基2 FFT编写课程设计报告

一、FFT程序编写说明

1、采样

```
void Sample(double* x,int M,double fs)
{
    for (int i = 0; i <=M; i++) {
        x[i] = 0.8 * sin((double)(2 * pi * 103 * i) / fs) + sin((double)(2 * pi
* 107 * i) / fs) + 0.1 * sin((double)(2 * pi * 115 * i) / fs);
    }

}
```

对全局数组*x[n]*以采样率*fs*采样，由于之后要进行加窗处理，因此只采*0~M*这*M+1*个点

2、加窗&补零

```
void Windowing(double* x,int M,int N,int window_type)
{
    switch (window_type) {
    case 1://矩形窗

        break;
    case 2://三角窗
        for (int i = 0; i <= (int)(M / 2); i++) {
            x[i] = x[i] * 2 * i / M;
        }
        for (int i = (int)(M / 2) + 1; i <= M; i++) {
            x[i] = 2*x[i]-x[i] * 2 * i / M;
        }
        break;
    case 3://汉宁窗
        for (int i = 0; i <= M; i++) {
            x[i] = x[i] * 0.5 * (1-cos(2*pi*i/M));
        }
        break;
    case 4://哈明窗
        for (int i = 0; i <= M; i++) {
            x[i] = x[i] * (0.54 - 0.46*cos(2 * pi * i / M));
        }
        break;
    case 5://布莱克曼窗
        for (int i = 0; i <= M; i++) {
            x[i] = x[i] * (0.42 - 0.5 * cos(2 * pi * i / M) + 0.08 * cos(4 * pi
* i / M));
        }
        break;
    }
    for (int i = M + 1; i < N; i++) { //补零
        x[i] = 0;
```

```
    }
}
```

通过参数*window_type*选择加窗类型，并实现加窗之后数列补零到长度*N*

3、Rader算法实现输入序列倒位序

```
void Rader(double *x, std::complex<double>* X,int M,int N)
{
    int cur_rev = 0;
    int k = N / 2;   //权系数初始值
    X[0].real(x[cur_rev]);
    X[0].imag(0);
    for (int j = 1; j <= N - 1; j++) {
        if (cur_rev < k) { //j-1的倒序数的最高位为0，说明从j-1到j只有最低位从0变1，没有进位
            cur_rev = cur_rev + k;//把j-1对应的倒序数的最高位从0变成1得到当前j的倒序数

        }
        else {
            while (cur_rev >= k) {    //j-1的倒序数的最高位为1，则从j-1到j有进位
                cur_rev = cur_rev - k;//把j-1的倒序数最高位从1变为0
                k = k / 2;            //次高位为1，则更新权系数，这个循环实际上在求从j-1到j进了几位
            }
            cur_rev = cur_rev + k;    //最高位为0时跳出循环，把最高位置1得到当前倒序数
            k = N / 2;//还原权系数值
        }
        X[j].real(x[cur_rev]);
        X[j].imag(0);
    }
 }
```

得到倒位序*x[n]*并转换成complex型数组*X[n]*

4、FFT函数

```
std::complex<double>* FFT(std::complex<double>* X, std::complex<double>* X_temp,int N)
{

    std::complex<double>* W_N = (std::complex<double> *)malloc((int)(sizeof(std::complex<double>) * N/2));
    for (int i = 0; i < (int)(N / 2); i++) {
        W_N[i] = std::complex<double>(cos(2 * pi * i / N), -sin(2 * pi *i / N));
    }
    int m = (int)round(log(N) / log(2));//蝶形层数
    for (int i=1; i <= m; i++) { //m层
        for (int j=0; j < (int)pow(2,m - i); j++) { //每一层做2^i点DFT的次数
            for (int k = 0; k < (int)pow(2,i - 1); k++) { //DFT的前半部分
                X[k + j * (int)pow(2, i) + (int)pow(2, i - 1)] = W_N[k*(int)pow(2,m-i)]* X[k + j * (int)pow(2, i) + (int)pow(2, i - 1)];
                X_temp[k] = X[k + j * (int)pow(2, i)];
```

```
                X[k + j * (int)pow(2, i)] = X[k + j * (int)pow(2, i)]+X[k + j *
(int)pow(2, i) + (int)pow(2, i - 1)];
            }
            for (int k = (int)pow(2, i - 1); k <(int)pow(2,i); k++) { //DFT的后半
部分
                X[k + j * (int)pow(2, i)] = -X[k + j * (int)pow(2, i)] +
X_temp[k - (int)pow(2, i - 1)];
            }
        }

    }
    return X;

}
```

