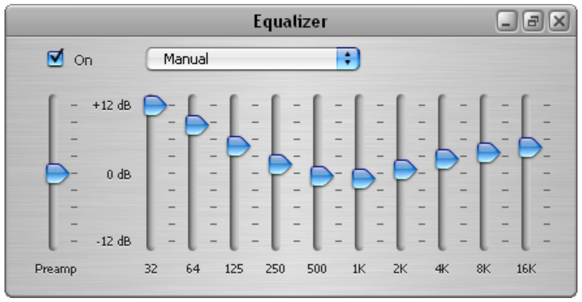
CS314 Lab 3: Java to C

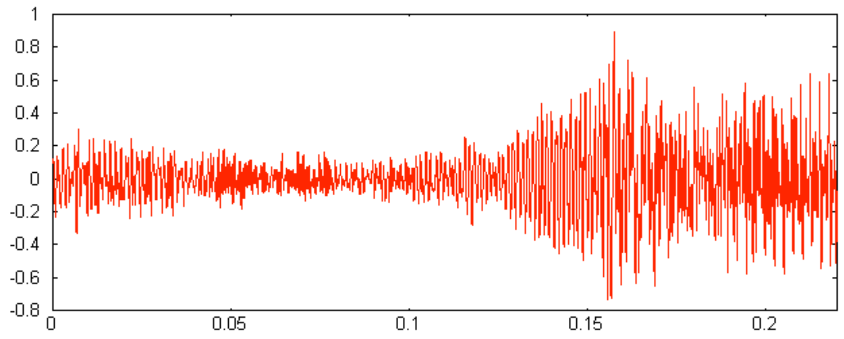
Introduction

Say you're enjoying a little music on your computer while programming. The song you're listening to could use a little more bass so you bring up the graphic equalizer and adjust the values a little

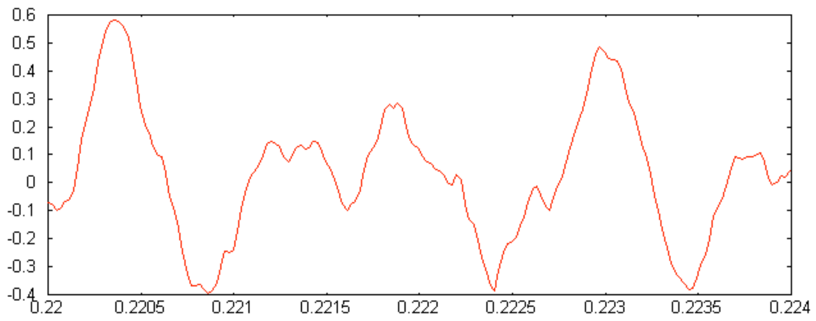


You notice that it takes a fraction of a second for your changes to take affect, and during those milliseconds you wonder, "How does that work?" You keep working, but questions keep coming up. "Changes to the equalizer don't permanently change my song on the disk because I can change the values whenever I want and when I replay the song it doesn't have the previous changes. So it must be doing things on the fly and not modifying the original music file. How does it allow the parts of the song at low frequencies to pass through unchanged, or boosted, while lowering mid or high frequency parts?

Well, a sound file represents a bunch of sound waves. The fact that it's in digital form really doesn't change its nature. It still looks something like this



Up close it looks like a wave



So if you take out the high frequency parts (waves spaced closer together) then how does it not mess up the rest? Old fashioned stereos have knobs for bass and treble that do essentially the same thing, so it can't be that hard -- although those aren't digital. Hmmmm?

Digital Filtering

Well, if you've ever wondered such things then you're in luck. Because after finishing this lab not only will you understand what's going on, but you will have done it! In two different programming languages!

The answer lies in something called *Time Domain Digital Filtering*. The idea is that you have a stream of sound data coming in (the time domain data: one data sample per time interval). That data stream contains many different frequencies, which we hear as ranging between low pitched sound (low frequencies) and high pitches (high frequencies). The human ear can hear sound ranging from roughly 20 Hz to 20 kHz, where a Hertz (Hz) is a cycle per second. A 1 Hz wave would perform a complete wave cycle in one second. A 20 kHz signal would perform one cycle in 1/20,000 of a second (or equivalently 20,000 cycles in one second). When that sound is captured digitally we record the value of the signal as a number and we also record what time it was when we recorded it. The time between samples is important as it tells us what range of frequencies we can store.

