WavSynth

A Script-Oriented WAV Synthesizer and WAV Programming Language

A Psychotechnica Product
for the
Public Domain

Version 2
20060101

Foreword to WavSynth version 2 (and 3)

The WavSynth application allows you to craft everything from basic waveforms and sound effects to synthetic instruments and complete musical compositions, with no reliance upon pre-existing sound samples. All sounds are built up entirely from numerical sequences specified in the WavSynth language. Included with this package are a number of songs, and the musical instrument specifications they employ may be used immediately for those who do not wish to craft their own instruments. Of course you are encouraged to copy and modify these instrument definitions, and to craft instruments that no one has ever heard before.

With version 2, many new features have been added. Support for instrument definition has been extended by allowing the retrieval of multi-command sequences by name. Most importantly, the `STORETONESEQ` and `PLAYTONESEQ` commands provide far better support for chord specification, after-instrument applied note envelopes, individual note sustains and volumes, and global control of compositional key and tempo.

A `STOREPERCSEQ` command has been added to accommodate the unique requirements of percussion sequence specification. In contrast to a tonal (melodic) sequence, where a series of notes or chords are specified and subsequently applied to one or another single instrument, a percussion sequence typically involves many percussion instruments at once. Hence, the specification must provide a convenient means to indicate the timing sequence for "chords of instruments" rather than chords of notes. Correspondingly, a `PLAYPERCSEQ` command has been added to realize the stored percussion composition.

Close to 40 new wave generation and modulation functions have been added, bring the total number of waveform features to over 80.

Finally, the structure of this document has been revised. After a brief introduction the general WavSynth application and specification structure, a tutorial providing instruction in instrument generation and creation of a musical composition is given. This is followed (as before) by the usage guide to each WavSynth command, but here the organization has also changed, with functions presented according to the effect they produce (e.g., all amplitude modulations are grouped together, as are frequency modulations, special effects, etc.)

Contents

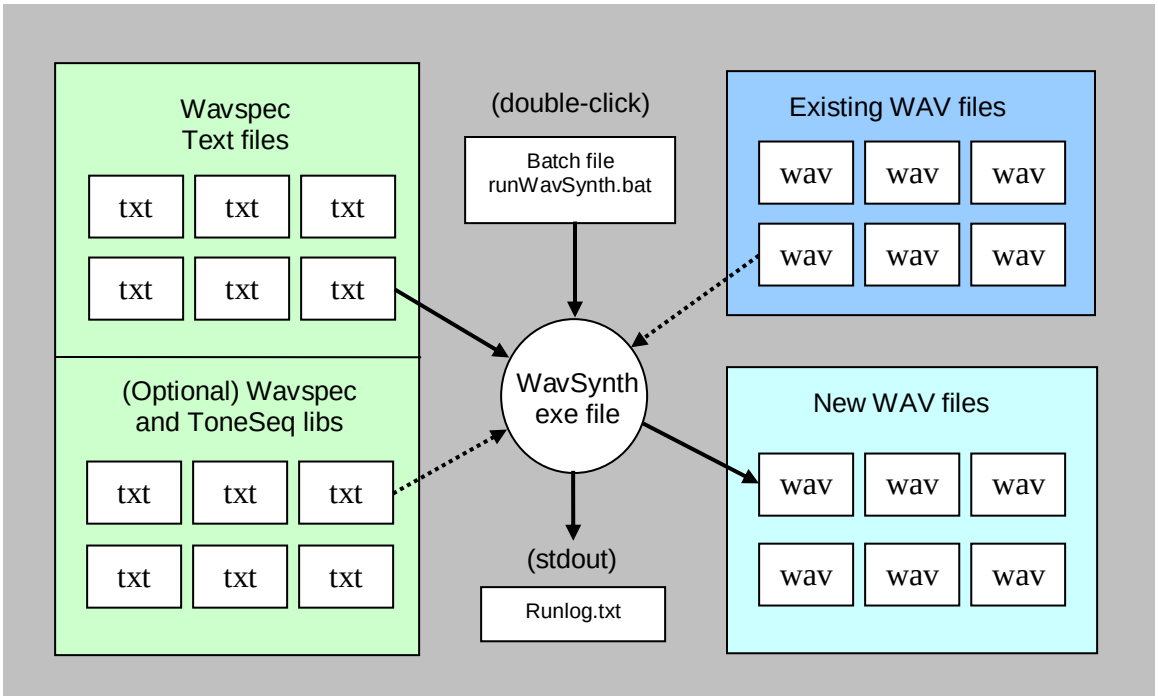
Contents (continued)

Contents (continued)

1. Overview of WavSynth

The WavSynth program accepts a plain text specification file, called a "wavspec" file, as input. According to the instructions you provide in that wavspec file, WavSynth will generate and modulate waveforms, and may load other text specifications or preexisting WAV files as well, and finally output one or more finished WAV creations.

This overview is depicted in the following diagram:



Although you may use a DOS-style commandline window to invoke the WavSynth program directly, most Windows users will prefer to double-click on a batch file in a standard Windows folder view. Such a batch file is included, called "runWavSynth.bat", and it contains the following single line of text:

```
wavsynth -i samplewavspec.txt > runlog.txt
```

You may edit this batch file by right-clicking on the batch file icon and selecting "edit". You will want to replace the name "samplewavspec.txt" with whatever instruction file you wish to have processed.

In addition to any WAV files that are produced, each run of WavSynth will overwrite a text file called "runlog.txt". This file lists each operation as it is performed, and will contain comments for most errors you may make in a wavspec file so that you may correct them.

A Quick Description of Binary WAV Files

Although Appendix A describes binary WAV files in gory detail, here it should suffice to say that a WAV file is simply a format for storing long sequences of numbers intended to describe the height (amplitude) of a waveform as it varies over time. These files can be listened to using any of a number of standard WAV players such as Windows Media Player, RealPlayer, etc. Each binary file begins with header information that tells the player application whether the file contains mono (single channel) or stereo (two channel) data, and how long a sequence of numbers represents 1 seconds worth of sound, referred to as the "samples per second" (sps), or simply the "sample rate".

Irrespective of these details, WavSynth will create (or convert upon loading) any wav data to stereo data, at 44,100 samples per second, with each numeric sample stored internally as a floating point value for maximal accuracy in subsequent manipulations. If loaded from an existing WAV file, the values (amplitudes) are scaled into the numeric range [-1.0,1.0], corresponding to the minimal and maximal values supported by that file's original format. When you `SAVE` a WavSynth created or modified file, you can specify the output format. The default will be 44,100 sps, 16-bit, stereo.

Importantly, WavSynth actually allows you to "employ" wavedata in two different ways. The first, naturally, is as "sound" to be heard or modified with various operations. In such a case, the wavedata should be "normalized" to the amplitude range [-1.0,1.0], or perhaps [-0.8, 0.8], etc, before saving to file. However, you will also be using wavedata purely as varying "modification values", in applying certain operations to the first form of wavedata. This second form is not intended to be "heard" directly, and its amplitudes may take on any real values as needed.

