广州大学学生实验报告

开课学院及实验室：机电学院 电子楼501　　2018 年 6 月 1 日,6 月 8 日

学院	机电学院	年级、专业、班	电信151	姓名	苏伟强	学号	1507400051
实验课程名称	音频DSP技术与应用					成绩	
实验项目名称	实验五 数字信号处理算法实验					指导老师	张承云

1 实验目的

掌握 FIR，IIR，FFT 滤波的 DSP 实现方法。

2 实验设备

硬件：ADSP-21489 EZ-Borad 开发板；软件：Matlab, Visual DSP++。

3 实验内容

3.1 软件仿真

软件仿真目标：输入一个音频数据，搭建数据流，生成软件仿真产生的 *FIR*，*IIR*，*FFT* 数据文件；将原始输入数据导入 *Matlab*，在 *Matlab* 中，结合 *Matlab* 自身的函数，对 *DSP* 仿真生成的 *FIR*，*IIR*，*FFT* 数据的准确性进行验证。

3.1.1 使用 Matlab 辅助生成滤波器系数

打开 *Matlab* 的 *FDAtool* 工具，出现如下图 1滤波器设计界面，选择 *IIR* 或 *FIR* 设计滤波器，注意这里需同输入的音频信号保持一致的采样频率，即 $Fs = 48000Hz$，自定义参数后，点击 *DesignFilter* 来生成滤波器，生成后可以在该界面上快捷框选择 $MagnitudeandPhaseRespenses$ 来直观查看滤波器冲击响应的幅度谱和相位谱。确认无误之后，通过 $File->Export$，选择导出 $CoefficientFile(ASCII)$，导出方式为 *Decimal*，点击 *Export* 确认导出，导出的文件为 *.fcf* 文件，导出后，*Matlab* 自动打开，如下??所示为设计一个 *FIR*，*IIR* 滤波器的导出文件，这里省略了部分内容。

FIR 滤波器导出系数文件

```
% Generated by MATLAB(R) 8.0 and the Signal Processing Toolbox 6.18.
% Generated on: 30-Jun-2018 10:02:49
% Coefficient Format: Decimal
% Discrete-Time FIR Filter (real)
%-----------------------------------
% Filter Structure  : Direct-Form FIR
% Filter Length     : 51
% Stable            : Yes
% Linear Phase      : Yes (Type 1)
```

```
10
11 Numerator:
12 -0.00091909820848023365
13 -0.0027176960266135104
14 -0.0024869527598547769
15 0.0036614383834880021
16 0.013650925230654665
17 0.017351165901097833
18 0.0076653061904350282
19 ......
```

IIR 滤波器导出系数文件

```
1 % Generated by MATLAB(R) 8.0 and the Signal Processing Toolbox 6.18.
2 % Generated on: 30-Jun-2018 10:11:23
3 % Coefficient Format: Decimal
4 % Discrete-Time IIR Filter (real)
5 %-------------------------------
6 % Filter Structure    : Direct-Form II, Second-Order Sections
7 % Number of Sections  : 16
8 % Stable              : Yes
9 % Linear Phase        : No
10
11 SOS Matrix:
12 1  2  1  1  -0.55062564838743377  0.90749950521054146
13 1  2  1  1  -0.50422398043437289  0.74675298183186856
14 1  2  1  1  -0.46559303757538822  0.61292611669194164
15 1  2  1  1  -0.43325944930284865  0.50091480045163173
16 1  2  1  1  -0.40609746380147993  0.40681915841945926
17 ......
18
19 Scale Values:
20 0.33921846420577689
21 0.31063225034937392
22 0.28683326977913837
23 ......
```

可以看到 FIR 滤波器阶数为 51 阶，并且在通带内为线性相位。IIR 滤波器为 16 阶，通带内不是线性相位。

这里需要注意的是，根据滤波器差分方程定义

$$y[n] = \sum_{i=0}^{M} b_i x[n-i] - \sum_{i=0}^{M} a_i y[n-i] \tag{1}$$

式中，N 为过去输入数，M 为过去输出数。若 $i \geq 1$ 时，$a_i = 0$，则仅有系数 b_i，描述的是一个 FIR 系统，可以看到，输出仅与输入相关，在系数文件中可以看到，的确是只有 b 参数，即 $Numerator$。若 $a_i = 0$，则滤波器系数包含了 a_i a_i，描述的是一个 IIR 系统，可以看到，输出不仅与输入有关，而且与过去的输出有关，在系数文件中可以看到其表示为直接 II 型，二阶节形式。

查看 $VisualDSP++$ 帮助，可以看到到在阐述 FIR 滤波器的时候，有以下说明：“the coefficients must be stored in reverse order in the array *coeffs*”，也就是说，从 $Matlab$ 生成的 FIR 滤波器参数，必须倒序放置进 DSP 程序中 FIR 滤波的参数数组 $coeffs$。在阐述 *biquad* 滤波器的时候也有相似的描述：“The number of biquad sections is specified by the parameter *sections*, and each biquad section is represented by five coefficients A1, A2, B0, B1, and B2. The biquad functions assume that the value of

A0 is 1.0, and A1 and A2 should be scaled accordingly”，即在 *DSP* 程序中，变量 *section* 描述的是 *IIR* 滤波器的二阶节数，每个二阶节形式通过 A1, A2, B0, B1, and B2 来表达，其中 A0 已经默认为 1，所以我们无需将 *Matlab* 生成的 A0 参数导入该系数数组。同时注意到以下描述：“For the vector versions of the biquad function, the five coefficients must be stored in the order: A2, A1, B2, B1, B0 ”，即同 *FIR* 滤波一样，每个二阶节的参数必须的反序存入.