设计思路是按照蝶形分层计算。对*N*=2^m点，共有$log_2N = m$层蝶形。最外层*i*对蝶形层数循环，第*i*层需要做2^{m-i}次2^i点的DFT。将2^i点DFT分为前后两部分计算，下标分别为0~$2^{i-1} - 1$和2^{i-1}~$2^i - 1$。

在计算前半部分时，需要将后半部分的值乘上旋转因子。每*i*层蝶形对应的旋转因子是W_N^0、$W_N^{2^{m-i}}$、$W_N^{2\times 2^{m-i}}$、$W_N^{3\times 2^{m-i}}$……共2^{i-1}个旋转因子，对应乘到后半部分做DFT的点上即可。由于后半部分计算需要用到前半部分的值，因此前半部分更新前的值需要储存起来，最多需要2^{m-1}点，因此申请$\frac{N}{2}$个点的*X_temp*用来暂存前半部分。更新前半部分的项只需要本身加上下标增加2^{i-1}对应的项。对每个*j*，更新*X[k]*下标需要偏移$j \times 2^i$。

计算后半部分只需要本身乘-1再加上之前保存的前半部分值，两者下标相差2^{i-1}。

5、DFT函数

```
std::complex<double>* DFT(complex<double>* X_DFT,double* x,int N)
{

    for (int i = 0; i < N; i++) { //N点DFT
        X_DFT[i] = std::complex<double>{ 0,0 };
        for (int j = 0; j < N; j++) { //每点进行N次复乘
            X_DFT[i] = X_DFT[i] + std::complex<double>{cos(2 * pi * i * j / N),-
sin(2 * pi * i * j / N)}*std::complex<double>{ x[j], 0 };
        }
        //cout << X_DFT[i] << endl;

    }
    return X_DFT;

}
```

