Digital Signal Processing: A Computer Science Perspective
Jonathan Y. Stein
Copyright © 2000 John Wiley & Sons, Inc.
Print ISBN 0-471-29546-9 Online ISBN 0-471-20059-X

14

The Fast Fourier Transform

It is difficult to overstate the importance of the FFT algorithm for DSP. We have often seen the essential duality of signals in our studies so far; we know that exploiting both the time and the frequency aspects is critical for signal processing. We may safely say that were there not a fast algorithm for going back and forth between time and frequency domains, the field of DSP as we know it would never have developed.

The discovery of the first FFT algorithm predated the availability of hardware capable of actually exploiting it. The discovery dates from a period when the terms *calculator* and *computer* referred to people, particularly adept at arithmetic, who would perform long and involved rote calculations for scientists, engineers, and accountants. These *computers* would often exploit symmetries in order to save time and effort, much as a contemporary programmer exploits them to reduce electronic computer run-time and memory. The basic principle of the FFT ensues from the search for such time-saving mechanisms, but its discovery also encouraged the development of DSP hardware. Today's DSP chips and special-purpose FFT processors are children of both the microprocessor age and of the DSP revolution that the FFT instigated.

In this chapter we will discuss various algorithms for calculating the DFT, all of which are known as *the* FFT. Without a doubt the most popular algorithms are radix-2 DIT and DIF, and we will cover these in depth. These algorithms are directly applicable only for signals of length $N = 2^m$, but with a little ingenuity other lengths can be accommodated. Radix-4, split radix, and FFT842 are even faster than basic radix-2, while mixed-radix and prime factor algorithms directly apply to N that are not powers of two. There are special cases where the fast Fourier transform can be made even faster. Finally we present alternative algorithms that in specific circumstances may be faster than the FFT.

14.1 Complexity of the DFT

Let us recall the previously derived formula (4.32) for the N-point DFT

$$X_k = \sum_{n=0}^{N-1} x_n W_N^{nk}$$

where the N^{th} root of unity is defined as

$$W_N \equiv e^{-\mathrm{i}\frac{2\pi}{N}} = \cos\left(\frac{2\pi}{N}\right) - \mathrm{i}\sin\left(\frac{2\pi}{N}\right)$$

How many calculations must we perform to find one X_k from a set of N time domain values x_n? Assume that the complex constant W_N and its powers W_N^j have all been precalculated and stored for our use. Looking closely at (4.32) we see N complex multiplications and $N-1$ complex additions are performed in the loop on n. Now to find the *entire* spectrum we need to do N such calculations, one for each value of k. So we expect to have to carry out N^2 complex multiplications, and $N(N-1)$ complex additions.

This is actually a slight overestimate. By somewhat trickier programming we can take advantage of the fact that $W_N^0 = 1$, so that each of the $X_{k>0}$ takes $N-1$ multiplications and additions, while X_0 doesn't require any multiplications. We thus really need only $(N-1)^2$ complex multiplications.

A complex addition requires the addition of real and imaginary parts, and is thus equivalent to two real additions. A complex multiplication can be performed as four real multiplications and two additions $(a + \mathrm{i}b)(c + \mathrm{i}d) = (ac - bd) + \mathrm{i}(bc + ad)$ or as three multiplications and five additions $(a+\mathrm{i}b)(c+\mathrm{i}d) = a(c+d) - d(a+b) + \mathrm{i}\Big(a(c+d) + c(b-a)\Big)$. The latter form may be preferred when multiplication takes much more time than addition, but can be less stable numerically. Other combinations are possible, but it can be shown that there is no general formula for the complex product with less than three multiplications. Using the former, more common form, we find that the computation of the entire spectrum requires $4(N-1)^2$ real multiplications and $2(N-1)(2N-1)$ real additions.

Actually, the calculation of a single X_k can be performed more efficiently than we have presented so far. For example, Goertzel discovered a method of transforming the iteration in equation (4.32) into a recursion. This has the effect of somewhat reducing the computational complexity and also saves the precomputation and storage of the W table. Goertzel's algorithm, to be presented in Section 14.8, still has asymptotic complexity of order $O(N)$ per calculated X_k, although with a somewhat smaller constant than the direct

method. It thus leaves the complexity of calculation of the entire spectrum at $O(N^2)$, while the FFT algorithms to be derived presently are less complex. Goertzel's algorithm thus turns out to be attractive when a single, or only a small number of X_k values are needed, but is not the algorithm of choice for calculating the entire spectrum.

Returning to the calculation of the entire spectrum, we observe that both the additions and multiplications increase as $O(N^2)$ with increasing N. Were this direct calculation the only way to find the DFT, real-time calculation of large DFTs would be impractical. It is general rule in DSP programming that only algorithms with linear asymptotic complexity can be performed in real-time for large N. Let us now see why this is the case.

The criterion for real-time calculation is simple for algorithms that process a single input sample at a time. Such an algorithm must finish all its computation for each sample before the next one arrives. This restricts the number of operations one may perform in such computations to the number performable in a sample interval t_s. This argument does not directly apply to the DFT since it is inherently a block-oriented calculation. One cannot perform a DFT on a single sample, since frequency is only defined for signals that occupy some nonzero interval of time; and we often desire to process large blocks of data since the longer we observe the signal the more accurate frequency estimates will be.

For block calculations one accumulates samples in an array, known as a buffer, and then processes this buffer as a single entity. A technique known as *double-buffering* is often employed in real-time implementations of block calculations. With double-buffering two buffers are employed. While the samples in the *processing buffer* are being processed, the *acquisition buffer* is acquiring samples from the input source. Once processing of the first buffer is finished and the output saved, the buffers are quickly switched, the former now acquiring samples and the latter being processed.

How can we tell if block calculations can be performed in real-time? As for the single sample case, one must be able to finish all the processing needed in time. Now 'in time' means completing the processing of one entire buffer, before the second buffer becomes full. Otherwise a condition known as *data-overrun* occurs, and new samples overwrite previously stored, but as yet unprocessed, ones. It takes $N\Delta t$ seconds for N new samples to arrive. In order to keep up we must process all N old samples in the processing buffer before the acquisition buffer is completely filled. If the complexity is linear (i.e., the processing time for N samples is proportional to N), then $C = qN$ for some q. This q is the *effective time per sample* since each sample effectively takes q time to process, independent of N. Thus, the selection of

buffer size is purely a memory issue, and does not impact the ability to keep up with real-time. However, if the complexity is superlinear (for example, $T_{processing} = qN^p$ with $p > 1$), then as N increases we have less and less time to process each sample, until eventually some N is reached where we can no longer keep up, and data-overrun is inevitable.

Let's clarify this by plugging in some numbers. Assume we are acquiring input at a sample rate of 1000 samples per second (i.e., we obtain a new sample every millisecond) and are attempting to process blocks of length 250. We start our processor, and for one-quarter of a second, we cannot do any processing, until the first acquisition buffer fills. When the buffer is full we quickly switch buffers, start processing the 250 samples collected, while the second buffer of length 250 fills. We must finish the processing within a quarter of second, in order to be able to switch buffers back when the acquisition buffer is full. When the dependence of the processing time on buffer length is strictly linear, $T_{processing} = qN$, then if we can process a buffer of $N = 250$ samples in 250 milliseconds or less, we can equally well process a buffer of 500 samples in 500 milliseconds, or a buffer of $N = 1000$ samples in a second. Effectively we can say that when the single sample processing time is no more than $q = 1$ millisecond per sample, we can maintain real-time processing.

What would happen if the buffer processing time depended quadratically on the buffer size—$T_{processing} = qN^2$? Let's take q to be 0.1 millisecond per sample squared. Then for a small 10-millisecond buffer (length $N = 10$), we will finish processing in $T_{processing} = 0.1 \cdot 10^2 = 10$ milliseconds, just in time! However, a 100-millisecond buffer of size $N = 100$ will require $T_{processing} = 0.1 \cdot 100^2$ milliseconds, or one second, to process. Only by increasing our computational power by a factor of ten would we be able to maintain real-time! However, even were we to increase the CPU power to accommodate this buffer-size, our 250-point buffer would still be out of our reach.

As we have mentioned before, the FFT is an algorithm for calculating the DFT more efficiently than quadratically, at least for certain values of N. For example, for powers of two, $N = 2^k$, its complexity is $O(N \log_2 N)$. This is only very slightly superlinear, and thus while *technically* the FFT is not suitable for real-time calculation in the asymptotic $N \to \infty$ limit, in practice it *is* computable in real-time even for relatively large N. To grasp the speed-up provided by the FFT over direct calculation of (4.29), consider that the ratio between the complexities is proportional to $\frac{N}{\log_2 N}$. For $N = 2^4 = 16$ the FFT is already four times faster than the direct DFT, for $N = 2^{10} = 1024$

it is over one hundred times faster, and for $N = 2^{16}$ the ratio is 4096! It is common practice to compute $1K$- or $64K$-point FFTs in real-time, and even much larger sizes are not unusual.

The basic idea behind the FFT is the very exploitation of the N^2 complexity of the direct DFT calculation. Due to this second-order complexity, it is faster to calculate a lot of small DFTs than one big one. For example, to calculate a DFT of length N will take N^2 multiplications, while the calculation of two DFTs of length $\frac{N}{2}$ will take $2(\frac{N}{2})^2 = \frac{N^2}{2}$, or half that time. Thus if we can somehow piece the two partial results together to one spectrum in less than $\frac{N^2}{2}$ time then we have found a way to save time. In Sections 14.3 and 14.4 we will see several ways to do just that.

EXERCISES

14.1.1 Finding the maximum of an N-by-N array of numbers can be accomplished in $O(N^2)$ time. Can this be improved by partitioning the matrix and exploiting the quadratic complexity as above?

14.1.2 In exercise 4.7.4 you found explicit equations for the $N = 4$ DFT for $N = 4$. Count up the number of *complex* multiplications and additions needed to compute X_0, X_1, X_2, and X_3. How many *real* multiplications and additions are required?

14.1.3 Define temporary variables that are used more than once in the above equations. How much can you save? How much memory do you need to set aside? (Hint: Compare the equations for X_0 and X_2.)

14.1.4 Up to now we have not taken into account the task of finding the trigonometric W factors themselves, which can be computationally intensive. Suggest at least two solutions, one that requires a large amount of auxiliary memory but practically no CPU, and one that requires little memory but is more CPU intensive.

14.1.5 A computational system is said to be 'real-time-oriented' when the time it takes to perform a task can be guaranteed. Often systems rely on the weaker criterion of *statistical real-time*, which simply means that on-the-average enough computational resources are available. In such cases double buffering can be used in the acquisition hardware, in order to compensate for peak MIPS demands. Can hardware buffering truly make an arbitrary system as reliable as a real-time-oriented one?

14.1.6 Explain how double-buffering can be implemented using a single circular buffer.

14.2 Two Preliminary Examples

Before deriving the FFT we will prepare ourselves by considering two somewhat more familiar examples. The ideas behind the FFT are very general and not restricted to the computation of equation (14.1). Indeed the two examples we use to introduce the basic ideas involve no DSP at all.

