CALIFORNIA STATE UNIVERSITY
LONG BEACH

Dec 2019
EE 489

Lab # 4

Fast Fourier Transform

prepared by
Brian Fox
Arya Daroui
Inessa Lopez

1. Introduction

The goal of this project is to design and implement a 128-point radix-2 decimation in frequency (DIF) fast Fourier transform (FFT) in C code using C55x hardware in order to perform discrete Fourier transform (DFT), as well as to utilize the same FFT algorithm to perform inverse DFT on the spectrum generated by the initial FFT. This is all to become familiar with and gain firsthand experience working with the FFT algorithm and its digital signal processing hardware implementation.

2. Methodology and discussion

2.1 General

To start with we were given a C code Code Composer Studio (CCS) project that implemented FFT using the decimation in time (DIT) method from the text Real-Time Digital Signal Processing: Fundamentals, ImplementationsandApplications by Kuo, Lee, and Tian, that we were to modify to perform DIF FFT using two different scaling methods, as well as IDFT using the same DIF FFT we created. The input data given was 1664 points in length, and was a sinusoid with continuously rising frequency, or a chirp signal, as can be seen in Figure 3. We were to perform thirteen 128-point FFTs on this input data, generating thirteen spectra, which should obviously each show increasing frequency. This output was 832 points long since the code was written to only show the left half of each of the thirteen spectra, which accounts for the length being halved. The code naturally outputs all thirteen spectra simply concatenated onto one another in the order they were performed, and we chose to output it the same way and split them off for individual inspection manually after the code was run. The full output we got is shown in Figure 4. To be able to see the individual spectra better, we also split the thirteen spectra into thirteen plots, allowing for accurate comparison between the locations of the frequency domain impulses, seeing the monotonically rising frequency. Since the FFTs each result in what is approximately a single sinusoid, the resultant DFT calculated very nearly approximates that of a single-frequency sinusoid, which looks like either a single impulse shifted along the independent frequency axis to the frequency of the sinusoid, or a pair of impulses shifted right to the frequency and left to the negative of the frequency of the sinusoid.

2.2 Bit reversal

The first part of this lab was converting the given DIT FFT code to be DIF FFT. This required a few steps. For one, the bit reversal needs to happen at a different time. Bit reversal is the process of shifting either the FFTs input or output bit order based on its location bits order being reversed. In a 3-bit FFT, which equates to an 8-point FFT since $2^3 = 8$, the locations of the bits are represented by three bits themselves. Obviously, the point in position 000 and 111 remain unchanged, but the point in location 001 must be shifted to position 100 and vice-versa. This works exactly the same way for the 7-bit, or 128-point, FFT we use in this lab. In this instance, the point in position 0000000 and 1111111 remain unaffected, but the point in position 0111010 swaps with that in position 0101110. The concept is exactly the same for 2-bit, 4-bit, 5-bit, and 6-bit FFT, all of which are used in this experiment. All of this says nothing of the difference between bit reversal in DIF vs bit reversal in DIT because everything up to this point is exactly the same for each. The difference is in the timing. When utilizing the DIT algorithm, as given in this experiment, bit reversal happens before the FFT algorithm, since the algorithm is designed around the inputs being in bit-reversed order, yielding an output thats in natural order. When using DIF, however, the natural order of the input is correct, but the output will need to be bit reversed to yield the desired answer, which is to say the output is in bit-reversed order. The reason for the vague wording of input and output is because both DIF and DIT implementations of the FFT algorithm can be used to perform DFT, converting a time-domain signal to the signals spectrum, or inverse DFT (IDFT), converting a spectrum into a time-domain signal.

2.3 The FFT

Fourier transform has long been used as a means of understanding and modifying continuous-time signals x(t), by converting them to their frequency domain counterparts. This has many benefits, including allowing a better understanding of signals by showing their frequency spectrum and making use of properties such as the bi-directional conversion between multiplication and convolution between time and frequency domain computations. Discrete-time Fourier transform (DTFT) allows these same benefits for discrete or sampled systems $x[nTs]$ or $x[n]$, but gives a continuous-time frequency spectrum $X(\omega)$. One of the biggest reasons to sample a time-domain signal is the simplification of computer-based calculations, as infinite points no longer need to be taken into consideration by a program. This benefit is much of the impetus of digital signal processing and digitization of signals in general. The discrete Fourier transform (DFT) allows for the sampling of the DTFT, gaining these same benefits as sampling in the time-domain of required storage size and expedited numerical calculations, and is represented as $X(k) = \sum_{n=0}^{N-1} x(n)e^{-j(\frac{2\pi kn}{N})}, \quad k = 0, 1, ...N-1$. The variable k represents the frequency index, where X(k) is the kth DFT coefficient, and it is bounded from 0 to N-1, creating N equally spaced discrete radian frequencies $0 \geq \omega \geq 2\pi$ giving a frequency resolution of $\frac{f_s}{N}$ Hz. The summation is bounded from 0 to N-1 because this is the equation for a finite duration signal $x[n]$ of length N. This sampled DTFT is a number of impulses in the time domain, which as it turns out is the exact DTFT of a complex exponential or sinusoid. This means that the DFT is a representation of the Fourier Series of a time-domain signal,

with the coefficients of the sinusoids being the heights of each impulse, and its corresponding frequency is shown by the shift along the independent frequency axis. FFT is a set of algorithms to efficiently calculate the DFT, which is extremely important for applications on small low-powered DSP chips, such as the C55x chipset from Texas Instruments.

