efforts were made in four key areas: *radar & sonar*, where national security was at risk; *oil exploration*,

where large amounts of money could be made; *space exploration*, where the

FIGURE 1-1
DSP has revolutionized many areas in science and engineering. A few of these diverse applications are shown here.

data are irreplaceable; and *medical imaging*, where lives could be saved.

The personal computer revolution of the 1980s and 1990s caused DSP to explode with new applications. Rather than being motivated by military and government needs, DSP was suddenly driven by the commercial marketplace. Anyone who thought they could make money in the rapidly expanding field was suddenly a DSP vendor. DSP reached the public in such products as: mobile telephones, compact disc players, and electronic voice mail. Figure 1-1 illustrates a few of these varied applications.

This technological revolution occurred from the top-down. In the early 1980s, DSP was taught as a *graduate* level course in electrical engineering. A decade later, DSP had become a standard part of the *undergraduate* curriculum. Today, DSP is a *basic skill* needed by scientists and engineers

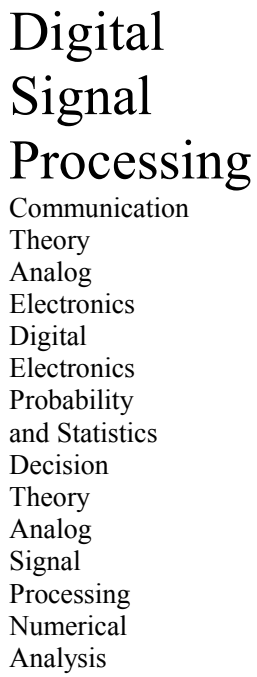


FIGURE 1-2
Digital Signal Processing has fuzzy and overlapping borders with many other areas of science, engineering and mathematics.

in many fields. As an analogy, DSP can be compared to a previous technological revolution: *electronics*. While still the realm of electrical engineering, nearly every scientist and engineer has some background in basic circuit design. Without it, they would be lost in the technological world. DSP has the same future.

This recent history is more than a curiosity; it has a tremendous impact on *your* ability to learn and use DSP. Suppose you encounter a DSP problem, and turn to textbooks or other publications to find a solution. What you will typically find is page after page of equations, obscure mathematical symbols, and unfamiliar terminology. It's a nightmare! Much of the DSP literature is baffling even to those experienced in the field. It's not that there is anything wrong with this material, it is just intended for a very specialized audience. State-of-the-art researchers need this kind of detailed mathematics to understand the theoretical implications of the work.

A basic premise of this book is that most practical DSP techniques can be learned and used without the traditional barriers of detailed mathematics and theory. *The Scientist and Engineer's Guide to Digital Signal Processing* is written for those who want to use DSP as a *tool*, not a new *career*.

The remainder of this chapter illustrates areas where DSP has produced revolutionary changes. As you go through each application, notice that DSP is very *interdisciplinary*, relying on the technical work in many adjacent fields. As Fig. 1-2 suggests, the borders between DSP and other technical disciplines are not sharp and well defined, but rather fuzzy and overlapping. If you want to specialize in DSP, these are the allied areas you will also need to study.

Telecommunications

Telecommunications is about transferring information from one location to another. This includes many forms of information: telephone conversations,

television signals, computer files, and other types of data. To transfer the information, you need a *channel* between the two locations. This may be a wire pair, radio signal, optical fiber, etc. Telecommunications companies receive *payment* for transferring their customer's information, while they must *pay* to establish and maintain the channel. The financial bottom line is simple: the more information they can pass through a single channel, the more money they make. DSP has revolutionized the telecommunications industry in many areas: signaling tone generation and detection, frequency band shifting, filtering to remove power line hum, etc. Three specific examples from the telephone network will be discussed here: multiplexing, compression, and echo control.

Multiplexing

There are approximately *one billion* telephones in the world. At the press of a few buttons, switching networks allow any one of these to be connected to any other in only a few seconds. The immensity of this task is mind boggling! Until the 1960s, a connection between two telephones required passing the analog voice signals through mechanical switches and amplifiers. One connection required one pair of wires. In comparison, DSP converts audio signals into a stream of serial digital data. Since bits can be easily intertwined and later separated, many telephone conversations can be transmitted on a single channel. For example, a telephone standard known as the *T-carrier system* can simultaneously transmit 24 voice signals. Each voice signal is sampled 8000 times per second using an 8 bit companded (logarithmic compressed) analog-to-digital conversion. This results in each voice signal being represented as 64,000 bits/sec, and all 24 channels being contained in 1.544 megabits/sec. This signal can be transmitted about 6000 feet using ordinary telephone lines of 22 gauge copper wire, a typical interconnection distance. The financial advantage of digital transmission is enormous. Wire and analog switches are expensive; digital logic gates are cheap.

Compression

When a voice signal is digitized at 8000 samples/sec, most of the digital information is *redundant*. That is, the information carried by any one sample is largely duplicated by the neighboring samples. Dozens of DSP algorithms have been developed to convert digitized voice signals into data streams that require fewer bits/sec. These are called **data compression** algorithms. Matching **uncompression** algorithms are used to restore the signal to its original form. These algorithms vary in the amount of compression achieved and the resulting sound quality. In general, reducing the data rate from 64 kilobits/sec to 32 kilobits/sec results in no loss of sound quality. When compressed to a data rate of 8 kilobits/sec, the sound is noticeably affected, but still usable for long distance telephone networks. The highest achievable compression is about 2 kilobits/sec, resulting in

sound that is highly distorted, but usable for some applications such as military and undersea communications.

Echo control

Echoes are a serious problem in long distance telephone connections. When you speak into a telephone, a signal representing your voice travels to the connecting receiver, where a portion of it returns as an echo. If the connection is within a few hundred miles, the elapsed time for receiving the echo is only a few milliseconds. The human ear is accustomed to hearing echoes with these small time delays, and the connection sounds quite

normal. As the distance becomes larger, the echo becomes increasingly noticeable and irritating. The delay can be several hundred milliseconds for intercontinental communications, and is particularly objectionable. Digital Signal Processing attacks this type of problem by measuring the returned signal and generating an appropriate *antisignal* to cancel the offending echo. This same technique allows speakerphone users to hear and speak at the same time without fighting audio feedback (squealing). It can also be used to reduce environmental noise by canceling it with digitally generated *antinoise*.

Audio Processing

The two principal human senses are vision and hearing. Correspondingly, much of DSP is related to image and audio processing. People listen to both *music* and *speech*. DSP has made revolutionary changes in both these areas.

Music

The path leading from the musician's microphone to the audiophile's speaker is remarkably long. Digital data representation is important to prevent the degradation commonly associated with analog storage and manipulation. This is very familiar to anyone who has compared the musical quality of cassette tapes with compact disks. In a typical scenario, a musical piece is recorded in a sound studio on multiple channels or tracks. In some cases, this even involves recording individual instruments and singers separately. This is done to give the sound engineer greater flexibility in creating the final product. The complex process of combining the individual tracks into a final product is called *mix down*. DSP can provide several important functions during mix down, including: filtering, signal addition and subtraction, signal editing, etc. One of the most interesting DSP applications in music preparation is *artificial reverberation*. If the individual channels are simply added together, the resulting piece sounds frail and diluted, much as if the musicians were playing outdoors. This is because listeners are greatly influenced by the echo or reverberation content of the music, which is usually minimized in the sound studio. DSP allows artificial echoes and reverberation to be added during mix down to simulate various ideal listening environments. Echoes with delays of a few hundred milliseconds give the impression of cathedral like

locations. Adding echoes with delays of 10-20 milliseconds provide the perception of more modest size listening rooms.

Speech generation

Speech generation and recognition are used to communicate between humans and machines. Rather than using your hands and eyes, you use your mouth and ears. This is very convenient when your hands and eyes should be doing something else, such as: driving a car, performing surgery, or (unfortunately) firing your weapons at the enemy. Two approaches are used for computer generated speech: *digital recording* and *vocal tract simulation.* In digital recording, the voice of a human speaker is digitized and stored, usually in a compressed form. During playback, the stored data are uncompressed and converted back into an analog signal. An entire hour of recorded speech requires only about three megabytes of storage, well within the capabilities of even small computer systems. This is the most common method of digital speech generation used today.

Vocal tract simulators are more complicated, trying to mimic the physical mechanisms by which humans create speech. The human vocal tract is an acoustic cavity with resonant frequencies determined by the size and shape of

the chambers. Sound originates in the vocal tract in one of two basic ways, called *voiced* and *fricative* sounds. With voiced sounds, vocal cord vibration produces near periodic pulses of air into the vocal cavities. In comparison, fricative sounds originate from the noisy air turbulence at narrow constrictions, such as the teeth and lips. Vocal tract simulators operate by generating digital signals that resemble these two types of excitation. The characteristics of the resonate chamber are simulated by passing the excitation signal through a digital filter with similar resonances. This approach was used in one of the very early DSP success stories, the *Speak & Spell*, a widely sold electronic learning aid for children.

Speech recognition

The automated recognition of human speech is immensely more difficult than speech generation. Speech recognition is a classic example of things that the human brain does well, but digital computers do poorly. Digital computers can store and recall vast amounts of data, perform mathematical calculations at blazing speeds, and do repetitive tasks without becoming bored or inefficient. Unfortunately, present day computers perform very poorly when faced with raw sensory data. Teaching a computer to send you a monthly electric bill is easy. Teaching the same computer to understand your voice is a major undertaking.

Digital Signal Processing generally approaches the problem of voice recognition in two steps: *feature extraction* followed by *feature matching*. Each word in the incoming audio signal is isolated and then analyzed to identify the type of excitation and resonate frequencies. These parameters are then compared with previous examples of spoken words to identify the closest match. Often, these systems are limited to only a few hundred words; can only accept speech with distinct pauses between words; and must be retrained for each individual speaker. While this is adequate for many commercial

applications, these limitations are humbling when compared to the abilities of human hearing. There is a great deal of work to be done in this area, with tremendous financial rewards for those that produce successful commercial products.

Echo Location

A common method of obtaining information about a remote object is to bounce a *wave* off of it. For example, radar operates by transmitting pulses of radio waves, and examining the received signal for echoes from aircraft. In sonar, sound waves are transmitted through the water to detect submarines and other submerged objects. Geophysicists have long probed the earth by setting off explosions and listening for the echoes from deeply buried layers of rock. While these applications have a common thread, each has its own specific problems and needs. Digital Signal Processing has produced revolutionary changes in all three areas.

Radar

Radar is an acronym for *RAdio Detection And Ranging*. In the simplest radar system, a radio transmitter produces a pulse of radio frequency energy a few microseconds long. This pulse is fed into a highly directional antenna, where the resulting radio wave propagates away at the speed of light. Aircraft in the path of this wave will reflect a small portion of the energy back toward a receiving antenna, situated near the transmission site. The distance to the object is calculated from the elapsed time between the transmitted pulse and the received echo. The direction to the object is found more simply; you know *where* you pointed the directional antenna

when the echo was received.
The operating range of a radar system is determined by two parameters: how much energy is in the initial pulse, and the noise level of the radio receiver. Unfortunately, increasing the energy in the pulse usually requires making the pulse *longer*. In turn, the longer pulse reduces the accuracy and precision of the elapsed time measurement. This results in a conflict between two important parameters: the ability to detect objects at long range, and the ability to accurately determine an object's distance.
DSP has revolutionized radar in three areas, all of which relate to this basic problem. First, DSP can *compress* the pulse after it is received, providing better distance determination without reducing the operating range. Second, DSP can filter the received signal to decrease the noise. This increases the range, without degrading the distance determination. Third, DSP enables the rapid selection and generation of different pulse shapes and lengths. Among other things, this allows the pulse to be optimized for a particular detection problem. Now the impressive part: much of this is done at a sampling rate comparable to the radio frequency used, as high as several hundred megahertz! When it comes to radar, DSP is as much about high-speed hardware design as it is about algorithms.

Sonar
Sonar is an acronym for *SOund NAvigation and Ranging*. It is divided into two categories, *active* and *passive*. In active sonar, sound pulses between 2 kHz and 40 kHz are transmitted into the water, and the resulting echoes detected and analyzed. Uses of active sonar include: detection & localization of undersea bodies, navigation, communication, and mapping the sea floor. A maximum operating range of 10 to 100 kilometers is typical. In comparison, passive sonar simply *listens* to underwater sounds, which includes: natural turbulence, marine life, and mechanical sounds from submarines and surface vessels. Since passive sonar emits no energy, it is ideal for covert operations. You want to detect *the other guy*, without him detecting *you*. The most important application of passive sonar is in military surveillance systems that detect and track submarines. Passive sonar typically uses lower frequencies than active sonar because they propagate through the water with less absorption. Detection ranges can be thousands of kilometers.
DSP has revolutionized sonar in many of the same areas as radar: pulse generation, pulse compression, and filtering of detected signals. In one view, sonar is *simpler* than radar because of the lower frequencies involved. In another view, sonar is more *difficult* than radar because the environment is much less uniform and stable. Sonar systems usually employ extensive arrays of transmitting and receiving elements, rather than just a single channel. By properly controlling and mixing the signals in these many elements, the sonar system can steer the emitted pulse to the desired location and determine the direction that echoes are received from. To handle these multiple channels, sonar systems require the same massive DSP computing power as radar.
Reflection seismology
As early as the 1920s, geophysicists discovered that the structure of the earth's crust could be probed with sound. Prospectors could set off an explosion and record the echoes from boundary layers more than ten kilometers below the surface. These echo seismograms were interpreted by the raw eye to map the subsurface structure. The reflection seismic method rapidly became the

primary method for locating petroleum and mineral deposits, and remains so today.
In the ideal case, a sound pulse sent into the ground produces a single echo for each boundary layer the pulse passes through. Unfortunately, the situation is not usually this simple. Each echo returning to the surface must pass through all the other boundary layers above where it originated. This can result in the echo bouncing between layers, giving rise to *echoes of echoes* being detected at the surface. These secondary echoes can make the detected signal very complicated and difficult to interpret. Digital Signal Processing has been widely used since the 1960s to isolate the primary from the secondary echoes in reflection seismograms. How did the early geophysicists manage without DSP? The answer is simple: they looked in *easy* places, where multiple reflections were minimized. DSP allows oil to be found in *difficult* locations, such as under the ocean.

Image Processing

Images are signals with special characteristics. First, they are a measure of a parameter over *space* (distance), while most signals are a measure of a parameter over *time*. Second, they contain a great deal of information. For example, more than 10 megabytes can be required to store one second of television video. This is more than a thousand times greater than for a similar length voice signal. Third, the final judge of quality is often a subjective human evaluation, rather than an objective criterion. These special characteristics have made image processing a distinct subgroup within DSP.

Medical

In 1895, Wilhelm Conrad Röntgen discovered that x-rays could pass through substantial amounts of matter. Medicine was revolutionized by the ability to look inside the living human body. Medical x-ray systems spread throughout the world in only a few years. In spite of its obvious success, medical x-ray imaging was limited by four problems until DSP and related techniques came along in the 1970s. First, overlapping structures in the body can hide behind each other. For example, portions of the heart might not be visible behind the ribs. Second, it is not always possible to distinguish between similar tissues. For example, it may be able to separate bone from soft tissue, but not distinguish a tumor from the liver. Third, x-ray images show *anatomy*, the body's structure, and not *physiology*, the body's operation. The x-ray image of a living person looks exactly like the x-ray image of a dead one! Fourth, x-ray exposure can cause cancer, requiring it to be used sparingly and only with proper justification.
The problem of overlapping structures was solved in 1971 with the introduction of the first **computed tomography** scanner (formerly called computed axial tomography, or *CAT* scanner). Computed tomography (CT) is a classic example of Digital Signal Processing. X-rays from many directions are passed through the section of the patient's body being examined. Instead of simply forming images with the detected x-rays, the signals are converted into digital data and stored in a computer. The information is then used to *calculate* images that appear to be *slices through the body*. These images show much greater detail than conventional techniques, allowing significantly better diagnosis and treatment. The impact of CT was nearly as large as the original introduction of x-ray imaging itself. Within only a few years, every major hospital in the world had access to a CT scanner. In 1979, two of CT's principle contributors, Godfrey N. Hounsfield and Allan M. Cormack, shared the Nobel Prize in Medicine. *That's good DSP!*

The last three x-ray problems have been solved by using penetrating energy other than x-rays, such as radio and sound waves. DSP plays a key role in all these techniques. For example, Magnetic Resonance Imaging (MRI) uses magnetic fields in conjunction with radio waves to probe the interior of the human body. Properly adjusting the strength and frequency of the fields cause the atomic nuclei in a localized region of the body to resonate between quantum energy states. This resonance results in the emission of a secondary radio

wave, detected with an antenna placed near the body. The strength and other characteristics of this detected signal provide information about the localized region in resonance. Adjustment of the magnetic field allows the resonance region to be scanned throughout the body, mapping the internal structure. This information is usually presented as images, just as in computed tomography. Besides providing excellent discrimination between different types of soft tissue, MRI can provide information about physiology, such as blood flow through arteries. MRI relies totally on Digital Signal Processing techniques, and could not be implemented without them.

Space

Sometimes, you just have to make the most out of a bad picture. This is frequently the case with images taken from unmanned satellites and space exploration vehicles. No one is going to send a repairman to Mars just to tweak the knobs on a camera! DSP can improve the quality of images taken under extremely unfavorable conditions in several ways: brightness and contrast adjustment, edge detection, noise reduction, focus adjustment, motion blur reduction, etc. Images that have spatial distortion, such as encountered when a flat image is taken of a spherical planet, can also be *warped* into a correct representation. Many individual images can also be combined into a single database, allowing the information to be displayed in unique ways. For example, a video sequence simulating an aerial flight over the surface of a distant planet.

Commercial Imaging Products

The large information content in images is a problem for systems sold in mass quantity to the general public. Commercial systems must be *cheap*, and this doesn't mesh well with large memories and high data transfer rates. One answer to this dilemma is *image compression*. Just as with voice signals, images contain a tremendous amount of redundant information, and can be run through algorithms that reduce the number of bits needed to represent them. Television and other moving pictures are especially suitable for compression, since most of the image remain the same from frame-to-frame. Commercial imaging products that take advantage of this technology include: video telephones, computer programs that display moving pictures, and digital television.

CHAPTER

2 Statistics, Probability and Noise

Statistics and probability are used in Digital Signal Processing to characterize signals and the processes that generate them. For example, a primary use of DSP is to reduce interference, noise, and other undesirable components in acquired data. These may be an inherent part of the signal being measured, arise from imperfections in the data acquisition system, or be introduced as an unavoidable byproduct of some DSP operation. Statistics and probability allow these disruptive features to be measured and classified, the first step in developing strategies to remove the

offending components. This chapter introduces the most important concepts in statistics and probability, with emphasis on how they apply to acquired signals.

Signal and Graph Terminology

A *signal* is a description of how one parameter is related to another parameter. For example, the most common type of signal in analog electronics is a *voltage* that varies with *time*. Since both parameters can assume a continuous range of values, we will call this a **continuous signal**. In comparison, passing this signal through an analog-to-digital converter forces each of the two parameters to be *quantized*. For instance, imagine the conversion being done with 12 bits at a sampling rate of 1000 samples per second. The voltage is curtailed to 4096 (2_{12}) possible binary levels, and the time is only defined at one millisecond increments. Signals formed from parameters that are quantized in this manner are said to be **discrete signals** or **digitized signals**. For the most part, continuous signals exist in nature, while discrete signals exist inside computers (although you can find exceptions to both cases). It is also possible to have signals where one parameter is continuous and the other is discrete. Since these mixed signals are quite uncommon, they do not have special names given to them, and the nature of the two parameters must be explicitly stated.

Figure 2-1 shows two discrete signals, such as might be acquired with a digital data acquisition system. The **vertical axis** may represent voltage, light

intensity, sound pressure, or an infinite number of other parameters. Since we don't know what it represents in this particular case, we will give it the generic label: **amplitude**. This parameter is also called several other names: the **yaxis**, the **dependent variable**, the **range**, and the **ordinate**.

The **horizontal axis** represents the other parameter of the signal, going by such names as: the **x-axis**, the **independent variable**, the **domain**, and the **abscissa**. *Time* is the most common parameter to appear on the horizontal axis of acquired signals; however, other parameters are used in specific applications. For example, a geophysicist might acquire measurements of rock density at equally spaced *distances* along the surface of the earth. To keep things general, we will simply label the horizontal axis: **sample number**. If this were a continuous signal, another label would have to be used, such as: *time*, *distance*, *x*, etc.

The two parameters that form a signal are generally not interchangeable. The parameter on the y-axis (the dependent variable) is said to be a **function** of the parameter on the x-axis (the independent variable). In other words, the independent variable describes *how* or *when* each sample is taken, while the dependent variable is the actual measurement. Given a specific value on the x-axis, we can always find the corresponding value on the y-axis, but usually not the other way around.

Pay particular attention to the word: *domain*, a very widely used term in DSP. For instance, a signal that uses time as the independent variable (i.e., the parameter on the horizontal axis), is said to be in the **time domain**. Another common signal in DSP uses frequency as the independent variable, resulting in the term, **frequency domain**. Likewise, signals that use distance as the independent parameter are said to be in the **spatial domain** (distance is a measure of space). The type of parameter on the horizontal axis *is* the domain of the signal; it's that simple. What if the x-axis is labeled with something very generic, such as *sample number*? Authors commonly refer to these signals as being in the *time* domain. This is because sampling at equal intervals of time is the most common way of obtaining signals, and they don't have anything more specific to call it.

Although the signals in Fig. 2-1 are discrete, they are displayed in this figure as continuous lines. This is because there are too many samples to be distinguishable if they were displayed as individual markers. In graphs that portray shorter signals, say less than 100 samples, the individual markers are usually shown. Continuous lines may or may not be drawn to connect the markers, depending on how the author wants you to view the data. For instance, a continuous line could imply what is happening *between* samples, or simply be an aid to help the reader's eye follow a trend in noisy data. The point is, examine the labeling of the horizontal axis to find if you are working with a discrete or continuous signal. Don't rely on an illustrator's ability to draw dots.

The variable, ***N***, is widely used in DSP to represent the total number of samples in a signal. For example, for the signals in Fig. 2-1. To *N* ' 512

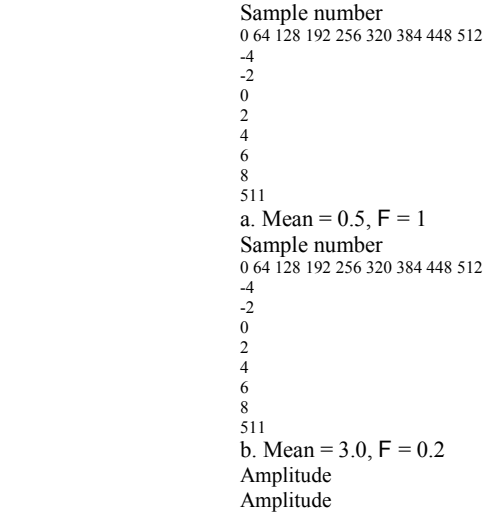


FIGURE 2-1
Examples of two digitized signals with different means and standard deviations.

EQUATION 2-1
Calculation of a signal's mean. The signal is contained in x_0 through x_{N-1}, i is an index that runs through these values, and μ is the mean.

$$\mu \text{ ' } \frac{1}{N} \sum_{i \text{ ' } 0}^{N \& 1} x_i$$

keep the data organized, each sample is assigned a **sample number** or **index**. These are the numbers that appear along the horizontal axis. Two notations for assigning sample numbers are commonly used. In the first notation, the sample indexes run from 1 to *N* (e.g., 1 to 512). In the second notation, the sample indexes run from 0 to (e.g., 0 to 511). *N*& 1 Mathematicians often use the first method (1 to *N*), while those in DSP commonly uses the second (0 to). In this book, we will use the second *N*& 1 notation. Don't dismiss this as a trivial problem. It *will* confuse you sometime during your career. Look out for it!

Mean and Standard Deviation

The **mean**, indicated by **µ** (a lower case Greek *mu*), is the statistician's jargon for the average value of a signal. It is found just as you would expect: add all of the samples together, and divide by *N*. It looks like this in mathematical

form:
In words, sum the values in the signal, , by letting the index, *i*, run from 0 x_i to . Then finish the calculation by dividing the sum by *N*. This is *N*& 1 identical to the equation: . If you are not already μ ' (x_0
% x_1
% x_2
% þ% $x_{N\&1}$) /*N*
familiar with E (upper case Greek *sigma*) being used to indicate *summation*, study these equations carefully, and compare them with the computer program in Table 2-1. Summations of this type are abundant in DSP, and you need to understand this notation fully.

EQUATION 2-2
Calculation of the standard deviation of a signal. The signal is stored in , μ is the x_i mean found from Eq. 2-1, *N* is the number of samples, and is the standard deviation. *s*

$$F_2 ' \frac{1}{N\&1} \sum_{i ' 0}^{N\&1} (x_i \& \mu)_2$$

In electronics, the *mean* is commonly called the **DC** (direct current) value. Likewise, **AC** (alternating current) refers to how the signal fluctuates around the mean value. If the signal is a simple repetitive waveform, such as a sine or square wave, its excursions can be described by its peak-to-peak amplitude. Unfortunately, most acquired signals do not show a well defined peak-to-peak value, but have a random nature, such as the signals in Fig. 2-1. A more generalized method must be used in these cases, called the **standard deviation**, denoted by F (a lower case Greek *sigma*).
As a starting point, the expression, , describes how far the sample *x_i & μ* i_{th} *deviates* (differs) from the mean. The *average deviation* of a signal is found by summing the deviations of all the individual samples, and then dividing by the number of samples, *N*. Notice that we take the absolute value of each deviation before the summation; otherwise the positive and negative terms would average to zero. The average deviation provides a single number representing the typical distance that the samples are from the mean. While convenient and straightforward, the average deviation is almost never used in statistics. This is because it doesn't fit well with the physics of how signals operate. In most cases, the important parameter is not the *deviation* from the mean, but the *power* represented by the deviation from the mean. For example, when random noise signals combine in an electronic circuit, the resultant noise is equal to the combined *power* of the individual signals, not their combined *amplitude*.
The *standard deviation* is similar to the *average deviation*, except the averaging is done with power instead of amplitude. This is achieved by squaring each of the deviations before taking the average (remember, power % voltage2). To finish, the *square root* is taken to compensate for the initial squaring. In equation form, the standard deviation is calculated:
In the alternative notation: . F' (x_0
& μ)2% (x_1

& μ)2% þ% (*xN*&1
& μ)2 / (*N*&1)

Notice that the average is carried out by dividing by instead of *N.* This *N*& 1 is a subtle feature of the equation that will be discussed in the next section. The term, F2, occurs frequently in statistics and is given the name **variance.** The standard deviation is a measure of how far the signal fluctuates from the mean. The variance represents the power of this fluctuation. Another term you should become familiar with is the **rms (root-mean-square)** value, frequently used in electronics. By definition, the standard deviation only measures the AC portion of a signal, while the rms value measures both the AC and DC components. If a signal has no DC component, its rms value is identical to its standard deviation. Figure 2-2 shows the relationship between the standard deviation and the peak-to-peak value of several common waveforms.

Vpp
F
Vpp
F
Vpp
F
Vpp
F

FIGURE 2-2
Ratio of the peak-to-peak amplitude to the standard deviation for several common waveforms. For the square wave, this ratio is 2; for the triangle wave it is ; for the sine wave it is . While random 12 ' 3.46 2 2 ' 2.83 noise has no *exact* peak-to-peak value, it is *approximately* 6 to 8 times the standard deviation.
a. Square Wave, Vpp = 2F
c. Sine wave, Vpp = 2 2F d. Random noise, Vpp . 6-8 F
b. Triangle wave, Vpp = 12 F

```
100 CALCULATION OF THE MEAN AND STANDARD DEVIATION
110 '
120 DIM X[511] 'The signal is held in X[0] to X[511]
130 N% = 512 'N% is the number of points in the signal
140 '
150 GOSUB XXXX 'Mythical subroutine that loads the signal into X[ ]
160 '
170 MEAN = 0 'Find the mean via Eq. 2-1
180 FOR I% = 0 TO N%-1
190 MEAN = MEAN + X[I%]
200 NEXT I%
210 MEAN = MEAN/N%
220 '
230 VARIANCE = 0 'Find the standard deviation via Eq. 2-2
240 FOR I% = 0 TO N%-1
250 VARIANCE = VARIANCE + ( X[I%] - MEAN )^2
260 NEXT I%
270 VARIANCE = VARIANCE/(N%-1)
280 SD = SQR(VARIANCE)
290 '
300 PRINT MEAN SD 'Print the calculated mean and standard deviation
310 '
320 END
```

TABLE 2-1

Table 2-1 lists a computer routine for calculating the mean and standard deviation using Eqs. 2-1 and 2-2. The programs in this book are intended to convey *algorithms* in the most straightforward way; all other factors are treated as secondary. Good programming techniques are disregarded if it makes the program logic more clear. For instance: a simplified version of

BASIC is used, line numbers are included, the only control structure allowed is the FOR-NEXT loop, there are no I/O statements, etc. Think of these programs as an alternative way of understanding the equations used

$$F_2 \leftarrow 1$$

$$N-1$$

$$\sum_{i=0}^{N-1} x_i^2$$

$$-\frac{1}{N}\sum_{i=0}^{N-1} x_i$$

2 EQUATION 2-3

Calculation of the standard deviation using running statistics. This equation provides the same result as Eq. 2-2, but with less roundoff noise and greater computational efficiency. The signal is expressed in terms of three accumulated parameters: *N*, the total number of samples; *sum*, the sum of these samples; and *sum of squares*, the sum of the squares of the samples. The mean and standard deviation are then calculated from these three accumulated parameters.

or using a simpler notation,

$$F_2 \leftarrow 1$$

$$N-1$$

$$\textit{sum of squares} - \frac{\textit{sum}^2}{N}$$

in DSP. If you can't grasp one, maybe the other will help. In BASIC, the % character at the end of a variable name indicates it is an integer. All other variables are floating point. Chapter 4 discusses these variable types in detail.

This method of calculating the mean and standard deviation is adequate for many applications; however, it has two limitations. First, if the mean is much larger than the standard deviation, Eq. 2-2 involves subtracting two numbers that are very close in value. This can result in excessive round-off error in the calculations, a topic discussed in more detail in Chapter 4. Second, it is often desirable to recalculate the mean and standard deviation as new samples are acquired and added to the signal. We will call this type of calculation: **running statistics.** While the method of Eqs. 2-1 and 2-2 can be used for running statistics, it requires that *all* of the samples be involved in each new calculation. This is a very inefficient use of computational power and memory.

A solution to these problems can be found by manipulating Eqs. 2-1 and 2-2 to provide another equation for calculating the standard deviation:

While moving through the signal, a running tally is kept of three parameters: (1) the number of samples already processed, (2) the sum of these samples, and (3) the sum of the squares of the samples (that is, square the value of each sample and add the result to the accumulated value). After any number of samples have been processed, the mean and standard deviation can be efficiently calculated using only the current value of the three parameters.

Table 2-2 shows a program that reports the mean and standard deviation in this manner as each new sample is taken into account. This is the method used in hand calculators to find the statistics of a sequence of numbers. Every time you enter a number and press the E (summation) key, the three parameters are updated. The mean and standard deviation can then be found whenever desired, without having to recalculate the entire sequence.


```
100 'MEAN AND STANDARD DEVIATION USING RUNNING STATISTICS
110 '
120 DIM X[511] 'The signal is held in X[0] to X[511]
130 '
140 GOSUB XXXX 'Mythical subroutine that loads the signal into X[ ]
150 '
160 N% = 0 'Zero the three running parameters
170 SUM = 0
180 SUMSQUARES = 0
190 '
200 FOR I% = 0 TO 511 'Loop through each sample in the signal
210 '
220 N% = N%+1 'Update the three parameters
230 SUM = SUM + X[I%]
240 SUMSQUARES = SUMSQUARES + X[I%]^2
250 '
260 MEAN = SUM/N% 'Calculate mean and standard deviation via Eq. 2-3
270 IF N% = 1 THEN SD = 0: GOTO 300
280 SD = SQR( (SUMSQUARES - SUM^2/N%) / (N%-1) )
290 '
300 PRINT MEAN SD 'Print the running mean and standard deviation
310 '
320 NEXT I%
330 '
340 END
```

TABLE 2-2

Before ending this discussion on the mean and standard deviation, two other terms need to be mentioned. In some situations, the *mean* describes what is being measured, while the *standard deviation* represents noise and other interference. In these cases, the standard deviation is not important in itself, but only in *comparison* to the mean. This gives rise to the term: **signal-to-noise ratio (SNR)**, which is equal to the mean divided by the standard deviation. Another term is also used, the **coefficient of variation (CV)**. This is defined as the standard deviation divided by the mean, multiplied by 100 percent. For example, a signal (or other group of measure values) with a CV of 2%, has an SNR of 50. Better data means a *higher* value for the SNR and a *lower* value for the CV.

Signal vs. Underlying Process

Statistics is the science of interpreting *numerical data*, such as acquired signals. In comparison, **probability** is used in DSP to understand the *processes* that generate signals. Although they are closely related, the distinction between the **acquired signal** and the **underlying process** is key to many DSP techniques.

For example, imagine creating a 1000 point signal by flipping a coin 1000 times. If the coin flip is heads, the corresponding sample is made a value of one. On tails, the sample is set to zero. The *process* that created this signal has a mean of exactly 0.5, determined by the relative probability of each possible outcome: 50% heads, 50% tails. However, it is unlikely that the actual 1000 point signal will have a mean of exactly 0.5. Random chance

EQUATION 2-4
Typical error in calculating the mean of an underlying process by using a finite number of samples, *N*. The parameter, , is the s standard deviation.

$$\textit{Typical error} \approx \frac{\sigma}{N^{1/2}}$$

will make the number of ones and zeros slightly different each time the signal is generated. The *probabilities* of the underlying process are constant, but the *statistics* of the acquired signal change each time the experiment is repeated. This random irregularity found in actual data is called by such names as: **statistical variation**, **statistical fluctuation**, and **statistical noise.**

This presents a bit of a dilemma. When you see the terms: *mean* and *standard deviation*, how do you know if the author is referring to the statistics of an actual signal, or the probabilities of the underlying process that created the signal? Unfortunately, the only way you can tell is by the context. This is not so for all terms used in statistics and probability. For example, the *histogram* and *probability mass function* (discussed in the next section) are matching concepts that are given separate names.

Now, back to Eq. 2-2, calculation of the standard deviation. As previously mentioned, this equation divides by *N*-1 in calculating the average of the squared deviations, rather than simply by *N*. To understand why this is so, imagine that you want to find the mean and standard deviation of some *process* that generates signals. Toward this end, you acquire a signal of *N* samples from the process, and calculate the mean of the signal via Eq. 2.1. You can then use this as an *estimate* of the mean of the underlying process; however, you know there will be an error due to statistical noise. In particular, for random signals, the typical error between the mean of the *N* points, and the mean of the underlying process, is given by:

If *N* is small, the statistical noise in the calculated mean will be very large. In other words, you do not have access to enough data to properly characterize the process. The larger the value of *N*, the smaller the expected error will become. A milestone in probability theory, the *Strong Law of Large Numbers*, guarantees that the error becomes zero as *N* approaches infinity.

In the next step, we would like to calculate the standard deviation of the acquired signal, and use it as an estimate of the standard deviation of the underlying process. Herein lies the problem. Before you can calculate the standard deviation using Eq. 2-2, you need to already know the mean, μ. However, you don't know the mean of the underlying process, only the mean of the *N* point signal, *which contains an error due to statistical noise*. This error tends to reduce the calculated value of the standard deviation. To compensate for this, *N* is replaced by *N*-1. If *N* is large, the difference doesn't matter. If *N* is small, this replacement provides a more accurate

Sample number
0 64 128 192 256 320 384 448 512
-4
-2
0
2
4
6
8
511
a. Changing mean and standard deviation
Sample number
0 64 128 192 256 320 384 448 512

-4
-2
0
2
4
6
8
511
b. Changing mean, constant standard deviation
Amplitude
Amplitude
FIGURE 2-3
Examples of signals generated from nonstationary processes. In (a), both the mean and standard deviation change. In (b), the standard deviation remains a constant value of one, while the mean changes from a value of zero to two. It is a common analysis technique to break these signals into short segments, and calculate the statistics of each segment individually.

estimate of the standard deviation of the underlying process. In other words, Eq. 2-2 is an *estimate* of the standard deviation of the *underlying process*. If we divided by *N* in the equation, it would provide the standard deviation of the *acquired signal*.

As an illustration of these ideas, look at the signals in Fig. 2-3, and ask: are the variations in these signals a result of statistical noise, or is the underlying process changing? It probably isn't hard to convince yourself that these changes are too large for random chance, and must be related to the underlying process. Processes that change their characteristics in this manner are called **nonstationary**. In comparison, the signals previously presented in Fig. 2-1 were generated from a stationary process, and the variations result completely from statistical noise. Figure 2-3b illustrates a common problem with nonstationary signals: the slowly changing *mean* interferes with the calculation of the *standard deviation*. In this example, the standard deviation of the signal, over a short interval, is *one*. However, the standard deviation of the entire signal is 1.16. This error can be nearly eliminated by breaking the signal into short sections, and calculating the statistics for each section individually. If needed, the standard deviations for each of the sections can be averaged to produce a single value.

The Histogram, Pmf and Pdf

Suppose we attach an 8 bit analog-to-digital converter to a computer, and acquire 256,000 samples of some signal. As an example, Fig. 2-4a shows 128 samples that might be a part of this data set. The value of each sample will be one of 256 possibilities, 0 through 255. The **histogram** displays the *number of samples* there are in the signal that have each of these *possible values*. Figure (b) shows the histogram for the 128 samples in (a). For

Value of sample
90 100 110 120 130 140 150 160 170
0
1
2
3
4
5
6
7
8
9
b. 128 point histogram
Value of sample
90 100 110 120 130 140 150 160 170
0
2000
4000
6000
8000
10000
c. 256,000 point histogram

Sample number
0 16 32 48 64 80 96 112 128
0
64
128
192
127
255
a. 128 samples of 8 bit signal
Amplitude
Number of occurences Number of occurences
FIGURE 2-4
Examples of histograms. Figure (a) shows 128 samples from a very long signal, with each sample being an integer between 0 and 255. Figures (b) and (c) show histograms using 128 and 256,000 samples from the signal, respectively. As shown, the histogram is smoother when more samples are used.
EQUATION 2-5
The sum of all of the values in the histogram is equal to the number of points in the signal. In this equation, H_i is the histogram, N is the number of points in the signal, and M is the number of points in the histogram.

$$N = \sum_{i=0}^{M-1} H_i$$

example, there are 2 samples that have a value of 110, 7 samples that have a value of 131, 0 samples that have a value of 170, etc. We will represent the histogram by H_i, where i is an index that runs from 0 to M-1, and M is the number of possible values that each sample can take on. For instance, H_{50} is the number of samples that have a value of 50. Figure (c) shows the histogram of the signal using the full data set, all 256k points. As can be seen, the larger number of samples results in a much smoother appearance. Just as with the mean, the statistical noise (roughness) of the histogram is inversely proportional to the square root of the number of samples used.

From the way it is defined, the sum of all of the values in the histogram must be equal to the number of points in the signal:

The histogram can be used to efficiently calculate the mean and standard deviation of very large data sets. This is especially important for *images*, which can contain millions of samples. The histogram groups samples

EQUATION 2-6
Calculation of the mean from the histogram. This can be viewed as combining all samples having the same value into groups, and then using Eq. 2-1 on each group.

$$\mu = \frac{1}{N} \sum_{i=0}^{M-1} i\, H_i$$

EQUATION 2-7
Calculation of the standard deviation from the histogram. This is the same concept as Eq. 2-2, except that all samples having the same value are operated on at once.

$$F_2 = \frac{1}{N-1}$$

$\sum_{i=0}^{M-1} (i - \mu)^2 H_i$

```
100 'CALCULATION OF THE HISTOGRAM, MEAN, AND STANDARD DEVIATION
110 '
120 DIM X%[25000] 'X%[0] to X%[25000] holds the signal being processed
130 DIM H%[255] 'H%[0] to H%[255] holds the histogram
140 N% = 25001 'Set the number of points in the signal
150 '
160 FOR I% = 0 TO 255 'Zero the histogram, so it can be used as an accumulator
170 H%[I%] = 0
180 NEXT I%
190 '
200 GOSUB XXXX 'Mythical subroutine that loads the signal into X%[ ]
210 '
220 FOR I% = 0 TO 25000 'Calculate the histogram for 25001 points
230 H%[ X%[I%] ] = H%[ X%[I%] ] + 1
240 NEXT I%
250 '
260 MEAN = 0 'Calculate the mean via Eq. 2-6
270 FOR I% = 0 TO 255
280 MEAN = MEAN + I% * H%[I%]
290 NEXT I%
300 MEAN = MEAN / N%
310 '
320 VARIANCE = 0 'Calculate the standard deviation via Eq. 2-7
330 FOR I% = 0 TO 255
340 VARIANCE = VARIANCE + H%[I%] * (I%-MEAN)^2
350 NEXT I%
360 VARIANCE = VARIANCE / (N%-1)
370 SD = SQR(VARIANCE)
380 '
390 PRINT MEAN SD 'Print the calculated mean and standard deviation.
400 '
410 END
```

TABLE 2-3

together that have the same value. This allows the statistics to be calculated by working with a few groups, rather than a large number of individual samples. Using this approach, the mean and standard deviation are calculated from the histogram by the equations:

Table 2-3 contains a program for calculating the histogram, mean, and standard deviation using these equations. Calculation of the histogram is very fast, since it only requires indexing and incrementing. In comparison,

calculating the mean and standard deviation requires the time consuming operations of addition and multiplication. The strategy of this algorithm is to use these slow operations only on the few numbers in the histogram, not the many samples in the signal. This makes the algorithm much faster than the previously described methods. Think a factor of ten for very long signals with the calculations being performed on a general purpose computer.

The notion that the acquired signal is a noisy version of the underlying process is very important; so important that some of the concepts are given different names. The *histogram* is what is formed from an acquired signal. The corresponding curve for the underlying process is called the **probability mass function (pmf)**. A histogram is always calculated using a finite number of samples, while the pmf is what *would be* obtained with an infinite number of samples. The pmf can be estimated (inferred) from the histogram, or it may be deduced by some mathematical technique, such as in the coin

flipping example.

Figure 2-5 shows an example pmf, and one of the possible histograms that could be associated with it. The key to understanding these concepts rests in the units of the vertical axis. As previously described, the vertical axis of the histogram is the *number of times* that a particular value occurs in the signal. The vertical axis of the pmf contains similar information, except expressed on a *fractional basis*. In other words, each value in the histogram is divided by the total number of samples to approximate the pmf. This means that each value in the pmf must be between zero and one, and that the sum of all of the values in the pmf will be equal to one.

The pmf is important because it describes the *probability* that a certain value will be generated. For example, imagine a signal with the pmf of Fig. 2-5b, such as previously shown in Fig. 2-4a. What is the probability that a sample taken from this signal will have a value of 120? Figure 2-5b provides the answer, 0.03, or about 1 chance in 34. What is the probability that a randomly chosen sample will have a value greater than 150? Adding up the values in the pmf for: 151, 152, 153,@@@, 255, provides the answer, 0.0122, or about 1 chance in 82. Thus, the signal would be expected to have a value exceeding 150 on an average of every 82 points. What is the probability that any one sample will be between 0 and 255? Summing all of the values in the pmf produces the probability of 1.00, that is, a certainty that this will occur.

The histogram and pmf can only be used with discrete data, such as a digitized signal residing in a computer. A similar concept applies to continuous signals, such as voltages appearing in analog electronics. The **probability density function (pdf)**, also called the **probability distribution function**, is to continuous signals what the probability mass function is to discrete signals. For example, imagine an analog signal passing through an analog-to-digital converter, resulting in the digitized signal of Fig. 2-4a. For simplicity, we will assume that voltages between 0 and 255 millivolts become digitized into digital numbers between 0 and 255. The pmf of this digital

Value of sample
90 100 110 120 130 140 150 160 170
0
2000
4000
6000
8000
10000
a. Histogram
Signal level (millivolts)
90 100 110 120 130 140 150 160 170
0.000
0.010
0.020
0.030
0.040
0.050
0.060
c. Probability Density Function (pdf)
Value of sample
90 100 110 120 130 140 150 160 170
0.000
0.010
0.020
0.030
0.040
0.050
0.060
b. Probability Mass Function (pmf)
Probability of occurence Probability density
Number of occurences

FIGURE 2-5

The relationship between (a) the histogram, (b) the probability mass function (pmf), and (c) the probability density function (pdf). The histogram is calculated from a finite number of samples. The pmf describes the probabilities of the underlying process. The pdf is similar to the pmf, but is used with continuous rather than discrete signals. Even though the vertical axis of (b) and (c) have the same values (0 to 0.06), this is only a coincidence of this example. The amplitude of these three curves is determined by: (a) the sum of the values in the histogram being equal to the number of samples in the signal; (b) the sum of the values in the pmf being equal to one, and (c) the area under the pdf curve being equal to one.

signal is shown by the *markers* in Fig. 2-5b. Similarly, the pdf of the analog signal is shown by the *continuous line* in (c), indicating the signal can take on a continuous range of values, such as the voltage in an electronic circuit.

The vertical axis of the pdf is in units of **probability density**, rather than just probability. For example, a pdf of 0.03 at 120.5 *does not* mean that the a voltage of 120.5 millivolts will occur 3% of the time. In fact, the probability of the continuous signal being exactly 120.5 millivolts is infinitesimally small. This is because there are an infinite number of possible values that the signal needs to divide its time between: 120.49997, 120.49998, 120.49999, etc. The chance that the signal happens to be exactly 120.50000þ is very remote indeed!

To calculate a *probability*, the *probability density* is multiplied by a *range* of values. For example, the probability that the signal, at any given instant, will be between the values of 120 and 121 is: (121&120) × 0.03 ’ 0.03. The probability that the signal will be between 120.4 and 120.5 is:

, etc. If the pdf is not constant over the range of (120.5&120.4) × 0.03 ’ 0.003 interest, the multiplication becomes the integral of the pdf over that range. In other words, the area under the pdf bounded by the specified values. Since the value of the signal must always be *something*, the total area under the pdf

Time (or other variable)

0 16 32 48 64 80 96 112 128

-2

-1

0

1

2

a. Square wave

127

pdf

FIGURE 2-6

Three common waveforms and their probability density functions. As in these examples, the pdf graph is often rotated one-quarter turn and placed at the side of the signal it describes. The pdf of a square wave, shown in (a), consists of two infinitesimally narrow spikes, corresponding to the signal only having two possible values. The pdf of the triangle wave, (b), has a constant value over a range, and is often called a *uniform* distribution. The pdf of random noise, as in (c), is the most interesting of all, a bell shaped curve known as a *Gaussian*.

Time (or other variable)

0 16 32 48 64 80 96 112 128
-2
-1
0
1
2
127
pdf
b. Triangle wave
Time (or other variable)
0 16 32 48 64 80 96 112 128
-2
-1
0
1
2
127
pdf
c. Random noise
Amplitude Amplitude Amplitude
curve, the integral from to , will always be equal to one. This is &4 %4 analogous to the sum of all of the pmf values being equal to one, and the sum of all of the histogram values being equal to *N*.

The histogram, pmf, and pdf are very similar concepts. Mathematicians always keep them straight, but you will frequently find them used interchangeably (and therefore, incorrectly) by many scientists and


```
100 'CALCULATION OF BINNED HISTOGRAM
110 '
120 DIM X[25000] 'X[0] to X[25000] holds the floating point signal,
130 ' 'with each sample having a value between 0.0 and 10.0.
140 DIM H%[999] 'H%[0] to H%[999] holds the binned histogram
150 '
160 FOR I% = 0 TO 999 'Zero the binned histogram for use as an accumulator
170 H%[I%] = 0
180 NEXT I%
190 '
200 GOSUB XXXX 'Mythical subroutine that loads the signal into X%[ ]
210 '
220 FOR I% = 0 TO 25000 ' 'Calculate the binned histogram for 25001 points
230 BINNUM% = INT( X[I%] * 100 )
240 H%[ BINNUM%] = H%[ BINNUM%] + 1
250 NEXT I%
260 '
270 END
```

TABLE 2-4

engineers. Figure 2-6 shows three *continuous* waveforms and their *pdfs*. If these were *discrete signals*, signified by changing the horizontal axis labeling to "sample number," *pmfs* would be used.

A problem occurs in calculating the histogram when the number of levels each sample can take on is much larger than the number of samples in the signal. This is always true for signals represented in *floating point* notation, where each sample is stored as a fractional value. For example, integer representation might require the sample value to be 3 or 4, while floating point allows millions of possible fractional values *between* 3 and 4. The previously described approach for calculating the histogram involves counting the number of samples that have each of the possible quantization levels. This is not possible with floating point data because there are *billions* of possible levels that would have to be taken into account. Even worse, nearly all of these possible levels would have no samples that correspond to them. For example, imagine a 10,000 sample signal, with each sample having one billion possible values. The conventional histogram would consist of one billion data points, with all but about 10,000 of them

having a value of zero.

The solution to these problems is a technique called **binning**. This is done by arbitrarily selecting the length of the histogram to be some convenient number, such as 1000 points, often called **bins**. The value of each bin represents the total number of samples in the signal that have a value within a *certain range*. For example, imagine a floating point signal that contains values between 0.0 and 10.0, and a histogram with 1000 bins. Bin 0 in the histogram is the number of samples in the signal with a value between 0 and 0.01, bin 1 is the number of samples with a value between 0.01 and 0.02, and so forth, up to bin 999 containing the number of samples with a value between 9.99 and 10.0. Table 2-4 presents a program for calculating a binned histogram in this manner.

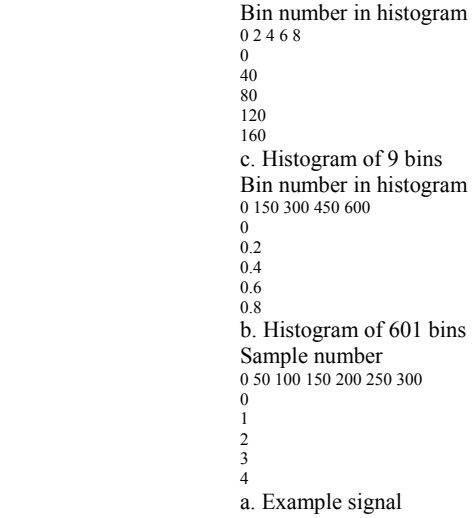
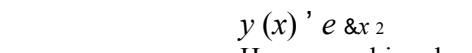


FIGURE 2-7
Example of binned histograms. As shown in (a), the signal used in this example is 300 samples long, with each sample a floating point number uniformly distributed between 1 and 3. Figures (b) and (c) show binned histograms of this signal, using 601 and 9 bins, respectively. As shown, a large number of bins results in poor resolution along the *vertical axis*, while a small number of bins provides poor resolution along the *horizontal axis.* Using more samples makes the resolution better in both directions.



$y(x) = e^{-x^2}$

How many bins should be used? This is a compromise between two problems. As shown in Fig. 2-7, too many bins makes it difficult to estimate the *amplitude* of the underlying pmf. This is because only a few samples fall into each bin, making the statistical noise very high. At the other extreme, too few of bins makes it difficult to estimate the underlying pmf in the *horizontal* direction. In other words, the number of bins controls a tradeoff between resolution along the y-axis, and resolution along the x-axis.

The Normal Distribution

Signals formed from random processes usually have a bell shaped pdf. This is called a **normal distribution**, a **Gauss distribution**, or a **Gaussian**, after the great German mathematician, Karl Friedrich Gauss (1777-1855). The

reason why this curve occurs so frequently in nature will be discussed shortly in conjunction with *digital noise generation*. The basic shape of the curve is generated from a *negative squared exponent*:

$$P(x) = \frac{1}{2\sigma}\, e^{-(x-\mu)^2/2\sigma^2}$$

EQUATION 2-8

Equation for the *normal distribution*, also called the *Gauss distribution*, or simply a *Gaussian*. In this relation, $P(x)$ is the probability distribution function, μ is the mean, and σ is the standard deviation.

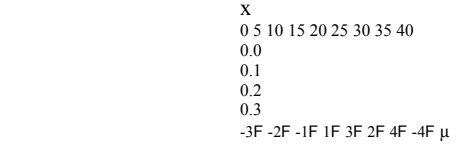


FIGURE 2-8

Examples of Gaussian curves. Figure (a) shows the shape of the raw curve without normalization or the addition of adjustable parameters. In (b) and (c), the complete Gaussian curve is shown for various means and standard deviations.

y(x)
P(x) P(x)

This raw curve can be converted into the complete Gaussian by adding an adjustable mean, μ, and standard deviation, σ. In addition, the equation must be normalized so that the total area under the curve is equal to *one*, a requirement of all probability distribution functions. This results in the general form of the normal distribution, one of the most important relations in statistics and probability:

Figure 2-8 shows several examples of Gaussian curves with various means and standard deviations. The *mean* centers the curve over a particular value, while the *standard deviation* controls the width of the bell shape.

An interesting characteristic of the Gaussian is that the *tails* drop toward zero very rapidly, much faster than with other common functions such as decaying exponentials or $1/x$. For example, at two, four, and six standard

deviations from the mean, the value of the Gaussian curve has dropped to about 1/19, 1/7563, and 1/166,666,666, respectively. This is why normally distributed signals, such as illustrated in Fig. 2-6c, *appear* to have an approximate peak-to-peak value. In principle, signals of this type can

experience excursions of unlimited amplitude. In practice, the sharp drop of the Gaussian pdf dictates that these extremes almost never occur. This results in the waveform having a relatively bounded appearance with an apparent peakto-peak amplitude of about 6-8F.

As previously shown, the integral of the pdf is used to find the probability that a signal will be within a certain range of values. This makes the integral of the pdf important enough that it is given its own name, the **cumulative distribution function (cdf)**. An especially obnoxious problem with the Gaussian is that it cannot be integrated using elementary methods. To get around this, the integral of the Gaussian can be calculated by *numerical integration*. This involves sampling the continuous Gaussian curve very finely, say, a few million points between -10F and +10F. The samples in this discrete signal are then *added* to simulate *integration*. The discrete curve resulting from this simulated integration is then stored in a table for use in calculating probabilities.

The cdf of the normal distribution is shown in Fig. 2-9, with its numeric values listed in Table 2-5. Since this curve is used so frequently in probability, it is given its own symbol: (upper case Greek *phi*). For M(x) example, has a value of 0.0228. This indicates that there is a 2.28% M(&2) probability that the value of the signal will be between -4 and two standard deviations below the mean, at any randomly chosen time. Likewise, the value: , means there is an 84.13% chance that the value of the M(1) ' 0.8413 signal, at a randomly selected instant, will be between -4 and one standard deviation above the mean. To calculate the probability that the signal will be will be *between* two values, it is necessary to subtract the appropriate numbers found in the table. For example, the probability that the M(x) value of the signal, at some randomly chosen time, will be between two standard deviations below the mean and one standard deviation above the mean, is given by: , or 81.85% M(1) & M(&2) ' 0.8185

Using this method, samples taken from a normally distributed signal will be within ±1F of the mean about 68% of the time. They will be within ±2F about 95% of the time, and within ±3F about 99.75% of the time. The probability of the signal being more than 10 standard deviations from the mean is so minuscule, it would be expected to occur for only a few *microseconds* since the beginning of the universe, about 10 billion years!

Equation 2-8 can also be used to express the probability mass function of normally distributed *discrete* signals. In this case, *x* is restricted to be one of the quantized levels that the signal can take on, such as one of the 4096 binary values exiting a 12 bit analog-to-digital converter. Ignore the 1/ 2BF term, it is only used to make the total area under the pdf curve equal to *one*. Instead, you must include whatever term is needed to make the sum of all the values in the pmf equal to *one*. In most cases, this is done by

x
-4 -3 -2 -1 0 1 2 3 4
0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
1.0
M(x)

FIGURE 2-9 & TABLE 2-5

M(x), the cumulative distribution function of the normal distribution (mean = 0, standard deviation = 1). These values are calculated by numerically integrating the normal distribution shown in Fig. 2-8b. In words, M(x) is the probability that the value of a normally distributed signal, at some randomly chosen time, will be less than x. In this table, the value of x is expressed in units of standard deviations referenced to the mean.

x	M(x)
-3.4	.0003
-3.3	.0005
-3.2	.0007
-3.1	.0010
-3.0	.0013
-2.9	.0019
-2.8	.0026
-2.7	.0035
-2.6	.0047
-2.5	.0062
-2.4	.0082
-2.3	.0107
-2.2	.0139
-2.1	.0179
-2.0	.0228
-1.9	.0287
-1.8	.0359
-1.7	.0446
-1.6	.0548
-1.5	.0668
-1.4	.0808
-1.3	.0968
-1.2	.1151
-1.1	.1357
-1.0	.1587
-0.9	.1841
-0.8	.2119
-0.7	.2420
-0.6	.2743
-0.5	.3085
-0.4	.3446
-0.3	.3821
-0.2	.4207
-0.1	.4602
0.0	.5000

x	M(x)
0.0	.5000
0.1	.5398
0.2	.5793
0.3	.6179
0.4	.6554
0.5	.6915
0.6	.7257
0.7	.7580
0.8	.7881
0.9	.8159
1.0	.8413
1.1	.8643
1.2	.8849
1.3	.9032
1.4	.9192
1.5	.9332
1.6	.9452
1.7	.9554
1.8	.9641
1.9	.9713
2.0	.9772

2.1 .9821
2.2 .9861
2.3 .9893
2.4 .9918
2.5 .9938
2.6 .9953
2.7 .9965
2.8 .9974
2.9 .9981
3.0 .9987
3.1 .9990
3.2 .9993
3.3 .9995
3.4 .9997

generating the curve without worrying about normalization, summing all of the unnormalized values, and then dividing all of the values by the sum.

Digital Noise Generation

Random noise is an important topic in both electronics and DSP. For example, it limits how small of a signal an instrument can measure, the distance a radio system can communicate, and how much radiation is required to produce an xray image. A common need in DSP is to generate signals that resemble various types of random noise. This is required to test the performance of algorithms that must *work* in the presence of noise.

The heart of digital noise generation is the **random number generator**. Most programming languages have this as a standard function. The *BASIC* statement: *X = RND*, loads the variable, *X*, with a new random number each time the command is encountered. Each random number has a value between zero and one, with an equal probability of being anywhere between these two extremes. Figure 2-10a shows a signal formed by taking 128 samples from this type of random number generator. The mean of the underlying process that generated this signal is 0.5, the standard deviation is , and the 1 / 12 ’ 0.29 distribution is uniform between zero and one.

EQUATION 2-9

Generation of normally distributed random numbers. R_1 and R_2 are random numbers with a uniform distribution between zero and one. This results in *X* being normally distributed with a mean of zero, and a standard deviation of one. The log is base *e,* and the cosine is in radians.

$$X \text{’ } (\&2 \log R_1)^{1/2} \cos(2\mathrm{B} R_2)$$

Algorithms need to be tested using the same kind of data they will encounter in actual operation. This creates the need to generate digital noise with a *Gaussian* pdf. There are two methods for generating such signals using a random number generator. Figure 2-10 illustrates the first method. Figure (b) shows a signal obtained by adding two random numbers to form each sample, i.e., *X = RND+RND*. Since each of the random numbers can run from zero to one, the sum can run from zero to two. The mean is now *one*, and the standard deviation is (remember, when 1 / 6 independent random signals are added, the variances also add). As shown, the pdf has changed from a *uniform* distribution to a *triangular* distribution. That is, the signal spends more of its time around a value of *one*, with less time spent near *zero* or *two*.

Figure (c) takes this idea a step further by adding twelve random numbers to produce each sample. The mean is now *six*, and the standard deviation is *one*. What is most important, the pdf has virtually become a *Gaussian*.

This procedure can be used to create a normally distributed noise signal with an arbitrary mean and standard deviation. For each sample in the signal: (1) add twelve random numbers, (2) subtract six to make the mean equal to zero, (3) multiply by the standard deviation desired, and (4) add the desired mean.

The mathematical basis for this algorithm is contained in the **Central Limit Theorem**, one of the most important concepts in probability. In its simplest form, the Central Limit Theorem states that a *sum* of random numbers becomes normally distributed as more and more of the random numbers are added together. The Central Limit Theorem *does not* require the individual random numbers be from any particular distribution, or even that the random numbers be from the *same* distribution. The Central Limit Theorem provides the reason why normally distributed signals are seen so widely in nature. Whenever many different random forces are interacting, the resulting pdf becomes a Gaussian.

In the second method for generating normally distributed random numbers, the random number generator is invoked *twice*, to obtain R_1 and R_2. A normally distributed random number, X, can then be found:

Just as before, this approach can generate normally distributed random signals with an arbitrary mean and standard deviation. Take each number generated by this equation, multiply it by the desired standard deviation, and add the desired mean.

Sample number
0 16 32 48 64 80 96 112 128
0
1
2
3
4
5
6
7
8
9
10
11
12
a. X = RND
127
mean = 0.5, F = 1/% 12
Sample number
0 16 32 48 64 80 96 112 128
0
1
2
3
4
5
6
7
8
9
10
11
12
c. X = RND+RND+ ... +RND (12 times)
127
mean = 6.0, F = 1
Sample number
0 16 32 48 64 80 96 112 128
0
1
2
3
4
5
6
7
8
9

10
11
12
b. X = RND+RND
127
mean = 1.0, F = 1/% 6
pdf
pdf
pdf
FIGURE 2-10
Converting a uniform distribution to a Gaussian distribution. Figure (a) shows a signal where each sample is generated by a random number generator. As indicated by the pdf, the value of each sample is uniformly distributed between zero and one. Each sample in (b) is formed by adding *two* values from the random number generator. In (c), each sample is created by adding *twelve* values from the random number generator. The pdf of (c) is very nearly Gaussian, with a mean of *six*, and a standard deviation of *one.*
Amplitude Amplitude Amplitude

EQUATION 2-10
Common algorithm for generating uniformly distributed random numbers between zero and one. In this method, *S* is the seed, *R* is the new random number, and *a*,*b*,& *c* are appropriately chosen constants. In words, the quantity *aS+b* is divided by *c*, and the remainder is taken as *R*.
R ' (*aS*% *b*) modulo *c*
Random number generators operate by starting with a **seed**, a number between zero and one. When the random number generator is invoked, the seed is passed through a fixed algorithm, resulting in a new number between zero and one. This new number is reported as the *random number*, and is then internally stored to be used as the seed the next time the random number generator is called. The algorithm that transforms the seed into the new random number is often of the form:
In this manner, a continuous sequence of random numbers can be generated, all starting from the same seed. This allows a program to be run multiple times using exactly the same random number sequences. If you want the random number sequence to change, most languages have a provision for **reseeding** the random number generator, allowing you to choose the number first used as the seed. A common technique is to use the *time* (as indicated by the system's clock) as the seed, thus providing a new sequence each time the program is run. From a pure mathematical view, the numbers generated in this way cannot be absolutely random since each number is fully determined by the *previous* number. The term **pseudo-random** is often used to describe this situation. However, this is not something you should be concerned with. The sequences generated by random number generators are statistically random to an exceedingly high degree. It is very unlikely that you will encounter a situation where they are not adequate.

Precision and Accuracy

Precision and accuracy are terms used to describe systems and methods that *measure*, *estimate*, or *predict*. In all these cases, there is some parameter you wish to know the value of. This is called the **true value**, or simply, **truth**. The method provides a **measured value**, that you want to be as close to the true value as possible. *Precision* and *accuracy* are ways of describing the error that can exist between these two values.
Unfortunately, precision and accuracy are used interchangeably in non-technical settings. In fact, dictionaries define them by referring to each other! In spite of this, science and engineering have very specific definitions for each. You should make a point of using the terms correctly, and quietly tolerate others

when they use them incorrectly.

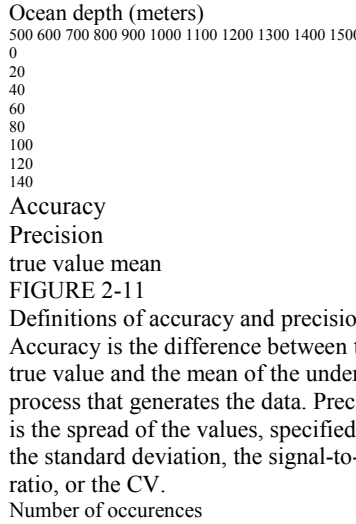


FIGURE 2-11
Definitions of accuracy and precision. Accuracy is the difference between the true value and the mean of the under-lying process that generates the data. Precision is the spread of the values, specified by the standard deviation, the signal-to-noise ratio, or the CV.

As an example, consider an oceanographer measuring water depth using a *sonar* system. Short bursts of sound are transmitted from the ship, reflected from the ocean floor, and received at the surface as an echo. Sound waves travel at a relatively constant velocity in water, allowing the depth to be found from the elapsed time between the transmitted and received pulses. As with all empirical measurements, a certain amount of error exists between the measured and true values. This particular measurement could be affected by many factors: random noise in the electronics, waves on the ocean surface, plant growth on the ocean floor, variations in the water temperature causing the sound velocity to change, etc.

To investigate these effects, the oceanographer takes many successive readings at a location known to be *exactly* 1000 meters deep (the true value). These measurements are then arranged as the histogram shown in Fig. 2-11. As would be expected from the Central Limit Theorem, the acquired data are normally distributed. The *mean* occurs at the center of the distribution, and represents the best estimate of the depth based on all of the measured data. The *standard deviation* defines the width of the distribution, describing how much variation occurs between successive measurements.

This situation results in two general types of error that the system can experience. First, the mean may be shifted from the true value. The amount of this shift is called the **accuracy** of the measurement. Second, individual measurements may not agree well with each other, as indicated by the width of the distribution. This is called the **precision** of the measurement, and is expressed by quoting the standard deviation, the signal-to-noise ratio, or the CV.

Consider a measurement that has good accuracy, but poor precision; the histogram is centered over the true value, but is very broad. Although the measurements are correct *as a group*, each individual reading is a poor measure of the true value. This situation is said to have poor *repeatability;* measurements taken in succession don't agree well. Poor precision results from **random errors**. This is the name given to errors that change each

time the measurement is repeated. Averaging several measurements will *always* improve the precision. In short, *precision is a measure of random noise.*

Now, imagine a measurement that is very precise, but has poor accuracy. This makes the histogram very slender, but not centered over the true value. Successive readings are close in value; however, they *all* have a large error. Poor accuracy results from **systematic errors.** These are errors that become repeated in exactly the same manner each time the measurement is conducted. Accuracy is usually dependent on how you *calibrate* the system. For example, in the ocean depth measurement, the parameter directly measured is elapsed time. This is converted into depth by a calibration procedure that relates *milliseconds* to *meters*. This may be as simple as multiplying by a fixed velocity, or as complicated as dozens of second order corrections. Averaging individual measurements does nothing to improve the accuracy. In short, *accuracy is a measure of calibration.*

In actual practice there are many ways that precision and accuracy can become intertwined. For example, imagine building an electronic amplifier from 1% resistors. This tolerance indicates that the value of each resistor will be within 1% of the stated value over a wide range of conditions, such as temperature, humidity, age, etc. This error in the resistance will produce a corresponding error in the gain of the amplifier. Is this error a problem of accuracy or precision?

The answer depends on how you take the measurements. For example, suppose you build *one* amplifier and test it several times over a few minutes. The error in gain remains constant with each test, and you conclude the problem is *accuracy*. In comparison, suppose you build *one thousand* of the amplifiers. The gain from device to device will fluctuate randomly, and the problem appears to be one of *precision*. Likewise, any one of these amplifiers will show gain fluctuations in response to temperature and other environmental changes. Again, the problem would be called *precision*.

When deciding which name to call the problem, ask yourself two questions. First: Will averaging successive readings provide a better measurement? If yes, call the error precision; if no, call it accuracy. Second: Will calibration correct the error? If yes, call it accuracy; if no, call it precision. This may require some thought, especially related to how the device will be calibrated, and how often it will be done.

CHAPTER

3 ADC and DAC

Most of the signals directly encountered in science and engineering are *continuous*: light intensity that changes with distance; voltage that varies over time; a chemical reaction rate that depends on temperature, etc. Analog-to-Digital Conversion (ADC) and Digital-to-Analog Conversion (DAC) are the processes that allow digital computers to interact with these everyday signals. Digital information is different from its continuous counterpart in two important respects: it is *sampled*, and it is *quantized*. Both of these restrict how much information a digital signal can contain. This chapter is about *information management*: understanding what information you need to retain, and what information you can afford to lose. In turn, this dictates the selection of the sampling frequency, number of bits, and type of analog filtering needed for converting between the analog and digital realms.

Quantization

First, a bit of trivia. As you know, it is a *digital* computer, not a *digit* computer. The information processed is called *digital* data, not *digit* data. Why then, is analog-to-digital conversion generally called: *digitize and*

digitization, rather than *digitalize and digitalization*? The answer is nothing you would expect. When electronics got around to inventing digital techniques, the preferred names had already been snatched up by the medical community nearly a century before. *Digitalize* and *digitalization* mean to administer the heart stimulant *digitalis*.

Figure 3-1 shows the electronic waveforms of a typical analog-to-digital conversion. Figure (a) is the analog signal to be digitized. As shown by the labels on the graph, this signal is a *voltage* that varies over *time*. To make the numbers easier, we will assume that the voltage can vary from 0 to 4.095 volts, corresponding to the digital numbers between 0 and 4095 that will be produced by a 12 bit digitizer. Notice that the block diagram is broken into two sections, the sample-and-hold (S/H), and the analog-to-digital converter (ADC). As you probably learned in electronics classes, the sample-and-hold is required to keep the voltage entering the ADC constant while the

conversion is taking place. However, this is *not* the reason it is shown here; breaking the digitization into these two stages is an important theoretical model for understanding digitization. The fact that it happens to look like common electronics is just a fortunate bonus.

