ComS 327 Project 4

Tools for digital sound

November 4, 2015

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 to manipulate and generate 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.

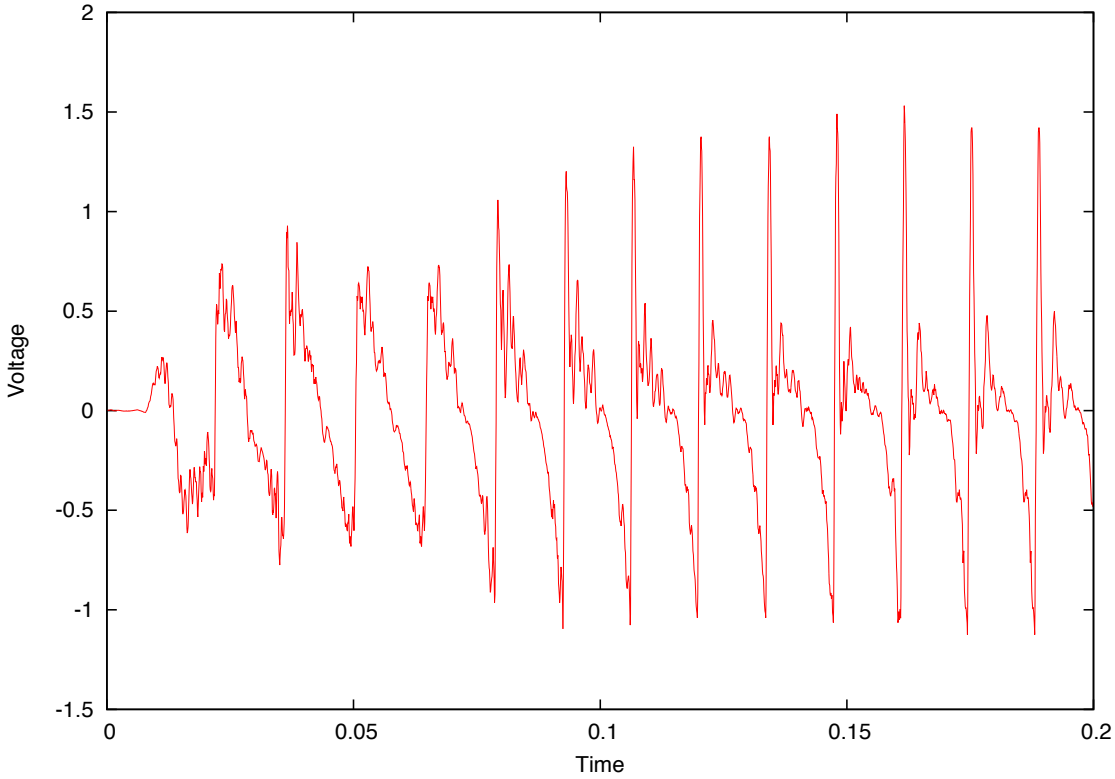

Figure 1: An audio signal

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, provided they meet the design requirements specified later.

0.1 Warning 1: Think before you code

It might be possible to complete this assignment (maybe even on time) by immediately starting to write code, and continuing to write code until completion. However, your life will be *significantly* easier if you spend some time sketching things out *on paper* before you start coding. For example, you may have that one program requires the functionality provided by another; taking some time to think about how to divide things into separate C++ classes

