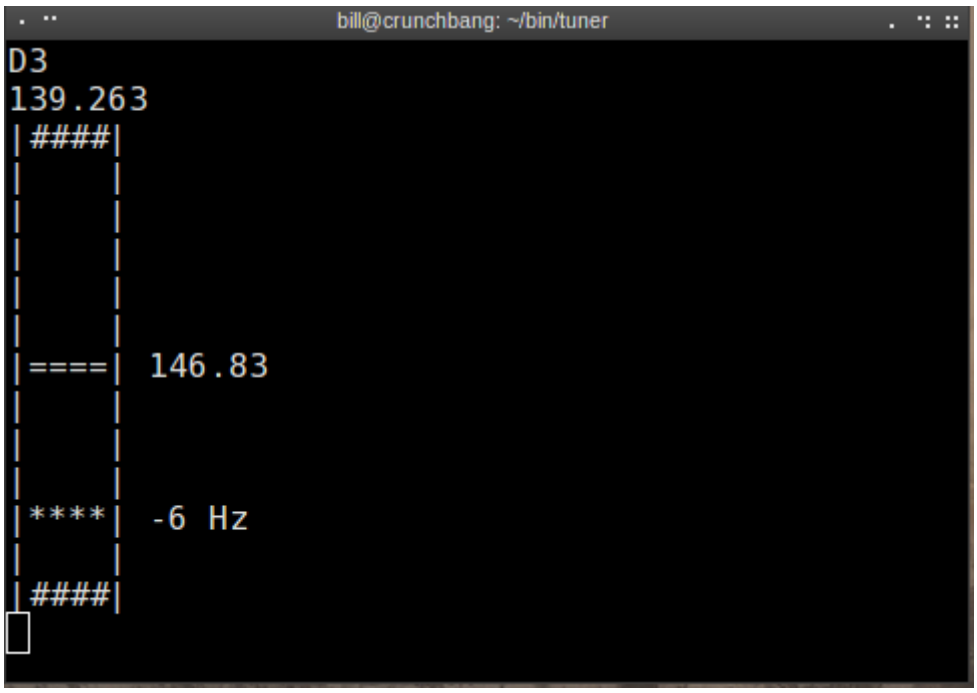
Guitar Tuner for Linux OS

FFT Algorithm Application in C++

Bill Indelicato



Professor Maurice Aburdene

Bucknell University

701 Moore Avenue

Lewisburg, PA 17837

Guitar Tuner for Linux OS

FFT Algorithm Application in C++

Table of Contents

Abstract:

With this project, I sought to understand, implement and utilize the Cooley-Tukey Fast Fourier Transform algorithm coded in C++. The program I wrote to implement this algorithm was a guitar tuner, which communicates with the default audio source on a linux based operating system. Throughout the coding process, I learned specifics relevant to signal processing, coding in C, and the operation of the Linux operating system.

Introduction:

Currently, there is an abundance of operational programs being developed for both the Macintosh operating system and the Windows operating system. Development under Linux is still progressing, yet there are some major gaps within the current available packages for many distributions. As a computer engineer, much of my time is spent at my computer, playing in my Linux sandbox. However, I am also a bassist, who spends all of my free time away from my computer practicing. And as a musician, my biggest qualm about many Linux distributions is their lack of bug-free artistic tools. Linux has become a tool of computer technicians, and it is a shame to see that many distributions are not user-friendly.

This project was an attempt to start filling the artistic software gap on many Linux distributions. The guitar tuner will run on all Linux distributions with the proper dependencies installed. Within the code, the tuner implements a fast fourier transform on an input buffer from the default input device on the running machine.

