CHAPTER 12

The Fast Fourier Transform

There are several ways to calculate the Discrete Fourier Transform (DFT), such as solving simultaneous linear equations or the *correlation* method described in Chapter 8. The Fast Fourier Transform (FFT) is another method for calculating the DFT. While it produces the same result as the other approaches, it is incredibly more efficient, often reducing the computation time by *hundreds*. This is the same improvement as flying in a jet aircraft versus walking! If the FFT were not available, many of the techniques described in this book would not be practical. While the FFT only requires a few dozen lines of code, it is one of the most complicated algorithms in DSP. But don't despair! You can easily use published FFT routines without fully understanding the internal workings.

Real DFT Using the Complex DFT

J.W. Cooley and J.W. Tukey are given credit for bringing the FFT to the world in their paper: "An algorithm for the machine calculation of complex Fourier Series," *Mathematics Computation*, Vol. 19, 1965, pp 297-301. In retrospect, others had discovered the technique many years before. For instance, the great German mathematician Karl Friedrich Gauss (1777-1855) had used the method more than a century earlier. This early work was largely forgotten because it lacked the tool to make it practical: the *digital computer*. Cooley and Tukey are honored because they discovered the FFT at the right time, the beginning of the computer revolution.

The FFT is based on the *complex DFT,* a more sophisticated version of the *real DFT* discussed in the last four chapters. These transforms are named for the way each represents data, that is, using complex numbers or using real numbers. The term *complex* does not mean that this representation is difficult or complicated, but that a specific type of mathematics is used. Complex mathematics often *is* difficult and complicated, but that isn't where the name comes from. Chapter 29 discusses the complex DFT and provides the background needed to understand the details of the FFT algorithm. The

章

12

快速傅里叶变换

计算离散傅里叶变换（DFT）有多种方法，例如求解联立方程组或采用第八章介绍的*相关法*。快速傅里叶变换（FFT）是另一种计算DFT的方法。虽然它能产生与其他方法相同的结果，但效率要高得多，通常能将计算时间缩短*数百倍*。这就好比坐喷气式飞机飞行与步行相比，效率提升简直天差地别！如果 FFT 不可用，本书描述的许多技术将难以实际应用。虽然 FFT 仅需几十行代码，却是数字信号处理领域最复杂的算法之一。不过别灰心！即使不完全理解其内部原理，你也能轻松使用现成的 FFT 程序。

用复数DFT求解Real DFT

J.W.库利和J.W.图基因在论文《复杂 Fourier Series 的机器计算算法》（*Mathematics Computation*，第19卷，1965年，第297-301页）中将该 FFT 引入世界而获得学术认可。回溯历史，这项技术其实早在多年前就被他人发现。例如，德国数学巨匠卡尔·弗里德里希·高斯（1777-1855）早在一个多世纪前就已运用过该方法。由于缺乏使其实用化的工具——*数字计算机*，这些早期成果大多被遗忘。库利和图基之所以受到表彰，是因为他们在计算机革命初期恰逢其时地发现了这一 FFT 。

该 FFT 基于*复数DFT*，这是前四章讨论的*实数DFT*的进阶版本。这些变换的命名源于其数据表示方式——即使用复数或实数。术语*复数*并非指表示方式复杂，而是指采用了特定数学形式。复数数学本身往往*复杂*，但这并非名称由来。第29章将详细探讨复数DFT，并为理解 FFT 算法细节提供必要背景知识。

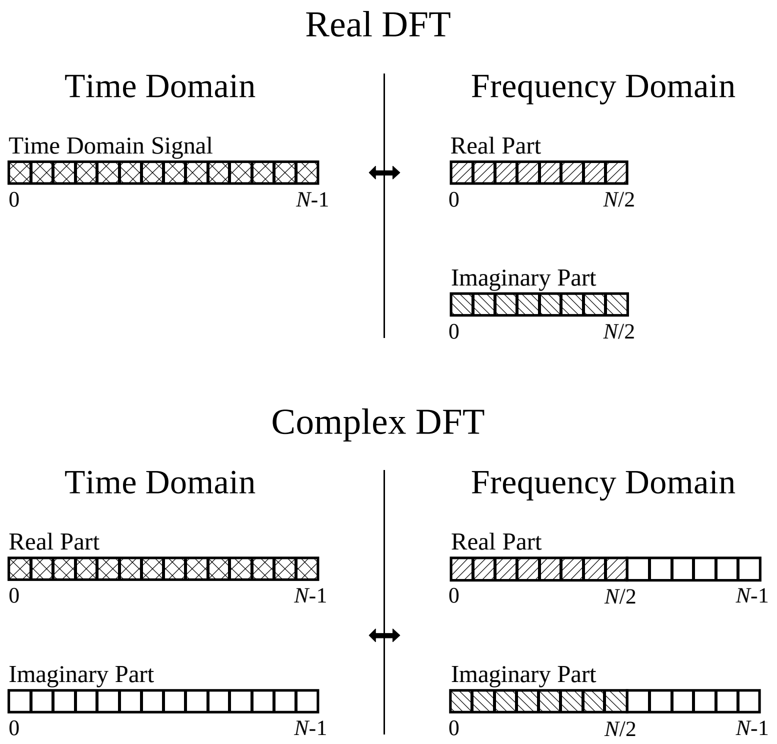

FIGURE 12-1
Comparing the real and complex DFTs. The real DFT takes an N point time domain signal and creates two $N/2+1$ point frequency domain signals. The complex DFT takes two N point time domain signals and creates two N point frequency domain signals. The crosshatched regions shows the values common to the two transforms.

topic of this chapter is simpler: how to use the FFT to calculate the real DFT, without drowning in a mire of advanced mathematics.

Since the FFT is an algorithm for calculating the complex DFT, it is important to understand how to transfer *real DFT* data into and out of the *complex DFT* format. Figure 12-1 compares how the real DFT and the complex DFT store data. The real DFT transforms an N point time domain signal into two $N/2+1$ point frequency domain signals. The time domain signal is called just that: the *time domain signal*. The two signals in the frequency domain are called the *real part* and the *imaginary part*, holding the amplitudes of the cosine waves and sine waves, respectively. This should be very familiar from past chapters.

In comparison, the complex DFT transforms two N point time domain signals into two N point frequency domain signals. The two time domain signals are called the *real part* and the *imaginary part*, just as are the frequency domain signals. In spite of their names, all of the values in these arrays are just ordinary numbers. (If you are familiar with complex numbers: the j's are not included in the array values; they are a part of the *mathematics*. Recall that the operator, *Im*( ), returns a real number).

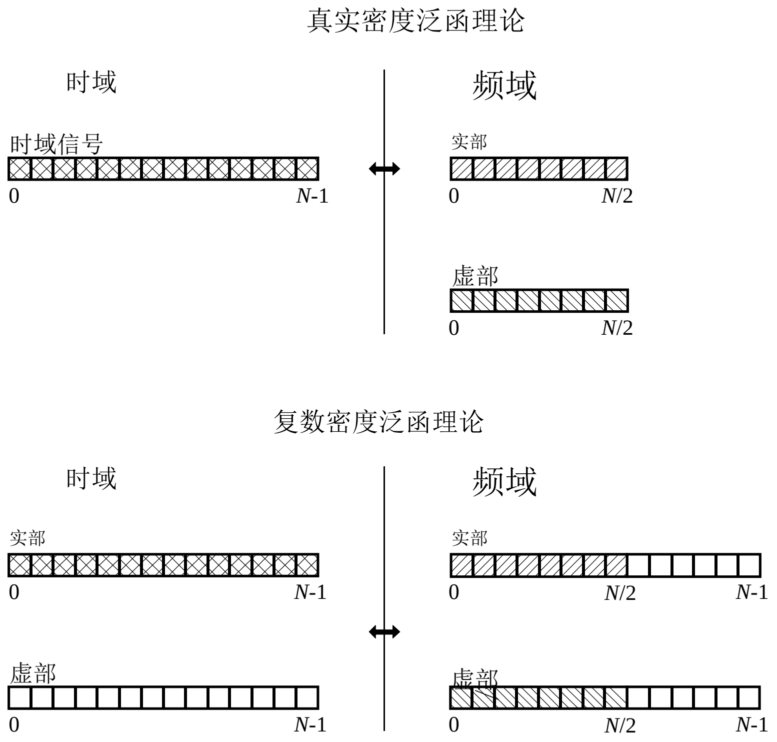

图12-1
比较实数和复数DFT。实数DFT接收一个N点时域信号，并生成两个$N/2+1$点频域信号。复数DFT接收两个N点时域信号，并生成两个N点频域信号。交叉阴影区域显示两种变换的共同值。

本章主题较为简单：如何运用 FFT 计算实际DFT，避免深陷高等数学的泥潭。

由于 FFT 是一种用于计算复数DFT的算法，因此理解如何将*实DFT*数据转换为*复DFT*格式并反向转换至关重要。图12-1对比了实DFT与复DFT的数据存储方式。实DFT将一个N点时域信号转换为两个$N/2+1$点频域信号。时域信号仍保持其名称：即*时域信号*。频域中的两个信号分别称为*实部*和*虚部*，分别表示余弦波和正弦波的振幅。这部分内容应与前几章内容高度吻合。

相比之下，复数密度泛函理论（DFT）将两个N点时域信号转换为两个N点频域信号。这两个时域信号被称为*实部*和*虚部*，与频域信号的命名方式相同。尽管名称如此，这些数组中的所有数值都只是普通数字。（如果你熟悉复数：数组值中不包含j；它们属于*数学*范畴。请记住运算符Im（）返回的是实数）。

Suppose you have an N point signal, and need to calculate the *real DFT* by means of the *Complex DFT* (such as by using the FFT algorithm). First, move the N point signal into the real part of the complex DFT's time domain, and then set all of the samples in the imaginary part to *zero*. Calculation of the complex DFT results in a real and an imaginary signal in the frequency domain, each composed of N points. Samples 0 through $N/2$ of these signals correspond to the real DFT's spectrum.

As discussed in Chapter 10, the DFT's frequency domain is periodic when the negative frequencies are included (see Fig. 10-9). The choice of a single period is arbitrary; it can be chosen between -1.0 and 0, -0.5 and 0.5, 0 and 1.0, or any other one unit interval referenced to the sampling rate. The complex DFT's frequency spectrum includes the negative frequencies in the 0 to 1.0 arrangement. In other words, one full period stretches from sample 0 to sample $N-1$, corresponding with 0 to 1.0 times the sampling rate. The positive frequencies sit between sample 0 and $N/2$, corresponding with 0 to 0.5. The other samples, between $N/2+1$ and $N-1$, contain the negative frequency values (which are usually ignored).

