# **Digital Filter Implementation**

In this chapter we will delve more deeply into the practical task of using digital filters. We will discuss how to accurately and efficiently implement FIR and IIR filters.

You may be asking yourself why this chapter is important. We already know what a digital filter is, and we have (or can find) a program to find the coefficients that satisfy design specifications. We can inexpensively acquire a DSP processor that is so fast that computational efficiency isn't a concern, and accuracy problems can be eliminated by using floating point processors. Aren't we ready to start programming without this chapter?

Not quite. You should think of a DSP processor as being similar to a jet plane; when flown by a qualified pilot it can transport you very quickly to your desired destination, but small navigation errors bring you to unexpected places and even the slightest handling mistake may be fatal. This chapter is a crash course in digital filter piloting.

In the first section of this chapter we discuss technicalities relating to computing convolutions in the time domain. The second section discusses the circular convolution and how it can be used to filter in the frequency domain; this is frequently the most efficient way to filter a signal. Hard real-time constraints often force us to filter in the time domain, and so we devote the rest of the chapter to more advanced time domain techniques. We will exploit the graphical techniques developed in Chapter 12 in order to manipulate filters. The basic building blocks we will derive are called *structures*, and we will study several FIR and HR structures. More complex filters can be built by combining these basic structures.

Changing sampling rate is an important application for which special filter structures known as *polyphase filters* have been developed. Polyphase filters are more efficient for this application than general purpose structures.

We also deal with the effect of finite precision on the accuracy of filter computation and on the stability of IIR filters.

# 15.1 Computation of Convolutions

We have never fully described how to properly compute the convolution sum in practice. There are essentially four variations. Two are *causal*, as required for real-time applications; the other two introduce explicit delays. Two of the convolution procedures process one input at a time in a real-time-oriented fashion (and must store the required past inputs in an internal FIFO), the other two operate on arrays of inputs.

First, there is the *causal FIFO* way

$$y_n = \sum_{l=0}^{L-1} a_l \, x_{n-l} \tag{15.1}$$

which is eminently suitable for real-time implementation. We require two buffers of length L—one constant buffer to store the filter coefficients, and a FIFO buffer for the input samples. The FIFO is often unfortunately called the static buffer; not that it is static-it is changing all the time. The name is borrowed from computer languages where static refers to buffers that survive and are not zeroed out upon each invocation of the convolution procedure. We usually clear the static buffer during program initialization, but for continuously running systems this precaution is mostly cosmetic, since after L inputs all effects of the initialization are lost. Each time a new input arrives we push it into the static buffer of length L, perform the convolution on this buffer by multiplying the input values by the filter coefficients that overlap them, and accumulating. Each coefficient requires one multiply-and-accumulate (MAC) operation. A slight variation supported by certain DSP architectures (see Section 17.6), is to combine the push and convolve operations. In this case the place shifting of the elements in the buffer occurs as part of the overall convolution, in parallel with the computation.

In equation (15.1) the index of summation runs over the filter coefficients. We can easily modify this to become the *causal array* method

$$y_n = \sum_{i=n-(L-1)}^n a_{n-i} x_i \tag{15.2}$$

where the index i runs over the inputs, assuming these exist. This variation is still causal in nature, but describes inputs that have already been placed in an array by the calling application. Rather than dedicating further memory inside our convolution routine for the FIFO buffer, we utilize the existing buffering and its indexation. This variation is directly suitable for off-line computation where we compute the entire output vector in one invocation. When programming we usually shift the indexes to the range  $0 \dots L - 1$  or  $1 \dots L$ .

In off-line calculation there is no need to insist on explicit causality since all the input values are available in a buffer anyway. We know from Chapter 6 that the causal filter introduces a delay of half the impulse response, a delay that can be removed by using a noncausal form. Often the largest filter coefficients are near the filter's center, and then it is even more natural to consider the middle as the position of the output. Assuming an odd number of taps, it is thus more symmetric to index the  $L = 2\lambda + 1$  taps as  $a_{-\lambda} \dots a_0 \dots a_{\lambda}$ , and the explicitly *noncausal FIFO* procedure looks like this.

$$y_n = \sum_{l=-\lambda}^{\lambda} a_l x_{n-l} \tag{15.3}$$

The corresponding *noncausal array*-based procedure is obtained, once again, by a change of summation variable

$$y_n = \sum_{i=n-\lambda}^{n+\lambda} a_{n-i} x_i \tag{15.4}$$

assuming that the requisite inputs exist. This symmetry comes at a price; when we get the  $n^{\text{th}}$  input, we can compute only the  $(n-\lambda)^{\text{th}}$  output. This form makes explicit the buffer delay of  $\lambda$  between input and output.

In all the above procedures, we assumed that the input signal existed for all times. Infinite extent signals pose no special challenge to real-time systems but cannot really be processed off-line since they cannot be placed into finite-length vectors. When the input signal is of finite time duration and has only a finite number N of nonzero values, some of the filter coefficients will overlap zero inputs. Assume that we desire the same number of outputs as there are inputs (i.e., if there are N inputs,  $n = 0, \ldots N - 1$ , we expect Noutputs). Since the input signal is identically zero for n < 0 and  $n \ge N$ , the first output,  $y_0$ , actually requires only  $\lambda + 1$  multiplications, namely  $a_0x_0$ ,  $a_{-1}x_1$ , through  $a_{-\lambda}x_{\lambda}$ , since  $a_1$  through  $a_{\lambda}$  overlap zeros.

Only after  $\lambda$  shifts do we have the filter completely overlapping signal.

 $a_{\lambda}$  $a_1$  $a_0$  $a_{-\lambda+1}$  $a_{-\lambda}$  $a_{\lambda-1}$  $a_{\lambda-2}$  $a_{-1}$ . . .  $x_0$  $x_1$  $x_2$  $x_{\lambda-1}$   $x_{\lambda}$   $x_{\lambda+1}$  ...  $x_{2\lambda-1}$  $x_{2\lambda}$  $x_{2\lambda+1}$  ... . . .

Likewise the last  $\lambda$  outputs have the filter overlapping zeros as well.

 $a_{\lambda}$  $a_{\lambda-1}$ an  $a_1$  $a_0$  $a_{-1}$  $a_{-2}$  $\ldots a_{-\lambda+1}$  $a_{-\lambda}$ 0 0 0 0  $x_{N-l} \quad x_{N-2} \quad \dots \quad x_{N-2} \quad x_{N-1}$  $x_N$ . . .

The programming of such convolutions can take the finite extent into account and not perform the multiplications by zero (at the expense of more complex code). For example, if the input is nonzero only for N samples starting at zero, and the entire input array is available, we can save some computation by using the following sums.

$$y_n = \sum_{i=\max(0,n-(L-1))}^{\min(N-1,n)} a_{n-i} x_i = \sum_{i=\max(0,n-\lambda)}^{\min(N-1,n+\lambda)} a_{n-i} x_i$$
(15.5)

The improvement is insignificant for  $N \gg L$ .

We have seen how to compute convolutions both for real-time-oriented cases and for off-line applications. We will see in the next section that these straightforward computations are *not* the most efficient ways to compute convolutions. It is almost always more efficient to perform convolution by going to the frequency domain, and only harsh real-time constraints should prevent one from doing so.

#### EXERCISES

- 15.1.1 Write two routines for array-based noncausal convolution of an input signal x by an odd length filter a that does not perform multiplications by zero. The routine convolve(N, L, x, a, y) should return an output vector y of the same length N as the input vector. The filter should be indexed from 0 to L-1 and stored in reverse order (i.e.,  $a_0$  is stored in a[L-1]). The output  $y_i$  should correspond to the middle of the filter being above  $x_i$  (e.g., the first and last outputs have about half the filter overlapping nonzero input signal values). The first routine should have the input vector's index as the running index, while the second should use the filter's index.
- 15.1.2 Assume that a noncausal odd-order FIR filter is symmetric and rewrite the above routines in order to save multiplications. Is such a procedure useful for real-time applications?
- 15.1.3 Assume that we only want to compute output values for which all the filter coefficients overlap observed inputs. How many output values will there be? Write a routine that implements this procedure. Repeat for when we want all outputs for which *any* inputs are overlapped.