Preview of Section 2 and Section 3

In Section 2, we will introduce the general structure of the text-based wavspec file, to familiarize you with its purpose and terminology. Then, in section 3, we will provide a tutorial that will lead you through simple waveform and sound generation, the definition of instrument sounds, the definition of a melody, and execution of the melody using one or more instruments, and the mixing of composed musical phrases into a final composition.

2. WavSpec File Structure

2.1 Minimal WavSpec Structure

The "wavspec" file is the controlling file for your WavSynth session.

Each wavspec file is simply a text file containing a structure of parenthetically-delimited commands. The minimal format is given by this example:

```
(
   projectname
   ( SYNTHINIT )
   ( SYNTHCMDS )
)
```

The term "`projectname`" is any name you wish to use to identify your wavspec file. It must be a string token, meaning it cannot contain whitespace, and you cannot override this by using quotes. Also, string tokens cannot contain parentheses!

The "`SYNTHINIT`" section may be empty (as shown) but the "`SYNTHCMDS`" section must contain the commands you wish to process. They are processed in the order they appear.

For a trivial example, suppose you wish to create a wav file containing 2 seconds of a simple sine wave, at frequency 100 hz, amplitude 0.9, and name that file "`my100Hz.wav`". The following wavspec file will do:

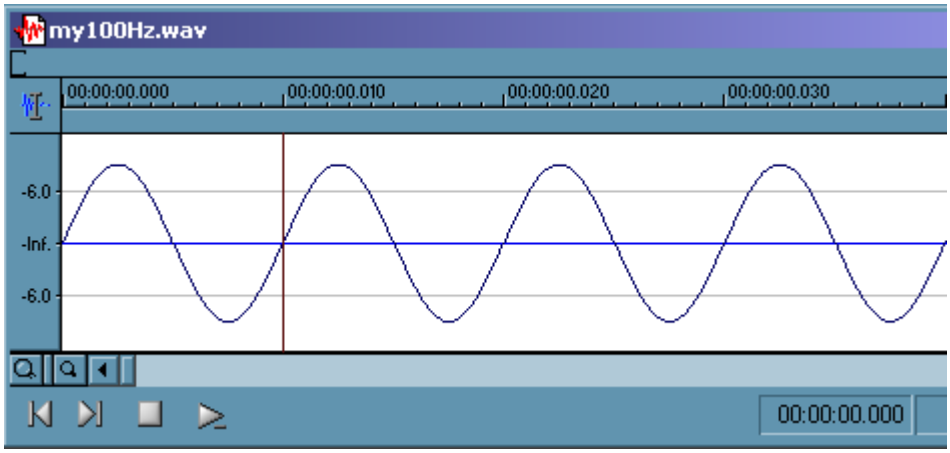
```
(
   projectname
   ( SYNTHINIT )
   ( SYNTHCMDS
      ( tmp ( SIMPFORM SIN 100 0.9 2.0 ) )
      ( SAVE tmp my100Hz.wav )
   )
)
```

If you place these commands in an ordinary text file, call the file "`mytest.txt`", and then edit the batch file "`runWavSynth.bat`" to invoke it, you can double-click on the batch file and you should see the WAV file "`my100Hz.wav`" appear in the directory.

Note that I used two separate commands. The first invokes the `SIMPFORM` routine to create the wavedata and hold it in a memory location under the (arbitrary) name "`tmp`". By doing this, I make that waveform data available to many subsequent commands without having to invoke `SIMPFORM` again. In the case shown, I did not really need to do that, and I could have used the single command

```
   ( SYNTHCMDS
      ( SAVE ( SIMPFORM SIN 100 0.9 2.0 ) my100Hz.wav )
   )
```

You can generally use some default Windows application to listen to the waveforms you generate (`my100Hz.wav` will be rather boring) but if you have a product like Sound Forge Studio from Sonic Foundry, you can actually see the flow of the finished waveform in varying detail, and can even watch the form changing as it plays. I found this product to be invaluable in helping me to confirm the behaviors of my WavSynth program. Sound Forge Studio also has many tools you can use to do fast editing and merging of wav files. I have used it in this document to display the look of various example waveforms. For instance, the wav file "`my100Hz.wav`" looks like this:



Notice that the sine wave completes one full cycle after exactly 0.01 seconds, corresponding to our intended frequency of 100 cycles per second.

We created the simple sinewave data using the command

```
( tmp ( SIMPFORM SIN 100 0.9 2.0 ) )
```

In general, most WavSynth commands have the common structure

```
( dataname ( COMMAND parm1 parm2 parm3 ... ) )
```

That is, you begin by providing a "`dataname`", and then invoke some `COMMAND` with additional parameters to create the desired wavedata, and make that data available to you under whatever `dataname` you supplied. Such wavedata will sometimes be referred to as "homed" wavedata, for it will persist in memory under the given name until you explicitly `FREE` it, or when the WavSynth program exits.

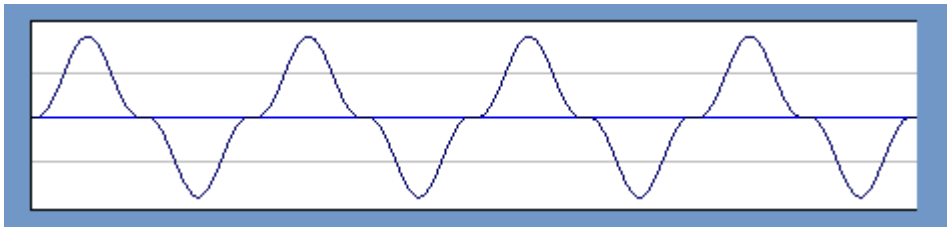
Some commands (like `SIMPFORM`) need no wavedata to begin with – they create the data from scratch. Other commands operate on existing wavedata and return new or modified versions. For instance, we could change the shape of our sinewave data using the NEWPOWER command

```
( tmp2 ( NEWPOWER tmp 3.0 ) )
```

The following wavspec file illustrates this modification:

```
(
   projectname
   ( SYNTHINIT )
   ( SYNTHCMDS
      ( tmp  ( SIMPFORM SIN 100 0.9 2.0 ) )
      ( tmp2 ( NEWPOWER tmp 3.0 ) )
      ( SAVE tmp2 my100Hz-P3.wav )
   )
)
```

The resulting wavedata looks like this:



Notice that one of the parameters passed to the NEWPOWER command was the pre-existing (homed) wavedata "tmp". It has not itself been modified by the NEWPOWER command, and either of the two sets of wavedata tmp and tmp2 can continue to be used in subsequent commands. On the other hand, if there were no reason to generate tmp except to make tmp2, we could have used one just command, as follows

```
(
   projectname
   ( SYNTHINIT )
   ( SYNTHCMDS
      ( tmp ( NEWPOWER ( SIMPFORM SIN 100 0.9 2.0 ) 3.0 ) )
      ( SAVE tmp my100HzP3.wav )
   )
)
```

In this case, the wavedata that NEWPOWER is acting upon (from SIMPFORM) is called "homeless" wavedata – data that is temporarily created but has not be given a named location. That data is acted upon directly to produce the final product, saving memory.

