`blow_chunks`
user manual

Tristan Quaife

February 8, 2026

Contents

Chapter 1

Quick start

1.1 Getting the code

You can download the code from `https://github.com/tquaife/blow_chunks` or, better, use `git` to check out the repository:

```
git checkout https://github.com/tquaife/blow_chunks
```

1.2 Compilation

`blow_chunks` has no external dependencies so, assuming you have a C compiler and `make` installed, it should be as easy as opening a terminal, changing directory to the location of the source code and typing:

```
make
```

If you don't have these tools, search for how to install C development tools for your system.

1.3 A first sound

`blow_chunks` reads ascii text from the standard input and writes wav files to the standard output. A very simple example of this is as follows. Open the terminal and type:

```
echo sin 220 1 1|./blow_chunks > a.wav
```

This command produces a one second sample of the note A at 220Hz using a sine wave oscillator. On most linux systems, you can listen to the sound by typing:

```
open a.wav
```

which will use your default media player to play the wav file. Alternatively, if you have `SoX` installed, you can use the `play` command:

```
play a.wav
```

You can also use `play` to directly preview the sound without writing to a file:

```
echo sin 220 1 1|./blow_chunks|play -
```

Note, however, that the audio will skip if you try this with big input files.

1.4 Basic `blow_chunks` instructions

What's going on in the command described in the previous section? The text being passed to `blow_chunks` is "`sin 220 1 1`", which is an example of a oscillator, the main building block of sounds in `blow_chunks` . Most oscillator commands look something like:

```
oscillator_name frequency phase volume
```

The different oscillators are described later in this document, but valid examples include `sin`, `sqr` and `sup` which are a sine wave, a square wave and a rising sawtooth wave, respectively. They all take the same arguments. The `frequency` argument controls the pitch, `phase` controls the phase (but will not have a noticeable effect in these simple examples) and volume controls the volume of the sound. All of these arguments can take on any floating point value, including negative numbers, but typically `frequency` is a value above 0.0Hz and less than 22000.0Hz, and `phase` and `volume` will have values between 0.0 and 1.0.

1.5 Modulating modulators

A key concept in `blow_chunks` is that the arguments to oscillators (i.e. the numbers following the name of the oscillator) can be replaced with more oscillators, which we refer to as modulators. This is done by following the numeric argument with curly brackets and specifying a new oscillator inside the brackets. This allows `blow_chunks` to implement a variety of synthesis techniques, such as FM (or "frequency modulation") synthesis. For example, we can modify our original example as follows:

```
echo sin 220 {sin 220 1 1} 1 1|./blow_chunks >afm220.wav
```

Here, the frequency of our original oscillator is being modulated at its original frequency. Listen to the resulting sound. Basic FM synthesis is often described as making the original waveform sound "brighter" and was the foundation of many synths of the 1980's. Of course you don't need to modulate the frequency by itself, you can also try out other frequencies, potentially triggering all sorts of side-harmonics, some of which are less pleasing to the ear than others:

```
echo sin 220 {sin 356 1 1} 1 1|./blow_chunks >afm356.wav
```

It is possible to keep nesting modulators indefinitely, at least within the constraints of your computer's memory, which will be a *lot* of modulators on modern computers. In the following example, we will use a low-frequency sawtooth oscillator to modulate the amplitude of the FM effect. In this example, we will use a frequency of 1Hz:

```
echo "sin 220 {sin 356 1 1{sup 1 1 1}} 1 1"|./blow_chunks >afm356_fade.wav
```

N.B. The above should all be typed on one line.

Have a listen and convince yourself of how the above command is being used to generate dynamics in the sound. Also note how quickly we have been able to build up something that is interesting. At this stage, it is worth having a go at adding modulators into the above sound and experimenting with the different effects.

1.6 Envelopes

Envelopes are used to change the value of a modulator argument as a function of time. They are delineated using triangular brackets <> and are filled with one or more time-value pairs that are separated by a colon. Unlike a modulator, which follows the argument, an envelope replaces it. A simple example is to fade a sound in:

```
echo "sin 220 1 <0.:0. 0.8:1.0>" |./blow_chunks >a_fade_in.wav
```

In this example, the amplitude of the oscillator is replaced with an envelope, which has a value of zero at a time of zero seconds, increasing to an amplitude of 1.0 at 0.8 seconds. Times are always in seconds and there must be no space between the colon and the time and value pair. At any time before the first time in the envelope list, the value is set to the first value in the list and at any time after the last time in the envelope list, the value is set to the final value in the list.

