ComS 229 Project 1 (Part 1)

Tools for digital sound

September 2, 2013

0 Introduction

Sound is the oscillation of pressure that propagates in the form of a wave. Sound waves are transmitted through some medium (solid, liquid, or gas); they cannot travel through a vacuum. The human ear detects the pressure changes and decodes this into what we hear. Sound can be converted into an electrical signal using microphones or other transducer, by which pressure changes become voltage changes. The reverse, converting voltage changes into pressure changes, can be done via speakers. An analog audio signal may be visualized by plotting the voltage as a function of time (e.g., using an oscilloscope). Note that the voltage level and time are continuous quantities. An example of this is shown in Figure 1.

Pulse–code modulation (PCM) is one method for digitally representing an audio signal, and is the standard used for compact discs (music CDs) and computer sound. A PCM stream is obtained from an audio signal by discretizing both the voltage level and time. Time is discretized by choosing a uniform *sampling rate*, measured as the number of samples taken per second. Each sample is a discretized measure of the voltage level. The resolution of the samples is governed by the *bit depth*; namely, the number of bits allowed for each sample. For example, Figure 2 shows a digitized signal using a sample rate of 10 samples per second, and a resolution of 3 bits using the 8 possible values: -3, -2, -1, 0, 1, 2, 3, 4 (this would be horribly poor quality for an audio signal). The difference between the sample and the actual signal level is known as the *quantization error*; using a higher bit depth reduces this. The sampling rate determines the highest frequency that can be correctly recovered from the sampled data. This is known as the *Nyquist frequency* and is half of the sampling rate. Compact discs use 16 bits of resolution (per channel) and 44,100 samples per second — enough to recover the entire range of human hearing (20 Hz to 20,000 Hz).

For this project, you will write some programs in C to manipulate PCM streams. You will work entirely with *uncompressed* streams (e.g., `.wav` files); popular formats like `.aac`, `.m4a` and `.mp3` instead use lossy compression on the stream to reduce the storage requirements.

0.0 Warning 0: These are 'loose' specs

This document does not attempt to *rigorously* define what your code must do. There *are* cases that are left unspecified: some on purpose, and some because I did not think of everything. Similarly, the project is quite open–ended: there are several different, reasonable designs that can work effectively; it is up to you to choose one, and deal with the consequences of that choice.

0.1 Warning 1: These are 'minimum' specs

This document describes the minimum of what your code should support. You might think of it as a contract: if the input is as specified, then the program must perform as specified. You are free to implement additional features or to go beyond any limits specified here. Also, beware that your code *will* be tested *beyond* the stated limits in some ways; you are allowed to either (a) fail cleanly with an appropriate error message, or (b) handle those cases correctly, as you choose. Any other behavior (e.g., program crash, or incorrect handling of an extreme case) may result in loss of points.