2.4 How FFT works

Since the variant we were expected to code was DIF FFT, that will be the focus of this section. Since $X(k) = \sum_{n=0}^{N-1} x(n)W_N^{kn}, \quad k = 0, 1, \ldots N-1$ we can split the summation in half as in

$$X(k) = \sum_{n=0}^{\frac{N}{2}-1} x(n)W_N^{kn} \quad + \quad \sum_{n=0\frac{N}{2}}^{N-1} x(n)W_N^{kn}$$

shifting the second portion by $N/2$

$$X(k) = \sum_{n=0}^{\frac{N}{2}-1} x(n)W_N^{kn} \quad + \quad \sum_{n=0}^{\frac{N}{2}-1} x(n+\frac{N}{2})W_N^{k(n+\frac{N}{2})}$$

$$X(k) = \sum_{n=0}^{\frac{N}{2}-1} x(n)W_N^{kn} \quad + \quad W_N^{\frac{kN}{2}} \sum_{n=0}^{\frac{N}{2}-1} x(n+\frac{N}{2})W_N^{kn}$$

remembering that

$$W_N^{\frac{kN}{2}} = e^{-j(\frac{2\pi k(N/2)}{N})} = e^{-j\pi k} = (-1)^k$$

In this we can see that each DFT of length N has been split into two $N/2$ length DFTs, which act differently and can be separated again based on k being even or odd. Since the summations and final twiddle factors are the same, they can be combined in both the even and odd cases. Here we will use a new variable m, where for even cases k will be replaced by $2m$, and for odd cases k will be replaced by $2m+1$.

k even:
$$X(2m) = \sum_{n=0}^{\frac{N}{2}-1} [x(n) + x(n+\frac{N}{2})]W_N^{(2m)n}$$

$$X(2m) = \sum_{n=0}^{\frac{N}{2}-1} [x(n) + x(n+\frac{N}{2})]W_{(\frac{N}{2})}^{mn}$$

$$feven(n) = x(n) + x(n+\frac{N}{2})$$

k odd:
$$X(2m+1) = \sum_{n=0}^{\frac{N}{2}-1} [x(n) - x(n+\frac{N}{2})]W_N^{(2m+1)n}$$

$$X(2m+1) = \sum_{n=0}^{\frac{N}{2}-1} \left([x(n) - x(n+\frac{N}{2})]W_N^n\right) W_{(\frac{N}{2})}^{mn}$$

$$fodd(n) = x(n) - x(n+\frac{N}{2})$$

These are the final equations used when going to the next step, which is the butterfly, as shown below in Figure 1.

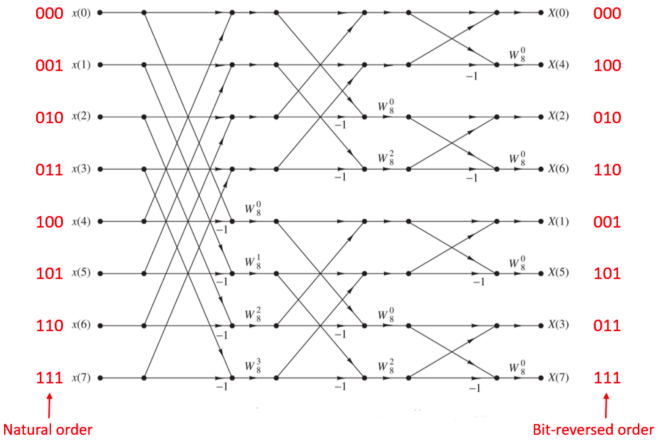

Figure 1. Decimation in frequency butterfly FFT

The example butterfly drawing shows an 8-point FFT, but the numbers scale up to any size, 128 in our case. n will be incremented from $n = 0, 1, 2 \ldots \frac{N}{2} - 1$ in each the even and odd equations of k, giving for even k:

$$feven(0) = x(0) + x(0+4)$$
$$feven(1) = x(1) + x(5)$$
$$feven(2) = x(2) + x(6)$$
$$feven(3) = x(3) + x(7)$$

