实验 8 快速傅立叶变换问题

8.1 实验问题

在数值电路的传输中，为了避免信号干扰，需要把一个连续信号 **x(t)**先通过取样离散化为一列数值脉冲信号 **x(0)，x(1)，…… ,**然后再通过编码送到传输电路中。如果取样间隔很小，而连续信号的时间段又很长，则所得到的数值脉冲序列将非常庞大。因此，传输这个编码信号就需要长时间的占用传输电路，相应地也需要付出昂贵的电路费用。

那么能否经过适当处理是使上述的数值脉冲序列变短，而同时又不会丧失有用的信息？的经过研究，人们发现，如果对上述数值脉冲序列作如下的变换处理：

$$X(n)=\sum_{k=0}^{N-1}x(k)e^{-2pnki/N},n=0,1,...,N-1,i=\sqrt{-1} \qquad (7.1)$$

则所得到的新序列 **X(0)，X(1) ，**……将非常有序，其值比较大的点往往集中在某一很狭窄的序列段内，这将非常有利于编码和存储，从而达到压缩信息的目的。

公式（**7.1**）就是所谓的离散傅立叶变换，简称 **DFT**。现在我们来分析一下计算 **DFT** 所需要的工作量。如果我们不考虑公式（**7.1**）中指数项的运算，那么计算其每一个点 **X (n)** 需要 **N** 次复数乘法和 **N-1** 次的复数加法。显然当 **N** 很大时，这个工作量也非常巨大。正是由于这个原因，使得 **DFT** 的应用范围在过去很长的时间里受到了严格的限制。

注意到公式（**7.1**）是非常有规律性的，那么能否利用这种规律性来降低 **DFT** 的计算时间？

8.2 问题背景与分析

1965 年，凯莱和塔柯的提出了一种用于计算 **DFT** 的数学方法，大大减少了 **DFT** 的计算时间，同时又特别适用于硬件处理，这就是所谓的快速傅里叶变换，简称 **FFT**。

鉴于 **DFT** 的数据结构可以通过傅立叶变换的离散化获得，亦可通过三角插值得到，而本质上又同连续傅里叶分析有着极为密切的关系，因此，本部分将从傅立叶级数级数和傅立叶积分入手，通过离散导出 **DFT** 结构的来源，接着分析 **FFT** 的工作原理。

8.2.1 傅立叶变换

如果 **x(t)**是定义在整个实轴上的实值或复值函数，则其傅立叶变换可由下式给出：

$$X(f)=\int_{-\infty}^{\infty}x(t)e^{-2nift/T}dt,i=\sqrt{-1} \qquad (7.2)$$

若对任意参数 **f**，上述积分都存在，则（**7.2**）式子确定了一个函数 **X(f)**，称为 **x(t)** 的傅立叶变换。如果已知 **X(f)** 则利用如下的傅立叶逆变换，还可复原 **x(t)** ：

$$x(t)=\int_{-\infty}^{\infty}X(f)e^{2nift/T}df,i=\sqrt{-1} \qquad (7.3)$$

若 **x(t)** 和 **X(f)** 同时满足（**7.2**）、（**7.3**）式，则称他们是一个傅立叶变换对，记为 $x(t)\Leftrightarrow X(f)$ 。。通常 **X(f)** 是一个复函数，因此可以写成如下两部分：

$$X(f)=R(f)+I(f)i \quad , \quad i=\sqrt{-1} \qquad (7.4)$$

式子中 **R(f)** ，**I(f)** 分别是 **X(f)** 的实部和虚部。将上式表示为指数形式：

$$X(f)=|X(f)|e^{if(f)} \qquad i=\sqrt{-1} \qquad (7.5)$$

其中

$$|X(f)|=\sqrt{R^2(f)+I^2(f)} \qquad (7.6)$$

$$f(f)=tg^{-1}(\frac{I(f)}{R(f)}) \qquad (7.7)$$

工程技术中，常将 **x(t)** 看成一时间信号，相应的空间，称为时间域和空域；将其傅立叶变换 **X(f)** 看成频率函数，相应的空间称为频域。$|X(f)|$ 称为 **x(t)** 的傅立叶谱，而 $f(f)$ 称为其相角，这在物理上是有良好背景的。的譬如此频率的的含义可以这样来理解：应用欧拉公式可将指数项表示成正弦-余弦的形式，如果把（**7.2**）式解释成离散项和的极限，则显然 **X(f)**是包含了无限项正弦-余弦的和，而且 **f** 的每一个值确定了所对应的正弦-余弦的频率。

