一天掌握 DFT

作者：　Stephan M.Bernsee
翻译：　韩星

前言

如果了解信号处理，你一定会认为这个标题十分夸张。我也支持你的看法。我们当然不可能脱离实践和对数学的深入研究，一天之内就学会傅立叶变换的所有知识。然而，这本在线教程会通过一些非传统的方法向你解释什么是傅立叶变换、为什么要做傅立叶变换，使你的学习变得更加简单。重要的是：你将完全摆脱复杂的数学公式而掌握傅立叶变换的基本原理。本文将尝试以声音信号处理为例，用六个步骤来介绍傅立叶变换。

第一步：一些简单基础知识

要看懂本段，以下 4 点必不可少：加法、乘法、什么是 sine 和 cosine 曲线、什么是正弦曲线。显然前两点这里将省略，我只对最后一点稍加解释。你或许还记得在学校里学过三角函数，就是与角度一起用来求边长或者知道边长来求角度的公式。我们不需要完全回忆起它们，只需要知道 sine 和 cosine 曲线是什么样子就可以了。这个非常简单：如图 1 所示，它们看起来像是由高峰和低谷组成，从观测者左边到右边无穷延伸的波浪。

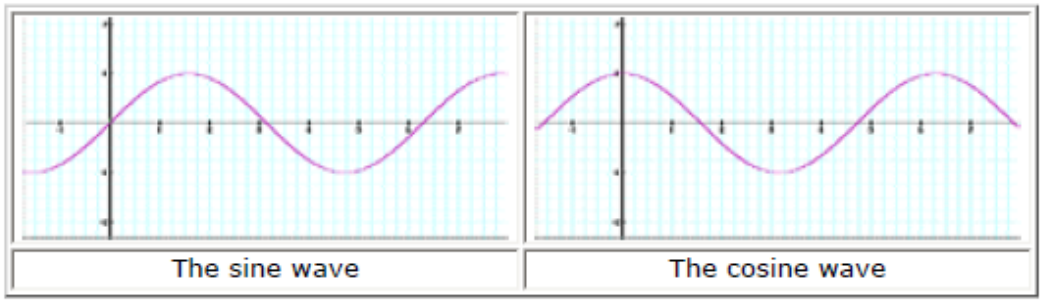

图 1 sine 和 cosine 曲线

如你所见，这两个信号都是周期的，这意味着在一段特定的时间后波形是重复的。Sine 和 cosine 波形看起来很像，只不过 sine 波形开始于零点而 cosine 波形开始于它的最大值。在实践中，我们怎么知道在给定时间内观测的信号是从零开始还是从最大值开始呢？这是一个好问题：我们不知道。现实中我们无法辨别 sine 和 cosine 波形，因此我们将这种类型的波形统称为正弦曲线。正弦曲线的重要特性就是频率，它告诉我们在给定时间会出现多少波

峰和波谷。频率高意味着给定时间内出现的波峰和波谷多，反之亦然。

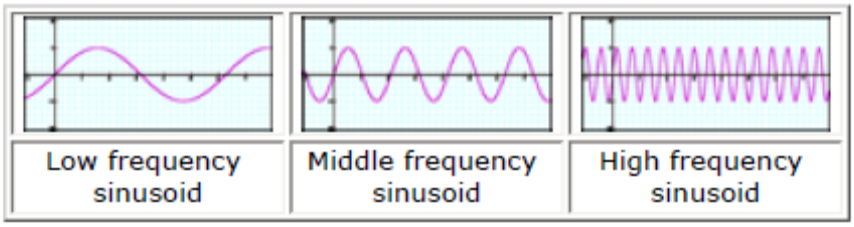

图 2 不同频率的 sine 波形

第二步：掌握傅立叶理论基础

Jean-Baptiste Joseph Fourier 是一个父母“又爱又恨”的孩子，因为他在 14 时就开始向他们提一些复杂的数学问题。虽然傅里叶一生中有很多富有成效的研究成果，但其中最重要的还是他发现了金属的热传导性。他推导出热在特定介质中的传导方程，并用有限次三角函数解出了该方程。与本文主题相关的是，傅里叶发现了一个通用规律——无论多复杂的信号，都可以用一系列不同的正弦曲线的和来表示。