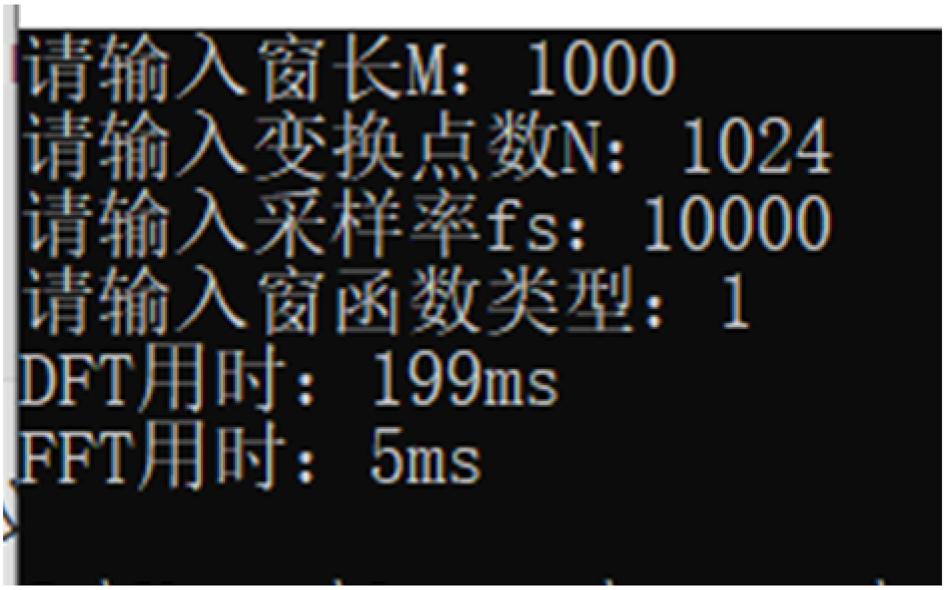
按照DFT定义直接进行计算

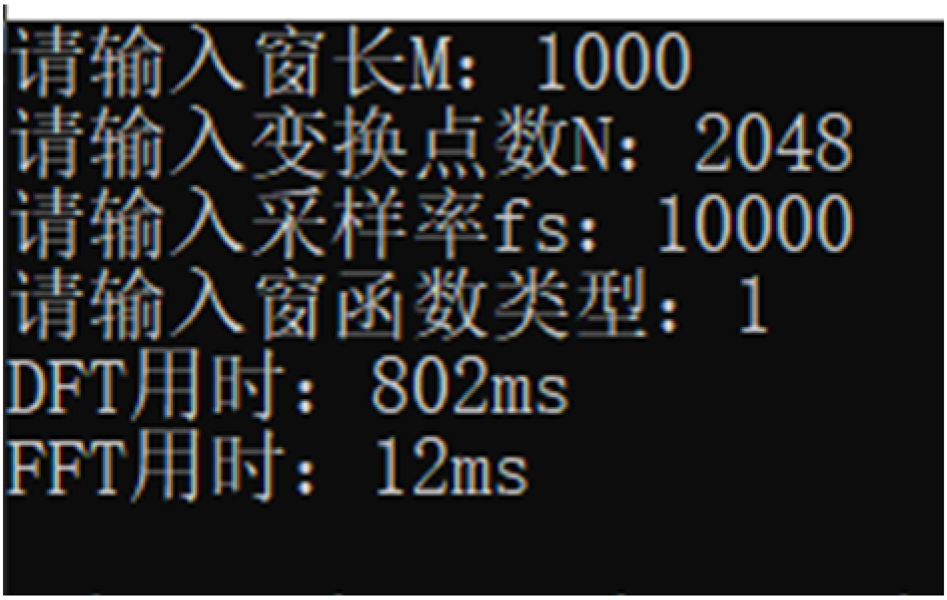
二、DFT和FFT运行时间比较

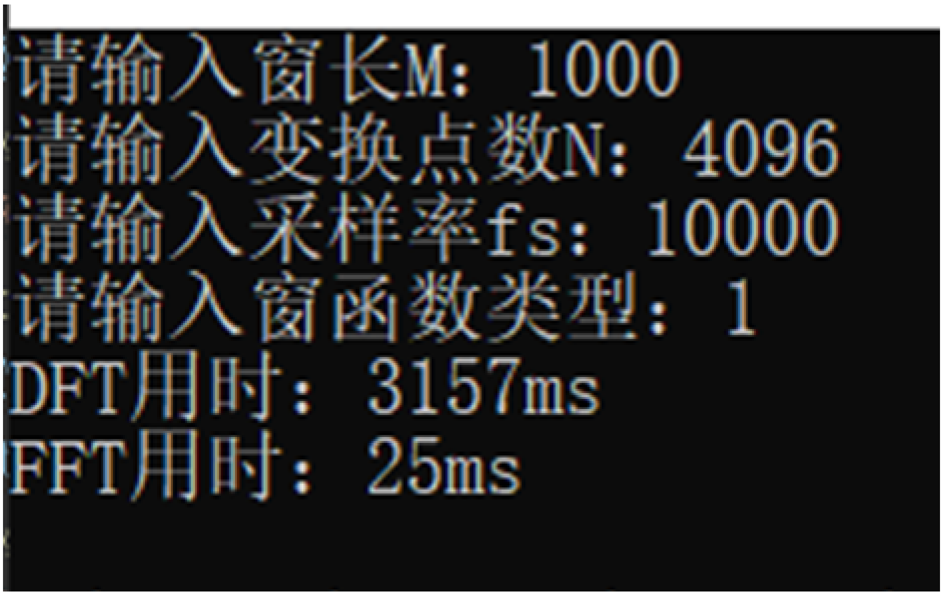
```
    start = clock();
    X_DFT=DFT(X_DFT,x,N);
    end = clock();
    cout << "DFT用时: " << 1000 * (float)(end - start) / CLOCKS_PER_SEC<< "ms" << endl;
    start = clock();
    Rader(x,X,N);//得到倒位序的x[n]
    X=FFT(X,X_temp,N);
    end = clock();
    cout << "FFT用时: " << 1000 * (float)(end - start) / CLOCKS_PER_SEC<< "ms" << endl;
```