572

## 15.2 FIR Filtering in the Frequency Domain

After our extensive coverage of convolutions, you may have been led to believe that FIR filtering and straightforward computation of the convolution sum as in the previous section were one and the same. In particular, you probably believe that to compute N outputs of an L-tap filter takes NLmultiplications and N(L-1) additions. In this section we will show how FIR filtering can be accomplished with significantly fewer arithmetic operations, resulting both in computation time savings and in round-off error reduction.

If you are unconvinced that it is possible to reduce the number of multiplications needed to compute something equivalent to N convolutions, consider the simple case of a two-tap filter  $(a_0, a_1)$ . Straightforward convolution of any two consecutive outputs  $y_n$  and  $y_{n+1}$  requires four multiplications (and two additions). However, we can rearrange the computation

$$y_n = a_1 x_n + a_0 x_{n+1} = a_1 (x_n + x_{n+1}) - (a_1 - a_0) x_{n+1}$$
  
$$y_{n+1} = a_1 x_{n+1} + a_0 x_{n+2} = a_0 (x_{n+1} + x_{n+2}) + (a_1 - a_0) x_{n+1}$$

so that only three multiplications are required. Unfortunately, the number of additions was increased to four  $(a_1 - a_0 \text{ can be precomputed})$ , but nonetheless we have made the point that the number of operations may be decreased by identifying redundancies. This is precisely the kind of logic that led us to the FFT algorithm, and we can expect that similar gains can be had for FIR filtering. In fact we can even more directly exploit our experience with the FFT by filtering in the frequency domain.

We have often stressed the fact that filtering a signal in the time domain is equivalent to multiplying by a frequency response in the frequency domain. So we should be able to perform an FFT to jump over to the frequency domain, multiply by the desired frequency response, and then iFFT back to the time domain. Assuming both signal and filter to be of length N, straight convolution takes  $O(N^2)$  operations, while the FFT  $(O(N \log N))$ , multiplication (O(N)), and iFFT (once again  $O(N \log N))$  clock in at  $O(N \log N)$ . This idea is *almost* correct, but there are two caveats. The first problem arises when we have to filter an infinite signal, or at least one longer than the FFT size we want to use; how do we piece together the individual results into a single coherent output? The second difficulty is that property (4.47) of the DFT specifies that multiplication in the digital frequency domain corresponds to *circular* convolution of the signals, and not *linear* convolution.

As discussed at length in the previous section, the convolution sum contains shifts for which the filter coefficients extend outside the signal. There



Figure 15.1: Circular convolution for a three-coefficient filter. For shifts where the index is outside the range  $0 \dots N - 1$  we assume it wraps around periodically, as if the signal were on a circle.

we assumed that when a nonexistent signal value is required, it should be taken to be zero, resulting in what is called *linear convolution*. Another possibility is *circular convolution*, a quantity mentioned before briefly in connection with the aforementioned property of the DFT. Given a signal with L values  $x_0, x_1 \dots x_{L-1}$  and a set of M coefficients  $a_0, a_1 \dots a_{M-1}$  we defined the circular (also called cyclic) convolution to be

$$y_l = a \circledast x \equiv \sum_m a_m \, x_{(l-m) \bmod L}$$

where mod is the integer modulus operation (see appendix A.2) that always returns an integer between 0 and L-1. Basically this means that when the filter is outside the signal range rather than overlapping zeros we wrap the signal around, as depicted in Figure 15.1.

Linear and circular convolution agree for all those output values for which the filter coefficients overlap true signal values; the discrepancies appear only at the edges where some of the coefficients jut out. Assuming we have a method for efficiently computing the circular convolution (e.g., based on the FFT), can it somehow be used to compute a linear convolution? It's not hard to see that the answer is yes, for example, by zero-padding the signal to force the filter to overlap zeros. To see how this is accomplished, let's take a length-L signal  $x_0 \ldots x_{L-1}$ , a length M filter  $a_0 \ldots a_{M-1}$ , and assume that M < L. We want to compute the L linear convolution outputs  $y_0 \ldots y_{L-1}$ . The L - M + 1 outputs  $y_{M-1}$  through  $y_{L-1}$  are the same for circular and linear convolution, since the filter coefficients all overlap true inputs. The other M - 1 outputs  $y_0$  through  $y_{M-2}$  would normally be different, but if we artificially extend the signal by  $x_{-M+1} = 0$ , through  $x_{-1} = 0$  they end up being the same. The augmented input signal is now of length N = L+M-1, and to exploit the FFT we may desire this N to be a power of two. It is now easy to state the entire algorithm. First we append M-1 zeros to the beginning of the input signal (and possibly more for the augmented signal buffer to be a convenient length for the FFT). We similarly zero-pad the filter to the same length. Next we FFT both the signal and the filter. These two frequency domain vectors are multiplied resulting in a frequency domain representation of the desired result. A final iFFT retrieves N values  $y_n$ , and discarding the first M-1 we are left with the desired L outputs.

If N is small enough for a single FFT to be practical we can compute the linear convolution as just described. What can be done when the input is very large or infinite? We simply break the input signal into blocks of length N. The first output block is computed as described above; but from then on we needn't pad with zeros (since the input signal isn't meant to be zero there) rather we use the actual values that are available. Other than that everything remains the same. This technique, depicted in Figure 15.2, is called the *overlap save* method, since the FFT buffers contain M-1 input values saved from the previous buffer. In the most common implementations the M-1 last values in the buffer are copied from its end to its beginning, and then the buffer is filled with N new values from that point on. An even better method uses a circular buffer of length L, with the buffer pointer being advanced by N each time.

You may wonder whether it is really necessary to compute and then discard the first M-1 values in each FFT buffer. This discarding is discarded in an alternative technique called *overlap add*. Here the inputs are not overlapped, but rather are zero-padded at their ends. The linear convolution can be written as a sum over the convolutions of the individual blocks, but the first M-1 output values of each block are missing the effect of the previous inputs that were not *saved*. To compensate, the corresponding outputs are *added* to the outputs from the previous block that corresponded to the zero-padded inputs. This technique is depicted in Figure 15.3.

If computation of FIR filters by the FFT is so efficient, why is straightforward computation of convolution so prevalent in applications? Why do DSP processors have special hardware for convolution, and why do so many software filters use it exclusively? There are two answers to these questions. The first is that the preference is firmly grounded in ignorance and laziness. Straightforward convolution is widely known and relatively simple to code compared with overlap save and add. Many designers don't realize that savings in real-time can be realized or don't want to code FFT, overlap, etc. The other reason is more fundamental and more justifiable. In real-time applications there is often a limitation on *delay*, the time between an input appearing and the corresponding output being ready. For FFT-based tech-



**Figure 15.2:** Overlap save method of filtering in the frequency domain. The input signal  $x_n$  is divided into blocks of length L, which are augmented with M-1 values saved from the previous block, to fill a buffer of length N = L + M - 1. Viewed another way, the input buffers of length N overlap. The buffer is converted to the frequency domain and multiplied there by N frequency domain filter values. The result is converted back into the time domain, M-1 incorrect values discarded, and L values output.