然而，根据文档说明，无论是 *FIR* 还是 *IIR* 滤波，都需要将参数数组 $coeffs$ 保存在程序存储空间 *pm* 中。

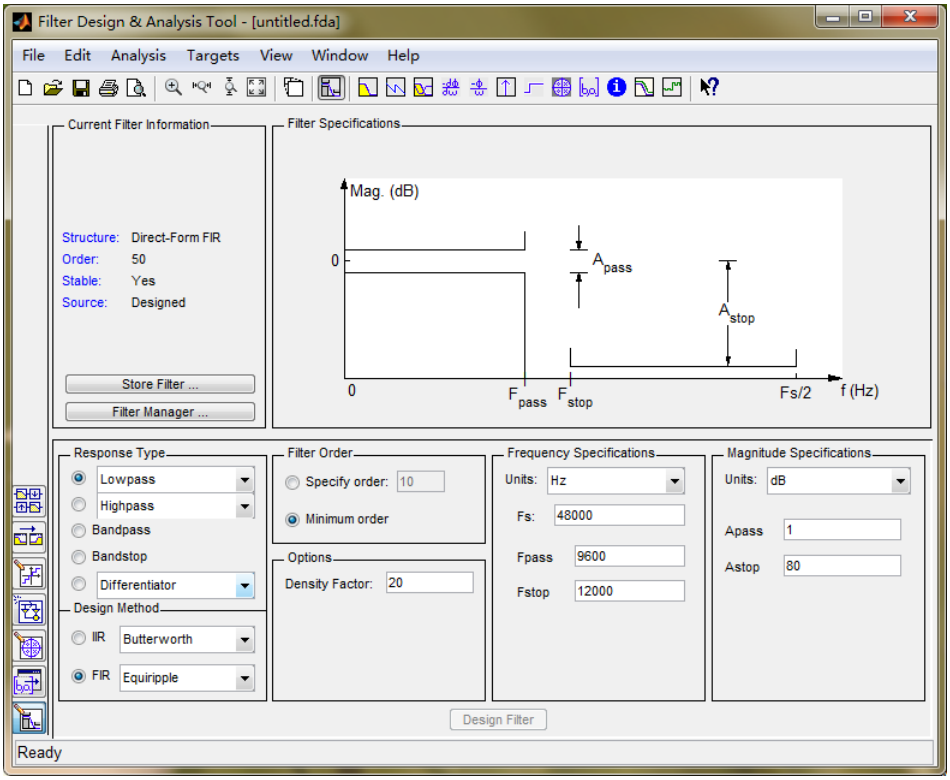

图 1: fdatool

3.1.2 FIR IIR FFT IFFT 频域滤波重叠相加法的 DSP 实现

在 *VisualDSP++* 帮助文档中搜索，可以看到 *FIR*，*IIR*，*FFT* 的实现方法，注意事项以及相关例程说明。

FIR 滤波器使用库函数，如下所示，使用 *___SIMDSHARC___* 模式。其中参数描述已经注释。

FIR 实现方法

```
/*Finite impulse response (FIR) filter*/
#include <filters.h>
float *fir (const float  dm input[],//输入浮点数据
            float        dm output[], //处理输出浮点数据
            const float  pm coeffs[], //FIR参数数组，放在程序存储空间中
            float        dm state[],  //处理过程中间状态，仅需初始化一次
            int             samples,  //输入输出数据数
            int             taps);    //FIR参数个数
```

以下是 *IIR* 的实现方法，这里使用二阶节级联的方式，直接可以调用 *biquad* 完成，代码如下

IIR 实现方法

```
#include <filter.h>
float *biquad (const float   dm input[],    //输入浮点数据
               float         dm output[],   //输出浮点数据
               const float   pm coeffs[],   //IIR参数数组，放在程序存储空间中
               float         dm state[],    //处理过程中间状态，仅需初始化一次
               int             samples,     //输入输出数据数
               int             sections);   //IIR含二阶节个数
```

以下是 *FFT, IFFT* 实现方法，调用库函数 *cff, ifft*。需要注意的是，这里都是使用复数输入，并且结果以复数形式返回，我们需要将输入音频放在复数的实部。非常方便的是 *DSP* 已经将其定义为新的结构体数据类型 *complex_float*，通过 *complex_float.re*，*complex_float.im* 可以访问复数的实部和虚部。

FFT，IFFT 实现方法

```
#include <filter.h>
/*fft*/
void cfft (const complex_float in[];     //输入复数数据
           complex_float t[];            //临时中间变量
           complex_float out[];          //输出浮点数据
           const complex_float w[];      //旋转表
           int wst;
           int n);                       //采样点数

/*ifft*/
complex_float  ifft (complex_float dm input[],//输入复数数据
                     complex_float dm temp[],  //临时中间变量
                     complex_float dm output[], //临时中间变量
                     const complex_float pm twiddle[],//旋转表
                     int twiddle_stride,
                     int n );                  //采样点数
```

