

# A Manual for SAWDUST

Herbert Brün

Designed and Implemented by Gary Grossman

Enhanced by Jody Kravitz

Ported and maintained by Keith Johnson and Arun Chandra

Manual written by Arun Chandra

[arunc@evergreen.edu](mailto:arunc@evergreen.edu)

# Contents

<b>1</b>	<b>SAWDUST: Computer music project</b>	<b>4</b>
1.1	A brief history of the project . . . . .	4
1.2	Excerpt from the original proposal . . . . .	4
<b>2</b>	<b>General Notes</b>	<b>6</b>
2.1	Starting the Sawdust program . . . . .	6
2.2	Terminating a command . . . . .	6
2.3	Objects . . . . .	7
2.4	Help: display commands understood by Sawdust . . . . .	7
<b>3</b>	<b>element, elist: define one or more elements</b>	<b>8</b>
3.1	The element command: define one element . . . . .	8
3.2	The elist command: define a list of elements . . . . .	8
<b>4</b>	<b>link: define a link</b>	<b>9</b>
4.1	The link command . . . . .	9
<b>5</b>	<b>Combining elements to form links</b>	<b>10</b>
<b>6</b>	<b>Combining Links: mingle and merge</b>	<b>11</b>
6.1	mingle: concatenate a sequence of links . . . . .	11
6.2	merge: interlace the constituent elements of a sequence of links . . . . .	11
6.3	Calculating the duration and frequency of merges and mingles . . . . .	11
6.4	Examples of a merge and a mingle . . . . .	13
<b>7</b>	<b>Transforming <i>linkIn</i> into <i>linkOut</i>: vary</b>	<b>14</b>
7.1	The algorithm for vary . . . . .	15
7.2	The command vary . . . . .	15
7.3	steps or time? . . . . .	16
7.4	freeze? . . . . .	17
7.5	amplitude and sample degrees . . . . .	17
7.6	upper and lower sample bounds . . . . .	17
7.7	statements . . . . .	17
<b>8</b>	<b>Transforming <i>linkIn</i> into <i>linkOut</i>: turn</b>	<b>18</b>
8.1	The problem addressed by <i>turn</i> . . . . .	19
8.2	The <i>turn</i> algorithm . . . . .	19
8.3	<i>turn</i> amplitudes . . . . .	19
8.4	<i>turn</i> samples . . . . .	20
8.5	The command turn . . . . .	21
8.6	name, first link name, and last link name . . . . .	21
8.7	position? and rotate . . . . .	21
8.8	Amplitude range; use default values? . . . . .	21
8.9	turnsum? . . . . .	22
8.10	statements and Wavy? . . . . .	22
<b>9</b>	<b>Displaying and Removing Objects: show, status, delete, forget</b>	<b>23</b>
9.1	The show command . . . . .	23
9.2	The status command . . . . .	23
9.3	The delete command . . . . .	24
9.4	The forget command . . . . .	24

<b>10 Data Input/Output: save, restore, write, read</b>	<b>25</b>
10.1 The save and restore commands . . . . .	25
10.2 The write and read commands . . . . .	25
<b>11 Creating and Playing samples: play, replay, store</b>	<b>26</b>
11.1 The play command . . . . .	26
11.2 The store command . . . . .	26
11.3 The replay command . . . . .	27
11.4 Note for the play and store commands . . . . .	27
11.5 Generating silence with the play and store commands . . . . .	27
<b>12 Soundfile Manipulation: name, clear</b>	<b>28</b>
<b>13 Setting Variables: <i>exact</i>, <i>seed</i>, <i>vquick</i>, <i>range</i></b>	<b>29</b>
<b>14 Gary Grossman: Instruments, Cybernetics, and Computer Music (1987)</b>	<b>30</b>
<b>15 Example of a Sawdust Session</b>	<b>35</b>

# 1 SAWDUST: Computer music project

SAWDUST is a computer program for composing waveforms. It was conceived by Herbert Brün, designed and implemented by Gary Grossman, and enhanced by Jody Kravitz, ported and maintained by Keith Johnson and Arun Chandra.

Brün has written about the program:

The computer program which I called SAWDUST allows me to work with the smallest parts of waveforms, to link them and to mingle or merge them with one another. Once composed, the links and mixtures are treated, by repetition, as periods, or by various degrees of continuous change, as passing moments of orientation in a process of transformations.

## 1.1 A brief history of the project

In 1972, Brün made a proposal to the University of Illinois Research Board to fund the SAWDUST project. In 1976, Gary Grossman completed the first version of SAWDUST, written in the then-new C programming language under UNIX, and running on a PDP 11/50 at the University of Illinois Digital Computer Lab. To this computer were attached a digital-to-analog converter (built by Jody Kravitz), a 1/4", half-track, reel-to-reel tape recorder, and a stereo amplifier and speakers. The waveforms were generated by the computer, transferred to analog tape, then taken to the University of Illinois Electronic Music Studios for mixing. With this system, Brün wrote *Dust* (1976), *More Dust* (1977), *Dustiny* (1978), and *A Mere Ripple* (1979).

Jody Kravitz, using functions given by Elizabeth Mitro, wrote the code for the *turn* algorithm. With this second version of SAWDUST, Brün completed *U-TURN-TO* (1980) and *i toLD YOu so!* (1981).

In the late 1980s, Keith Johnson ported SAWDUST to 16-bit PCs, built a custom D/A and A/D converter for the PC, and enabled SAWDUST to work with a sound i/o system developed at the University of Iowa by David Muller and Adam Cain. With this version of the program, Brün completed *Aufhören!* (1989) for ensemble and tape, and *on stilts among ducks* (1997) for viola and tape.

In the middle 1990s, Arun Chandra (the current maintainer) ported SAWDUST back to Unix systems (NeXT, IBM RSC-6000, and SGIs). SAWDUST now uses PortAudio 18.1 for its sound output, and runs under 32-bit Windows, Linux, and Mac OS X systems

## 1.2 Excerpt from the original proposal

In 1972, Brün wrote the following in his original proposal to the University of Illinois Research Board:

**The current direction:** Computer assisted sound synthesis, when related to musical composition, has been based on the validity of the following observations: The description of a desired sound can be transformed into a set of instructions within a computer program. Under the control of this program the computer system will output a digital approximation to one of the many waveforms that would, upon D/A conversion, generate the described sound.

With respect to the description of a sound, many waveforms are equivalent.

This was, and still is, satisfactory for musical purposes because of a seemingly trivial truism: Even the discriminating ear of a professional musician is incapable of distinguishing from one another the numerous different wave forms which, all, would generate, according to this musician's description, one and the same sound. As, in this respect, the composer is no better than the musician, and as the composer can only compose when and where he either discovers or draws controllable distinctions, he never consciously composes in waveforms, rarely listens to waveforms. It will not dampen any composer's gratitude for two flawless performances of his work, if he is shown, afterwards, that the waveforms, generated by the performances and filmed via Oscilloscope, had produced two totally different movies.

The history of musical composition has been characterized by technological restrictions: The composer, the musician, the listener, all had to find their musical universe through descriptions of sounds and sound relations, since it was technically impossible to extend compositional instructions and performing control

to the level of the waveform, its constituents, and its transformational properties. Even the centuries old search for ever new instruments, new waveform generators, was a search for sound and, like everything else in music, guided by two main criteria: 1) With respect to the description of a desired sound (as heard), many waveforms are equivalent; 2) This relation of “One to Many” is not only satisfactory and, given the wide tolerance margin in our auditory system, an advantage; it is, in the absence of applicable technical alternatives, a kind of fundamental condition to be taken for granted. Thus, in music, not the waveform, but the sound as heard and described is the basic element and standard.

The current direction has overlooked the advent of the applicable technical alternative.

**The discrete step:** Computer assisted waveform synthesis is an applicable alternative. Suddenly it is possible to compose, to transform, to control waveforms and waveform sequences. The relation of a waveform to the sound it generates is “One to One”. With respect to the description of a desired waveform no two sounds are equivalent. Waveforms, that are equivalent under the description of a sound, become distinguishable and musically different if the composer makes use of their transformational properties.

In the presence of an applicable alternative the two criteria of the current direction cease to be “necessary” criteria. No longer fundamental conditions for all music making, they still retain their usefulness as optional restrictions that a composer may choose for defining the musical universe for his next work.

If it can be shown that there exist significant musical ideas which require compositional thinking where not the sound but the waveform is the basic element and standard, then it can also be shown how the computer not only helps the composer to the fulfillment of up to now unfulfillable desires, but actually assists the composer in generating desires he never knew before.

## 2 General Notes

Sawdust is a program for creating waveforms.

To use Sawdust, you

1. define a set of *elements*; then
2. define a set of *links*, which are sequences of elements; then
3. define a set of transformations between the links; and then
4. play the defined transformations, links, or silences, in a user-specified sequence.

The `play` command lets you create a sequence of waveforms: you might have a `link`, followed by a `vary`, followed by a `silence`, followed by another `link`, and then a `turn`...etc.

Sawdust does *not* let you mix different sequences together: for that you use some other program (such as SoundEditor on an SGI, or CoolEdit on a PC, or RT on a NeXT or SGI).

