

A Manual for TrikTraks

Arun Chandra
arunc@evergreen.edu

Contents

1	Introduction: TRIKTRAKS	2
2	How to create and play a five-second sound	3
3	Changing the frequency over time	6
4	Playing the initial and final states	8
5	Changing the amplitude over time	9
6	Notes for the lazy and playful	10
7	Non-linear changes over time: why TRIKTRAKS is called TRIKTRAKS	11
8	Setting the number of “periods” of a path	13
9	Applying transformations to amplitudes	15
10	Specifying the range of a path	16
11	A variety of transformational paths	17
12	Command line options	21
13	Overall Format of Input Datafile	22
14	Acknowledgments	23

1 Introduction: TRIKTRAKS

TRIKTRAKS is a program for composing and synthesizing waveforms. You give TRIKTRAKS a datafile that specifies the waveform you want, and TRIKTRAKS generates a WAV soundfile which you then can hear.

Upon being given input, TRIKTRAKS will generate one sound. This sound can be arbitrarily long and arbitrarily complex.

For a general explanation on how a computer makes sound and the common fundamentals of both WIGOUT and TRIKTRAKS, please see the *The Synthesis Algorithms Used by TRIKTRAKS and WIGOUT* .

Whereas WIGOUT allows you to compose the waveform's shape and the rate at which it changes, TRIKTRAKS allows you to compose the waveform's transformational paths.

2 How to create and play a five-second sound

Let's say you'd like to create a 5-second tone. These are the steps:

1. Create a datafile with your input.
2. Save it as a "text" file.
3. Run TRIKTRAKS on your datafile to synthesize the sound.
4. Listen to the sound on your computer system.

Creating the datafile

First, create a datafile with the following information. You don't need to type the hash mark (#) or anything that follows it on a line. That's just a comment.

Use your favorite word processor. Type in all the information below, and then save it as a file called *data*. Be sure to save the resulting file as a plain text file. (Some word processors call this a "DOS file".) Remember to save the file in the same directory where your TRIKTRAKS program resides.

Here's the datafile to generate a 5-second sound.

```
output(sound.wav)    # name of the output soundfile
duration(5)          # duration of the transformation in seconds

20                   # element 0: duration in samples
1000                  # amplitude

30                   # element 1: duration in samples
-10000                # amplitude

40                   # element 2: duration in samples
10000                 # amplitude

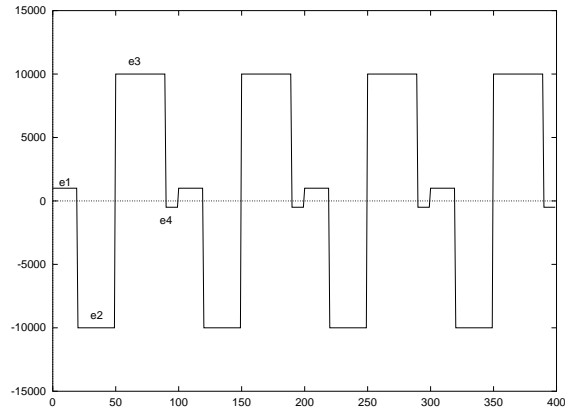
10                   # element 3: duration in samples
-500                  # amplitude
```

The first line has the command *output()*. This tells TRIKTRAKS to call your output soundfile "sound.wav". (It's a custom on Intel computers to give soundfiles the extension ".wav".)

The second line has the command *duration()*. This tells TRIKTRAKS how long you want the transformation to be, in seconds.

The following lines define the elements that make up your waveform. There are four elements defined in this datafile. The first line of each pair specifies the duration in samples of the element, and the second line specifies its amplitude.

These elements create a waveform that looks like this:



Running TRIKTRAKS

After you've saved your datafile (and named it "data"), exit your word processor.

(If you're using Windows95 or WindowsNT, open up a DOS window.)

Change directory to your TRIKTRAKS directory.

To synthesize the sound, type the following:

```
tt data
```

TRIKTRAKS will respond with

```

initial final type max min periods degree am fc fm seed
e0s 20 20 zero 20 20 1 0 0.000 0.000 0.000 0
a 1000 1000 zero 1000 1000 1 0 0.000 0.000 0.000 0
e1s 30 30 zero 30 30 1 0 0.000 0.000 0.000 0
a -10000 -10000 zero -10000 -10000 1 0 0.000 0.000 0.000 0
e2s 40 40 zero 40 40 1 0 0.000 0.000 0.000 0
a 10000 10000 zero 10000 10000 1 0 0.000 0.000 0.000 0
e3s 10 10 zero 10 10 1 0 0.000 0.000 0.000 0
a -500 -500 zero -500 -500 1 0 0.000 0.000 0.000 0
writing samples 10% 20% 30% 40% 50% 60% 70% 80% 90% Done. 4800 states.
goal arrival difference
e0: 20 20 0
1000 1000 0
e1: 30 30 0
-10000 -10000 0
e2: 40 40 0
10000 10000 0
e3: 10 10 0
-500 -500 0
sound samples in sound.wav
```

This means the sound has been successfully created, and the samples are in the soundfile "sound.wav", in the same directory as your TRIKTRAKS executable.

