# Part A

**Q1**

**a).**

Divide the long sequence x[n] into arbitrary length segment (consider the length is L), use the divided segment to calculate the circular convolution with filter h(n), then get the output y[n] by adding the results of convolution of these segments and filter.

**Q2**

**a).**

As

Hence,

Based on the process of proof, it can be concluded that after we get the results of DFT of x[n] (which is X[k]), we calculate the DFT of X[k]. When flipping the DFT of X[k] in the left-right direction, calculate the N-point circular shift. Multiplying the , then we get the initial sequence x[n].

**Q3**

**a).**

if X[k] is conjugate symmetric, and x[n] = IDFT{X[k]} is real.

Consider

then

Hence

As x[n] is real,

So

**b).**

if substitute k with n, then we get

so

if substitute n with k, calculate the conjugate of

as X[k] is conjugate symmetric, hence

Because

it can prove that is conjugate symmetric.

Similarly, if we substitute k with n in formula , then we can get

as

hence

So,

As X[k] is conjugate symmetric,

is conjugate anti-symmetric.

**c).**

As shown in the title, we know

Q[k]=, q[n]=IDFT{Q[k]};

Hence,.

Similarly,

As X[N-n]=-X[n]

=2x[2n+1]

Hence, .

**d).**

Suppose there is a real sequence x[n] with a length of 2N, and we know the DFT of it which is X[k].

Let

Then

By these two formulas we can compute the initial sequence x[n].

**e).**

If the order of filter is M and the block size is N

The total complex multiplication is .

So computations per output data point equals to , and this can be approximated to be according to the Proakis textbook.

# Part B

**Task 1.**

**a).**

the two functions are shown below:

iffta(x);

function x = iffta(X)

    X\_conj = conj(X);

    x\_conj = fft(X\_conj);

    x = conj(x\_conj);

    x = x / length(X);

end

ifftb(x);

function x = ifftb(X)

   y = fft(X);

   y = y / length(X);

   y = fliplr(y);              % flip array left to right

   x = circshift(y, 1, 2);       % shift by 1 in the 2nd dimension (right shift by 1)

end

**b).**

The code to test the function we designed and the function in the Matlab library.

clear;

close all;

x = randn(1, 16);

N = length(x) / 2;

n = 1:N;

Z = fft(x);

%iffta; Z = X;

X\_conj = conj(Z);

x\_conj = fft(X\_conj);

a\_1 = conj(x\_conj);

a = a\_1 / length(Z);

%ifftb; Z = X;

y = fft(Z);

y = y / length(Z);

y = fliplr(y);              % flip array left to right

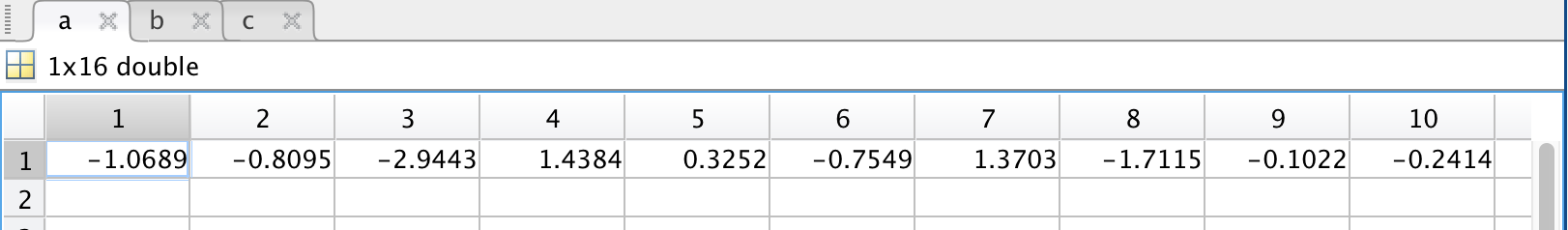
b = circshift(y, 1, 2);       % shift by 1 in the 2nd dimension (right shift by 1)