在以后的叙述中，我们不妨用 **t** 表示时间，用 **f** 表示频率；同时用小写字母表示时间函数，并用相应的大写字母表示其傅立叶变换。

傅立叶变换可很容易地推广到二维情形。假设 **x(t , s)**是连续的和可积的，且 **X(f , g)**是可积的，则相应的傅立叶变换对如下：

$$X(f,g)=\int_{-\infty}^{\infty}\int_{-\infty}^{\infty}x(t,s)e^{-2ni(ft+gs)/T}dtds, i=\sqrt{-1} \tag{7.7}$$

$$x(t,s)=\int_{-\infty}^{\infty}X(f,g)e^{2ni(ft+gs)/T}dfdg, i=\sqrt{-1} \tag{7.8}$$

8.2.2 离散傅立叶变换

尽管傅立叶级数和傅立叶变换具有非常优美的数学结构，但并不实用，并为他们都无法用有限字长的计算机作逻辑上的运算。为此，我们必须建立傅立叶变换的数值方法，并由此导出 **DFT** 数据结构的来源。

1）傅立叶积分的离散化

由于傅立叶变换无法用数字计算机进行逻辑运算，工程分析中，通常采用抽样的方法，观测 **x(t)**的一些离散值，然后利用数值积分将傅立叶变换离散化。

函数抽样是函数插值的逆过程，假定用取 **2N+1** 个互相间隔为 Δt 的节点的方法，当一个连续函数 **x(t)** 离散化为的一个序列：

$$\{x(-N\Delta t),...,x(-\Delta t),x(0),x(\Delta t),...,x(N\Delta t),\}$$

于是当 **N** 充分大时，有：

$$X(f)\approx\int_{-N\Delta t}^{N\Delta t}x(t)e^{-2nift}dt \tag{7.9}$$

现在我们把（**7.9**）式中的求积函数当成周期为 $2N\Delta t$ 的函数，以 $j\Delta t, j=0,\pm1,\pm2,...\pm N$ 为节点，对（**7.9**）式用复化梯形公式做数值积分得

$$X(f)\approx\Delta t\sum_{n=1-N}^{N}x(n\Delta t)e^{-2nifn\Delta t} \tag{7.10}$$

对 **f** 进行离散化，取 $f=j/(N\Delta t), j=0,1,2,...N-1$， 则上式可改写：

$$X(\frac{j}{N\Delta t})\approx\Delta t\sum_{n=1-N}^{N}x(n\Delta t)e^{-2nijn/N}, j=0,1,...,N-1 \tag{7.11}$$

这就是一维傅立叶积分的离散表达式。

2）离散傅立叶变换

在上面，我们普便的遇到了带有复指数乘积项的和式。实际上，这种特殊的数据结构可利用以下更为一般的方式定义。

给定 **N** 个实或复的数列**{x(0), x(1),……x(N-1)}**，定义

$$X(n)=\sum_{k=0}^{N-1}x(k)e^{-2pnki/N}, n=0,1,...,N-1, i=\sqrt{-1} \tag{7.12}$$

为**{x(k)}**的离散傅立叶变换，简称 **DFT**。

我们指出，（**7.11**）式可以转化成上述一般形式。这说明（**7.12**）式与傅立叶变换之间存在内在的联系，渴望获得与连续傅立叶变换相类似的性质，事实上确实如此。下面我们就来做这件事。