Which is shown in each of the upward pointing diagonal arrows in the left set of butterflies. The same for odd k:

$$fodd(0) = [x(0) - x(0+4)]W_8^0$$
$$feven(1) = [x(1) - x(5)]W_8^1$$
$$feven(2) = [x(2) - x(6)]W_8^2$$
$$feven(3) = [x(3) - x(7)]W_8^3$$

which is shown below as the diagonal downward pointing arrows in the same column, complete with the -1 multiplier on the bottom term as well as the W_N^n terms. This has generated two four-point halves of the 8 original points (again, in this example using the butterfly diagram shown below in Figure 1, ours started with 128 points, but showing this graphically would be unwieldy. These steps can be generalized to any size FFT, with N changing to the desired length). This two 4-point DFTs can be split as described above into two 2-point DFTs each, for a total of four 2-point DFTs from the 8-point at the start. Clearly in the example of our experiment, this began with 128-point DFT, split into two 64-point, four 32-point, eight 16-point, sixteen 8-point, thirty-two 4-point, and finally sixty-four 2-point DFTs.

As described above, because our experiment used DIF, the final step is to bit-reverse, as this efficient method's minor drawback is it scrambles the order of the outputs. For DIT, the inputs must be scrambled using bit-reversal to yield an output that is in natural order.

2.5 Why FFT matters

The reason why this is so important is the incredible efficiency. To compute DFT directly requires N^2 complex multiplications, whereas FFT algorithm requires only $\frac{N}{2}log_2N$ complex multiplications. For an 8-point DFT, that would require 64 complex multiplications using standard DFT, but only 12 for the FFT algorithm. This savings becomes greater as the N is increased. At 128, which is what we use in our lab, standard DFT requires 16384 complex multiplications, but FFT only 448. 1024-point DFT uses 1,048,576 complex multiplications, with FFT only needing 5120, which is a savings of over 99.5% of the complex multiplications needed, allowing much cheaper, lower-power, and smaller equipment to be used.

2.6 The twiddle factor

There are two major parts to the DFT equations summation, which are the time-domain signal $x[n]$ and the complex exponential, which is called the twiddle factor. This calculation is extremely redundant in calculating an entire DFT by hand and is one of the sources of computational savings in the FFT. This twiddle factor is $e^{-j(\frac{2\pi kn}{N})}$ which can be written as W_N^k, and is merely a mathematical expression describing a rotating vector, which rotates in increments of $1/N$ as n and k increment. For example, if N is 2, then the entire 2π circle on the complex plane would have only two vectors, at 1 and -1. If we increase N to 4, then we get four complex vectors, with values $1, j, -1, -j$, as the circle is being quartered. At an N value of 8, we have the same four as with 4, but we add $\sqrt{2}+j\sqrt{2}, \sqrt{2}-j\sqrt{2}, -\sqrt{2}-j\sqrt{2}, -\sqrt{2}+j\sqrt{2}$ as the circle is being divided 8 times. To calculate the DFT without shortcuts would require calculating this twiddle factor for every value of n which, by hand or computer-aided calculation, would be a waste of time, as the twiddle factor repeats and enjoys many symmetries, so avoiding repeated and unnecessary calculations helps greatly in an efficient algorithm. Our code calculates one loop around the circle, making a single W array that is repeatedly used for all FFT calculations.

2.7 Scaling

We implemented both $1/2$ and $1/N$ scaling to our DIF FFT. When the half scale flag was set to `TRUE`, we applied the half scaling at every butterfly computation. When the reciprocal scale flag was set to `TRUE`, we applied the reciprocal scaling at the end of the whole FFT. We found that they had both identical FFT output and negligibly different IFFT output.

2.8 IFFT and a discovered bug

To obtain the IFFT, we simply took the complex conjugate of our twiddle factor that we conveniently moved inside of our `ifft()` function by adding a negative to the imaginary part.

Unfortunately, we discovered a bug in the given code that at the end of reading one chunk of 128 samples, the `j` variable is offset by one because of the post increment done in reading. That is to say that after reading $0 \rightarrow 127$, the next chunk it reads is $129 \rightarrow 256$, which it is then off by two. By the end, the last chunk it reads will be offset by 13, and will read values that are incorrect afterward. This bug is fixed by placing a single `j--` decrement after reading in the chunk.

When we fixed this bug, we found that we would have the corrrect IFFT output of the chirp signal, as shown in Figure 6. Without fixing the bug, the offset would correcpt the data, yielding Figure 7. Further, we found that in the frequency domain, the unusual impulse sizes and spectral leakage were also fixed, as shown in Figure 5.

```
1  // PROCESS and WRITE FFT
2  for(j = 0; j < (13 * N); j++)       // da
3  ta file has 1664 = 13*128 data
4  {
5      for(i = 0; i < N; i++)
6      {
7          X[i].re = input7_f[j++];    // co
8          nstruct input samples
9          X[i].im = 0.0;
10     }
11     j--; // this is needed otherwise j wi
12     ll be one more than it's supposed to
13 .
14 .
15 .
```

3. Conclusion

We successfully performed discrete Fourier transform of a 128-point radix-2 by implementing decimation in frequency fast Fourier transform and taking the inverse DFT of the spectrum generated by the initial FFT in C code. By doing so, we were able to gain experience utilizing the FFT algorithm and how to implement it with the C55x hardware. We also learned about the benefits of FFT and its efficiency over DFT due to less computations required.