%inbuilt ifft function

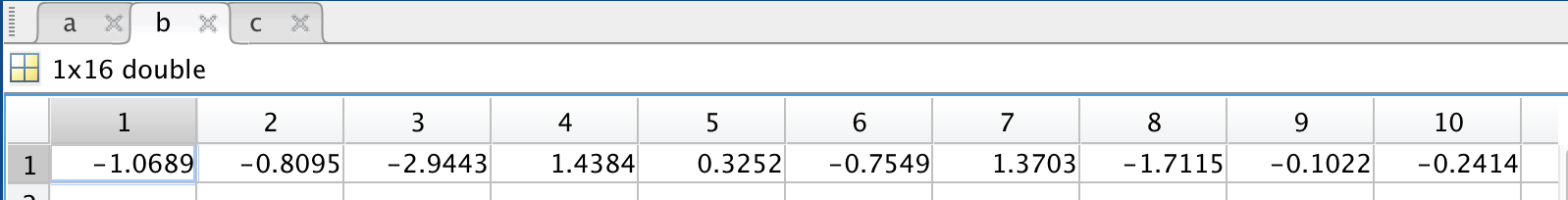
c = ifft(Z);

The results are shown below:

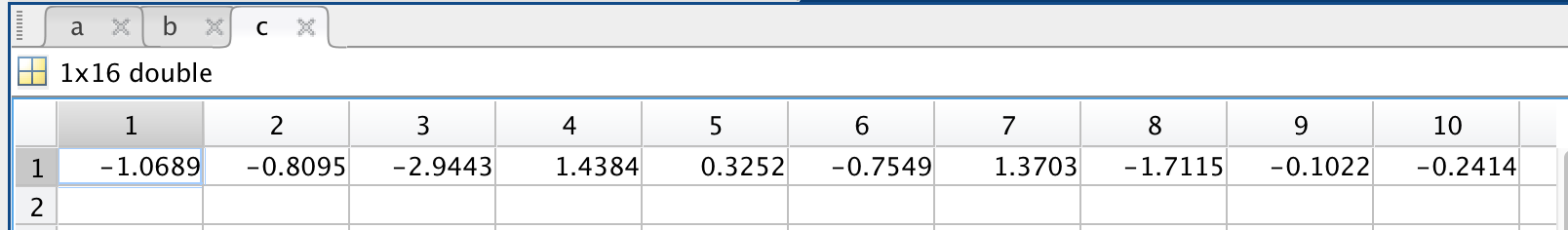
Results of iffta(x);



Results of ifftb(x);



Results of ifft function in Matlab;



**Task 2**

**a).**

The function which used to test conjugate symnmetry;

function y = isConjugateSymmetric(X)

   X = fft(x);

   Y = conj(X);

   y = fliplr(Y);              % flip array left to right

   x = circshift(y, 1, 2);       % shift by 1 in the 2nd dimension (right shift by 1)

   compare\_number = eps('single');

   bool = any(abs(x-X) > tolerance);

   outcome = ~bool

end

**b).**

The functions ifftc(X);

function x = ifftcs(X)

    if ~isConjugateSymmetric(X)

        error('input is not conjugate symmetric');

    end

    N = length(X) / 2;

    if N ~= round(N)

        error('input is not a length-2N sequence');

    end

    k = 1:N;

    X0(k) = X(k) + X(k + N);

    W = exp((k \* 1j \* pi) / N);

    X1(k) = W .\* (X(k) - X(k + N));

    Q = X0 + 1j \* X1;

    q = ifft(Q);

    x(k\*2+1) = 0.5 \* real(q(k));