**Figure 15.3:** Overlap add method of filtering in the frequency domain. The input signal  $x_n$  is divided into blocks of length L, to which are added M-1 zeros to fill a buffer of length N = L + M - 1. This buffer is converted to the frequency domain and multiplied there by N frequency domain filter values. The result is converted back into the time domain, M-1 partial values at the beginning of the buffer are overlapped and then added to the M-1 last values from the previous buffer.

niques this delay is composed of two parts. First we have to fill up the signal buffer (and true gains in efficiency require the use of large buffers), resulting in *buffer delay*, and then we have to perform the entire computation (FFT, block multiplication, iFFT), resulting in algorithmic delay. Only after all this computation is completed can we start to output the  $y_n$ . While the input sample that corresponds to the last value in a buffer suffers only the algorithmic delay, the first sample suffers the sum of both delays. For applications with strict limitations on the allowed delay, we must use techniques where the computation is spread evenly over time, even if they require more computation overall.

#### EXERCISES

- 15.2.1 Explain why circular convolution requires specification of the buffer size while linear convolution doesn't. Explain why linear convolution can be considered circular convolution with an infinite buffer.
- 15.2.2 The circular convolution  $y_0 = a_0x_0 + a_1x_1$ ,  $y_1 = a_1x_0 + a_0x_1$  implies four multiplications and two additions. Show that it can be computed with two multiplications and four additions by precomputing  $G_0 = \frac{1}{2}(a_0 + a_1)$ ,  $G_1 = \frac{1}{2}(a_0 a_1)$ , and for each  $x_0, x_1$  computing  $z_0 = x_0 + x_1$  and  $z_1 = x_0 x_1$ .
- 15.2.3 Convince yourself that overlap save and overlap add really work by coding routines for straightforward linear convolution, for OA and for OS. Run all three and compare the output signals.
- 15.2.4 Do you expect OA/OS to be more or less numerically accurate than straightforward convolution in the time domain?
- 15.2.5 Compare the number of operations per time required for filtering an infinite signal by a filter of length M, using straightforward time domain convolution with that using the FFT. What length FFT is best? When is the FFT method worthwhile?
- 15.2.6 One can compute circular convolution using an algorithm designed for linear convolution, by replicating parts of the signal. By copying the L-2 last values before  $x_0$  (the cyclic prefix) and the L-2 first values after  $x_{N-1}$  (the cyclic suffix), we obtain a signal that looks like this.

 $\begin{array}{cccc} 0, 0, & x_{N-L+1}, x_{N-L+2}, \dots x_{N-2}, x_{N-1}, \\ & x_0, x_1, \dots x_{N-2}, x_{N-1}, \\ & x_0, x_1, \dots x_{L-3}, x_{L-2}, & 0, 0 \end{array}$ 

Explain how to obtain the desired circular convolution.

15.2.7 Can IIR filtering be performed in the frequency domain using techniques similar to those of this section? What about LMS adaptive filtering?

### 15.3 FIR Structures

In this section we return to the time domain computation of convolution of Section 15.1 and to the utilization of graphic techniques for FIR filtering commenced in Section 12.2. In the context of digital filters, graphic implementations are often called *structures*.



Figure 15.4: Direct form implementation of the FIR filter. This form used to be known as the 'tapped delay line', as it is a direct implementation of the weighted sum of delayed taps of the input signal.

In Figure 12.5, reproduced here with slight notational updating as Figure 15.4, we saw one graphic implementation of the linear convolution. This structure used to be called the 'tapped delay line'. The image to be conjured up is that of the input signal being delayed by having to travel with finite velocity along a line, and values being tapped off at various points corresponding to different delays. Today it is more commonly called the *direct form* structure. The direct form implementation of the FIR filter is so prevalent in DSP that it is often considered sufficient for a processor to efficiently compute it to be considered a DSP processor. The basic operation in the tapped delay line is the multiply-and-accumulate (MAC), and the number of MACs per second (i.e., the number of taps per second) that a DSP can compute is the universal benchmark for DSP processor strength.



Figure 15.5: Transposed form implementation of the FIR filter. Here the present input  $x_n$  is multiplied simultaneously by all L filter coefficients, and the intermediate products are delayed and summed.



Figure 15.6: Cascading simple filters. On the left the output y is created by filtering w, itself the output of filtering x. On the right is the equivalent single filter system.

Another graphic implementation of the FIR filter is the transposed structure depicted in Figure 15.5. The most striking difference between this form and the direct one is that here the undelayed input  $x_n$  is multiplied in parallel by all the filter coefficients, and it is these intermediate products that are delayed. Although theoretically equivalent to the direct form the fact that the computation is arranged differently can lead to slightly different numeric results in practice. For example, the round-off noise and overflow errors will not be the same in general.

The transposed structure can be advantageous when we need to partition the computation. For example, assume you have at your disposal digital filter hardware components that can compute L' taps, but your filter specification can only be satisfied with L > L' taps. Distributing the computation over several components is somewhat easier with the transposed form, since we need only provide the new input  $x_n$  to all filter components in parallel, and connect the upper line of Figure 15.5 in series. The first component in the series takes no input, and the last component provides the desired output. Were we to do the same thing with the direct form, each component would need to receive *two* inputs from the previous one, and provide *two* outputs to the following one.

However, if we really want to neatly partition the computation, the best solution would be to satisfy the filter specifications by cascading several filters in series. The question is whether general filter specifications can be satisfied by cascaded subfilters, and if so how to find these subfilters.

In order to answer these questions, let's experiment with cascading simple filters. As the simplest case we'll take the subfilters to depend on the present and previous inputs, and to have unity DC gain (see Figure 15.6).

$$w_n = ax_n + bx_{n-1} a + b = 1 y_n = cw_n + dw_{n-1} c + d = 1 (15.6)$$

Substituting, we see that the two in series are equivalent to a single filter that depends on the present and two past inputs.

$$y_n = c(ax_n + bx_{n-1}) + d(ax_{n-1} + bx_{n-2})$$
  
=  $acx_n + (ad + bc)x_{n-1} + bdx_{n-2}$  (15.7)  
=  $Ax_n + Bx_{n-1} + Cx_{n-2}$ 

Due to the unity gain constraints the original subfilters only have one free parameter each, and it is easy to verify that the DC gain of the combined filter is unity as expected (A + B + C = 1). So we started with two free parameters, ended up with two free parameters, and the relationship from a, b, c, d to A, B, C is invertible. Given any unity DC gain filter of the form in the last line of equation (15.7) we can find parameters a, b, c, d such that the series connection of the two filters in equation (15.6) forms an equivalent filter. More generally, if the DC gain is nonunity we have four independent parameters in the cascade form, and only three in the combined form. This is because we have the extra freedom of arbitrarily dividing the gain between the two subfilters.

This is one of the many instances where it is worthwhile to simplify the algebra by using the zT formalism. The two filters to be cascaded are described by

$$w_n = (a + bz^{-1}) x_n$$
  
 $y_n = (c + dz^{-1}) w_n$ 

and the resultant filter is given by the product.

$$y_n = (c + dz^{-1})(a + bz^{-1}) x_n$$
  
=  $(ac + (ad + bc)z^{-1} + bdz^{-2}) x_n$   
=  $(A + Bz^{-1} + Cz^{-2}) x_n$ 

We see that the A, B, C parameters derived here by formal multiplication of polynomials in  $z^{-1}$  are exactly those derived above by substitution of the intermediate variable  $w_n$ . It is suggested that the reader experiment with more complex subfilters and become convinced that this is always the case.