Using the Windows FileManager, find the output soundfile on your hard disk ("sound.wav"), and click twice on it. That should open up the Windows MediaPlayer, and with it you should be able to play and hear the soundfile.

The steps to create and listen to a sound

So, to reiterate, the steps needed for creating a sound using TRIKTRAKS are

1. Create a datafile using a word processor. Save the file as a plain text file (or DOS file). Remember to save the datafile in the same directory as your tt.exe executable resides.
2. Open up a DOS window. Change to the TRIKTRAKS directory.
3. Run TRIKTRAKS on your input datafile. That will create the output soundfile.
4. Open up the program MediaPlayer.
5. Select the output soundfile in your TRIKTRAKS directory, and play it.

3 Changing the frequency over time

Here's the same datafile as above.

```
output(sound.wav) # name of the output soundfile
duration(5)       # duration of the transformation

20               # element 0: duration in samples
1000             # amplitude

30               # element 1: duration in samples
-10000          # amplitude

40               # element 2: duration in samples
10000           # amplitude

10               # element 3: duration in samples
-500            # amplitude
```

This created a static tone for 5 seconds.

Make the following changes to your datafile:

```
output(sound.wav) # name of the output soundfile
duration(5)       # duration of the transformation

20 40            # element 0: duration in samples
1000            # amplitude

30 60            # element 1: duration in samples
-10000         # amplitude

40 80            # element 2: duration in samples
10000          # amplitude

10 20            # element 3: duration in samples
-500           # amplitude
```

Element 0 (duration) now is given two values: an initial value of 20 samples and a final value of 40 samples. This means that this element will begin with a duration of 20 samples and will slowly change until, at the end of the transformation, its duration will be 40 samples.

Something similar happens to elements 1, 2, and 3: their durations begin with one value and end with a value twice as big.

For all the elements, the amplitude stays the same.

Make the changes above, save the file as “data2”, run TRIKTRAKS on it, and play the result in the MediaPlayer. You should hear a 5-second sound that goes down in pitch.

Determining the starting and ending frequency

[Note: this section can be skipped on first reading.]

OK, so that generates a glissando from the initial to the final state. How do you know the initial and final frequencies?

Here's the general formula for determining the frequency, given the sampling rate and the state duration:

$$frequency = sampling_rate / state_duration$$

The duration of the state is the sum of durations of its elements. So the initial state has a duration of

$$state_duration = 20 + 30 + 40 + 10 = 100 \text{ samples}$$

The default sampling rate for TRIKTRAKS is 44100 samples per second, so to find the frequency, you solve for the frequency:

$$freq = 44100/100$$

$$freq = 441.0$$

The initial frequency is 441 hertz.

The final frequency can be determined in the same way:

$$freq = 44100/(40 + 60 + 80 + 20)$$

$$freq = 44100/200$$

$$freq = 220.5$$

The final frequency is 220.5 hertz. The sound starts at 441.0 hertz, and descends until it ends at 220.5 hertz, which is one octave lower.

4 Playing the initial and final states

With the above datafile, the sound starts and its frequency immediately begins descending. When it reaches its goal, the sound immediately stops.

Sometimes it's useful to hear the initial and final states as steady tones, before and after the transformation. Here's how you can do it:

```
output(sound.wav)    # name of the output soundfile
duration(5)          # duration of the sound
initial(1)           # play the initial state for these many seconds
final(1)             # play the final state for these many seconds

20 40                # element 0: duration in samples
1000                 # amplitude

30 60                # element 1: duration in samples
-10000              # amplitude

40 80                # element 2: duration in samples
10000               # amplitude

10 20                # element 3: duration in samples
-500                # amplitude
```

Two commands are given to TRIKTRAKS: *initial(1)* and *final(1)*.

This means that the initial state will be played for 1 second, after which the transformation will begin, reach its final state, and then the final state will be played for 1 second. The total duration of the sound will thus be 7 seconds.

So you will hear the first state for 1 second, a 5 second glissando to the final state, and then the final state for 1 second.

Easy, eh?

5 Changing the amplitude over time

Let's suppose you wanted the tone to make a decrescendo over its duration.

