Digital Signal Processing

We have already used RC filters to low-pass-filter high-frequency PWM signals, creating analog output signals. Filters have many other uses: for example, suppressing high-frequency or 60 Hz electrical noise, extracting high-frequency components from change-sensitive sensors, and integrating or differentiating a signal. If the signal is an analog voltage, these filters can be implemented with resistors, capacitors, and op amps (Appendix B).

Filters can also be implemented in software. In this case, the signal is first converted to digital form, perhaps using an analog-to-digital converter to sample an analog signal at fixed time increments. Once in this form, a *digital filter* can be used to difference or integrate the signal, or to suppress, enhance, or extract different frequency components in the signal. Digital filters offer advantages over their analog electronic counterparts:

- No need for extra external components, such as resistors, capacitors, and op amps.
- Tremendous flexibility in the filter design. Filters with excellent properties can be implemented very easily in software.
- The ability to operate on signals that do not originate from analog voltages.

Digital filtering is one example of *digital signal processing* (DSP). We start this chapter by providing some background on sampled signal representation. We then provide an introduction to the *fast Fourier transform* (FFT), which can be used to decompose a digital signal into its frequency components. The FFT is among the most important and heavily used algorithms in video, audio, and many other signal processing and control applications. We then discuss a class of digital filters called *finite impulse response* (FIR) filters, which calculate their output values as weighted sums of their past input samples. Next we briefly describe *infinite impulse response* (IIR) filters, which calculate their output as weighted sums of their past inputs and outputs. We also discuss FFT-based filters. Finally we conclude with sample code for DSP on a PIC32.

This chapter provides a brief introduction and some practical hints on how to use FFTs, FIR filters, and IIR filters. We skip most of the mathematical underpinnings, which are covered in books and courses focusing solely on signal processing.

22.1 Sampled Signals and Aliasing

Let $x(t)$ be a periodic signal that varies as a function of (continuous) time with period T ($x(t) = x(t+T)$), and therefore frequency $f = 1/T$. A periodic signal $x(T)$ can be written as the sum of a DC (constant) component and an infinite sequence of sinusoids at frequencies $f, 2f, 3f$, etc.:

$$x(t) = A_0 + \sum_{m=1}^{\infty} A_m \sin(2\pi mft + \phi_m). \tag{22.1}$$

Thus the T-periodic signal $x(t)$ can be uniquely represented by the amplitudes $A_0, A_1, \ldots$ and the phases $\phi_1, \phi_2, \ldots$ of the component sinusoids. These amplitudes and phases form the *frequency domain* representation of $x(t)$.

An example is a square wave signal that swings between $+1$ and -1 at frequency f and 50% duty cycle. The Fourier series that creates this signal is given by $A_m = 0$ for even m and $A_m = 4/(m\pi)$ for odd m, with all phases $\phi_m = 0$. Figure 22.1 illustrates the first four components of the square wave.

The first step in analyzing an analog signal using DSP is to sample the continuous-time signal $x(t)$ at time intervals T_s (sampling frequency $f_s = 1/T_s$), yielding N samples $x(n) \equiv x(nT_s) = x(t)$ for $n = 0, 1, 2, \ldots, N-1$, as shown in Figure 22.2. The sampling process also quantizes the signal; for example, the PIC32's 10-bit ADC module converts a continuous voltage to one of 1024 levels. While quantization is an important consideration in DSP, in this chapter we ignore quantization effects and assume that $x(n)$ can take arbitrary real values.

Suppose the original analog input signal is a sinusoid

$$x(t) = A\sin(2\pi ft + \phi),$$

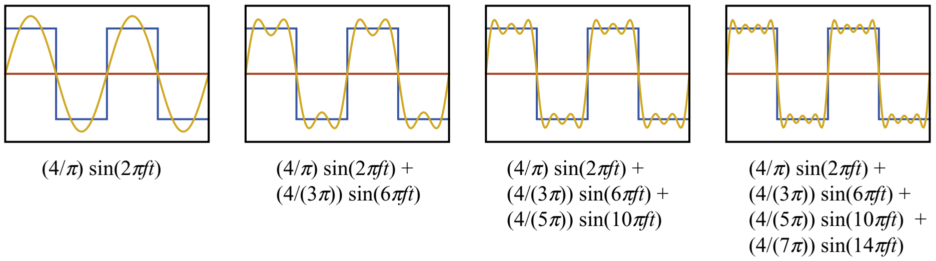

Figure 22.1
An illustration of the sum of the first four nonzero frequency components of the Fourier series of a square wave. The sum converges to the square wave as higher frequency components are included in the sum.

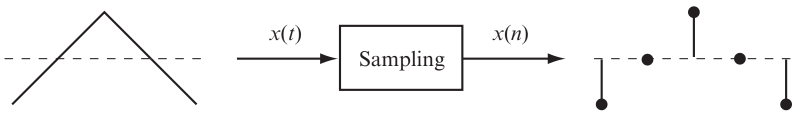

Figure 22.2
The sampling module converts the continuous-time signal $x(t)$ to a discrete-time signal $x(n)$.

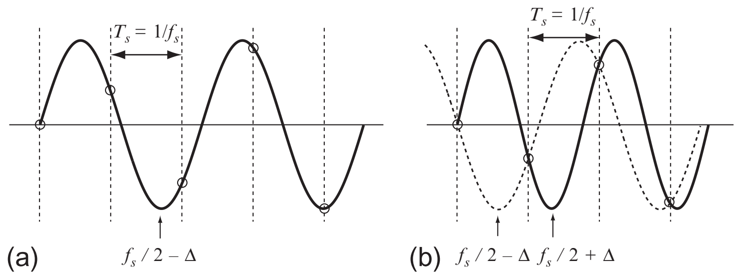

Figure 22.3
(a) The underlying sinusoid $x(t)$ with frequency $f = f_s/2 - \Delta$, $\Delta > 0$, can be reconstructed from its samples $x(n)$, shown as circles. (b) An input sinusoid of frequency $f = f_s/2 + \Delta$, however, appears to be a signal of frequency $f = f_s/2 - \Delta$ with a different phase.

where f is the frequency, $T = 1/f$ is the period, A is the amplitude, and ϕ is the phase. Given samples $x(n)$ taken at the sampling frequency f_s, and knowing the input is a sinusoid, it is possible to use the samples to uniquely determine A, f, and ϕ of the underlying signal, provided f is less than $f_s/2$. As we increase the signal frequency f beyond $f_s/2$, however, something interesting happens, as illustrated in Figure 22.3. The samples of a signal with frequency $f_1 = f_s/2 + \Delta$, $\Delta > 0$, with phase ϕ_1, are indistinguishable from the samples of a signal with lower frequency $f_2 = f_s/2 - \Delta$ with a different phase ϕ_2. For example, for $\Delta = f_s/2$, an input signal of frequency $f_1 = f_s/2 + \Delta = f_s$ looks the same as a constant (DC) input signal ($f_2 = f_s/2 - \Delta = 0$), because our once-per-cycle sampling returns the same value each time.

The phenomenon of signals of frequency greater than $f_s/2$ "posing" as signals of frequency between 0 and $f_s/2$ is called *aliasing*. The frequency $f_s/2$, the highest frequency we can uniquely represent with a discrete-time signal, is known as the *Nyquist frequency* f_N. Because we cannot distinguish higher-frequency signals from lower-frequency signals, we assume that all input frequencies are in the range $[0, f_N]$. If the sampled signal is obtained from an analog voltage, it is common to put an analog electronic low-pass *anti-aliasing* filter before the sampler to remove frequency components greater than f_N.

Aliasing is familiar from watching a low-frame-rate video of a spinning wheel. Your eyes track a mark on the wheel as it speeds up at a constant rate, and initially you see the wheel spinning forward faster and faster. In other words, the wheel appears to have an increasingly positive rotational frequency. As the wheel continues to accelerate, just after the point where the video camera captures only two images per revolution, the wheel begins to appear to be rotating backwards at a high speed (rotating with a large negative rotational frequency). As its actual forward speed increases further, the apparent negative speed begins to slow (the negative rotational frequency grows toward zero), until eventually the wheel appears to be at rest again (zero frequency) when the camera takes exactly one image per revolution.

22.2 The Discrete Fourier Transform

To design digital filters, it is important to understand the frequency domain representation of a digital signal. This representation allows us to see the amount of signal present at different frequencies, and to assess the performance of filters designed to suppress signals or noise at unwanted frequencies.

To find the frequency domain representation of an N-sample signal $x(n)$, $n = 0, \dots, N-1$, we use the *discrete Fourier transform* (DFT) of x. As we will see, the DFT allows us to calculate the frequency domain representation of the N-periodic signal that repeats the same N samples infinitely. Assuming N is even, this representation is given by $N/2$ sinusoid phases ϕ_m, where $m = 1 \dots N/2$, and $N/2 + 1$ amplitudes: the DC amplitude A_0 and the sinusoidal amplitudes A_m. The frequencies of the sinusoids are $mf = mf_s/N$ in (22.1). The spacing between frequencies represented by the A_m and ϕ_m is $f_s/N = 1/(NT_s)$—the more samples N, the higher the frequency resolution.

The DFT $X(k), k = 0, \dots, N-1$, of $x(n), n = 0, \dots, N-1$, is given by

$$X(k) = \sum_{n=0}^{N-1} x(n)\mathrm{e}^{-j2\pi kn/N}, \quad k = 0, \dots, N-1. \tag{22.2}$$

Considering that $\mathrm{e}^{-j2\pi kn/N} = \cos(2\pi kn/N) - j\sin(2\pi kn/N)$, generally the $X(k)$ are complex numbers. Additionally, the form of (22.2) shows that $X(N/2 + \Delta)$, where $\Delta \in \{1, 2, \dots, N/2 - 1\}$, is the complex conjugate of $X(N/2 - \Delta)$. In particular, this means that their magnitudes are equal, $|X(N/2 - \Delta)| = |X(N/2 + \Delta)|$.

Without going into details, the normalized "frequency" k/N associated with the sinusoids in $X(k)$ represents the actual frequency kf_s/N. Thus $X(N/2)$ is associated with the Nyquist frequency $f_N = f_s/2$. Higher frequencies ($k > N/2$) are equivalent to negative frequencies $(k - N)f_s/N$, as described in the spinning wheel aliasing analogy.

The $X(k)$ contain all the information we need to find the A_m and ϕ_m frequency domain representation of the sampled signal x. For a given $X(k) = a + bj$, the magnitude

$|X(k)| = \sqrt{a^2 + b^2}$ corresponds to N times the magnitude of the frequency component at $f_s k/N$, and the phase is given by the angle of $X(k)$ in the complex plane, i.e., the two-argument arctangent atan2(b, a). In this chapter we focus only on the amplitudes of the frequency components, ignoring the phase, since the phase is essentially random when the amplitude $\sqrt{a^2 + b^2}$ is near zero.

Because the $|X(k)|$ represent the magnitudes as a component at DC ($|X(0)|$), a component at the Nyquist frequency f_N ($|X(N/2)|$), and equal-magnitude complex conjugate pairs $X(k)$ for all other k, the A_m can be calculated as

$$A_0 = |X(0)|/N, \tag{22.3}$$

$$A_{N/2} = |X(N/2)|/N, \tag{22.4}$$

$$A_m = 2|X(m)|/N, \quad \text{for all } m = 1, \ldots, N/2 - 1, \tag{22.5}$$

where the frequency corresponding to A_m is mf_s/N.

22.2.1 The Fast Fourier Transform

The *Fast Fourier Transform* (FFT) refers to one of several methods for efficiently calculating the DFT. Many implementations of the FFT require that N be a power of two. If we have a number of samples that is not a power of two, we can simply "pad" the signal with "virtual" samples of value zero at the end. This process is called "zero padding." For example, if we have 1000 samples, we can pad the signal with 24 zeros to reach $2^{10} = 1024$ samples.

