# ECE 5630: Programming #3

Due on Thursday, Dec 11, 2014

*Scott Budge 3:00pm*

**Tyler Travis A01519795**

# Contents

**Problem 1** 3

**Problem 2** 4

(a) . . . 7

(b) . . . 8

(c) . . . 8

**Problem 3** 9

## Problem 1

Create the signal flow-graph for the butterfly for a decimation-in-time radix-6 fast Fourier transform (FFT). (Only one stage.)

# Problem 2

In C or C++, write a function that performs the decimation-in-time radix-6 fast Fourier transform (FFT).

**(a)** What are the number of multiplies and adds required to perform a 1296-point DFT? What about a radix-6 FFT?

**(b)** Verify that your FFT works as expected by computing the FFT of 1296 points of a signal created by adding together sinusoids of frequencies at $f = 17.01Hz$, $f = 297.71Hz$, $f = 425.35Hz$, and $f = 2637Hz$. Use a sample rate of 11.025kHz to create the test sinusoids.

**(c)** Plot the magnitude of the FFT output. Which bins have values larger than the others? (Remember that there may be some computation noise in each bin.)

---

Listing 1 shows the first program.

Listing 1: Program 1 - main.cpp

```cpp
// Tyler Travis A01519795
#include <complex>
#include <iostream>
#include <fstream>
#include <cmath>
#include <cstdlib>
#include <cstring>

#define MAX_POWER 4

void fft6(int, int, std::complex<double>*);
void twiddle(std::complex<double>*, int, double);
void bit_reorder(std::complex<double>*, int);

const std::complex<double> WN[] = {1.0, 1.0/2.0+sqrt(3.0)/2.0i, -1.0/2.0+sqrt(3.0)/2.0i,
                                  -1.0, -1.0/2.0-sqrt(3.0)/2.0i, 1.0/2.0-sqrt(3.0)/2.0i};

int main(int argc, char** argv)
{
  const int N = atoi(argv[1]);
  std::ofstream x_dat("../data/x.dat");
  std::ofstream y_dat("../data/y.dat");
  std::complex<double> x[N];
  double freq1 = 17.01/11025;
  double freq2 = 297.74/11025;
  double freq3 = 425.35/11025;
  double freq4 = 2637/11025;
  for(int n = 0; n < N; ++n)
  {
    x[n] = cos(2*M_PI*freq1*n) + cos(2*M_PI*freq2*n) + cos(2*M_PI*freq3*n) + cos(2*M_PI*freq4*n)
  }
  for(int i = 0; i < N; ++i)
  {
    x_dat << x[i].real() << '\t' << x[i].imag() << std::endl;
  }
```