Not only is the multiplication of polynomials simpler than the substitution, the zT formalism has further benefits as well. For example, it is hard to see from the substitution method that the subfilters commute, that is, had we cascaded

$$v_n = cx_n + dx_{n-1} \qquad c+d=1$$
  
$$y_n = aw_n + bw_{n-1} \qquad a+b=1$$



Figure 15.7: Cascade form implementation of the FIR filter. Here the input is filtered successively by M 'second-order sections', that is, simple FIR filters that depend on the present input and two past inputs. The term 'second-order' refers to the highest power of  $z^{-1}$  being two, and 'section' is synonymous with what we have been calling 'subfilter'. If  $c_m = 0$  the section is first order.

we would have obtained the same filter. However, this is immediately obvious in the zT formalism, from the commutativity of multiplication of polynomials.

$$(c + dz^{-1})(a + bz^{-1}) = (a + bz^{-1})(c + dz^{-1})$$

Even more importantly, in the zT formalism it is clear that arbitrary filters can be decomposed into cascades of simple subfilters, called *sections*, by factoring the polynomial in zT. The fundamental theorem of algebra (see Appendix A.6) guarantees that all polynomials can be factored into linear factors (or linear and quadratic if we use only real arithmetic); so any filter can be decomposed into cascades of 'first-order' and 'second-order' sections.

$$h_0 + h_1 z^{-1}$$
  $h_0 + h_1 z^{-1} + h_2 z^{-2}$ 

The corresponding structure is depicted in Figure 15.7.

The *lattice structure* depicted in Figure 15.8 is yet another implementation that is built up of basic sections placed in series. The diagonal lines that give it its name make it look very different from the structures we have seen so far, and it becomes even stranger once you notice that the two coefficients on the diagonals of each section are equal. This equality makes the lattice structure numerically robust, because at each stage the numbers being added are of the same order-of-magnitude.



Figure 15.8: Lattice form implementation of the FIR filter. Here the input is filtered successively by M lattice stages, every two of which is equivalent to a direct form second-order section.

In order to demonstrate that arbitrary FIR filters can be implemented as lattices, it is sufficient to show that a general second-order section can be. Then using our previous result that general FIR filters can be decomposed into second-order sections the proof is complete. A second-order section has three free parameters, but one degree of freedom is simply the DC gain. For simplicity we will use the following second-order section.

$$y_n = x_n + h_1 x_{n-1} + h_2 x_{n-2}$$

A single lattice stage has only a single free parameter, so we'll need two stages to emulate the second-order section. Following the graphic implementation for two stages we find

$$y_n = x_n + k_1 x_{n-1} + k_2 (k_1 x_{n-1} + x_{n-2})$$
  
=  $x_n + k_1 (1 + k_2) x_{n-1} + k_2 x_{n-2}$ 

and comparing this with the previous expression leads to the connection between the two sets of coefficients (assuming  $h_2 \neq -1$ ).

$$\begin{array}{rcl} h_1 &=& k_1(1+k_2) & & k_1 &=& \frac{h_1}{1+h_2} \\ h_2 &=& k_2 & & k_2 &=& h_2 \end{array}$$

#### EXERCISES

- 15.3.1 Consider the L-tap FIR filter  $h_0 = 1, h_1 = \lambda, h_2 = \lambda^2, \dots h_{L-1} = \lambda^{L-1}$ . Graph the direct form implementation. How many delays and how many MACS are required? Find an equivalent filter that utilizes feedback. How many delays and arithmetic operations are required now?
- 15.3.2 Why did we discuss series connection of simple FIR filter sections but not parallel connection?

- 15.3.3 We saw in Section 7.2 that FIR filters are linear-phase if they are either symmetric  $h_{-n} = h_n$  or antisymmetric  $h_{-n} = -h_n$ . Devise a graphic implementation that exploits these symmetries. What can be done if there are an even number of coefficients (half sample delay)? What are the advantages of such a implementation? What are the disadvantages?
- 15.3.4 Obtain a routine for factoring polynomials (these are often called polynomial root finding routines) and write a program that decomposes a general FIR filter specified by its impulse response  $h_n$  into first- and second-order sections. Write a program to filter arbitrary inputs using the direct and cascade forms and compare the numeric results.

## **15.4** Polyphase Filters

The structures introduced in the last section were general-purpose (i.e., applicable to most FIR filters you may need). In this section we will discuss a special purpose structure, one that is applicable only in special cases; but these special cases are rather prevalent, and when they do turn up the general-purpose implementations are often not good enough.

Consider the problem of reducing the sampling frequency of a signal to a fraction  $\frac{1}{M}$  of its original rate. This can obviously be carried out by decimation by M, that is, by keeping only one sample out of each M and discarding the rest. For example, if the original signal sampled at  $f_s$  is

decimating by 4 we obtain a new signal  $y_n$  with sampling frequency  $\frac{f_s}{4}$ .

 $y_n = \ldots x_{-12}, x_{-8}, x_{-4}, x_0, x_4, x_8, \ldots$ 

Of course

$$y_n = \dots x_{-11}, x_{-7}, x_{-3}, x_1, x_5, x_9, \dots$$
  
 $y_n = \dots x_{-10}, x_{-6}, x_{-2}, x_2, x_6, x_{10}, \dots$   
 $y_n = \dots x_{-9}, x_{-5}, x_{-1}, x_3, x_7, x_{11}, \dots$ 

corresponding to different phases of the original signal, would be just as good.

Actually, just as *bad* since we have been neglecting aliasing. The original signal x can have energy up to  $\frac{f_s}{2}$ , while the new signal y must not have appreciable energy higher than  $\frac{f_s}{2M}$ . In order to eliminate the illegal components we are required to low-pass filter the original signal before decimating. For definiteness assume once again that we wish to decimate by 4, and to use a causal FIR antialiasing filter h of length 16. Then

$$\begin{split} w_0 &= h_0 x_0 + h_1 x_{-1} + h_2 x_{-2} + h_3 x_{-3} + \dots + h_{15} x_{-15} \\ w_1 &= h_0 x_1 + h_1 x_0 + h_2 x_{-1} + h_3 x_{-2} + \dots + h_{15} x_{-14} \\ w_2 &= h_0 x_2 + h_1 x_1 + h_2 x_0 + h_3 x_{-1} + \dots + h_{15} x_{-13} \\ w_3 &= h_0 x_3 + h_1 x_2 + h_2 x_1 + h_3 x_0 + \dots + h_{15} x_{-12} \\ w_4 &= h_0 x_4 + h_1 x_3 + h_2 x_2 + h_3 x_1 + \dots + h_{15} x_{-11} \end{split}$$
(15.8)

but since we are going to decimate anyway

$$y_n = \ldots w_{-12}, w_{-8}, w_{-4}, w_0, w_4, w_8, \ldots$$

we needn't compute all these convolutions. Why should we compute  $w_1$ ,  $w_2$ , or  $w_3$  if they won't affect the output in any way? So we compute only  $w_0, w_4, w_8, \ldots$ , each requiring 16 multiplications and 15 additions.

More generally, the proper way to reduce the sample frequency by a factor of M is to eliminate frequency components over  $\frac{f_s}{2M}$  using a low-pass filter of length L. This would usually entail L multiplications and additions per input sample, but for this purpose only L per output sample (i.e., only an average of  $\frac{L}{M}$  per input sample are really needed). The straightforward real-time implementation cannot take advantage of this savings in computational complexity. In the above example, at time n = 0, when  $x_0$  arrives, we need to compute the entire 16-element convolution. At time n = 1 we merely collect  $x_1$  but need not perform any computation. Similarly for n = 2 and n = 3 no computation is required, but when  $x_4$  arrives we have to compute another 16-element convolution. Thus the DSP processor must still be able to compute the entire convolution in the time between two samples, since the peak computational complexity is unchanged.

