Fast Convolution

Computer Assignment 3

Nik Jensen

ECE 5630

The purpose of this programming assignment was to give practice in performing digital signal processing, to help solidify my understanding of the DFT, and to solidify my understanding of the overlap-add and overlap-save method for convolution. This assignment consisted of designing a linear-phase FIR digital filter, convolving that filter with created cosines of limited length and filtering with a provided sound file. This filtering was expanded to two programs; one, with standard convolution, and the other, with a provided FFT algorithm to filter in the frequency domain. Lastly, the two programs were compared for computational efficiency and time elapsed for the filtering.

Design of an FIR Digital Filter

To begin, the first portion of the assignment was to design our FIR digital filter. The FIR filter was required to be a 256-coefficient low pass filter, with a 300 Hz passband, 400 Hz stopband, a signal sample rate of 11.025 kHz, and a unity gain in the passband. For my design, I used MATLAB's fdatool using the given parameters to plot and create the impulse response, h(n), and the magnitude response and phase response. This can be seen in Figure 1 below. This filter was also compared to a desired magnitude response that is included in Figure 2 below.

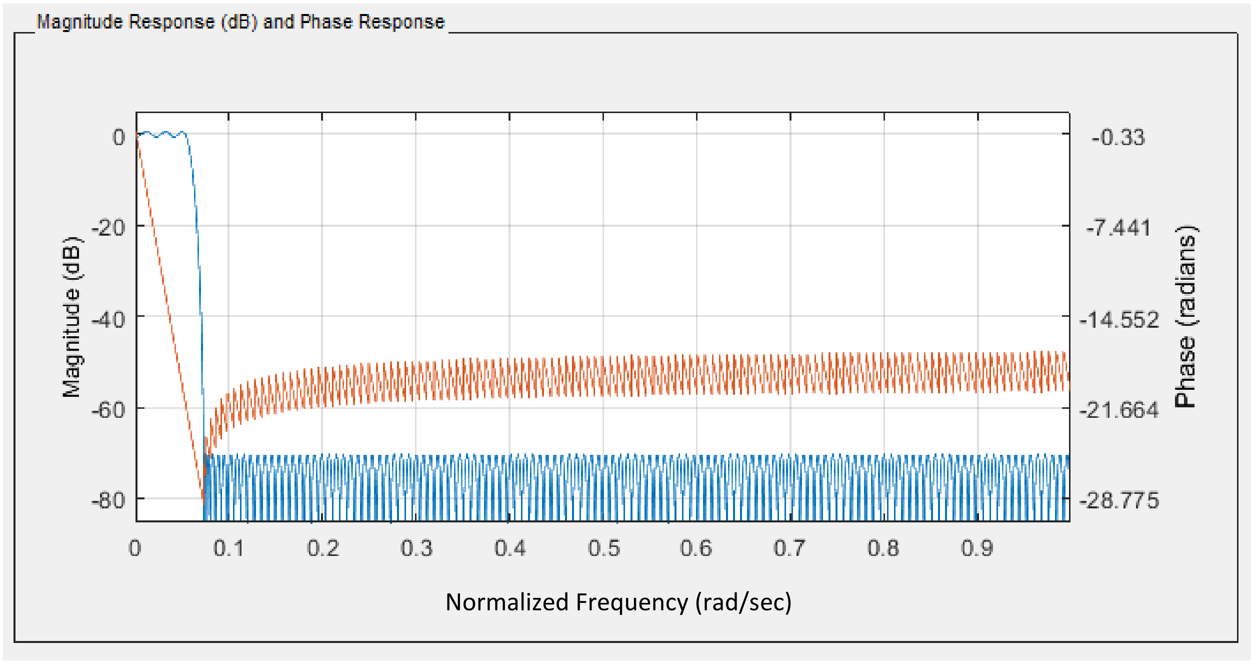
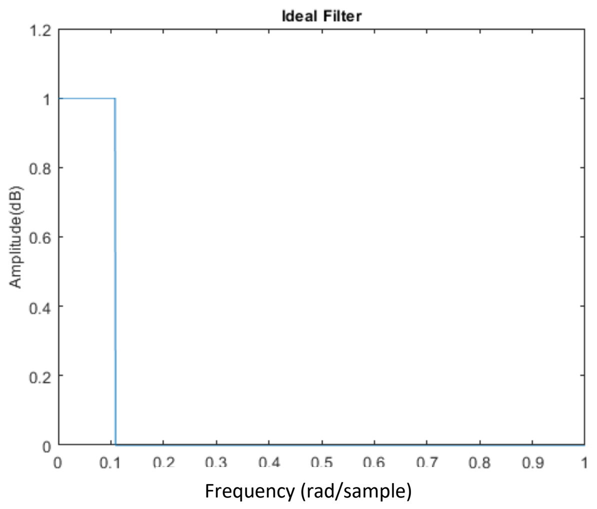


