Chapter 5

The Fast Fourier Transform

The invention of the fast Fourier transform, by Cooley and Tukey in 1965, was a major breakthrough in digital signal processing and, in retrospect, in applied science in general. Until then, practical use of the discrete Fourier transform was limited to problems in which the number of data to be processed was relatively small. The difficulty in applying the Fourier transform to real problems is that, for an input sequence of length N, the number of arithmetic operations in direct computation of the DFT is proportional to N^2. If, for example, $N = 1000$, about a million operations are needed. In the 1960s, such a number was considered prohibitive in most applications.

Cooley and Tukey's discovery was that when N, the DFT length, is a composite number (i.e., not a prime), the DFT operation can be decomposed to a number of DFTs of shorter lengths. They showed that the total number of operations needed for the shorter DFTs is smaller than the number needed for direct computation of the length-N DFT. Each of the shorter DFTs, in turn, can be decomposed and performed by yet shorter DFTs. This process can be repeated until all DFTs are of prime lengths—the prime factors of N. Finally, the DFTs of prime lengths are computed directly. The total number of operations in this scheme depends on the factorization of N into prime factors, but is usually much smaller than N^2. In particular, if N is an integer power of 2, the number of operations is on the order of $N \log_2 N$. For large N this can be smaller than N^2 by many orders of magnitude. Immediately, the discrete Fourier transform became an immensely practical tool. The algorithms discovered by Cooley and Tukey soon became known as the *fast Fourier transform*, or FFT. This name should not mislead you: FFT algorithms are just computational schemes for computing the DFT; they are not new transforms!

Since Cooley and Tukey's pioneering work, there have been enormous developments in FFT algorithms, and fast algorithms in general.[1] Today this is unquestionably one of the most highly developed areas of digital signal processing.

This chapter serves as an introduction to FFT. We first explain the general principle of DFT decomposition, show how it reduces the number of operations, and present a recursive implementation of this decomposition. We then discuss in detail the most common special case of FFT: the radix-2 algorithms. Next we present the radix-4 FFT, which is more efficient than radix-2 FFT. Finally we discuss a few FFT-related topics: FFTs of real sequences, linear convolutions using FFT, and the chirp Fourier transform algorithm for computing the DFT at a selected frequency range.

5.1 Operation Count

Before entering the main topic of this chapter, let us discuss the subject of operation count. It is common, in evaluating the computational complexity of a numerical algorithm, to count the number of real multiplications and the number of real additions. By "real" we mean either fixed-point or floating-point operations, depending on the specific computer and the way arithmetic is implemented on it. Subtraction is considered equivalent to addition. Divisions, if present, are counted separately. Other operations, such as loading from memory, storing in memory, indexing, loop counting, and input-output, are usually not counted, since they depend on the specific architecture and the implementation of the algorithm. Such operations represent overhead: They should not be ignored, and their contribution to the total load should be estimated (at least roughly) and taken into account.

Modern-day DSP microprocessors typically perform a real multiplication *and* a real addition in a single machine cycle, so the traditional adage that "multiplication is much more time-consuming than addition" has largely become obsolete. The operation is $y = ax + b$, and is usually called MAC (for multiply/accumulate). When an algorithm such as FFT is to be implemented on a machine equipped with MAC instruction, it makes more sense to count the *maximum* of the number of additions and the number of multiplications (rather than their sum).

The DFT operation is, in general, a multiplication of a complex $N \times N$ matrix by a complex N-dimensional vector. Therefore, if we do not make any attempt to save operations, it will require N^2 complex multiplications and $N(N-1)$ complex additions. However, the elements of the DFT matrix on the first row and on the first column are 1. It is therefore possible to eliminate $2N - 1$ multiplications, ending up with $(N-1)^2$ complex multiplications and $N(N-1)$ complex additions. Each complex multiplication requires four real multiplications and two real additions; each complex addition requires two real additions. Therefore, straightforward computation of the DFT requires $4(N-1)^2$ real multiplications and $4(N-0.5)(N-1)$ real additions. If the input vector is real, the number of operations can be reduced by half.

The preceding operation count ignores the need to evaluate the complex numbers W_N^{-nk}. Usually, it is assumed that these numbers are computed off line and stored. If this is not true, the computation of these numbers must be taken into account. More on this will be said in Section 5.2.3.

5.2 The Cooley–Tukey Decomposition

5.2.1 Derivation of the CT Decomposition

The Cooley–Tukey (CT) decomposition of the discrete Fourier transform is based on the factorization of N, the DFT length, as a product of numbers smaller than N. Let us thus assume that N is not a prime, so it can be written as $N = PQ$, where both factors are greater than 1. Such a factorization is usually not unique; however, all we need right now is the existence of one such factorization.

We are given the DFT formula

$$X^{\mathrm{d}}[k] = \sum_{n=0}^{N-1} x[n] W_N^{-nk}. \tag{5.1}$$

We begin by dividing the range of integers 0 to $N-1$ in two different ways. For the time index n, division is into Q intervals of length P each. For the frequency index k,

division is into P intervals of length Q each. We then express the variables n and k in the form

$$n = Pq + p, \quad 0 \le q \le Q-1,\ 0 \le p \le P-1, \tag{5.2a}$$

$$k = Qs + r, \quad 0 \le s \le P-1,\ 0 \le r \le Q-1. \tag{5.2b}$$

Figure 5.1 illustrates this division for $P = 3$ and $Q = 2$.

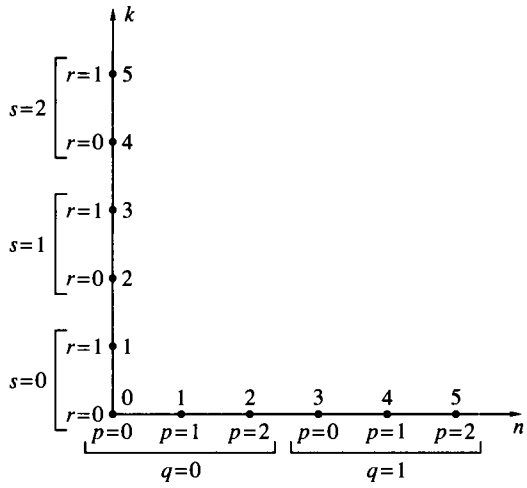

Figure 5.1 Cooley-Tukey decomposition of n and k in the DFT: $P = 3$, $Q = 2$.

The product nk appearing in the exponent of W_N in the DFT formula can be written in terms of p, q, r, s as

$$nk = (Qs + r)(Pq + p) = Nsq + Qsp + Prq + rp. \tag{5.3}$$

Therefore,

$$W_N^{-nk} = W_N^{-Nsq} W_N^{-Qsp} W_N^{-Prq} W_N^{-rp}. \tag{5.4}$$

We have the relationships (Problem 5.1)

$$W_N^N = 1, \quad W_N^Q = W_P, \quad W_N^P = W_Q. \tag{5.5}$$

Therefore,

$$W_N^{-nk} = W_P^{-sp} W_Q^{-rq} W_N^{-rp}. \tag{5.6}$$

Substitution of (5.6) in (5.1) gives

$$\begin{aligned} X^{\mathrm{d}}[Qs + r] &= \sum_{p=0}^{P-1} \sum_{q=0}^{Q-1} x[Pq + p] W_P^{-sp} W_Q^{-rq} W_N^{-rp} \\ &= \sum_{p=0}^{P-1} W_N^{-rp} \left[\sum_{q=0}^{Q-1} x[Pq + p] W_Q^{-rq} \right] W_P^{-sp}. \end{aligned} \tag{5.7}$$

Define the following P sequences, each of length Q:

$$x_p[q] = x[Pq + p], \quad 0 \le q \le Q-1,\ 0 \le p \le P-1. \tag{5.8}$$

Each of the $x_p[q]$ is called a *decimated* sequence, because it is obtained by choosing one out of every P elements of $x[n]$, such that the selected elements are uniformly

spaced. The DFT of $x_p[q]$ is

$$X_p^{\mathrm{d}}[r] = \sum_{q=0}^{Q-1} x_p[q] W_Q^{-rq}. \tag{5.9}$$

Substitution of (5.9) in (5.7) gives

$$X^{\mathrm{d}}[Qs + r] = \sum_{p=0}^{P-1} W_N^{-rp} X_p^{\mathrm{d}}[r] W_P^{-sp}. \tag{5.10}$$

Let us define, for each $0 \le r \le Q-1$, the length-P sequence

$$y_r[p] = W_N^{-rp} X_p^{\mathrm{d}}[r], \quad 0 \le p \le P-1. \tag{5.11}$$

Substitution of (5.11) in (5.10) gives

$$X^{\mathrm{d}}[Qs + r] = \sum_{p=0}^{P-1} y_r[p] W_P^{-sp}. \tag{5.12}$$

Equation (5.12), together with the auxiliary definitions (5.8), (5.9), (5.11), is the Cooley-Tukey decomposition of the DFT. The decomposition can be implemented by the following procedure:

1. Form the P decimated sequences $x_p[q]$ and compute the Q-point DFT of each.
2. Multiply each output of each DFT by the corresponding complex number W_N^{-rp}. These numbers are called *twiddle factors.*
3. For each r, compute the P-point DFT of the sequence $y_r[p]$ to get $X^{\mathrm{d}}[Qs + r]$.

Figure 5.2 illustrates the CT decomposition for $N = 6$, $P = 3$, $Q = 2$. The three decimated sequences are $\{x[0], x[3]\}$, $\{x[1], x[4]\}$, and $\{x[2], x[5]\}$. Each of these sequences is transformed using a DFT_2 operation. The outputs of the three DFTs are multiplied by the six twiddle factors. Then the order of the numbers is changed, to yield two sequences of three numbers each, $\{y_0[0], y_0[1], y_0[2]\}$ and $\{y_1[0], y_1[1], y_1[2]\}$. Finally, each of the two sequences is transformed using a DFT_3 operation to obtain the DFT of the given sequence $x[n]$.