如下图所示：

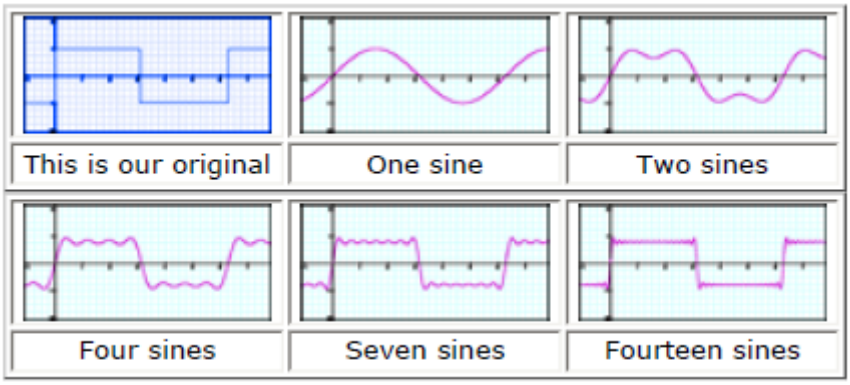

图 3 用 sine 曲线重构给定信号

上图利用 sine 信号按一定规律求和来近似原始信号。我们简要地介绍一下这里所说的“一定规律”。正如你看到的，sine 信号越多，则结果越逼近原始信号。理想情况下信号是连续的，我们可以通过无穷小的间隔采样来测量原始信号，该精度只受限于测量设备，此时我们需要利用无穷多的 sine 曲线来完美地重建原始信号。幸运的是，作为数字信号处理者的我们并不是生活在理想世界里。取而代之的是，我们处理的“现实世界”里的信号都是以有限间隔测量的有限精度的信号。因此，我们不需要无穷多的 sine 信号，而只需要“足够多的”。何谓“足够多的”，我们将在下面介绍。此时，重要的是你要能够理解任何在你计算

机中的信号都是由一系列简单 sine 信号按照一定规律组合构成的。

第三步：多少才是“足够多的”

正如我们上面讨论的一样，一个形状复杂的信号可以由一系列简单的 sine 曲线构成。我们也许会问，多少 sine 曲线才能构成我们计算机中给定的信号呢？好吧，我们以 sine 曲线为例，它能帮助我们了解信号是如何构成的。大多数情况下，我们处理的信号具有复杂的结构，所以我们不能清楚地了解它们包含多少组成分量。如果我们不知道原始信号由多少 sine 曲线构成，我们就无法确定所需 sine 波形数量的上限。至此，我们必须回答多少才是“足够多的”这个问题。让我们尝试凭直觉来探索该问题的答案：假设我们有 1000 个信号采样点。具有最短周期的 sine 曲线是相邻波峰和波谷出现相邻采用点的曲线。所以，1000 个采样点采样到的最高频率 sine 曲线包含 500 个波峰和 500 个波谷，每个采样点都在波峰或波谷处。下图中的黑点表示采样点，sine 曲线的最高频率如下图所示：

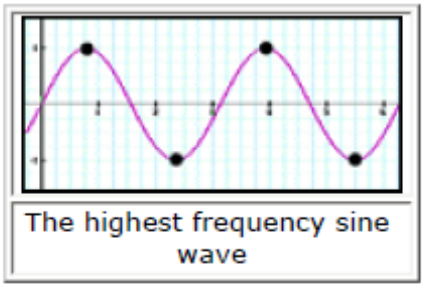

图 4 高频 sine 曲线

现在，我们来看这个 sine 曲线的最低频率是多少。如果只有一个采样点，我们能否判断这个点是峰值或是谷值呢？答案是不能，因为有许多周期不同的 sine 曲线都可以经过这个点。