On the following page, we explore the amplitude modulation of one waveform with another, and introduce the concept of a "waveform library", a construct that provides better efficiency in varied waveform generation and helps you to organize your creations.

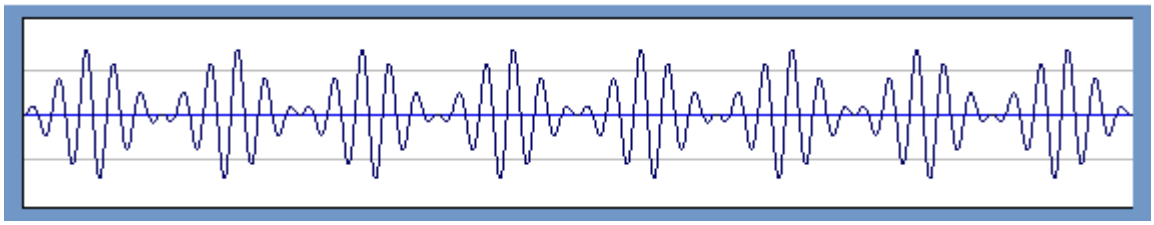
2.2 Advanced WavSpec Structure

If you get serious producing sound effects and music using WavSynth, you will create many wavspec files of considerable complexity. In particular, you will find yourself defining and re-using scores of different envelopes and other waveforms, to be used in new ways to produce effects. It can become tedious to repeatedly "copy" the text of all your favorite waveform definitions into each new wavspec file. For this reason, the wavspec structure supports several features that mitigate these complexities.

In order to motivate this section, consider the following outline of a moderate sized wavspec file. It generates many waveforms, and uses one of them to modulate the amplitude of the last one:

```
(
   projectname
   ( SYNTHINIT )
   ( SYNTHCMDS
      ( wf1 ( SIMPFORM SIN 10 0.8 2.0 ) )
      ( wf2 ( SIMPFORM SQR 10 0.8 2.0 ) )
      ( wf3 ( SIMPFORM SAW 10 0.8 2.0 ) )
      ( wf4 ( SIMPFORM SW2 10 0.8 2.0 ) )
      ( wf5 ( SIMPFORM ELI 10 0.8 2.0 ) )

      ( thing1 ( SIMPFORM SIN 100 0.8 2.0 ) )
      ( thing2 ( NEWAMODW thing1 wf4 0.9 ) )
      ( SAVE thing2 trial.wav )
   )
)
```

Take note of the command `NEWAMODW`. It operates on "`thing1`", a simple 100hz sine wave, by modulating its amplitude using `wf4`, a type of 10hz sawtooth wave. The result is called "`thing2`", and looks like this:



Suppose you wanted to know how it would turn out if you had used `wf5` as an amplitude modulator instead of `wf4`. You could simply edit the definition of `thing2` to use `wf5`. Convenient, but very inefficient. Why? Because every time you run WavSynth to process this wavspec file, ALL of the waveforms are produced, even though you only end up using a few of them. This is because they are all listed under the `SYNTHCMDS` section.

Below, we introduce a new wavspec section called `SYNTHWLIB` that addresses this.

2.2.1 The SYNTHWLIB Section

Contrast the previous wavspec with this new one, in which we introduce a new section identified by the keyword "SYNTHWLIB":

```
(
   projectname
   ( SYNTHINIT )
   ( SYNTHWLIB
      ( wf1 ( SIMPFORM SIN 10 0.8 2.0 ) )
      ( wf2 ( SIMPFORM SQR 10 0.8 2.0 ) )
      ( wf3 ( SIMPFORM SAW 10 0.8 2.0 ) )
      ( wf4 ( SIMPFORM SW2 10 0.8 2.0 ) )
      ( wf5 ( SIMPFORM ELI 10 0.8 2.0 ) )
      ( thing1 ( SIMPFORM SIN 100 0.8 2.0 ) )
   )
   ( SYNTHCMDS
      ( thing2 ( NEWAMODW thing1 wf4 0.9 ) )
      ( SAVE thing2 trial.wav )
   )
)
```

When WavSynth processes this file (after processing the empty SYNTHINIT section) it will jump directly to the SYNTHCMDS section, and always process the commands it finds there **in the order they appear**.

First, it will encounter the definition for "thing2", and call upon the NEWAMODW command to produce it. The NEWAMODW function will in turn see that it needs two sets of wavedata named "thing1" and "wf4", and will check (in WavSynth's internal "wavbank") to see if they have already been produced. But they have not. So, it will next search in the (new) SYNTHWLIB section to see if there are definitions for these objects. Thankfully, it finds each of them, and executes their definitions (they now become "homed" data, just as if they had been listed in the SYNTHCMDS section as before). Finally, NEWAMODW has all the waveforms it needs to produce "thing2".

Note that ONLY those waveforms needed by the SYNTHCMDS section were ever sought and produced. All the others (wf1, wf2, wf3, wf5) were ignored as unused. This saves considerable processing time and system resources (memory.)

Fine – but you may have hundreds of "favorite" waveforms you like to use, and you may have many different project (wavspec) files. Do you have to copy your often-used forms into each wavspec file manually?

Fortunately, you do not. The following command demonstrates how you can place any number of waveform definitions into one or more external "wavlib" files, and have them loaded automatically. This will be very useful when we begin defining "instruments".

2.2.2 The LOADWLIB Command

Adding a SYNTHWLIB section to hold a "library" of envelopes and other waveforms helps considerably in organizing your constructions and making the wavspec processing very efficient. But as you create many different wavspec files for different projects, you would have to copy all of your favorite "forms" into each new file.