For example, assume an underlying analog signal

$$x(t) = 0.5 + \sin(2\pi(20\text{ Hz})t + \pi/4) + 0.5\sin(2\pi(200\text{ Hz})t + \pi/2)$$

with components at DC, 20 Hz, and 200 Hz. We collect 1000 samples at $f_s = 1$ kHz (0.001 s intervals) and zero pad to get $N = 1024$. The signal and its FFT magnitude plot is shown in Figure 22.4. The magnitude components are spaced at frequency intervals of f_s/N, or 0.9766 Hz. The DC, 20 Hz ($0.04\,f_N$), and 200 Hz ($0.4\,f_N$) components are clearly visible, though the numerical procedure has spread the components over several nearby frequencies since the actual signal frequencies are not represented exactly as mf_s/N for any integer m.

The mathematics of the DFT (and FFT) implicitly assume that the signal repeats every N samples. Thus the DFT may have a significant component at the lowest nonzero frequency, f_s/N, even if this frequency is not present in the original signal. This component suggests the following:

- If the original signal is known to be periodic with frequency f, the N samples should contain several complete cycles of the signal (as opposed to one or less than one cycle).

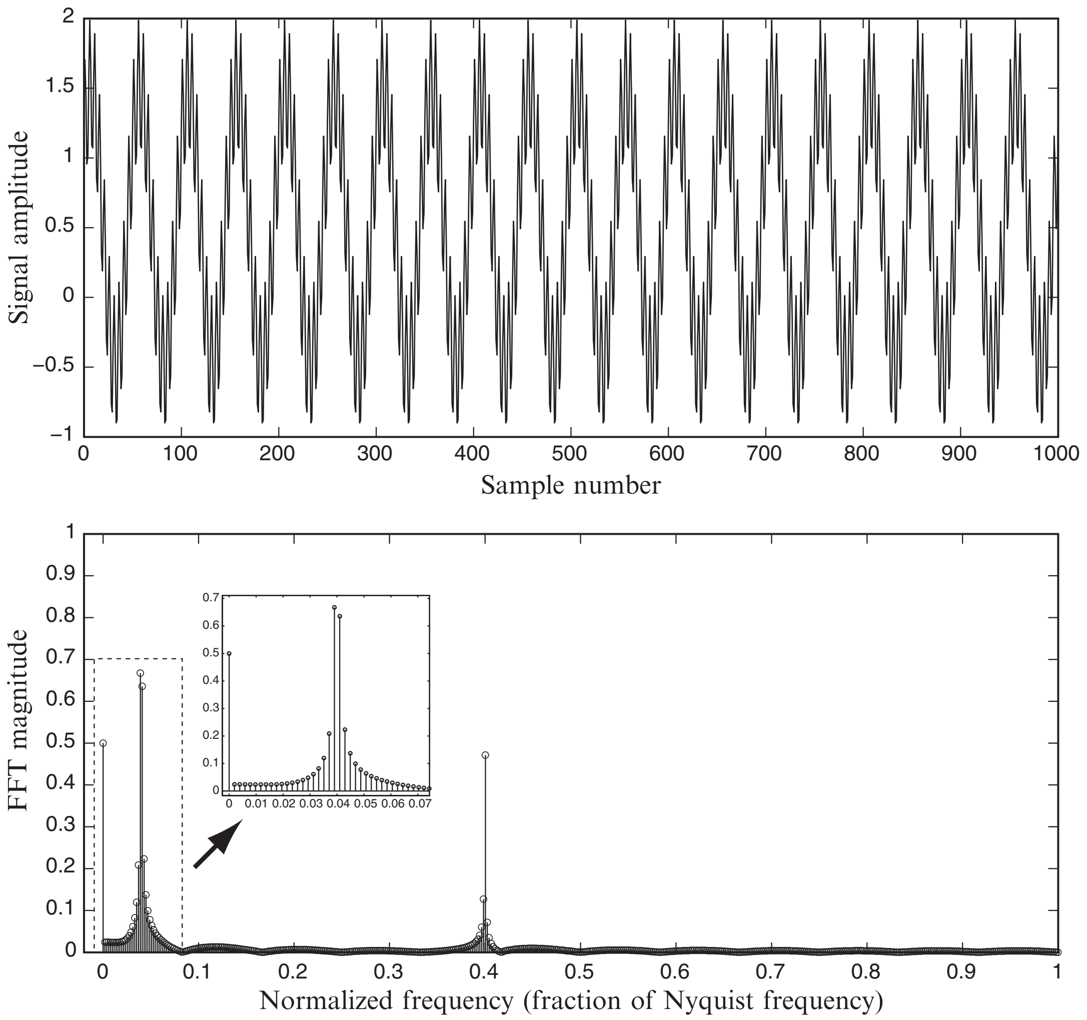

Figure 22.4
(Top) The original sampled signal. (Bottom) The FFT magnitude plot, with a portion of it magnified.

Having many cycles means that the smallest nonzero frequency represented, f_s/N, is much smaller than f, isolating the non-DC signals we care about (at f and higher) from the lower frequencies that appear due to the finite sampling.

- If the original signal is known to be periodic with frequency f, then, if possible, the samples should contain an integer number of cycles. In this case, there should be very little magnitude at the lowest nonzero frequency f_s/N.
- If the original signal is not periodic, zero padding can be used to isolate the lowest nonzero frequency component of the repeated signal from f_s/N.

22.2.2 The FFT in MATLAB

Given an even number of samples N in a row vector `x = [x(1) ... x(N)]` in MATLAB (note the index starts at 1 in MATLAB), the command

```
X = fft(x);
```

returns an N-vector `X = [X(1) ... X(N)]` of complex numbers corresponding to the amplitude and phase at different frequencies. Let us try an FFT of $N = 200$ samples of a 50% duty cycle square wave, where each period consists of 10 samples equal to 2 and 10 samples equal to 0 (i.e., the square wave of Figure 22.1 plus a DC offset of 1). The frequency of the square wave is $f_s/20$, and our entire sampled signal consists of 10 full cycles. According to Figure 22.1, the continuous-time square wave consists of frequency components at $f_s/20$, $3f_s/20$, $5f_s/20$, $7f_s/20$, etc. Thus we expect the frequency domain magnitude representation of the sampled square wave to consist of the DC component and nonzero components at these frequencies.

Let us build the signal and plot it (Figure 22.5(a)):

```
x=0;   % clear any array that might already be in x
x(1:10) = 2;
x(11:20)= 0;
x = [x x x x x x x x x x];
N = length(x);
plot(x,'Marker','o');
axis([-5 205 -0.1 2.1]);
```

Now let us plot the FFT amplitude plot, using the procedure described in Section 22.2:

```
plotFFT(x);
```

This code uses our custom MATLAB function `plotFFT`:

Code Sample 22.1 `plotFFT.m`. **Plotting the Single-Sided FFT Magnitude Plot of a Sampled Signal** `x` **with an Even Number of Samples.**

```
function plotFFT(x)

if mod(length(x),2) == 1           % x should have an even number of samples
  x = [x 0];                       % if not, pad with a zero
end
N = length(x);
X = fft(x);
mag(1) = abs(X(1))/N;              % DC component
mag(N/2+1) = abs(X(N/2+1))/N;      % Nyquist frequency component
mag(2:N/2) = 2*abs(X(2:N/2))/N;    % all other frequency components
freqs = linspace(0, 1, N/2+1);     % make x-axis as fraction of Nyquist freq
stem(freqs, mag);                  % plot the FFT magnitude plot
axis([-0.05 1.05 -0.1*max(mag) 1.1*max(mag)]);
xlabel('Frequency (as fraction of Nyquist frequency)');
```

```
    ylabel('Magnitude');
    title('Single-Sided FFT Magnitude');
    set(gca,'FontSize',18);
```

Figure 22.5 shows the original signal and the single-sided FFT magnitude plot. Notice that the FFT very clearly picks out the frequency components at DC, $0.1f_N, 0.3f_N, 0.5f_N, 0.7f_N$, and $0.9f_N$.

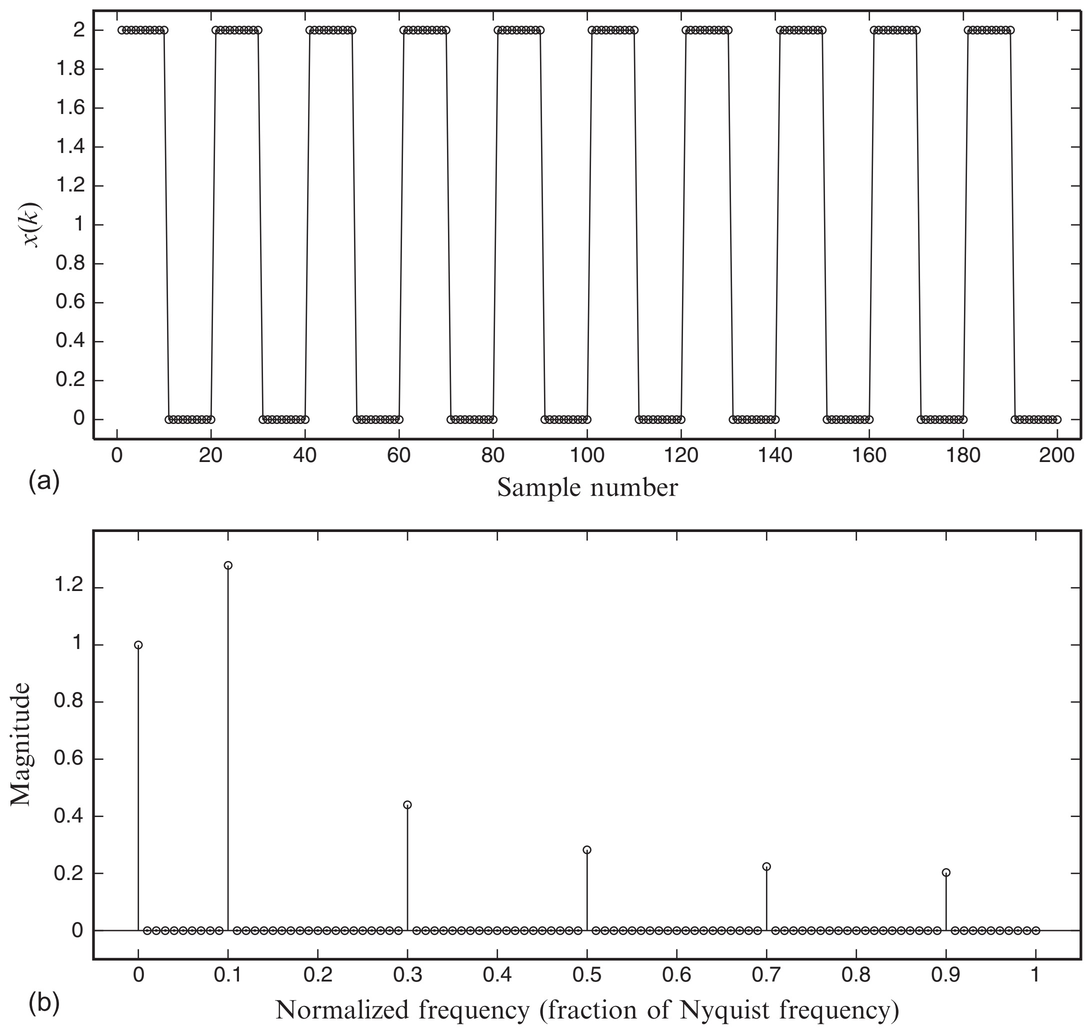

Figure 22.5
(a) The original sampled signal. (b) The single-sided FFT magnitude plot with frequencies expressed as a fraction of the Nyquist frequency.

FFT with $N = 2^n$

For efficiency reasons, the MIPS PIC32 DSP code only performs FFTs on sampled signals that have a power-of-2 length. Let us increase the number of samples with MATLAB from 200 to the next highest power of 2, $2^8 = 256$. We can either pad the original $x(k)$ with 56 zeros at the end, or we can take more samples.