Make the following changes to the datafile you created in the last step:

```
output(sound.wav)  # name of the output soundfile
duration(5)        # duration of the sound
initial(1)         # play the initial state for these many seconds
final(1)           # play the final state for these many seconds

20 40              # element 0: duration in samples
1000 -500          # amplitude

30 60              # element 1: duration in samples
-10000 500         # amplitude

40 80              # element 2: duration in samples
10000 -500         # amplitude

10 20              # element 3: duration in samples
-500 500           # amplitude
```

Notice that the amplitude for element 0 changes from 1000 to -500, element 1 changes from -10000 to 500, and elements 2 and 3 also change their amplitudes.

Save the file as “data3”, run TRIKTRAKS on it, and play the resulting soundfile with MediaPlayer. You should hear the same glissando, except now it will reduce in volume.

If you wanted to make a crescendo, you would do the same thing, only making sure that the greatest difference of amplitudes in the final state is larger than greatest difference in the initial state.

Determining the initial and final amplitude

[Note: this section can be skipped on first reading.]

How do you determine the initial (or the final) dynamic? By determining the greatest difference between amplitudes in a state.

In the initial state, the greatest difference of amplitudes is between elements 1 and 2, 10000 and -10000, so

$$10000 - -10000 = 20000$$

The difference is 20000. That can be translated into decibels by

$$20 * \log_{10}(20000) = 86.02 \text{ dB}$$

So the initial dynamic is about 86 dB.

Let's say you wanted a final dynamic of 60 dB. What is the greatest difference in amplitudes you must have?

To translate dB into amplitudes, use the following formula:

$$\text{amplitude} = 10^{\text{decibels}/20}$$

$$1000 = 10^{60/20}$$

So if you want a level of 60 dB, you need the greatest difference to be 1000, which is 500 to -500.

6 Notes for the lazy and playful

You don't need to know the exact value in decibels or the exact frequency in hertz in order to use TRIKTRAKS. All you need to do is to feed TRIKTRAKS numbers, and it will create the frequencies, changes, etc.

The best way to play with TRIKTRAKS is to put in arbitrary numbers, and hear what happens.

When you get to a point where you might want to make a composition with TRIKTRAKS, then it can be helpful to know exactly what is going on in terms of hertz and decibels.

7 Non-linear changes over time: why TRIKTRAKS is called TRIK-TRAKS

TRIKTRAKS starts being fun when you abandon linear changes over time and start in with non-linear ones.

Linear changes are a default in this program. They are “stipulated” by not calling for something else. But they are nothing special. All that is required is *some* path for the changes to follow. The shape of the path will affect the sound.

Take the same data file as above, without the amplitude changes, and add the following to it:

```
output(sound.wav)    # name of the output soundfile
duration(5)          # duration of the transformation
initial(1)           # play the initial state for these many seconds
final(1)             # play the final state for these many seconds

20 40                # element 0: duration in samples
1000                 # amplitude

30 60 sin            # element 1: duration in samples
-10000              # amplitude

40 80                # element 2: duration in samples
10000               # amplitude

10 20 sin            # element 3: duration in samples
-500                # amplitude
```

Save the file, run TRIKTRAKS on it, and listen to the output. You should again hear an initial 440-hertz tone, a final 220-hertz tone, but a varying glissando in the middle.

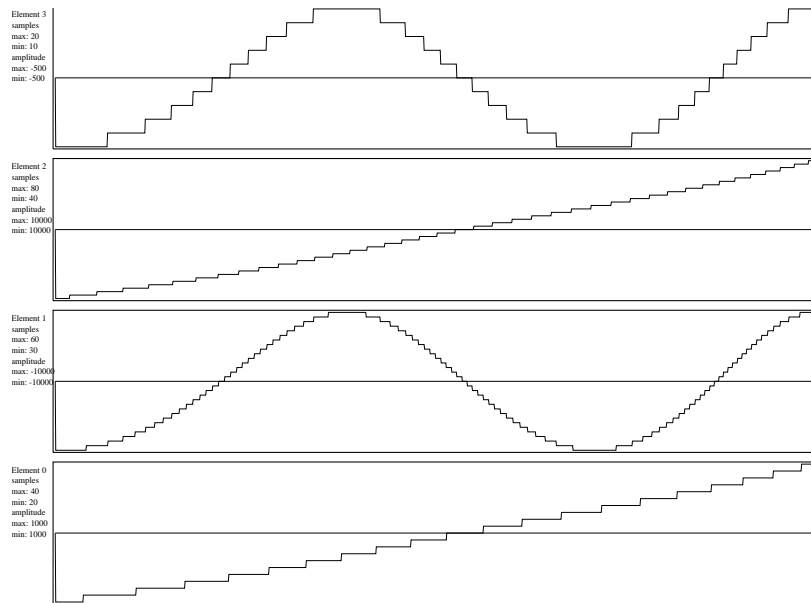
So what’s going on?