You can avoid this by placing your favorite waveform specifications into one or more "wavlib" text files, and having them loaded automatically at runtime. Taking the previous example, you could create a separate text file to hold exactly these contents:

```
( MyFavoriteForms
   ( wf1 ( SIMPFORM SIN 10 0.8 2.0 ) )
   ( wf2 ( SIMPFORM SQR 10 0.8 2.0 ) )
   ( wf3 ( SIMPFORM SAW 10 0.8 2.0 ) )
   ( wf4 ( SIMPFORM SW2 10 0.8 2.0 ) )
   ( wf5 ( SIMPFORM ELI 10 0.8 2.0 ) )
   ( thing1 ( SIMPFORM SIN 100 0.8 2.0 ) )
)
```

Suppose you save this file under the name "wavlib-01.txt". You could then change the wavspec file listed on the previous page to add a **LOADWLIB** command to the SYNTHINIT section, instead of literally copying those specifications into a SYNTHWLIB section. The format would appear as depicted below:

```
(
   projectname
   ( SYNTHINIT
      ( LOADWLIB wavlib-01.txt )
   )
   ( SYNTHCMDS
      ( thing2 ( NEWAMODW thing1 wf4 0.9 ) )
      ( SAVE thing2 trial.wav )
   )
)
```

When WavSynth processes this file, it will always execute the special commands listed in the SYNTHINIT section (if there are any) before processing the standard commands in the SYNTHCMDS section. This will have the effect of reproducing the missing SYNTHWLIB section. You may specify as many separate wavlib files as you please. I tend to create separate libs for envelopes and for other waveforms, instruments, etc, and use a command like

```
( LOADWLIB envelopes.txt waveforms.txt violins.txt )
```

The contents specified in each lib file will be placed into a "virtual" SYNTHWLIB section, prior to executing your SYNTHCMDS instructions.

QUESTION:

Can you use LOADWLIB if your project already contains a SYNTHWLIB section?

Yes, the following wavspec file will work just fine:

```
(
   projectname
   ( SYNTHINIT
      ( LOADWLIB wavlib-01.txt )
   )
   ( SYNTHWLIB
      ( envXY ( ENVELOPE [an envelope specification . . .] ) )
      ( env22 ( ENVELOPE [another envelope specification . . .] ) )
      ( bleem ( SIMPFORM SW2 100 0.5 2.0 ) )
   )
   ( SYNTHCMDS
      ( thing2 ( NEWAMODW thing1 wf4 0.9 ) )
      ( SAVE thing2 trial.wav )
   )
)
```

The "local" specifications for "envXY", "env22", and "bleem", as well as any that occur in the external file "wavlib-01.txt" will ALL be placed into a global specification space, and be available to the commands you use in your SYNTHCMDS section.

QUESTION: (Important)

What will happen if you create a local definition for "envXY" in your SYNTHWLIB, (as shown above), and it so happens that the externally loaded form library "wavlib-01.txt" also contains a (different) waveform with the name "envXY"?

What if you load two libraries, "wavlib-01.txt" and "wavlib-02.txt", and some specifications in each have the same name?

The answer is that ALL of them get loaded, with local definitions first, then the members of each library file in turn. This means that when one of your SYNTHCMDS reads:

```
( coolOne ( NEWAMODW wf222 envXY 0.9 ) )
```

WavSynth will search for a definition for "wf222" and "envXY" in the SYNTHWLIB, and use the FIRST ONE it finds. This may lead to unexpected results if it finds one you did not expect it to use.

LESSON: Develop a good naming scheme for your often-used waveforms!

2.2.3 The LOADTLIB and LOADPLIB Commands

Just as you can place potentially-used waveforms manually into a SYNTHWLIB section, or have them loaded automatically (in SYNTHINIT) using the LOADWLIB command, you can manually create local sections named SYNTHTLIB and SYNTHPLIB for defining tonal or percussion sequences, or have them loaded automatically from external files using the LOADTLIB and LOADPLIB commands. The use of these sections and commands exactly parallels that of SYNTHWLIB and LOADWLIB – local definitions get the highest priority in the event of name-collisions, followed by any external files in the order they are loaded.

Because tonal sequences and percussive sequences get their own sections, there is no need to worry about name-collisions between waveforms and these other constructs.

Also, because tonal melody specifications, and complex percussion sequences can be rather lengthy, it is rare to place them directly into your main wavspec file, at least in their entirety. Personally, as I construct a piece of music, I find it most useful to place the most stable parts of the composition into external "tone library" files, under the given song-name, load those parts using the LOADPLIB command, but place the unfinished parts of the composition directly in the wavspec file where the "mixing" occurs. This makes it easy to try different modifications, and then invoke WavSynth to put it all together.

The Melodic or Tonal sequences are specified using the STORETONESEQ command, and percussion sequences are specified with the STOREPERCSEQ command. As these are rather involved constructs with their own unique formats, we will defer delving into these until the tutorial in the next section.

Before that, we must revisit a few issues regarding waveform generation.

2.2.4 Compound Sequence Specification

2.2.4.1 SYNTH
2.2.4.2 RETURN

Discussion: Recall that you can place waveform definitions in external files (called "wavlibs"), automatically load them into your active wavspec file using the LOADWLIB command in the SYNTHINIT section, and then call upon the waveforms by name in your SYNTHCMDS section. An external wavlib file might look like:

```
( mystuff
   ( sin100 ( SIMPFORM SIN 100 0.9 4.0 ) )
   ( saw100 ( SIMPFORM SAW 100 0.9 4.0 ) )
   ( weird1 ( NEWPMODW ( NEWNOISE 0.5 0.9 4.0 ) ( SIMPFORM . . . ) )
   ( weird2 ( NEWPMODW ( NEWNOISE 0.5 0.9 4.0 ) saw100 )
   [etc etc . . .]
)
```

The idea was, in your SYNTHCMDS section, you could now issue commands like

```
( wav27 ( NEWREVERSE weird1 ) )
```

and not have to recall how weird1 had been defined. Its definition would simply be "found" and executed the first time it was referenced, and the resulting wavedata would thereafter be available in memory by that name. Moreover, if wav27 was the only reason you needed weird1, you could simply (FREE weird1) and continue to use wav27.

But there are two major drawbacks. The first of these is seen if you instead issue

```
( wav27 ( NEWREVERSE weird2 ) )
```

Notice that the definition of weird2 involves another "named" wavedata, "saw100". No problem so far, you need not remember that fact, because the definition for saw100 will then (automatically) be sought, found, executed, and placed into memory, THEN applied to NEWNOISE using NEWPMODW, and thereafter be available as weird2. But what happens if, as before, you no longer need weird2 and want to free it from memory? If you issue the command (FREE weird2), its data will indeed be freed from memory, but the wavedata "saw100" has not been freed, and you wouldn't know to free it unless you opened your wavlib file and read it to tell what had transpired, which tends to defeat the whole "define it and forget it" reason for creating external wavlib files in the first place.

Secondly, note that each externally named wavedata definition involves only a single (if sometimes nested) command. There is no way to save and later invoke a *sequence* of independent commands under a single name.

This situation has now been fixed by introducing the SYNTH and RETURN commands.