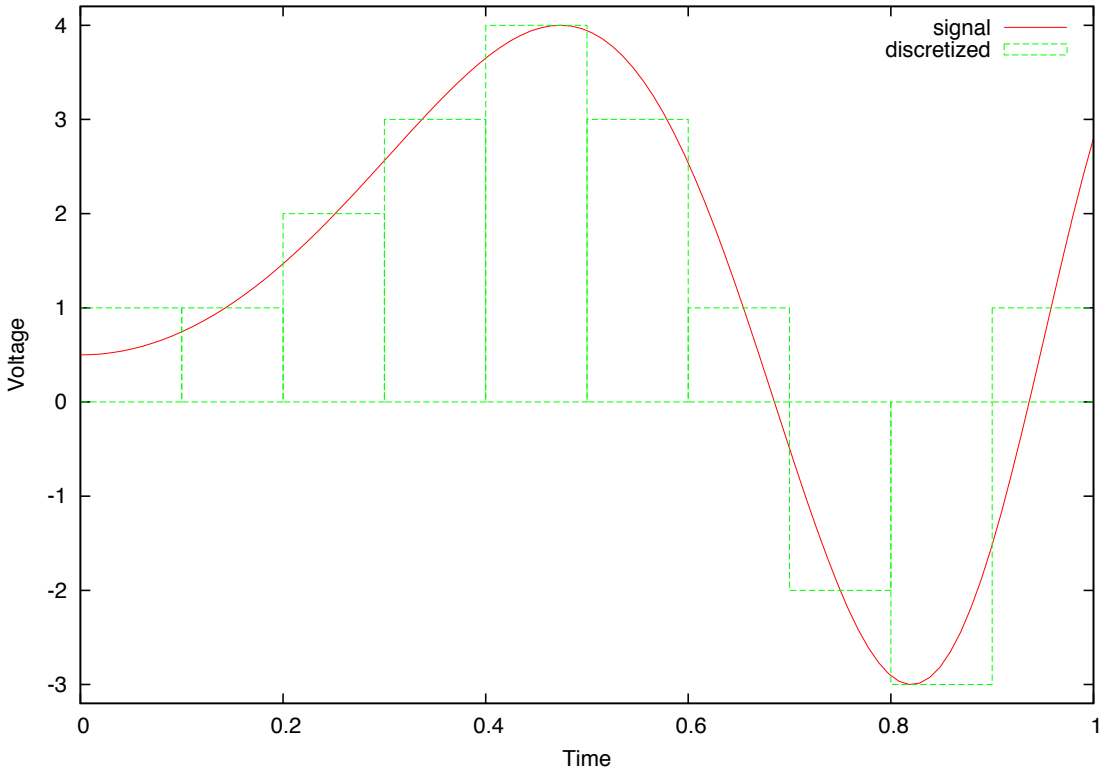

Figure 2: A discretized audio signal

that can be re-used across programs will *save* you time in the long run. Try to provide loose coupling among classes, program to interfaces, and use design patterns whenever possible.

0.2 Warning 2: Math

You *will* need to work out some math for this project. Effectively, this means that part of the project requires non–trivial mathematical derivations for your code to work properly. I *strongly* encourage you to work out these derivations *on paper* and *before* you start coding. The math itself is easy enough (mostly algebra, and some Trig), but trying to work it out in your head while implementing is probably not the smartest choice. See Warning 1.

1 The programs

You must write the 6 programs described below. All of these are command–line tools that follow the Unix style of passing switches and arguments on the command line. As such, your programs should *never* prompt the user for *anything*. You may assume that all switches appear before the first argument. The requirements below are *minimal* requirements; you may provide extra functionality or handle extra cases as you choose. For example, you might include a `--debug` switch that causes extra debugging information to be written to standard error.

As a general rule, your programs should be "user friendly". That means they should print useful error messages whenever possible. Additionally, your programs should print any messages or diagnostic information to standard error, with standard output reserved for sound output. I do not require that your programs be absolutely "quiet", but take care that the default behavior prints at most a few messages.

All programs must read and write `.cs229` format for sound files, described in Appendix A. Optionally, and for extra credit, you may write a conversion program that produces `.wav` format, described in Appendix B, from `.cs229` format. Similarly, all programs should accept the `-h` switch, which causes the program to display a "help" screen and then terminate. The help screen should describe the program, its usage, and all legal switches (including any extra ones you implemented).

1.0 `sndinfo`

This program reads all sound files passed as arguments, and for each one, displays the following.

- The file name
- The file type (`.cs229` or optionally for bonus points `.wav`)
- The sample rate
- The bit depth
- The number of channels
- The number of samples
- The length of the sound (in seconds)

If no files are passed as arguments, then the program should read from standard input. The only required switch for this program is `-h`.