Let us try the zero-padding option first:

```
x = 0;
x(1:10) = 2;
x(11:20)= 0;
x = [x x x x x x x x x x];       % get the signal samples
N = 2^nextpow2(length(x));       % compute the number of zeros to pad
xpad = [x zeros(1,N-length(x))]; % add the zero padding
plotFFT(xpad);
```

And now if the signal were sampled 256 times in the first place:

```
x = 0;
x(1:10) = 2;
x(11:20)= 0;
x = [x x x x x x x x x x x x 2*ones(1,10) zeros(1,6)];
plotFFT(x);
```

The results are plotted in Figure 22.6. The frequency components are still visible, though the results are not as clear as in Figure 22.5. A major reason for the lower quality plot is that the signal frequencies $0.1f_N, 0.3f_N$, etc., are not exactly represented in the FFT, as they were before. The frequency intervals are now $f_s/256$, not $f_s/200$. As a result, the FFT spreads the original frequency components across nearby frequencies rather than concentrating them in spikes at exact frequencies. This kind of spread is typical in most applications, as it is unlikely that the original signal will have component frequencies exactly at frequencies of the form mf_s/N.

22.2.3 The Inverse Fast Fourier Transform

Given the frequency domain representation $X(k)$ obtained from `X = fft(x)`, the inverse FFT uses the FFT algorithm to recover the original time-domain signal $x(n)$. In MATLAB, this is the procedure:

```
N = length(x);
X = fft(x);
xrecovered = fft(conj(X)/N);
plot(real(xrecovered));
```

The inverse FFT is accomplished by applying `fft` to the complex conjugate of the frequency representation `X` (the imaginary components of all entries are multiplied by -1), scaled by `1/N`.

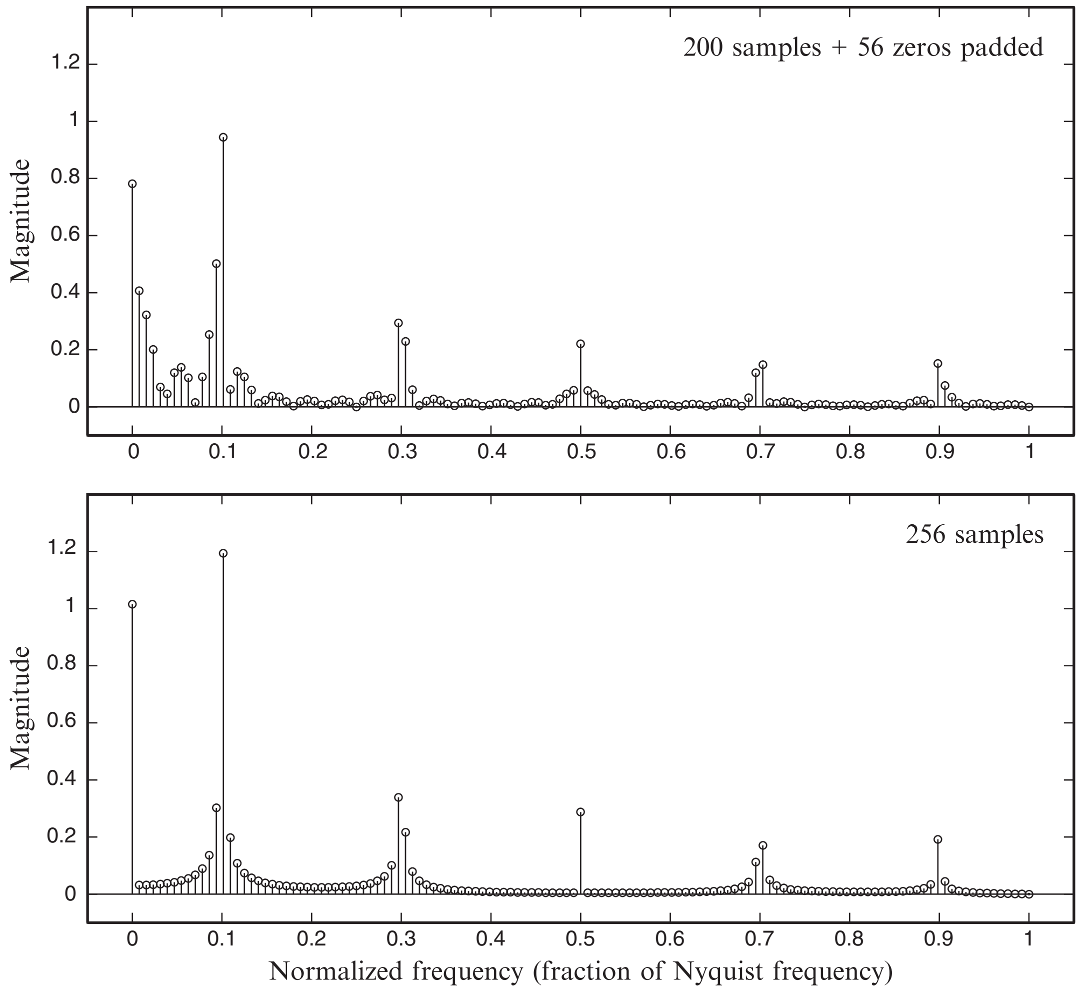

Figure 22.6
(Top) The FFT of the 200-sample square wave signal with 56 zeros padded. (Bottom) The FFT of the 256-sample square wave signal.

The vector `xrecovered` is equal to the original `x` up to numerical errors, so its entries have essentially zero imaginary components. The `real` operation returns only the real components, ensuring that the imaginary components are exactly zero.

22.3 Finite Impulse Response (FIR) Digital Filters

Now that we understand frequency domain representations of sampled signals, we turn our attention to filtering those signals (Figure 22.7). A finite impulse response (FIR) filter

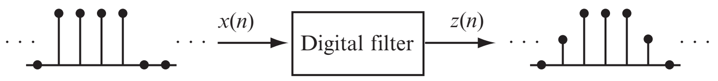

Figure 22.7
A digital filter produces filtered output $z(n)$ based on the inputs $x(n)$.

produces a filtered signal $z(n)$ by multiplying the $P+1$ current and past inputs $x(n-j), j = 0 \ldots P$, by *filter coefficients* b_j and adding:

$$z(n) = \sum_{j=0}^{P} b_j x(n-j).$$

Such filters can be used for several operations, such as differencing a signal or suppressing low-frequency or high-frequency components. For example, if $P = 1$ and we choose $b_0 = 1$ and $b_1 = -1$, then

$$z(n) = x(n) - x(n-1),$$

i.e., the output of the filter at time n is the difference between the input at time n and the input at time $n-1$. This differencing filter in discrete time is similar to a derivative filter in continuous time.

Since FIR filtering is a linear operation on the samples, filters in series can be performed in any order. For example, a differencing filter followed by a low-pass filter gives equivalent output to the low-pass filter followed by the differencing filter.

An FIR filter has $P+1$ coefficients, and P is called the *order* of the filter. The filter coefficients are directly evident in the *impulse response*, which is the response $z(n)$ to a unit impulse $\delta(n)$, where

$$\delta(n) = \begin{cases} 1 & \text{for } n = 0 \\ 0 & \text{otherwise.} \end{cases}$$

The output is simply $z(0) = b_0, z(1) = b_1, z(2) = b_2$, etc. The impulse response is typically written as $h(k)$.

Any input signal x can be represented as the sum of scaled and time-shifted impulses. For example,

$$x = 3\delta(n) - 2\delta(n-2)$$

is a signal that has value 3 at time $n = 0$ and -2 at time $n = 2$ (Figure 22.8(left)). Because an FIR filter is linear, the output is simply the sum of the scaled and time-shifted impulse responses,

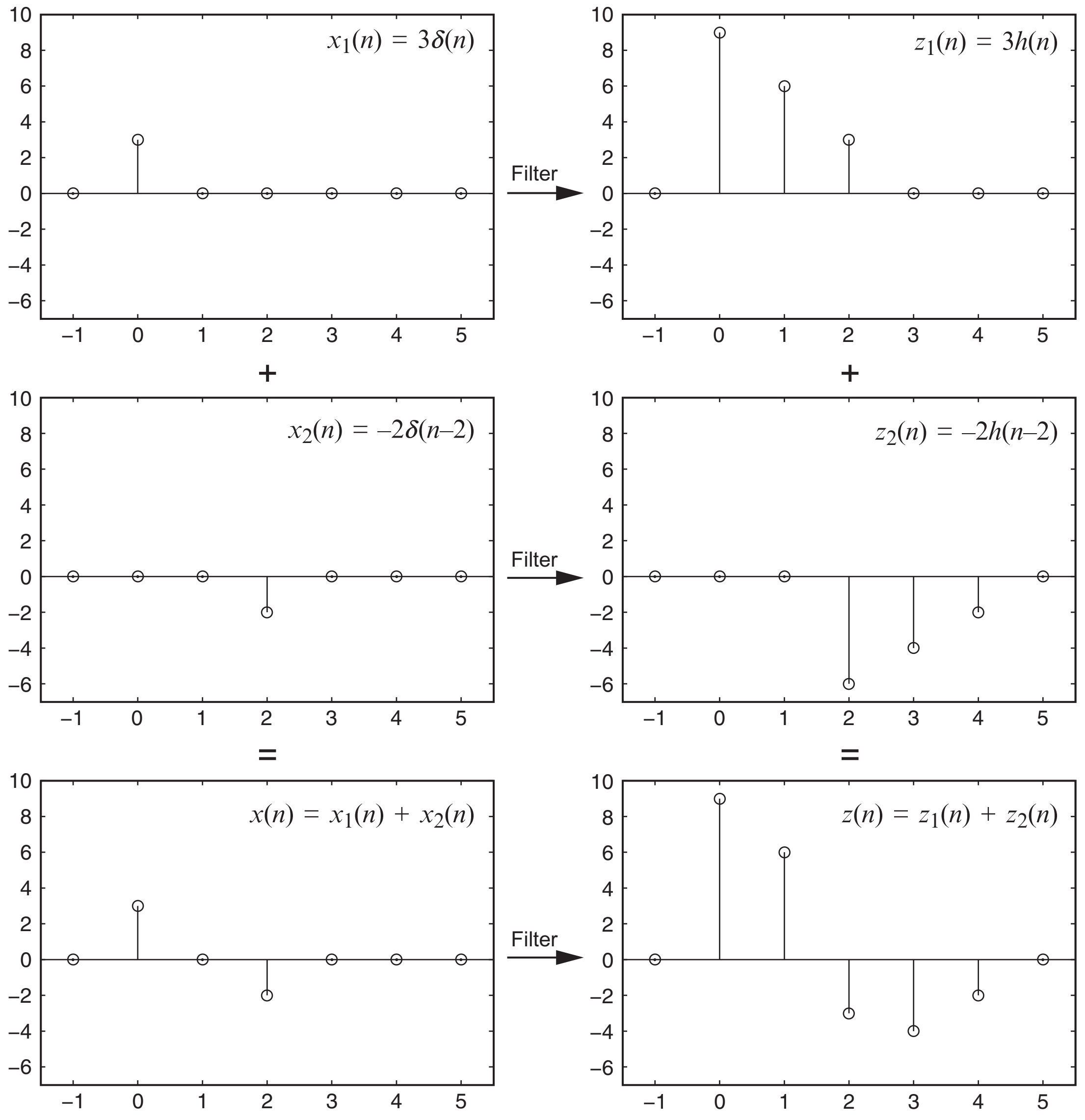

Figure 22.8
(Left) The two scaled and time-shifted impulses sum to give the signal *x* in time. (Left to right) The filter with coefficients $b_0 = 3, b_1 = 2, b_2 = 1$ applied to each of the individual and composite signals. (Right) The response *z* to the signal *x* can be obtained by summing the responses z_1 and z_2 to the individual components of *x*.

$$z = 3h(n) - 2h(n-2).$$

For the second-order filter with coefficients $b_0 = 3, b_1 = 2, b_2 = 1$, for example, the response to the input x is shown in Figure 22.8(right). When reading these signals, be aware that the leftmost samples are oldest; for example, the output $z(0)$ happens three timesteps before the output $z(3)$.