How many comparisons are required to find the maximum or minimum element in a sequence of N elements? It is obvious that $N-1$ comparisons are absolutely needed if all elements are to be considered. But what if we wish to simultaneously find the maximum *and* minimum? Are twice this number really needed? We will now show that we can get away with only $1\frac{1}{2}$ times the number of comparisons needed for the first problem. Before starting we will agree to simplify the above number of comparisons to N, neglecting the 1 under the asymptotic assumption $N \gg 1$.

A fundamental tool employed in the reduction of complexity is that of splitting long sequences into smaller subsequences. How can we split a sequence with N elements

$$x_0,\ x_1,\ x_2,\ x_3,\ \ldots\ x_{N-2},\ x_{N-1}$$

into two subsequences of half the original size (assume for simplicity's sake that N is even)? One way is to consider pairs of adjacent elements, such as x_1, x_2 or x_3, x_4, and place the smaller of each pair into the first subsequence and the larger into the second. For example, assuming $x_0 < x_1$, $x_2 > x_3$ and $x_{N-2} < x_{N-1}$, we obtain

$$\begin{array}{llll} x_0\ x_3 & \ldots\ \min(x_{2l}, x_{2l+1}) & \ldots & x_{N-2} \\ x_1\ x_2 & \ldots\ \max(x_{2l}, x_{2l+1}) & \ldots & x_{N-1} \end{array}$$

This splitting of the sequence requires $\frac{N}{2}$ comparisons. Students of sorting and searching will recognize this procedure as the first step of the *Shell sort*.

Now, the method of splitting the sequence into subsequences guarantees that the minimum of the entire sequence must be one of the elements of the first subsequence, while the maximum must be in the second. Thus to complete our search for the minimum and maximum of the original sequence, we must find the minimum of the first subsequence and the maximum of the second. By our previous result, each of these searches requires $\frac{N}{2}$ comparisons. Thus the entire process of splitting and two searches requires $\frac{N}{2} + 2\frac{N}{2} = \frac{3N}{2}$ comparisons, or $1\frac{1}{2}$ times that required for the minimum or maximum alone.

Can we further reduce this factor? What if we divide the original sequence into adjacent quartets, choosing the minimum and maximum of the

four? The splitting would then cost four comparisons per quartet, or N comparison altogether, and then two $\frac{N}{4}$ searches must be carried out. Thus we require $N + 2\frac{N}{4}$ and a factor of $1\frac{1}{2}$ is still needed. Indeed, after a little reflection, the reader will reach the conclusion that no further improvement is possible. This is because the new problems of finding only the minimum *or* maximum of a subsequence are simpler than the original problem.

When a problem *can* be reduced recursively to subproblems *similar* to the original, the process may be repeated to attain yet further improvement. We now discuss an example where such recursive repetition is possible. Consider multiplying two $(N+1)$-digit numbers A and B to get a product C using long multiplication (which from Section 6.8 we already know to be a convolution).

$$
\begin{array}{cccccccc}
 & & & A_N & A_{N-1} & \cdots & A_1 & A_0 \\
 & & & B_N & B_{N-1} & \cdots & B_1 & B_0 \\
\hline
 & & & B_0A_N & B_0A_{N-1} & \cdots & B_0A_1 & B_0A_0 \\
 & & B_1A_N & B_1A_{N-1} & & \cdots & B_1A_0 & \\
 & & & & & \vdots & & \\
B_NA_N & \cdots & B_NA_1 & B_NA_0 & & & & \\
\hline
C_{2N} & \cdots & C_{N+1} & C_N & C_{N-1} & \cdots & C_1 & C_0
\end{array}
$$

Since we must multiply every digit in the top number by every digit in the bottom number, the number of one-digit multiplications is N^2. You are probably used to doing this for decimal digits, but the same multiplication algorithm can be utilized for N-bit binary numbers. The hardware-level complexity of straightforward multiplication of two N-bit numbers is proportional to N^2.

Now assume N is even and consider the left $\frac{N}{2}$ digits and the right $\frac{N}{2}$ digits of A and B separately. It does not require much algebraic prowess to convince oneself that

$$
\begin{aligned}
A &= A_L 2^{\frac{N}{2}} + A_R \\
B &= B_L 2^{\frac{N}{2}} + B_R \\
C &= A_L B_L 2^N + (A_L B_R + A_R B_L) 2^{\frac{N}{2}} + A_R B_R \\
&= A_L B_L (2^N + 2^{\frac{N}{2}}) + (A_L - A_R)(B_R - B_L) 2^{\frac{N}{2}} + A_R B_R (2^{\frac{N}{2}} + 1)
\end{aligned}
\tag{14.1}
$$

involving only three multiplications of $\frac{N}{2}$-length numbers. Thus we have reduced the complexity from N^2 to $3(\frac{N}{2})^2 = \frac{3}{4}N^2$ (plus some shifting and adding operations). This is a savings of 25%, but does not reduce the asymptotic form of the complexity from $O(N^2)$. However, in this case we have only

just begun! Unlike for the previous example, we have reduced the original multiplication problem to three similar but simpler multiplication problems!

We can now carry out the three $\frac{N}{2}$-bit multiplications in equation (14.1) similarly (assuming that $\frac{N}{2}$ is still even) and continue recursively. Assuming N to have been a power of two, we can continue until we multiply individual bits. This leads to an algorithm for multiplication of two N-bit numbers, whose asymptotic complexity is $O(N^{\log_2(3)}) \approx O(N^{1.585})$. The slightly more sophisticated Toom-Cook algorithm divides the N-bit numbers into more than two groups, and its complexity can be shown to be $O(N \log(N) 2^{\sqrt{2\log(N)}})$. This is still not the most efficient way to multiply numbers. Realizing that each column sum of the long multiplication in equation (14.1) can be cast into the form of a convolution, it turns out that the best way to multiply large numbers is to exploit the FFT!

EXERCISES

14.2.1 The reader who has implemented the Shell sort may have used a different method of choosing the pairs of elements to be compared. Rather than comparing adjacent elements x_{2l} and x_{2l+1}, it is more conventional to consider elements in the same position in the first and second halves the sequence, x_k and $x_{\frac{N}{2}+k}$ Write down a general form for the new sequence. How do we find the minimum and maximum elements now? These two ways of dividing a sequence into two subsequences are called *decimation* and *partition.*

14.2.2 Devise an algorithm for finding the *median* of N numbers in $O(N \log N)$.

14.2.3 The product of two two-digit numbers, ab and cd, can be written $ab * cd = (10 * a + b) * (10 * c + d) = 100ac + 10(ad + bc) + bd$. Practice multiplying two-digit numbers in your head using this rule. Try multiplying a three-digit number by a two-digit one in similar fashion.

14.2.4 We often deal with complex-valued signals. Such signals can be represented as vectors in two ways, *interleaved*

$$\Re(x_1)\Im(x_1), \Re(x_2), \Im(x_2), \ldots \Re(x_N), \Im(x_N)$$

or *separated*

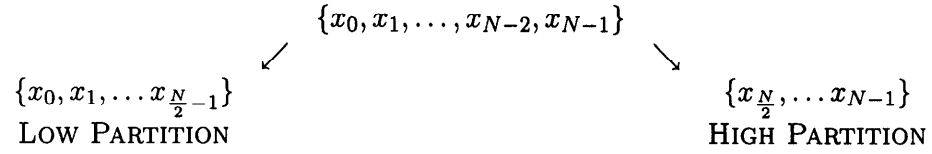
$$\Re(x_1)\Re(x_2), \ldots \Re(x_N), \Im(x_1), \Im(x_2), \ldots \Im(x_N)$$

Devise an efficient in-place algorithm changing between interleaved and separated representations. Efficient implies that each element accessed is moved immediately to its final location. In-place means here that if extra memory is used it must be of constant size (independent of N). What is the algorithm's complexity?

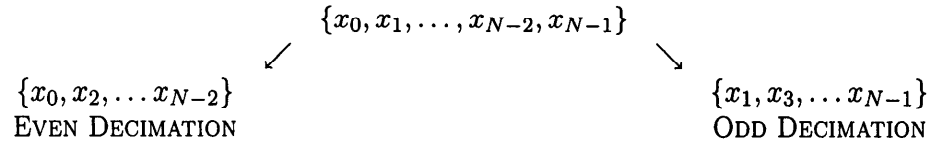
14.3 Derivation of the DIT FFT

Without further ado, we turn to the derivation of the our first FFT algorithm. As mentioned in the previous section, we want to exploit the fact that it is better to compute many small DFTs than a single large one; as a first step let's divide the sequence of signal values into two equal parts.

There are two natural ways of methodically dividing a sequence into two subsequences, *partition* and *decimation*. By partition we mean separating the sequence at the half-way point



while decimation is the separation of the even-indexed from odd-indexed elements.



Put another way, partition divides the sequence into two groups according to the MSB of the index, while decimation checks the LSB.

Either partition and decimation may be employed to separate the original signal into half-sized signals for the purpose of computation reduction. Decimation in time implies partition in frequency (e.g., doubling the time *duration* doubles the frequency *resolution*), while partition in time signifies decimation in frequency. These two methods of division lead to somewhat different FFT algorithms, known conventionally as the **D**ecimation **I**n **T**ime (DIT) and **D**ecimation **I**n **F**requency (DIF) FFT algorithms. We will here consider *radix-2* partition and decimation, that is, division into *two* equal-length subsequences. Other partition and decimation radixes are possible, leading to yet further FFT algorithms.

We will now algebraically derive the radix-2 DIT algorithm. We will need the following trigonometric identities.

$$\begin{aligned} W_N^N &= 1 \\ W_N^{\frac{N}{2}} &= -1 \\ W_N^2 &= W_{\frac{N}{2}} \end{aligned} \tag{14.2}$$

The FFT's efficiency results from the fact that in the complete DFT many identical multiplications are performed multiple times. As a matter of fact, in X_k and $X_{k+\frac{N}{2}}$ all the multiplications are the same, although every other addition has to be changed to a subtraction.

$$\begin{aligned} X_k &= \sum_{n=0}^{N-1} x_n W_N^{nk} \\ X_{k+\frac{N}{2}} &= \sum_{n=0}^{N-1} x_n W_N^{nk} W_N^{\frac{Nn}{2}} \\ &= \sum_{n=0}^{N-1} x_n W_N^{nk} (-1)^n \end{aligned}$$

The straightforward computation of the entire spectrum ignores this fact and hence entails wasteful recalculation.