DFT运行时间只包括对加窗补零后的序列进行DFT的过程；FFT运行时间包括得到倒位序序列和FFT计算。

1、改变变换点数N

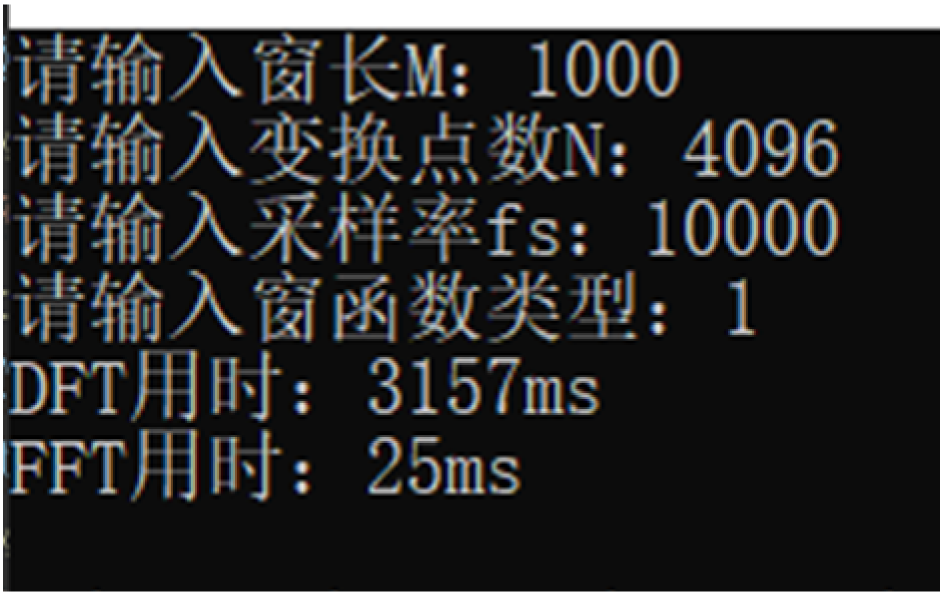


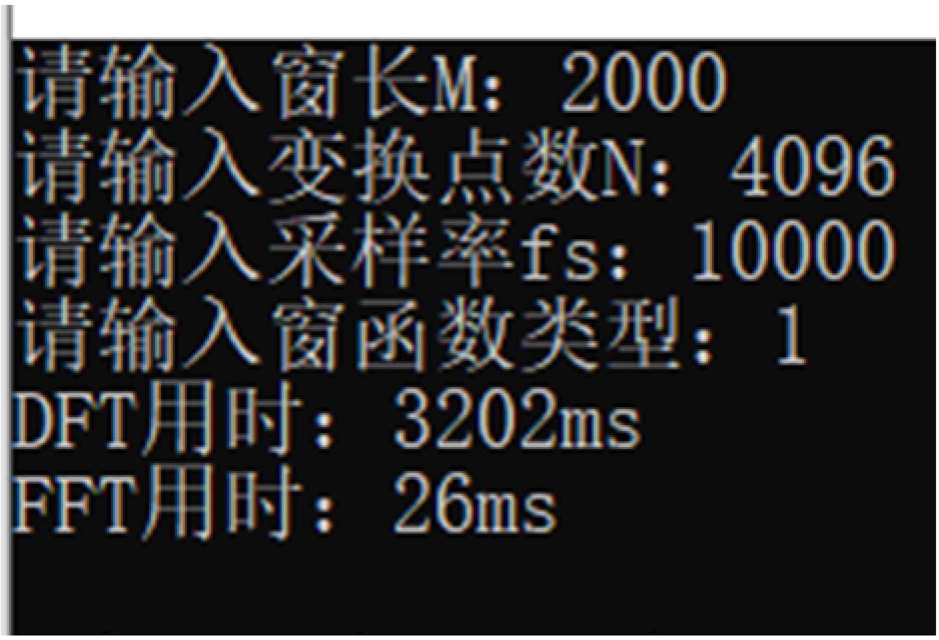


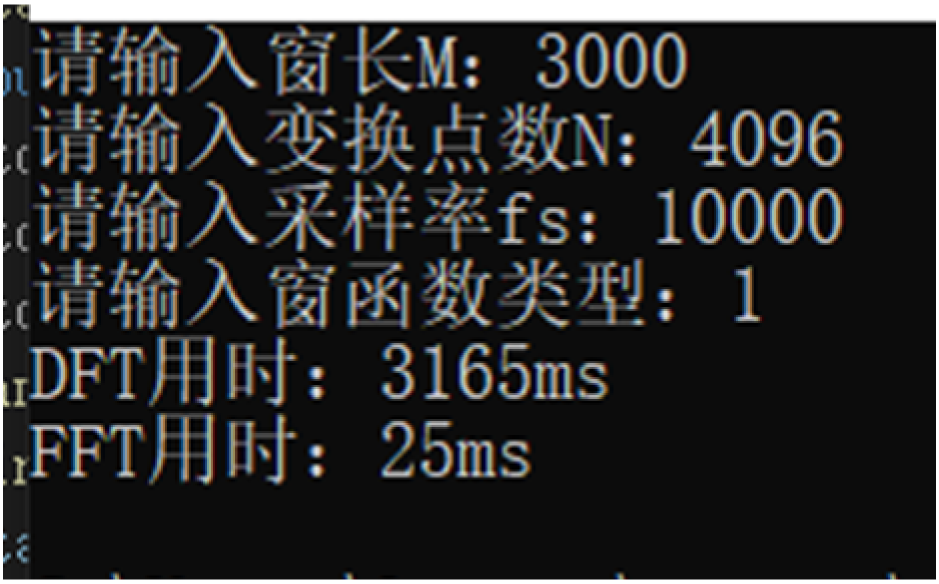


