Han Lin
4711958

MUS421 Design Project

Converting human vocal voice in a pop song to synthesized instrumental music using Synthesis Tool Kit (STK)

Design Objective

1. To generate music score of the melody in a pop song
2. To synthesize the music with the generated music score with (Synthesis Tool Kit) STK.

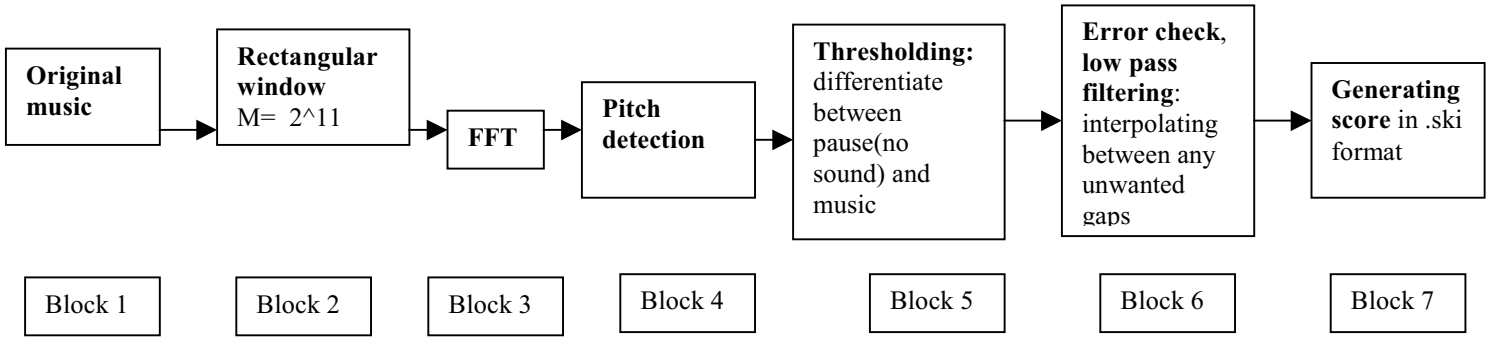
The Product

"Readme.txt ": The program is written in C "pitch.cpp" and is compiled to run under MS DOS environment. The compiled output execution file "pitch.exe" takes in any stereo music file in 16 bit stereo ".wav" format, and by performing Rectangular window, FFT, and pitch detection, produces an output text in ".ski" format which can be read by the STK. USAGE(under DOS command prompt): pitch [input file name .wav] > [output file name .ski] The default input file is "goon.wav" The main purpose of this program is to detect the melody in the pop songs and generate a synthesized music with the same melody. The program only works for pop songs with weak background music such that the melody is dominant. And it has been successfully tested with recorded pure singing voice (myself), and sections of the pop songs such as "My Heart Will Go On – by Cline Dion,"

General Design Strategy

The original music in .wav format is divided into sections by rectangular window. Then for each section, FFT transforms the signal into frequency domain, under which the dominant note/pitch of that section is detected based on peak detection. A threshold based on the fraction of energy of the whole song is used to differentiate between a note and a pause. And extensive error check is then applied to eliminate incorrect pitch due to noise. Finally, low pass filter and zero order interpolation is applied to ensure that there is no short unwanted gap between the notes. Music score based on the detected pitch is generated and the output is in the ".ski" format, which can be read by the STK. Most of the testing is done in Matlab and in C, while the final program is written in C for better performance.

Block Diagram

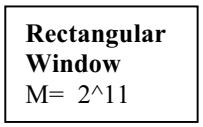


Detailed Design Strategy

Block 1:

Strategy: The original music is in 16-bit stereo .wav file with 44100Hz sampling rate. The C program is designed to read in different variations of .wav format based on the header file. See "wave.h" for more detail.