1.1 `sndcat`

This program reads all sound files passed as arguments, and writes a single sound file that is the concatenation of the inputs. If no files are passed as arguments, then the program should read from standard input. You must support the following switches. Note that there are situations that make it hard to concatenate sound files such as ones with different numbers of channels. You must choose how to resolve these types of situations. One way here is to convert each sound to the maximum number of channels of any sound file, or simply give an error to the user.

`-h`	: show the help screen.
`-o file`	: specify the output file name; if omitted, write to standard output.
`-w`	: optionally for bonus points output in `.wav` format instead of `.cs229` format.

1.2 `sndmix`

This program reads all sound files passed as arguments, and "mixes" them into a single sound file. The program is invoked as

`sndmix [switches] mult1 file1 ...mult`n `file`n

where the sample data of `file`i is multiplied by `mult`i (a real value between -10 and 10); the scaled data is summed to produce the output file. You must support the following switches.

`-h`	: show the help screen.
`-o file`	: specify the output file name; if omitted, write to standard output.

1.3 `sndgen`

This program is a mess of switches. It produces a sound of a specified frequency and waveform (see Appendix C), using a simple ADSR envelope (see Appendix D). You must support the following switches. Note this program would benefit from several design patterns.

`-h`	: show the help screen.
`-o file`	: specify the output file name; if omitted, write to standard output.

`--bits` n	: use a bit depth of n
`--sr` n	: use a sample rate of n
`-f` r	: use a frequency of r Hz
`-t` r	: total duration of r seconds
`-v` p	: Peak volume. $0 \leq p \leq 1$
`-a` r	: attack time of r seconds
`-d` r	: decay time of r seconds
`-s` p	: sustain volume. $0 \leq p \leq 1$
`-r` r	: release time of r seconds
`--sine`	: generate a sine wave
`--triangle`	: generate a triangle wave
`--sawtooth`	: generate a sawtooth wave
`--pulse`	: generate a pulse wave
`--pf` p	: Fraction of the time the pulse wave is "up"

Note that the volume parameters are given as the percentage of the maximum volume.

1.4 Class design

You are to create an overall UML class diagram for all the programs you implement and turn in for grading. (including bonus problems) Many of the programs will (and should) share classes. You should turn in a PDF file of your UML class diagrams together with any documentation you used to generate, derive, figure-out the classes. For example, CRC cards are one way to come up with these classes. You should also have good descriptions of the classes in these documents. These descriptions would probably make good comments at the beginning of your C++ classes. It would also not hurt to point out where you use any patterns.

1.5 `sndcvt` Bonus problem

This program converts between `.wav` and `.cs229` formats. If no file passed as an argument, then the program should read from standard input. If the input file is of type `.wav` then the program produces a sound file of type `.cs229`. Also, if the input file is of type `.cs229` then the output produces a sound file of type `.wav`. Your program may overwrite existing files without warning.

The program is invoked as

```
sndcvt [switches] inputfile
```

You must support the following switches.

`-h`	: show the help screen.
`-o file`	: specify the output file name; if omitted, write to standard output.

1.6 `sndfx` Bonus problem

This program adds sound effects to a specified sound file. This is a very open ended bonus problem. The idea is to add sound processing to an existing sound file. It is your job to design the user commandline interface similar to the other programs in the assignment. The idea behind this program is given an input soundfile, the user can addon (in order) several sound effects. You may choose the effects you add. Examples are backwards, reverb, echo, pitchup, pitchdown, fadeinout. Effects could be added more than once and order matters. In addition, effects may have parameters that are dependent on the type of effect. (The command line interface may be complicated.) An example of using this program may result in: reverse the samples, add reverb (with appropriate paramters), reverse the samples, followed echo (with appropriate parameters). (I have no idea what this would sound like.) Note that this is a real nice place to use a decorator pattern.

1.7 `sndplay` Bonus problem

This program reads a music file in `.abc229` format (see Appendix E) and produces a sound file of that music. You must support the following switches.

`-h`	: show the help screen.
`-o file`	: specify the output file name; if omitted, write to standard output.
`-w`	: output in `.wav` format instead of `.cs229` format.
`--bits` n	: use a bit depth of n
`--sr` n	: use a sample rate of n
`--mute` n	: do not play instrument n

Appendix F describes how to determine the frequency of a given note.