以下介绍基于重叠相加法的 *FIR* 滤波。首先介绍其原理。有限长序列的 *DFT* 有一个重要的性质，两个序列 N 点的循环卷积的结果为他们各自的 *DFT* 序列的乘积的 *IDFT*，即

$$x_1[k] \textcircled{N} x_2[k] = IDFT\{X_1[m]X_2[m]\} \tag{2}$$

我们知道，*DFT* 和 *IDFT* 有快速算法，即 *FFT* 和 *IFFT*。由式 2可知，我们可以通过计算两个序列的 *DFT* 乘积的 *DFT* 逆变换来计算两个序列的循环卷积，则会加快循环卷积计算的速度，起到优化算法的作用。然而我们又知道当循环卷积的长度 $N = L \geq N_1 + N_2 - 1$ 的时候，循环卷积的结果与线性卷积的结果一致，即

$$x_1[k] * x_2[k] = x_1[k] \textcircled{L} x_2[k] \qquad L \geq N_1 + N_2 - 1 \tag{3}$$

进一步的我们得到结论：计算两个序列的 *DFT* 乘积的 *DFT* 逆变换来计算两个序列的线性卷积。我们知道一切的滤波其实就是信号长序列与滤波器短序列线性卷积结果的重叠相加，通过上述方法，我们可以加速这个滤波的过程。我们将 $x_1[k], x_2[k]$ 代替为 $x[k], h[k]$ 以表示数字信号的滤波过程，具体流程如下图 2

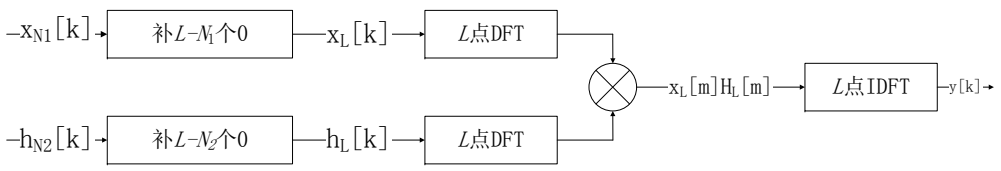

图 2: 频域滤波流程图

将所有输入序列分段进行以上过程得到 $y_1[k]$，$y_2[k]$...，。由于数字信号处理过程中信号具有连续的输入性质，卷积部分会有重叠，而我们进行单段的卷积的时候将重叠部分视为 0，为了得到准确的输出，需进行如下图 3重叠相加，可以得到最终正确的滤波结果。