The output z is called the *convolution* of the input x and the filter's impulse response h, commonly written $z = x * h$. The convolution is obtained by simply summing the scaled and time-shifted impulse responses corresponding to each sample $x(n)$, as illustrated in Figure 22.8.

An FIR filter response can be calculated using MATLAB's `conv` command. We collect the filter coefficients into the impulse response vector `h = b = [b0 b1 b2] = [3 2 1]` and the input into the vector `x = [3 0 -2]`, and then

```
z = conv(h,x)
```

produces `z = [9 6 -3 -4 -2]`.

"Finite Impulse Response" filters get their name from the fact that the impulse response goes to zero in finite time (i.e., there is a finite number of filter coefficients). As a result, for any input that goes to zero eventually, the response goes to zero eventually. The output of an "Infinite Impulse Response" filter (Section 22.4) may *never* go to zero.

A filter is fully characterized by its impulse response. Often it is more convenient to look at the filter's *frequency response*, however. Because the filter is linear, a sinusoidal input will produce a sinusoidal output, and the filter's frequency response consists of its frequency-dependent gain (the ratio of the filter's output sinusoid amplitude to the input amplitude) and phase (the shift in the phase of the output sinusoid relative to the input sinusoid).[1] To begin to understand the discrete-time frequency response, we start with the simplest of FIR filters: the moving average filter.

22.3.1 Moving Average Filter

Suppose we have a sensor signal $x(n)$ that has been corrupted by high-frequency noise (Figure 22.9). We would like to find the low-frequency signal underneath the noise.

The simplest filter to try is a *moving average filter* (MAF). A moving average filter calculates the output $z(n)$ as a running average of the input signals $x(n)$,

$$z(n) = \frac{1}{P+1} \sum_{j=0}^{P} x(n-j), \tag{22.6}$$

[1] See Appendix B.2.2 for related information on frequency response for analog circuits.

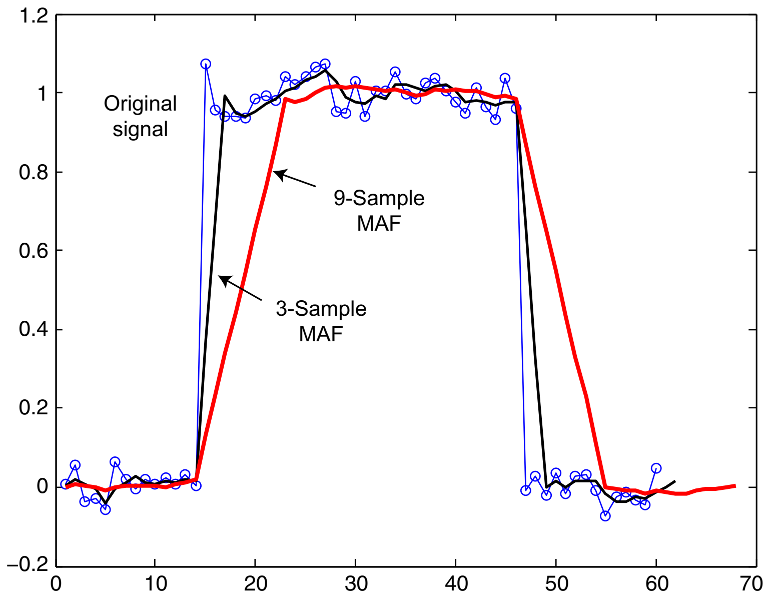

Figure 22.9
The original noisy signal with samples $x(n)$ given by the circles, the signal $z(n)$ resulting from filtering with a three-point MAF ($P = 2$), and the signal $z(n)$ from a nine-point MAF ($P = 8$). The signal gets smoother and more delayed as the number of samples in the MAF increases.

i.e., the FIR filter coefficients are $b_0 = b_1 = \cdots = b_P = 1/(P+1)$. The output $z(n)$ is a smoothed and delayed version of $x(n)$. The more samples $P+1$ we average over, the smoother and more delayed the output. The delay occurs because the output $z(n)$ is a function of only the current and previous inputs $x(n-j), 0 \leq j \leq P$ (see Figure 22.9).

To find the frequency response of a third-order, four-sample MAF, we test it on some sinusoidal inputs at different frequencies (Figure 22.10). We find that the phase ϕ of the (reconstructed) output sinusoid relative to the input sinusoid, and the ratio G of the amplitude of their amplitudes, depend on the frequency. For the four test frequencies in Figure 22.10, we get the following table:

Frequency	**Gain G**	**Phase ϕ**
$0.25f_N$	0.65	$-67.5°$
$0.5f_N$	0	NA
$0.67f_N$	0.25	$0°$
f_N	0	NA

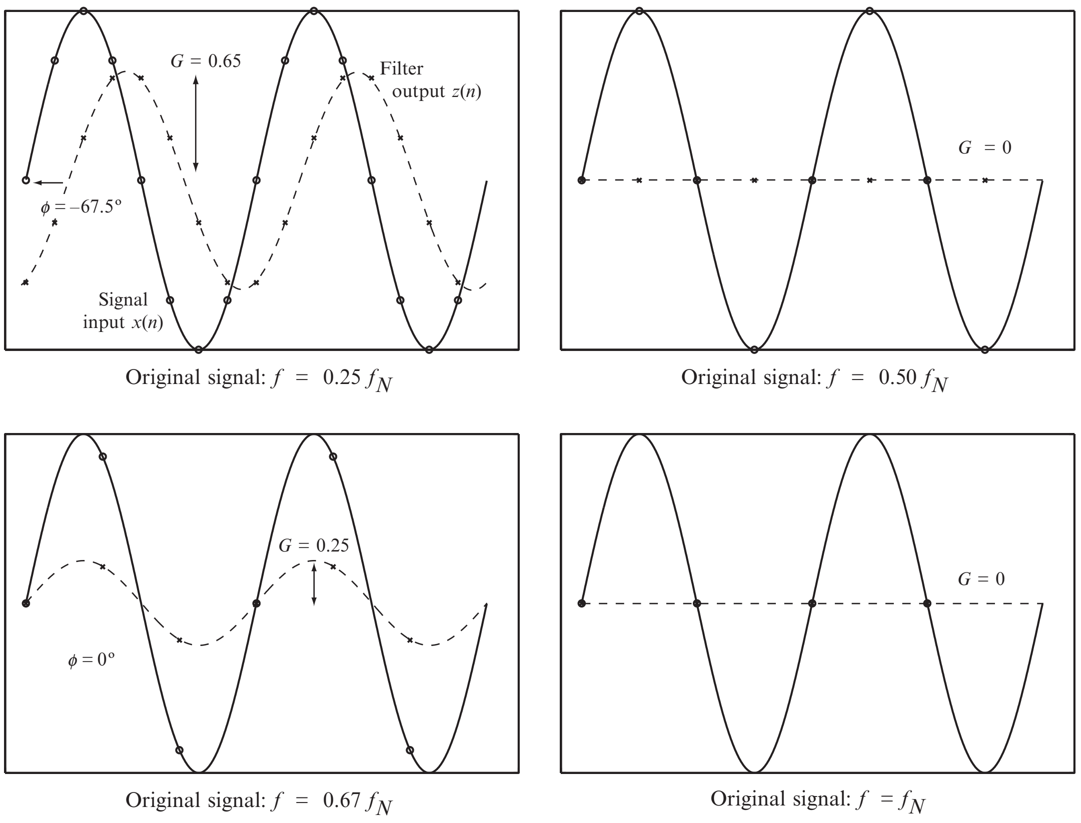

Figure 22.10
Testing the frequency response of a four-sample MAF with different input frequencies.

Testing the response at many different frequencies, we can plot the frequency response in Figure 22.11. Two things to note about the gain plot:

- Gains are usually plotted on a log scale. This allows representation of a much wider range of gains.
- Gains are often expressed in decibels, which are related to gains by the following relationship:

$$M_{\mathrm{dB}} = 20 \log_{10} G.$$

So $G = 1$ corresponds to 0 dB, $G = 0.1$ corresponds to -20 dB, and $G = 0.01$ corresponds to -40 dB.

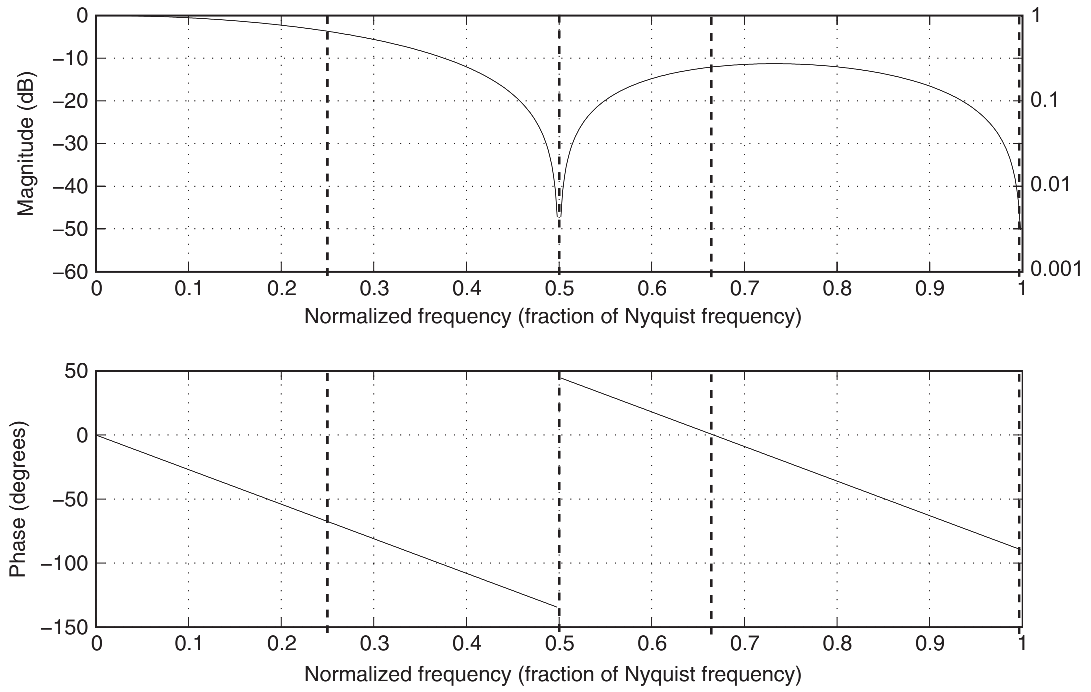

Figure 22.11
The frequency response of a four-sample MAF. Test frequencies are shown as dotted lines.

Examining Figure 22.11 shows that low frequencies are passed with a gain of $G = 1$ and no phase shift. The gain drops monotonically as the frequency increases, until it reaches $G = 0$ ($-\infty$ dB) at input frequencies $f = 0.5f_N$. (The plot truncates the dip to $-\infty$.) The gain then begins to rise again, before falling once more to $G = 0$ at $f = f_N$. The MAF behaves somewhat like a low-pass filter, but not a very good one; high frequency signals can get through with gains of 0.25 or more. Still, it works reasonably well as a signal smoother for input frequencies below $0.5f_N$.

Given a set of filter coefficients `b = [b0 b1 b2 ...]`, we can plot the frequency response in MATLAB using

```
  freqz(b);
```

Causal vs. acausal filters

A filter is called *causal* if its output is the result of only current and past inputs, i.e., past inputs "cause" the current output. Causal filters are the only option for real-time implementation. If we are post-processing data, however, we can choose an *acausal* version of the filter, where the outputs at time n are a function of past as well as future inputs. Such acausal filters can eliminate the delay associated with only using past inputs to calculate the current value. For example, a five-sample MAF which calculates the average of the past two inputs, the current input, and the next two inputs is acausal.

Zero padding

When a filter is first initialized, there are no past inputs. In this case we can assume the nonexistent past inputs were all zero. The output transient caused by this assumption will end at the $(P + 1)$th input.

22.3.2 FIR Filters Generally

FIR filters can be used for low-pass filtering, high-pass filtering, bandpass filtering, bandstop (notch) filtering, and other designs. MATLAB provides many useful functions for filter design, such as `fir1`, `fdatool`, and `design`. In this section we work with `fir1`. See the MATLAB documentation for more details.