1.8 User defined bonus problem

You may propose a bonus problem related to sound processing and this assignment by Emailing Jim. If the bonus problem is accepted, it will be posted on Blackboard together with the number of points it is worth.

2 Limits

You may assume (and enforce) the following limits.

- The bit depth will be 8, 16, or 32 bits.
- The number of channels will be less than 128.
- The number of instruments in a music file will be at most 16.

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 a `.wav` 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 Blackboard).
- Discussions to clarify assignment requirements, file formats, etc.; again, please post these on Blackboard.
- High–level problem solving (but be careful — this is a slippery slope).
- Generic C and 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 Blackboard for discussion.

4 Submitting your work

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

Your executables will be tested on `pyrite.cs.iastate.edu` and output files downloaded and played on a PC. You should **test early, and test often on** `pyrite`.

5 Grading

The distribution of points corresponds *roughly* to the degree of difficulty and length of time required for each component. Your mileage may vary.

Class Design Documents (UML and suporting documents) : 20 points

sndinfo : 10 points

sndcat : 15 points

sndmix : 15 points

sndgen : 20 points

makefile : 10 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 zipfile`").
>
> 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 10 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 and style : 10 points

> As part of this, your `README` text file should explain the purpose of each of the source files. Source files (especially headers) should be well documented. Documentation should *always* be written assuming the reader is another programmer. A good way to think about documentation is to imagine that in 5 years you will dust off this project and add more to it. What notes should you leave for yourself to make your life easier?

sndcvt : (Bonus) 15 points

sndfx : (Bonus) 15 points

sndplay : (Bonus) 30 points

Total : 100 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 may be upper, lower, or mixed case. A file is structured as follows:

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

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`, `BitRes`.

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

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

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 (`BitRes`) *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 8, then the legal range for all sample data is from -127 to 127.
- 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 WAV file format