The obvious remedy is to distribute the computation over all the times, rather than sitting idly by and then having to race through the convolution. We already know of two ways to do this; by partitioning the input signal or by decimating it. Focusing on  $w_0$ , partitioning the input leads to structuring the computation in the following way:

$$w_{0} = h_{0}x_{0} + h_{1}x_{-1} + h_{2}x_{-2} + h_{3}x_{-3} \\ + h_{4}x_{-4} + h_{5}x_{-5} + h_{6}x_{-6} + h_{7}x_{-7} \\ + h_{8}x_{-8} + h_{9}x_{-9} + h_{10}x_{-10} + h_{11}x_{-11} \\ + h_{12}x_{-12} + h_{13}x_{-13} + h_{14}x_{-14} + h_{15}x_{-15}$$

Decimation implies the following order:

$$w_{0} = h_{0}x_{0} + h_{4}x_{-4} + h_{8}x_{-8} + h_{12}x_{-12} \\ + h_{1}x_{-1} + h_{5}x_{-5} + h_{9}x_{-9} + h_{13}x_{-13} \\ + h_{2}x_{-2} + h_{6}x_{-6} + h_{10}x_{-10} + h_{14}x_{-14} \\ + h_{3}x_{-3} + h_{7}x_{-7} + h_{11}x_{-11} + h_{15}x_{-15}$$

In both cases we should compute only a single row of the above equations during each time interval, thus evenly distributing the computation over the M time intervals.

Now we come to a subtle point. In a real-time system the input signal  $x_n$  will be placed into a buffer  $\Xi$ . In order to conserve memory this buffer will usually be taken to be of length L, the length of the low-pass filter. The convolution is performed between two buffers of length L, the input buffer and the filter coefficient table; the coefficient table is constant, but a new input  $x_n$  is appended to the input buffer every sampling time.

In the above equations for computing  $w_0$  the subscripts of  $x_n$  are absolute time indices; let's try to rephrase them using input buffer indices instead. We immediately run into a problem with the partitioned form. The input values in the last row are no longer available by the time we get around to wanting them. But this obstacle is easily avoided by reversing the order.

With the understanding that the input buffer updates from row to row, and using a rather uncommon indexing notation for the input buffer, we can now rewrite the partitioned computation as

$$w_0 = h_{12}\Xi_{-12} + h_{13}\Xi_{-13} + h_{14}\Xi_{-14} + h_{15}\Xi_{-15} \\ + h_8\Xi_{-9} + h_9\Xi_{-10} + h_{10}\Xi_{-11} + h_{11}\Xi_{-12} \\ + h_4\Xi_{-6} + h_5\Xi_{-7} + h_6\Xi_{-8} + h_7\Xi_{-9} \\ + h_0\Xi_{-3} + h_1\Xi_{-4} + h_2\Xi_{-5} + h_3\Xi_{-6}$$

and the decimated one as follows.

$$w_0 = h_0 \Xi_{-3} + h_4 \Xi_{-7} + h_8 \Xi_{-11} + h_{12} \Xi_{-15} \\ + h_1 \Xi_{-3} + h_5 \Xi_{-7} + h_9 \Xi_{-11} + h_{13} \Xi_{-15} \\ + h_2 \Xi_{-3} + h_6 \Xi_{-7} + h_{10} \Xi_{-11} + h_{14} \Xi_{-15} \\ + h_3 \Xi_{-3} + h_7 \Xi_{-7} + h_{11} \Xi_{-11} + h_{15} \Xi_{-15}$$



Figure 15.9: The polyphase decimation filter. We depict the decimation of an input signal  $x_n$  by a factor of four, using a polyphase filter. Each decimator extracts only inputs with index divisible by 4, so that the combination of delays and decimators results in all the possible decimation phases.  $h^{[k]}$  for k = 0, 1, 2, 3 are the subfilters;  $h^{[0]} = (h_0, h_1, h_2, h_3)$ ,  $h^{[1]} = (h_4, h_5, h_6, h_7)$ , etc.

While the partitioned version is rather inelegant, the decimated structure is seen to be quite symmetric. It is easy to understand why this is so. Rather than low-pass filtering and then decimating, what we did is to decimate and then low-pass filter at the lower rate. Each row corresponds to a different decimation phase as discussed at the beginning of the section. The low-pass filter coefficients are different for each phase, but the sum of all contributions results in precisely the desired full-rate low-pass filter.

In the general case we can describe the mechanics of this algorithm as follows. We design a low-pass filter that limits the spectral components to avoid aliasing. We decimate this filter creating M subfilters, one for each of the M phases by which we can decimate the input signal. This set of M subfilters is called a *polyphase filter*. We apply the first polyphase subfilter to the decimated buffer; we then shift in a new input sample and apply the second subfilter in the same way. We repeat this procedure M times to compute the first output. Finally, we reset and commence the computation of the next output. This entire procedure is depicted in Figure 15.9.

A polyphase filter implementation arises in the problem of interpolation as well. By interpolation we mean *increasing* the sampling frequency by an integer factor N. A popular interpolation method is zero insertion, inserting N-1 zeros between every two samples of the original signal x. If we interpret this as a signal of sampling rate  $Nf_s$ , its spectrum under  $\frac{f_s}{2}$  is the same as that of the original signal, but new components appear at higher frequencies. Low-pass filtering this artificially generated signal removes the higher-frequency components, and gives nonzero values to the intermediate samples.

In a straightforward implementation of this idea we first build a new signal  $w_n$  at N times the sampling frequency. For demonstration purposes we take N = 4.

Now the interpolation low-pass filter performs the following convolution.

$y_0$	=	$h_0 w_0$	+	$h_1w_{-1}$	+	$h_2w_{-2}$	+	$h_3w_{-3}$	+	• • •	+	$h_{15}w_{-15}$
$y_1$	=	$h_0w_1$	+	$h_1w_0$	+	$h_2 w_{-1}$	+	$h_{3}w_{-2}$	+	• • •	+	$h_{15}w_{-14}$
$y_2$	=	$h_0w_2$	+	$h_1w_1$	+	$h_2w_0$	+	$h_{3}w_{-1}$	+	• • •	+	$h_{15}w_{-13}$
$y_3$	=	$h_0w_3$	+	$h_1w_2$	+	$h_2w_1$	+	$h_3w_0$	+	• • •	+	$h_{15}w_{-12}$

However, most of the terms in these convolutions are zero, and we can save much computation by ignoring them.

$$y_0 = h_0w_0 + h_4w_{-4} + h_8w_{-8} + h_{12}w_{-12}$$
  

$$= h_0x_0 + h_4x_{-1} + h_8x_{-2} + h_{12}x_{-3}$$
  

$$y_1 = h_1w_0 + h_5w_{-4} + h_9w_{-8} + h_{13}w_{-12}$$
  

$$= h_1x_0 + h_5x_{-1} + h_9x_{-2} + h_{13}x_{-3}$$
  

$$y_2 = h_2w_0 + h_6w_{-4} + h_{10}w_{-8} + h_{14}w_{-12}$$
  

$$= h_2x_0 + h_6x_{-1} + h_{10}x_{-2} + h_{14}x_{-3}$$
  

$$y_3 = h_3w_0 + h_7w_{-4} + h_{11}w_{-8} + h_{15}w_{-12}$$
  

$$= h_3x_0 + h_7x_{-1} + h_{11}x_{-2} + h_{15}x_{-3}$$

Once again this is a polyphase filter, with the input fixed but the subfilters being changed; but this time the absolute time indices of the signal are fixed, not the buffer-relative ones! Moreover, we do not need to add the subfilter outputs; rather each contributes a different output phase. In actual implementations we simply interleave these outputs to obtain the desired



Figure 15.10: The polyphase interpolation filter. We depict the interpolation of an input signal  $x_n$  by a factor of four, using a polyphase filter. Each subfilter operates on the same inputs but with different subfilters, and the outputs are interleaved by zero insertion and delay.

interpolated signal. For diagrammatic purposes we can perform the interleaving by zero insertion and appropriate delay, as depicted in Figure 15.10.

We present this rather strange diagram for two reasons. First, because its meaning is instructive. Rather than zero inserting and filtering at the high rate, we filter at the low rate and combine the outputs. Second, comparison with Figure 15.9 emphasizes the inverse relationship between decimation and interpolation. Transposing the decimation diagram (i.e., reversing all the arrows, changing decimators to zero inserters, etc.) converts it into the interpolation diagram.