A "good" filter is one that

- passes the frequencies we want to keep with gain close to 1,
- highly attenuates the frequencies we do not want, and
- provides a sharp transition in gain between the passed and attenuated frequencies.

The number of filter coefficients increases with the sharpness of the desired transition and the degree of attenuation needed in the stopped frequencies. This relationship is a general principle: the sharper the transitions in the frequency domain, the smoother and longer the impulse response (i.e., more coefficients are needed in the filter). The converse is also true: the sharper the transition in the impulse response, the smoother the frequency response. We saw this phenomenon with the moving average filter. It has a sharp transition between filter coefficients of 0 and $1/(P + 1)$, and the resulting frequency response has only slow transitions.

High-order filters are fine for post-processing data or for non-time-critical applications, but they may be inappropriate for real-time control because of unacceptable delay.

The MATLAB filter design function `fir1` takes the order of the filter, the frequencies you would like to pass or stop (expressed as a fraction of the Nyquist frequency), and other options, and returns a list of filter coefficients. MATLAB considers the cutoff frequency to be where the gain is 0.5 (−6 dB). Here are some examples using `fir1`:

```
b = fir1(10,0.2);               % 10th-order, 11-sample LPF with cutoff freq
                                  of 0.2 fN
b = fir1(10,0.2,'high');        % HPF cutting off frequencies below 0.2 fN
b = fir1(150,[0.1 0.2]);        % 150th-order bandpass filter with passband 0.1
                                  to 0.2 fN
b = fir1(50,[0.1 0.2],'stop');  % 50th-order bandstop filter with notch at 0.1 to
                                  0.2 fN
```

You can then plot the frequency response of your designed filter using `freqz(b)`.

If the order of your specified filter is not high enough, you will not be able to meet your design criteria. For example, if you want a low-pass filter that cuts off frequencies at 0.1 f_N, and you only allow seven coefficients (sixth order),

```
b = fir1(6,0.1);
```

you will find that the filter coefficients that MATLAB returns do not achieve your aims.

Examples

In the examples in Figures 22.12–22.19, we work with a 1000-sample signal, with components at DC, 0.004f_N, 0.04f_N, and 0.8f_N. The original signal `x` is plotted in Figure 22.12.

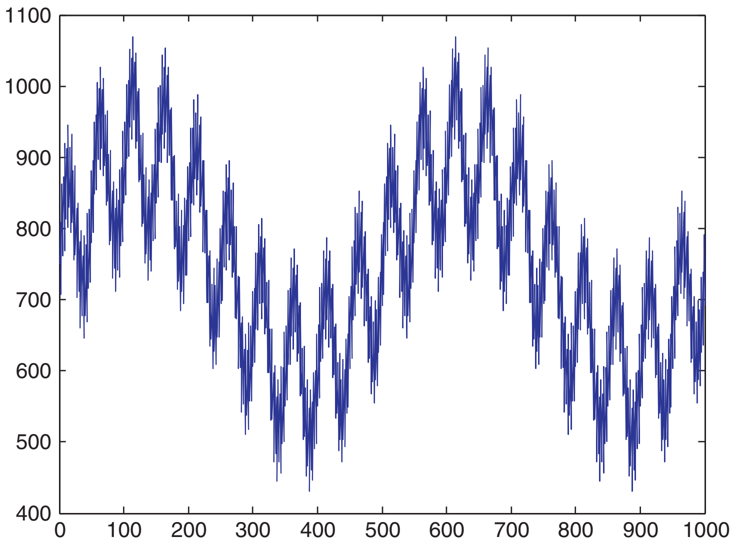

Figure 22.12
The original 1000-sample signal `x`.

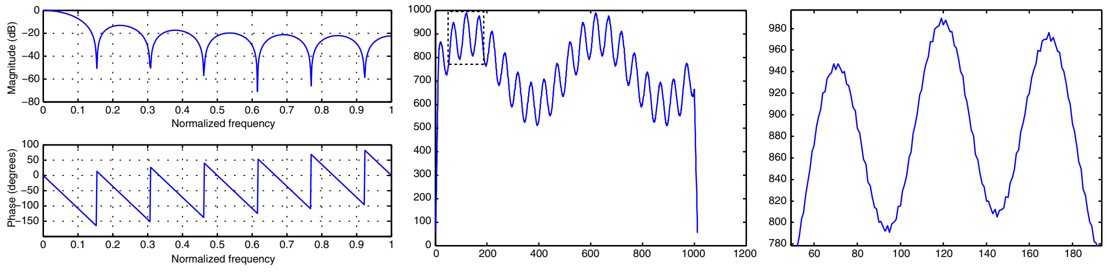

Figure 22.13

Moving average filter: `maf=ones(13,1)/13; freqz(maf); plot(conv(maf,x))`. **Left**: The frequency response of the 12th-order (13-sample) MAF. **Middle**: The result of the MAF applied to (convolved with) the original signal. Since the original signal has 1000 samples, and the MAF has 13 samples, the filtered signal has 1012 samples. (In general, if two signals of length j and k are convolved with each other, the result will have length $j + k - 1$.) This is equivalent to first "padding" the 1000 samples with 12 samples equal to zero on either end (sample numbers -11 to 0, and 1001 to 1012), then applying the 13-sample filter 1012 times, over samples -11 to 1, then -10 to 2, etc., up to samples 1000-1012. This zero-padding explains why the signal drops to close to zero at either end. **Right**: Zoomed in on the smoothed signal.

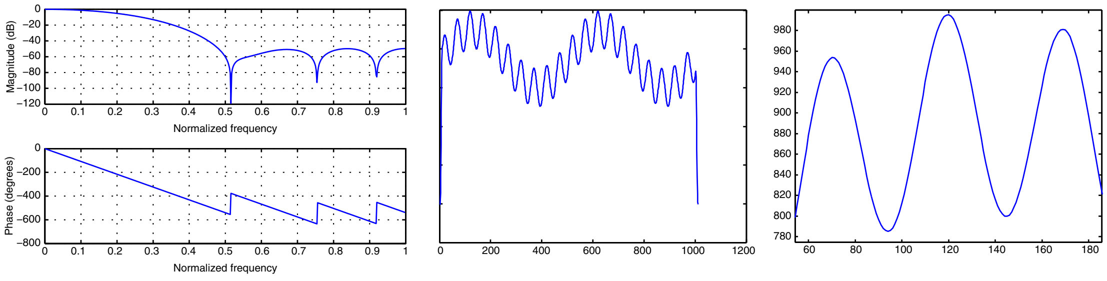

Figure 22.14

`lpf=fir1(12,0.2); freqz(lpf); plot(conv(lpf,x))`. **Left**: The frequency response of a 12th-order LPF with cutoff at $0.2f_N$. **Middle**: The signal smoothed by the LPF. **Right**: A zoomed-in view, showing less high-frequency content than the 12th-order MAF.

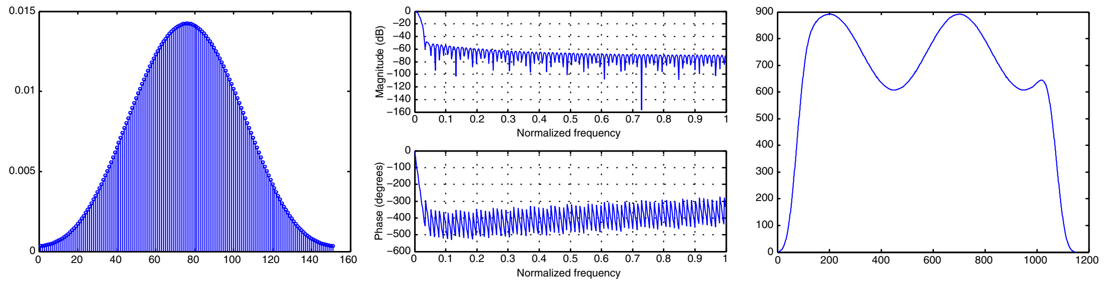

Figure 22.15

`lpf=fir1(150,0.01)`. **Left**: `stem(lpf)`. The coefficients of a 150th-order FIR LPF with a cutoff at $0.01f_N$. **Middle**: `freqz(lpf)`. The frequency response. **Right**: `plot(conv(lpf,x))`. The smoothed signal, where only the $0.004f_N$ and DC components get through.

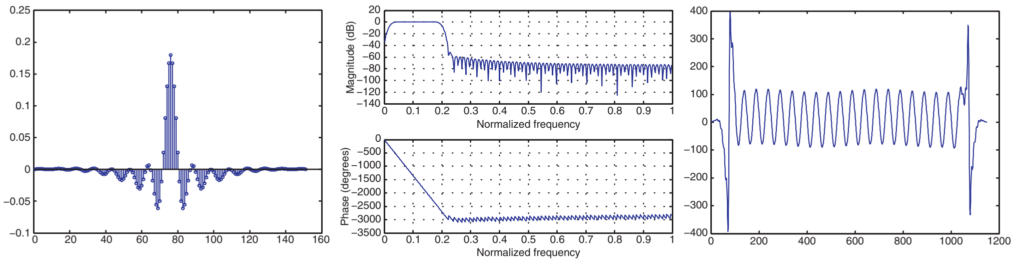

Figure 22.16

`bpf=fir1(150,[0.02,0.2])`. **Left**: `stem(bpf)`. The coefficients of a 150th-order bandpass filter with a passband from $0.02f_N$ to $0.2f_N$. **Middle**: `freqz(bpf)`. The frequency response. **Right**: `plot(conv(bpf,x))`. The signal consisting mostly of the $0.04f_N$ component, with small DC and $0.004f_N$ components.

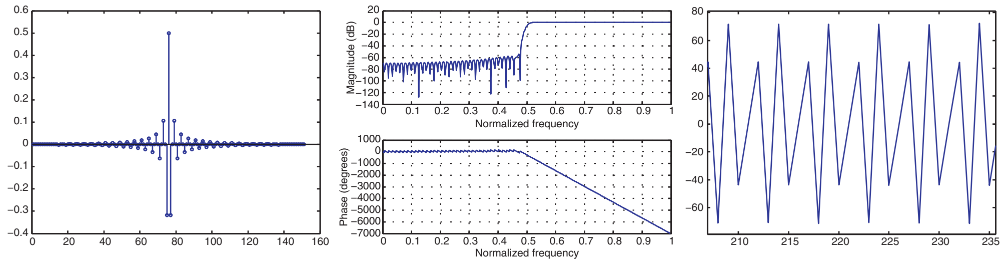

Figure 22.17

`hpf=fir1(150,0.5,'high')`. **Left**: `stem(hpf)`. The coefficients of a 150th-order high-pass filter with frequencies below $0.5f_N$ cut off. **Middle**: `freqz(hpf)`. The frequency response. **Right**: `plot(conv(hpf,x))`. Zoomed in on the high-passed signal.

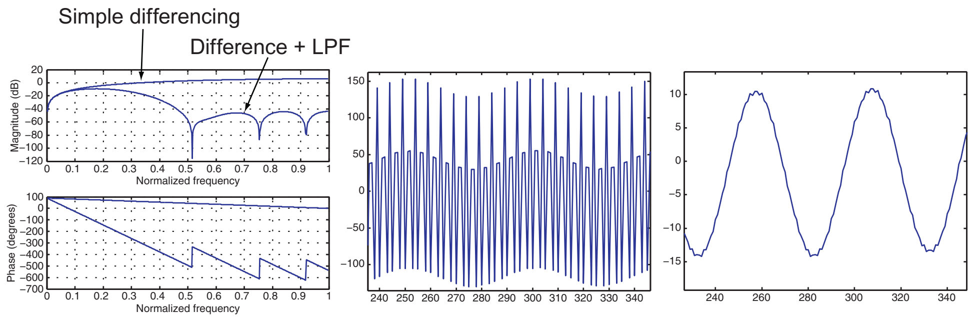

Figure 22.18

A simple differencing (or "velocity") filter has coefficients $b[0] = 1, b[1] = -1$, or written in MATLAB, `b = [1 -1]`. (Note the order: the coefficient that goes with the most recent input is on the left.) A differencing filter responds more strongly to signals with larger slopes (i.e., higher frequency signals) and has zero response to constant (DC) signals. Usually the signal "velocities" we are interested in, though, are those at low frequency; higher-frequency signals tend to come from sensing noise. Thus a better filter is probably a differencing filter convolved with a low-pass filter. **Left**: `b1 = [1 -1]; b2 = conv(b1,fir1(12,0.2)); freqz(b1); hold on; freqz(b2)`. This plot shows the frequency response of the differencing filter, as well as a differencing filter convolved with a 12th-order FIR LPF with cutoff at $0.2f_N$. At low frequencies, where the signals we are interested in live, the two filters have the same response. At high frequencies, the simple differencing filter has a large (unwanted) response, while the other filter attenuates this noise. **Middle**: `plot(conv(b1,x))`. Zoomed in on the signal filtered by the simple difference filter. **Right**: `plot(conv(b2,x))`. Zoomed in on the signal filtered by the difference-plus-LPF.