As shown by the difference between (a) and (b), the output of the sample-andhold is allowed to change only at periodic intervals, at which time it is made identical to the instantaneous value of the input signal. Changes in the input signal that occur between these sampling times are completely ignored. That is, **sampling** converts the *independent variable* (time in this example) from continuous to discrete.

As shown by the difference between (b) and (c), the ADC produces an integer value between 0 and 4095 for each of the flat regions in (b). This introduces an error, since each plateau can be *any* voltage between 0 and 4.095 volts. For example, both 2.56000 volts and 2.56001 volts will be converted into digital number 2560. In other words, **quantization** converts the *dependent variable* (voltage in this example) from continuous to discrete.

Notice that we carefully avoid comparing (a) and (c), as this would lump the sampling and quantization together. It is important that we analyze them separately because they degrade the signal in different ways, as well as being controlled by different parameters in the electronics. There are also cases where one is used without the other. For instance, sampling without quantization is used in switched capacitor filters.

First we will look at the effects of quantization. Any one sample in the digitized signal can have a maximum error of **±½ LSB** (**Least Significant Bit**, jargon for the distance between adjacent quantization levels). Figure (d) shows the quantization error for this particular example, found by subtracting (b) from (c), with the appropriate conversions. In other words, the digital output (c), is equivalent to the continuous input (b), *plus* a quantization error (d). An important feature of this analysis is that the quantization error appears very much like *random noise*.

This sets the stage for an important model of quantization error. In most cases, *quantization results in nothing more than the addition of a specific amount of random noise to the signal*. The additive noise is uniformly distributed between ±½ LSB, has a mean of zero, and a standard deviation of LSB 1/ 12 (-0.29 LSB). For example, passing an analog signal through an 8 bit digitizer adds an rms noise of: , or about 1/900 of the full scale value. A 12 0.29/256 bit conversion adds a noise of: , while a 16 bit 0.29 /4096 . 1 /14,000 conversion adds: . Since quantization error is a 0.29/65536 . 1 /227,000

random noise, the *number of bits* determines the *precision* of the data. For example, you might make the statement: "We increased the precision of the measurement from 8 to 12 bits."

This model is extremely powerful, because the random noise generated by quantization will simply add to whatever noise is already present in the

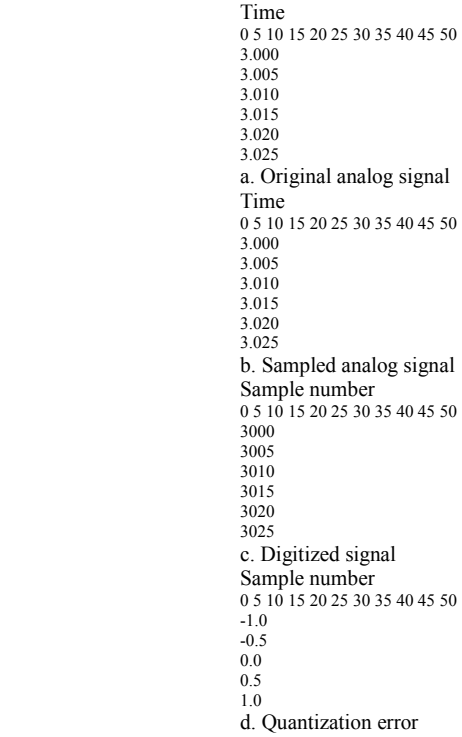


FIGURE 3-1
Waveforms illustrating the digitization process. The conversion is broken into two stages to allow the effects of *sampling* to be separated from the effects of *quantization*. The first stage is the sample-and-hold (S/H), where the only information retained is the instantaneous value of the signal when the periodic sampling takes place. In the second stage, the ADC converts the voltage to the nearest integer number. This results in each sample in the digitized signal having an error of up to ±½ LSB, as shown in (d). As a result, quantization can usually be modeled as simply adding noise to the signal.
Amplitude (in volts) Amplitude (in volts)
Digital number
Error (in LSBs)

analog signal. For example, imagine an analog signal with a maximum amplitude of 1.0 volt, and a random noise of 1.0 millivolt rms. Digitizing this signal to 8 bits results in 1.0 volt becoming digital number 255, and 1.0 millivolt becoming 0.255 LSB. As discussed in the last chapter, random noise

signals are combined by adding their *variances*. That is, the signals are added in quadrature: . The total noise on the digitized signal is $A_2 \% B_2 ' C$ therefore given by: LSB. This is an increase of about $0.255_2 \% 0.29_2 ' 0.386$ 50% over the noise already in the analog signal. Digitizing this same signal to 12 bits would produce virtually no increase in the noise, and *nothing* would be lost due to quantization. When faced with the decision of how many bits are needed in a system, ask two questions: (1) How much noise is *already* present in the analog signal? (2) How much noise can be *tolerated* in the digital signal?

When isn't this model of quantization valid? Only when the quantization error cannot be treated as random. The only common occurrence of this is when the analog signal remains at about the same value for many consecutive samples, as is illustrated in Fig. 3-2a. The output remains *stuck* on the same digital number for many samples in a row, even though the analog signal may be changing up to ±½ LSB. Instead of being an additive random noise, the quantization error now looks like a thresholding effect or weird distortion.

Dithering is a common technique for improving the digitization of these slowly varying signals. As shown in Fig. 3-2b, a small amount of random noise is added to the analog signal. In this example, the added noise is normally distributed with a standard deviation of 2/3 LSB, resulting in a peakto-peak amplitude of about 3 LSB. Figure (c) shows how the addition of this dithering noise has affected the digitized signal. Even when the original analog signal is changing by less than ±½ LSB, the added noise causes the digital output to randomly toggle between adjacent levels.

To understand how this improves the situation, imagine that the input signal is a constant analog voltage of 3.0001 volts, making it one-tenth of the way between the digital levels 3000 and 3001. Without dithering, taking 10,000 samples of this signal would produce 10,000 identical numbers, all having the value of 3000. Next, repeat the thought experiment with a small amount of dithering noise added. The 10,000 values will now oscillate between two (or more) levels, with about 90% having a value of 3000, and 10% having a value of 3001. Taking the average of all 10,000 values results in something close to 3000.1. Even though a single measurement has the inherent ±½ LSB limitation, the statistics of a large number of the samples can do much better. This is quite a strange situation: *adding noise provides more information*.

Circuits for dithering can be quite sophisticated, such as using a computer to generate random numbers, and then passing them through a DAC to produce the added noise. After digitization, the computer can *subtract*

Time (or sample number)
0 5 10 15 20 25 30 35 40 45 50
3000
3001
3002
3003
3004
3005
original analog signal
digital signal
c. Digitization of dithered signal
Time (or sample number)
0 5 10 15 20 25 30 35 40 45 50
3000
3001
3002
3003
3004
3005

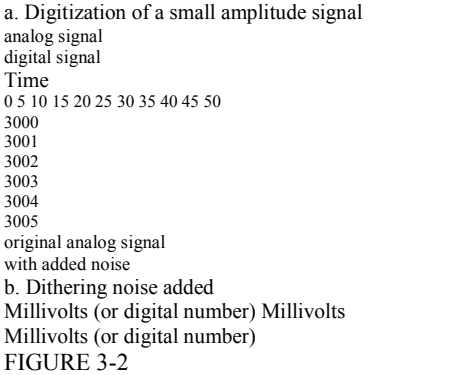


FIGURE 3-2
Illustration of dithering. Figure (a) shows how an analog signal that varies less than ±½ LSB can become *stuck* on the same quantization level during digitization. Dithering improves this situation by adding a small amount of random noise to the analog signal, such as shown in (b). In this example, the added noise is normally distributed with a standard deviation of 2/3 LSB. As shown in (c), the added noise causes the digitized signal to toggle between adjacent quantization levels, providing more information about the original signal.

the random numbers from the digital signal using floating point arithmetic. This elegant technique is called **subtractive dither**, but is only used in the most elaborate systems. The simplest method, although not always possible, is to use the noise already present in the analog signal for dithering.

The Sampling Theorem

The definition of *proper sampling* is quite simple. Suppose you sample a continuous signal in some manner. If you can exactly *reconstruct* the analog signal from the samples, you must have done the sampling *properly*. Even if the sampled data appears confusing or incomplete, the key information has been captured if you can reverse the process.

Figure 3-3 shows several sinusoids before and after digitization. The continuous line represents the analog signal entering the ADC, while the square markers are the digital signal leaving the ADC. In (a), the analog signal is a constant DC value, a cosine wave of *zero* frequency. Since the analog signal is a series of straight lines between each of the samples, all of the information needed to reconstruct the analog signal is contained in the digital data. According to our definition, this is *proper sampling*.

The sine wave shown in (b) has a frequency of 0.09 of the sampling rate. This might represent, for example, a 90 cycle/second sine wave being sampled at 1000 samples/second. Expressed in another way, there are 11.1 samples taken over each complete cycle of the sinusoid. This situation is more complicated than the previous case, because the analog signal cannot be reconstructed by simply drawing straight lines between the data points. Do these samples properly represent the analog signal? The answer is yes, because no other sinusoid, or combination of sinusoids, will produce this pattern of samples (within the reasonable constraints listed below). These samples correspond to only one analog signal, and therefore the analog signal can be exactly reconstructed. Again, an instance of *proper sampling*.

In (c), the situation is made more difficult by increasing the sine wave's frequency to 0.31 of the sampling rate. This results in only 3.2 samples per

sine wave cycle. Here the samples are so sparse that they don't even appear to follow the general trend of the analog signal. Do these samples properly represent the analog waveform? Again, the answer is yes, and for exactly the same reason. The samples are a unique representation of the analog signal. All of the information needed to reconstruct the continuous waveform is contained in the digital data. How you go about doing this will be discussed later in this chapter. Obviously, it must be more sophisticated than just drawing straight lines between the data points. As strange as it seems, this is *proper sampling* according to our definition.

In (d), the analog frequency is pushed even higher to 0.95 of the sampling rate, with a mere 1.05 samples per sine wave cycle. Do these samples properly represent the data? *No, they don't!* The samples represent a *different* sine wave from the one contained in the analog signal. In particular, the original sine wave of 0.95 frequency misrepresents itself as a sine wave of 0.05 frequency in the digital signal. This phenomenon of sinusoids changing frequency during sampling is called **aliasing**. Just as a criminal might take on an assumed name or identity (an *alias*), the sinusoid assumes another frequency that is not its own. Since the digital data is no longer uniquely related to a particular analog signal, an unambiguous reconstruction is impossible. There is nothing in the sampled data to suggest that the original analog signal had a frequency of 0.95 rather than 0.05. The sine wave has hidden its true identity completely; the perfect crime has been committed! According to our definition, this is an example of *improper sampling*.

This line of reasoning leads to a milestone in DSP, the **sampling theorem**. Frequently this is called the *Shannon* sampling theorem, or the *Nyquist* sampling theorem, after the authors of 1940s papers on the topic. The sampling theorem indicates that a continuous signal can be properly sampled, *only if it does not contain frequency components above one-half of the sampling rate*. For instance, a sampling rate of 2,000 samples/second requires the analog signal to be composed of frequencies below 1000 cycles/second. If frequencies above this limit *are* present in the signal, they will be aliased to frequencies between 0 and 1000 cycles/second, combining with whatever information that was legitimately there.

Time (or sample number)
-3
-2
-1
0
1
2
3
c. Analog frequency = 0.31 of sampling rate
Time (or sample number)
-3
-2
-1
0
1
2
3
d. Analog frequency = 0.95 of sampling rate
Time (or sample number)
-3
-2
-1
0
1
2
3
a. Analog frequency = 0.0 (i.e., DC)
Time (or sample number)
-3

-2
-1
0
1
2
3
b. Analog frequency = 0.09 of sampling rate
Amplitude Amplitude
Amplitude
FIGURE 3-3
Illustration of proper and improper sampling. A continuous signal is sampled *properly* if the samples contain all the information needed to recreate the original waveform. Figures (a), (b), and (c) illustrate *proper sampling* of three sinusoidal waves. This is certainly not obvious, since the samples in (c) do not even appear to capture the shape of the waveform. Nevertheless, each of these continuous signals forms a unique one-to-one pair with its pattern of samples. This guarantees that reconstruction can take place. In (d), the frequency of the analog sine wave is greater than the Nyquist frequency (one-half of the sampling rate). This results in *aliasing*, where the frequency of the sampled data is different from the frequency of the continuous signal. Since aliasing has corrupted the information, the original signal cannot be reconstructed from the samples.
Amplitude

Two terms are widely used when discussing the sampling theorem: the **Nyquist frequency** and the **Nyquist rate**. Unfortunately, their meaning is not standardized. To understand this, consider an analog signal composed of frequencies between DC and 3 kHz. To properly digitize this signal it must be sampled at 6,000 samples/sec (6 kHz) or higher. Suppose we choose to sample at 8,000 samples/sec (8 kHz), allowing frequencies between DC and 4 kHz to be properly represented. In this situation there are four important frequencies: (1) the highest frequency in the signal, 3 kHz; (2) twice this frequency, 6 kHz; (3) the sampling rate, 8 kHz; and (4) one-half the sampling rate, 4 kHz. Which of these four is the *Nyquist frequency* and which is the *Nyquist rate*? It depends who you ask! All of the possible combinations are

used. Fortunately, most authors are careful to define how they are using the terms. In this book, they are both used to mean *one-half the sampling rate*.

Figure 3-4 shows how frequencies are changed during aliasing. The key point to remember is that a digital signal *cannot* contain frequencies above one-half the sampling rate (i.e., the Nyquist frequency/rate). When the frequency of the continuous wave is below the Nyquist rate, the frequency of the sampled data is a match. However, when the continuous signal's frequency is above the Nyquist rate, aliasing *changes* the frequency into something that *can* be represented in the sampled data. As shown by the zigzagging line in Fig. 3-4, every continuous frequency above the Nyquist rate has a corresponding digital frequency between zero and one-half the sampling rate. If there happens to be a sinusoid already at this lower frequency, the aliased signal will add to it, resulting in a loss of information. Aliasing is a double curse; information can be lost about the higher *and* the lower frequency. Suppose you are given a digital signal containing a frequency of 0.2 of the sampling rate. If this signal were obtained by proper sampling, the original analog signal *must* have had a frequency of 0.2. If aliasing took place during sampling, the digital frequency of 0.2 could have come from any one of an infinite number of frequencies in the analog signal: 0.2, 0.8, 1.2, 1.8, 2.2, þ .

Just as aliasing can change the frequency during sampling, it can also change the *phase*. For example, look back at the aliased signal in Fig. 3-3d. The aliased digital signal is *inverted* from the original analog signal; one is a sine wave while the other is a negative sine wave. In other words, aliasing has changed the frequency *and* introduced a 180E phase shift. Only two phase shifts are possible: 0E (no phase shift) and 180E (inversion). The zero phase shift occurs for analog frequencies of 0 to 0.5, 1.0 to 1.5, 2.0 to 2.5, etc. An

inverted phase occurs for analog frequencies of 0.5 to 1.0, 1.5 to 2.0, 2.5 to 3.0, and so on.

Now we will dive into a more detailed analysis of sampling and how aliasing occurs. Our overall goal is to understand what happens to the information when a signal is converted from a continuous to a discrete form. The problem is, these are very different things; one is a *continuous waveform* while the other is an *array of numbers*. This "apples-to-oranges" comparison makes the analysis very difficult. The solution is to introduce a theoretical concept called the **impulse train**.

Figure 3-5a shows an example analog signal. Figure (c) shows the signal sampled by using an *impulse train*. The impulse train is a continuous signal consisting of a series of narrow spikes (impulses) that match the original signal at the sampling instants. Each impulse is infinitesimally narrow, a concept that will be discussed in Chapter 13. Between these sampling times the value of the waveform is zero. Keep in mind that the impulse train is a *theoretical* concept, not a waveform that can exist in an electronic circuit. Since both the original analog signal and the impulse train are continuous waveforms, we can make an "apples-apples" comparison between the two.

Continuous frequency (as a fraction of the sampling rate)
0.0 0.5 1.0 1.5 2.0 2.5
0.0
0.1
0.2
0.3
0.4
0.5
DC
Nyquist
Frequency
GOOD
ALIASED

FIGURE 3-4
Conversion of analog frequency into digital frequency during sampling. Continuous signals with a frequency less than one-half of the sampling rate are directly converted into the corresponding digital frequency. Above one-half of the sampling rate, aliasing takes place, resulting in the frequency being misrepresented in the digital data. Aliasing always changes a higher frequency into a lower frequency between 0 and 0.5. In addition, aliasing may also change the phase of the signal by 180 degrees.

Continuous frequency (as a fraction of the sampling rate)
0.0 0.5 1.0 1.5 2.0 2.5
-90
0
90
180
270
Digital frequency Digital phase (degrees)

Now we need to examine the relationship between the impulse train and the discrete signal (an array of numbers). This one is easy; in terms of *information content*, they are *identical*. If one is known, it is trivial to calculate the other. Think of these as different ends of a bridge crossing between the analog and digital worlds. This means we have achieved our overall goal once we understand the consequences of changing the waveform in Fig. 3-5a into the waveform in Fig. 3.5c.

Three continuous waveforms are shown in the left-hand column in Fig. 3-5. The corresponding *frequency spectra* of these signals are displayed in the righthand column. This should be a familiar concept from your knowledge of electronics; every waveform can be viewed as being composed of sinusoids of varying amplitude and frequency. Later chapters will discuss the frequency domain in detail. (You may want to revisit this discussion after becoming more

familiar with frequency spectra).
Figure (a) shows an analog signal we wish to sample. As indicated by its frequency spectrum in (b), it is composed only of frequency components between 0 and about 0.33 f_s, where f_s is the sampling frequency we intend to

use. For example, this might be a speech signal that has been filtered to remove all frequencies above 3.3 kHz. Correspondingly, f_s would be 10 kHz (10,000 samples/second), our intended sampling rate.
Sampling the signal in (a) by using an impulse train produces the signal shown in (c), and its frequency spectrum shown in (d). This spectrum is a *duplication* of the spectrum of the original signal. Each multiple of the sampling frequency, f_s, $2f_s$, $3f_s$, $4f_s$, etc., has received a *copy* and a *left-forright flipped copy* of the original frequency spectrum. The copy is called the **upper sideband**, while the flipped copy is called the **lower sideband**.
Sampling has generated *new* frequencies. Is this proper sampling? The answer is yes, because the signal in (c) can be transformed back into the signal in (a) by eliminating all frequencies above $\frac{1}{2}f_s$. That is, an analog low-pass filter will convert the impulse train, (b), back into the original analog signal, (a).
If you are already familiar with the basics of DSP, here is a more technical explanation of why this spectral duplication occurs. (Ignore this paragraph if you are new to DSP). In the time domain, sampling is achieved by multiplying the original signal by an impulse train of *unity amplitude* spikes. The frequency spectrum of this unity amplitude impulse train is also a unity amplitude impulse train, with the spikes occurring at multiples of the sampling frequency, f_s, $2f_s$, $3f_s$, $4f_s$, etc. When two time domain signals are multiplied, their frequency spectra are convolved. This results in the original spectrum being duplicated to the location of each spike in the impulse train's spectrum. Viewing the original signal as composed of both positive and negative frequencies accounts for the upper and lower sidebands, respectively. This is the same as amplitude modulation, discussed in Chapter 10.
Figure (e) shows an example of *improper sampling*, resulting from too low of sampling rate. The analog signal still contains frequencies up to 3.3 kHz, but the sampling rate has been lowered to 5 kHz. Notice that along the horizontal axis are spaced closer in (f) than in (d). f_s, $2f_s$, $3f_s$ þ
The frequency spectrum, (f), shows the problem: the duplicated portions of the spectrum have invaded the band between zero and one-half of the sampling frequency. Although (f) shows these overlapping frequencies as retaining their separate identity, in actual practice they add together forming a single confused mess. Since there is no way to separate the overlapping frequencies, information is lost, and the original signal cannot be reconstructed. This overlap occurs when the analog signal contains frequencies greater than one-half the sampling rate, that is, we have proven the sampling theorem.

Digital-to-Analog Conversion

In theory, the simplest method for digital-to-analog conversion is to pull the samples from memory and convert them into an *impulse train*. This is

Time
0 1 2 3 4 5
-3
-2
-1
0

1
2
3
a. Original analog signal
Frequency
0 100 200 300 400 500 600
0
1
2
3
b. Original signal's spectrum
0 f 2f 3fs s s
Time
0 1 2 3 4 5
-3
-2
-1
0
1
2
3
original signal
impulse train
c. Sampling at 3 times highest frequency
Frequency
0 100 200 300 400 500 600
0
1
2
3
d. Duplicated spectrum from sampling
upper
sideband
lower
sideband
0 f 2f 3fs s s
Time
0 1 2 3 4 5
-3
-2
-1
0
1
2
3
e. Sampling at 1.5 times highest frequency
original signal
impulse train
Frequency
0 100 200 300 400 500 600
0
1
2
3
f. Overlapping spectra causing aliasing
0 2f 4f 6fs s s fs 3f 5fs s
Time Domain Frequency Domain

FIGURE 3-5

The sampling theorem in the time and frequency domains. Figures (a) and (b) show an analog signal composed of frequency components between zero and 0.33 of the sampling frequency, f_s. In (c), the analog signal is sampled by converting it to an impulse train. In the frequency domain, (d), this results in the spectrum being duplicated into an infinite number of upper and lower sidebands. Since the original frequencies in (b) exist undistorted in (d), proper sampling has taken place. In comparison, the analog signal in (e) is sampled at 0.66 of the sampling frequency, a value exceeding the Nyquist rate. This results in aliasing, indicated by the sidebands in (f) overlapping.

Amplitude
Amplitude
Amplitude
Amplitude Amplitude
Amplitude

EQUATION 3-1

High frequency amplitude reduction due to the zeroth-order hold. This curve is plotted in Fig. 3-6d. The sampling frequency is represented by . For . fsf ' 0, $H(f)$ ' 1

$H(f)$ ' /
00
sin(Bf /f_s)
Bf /f_s
/00
illustrated in Fig. 3-6a, with the corresponding frequency spectrum in (b). As just described, the original analog signal can be perfectly reconstructed by passing this impulse train through a low-pass filter, with the cutoff frequency equal to one-half of the sampling rate. In other words, the original signal and the impulse train have identical frequency spectra below the Nyquist frequency (one-half the sampling rate). At higher frequencies, the impulse train contains a duplication of this information, while the original analog signal contains nothing (assuming aliasing did not occur).

While this method is mathematically pure, it is difficult to generate the required narrow pulses in electronics. To get around this, nearly all DACs operate by holding the last value until another sample is received. This is called a **zeroth-order hold**, the DAC equivalent of the sample-and-hold used during ADC. (A first-order hold is straight lines between the points, a second-order hold uses parabolas, etc.). The zeroth-order hold produces the staircase appearance shown in (c).

In the frequency domain, the zeroth-order hold results in the spectrum of the impulse train being *multiplied* by the dark curve shown in (d), given by the equation:

This is of the general form: , called the **sinc function** or **sinc(x)**. sin (Bx) / (Bx)

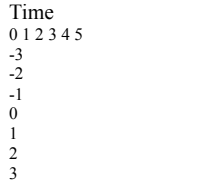
The sinc function is very common in DSP, and will be discussed in more detail in later chapters. If you already have a background in this material, the zerothorder hold can be understood as the convolution of the impulse train with a rectangular pulse, having a width equal to the sampling period. This results in the frequency domain being *multiplied* by the Fourier transform of the rectangular pulse, i.e., the sinc function. In Fig. (d), the light line shows the frequency spectrum of the impulse train (the "correct" spectrum), while the dark line shows the sinc. The frequency spectrum of the zeroth order hold signal is equal to the product of these two curves.

The analog filter used to convert the zeroth-order hold signal, (c), into the reconstructed signal, (f), needs to do two things: (1) remove all frequencies above one-half of the sampling rate, and (2) boost the frequencies by the reciprocal of the zeroth-order hold's effect, i.e., *1/sinc(x)*. This amounts to an amplification of about 36% at one-half of the sampling frequency. Figure (e) shows the ideal frequency response of this analog filter.

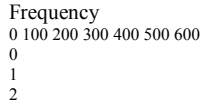
This *1/sinc(x)* frequency boost can be handled in four ways: (1) ignore it and accept the consequences, (2) design an analog filter to include the *1/sinc(x)*

FIGURE 3-6
Analysis of digital-to-analog conversion. In (a), the digital data are converted into an impulse train, with the spectrum in (b). This is changed into the reconstructed signal, (f), by using an electronic low-pass filter to remove frequencies above one-half the sampling rate [compare (b) and (g)]. However, most electronic DACs create a zeroth-order hold waveform, (c), instead of an impulse train. The spectrum of the zeroth-order hold is equal to the spectrum of the impulse train multiplied by the sinc function shown in (d). To convert the zeroth-order hold into the reconstructed signal, the analog filter must remove all frequencies above

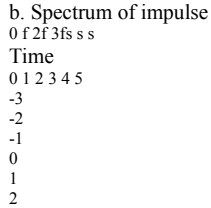
the Nyquist rate, *and* correct for the sinc, as shown in (e).



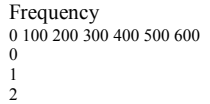
a. Impulse train



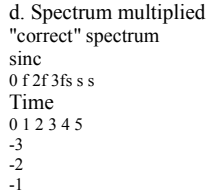
b. Spectrum of impulse train



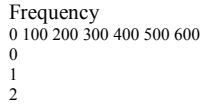
c. Zeroth-order hold



d. Spectrum multiplied by sinc

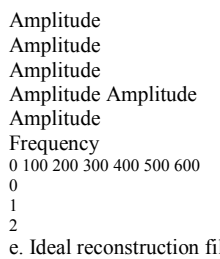


f. Reconstructed analog signal

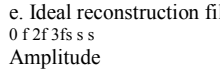


g. Reconstructed spectrum

Time Domain Frequency Domain



e. Ideal reconstruction filter



Amplitude

Digital
Processing ADC DAC Analog
Filter
Analog
Filter
Analog
Input

FIGURE 3-7
Analog electronic filters used to comply with the sampling theorem. The electronic filter placed before an ADC is called an *antialias filter*. It is used to remove frequency components above one-half of the sampling rate that would alias during the sampling. The electronic filter placed after a DAC is called a *reconstruction filter*. It also eliminates frequencies above the Nyquist rate, and may include a correction for the zeroth-order hold.

response, (3) use a fancy *multirate* technique described later in this chapter, or (4) make the correction in software before the DAC (see Chapter 24).

Before leaving this section on sampling, we need to dispel a common myth about analog versus digital signals. As this chapter has shown, the amount of information carried in a digital signal is limited in two ways: First, the number of bits per sample limits the resolution of the *dependent* variable. That is, small changes in the signal's amplitude may be lost in the quantization noise. Second, the sampling rate limits the resolution of the *independent variable*, i.e., closely spaced events in the analog signal may be lost between the samples. This is another way of saying that frequencies above one-half the sampling rate are lost.

Here is the myth: "Since analog signals use continuous parameters, they have infinitely good resolution in both the independent and the dependent variables." Not true! Analog signals are limited by the same two problems as digital signals: *noise* and *bandwidth* (the highest frequency allowed in the signal). The noise in an analog signal limits the measurement of the waveform's amplitude, just as quantization noise does in a digital signal. Likewise, the ability to separate closely spaced events in an analog signal depends on the highest frequency allowed in the waveform. To understand this, imagine an analog signal containing two closely spaced pulses. If we place the signal through a low-pass filter (removing the high frequencies), the pulses will blur into a single blob. For instance, an analog signal formed from frequencies between DC and 10 kHz will have *exactly* the same resolution as a digital signal sampled at 20 kHz. It must, since the sampling theorem guarantees that the two contain the same information.

Analog Filters for Data Conversion

Figure 3-7 shows a block diagram of a DSP system, *as the sampling theorem dictates it should be*. Before encountering the analog-to-digital converter,

the input signal is processed with an electronic low-pass filter to remove all frequencies above the Nyquist frequency (one-half the sampling rate). This is done to prevent aliasing during sampling, and is correspondingly called an **antialias filter**. On the other end, the digitized signal is passed through a digital-to-analog converter and another low-pass filter set to the Nyquist frequency. This output filter is called a **reconstruction filter**, and may include the previously described zeroth-order-hold frequency boost. Unfortunately, there is a serious problem with this simple model: the limitations of electronic filters can be as bad as the problems they are trying to prevent.

If your main interest is in software, you are probably thinking that you don't

need to read this section. *Wrong*! Even if you have vowed never to touch an oscilloscope, an understanding of the properties of analog filters is important for successful DSP. First, the characteristics of every digitized signal you encounter will depend on what type of antialias filter was used when it was acquired. If you don't understand the nature of the antialias filter, you cannot understand the nature of the digital signal. Second, the future of DSP is to replace *hardware* with *software*. For example, the *multirate* techniques presented later in this chapter reduce the need for antialias and reconstruction filters by fancy software tricks. If you don't understand the hardware, you cannot design software to replace it. Third, much of DSP is related to digital filter design. A common strategy is to start with an equivalent *analog filter*, and convert it into software. Later chapters assume you have a basic knowledge of analog filter techniques.

Three types of analog filters are commonly used: **Chebyshev**, **Butterworth**, and **Bessel** (also called a Thompson filter). Each of these is designed to optimize a different performance parameter. The complexity of each filter can be adjusted by selecting the number of **poles** and **zeros**, mathematical terms that will be discussed in later chapters. The more poles in a filter, the more electronics it requires, and the better it performs. Each of these names describe what the filter *does*, not a particular arrangement of resistors and capacitors. For example, a six pole Bessel filter can be implemented by many different types of circuits, all of which have the same overall characteristics. For DSP purposes, the characteristics of these filters are more important than how they are constructed. Nevertheless, we will start with a short segment on the electronic design of these filters to provide an overall framework.

Figure 3-8 shows a common building block for analog filter design, the modified Sallen-Key circuit. This is named after the authors of a 1950s paper describing the technique. The circuit shown is a two pole low-pass filter that can be configured as any of the three basic types. Table 3-1 provides the necessary information to select the appropriate resistors and capacitors. For example, to design a 1 kHz, 2 pole Butterworth filter, Table 3-1 provides the parameters: $k_1 = 0.1592$ and $k_2 = 0.586$. Arbitrarily selecting $R_1 = 10K$ and $C = 0.01uF$ (common values for op amp circuits), R and R_f can be calculated as 15.95K and 5.86K, respectively. Rounding these last two values to the nearest 1% standard resistors, results in $R = 15.8K$ and $R_f = 5.90K$ All of the components should be 1% precision or better.

TABLE 3-1

Parameters for designing Bessel, Butterworth, and Chebyshev (6% ripple) filters.

	Bessel		**Butterworth**		**Chebyshev**	
# poles	k_1	k_2	k_1	k_2	k_1	k_2
2	stage 1	0.1251	0.268	0.1592	0.586	0.1293	0.842
4	stage 1	0.1111	0.084	0.1592	0.152	0.2666	0.582
	stage 2	0.0991	0.759	0.1592	1.235	0.1544	1.660
6	stage 1	0.0990	0.040	0.1592	0.068	0.4019	0.537
	stage 2	0.0941	0.364	0.1592	0.586	0.2072	1.448
	stage 3	0.0834	1.023	0.1592	1.483	0.1574	1.846
8	stage 1	0.0894	0.024	0.1592	0.038	0.5359	0.522
	stage 2	0.0867	0.213	0.1592	0.337	0.2657	1.379
	stage 3	0.0814	0.593	0.1592	0.889	0.1848	1.711
	stage 4	0.0726	1.184	0.1592	1.610	0.1582	1.913

FIGURE 3-8

The modified Sallen-Key circuit, a building block for active filter design. The circuit shown implements a 2 pole low-pass filter. Higher order filters (more poles) can be

formed by cascading stages. Find k_1 and k_2 from Table 3-1, arbitrarily select R_1 and C (try 10K and 0.01µF), and then calculate R and R_f from the equations in the figure. The parameter, f_c, is the cutoff frequency of the filter, in hertz.

Rf
R1
C
C
R R
R'
k_1
$C f_c$
R_f
$' R_1 k_2$
402S
10K
0.01µF
0.01µF
10K 10K
3.65K
10K
0.01µF
0.01µF
9.53K 9.53K
10.2K
10K
0.01µF
0.01µF
8.25K 8.25K
$k_1 = 0.0990$
$k_2 = 0.040$
stage 1
$k_1 = 0.0941$
$k_2 = 0.364$
stage 2
$k_1 = 0.0834$
$k_2 = 1.023$
stage 3

FIGURE 3-9
A six pole Bessel filter formed by cascading three Sallen-Key circuits. This is a low-pass filter with a cutoff frequency of 1 kHz.

The particular op amp used isn't critical, as long as the unity gain frequency is more than 30 to 100 times higher than the filter's cutoff frequency. This is an easy requirement as long as the filter's cutoff frequency is below about 100 kHz.

Four, six, and eight pole filters are formed by cascading 2,3, and 4 of these circuits, respectively. For example, Fig. 3-9 shows the schematic of a 6 pole

time
low f
high f
time time
high R
low R
time
Resistor-Capacitor
Switched Capacitor
R
C
C C/100
f

FIGURE 3-10
Switched capacitor filter operation. Switched capacitor filters use switches and capacitors to mimic

resistors. As shown by the equivalent step responses, two capacitors and one switch can perform the same function as a resistor-capacitor network.
voltage
voltage voltage
voltage
Bessel filter created by cascading three stages. Each stage has different values for k_1 and k_2 as provided by Table 3-1, resulting in different resistors and capacitors being used. Need a high-pass filter? Simply swap the R and C components in the circuits (leaving R_f and R_1 alone).
This type of circuit is very common for small quantity manufacturing and R&D applications; however, serious production requires the filter to be made as an *integrated circuit*. The problem is, it is difficult to make resistors directly in silicon. The answer is the **switched capacitor filter**. Figure 3-10 illustrates its operation by comparing it to a simple RC network. If a step function is fed into an RC low-pass filter, the output rises exponentially until it matches the input. The voltage on the capacitor doesn't change instantaneously, because the resistor restricts the flow of electrical charge.
The switched capacitor filter operates by replacing the basic resistorcapacitor network with two capacitors and an electronic switch. The newly added capacitor is much smaller in value than the already existing capacitor, say, 1% of its value. The switch alternately connects the small capacitor between the input and the output at a very high frequency, typically 100 times faster than the cutoff frequency of the filter. When the switch is connected to the input, the small capacitor rapidly charges to whatever voltage is presently on the input. When the switch is connected to the output, the charge on the small capacitor is transferred to the large capacitor. In a resistor, the rate of charge transfer is determined by its resistance. In a switched capacitor circuit, the rate of charge transfer is determined by the value of the small capacitor *and* by the switching frequency. This results in a very useful feature of switched capacitor

filters: *the cutoff frequency of the filter is directly proportional to the clock frequency used to drive the switches*. This makes the switched capacitor filter ideal for data acquisition systems that operate with more than one sampling rate. These are easy-to-use devices; pay ten bucks and have the performance of an eight pole filter inside a single 8 pin IC.
Now for the important part: the characteristics of the three classic filter types. The first performance parameter we want to explore is **cutoff frequency sharpness**. A low-pass filter is designed to block all frequencies above the cutoff frequency (the **stopband**), while passing all frequencies below (the **passband**). Figure 3-11 shows the frequency response of these three filters on a logarithmic (dB) scale. These graphs are shown for filters with a one hertz cutoff frequency, but they can be directly scaled to whatever cutoff frequency you need to use. How do these filters rate? The Chebyshev is clearly the best, the Butterworth is worse, and the Bessel is absolutely ghastly! As you probably surmised, this is what the Chebyshev is designed to do, **roll-off** (drop in amplitude) as rapidly as possible.
Unfortunately, even an 8 pole Chebyshev isn't as good as you would like for an antialias filter. For example, imagine a 12 bit system sampling at 10,000 samples per second. The sampling theorem dictates that any frequency above 5 kHz will be aliased, something you want to avoid. With a little guess work, you decide that all frequencies above 5 kHz must be reduced in amplitude by a factor of 100, insuring that any aliased frequencies will have an amplitude of less than one percent. Looking at Fig. 3-11c, you find that an 8 pole

Chebyshev filter, with a cutoff frequency of 1 hertz, doesn't reach an attenuation (signal reduction) of 100 until about 1.35 hertz. Scaling this to the example, the filter's cutoff frequency must be set to 3.7 kHz so that everything above 5 kHz will have the required attenuation. This results in the frequency band between 3.7 kHz and 5 kHz being wasted on the inadequate roll-off of the analog filter.

A subtle point: the attenuation factor of 100 in this example is probably sufficient even though there are 4096 steps in 12 bits. From Fig. 3-4, 5100 hertz will alias to 4900 hertz, 6000 hertz will alias to 4000 hertz, etc. You don't care what the amplitudes of the signals between 5000 and 6300 hertz are, because they alias into the unusable region between 3700 hertz and 5000 hertz. In order for a frequency to alias into the filter's passband (0 to 3.7 kHz), it must be greater than 6300 hertz, or 1.7 times the filter's cutoff frequency of 3700 hertz. As shown in Fig. 3-11c, the attenuation provided by an 8 pole Chebyshev filter at 1.7 times the cutoff frequency is about 1300, much more adequate than the 100 we started the analysis with. The moral to this story: *In most systems, the frequency band between about 0.4 and 0.5 of the sampling frequency is an unusable wasteland of filter roll-off and aliased signals*. This is a direct result of the limitations of analog filters.

The frequency response of the perfect low-pass filter is *flat* across the entire passband. All of the filters look great in this respect in Fig. 3-11, but only because the vertical axis is displayed on a *logarithmic* scale. Another story is told when the graphs are converted to a *linear* vertical scale, as is shown

Frequency (hertz)
0 1 2 3 4 5
0.0001
0.001
0.01
0.1
1
10
8
4
2 pole
a. Bessel
ideal
Frequency (hertz)
0 1 2 3 4 5
0
0.2
0.4
0.6
0.8
1
1.2
1.4
1.6
8
4
2 pole
a. Bessel
ideal
Frequency (hertz)
0 1 2 3 4 5
0.0001
0.001
0.01
0.1
1
10
8
4
2 pole
b. Butterworth
Frequency (hertz)
0 1 2 3 4 5
0
0.2
0.4
0.6

0.8
1
1.2
1.4
1.6
8
4
2 pole
b. Butterworth
Frequency (hertz)
0 1 2 3 4 5
0.0001
0.001
0.01
0.1
1
10
8
4
2 pole
c. Chebyshev (6% ripple)
Frequency (hertz)
0 1 2 3 4 5
0
0.2
0.4
0.6
0.8
1
1.2
1.4
1.6
4
2 pole
8
c. Chebyshev (6% ripple)
Linear scale
Amplitude Amplitude
Amplitude Amplitude
Amplitude
Log scale
Amplitude
FIGURE 3-12
Frequency response of the three filters on a *linear* scale. The Butterworth filter provides the flattest passband.
FIGURE 3-11
Frequency response of the three filters on a *logarithmic* scale. The Chebyshev filter has the sharpest roll-off.
in Fig. 3-12. **Passband ripple** can now be seen in the Chebyshev filter (wavy variations in the amplitude of the passed frequencies). In fact, the Chebyshev filter obtains its excellent roll-off by *allowing* this passband ripple. When more passband ripple is allowed in a filter, a faster roll-off

Time (seconds)
0 1 2 3 4
0.0
0.2
0.4
0.6
0.8
1.0
1.2
1.4
1.6
8 pole
4
2
a. Bessel
Time (seconds)
0 1 2 3 4
0.0
0.2
0.4
0.6

0.8
1.0
1.2
1.4
1.6
8 pole
2
b. Butterworth
4
FIGURE 3-13
Step response of the three filters. The times shown on the horizontal axis correspond to a one hertz cutoff frequency. The Bessel is the optimum filter when overshoot and ringing must be minimized.
Time (seconds)
0 1 2 3 4
0.0
0.2
0.4
0.6
0.8
1.0
1.2
1.4
1.6
8 pole
4
2
c. Chebyshev (6% ripple)
Amplitude Amplitude
Amplitude
can be achieved. All the Chebyshev filters designed by using Table 3-1 have a passband ripple of about 6% (0.5 dB), a good compromise, and a common choice. A similar design, the **elliptic filter**, allows ripple in both the passband *and* the stopband. Although harder to design, elliptic filters can achieve an even better tradeoff between roll-off and passband ripple.

In comparison, the Butterworth filter is optimized to provide the sharpest rolloff possible *without* allowing ripple in the passband. It is commonly called the *maximally flat filter*, and is identical to a Chebyshev designed for zero passband ripple. The Bessel filter has no ripple in the passband, but the rolloff is far worse than the Butterworth.

The last parameter to evaluate is the **step response**, how the filter responds when the input rapidly changes from one value to another. Figure 3-13 shows the step response of each of the three filters. The horizontal axis is shown for filters with a 1 hertz cutoff frequency, but can be scaled (inversely) for higher cutoff frequencies. For example, a 1000 hertz cutoff frequency would show a step response in *milliseconds*, rather than *seconds*. The Butterworth and Chebyshev filters **overshoot** and show **ringing** (oscillations that slowly decreasing in amplitude). In comparison, the Bessel filter has neither of these nasty problems.

Time
0 100 200 300 400 500
-0.5
0.0
0.5
1.0
1.5
a. Pulse waveform
Time
0 100 200 300 400 500
-0.5
0.0
0.5
1.0
1.5
b. After Bessel filter

FIGURE 3-14
Pulse response of the Bessel and Chebyshev filters. A key property of the Bessel filter is that the rising and falling edges in the filter's output looking similar. In the jargon of the field, this is called *linear phase*. Figure (b) shows the result of passing the pulse waveform in (a) through a 4 pole Bessel filter. Both edges are smoothed in a similar manner. Figure (c) shows the result of passing (a) through a 4 pole Chebyshev filter. The left edge overshoots on the *top*, while the right edge overshoots on the *bottom*. Many applications cannot tolerate this distortion.

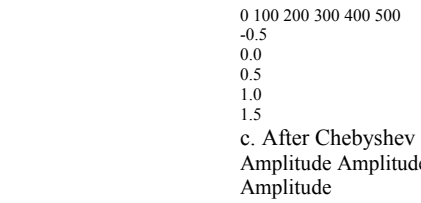


Figure 3-14 further illustrates this very favorable characteristic of the Bessel filter. Figure (a) shows a pulse waveform, which can be viewed as a rising step followed by a falling step. Figures (b) and (c) show how this waveform would appear after Bessel and Chebyshev filters, respectively. If this were a video signal, for instance, the distortion introduced by the Chebyshev filter would be devastating! The overshoot would change the brightness of the *edges* of objects compared to their *centers*. Worse yet, the left side of objects would look bright, while the right side of objects would look dark. Many applications cannot tolerate poor performance in the step response. This is where the Bessel filter shines; no overshoot and symmetrical edges.

Selecting The Antialias Filter

Table 3-2 summarizes the characteristics of these three filters, showing how each optimizes a particular parameter at the expense of everything else. The Chebyshev optimizes the *roll-off*, the Butterworth optimizes the *passband flatness*, and the Bessel optimizes the *step response*.

The selection of the antialias filter depends almost entirely on one issue: *how information is represented in the signals you intend to process*. While

TABLE 3-2
Characteristics of the three classic filters. The Bessel filter provides the best step response, making it the choice for time domain encoded signals. The Chebyshev and Butterworth filters are used to eliminate frequencies in the stopband, making them ideal for frequency domain encoded signals. Values in this table are in the units of *seconds* and *hertz*, for a one hertz cutoff frequency.

	Step Response			Frequency Response	
	Voltage gain at DC	**Overshoot**	**Time to settle to 1%**	**Time to settle to 0.1%**	**Ripple in passband**	**Frequency for x100 attenuation**	**Frequency for x1000 attenuation**

Bessel
2 pole 1.27 0.4% 0.60 1.12 0% 12.74 40.4
4 pole 1.91 0.9% 0.66 1.20 0% 4.74 8.45
6 pole 2.87 0.7% 0.74 1.18 0% 3.65 5.43
8 pole 4.32 0.4% 0.80 1.16 0% 3.35 4.53
Butterworth
2 pole 1.59 4.3% 1.06 1.66 0% 10.0 31.6
4 pole 2.58 10.9% 1.68 2.74 0% 3.17 5.62
6 pole 4.21 14.3% 2.74 3.92 0% 2.16 3.17
8 pole 6.84 16.4% 3.50 5.12 0% 1.78 2.38
Chebyshev
2 pole 1.84 10.8% 1.10 1.62 6% 12.33 38.9
4 pole 4.21 18.2% 3.04 5.42 6% 2.59 4.47
6 pole 10.71 21.3% 5.86 10.4 6% 1.63 2.26
8 pole 28.58 23.0% 8.34 16.4 6% 1.34 1.66

there are many ways for information to be encoded in an analog waveform, only two methods are common, **time domain encoding**, and **frequency domain encoding**. The difference between these two is critical in DSP, and will be a reoccurring theme throughout this book.

In *frequency domain encoding*, the information is contained in *sinusoidal waves* that combine to form the signal. Audio signals are an excellent example of this. When a person hears speech or music, the perceived sound depends on the frequencies present, and not on the particular *shape* of the waveform. This can be shown by passing an audio signal through a circuit that changes the phase of the various sinusoids, but retains their frequency and amplitude. The resulting signal *looks* completely different on an oscilloscope, but *sounds* identical. The pertinent information has been left intact, even though the waveform has been significantly altered. Since aliasing misplaces and overlaps frequency components, it directly destroys information encoded in the frequency domain. Consequently, digitization of these signals usually involves an antialias filter with a sharp cutoff, such as a Chebyshev, Elliptic, or Butterworth. What about the nasty step response of these filters? It doesn't matter; the encoded information isn't affected by this type of distortion.

In contrast, *time domain encoding* uses the *shape of the waveform* to store information. For example, physicians can monitor the electrical activity of a

person's heart by attaching electrodes to their chest and arms (an electrocardiogram or EKG). The *shape* of the EKG waveform provides the information being sought, such as when the various chambers contract during a heartbeat. Images are another example of this type of signal. Rather than a waveform that varies over *time*, images encode information in the shape of a waveform that varies over *distance*. Pictures are formed from regions of brightness and color, and how they relate to other regions of brightness and color. You don't look at the *Mona Lisa* and say, *"My, what an interesting collection of sinusoids."*

Here's the problem: The sampling theorem is an analysis of what happens in the frequency domain during digitization. This makes it ideal to under-stand the analog-to-digital conversion of signals having their information encoded in the frequency domain. However, the sampling theorem is little help in understanding how time domain encoded signals should be digitized. Let's take a closer look.

Figure 3-15 illustrates the choices for digitizing a time domain encoded signal. Figure (a) is an example analog signal to be digitized. In this case, the information we want to capture is the *shape* of the rectangular pulses. A short burst of a high frequency sine wave is also included in this example signal.

This represents wideband noise, interference, and similar junk that always appears on analog signals. The other figures show how the digitized signal would appear with different antialias filter options: a Chebyshev filter, a Bessel filter, and no filter.

It is important to understand that *none* of these options will allow the original signal to be reconstructed from the sampled data. This is because the original signal inherently contains frequency components greater than one-half of the sampling rate. Since these frequencies cannot exist in the digitized signal, the reconstructed signal cannot contain them either. These high frequencies result from two sources: (1) noise and interference, which you would like to eliminate, and (2) sharp edges in the waveform, which probably contain information you want to retain.

The Chebyshev filter, shown in (b), attacks the problem by aggressively removing all high frequency components. This results in a filtered analog signal that *can* be sampled and later perfectly reconstructed. However, the reconstructed analog signal is identical to the *filtered signal*, not the *original signal*. Although nothing is lost in sampling, the waveform has been severely distorted by the antialias filter. As shown in (b), the cure is worse than the disease! Don't do it!

The Bessel filter, (c), is designed for just this problem. Its output closely resembles the original waveform, with only a gentle rounding of the edges. By adjusting the filter's cutoff frequency, the smoothness of the edges can be traded for elimination of high frequency components in the signal. Using more poles in the filter allows a *better* tradeoff between these two parameters. A common guideline is to set the cutoff frequency at about one-quarter of the sampling frequency. This results in about two samples

along the rising portion of each edge. Notice that both the Bessel and the Chebyshev filter have removed the burst of high frequency noise present in the original signal.

The last choice is to use no antialias filter at all, as is shown in (d). This has the strong advantage that the value of each sample is *identical* to the value of the original analog signal. In other words, it has perfect edge sharpness; a change in the original signal is immediately mirrored in the digital data. The disadvantage is that aliasing can distort the signal. This takes two different forms. First, high frequency interference and noise, such as the example sinusoidal burst, will turn into meaningless samples, as shown in (d). That is, any high frequency noise present in the analog signal will appear as aliased noise in the digital signal. In a more general sense, this is not a problem of the sampling, but a problem of the upstream analog electronics. It is not the ADC's purpose to reduce noise and interference; this is the responsibility of the analog electronics before the digitization takes place. It may turn out that a Bessel filter should be placed before the digitizer to control this problem. However, this means the filter should be viewed as part of the analog processing, not something that is being done for the sake of the digitizer.

The second manifestation of aliasing is more subtle. When an event occurs in the analog signal (such as an edge), the digital signal in (d) detects the change on the *next* sample. There is no information in the digital data to indicate what happens *between* samples. Now, compare using *no filter* with using a *Bessel filter* for this problem. For example, imagine drawing straight lines between the samples in (c). The time when this constructed line crosses one-half the amplitude of the step provides a *subsample*

estimate of when the edge occurred in the analog signal. When no filter is used, this subsample information is completely lost. You don't need a fancy theorem to evaluate how this will affect your particular situation, just a good understanding of what you plan to do with the data once is it acquired.

Multirate Data Conversion

There is a strong trend in electronics to replace *analog circuitry* with *digital algorithms*. Data conversion is an excellent example of this. Consider the design of a digital voice recorder, a system that will digitize a voice signal, store the data in digital form, and later reconstruct the signal for playback. To recreate intelligible speech, the system must capture the frequencies between about 100 and 3000 hertz. However, the analog signal produced by the microphone also contains much higher frequencies, say to 40 kHz. The brute force approach is to pass the analog signal through an eight pole low-pass Chebyshev filter at 3 kHz, and then sample at 8 kHz. On the other end, the DAC reconstructs the analog signal at 8 kHz with a zeroth order hold. Another Chebyshev filter at 3 kHz is used to produce the final voice signal.

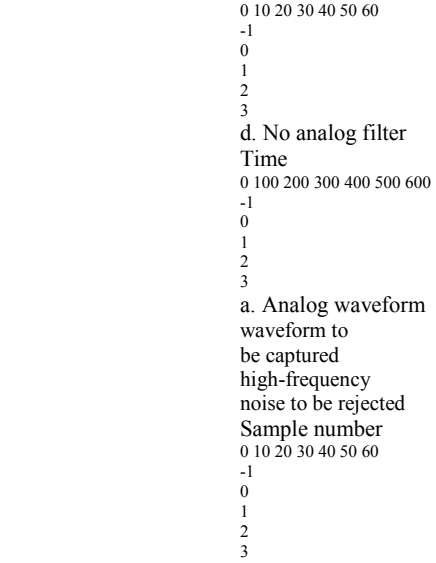
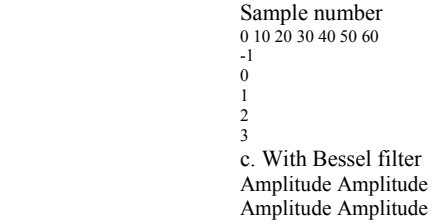


FIGURE 3-15

Three antialias filter options for time domain encoded signals. The goal is to eliminate high frequencies (that will alias during sampling), while simultaneously retaining edge sharpness (that carries information). Figure (a) shows an example analog signal containing both sharp edges and a high frequency noise burst. Figure (b) shows the digitized signal using a *Chebyshev filter*. While the high frequencies have been effectively removed, the edges have been grossly distorted. This is usually a terrible solution. The *Bessel filter*, shown in (c), provides a gentle edge smoothing while removing the high frequencies. Figure (d) shows the digitized signal using *no antialias filter*. In this case, the edges have retained perfect sharpness; however, the high frequency burst has aliased into several meaningless samples.



There are many useful benefits in sampling *faster* than this direct analysis. For

example, imagine redesigning the digital voice recorder using a 64 kHz sampling rate. The antialias filter now has an easier task: pass all freq-uencies below 3 kHz, while rejecting all frequencies above 32 kHz. A similar simplification occurs for the reconstruction filter. In short, the higher sampling rate allows the eight pole filters to be replaced with simple resistor-capacitor (RC) networks. The problem is, the digital system is now swamped with data from the higher sampling rate.

The next level of sophistication involves **multirate** techniques, using more than one sampling rate in the same system. It works like this for the digital voice recorder example. First, pass the voice signal through a simple RC low
pass filter and sample the data at 64 kHz. The resulting digital data contains the desired voice band between 100 and 3000 hertz, but also has an unusable band between 3 kHz and 32 kHz. Second, remove these unusable frequencies in *software*, by using a *digital* low-pass filter at 3 kHz. Third, resample the digital signal from 64 kHz to 8 kHz by simply discarding every seven out of eight samples, a procedure called **decimation.** The resulting digital data is equivalent to that produced by aggressive analog filtering and direct 8 kHz sampling.

Multirate techniques can also be used in the output portion of our example system. The 8 kHz data is pulled from memory and converted to a 64 kHz sampling rate, a procedure called **interpolation**. This involves placing seven samples, with a value of zero, between each of the samples obtained from memory. The resulting signal is a digital *impulse train*, containing the desired voice band between 100 and 3000 hertz, plus spectral duplications between 3 kHz and 32 kHz. Refer back to Figs. 3-6 a&b to understand why this it true. Everything above 3 kHz is then removed with a *digital* low-pass filter. After conversion to an analog signal through a DAC, a simple RC network is all that is required to produce the final voice signal.

Multirate data conversion is valuable for two reasons: (1) it replaces analog components with software, a clear economic advantage in massproduced products, and (2) it can achieve higher levels of performance in critical applications. For example, compact disc audio systems use techniques of this type to achieve the best possible sound quality. This increased performance is a result of replacing analog components (1% precision), with digital algorithms (0.0001% precision from round-off error). As discussed in upcoming chapters, digital filters outperform analog filters by *hundreds of times* in key areas.

Single Bit Data Conversion

A popular technique in telecommunications and high fidelity music reproduction is **single bit ADC and DAC**. These are multirate techniques where a higher sampling rate is traded for a lower number of bits. In the extreme, only a single bit is needed for each sample. While there are many different circuit configurations, most are based on the use of **delta modulation**. Three example circuits will be presented to give you a flavor of the field. All of these circuits are implemented in IC's, so don't worry where all of the individual transistors and op amps should go. No one is going to ask you to build one of these circuits from basic components.

Figure 3-16 shows the block diagram of a typical delta modulator. The analog input is a voice signal with an amplitude of a few volts, while the output signal is a stream of digital ones and zeros. A comparator decides which has the greater voltage, the incoming analog signal, or the voltage stored on the capacitor. This decision, in the form of a digital one or zero,

is applied to the input of the latch. At each clock pulse, typically at a few hundred kilohertz, the latch transfers whatever digital state appears on its

digital
latch comparator
charge
injector
negative
charge
injector
positive
clock
analog
input delta
modulated
output
clock
clock
FIGURE 3-16
Block diagram of a delta modulation circuit. The input voltage is compared with the voltage stored on the capacitor, resulting in a digital zero or one being applied to the input of the latch. The output of the latch is updated in synchronization with the clock, and used in a feedback loop to cause the capacitor voltage to track the input voltage.
input, to its output. This latch insures that the output is synchronized with the clock, thereby defining the sampling rate, i.e., the rate at which the 1 bit output can update itself.

A feedback loop is formed by taking the digital output and using it to drive an electronic switch. If the output is a digital *one*, the switch connects the capacitor to a *positive charge injector*. This is a very loose term for a circuit that increases the voltage on the capacitor by a fixed amount, say 1 millivolt per clock cycle. This may be nothing more than a resistor connected to a large positive voltage. If the output is a digital *zero*, the switch is connected to a *negative charge injector*. This *decreases* the voltage on the capacitor by the same fixed amount.

Figure 3-17 illustrates the signals produced by this circuit. At time equal zero, the analog input and the voltage on the capacitor both start with a voltage of zero. As shown in (a), the input signal suddenly increases to 9.5 volts on the eighth clock cycle. Since the input signal is now more positive than the voltage on the capacitor, the digital output changes to a *one*, as shown in (b). This results in the switch being connected to the positive charge injector, and the voltage on the capacitor increasing by a small amount on each clock cycle. Although an increment of 1 volt per clock cycle is shown in (a), this is only for illustration, and a value of 1 millivolt is more typical. This staircase increase in the capacitor voltage continues until it exceeds the voltage of the input signal. Here the system reached an equilibrium with the output oscillating between a digital one and zero, causing the voltage on the capacitor to oscillate between 9 volts and 10

volts. In this manner, the feedback of the circuit forces the capacitor voltage to track the voltage of the input signal. If the input signal changes very rapidly, the voltage on the capacitor changes at a constant rate until a match is obtained. This constant rate of change is called the **slew rate**, just as in other electronic devices such as op amps.

Now, consider the characteristics of the delta modulated output signal. If the analog input is *increasing* in value, the output signal will consist of more ones than zeros. Likewise, if the analog input is *decreasing* in value, the output will

consist of more zeros than ones. If the analog input is constant, the digital output will alternate between zero and one with an equal number of each. Put in more general terms, the relative number of ones versus zeros is directly proportional to the *slope* (derivative) of the analog input.

This circuit is a cheap method of transforming an analog signal into a serial stream of ones and zeros for transmission or digital storage. An especially attractive feature is that all the bits have the same meaning, unlike the conventional serial format: *start bit, LSB, ,MSB, stop bit*. The circuit at @ @ @ the receiver is identical to the feedback portion of the transmitting circuit. Just as the voltage on the capacitor in the transmitting circuit follows the analog input, so does the voltage on the capacitor in the receiving circuit. That is, the capacitor voltage shown in (a) also represents how the reconstructed signal would appear.

A critical limitation of this circuit is the unavoidable tradeoff between (1) maximum slew rate, (2) quantization size, and (3) data rate. In particular, if the maximum slew rate and quantization size are adjusted to acceptable values for voice communication, the data rate ends up in the MHz range. This is too high to be of commercial value. For instance, conventional sampling of a voice signal requires only about 64,000 bits per second.

A solution to this problem is shown in Fig. 3-18, the Continuously Variable Slope Delta (**CVSD**) modulator, a technique implemented in the Motorola MC3518 family. In this approach, the clock rate and the quantization size are set to something acceptable, say 30 kHz, and 2000 levels. This results in a terrible slew rate, which you correct with additional circuitry. In operation, a shift resister continually looks at the last four bits that the system has produced. If the circuit is in a slew rate limited condition, the last four bits will be all ones (positive slope) or all zeros (negative slope). A logic circuit detects this situation and produces an analog signal that increases the level of charge produced by the charge injectors. This boosts the slew rate by increasing the size of the voltage steps being applied to the capacitor.

An analog filter is usually placed between the logic circuitry and the charge injectors. This allows the step size to depend on how long the circuit has been in a slew limited condition. As long as the circuit is slew limited, the step size keeps getting larger and larger. This is often called a *syllabic filter*, since its characteristics depend on the average length of the syllables making up speech. With proper optimization (from the chip manufacturer's

Time (clock cycles)
0 10 20 30 40 50 60 70 80 90 100 110 120
-10
-5
0
5
10
15
a. Analog signals
signal voltage
capacitor voltage
Time (clock cycles)
0 10 20 30 40 50 60 70 80 90 100 110 120
-1
0
1
2
3
slew rate limited
b. Digital output

FIGURE 3-17

Example of signals produced by the delta modulator in Fig. 3-16. Figure (a) shows the analog input signal, and the corresponding voltage on the capacitor. Figure (b) shows the delta modulated output, a digital stream of ones and zeros.

Digital logic state Amplitude (volts)
spec sheet, not your own work), data rates of 16 to 32 kHz produce acceptable quality speech. The continually changing step size makes the digital data difficult to understand, but fortunately, you don't need to. At the receiver, the analog signal is reconstructed by incorporating a syllabic filter that is identical to the one in the transmission circuit. If the two filters are matched, little distortion results from the CVSD modulation. CVSD is probably the easiest way to digitally transmit a voice signal.

While CVSD modulation is great for encoding voice signals, it cannot be used for general purpose analog-to-digital conversion. Even if you get around the fact that the digital data is related to the *derivative* of the input signal, the *changing step size* will confuse things beyond repair. In addition, the DC level of the analog signal is usually not captured in the digital data.

The **delta-sigma** converter, shown in Fig. 3-19, eliminates these problems by cleverly combining analog electronics with DSP algorithms. Notice that the voltage on the capacitor is now being compared with ground potential. The feedback loop has also been modified so that the voltage on the

digital
latch comparator
charge
injector
negative
charge
injector
positive
clock
analog
input delta
modulated
output
4 bit
shift
register
all 1's,
all 0's
detect
syllabic
filter
CVSD Modifications
1
0
clock
clock
logic

FIGURE 3-18
CVSD modulation block diagram. A logic circuit is added to the basic delta modulator to improve the slew rate.

capacitor is *decreased* when the circuit's output is a digital *one*, and *increased* when it is a digital *zero*. As the input signal increases and decreases in voltage, it tries to raise and lower the voltage on the capacitor. This change in voltage is detected by the comparator, resulting in the charge injectors producing a *counteracting* charge to keep the capacitor at zero volts.

If the input voltage is positive, the digital output will be composed of more ones than zeros. The excess number of ones being needed to generate the *negative* charge that cancels with the *positive* input signal. Likewise, if the input voltage is negative, the digital output will be composed of more zeros than ones, providing a net positive charge injection. If the input signal is equal to zero volts, an equal number of ones and zeros will be generated in the

output, providing an overall charge injection of zero.
The relative number of ones and zeros in the output is now related to the *level* of the input voltage, not the *slope* as in the previous circuit. This is much simpler. For instance, you could form a 12 bit ADC by feeding the digital output into a counter, and counting the number of *ones* over 4096 clock cycles. A digital number of 4095 would correspond to the maximum positive input voltage. Likewise, digital number 0 would correspond to the maximum negative input voltage, and 2048 would correspond to an input voltage of zero. This also shows the origin of the name, *delta-sigma*: delta modulation followed by summation (sigma).
The ones and zeros produced by this type of delta modulator are very easy to transform back into an analog signal. All that is required is an analog lowpass filter, which might be as simple as a single RC network. The high

Digital
Latch comparator
charge
injector
negative
charge
injector
positive
clock
analog
input delta
modulated
signal
0
1
Counter Latch
digital
output
begin ADC
RESET to
end ADC
cycle
LATCH to
cycle
Digital
digital
output
low-pass
filter
Decimate
clock
clock

FIGURE 3-19
Block diagram of a delta-sigma analog-to-digital converter. In the simplest case, the pulses from a delta modulator are counted for a predetermined number of clock cycles. The output of the counter is then latched to complete the conversion. In a more sophisticated circuit, the pulses are passed through a digital low-pass filter and then resampled (decimated) to a lower sampling rate.

and low voltages corresponding to the digital ones and zeros average out to form the correct analog voltage. For example, suppose that the ones and zeros are represented by 5 volts and 0 volts, respectively. If 80% of the bits in the data stream are *ones*, and 20% are *zeros*, the output of the low-pass filter will be 4 volts.
This method of transforming the single bit data stream back into the original waveform is important for several reasons. First, it describes a slick way to replace the counter in the delta-sigma ADC circuit. Instead of simply counting the pulses from the delta modulator, the binary signal is passed through a *digital* low-pass filter, and then *decimated* to reduce the sampling rate. For

example, this procedure might start by changing each of the ones and zeros in the digital stream into a 12 bit sample; ones become a value of 4095, while zeros become a value of 0. Using a digital low-pass filter on this signal produces a digitized version of the original waveform, just as an analog lowpass filter would form an analog recreation. Decimation then reduces the sampling rate by discarding most of the samples. This results in a digital signal that is equivalent to direct sampling of the original waveform.

This approach is used in many commercial ADC's for digitizing voice and other audio signals. An example is the National Semiconductor ADC16071, which provides 16 bit analog-to-digital conversion at sampling rates up to 192 kHz. At a sampling rate of 100 kHz, the delta modulator operates with a clock frequency of 6.4 MHz. The low-pass digital filter is a 246 point FIR, such as described in Chapter 16. This removes all frequencies in the digital data above 50 kHz, ½ of the eventual sampling rate. Conceptually, this can be

viewed as forming a digital signal at 6.4 MHz, with each sample represented by 16 bits. The signal is then decimated from 6.4 MHz to 100 kHz, accomplished by deleting every 63 out of 64 samples. In actual operation, much more goes on inside of this device than described by this simple discussion.

Delta-sigma converters can also be used for digital-to-analog conversion of voice and audio signals. The digital signal is retrieved from memory, and converted into a delta modulated stream of ones and zeros. As mentioned above, this single bit signal can easily be changed into the reconstructed analog signal with a simple low-pass analog filter. As with the antialias filter, usually only a single RC network is required. This is because the majority of the filtration is handled by the high-performance digital filters.

Delta-sigma ADC's have several quirks that limit their use to specific applications. For example, it is difficult to multiplex their inputs. When the input is switched from one signal to another, proper operation is not established until the digital filter can clear itself of data from the previous signal. Deltasigma converters are also limited in another respect: you don't know exactly *when* each sample was taken. Each acquired sample is a composite of the one bit information taken over a segment of the input signal. This is not a problem for signals encoded in the frequency domain, such as audio, but it is a significant limitation for time domain encoded signals. To understand the shape of a signal's waveform, you often need to know the precise instant each sample was taken. Lastly, most of these devices are specifically designed for audio applications, and their performance specifications are quoted accordingly. For example, a 16 bit ADC used for voice signals does not necessarily mean that each sample has 16 bits of precision. Much more likely, the manufacturer is stating that *voice signals* can be digitized to 16 bits of *dynamic range*. Don't expect to get a full 16 bits of useful information from this device for general purpose data acquisition.

While these explanations and examples provide an introduction to single bit ADC and DAC, it must be emphasized that they are simplified descriptions of sophisticated DSP and integrated circuit technology. You wouldn't expect the manufacturer to tell their *competitors* all the internal workings of their chips, so don't expect them to tell *you*.

5 Linear Systems

Most DSP techniques are based on a divide-and-conquer strategy called *superposition*. The signal being processed is broken into simple components, each component is processed individually, and the results reunited. This approach has the tremendous power of breaking a single complicated problem into many easy ones. Superposition can only be used with *linear systems*, a term meaning that certain mathematical rules apply. Fortunately, most of the applications encountered in science and engineering fall into this category. This chapter presents the foundation of DSP: what it means for a system to be linear, various ways for breaking signals into simpler components, and how superposition provides a variety of signal processing techniques.

Signals and Systems

A **signal** is a description of how one parameter varies with another parameter. For instance, voltage changing over time in an electronic circuit, or brightness varying with distance in an image. A **system** is any process that produces an *output signal* in response to an *input signal*. This is illustrated by the block diagram in Fig. 5-1. Continuous systems input and output continuous signals, such as in analog electronics. Discrete systems input and output discrete signals, such as computer programs that manipulate the values stored in arrays.

Several rules are used for naming signals. These aren't always followed in DSP, but they are very common and you should memorize them. The mathematics is difficult enough without a clear notation. First, *continuous* signals use parentheses, such as: and , while *discrete* signals use $x(t)$ $y(t)$ brackets, as in: and . Second, signals use lower case letters. Upper $x[n]$ $y[n]$ case letters are reserved for the frequency domain, discussed in later chapters. Third, the name given to a signal is usually descriptive of the parameters it represents. For example, a *voltage* depending on *time* might be called: , or $v(t)$ a stock market *price* measured each *day* could be: . $p[d]$

FIGURE 5-1

Terminology for signals and systems. A system is any process that generates an output signal in response to an input signal. Continuous signals are usually represented with parentheses, while discrete signals use brackets. All signals use lower case letters, reserving the upper case for the frequency domain (presented in later chapters). Unless there is a better name available, the input signal is called: $x(t)$ or $x[n]$, while the output is called: $y(t)$ or $y[n]$.

Signals and systems are frequently discussed without knowing the exact parameters being represented. This is the same as using x and y in algebra, without assigning a physical meaning to the variables. This brings in a fourth rule for naming signals. If a more descriptive name is not available, the input signal to a discrete system is usually called: , and the output signal: . $x[n]$ $y[n]$ For continuous systems, the signals: and are used. $x(t)$ $y(t)$

There are many reasons for wanting to understand a *system*. For example, you may want to *design* a system to remove noise in an electrocardiogram, sharpen an out-of-focus image, or remove echoes in an audio recording. In other cases,

the system might have a distortion or interfering effect that you need to characterize or measure. For instance, when you speak into a telephone, you expect the other person to hear something that resembles your voice. Unfortunately, the input signal to a transmission line is seldom identical to the output signal. If you understand how the transmission line (the system) is changing the signal, maybe you can compensate for its effect. In still other cases, the system may represent some physical process that you want to study or analyze. Radar and sonar are good examples of this. These methods operate by comparing the transmitted and reflected signals to find the characteristics of a remote object. In terms of system theory, the problem is to find the system that changes the transmitted signal into the received signal.
At first glance, it may seem an overwhelming task to understand all of the possible systems in the world. Fortunately, most useful systems fall into a category called **linear systems**. This fact is extremely important. *Without* the linear system concept, we would be forced to examine the individual

System
System
x[n] y[n]
IF
THEN
k x[n] *k* y[n]
FIGURE 5-2
Definition of homogeneity. A system is said to be *homogeneous* if an amplitude change in the input results in an identical amplitude change in the output. That is, if $x[n]$ results in $y[n]$, then $kx[n]$ results in $ky[n]$, for any signal, $x[n]$, and any constant, k.
characteristics of many unrelated systems. *With* this approach, we can focus on the traits of the linear system category as a whole. Our first task is to identify what properties make a system linear, and how they fit into the everyday notion of electronics, software, and other signal processing systems.