Problem 2 continued on next page. . .

```
  std::cout << "FFT" << std::endl;
  fft6(0, N, x);
  bit_reorder(x, N);
  fft6(1, N, x);
  bit_reorder(x, N);
  for(int i = 0; i < N; ++i)
  {
    y_dat << x[i].real() << '\t' << x[i].imag() << std::endl;
  }
  return 0;
}

void bit_reorder(std::complex<double>* x, int N)
{
  int power, N1, N2, N3;
  int N4 = 1;
  std::complex<double> temp[N];
  for(int i = 0; i < N; ++i)
  {
    temp[i] = x[i];
  }

  for(int i = 0; i <= MAX_POWER; ++i)
  {
    if(pow(6,i) == N)
    {
      power = i;
      N1 = pow(6,i)/6.0;
      if(N1 > 1)
      {
        N2 = N1/6.0;
      }
      else
      {
        N2 = 1;
      }
      if(N2 > 1)
      {
        N3 = N2/6.0;
      }
      else
      {
        N3 = 1;
      }
    }
  }

  std::cout << "power: " << power << std::endl;
  std::cout << "N:  " << N << std::endl;
  std::cout << "N1: " << N1 << std::endl;
  std::cout << "N2: " << N2 << std::endl;
  std::cout << "N3: " << N3 << std::endl;
  int index = 0;
  for(int i = 0; i < 6; i++)
  {
    for(int j = 0; j < N1/N2; j++)
    {
      for(int k = 0; k < N2/N3; k++)
      {
        for(int l = 0; l < N3/N4; l++)
        {
          if(index > N)
            break;

          std::cout << i << ',' << j << ',' << k << ',' << l << '\t';
          if(N1 == 1)
          {
            std::cout << index << "->" << i << std::endl;
            x[i] = temp[index++];
          }
          else if(N2 == 1)
          {
            std::cout << index << "->" << i+j*6+k*N1 << std::endl;
            x[i+j*N1] = temp[index++];
          }
          else if(N3 == 1)
          {
            std::cout << index << "->" << i*N3+j*N2+k*N1 << std::endl;
            x[i*N3 + j*N2 + k*N1] = temp[index++];
          }
          else
          {
            std::cout << index << "->" << i*N4+j*N3+k*N2+l*N1 << std::endl;
            x[i*N4 + j*N3 + k*N2 + l*N1] = temp[index++];
          }
        }
      }
    }
  }
}

void twiddle(std::complex<double>* W, int N, double k)
{
  W->real(cos(k*2*M_PI/(double)N));
  W->imag(-sin(k*2*M_PI/(double)N));
}

void fft6(int in, int N, std::complex<double>* x)
{
  std::complex<double> W, butterfly[6];

  int N1 = 6;
  int N2 = N/6;

  if(in == 1)
  {
    std::cout << "Conj" << std::endl;
    for(int i = 0; i < N; i++)
    {
      x[i] = std::conj(x[i]);
    }
  }
  for(int n = 0; n < N2; n++)
  {
    butterfly[0] = (WN[0]*x[n] + WN[0]*x[N2+n] + WN[0]*x[2*N2+n] + WN[0]*x[3*N2+n] + WN[0]*x[4*N
    butterfly[1] = (WN[0]*x[n] + WN[1]*x[N2+n] + WN[2]*x[2*N2+n] + WN[3]*x[3*N2+n] + WN[4]*x[4*N
    butterfly[2] = (WN[0]*x[n] + WN[2]*x[N2+n] + WN[4]*x[2*N2+n] + WN[0]*x[3*N2+n] + WN[2]*x[4*N
    butterfly[3] = (WN[0]*x[n] + WN[3]*x[N2+n] + WN[0]*x[2*N2+n] + WN[3]*x[3*N2+n] + WN[0]*x[4*N
    butterfly[4] = (WN[0]*x[n] + WN[4]*x[N2+n] + WN[2]*x[2*N2+n] + WN[0]*x[3*N2+n] + WN[4]*x[4*N
    butterfly[5] = (WN[0]*x[n] + WN[5]*x[N2+n] + WN[4]*x[2*N2+n] + WN[3]*x[3*N2+n] + WN[2]*x[4*N
    for(int k = 0; k < N1; ++k)
    {
      twiddle(&W, N, (double)k*(double)n);
      x[n + N2*k] = butterfly[k]*W;
    }
  }
  if(N2 != 1)
  {
    for(int k = 0; k < N1; k++)
    {
      fft6(2, N2, &x[N2*k]);
    }
  }
  if(in == 1)
  {
    for(int i = 0; i < N; i++)
    {
      x[i] /= N;
    }
  }
  if(in == 1)
  {
    std::cout << "Conj" << std::endl;
    for(int i = 0; i < N; i++)
    {
      x[i] = std::conj(x[i]);
    }
  }
}
```

## (a)

What are the number of multiplies and adds required to perform a 1296-point DFT? What about a radix-6 FFT?

### (b)

Verify that your FFT works as expected by computing the FFT of 1296 points of a signal created by adding together sinusoids of frequencies at $f = 17.01Hz$, $f = 297.71Hz$, $f = 425.35Hz$, and $f = 2637Hz$. Use a sample rate of 11.025kHz to create the test sinusoids.