Body:

The general procedure used in approaching this project is as follows: Read data from the linux machine's default input device into a data buffer. Compute the fast Fourier transform on the audio data in the input data buffer to transform the audio data from the time domain into the frequency domain. Then, normalize the frequency data so that each value corresponds to some real frequency in Hz. Determine the frequency with the maximum magnitude from this transformed data, which will correspond to the fundamental frequency of the sound recorded into the buffer. Then, compare this frequency to an array of musical pitches in order to determine which string is being tuned. Line up the frequency of this constant musical pitch against the frequency of the fundamental determined, and print the note name and difference to the standard output. All of this must occur within a loop until the user signals an interrupt.

The first problem that needs to be tackled in this project is the acquisition of an input signal. On a Unix machine, this input signal must originate from a microphone connected to the sound card on the machine. **O**ne must have an understanding about sound drivers, specifically microphone devices, in order to create a vanilla application. This procedure was both beyond my scope of knowledge and would have taken a significant amount of time to understand and code myself. Therefore, I used an external sound application programming interface (API) to acquire my microphone signal. The API I chose was RtAudio. It is coded with portability in mind (it can be compiled on Windows and Macintosh OS's), and is much simpler to code than the alternatives. Another benefit is that it is written in C++, the language I will be primarily coding in.

In order to use any audio API, the same general procedure is followed (abstractly):

```
open_audio_device();          //pick a device
set_parameters();             //set up options

while(!done)                  //start audio stream
{
        audio_callback_fcn(); //to read or write data
}
close_audio_device();         //free up device for other applications
```

As shown above, the sound card is probed to search for either the default input device or default output device (if none is specified). It then sets up stream parameters that will control the properties of the stream. Then, the stream is started. It will continue until either the stream buffer is full (for input) or until an exception / error is produced. The error handling used in coding audio API's can allow the programmer to control the flow of the program. For instance, the programmer can force an error to determine when to jump out of the stream loop. The audio call-back function is what is called whenever the stream is looking for data to read or write. This is the one function that the programmer has the most control of (see the function [`int record()`] in Appendix I). Finally, the device is closed and opened up to other applications.

Once the input device is properly opened for recording and is properly operational, the next step is to implement the Fast Fourier Transform (FFT). The Fourier Transform is a way of converting data in the time domain into the frequency domain. This will be necessary in order to determine the frequency of a note played into the guitar tuner. The general form of the Discrete Fourier Transform (DFT) used is as follows, where

$$N = number\ of\ samples\ ,\ i = \sqrt{-1}$$
$$x[n] = input\ data\ (time\ domain)$$
$$X[f] = transformed\ data\ (frequency\ domain)$$

$$X[f] = \sum_{n=0}^{N-1} x_n e^{-(ni\frac{2\pi f}{N})}$$

In order to implement this standard definition of a DFT in C++, only two loops are required... in pseudo-code:

```
for(int f = 0; f < num_samples; f++)
{
	for(int n = 0; n < num_samples; n++)
	{
		X[f] += x[n] * exp(f);
	}
}
```

Where exp() is a function that returns the exponential in Eq 1. above.

You will notice that in order to implement an DFT, one must first implement a class in order to handle the complex numbers. C++ has a standard library that handles such numbers. In can be added to the program by the `#include<complex>` preprocessor directive. This will now allow us to tackle the exp() function. Note that $r*e^{i\theta}$ is a complex polar number with length r at an angle of θ.

Theta in this case is $\frac{-n2\pi f}{N}$ The complex header includes a function called polar(r, θ), which converts from polar form to complex form. Then, exp(f) can be defined as follows:

```
for(int f = 0; f < num_samples; f++)
{
	for(int n = 0; n < num_samples; n++)
	{
		X[f] += x[n] * polar( 1, (-2*PI*n*f)/N );
	}
}
```

Where X[f] is initialized to an array of complex numbers of length N, and PI is a preprocessor defined constant.

However, doing these summation for N samples at a high sample rate (44.1 kHz for most recording devices) is highly inefficient. This tuner needs to be able to spit out calculations fast enough to react to a slight change in pitch, or else it will be highly inefficient and useless. This is where the Cooley-Tukey FFT algorithm comes into play.