For example, CD standard audio is recorded at 44,100 Hz (called 44.1 kHz sampling rate). This means that every 1/44,100th of a second an audio sample is recorded. It turns out that the highest frequency that can be represented in this format is exactly half this number or 22,050 Hz. Very conveniently just above normal hearing range. Also, typical audio is 16 bit stereo. This means that each sample is recorded as a 16 bit integer and we have two samples, one for each channel of stereo (left and right channels).

It is possible to filter this signal by applying an algorithm to it. Say we have an audio signal in an array called *X* of length *N*. We can apply a filter to this data and compute a new filtered array *Y*, by performing the following math:

$$Y[k] = \sum_{n=0}^{N-1} H[n]X[k-n]$$

This is called a 1-D convolution of the data in *X* with the "kernel" in *H*, producing new data in *Y*. The exact values in the array *H* are what determine the behavior of the filter. Variations that exist are

- Low pass filter – Pass low frequencies and reduce high frequencies
- High pass filter – Pass high frequencies and reduce low frequencies
- Bandpass filter – Pass a range of frequencies and reduce all others
- Notch filter – Pass all frequencies and reduce a range
- General filter – Customized and engineered filter response

The kernel, *H*, is typically short compared to the length of the data – somewhere in the tens to hundreds, compared to the audio data which is streaming or millions of samples long. For example, a single 4 minute song from a CD would contain 4*60*44100 = 10,584,000 samples. How big is that by the way? With 2 channels and 16 bit samples it would be 10,584,000*2*2 bytes or a little over 40MB.

For this lab we will write a program to read in an audio data file, filter it, and write a new, filtered, audio file. Some preliminaries first. How do we get an audio file and then how do we get the data out of it? As you're probably guessing, audio files come in various formats: .wav, .mp3, .aac, .ogg, ... These are typically compressed formats and not easy to read without a library. We need a utility to convert these formats into something easy for us to read. Lucky for us there is a good open source program that works on linux and on Windows. It is called Sox and is available at http://sox.sourceforge.net (`sudo apt-get install sox`). Read the help files or the man page (`$ man sox` from the bash shell) to see how it works in detail.

To convert a file called audio.wav, ripped from a CD for example, into a simple binary format that we can read, we need to do this:

```
$ sox audio.wav -r 44100 -s -b 16 -c 1 audio.raw
```

This will output a file called audio.raw that includes only 1 channel of audio, sampled at 44.1 kHz, written to a binary file as signed integer (2's complement) data using 16 bits. There is no header and the resulting file is pure binary data: the first 2 bytes are the first sample, the next 2 bytes are the second sample, and so on.

So all we need to do is read in this binary data, filter it as we read it (by applying the formula to the left) and write it back out in binary format, say as a file called audio_f.raw. Then to convert it back into something we can listen to we just use the sox program in reverse:

```
$ sox -r 44100 -s -b 16 -c 1 audio_f.raw audio_f.wav
```

Then we can listen to our new filtered file. Not exactly the same as streaming audio, but the algorithm is the same.

Low pass Filter

Before we get going on the code, we're missing one important thing. How do we calculate the "kernel" *H*? That depends on what kind of filtering you want to do. For this lab you'll only need to implement a low pass filter. One example of a kernel for a low pass filter is given below. Note that this filter is a rather crude one. There are others that work much better, but they are harder to create. This one will work for us. For a filter of length N

$$H[n] = H[N-1-n] = \frac{\sin(m\lambda)}{m\pi}$$

where, $m = n - \frac{(N-1)}{2}$, $\lambda = \frac{f_c \pi}{f_s}$, and $H[{(N-1)}/{2}] = \frac{\lambda}{\pi}$

