University of Washington
Computer Science & Engineering

CSE 303: Concepts and Tools for Software Development, Winter 2008

CSE 303 Homework 7B
Due: Tuesday, 3/4/08, 11:59PM
Turnin: turnin instruction

FAQ

1. How do I print this assignment?

Print it like any other web page, but selecting landscape mode. An example is here.

1. HW7B Q&A Wiki Page

Teams

You may work in teams on this assignment. See this page for more information.

Overview

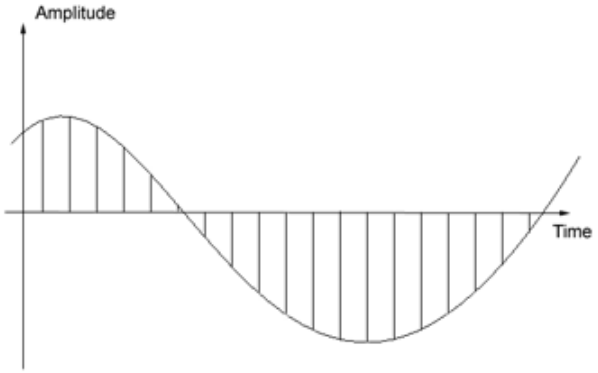
The goals are:

1. Do some more C programming (sound processing routines).
1. Use a significant, realistic library (Sox).
1. Update a simple makefile.
1. Write a shell script.

The application we're going to build is intended to do some simple processing of wave audio files. For the application to make sense, we have to know a little bit about wave files

Wave files

Wave files consist of a header, which describes their contents, followed by audio data. The audio data is simple digital encoding of an audio signal: the amplitude of the audio signal is sampled at regular intervals, and then discretized into an integer.



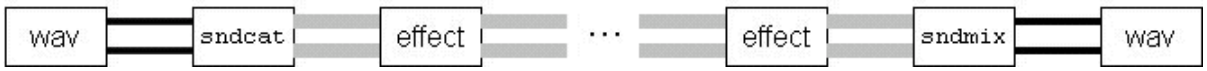
The header describes three properties of interest to us:

- the number of channels - each channel is a separate stream of samples, corresponding (say) to a single microphone
- the sampling frequency - the number of samples taken per second
- sample size (also called bit depth or resolution) - the number of bits stored for each sample. The more bits, the more distinct integer values there are possible, and so the more accurately the integer can represent the actual amplitude of the signal.

Typical values, corresponding to CD quality audio, are 2 channels (stereo), a sampling rate of 44100 samples/second (on each channel), and a sample size of 16 bits. The wave file format is capable of storing data with a large number of choices for each of these characteristics; the CD quality settings are by far the most common, though.

Overall Application Architecture

We're (basically) building an audio processing pipeline. It's basic use looks like this:



We start with a .wav file. The thin, dark lines coming out of it indicate that is has small sample size (16 bits, or about 64,000 distinct values) and that nearly all those values are being used - samples range from about -32000 to about +32000 in

value.

`sndcat` is a particular program you'll write. In this simple use, it increases the sample size (to 32 bits, corresponding to value from between about -2 billion to about +2 billion). Thus the wider lines. Because it is copying values from the wave file, it isn't using much of its potential range. Thus the lighter color.

The sound data now passes through one more effects programs. We will be implementing only three: two kinds of echo, and a volume control. (More on these later).

When done with the effects, the `sndmix` program converts from 32-bit samples back to 16-bit samples. It's output can be written to a new wave file, which can then be played by any of the thousands of programs capable of playing them.

This audio processing pipeline corresponds directly to a shell command line pipeline. For example, it might have come from this:

```
$ sndcat Amy.wav - | sndecho - - 25 .5 | sndallpass - - 30 .4 | sndmix -
Amy-with-echos.wav
```

Each of these programs takes (at least) two file names as the initial arguments (input file followed by output file). We follow a pretty typical Unix convention that the name '-' means stdin or stdout, depending on whether its naming an input or an output. So, the `sndcat` program reads file `Amy.wav` and writes stdout, for instance.