Polyphase structures are useful in other applications as well. Decimation and interpolation by large composite factors may be carried out in stages, using polyphase filters at every stage. More general sampling frequency changes by rational factors  $\frac{N}{M}$  can be carried out by interpolating by N and then decimating by M. Polyphase filters are highly desirable in this case as well. Filter banks can be implemented using mixers, narrowband filters, and decimators, and once again polyphase structures reduce the computational load.

#### EXERCISES

- 15.4.1 A commutator is a diagrammatic element that chooses between M inputs  $1 \dots M$  in order. Draw diagrams of the polyphase decimator and interpolator using the commutator.
- 15.4.2 Both 32 KHz and 48 KHz are common sampling frequencies for music, while CDs uses the unusual sampling frequency of 44.1 KHz. How can we convert between all these rates?
- 15.4.3 The simple decimator that extracts inputs with index divisible by M is not a time-invariant system, but rather periodically time varying. Is the entire decimation system of Figure 15.9 time-invariant?
- 15.4.4 Can the polyphase technique be used for IIR filters?
- 15.4.5 When the decimation or interpolation factor M is large, it may be worthwhile to carry out the filtering in stages. For example, assume  $M = M_1 M_2$ , and that we decimate by  $M_1$  and then by  $M_2$ . Explain how to specify filter responses.
- 15.4.6 A half-band filter is a filter whose frequency response obeys the symmetry  $H(\omega) = 1 H(\omega_{mid} \omega)$  around the middle of the band  $\omega_{mid} = \frac{f_s}{4}$ . For every low-pass half-band filter there is a high-pass half-band filter called its 'mirror filter'. Explain how mirror half-band filters can be used to efficiently compute a bank of filters with  $2^m$  bands.

## **15.5** Fixed Point Computation

Throughout this book we stress the advantages of DSP as contrasted with analog processing. In this section we admit that digital processing has a disadvantage as well, one that derives from the fact that only a finite number of bits can be made available for storage of signal values and for computation. In Section 2.7 we saw how digitizing an analog signal inevitably adds quantization noise, due to imprecision in representing a real number by a finite number of bits. However, even if the digitizer has a sufficient number of bits and we ensure that analog signals are amplified such that the digitizer's dynamic range is optimally exploited, we still have problems due to the nature of digital computation.

In general, the sum of two b-bit numbers will have b + 1 bits. When floating point representation (see Appendix A.3) is being used, a (b+1)-bit result can be stored with b bits of mantissa and a larger exponent, causing a slight round-off error. This round-off error can be viewed as a small additional additive noise that in itself may be of little consequence. However, since hundreds of computations may need to be performed the final result may have become hopelessly swamped in round-off noise. Using fixed point representation exacerbates the situation, since should b+1 exceed the fixed number of bits the hardware provides, an *overflow* will occur. To avoid overflow we must ensure that the terms to be added contain fewer bits, reducing dynamic range even when overflow would *not* have occurred. Hence fixed point hardware cannot even consistently exploit the bits it potentially has.

Multiplication is even worse than addition since the product of two numbers with b bits can contain 2b bits. Of course the multiply-and-accumulate (MAC) operation, so prevalent in DSP, is the worst offender of all, endlessly summing products and increasing the number of required bits at each step! This would certainly render all fixed point DSP processors useless, were it not for *accumulators*. An accumulator is a special register with extra bits that is used for accumulating intermediate results. The MAC operation is performed using an accumulator with sufficient bits to prevent overflow; only at the end of the convolution is the result truncated and stored back in a normal register or memory. For example, a 16-bit processor may have a 48-bit accumulator; since each individual product returns a 32-bit result, an FIR filter of length 16 can be performed without prescaling with no fear of overflow.

We can improve our estimate of the required input prescaling if we know the filter coefficients  $a_l$ . The absolute value of the convolution output is

$$|y_n| = |\sum_l h_l x_{n-l}| \le \sum_l |h_l| |x_{n-l}| \le \left(\sum_l |h_l|\right) x_{max}$$

where  $x_{max}$  is the maximal absolute value the input signal takes. In order to ensure that  $y_n$  never overflows in an accumulator of b bits, we need to ensure that the maximal x value does not exceed the following bound.

$$x_{max} \le \frac{2^b}{\sum_l |h_l|} \tag{15.9}$$

This worst-case analysis of the possibility of overflow is often too extreme. The input scaling implied for even modest filter lengths would so drastically reduce the SNR that we are usually willing to risk possible but improbable overflows. Such riskier scaling methods are obtained by replacing the sum of absolute values in equation (15.9) with different combinations of the  $h_l$  coefficients. One commonly used criterion is

$$x_{max} \le \frac{2^b}{\sqrt{\sum_l |h_l|^2}}$$

which results from requiring the output energy to be sufficiently low; another is

$$x_{max} \le \frac{2^b}{H_{max}}$$

where  $H_{max}$  is the maximum value of the filter's frequency response, resulting from requiring that the output doesn't overflow in the frequency domain.

When a result overflow does occur, its effect is hardware dependent. Standard computers usually set an overflow flag to announce that the result is meaningless, and return the meaningless least significant bits. Thus the product of two positive numbers may be negative and the product of two large numbers may be small. Many DSP processors have a *saturation arithmetic* mode, where calculations that overflow return the largest available number of the appropriate sign. Although noise is still added in such cases, its effect is much less drastic. However, saturation introduces clipping nonlinearity, which can give rise to harmonic distortion.

Even when no overflow takes place, digital filters (especially IIR filters) may act quite differently from their analog counterparts. As an example, take the simple AR filter

$$y_n = x_n - 0.9y_{n-1} \tag{15.10}$$

whose true impulse response is  $h_n = (-0.9)^n u_n$ . For simplicity, let's examine the somewhat artificial case of a processor accurate to within one decimal digit after the decimal point (i.e., we'll assume that the multiplication  $0.9y_{n-1}$  is rounded to a single decimal digit to the right of the point). Starting with  $x_0 = 1$  the true output sequence should oscillate while decaying exponentially. However, it is easy to see that under our quantized arithmetic  $-0.9 \cdot -0.4 = +0.4$  and conversely  $-0.9 \cdot 0.4 = -0.4$  so that 0.4, -0.4 is a cycle, called a *limit cycle*. In Figure 15.11 we contrast the two behaviors.

The appearance of a limit cycle immediately calls to mind our study of chaos in Section 5.5, and the relationship is not coincidental. The fixed point arithmetic transforms the initially *linear* recursive system into a nonlinear one, one whose long time behavior displays an attractor that is not a fixed point. Of course, as we learned in that section, the behavior could have been even worse!

There is an alternative way of looking at the generation of the spurious oscillating output. We know that stable IIR filters have all their poles inside the unit circle, and thus cannot give rise to spurious oscillations. However,



Figure 15.11: The behavior of a simple AR filter using fixed point arithmetic. The decaying plot depicts the desired behavior, while the second plot is the behavior that results from rounding to a single digit after the decimal point.

the quantization of the filter coefficients causes the poles to stray from their original positions, and in particular a pole may wander outside the unit circle. Once excited, such a pole causes oscillating outputs even when the input vanishes.