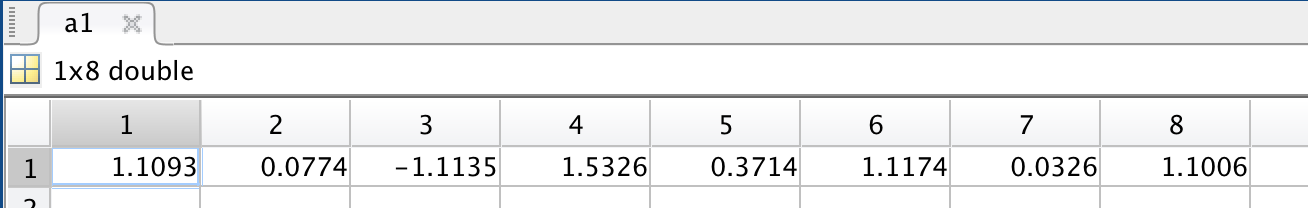
    x(k\*2) = 0.5 \* imag(q(k));

end

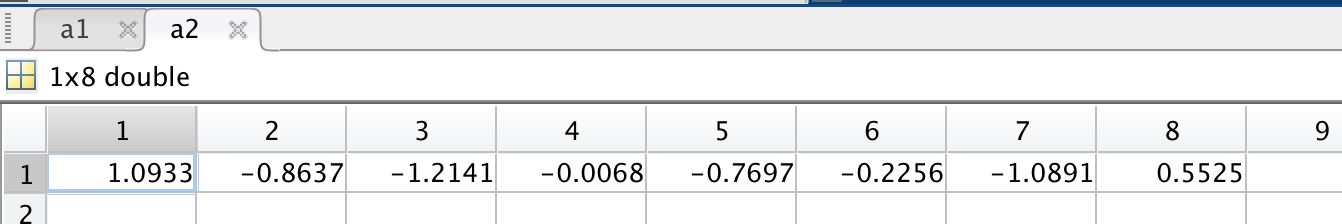
**c).**

The results of test functions we designed;

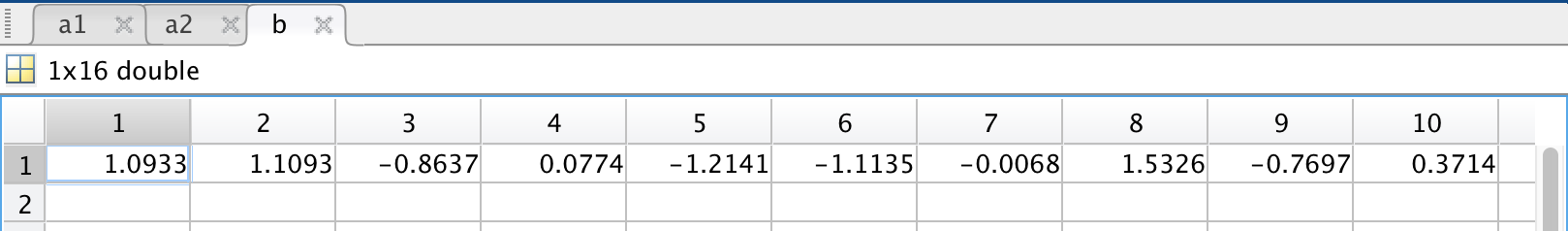
x[2n];



x[2n+1]



ifft(X[k]);



**Task 3**

1. **Write a Matlab function *overlapaddreal(B,x,N)*.**

The Matlab implementation is shown as below

function X = overlapadd(B, x, N)

%N is given in this case

N1=length(x);

M=length(B);

%therefore we calculate L based on N and M

L = N - M +1;

%new zero padded array of x

x=[x zeros(1,mod(-N1,L))];%to make sure x has interger number \* L's length

N2=length(x);

%pad zeros to filter coef array

B=[B zeros(1,L-1)];

%perform N = L+M-1 point FFT to the h

H\_k=fft(B,L+M-1);

%divide the new x array into

S=N2/L;

index=1:L;

X=zeros(M-1);%output

for stage=1:S

%take L elements from the x array, pad M-1 zeros to the end

xm=[x(index) zeros(1,M-1)];