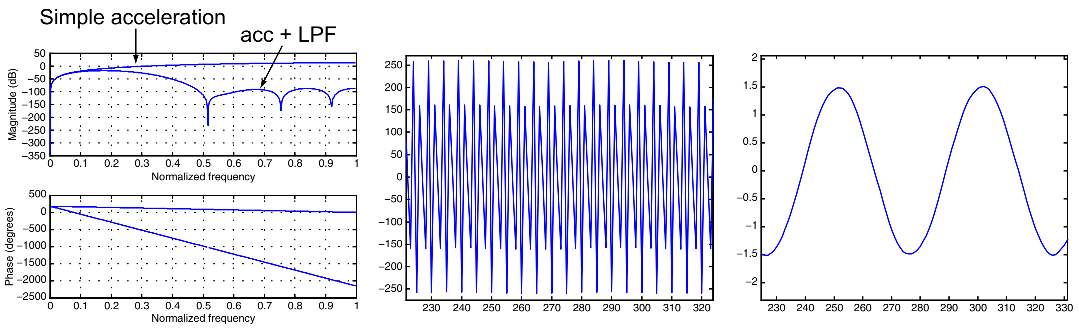

Figure 22.19
We can also make a double-differencing (or "acceleration") filter by taking the difference of two consecutive difference samples, i.e., convolving two differencing filters. This gives a simple filter with coefficients `[1 -2 1]`. This filter amplifies high frequency noise even more than a differencing filter. A better choice would be to use a filter that is the convolution of two difference-plus-low-pass filters from the previous example. **Left**: `bvel=[1 -1]; bacc=conv(bvel,bvel); bvellpf=conv(bvel,fir1(12,0.2)); bacclpf=conv(bvellpf,bvellpf); freqz(bacc); hold on; freqz(bacclpf)`. The low-frequency response of the two filters is identical, while the low-pass version attenuates high frequency noise. **Middle**: `plot(conv(bacc,x))`. Zoomed in on the second derivative of the signal, according to the simple acceleration filter. **Right**: `plot(conv(bacclpf,x))`. Zoomed in on the second derivative of the signal, according to the low-passed version of the acceleration filter.

22.4 Infinite Impulse Response (IIR) Digital Filters

The class of infinite impulse response (IIR) filters generalizes FIR filters to the following form:

$$\sum_{i=0}^{Q} a_i z(n-i) = \sum_{j=0}^{P} b_j x(n-j),$$

or, written in a more useful form for us,

$$z(n) = \frac{1}{a_0}\left(\sum_{j=0}^{P} b_j x(n-j) - \sum_{i=1}^{Q} a_i z(n-i)\right), \tag{22.7}$$

where the output $z(n)$ is a weighted sum of Q past outputs, P past inputs, and the current input. Some differences between FIR and IIR filters are highlighted below:

- IIR filters may be *unstable*, that is, their output may persist indefinitely and grow without bound even if the input is bounded in value and duration. Instability is impossible with FIR filters.
- IIR filters often use fewer coefficients to achieve the same magnitude response transition sharpness. Hence they can be more computationally efficient than FIR filters.

- IIR filters generally have a nonlinear phase response (phase does not change linearly with frequency, as with most FIR filters). This nonlinearity may or may not be acceptable, depending on the application. A linear phase response ensures that the time (not phase) delay associated with signals at all frequencies is the same. A nonlinear phase response, on the other hand, may cause different time delays at different frequencies, which may result in unacceptable distortion (e.g., in an audio application).

Because of roundoff errors in computation, an IIR filter that is theoretically stable may be unstable when implemented directly in the form of (22.7). Because of the possibility of instability, IIR filters with many coefficients are usually implemented as a cascade of filters with $P = 2$ and $Q = 2$. It is relatively easy to ensure that these low-order filters are stable, ensuring the stability of the cascade of filters.

Popular IIR digital filters include Chebyshev and Butterworth filters, which include low-pass, high-pass, bandpass, and bandstop versions. MATLAB offers design tools for these filters; you can refer to the documentation on `cheby1`, `cheby2`, and `butter`. Given a set of coefficients `b` and `a` defining the IIR filter, the MATLAB command `freqz(b,a)` plots the frequency response and `filter(b,a,signal)` returns the filtered version of `signal`.

One of the simplest IIR filters is the integrator

```
z(n) = z(n-1) + x(n)*Ts
```

where `Ts` is the sample time. The coefficients are `a = [1 -1]` and `b = [Ts]`. The behavior of the integrator on the sample signal in Figure 22.12 is shown in Figure 22.20.

22.5 FFT-Based Filters

Another option for filtering signals is to first FFT the signal, then set certain frequency components of the signal to zero, then invert the FFT. Assume we are working with the 256-sample square wave we looked at in Section 22.2.2, and we want to extract only the component at $0.1f_N$. First we build the signal and FFT it:

```
x = 0;
x(1:10) = 2;
x(11:20)= 0;
x = [x x x x x x x x x x x x 2*ones(1,10) zeros(1,6)];
N = length(x);
X = fft(x);
```

The element `X(1)` is the DC component, `X(2)` and `X(256)` correspond to frequency f_s/N, `X(3)` and `X(255)` correspond to frequency $2f_s/N$, `X(4)` and `X(254)` correspond to frequency $3f_s/N$,

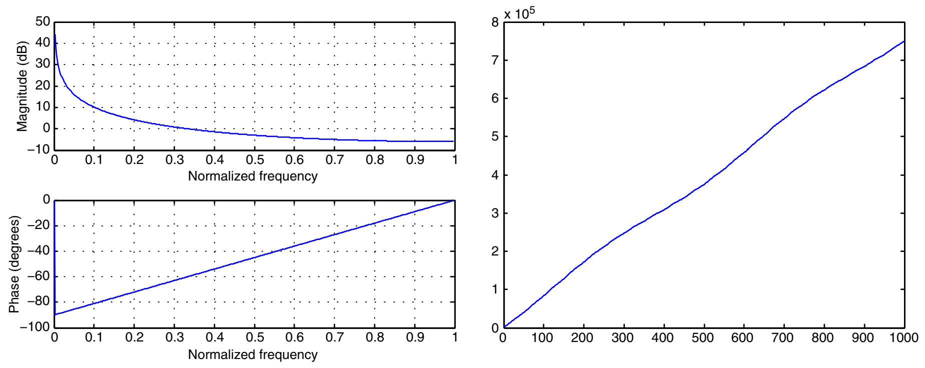

Figure 22.20

Left: `a=[1 -1]; b=[1]; freqz(b,a)`. Note that the frequency response of the integrator is infinite to DC signals (the integral of a nonzero constant signal goes to infinity) and low for high frequency signals. This is opposite of the differencing filter. **Right**: `plot(filter(b,a,x))`. The `filter` command applies the filter with coefficients `b` and `a` to `x`. This generalizes `conv` to IIR filters. (We cannot simply use `conv` for IIR filters, since the impulse response is not finite.) The upward slope of the integral is due to the nonzero DC term. We can also see the wiggle due to the 2 Hz term. It is basically impossible to see the 20 and 400 Hz terms in the signal.

etc., until `X(129)` corresponds to frequency $128f_s/N = f_s/2 = f_N$. So the frequencies we care about are near index 14 and its counterpart $258 - 14 = 244$. To cancel other frequencies, we can do

```
halfwidth = 3;
Xfiltered = zeros(1,256);
Xfiltered(14-halfwidth:14+halfwidth) = X(14-halfwidth:14+halfwidth);
Xfiltered(244-halfwidth:244+halfwidth) = X(244-halfwidth:244+halfwidth);
xrecovered = fft(conj(Xfiltered)/N);
plot(real(xrecovered));
```

The result is the (approximate) sinusoidal component of the square wave at $0.1f_N$, shown in Figure 22.21.

This is simple! (Of course the cost is in computing the FFT and inverse FFT.) FFT-based filter design tools allow you to specify an arbitrary frequency response (e.g., by drawing the magnitude response) and the size of the filter you are willing to accept, then use an inverse FFT to find filter coefficients that best match the desired response. The more coefficients you allow, the closer the approximation.

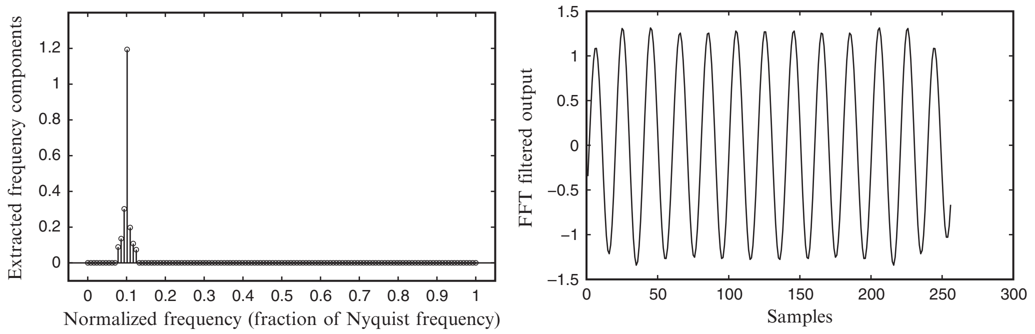

Figure 22.21
(Left) The extracted frequencies from the FFT magnitude plot in Figure 22.6. (Right) The FFT filter output, which is approximately a sinusoid at $0.1f_N$.

22.6 DSP on the PIC32

MIPS provides a DSP library for the MIPS M4K CPU on the PIC32, including FFT, FIR filtering, IIR filtering, and other DSP functions, described in Microchip's "32-Bit Language Tools Libraries" manual. For efficiency, the code is written in assembly language, optimizing the number of instructions. It also uses 16- and 32-bit fixed-point numbers to represent values, unlike MATLAB's double-precision floating point numbers. Fixed-point math is identical to integer math, and, as we have seen, integer math is significantly faster than floating point math. We will revisit fixed-point math soon.

This section presents an example demonstrating FIR filtering and the FFT on the PIC32, comparing the results to results you get in MATLAB. The MATLAB code generates a 1024-sample square wave as well as a 48-coefficient low-pass FIR filter with a cutoff frequency at $0.02f_N$. It sends these to the PIC32, which calculates the FIR-filtered signal, the FFT of the original signal, and the FFT of the filtered signal, and sends the results back to MATLAB for plotting. MATLAB also calculates the filtered signal and the FFTs of the original and filtered signals and compares the results to the PIC32's results. The results are indistinguishable (Figure 22.22). MATLAB also prints out the time it takes to do a 1024-sample FFT on the PIC32, about 13 ms.

The code for this example consists of the following files:

PIC32 code

- `dsp_fft_fir.c`. Communicates with the host and invokes the PIC32 DSP functions in `nudsp.c`.
- `nudsp.c`. Code that calls the MIPS functions.
- `nudsp.h`. Header file with prototypes for functions in `nudsp.c`.