Example 5.1 Let us continue to explore the details of the Cooley-Tukey decomposition for $N = 6$, $P = 3$, $Q = 2$. For convenience, we express the DFT operations as matrix-vector multiplications. Using the definition (5.9) of the sequences $X_p^{\mathrm{d}}[r]$ and the block diagram shown in Figure 5.2, we see that

$$\begin{bmatrix} X_0^{\mathrm{d}}[0] \\ X_0^{\mathrm{d}}[1] \\ X_1^{\mathrm{d}}[0] \\ X_1^{\mathrm{d}}[1] \\ X_2^{\mathrm{d}}[0] \\ X_2^{\mathrm{d}}[1] \end{bmatrix} = \begin{bmatrix} W_2^0 & 0 & 0 & W_2^0 & 0 & 0 \\ W_2^0 & 0 & 0 & W_2^{-1} & 0 & 0 \\ 0 & W_2^0 & 0 & 0 & W_2^0 & 0 \\ 0 & W_2^0 & 0 & 0 & W_2^{-1} & 0 \\ 0 & 0 & W_2^0 & 0 & 0 & W_2^0 \\ 0 & 0 & W_2^0 & 0 & 0 & W_2^{-1} \end{bmatrix} \begin{bmatrix} x[0] \\ x[1] \\ x[2] \\ x[3] \\ x[4] \\ x[5] \end{bmatrix}. \tag{5.13}$$

The operation of multiplication by the twiddle factors can be described by

$$\begin{bmatrix} y_0[0] \\ y_0[1] \\ y_0[2] \\ y_1[0] \\ y_1[1] \\ y_1[2] \end{bmatrix} = \begin{bmatrix} W_6^0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & W_6^0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & W_6^0 & 0 \\ 0 & W_6^0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & W_6^{-1} & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & W_6^{-2} \end{bmatrix} \begin{bmatrix} X_0^{\mathrm{d}}[0] \\ X_0^{\mathrm{d}}[1] \\ X_1^{\mathrm{d}}[0] \\ X_1^{\mathrm{d}}[1] \\ X_2^{\mathrm{d}}[0] \\ X_2^{\mathrm{d}}[1] \end{bmatrix}. \tag{5.14}$$

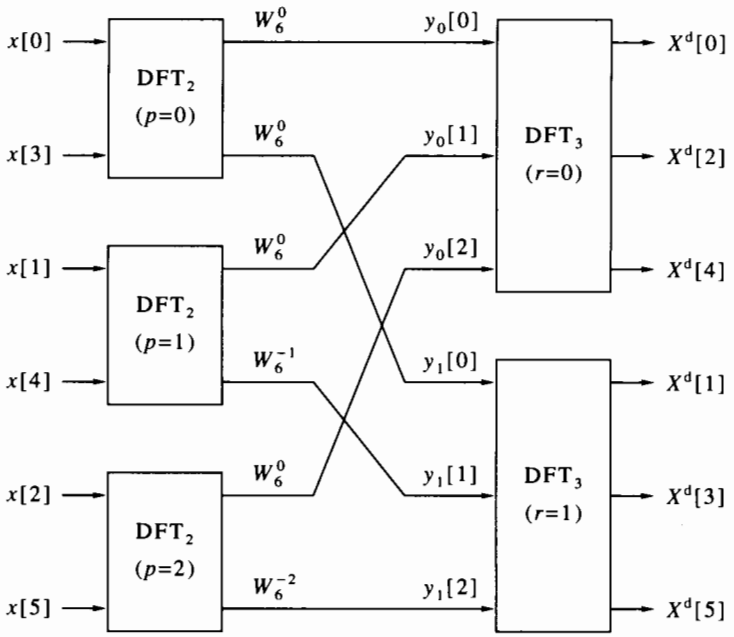

Figure 5.2 Cooley-Tukey DFT algorithm, illustrated for $P = 3$, $Q = 2$.

Finally,

$$\begin{bmatrix} X^{\mathrm{d}}[0] \\ X^{\mathrm{d}}[1] \\ X^{\mathrm{d}}[2] \\ X^{\mathrm{d}}[3] \\ X^{\mathrm{d}}[4] \\ X^{\mathrm{d}}[5] \end{bmatrix} = \begin{bmatrix} W_3^0 & W_3^0 & W_3^0 & 0 & 0 & 0 \\ 0 & 0 & 0 & W_3^0 & W_3^0 & W_3^0 \\ W_3^0 & W_3^{-1} & W_3^{-2} & 0 & 0 & 0 \\ 0 & 0 & 0 & W_3^0 & W_3^{-1} & W_3^{-2} \\ W_3^0 & W_3^{-2} & W_3^{-4} & 0 & 0 & 0 \\ 0 & 0 & 0 & W_3^0 & W_3^{-2} & W_3^{-4} \end{bmatrix} \begin{bmatrix} y_0[0] \\ y_0[1] \\ y_0[2] \\ y_1[0] \\ y_1[1] \\ y_1[2] \end{bmatrix}. \tag{5.15}$$

Denoting the matrices in (5.13), (5.14), (5.15) by $\boldsymbol{G}_{6,1}$, $\boldsymbol{G}_{6,2}$, $\boldsymbol{G}_{6,3}$, respectively, and combining these three equations, we can write

$$\boldsymbol{X}_6^{\mathrm{d}} = \boldsymbol{G}_{6,3}\boldsymbol{G}_{6,2}\boldsymbol{G}_{6,1}\boldsymbol{x}_6. \tag{5.16}$$

Comparing this result with (4.20), we arrive at the following decomposition of the DFT_6 matrix $\boldsymbol{F}_6$:

$$\boldsymbol{F}_6 = \boldsymbol{G}_{6,3}\boldsymbol{G}_{6,2}\boldsymbol{G}_{6,1}. \tag{5.17}$$

We can now count the number of arithmetic operations needed for computing a 6-point DFT by CT decomposition and compare it with the number of operations in a direct computation. In (5.13) there are no multiplications, since $W_2^0 = 1$ and $W_2^{-1} = -1$. There is one addition per row, or a total of 6 additions (subtractions and additions are counted together). In (5.14) there are 2 multiplications, by W_6^{-1} and W_6^{-2}, and no additions. Finally, in (5.15) there are 8 multiplications and 12 additions. In summary, the DFT computation by CT decomposition requires 10 complex multiplications and 18 complex additions. For comparison, direct multiplication by $\boldsymbol{F}_6$ requires 25 multiplications and 30 additions. Therefore, the CT decomposition reduces the number of both multiplications and additions. This reduction is achieved by replacing a large DFT by a number of small DFTs, and is a special case of a general property of the CT decomposition. □

The matrix decomposition derived in Example 5.1 for F_6 can be generalized to any composite N. The CT decomposition implies a decomposition of F_N to a product of three matrices: The rightmost describes the P Q-point DFTs, the center describes the multiplication by the twiddle factors, and the leftmost describes the Q P-point DFTs. Saving in numerical operations results from the CT decomposition because each of the factors in the matrix decomposition is sparse: The number of nonzero entries is a small percentage of the total number of entries in the matrix. Therefore, multiplication by each such matrix requires a number of operations much smaller than the square of its dimension.

We finally observe that the CT decomposition applies to the inverse DFT equally well. The necessary changes are:

1. Replacing the small DFTs by inverse DFTs.
2. Replacing the twiddle factors W_N^{-rp} by W_N^{rp}.

5.2.2 Recursive CT Decomposition and Its Operation Count

The number of operations needed for computing a length-N DFT via the CT decomposition is computed as follows. The procedure requires P DFTs of length Q, then the multiplications by the twiddle factors, and finally Q DFTs of length P. Suppose we compute each of the DFTs directly. A P-point DFT then requires $(P-1)^2$ complex multiplications and $P(P-1)$ complex additions. Similarly, a Q-point DFT requires $(Q-1)^2$ complex multiplications and $Q(Q-1)$ complex additions. The total number of twiddle factors is PQ. However, $P+Q-1$ of these factors are 1, so $(P-1)(Q-1)$ complex multiplications are required for the twiddle factors. Therefore, the total number of complex multiplications is $N(P+Q-3)+1$ and the total number of complex additions is $N(P+Q-2)$. By comparison, the corresponding numbers in a direct computation of an N-point DFT are $(N-1)^2$ and $N(N-1)$. Therefore, if $P+Q \ll N$, we have a significant reduction in the number of operations. This is the principle of operation of Cooley–Tukey FFT algorithms.

Obviously, we do not have to stop at one stage of the decomposition. As long as either P or Q (or both) is composite, we can factor it and use the CT decomposition for each of the shorter DFTs. In general, we can continue this procedure recursively until we are left only with DFTs of prime lengths, the prime factors of N. If we carry out the CT decomposition until we are left with DFTs whose lengths are all prime numbers, the resulting algorithm is called a *mixed radix* FFT. In the special case of N an integer power of a single prime number, it is called a *prime radix* FFT. If, say, $N = P^r$, the algorithm is called a *radix-P* FFT.

The total number of operations in recursive implementation of the CT decomposition can be computed as follows. Let the number of complex additions in an N-point DFT be $\mathcal{A}_c(N)$ and the number of complex multiplications be $\mathcal{M}_c(N)$. Then we get by counting as before,

$$\mathcal{A}_c(N) = P\mathcal{A}_c(Q) + Q\mathcal{A}_c(P), \tag{5.18a}$$

$$\mathcal{M}_c(N) = P\mathcal{M}_c(Q) + Q\mathcal{M}_c(P) + (P-1)(Q-1). \tag{5.18b}$$

We can substitute in these equalities in a recursive manner, until both arguments P and Q are prime. Then we substitute the number of multiplications or additions for the DFTs of corresponding primes.

Example 5.2 Let $N = 30 = 2 \times 3 \times 5$. We have

$$\mathcal{A}_c(2) = 2, \quad \mathcal{A}_c(3) = 6, \quad \mathcal{A}_c(5) = 20,$$

$$\mathcal{M}_c(2) = 0, \quad \mathcal{M}_c(3) = 4, \quad \mathcal{M}_c(5) = 16.$$

Therefore,

$$\mathcal{A}_c(6) = 6 + 12 = 18, \quad \mathcal{M}_c(6) = 8 + 2 = 10,$$

and finally,

$$\mathcal{A}_c(30) = 120 + 90 = 210, \quad \mathcal{M}_c(6) = 96 + 50 + 20 = 166.$$

For comparison, direct computation of a 30-point DFT requires 841 complex multiplications and 870 complex additions. □

5.2.3 Computation of the Twiddle Factors

In counting the operations for direct DFT and for Cooley-Tukey FFT, we ignored the need to compute the twiddle factors W_N^{-pr}. There are only N distinct twiddle factors, since the sequence W_N^n is periodic with period N. There are two common approaches to twiddle factor computation:

1. In special-purpose applications (which are often in real time), the length N is usually fixed, but the FFT must be performed repeatedly on different input sequences. In this case, the most efficient solution is to precompute and store the N twiddle factors in a lookup table. Then, each time one of them is needed, we compute the corresponding position in the lookup table and retrieve its value. Specifically, denote

$$W[n] = W_N^{-n}, \quad 0 \le n \le N - 1. \tag{5.19}$$

Then

$$W_N^{-rp} = W_N^{-(rp \bmod N)} = W[rp \bmod N]. \tag{5.20}$$

Note that the factors appearing in the small FFTs are also available from the table, for example,

$$W_Q^{-m} = W_N^{-Pm} = W[Pm \bmod N].$$

2. In general purpose implementations (such as computer subroutines in mathematical software), the length N usually varies. In this case precomputing and storage is not practical, and the solution is to compute the twiddle factors either prior to the algorithm itself or on the fly. The simplest way is to compute W_N^{-1} directly by cosine and sine, and then use the recursion $W_N^{-(m+1)} = W_N^{-m} W_N^{-1}$. Note, however, that this may be subject to roundoff error accumulation if N is large or the computer word length is small. The opposite (and most conservative) approach is to compute each twiddle factor directly by the appropriate trigonometric functions. Various other schemes have been devised, but are not discussed here.

