Guide to BOOK Chapter 5 Examples:

Victor Lazzarini

Introduction to Digital Audio Signals

Wheel Animation

A visual example of aliasing is provided by a small python program that animates a rotating wheel at different speeds. If you move the slider to the right, you will speed it up from 0Hz to the sampling rate; you'll see that above ½ sampling rate, the movement will be reversed and the wheel will slow down. This is the visual analogue to aliasing in digital audio. To run it, just type

```
$ python wheel.py
```

(you will need Python, see below).

Csound Examples

You can run these examples in Csound 5, they provide some examples of the ideas discussed in the chapter.

ssb.csd

This Csound example shows an application of complex signals: single-sideband modulation. The idea is straightforward: (1) split the signal spectrum into positive frequencies and negative frequencies (or use a complex sinusoid to start with, which we do here); (2) ring-modulate these two signals (multiply them) by a complex sinusoidal; (3) the result is that the positive frequencies will form the upper sideband and the negative frequencies will form the lower sideband, so you can output them separately. If the two signals are mixed you will have the ordinary ring-modulated signal that contains both sidebands (difference and sum frequencies). See the comments in the Csound code for more details on how this is achieved.

fm-pm.csd

The differences in implementation between **phase modulation** and **frequency modulation** are shown in this Csound example. You will see that the differences in sound result are small, but in some cases significant.

Building the Programming Examples

These examples will build on most systems, and have been tested on OS X, Linux and Windows. To build the examples you will first need a C compiler. I recommend using **gcc**, if

you have a choice (on OS X and Linux you'll get it by default). On Windows **gcc** can be got by either installing **MinGW/MSYS** or **Cygwin** (www.cygwin.com).

Examples are built using **scons**, which is a handy utility for code maintenance and programming projects. It looks after re-building any files that have been modified and gets all the right components together to build the whole project. See the 'scons-guide' document for more details. If you don't have **scons,** you can get it from **www.scons.org**, and you will also need Python, which you can get from **www.python.org.** Both software are of easy installation.

You will also need to install **libsndfile**, if you do not have it already. In fact, **scons** will look for it and tell you if you don't have it. Then you can get it from **www.mega-nerd.com/libsndfile.** It's all very simple to build (if you need to) and install, all the instructions for your system are provided.

Once your system is ready for it, all you need to do is type the following at your shell/terminal (`$` is the prompt) and all examples will be built:

```
$ scons
```

You can also build each example separately just by invoking the **gcc** command directly from the shell:

```
$ gcc  -o sine sine.c -I/usr/local/include -L/usr/local/lib -lsndfile
```

The `-I` flag tells gcc to look for extra header files in `/usr/local/include` and the `-L` tells it to do the same for libraries. The required library is then supplied for with `-l` flag.

sine.c

This is a simple example on how to synthesize a sine wave. It is a very basic example, as it takes no parameters and always produces a full-amplitude 440 Hz sine wave in the file *sinetest.wav*. See **Project 1** below for some ideas on how to improve it.

mix.c

This demonstrates how mixing two signals is just a matter of summing them, sample by sample. This example is more advanced, as it takes two input filenames from the command-line and an output filename, that will contain the final mix. It can mix any two files of the same format (channels and sampling-rate). See **Project 2** for ways of enhancing this program into a fully-functional mixer.

gain.c

This next example is a demonstration of how amplitude adjustments (gain) are just based on the multiplication of a signal by a scalar constant (or also a slowly-varying signal such as an envelope). **Project 3** will give you some ideas on how to modify this program for some extra functionality.

ring.c

Another use of multiplication is shown by this example. As already hinted by the **ssb** example above, ring modulation is the multiplication of two audio signals, sample by sample. This produces a spectrum that will contain the sums and differences between all components of the two inputs (sums making the upper sideband and differences constituting the lower sideband).

synth.c

This shows how a basic wavetable oscillator works and how tables with different waveshapes can be created using the Fourier series. **Project 4** gives some ideas on how to generate a more varied palette of sounds from this synthesizer.

fmsynth.c

This final example is an implementation of FM synthesis (using one modulator and one carrier), using direct calls to the sine function in the C maths library. Although not the typical way of implementing FM, this demonstrates the technique in good detail. See **Project 5** for some ways of enhancing this FM synthesizer.