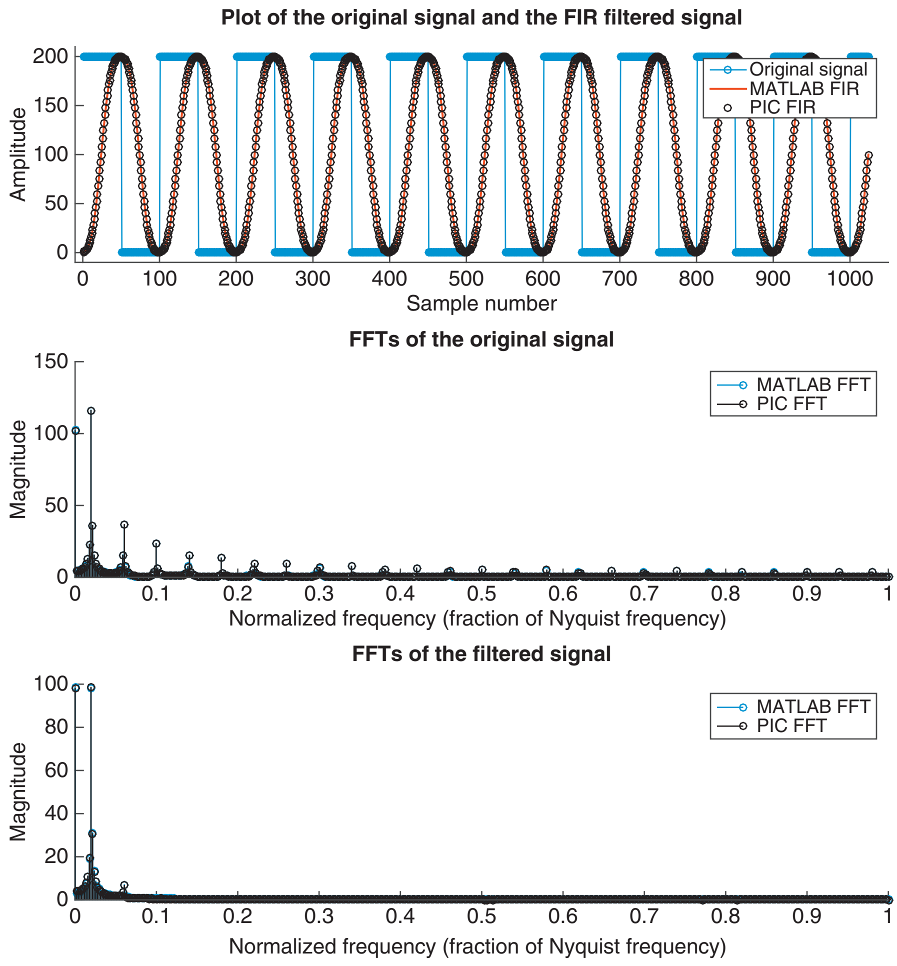

Figure 22.22
(Top) The 1024 samples of the original square wave signal, the MATLAB low-pass-filtered signal, and the PIC32 low-pass-filtered signal. The results are indistinguishable. (Middle) The MATLAB and PIC32 FFTs of the original signal. (Bottom) The MATLAB and PIC32 FFTs of their low-pass-filtered signals. Although it is difficult to see, the FFT results on the two platforms are indistinguishable.

Host computer code. We focus on the MATLAB interface, but we also provide code for Python.

- `sampleFFT.m` if you are using MATLAB, or
- `sampleFFT.py` if you are using Python. Python users need the freely available packages pyserial (for UART communication), matplotlib (for plotting), numpy (for matrix operations), and scipy (for DSP).

The MATLAB client code is given below. The Python code is similar. Load the PIC32 executable, then in MATLAB run `sampleFFT` to do the test.

Code Sample 22.2 `sampleFFT.m`**. MATLAB Client for FIR and FFT.**

```
% Compute the FFT of a signal and FIR filter the signal in both MATLAB and on the PIC32
% and compare the results
% open the serial port
port ='/dev/tty.usbserial-00001014A';  % modify for your own port

if ~isempty(instrfind)  % closes the port if it was open
  fclose(instrfind);
  delete(instrfind);
end
fprintf('Opening serial port %s\n',port);
ser = serial(port, 'BaudRate', 230400, 'FlowControl','hardware');
fopen(ser);

% generate the input signal
xp(1:50) = 200;
xp(51:100)= 0;
x = [xp xp xp xp xp xp xp xp xp xp 200*ones(1,24)];

% now, create the FIR filter
Wn = 0.02; % let's see if we can just get the lowest frequency sinusoid

ord = 47; % ord+1 must be a multiple of 4
fir_coeff = fir1(ord,Wn);

N = length(x);
Y = fft(x);    % computer MATLAB's fft

xfil = filter(fir_coeff,1,x); % filter the signal
Yfil = fft(xfil);             % fft the filtered signal

% generate data for FFT plots for the original signal
mag = 2*abs(Y(1:N/2+1))/N;
mag(1) = mag(1)/2;
mag(N/2+1) = mag(N/2+1)/2;

% generate data for FFT plots for the filtered signal
magfil = 2*abs(Yfil(1:N/2+1))/N;
magfil(1) = magfil(1)/2;
magfil(N/2+1) = magfil(N/2+1)/2;

freqs = linspace(0,1,N/2+1);

% send the original signal to the pic32
fprintf(ser,'%d\n',N); % send the length
for i=1:N
  fprintf(ser,'%f\n',x(i)); % send each sample in the signal
end

% send the fir filter coefficients
fprintf(ser,'%d\n',length(fir_coeff));
for i=1:length(fir_coeff)
  fprintf(ser,'%f\n',fir_coeff(i));
end
```

```
% now we can read in the values sent from the PIC.
elapsedns = fscanf(ser,'%d');
disp(['The first 1024-sample FFT took ',num2str(elapsedns/1000.0),' microseconds.']);
Npic = fscanf(ser,'%d');
data = zeros(Npic,4); % the columns in data are
                      % original signal, fir filtered, orig fft, fir fft
for i=1:Npic
  data(i,:) = fscanf(ser,'%f %f %f %f');
end

xpic = data(:,1);            % original signal from the pic
xfirpic = data(:,2);         % fir filtered signal from pic
Xfftpic = data(1:N/2+1,3);   % fft signal from the pic
Xfftfir = data(1:N/2+1,4);   % fft of filtered signal from the pic

                             % used to plot the fft pic signals
Xfftpic = 2*abs(Xfftpic);
Xfftpic(1) = Xfftpic(1)/2;
Xfftpic(N/2+1) = Xfftpic(N/2+1)/2;

Xfftfir = 2*abs(Xfftfir);
Xfftfir(1) = Xfftfir(1)/2;
Xfftfir(N/2+1) = Xfftfir(N/2+1)/2;

% now we are ready to plot
subplot(3,1,1);
hold on;
title('Plot of the original signal and the FIR filtered signal')
xlabel('Sample number')
ylabel('Amplitude')
plot(x,'Marker','o');
plot(xfil,'Color','red','LineWidth',2);
plot(xfirpic,'o','Color','black');
hold off;
legend('Original Signal','MATLAB FIR', 'PIC FIR')
axis([-10,1050,-10,210])
set(gca,'FontSize',18);

subplot(3,1,2);
hold on;
title('FFTs of the original signal')
ylabel('Magnitude')
xlabel('Normalized frequency (fraction of Nyquist Frequency)')
stem(freqs,mag)
stem(freqs,Xfftpic,'Color','black')
legend('MATLAB FFT', 'PIC FFT')
hold off;
set(gca,'FontSize',18);

subplot(3,1,3);
hold on;
title('FFTs of the filtered signal')
ylabel('Magnitude')
xlabel('Normalized frequency (fraction of Nyquist Frequency)')
stem(freqs,magfil)
stem(freqs,Xfftfir,'Color','black')
legend('MATLAB FFT', 'PIC FFT')
hold off;
set(gca,'FontSize',18);

fclose(ser);
```

The PIC32 code `dsp_fft_fir.c` contains the `main` function. It reads the signal and filter information from MATLAB, invokes functions from our `nudsp.{h,c}` library to compute the filtered signal and the FFTs of the original and filtered signals, and sends the data back to the host for plotting.

Code Sample 22.3 `dsp_fft_fir.c`. Communicates with the Client and Uses `nudsp` to Perform Signal Processing Operations.

```
#include "NU32.h"
#include "nudsp.h"
// Receives a signal and FIR filter coefficients from the computer.
// filters the signal and ffts the signal and filtered signal, returning the results
// We omit error checking for clarity, but always include it in your own code.

#define SIGNAL_LENGTH 1024
#define FFT_SCALE 10.0
#define FIR_COEFF_SCALE 10.0
#define FIR_SIG_SCALE 10.0
#define NS_PER_TICK 25        // nanoseconds per core clock tick

#define MSG_LEN 128

int main(void) {
  char msg[MSG_LEN];                           // communication buffer
  double fft_orig[SIGNAL_LENGTH] = {};         // fft of the original signal
  double fft_fir[SIGNAL_LENGTH] = {};          // fft of the FIR filtered signal
  double xfir[SIGNAL_LENGTH] = {};             // the FIR filtered signal
  double sig[SIGNAL_LENGTH] = {};              // the signal
  double fir[MAX_ORD] = {};                    // the FIR filter coefficients
  int i = 0;
  int slen, clen;                              // signal and coefficient lengths
  int elapsedticks;                            // duration of FFT in core ticks

  NU32_Startup();

  while (1) {
    // read the signal from the UART.
    NU32_ReadUART3(msg, MSG_LEN);
    sscanf(msg,"%d",&slen);
    for(i = 0; i < slen; ++i) {
      NU32_ReadUART3(msg, MSG_LEN);
      sscanf(msg,"%f",&sig[i]);
    }

    // read the filter coefficients from the UART
    NU32_ReadUART3(msg,MSG_LEN);
    sscanf(msg,"%d", &clen);
    for(i = 0; i < clen; ++i) {
      NU32_ReadUART3(msg,MSG_LEN);
      sscanf(msg,"%f",&fir[i]);
    }

    // FIR filter the signal
    nudsp_fir_1024(xfir, sig, fir, clen, FIR_COEFF_SCALE, FIR_SIG_SCALE);

    // FFT the original signal; also time the FFT and send duration in ns
    _CP0_SET_COUNT(0);
```

```
      nudsp_fft_1024(fft_orig, sig, FFT_SCALE);
      elapsedticks = _CP0_GET_COUNT();
      sprintf(msg,"%d\r\n",elapsedticks*NS_PER_TICK);  // the time in ns
      NU32_WriteUART3(msg);
      // FFT the FIR signal
      nudsp_fft_1024(fft_fir, xfir, FFT_SCALE);

      // send the results to the computer
      sprintf(msg,"%d\r\n",SIGNAL_LENGTH);  // send the length
      NU32_WriteUART3(msg);
    for (i = 0; i < SIGNAL_LENGTH; ++i) {
      sprintf(msg,"%12.6f %12.6f %12.6f %12.6f\r\n",sig[i],xfir[i],fft_orig[i],fft_fir[i]);
      NU32_WriteUART3(msg);
     }
    }
    return 0;
  }
```

Code Sample 22.4 `nudsp.h`**. Header File for FIR and FFT of Signals Represented as** `double` **Arrays.**

```
#ifndef NU__DSP__H__
#define NU__DSP__H__
// wraps some dsp library functions making it easier to use them with doubles
// all provided operations assume signal lengths of 1024 elements

#define MAX_ORD 128        // maximum order of the FIR filter

// compute a scaling factor for converting doubles into Q15 numbers
double nudsp_qform_scale(double * din, int len, double div);

// FFT a signal that has 1024 samples
void nudsp_fft_1024(double * dout, double * din, double div);

// FIR filter a signal that has 1024 samples
// arguments are dout (output), din (input), c (FIR coeffs), nc (number of coeffs),
//   div_c (coeffs scale factor for Q15), and div_sig (signal scale factor for Q15)
void
  nudsp_fir_1024(double *dout,double *din,double *c,int nc,double div_c,double div_sig);

#endif
```

The code `nudsp.c`, below, uses the MIPS `dsp` library to perform signal processing operations on arrays of type `double`. The `dsp` library, however, represents numbers in a fixed-point fractional format, either Q15 (a 16-bit format) or Q31 (a 32-bit format). The representation is the same as a two's complement integer, with the most significant bit corresponding to the sign, but the values of the bits are interpreted differently. For example, for Q15, bit 14 is the 2^{-1} column, bit 13 is the 2^{-2} column, etc., down to bit 0, the 2^{-15} column (see Table 22.1). This interpretation means that Q15 can represent fractional values from -1 to $1 - 2^{-15}$.