Requirements for Linearity

A system is called *linear* if it has two mathematical properties: **homogeneity** (hÇma-gen-~-ity) and **additivity**. If you can show that a system has both properties, then you have proven that the system is linear. Likewise, if you can show that a system doesn't have one or both properties, you have proven that it isn't linear. A third property, **shift invariance**, is not a strict requirement for linearity, but it is a mandatory property for most DSP techniques. When you see the term *linear system* used in DSP, you should assume it includes *shift invariance* unless you have reason to believe otherwise. These three properties form the mathematics of how linear system theory is defined and used. Later in this chapter we will look at more intuitive ways of understanding linearity. For now, let's go through these formal mathematical properties.
As illustrated in Fig. 5-2, homogeneity means that a change in the input signal's amplitude results in a corresponding change in the output signal's amplitude. In mathematical terms, if an input signal of results in an output signal of $x[n]$, an input of results in an output of , for any input signal and $y[n]$ $k\,x[n]$ $k\,y[n]$ constant, k.

System
System

IF
THEN
System
AND IF
$x_1[n]$ $y_1[n]$
$x_2[n]$ $y_2[n]$
$y_1[n]+y_2[n]$ $x_1[n]+x_2[n]$
FIGURE 5-3
Definition of additivity. A system is said to be *additive* if added signals pass through it without interacting. Formally, if $x_1[n]$ results in $y_1[n]$, and if $x_2[n]$ results in $y_2[n]$, then $x_1[n]+x_2[n]$ results in $y_1[n]+y_2[n]$.
A simple resistor provides a good example of both homogenous and nonhomogeneous systems. If the input to the system is the voltage across the resistor, , and the output from the system is the current through the resistor, $v(t)$, the system is homogeneous. Ohm's law guarantees this; if the voltage is $i(t)$ increased or decreased, there will be a corresponding increase or decrease in the current. Now, consider another system where the input signal is the voltage across the resistor, , but the output signal is the power being dissipated in $v(t)$ the resistor, . Since power is proportional to the square of the voltage, if $p(t)$ the input signal is increased by a factor of *two*, the output signal is increase by a factor of *four*. This system is not homogeneous and therefore cannot be linear.
The property of additivity is illustrated in Fig. 5-3. Consider a system where an input of produces an output of . Further suppose that a different $x_1[n]$ $y_1[n]$ input, , produces another output, . The system is said to be *additive*, $x_2[n]$ $y_2[n]$ if an input of results in an output of , for all possible $x_1[n]$ % $x_2[n]$ $y_1[n]$ % $y_2[n]$ input signals. In words, signals added at the input produce signals that are added at the output.

System
System
x[n] y[n]
x[n+*s*] y[n+*s*]
IF
THEN
FIGURE 5-4
Definition of shift invariance. A system is said to be *shift invariant* if a shift in the input signal causes an identical shift in the output signal. In mathematical terms, if $x[n]$ produces $y[n]$, then $x[n+s]$ produces $y[n+s]$, for any signal, $x[n]$, and any constant, s.
The important point is that *added* signals pass through the system without interacting. As an example, think about a telephone conversation with your Aunt Edna and Uncle Bernie. Aunt Edna begins a rather lengthy story about how well her radishes are doing this year. In the background, Uncle Bernie is yelling at the dog for having an accident in his favorite chair. The two voice signals are added and electronically transmitted through the telephone network. Since this system is additive, the sound you hear is the sum of the two voices as they would sound if transmitted individually. You hear *Edna* and *Bernie*, not the creature, *Ednabernie*.
A good example of a *nonadditive* circuit is the mixer stage in a radio

transmitter. Two signals are present: an audio signal that contains the voice or music, and a carrier wave that can propagate through space when applied to an antenna. The two signals are added and applied to a nonlinearity, such as a pn junction diode. This results in the signals *merging* to form a third signal, a modulated radio wave capable of carrying the information over great distances.

As shown in Fig. 5-4, shift invariance means that a shift in the input signal will result in nothing more than an identical shift in the output signal. In more formal terms, if an input signal of results in an output of , an input $x[n]$ $y[n]$ signal of results in an output of , for any input signal and any $x[n\% s]$ $y[n\% s]$ constant, s. Pay particular notice to how the mathematics of this shift is written, it will be used in upcoming chapters. By adding a constant, s, to the independent variable, n, the waveform can be advanced or retarded in the horizontal direction. For example, when , the signal is shifted *left* by two s' 2 samples; when , the signal is shifted *right* by two samples. s' &2

Shift invariance is important because it means the characteristics of the system do not change with time (or whatever the independent variable happens to be). If a *blip* in the input causes a *blop* in the output, you can be assured that another *blip* will cause an identical *blop*. Most of the systems you encounter will be shift invariant. This is fortunate, because it is difficult to deal with systems that change their characteristics while in operation. For example, imagine that you have designed a digital filter to compensate for the degrading effects of a telephone transmission line. Your filter makes the voices sound more natural and easier to understand. Much to your surprise, along comes winter and you find the characteristics of the telephone line have changed with temperature. Your compensation filter is now mismatched and doesn't work especially well. This situation may require a more sophisticated algorithm that can *adapt* to changing conditions.

Why do homogeneity and additivity play a critical role in linearity, while shift invariance is something on the side? This is because linearity is a very broad concept, encompassing much more than just signals and systems. For example, consider a farmer selling oranges for $2 per crate and apples for $5 per crate. If the farmer sells only oranges, he will receive $20 for 10 crates, and $40 for 20 crates, making the exchange *homogenous*. If he sells 20 crates of oranges and 10 crates of apples, the farmer will receive: . This 20×$2 % 10×$5 ' $90 is the same amount as if the two had been sold individually, making the transaction *additive*. Being both homogenous and additive, this sale of goods is a linear process. However, since there are no signals involved, this is not a *system*, and *shift invariance* has no meaning. Shift invariance can be thought of as an additional aspect of linearity needed when signals and systems are involved.

Static Linearity and Sinusoidal Fidelity

Homogeneity, additivity, and shift invariance are important because they provide the mathematical basis for defining linear systems. Unfortunately, these properties alone don't provide most scientists and engineers with an intuitive feeling of what linear systems are about. The properties of **static linearity** and **sinusoidal fidelity** are often of help here. These are not especially important from a mathematical standpoint, but relate to how humans think about and understand linear systems. You should pay special attention to this section.

Static linearity defines how a linear system reacts when the signals aren't

changing, i.e., when they are *DC* or *static*. The static response of a linear system is very simple: *the output is the input multiplied by a constant*. That is, a graph of the possible input values plotted against the corresponding output values is a straight line that passes through the origin. This is shown in Fig. 5-5 for two common linear systems: Ohm's law for resistors, and Hooke's law for springs. For comparison, Fig. 5-6 shows the static relationship for two nonlinear systems: a pn junction diode, and the magnetic properties of iron.

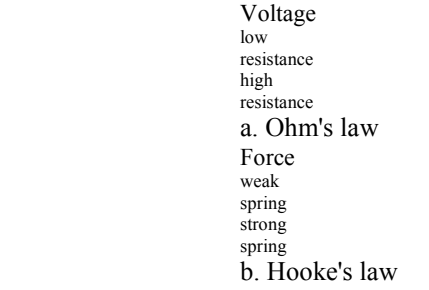


FIGURE 5-5
Two examples of static linearity. In (a), Ohm's law: the current through a resistor is equal to the voltage across the resistor divided by the resistance. In (b), Hooke's law: The elongation of a spring is equal to the applied force multiplied by the spring stiffness coefficient.

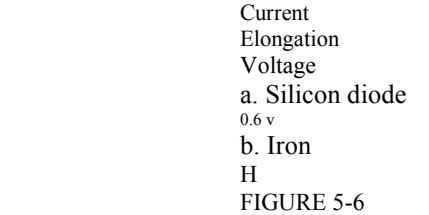


FIGURE 5-6
Two examples of DC nonlinearity. In (a), a silicon diode has an exponential relationship between voltage and current. In (b), the relationship between magnetic intensity, H, and flux density, B, in iron depends on the history of the sample, a behavior called *hysteresis*.

All linear systems have the property of *static linearity*. The opposite is usually true, but not always. There are systems that show static linearity, but are not linear with respect to changing signals. However, a very common class of systems can be completely understood with static linearity alone. In these systems it doesn't matter if the input signal is static or changing. These are called **memoryless** systems, because the output depends only on the present state of the input, and not on its history. For example, the instantaneous current in a resistor depends only on the instantaneous voltage across it, and not on how the signals came to be the value they are. If a system has static linearity, and is memoryless, then the system must be linear. This provides an important way to understand (and prove) the linearity of these simple systems.

An important characteristic of linear systems is how they behave with sinusoids, a property we will call **sinusoidal fidelity**: *If the input to a linear system is a sinusoidal wave, the output will also be a sinusoidal wave, and at exactly the same frequency as the input.* Sinusoids are the only waveform that have this property. For instance, there is no reason to expect that a square wave entering a linear system will produce a square wave on the output. Although a sinusoid on the input guarantees a sinusoid on the output, the two may be different in *amplitude* and *phase*. This

should be familiar from your knowledge of electronics: a circuit can be described by its *frequency response*, graphs of how the circuit's gain and phase vary with frequency.

Now for the reverse question: If a system always produces a sinusoidal output in response to a sinusoidal input, is the system guaranteed to be linear? The answer is no, but the exceptions are rare and usually obvious. For example, imagine an evil demon hiding inside a system, with the goal of trying to mislead you. The demon has an oscilloscope to observe the input signal, and a sine wave generator to produce an output signal. When you feed a sine wave into the input, the demon quickly measures the frequency and adjusts his signal generator to produce a corresponding output. Of course, this system is not linear, because it is not additive. To show this, place the sum of two sine waves into the system. The demon can only respond with a single sine wave for the output. This example is not as contrived as you might think; *phase lock loops* operate in much this way.

To get a better feeling for linearity, think about a technician trying to determine if an electronic device is linear. The technician would attach a sine wave generator to the input of the device, and an oscilloscope to the output. With a sine wave input, the technician would look to see if the output is also a sine wave. For example, the output cannot be clipped on the top or bottom, the top half cannot look different from the bottom half, there must be no distortion where the signal crosses zero, etc. Next, the technician would vary the amplitude of the input and observe the effect on the output signal. If the system is linear, the amplitude of the output must track the amplitude of the input. Lastly, the technician would vary the input signal's frequency, and verify that the output signal's frequency changes accordingly. As the frequency is changed, there will likely be amplitude and phase changes seen in the output, but these are perfectly permissible in a linear system. At some frequencies, the output may even be *zero*, that is, a sinusoid with zero amplitude. If the technician sees all these things, he will conclude that the system is linear. While this conclusion is not a rigorous mathematical proof, the level of confidence is justifiably high.

Examples of Linear and Nonlinear Systems

Table 5-1 provides examples of common linear and nonlinear systems. As you go through the lists, keep in mind the mathematician's view of linearity (*homogeneity*, *additivity*, and *shift invariance*), as well as the informal way most scientists and engineers use (*static linearity* and *sinusoidal fidelity*).

Table 5-1

Examples of linear and nonlinear systems. Formally, linear systems are defined by the properties of *homogeneity*, *additivity*, and *shift invariance*. Informally, most scientists and engineers think of linear systems in terms of *static linearity* and *sinusoidal fidelity*.

Examples of *Linear* Systems

Wave propagation such as sound and electromagnetic waves
Electrical circuits composed of resistors, capacitors, and inductors
Electronic circuits, such as amplifiers and filters
Mechanical motion from the interaction of masses, springs, and dashpots (dampeners)
Systems described by differential equations such as resistor-capacitor-inductor networks
Multiplication by a constant, that is, amplification or attenuation of the signal
Signal changes, such as echoes, resonances, and image blurring
The unity system where the output is always equal to the input
The null system where the output is always equal to the zero, regardless of the input
Differentiation and integration, and the analogous operations of *first difference* and *running sum* for discrete signals

Small perturbations in an otherwise nonlinear system, for instance, a small signal being amplified by a properly biased transistor
Convolution, a mathematical operation where each value in the output is expressed as the sum of values in the input multiplied by a set of weighing coefficients.
Recursion, a technique similar to convolution, except previously calculated values in the output are used in addition to values from the input

Examples of *Nonlinear* Systems

Systems that do not have static linearity, for instance, the voltage and power in a resistor: , the radiant energy emission of a hot object depending on its temperature: $P' V_2 R$, the intensity of light transmitted through a thickness of translucent material: $R ' kT_4$, etc. $I ' e \&"_T$
Systems that do not have sinusoidal fidelity, such as electronics circuits for: peak detection, squaring, sine wave to square wave conversion, frequency doubling, etc.
Common electronic distortion, such as clipping, crossover distortion and slewing
Multiplication of one signal by another signal, such as in amplitude modulation and automatic gain controls
Hysteresis phenomena, such as magnetic flux density versus magnetic intensity in iron, or mechanical stress versus strain in vulcanized rubber
Saturation, such as electronic amplifiers and transformers driven too hard
Systems with a threshold, for example, digital logic gates, or seismic vibrations that are strong enough to pulverize the intervening rock

x[n] y[n]
IF
THEN
x[n] y[n]
System System
A B
System System
B A

FIGURE 5-7
The commutative property for linear systems. When two or more linear systems are arranged in a cascade, the order of the systems does not affect the characteristics of the overall combination.

Special Properties of Linearity

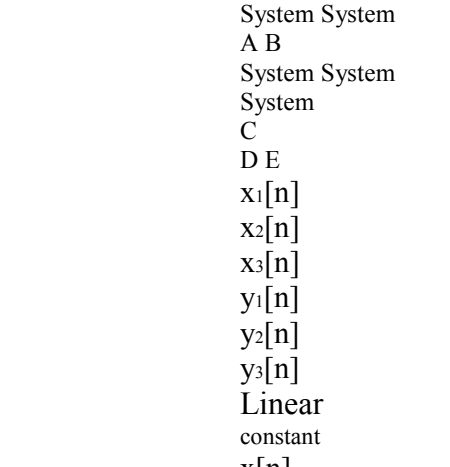
Linearity is **commutative**, a property involving the combination of two or more systems. Figure 5-10 shows the general idea. Imagine two systems combined in a **cascade**, that is, the output of one system is the input to the next. If each system is linear, then the overall combination will also be linear. The commutative property states that the order of the systems in the cascade can be rearranged without affecting the characteristics of the overall combination. You probably have used this principle in electronic circuits. For example, imagine a circuit composed of two stages, one for amplification, and one for filtering. Which is best, amplify and then filter, or filter and then amplify? If both stages are linear, the order doesn't make any difference and the overall result is the same. Keep in mind that actual electronics has *nonlinear* effects that may make the order important, for instance: interference, DC offsets, internal noise, slew rate distortion, etc.

Figure 5-8 shows the next step in linear system theory: multiple inputs and outputs. A system with multiple inputs and/or outputs will be linear if it is composed of *linear subsystems* and *additions of signals*. The complexity does not matter, only that nothing *nonlinear* is allowed inside of the system.

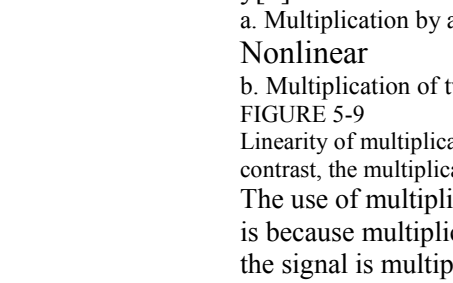
To understand what linearity means for systems with multiple inputs and/or outputs, consider the following thought experiment. Start by placing a signal

on one input while the other inputs are held at zero. This will cause the multiple outputs to respond with some pattern of signals. Next, repeat the procedure by placing another signal on a different input. Just as before, keep all of the other inputs at zero. This second input signal will result in another pattern of signals appearing on the multiple outputs. To finish the experiment, place both signals on their respective inputs simultaneously. The signals appearing on the outputs will simply be the *superposition* (sum) of the output signals produced when the input signals were applied separately.

FIGURE 5-8
Any system with multiple inputs and/or outputs will be linear if it is composed of linear systems and signal additions.



a. Multiplication by a constant



b. Multiplication of two signals

FIGURE 5-9
Linearity of multiplication. Multiplying a signal by a constant is a linear operation. In contrast, the multiplication of two signals is nonlinear.

The use of multiplication in linear systems is frequently misunderstood. This is because multiplication can be either linear or nonlinear, depending on what the signal is multiplied by. Figure 5-9 illustrates the two cases. A system that multiplies the input signal by a *constant*, is linear. This system is an amplifier or an attenuator, depending if the constant is greater or less than one, respectively. In contrast, multiplying a signal by *another signal* is nonlinear. Imagine a sinusoid multiplied by another sinusoid of a different frequency; the resulting waveform is clearly not sinusoidal.

Another commonly misunderstood situation relates to parasitic signals added in electronics, such as DC offsets and thermal noise. Is the addition of these extraneous signals linear or nonlinear? The answer depends on where the contaminating signals are viewed as originating. If they are viewed as coming from *within* the system, the process is nonlinear. This is because a sinusoidal input does not produce a pure sinusoidal output. Conversely, the extraneous signal can be viewed as *externally* entering the system on a separate input of a multiple input system. This makes the process linear, since only a signal

addition is required within the system.

decomposition
synthesis
x[n]
$x_0[n]$
$x_1[n]$
$x_2[n]$
FIGURE 5-10
Illustration of synthesis and decomposition of signals. In synthesis, two or more signals are added to form another signal. Decomposition is the opposite process, breaking one signal into two or more additive component signals.

Superposition: the Foundation of DSP

When we are dealing with linear systems, the only way signals can be combined is by *scaling* (multiplication of the signals by constants) followed by *addition*. For instance, a signal cannot be multiplied by another signal. Figure 5-10 shows an example: three signals: , , and are added to form $x_0[n]$ $x_1[n]$ $x_2[n]$ a fourth signal, . This process of combining signals through scaling and x [n] addition is called **synthesis**.

Decomposition is the inverse operation of synthesis, where a single signal is broken into two or more additive components. This is more involved than synthesis, because there are infinite possible decompositions for any given signal. For example, the numbers 15 and 25 can only be synthesized (added) into the number 40. In comparison, the number 40 can be decomposed into: or or , etc. 1% 39 2% 38 &30.5% 60% 10.5

Now we come to the heart of DSP: **superposition**, the overall strategy for understanding how signals and systems can be analyzed. Consider an input

System
x[n]
$x_0[n]$
$x_1[n]$
$x_2[n]$
$y_0[n]$
$y_1[n]$
$y_2[n]$
y[n]
System
System
The Fundamental
Concept of DSP
decomposition
synthesis
FIGURE 5-11
The fundamental concept in DSP. Any signal, such as , can be *decomposed* into x [n] a group of additive components, shown here by the signals: and . Passing $x_1[n], x_2[n], x_3[n]$ these components through a linear system produces the signals, and . $y_1[n], y_2[n], y_3[n]$ The *synthesis* (addition) of these output

signals forms , the same signal produced $y[n]$
when is passed through the system. $x[n]$
signal, called , passing through a linear system, resulting in an output $x[n]$
signal, . As illustrated in Fig. 5-11, the input signal can be decomposed $y[n]$
into a group of simpler signals: , , , etc. We will call these the $x_0[n]$ $x_1[n]$ $x_2[n]$
input signal components. Next, each input signal component is individually passed through the system, resulting in a set of **output signal components**:
, , , etc. These output signal components are then synthesized $y_0[n]$ $y_1[n]$ $y_2[n]$
into the output signal, . $y[n]$

Here is the important part: the output signal obtained by this method is *identical* to the one produced by directly passing the input signal through the system. This is a very powerful idea. Instead of trying to understanding how *complicated* signals are changed by a system, all we need to know is how *simple* signals are modified. In the jargon of signal processing, the input and output signals are viewed as a *superposition* (sum) of simpler waveforms. This is the basis of nearly all signal processing techniques.

As a simple example of how superposition is used, multiply the number 2041 by the number 4, in your head. How did you do it? You might have imagined 2041 match sticks floating in your mind, quadrupled the mental image, and started counting. Much more likely, you used superposition to simplify the problem. The number 2041 can be decomposed into: . Each of 2000 % 40 % 1 these components can be multiplied by 4 and then synthesized to find the final answer, i.e., . 8000 % 160 % 4 ' 8164

Common Decompositions

Keep in mind that the goal of this method is to replace a complicated problem with several easy ones. If the decomposition doesn't simplify the situation in some way, then nothing has been gained. There are two main ways to decompose signals in signal processing: *impulse decomposition* and *Fourier decomposition.* They are described in detail in the next several chapters. In addition, several minor decompositions are occasionally used. Here are brief descriptions of the two major decompositions, along with three of the minor ones.

Impulse Decomposition

As shown in Fig. 5-12, impulse decomposition breaks an N samples signal into N component signals, each containing N samples. Each of the component signals contains one point from the original signal, with the remainder of the values being zero. A single nonzero point in a string of zeros is called an *impulse*. Impulse decomposition is important because it allows signals to be examined one sample at a time. Similarly, systems are characterized by how they respond to impulses. By knowing how a system responds to an impulse, the system's output can be calculated for any given input. This approach is called *convolution*, and is the topic of the next two chapters.

Step Decomposition

Step decomposition, shown in Fig. 5-13, also breaks an N sample signal into N component signals, each composed of N samples. Each component signal is a *step*, that is, the first samples have a value of zero, while the last samples are some constant value. Consider the decomposition of an N point signal, , into the components: . The component $x[n]$ $x_0[n]$, $x_1[n]$, $x_2[n]$, þ$x_{N\&1}[n]$ k $_{th}$ signal, , is composed of zeros for points 0 through , while the $x_k[n]$ $k\&1$ remaining points have a value of: . For example, the $x[k]$ & $x[k\&1]$ 5 $_{th}$ component signal, , is composed of zeros for points 0 through 4, while $x_5[n]$ the remaining samples have a value of: (the difference between $x[5]$ & $x[4]$

Impulse Decomposition

Step Decomposition

x[n]
x_0[n]
x_1[n]
x_2[n]
x_{27}[n]
x[n]
x_0[n]
x_1[n]
x_2[n]
x_{27}[n]

FIGURE 5-12
Example of impulse decomposition. An *N* point signal is broken into *N* components, each consisting of a single nonzero point.

FIGURE 5-13
Example of step decomposition. An *N* point signal is broken into *N* signals, each consisting of a step function

sample 4 and 5 of the original signal). As a special case, has all of its $x_0[n]$ samples equal to . Just as impulse decomposition looks at signals one point $x[0]$ at a time, step decomposition characterizes signals by the *difference* between adjacent samples. Likewise, systems are characterized by how they respond to a *change* in the input signal.

$$x_E[n] = \frac{x[n] + x[N-n]}{2}$$

$$x_O[n] = \frac{x[n] - x[N-n]}{2}$$

EQUATION 5-1
Equations for even/odd decomposition. These equations separate a signal, , $x[n]$ into its even part, , and its odd part, $x_E[n]$. Since this decomposition is based $x_O[n]$ on circularly symmetry, the zeroth samples in the even and odd signals are calculated: , and . $x_E[0] = x[0]$ $x_O[0] = 0$ All of the signals are *N* samples long, with indexes running from 0 to *N*-1

Even/Odd Decomposition

The even/odd decomposition, shown in Fig. 5-14, breaks a signal into two component signals, one having **even symmetry** and the other having **odd symmetry**. An *N* point signal is said to have even symmetry if it is a mirror image around point . That is, sample must equal , $N/2$ $x[N/2+1]$ $x[N/2-1]$ sample must equal , etc. Similarly, odd symmetry occurs $x[N/2+2]$ $x[N/2-2]$ when the matching points have equal magnitudes but are opposite in sign, such as: , , etc. These definitions $x[N/2+1] = -x[N/2-1]$ $x[N/2+2] = -x[N/2-2]$ assume that the signal is composed of an even number of samples, and that the indexes run from 0 to . The decomposition is calculated from the $N-1$ relations:

This may seem a strange definition of left-right symmetry, since $N/2-½$ (between two samples) is the exact center of the signal, not . Likewise, $N/2$

this off-center symmetry means that sample zero needs special handling.
What's this all about?
This decomposition is part of an important concept in DSP called **circular symmetry**. It is based on viewing the *end* of the signal as connected to the *beginning* of the signal. Just as point is next to point , point $x[4]$ $x[5]$ $x[N\&1]$ is next to point . Picture a snake biting its own tail. When even and odd $x[0]$ signals are viewed in this circular manner, there are actually *two* lines of symmetry, one at point and another at point . For example, in an $x[N/2]$ $x[0]$ even signal, this symmetry around means that point equals point $x[0]$ $x[1]$, point equals point , etc. In an odd signal, point 0 and $x[N\&1]$ $x[2]$ $x[N\&2]$ point always have a value of zero. In an even signal, point 0 and point $N/2$ are equal to the corresponding points in the original signal. $N/2$
What is the motivation for viewing the last sample in a signal as being next to the first sample? There is nothing in conventional data acquisition to support this circular notion. In fact, the first and last samples generally have less in common than any other two points in the sequence. It's common sense! The missing piece to this puzzle is a DSP technique called *Fourier analysis*. The mathematics of Fourier analysis inherently views the signal as being circular, although it usually has no physical meaning in terms of where the data came from. We will look at this in more detail in Chapter 10. For now, the important thing to understand is that Eq. 5-1 provides a valid decomposition, simply because the even and odd parts can be added together to reconstruct the original signal.

Even/Odd
Decomposition
Interlaced
Decomposition
x[n]
$x_E[n]$
$x_O[n]$
x[n]
$x_E[n]$
$x_O[n]$
0 N/2 N
even symmetry
odd symmetry
even samples
odd samples
FIGURE 5-14
Example of even/odd decomposition. An N point signal is broken into two N point signals, one with even symmetry, and the other with odd symmetry.
FIGURE 5-15
Example of interlaced decomposition. An N point signal is broken into two N point signals, one with the odd samples set to zero, the other with the even samples set to zero.

Interlaced Decomposition

As shown in Fig. 5-15, the interlaced decomposition breaks the signal into two component signals, the *even sample* signal and the *odd sample* signal (not to be confused with even and odd symmetry signals). To find the even sample signal, start with the original signal and set all of the odd numbered samples to zero. To find the odd sample signal, start with the original signal and set all of the even numbered samples to zero. It's that simple.

At first glance, this decomposition might seem trivial and uninteresting. This is ironic, because the interlaced decomposition is the basis for an extremely important algorithm in DSP, the Fast Fourier Transform (FFT). The procedure for calculating the Fourier decomposition has been known for several hundred years. Unfortunately, it is frustratingly slow, often requiring minutes or hours to execute on present day computers. The FFT is a family of algorithms developed in the 1960s to reduce this computation time. The strategy is an exquisite example of DSP: reduce the signal to elementary components by repeated use of the interlace transform; calculate the Fourier decomposition of the individual components; synthesize the results into the final answer. The

results are dramatic; it is common for the speed to be improved by a factor of *hundreds* or *thousands*.

Fourier Decomposition

Fourier decomposition is very mathematical and not at all obvious. Figure 5-16 shows an example of the technique. Any N point signal can be decomposed into signals, half of them sine waves and half of them $N\%2$ cosine waves. The lowest frequency cosine wave (called in this $x_{C0}[n]$ illustration), makes *zero* complete cycles over the N samples, i.e., it is a DC signal. The next cosine components: , , and , make 1, 2, $x_{C1}[n]$ $x_{C2}[n]$ $x_{C3}[n]$ and 3 complete cycles over the N samples, respectively. This pattern holds for the remainder of the cosine waves, as well as for the sine wave components. Since the frequency of each component is fixed, the only thing that changes for different signals being decomposed is the *amplitude* of each of the sine and cosine waves.

Fourier decomposition is important for three reasons. First, a wide variety of signals are inherently created from superimposed sinusoids. Audio signals are a good example of this. Fourier decomposition provides a direct analysis of the information contained in these types of signals. Second, linear systems respond to sinusoids in a unique way: a sinusoidal input always results in a sinusoidal output. In this approach, systems are characterized by how they change the amplitude and phase of sinusoids passing through them. Since an input signal can be decomposed into sinusoids, knowing how a system will react to sinusoids allows the output of the system to be found. Third, the Fourier decomposition is the basis for a broad and powerful area of mathematics called *Fourier analysis*, and the even more advanced *Laplace* and *z-transforms*. Most cutting-edge DSP algorithms are based on some aspect of these techniques.

Why is it even possible to decompose an arbitrary signal into sine and cosine waves? How are the amplitudes of these sinusoids determined for a particular signal? What kinds of systems can be designed with this technique? These are the questions to be answered in later chapters. The details of the Fourier decomposition are too involved to be presented in this brief overview. For now, the important idea to understand is that when all of the component sinusoids are added together, the original signal is exactly reconstructed. Much more on this in Chapter 8.

Alternatives to Linearity

To appreciate the importance of linear systems, consider that there is only *one* major strategy for analyzing systems that are nonlinear. That strategy is to make the nonlinear system *resemble* a linear system. There are three common ways of doing this:

First, ignore the nonlinearity. If the nonlinearity is small enough, the system can be approximated as linear. Errors resulting from the original assumption

are tolerated as noise or simply ignored.

FIGURE 5-16
Illustration of Fourier decomposition. An *N* point signal is decomposed into *N*+2 signals, each having *N* points. Half of these signals are cosine waves, and half are sine waves. The frequencies of the sinusoids are fixed; only the amplitudes can change.

Second, keep the signals very small. Many nonlinear systems appear linear if the signals have a very small amplitude. For instance, transistors are very nonlinear over their full range of operation, but provide accurate linear amplification when the signals are kept under a few millivolts. Operational amplifiers take this idea to the extreme. By using very high open-loop gain together with negative feedback, the input signal to the op amp (i.e., the difference between the inverting and noninverting inputs) is kept to only a few microvolts. This minuscule input signal results in excellent linearity from an otherwise ghastly nonlinear circuit.

Third, apply a linearizing transform. For example, consider two signals being multiplied to make a third: . Taking the logarithm of the $a[n] = b[n] \times c[n]$ signals changes the nonlinear process of multiplication into the linear process of addition: . The fancy name for this $\log(a[n]) = \log(b[n]) + \log(c[n])$ approach is *homomorphic* signal processing. For example, a visual image can be modeled as the reflectivity of the scene (a two-dimensional signal) being multiplied by the ambient illumination (another two-dimensional signal). Homomorphic techniques enable the illumination signal to be made more uniform, thereby improving the image.

In the next chapters we examine the two main techniques of signal processing: *convolution* and *Fourier analysis*. Both are based on the strategy presented in this chapter: (1) decompose signals into simple additive components, (2) process the components in some useful manner, and (3) synthesize the components into a final result. This is DSP.

CHAPTER

6 Convolution

Convolution is a mathematical way of combining two signals to form a third signal. It is the single most important technique in Digital Signal Processing. Using the strategy of impulse decomposition, systems are described by a signal called the *impulse response*. Convolution is important because it relates the three signals of interest: the input signal, the output signal, and the impulse response. This chapter presents convolution from two different viewpoints, called the input side algorithm and the output side algorithm. Convolution provides the mathematical framework for DSP; there is nothing more important in this book.

The Delta Function and Impulse Response

The previous chapter describes how a signal can be decomposed into a group of components called **impulses**. An impulse is a signal composed of all zeros, except a single nonzero point. In effect, impulse decomposition provides a way to analyze signals one sample at a time. The previous chapter also presented the fundamental concept of DSP: the input signal is decomposed into simple additive components, each of these components is passed through a linear system, and the resulting output components are synthesized (added). The signal resulting from this divide-and-conquer procedure is identical to that obtained by directly passing the original signal through the system. While many different decompositions are possible, two form the backbone of signal processing: impulse decomposition and Fourier decomposition. When impulse decomposition is used, the procedure can be described by a mathematical operation called **convolution**. In this chapter (and most of the following ones) we will only be dealing with *discrete* signals. Convolution also applies to *continuous* signals, but the mathematics is more complicated. We will look at how continious signals are processed in Chapter 13.

Figure 6-1 defines two important terms used in DSP. The first is the **delta function**, symbolized by the Greek letter delta, . The delta function is ***[*n*]** a *normalized* impulse, that is, sample number zero has a value of one, while

all other samples have a value of zero. For this reason, the delta function is frequently called the **unit impulse**.

The second term defined in Fig. 6-1 is the **impulse response**. As the name suggests, the impulse response is the signal that exits a system when a delta function (unit impulse) is the input. If two systems are different in any way, they will have different impulse responses. Just as the input and output signals are often called and , the impulse response is usually given the *x*[*n*] *y*[*n*] symbol, . Of course, this can be changed if a more descriptive name is ***h*[*n*]** available, for instance, might be used to identify the impulse response of *f* [*n*] a *filter*.

Any impulse can be represented as a *shifted* and *scaled* delta function. Consider a signal, , composed of all zeros except sample number 8, *a*[*n*] which has a value of -3. This is the same as a delta function shifted to the right by 8 samples, and multiplied by -3. In equation form:

. Make sure you understand this notation, it is used in *a*[*n*] ' &3*[*n*&8] nearly all DSP equations.

If the input to a system is an impulse, such as , what is the system's &3*[*n*&8] output? This is where the properties of homogeneity and shift invariance are used. Scaling and shifting the input results in an identical scaling and shifting of the output. If results in , it follows that results in *[*n*] *h*[*n*] &3*[*n*&8] . In words, the output is a version of the impulse response that has &3*h*[*n*&8] been *shifted* and *scaled* by the same amount as the delta function on the input. If you know a system's impulse response, you immediately know how it will react to *any* impulse.

Convolution

Let's summarize this way of understanding how a system changes an input signal into an output signal. First, the input signal can be decomposed into a set of impulses, each of which can be viewed as a scaled and shifted delta function. Second, the output resulting from each impulse is a scaled and shifted version of the impulse response. Third, the overall output signal can be found by adding these scaled and shifted impulse responses. In other words, if we

know a system's impulse response, then we can calculate what the output will be for any possible input signal. This means we know *everything* about the system. There is nothing more that can be learned about a linear system's characteristics. (However, in later chapters we will show that this information can be represented in different forms).

The impulse response goes by a different name in some applications. If the system being considered is a *filter*, the impulse response is called the **filter kernel**, the **convolution kernel**, or simply, the **kernel**. In image processing, the impulse response is called the **point spread function**. While these terms are used in slightly different ways, they all mean the same thing, the signal produced by a system when the input is a delta function.

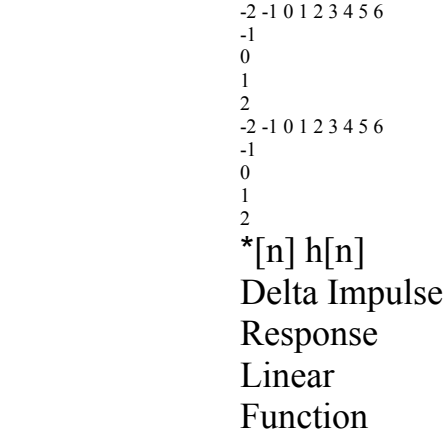


FIGURE 6-1
Definition of *delta function* and *impulse response*. The delta function is a normalized impulse. All of its samples have a value of zero, except for sample number zero, which has a value of one. The Greek letter delta, , is used to identify the delta function. The *impulse response* of a linear system, usually *[*n*] denoted by , is the output of the system when the input is a delta function. *h*[*n*]

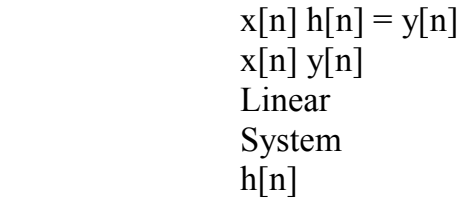


FIGURE 6-2
How convolution is used in DSP. The output signal from a linear system is equal to the input signal *convolved* with the system's impulse response. Convolution is denoted by a star when writing equations.

Convolution is a formal mathematical operation, just as multiplication, addition, and integration. Addition takes two *numbers* and produces a third *number*, while convolution takes two *signals* and produces a third *signal*. Convolution is used in the mathematics of many fields, such as probability and statistics. In linear systems, convolution is used to describe the relationship between three signals of interest: the input signal, the impulse response, and the output signal.

Figure 6-2 shows the notation when convolution is used with linear systems. An input signal, , enters a linear system with an impulse response, , *x*[*n*] *h*[*n*] resulting in an output signal, . In equation form: . *y*[*n*] *x*[*n*] t *h*[*n*] ' *y*[*n*]

Expressed in words, the input signal convolved with the impulse response is equal to the output signal. Just as addition is represented by the plus, +, and multiplication by the cross, ×, convolution is represented by the star, t. It is unfortunate that most programming languages also use the star to indicate multiplication. A star in a computer program means multiplication, while a star in an equation means convolution.

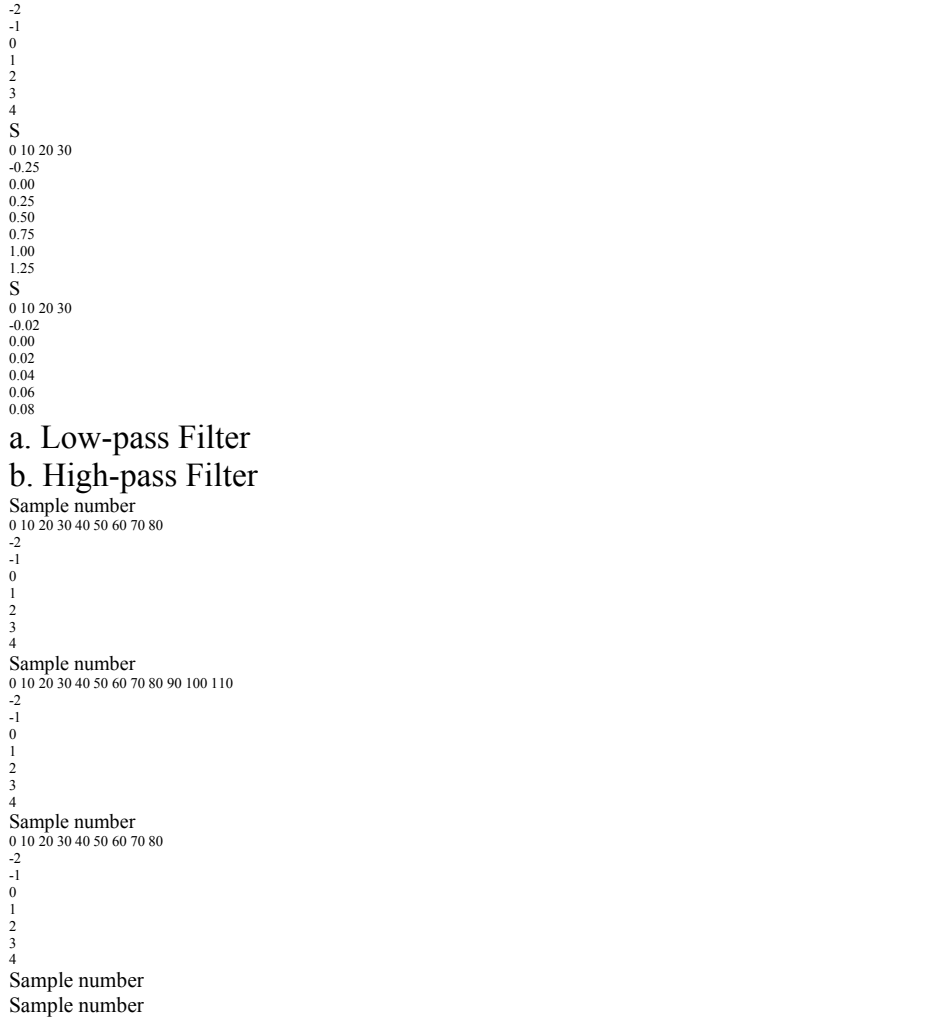


FIGURE 6-3

Examples of low-pass and high-pass filtering using convolution. In this example, the input signal is a few cycles of a sine wave plus a slowly rising ramp. These two components are separated by using properly selected impulse responses.

Figure 6-3 shows convolution being used for low-pass and high-pass filtering. The example input signal is the sum of two components: three cycles of a sine wave (representing a high frequency), plus a slowly rising ramp (composed of low frequencies). In (a), the impulse response for the low-pass filter is a smooth arch, resulting in only the slowly changing ramp waveform being passed to the output. Similarly, the high-pass filter, (b), allows only the more rapidly changing sinusoid to pass.

Figure 6-4 illustrates two additional examples of how convolution is used to process signals. The inverting attenuator, (a), flips the signal top-for-bottom, and reduces its amplitude. The discrete derivative (also called the first difference), shown in (b), results in an output signal related to the *slope* of the input signal.

Notice the lengths of the signals in Figs. 6-3 and 6-4. The input signals are 81 samples long, while each impulse response is composed of 31 samples. In most DSP applications, the input signal is hundreds, thousands, or even millions of samples in length. The impulse response is usually much shorter, say, a few points to a few hundred points. The mathematics behind convolution doesn't restrict how long these signals are. It does, however, specify the length of the output signal. The length of the output signal is

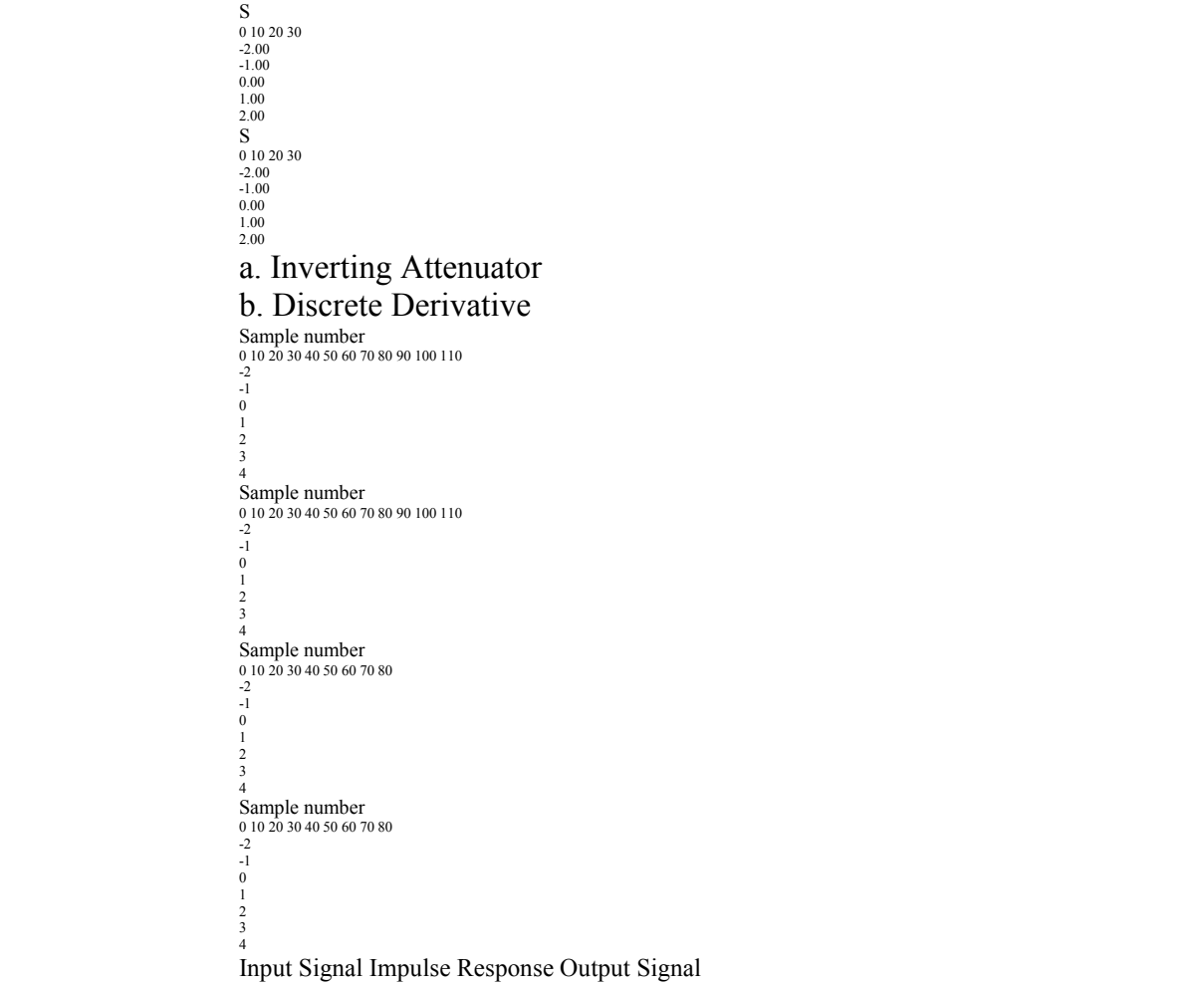


FIGURE 6-4

Examples of signals being processed using convolution. Many signal processing tasks use very simple impulse responses. As shown in these examples, dramatic changes can be achieved with only a few nonzero points.

equal to the length of the input signal, plus the length of the impulse response, minus one. For the signals in Figs. 6-3 and 6-4, each output signal is: samples long. The input signal runs from sample 81% 31& 1 ’ 111

0 to 80, the impulse response from sample 0 to 30, and the output signal from sample 0 to 110.

Now we come to the detailed mathematics of convolution. As used in Digital Signal Processing, convolution can be understood in two separate ways. The first looks at convolution from the **viewpoint of the input signal**. This involves analyzing how each sample in the input signal *contributes* to many points in the output signal. The second way looks at convolution from the **viewpoint of the output signal**. This examines how each sample in the output signal has *received* information from many points in the input signal. Keep in mind that these two perspectives are different ways of thinking about the same mathematical operation. The first viewpoint is important because it provides a *conceptual* understanding of how convolution pertains to DSP. The second viewpoint describes the *mathematics* of convolution. This typifies one of the most difficult tasks you will encounter in DSP: making your conceptual understanding fit with the jumble of mathematics used to communicate the ideas.

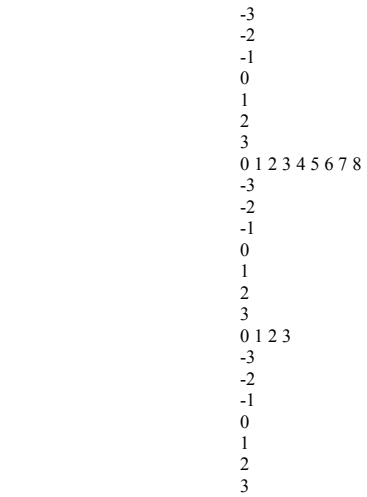


FIGURE 6-5

Example convolution problem. A nine point input signal, convolved with a four point impulse response, results in a twelve point output signal. Each point in the input signal contributes a scaled and shifted impulse response to the output signal. These nine scaled and shifted impulse responses are shown in Fig. 6-6.

Now examine sample , the last point in the input signal. This sample is at $x[8]$ index number eight, and has a value of -0.5. As shown in the lower-right graph of Fig. 6-6, results in an impulse response that has been shifted to the right $x[8]$ by eight points and multiplied by -0.5. Place holding zeros have been added at points 0-7. Lastly, examine the effect of points and . Both these $x[0]$ $x[7]$ samples have a value of zero, and therefore produce output components consisting of all zeros.

The Input Side Algorithm

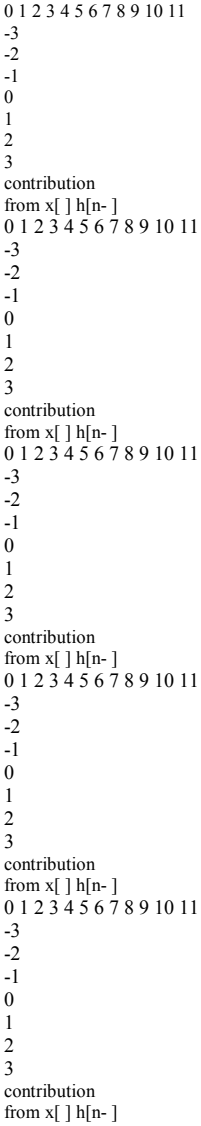
Figure 6-5 shows a simple convolution problem: a 9 point input signal, , $x[n]$ is passed through a system with a 4 point impulse response, , resulting $h[n]$ in a point output signal, . In mathematical terms, is 9% 4& 1 ' 12 $y[n]$ $x[n]$ convolved with to produce . This first viewpoint of convolution is $h[n]$ $y[n]$ based on the fundamental concept of DSP: decompose the input, pass the components through the system, and synthesize the output. In this example, each of the nine samples in the input signal will contribute a scaled and

shifted version of the impulse response to the output signal. These nine signals are shown in Fig. 6-6. Adding these nine signals produces the output signal, . $y[n]$

Let's look at several of these nine signals in detail. We will start with sample number four in the input signal, i.e., . This sample is at index number four, $x[4]$ and has a value of 1.4. When the signal is decomposed, this turns into an impulse represented as: . After passing through the system, the $1.4^*[n\&4]$ resulting output component will be: . This signal is shown in the $1.4\ h[n\&4]$ center box of the nine signals in Fig. 6-6. Notice that this is the impulse response, , multiplied by 1.4, and shifted four samples to the right. Zeros $h[n]$ have been added at samples 0-3 and at samples 8-11 to serve as place holders. To make this more clear, Fig. 6-6 uses *squares* to represent the data points that come from the shifted and scaled impulse response, and *diamonds* for the added zeros.

FIGURE 6-6

Output signal components for the convolution in Fig. 6-5. In these signals, each point that results from a scaled and shifted impulse response is represented by a square marker. The remaining data points, represented by diamonds, are zeros that have been added as place holders.



0 1 2 3 4 5 6 7 8 9 10 11
-3
-2
-1
0
1
2
3
contribution
from x[ ] h[n-]
0 1 2 3 4 5 6 7 8 9 10 11
-3
-2
-1
0
1
2
3
contribution
from x[ ] h[n-]
0 1 2 3 4 5 6 7 8 9 10 11
-3
-2
-1
0
1
2
3
contribution
from x[ ] h[n-]
0 1 2 3 4 5 6 7 8 9 10 11
-3
-2
-1
0
1
2
3
contribution
from x[ ] h[n-]
0 0 1 1 2 2
3 3 4 4 5 5
6 6 7 7 8 8

In this example, is a nine point signal and is a four point signal. In $x[n]$ $h[n]$ our next example, shown in Fig. 6-7, we will reverse the situation by making $x[n]$ a four point signal, and a nine point signal. The same two waveforms are $h[n]$ used, they are just swapped. As shown by the output signal components, the four samples in result in four shifted and scaled versions of the nine point $x[n]$ impulse response. Just as before, leading and trailing zeros are added as place holders.

But wait just one moment! The output signal in Fig. 6-7 is *identical* to the output signal in Fig. 6-5. This isn't a mistake, but an important property. Convolution is *commutative*: . The mathematics does $a[n]$t$b[n]$ ' $b[n]$t$a[n]$ not care which is the input signal and which is the impulse response, only that *two signals are convolved with each other*. Although the mathematics may allow it, exchanging the two signals has no physical meaning in system theory. The input signal and impulse response are two totally different things and exchanging them doesn't make sense. What the commutative property provides is a *mathematical tool* for manipulating equations to achieve various results.

TABLE 6-1

```
100 'CONVOLUTION USING THE INPUT SIDE ALGORITHM
110 '
120 DIM X[80] 'The input signal, 81 points
130 DIM H[30] 'The impulse response, 31 points
140 DIM Y[110] 'The output signal, 111 points
```

```
150 '
160 GOSUB XXXX 'Mythical subroutine to load X[ ] and H[ ]
170 '
180 FOR I% = 0 TO 110 'Zero the output array
190 Y(I%) = 0
200 NEXT I%
210 '
220 FOR I% = 0 TO 80 'Loop for each point in X[ ]
230 FOR J% = 0 TO 30 'Loop for each point in H[ ]
240 Y[I%+J%] = Y[I%+J%] + X[I%]tH[J%]
250 NEXT J%
260 NEXT I% '(remember, t is multiplication in programs!)
270 '
280 GOSUB XXXX 'Mythical subroutine to store Y[ ]
290 '
300 END
```

A program for calculating convolutions using the input side algorithm is shown in Table 6-1. Remember, the programs in this book are meant to convey *algorithms* in the simplest form, even at the expense of good programming style. For instance, all of the input and output is handled in **mythical** subroutines (lines 160 and 280), meaning we do not define how these operations are conducted. Do not skip over these programs; they are a key part of the material and you need to understand them in detail.

The program convolves an 81 point input signal, held in array X[ ], with a 31 point impulse response, held in array H[ ], resulting in a 111 point output signal, held in array Y[ ]. These are the same lengths shown in Figs. 6-3 and 6-4. Notice that the names of these arrays use upper case letters. This is a violation of the naming conventions previously discussed, because upper case letters are reserved for frequency domain signals. Unfortunately, the simple BASIC used in this book does not allow lower case variable names. Also notice that line 240 uses a star for *multiplication*. Remember, a star in a program means multiplication, while a star in an equation means convolution. A star in text (such as documentation or program comments) can mean either.

The mythical subroutine in line 160 places the input signal into X[ ] and the impulse response into H[ ]. Lines 180-200 set all of the values in Y[ ] to zero. This is necessary because Y[ ] is used as an accumulator to sum the output components as they are calculated. Lines 220 to 260 are the heart of the program. The FOR statement in line 220 controls a loop that steps through each point in the input signal, X[ ]. For each sample in the input signal, an inner loop (lines 230-250) calculates a scaled and shifted version of the impulse response, and adds it to the array accumulating the output signal, Y[ ]. This nested loop structure (one loop within another loop) is a key characteristic of convolution programs; become familiar with it.

FIGURE 6-7

A second example of convolution. The waveforms for the input signal and impulse response are exchanged from the example of Fig. 6-5. Since convolution is commutative, the output signals for the two examples are identical.

```
0 1 2 3 4 5 6 7 8 9 10 11
-3
-2
-1
0
1
2
3
0 1 2 3 4 5 6 7 8
-3
-2
-1
```

0
1
2
3
0 1 2 3
-3
-2
-1
0
1
2
3
h[n] x[n] y[n]
0 1 2 3 4 5 6 7 8 9 10 11
-3
-2
-1
0
1
2
3
contribution
from x[ ] h[n-]
0 1 2 3 4 5 6 7 8 9 10 11
-3
-2
-1
0
1
2
3
contribution
from x[ ] h[n-]
0 1 2 3 4 5 6 7 8 9 10 11
-3
-2
-1
0
1
2
3
contribution
from x[ ] h[n-]
0 1 2 3 4 5 6 7 8 9 10 11
-3
-2
-1
0
1
2
3
contribution
from x[ ] h[n-]
0 0 1 1
2 2 3 3

Output signal components

Keeping the indexing straight in line 240 can drive you crazy! Let's say we are halfway through the execution of this program, so that we have just begun action on sample X[40], i.e., I% = 40. The inner loop runs through each point in the impulse response doing three things. First, the impulse response is *scaled* by multiplying it by the value of the input sample. If this were the only action taken by the inner loop, line 240 could be written, Y[J%] = X[40]*H[J%]. Second, the scaled impulse is *shifted* 40 samples to the right by adding this number to the index used in the output signal. This second action would change line 240 to: Y[40+J%] = X[40]*H[J%]. Third, Y[ ] must accumulate (*synthesize*) all the signals resulting from each sample in the input signal. Therefore, the new information must be added to the information that is already in the array. This results in the final command: Y[40+J%] = Y[40+J%] + X[40]*H[J%]. Study this carefully; it is *very* confusing, but *very* important.

The Output Side Algorithm

The first viewpoint of convolution analyzes how each sample in the *input signal* affects many samples in the output signal. In this second viewpoint, we reverse this by looking at individual samples in the *output signal*, and finding the contributing points from the input. This is important from both mathematical and practical standpoints. Suppose that we are given some input signal and impulse response, and want to find the convolution of the two. The most straightforward method would be to write a program that loops through the *output signal*, calculating one sample on each loop cycle. Likewise, equations are written in the form: *some combination of* $y[n]$ ' *other variables*. That is, sample n in the output signal is equal to some combination of the many values in the input signal and impulse response. This requires a knowledge of how each sample in the output signal can be calculated independently of all other samples in the output signal. The output side algorithm provides this information.

Let's look at an example of how a single point in the output signal is influenced by several points from the input. The example point we will use is in Fig. $y[6]$ 6-5. This point is equal to the sum of all the sixth points in the nine output components, shown in Fig. 6-6. Now, look closely at these nine output components and identify which can affect . That is, find which of these $y[6]$ nine signals contains a nonzero sample at the sixth position. Five of the output components only have *added* zeros (the diamond markers) at the sixth sample, and can therefore be ignored. Only four of the output components are capable of having a nonzero value in the sixth position. These are the output components generated from the input samples: . By $x[3]$, $x[4]$, $x[5]$, and $x[6]$ adding the sixth sample from each of these output components, is $y[6]$ determined as: . That is, four $y[6] ' x[3]h[3] \% x[4]h[2] \% x[5]h[1] \% x[6]h[0]$ samples from the input signal are multiplied by the four samples in the impulse response, and the products added.

Figure 6-8 illustrates the output side algorithm as a **convolution machine**, a flow diagram of how convolution occurs. Think of the input signal, , and $x[n]$ the output signal, , as fixed on the page. The convolution machine, $y[n]$ everything inside the dashed box, is free to move left and right as needed. The convolution machine is positioned so that its output is aligned with the output sample being calculated. Four samples from the input signal fall into the inputs of the convolution machine. These values are multiplied by the indicated samples in the impulse response, and the products are added. This produces the value for the output signal, which drops into its proper place. For example, is shown being calculated from the four input samples: $y[6]$. $x[3]$, $x[4]$, $x[5]$, and $x[6]$

To calculate , the convolution machine moves one sample to the right. This $y[7]$ results in another four samples entering the machine, through , and the $x[4]$ $x[7]$ value for dropping into the proper place. This process is repeated for all $y[7]$ points in the output signal needing to be calculated.

0 1 2 3 4 5 6 7 8
-3
-2
-1
0
1
2
3
-3 -2 -1 0
-3.0
-2.0
-1.0

0.0
1.0
2.0
3.0

x[n]
y[n]
h[n] 0
-1
-2
-3
2
3
1
(flipped)

FIGURE 6-8
The convolution machine. This is a flow diagram showing how each sample in the output signal is influenced by the input signal and impulse response. See the text for details.

0 1 2 3 4 5 6 7 8 9 10 11
-3
-2
-1
0
1
2
3

The arrangement of the impulse response *inside* the convolution machine is very important. The impulse response is *flipped left-for-right*. This places sample number zero on the right, and increasingly positive sample numbers running to the left. Compare this to the normal impulse response in Fig. 6-5 to understand the geometry of this flip. Why is this flip needed? It simply falls out of the mathematics. The impulse response describes how each point in the input signal affects the output signal. This results in each point in the output signal being affected by points in the input signal weighted by a *flipped* impulse response.

FIGURE 6-9
The convolution machine in action. Figures (a) through (d) show the convolution machine set to calculate four different output signal samples, y[0], y[3], y[8], and y[11].

0 1 2 3 4 5 6 7 8
-3
-2
-1
0
1
2
3
-3 -2 -1 0
-3.0
-2.0
-1.0
0.0
1.0
2.0
3.0

x[n]
y[n]
h[n] 0
-1
-2
-3
2
3
1
(flipped)

a. Set to calculate y[0]

0 1 2 3 4 5 6 7 8
-3
-2
-1
0
1
2
3
-3 -2 -1 0
-3.0
-2.0
-1.0
0.0

1.0
2.0
3.0
x[n]
y[n]
h[n] 0
-1
-2
-3
2
3
1
(flipped)
b. Set to calculate y[3]
0 1 2 3 4 5 6 7 8 9 10 11
-3
-2
-1
0
1
2
3
0 1 2 3 4 5 6 7 8 9 10 11
-3
-2
-1
0
1
2
3
Figure 6-9 shows the convolution machine being used to calculate several samples in the output signal. This diagram also illustrates a real nuisance in convolution. In (a), the convolution machine is located fully to the left with its output aimed at . In this position, it is trying to receive input from *y*[0] samples: . The problem is, three of these samples: *x*[&3], *x*[&2], *x*[&1], and *x*[0] , do not exist! This same dilemma arises in (d), where *x*[&3], *x*[&2], and *x*[&1] the convolution machine tries to accept samples to the right of the defined input signal, points . *x*[9], *x*[10], and *x*[11]

One way to handle this problem is by *inventing* the nonexistent samples. This involves adding samples to the ends of the input signal, with each of the added samples having a value of *zero*. This is called **padding** the signal with zeros. Instead of trying to access a nonexistent value, the convolution machine receives a sample that has a value of zero. Since this zero is eliminated during the multiplication, the result is mathematically the same as *ignoring* the nonexistent inputs.

0 1 2 3 4 5 6 7 8 9 10 11
-3
-2
-1
0
1
2
3
0 1 2 3 4 5 6 7 8 9 10 11
-3
-2
-1
0
1
2
3
0 1 2 3 4 5 6 7 8
-3
-2
-1
0
1
2
3
-3 -2 -1 0
-3.0
-2.0
-1.0
0.0
1.0
2.0
3.0
x[n]
y[n]
h[n] 0
-1
-2
-3
2
3

1
(flipped)
c. Set to calculate y[8]
0 1 2 3 4 5 6 7 8
-3
-2
-1
0
1
2
3
-3 -2 -1 0
-3.0
-2.0
-1.0
0.0
1.0
2.0
3.0
x[n]
y[n]
h[n] 0
-1
-2
-3
2
3
1
(flipped)
d. Set to calculate y[11]
Figure 6-9 (continued)

The important part is that the far left and far right samples in the output signal are based on *incomplete* information. In DSP jargon, **the impulse response is not fully immersed in the input signal**. If the impulse response is *M* points in length, the first *and* last samples in the output signal are based *M*&1 on less information than the samples between. This is analogous to an electronic circuit requiring a certain amount of time to stabilize after the power is applied. The difference is that this transient is easy to ignore in electronics, but very prominent in DSP.

Figure 6-10 shows an example of the trouble these end effects can cause. The input signal is a sine wave plus a DC component. The desire is to remove the DC part of the signal, while leaving the sine wave intact. This calls for a highpass filter, such as the impulse response shown in the figure. The problem is, the first and last 30 points are a mess! The shape of these end regions can be understood by imagining the input signal padded with 30 zeros on the left side, samples through , and 30 zeros on the right, samples *x*[&1] *x*[&30] *x*[81] through . The output signal can then be viewed as a filtered version *x*[110] of this longer waveform. These "end effect" problems are widespread in

EQUATION 6-1
The convolution summation. This is the formal definition of convolution, written in the shorthand: . In this *y* [*n*] ' *x* [*n*] t *h*[*n*] equation, is an *M* point signal with *h*[*n*] indexes running from 0 to *M*-1.

$$y\,[i\,] \;'\; \mathrm{j}_{M\&1}\; h[\,j\,]\; x\,[i\&j\,]$$

j '0

DSP. As a general rule, expect that the beginning and ending samples in processed signals will be quite useless.

Now the math. Using the convolution machine as a guideline, we can write the standard equation for convolution. If is an *N* point signal running from 0 *x*[*n*] to *N*-1, and is an *M* point signal running from 0 to *M*-1, the convolution *h*[*n*] of the two: , is an *N*+*M*-1 point signal running from 0 to *y*[*n*] ' *x*[*n*] t *h*[*n*] *N*+*M*-2, given by:

This equation is called the **convolution sum**. It allows each point in the output signal to be calculated independently of all other points in the output

signal. The index, *i*, determines which sample in the output signal is being calculated, and therefore corresponds to the left-right position of the convolution machine. In computer programs performing convolution, a loop makes this index run through each sample in the output signal. To calculate one of the output samples, the index, *j*, is used *inside* of the convolution machine. As *j* runs through 0 to *M*-1, each sample in the impulse response, is multiplied by the proper sample from the input *h*[*j*], signal, All these products are added to produce the output sample *x*[*i*& *j*]. being calculated. Study Eq. 6-1 until you fully understand how it is implemented by the convolution machine. Much of DSP is based on this equation. (Don't be confused by the *n* in . This is merely *y*[*n*] ' *x*[*n*] t *h*[*n*] a place holder to indicate that *some* variable is the index into the array. Sometimes the equations are written: , just to avoid having *y*[ ] ' *x*[ ] t *h*[ ] to bring in a meaningless symbol).

Table 6-2 shows a program for performing convolutions using the *output side algorithm*, a direct use of Eq. 6-1. This program produces the same output signal as the program for the *input side algorithm*, shown previously in Table 6-1. Notice the main difference between these two programs: the input side algorithm loops through each sample in the *input signal* (line 220 of Table 6-1), while the output side algorithm loops through each sample in the *output signal* (line 180 of Table 6-2).

Here is a detailed operation of this program. The FOR-NEXT loop in lines 180 to 250 steps through each sample in the output signal, using I% as the index. For each of these values, an inner loop, composed of lines 200 to 230, calculates the value of the output sample, Y[I%]. The value of Y[I%] is set to zero in line 190, allowing it to accumulate the products inside of the convolution machine. The FOR-NEXT loop in lines 200 to 240 provide a direct implementation of Eq. 6-1. The index, J%, steps through each

sample in the impulse response. Line 230 provides the multiplication of each sample in the impulse response, H[J%], with the appropriate sample from the input signal, X[I%-J%], and adds the result to the accumulator.

In line 230, the sample taken from the input signal is: X[I%-J%]. Lines 210 and 220 prevent this from being outside the defined array, X[0] to X[80]. In other words, this program handles undefined samples in the input signal by *ignoring* them. Another alternative would be to define the input signal's array from X[-30] to X[110], allowing 30 zeros to be padded on each side of the true data. As a third alternative, the FOR-NEXT loop in line 180 could be changed to run from 30 to 80, rather than 0 to 110. That is, the program would only calculate the samples in the output signal where the impulse response is *fully immersed* in the input signal. The important thing is that you must use one of these three techniques. If you don't, the program will crash when it tries to read the out-of-bounds data.

S
0 10 20 30
-0.5
0.0
0.5
1.0
1.5
Sample number
0 10 20 30 40 50 60 70 80
-4
-2
0
2
4
Sample number
0 10 20 30 40 50 60 70 80 90 100 110
-4
-2
0
2

4
Input signal Impulse response Output signal
unusable usable unusable
Sample number
Amplitude
Amplitude
Amplitude
FIGURE 6-10
End effects in convolution. When an input signal is convolved with an *M* point impulse response, the first and last *M*-1 points in the output signal may not be usable. In this example, the impulse response is a high-pass filter used to remove the DC component from the input signal.

```
100 'CONVOLUTION USING THE OUTPUT SIDE ALGORITHM
110 '
120 DIM X[80] 'The input signal, 81 points
130 DIM H[30] 'The impulse response, 31 points
140 DIM Y[110] 'The output signal, 111 points
150 '
160 GOSUB XXXX 'Mythical subroutine to load X[ ] and H[ ]
170 '
180 FOR I% = 0 TO 110 'Loop for each point in Y[ ]
190 Y[I%] = 0 'Zero the sample in the output array
200 FOR J% = 0 TO 30 'Loop for each point in H[ ]
210 IF (I%-J% < 0) THEN GOTO 240
220 IF (I%-J% > 80) THEN GOTO 240
230 Y(I%) = Y(I%) + H(J%) t X(I%-J%)
240 NEXT J%
250 NEXT I%
260 '
270 GOSUB XXXX 'Mythical subroutine to store Y[ ]
280 '
290 END
```

TABLE 6-2

The Sum of Weighted Inputs

The characteristics of a linear system are completely described by its impulse response. This is the basis of the input side algorithm: each point in the input signal contributes a scaled and shifted version of the impulse response to the output signal. The mathematical consequences of this lead to the output side algorithm: each point in the output signal receives a contribution from many points in the input signal, multiplied by a *flipped* impulse response. While this is all true, it doesn't provide the full story on why convolution is important in signal processing.

Look back at the convolution machine in Fig. 6-8, and ignore that the signal inside the dotted box is an *impulse response*. Think of it as a set of **weighing coefficients** that happen to be embedded in the flow diagram. In this view, each sample in the output signal is equal to a *sum of weighted inputs*. Each sample in the output is influenced by a region of samples in the input signal, as determined by what the weighing coefficients are chosen to be. For example, imagine there are ten weighing coefficients, each with a value of onetenth. This makes each sample in the output signal the *average* of ten samples from the input.

Taking this further, the weighing coefficients do not need to be restricted to the *left side* of the output sample being calculated. For instance, Fig. 6-8 shows $y[6]$ being calculated from: . Viewing the convolution $x[3]$, $x[4]$, $x[5]$, and $x[6]$ machine as a sum of weighted inputs, the weighing coefficients could be chosen *symmetrically* around the output sample. For example, might receive $y[6]$ contributions from: . Using the same indexing $x[4]$, $x[5]$, $x[6]$, $x[7]$, and $x[8]$ notation as in Fig. 6-8, the weighing coefficients for these five inputs would be

held in: . In other words, the impulse $h[2]$, $h[1]$, $h[0]$, $h[-1]$, and $h[-2]$ response that corresponds to our selection of symmetrical weighing coefficients requires the use of *negative indexes*. We will return to this in the next chapter. Mathematically, there is only one concept here: convolution as defined by Eq. 6-1. However, science and engineering problems approach this single concept from two distinct directions. Sometimes you will want to think of a system in terms of what its impulse response looks like. Other times you will understand the system as a set of weighing coefficients. You need to become familiar with both views, and how to toggle between them.

EQUATION 7-1
The delta function is the identity for convolution. Any signal convolved with a delta function is left unchanged.

$$x[n] * \delta[n] = x[n]$$

Properties of Convolution

A linear system's characteristics are completely specified by the system's impulse response, as governed by the mathematics of convolution. This is the basis of many signal processing techniques. For example: Digital filters are created by *designing* an appropriate impulse response. Enemy aircraft are detected with radar by *analyzing* a measured impulse response. Echo suppression in long distance telephone calls is accomplished by creating an impulse response that *counteracts* the impulse response of the reverberation. The list goes on and on. This chapter expands on the properties and usage of convolution in several areas. First, several common impulse responses are discussed. Second, methods are presented for dealing with cascade and parallel combinations of linear systems. Third, the technique of *correlation* is introduced. Fourth, a nasty problem with convolution is examined, the computation time can be unacceptably long using conventional algorithms and computers.