The most widely used form of this algorithm (called radix-2) works by splitting the odd terms of the DFT from the even terms, as follows:

$$X[f] = \sum_{n=0}^{N/2-1} x_{2n} e^{-[(2n)i\frac{2\pi f}{N/2}]} + \sum_{n=0}^{N/2-1} x_{2n+1} e^{-[(2n+1)i\frac{2\pi f}{N/2}]}$$

$$= \sum_{n=0}^{N/2-1} x_{2n} e^{-[(2n)i\frac{2\pi f}{N/2}]} + e^{-[i\frac{2\pi f}{N}]} \sum_{n=0}^{N/2-1} x_{2n+1} e^{-[(2n)i\frac{2\pi f}{N/2}]}$$

$$= EVEN_f + e^{-(i\frac{2\pi f}{N})} ODD_f$$

Where N = N/2 since there are half as many terms (one even, one odd). This is the asymptote of reflection seen in the plot of frequency vs. time. The terms beyond this point can be calculated

simply by

$$X[f]_{f<N/2} = EVEN_f + e^{-(i\frac{2\pi f}{N})} ODD_f$$
$$X[f]_{N/2<N} = EVEN_f - e^{-(i\frac{2\pi f}{N})} ODD_f$$

Once the terms are split, the code is then implemented even faster by recursion. The base case for this algorithm is when N = 1, when there are no more terms to be calculated. Each recursive call divides the number of samples by a factor of 2 until `num_samples` =1 is reached. This algorithm requires that `num_samples` be some power of 2:

```
complex* FFT( complex x, num_samples )
{
	complex* X = new complex[num_samples];

	if ( num_samples == 1 )                              //base case
	{
		X[0] = x[0];
		return X;
	}
	else
	{
		complex* even = new complex[num_samples/2];
		complex* odd  = new complex[num_samples/2];

		for(n = 0; n < num_samples/2; n++)           //split the data into
		{                                            //even and odd terms
			even[n] = x[2*n];
			odd[n]  = x[2*n+1];
		}

		complex* EVEN = FFT( even, num_samples/2 );        //recursive calls
		complex* ODD  = FFT( odd , num_samples/2 );

		delete [] even;
		delete [] odd;

		for(f = 0; f < num_samples/2; f++)           //combine terms
		{
			ODD[f] = ODD[f] * polar(1, -(2*PI*f)/N));
			X[f]   = EVEN[f] + ODD[f];
			X[f+N/2]= EVEN[f] - ODD[f];
		}

		delete [] EVEN;
		delete [] ODD;

		return X;
	}
}
```

This algorithm increases the speed quite a bit for larger sample sizes. In order to force the number of samples of some given signal (x[n] in this case) simply truncate 0's onto the end of the data, until the nearest power of 2 is reached.

Once the FFT of the data from the microphone is collected, it must be normalized to a

meaningful frequency scale. Right now, the frequencies are separated by some factor determined only by the size of the array, called the frequency resolution. In order to normalize this scale, we must determine the frequency resolution:

$$f_{res} = \frac{Sample\ Rate}{Number\ of\ Samples}$$

In order to normalize the frequencies from the FFT, we need to make an array of frequencies from 0 Hz up to $(number\ of\ samples) \cdot f_{res}$ Hz.

We are looking for the fundamental frequency of the signal. This is simply the largest number in the FFT array. In order to find the maximum, use the following C algorithm

```
#include<cmath>

int max = 0;
int maxi = 0;

for (f=0; f< num_samples; f++) {
        if( abs(X[f]) > max )
        {
                max  = abs(X[f]);
                maxi = f;
        }
}

fundamental = maxi * fres;           //frequency resolution * term
```

The maximum value is acquired from the absolute magnitude of the complex number. Then, the term at which this frequency is found is stored, and multiplied by the frequency resolution to find the fundamental frequency.