Element 0 has an initial duration of 20 samples, and a final duration of 40 samples. This means that over 5 seconds, the value 20 slowly increases until it gets to 40 (20, 21, 22, 23, etc.). This increment happens evenly over the course of 5 seconds.

Element 1 has an initial duration of 30 samples, and a final duration of 60 samples, but follows the path of a sine wave to get from the initial to the final value.

Similarly with elements 2 and 3: 2 follows a linear path from 40 to 80, and 3 follows the path of a sine wave from 10 to 20.

Here’s a plot of what the transformations look like for all four elements. (Note: this is a plot of the *transformations*, and not of the sound itself.) The elements are listed bottom to top, with the first element on the bottom.



The horizontal axis represents *iterations* (or time), and the vertical axis represents the normalized magnitude of the variable. (“Normalized magnitude” just means that the values have been scaled to fit together on one plot.)

In this plot, the first element (at the bottom) has a choppy diagonal line, and a horizontal line in the center. The choppy, diagonal line represents the duration, ascending from 20 to 40 samples. The horizontal line represents the amplitude, which does not change.

The second element (second from bottom) has a sine wave and a horizontal line. The sine wave represents the duration, oscillating between 30 and 60 samples, and the horizontal line represents the amplitude, which is fixed at -10000. (The other two elements should be interpreted in a similar way.)

So, in this plot you can see that the durations of elements 1 and 3 follow the path of a sine wave, whereas the durations for elements 0 and 2 follow a linear path.

So, elements 0 and 2 are getting steadily longer, but elements 1 and 3 first get longer, then shorter, then longer again.

Since the frequency you hear is dependent on the *sum* of durations of all four elements at one moment, the frequency you hear keeps changing.

This changing “path” of durations (and of amplitudes too) is where the name comes from: TRIKTRAKS.

8 Setting the number of “periods” of a path

The path followed by a transformation can be given a number of “periods”. When you request a path, by default TRIKTRAKS gives you at least one period, and less than two.

You can specify the minimum number of periods you’d like in the following way:

```
output(sound.wav)    # name of the output soundfile
duration(5)          # duration of the transformation
initial(1)           # play the initial state for these many seconds
final(1)             # play the final state for these many seconds

20 40                # element 0: duration in samples
1000                 # amplitude

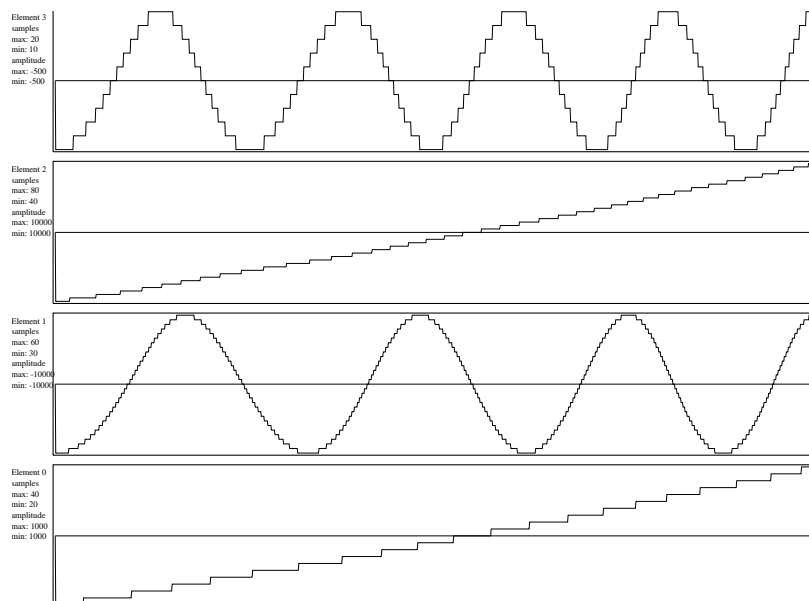
30 60 sin periods(3) # element 1: duration in samples
-10000              # amplitude

40 80                # element 2: duration in samples
10000               # amplitude

10 20 sin periods(4) # element 3: duration in samples
-500                # amplitude
```

The above datafile is identical with the previous one, except for the addition of the period specifications. Now, the 2nd element’s duration follows the path of a sine wave that will have at least 3 periods. The 4th element’s duration follows the path of a sine wave that will have at least 5 periods.