Sawdust creates 16-bit signed, integer samples, at a sampling rate of 44100 samples per second.

### 2.1 Starting the Sawdust program

To start Sawdust, you type

```
sawdust
```

... and SAWDUST responds with

```
S A W D U S T
```

```
Compiled on 08/22/04 22:52:24
```

```
OS: Darwin 6.8 Machine: Power Macintosh
```

```
Using PortAudio 18.1
```

```
*
```

The asterisk is Sawdust's prompt: the program is waiting for a command.

To quit Sawdust, type `exit` or `quit`, or (on a PC) type `Control-D`.

### 2.2 Terminating a command

In general, you can terminate a command in Sawdust by typing a forward slash: `/`.

## 2.3 Objects

An *object* is a general term for an element, a link, a mingle, a merge, a turn, or a vary.

## 2.4 Help: display commands understood by Sawdust

```
* help                                     # type the 'help' command
commands:
  change      clear      delete      element
  elist       exact      exit        forget
  help        link       merge       mingle
  name        play       quit        range
  rate        read       replay      restore
  save        seed       show        status
  store       turn       vary        vquick
  write
*                                             # back to the Sawdust prompt
```

**change** (This command currently doesn't work.)  
**clear** deletes the default soundfile left from previous sessions(s)  
**delete** delete an object  
**element** define one element  
**elist** define a list of elements  
**exact** cause Vary to play the *exact* time requested  
**forget** delete *all* defined objects  
**help** list all commands understood by Sawdust  
**link** define a link  
**mingle** concatenate a list of links  
**merge** interlace the constituent elements of a list of links.  
**name** rename the default output soundfile  
**play** create (and hear) the samples for an object or sequence of objects.  
**range** set the amplitude range  
**read** read a text file of input data  
**replay** play a previously created sequence of samples  
**restore** restore objects that were saved with **save** (in binary)  
**save** save objects to a binary file  
**seed** set Vary's random number seed  
**show** show information about currently defined objects  
**status** show current state of Sawdust  
**store** create samples for a sequence of objects without playing them  
**turn** an algorithm for transforming an initial link to a final link  
**vary** another algorithm for transforming an initial link to a final link  
**vquick** Sets the value for *cutrate* used by Vary to calculate polynomials  
**write** write objects to a text file  
**quit** exit the Sawdust program

### 3 element, elist: define one or more elements

An element is the smallest specifiable portion of a waveform.

Each element has:

An identifier (a name),

An amplitude (between 0-4096, but this can be reset with the range command, see below), and

A duration (given in samples).

#### 3.1 The element command: define one element

element command lets you define a single element:

```
*      element                # command to define an element
name:      e6                # set the identifier
amplitude= 250               # its amplitude
samples=    25               # its duration in samples
```

#### 3.2 The elist command: define a list of elements

elist allows you to define a number of elements in a row.

```
*      elist
name prefix:      e          # prefix for all the elements
starting number = 0          # initial number
name:  e0
amplitude=      1000
samples=        10
name:  e1
amplitude=      2000
samples=        20
name:  e2
amplitude=      3000
samples=        30
name:  e3
amplitude=      4000
samples=        40
name:  e4
amplitude=      /           # end input with the '/' character
```

## 4 link: define a link

A link is a sequence of elements. Each link has:

- An identifier,
- A sequence of elements, and
- Statements: the number of times it is to be played.

### 4.1 The link command

The link command lets you specify sequences of elements, which can later be played, or used as the initial or final state for a turn or a vary.

```
* link
name:      l3          # identifier
0:         e0          # list of constituent elements
1:         e2
2:         e1
3:         e3
4:         # end input of list with blank line
statements= 441        # number of iterations when played
```

When played, a link creates a frequency that is:

$$frequency = \frac{sampling\_rate}{link\_duration}$$

The link's duration (which is the waveform's period) is the sum of its elements' durations:

$$link\_duration = \sum_{i=0}^n element_i$$

where  $element_i$  is the duration of the  $i$ 'th element.

A link's *statements* determines how long it will sound.

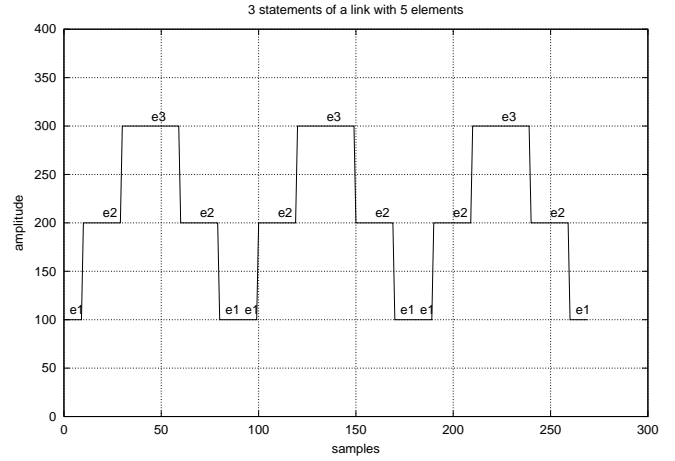
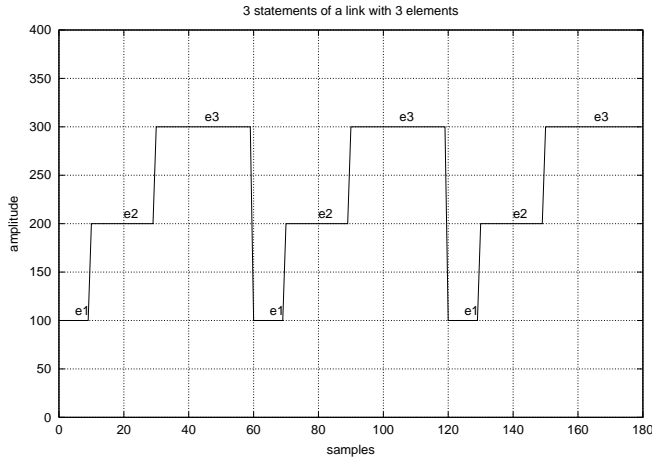
If you want the link to sound for  $N$  seconds, set *statements* to:

$$statements = \frac{N * sampling\_rate}{link\_duration}$$

Sawdust uses a sampling rate of 44100 samples per second.

## 5 Combining elements to form links

Here are two examples of links. Both links use the same set of elements:  $e1$ ,  $e2$ ,  $e3$ , but in different sequences: the link on the left is  $e1, e2, e3$ ; the link on the right is  $e1, e2, e3, e2, e1$ .



The duration of the first link is the sum of its elements' durations:  $e1 + e2 + e3$ . Its frequency is  $\text{sampling\_rate} / \text{link\_duration}$ .

The duration of the second link is  $e1 + e2 + e3 + e2 + e1$ . Its frequency will thus be lower than the first link, since its duration is longer.

## 6 Combining Links: mingle and merge

`mingle` and `merge` are two commands for combining links.

### 6.1 mingle: concatenate a sequence of links

*Explanation:*

```
*    mingle
name:      mg1                # name of mingle
0:         l1                 # constituent links
1:         l2
2:         l3
4:
statements= 1                  # number of mingle iterations
```

`mingle` concatenates a sequence of links:

$$l1 \rightarrow l2 \rightarrow l3 \rightarrow l1 \rightarrow l2 \rightarrow l3 \rightarrow l1 \rightarrow \dots (etc.)$$

Each link will be played *only* the number of statements given it. For example, if a mingle consists of links  $l1$ ,  $l2$ , and  $l3$ , and if  $l1$  was given 6 statements,  $l2$  4 statements and  $l3$  2 statements, when played the mingle will be:

$$l1 \rightarrow l2 \rightarrow l3 \rightarrow l1 \rightarrow l2 \rightarrow l3 \rightarrow l1 \rightarrow l2 \rightarrow l1 \rightarrow l2 \rightarrow l1 \rightarrow l1$$

$l1$  will be played 6 times,  $l2$  4 times, and  $l3$  2 times.

The mingle's number of statements determines the number of times the entire sequence will be played.

### 6.2 merge: interlace the constituent elements of a sequence of links

*Explanation:*

```
*    merge
name:      mr1                # name of merge
0:         l1                 # constituent links
1:         l2
statements= 1                  # number of merge iterations
```

`merge` takes a sequence of links, and interlaces their constituent elements.

For example, if  $l1$  consists of elements  $a1$   $a2$   $a3$ , (2 statements) and  $l2$  consists of elements  $b1$   $b2$  (2 statements), the resulting merge would be:

$$a1 \rightarrow b1 \rightarrow a2 \rightarrow b2 \rightarrow a3 \rightarrow b1 \rightarrow a1 \rightarrow b2 \rightarrow a2 \rightarrow a3$$

As in a mingle, each link is played only the number of statements given it.

### 6.3 Calculating the duration and frequency of merges and mingles

Both `mingle` and `merge` will create the same frequency, only if all their links contain the same number of elements:

$$f = \text{sampling\_rate} / \sum_{i=0}^N \text{link\_duration}_i$$

where  $\text{link\_duration}_i$  is the duration of the  $i$ 'th link.