5.2.4 Computation of the Inverse DFT

So far we discussed only the direct DFT. As we saw in Section 4.1, the inverse DFT differs from the direct DFT in two respects: (1) instead of negative powers of W_N, it uses positive powers; (2) there is an additional division of each output value by N. Every FFT algorithm can be thus modified to perform inverse DFT, by using positive powers of W_N as twiddle factors and multiplying each component of the output (or of the input) by N^{-1}. This entails N extra multiplications, so the computational load increases only slightly. Most FFT computer programs are written so that they can be

switched from direct to inverse FFT by an input flag. MATLAB offers an exception: It uses two different calls, `fft` and `ifft`, for the two operations.

5.2.5 Time Decimation and Frequency Decimation

As a special case of the CT procedure, consider choosing P as the *smallest* prime factor of the length of the DFT at each step of the recursion. Then the next step of the recursion needs to be performed for the P DFTs of length Q, whereas in the Q DFTs of length P no further computational savings are possible. The algorithms thus obtained are called *time-decimated* FFTs. Of special importance is the radix-2, time-decimated FFT, to be studied in Section 5.3. In a dual manner, consider choosing Q as the smallest prime factor of the length of the DFT at each step of the recursion. Then the next step of the recursion needs to be performed for the Q DFTs of length P, whereas in the P DFTs of length Q no further computational savings are possible. The algorithms thus obtained are called *frequency-decimated* FFTs. Of special importance is the radix-2 frequency-decimated FFT, also studied in Section 5.3.

5.2.6 MATLAB Implementation of Cooley–Tukey FFT

Programs 5.1, 5.2, and 5.3 implement a frequency-decimated FFT algorithm in MATLAB. The implementation is recursive and is based on the Cooley–Tukey decomposition. The main program, `edufft`, prepares the sequence W_N^{-n} or W_N^n, depending on whether direct or inverse FFT is to be computed. It then calls `ctrecur` to do the actual FFT computation, and finally normalizes by N^{-1} in the case of inverse FFT. The program `ctrecur` first tries to find the smallest possible prime factor of N. If such a factor is not found, that is, if N is a prime, it calls `primedft` to compute the DFT. Otherwise it sets this prime factor to Q, sets P to N/Q, and then starts the Cooley–Tukey recursion. Figure 5.3 illustrates how the program performs the recursion (for $Q = 3$, $P = 4$). The first step is to arrange the input vector (shown in part a) in a matrix whose rows are the decimated sequences of length Q (shown in part b). Then, since Q is a prime, `primedft` is called to compute the DFT of each row. The next step is to multiply the results by the twiddle factors (shown in part c). The recursion is now invoked, that is, `ctrecur` calls itself for the columns (shown in part d). Finally, the result is rearranged as a vector (shown in part e) and the program exits.

The powers of W_N are computed only once, in `edufft`, and are passed as arguments to the other routines, thus eliminating redundant operations. Even so, the program runs very slow if N is highly composite. This is due to the high overhead of MATLAB in performing recursions and passing parameters. Therefore, this program should be regarded as educational, given here only for illustration, rather than for use in serious applications. The MATLAB routines `fft` and `ifft` should be used in practice, because they are implemented efficiently in an internal code.

5.3 Radix-2 FFT

Suppose the length of the input sequence is an integer power of 2, say $N = 2^r$. We can then choose $P = 2$, $Q = N/2$ and continue recursively until the entire DFT is built out of 2-point DFTs. This is a special case of time-decimated FFT. In a dual manner we can start with $Q = 2$, $P = N/2$, and continue recursively. This is a special case of frequency-decimated FFT. Both are called *radix-2* FFTs.

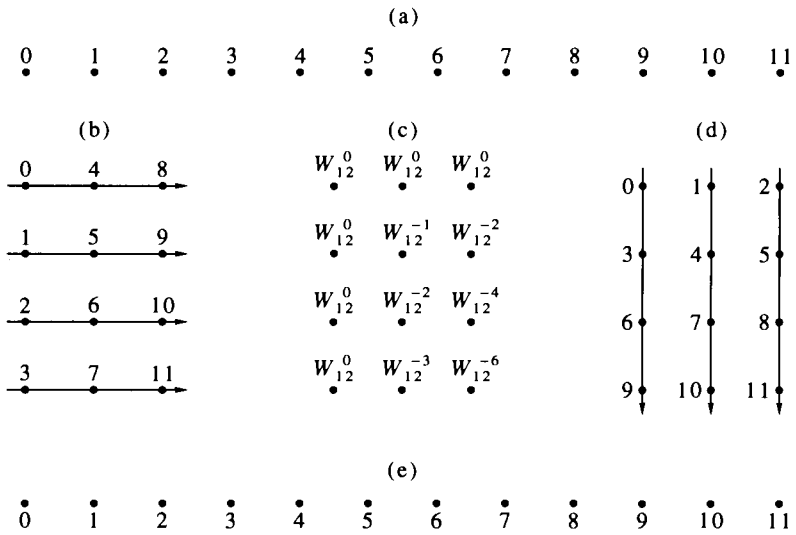

Figure 5.3 Illustration of the steps in the recursive computation of Cooley-Tukey FFT, shown for $Q = 3, P = 4$: (a) the input vector in its natural order; (b) rearrangement of the input vector as a matrix and performing DFT on each row; (c) element-by-element multiplication by the twiddle factors; (d) performing DFT on each column and reindexing of the elements of the matrix; (e) rearrangement of the output as a vector in natural order.

Because of their simplicity of implementation and high computational efficiency, radix-2 FFTs are the most commonly used FFT algorithms. To count their number of operations, observe first that a 2-point DFT is given by

$$\begin{bmatrix} X^{\mathrm{d}}[0] \\ X^{\mathrm{d}}[1] \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \begin{bmatrix} x[0] \\ x[1] \end{bmatrix}. \tag{5.21}$$

Thus, a 2-point DFT requires two additions and *no* multiplications. Therefore, the recursive formulas for the number of operations are

$$\mathcal{A}_{\mathrm{c}}(N) = 0.5N\mathcal{A}_{\mathrm{c}}(2) + 2\mathcal{A}_{\mathrm{c}}(N/2) = N + 2\mathcal{A}_{\mathrm{c}}(N/2), \tag{5.22a}$$

$$\mathcal{M}_{\mathrm{c}}(N) = 0.5N\mathcal{M}_{\mathrm{c}}(2) + 2\mathcal{M}_{\mathrm{c}}(N/2) + 0.5N - 1 = 2\mathcal{M}_{\mathrm{c}}(N/2) + 0.5N - 1. \tag{5.22b}$$

The solutions of these difference equations can be verified to be (Problem 5.21)

$$\mathcal{A}_{\mathrm{c}}(N) = N\log_2 N, \tag{5.23a}$$

$$\mathcal{M}_{\mathrm{c}}(N) = 0.5N(\log_2 N - 2) + 1. \tag{5.23b}$$

The corresponding count of real operations is

$$\mathcal{A}_{\mathrm{r}}(N) = 3N\log_2 N - 2N + 2, \tag{5.24a}$$

$$\mathcal{M}_{\mathrm{r}}(N) = 2N(\log_2 N - 2) + 4. \tag{5.24b}$$

The operation count is the same for both types of radix-2 FFT (time decimated and frequency decimated).

As an example of the computational load, consider $N = 1024$. Direct computation of the DFT would require about 1024^2, or a million complex operations of each kind. By comparison, a radix-2 FFT algorithm requires about 10×1024 complex additions and 4×1024 complex multiplications. Therefore, in this case there is a 100-fold reduction in the number of additions, and 250-fold reduction in the number of multiplications.

5.3.1 The 2-Point DFT Butterfly

The basic building block of radix-2 FFT algorithms is the 2-point DFT. Figure 5.4 illustrates a 2-point DFT. In this figure and in the subsequent ones, we use the following conventions:

1. A line with an arrow indicates signal flow.
2. A circle around a + sign, with two or more lines leading to it, indicates addition.
3. A constant number above a line indicates multiplication of the signal flowing in that line by the constant number.

Because of its visual shape, the 2-point DFT operation is called a *butterfly*.

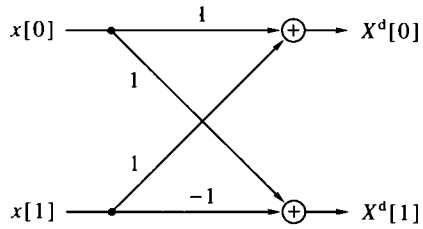

Figure 5.4 The butterfly of a 2-point DFT.

5.3.2 Time-Decimated Radix-2 FFT

The time-decimated radix-2 FFT is obtained by applying the CT decomposition recursively, each time taking $P = 2$ and $Q = N/2$. Figure 5.5 illustrates the flow of computation in the case of $N = 2^3 = 8$. As we see, the main feature of this algorithm is that all internal DFT operations are of length 2; that is, they are replicas of the butterfly shown in Figure 5.4. There are $\log_2 N$ main sections (three in this case), cascaded from left to right. They correspond to the $\log_2 N$ stages in the recursion. In each section there are $0.5N$ butterflies, so the total number of butterflies is $0.5N \log_2 N$. The figure is best read from right to left, although the flow of computation is from left to right. The output variables are ordered in their natural sequential order for convenience, from $X^d[0]$ to $X^d[N-1]$. They are obtained by combining the outputs of two DFTs of length $0.5N$ each (at the next-to-last section), after multiplying them by the N twiddle factors. Note that, in the case $N = 8$, five of the twiddle factors have value 1. In general there will be $0.5N + 1$ such factors. Next, the output values of each of the two DFTs of length $0.5N$ are obtained by combining the outputs of two DFTs of length $0.25N$ each (at the second-to-last section), after multiplying them by the $0.5N$ twiddle factors. Now three of four twiddle factors have value 1. Finally, the first section shows the $0.5N$ DFTs of length 2 that operate on the input sequence. Since this is the first section, all twiddle factors are identically 1.