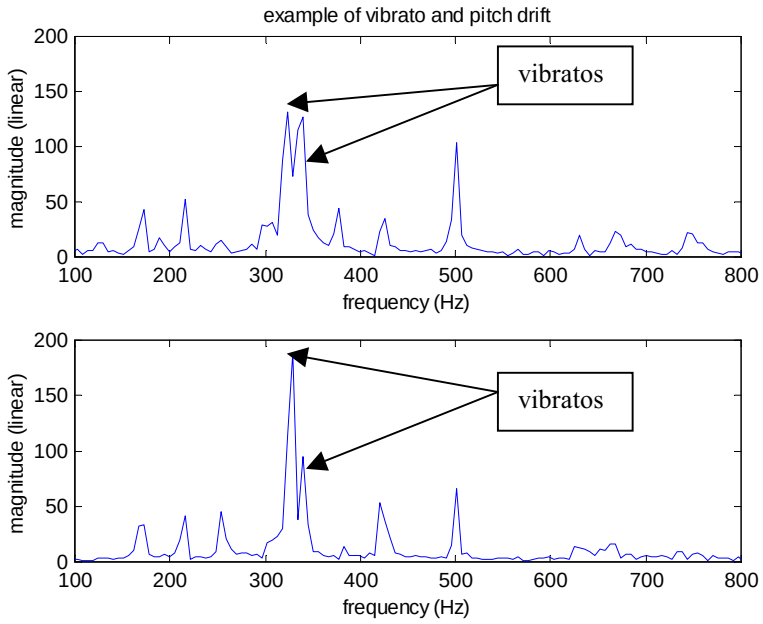
Block 2:



Strategy: Various window types and window lengths are tested empirically. Concerning window types, non-overlap rectangular window works equally well as the 50% overlapping Hanning and Hamming window after empirical testing. Thus, non-overlapping rectangular window is finally implemented to optimize the computational complexity. However, the program has built-in 50% overlapping Hanning, Hamming window functions in case user need to modify the program to perform other acoustic applications, such as adaptive filtering.

Concerning window lengths, window length M is chosen to be powers of 2 so that FFT algorithm can be implemented without zero padding which sacrifices the performance. The window length should be short enough so melody within window length can be considered stationary and to follow the vibratos, and long enough so that it gives us enough resolution for pitch detection in frequency domain.

Examples of window length that is too long (2^13) to resolve vibratos



After extensive testing, M=2^11 and 2^12 gives optimal results. If M= 2^11 is chosen, the duration of each window section is 2^14 / 44100Hz = 0.0464sec and the resolution (delta_f) =21.53 Hz/pt, while M=2^12 will have 0.0928 sec window and resolution 10.77 Hz/pt. The default M length is set to 2^12 since it gives empirically better results for the default wave file "goon.wav", however, user can change the M value easily by changing declaration in the program if following fast vibratos and pitch drift is a concern.

Block 3: FFT

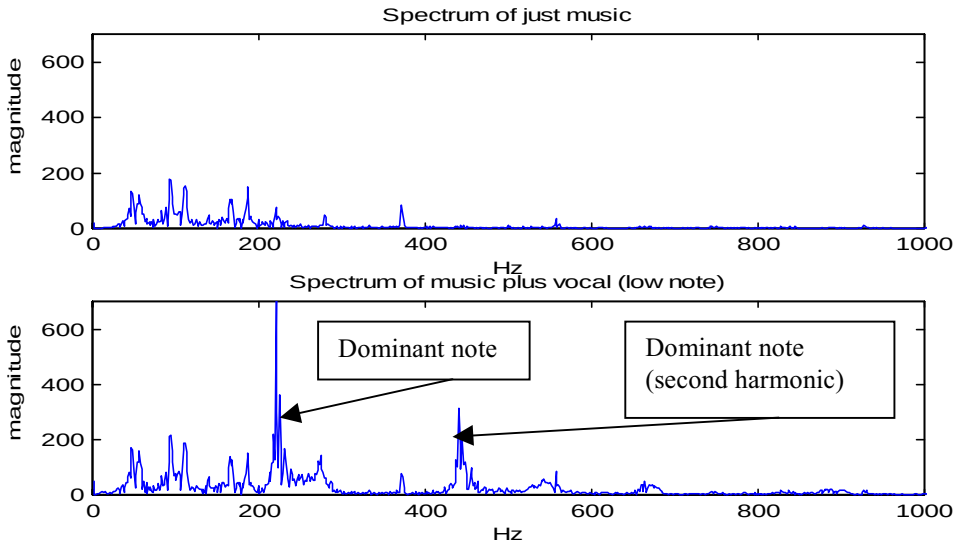
Strategy: Fast Fourier Transform is used to transform signal into the frequency domain. "fft.h" includes both Radix –2 Decimate in Time and Radix –2 Decimate in Frequency algorithm, both implemented with butterfly technique. Radix -2 Decimate in Time method is used in this project. If the window size is not powers of 2, a slower DFT algorithm is used. See "fft.h" for more detail.