In order to derive the DIT algorithm we separate the sum in (4.32) into sums over even- and odd-indexed elements, utilizing the identities (14.2)

$$\begin{aligned} X_k &= \sum_{n=0}^{N-1} x_n W_N^{nk} = \sum_{n=0}^{\frac{N}{2}-1} \left(x_{2n} W_N^{2nk} + x_{2n+1} W_N^{(2n+1)k} \right) \\ &= \sum_{n=0}^{\frac{N}{2}-1} x_n^E W_{\frac{N}{2}}^{nk} + W_N^k \sum_{n=0}^{\frac{N}{2}-1} x_n^O W_{\frac{N}{2}}^{nk} \end{aligned} \tag{14.3}$$

where we have defined even and odd subsequences.

$$\begin{aligned} x_n^E &= x_{2n} \qquad \text{for } n = 0, 1, \ldots \frac{N}{2} - 1 \\ x_n^O &= x_{2n+1} \end{aligned}$$

After a moment of contemplation it is apparent that the first term is the DFT of the even subsequence, while the second is W_N^k times the DFT of the odd subsequence; therefore we have discovered a recursive procedure for computing the DFT given the DFTs of the even and odd decimations.

Recalling the relationship between X_k and $X_{k+\frac{N}{2}}$ we can immediately write

$$X_{k+\frac{N}{2}} = \sum_{n=0}^{\frac{N}{2}-1} x_n^E W_{\frac{N}{2}}^{nk} - W_N^k \sum_{n=0}^{\frac{N}{2}-1} x_n^O W_{\frac{N}{2}}^{nk}$$

this being the connection between parallel elements in different *partitions* of the frequency domain.

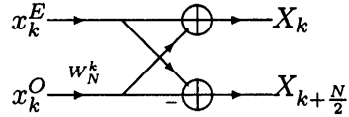
Now let us employ the natural notation

$$X_k^E = \sum_{n=0}^{\frac{N}{2}-1} x_n^E W_{\frac{N}{2}}^{nk} \qquad X_k^O = \sum_{n=0}^{\frac{N}{2}-1} x_n^O W_{\frac{N}{2}}^{nk}$$

and write our results in the succinct recursive form

$$\begin{aligned} X_k &= X_k^E + W_N^k X_k^O \\ X_{k+\frac{N}{2}} &= X_k^E - W_N^k X_k^O \end{aligned} \tag{14.4}$$

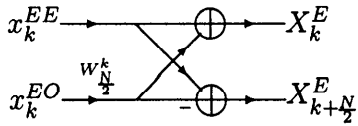
which is a computational topology known as a *butterfly* (for reasons soon to be apparent). In this context the W_N^k factor is commonly called a *twiddle factor* (for reasons that we won't adequately explain). We note that the butterfly is basically an in-place computation, replacing two values with two new ones. Using our standard graphical notation we can depict equation (14.4) in the following way:



which, not accidentally, is very similar to the two-point DFT diagram presented in Section 12.2 (actually, remembering the second identity of (14.2) we recognize the two-point FFT as a special case of the DIT butterfly). Rotating the diagram by 90° and using some imagination clarifies the source of the name 'butterfly'. We will soon see that the butterfly is the only operation needed in order to perform the FFT, other than calculation (or table lookups) of the twiddle factors.

Now, is this method of computation more efficient than the straightforward one? Instead of $(N-1)^2$ multiplications and $N(N-1)$ additions for the simultaneous computation of X_k for all k, we now have to compute two $\frac{N}{2}$-point DFTs, one additional multiplication (by the twiddle factor), and two new additions, for a grand total of $2(\frac{N}{2}-1)^2+1 = \frac{N^2}{2}-2N+3$ multiplications and $2\left(\frac{N}{2}(\frac{N}{2}-1)\right)+2 = \frac{N^2}{2}-N+2$ additions. The savings may already be significant for large enough N!

But why stop here? We are assuming that X_k^E and X_k^O were computed by the straightforward DFT formula! We can certainly save on their computation by using the recursion as well! For example, we can find X_k^E by the following butterfly.



As in any recursive definition, we must stop somewhere. Here the obvious final step is the reduction to a two-point DFT, computed by the simplest butterfly. Thus the entire DFT calculation has been recursively reduced to computation of butterflies.

We graphically demonstrate the entire decomposition for the special case of $N = 8$ in the series of Figures 14.1–14.4. The first figure depicts the needed transform as a black box, with x_0 through x_7 as inputs, and X_0 through X_7 as outputs. The purpose of the graphical derivation is to fill in this box.

In Figure 14.2 we slightly rearrange the order of the inputs, and decompose the eight-point transform into two four-point transforms, using equation (14.3). We then continue by decomposing each four-point transform into two two-point transforms in Figure 14.3, and finally substitute our diagram for the two-point butterfly in order to get Figure 14.4. The final figure simply rearranges the inputs to be in the same order as in the first figure. The required permutation is carried out in a black box labeled *Bit Reversal*; the explanation for this name will be given later.

For a given N, how many butterflies must we perform to compute an N-point DFT? Assuming that $N = 2^m$ is a power of 2, we have $m = \log_2 N$ layers of butterflies, with $\frac{N}{2}$ butterflies to be computed in each layer. Since each butterfly involves one multiplication and two additions (we are slightly overestimating, since some of the multiplications are trivial), we require about $\frac{N}{2} \log_2 N$ complex multiplications and $N \log_2 N$ complex additions. We have thus arrived at the desired conclusion, that the complexity of the DIT FFT is $O(N \log N)$ (the basis of the logarithm is irrelevant since all logarithms are related to each other by a multiplicative constant).

Similarly for radix-R DIT FFT, we would find $\log_R N$ layers of $\frac{N}{R}$ butterflies, each requiring $R - 1$ multiplications. The improvement for radixes greater than two is often not worth the additional coding effort, but will be discussed in Section 14.4.

The DIT FFT we have derived and depicted here is an in-place algorithm. At each of its m layers we replace the existing array with a new one of identical size, with no additional array allocation needed. There is one last technical problem related to this in-place computation we need to solve. After all the butterflies are computed in-place we obtain all the desired X_k, but they are not in the right order. In order to obtain the spectrum in the correct order we need one final in-place stage to unshuffle them (see Figure 14.5).

To understand how the X_k are ordered at the end of the DIT FFT, note that the butterflies themselves do not destroy ordering. It is only the successive in-place decimations that change the order. We know that dec-

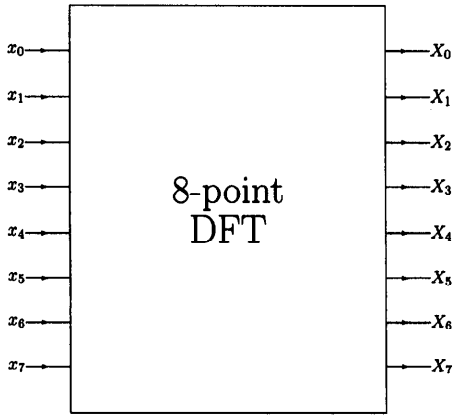


Figure 14.1: An eight-point DFT.

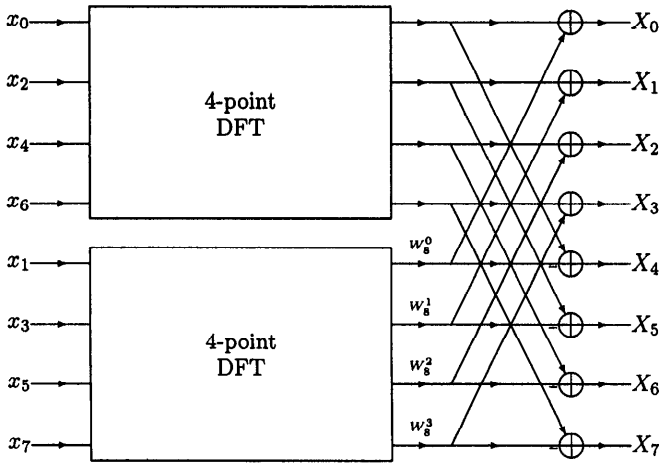


Figure 14.2: An eight-point DFT, divided into two four-point FFTs.

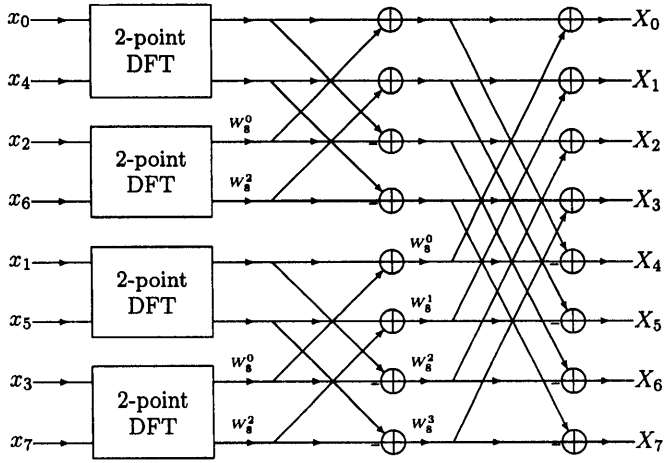


Figure 14.3: An eight-point DFT, divided into four two-point FFTs.

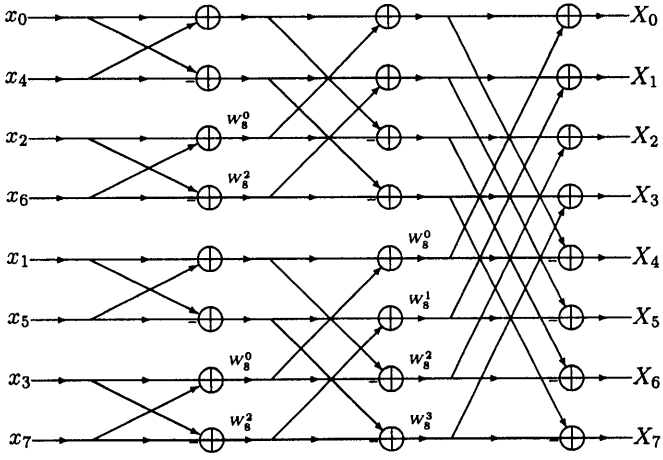


Figure 14.4: The full eight-point radix-2 DIT DFT.