%take N point DFT of this subsequence

X1=fft(xm,L+M-1);

%do linear convolution of this subsequence x\_1(eg) and the H array

Y=X1.\*H\_k;

%do inverse DFT of sub y sequence

Y=ifft(Y);

%Samples Added in every stage

Z=X((length(X)-M+2):length(X))+Y(1:M-1);

X=[X(1:(stage-1)\*L) Z Y(M:M+L-1)];

index=stage\*L+1:(stage+1)\*L;

end

%overlap DTF output is X

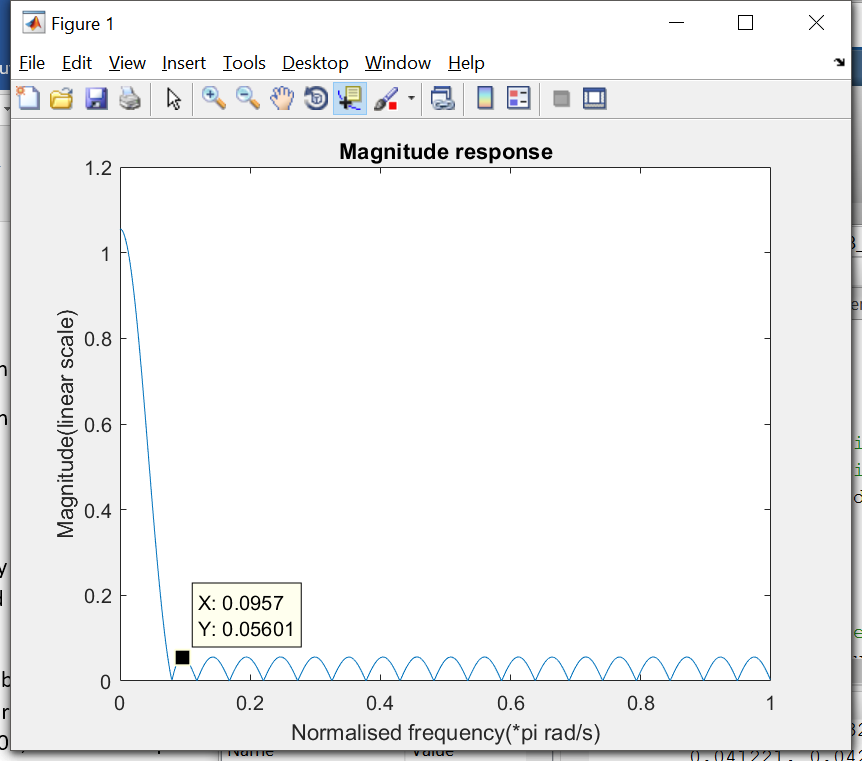
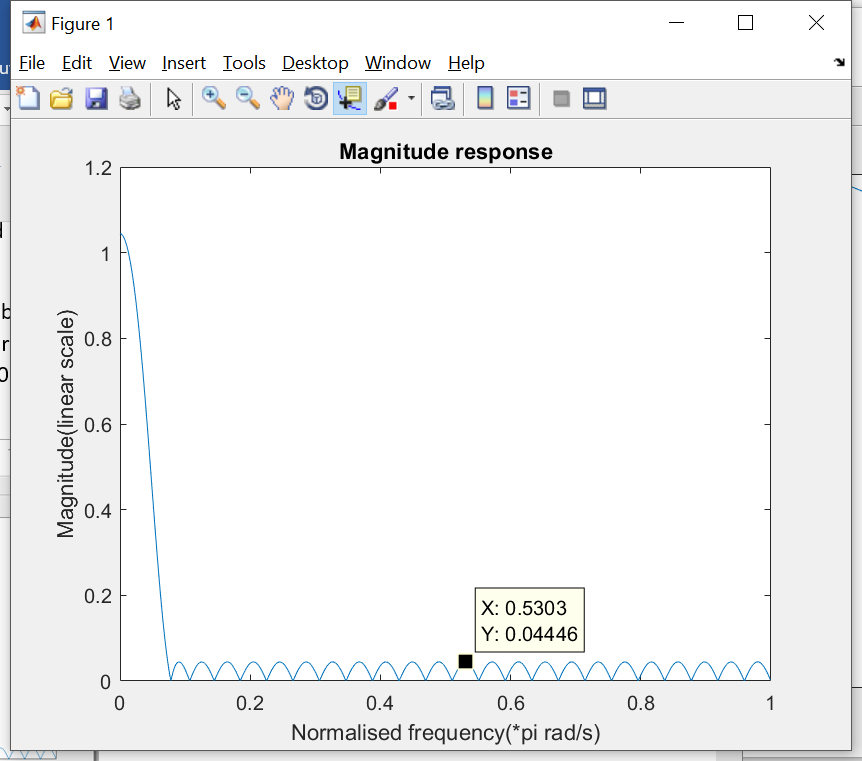
X = X(1:length(X)-length(Z)-mod(-N1,L));