As we see from Figure 5.5, the input variables are supplied in a permuted order, rather than in their natural order. The permutation is formed by recursively moving the even indices to the beginning and the odd indices to the end. This is precisely the time-decimation operation. A little reflection reveals that the permutation can be mathematically described as follows:

1. Express each index n in the range $[0, N-1]$ as an r-digit binary number, say

$$(n)_2 = n_{r-1} n_{r-2} \cdots n_1 n_0.$$

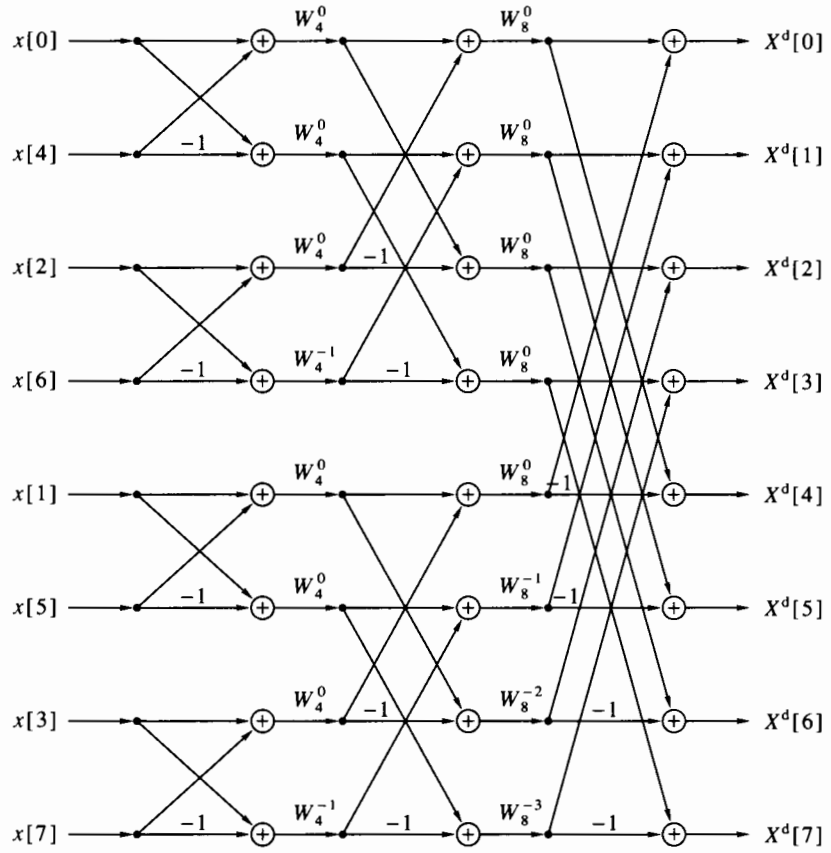

Figure 5.5 Time-decimated radix-2 FFT, shown for $N = 8$. All branches not marked otherwise have gains 1.

2. Reverse the order of bits to form

$$(\tilde{n})_2 = n_0 n_1 \cdots n_{r-2} n_{r-1}.$$

3. Replace $x[n]$ in the input sequence by $x[\tilde{n}]$.

This operation is called *bit reversal.* In the time-decimated radix-2 FFT, bit reversal is performed on the input sequence before starting the butterfly and twiddle factor operations. The output sequence will then be obtained in its natural order.

Example 5.3 For $N = 8$, the binary representations of the numbers 0 through 7 are 000, 001, 010, 011, 100, 101, 110, 111. After bit reversal, they change to 000, 100, 010, 110, 001, 101, 011, 111. Converting back to decimal, we get the sequence 0, 4, 2, 6, 1, 5, 3, 7. This agrees with the ordering of the input sequence in Figure 5.5. □

Another interesting observation can be made from Figure 5.5: Once the output variables of each section have been computed, there is no more need for the input variables. Therefore, we can perform the entire algorithm with N storage words, plus a few temporary variables, plus storage for the twiddle factors. This use of storage is called *in-place computation*, and it is effective if N is large. When in-place computation is used, the output sequence overwrites the input sequence. Therefore, the input sequence must be saved if needed for further use.

5.3.3 Frequency-Decimated Radix-2 FFT

The frequency-decimated radix-2 FFT is obtained by applying the CT decomposition recursively, each time taking $P = N/2$ and $Q = 2$. The frequency-decimated algorithm is the dual of the time-decimated algorithm, and the two have similar properties. Figure 5.6 illustrates the flow of computation in the case of $N = 2^3 = 8$. As we see, the input sequence is now given in its natural order, whereas the output appears in a bit-reversed order. The appearance of the sections is reversed, and the twiddle factors change positions. The total numbers of butterflies, twiddle factors, and operations are identical to those of the time-decimated algorithm. Practically, there is no reason to prefer one over the other, and the choice between the two is usually made arbitrarily.

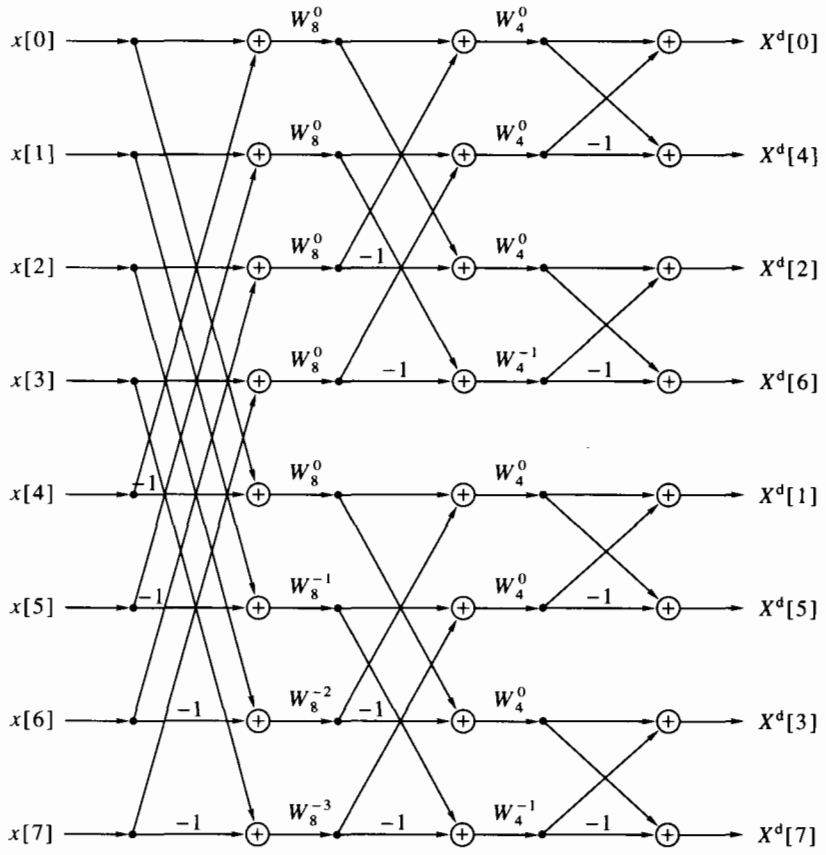

Figure 5.6 Frequency-decimated radix-2 FFT, shown for $N = 8$. All branches not marked otherwise have gains 1.

5.3.4 Signal Scaling in Radix-2 FFT*

When implementing FFT algorithms in fixed point, it is necessary to scale the input signal, the output signal, and the intermediate signals at the various sections so as to make the best use of the computer's accuracy. If the algorithm is implemented in floating point (such as in MATLAB), scaling is almost never a problem. However, many real-time signal processing systems operate in fixed point, because fixed-point

hardware is usually faster and cheaper than floating-point hardware. It is therefore expedient to scale the signals in such a way that they will be represented by as many significant bits as possible, while simultaneously avoiding overflows.

Overflows are potentially harmful to the operation of a numerical algorithm if not detected and handled properly. For example, consider an implementation that can handle only numbers in the range $(-1, 1)$. If the numbers 0.85 and 0.2 are to be added, the theoretical result is 1.05. This result, however, causes overflow. In most fixed-point implementations, such an overflow will cause the result to appear as -0.95 (or nearly so). The difference between the true result and the apparent one is equal to the full dynamic range of the signal! Scaling, possibly accompanied by overflow detection and handling, is aimed at solving this kind of problems.

Consider the scaling problem in radix-2 FFT. Assume that the computer represents numbers as fractions in the range $[-1, 1]$, and that the input signal is scaled such that $|x[n]| \leq 1$ for all n. If the input signal is real, it means that $-1 \leq x[n] \leq 1$. If the input signal is complex, it means that $(\Re\{x[n]\})^2 + (\Im\{x[n]\})^2 \leq 1$. Now, since $|W_N| = 1$, we have

$$|X^{\mathrm{d}}[k]| = \left| \sum_{n=0}^{N-1} x[n] W_N^{-kn} \right| \leq \sum_{n=0}^{N-1} |x[n]| \cdot |W_N^{-kn}| = \sum_{n=0}^{N-1} |x[n]| \leq N. \tag{5.25}$$

This shows that fixed-point DFT is not safe from overflow: The output signal can be as large as N times the largest magnitude of the input signal. Indeed, this happens if $x[n] = \exp(j2\pi n m_0/N)$ for some integer m_0 (Problem 5.16).

Looking more closely into the inner structure of the radix-2 FFT, we observe the following:

1. Multiplication of a signal by a twiddle factor preserves the absolute value of the signal, since the absolute value of any twiddle factor is 1. Therefore, this operation does not cause overflow.
2. A butterfly operation can lead to overflow, since it involves one addition and one subtraction. However, if the absolute values of the inputs to the butterfly are less than 1, the absolute values of the outputs are less than 2. Therefore, we can eliminate overflow at a butterfly operation if we force both inputs to have absolute values less than 0.5.

There are two common approaches to elimination of overflow in butterfly operations:

1. The entire array is multiplied by 0.5 before each stage of the FFT is executed. Recall that the algorithm uses in-place computation. Therefore, the input vector is multiplied by 0.5 before performing the first stage, then the resulting array is multiplied by 0.5 before performing the second stage, and so on. There are $\log_2 N$ stages, so the output array is scaled by $0.5^{\log_2 N}$, that is, by N^{-1}, with respect to the input. Clearly, $|X^{\mathrm{d}}[k]| \leq 1$ at the output. Multiplication by 0.5 can be conveniently performed by shifting the fixed-point number one bit to the right (with sign extension). We remark that this scaling is superior to one-time normalization of the input by N^{-1}, because it makes better use of the computer word length at the intermediate stages (i.e., bits are lost successively, one per stage). Even so, the loss of significant bits in this scheme can be excessive. For example, if $N = 1024$, we lose 10 bits of accuracy along the way.
2. The array is not multiplied by 0.5 before the stage. However, overflow detection is inserted at each butterfly (most computers can test for overflow in the hardware). If overflow is detected, the entire array at the present stage is multiplied by 0.5

and the offending overflow is fixed as well. A counter keeps track of the number of times this happens. If overflow is detected at m stages (necessarily $m \leq \log_2 N$), it means that the output is scaled by 0.5^m with respect to the input. This scheme is called *block floating point*. It provides variable scaling, depending on the input signal, and is nearly optimal for any given signal. For example, the FFT of the signal $\{x[n] = 1,\ 0 \leq n \leq N-1\}$ will be scaled by N^{-1}, as in the first scheme; however, the FFT of $x[n] = \delta[n]$ will not be scaled at all.

Example 5.4 We illustrate block floating-point scaling for time-decimated radix-2 FFT with $N = 8$. The input signal in this example is

$$x[n] = 0.65^{n+1}, \quad 0 \leq n \leq 7.$$

We assume that the computer is accurate to ± 0.0001 and uses truncation arithmetic (for comparison, a 16-bit fixed-point computer has accuracy about three times better). Table 5.1 shows the signal values during the computation. The first column gives the input signal, in bit-reversed order. The second, third, and fourth columns give the outputs of the three stages (see Figure 5.5). The fourth column is the output signal $X^{\mathrm{d}}[k]$. At the first output of the second stage there is overflow, so the entire vector is multiplied by 0.5. Therefore, the output signal is scaled by 0.5 in this case. □

Input	1st stage output	2nd stage output†	3rd stage output
0.6500	0.7660	0.5448	0.8989
0.1160	0.5340	$0.2670 - j0.1128$	$0.3378 - j0.2873$
0.2746	0.3236	0.2212	$0.2212 - j0.1438$
0.0490	0.2256	$0.2670 + j0.1128$	$0.1962 - j0.0617$
0.4225	0.4979	0.3541	0.1907
0.0754	0.3471	$0.1735 - j0.0733$	$0.1962 + j0.0617$
0.1785	0.2103	0.1438	$0.2212 + j0.1438$
0.0318	0.1467	$0.1735 + j0.0733$	$0.3378 + j0.2873$

Table 5.1 The signals in Example 5.4; †in this column there is scaling by 0.5.