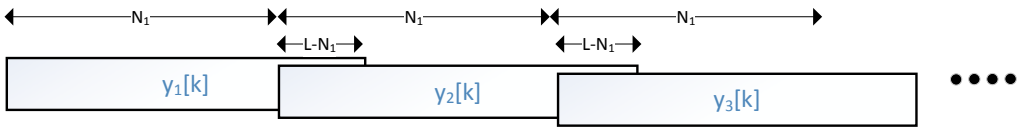

图 3: 重叠相加法示意图

算法先在 $Matlab$ 中实现，代码见附录，然后将算法改写为 DSP，见如下代码，包括一个初始化函数和实现函数

重叠相加法频域滤波

```
complex_float pm fir_ifft_twiddle[FFT_FIR_LEN/2]; //FFT旋转表
complex_float pm fir_fft_twiddle[FFT_FIR_LEN/2];  //IFFT旋转表


complex_float FIR_FFT_FREQZONE[FFT_FIR_LEN];     //滤波器频域序列

int twiddle_stride = 1;

complex_float FIR_FFT_Input[FFT_FIR_LEN];         //输入缓冲区
complex_float FIR_FFT_Output[FFT_FIR_LEN];        //输出缓冲区(for fft)
complex_float FIR_IFFT_Output[FFT_FIR_LEN];       //输出缓冲(for ifft)

complex_float LastResult[FFT_FIR_LEN];        //上一次的计算结果

void FIR_BY_FFT_INIT()
{
        int index = 0;

        twidfft(fir_fft_twiddle, FFT_FIR_LEN);  //用于计算FIR滤波器序列FFT的旋转表
        twidfft(fir_ifft_twiddle, FFT_FIR_LEN);

        memset(LastResult,0,sizeof(complex_float) * FFT_FIR_LEN);  //将上次计算结果清零

        complex_float temp[FFT_FIR_LEN];  //FFT计算临时变量
        complex_float FIR_FFT_TIMEZONE[FFT_FIR_LEN];//定义复数数组用来存放FIR滤波器序列

        memset(FIR_FFT_TIMEZONE,0,sizeof(complex_float) * FFT_FIR_LEN); //清零FIR序列
        for(index=0; index<TAPS; index++)   //为FIR滤波器序列赋值，为被赋值的为0
        {
                FIR_FFT_TIMEZONE[index].re = fir_coeffs[index];
                FIR_FFT_TIMEZONE[index].im = 0;
        }

//计算FIR滤波器的频域序列
        cfft(FIR_FFT_TIMEZONE, temp, FIR_FFT_FREQZONE,
                fir_fft_twiddle, twiddle_stride, FFT_FIR_LEN);
}

void FIR_BY_FFT(complex_float* input,complex_float* output,int sample_len)
{
```

```
41          int index  = 0;
42
43          complex_float temp[FFT_FIR_LEN];   //FFT计算临时变量
44
45          //初始化临时数组，输入信号（时域），输入信号(频域)，输出信号（时域）为0
46          memset(temp,0,sizeof(complex_float)*FFT_FIR_LEN);
47          memset(FIR_FFT_Input,0,sizeof(complex_float)*FFT_FIR_LEN);
48          memset(FIR_FFT_Output,0,sizeof(complex_float)*FFT_FIR_LEN);
49          memset(FIR_IFFT_Output,0,sizeof(complex_float)*FFT_FIR_LEN);
50
51          //为输入信号（时域）赋值，未赋值的为0
52          for(index=0; index<sample_len; index++)
53          {
54                  FIR_FFT_Input[index].re = input[index].re;
55          }
56
57          //计算输入信号(频域)
58          cfft(FIR_FFT_Input, temp, FIR_FFT_Output,
59                   fir_fft_twiddle, twiddle_stride, FFT_FIR_LEN);
60
61          //计算输入信号（频域）与FIR滤波器序列（频域）的乘积
62          for(index=0; index<FFT_FIR_LEN; index++)
63          {
64                  FIR_FFT_Output[index].re = FIR_FFT_FREQZONE[index].re *
65                                  FIR_FFT_Output[index].re;
66          }
67
68          //乘积的结果进行逆变换
69          ifft(FIR_FFT_Output, temp, FIR_IFFT_Output,
70                          fir_ifft_twiddle, twiddle_stride, FFT_FIR_LEN);
71
72          //重叠相加
73          for(index = 0; index<sample_len; index++)
74          {
75                  output[index].re = FIR_IFFT_Output[index].re + LastResult[TAPS+index].re;
76          }
77          //存储本次的计算结果到LastResult
78          memcpy(LastResult,FIR_IFFT_Output,FFT_FIR_LEN*sizeof(complex_float));
79  }
```

3.1.3 搭建仿真数据流

为了方便验证相位谱，这里输入的仿真测试数据由 4 个正弦波组成，其中 2 个在通带范围内，两个在阻带范围。使用 *Matlab* 来生成这个 *.dat* 数据，程序如附录所示。在 DSP 仿真中搭建 4 个数据流，如图5-8所示，输入仿真数据 sinDAT.dat，将 *FFT,FIR,IIR* 结果保存到 outFFT.dat，outFIR.dat，outIIR.dat 中，以方便数据读取验证。