Common Impulse Responses

Delta Function

The simplest impulse response is nothing more that a delta function, as shown in Fig. 7-1a. That is, an impulse on the input produces an identical impulse on the output. This means that *all* signals are passed through the system *without change*. Convolving any signal with a delta function results in exactly the same signal. Mathematically, this is written:

This property makes the delta function the **identity** for convolution. This is analogous to *zero* being the identity for addition , and *one* being the ($a + 0 = a$) identity for multiplication . At first glance, this type of system ($a \times 1 = a$)

EQUATION 7-2
A system that amplifies or attenuates has a scaled delta function for an impulse response. In this equation, k determines the amplification or attenuation.

$$x[n] * k\delta[n] = k\,x[n]$$

EQUATION 7-3
A relative shift between the input and output signals corresponds to an impulse response that is a shifted delta function. The variable, s, determines the amount of

shift in this equation.

$x[n](*[n\%s]) \leftrightarrow x[n\%s]$

may seem trivial and uninteresting. Not so! Such systems are the ideal for data storage, communication and measurement. Much of DSP is concerned with passing information through systems without change or degradation.

Figure 7-1b shows a slight modification to the delta function impulse response. If the delta function is made larger or smaller in amplitude, the resulting system is an **amplifier** or **attenuator**, respectively. In equation form, amplification results if k is *greater than one*, and attenuation results if k is *less than one*:

The impulse response in Fig. 7-1c is a delta function with a **shift**. This results in a system that introduces an identical shift between the input and output signals. This could be described as a signal **delay**, or a signal **advance**, depending on the direction of the shift. Letting the shift be represented by the parameter, s, this can be written as the equation:

Science and engineering are filled with cases where one signal is a shifted version of another. For example, consider a radio signal transmitted from a remote space probe, and the corresponding signal received on the earth. The time it takes the radio wave to propagate over the distance causes a delay between the transmitted and received signals. In biology, the electrical signals in adjacent nerve cells are shifted versions of each other, as determined by the time it takes an action potential to cross the synaptic junction that connects the two.

Figure 7-1d shows an impulse response composed of a delta function plus a shifted and scaled delta function. By superposition, the output of this system is the input signal plus a delayed version of the input signal, i.e., an *echo*. Echoes are important in many DSP applications. The addition of echoes is a key part in making audio recordings sound natural and pleasant. Radar and sonar analyze echoes to detect aircraft and submarines. Geophysicists use echoes to find oil. Echoes are also very important in telephone networks, because you want to *avoid* them.

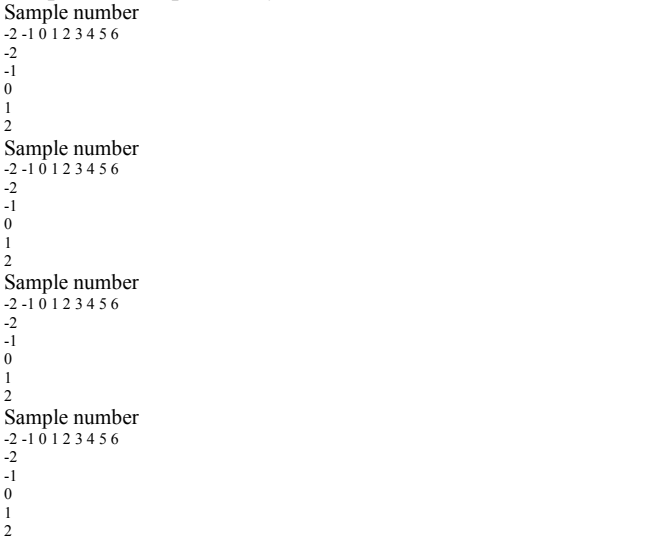


FIGURE 7-1
Simple impulse responses using shifted and scaled delta functions.
a. Identity
The delta function is the *identity* for convolution. Convolving a signal with

the delta function leaves the signal unchanged. This is the goal of systems that transmit or store signals.
b. Amplification & Attenuation
Increasing or decreasing the amplitude of the delta function forms an impulse response that *amplifies* or *attenuates*, respectively. This impulse response will amplify the signal by 1.6.
c. Shift
Shifting the delta function produces a corresponding shift between the input and output signals. Depending on the direction, this can be called a *delay* or an *advance*. This impulse response delays the signal by four samples.
d. Echo
A delta function plus a shifted and scaled delta function results in an *echo* being added to the original signal. In this example, the echo is delayed by four samples and has an amplitude of 60% of the original signal.
Amplitude Amplitude Amplitude Amplitude

Calculus-like Operations

Convolution can change discrete signals in ways that resemble integration and differentiation. Since the terms "derivative" and "integral" specifically refer to operations on *continuous* signals, other names are given to their discrete counterparts. The discrete operation that mimics the *first derivative* is called the **first difference**. Likewise, the discrete form of the *integral* is called the

EQUATION 7-4
Calculation of the first difference. In this relation, is the original signal, $x[n]$ and is the first difference. $y[n]$

$$y[n] \text{ ' } x[n] \text{ \& } x[n\text{\&}1]$$

Sample number
-2 -1 0 1 2 3 4 5 6
-2
-1
0
1
2
Sample number
-2 -1 0 1 2 3 4 5 6
-2
-1
0
1
2
FIGURE 7-2
Impulse responses that mimic calculus operations.
a. First Difference
This is the discrete version of the *first derivative*. Each sample in the output signal is equal to the *difference* between adjacent samples in the input signal. In other words, the output signal is the *slope* of the input signal.
b. Running Sum
The running sum is the discrete version of the *integral*. Each sample in the output signal is equal to the sum of all samples in the input signal to the *left*.

Note that the impulse response extends
to infinity, a rather nasty feature.
Amplitude Amplitude
running sum. It is also common to hear these operations called the **discrete derivative** and the **discrete integral**, although mathematicians frown when they hear these informal terms used.
Figure 7-2 shows the impulse responses that implement the first difference and the running sum. Figure 7-3 shows an example using these operations. In 7-3a, the original signal is composed of several sections with varying slopes. Convolving this signal with the first difference impulse response produces the signal in Fig. 7-3b. Just as with the first derivative, the amplitude of each point in the first difference signal is equal to the *slope* at the corresponding location in the original signal. The running sum is the inverse operation of the first difference. That is, convolving the signal in (b), with the running sum's impulse response, produces the signal in (a).
These impulse responses are simple enough that a full convolution program is usually not needed to implement them. Rather, think of them in the alternative mode: each sample in the output signal is a *sum of weighted samples from the input*. For instance, the first difference can be calculated:
That is, each sample in the output signal is equal to the difference between two adjacent samples in the input signal. For instance, . It $y[40] \leftarrow x[40] - x[39]$ should be mentioned that this is not the only way to define a *discrete derivative*. Another common method is to define the slope symmetrically around the point being examined, such as: . $y[n] \leftarrow (x[n+1] - x[n-1])/2$

FIGURE 7-3
Example of calculus-like operations. The
signal in (b) is the *first difference* of the
signal in (a). Correspondingly, the signal is
(a) is the *running sum* of the signal in (b).
These processing methods are used with
discrete signals the same as differentiation
and integration are used with continuous
signals.
Sample number
0 10 20 30 40 50 60 70 80
-0.2
-0.1
0.0
0.1
0.2
Sample number
0 10 20 30 40 50 60 70 80
-2.0
-1.0
0.0
1.0
2.0
First
Difference
Running
Sum
a. Original signal
b. First difference
Amplitude Amplitude
EQUATION 7-5
Calculation of the running sum. In this
relation, is the original signal, and $x[n]$ $y[n]$
is the running sum.
$$y[n] \leftarrow x[n] + y[n-1]$$
Using this same approach, each sample in the running sum can be calculated

by summing all points in the original signal to the *left* of the sample's location. For instance, if is the running sum of , then sample is found by $y[n]$ $x[n]$ $y[40]$ adding samples through . Likewise, sample is found by adding $x[0]$ $x[40]$ $y[41]$ samples through . Of course, it would be very inefficient to calculate $x[0]$ $x[41]$ the running sum in this manner. For example, if has already been $y[40]$ calculated, can be calculated with only a single addition: $y[41]$. In equation form: $y[41]$ ' $x[41]$% $y[40]$

Relations of this type are called **recursion equations** or **difference equations**. We will revisit them in Chapter 19. For now, the important idea to understand is that these relations are *identical* to convolution using the impulse responses of Fig. 7-2. Table 7-1 provides computer programs that implement these calculus-like operations.


```
100 'Calculation of the running sum
110 Y[0] = X[0]
120 FOR I% = 1 TO N%-1
130 Y[I%] = Y[I%-1] + X[I%]
140 NEXT I%
```

```
100 'Calculation of the First Difference
110 Y[0] = 0
120 FOR I% = 1 TO N%-1
130 Y[I%] = X[I%] - X[I%-1]
140 NEXT I%
```

Table 7-1

Programs for calculating the first difference and running sum. The original signal is held in X[ ], and the processed signal (the first difference or running sum) is held in Y[ ]. Both arrays run from 0 to N%-1.

b. Square pulse

a. Exponential

FIGURE 7-4

Typical low-pass filter kernels. Low-pass filter kernels are formed from a group of adjacent positive points that provide an averaging (smoothing) of the signal. As discussed in later chapters, each of these filter kernels is best for a particular purpose. The exponential, (a), is the simplest recursive filter. The rectangular pulse, (b), is best at reducing noise while maintaining edge sharpness. The sinc function in (c), a curve of the form: , is used to separate one $\sin(x)/(x)$ band of frequencies from another.

c. Sinc

Amplitude Amplitude
Amplitude

Low-pass and High-pass Filters

The design of digital filters is covered in detail in later chapters. For now, be satisfied to understand the general shape of low-pass and high-pass *filter kernels* (another name for a filter's impulse response). Figure 7-4 shows several common low-pass filter kernels. In general, low-pass filter kernels are composed of a group of *adjacent positive points*. This results in each sample in the output signal being a weighted average of many adjacent points from the input signal. This averaging *smoothes* the signal, thereby removing highfrequency components. As shown by the sinc function in (c), some low-pass filter kernels include a few negative valued samples in the tails. Just as in analog electronics, digital low-pass filters are used for noise reduction, signal separation, wave shaping, etc.

Sample number
-20 -15 -10 -5 0 5 10 15 20
-0.50
0.00
0.50
1.00
1.50
b. Square pulse
Sample number
-20 -15 -10 -5 0 5 10 15 20
-0.5
0.0
0.5
1.0
1.5
a. Exponential

FIGURE 7-5
Typical high-pass filter kernels. These are formed by subtracting the corresponding lowpass filter kernels in Fig. 7-4 from a delta function. The distinguishing characteristic of high-pass filter kernels is a spike surrounded by many adjacent negative samples.

Sample number
-20 -15 -10 -5 0 5 10 15 20
-0.50
0.00
0.50
1.00
1.50
c. Sinc
Amplitude Amplitude
Amplitude

The cutoff frequency of the filter is changed by making filter kernel wider or narrower. If a low-pass filter has a gain of *one* at DC (zero frequency), then the sum of all of the points in the impulse response must be equal to *one*. As illustrated in (a) and (c), some filter kernels *theoretically* extend to infinity without dropping to a value of zero. In actual practice, the tails are truncated after a certain number of samples, allowing it to be represented by a finite number of points. How else could it be stored in a computer?

Figure 7-5 shows three common high-pass filter kernels, derived from the corresponding low-pass filter kernels in Fig. 7-4. This is a common strategy in filter design: first devise a low-pass filter and then transform it to what you need, high-pass, band-pass, band-reject, etc. To understand the low-pass to high-pass transform, remember that a delta function impulse response passes the entire signal, while a low-pass impulse response passes only the lowfrequency components. By superposition, a filter kernel consisting of a delta function minus the low-pass filter kernel will pass the entire signal minus the

low-frequency components. A high-pass filter is born! As shown in Fig. 7-5, the delta function is usually added at the center of symmetry, or sample zero if the filter kernel is not symmetrical. High-pass filters have zero gain at DC (zero frequency), achieved by making the sum of all the points in the filter kernel equal to *zero*.

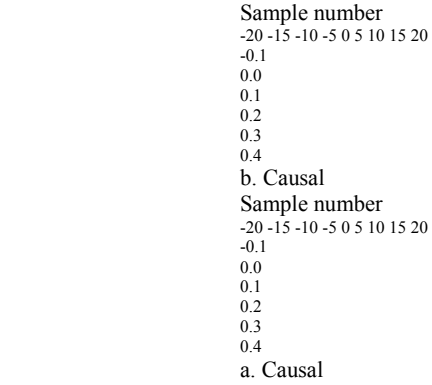
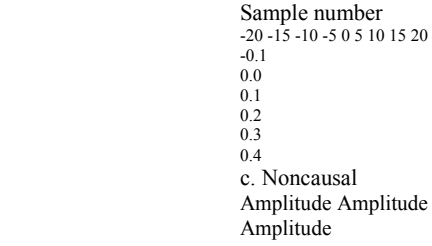


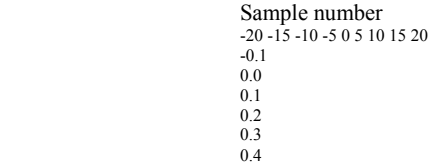
FIGURE 7-6
Examples of causal signals. An impulse response, or any signal, is said to be *causal* if all negative numbered samples have a value of zero. Three examples are shown here. Any noncausal signal with a finite number of points can be turned into a causal signal simply by shifting.

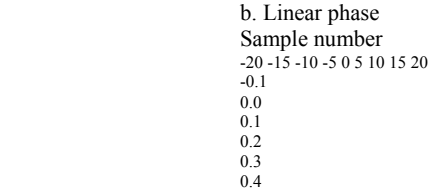


Causal and Noncausal Signals

Imagine a simple analog electronic circuit. If you apply a short pulse to the input, you will see a response on the output. This is the kind of cause and effect that our universe is based on. One thing we definitely know: *any effect must happen after the cause*. This is a basic characteristic of what we call *time*. Now compare this to a DSP system that changes an input signal into an output signal, both stored in arrays in a computer. If this mimics a real world system, it must follow the same principle of *causality* as the real world does. For example, the value at sample number eight in the input signal can only affect sample number eight or greater in the output signal. Systems that operate in this manner are said to be **causal.** Of course, digital processing doesn't necessarily have to function this way. Since both the input and output signals are arrays of numbers stored in a computer, any of the input signal values can affect any of the output signal values.

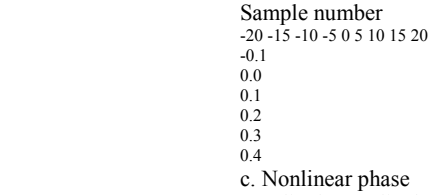
As shown by the examples in Fig. 7-6, the impulse response of a causal system must have a value of zero for all *negative numbered* samples. Think of this from the input side view of convolution. To be causal, an impulse in the input signal at sample number *n* must only affect those points in the output signal with a sample number of *n* or greater. In common usage, the term *causal* is applied to *any* signal where all the negative numbered samples have a value of zero, whether it is an impulse response or not.



b. Linear phase


a. Zero phase

FIGURE 7-7
Examples of phase linearity. Signals that have a left-right symmetry are said to be *linear phase*. If the axis of symmetry occurs at sample number zero, they are additionally said to be *zero phase*. Any linear phase signal can be transformed into a zero phase signal simply by shifting. Signals that do not have a leftright symmetry are said to be *nonlinear phase*. Do not confuse these terms with the *linear* in linear systems. They are completely different concepts.



c. Nonlinear phase

Amplitude Amplitude
Amplitude

Zero Phase, Linear Phase, and Nonlinear Phase

As shown in Fig. 7-7, a signal is said to be **zero phase** if it has left-right symmetry around sample number zero. A signal is said to be **linear phase** if it has left-right symmetry, but around some point other than zero. This means that any linear phase signal can be changed into a zero phase signal simply by shifting left or right. Lastly, a signal is said to be **nonlinear phase** if it does not have left-right symmetry.

You are probably thinking that these names don't seem to follow from their definitions. What does *phase* have to do with *symmetry*? The answer lies in the frequency spectrum, and will be discussed in more detail in later chapters. Briefly, the frequency spectrum of any signal is composed of two parts, the magnitude and the phase. The frequency spectrum of a signal that is symmetrical around zero has a phase that is zero. Likewise, the frequency spectrum of a signal that is symmetrical around some nonzero point has a phase that is a straight line, i.e., a linear phase. Lastly, the frequency spectrum of a signal that is not symmetrical has a phase that is not a straight line, i.e., it has a nonlinear phase.

A special note about the potentially confusing terms: *linear* and *nonlinear phase*. What does this have to do the concept of system linearity discussed in previous chapters? Absolutely nothing! System linearity is the broad concept

EQUATION 7-6
The commutative property of convolution.

This states that the order in which signals
are convolved can be exchanged.

$a[n] \circledast b[n] = b[n] \circledast a[n]$

b[n] a[n] y[n]

a[n] b[n] y[n]

IF

THEN

FIGURE 7-8

The commutative property in system theory. The commutative property of convolution allows the input signal and the impulse response of a system to be exchanged without changing the output. While interesting, this usually has no physical significance. (A signal appearing inside of a box, such as b[n] and a[n] in this figure, represent the *impulse response* of the system).

that nearly all of DSP is based on (superposition, homogeneity, additivity, etc). *Linear* and *nonlinear phase* mean that the phase is, or is not, a straight line. In fact, a system must be *linear* even to say that the phase is zero, linear, or nonlinear.

Mathematical Properties

Commutative Property

The commutative property for convolution is expressed in mathematical form: In words, the order in which two signals are convolved makes no difference; the results are identical. As shown in Fig. 7-8, this has a strange meaning for system theory. In any linear system, the input signal and the system's impulse response can be *exchanged* without changing the output signal. This is interesting, but usually doesn't have any physical meaning. The input signal and the impulse response are very different things. Just because the mathematics *allows* you to do something, doesn't mean that it makes sense to do it. For example, suppose you make: \$10/hour × 2,000 hours/year = \$20,000/year. The commutative property for multiplication provides that you can make the same annual salary by only working 10 hours/year at \$2000/hour. Let's see you convince your boss that this is meaningful! In spite of this, the commutative property sees great use in DSP for manipulating equations, just as in ordinary algebra.

EQUATION 7-7

The associative property of convolution
describes how three or
more signals are convolved.

$(a[n] \circledast b[n]) \circledast c[n] = a[n] \circledast (b[n] \circledast c[n])$

IF

THEN

$h_1[n]$ x[n] $h_2[n]$ y[n]

$h_2[n]$ x[n] $h_1[n]$ y[n]

ALSO

$h_1[n]$ $h_2[n]$ x[n] y[n]

FIGURE 7-9

The associative property in system theory. The associative property provides two important characteristics of *cascaded* linear systems. First, the order of the systems can be rearranged without changing the overall operation of the cascade. Second, two or more systems in a cascade can be replaced by a single system. The impulse response of the replacement system is found by convolving the impulse responses of the stages being replaced.

Associative Property

Is it possible to convolve three or more signals? The answer is yes, and the associative property describes how: convolve two of the signals to produce an intermediate signal, then convolve the intermediate signal with the third signal. The associative property provides that the order of the convolutions doesn't matter. As an equation:

The associative property is used in system theory to describe how **cascaded systems** behave. As shown in Fig. 7-9, two or more systems are said to be in a *cascade* if the output of one system is used as the input for the next system. From the associative property, the order of the systems can be rearranged without changing the overall response of the cascade. Further, any number of cascaded systems can be replaced with a *single* system. The impulse response of the replacement system is found by convolving the impulse responses of all of the original systems.

EQUATION 7-8

The distributive property of convolution describes how parallel systems are analyzed.

$a[n](b[n] \% a[n](c[n] \text{ ' } a[n] \text{ (} (b[n] \% c[n] \text{)}$

IF

THEN

x[n] y[n]

$h_1[n] + h_2[n]$ x[n] y[n]

$h_1[n]$

$h_2[n]$

FIGURE 7-10

The distributive property in system theory. The distributive property shows that parallel systems with added outputs can be replaced with a single system. The impulse response of the replacement system is equal to the sum of the impulse responses of all the original systems.

Distributive Property

In equation form, the distributive property is written:

The distributive property describes the operation of **parallel systems with added outputs**. As shown in Fig. 7-10, two or more systems can share the same input, , and have their outputs added to produce . The $x[n]$ $y[n]$ distributive property allows this combination of systems to be replaced with a single system, having an impulse response equal to the *sum* of the impulse responses of the original systems.

Transference between the Input and Output

Rather than being a formal mathematical property, this is a way of thinking about a common situation in signal processing. As illustrated in Fig. 7-11, imagine a linear system receiving an input signal, , and generating an $x[n]$ output signal, . Now suppose that the input signal is changed in some $y[n]$ linear way, resulting in a new input signal, which we will call . This x 3$[n]$ results in a new output signal, . The question is, how does the change in y 3$[n]$

h[n] x[n] y[n]

h[n] x [n] y [n]

IF

THEN

FIGURE 7-11
Tranference between the input and output. This is a way of thinking about a common situation in signal processing. A linear change made to the input signal results in the same linear change being made to the output signal.

the input signal relate to the change in the output signal? The answer is: *the output signal is changed in exactly the same linear way that the input signal was changed*. For example, if the input signal is amplified by a factor of two, the output signal will also be amplified by a factor of two. If the derivative is taken of the input signal, the derivative will also be taken of the output signal. If the input is filtered in some way, the output will be filtered in an identical manner. This can easily be proven by using the associative property.

The Central Limit Theorem

The Central Limit Theorem is an important tool in probability theory because it mathematically explains why the Gaussian probability distribution is observed so commonly in nature. For example: the amplitude of thermal noise in electronic circuits follows a Gaussian distribution; the cross-sectional intensity of a laser beam is Gaussian; even the pattern of holes around a dart board bull's eye is Gaussian. In its simplest form, the Central Limit Theorem states that a Gaussian distribution results when the observed variable is the sum of many random processes. Even if the component processes do not have a Gaussian distribution, the sum of them will.

The Central Limit Theorem has an interesting implication for convolution. If a pulse-like signal is convolved with *itself* many times, a Gaussian is produced. Figure 7-12 shows an example of this. The signal in (a) is an

FIGURE 7-12
Example of convolving a pulse waveform with itself. The Central Limit Theorem shows that a Gaussian waveform is produced when an arbitrary shaped pulse is convolved with itself many times. Figure (a) is an example pulse. In (b), the pulse is convolved with itself *once*, and begins to appear smooth and regular. In (c), the pulse is convolved with itself *three* times, and closely approximates a Gaussian.

c. x[n] x[n] x[n] x[n]
Amplitude Amplitude
Amplitude
irregular pulse, purposely chosen to be very unlike a Gaussian. Figure (b) shows the result of convolving this signal with itself *one* time. Figure (c) shows the result of convolving this signal with itself *three* times. Even with only three convolutions, the waveform looks very much like a Gaussian. In mathematics jargon, the procedure *converges* to a Gaussian *very quickly*. The width of the resulting Gaussian (i.e., F in Eq. 2-7 or 2-8) is equal to the width of the original pulse (expressed as F in Eq. 2-7) multiplied by the square root of the number of convolutions.

Correlation

The concept of correlation can best be presented with an example. Figure 7-13 shows the key elements of a radar system. A specially designed antenna transmits a short burst of radio wave energy in a selected direction. If the propagating wave strikes an object, such as the helicopter in this illustration, a small fraction of the energy is reflected back toward a radio receiver located near the transmitter. The transmitted pulse is a specific shape that we have selected, such as the triangle shown in this example. The received signal will consist of two parts: (1) a shifted and scaled version of the transmitted pulse, and (2) random noise, resulting from interfering radio waves, thermal noise in the electronics, etc. Since radio signals travel at a known rate, the speed of

TRANSMIT RECEIVE
Sample number (or time)
-10 0 10 20 30 40 50 60 70 80
-100
0
100
200
Sample number (or time)
-10 0 10 20 30 40 50 60 70 80
-0.1
0
0.1
0.2

FIGURE 7-13
Key elements of a radar system. Like other echo location systems, radar transmits a short pulse of energy that is reflected by objects being examined. This makes the received waveform a shifted version of the transmitted waveform, plus random noise. Detection of a known waveform in a noisy signal is the fundamental problem in echo location. The answer to this problem is *correlation*.

Received amplitude Transmitted amplitude

light, the shift between the transmitted and received pulse is a direct measure of the distance to the object being detected. This is the problem: given a signal of some known shape, what is the best way to determine where (or if) the signal occurs in *another* signal. Correlation is the answer.

Correlation is a mathematical operation that is very similar to convolution. Just as with convolution, correlation uses two signals to produce a third signal. This third signal is called the **cross-correlation** of the two input signals. If a signal is correlated with *itself*, the resulting signal is instead called the **autocorrelation**. The convolution machine was presented in the last chapter to show how convolution is performed. Figure 7-14 is a similar

illustration of a **correlation machine**. The received signal, , and the $x[n]$

cross-correlation signal, , are fixed on the page. The waveform we are $y[n]$ looking for, , commonly called the **target** signal, is contained *within* the $t[n]$ correlation machine. Each sample in is calculated by moving the $y[n]$ correlation machine left or right until it points to the sample being worked on. Next, the indicated samples from the received signal fall into the correlation machine, and are multiplied by the corresponding points in the target signal. The sum of these products then moves into the proper sample in the crosscorrelation signal.

The amplitude of each sample in the cross-correlation signal is a measure of how much the received signal *resembles* the target signal, *at that location*. This means that a peak will occur in the cross-correlation signal for every target signal that is present in the received signal. In other words, the value of the cross-correlation is maximized when the target signal is *aligned* with the same features in the received signal.

What if the target signal contains samples with a negative value? Nothing changes. Imagine that the correlation machine is positioned such that the target signal is perfectly aligned with the matching waveform in the received signal. As samples from the received signal fall into the correlation machine, they are multiplied by their matching samples in the target signal. Neglecting noise, a positive sample will be multiplied by itself, resulting in a positive number. Likewise, a negative sample will be multiplied by itself, also resulting in a positive number. Even if the target signal is completely negative, the peak in the cross-correlation will still be positive.

If there is noise on the received signal, there will also be noise on the crosscorrelation signal. It is an unavoidable fact that random noise looks a certain amount like any target signal you can choose. The noise on the cross-correlation signal is simply measuring this similarity. Except for this noise, the peak generated in the cross-correlation signal is symmetrical between its left and right. This is true even if the target signal isn't symmetrical. In addition, the width of the peak is twice the width of the target signal. Remember, the cross-correlation is trying to *detect* the target signal, not *recreate* it. There is no reason to expect that the peak will even look like the target signal.

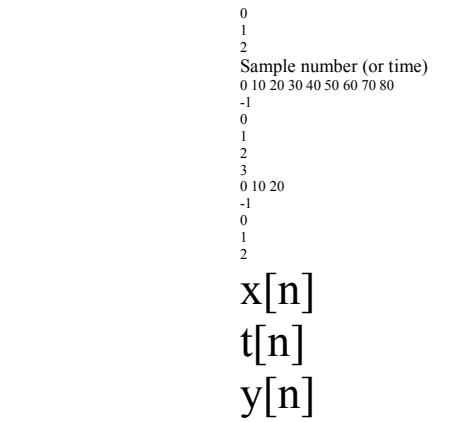
Correlation is the *optimal* technique for detecting a known waveform in random noise. That is, the peak is higher above the noise using correlation than can be produced by any other linear system. (To be perfectly correct, it is only optimal for *random white noise*). Using correlation to detect a known waveform is frequently called **matched filtering**. More on this in Chapter 17.

The correlation machine and convolution machine are identical, except for one small difference. As discussed in the last chapter, the signal inside of the convolution machine is *flipped* left-for-right. This means that samples numbers: run from the right to the left. In the correlation machine 1, 2, 3þ this flip doesn't take place, and the samples run in the normal direction.

FIGURE 7-14

The correlation machine. This is a flowchart showing how the cross-correlation of two signals is calculated. In this example, is the cross-correlation of and . The dashed box is moved left or right so that its output points at $y[n]$ $x[n]$ $t[n]$ the sample being calculated in . The indicated samples from are multiplied by the corresponding samples in , $y[n]$ $x[n]$ $t[n]$ and the products added. The correlation machine is identical to the convolution machine (Figs. 6-8 and 6-9), except that the signal inside of the dashed box is *not* reversed. In this illustration, the only samples calculated in are where $y[n]$ $t[n]$ is fully *immersed* in . $x[n]$

Sample number (or time)
0 10 20 30 40 50 60 70 80
-1



Since this signal reversal is the only difference between the two operations, it is possible to represent *correlation* using the same mathematics as *convolution*. This requires *preflipping* one of the two signals being correlated, so that the left-for-right flip inherent in convolution is canceled. For instance, when $a[n]$ and , are convolved to produce , the equation is written: $b[n]$ $c[n]$. In comparison, the cross-correlation of and can $a[n](b[n] ' c [n] a[n] b[n]$

be written: . That is, flipping left-for-right is $a[n](b[\&n] ' c [n] b [n]$ accomplished by reversing the sign of the index, i.e., . $b [\&n]$

Don't let the mathematical similarity between convolution and correlation fool you; they represent very different DSP procedures. Convolution is the relationship between a system's input signal, output signal, and impulse response. Correlation is a way to detect a known waveform in a noisy background. The similar mathematics is only a convenient coincidence.

Speed

Writing a program to convolve one signal by another is a simple task, only requiring a few lines of code. Executing the program may be more painful. The problem is the large number of additions and multiplications required by the algorithm, resulting in long execution times. As shown by the programs in the last chapter, the time-consuming operation is composed of multiplying two numbers and adding the result to an accumulator. Other parts of the algorithm, such as indexing the arrays, are very quick. The multiply-accumulate is a basic building block in DSP, and we will see it repeated in several other important algorithms. In fact, the speed of DSP computers is often *specified* by how long it takes to preform a multiply-accumulate operation.

If a signal composed of N samples is convolved with a signal composed of M samples, multiply-accumulations must be preformed. This can be seen $N \times M$ from the programs of the last chapter. Personal computers of the mid 1990's requires about one microsecond per multiply-accumulation (100 MHz Pentium using single precision floating point, see Table 4-6). Therefore, convolving a 10,000 sample signal with a 100 sample signal requires about one second. To process a one million point signal with a 3000 point impulse response requires nearly an hour. A decade earlier (80286 at 12 MHz), this calculation would have required three days!

The problem of excessive execution time is commonly handled in one of three ways. First, simply keep the signals as short as possible and use integers instead of floating point. If you only need to run the convolution a few times, this will probably be the best trade-off between execution time and programming effort. Second, use a computer designed for DSP. DSP microprocessors are available with multiply-accumulate times of only a few

tens of nanoseconds. This is the route to go if you plan to perform the convolution many times, such as in the design of commercial products.
The third solution is to use a better algorithm for implementing the convolution. Chapter 17 describes a very sophisticated algorithm called *FFT convolution*. FFT convolution produces exactly the same result as the convolution algorithms presented in the last chapter; however, the execution time is dramatically reduced. For signals with thousands of samples, FFT convolution can be hundreds of times faster. The disadvantage is program complexity. Even if you are familiar with the technique, expect to spend several hours getting the program to run.
141
CHAPTER

8 The Discrete Fourier Transform

Fourier analysis is a family of mathematical techniques, all based on decomposing signals into sinusoids. The discrete Fourier transform (DFT) is the family member used with *digitized* signals. This is the first of four chapters on the **real DFT**, a version of the discrete Fourier transform that uses real numbers to represent the input and output signals. The **complex DFT**, a more advanced technique that uses complex numbers, will be discussed in Chapter 31. In this chapter we look at the mathematics and algorithms of the Fourier decomposition, the heart of the DFT.

The Family of Fourier Transform

Fourier analysis is named after **Jean Baptiste Joseph Fourier** (1768-1830), a French mathematician and physicist. (Fourier is pronounced: , and is for@¯e@¯a always capitalized). While many contributed to the field, Fourier is honored for his mathematical discoveries and insight into the practical usefulness of the techniques. Fourier was interested in heat propagation, and presented a paper in 1807 to the Institut de France on the use of sinusoids to represent temperature distributions. The paper contained the controversial claim that any continuous periodic signal could be represented as the sum of properly chosen sinusoidal waves. Among the reviewers were two of history's most famous mathematicians, Joseph Louis Lagrange (1736-1813), and Pierre Simon de Laplace (1749-1827).
While Laplace and the other reviewers voted to publish the paper, Lagrange adamantly protested. For nearly 50 years, Lagrange had insisted that such an approach could not be used to represent signals with *corners*, i.e., discontinuous slopes, such as in square waves. The Institut de France bowed to the prestige of Lagrange, and rejected Fourier's work. It was only after Lagrange died that the paper was finally published, some 15 years later. Luckily, Fourier had other things to keep him busy, political activities, expeditions to Egypt with Napoleon, and trying to avoid the guillotine after the French Revolution (literally!).

Sample number
0 4 8 12 16
-40
-20
0
20
40
60
80
DECOMPOSE
SYNTHESIZE

FIGURE 8-1a
(see facing page)
Amplitude
Who was right? It's a split decision. Lagrange was correct in his assertion that a summation of sinusoids cannot form a signal with a corner. However, you can get *very* close. So close that the difference between the two has *zero energy*. In this sense, Fourier was right, although 18th century science knew little about the concept of energy. This phenomenon now goes by the name: *Gibbs Effect*, and will be discussed in Chapter 11.

Figure 8-1 illustrates how a signal can be decomposed into sine and cosine waves. Figure (a) shows an example signal, 16 points long, running from sample number 0 to 15. Figure (b) shows the Fourier decomposition of this signal, nine cosine waves and nine sine waves, each with a different frequency and amplitude. Although far from obvious, these 18 sinusoids add to produce the waveform in (a). It should be noted that the objection made by Lagrange only applies to *continuous* signals. For *discrete* signals, this decomposition is mathematically exact. There is no difference between the signal in (a) and the *sum* of the signals in (b), just as there is no difference between 7 and 3+4.

Why are sinusoids used instead of, for instance, square or triangular waves? Remember, there are an infinite number of ways that a signal can be decomposed. The goal of decomposition is to end up with something *easier* to deal with than the original signal. For example, impulse decomposition allows signals to be examined one point at a time, leading to the powerful technique of convolution. The component sine and cosine waves are simpler than the original signal because they have a property that the original signal does not have: *sinusoidal fidelity*. As discussed in Chapter 5, a sinusoidal input to a system is guaranteed to produce a sinusoidal output. Only the amplitude and phase of the signal can change; the frequency and wave shape must remain the same. Sinusoids are the only waveform that have this useful property. While square and triangular decompositions are *possible*, there is no general reason for them to be *useful*.

The general term: *Fourier transform*, can be broken into four categories, resulting from the four basic types of signals that can be encountered.

0 2 4 6 8 10 12 14 16
-8
-4
0
4
8
0 2 4 6 8 10 12 14 16
-8
-4
0
4
8
0 2 4 6 8 10 12 14 16
-8
-4
0
4
8
0 2 4 6 8 10 12 14 16
-8
-4
0
4
8
0 2 4 6 8 10 12 14 16
-8
-4
0
4
8
0 2 4 6 8 10 12 14 16

-8
-4
0
4
8
0 2 4 6 8 10 12 14 16
-8
-4
0
4
8
0 2 4 6 8 10 12 14 16
-8
-4
0
4
8
0 2 4 6 8 10 12 14 16
-8
-4
0
4
8
0 2 4 6 8 10 12 14 16
-8
-4
0
4
8
0 2 4 6 8 10 12 14 16
-8
-4
0
4
8
0 2 4 6 8 10 12 14 16
-8
-4
0
4
8
0 2 4 6 8 10 12 14 16
-8
-4
0
4
8
0 2 4 6 8 10 12 14 16
-8
-4
0
4
8
0 2 4 6 8 10 12 14 16
-8
-4
0
4
8
0 2 4 6 8 10 12 14 16
-8
-4
0
4
8
0 2 4 6 8 10 12 14 16
-8
-4
0
4
8
0 2 4 6 8 10 12 14 16
-8
-4
0
4
8

Cosine Waves

Sine Waves

FIGURE 8-1b

Example of Fourier decomposition. A 16 point signal (opposite page) is decomposed into 9 cosine waves and 9 sine waves. The frequency of each sinusoid is fixed; only the amplitude is changed

depending on the shape of the waveform being decomposed.

A signal can be either *continuous* or *discrete*, and it can be either *periodic* or *aperiodic*. The combination of these two features generates the four categories, described below and illustrated in Fig. 8-2.

Aperiodic-Continuous

This includes, for example, decaying exponentials and the Gaussian curve. These signals extend to both positive and negative infinity *without* repeating in a periodic pattern. The Fourier Transform for this type of signal is simply called the **Fourier Transform**.

Periodic-Continuous

Here the examples include: sine waves, square waves, and any waveform that repeats itself in a regular pattern from negative to positive infinity. This version of the Fourier transform is called the **Fourier Series**.

Aperiodic-Discrete

These signals are only defined at discrete points between positive and negative infinity, and do not repeat themselves in a periodic fashion. This type of Fourier transform is called the **Discrete Time Fourier Transform.**

Periodic-Discrete

These are discrete signals that repeat themselves in a periodic fashion from negative to positive infinity. This class of Fourier Transform is sometimes called the Discrete Fourier Series, but is most often called the **Discrete Fourier Transform.**

You might be thinking that the names given to these four types of Fourier transforms are confusing and poorly organized. You're right; the names have evolved rather haphazardly over 200 years. There is nothing you can do but memorize them and move on.

These four classes of signals all extend to positive and negative *infinity*. Hold on, you say! What if you only have a finite number of samples stored in your computer, say a signal formed from 1024 points. Isn't there a version of the Fourier Transform that uses finite length signals? No, there isn't. Sine and cosine waves are *defined* as extending from negative infinity to positive infinity. You cannot use a group of infinitely long signals to synthesize something finite in length. The way around this dilemma is to make the finite data *look like* an infinite length signal. This is done by imagining that the signal has an infinite number of samples on the left and right of the actual points. If all these "imagined" samples have a value of zero, the signal looks *discrete* and *aperiodic*, and the Discrete Time Fourier Transform applies. As an alternative, the imagined samples can be a duplication of the actual 1024 points. In this case, the signal looks discrete and periodic, with a period of 1024 samples. This calls for the Discrete Fourier Transform to be used.

As it turns out, an *infinite* number of sinusoids are required to synthesize a signal that is *aperiodic*. This makes it impossible to calculate the Discrete Time Fourier Transform in a computer algorithm. By elimination, the only

Type of Transform Example Signal

Fourier Transform *signals that are continious and aperiodic*

Fourier Series *signals that are continious and periodic*

Discrete Time Fourier Transform *signals that are discrete and aperiodic*

Discrete Fourier Transform *signals that are discrete and periodic*

FIGURE 8-2
Illustration of the four Fourier transforms. A signal may be continuous or discrete, and it may be periodic or aperiodic. Together these define four possible combinations, each having its own version of the Fourier transform. The names are not well organized; simply memorize them.

type of Fourier transform that can be used in DSP is the DFT. In other words, digital computers can only work with information that is *discrete* and *finite* in length. When you struggle with theoretical issues, grapple with homework problems, and ponder mathematical mysteries, you may find yourself using the first three members of the Fourier transform family. When you sit down to your computer, you will only use the DFT. We will briefly look at these other Fourier transforms in future chapters. For now, concentrate on understanding the Discrete Fourier Transform.

Look back at the example DFT decomposition in Fig. 8-1. On the face of it, it appears to be a 16 point signal being decomposed into 18 sinusoids, each consisting of 16 points. In more formal terms, the 16 point signal, shown in (a), must be viewed as a single period of an infinitely long periodic signal. Likewise, each of the 18 sinusoids, shown in (b), represents a 16 point segment from an infinitely long sinusoid. Does it really matter if we view this as a 16 point signal being synthesized from 16 point sinusoids, or as an infinitely long periodic signal being synthesized from infinitely long sinusoids? The answer is: *usually no, but sometimes, yes.* In upcoming chapters we will encounter properties of the DFT that seem baffling if the signals are viewed as finite, but become obvious when the periodic nature is considered. The key point to understand is that this periodicity is invoked in order to use a *mathematical tool*, i.e., the DFT. It is usually meaningless in terms of where the signal originated or how it was acquired.

Each of the four Fourier Transforms can be subdivided into **real** and **complex** versions. The real version is the simplest, using ordinary numbers and algebra for the synthesis and decomposition. For instance, Fig. 8-1 is an example of the **real DFT**. The complex versions of the four Fourier transforms are immensely more complicated, requiring the use of *complex numbers*. These are numbers such as: , where *j* is equal to 3%4 *j* &1 (electrical engineers use the variable *j*, while mathematicians use the variable, *i*). Complex mathematics can quickly become overwhelming, even to those that specialize in DSP. In fact, a primary goal of this book is to present the fundamentals of DSP *without* the use of complex math, allowing the material to be understood by a wider range of scientists and engineers. The complex Fourier transforms are the realm of those that specialize in DSP, and are willing to sink to their necks in the swamp of mathematics. If you are so inclined, Chapters 30-33 will take you there.

The mathematical term: **transform**, is extensively used in Digital Signal Processing, such as: Fourier transform, Laplace transform, Z transform, Hilbert transform, Discrete Cosine transform, etc. Just what is a transform? To answer this question, remember what a *function* is. A function is an algorithm or procedure that changes one value into another value. For example, is a function. You pick some value for *x*, plug it into the *y* ' 2*x*%1 equation, and out pops a value for *y*. Functions can also change *several* values into a single value, such as: , where and *c* are *y* ' 2*a* % 3*b* % 4*c* *a*, *b*, changed into *y*.

Transforms are a direct extension of this, allowing both the input and output to have *multiple* values. Suppose you have a signal composed of 100 samples. If you devise some equation, algorithm, or procedure for changing these 100 samples into another 100 samples, you have yourself a transform. If you think

it is useful enough, you have the perfect right to attach your last name to it and expound its merits to your colleagues. (This works best if you are an eminent 18th century French mathematician). Transforms are not limited to any specific type or number of data. For example, you might have 100 samples of discrete data for the input and 200 samples of discrete data for the output. Likewise, you might have a continuous signal for the input and a continuous signal for the output. Mixed signals are also allowed, discrete in and continuous out, and vice versa. In short, a transform is any fixed procedure that changes one chunk of data into another chunk of data. Let's see how this applies to the topic at hand: the Discrete Fourier transform.

Notation and Format of the Real DFT

As shown in Fig. 8-3, the discrete Fourier transform changes an *N* point input signal into two point output signals. The input signal contains the *N*/2%1 signal being decomposed, while the two output signals contain the *amplitudes* of the component sine and cosine waves (scaled in a way we will discuss shortly). The input signal is said to be in the **time domain**. This is because the most common type of signal entering the DFT is composed of

FIGURE 8-3

DFT terminology. In the time domain, consists of *N* points running from 0 to . In the frequency domain, *x*[ ] *N*&1 the DFT produces two signals, the real part, written: , and the imaginary part, written: . Each of *ReX*[ ] *Im X*[ ] these frequency domain signals are points long, and run from 0 to . The Forward DFT transforms from *N*/2%1 *N*/2 the time domain to the frequency domain, while the Inverse DFT transforms from the frequency domain to the time domain. (Take note: this figure describes the **real DFT**. The **complex DFT**, discussed in Chapter 31, changes *N* complex points into another set of *N* complex points).

samples taken at regular intervals of *time*. Of course, any kind of sampled data can be fed into the DFT, regardless of how it was acquired. When you see the term "time domain" in Fourier analysis, it may actually refer to samples taken over time, or it might be a general reference to any discrete signal that is being decomposed. The term **frequency domain** is used to describe the amplitudes of the sine and cosine waves (including the special scaling we promised to explain).

The frequency domain contains exactly the same information as the time domain, just in a different form. If you know one domain, you can calculate the other. Given the time domain signal, the process of calculating the frequency domain is called **decomposition**, **analysis,** the **forward DFT**, or simply, **the DFT**. If you know the frequency domain, calculation of the time domain is called **synthesis**, or the **inverse DFT**. Both synthesis and analysis can be represented in equation form and computer algorithms.

The number of samples in the time domain is usually represented by the **variable *N***. While *N* can be any positive integer, a power of two is usually chosen, i.e., 128, 256, 512, 1024, etc. There are two reasons for this. First, digital data storage uses binary addressing, making powers of two a natural

signal length. Second, the most efficient algorithm for calculating the DFT, the Fast Fourier Transform (FFT), usually operates with *N* that is a power of two. Typically, *N* is selected between 32 and 4096. In most cases, the samples run from 0 to , rather than 1 to *N*. *N*&1

Standard DSP notation uses **lower case letters** to represent time domain signals, such as , , and . The corresponding **upper case letters** are *x*[ ] *y*[ ] *z*[ ]

used to represent their frequency domains, that is, , , and . For *X*[ ] *Y*[ ] *Z*[ ]

illustration, assume an *N* point time domain signal is contained in . The *x*[ ]

frequency domain of this signal is called , and consists of two parts, each *X*[ ]

an array of samples. These are called the **Real part of** , written *N*/2%1 ***X*[ ]**

as: , and the **Imaginary part of** , written as: . The values ***Re X*[ ] *X*[ ] *Im X*[ ]**

in are the amplitudes of the cosine waves, while the values in *Re X*[ ] *Im X*[ ]

are the amplitudes of the sine waves (not worrying about the scaling factors for the moment). Just as the time domain runs from to , the frequency *x*[0] *x*[*N*&1]

domain signals run from to , and from to . *ReX*[0] *ReX*[*N*/2] *ImX*[0] *Im X*[*N*/2]

Study these notations carefully; they are critical to understanding the equations in DSP. Unfortunately, some computer languages don't distinguish between lower and upper case, making the variable names up to the individual programmer. The programs in this book use the array XX[ ] to hold the time domain signal, and the arrays REX[ ] and IMX[ ] to hold the frequency domain signals.

The names *real part* and *imaginary part* originate from the complex DFT, where they are used to distinguish between *real* and *imaginary* numbers. Nothing so complicated is required for the real DFT. Until you get to Chapter 31, simply think that "real part" means the *cosine wave amplitudes,* while "imaginary part" means the *sine wave amplitudes*. Don't let these suggestive names mislead you; everything here uses ordinary numbers.

Likewise, don't be misled by the *lengths* of the frequency domain signals. It is common in the DSP literature to see statements such as: "The DFT changes an *N* point time domain signal into an *N* point frequency domain signal." This is referring to the *complex DFT*, where each "point" is a complex number (consisting of real and imaginary parts). For now, focus on learning the real DFT, the difficult math will come soon enough.

The Frequency Domain's Independent Variable

Figure 8-4 shows an example DFT with . The time domain signal is *N* ' 128

contained in the array: to . The frequency domain signals are *x*[0] *x*[127]

contained in the two arrays: to , and to . *ReX*[0] *ReX*[64] *Im X*[0] *Im X*[64]

Notice that 128 points in the time domain corresponds to 65 points in each of the frequency domain signals, with the frequency indexes running from 0 to 64. That is, *N* points in the time domain corresponds to points in the *N*/2%1

frequency domain (not points). Forgetting about this extra point is a *N*/2

common bug in DFT programs.

The horizontal axis of the frequency domain can be referred to in **four different ways**, all of which are common in DSP. In the first method, the horizontal axis is labeled from 0 to 64, corresponding to the 0 to *N*/2

samples in the arrays. When this labeling is used, the index for the

frequency domain is an integer, for example, and , where *k Re X*[*k*] *Im X*[*k*]

runs from 0 to in steps of one. Programmers like this method because *N*/2

it is how they write code, using an index to access array locations. This notation is used in Fig. 8-4b.

Sample number

0 16 32 48 64 80 96 112 128
-2
-1
0
1
2
a. x[ ]
127
Frequency (sample number)
0 16 32 48 64
-8
-4
0
4
8
b. Re X[ ]
Frequency (fraction of sampling rate)
0 0.1 0.2 0.3 0.4 0.5
-8
-4
0
4
8
c. Im X[ ]
Frequency Domain Time Domain
FIGURE 8-4
Example of the DFT. The DFT converts the time domain signal, , into the frequency $x[\ ]$ domain signals, and . The $ReX[\ ]$ $Im\ X[\ ]$ horizontal axis of the frequency domain can be labeled in one of three ways: (1) as an array index that runs between 0 and , (2) as a $N/2$ fraction of the sampling frequency, running between 0 and 0.5, (3) as a natural frequency, running between 0 and B. In the example shown here, (b) uses the first method, while (c) use the second method.
Amplitude Amplitude
Amplitude
In the second method, used in (c), the horizontal axis is labeled as a *fraction of the sampling rate*. This means that the values along the horizonal axis always run between 0 and 0.5, since discrete data can only contain frequencies between DC and one-half the sampling rate. The index used with this notation is *f*, for frequency. The real and imaginary parts are written: and $ReX\,[f\,]$ $ImX\,[f\,]$ $f\ N/2\%1$, where takes on equally spaced values between 0 and 0.5. To convert from the first notation, *k*, to the second notation, , divide the f horizontal axis by N. That is, . Most of the graphs in this book use this $f\ '\ k/N$ second method, reinforcing that discrete signals only contain frequencies between 0 and 0.5 of the sampling rate.

The third style is similar to the second, except the horizontal axis is multiplied by 2B. The index used with this labeling is T, a lower case Greek *omega*. In this notation, the real and imaginary parts are written: and , where T takes on equally spaced values $ReX\,[T]$ $ImX\,[T]$ $N/2\%1$ between 0 and B. The parameter, T, is called the **natural frequency**, and has the units of **radians**. This is based on the idea that there are 2B radians in a circle. Mathematicians like this method because it makes the equations shorter. For instance, consider how a cosine wave is written in each of these first three notations: using *k*: , using : $c\,[n\,]\ '\ \cos(2Bkn\ /N)\ f$, and using T: . $c\,[n]\ '\ \cos(2Bfn)$ $c\,[n\,]\ '\ \cos(Tn)$

$$c_k\,[i\,]\ '\ \cos(\ 2Bki\ /N)$$

EQUATION 8-1
Equations for the DFT basis functions. In

these equations, and are the $c_k[\,i]$ $s_k[\,i]$ cosine and sine waves, each N points in length, running from to . The i ' 0 N&1 parameter, k, determines the frequency of the wave. In an N point DFT, k takes on values between 0 and . N/2

$s_k[\,i\,]$ ' sin (2Bki /N)

The fourth method is to label the horizontal axis in terms of the analog frequencies used in a *particular* application. For instance, if the system being examined has a sampling rate of 10 kHz (i.e., 10,000 samples per second), graphs of the frequency domain would run from 0 to 5 kHz. This method has the advantage of presenting the frequency data in terms of a *real world* meaning. The disadvantage is that it is tied to a particular sampling rate, and is therefore not applicable to general DSP algorithm development, such as designing digital filters.

All of these four notations are used in DSP, and you need to become comfortable with converting between them. This includes both graphs and mathematical equations. To find which notation is being used, look at the independent variable and its range of values. You should find one of four notations: k (or some other integer index), running from 0 to ; f, running N/2 from 0 to 0.5; T, running from 0 to B; or a frequency expressed in hertz, running from DC to one-half of an actual sampling rate.

DFT Basis Functions

The sine and cosine waves used in the DFT are commonly called the DFT **basis functions**. In other words, the output of the DFT is a set of numbers that represent amplitudes. The basis functions are a set of sine and cosine waves with *unity* amplitude. If you assign each amplitude (the frequency domain) to the proper sine or cosine wave (the basis functions), the result is a set of *scaled* sine and cosine waves that can be added to form the time domain signal.

The DFT basis functions are generated from the equations:

where: is the cosine wave for the amplitude held in , and is $c_k[\,]$ $ReX[k]$ $s_k[\,]$ the sine wave for the amplitude held in . For example, Fig. 8-5 shows $Im\ X[k]$ some of the 17 sine and 17 cosine waves used in an point DFT. Since N ' 32 these sinusoids add to form the input signal, they must be the same *length* as the input signal. In this case, each has 32 points running from to 31. The i ' 0 parameter, k, sets the frequency of each sinusoid. In particular, is the $c_1[\,]$ cosine wave that makes *one* complete cycle in N points, is the cosine $c_5[\,]$ wave that makes *five* complete cycles in N points, etc. This is an important concept in understanding the basis functions; the frequency parameter, k, is equal to the number of complete cycles that occur over the N points of the signal.

Sample number
0 8 16 24 32
-2
-1
0
1
2
a. $c_0[\,]$
Sample number
0 8 16 24 32
-2
-1
0
1
2
c. $c_2[\,]$

Sample number
0 8 16 24 32
-2
-1
0
1
2
e. $c_{10}[\ ]$
Sample number
0 8 16 24 32
-2
-1
0
1
2
f. $s_{10}[\ ]$
Sample number
0 8 16 24 32
-2
-1
0
1
2
h. $s_{16}[\ ]$
Sample number
0 8 16 24 32
-2
-1
0
1
2
b. $s_0[\ ]$
Sample number
0 8 16 24 32
-2
-1
0
1
2
d. $s_2[\ ]$
Sample number
0 8 16 24 32
-2
-1
0
1
2
g. $c_{16}[\ ]$
FIGURE 8-5
DFT basis functions. A 32 point DFT has 17 discrete cosine waves and 17 discrete sine waves for its basis functions. Eight of these are shown in this figure. These are discrete signals; the continuous lines are shown in these graphs only to help the reader's eye follow the waveforms.
Amplitude
Amplitude
Amplitude Amplitude
Amplitude
Amplitude
Amplitude Amplitude

EQUATION 8-2
The synthesis equation. In this relation, is the signal being $x[i]$ synthesized, with the index, i, running from 0 to . N&1 $Re\,\bar{X}[k]$ and hold the amplitudes of the cosine and sine waves, $Im\,\bar{X}[k]$ respectively, with k running from 0 to . Equation 8-3 provides $N/2$ the normalization to change this equation into the inverse DFT.

$$x[i] = \sum_{k=0}^{N/2} Re\,\bar{X}[k]\cos(2\pi ki/N) + \sum_{k=0}^{N/2} Im\,\bar{X}[k]\sin(2\pi ki/N)$$

Let's look at several of these basis functions in detail. Figure (a) shows the cosine wave . This is a cosine wave of zero frequency, which is a constant $c_0[\ ]$

value of one. This means that holds the average value of all the points *ReX*[0] in the time domain signal. In electronics, it would be said that holds *ReX*[0] the **DC offset**. The sine wave of zero frequency, , is shown in (b), a s_0[ ] signal composed of all *zeros*. Since this can not affect the time domain signal being synthesized, the value of is *irrelevant*, and always set to zero. *Im X*[0] More about this shortly.

Figures (c) & (d) show & , the sinusoids that complete *two* cycles in c_2[ ] s_2[ ] the *N* points. These correspond to & , respectively. Likewise, *Re X*[2] *Im X*[2] (e) & (f) show & , the sinusoids that complete *ten* cycles in the *N* c_{10}[ ] s_{10}[ ] points. These sinusoids correspond to the amplitudes held in the arrays & . The problem is, the samples in (e) and (f) no longer *ReX*[10] *Im X*[10] *look* like sine and cosine waves. If the continuous curves were not present in these graphs, you would have a difficult time even detecting the pattern of the waveforms. This may make you a little uneasy, but don't worry about it. From a mathematical point of view, these samples do form discrete sinusoids, even if your eye cannot follow the pattern.

The highest frequencies in the basis functions are shown in (g) and (h). These are & , or in this example, & . The discrete cosine $c_{N/2}$[ ] $s_{N/2}$[ ] c_{16}[ ] s_{16}[ ] wave alternates in value between 1 and -1, which can be interpreted as sampling a continuous sinusoid at the *peaks*. In contrast, the discrete sine wave contains all zeros, resulting from sampling at the *zero crossings*. This makes the value of the same as , always equal to zero, and not *Im X*[*N*/2] *Im X*[0] affecting the synthesis of the time domain signal.

Here's a puzzle: If there are *N* samples entering the DFT, and samples *N*%2 exiting, where did the extra information come from? The answer: two of the output samples contain *no* information, allowing the other *N* samples to be fully independent. As you might have guessed, the points that carry no information are and , the samples that always have a value of zero. *Im X*[0] *Im X*[*N*/2]

Synthesis, Calculating the Inverse DFT

Pulling together everything said so far, we can write the **synthesis equation**:

EQUATIONS 8-3

Conversion between the sinusoidal amplitudes and the frequency domain values. In these equations, $Re\,\bar{X}[k]$ and hold the amplitudes of $Im\,\bar{X}[k]$ the cosine and sine waves needed for synthesis, while and *Re X*[*k*] *Im X*[*k*] hold the real and imaginary parts of the frequency domain. As usual, *N* is the number of points in the time domain signal, and *k* is an index that runs from 0 to *N*/2.

$$Re\,\bar{X}[k] \text{ ' } ReX[k]$$

$$\frac{N}{2}$$

$$Im\,\bar{X}[k] \text{ ' \& } ImX[k]$$

$$\frac{N}{2}$$

$$Re\,\bar{X}[0] \text{ ' } ReX[0]$$

$$N$$

$$Re\,\bar{X}[N/2] \text{ ' } ReX[N/2]$$

$$N$$

except for two special cases:

In words, any *N* point signal, , can be created by adding cosine *x*[*i*] *N*/2% 1 waves and sine waves. The amplitudes of the cosine and sine waves *N*/2% 1

are held in the arrays and , respectively. The synthesis $Im\ \bar{X}[k]$ $Re\ \bar{X}[k]$ equation multiplies these amplitudes by the basis functions to create a set of scaled sine and cosine waves. Adding the scaled sine and cosine waves produces the time domain signal, . $x[i]$

In Eq. 8-2, the arrays are called and , rather than and $Im\ \bar{X}[k]$ $Re\ \bar{X}[k]$ $Im\ X[k]$. This is because the *amplitudes needed for synthesis* (called in this $Re\ X[k]$ discussion: and), are slightly different from the *frequency* $Im\ \bar{X}[k]$ $Re\ \bar{X}[k]$ *domain of a signal* (denoted by: and). This is the scaling $Im\ X[k]$ $Re\ X[k]$ factor issue we referred to earlier. Although the conversion is only a simple normalization, it is a common bug in computer programs. Look out for it! In equation form, the conversion between the two is given by:

Suppose you are given a frequency domain representation, and asked to synthesize the corresponding time domain signal. To start, you must find the amplitudes of the sine and cosine waves. In other words, given and $Im\ X[k]$, you must find and . Equation 8-3 shows this in a $Re\ X[k]$ $Im\ \bar{X}[k]$ $Re\ \bar{X}[k]$ mathematical form. To do this in a computer program, three actions must be taken. First, divide all the values in the frequency domain by . Second, $N/2$ change the sign of all the imaginary values. Third, divide the first and last samples in the real part, and , by two. This provides the $ReX[0]$ $ReX[N/2]$ amplitudes needed for the synthesis described by Eq. 8-2. Taken together, Eqs. 8-2 and 8-3 *define* the inverse DFT.

The entire Inverse DFT is shown in the computer program listed in Table 8-1. There are two ways that the synthesis (Eq. 8-2) can be programmed, and both are shown. In the first method, each of the scaled sinusoids are generated one at a time and added to an accumulation array, which ends up becoming the time domain signal. In the second method, each sample in the time domain signal is calculated one at a time, as the sum of all the


```
100 'THE INVERSE DISCRETE FOURIER TRANSFORM
110 'The time domain signal, held in XX[ ], is calculated from the frequency domain signals,
120 'held in REX[ ] and IMX[ ].
130 '
140 DIM XX[511] 'XX[ ] holds the time domain signal
150 DIM REX[256] 'REX[ ] holds the real part of the frequency domain
160 DIM IMX[256] 'IMX[ ] holds the imaginary part of the frequency domain
170 '
180 PI = 3.14159265 'Set the constant, PI
190 N% = 512 'N% is the number of points in XX[ ]
200 '
210 GOSUB XXXX 'Mythical subroutine to load data into REX[ ] and IMX[ ]
220 '
230
240 ' 'Find the cosine and sine wave amplitudes using Eq. 8-3
250 FOR K% = 0 TO 256
260 REX[K%] = REX[K%] / (N%/2)
270 IMX[K%] = -IMX[K%] / (N%/2)
280 NEXT K%
290 '
300 REX[0] = REX[0] / 2
310 REX[256] = REX[256] / 2
320 '
330 '
340 FOR I% = 0 TO 511 'Zero XX[ ] so it can be used as an accumulator
350 XX[I%] = 0
360 NEXT I%
370 '
380 ' Eq. 8-2 SYNTHESIS METHOD #1. Loop through each
```

```
390 ' frequency generating the entire length of the sine and cosine
400 ' waves, and add them to the accumulator signal, XX[ ]
410 '
420 FOR K% = 0 TO 256 'K% loops through each sample in REX[ ] and IMX[ ]
430 FOR I% = 0 TO 511 'I% loops through each sample in XX[ ]
440 '
450 XX[I%] = XX[I%] + REX[K%] * COS(2*PI*K%*I%/N%)
460 XX[I%] = XX[I%] + IMX[K%] * SIN(2*PI*K%*I%/N%)
470 '
480 NEXT I%
490 NEXT K%
500 '
510 END
```

Alternate code for lines 380 to 510

```
380 ' Eq. 8-2 SYNTHESIS METHOD #2. Loop through each
390 ' sample in the time domain, and sum the corresponding
400 ' samples from each cosine and sine wave
410 '
420 FOR I% = 0 TO 511 'I% loops through each sample in XX[ ]
430 FOR K% = 0 TO 256 'K% loops through each sample in REX[ ] and IMX[ ]
440 '
450 XX[I%] = XX[I%] + REX[K%] * COS(2*PI*K%*I%/N%)
460 XX[I%] = XX[I%] + IMX[K%] * SIN(2*PI*K%*I%/N%)
470 '
480 NEXT K%
490 NEXT I%
500 '
510 END
```

TABLE 8-1

Sample number

0 8 16 24 32

0

10

20

30

40

50

a. The time domain signal

Frequency sample number

0 4 8 12 16

0

10

20

30

40

50

b. Re X[ ] (the frequency domain)

Frequency sample number

0 4 8 12 16

0.0

1.0

2.0

3.0

c. Re X[ ] (cosine wave amplitudes)

Frequency Domain Time Domain

FIGURE 8-6

Example of the Inverse DFT. Figure (a) shows an example time domain signal, an impulse at sample zero with an amplitude of 32. Figure (b) shows the real part of the frequency domain of this signal, a constant value of 32. The imaginary part of the frequency domain (not shown) is composed of all zeros. Figure(c) shows the amplitudes of the cosine waves needed to reconstruct (a) using Eq. 8-2. The values in (c) are found from (b) by using Eq. 8-3.

I DFT

Amplitude

Amplitude

Amplitude

Eq. 8.2 Eq. 8.3

corresponding samples in the cosine and sine waves. Both methods produce the same result. The difference between these two programs is very minor; the inner and outer loops are swapped during the synthesis.

Figure 8-6 illustrates the operation of the Inverse DFT, and the slight differences between the frequency domain and the amplitudes needed for synthesis. Figure 8-6a is an example signal we wish to synthesize, an impulse at sample zero with an amplitude of 32. Figure 8-6b shows the frequency domain representation of this signal. The real part of the frequency domain is a constant value of 32. The imaginary part (not shown) is composed of all zeros. As discussed in the next chapter, this is an important DFT *pair*: an impulse in the time domain corresponds to a constant value in the frequency domain. For now, the important point is that (b) is the *DFT* of (a), and (a) is the *Inverse DFT* of (b).

Frequency sample number
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
0
1
2
3
4
5
6
7
8
9
10
1/N
width:
1/N
width:
2/N
width:
1/N

FIGURE 8-7
The bandwidth of frequency domain samples. Each sample in the frequency domain can be thought of as being contained in a frequency band of width 2/*N*, expressed as a fraction of the total bandwidth. An exception to this is the first and last samples, which have a bandwidth only one-half this wide, 1/*N*.

Amplitude

Equation 8-3 is used to convert the frequency domain signal, (b), into the amplitudes of the cosine waves, (c). As shown, all of the cosine waves have an amplitude of *two*, except for samples 0 and 16, which have a value of *one*. The amplitudes of the sine waves are not shown in this example because they have a value of zero, and therefore provide no contribution. The synthesis equation, Eq. 8-2, is then used to convert the amplitudes of the cosine waves, (b), into the time domain signal, (a).

This describes *how* the frequency domain is different from the sinusoidal amplitudes, but it doesn't explain *why* it is different. The difference occurs because the frequency domain is defined as a **spectral density**. Figure 8-7 shows how this works. The example in this figure is the real part of the frequency domain of a 32 point signal. As you should expect, the samples run from 0 to 16, representing 17 frequencies equally spaced between 0 and 1/2 of the sampling rate. S*pectral density* describes how much signal (amplitude) is present *per unit of bandwidth*. To convert the sinusoidal amplitudes into a spectral density, divide each amplitude by the bandwidth represented by each amplitude. This brings up the next issue: how do we determine the bandwidth of each of the discrete frequencies in the frequency domain?

As shown in the figure, the bandwidth can be defined by drawing dividing lines between the samples. For instance, sample number 5 occurs in the band between 4.5 and 5.5; sample number 6 occurs in the band between 5.5 and 6.5, etc. Expressed as a fraction of the total bandwidth (i.e.,), the $N/2$ bandwidth of each sample is . An exception to this is the samples on $2/N$ each end, which have one-half of this bandwidth, . This accounts for $1/N$ the scaling factor between the sinusoidal amplitudes and frequency $2/N$ domain, as well as the additional factor of two needed for the first and last samples.

Why the negation of the imaginary part? This is done solely to make the *real DFT* consistent with its big brother, the *complex DFT*. In Chapter 29 we will show that it is necessary to make the mathematics of the complex DFT work. When dealing only with the real DFT, many authors do not include this negation. For that matter, many authors do not even include

the scaling factor. Be prepared to find both of these missing in some $2/N$ discussions. They are included here for a tremendously important reason: The most efficient way to calculate the DFT is through the Fast Fourier Transform (FFT) algorithm, presented in Chapter 12. The FFT generates a frequency domain defined according to Eq. 8-2 and 8-3. If you start messing with these normalization factors, your programs containing the FFT are not going to work as expected.

Analysis, Calculating the DFT

The DFT can be calculated in three completely different ways. First, the problem can be approached as a set of *simultaneous equations*. This method is useful for understanding the DFT, but it is too inefficient to be of practical use. The second method brings in an idea from the last chapter: *correlation.* This is based on detecting a known waveform in another signal. The third method, called the Fast Fourier Transform (FFT), is an ingenious algorithm that decomposes a DFT with N points, into N DFTs each with a single point. The FFT is typically hundreds of times faster than the other methods. The first two methods are discussed here, while the FFT is the topic of Chapter 12. It is important to remember that all three of these methods produce an identical output. Which should you use? In actual practice, *correlation* is the preferred technique if the DFT has less than about 32 points, otherwise the *FFT* is used.

DFT by Simultaneous Equations

Think about the DFT calculation in the following way. You are given N values from the time domain, and asked to calculate the N values of the frequency domain (ignoring the two frequency domain values that you know must be zero). Basic algebra provides the answer: to solve for N unknowns, you must be able to write N linearly independent equations. To do this, take the first sample from each sinusoid and add them together. The sum must be equal to the first sample in the time domain signal, thus providing the first equation. Likewise, an equation can be written for each of the remaining points in the time domain signal, resulting in the required N equations. The solution can then be found by using established methods for solving simultaneous equations, such as Gauss Elimination. Unfortunately, this method requires a tremendous number of calculations, and is virtually never used in DSP. However, it is important for another reason, it shows *why* it is possible to decompose a signal into sinusoids, how *many* sinusoids are needed, and that the basis functions must be linearly independent (more about this shortly).

DFT by Correlation

Let's move on to a better way, the *standard way* of calculating the DFT. An example will show how this method works. Suppose we are trying to calculate the DFT of a 64 point signal. This means we need to calculate the 33 points in the real part, and the 33 points in the imaginary part of the frequency domain. In this example we will only show how to calculate a single sample, , i.e., the amplitude of the sine wave that makes three complete cycles *Im X*[3]

EQUATION 8-4

The analysis equations for calculating the DFT. In these equations, is the $x[i]$ time domain signal being analyzed, and & are the frequency $Re\ X[k]$ $Im\ X[k]$ domain signals being calculated. The index i runs from 0 to , while the N&1 index k runs from 0 to . $N/2$

$$ReX[k] \text{ ' j } \sum_{i \text{ '} 0}^{N \text{ \& } 1} x\ [i\ ] \cos(2\text{B}k\ i\ /N\ )$$

$$ImX[k] \text{ ' \& j } \sum_{i \text{ '} 0}^{N \text{ \& } 1} x\ [i\ ] \sin\ (2\text{B}k\ i\ /N\ )$$

between point 0 and point 63. All of the other frequency domain values are calculated in a similar manner.

Figure 8-8 illustrates using correlation to calculate . Figures (a) and *Im X*[3] (b) show two example time domain signals, called: and , $x1[\ ]$ $x2[\ ]$ respectively. The first signal, , is composed of nothing but a sine wave $x1[\ ]$ that makes three cycles between points 0 and 63. In contrast, is $x2[\ ]$ composed of several sine and cosine waves, *none* of which make three cycles between points 0 and 63. These two signals illustrate what the algorithm for calculating must do. When fed , the algorithm must produce a *Im X*[3] $x1[\ ]$ value of 32, the amplitude of the sine wave present in the signal (modified by the scaling factors of Eq. 8-3). In comparison, when the algorithm is fed the other signal, , a value of zero must be produced, indicating that this $x2[\ ]$ particular sine wave is not present in this signal.