4. Data

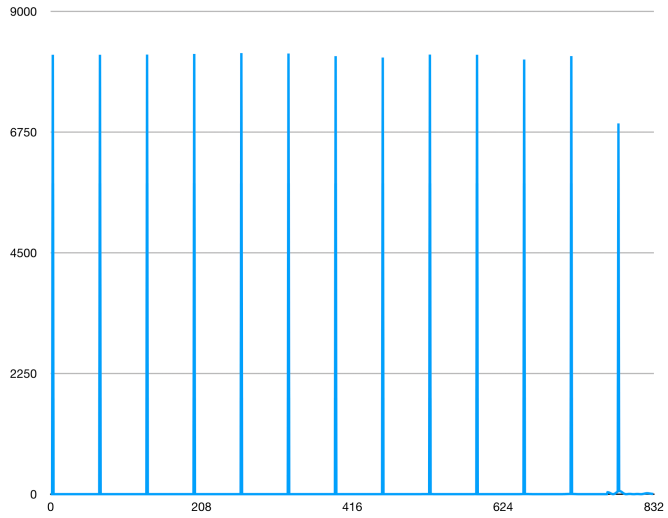

Figure 2. Given FFT output

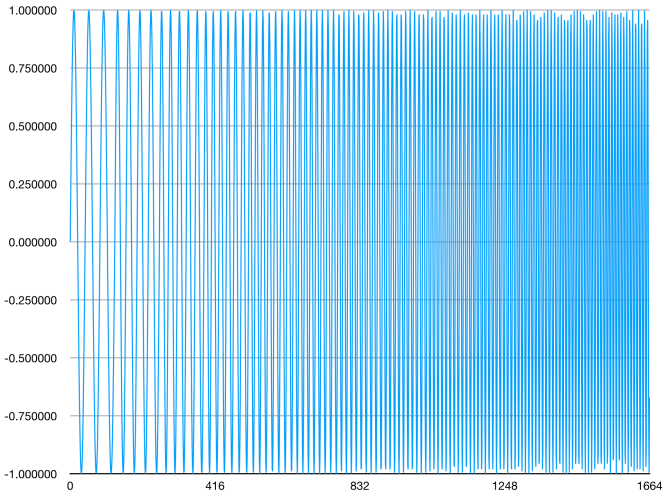

Figure 3. Given time-domain input

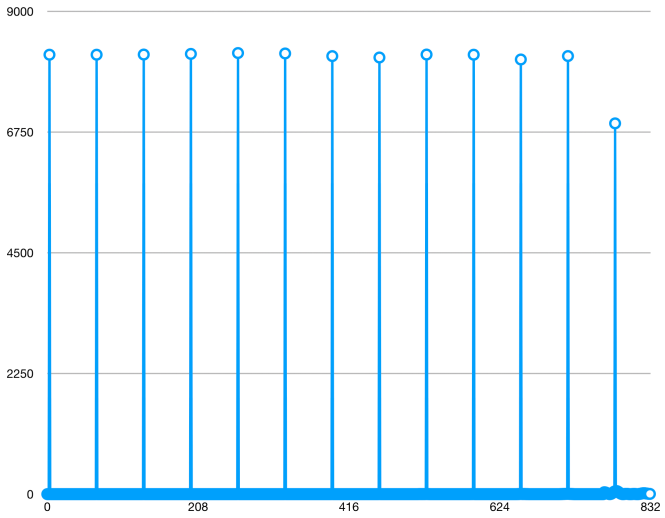

Figure 4. Our FFT output without fixing the `j--` bug

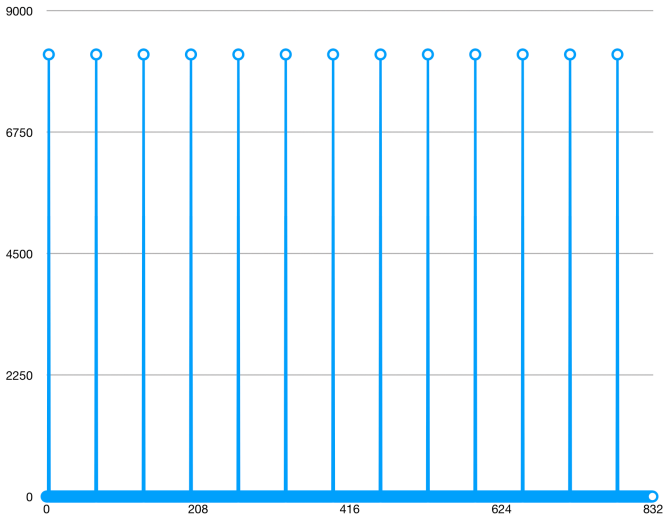

Figure 5. Our FFT output after fixing the `j--` bug

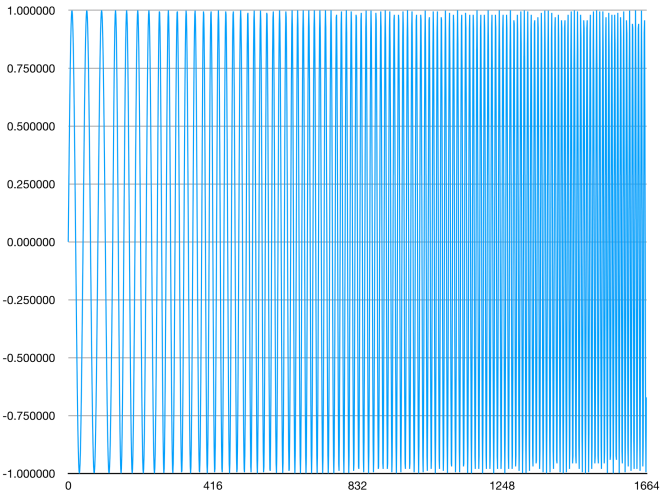

Figure 6. Our IFFT output after fixing the `j--` bug

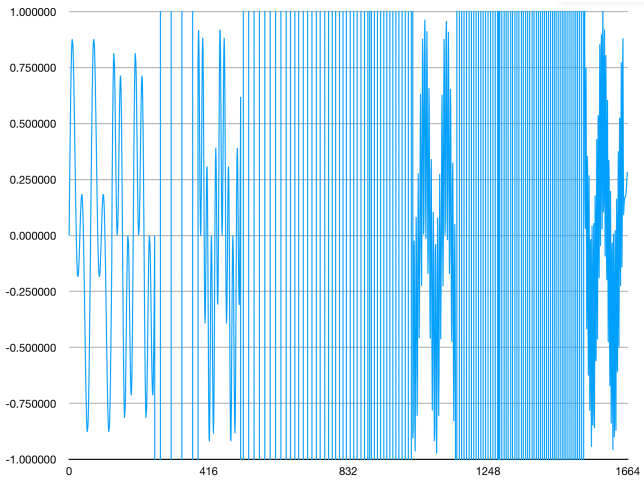

Figure 7. Our IFFT output without fixing the `j--` bug

5. Code

5.1 Testing

```
/*
* float_fftTest.c
*
*  Created on: May 29, 2012
*      Author: BLEE
*
*  Description: This is the 128-point floating-point FFT experiment test
program
*
*  For the book "Real Time Digital Signal Processing:
*                Fundamentals, Implementation and Application, 3rd Ed"
*                By Sen M. Kuo, Bob H. Lee, and Wenshun
*                Publisher: John Wiley and Sons, Ltd
*/


// I changed the fopen call to be in the if statement based on FFT flag.
Still need to open spectrum data and use this as input instead of signal.
Should be
    done then. Also I did twiddle factor fix.

#include <stdio.h>
#include "tistdtypes.h"
#include <math.h>
// #include "fcomplex.h"    // *** Floating-point complex.h header file  //
    arya commented this line out
#include "float_fft.h"
#include "input_f.dat"  // Test data file



// #define isHalfScale      1   // scales by 0.5 at butterfly computation
// #define isRecipScale 0   // scale input samples by 1/N at beginning of
FFT
// #define isOneSided           1   // output one-sided spectrum instead of
2
sided

int main()
{