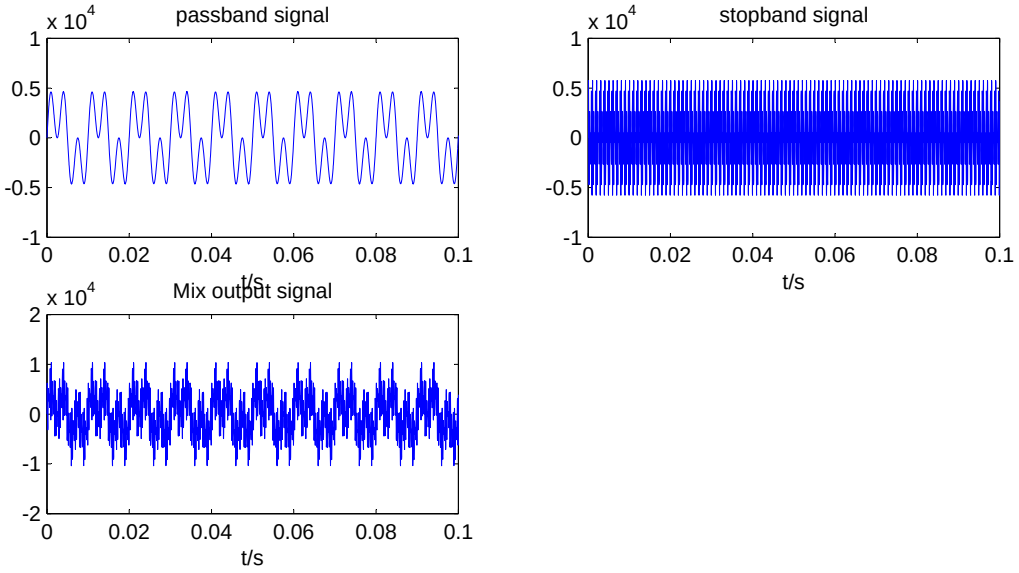

图 4: 仿真输入测试数据，4 个正弦波叠加，2 个通带，2 个阻带

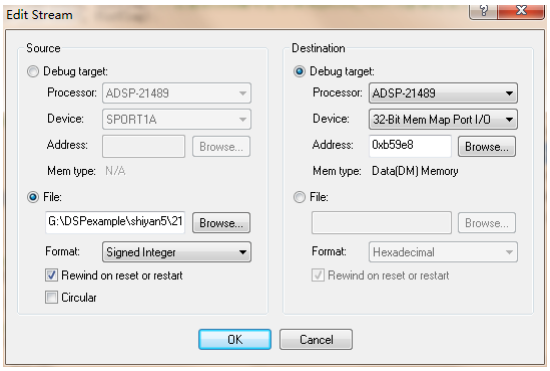

图 5: sinDAT.dat->DataIn

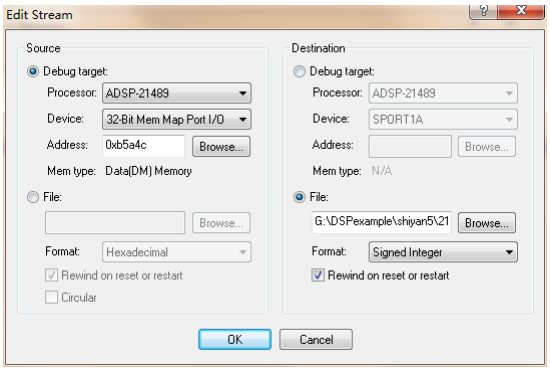

图 6: DataOutFFT->outFFT.dat

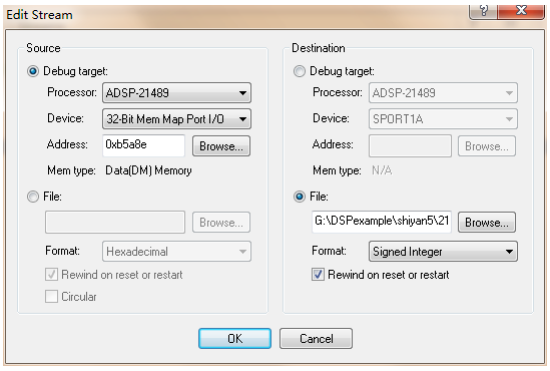

图 7: DataOutFIR->outFIR.dat

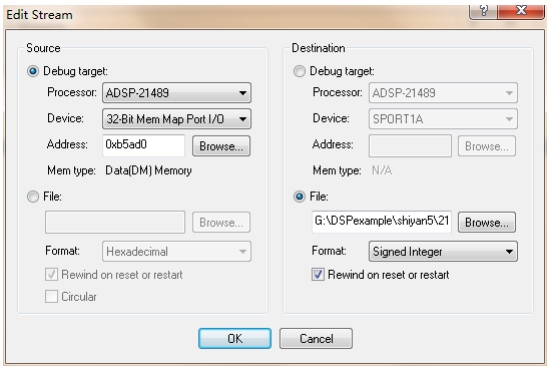

图 8: DataOutIIR->outIIR.dat

3.1.4 在 Matlab 中对仿真数据进行验证

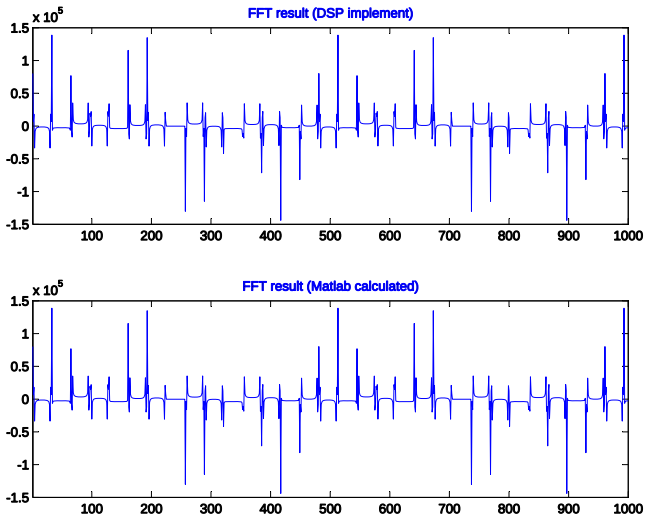

图 9: 验证 FFT 数据

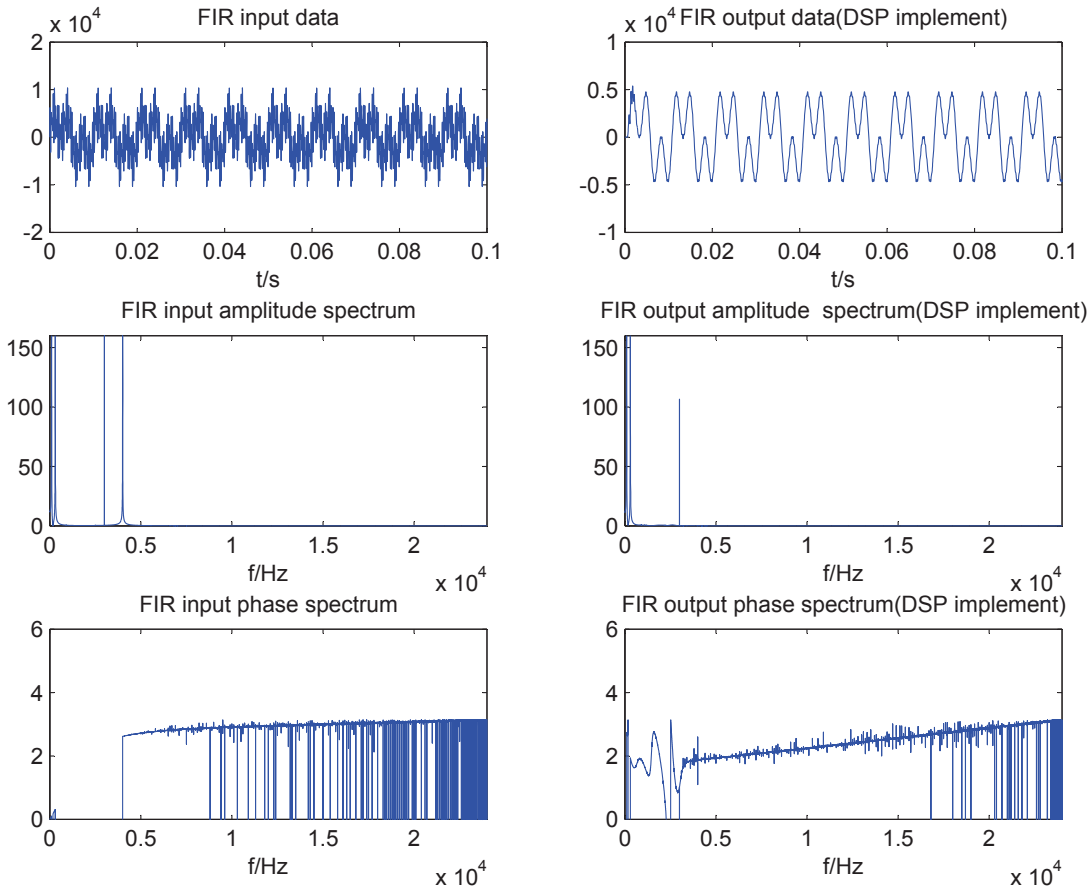

图 10: 验证 FIR 滤波结果

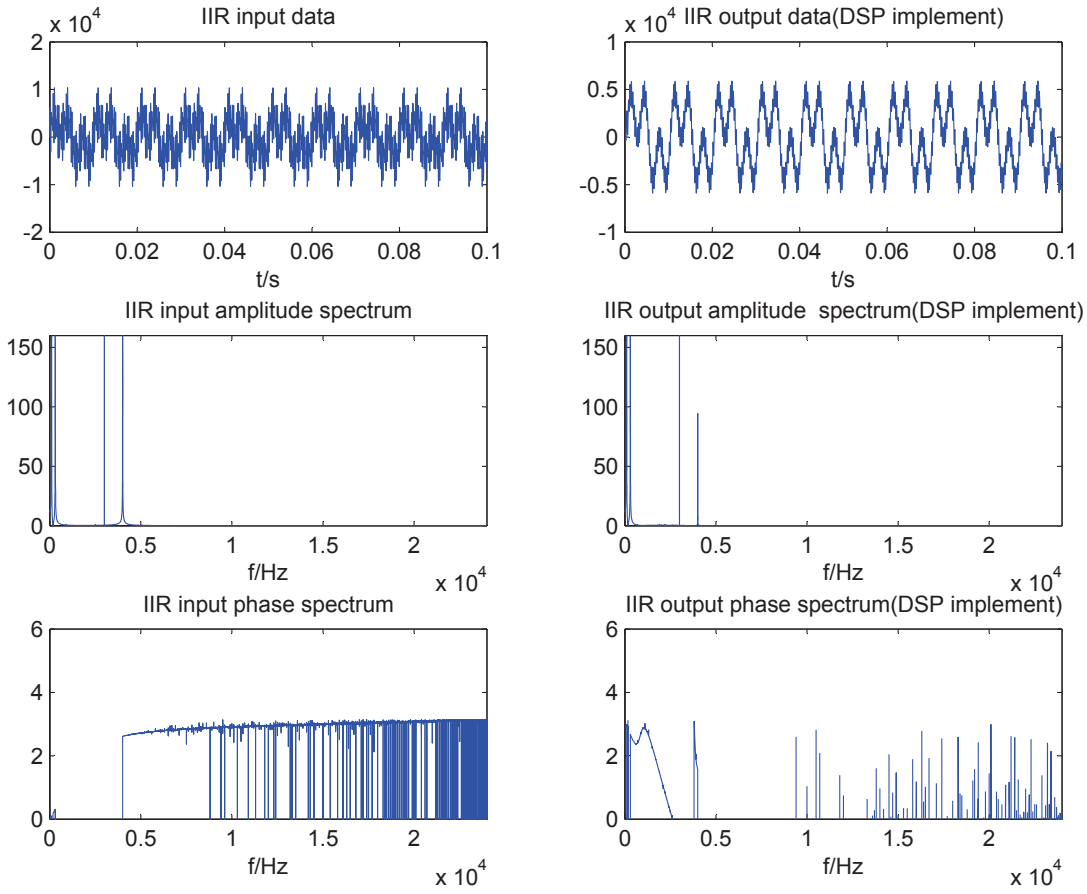

图 11: 验证 IIR 滤波结果

在 *Matlab* 中 *load* 前述 *DSP* 仿真输出结果文件 outFFT.dat，outFIR.dat，outIIR.dat，对仿真数据进行验证，代码见附录。如图 9为 FFT 数据的验证图，这里为了方便直观查看，只查看全部 65536 个点的前 1000 个点，通过调用 *Matlab* 函数 fft 并且输出结果，将该结果与仿真结果进行比较，可以看到仿真 FFT 数据还是比较准确的。如图 10，11所示为 FIR 和 IIR 滤波结果的验证，从幅度谱可以看到，两者都完成了低通滤波，但是相位谱有所区别，为了方便观察，这里将信号的范围限制在 $0-0.1s$ 时间范围内，可以看到，FIR 滤波器，将 4所示输入信号的高频部分滤除了，保留了低频部分的两个正弦波的叠加和，并且，信号的形状未发生畸变，这说明 FIR 滤波器在通带范围内的相位特性是线性的，反观图 11的 IIR 验证结果，其输出信号即使也保留了低频的两个正弦波的叠加和，但是其通带信号的相位产生了非线性抖动，信号有畸变，其通带内并未具备线性相位。