The concept of correlation was introduced in Chapter 7. As you recall, to detect a known waveform contained in another signal, multiply the two signals and add all the points in the resulting signal. The single number that results from this procedure is a measure of how similar the two signals are. Figure 8-8 illustrates this approach. Figures (c) and (d) both display the signal we are looking for, a sine wave that makes 3 cycles between samples 0 and 63. Figure (e) shows the result of multiplying (a) and (c). Likewise, (f) shows the result of multiplying (b) and (d). The sum of all the points in (e) is 32, while the sum of all the points in (f) is zero, showing we have found the desired algorithm.

The other samples in the frequency domain are calculated in the same way. This procedure is formalized in the *analysis equation*, the mathematical way to calculate the frequency domain from the time domain:

In words, each sample in the frequency domain is found by multiplying the time domain signal by the sine or cosine wave being looked for, and adding the resulting points. If someone asks you what you are doing, say with confidence: "I am correlating the input signal with each basis function." Table 8-2 shows a computer program for calculating the DFT in this way.

The analysis equation does *not* require special handling of the first and last points, as did the synthesis equation. There is, however, a negative sign in the

imaginary part in Eq. 8-4. Just as before, this negative sign makes the *real DFT* consistent with the *complex DFT*, and is not always included.

Sample number
0 16 32 48 64
-2
-1
0
1
2
a. x1[ ], signal being analyzed
Sample number
0 16 32 48 64
-2
-1
0
1
2
b. x2[ ], signal being analyzed
Sample number
0 16 32 48 64
-2
-1
0
1
2
d. s_3[ ], basis function being sought
Sample number
0 16 32 48 64
-2
-1
0
1
2
c. s_3[ ], basis function being sought
Sample number
0 16 32 48 64
-2
-1
0
1
2
e. x1[ ] × s_3[ ]
Sample number
0 16 32 48 64
-2
-1
0
1
2
f. x2[ ] × s_3[ ]
Amplitude
Amplitude
Amplitude
Amplitude
Example 2 Example 1
FIGURE 8-8
Two example signals, (a) and (b), are analyzed for containing the specific basis function shown in (c) and (d). Figures (e) and (f) show the result of multiplying each example signal by the basis function. Figure (e) has an average of 0.5, indicating that contains the basis function with an amplitude of 1.0. Conversely, (f) has x_1[ ] a zero average, indicating that does not contain the basis function. x_2[ ]
Amplitude
Amplitude
In order for this correlation algorithm to work, the basis functions must have an interesting property: each of them must be completely *uncorrelated* with all of the others. This means that if you multiply any two of the basis functions, the sum of the resulting points will be equal to zero. Basis functions that have this property are called **orthognal**. Many other

```
100 'THE DISCRETE FOURIER TRANSFORM
110 'The frequency domain signals, held in REX[ ] and IMX[ ], are calculated from
120 'the time domain signal, held in XX[ ].
130 '
140 DIM XX[511] 'XX[ ] holds the time domain signal
150 DIM REX[256] 'REX[ ] holds the real part of the frequency domain
160 DIM IMX[256] 'IMX[ ] holds the imaginary part of the frequency domain
170 '
180 PI = 3.14159265 'Set the constant, PI
190 N% = 512 'N% is the number of points in XX[ ]
200 '
210 GOSUB XXXX 'Mythical subroutine to load data into XX[ ]
220 '
230 '
240 FOR K% = 0 TO 256 'Zero REX[ ] & IMX[ ] so they can be used as accumulators
250 REX[K%] = 0
260 IMX[K%] = 0
270 NEXT K%
280 '
290 ' 'Correlate XX[ ] with the cosine and sine waves, Eq. 8-4
300 '
310 FOR K% = 0 TO 256 'K% loops through each sample in REX[ ] and IMX[ ]
320 FOR I% = 0 TO 511 'I% loops through each sample in XX[ ]
330 '
340 REX[K%] = REX[K%] + XX[I%] * COS(2*PI*K%*I%/N%)
350 IMX[K%] = IMX[K%] - XX[I%] * SIN(2*PI*K%*I%/N%)
360 '
370 NEXT I%
380 NEXT K%
390 '
400 END
```

TABLE 8-2

orthognal basis functions exist, including: square waves, triangle waves, impulses, etc. Signals can be decomposed into these other orthognal basis functions using correlation, just as done here with sinusoids. This is not to suggest that this is *useful*, only that it is *possible*.

As previously shown in Table 8-1, the *Inverse DFT* has two ways to be implemented in a computer program. This difference involves *swapping* the inner and outer loops during the synthesis. While this does not change the output of the program, it makes a difference in how you *view* what is being done. The *DFT* program in Table 8-2 can also be changed in this fashion, by swapping the inner and outer loops in lines 310 to 380. Just as before, the output of the program is the same, but the way you *think* about the calculation is different. (These two different ways of viewing the DFT and inverse DFT could be described as "input side" and "output side" algorithms, just as for convolution).

As the program in Table 8-2 is written, it describes how an individual sample in the frequency domain is affected by all of the samples in the time domain. That is, the program calculates each of the values in the frequency domain in succession, not as a group. When the inner and outer loops are exchanged, the program loops through each sample in the time domain, calculating the

contribution of that point to the frequency domain. The overall frequency domain is found by adding the contributions from the individual time domain points. This brings up our next question: what kind of contribution does an individual sample in the time domain provide to the frequency domain? The answer is contained in an interesting aspect of Fourier analysis called *duality*.

Duality

The synthesis and analysis equations (Eqs. 8-2 and 8-4) are strikingly similar. To move from one domain to the other, the known values are multiplied by the basis functions, and the resulting products added. The fact that the *DFT* and the *Inverse DFT* use this same mathematical approach is really quite remarkable, considering the totally different way we arrived at the two procedures. In fact, the only significant difference between the two equations is a result of the time domain being *one* signal of *N* points, while the frequency domain is *two* signals of points. *N*/2% 1 As discussed in later chapters, the *complex DFT* expresses both the time and the frequency domains as complex signals of *N* points each. This makes the two domains completely symmetrical, and the equations for moving between them virtually *identical*.

This symmetry between the time and frequency domains is called **duality**, and gives rise to many interesting properties. For example, a single point in the frequency domain corresponds to a sinusoid in the time domain. By duality, the inverse is also true, a single point in the time domain corresponds to a sinusoid in the frequency domain. As another example, convolution in the time domain corresponds to multiplication in the frequency domain. By duality, the reverse is also true: convolution in the frequency domain corresponds to multiplication in the time domain. These and other duality relationships are discussed in more detail in Chapters 10 and 11.

Polar Notation

As it has been described so far, the frequency domain is a group of amplitudes of cosine and sine waves (with slight scaling modifications). This is called **rectangular** notation. Alternatively, the frequency domain can be expressed in **polar** form. In this notation, & are *ReX*[ ] *Im X*[ ] replaced with two other arrays, called the **Magnitude of** , written in ***X*[ ]** equations as: , and the **Phase of** , written as: . ***Mag X*[ ] *X*[ ] *Phase X*[ ]** The magnitude and phase are a pair-for-pair replacement for the real and imaginary parts. For example, and are calculated *Mag X*[0] *Phase X*[0] using only and . Likewise, and are *ReX*[0] *Im X*[0] *Mag X*[14] *Phase X*[14] calculated using only and , and so forth. To understand *ReX*[14] *Im X*[14] the conversion, consider what happens when you add a cosine wave and a sine wave of the same frequency. The result is a cosine wave of the same

EQUATION 8-5

The addition of a cosine and sine wave results in a cosine wave with a different amplitude and phase shift. The information contained in *A* & *B* is transferred to two other variables, *M* and 2.

$$A\cos(x) \% B\sin(x) ' M\cos(x \% 2)$$

FIGURE 8-9

Rectangular-to-polar conversion. The addition of a cosine wave and a sine wave (of the same frequency) follows the same mathematics as the addition of simple vectors.

B

A

M

2

M = (A + B)
2 = arctan(B/A)
2 2 1/2
EQUATION 8-6
Rectangular-to-polar conversion. The rectangular representation of the frequency domain, and , is *ReX*[*k*] *Im X*[*k*] changed into the polar form, *Mag X*[*k*] and . *Phase X*[*k*]

$$MagX[k] = (ReX[k]^2 + ImX[k]^2)^{1/2}$$

$$PhaseX[k] = \arctan \frac{ImX[k]}{ReX[k]}$$

EQUATION 8-7
Polar-to-rectangular conversion. The two arrays, and , are *Mag X*[*k*] *Phase X*[*k*] converted into and . *ReX*[*k*] *Im X*[*k*]

$$ReX[k] = MagX[k] \cos(PhaseX[k])$$

$$ImX[k] = MagX[k] \sin(PhaseX[k])$$

frequency, but with a new amplitude and a new phase shift. In equation form, the two representations are related:

The important point is that no information is lost in this process; given one representation you can calculate the other. In other words, the information contained in the amplitudes *A* and *B*, is also contained in the variables *M* and 2. Although this equation involves sine and cosine waves, it follows the same conversion equations as do simple vectors. Figure 8-9 shows the analogous vector representation of how the two variables, *A* and *B*, can be viewed in a rectangular coordinate system, while *M* and 2 are parameters in polar coordinates.

In polar notation, holds the amplitude of the cosine wave (*M* in Eq. *Mag X*[ ] 8-5 and Fig. 8-9), while holds the phase angle of the cosine wave *Phase X*[ ] (2 in Eq. 8-5 and Fig. 8-9). The following equations convert the frequency domain from rectangular to polar notation, and vice versa:

Frequency
0 0.1 0.2 0.3 0.4 0.5
-2
-1
0
1
2
a. Re X[ ]
Frequency
0 0.1 0.2 0.3 0.4 0.5
-1
0
1
2
c. Mag X[ ]
Frequency
0 0.1 0.2 0.3 0.4 0.5
-6
-4
-2
0
2
4
6
d. Phase X[ ]
Polar Rectangular
FIGURE 8-10
Example of rectangular and polar frequency domains. This example shows a frequency domain expressed in both rectangular and polar notation. As in this case, polar notation usually provides human observers with a better understanding of the characteristics of the signal. In comparison, the rectangular form is almost always used when math computations are required. Pay special notice to the fact that the first and last

samples in the phase must be zero, just as they are in the imaginary part.
Frequency
0 0.1 0.2 0.3 0.4 0.5
-2
-1
0
1
2
b. Im X[ ]
Amplitude
Amplitude Amplitude
Phase (radians)
Rectangular and polar notation allow you to think of the DFT in two different ways. With rectangular notation, the DFT decomposes an N point signal into $N/2\%$ 1 cosine waves and sine waves, each with a specified *amplitude*. In $N/2\%$ 1 polar notation, the DFT decomposes an N point signal into cosine $N/2\%$ 1 waves, each with a specified *amplitude* (called the *magnitude*) and *phase shift*. Why does polar notation use cosine waves instead of sine waves? Sine waves cannot represent the DC component of a signal, since a sine wave of zero frequency is composed of all zeros (see Figs. 8-5 a&b).
Even though the polar and rectangular representations contain exactly the same information, there are many instances where one is easier to use that the other. For example, Fig. 8-10 shows a frequency domain signal in both rectangular and polar form. Warning: Don't try to understand the shape of the real and imaginary parts; your head will explode! In comparison, the polar curves are straightforward: only frequencies below about 0.25 are present, and the phase shift is approximately proportional to the frequency. This is the frequency response of a low-pass filter.

When should you use rectangular notation and when should you use polar? Rectangular notation is usually the best choice for calculations, such as in equations and computer programs. In comparison, graphs are almost always in polar form. As shown by the previous example, it is nearly impossible for *humans* to understand the characteristics of a frequency domain signal by looking at the real and imaginary parts. In a typical program, the frequency domain signals are kept in rectangular notation until an observer needs to look at them, at which time a rectangular-to-polar conversion is done.
Why is it easier to understand the frequency domain in polar notation? This question goes to the heart of why decomposing a signal into sinusoids is *useful*. Recall the property of *sinusoidal fidelity* from Chapter 5: if a sinusoid enters a linear system, the output will also be a sinusoid, and at exactly the same frequency as the input. Only the amplitude and phase can change. Polar notation directly represents signals in terms of the amplitude and phase of the component cosine waves. In turn, systems can be represented by how they modify the amplitude and phase of each of these cosine waves.
Now consider what happens if rectangular notation is used with this scenario. A mixture of cosine and sine waves enter the linear system, resulting in a mixture of cosine and sine waves leaving the system. The problem is, a cosine wave on the input may result in both cosine and sine waves on the output. Likewise, a sine wave on the input can result in both cosine and sine waves on the output. While these cross-terms can be straightened out, the overall method doesn't match with why we wanted to use sinusoids in the first place.

Polar Nuisances

There are many nuisances associated with using polar notation. None of these are overwhelming, just really annoying! Table 8-3 shows a computer program for converting between rectangular and polar notation, and provides solutions

for some of these pests.

Nuisance 1: Radians vs. Degrees

It is possible to express the phase in either *degrees* or *radians*. When expressed in degrees, the values in the phase signal are between -180 and 180. Using radians, each of the values will be between -B and B, that is, between -3.141592 to 3.141592. Most computer languages require the use radians for their trigonometric functions, such as cosine, sine, arctangent, etc. It can be irritating to work with these long decimal numbers, and difficult to interpret the data you receive. For example, if you want to introduce a 90 degree phase shift into a signal, you need to add 1.570796 to the phase. While it isn't going to kill you to type this into your program, it does become tiresome. The best way to handle this problem is to define the constant, *P*I = 3.141592, at the beginning of your program. A 90 degree phase shift can then be written as . Degrees and radians are both widely used in DSP and you need to *PI* /2 become comfortable with both.


```
100 'RECTANGULAR-TO-POLAR & POLAR-TO-RECTANGULAR CONVERSION
110 '
120 DIM REX[256] 'REX[ ] holds the real part
130 DIM IMX[256] 'IMX[ ] holds the imaginary part
140 DIM MAG[256] 'MAG[ ] holds the magnitude
150 DIM PHASE[256] 'PHASE[ ] holds the phase
160 '
170 PI = 3.14159265
180 '
190 GOSUB XXXX 'Mythical subroutine to load data into REX[ ] and IMX[ ]
200 '
210 '
220 ' 'Rectangular-to-polar conversion, Eq. 8-6
230 FOR K% = 0 TO 256
240 MAG[K%] = SQR( REX[K%]^2 + IMX[K%]^2 ) 'from Eq. 8-6
250 IF REX[K%] = 0 THEN REX[K%] = 1E-20 'prevent divide by 0 (nuisance 2)
260 PHASE[K%] = ATN( IMX[K%] / REX[K%] ) 'from Eq. 8-6
270 ' 'correct the arctan (nuisance 3)
280 IF REX[K%] < 0 AND IMX[K%] < 0 THEN PHASE[K%] = PHASE[K%] - PI
290 IF REX[K%] < 0 AND IMX[K%] >= 0 THEN PHASE[K%] = PHASE[K%] + PI
300 NEXT K%
310 '
320 '
330 ' 'Polar-to-rectangular conversion, Eq. 8-7
340 FOR K% = 0 TO 256
350 REX[K%] = MAG[K%] * COS( PHASE[K%] )
360 IMX[K%] = MAG[K%] * SIN( PHASE[K%] )
370 NEXT K%
380 '
390 END
```

TABLE 8-3

Nuisance 2: Divide by zero error

When converting from rectangular to polar notation, it is very common to find frequencies where the real part is zero and the imaginary part is some nonzero value. This simply means that the phase is exactly 90 or -90 degrees. Try to tell your computer this! When your program tries to calculate the phase from: , a *divide by Phase X*[*k*] ' arctan(*Im X*[*k*] / *Re X*[*k*]) *zero error* occurs. Even if the program execution doesn't halt, the phase you obtain for this frequency won't be correct. To avoid this problem, the real part must be tested for being zero before the division. If it is zero, the imaginary part must be tested for being positive or negative, to determine whether to set the phase to B/2 or -B/2, respectively. Lastly, the division

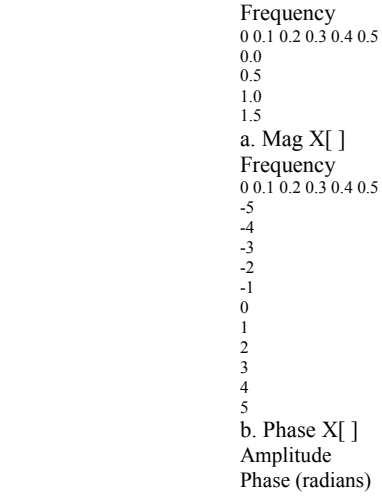
needs to be bypassed. Nothing difficult in all these steps, just the potential for aggravation. An alternative way to handle this problem is shown in line 250 of Table 8-3. If the real part is zero, change it to a negligibly small number to keep the math processor happy during the division.

Nuisance 3: Incorrect arctan

Consider a frequency domain sample where and . *ReX*[*k*] ' 1 *Im X*[*k*] ' 1 Equation 8-6 provides the corresponding polar values of and *Mag X*[*k*] ' 1.414 . Now consider another sample where and *Phase X*[*k*] ' 45E *ReX*[*k*] ' &1

FIGURE 8-11

The phase of small magnitude signals. At frequencies where the magnitude drops to a very low value, round-off noise can cause wild excursions of the phase. Don't make the mistake of thinking this is a meaningful signal.



. Again, Eq. 8-6 provides the values of and *Im X*[*k*] ' &1 *Mag X*[*k*] ' 1.414 . The problem is, the phase is wrong! It should be . *Phase X*[*k*] ' 45E &135E This error occurs whenever the real part is negative. This problem can be corrected by testing the real and imaginary parts after the phase has been calculated. If both the real and imaginary parts are negative, subtract 180E (or B radians) from the calculated phase. If the real part is negative and the imaginary part is positive, add (or B radians). Lines 340 and 350 of the 180E program in Table 8-3 show how this is done. If you fail to catch this problem, the calculated value of the phase will only run between -B/2 and B/2, rather than between -B and B. Drill this into your mind. If you see the phase only extending to ±1.5708, you have forgotten to correct the ambiguity in the arctangent calculation.

Nuisance 4: Phase of very small magnitudes

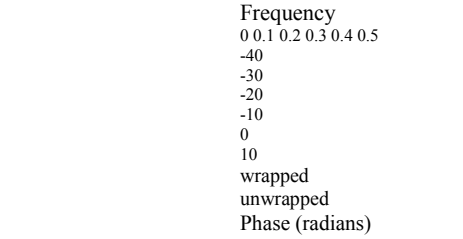
Imagine the following scenario. You are grinding away at some DSP task, and suddenly notice that part of the phase doesn't look right. It might be noisy, jumping all over, or just plain *wrong*. After spending the next hour looking through hundreds of lines of computer code, you find the answer. The corresponding values in the magnitude are so small that they are buried in round-off noise. If the magnitude is negligibly small, the phase doesn't have any meaning, and can assume unusual values. An example of this is shown in Fig. 8-11. It is usually obvious when an *amplitude* signal is lost in noise; the values are so small that you are forced to suspect that the values are meaningless. The phase is different. When a polar signal is contaminated with noise, the values in the phase are random numbers between -B and B. Unfortunately, this often *looks* like a real signal, rather than the nonsense it really is.

Nuisance 5: 2B ambiguity of the phase

Look again at Fig. 8-10d, and notice the several discontinuities in the data. Every time a point looks as if it is going to dip below -3.14592, it snaps back to 3.141592. This is a result of the periodic nature of sinusoids. For

FIGURE 8-12

Example of phase unwrapping. The top curve shows a typical phase signal obtained from a rectangular-to-polar conversion routine. Each value in the signal must be between -B and B (i.e., -3.14159 and 3.14159). As shown in the lower curve, the phase can be *unwrapped* by adding or subtracting integer multiplies of 2B from each sample, where the integer is chosen to minimize the discontinuities between points.




```
100 ' PHASE UNWRAPPING
110 '
120 DIM PHASE[256] 'PHASE[ ] holds the original phase
130 DIM UWPHASE[256] 'UWPHASE[ ] holds the unwrapped phase
140 '
150 PI = 3.14159265
160 '
170 GOSUB XXXX 'Mythical subroutine to load data into PHASE[ ]
180 '
190 UWPHASE[0] = 0 'The first point of all phase signals is zero
200 '
210 ' 'Go through the unwrapping algorithm
220 FOR K% = 1 TO 256
230 C% = CINT( (UWPHASE[K%-1] - PHASE[K%]) / (2 * PI) )
240 UWPHASE[K%] = PHASE[K%] + C%*2*PI
250 NEXT K%
260 '
270 END
```

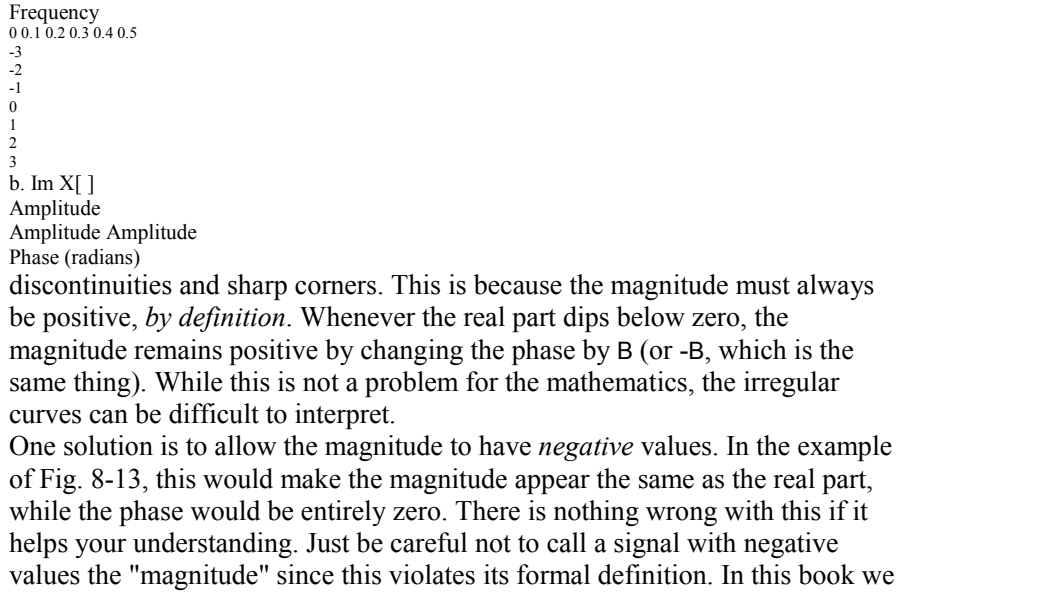
TABLE 8-4

example, a phase shift of is exactly the same as a phase shift of $q\ q\ p + 2$, $q\ p + 4$, etc. Any sinusoid is unchanged when you add an integer multiple of $q\ p + 6$, 2B to the phase. The apparent discontinuities in the signal are a result of the computer algorithm picking its favorite choice from an infinite number of equivalent possibilities. The smallest possible value is always chosen, keeping the phase between -B and B.

It is often easier to understand the phase if it does not have these discontinuities, even if it means that the phase extends above B, or below -B. This is called **unwrapping the phase**, and an example is shown in Fig. 8-12. As shown by the program in Table 8-4, a multiple of 2B is added or subtracted from each value of the phase. The exact value is determined by an algorithm that minimizes the difference between adjacent samples.

Nuisance 6: The magnitude is always positive (B ambiguity of the phase)

Figure 8-13 shows a frequency domain signal in rectangular and polar form. The real part is smooth and quite easy to understand, while the imaginary part is entirely zero. In comparison, the polar signals contain abrupt

FIGURE 8-13
Example signals in rectangular and polar form. Since the magnitude must always be positive (by definition), the magnitude and phase may contain abrupt discontinuities and sharp corners. Figure (d) also shows another nuisance: random noise can cause the phase to rapidly oscillate between B or -B.


discontinuities and sharp corners. This is because the magnitude must always be positive, *by definition*. Whenever the real part dips below zero, the magnitude remains positive by changing the phase by B (or -B, which is the same thing). While this is not a problem for the mathematics, the irregular curves can be difficult to interpret.

One solution is to allow the magnitude to have *negative* values. In the example of Fig. 8-13, this would make the magnitude appear the same as the real part, while the phase would be entirely zero. There is nothing wrong with this if it helps your understanding. Just be careful not to call a signal with negative values the "magnitude" since this violates its formal definition. In this book we use the weasel words: *unwrapped magnitude* to indicate a "magnitude" that is allowed to have negative values.

Nuisance 7: Spikes between B and -B

Since B and -B represent the same phase shift, round-off noise can cause adjacent points in the phase to rapidly switch between the two values. As shown in Fig. 8-13d, this can produce sharp breaks and spikes in an otherwise smooth curve. Don't be fooled, the phase isn't really this discontinuous.

9 Applications of the DFT

The Discrete Fourier Transform (DFT) is one of the most important tools in Digital Signal Processing. This chapter discusses three common ways it is used. First, the DFT can calculate a signal's *frequency spectrum*. This is a direct examination of information encoded in the frequency, phase, and amplitude of the component sinusoids. For example, human speech and hearing use signals with this type of encoding. Second, the DFT can find a system's frequency response from the system's impulse response, and vice versa. This allows systems to be analyzed in the *frequency domain*, just as convolution allows systems to be analyzed in the *time domain*. Third, the DFT can be used as an intermediate step in more elaborate signal processing techniques. The classic example of this is *FFT convolution*, an algorithm for convolving signals that is hundreds of times faster than conventional methods.

Spectral Analysis of Signals

It is very common for information to be encoded in the sinusoids that form a signal. This is true of naturally occurring signals, as well as those that have been created by humans. Many things oscillate in our universe. For example, speech is a result of vibration of the human vocal cords; stars and planets change their brightness as they rotate on their axes and revolve around each other; ship's propellers generate periodic displacement of the water, and so on. The *shape* of the time domain waveform is not important in these signals; the key information is in the *frequency*, *phase* and *amplitude* of the component sinusoids. The DFT is used to extract this information.

An example will show how this works. Suppose we want to investigate the sounds that travel through the ocean. To begin, a microphone is placed in the water and the resulting electronic signal amplified to a reasonable level, say a few volts. An analog low-pass filter is then used to remove all frequencies above 80 hertz, so that the signal can be digitized at 160 samples per second. After acquiring and storing several thousand samples, what next?

The first thing is to simply *look* at the data. Figure 9-1a shows 256 samples from our imaginary experiment. All that can be seen is a noisy waveform that conveys little information to the human eye. For reasons explained shortly, the next step is to multiply this signal by a smooth curve called a **Hamming window**, shown in (b). (Chapter 16 provides the equations for the Hamming and other windows; see Eqs. 16-1 and 16-2, and Fig. 16-2a). This results in a 256 point signal where the samples near the ends have been reduced in amplitude, as shown in (c).

Taking the DFT, and converting to polar notation, results in the 129 point frequency spectrum in (d). Unfortunately, this also looks like a noisy mess. This is because there is not enough information in the original 256 points to obtain a well behaved curve. Using a longer DFT does nothing to help this problem. For example, if a 2048 point DFT is used, the frequency spectrum becomes 1025 samples long. Even though the original 2048 points contain more information, the greater number of samples in the spectrum dilutes the information by the same factor. Longer DFTs provide better frequency resolution, but the same noise level.

The answer is to use more of the original signal in a way that doesn't increase the number of points in the frequency spectrum. This can be done by breaking the input signal into many 256 point *segments*. Each of these segments is multiplied by the Hamming window, run through a 256 point

DFT, and converted to polar notation. The resulting frequency spectra are then *averaged* to form a single 129 point frequency spectrum. Figure (e) shows an example of averaging 100 of the frequency spectra typified by (d). The improvement is obvious; the noise has been reduced to a level that allows interesting features of the signal to be observed. Only the *magnitude* of the frequency domain is averaged in this manner; the *phase* is usually discarded because it doesn't contain useful information. The random noise reduces in proportion to the *square-root* of the number of segments. While 100 segments is typical, some applications might average *millions* of segments to bring out weak features.

There is also a second method for reducing spectral noise. Start by taking a very long DFT, say 16,384 points. The resulting frequency spectrum is high resolution (8193 samples), but very noisy. A low-pass digital filter is then used to *smooth* the spectrum, reducing the noise at the expense of the resolution. For example, the simplest digital filter might average 64 adjacent samples in the original spectrum to produce each sample in the filtered spectrum. Going through the calculations, this provides about the same noise and resolution as the first method, where the 16,384 points would be broken into 64 segments of 256 points each.

Which method should you use? The first method is easier, because the digital filter isn't needed. The second method has the *potential* of better performance, because the digital filter can be tailored to optimize the tradeoff between noise and resolution. However, this improved performance is seldom worth the trouble. This is because both noise and resolution can be improved by using *more data* from the input signal. For example,

Sample number
0 32 64 96 128 160 192 224 256
-0.5
0.0
0.5
1.0
1.5
255
b. Hamming window
Sample number
0 32 64 96 128 160 192 224 256
-1.0
-0.5
0.0
0.5
1.0
255
c. Windowed signal
Frequency
0 0.1 0.2 0.3 0.4 0.5
0
1
2
3
4
5
6
7
8
9
10
d. Single spectrum
Frequency
0 0.1 0.2 0.3 0.4 0.5
0
1
2
3
4
5
6
7
8
9
10

e. Averaged spectrum FIGURE 9-1
An example of spectral analysis. Figure (a) shows 256 samples taken from a (simulated) undersea microphone at a rate of 160 samples per second. This signal is multiplied by the Hamming window shown in (b), resulting in the windowed signal in (c). The frequency spectrum of the windowed signal is found using the DFT, and is displayed in (d) (magnitude only). Averaging 100 of these spectra reduces the random noise, resulting in the averaged frequency spectrum shown in (e).
Sample number
0 32 64 96 128 160 192 224 256
-1.0
-0.5
0.0
0.5
1.0
255
a. Measured signal
DFT
Amplitude
Amplitude
Amplitude Amplitude
Amplitude
Time Domain Frequency Domain

imagine breaking the acquired data into 10,000 segments of 16,384 samples each. This resulting frequency spectrum is high resolution (8193 points) *and* low noise (10,000 averages). Problem solved! For this reason, we will only look at the averaged segment method in this discussion.

Figure 9-2 shows an example spectrum from our undersea microphone, illustrating the features that commonly appear in the frequency spectra of acquired signals. Ignore the sharp peaks for a moment. Between 10 and 70 hertz, the signal consists of a relatively flat region. This is called **white noise** because it contains an equal amount of all frequencies, the same as white light. It results from the noise on the time domain waveform being *uncorrelated* from sample-to-sample. That is, knowing the noise value present on any one sample provides no information on the noise value present on any other sample. For example, the random motion of electrons in electronic circuits produces white noise. As a more familiar example, the sound of the water spray hitting the shower floor is white noise. The white noise shown in Fig. 9-2 could be originating from any of several sources, including the analog electronics, or the ocean itself.

Above 70 hertz, the white noise rapidly decreases in amplitude. This is a result of the roll-off of the antialias filter. An ideal filter would pass all frequencies below 80 hertz, and block all frequencies above. In practice, a perfectly sharp cutoff isn't possible, and you should expect to see this gradual drop. If you don't, suspect that an aliasing problem is present.

Below about 10 hertz, the noise rapidly increases due to a curiosity called **1/f noise** (one-over-f noise). 1/f noise is a mystery. It has been measured in very diverse systems, such as traffic density on freeways and electronic noise in transistors. It probably could be measured in all systems, if you look low enough in frequency. In spite of its wide occurrence, a general theory and understanding of 1/f noise has eluded researchers. The cause of this noise can be identified in some specific systems; however, this doesn't answer the question of why 1/f noise is everywhere. For common analog electronics and most physical systems, the transition between white noise and 1/f noise occurs between about 1 and 100 hertz.

Now we come to the sharp peaks in Fig. 9-2. The easiest to explain is at 60 hertz, a result of electromagnetic interference from commercial electrical power. Also expect to see smaller peaks at multiples of this frequency (120, 180, 240 hertz, etc.) since the power line waveform is not a *perfect* sinusoid. It is also common to find interfering peaks between 25-40 kHz, a favorite for designers of switching power supplies. Nearby radio and television stations produce interfering peaks in the megahertz range. Low frequency peaks can be caused by components in the system vibrating when shaken. This is called *microphonics*, and typically creates peaks at 10 to 100 hertz.

Now we come to the actual signals. There is a strong peak at 13 hertz, with weaker peaks at 26 and 39 hertz. As discussed in the next chapter, this is the frequency spectrum of a nonsinusoidal periodic waveform. The peak at 13 hertz is called the fundamental frequency, while the peaks at 26 and 39

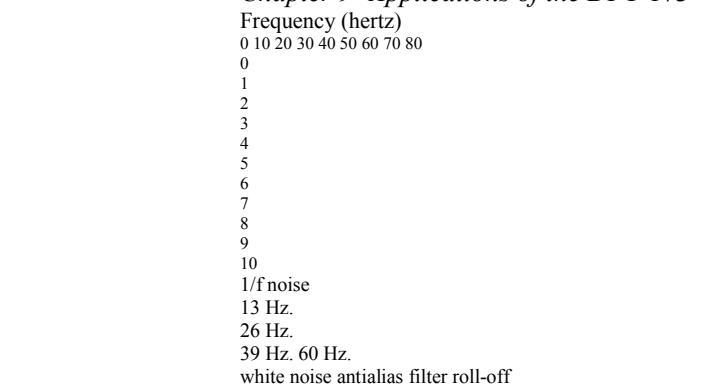


FIGURE 9-2
Example frequency spectrum. Three types of features appear in the spectra of acquired signals: (1) random noise, such as white noise and 1/f noise, (2) interfering signals from power lines, switching power supplies, radio and TV stations, microphonics, etc., and (3) real signals, usually appearing as a fundamental plus harmonics. This example spectrum (magnitude only) shows several of these features.

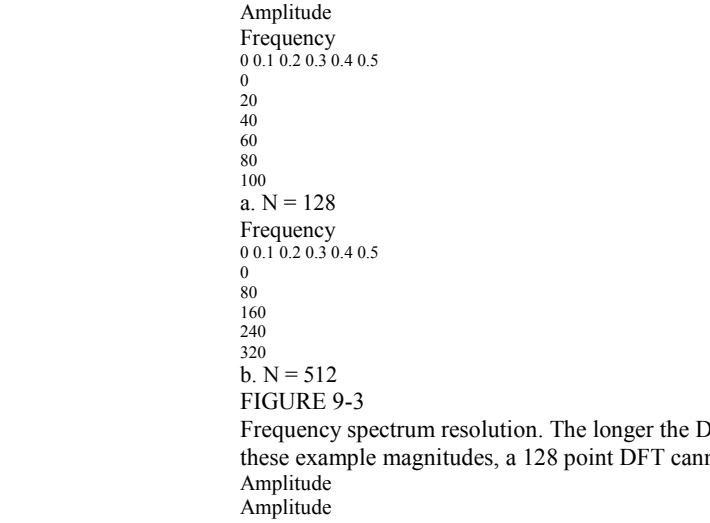


FIGURE 9-3
Frequency spectrum resolution. The longer the DFT, the better the ability to separate closely spaced features. In these example magnitudes, a 128 point DFT cannot resolve the two peaks, while a 512 point DFT can.

Amplitude
Amplitude

hertz are referred to as the second and third harmonic respectively. You would also expect to find peaks at other multiples of 13 hertz, such as 52, 65, 78 hertz, etc. You don't see these in Fig. 9-2 because they are buried

in the white noise. This 13 hertz signal might be generated, for example, by a submarines's three bladed propeller turning at 4.33 revolutions per second. This is the basis of *passive* sonar, identifying undersea sounds by their frequency and harmonic content.

Suppose there are peaks very close together, such as shown in Fig. 9-3. There are two factors that limit the frequency resolution that can be obtained, that is, how close the peaks can be without merging into a single entity. The first factor is the length of the DFT. The frequency spectrum produced by an N point DFT consists of samples equally spaced between zero and one- N/2%1 half of the sampling frequency. To separate two closely spaced frequencies, the sample spacing must be *smaller* than the distance between the two peaks. For example, a 512 point DFT is sufficient to separate the peaks in Fig. 9-3, while a 128 point DFT is not.

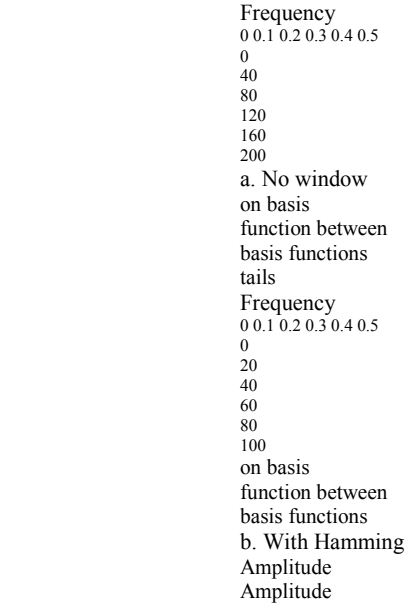


FIGURE 9-4

Example of using a window in spectral analysis. Figure (a) shows the frequency spectrum (magnitude only) of a signal consisting of two sine waves. One sine wave has a frequency exactly equal to a basis function, allowing it to be represented by a single sample. The other sine wave has a frequency *between* two of the basis functions, resulting in *tails* on the peak. Figure (b) shows the frequency spectrum of the same signal, but with a Hamming window applied before taking the DFT. The window makes the peaks look the same and reduces the tails, but broadens the peaks.

The second factor limiting resolution is more subtle. Imagine a signal created by adding two sine waves with only a slight difference in their frequencies. Over a short segment of this signal, say a few periods, the waveform will look like a *single* sine wave. The closer the frequencies, the longer the segment must be to conclude that more than one frequency is present. In other words, the *length* of the signal limits the frequency resolution. This is distinct from the first factor, because the *length of the input signal* does not have to be the same as the *length of the DFT*. For example, a 256 point signal could be padded with zeros to make it 2048 points long. Taking a 2048 point DFT produces a frequency spectrum with 1025 samples. The added zeros don't change the shape of the spectrum, they only provide more samples in the frequency domain. In spite of this very close sampling, the ability to separate closely spaced peaks would be only slightly better than using a 256 point DFT. When the DFT is the same length as the input signal, the resolution is limited about equally by these

two factors. We will come back to this issue shortly.
Next question: What happens if the input signal contains a sinusoid with a frequency *between* two of the basis functions? Figure 9-4a shows the answer. This is the frequency spectrum of a signal composed of two sine waves, one having a frequency *matching* a basis function, and the other with a frequency *between* two of the basis functions. As you should expect, the first sine wave is represented as a single point. The other peak is more difficult to understand. Since it cannot be represented by a single sample, it becomes a peak with **tails** that extend a significant distance away.
The solution? Multiply the signal by a Hamming window before taking the DFT, as was previously discussed. Figure (b) shows that the spectrum is changed in three ways by using the window. First, the two peaks are made to look more alike. This is good. Second, the tails are greatly reduced.

This is also good. Third, the window reduces the resolution in the spectrum by making the peaks wider. This is bad. In DSP jargon, windows provide a tradeoff between *resolution* (the width of the peak) and *spectral leakage* (the amplitude of the tails).
To explore the theoretical aspects of this in more detail, imagine an infinitely long discrete sine wave at a frequency of 0.1 the sampling rate. The frequency spectrum of this signal is an infinitesimally narrow peak, with all other frequencies being zero. Of course, neither this signal nor its frequency spectrum can be brought into a digital computer, because of their infinite and infinitesimal nature. To get around this, we change the signal in two ways, both of which distort the true frequency spectrum.
First, we *truncate* the information in the signal, by multiplying it by a window. For example, a 256 point *rectangular window* would allow 256 points to retain their correct value, while all the other samples in the infinitely long signal would be set to a value of zero. Likewise, the Hamming window would *shape* the retained samples, besides setting all points outside the window to zero. The signal is still infinitely long, but only a finite number of the samples have a nonzero value.
How does this windowing affect the frequency domain? As discussed in Chapter 10, when two time domain signals are *multiplied*, the corresponding frequency domains are *convolved*. Since the original spectrum is an infinitesimally narrow peak (i.e., a delta function), the spectrum of the windowed signal is the spectrum of the window shifted to the location of the peak. Figure 9-5 shows how the spectral peak would appear using four different window options (If you need a refresher on dB, look ahead to Chapter 14). Figure 9-5a results from a rectangular window. Figures (b) and (c) result from using two popular windows, the Hamming and the Blackman (as previously mentioned, see Eqs. 16-1 and 16-2, and Fig. 16-2a for information on these windows).
As shown in Fig. 9-5, all these windows have degraded the original spectrum by broadening the peak and adding tails composed of numerous side lobes. This is an unavoidable result of using only a portion of the original time domain signal. Here we can see the tradeoff between the three windows. The Blackman has the widest main lobe (bad), but the lowest amplitude tails (good). The rectangular window has the narrowest main lobe (good) but the largest tails (bad). The Hamming window sits between these two.
Notice in Fig. 9-5 that the frequency spectra are continuous curves, not discrete samples. After windowing, the time domain signal is still infinitely long, even though most of the samples are zero. This means that the frequency spectrum

consists of samples between 0 and 0.5, the same as a continuous line. 4/2%1 This brings in the second way we need to modify the time domain signal to allow it to be represented in a computer: *select N points from the signal*. These *N* points must contain all the nonzero points identified by the window, but may also include any number of the zeros. This has the effect

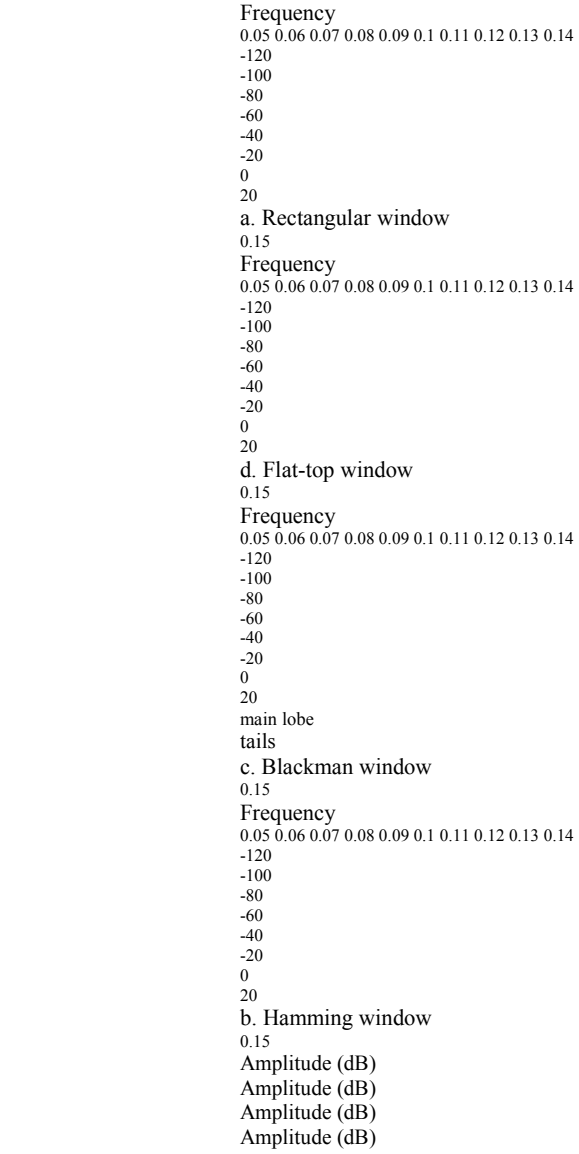


FIGURE 9-5

Detailed view of a spectral peak using various windows. Each peak in the frequency spectrum is a central lobe surrounded by tails formed from side lobes. By changing the window shape, the amplitude of the side lobes can be reduced at the expense of making the main lobe wider. The rectangular window, (a), has the narrowest main lobe but the largest amplitude side lobes. The Hamming window, (b), and the Blackman window, (c), have lower amplitude side lobes at the expense of a wider main lobe. The flat-top window, (d), is used when the amplitude of a peak must be accurately measured. These curves are for 255 point windows; longer windows produce proportionately narrower peaks.

of *sampling* the frequency spectrum's continuous curve. For example, if *N* is chosen to be 1024, the spectrum's continuous curve will be sampled 513 times between 0 and 0.5. If *N* is chosen to be much larger than the window length, the samples in the frequency domain will be close enough that the peaks and valleys of the continuous curve will be preserved in the new spectrum. If *N* is made the

same as the window length, the fewer number of samples in the spectrum results in the regular pattern of peaks and valleys turning into irregular tails, depending on where the samples happen to fall. This explains why the two peaks in Fig. 9-4a do not look alike. Each peak in Fig 9-4a is a *sampling* of the underlying curve in Fig. 9-5a. The presence or absence of the tails depends on where the samples are taken in relation to the peaks and valleys. If the sine wave exactly matches a basis function, the samples occur exactly at the valleys, eliminating the tails. If the sine wave is between two basis functions, the samples occur somewhere along the peaks and valleys, resulting in various patterns of tails.

This leads us to the **flat-top window**, shown in Fig. 9-5d. In some applications the *amplitude* of a spectral peak must be measured very accurately. Since the DFT's frequency spectrum is formed from samples, there is nothing to guarantee that a sample will occur exactly at the top of a peak. More than likely, the nearest sample will be slightly off-center, giving a value lower than the true amplitude. The solution is to use a window that produces a spectral peak with a *flat top*, insuring that one or more of the samples will always have the correct peak value. As shown in Fig. 9-5d, the penalty for this is a very broad main lobe, resulting in poor frequency resolution.

As it turns out, the shape we want for a flat-top window is exactly the same shape as the filter kernel of a low-pass filter. We will discuss the theoretical reasons for this in later chapters; for now, here is a cookbook description of how the technique is used. Chapter 16 discusses a low-pass filter called the *windowed-sinc*. Equation 16-4 describes how to generate the filter kernel (which we want to use as a window), and Fig. 16-4a illustrates the typical shape of the curve. To use this equation, you will need to know the value of two parameters: M and . These are found from the relations: , and f_c M ' N&2 , where N is the length of the DFT being used, and s is the number of f_c ' s /N samples you want on the flat portion of the peak (usually between 3 and 5). Table 16-1 shows a program for calculating the filter kernel (our window), including two subtle features: the normalization constant, K, and how to avoid a divide-by-zero error on the center sample. When using this method, remember that a DC value of *one* in the time domain will produce a peak of amplitude *one* in the frequency domain. However, a sinusoid of amplitude *one* in the time domain will only produce a spectral peak of amplitude *one-half*. (This is discussed in the last chapter: *Synthesis, Calculating the Inverse DFT*).

Frequency Response of Systems

Systems are analyzed in the *time domain* by using convolution. A similar analysis can be done in the *frequency domain*. Using the Fourier transform, every input signal can be represented as a group of cosine waves, each with a specified amplitude and phase shift. Likewise, the DFT can be used to represent every output signal in a similar form. This means that any linear system can be *completely* described by how it changes the amplitude and phase of cosine waves passing through it. This information is called the system's **frequency response.** Since both the impulse response and the frequency response contain complete information about the system, there must be a one-to-one correspondence between the two. Given one, you can calculate the other. The relationship between the impulse response and the frequency response is one of the foundations of signal processing: *A system's frequency response is the Fourier Transform of its impulse response*. Figure 9-6 illustrates these relationships.

Keeping with standard DSP notation, impulse responses use lower case

variables, while the corresponding frequency responses are upper case. Since $h[\ ]$ is the common symbol for the impulse response, is used for the frequency $H[\ ]$ response. Systems are described in the time domain by convolution, that is:

h[n] x[n] y[n]
H[f] X[f] Y[f]
TIME
DOMAIN
FREQUENCY
DOMAIN
FIGURE 9-6
Comparing system operation in the time and frequency domains. In the time domain, an input signal is *convolved* with an impulse response, resulting in the output signal, that is, . In the frequency $x[n]$ t $h[n]$ ' $y[n]$ domain, an input spectrum is *multiplied* by a frequency response, resulting in the output spectrum, that is, . The DFT and the Inverse DFT relate the signals in the two domain. $X[f] \times H[f]$ ' $Y[f]$
DFT
IDFT
IDFT
DFT
IDFT
DFT
. In the frequency domain, the input spectrum is *multiplied* $x[n]$ t $h[n]$ ' $y[n]$ by the frequency response, resulting in the output spectrum. As an equation:
. That is, *convolution* in the time domain corresponds to $X[f\,] \times H[\,f\,]$ ' $Y[\,f\,]$ *multiplication* in the frequency domain.

Figure 9-7 shows an example of using the DFT to convert a system's impulse response into its frequency response. Figure (a) is the impulse response of the system. Looking at this curve isn't going to give you the slightest idea what the system does. Taking a 64 point DFT of this impulse response produces the frequency response of the system, shown in (b). Now the function of this system becomes obvious, it passes frequencies between 0.2 and 0.3, and rejects all others. It is a band-pass filter. The *phase* of the frequency response could also be examined; however, it is *more* difficult to interpret and *less* interesting. It will be discussed in upcoming chapters.

Figure (b) is very jagged due to the low number of samples defining the curve. This situation can be improved by **padding** the impulse response with zeros before taking the DFT. For example, adding zeros to make the impulse response 512 samples long, as shown in (c), results in the higher resolution frequency response shown in (d).

How much resolution can you obtain in the frequency response? The answer is: *infinitely* high, if you are willing to pad the impulse response with an *infinite* number of zeros. In other words, there is nothing limiting the frequency resolution except the length of the DFT. This leads to a very important concept. Even though the impulse response is a *discrete* signal, the corresponding frequency response is *continuous*. An N point DFT of the impulse response provides *samples* of this continuous curve. If you N/2% 1 make the DFT longer, the resolution improves, and you obtain a better idea of

Sample number
0 8 16 24 32 40 48 56 64
-0.4
-0.2
0
0.2
0.4
a. Impulse response
63
Frequency
0 0.1 0.2 0.3 0.4 0.5
0.0

0.5
1.0
1.5
2.0
a. Impulse response b. Frequency response
Sample number
0 64 128 192 256 320 384 448 512
-0.4
-0.2
0
0.2
0.4
511
a. Impulse response c. padded with zeros
Frequency
0 0.1 0.2 0.3 0.4 0.5
0.0
0.5
1.0
1.5
2.0
d. Frequency response (high resolution)
Amplitude Amplitude
Amplitude Amplitude
Frequency Domain Time Domain
FIGURE 9-7
Finding the frequency response from the impulse response. By using the DFT, a system's impulse response, (a), can be transformed into the system's frequency response, (b). By padding the impulse response with zeros (c), higher resolution can be obtained in the frequency response, (d). Only the magnitude of the frequency response is shown in this example; discussion of the phase is postponed until the next chapter.

what the continuous curve looks like. Remember what the frequency response represents: amplitude and phase changes experienced by cosine waves as they pass through the system. Since the input signal can contain *any* frequency between 0 and 0.5, the system's frequency response *must* be a continuous curve over this range.

This can be better understood by bringing in another member of the Fourier transform family, the **Discrete Time Fourier Transform (DTFT)**. Consider an *N* sample signal being run through an *N* point DFT, producing an sample frequency domain. Remember from the last chapter that *N*/2% 1 the DFT considers the time domain signal to be *infinitely long* and *periodic*. That is, the *N* points are repeated over and over from negative to positive infinity. Now consider what happens when we start to pad the time domain signal with an ever increasing number of zeros, to obtain a finer and finer sampling in the frequency domain. Adding zeros makes the period of the time domain *longer*, while simultaneously making the frequency domain samples *closer together*.

Now we will take this to the extreme, by adding an *infinite* number of zeros to the time domain signal. This produces a different situation in two respects. First, the time domain signal now has an infinitely long period. In other words, it has turned into an *aperiodic* signal. Second, the frequency domain has achieved an infinitesimally small spacing between samples. That is, it has become a *continuous signal*. This is the DTFT, the procedure that changes a discrete aperiodic signal in the time domain into a frequency domain that is a continuous curve. In mathematical terms, a system's frequency response is found by taking the DTFT of its impulse response. Since this cannot be done in a computer, the DFT is used to calculate a *sampling* of the true frequency response. This is the difference between what you do in a computer (the DFT) and what you do with mathematical equations (the DTFT).

Convolution via the Frequency Domain

Suppose that you despise convolution. What are you going to do if given an input signal and impulse response, and need to find the resulting output signal?

Figure 9-8 provides an answer: transform the two signals into the frequency domain, multiply them, and then transform the result back into the time domain. This replaces one convolution with two DFTs, a multiplication, and an Inverse DFT. Even though the intermediate steps are very different, the output is *identical* to the standard convolution algorithm.

Does anyone hate convolution enough to go to this trouble? The answer is yes. Convolution is avoided for two reasons. First, convolution is *mathematically* difficult to deal with. For instance, suppose you are given a system's impulse response, and its output signal. How do you calculate what the input signal is? This is called **deconvolution**, and is virtually impossible to understand in the time domain. However, deconvolution can be carried out in the frequency domain as a simple *division*, the inverse operation of multiplication. The frequency domain becomes attractive whenever the complexity of the Fourier Transform is less than the complexity of the convolution. This isn't a matter of which you like better; it is a matter of which you hate less.

The second reason for avoiding convolution is *computation speed*. For example, suppose you design a digital filter with a kernel (impulse response) containing 512 samples. Using a 200 MHz personal computer with floating point numbers, each sample in the output signal requires about one millisecond to calculate, using the standard convolution algorithm. In other words, the throughput of the system is only about 1,000 samples per second. This is 40 times too slow for high-fidelity audio, and 10,000 times too slow for television quality video!

The standard convolution algorithm is slow because of the large number of multiplications and additions that must be calculated. Unfortunately, simply bringing the problem into the frequency domain via the DFT doesn't help at all. Just as many calculations are required to calculate the DFTs, as are required to directly calculate the convolution. A breakthrough was made in the problem in the early 1960s when the *Fast Fourier Transform* (FFT) was developed.

Sample number
0 128 256 384 512
-1
0
1
2
511
d. h[n]
Sample number
0 128 256 384 512
-2
-1
0
1
2
511
a. x[n]
Frequency
0 64 128 192 256
-60
-40
-20
0
20
40
60
b. Re X[f]
Frequency
0 64 128 192 256
-60
-40
-20
0
20
40
60
c. Im X[f]
Sample number

0 128 256 384 512
-30
-20
-10
0
10
20
30
511
g. y[n]
Frequency
0 64 128 192 256
-2000
-1000
0
1000
2000
h. Re Y[f]
Frequency
0 64 128 192 256
-2000
-1000
0
1000
2000
i. Im Y[f]
Frequency
0 64 128 192 256
-60
-40
-20
0
20
40
60
e. Re H[f]
Frequency
0 64 128 192 256
-60
-40
-20
0
20
40
60
f. Im H[f]

=

TIME

FREQUENCY

FIGURE 9-8

Frequency domain convolution. In the *time domain*, is convolved with resulting in , as is shown in Figs. $x[n]$ $h[n]$ $y[n]$ (a), (d), and (g). This same procedure to be accomplished in the *frequency domain*. The DFT is used to find the frequency spectrum of the input signal, (b) & (c), and the system's frequency response, (e) & (f). Multiplying these two frequency domain signals results in the frequency spectrum of the output signal, (h) & (i). The Inverse DFT is then used
to find the output signal, (g).

t =

x

IDFT
DFT
DFT
Amplitude
Amplitude
Amplitude
Amplitude
Amplitude Amplitude
Amplitude
Amplitude
Amplitude

The FFT is a clever algorithm for rapidly calculating the DFT. Using the FFT, convolution by multiplication in the frequency domain can be hundreds of times faster than conventional convolution. Problems that take hours of calculation

time are reduced to only minutes. This is why people get excited about the FFT, and processing signals in the frequency domain. The FFT will be presented in

EQUATION 9-1

Multiplication of frequency domain signals in rectangular form: . $Y[f] = X[f] \times H[f]$

$$ReY[f] = ReX[f]\,ReH[f] - ImX[f]\,ImH[f]$$
$$ImY[f] = ImX[f]\,ReH[f] + ReX[f]\,ImH[f]$$

EQUATION 9-2

Division of frequency domain signals in rectangular form, where: . $H[f] = Y[f] / X[f]$

$$ReH[f] = \frac{ReY[f]\,ReX[f] + ImY[f]\,ImX[f]}{ReX[f]^2 + ImX[f]^2}$$
$$ImH[f] = \frac{ImY[f]\,ReX[f] - ReY[f]\,ImX[f]}{ReX[f]^2 + ImX[f]^2}$$

Chapter 12, and the method of FFT convolution in Chapter 18. For now, focus on how signals are convolved by frequency domain multiplication.

To start, we need to define how to multiply one frequency domain signal by another, i.e., what it means to write: . In polar form, the $X[f] \times H[f] = Y[f]$ magnitudes are multiplied: , and the phases $MagY[f] = MagX[f] \times MagH[f]$ are added: . To understand this, imagine $PhaseY[f] = PhaseX[f] + PhaseH[f]$ a cosine wave entering a system with some amplitude and phase. Likewise, the output signal is also a cosine wave with some amplitude and phase. The polar form of the frequency response directly describes how the two amplitudes are related and how the two phases are related.

When frequency domain multiplication is carried out in *rectangular form* there are cross terms between the real and imaginary parts. For example, a sine wave entering the system can produce both cosine and sine waves in the output. To multiply frequency domain signals in rectangular notation:

Focus on understanding multiplication using *polar notation*, and the idea of cosine waves passing through the system. Then simply accept that these more elaborate equations result when the same operations are carried out in rectangular form. For instance, let's look at the *division* of one frequency domain signal by another. In polar form, the division of frequency domain signals is achieved by the inverse operations we used for multiplication. To calculate: , divide the magnitudes and subtract the phases, $H[f] = Y[f] / X[f]$ i.e., , . $MagH[f] = MagY[f] / MagX[f]$ $PhaseH[f] = PhaseY[f] - PhaseX[f]$ In rectangular form this becomes:

Now back to frequency domain convolution. You may have noticed that we cheated slightly in Fig. 9-8. Remember, the convolution of an N point signal with an M point impulse response results in an point output signal. $N+M-1$ We cheated by making the last part of the input signal all *zeros* to allow this expansion to occur. Specifically, (a) contains 453 nonzero samples, and (b) contains 60 nonzero samples. This means the convolution of the two, shown in (c), can fit comfortably in the 512 points provided.

Sample number
0 64 128 192 256
-2
-1
0
1
2
255
a. Input signal

Sample number
0 64 128 192 256
-2
-1
0
1
2
255
b. Impulse response
Sample number
0 64 128 192 256
-20
-15
-10
-5
0
5
10
15
20
c. Convolution of (a) and (b)
305 255
Sample number
0 256 512 768
-20
-15
-10
-5
0
5
10
15
20
767
d. Overlap of adjecent periods
Sample number
0 64 128 192 256
-20
-15
-10
-5
0
5
10
15
20
overlap
255
e. Circular convolution
Amplitude
FIGURE 9-9
Circular convolution. A 256 sample signal, (a), convolved with a 51 sample impulse response, (b), results in a 306 sample signal, (c). If this convolution is performed in the frequency domain using 256 point DFTs, the 306 points in the correct convolution cannot fit into the 256 samples provided. As shown in (d), samples 256 through 305 of the output signal are pushed into the next period to the right, where they *add* to the beginning of the next period's signal. Figure (e) is a single period of the resulting signal.

t =

Amplitude
Amplitude
Amplitude
Amplitude

Now consider the more general case in Fig. 9-9. The input signal, (a), is 256 points long, while the impulse response, (b), contains 51 nonzero points. This makes the convolution of the two signals 306 samples long, as shown in (c). The problem is, if we use frequency domain multiplication to perform the convolution, there are only 256 samples *allowed* in the output signal. In other words, 256 point DFTs are used to move (a) and (b) into the frequency

domain. After the multiplication, a 256 point Inverse DFT is used to find the output signal. How do you squeeze 306 values of the correct signal into the 256 points provided by the frequency domain algorithm? The answer is, you can't! The 256 points end up being a distorted version of the correct signal. This process is called **circular convolution**. It is important because you want to *avoid* it.

To understand circular convolution, remember that an *N* point DFT views the

time domain as being an infinitely long periodic signal, with N samples per period. Figure (d) shows three periods of how the DFT views the output signal in this example. Since , each period consists of 256 points: 0-255, N ' 256 256-511, and 512-767. Frequency domain convolution tries to place the 306 point *correct output signal*, shown in (c), into each of these 256 point periods. This results in 49 of the samples being pushed into the neighboring period to the right, where they overlap with the samples that are legitimately there. These overlapping sections add, resulting in each of the periods appearing as shown in (e), the *circular convolution*.

Once the nature of circular convolution is understood, it is quite easy to avoid. Simply pad each of the signals being convolved with enough zerosto allow the output signal room to handle the points in the correct convolution. $N\%M\&1$ For example, the signals in (a) and (b) could be padded with zeros to make them 512 points long, allowing the use of 512 point DFTs. After the frequency domain convolution, the output signal would consist of 306 nonzero samples, plus 206 samples with a value of zero. Chapter 18 explains this procedure in detail.

Why is it called *circular* convolution? Look back at Fig. 9-9d and examine the center period, samples 256 to 511. Since all of the periods are the same, the portion of the signal that flows out of this period to the *right*, is the same that flows into this period from the *left*. If you only consider a single period, such as in (e), it *appears* that the right side of the signal is somehow *connected* to the left side. Imagine a snake biting its own tail; sample 255 is located next to sample 0, just as sample 100 is located next to sample 101. When a portion of the signal exits to the right, it magically reappears on the left. In other words, the N point time domain behaves as if it were *circular*.

In the last chapter we posed the question: does it really matter if the DFT's time domain is viewed as being N points, rather than an infinitely long periodic signal of period N? Circular convolution is an example where it *does* matter. If the time domain signal is understood to be *periodic*, the distortion encountered in circular convolution can be simply explained as the signal expanding from one period to the next. In comparison, a rather bizarre conclusion is reached if only N points of the time domain are considered. That is, frequency domain convolution acts as if the time domain is somehow wrapping into a circular ring with sample 0 being positioned next to sample N-1.

CHAPTER

10 Fourier Transform Properties

The time and frequency domains are alternative ways of representing signals. The Fourier transform is the mathematical relationship between these two representations. If a signal is modified in one domain, it will also be changed in the other domain, although usually not in the same way. For example, it was shown in the last chapter that *convolving* time domain signals results in their frequency spectra being *multiplied.* Other mathematical operations, such as addition, scaling and shifting, also have a matching operation in the opposite domain. These relationships are called *properties* of the Fourier Transform, how a mathematical change in one domain results in a mathematical change in the other domain.

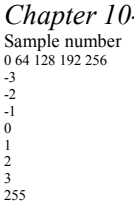
Linearity of the Fourier Transform

The Fourier Transform is *linear,* that is, it possesses the properties of *homogeneity* and *additivity*. This is true for all four members of the Fourier

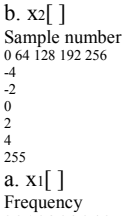
transform family (Fourier transform, Fourier Series, DFT, and DTFT).
Figure 10-1 provides an example of how homogeneity is a property of the Fourier transform. Figure (a) shows an arbitrary time domain signal, with the corresponding frequency spectrum shown in (b). We will call these two signals: and , respectively. *Homogeneity* means that a change in $x[\ ]$ $X[\ ]$ amplitude in one domain produces an identical change in amplitude in the other domain. This should make intuitive sense: when the amplitude of a time domain waveform is changed, the amplitude of the sine and cosine waves making up that waveform must also change by an equal amount.
In mathematical form, if and are a Fourier Transform pair, then $x[\ ]$ $X[\ ]$ $k\ x[\ ]$ and are also a Fourier Transform pair, for any constant k. If the $kX[\ ]$ frequency domain is represented in *rectangular* notation, means that both $kX[\ ]$ the real part and the imaginary part are multiplied by k. If the frequency domain is represented in *polar* notation, means that the magnitude is $kX[\ ]$ multiplied by k, while the phase remains unchanged.

Sample number
0 64 128 192 256
-3
-2
-1
0
1
2
3
255
c. k x[ ]
Sample number
0 64 128 192 256
-3
-2
-1
0
1
2
3
255
a. x[ ]
Frequency
0 0.1 0.2 0.3 0.4 0.5
0
10
20
30
40
50
b. X[ ]
Frequency
0 0.1 0.2 0.3 0.4 0.5
0
10
20
30
40
50
d. k X[ ]
Amplitude Amplitude
Amplitude Amplitude
Frequency Domain Time Domain
FIGURE 10-1
Homogeneity of the Fourier transform. If the amplitude is changed in one domain, it is changed by the same amount in the other domain. In other words, *scaling* in one domain corresponds to *scaling* in the other domain.
F.T.
F.T.
Additivity of the Fourier transform means that *addition* in one domain corresponds to *addition* in the other domain. An example of this is shown in Fig. 10-2. In this illustration, (a) and (b) are signals in the time domain called and , respectively. Adding these signals produces a third $x_1[\ ]$ $x_2[\ ]$ time domain signal called , shown in (c). Each of these three signals $x_3[\ ]$

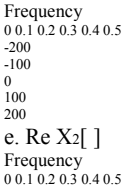
has a frequency spectrum consisting of a real and an imaginary part, shown in (d) through (i). Since the two time domain signals *add* to produce the third time domain signal, the two corresponding spectra *add* to produce the third spectrum. Frequency spectra are added in rectangular notation by adding the real parts to the real parts and the imaginary parts to the imaginary parts. If: , then: $x_1[n]$ % $x_2[n]$ ' $x_3[n]$ $ReX_1[f]$ % $ReX_2[f]$ ' $ReX_3[f]$ and . Think of this in terms of cosine and sine $ImX_1[f]$ % $ImX_2[f]$ ' $ImX_3[f]$ waves. All the cosine waves add (the real parts) and all the sine waves add (the imaginary parts) with no interaction between the two.
Frequency spectra in polar form cannot be directly added; they must be converted into rectangular notation, added, and then reconverted back to



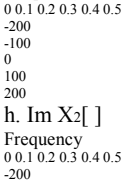
b. $x_2[\ ]$



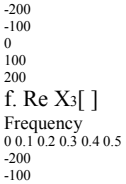
a. $x_1[\ ]$



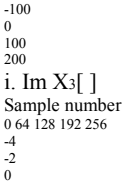
e. Re $X_2[\ ]$



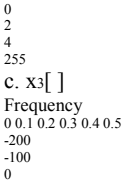
h. Im $X_2[\ ]$



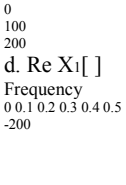
f. Re $X_3[\ ]$



i. Im $X_3[\ ]$



c. $x_3[\ ]$



d. Re $X_1[\ ]$

-100
0
100
200
g. Im X1[ ]
Amplitude
Amplitude
Amplitude
Amplitude
Amplitude
Amplitude
Amplitude
Amplitude
Amplitude
FIGURE 10-2
Additivity of the Fourier transform. Adding two or more signals in one domain results in the corresponding signals being added in the other domain. In this illustration, the time domain signals in (a) and (b) are added to produce the signal in (c). This results in the corresponding real and imaginary parts of the frequency spectra being added.
Frequency Domain Time Domain
F.T.
F.T.
F.T.
+ + +
= = =
polar form. This can also be understood in terms of how sinusoids behave. Imagine adding two sinusoids having the same frequency, but with different amplitudes (and) and phases (and). If the two phases happen to A_1 A_2 θ_1 θ_2 be same ( ), the amplitudes will add ( ) when the sinusoids are θ_1 ' θ_2 A_1 % A_2 added. However, if the two phases happen to be exactly opposite ( ), θ_1 ' & θ_2 the amplitudes will *subtract* ( ) when the sinusoids are added. The point A_1 & A_2 is, when sinusoids (or spectra) are in polar form, they *cannot* be added by simply adding the magnitudes and phases.

In spite of being linear, the Fourier transform is *not* shift invariant. In other words, a shift in the time domain *does not* correspond to a shift in the frequency domain. This is the topic of the next section.

Characteristics of the Phase

In mathematical form: if & , then a shift in the $x[n]$: $Mag\ X[f]$ $Phase\ X[f]$ time domain results in: & , (where $x[n$ % $s]$: $Mag\ X[f]$ $Phase\ X[f]$ % $2\pi s f$ is expressed as a fraction of the sampling rate, running between 0 and 0.5). In words, a shift of s samples in the time domain leaves the magnitude unchanged, but adds a linear term to the phase, . Let's look at an example of how $2\pi s f$ this works.
Figure 10-3 shows how the phase is affected when the time domain waveform is shifted to the left or right. The magnitude has not been included in this illustration because it isn't interesting; it is not changed by the time domain shift. In Figs. (a) through (d), the waveform is gradually shifted from having the peak centered on sample 128, to having it centered on sample 0. This sequence of graphs takes into account that the DFT views the time domain as *circular*; when portions of the waveform exit to the right, they reappear on the left.
The time domain waveform in Fig. 10-3 is symmetrical around a vertical axis, that is, the left and right sides are mirror images of each other. As

mentioned in Chapter 7, signals with this type of symmetry are called *linear phase*, because the phase of their frequency spectrum is a *straight line*. Likewise, signals that don't have this left-right symmetry are called *nonlinear phase*, and have phases that are something other than a straight line. Figures (e) through (h) show the phase of the signals in (a) through (d). As described in Chapter 7, these phase signals are *unwrapped*, allowing them to appear without the discontinuities associated with keeping the value between B and -B.

When the time domain waveform is shifted to the right, the phase remains a straight line, but experiences a *decrease* in slope. When the time domain is shifted to the left, there is an *increase* in the slope. This is the main property you need to remember from this section; a shift in the time domain corresponds to changing the slope of the phase.

Figures (b) and (f) display a unique case where the phase is entirely zero. This occurs when the time domain signal is *symmetrical* around sample *zero*. At first glance, this symmetry may not be obvious in (b); it may appear that the signal is symmetrical around sample 256 (i.e., *N*/2) instead. Remember that the DFT views the time domain as circular, with sample zero inherently connected to sample *N*-1. Any signal that is symmetrical around sample zero will also be symmetrical around sample *N*/2, and vice versa. When using members of the Fourier Transform family that do not view the time domain as periodic (such as the DTFT), the symmetry must be around sample zero to produces a zero phase.