Calculating a *real Inverse DFT* using a *complex Inverse DFT* is slightly harder. This is because you need to insure that the negative frequencies are loaded in the proper format. Remember, points 0 through $N/2$ in the complex DFT are the same as in the real DFT, for both the real and the imaginary parts. For the real part, point $N/2+1$ is the same as point $N/2-1$, point $N/2+2$ is the same as point $N/2-2$, etc. This continues to point $N-1$ being the same as point 1. The same basic pattern is used for the imaginary part, except the sign is changed. That is, point $N/2+1$ is the negative of point $N/2-1$, point $N/2+2$ is the negative of point $N/2-2$, etc. Notice that samples 0 and $N/2$ do not have a matching point in this duplication scheme. Use Fig. 10-9 as a guide to understanding this symmetry. In practice, you load the real DFT's frequency spectrum into samples 0 to $N/2$ of the complex DFT's arrays, and then use a subroutine to generate the negative frequencies between samples $N/2+1$ and $N-1$. Table 12-1 shows such a program. To check that the proper symmetry is present, after taking the inverse FFT, look at the imaginary part of the time domain. It will contain all zeros if everything is correct (except for a few parts-per-million of noise, using single precision calculations).

```
6000 'NEGATIVE FREQUENCY GENERATION
6010 'This subroutine creates the complex frequency domain from the real frequency domain.
6020 'Upon entry to this subroutine, N% contains the number of points in the signals, and
6030 'REX[ ] and IMX[ ] contain the real frequency domain in samples 0 to N%/2.
6040 'On return, REX[ ] and IMX[ ] contain the complex frequency domain in samples 0 to N%-1.
6050 '
6060 FOR K% = (N%/2+1) TO (N%-1)
6070   REX[K%] =  REX[N%-K%]
6080   IMX[K%] = -IMX[N%-K%]
6090 NEXT K%
6100 '
6110 RETURN
```

TABLE 12-1

假设你有一个*N*点信号，需要通过*复数DFT*（例如使用 FFT 算法）来计算*实数DFT*。首先，将*N*点信号移入复数DFT时域的实部，然后将虚部的所有样本设为*零*。复数DFT的计算结果在频域中产生一个实信号和一个虚信号，每个信号由*N*个点组成。这些信号的第0到*N*/2个样本对应于实数DFT的频谱。

【补充】：

如第10章所述，当包含负频率时，DFT的频域呈现周期性（见图10-9）。单周期的选择具有任意性，可在-1.0至0、-0.5至0.5、0之间进行选择。1.0，或任何其他以采样率为基准的单位间隔。复数DFT的频谱包含从0到1.0排列中的负频率。换言之，一个完整周期从样本0延伸到样本*N*-1，对应于0到1.0倍的采样率。正频率位于样本0与*N*/2之间，对应于0到0.5。其余样本位于*N*/2+1与*N*-1之间，包含负频率值（通常被忽略）。

【补充】：

使用*复数逆DFT*计算*实数逆DFT*会稍显困难。这是因为需要确保负频点以正确格式加载。需注意，复数DFT中点0至*N*/2与实数DFT中的点在实部和虚部上完全相同。对于实部，点*N*/2+1等同于点*N*/2-1，点*N*/2+2等同于点*N*/2-2，依此类推，点*N*-1等同于点1。虚部采用相同的基本模式，仅符号相反：点*N*/2+1为点*N*/2-1的负值，点*N*/2+2为点*N*/2-2的负值，依此类推。请注意，样本0和*N*/2在此复制方案中没有匹配点。请参考图10-9来理解这种对称性。实际操作中，将真实DFT的频谱加载到复DFT数组的样本0至*N*/2，然后使用子程序在样本*N*/2+1与*N*-1之间生成负频率。表12-1展示了此类程序。为验证正确对称性，取逆 FFT 后观察时域虚部：若一切正常（使用单精度计算时仅存在百万分之几的噪声），该部分应全部为零。

```
6000 负频生成
6010 ‘该子程序从实频域生成复频域。6020′ 进入该子程序时，N%表示信号的采样点
数，6030 ‘REX[]和 IMX []则存储0到N%/2采样点的实频域数据。
6040 ‘返回时，REX[]和 IMX []包含0到N-1样本的复频域。6050 ′
6060 K% =（N%/2+1） 至（N%-1） 6070
REX[K%]  =  REX[N%-K%]  6080  IMX
[K%] = -IMX[N%-K%]
6090 下一个 K%
6100 '
6110 返回
```

表12-1

How the FFT works

The FFT is a complicated algorithm, and its details are usually left to those that specialize in such things. This section describes the general operation of the FFT, but skirts a key issue: the use of *complex numbers*. If you have a background in complex mathematics, you can read between the lines to understand the true nature of the algorithm. Don't worry if the details elude you; few scientists and engineers that use the FFT could write the program from scratch.

In complex notation, the time and frequency domains each contain *one signal* made up of *N complex points*. Each of these complex points is composed of two numbers, the real part and the imaginary part. For example, when we talk about complex sample $X[42]$, it refers to the combination of $ReX[42]$ and $ImX[42]$. In other words, each complex variable holds two numbers. When two complex variables are multiplied, the four individual components must be combined to form the two components of the product (such as in Eq. 9-1). The following discussion on *"How the FFT works"* uses this jargon of complex notation. That is, the singular terms: *signal*, *point*, *sample*, and *value*, refer to the *combination* of the real part and the imaginary part.

The FFT operates by decomposing an N point time domain signal into N time domain signals each composed of a single point. The second step is to calculate the N frequency spectra corresponding to these N time domain signals. Lastly, the N spectra are synthesized into a single frequency spectrum.

Figure 12-2 shows an example of the time domain decomposition used in the FFT. In this example, a 16 point signal is decomposed through four

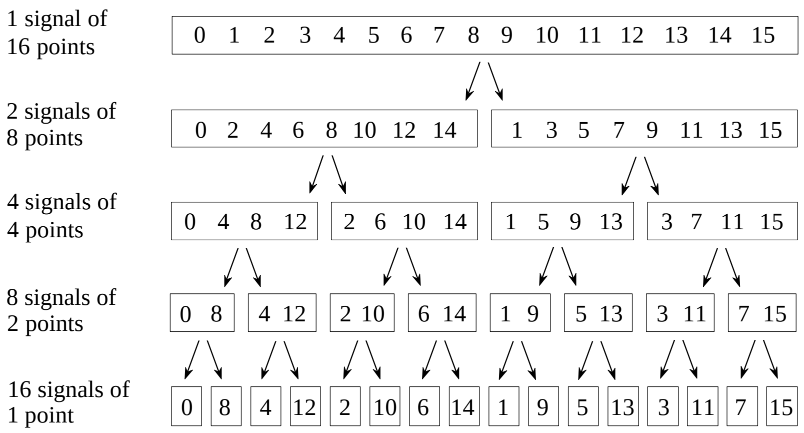

FIGURE 12-2
The FFT decomposition. An N point signal is decomposed into N signals each containing a single point. Each stage uses an *interlace decomposition*, separating the even and odd numbered samples.

FFT 如何工作

FFT 算法结构复杂，其具体实现通常由相关领域专家负责。本节虽阐述了 FFT 的基本运作原理，但刻意避开了一个核心问题——*复数的应用*。若具备复数数学背景，读者便能透过表面理解算法本质。若细节难以理解也无需担心，毕竟很少有使用 FFT 的科学家或工程师能从零编写程序。

【补充】：见笔记：复数记法与 FFT 的基本策略

在复数表示法中，时域和频域各自包含一个*信号*，该信号由N个*复数点*组成。每个复数点由两个数值构成，即实部和虚部。例如，当我们讨论复样本$X[42]$时，它指的是$ReX[42]$和$Im\ X[42]$的组合。换言之，每个复变量包含两个数值。当两个复变量相乘时，必须将四个独立分量组合成乘积的两个分量（如等式9-1所示）。下文关于“FFT 如何工作”的讨论将使用这种复数表示法的术语。也就是说，奇异项：*信号*、*点*、*样本*和*值*，指的是实部和虚部的*组合*。

该 FFT 通过将一个N点时域信号分解为N个时域信号来实现，每个时域信号由单个点组成。第二步是计算与这些N个时域信号对应的N个频谱。最后，将N个频谱合成到一个单一频谱中。

图12-2展示了 FFT 中采用的时间域分解示例。该示例中，一个16点信号通过四个

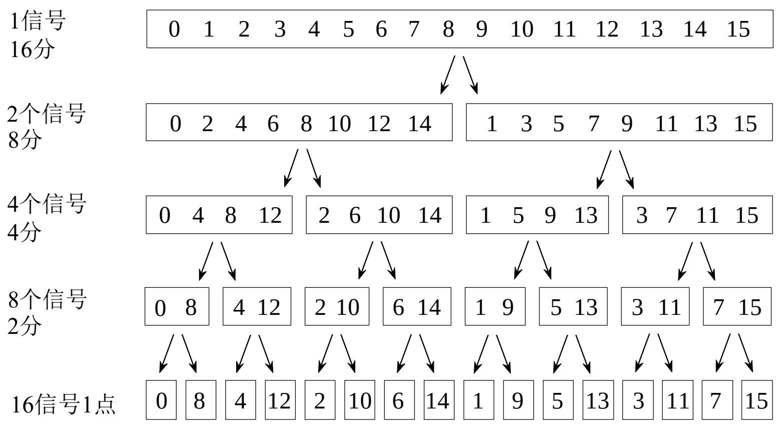

图12-2
FFT 分解。一个N点信号被分解为N个信号，每个信号包含一个点。每个阶段使用*交错分解*，将偶数和奇数编号的样本分开。

Sample numbers in normal order		Sample numbers after bit reversal	
Decimal	*Binary*	*Decimal*	*Binary*
0	0000	0	0000
1	0001	8	1000
2	0010	4	0100
3	0011	12	1100
4	0100	2	0010
5	0101	10	1010
6	0110	6	0100
7	0111	14	1110
8	1000	1	0001
9	1001	9	1001
10	1010	5	0101
11	1011	13	1101
12	1100	3	0011
13	1101	11	1011
14	1110	7	0111
15	1111	15	1111

FIGURE 12-3
The FFT bit reversal sorting. The FFT time domain decomposition can be implemented by sorting the samples according to bit reversed order.

separate stages. The first stage breaks the 16 point signal into two signals each consisting of 8 points. The second stage decomposes the data into four signals of 4 points. This pattern continues until there are N signals composed of a single point. An **interlaced decomposition** is used each time a signal is broken in two, that is, the signal is separated into its even and odd numbered samples. The best way to understand this is by inspecting Fig. 12-2 until you grasp the pattern. There are $Log_2 N$ stages required in this decomposition, i.e., a 16 point signal (2^4) requires 4 stages, a 512 point signal (2^7) requires 7 stages, a 4096 point signal (2^{12}) requires 12 stages, etc. Remember this value, $Log_2 N$; it will be referenced many times in this chapter.

Now that you understand the structure of the decomposition, it can be greatly simplified. The decomposition is nothing more than a *reordering* of the samples in the signal. Figure 12-3 shows the rearrangement pattern required. On the left, the sample numbers of the original signal are listed along with their binary equivalents. On the right, the rearranged sample numbers are listed, also along with their binary equivalents. The important idea is that the binary numbers are the *reversals* of each other. For example, sample 3 (0011) is exchanged with sample number 12 (1100). Likewise, sample number 14 (1110) is swapped with sample number 7 (0111), and so forth. The FFT time domain decomposition is usually carried out by a **bit reversal sorting** algorithm. This involves rearranging the order of the N time domain samples by counting in binary with the bits flipped left-for-right (such as in the far right column in Fig. 12-3).

样本数 正常顺序			样本数 位反转后	
十进位的	*二进制*		*十进位的*	*二进制*
0	0000		0	0000
1	0001		8	1000
2	0010		4	0100
3	0011		12	1100
4	0100		2	0010
5	0101		10	1010
6	0110	⇨	6	0100
7	0111		14	1110
8	1000		1	0001
9	1001		9	1001
10	1010		5	0101
11	1011		13	1101
12	1100		3	0011
13	1101		11	1011
14	1110		7	0111
15	1111		15	1111

图12-3

FFT 位反转排序。 FFT 时域分解可通过按位反转顺序对样本进行排序来实现。

分阶段处理。第一阶段将16点信号分解为两个各含8点的信号。第二阶段将数据分解为四个各含4点的信号。该模式持续进行，直至形成由单点构成的*N*个信号。每次信号被二分时均采用**交错分解**，即信号被分离为其偶数和奇数编号的样本。理解此过程的最佳方式是观察图12-2直至掌握其模式。该分解过程需要$Log_2 N$个阶段，即16点信号（2^4）需4个阶段，512点信号（2^7）需7个阶段，4096点信号（2^{12}）需12个阶段，依此类推。请牢记此数值$Log_2 N$；本章将多次引用该数值。

既然你已经理解了分解的结构，它就可以大大简化。分解只不过是信号中样本的*重新排序*。图12-3显示了所需的重新排列模式。
 左侧列出了原始信号的样本编号及其对应的二进制数值，右侧则展示了重新排列后的样本编号与二进制对应关系。其核心原理在于：这些二进制数值彼此*互为镜像*。例如，样本3（0011）与样本12（1100）互换位置，样本14（1110）与样本7（0111）同样互换，依此类推。 FFT 时域分解通常通过**位反转排序**算法实现，该算法通过按二进制计数方式将*N*个时域样本的顺序进行左到右的位反转排列（如图12-3最右侧列所示）。

The next step in the FFT algorithm is to find the frequency spectra of the 1 point time domain signals. Nothing could be easier; the frequency spectrum of a 1 point signal is equal to *itself*. This means that *nothing* is required to do this step. Although there is no work involved, don't forget that each of the 1 point signals is now a frequency spectrum, and not a time domain signal.