5.4 Radix-4 Algorithms*

As we saw in the preceding section, the computational efficiency of radix-2 algorithms is largely a result of the absence of multiplications in the 2×2 butterflies. In this section we investigate another radix whose corresponding butterflies can be implemented without multiplications: radix-4. The resulting algorithms will turn out to be more efficient than radix-2 algorithms.

Consider the 4-point DFT operation. By our discussion of radix-2 FFT, it can be computed with eight complex additions and one complex multiplication. However, the twiddle factor appearing in the multiplication is W_4^{-1}, which is equal to $-j$. Now, multiplication of the complex number $a + jb$ by $-j$ gives $b - ja$, which amounts to replacing the real part by the imaginary part, and replacing the imaginary part by the negative of the real part. Therefore, no numerical multiplication is needed. To reiterate: A 4-point DFT can be performed with eight complex additions and no multiplications. At the point where multiplication by $-j$ is called for, we replace it by the

exchange operation $a + jb \longrightarrow b - ja$. When writing computer code in a low-level language (e.g., in C or Assembler), the exchange operation can be hard-coded, so it does not require extra machine time. Figure 5.7 shows the basic 4-point DFT building block.

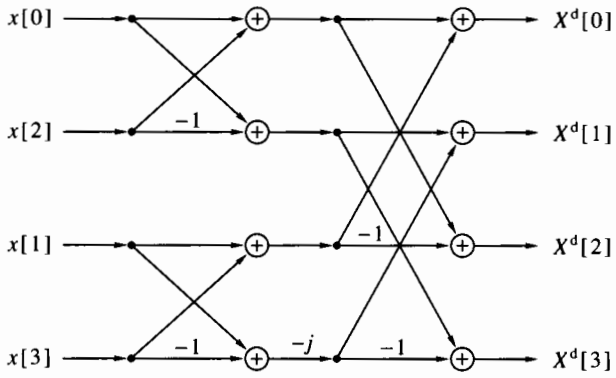

Figure 5.7 Multiplication-free 4-point DFT building block.

We can use the multiplication-free property of the 4-point DFT to construct radix-4 FFTs that are more efficient than radix-2 FFTs if N is an integer power of 4. The recursive formulas for the number of operations are

$$\mathcal{A}_c(N) = 0.25N\mathcal{A}_c(4) + 4\mathcal{A}_c(N/4) = 2N + 4\mathcal{A}_c(N/4), \tag{5.26a}$$

$$\begin{aligned}\mathcal{M}_c(N) &= 0.25N\mathcal{M}_c(4) + 4\mathcal{M}_c(N/4) + 3(0.25N - 1)\\ &= 4\mathcal{M}_c(N/4) + 0.75N - 3.\end{aligned} \tag{5.26b}$$

The solutions of these difference equations can be verified to be (Problem 5.21)

$$\mathcal{A}_c(N) = 2N\log_4 N = N\log_2 N, \tag{5.27a}$$

$$\mathcal{M}_c(N) = 0.75N\log_4 N - N + 1 = 0.375N\log_2 N - N + 1. \tag{5.27b}$$

The corresponding counts of real operations are

$$\mathcal{A}_r(N) = 2.75N\log_2 N - 2N + 2, \tag{5.28a}$$

$$\mathcal{M}_r(N) = 1.5N\log_2 N - 4N + 4. \tag{5.28b}$$

The operation count is the same for both types of radix-4 FFT (time-decimated and frequency-decimated). Instead of bit reversal, we must perform reversal of the base-4 digits of the indices.

As we see, radix-4 FFT achieves about 25 percent reduction in the number of multiplications, as well as a small reduction in the number of additions. This reduction is achieved provided N is an integer power of 4 (i.e., an even power of 2), and assuming that the computer program handles the multiplications by $-j$ as we have described. If N is an odd power of 2, we have the option of doing radix-2 at one section and radix-4 at the remaining sections.

5.5 DFTs of Real Sequences

The FFT algorithms that we have described operate on complex sequences. If the input sequence is real, no computational advantage is gained, since after the first section all

variables become complex. On the other hand, we know that direct computation of the DFT permits 50 percent reduction in multiplications if the input sequence is real, since the product of a real number and a complex one requires only two real multiplications. A question arises as to whether there is a way to achieve a similar reduction with FFT. The answer is a conditional yes: We can compute the FFTs of *two* real sequences in a number of operations only slightly higher than that needed for a single complex FFT. Frequently we must compute the FFTs of many real sequences successively. We can then combine them in pairs, perform a single complex FFT for each pair, thus reducing the total count of operations by almost 50 percent.

The basic idea is to build a single complex sequence using the two real sequences as its real and imaginary parts. If the input sequences are $x[n]$ and $y[n]$, we form

$$z[n] = x[n] + jy[n] \tag{5.29}$$

and compute $Z^{\mathrm{d}}[k]$, the FFT of $z[n]$. The next step is to extract $X^{\mathrm{d}}[k]$ and $Y^{\mathrm{d}}[k]$ from $Z^{\mathrm{d}}[k]$. We have

$$x[n] = 0.5(z[n] + \bar{z}[n]), \tag{5.30a}$$

$$y[n] = j0.5(\bar{z}[n] - z[n]). \tag{5.30b}$$

Therefore, by the conjugation property of the DFT (4.38),

$$X^{\mathrm{d}}[k] = 0.5(Z^{\mathrm{d}}[k] + \bar{Z}^{\mathrm{d}}[N-k]), \tag{5.31a}$$

$$Y^{\mathrm{d}}[k] = j0.5(\bar{Z}^{\mathrm{d}}[N-k] - Z^{\mathrm{d}}[k]). \tag{5.31b}$$

As we see, the operation overhead is $2N$ additions and $2N$ multiplications. However, the multiplications are by 0.5, so they can be performed with right shifts.

Problem 5.18 explores the computation of FFTs of real sequences further.

5.6 Linear Convolution by FFT

In Section 4.7 we showed how to perform linear convolution of two finite-duration sequences using circular convolution. The method involved zero padding the two sequences to a length equal to the sum of the individual lengths minus 1. Now, equipped with the fast Fourier transform, we further develop this idea.

Let the two sequences be $\{x[n],\ 0 \le n \le N_1 - 1\}$ and $\{y[n],\ 0 \le n \le N_2 - 1\}$. Define N to be the smallest power of 2 not smaller than $N_1 + N_2 - 1$ and let $x_{\mathrm{a}}[n]$, $y_{\mathrm{a}}[n]$ be the corresponding zero-padded sequences. Then, by Problem 4.31, the convolution $\{x * y\}[n]$ can be obtained by performing $\{x_{\mathrm{a}} \circledast y_{\mathrm{a}}\}[n]$ and retaining the first $N_1 + N_2 - 1$ elements of the result. Furthermore, using the convolution property of the DFT, we can obtain the latter by the operation

$$z_{\mathrm{a}}[n] = \mathrm{IDFT}\{X_{\mathrm{a}}^{\mathrm{d}}[k] Y_{\mathrm{a}}^{\mathrm{d}}[k]\}. \tag{5.32}$$

The total number of operations, assuming use of a radix-2 FFT, is three times the number of operations for a single N-point FFT: Two for the direct DFTs and one for the inverse DFT. This amounts to about $6N \log_2 N$ real multiplications and $9N \log_2 N$ real additions (less if the input sequences are real and we transform them together as described in Section 5.5). We also need N complex multiplications, or $4N$ real multiplications, to compute the product of the FFTs. By comparison, direct computation of the convolution requires about $N_1 N_2$ multiplications and a similar number of additions (Problem 5.14). Therefore, judging by the number of multiplications, it is preferable to perform the convolution by FFT whenever

$$N_1 N_2 > (N_1 + N_2 - 1)[6 \log_2(N_1 + N_2 - 1) + 4]. \tag{5.33}$$

In digital signal processing it commonly happens that one of the two sequences, say $y[n]$, is known *a priori* and has a fixed length N_2, whereas the other sequence, $x[n]$, is known only in real time, and its length is not fixed and is potentially much larger than N_2. In Chapter 9 we shall learn more about such situations. For now we concern ourselves with the problem of computing the linear convolution $x * y$ under these conditions. Using zero padding is possible, but may not be advisable, for two reasons: (1) The sequence $y[n]$ will have to be padded by many zeros, resulting in many unnecessary computations, and (2) the DFT will have to be performed on very long sequences, which may be inconvenient or impossible. A better approach is to split the long sequence $x[n]$ to segments, each of length N_1 which is of the same order as N_2, convolve $y[n]$ with each of the segments separately, and properly add up the partial results to form the desired convolution $x * y$. This method is known as *overlap-add* (OLA) convolution; we now describe its details.

Let us write the sequence $x[n]$ as

$$x[n] = \sum_i x_i[n], \tag{5.34}$$

where

$$x_i[n] = \begin{cases} x[n], & N_1 i \le n \le N_1(i+1) - 1, \\ 0, & \text{otherwise.} \end{cases} \tag{5.35}$$

We have left the range of the index i unspecified on purpose, to emphasize that the length of $x[n]$ need not be known in advance. If this length is not an integer multiple of N_1, we pad $x[n]$ with zeros to make it so. By linearity of the convolution operator we have

$$z[n] = \{x * y\}[n] = \sum_i \{x_i * y\}[n]. \tag{5.36}$$

The convolution $\{x_i * y\}$ has length $N_1 + N_2 - 1$, and it is nonzero for the range of time points

$$N_1 i \le n \le N_1 i + N_1 + N_2 - 2.$$

We can write $\{x_i * y\}$ as a sum of two sequences, say

$$z_i[n] = \{x_i * y\}[n] = u_i[n] + v_i[n], \tag{5.37}$$

where $u_i[n]$ is nonzero in the range

$$N_1 i \le n \le N_1(i+1) - 1,$$

and $v_i[n]$ is nonzero in the range

$$N_1(i+1) \le n \le N_1(i+1) + N_2 - 2.$$

The range of definition of $u_i[n]$ coincides with that of $x_i[n]$. On the other hand, the range of definition of $v_i[n]$ coincides with the initial part of the range of $x_{i+1}[n]$. Therefore, the proper sequence of operations, implied by (5.36) and (5.37), is

1. Zero-pad $x_i[n]$ and $y[n]$ to a length $N_1 + N_2 - 1$.
2. Perform the circular convolution $\{x_i \circledast y\}$, which is equal to the linear convolution $\{x_i * y\}$. The circular convolution is performed by FFT, as explained earlier.
3. Use $u_i[n]$ as a partial result for the ith stage. Add to its initial part the sequence $v_{i-1}[n]$ from the previous stage. Save $v_i[n]$ for the $(i+1)$st stage.