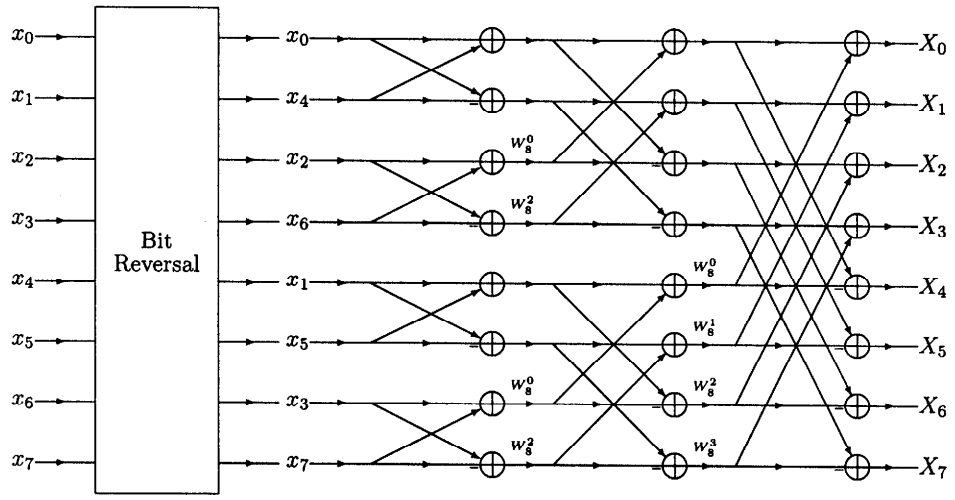


Figure 14.5: The full eight-point radix-2 DIT DFT, with bit reversal on inputs.

imation uses the LSB of the indices to decide how to divide the sequence into subsequences, so it is only natural to investigate the effect of in-place decimation on the binary representation of the indices. For example, for 2^4 element sequences, there are four stages, the indices of which are permuted as follows.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0000	0001	0010	0011	0100	0101	0110	0111	1000	1001	1010	1011	1100	1101	1110	1111
0	2	4	6	8	10	12	14	1	3	5	7	9	11	13	15
0000	0010	0100	0110	1000	1010	1100	1110	0001	0011	0101	0111	1001	1011	1101	1111
0	4	8	12	2	6	10	14	1	5	9	13	3	7	11	15
0000	0100	1000	1100	0010	0110	1010	1110	0001	0101	1001	1101	0011	0111	1011	1111
0	8	4	12	2	10	6	14	1	9	5	13	3	11	7	15
0000	1000	0100	1100	0010	1010	0110	1110	0001	1001	0101	1101	0011	1011	0111	1111

Looking carefully we observe that the elements of the second row can be obtained from the matching ones of the first by a circular left shift. Why is that? The first half of the second row, 0246..., is obtained by an arithmetic left shift of the first row elements, while the second half is identical except for having the LSB set. Since the second half of the first row has the MSB set the net effect is the observed circular left shift.

The transition from second to third row is a bit more complex. Elements from the first half and second half of the second row are not intermingled, rather are separately decimated. This corresponds to clamping the LSB and circularly left shifting the more significant bits, as can be readily verified in the example above. Similarly to go from the third row to the fourth we

clamp the two least significant bits and circularly shift the rest. The net effect of the m stages of in-place decimation of a sequence of length 2^m is *bit reversal* of the indices.

In order to unshuffle the output of the DIT FFT we just need to perform an initial stage of bit reversal on the x_n, as depicted in Figure 14.5. Although this stage contains no computations, it may paradoxically consume a lot of computation time because of its strange indexing. For this reason many DSP processors contain special addressing modes that facilitate efficient bit-reversed access to vectors.

EXERCISES

14.3.1 Draw the flow diagram for the 16-point DIT FFT including bit reversal. (Hint: Prepare a *large* piece of paper.)

14.3.2 Write an explicitly recursive program for computation of the FFT. The main routine `FFT(N,X)` should first check if `N` equals 2, in which case it replaces the two elements of `X` with their DFT. If not, it should call `FFT(N/2,Y)` as needed.

14.3.3 Write a nonrecursive DIT FFT routine. The main loop should run $m = \log_2 N$ times, each time computing $\frac{N}{2}$ butterflies. Test the routine on sums of sinusoids. Compare the run time of this routine with that of N straightforward DFT computations.

14.3.4 Rather than performing bit reversal as the first stage, we may leave the inputs and shuffle the outputs into the proper order. Show how to do this for an eight-point signal. Are there any advantages to this method?

14.3.5 Write an efficient high-level-language routine that performs bit reversal on a sequence of length $N = 2^m$. The routine should perform no more than N element interchanges, and use only integer addition, subtraction, comparison, and single bit shifts.

14.4 Other Common FFT Algorithms

In the previous section we saw the radix-2 DIT algorithm, also known as the Cooley-Tukey algorithm. Here we present a few more FFT algorithms, radix-2 DIF, the prime factor algorithm (PFA), non-power-of-two radixes, split-radix, etc. Although different in details, there is a strong family resemblance between all these algorithms. All reduce the N^2 complexity of

the straightforward DFT to $N \log N$ by restructuring the computation, all exploit symmetries of the W_N^{nk}, and all rely on the length of the signal N being highly composite.

First let us consider the decimation in frequency (DIF) FFT algorithm. The algebraic derivation follows the same philosophy as that of the DIT. We start by partitioning the time sequence, into left and right subsequences

$$\begin{aligned} x_n^L &= x_n \qquad \text{for } n = 0, 1, \ldots \frac{N}{2} - 1 \\ x_n^R &= x_{n+\frac{N}{2}} \end{aligned}$$

and splitting the DFT sum into two sums.

$$\begin{aligned} X_k &= \sum_{n=0}^{N-1} x_n W_N^{nk} = \sum_{n=0}^{\frac{N}{2}-1} x_n W_N^{nk} + \sum_{n=\frac{N}{2}}^{N-1} x_n W_N^{nk} W_N^{k\frac{N}{2}} \qquad (14.5) \\ &= \sum_{n=0}^{\frac{N}{2}-1} x_n^L W_N^{nk} + \sum_{n=0}^{\frac{N}{2}-1} x_n^R W_N^{nk} \end{aligned}$$

Now let's compare the even and odd X_k (decimation in the frequency domain). Using the fact that $W_N^2 = W_{\frac{N}{2}}$

$$\begin{aligned} X_{2k} &= \sum_{n=0}^{\frac{N}{2}-1} (x_n^L W_{\frac{N}{2}}^{nk} + x_n^R W_{\frac{N}{2}}^{nk} W_N^{Nk}) \\ X_{2k+1} &= \sum_{n=0}^{\frac{N}{2}-1} (x_n^L W_{\frac{N}{2}}^{nk} + x_n^R W_{\frac{N}{2}}^{nk} W_N^{Nk}) W_N^n \end{aligned}$$

and then substituting $W_N^{kN} = 1$ and $W_N^{\frac{N}{2}} = -1$ we find

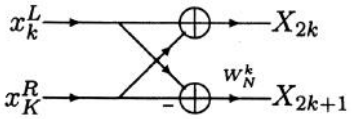
$$\begin{aligned} X_{2k} &= \sum_{n=0}^{\frac{N}{2}-1} (x_n^L + x_n^R) W_{\frac{N}{2}}^{nk} \\ X_{2k+1} &= \sum_{n=0}^{\frac{N}{2}-1} (x_n^L - x_n^R) W_{\frac{N}{2}}^{Nk} W_N^n \end{aligned}$$

which by linearity of the DFT gives the desired expression.

$$\begin{aligned} X_{2k} &= (X_k^L + X_k^R) \qquad (14.6) \\ X_{2k+1} &= (X_k^L - X_k^R) W_N^n \end{aligned}$$

Just as for the DIT we found similarity between Fourier components in different frequency partitions, for DIF we find similarity between frequency components that are related by decimation.

It is thus evident that the DIF butterfly can be drawn



which is different from the DIT butterfly, mainly in the position of the twiddle factor.

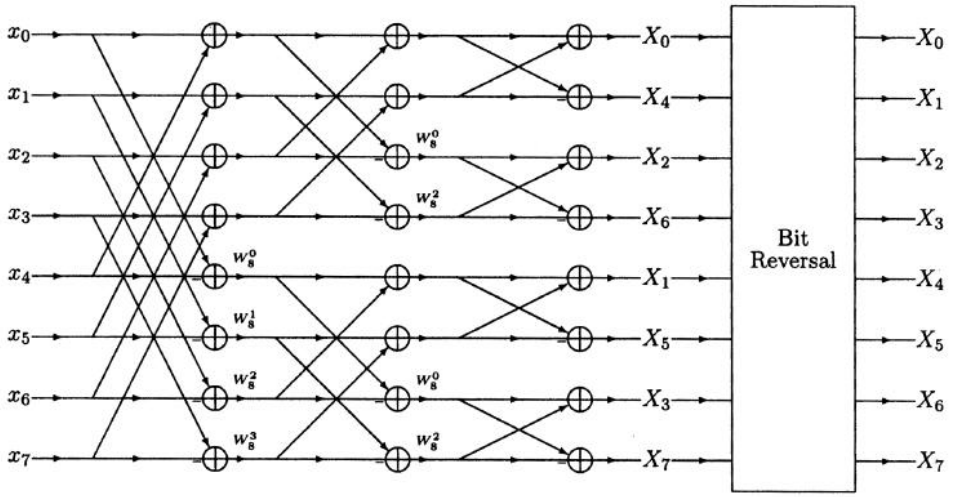


Figure 14.6: Full eight-point radix-2 DIF DFT, with bit reversal on outputs.

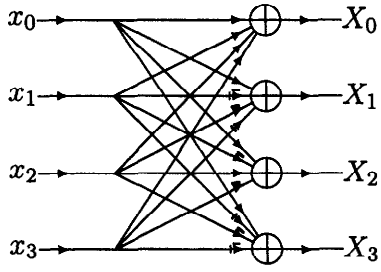
We leave as an exercise to complete the decomposition, mentioning that once again bit reversal is required, only this time it is the outputs that need to be bit reversed. The final eight-point DIF DFT is depicted in Figure 14.6.

Next let's consider FFT algorithms for radixes other than radix-2 in more detail, the most important of which is radix-4. The radix-4 DIT FFT can only be used when N is a power of 4, in which case, of course, a radix-2 algorithm is also applicable; but using a higher radix can reduce the computational complexity at the expense of more complex programming. The derivation of the radix-4 DIT FFT is similar to that of the radix-2 algorithm. The original sequence is decimated into four subsequences $x_{4j}, x_{4j+1}, x_{4j+2}, x_{4j+3}$ (for $j = 0 \ldots \frac{N}{4} - 1$), each of which is further decimated into four subsubse-

quences, etc. The basic 'butterfly' is based on the four-point DFT

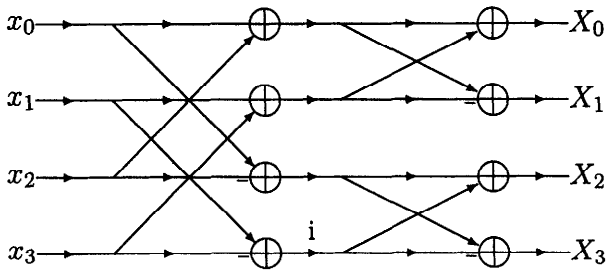
$$\begin{array}{rcccccccc} X_0 & = & x_0 & + & x_1 & + & x_2 & + & x_3 \\ X_1 & = & x_0 & - & \mathrm{i}x_1 & - & x_2 & + & \mathrm{i}x_3 \\ X_2 & = & x_0 & - & x_1 & + & x_2 & - & x_3 \\ X_3 & = & x_0 & + & \mathrm{i}x_1 & - & x_2 & - & \mathrm{i}x_3 \end{array} \tag{14.7}$$

which graphically is



where we have employed two ad-hoc short-hand notations, namely that a line above an arrow means multiplication by -1, while a line before an arrow means multiplication by i. We see that the four-point DFT requires 12 complex additions but no true multiplications. In fact only the radix-2 and radix-4 butterflies are completely multiplication free, and hence the popularity of these radixes.