The last step in the FFT is to combine the *N* frequency spectra in the exact reverse order that the time domain decomposition took place. This is where the algorithm gets messy. Unfortunately, the bit reversal shortcut is not applicable, and we must go back one stage at a time. In the first stage, 16 frequency spectra (1 point each) are synthesized into 8 frequency spectra (2 points each). In the second stage, the 8 frequency spectra (2 points each) are synthesized into 4 frequency spectra (4 points each), and so on. The last stage results in the output of the FFT, a 16 point frequency spectrum.

Figure 12-4 shows how two frequency spectra, each composed of 4 points, are combined into a single frequency spectrum of 8 points. This synthesis must *undo* the interlaced decomposition done in the time domain. In other words, the frequency domain operation must correspond to the time domain procedure of *combining* two 4 point signals by interlacing. Consider two time domain signals, *abcd* and *efgh*. An 8 point time domain signal can be formed by two steps: dilute each 4 point signal with zeros to make it an

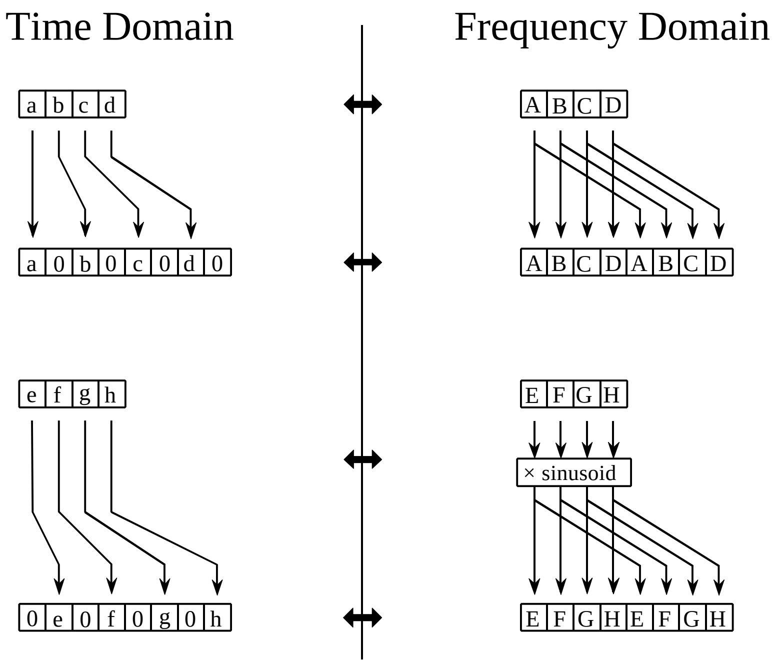

FIGURE 12-4
The FFT synthesis. When a time domain signal is diluted with zeros, the frequency domain is duplicated. If the time domain signal is also shifted by one sample during the dilution, the spectrum will additionally be multiplied by a sinusoid.

FFT 算法的下一步是找到1点时域信号的频谱。没有什么比这更容易的了；1点信号的频谱等于*它自己*。这意味着*不需要任何东西*来完成这一步。虽然没有工作涉及，但不要忘记，每个1点信号现在是一个频谱，而不是时域信号。

FFT 的最后一步是按照与时域分解完全相反的顺序合并*N*个频谱。这正是算法变得复杂的地方。遗憾的是，位反转捷径在此不适用，我们必须逐级回溯。第一阶段将16个频谱（每个1个点）合成为8个频谱（每个2个点）。第二阶段将8个频谱（每个2个点）合成为4个频谱（每个4个点），依此类推。最终阶段将生成 FFT 的输出结果——一个16点频谱。

图12-4展示了如何将两个由4个点组成的频谱合并为一个8点的频谱。这种合成必须*撤销*时域中进行的交织分解。换言之，频域操作必须对应于时域中通过交织*合并*两个4点信号的过程。考虑两个时域信号*abcd*和*efgh*。通过两个步骤可以形成一个8点时域信号：用零点稀释每个4点信号使其成为

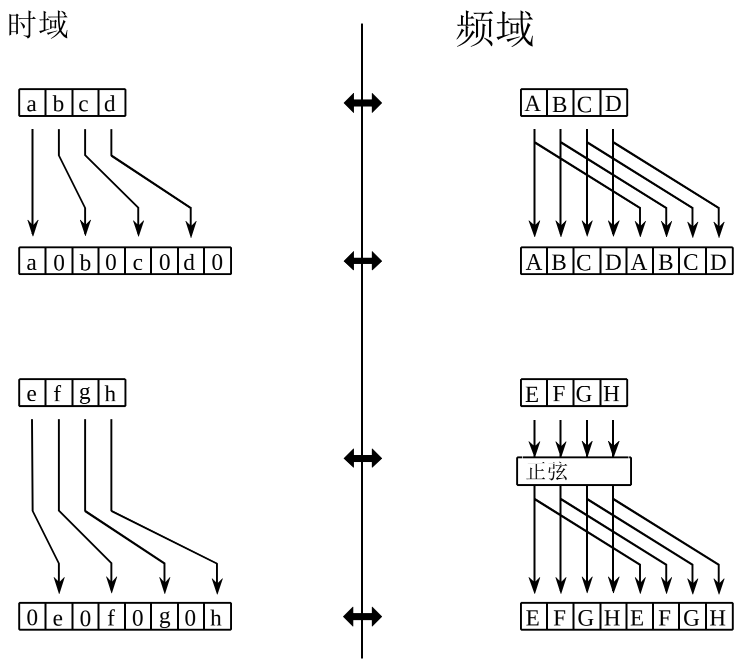

图12-4
FFT 合成。当时域信号被零值稀释时，频域信号会复制。若稀释过程中时域信号还发生了一个采样点的偏移，频谱还会额外乘以一个正弦波。

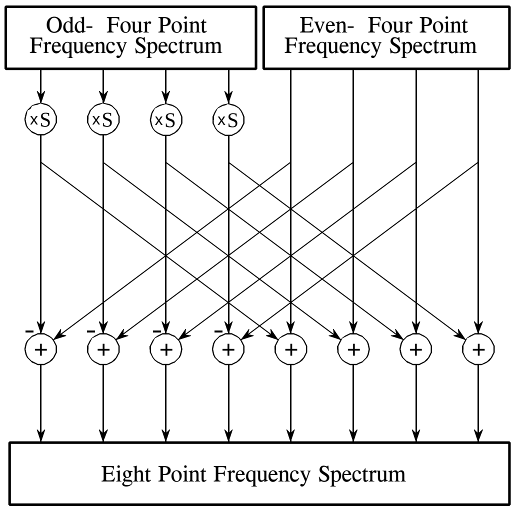
Odd- Four Point Frequency Spectrum
Even- Four Point Frequency Spectrum
×S
×S
×S
×S
-
-
-
-
+
+
+
+
+
+
+
+
Eight Point Frequency Spectrum

FIGURE 12-5
FFT synthesis flow diagram. This shows the method of combining two 4 point frequency spectra into a single 8 point frequency spectrum. The ×S operation means that the signal is multiplied by a sinusoid with an appropriately selected frequency.

8 point signal, and then add the signals together. That is, *abcd* becomes *a0b0c0d0*, and *efgh* becomes *0e0f0g0h*. Adding these two 8 point signals produces *aebfcgdh*. As shown in Fig. 12-4, diluting the time domain with zeros corresponds to a *duplication* of the frequency spectrum. Therefore, the frequency spectra are combined in the FFT by duplicating them, and then adding the duplicated spectra together.

In order to match up when added, the two time domain signals are diluted with zeros in a slightly different way. In one signal, the *odd points* are zero, while in the other signal, the *even points* are zero. In other words, one of the time domain signals (*0e0f0g0h* in Fig. 12-4) is shifted to the right by one sample. This time domain shift corresponds to multiplying the spectrum by a *sinusoid*. To see this, recall that a shift in the time domain is equivalent to convolving the signal with a shifted delta function. This multiplies the signal's spectrum with the spectrum of the shifted delta function. The spectrum of a shifted delta function is a sinusoid (see Fig 11-2).

Figure 12-5 shows a flow diagram for combining two 4 point spectra into a single 8 point spectrum. To reduce the situation even more, notice that Fig. 12-5 is formed from the basic pattern in Fig 12-6 repeated over and over.

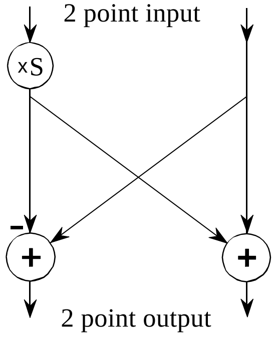

FIGURE 12-6
The FFT butterfly. This is the basic calculation element in the FFT, taking two complex points and converting them into two other complex points.

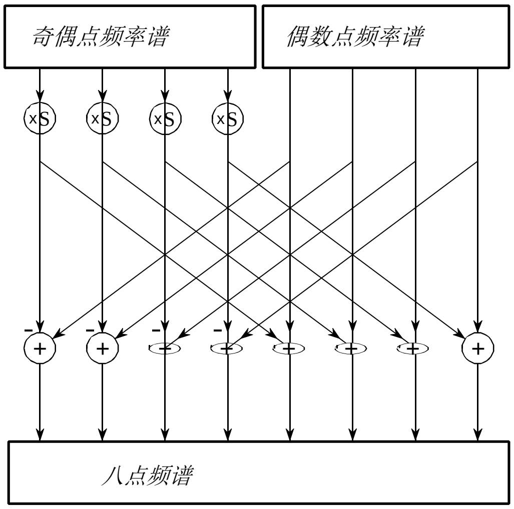

图12-5
FFT 合成流程图。该图展示了将两个4点频率谱合并为单一8点频率谱的方法。×S操作表示信号与频率适当选择的正弦波相乘。

8点信号，然后将这些信号相加。也就是说，*abcd*变为*a0b0c0d0*，而*efgh*变为*0e0f0g0h*。将这两个8点信号相加得到*aebfcgdh*。如图12-4所示，用零点稀释时域相当于对频谱进行*复制*。因此，频谱在 FFT 中通过复制后再将复制后的频谱相加来实现组合。

为了在叠加时保持同步，两个时域信号会以略微不同的方式用零值进行稀释处理。其中一个信号的*奇数点*被设为零，而另一个信号的*偶数点*被设为零。换句话说，其中一个时域信号（图12-4中的*0e0f0g0h*）会被右移一个采样点。这种时域偏移相当于将频谱乘以*正弦波*。具体来说，时域偏移等同于将信号与偏移后的δ函数进行卷积运算。这种操作会将信号频谱与偏移后的δ函数频谱相乘。而偏移后的δ函数频谱本身就是一个正弦波（参见图11-2）。

图12-5展示了将两个4点光谱合并为单一8点光谱的流程图。为进一步简化该过程，需注意图12-5是由图12-6中的基本模式通过重复叠加形成的。

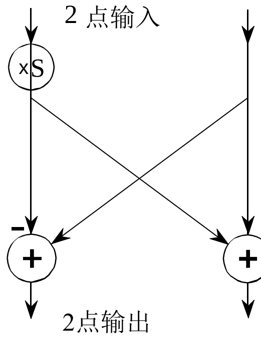

图12-6
FFT 蝴蝶。这是 FFT 中的基本计算单元，用于将两个复数点转换为另外两个复数点。

This simple flow diagram is called a **butterfly** due to its winged appearance. The butterfly is the basic computational element of the FFT, transforming two complex points into two other complex points.

Figure 12-7 shows the structure of the entire FFT. The time domain decomposition is accomplished with a bit reversal sorting algorithm. Transforming the decomposed data into the frequency domain involves *nothing* and therefore does not appear in the figure.