end

**Task 4**

1. **The sampling frequency is 24kHz. Design an FIR filter according to the following specifications Passband [0, 220 Hz] Stopband Above 880 Hz Passband ripple 0.05 Stopband ripple 0.05**

The filtered has been designed by using Matlab function *firpm()* with the result from *firpmord()*, however, the original filter order by firpmord() doesn’t satisfy the design requirements, where stopband ripple is 0.005601>0.05, to solve this problem, we increase the filter order by 10, and the maximum stopband ripple is 0.04446<0.05 now.

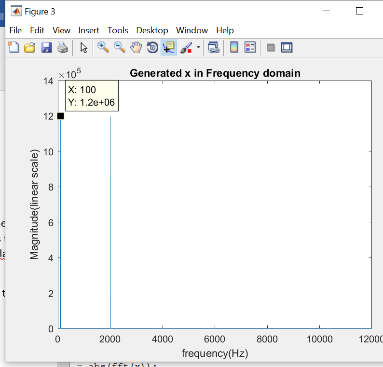
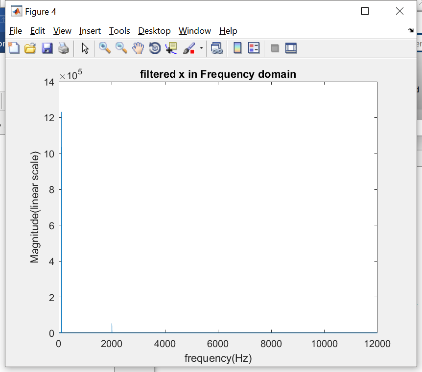
 

The Code for filter design has been attached to the appendix.

1. Generate a suitable test signal and implement the filter in a) using the Matlab routine you developed in Task 3. What is the optimal block length? Verify that your routine gives the same output as the filter and fftfilt command in Matlab. Explain briefly how the Matlab commands ***filter*** and ***fftfilt*** implements a linear filter.

A test signal consist of 2 sine wave components (w1 = 2\*pi\*100Hz, which will be able to pass the filter, and w2 = 2\*pi\*1000Hz, which is at the stopband of the filter) has been generated.

The test signal and after the FIR filter in frequency domain has been plotted in the figures below.



**Optimal block length**

It has been analysed that the total number of multiplication for a N point FFT is . A local minimum of this function at v = 8.65 has been found. and we rounded it up to 9. Therefore, the optimal block length of our filter will be .

**Verification**

The routine we developed in Part3. (c) has been verified by passing the result of function overlapadd() into a for loop to c to compare with the output from ***fftfilt(),*** and it can be shown the results are identical.