Block 4: Pitch detection

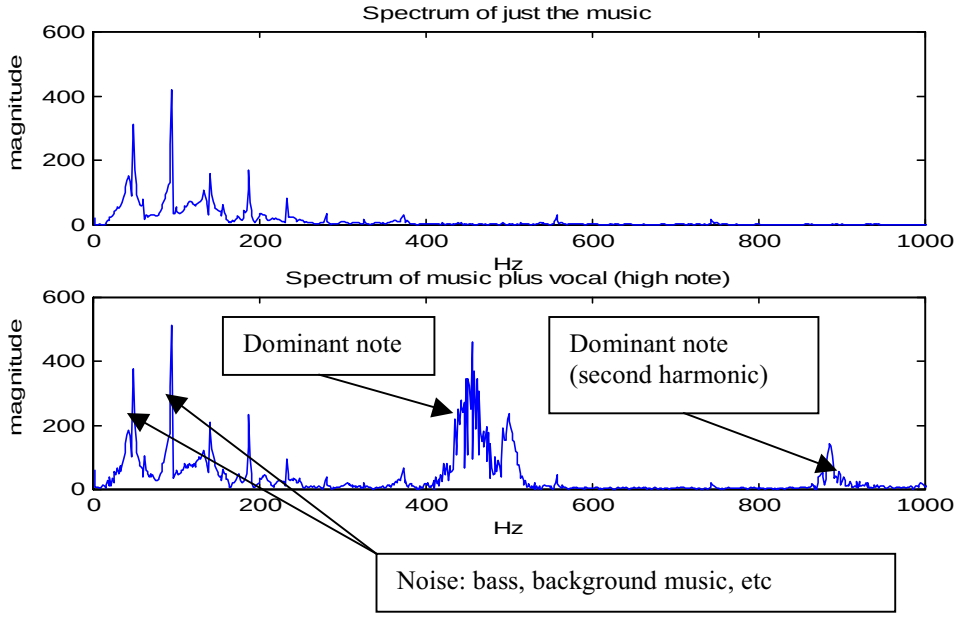
Strategy:

I first use a sample music file that does not contain any vocal part and deliberately adding vocal parts with different pitches to the sample music file. I thus found the following facts. (1) Human vocal usually has dominant notes between 100 Hz to 800Hz. (2) For each note in the window it often contains at least one other dominant harmonic note one octave away. (3) Male voice has more differentiable harmonics while female voice has fewer harmonics. (4)Only songs with soft background music (no strong bass and other instrumental music) can the dominant pitch be easily detectable. Since background music, especially bass sound will contribute to noise in peak detection.

Example of a low female vocal with weak background music (Detectable Pitch)



Example of a high female vocal with strong background music: bass, instrumental, etc. (Undetectable Pitch)



Thus, the pitch detection algorithm simply works as following: to search in the frequency domain from 100 to 800 Hz and find the frequency bin that has the maximum energy.

