

数字信号处理课程设计——FFT

BobAnkh

说明：本项目基于 mingw-w64 与 c++17 的环境编译调试，采用 -O2 作为编译参数；在 VisualStudio2017 中以 release 模式编译测试得到了同样结果

一、 FFT 程序设计说明

本 FFT 程序主要包含在 fft.cpp 文件中，需要读取输入数据文件 data.txt，同时会将 DFT 结果输出到 dft_result.txt 中，将 FFT 结果输出到 fft_result.txt 中，并且会在终端输出 DFT 和 FFT 计算分别所用的时钟周期数（利用 clock() 函数读取系统时钟周期数作差得到）。程序主要分为三个部分：

1、 初始数据处理准备

从 data.txt 文件中读取数据，数据格式为每行一个复数数据的实部和虚部，以空格隔开。因为 C++ 自带 complex 模板类，所以直接采用其来构造复数对象，并且采用 STL 中的 vector 来存储复数对象数组。同时为了更好的泛用性，对数据长度也做了检查，如果不足 2 的幂次则会补零到 2 的幂次。

2、 直接计算 DFT

同样定义了 vector 容器用于存放 DFT 的结果，并且生成了 W_N^1 ，采用两层循环按照 DFT 的定义公式逐个计算结果。计算完成后将 DFT 结果写入到文件中，格式与输入文件相同。计算相关的核心代码如下：

```
// DFT

vector<complex<double>> X_DFT; // 定义了复数类型的 vector 容器用于
存放直接 DFT 的结果

complex<double> W = polar(1.0, -2 * PI / N);
int dft_start_time = clock();
for (int i = 0; i < N; i++){
    complex<double> dft_tmp(0,0);
    for (int j = 0; j < N; j++){
        dft_tmp = dft_tmp + c_x[j] * pow(W, i * j);
    }
    X_DFT.push_back(dft_tmp);
}

int dft_end_time = clock();
```

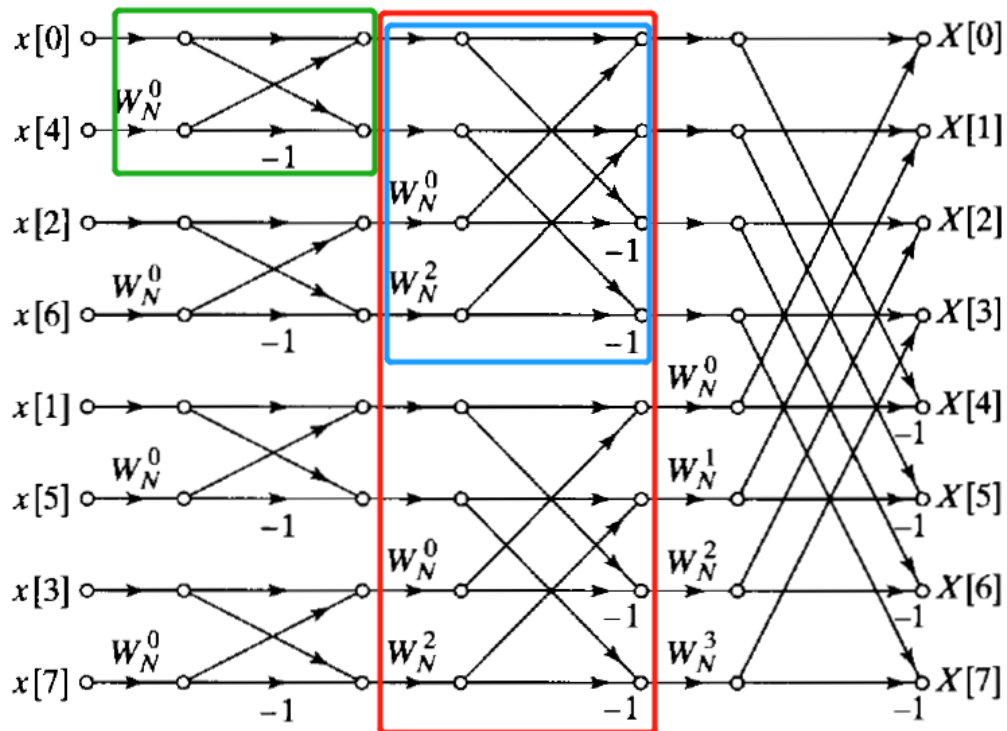
3、 计算 DIT-FFT

本次课程设计采用了 DIT-FFT 的形式来实现 FFT。同样定义了 `vector` 容器用于存放 FFT 的结果。

因为采用了 DIT-FFT，所以对于输入需要先进行倒位序处理，该模块采用循环实现，对于每个位置的输入，直接检索其对应的倒位序，将该倒位序对应的数据填充至此。相关代码如下：

```
// DIT_FFT
// 得到调整顺序后的输入序列 (实现倒位序)
vector<complex<double>> X_DIT_FFT(N); // 定义了复数类型的 vector
容器用于存放 DIT-FFT 的结果
for (int i = 0; i < N; i++){
    int rev_seq = 0;
    int tmp_d = i;
    for (int j = m - 1; j >= 0 ; j--){
        int tmp = tmp_d % 2;
        tmp_d = tmp_d / 2;
        rev_seq = rev_seq + (1<<j) * tmp;
    }
    X_DIT_FFT[i] = c_x[rev_seq];
}
```