Envelopes can be used top replace any of the modulator arguments, for example to create portamento type effects:

```
echo "sin <0.1:220 0.9:440> 1 1" |./blow_chunks >a_octave_slide.wav
```

However, envelopes cannot be nested inside other envelopes, and you cannot place a modulator inside an envelope. You can, however, put an envelope in front of a modulator, or inside a modulator. For example, extending the above example to include FM:

```
echo "sin <0.1:220 0.9:440> {sin <0.1:220 0.9:440> 1 2} 1 1"
    |./blow_chunks >a_octave_slide_with_FM.wav
```

1.7 Basic sound duration control

At this point, you may want to make sounds be a bit longer than one second, so you can explore the effects of different low frequency modulators. For basic sounds, this can be controlled using the optional argument `-d` to `blow_chunks` . This creates the previous sound for 2.5 seconds we can do the following:

```
echo sin 220 {sin 356 1 1{sup 1 1 1}} 1 1|./blow_chunks -d 2.5 >
    afm356_fade_2p5s.wav
```

1.8 Additive synthesis and working with files

A well as providing a flexible modulation framework, `blow_chunks` allows multiple waveforms to be added together to make more complex sounds. However, it becomes cumbersome to do this using the command line, so it is better to write the instructions to a text file. Open a text editor and create a file called `cmajor.txt` with the following contents:

```
;a C major chord
sn3 261.6 1 1
sn3 329.6 1 1
sn3 392.0 1 1
```

We can now generate the corresponding wav file as follows:

```
./blow_chunks -d 2.5 < cmajor.txt > cmajor.wav
```

Note that `blow_chunks` supports comments. Anything after a ";" character until the end of the line is ignored. This is useful for leaving notes explaining what different parts of your file are doing. Blank lines are also ignored, which can be helpful for clearly separating different blocks of instructions. The `sn3` oscillator is described later, but produces a more satisfying result than `sin` in this example.

1.9 Stereo

Top level oscillators, i.e. those not enclosed in curly brackets "{}" can posses more than one volume value. By specifying two volume values the resulting sound file is in stereo (the resulting wav file will have two *channels*). Note that all top level oscillators must have the same number of volume values or `blow_chunks` will throw an error and exit. Similarly, all oscillators being used as modulators can only have a single volume value. In principle `blow_chunks` supports an arbitrary number of channels, so it is possible to create quadraphonic sounds etc, but you will only hear these effects if you have an audio set up that can play them.

The simplest example is to pan a sound to one channel:

```
;C panned hard left
sn3 261.6 1 1 0
```

By modulating the volume channels it's possible to generate effects in stereo. In the following example a sin wave, with two channels, has the volume of each channel modulated by a low frequency sine oscillator with a phase difference of 180°:

```
;A sweeping from right to left
sin 220 1 1 {sin 1 1 1} 1 {sin 1 0.5 1}
```

It's also possible to use low frequency modulated channels to create arpeggiators. A good choice of modulator for this purpose is the smoothed square wave modulator `sqx`. The following example is a stereo arpeggiated C major chord:

```
;C major stereo arpeggiator
```

```
;suggest generating at least 6 second sample
sn3 261.6 1 1 {sqx 1 0 1} 1 {sqx 1 0.5 1}
sn3 329.6 1 1 {sqx 1 0.16 1} 1 {sqx 1 0.67 1}
sn3 392.0 1 1 {sqx 1 0.33 1} 1 {sqx 1 0.83 1}
```