首先说明，由（**7.12**）式确定的序列**{X(n)}**可以恢复为原序列**{x(k)}**。

的事实上，在（**7.12**）式两边同乘以 $e^{2nijn/N}$，并对 **n** 从 **0** 到 **n-1** 求和得：

$$\begin{aligned}\sum_{n=0}^{N-1}X(n)e^{2pnji/N}&=\sum_{n=0}^{N-1}\sum_{k=0}^{N-1}x(k)e^{2pn(j-k)i/N}\\&=\sum_{k=0}^{N-1}x(k)\sum_{n=0}^{N-1}[e^{2pn(j-k)i/N}]^n\end{aligned} \tag{7.13}$$

$$= Nx(j) + \sum_{k=0,k\neq j}^{N-1} x(k)\frac{1-e^{2n(j-k)i}}{1-e^{2n(j-k)i/N}} \qquad (7.14)$$

注意到（7.14）式的和式中分子部分为 0，从而

$$x(j) = \frac{1}{N}\sum_{n=0}^{N-1} X(n)e^{2pnji/N},\ j = 0,1,...,N-1 \qquad (7.15)$$

今后我们称（7.15）式是（7.12）式的逆变换，并称{x(k)}和{X(n)}为离散傅立叶变换对，简记为 $x(k) \Leftrightarrow X(n)$ 离散傅立叶变换的这种可恢复性质，在工程技术中有着极为重要的应用。因为，可以通过抽样获得一组的离散值，并利用 DFT 转换为另一组数据，通过对变换数据的修改以及逆变换，达到对原数据的修正。这也正是 DFT 最具魅力的地方。

应用 DFT 作数值分析，抽样也变得相当简单。这时，可以抽取函数的任一片断，而无需象傅立叶变换离散化那样做对称抽样。

8.2.3 快速傅里叶变换

本节我们将把注意力集中在如何计算 DFT 上。如果我们不考虑（7.12）式中的复指数部分的运算，则求解（7.12）式共需要 N * N 次乘法和 N * (N-1)次加法。显然当 N 很大时，其工作量是相当可观的。为此，凯莱和卡柯提出了一种专门用于处理 DFT 的快速算法(FFT)，
大大减少了 DFT 的计算时间。

充分理解 FFT 的工作原理，这并不需要高深的数学知识，只要时刻的盯住 DFT 的数据结构即可。为了便于理解，我们先从最简单的情形入手。

1) FFT 直观发展

注意到（7.12）式可以表示成一个矩阵运算，而 FFT 实际上是一个矩阵分解算法，它对 N 的要求有一定的限制，通常 N 取成 2^r，其中 r 是正整数。

为了更加直观，这里我们假定 r=2, N =4 ，并引进记号

$$W_N = e^{-2pi/N}, i = \sqrt{-1} \qquad (7.16)$$

则（7.12）式可改写为如下的矩阵形式：

$$\begin{pmatrix} X(0) \\ X(1) \\ X(2) \\ X(3) \end{pmatrix} = \begin{pmatrix} W_4^0 & W_4^0 & W_4^0 & W_4^0 \\ W_4^0 & W_4^1 & W_4^2 & W_4^3 \\ W_4^0 & W_4^2 & W_4^4 & W_4^6 \\ W_4^0 & W_4^3 & W_4^6 & W_4^9 \end{pmatrix} \begin{pmatrix} x(0) \\ x(1) \\ x(2) \\ x(3) \end{pmatrix} \qquad (7.17)$$

注意到 $W_N^{kn} = 1$，k 为整数，则（7.17）式还可简化为

$$\begin{pmatrix} X(0) \\ X(1) \\ X(2) \\ X(3) \end{pmatrix} = \begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & W_4^1 & W_4^2 & W_4^3 \\ 1 & W_4^2 & W_4^0 & W_4^2 \\ 1 & W_4^3 & W_4^2 & W_4^1 \end{pmatrix} \begin{pmatrix} x(0) \\ x(1) \\ x(2) \\ x(3) \end{pmatrix} \qquad (7.18)$$