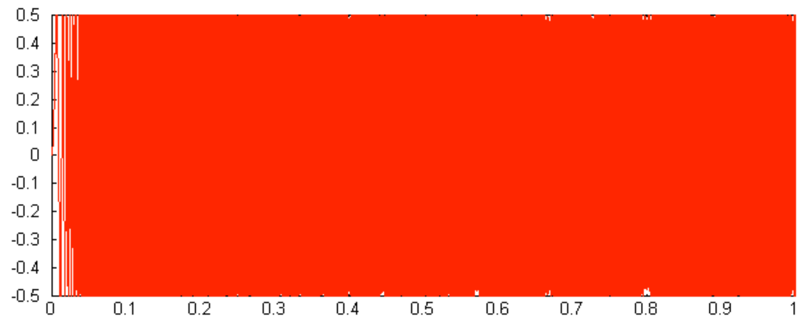
Here, f_c is the "cutoff frequency" and is given in Hz. This is roughly the frequency at which the transition between passed and rejected frequencies lies. You can think of the filter as passing frequencies below this and rejecting those above it. Also, f_s is the "sampling frequency." For everything we're doing this number should be 44100.0 Hz. In addition, these formulas require that the filter length N be odd.

Let's look at some examples. A great way to see how a filter works is to plot the input and the output. I've created a sample input file, called chirp.wav, that has a chirp signal that lasts for one second. This is a sound that starts at low frequencies (20 Hz) and ramps up at a regular rate to high frequencies (20 kHz) by the end of the file. To graph it we need to convert it into a format that is easy for a plotting program to read. Sox takes care of us there as well:

```
$ sox chirp.wav chirp.dat
```

The .dat file is an ASCII file of time versus audio value. Watch out, ASCII output for audio files can be huge! Hundreds of megabytes for a single song!!!

Here's what this file looks like:



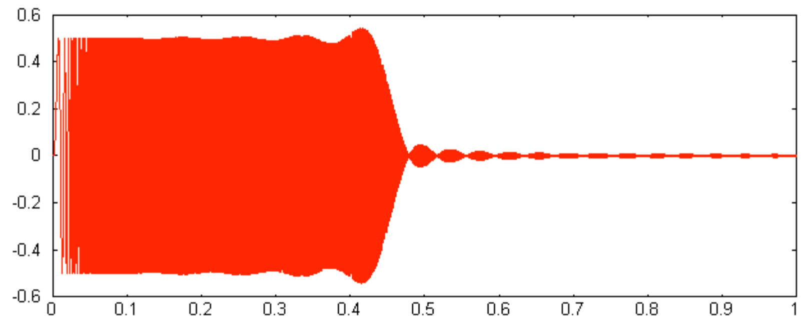
Doesn't look like much here but in GNUPlot (http://www.gnuplot.info/ or sudo apt-get install gnuplot) you can see for yourself:

```
gnuplot> plot 'chirp.dat' using 1:2 notitle with lines
```

If I were to filter this at the halfway point (fc = 10000 Hz) using the provided Java code as follows (sampling frequency 44.1 kHz, filter length N = 101)

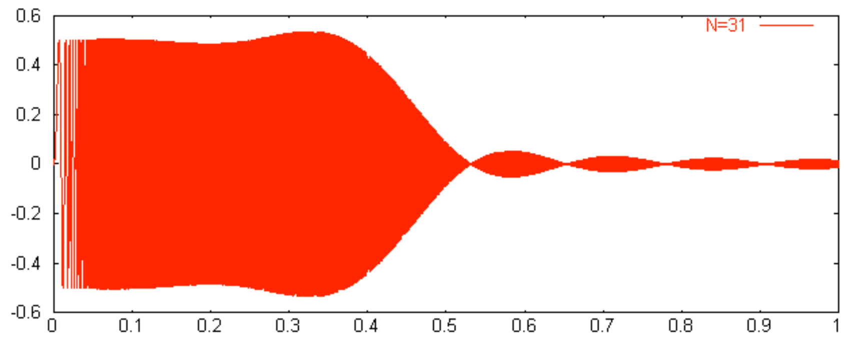
```
> java Lab3 chirp.raw out.raw 44100 10000 101
```

you would get this

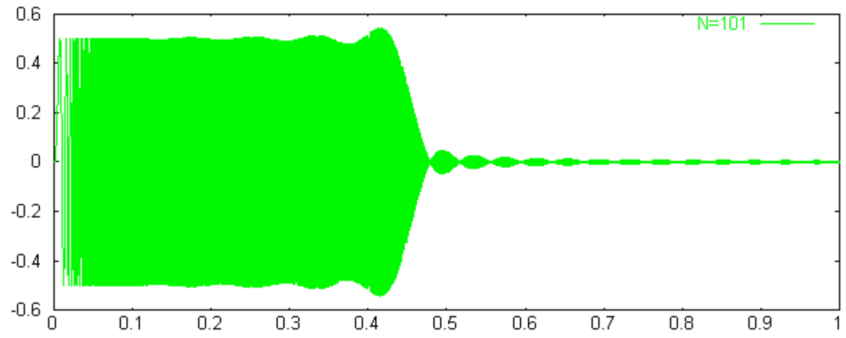


You can see it got rid of the high frequencies and left the low ones. It didn't do a perfect job though. The way these filters work is that generally longer ones do a better job, but take more time and computation to do it. Here are plots with three different filter lengths, all other parameters the same:

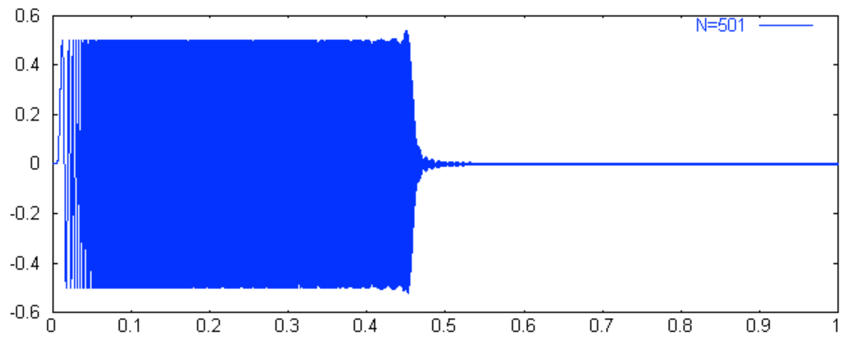
Filter length of N = 31



Filter length of N = 101



Filter length of N = 501



Larger filter lengths do a better job and are more crisp. However the filter length of 501 is nearly 5 times slower than the filter using 101.

I have written the Java code to perform this filtering. Your task is to convert it into C. On the web page are links to all three files as well as some sample input audio files.

Requirements -- Java

Read and completely understand the code. Note comments in the source code concerning Endian-ness. Java does everything in Big Endian. Intel PC's do things in Little Endian. So when we are reading a binary file in Java that was written by something other than Java we must convert from Little Endian to Big Endian. That is what the lines of code are doing involving the bitwise operations. This is specific to Java, and only if you are on a Little Endian machine. If you are running on an old Mac or Sun or some ARM mobile phones, then this Java code will in fact be wrong! Converting wouldn't be necessary.

When you are implementing the C version you are working in the native formats and so no conversions are required.

Requirements -- C

Take the starting code, Lab3.c, and make it work like the Java version. Note that there are only a few comments in the code. **The starter code is an example of reading in a binary file and writing it back out again, without modification.** You'll obviously need to add your own code to perform the filtering.

Specific requirements:

- You must use the `struct run_info`
- You must use function prototypes
- You must write a function to create the filter, that accepts the filter parameters and returns the filter array.
- You must use dynamic memory allocation, using the `malloc` and `free` functions.
- Your code should do its best to safely handle errors.

CS314 Lab 3: Java to C

When you are finished I need two files: Lab3.c and Lab3.c.txt. At the top of each file make sure and include your name. In the source code you must include a modest amount of commenting. **In the Lab3.c.txt file I need copious amounts of comments. Whenever there is a C language (syntax, semantics, library, …) issue that you don't know (and therefore have to look up) then include a comment that tells you, and me, the answer.** For example, let's start at the beginning and take a look at the lines

```
#include <math.h>

#define PI 3.1415926535897931
```

I'm guessing you have an idea of what these do but either don't know the details or aren't comfortable with them. So here is how they should be commented:

```
// This preprocessor directive takes the contents of the
math.h header file and replaces this line with it.  Type
man math on the linux command line to learn more.  It
gives us the sin() function that we need for the filter.

#include <math.h>

Replace the string PI with the string 3.1415926535897931
everywhere in this file before handing to the c compiler

#define PI 3.1415926535897931
```

Do this in the Lab3.c.txt file everywhere something isn't obvious. **You should have lots of comments in this file and they should be instructive about the C language, NOT the algorithm begin performed. Let me say that again, don't comment as you usually do to help the reader follow your code. Comment this as if you're explaining it to someone who doesn't know a thing about C.**