The frequency domain synthesis requires three loops. The outer loop runs through the $Log_2 N$ stages (i.e., each level in Fig. 12-2, starting from the bottom and moving to the top). The middle loop moves through each of the individual frequency spectra in the stage being worked on (i.e., each of the boxes on any one level in Fig. 12-2). The innermost loop uses the butterfly to calculate the points in each frequency spectra (i.e., looping through the samples inside any one box in Fig. 12-2). The overhead boxes in Fig. 12-7 determine the beginning and ending indexes for the loops, as well as calculating the sinusoids needed in the butterflies. Now we come to the heart of this chapter, the actual FFT programs.

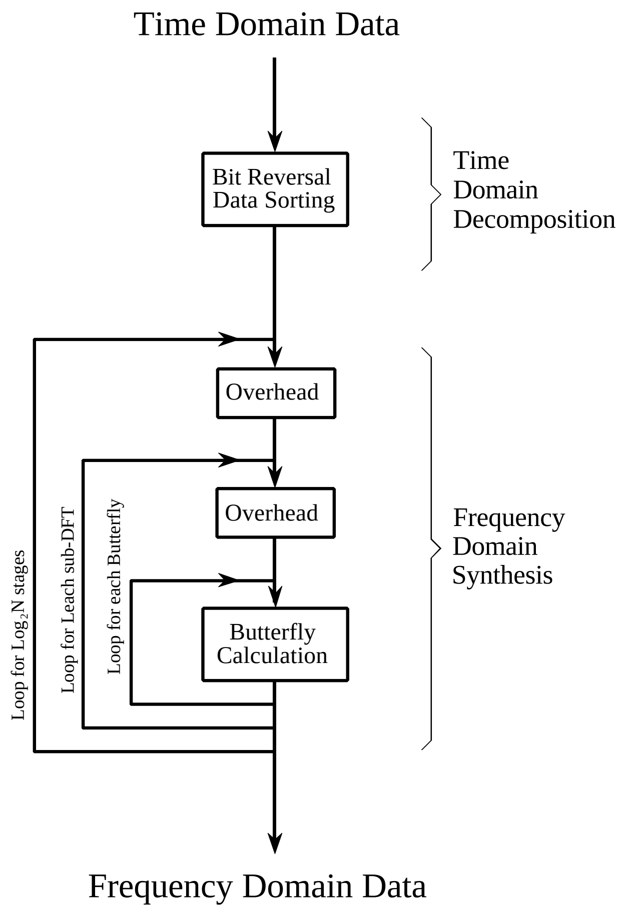

FIGURE 12-7
Flow diagram of the FFT. This is based on three steps: (1) decompose an *N* point time domain signal into *N* signals each containing a single point, (2) find the spectrum of each of the *N* point signals (nothing required), and (3) synthesize the *N* frequency spectra into a single frequency spectrum.

这个简单的流程图因其翅膀状外观被称为**蝴蝶**。蝴蝶是 FFT 的基本计算单元，将两个复数点转换为另外两个复数点。

图12-7展示了整个 FFT 的结构。时域分解采用位反转排序算法完成。将分解后的数据转换为频域涉及*无任何操作*，因此未在图中显示。

频域合成需要三个循环。外层循环遍历$Log_2 N$个阶段（即从图12-2底部到顶部的每一层）。中间循环遍历当前处理阶段中的各个独立频谱（即图12-2中任意一层的各个方框）。最内层循环使用蝶形滤波器计算各频谱中的点（即遍历图12-2中任意方框内的采样点）。图12-7中的辅助方框用于确定循环的起始和结束索引，并计算蝶形滤波器所需的正弦波。现在我们进入本章的核心内容——实际的 FFT 程序。

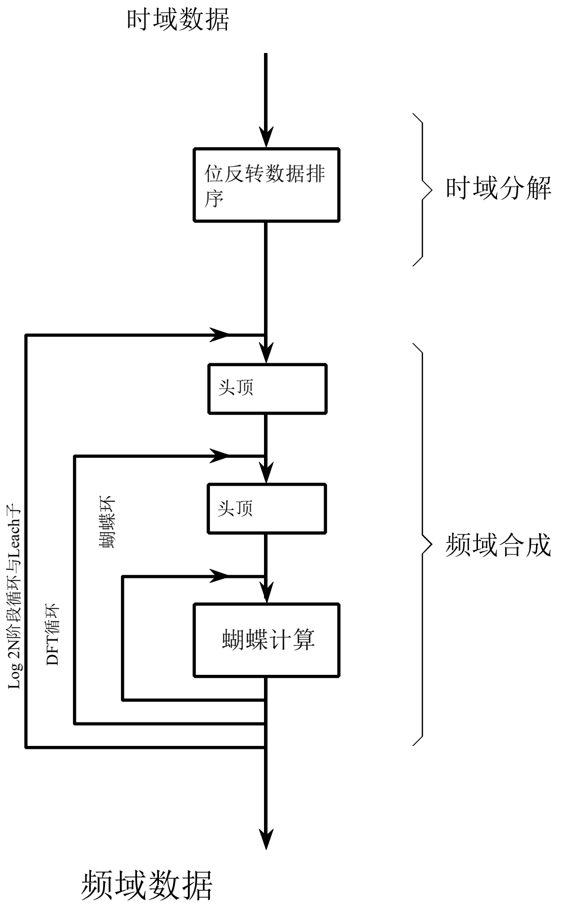

图12-7
FFT 的流程图。这是基于三个步骤:(1)将一个N点时域信号分解为N个信号，每个信号包含一个点，(2)找到每个N点信号的频谱（不需要），(3)将N个频谱合成到一个频谱。

FFT Programs

As discussed in Chapter 8, the *real DFT* can be calculated by correlating the time domain signal with sine and cosine waves (see Table 8-2). Table 12-2 shows a program to calculate the *complex DFT* by the same method. In an apples-to-apples comparison, this is the program that the FFT improves upon.

Tables 12-3 and 12-4 show two different FFT programs, one in FORTRAN and one in BASIC. First we will look at the BASIC routine in Table 12-4. This subroutine produces exactly the same output as the correlation technique in Table 12-2, except it does it *much faster*. The block diagram in Fig. 12-7 can be used to identify the different sections of this program. Data are passed to this FFT subroutine in the arrays: REX[ ] and IMX[ ], each running from sample 0 to $N-1$. Upon return from the subroutine, REX[ ] and IMX[ ] are overwritten with the frequency domain data. This is another way that the FFT is highly optimized; the same arrays are used for the input, intermediate storage, and output. This efficient use of memory is important for designing fast hardware to calculate the FFT. The term **in-place computation** is used to describe this memory usage.

While all FFT programs produce the same numerical result, there are subtle variations in programming that you need to look out for. Several of these

```
5000 'COMPLEX DFT BY CORRELATION
5010 'Upon entry, N% contains the number of points in the DFT, and
5020 'XR[ ] and XI[ ] contain the real and imaginary parts of the time domain.
5030 'Upon return, REX[ ] and IMX[ ] contain the frequency domain data.
5040 'All signals run from 0 to N%-1.
5050 '
5060 PI = 3.14159265                   'Set constants
5070 '
5080 FOR K% = 0 TO N%-1                'Zero REX[ ] and IMX[ ], so they can be used
5090   REX[K%] = 0                     'as accumulators during the correlation
5100   IMX[K%] = 0
5110 NEXT K%
5120 '
5130 FOR K% = 0 TO N%-1                'Loop for each value in frequency domain
5140   FOR I% = 0 TO N%-1              'Correlate with the complex sinusoid, SR & SI
5150 '
5160     SR =  COS(2*PI*K%*I%/N%)      'Calculate complex sinusoid
5170     SI  = -SIN(2*PI*K%*I%/N%)
5180     REX[K%] = REX[K%] + XR[I%]*SR - XI[I%]*SI
5190     IMX[K%] = IMX[K%] + XR[I%]*SI + XI[I%]*SR
5200 '
5210   NEXT I%
5220 NEXT K%
5230 '
5240 RETURN
```

TABLE 12-2

FFT 计划

如第8章所述，*实际DFT*可以通过将时域信号与正弦和余弦波相关联来计算（见表8-2）。表12-2显示了一个通过相同方法计算*复DFT*的程序。在同类比较中，这是 FFT 改进的程序。

表12-3和12-4展示了两种不同的 FFT 程序，一种使用Fortran语言编写，另一种使用Basic语言编写。首先我们来看表12-4中的Basic子程序。该子程序产生的输出与表12-2中的相关技术完全一致，但运行速度*快得多*。图12-7的框图可用于识别该程序的不同部分。数据通过数组REX[]和 IMX []传递给该 FFT 子程序，每个数组的运行范围从样本0到*N*-1。子程序返回后，REX[]和 IMX []会被频率域数据覆盖。这是 FFT 高度优化的另一种方式——相同的数组用于输入、中间存储和输出。这种高效的内存使用对于设计快速硬件计算 FFT 至关重要。术语**在地计算**用于描述这种内存使用方式。

虽然所有 FFT 程序都会得出相同的数值结果，但编程过程中存在一些需要特别注意的细微差异。其中几个

```
5000 的关联DFT
5010  ‘输入时，N%包含DFT中的点数，且
5020  ‘XR[ ]和XI[ ]包含时域的实部和虚部。
5030  ‘返回后，REX[ ]和 IMX [ ]将包含频域数据。
5040  ‘所有信号的取值范围为0至N%-1。
5050 '
5060 π = 3.14159265                                  “设置常量”
5070 '
5080 当K%等于0至N-1时                                零REX[ ]和 IMX [ ]，因此可直接使用
5090   REX[K%] = 0                                   作为相关性期间的累积量
5100   IMX[K%] = 0
5110 下一个K%
5120 '
5130 当K%等于0至N-1时                                频域内每个值的循环
5140 I% = 0 至 N%-1                        与复正弦波、SR及SI相关
5150 '
5160 SR = 硫化羰（2*PI*K%*I%/N%）  ‘计算复正弦波
5170 SI = -SIN（2*PI*K%*I%/N%）
5180    REX[K%] = REX[K%] + XR[I%]*SR - XI[I%]*SI 5190
IMX[K%] = IMX[K%] + XR[I%]*SI + XI[I%]*SR
5200 '
5210 下一个我
5220 下一个 K%
5230 '
5240 返回
```

表12-2

of these differences are illustrated by the FORTRAN program listed in Table 12-3. This program uses an algorithm called **decimation in frequency**, while the previously described algorithm is called **decimation in time**. In a decimation in frequency algorithm, the bit reversal sorting is done *after* the three nested loops. There are also FFT routines that completely eliminate the bit reversal sorting. None of these variations significantly improve the performance of the FFT, and you shouldn't worry about which one you are using.

The *important* differences between FFT algorithms concern how data are passed to and from the subroutines. In the BASIC program, data enter and leave the subroutine in the arrays REX[ ] and IMX[ ], with the samples running from index 0 to $N-1$. In the FORTRAN program, data are passed in the complex array *X*( ), with the samples running from 1 to *N*. Since this is an array of complex variables, each sample in X( ) consists of two numbers, a real part and an imaginary part. The length of the DFT must also be passed to these subroutines. In the BASIC program, the variable N% is used for this purpose. In comparison, the FORTRAN program uses the variable M, which is defined to equal $Log_2 N$. For instance, M will be

```
      SUBROUTINE  FFT(X,M)
      COMPLEX X(4096),U,S,T
      PI=3.14159265
      N=2**M
      DO 20 L=1,M
      LE=2**(M+1-L)
      LE2=LE/2
      U=(1.0,0.0)
      S=CMPLX(COS(PI/FLOAT(LE2)),-SIN(PI/FLOAT(LE2)))
      DO 20 J=1,LE2
      DO 10 I=J,N,LE
      IP=I+LE2
      T=X(I)+X(IP)
      X(IP)=(X(I)-X(IP))*U
10    X(I)=T
20    U=U*S
      ND2=N/2
      NM1=N-1
      J=1
      DO 50 I=1,NM1
      IF(I.GE.J) GO TO 30
      T=X(J)
      X(J)=X(I)
      X(I)=T
30    K=ND2
40    IF(K.GE.J) GO TO 50
      J=J-K
      K=K/2
      GO TO 40
50    J=J+K
      RETURN
      END
```

TABLE 12-3
The Fast Fourier Transform in FORTRAN. Data are passed to this subroutine in the variables *X*( ) and *M*. The integer, *M*, is the base two logarithm of the length of the DFT, i.e., *M* = 8 for a 256 point DFT, *M* = 12 for a 4096 point DFT, etc. The complex array, *X*( ), holds the time domain data upon entering the DFT. Upon return from this subroutine, *X*( ) is overwritten with the frequency domain data. Take note: this subroutine requires that the input and output signals run from *X*(1) through *X*(*N*), rather than the customary *X*(0) through *X*(*N*-1).