**How *fftfilt()* works comparing to *filter()***

It can be found in the official documentation that fftfilt() use the implementation of overlapadd method as based on FFT, which is the same as the filter we implemented in overlapadd(). The advantage of this method is overlapadd implementation of linear FIR filter can process a longer input sequence more efficiently (less computation) comparing to the normal *filter() function.*

**(c) How many complex multiplications and additions per second are required to implement the filter in each case?**

It can be estimated that if we implement the filter using *filter(),* linear convolution will be performed between h[n](filter coefficients) and input signal sequence, which will result in M\*Fs= 50\*24000 = **1200000** multiplication per second.

In fftfilt() where overlapadd is implemented, multiplication will be carried out.

In overlapadd with symmetry, it has been calculated that we have optimal block length when v = 8.75, and the number of multiplication per output data point is , therefore, the total number of multiplication per second is 6.06\*24000 = **145630**.

The matlab code for generating the signal and graphs above has been added to the appendix.

# Part C DSP implementation

The coding for DSP has been implemented for nomal time domain FIR filter in function *process\_time()*, and the overlapadd implemtation is in function *process\_block(fract32 output[]).*

The audio test signal generated from Matlab in PartB task 4 has ben inputted to the DSP board and it can be heard that only the low frequency component of the signal is outputted from the DSP board for both the time domain implementation and FFT overlap add implementation. The result has been estimated as the same we can hear from Matlab.

The code \_LabTaskC.c and prams.h has been attached to the appendix.

**Appendix**

**Part B Task 1**

iffta(x)

function x = iffta(X)

    X\_conj = conj(X);

    x\_conj = fft(X\_conj);

    x = conj(x\_conj);

    x = x / length(X);

end

ifftb(x)

function x = ifftb(X)

   y = fft(X);

   y = y / length(X);

   y = fliplr(y);              % flip array left to right

   x = circshift(y, 1, 2);       % shift by 1 in the 2nd dimension (right shift by 1)

end

**test of ifft and function designed**

clear;

close all;

x = randn(1, 16);

N = length(x) / 2;

n = 1:N;

Z = fft(x);

%iffta; Z = X;

X\_conj = conj(Z);

x\_conj = fft(X\_conj);

a\_1 = conj(x\_conj);

a = a\_1 / length(Z);

%ifftb; Z = X;

y = fft(Z);

y = y / length(Z);

y = fliplr(y);              % flip array left to right

b = circshift(y, 1, 2);       % shift by 1 in the 2nd dimension (right shift by 1)

%inbuilt ifft function

c = ifft(Z);

test conjugate symmetry

function y = isConjugateSymmetric(X)

   X = fft(x);

   Y = conj(X);

   y = fliplr(Y);              % flip array left to right

   x = circshift(y, 1, 2);       % shift by 1 in the 2nd dimension (right shift by 1)

   compare\_number = eps('single');

   bool = any(abs(x-X) > tolerance);

   outcome = ~bool

end

ifftc(x);

function x = ifftcs(X)

    if ~isConjugateSymmetric(X)

        error('input is not conjugate symmetric');

    end

    N = length(X) / 2;

    if N ~= round(N)

        error('input is not a length-2N sequence');

    end

    k = 1:N;

    X0(k) = X(k) + X(k + N);

    W = exp((k \* 1j \* pi) / N);

    X1(k) = W .\* (X(k) - X(k + N));

    Q = X0 + 1j \* X1;

    q = ifft(Q);

    x(k\*2+1) = 0.5 \* real(q(k));

    x(k\*2) = 0.5 \* imag(q(k));

end

**test ifft and code designed**

clear;

close all;

x = randn(1, 16);

N = length(x)/2;

k = 1:N;

Z = fft(x);

%ifftcs

 X0(k) = Z(k) + Z(k + N);

 W = exp(1j \* pi/ N).^(k-1);

 X1(k) = W .\* (Z(k) - Z(k + N));

 Q = X0 + 1j \* X1;

 q = ifft(Q);

 x(k\*2+1) = 0.5 \* real(q(k));

 x(k\*2) = 0.5 \* imag(q(k));

 a1 = [x(k\*2)];

 a2 = [x(k\*2+1)];