    complex X[N];       // Declare input array
    complex W[EXP];     // Twiddle e^(-j2pi/N) table
    complex temp;       // generic temporary variable to hold temporary
    values
    int16   spectrum[N];// array of the samples of a (max) 128 sample
    spectrum
    complex allSpectra[N*13];//array of sample of input file, full size
    double   signal[N];
    bool isHalfScale;       // scales by 0.5 at butterfly computation
    bool isRecipScale;  // scale input samples by 1/N at beginning of FFT
```

```
    bool isOneSided;        // output one-sided spectrum instead of 2 sided
    uint16 i, j, L, LE, LE1, k, n; // generic index varabiels. L, LE, and
    LE1
        relate to the bitdepth
    FILE* fpFFT;  // file pointer
    FILE* fpIFFT; // file pointer
    uint16 side = 1;

    // fprintf(fp, "Bin (at each 128-FFT frame)\tFFT spectrum\n"); //for
    debug
    n = 0;
    printf("Exp --- started\n");

    // fpFFT  = fopen("..\\Output\\FFT_spectrum.xls","wt"); // *** for
    windows
    // fpIFFT = fopen("..\\Output\\IFFT_signal.xls","wt");  // *** for
    windows
    fpFFT  = fopen("./Output/FFT_spectrum.xls","wt");   // for mac
    fpIFFT = fopen("./Output/IFFT_signal.xls","wt");    // for mac

    isHalfScale = FALSE;
    isRecipScale = TRUE;
    isOneSided = TRUE;

    // PROCESS and WRITE FFT
    for(j = 0; j < (13 * N); j++)       // data file has 1664 = 13*128 data
    {
        for(i = 0; i < N; i++)
        {
            X[i].re = input7_f[j++];    // construct input samples
            X[i].im = 0.0;
        }
        j--; // this is needed otherwise j will be one more than it's
        supposed to

        // PROCESS FFT