Figure 5.8 illustrates the various sequences, their proper timings, and the way they are combined. The signal $y[n]$ in this example has length $N_2 = 4$ and its values are

$\{0.1, 0.5, 0.25, 0.15\}$. The length chosen for the sequences $x_i[n]$ is $N_1 = 5$. Accordingly, the lengths of $z_i[n]$, $u_i[n]$ and $v_i[n]$ are 8, 5, and 3, respectively. The reason for the name *overlap-add* should be now clear: The tail $v_i[n]$ of the convolution $\{x_i * y\}[n]$ overlaps with the head of $\{x_{i+1} * y\}[n]$, and must be added to it.

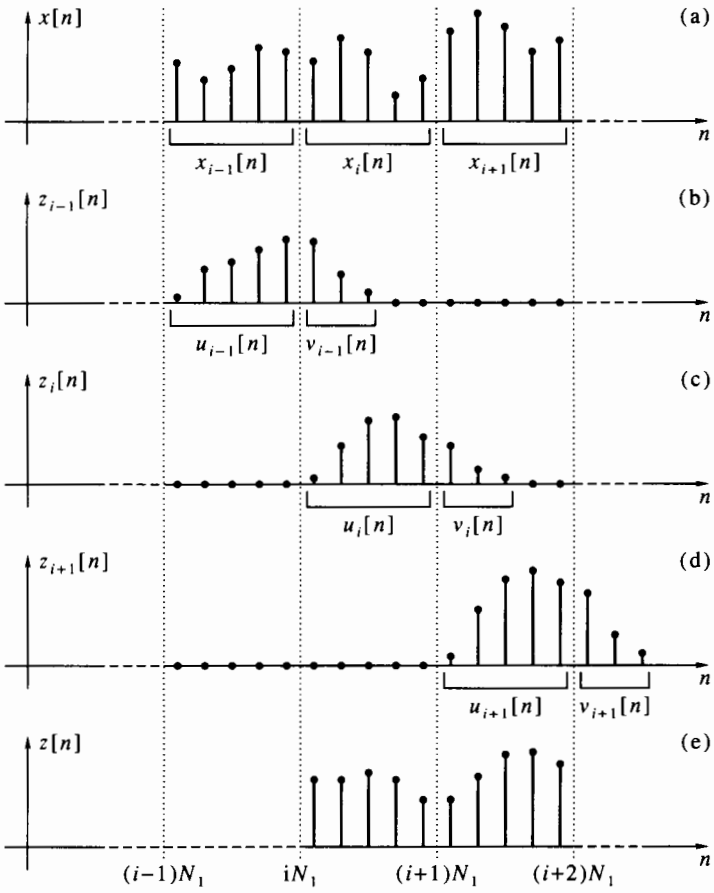

Figure 5.8 Illustration of overlap-add convolution: (a) the input signal $x[n]$; (b), (c), (d) the three output segments $z_{i-1}[n]$ through $z_{i+1}[n]$; (e) a segment of the output signal $z[n]$ corresponding to $x_i[n]$ and $x_{i+1}[n]$.

The procedure `ola` in Program 5.4 is a MATLAB implementation of the overlap-add method. The length of the sequence `x` is assumed to be much greater than the length of `y`. The FFT of `y` is computed first and used throughout. Then the program loops over the segments, and finally handles the tail of the sequence `x`, which may be shorter than the other segments.

Linear convolution of a long sequence $x[n]$ by a fixed-length sequence $y[n]$ can also be performed by a method called *overlap-save*, which is dual to overlap-add. The details of the overlap-save method are left as an exercise (Problem 5.23).

We now discuss the optimal choice of the FFT length N in the overlap-add method. We assume that the sequences in question are real. As we saw in Section 5.5, this enables the simultaneous computation of the FFTs of two consecutive segments (Program 5.4 does not work this way, however). Since we have the freedom to choose N_1,

let us choose it such that $N_1 + N_2 - 1$ is a power of 2. The FFT of the zero-padded $y[n]$ needs to be computed only once, so we ignore the operations it requires. For each pair of segments of N_1 output points we need $4(N_1 + N_2 - 1)\log_2(N_1 + N_2 - 1)$ multiplications for the two FFTs (a direct one and an inverse one), as well as $4(N_1 + N_2 - 1)$ multiplications for the product of the FFTs. It is convenient, in this application, to count the number of operations per sample of the signal $x[n]$. Using the overlap-add method for convolution is more efficient than direct convolution when the number of operations per sample is smaller. We thus get the criterion

$$2\left(1 + \frac{N_2 - 1}{N_1}\right)[1 + \log_2(N_1 + N_2 - 1)] < N_2. \tag{5.38}$$

Table 5.2 shows the optimal choice of the parameter $N_1 + N_2 - 1$ for various values of N_2. The optimum is defined as the value for which the left side of (5.38) is minimal, under the constraint that this parameter is an integer power of 2. For $N_2 < 19$ the inequality (5.38) does not hold, meaning that direct convolution is more efficient than overlap-add.

Range of N_2	Optimal $N_1 + N_2 - 1$
19-26	128
27-47	256
48-86	512
87-158	1024

Table 5.2 Optimal choice of the segment length as a function of the length of $y[n]$ for OLA convolution of real sequences by radix-2 FFT.

5.7 DFT at a Selected Frequency Range

5.7.1 The Chirp Fourier Transform

Since the FFT is an implementation of the DFT, it provides a frequency resolution of $2\pi/N$, where N is the length of the input sequence. If this resolution is not sufficient in a given application, we have the option of zero padding the input sequence, as described in Section 4.4. However, this may be unduly expensive in operation count if the required resolution is much higher than $2\pi/N$. In practice, one is seldom interested in high resolution over the entire frequency band. More often one is interested only in a relatively small part of the band. For example, suppose we wish to determine the frequency θ_0 of a sinusoid to a good accuracy from N data points. Performing FFT on the data will enable determination of θ_0 to an accuracy $\pm\pi/N$. If we could compute $X^f(\theta)$ at high resolution in an interval of width $2\pi/N$ around the maximum point of the FFT, we might be able to improve the accuracy of θ_0. A special technique, called *chirp Fourier transform*, accomplishes this.[2]

Suppose we are given a sequence $\{x[n],\ 0 \le n \le N-1\}$ and we wish to compute

$$X^f(\theta[k]) = \sum_{n=0}^{N-1} x[n]e^{-j\theta[k]n}, \tag{5.39}$$

where $\Delta\theta$ is the desired frequency resolution and

$$\theta[k] = \theta_0 + k\Delta\theta, \quad 0 \le k \le K-1. \tag{5.40}$$

Figure 5.9 illustrates the desired operation, showing $X^f(\theta)$ and the desired range of computation, as well as a magnification of the desired range and the desired transform values $\{X^f(\theta[k]), 0 \le k \le K-1\}$.

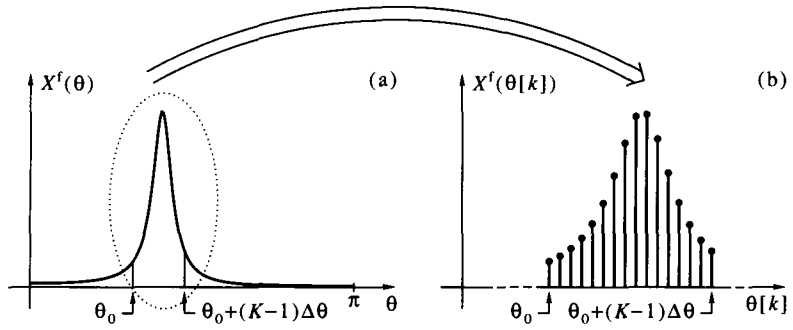

Figure 5.9 Chirp Fourier transform: (a) Full range of $X^f(\theta)$; (b) selected range of $X^f(\theta)$ and the sampled values $X^f(\theta[k])$.

Direct computation of (5.39) would require NK complex multiply-add operations. This can be high if K is large: for example, if it is of the same order as N. The chirp Fourier transform, which we now describe, reduces the number of operations to a small multiple of $(N+K)\log_2(N+K)$.

Define $W = e^{j\Delta\theta}$. Then we can rewrite (5.39) as

$$X^f(\theta[k]) = \sum_{n=0}^{N-1} x[n]e^{-j\theta_0 n}W^{-nk}. \tag{5.41}$$

Let us now use the identity

$$nk = 0.5[n^2 + k^2 - (k-n)^2]$$

in (5.41) to get

$$X^f(\theta[k]) = W^{-0.5k^2}\sum_{n=0}^{N-1} x[n]e^{-j\theta_0 n}W^{-0.5n^2}W^{0.5(k-n)^2}. \tag{5.42}$$

Define the two sequences

$$g[n] = x[n]e^{-j\theta_0 n}W^{-0.5n^2}, \quad h[n] = W^{0.5n^2}. \tag{5.43}$$

Then we have from (5.42)

$$X^f(\theta[k]) = W^{-0.5k^2}\{g * h\}[k]. \tag{5.44}$$

The sequence $g[n]$ is finite, having the same length as $x[n]$, whereas $h[n]$ is infinite. However, since we are interested in $X^f(\theta[k])$ only on a finite set of ks, we need only the finite segment $\{h[n],\ -(N-1) \le n \le (K-1)\}$. We can therefore summarize the chirp Fourier transform algorithm as follows:

1. Find a number L that is an integer power of 2 and is not smaller than $N+K-1$.
2. Form the sequence $g[n]$ as defined in (5.43) and zero-pad it to a length L.
3. Form the sequence $\{h[n],\ 0 \le n \le K-1\}$, as defined in (5.43); concatenate to it the sequence $\{h[n],\ -(L-K) \le n \le -1\}$; note that the sequence thus formed is a circular shift of $h[n]$.

4. Perform the operation

$$\text{IFFT}\{\text{FFT}\{g[n]\} \cdot \text{FFT}\{h[n]\}\}.$$

Multiply the first K terms of the resulting sequence by $W^{-0.5k^2}$ to get the sequence $\{X^{\text{f}}(\theta[k]),\ 0 \le k \le K-1\}$.

Figure 5.10 shows a block diagram of the chirp Fourier transform algorithm, and the procedure `chirpf` in Program 5.5 gives its MATLAB implementation.

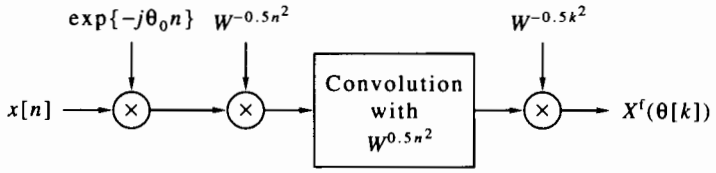

Figure 5.10 Block diagram of the chirp Fourier transform algorithm.

The computational complexity of the chirp Fourier transform algorithm is proportional to $L \log_2 L$, which is close to $(N+K)\log_2(N+K)$, plus a small multiple of N and another small multiple of K for the auxiliary sequences.