You can also just play the output, rather than write a file:

```
$ sndcat Amy.wav - | sndecho - - 25 .5 | sndallpass - - 30 .4 | sndmix -
- | play -t wav -
```

`play` is a Sox executable, which you installed in hw7A.

The one slight extension to the simple pipeline above is this:

`sndcat` can read an arbitrary number of input files. It combines the channels contained in each into one wave with many channels. (For example, if it combines four stereo wave files, it creates an output that is one wave with 8 channels.) Each of the other modules is capable of dealing with a single input, having any number of channels. At the other end, `sndmix` takes

a wave with 2N channels and turns it back into 2 channels, by summing all the even numbered channels into output channel 0, and summing all the odds into output channel 1.

Sox: Reading/Writing Wave Files

Sox can do many things, the least of which is reading and writing wave files. That's all we're going to use it for, though.

Here are the interfaces we need (defined in `sox.h`, which you installed in `.../cse303XX/usr/include` in part A of this assignment).

```
if (sox_format_init() != SOX_SUCCESS) exit(-1);
```

You must call this before you can use any other Sox methods.

```
sox_format_t* pSoxInFile =
 sox_open_read(inFileName, NULL, strcmp(inFileName, "-")==0 ? "wav" : NULL );
```

Here `inFileName` is the char* name of the file. Normally Sox can guess the type of audio file from the file extension (e.g., '.wav'), but if it's reading stdin that won't work. In that case, the third parameter is a string telling it what the file type is.

To create the output file, you do this:

```
  sox_signalinfo_t outInfo = pSoxInFile->signal;
  outInfo.size = 4;

  sox_format_t* pSoxOutFile = sox_open_write(NULL,
                                outFileName,
                                &outInfo,
                                "wav",  // file type
                                NULL,  // comment
                                0,     // length
                              NULL,  // instr
                                NULL   // loops
                                );
```

(If Sox sees that output file name is "-" it interprets it to mean write to stdout.) Here `outInfo` holds information describing the characteristics of the audio data. `outInfo` is a structure. For the most part, we want the output file to have the same characteristics as the input file (e.g., same number of channels). One thing we might want to force is the sample size. The `size` element of the structure indicates the sample size; 4 means 4 bytes, which is 32-bits (which is the size of an `int` in both C and Java, and is the maximum size Sox allows). All those NULL and 0 arguments are about other things Sox can do that

we're not interested in.

You can now read the input file and write the output file. You read samples into an array, called a buffer. You can then monkey with them as you please (that's what audio effects are, monkeying with the sound sample values), and then write the buffer to the output file. The buffer can be of any length you choose (although ridiculously small values, like 1 element, seem to fail silently). For performance reasons, you probably want at least 1000 or so samples per read; more might be slightly better, but probably not by much.

You can create the buffer using any of the methods you know about (e.g., `malloc()`), but here's one example

```
sox_sample_t buf[BUFSIZE_IN_SAMPLES];
```

Now you're ready to read into the buffer:

```
int len = sox_read(pSoxInFile, buf, BUFSIZE_IN_SAMPLES);
```

The return value is the number of samples actually read, in total (i.e., the sum across all channels). A return value of 0 means EOF. Normally (which is what normally happens), on successive reads you should see BUFSIZE_IN_SAMPLES returned a lot of times, then some positive number smaller than that returned once, then 0. (`man libsox` indicates that SOX_EOF will eventually be returned, but I haven't observed that to be true. SOX_EOF has integer value -1.)

The data in the buffer has this meaning (for the typical case of stereo data), where "C c" means channel c, "S s" means sample number s from channel c, and element 0 of the array is written at the far right:

…	C 1 S 2	C 0 S 2	C 1 S 1	C 0 S 1	C 1 S 0	C 0 S 0
…	5	4	3	2	1	0

Each time you do a `sox_read()` the buffer is filled with the next consecutive set of samples. So, if the buffer is 1000 elements, and the data is stereo, on the second read the 0th element of the buffer holds sample 500 of channel 0, etc.