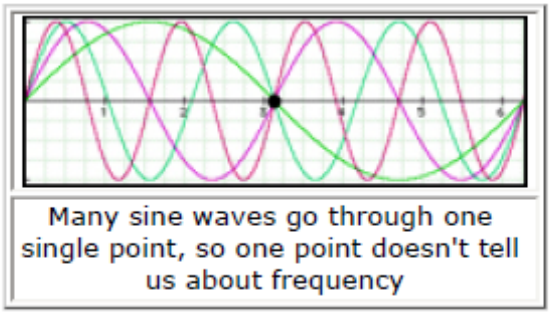

图 5 穿过零点的一组 sine 曲线

所以，一个采样点不足以反映频率信息。如果我们有两个采样点，是不是只有最低频率

的 sine 曲线经过这两个点？这种情况就简单多了，只有一条较低频率的 sine 曲线穿过这两点。如下图所示：

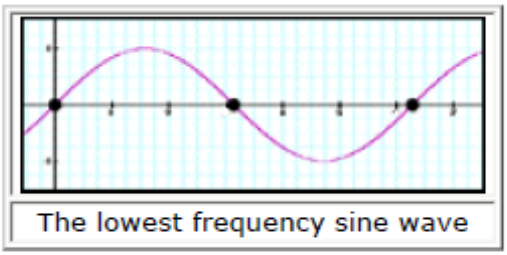

图 6 低频 sine 曲线

假想上图中最左边的两个采样点是两颗钉子，中间曲线是一根绳子。绳子的最低频率正如图中曲线所示。如果上图中的两个采样点变为 1001*个，两颗钉子是第一个和最后一个采样点，即采样点 1 和 1001*，这时绳长变为原来的 1000 倍。根据我们从乐器上得来的经验，绳子越长，频率越低。由于我们将钉子之间的距离拉大从而得到了更低的频率。在采样间隔不变得情况下，如果我们有 2001**个采样点，钉子变为采样点 1 和 2001**。事实上，此时频率降低到了原来频率的一半，因为我们的钉子之间的距离是原来 1001*个采样点时距离的两倍。因此，如果我们有更多的采样点，我们就可以得到更低的频率，因为“钉子”之间得距离变得更远。这点对理解下面的内容非常重要。

图 6 还告诉我们，在两颗“钉子”之后，曲线开始以负斜率重复。这意味着相邻两点恰好包含了半个 sine 曲线，换句话说就是包含一个波峰或一个波谷，或 1 / 2 周期。

综上所述：sine 曲线的最高频率出现在相邻采样点恰好在曲线的相邻波峰和波谷时（图 4），而最低频率出现在采样点恰好在波形半周期的边界时（图 6）。也就是说当最高频率保持不变时，即最小采样间隔不变时，采样点越多则 sine 曲线频率越低。这就导致我们需要更多的 sine 曲线去表示一个更长的未知信号，因为我们是从最低频率 sine 曲线开始重构信号的。

无论如何，至此我们仍然不知道需要多少 sine 曲线来构建原始信号。然而，当我们知道 sine 曲线的最高频率和最低频率，我们就可以计算出有多少种频率的 sine 曲线介于两者之间。因为我们已经用两颗钉子说明了如何产生最低频率，我们仍然用这两颗钉子说明其他频率如何产生。假象 sine 曲线是吉他上固定在两点之间的琴弦，它们只能在两点之间摆动（除非琴弦断掉），如图 7 所示。由图可知：最低频率（一次谐波）为在两点之间包含 1/2 个周期，二次谐波为 1 个周期，三次谐波为 3/2 个周期……1000 次谐波为 500 个周期。