上式以及以后的式子中没有把 W_4^0 写成 1 ，完全是为了以后推广的需要。第二步我们要做的是，把上述矩阵分解为两个矩阵的乘积：

$$\begin{pmatrix} X(0) \\ X(1) \\ X(2) \\ X(3) \end{pmatrix} = \begin{pmatrix} 1 & W_4^0 & 0 & 0 \\ 1 & W_4^2 & 0 & 0 \\ 0 & 0 & 1 & W_4^1 \\ 0 & 0 & 1 & W_4^3 \end{pmatrix} \begin{pmatrix} 1 & 0 & W_4^0 & 0 \\ 0 & 1 & 0 & W_4^0 \\ 1 & 0 & W_4^2 & 0 \\ 0 & 1 & 0 & W_4^2 \end{pmatrix} \begin{pmatrix} x(0) \\ x(1) \\ x(2) \\ x(3) \end{pmatrix} \qquad (7.19)$$

上述分解基于以后要讲到的 FFT 算法理论，其中第一行和第二行的位置做了变换,，这是 FFT 本身所要求的。以后将会看到，这种交换有利于数据存储，成为输出倒置。今引进

$$\begin{pmatrix} x_1(0) \\ x_1(1) \\ x_1(2) \\ x_1(3) \end{pmatrix} = \begin{pmatrix} 1 & 0 & W_4^0 & 0 \\ 0 & 1 & 0 & W_4^0 \\ 1 & 0 & W_4^2 & 0 \\ 0 & 1 & 0 & W_4^2 \end{pmatrix} \begin{pmatrix} x(0) \\ x(1) \\ x(2) \\ x(3) \end{pmatrix} \quad (7.20)$$

则

$$\begin{pmatrix} X(0) \\ X(1) \\ X(2) \\ X(3) \end{pmatrix} = \begin{pmatrix} x_2(0) \\ x_2(1) \\ x_2(2) \\ x_2(3) \end{pmatrix} = \begin{pmatrix} 1 & W_4^0 & 0 & 0 \\ 1 & W_4^2 & 0 & 0 \\ 0 & 0 & 1 & W_4^1 \\ 0 & 0 & 1 & W_4^3 \end{pmatrix} \begin{pmatrix} x_1(0) \\ x_1(1) \\ x_1(2) \\ x_1(3) \end{pmatrix} \quad (7.21)$$

于是求解 X(n)分成了两次矩阵运算。下面我们来分析一下这个过程的工作量。

计算 $x_1(0)$ 需要的一个复数乘法和一个复数加法，即 $x_1(0) = x(0) + W_4^0 x(2)$

这里没有用 1 代替 W_4^0 是因为在一般情况下这一项通常不为 1。计算 $x_1(1)$ 同样需要一个复数乘法和一个复数加法。计算 $x_1(2)$ 只需要一个复数加法，这是因为 $W_4^2 = -W_4^0$，从而

$$x_1(2) = x(0) + W_4^2 x(2) = x(0) - W_4^0 x(2)$$

其中 $W_4^0 x(2)$ 在计算 $x_1(0)$ 时已经计算过。同样的原因，计算 $x_1(3)$ 只需要一个复数加法。

总结上述可知，计算 $x_1(k)$ 共需 4 个复数加法和个 2 复数乘法。类似的，计算 $x_2(k)$ 也需要 4 个复数加法和 2 个复数乘法。于是计算 X(n) 共需要 8 个复数加法和 4 个复数乘法。

如果直接利用（7.12）式进行计算，则共需要 16 个复数乘法和 12 个复数加法。显然，上述分解算法具有更高的效率，这正是 FFT 算法的思想。

更一般的，对 $N = 2^r$，FFT 算法将把原矩阵分解为 r 个 $N \times N$ 矩阵的乘积，每个因子矩阵具有最小数据的复数加法和复数乘法运算。如果推广上述结果，则当 $N = 2^r$ 时，FFT 需要 Nr/2 个复数乘法和 Nr 个复数加法。相应的，直接算法需要 N^2 个复数乘法和 N(N-1) 个复数加法，两者的工作量之比为：乘法 2N/r，加法 （7.N-1）/r，如果 N=1024 ，则 FFT 算法的乘法运算次数将降低为直接法的二百分之一，显然工作量节省是相当可观的。