If all their links do not contain the same number of elements, determining the frequency is more difficult. In general, the frequency created by a merge will be lower than that of a mingle, since its period length will be longer.

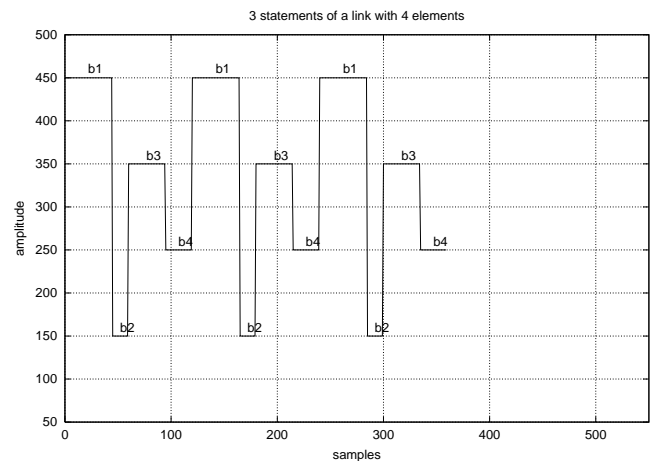
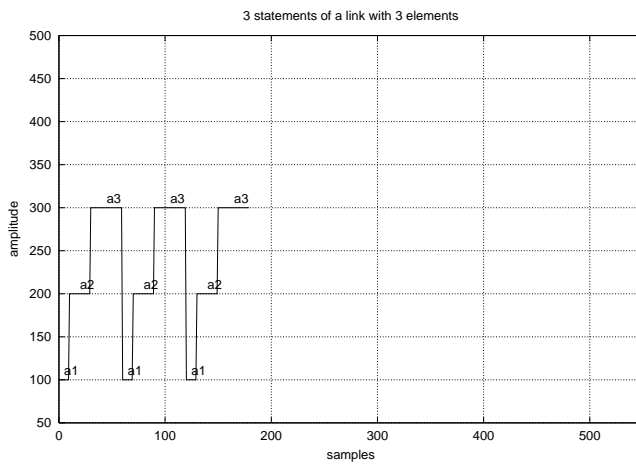
Mingles and merges calculate their durations in the same way:

$$duration = M\_statements \times \sum_{i=0}^N (link\_statements_i \times link\_duration_i)$$

where  $M\_statements$  is the number of statements given to a mingle or a merge.

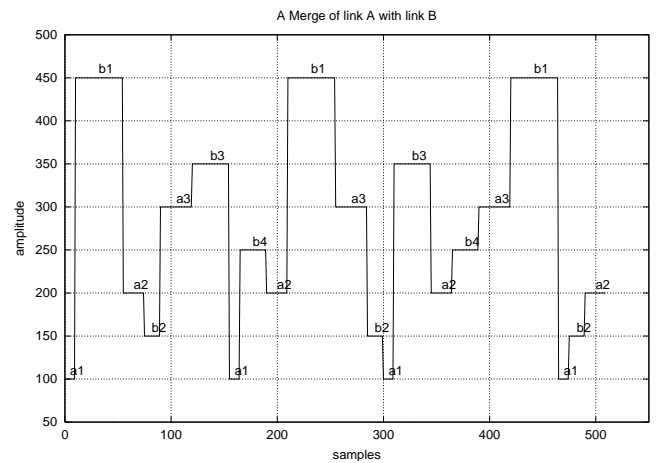
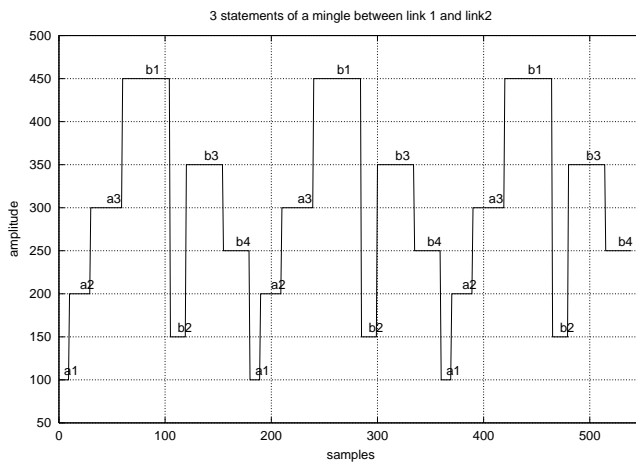
## 6.4 Examples of a merge and a mingle

Here are two links: link  $A$  has three elements, and link  $B$  has four elements. Each link is given 3 statements.



The elements of link  $A$  are  $a1$ ,  $a2$ ,  $a3$ . The elements of link  $B$  are  $b1$ ,  $b2$ ,  $b3$ ,  $b4$ .

Below on the left are three statements of a **mingle** of link  $A$  with link  $B$ . On the right is a **merge** of  $A$  with  $B$ .



A *mingle* concatenates its links; a *merge* interlaces the constituent elements of its links.

## 7 Transforming *linkIn* into *linkOut*: vary

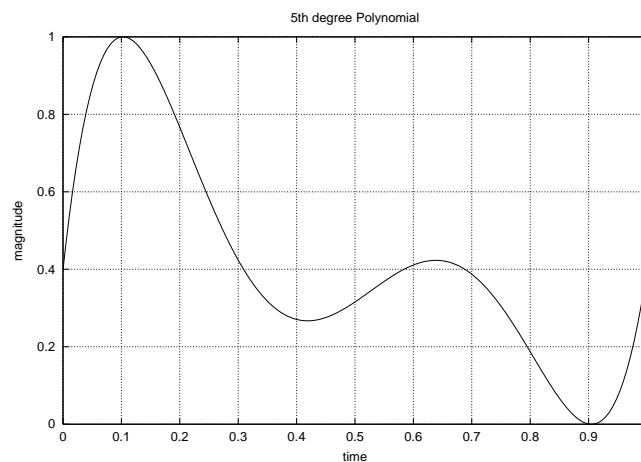
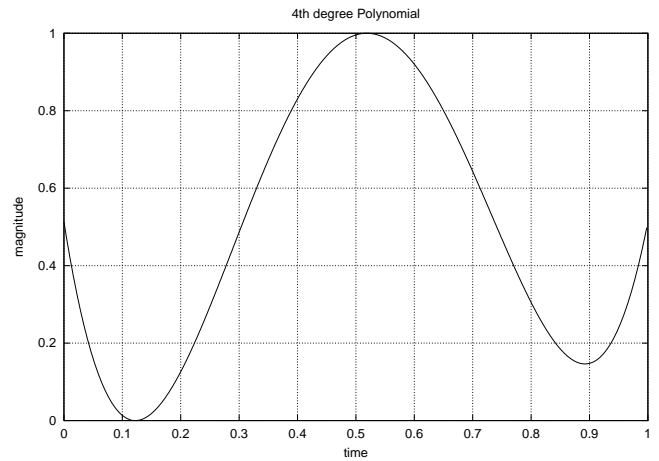
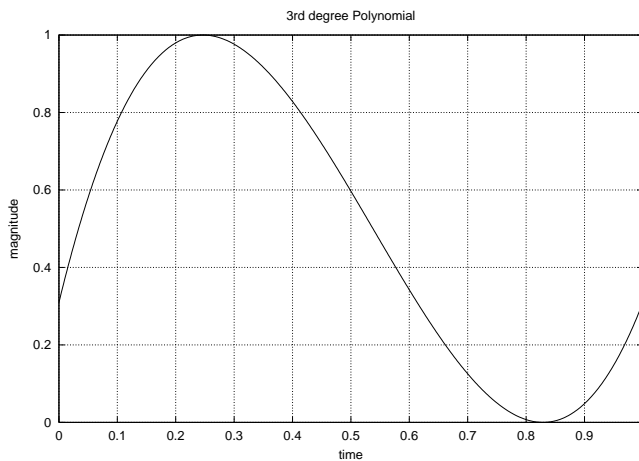
*vary* transforms an initial link into a final one. The two links do not have to have the same number of elements: if they are different, Sawdust silently adds “zero” elements to the smaller of two links. (A “zero” element is one whose amplitude and duration are set to zero.)