It should be noted that in order to get an accurate tuning, the frequency resolution needs to be kept smaller. At lower frequencies, consecutive pitches have frequencies that are only about 5Hz appart, so the frequency resolution needs to be at least 2.5 Hz in order to accurately tune these low frequencies (i.e. for bassists like myself).

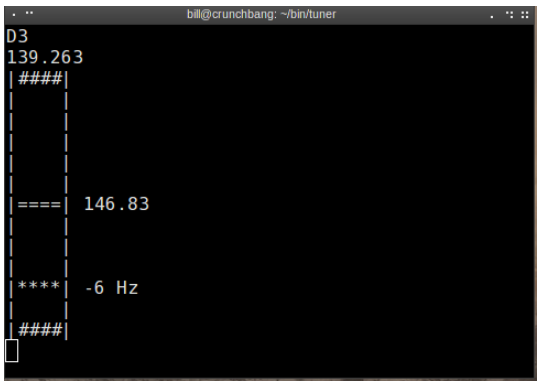
Determining this pitch is the next step of the coding process. Every musical pitch has a corresponding frequency. Each of these pitches also has a note name that corresponds to the octave it falls within. Pitches range from A up to G. A standard five-stringed bass sounds its lowest note at B1, which corresponds to 61.735 Hz. A standard 6 string guitar sounds its highest note at E4, which

corresponds to 329.63 Hz. The tuner must contain accurate information about these pitches in order to determine what note the musician wishes to tune.

Both guitars and basses are tuned in fourths, meaning that every string is 5 pitches away from the last (except the 2nd highest string on a guitar, tuned a major 3rd, four pitches away from the last). In order to determine the pitch being played, the tuner simply compares the fundamental calculated from the FFT to an array of constant pitch values. If the fundamental is within a specified range, that note is determined to be the one to tune.

This entire process from input buffer data acquisition until pitch detection is repeated until the user signals the program to stop tuning. As they tune their instrument, the fundamental frequency of the pitch played will shift until it is within a range of +/- one frequency resolution. At this point, it is declared to be in tune.

In order to display the data, the tuner displays the note played, its frequency, and a sliding scale to determine how close to the constant pitch the frequency is. The closer the user is to the constant pitch, the more precice the scale becomes, allowing for a more accurate tuning. To either side of the constant pitch, there is a 1 Hz gap and four 2 Hz gaps beyond that point. The pitch is declared in tune when it is +/- 1 Hz from the pitch. Here is an example of a D string which needs to be tuned up (frequency increase) by about 6Hz before it is in tune:



Conclusion:

The guitar tuner created runs, albeit a little sluggishly. If a pitch is incorrectly detected every half a second, the application waits until the next half second to try again. This sometimes is frustrating, especially if there is a lot of background noise that interferes with the calculation. Even the fan in a laptop seems to effect the miss rate.

One major frustration I had with the audio API was that at the beginning of every recorded piece of data, there would be a loud click. This click was recorded and the first 200 or so data points of every input buffer I was recording had a maximum at these points. This meant that I lost about 200 data points for every recording, and had to adjust the FFT data to account for this. But with 22050 data points, this did note make the biggest difference.

Changing a guitar's tuners randomly and tuning it with only this application, I was able to get it accurately in tune. Since higher pitches are spaced farther apart, it tunes the higher strings much more accurately. The lowest note on a guitar (82 Hz, E2) is only 10Hz away from the lowest note on a 5-string bass (62 Hz, B1). This constraint determined how I scaled the frequency scale from +/- 10Hz on the tuner interface. At lower frequencies, it is difficult to accurately discern whether a note is a low E2 or a high B1. The solution to this would be to scale the frequency axis for each possible pitch, which I will implement in later versions of this application.