Block 5:

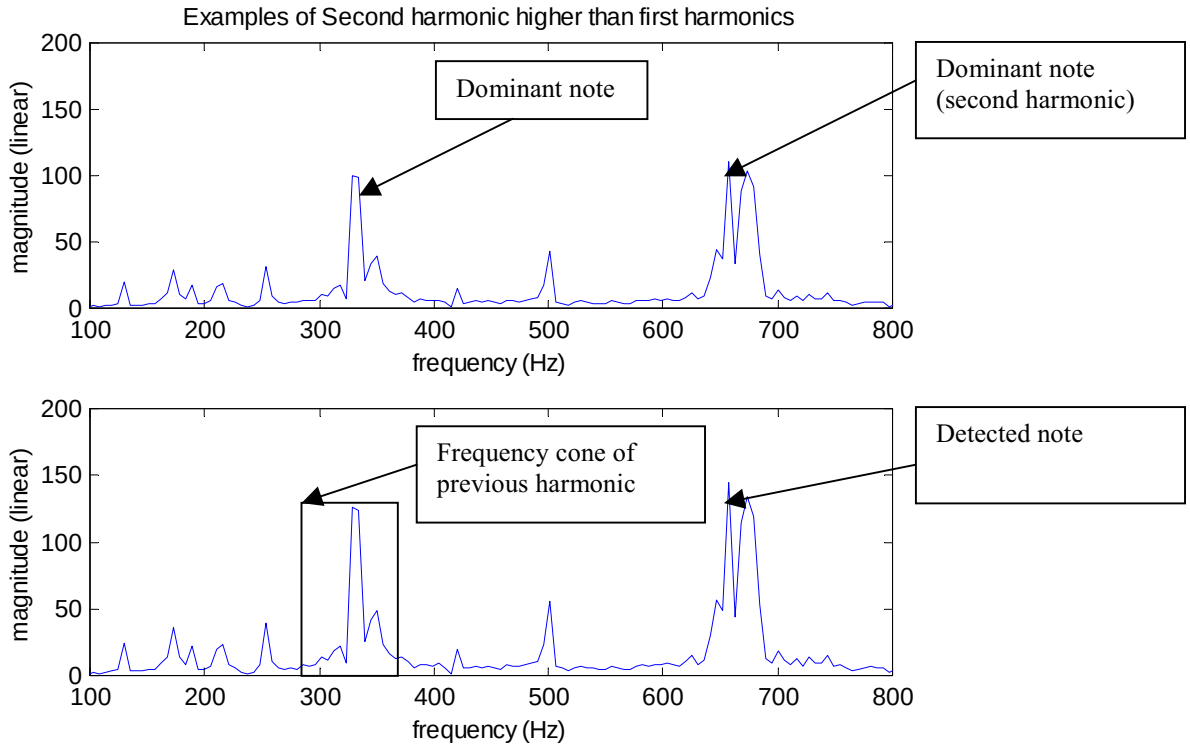
Thresholding: differentiate between pause(no sound) and music

Strategy: In order to differentiate between pause (no sound) and notes (sound), thresholding is used. Based on the Parseval's theorem the total energy in time domain is proportional to energy in frequency domain. Thus, total energy of the whole song in the time domain is first calculated and the mean (average) energy for the frequency domain is determined. Then empirically determined scale times this mean energy is used as a threshold value for differentiating pause (no sound) and notes (sound). In order to avoid any falsely determined notes when the dominate note has lower energy than the background noise; for example, the ending of a note, a relatively high threshold value (200*mean energy) is used. Any falsely determined pause (gaps) is interpolated through the low pass filter which will be described in the later block. The idea is to have falsely determined pause than to have falsely determined note since human is really sensitive to dissonant sound.

Block 6:

Error check, **low pass filtering**: interpolating between any unwanted gaps

Strategy: One of the major errors that can result from peak/pitch detection is that at times, the second harmonic has higher energy than the first harmonic, and the detected note is one octave higher than expected.



Thus, whenever a pitch is detected, the frequency cone around the previous harmonic is checked and if the peak within the frequency cone also has higher energy than the threshold, the lower harmonic becomes the new note. In addition, low pass filtering is performed and any short gaps of pause (no sound) that does not last longer than a few windows (default is 5 which is determined empirically) long is determined to be false gaps. False gaps are interpolated with the previous note (zero order hold).