1.10 Variables

`blow_chunks` supports variables, including user defined and predefined ones (which can be overwritten by the user). The strength of user defined variables is largely in conjunction with other parts of the `blow_chunks` interface, but the predefined variables have some more obvious applications. The frequency values of notes in the twelve tone equal temperament tuning with A referenced to 440Hz are all stored as variables. All variables begin with the "$" character, and the frequency values can be retrived by typing, for example, $A2 (which is 110Hz). Flats are specified with a "b" character (e.g. $Ab2) and sharps with an "s" character (e.g. $As2). That means we can rewrite input files so that they contain the note names, which is far more convenient. The C major chord example now becomes:

```
;C major stereo arpeggiator, using in-built
;variables to specify the notes
;suggest generating at least 6 second sample
sn3 $C4 1 1 {sqx 1 0 1} 1 {sqx 1 0.5 1}
sn3 $E4 1 1 {sqx 1 0.16 1} 1 {sqx 1 0.67 1}
sn3 $G4 1 1 {sqx 1 0.33 1} 1 {sqx 1 0.83 1}
```

To provide and example of the user defined variables, imagine you wanted to experiment with the speed of the arpeggiation in the previous example. It would be necessary to change the frequency of each of the low frequency modulators in turn to the same value. In more complex sounds this could become onerous and a likely source of error. Instead, we can set variable to hold that value, as follows:

```
;A slower C major stereo arpeggiator, using in-built
;variables to specify the notes and a speed variable
;suggest generating at least 6 second sample
LfoSpeed=0.8
sn3 $C4 1 1 {sqx $LfoSpeed 0 1} 1 {sqx $LfoSpeed 0.5 1}
sn3 $E4 1 1 {sqx $LfoSpeed 0.16 1} 1 {sqx $LfoSpeed 0.67 1}
sn3 $G4 1 1 {sqx $LfoSpeed 0.33 1} 1 {sqx $LfoSpeed 0.83 1}
```

It's good practice to name variables in a meaningful way, but any combination of alphanumeric characters are permitted. No spaces or punctuation can be used, and the variable names are case sensitive.

1.11 The @sequence command

Commands in `blow_chunks` start with the "@" character and probably the most import is the @sequence command which allows oscillators to start and stop at different times. Commands typically take a number of arguments and @sequence requires two: the start time and duration, in seconds, of the oscillators that follow it until the next @sequence command is encountered.

```
;C major arpeggio
@sequence 0.0 0.2
sn3 $C4 1 1 1
@sequence 0.2 0.2
sn3 $E4 1 1 1
@sequence 0.4 0.2
sn3 $G4 1 1 1
@sequence 0.6 0.2
sn3 $C5 1 1 1
@sequence 0.8 0.2
sn3 $G4 1 1 1
@sequence 1.0 0.2
sn3 $E4 1 1 1
@sequence 1.2 0.4
sn3 $C4 1 1 1
```

@sequence commands do not need to come in any particular order, so it is possible to organise the input file in whatever way is most intuitive. The duration of the wav file is set to whichever is longer between the maximum length of the sequences *or* the duration specified on the command line.

1.12 Maths

`blow_chunks` can parse basic mathematical expressions: addition (+), subtraction (-), multiplication (*) and division(/). In conjunction with variables, this allows for much easier control over the input files. Taking the previous C major arpeggio, consider the steps needed to change the speed of the arpeggio. It would be necessary to change the @sequence start time and duration throughout. This can be avoided as follows:

```
;C major arpeggio with speed control
beatLen=0.2
timeStep=0.0
@sequence $timeStep $beatLen
sn3 $C4 1 1 1
timeStep=$timeStep+$beatLen
@sequence $timeStep $beatLen
sn3 $E4 1 1 1
timeStep=$timeStep+$beatLen
@sequence $timeStep $beatLen
sn3 $G4 1 1 1
timeStep=$timeStep+$beatLen
@sequence $timeStep $beatLen
sn3 $C5 1 1 1
timeStep=$timeStep+$beatLen
@sequence $timeStep $beatLen
sn3 $G4 1 1 1
timeStep=$timeStep+$beatLen
```

```
@sequence $timeStep $beatLen
sn3 $E4 1 1 1
timeStep=$timeStep+$beatLen
@sequence $timeStep $beatLen
sn3 $C4 1 1 1
```

This is a little more effort to set up but now there is a single variable (`beatLen`) which is the only thing needed to change the speed of the arpeggio. For longer files, this can save a lot of work and help avoid errors.