        fft(X, EXP, isHalfScale, isRecipScale); // perform FFT with scale.
        this should not return void and alter array like this. would be
        better
        to return array ptr
        bit_rev(X, EXP);

        for (i = 0; i < N; i++)
        {
            allSpectra[j - N + i + 1].re = X[i].re;
            allSpectra[j - N + i + 1].im = X[i].im;
        }

        // SET sided-ness
        if(isOneSided)
        {
            side = 2;
        }
        else
```

```
        {
            side = 1;
        }

        // WRITE FFT spectrum to file
        for(i = 0; i < N / side; i++)             // verify FFT result. was
        N/ 2 to only get one-sided spectrum
        {
            temp.re = X[i].re * X[i].re;
            temp.im = X[i].im * X[i].im;
            spectrum[i] = (int16)((temp.re + temp.im) * 32767); // this is not
                actual magnitude; This is magnitude squared
            fprintf(fpFFT, "%d\t%d\n", n++, spectrum[i]);
        }
    }

    n = 0;
    isHalfScale = FALSE;
    isRecipScale = FALSE;

    // PROCESS and WRITE IFFT
    for(j = 0; j < 13; j++)
    {
        for(i = 0; i < N; i++)
        {
            X[i].re = allSpectra[i + (N * j)].re;
            X[i].im = allSpectra[i + (N * j)].im;
        }

        // PROCESS IFFT
        ifft(X, EXP, isHalfScale, isRecipScale); // perform FFT with scale
        bit_rev(X, EXP);

        // WRITE IFFT signal to file
        for(i = 0; i < N; i++)            // verify FFT result.
        {
            fprintf(fpIFFT,"%d\t%f\n", n++, X[i].re);
        }
    }

    fclose(fpFFT);
    fclose(fpIFFT);
    printf("Exp --- completed\n");
    return 0;
}
```