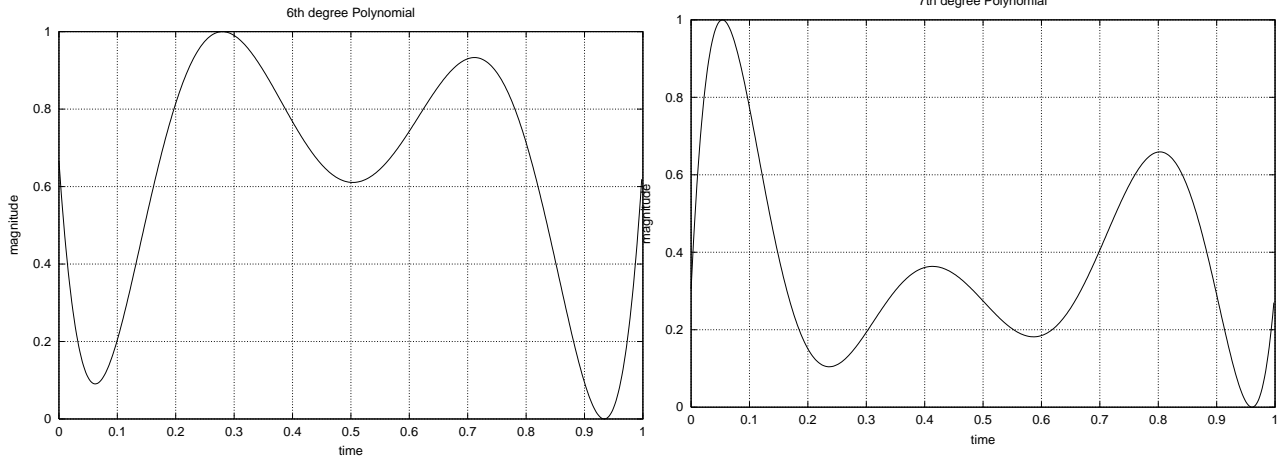
The following description is by Gary Grossman, Sawdust’s principal designer and implementer:

When a *vary* object is played, Sawdust selects polynomials that connect the amplitude value of each element of the first link with the amplitude value of the corresponding element of the last link. It does the same for the sample hold values of the corresponding elements of the two links. After first playing a single iteration of the first link, it computes and plays a new link at each iteration, according to the evaluation of the polynomial for each amplitude and sample hold value, for the number of iterations given as its repeat specification.

The polynomials can be of degrees 3 to 7.

Here are plots of the polynomials used by *vary*:





The above plots show the variety of paths which can be followed by the amplitude and sample values of each element.

## 7.1 The algorithm for vary

vary makes a transformation from the first link to the last link. These links do not have to have the same number of elements. If one link has fewer elements than the other, Sawdust silently adds zero-valued elements to the link with fewer elements.

Each element in the first link is connected to its corresponding element in the last link with a polynomial. At each step of the transformation, the magnitude of each elements' amplitude and duration are changed, as determined by its polynomial.

Thus, the amplitude and sample value for each element may follow different polynomials, and so their transformations may be independent of each other.

As the amplitudes raise and lower, following their polynomial, they will increase up to 4096, and lower down to zero.

The sample values, on the other hand, will grow and shrink only up to user-determined amounts (see *upper and lower sample bounds?* below).

## 7.2 The command vary

To define a vary, you must input

- A name for the vary.
- The initial and final link names
- Whether the last link will be played.
- Whether you want to specify the transformation duration in terms of *steps* or *time*.
- Whether the roots of the polynomial should be *frozen* for each statement of the vary.
- The polynomial degrees for the amplitude and sample values.
- The upper and lower bounds for the samples.
- The number of statements (iterations) of the vary.

*Example:*

```
*    vary
name:      v2
first link name:  l1
(samples = 60)
```

```

last link name:      12
(samples = 100)
play last link?      y
steps or time?       time
time=                441000
freeze?              y
4 elements in link
amplitude degrees:   3 4 5 6
sample degrees:      6 5 4 3
upper sample bounds = 10000
lower sample bounds = 0
statements=          1

```

*Explanation:*

```

*    vary
name:      v2                # name of vary (identifier)

first link name:  l1        # name of the first link
(samples = 60)     # Sawdust reports first link duration
last link name:   12        # name of the last link
(samples = 100)    # Sawdust reports last link duration

play last link?   y        #

steps or time?    time     # calculate the transition with
                           # 'number of steps' or 'samples'?

time=             441000    # duration of transformation in samples

freeze?           y        # at each 'statement', either use
                           # the same zero crossings
                           # (freeze=y), or 'jiggle' them randomly
                           # a little bit.

4 elements in link
amplitude degrees: 3 4 5 6  # polynomial degrees for each amplitude
sample degrees:    6 5 4 3  # polynomial degrees for each sample

upper sample bounds = 10000 # limits
lower sample bounds = 0

statements=        1        # how many times should it be played?

```

### 7.3 steps or time?

The question *steps or time?* asks if the vary's duration should be calculated in terms of *steps* (the number of steps you wish the transformation to take) or *time* (the number of samples you wish the transformation to take).

The calculation of *steps* or *time* is made using the following functions:

$$time = \frac{steps}{2} (first\_link\_samples + last\_link\_samples)$$

$$steps = \frac{2 \times time}{first\_link\_samples + last\_link\_samples}$$

## 7.4 freeze?

The question *freeze?* asks whether the zero-crossings of the polynomials should remain the same from statement to statement, or whether they should change.

If *freeze?* is answered “y”, then each statement of the vary will be identical with the others.

If *freeze?* is answered “n”, then a small, random value will be added to each zero-crossing, and thus each statement of the vary will be different from the others.

## 7.5 amplitude and sample degrees

This assigns a polynomial to the elements in the links. If the first and last links do not have the same number of elements, “zeroed” elements are added to the one with fewer elements.

```
4 elements in link
amplitude degrees:  3 4 5 6    # polynomial degrees for each amplitude
sample degrees:    6 3        # polynomial degrees for each sample
```

In the above example, Sawdust reports there are 4 elements in the link.

It then asks for the polynomials to be assigned to the amplitudes. In the response above, the first element’s amplitude is given a 3rd degree polynomial, the second element’s amplitude is given a 4th degree polynomial, the third a 5th degree, and the fourth a 6th degree polynomial.

Sawdust then asks for the sample assignments. In the above example, the first element’s sample is given a 6th degree polynomial, and the second, third, and fourth elements are all given a 3rd degree polynomial.

If you type in fewer degree values than there are elements, Sawdust will take the last degree value, and apply it to the remaining elements. (This applies to the amplitudes as well.)

## 7.6 upper and lower sample bounds

The *upper and lower sample bounds* assigns limits to how long and short a sample value can become.

As a rule of thumb, 100 and 10 are good upper and lower bounds.

The lower bound cannot be less than 0.

For elucidation, if the upper bound is set at 1000, that means that an element could have the frequency  $sampling\_rate / 1000$  which at the sampling rate of 44100 samples per second, means a frequency of 44 Hz., which is pretty low. If you make the upper bound greater than that, the frequency will be even lower.

## 7.7 statements

*statements* is the number of times the vary will be repeated.

## 8 Transforming *linkIn* into *linkOut*: turn

**turn** transforms an initial link into a final one.

*Example:*

```
*      turn
name:      t2
first link name:  l1
      3 elem;  60 samples
position?    1
last link name:  l2
      4 elem;  100 samples
position?    1
rotate?     n
play last link?  y
Amplitude range 1000 to 4000
use default values ?  y
Linkstates = 41, linksum 3280
turnsum?    44100
statements=   10
Wavy ? y
```

*Explanation:*

*      turn	
name:      t2	# name of turn (identifier)
first link name:  l1	# initial link
3 elem;  60 samples	# duration of link in samples
position?    1	# which element should the
	# initial link start with?
last link name:  l2	# final link
4 elem;  100 samples	# duration of link in samples
position?    1	# which element should the final
	# link start with?
rotate?     n	# if 'no' then
	# (e1 e2 e3) (e1 e2 e3) (etc)
	# else
	# (e1 e2 e3) (e2 e3 e1) (e3 e1 e2) (etc)
play last link?  y	#
Amplitude range 1000 to 4000	
use default values ?  y	#
Linkstates = 41, linksum 3280	#
turnsum?    44100	# duration of turn in samples
statements=   10	# how many times should it be played?
Wavy ? y	# if Wavy is 'y', then:
	#  l1 -> l2 -> l1 -> l2 (etc.)

```
# if Wavy is 'n', then:
#   11 -> 12   11 -> 12 (etc.)
```

## 8.1 The problem addressed by *turn*

The algorithm *turn* is a solution to the problem: how to maintain a constant rate of change of pitch during a transformation?

This problem is a result of a premise of SAWDUST, and applies to both *vary* and *turn*: An initial link is linearly transformed into a final link.

For example, if there are two links, with a sum of 100 samples and 50 samples, and a transformation is made from the first to the second, the resulting sound would go up in pitch by one octave. (The first link would create a frequency of 441 Hz, and the second 882 Hz, at a sampling rate of 44100.)

However, the resulting in a sound would appear to be ascending at a faster and faster rate.

The opposite is true when moving from the second to the first link: as the pitch went down, it would appear to be getting slower and slower.

Brün invented a remarkable solution for this problem, that remained consistent with the linear change premises of SAWDUST.

## 8.2 The *turn* algorithm

In a *turn*, *linkIn* is transformed into *linkOut*.

Both *linkIn* and *linkOut* have constituent elements (but not necessarily the same number of elements). Each element has an amplitude (0–4096) and a duration (in samples).

The duration of each element in *linkIn* is transformed (step-wise, by integers) into the duration of the corresponding element in *linkOut*.

The amplitude of each element in *linkIn* follows the path of a triangle wave, as it is transformed into the corresponding element's amplitude in *linkOut*.

Thus, the transformations of the durations and the amplitudes are mutually independent.

If *linkIn* and *linkOut* do not have the same number of elements, SAWDUST will add  $n$  “empty” elements to the smaller of the two, where  $n$  is the difference between the number of elements in *linkIn* and *linkOut*. (An “empty” element has zero amplitude and zero duration.)