Each element of the buffer is a `sox_sample_t`, which is a (32-bit) integer. If the file Sox is reading is only 16-bits, those bits are put in the high order positions. Using a decimal example, suppose each element of the buffer can hold numbers of up to 8 decimal digits, but the file samples are only 4 digits each. If a sample in the file is 1234, in the buffer it will be 12340000.

To write data, you do this:

```
int len = sox_write(pSoxOutFile, buf, numSamplesToWrite);
```

The length returned is the number of samples written. Unless there's some grievous and unusual error when Sox tries to write, it should equal numSamplesToWrite.

When you're all done, you do this:

```
sox_close(pSoxInFile);
sox_close(pSoxOutFile);
```

Audio Problems

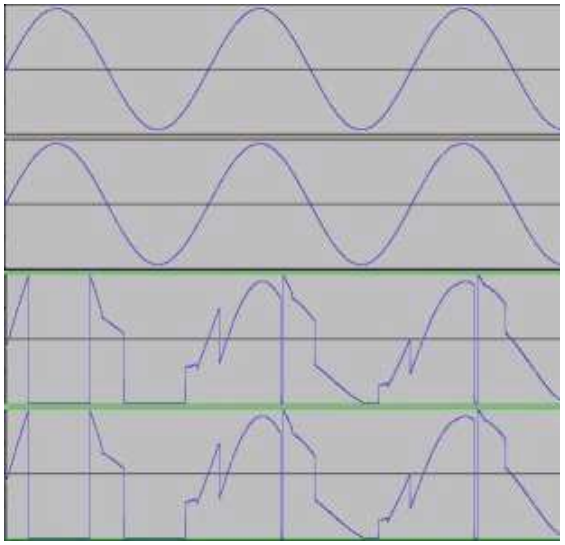
The digitization of sound presents a couple of problems that you'll run into immediately in this assignment. These result from the limited range of values available for recording the sound sample's amplitude. The two problems are:

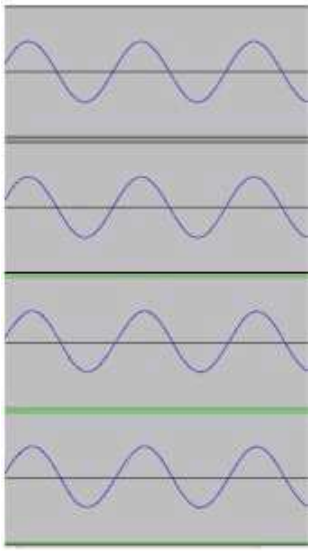
1. loss of resolution
1. clipping

Suppose the sound file allows samples of four decimal digits, so the range [-9999, 9999]. When taking the samples, an arbitrary gain is applied to the analogue signal - it is multiplied by an arbitrary real number. If the gain is set so that the maximum amplitude is about 9999, the sound file distinguishes about 20000 distinct amplitude values. If the gain is set so that the maximum is about 100, there are only about 200 distinct values. Obviously, the latter is a much coarser approximation to the true sound signal, and so it sounds worse when used to play back.

That was the issue of resolution - how many of the distinct amplitude values available are actually being used? In general, we want to use all of them, because the result sounds best.

At the same time, something very bad happens if the maximum amplitude is too large to fit in the available digits - clipping. Suppose, in the same example, the gain is set so that the maximum amplitude is 10500. Samples taken when the amplitude is greater than 9999 have some bad value -- they might be taken as 9999, or they might be some large negative number. Here's an example of a wave file before processing (the sine wave on top) and after adding an echo effect (the bottom). On the left, the original signal has high enough peak values that integer overflow occurs when processed - this is clipping. On the right, the smaller peak values allow the echo processing to complete without integer overflow -- no clipping. (Because the input is a sign wave, it's difficult to see that the processing has done anything. Don't worry that the two look about the same - interesting sounds aren't simple sine waves.)