变换点数*N*对运行时间有明显影响，且*N*增加时FFT的运行时间增加量相比DFT要小得多。

2、改变窗长M

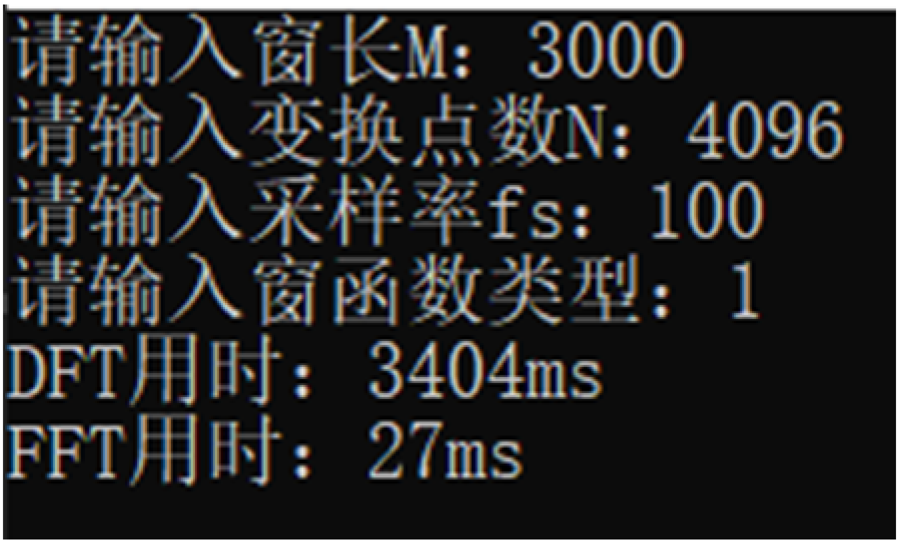


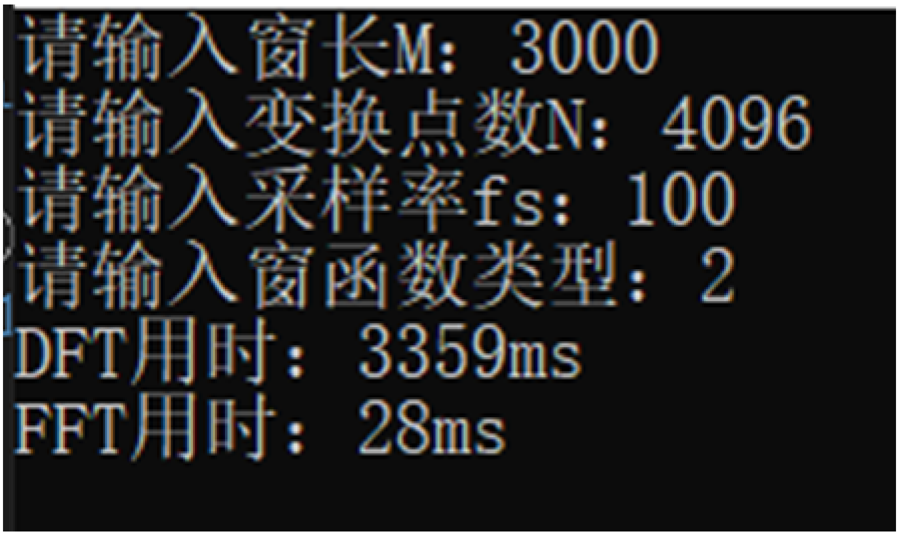


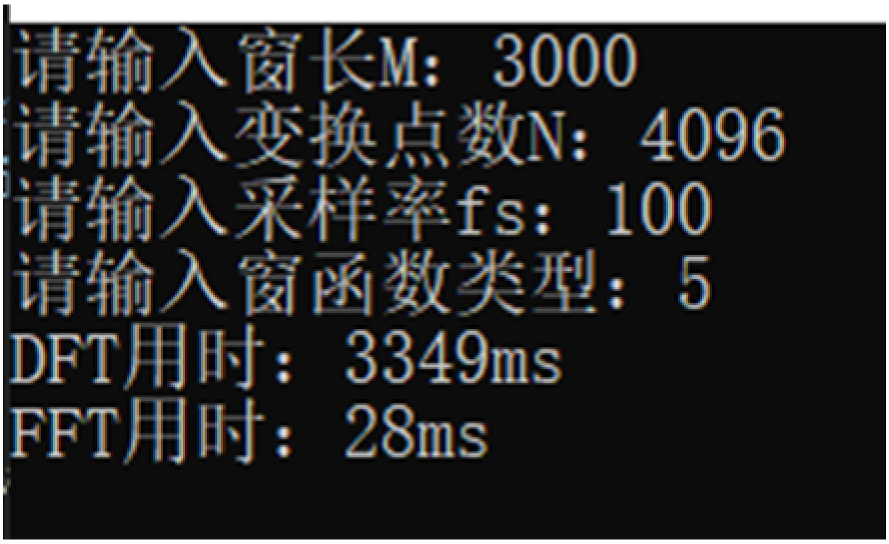


都使用矩形窗，窗长*M*对运行时间基本没有影响。

3、改变窗类型