Now if we compare this butterfly with computation of the four-point DFT via a radix-2 DIT



we are surprised to see that only eight complex additions and no multiplications are needed. Thus it is more efficient to compute a four-point DFT using radix-2 butterflies than radix-4! However, this does not mean that for large N the radix-2 FFT is really better. Recall that when connecting stages of DIT butterflies we need to multiply half the lines with twiddle factors, leading to $O(\frac{N}{2}\log_2 N)$ multiplications. Using radix-4 before every stage we

multiply three-quarters of the lines by nontrivial twiddle factors, but there are only $\log_4 N = \frac{1}{2}\log_2 N$ stages, leading to $\frac{3}{8}\log_2 N$ multiplications. So a full radix-4 decomposition algorithm needs somewhat fewer multiplications than the radix-2 algorithm, and in any case we could compute the basic four-point DFT using radix-2 butterflies, reducing the number of additions to that of the radix-2 algorithm.

Of course a radix-4 decomposition is only half as applicable as a radix-2 one, since only half of the powers of two are powers of four as well. However, every power of two that is not a power of four is twice a power of four, so two radix-4 algorithms can be used and then combined with a final radix-2 stage. This is the called the FFT42 routine. Similarly, the popular FFT842 routine performs as many efficiently coded radix-8 (equation (4.52)) stages as it can, finishing off with a radix-4 or radix-2 stage as needed.

Another trick that combines radix-2 and radix-4 butterflies is called the *split-radix* algorithm. The starting point is the radix-2 DIF butterfly. Recall that only the odd-indexed X_{2k+1} required a twiddle factor, while the even-indexed X_{2k} did not. Similarly all even-indexed outputs of a full-length 2^m DIF FFT are derivable from those of the two length 2^{m-1} FFTs without multiplication by twiddle factors (see Figure 14.6), while the odd-indexed outputs require a twiddle factor each. So split-radix algorithms compute the even-indexed outputs using a radix-2 algorithm, but the odd-indexed outputs using a more efficient radix-4 algorithm. This results in an unusual 'L-shaped' butterfly, but fewer multiplications and additions than any of the standard algorithms.

For lengths that are not powers of two the most common tactic is to use the next larger power of two, zero-padding all the additional signal points. This has the definite advantage of minimizing the number of FFT routines we need to have on hand, and is reasonably efficient if the zero padding is not excessive and a good power-of-two routine (e.g., split-radix or FFT842) is used. The main disadvantage is that we don't get the same number of spectral points as there were time samples. For general spectral analysis this may be taken as an advantage since we get *higher* spectral resolution, but some applications may require conservation of the number of data points.

Moreover, the same principles that lead us to the power-of-two FFT algorithms are applicable to any N that is not prime. If $N = R^m$ then radix-R algorithms are appropriate, but complexity reduction is possible even for lengths that are not simple powers. For example, assuming $N = N_1 N_2$ we can decompose the original sequence of length N into N_2 subsequences of length N_1. One can then compute the DFT of length N by first computing N_2 DFTs of length N_1, multiplying by appropriate twiddle factors, and

finally computing N_1 DFTs of length N_2. Luckily these computations can be performed in-place as well. Such algorithms are called *mixed-radix* FFT algorithms. When N_1 and N_2 are prime numbers (or at least have no common factors) it is possible to eliminate the intermediate step of multiplication by twiddle factors, resulting in the *prime factor* FFT algorithm.

An extremely multiplication-efficient prime factor algorithm was developed by Winograd that requires only $O(N)$ multiplications rather than $O(N \log N)$, at the expense of many more additions. However, it cannot be computed in-place and the indexing is complex. On DSPs with pipelined multiplication and special indexing modes (see Chapter 17) Winograd's FFT runs slower than good implementations of power-of-two algorithms.

EXERCISES

14.4.1 Complete the derivation of the radix-2 DIF FFT both algebraically and graphically. Explain the origin of the bit reversal.

14.4.2 Redraw the diagram of the eight-point radix-2 DIT so that its inputs are in standard order and its outputs bit reversed. This is Cooley and Tukey's original FFT! How is this diagram different from the DIF?

14.4.3 Can a radix-2 algorithm with *both* input and output in standard order be performed in-place?

14.4.4 In addition to checking the LSB (decimation) and checking the MSB (partition) we can divide sequences in two by checking other bits of the binary representation of the indices. Why are only DIT and DIF FFT algorithms popular? Design an eight-point FFT based on checking the middle bit.

14.4.5 A radix-2 DIT FFT requires a final stage of bit reversal. What is required for a radix-4 DIT FFT? Demonstrate this operation on the sequence $0 \ldots 63$.

14.4.6 Write the equations for the radix-8 DFT butterfly. Explain how the FFT842 algorithm works.

14.4.7 Filtering can be performed in the frequency domain by an FFT, followed by multiplying the spectrum by the desired frequency response, and finally an IFFT. Do we need the bit-reversal stage in this application?

14.4.8 Show how to compute a 15-point FFT by decimating the sequence into five subsequences of length three. First express the time index $n = 0 \ldots 14$ as $n = 3n_1 + n_2$ with $n_1 = 0, 1, 2, 3, 4$ and $n_2 = 0, 1, 2$ and the frequency index in the opposite manner $k = k_1 + 5k_2$ with $k_1 = 0, 1, 2$ and $k_2 = 0, 1, 2$ (these are called *index maps*). Next rewrite the FFT substituting these expressions for the indices. Finally rearrange in order to obtain the desired form. What is the computational complexity? Compare with the straightforward DFT and with the 16-point radix-2 FFT.

14.4.9 Research the prime factor FFT. How can the 15-point FFT be computed now? How much complexity reduction is obtained?

14.4.10 Show that if $N = \Pi n_i$ then the number of operations required to perform the DFT is about $(\Sigma n_i)(\Pi n_i)$.

14.5 The Matrix Interpretation of the FFT

In Section 4.9 we saw how to represent the DFT as a matrix product. For example, we can express the four-point FFT of equation (14.7) in the matrix form

$$\begin{pmatrix} X_0 \\ X_1 \\ X_2 \\ X_3 \end{pmatrix} = \begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & -\mathrm{i} & -1 & \mathrm{i} \\ 1 & -1 & 1 & -1 \\ 1 & \mathrm{i} & -1 & -\mathrm{i} \end{pmatrix} \begin{pmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{pmatrix}$$

by using the explicit matrix given in (4.51).

Looking closely at this matrix we observe that rows 0 and 2 are similar, as are rows 1 and 3, reminding us of even/odd decimation! Pursuing this similarity it is not hard to find that

$$\underline{\underline{W_4}} = \begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & -\mathrm{i} & -1 & \mathrm{i} \\ 1 & -1 & 1 & -1 \\ 1 & \mathrm{i} & -1 & -\mathrm{i} \end{pmatrix} = \begin{pmatrix} 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & -\mathrm{i} \\ 1 & 0 & -1 & 0 \\ 0 & 1 & 0 & \mathrm{i} \end{pmatrix} \begin{pmatrix} 1 & 0 & 1 & 0 \\ 1 & 0 & -1 & 0 \\ 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & -1 \end{pmatrix}$$

which is a factoring of the DFT matrix into the product of two sparser matrices. So far we have not gained anything since the original matrix multiplication $\underline{X} = \underline{\underline{W_4}}\,\underline{x}$ took $4^2 = 16$ multiplications, while the rightmost matrix times $\underline{x}$ takes eight and then the left matrix times the resulting vector requires a further eight. However, in reality there were in the original only six nontrivial multiplications and only four in the new representation.

Now for the trick. Reversing the middle two columns of the rightmost matrix we find that we can factor the matrix in a more sophisticated way

$$\underline{\underline{W_4}} = \begin{pmatrix} 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & -\mathrm{i} \\ 1 & 0 & -1 & 0 \\ 0 & 1 & 0 & \mathrm{i} \end{pmatrix} \begin{pmatrix} 1 & 1 & 0 & 0 \\ 1 & -1 & 0 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & -1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

that can be written symbolically as

$$\underline{\underline{W_4}} = \begin{pmatrix} \underline{\underline{I_2}} & \underline{\underline{T_2}} \\ \underline{\underline{I_2}} & -\underline{\underline{T_2}} \end{pmatrix} \begin{pmatrix} \underline{\underline{W_2}} & \underline{\underline{0_2}} \\ \underline{\underline{0_2}} & \underline{\underline{W_2}} \end{pmatrix} \underline{\underline{C_4}} \tag{14.8}$$

where $\underline{\underline{I_2}}$ is the two-by-two identity matrix, $\underline{\underline{T_2}}$ is a two-by-two diagonal matrix with twiddle factors as elements ($W_4^0 = 1$ and $W_4^1 = -\mathrm{i}$), $\underline{\underline{W_2}}$ is the two-point DFT matrix (butterfly), $\underline{\underline{0_2}}$ is the two-by-two null matrix, and $\underline{\underline{C_4}}$ is a four-by-four column permutation matrix.

This factorization has essentially reduced the four-dimensional DFT computation (matrix product) into two two-dimensional ones, with some rearranging and a bit reversal. This decomposition is quite general

$$\underline{\underline{W_{2m}}} = \begin{pmatrix} \underline{\underline{I_m}} & \underline{\underline{T_m}} \\ \underline{\underline{I_m}} & -\underline{\underline{T_m}} \end{pmatrix} \begin{pmatrix} \underline{\underline{W_m}} & \underline{\underline{0_m}} \\ \underline{\underline{0_m}} & \underline{\underline{W_m}} \end{pmatrix} \underline{\underline{C_m}} \tag{14.9}$$

where T_m is the diagonal m-by-m matrix with elements W_{2m}^i, the twiddle factors for the $2m$-dimensional DFT.

Looking carefully at the matrices we recognize the first step in the decomposition of the DFT that leads to the radix-2 DIT algorithm. Reading from right to left (the way the multiplications are carried out) the column permutation C_m is the in-place decimation that moves the even-numbered elements up front; the two W_m are the half size DFTs, and the leftmost matrix contains the twiddle factors. Of course, we can repeat the process for the blocks of the middle matrix in order to recurse down to $\underline{\underline{W_2}}$.

EXERCISES

14.5.1 Show that by normalizing $\underline{\underline{W_N}}$ by $\frac{1}{\sqrt{N}}$ we obtain a unitary matrix. What are its eigenvalues?