Try synthesizing this sound, and listen to it. The plot of the transformation looks like this:



You can give high numbers for the periods as well. Acoustically things get very messy, and intriguing, very fast. Here’s the previous example, but with a high number of periods for one of the elements:

```
output(sound.wav)    # name of the output soundfile
duration(5)          # duration of the transformation
initial(1)           # play the initial state for these many seconds
final(1)             # play the final state for these many seconds
```

```

20 40          # element 0: duration in samples
1000          # amplitude

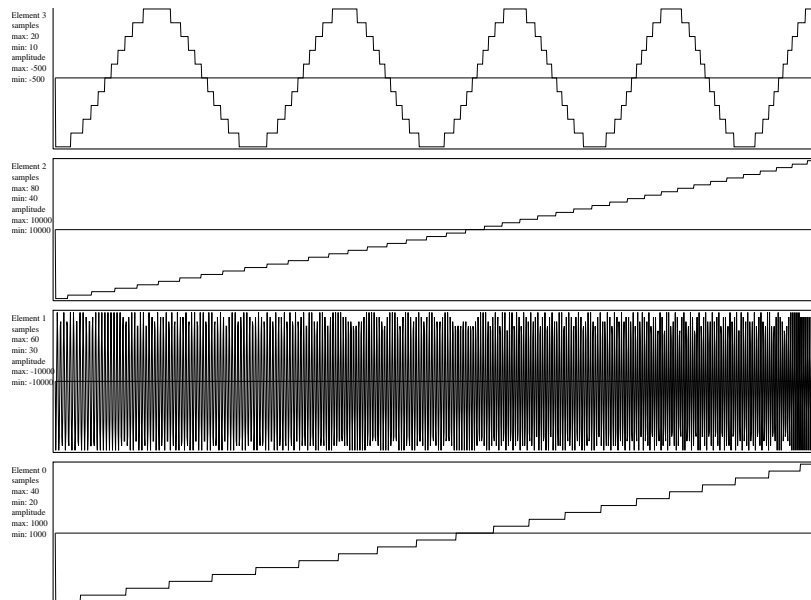
30 60 sin periods(300) # element 1: duration in samples
-10000       # amplitude

40 80          # element 2: duration in samples
10000        # amplitude

10 20 sin periods(4) # element 3: duration in samples
-500         # amplitude

```

And here's the plot of the transformations:



9 Applying transformations to amplitudes

All of the transformation paths above were applied only to the elements' durations. They can also be applied to the elements' amplitudes.

Here's an example similar to the last few, where both the durations are changing for elements 1 and 3, and the amplitudes are changing for elements 0 and 2:

```
output(sound.wav)    # name of the output soundfile
duration(5)          # duration of the transformation
initial(1)           # play the initial state for these many seconds
final(1)             # play the final state for these many seconds

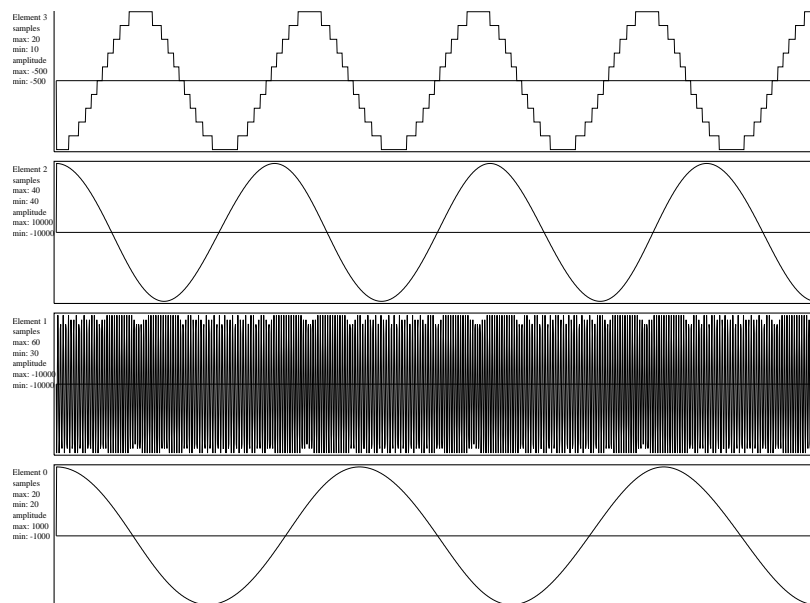
20 40                # element 0: duration in samples
1000 -1000 sin periods(2) # amplitude

30 60 sin periods(300) # element 1: duration in samples
-10000                # amplitude

40 80                # element 2: duration in samples
10000 -10000 sin periods(3) # amplitude

10 20 sin periods(4) # element 3: duration in samples
-500                  # amplitude
```

Here's the plot of the transformations:



Make these changes to your datafile and hear what it sounds like.