It probably isn't surprising that clipping is very, very noticable on playback. You'll hear some clicking or popping, or worse.

The trick, then, is to maximize the resolution while avoiding clipping. Some of the functionality of the audio processing components you're building is entirely motivated by this tradeoff.

What To Implement

There is a page for each component to be implemented.

Mixing

1. `sndcat`
   Combines N waves files into 1 (with many channels).
1. `sndmix`
   Converts a single wave file with many channels into a single wave file with 2 channels.

Effects

1. `sndgain`
   A volume control.
1. `sndscale`
   A smart volume control.
1. `sndecho`

A simple echo loop.

- `sndallpass`

  A better behaved echo.

What Else To Write

1. You should maintain the initially distributed `makefile` so that it builds your applications. The command '`make`' should build all applications. The makefile should have a `clean` target and a `depend` target. Other aspects of the make file should follow the conventions shown by the samples discussed in class.

2. You should create a bash script, called `examplePipe`, that processes sounds files and creates what you think is a pleasing effect, at least on many audio files. It would be natural to adapt the `listen` script to do this. I expect your changes will be about what the effects pipeline looks like, and what the parameters for each effect are. This is intended to be some modest experience with shell scripts, not a test of writing sophisticated scripts, so you just need to demonstrate you can make changes to an existing script. (Plus, it's fun to try out different effect combinations, and the ones in `listen` are complete hacks.)

What You Get

- Directory `/cse/courses/cse303/08wi/hw7/bin` contains executables for `sndallpass, sndcat, sndecho, sndgain, sndmix,` and `sndscale`.
- It also contains a shell script, `listen`, that is invoked like this:

  `listen` *`sndXXX size filename`*

  where *sndXXX* is the name of an effect executable (e.g., sndallpass or sndecho); *size* is one of 'small', 'medium', or 'large' (without the quotes); and filename is the name of a wave file. It will play the file, applying some nearly random combination of effects to it. The *size* parameter selects which random combination to appply.
- That directory also contains `gentest`, a program that generates a one second long wave file with very simple data. It is invoked like:

  ```
  gentest outfilename impulse
  ```

  or

  ```
  gentest outfilename sine frequency
  ```

  In the first case, a single full amplitude sample is generated on each of the two channels, with other samples being zero. In the second case, a full amplitude sine wave is generated.
- There is also executable `sndreport`, invoked like:

  ```
  sndreport inputfile outputfile
  ```

  and following the convention that '-' means stdin/stdout. Its purpose is to report the factor by which the amplitudes in a sound file can be scaled up before clipping occurs. That value is in the rightmost column of what it prints. (It prints to stderr.) The first column is the number of samples in the file. (The other two interesting columns are the absolute value of the largest magnitude sample, and that value relative the largest that can be represented in the file.)
- Finally (for that directory), there is a program invoked like this:

  ```
  snddiff file0 file1
  ```

  Compares the header and sample data values of the two files and reports whether they are the same or different. *This program has not been robustly tested.* If you hand it two files with different numbers of channels, for instance, it should tell you that the number of channels is different, but it's behavior after that is undefined.

1. Directory `/cse/courses/cse303/08wi/hw7/src` contains a makefile, a truly minimal skeleton for `sndcp.c`, and a `.h` file whose purpose is just to avoid having to type multiple system includes into each source file you write -- you instead just include this one. (`sndcp.c` is an example of doing this.)
   You MUST edit the makefile before you can expect it to work. Instructions are given in the comment at the top (item #1, in particular).

Suggested Implementation Roadmap

See this page.