Figure 1 Actual magnitude response of our low pass filter



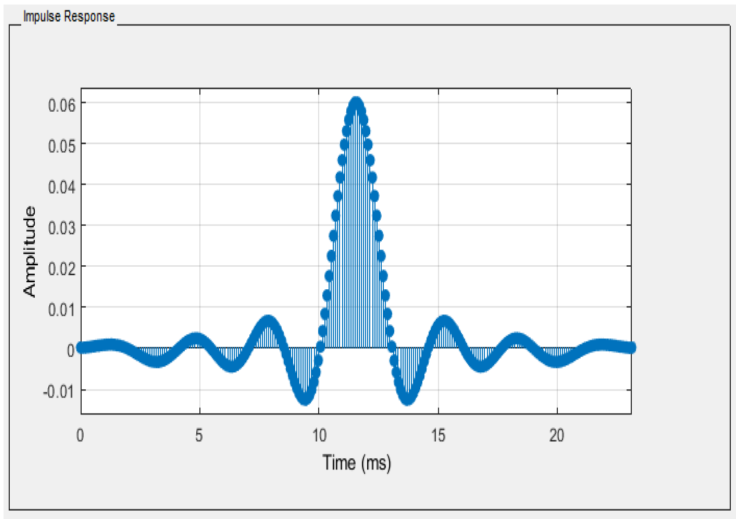


Figure 2 Our desired magnitude response of our low pass filter and actual impulse response

Cosine Filtering

To start with the programming, the cosine task was to create cosines of different frequencies. The frequencies used were f = 10Hz, f = 40Hz, f = 150Hz, f = 350Hz, and f = 500Hz. This was easily done in a Matlab script that outputted the results to multiple binary files to be filtered with our programs. With the combined results of the second program, these cosine's magnitude responses have been provided in Figure 3.

The first task was to filter the cosine signals with our FIR filters in the time domain. The code for this can be found in the appendix of this document. To compare the efficiency in calculations with this filtering convolution algorithm, it was calculated to find the number of multiples and adds per output sample. For filtering in the time domain, the number of calculations per output sample was, (*length of the FIR filter)* multiples and (*length of the FIR filter*) adds. This resulted in 255 multiples and adds, which is slightly costly in terms of computations per output sample.

The second task involved creating a program to filter our input in the frequency domain. This was done by using the provided fft842.c algorithm to act as a 512-DFT function. By plugging in the input cosine and filters into our algorithm, multiplying in the frequency domain, and taking the result's inverse DFT, we can essentially filter our cosines. The trick for the second program, is the implementation of an overlap-add or overlap-save algorithm inside the computation. For my program, I used the overlap-save method. The resulting magnitude response plots can be found with the included convolution method plots as seen in Figure 3. While not completely matched to the convolution results, the DFT results still match the frequency of the original desired cosine frequency response.