5.7.2 Zoom FFT*

The so-called *zoom FFT* is an alternative to chirp Fourier transform for computing the DFT of a sequence at a selected frequency range. Suppose we are interested in computing the DFT at the points

$$k_0 \le k \le k_0 + K - 1$$

for some k_0 and K. This problem is similar to the one posed in (5.39), (5.40) and depicted in Figure 5.9, with $\theta_0 = 2\pi k_0/N$ and $\Delta\theta = 2\pi/N$. Suppose further that the sequence $x[n]$ has length $N = KL$ for some integer L. The sequence may be zero padded if we want the frequency resolution $\Delta\theta = 2\pi/N$ to be better than the resolution corresponding to the given signal length.

Let the index n be expressed as

$$n = l + mL, \quad 0 \le l \le L-1,\ 0 \le m \le K-1.$$

Then

$$\begin{aligned} X^{\text{d}}[k] &= \sum_{n=0}^{N-1} x[n] W_N^{-kn} = \sum_{m=0}^{K-1} \sum_{l=0}^{L-1} x[l+mL] W_N^{-k(l+mL)} \\ &= \sum_{l=0}^{L-1} \left[\sum_{m=0}^{K-1} x[l+mL] W_K^{-km} \right] W_N^{-kl}. \end{aligned} \tag{5.45}$$

Let

$$X_l^{\text{d}}[r] = \sum_{m=0}^{K-1} x[l+mL] W_K^{-rm} \tag{5.46}$$

be the DFT of the decimated sequence $\{x[l+mL],\ 0 \le m \le K-1\}$. Then, since $X_l^{\text{d}}[r]$ is periodic in r with period K, we have

$$X_l^{\text{d}}[k] = X_l^{\text{d}}[k \bmod K].$$

We therefore conclude from (5.45), (5.46) that $X^{\text{d}}[k]$ can be computed in the desired frequency range using the following procedure:

1. Compute the K-point DFTs of the L decimated sequences $x[l + mL]$, as given in (5.46).
2. For each k in the desired range let

$$X^{\mathrm{d}}[k] = \sum_{l=0}^{L-1} X_l^{\mathrm{d}}[k \bmod K] W_N^{-kl}. \tag{5.47}$$

The first step of this procedure requires about $0.5LK(\log_2 K - 2)$ complex multiplications (assuming that K is an integer power of 2), and the second requires LK complex multiplications (we cannot do this step by FFT). Therefore, the total number of complex multiplications is about $0.5N \log_2 K$. In addition, we need all the numbers $\{W_N^n,\ 0 \le n \le N-1\}$. This is more efficient than N-point FFT if $L > 4$, and much more efficient if $L \gg K$. Also, it is usually convenient, if N is large, to perform several short FFTs, rather than one large FFT.

5.8 Summary and Complements

5.8.1 Summary

In this chapter we introduced fast Fourier transform algorithms, in particular the Cooley-Tukey algorithms. The CT decomposition of the DFT enables reducing the number of arithmetic operations whenever the length of the DFT is a composite number. The CT decomposition can be performed recursively, until only prime-length DFTs are left to be performed, and these are computed directly.

Of particular importance are radix-2 FFT algorithms, for the case in which the DFT length is an integer power of 2. The two main algorithms are the time decimated and the frequency decimated. The computational complexity of these algorithms is proportional to $N \log_2 N$. The radix-4 FFT algorithm is more efficient than radix-2 algorithms in multiplication count, but not in addition count.

An important use of FFT is for performing linear convolutions. We presented the overlap-add method for performing a linear convolution between a fixed-length sequence and a sequence of an indefinite length. Finally, we presented variations of the FFT that enable computation of the DFT at a selected frequency range: the chirp Fourier transform and the zoom FFT.

We have not treated FFT algorithms other than the ones based on the CT decomposition. Two alternative approaches, important in certain applications, are as follows:

1. Prime factor FFT [Good, 1958; Thomas, 1963; Burrus, 1988]. These algorithms are useful when the DFT length is a product of different primes, each appearing once in the factorization. It is possible, in this case, to perform the DFT without twiddle factor multiplications.
2. Winograd Fourier transform algorithms, based on performing DFTs of small prime numbers by convolutions [Winograd, 1978]. Winograd algorithms require a number of multiplications proportional to N (as opposed to $N \log_2 N$). However, they require considerably more additions than Cooley-Tukey FFT.

5.8.2 Complements

1. [p. 133] The term *fast algorithms* usually refers to computational schemes that implement certain calculations in a relatively small number of operations. An example from computer science: Direct sorting of N numbers requires about N^2

comparisons; sorting by the *quicksort* algorithm requires about $N \log_2 N$ comparisons on the average, so it is a fast algorithm.

2. [p. 151] A signal of the form

$$x(t) = e^{j0.5\alpha t^2}, \quad \alpha > 0,$$

is called a *chirp*. It is a complex signal whose magnitude is constant and whose frequency increases linearly in time. A chirp signal has the following property: If its duration is the interval $[0, T_0]$, its bandwidth is about αT_0. Therefore, bandwidth increases simultaneously with the duration. In radar and sonar applications, the signal duration determines the resolution to which the velocity of a target can be determined and the signal bandwidth determines the resolution to which the range can be determined. A chirp signal allows simultaneous control of the velocity resolution (through the parameter T_0) and the range resolution (through the parameter α). This property is the reason for its importance in radar and sonar applications. For further reading on this subject, see Rihaczek [1985].

We finally remark that the chirp Fourier transform has nothing to do with this particular application; it got its name because it uses the discrete-time chirp signal $W^{0.5n^2}$.

5.9 MATLAB Programs

Program 5.1 FFT and IFFT, by Cooley-Tukey frequency decimation recursion.

```
function X = edufft(x,swtch);
% Synopsis: X = edufft(x,swtch).
% An "educational" FFT algorithm, based on Cooley-Tukey frequency
% decimation procedure. Runs slow due to MATLAB recursion overhead.
% Input parameters:
% x: the input vector
% swtch: 0 for FFT, 1 for IFFT.
% Output:
% X: the output vector.

N = length(x); x = reshape(x,1,N);
if (swtch), W = exp((j*2*pi/N)*(0:N-1));
else, W = exp((-j*2*pi/N)*(0:N-1)); end
X = ctrecur(x,W); if (swtch), X = (1/N)*X; end
```

Program 5.2 Cooley-Tukey frequency decimation recursion.

```
function X = ctrecur(x,W);
% Synopsis: X = ctrecur(x,W).
% Cooley-Tukey frequency-decimation recursion.
% Input parameters:
% x: the input vector, assumed to be a row
% W: vector of powers of W, assumed to be a row.
% Output:
% y: the output row vector.

N = length(x); Q = N;
for i = 2:floor(sqrt(N)),
   if (rem(N,i) == 0), Q = i; break, end, end

if (Q == N), X = primedft(x,W);
else,
   P = N/Q; tmp = reshape(x,P,Q);
   for p = 0:P-1,
   tmp(p+1,:) = primedft(tmp(p+1,:),W(1:P:N));
   if (p > 0), tmp(p+1,2:Q) = tmp(p+1,2:Q).*W(rem(p*(1:Q-1),N)+1); end
   end
   for q = 1:Q, tmp(:,q) = (ctrecur(tmp(:,q).',W(1:Q:N))).'; end
   X = reshape(tmp.',1,N);
end
```

Program 5.3 Brute-force DFT or conjugate DFT for prime length.

```
function y = primedft(x,W);
% Synopsis: y =primedft(x,W).
% Brute-force DFT or conjugate DFT in case of prime length.
% Input parameters:
% x: the input vector, assumed to be a row
% W: vector of powers of W, assumed to be a row.
% Output:
% y: the output row vector.

N = length(x); n = 1:N-1; y = zeros(1,N); y(1) = sum(x);
for k = 1:N-1,
   y(k+1) = x(1) + sum(x(2:N).*W(rem(k*n,N)+1)); end
```

Program 5.4 Overlap-add convolution.

```
function z = ola(x,y,N);
% Synopsis: z = ola(x,y,N).
% Computes the convolution z = x*y by the overlap-add method.
% Input parameters:
% x: the long input sequence
% y: the short  input sequence
% N: length of the FFT.
% Output parameters:
% z: the output sequence.

N2 = length(y); lx = length(x); y = reshape(y,1,N2);
x = reshape(x,1,lx); z = zeros(1,N2+lx-1);
lz = length(z); rflag = 0; N1 = N-N2+1;
if (any(imag(x)) | any(imag(y))), rflag = 1; end
nframe = floor(lx/N1); ltail = lx - N1*nframe;
Y = fft([y, zeros(1,N-N2)]);
for k = 1:nframe,
   nst = (k-1)*N1;
   temp = ifft(fft([x(1,nst+1:nst+N1), zeros(1,N-N1)]).*Y);
   if (rflag), temp = real(temp); end
   z(1,nst+1:nst+N) = z(1,nst+1:nst+N) + temp;
end
if (ltail > 0),
   nst = nframe*N1+1; temp = [x(1,nst:lx), zeros(1,N-ltail)];
   temp = ifft(fft(temp).*Y);
   if (rflag), temp = real(temp); end
   z(1,nst:lz) = z(1,nst:lz) + temp(1,1:N2+ltail-1);
end
```

Program 5.5 The chirp Fourier transform algorithm.

```
function X = chirpf(x,theta0,dtheta,K);
% Synopsis: X = chirpf(x,theta0,dtheta,K).
% Computes the chirp Fourier transform on a frequency
% interval.
% Input parameters:
% x  : the input vector
% theta0 : initial frequency (in radians)
% dtheta : frequency increment (in radians)
% K  : number of points on the frequency axis.
% Output:
% X: the chirp Fourier transform of x.

N = length(x); x = reshape(x,1,N); n = 0:N-1;
g = x.*exp(-j*(0.5*dtheta*n+theta0).*n);
L = 1; while (L < N+K-1), L = 2*L; end
g = [g, zeros(1,L-N)];
h = [exp(j*0.5*dtheta*(0:K-1).^2), ...
     exp(j*0.5*dtheta*(-L+K:-1).^2)];
X = ifft(fft(g).*fft(h));
X = X(1:K).*exp(-j*0.5*dtheta*(0:K-1).^2);
```

5.10 Problems

5.1 Explain why the relationships in (5.5) are true.

5.2 Write MATLAB statements that compute the integers p, q for given n, P, and Q, as defined in (5.2a).

5.3 In our count of operations for FFT algorithms, we assumed that a complex multiplication requires four real multiplications and two real additions. However, it is also possible to perform a complex multiplication with three real multiplications and three real additions. To see this, suppose $a = a_r + ja_i$ is an intermediate complex number appearing in the algorithm and $W = W_r + jW_i$ is a twiddle factor used to multiply a. Then

$$\begin{aligned} aW &= (a_r W_r - a_i W_i) + j(a_r W_i + a_i W_r) \\ &= [(a_r - a_i)W_i + a_r(W_r - W_i)] + j[(a_r - a_i)W_i + a_i(W_r + W_i)]. \end{aligned}$$

The numbers $(W_r - W_i)$ and $(W_r + W_i)$ can be prepared off line, since the W are themselves prepared off line. Then we are left with three multiplications and three additions that must be performed on line, since they depend on a.
Compute the total number of operations in the radix-2 algorithm when this scheme is used.