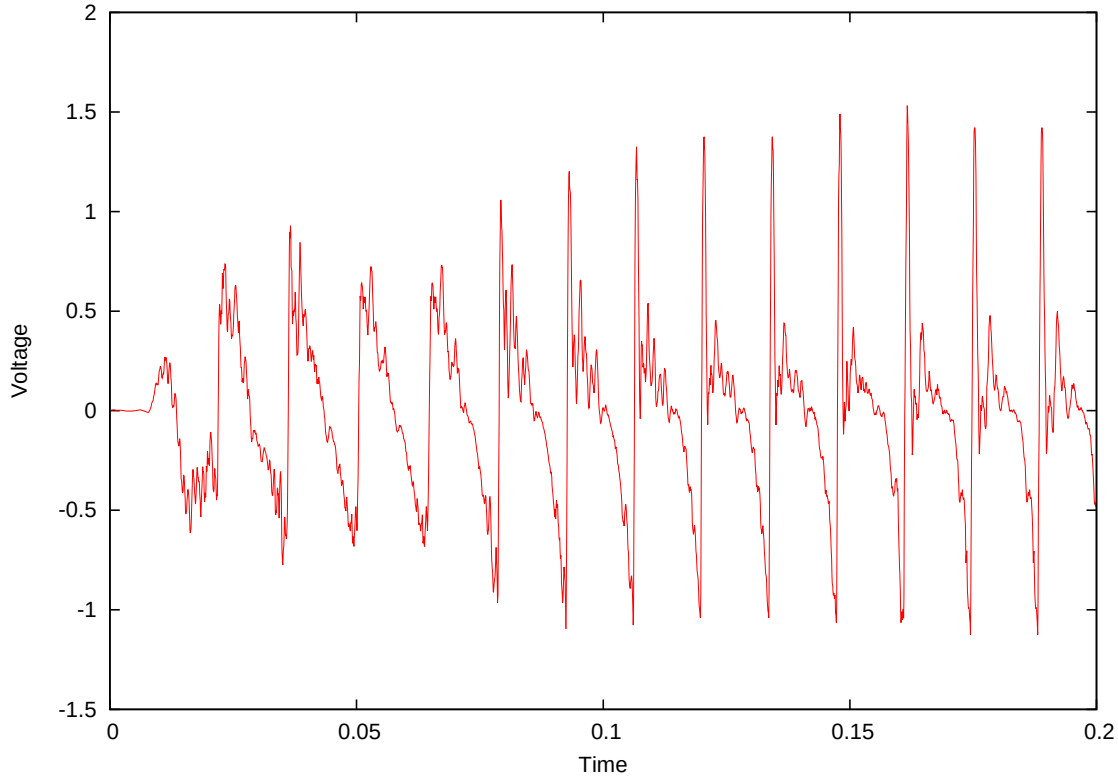

Figure 1: An audio signal

0.2 Warning 2: There will be more

The next part(s) of this project may require you to implement new programs, or may require you to re-implement programs from earlier parts, or both. Design your code carefully, with future expansion in mind. Part of your grade is based on how well your source code is organized and documented.

0.3 Warning 3: There might be "math"

You may need to work out some mathematical derivations for this project. I *strongly* encourage you to work out these derivations *on paper* and *before* you start coding. The math itself is easy enough, but trying to work it out in your head while implementing is probably not the smartest choice.

1 The programs

You must write the programs described below, in ANSI C (i.e., must compile on `pyrite` using "`gcc -ansi`"). As a general rule, your programs should be "user friendly". That means they should print useful error messages whenever possible. Also, your programs should work both for "CS229 format" files (described in Appendix A) and for AIFF files (described in Appendix B).

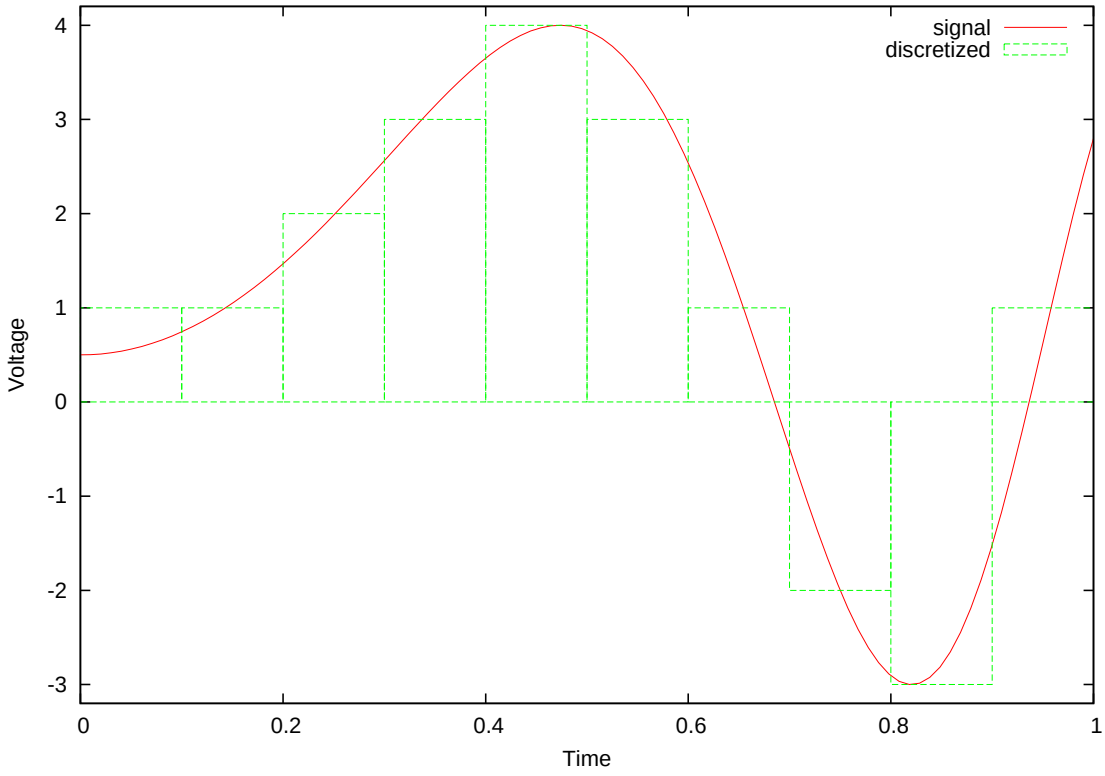

Figure 2: A discretized audio signal

1.0 `sndinfo`

The `sndinfo` program should read a sound file, and display the following to standard output.