10 Specifying the range of a path

In all the above examples, the chosen path is constrained to the limits of its initial and final values.

So, if the initial value is 10 and the final value is 40, that means that a path will oscillate between the values of 10 and 40.

The range of the path can be specified, if you wish:

```
output(sound.wav)    # name of the output soundfile
duration(5)          # duration of the transformation
initial(1)           # play the initial state for these many seconds
final(1)             # play the final state for these many seconds

20 40                # element 0: duration in samples
1000 -1000 sin periods(2) # amplitude

30 60 sin range(100,10) periods(300) # element 1: duration in samples
-10000               # amplitude

40 80                # element 2: duration in samples
10000 -10000 sin periods(3) # amplitude

10 20 sin range(50,5) periods(4) # element 3: duration in samples
-500                 # amplitude
```

In this example, the path of element 1 (duration) is given a range greater than its initial and final values.

The specified range cannot be less than the initial and final values. If it is, then TRIKTRAKS will silently change the range to match the initial and final values.

Try it, and hear what it sounds like!

11 A variety of transformational paths

In all the above examples, the only path I showed you was the sine wave. There are other paths which can also be tried out.

Standard paths

- sin
- squ (square wave)
- saw (sawtooth wave)
- tri (triangle wave)

These four paths are used just like the sin path was used above. Try replacing some of the “sin” path with some of these other paths. Of course, you can mix and match paths as you prefer: for example, and element’s duration could follow the path of a square wave, while its amplitude follows the path of a sawtooth wave.

Here’s a datafile to create a sound using all the standard paths, followed by a plot of the paths used.

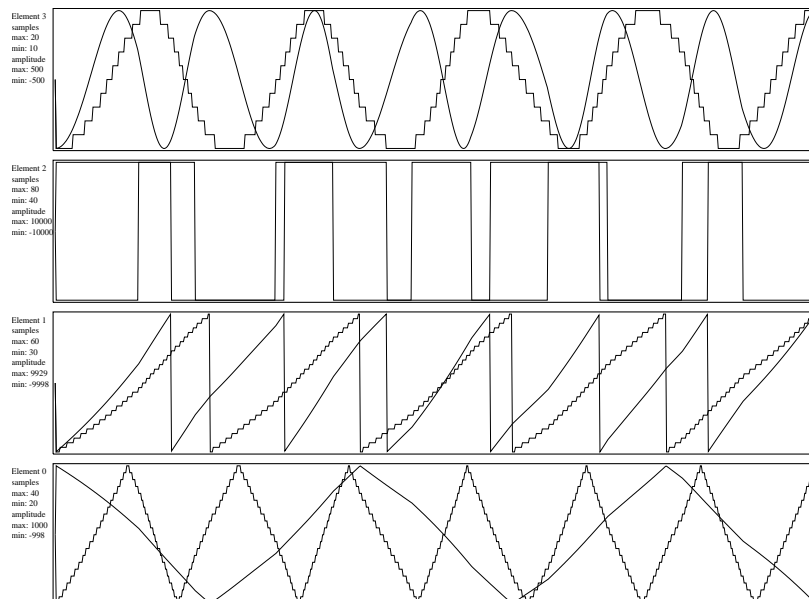
```
output(sound.wav)      # name of the output soundfile
duration(5)            # duration of the transformation
initial(1)             # play the initial state for these many seconds
final(1)               # play the final state for these many seconds

20 40 tri periods(6) # element 0: duration in samples
1000 -1000 tri periods(2) # amplitude

30 60 saw periods(4) # element 1: duration in samples
-10000 10000 saw periods(6) # amplitude

40 80 squ periods(5) # element 2: duration in samples
10000 -10000 squ periods(3) # amplitude

10 20 sin periods(4) # element 3: duration in samples
-500 500 sin periods(7) # amplitude
```



Polynomial paths

The polynomial path is slightly different than the other paths, in that it takes an argument, which is the degree of the polynomial to be generated. The degree can be in the range of 3 to 10 (poly(3) to poly(10)).

So you might use it like this:

```
20 40 poly(5) periods(7) range(50,10)
10000 -10000 poly(3) periods(4) range(30000,-30000)
```

Here's a datafile that uses polynomial paths, followed by a plot of the paths.