As illustration, suppose you had already stored away a melody called "Lullaby" in a file called "Lullaby.txt", and you wanted to craft a piano-like instrument to play that melody. Your wavspec file might look as follows:

```
( myExperiment
   ( SYNTHINIT
      ( LOADTLIB mySongs/Lullaby.txt )
   )
   ( SYNTHCMDS
      ( basis1 ( SIMPFORM SW2 3200 0.999 4.0 ) )
      ( basis2 ( REVERSE ( ANTIFSMEAR basis1 0.7071 0.75 ) ) )
      ( FREE basis1 )
      ( mypiano ( CHORUS basis2 800.0 7 0.01 0.8 3 1 ) )
      ( FREE basis2 )

      ( mySong ( PLAYTONESEQ Lullaby mypiano ( REVERB ) ) )
      ( SAVE mySong MyLovelyLullaby.wav )
   )
)
```

Notice that crafting the "piano" involved 5 commands and two intermediate sets of wavedata (basis1 and basis2) which were freed from memory once no longer needed. How could you "store away" the entire command sequence used to create the "piano" instrument, so it could be called up by name? To see this, consider the following external wavlib file (compare to previous):

```
( mystuff
   ( sin100 ( SIMPFORM SIN 100 0.9 4.0 ) )
   ( saw100 ( SIMPFORM SAW 100 0.9 4.0 ) )
   ( mypiano
      ( SYNTH
         ( basis1 ( SIMPFORM SW2 3200 0.999 4.0 ) )
         ( basis2 ( REVERSE ( ANTIFSMEAR basis1 0.7071 0.75 ) ) )
         ( FREE basis1 )
         ( apiano ( CHORUS basis2 800.0 7 0.01 0.8 3 1 ) )
         ( FREE basis2 )
         ( RETURN apiano )
      )
   )
   ( weird1 ( NEWPMODW ( NEWNOISE 0.5 0.9 4.0 ) ( SIMPFORM . . . ) )
   ( weird2 ( NEWPMODW ( NEWNOISE 0.5 0.9 4.0 ) saw100 )
)
```

Essentially, the `SYNTH` command identifies a command-sequence group, whose commands are executed sequentially just as if they had appeared in your wavspec command section. Any wavedata no longer needed have been freed, and the wavedata that represents the desired result (`apiano`) is identified by the `RETURN` command, and automatically renamed "`mypiano`" as the externally visible name.

Compare the wavspec at the top of this page to the revised one on the next page.

New wavspec file:

```
( myExperiment
   ( SYNTHINIT
      ( LOADWLIB myForms.txt )
      ( LOADTLIB mySongs/Lullaby.txt )
   )
   ( SYNTHCMDS
      ( mySong ( PLAYTONESEQ Lullaby mypiano ( REVERB ) ) )
      ( SAVE mySong MyLovelyLullaby.wav )
   )
)
```

Notice that you can now call upon the instrument "`mypiano`" and have the entire complex of its definition (in the wavlib "`myForms.txt`") automatically execute to make the wavedata available for `PLAYTONESEQ Lullaby`. Moreover, the intermediate waveforms that were generated to create the "piano" sound have already be disposed of.

2.3 Complete WavSpec Structure Outline

Below you will find the complete structural outline to a general wavspec file. Note that only those sections indicated in **boldface** type are required – all others are optional.

```
(
   projectname
   ( SYNTHINIT
      ( LOADWLIB wavlib-01.txt )     [optionally load wavspec libs]
      ( LOADWLIB wavlib-02.txt )     [optionally load wavspec libs]
      ( LOADTLIB tonelib-01.txt )    [optionally load tone libs]
      ( LOADPLIB perclib-01.txt )    [optionally load perc libs]
      ( LOADFLIB fourierlib-01.txt ) [optionally load Fourier libs]
   )
   ( SYNTHWLIB          [optional section for local waveform specs]
        [place potentially-used waveform specs here]
   )
   ( SYNTHTLIB          [optional section for local tone sequences]
        [place potentially-used tone sequences here]
   )
   ( SYNTHPLIB          [optional section for local perc sequences]
        [place potentially-used percussion sequences here]
   )
   ( SYNTHCMDS
        [place sequentially executed WAV synthesis commands here]
   )
)
```

To reiterate, only the SYNTHCMDS section, and a (possibly empty) SYNTHINIT section are required. The three sections named SYNTHWLIB, SYNTHTLIB and SYNTHPLIB are all optional and may be omitted. Moreover, if you load any external waveform specifications, tonal or percussion sequences using the "LOAD*LIB" commands, they become available whether or not the corresponding sections have been defined manually.

Notice that I indicate, by example, loading 4 libraries named wavlib-01.txt, wavlib-02.txt, tonelib-01.txt, and perclib-01.txt. More realistically, these might be named

wavlib-01.txt	[(wavlib) to hold common basic waveforms]
envelopes.txt	[(wavlib) to hold common note-strike amplitude envelopes]
horns-01.txt	[(wavlib) instruments crafted to sound like horns]
pianos-01.txt	[(wavlib) instruments crafted to sound like pianos]
mySong.txt	[(tonelib) to hold melodic parts of mySong]
mySongPercs.txt	[(perclib) to hold percussion parts of mySong]

Remember, you can load as many "libs" as you like – the only ones that actually get processed are those named elements that end up being references by commands given in the SYNTHCMDS section.

3. WavSynth Composition Tutorial

Overview

In this tutorial, we will craft a few musical instruments, learn about envelope generation and how these can be applied to shape the way an instrument sound is formed. We will then introduce the STORETONESEQ and PLAYTONESEQ commands in defining a chordal melody and "playing" the melody by applying one or more instruments. We will also create a short percussive rhythm phrase using STOREPERCSEQ and PLAYTONESEQ. Lastly, we will take a few of these phrases and mix them into the first few bars of a song.

In short, beginning purely from scratch, a "song" can be created in a single, short wavspec file by following this simple "recipe":

Create Instrument(s)

Define Melodic Phrase(s)

Then repeatedly

Apply a selected Instrument to a selected Melodic Phrase
Mix with previous phrases

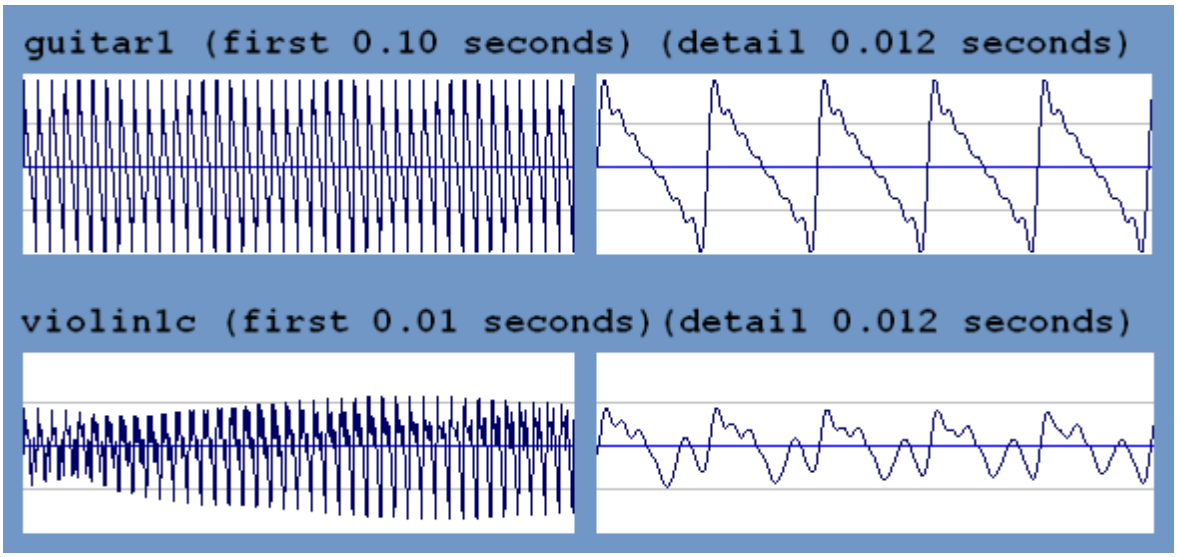
Until done.