A `.wav` file has the following format. 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 "`RIFF`". That's a Microsoft proprietary format for multimedia in general, which stands for "Resource Interchange File Format". The next four bytes are a little–endian integer that gives the *remaining* number

```
Cs229

# A really short sound to illustrate this file format

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

BitRes      8

# Because we can:
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

of bytes in the file; i.e., the total file size minus 8 bytes. The next four bytes specify the format, which will be the (ASCII) characters "`WAVE`" for a `.wav` file.

The rest of the file is comprised of zero or more "chunks". Each "chunk" begins with a four byte ID (usually, ASCII characters), and a four byte little–endian integer specifying the *remaining* number of bytes in the "chunk" (again, the total chunk size minus 8 bytes). There are many different types of chunks, but for this project, only two types of chunks are important: the format chunk, and the data chunk. Theoretically, chunks may appear in *any* order in the file, but many utilities require that the format chunk is first, and the data chunk is second.

The format chunk ID is the four–character sequence "`fmt `" (note the trailing space). The chunk size is usually 16, but can be more. After the chunk size, there are the following little–endian integers in this order.

AudioFormat (2 bytes). A value of 1 here corresponds to PCM data, and this is the only type of data that you are required to handle.

NumChannels (2 bytes). The number of channels.

SampleRate (4 bytes). The sample rate.

ByteRate (4 bytes). Should equal the product of: sample rate, number of channels, and byte depth.

BlockAlign (2 bytes). Should equal number of channels multiplied by the byte depth. In other words, the total number of bytes in an entire (all channels) sample.

BitDepth (2 bytes). Will be 8, 16, or 32.

Any extra bytes may be ignored.

The data chunk ID is the four–character sequence "`data`". Then we have the chunk size as usual. The rest of the chunk contains the actual sound data, stored in binary, with each sample consisting of the data for all channels. If the bit depth is 8, then the sample data are *unsigned*; otherwise, the sample data are *signed*, 2's complement, little–endian integers.

The following hyperlink is a good source for additional information.

- `http://www-mmsp.ece.mcgill.ca/Documents/AudioFormats/WAVE/WAVE.html`

C Waveforms