14.5.2 Define $\underline{\underline{\tilde{W}_N}}$ to be the DFT matrix after bit-reversal permutation of its rows. Write down $\underline{\underline{\tilde{W}_2}}$ and $\underline{\underline{\tilde{W}_4}}$. Show that $\underline{\underline{\tilde{W}_4}}$ can be written as follows.

$$\begin{pmatrix} \underline{\underline{\tilde{W}_2}} & \underline{\underline{0_2}} \\ \underline{\underline{0_2}} & \underline{\underline{\tilde{W}_2}} \end{pmatrix} \begin{pmatrix} \underline{\underline{I_2}} & \underline{\underline{I_2}} \\ \underline{\underline{T_4}} & -\underline{\underline{T_4}} \end{pmatrix}$$

To which FFT algorithm does this factorization correspond?

14.5.3 The Hadamard matrix of order 2 is defined to be

$$\underline{\underline{H_2}} = \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix}$$

and for all other powers of two we define by recursion.

$$\underline{\underline{H_{2^{m+1}}}} = \begin{pmatrix} \underline{\underline{H_{2^m}}} & \underline{\underline{H_{2^m}}} \\ \underline{\underline{H_{2^m}}} & -\underline{\underline{H_{2^m}}} \end{pmatrix}$$

Build the Hadamard matrices for $m = 2$ and 3. Show that they are symmetric and orthogonal. What is the inverse of $\underline{\underline{H_{2^m}}}$?

14.5.4 Use the Hadamard matrix instead of $\underline{\underline{W}}$ to define a transform. How many additions and multiplications are needed to compute the transform? What are the twiddle factors? What about bit reversal? Can we compute the Hadamard transform faster by using higher radix algorithms?

14.6 Practical Matters

A few more details must be worked out before you are ready to properly compute FFTs in practice. For concreteness we discuss the radix-2 FFT (either DIT or DIF) although similar results hold for other algorithms. First, we have been counting the multiplications and additions but neglecting computation of the twiddle factors. Repeatedly calling library routines to compute sine and cosine functions would take significantly more time than all the butterfly multiplications and additions we have been discussing. There are two commonly used tactics: storing the W_n^k in a table, and generating them in real-time using trigonometric identities.

By far the most commonly used method in real-time implementations is the use of twiddle factor tables. For a 16-point FFT we will need to store $W_{16}^0 = 1$, $W_{16}^1 = \cos(\frac{2\pi}{16}) - \mathrm{i}\sin(\frac{2\pi}{16})$, ... $W_{16}^7 = \cos(\frac{2\pi 7}{16}) - \mathrm{i}\sin(\frac{2\pi 7}{16})$, requiring 16 real memory locations. When these tables reside in fast (preferably 'on-chip') memory and the code is properly designed, the table lookup time should be small (but not negligible) compared with the rest of the computation. So the only drawback is the need for setting aside memory for this purpose. When the tables must be 'off-chip' the toll is higher, and the author has even seen poorly written code where the table lookup completely dominated the run-time.

Where do the tables come from? For most applications the size N of the FFT is decided early on in the code design, and the twiddle factor tables can be precomputed and stored as constants. Many DSP processors have special 'table memory' that is ideal for this purpose. For general-purpose (library) routines the twiddle factor tables are usually initialized upon the call to the FFT routine. On general-purpose computers one can usually get away with calling the library trigonometric functions to fill up the tables; but on DSPs one either stores only entire table in program code memory, or stores only W_N itself and derives the rest of the required twiddle factors using

$$W_N^k = W_N^{k-1} W_N$$

or the equivalent trigonometric identities (A.23). When N is large numeric errors may accumulate while recursing, and it is preferable to periodically reseed the recursion (e.g., with $W_N^{\frac{N}{4}} = -\mathrm{i}$).

For those applications where the twiddle factors cannot be stored, they must be generated in real-time as required. Once again the idea is to know only W_N and to generate W_N^{nk} as required. In each stage we can arrange the butterfly computation so that the required twiddle factor exponents form increasing sequences of the form $W_N^0, W_N^n, W_N^{2n}, \ldots$. Then the obvious identity

$$W_N^{nk} = W_N^{(n-1)k} W_N^k$$

or its trigonometric equivalent can be used. This is the reason that general-purpose FFT routines, rather than having *two* loops (an outside loop on stages and a nested loop on butterflies), often have *three* loops. Inside the loop on stages is a loop on *butterfly groups* (these groups are evident in Figures 14.5 and 14.6), each of which has an increasing sequence of twiddle factors, and nested inside this loop is the loop on the butterflies in the group.

Another concern is the numeric accuracy of the FFT computation. Every butterfly potentially contributes round-off error to the calculation, and since each final result depends on $\log_2 N$ butterflies in series, we expect this numeric error to increase linearly with $\log_2 N$. So larger FFTs will be less accurate than smaller ones, but the degradation is slow. However, this prediction is usually only relevant for floating point computation. For fixed point processors there is a much more serious problem to consider, that of overflow. Overflow is always a potential problem with fixed point processing, but the situation is particularly unfavorable for FFT computation. The reason for this is not hard to understand. For simplicity, think of a single sinusoid of frequency $\frac{k}{N}$ so that an integer number of cycles fits into the

FFT input buffer. The FFT output will be nonzero only in the k^{th} bin, and so all the energy of the input signal will be concentrated into a single large number. For example, if the signal was pure DC $x_n = 1$, the only nonzero bin is $X_0 = \sum x_n = N$, and similarly for all other single-bin cases. It is thus clear that if the input signal almost filled the dynamic range of the fixed point word, then the single output bin will most certainly overflow it.

The above argument may lead you to believe that overflow is only of concern in special cases, such as when only a single bin or a small number of bins are nonzero. We will now show that it happens even for the opposite case of white noise, when all the output bins are equal in size. From Parseval's relation for the DFT (4.42) we know that (using our usual normalization), if the sum of the input squared is E^2, then the sum of the output squared will be NE^2. Hence the rms value of the output is greater by a factor $\sqrt{N}$ than the input rms. For white noise this implies that the typical output value is greater than a typical input value by this factor!

Summarizing, narrow-band FFT components scale like N while wide-band, noise-like components scale like $\sqrt{N}$. For large N both types of output bins are considerably larger than typical input bins, and hence there is a serious danger of overflow. The simplest way to combat this threat is to restrict the size of the inputs in order to ensure that no overflows can occur. In order to guarantee that the output be smaller than the largest allowed number, the input must be limited to $\frac{1}{N}$ of this maximum. Were the input originally approximately full scale, we would need to divide it by N; resulting in a scaling of the output spectrum by $\frac{1}{N}$ as well. The problem with this prescaling is that crudely dividing the input signal by a factor of N increases the numeric error-to-signal ratio. The relative error, which for floating point processing was proportional to $\log_2 N$, becomes approximately proportional to N. This is unacceptably high.

In order to confine the numeric error we must find a more sophisticated way to avoid overflows; this necessitates intervening with the individual computations that may overflow, namely the butterflies. Assume that we store complex numbers in two memory locations, one containing the real part and the other the imaginary part, and that each of these memories can only store real numbers between -1 and $+1$. Consider butterflies in the first stage of a radix-2 DIT. These involve only addition and subtraction of pairs of such complex numbers. The worst case is when adding complex numbers both of which are equal to $+1$, -1, $+\mathrm{i}$ or $-\mathrm{i}$, where the absolute value is doubled. Were this worst case to transpire at every stage, the overall gain after $\log_2 N$ stages would be $2^{\log_2 N} = N$, corresponding to the case of a single spectral

line. For white noise x_n we see from the DIT butterfly of equation (14.4)

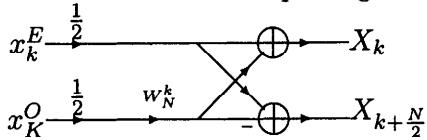
$$\begin{aligned} X_k &= X_k^E + W_N^k X_k^O \\ X_{k+\frac{N}{2}} &= X_k^E - W_N^k X_k^O \end{aligned}$$

that $|X_k|^2 + |X_{k+\frac{N}{2}}|^2 = 2\left(|X_k^E|^2 + |X_k^O|^2\right)$ and if the expected values of all X_k are the same (as for white random inputs) then each must be larger than the corresponding butterfly input by a factor of $\sqrt{2}$. In such a case the overall gain is $\sqrt{2}^{\log_2 N} = \sqrt{N}$ as predicted above.

It is obvious that the worst-case butterfly outputs can only be guaranteed to fit in our memory locations if the input real and imaginary parts are limited in absolute value to $\frac{1}{2}$. However, the butterfly outputs, now themselves between -1 and $+1$, become inputs to the next stage and so may cause overflow there. In order to eliminate this possibility, we must limit the original inputs to $\frac{1}{4}$ in absolute value. Since the same analysis holds for the other butterflies, we reach the previous conclusion that the input absolute values must be prescaled by $\frac{1}{2}^{\log_2 N} = \frac{1}{N}$.

Even with this prescaling we are not completely safe. The worst case we discussed above was only valid for the first stage, where only additions and subtractions take place. For stages with nontrivial twiddle factors the increase can exceed even a factor of two. For example, consider a butterfly containing a rotation by 45°, $X_k^E = 1$, $X_k^O = 1 + \mathrm{i}$. After rotation $1 + \mathrm{i}$ becomes $\sqrt{2}$, which is added to 1 to become $1 + \sqrt{2} \approx 2.414$. Hence, the precise requirement for the complex inputs is for their *length* not to exceed $\frac{1}{N}$. With this restriction, there will be no overflows.

There is an alternative way of avoiding overflow at the second stage of butterflies. Rather than reducing the input to the first stage of butterflies to $\frac{1}{2}$ to $\frac{1}{4}$ we can directly divide the input to the *second* butterfly by 2. For the radix-2 DIT case this translates to replacing our standard DIT by



a failsafe butterfly computation that inherently avoids fixed point overflow. We can now replace *all* butterflies in the FFT with this failsafe one, resulting in an output spectrum divided by N, just as when we divided the input by N. The advantage of this method over input prescaling is that the inputs to each stage are always the maximum size they can be without being susceptible to overflow. With input prescaling only the *last* butterflies have such maximal inputs; all the previous ones receiving smaller inputs.

Due to the butterflies working with maximal inputs the round-off error is significantly reduced as compared with that of the input prescaling method. Some numeric error is introduced by each butterfly, but this noise is itself reduced by a factor of two by the failsafe butterfly; the overall error-to-signal ratio is proportional to $\sqrt{N}$. A further trick can reduce the numeric error still more. Rather than using the failsafe butterfly throughout the FFT, we can (at the expense of further computation) first check if any overflows will occur in the present stage. If yes, we use the failsafe butterfly (and save the appropriate scaling factor), but if not, we can use the regular butterfly. We leave as an exercise to show that this data-dependent prescaling, does not require double computation of the overflowing stages.

EXERCISES

14.6.1 We can reduce the storage requirements of twiddle factor tables by using trigonometric symmetries (A.22). What is the minimum size table needed for $N = 8$? In general? Why are such economies of space rarely used?

14.6.2 What is the numeric error-to-signal ratio for the straightforward computation of the DFT, for floating and fixed point processors?