5.2 FFT

```
/*
* float_fft.c
*
*  Created on: May 29, 2012
*      Author: BLEE & Arya & Brian & Inessa
```

```
*
*  Description: Floating-point complex radix-2 decimation-in-frew FFT
*               Perform in place FFT the output overwrite the input array
*
*  From the book "Real Time Digital Signal Processing:
*                Fundamentals, Implementation and Application, 3rd Ed"
*                By Sen M. Kuo, Bob H. Lee, and Wenshun Tian
*                Publisher: John Wiley and Sons, Ltd
*/

#include "tistdtypes.h"
#include "fcomplex.h"       // Floating-point complex.h header file
#include <math.h>


#define pi 3.1415926535897

void fft(complex*, uint16, uint16, uint16);

void fft(complex* X, uint16 EXP, uint16 hFlag, uint16 rFlag)
{
    complex  temp1; // Temporary storage of complex variable
    complex  temp2; // Temporary storage of complex variable
    complex W[7];       // Twiddle e^(-j2pi/N) table
    complex watch1, watch2, watch3; // for debug
    complex  U;     // Twiddle factor W^k
    uint16 a, j, i; // a is index for higher point in butterfly. j and i
    are generic loop indices
    uint16 b;       // Index for lower point in butterfly
    uint16 level;   // level is where the stages lie.
    uint16 stageSize;   // Number of points in sub DFT at stage level and
    offset to next DFT in stage
    uint16 reach;       // Number of butterflies in one DFT a stage level.
    Also is offset to lower point in butterfly at stage level
    uint16 bits = EXP;  // reassigned to a less confusing name
    float hScale, rScale;       // scale to be applied
    uint16 N = 1 << bits;// Number of points for FFT
    uint16 stage = 0;   // stage is where the butterfly computations
    actaully criss cross. It is different from the level

    // Calculate Twiddle Factor
    for(level = 1; level <= EXP; level++)
    {
        stageSize = 1 << level;                     //
        stageSize=2^level=points of sub DFT
        reach = stageSize >> 1;                     // number of butterflies
        in sub-DFT
        W[level - 1].re = cos(pi / reach);
        W[level - 1].im = sin(pi / reach);
    }

    // Calculate Scaling Factor
    if(hFlag == 1)
    {
```

```
        hScale = 0.5;
    }
    else
    {
        hScale = 1.0;
    }

    // if(rFlag == 1)
    // {
    //  rScale = 1.0/N;
    // }
    // else
    // {
    //  rScale = 1.0;
    // }

    // OUR FFT, DIF :-)
    for(level = bits; level < 8; level--)   // FFT of length 2^bits
    {
        stageSize = 1 << level;         // stageSize=2^level=points of sub
        DFT
        : Will be 128, 64, 32, 16, ...
        reach = stageSize >> 1;         // Number of butterflies in sub-DFT
        U.re  = 1.0;
        U.im  = 0.0;
        stage = -1;                 // gets incremented to 0 before use

        // for (j=0; j<N; j+=stageSize)     // alternate option for smarter
        butterfly
        for(j = 0; j < reach; j++)
        {
            stage++;
            // for(a=j; a<reach+stageSize*stage; a++) // alternate option for
            smarter butterfly
            for(a = j; a < N; a += stageSize)
            {
                b = a + reach;

                temp1.re = X[a].re + X[b].re;
                temp1.im = X[a].im + X[b].im;

                temp2.re = X[a].re - X[b].re;
                temp2.im = X[a].im - X[b].im;

                X[a].re = temp1.re * hScale;
                X[a].im = temp1.im * hScale;

                X[b].re = (temp2.re * U.re - temp2.im * U.im) * hScale; //
                test?
                X[b].im = (temp2.im * U.re + temp2.re * U.im) * hScale;

                // watch2.re = X[b].re;                         // for debug
                // watch2.im = X[b].im;                         // for debug
                // watch3 = (int16)((temp1.re+temp1.im)*32767); // for debug
```

```
114
115                 // printf("%d -> %d\n", a,b);
116                 // printf("reach: #%d\n", reach);
117                 // printf("b: #%d\n", b);
118             }
119
120             // Recursive compute W^k as U*W^(k-1)
121             temp1.re = U.re * W[level - 1].re - U.im * W[level - 1].im;
122             U.im     = U.re * W[level - 1].im + U.im * W[level - 1].re;
123             U.re     = temp1.re;
124             // printf("-- stage complete ---\n");   // for debug
125         }
126         // printf("\n--- level complete ---\n");    // for debug
127     }
128
129     if(rFlag == 1)
130     {
131         rScale = 1.0/N;
132         for (i = 0;  i<N; i++)
133         {
134             X[i].re = X[i].re * rScale;
135             X[i].im = X[i].im * rScale;
136         }
137     }
138 }
139
140         /* ORIGINAL, GIVEN FFT, DIT
141         for (level=1; level<=bits; level++)     // FFT of length 2^bits
142         {
143             stageSize=1<<level;         // stageSize=2^level=points of sub
144             DFT
145             : Will be 2, 4, 8, 16, 32, 64, 128
146             reach=stageSize>>1;         // Number of butterflies in sub-DFT
147             U.re = 1.0;
148             U.im = 0.0;
149
150             for (j=0; j<reach;j++)
151             {
152                 for(a=j; a<N; a+=stageSize) // Butterfly computations
153                 {
154                     b=a+reach;
155                     temp1.re = (X[b].re*U.re - X[b].im*U.im)*scale;
156                     temp1.im = (X[b].im*U.re + X[b].re*U.im)*scale;
157
158                     X[b].re = X[a].re*scale - temp1.re;
159                     X[b].im = X[a].im*scale - temp1.im;
160
161                     X[a].re = X[a].re*scale + temp1.re;
162                     X[a].im = X[a].im*scale + temp1.im;
163
164                     printf("%d -> %d\n", a,b);
165                 }
166
167                 // Recursive compute W^k as U*W^(k-1)
```

```
                temp1.re = U.re*W[level-1].re - U.im*W[level-1].im;
                U.im = U.re*W[level-1].im + U.im*W[level-1].re;
                U.re = temp1.re;
            }
        }
    */
```