 %ifft function

 b = ifft(Z);

**Part B Task 4**

%filter design

clc;

clear;

close all;

N = 2^9; % block length

fs = 24E3;%samplin frequency

% FIR filter design

[ORD, passband\_edge, frequency\_band\_mag, w] = firpmord([220 880], [1 0], [0.05 0.05], fs);

ORD = ORD + 10;

num = firpm(ORD, passband\_edge, frequency\_band\_mag, w);

%verify the filter

[h, w] = freqz(num, 1, 2^10);

plot(w/pi, abs(h));

title('Magnitude response');

xlabel('Normalised frequency(\*pi rad/s)');

ylabel('Magnitude(linear scale)');

%PHASE

figure;

plot(w/pi, phase(h));

title('Phase response');

xlabel('Normalised frequency(\*pi rad/s)');

ylabel('Phase(rad)');

%num is the filter coefficients we need

%use print array for this array

print\_array(num, 'b');

%%%%Test signal genration

%central frequency at around

%r = r1+r2, where r1 at 100Hz,

f1 = 100;

w1 = 2\*pi\*f1;

%r2 at 1000Hz

f2 = 2000;

w2 = 2\*pi\*f2;

%test signal length is the same as the sampling frequency

n\_sampled = (1:fs\*100)/fs;

x = sin(w1\*n\_sampled)+sin(w2\*n\_sampled);

%plot this generated signal in frequency domain

figure;

N=fs\*100;

X1\_mags = abs(fft(x));

fax\_bins = [0 : N-1]; %frequency axis in bins

N\_2 = ceil(N/2);

plot(fax\_bins(1:N\_2)\*fs/N, X1\_mags(1:N\_2));

title('Generated x in Frequency domain');

xlabel('frequency(Hz)');

ylabel('Magnitude(linear scale)');

%sound(x, fs);

%%try to filter this signal with the filter we designed

x\_filtered = filter(num, 1, x);

%plot this generated signal in frequency domain

figure;

N=fs\*100;

X1\_mags = abs(fft(x\_filtered));

fax\_bins = [0 : N-1]; %frequency axis in bins

N\_2 = ceil(N/2);

plot(fax\_bins(1:N\_2)\*fs/N, X1\_mags(1:N\_2));

title('filtered x in Frequency domain');

xlabel('frequency(Hz)');

ylabel('Magnitude(linear scale)');

%sound(x, fs);

y\_overlapadd = overlapadd(num, x, N);

y\_filter = filter(num, 1, x);

y\_fftfilt = fftfilt(num, x);

for i= 1:1:length(y\_overlapadd)

if((y\_overlapadd(i)) ~= (y\_fftfilt(i)))

fprintf('WRONG!!!\n');

end

end

fprintf('the overlap add output is the same as the fftfilt() function output' );

**Part C**

\_LabTaskC.c

#include "SPWS3.h"

#include "Params.h"

complex\_fract32 twiddle[N/2] = { 0 };

complex\_fract32 filter\_fft[N] = { 0 };

complex\_fract32 input\_freq[N] = { 0 };

complex\_fract32 output\_fft[N] = { 0 };

fract32 output\_save[M-1] = { 0 };

// array b

float b[] = { -0.022345, 0.000024, 0.000477, 0.001261, 0.002341,

0.003740, 0.005422, 0.007399, 0.009625, 0.012098,

0.014760, 0.017602, 0.020547, 0.023589, 0.026627,

0.029677, 0.032549, 0.035320, 0.037891, 0.040180,

0.042183, 0.043828, 0.045099, 0.045957, 0.046392,

0.046392, 0.045957, 0.045099, 0.043828, 0.042183,

0.040180, 0.037891, 0.035320, 0.032549, 0.029677,

0.026627, 0.023589, 0.020547, 0.017602, 0.014760,

0.012098, 0.009625, 0.007399, 0.005422, 0.003740,

0.002341, 0.001261, 0.000477, 0.000024, -0.022345 };

float process\_time(float x0)