## 8.3 *turn* amplitudes

In a *turn*, each elements' amplitude in *linkIn* follows the path of a triangle wave, as it is transformed into its corresponding elements' amplitude in *linkOut*.

The amplitudes' paths are determined by the following:

```
if ( amp >= midpoint ) then
  1. amp descends until it reaches minimum.
  2. ascends until it reaches maximum.
  3. descends to its destination value
else if ( amp < midpoint ) then
  1. amp ascends until it reaches maximum.
  2. descends until it reaches minimum.
  3. ascends to its destination value.
```

*midpoint* is equal to 4096/2. By default, *minimum* and *maximum* are set to the smallest and largest amplitudes in *linkIn* and *linkOut*.

If  $\text{amp} \geq \text{midpoint}$ , the amplitude increment is:

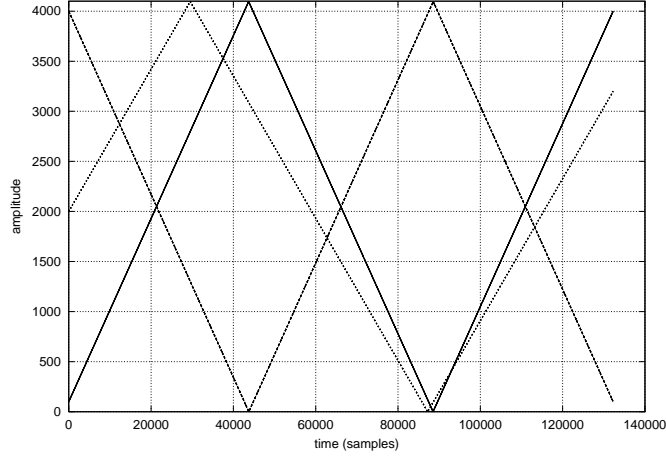
$$\text{inc}_i = \frac{2\text{max} - 2\text{min} + \text{linkIn}_i - \text{linkOut}_i}{\text{turnsum}}$$

if  $\text{amp} < \text{midpoint}$ , the amplitude increment is:

$$\text{inc}_i = \frac{2\text{max} - 2\text{min} + \text{linkOut}_i - \text{linkIn}_i}{\text{turnsum}}$$

In both cases,  $\text{turnsum}$  is the duration of the turn in samples.

Here is a plot of the amplitude paths for three elements, where  $\text{linkIn}$  has amplitudes 100, 4000, 2000, and  $\text{linkOut}$  has amplitudes 4000, 100, 3200.



In the above plot, the element that starts with an amplitude of 2000, rises to its maximum (4096), then descends to its minimum (0), and then rises to its final value (3200).

In contrast the element whose initial value is 4000, first descends to its minimum value (0), then rises to its maximum (4096), and then descends to its final value (100).

## 8.4 turn samples

The transformation of durations in a *turn* follows a step-wise, integer path, from the initial to the final link. However, each state in the transformation is repeated, with the shorter states (the higher pitched ones) repeated more often than the longer states (the lower pitched ones).

The following function (written by Elizabeth Mitro) determines the number of steps in the transformation between the initial and the final link:

$$\text{linkStates} = \sum_{i=0}^N |\text{initial\_link}_i - \text{final\_link}_i|$$

where

$\text{link}_i$  is the duration (in samples) of element  $i$  in  $\text{link}$

$N$  is the maximum number of elements in the *initial link* and the *final link*,

$\text{linkStates}$  is thus the sum of the difference in durations of the initial and final elements: it is the minimum number of steps necessary for the transformation.

$\text{linkSum}$  is the sum of durations of the  $\text{linkStates}$ : it is the minimum duration necessary for the transformation.

Once the  $\text{linkStates}$  in the transformation is known, the duration of each step in the transformation can be determined:

$$\text{step\_duration} = \frac{\text{turnsum}}{\text{linkStates}}$$

where

*turnsum* is the duration of the turn in samples, and has been entered by the user

Once the *step\_duration* is known, the number of repetitions of the link at that step is calculated:

$$\text{number\_of\_repeats} = \frac{\text{step\_duration}}{\text{link\_duration}}$$

The above calculation ensures that the smaller the link, the greater the number of repetitions.

Thus, in a transformation from a high-pitched to a low-pitched link, the high-pitched links will be repeated more times than the low-pitched links, resulting in an “even” descent.

## 8.5 The command **turn**

To define a turn, you must input

- A name for the turn.
- The first and last link names.
- The element starting position of the first and last links.
- Whether the turn should be “rotated”.
- Whether the last link should be played.
- Whether the default values should be used for the amplitudes.
- The duration of the turn in samples (the “turnsum”).
- The number of statements (iterations) of the turn.
- Whether sequential statements are to be “wavy” or not.

## 8.6 **name, first link name, and last link name**

*name* asks for an identifier for this turn.

*first link name* asks for the name of the link to be assigned as *linkIn*.

*last link name* asks for the name of the link to be assigned as *linkOut*.

## 8.7 **position? and rotate**

*rotate?* and *position?* need each other.

If *rotate?* is answered “y”, then at every state of the transformation, the sequence of elements will cycle.

For example, if there are three elements:

$$e1\ e2\ e3 \mid e2\ e3\ e1 \mid e3\ e1\ e2 \mid e1\ e2\ e3\ (etc.)$$

were every group of three above is a state of the transformation.

The number given in response to *position?* determines the starting element for the first state in the rotation, and the starting element for the final state in the rotation.

In effect, *rotate* increases the period length of the transformation by the number of elements in the transformation. So, if there are three elements in the transformation, the period of a rotated transformation will be three times the length of a non-rotated transformation.

## 8.8 **Amplitude range; use default values?**

By default, *turn* uses the greatest and smallest amplitudes in the first and last links for the minimum and maximum values to which all the amplitudes will grow and shrink.

If you wish, you can override these default values, and enter other values.

## 8.9 turnsum?

turnsum? asks for the duration of the turn in samples.

At a sampling rate of 44100 samples per second, if you wanted a turn to last for 10 seconds, you would type in 441000.

## 8.10 statements and Wavy?

“Statements” is the number of times the turn will be played.

If “statements” is greater than 1, then the question “Wavy?” will be asked.

A “wavy” turn alternates transformations between linkIn to linkOut, and linkOut to linkIn:

$$linkIn \rightarrow linkOut \rightarrow linkIn \rightarrow linkOut \dots$$

The first statement makes a transformation from linkIn to linkOut, the second statement makes a transformation from linkOut to linkIn, the third statement is the same as the first, the fourth is the same as the second, etc.

A non-“wavy” turn has all the statements the same: the transformations are only from linkIn to linkOut.

## 9 Displaying and Removing Objects: show, status, delete, forget

show Show information about defined objects  
status Show current state of SAWDUST (maximum number of objects, current range, etc.)  
delete Delete one object  
forget Delete all objects

### 9.1 The show command

The show command displays information about current defined objects:

```
*      show                                # type the command 'show'
Show What? : all                          # show everything
defined objects:
  e0          e1          e2          e3
  11          12          t1          v1
  e4          e5          e6          e7
  13          14          15
  8 elements, 5 links, 0 merges, 0 mingles, 1 vary's, 1
turns
15 total
Show What? : 13                          # what else to show? a link
13 Link
Expand sub-objects? (y/n) y              # now show the link's parts
  e4 Element 1500a, 55s
  e5 Element 2500a, 45s
  e6 Element 3500a, 35s
  e7 Element 500a, 25s
  275 statements 160 samples
Show What? :                             # a blank line ends the 'show'

*                                          # back to Sawdust's prompt.
```

### 9.2 The status command

The status command tells you the current state of Sawdust:

```
*      status                                # type the 'status' command
Rate = 44100
Auto-seed : 0
Cutrate   : 500.000000
Exact-vary : OFF
Limits: Amp Range  Objects  Ele/Link  Ele/Vary
      Max:    4095    280      100      24
      Min:      0
*

```

The variables Rate and Limits, are general and apply to all the commands in Sawdust. They tell you:

Rate Sawdust's sampling rate (44100 samples per second)

Limits The current capacities of Sawdust:  
 The amplitude range is 0 to 4095 (can be reset by `range`).  
 The maximum number of objects that Sawdust can handle in one session: 280.  
 The maximum number of elements that can be in a link: 100.  
 The maximum number of elements that can be in a vary: 24.

The following variables are specific to vary, and do not affect the other commands:

Auto-seed The current random number seed. This can be set by the command `seed`

Cutrate Specifies the number of points along the polynomial to be sampled when looking for its minimum and maximum values. Cutrate can be set by the command `vquick`.

Exact-vary Whether vary will play the *exact* time requested or not.  
 It can be set with the `exact` command.