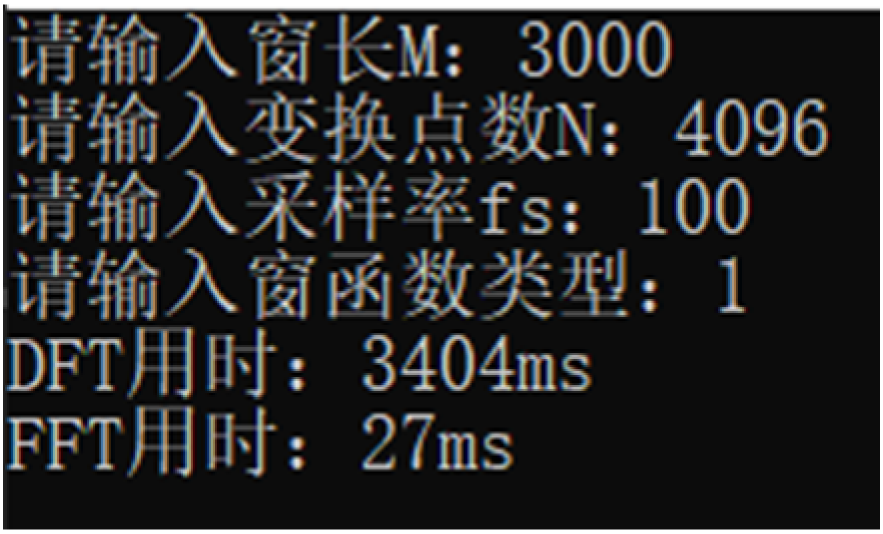


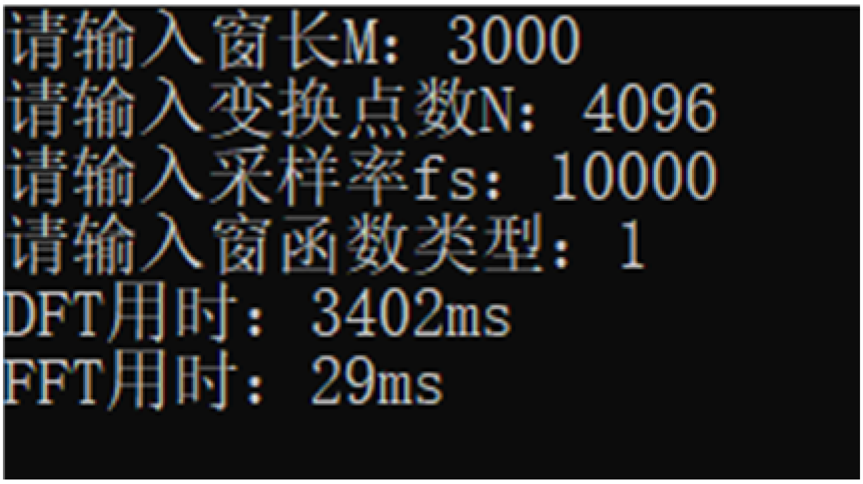




窗的类型对运行时间基本没有影响。

4、改变采样率fs





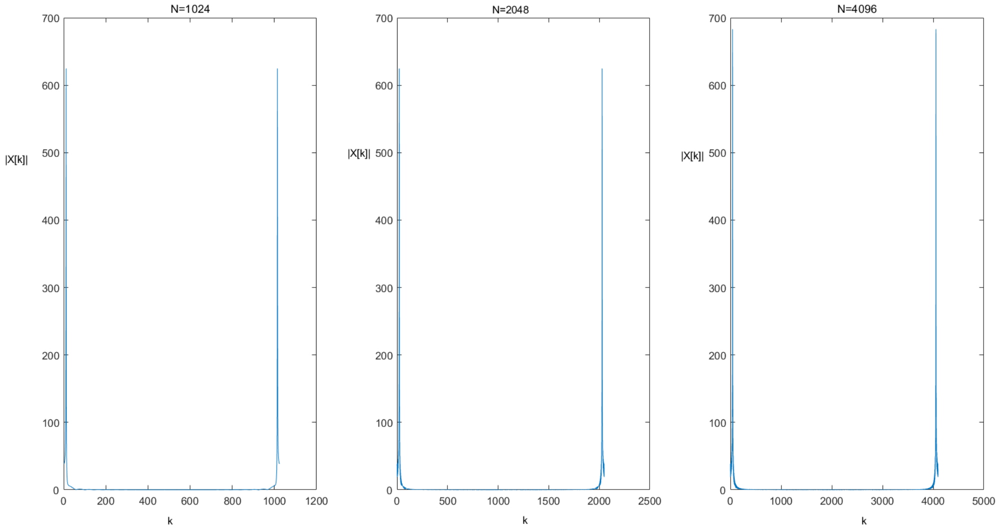
设置采样率$N = 4096$，$M = 3000$，使用矩形窗。

可以看出采样率fs对运行时间基本没有影响。