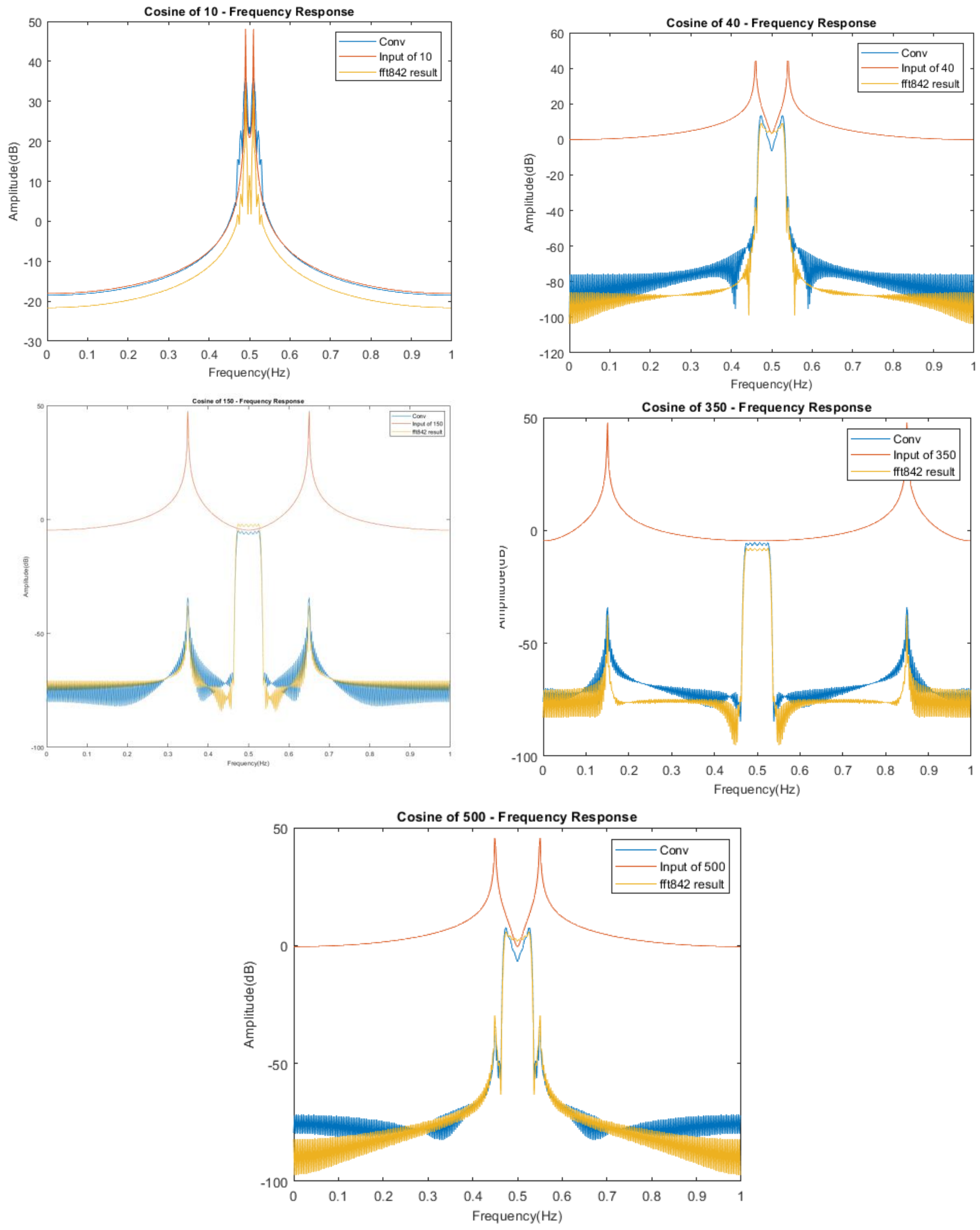


Figure 3 Magnitude Response Plots for our cosine inputs

The resulting computational efficiency was apparent as for each N-point DFT, we have 2NLog2(N) multiples and 4NLog2(N) additions. Because we are looking at the number of additions and multiplies in a single output sample, we can divide these by L, the length of our input signal, resulting in around 36 multiplies and 72 adds. This is 220 multiples less, and 184 additions less in terms of computational work for a filtered signal. These results can be reflected for the overlap-add function.

Sound File Filtering

The last portion of this programming assignment was to run the provided input file through our two programmed filters and compare the results. As what should be expected, they sound the same. Sadly, my results did not sound completely the same. From the sounds of it, my fft algorithm most likely did not implement completely correct, and the sound file sounded slightly off. My current guess is that this was due to my overlap-save algorithm.

