浙江大学

本科课程论文

学生姓名 周灿松

学生学号 3190105055

指导教师 胡浩基

专业 信息工程

所在学院 信息与电子工程学院

完成日期 2021 年 6 月 25 日

快速傅里叶变换的 MATLAB 实现

周灿松

2021 年 6 月 25 日

摘要：本次大作业采用了非递归的方法实现了基 2 的快速傅里叶变换算法，并且利用插值的方法实现了长度非 2^n 的序列的傅里叶变换。在数据规模小于 2^{17} 次方时表现良好。本方法一个比较巧妙的点在于利用了一种特别的方法实现了二进制倒序。

关键词：fft

1 问题的提出

因为 LTI 系统对复指数信号响应具有一种特别简单的形式，因此将用复指数信号 e^{st} 或者 z^n 作为一类基本信号来表示一般任意信号。然后根据 LTI 系统的叠加性质,LTI 系统对任意一个由这些基本信号的线性组合而成的输入信号的响应就是系统对这些基本信号单个响应的线性组合, 这就提供了另一种非常方便的 LTI 系统分析方法: 变换域分析法。

而提及变换频域分析，就不得不提到傅里叶变换了。基于傅里叶变换的频域分析法为信号与系统的分析、设计和理解提供了一种非常有效的基本方法，这使得我们能够从频域的角度获得对信号和 LTI 系统的性质的更加深入的理解，也使得信号与 LTI 系统得到了更为广泛的应用。

而随着微电子技术以及数字计算机的出现与发展，使得离散时间信号与系统的具体应用领域得到了很大拓展与发展。20 世纪 60 年代初期，库利和图基提出的快速傅里叶变换算法，使得傅里叶变换的运算量减少了几个数量级，极大地推动了数字信号处理这一学科的发展。直到现在，FFT 算法依旧是数字信号处理的经典方法。而本次设计作业，通过自己学习并编写 FFT 算法，能够极大程度地加深对傅里叶变换的理解和应用，了解计算机进行离散傅里叶变换的过程。

2 原理及算法

在查阅了相关资料之后，发现 FFT 大致分为两类：固定基以及混合基实现。MATLAB 中 fft 便是采用了混合基的方法实现的快速傅里叶变换，这种方法有点在于可以处理非 2^n 个点的傅里叶变换，计算速度较快。但是由于时间的紧迫以及能力的限制，混合基算法我仅了解了相关知识，在实现 FFT 算法时仍旧选择了基 2 的算法实现。此种方法的方便之处在于算法实现较为简单，编程实现难度较低，虽

然运算速度比不上 MATLAB 中的 fft，但是在编写此种算法的过程中也能够让人体会到 FFT 的巧妙之处。

2.1 程序实现方式

胡老师给的推荐代码中采用的是递归的方式进行实现，但我认为递归的方式虽然编程方式简单，但由于递归算法本身的限制，有着一些不好之处。递归需要调用系统的堆栈，消耗的存储空间会比非递归的方式大很多，而且在递归层数太深，也多了系统崩溃的可能性。相对于递归算法，非递归的方式虽然稍微复杂一点且没有递归的方式容易让人理解，但它的执行时间仅与循环的次数有关，没有太多的额外开销，同时也节约了创建新的空间的时间，效率更高。

在利用蝶形图作为编程工具之后，代码的编写难度已经得到了很大程度的减小，所以在经过一系列权衡之后，我决定采用非递归的方式进行实现，最后结果也证明了这能够极大地减少程序的运行时间

2.2 编码倒序的实现

因为程序选用了非递归的方法实现，编码倒叙的实现就不能像递归一样很自然地奇偶分离了，在此处，利用倒叙后二进制代码的特征，采用了一种非常巧妙的方法将其实现。因为 MATLAB 中移位操作符被用作了别的用途且数组下标从 1 开始，介绍算法时采用 C 语言代码。

```
1     static int rev[X];//X为序列长度
2     for(int i=0;i<X;i++)
3         rev[i]= (rev[i>>1]>>1)|((i&1)<<(K-1)) ;//K为二进制编码的位数
```