Part 1: Instrument Creation

We will begin by creating two simple "string-like" instruments. One will serve as a bass guitar, and the second will sound a bit more like an electric violin. Both use the same basic waveform, a Fourier approximation to a sawtooth wave:

```
( tutorial
   ( SYNTHINIT )
   ( SYNTHCMDS
      ( guitar1 ( FOURIER INTEGERS 7 400.0 0.999 2.0 ) )
      ( violin1c ( CHORUS guitar1 400.0 ) )
      ( SAVE guitar1 guitar1.wav )
      ( SAVE violin1c violin1.wav )
   )
)
```

The wavspec file above creates two wav samples, each 2 seconds long, at 400hz. If you save this text as "mytutorial.txt" and run "wavsynth –i mytutorial.txt > runlog.txt" via the batch file, you should find two rather boring wav files created. The first will sound a bit like the Emergency Broadcast Signal, the second is just a chorusing effect applied to the first, and is (thus) quasi-periodic. The "look" of these waveforms appears below.

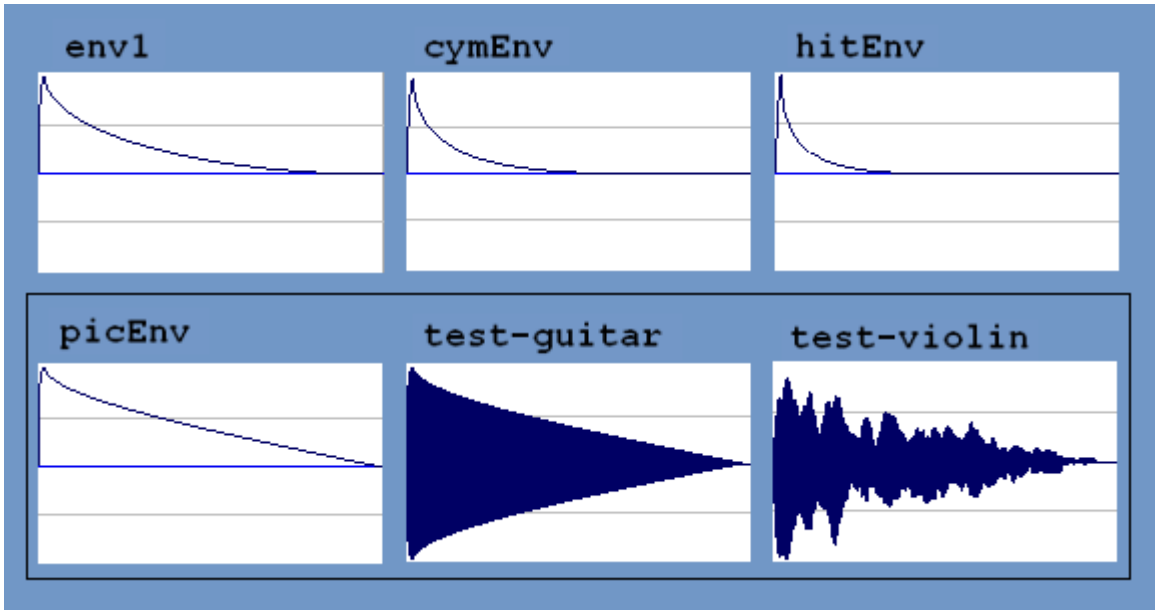


These certainly do not sound much like a guitar or violin, yet. We must "shape" them using some amplitude envelopes.

The wavspec file below produces our guitar and violin "basic" wavedata, and four envelopes. The first envelope "`env1`" simply rises from 0 to 0.99 in the first 1/50th of its length (shape = `HEL`, hard-elliptic) and then decays back to 0 over the remaining 49/50th of its length, using the same `HEL` shape. The other three envelopes (`picEnv`, `cymEnv`, and `hitEnv`) are produced by applying a power function to the first envelope.

In the new wavspec file below, we generate these envelopes, and then test-shape the amplitude of the guitar and violin data by applying the picEnv. We save the results to give them a listen. The following page graphically depicts these waveforms.

```
( tutorial
   ( SYNTHINIT )
   ( SYNTHCMDS
      ( guitar1 ( FOURIER INTEGERS 7 400.0 0.999 2.0 ) )
      ( violin1c ( CHORUS guitar1 400.0 ) )

      ( env1 ( ENVELOPE ( DURA 2.0 )
            ( COMPONENTS ( HEL 1 0.99 ) ( HEL 49 0.0 ) ) ) )

      ( picEnv ( NEWPOWER env1 0.5 ) )
      ( cymEnv ( NEWPOWER env1 2.0 ) )
      ( hitEnv ( NEWPOWER env1 3.0 ) )

      ( test-guitar ( NEWAMODENV guitar1 picEnv ) )
      ( SAVE test-guitar test-guitar.wav )

      ( test-violin ( NEWAMODENV violin1c picEnv ) )
      ( SAVE test-violin test-violin.wav )

   )
)
```