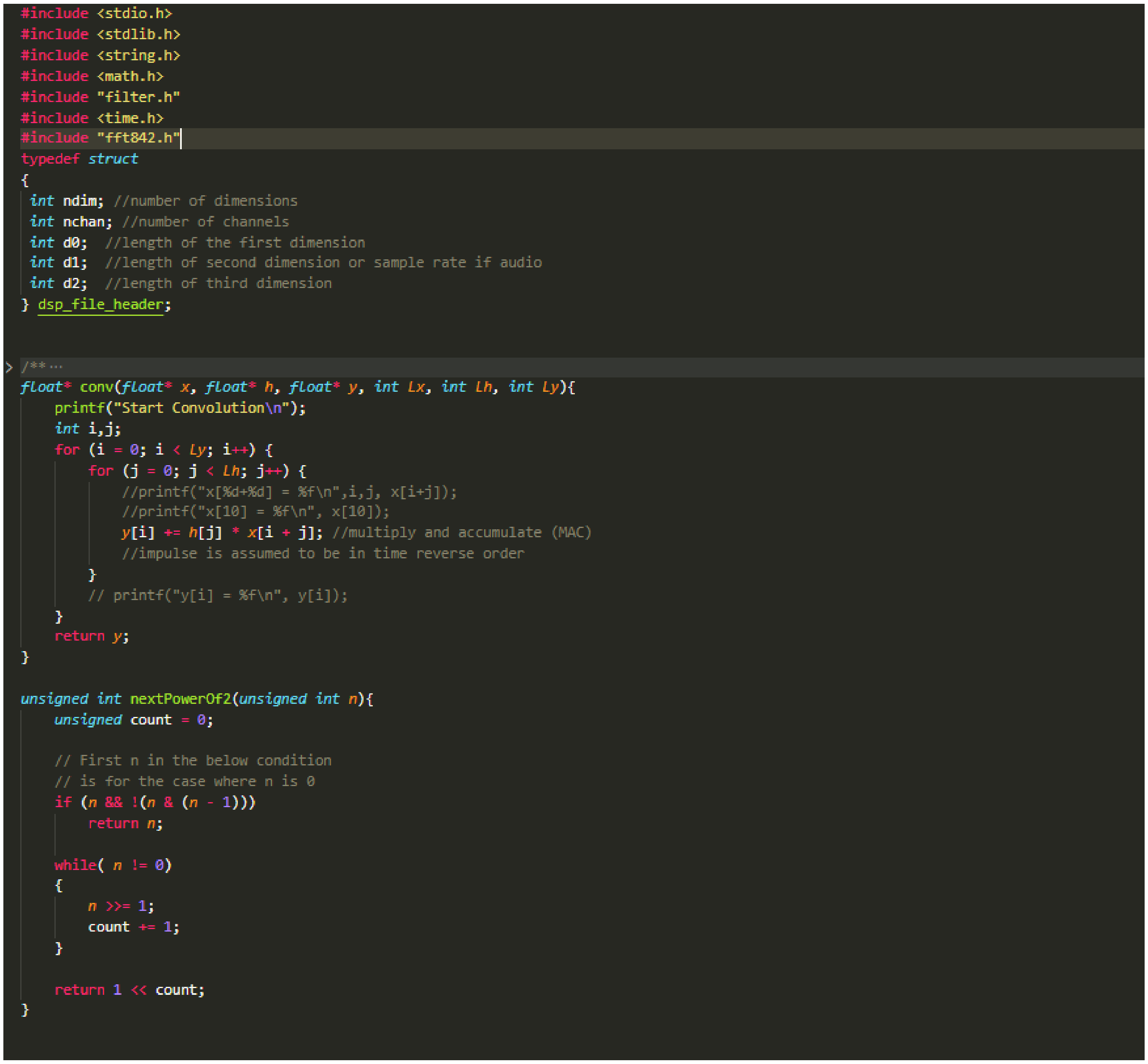
I was also tasked to record and report the speed at which this algorithm takes. After varied results, most likely due to my processor and applications running, I resulted in 0.453 seconds for the convolution program, and 0.109 seconds for the FFT algorithm.

Conclusion

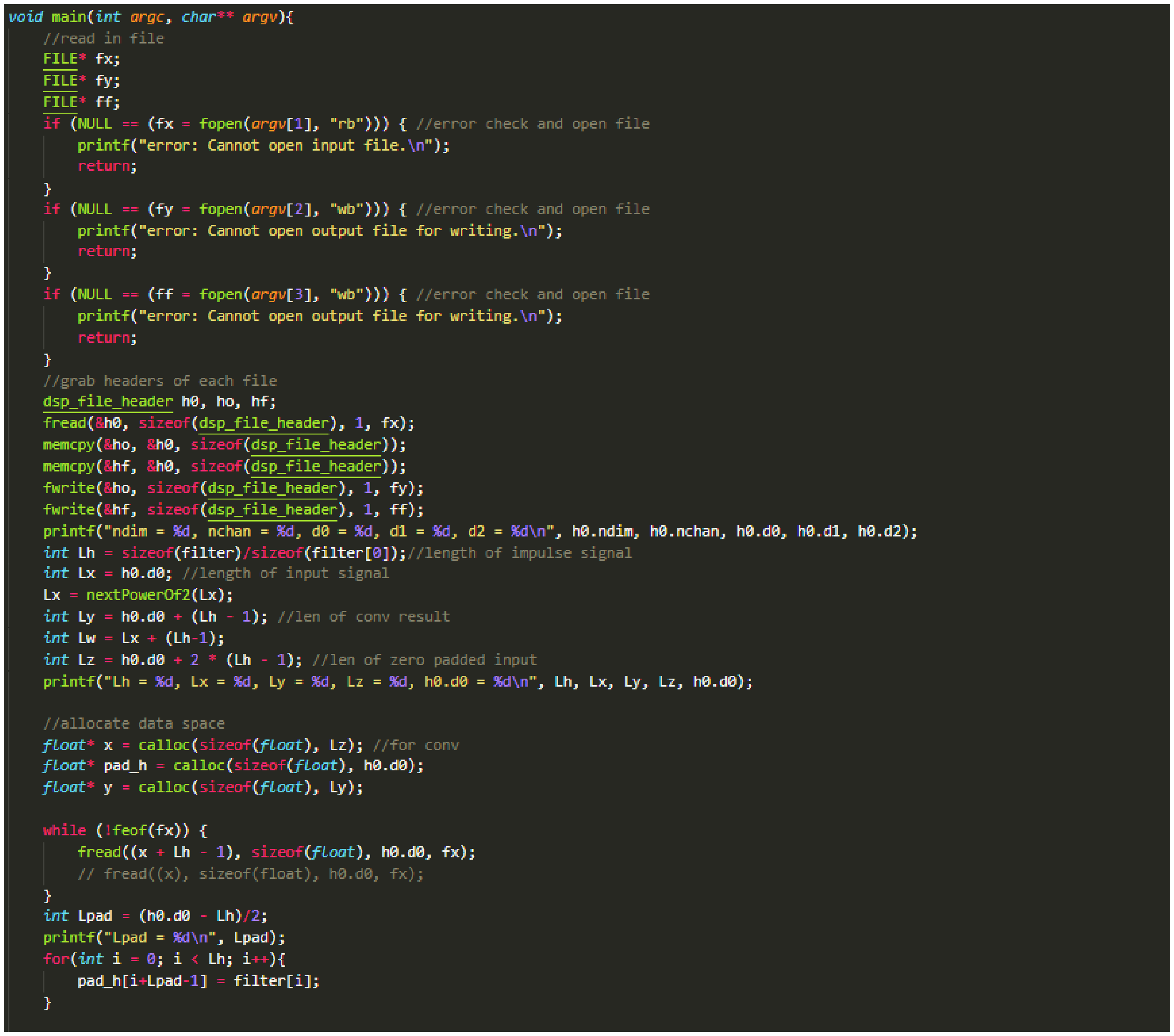
I believe working with this programming assignment has given me a better insight into how the DFT works, overlap methods for optimization, and discrete signal processing. This assignment proved that the overlap-save algorithm is computationally more efficient, but higher in complexity when implementing the method. Comparing the timed results of not just the sound file, but with the cosine waveforms, the overlap-save algorithm is very efficient.