For a recursive function, the DFT I implemented from the Cooley-Tukey algorithm actually runs better than expected. I am able to quickly calculate the FFT and move on to frequency analysis in much less time then recording takes. Yet despite the number of optimizations done to

the initial Discrete Fourier Transform described on page 6, the algorithm was not fast enough to responsively tune to my liking. I tried removing the memory allocations and implementing the code without recursion, but nothing seemed to improve performance. The data acquisition takes approximately half a second of data to transform, but at least a quarter of a second is added onto this value due to the sluggishness of the FFT. The program also eats up processor power, pushing on ahead maxing out the load at all stages. This application should be optimized to run at a slower clock speed, which is beyond my programming knowledge at this point. In later versions, I will attempt to multi-thread the application, so that while one set of data is being recorded, the other can be analyzed. This should eliminate the time it takes for the FFT to be computed, thus increasing the responsiveness of the tuner.

Overall, this application successfully implemented the Cooley-Tukey Fast Fourier Transform and works as expected. With some fine tuning, and possibly the addition of a graphical interface, it will be more reliable, user-friendly and useful.

Appendix I: *main.cpp*

```
#include "RtAudio.h"
#include <iostream>
#include <string>
#include <complex>
#include <cmath>
#include <fstream>
#include<unistd.h>

#define PI 3.14592653589

using namespace std;

typedef signed short BUFF;

struct data{
	signed short* buffer;		//location of data
	unsigned long bufferBytes;	//#of bytes of data
	unsigned long totalFrames;	//number of frames per sample (t * fs)
	unsigned long frameCounter;	//number of current frames used
	unsigned int channels;		//number of channels to record
};


struct note {
	static const float f[];
	static const string p[];
	static const int total;
} notes;


const float note::f[] = {
	61.735,82.407, 110.00, 146.83, 196.00, 246.94,329.63, 440.00, 466.16, 493.88
};

const string note::p[] = {
	"B1","E2","A2","D3","G3","B3","E4","A4"
};
const int note::total = 8;

void clean(RtAudio adc, data d)
{
	if ( adc.isStreamOpen() ) adc.closeStream();
	if (d.buffer) free(d.buffer);
	exit(0);
}

string findNote(float fund)
{
	for (int i = 0; i < notes.total; i++) {
		if (abs(fund - notes.f[i] )< 10 ) {
			return notes.p[i];
		}
	}
	return "0";
}

float findFund(float fund)
{
	for (int i = 0; i < notes.total; i++) {
		if (abs(fund - notes.f[i] )< 10 ) {
			return notes.f[i];
		}
	}
	return 0;
}


complex<float>* FFT( complex<float>* x, int NFFT )
{
	complex<float>* X = new complex<float>[NFFT];

	if ( NFFT == 1 )
	{
		X[0] = x[0];
		return X;
	}
```

```
        else
        {
                complex<float>* even = new complex<float>[NFFT/2];
                complex<float>* odd  = new complex<float>[NFFT/2];
                int n;

                for(n = 0; n < NFFT/2; n++)
                {
                        even[n] = x[2*n];
                        odd[n]  = x[2*n+1];
                }

                complex<float>* EVEN = FFT( even, NFFT/2 );
                complex<float>* ODD = FFT( odd, NFFT/2 );
                complex<float> twiddle;

                delete [] even;
                delete [] odd;

                for(n = 0; n < NFFT/2; n++)
                {
                        twiddle= polar( 1.0, -(2*PI*n)/NFFT );
                        ODD[n] = ODD[n] * twiddle;
                        X[n]   = EVEN[n] + ODD[n];
                        X[n+NFFT/2] = EVEN[n] - ODD[n];
                }

                delete [] EVEN;
                delete [] ODD;

                return X;
        }
}



int record( void *outputBuffer, void *inputBuffer, unsigned int nBufferFrames,
                double streamTime, RtAudioStreamStatus status, void *userData )
{
        data* d = (data*) userData;

        if ( status )
                std::cout << "Stream overflow detected!" << std::endl;

        unsigned int frames = nBufferFrames;        //number of new frames

        //If # new frames is more than it can hold,
        //Only copy in as much as buffer size will hold
        if( d->frameCounter + frames > d->totalFrames ) {
                frames = d->totalFrames - d->frameCounter;