5.3 IFFT

```
/*
* ifft.c
*
*  Created on: May 29, 2012
*      Author: BLEE and Brian & Arya & Inessa
*
*  Description: Floating-point complex radix-2 decimation-in-freq IFFT
*               Perform in place FFT the output overwrite the input array
*
*  Adapted from the book "Real Time Digital Signal Processing:
*                Fundamentals, Implementation and Application, 3rd Ed"
*                By Sen M. Kuo, Bob H. Lee, and Wenshun Tian
*                Publisher: John Wiley and Sons, Ltd
*
*/



#include "tistdtypes.h"
// #include "fcomplex.h"       // Floating-point complex.h header file
#include <math.h>


#define pi 3.1415926535897

void ifft(complex*, uint16, uint16, uint16);

void ifft(complex* X, uint16 EXP, uint16 hFlag, uint16 rFlag)
{
    complex  temp1; // Temporary storage of complex variable
    complex  temp2; // Temporary storage of complex variable
    // complex watch1, watch2, watch3; // for debug
    complex W[7];       // Twiddle e^(-j2pi/N) table
    complex  U;     // Twiddle factor W^k
    uint16 a, j, i; // a is index for higher point in butterfly. j and i are
    generic loop indices
    uint16 b;       // Index for lower point in butterfly
    uint16 level;   // level is where the stages lie.
    uint16 stageSize;   // Number of points in sub DFT at stage level and
    offset to next DFT in stage
    uint16 reach;       // Number of butterflies in one DFT a stage level.
    Also is offset to lower point in butterfly at stage level
    uint16 bits = EXP;  // reassigned to a less confusing name
    float hScale, rScale;       // scale to be applied
```

```
45      uint16 N = 1 << bits;       // Number of points for FFT
46      uint16 stage = 0;           // stage is where the butterfly computations
47      actaully criss cross. It is different from the level
48
49      // CALC Twiddle Factor
50      for(level = 1; level <= EXP; level++)
51      {
52          stageSize = 1 << level;                     //
53          stageSize=2^level=points of sub DFT
54          reach = stageSize >> 1;                     // number of butterflies
55          in sub-DFT
56          W[level - 1].re = cos(pi / reach);          // We chose to use the
57          complex conjugate method instead of negative twiddle factor
58          W[level - 1].im = sin(pi / reach);
59      }
60
61      // PROC the conjugate
62      for (i = 0;  i<N; i++)
63      {
64          X[i].im = -X[i].im;
65      }
66
67      // SET Scaling Factor
68      if(hFlag == 1)
69      {
70          hScale = 0.5;
71      }
72      else
73      {
74          hScale = 1.0;
75      }
76
77
78      if(rFlag == 1)
79      {
80          rScale = 1.0/N;
81      }
82      else
83      {
84          rScale = 1.0;
85      }
86
87      // OUR FFT, DIF :-)
88      for(level = bits; level < 8; level--)   // FFT of length 2^bits
89      {
90          stageSize = 1 << level;         // stageSize=2^level=points of sub
91          DFT
92          : Will be 128, 64, 32, 16, ...
93          reach = stageSize >> 1;         // Number of butterflies in sub-DFT
94          U.re  = 1.0;
95          U.im  = 0.0;
96          stage = -1;                 // gets incremented to 0 before use
97
98          // for (j=0; j<N; j+=stageSize)     // alternate option for smarter
```

```
        butterfly
        for(j = 0; j < reach; j++)
        {
            stage++;
            // for(a=j; a<reach+stageSize*stage; a++) // alternate option for
            smarter butterfly
            for(a = j; a < N; a += stageSize)
            {
                b = a + reach;

                temp1.re = X[a].re + X[b].re;
                temp1.im = X[a].im + X[b].im;

                temp2.re = X[a].re - X[b].re;
                temp2.im = X[a].im - X[b].im;

                X[a].re = temp1.re * hScale;
                X[a].im = temp1.im * hScale;

                X[b].re = (temp2.re * U.re - temp2.im * U.im) * hScale;
                X[b].im = (temp2.im * U.re + temp2.re * U.im) * hScale;
            }

            // Recursive compute W^k as U*W^(k-1)
            temp1.re = U.re * W[level - 1].re - U.im * W[level - 1].im;
            U.im     = U.re * W[level - 1].im + U.im * W[level - 1].re;
            U.re     = temp1.re;
        }
    }
}
```

5.4 Bit reverse

```
/*
* fbit_rev.c
*
*  Created on: May 29, 2012
*      Author: BLEE
*
*  Description: This is the FFT bit reversal function
*               Arrange input samples in bit-reverse addressing order
*               - the index j is the bit reverse of i
*
*  For the book "Real Time Digital Signal Processing:
*                Fundamentals, Implementation and Application, 3rd Ed"
*                By Sen M. Kuo, Bob H. Lee, and Wenshun Tian
*                Publisher: John Wiley and Sons, Ltd
*/

#include "tistdtypes.h"
// #include "fcomplex.h"    // *** Floating-point complex.h header file

void bit_rev(complex* X, int16 EXP)
```

```
{
    uint16 i, j, k;
    uint16 N = 1 << EXP;    // Number of points for FFT
    uint16 N2 = N >> 1;
    complex temp;       // Temporary storage of the complex variable

    for(j = 0, i = 1; i < N - 1; i++)
    {
        k = N2;
        while(k <= j)
        {
                j -= k;
            k >>= 1;
        }

        j += k;

        if (i < j)
        {
            temp = X[j];
            X[j] = X[i];
            X[i] = temp;
        }
        // printf("%d -- %d -- %d\n", i, j, k); // for debug
    }
}
```