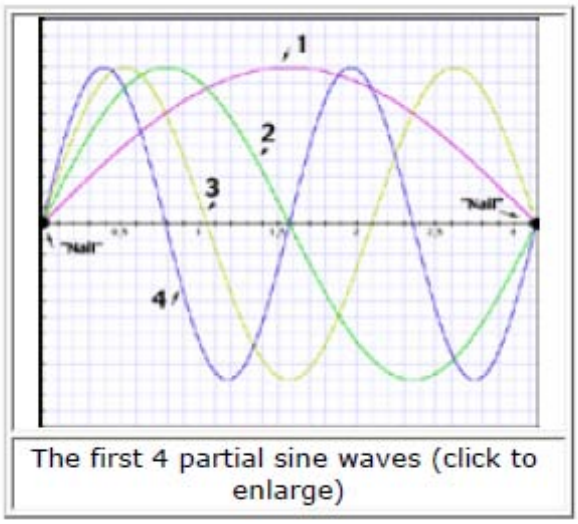

图 7 4 次谐波曲线

按照上述方法，构建一条具有 1000 个采样点的波形，我们需要 1000 条 sine 曲线。事实上，我们需要与采样点相同数量的 sine 曲线来构建原始曲线。

备注　　*原文为 1000，译者认为有误

　　**原文为 2000，译者认为有误

第四步：构建法则

从上面几段内容我们知道，构建信号需要一组 sine 曲线。我们考虑了曲线的频率，并且求出了构建信号需要的最高和最低 sine 曲线的频率。我们知道采样点的数量决定了最低频率（一次谐波），但是我们还不知道如何用这些具有谐波频率的 sine 曲线去构建原始的信号。通过叠加谐波 sine 曲线的形式构建原始信号，我们还需要知道一个参数。事实上，频率不是我们唯一关心的。我们还需要知道各次 sine 谐波的幅度，即构建原始信号需要知道每个谐波分量的大小。幅度是 sine 波形峰值的大小，即峰值到零基准线的距离。幅度越大，则我们听到的声音越大。所以，如果一个包含重低音的信号，无疑该信号的低频分量大于高频分量。更一般的是，低音中的低频信号幅度要比高频信号的幅度大。通过分析，我们需要确定各次谐波曲线的幅值来完成构建信号的法则。

第五步：离散 sine 变换的实现

如果能读到这里，您几乎就要完成探索傅里叶变换的旅程了。我们已经知道需要多少 sine 曲线，并且需要知道不同 sine 曲线的幅度来构建原始信号。需要多少 sine 曲线取决于采样点数的多少，因为它决定了 sine 谐波的最高和最低频率。然而我们还不知道怎样通过

采样信号来确定构建法则。直觉地，我们会将采样信号和已知 sine 曲线比较来确定该 sine 谐波的幅度。如果刚好相等，我们就认为原信号包含相同分量的该次谐波；如果我们发现该 sine 曲线与原信号不匹配，则认为原信号不包含该频率分量。我们如何有效地比较已知谐波和采样信号呢？十分幸运，数字信号处理方面的研究者早就给我们准备好了方法。事实上，该方法就是简单的乘法和加法的组合——将已知单位幅值的参考频率 sine 曲线与采样信号相乘。将乘得的结果相加，我们就得到了我们想要的 sine 谐波的幅值。为了说明该方法，下面是一段实现该方法的 C 语言代码：

Listing 1.1: The direct realization of the Discrete Sine Transform (DST):

```
#define M_PI 3.14159265358979323846

        long bin,k;
        double arg;
        for (bin = 0; bin < transformLength; bin++) {

                transformData[bin] = 0.;
                for (k = 0; k < transformLength; k++) {

                        arg = (float)bin * M_PI *(float)k / (float)
                        transformLength;
                        transformData[bin] += inputData[k] * sin(arg);

                }

        }
```