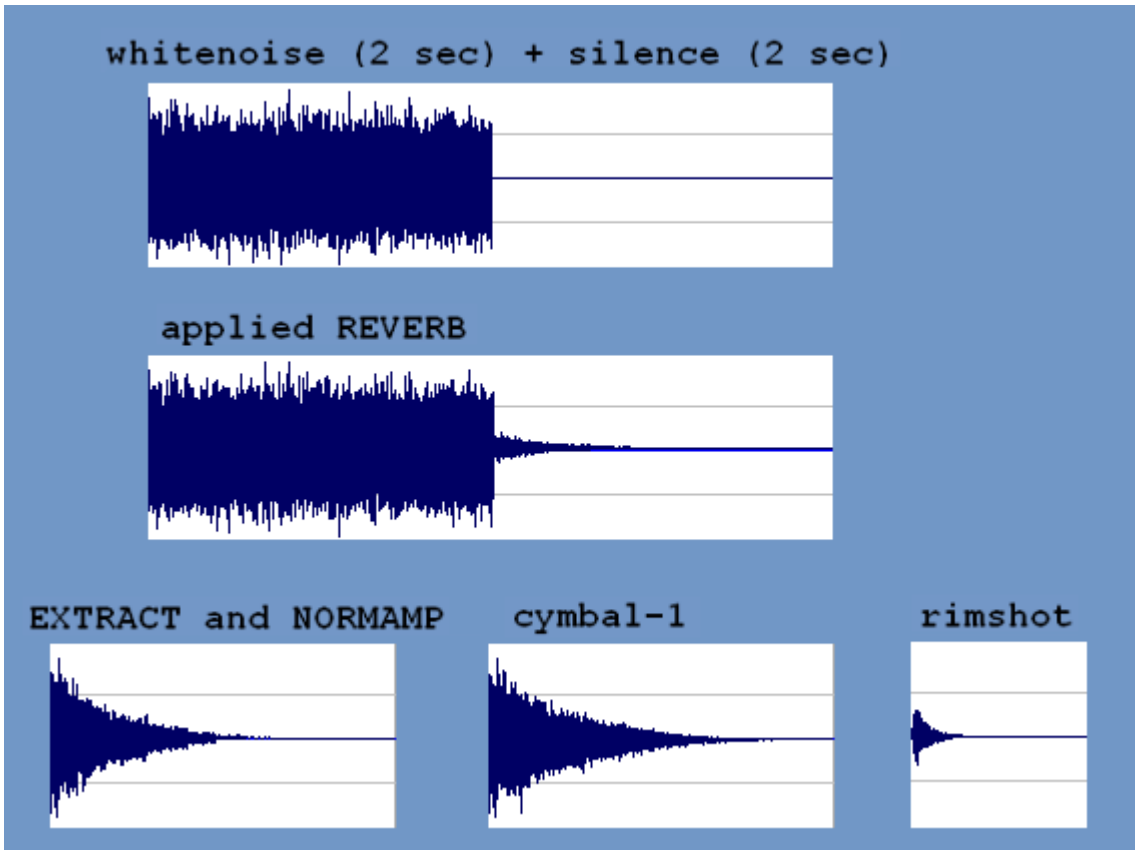


Each of the samples illustrated above are of a full 2.0 second duration. The `test-guitar` and `test-violin` appear solid-blue at this scale, but in detail, they will appear more like the forms on the previous page. Also the sounds of these instruments will improve when we play them at lower pitch and apply other effects.

Next, we will craft a few simple percussion instruments, a cymbal and a rimshot (the sound of striking the metallic rim of a snare drum.) In this case, our "basic" waveform will be white-noise.

```
( tutorial
   ( SYNTHINIT )
   ( SYNTHCMDS
      ( guitar1 ( FOURIER INTEGERS 7 400.0 0.999 2.0 ) )
      ( violin1c ( CHORUS guitar1 400.0 ) )

      ( env1 ( ENVELOPE ( DURA 2.0 )
           ( COMPONENTS ( HEL 1 0.99 ) ( HEL 49 0.0 ) ) ) )
      ( picEnv ( NEWPOWER env1 0.5 ) )
      ( cymEnv ( NEWPOWER env1 2.0 ) )
      ( hitEnv ( NEWPOWER env1 3.0 ) )

      ( noise ( APPEND ( NEWNOISE 0.01 0.99 2.0 ) ( SILENCE 2.0 ) ) )
      ( rnoise ( EXTRACT ( REVERB noise ) 2.001 4.0 ) )
      ( NORMAMP rnoise 0.9 )
      ( cymbal-1 ( REVERB ( REVERB rnoise ) ) )
      ( tmp-hit ( NEWAMODENV cymbal-1 hitEnv 0.8 0.5 ) )
      ( rimshot ( SMOOTH tmp-hit 0.0001 ) )

      ( SAVE cymbal-1 cymbal-1.wav )
      ( SAVE rimshot rimshot.wav )
   )
)
```

Let us explore these steps in some detail. Notice that I start by generating 2 seconds of white noise using the NEWNOISE function, but I then APPEND another 2 seconds of SILENCE, and call this wavedata "noise". Why the 2 seconds of silence? Because I am going to apply REVERB, and when reverb is applied to a sound, the reverberant echoes carry beyond the original sound, and I wish to capture the part of this reverberance where it spills over into the silent area. This pure reverberance has much lower volume, so I raise it back up to amplitude 0.9 with the NORMAMP function.



I create the instrument `cymbal-1` by applying two more rounds of REVERB. This has the effect of producing a slightly metallic "ring" to the latter part of the cymbal. I create `rimshot` directly from `cymbal-1` in two steps. First, I apply the rapid-decay envelope "`hitEnv`" to dampen the sound, then I apply SMOOTH with a sliding window of 0.0001 seconds in order to trim away some of the high frequency "hiss".

In the above wavspec file, I did not bother to save the original "`noise`" and "`rnoise`" wavedata to file, but you might do this in order to hear the sound of these intermediate constructs.

Now, armed with our instruments, we are ready to write some music.

Part 2: Melody and Percussion Definition

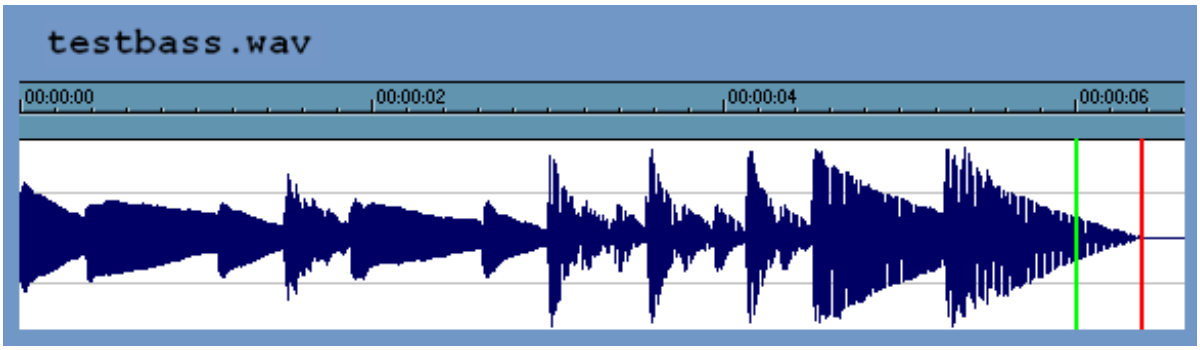
Admittedly, long melodic and percussive sequences are rather tedious to define. Because they are often long, one generally places them into a separate "tonelib" or "perclib" file, and loads them as needed at runtime, rather than place them directly into the controlling wavspec file. However, to quickly illustrate their application, I will keep them in the wavspec, as depicted below. For clarity, I have temporarily removed elements pertaining to percussion instruments, so we can focus upon melody.

```
( tutorial
   ( SYNTHINIT )
   ( SYNTHCMDS
      ( guitar1 ( FOURIER INTEGERS 7 400.0 0.999 2.0 ) )

      ( env1 ( ENVELOPE ( DURA 2.0 )
               ( COMPONENTS ( HEL 1 0.99 ) ( HEL 49 0.0 ) ) ) )

      ( picEnv ( NEWPOWER env1 0.5 ) )

      ( STORETONESEQ bassline1
         ( 0.00 ( G2 0.30 1.6 ) )
         ( 0.20 ( G2 0.90 ) )
         ( 0.60 ( G2 0.30 1.2 ) )
         ( 0.80 ( F2 0.30 1.6 ) )
         ( 1.00 ( F2 0.90 ) )
         ( 1.40 ( F2 0.30 1.2 ) )
         ( 1.60 ( C2 0.12 1.6 ) ( C3 0.12 1.2 ) )
         ( 1.70 ( C2 0.12 ) )
         ( 1.80 ( C2 0.12 ) )
         ( 1.90 ( C2 0.12 1.6 ) ( C3 0.12 1.2 ) )
         ( 2.00 ( C2 0.12 ) )
         ( 2.10 ( C2 0.12 ) )
         ( 2.20 ( C2 0.12 1.6 ) ( C3 0.12 1.2 ) )
         ( 2.30 ( C2 0.12 ) )
         ( 2.40 ( D2 0.60 1.6 ) ( D3 0.60 1.2 ) )
         ( 2.80 ( C2 0.60 1.6 ) ( C3 0.60 1.2 ) )
      )

      ( testbass ( PLAYTONESEQ bassline1 guitar1 ( TEMPO 0.533333 )
            ( KEYMOD -60 ) ( KEYENV picEnv 50 ) ( REVERB ) ) )

      ( SAVE testbass testbass.wav )
   )
)
```