14.6.3 In the text we discussed the failsafe prescaling butterfly. An alternative is the failsafe postscaling butterfly, which divides by two *after* the butterfly is computed. What special computational feature is required of the processor for postscaling to work? Explain data-dependent postscaling. How does it solve the problem of double computation?

14.6.4 In the text we ignored the problem of the finite resolution of the twiddle factors. What do you expect the effect of this quantization to be?

14.7 Special Cases

The FFT is fast, but for certain special cases it can be made it even faster. The special cases include signals with many zeros in either the time or frequency domain representations, or with many values about which we do not care. We can save computation time by avoiding needless operations such as multiplications by zero, or by not performing operations that lead to unwanted results. You may think that such cases are unusual and not wish to expend the effort to develop special code for them, but certain special

signals often arise in practice. These most common applications cases are zero-padding, interpolation, zoom-FFT and real-valued signals.

We have mentioned the use of zero-padding (adding zeros at the end of the signal) to force a signal to some useful length (e.g., a power of two) or to increase spectral resolution. It is obvious that some savings are obtainable in the first FFT stage, since we can avoid multiplications with zero inputs. Unfortunately, the savings do not carry over to the second stage of either standard DIT or DIF, since the first-stage butterflies mix signal values from widely differing places. Only were the zero elements to be close in bit-reversed order would the task of pruning unnecessary operations be simple.

However, recalling the time partitioning of equation (14.5) we can perform the FFT of the fully populated left half and that of the sparse right half separately, and then combine them with a single stage of DIF butterflies. Of course the right half is probably not *all* zeros, and hence we can't realize all the savings we would wish; however, *its* right half may be all-zero and thus trivial, and the combining of its two halves can also be accomplished in a single stage.

Another application that may benefit from this same ploy is interpolation. Zero-padding in the time domain increased spectral resolution; the dual to this is that zero-padding in the frequency domain can increase time resolution (i.e., perform interpolation). To see how the technique works, assume we want to double the sampling rate, adding a new signal value in between every two values. Assume further that the signal has no DC component. We take the FFT of the signal to be interpolated (with no savings), double the number of points in the spectrum by zero-padding, and finally take the IFFT. This final IFFT can benefit from heeding of zeros; and were we to desire a quadrupling of the sampling rate, the IFFT's argument would have fully three-quarters of its elements zero.

Using a similar ploy, but basing ourselves in the time decimation of equation (14.3), we can save time if a large fraction of either the even- or odd-indexed signal values are zero. This would seem to be an unusual situation, but once again it has its applications.

Probably the most common special case is that of real-valued signals. The straightforward way of finding their FFT is to simply use a complex FFT routine, but then many complex multiplications and additions are performed with one component real. In addition the output has to be Hermitian symmetric (in the usual indexation this means $X_{N-k} = X_k^*$) and so computation of half of the outputs is redundant. We could try pruning the computations, both from the input side (eliminating all operations involv-

ing zeros) and from the output side (eliminating all operations leading to unneeded results), but once again the standard algorithms don't lend themselves to simplification of the intermediate stages. Suppose we were to make the mistake of inputting the vector of $2N$ real signal values R_i into a complex N-point FFT routine that expects interleaved input $(R_0, I_0, R_1, I_1, \ldots$ where $x_i = R_i + \mathrm{i}I_i)$. Is there any way we could recover? The FFT thinks that the signal is $x_0 = R_0 + \mathrm{i}R_1$, $x_1 = R_2 + \mathrm{i}R_3$, etc. and computes a single spectrum X_k. If the FFT routine is a radix-2 DIT the even and odd halves are not mixed until the last stage, but for *any* FFT we can unmix the FFTs of the even and odd subsequences by the inverse of that last stage of DIT butterflies.

$$\begin{aligned} X_k^E &= \tfrac{1}{2}(X_k + X_{N-k}^*) \\ X_k^O &= \tfrac{1}{2}(X_k - X_{N-k}^*) \end{aligned}$$

The desired FFT is now given (see equation (14.4)) by

$$\begin{aligned} R_k &= X_k^E + W_{2N}^k X_k^O & k &= 0 \ldots N-1 \\ R_k &= X_{k-N}^E - W_{2N}^k X_{k-N}^O & k &= N \ldots 2N-1 \end{aligned}$$

and is clearly Hermitian symmetric.

The final special case we mention is the *zoom FFT* used to zoom in on a small area of the spectrum. Obviously for a very high spectral resolution the uncertainty theorem requires an input with very large N, yet we are only interested in a small number of spectral values. Pruning can be very efficient here, but hard to implement since the size and position of the zoom window are usually variable. When only a very small number of spectral lines are required, it may be advantageous to compute the straight DFT, or use Goertzel's algorithm. Another attractive method is mix the signal down so that the area of interest is at DC, low-pass filter and reduce the sampling rate, and only then perform the FFT.

EXERCISES

14.7.1 How can pruning be used to reduce the complexity of zero-padded signals? Start from the diagram of the eight-point DIF FFT and assume that only the first two points are nonzero. Draw the resulting diagram. Repeat with the first four points nonzero, and again with the first six points.

14.7.2 How can the FFT of two real signals of length N be calculated using a single complex FFT of length N?

14.7.3 How can the FFT of four real symmetric ($x_{N-n} = x_n$) signals of length N be calculated using a single complex FFT of length N?

14.7.4 We are interested only in the first two spectral values of an eight-point FFT. Show how pruning can reduce the complexity. Repeat with the first four and six spectral values.

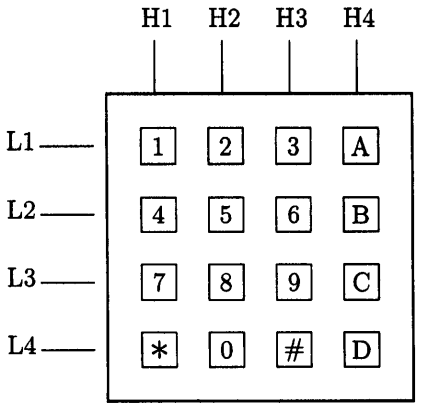
14.8 Goertzel's Algorithm

The fast Fourier transform we have been studying is often the most efficient algorithm to use; however, it is not a panacea. The prudent signal processing professional should be familiar with alternative algorithms that may be more efficient in other circumstances.

In the derivation of the FFT algorithm our emphasis was on finding computations that were superfluous due to their having been previously calculated. This leads to significant economy when the entire spectrum is required, due to symmetries between various frequency components. However, the calculation of each single X_k is not improved, and so the FFT is not the best choice when only a single frequency component, or a small number of components are needed. It *is* true that the complexity of the computation of any component *must* be at least $O(N)$, since every x_n must be taken into account! However, the coefficient of the N in the complexity may be reduced, as compared with the straightforward calculation of equation (14.1). This is the idea behind Goertzel's algorithm.

There are many applications when only a single frequency component, or a small number of components are required. For example, telephony signaling is typically accomplished by the use of tones. The familiar push-button dialing employs a system known as **D**ual **T**one **M**ultiple **F**requency (DTMF) tones, where each row and each column determine a frequency (see figure 14.7). For example, the digit 5 is transmitted by simultaneously emitting the two tones L2 and H2. To decode DTMF digits one must monitor only eight frequencies. Similarly telephone exchanges use a different multifrequency tone system to communicate between themselves, and modems and fax machines also use specific tones during initial stages of their operation.

One obvious method of decoding DTMF tones is to apply eight bandpass filters, calculate the energy of their outputs, pick the maximum from both the low group and the high group, and decode the meaning of this pair of maxima as a digit. That's precisely what we suggest, but we propose



Tone	Frequency (Hz)
L1	697
L2	770
L3	852
L4	941
H1	1209
H2	1336
H3	1477
H4	1633

Figure 14.7: The DTMF telephony signaling method. (Note: The A, B, C, and D tones are not available on the standard telephone, being reserved for special uses.)

using Goertzel's DFT instead of a band-pass filter. Of course we *could* use a regular FFT as a bank of band-pass filters, but an FFT with enough bins for the required frequency resolution would be much higher in computational complexity.

When we are interested only in a single frequency component, or a small number of them that are not simply related, the economies of the FFT are to no avail, and the storage of the entire table of trigonometric constants wasteful. Assuming all the required W_N^{nk} to be precomputed, the basic DFT formula for a single X_k requires N complex multiplications and $N-1$ complex additions to be calculated. In this section we shall assume that the x_n are real, so that this translates to $2N$ real multiplications and $2(N-1)$ real additions.

Recalling the result of exercise 4.7.2, we need only know W_N^k and calculate the other twiddle factors as powers. In particular, when we are interested in only the k^{th} spectral component, we saw in equation (4.55) that the DFT becomes a polynomial in $W \equiv W_N^k \equiv e^{-\mathrm{i}\frac{2\pi k}{N}}$.

$$X_k = \sum_{n=0}^{N-1} x_n W_N^{nk} = x_0 + x_1 W + x_2 W^2 + \ldots x_{N-1} W^{N-1}$$

This polynomial is best calculated using Horner's rule

$$X_k = \Big(\big(\cdots (x_{N-1} W + x_{N-2}) W + \ldots + x_2 \big) W + x_1 \Big) W + x_0$$

which can be written as a recursion.

Given: x_n for $n \leftarrow 0 \ldots N-1$
$P_{N-1} \leftarrow x_{N-1}$
for $n \leftarrow N-1$ down to 0
$P_n \leftarrow P_{n+1}W + x_n$
$X_k \leftarrow P_0$

We usually prefer recursion indices to run in ascending order, and to do this we define $V \equiv W^{-1} = W_N^{-k} \equiv e^{+\mathrm{i}\frac{2\pi k}{N}}$. Since $V^N = 1$ we can write

$$X_k = \sum_{n=0}^{N-1} x_n V^{N-n} = x_0 V^N + x_1 V^{N-1} + \ldots + x_{N-2} V^2 + x_{N-1} V$$

which looks like a convolution. Unfortunately the corresponding recursion isn't of the right form; by multiplying X_k by a phase factor (which won't effect the squared value) we get

$$X'_k = x_0 V^{N-1} + x_1 V^{N-2} + \ldots + x_{N-2} V + x_{N-1}$$

which translates to the following recursion.

Given: x_n for $n \leftarrow 0 \ldots N-1$
$P_0 \leftarrow x_0$
for $n \leftarrow 1$ to $N-1$
$P_n \leftarrow P_{n-1}V + x_n$
$X'_k \leftarrow P_{N-1}$

This recursion has an interesting geometric interpretation. The complex frequency component X_k can be calculated by a sequence of basic $N-1$ moves, each of which consists of rotating the previous result by the angle $\frac{2\pi k}{N}$ and adding the new real x_n.