The advantage of a fixed-point format over a floating point format is that all the rules of integer math apply, allowing fast integer operations. The disadvantage is that it covers a smaller range of values than floating point numbers with the same number of bits (though with

Table 22.1: The fixed-point Q15 representation is equivalent to the 16-bit two's complement integer representation, and it uses the same mathematical operations

Binary	`int16` Interpretation	Q15 Interpretation
0000000000000000	0	0
0000000000000001	1	2^{-15}
0000000000000010	2	2^{-14}
0000000000000011	3	$2^{-14} + 2^{-15}$
⋮		
0111111111111111	32,767	$1 - 2^{-15}$
1000000000000000	$-32,768$	-1
1000000000000001	$-32,767$	$-1 + 2^{-15}$
1000000000000010	$-32,766$	$-1 + 2^{-14}$
⋮		
1111111111111111	-1	-2^{-15}

The only difference is that consecutive numbers in Q15 are separated by 2^{-15}, covering the range -1 to $1 - 2^{-15}$, as opposed to the `int16` representation, which has consecutive numbers separated by 1, covering the range $-32,768$ to $32,767$.

uniform resolution over the range, unlike floating point numbers). If a signal is represented as an array of `double`s, before using it in a fixed-point computation the signal should be scaled so that (1) the maximum range of the scaled signal is well less than the fixed-point format's range, to allow headroom for additions and subtractions without causing overflow; and (2) to make sure that there is sufficient resolution in the scaled signal's representation, avoiding quantization effects that significantly alter the shape of the signal.

The `dsp` library defines four data types to hold both real and complex fixed-point Q15 and Q31 numbers: `int16`, `int16c`, `int32`, and `int32c`, where the number indicates the number of bits in the representation and the `c` is added to indicate that the type is a `struct` with both real (`.re`) and imaginary (`.im`) values. Our code only uses Q15 numbers (`int16` and `int16c`), as they provide enough precision for our purposes and some `dsp` functions only accept Q15 arguments. As described above and in Table 22.1, Q15 numbers in the range -1 to $1 - 2^{-15}$ can be interpreted as `int16` numbers a factor 2^{15} larger. Therefore, in the rest of this section, we refer only to `int16` integers in the range $-32,768$ to 32,767.

The function `nudsp_qform_scale` computes an appropriate scaling factor for a signal, which is used to convert an array of `double`s into an array of `int16` integers. The scaling normalizes the signal by its largest magnitude value, mapping that value to `1/div`, where `div` is a scaling factor used to provide headroom to prevent overflow. (The sample code `dsp_fft_fir.c` chooses `div = 10.0` for signals used in the FFT and FIR filters, scaling them to the range $[-0.1, 0.1]$.) This scaling factor is then multiplied by `QFORMAT` = 2^{15}, which scales the signal up to use the range offered by the `int16` data type. To convert `double`s to `int16`s we multiply by the calculated scaling factor, and to convert back to `double`s we divide by the scaling factor.

The next function, `nudsp_fft_1024`, uses the `dsp` library function `mips_fft16` to perform an FFT on a signal represented as an array of 1024 `double`s. First, the signal must be converted into an array of complex `int16` numbers; we use `nudsp_qform_scale` to compute the scaling factor. Next we copy the *twiddle* factors, parameters used in the FFT algorithm, into RAM. The `dsp` library (through `fftc.h`) provides precomputed factors and places them in an array in flash called `fft16c1024`; we load them into RAM for greater speed. Finally, we call `mips_fft16` with a buffer for the result of the computation, the source signal, the twiddle factors, a scratch array, and the $\log_2$ of the signal length (the signal length must always be a power of 2, and here we assume it is 1024). The scratch array just provides extra memory for the `mips_fft16` function to perform temporary calculations. After computing the FFT, we convert the magnitudes back into `double`s, using the scaling factor computed earlier.

The final function, `nudsp_fir_1024`, applies an FIR filter to a signal represented as an array of 1024 `double`s. The first step prior to using the `dsp` library's FIR function is to scale the signal and the coefficients by `scale_c` and `scale_s`, respectively, converting them to `int16`s. After performing the scaling, we must call `mips_fir16_setup` to initialize a coefficient buffer that is twice as long as the actual number of filter coefficients. Finally, the call to `mips_fir16` performs the filtering operation. In addition to the input and output buffers and prepared coefficients, the filter also requires a buffer long enough to hold the last K samples, where K is the number of filter coefficients. The filter returns its result as `int16` numbers. To convert the result back to `double`s, we must divide by the product of the scaling factors of the coefficients and the signal, because these numbers are multiplied during the filter operation. Since both scaling factors `scale_c` and `scale_s` contain a factor `QFORMAT` to convert to the `int16` range, we eliminate one of these scaling factors by multiplying by `QFORMAT`.

Code Sample 22.5 `nudsp.c`**. Implements FIR and FFT Operations for Arrays of Type** `double`**.**

```
#include <math.h>       // C standard library math, for sqrt
#include <stdlib.h>     // for max
#include <string.h>     // for memcpy
#include <dsplib_dsp.h> // for int16, int16c data types and FIR and FFT functions
#include <fftc.h>       // for the FFT twiddle factors, stored in flash
#include "nudsp.h"

#define TWIDDLE fft16c1024 // FFT twiddle factors for int16 (Q15), 1024 signal length
#define LOG2N 10           // log base 2 of the length, assumed to be 1024
#define LEN (1 << LOG2N)   // the length of the buffer
#define QFORMAT (1 << 15)  // multiplication factor to map range (-1,1) to int16 range

// compute the scaling factor to convert an array of doubles to int16 (Q15)
// The scaling is performed so that the largest magnitude number in din
// is mapped to 1/div; thus the divisor gives extra headroom to avoid overflow
```

```
double nudsp_qform_scale(double * din, int len, double div) {
  int i;
  double maxm = 0.0;

  for (i = 0; i< len; ++i) {
    maxm = max(maxm, fabs(din[i]));
  }
  return (double)QFORMAT/(maxm * div);
}

// Performs an FFT on din (assuming it is 1024 long), returning its magnitude in dout
// dout - pointer to array where answer will be stored
// din - pointer to double array to be analyzed
// div - input scaling factor.  max magnitude input is mapped to 1/div

void nudsp_fft_1024(double *dout, double *din, double div)
{
  int i = 0;
  int16c twiddle[LEN/2];
  int16c dest[LEN], src[LEN];
  int16c scratch[LEN];
  double scale = nudsp_qform_scale(din,LEN,div);

  for (i=0; i< LEN; i++) {                        // convert to int16 (Q15)
    src[i].re = (int) (din[i] * scale);
    src[i].im = 0;
  }
  memcpy(twiddle, TWIDDLE, sizeof(twiddle));      // copy the twiddle factors to RAM
  mips_fft16(dest, src, twiddle, scratch, LOG2N); // perform FFT
  for (i = 0; i < LEN; i++) {                     // convert the results back to doubles
    double re = dest[i].re / scale;
    double im = dest[i].im / scale;
    dout[i] = sqrt(re*re + im*im);
  }
}

// Perform a finite impulse response filter of a signal that is 1024 samples long
// dout - pointer to result array
// din - pointer to input array
// c - pointer to coefficient array
// nc - the number of coefficients
// div_c - for scaling the coefficients
// The maximum magnitude coefficient is mapped to 1/div_c in int16 (Q15)
// div_sig - for scaling the input signal
// The maximum magnitude input is mapped to 1/div_sig in int16 (Q15)
void
  nudsp_fir_1024(double *dout,double *din,double *c,int nc,double div_c,double div_sig)
{
  int16 fir_coeffs[MAX_ORD], fir_coeffs2x[2*MAX_ORD];
  int16 delay[MAX_ORD] = {};
  int16 din16[LEN], dout16[LEN];
  int i=0;
  double scale_c = nudsp_qform_scale(c, nc, div_c);      // scale coeffs to Q15
  double scale_s = nudsp_qform_scale(din, LEN, div_sig); // scale signal to Q15
  double scale = 0.0;

  for (i = 0; i< nc; ++i) {                              // convert FIR coeffs to Q15
    fir_coeffs[i] = (int) (c[i]*scale_c);
  }
  for (i = 0; i<LEN; i++) {                              // convert input signal to Q15
    din16[i] = (int) (din[i]*scale_s);
    }
    mips_fir16_setup(fir_coeffs2x, fir_coeffs, nc);        // set up the filter
    mips_fir16(dout16, din16, fir_coeffs2x, delay, LEN, nc, 0);  // run the filter
    scale = (double)QFORMAT/(scale_c*scale_s);            // convert back to doubles
    for (i = 0; i<LEN; i++) {
      dout[i] = dout16[i]*scale;
    }
  }
```

22.7 Exercises

1. Use MATLAB to find the coefficients of a 20th-order low-pass FIR filter with a cutoff frequency of $0.5f_N$. Then do the same for a 100th-order low-pass FIR filter. Plot the frequency response of each and discuss the relative merits of each in terms of both the magnitude response and the phase response. For a real-time filter, i.e., one that is performing the filtering on data as it comes in, what are the implications of the different phase responses of the two filters?
2. Experiment with MATLAB's `sound` function, which allows you to play a vector of numbers as a sound waveform through your computer's speakers. Create a one-second signal sampled at 8.192 kHz that is the sum of a 500 and a 2500 Hz sinusoid, each of amplitude 0.5, and play it through your speakers. In MATLAB, design a low-pass FIR filter to extract only the 500 Hz tone and a high-pass FIR filter to extract only the 2500 Hz tone. Plot the frequency response of each filter and verify by audio that the filtered sounds are correct.
3. The PIC32 `dsp` library implements FIR and IIR filters, but it performs a batch filter: all data must be collected, and then you filter it. Often filters must be calculated in real-time for real-time control. For example, noisy sensor data could be low-pass filtered, or position readings could be differenced to get velocity readings.
   Implement your own PIC32 real-time FIR filter library, `FIR.{c,h}`. This library provides variables (e.g., arrays or `structs`) to hold the filter coefficients and the most recently taken samples and calculates the output of the filter. The library should be easy to use and computationally efficient, though you are welcome to use `double`s, not fixed-point math. How will you let the user "shift in" the next sensor reading while "shifting out" the oldest sensor reading? How will you handle initial conditions when no prior readings have been taken?
   Design a ninth-order low-pass FIR filter with a cutoff frequency at $0.2\,f_N$ and plot both the input and filtered output for a 1000-input signal consisting of two equal amplitude sinusoids, one at $0.1f_N$ and one at $0.5f_N$.
4. Augment the PIC32 `nudsp.{c,h}` library with an inverse Fast Fourier Transform capability. Test it on a sample signal by taking the FFT and then the inverse FFT and confirming that the original signal is recovered.

5. Use the PIC32 inverse FFT capability to implement an FFT-based filter in `nudsp.{c,h}`. The user specifies the frequency ranges to pass or stop, allowing the FFT-based filter to act as a low-pass, high-pass, bandpass, or bandstop filter. After taking the FFT, the code should zero out the appropriate components and take the inverse FFT, yielding the new filtered signal.

Further Reading

32-Bit language tools libraries. (2012). Microchip Technology Inc.

Oppenheim, A. V., & Schafer, R. W. (2009). *Discrete-time signal processing* (3rd ed.). Upper Saddle River, NJ: Prentice Hall.

Signal processing toolbox help. (2015). The MathWorks Inc.