Sample number
0 64 128 192 256 320 384 448 512
-1
0
1
2
511
d.
Frequency
0 0.1 0.2 0.3 0.4 0.5
-900
-600
-300
0
300
600
900
e.
Frequency
0 0.1 0.2 0.3 0.4 0.5
-900
-600
-300
0
300
600
900
f.
Phase (radians) Phase (radians) Phase (radians)
FIGURE 10-3
Phase changes resulting from a time domain shift.
Time Domain Frequency Domain
Amplitude Amplitude Amplitude Amplitude
Phase (radians)

Sample number
0 8 16 24 32
-2
-1
0
1
2
a. A low frequency
1 sample shift
= 1/32 cycle
Sample number
0 8 16 24 32
-2
-1
0
1
2
b. 1/2 of sampling frequency
1 sample shift
= 1/2 cycle
FIGURE 10-4
The relationship between samples and phase. Figures (a) and (b) show low and high frequency sinusoids, respectively. In (a), a one sample shift is equal to 1/32 of a cycle. In (b), a one sample shift is equal to 1/2 of a cycle. This is why a shift in the waveform changes the phase more at high frequencies than at low frequencies.
Amplitude
Amplitude
Figures (d) and (h) shows something of a riddle. First imagine that (d) was formed by shifting the waveform in (c) slightly more to the right. This means that the phase in (h) would have a slightly more negative slope than in (g). This phase is shown as line 1. Next, imagine that (d) was formed by starting with (a) and shifting it to the left. In this case, the phase should have a slightly more positive slope than (e), as is illustrated by line 2. Lastly, notice that (d) is symmetrical around sample $N/2$, and should therefore have a zero phase, as illustrated by line 3. Which of these three phases is correct? They all are, depending on how the B and $2B$ phase ambiguities (discussed in Chapter 8) are arranged. For instance, every sample in line 2 differs from the corresponding sample in line 1 by an integer multiple of $2B$, making them equal. To relate line 3 to lines 1 and 2, the B ambiguities must also be taken

into account.

To understand why the phase behaves as it does, imagine shifting a waveform by *one* sample to the right. This means that all of the sinusoids that compose the waveform must also be shifted by *one* sample to the right. Figure 10-4 shows two sinusoids that might be a part of the waveform. In (a), the sine wave has a very low frequency, and a one sample shift is only a small fraction of a full cycle. In (b), the sinusoid has a frequency of one-half of the sampling rate, the highest frequency that can exist in sampled data. A one sample shift at this frequency is equal to an entire 1/2 cycle, or B radians. That is, when a shift is expressed in terms of a phase change, it becomes *proportional* to the frequency of the sinusoid being shifted.

For example, consider a waveform that is symmetrical around sample zero, and therefore has a zero phase. Figure 10-5a shows how the phase of this signal changes when it is shifted left or right. At the highest frequency, one-half of the sampling rate, the phase increases by B for each one sample shift to the left, and decreases by B for each one sample shift to the right. At zero frequency there is no phase shift, and all of the frequencies between follow in a straight line.

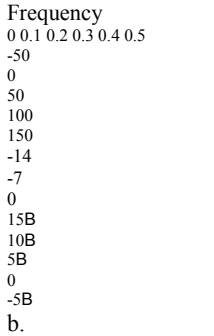
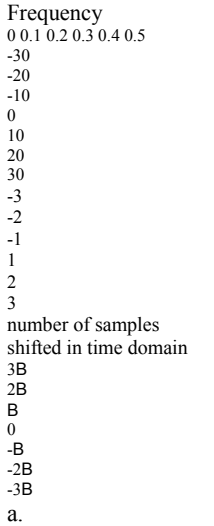


FIGURE 10-5

Phases resulting from time domain shifting. For each sample that a time domain signal is shifted in the positive direction (i.e., to the right), the phase at frequency 0.5 will decrease by B radians. For each sample shifted in the negative direction (i.e., to the left), the phase at frequency 0.5 will increase by B radians. Figure (a) shows this for a linear phase (a straight line), while (b) is an example using a nonlinear phase.



Phase (radians)

All of the examples we have used so far are *linear* phase. Figure 10-5b shows that *nonlinear* phase signals react to shifting in the same way. In this example the nonlinear phase is a straight line with two rectangular pulses. When the time domain is shifted, these nonlinear features are simply superimposed on the changing slope.

What happens in the *real* and *imaginary parts* when the time domain waveform is shifted? Recall that frequency domain signals in rectangular notation are nearly impossible for humans to understand. The real and imaginary parts typically look like random oscillations with no apparent pattern. When the time domain signal is shifted, the wiggly patterns of the real and imaginary parts become even more oscillatory and difficult to interpret. Don't waste your time trying to understand these signals, or how they are changed by time domain shifting.

Figure 10-6 is an interesting demonstration of what information is contained in the *phase*, and what information is contained in the *magnitude*. The waveform in (a) has two very distinct features: a rising edge at sample number 55, and a falling edge at sample number 110. Edges are very important when information is encoded in the *shape* of a waveform. An edge indicates *when* something happens, dividing whatever is on the left from whatever is on the right. It is time domain encoded information in its purest form. To begin the demonstration, the DFT is taken of the signal in (a), and the frequency spectrum converted into polar notation. To find the signal in (b), the phase is replaced with random numbers between -B and B, and the inverse DFT used to reconstruct the time domain waveform. In other words, (b) is based only on the information contained in the *magnitude*. In a similar manner, (c) is found by replacing the magnitude with small random numbers before using the inverse DFT. This makes the reconstruction of (c) based solely on the information contained in the *phase*.

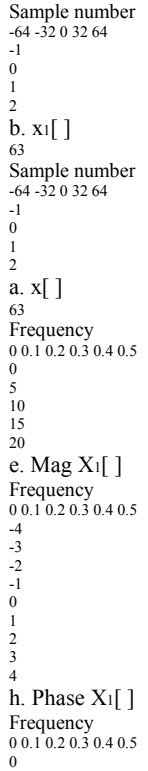
Sample number
0 64 128 192 256
-2
-1
0
1
2
3
255
a. Original signal
Sample number
0 64 128 192 256
-2
-1
0
1
2
3
255
b. Reconstructed from the magnitude
Sample number
0 64 128 192 256
-2
-1
0
1
2
3
255
c. Reconstructed from the phase
Amplitude Amplitude
Amplitude

FIGURE 10-6
Information contained in the phase. Figure (a) shows a pulse-like waveform. The signal in (b)

is created by taking the DFT of (a), replacing the *phase* with random numbers, and taking the Inverse DFT. The signal in (c) is found by taking the DFT of (a), replacing the *magnitude* with random numbers, and taking the Inverse DFT. The location of the *edges* is retained in (c), but not in (b). This shows that the phase contains information on the location of events in the time domain signal.

The result? The locations of the edges are clearly present in (c), but totally absent in (b). This is because an edge is formed when many sinusoids *rise* at the same location, possible only when their *phases* are coordinated. In short, much of the information about the shape of the time domain waveform is contained in the *phase*, rather than the *magnitude*. This can be contrasted with signals that have their information encoded in the frequency domain, such as audio signals. The magnitude is most important for these signals, with the phase playing only a minor role. In later chapters we will see that this type of understanding provides strategies for designing filters and other methods of processing signals. Understanding how information is represented in signals is always the first step in successful DSP.

Why does left-right symmetry correspond to a zero (or linear) phase? Figure 10-7 provides the answer. Such a signal can be decomposed into a left half and a right half, as shown in (a), (b) and (c). The sample at the center of symmetry (zero in this case) is divided equally between the left and right halves, allowing the two sides to be perfect mirror images of each other. The magnitudes of these two halves will be *identical*, as shown in (e) and (f), while the phases will be opposite in sign, as in (h) and (i). Two important concepts fall out of this. First, every signal that is symmetrical between the left and right will have a linear phase *because* the nonlinear phase of the left half exactly cancels the nonlinear phase of the right half.



5
10
15
20
f. Mag X2[ ]
Frequency
0 0.1 0.2 0.3 0.4 0.5
-4
-3
-2
-1
0
1
2
3
4
i. Phase X2[ ]
Frequency
0 0.1 0.2 0.3 0.4 0.5
0
5
10
15
20
d. Mag X[ ]
Frequency
0 0.1 0.2 0.3 0.4 0.5
-4
-3
-2
-1
0
1
2
3
4
g. Phase X[ ]
Sample number
-64 -32 0 32 64
-1
0
1
2
c. x2[ ]
63
Frequency Domain Time Domain
Decompose
Amplitude
Amplitude
Amplitude
Amplitude
Amplitude
Amplitude
Amplitude
Amplitude
Amplitude
FIGURE 10-7
Phase characteristics of left-right symmetry. A signal with left-right symmetry, shown in (a), can be decomposed into a right half, (b), and a left half, (c). The magnitudes of the two halves are identical, (e) and (f), while the phases are the negative of each other, (h) and (i).

Second, imagine flipping (b) such that it becomes (c). This left-right flip in the time domain does nothing to the magnitude, but changes the sign of every point in the phase. Likewise, changing the sign of the phase flips the time domain signal left-for-right. If the signals are continuous, the flip is around zero. If the signals are discrete, the flip is around sample zero *and* sample $N/2$, simultaneously.

Changing the sign of the phase is a common enough operation that it is given its own name and symbol. The name is **complex conjugation**, and it is

represented by placing a star to the upper-right of the variable. For example, if consists of and , then is called the $X[f]$ $Mag\ X[f]$ $Phase\ X[f]$ $X_t[f]$ *complex conjugate* and is composed of and . In $Mag\ X[f]$ &$Phase\ X[f]$ rectangular notation, the complex conjugate is found by leaving the real part alone, and changing the sign of the imaginary part. In mathematical terms, if $X[f]$

is composed of and , then is made up of and $ReX[f]$ $ImX[f]$ $X_t[f]$ $ReX[f]$. $-ImX[f]$
Here are several examples of how the complex conjugate is used in DSP. If has a Fourier transform of , then has a Fourier transform of $x[n]$ $X[f]$ $x[-n]$. In words, flipping the time domain left-for-right corresponds to $X_t[f]$ changing the sign of the phase. As another example, recall from Chapter 7 that correlation can be performed as a convolution. This is done by flipping one of the signals left-for-right. In mathematical form, is convolution, $a[n] * b[n]$ while is correlation. In the frequency domain these operations $a[n] * b[-n]$ correspond to and , respectively. As the last $A[f] \times B[f]$ $A[f] \times B_t[f]$ example, consider an arbitrary signal, , and its frequency spectrum, . $x[n]$ $X[f]$ The frequency spectrum can be changed to *zero phase* by multiplying it by its complex conjugate, that is, . In words, whatever phase $X[f] \times X_t[f]$ $X[f]$ happens to have will be canceled by adding its opposite (remember, when frequency spectra are multiplied, their phases are added). In the time domain, this means that (a signal convolved with a left-right flipped $x[n] * x[-n]$ version of itself) will have left-right symmetry around sample zero, regardless of what is. $x[n]$
To many engineers and mathematicians, this kind of manipulation *is* DSP. If you want to be able to communicate with this group, get used to using their language.

Periodic Nature of the DFT

Unlike the other three Fourier Transforms, the DFT views *both* the time domain and the frequency domain as *periodic*. This can be confusing and inconvenient since most of the signals used in DSP are *not* periodic. Nevertheless, if you want to use the DFT, you must conform with the DFT's view of the world.
Figure 10-8 shows two different interpretations of the time domain signal. First, look at the upper signal, the time domain viewed as *N* points. This represents how digital signals are typically acquired in scientific experiments and engineering applications. For instance, these 128 samples might have been acquired by sampling some parameter at regular intervals of *time*. Sample 0 is distinct and separate from sample 127 because they were acquired at *different* times. From the way this signal was formed, there is no reason to think that the samples on the left of the signal are even related to the samples on the right.
Unfortunately, the DFT doesn't see things this way. As shown in the lower figure, the DFT views these 128 points to be a single period of an infinitely long periodic signal. This means that the left side of the acquired signal is

FIGURE 10-8
Periodicity of the DFT's time domain signal. The time domain can be viewed as *N* samples in length, shown in the upper figure, or as an infinitely long periodic signal, shown in the lower figure.
The time domain
viewed as *N* points
The time domain

viewed as periodic
Amplitude
Amplitude
connected to the right side of a duplicate signal. Likewise, the right side of the acquired signal is connected to the left side of an identical period. This can also be thought of as the right side of the acquired signal wrapping around and connecting to its left side. In this view, sample 127 occurs next to sample 0, just as sample 43 occurs next to sample 44. This is referred to as being **circular**, and is identical to viewing the signal as being *periodic*.

The most serious consequence of time domain periodicity is **time domain aliasing.** To illustrate this, suppose we take a time domain signal and pass it through the DFT to find its frequency spectrum. We could immediately pass this frequency spectrum through an Inverse DFT to reconstruct the original time domain signal, but the entire procedure wouldn't be very interesting. Instead, we will modify the frequency spectrum in some manner before using the Inverse DFT. For instance, selected frequencies might be deleted, changed in amplitude or phase, shifted around, etc. These are the kinds of things routinely done in DSP. Unfortunately, these changes in the frequency domain can create a time domain signal that is too long to fit into

a single period. This forces the signal to spill over from one period into the adjacent periods. When the time domain is viewed as *circular*, portions of the signal that overflow on the right suddenly seem to reappear on the left side of the signal, and vice versa. That is, the overflowing portions of the signal *alias* themselves to a new location in the time domain. If this new location happens to already contain an existing signal, the whole mess adds, resulting in a loss of information. Circular convolution resulting from frequency domain multiplication (discussed in Chapter 9), is an excellent example of this type of aliasing.

Periodicity in the frequency domain behaves in much the same way, but is more complicated. Figure 10-9 shows an example. The upper figures show the magnitude and phase of the frequency spectrum, viewed as being composed of samples spread between 0 and 0.5 of the sampling rate. This is the $N/2 \% 1$ simplest way of viewing the frequency spectrum, but it doesn't explain many of the DFT's properties.

The lower two figures show how the DFT views this frequency spectrum as being periodic. The key feature is that the frequency spectrum between 0 and 0.5 appears to have a *mirror image* of frequencies that run between 0 and -0.5. This mirror image of **negative frequencies** is slightly different for the magnitude and the phase signals. In the magnitude, the signal is flipped leftfor-right. In the phase, the signal is flipped left-for-right, *and* changed in sign. As you recall, these two types of symmetry are given names: the magnitude is said to be an **even** signal (it has *even* symmetry), while the phase is said to be an **odd** signal (it has *odd* symmetry). If the frequency spectrum is converted into the real and imaginary parts, the *real part* will always be *even*, while the *imaginary part* will always be *odd*.

Taking these negative frequencies into account, the DFT views the frequency domain as periodic, with a period of 1.0 times the sampling rate, such as -0.5 to 0.5, or 0 to 1.0. In terms of sample numbers, this makes the length of the frequency domain period equal to N, the same as in the time domain.

The periodicity of the frequency domain makes it susceptible to **frequency domain aliasing**, completely analogous to the previously described time domain aliasing. Imagine a time domain signal that corresponds to some frequency spectrum. If the time domain signal is modified, it is obvious that

the frequency spectrum will also be changed. If the modified frequency spectrum cannot fit in the space provided, it will push into the adjacent periods. Just as before, this aliasing causes two problems: frequencies aren't where they should be, and overlapping frequencies from different periods add, destroying information.

Frequency domain aliasing is more difficult to understand than time domain aliasing, since the periodic pattern is more complicated in the frequency domain. Consider a single frequency that is being forced to move from 0.01 to 0.49 in the frequency domain. The corresponding negative frequency is therefore moving from -0.01 to -0.49. When the positive frequency moves

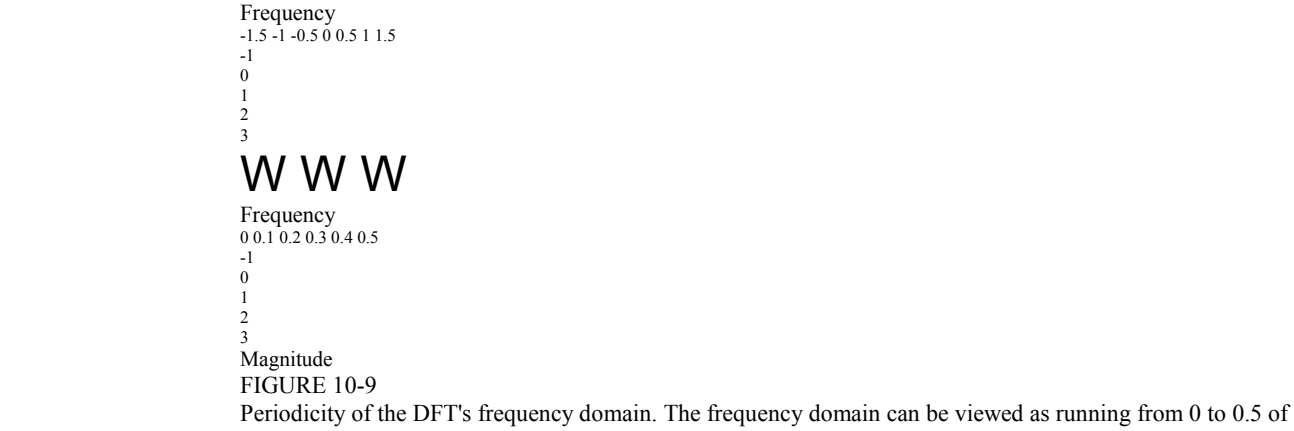


FIGURE 10-9

Periodicity of the DFT's frequency domain. The frequency domain can be viewed as running from 0 to 0.5 of the sampling rate (upper two figures), or an infinity long periodic signal with every other 0 to 0.5 segment flipped left-for-right (lower two figures).

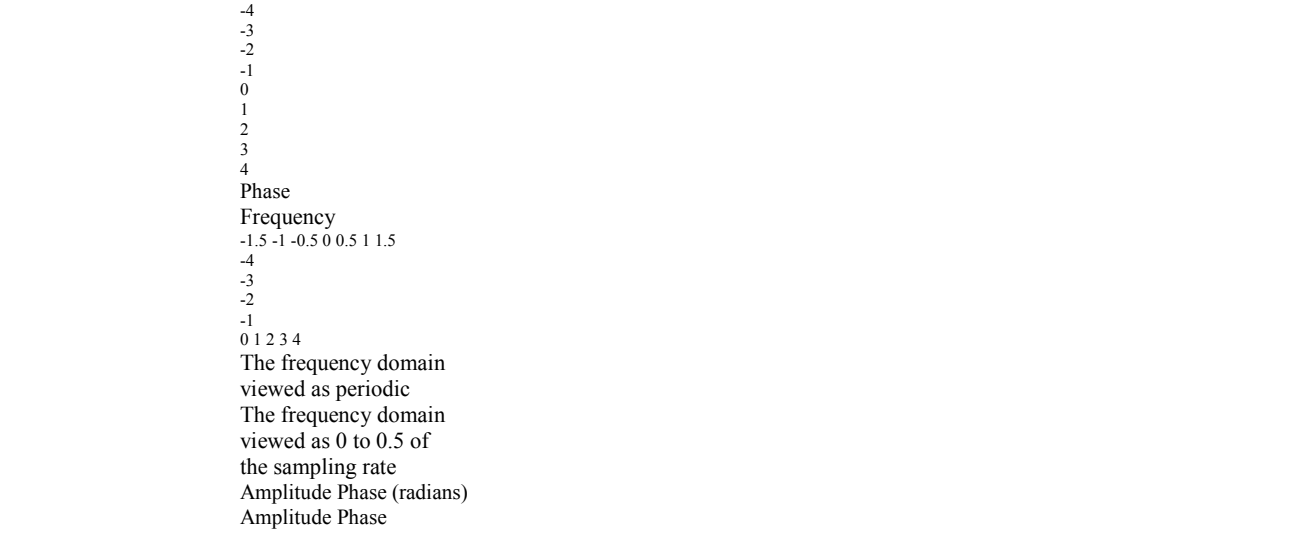


FIGURE 10-10

Examples of aliasing in the time and frequency domains, when only a single period is considered. In the time domain, shown in (a), portions of the signal that exits to the right, reappear on the left. In the frequency domain, (b), portions of the signal that exit to the right, reappear on the right as if they had been folded over.

a. Time domain aliasing b. Frequency domain aliasing

across the 0.5 barrier, the negative frequency is pushed across the -0.5 barrier. Since the frequency domain is periodic, these same events are occurring in the other periods, such as between 0.5 and 1.5. A clone of the positive frequency is crossing frequency 1.5 from left to right, while a clone of the negative frequency is crossing 0.5 from right to left. Now imagine what this looks like if you can only see the frequency band of 0 to 0.5. It appears that a frequency leaving to the *right*, reappears on the *right*, but moving in the opposite direction.

Figure 10-10 illustrates how aliasing appears in the time and frequency domains when only a single period is viewed. As shown in (a), if one end of a time domain signal is too long to fit inside a single period, the protruding end will be *cut off* and *pasted* onto the other side. In comparison, (b) shows that when a frequency domain signal overflows the period, the protruding end is *folded over*. Regardless of where the aliased segment ends up, it adds to whatever signal is already there, destroying information.

Let's take a closer look at these strange things called *negative frequencies*. Are they just some bizarre artifact of the mathematics, or do they have a real world meaning? Figure 10-11 shows what they are about. Figure (a) is a discrete signal composed of 32 samples. Imagine that you are given the task of finding the frequency spectrum that corresponds to these 32 points. To make your job easier, you are told that these points represent a discrete cosine wave. In other words, you must find the frequency and phase shift (and 2) such that matches the given $f\,x[n]$ ' cos(2Bnf /N % 2) samples. It isn't long before you come up with the solution shown in (b), that is, and . f' 3 2 ' &B/4

If you stopped your analysis at this point, you only get 1/3 credit for the problem. This is because there are two other solutions that you have missed. As shown in (c), the second solution is and . Even f' &3 2 ' B/4 if the idea of a *negative frequency* offends your sensibilities, it doesn't

Sample number
0 8 16 24 32
-2
-1
0
1
2
a. Samples
Sample number
0 8 16 24 32
-2
-1
0
1
2
b. Solution #1
f = 3, 2 = -B/4

FIGURE 10-11

The meaning of negative frequencies. The problem is to find the frequency spectrum of the discrete signal shown in (a). That is, we want to find the frequency and phase of the sinusoid that passed through all of the samples. Figure (b) is a solution using a *positive* frequency, while (c) is a solution using a *negative* frequency. Figure (d) represents a family of solutions to the problem.

Sample number
0 8 16 24 32
-2
-1
0
1

2
c. Solution #2
f = -3, 2 = B/4
Sample number
0 8 16 24 32
-2
-1
0
1
2
d. Solution #3
f = 35, 2 = -B/4
Amplitude Amplitude
Amplitude Amplitude
change the fact that it is a mathematically valid solution to the defined problem. Every *positive* frequency sinusoid can alternately be expressed as a *negative* frequency sinusoid. This applies to continuous as well as discrete signals

The third solution is not a single answer, but an infinite family of solutions. As shown in (d), the sinusoid with and passes through all of *f*' 35 2' &B/4 the discrete points, and is therefore a correct solution. The fact that it shows oscillation between the samples may be confusing, but it doesn't disqualify it from being an authentic answer. Likewise, , , , and *f*' ±29 *f*' ±35 *f*' ±61 are all solutions with multiple oscillations between the points. *f*' ±67

Each of these three solutions corresponds to a different section of the frequency spectrum. For discrete signals, the first solution corresponds to frequencies between 0 and 0.5 of the sampling rate. The second solution

results in frequencies between 0 and -0.5. Lastly, the third solution makes up the infinite number of duplicated frequencies below -0.5 and above 0.5. If the signal we are analyzing is continuous, the first solution results in frequencies from zero to positive infinity, while the second solution results in frequencies from zero to negative infinity. The third group of solutions does not exist for continuous signals.

Many DSP techniques do not require the use of negative frequencies, or an understanding of the DFT's periodicity. For example, two common ones were described in the last chapter, *spectral analysis*, and the *frequency response* of systems. For these applications, it is completely sufficient to view the time domain as extending from sample 0 to *N*-1, and the frequency domain from zero to one-half of the sampling frequency. These techniques can use a simpler view of the world because they never result in portions of one period moving into another period. In these cases, looking at a single period is just as good as looking at the entire periodic signal.

However, certain procedures can *only* be analyzed by considering how signals overflow between periods. Two examples of this have already been presented, *circular convolution* and *analog-to-digital conversion*. In circular convolution, multiplication of the frequency spectra results in the time domain signals being convolved. If the resulting time domain signal is too long to fit inside a single period, it overflows into the adjacent periods, resulting in *time domain aliasing*. In contrast, analog-to-digital conversion is an example of *frequency domain aliasing*. A nonlinear action is taken in the time domain, that is, changing a continuous signal into a discrete signal by sampling. The problem is, the spectrum of the original analog signal may be too long to fit inside the discrete signal's spectrum. When we force the situation, the ends of the spectrum protrude into adjacent periods. Let's look at two more examples where the periodic nature of the DFT is important, *compression & expansion* of signals, and *amplitude modulation*.

Compression and Expansion, Multirate methods

As shown in Fig. 10-12, a *compression* of the signal in one domain results in an *expansio*n in the other, and vice versa. For continuous signals, if is the $X(f)$ Fourier Transform of , then is the Fourier Transform of , $x(t)$ $1/k \times X(f/k)$ $x(kt)$ where k is the parameter controlling the expansion or contraction. If an event happens *faster* (it is compressed in time), it must be composed of *higher* frequencies. If an event happens *slower* (it is expanded in time), it must be composed of *lower* frequencies. This pattern holds if taken to either of the two extremes. That is, if the time domain signal is compressed so far that it becomes an *impulse*, the corresponding frequency spectrum is expanded so far that it becomes a *constant value*. Likewise, if the time domain is expanded until it becomes a constant value, the frequency domain becomes an impulse. Discrete signals behave in a similar fashion, but there are a few more details. The first issue with discrete signals is *aliasing*. Imagine that the

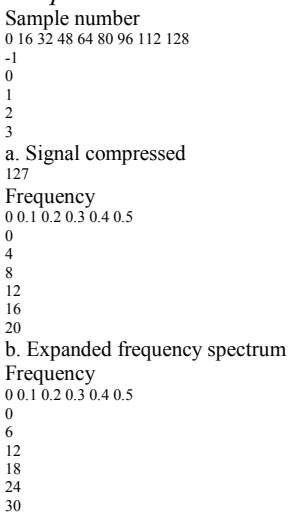


FIGURE 10-12

Compression and expansion. Compressing a signal in one domain results in the signal being expanded in the other domain, and vice versa. Figures (c) and (d) show a discrete signal and its spectrum, respectively. In (a) and (b), the time domain signal has been compressed, resulting in the frequency spectrum being expanded. Figures (e) and (f) show the opposite process. As shown in these figures, discrete signals are expanded or contracted by expanding or contracting the underlying continuous waveform. This underlying waveform is then resampled to find the new discrete signal.
Amplitude
Amplitude
Amplitude Amplitude
Amplitude Amplitude
pulse in (a) is compressed several times more than is shown. The frequency spectrum is expanded by an equal factor, and several of the humps in (b) are pushed to frequencies beyond 0.5. The resulting aliasing breaks the simple expansion/contraction relationship. This type of aliasing can also happen in the

time domain. Imagine that the frequency spectrum in (f) is compressed much harder, resulting in the time domain signal in (e) expanding into neighboring periods.

A second issue is to define exactly what it means to compress or expand a discrete signal. As shown in Fig. 10-12a, a discrete signal is compressed by compressing the underlying *continuous* curve that the samples lie on, and then resampling the new continuous curve to find the new discrete signal. Likewise, this same process for the expansion of discrete signals is shown in (e). When a discrete signal is compressed, events in the signal (such as the width of the pulse) happen over a *fewer* number of samples. Likewise, events in an expanded signal happen over a *greater* number of samples.

An equivalent way of looking at this procedure is to keep the underlying continuous waveform the same, but resample it at a different sampling rate. For instance, look at Fig. 10-13a, a discrete Gaussian waveform composed of 50 samples. In (b), the same underlying curve is represented by 400 samples. The change between (a) and (b) can be viewed in two ways: (1) the sampling rate has been kept constant, but the underlying waveform has been expanded to be eight times wider, or (2) the underlying waveform has been kept constant, but the sampling rate has increased by a factor of eight. Methods for changing the sampling rate in this way are called **multirate** techniques. If more samples are added, it is called **interpolation**. If fewer samples are used to represent the signal, it is called **decimation**. Chapter 3 describes how multirate techniques are used in ADC and DAC.

Here is the problem: if we are given an arbitrary discrete signal, how do we know what the underlying continuous curve is? It depends on if the signal's information is encoded in the *time domain* or in the *frequency domain*. For time domain encoded signals, we want the underlying continuous waveform to be a smooth curve that passes through all the samples. In the simplest case, we might draw straight lines between the points and then round the rough corners. The next level of sophistication is to use a curve fitting algorithm, such as a spline function or polynomial fit. There is not a single "correct" answer to this problem. This approach is based on minimizing irregularities in the *time domain* waveform, and completely ignores the frequency domain.

When a signal has information encoded in the frequency domain, we ignore the time domain waveform and concentrate on the frequency spectrum. As discussed in the last chapter, a finer sampling of a frequency spectrum (more samples between frequency 0 and 0.5) can be obtained by padding the time domain signal with zeros before taking the DFT. Duality allows this to work in the opposite direction. If we want a finer sampling in the time domain

(interpolation), pad the frequency spectrum with zeros before taking the Inverse DFT. Say we want to interpolate a 50 sample signal into a 400 sample signal. It's done like this: (1) Take the 50 samples and add zeros to make the signal 64 samples long. (2) Use a 64 point DFT to find the frequency spectrum, which will consist of a 33 point real part and a 33 point imaginary part. (3) Pad the right side of the frequency spectrum

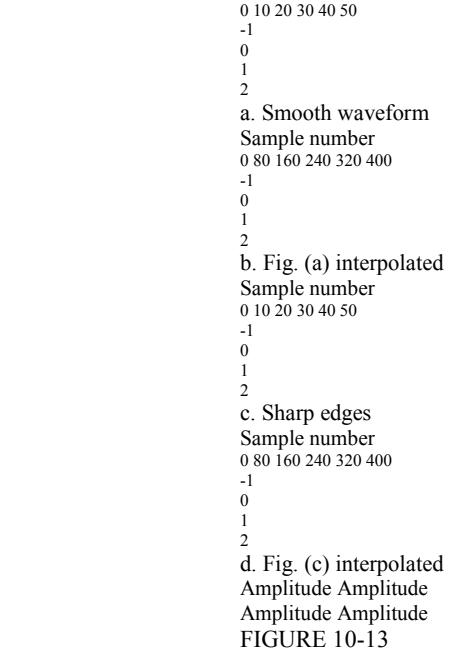


FIGURE 10-13
Interpolation by padding the frequency domain. Figures (a) and (c) each consist of 50 samples. These are interpolated to 400 samples by padding the frequency domain with zeros, resulting in (b) and (d), respectively. (Figures (b) and (d) are discrete signals, but are drawn as continuous lines because of the large number of samples).

(both the real and imaginary parts) with 224 zeros to make the frequency spectrum 257 points long. (4) Use a 512 point Inverse DFT to transform the data back into the time domain. This will result in a 512 sample signal that is a high resolution version of the 64 sample signal. The first 400 samples of this signal are an interpolated version of the original 50 samples.

The key feature of this technique is that the interpolated signal is composed of *exactly* the same frequencies as the original signal. This may or may not provide a well-behaved fit in the time domain. For example, Figs. 10-13 (a) and (b) show a 50 sample signal being interpolated into a 400 sample signal by this method. The interpolation is a smooth fit between the original points, much as if a curve fitting routine had been used. In comparison, (c) and (d) show another example where the time domain is a mess! The oscillatory behavior shown in (d) arises at edges or other discontinuities in the signal. This also includes any discontinuity between sample zero and N-1, since the time domain is viewed as being circular. This overshoot at discontinuities is called the *Gibbs effect*, and is discussed in Chapter 11. Another frequency domain interpolation technique is presented in Chapter 3, adding zeros between the time domain samples and low-pass filtering.

Multiplying Signals (Amplitude Modulation)

An important Fourier transform property is that *convolution* in one domain corresponds to *multiplication* in the other domain. One side of this was

discussed in the last chapter: time domain signals can be convolved by multiplying their frequency spectra. Amplitude modulation is an example of the reverse situation, multiplication in the time domain corresponds to convolution in the frequency domain. In addition, amplitude modulation provides an excellent example of how the elusive *negative* frequencies enter into everyday science and engineering problems.

Audio signals are great for short distance communication; when you speak, someone across the room hears you. On the other hand, radio frequencies are very good at propagating long distances. For instance, if a 100 volt, 1 MHz sine wave is fed into an antenna, the resulting radio wave can be detected in the next *room*, the next *country*, and even on the next *planet*. **Modulation** is the process of merging two signals to form a third signal with desirable characteristics of both. This always involves nonlinear processes such as multiplication; you can't just add the two signals together. In radio communication, modulation results in radio signals that can propagate long distances *and* carry along audio or other information.

Radio communication is an extremely well developed discipline, and many modulation schemes have been developed. One of the simplest is called **amplitude modulation**. Figure 10-14 shows an example of how amplitude modulation appears in both the time and frequency domains. Continuous signals will be used in this example, since modulation is usually carried out in analog electronics. However, the whole procedure could be carried out in discrete form if needed (the shape of the future!).

Figure (a) shows an audio signal with a DC bias such that the signal always has a positive value. Figure (b) shows that its frequency spectrum is composed of frequencies from 300 Hz to 3 kHz, the range needed for voice communication, plus a spike for the DC component. All other frequencies have been removed by analog filtering. Figures (c) and (d) show the **carrier wave**, a pure sinusoid of much higher frequency than the audio signal. In the time domain, amplitude modulation consists of *multiplying* the audio signal by the carrier wave. As shown in (e), this results in an oscillatory waveform that has an instantaneous amplitude proportional to the original audio signal. In the jargon of the field, the *envelope* of the carrier wave is equal to the modulating signal. This signal can be routed to an antenna, converted into a radio wave, and then detected by a receiving antenna. This results in a signal identical to (e) being generated in the radio receiver's electronics. A *detector* or *demodulator* circuit is then used to convert the waveform in (e) back into the waveform in (a).

Since the time domain signals are multiplied, the corresponding frequency spectra are convolved. That is, (f) is found by convolving (b) & (d). Since the spectrum of the carrier is a shifted delta function, the spectrum of the

Time (milliseconds)
0.0 0.2 0.4 0.6 0.8 1.0
-2
-1
0
1
2
a. Audio signal
Frequency (kHz)
0 5 10 15 20 25 30 35 40
0
1
2
3
4
b. Audio signal
Frequency (kHz)
0 5 10 15 20 25 30 35 40

0
1
2
3
4
d. Carrier signal
Frequency Domain Time Domain
Time (milliseconds)
0.0 0.2 0.4 0.6 0.8 1.0
-2
-1
0
1
2
c. Carrier signal
Frequency (kHz)
0 5 10 15 20 25 30 35 40
0
1
2
3
4
f. Modulated signal
lower
sideband
upper
sideband carrier
Time (milliseconds)
0.0 0.2 0.4 0.6 0.8 1.0
-2
-1
0
1
2
e. Modulated signal
FIGURE 10-14
Amplitude modulation. In the time domain, amplitude modulation is achieved by multiplying the audio signal, (a), by the carrier signal, (c), to produce the modulated signal, (e). Since multiplication in the time domain corresponds to convolution in the frequency domain, the spectrum of the modulated signal is the spectrum of the audio signal shifted to the frequency of the carrier.
Amplitude
Voltage
Amplitude Amplitude
Voltage Voltage
modulated signal is equal to the audio spectrum *shifted* to the frequency of the carrier. This results in a modulated spectrum composed of three components: a **carrier wave**, an **upper sideband**, and a **lower sideband**.
These correspond to the three parts of the original audio signal: the DC component, the positive frequencies between 0.3 and 3 kHz, and the negative

EQUATION 10-1
The DTFT analysis equation. In this relation, is the time domain signal $x[n]$ with n running from 0 to . The N&1 frequency spectrum is held in: (T) $Re\ X$ and (T), with T between 0 and pi. $Im\ X$

$$ReX(T) ' j\%4$$
$$n' \&4$$
$$x\ [n]\ \cos(Tn)$$
$$ImX(T) ' \& j\%4$$
$$n' \&4$$
$$x\ [n]\ \sin(Tn)$$
$$x[n] ' 1$$
B m
B
0
$$ReX(T)\ \cos\ (Tn) \& ImX(T)\ \sin(Tn)\ dT$$ EQUATION 10-2
The DTFT synthesis

equation.
frequencies between -0.3 and -3 kHz, respectively. Even though the negative frequencies in the original audio signal are somewhat elusive and abstract, the resulting frequencies in the lower sideband are as real as you could want them to be. The ghosts have taken human form!
Communication engineers live and die by this type of frequency domain analysis. For example, consider the frequency spectrum for television transmission. A standard TV signal has a frequency spectrum from DC to 6 MHz. By using these frequency shifting techniques, 82 of these 6 MHz wide channels are stacked end-to-end. For instance, channel 3 is from 60 to 66 MHz, channel 4 is from 66 to 72 MHz, channel 83 is from 884 to 890 MHz, etc. The television receiver moves the desired channel back to the DC to 6 MHz band for display on the screen. This scheme is called **frequency domain multiplexing**.

The Discrete Time Fourier Transform

The Discrete Time Fourier Transform (DTFT) is the member of the Fourier transform family that operates on *aperiodic*, *discrete* signals. The best way to understand the DTFT is how it relates to the DFT. To start, imagine that you acquire an N sample signal, and want to find its frequency spectrum. By using the DFT, the signal can be decomposed into $N/2$ 1 sine and cosine waves, with frequencies equally spaced between zero and one-half of the sampling rate. As discussed in the last chapter, padding the time domain signal with zeros makes the period of the time domain *longer*, as well as making the spacing between samples in the frequency domain *narrower*. As N approaches infinity, the time domain becomes *aperiodic*, and the frequency domain becomes a *continuous* signal. This is the DTFT, the Fourier transform that relates an *aperiodic*, *discrete* signal, with a *periodic*, *continuous* frequency spectrum.
The mathematics of the DTFT can be understood by starting with the synthesis and analysis equations for the DFT (Eqs. 8-2, 8-3 and 8-4), and taking N to infinity:
Chapter 10- Fourier Transform Properties 207
There are many subtle details in these relations. First, the time domain signal, , is still discrete, and therefore is represented by *brackets*. In comparison, $x[n]$ the frequency domain signals, & , are continuous, and are thus $ReX(T)$ $Im\ X(T)$ written with *parentheses*. Since the frequency domain is continuous, the synthesis equation must be written as an integral, rather than a summation. As discussed in Chapter 8, frequency is represented in the DFT's frequency domain by one of three variables: k, an index that runs from 0 to $N/2$; f, the fraction of the sampling rate, running from 0 to 0.5; or T, the fraction of the sampling rate expressed as a natural frequency, running from 0 to B. The spectrum of the DTFT is continuous, so either or T can be used. The f common choice is T, because it makes the equations shorter by eliminating the always present factor of . Remember, when T is used, the frequency 2B spectrum extends from 0 to B, which corresponds to DC to one-half of the sampling rate. To make things even more complicated, many authors use S (an upper case omega) to represent this frequency in the DTFT, rather than T (a lower case omega).
When calculating the inverse DFT, samples 0 and $N/2$ must be divided by two (Eq. 8-3) before the synthesis can be carried out (Eq. 8-2). This is not necessary with the DTFT. As you recall, this action in the DFT is related to the frequency spectrum being defined as a *spectral density*, i.e., amplitude per unit of bandwidth. When the spectrum becomes continuous,

the special treatment of the end points disappear. However, there is still a normalization factor that must be included, the 2/*N* in the DFT (Eq. 8-3) becomes 1/B in the DTFT (Eq. 10-2). Some authors place these terms in front of the *synthesis* equation, while others place them in front of the *analysis* equation. Suppose you start with some time domain signal. After taking the Fourier transform, and then the Inverse Fourier transform, you want to end up with what you started. That is, the 1/B term (or the 2/*N* term) must be encountered somewhere along the way, either in the forward or in the inverse transform. Some authors even split the term between the two transforms by placing in front of both. 1/ B

Since the DTFT involves infinite summations and integrals, it cannot be calculated with a digital computer. Its main use is in theoretical problems as an alternative to the DFT. For instance, suppose you want to find the frequency response of a system from its impulse response. If the impulse response is known as an *array of numbers*, such as might be obtained from an experimental measurement or computer simulation, a DFT program is run on a computer. This provides the frequency spectrum as another *array of numbers*, equally spaced between 0 and 0.5 of the sampling rate.

In other cases, the impulse response might be know as an *equation*, such as a sinc function (described in the next chapter) or an exponentially decaying sinusoid. The DTFT is used here to mathematically calculate the frequency domain as another *equation*, specifying the entire continuous curve between 0 and 0.5. While the DFT could also be used for this calculation, it would only provide an equation for *samples* of the frequency response, not the entire curve.

EQUATION 10-3

Parseval's relation. In this equation, is $x[i]$ a time domain signal with i running from 0 to N-1, and is its *modified* frequency $X[k]$ spectrum, with k running from 0 to $N/2$. The *modified* frequency spectrum is found by taking the DFT of the signal, and dividing the first and last frequencies (sample 0 and $N/2$) by the square-root of two.

$$\sum_{i=0}^{N-1} x[i]^2 = \frac{2}{N} \sum_{k=0}^{N/2} Mag\ X[k]^2$$

Parseval's Relation

Since the time and frequency domains are equivalent representations of the same signal, they must have the same *energy*. This is called Parseval's relation, and holds for all members of the Fourier transform family. For the DFT, Parseval's relation is expressed:

The left side of this equation is the total energy contained in the time domain signal, found by summing the energies of the N individual samples. Likewise, the right side is the energy contained in the frequency domain, found by summing the energies of the sinusoids. Remember from physics that $N/2\% 1$ *energy* is proportional to the *amplitude squared*. For example, the energy in a spring is proportional to the displacement squared, and the energy stored in

a capacitor is proportional to the voltage squared. In Eq. 10-3, is the $X[f]$ frequency spectrum of , with one slight modification: the first and last $x[n]$ frequency components, & , have been divided by . This $X[0]$ $X[N/2]$ 2 modification, along with the $2/N$ factor on the right side of the equation, accounts for several subtle details of calculating and summing energies.
To understand these corrections, start by finding the frequency domain representation of the signal by using the DFT. Next, convert the frequency domain into the amplitudes of the sinusoids needed to reconstruct the signal, as previously defined in Eq. 8-3. This is done by dividing the first and last points (sample 0 and $N/2$) by 2, and then dividing all of the points by $N/2$. While this provides the amplitudes of the sinusoids, they are expressed as a *peak* amplitude, not the *root-mean-square* (rms) amplitude needed for energy calculations. In a sinusoid, the peak amplitude is converted to rms by dividing by . This correction must be made to all of the frequency domain values, 2 *except* sample 0 and $N/2$. This is because these two sinusoids are unique; one is a constant value, while the other alternates between two constant values. For these two special cases, the *peak* amplitude is already equal to the *rms* value.
All of the values in the frequency domain are squared and then summed. The last step is to divide the summed value by N, to account for each sample in the frequency domain being converted into a sinusoid that covers N values in the time domain. Working through all of these details produces Eq. 10-3.
While Parseval's relation is interesting from the physics it describes (conservation of energy), it has few practical uses in DSP.

CHAPTER

11 Fourier Transform Pairs

For every *time domain* waveform there is a corresponding *frequency domain* waveform, and vice versa. For example, a rectangular pulse in the time domain coincides with a sinc function [i.e., sin(x)/x] in the frequency domain. Duality provides that the reverse is also true; a rectangular pulse in the frequency domain matches a sinc function in the time domain. Waveforms that correspond to each other in this manner are called *Fourier transform pairs*. Several common pairs are presented in this chapter.

Delta Function Pairs

For discrete signals, the delta function is a simple waveform, and has an equally simple Fourier transform pair. Figure 11-1a shows a delta function in the time domain, with its frequency spectrum in (b) and (c). The magnitude is a constant value, while the phase is entirely zero. As discussed in the last chapter, this can be understood by using the expansion/compression property. When the time domain is compressed until it becomes an impulse, the frequency domain is expanded until it becomes a constant value.
In (d) and (g), the time domain waveform is shifted four and eight samples to the right, respectively. As expected from the properties in the last chapter, shifting the time domain waveform does not affect the magnitude, but adds a linear component to the phase. The phase signals in this figure have not been *unwrapped*, and thus extend only from -B to B. Also notice that the horizontal axes in the frequency domain run from -0.5 to 0.5. That is, they show the *negative* frequencies in the spectrum, as well as the *positive* ones. The negative frequencies are redundant information, but they are often included in DSP graphs and you should become accustomed to seeing them.
Figure 11-2 presents the same information as Fig. 11-1, but with the

frequency domain in *rectangular form*. There are two lessons to be learned here. First, compare the polar and rectangular representations of the

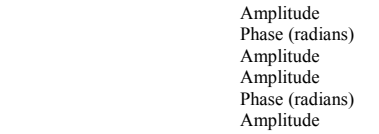
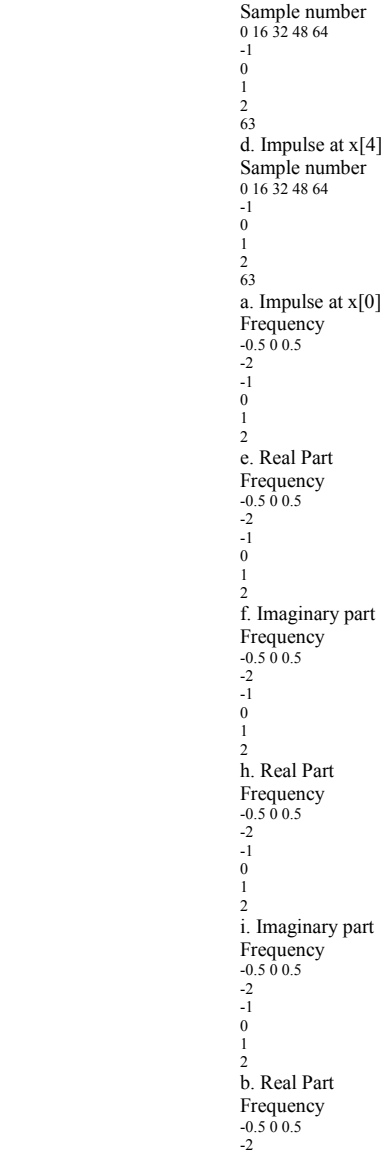


FIGURE 11-1
Delta function pairs in *polar form*. An impulse in the time domain corresponds to a constant magnitude and a linear phase in the frequency domain.

frequency domains. As is usually the case, the polar form is much easier to understand; the magnitude is nothing more than a constant, while the phase is a straight line. In comparison, the real and imaginary parts are sinusoidal oscillations that are difficult to attach a meaning to.

The second interesting feature in Fig. 11-2 is the *duality* of the DFT. In the conventional view, each sample in the DFT's frequency domain corresponds to a sinusoid in the time domain. However, the reverse of this is also true, each sample in the time domain corresponds to sinusoids in the frequency domain. Including the negative frequencies in these graphs allows the duality property to be more symmetrical. For instance, Figs. (d), (e), and



-1
0
1
2
c. Imaginary part
Sample number
0 16 32 48 64
-1
0
1
2
63
g. Impulse at x[8]
Frequency Domain Time Domain
Amplitude
Amplitude
Amplitude
Amplitude
Amplitude
Amplitude
Amplitude
Amplitude
Amplitude
FIGURE 11-2
Delta function pairs in *rectangular form*. Each sample in the time domain results in a cosine wave in the real part, and a negative sine wave in the imaginary part of the frequency domain.

(f) show that an impulse at sample number four in the time domain results in four cycles of a cosine wave in the real part of the frequency spectrum, and four cycles of a negative sine wave in the imaginary part. As you recall, an impulse at sample number four in the real part of the frequency spectrum results in four cycles of a cosine wave in the time domain. Likewise, an impulse at sample number four in the imaginary part of the frequency spectrum results in four cycles of a negative sine wave being added to the time domain wave.

As mentioned in Chapter 8, this can be used as another way to calculate the DFT (besides correlating the time domain with sinusoids). Each sample in the time domain results in a cosine wave being added to the real part of the

EQUATION 11-1
DFT spectrum of a rectangular pulse. In this equation, N is the number of points in the time domain signal, all of which have a value of zero, except M adjacent points that have a value of one. The frequency spectrum is contained in , where k runs from 0 to $X[k]$ $N/2$. To avoid the division by zero, use . The *sine* function uses radians, $X[0] \,'\, M$ not degrees. This equation takes into account that the signal is *aliased*.

$$Mag\ X\ [\ k]\ '\ /$$
$$00$$
$$0\ \sin(\mathrm{B}kM/N\ )$$
$$\sin\ (\mathrm{B}k\ /N\ )$$
$$/00$$
$$0$$

frequency domain, and a negative sine wave being added to the imaginary part. The *amplitude* of each sinusoid is given by the *amplitude* of the time domain sample. The *frequency* of each sinusoid is provided by the *sample number* of the time domain point. The algorithm involves: (1) stepping through each time domain sample, (2) calculating the sine and cosine waves that correspond to each sample, and (3) adding up all of the contributing sinusoids. The resulting program is nearly identical to the correlation method (Table 8-2), except that

the outer and inner loops are exchanged.

The Sinc Function

Figure 11-4 illustrates a common transform pair: the *rectangular pulse* and the *sinc function* (pronounced “sink”). The sinc function is defined as:

, however, it is common to see the vague statement: "the sinc(*a*) ’ sin(B*a*)/ (B*a*) sinc function is of the general form: ." In other words, the sinc is a sine sin(*x*)/*x* wave that decays in amplitude as 1/*x*. In (a), the rectangular pulse is symmetrically centered on sample zero, making one-half of the pulse on the right of the graph and the other one-half on the left. This appears to the DFT as a single pulse because of the time domain periodicity. The DFT of this signal is shown in (b) and (c), with the *unwrapped* version in (d) and (e).

First look at the unwrapped spectrum, (d) and (e). The *unwrapped magnitude* is an oscillation that decreases in amplitude with increasing frequency. The phase is composed of all zeros, as you should expect for a time domain signal that is symmetrical around sample number zero. We are using the term *unwrapped magnitude* to indicate that it can have both positive and negative values. By definition, the *magnitude* must always be positive. This is shown in (b) and (c) where the magnitude is made all positive by introducing a phase shift of B at all frequencies where the unwrapped magnitude is negative in (d).

In (f), the signal is shifted so that it appears as one contiguous pulse, but is no longer centered on sample number zero. While this doesn't change the magnitude of the frequency domain, it does add a linear component to the phase, making it a jumbled mess. What does the frequency spectrum look like as real and imaginary parts ? Too confusing to even worry about.

An *N* point time domain signal that contains a unity amplitude rectangular pulse *M* points wide, has a DFT frequency spectrum given by:

Frequency
0 0.1 0.2 0.3 0.4 0.5
-6
-4
-2
0
2
4
6
e. Phase
Frequency
0 0.1 0.2 0.3 0.4 0.5
-5
0
5
10
15
20
g. Magnitude
Frequency
0 0.1 0.2 0.3 0.4 0.5
-6
-4
-2
0
2
4
6
h. Phase
Frequency
0 0.1 0.2 0.3 0.4 0.5
-5
0
5
10
15
20
b. Magnitude
Frequency
0 0.1 0.2 0.3 0.4 0.5
-6
-4

-2
0
2
4
6
c. Phase
Sample number
0 32 64 96 128
-1
0
1
2
127
f. Rectangular pulse
Frequency Domain Time Domain
or
Amplitude
Phase (radians)
Amplitude
Phase (radians)
Amplitude
Phase (radians)
Amplitude
FIGURE 11-3
DFT of a rectangular pulse. A rectangular pulse in one domain corresponds to a sinc function in the other domain.
Sample number
0 32 64 96 128
-1
0
1
2
127
a. Rectangular pulse
Frequency
0 0.1 0.2 0.3 0.4 0.5
-5
0
5
10
15
20
d. *Unwrapped* Magnitude
Amplitude
EQUATION 11-2
Equation 11-1 rewritten in terms of the sampling frequency. The parameter, , is *f* the fraction of the sampling rate, running continiously from 0 to 0.5. To avoid the division by zero, use . *Mag X*(0)' *M*

$$Mag\,X(f)\text{'}/$$

00

$$0\ \sin(\mathrm{B}fM)$$

$$\sin(\mathrm{B}f)$$

/000

Alternatively, the DTFT can be used to express the frequency spectrum as a fraction of the sampling rate, *f*:

In other words, Eq. 11-1 provides *samples* in the frequency spectrum, *N*/2% 1 while Eq. 11-2 provides the *continuous curve* that the samples lie on. These

x
0.0 0.2 0.4 0.6 0.8 1.0 1.2 1.4 1.6
0
0.2
0.4
0.6
0.8
1
1.2
1.4
1.6

y(x) = x
y(x) = sin(x)
FIGURE 11-4
Comparing x and $sin(x)$. The functions: , $y(x)$ ' x and are similar for small values of x, $y(x)$ ' $sin(x)$ and only differ by about 36% at 1.57 (B/2). This describes how aliasing distorts the frequency spectrum of the rectangular pulse from a pure sinc function.
y(x)
equations only provide the magnitude. The phase is determined solely by the left-right positioning of the time domain waveform, as discussed in the last chapter.
Notice in Fig. 11-3b that the amplitude of the oscillation does not decay to zero before a frequency of 0.5 is reached. As you should suspect, the waveform continues into the next period where it is *aliased*. This changes the shape of the frequency domain, an effect that is included in Eqs. 11-1 and 11-2.
It is often important to understand what the frequency spectrum looks like when aliasing *isn't* present. This is because *discrete signals* are often used to represent or model *continuous signals*, and continuous signals don't alias. To remove the aliasing in Eqs. 11-1 and 11-2, change the denominators from respectively. Figure 11-4 shows sin (B k / N) to B k / N and from sin (B f) to B f, the significance of this. The quantity can only run from 0 to 1.5708, since Bf f can only run from 0 to 0.5. Over this range there isn't much difference between and . At zero frequency they have the same value, and sin(Bf) Bf at a frequency of 0.5 there is only about a 36% difference. Without aliasing, the curve in Fig. 11-3b would show a slightly lower amplitude near the right side of the graph, and no change near the left side.
When the frequency spectrum of the rectangular pulse is *not* aliased (because the time domain signal is continuous, or because you are ignoring the aliasing), it is of the general form: , i.e., a sinc function. For sin (x)/ x continuous signals, the *rectangular pulse* and the *sinc function* are Fourier transform pairs. For discrete signals this is only an approximation, with the error being due to aliasing.
The sinc function has an annoying problem at , where becomes x ' 0 sin (x)/x *zero* divided by *zero*. This is not a difficult mathematical problem; as x becomes very small, approaches the value of x (see Fig. 11-4). $sin(x)$

EQUATION 11-3
Inverse DFT of the rectangular pulse. In the frequency domain, the pulse has an amplitude of one, and runs from sample number 0 through sample number M-1. The parameter N is the length of the DFT, and is the time domain signal with i running $x[i]$ from 0 to N-1. To avoid the division by zero, use . $x[0]$ ' (2M&1)/N
$x [i]$ ' 1
N
sin (2Bi (M & 1/2) /N)
sin(Bi /N)
This turns the sinc function into , which has a value of *one*. In other words, x/x as x becomes smaller and smaller, the value of approaches *one*, which $sinc$ (x) includes . Now try to tell your computer this! All it sees is a $sinc$ (0) ' 1

division by zero, causing it to complain and stop your program. The important point to remember is that your program must include special handling at x ' 0 when calculating the sinc function.

A key trait of the sinc function is the location of the **zero crossings**. These occur at frequencies where an integer number of the sinusoid's cycles fit evenly into the rectangular pulse. For example, if the rectangular pulse is 20 points wide, the first zero in the frequency domain is at the frequency that makes one complete cycle in 20 points. The second zero is at the frequency that makes two complete cycles in 20 points, etc. This can be understood by remembering how the DFT is calculated by correlation. The amplitude of a frequency component is found by multiplying the time domain signal by a sinusoid and adding up the resulting samples. If the time domain waveform is a rectangular pulse of unity amplitude, this is the same as *adding* the sinusoid's samples that are within the rectangular pulse. If this summation occurs over an integral number of the sinusoid's cycles, the result will be zero.

The sinc function is widely used in DSP because it is the Fourier transform pair of a very simple waveform, the rectangular pulse. For example, the sinc function is used in *spectral analysis*, as discussed in Chapter 9. Consider the analysis of an infinitely long discrete signal. Since the DFT can only work with *finite* length signals, N samples are selected to represent the longer signal. The key here is that "selecting N samples from a longer signal" is the same as multiplying the longer signal by a rectangular pulse. The *ones* in the rectangular pulse retain the corresponding samples, while the *zeros* eliminate them. How does this affect the frequency spectrum of the signal? Multiplying the time domain by a rectangular pulse results in the frequency domain being *convolved* with a *sinc function*. This reduces the frequency spectrum's resolution, as previously shown in Fig. 9-5a.

Other Transform Pairs

Figure 11-5 (a) and (b) show the duality of the above: a rectangular pulse in the frequency domain corresponds to a sinc function (plus aliasing) in the time domain. Including the effects of aliasing, the time domain signal is given by:

EQUATION 11-4

Inverse DTFT of the rectangular pulse. In the frequency domain, the pulse has an amplitude of one, and runs from zero frequency to the cutoff frequency, , a value f_c between 0 and 0.5. The time domain signal is held in with i running from 0 to N-1. To $x[i]$ avoid the division by zero, use . $x[0]$ ' $2f_c$

$$x[i] = \frac{\sin(2\pi f_c i)}{i\pi}$$

To eliminate the effects of aliasing from this equation, imagine that the frequency domain is so finely sampled that it turns into a continuous curve. This makes the time domain infinitely long with no periodicity. The DTFT is the Fourier transform to use here, resulting in the time domain signal being given by the relation:

This equation is very important in DSP, because the rectangular pulse in the frequency domain is the perfect *low-pass filter*. Therefore, the sinc function described by this equation is the filter kernel for the perfect low-pass filter. This is the basis for a very useful class of digital filters called the *windowedsinc filters*, described in Chapter 15.

Figures (c) & (d) show that a triangular pulse in the time domain coincides with a sinc function *squared* (plus aliasing) in the frequency domain. This transform pair isn't as important as the *reason* it is true. A point 2*M*& 1 triangle in the time domain can be formed by convolving an *M* point rectangular pulse with itself. Since convolution in the time domain results in multiplication in the frequency domain, convolving a waveform with itself will *square* the frequency spectrum.

Is there a waveform that is its own Fourier Transform? The answer is yes, and there is *only* one: the Gaussian. Figure (e) shows a Gaussian curve, and (f) shows the corresponding frequency spectrum, also a Gaussian curve. This relationship is only true if you ignore aliasing. The relationship between the standard deviation of the time domain and frequency domain is given by:

. While only one side of a Gaussian is shown in (f), the negative 2BF$_f$ ' 1/F$_t$ frequencies in the spectrum complete the full curve, with the center of symmetry at zero frequency.

Figure (g) shows what can be called a **Gaussian burst**. It is formed by multiplying a sine wave by a Gaussian. For example, (g) is a sine wave multiplied by the same Gaussian shown in (e). The corresponding frequency domain is a Gaussian centered somewhere other than zero frequency. As before, this transform pair is not as important as the *reason* it is true. Since the time domain signal is the multiplication of two signals, the frequency domain will be the convolution of the two frequency spectra. The frequency spectrum of the sine wave is a delta function centered at the frequency of the sine wave. The frequency spectrum of a Gaussian is a Gaussian centered at zero frequency. Convolving the two produces a Gaussian centered at the frequency of the sine wave. This should look familiar; it is identical to the procedure of *amplitude modulation* described in the last chapter.

Sample number
0 16 32 48 64 80 96 112 128
-1
0
1
2
a. Sinc
127
3
Frequency
0 0.1 0.2 0.3 0.4 0.5
-5
0
5
10
15
b. Rectangular pulse
Frequency
0 0.1 0.2 0.3 0.4 0.5
-5
0
5
10
15
d. Sinc squared

Frequency Domain Time Domain

Sample number
0 16 32 48 64 80 96 112 128
-1
0
1
2
c. Triangle
127
Sample number
0 16 32 48 64 80 96 112 128

FIGURE 11-5
Common transform pairs.

Gibbs Effect

Figure 11-6 shows a time domain signal being synthesized from sinusoids. The signal being reconstructed is shown in the last graph, (h). Since this signal is 1024 points long, there will be 513 individual frequencies needed for a complete reconstruction. Figures (a) through (g) show what the reconstructed signal looks like if only *some* of these frequencies are used. For example, (f) shows a reconstructed signal using frequencies 0 through 100. This signal was created by taking the DFT of the signal in (h), setting frequencies 101 through 512 to a value of zero, and then using the Inverse DFT to find the resulting time domain signal.

As more frequencies are added to the reconstruction, the signal becomes closer to the final solution. The interesting thing is *how* the final solution is approached at the *edges* in the signal. There are three sharp edges in (h). Two are the edges of the rectangular pulse. The third is between sample numbers 1023 and 0, since the DFT views the time domain as periodic. When only some of the frequencies are used in the reconstruction, each edge shows *overshoot* and *ringing* (decaying oscillations). This overshoot and ringing is known as the **Gibbs effect,** after the mathematical physicist Josiah Gibbs, who explained the phenomenon in 1899.

Look closely at the overshoot in (e), (f), and (g). As more sinusoids are added, the *width* of the overshoot decreases; however, the *amplitude* of the overshoot remains about the same, roughly 9 percent. With discrete signals this is not a problem; the overshoot is eliminated when the last frequency is added. However, the reconstruction of continuous signals cannot be explained so easily. An infinite number of sinusoids must be added to synthesize a continuous signal. The problem is, the amplitude of the overshoot does not decrease as the number of sinusoids approaches infinity, it stays about the same

9%. Given this situation (and other arguments), it is reasonable to question if a summation of continuous sinusoids *can* reconstruct an edge. Remember the squabble between Lagrange and Fourier?

The critical factor in resolving this puzzle is that the *width* of the overshoot becomes smaller as more sinusoids are included. The overshoot is still present with an infinite number of sinusoids, but it has *zero* width. Exactly at the discontinuity the value of the reconstructed signal converges to the midpoint of the step. As shown by Gibbs, the summation converges to the signal in the sense that the *error* between the two has zero energy.

Problems related to the Gibbs effect are frequently encountered in DSP. For example, a low-pass filter is a *truncation* of the higher frequencies, resulting in overshoot and ringing at the edges in the *time domain*. Another common procedure is to truncate the ends of a time domain signal to prevent them from extending into neighboring periods. By duality, this distorts the edges in the *frequency domain*. These issues will resurface in future chapters on filter design.

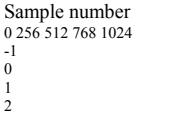
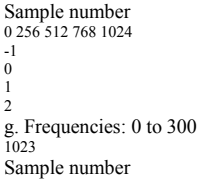


FIGURE 11-6.
The Gibbs effect.



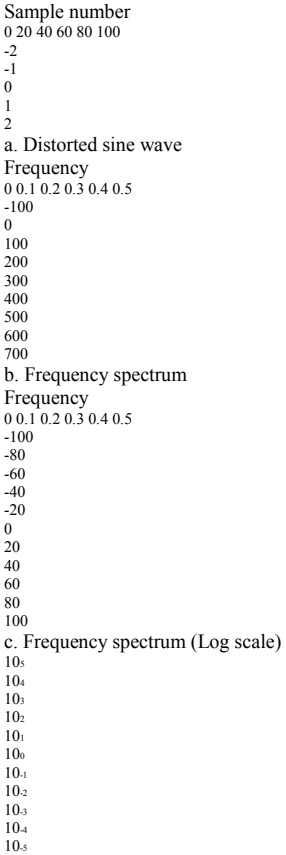
0 256 512 768 1024
-1
0
1
2
h. Frequencies: 0 to 512
1023
Amplitude Amplitude Amplitude Amplitude
Amplitude Amplitude Amplitude Amplitude

Sample number
0 128 256 384 512 640 768 896 1024
-2
-1
0
1
2
a. Sine wave
1023
Frequency
0 0.02 0.04 0.06 0.08 0.1
0
100
200
300
400
500
600
700
b. Fundamental
Frequency
0 0.02 0.04 0.06 0.08 0.1
0
100
200
300
400
500
600
700
d. Fundamental plus
even and odd harmonics
Frequency Domain Time Domain
Sample number
0 128 256 384 512 640 768 896 1024
-2
-1
0
1
2
c. Asymmetrical distortion
1023
Sample number
0 128 256 384 512 640 768 896 1024
-2
-1
0
1
2
e. Symmetrical distortion
1023
Frequency
0 0.02 0.04 0.06 0.08 0.1
0
100
200
300
400
500
600
700
f. Fundamental plus
odd harmonics
FIGURE 11-7
Example of harmonics. Asymmetrical distortion, shown in (c), results in even and odd harmonics, (d), while symmetrical distortion, shown in (e), produces only odd harmonics, (f).
Amplitude
Amplitude Amplitude Amplitude
Amplitude Amplitude

Harmonics

If a signal is periodic with frequency *f*, the only frequencies composing the signal are integer multiples of *f*, i.e., *f*, 2*f*, 3*f*, 4*f*, etc. These frequencies are

called **harmonics.** The **first harmonic** is *f*, the **second harmonic** is 2*f*, the **third harmonic** is 3*f*, and so forth. The first harmonic (i.e., *f*) is also given a special name, the **fundamental frequency**. Figure 11-7 shows an



Frequency Domain Time Domain

FIGURE 11-8
Harmonic aliasing. Figures (a) and (b) show a distorted sine wave and its frequency spectrum, respectively. Harmonics with a frequency greater than 0.5 will become aliased to a frequency between 0 and 0.5. Figure (c) displays the same frequency spectrum on a logarithmic scale, revealing many aliased peaks with very low amplitude.

Amplitude
Amplitude Amplitude

example. Figure (a) is a pure sine wave, and (b) is its DFT, a single peak. In (c), the sine wave has been distorted by poking in the tops of the peaks. Figure (d) shows the result of this distortion in the frequency domain. Because the distorted signal is periodic with the same frequency as the original sine wave, the frequency domain is composed of the original peak plus harmonics. Harmonics can be of any amplitude; however, they usually become smaller as they increase in frequency. As with any signal, *sharp edges* result in *higher frequencies*. For example, consider a common TTL logic gate generating a 1 kHz square wave. The edges rise in a few nanoseconds, resulting in harmonics being generated to nearly 100 MHz, the *ten-thousandth* harmonic!

Figure (e) demonstrates a subtlety of harmonic analysis. If the signal is

symmetrical around a horizontal axis, i.e., the top lobes are mirror images of the bottom lobes, all of the even harmonics will have a value of zero. As shown in (f), the only frequencies contained in the signal are the fundamental, the third harmonic, the fifth harmonic, etc.

All *continuous* periodic signals can be represented as a summation of harmonics, just as described. *Discrete* periodic signals have a problem that disrupts this simple relation. As you might have guessed, the problem is *aliasing*. Figure 11-8a shows a sine wave distorted in the same manner as before, by poking in the tops of the peaks. This waveform looks much less

EQUATION 11-7

Phase of the chirp system. $Phase\ X[k] = \pi k^2$

regular and smooth than in the previous example because the sine wave is at a much higher frequency, resulting in fewer samples per cycle. Figure (b) shows the frequency spectrum of this signal. As you would expect, you can identify the fundamental and harmonics. This example shows that harmonics can extend to frequencies greater than 0.5 of the sampling frequency, and will be *aliased* to frequencies somewhere between 0 and 0.5. You don't notice them in (b) because their amplitudes are too low. Figure (c) shows the frequency spectrum plotted on a logarithmic scale to reveal these low amplitude aliased peaks. At first glance, this spectrum looks like random noise. It isn't; this is a result of the many harmonics overlapping as they are aliased.

It is important to understand that this example involves distorting a signal *after* it has been digitally represented. If this distortion occurred in an analog signal, you would remove the offending harmonics with an antialias filter *before* digitization. Harmonic aliasing is only a problem when nonlinear operations are performed directly on a discrete signal. Even then, the amplitude of these aliased harmonics is often low enough that they can be ignored.

The concept of harmonics is also useful for another reason: it explains why the DFT views the time and frequency domains as *periodic*. In the frequency domain, an N point DFT consists of $N/2+1$ equally spaced frequencies. You can view the frequencies *between* these samples as (1) having a value of zero, or (2) not existing. Either way they don't contribute to the synthesis of the time domain signal. In other words, a *discrete* frequency spectrum consists of *harmonics*, rather than a continuous range of frequencies. This requires the time domain to be periodic with a frequency equal to the lowest sinusoid in the frequency domain, i.e., the fundamental frequency. Neglecting the DC value, the lowest frequency represented in the frequency domain makes one complete cycle every N samples, resulting in the time domain being periodic with a period of N. In other words, if one domain is *discrete*, the other domain must be *periodic*, and vice versa. This holds for all four members of the Fourier transform family. Since the DFT views both domains as discrete, it must also view both domains as periodic. The samples in each domain represent harmonics of the periodicity of the opposite domain.

Chirp Signals

Chirp signals are an ingenious way of handling a practical problem in echo location systems, such as radar and sonar. Figure 11-9 shows the frequency response of the chirp system. The magnitude has a constant value of one, while the phase is a parabola:

Frequency

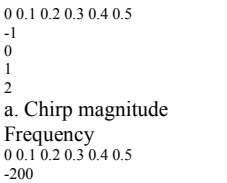


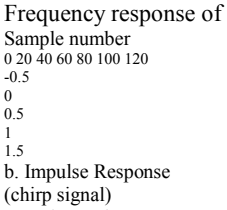
FIGURE 11-9
Frequency response of the chirp system. The magnitude is a constant, while the phase is a parabola.