### 9.3 The delete command

The `delete` command deletes one object from Sawdust's memory. Sawdust won't let you delete an object if it is currently being used by another object, i.e., you can't delete an element if it's being used by a link.

```
*      delete                                # type the 'delete' command
name:   e1                                # delete 'e1'
e1 is used by other objects; cannot delete. # Sawdust won't let you
name:   e2                                # delete 'e2'
e2 is used by other objects; cannot delete. # Sawdust won't let you
name:   e8                                # e8 can be deleted safely
name:   f1                                # but f1 doesn't exist.
not defined: f1.
name:   /                                # end the delete command

*                                           # back to Sawdust's prompt
```

### 9.4 The forget command

The `forget` command deletes ALL the objects in Sawdust's memory. It will ask for confirmation before deleting everything.

```
*      forget                                # type the 'forget' command
This command deletes all currently defined objects.
delete all objects?    y                    # ask for confirmation

*                                           # back to Sawdust's prompt
```

## 10 Data Input/Output: save, restore, write, read

These commands allow you to save the work you've started, or retrieve previously saved work.

```
save    Save objects to a binary file
restore Restore objects from a binary file
write   Write objects to a text file
read    Read objects saved in a text file
```

save and restore are used as a pair; likewise with write and read.

### 10.1 The save and restore commands

The save command saves all the objects you've defined in your current Sawdust session. The restore command reloads the objects you saved using the save command.

The data are saved to a binary file, which is compact, and can be read quickly by Sawdust. However, this file cannot be edited by a text editor: you cannot use vi (or any other text editor) to edit this file.

```
*      save                                # type the 'save' command
save file name:mydata                     # Sawdust asks for a filename

*                                           # return to the Sawdust prompt
*      save                                # another 'save' command
save file name:mydata
save file already exists; do you wish to overwrite it?y

*
```

As show in the example above, Sawdust will ask for confirmation before overwriting an existing save file.

```
*      restore                             # type the 'restore' command
save file name:data                       # Sawdust asks for a filename
data is not a save file                    # file 'data' was not found

*      restore                             # try restore again.
save file name:mydata

*                                           # the data has been restored
```

### 10.2 The write and read commands

The write and read commands function in the same way as the save and restore commands, except that the objects are written to a text file, that can be read and edited by a text editor.

## 11 Creating and Playing samples: play, replay, store

play Create the samples for an object or sequence of objects, and let the user hear the sequence  
store Create the samples for an object or sequence of objects, but do not let the user hear them.  
replay Play the current sequence of objects

### 11.1 The play command

play is the command you use when you want to hear the objects you've created.

The play command does three things:

1. creates the samples for one or more objects;
2. appends those samples to the output soundfile; and
3. lets the user hear the current sequence of objects

In the example below, the play command is invoked to create a sequence of three objects: a **link** followed by a **vary** followed by another **link**. All three objects are assumed to have been defined.

Press RETURN to end the list of objects you want to hear.

```
*    play

:link1                                # 1st object: a link

44100 samples, 1.0000 seconds, 0.0167 minutes # link1's duration

:vary1                                # 2nd object: a vary
.....
440958 samples, 9.9990 seconds, 0.1667 minutes # vary1's duration

:link2                                # 3rd object: a link
.
44100 samples, 1.0000 seconds, 0.0167 minutes # link2's duration

:                                     # type RETURN
total:    529158 samples, 11.9990 seconds, 0.2000 minutes # summed duration
type CR when ready:                    # type another RETURN
                                         # Sawdust plays the
                                         # sequence
*                                     # back to Sawdust prompt
```

At this point, were you to type play again and create other objects, the 2nd group would be appended to the end of the first group, and play would play back both the first and second groups, in order.

### 11.2 The store command

The store command allows you to append objects to the current sequence without hearing them: it's like play without its last step.

```
*    store                            # invoke the command

:link1                                # 1st object

44100 samples, 1.0000 seconds, 0.0167 minutes
```

```

:vary1                                # 2nd object
.....
440958 samples, 9.9990 seconds, 0.1667 minutes

:link2                                # 3rd object
.
44100 samples, 1.0000 seconds, 0.0167 minutes

:                                     # Type a RETURN
total:    529158 samples, 11.9990 seconds, 0.2000 minutes

*                                     # Back to Sawdust's prompt

```

As with the play command, if you were to type the store command again, and create a second group of objects, the second group would be appended to the end of the first group.

### 11.3 The replay command

The replay command replays the current sequence of objects.

```

*      replay
type CR when ready:                # Type a RETURN
                                     # Sawdust plays the sequence
*                                     # Back to the prompt

```

### 11.4 Note for the play and store commands

Both commands append samples to current output soundfile. This means that if you have left a soundfile from a previous session with Sawdust, your current samples will be appended to it, and when you play the samples, you will hear both the samples left from the previous session, as well as those from the current session.

Use the name or clear commands to delete or rename the output soundfile.

### 11.5 Generating silence with the play and store commands

Both play and store will generate silence, if you input a duration rather than the name of an object.

The duration is given in samples, so if you wanted a silence of 2 seconds, you would type in the number 88200, since Sawdust's sampling rate is 44100 samples per second.

## 12 Soundfile Manipulation: `name`, `clear`

`name`    Rename the default output soundfile

`clear`   Delete soundfile left from previous sessions(s)

When Sawdust creates and stores samples, it stores them to a soundfile on your computer's hard disk. Under 32-bit Windows systems and Linux systems ("little-endian" machines) the soundfile is called `sawdust.wav`, under Mac OS X (a "big-endian" machine) it is called `sawdust.aiff`.

`clear` deletes this soundfile. `name` allows you to rename it.

### 13 Setting Variables: *exact*, *seed*, *vquick*, *range*

- `exact`    Make vary play exact time requested
  - `seed`    Set the random number seed for vary (default=0)
  - `vquick`    Set the value for *cutrate*: the number of points along the polynomial to be sampled when looking for the polynomial's maximum value
  - `range`    Set amplitude range: either the default: (0,4095), or (-4096,4095)
- In general, you should never have to set these.

## 14 Gary Grossman: Instruments, Cybernetics, and Computer Music (1987)

### Abstract

Using *cybernetics* as the study of control mechanisms, this paper sketches an analysis of mechanical and computer-based tools for producing music, in terms of their control mechanisms and the relations that they imply between musicians and their tools. After defining control mechanisms and their effect on the use and evolution of tools in general, the paper applies these concepts to mechanical musical instruments and then to computer-based music systems in an attempt to understand the current state in the evolution of computer-based tools for music production.<sup>1</sup>

### Introduction

*Cybernetics*, taken in one of its simplest senses, is the study of control mechanisms and their effect on the relations between people and their tools. Using that definition, I have attempted to sketch an analysis of some tools for producing music, in terms of their control mechanisms and the relations that they imply between musicians, instruments, and computer-based music systems.

### Tools

#### *The Parts of a Tool*

For the purpose of this analysis, a tool is composed of two parts: an *effective mechanism*, which directly accomplishes the purpose for which the tool is employed, and a *control mechanism*, which mediates between the person using the tool and the effective mechanism.

#### *Control Mechanism as Analogue*

In fulfilling its function, the control mechanism acts as an analogue of the effective mechanism. For example, the effective mechanism of a knife is its edge, and the control mechanism is its handle. Whatever forces are applied to the handle are analogously applied to the edge.

The analogy between the effective and control mechanisms imposes a distance between them, and also between the user of the tool and the effective mechanism. In many cases, this distance makes the tool usable: we cannot grasp the edge of the knife to use it; we are neither large enough nor strong enough to manipulate the parts of a construction crane directly; we are not fast enough to feed data to the CPU of a computer.

#### *Limits of a Tool*

While the effective mechanism of a tool imposes limits on its usability through its physical characteristics, such

as sharpness, size, or speed, what is far more important for the control mechanism is the extent to which it is a complete analogue of the effective mechanism. A vehicle with wings and a jet engine, but with only a steering wheel, a brake, and an accelerator, would not be an airplane; it would be an automobile, and a clumsy one. Nothing can be accomplished with the effective mechanism that is not provided for in the control mechanism as its analogue.

#### *Evolution and Tools*

Every tool is continuously involved with the society in which it is used in a mutual process of shaping and evolution. The evolution of each tool is steered by a socially defined model that defines its use. In order for a new tool to evolve, a new model must first exist in the mind of at least one person who wants to use that new tool. And the use of a tool according to its model creates a new social environment that may create desires for a change in how the tool is used, and thus in the tool itself.

The examples which are most characteristic of this process are, unfortunately for us all, weapons. But, fortunately for the author, who might otherwise have encountered insuperable difficulty in fitting his ideas into the allotted space, the case is so manifest as to require no explication here.

### Mechanical Musical Instruments as Tools

#### *Control Mechanisms of Mechanical Instruments*

Mechanical musical instruments: strings, woodwinds, brass, and percussion, can be analyzed in the same terms as other tools. For instance, the effective mechanism of a trombone is a column of air, confined in a brass tube, that is excited by the vibrating lips of the player. The control mechanism for the pitch produced by the instrument has two parts: (1) the lips of the player, acted upon

<sup>1</sup>This paper was presented at the International Computer Music Conference, in Urbana, Illinois, August, 1987. Gary Grossman was the primary architect and programmer for the original version of Sawdust, and worked on the project from approximately 1972 until 1980. He had been a music composition student at the University of Illinois since the early 1960s, and had continued his work as a composer, performer, and programmer.

by the tension of the surrounding musculature and by the air pressure applied by the player's respiratory system, excites the tube to vibrating, primarily at the frequency of one of the harmonic partials of the tube's fundamental frequency; (2) the fundamental is determined by the length of the tube, and so the length of the tube must be changed to select different fundamentals.