The order in which mathematical operations are carried out are multiplication and division first followed by addition and subtraction. `blow_chunks` doesn't support parentheses in mathematical expressions, and the mathematical operators must appear between two numbers (or variables which contain numbers).

Chapter 2

Oscillator reference

Oscillators all have the same arguments:

```
name frequency phase channel1_volume [[[ channel2_volume ] channel3_volume
    ] ... ]
```

Frequency is defined in Hertz and should be a positive number. Phase is measured from 0 to 1. Any value can be passed as the phase argument, but it will be wrapped into the range from 0 to 1, for example a value of 1.5 is the same as 0.5. The number of channels in the resulting wav file is determined by the number of channels used for the top-level oscillators and they must all have the same number of channels (`blow_chunks` will exit with an error message if this is not the case).

Modulators (i.e. oscillators that are used to modulate the arguments of another oscillator) have the same argument list but can only have a single volume specified, i.e.:

```
name frequency phase volume
```

2.1 `sin`

The `sin` oscillator uses the function from the C maths library. Sample output is shown in Figure 2.1.

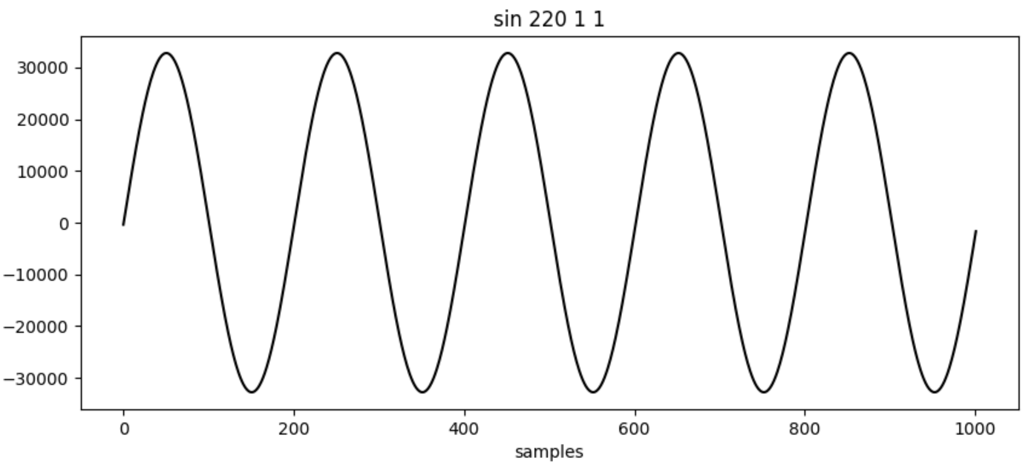

Figure 2.1: Output from sine wave operator at 220Hz

2.2 sqr

The `sqr` oscillator is a square wave. It is generated by calling the `sin` oscillator and setting the value to either -1 or +1 depending on whether the value of the sine wave is positive of negative. Sample output is shown in Figure 2.2. For most practical applications the smoothed `sqx` wave will produce nicer output.

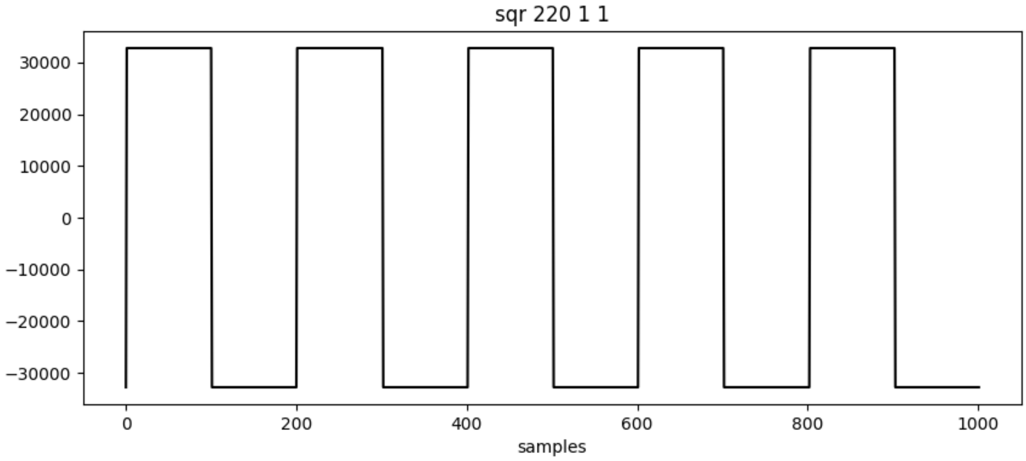

Figure 2.2: Output from square wave operator at 220Hz

2.3 sqx

The `sqx` oscillator is a smoothed square wave. Sample output is shown in Figure 2.2. The equation that generates the waveform is:

$$v = \frac{2}{\pi}\text{atan}\left(\sin\left(t\right)/\delta\right), \tag{2.1}$$

where v is the sample value, t is the time index and δ is a parameter the controls the degree of smoothing and is fixed at 0.005. The smoothed waveform will reduce audible clicking that is often generated by a square wave.

2.4 tri

2.5 trx

2.6 sup

2.7 sdn

2.8 sn3

2.9 sn5

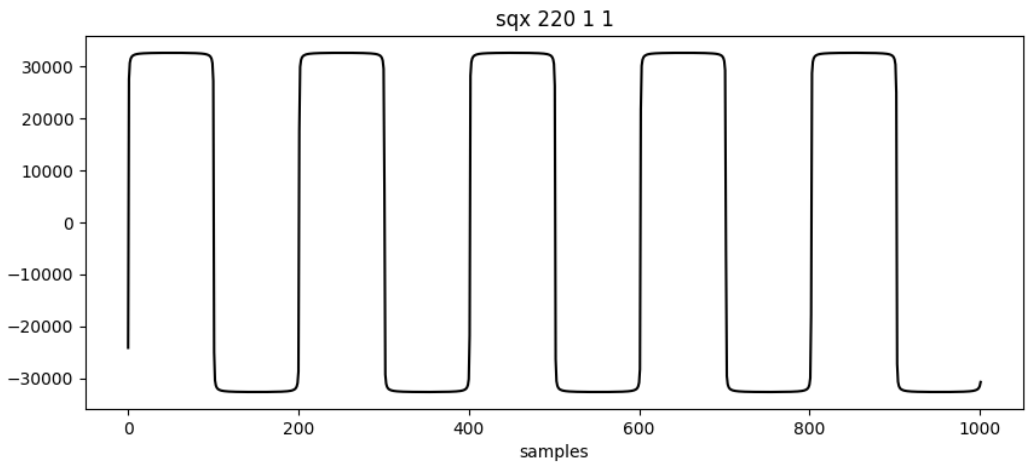

Figure 2.3: Output from the `sqx` square wave operator at 220Hz

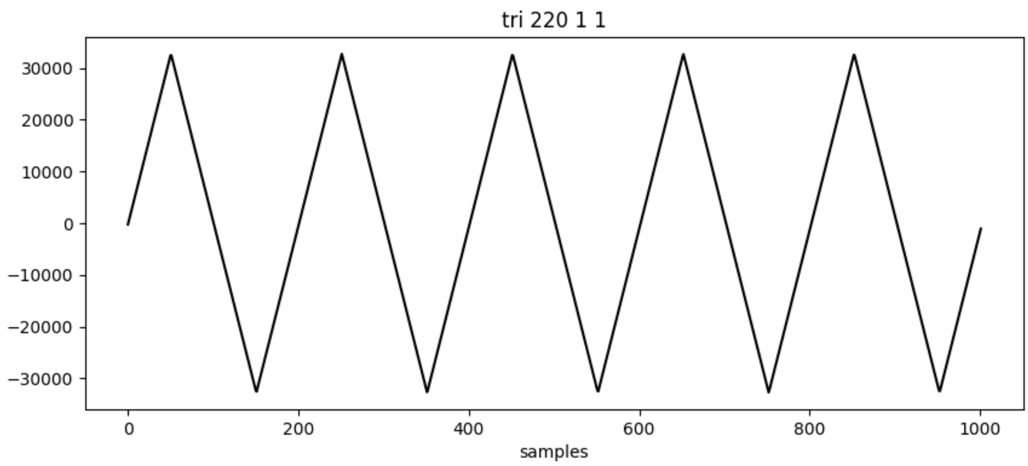

Figure 2.4: Output from the `tri` oscillator at 220Hz

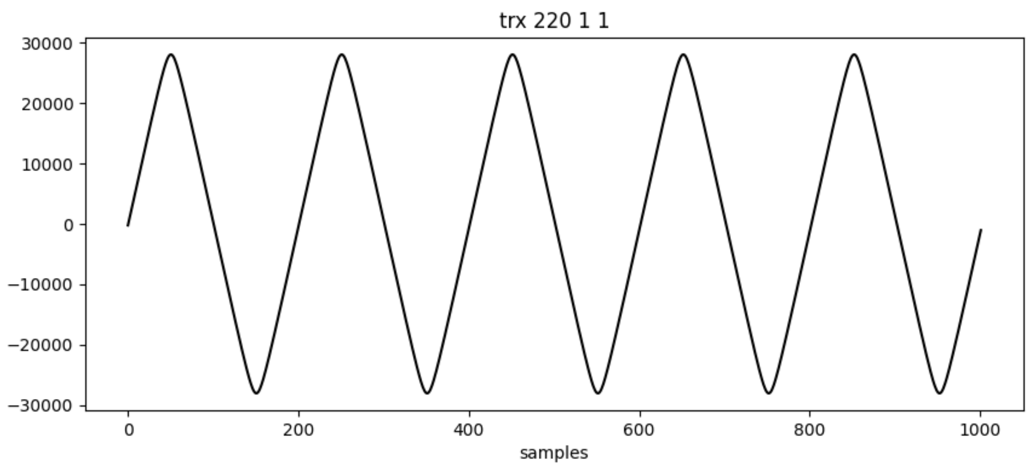

Figure 2.5: Output from the `trx` oscillator at 220Hz

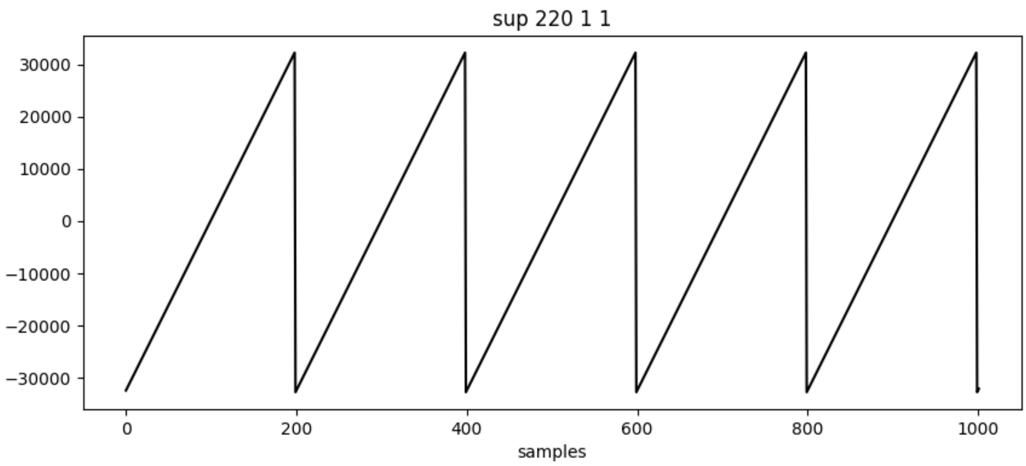