- The file name
- The file format (`CS229` or `AIFF`)
- The sample rate
- The bit depth
- The number of channels
- The number of samples
- The length of the sound

The program should prompt the user for the name of the file to read (to standard error), and read the filename from standard input. The program output should be exactly as follows (so that we can automatically test your code):

```
Enter the pathname of a sound file:
<read from standard input>
------------------------------------------------------------ <60 dashes>
   Filename: <filename>
     Format: <CS229 or AIFF>
Sample Rate: <sample rate>
  Bit Depth: <bit depth>
   Channels: <number of channels>
    Samples: <number of samples>
   Duration: <h>:<mm>:<ss.ff>
------------------------------------------------------------ <60 dashes>
```

The duration should be rounded to the nearest hundredth of a second.

1.1 `sndconv`

This program converts from one file format to another. The program should first read in the name of the input file (with a nice prompt, of course), and then read in the name of the output file (also with a nice prompt). The output file should be written in the opposite format as the input file. In other words, if the input format is CS229, then the output format should be AIFF; if the input format is AIFF, then the output format should be CS229. All helpful messages should be written to standard error.

2 Limits

You may set the following limits.

- Supported bit depths are 8, 16, or 32 bits.
- Up to 32 channels are supported.
- The sample rate is an integer.

3 A note on working together

This assignment is intended as an *individual* project. However, some amount of discussion with other students is expected and encouraged. The discussion below should help resolve the grey area of "how much collaboration is too much".

3.0 Not allowed

Basically, any activity where a student is able to bypass intended work for the project, is not allowed. For example, the following are definitely not allowed.

- Working in a group (i.e., treating this as a "group project").
- Posting or sharing code.
- Discussing solutions at a level of detail where someone is likely to duplicate your code.
- Using a snippet of code found on the Internet, that implements part of the assignment. Generic code may be OK, but (for example) code that processes an AIFF file is definitely not OK. If you have any doubt, check with the instructor first.

As a general rule, if you cannot honestly say that the code is yours (including the ideas behind it), then you should not turn it in.

3.1 Allowed

- Sharing test files (please post them on Piazza).
- Discussions to clarify assignment requirements, file formats, etc.; again, please post these on Piazza.
- High–level problem solving (but be careful — this is a slippery slope).
- Generic C discussions. If you have trouble with your code and can distill it down to a short, generic example, then this may be posted on Piazza for discussion.

4 Submitting your work

You should turn in a gzipped tarball containing your source code, makefile, and a `README` file that documents your work. The tarball should be uploaded in Blackboard.

Your executables will be tested on `pyrite.cs.iastate.edu`. You should **test early, and test often on `pyrite`**. We will use some scripts to check your code; this means you should not change the name of the executables or the order in which your programs prompt for input.

5 Grading

The following distribution of points will be used for this part of the project.

sndinfo : 100 points

sndconv : 100 points

makefile : 20 points

Simply typing "`make`" should build all of your executables. Also, "`make clean`" should remove the executables and any object files. You may include other targets for your convenience as you choose (e.g., "`make tarball`").

Also, note that if your code does not *build* because of compiler or linker errors (either using `make` or by hand), you will likely lose much more than 20 points. Effectively: the TAs reserve the right to not grade your project *at all* if it fails to compile or link. **Test early, and test often on `pyrite`.**

Documentation & style : 20 points

Based on the `README` file only.