This idea leads us to investigate the effect of coefficient quantization on the position of the filter's poles and zeros, and hence on its transfer function. Let's express the transfer function

$$H(z) = \frac{A(z^{-1})}{B(z^{-1})} = \frac{\sum_{l=0}^{L} a_l z^{-l}}{1 - \sum_{m=1}^{M} b_m z^{-m}} = \frac{\prod_{l=1}^{L} (z - \zeta_l)}{\prod_{m=1}^{M} (z - \pi_m)}$$
(15.11)

and consider the effect of quantizing the  $b_m$  coefficients on the pole positions  $\pi_m$ . The quantization introduces round-off error, so that the effective coefficient is  $b_m + \delta b_m$ , and assuming that this round-off error is small, its effect on the position of pole k may be approximated by the first-order contributions.

$$\delta \pi_k = \sum_{m=1}^M \frac{\partial \pi_k}{\partial b_m} \delta b_m$$

After a bit of calculation we can find that

$$\frac{\partial \pi_k}{\partial b_m} = \frac{\pi_k^{M-m}}{\prod_{\substack{j=1\\j \neq k}}^M (\pi_k - \pi_j)}$$
(15.12)

i.e., the effect of variation of the  $m^{\text{th}}$  coefficient on the  $k^{\text{th}}$  pole depends on the positions of all the poles.

In particular, if the original filter has poles that are close together (i.e., for which  $\pi_k - \pi_i$  is small), small coefficient round-off errors can cause significant movement of these poles. Since close poles are a common occurrence, straightforward implementation of IIR filters as difference equations often lead to instability when fixed point arithmetic is employed. The most common solution to this problem is to implement IIR filters as cascades of subfilters with poles as far apart as possible. Since each subfilter is separately computed, the round-off errors cannot directly interact, and pole movement can be minimized. Carrying this idea to the extreme we can implement IIR filters as cascades of second-order sections, each with a single pair of conjugate poles and a single pair of conjugate zeros (if there are real poles or zeros we use first-order structures). In order to minimize strong gains that may cause overflow we strive to group together zeros and poles that are as close together as possible. This still leaves considerable freedom in the placement order of the sections. Empirically, it seems that the best strategy is to order sections monotonically in the radius of their poles, either from smallest to largest (those nearest the unit circle) or vice versa. The reasoning is not hard to follow. Assume there are poles with very small radius. We wouldn't want to place them first since this would reduce the number of effective bits in the signal early on in the processing, leading to enhanced round-off error. Ordering the poles in a sequence with progressively decreasing radius ameliorates this problem. When there are poles very close to the unit circle placing them first would increase the chance of overflow, or require reducing the dynamic range in order to avoid overflow. Ordering the poles in a sequence with progressively increasing radius is best in this case. When there are both small and large poles it is hard to know which way is better, and it is prudent to directly compare the two alternative orders. Filter design programs that include fixed point optimization routines take such pairing and ordering considerations into account.

#### EXERCISES

15.5.1 A pair of conjugate poles with radius r < 1 and angles  $\pm \theta$  contribute a second-order section

$$(z - re^{i\theta})(z - re^{-i\theta}) = z^2 \left(1 - 2r\cos\theta z^{-1} + r^2 z^{-2}\right)$$

with coefficients  $b_1 = 2r \cos \theta$  and  $b_2 = -r^2$ . If we quantize these coefficients to b bits each, how many distinct pole locations are possible? To how many bits has the radius r been quantized? Plot all the possible poles for 4-8 bits. What can you say about the quantization of real poles?

- 15.5.2 As we discussed in Section 14.6, fixed point FFTs are vulnerable to numerical problems as well. Compare the accuracy and overflow characteristics of frequency domain and time domain filtering.
- 15.5.3 Develop a strategy to eliminate limit cycles, taking into account that limit cycles can be caused by round-off or overflow errors.
- 15.5.4 Complete the derivation of the dependence of  $\pi_k$  on  $\delta b_m$ .
- 15.5.5 What can you say about the dependence of zero position  $\zeta_l$  on small changes in numerator coefficients  $a_l$ ? Why do you think fixed point FIR filters are so often computed in direct form rather than cascade form?
- 15.5.6 We saw that it is possible to prescale the input in order to ensure that an FIR filter will never overflow. Is it possible to guarantee that an IIR filter will not overflow?
- 15.5.7 In the text we saw a system whose impulse response should have decayed to zero, but due to quantization was a 2-cycle. Find a system whose impulse response is a nonzero constant. Find a system with a 4-cycle. Find a system that goes into oscillation because of overflow.

## 15.6 IIR Structures

We return now to structures for general filters and consider the case of IIR filters. We already saw how to diagram the most general IIR filter in Figures 12.8.B and 12.11, but know from the previous section that this direct form of computation is not optimal from the numerical point of view. In this section we will see better approaches.

The general cascade of second-order IIR sections is depicted in Figure 15.12. Each section is an independent first- or second-order ARMA filter, with its own coefficients and static memory. The only question left is how to best implement this second-order section. There are three different structures in common use: the direct form (also called the *direct form I*) depicted in Figure 15.13, the canonical form (also called *direct form II*) depicted in Figure 15.14, and the transposed form (also called *transposed form II*) depicted in Figure 15.15. Although all three are valid implementations of precisely the same filter, numerically they may give somewhat different results.



Figure 15.12: General cascade implementation of an IIR filter. Each section implements an independent (first- or) second-order section symbolized by the transfer function appearing in the rectangle. Note that a zero in any of these subfilters results in a zero of the filter as a whole.



Figure 15.13: Direct form implementation of a second-order IIR section. This structure is derived by placing the MA (all-zero) filter before the AR (all-pole) one.



Figure 15.14: Canonical form implementation of a second-order IIR section. This structure is derived by placing the AR (all-pole) filter before the MA (all-zero) one and combining common elements. (Why didn't we draw a filled circle for  $w_{n-2}^{[k]}$ ?)



Figure 15.15: Transposed form implementation of a second-order IIR section. Here only the intermediate variables are delayed. Although only three adders are shown the center one has three inputs, and so there are actually four additions.

An IIR filter implemented using direct form sections is computed as follows:

In real-time applications the loop over time will normally be an infinite loop. Each new input sample is first MA filtered to give the intermediate signal  $w_n^{[0]}$ 

$$w_n^{[0]} = a_0^{[0]} x_n + a_1^{[0]} x_{n-1} + a_2^{[0]} x_{n-2}$$

and then this signal is AR filtered to give the section's output

$$y_n^{[0]} = w_n^{[0]} - b_1^{[0]} y_{n-1}^{[0]} - b_2^{[0]} y_{n-2}^{[0]}$$

the subtraction either being performed once, or twice, or negative coefficients being stored. This section output now becomes the input to the next section

$$x_n^{[1]} \leftarrow y_n^{[0]}$$

and the process repeats until all K stages are completed. The output of the final stage is the desired result.

$$y_n = y_n^{[K-1]}$$

Each direct form stage requires five multiplications, four additions, and four delays. In the diagrams we have emphasized memory locations that have to be stored (static memory) by a circle. Note that  $w_n$  is generated each time and does not need to be stored, so that there are only two saved memory locations.

As we saw in Section 12.3 we can reverse the order of the MA and AR portions of the second-order section, and then regroup to save memory locations. This results in the structure known as canonical (meaning 'accepted' or 'simplest') form, an appellation well deserved because of its use of the least number of delay elements. While the direct form requires delayed versions of both  $x_n$  and  $y_n$ , the canonical form only requires storage of  $w_n$ .

The computation is performed like this

and once again we can either stored negative b coefficients or perform subtraction(s). Each canonical form stage requires five multiplications, four additions, two delays, and two intermediate memory locations.

The transposed form is so designated because it can be derived from the canonical form using the *transposition theorem*, which states that reversing all the arc directions, changing adders to tee connections and vice-versa, and interchanging the input and output does not alter the system's transfer function. It is also canonical in the sense that it also uses only two delays, but we need to save a single value of two different signals (which we call  $u_n$  and  $v_n$ ), rather than two lags of a single intermediate signal. The full computation is

Don't be fooled by Figure 15.15 into thinking that there are only three additions in the transposed form. The center adder is a three-input adder, which has to be implemented as two separate additions. Hence the transposed form requires five multiplications, four additions, two delays, and two intermediate memory locations, just like the canonical form.