主体运算部分，采用三层循环实现。最外层的循环是逐级的，即一级一级依次完成运算，如下示意图中红色框表示一级；中层循环是逐蝶形块的（在不同层中会由不同数量的最小蝶形构成），如下示意图中蓝色框表示一个蝶形块；最内层循环就是蝶形块内每一个最小蝶形的运算，如下示意图中绿色框表示一个最小蝶形，并且由于 DIT-FFT 无需引入额外存储，所以这里做的就是原地变换。如此三层循环完成 DIT-FFT 计算，计算完成后同样将结果写入到文件中。示意图如下：



计算相关的核心代码如下：

```
// DIT-DFT 运算
int fft_start_time = clock();

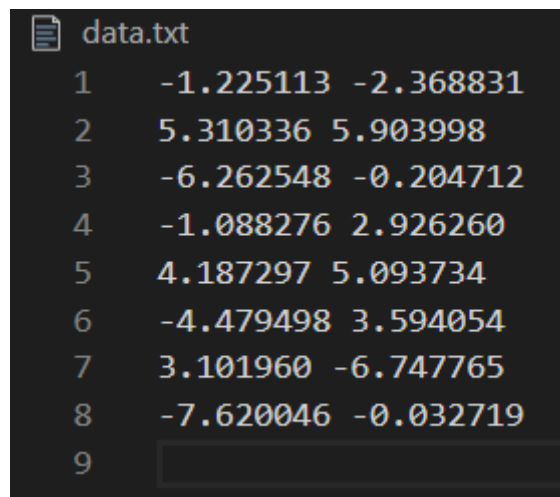
// 每一级运算
for (int i = 0; i < m; i++){
    // 每一组蝶形块运算
    for(int j = 0; j < N; j=j + (1<<(i+1))){
        int bf_num = 1<<i;
        // 每个最小蝶形计算
        for (int k = 0; k < bf_num; k++){
            complex<double> input1 = X_DIT_FFT[j + k];
            complex<double> input2 = X_DIT_FFT[j + k + bf_num
] * pow(polar(1.0, -2 * PI / (1<<(i+1))), k);
            X_DIT_FFT[j + k] = input1 + input2;
            X_DIT_FFT[j + k + bf_num] = input1 - input2;
        }
    }
}

int fft_end_time = clock();
```

二、 DFT 与 FFT 计算时间比较

我在本次代码中采用的是 C++ 自带的 `clock()` 函数来计算 DFT 和 FFT 运行的时间。`clock()` 函数返回的是系统时钟周期数，而根据 C++ 本身的定义，也可以将时钟周期数转换为实际用时（1 个时钟周期表示运行 1ms），所以也可以直接认为计算的是两者运行的时间（以 ms 为单位）

先检验 DFT 和 FFT 计算的正确性。为了方便手动验证，所以只用 MATLAB 随机生成 8 个复数，输出到文件 `data.txt` 中：



```
data.txt
1    -1.225113 -2.368831
2     5.310336 5.903998
3    -6.262548 -0.204712
4    -1.088276 2.926260
5     4.187297 5.093734
6    -4.479498 3.594054
7     3.101960 -6.747765
8    -7.620046 -0.032719
9
```

将随机生成的 8 个复数同样保存在 `xn` 中：

```
>> xn
xn =

1 至 6 列

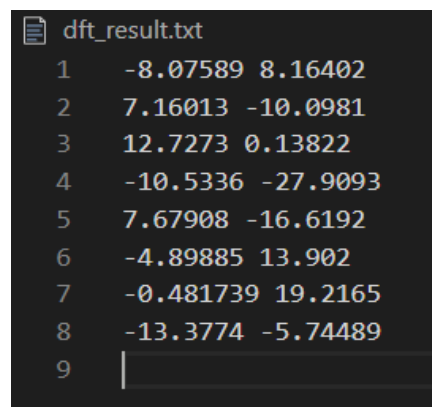
-1.2251 - 2.3688i    5.3103 + 5.9040i   -6.2625 - 0.2047i   -1.0883 + 2.9263i    4.1873 + 5.0937i   -4.4795 + 3.5941i

7 至 8 列

3.1020 - 6.7478i   -7.6200 - 0.0327i
```