首先我们初始化一个长为 X 的全 0 数组，以 3 位二进制数为例，观察序列我们可以发现 x 的倒叙的低 2 位刚好为 x»1 倒叙的高 2 位，所以我们可以通过将 x»1 的二进制编码右移一位得到 x 处的低 2 位；至于 x 的最高位，我们将 x 与 1 取与就可得到，将其左移 2 位于上一步的结果取或就得到了 x 的倒叙序列. 通过循环，我们在计算 x 处值时，前 x 个数的倒叙已经计算得出，所以此算法能够实现。

例子：求 3 的倒叙

$\because i = 3; 3 >> 1 = 1$

$\therefore rev[i >> 1] = rev[1] >> 1 = 010, (3\&1) << 2 = 100$

$\therefore rev[3] = 010|100 = 110$

结果正确。

序号	二进制	倒序后二进制
0	000	000
1	001	100
2	010	010
3	011	110
4	100	001
5	101	101
6	110	011
7	111	111

在利用 MATLAB 实现时，由于 MATLAB 寻址从 1 开始，并且移位操作不够方便，在其中一部分代码我采用了除 2，向下取整然后强制类型转换的方式实现，代码如下：

```
1      rev = zeros(length_after_add_zero);
2      for i = 0:1:length_after_add_zero-1
3          rev(i+1) = bitor(int32(floor( (rev( floor(i/2)  + 1 ))/2 ) ) , ...
               bitshift(int32(bitand(int32(i),1)), bit_counter-1)) + 1;
4      end
```

2.3 蝶形运算设计

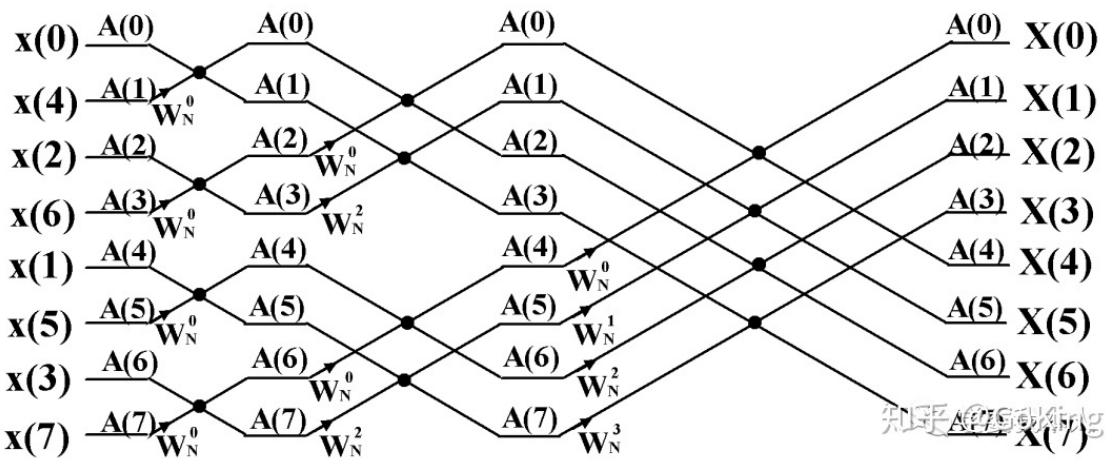

图 1: 蝶形图

定义旋转因子：$W_N^P = e^{-j\frac{2\pi}{N}P}$

按照蝶形图从左向右编程，L 为蝶形运算的层数，B 为每个蝶形运算输入的两个数据的间隔，P 则是旋转因子的 P 值，下文中的代码便能实现 bit_counter 层的蝶形图。

```
    for L = 1:1:bit_counter
        B =int32(pow2(L-1));
        for m = 0:1:B-1
            k = int32(pow2(bit_counter - L));
            P = m*k;
            for i = 0:1:k-1
                r = m + 2*B*i;
                temp = x(r+1);
                x(r+1) = x(r+1) + x(r+B+1) * ...
                    exp(-1j*double(2*pi*double(P)/double(length_add)));
                x(r+B+1) = temp - x(r+B+1) * ...
                    exp(-1j*double(2*pi*double(P)/double(length_add)));
            end
        end
    end
```