There are two methods of changing the length of the tube: a slide and a set of valves; and the two methods define two different instruments. That they are two different instruments can be demonstrated by imagining a player attempting to play the scales in the opening scene of Verdi's *Othello* on a slide trombone, or the glissandi in Bartok's *Concerto for Orchestra* on a valve trombone.

### *Evolution and Instruments*

Musical instruments, like all tools, are involved in a continual process of evolution with the needs and expectations of the society in which they are used. While the evolution of some instruments, such as the strings, has been primarily in terms of the technological improvements of their effective mechanisms, the evolution of many instruments, such as the winds, has been primarily in terms of the conceptual and technological improvements of their control mechanisms.

To take the clarinet as an example: from its historically hazy invention by J. C. Denner of Nuremberg around 1700, the evolution of the clarinet has been primarily through placing and adding holes and keys and levers to permit more correct intonation and to facilitate more rapid and sure passing from any pitch to any other. (Space does not permit the discussion of the two competing control systems that have evolved for the clarinet. Suffice it to say that they are, to everyone but a clarinetist, essentially equivalent.)

The influence between society and the clarinet has been both mutual and circular: with improvements in the clarinet's ability to play more or less in tune in all keys and to play chromatically, composers were encouraged to use it in new ways; the clarinet parts of *Til Eulenspiegel* could not have been conceived for a clarinet of the early 19th century. Conversely, *Til* as a composition is inconceivable without clarinets. And the existence of works like *Til* encouraged, via performers' demands and suggestions to instrument makers, further improvements in the clarinet's control system through the 20th century to today.

As with all instruments, the development of the control systems of the woodwind instruments has been in accord with a model that remained virtually unchanged until the middle of the 20th century. That model considered

a woodwind as producing one pitch at a time by exciting its air column with a vibrating reed or by blowing over an open hole. The timbre was to be recognizably uniform throughout the instrument's compass. Any other kind of sound produced by the instrument, including multiphonics or the clicking of the keywork, was considered a regrettable by-product of a necessarily imperfect physical implementation.

Compositions for woodwinds were really compositions for this model, and the technical judgment of a performance was predicated on the ability of the performer to play in conformance with the model. In that sense, the performer performed music using the instrument and, at the same time, performed the social definition of the instrument.

Beginning in the late 1950s, composers and performers recognized that musical instruments were capable of producing sounds that did not lie within the previously accepted social models of the instruments. They began to compose and perform using any sound that a given instrument could be reliably demonstrated to produce. This represented an attempt to explore, not merely to master, the instruments. And these explorations have not only produced new social models of each instrument, but a new meaning of "musical instrument."

Depending on the instrument, these explorations met with varying success. This was to some extent a function, for each instrument, of the extent to which its control mechanism, which had evolved under a more restricted model, accidentally permitted the exploitation of these aspects of its effective mechanism. For example, the number of different multiphonics that can be produced on a woodwind instrument is a function of the independence with which the holes in its tube can be closed. In the case of the flute, nearly all of the holes can be independently closed. In contrast, the mechanism of the saxophone precludes opening certain holes while others are closed, to facilitate certain common passages and trills in the keys in which it was normally played.

In the case of key clicks, a characteristic of the implementation of the control mechanism, viewed as a defect under the previous model, is viewed under the new model as a useful function of the effective mechanism of the instrument. The extent to which the bassoon produces the loudest and most varied key clicks, once seen as a necessary evil by builders, performers, composers, and listeners, can now be welcomed as a virtue by all. This rivals the transformation wrought by the marketing experts who turned waxed paper sandwich bags into "microwave convenience bags."

Assuming that further evolution of mechanical musical instruments will not be precluded economically by

the advent of electronic instruments, what form will this evolution take in response to the new social definitions of the instruments? While it is unlikely that woodwind instruments will be designed to exploit key clicks, is it unreasonable to expect designers to take facility in producing multiphonics into account? Is there a new Adolphe Sax engaged even now in constructing a “Multiphone?”

## Computer-Based Music Systems and Tools

In analyzing mechanical instruments, it was reasonably easy to identify and distinguish their effective and control mechanisms. But even if we confine our study of computer-based music systems to delayed performance systems that employ a general-purpose computer to compute waveform samples for digital-to-analog conversion, our analysis encounters inherent difficulties.

The invention of the computer was a principal impetus for the development of the study of cybernetics, because it is a tool that can be used to recursively redefine its own control mechanism (the software). Each layer of control mechanism is built on, and in terms of, the layer below. More important, each layer acts as part of the effective mechanism for the layers above, and as part of the control mechanism for the layers below. And all but the lowest of these layers (the hardware) can be created and modified by the same computer on which they are to operate.

These are the properties that have made the computer, which only a few decades ago was usable by only a handful of mystical practitioners to solve a restricted set of problems, the archetypical tool of our time. But because of these properties, the analysis of a computer system into control and effective mechanisms is by no means obvious.

The key to the analysis is not *what* is being analyzed, but *where* it lies in the structure and *when* we are looking at the process in which it takes part. We can then appeal to our definitions: when a part of a computer system is engaged in directly accomplishing the purpose for which the computer is employed, it is part of the effective mechanism. And when a part of a computer system is engaged in mediating between the person using the system and the effective mechanism, it is part of the control mechanism.

What, then, are the effective and control mechanisms of a computer-based music system? The digital-to-analog converter and the analog electronics clearly are part of the effective mechanism. So too are the software and hardware for storage of the samples and for deliv-

ering them to the converter at regular intervals. And the software embodying the algorithms for computing the samples must also be considered part of the effective mechanism. Everything else, including the software and the hardware whereby the user defines, selects, and directs the sample computation algorithms, must be classified as the control mechanism.

## Music *N* Systems

The pioneering and still dominant family of delayed-performance computer music systems is of course the “Music *N*” family, initiated in the early 1960s by M. V. Mathews of Bell Telephone Laboratories and epitomized by Music V.<sup>2</sup> The Music *N* family of systems has been described and has been subjected to detailed critique elsewhere by Loy and Abbott;<sup>3</sup> the description of the score and of the control mechanism given below is a gross simplification intended to be the minimum necessary to support the discussion.

### *The Music N Score*

The input to a Music *N* system consists of a *score*: an ordered set of statements that define *instruments*, generate *functions*, or specify *notes*.

An instrument consists of interconnected *unit generators* including oscillators, envelope generators, adders, multipliers, random generators, and filters, with the output of one becoming the input of another. New instrument definitions can be introduced at any time during the generation of a piece.

A Music *N* function definition specifies the shape of a waveform to be produced by the oscillators that use it beginning at a given time.

A Music *N* note specifies the invocation of an instrument, with a given setting of the inputs for the unit generators that compose it, at a specified time for a specified duration.

According to the distinction established in section 4, the Music *N* control mechanism comprises the software for reading the score, encoding the definitions and specifications, sorting the defining and specifying events in order of increasing start time, and interpreting the sorted list in time order to set up and invoke the computation algorithms at the specified times for the specified durations.

<sup>2</sup>Mathews, M. V., *The Technology of Computer Music*, The MIT Press, Cambridge, MA, 1969.

<sup>3</sup>Loy, G., and Abbott, C., “Programming Languages for Computer Music Synthesis, Performance, and Composition”, ACM Computing Surveys, vol. 17, no. 2, June 1985, pp. 244–250.

## Evolution and Music N

From the composer's point of view, the model of Music *N* consists of sets of interconnected modules that can come into being at specified times. These sets of modules include oscillators that can change their wave shapes at specified times. Each set can be turned on at specified times for specified durations with specified settings of the modules' controls.

This model has served to make the Music *N* systems predominant among delayed-performance computer music software. How did this model evolve?

Loy and Abbott<sup>4</sup> have cited a number of factors that have contributed to the success of the Music *N* systems, including the simplicity and power of its interface, and the efficiency of implementation that is implied by the interface model. The evolution of tools suggests additional factors that may have contributed to the widespread initial acceptance of these systems: Music *N* met an immediate need with a tool that conformed to a familiar model.

In the electronic music studios of the early to mid 1960s, voltage-controlled modules were in the process of becoming the standard. They were particularly successful because of the inter-compatibility of their control systems, which permitted the output voltage of any module to be used as the control voltage for any other module. For instance, a composer could produce an FM signal by simply connecting the output of one oscillator to the frequency control of another, or could produce an AM signal by connecting the output of an oscillator to the amplitude control of an amplifier.

There were three practical difficulties that composers encountered in using a studio build of these modules: (1) there were never enough modules or patch cords to set up the most complex configurations that the composer required; (2) it was tedious to change the setup of interconnections and knob settings from one configuration to another; and (3) the stability and accuracy of the voltages output by the modules under less than ideal environmental conditions was sometimes a source of frustration.

Whether or not its designers intended it to do so, Music *N* solved every one of these problems by transporting the voltage-controlled studio model from the analog domain to the digital-to-analog domain. And the very composers most likely to use a computer-based music system were those who were already familiar with the voltage-controlled electronic music studio.