此外，通过查看 FIR 仿真输出文件与原始输入数据文件，发现其相差了 20 个数据的延时，时延为 $\frac{20}{fs}=\frac{20}{48000}=0.0083s$，理论时延为 $\frac{M-1}{2fs}=\frac{41-1}{2*48000}=0.0083s$(M 为滤波器阶数)，实际和理论相符，$FIR$ 滤波产生了 $0.0083s$ 的时延

3.2 硬件实现

硬件实现软件仿真的结果，讲原仿真的程序 *main* 函数改名为 *debugMain*，新建一个 *main* 函数，选择硬件仿真会话 EZ-KIT Lite via Debug Agent。硬件仿真程序主要完成以下功能，默认状态为 FFT 后 $IFFT$ 输出；按下按键 9，LED7 亮，实现 FIR 滤波，按下按键 8，LED8 亮，实现 IIR 滤波；同时按

下按键 8, 9，LED7, 8 都亮，实现频域滤波重叠相 FIR。

需要注意的是，实验中出现了输出音频噪声的问题，这是因为在进行滤波的时候共用了系数的原因，一个通道必须有一组系数。为了解决这个问题，只处理一个输入声道，将结果输出到两个声道。以下为实现部分的代码，通过全局的标志位来控制模式切换

blockProcess_audio.c

```
void memcopy(float *input, float *output, unsigned int number)
{
        int i,j,k;
        complex_float cplx_input[N_FFT];
        complex_float cplx_fftoutput[N_FFT];
        complex_float cplx_output[N_FFT];
        for(j = 0;j<N_FFT;j++)
        {
                cplx_input[j].re = input[j];
                cplx_input[j].im = 0;
        }
        if (firflag)//FIR滤波
        {
                FIR_Filter_M(input,output,number);
        }
        else if(iirflag)//IIR滤波