                d->bufferBytes = frames * d->channels * sizeof( BUFF );
        }

        //Copy the new data at the starting position
        unsigned long start = d->frameCounter * d->channels;
        memcpy( d->buffer+start, inputBuffer, d->bufferBytes);

        /*To hear the input:*/
        //memcpy( outputBuffer, inputBuffer, d->bufferBytes);

        //Increment frame counter
        d->frameCounter += frames;

        //Check boundary once more
        if (d->frameCounter >= d->totalFrames ) return 2;

        return 0;
}

float run()
{

        RtAudio adc;
        if ( adc.getDeviceCount() < 1 ) {
                std::cout << "\nNo audio devices found!\n";
                exit( 0 );
        }
```

```
adc.showWarnings( false );
//double t = 1;
int i = 0;
//Initialize ADC parameters
RtAudio::StreamParameters parameters;
parameters.deviceId = adc.getDefaultInputDevice();
parameters.nChannels = 1;
parameters.firstChannel = 0;
unsigned int fs = 44100;
unsigned int bufferFrames = 256;
unsigned long space;
data d;



//Initialize storage space
d.buffer = 0;
d.channels = parameters.nChannels;
d.bufferBytes = bufferFrames * d.channels * sizeof( BUFF );
d.totalFrames = (unsigned long) fs/2;
d.frameCounter = 0;
space = d.totalFrames * d.channels * sizeof(BUFF);
d.buffer = (BUFF*)malloc(space);

//Open the stream and start recording

try {
       adc.openStream( NULL, &parameters, RTAUDIO_SINT16,
                      fs, &bufferFrames, &record, (void *)&d );
       adc.startStream();
}

catch ( RtError& e ) {
       //e.printMessage();
       exit( 0 );
}

while( adc.isStreamRunning() ) {
       sleep( 0 );
}


//Stop the stream

try {

       adc.stopStream();
}
catch (RtError& e) {
       //e.printMessage();
       clean(adc, d);
}

//compute the FFT
int    NFFT = d.totalFrames-200;

float          freq[NFFT];
float          fft[NFFT];
float fres = (float) fs / NFFT;
float temp;


complex<float>* ctins = new complex<float>[NFFT];
for (i = 0; i < NFFT; i++) {
       ctins[i].real(float(d.buffer[i+200]));
}

complex<float>* ctfft = FFT(ctins, NFFT);


delete [] ctins;

//Analyze the frequency
for (i = 0; i < NFFT; i++) {
       freq[i] = ((float)i)*fres;
       //temp = (float) sout[i].r;
       temp = (float) abs(ctfft[i]);
       fft[i] = temp;
```

```
	}


	float		fund = 0;
	int		fundi = 0;
	for (i=0; i< NFFT; i++) {
		if( abs(ctfft[i]) > fund )
		{
			fund = fft[i];
			fundi = i;
		}

	}

	delete [] ctfft;



	//Close stream
	if ( adc.isStreamOpen() ) adc.closeStream();
	if (d.buffer) free(d.buffer);


	return freq[fundi];

}

int main(int argc, const char *argv[])
{
	double freq = 0;
	string end;

	while(cin>>end) {
		freq = run();

		string NOTE = findNote( freq );
		float FUND = findFund( freq );
		if(NOTE != "0")
		{
			system("clear");

			cout << NOTE << endl;
			cout << freq << endl;

			cout << "|####|" << endl;


			if( freq - FUND > 8 )
				cout << "|****|" << " +8 Hz" << endl;
			else
				cout << "|    |" << endl;
			if( ( freq - FUND > 6 ) && ( freq - FUND < 8) )
				cout << "|****|" << " +6 Hz" << endl;
			else
				cout << "|    |" << endl;
			if( ( freq - FUND > 4 ) && ( freq - FUND < 6) )
				cout << "|****|" << " +4 Hz" << endl;
			else
				cout << "|    |" << endl;
			if( ( freq - FUND > 2 ) && ( freq - FUND < 4) )
				cout << "|****|" << " +2 Hz" << endl;
			else
				cout << "|    |" << endl;
			if( ( freq - FUND > 1 ) && ( freq - FUND < 2) )
				cout << "|****|" << " +1 Hz" << endl;
			else
				cout << "|    |" << endl;

			cout << "|====|" << " " << FUND << " Hz" << endl;

			if( ( freq - FUND < -1 ) && ( freq - FUND > -2) )
				cout << "|****|" << " -1 Hz" << endl;
			else
				cout << "|    |" << endl;
			if( ( freq - FUND < -2 ) && ( freq - FUND > -4) )
				cout << "|****|" << " -2 Hz" << endl;
			else
				cout << "|    |" << endl;
			if( ( freq - FUND < -4 ) && ( freq - FUND > -6) )
```

```
                        cout << "|****|" << " -4 Hz" << endl;
                else
                        cout << "|    |" << endl;
                if( ( freq - FUND < -6 ) && ( freq - FUND > -8) )
                        cout << "|****|" << " -6 Hz" << endl;
                else
                        cout << "|    |" << endl;
                if( freq - FUND < -8 )
                        cout << "|****|" << " -8 Hz" << endl;
                else
                        cout << "|    |" << endl;


                cout << "|####|" << endl;
        }

    }

    return 0;
}
```