### (c)

Plot the magnitude of the FFT output. Which bins have values larger than the others? (Remember that there may be some computation noise in each bin.)

# Problem 3

Use the Matlab function wavread() to generate the samples of the file galway11_mono_45sec.wav. use your FFT fro 1. aboe, and the frquency-domain fast convolution program and filter from Programming Assignment 2, to filter the sound file. Use a FFT length of 1296 points. The result should be the same as for the last programming assignment. Does the filter remove the high frequency components? Does the processed file sound as you expected? Write out the final reslts in a .wav file for the instructor to listen to.

---

Listing 2 shows the first program.

Listing 2: Program 1 - main3.cpp

```
#include <iostream>
#include <fstream>
#include <vector>
#include <cstdio>
#include <cstdlib>
#include <cmath>
#include <cstring>
#include <complex>

#define MAX_POWER 4
// Filter Length
#define Nf 256
// Length of Signal
#define N 496125
// Sampling frequency
const double Fs = 11025;

const std::complex<double> WN[] = {1.0, 1.0/2.0+sqrt(3.0)/2.0i, -1.0/2.0+sqrt(3.0)/2.0i,
                                  -1.0, -1.0/2.0-sqrt(3.0)/2.0i, 1.0/2.0-sqrt(3.0)/2.0i};

void fft6(int, int, std::complex<double>*);
void twiddle(std::complex<double>*, int, double);
void bit_reorder(std::complex<double>*, int);

int main()
{
  // Input stream for filter
  std::ifstream filterIn("../data/LowPassFilter.dat");
  // Input stream for signal x[n]
  std::ifstream xIn("../data/flute.dat");

  // filter of length Nf = 256
  // Nf*4 for zero padding
  std::complex<double> h[4*Nf];

  // input vairable
  double in;

  // Read in the filter data
  for(int i = 0; i < 4*Nf; ++i)
  {
```

```
  h[i] = 0;
}
for(int n = 0; n < Nf; ++n)
{
  filterIn >> in;
  h[n] = in;
}

// Output streams for the input x int argc, char** argvsignal
// and the output y signalt
std::ofstream x_dat("../data/x3.dat");
std::ofstream y_dat("../data/y3.dat");
std::ofstream H_dat("../data/H3.dat");


// input x signal of length N = 25600
std::complex<double>* x;
x = (std::complex<double>*)malloc(sizeof(std::complex<double>)*N);

// output y signal of Length N = 25600
std::complex<double>* y;
y = (std::complex<double>*)malloc(sizeof(std::complex<double>)*(N+Nf-1));

// Generate input signal x[n]
for(int n = 0; n < N; ++n)
{
  xIn >> in;
  x[n].real(in);
  x[n].imag(0);
  x_dat << x[n].real() << std::endl;
}

//const int nfft = 1024;

int M = 256;
int overlap = M-1;
int nfft = 1296;
int stepsize = nfft - overlap;

std::complex<double> H[nfft];
memcpy(H, h, sizeof(h));
// generate fft
fft6(0, nfft, H);
bit_reorder(H, nfft);
for(int i = 0; i < nfft; ++i)
{
  H_dat << H[i].real() << "\t" << H[i].imag() <<std::endl;
}

std::complex<double> yt[nfft];
std::complex<double> xt[nfft];

int position = 0;
```

```
  while(position + nfft <= N)
  {
    for(int j = 0; j < nfft; ++j)
    {