        {
                IIR_Filter_M(input,output,number);
        }
        else if(iirflag&&firflag)//重叠相加法FIR
        {
                FIR_BY_FFT(cplx_input, cplx_output,number);
                for(k = 0; k<N_FFT; k++)
                {
                        output[k] = cplx_output[k].re;
                }
        }
        //默认为FFT后IFFT输出
        else
        {
                FFT_M(cplx_input,cplx_fftoutput);
                IFFT_M(cplx_fftoutput,cplx_output);
                for(k = 0; k<N_FFT; k++)
                {
                        output[k] = cplx_output[k].re;
                }
        }
}
```

4 实验总结与体会

在验证 FIR 滤波器通带的线性相位的时候，一开始使用的是白噪声序列，这样虽然方便查看幅度滤波效果，但是相位谱很难看出通带线性相位，波形的变化很难对比，为此花了很长的时间进行试验和思考，最后在老师的指导下使用四个正弦波的叠加（2 个通带内，两个阻带内）进行仿真信号的输入，经过滤波后只剩下 2 个通带正弦波，这个时候不观察其相位谱，直接观察其滤波输出后的信号的形状，即可

判断其在通带内是否是线性相位的。若是线性相位，则与原输入信号通带内的信号对比，信号的波形没有变化，只是有群延迟，若不是线性相位，则信号的波形会因为相位的非线性抖动发生干扰畸变。这种方法很巧妙！

在进行硬件仿真进行滤波的时候，出现了输出音频噪声的问题，这是因为在进行滤波的时候共用了系数的原因，一个通道必须有一组系数。为了解决这个问题，只处理一个输入声道，将结果输出到两个声道，牺牲了一个声道的信息。