这些差异可通过表12-3中列出的fortran程序进行说明。该程序采用名为**频率抽取法**的算法，而前文所述算法则称为**时间抽取法**。在频率抽取法中，位反转排序操作是在三个嵌套循环*之后*执行的。此外还存在完全省略位反转排序的 FFT 例程。这些变体均未显著提升 FFT 性能，因此无需特别关注具体采用哪种方案。

*重要*的 FFT 算法差异在于数据如何在子程序间传递。在基础程序中，数据通过数组 REX[ ] 和 IMX [ ] 进入和退出子程序，样本从索引 0 到N-1运行。在 Fortran 程序中，数据通过复数数组X（）传递，样本从 1 到N运行。由于这是复变量数组，X（）中的每个样本由两个数值组成：实部和虚部。DFT 的长度也必须传递给这些子程序。在基础程序中，使用变量 N% 来实现此功能。相比之下，Fortran 程序使用变量 M，其定义为Log_2 N。例如，M 将是

```
      子程序 FFT（X，M）
      X复合体（4096），U，S，T
      π=3.14159265
      N=2**M
      DO 20 L=1，M
      LE=2**(M+1-L)
      LE2=LE/2
      U=(1.0,0.0)
      S= CMPLX（硫化羰（PI/float（LE2）），-SIN（PI/float（LE2）））
      DO 20 J=1，LE2
      DO 10 I=J，N，LE
      IP=I+LE2
      T=X (I)+X（IP）
      X（IP）=（X   (I)-X（IP））*
U 10  X(I)=T
20    U=U*S
      ND2=N/2
      NM1=N-1
      J=1
      DO 50 I=1,NM1
30    
      T=X(J)
      X（J）=X（I）
      X（I）=T
30    K=ND2
40    如果（K.GE.J）转到 50
      J=J-K

      转到 40

50    J=J+K 返回

      结束
```

表12-3
Fortran中的快速傅里叶变换
数据被传递至该子程序中
variables X( ) and M. The integer, M, is the
DFT长度的以2为底的对数，IF（I.GE.J）转至
i.e.，对于256点DFT，M= 8，对于
4096点DFT等。复数组，X（），
在进入时域数据
DFT。从该子程序返回后，X（）是
用频域数据覆盖。
注意：该子程序要求
输入和输出信号从X(1)开始
X（N），而非常规的X(0) 到K=K/2
X(N-1).

```
1000 'THE FAST FOURIER TRANSFORM
1010 'Upon entry, N% contains the number of points in the DFT, REX[ ] and
1020 'IMX[ ] contain the real and imaginary parts of the input.  Upon return,
1030 'REX[ ] and IMX[ ] contain the DFT output. All signals run from 0 to N%-1.
1040 '
1050 PI = 3.14159265                               'Set constants
1060 NM1% = N%-1
1070 ND2% = N%/2
1080 M% = CINT(LOG(N%)/LOG(2))
1090 J% = ND2%
1100 '
1110 FOR I% = 1 TO N%-2                            'Bit reversal sorting
1120   IF I% >= J% THEN GOTO 1190
1130   TR = REX[J%]
1140   TI = IMX[J%]
1150   REX[J%] = REX[I%]
1160   IMX[J%] = IMX[I%]
1170   REX[I%] = TR
1180   IMX[I%] = TI
1190   K% = ND2%
1200   IF K% > J% THEN GOTO 1240
1210   J% = J%-K%
1220   K% = K%/2
1230   GOTO 1200
1240   J% = J%+K%
1250 NEXT I%
1260 '
1270 FOR L% = 1 TO M%                              'Loop for each stage
1280   LE% = CINT(2^L%)
1290   LE2% = LE%/2
1300   UR = 1
1310   UI = 0
1320   SR =  COS(PI/LE2%)                          'Calculate sine & cosine values
1330   SI  = -SIN(PI/LE2%)
1340   FOR J% = 1 TO LE2%                          'Loop for each sub DFT
1350     JM1% = J%-1
1360     FOR I% = JM1% TO NM1% STEP LE%            'Loop for each butterfly
1370       IP% = I%+LE2%
1380       TR = REX[IP%]*UR - IMX[IP%]*UI          'Butterfly calculation
1390       TI = REX[IP%]*UI + IMX[IP%]*UR
1400       REX[IP%] = REX[I%]-TR
1410       IMX[IP%] = IMX[I%]-TI
1420       REX[I%]  = REX[I%]+TR
1430       IMX[I%]  = IMX[I%]+TI
1440     NEXT I%
1450     TR = UR
1460     UR = TR*SR - UI*SI
1470     UI = TR*SI + UI*SR
1480   NEXT J%
1490 NEXT L%
1500 '
1510 RETURN
```

TABLE 12-4
The Fast Fourier Transform in BASIC.

```
1000 “快速傅里叶变换”
1010 ‘输入时，N%存储DFT的点数，REX[]和1020′ IMX []分别存储输入的实部和虚部。返回时，1030 ‘REX[]和 IMX []存储DFT输出。所有信号范围从0到N%-1。1040′
1050 PI = 3.14159265                                   “设置常量”
1060 NM1% = N%-1
1070 ND2% = N%/2
1080 M% = CINT（LOG（N%）/LOG
(2)）1090 J% = ND2%
1100 '
1110 I% = 1 至 N%-2                              位反转排序
1120   IF I% >= J% THEN GOTO 1190
1130   TR = REX[J%]
1140   TI = IMX[J%] 1150   REX
[J%] = REX[I%] 1160   IMX[J%] =
IMX[I%] 1170   REX[I%] = TR
1180   IMX[I%] = TI
1190   K% = ND2%
1200 如果 K% > J% 则转至 1240 1210
J% = J% - K%
1220   K% = K%/2
1230 转到 1200
1240   J% = J%+K%
1250 下一个我
1260 '
1270 L% = 1 至 M%                                各阶段循环
1280   LE% = CINT(2^L%) 1290
LE2% = LE%/2 1300   UR = 1
1310   UI = 0
1320 SR = 硫化羰（PI/LE2%）                               计算正弦和余弦值
1330 SI = -SIN（PI/LE2%）
1340 J% = 1 至 LE2%                              对每个子DFT进行循环
1350     JM1% = J%-1
1360 FOR I% = JM1% TO NM1% STEP LE%  ‘循环遍历每个蝶形结构 1370
IP% = I%+LE2%
1380 TR = REX[IP%]*UR - IMX [IP%]*UI  ‘蝴蝶计算
1390       TI = REX[IP%]*UI + IMX[IP%]*UR
1400       REX[IP%]  =  REX[I%]-TR1410
IMX[IP%]  =  IMX[I%]-TI  1420       REX
[I%]  =  REX[I%]+TR1430       IMX[I%]  =
IMX[I%]+TI
1440 下一个我
1450     TR = UR
1460     UR = TR*SR - UI*SI 1470
UI = TR*SI + UI*SR
1480 下一个 J%
1490 下一个 L%
1500 '
1510 返回
```

表12-4
快速傅里叶变换的基本原理。

8 for a 256 point DFT, 12 for a 4096 point DFT, etc. The point is, the programmer who writes an FFT subroutine has many options for interfacing with the host program. Arrays that run from 1 to N, such as in the FORTRAN program, are especially aggravating. Most of the DSP literature (including this book) explains algorithms assuming the arrays run from sample 0 to $N-1$. For instance, if the arrays run from 1 to N, the symmetry in the frequency domain is around points 1 and $N/2+1$, rather than points 0 and $N/2$,

Using the complex DFT to calculate the real DFT has another interesting advantage. The complex DFT is more symmetrical between the time and frequency domains than the real DFT. That is, the **duality** is stronger. Among other things, this means that the Inverse DFT is nearly identical to the Forward DFT. In fact, the easiest way to calculate an *Inverse FFT* is to calculate a *Forward FFT*, and then adjust the data. Table 12-5 shows a subroutine for calculating the Inverse FFT in this manner.

Suppose you copy one of these FFT algorithms into your computer program and start it running. How do you know if it is operating properly? Two tricks are commonly used for debugging. First, start with some arbitrary time domain signal, such as from a random number generator, and run it through the FFT. Next, run the resultant frequency spectrum through the Inverse FFT and compare the result with the original signal. They should be *identical*, except round-off noise (a few parts-per-million for single precision).

The second test of proper operation is that the signals have the correct *symmetry*. When the imaginary part of the time domain signal is composed of all zeros (the normal case), the frequency domain of the complex DFT will be symmetrical around samples 0 and $N/2$, as previously described.

```
2000 'INVERSE FAST FOURIER TRANSFORM SUBROUTINE
2010 'Upon entry, N% contains the number of points in the IDFT, REX[ ] and
2020 'IMX[ ] contain the real and imaginary parts of the complex frequency domain.
2030 'Upon return, REX[ ] and IMX[ ] contain the complex time domain.
2040 'All signals run from 0 to N%-1.
2050 '
2060 FOR K% = 0 TO N%-1                    'Change the sign of IMX[ ]
2070   IMX[K%] = -IMX[K%]
2080 NEXT K%
2090 '
2100 GOSUB 1000                            'Calculate forward FFT  (Table 12-3)
2110 '
2120 FOR I% = 0 TO N%-1                    'Divide the time domain by N% and
2130   REX[I%] =  REX[I%]/N%               'change the sign of IMX[ ]
2140   IMX[I%] = -IMX[I%]/N%
2150 NEXT I%
2160 '
2170 RETURN
```

TABLE 12-5

8 用于 256 点 DFT，12 用于 4096 点 DFT，等等。关键在于，编写 FFT 子程序的程序员在与宿主程序交互时有许多选择。从 1 到*N*的数组（如 Fortran 程序中）尤其令人头疼。大多数 DSP 文献（包括本书）在解释算法时都假设数组从采样点 0 运行到*N*-1。例如，如果数组从 1 运行到*N*，频域中的对称性会集中在点 1 和*N*/2+1附近，而非点
0和*N*/2，

运用复数DFT计算实数DFT还具有另一个显著优势。相较于实数DFT，复数DFT在时域与频域间展现出更强的对称性，即**对偶性**更为显著。这意味着逆DFT与正DFT几乎完全等效。实际上，计算*逆 FFT*最简便的方法就是先计算*正 FFT*，再对数据进行调整。表12-5展示了采用此方法计算逆 FFT 的子程序示例。

假设你将这些 FFT 算法之一复制到计算机程序中并启动运行。如何判断其是否正常工作？调试时通常采用两种技巧：首先，从任意时域信号（例如随机数生成器输出）开始，将其输入 FFT ；接着，将生成的频谱信号输入逆 FFT ，与原始信号对比。除舍入噪声（单精度情况下为百万分之几）外，两者应*完全一致* 。

第二个正确操作的测试是信号具有正确的*对称性* 。当时域信号的虚部由所有零组成时（正常情况），复数DFT的频域将围绕样本0和*N*/2对称，如前所述。

```
2000 的逆快速傅里叶变换子程序
2010 'Uponentry' 表示 IDFT 中的点数，REX[]和2020 ' IMX []分别表示复频域中的实部和虚
部。
2030年回归后，REX[ ]和 IMX [ ]包含复数时间域。
2040年所有信号均从0运行至N%-1。
2050 '
2060 当K%等于0至N-1时                              改变IMX[ ]的符号
2070   IMX[K%] = -IMX[K%]
2080 下一个K%
2090 '
2100 GOSUB 1000                                 计算远期 FFT（表12-3）
2110 '
2120 I% = 0 至 N%-1                             将时间域按N%进行划分
2130   REX[I%]  =   REX[I%]/N%IMX               改变IMX[ ]的符号
2140   [I%] = -IMX[I%]/N%
2150 下一次输入
2160 '
2170 返回
```

表12-5

Likewise, when this correct symmetry is present in the frequency domain, the Inverse DFT will produce a time domain that has an imaginary part composes of all zeros (plus round-off noise). These debugging techniques are essential for using the FFT; become familiar with them.