The cascade forms we have just studied are numerically superior to direct implementation of the difference equation, especially when pole-zero pairing and ordering are properly carried out. However, the very fact that the signal has to travel through section after section in series means that round-off



Figure 15.16: *Parallel form* implementation of the IIR filter. In this form the subfilters are placed in parallel, and so round-off errors do not accumulate. Note that a pole in any of these subfilters results in a pole of the filter as a whole.

errors accumulate. Parallel connection of second-order sections, depicted in Figure 15.16, is an alternative implementation of the general IIR filter that does not suffer from round-off accumulation. The individual sections can be implemented in direct, canonical, or transposed form; and since the outputs are all simply added together, it is simpler to estimate the required number of bits.

The second-order sections in cascade form are guaranteed to exist by the fundamental theorem of algebra, and are found in practice by factoring the system function. Why are general system functions expressible as sums of second-order filters, and how can we perform this decomposition? The secret is the 'partial fraction expansion' familiar to all students of indefinite integration. Using partial fractions, a general system function can be written as the sum of first-order sections

$$H(z) = \sum_{k=1}^{K} \frac{\Gamma_k}{1 + \gamma_k z^{-1}}$$
(15.13)

with  $\Gamma_k$  and  $\gamma_k$  possibly complex, or as the sum of second-order sections

$$H(z) = \sum_{k=1}^{K} \frac{A_k + B_k z^{-1}}{1 + \alpha_k z^{-1} + \beta_k z^{-2}}$$
(15.14)

with all coefficients real. If there are more zeros than poles in the system function, we need an additional FIR filter in parallel with the ARMA sections.

The decomposition is performed in practice by factoring the denominator of the system function into real first- and second-order factors, writing the partial fraction expansion, and comparing. For example, assume that the system function is

$$H(z) = \frac{1 + az^{-1} + bz^{-2}}{(1 + cz^{-1})(1 + dz^{-1} + ez^{-2})}$$

then we write

$$H(z) = \frac{A}{1+cz^{-1}} + \frac{B+Cz^{-1}}{1+dz^{-1}+ez^{-2}}$$
  
= 
$$\frac{(A+B) + (Ad+Bc+C)z^{-1} + (Ae+Cc)z^{-2}}{(1+cz^{-1})(1+dz^{-1}+ez^{-2})}$$

and compare. This results in three equations for the three variables A, B, and C.

#### EXERCISES

- 15.6.1 An arbitrary IIR filter can always be factored into cascaded first-order sections, if we allow complex-valued coefficients. Compare real-valued secondorder sections with complex-valued first-order sections from the points of view of computational complexity and numerical stability.
- 15.6.2 A second-order all-pass filter section has the following transfer function.

$$\frac{c + dz^{-1} + z^{-2}}{1 + dz^{-1} + cz^{-2}}$$

Diagram it in direct form. How many multiplications are needed? Redraw the section emphasizing this.

- 15.6.3 Apply the transposition theorem to the direct form to derive a noncanonical transposed section.
- 15.6.4 The lattice structure presented for the FIR filter in Section 15.3 can be used for IIR filters as well. Diagram a two-pole AR filter. How can lattice techniques be used for ARMA filters?

## 15.7 FIR vs. IIR

Now that we have seen how to implement both FIR and IIR filters, the question remains as to which to use. Once again we suggest first considering whether it is appropriate to filter in the frequency domain. Frequency domain filtering is almost universally applicable, is intrinsically stable, and the filter designer has complete control over the phase response. The run-time code is often the most computationally efficient technique, and behaves well numerically if the FFTs are properly scaled. Phase response is completely controllable. Unfortunately, it *does* introduce considerable buffer and algorithmic delay; it *does* require more complex code; and it possibly requires more table and scratch memory. Of course we cannot really multiply any frequency component by infinity and so true poles on the frequency axis are not implementable, but IIR filters with such poles would be unstable anyway.

Assuming you have come to the conclusion that time domain filtering is appropriate, the next question has to do with the type of filter that is required. Special filters (see Section 7.3) have their own special considerations. In general, integrators should be IIR, differentiators even order FIR (unless the half sample delay is intolerable), Hilbert transforms odd order FIR with half the coefficients zero (although IIR designs are possible), decimators and integrators should be polyphase FIR, etc. Time-domain filter specifications immediately determine the FIR filter coefficients, but can also be converted into an IIR design by Prony's method (see Section 13.6). When the sole specification is one of the standard forms of Section 7.1, such as low-pass, IIR filters can be readily designed while optimal FIR designs require more preparation. If the filter design must be performed in run-time then this will often determine the choice of filter type. Designing a standard IIR filter reduces to a few equations, and the suboptimal windowing technique for designing FIR filters can sometimes be used as well. From now on we'll assume that we have a constant prespecified frequency domain specification.

It is important to determine whether a true linear-phase filter or only a certain degree of phase linearity is required (e.g., communications signals that contain information in their phase, or simultaneous processing of multiple signals that will later be combined). Recall from Section 7.2 that symmetric or antisymmetric FIR filters are precisely linear-phase, while IIR filters can only approximate phase linearity. However, IIR filters can have their phase flattened to a large degree, and if sufficient delay is allowed the pseudo-IIR filter of exercise 7.2.5 may be employed for precise phase linearity.

Assuming that both FIR and IIR filters are still in the running (e.g., only the amplitude of the frequency response is of interest), the issue of computational complexity is usually the next to be considered. IIR filters with a relatively small number of coefficients can be designed to have very sharp frequency response transitions (with the phase being extremely nonlinear near these transitions) and very strong stop-band attenuation. For a given specification elliptical IIR filters will usually have dramatically lower computational complexity than FIR filters, with the computational requirements ratio sometimes in the thousands. Only if the filters are relatively mild and when a large amount of pass-band ripple can be tolerated will the computational requirements be similar or even in favor of the FIR. Chebyshev IIR filters are less efficient than elliptical designs but still usually better performers than FIR filters. Butterworth designs are the least flexible and hence require the highest order and the highest computational effort. If phase linearity compensation is attempted for a Butterworth IIR filter the total computational effort may be comparable to that of an FIR filter.

The next consideration is often numerical accuracy. It is relatively simple to determine the worst-case number of bits required for overflow-free FIR computation, and if sufficient bits are available in the accumulator and the quantized coefficients optimized, the round-off error will be small. Of course long filters and small registers will force us to prescale down filter coefficients or input signals causing 6 dB of SNR degradation for each lost bit. For IIR filters determining the required number of bits is much more complex, depending on the filter characteristics and input signal frequency components. FIR filters are inherently stable, while IIR filters may be unstable or may become unstable due to numerical problems. This is of overriding importance for filters that must be varied as time goes on; an IIR filter must be continuously monitored for stability (possibly a computationally intensive task in itself) while FIR filters may be used with impunity.

Finally, all things being equal, personal taste and experience comes into play. Each DSP professional accumulates over time a bag of fully honed and well-oiled tools. It is perfectly legitimate that the particular tool that 'feels right' to one practitioner may not even be considered by another. The main problem is that when you have only a hammer every problem looks like a nail. We thus advise that you work on as many different applications as possible, collecting a tool or two from each.

# **Bibliographical Notes**

Most general DSP texts, e.g., [186, 200] and Chapters 6 and 7 of [241], cover digital filter structures to some degree. Also valuable are libraries and manuals that accompany specific DSP processors.

The idea of using the DFT to compute linear convolutions appears to have been invented simultaneously at MIT [124], at Bell Labs [100] and by Sande at Princeton.

Polyphase filtering was developed extensively at Bell Labs, and a good review of polyphase filters for interpolation and decimation is [48].

The effect of numerical error on filters has an extensive bibliography, e.g., [151, 215, 114, 115].