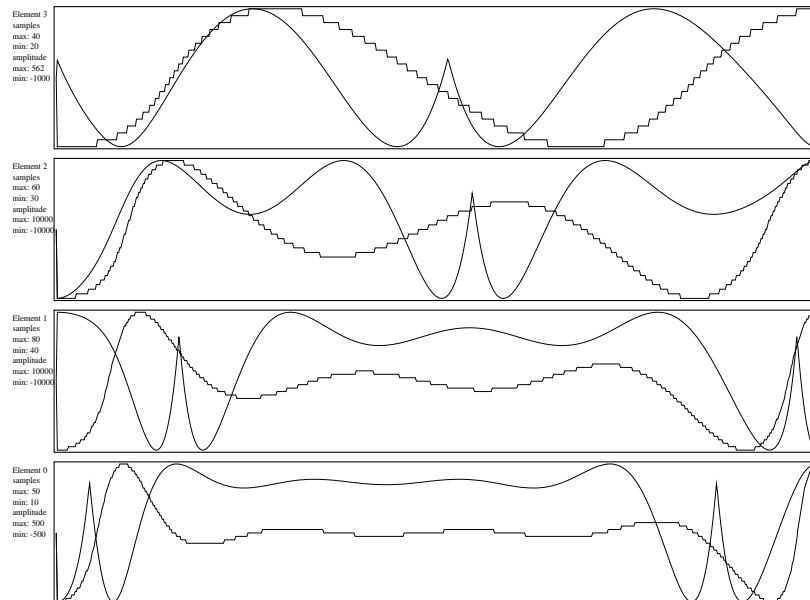
```
output(sound.wav)      # name of the output soundfile
duration(5)            # duration of the transformation
initial(1)             # play the initial state for these many seconds
final(1)               # play the final state for these many seconds

10 20 poly(9)          # element 0: duration in samples
-500 500 poly(10)     # amplitude

40 80 poly(7)          # element 1: duration in samples
10000 -10000 poly(8) # amplitude

30 60 poly(5)          # element 2: duration in samples
-10000 10000 poly(6) # amplitude

20 40 poly(3)          # element 3: duration in samples
1000 -1000 poly(4)    # amplitude
```



AMFM paths

The amfm path is the most complicated path of all. Here is the function used to generate it:

$$f(t) = (1 + \sin(2\pi a_m t / sr)) * \sin(2\pi f_c t / sr + f_c / f_m * \sin(2\pi f_m t / sr))$$

Never mind if you don't follow the math here. What's significant about this is that it takes the John Chowning's "classic" function for frequency modulation:

$$FM(t) = \sin(2\pi f_c t / sr + f_c / f_m * \sin(2\pi f_m t / sr))$$

and applies the amplitude modulation function to it:

$$AM(t) = (1 + \sin(2\pi a_m t / sr))$$

There are 3 variables you need to set: a_m , f_c , f_m . Here's how you can do it:

```
20 40 amfm(am=0.1, fc=0.2, fm=0.3) periods(10)
1000 -1000 amfm(am=0.01, fc=0.02, fm=0.03) range(5000,-5000)
```

Of course, you can also give the number of periods and the range to this path, as shown above.

Here is a datafile that uses the amfm paths, followed by a plot of the paths.

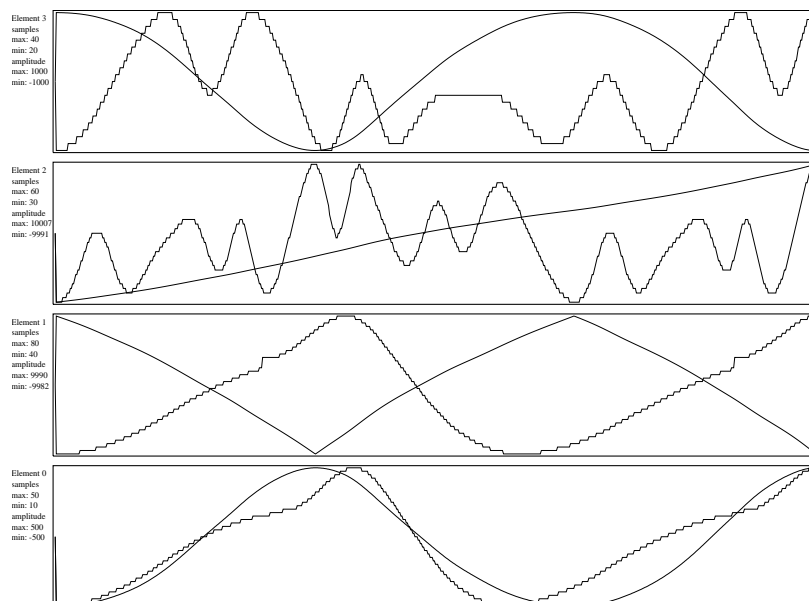
```
output(sound.wav) # name of the output soundfile
duration(5) # duration of the transformation
initial(1) # play the initial state for these many seconds
final(1) # play the final state for these many seconds

10 50 amfm(am=0.1,fc=0.2, fm=0.3) # duration in samples
-500 500 sin # amplitude

40 80 amfm(am=0.01,fc=0.02,fm=0.03) # duration in samples
10000 -10000 tri # amplitude

30 60 amfm(am=0.9, fm=0.8, fc=0.7) # duration in samples
-10000 10000 # amplitude

20 40 amfm(am=0.6, fm=0.5, fc=0.4) # duration in samples
1000 -1000 sin # amplitude
```