Speed and Precision Comparisons

When the DFT is calculated by correlation (as in Table 12-2), the program uses two nested loops, each running through N points. This means that the total number of operations is proportional to *N times N*. The time to complete the program is thus given by:

EQUATION 12-1
DFT execution time. The time required to calculate a DFT by correlation is proportional to the length of the DFT squared.

$$ExecutionTime = k_{DFT} N^2$$

where N is the number of points in the DFT and k_{DFT} is a constant of proportionality. If the sine and cosine values are calculated *within* the nested loops, k_{DFT} is equal to about 25 microseconds on a Pentium at 100 MHz. If you *precalculate* the sine and cosine values and store them in a look-up-table, k_{DFT} drops to about 7 microseconds. For example, a 1024 point DFT will require about 25 seconds, or nearly 25 milliseconds per point. That's slow!

Using this same strategy we can derive the execution time for the FFT. The time required for the bit reversal is negligible. In each of the $Log_2 N$ stages there are $N/2$ butterfly computations. This means the execution time for the program is approximated by:

EQUATION 12-2
FFT execution time. The time required to calculate a DFT using the FFT is proportional to N multiplied by the logarithm of N.

$$ExecutionTime = k_{FFT} N \log_2 N$$

The value of k_{FFT} is about 10 microseconds on a 100 MHz Pentium system. A 1024 point FFT requires about 70 milliseconds to execute, or 70 microseconds per point. This is more than 300 times faster than the DFT calculated by correlation!

Not only is $NLog_2 N$ less than N^2, it increases much more slowly as N becomes larger. For example, a 32 point FFT is about *ten* times faster than the correlation method. However, a 4096 point FFT is *one-thousand* times faster. For small values of N (say, 32 to 128), the FFT is important. For large values of N (1024 and above), the FFT is absolutely critical. Figure 12-8 compares the execution times of the two algorithms in a graphical form.

同样地，当这种正确的对称性存在于频域时，逆DFT将生成一个虚部全为零（加上舍入噪声）的时域。这些调试技术对于使用 FFT 至关重要，务必熟练掌握。

速度与精度比较

当通过相关性计算DFT时（如表12-2所示），程序使用两个嵌套循环，每个循环运行通过*N*个点。这意味着总操作数与*N乘以N*成正比。因此，程序完成所需的时间由以下公式给出：

方程12-1
DFT执行时间。通过相关性计算DFT所需的时间与DFT长度的平方成正比。

$$ExecutionTime = k_{DFT} \; N^2$$

其中*N*是DFT中的点数，k_{DFT}是比例常数。如果在嵌套循环*内*计算正弦和余弦值，k_{DFT}在100 MHz的奔腾处理器上约为25微秒。若将正弦和余弦值*预计算*并存储在查找表中，k_{DFT}可降至约7微秒。例如，1024点DFT需要约25秒，即每个点耗时近25毫秒。这太慢了！

采用相同策略，我们可以推导出 FFT 的执行时间。位反转所需时间可忽略不计。在每个$Log_2\,N$阶段中，存在*N*/2个蝶形运算。这意味着该程序的执行时间可近似表示为：

方程12-2
FFT 执行时间。使用 FFT 计算DFT所需的时间与*N*乘以N的对数成正比。

$$ExecutionTime = k_{FFT}\, N \; \log_2 N$$

在100MHz奔腾系统上，of k_{FFT}的计算时间约为10微秒。执行1024点 FFT 需要约70毫秒，即每个点耗时70微秒。这比通过相关性计算的DFT速度快300多倍！

不仅$N Log_2\, N$小于N^2，而且随着*N*的增大，其增长速度会显著放缓。例如，32点 FFT 比相关方法快约*十*倍。然而，4096点 FFT 则快*一千*倍。对于*N*的小值（例如32到128）， FFT 具有重要影响；而对于*N*的大值（1024及以上）， FFT 则至关重要。图12-8以图形化形式比较了两种算法的执行时间。

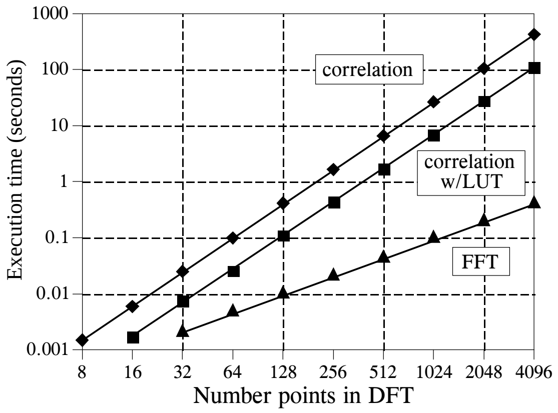

FIGURE 12-8
Execution times for calculating the DFT. The *correlation* method refers to the algorithm described in Table 12-2. This method can be made faster by precalculating the sine and cosine values and storing them in a look-up table (LUT). The FFT (Table 12-3) is the fastest algorithm when the DFT is greater than 16 points long. The times shown are for a Pentium processor at 100 MHz.

The FFT has another advantage besides raw speed. The FFT is calculated more *precisely* because the fewer number of calculations results in less round-off error. This can be demonstrated by taking the FFT of an arbitrary signal, and then running the frequency spectrum through an Inverse FFT. This reconstructs the original time domain signal, *except* for the addition of round-off noise from the calculations. A single number characterizing this noise can be obtained by calculating the standard deviation of the difference between the two signals. For comparison, this same procedure can be repeated using a DFT calculated by correlation, and a corresponding Inverse DFT. How does the round-off noise of the FFT compare to the DFT by correlation? See for yourself in Fig. 12-9.

Further Speed Increases

There are several techniques for making the FFT even faster; however, the improvements are only about 20-40%. In one of these methods, the time

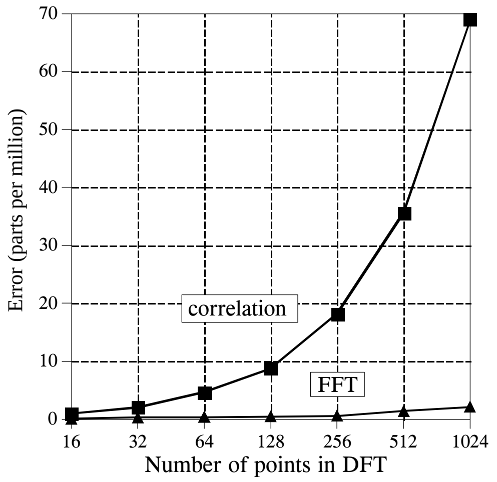

FIGURE 12-9
DFT precision. Since the FFT calculates the DFT *faster* than the correlation method, it also calculates it with less round-off error.

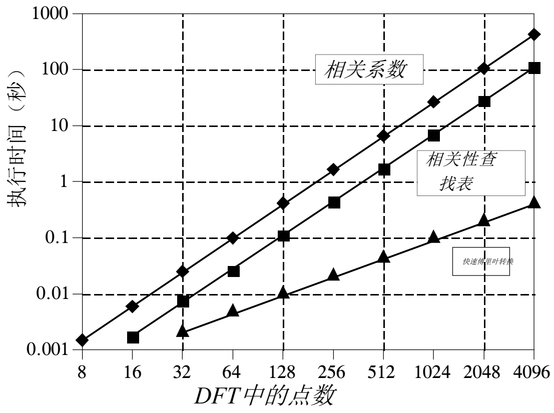

图12-8
计算DFT的执行时间。*相关性*方法指表12-2中描述的算法。通过预计算正弦和余弦值并存储在查找表（LUT）中，该方法可实现加速。当DFT长度超过16点时，FFT（表12-3）是最快算法。所示时间基于100MHz奔腾处理器的测试结果。

除了原始速度优势外， FFT 还具备另一项显著特性。由于计算次数减少导致舍入误差降低，因此 FFT 的计算结果更加*精确*。具体验证方法是：对任意信号进行 FFT 处理后，再通过逆 FFT 处理其频谱。这种处理方式能重建原始时域信号，*但会*产生计算过程中的舍入噪声。通过计算两种信号差值的标准差，即可获得表征该噪声的单一数值。作为对比，可采用相关性计算得到的DFT及其对应的逆DFT重复相同流程。 FFT 的舍入噪声与相关性DFT相比有何差异？请参见图12-9进行对比验证。

进一步提高速度

有几种技术可使 FFT 进一步加快，但改进幅度仅约20%至40%。其中一种方法中，时间

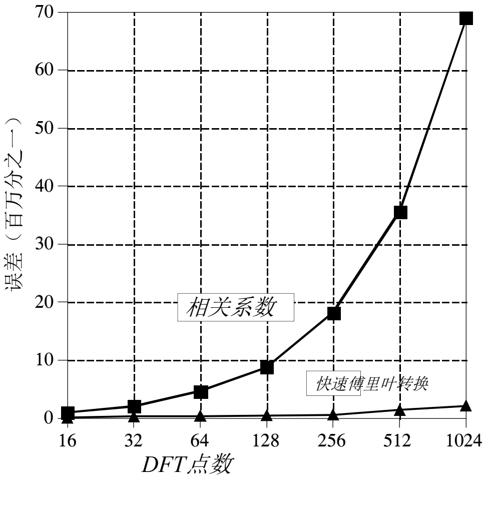

图12-9
DFT精度。由于 FFT 计算DFT的速度比相关方法更快，因此其计算过程中的舍入误差也更小。

domain decomposition is stopped two stages early, when each signals is composed of only four points. Instead of calculating the last two stages, highly optimized code is used to jump directly into the frequency domain, using the simplicity of four point sine and cosine waves.

Another popular algorithm eliminates the wasted calculations associated with the imaginary part of the time domain being zero, and the frequency spectrum being symmetrical. In other words, the FFT is modified to calculate the *real DFT*, instead of the *complex DFT*. These algorithms are called the **real FFT** and the **real Inverse FFT** (or similar names). Expect them to be about 30% faster than the conventional FFT routines. Tables 12-6 and 12-7 show programs for these algorithms.

There are two small disadvantages in using the *real FFT*. First, the code is about twice as long. While your computer doesn't care, you must take the time to convert someone else's program to run on your computer. Second, debugging these programs is slightly harder because you cannot use symmetry as a check for proper operation. These algorithms *force* the imaginary part of the time domain to be zero, and the frequency domain to have left-right symmetry. For debugging, check that these programs produce the same output as the conventional FFT algorithms.