Each rotation is a complex multiply, contributing either $4(N-1)$ real multiplications and $2(N-1)$ real additions or $3(N-1)$ and $5(N-1)$ respectively. The addition of a real x_n contributes a further $(N-1)$ additions. We see that the use of the recursion rather than expanding the polynomial has not yet saved any computation time. This is due to the use of complex state variables P. Were the state variables to be real rather than complex, we would save about half the multiplies. We will now show that the complex state variables can indeed be replaced by real ones, at the expense of introducing another time lag (i.e., recursion two steps back).

Since we are assuming that x_n are real, we see from our basic recursion step $P_n \leftarrow P_{n-1}V + x_n$ that $P_n - P_{n-1}V$ must be real at every step.

We can implicitly define a sequence Q_n as follows

$$P_n = Q_n - WQ_{n-1} \tag{14.10}$$

and will now show that the Q are indeed real-valued. Substituting into the recursion step

$$\begin{aligned} P_n &\leftarrow x_n + P_{n-1}V \\ Q_n - WQ_{n-1} &\leftarrow x_n + (Q_{n-1} - WQ_{n-2})\,V \\ Q_n &\leftarrow x_n + (V+W)Q_{n-1} + (WV)Q_{n-2} \\ Q_n &\leftarrow x_n + AQ_{n-1} - Q_{n-2} \end{aligned}$$

where $A \equiv V + W = 2\cos(\frac{2\pi k}{N})$. Since the inputs x_n are real and A is real, assuming the Qs start off real (and we can start with zero) they remain real.

The new algorithm is:

Given: x_n for $n = 0 \ldots N-1$
$Q_{-2} \leftarrow 0, \quad Q_{-1} \leftarrow 0$
$Q_0 \leftarrow x_0$
for $n \leftarrow 1$ to $N-1$
    $Q_n \leftarrow x_n + AQ_{n-1} - Q_{n-2}$
$X'_k \leftarrow Q_{N-1} - WQ_{N-2}$

and the desired energy

$$|X_k|^2 = Q_{N-1}^2 + Q_{N-2}^2 - AQ_{N-1}Q_{N-2}$$

must be computed at the end. This recursion requires only a single frequency-dependent coefficient A and requires keeping two lags of Q. Computationally there is only a single real multiplication and two real additions per iteration, for a total of $N-1$ multiplications and $2(N-1)$ additions.

There is basically only one design parameter to be determined before using Goertzel's algorithm, namely the number of points N. Goertzel's algorithm can only be set up to detect frequencies of the form $f = \frac{k}{N} f_s$ where f_s is the sampling frequency; thus selecting larger N allows finer resolution in center frequencies. In addition, as we shall see in the exercises, larger N implies narrower bandwidth as well. However, larger N also entails longer computation time and delay.

EXERCISES

14.8.1 Since Goertzel's algorithm is equivalent to the DFT, the power spectrum response (for $\omega = 2\pi k/N$) is $P(k) = \left(\frac{\sin(\pi k)}{N \sin(\pi k/N)}\right)^2$. Show that the half power point is at 0.44, i.e., the half power bandwidth is 0.88 bin, where each bin is simply $\frac{f_s}{N}$. What is the trade-off between time accuracy and frequency accuracy when using Goertzel's algorithm as a tone detector?

14.8.2 DTMF tones are allowed to be inaccurate in frequency by 1.5%. What would be the size of an FFT that has bins of about this size? How much computation is saved by using Goertzel's algorithm?

14.8.3 DTMF tones are used mainly by customers, while telephone companies use different multitone systems for their own communications. In North America, telephone central offices communicate using *MF trunk tones* 700, 900, 1100, 1300, 1500, 1700, 2600 and 3700 Hz, according to the following table.

Tone	1	2	3	4	5	6
Frequencies	700+900	700+1100	900+1100	700+1300	900+1300	1100+1300
Tone	7	8	9	0	KP	ST
Frequencies	700+1500	900+1500	1100+1500	1300+1500	1100+1700	1500+1700

All messages start with KP and end with ST. Assuming a sampling rate of 8 KHz, what is the minimum N that exactly matches these frequencies? What will the accuracy (bandwidth) be for this N? Assuming N is required to be a power of two, what error will be introduced?

14.8.4 Repeat the previous question for DTMF tones, which are purposely chosen to be nonharmonically related. The standard requires detection if the frequencies are accurate to within $\pm 1.5\%$ and nonoperation for deviation of $\pm 3.5\%$ or more. Also the minimal on-time is 40 milliseconds, but tones can be transmitted at a rate of 10 per second. What are the factors to be considered when choosing N?

14.9 FIFO Fourier Transform

The FFT studied above calculates the spectrum *once*; when the spectrum is required to be updated as a function of time the FFT must be reapplied for each time shift. The worst case is when we wish to *slide* through the data one sample at a time, calculating $\text{DFT}(x_0 \ldots x_{N-1})$, $\text{DFT}(x_1 \ldots x_N)$, etc. When this must be done M times, the complexity using the FFT is $O(MN \log N)$. For this case there is a more efficient algorithm, the *FIFO Fourier transform*, which instead of completely recalculating X_k for each shift, updates the previous one.

There is a well-known trick for updating a moving *simple* average

$$A_m = \sum_{n=0}^{N-1} x_{m+n}$$

that takes computation time that is independent of N. The trick employs a FIFO of length N, holding the samples to be summed. First we wait until the FIFO is full, and then sum it up once to obtain A_0. Thereafter, rather than summing N elements, we update the sum using

$$A_{m+1} = A_m + x_{m+N} - x_m$$

recursively. For example, the second sum A_1 is derived from A_0 by removing the unnecessary x_0 and adding the new term x_N.

Unfortunately, this trick doesn't generalize to moving averages with coefficients, such as general FIR filters. However, a slightly modified version of it *can* be used for the recursive updating of the components of a DFT

$$X_{km} = \sum_{n=m}^{m+N-1} x_n W_N^{nk} \tag{14.11}$$

where in this case we do not 'reset the clock' as would be the case were we to call a canned FFT routine for each m. After a single initial DFT or FFT has been computed, to compute the next we need only update via the **FIFO Fourier Transform** (FIFOFT)

$$X_{km+1} = X_{km} + (x_{m+N} - x_m)W_N^{mk} \tag{14.12}$$

requiring only two complex additions and one complex multiplication per desired frequency component.

Let's prove equation (14.12). Rewriting equation (14.11) for m and $m+1$

$$\begin{aligned} X_{km} &= \sum_{n=0}^{N-1} x_{m+n} W_N^{(m+n)k} \\ X_{km+1} &= \sum_{n=0}^{N-1} x_{m+1+n} W_N^{(m+1+n)k} \\ &= \sum_{n=1}^{N} x_{m+n} W_N^{(m+n)k} \\ &= \sum_{n=0}^{N-1} x_{m+n} W_N^{(m+n)k} - x_m W_N^{mk} + x_{m+N} W_N^{(m+N)k} \end{aligned}$$

and since $W_N^{Nk} = 1$ we obtain equation (14.12).

When all N frequency components are required, the FIFOFT requires N complex multiplications and $2N$ complex additions per shift. For $N > 4$ this is less than the $\frac{N}{2}\log_2 N$ multiplications and $N\log_2 N$ additions required by the FFT. Of course after N shifts we have performed $O(N^2)$ multiplications compared with $O(N \log N)$ for a single additional FFT, but we have received a lot more information as well.

Another, perhaps even more significant advantage of the FIFOFT is the fact that it does not introduce notable delay. The FFT can only be used in real-time processing when the delay between the input buffer being filled and FFT result becoming available is small enough. The FIFOFT is truly real-time, similar to direct computation of a convolution. Of course the first computation must somehow be performed (perhaps not in real-time), but in many applications we can just start with zeros in the FIFO and wait for the answers to become correct. The other problem with the FIFOFT is that numeric errors may accumulate, especially if the input is of large dynamic range. In such cases the DFT should be periodically reinitialized by a more accurate computation.

EXERCISES

14.9.1 For what types of MA filter coefficients are there FIFO algorithms with complexity independent of N?

14.9.2 Sometimes we don't actually need the recomputation for *every* input sample, but only for every r samples. For what r does it become more efficient to use the FFT rather than the FIFOFT?

14.9.3 Derive a FIFOFT that uses N complex additions and multiplications per desired frequency component, for the case of resetting the clock.

$$X_{km} = \sum_{n=0}^{N-1} x_m + nW_N^{nk} \tag{14.13}$$

14.9.4 The FIFOFT as derived above does not allow for windowing of the input signal before transforming. For what types of windows can we define a moving average FT with complexity independent of N?

Bibliographical Notes

More detail on the use of FFT like algorithms for multiplication can be found in Chapter 4 of the second volume of Knuth [136].

An early reference to computation of Fourier transforms is the 1958 paper by Blackman and Tukey that was later reprinted as a book [19].

The radix-2 DIT FFT was popularized in 1965 by James Cooley of IBM and John Tukey of Princeton [45]. Cooley recounts in [43] that the complexity reduction idea was due to Tukey, and that the compelling applications were military, including seismic verification of Russian compliance with a nuclear test ban and long-range acoustic detection of submarines. Once Cooley finished his implementation, IBM was interested in publishing the paper in order to ensure that such algorithms did not become patented. The first known full application of the newly published FFT was by an IBM geophysicist named Lee Alsop who was studying seismographic records of an earthquake that had recently taken place in Alaska. Using 2048 data points, the FFT reduced the lengthy computation to seconds.

Gordon Sande, a student of Tukey's at Princeton, heard about the complexity reduction and worked out the DIF algorithm. After Cooley sent his draft paper to Tukey and asked the latter to be a co-author, Sande decided not to publish his work.

Actually, radix-2 FFT-like algorithms have a long history. In about 1805 the great mathematician Gauss [177] used, but did not publish, an algorithm essentially the same as Cooley and Tukey's two years before Fourier's presentation of his theorem at the Paris Institute! Although eventually published posthumously in 1866, the idea did not attract a lot of attention. Further historical information is available in [44].

The classic reference for special real-valued FFT algorithms is [248]. The split-radix algorithm is discussed in [57, 247, 56].

The prime factor FFT was introduced in [137], based on earlier ideas (e.g. [240, 29]) and an in-place algorithm given in [30]. The extension to real-valued signals is given in [98].

Winograd's prime factor FFT [283, 284] is based on a reduction of a DFT of prime length N into a circular convolution of length $N-1$ first published as a letter by Rader [214]. A good account is found in the McClellan and Rader book on number theory in DSP [166].

Goertzel's original article is [77]. The MAFT is treated in [7, 249]. The zoom FFT can be found in [288].

A somewhat dated but still relevant review of the FFT and its applications can be found in [16] and much useful material including FORTRAN language sources came out of the 1968 Arden House workshop on the FFT, reprinted in the June 1969 edition of the IEEE Transactions on Audio and Electroacoustics (AU-17(2) pp. 66-169). Many original papers are reprinted in [209, 166]). Modern books on the FFT include [26, 28, 31, 246] and Chapter 8 of [241]. Actual code can be found in the last reference, as well as in [31], [30, 247, 17], [41, 198] etc.