5.4 Verify (5.17) by direct multiplication of the three matrices.

5.5 Suppose we wish to implement a 27-point DFT by radix-3 time-decimated FFT. Write down explicitly the proper ordering of the 27 input data.

5.6 Write a MATLAB program `bitrev` that performs bit reversal. The calling sequence of the program should be

```
n = bitrev(r)
```

where r is a positive integer. The output n is a vector of length 2^r whose components are the integers from 0 to $2^r - 1$ in a bit-reversed order. Hint: It is convenient to implement the program recursively. Verify that the program does not call itself more times than necessary.

5.7 Let $N = p^r$, where p is a prime. Derive recursive expressions for the numbers of complex multiplications and complex additions in a radix-p time-decimated FFT, similarly to the derivation of the case $p = 2$. Then solve the recursions and find explicit expressions.

5.8 Let the sequence $x[n]$ have length $N = 2^{2r+1}$. Consider the following three methods for computing $X^d[k]$:

(a) By radix-2 FFT.

(b) Decomposition of N to $N = PQ$, $Q = 2$, $P = 4^r$, then using the CT decomposition such that the length-P DFTs are performed by radix-4 FFT.

(c) Zero padding the sequence to length 4^{r+1}, performing radix-4 FFT on the zero-padded sequence, then discarding the odd-index points of the transform.

Compute the number of complex multiplications and the number of complex additions for each of the methods and conclude which one is the most efficient.

5.9 Let the sequence $x[n]$ have length $N = 3 \times 4^r$. Suppose we are interested in the DFT of $x[n]$ at N or more equally spaced frequency points. Consider the following two methods of computation:

(a) Decomposition of N to $N = PQ$, $Q = 3$, $P = 4^r$, then using the CT decomposition such that the length-P DFTs are performed by radix-4 FFT.

(b) Zero padding the sequence to length 4^{r+1} and performing radix-4 FFT on the zero-padded sequence.

Compute the number of complex multiplications for each of the methods and conclude which one is the most efficient (when only multiplications are of concern).

5.10 We are given a sequence of length $N = 240$.

(a) Compute the number of complex multiplications and the number of complex additions needed in a full (i.e., recursive) CT decomposition.

(b) Compute the corresponding numbers of operations if the sequence is zero-padded to length 256 before the FFT.

5.11 It is required to compute the DFT of a sequence of length $N = 24$. Zero padding is permitted. Find the number of complex multiplications needed for each of the following solutions and state which is the most efficient:

(a) Cooley–Tukey recursive decomposition of 24 into primes.

(b) Zero padding to $M = 25$ and using radix-5 FFT.

(c) Zero padding to $M = 27$ and using radix-3 FFT.

(d) Zero padding to $M = 32$ and using radix-2 FFT.

(e) Zero padding to $M = 64$ and using radix-4 FFT.

5.12 Write down the twiddle factors $\{W_8^n,\ 0 \le n \le 7\}$ explicitly. Then show that multiplication of any complex number by W_8^n requires either no multiplications, or two multiplications at most.

5.13 Consider the sequences given in Problem 4.10. Let $\mathcal{M}_{c1}$ be the number of complex multiplications in zero padding $y[n]$ to the nearest power of 2, then performing radix-2 FFT on the zero-padded sequence. Let $\mathcal{M}_{c2}$ be the number of complex multiplications in computing the radix-2 FFTs of $x_1[n]$ and $x_2[n]$ first (zero padding as necessary), then computing the DFT of $y[n]$ using the result of Problem 4.10.

(a) If $N = 2^r$, $r \ge 1$, show that $\mathcal{M}_{c2} \le \mathcal{M}_{c1}$.

(b) If N is not an integer power of 2, does it remain true that $\mathcal{M}_{c2} \le \mathcal{M}_{c1}$? If so, prove it; if not, give a counterexample.

5.14 Count the number of multiplications and additions needed for linear convolution of sequences of lengths N_1, N_2, if computed directly. Avoid multiplications and additions of zeros.

5.15 We are given two real sequences $x[n]$, $y[n]$ of length 2^r each. Compute the number of real multiplications needed for the linear convolution of the two sequences, first by direct computation, and then by using radix-2 FFT in the most efficient manner.

5.16 Assume that the input signal to a DFT satisfies $|x[n]| \leq 1$ for all $0 \leq n \leq N-1$. Show that the largest possible value of any of the $|X^{\mathrm{d}}[k]|$ is N. Find all sequences $x[n]$ for which $|X^{\mathrm{d}}[k]| = N$ for some k.

5.17 In Section 5.5 we saw how to compute the DFTs of two real sequences by one complex DFT, thus saving about 50 percent of the operations. Show how to do the reverse operation: Suppose we are given the DFTs $X^{\mathrm{d}}[k]$, $Y^{\mathrm{d}}[k]$ of two real sequences, and show how to compute the signals $x[n]$, $y[n]$ by one complex IDFT.

5.18 In Section 5.5 we showed how to compute the DFTs of two real sequences simultaneously with almost 50 percent saving in computations. Suppose, however, that we need the DFT of only *one* real sequence $x[n]$ of even length. Show that this can also be done at only slightly more than 50 percent of the number of operations in a complex FFT. Hint: Form the complex input sequence

$$y[n] = x[2n] + jx[2n+1], \quad 0 \leq n \leq 0.5N - 1,$$

then use ideas from Section 5.5 and from the time-decimated radix-2 FFT. Count the number of operations needed to obtain the final result.

5.19* A discrete-time periodic signal $x[n]$ with period N is given at the input of a linear time invariant filter

$$H^z(z) = \sum_{l=0}^{L-1} h[l]z^{-l}.$$

Let $y[n]$ denote the output signal.

(a) Show that $y[n]$ is periodic and find its period.

(b) Suppose that $N = 2^r$, and that $2r < L < N$. Show how to compute in an efficient manner the samples of $y[n]$ over one period.

(c) Repeat part b when L is an integer multiple of N, say $L = MN$. Show that the number of operations is only slightly larger than that in part b.

(d) Repeat part b when L is larger than N, but not necessarily an integer multiple of N.

5.20* Obtain a factorization of $\boldsymbol{F}_8$ to a product of five matrices from the time-decimated radix-2 FFT; use Figure 5.5.

5.21* Equations such as (5.22) can be solved by converting them to linear difference equations in the variable $r = \log_2 N$. With this change of variable, they become

$$\mathcal{A}_{\mathrm{c}}[r] = 2\mathcal{A}_{\mathrm{c}}[r-1] + 2^r, \tag{5.48a}$$

$$\mathcal{M}_{\mathrm{c}}[r] = 2\mathcal{M}_{\mathrm{c}}[r-1] + 2^{r-1} - 1. \tag{5.48b}$$

These difference equations can be solved using techniques studied in Chapter 7. If you know how to solve linear difference equations, continue to solve this problem now. If not, defer it until you have studied Chapter 7.

(a) Solve the difference equations and verify (5.23).

(b) Use the same technique to obtain and solve difference equations for radix-4 FFT, and verify the radix-4 operation count (5.27).

5.22* The purpose of this problem is to present the principle of operation of *split-radix FFT*. Here we derive the time-decimated version.

(a) Assume that N, the length of the input sequence, is an integer multiple of 4. Decimate the input sequence $x[n]$ as follows. First take the even-index elements $\{x[2m],\ 0 \le m \le 0.5N-1\}$ and assume we have computed their $0.5N$-DFT. Denote the result by $\{F^{\mathrm{d}}[k],\ 0 \le k \le 0.5N-1\}$. Next take the elements $\{x[4m+1],\ 0 \le m \le 0.25N-1\}$ and assume we have computed their $0.25N$-DFT. Denote the result by $\{G^{\mathrm{d}}[k],\ 0 \le k \le 0.25N-1\}$. Finally take the elements $\{x[4m+3],\ 0 \le m \le 0.25N-1\}$ and assume we have computed their $0.25N$-DFT. Denote the result by $\{H^{\mathrm{d}}[k],\ 0 \le k \le 0.25N-1\}$. Show that $X^{\mathrm{d}}[k],\ 0 \le k \le N-1\}$ can be computed in terms of $F^{\mathrm{d}}[k], G^{\mathrm{d}}[k], H^{\mathrm{d}}[k]$.

(b) Draw the butterfly that describes the result of part a. Count the number of complex operations in the butterfly, and the number of operations for twiddle factor multiplications. Remember that multiplication by j does not require actual multiplication, only exchange of the real and imaginary parts.

(c) Take $N = 16$ and count the total number of butterflies of the type you obtained in part b that are needed for the computation. Note that eventually N ceases to be an integer multiple of 4 and then we must use the usual 2×2 butterflies. Count the number of those as well. Also, count the number of twiddle factor multiplications.

(d) Repeat part c for $N = 32$. Attempt to draw a general conclusion about the multiple of $N \log_2 N$ that appears in the count of complex multiplications.

Split-radix FFT is more efficient than radix-2 or radix-4 FFT. However, it is more complicated to program, hence it is less widely used.

5.23* The purpose of this problem is to develop the overlap-save method of linear convolution, in a manner similar to the overlap-add method. We denote the long sequence by $x[n]$, the fixed-length sequence by $y[n]$, and the length of $y[n]$ by N_2. We take N as a power of 2 greater than N_2, and denote $N_1 = N - N_2 + 1$. So far everything is the same as in the case of overlap-add.

(a) Show that, if $y[n]$ is zero-padded to length N and circular convolution is performed between the zero-padded sequence and a length-N segment of $x[n]$, then the last N_1 points of the result are identical to corresponding N_1 points of the linear convolution, whereas the first $N_2 - 1$ points are different. Specify the range of indices of the linear convolution thus obtained.

(b) Break the input sequence $x[n]$ into overlapping segments of length N each, where the overlap is $N_2 - 1$. Denote the ith segment by $\{x_i[n],\ 0 \le n \le N-1\}$. Express $x_i[n]$ in terms of a corresponding point of $x[n]$.

(c) Show how to discard parts of the circular convolutions $\{x_i \circledast y\}[n]$ and patch the remaining parts together so as to obtain the desired linear convolution $\{x * y\}[n]$.

5.24* Program the zoom FFT (see Section 5.7.2) in MATLAB. The inputs to the program are the sequence $x[n]$, and initial index k_0, and the number of frequency points K. Hints: (1) Zero-pad the input sequence if its length is not a product of K; (2) use the MATLAB feature of performing FFTs of all columns of a matrix simultaneously.

5.25* Using the material in Section 4.9, write a MATLAB program `fdct` that computes the four DCTs using FFT. The calling sequence of the program should be

```
X = fdct(x,typ)
```

where `x` is the input vector, `typ` is the DCT type (from 1 to 4), and `X` is the output vector.