使用 C++ 程序做 DFT 和 FFT，分别查看其输出结果文件：

DFT (`dft_result.txt`):



```
dft_result.txt
1    -8.07589 8.16402
2     7.16013 -10.0981
3    12.7273 0.13822
4    -10.5336 -27.9093
5     7.67908 -16.6192
6    -4.89885 13.902
7    -0.481739 19.2165
8    -13.3774 -5.74489
9
```

FFT（fft_result.txt）

```
fft_result.txt
1  -8.07589 8.16402
2  7.16013 -10.0981
3  12.7273 0.13822
4  -10.5336 -27.9093
5  7.67908 -16.6192
6  -4.89885 13.902
7  -0.481739 19.2165
8  -13.3774 -5.74489
9
```

对 xn 在 MATLAB 中采用 fft 进行变换，得到结果：

```
>> fft(xn)

ans =

1 至 6 列

-8.0759 + 8.1640i    7.1601 -10.0981i    12.7273 + 0.1382i   -10.5336 -27.9093i    7.6791 -16.6192i    -4.8988 +13.9020i

7 至 8 列

-0.4817 +19.2165i   -13.3774 - 5.7449i
```

由此可以说明 DFT 和 FFT 计算程序的正确性

接着比较 DFT 和 FFT 计算时间。继续随机生成不同长度的数组，得到变换所需的时间，将数据收集如下：

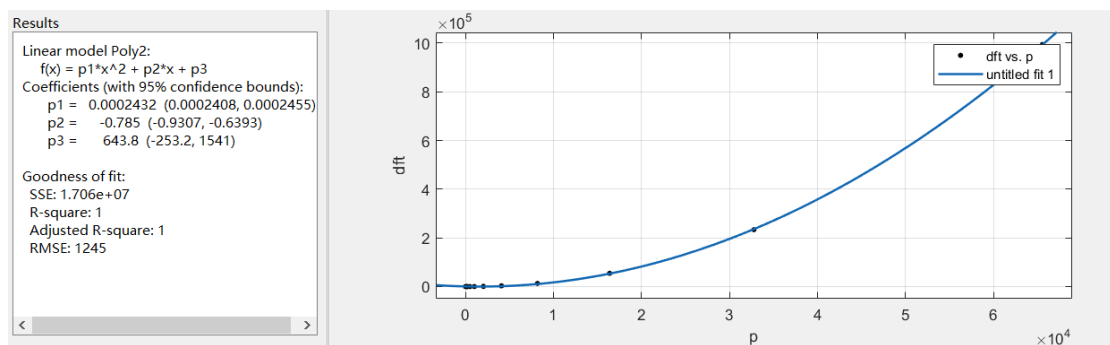
数组长度	DFT 所用时间(ms)	FFT 所用时间(ms)
8	0	0
16	0	0
32	0	0
64	1	0
128	2	0
256	8	0
512	34	0
1024	151	0
2048	654	1
4096	2880	1
8192	12893	2

16384	54280	6
32768	233685	14
65536	994017	31

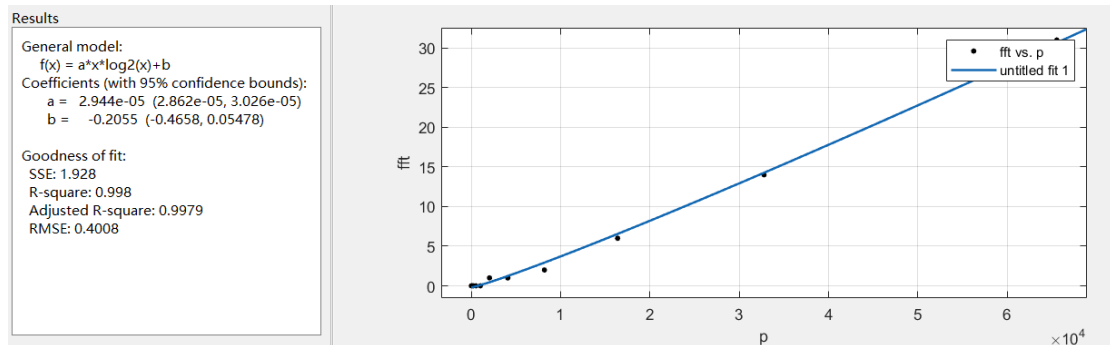
将上表中的数据采用 MATLAB 分别用不同公式进行拟合之后（使用 cftool 进行拟合），得到如下结果，可以看到，拟合结果都比较好， R^2 分别是 1 和 0.998，FFT 略有拟合误差的原因也是因为其数值较小因而以时钟周期数（ms）为单位会有一定的误差。