For this project, you will generate some (simple) periodic waveforms. In this case, *periodic* refers to the fact that the waveform is comprised of a *repeating* pattern. The *frequency* specifies the number of times the pattern appears per unit time. The *amplitude* is half the distance from the highest point of the wave to the lowest point; if we assume that the wave is centered around zero, then the lowest point will be at the negative of the amplitude, and the highest point will be at the positive of the amplitude. You will need to generate sine, triangle, sawtooth, and pulse waveforms, shown in Figure 4. Note that in the figure, all waveforms have an amplitude of 1 and a frequency of 2 Hz, and the pulse waveform spends 50% of the time "up". Your generator must be able to vary these quantities.

Not surprisingly, all of these waveforms will sound very "synthetic". That's because an actual instrument has a *much* more complex waveform. Feel free to experiment with other waveforms as you choose.

D ADSR envelope

When an actual instrument produces a sound, the amplitude of the resulting waveform will change over time. The nature of this will depend strongly on the instrument. An *envelope generator* is used to capture this behavior. For this project, you will use a simple ADSR envelope, which specifies four parameters.

Attack time is the time for the sound to build up to its initial, peak volume.

Decay time is the time for the sound to fade down to the sustained volume.

Sustain level is the volume level held "while the key is pressed".

Release time is the time for the sound to fade to zero.

Figure 5 shows a plot of amplitude versus time for a generic ADSR envelope. Note that this should work independently of the choice of waveform. As the duration t decreases, the amount of time spent in the sustain stage should decrease. As t decreases past the point where the sustain stage takes zero time, one or more of the remaining stages will need to be shortened, using the following precedence rules.

- The "release" stage has highest precedence, and the amplitude at the start of this stage should be the final amplitude of the previous stage.
- No sound should be produced if the duration t is less than the release time.
- The "attack" stage has middle precedence. If this stage is shortened, then final amplitude of this stage will be less than the peak volume.
- The "decay" stage has lowest precedence. If this stage is shortened, than the final amplitude of this stage will be higher than the sustain volume.

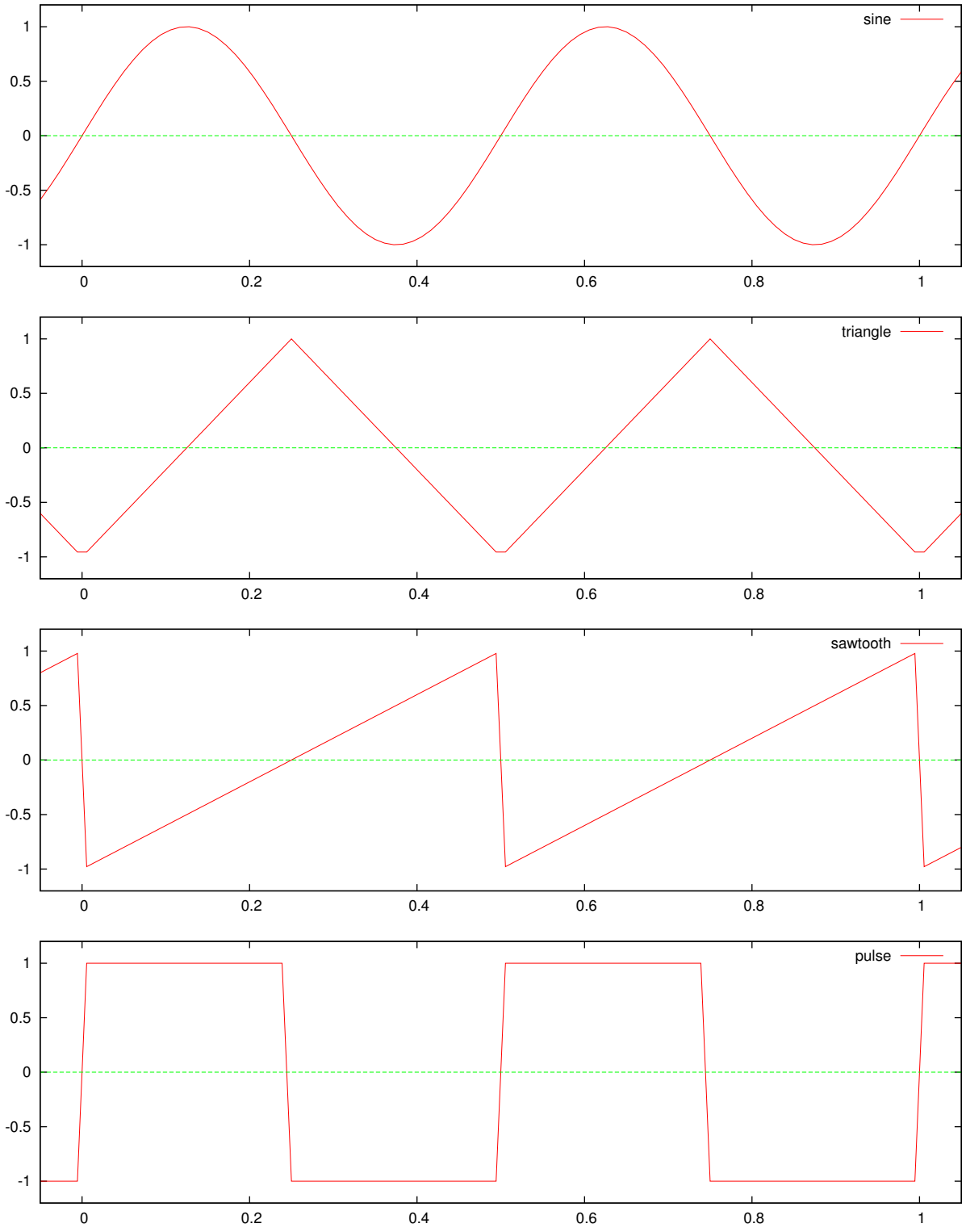

Figure 4: Various simple waveforms

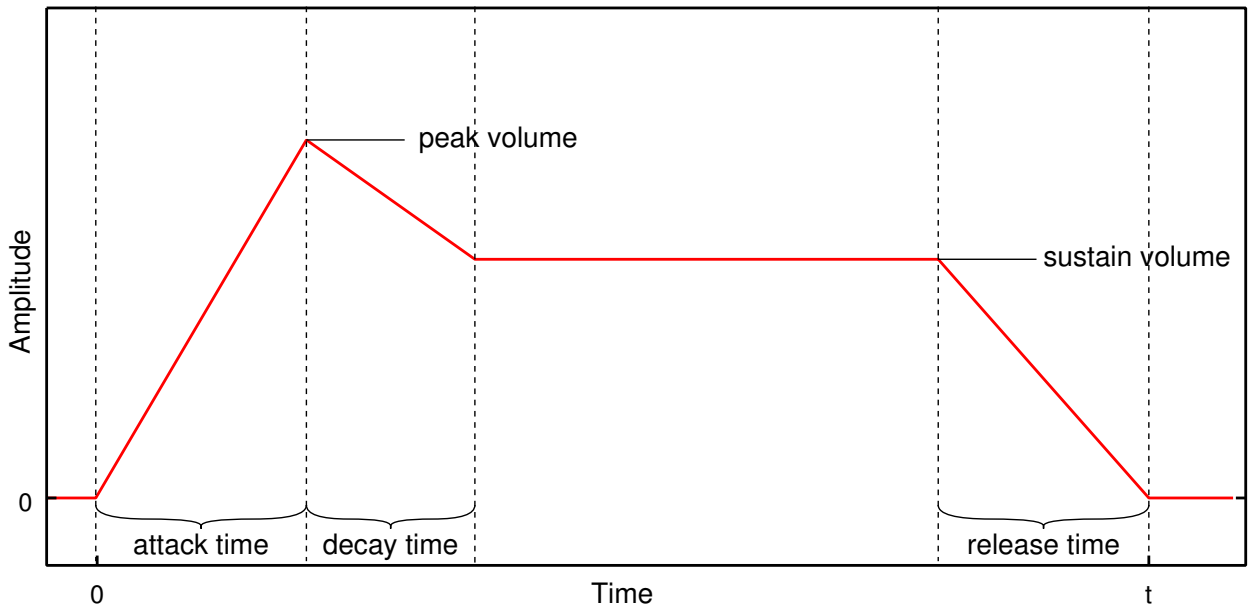

Figure 5: The ADSR envelope for a sound of duration t

E ABC229 file format

The `.abc229` format is inspired by, but is quite different from, the ABC notation used for text–based music. The format specifies a number of instruments. For each instrument, the waveform and ADSR envelope is given, along with the notes to play.

Like `.cs229` format, keywords may be upper, lower, or mixed case. Be sure you can handle text files created both in UNIX and in DOS. A file is structured as follows.