2.4 实现非 2^n 个点的 FFT

因为采用了基二的傅里叶算法，这就导致了无法实现非 2^n 个点的傅里叶变换，在处理这一点时，我采取了补零的方法实现。

课上我们学到，补零之后频域图的包络并不会发生改变，所以我们可以在补零之后利用插值的方法求出非 2^n 个点的傅里叶变换，插值时采用了最为简单的一次插值。在点的数目较少时，这会带来较大的误差，为了尽可能减少这一误差，我至少将序列补零至了 2^{10} 个点，这虽然带来了一定的时间消耗，但就算是只有 3 个点的傅里叶变换，也能保证小数点后 4 为和 MATLAB 提供的 fft 函数计算出的值完全相同，我个人认为此处带来的时间消耗是值得的。具体代码如下：

```
    if(length_after_add_zero ≠ length_x)
    for m = 1:1:length_x
        number = ...
            floor(double((m-1)*length_after_add_zero)/double(length_x));
        after = double(number+1)/double(length_after_add_zero);
```

```
5           before = double(number)/double(length_after_add_zero);
6           this = double(m-1)/double(length_x);
7           Y(m) = (after-this)*x(number+1)*double(length_after_add_zero) ...
               + (this - before)*x(number+2)*double(length_after_add_zero);
8       end
9       end
```

3 验证

3.1 代码

```
1       clc
2       clear
3       %%
4       N=128,A0=255;PI=3.1415926;
5       i=1:N;
6       x=A0 * (sin(2*PI*i/25)+sin(2*PI* i * 0.4 ));
7       tic;
8       y=fft(x,N);%进行傅里叶变换求得幅度谱
9       toc;
10      z=abs(y);%对求得的幅度谱加绝对值
11      figure(1);
12      subplot(2,2,1);plot(i,x);title('时域图');
13      subplot(2,2,2);plot(i,z);title('MATLAB自带fft测试结果')
14      tic;
15      var2 = fftNew(x);
16      toc;
17      subplot(2,2,3);plot(i,x);title('时域图');
18      subplot(2,2,4);plot(i,abs(var2));title('自己所写fftNew测试结果');
19      %%
20      N=234,A0=255;PI=3.1415926;
21      i=1:N;
22      x=A0 * (sin(2*PI*i/25)+sin(2*PI* i * 0.4 ));
```

```
    tic;
    y=fft(x,N);%进行傅里叶变换求得幅度谱
    toc;
    z=abs(y);%对求得的幅度谱加绝对值
    figure(2);
    subplot(2,2,1);plot(i,x);title('时域图');
    subplot(2,2,2);plot(i,z);title('MATLAB自带fft测试结果')
    tic;
    var2 = fftNew(x);
    toc;
    subplot(2,2,3);plot(i,x);title('时域图');
    subplot(2,2,4);plot(i,abs(var2));title('自己所写fftNew测试结果');
```