Appendix II: *Makefile*

```
PROGRAMS = main
RM = /bin/rm
SRC_PATH = .
INCLUDE = include/
OBJECT_PATH = include/
vpath %.o $(OBJECT_PATH)

OBJECTS =      RtAudio.o kiss_fft.o kiss_fftr.o

CC       = g++
DEFS     =   -DHAVE_GETTIMEOFDAY -D__LINUX_ALSA__
CFLAGS   = -g -O2 -Wall
CFLAGS  += -I$(INCLUDE) -Iinclude/
LIBRARY  = -lpthread -lasound -lm

#%.o : $(SRC_PATH)/%.cpp
       ##$(CC) $(CFLAGS) $(DEFS) -c $(<) -o $(OBJECT_PATH)/$@

#%.o : /include/%.cpp
       ##$(CC) $(CFLAGS) $(DEFS) -c $(<) -o $(OBJECT_PATH)/$@
kiss_fft.o: include/kiss_fft.c include/_kiss_fft_guts.h
       gcc -c include/kiss_fftr.c -o include/kiss_fft.o
kiss_fftr.o: include/kiss_fft.c include/_kiss_fft_guts.h
       gcc -c include/kiss_fft.c -o include/kiss_fftr.o


main : main.cpp $(OBJECTS)
       $(CC) $(CFLAGS) $(DEFS) -o main main.cpp $(OBJECT_PATH)*.o $(LIBRARY)

all : $(PROGRAMS)

clean :
       $(RM) -f $(OBJECT_PATH)/*.o
       $(RM)  *.txt
```

References:

[1] G.P. Scavone, The RtAudio Home Page. McGill University. 2010
http://www.music.mcgill.ca/~gary/rtaudio/

[2] J. Trantor, Introduction to Sound Programming with ALSA,
Linux Journal, Issue#126, Oct 2004.

[3] S. G. Johnson and M. Frigo, Implementing FFTs in practice, Fast Fourier Transforms, Ch.11, Rice University, Sept 2008.

[4] S. G. Johnson and M. Frigo, FFTW 3.3 Documentation
http://www.fftw.org/fftw3_doc/

[5] M. Borgeding, Kiss FFT C Library,
http://sourceforge.net/projects/kissfft/

[6] D. Coetzee, Cooley-Tukey FFT Algorithm, LiteralPrograms.org
http://en.literateprograms.org/Cooley-Tukey_FFT_algorithm_(C)