Listening to the sound file, the results are mostly to what I expected. The lowpass filter removed a good portion of the high frequencies, and kept our lower frequencies in the result. I would like to note the problem with the FFT sound file. As seen in Figure 3, the frequency response of our FFT842 and convolution do not match. They have the same behavior, but are not the same. This is most likely due to my overlap-save implementation, as it is a lot more complex than convolution and a lot to handle with the given time I had. Because of this problem, my sound files did not sound the same, and the FFT sound file kept or even created some sound problems. Although, the FFT sound file did follow a lot of the behavior of the convolution filtered result.

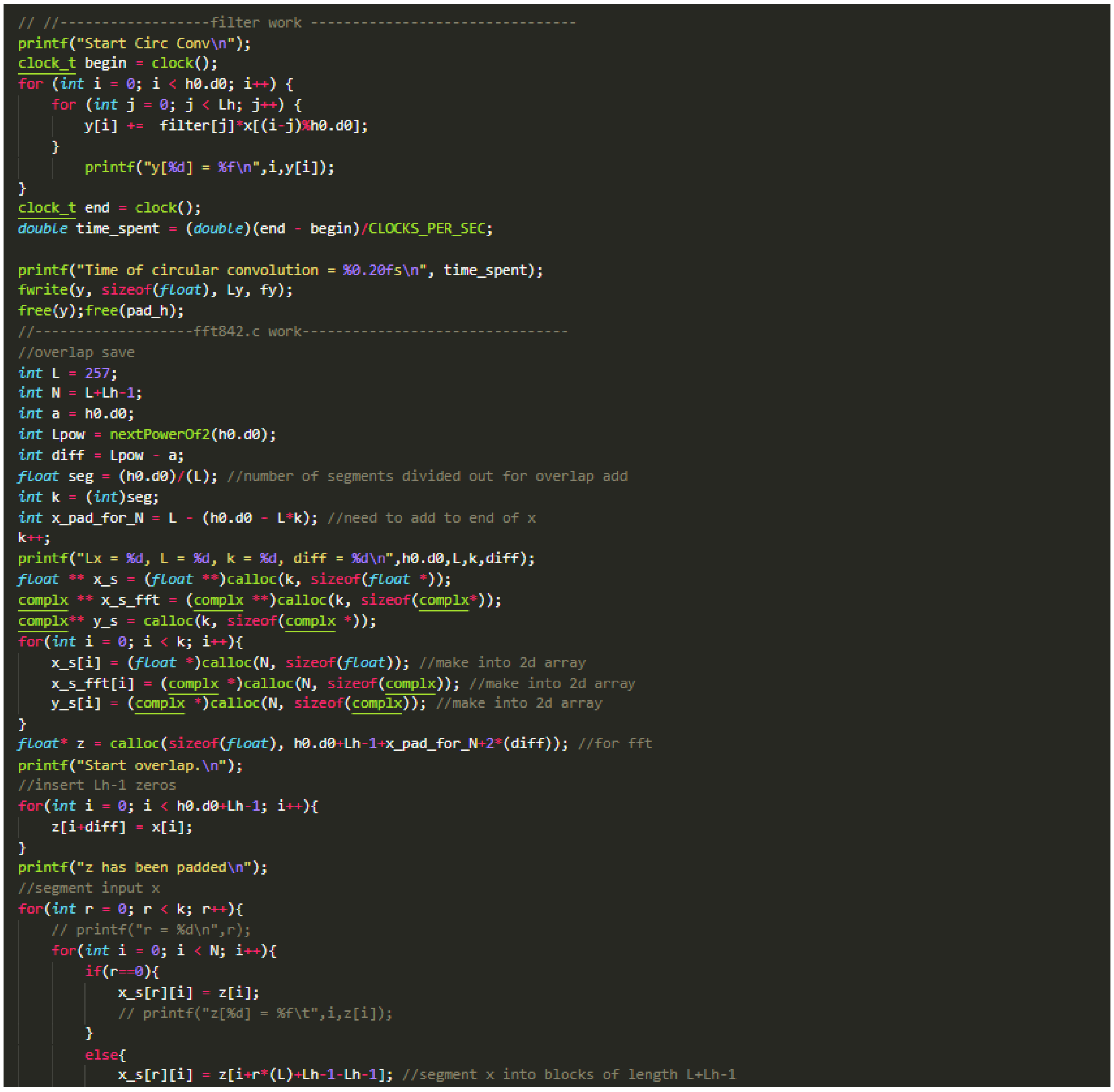
Code



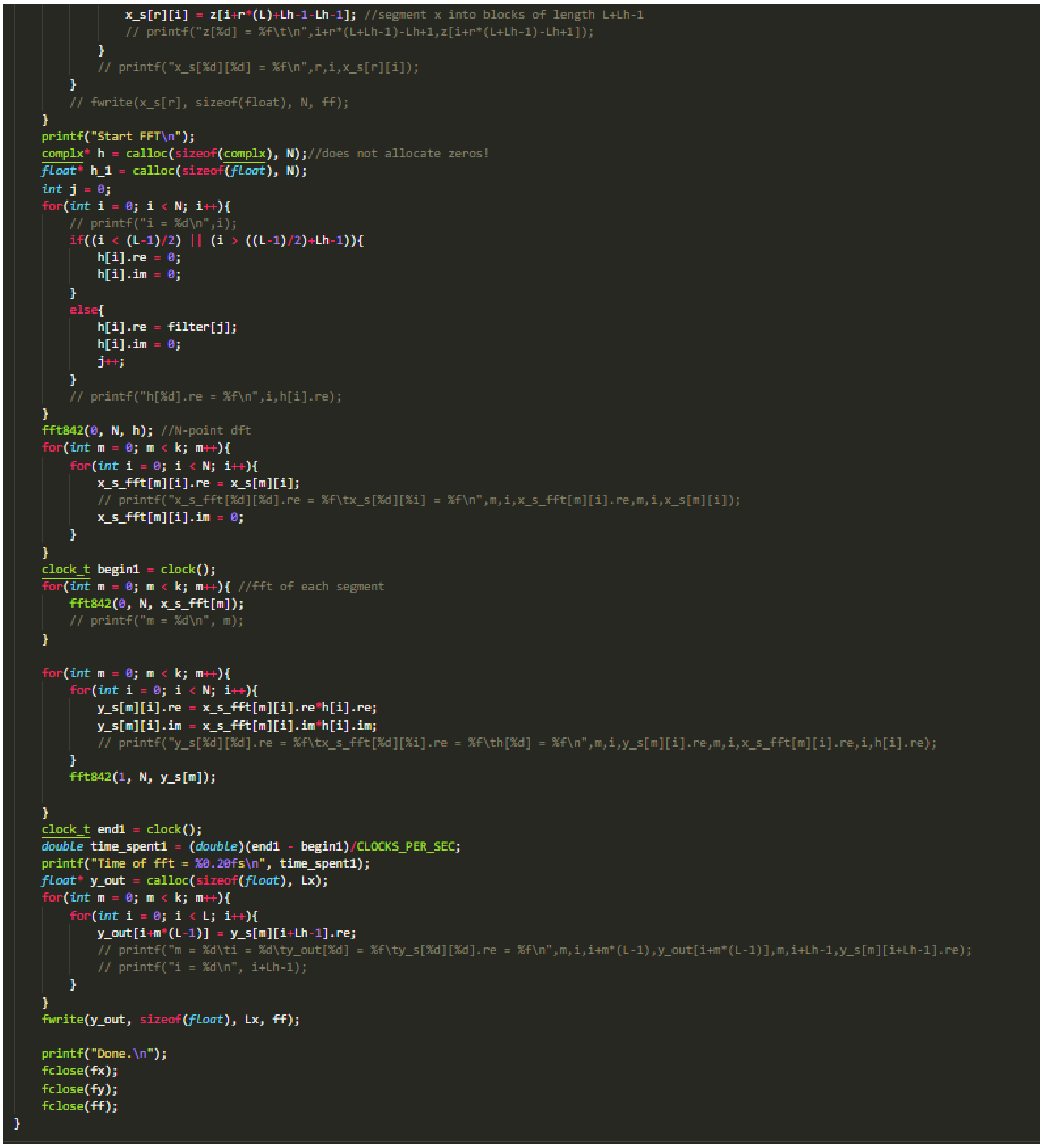
```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>
#include "filter.h"
#include <time.h>
#include "fft842.h"
typedef struct
{
 int ndim; //number of dimensions
 int nchan; //number of channels
 int d0;  //length of the first dimension
 int d1;  //length of second dimension or sample rate if audio
 int d2;  //length of third dimension
} dsp_file_header;


/**...
float* conv(float* x, float* h, float* y, int Lx, int Lh, int Ly){
    printf("Start Convolution\n");
    int i,j;
    for (i = 0; i < Ly; i++) {
        for (j = 0; j < Lh; j++) {
            //printf("x[%d+%d] = %f\n",i,j, x[i+j]);
            //printf("x[10] = %f\n", x[10]);
            y[i] += h[j] * x[i + j]; //multiply and accumulate (MAC)
            //impulse is assumed to be in time reverse order
        }
        // printf("y[i] = %f\n", y[i]);
    }
    return y;
}

unsigned int nextPowerOf2(unsigned int n){
    unsigned count = 0;

    // First n in the below condition
    // is for the case where n is 0
    if (n && !(n & (n - 1)))
        return n;

    while( n != 0)
    {
        n >>= 1;
        count += 1;
    }

    return 1 << count;
}
```



```c
void main(int argc, char** argv){
    //read in file
    FILE* fx;
    FILE* fy;
    FILE* ff;
    if (NULL == (fx = fopen(argv[1], "rb"))) { //error check and open file
        printf("error: Cannot open input file.\n");
        return;
    }
    if (NULL == (fy = fopen(argv[2], "wb"))) { //error check and open file
        printf("error: Cannot open output file for writing.\n");
        return;
    }
    if (NULL == (ff = fopen(argv[3], "wb"))) { //error check and open file
        printf("error: Cannot open output file for writing.\n");
        return;
    }
    //grab headers of each file
    dsp_file_header h0, ho, hf;