Total part 1 : 240 points

A CS229 file format

The CS229 sound format is a *plain text* file (probably the only sound format ever to use plain text). Be sure that you can handle text files created both in UNIX and in DOS. The intention of the format is to be flexible enough to be easy to use, yet rigid enough to be easy for your programs to read. For the most part, the file consists of *keywords* and *values*, both of which are guaranteed to be "contiguous" text. Keywords are case–sensitive. A file is structured as follows:

```
CS229
<header>
StartData
<samples>
<EOF>
```

```
CS229

# A really short sound to illustrate this file format

Samples     4
# We are doing something bizarre with the channels?
Channels    3

BitDepth    8

SampleRate  11025

# From here on things are more rigid
StartData
0   -127  127
10  45    -103
20  83    -4
30  0     99
```

Figure 3: An example CS229 file

where "`CS229`" and "`StartData`" are keywords, and "`<EOF>`" means "end of file". Note that the keyword "`CS229`" appears at the *very* beginning of the file. The "`<header>`" section consists of zero or more lines formatted in one of these three ways.

1. Blank (should be ignored)
2. Starting with "`#`" (should be ignored)
3. "`keyword` (whitespace) (value)" where (whitespace) refers to any number of tabs and spaces, and (value) refers to an integer. Legal keywords are: `SampleRate`, `Samples`, `Channels`, `BitDepth`.

The "`<samples>`" section consists of a line for each sample, where each sample is of the form

"$(\text{value})_1$ (whitespace) $(\text{value})_2$ (whitespace) $\ldots$ $(\text{value})_c$"

where c is the number of channels. The file itself has the following restrictions.

- The sample rate (`SampleRate`), number of channels (`Channels`), and bit depth (`BitDepth`) *must* be specified. The number of samples (`Samples`) is *optional*.
- The values specified for each sample are *signed integers* and are in the appropriate range for the bit depth. For example, if the bit depth is n, then the legal range for all sample data is from -2^{n-1} to $2^{n-1} - 1$.
- If the number of samples is specified, then there must be exactly that much sample data in the file.

The file shown in Figure 3 is an example of a legal CS229 file.

B AIFF file format

An AIFF file has the following format (for this project, anyway). Note that it is a "binary" file, so you will need to use a utility like `hexdump` to view one properly.

The first four bytes of the file are the (ASCII) characters "`FORM`". The next four bytes are an unsigned integer which specify the *remaining* number of bytes in the file (i.e., not including the first 8 bytes of the file). The next four bytes are the (ASCII) characters "`AIFF`".

The rest of the file is comprised of two or more "chunks". Each chunk begins with a four byte ID (usually, ASCII characters), and a four byte integer specifying the *remaining* number of bytes in the chunk (again, the total chunk size minus 8 bytes). Two of the chunks are required, the rest are optional (and you may ignore them).

Exactly one "Common" chunk is required, which has the following format. The ID is "`COMM`", and the size is always at least 18. Then, the chunk contains the following integers, in this order.

NumChannels (2 bytes). The number of channels.

NumSampleFrames (4 bytes, unsigned). The total number of samples (each sample includes a data point for every channel).

SampleSize (2 bytes). The number of bits in each sample data point. For this project, the value will be 8, 16, or 32.

SampleRate (10 bytes). An IEEE Standard 754 "extended precision" (80–bit) floating point value, representing the sample rate in sample frames per second.

Any extra bytes beyond those 18 may be ignored. Note that every AIFF file must contain exactly one Common chunk.

A "Sound Data" chunk is also required, unless the Common chunk specifies 0 for the total number of samples. The ID is "`SSND`", and the size is always at least 8. Next, the chunk contains two integers:

Offset (4 bytes, unsigned). Number of bytes to skip in the sample data, to reach the first sample. This is normally 0, and is used in case applications need to align the data to fixed–size blocks.

BlockSize (4 bytes, unsigned). Size of a block that the sample data is aligned to. This is normally 0, meaning "not aligned". If the waveform data is block aligned, there may need to be padding bytes at the end of the data to guarantee that it ends on a block boundary.

Finally, the rest of the chunk stores the sample data, where each data point is signed, and is stored using the number of bits (8, 16, or 32) specified in the Common chunk. A single sample consists of a data point for each channel. There cannot be more than one Sound Data chunk.

B.1 Some important notes

- All integers are stored in *big endian* format, i.e., most significant byte first.
- All signed integers are stored using 2's complement.
- For alignment purposes, every chunk is required to be an even number of bytes, *even if the specified number of bytes is odd.* For example, if the specified chunk size is 3, then there will be 3 bytes of data, followed by a single "padding" byte of 0 that should be ignored.
- You must be able to read files with any legal chunk in it.

The following hyperlinks are good sources for additional information.

- `http://muratnkonar.com/aiff/aboutiff.html`
- `http://muratnkonar.com/aiff/index.html`