      xt[j] = x[j + position];
    }

    fft6(0, nfft, xt);
    bit_reorder(xt, nfft);

    for(int k = 0; k < nfft; ++k)
    {
      yt[k] = xt[k] * H[k];
    }
    fft6(1, nfft, yt);
    bit_reorder(yt, nfft);
    for(int j = M-1; j < nfft; ++j)
    {
      y[j-M+position] = yt[j];
    }
    position += stepsize;
  }
  for(int n = 0; n < N; ++n)
  {
    y_dat << y[n].real() << std::endl;
  }
  free(x);
  //free(y);
  return 0;
}

void bit_reorder(std::complex<double>* x, int n)
{
  int power, N1, N2, N3;
  int N4 = 1;
  std::complex<double> temp[n];
  for(int i = 0; i < n; ++i)
  {
    temp[i] = x[i];
  }

  for(int i = 0; i <= MAX_POWER; ++i)
  {
    if(pow(6,i) == n)
    {
      power = i;
      N1 = pow(6,i)/6.0;
      if(N1 > 1)
      {
        N2 = N1/6.0;
      }
      else
      {
        N2 = 1;
      }
      if(N2 > 1)
      {
        N3 = N2/6.0;
      }
      else
      {
        N3 = 1;
      }
    }
  }

  int index = 0;
  for(int i = 0; i < 6; i++)
  {
    for(int j = 0; j < N1/N2; j++)
    {
      for(int k = 0; k < N2/N3; k++)
      {
        for(int l = 0; l < N3/N4; l++)
        {
          if(index > N)
            break;

          if(N1 == 1)
          {
            x[i] = temp[index++];
          }
          else if(N2 == 1)
          {
            x[i+j*N1] = temp[index++];
          }
          else if(N3 == 1)
          {
            x[i*N3 + j*N2 + k*N1] = temp[index++];
          }
          else
          {
            x[i*N4 + j*N3 + k*N2 + l*N1] = temp[index++];
          }
        }
      }
    }
  }
}

void twiddle(std::complex<double>* W, int n, double k)
{
  W->real(cos(k*2*M_PI/(double)n));
  W->imag(-sin(k*2*M_PI/(double)n));
}
```

```
void fft6(int in, int M, std::complex<double>* x)
{
  std::complex<double> W, butterfly[6];

  int N1 = 6;
  int N2 = M/6;

  if(in == 1)
  {
    for(int i = 0; i < M; i++)
    {
      x[i] = std::conj(x[i]);
    }
  }
  for(int n = 0; n < N2; n++)
  {
    butterfly[0] = (WN[0]*x[n] + WN[0]*x[N2+n] + WN[0]*x[2*N2+n] + WN[0]*x[3*N2+n] + WN[0]*x[4*N
    butterfly[1] = (WN[0]*x[n] + WN[1]*x[N2+n] + WN[2]*x[2*N2+n] + WN[3]*x[3*N2+n] + WN[4]*x[4*N
    butterfly[2] = (WN[0]*x[n] + WN[2]*x[N2+n] + WN[4]*x[2*N2+n] + WN[0]*x[3*N2+n] + WN[2]*x[4*N
    butterfly[3] = (WN[0]*x[n] + WN[3]*x[N2+n] + WN[0]*x[2*N2+n] + WN[3]*x[3*N2+n] + WN[0]*x[4*N
    butterfly[4] = (WN[0]*x[n] + WN[4]*x[N2+n] + WN[2]*x[2*N2+n] + WN[0]*x[3*N2+n] + WN[4]*x[4*N
    butterfly[5] = (WN[0]*x[n] + WN[5]*x[N2+n] + WN[4]*x[2*N2+n] + WN[3]*x[3*N2+n] + WN[2]*x[4*N
    for(int k = 0; k < N1; ++k)
    {
      twiddle(&W, M, (double)k*(double)n);
      x[n + N2*k] = butterfly[k]*W;
    }
  }
  if(N2 != 1)
  {
    for(int k = 0; k < N1; k++)
    {
      fft6(2, N2, &x[N2*k]);
    }
  }
  if(in == 1)
  {
    for(int i = 0; i < M; i++)
    {
      x[i] /= M;
    }
  }
  if(in == 1)
  {
    for(int i = 0; i < M; i++)
    {
      x[i] = std::conj(x[i]);
    }
  }
}
```