    fread(&h0, sizeof(dsp_file_header), 1, fx);
    memcpy(&ho, &h0, sizeof(dsp_file_header));
    memcpy(&hf, &h0, sizeof(dsp_file_header));
    fwrite(&ho, sizeof(dsp_file_header), 1, fy);
    fwrite(&hf, sizeof(dsp_file_header), 1, ff);
    printf("ndim = %d, nchan = %d, d0 = %d, d1 = %d, d2 = %d\n", h0.ndim, h0.nchan, h0.d0, h0.d1, h0.d2);
    int Lh = sizeof(filter)/sizeof(filter[0]);//length of impulse signal
    int Lx = h0.d0; //length of input signal
    Lx = nextPowerOf2(Lx);
    int Ly = h0.d0 + (Lh - 1); //len of conv result
    int Lw = Lx + (Lh-1);
    int Lz = h0.d0 + 2 * (Lh - 1); //len of zero padded input
    printf("Lh = %d, Lx = %d, Ly = %d, Lz = %d, h0.d0 = %d\n", Lh, Lx, Ly, Lz, h0.d0);

    //allocate data space
    float* x = calloc(sizeof(float), Lz); //for conv
    float* pad_h = calloc(sizeof(float), h0.d0);
    float* y = calloc(sizeof(float), Ly);

    while (!feof(fx)) {
        fread((x + Lh - 1), sizeof(float), h0.d0, fx);
        // fread((x), sizeof(float), h0.d0, fx);
    }
    int Lpad = (h0.d0 - Lh)/2;
    printf("Lpad = %d\n", Lpad);
    for(int i = 0; i < Lh; i++){
        pad_h[i+Lpad-1] = filter[i];
    }
```



```c
// //------------------filter work --------------------------------
printf("Start Circ Conv\n");
clock_t begin = clock();
for (int i = 0; i < h0.d0; i++) {
    for (int j = 0; j < Lh; j++) {
        y[i] +=  filter[j]*x[(i-j)%h0.d0];
    }
        printf("y[%d] = %f\n",i,y[i]);
}
clock_t end = clock();
double time_spent = (double)(end - begin)/CLOCKS_PER_SEC;

printf("Time of circular convolution = %0.20fs\n", time_spent);
fwrite(y, sizeof(float), Ly, fy);
free(y);free(pad_h);
//------------------fft842.c work-------------------------------
//overlap save
int L = 257;
int N = L+Lh-1;
int a = h0.d0;
int Lpow = nextPowerOf2(h0.d0);
int diff = Lpow - a;
float seg = (h0.d0)/(L); //number of segments divided out for overlap add
int k = (int)seg;
int x_pad_for_N = L - (h0.d0 - L*k); //need to add to end of x
k++;
printf("Lx = %d, L = %d, k = %d, diff = %d\n",h0.d0,L,k,diff);
float ** x_s = (float **)calloc(k, sizeof(float *));
complx ** x_s_fft = (complx **)calloc(k, sizeof(complx*));
complx** y_s = calloc(k, sizeof(complx *));
for(int i = 0; i < k; i++){
    x_s[i] = (float *)calloc(N, sizeof(float)); //make into 2d array
    x_s_fft[i] = (complx *)calloc(N, sizeof(complx)); //make into 2d array
    y_s[i] = (complx *)calloc(N, sizeof(complx)); //make into 2d array
}
float* z = calloc(sizeof(float), h0.d0+Lh-1+x_pad_for_N+2*(diff)); //for fft
printf("Start overlap.\n");
//insert Lh-1 zeros
for(int i = 0; i < h0.d0+Lh-1; i++){