FIGURE 11-10
The chirp system. The impulse response of a chirp system is a chirp signal.


The parameter " introduces a linear slope in the phase, that is, it simply shifts the impulse response left or right as desired. The parameter $ controls the *curvature* of the phase. These two parameters must be chosen such that the phase at frequency 0.5 (i.e. k = *N*/2) is a multiple of 2B. Remember, whenever the phase is directly manipulated, frequency 0 and 0.5 must both have a phase of zero (or a multiple of 2B, which is the same thing).

Figure 11-10 shows an impulse entering a chirp system, and the impulse response exiting the system. The impulse response is an oscillatory burst that starts at a low frequency and changes to a high frequency as time progresses. This is called a *chirp* signal for a very simple reason: it sounds like the chirp of a bird when played through a speaker.

The key feature of the chirp system is that it is completely *reversible*. If you run the chirp signal through an *antichirp* system, the signal is again made into an impulse. This requires the antichirp system to have a magnitude of one, and the *opposite* phase of the chirp system. As discussed in the last

chapter, this means that the impulse response of the antichirp system is found by preforming a left-for-right flip of the chirp system's impulse response. Interesting, but what is it good for?

Consider how a radar system operates. A short burst of radio frequency energy is emitted from a directional antenna. Aircraft and other objects reflect some of this energy back to a radio receiver located next to the transmitter. Since radio waves travel at a constant rate, the elapsed time between the transmitted and received signals provides the distance to the target. This brings up the first requirement for the pulse: it needs to be as short as possible. For example, a 1 microsecond pulse provides a radio burst about 300 meters long.

This means that the distance information we obtain with the system will have a resolution of about this same length. If we want better distance resolution, we need a shorter pulse.
The second requirement is obvious: if we want to detect objects farther away, you need more energy in your pulse. Unfortunately, *more energy* and *shorter pulse* are conflicting requirements. The electrical power needed to provide a pulse is equal to the energy of the pulse divided by the pulse length. Requiring both *more energy* and a *shorter pulse* makes electrical power handling a limiting factor in the system. The output stage of a radio transmitter can only handle so much power without destroying itself.
Chirp signals provide a way of breaking this limitation. Before the impulse reaches the final stage of the radio transmitter, it is passed through a chirp system. Instead of bouncing an impulse off the target aircraft, a chirp signal is used. After the chirp echo is received, the signal is passed through an antichirp system, restoring the signal to an impulse. This allows the portions of the system that measure distance to see short pulses, while the power handling circuits see long duration signals. This type of waveshaping is a fundamental part of modern radar systems.
225
CHAPTER

12 The Fast Fourier Transform

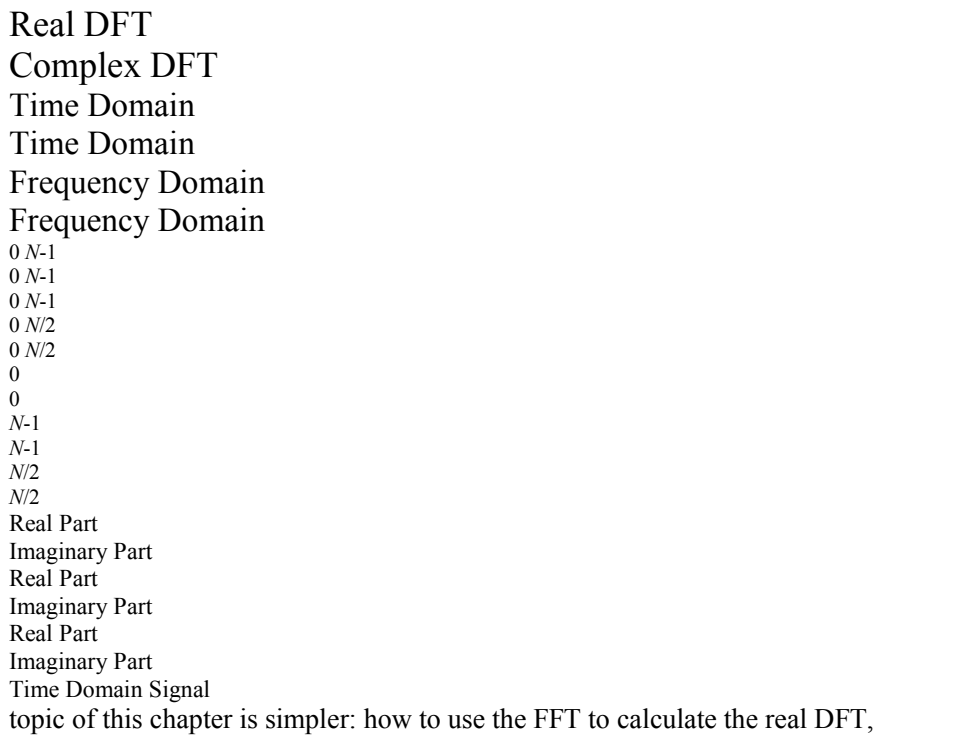
There are several ways to calculate the Discrete Fourier Transform (DFT), such as solving simultaneous linear equations or the *correlation* method described in Chapter 8. The Fast Fourier Transform (FFT) is another method for calculating the DFT. While it produces the same result as the other approaches, it is incredibly more efficient, often reducing the computation time by *hundreds*. This is the same improvement as flying in a jet aircraft versus walking! If the FFT were not available, many of the techniques described in this book would not be practical. While the FFT only requires a few dozen lines of code, it is one of the most complicated algorithms in DSP. But don't despair! You can easily use published FFT routines without fully understanding the internal workings.

Real DFT Using the Complex DFT

J.W. Cooley and J.W. Tukey are given credit for bringing the FFT to the world in their paper: "An algorithm for the machine calculation of complex Fourier Series," *Mathematics Computation*, Vol. 19, 1965, pp 297-301. In retrospect, others had discovered the technique many years before. For instance, the great German mathematician Karl Friedrich Gauss (1777-1855) had used the method more than a century earlier. This early work was largely forgotten because it lacked the tool to make it practical: the *digital computer*. Cooley and Tukey are honored because they discovered the FFT at the right time, the beginning of the computer revolution.
The FFT is based on the *complex DFT*, a more sophisticated version of the *real DFT* discussed in the last four chapters. These transforms are named for the way each represents data, that is, using complex numbers or using real numbers. The term *complex* does not mean that this representation is difficult or complicated, but that a specific type of mathematics is used. Complex mathematics often *is* difficult and complicated, but that isn't where the name comes from. Chapter 29 discusses the complex DFT and provides the background needed to understand the details of the FFT algorithm. The

FIGURE 12-1

Comparing the real and complex DFTs. The real DFT takes an *N* point time domain signal and creates two point frequency domain signals. The complex DFT takes two *N* point time *N*/2% 1 domain signals and creates two *N* point frequency domain signals. The crosshatched regions shows the values common to the two transforms.



topic of this chapter is simpler: how to use the FFT to calculate the real DFT, without drowning in a mire of advanced mathematics.

Since the FFT is an algorithm for calculating the complex DFT, it is important to understand how to transfer *real DFT* data into and out of the *complex DFT* format. Figure 12-1 compares how the real DFT and the complex DFT store data. The real DFT transforms an *N* point time domain signal into two point frequency domain signals. The time domain *N*/2 % 1 signal is called just that: the *time domain signal*. The two signals in the frequency domain are called the *real part* and the *imaginary part*, holding the amplitudes of the cosine waves and sine waves, respectively. This should be very familiar from past chapters.

In comparison, the complex DFT transforms two *N* point time domain signals into two *N* point frequency domain signals. The two time domain signals are called the *real part* and the *imaginary part*, just as are the frequency domain signals. In spite of their names, all of the values in these arrays are just ordinary numbers. (If you are familiar with complex numbers: the *j*'s are not included in the array values; they are a part of the *mathematics*. Recall that the operator, *Im*( ), returns a real number).


```
6000 'NEGATIVE FREQUENCY GENERATION
6010 'This subroutine creates the complex frequency domain from the real frequency domain.
6020 'Upon entry to this subroutine, N% contains the number of points in the signals, and
6030 'REX[ ] and IMX[ ] contain the real frequency domain in samples 0 to N%/2.
6040 'On return, REX[ ] and IMX[ ] contain the complex frequency domain in samples 0 to N%-1.
6050 '
6060 FOR K% = (N%/2+1) TO (N%-1)
6070 REX[K%] = REX[N%-K%]
6080 IMX[K%] = -IMX[N%-K%]
6090 NEXT K%
6100 '
```

6110 RETURN
TABLE 12-1
Suppose you have an *N* point signal, and need to calculate the *real DFT* by means of the *Complex DFT* (such as by using the FFT algorithm). First, move the *N* point signal into the real part of the complex DFT's time domain, and then set all of the samples in the imaginary part to *zero*. Calculation of the complex DFT results in a real and an imaginary signal in the frequency domain, each composed of *N* points. Samples 0 through *N*/2 of these signals correspond to the real DFT's spectrum.

As discussed in Chapter 10, the DFT's frequency domain is periodic when the negative frequencies are included (see Fig. 10-9). The choice of a single period is arbitrary; it can be chosen between -1.0 and 0, -0.5 and 0.5, 0 and 1.0, or any other one unit interval referenced to the sampling rate. The complex DFT's frequency spectrum includes the negative frequencies in the 0 to 1.0 arrangement. In other words, one full period stretches from sample 0 to sample , corresponding with 0 to 1.0 times the sampling rate. The positive *N*&1 frequencies sit between sample 0 and , corresponding with 0 to 0.5. The *N*/2 other samples, between and , contain the negative frequency *N*/2% 1 *N*&1 values (which are usually ignored).

Calculating a *real Inverse DFT* using a *complex Inverse DFT* is slightly harder. This is because you need to insure that the negative frequencies are loaded in the proper format. Remember, points 0 through in the *N*/2 complex DFT are the same as in the real DFT, for both the real and the imaginary parts. For the real part, point is the same as point *N*/2% 1 , point is the same as point , etc. This continues to *N*/2& 1 *N*/2% 2 *N*/2& 2 point being the same as point 1. The same basic pattern is used for *N*&1 the imaginary part, except the sign is changed. That is, point is the *N*/2% 1 negative of point , point is the negative of point , etc. *N*/2& 1 *N*/2% 2 *N*/2& 2 Notice that samples 0 and do not have a matching point in this *N*/2 duplication scheme. Use Fig. 10-9 as a guide to understanding this symmetry. In practice, you load the real DFT's frequency spectrum into samples 0 to of the complex DFT's arrays, and then use a subroutine to *N*/2 generate the negative frequencies between samples and . Table *N*/2% 1 *N*&1 12-1 shows such a program. To check that the proper symmetry is present, after taking the inverse FFT, look at the imaginary part of the time domain. It will contain all zeros if everything is correct (except for a few parts-permillion of noise, using single precision calculations).

FIGURE 12-2
The FFT decomposition. An *N* point signal is decomposed into *N* signals each containing a single point. Each stage uses an *interlace decomposition*, separating the even and odd numbered samples.

1 signal of
16 points
2 signals of
8 points
4 signals of
4 points
8 signals of
2 points
16 signals of
1 point

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
0 2 4 6 8 10 12 14 1 3 5 7 9 11 13 15
0 4 8 12 2 6 10 14 1 5 9 13 3 7 11 15
8 4 12 2 10 6 14 1 9 5 13 3 11 7 15 0
8 4 12 2 10 6 14 1 9 5 13 3 11 7 15 0

How the FFT works

The FFT is a complicated algorithm, and its details are usually left to those that specialize in such things. This section describes the general operation of the FFT, but skirts a key issue: the use of *complex numbers*. If you have a background in complex mathematics, you can read between the lines to understand the true nature of the algorithm. Don't worry if the details elude you; few scientists and engineers that use the FFT could write the program from scratch.

In complex notation, the time and frequency domains each contain *one signal* made up of *N complex points*. Each of these complex points is composed of two numbers, the real part and the imaginary part. For example, when we talk about complex sample , it refers to the combination of and *X*[42] *ReX*[42] . In other words, each complex variable holds two numbers. When *ImX*[42] two complex variables are multiplied, the four individual components must be combined to form the two components of the product (such as in Eq. 9-1). The following discussion on *"How the FFT works"* uses this jargon of complex notation. That is, the singular terms: *signal, point, sample*, and *value*, refer to the *combination* of the real part and the imaginary part.

The FFT operates by decomposing an *N* point time domain signal into *N* time domain signals each composed of a single point. The second step is to calculate the *N* frequency spectra corresponding to these *N* time domain signals. Lastly, the *N* spectra are synthesized into a single frequency spectrum.

Figure 12-2 shows an example of the time domain decomposition used in the FFT. In this example, a 16 point signal is decomposed through four

Sample numbers in normal order		Sample numbers after bit reversal	
Decimal	*Binary*	*Decimal*	*Binary*
0	0000	0	0000
1	0001	8	1000
2	0010	4	0100
3	0011	12	1100
4	0100	2	0010
5	0101	10	1010
6	0110	6	0100
7	0111	14	1110
8	1000	1	0001
9	1001	9	1001
10	1010	5	0101
11	1011	13	1101
12	1100	3	0011
13	1101	11	1011
14	1110	7	0111
15	1111	15	1111

FIGURE 12-3

The FFT bit reversal sorting. The FFT time domain decomposition can be implemented by sorting the samples according to bit reversed order.

separate stages. The first stage breaks the 16 point signal into two signals each consisting of 8 points. The second stage decomposes the data into four signals of 4 points. This pattern continues until there are *N* signals composed of a single point. An **interlaced decomposition** is used each time a signal is broken in two, that is, the signal is separated into its even and odd numbered samples. The best way to understand this is by inspecting Fig. 12-2 until you grasp the pattern. There are stages required in this decomposition, i.e., Log_2N a 16 point signal (2_4) requires 4 stages, a 512 point signal (2_7) requires 7

stages, a 4096 point signal (2_{12}) requires 12 stages, etc. Remember this value, ; it will be referenced many times in this chapter. Log_2N

Now that you understand the structure of the decomposition, it can be greatly simplified. The decomposition is nothing more than a *reordering* of the samples in the signal. Figure 12-3 shows the rearrangement pattern required. On the left, the sample numbers of the original signal are listed along with their binary equivalents. On the right, the rearranged sample numbers are listed, also along with their binary equivalents. The important idea is that the binary numbers are the *reversals* of each other. For example, sample 3 (0011) is exchanged with sample number 12 (1100). Likewise, sample number 14 (1110) is swapped with sample number 7 (0111), and so forth. The FFT time domain decomposition is usually carried out by a **bit reversal sorting** algorithm. This involves rearranging the order of the *N* time domain samples by counting in binary with the bits flipped left-for-right (such as in the far right column in Fig. 12-3).

a b c d
a b c d 0 0 0 0
A B C D
A B C D A B C D
e f g h
e f g h 0 0 0 0
E F G H
F G H E F G H
× sinusoid

Time Domain Frequency Domain

E

FIGURE 12-4
The FFT synthesis. When a time domain signal is diluted with zeros, the frequency domain is duplicated. If the time domain signal is also shifted by one sample during the dilution, the spectrum will additionally be multiplied by a sinusoid.

The next step in the FFT algorithm is to find the frequency spectra of the 1 point time domain signals. Nothing could be easier; the frequency spectrum of a 1 point signal is equal to *itself*. This means that *nothing* is required to do this step. Although there is no work involved, don't forget that each of the 1 point signals is now a frequency spectrum, and not a time domain signal.

The last step in the FFT is to combine the *N* frequency spectra in the exact reverse order that the time domain decomposition took place. This is where the algorithm gets messy. Unfortunately, the bit reversal shortcut is not applicable, and we must go back one stage at a time. In the first stage, 16 frequency spectra (1 point each) are synthesized into 8 frequency spectra (2 points each). In the second stage, the 8 frequency spectra (2 points each) are synthesized into 4 frequency spectra (4 points each), and so on. The last stage results in the output of the FFT, a 16 point frequency spectrum.

Figure 12-4 shows how two frequency spectra, each composed of 4 points, are combined into a single frequency spectrum of 8 points. This synthesis must *undo* the interlaced decomposition done in the time domain. In other words, the frequency domain operation must correspond to the time domain procedure of *combining* two 4 point signals by interlacing. Consider two time domain signals, *abcd* and *efgh*. An 8 point time domain signal can be formed by two steps: dilute each 4 point signal with zeros to make it an

+ + + + + + + +
Eight Point Frequency Spectrum
Odd- Four Point

Frequency Spectrum
Even- Four Point
Frequency Spectrum
S x S x S x S x
FIGURE 12-5
FFT synthesis flow diagram. This shows the method of combining two 4 point frequency spectra into a single 8 point frequency spectrum. The ×S operation means that the signal is multiplied by a sinusoid with an appropriately selected frequency.
2 point input
2 point output
S x
FIGURE 12-6
The FFT butterfly. This is the basic calculation element in the FFT, taking two complex points and converting them into two other complex points.

8 point signal, and then add the signals together. That is, *abcd* becomes *a0b0c0d0*, and *efgh* becomes *0e0f0g0h.* Adding these two 8 point signals produces *aebfcgdh*. As shown in Fig. 12-4, diluting the time domain with zeros corresponds to a *duplication* of the frequency spectrum. Therefore, the frequency spectra are combined in the FFT by duplicating them, and then adding the duplicated spectra together.

In order to match up when added, the two time domain signals are diluted with zeros in a slightly different way. In one signal, the *odd points* are zero, while in the other signal, the *even points* are zero. In other words, one of the time domain signals (*0e0f0g0h* in Fig. 12-4) is shifted to the right by one sample. This time domain shift corresponds to multiplying the spectrum by a *sinusoid*. To see this, recall that a shift in the time domain is equivalent to convolving the signal with a shifted delta function. This multiplies the signal's spectrum with the spectrum of the shifted delta function. The spectrum of a shifted delta function is a sinusoid (see Fig 11-2).

Figure 12-5 shows a flow diagram for combining two 4 point spectra into a single 8 point spectrum. To reduce the situation even more, notice that Fig. 12-5 is formed from the basic pattern in Fig 12-6 repeated over and over.

Time Domain Data
Frequency Domain Data
Bit Reversal
Data Sorting
Overhead
Overhead
Calculation
Decomposition
Synthesis
Time
Domain
Frequency
Domain
Butterfly
FIGURE 12-7
Flow diagram of the FFT. This is based on three steps: (1) decompose an *N* point time domain signal into *N* signals each

containing a single point, (2) find the
spectrum of each of the *N* point signals
(nothing required), and (3) synthesize the
N frequency spectra into a single
frequency spectrum.
Loop for each Butterfly
Loop for Leach sub-DFT
Loop for Log_2N stages

This simple flow diagram is called a **butterfly** due to its winged appearance. The butterfly is the basic computational element of the FFT, transforming two complex points into two other complex points.

Figure 12-7 shows the structure of the entire FFT. The time domain decomposition is accomplished with a bit reversal sorting algorithm. Transforming the decomposed data into the frequency domain involves *nothing* and therefore does not appear in the figure.

The frequency domain synthesis requires three loops. The outer loop runs through the stages (i.e., each level in Fig. 12-2, starting from the bottom Log_2N and moving to the top). The middle loop moves through each of the individual frequency spectra in the stage being worked on (i.e., each of the boxes on any one level in Fig. 12-2). The innermost loop uses the butterfly to calculate the points in each frequency spectra (i.e., looping through the samples inside any one box in Fig. 12-2). The overhead boxes in Fig. 12-7 determine the beginning and ending indexes for the loops, as well as calculating the sinusoids needed in the butterflies. Now we come to the heart of this chapter, the actual FFT programs.


```
5000 'COMPLEX DFT BY CORRELATION
5010 'Upon entry, N% contains the number of points in the DFT, and
5020 'XR[ ] and XI[ ] contain the real and imaginary parts of the time domain.
5030 'Upon return, REX[ ] and IMX[ ] contain the frequency domain data.
5040 'All signals run from 0 to N%-1.
5050 '
5060 PI = 3.14159265 'Set constants
5070 '
5080 FOR K% = 0 TO N%-1 'Zero REX[ ] and IMX[ ], so they can be used
5090 REX[K%] = 0 'as accumulators during the correlation
5100 IMX[K%] = 0
5110 NEXT K%
5120 '
5130 FOR K% = 0 TO N%-1 'Loop for each value in frequency domain
5140 FOR I% = 0 TO N%-1 'Correlate with the complex sinusoid, SR & SI
5150 '
5160 SR = COS(2*PI*K%*I%/N%) 'Calculate complex sinusoid
5170 SI = -SIN(2*PI*K%*I%/N%)
5180 REX[K%] = REX[K%] + XR[I%]*SR - XI[I%]*SI
5190 IMX[K%] = IMX[K%] + XR[I%]*SI + XI[I%]*SR
5200 '
5210 NEXT I%
5220 NEXT K%
5230 '
5240 RETURN
```

TABLE 12-2

FFT Programs

As discussed in Chapter 8, the *real DFT* can be calculated by correlating the time domain signal with sine and cosine waves (see Table 8-2). Table 12-2 shows a program to calculate the *complex DFT* by the same method. In an apples-to-apples comparison, this is the program that the FFT improves upon.

Tables 12-3 and 12-4 show two different FFT programs, one in FORTRAN and one in BASIC. First we will look at the BASIC routine in Table 12-4. This subroutine produces exactly the same output as the correlation technique in Table 12-2, except it does it *much faster*. The block diagram in Fig. 12-7 can be used to identify the different sections of this program. Data are passed to this FFT subroutine in the arrays: REX[ ] and IMX[ ], each running from sample 0 to . Upon return from the subroutine, REX[ ] and IMX[ ] are *N*&1 overwritten with the frequency domain data. This is another way that the FFT is highly optimized; the same arrays are used for the input, intermediate storage, and output. This efficient use of memory is important for designing fast hardware to calculate the FFT. The term **in-place computation** is used to describe this memory usage.

While all FFT programs produce the same numerical result, there are subtle variations in programming that you need to look out for. Several of these

TABLE 12-3

The Fast Fourier Transform in FORTRAN. Data are passed to this subroutine in the variables *X*( ) and *M*. The integer, *M*, is the base two logarithm of the length of the DFT, i.e., *M* = 8 for a 256 point DFT, *M* = 12 for a 4096 point DFT, etc. The complex array, *X*( ), holds the time domain data upon entering the DFT. Upon return from this subroutine, *X*( ) is overwritten with the frequency domain data. Take note: this subroutine requires that the input and output signals run from *X*(1) through *X*(*N*), rather than the customary *X*(0) through *X*(*N*-1).

```
SUBROUTINE FFT(X,M)
COMPLEX X(4096),U,S,T
PI=3.14159265
N=2**M
DO 20 L=1,M
LE=2**(M+1-L)
LE2=LE/2
U=(1.0,0.0)
S=CMPLX(COS(PI/FLOAT(LE2)),-SIN(PI/FLOAT(LE2)))
DO 20 J=1,LE2
DO 10 I=J,N,LE
IP=I+LE2
T=X(I)+X(IP)
X(IP)=(X(I)-X(IP))*U
10 X(I)=T
20 U=U*S
ND2=N/2
NM1=N-1
J=1
DO 50 I=1,NM1
IF(I.GE.J) GO TO 30
T=X(J)
X(J)=X(I)
X(I)=T
30 K=ND2
40 IF(K.GE.J) GO TO 50
J=J-K
K=K/2
GO TO 40
50 J=J+K
RETURN
```

END

important differences are illustrated by the FORTRAN program listed in Table 12-3. This program uses an algorithm called **decimation in frequency**, while the previously described algorithm is called **decimation in time**. In a decimation in frequency algorithm, the bit reversal sorting is done *after* the three nested loops. There are also FFT routines that completely eliminate the bit reversal sorting. None of these variations significantly improve the performance of the FFT, and you shouldn't worry about which one you are using.

The *important* differences between FFT algorithms concern how data are passed to and from the subroutines. In the BASIC program, data enter and leave the subroutine in the arrays REX[ ] and IMX[ ], with the samples running from index 0 to . In the FORTRAN program, data are passed *N*&1 in the complex array , with the samples running from 1 to *N*. Since this *X*( ) is an array of complex variables, each sample in X( ) consists of two numbers, a real part and an imaginary part. The length of the DFT must also be passed to these subroutines. In the BASIC program, the variable N% is used for this purpose. In comparison, the FORTRAN program uses the variable M, which is defined to equal . For instance, M will be Log_2N

TABLE 12-4

The Fast Fourier Transform in BASIC.

```
1000 'THE FAST FOURIER TRANSFORM
1010 'Upon entry, N% contains the number of points in the DFT, REX[ ] and
1020 'IMX[ ] contain the real and imaginary parts of the input. Upon return,
1030 'REX[ ] and IMX[ ] contain the DFT output. All signals run from 0 to N%-1.
1040 '
1050 PI = 3.14159265 'Set constants
1060 NM1% = N%-1
1070 ND2% = N%/2
1080 M% = CINT(LOG(N%)/LOG(2))
1090 J% = ND2%
1100 '
1110 FOR I% = 1 TO N%-2 'Bit reversal sorting
1120 IF I% >= J% THEN GOTO 1190
1130 TR = REX[J%]
1140 TI = IMX[J%]
1150 REX[J%] = REX[I%]
1160 IMX[J%] = IMX[I%]
1170 REX[I%] = TR
1180 IMX[I%] = TI
1190 K% = ND2%
1200 IF K% > J% THEN GOTO 1240
1210 J% = J%-K%
1220 K% = K%/2
1230 GOTO 1200
1240 J% = J%+K%
1250 NEXT I%
1260 '
1270 FOR L% = 1 TO M% 'Loop for each stage
1280 LE% = CINT(2^L%)
1290 LE2% = LE%/2
1300 UR = 1
1310 UI = 0
1320 SR = COS(PI/LE2%) 'Calculate sine & cosine values
1330 SI = -SIN(PI/LE2%)
1340 FOR J% = 1 TO LE2% 'Loop for each sub DFT
1350 JM1% = J%-1
1360 FOR I% = JM1% TO NM1% STEP LE% 'Loop for each butterfly
```

```
1370 IP% = I%+LE2%
1380 TR = REX[IP%]*UR - IMX[IP%]*UI 'Butterfly calculation
1390 TI = REX[IP%]*UI + IMX[IP%]*UR
1400 REX[IP%] = REX[I%]-TR
1410 IMX[IP%] = IMX[I%]-TI
1420 REX[I%] = REX[I%]+TR
1430 IMX[I%] = IMX[I%]+TI
1440 NEXT I%
1450 TR = UR
1460 UR = TR*SR - UI*SI
1470 UI = TR*SI + UI*SR
1480 NEXT J%
1490 NEXT L%
1500 '
1510 RETURN
```



```
2000 'INVERSE FAST FOURIER TRANSFORM SUBROUTINE
2010 'Upon entry, N% contains the number of points in the IDFT, REX[ ] and
2020 'IMX[ ] contain the real and imaginary parts of the complex frequency domain.
2030 'Upon return, REX[ ] and IMX[ ] contain the complex time domain.
2040 'All signals run from 0 to N%-1.
2050 '
2060 FOR K% = 0 TO N%-1 'Change the sign of IMX[ ]
2070 IMX[K%] = -IMX[K%]
2080 NEXT K%
2090 '
2100 GOSUB 1000 'Calculate forward FFT (Table 12-3)
2110 '
2120 FOR I% = 0 TO N%-1 'Divide the time domain by N% and
2130 REX[I%] = REX[I%]/N% 'change the sign of IMX[ ]
2140 IMX[I%] = -IMX[I%]/N%
2150 NEXT I%
2160 '
2170 RETURN
```

TABLE 12-5

8 for a 256 point DFT, 12 for a 4096 point DFT, etc. The point is, the programmer who writes an FFT subroutine has many options for interfacing with the host program. Arrays that run from 1 to *N,* such as in the FORTRAN program, are especially aggravating. Most of the DSP literature (including this book) explains algorithms assuming the arrays run from sample 0 to . For instance, if the arrays run from 1 to *N*, the symmetry *N*&1 in the frequency domain is around points 1 and , rather than points *N*/2% 1 0 and , *N*/2

Using the complex DFT to calculate the real DFT has another interesting advantage. The complex DFT is more symmetrical between the time and frequency domains than the real DFT. That is, the **duality** is stronger. Among other things, this means that the Inverse DFT is nearly identical to the Forward DFT. In fact, the easiest way to calculate an *Inverse FFT* is to calculate a *Forward FFT*, and then adjust the data. Table 12-5 shows a subroutine for calculating the Inverse FFT in this manner.

Suppose you copy one of these FFT algorithms into your computer program and start it running. How do you know if it is operating properly? Two tricks are commonly used for debugging. First, start with some arbitrary time domain signal, such as from a random number generator, and run it through the FFT. Next, run the resultant frequency spectrum through the Inverse FFT and compare the result with the original signal. They should be *identical*, except round-off noise (a few parts-per-million for single precision).

The second test of proper operation is that the signals have the correct

symmetry. When the imaginary part of the time domain signal is composed of all zeros (the normal case), the frequency domain of the complex DFT will be symmetrical around samples 0 and , as previously described. *N*/2

EQUATION 12-1

DFT execution time. The time required to calculate a DFT by correlation is proportional to the length of the DFT squared.

$$ExecutionTime \approx k_{DFT} N^2$$

EQUATION 12-2

FFT execution time. The time required to calculate a DFT using the FFT is proportional to *N* multiplied by the logarithm of *N*.

$$ExecutionTime \approx k_{FFT} N \log_2 N$$

Likewise, when this correct symmetry is present in the frequency domain, the Inverse DFT will produce a time domain that has an imaginary part composes of all zeros (plus round-off noise). These debugging techniques are essential for using the FFT; become familiar with them.

Speed and Precision Comparisons

When the DFT is calculated by correlation (as in Table 12-2), the program uses two nested loops, each running through *N* points. This means that the total number of operations is proportional to *N times N.* The time to complete the program is thus given by:

where *N* is the number of points in the DFT and k_{DFT} is a constant of proportionality. If the sine and cosine values are calculated *within* the nested loops, k_{DFT} is equal to about 25 microseconds on a Pentium at 100 MHz. If you *precalculate* the sine and cosine values and store them in a look-up-table, k_{DFT} drops to about 7 microseconds. For example, a 1024 point DFT will require about 25 seconds, or nearly 25 milliseconds per point. That's slow!

Using this same strategy we can derive the execution time for the FFT. The time required for the bit reversal is negligible. In each of the stages $Log_2 N$ there are butterfly computations. This means the execution time for the *N*/2 program is approximated by:

The value of k_{FFT} is about 10 microseconds on a 100 MHz Pentium system. A 1024 point FFT requires about 70 milliseconds to execute, or 70 microseconds per point. This is more than 300 times faster than the DFT calculated by correlation!

Not only is less than , it increases much more slowly as $N\ N Log_2 N\ N^2$ becomes larger. For example, a 32 point FFT is about *ten* times faster than the correlation method. However, a 4096 point FFT is *one-thousand* times faster. For small values of *N* (say, 32 to 128), the FFT is important. For large values of *N* (1024 and above), the FFT is absolutely critical. Figure 12-8 compares the execution times of the two algorithms in a graphical form.

Number points in DFT
8 16 32 64 128 256 512 1024 2048 4096
0.001
0.01
0.1
1
10
100
1000
FFT

correlation
correlation
w/LUT
FIGURE 12-8
Execution times for calculating the DFT. The *correlation* method refers to the algorithm described in Table 12-2. This method can be made faster by precalculating the sine and cosine values and storing them in a look-up table (LUT). The FFT (Table 12-3) is the fastest algorithm when the DFT is greater than 16 points long. The times shown are for a Pentium processor at 100 MHz.
Execution time (seconds)
Number of points in DFT
16 32 64 128 256 512 1024
0
10
20
30
40
50
60
70
FFT
correlation
FIGURE 12-9
DFT precision. Since the FFT calculates the DFT *faster* than the correlation method, it also calculates it with less round-off error.
Error (parts per million)

The FFT has another advantage besides raw speed. The FFT is calculated more *precisely* because the fewer number of calculations results in less round-off error. This can be demonstrated by taking the FFT of an arbitrary signal, and then running the frequency spectrum through an Inverse FFT. This reconstructs the original time domain signal, *except* for the addition of roundoff noise from the calculations. A single number characterizing this noise can be obtained by calculating the standard deviation of the difference between the two signals. For comparison, this same procedure can be repeated using a DFT calculated by correlation, and a corresponding Inverse DFT. How does the round-off noise of the FFT compare to the DFT by correlation? See for yourself in Fig. 12-9.

Further Speed Increases

There are several techniques for making the FFT even faster; however, the improvements are only about 20-40%. In one of these methods, the time


```
4000 'INVERSE FFT FOR REAL SIGNALS
4010 'Upon entry, N% contains the number of points in the IDFT, REX[ ] and
4020 'IMX[ ] contain the real and imaginary parts of the frequency domain running from
4030 'index 0 to N%/2. The remaining samples in REX[ ] and IMX[ ] are ignored.
4040 'Upon return, REX[ ] contains the real time domain, IMX[ ] contains zeros.
4050 '
4060 '
4070 FOR K% = (N%/2+1) TO (N%-1) 'Make frequency domain symmetrical
4080 REX[K%] = REX[N%-K%] '(as in Table 12-1)
4090 IMX[K%] = -IMX[N%-K%]
4100 NEXT K%
4110 '
4120 FOR K% = 0 TO N%-1 'Add real and imaginary parts together
4130 REX[K%] = REX[K%]+IMX[K%]
4140 NEXT K%
4150 '
```

```
4160 GOSUB 3000 'Calculate forward real DFT (TABLE 12-6)
4170 '
4180 FOR I% = 0 TO N%-1 'Add real and imaginary parts together
4190 REX[I%] = (REX[I%]+IMX[I%])/N% 'and divide the time domain by N%
4200 IMX[I%] = 0
4210 NEXT I%
4220 '
4230 RETURN
```

TABLE 12-6

domain decomposition is stopped two stages early, when each signal is composed of only four points. Instead of calculating the last two stages, highly optimized code is used to jump directly into the frequency domain, using the simplicity of four point sine and cosine waves.

Another popular algorithm eliminates the wasted calculations associated with the imaginary part of the time domain being zero, and the frequency spectrum being symmetrical. In other words, the FFT is modified to calculate the *real DFT*, instead of the *complex DFT*. These algorithms are called the **real FFT** and the **real Inverse FFT** (or similar names). Expect them to be about 30% faster than the conventional FFT routines. Tables 12-6 and 12-7 show programs for these algorithms.

There are two small disadvantages in using the *real FFT*. First, the code is about twice as long. While your computer doesn't care, you must take the time to convert someone else's program to run on your computer. Second, debugging these programs is slightly harder because you cannot use symmetry as a check for proper operation. These algorithms *force* the imaginary part of the time domain to be zero, and the frequency domain to have left-right symmetry. For debugging, check that these programs produce the same output as the conventional FFT algorithms.

Figures 12-10 and 12-11 illustrate how the real FFT works. In Fig. 12-10, (a) and (b) show a time domain signal that consists of a pulse in the real part, and all zeros in the imaginary part. Figures (c) and (d) show the corresponding frequency spectrum. As previously described, the frequency domain's real part has an *even* symmetry around sample 0 and sample , while the imaginary $N/2$ part has an *odd* symmetry around these same points.

Sample number
0 16 32 48 64
-1
0
1
2
63
a. Real part
Freqeuncy
0 16 32 48
-8
-4
0
4
8
c. Real part (even symmetry)
63
Frequency
0 16 32 48
-8
-4
0
4
8
d. Imaginary part (odd symmetry)
63

Frequency Domain Time Domain

Sample number
0 16 32 48 64
-1
0

1
2
63
b. Imaginary part
FIGURE 12-10
Real part symmetry of the DFT.
Amplitude
Amplitude Amplitude
Amplitude
Now consider Fig. 12-11, where the pulse is in the imaginary part of the time domain, and the real part is all zeros. The symmetry in the frequency domain is *reversed*; the real part is odd, while the imaginary part is even. This situation will be discussed in Chapter 29. For now, take it for granted that this is how the complex DFT behaves.

What if there is a signal in *both parts* of the time domain? By additivity, the frequency domain will be the *sum* of the two frequency spectra. Now the key element: a frequency spectrum composed of these two types of symmetry can be perfectly separated into the two component signals. This is achieved by the *even/odd decomposition* discussed in Chapter 5. In other words, two real DFT's can be calculated for the price of single FFT. One of the signals is placed in the real part of the time domain, and the other signal is placed in the imaginary part. After calculating the complex DFT (via the FFT, of course), the spectra are separated using the even/odd decomposition. When two or more signals need to be passed through the FFT, this technique reduces the execution time by about 40%. The improvement isn't a full factor of two because of the calculation time required for the even/odd decomposition. This is a relatively simple technique with few pitfalls, nothing like writing an FFT routine from scratch.

Sample number
0 16 32 48 64
-1
0
1
2
63
a. Real part
Frequency
0 16 32 48
-8
-4
0
4
8
c. Real part (odd symmetry)
63
Frequency
0 16 32 48
-8
-4
0
4
8
d. Imaginary part (even symmetry)
63
Frequency Domain Time Domain
Sample number
0 16 32 48 64
-1
0
1
2
63
b. Imaginary part
FIGURE 12-11
Imaginary part symmetry of the DFT.
Amplitude
Amplitude Amplitude
Amplitude
The next step is to modify the algorithm to calculate a *single* DFT faster. It's

ugly, but here is how it is done. The input signal is broken in half by using an interlaced decomposition. The even points are placed into the real part of $N/2$ the time domain signal, while the odd points go into the imaginary part. $N/2$ An point FFT is then calculated, requiring about one-half the time as an $N/2$ N point FFT. The resulting frequency domain is then separated by the even/odd decomposition, resulting in the frequency spectra of the two interlaced time domain signals. These two frequency spectra are then combined into a single spectrum, just as in the last synthesis stage of the FFT.

To close this chapter, consider that the *FFT* is to *Digital Signal Processing* what the *transistor* is to *electronics*. It is a foundation of the technology; everyone in the field knows its characteristics and how to use it. However, only a small number of specialists really understand the details of the internal workings.


```
3000 'FFT FOR REAL SIGNALS
3010 'Upon entry, N% contains the number of points in the DFT, REX[ ] contains
3020 'the real input signal, while values in IMX[ ] are ignored. Upon return,
3030 'REX[ ] and IMX[ ] contain the DFT output. All signals run from 0 to N%-1.
3040 '
3050 NH% = N%/2-1 'Separate even and odd points
3060 FOR I% = 0 TO NH%
3070 REX(I%) = REX(2*I%)
3080 IMX(I%) = REX(2*I%+1)
3090 NEXT I%
3100 '
3110 N% = N%/2 'Calculate N%/2 point FFT
3120 GOSUB 1000 '(GOSUB 1000 is the FFT in Table 12-3)
3130 N% = N%*2
3140 '
3150 NM1% = N%-1 'Even/odd frequency domain decomposition
3160 ND2% = N%/2
3170 N4% = N%/4-1
3180 FOR I% = 1 TO N4%
3190 IM% = ND2%-I%
3200 IP2% = I%+ND2%
3210 IPM% = IM%+ND2%
3220 REX(IP2%) = (IMX(I%) + IMX(IM%))/2
3230 REX(IPM%) = REX(IP2%)
3240 IMX(IP2%) = -(REX(I%) - REX(IM%))/2
3250 IMX(IPM%) = -IMX(IP2%)
3260 REX(I%) = (REX(I%) + REX(IM%))/2
3270 REX(IM%) = REX(I%)
3280 IMX(I%) = (IMX(I%) - IMX(IM%))/2
3290 IMX(IM%) = -IMX(I%)
3300 NEXT I%
3310 REX(N%*3/4) = IMX(N%/4)
3320 REX(ND2%) = IMX(0)
3330 IMX(N%*3/4) = 0
3340 IMX(ND2%) = 0
3350 IMX(N%/4) = 0
3360 IMX(0) = 0
3370 '
3380 PI = 3.14159265 'Complete the last FFT stage
3390 L% = CINT(LOG(N%)/LOG(2))
3400 LE% = CINT(2^L%)
3410 LE2% = LE%/2
3420 UR = 1
3430 UI = 0
3440 SR = COS(PI/LE2%)
3450 SI = -SIN(PI/LE2%)
3460 FOR J% = 1 TO LE2%
3470 JM1% = J%-1
3480 FOR I% = JM1% TO NM1% STEP LE%
```

```
3490 IP% = I%+LE2%
3500 TR = REX[IP%]*UR - IMX[IP%]*UI
3510 TI = REX[IP%]*UI + IMX[IP%]*UR
3520 REX[IP%] = REX[I%]-TR
3530 IMX[IP%] = IMX[I%]-TI
3540 REX[I%] = REX[I%]+TR
3550 IMX[I%] = IMX[I%]+TI
3560 NEXT I%
3570 TR = UR
3580 UR = TR*SR - UI*SI
3590 UI = TR*SI + UI*SR
3600 NEXT J%
3610 RETURN TABLE 12-7
```


CHAPTER

13 Continuous Signal Processing

Continuous signal processing is a parallel field to DSP, and most of the techniques are nearly identical. For example, both DSP and continuous signal processing are based on linearity, decomposition, convolution and Fourier analysis. Since continuous signals cannot be directly represented in digital computers, don't expect to find computer programs in this chapter. Continuous signal processing is based on *mathematics*; signals are represented as equations, and systems change one equation into another. Just as the *digital computer* is the primary tool used in DSP, *calculus* is the primary tool used in continuous signal processing. These techniques have been used for centuries, long before computers were developed.

The Delta Function

Continuous signals can be decomposed into scaled and shifted *delta functions*, just as done with discrete signals. The difference is that the continuous delta function is much more complicated and mathematically abstract than its discrete counterpart. Instead of defining the continuous delta function by what it *is*, we will define it by the *characteristics it has*.

A thought experiment will show how this works. Imagine an electronic circuit composed of linear components, such as resistors, capacitors and inductors. Connected to the input is a signal generator that produces various shapes of short *pulses*. The output of the circuit is connected to an oscilloscope, displaying the waveform produced by the circuit in response to each input pulse. The question we want to answer is: *how is the shape of the output pulse related to the characteristics of the input pulse*? To simplify the investigation, we will only use input pulses that are much shorter than the output. For instance, if the system responds in milliseconds, we might use input pulses only a few microseconds in length.

After taking many measurement, we come to three conclusions: First, the *shape* of the input pulse does not affect the shape of the output signal. This

is illustrated in Fig. 13-1, where various shapes of short input pulses produce exactly the same shape of output pulse. Second, the shape of the output waveform is totally determined by the characteristics of the system, i.e., the value and configuration of the resistors, capacitors and inductors. Third, the *amplitude* of the output pulse is directly proportional to the *area* of the input pulse. For example, the output will have the same amplitude for inputs of: 1 volt for 1 microsecond, 10 volts for 0.1 microseconds, 1,000 volts for 1 nanosecond, etc. This relationship also allows for input pulses with *negative* areas. For instance, imagine the combination of a 2 volt pulse lasting 2 microseconds being quickly followed by a -1 volt pulse

lasting 4 microseconds. The total area of the input signal is *zero*, resulting in the output doing *nothing*.

Input signals that are brief enough to have these three properties are called **impulses**. In other words, an impulse is any signal that is entirely zero except for a short *blip* of arbitrary shape. For example, an impulse to a microwave transmitter may have to be in the *picosecond* range because the electronics responds in *nanoseconds*. In comparison, a volcano that erupts for *years* may be a perfectly good impulse to geological changes that take *millennia*.

Mathematicians don't like to be limited by any particular system, and commonly use the term *impulse* to mean a signal that is short enough to be an impulse to *any possible* system. That is, a signal that is *infinitesimally* narrow. The **continuous delta function** is a normalized version of this type of impulse. Specifically, the continuous delta function is mathematically defined by three idealized characteristics: (1) the signal must be infinitesimally brief, (2) the pulse must occur at time zero, and (3) the pulse must have an area of one.

Since the delta function is defined to be infinitesimally narrow *and* have a fixed area, the amplitude is implied to be *infinite*. Don't let this bother you; it is completely unimportant. Since the amplitude is part of the *shape* of the impulse, you will never encounter a problem where the amplitude makes any difference, infinite or not. The delta function is a mathematical construct, not a real world signal. Signals in the real world that *act* as delta functions will always have a finite duration and amplitude.

Just as in the discrete case, the continuous delta function is given the mathematical symbol: . Likewise, the output of a continuous system in *( ) response to a delta function is called the **impulse response**, and is often denoted by: . Notice that parentheses, ( ), are used to denote continuous *h*( ) signals, as compared to brackets, [ ], for discrete signals. This notation is used in this book and elsewhere in DSP, but isn't universal. Impulses are displayed in graphs as vertical arrows (see Fig. 13-1d), with the *length* of the arrow indicating the *area* of the impulse.

To better understand real world impulses, look into the night sky at a *planet* and a *star*, for instance, Mars and Sirius. Both appear about the same brightness and size to the unaided eye. The reason for this similarity is not

*

Linear
System
Linear
System
Linear
System
Linear
System

a.

b.

c

d.

(t)

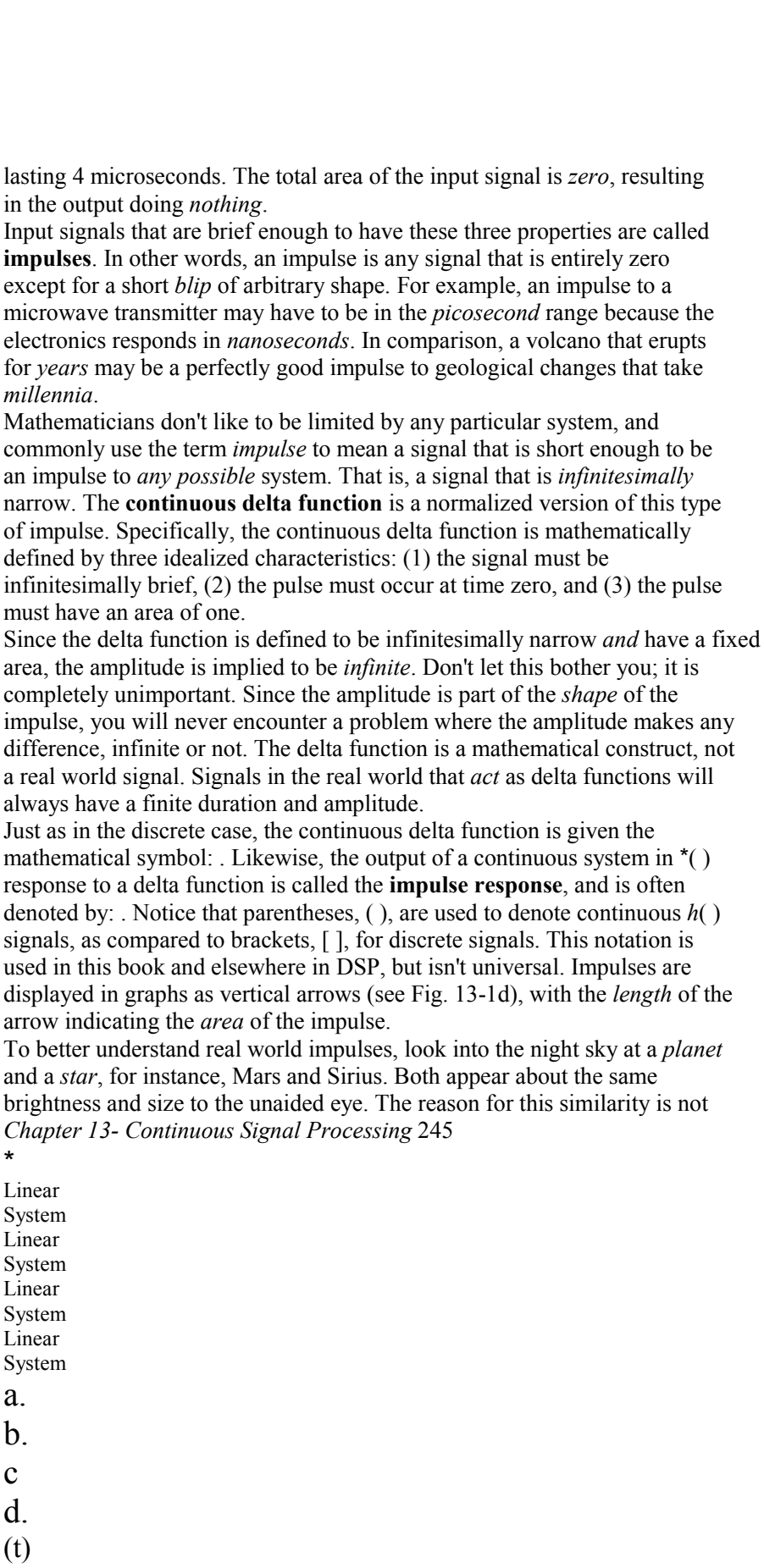


FIGURE 13-1
The continuous delta function. If the input to a linear system is brief compared to the resulting

output, the shape of the output depends only on the characteristics of the system, and not the shape of the input. Such short input signals are called *impulses*. Figures a,b & c illustrate example input signals that are impulses for this particular system. The term *delta function* is used to describe a normalized impulse, i.e., one that occurs at and has an area of one. The mathematical symbols *t* ' 0 for the delta function are shown in (d), a vertical arrow and . *(*t*)

obvious, since the viewing geometry is drastically different. Mars is about 6000 kilometers in diameter and 60 million kilometers from earth. In comparison, Sirius is about 300 times larger and over one-million times farther away. These dimensions should make Mars appear more than *three-thousand* times larger than Sirius. How is it possible that they look alike?

These objects look the same because they are small enough to be *impulses* to the human visual system. The perceived shape is the impulse response of the eye, not the actual image of the star or planet. This becomes obvious when the two objects are viewed through a small telescope; Mars appears as a dim disk, while Sirius still appears as a bright impulse. This is also the reason that stars twinkle while planets do not. The image of a star is small enough that it can be briefly blocked by particles or turbulence in the atmosphere, whereas the larger image of the planet is much less affected.

$$y(t) = \int_{-\infty}^{+\infty} x(\tau)\, h(t-\tau)\, d\tau$$

EQUATION 13-1
The convolution integral. This equation defines the meaning of: . *y* (*t*) ' *x* (*t*)t*h*(*t*)

Convolution

Just as with discrete signals, the convolution of continuous signals can be viewed from the *input signal*, or the *output signal*. The input side viewpoint is the best *conceptual* description of how convolution operates. In comparison, the output side viewpoint describes the *mathematics* that must be used. These descriptions are virtually identical to those presented in Chapter 6 for discrete signals.

Figure 13-2 shows how convolution is viewed from the input side. An input signal, , is passed through a system characterized by an impulse response, *x* (*t*) , to produce an output signal, . This can be written in the familiar *h*(*t*) *y* (*t*) mathematical equation, . The input signal is divided into *y* (*t*) ' *x*(*t*)t*h* (*t*) narrow columns, each short enough to act as an *impulse* to the system. In other words, the input signal is decomposed into an infinite number of scaled and shifted delta functions. Each of these impulses produces a scaled and shifted version of the impulse response in the output signal. The final output signal is then equal to the combined effect, i.e., the sum of all of the individual responses.

For this scheme to work, the width of the columns must be much shorter than the response of the system. Of course, mathematicians take this to the extreme by making the input segments *infinitesimally* narrow, turning the situation into a calculus problem. In this manner, the input viewpoint describes how a single point (or narrow region) in the input signal affects a larger portion of output signal.

In comparison, the output viewpoint examines how a single point in the output signal is determined by the various values from the input signal. Just as with discrete signals, each instantaneous value in the output signal is affected by a section of the input signal, weighted by the impulse response flipped

left-for-right. In the discrete case, the signals are multiplied and *summed*. In the continuous case, the signals are multiplied and *integrated*. In equation form:

This equation is called the convolution integral, and is the twin of the convolution sum (Eq. 6-1) used with discrete signals. Figure 13-3 shows how this equation can be understood. The goal is to find an expression for calculating the value of the output signal at an arbitrary time, *t*. The first step is to change the independent variable used to move through the input signal and the impulse response. That is, we replace *t* with J (a lower case

a
b
c
a
b
c
time (t) time(t)
x(t) y(t) Linear
System

FIGURE 13-2
Convolution viewed from the input side. The input signal, , is divided into narrow segments, $x(t)$ each acting as an impulse to the system. The output signal, , is the sum of the resulting scaled $y(t)$ and shifted impulse responses. This illustration shows how three points in the input signal contribute to the output signal.

time (J) time(t)
x(J) y(t)
t t h(t-J)
?
x(J)
Linear
System

FIGURE 13-3
Convolution viewed from the output side. Each value in the output signal is influenced by many points from the input signal. In this figure, the output signal at time *t* is being calculated. The input signal, , is *weighted* (multiplied) by the flipped and shifted impulse response, given by . $x(J)\,h(t\&J)$ Integrating the weighted input signal produces the value of the output point, $y(t)$

Greek tau). This makes and become and , respectively. $x(t)\ h(t)\ x(J)\ h(J)$ This change of variable names is needed because *t* is already being used to represent the point in the output signal being calculated. The next step is to flip the impulse response left-for-right, turning it into . Shifting the $h(\&J)$ flipped impulse response to the location *t*, results in the expression becoming . The input signal is then weighted by the flipped and shifted impulse $h(t\&J)$ response by multiplying the two, i.e., . The value of the output $x(J)\ h(t\&J)$ signal is then found by integrating this weighted input signal from negative to positive infinity, as described by Eq. 13-1.

If you have trouble understanding how this works, go back and review the same concepts for discrete signals in Chapter 6. Figure 13-3 is just another way of describing the convolution machine in Fig. 6-8. The only difference is that integrals are being used instead of summations. Treat this as an extension of what you already know, not something new.

An example will illustrate how continuous convolution is used in real world problems and the mathematics required. Figure 13-4 shows a simple continuous linear system: an electronic low-pass filter composed of a single resistor and a single capacitor. As shown in the figure, an impulse entering this system produces an output that quickly jumps to some value, and then

exponentially decays toward zero. In other words, the impulse response of this simple electronic circuit is a *one-sided exponential*. Mathematically, the

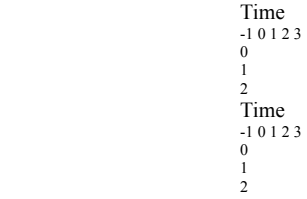


FIGURE 13-4
Example of a continuous linear system. This electronic circuit is a low-pass filter composed of a single resistor and capacitor. The impulse response of this system is a one-sided exponential.

$h(t) = 0$ for $t < 0$

$h(t) = e^{-t}$ for $t \geq 0$

$x(t) = 1$ for $0 \leq t \leq 1$

$x(t) = 0$ otherwise

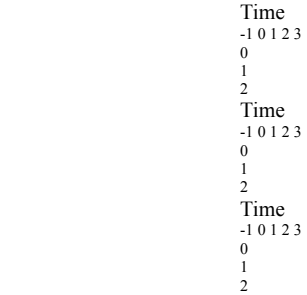


FIGURE 13-5
Example of continuous convolution. This figure illustrates a square pulse entering an RC low-pass filter (Fig. 13-4). The square pulse is convolved with the system's impulse response to produce the output.

impulse response of this system is broken into two sections, each represented by an equation:

where (R is in ohms, C is in farads, and t is in seconds). Just as in $\tau = 1/RC$ the discrete case, the continuous impulse response contains complete information about the system, that is, how it will react to all possible signals.

To pursue this example further, Fig. 13-5 shows a square pulse entering the system, mathematically expressed by:

Since both the input signal and the impulse response are completely known as mathematical expressions, the output signal, , can be calculated by $y(t)$ evaluating the convolution integral of Eq. 13-1. This is complicated by the fact that both signals are defined by *regions* rather than a single

t t t
(t > 1) (0 # t # 1) (t < 0)
FIGURE 13-6
Calculating a convolution by segments. Since many continuous signals are defined by *regions*, the convolution calculation must be performed region-by-region. In this example, calculation of the output signal is broken into three sections: (a) no overlap, (b) partial overlap, and (c) total overlap, of the input signal and the shiftedflipped impulse response.

$y(t) = 0$ for t < 0

$$y(t) = m_t$$
0
$$1@ "e \&"(t\& J) dJ$$
$$y(t) = e \&"t [e "J] /$$
t
0
$$y(t) = e \&"t [e "t \& 1]$$
$$y(t) = 1 \& e \& "t$$
$$y(t) = m$$
4
&4
$x(J)\, h(t\& J)\, dJ$ (start with Eq. 13-1)
(plug in the signals)
for 0 # t # 1
(evaluate the integral)
(reduce)

mathematical expression. This is very common in continuous signal processing. It is usually essential to draw a picture of how the two signals shift over each other for various values of *t*. In this example, Fig. 13-6a shows that the two signals do not overlap at all for . This means that $t < 0$ the product of the two signals is zero at all locations along the J axis, and the resulting output signal is:

A second case is illustrated in (b), where *t* is between 0 and 1. Here the two signals partially overlap, resulting in their product having nonzero values between and . Since this is the only nonzero region, it is the only J' 0 J' *t* section where the integral needs to be evaluated. This provides the output signal for , given by: 0 # *t* #1

$$y(t) = m$$
1
0
$$1@ "e \&"(t\& J) dJ$$
$$y(t) = e \&"t [e "J] /$$
1
0
$$y(t) = [e "\& 1] e \&"t$$
(plug into Eq. 13-1)
for t > 1
(evaluate the integral)

Figure (c) shows the calculation for the third section of the output signal, where t > 1. Here the overlap occurs between and , making the J' 0 J' 1 calculation the same as for the second segment, except a change to the limits of integration:

The waveform in each of these three segments should agree with your knowledge of electronics: (1) The output signal must be zero until the input signal becomes nonzero. That is, the first segment is given by for $y(t) = 0$. (2) When the step occurs, the RC circuit exponentially increases to match $t < 0$

the input, according to the equation: . (3) When the input is $y(t) = 1 - e^{-t}$ returned to zero, the output exponentially decays toward zero, given by the equation: (where , the voltage on the capacitor just $y(t) = ke^{-t}k = e^{-1}$ before the discharge was started).

More intricate waveforms can be handled in the same way, although the mathematical complexity can rapidly become unmanageable. When faced with a nasty continuous convolution problem, you need to spend significant time evaluating *strategies* for solving the problem. If you start blindly evaluating integrals you are likely to end up with a mathematical mess. A common strategy is to break one of the signals into simpler additive components that can be *individually* convolved. Using the principles of linearity, the resulting waveforms can be added to find the answer to the original problem.

Figure 13-7 shows another strategy: modify one of the signals in some linear way, perform the convolution, and then undo the original modification. In this example the modification is the *derivative*, and it is undone by taking the *integral*. The derivative of a unit amplitude square pulse is two *impulses*, the first with an area of one, and the second with an area of negative one. To understand this, think about the opposite process of taking the integral of the two impulses. As you integrate past the first impulse, the integral rapidly increases from zero to one, i.e., a step function. After passing the negative impulse, the integral of the signal rapidly returns from one back to zero, completing the square pulse.

Taking the derivative simplifies this problem because convolution is easy when one of the signals is composed of impulses. Each of the two impulses in contributes a scaled and shifted version of the impulse response to $x_)(t)$

Time
-1 0 1 2 3
-2
-1
0
1
2
Time
-1 0 1 2 3
-2
-1
0
1
2
h(t) yN(t)
Time
-1 0 1 2 3
0
1
2
Time
-1 0 1 2 3
0
1
2
Time
-1 0 1 2 3
0
1
2
x(t) h(t) y(t)
I d/dt
Time
-1 0 1 2 3
-2
-1
0
1
2

xN(t)
Amplitude
Amplitude
Amplitude
FIGURE 13-7
A strategy for convolving signals. Convolution problems can often be simplified by clever use of the rules governing linear systems. In this example, the convolution of two signals is simplified by taking the derivative of one of them. After performing the convolution, the derivative is undone by taking the integral.
Amplitude
Amplitude
Amplitude
the derivative of the output signal, . That is, by inspection it is known $y_)(t)$ that: . The output signal, , can then be found by $y_)(t) ' h(t) & h(t\&1) y(t)$ plugging in the exact equation for , and integrating the expression. $h(t)$
A slight nuisance in this procedure is that the DC value of the input signal is lost when the derivative is taken. This can result in an error in the DC value of the calculated output signal. The mathematics reflects this as the arbitrary constant that can be added during the integration. There is no systematic way of identifying this error, but it can usually be corrected by inspection of the problem. For instance, there is no DC error in the example of Fig. 13-7. This is known because the calculated output signal has the correct DC value when t becomes very large. If an error is present in a particular problem, an appropriate DC term is manually added to the output signal to complete the calculation.
This method also works for signals that can be reduced to impulses by taking the derivative *multiple* times. In the jargon of the field, these signals are called *piecewise polynomials*. After the convolution, the initial operation of multiple derivatives is undone by taking multiple integrals. The only catch is that the lost DC value must be found at each stage by finding the correct constant of integration.

$x(t) ' 1$
B m
%4
0
$Re\ X(\mathsf{T}) \cos(\mathsf{T}t) \& Im\ X(\mathsf{T}) \sin(\mathsf{T}t)\ d\mathsf{T}$
EQUATION 13-2
The Fourier transform synthesis equation. In this equation, is the time $x(t)$ domain signal being synthesized, and & are the real and $ReX(\mathsf{T})$ $Im\ X(\mathsf{T})$ imaginary parts of the frequency spectrum, respectively.
Before starting a difficult continuous convolution problem, there is another approach that you should consider. Ask yourself the question: *Is a mathematical expression really needed for the output signal, or is a graph of the waveform sufficient?* If a graph is adequate, you may be better off to handle the problem with *discrete* techniques. That is, approximate the continuous signals by samples that can be directly convolved by a computer program. While not as mathematically pure, it can be much easier.

The Fourier Transform

The Fourier Transform for continuous signals is divided into two categories, one for signals that are *periodic*, and one for signals that are *aperiodic*. Periodic signals use a version of the Fourier Transform called the **Fourier Series**, and are discussed in the next section. The Fourier Transform used with aperiodic signals is simply called the **Fourier Transform**. This chapter describes these Fourier techniques using only *real* mathematics, just as the last several chapters have done for discrete signals. The more powerful use of *complex* mathematics will be reserved for Chapter 31.

Figure 13-8 shows an example of a continuous aperiodic signal and its frequency spectrum. The time domain signal extends from negative infinity to positive infinity, while each of the frequency domain signals extends from zero to positive infinity. This frequency spectrum is shown in rectangular form (real and imaginary parts); however, the polar form (magnitude and phase) is also used with continuous signals. Just as in the discrete case, the **synthesis equation** describes a recipe for constructing the time domain signal using the data in the frequency domain. In mathematical form:

In words, the time domain signal is formed by adding (with the use of an integral) an infinite number of scaled sine and cosine waves. The real part of the frequency domain consists of the scaling factors for the cosine waves, while the imaginary part consists of the scaling factors for the sine waves. Just as with discrete signals, the synthesis equation is usually written with *negative* sine waves. Although the negative sign has no significance in this discussion, it is necessary to make the notation compatible with the complex mathematics described in Chapter 29. The key point to remember is that some authors put this negative sign in the equation, while others do not.

Also notice that frequency is represented by the symbol, T, a lower case

FIGURE 13-8

Example of the Fourier Transform. The time domain signal, , extends from negative to positive infinity. $x(t)$ The frequency domain is composed of a real part, , and an imaginary part, , each extending from $ReX(\mathsf{T})$ $Im\ X(\mathsf{T})$ zero to positive infinity. The frequency axis in this illustration is labeled in *cycles per second* (hertz). To convert to natural frequency, multiply the numbers on the frequency axis by 2B.

Frequency (hertz)
0 20 40 60 80 100 120 140
0
20
40
60
80
100
b. *Re X*(T)
Frequency (hertz)
0 20 40 60 80 100 120 140
0
20
40
60
80
100
c. *Im X*(T)
Time (milliseconds)
-50 -40 -30 -20 -10 0 10 20 30 40 50
-8
-4
0
4
8
a. *x*(*t*)

Time Domain Frequency Domain

Amplitude Amplitude
Amplitude

$ReX(\mathsf{T})$ ’ m
%4
&4

$x(t)\cos(\mathsf{T}t)\,dt$ EQUATION 13-3

The Fourier transform analysis equations. In this equation, & are the real $ReX(\mathsf{T})$ $Im\ X(\mathsf{T})$ and imaginary parts of the frequency spectrum, respectively, and is the time $x(t)$ domain signal being analyzed. $ImX(\mathsf{T})$ ’ & m
%4

&4
$x(t)\sin(\Omega t)\,dt$
Greek omega. As you recall, this notation is called the **natural frequency**, and has the units of radians per second. That is, , where f is the $\Omega' 2\pi f$ frequency in cycles per second (hertz). The natural frequency notation is favored by mathematicians and others doing signal processing by *solving equations*, because there are usually fewer symbols to write.
The **analysis equations** for continuous signals follow the same strategy as the discrete case: *correlation* with sine and cosine waves. The equations are:

$h(t) ' 0$
$h(t) ' \alpha e^{-\alpha t}$ for t $ 0
for t < 0
$Re\,H(\Omega) ' \alpha^2$
$\alpha^2 \% \Omega^2$
$Re\,H(\Omega) ' \alpha e^{-\alpha t}$
$\alpha^2 \% \Omega^2$
$[-\alpha \cos(\Omega t) \% \Omega \sin(\Omega t)] /$
%4
0
$Re\,H(\Omega) ' m$
%4
0
$\alpha e^{-\alpha t}\cos(\Omega t)\,dt$
$Re\,H(\Omega) ' m$
%4
&4
$h(t)\cos(\Omega t)\,dt$ (start with Eq. 13-3)
(plug in the signal)
(evaluate)
$Im\,H(\Omega) '$
$-\Omega\alpha$
$\alpha^2 \% \Omega^2$
As an example of using the analysis equations, we will find the frequency response of the RC low-pass filter. This is done by taking the Fourier transform of its impulse response, previously shown in Fig. 13-4, and described by:
The frequency response is found by plugging the impulse response into the analysis equations. First, the real part:
Using this same approach, the imaginary part of the frequency response is calculated to be:
Just as with discrete signals, the rectangular representation of the frequency domain is great for mathematical manipulation, but difficult for human understanding. The situation can be remedied by converting into polar notation with the standard relations: $Mag\,H(\Omega) ' [Re\,H(\Omega)^2 \% Im\,H(\Omega)^2]^{1/2}$ and . Working through the algebra $Phase\,H(\Omega) ' \arctan [Re\,H(\Omega) / Im\,H(\Omega)]$

$Mag\,H(\Omega) ' \alpha$
$[\alpha^2 \% \Omega^2]^{1/2}$
$Phase\,H(\Omega) ' \arctan -\Omega$
α
Frequency (hertz)
0 1000 2000 3000 4000 5000 6000

-1.6
-1.2
-0.8
-0.4
0.0
0.4
0.8
1.2
1.6
b. Phase
FIGURE 13-9
Frequency response of an RC low-pass filter. These curves were derived by calculating the Fourier transform of the impulse response, and then converting to polar form.
Frequency (hertz)
0 1000 2000 3000 4000 5000 6000
0.0
0.2
0.4
0.6
0.8
1.0
1.2
a. Magnitude
Phase (radians)
Amplitude
provides the frequency response of the RC low-pass filter as magnitude and phase (i.e., polar form):
Figure 13-9 shows graphs of these curves for a cutoff frequency of 1000 hertz (i.e.,). "' 2B1000

The Fourier Series

This brings us to the last member of the Fourier transform family: the *Fourier series*. The time domain signal used in the Fourier series is *periodic* and *continuous*. Figure 13-10 shows several examples of continuous waveforms that repeat themselves from negative to positive infinity. Chapter 11 showed that periodic signals have a frequency spectrum consisting of **harmonics**. For instance, if the time domain repeats at 1000 hertz (a period of 1 millisecond), the frequency spectrum will contain a first harmonic at 1000 hertz, a second harmonic at 2000 hertz, a third harmonic at 3000 hertz, and so forth. The first harmonic, i.e., the frequency that the time domain repeats itself, is also called the **fundamental frequency.** This means that the frequency spectrum can be viewed in two ways: (1) the frequency spectrum is *continuous*, but zero at all frequencies except the harmonics, or (2) the frequency spectrum is *discrete*, and only *defined* at the harmonic frequencies. In other words, the frequencies between the harmonics can be thought of as having a value of zero, or simply

$$x(t) = a_0 + \sum_{n=1}^{\infty} a_n \cos(2\pi f t\, n) + \sum_{n=1}^{\infty} b_n \sin(2\pi f t\, n)$$

EQUATION 13-4
The Fourier series synthesis equation. Any periodic signal, , can $x(t)$ be reconstructed from sine and cosine waves with frequencies that are multiples of the fundamental, f. The a_n and b_n coefficients hold the amplitudes of the cosine and sine waves, respectively.

$$a_0 = \frac{1}{T} \int_{-T/2}^{T/2}$$

$$x(t)\,dt\,a_n$$
$$\frac{2}{T}\int_{-T/2}^{T/2} x(t)\cos\frac{2\pi tn}{T}\,dt$$
$$b_n$$
$$\frac{2}{T}\int_{-T/2}^{T/2} x(t)\sin\frac{2\pi tn}{T}\,dt$$

EQUATION 13-5

Fourier series analysis equations. In these equations, is $x(t)$ the time domain signal being decomposed, is the DC a_0 component, & hold the amplitudes of the cosine and a_n b_n sine waves, respectively, and T is the period of the signal, i.e., the reciprocal of the fundamental frequency.

not existing. The important point is that they do not contribute to forming the time domain signal.