Projects

Here I'll propose some ideas that you can try for yourself. These will be mainly based on enhancing the examples that have been provided for this chapter. However, they'll give you some useful insight into audio programming.

Project 1

The simple sine wave generator implemented in **sine.c** can be modified to allow for more flexibility in generating sines that could be used as test tones for a variety of applications. To do this you can:

1. Let the user specify the output filename.
2. Allow for variable parameters (amplitude, frequency, sampling rate).
3. Let the sound duration be set by the user.

When tackling these modifications, try to do one at a time, so that you don't get lost in the various tasks involved. In fact, all these changes are based on one addition: getting user input. You can do it in two ways:

1. Prompt for input, which you can do by using **printf()** and **scanf()** (from **stdio.h**).
2. Read the command-line arguments, that the other examples do, so you can check how it's done by looking at them. I'll give you hint: if you define the main function to take two parameters, an argument count and an array of strings, then you can retrieve each argument as a string. Use **atof()** or **atoi()** to convert string arguments to floats or ints (these are defined in **stdlib.h**).

Project 2

The two-soundfile mixer in **mix.c** can be transformed into a more flexible mixer by the following modifications:

1. Allow for the volume of each input to be set separately (see gain.c).
2. Provide a way of offsetting the start of one of the files.
3. Make sure the mixed output duration is the result of the largest maximum soundfile duration plus any offsets (at the moment, the output duration is set by the 1st input.).

As before, try tackling the points one at a time. Extra input will be required from the user; this can be got as in project 1. Modifications 2 and 3 will allow for portions of one of the inputs to be mixed with silence (as the other soundfile might be offset or finished), so this will have to be addressed. One hint about libsndfile can be handy: soundfile duration (in frames) can be taken from the SF_INFO data structure member **frames**.

Project 3

The **gain** program simply scales amplitude. We can try modifying it into a 3-stage envelope shaper by adding a variable gain control. The 3-stage envelope will have the following extra parameters: 1. attack time and 2. decay time. The envelope duration is the soundfile duration, so attack time is calculated from the beginning of the file and decay time from the end.

Once we have these two parameters in seconds, we will need to convert them to samples, by multiplying them by the sampling rate. The variable gain value is then calculated using a simple ramping function:

$$g = gain \times \frac{sample\_count}{dur} \qquad \text{for the attack period}$$

$$g = gain \times \frac{dur - sample\_count}{dur} \qquad \text{for the decay period}$$

and $g = gain$ elsewhere. The *dur* parameter is the attack/decay duration in samples and the *sample_count* will start from 0 and count up samples until the envelope segment is finished. You will need some if/elses to select between the different envelope segments.

Project 4

The wavetable synthesizer in *synth.c* can be enhanced by adding some of the things you'll have practiced by now.

1. You can allow for other types of waveshapes based on different mixtures of sinusoids using the Fourier series concept.

2. You can add envelopes to amplitude and frequency.

The first task should be easily implemented by modifying how the wave table is filled. You can prompt for parameters, or read them from a text file (HINT: use **fopen(), fclose()** and **fscanf()** to open/close and read from text files). The second task is very much based on what you have already done in Project 3. You will then have a nice wavetable synthesizer that you can use to make your own sounds.

Project 5

This project is in the same vein as the previous one. Starting with the FM synth in *fmsynth.c*, you can add envelopes to:

1. Amplitude.
2. Carrier frequency.
3. Index of modulation.

Basically you can add envelopes to all parameters. Notice that I intentionally left out modulator frequency. You can, of course, make this one variable, but you will have to go back to the text and read about the modifications you'll need to make. (HINT: if the frequency varies, then we need to calculate the varying sine/cosine phase (angle) on the basis of an increment proportional to the frequency, otherwise it will not be set correctly.

Finally, you can try putting all these projects together, plus the ring modulator into a big synthesizer/processor, that will generate audio, process audio and mix it. But I'll leave the desing to you and your imagination…