Block 7:

Generating score in .ski format

Strategy: After all the pitches are detected, the music score of the melody is generated. The detected pitch in Hz is converted to pitch numbers in .ski format, see "notes.h" for detail. Thus, for every window length duration, one pitch number is generated, and pitch number zero correspond to a pause (no sound). The .ski file can be read by the STK to generate synthesized instrumental music such as the clarinet.

Conclusion: The program successfully generates the music score of the melody of the pop song. I used my own recorded singing voice (male) as well as the theme song of Titanic "My Heart Will Go On" – by Cline Dion (female) for extensive testing. As long as the melody dominates the background music, the program gives very accurate results for pitch detection. However, the program seems to be working best for the female vocal voice, which has a higher frequency range, because for the extreme low notes of male voice, we need high frequency resolution (longer window) and thus losing the vibrato. The generated .ski file works well with STK, I played the .ski file with several synthesized instruments on STK, such as plucked, clarinet, flute, etc. all giving pretty prominent results. Just as an example, I generated a few .wav files with the STK corresponding to different variations of the opening section of the Titanic theme song; they will be enclosed in the E-mail.

Concerning the timing performance of the program, I have tested the program with a PC containing AMD 300 CPU, and 64MB of RAM. It takes the program about 13 seconds

to process 15 seconds of .wav file and to generate the score. Although real-time music score generation is not yet implemented, I realized that real time processing is definitely possible, if interacting with the real time functions of STK.

Appendix 1 "pitch.cpp"

```
// This program can read in STEREO 16-bit WAVE format file,
// perform pitch detection, and generate the output .txt file that
//is in the format of .ski and can be read by the Synthesis Tool Kit

#include<stdio.h>
#include<string.h>
#include<math.h>
#include<time.h>

#define WINDOW 4096 //2048
double Window_function[WINDOW];
double sum_bin[WINDOW];
short l_data[WINDOW], r_data[WINDOW];

#include"fft.h"
#include"wave_rec.h"
#include"notes.h"

complex f_data[WINDOW]; // frequency domain data
//complex spectrum[WINDOW]; // averaging
double total_energy, mean_energy;

void make_Window(void)
{
 for(unsigned long i=0; i<WINDOW; i++)
 // Hamming window
//    Window_function[i]=25.0/46-(1-25.0/46)/2*cos(2*M_PI/WINDOW*(i+.5));
 // Hanning window
   Window_function[i]=0.5*(1-cos(2*M_PI/WINDOW*(i+.5)));
}

void report_time(unsigned long timer)
{
 unsigned long hour, min;
 hour=timer/3600; timer-=hour*3600;
 min=timer/60; timer-=min*60;
 printf("\nTotal elapsed time is %02ld:%02ld:%02ld\n", hour, min, timer);
}

void Window_op(short data[]) // take window operation
{
 for(unsigned long i=0; i<WINDOW; i++) data[i]*=Window_function[i];
}
```

```
void pitch_detect(WAVEheader &head) // pitch detection
{

 long i, j, top_pt, bot_pt, max_index, bins;
 char note_char;
 long bin_start, bin_end;
 double note, delta_t, delta_f, top_f, bot_f, mean, int_f,temp, max ;
 double max_energy, energy;
 double ratio, decay=1;
 delta_t=1.0/head.nSamplesPerSec;
 delta_f=1.0/(delta_t*WINDOW);      //5.38
 top_f=800;bot_f=150;
 top_pt=top_f/delta_f;
 bot_pt=bot_f/delta_f;
 mean=0;
 max=0;
 note=10;

 // Deal with the left channel
 for(i=0; i<WINDOW; i++) f_data[i]=l_data[i];
 FFT(f_data, WINDOW);

  for (total_energy=0, max_energy=0, i=bot_pt; i<top_pt; i++)
  {
   energy=abs(f_data[i]);
   energy*=energy;
   total_energy+=energy;
   if(energy>max_energy)
   {
    max_energy=energy;
    max_index=i;
   }
  }
  if(abs(f_data[max_index/2])>abs(f_data[max_index])*.13)
  {
   bot_pt=max_index/4;
   top_pt=max_index*3/4;
  }
  for(max_energy=0, i=bot_pt; i<=top_pt; i++)
  {
   energy=abs(f_data[i]);
   energy*=energy;
   if(energy>max_energy)
   {
    max_energy=energy;
    max_index=i;
   }
  }

  note=max_index*delta_f;


static double last_note=0;
static int nosound_count=0, print_count=1;
```

```
if(max_energy>mean_energy*200) { last_note=note; nosound_count=0; }
{
 nosound_count++;
 if(nosound_count>=5) note=0;
 else note=last_note;
}

if(note>0)
 printf("NoteOn %f 1 %d 127.0\n", delta_t*WINDOW, midi_note(note));
else
 printf("NoteOff %f 1 %d 127.0\n", delta_t*WINDOW, 0);
}

void main(int argc, char **argv)
{
 unsigned long timer=time(NULL);
 char in_filename[80]="goon.wav";
 WAVEheader in_head;
 int in_file;
 int status; // status for file ending, 0: file ends, 1: otherwise
 int step, old_step=0; // progressing steps, 0-100
 unsigned long read_length=0, total_length;
 make_Window();
 if(argc==2) strcpy(in_filename, argv[1]);
 in_file=wave_open(in_filename, in_head);
 if(in_file==-1) { printf("Can't find file \"%s\".", in_filename); return; }
 else if(in_file==-2) { printf("This WAVE file format is not correct."); return; }
 total_length=in_head.data_ckSize;
 total_energy=0;
 unsigned long window_number=0;
 double energy;
 do
 {
  window_number++;
  status=read_window(in_file);
  for(int i=0; i<WINDOW; i++) { energy=r_data[i]; total_energy+=energy*energy; }
 }
 while(status);
 mean_energy=total_energy/WINDOW/window_number*WINDOW;
 in_wave_close(in_file);
 in_file=wave_open(in_filename, in_head);
 do
 {
  read_length+=WINDOW*2*sizeof(short);
  step=100*read_length/total_length;
  status=read_window(in_file);
  pitch_detect(in_head);
 }
 while(status);
 in_wave_close(in_file);
 timer=time(NULL)-timer;
 report_time(timer);
}
```