    z[i+diff] = x[i];
}
printf("z has been padded\n");
//segment input x
for(int r = 0; r < k; r++){
    // printf("r = %d\n",r);
    for(int i = 0; i < N; i++){
        if(r==0){
            x_s[r][i] = z[i];
            // printf("z[%d] = %f\t",i,z[i]);
        }
        else{
            x_s[r][i] = z[i+r*(L)+Lh-1-Lh-1]; //segment x into blocks of length L+Lh-1
```



```c
                x_s[r][i] = z[i+r*(L)+Lh-1-Lh-1]; //segment x into blocks of length L+Lh-1
                // printf("z[%d] = %f\t\n",i+r*(L+Lh-1)-Lh+1,z[i+r*(L+Lh-1)-Lh+1]);
            }
            // printf("x_s[%d][%d] = %f\n",r,i,x_s[r][i]);
        }
        // fwrite(x_s[r], sizeof(float), N, ff);
    }
    printf("Start FFT\n");
    complx* h = calloc(sizeof(complx), N);//does not allocate zeros!
    float* h_1 = calloc(sizeof(float), N);
    int j = 0;
    for(int i = 0; i < N; i++){
        // printf("i = %d\n",i);
        if((i < (L-1)/2) || (i > ((L-1)/2)+Lh-1)){
            h[i].re = 0;
            h[i].im = 0;
        }
        else{
            h[i].re = filter[j];
            h[i].im = 0;
            j++;
        }
        // printf("h[%d].re = %f\n",i,h[i].re);
    }
    fft842(0, N, h); //N-point dft
    for(int m = 0; m < k; m++){
        for(int i = 0; i < N; i++){
            x_s_fft[m][i].re = x_s[m][i];
            // printf("x_s_fft[%d][%d].re = %f\tx_s[%d][%i] = %f\n",m,i,x_s_fft[m][i].re,m,i,x_s[m][i]);
            x_s_fft[m][i].im = 0;
        }
    }
    clock_t begin1 = clock();
    for(int m = 0; m < k; m++){ //fft of each segment
        fft842(0, N, x_s_fft[m]);
        // printf("m = %d\n", m);
    }

    for(int m = 0; m < k; m++){
        for(int i = 0; i < N; i++){
            y_s[m][i].re = x_s_fft[m][i].re*h[i].re;
            y_s[m][i].im = x_s_fft[m][i].im*h[i].im;
            // printf("y_s[%d][%d].re = %f\tx_s_fft[%d][%i].re = %f\th[%d] = %f\n",m,i,y_s[m][i].re,m,i,x_s_fft[m][i].re,i,h[i].re);
        }
        fft842(1, N, y_s[m]);

    }
    clock_t end1 = clock();
    double time_spent1 = (double)(end1 - begin1)/CLOCKS_PER_SEC;
    printf("Time of fft = %0.20fs\n", time_spent1);
    float* y_out = calloc(sizeof(float), Lx);
    for(int m = 0; m < k; m++){
        for(int i = 0; i < L; i++){
            y_out[i+m*(L-1)] = y_s[m][i+Lh-1].re;
            // printf("m = %d\ti = %d\ty_out[%d] = %f\ty_s[%d][%d].re = %f\n",m,i,i+m*(L-1),y_out[i+m*(L-1)],m,i+Lh-1,y_s[m][i+Lh-1].re);
            // printf("i = %d\n", i+Lh-1);
        }
    }
    fwrite(y_out, sizeof(float), Lx, ff);

    printf("Done.\n");
    fclose(fx);
    fclose(fy);
    fclose(ff);
}
```