3.2 $N = 2^n$ 时正确性验证

由下图可以看出，在 $N = 2^n$ 时，fftNew 计算出的结果和 fft 函数结果一样，结果正确。

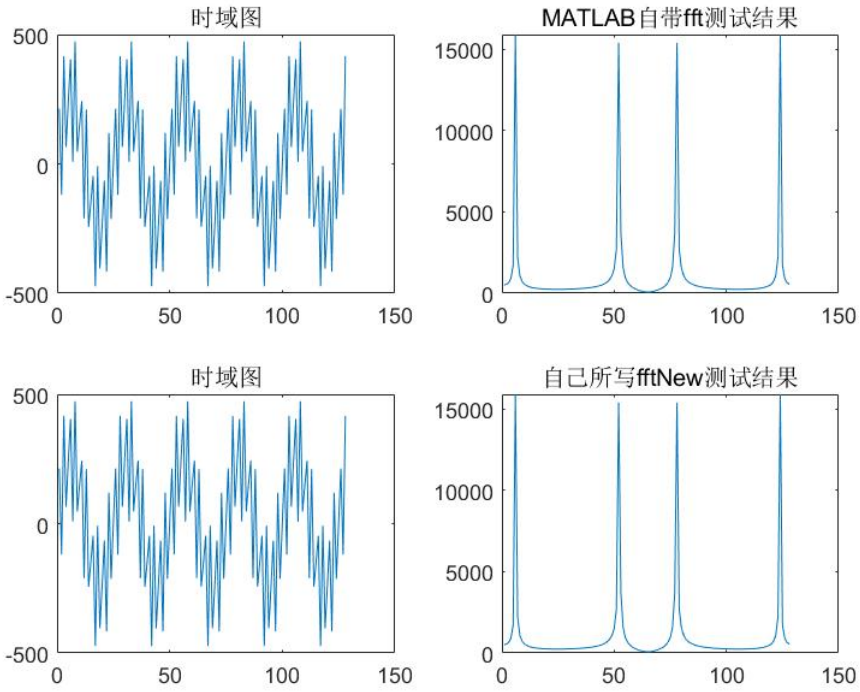

图 2: $N = 2^n$

3.3 $N! = 2^N$ 时正确性验证

由下图可以看出，当 $N! = 2^N$ 时结果正确

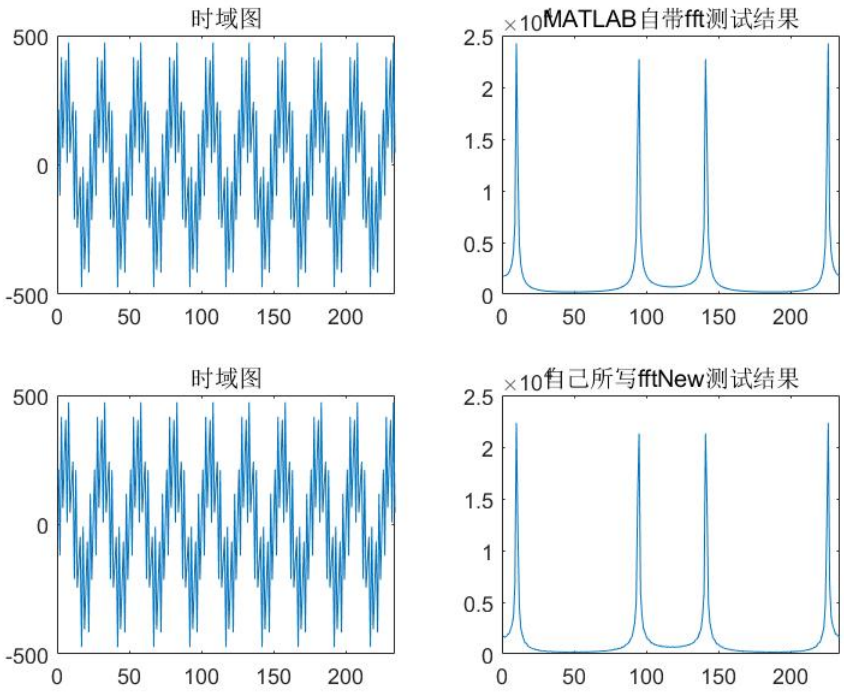

图 3: $N! = 2^n$

3.4 运行时间

第一个时间为系统 fft 耗时，第二个时间为 fftNew 耗时，运行时间较为理想。

```
N =

   128

历时 0.000080 秒。
历时 0.002289 秒。

N =

   234

历时 0.000086 秒。
历时 0.002024 秒。
```

图 4:

4 结论

通过上述的验证可以发现，自己编写的 fft 函数正确性能够经受住考验，不过相较于 MATLAB 中的 fft，在运行速度上就是硬伤了。而且，除了运行速度，当数据规模达到 2^17 次方时就没法计算了。