2）以 2 为底的 FFT 算法。

假定 $N = 2^r$，则对任何不大于 N 的数都可以用不超过 r 位的二进制数表示。譬如，当 N=4 时，则十进制数 0，1，2，3 可以分别表示成二进制数 00，01，10，11。更一般的，当 n,k < N，可以用二进制重记为

$$\begin{cases} n = (n_0, n_1, ...n_{r-1}) = (n_0 + 2n_1 + ... + 2^{r-1} n_{r-1}) \\ k = (k_0, k_1, ...k_{r-1}) = (k_0 + 2k_1 + ... + 2^{r-1} k_{r-1}) \end{cases} \quad (7.22)$$

于是（7.12）式可以改写为

$$X(n_0, n_1, ...n_{r-1}) = \sum_{k_0=0}^{1} \sum_{k_1=0}^{1} ..... \sum_{k_{r-1}=0}^{1} x(k_0, k_1, ...k_{r-1}) W_N^p \quad (7.23)$$

其中 $p = (n_0 + 2n_1 + ... + 2^{r-1} n_{r-1})(k_0 + 2k_1 + ... + 2^{r-1} k_{r-1})$。

利用 $W_N^{a+b} = W_N^a + W_N^b$，则 W_N^p 可改写成

$$W_N^p = W_N^{2^{r-1} k_{r-1} n} W_N^{2^{r-2} k_{r-2} n} ..... W_N^{k_0 n} \quad (7.24)$$

现在我们来考虑（7.24）式中的每一项，并利用 $W_N^{2^r l} = W_N^{Nl} = 1$ 以简化之。首先考虑（7.24）式中的第一项，得

$$W_N^{2^{r-1}k_{r-1}n} = W_N^{2^{r-1}k_{r-1}(n_0+2n_1+.....+2^{r-1}n_{r-1})}$$
$$= W_N^{2^{r-1}n_0k_{r-1}} W_N^{2^r n_1 k_{r-1}} .....W_N^{2^r(2^{r-2}n_{r-1}k_{r-2})}$$
$$= W_N^{2^{r-1}n_0k_{r-1}}$$

类似的，对于（**7.24**）式中的第二项，有

$$W_N^{2^{r-2}k_{r-2}n} = W_N^{2^{r-2}(n_0+2n_1)k_{r-2}} W_N^{2^r n_2 k_{r-2}} .....W_N^{2^r(2^{r-3}n_{r-1}k_{r-2})}$$
$$= W_N^{2^{r-2}(n_0+2n_1)k_{r-2}}$$

更一般的可得

$$W_N^{2^{k-j}k_{r-j}n} = W_N^{2^{r-j}k_{r-j}(n_0+2n_1+.....+2^{j-1}n_{j-1})} \qquad (7.25)$$

应用（**7.25**）式，（**7.22**）式可以改写为

$$X(n_0,n_1,.....n_{r-1}) = \sum_{k_0=0}^{1}\sum_{k_1=0}^{1}.....\sum_{k_{r-1}=0}^{1} x(k_0,k_1,.....k_{r-1})$$
$$\times W_N^{2^{r-1}n_0k_{r-1}} W_N^{2^{r-2}(n_0+2n_1)k_{r-2}} .....$$
$$\times W_N^{k_0(n_0+2n_1+.....+2^{r-1}n_{r-1})}$$

利用分次求和，并对中间结果进行标号，上式可进一步改写为如下过程：

$$\begin{cases} x_1(k_0,k_1,.....k_{r-2},n_0) = \sum_{k_{r-1}=0}^{1} x(k_0,k_1,.....k_{r-1}) W_N^{2^{r-1}n_0k_{r-1}} \\ = x(k_0,k_1,.....k_{r-2},0) + x(k_0,k_1,.....k_{r-2},1) W_N^{2^{r-1}n_0}, \\ x_2(k_0,k_1,.....k_{r-3},n_1,n_0) = \sum_{k_{r-2}=0}^{1} x(k_0,k_1,.....k_{r-2},n_0) W_N^{2^{r-2}(n_0+2n_1)k_{r-2}} \\ = x_1(k_0,k_1,.....k_{r-3},0,n_0) + x_1(k_0,k_1,.....k_{r-3},1,n_0) W_N^{2^{r-2}(n_0+2n_1)} \\ ..... \\ x_r(n_{r-2},n_{r-1},.....,n_0) = \sum_{k_0=0}^{1} x_{r-1}(k_0,n_{r-2},.....,n_0) W_N^{(n_0+2n_1+.....+2^{r-1}n_{r-1})k_0} \\ = x_{r-1}(0,n_{r-2},.....,n_0) + x_{r-1}(1,n_{r-2},.....,n_0) W_N^{(n_0+2n_1+.....+2^{r-1}n_{r-1})} \\ X(n_0,n_1,.....n_{r-1}) = x_r(n_{r-2},n_{r-1},.....,n_0). \end{cases} \qquad (7.26)$$