```
Abc229
<header>
<instrument>
...
<instrument>
<EOF>
```

In other words, the file starts with the "`Abc229`" keyword, then contains a `<header>` section, followed by zero or more `<instrument>` sections. 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 (value) is an integer. The only legal keyword is: `Tempo`. The tempo specifies the number of *counts* per minute.

An "<instrument>" section has the following format:

```
Instrument n
<waveform>
Score
[
<notes>
]
```

where "Instrument" and "Score" are keywords, and the instrument number n must be equal to the number of instrument sections prior to this one in the file. The "<waveform>" 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 (value) depends on the keyword. Legal keywords are Volume, Attack, Decay, Sustain, Release, and PulseFrac, which take a real value, and WaveForm, which takes a value that is one of the following keywords: Sine, Sawtooth, Triangle, Pulse.

The "<notes>" section is a sequence of notes. Notes may be separated by any amount of whitespace, including none, may contain a number of "|" characters that are to be ignored, and may be broken over several lines. A note is given by the following sequence *with no spaces*.

- A letter "a", ..., "g" or "A", ..., "G", where the case *is* significant: lower case letters are an octave above upper case letters. Letters "z" and "Z" may also appear, as rests (no sound is produced). The note "c" refers to middle C, and the note "B" refers to the B just below middle C.
- (Optionally) a sharp symbol. For the musically illiterate, see Appendix F for which letters may be followed by "#", along with discussion of octaves.
- At least zero and at most two octave modifiers. Each "'" character means "up one octave" and each "," character means "down one octave".
- (Optionally) a count multiplier, indicating how long to play the note. If missing, the note should be played for a duration of one count. Otherwise, the multiplier is either of the form "n/d", or "n" which is equivalent to "n/1". n may be any integer from 1 to 16, and d may be 1, 2, 4, 8, or 16.

The file shown in Figure 6 is an example of a legal .abc229 file. Instrument 0 plays a simple scale starting from middle C.

```
Abc229
%
% Plays a quick scale
%

Tempo 240

% Note: first instrument is number 0

Instrument 0
Waveform  Sine
Volume    0.2
Attack    0.002
Decay     1.5
Sustain   0
Release   0.002

Score
[
c c# d d# e f f# g g# a a# b c'3 z
]

Instrument 1
Waveform  Sine
Volume    0.2
Attack    0.002
Decay     1.5
Sustain   0
Release   0.002

Score
[
e f f# g g# a a# b c' c#' d' d#' e'3 z
]

Instrument 2
Waveform  Triangle
Volume    0.2
Attack    0.002
Decay     0.002
Sustain   0.2
Release   0.002

Score
[
G G# A A# B c c# d d# e f f# g3 z
]
```


Figure 6: An example `.abc229` file

F From notes to frequencies

For this project, the music files assume an *equally–tempered* scale divided into 12 intervals, namely the notes

$$\text{A, A\#, B, C, C\#, D, D\#, E, F, F\#, G, G\#}$$

which repeat. Each group of 12 notes is called an *octave*.

The frequency of any note in any octave is precisely twice the frequency of the same note in the octave below, and half the frequency of the same note in the octave above. Additionally, the ratios of frequencies between adjacent notes is a constant s (in an equally–tempered scale). From these two facts we obtain

$$s = \sqrt[12]{2}$$

since we must have $s^{12} = 2$.

The note "`a`" (A above middle C) has a frequency of precisely 440 Hz. You should be able to compute the frequencies of all other notes. For instance, A# above middle C has a frequency of $440 \cdot s$ Hz, B above middle C has a frequency of $440 \cdot s^2$ Hz, and so on. Similarly, G# above middle C has a frequency of $440 \cdot s^{-1}$ Hz, and G above middle C has a frequency of $440 \cdot s^{-2}$ Hz.