Programming/Debugging

1. A full set of implemented modules is in (I mean, "will be in") `/cse/courses/cse303/08wi/hw7/bin/`. They can be used to fill in pieces of audio processing pipelines you'd like to run until you get each of those pieces working yourself. You could make this "substitution" happen transparently by putting your project directory ahead of that directory in your PATH variable. It might be less error prone, though, to just give the full path name for the solution executables when you want to use them.
1. Your installation of Sox installed some man pages (see HW7A). Like nearly all open source projects, the documentation doesn't seem to be 100% reliable on the details. It can help explain the overall structure and use, but you might want to take the few minutes required to verify the particulars with a little test.
1. Sox itself can play audio files. When it does, it provides some information about their encoding.
1. Audacity is an open source sound processing application. It will let you view and play sound files. (Like Sox, it includes all the effects we are implementing.) At as of the time I'm writing this, I haven't been able to get it installed on any Lab machines. It is easy to install on your own machine (where you can be root, or its Windows equivalent).
1. There are other applications like Audacity out there. If you know of one that you think would be generally useful, please add a pointer to it to the assignment Wiki.
1. `gentest` generates some simple sound files. Processing them and viewing the results (e.g., with Audacity) can help point out problems. (For example, at one point I had a bug and wasn't applying echo to the right channel. I didn't notice when I listened to the result, but I could see it in Audacity.)

How Will We Evaluate Your Code?

We will look at both form and function. For function:

1. We can automate testing of your makefile. We can automate testing whether or not your applications run. We can automate running your `examplePipe` script and listen to the results.
1. We can also run `snddiff` to compare your results with ours. We expect to get matches on the output sample rate, sample size, and number of channels. For most everything else (e.g., the length field stored in the header, and the samples themselves), its not unexpected to see some differences, because exactly what happens (especially at the beginning and end of the file) depends on specifics of the implementations at a level below any notion of "correct."

   All that's to say that we will look at things we should agree on, (like the number of channels, etc.), and we will listen to the output as a test of obvious problems, but we won't try to do bit-by-bit comparisons of your sound samples with ours.

1. Finally, we can measure how long it takes your code to complete some audio pipeline for some reference input. This isn't a speed contest, so anything reasonable is okay. Egregiously long times are bad, though.

For form, we'd like to see reasonably clean code. What that means is hard to define. Many, many conditionals (e.g., if statements) always make code complicated, and so should be avoided if at all possible. Many, many comments always make code unreadable, so should be avoided - comment only things you think won't be immediately understandable by someone fluent in both C and the problem your code is solving. As with run time, don't obsess over this; we won't. Try not to write horrible code, though. (If you're unsure whether or not your code is horrible, come talk with us.)

More Information

Much of this MIT Master's Thesis is readable, and skimable. If (like me) you know nothing or next to nothing about transforms, you can still get some useful information out of it by glossing over those parts. This reading is entirely optional, though, and is provided just in case you're interested.

As with most everything, there is a lot of information on the web. And, as with most everything, different sources contradict each other in ways big and small. The general ideas are consistent, from what I've seen. For this assignment, the definitive definitions of everything are the ones contained in this assignment.

Extra Credit

Once everything in the basic assignment is implemented, you can receive extra credit for one or the other (but not both) of these:

1. Implement one of the diffuse reverberators shown on page 56 of that Master's thesis. You can simply leave out the LPF (low pass filter) component in doing this.
   This is worth points up to 5% of the regular assignment points.
1. Implement nested filters in a way that is sufficiently general that it is easy to create new reverators that use them. You do not have to provide a language for the invoking user to specify the reverberator - you can create it in C code, so long as doing so is basically trivial. (This probably means that the specification is some kind of data - an array of structs, say -- and that the executable code itself knows nothing about any particular reverberator.)
   Now implement all three diffuse reverberators on page 56. (Implementing the LPF components would be nice, but is optional.)
   This is worth points up to 10% of the regular assignment points.

Some extra credit might be possible for other things you might be more interested in doing -- creating more sophisticated effects, for instance. (Anything in the frequency domain class of effects certainly qualifies.)

More on Turnin

1. Please do a `make clean` before doing turnin.
1. Do not turnin any sound files.

Computer Science & Engineering
University of Washington
Box 352350
Seattle, WA 98195-2350
(206) 543-1695 voice, (206) 543-2969 FAX
[comments to zahorjan at cs.washington.edu]