图 8 DST 的 C 语言实现

这段代码将储存在 inputData[0…transformLength-1]中的采样点转换为不同 sine 谐波幅值的数组 transformData[0…transformLength-1]。根据惯例，我们将参考频率 Bins 称为频率台阶，这意味着我们可以想象它是一个存放所求 sine 谐波分量幅值的容器。离散 sine 变换（DST）是在我们对信号一无所知时用到的一种处理方法，否则的话，我们可以用更高效的方法来确定各频率分量的幅值。

可能有人会问：为什么这样计算 DST 变换的赋值呢？为了形象地说明为什么将原始信号与已知频率分量相乘得到幅值，我们可以粗略地将此方法与自然世界中谐振器做个类比。sin（arg）项是谐振器的谐振频率。如果输入信号有我们寻找的谐波分量，输出就是与参考分量波形谐振的幅值。因为参考波形是单位幅度，所以输出直接就是参考频率分量的实际幅值。因为谐振器只是一个简单的滤波器，所以这种变换可以看作是信号经过一个中心频率在参考频点，且具有非常窄带宽的带通滤波器。这还有助于理解为什么傅里叶变换为信号滤波

提供了一个有效手段。

为了说明的完整性，必须指出以上方法是可逆的，如果已知信号的各次谐波分量，可以通过简单地相加来重建原信号。这个证明过程留给读者自己练习。同样的方法也适用于以 cosine 曲线为基础的信号重建，我们只需要将 sin(arg)换为 cos(arg)就可以得到离散 cosine 变换（DCT）。

现在，就像我们在本文开始时讨论的那样，实践中我们无法辨别 sine 或 cosine 信号。取而代之的是我们经常以正弦信号为测量基准，所以 sine 和 cosine 变换在实践中用处不大，除非是在一些特定的场合。正弦波形由于可以起始于周期中的任何位置，所以它比 sine 或 cosine 曲线更具代表性。我们记得 sine 曲线总是从 0 开始，cosine 曲线总是从 1 开始。当我们以 sine 曲线为参考时，cosine 曲线起始比它晚 1/4 个周期。通常用角度或弧度来表示这个偏移，这两个单位在三角函数中也经常用到。一个周期为 360° 或 2Π。因此 cosine 相对 sine 波形有 90° 或 1/2Π的偏移。这个偏移称为相位，我们可以看到 cosine 曲线是相对 sine 曲线有 90° 或 1/2Π相位偏移的正弦曲线。

相位有什么用？当我们不能总是限制信号起始于 0° 或 90° 时，通过频率、幅度和相位我们就可以唯一描述信号在任何一个时刻的形状。使用 sine 或 cosine 变换必须限制起始相位为 0° 或 90°，否则任意的相位会导致相近频率出现虚假峰值。这就像把一个圆石头放进一个方孔：你需要更小的圆石头去填充剩下的空间，需要更更小的石头再去填充剩下的空间。所以我们需要一种更具普遍意义的变换，该变换应建立在具有任意相位的正弦曲线的基础之上。

第六步：离散傅立叶变换

从 sine 变换到傅立叶变换的过度很简单，只需要用一种更“一般”的方法表示即可。sine 变换中我们用 sine 曲线表示每个频率分量，傅立叶变换中我们用 sine 和 cosine 曲线一起表示。也就是说，我们寻找的每个频率分量，我们都将测量信号与该具有该频率的 sine 和 cosine 曲线相比较。如果信号像 sine 曲线，则 sine 部分的幅度会大一些。如果信号更像 cosine 曲线，则 cosine 部分的幅度会大一些。如果信号像 sine 曲线取反的形式，则 sine 部分具有较大的负幅度。可以用带有正负号的 sine 和 cosine 相位表示任何给定频率的正弦曲线。