三、频谱分析

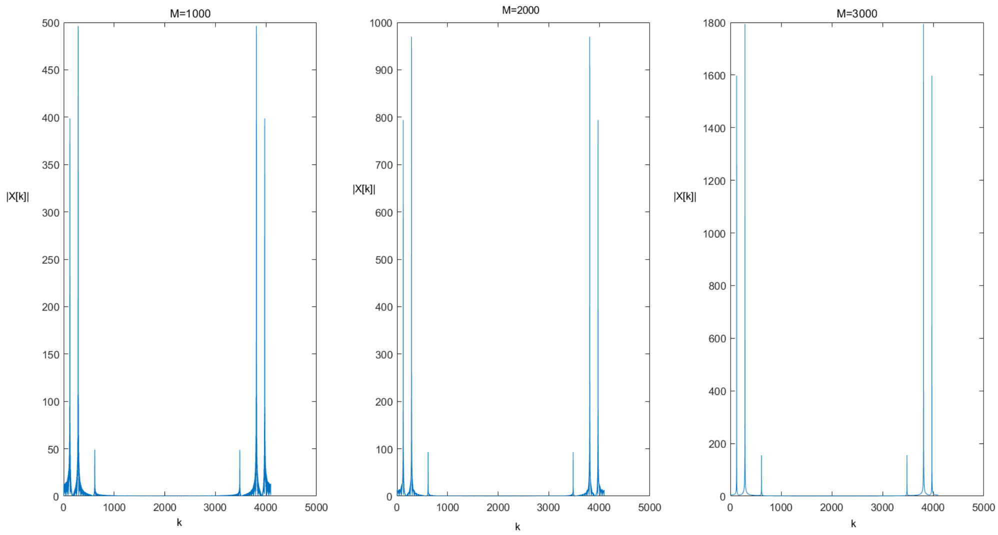
程序运行将结果输出到.csv文件中，导入matlab作图

1、改变变换点数N



变换点数增多，DFT结果更接近DTFT结果，但窗长M限制了分辨率，$\Delta f=\frac{fs}{M}=\frac{10000}{1000}=10Hz$，和信号中的最大频率差相当，频谱上基本只能看到正负频率各一个峰。