上述递推方程正是凯莱和卡柯关于 **FFT** 算法的原始公式，注意到（**7.26**）式共有 **r** 重和式，每重和式共有 **N** 个方程，每个方程仅需一次乘法运算，因此 **FFT** 算法的总计算量为 **Nr** 次乘法和 **Nr** 次加法。如果同时考虑到 $W_N^p = -W_N^{p+N/2}$ ，则乘法的运算此书还可减少一半。

3） FFT 的数据结构

在公式（**7.26**）中，我们引进了中间变量 $x_1(k), x_2(k), ..., x_r(k)$ ，然而实际编程运算时并不需要这些数组。譬如在（**7.26**）式中计算 $x_1(k_0,k_1,.....k_{r-1},0)$ 和 $x_1(k_0,k_1,.....k_{r-1},1)$ 时，只用到了 $x(k_0,k_1,.....k_{r-1},0)$ 和 $x(k_0,k_1,.....k_{r-1},1)$ 的值，且在计算其他分量时将不再使用这两个数据点。一次，我们依然可以把 $x_1(k_0,k_1,.....k_{r-1},0)$ 和 $x_1(k_0,k_1,.....k_{r-1},1)$ 存放在 $x(k_0,k_1,.....k_{r-1},0)$ 和 $x(k_0,k_1,.....k_{r-1},1)$ 所在的单元内。对于其他中间变量的存储也是如此，并不需要辅助内存，但最终结果 **X(n)** 将在第 n' 单元找到。其中

$n = (n_0 + 2n_1 + ... + 2^{r-1}n_{r-1})$， $n' = (n_{r-1} + 2n_{r-2} + ... + 2^{r-1}n_0)$

例如当 **N = 4** 时，由于 **0**，**1**，**2**，**3** 克表示成二进制数 **00**，**01**，**10**，**11** ，于是

$$\begin{cases} X(0) = X(0,0) = x_2(0,0) = x_2(0), \\ X(1) = X(1,0) = x_2(1,0) = x_2(2), \\ X(2) = X(0,1) = x_2(0,1) = x_2(1), \\ X(3) = X(1,1) = x_2(1,1) = x_2(3), \end{cases}$$

这就是为什么（**7.19**）式中 **X(1)**和 **X(2)**错位的原因。

以上我们给出了以 **2** 为底 **FFT** 算法。**FFT** 算法还可以以其他的自然数为底进行计算，读者可以自己推导。

8.3 算例：

考虑函数：

T	0.5	0.75	1	1.25
Y(t)	2	3	4	4

令 **x(k)=y(0.5+0.25*k),k=0,1,2,3** ，则得到如下的离散抽样函数：

K	0	1	2	3
X(k)	2	3	4	4

利用这四个抽样值进行傅立叶变换如下：

$$X(0) = \sum_{k=0}^{3} x(k)e^{0} = x(0) + x(1) + x(2) + x(3) = 13$$

$$X(1) = \sum_{k=0}^{3} x(k)e^{-2pki/4} = 2e^{0} + 3e^{-pi/2} + 4e^{-pi} + 4e^{-3pi/2} = -2 + i$$

$$X(1) = \sum_{k=0}^{3} x(k)e^{-4pki/4} = 2e^{0} + 3e^{-pi} + 4e^{-2pi} + 4e^{-3pi} = -1$$

$$X(1) = \sum_{k=0}^{3} x(k)e^{-6pki/4} = 2e^{0} + 3e^{-3pi/2} + 4e^{-3pi} + 4e^{-9pi/2} = -2 - i$$