Listing 1.2: The direct realization of the Discrete Fourier Transform[3]:

```
#define M_PI 3.14159265358979323846

        long bin, k;
        double arg, sign = -1.; /* sign = -1 -> FFT, 1 -> iFFT */

        for (bin = 0; bin <= transformLength/2; bin++) {

                cosPart[bin] = (sinPart[bin] = 0.);
                for (k = 0; k < transformLength; k++) {

                        arg = 2.*(float)bin*M_PI*(float)k/(float)
                        transformLength;

                        sinPart[bin] += inputData[k] * sign * sin(arg);
                        cosPart[bin] += inputData[k] * cos(arg);

                }

        }
```

图 9 离散傅立叶变换的 C 语言实现

傅立叶变换带给我们什么好处？这个问题仍没解决。我曾经宣称傅立叶变换比 sine 和 cosine 变换好在它使用了正弦曲线。但是我们没有看到正弦曲线，这里仍然只有 sine 和 cosine 曲线啊。好吧，这需要一个额外的步骤对该问题加以解释：

Listing 1.3: Getting sinusoid frequency, magnitude and phase from the Discrete Fourier Transform:

```
#define M_PI 3.14159265358979323846

        long bin;
        for (bin = 0; bin <= transformLength/2; bin++) {

                /* frequency */
                frequency[bin] = (float)bin * sampleRate / (float)transformLength;

                /* magnitude */
                magnitude[bin] = 20. * log10( 2. * sqrt(sinPart[bin]*sinPart[bin] +
                cosPart[bin]*cosPart[bin]) / (float)transformLength);

                /* phase */
                phase[bin] = 180.*atan2(sinPart[bin], cosPart[bin]) / M_PI - 90.;

        }
```

图 10 DFT 数据转换

运行图 10 中的代码，我们可以将输入信号以正弦曲线和的形式最终表示出来了。k 次谐波的正弦曲线由 frequency[k],magnitude[k]和 phase[k]共同确定。它们的单位分别是 Hz，dB 和度。请注意该结果已将 sine 和 cosine 曲线转换为单一的正弦曲线，我们把 k 次正弦谐波的幅度称作“magnitude”，这个值始终为正。我们可以将幅值 1 与相位+或-180°联合来表示幅值-1。通常将傅里叶变换得到的数组 magnitude[]称作信号的幅度谱，phase[]称作信号的相位谱。

当幅值的单位是分贝时，我们的输入信号通常假定为[-1,1]分贝，对应幅值为 0dB 的全数字量程。作为一个有趣的 DFT 应用，图 10 的代码可以用在基于 DFT 的频谱分析程序里。

结论

正如我们看到的那样，傅里叶变换和相关的离散 sine 和 cosine 变换为我们提供了一个便利的工具，该工具可以将一个信号分解为一系列谐波频率的和。这些谐波信号可以是 sine，cosine 或正弦信号（正弦信号用 sine 和 cosine 联合表示）。傅里叶变换中用正弦信号引入相位的概念，从而使该变换更明确有效地适用于对纯 sine 信号、cosine 信号以及其他形式信号的分析。

傅里叶变换与待分析的信号无关，无论是正弦曲线或其他更复杂的信号，傅里叶变换的步骤都是一样的。这就是为什么傅里叶变换被称作非参数变换的原因，也意味着该方法在处理有已知信息的信号时不是特别有效。

我们现在也知道我们是在固定的频率点上重建输入信号，这些频点与输入信号包含的频率没有关系。因为我们选择参考 sine 和 cosine 曲线是根据对频率的感知，所以我们用来构建信号的这些频点是人为选择的。说到这里你立刻会想到这种情况，原始信号的频率落在两个谐波频点之间，而这种情况很容易发生。因此，一个恰巧落在两个频点之间的信号是不能用 DFT 变换来精确表示的。与原始信号频率相近的频点会尝试修正这个频率上的偏差，因此原始信号的能量就被分配在相近的几个频点上。这就是为什么 DFT 不能还原声音信号中基音和和声的原因（这也是为什么我们把 sine 和 cosine 曲线叫做分音，而不是和声或泛音）。