The values for am , fm , and fc can be any floating-point number. Values greater than 1 tend to become “noisy” very fast.

12 Command line options

Here's a list of all the command line options, and what they do:

<code>-o <i>soundfile</i></code>	put output in <i>soundfile</i> (default: <i>tt.wav</i>)
<code>-R <i>rate</i></code>	set sampling rate to <i>rate</i> (default: 44100)
<code>-S</code>	show input data to the screen
<code>-r</code>	ramp the first and final states of initial and final
<code>-s <i>dur</i></code>	add <i>dur</i> seconds of silence at end of sound
<code>-t</code>	make tracing files of paths created
<code>-d</code>	debug

Specifying the output soundfile: -o If you wish, you can specify the name of the output soundfile on the command line:

```
tt -o newfile.wav mydata
```

Of course, you can also put this information into the datafile itself.

Setting the sampling rate: -R *rate* TRIKTRAKS uses a default sampling rate of 44100 samples per second. This can be changed on the command line this way:

```
tt -R 48000 mydata
```

This would set the sampling rate to 48000 samples per second.

TRIKTRAKS only recognizes the following sampling rates: 11025, 22050, 44100, 48000, and 96000.

Show input data to the screen: -S This option displays to the screen the input data TRIKTRAKS thinks its using.

Ramp the initial and final states: -r This will cause a the first state of the waveform to be ramped from 0 to 1 (or “nothing” to “full”) over the duration of that first state, and the final state of the waveform to be ramped the opposite way, from 1 to 0.

```
tt -r mydata
```

Add silence to the end of the soundfile: -s *dur* This will cause *dur* seconds of silence to be added at the end of the soundfile.

```
tt -s 10 mydata
```

The above will add 10 seconds of silence to the sound generated by *mydata*.

Creating plots of the transformations: -t TRIKTRAKS can create plots of the transformations your specify. Just specify *-t* on the command line, and TRIKTRAKS will output a Encapsulated PostScript file that you can view or print. All the examples in this document were generated with TRIKTRAKS.

```
tt -t datafile
```

Debugging information: -d This option will print debugging information, along with ASCII files of all the paths followed by your transformation. This is marginally helpful to a programmer.

```
tt -d mydata
```

It will create some files “ad.x”, and some files called “wt.xa” and “wt.xs”, where “x” is a number.

13 Overall Format of Input Datafile

Initialization

duration()	duration of the transformation in seconds.
initial()	length of time in seconds to play the initial state.
final()	length of time in seconds to play the final state.
silence()	length of time in seconds to add silence to the end of the sound.
output()	name of the output file.

```
initial(3)          # play initial state for these many seconds
duration(10)       # duration of transformation in seconds
final(4)           # play final state for these many seconds
silence(5.75)      # add these many seconds of silence to end of sound
output(grumpy.wav) # name for output soundfile
```

Initial and Final States

Transformation

14 Acknowledgments

So that's TRIKTRAKS! I hope you have some fun with it, and please drop me a line if the program crashes on you, or just stares blankly back at you from the computer screen.

I'd like to thank Professors Jim Beauchamp and Sever Tipei, co-directors of the Computer Music Studios, University of Illinois, who were encouraging to my work, and generous with access to the Studios.

Herbert Brün and Keith Johnson taught me about SAWDUST, the principles of its algorithms, and its radical address to compositional premises.

Conversations with Michael Brün were very helpful in developing and exploring the amfm algorithm.

The program was originally written in C on a 68040 NeXTStation running NeXTStep 3.1. Then it was rewritten in C++, and ported to an IBM RS6000 running AIX 3.1, an SGI Indy running IRIX 5.3, an SGI O2 running IRIX 6.3, and an Intel PC running Windows95. On all these machines, the GNU C compiler was used to compile the code.

Most of the graphics in this program were produced with TRIKTRAKS. Some were produced with *gnuplot*, version 3.5, written by Colin Kelley and Thomas Williams.