大部分结论与体会已经在实验过程部分给出，这里不做赘述。

5 实验完成后实验器材照片

图 12: 实验完成后器材

附录代码

Matlab 生成 4 个正弦波的仿真输入信号

```
close all;clc;clear;
N = 65536;
fs = 48000;
tfinal = (N-1)/fs;
t = 0:1/fs:tfinal;
anp = 3000;
figure;

%通带正弦波
pass1 = anp*sin(2*100*pi*t);
pass2 = anp*sin(2*300*pi*t);
```

```
pass = pass1 + pass2;
subplot(321);
plot(t,pass);xlim([0 0.1]);
title('passband signal');
ylim([-10000 10000]);
xlabel('t/s');

%阻带正弦波
stop1 = anp*sin(2*3000*pi*t);
stop2 = anp*sin(2*4000*pi*t);
stop = stop1 + stop2;
subplot(322);
plot(t,stop);xlim([0 0.1]);
title('stopband signal');
ylim([-10000 10000]);
xlabel('t/s');

%混合输出
in = floor(pass + stop);
subplot(323);
plot(t,in);xlim([0 0.1]);
title('Mix output signal');
ylim([-20000 20000]);
xlabel('t/s');
in = in';
save('sinin.dat', 'in');
```

在 Matlab 中编写频率滤波算法

```
function [ykPeriod_M] = useFFTtoFilter (xk, hk, N1,N2)
L = N1 + N2 -1;
pow = 1;
M = 0;
while M < L
M = 2.^pow;
pow = pow + 1;
end
xk_add0 = [xk , zeros(1 ,M-N1)];
hk_add0 = [hk , zeros(1 ,M-N2)];
Xm = fft(xk_add0 , M);
Hm = fft(hk_add0 , M);
Ym = Xm.*Hm;
ykPeriod_M = ifft(Ym);
ykPeriod_M = ykPeriod_M(1:L);
end
```

Matlab 中验证仿真结果

```
clc;close all;clear;
fs=48000;%采样频率
load('outFIR.dat');
load('outIIR.dat');
load('outFFT.dat');
load('sinDAT.dat');
in = sinDAT;
% FFT验证部分
```

```
rein = reshape(in, 32, 2048);
fftall = [];
for i = 1:2048
        temp = real(fft(rein(:,i)'));
        fftall = [fftall temp];
end

figure;
subplot(211);plot(outFFT);
xlim([1 1000]);
title('FFT result (DSP implement)');
subplot(212);plot(fftall);
title('FFT result (Matlab calculated)');
xlim([1 1000]);

% in = in(1:32);
%FIR验证部分
N1=length(in)-1;
t1=0:1/fs:N1/fs;
f1=[-N1/2:N1/2]*fs/N1;
Fin=fftshift(fft(in));
figure;
subplot(3,2,1);
plot(t1,in);
xlim([0 0.1]);
ylim([-20000 20000]);
xlabel('t/s');
title('FIR input data');


subplot(3,2,2);
N2=length(outFIR)-1;
t2=0:1/fs:N2/fs;
f2=[-N2/2:N2/2]*fs/N2;
Fout=fftshift(fft(outFIR));
plot(t2,outFIR);
xlim([0 0.1]);
ylim([-10000 10000]);
title('FIR output data(DSP implement)');
xlabel('t/s');


subplot(3,2,3);
plot(f1,abs(Fin)/fs);
title('FIR input amplitude spectrum');
ylim([0 160]);xlim([0 24000]);xlabel('f/Hz');
subplot(3,2,4);
plot(f2,abs(Fout)/fs);
title('FIR output amplitude  spectrum(DSP implement)');
ylim([0 160]);xlim([0 24000]);xlabel('f/Hz');


subplot(3,2,5);
plot(f1,angle(Fin));
title('FIR input phase spectrum');
ylim([0 6]);xlim([0 24000]);xlabel('f/Hz');
subplot(3,2,6);
```

```
66 plot(f2,angle(Fout));
67 title('FIR output phase spectrum(DSP implement)');
68 ylim([0 6]);xlim([0 24000]);xlabel('f/Hz');
69
70 %IIR验证部分
71
72 figure;
73 subplot(3,2,1);
74 plot(t1,in);
75 xlim([0 0.1]);
76 ylim([-20000 20000]);
77 xlabel('t/s');
78 title('IIR input data');
79
80 subplot(3,2,2);
81 N2=length(outIIR)-1;
82 t2=0:1/fs:N2/fs;
83 f2=[-N2/2:N2/2]*fs/N2;
84 Fout=fftshift(fft(outIIR));
85 plot(t2,outIIR);
86 xlim([0 0.1]);
87 ylim([-10000 10000]);
88 title('IIR output data(DSP implement)');
89 xlabel('t/s');
90
91
92 subplot(3,2,3);
93 plot(f1,abs(Fin)/fs);
94 title('IIR input amplitude spectrum');
95 ylim([0 160]);xlim([0 24000]);xlabel('f/Hz');
96 subplot(3,2,4);
97 plot(f2,abs(Fout)/fs);
98 title('IIR output amplitude  spectrum(DSP implement)');
99 ylim([0 160]);xlim([0 24000]);xlabel('f/Hz');
100
101 subplot(3,2,5);
102 plot(f1,angle(Fin));
103 title('IIR input phase spectrum');
104 ylim([0 6]);xlim([0 24000]);xlabel('f/Hz');
105 subplot(3,2,6);
106 plot(f2,angle(Fout));
107 title('IIR output phase spectrum(DSP implement)');
108 ylim([0 6]);xlim([0 24000]);xlabel('f/Hz');
```