简单地说，DFT 如果不做进一步的处理，它不过是一个每个通道具有相位信息的有轻微交叠的窄带带通滤波器。DFT 只对信号分析、滤波和一些无关紧要的场合来讲有用，而大部分情况下它都需要进一步的处理。并且 DFT 可以被看作是其它非 sine 和 cosine 变换的一个特例。此概念的展开将超出本文的讨论范围。

最后，提及“快速傅立叶变换”（FFT）这个 DFT 的有效快速算法是非常重要的，该算

法由 Cooley 和 Tukey 于 1969 年提出（然而它的根源要追溯到 Gauss 和其他学者的著作）。FFT 是一种高效的 DFT 算法，与上面介绍的算法相比结果相同，用时却更少。然而，Cooley 和 Tukey 提出的 FFT 算法要求转换信号长度必须是 2 的整次方。实践中大部分应用都满足该条件。关于 FFT 不同的实现方法的文献很多，可以说有足够多的 FFT 实现方法，并且有一些是不受 2 的整次方条件限制的。下面的函数 smbFit()就是 FFT 的一种实现方法。

Listing 1.4: The Discrete Fast Fourier Transform (FFT):

```
#define M_PI 3.14159265358979323846

void smbFft(float *fftBuffer, long fftFrameSize, long sign)

/*

        FFT routine, (C)1996 S.M.Bernsee. Sign = -1 is FFT, 1 is iFFT (inverse)

        Fills fftBuffer[0...2*fftFrameSize-1] with the Fourier transform of the time domain data
        in fftBuffer[0...2*fftFrameSize-1]. The FFT array takes and returns the cosine and sine
        parts in an interleaved manner, ie. fftBuffer[0] = cosPart[0], fftBuffer[1] = sinPart[0],
        asf. fftFrameSize must be a power of 2. It expects a complex input signal (see footnote
        2), ie. when working with 'common' audio signals our input signal has to be passed as
        {in[0],0.,in[1],0.,in[2],0.,...} asf. In that case, the transform of the frequencies of
        interest is in fftBuffer[0...fftFrameSize].

*/

{

        float wr, wi, arg, *p1, *p2, temp;
        float tr, ti, ur, ui, *p1r, *p1i, *p2r, *p2i;
        long i, bitm, j, le, le2, k;

        for (i = 2; i < 2*fftFrameSize-2; i += 2) {
```

```
        for (bitm = 2, j = 0; bitm < 2*fftFrameSize; bitm <<= 1) {

            if (i & bitm) j++;
            j <<= 1;

        }

        if (i < j) {

            p1 = fftBuffer+i; p2 = fftBuffer+j;
            temp = *p1; *(p1++) = *p2;
            *(p2++) = temp; temp = *p1;
            *p1 = *p2; *p2 = temp;

        }

    }

    for (k = 0, le = 2; k < (long) (log(fftFrameSize)/log(2.)); k++) {

        le <<= 1;
        le2 = le>>1;
        ur = 1.0;
        ui = 0.0;
        arg = M_PI / (le2>>1);
        wr = cos(arg);
        wi = sign*sin(arg);
```

```
        for (j = 0; j < le2; j += 2) {

            p1r = fftBuffer+j; p1i = p1r+1;
            p2r = p1r+le2; p2i = p2r+1;

            for (i = j; i < 2*fftFrameSize; i += le) {

                tr = *p2r * ur - *p2i * ui;
                ti = *p2r * ui + *p2i * ur;
                *p2r = *p1r - tr; *p2i = *p1i - ti;
                *p1r += tr; *p1i += ti;
                p1r += le; p1i += le;
                p2r += le; p2i += le;

            }
```

```
                    tr = ur*wr - ui*wi;
                    ui = ur*wi + ui*wr;
                    ur = tr;

                }

        }

}
```

图 11 FFT 的一种 C 语言实现