Appendix 2 “fft.h”

```
/ "fft.h" this program calculated the FFT and IFFT using radix-2
// Decimate in Time and Decimate in Frequency Algorithm


#ifndef __FFT_H
#define __FFT_H

#include<complex.h>

void BitReversion(complex data[], unsigned long N)
{
 unsigned long i, j, k;
 complex temp;
 for(j=0, i=0; i<N-1; i++)
 {
  if(i<j)
  {
   temp=data[i];
   data[i]=data[j];
   data[j]=temp;
  }
  k=N>>1;
  while(j>=k) { j-=k; k>>=1; }
  j+=k;
 }
}

// FFT_Time -- Decimation in Time
// N = 2^M
void FFT_Time(complex data[], unsigned long N, unsigned long M)
{
 unsigned long i, j, k, p;
 unsigned long N2=1;
 complex U, W, temp;
 BitReversion(data, N);
 for(i=0; i<M; i++)
 {
  U=1;
  W=complex(cos(M_PI/N2), -sin(M_PI/N2));
  for(j=0; j<N2; j++)
  {
   for(k=j; k<N; k+=(N2<<1))
   {
    p=k+N2;
    temp=data[p]*U;
    data[p]=data[k]-temp;
    data[k]+=temp;
   }
   U*=W;
  }
  N2<<=1;
 }
}
```

```
// FFT_Freq -- Decimation in Frequency
// N = 2^M
void FFT_Freq(complex data[], unsigned long N, unsigned long M)
{
 unsigned long i, j, k, p;
 unsigned long N2=N;
 complex U, W, temp;
 for(i=0; i<M; i++)
 {
  N2>>=1;
  U=1;
  W=complex(cos(M_PI/N2), -sin(M_PI/N2));
  for(j=0; j<N2; j++)
  {
   for(k=j; k<N; k+=(N2<<1))
   {
    p=k+N2;
    temp=(data[k]-data[p])*U;
    data[k]+=data[p];
    data[p]=temp;
   }
   U*=W;
  }
 }
 BitReversion(data, N);
}

//N2p is the smallest power-of-2 number and larger than 2*N, M=log(N2p)/log(2);
void CZT(complex data[], unsigned long N, unsigned long N2p, unsigned long M)
{
 unsigned long i;
 complex U, *Y=new complex[N2p], *V=new complex[N2p];
 double theta;
 for(i=0; i<N; i++)
 {
  theta=2*M_PI*i*i/2/N;
  U=complex(cos(theta), -sin(theta));
  Y[i]=data[i]*U;
  V[i]=conj(U);
 }
 for(i=N; i<N2p; i++)
 {
  theta=2*M_PI*(N2p-i)*(N2p-i)/2/N;
  Y[i]=0;
  V[i]=complex(cos(theta), sin(theta));
 }
 FFT_Time(Y, N2p, M);
 FFT_Time(V, N2p, M);
 for(i=0; i<N2p; i++) Y[i]=conj(Y[i]*V[i]);
 FFT_Time(Y, N2p, M);
 for(i=0; i<N; i++)
 {
  theta=2*M_PI*i*i/2/N;
  U=complex(cos(theta), -sin(theta));
  data[i]=conj(Y[i]/N2p)*U;
```

```
  }
  delete Y; delete V;
}

void FFT(complex data[], unsigned long N)
{
  unsigned long M=0, N2p=N-1;
  while(N2p) { M++; N2p>>=1; }
  N2p=1<<M;
  if(N==N2p) FFT_Time(data, N, M);
  else CZT(data, N, N2p<<1, M+1);
}

void IFFT(complex data[], unsigned long N)
{
  unsigned long i, M=0, N2p=N-1;
  while(N2p) { M++; N2p>>=1; }
  N2p=1<<M;
  for(i=0; i<N; i++) data[i]=conj(data[i]);
  if(N==N2p) FFT_Time(data, N, M);
  else CZT(data, N, N2p<<1, M+1);
  for(i=0; i<N; i++) data[i]=conj(data[i]/N);
}
#endif
```