Figure 2.6: Output from the `sup` oscillator at 220Hz

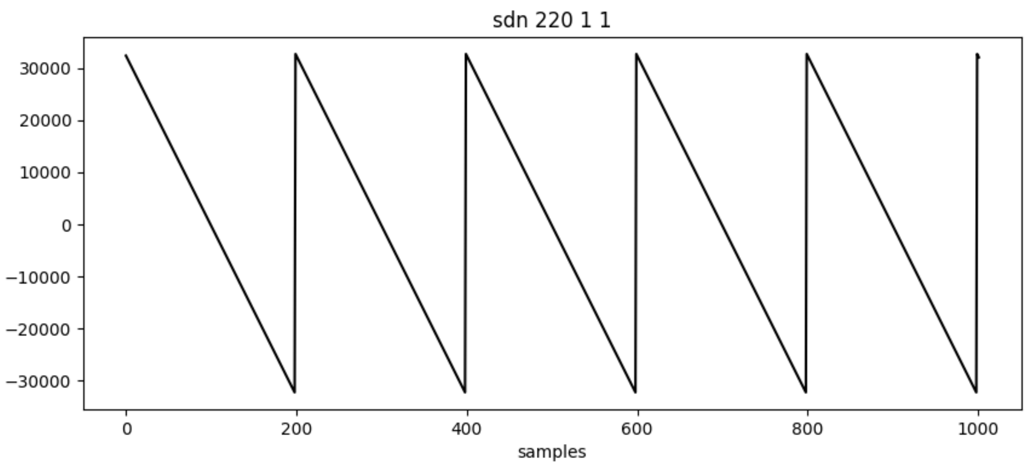

Figure 2.7: Output from the `sdn` oscillator at 220Hz

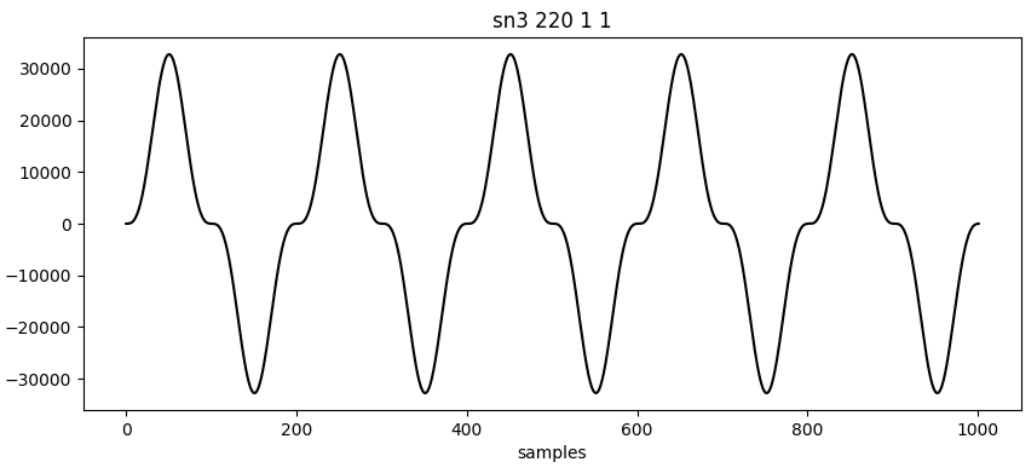

Figure 2.8: Output from the `sn3` oscillator at 220Hz

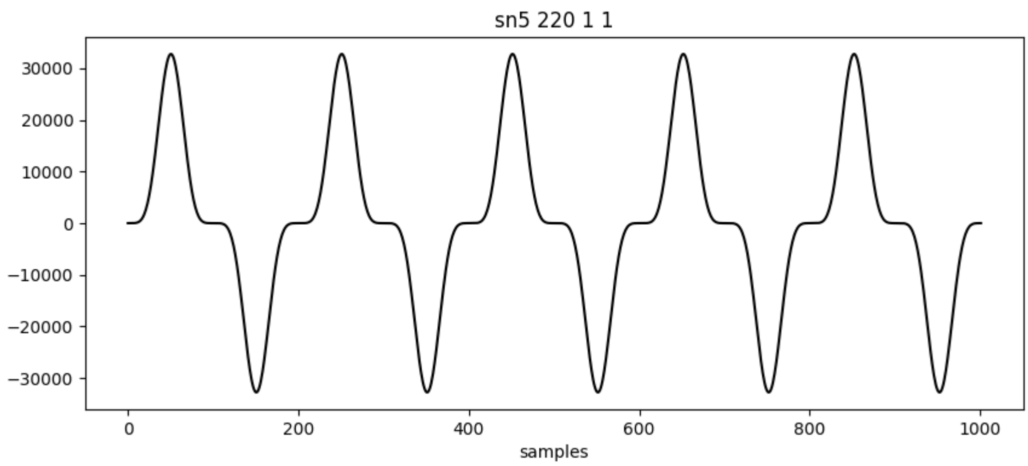

Figure 2.9: Output from the `sn5` oscillator at 220Hz