Figures 12-10 and 12-11 illustrate how the real FFT works. In Fig. 12-10, (a) and (b) show a time domain signal that consists of a pulse in the real part, and all zeros in the imaginary part. Figures (c) and (d) show the corresponding frequency spectrum. As previously described, the frequency domain's real part has an *even* symmetry around sample 0 and sample $N/2$, while the imaginary part has an *odd* symmetry around these same points.

```
4000 'INVERSE FFT FOR REAL SIGNALS
4010 'Upon entry, N% contains the number of points in the IDFT, REX[ ] and
4020 'IMX[ ] contain the real and imaginary parts of the frequency domain running from
4030 'index 0 to N%/2.  The remaining samples in REX[ ] and IMX[ ] are ignored.
4040 'Upon return, REX[ ] contains the real time domain, IMX[ ] contains zeros.
4050 '
4060 '
4070 FOR K% = (N%/2+1) TO (N%-1)                'Make frequency domain symmetrical
4080   REX[K%] =  REX[N%-K%]                    '(as in Table 12-1)
4090   IMX[K%] = -IMX[N%-K%]
4100 NEXT K%
4110 '
4120 FOR K% = 0 TO N%-1                         'Add real and imaginary parts together
4130   REX[K%] =  REX[K%]+IMX[K%]
4140 NEXT K%
4150 '
4160 GOSUB 3000                                 'Calculate forward real DFT (TABLE 12-6)
4170 '
4180 FOR I% = 0 TO N%-1                         'Add real and imaginary parts together
4190   REX[I%] = (REX[I%]+IMX[I%])/N%           'and divide the time domain by N%
4200   IMX[I%] = 0
4210 NEXT I%
4220 '
4230 RETURN
```

TABLE 12-6

当每个信号仅由四个点组成时，域分解提前两个阶段停止。不再计算最后两个阶段，而是采用高度优化的代码直接跳入频域，利用四点正弦和余弦波的简洁性。

另一种广受欢迎的算法通过消除与虚部为零和频谱对称性相关的冗余计算，实现了效率提升。具体来说，该 FFT 经过改进后采用*实数DFT*进行计算，而非传统的*复数DFT*。这类算法被称为**实数 FFT**和**实数逆 FFT**（或类似名称）。预计其运算速度将比传统 FFT 程序快约30%。表12-6和表12-7展示了这些算法的程序实现。

使用*真实FFT*存在两个小缺点。首先，代码长度大约是常规 FFT 的两倍。虽然计算机本身不会在意，但你必须花时间将别人的程序转换为能在你的计算机上运行。其次，调试这些程序会稍微困难些，因为你无法利用对称性来验证程序的正确运行。这些算法会*强制*时间域的虚部为零，并要求频域具有左右对称性。在调试时，需确保这些程序产生的输出与传统 FFT 算法完全一致。

图12-10和12-11展示了实 FFT 的工作原理。图12-10的(a)和(b)展示了时域信号，该信号实部为脉冲，虚部全为零。图(c)和(d)则显示了对应的频谱。如前所述，频域实部在采样点0和采样点$N/2$处具有*偶*对称性，而虚部在这些点处则呈现*奇*对称性。

```
4000 ‘真实信号的逆 FFT
4010 ‘进入时，N%包含 IDFT 中的点数，REX[ ]和
4020 ‘ IMX [ ]包含从4030’ 索引0到N%/2的频域实部和虚部。REX[ ]和 IMX [ ]中的其余样本被
忽略。
4040 ‘返回时，REX[]包含实时域， IMX []包含零值。
4050 '
4060 '
4070 K% =（N%/2+1）至（N%-1）                  使频域对称
4080  REX[K%] =  REX[N%-K%]        （如表12-1所示）4090 IMX [K%] = - IMX
[N%-K%]
4100 下一个 K%
4110 '
4120 K% = 0 至 N-1                         将实部与虚部相加
4130  REX[K%] =  REX[K%]+IMX[K%]
4140 下一个 K%
4150 '
4160 GOSUB 3000                           ‘计算正向实部密度泛函理论（表12-6）4170’
4180 I% = 0 至 N%-1                        将实部与虚部相加
4190 REX[I%] =（REX[I%]+IMX[I%]）/N%             并将时域除以N%
4200  IMX[I%] = 0
4210 下一个我
4220 '
4230 返回
```

表12-6

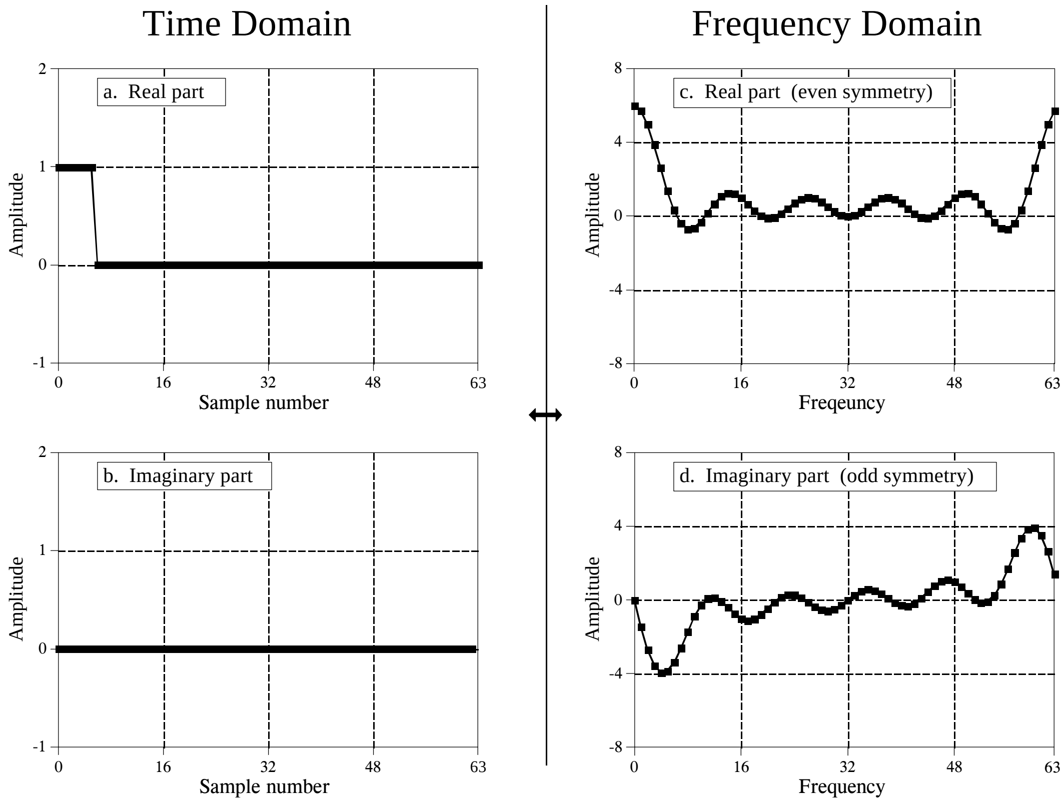

FIGURE 12-10
Real part symmetry of the DFT.

Now consider Fig. 12-11, where the pulse is in the imaginary part of the time domain, and the real part is all zeros. The symmetry in the frequency domain is *reversed*; the real part is odd, while the imaginary part is even. This situation will be discussed in Chapter 29. For now, take it for granted that this is how the complex DFT behaves.

What if there is a signal in *both parts* of the time domain? By additivity, the frequency domain will be the *sum* of the two frequency spectra. Now the key element: a frequency spectrum composed of these two types of symmetry can be perfectly separated into the two component signals. This is achieved by the *even/odd decomposition* discussed in Chapter 6. In other words, two real DFT's can be calculated for the price of single FFT. One of the signals is placed in the real part of the time domain, and the other signal is placed in the imaginary part. After calculating the complex DFT (via the FFT, of course), the spectra are separated using the even/odd decomposition. When two or more signals need to be passed through the FFT, this technique reduces the execution time by about 40%. The improvement isn't a full factor of two because of the calculation time required for the even/odd decomposition. This is a relatively simple technique with few pitfalls, nothing like writing an FFT routine from scratch.

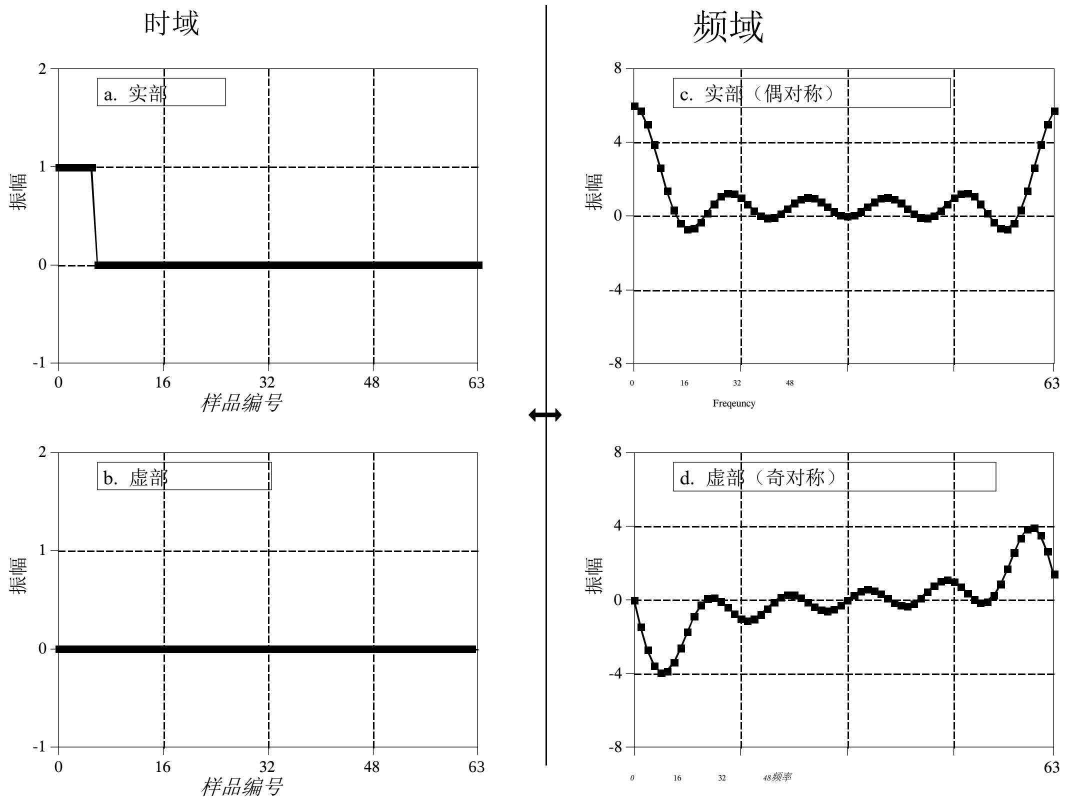

图12-10
DFT的实部对称性。

现在考虑图12-11，其中脉冲在时域的虚部，实部为零。频域的对称性*反转*；实部为奇，虚部为偶。这种情况将在第29章讨论。现在，我们默认这就是复DFT的行为方式。

如果时域*的两个部分*都存在信号呢？根据可加性原理，频域将呈现为两个频谱的*之和*。此时关键要素在于：由这两种对称性构成的频谱可以被完美分解为两个独立信号。这一过程通过第六章讨论的*偶/奇分解法*实现。换言之，只需一次 FFT 运算即可完成两次实数DFT计算——一个信号被分配到时域实部，另一个信号则分配到虚部。通过复数DFT计算（当然借助 FFT 实现）后，再利用偶/奇分解法进行频谱分离。当需要处理两个或更多信号时，该技术可将执行时间缩短约40%。不过由于偶/奇分解的计算开销，实际提升幅度并非完全翻倍。这种技术相对简单且不易出错，远比从头编写 FFT 程序要省事得多。

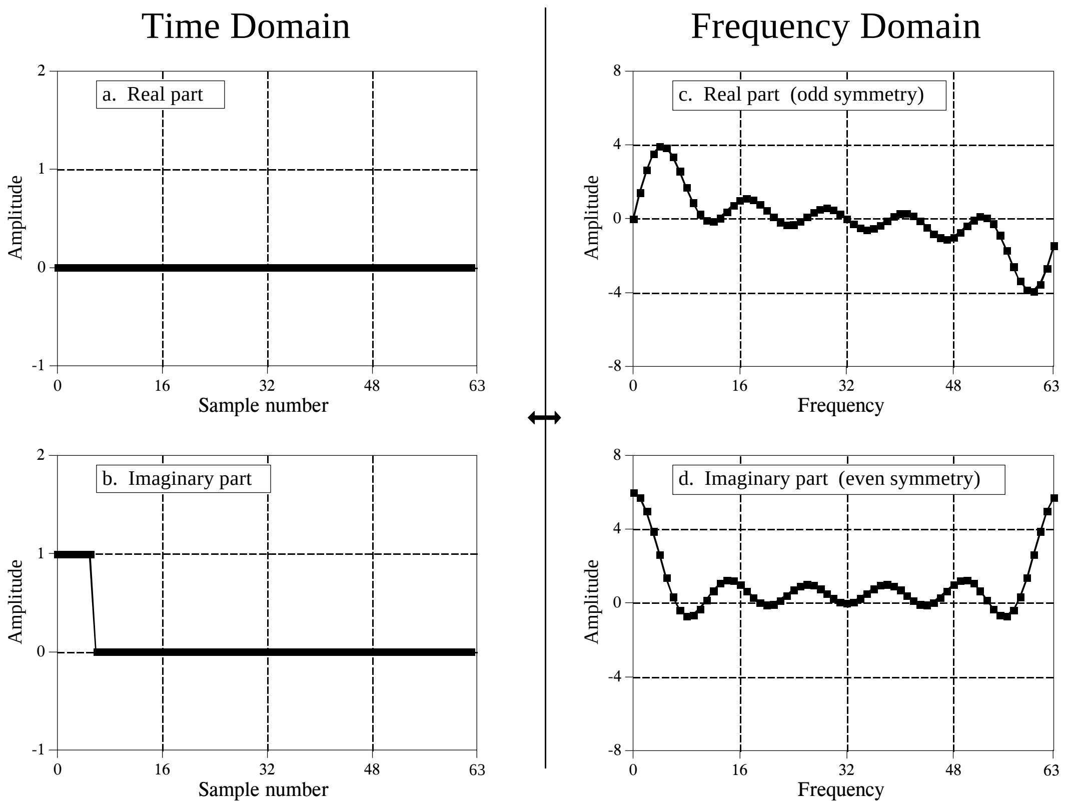

FIGURE 12-11
Imaginary part symmetry of the DFT.