而由拟合结果也可以较好地说明 DFT 用时是 $O(N^2)$ 的，而 FFT 用时是 $O(N\log_2 N)$ ，并且也可以明显看出 DFT 计算用时比 FFT 要大很多。

DFT 拟合结果：



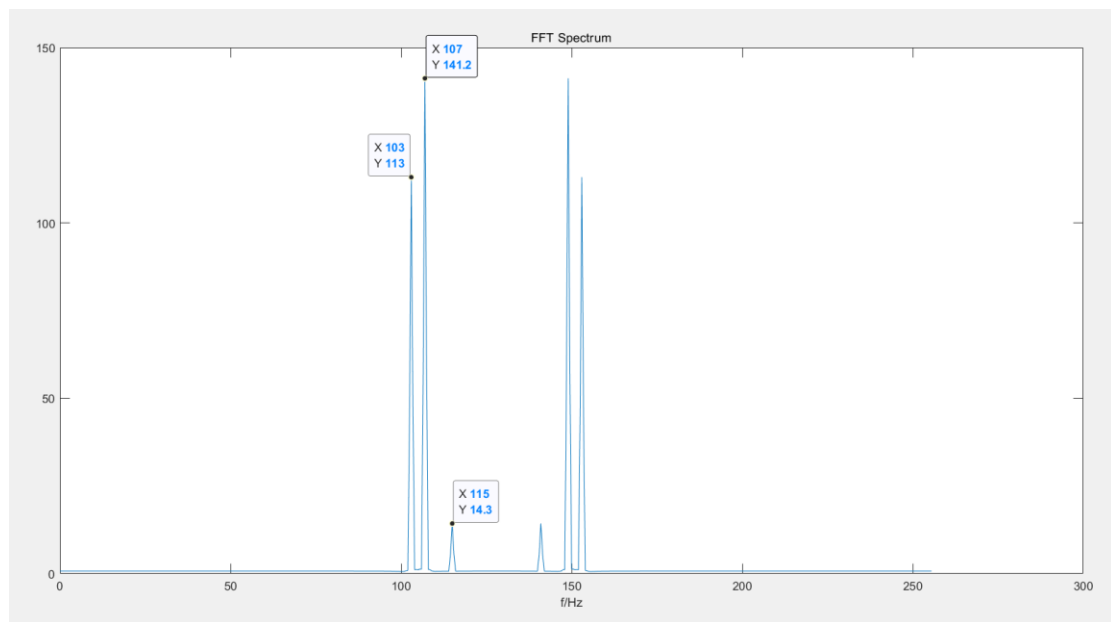
FFT 拟合结果：



三、 频谱分析

考虑到待分析信号最高频率只有 115Hz，所以采样频率只需要大于其两倍（大于 230Hz 即可），同时，考察 3 个信号频率，我认为选取 1Hz 的频率分辨力足够了，又因为实现的是基 2-FFT，所以综合考量下来，我选择了 256Hz 的采样率。在窗函数的选择方面，因为有两个频率相对还是比较接近的，所以选择可以自由选择对旁瓣抑制程度的切比雪夫窗作为窗函数，并且设置了 45dB 的抑制比例，使用 `mainlobe_len.m` 绘制出该窗的幅度谱，发现在此时的参数设置下，选用 512 个采样点数，可以达到 1Hz 的频率分辨力。

下图即为采用 FFT 结果所绘制出来的频谱图：



左半的 3 个尖峰分别就是 103Hz，107Hz，115Hz 这三个频点，也可以看到它们之间幅值的比值就是时域各频率信号幅度的比值，由此实现了对于所要求的连续信号的频谱分析。

四、 总结

这一次的课程设计让我自己动手实现了一下 DFT 和 FFT，一来我让我重新回顾了 C++ 的编写，二来让我对于 DFT 和 FFT 的原理以及实现有了更加深入清晰的认知，之前对于 FFT 只是会照着公式等硬算，但是经过这样一次大作业下来，对于其中一些步骤和整体层级之间的变化等等，都有了更清楚的了解，而且也真切地体会到了 FFT 算法在时间性能上的优势，可以说是收获颇丰。

五、 文件清单

文件名称	说明
fft.cpp	用于实现 DFT 和 FFT 的 C++源代码
fft_analysis.m	用于根据指定参数将待测信号数据输出到文件，并等待 C++程序完成变换，然后读取变换结果绘制频谱
random_data.m	用于随机生成 M 点的复数数据输出到文件，供 C++程序使用
mainlobe_len.m	用于绘制切比雪夫窗的幅度谱，确定主瓣宽度