The Fourier series **synthesis equation** creates a continuous periodic signal with a fundamental frequency, *f*, by adding scaled cosine and sine waves with frequencies: *f*, 2*f*, 3*f*, 4*f*, etc. The amplitudes of the cosine waves are held in the variables: etc., while the amplitudes of the sine a_1, a_2, a_3, a_4, waves are held in: and so on. In other words, the *"a"* and "*b*" b_1, b_2, b_3, b_4, coefficients are the real and imaginary parts of the frequency spectrum, respectively. In addition, the coefficient is used to hold the DC value of a_0 the time domain waveform. This can be viewed as the amplitude of a cosine wave with zero frequency (a constant value). Sometimes is grouped with a_0 the other "*a*" coefficients, but it is often handled separately because it requires special calculations. There is no coefficient since a sine wave b_0 of zero frequency has a constant value of zero, and would be quite useless. The synthesis equation is written:

The corresponding **analysis equations** for the Fourier series are usually written in terms of the *period* of the waveform, denoted by *T*, rather than the fundamental frequency, *f* (where). Since the time domain signal is $f = 1/T$ periodic, the sine and cosine wave correlation only needs to be evaluated over a single period, i.e., to , 0 to *T*, -*T* to 0, etc. Selecting different $-T/2$ $T/2$ limits makes the mathematics different, but the final answer is always the same. The Fourier series analysis equations are:

0 f 2f 3f 4f 5f 6f
A
0
0 f 2f 3f 4f 5f 6f
A
0

0 f 2f 3f 4f 5f 6f
A
0
0 f 2f 3f 4f 5f 6f
A
0
0 f 2f 3f 4f 5f 6f
A
0
0 f 2f 3f 4f 5f 6f
A
0
A

a. Pulse

b. Square

c. Triangle

d. Sawtooth

e. Rectified

Time Domain Frequency Domain

A
A
A
A
A
k
T
d = k/T
t = 0
t = 0
t = 0
t = 0
t = 0
t = 0

f. Cosine wave

(all even harmonics are zero)

a_0

' 0

a_n

' $2A$

nB

sin nB

2

b_n

' 0

(in this example) d ' 0.27

a_0

' $A\ d$

b_n

' 0

a_n

' $2A$

nB

sin (nBd)

(all even harmonics are zero)

a_0

' 0

a_n

' $4A$

(nB)^2

b_n

’ 0
a_0
’ 0
a_n
’ 0
b_n
’ A
nB
a_0
’ 2A/B
b_n
’ 0
a_n
’
&4A
B(4n 2&1)
(all other coefficients are zero)
a_1
’ A
FIGURE 13-10
Examples of the Fourier series. Six common time domain waveforms are shown, along with the equations to calculate their "a" and "b" coefficients.

-T 0 T 2T 3T -2T -3T
A
0
-T/2 T/2
-k/2 k/2
Time
FIGURE 13-11
Example of calculating a Fourier series. This is a pulse train with a duty cycle of $d = k/T$. The Fourier series coefficients are calculated by correlating the waveform with cosine and sine waves over any full period. In this example, the period from -T/2 to T/2 is used.
Amplitude
$x(t)$ ’ 0
$x(t)$ ’ A for -k/2 # t # k/2
otherwise
a_0
’ 1
T m
$T/2$
&$T/2$
$x(t)\,dt$ (start with Eq. 13-5)
a_0
’ 1
T m
$k/2$
&$k/2$
$A\,dt$
a_0
’ $A\,k$
T
a_0
’ $A\,d$
(plug in the signal)
(evaluate the integral)
(substitute: $d = k/T$)

Figure 13-11 shows an example of calculating a Fourier series using these equations. The time domain signal being analyzed is a *pulse train*, a square wave with unequal high and low durations. Over a single period from &*T*/2 to , the waveform is given by: *T*/2

The *duty cycle* of the waveform (the fraction of time that the pulse is "high") is thus given by . The Fourier series coefficients can be found by *d* ' *k*/*T* evaluating Eq. 13-5. First, we will find the DC component, : a_0

This result should make intuitive sense; the DC component is simply the average value of the signal. A similar analysis provides the "*a*" coefficients:

$$a_n = \frac{2}{T} \int_{-T/2}^{T/2} x(t) \cos\left(\frac{2\pi t n}{T}\right) dt$$

$$a_n = \frac{2}{T} \int_{-k/2}^{k/2} A \cos\left(\frac{2\pi t n}{T}\right) dt$$

$$a_n = \frac{2A}{T} \left[\frac{T}{2\pi n} \sin\left(\frac{2\pi t n}{T}\right)\right]_{-k/2}^{k/2}$$

$$a_n = \frac{2A}{n\pi} \sin(\pi n d)$$

(start with Eq. 13-4)
(plug in the signal)
(evaluate the integral)
(reduce)

The "*b*" coefficients are calculated in this same way; however, they all turn out to be *zero*. In other words, this waveform can be constructed using only cosine waves, with no sine waves being needed.

The "*a*" and "*b*" coefficients will change if the time domain waveform is shifted left or right. For instance, the "*b*" coefficients in this example will be zero *only* if one of the pulses is centered on . Think about it this way. *t* ' 0

If the waveform is *even* (i.e., symmetrical around), it will be composed *t* ' 0

solely of *even* sinusoids, that is, cosine waves. This makes all of the "*b*" coefficients equal to zero. If the waveform if *odd* (i.e., symmetrical but opposite in sign around), it will be composed of *odd* sinusoids, i.e., sine *t* ' 0 waves. This results in the "*a*" coefficients being zero. If the coefficients are converted to polar notation (say, M_n and 2_n coefficients), a shift in the time domain leaves the magnitude unchanged, but adds a linear component to the phase.

To complete this example, imagine a pulse train existing in an electronic circuit, with a frequency of 1 kHz, an amplitude of one volt, and a duty cycle of 0.2. The table in Fig. 13-12 provides the amplitude of each harmonic contained in this waveform. Figure 13-12 also shows the synthesis of the waveform using only the *first fourteen* of these harmonics. Even with this number of harmonics, the reconstruction is not very good. In mathematical jargon, the Fourier series *converges* very *slowly*. This is just another way of saying that sharp edges in the time domain waveform results in very high frequencies in the spectrum. Lastly, be sure and notice the overshoot at the sharp edges, i.e., the Gibbs effect discussed in Chapter 11.

An important application of the Fourier series is electronic **frequency multiplication**. Suppose you want to construct a very stable sine wave oscillator at 150 MHz. This might be needed, for example, in a radio

Time (milliseconds)
0 1 2 3 4
-0.5
0.0
0.5
1.0
1.5 frequency amplitude
(volts)
DC 0.20000
1 kHz 0.37420
2 kHz 0.30273
3 kHz 0.20182
4 kHz 0.09355
5 kHz 0.00000
6 kHz -0.06237
7 kHz -0.08649
8 kHz -0.07568
9 kHz -0.04158
10 kHz 0.00000
11 kHz 0.03402
12 kHz 0.05046
!
123 kHz 0.00492
124 kHz 0.00302
125 kHz 0.00000
126 kHz -0.00297
!
803 kHz 0.00075
804 kHz 0.00046
805 kHz 0.00000
806 kHz -0.00046

FIGURE 13-12
Example of Fourier series synthesis. The waveform being constructed is a pulse train at 1 kHz, an amplitude of one volt, and a duty cycle of 0.2 (as illustrated in Fig. 13-11). This table shows the amplitude of the harmonics, while the graph shows the reconstructed waveform using only the first fourteen harmonics.

Amplitude (volts)

transmitter operating at this frequency. High stability calls for the circuit to

be *crystal controlled*. That is, the frequency of the oscillator is determined by a resonating quartz crystal that is a part of the circuit. The problem is, quartz crystals only work to about 10 MHz. The solution is to build a crystal controlled oscillator operating somewhere between 1 and 10 MHz, and then *multiply* the frequency to whatever you need. This is accomplished by *distorting* the sine wave, such as by clipping the peaks with a diode, or running the waveform through a squaring circuit. The harmonics in the distorted waveform are then isolated with band-pass filters. This allows the frequency to be doubled, tripled, or multiplied by even higher integers numbers. The most common technique is to use sequential stages of doublers and triplers to generate the required frequency multiplication, rather than just a single stage. The Fourier series is important to this type of design because it describes the *amplitude* of the multiplied signal, depending on the type of distortion and harmonic selected.

261

CHAPTER

14 Introduction to Digital Filters

Digital filters are used for two general purposes: (1) separation of signals that have been combined, and (2) restoration of signals that have been distorted in some way. Analog (electronic) filters can be used for these same tasks; however, digital filters can achieve far superior results. The most popular digital filters are described and compared in the next seven chapters. This introductory chapter describes the parameters you want to look for when learning about each of these filters.

Filter Basics

Digital filters are a very important part of DSP. In fact, their extraordinary performance is one of the key reasons that DSP has become so popular. As mentioned in the introduction, filters have two uses: signal *separation* and signal *restoration*. Signal separation is needed when a signal has been contaminated with interference, noise, or other signals. For example, imagine a device for measuring the electrical activity of a baby's heart (EKG) while still in the womb. The raw signal will likely be corrupted by the breathing and heartbeat of the mother. A filter might be used to separate these signals so that they can be individually analyzed.

Signal restoration is used when a signal has been distorted in some way. For example, an audio recording made with poor equipment may be filtered to better represent the sound as it actually occurred. Another example is the deblurring of an image acquired with an improperly focused lens, or a shaky camera.

These problems can be attacked with either analog or digital filters. Which is better? Analog filters are cheap, fast, and have a large dynamic range in both amplitude and frequency. Digital filters, in comparison, are vastly superior in the level of performance that can be achieved. For example, a low-pass digital filter presented in Chapter 16 has a gain of 1 +/- 0.0002 from DC to 1000 hertz, and a gain of less than 0.0002 for frequencies above

1001 hertz. The entire transition occurs within only 1 hertz. Don't expect this from an op amp circuit! Digital filters can achieve *thousands* of times better performance than analog filters. This makes a dramatic difference in how filtering problems are approached. With analog filters, the emphasis is on handling limitations of the electronics, such as the accuracy and

stability of the resistors and capacitors. In comparison, digital filters are so good that the performance of the filter is frequently ignored. The emphasis shifts to the limitations of the *signals*, and the *theoretical* issues regarding their processing.

It is common in DSP to say that a filter's input and output signals are in the *time domain*. This is because signals are usually created by sampling at regular intervals of *time*. But this is not the only way sampling can take place. The second most common way of sampling is at equal intervals in *space*. For example, imagine taking simultaneous readings from an array of strain sensors mounted at one centimeter increments along the length of an aircraft wing. Many other domains are possible; however, time and space are by far the most common. When you see the term *time domain* in DSP, remember that it may actually refer to samples taken over time, or it may be a general reference to any domain that the samples are taken in.

As shown in Fig. 14-1, every linear filter has an **impulse response**, a **step response** and a **frequency response.** Each of these responses contains complete information about the filter, but in a different form. If one of the three is specified, the other two are fixed and can be directly calculated. All three of these representations are important, because they describe how the filter will react under different circumstances.

The most straightforward way to implement a digital filter is by *convolving* the input signal with the digital filter's *impulse response*. All possible linear filters can be made in this manner. (This should be obvious. If it isn't, you probably don't have the background to understand this section on filter design. Try reviewing the previous section on DSP fundamentals). When the *impulse response* is used in this way, filter designers give it a special name: the **filter kernel**.

There is also another way to make digital filters, called **recursion**. When a filter is implemented by convolution, each sample in the output is calculated by *weighting* the samples in the input, and adding them together. Recursive filters are an extension of this, using previously calculated values from the *output*, besides points from the *input*. Instead of using a filter kernel, recursive filters are defined by a set of **recursion coefficients**. This method will be discussed in detail in Chapter 19. For now, the important point is that all linear filters have an impulse response, even if you don't use it to implement the filter. To find the impulse response of a recursive filter, simply feed in an impulse, and see what comes out. The impulse responses of recursive filters are composed of sinusoids that exponentially decay in amplitude. In principle, this makes their impulse responses *infinitely long*. However, the amplitude eventually drops below the round-off noise of the system, and the remaining samples can be ignored. Because

Frequency
0 0.1 0.2 0.3 0.4 0.5
-0.5
0.0
0.5
1.0
1.5
c. Frequency response
Sample number
0 32 64 96 128
-0.1
0.0
0.1
0.2
127
a. Impulse response
0.3

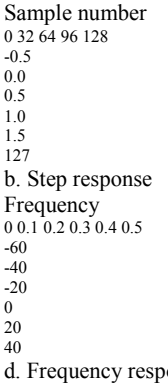


FIGURE 14-1
Filter parameters. Every linear filter has an impulse response, a step response, and a frequency response. The step response, (b), can be found by discrete integration of the impulse response, (a). The frequency response can be found from the impulse response by using the Fast Fourier Transform (FFT), and can be displayed either on a linear scale, (c), or in decibels, (d).

FFT

Integrate 20 Log( )

Amplitude

Amplitude Amplitude (dB)

Amplitude

of this characteristic, recursive filters are also called **Infinite Impulse Response or IIR** filters. In comparison, filters carried out by convolution are called **Finite Impulse Response or FIR** filters.

As you know, the *impulse response* is the output of a system when the input is an *impulse*. In this same manner, the *step response* is the output when the input is a *step* (also called an *edge*, and an *edge response*). Since the step is the integral of the impulse, the step response is the integral of the impulse response. This provides two ways to find the step response: (1) feed a step waveform into the filter and see what comes out, or (2) integrate the impulse response. (To be mathematically correct: *integration* is used with continuous signals, while *discrete integration*, i.e., a running sum, is used with discrete signals). The frequency response can be found by taking the DFT (using the FFT algorithm) of the impulse response. This will be reviewed later in this

$$dB \equiv 10 \log_{10} \frac{P_2}{P_1}$$

$$dB \equiv 20 \log_{10} \frac{A_2}{A_1}$$

EQUATION 14-1
Definition of decibels. Decibels are a way of expressing a *ratio* between two signals. Ratios of power (P_1 & P_2) use a different equation from ratios of amplitude (A_1 & A_2).

chapter. The frequency response can be plotted on a linear vertical axis, such as in (c), or on a logarithmic scale (decibels), as shown in (d). The linear scale is best at showing the passband ripple and roll-off, while the decibel scale is needed to show the stopband attenuation.

Don't remember decibels? Here is a quick review. A **bel** (in honor of Alexander Graham Bell) means that the power is changed by a *factor of ten*. For example, an electronic circuit that has 3 bels of amplification produces an

output signal with times the power of the input. A **decibel** 10×10×10 ’ 1000 **(dB)** is one-tenth of a bel. Therefore, the decibel values of: -20dB, -10dB, 0dB, 10dB & 20dB, mean the power ratios: 0.01, 0.1, 1, 10, & 100, respectively. In other words, every *ten* decibels mean that the power has changed by a factor of ten.

Here's the catch: you usually want to work with a signal's *amplitude*, not its *power*. For example, imagine an amplifier with 20dB of gain. By definition, this means that the power in the signal has increased by a factor of 100. Since amplitude is proportional to the square-root of power, the amplitude of the output is 10 times the amplitude of the input. While 20dB means a factor of 100 in power, it only means a factor of 10 in amplitude. Every *twenty* decibels mean that the amplitude has changed by a factor of ten. In equation form:

The above equations use the base 10 logarithm; however, many computer languages only provide a function for the base *e* logarithm (the natural log, written or). The natural log can be use by modifying the above $\log_e x$ $\ln x$ equations: and . dB ’ 4.342945 $\log_e (P_2/P_1)$ dB ’ 8.685890 $\log_e (A_2/A_1)$

Since decibels are a way of expressing the ratio between two signals, they are ideal for describing the gain of a system, i.e., the ratio between the output and the input signal. However, engineers also use decibels to specify the amplitude (or power) of a *single* signal, by referencing it to some standard. For example, the term: **dBV** means that the signal is being referenced to a 1 volt rms signal. Likewise, **dBm** indicates a reference signal producing 1 mW into a 600 ohms load (about 0.78 volts rms).

If you understand nothing else about decibels, remember two things: First, -3dB means that the amplitude is reduced to 0.707 (and the power is

60dB = 1000
40dB = 100
20dB = 10
0dB = 1
-20dB = 0.1
-40dB = 0.01
-60dB = 0.001

therefore reduced to 0.5). Second, memorize the following conversions between decibels and *amplitude* ratios:

How Information is Represented in Signals

The most important part of any DSP task is understanding how *information* is contained in the signals you are working with. There are many ways that information can be contained in a signal. This is especially true if the signal is manmade. For instance, consider all of the modulation schemes that have been devised: AM, FM, single-sideband, pulse-code modulation, pulse-width modulation, etc. The list goes on and on. Fortunately, there are only two ways that are common for information to be represented in naturally occurring signals. We will call these: **information represented in the time domain**, and **information represented in the frequency domain**.

Information represented in the time domain describes when something occurs and what the amplitude of the occurrence is. For example, imagine an experiment to study the light output from the sun. The light output is measured and recorded once each second. Each sample in the signal indicates what is happening at that instant, and the level of the event. If a solar flare occurs, the signal directly provides information on the time it occurred, the duration, the development over time, etc. Each sample contains information that is interpretable without reference to any other sample. Even if you have only one

sample from this signal, you still know something about what you are measuring. This is the simplest way for information to be contained in a signal.

In contrast, information represented in the frequency domain is more indirect. Many things in our universe show periodic motion. For example, a wine glass struck with a fingernail will vibrate, producing a ringing sound; the pendulum of a grandfather clock swings back and forth; stars and planets rotate on their axis and revolve around each other, and so forth. By measuring the frequency, phase, and amplitude of this periodic motion, information can often be obtained about the system producing the motion. Suppose we sample the sound produced by the ringing wine glass. The fundamental frequency and harmonics of the periodic vibration relate to the mass and elasticity of the material. A single sample, in itself, contains no information about the periodic motion, and therefore no information about the wine glass. The information is contained in the *relationship* between many points in the signal.

This brings us to the importance of the step and frequency responses. The *step response* describes how information represented in the *time domain* is being modified by the system. In contrast, the *frequency response* shows how information represented in the *frequency domain* is being changed. This distinction is absolutely critical in filter design because it is not possible to optimize a filter for both applications. Good performance in the time domain results in poor performance in the frequency domain, and vice versa. If you are designing a filter to remove noise from an EKG signal (information represented in the time domain), the step response is the important parameter, and the frequency response is of little concern. If your task is to design a digital filter for a hearing aid (with the information in the frequency domain), the frequency response is all important, while the step response doesn't matter. Now let's look at what makes a filter optimal for time domain or frequency domain applications.

Time Domain Parameters

It may not be obvious why the step response is of such concern in time domain filters. You may be wondering why the impulse response isn't the important parameter. The answer lies in the way that the human mind understands and processes information. Remember that the step, impulse and frequency responses all contain identical information, just in different arrangements. The step response is useful in time domain analysis because it matches the way humans view the information contained in the signals.

For example, suppose you are given a signal of some unknown origin and asked to analyze it. The first thing you will do is divide the signal into regions of similar characteristics. You can't stop from doing this; your mind will do it automatically. Some of the regions may be smooth; others may have large amplitude peaks; others may be noisy. This segmentation is accomplished by identifying the points that separate the regions. This is where the step function comes in. The step function is the purest way of representing a division between two dissimilar regions. It can mark when an event starts, or when an event ends. It tells you that whatever is on the *left* is somehow different from whatever is on the *right*. This is how the human mind views time domain information: a group of step functions dividing the information into regions of similar characteristics. The step response, in turn, is important because it describes how the dividing lines are being modified by the filter.

The step response parameters that are important in filter design are shown in Fig. 14-2. To distinguish events in a signal, the duration of the step response must be shorter than the spacing of the events. This dictates that the step response should be as *fast* (the DSP jargon) as possible. This is shown in Figs. (a) & (b). The most common way to specify the **risetime** (more jargon) is to quote the number of samples between the 10% and 90% amplitude levels. Why isn't a very fast risetime always possible? There are many reasons, noise reduction, inherent limitations of the data acquisition system, avoiding aliasing, etc.

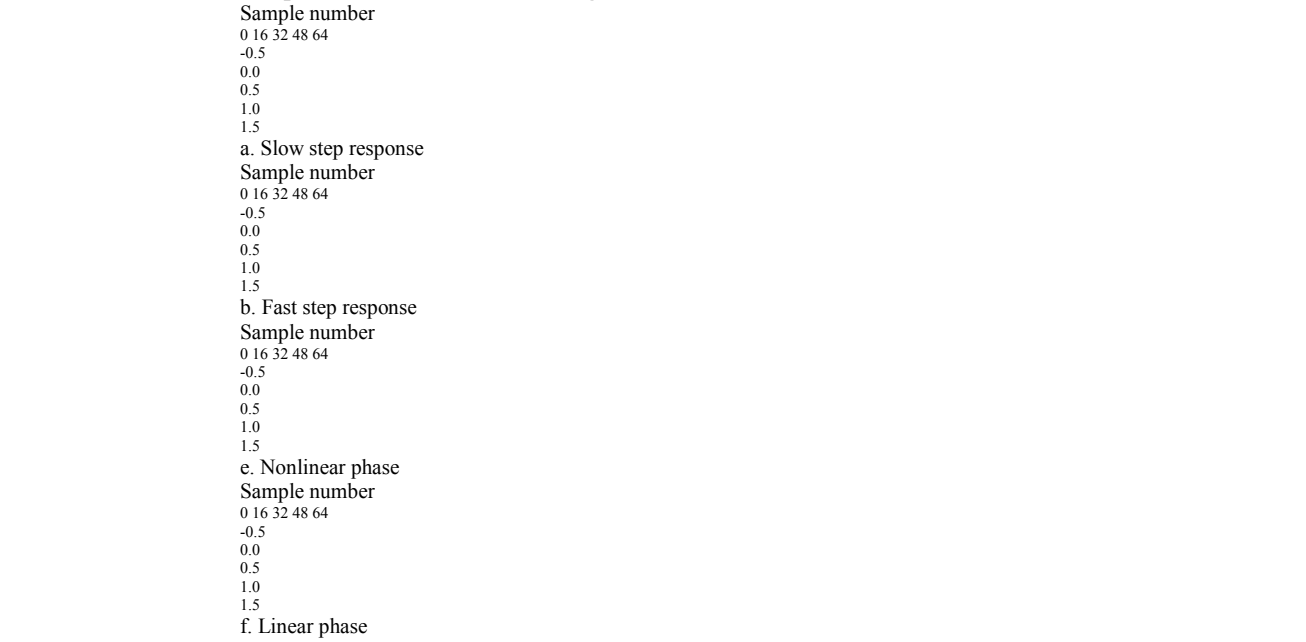
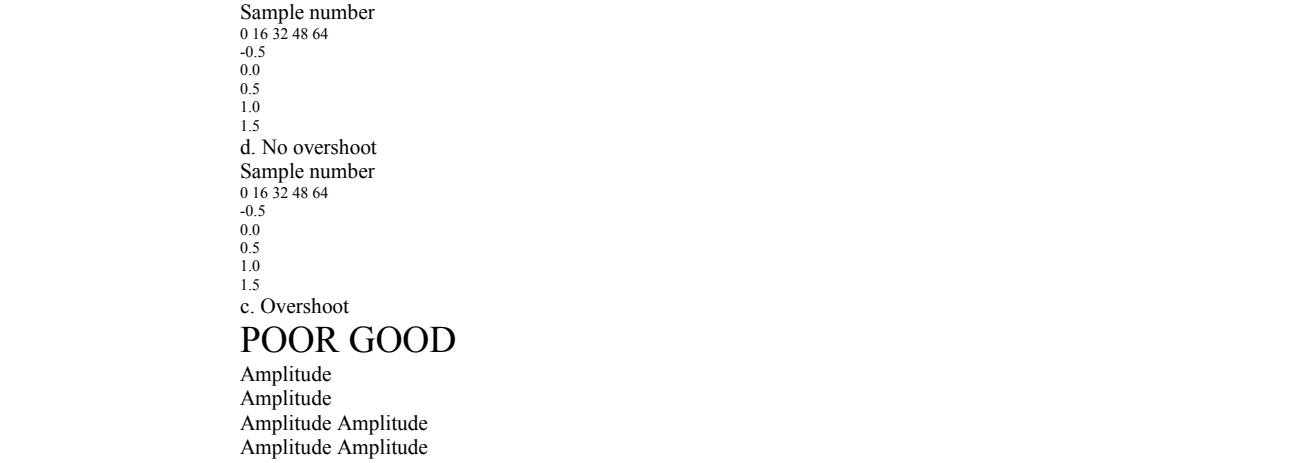


FIGURE 14-2

Parameters for evaluating *time domain* performance. The step response is used to measure how well a filter performs in the time domain. Three parameters are important: (1) transition speed (risetime), shown in (a) and (b), (2) overshoot, shown in (c) and (d), and (3) phase linearity (symmetry between the top and bottom halves of the step), shown in (e) and (f).



Figures (c) and (d) shows the next parameter that is important: **overshoot** in the step response. Overshoot must generally be eliminated because it changes the amplitude of samples in the signal; this is a basic distortion of the information contained in the time domain. This can be summed up in

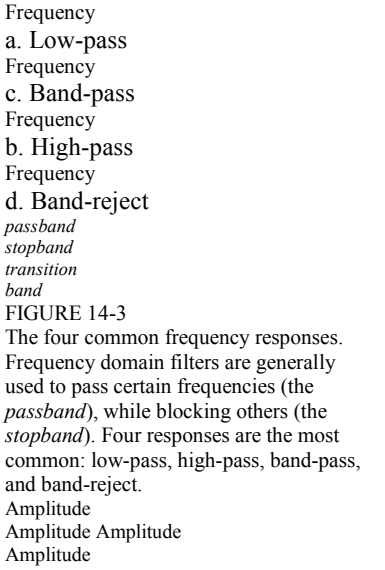


FIGURE 14-3
The four common frequency responses. Frequency domain filters are generally used to pass certain frequencies (the *passband*), while blocking others (the *stopband*). Four responses are the most common: low-pass, high-pass, band-pass, and band-reject.

Amplitude
Amplitude Amplitude
Amplitude

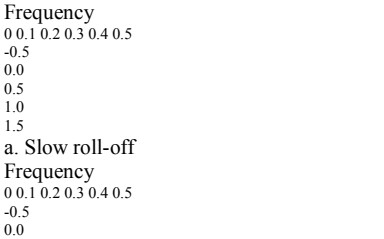
one question: Is the overshoot you observe in a signal coming from the thing you are trying to measure, or from the filter you have used?

Finally, it is often desired that the upper half of the step response be symmetrical with the lower half, as illustrated in (e) and (f). This symmetry is needed to make the *rising edges* look the same as the *falling edges*. This symmetry is called **linear phase**, because the frequency response has a phase that is a straight line (discussed in Chapter 19). Make sure you understand these three parameters; they are the key to evaluating time domain filters.

Frequency Domain Parameters

Figure 14-3 shows the four basic frequency responses. The purpose of these filters is to allow some frequencies to pass unaltered, while completely blocking other frequencies. The **passband** refers to those frequencies that are passed, while the **stopband** contains those frequencies that are blocked. The **transition band** is between. A **fast roll-off** means that the transition band is very narrow. The division between the passband and transition band is called the **cutoff frequency.** In analog filter design, the cutoff frequency is usually defined to be where the amplitude is reduced to 0.707 (i.e., -3dB). Digital filters are less standardized, and it is common to see 99%, 90%, 70.7%, and 50% amplitude levels defined to be the cutoff frequency.

Figure 14-4 shows three parameters that measure how well a filter performs in the frequency domain. To separate closely spaced frequencies, the filter must have a **fast roll-off**, as illustrated in (a) and (b). For the passband frequencies to move through the filter unaltered, there must be no **passband ripple**, as shown in (c) and (d). Lastly, to adequately block the stopband frequencies, it is necessary to have good **stopband attenuation**, displayed in (e) and (f).



0.5
1.0
1.5
b. Fast roll-off
Frequency
0 0.1 0.2 0.3 0.4 0.5
-120
-100
-80
-60
-40
-20
0
20
40
e. Poor stopband attenuation
Frequency
0 0.1 0.2 0.3 0.4 0.5
-120
-100
-80
-60
-40
-20
0
20
40
f. Good stopband attenuation
FIGURE 14-4
Parameters for evaluating *frequency domain* performance. The frequency responses shown are for low-pass filters. Three parameters are important: (1) roll-off sharpness, shown in (a) and (b), (2) passband ripple, shown in (c) and (d), and (3) stopband attenuation, shown in (e) and (f).
Frequency
0 0.1 0.2 0.3 0.4 0.5
-0.5
0.0
0.5
1.0
1.5
d. Flat passband
Frequency
0 0.1 0.2 0.3 0.4 0.5
-0.5
0.0
0.5
1.0
1.5
c. Ripple in passband
POOR GOOD
Amplitude (dB)
Amplitude (dB)
Amplitude Amplitude
Amplitude Amplitude
Why is there nothing about the *phase* in these parameters? First, the phase isn't important in most frequency domain applications. For example, the phase of an audio signal is almost completely random, and contains little useful information. Second, if the phase is important, it is very easy to make digital

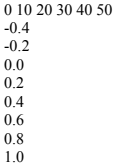
filters with a *perfect* phase response, i.e., all frequencies pass through the filter with a zero phase shift (also discussed in Chapter 19). In comparison, analog filters are ghastly in this respect.
Previous chapters have described how the DFT converts a system's impulse response into its frequency response. Here is a brief review. The quickest way to calculate the DFT is by means of the FFT algorithm presented in Chapter 12. Starting with a filter kernel N samples long, the FFT calculates the frequency spectrum consisting of an N point *real part* and an N point *imaginary part*. Only samples 0 to of the FFT's real and imaginary parts $N/2$ contain useful information; the remaining points are duplicates (negative frequencies) and can be ignored. Since the real and imaginary parts are difficult for humans to understand, they are usually converted into polar notation as described in Chapter 8. This provides the magnitude and phase

signals, each running from sample 0 to sample (i.e., samples in *N*/2 *N*/2%1 each signal). For example, an impulse response of 256 points will result in a frequency response running from point 0 to 128. Sample 0 represents DC, i.e., zero frequency. Sample 128 represents one-half of the sampling rate. Remember, no frequencies higher than one-half of the sampling rate can appear in sampled data.

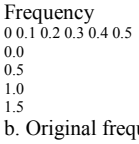
The number of samples used to represent the impulse response can be arbitrarily large. For instance, suppose you want to find the frequency response of a filter kernel that consists of 80 points. Since the FFT only works with signals that are a power of two, you need to add 48 zeros to the signal to bring it to a length of 128 samples. This *padding with zeros* does not change the impulse response. To understand why this is so, think about what happens to these added zeros when the input signal is convolved with the system's impulse response. The added zeros simply *vanish* in the convolution, and do not affect the outcome.

Taking this a step further, you could add *many* zeros to the impulse response to make it, say, 256, 512, or 1024 points long. The important idea is that longer impulse responses result in a closer spacing of the data points in the frequency response. That is, there are more samples spread between DC and one-half of the sampling rate. Taking this to the extreme, if the impulse response is padded with an *infinite* number of zeros, the data points in the frequency response are infinitesimally close together, i.e., a continuous line. In other words, the frequency response of a filter is really a *continuous* signal between DC and one-half of the sampling rate. The output of the DFT is a *sampling* of this continuous line. What length of impulse response should you use when calculating a filter's frequency response? As a first thought, try , but don't be afraid to change it if needed (such as insufficient *N*'1024 resolution or excessive computation time).

Keep in mind that the "good" and "bad" parameters discussed in this chapter are only generalizations. Many signals don't fall neatly into categories. For example, consider an EKG signal contaminated with 60 hertz interference. The information is encoded in the *time domain*, but the interference is best dealt with in the *frequency domain*. The best design for this application is



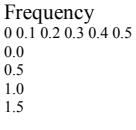
a. Original filter kernel



b. Original frequency response

FIGURE 14-5

Example of spectral inversion. The low-pass filter kernel in (a) has the frequency response shown in (b). A high-pass filter kernel, (c), is formed by changing the sign of each sample in (a), and adding one to the sample at the center of symmetry. This action in the time domain *inverts* the frequency spectrum (i.e., flips it top-forbottom), as shown by the high-pass frequency response in (d).



d. Inverted frequency response
Flipped
top-for-bottom
Sample number
0 10 20 30 40 50
-0.4
-0.2
0.0
0.2
0.4
0.6
0.8
1.0
c. Filter kernel with spectral inversion
Time Domain Frequency Domain
Amplitude Amplitude
Amplitude Amplitude
bound to have trade-offs, and might go against the conventional wisdom of this chapter. Remember the number one rule of education: *A paragraph in a book doesn't give you a license to stop thinking*.

High-Pass, Band-Pass and Band-Reject Filters

High-pass, band-pass and band-reject filters are designed by starting with a low-pass filter, and then converting it into the desired response. For this reason, most discussions on filter design only give examples of low-pass filters. There are two methods for the low-pass to high-pass conversion: **spectral inversion** and **spectral reversal**. Both are equally useful.

An example of *spectral inversion* is shown in 14-5. Figure (a) shows a lowpass filter kernel called a windowed-sinc (the topic of Chapter 16). This filter kernel is 51 points in length, although many of samples have a value so small that they appear to be zero in this graph. The corresponding

x[n] y[n]
*[n] - h[n] x[n] y[n]
h[n]
*[n]
Low-pass
All-pass
High-pass b. High-pass
in a single stage
a. High-pass by
adding parallel stages

FIGURE 14-6
Block diagram of spectral inversion. In (a), the input signal, , is applied to two $x[n]$ systems in parallel, having impulse responses of and . As shown in $h[n]$ $*[n]$ (b), the combined system has an impulse response of . This means that $*[n]$& $h[n]$ the frequency response of the combined system is the *inversion* of the frequency response of . $h[n]$

frequency response is shown in (b), found by adding 13 zeros to the filter kernel and taking a 64 point FFT. Two things must be done to change the low-pass filter kernel into a high-pass filter kernel. First, change the sign of each sample in the filter kernel. Second, add *one* to the sample at the center of symmetry. This results in the high-pass filter kernel shown in (c), with the frequency response shown in (d). Spectral inversion *flips* the frequency response *top-for-bottom*, changing the passbands into stopbands, and the stopbands into passbands. In other words, it changes a filter from low-pass to high-pass, high-pass to low-pass, band-pass to band-reject, or band-reject to

band-pass.
Figure 14-6 shows why this two step modification to the time domain results in an inverted frequency spectrum. In (a), the input signal, , is applied to $x[n]$ two systems in parallel. One of these systems is a low-pass filter, with an impulse response given by . The other system does *nothing* to the signal, $h[n]$ and therefore has an impulse response that is a delta function, . The $*[n]$ overall output, , is equal to the output of the all-pass system *minus* the $y[n]$ output of the low-pass system. Since the low frequency components are subtracted from the original signal, only the high frequency components appear in the output. Thus, a high-pass filter is formed.
This could be performed as a two step operation in a computer program: run the signal through a low-pass filter, and then subtract the filtered signal from the original. However, the entire operation can be performed in a signal stage by combining the two filter kernels. As described in Chapter

Sample number
0 10 20 30 40 50
-0.4
-0.2
0.0
0.2
0.4
0.6
0.8
1.0
a. Original filter kernel
Frequency
0 0.1 0.2 0.3 0.4 0.5
0.0
0.5
1.0
1.5
b. Original frequency response
FIGURE 14-7
Example of spectral reversal. The low-pass filter kernel in (a) has the frequency response shown in (b). A high-pass filter kernel, (c), is formed by changing the sign of every other sample in (a). This action in the time domain results in the frequency domain being flipped *left-for-right*, resulting in the high-pass frequency response shown in (d).
Frequency
0 0.1 0.2 0.3 0.4 0.5
0.0
0.5
1.0
1.5
d. Reversed frequency response
Flipped
left-for-right
Sample number
0 10 20 30 40 50
-0.4
-0.2
0.0
0.2
0.4
0.6
0.8
1.0
c. Filter kernel with spectral reversal
Time Domain Frequency Domain
Amplitude Amplitude
Amplitude Amplitude
7, parallel systems with added outputs can be combined into a single stage by adding their impulse responses. As shown in (b), the filter kernel for the highpass filter is given by: . That is, change the sign of all the samples, $*[n]$ & $h[n]$ and then add one to the sample at the center of symmetry.
For this technique to work, the low-frequency components exiting the low-pass filter must have the same phase as the low-frequency components exiting the all-pass system. Otherwise a complete subtraction cannot take place. This

places two restrictions on the method: (1) the original filter kernel must have left-right symmetry (i.e., a zero or linear phase), and (2) the impulse must be added at the center of symmetry.

The second method for low-pass to high-pass conversion, *spectral reversal*, is illustrated in Fig. 14-7. Just as before, the low-pass filter kernel in (a) corresponds to the frequency response in (b). The high-pass filter kernel, (c), is formed by *changing the sign of every other sample* in (a). As shown in (d), this flips the frequency domain *left-for-right*: 0 becomes 0.5 and 0.5

$h_1[n]$ x[n] $h_2[n]$ y[n]

$h_1[n]$ $h_2[n]$ x[n] y[n]

Band-pass

Low-pass High-pass a. Band-pass by cascading stages

b. Band-pass in a single stage

FIGURE 14-8
Designing a band-pass filter. As shown in (a), a band-pass filter can be formed by cascading a low-pass filter and a high-pass filter. This can be reduced to a single stage, shown in (b). The filter kernel of the single stage is equal to the *convolution* of the low-pass and highpass filter kernels.

becomes 0. The cutoff frequency of the example low-pass filter is 0.15, resulting in the cutoff frequency of the high-pass filter being 0.35.

Changing the sign of every other sample is equivalent to multiplying the filter kernel by a sinusoid with a frequency of 0.5. As discussed in Chapter 10, this has the effect of *shifting* the frequency domain by 0.5. Look at (b) and imagine the negative frequencies between -0.5 and 0 that are of mirror image of the frequencies between 0 and 0.5. The frequencies that appear in (d) are the negative frequencies from (b) shifted by 0.5.

Lastly, Figs. 14-8 and 14-9 show how low-pass and high-pass filter kernels can be combined to form band-pass and band-reject filters. In short, *adding* the filter kernels produces a *band-reject* filter, while *convolving* the filter kernels produces a *band-pass* filter. These are based on the way cascaded and parallel systems are be combined, as discussed in Chapter 7. Multiple combination of these techniques can also be used. For instance, a band-pass filter can be designed by adding the two filter kernels to form a stop-pass filter, and then use *spectral inversion* or *spectral reversal* as previously described. All these techniques work very well with few surprises.

Filter Classification

Table 14-1 summarizes how digital filters are classified by their *use* and by their *implementation*. The use of a digital filter can be broken into three categories: *time domain*, *frequency domain* and *custom*. As previously described, time domain filters are used when the information is encoded in the shape of the signal's waveform. Time domain filtering is used for such actions as: smoothing, DC removal, waveform shaping, etc. In contrast, frequency domain filters are used when the information is contained in the

x[n] y[n]

$h_1[n] + h_2[n]$ x[n] y[n]

$h_1[n]$

$h_2[n]$
Low-pass
High-pass
Band-reject b. Band-reject
in a single stage
a. Band-reject by
adding parallel stages
FIGURE 14-9
Designing a band-reject filter. As shown in (a), a band-reject filter is formed by the parallel combination of a low-pass filter and a high-pass filter with their outputs added. Figure (b) shows this reduced to a single stage, with the filter kernel found by *adding* the low-pass and high-pass filter kernels.
Recursion
Time Domain
Frequency Domain
Finite Impulse Response (FIR) Infinite Impulse Response (IIR)
Moving average (Ch. 15) Single pole (Ch. 19)
Windowed-sinc (Ch. 16) Chebyshev (Ch. 20)
Custom FIR custom (Ch. 17) Iterative design (Ch. 26)
(Deconvolution)
Convolution
FILTER IMPLEMENTED BY:
(smoothing, DC removal)
(separating frequencies)
FILTER USED FOR:
TABLE 14-1
Filter classification. Filters can be divided by their *use*, and how they are *implemented*.

amplitude, frequency, and phase of the component sinusoids. The goal of these filters is to separate one band of frequencies from another. Custom filters are used when a special action is required by the filter, something more elaborate than the four basic responses (high-pass, low-pass, band-pass and band-reject). For instance, Chapter 17 describes how custom filters can be used for *deconvolution*, a way of counteracting an unwanted convolution.

Digital filters can be implemented in two ways, by *convolution* (also called *finite impulse response* or *FIR*) and by *recursion* (also called *infinite impulse response* or *IIR*). Filters carried out by convolution can have far better performance than filters using recursion, but execute much more slowly.

The next six chapters describe digital filters according to the classifications in Table 14-1. First, we will look at filters carried out by convolution. The *moving average* (Chapter 15) is used in the time domain, the *windowed-sinc* (Chapter 16) is used in the frequency domain, and *FIR custom* (Chapter 17) is used when something special is needed. To finish the discussion of FIR filters, Chapter 18 presents a technique called FFT convolution. This is an algorithm for increasing the speed of convolution, allowing FIR filters to execute faster.

Next, we look at recursive filters. The *single pole* recursive filter (Chapter 19) is used in the time domain, while the *Chebyshev* (Chapter 20) is used in the frequency domain. Recursive filters having a custom response are designed by *iterative techniques*. For this reason, we will delay their discussion until Chapter 26, where they will be presented with another type of iterative procedure: the neural network.

As shown in Table 14-1, *convolution* and *recursion* are rival techniques; you must use one or the other for a particular application. How do you choose?

Chapter 21 presents a head-to-head comparison of the two, in both the time and frequency domains.

285

CHAPTER

16

$h[i] = \frac{\sin(2\pi f_c i)}{i\pi}$

Windowed-Sinc Filters

Windowed-sinc filters are used to separate one band of frequencies from another. They are very stable, produce few surprises, and can be pushed to incredible performance levels. These exceptional frequency domain characteristics are obtained at the expense of poor performance in the time domain, including excessive ripple and overshoot in the step response. When carried out by standard convolution, windowed-sinc filters are easy to program, but slow to execute. Chapter 18 shows how the FFT can be used to dramatically improve the computational speed of these filters.

Strategy of the Windowed-Sinc

Figure 16-1 illustrates the idea behind the windowed-sinc filter. In (a), the frequency response of the *ideal* low-pass filter is shown. All frequencies below the cutoff frequency, , are passed with unity amplitude, while all higher f_c frequencies are blocked. The passband is perfectly flat, the attenuation in the stopband is infinite, and the transition between the two is infinitesimally small. Taking the Inverse Fourier Transform of this ideal frequency response produces the ideal filter kernel (impulse response) shown in (b). As previously discussed (see Chapter 11, Eq. 11-4), this curve is of the general form: , called $\sin(x)/x$ the **sinc function**, given by:

Convolving an input signal with this filter kernel provides a *perfect* low-pass filter. The problem is, the sinc function continues to both negative and positive infinity without dropping to zero amplitude. While this infinite length is not a problem for *mathematics*, it is a show stopper for *computers*.

$$w[i] = 0.54 - 0.46\cos(2\pi i / M)$$

EQUATION 16-1

The Hamming window. These windows run from to M, $i = 0$ for a total of points. $M + 1$

$$w[i] = 0.42 - 0.5\cos(2\pi i / M) + 0.08\cos(4\pi i / M)$$ EQUATION 16-2

The Blackman window.

FIGURE 16-1 (facing page)

Derivation of the windowed-sinc filter kernel. The frequency response of the ideal low-pass filter is shown in (a), with the corresponding filter kernel in (b), a sinc function. Since the sinc is infinitely long, it must be truncated to be used in a computer, as shown in (c). However, this truncation results in undesirable changes in the frequency response, (d). The solution is to multiply the truncated-sinc with a smooth window, (e), resulting in the windowed-sinc filter kernel, (f). The frequency response of the windowed-sinc, (g), is smooth and well behaved. These figures are not to scale.

To get around this problem, we will make two modifications to the sinc function in (b), resulting in the waveform shown in (c). First, it is truncated to points, symmetrically chosen around the main lobe, where M is an $M+1$ even number. All samples outside these points are set to zero, or simply $M+1$ ignored. Second, the entire sequence is shifted to the right so that it runs from

0 to *M*. This allows the filter kernel to be represented using only *positive* indexes. While many programming languages allow *negative* indexes, they are a nuisance to use. The sole effect of this shift in the filter kernel is to *M*/2 shift the output signal by the same amount.

Since the modified filter kernel is only an approximation to the ideal filter kernel, it will not have an ideal frequency response. To find the frequency response that is obtained, the Fourier transform can be taken of the signal in (c), resulting in the curve in (d). It's a mess! There is excessive ripple in the passband and poor attenuation in the stopband (recall the Gibbs effect discussed in Chapter 11). These problems result from the abrupt discontinuity at the ends of the truncated sinc function. Increasing the length of the filter kernel does not reduce these problems; the discontinuity is significant no matter how long *M* is made.

Fortunately, there is a simple method of improving this situation. Figure (e) shows a smoothly tapered curve called a **Blackman window**. Multiplying the truncated-sinc, (c), by the Blackman window, (e), results in the **windowedsinc** filter kernel shown in (f). The idea is to reduce the abruptness of the truncated ends and thereby improve the frequency response. Figure (g) shows this improvement. The passband is now flat, and the stopband attenuation is so good it cannot be seen in this graph.

Several different windows are available, most of them named after their original developers in the 1950s. Only two are worth using, the **Hamming window** and the **Blackman window** These are given by:

Figure 16-2a shows the shape of these two windows for (i.e., 51 total *M'* 50 points in the curves). Which of these two windows should you use? It's a trade-off between parameters. As shown in Fig. 16-2b, the Hamming window has about a 20% faster *roll-off* than the Blackman. However,

Sample number
0 1
-0.5
0.0
0.5
1.0
1.5
M
e. Blackman or Hamming window
Frequency Domain
Frequency
0 0.5
-0.5
0.0
0.5
1.0
1.5
g. Windowed-sinc frequency response
fc
Sample number
0 1
-0.5
0.0
0.5
1.0
1.5
M
f. Windowed-sinc filter kernel
FIGURE 16-1
Amplitude
Amplitude Amplitude
Amplitude Amplitude
Amplitude
Amplitude

Sample number
0 10 20 30 40 50
0.0
0.5
1.0
1.5
a. Blackman and Hamming window
Blackman
Hamming
Frequency
0 0.1 0.2 0.3 0.4 0.5
0.0
0.5
1.0
1.5
b. Frequency response
Hamming
Blackman
Frequency
0 0.1 0.2 0.3 0.4 0.5
-120
-100
-80
-60
-40
-20
0
20
40
c. Frequency response (dB)
Blackman
Hamming
FIGURE 16-2
Characteristics of the Blackman and Hamming windows. The shapes of these two windows are shown in (a), and given by Eqs. 16-1 and 16-2. As shown in (b), the Hamming window results in about 20% faster roll-off than the Blackman window. However, the Blackman window has better stopband attenuation (Blackman: 0.02%, Hamming: 0.2%), and a lower passband ripple (Blackman:

0.02% Hamming: 0.2%).
Amplitude (dB) Amplitude
Amplitude
(c) shows that the Blackman has a better *stopband attenuation*. To be exact, the stopband attenuation for the Blackman is -74dB (-0.02%), while the Hamming is only -53dB (-0.2%). Although it cannot be seen in these graphs, the Blackman has a *passband ripple* of only about 0.02%, while the Hamming is typically 0.2%. In general, the Blackman should be your first choice; a slow roll-off is easier to handle than poor stopband attenuation.

There are other windows you might hear about, although they fall short of the Blackman and Hamming. The **Bartlett window** is a triangle, using straight lines for the taper. The **Hanning window,** also called the **raised cosine window**, is given by: . These two windows have $w[i]$ ’ 0.5& 0.5 cos(2Bi /M) about the same roll-off speed as the Hamming, but worse stopband attenuation (Bartlett: -25dB or 5.6%, Hanning -44dB or 0.63%). You might also hear of a **rectangular window**. This is the same as *no* window, just a truncation of the tails (such as in Fig. 16-1c). While the roll-off is -2.5 times faster than the Blackman, the stopband attenuation is only -21dB (8.9%).

Designing the Filter

To design a windowed-sinc, two parameters must be selected: the cutoff frequency, , and the length of the filter kernel, M. The cutoff frequency f_C

Frequency
0 0.1 0.2 0.3 0.4 0.5
0.0
0.5
1.0
1.5
fc=0.05 fc=0.25
fc=0.45
b. Roll-off vs. cutoff frequency
FIGURE 16-3
Filter length vs. roll-off of the windowed-sinc filter. As shown in (a), for M = 20, 40, and 200, the transition bandwidths are BW = 0.2, 0.1, and 0.02 of the sampling rate, respectively. As shown in (b), the shape of the frequency response does not change with different cutoff frequencies. In (b), M = 60.
Frequency
0 0.1 0.2 0.3 0.4 0.5
0.0
0.5
1.0
1.5
M=40 M=200
M=20
a. Roll-off vs. kernel length
Amplitude
Amplitude

$$M \approx \frac{4}{BW}$$

EQUATION 16-3
Filter length vs. roll-off. The length of the filter kernel, M, determines the transition bandwidth of the filter, BW. This is only an approximation since roll-off depends on the particular window being used.

is expressed as a fraction of the sampling rate, and therefore must be between 0 and 0.5. The value for M sets the *roll-off* according to the approximation: where BW is the width of the transition band, measured from where the curve just barely leaves one, to where it almost reaches zero (say, 99% to 1% of the curve). The transition bandwidth is also expressed as a fraction of the sampling frequency, and must between 0 and 0.5. Figure 16-3a shows an example of how this approximation is used. The three curves shown are

generated from filter kernels with: . From Eq. 16-3, the *M* ' 20, 40, and 200 transition bandwidths are: , respectively. Figure (b) *BW* ' 0.2, 0.1, and 0.02 shows that the shape of the frequency response does not depend on the cutoff frequency selected.

Since the time required for a convolution is proportional to the length of the signals, Eq. 16-3 expresses a trade-off between *computation time* (depends on the value of *M*) and *filter sharpness* (the value of *BW*). For instance, the 20% slower roll-off of the Blackman window (as compared with the Hamming) can be compensated for by using a filter kernel 20% longer. In other words, it could be said that the Blackman window is 20% slower to execute that an equivalent roll-off Hamming window. This is important because the execution speed of windowed-sinc filters is already terribly slow.

As also shown in Fig. 16-3b, the cutoff frequency of the windowed-sinc filter is measured at the *one-half amplitude* point. Why use 0.5 instead of the

$$h[i] = K \cdot \frac{\sin(2\pi f_C (i - M/2))}{i - M/2} \cdot \left(0.42 - 0.5\cos\frac{2\pi i}{M} + 0.08\cos\frac{4\pi i}{M}\right)$$

EQUATION 16-4

The windowed-sinc filter kernel. The cutoff frequency, , is expressed as a f_C fraction of the sampling rate, a value between 0 and 0.5. The length of the filter kernel is determined by M, which must be an even integer. The sample number i, is an integer that runs from 0 to M, resulting in total points in the filter $M+1$ kernel. The constant, K, is chosen to provide unity gain at zero frequency. To avoid a divide-by-zero error, for , use . $i = M/2$ $h[i] = 2\pi f_C K$

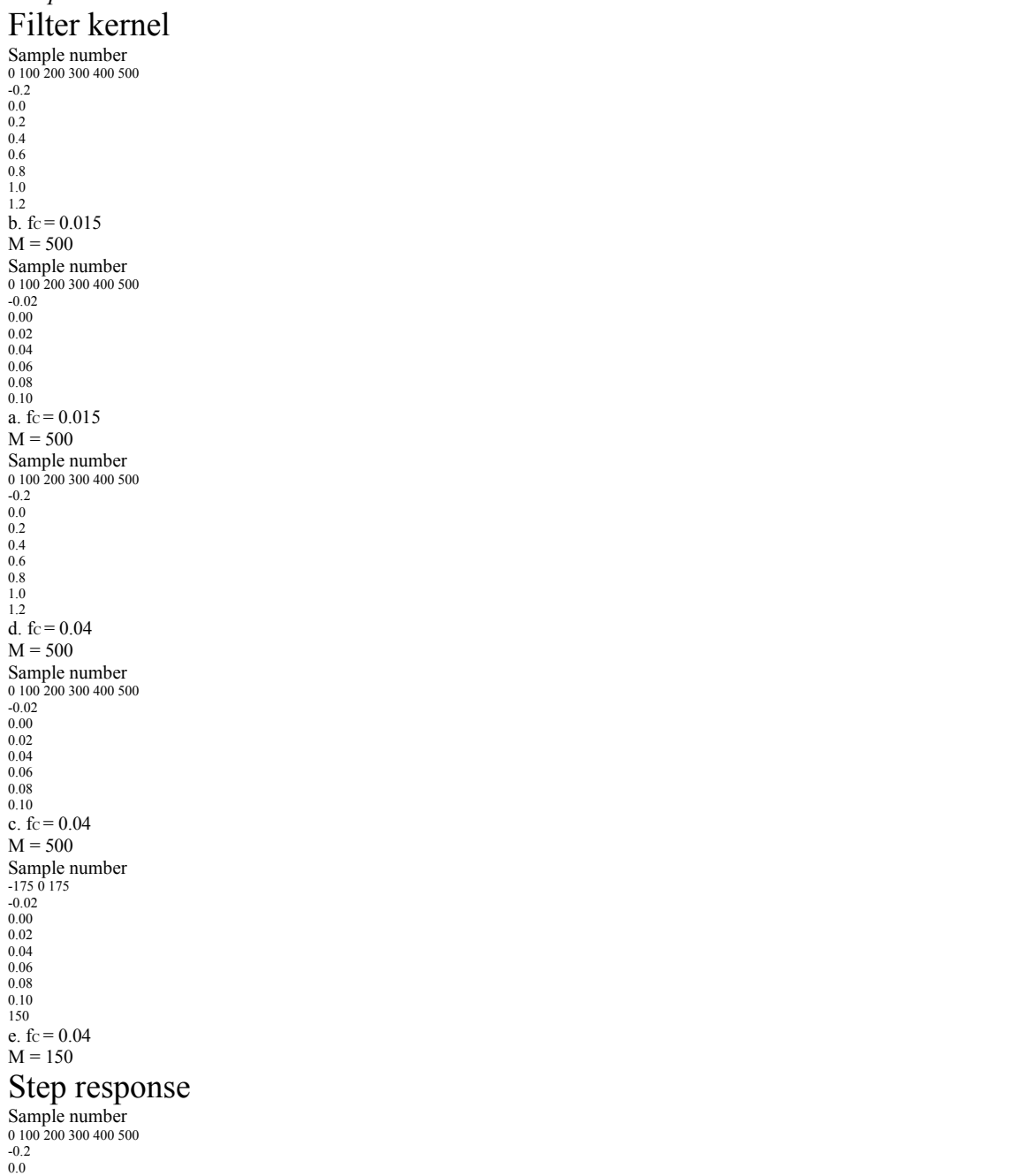
standard 0.707 (-3dB) used in analog electronics and other digital filters? This is because the windowed-sinc's frequency response is *symmetrical* between the passband and the stopband. For instance, the Hamming window results in a passband ripple of 0.2%, and an *identical* stopband attenuation (i.e., ripple in the stopband) of 0.2%. Other filters do not show this symmetry, and therefore have no advantage in using the one-half amplitude point to mark the cutoff frequency. As shown later in this chapter, this symmetry makes the windowedsinc ideal for *spectral inversion*.

After and *M* have been selected, the filter kernel is calculated from the f_C relation:

Don't be intimidated by this equation! Based on the previous discussion, you should be able to identify three components: the *sinc function*, the *M/2 shift*, and the *Blackman window*. For the filter to have unity gain at DC, the constant *K* must be chosen such that the sum of all the samples is equal to one. In practice, ignore *K* during the calculation of the filter kernel, and then *normalize* all of the samples as needed. The program listed in Table 16-1 shows how this is done. Also notice how the calculation is handled at the center of the sinc, , which involves a division by zero. $i = M/2$

This equation may be long, but it is easy to use; simply type it into your computer program and forget it. Let the computer handle the calculations. If you find yourself trying to evaluate this equation by hand, you are doing something very very wrong.

Let's be specific about where the filter kernel described by Eq. 16-4 is located in your computer array. As an example, *M* will be chosen to be 100. Remember, *M* must be an even number. The first point in the filter kernel is in array location 0, while the last point is in array location 100. This means that the entire signal is 101 points long. The center of symmetry is at point 50, i.e., . The 50 points to the left of point 50 are symmetrical with the 50 *M*/2 points to the right. Point 0 is the same value as point 100, and point 49 is the same as point 51. If you must have a specific number of samples in the filter kernel, such as to use the FFT, simply add zeros to one end or the other. For example, with , you could make samples 101 through 127 equal to *M'* 100 zero, resulting in a filter kernel 128 points long.



0.2
0.4
0.6
0.8
1.0
1.2
f. $f_C = 0.04$
M = 150
FIGURE 16-4
Example filter kernels and the corresponding step responses. The frequency of the sinusoidal oscillation is approximately equal to the cutoff frequency, , while *M* determines the kernel length. f_C
Amplitude
Amplitude
Amplitude Amplitude
Amplitude Amplitude
Figure 16-4 shows examples of windowed-sinc filter kernels, and their corresponding step responses. The samples at the beginning and end of the filter kernels are so small that they can't even be seen in the graphs. Don't make the mistake of thinking they are unimportant! These samples may be small in value; however, they collectively have a large effect on the

FIGURE 16-5
Example of windowed-sinc filters. The alpha and beta rhythms in an EEG are separated by low-pass and highpass filters with . The program to implement the low-pass filter is shown in Table 16-1. The program $M' 100$ for the high-pass filter is identical, except for a *spectral inversion* of the low-pass filter kernel.
Frequency
0 0.1 0.2 0.3 0.4 0.5
0.0
0.5
1.0
1.5
a. Low-pass filter
Frequency
0 0.1 0.2 0.3 0.4 0.5
0.0
0.5
1.0
1.5
b. High-pass filter
Amplitude
Amplitude
Alpha wave
Beta wave
Alpha wave
Beta wave
performance of the filter. This is also why floating point representation is typically used to implement windowed-sinc filters. Integers usually don't have enough dynamic range to capture the large variation of values contained in the filter kernel. How does the windowed-sinc filter perform in the time domain? Terrible! The step response has overshoot and ringing; this is *not* a filter for signals with information encoded in the time domain.

Examples of Windowed-Sinc Filters

An electroencephalogram, or EEG, is a measurement of the electrical activity of the brain. It can be detected as millivolt level signals appearing on electrodes attached to the surface of the head. Each nerve cell in the brain generates small electrical pulses. The EEG is the combined result of an enormous number of these electrical pulses being generated in a (hopefully) coordinated manner. Although the relationship between thought and this electrical coordination is very poorly understood, different frequencies in the EEG can be identified with specific mental states. If you close your eyes and relax, the predominant EEG pattern will be a slow oscillation between about 7 and 12 hertz. This waveform is called the *alpha rhythm*, and is associated with contentment and a decreased level of attention. Opening your eyes and looking around causes the EEG to change

to the *beta rhythm*, occurring between about 17 and 20 hertz. Other frequencies and waveforms are seen in children, different depths of sleep, and various brain disorders such as epilepsy.

In this example, we will assume that the EEG signal has been amplified by analog electronics, and then digitized at a sampling rate of 100 samples per second. Acquiring data for 50 seconds produces a signal of 5,000 points. Our goal is to separate the alpha from the beta rhythms. To do this, we will design a digital low-pass filter with a cutoff frequency of 14 hertz, or 0.14

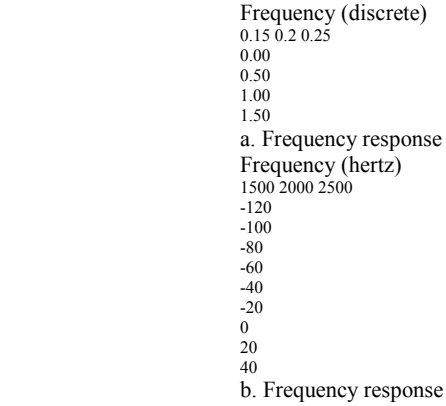


FIGURE 16-6
Example of a windowed-sinc band-pass filter. This filter was designed for a sampling rate of 10 kHz. When referenced to the analog signal, the center frequency of the passband is at 2 kHz, the passband is 80 hertz, and the transition bands are 50 hertz. The windowed-sinc uses 801 points in the filter kernel to achieve this roll-off, and a Blackman window for good stopband attenuation. Figure (a) shows the resulting frequency response on a linear scale, while (b) shows it in decibels. The frequency axis in (a) is expressed as a fraction of the sampling frequency, while (b) is expressed in terms of the analog signal before digitization.

of the sampling rate. The transition bandwidth will be set at 4 hertz, or 0.04 of the sampling rate. From Eq. 16-3, the filter kernel needs to be about 101 points long, and we will arbitrarily choose to use a Hamming window. The program in Table 16-1 shows how the filter is carried out. The frequency response of the filter, obtained by taking the Fourier Transform of the filter kernel, is shown in Fig. 16-5.

In a second example, we will design a *band-pass filter* to isolate a *signaling tone* in an audio signal, such as when a button on a telephone is pressed. We will assume that the signal has been digitized at 10 kHz, and the goal is to isolate an 80 hertz band of frequencies centered on 2 kHz. In terms of the sampling rate, we want to block all frequencies below 0.196 and above 0.204 (corresponding to 1960 hertz and 2040 hertz, respectively). To achieve a transition bandwidth of 50 hertz (0.005 of the sampling rate), we will make the filter kernel 801 points long, and use a Blackman window. Table 16-2 contains a program for calculating the filter kernel, while Fig. 16-6 shows the frequency response. The design involves several steps. First, *two* low-pass filters are designed, one with a cutoff at 0.196, and the other with a cutoff at 0.204. This second filter is then *spectrally inverted*, making it a high-pass filter (see Chapter 14, Fig. 14-6). Next, the two filter kernels are added, resulting in a band-reject filter (see Fig. 14-8). Finally, another *spectral inversion* makes this into the desired band-pass filter.

Pushing it to the Limit

The windowed-sinc filter can be pushed to incredible performance levels without nasty surprises. For instance, suppose you need to isolate a 1 *millivolt*

signal riding on a 120 *volt* power line. The low-pass filter will need


```
100 'LOW-PASS WINDOWED-SINC FILTER
110 'This program filters 5000 samples with a 101 point windowed-sinc filter,
120 'resulting in 4900 samples of filtered data.
130 '
140 DIM X[4999] 'X[ ] holds the input signal
150 DIM Y[4999] 'Y[ ] holds the output signal
160 DIM H[100] 'H[ ] holds the filter kernel
170 '
180 PI = 3.14159265
190 FC = .14 'Set the cutoff frequency (between 0 and 0.5)
200 M% = 100 'Set filter length (101 points)
210 '
220 GOSUB XXXX 'Mythical subroutine to load X[ ]
230 '
240 ' 'Calculate the low-pass filter kernel via Eq. 16-4
250 FOR I% = 0 TO 100
260 IF (I%-M%/2) = 0 THEN H[I%] = 2*PI*FC
270 IF (I%-M%/2) <> 0 THEN H[I%] = SIN(2*PI*FC * (I%-M%/2)) / (I%-M%/2)
280 H[I%] = H[I%] * (0.54 - 0.46*COS(2*PI*I%/M%) )
290 NEXT I%
300 '
310 SUM = 0 'Normalize the low-pass filter kernel for
320 FOR I% = 0 TO 100 'unity gain at DC
330 SUM = SUM + H[I%]
340 NEXT I%
350 '
360 FOR I% = 0 TO 100
370 H[I%] = H[I%] / SUM
380 NEXT I%
390 '
400 FOR J% = 100 TO 4999 'Convolve the input signal & filter kernel
410 Y[J%] = 0
420 FOR I% = 0 TO 100
430 Y[J%] = Y[J%] + X[J%-I%] * H[I%]
440 NEXT I%
450 NEXT J%
460 '
470 END
```

TABLE 16-1

a stopband attenuation of at least -120dB (one part in one-million for those that refuse to learn decibels). As previously shown, the Blackman window only provides -74dB (one part in five-thousand). Fortunately, greater stopband attenuation is easy to obtain. The input signal can be filtered using a conventional windowed-sinc filter kernel, providing an intermediate signal. The intermediate signal can then be passed through the filter a second time, further increasing the stopband attenuation to -148dB (1 part in 30 million, wow!). It is also possible to combine the two stages into a single filter. The kernel of the combined filter is equal to the *convolution* of the filter kernels of the two stages. This also means that convolving any filter kernel *with itself* results in a filter kernel with a much improved stopband attenuation. The price you pay is a longer filter kernel and a slower roll-off. Figure 16-7a shows the frequency response of a 201 point lowpass filter, formed by convolving a 101 point Blackman windowed-sinc with itself. Amazing performance! (If you really need more than -100dB of stopband attenuation, you should use double precision. Single precision


```
100 'BAND-PASS WINDOWED-SINC FILTER
```

```
110 'This program calculates an 801 point band-pass filter kernel
120 '
130 DIM A[800] 'A[ ] workspace for the lower cutoff
140 DIM B[800] 'B[ ] workspace for the upper cutoff
150 DIM H[800] 'H[ ] holds the final filter kernel
160 '
170 PI = 3.1415926
180 M% = 800 'Set filter kernel length (801 points)
190 '
200 ' 'Calculate the first low-pass filter kernel via Eq. 16-4,
210 FC = 0.196 'with a cutoff frequency of 0.196, store in A[ ]
220 FOR I% = 0 TO 800
230 IF (I%-M%/2) = 0 THEN A[I%] = 2*PI*FC
240 IF (I%-M%/2) <> 0 THEN A[I%] = SIN(2*PI*FC * (I%-M%/2)) / (I%-M%/2)
250 A[I%] = A[I%] * (0.42 - 0.5*COS(2*PI*I%/M%) + 0.08*COS(4*PI*I%/M%))
260 NEXT I%
270 '
280 SUM = 0 'Normalize the first low-pass filter kernel for
290 FOR I% = 0 TO 800 'unity gain at DC
300 SUM = SUM + A[I%]
310 NEXT I%
320 '
330 FOR I% = 0 TO 800
340 A[I%] = A[I%] / SUM
350 NEXT I%
360 ' 'Calculate the second low-pass filter kernel via Eq. 16-4,
370 FC = 0.204 'with a cutoff frequency of 0.204, store in B[ ]
380 FOR I% = 0 TO 800
390 IF (I%-M%/2) = 0 THEN B[I%] = 2*PI*FC
400 IF (I%-M%/2) <> 0 THEN B[I%] = SIN(2*PI*FC * (I%-M%/2)) / (I%-M%/2)
410 B[I%] = B[I%] * (0.42 - 0.5*COS(2*PI*I%/M%) + 0.08*COS(4*PI*I%/M%))
420 NEXT I%
430 '
440 SUM = 0 'Normalize the second low-pass filter kernel for
450 FOR I% = 0 TO 800 'unity gain at DC
460 SUM = SUM + B[I%]
470 NEXT I%
480 '
490 FOR I% = 0 TO 800
500 B[I%] = B[I%] / SUM
510 NEXT I%
520 '
530 FOR I% = 0 TO 800 'Change the low-pass filter kernel in B[ ] into a high-pass
540 B[I%] = - B[I%] 'filter kernel using spectral inversion (as in Fig. 14-5)
550 NEXT I%
560 B[400] = B[400] + 1
570 '
580 '
590 FOR I% = 0 TO 800 'Add the low-pass filter kernel in A[ ], to the high-pass
600 H[I%] = A[I%] + B[I%] 'filter kernel in B[ ], to form a band-reject filter kernel
610 NEXT I% 'stored in H[ ] (as in Fig. 14-8)
620 '
630 FOR I% = 0 TO 800 'Change the band-reject filter kernel into a band-pass
640 H[I%] = -H[I%] 'filter kernel by using spectral inversion
650 NEXT I%
660 H[400] = H[400] + 1
670 ' 'The band-pass filter kernel now resides in H[ ]
680 END
```

TABLE 16-2

Frequency
0 0.1 0.2 0.3 0.4 0.5
-200

-180
-160
-140
-120
-100
-80
-60
-40
-20
0
20
40
60
a. Incredible stopband attenuation
round-off noise
single precision
!
FIGURE 16-7
The incredible performance of the windowed-sinc filter. Figure (a) shows the frequency response of a windowed-sinc filter with increased stopband attenuation. This is achieved by convolving a windowed-sinc filter kernel with itself. Figure (b) shows the very rapid roll-off a 32,001 point windowed-sinc filter.
Frequency
0.1995 0.2 0.2005
0.0
0.5
1.0
1.5
b. Incredible roll-off !
Amplitude (dB)
Amplitude

round-off noise on signals in the *passband* can erratically appear in the *stopband* with amplitudes in the -100dB to -120dB range).

Figure 16-7b shows another example of the windowed-sinc's incredible performance: a low-pass filter with 32,001 points in the kernel. The frequency response appears as expected, with a roll-off of 0.000125 of the sampling rate. How good is this filter? Try building an analog electronic filter that passes signals from DC to 1000 hertz with less than a 0.02% variation, and blocks all frequencies above 1001 hertz with less than 0.02% residue. Now that's a filter! If you really want to be impressed, remember that both the filters in Fig. 16-7 use *single precision*. Using *double precision* allows these performance levels to be extended by a *million* times.

The strongest limitation of the windowed-sinc filter is the *execution time*; it can be unacceptably long if there are many points in the filter kernel and standard convolution is used. A high-speed algorithm for this filter (FFT convolution) is presented in Chapter 18. Recursive filters (Chapter 19) also provide good frequency separation and are a reasonable alternative to the windowed-sinc filter.

Is the windowed-sinc the optimal filter kernel for separating frequencies? No, filter kernels resulting from more sophisticated techniques can be better. But beware! Before you jump into this very mathematical field, you should consider exactly what you hope to gain. The windowed-sinc will provide *any* level of performance that you could possibly need. What the advanced filter design methods may provide is a slightly shorter filter kernel for a given level of performance. This, in turn, may mean a slightly faster execution speed. Be warned that you may get little return for the effort expended.