

blow\_chunks  
user manual

Tristan Quaife

March 8, 2025



---

# Contents

---

<b>1</b>	<b>Quick start</b>	<b>5</b>
1.1	Getting the code . . . . .	5
1.2	Compilation . . . . .	5
1.3	A first sound . . . . .	5
1.4	Basic <code>blow_chunks</code> instructions . . . . .	6
1.5	Modulating modulators . . . . .	6
1.6	Basic sound duration control . . . . .	7
1.7	Additive synthesis and working with files . . . . .	7



# 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 -
```

---

## 1.4 Basic `blow_chunks` instructions

What’s going on in the command described in the previous section? The command 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 `tri` which are a sine wave, a square wave and a triangular 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 triangular-wave oscillator to modulate the amplitude of the FM effect. Not that there’s nothing special about an oscillator that makes it “low frequency”; it is just a standard `blow_chunks` oscillator with a pitch that is below our hearing range. In this example, we will use a frequency of 1Hz:

---

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

---

## 1.7 Additive synthesis and working with files

As 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.