Appendix 3 “wave.h”

```
// "wave.h" This program can read STEREO 16-bit WAVE format and
// output STEREO 16-bit WAVE format

#include<io.h>
#include<fcntl.h>
#include<sys/stat.h>

typedef unsigned char BYTE;
typedef unsigned short WORD;
typedef unsigned long DWORD;

typedef struct
{
  char  ckID[4];        /* chunk id 'RIFF'           */
  DWORD ckSize;         /* chunk size                */
  char  wave_ckID[4];   /* wave chunk id 'WAVE'      */
  char  fmt_ckID[4];    /* format chunk id 'fmt '    */
  DWORD fmt_ckSize;     /* format chunk size         */
  WORD  formatTag;      /* format tag currently pcm  */
  WORD  nChannels;      /* number of channels        */
  DWORD nSamplesPerSec;  /* sample rate in hz        */
  DWORD nAvgBytesPerSec; /* average bytes per second  */
  WORD  nBlockAlign;    /* number of bytes per sample */
  WORD  nBitsPerSample;  /* number of bits in a sample */
  char  data_ckID[4];   /* data chunk id 'data'      */
  DWORD data_ckSize;    /* length of data chunk      */
} WAVEheader;
```

```
int format; // 0: not sutff, 1: 14-byte stuff
char stuff[14];

short in_buf[WINDOW*2], out_buf[WINDOW*2];

int wave_creat(char filename[], WAVEheader &head)
{
 _fmode=O_BINARY;
 int f=creat(filename, S_IWRITE);
 if(f==-1) return -1; // fail
 write(f, &head, sizeof(WAVEheader)-8);
 if(format) write(f, stuff, 14);
 write(f, head.data_ckID, 8);
 for(unsigned long i=0; i<2*WINDOW; i++) out_buf[i]=0;
 return f; // success
}

int wave_open(char filename[], WAVEheader &head)
{
 int f=open(filename, O_RDONLY+O_BINARY);
 if(f==-1) return -1; // fail in opening file
 read(f, &head, sizeof(WAVEheader)-8);
 read(f, stuff, 4);
 if(stuff[0]!='d')
 {
  format=1;
  read(f, stuff+4, 10);
  read(f, head.data_ckID, 8);
 }
 else
 {
  format=0;
  for(int i=0; i<4; i++) head.data_ckID[i]=stuff[i];
  read(f, &head.data_ckSize, 4);
 }
 if(head.nChannels==2 && head.nBlockAlign==4 && head.nBitsPerSample==16)
 {
  for(unsigned long i=0; i<WINDOW; i++) in_buf[i]=0;
  read(f, in_buf+WINDOW, WINDOW*sizeof(short));
  return f; // correct format to be read
 }
 else { close(f); return -2; } // different formats
}

int read_window(int f) // read a WINDOW, return 0: file end, 1: otherwise
{
 int status;
 unsigned long i;
 for(i=0; i<WINDOW; i++)
 { in_buf[i]=in_buf[i+WINDOW]; in_buf[i+WINDOW]=0; }
 if(read(f, in_buf+WINDOW, WINDOW*sizeof(short))/sizeof(short)==WINDOW)
  status=1;
 else status=0; // file end
 for(i=0; i<WINDOW; i++)
 {
  l_data[i]=in_buf[i<<1];
```

```
    r_data[i]=in_buf[(i<<1)+1];
  }
  return status;
}

void write_window(int f)
{
  unsigned long i;
  for(i=0; i<WINDOW; i++)
  { out_buf[i]=out_buf[i+WINDOW]; out_buf[i+WINDOW]=0; }
  for(i=0; i<WINDOW; i++)
  {
    out_buf[i<<1]+=l_data[i];
    out_buf[(i<<1)+1]+=r_data[i];
  }
  write(f, out_buf, WINDOW*sizeof(short));
}

int in_wave_close(int f)
{
  return close(f);
}

int out_wave_close(int f, WAVEheader &head)
{
  unsigned length=head.data_ckSize%(WINDOW*sizeof(short));
  write(f, out_buf+WINDOW, length);
  return close(f);
}
```