But the Music *N* family has been predominant for well over two decades, without undergoing substantial change other than in its implementation language and its

hardware base. Why has there been so little change in these systems, when computer software systems in general have evolved so radically during the same period?

The factors cited by Loy and Abbot must contribute to this state of affairs; Music *N* is an elegant, powerful system that is easy to learn. Certainly these are necessary conditions for the evolution of a system to stop. But they hardly seem sufficient, given the rapid evolution and differentiation of other kinds of software systems.

Could it be that the overwhelming prevalence of Music *N* is in some way responsible? Music *N* is taught in virtually every university course on computer music. Hardly a book is published on computer music that does not contain a substantial section that treats with it. Just as "IBM machines" were once the socially accepted model for all computers, are Music *N* systems the socially accepted model for all computer-based music systems?

## Sawdust

Sawdust<sup>5</sup> was conceived by Herbert Brün, designed and implemented by the author, and enhanced by Jody Kravitz. The original version ran under the Unix time sharing system on a PDP-11/50 at the Center for Advanced Computation of the University of Illinois, using a D/A interface designed and built by Jody Kravitz.

The composer using Sawdust works in terms of waveforms specified by sample amplitude values and the number of sample intervals during which each is to be held. These waveforms can be made to interact with one another according to pre-defined algorithms to produce sound events with continuously varying characteristics.

### The Sawdust Session

The composer interacts with Sawdust via a standard time-sharing terminal. During a session, the composer can define and edit *objects*, which specify waveforms and how they are to interact, and can interactively *play* the objects to hear the results of the specifications. As the composer defines each object, he names it so that it can be invoked to be incorporated in the definition of other objects or to be played. The composer can edit, delete, and redefine objects during the session. The state of a session can be saved so that the composer can resume work in another session. Thus the composer can incrementally build up his piece over one or several sessions, and then record it all at once.

The simplest object in Sawdust is the *element*, which consists of an integer amplitude as accepted by the D/A

<sup>4</sup>Ibid.

<sup>5</sup>Blum, T., "Herbert Brün: Project Sawdust", *Computer Music Journal*, March 1979, vol. 3, no. 1, pp. 6-7.

converter and the integer number of sample periods for which it is to be held. An element cannot produce sound by itself, because the result of playing it would be a DC signal.

Elements can be combined into a *link*, which defines an ordered list of objects that are to be played one after another and specifies the number of times the list is to be repeated. When a link is played, each object in the list is played in order and the list is repeated according to the specification. For instance, when a link consisting only of elements is played, it produces a static waveform whose duration is the link's repeat specification multiplied by the period of the waveform.

*Mingle* and *merge* objects define two levels of interleaving list of objects. When a mingle or merge object is played, each of its component objects remains part of the resulting sound according to its repeat specification.

*Grow* and *vary* objects define two ways of producing events that begin with one waveform and end with another. Both are intended to operate on a specified first link and last link, each of which consist only of elements. The duration of either kind of object is dependent on its repeat specification.

When a *grow* object is played, Sawdust first plays the waveform defined by a single iteration of the specified first link. It then randomly changes, within specified limits, the amplitude values and sample hold values of each of the elements, and plays the result. It continues this process for the number of iterations specified by the repeat count. For the last iteration, the specified last link is played.

When a *vary* object is played, Sawdust selects polynomials that connect the amplitude value of each element of the first link with the amplitude value of the corresponding element of the last link. It does the same for the sample hold values of the corresponding elements of the two links. After first playing a single iteration of the first link, it computes and plays a new link at each iteration, according to the evaluation of the polynomial for each amplitude and sample hold value, for the number of iterations given as its repeat specification.

The Sawdust control mechanism comprises the software for interacting with the composer, encoding and storing the object definitions, and invoking the appropriate play algorithms for the objects.

## *Evolution and Sawdust*

Just as key clicks, artifacts of the implementation of a woodwind instrument, have become an accepted part of our model of what a woodwind instrument can do, Sawdust was, in one sense, an attempt to exploit an artifact of the implementation of computer-based music systems. If a computer-based music system produces sound by feeding samples to a digital-to-analog converter, then why should that not be the terms in which the composer confronts the system?

Seen from the point of view of its principal designer and implementor, Sawdust succeeded all too well in answering this question. The determination and entry of all of the amplitude and sample hold values necessary to produce a piece of any complexity has always seemed tedious to one who was not the system's principal user. A more serious drawback of this low-level approach is that a waveform's period is always an integer times the sample period, and, consequently, the frequencies of audible waveforms are limited to the set of rational numbers formed by dividing the sampling frequency (40KHz) by the integers between 2 and 2,000.

At a higher level, Sawdust was a response to a desire to compose directly with waveforms, to produce pieces through the interaction of composer-specified waveforms with each other via predetermined algorithms. Sawdust has been used to compose a number of pieces that it seems unlikely could have been produced in any other way.

Sawdust was also a response to the practical question: How can a usable interactive music system be built with a time-shared 16-bit minicomputer as the computing resource? From this point of view, Sawdust was an unqualified success, with a worst-case compute time/real time ratio of 5 to 1 on a PDP-11/70.

If Music *N* constitutes the model for computer-based music systems, then what role do systems like Sawdust play in the evolution of computer-based tools for music production? Just as composers' and performers' desires drove the explorations of mechanical musical instruments that resulted in an expansion of our model of what these instruments can do, will similar desires continue to drive the exploration of computer-based music systems to define a new model of what these systems can and should be?

## 15 Example of a Sawdust Session

```
chandra[10]<~/newlib/sawdust>./sawdust * start sawdust
```

S A W D U S T

Compiled on 08/22/04 23:12:45

OS: Darwin 6.8 Machine: Power Macintosh

Using PortAudio 18.1

```
* elist # define a list of elements
name prefix: e # the name prefix for each element
starting number = 0
name: e0 # name of 1st element
amplitude= 1000 # its amplitude
samples= 10 # its length in samples
name: e1
amplitude= 2000
samples= 20
name: e2
amplitude= 3000
samples= 30
name: e3
amplitude= 4000
samples= 40
name: e4
amplitude= / # this is sign to end input of elements

* link # define a link
name:      l1 # its name
0: e0 # the elements in this link
1: e1
2: e2
3: # 'blank' end list of elements
statements= 735 # number of iterations (735 = 44100 / linkLength)

* link # define another link
name:      l2
0: e3
1: e2
2: e1
3: e0
4:
statements= 441 # 1 second of sound
```

```

* show # display defined objects
  Show What? : all # 'all' display all objects
defined objects:
  e0          e1          e2          e3
  l1          l2
  4 elements,  2 links,   0 merges,   0 mingles,   0 vary's,   0 turns
  6 total
  Show What? : e0 # display only element 0
e0 Element 1000a, 10s
  Show What? : e1
e1 Element 2000a, 20s
  Show What? : l1 # display only link 1
l1 Link
  Expand sub-objects? (y/n) y # show its constituent parts
e0 Element 1000a, 10s
e1 Element 2000a, 20s
e2 Element 3000a, 30s
  735 statements  60 samples
  Show What? : l2
l2 Link
  Expand sub-objects? (y/n) y
e3 Element 4000a, 40s
e2 Element 3000a, 30s
e1 Element 2000a, 20s
e0 Element 1000a, 10s
  441 statements  100 samples
  Show What? : / # end 'show' command with forward slash

* save # 'save' all the objects to binary file
save file name:arun.dat

* restore # 'restore' objects from saved file
save file name:arun.dat

EOF... bye # Control-D exits sawdust

#
#
# NEW SESSION
#
#

chandra[17]<~/newlib/sawdust>./sawdust

```

S A W D U S T

Compiled on 08/22/04 23:12:45

OS: Darwin 6.8 Machine: Power Macintosh  
Using PortAudio 18.1

```
* restore # restore saved objects
save file name:arun.dat

* show # display them
Show What? : all
defined objects:
    e0          e1          e2          e3
    l1          l2
    4 elements,  2 links,   0 merges,   0 mingles,   0 vary's,   0 turns
    6 total
Show What? :

* vary # define a vary
name:      v1
first link name: l1 # initial link
(samples = 60)
last link name: l2 # final link
(samples = 100)
play last link? y # i'm not sure what this does
steps or time? time # 'time' == samples, 'steps' == num steps
time= 441000 # SR is 44100, so this is 10 seconds
freeze? y # i'm not sure what this does
4 elements in link
amplitude degrees: 3 4 5 6 # define polynomial degress for each element
sample degrees: 6 5 4 3
upper sample bounds = 10000 # not sure
lower sample bounds = 0 # not sure
statements= 1

* save # save the objects
save file name:arun.dat
save file already exists; do you wish to overwrite it?y

* play # create the soundfile

:l1 # play link 1

44100 samples, 1.0000 seconds, 0.0167 minutes

:v1 # followed by vary 1
.....
440931 samples, 9.9984 seconds, 0.1666 minutes

:l2 # followed by link 2

44100 samples, 1.0000 seconds, 0.0167 minutes
```

total: 529131 samples, 11.9984 seconds, 0.2000 minutes  
type CR when ready:

\*

EOF... bye