The next step is to modify the algorithm to calculate a *single* DFT faster. It's ugly, but here is how it is done. The input signal is broken in half by using an interlaced decomposition. The $N/2$ even points are placed into the real part of the time domain signal, while the $N/2$ odd points go into the imaginary part. An $N/2$ point FFT is then calculated, requiring about one-half the time as an N point FFT. The resulting frequency domain is then separated by the even/odd decomposition, resulting in the frequency spectra of the two interlaced time domain signals. These two frequency spectra are then combined into a single spectrum, just as in the last synthesis stage of the FFT.

To close this chapter, consider that the *FFT* is to *Digital Signal Processing* what the *transistor* is to *electronics*. It is a foundation of the technology; everyone in the field knows its characteristics and how to use it. However, only a small number of specialists really understand the details of the internal workings.

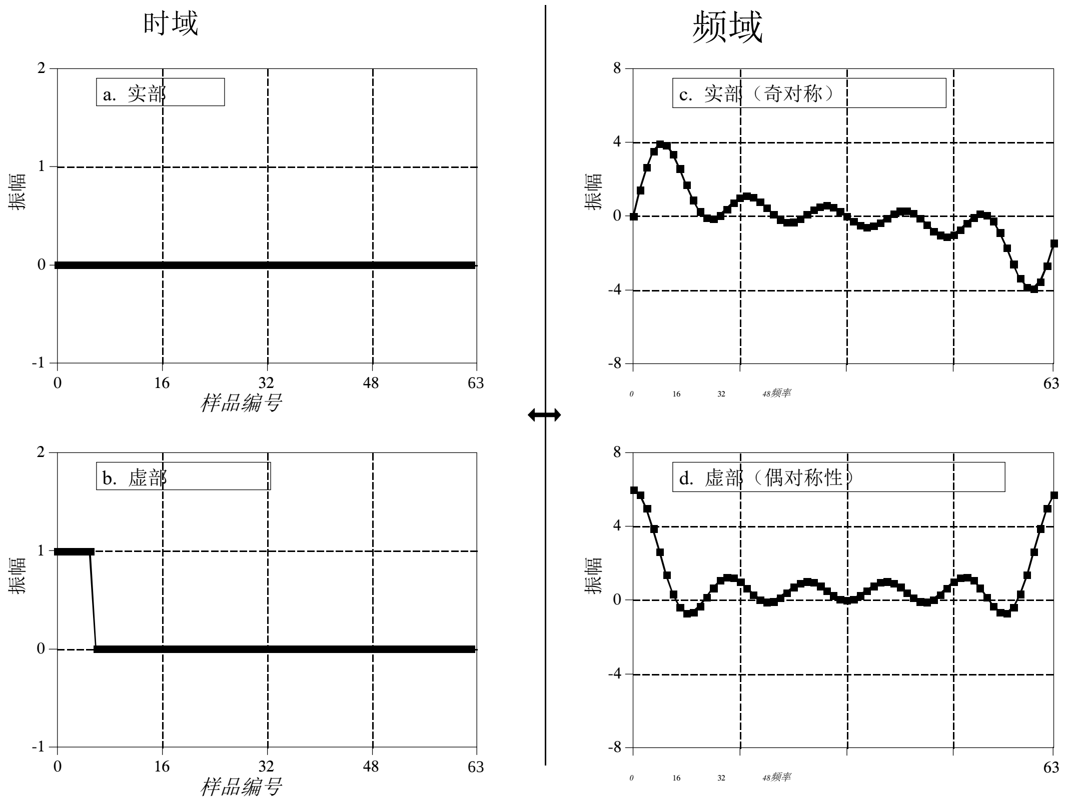

图12-11
DFT的虚部对称性。

下一步是修改算法以更快地计算*单个*DFT。虽然过程复杂，但具体操作如下：通过交错分解将输入信号一分为二，将$N/2$个偶数点放入时域信号的实部，而$N/2$个奇数点放入虚部。随后计算一个$N/2$点 FFT，所需时间约为N点 FFT 的一半。通过奇偶分解将得到的频域信号分离，得到两个交错时域信号的频谱。最后将这两个频谱合并为单一频谱，这与 FFT 的最后合成阶段相同。

在结束本章时，考虑一下*FFT*对于*数字信号处理*而言，就如同*晶体管*对于*电子学*一样。它是技术的基础；该领域的每个人都了解它的特性以及如何使用它。然而，只有少数专家真正理解其内部运作的细节。

```
3000 'FFT FOR REAL SIGNALS
3010 'Upon entry, N% contains the number of points in the DFT, REX[ ] contains
3020 'the real input signal, while values in IMX[ ] are ignored.  Upon return,
3030 'REX[ ] and IMX[ ] contain the DFT output. All signals run from 0 to N%-1.
3040 '
3050 NH% = N%/2-1                              'Separate even and odd points
3060 FOR I% = 0 TO NH%
3070   REX(I%) = REX(2*I%)
3080   IMX(I%) = REX(2*I%+1)
3090 NEXT I%
3100 '
3110 N% = N%/2                                 'Calculate N%/2 point FFT
3120 GOSUB 1000                                '(GOSUB 1000 is the FFT in Table 12-3)
3130 N% = N%*2
3140 '
3150 NM1% = N%-1                               'Even/odd frequency domain decomposition
3160 ND2% = N%/2
3170 N4% = N%/4-1
3180 FOR I% = 1 TO N4%
3190   IM%  = ND2%-I%
3200   IP2% = I%+ND2%
3210   IPM% = IM%+ND2%
3220   REX(IP2%) =  (IMX(I%) + IMX(IM%))/2
3230   REX(IPM%) =  REX(IP2%)
3240   IMX(IP2%) = -(REX(I%) - REX(IM%))/2
3250   IMX(IPM%) = -IMX(IP2%)
3260   REX(I%)   =  (REX(I%) + REX(IM%))/2
3270   REX(IM%)  =  REX(I%)
3280   IMX(I%)   =  (IMX(I%) - IMX(IM%))/2
3290   IMX(IM%)  = -IMX(I%)
3300 NEXT I%
3310 REX(N%*3/4) = IMX(N%/4)
3320 REX(ND2%) = IMX(0)
3330 IMX(N%*3/4) = 0
3340 IMX(ND2%) = 0
3350 IMX(N%/4) = 0
3360 IMX(0) = 0
3370 '
3380 PI = 3.14159265                                   'Complete the last FFT stage
3390 L% = CINT(LOG(N%)/LOG(2))
3400 LE% = CINT(2^L%)
3410 LE2% = LE%/2
3420 UR = 1
3430 UI = 0
3440 SR =  COS(PI/LE2%)
3450 SI = -SIN(PI/LE2%)
3460 FOR J% = 1 TO LE2%
3470   JM1% = J%-1
3480   FOR I% = JM1% TO NM1% STEP LE%
3490     IP% = I%+LE2%
3500     TR = REX[IP%]*UR - IMX[IP%]*UI
3510     TI = REX[IP%]*UI + IMX[IP%]*UR
3520     REX[IP%] = REX[I%]-TR
3530     IMX[IP%] = IMX[I%]-TI
3540     REX[I%]  = REX[I%]+TR
3550     IMX[I%]  = IMX[I%]+TI
3560   NEXT I%
3570   TR = UR
3580   UR = TR*SR - UI*SI
3590   UI = TR*SI + UI*SR
3600 NEXT J%
3610 RETURN
```

TABLE 12-7

```
3000 实信号FFT
3010 ‘输入时，N%存储DFT的点数，REX[ ]存储真实输入信号，而 IMX [ ]中
的值被忽略。返回时，3030’ REX[ ]和 IMX [ ]存储DFT输出。所有信号范围
从0到N%-1。3040 ‘
3050 NH% = N%/2-1                                    ‘分离偶数点与奇数点’
3060 I% = 0 至 NH%
3070 REX（I%）= REX（2*I%）
3080 IMX（I%）= REX（2*I%+1）
3090 下一个我
3100 '
3110 N% = N%/2                                     计算N%/2点 FFT
3120 GOSUB 1000                    （GOSUB 1000为表12-3中的 FFT）3130 N% = N%*2
3140 '
3150 NM1% = N%-1                                   奇偶频率域分解
3160 ND2% = N%/2
3170 N4% = N%/4-1
3180 I% = 1 至 N4%
3190   IM%  = ND2%-I%
3200   IP2% = I%+ND2%
3210   IPM% = IM%+ND2%
3220 REX（IP2%）=（IMX（I%）+ IMX（IM%））/2
3230 REX（IPM%）= REX（IP2%）
3240 IMX（IP2%）= -（REX（I%）- REX（IM%））/2
3250 IMX（IPM%）= - IMX（<|term_2|>%）
3260 REX（I%）=（REX（I%）+ REX（IM%））/2
3270 REX（IM%）= REX（I%）
3280 IMX（I%）=（IMX（I%）- IMX（IM%））/2
3290 IMX（IM%）= -IMX（I%）
3300 下一个我
3310 REX(N%*3/4) = IMX(N%/4)
3320 REX(ND2%) = IMX(0)
3330 IMX(N%*3/4) = 0
3340 IMX(ND2%) = 0
3350 IMX(N%/4) = 0
3360 IMX(0) = 0
3370 '
3380 PI = 3.14159265                                        完成最后的 FFT 阶段
3390 L% = CINT（LOG（N%）/LOG (2)）
3400 LE% = CINT(2^L%)
3410 LE2% = LE%/2
3420 UR = 1
3430 UI = 0
3440 SR = 硫化羰（PI/LE2%）
3450 SI = -SIN（PI/LE2%）
3460 J% = 1 至 LE2%
3470   JM1% = J%-1
3480 I% = JM1% 至 NM1% 步长 LE%
3490     IP% = I%+LE2%
3500     TR = REX[IP%]*UR - IMX[IP%]*UI
3510     TI = REX[IP%]*UI + IMX[IP%]*UR
3520     REX[IP%] = REX[I%]-TR
3530     IMX[IP%] = IMX[I%]-TI
3540     REX[I%]  = REX[I%]+TR
3550     IMX[I%]  = IMX[I%]+TI
3560 下一个我
3570   TR = UR
3580   UR = TR*SR - UI*SI
3590   UI = TR*SI + UI*SR
3600 下一个 J%
3610 返回
```

表12-7