{

    // TODO: 1. Implement the filter using time domain methods

static float x[BUFFER\_SIZE] = {0.0}; // BUFFER\_SIZE = (M-1) is defined in 'Params.h'

static int current = 0;

//return back forwading position of the array if current is negative

float y = b[0] \* (x0 + x[(current + BUFFER\_SIZE)%BUFFER\_SIZE]); // Macro 'REM(current)' is defined in 'Params.h'

x[current] = x0;

// save current x0 into x after 'y' is calculated, thus the size of 'x' can be reduced by 1 (from M to M-1).

for (int i = 1; i <= BUFFER\_SIZE/2-1; i++) {

y += b[i] \* (x[(current-i) + BUFFER\_SIZE)%BUFFER\_SIZE)] + x[((current+i) + BUFFER\_SIZE)%BUFFER\_SIZE]);

}

y += b[BUFFER\_SIZE/2] \* x[((current-BUFFER\_SIZE/2)) + BUFFER\_SIZE)%BUFFER\_SIZE];

current++;

current %= BUFFER\_SIZE;//INCREMENT EVERY BUFFER\_SIZE

return y;

}

void init\_process()

{

int i;

// calculate twiddle factors

twidfftrad2\_fr32(twiddle, N);

// copy filter coefficients to input array to do fft

for (i = 0; i < M; i++)

input\_data[i] = (1 << 30) \* b[i];

// [ note ]

// Here we should scale by (1 << 31)-1 for full scale, however

// doing so can cause overflows in fixed point, so we halve it

// here and put back the factor 2 on output.

// do fft

int filter\_blk\_exp;

rfft\_fr32(input\_data, filter\_fft, twiddle, 1, N, &filter\_blk\_exp, 1);

// rescale data points

for (i = 0; i < N; i++)

{

filter\_fft[i].re = filter\_fft[i].re << (filter\_blk\_exp);

filter\_fft[i].im = filter\_fft[i].im << (filter\_blk\_exp);

}

// clear input array

for (i = 0; i < M; i++)

input\_data[i] = 0;

}

void process\_block(fract32 output[])

{

// TODO: 2. Implement the filter using the overlap-add method

int index = 0;

int blk\_exp;

//input\_data defined, input\_fft->input\_freq, twiddle, N,

rfft\_fr32(input\_data, input\_freq, twiddle, 1, N, &blk\_exp, 1);

// conjugate symmetry

output\_fft[0] = cmlt\_fr32(filter\_fft[0], input\_freq[0]);

index = 1;

while(index<N/2){

//multiply H[k] and x[k]

output\_fft[index] = cmlt\_fr32(filter\_fft[index], input\_fft[index]);

//secong half of the FFT is conjugate of the first half

output\_fft[N-index] = conj\_fr32(output\_fft[index]);//complex conjugate

index++;

}

output\_fft[N/2] = cmlt\_fr32(filter\_fft[N/2], input\_fft[N/2]);

complex\_fract32 output\_complex[N]= { 0 };

//inverse FFT of the output array

ifft\_fr32(output\_fft, output\_complex, twiddle, 1, N, &blk\_exp, 1);

    int i;

    for (i = 0; i < N; i++)

    {

//re scale data point as what he did in init\_process()

        output[index] = output\_complex[index].re << (blk\_exp);

}

// overlap add

index = 0;

while( index < M-1 ) {

output[index] += output\_save[index];

output\_save[index] = output[L+index];

index++

}

}

prams.h

// TODO: 0. Modify these constants to match the filter you have designed

// length of filter

#define M 49

// buffer size

#define N 512 //this is dertermined by the optimal block length

// input data processing block size

#define L (N-M+1)

#define BUFFER\_SIZE (M-1)