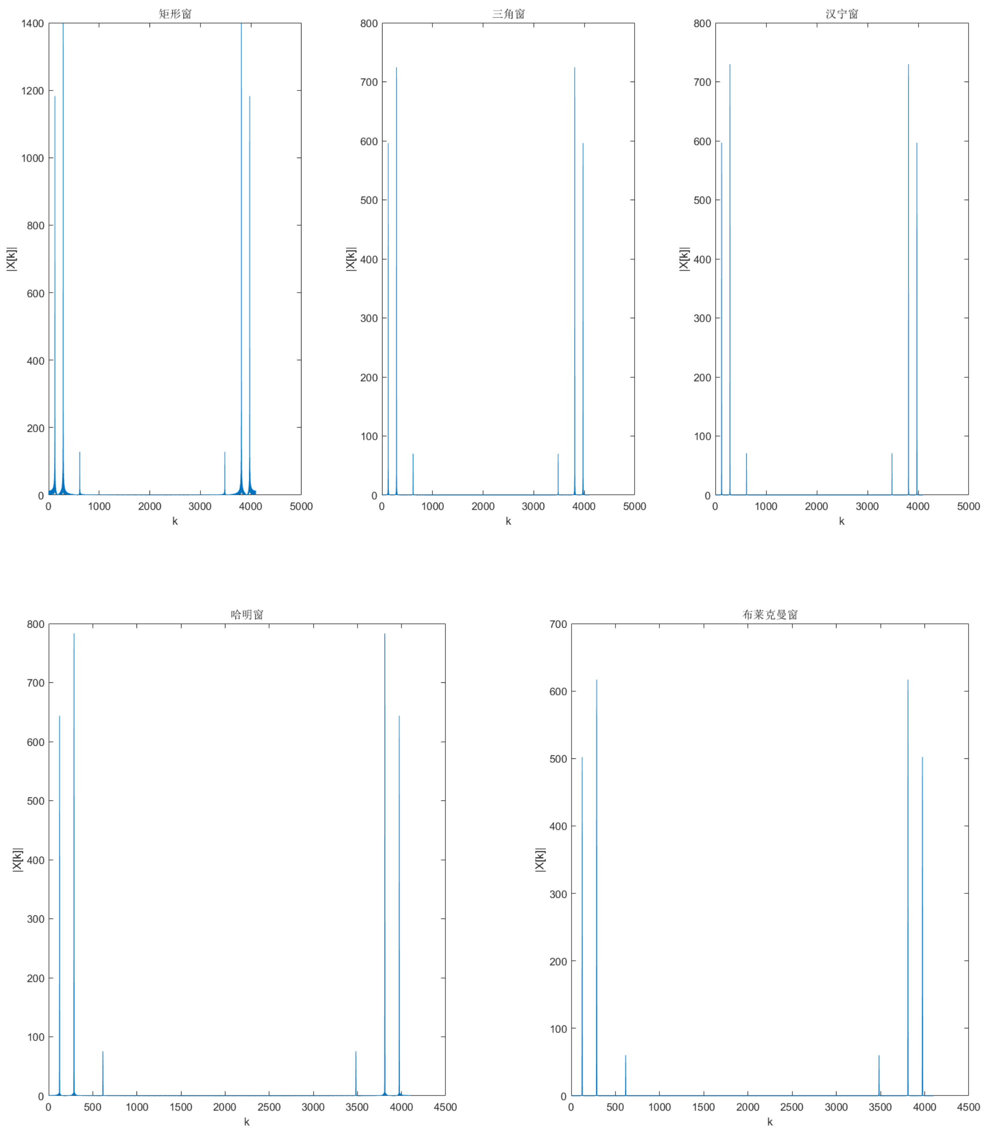
2、改变窗长M



设置采样率$fs=100$，$N=4096$，选用矩形窗函数。

可以明显分辨出正负频率各有三个峰，频率对应幅度也和原信号的幅度吻合。随M增大窗函数的主瓣宽度减小，主瓣交叠减小，反映到频谱上随M增大，原信号频率附近的假谱现象更不明显。

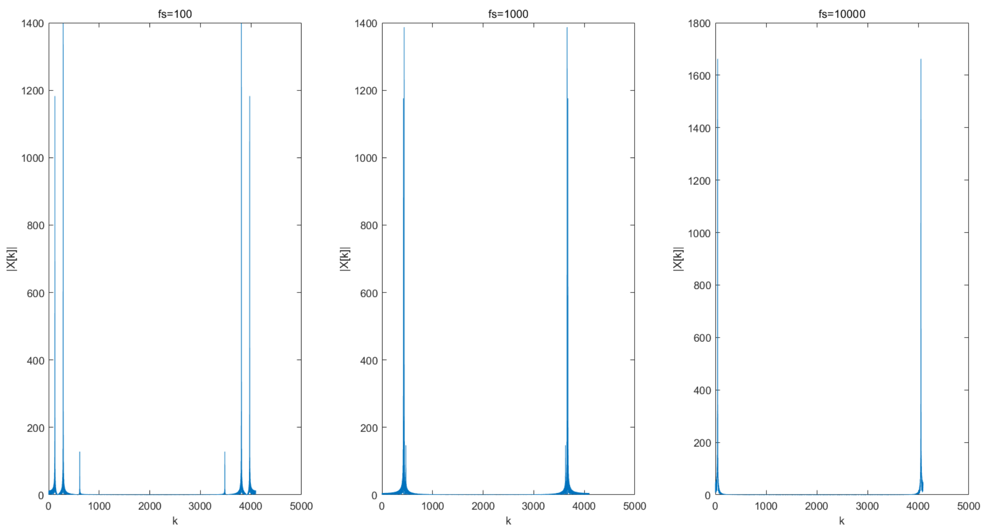
3、改变窗类型



设置采样率$fs = 100$，$N = 4096$，$M = 3000$。

可以看出矩形窗旁瓣的影响最大，三角窗、汉宁窗和哈明窗基本看不出区别，布莱克曼窗在单一频点很干净，基本看不到旁瓣电平的影响。

4、改变采样率fs



设置$N = 4096$，$M = 3000$，选用矩形窗。

由图可见，采样率fs增大，三个频率对应的峰逐渐合并在一起，fs越大越难以分辨三个峰。