Appendix 4 “notes.h”

```
#ifndef __NOTES_H
#define __NOTES_H

#include<math.h>
#define C0 16.35159783

/*
Octave||                    Note Numbers
   #  ||
      || C   | C#  | D   | D#  | E   | F   | F#  | G   | G#  | A   | A#  | B
-------------------------------------------------------------------------------
   0  ||   0 |   1 |   2 |   3 |   4 |   5 |   6 |   7 |   8 |   9 |  10 | 11
   1  ||  12 |  13 |  14 |  15 |  16 |  17 |  18 |  19 |  20 |  21 |  22 | 23
   2  ||  24 |  25 |  26 |  27 |  28 |  29 |  30 |  31 |  32 |  33 |  34 | 35
   3  ||  36 |  37 |  38 |  39 |  40 |  41 |  42 |  43 |  44 |  45 |  46 | 47
   4  ||  48 |  49 |  50 |  51 |  52 |  53 |  54 |  55 |  56 |  57 |  58 | 59
   5  ||  60 |  61 |  62 |  63 |  64 |  65 |  66 |  67 |  68 |  69 |  70 | 71
   6  ||  72 |  73 |  74 |  75 |  76 |  77 |  78 |  79 |  80 |  81 |  82 | 83
   7  ||  84 |  85 |  86 |  87 |  88 |  89 |  90 |  91 |  92 |  93 |  94 | 95
   8  ||  96 |  97 |  98 |  99 | 100 | 101 | 102 | 103 | 104 | 105 | 106 | 107
   9  || 108 | 109 | 110 | 111 | 112 | 113 | 114 | 115 | 116 | 117 | 118 | 119
  10  || 120 | 121 | 122 | 123 | 124 | 125 | 126 | 127 |
*/
```

```
// return the MIDI note number by specifying a frequency
unsigned int midi_note(double freq)
{
  return 12*log(freq/C0)/log(2)+0.5;
}

#endif
```