The STORETONESEQ command create the "score", but not the sound. Here, I chose to separate most notes by 0.20 seconds (think "quarter notes"). I wrote the notes so they would sustain a bit past the next scheduled note, and some notes are sounded louder than others (they have the third parameter 1.6 or 1.2). Naturally, chords are created when you make multiple notes sound at the same time.

In contrast, the PLAYTONESEQ command creates the actual wavedata, applying our stored melody "baseline1" to the instrument "guitar1". Give testbase.wav a listen!

We need to explain the PLAYTONESEQ command in a bit more detail, in particular, the selection of (TEMPO 0.533333). To motivate this, here is a view of the resulting wavedata, "testbass".



Notice that I have placed a green and red line on the image. The green line represents the "rhythmic length" of the phrase. Because I have made each of the notes sustain beyond the next scheduled note (by about 50%), the notes blend over one another nicely, and the last one (thus) extends beyond the rhythmic length of 6.0 seconds. This is perfectly fine, because when we decide to add a second phrase to extend this one, we will simply lay it down at the 6.0 second mark, allowing the extra sustain of the first phrase to "sound through" into the second one naturally.

But none of these lengths seem to correspond to the original score. Consider, the last two notes (chords) of the score were

```
( 2.40 ( D2 0.60 1.6 ) ( D3 0.60 1.2 ) )
( 2.80 ( C2 0.60 1.6 ) ( C3 0.60 1.2 ) )
```

If not for the extended sustains (0.60 seconds each), the note at 2.40 would last until 2.80, for a sustain of 0.40 seconds. If the very last note sustained another 0.40 seconds, the entire piece would have a "rhythmic length" of 3.2 seconds. Let us refer to this value, 3.2 seconds, as the "scored rhythmic length" of the phrase. Played this way, it was far too fast, and I initially tried it with a TEMPO of 0.5 to play it half as fast. This would give the entire phrase a rhythmic length of 6.4 seconds (since 3.2/0.5 = 6.4). But this was just a bit too slow. I decided that a rhythmic length of 6.0 seconds would be best. This meant that I needed to apply the formula

```
TEMPO = "scored rhythmic length"/"desired rhythmic length"
      = 3.2/6.0 = 5.33333 …
```

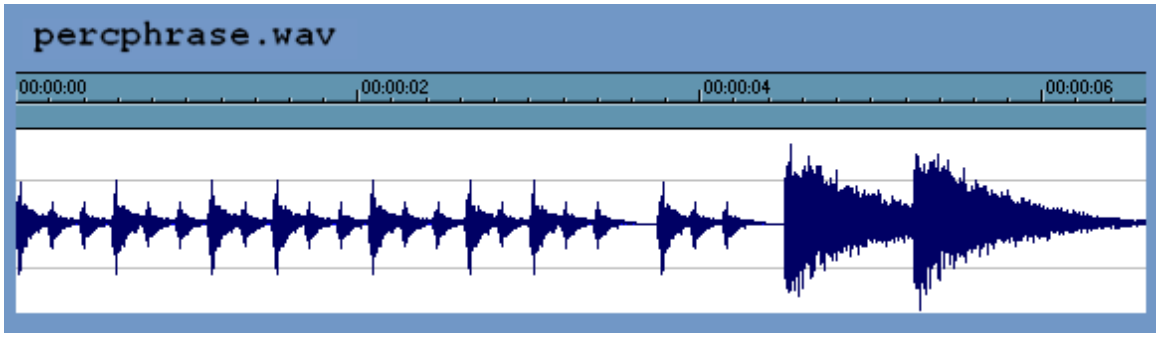
Next, our "guitar1" was formed at 400hz, a rather high tone, and the score adds a few octaves on top of this. But I wanted a "bass guitar" sound, so I dropped everything 5 octaves (5x12 = 60 semitones) using (KEYMOD -60). I also applied our "picEnv" envelope to shape the guitar strikes. Since picEnv is 1-part attack, 49-parts decay, I must let PLAYTONESEQ know that the attack is only 1/50th of the envelope, so that when it stretches or compresses the envelope to longer or shorter notes, only the decay part is affected. Finally, I have REVERB applied to the result.

Almost in parallel to the way we created the first melodic phrase, we now create the percussive phrase. Again, for clarity, I removed from the wavspec file all but what is needed for the percussion generation. That file looks as follows

```
( tutorial
   ( SYNTHINIT )
   ( SYNTHCMDS
      ( env1 ( ENVELOPE ( DURA 2.0 )
               ( COMPONENTS ( HEL 1 0.99 ) ( HEL 49 0.0 ) ) ) )

      ( hitEnv ( NEWPOWER env1 3.0 ) )

      ( noise ( APPEND ( NEWNOISE 0.01 0.99 2.0 ) ( SILENCE 2.0 ) ) )
      ( rnoise ( EXTRACT ( REVERB noise ) 2.001 4.0 ) )
      ( NORMAMP rnoise 0.9 )
      ( cymbal-1 ( REVERB ( REVERB rnoise ) ) )
      ( tmp-hit ( NEWAMODENV cymbal-1 hitEnv 0.8 0.5 ) )
      ( rimshot ( SMOOTH tmp-hit 0.0001 ) )

      ( STOREPERCSEQ percs
         ( INSTMAP ( cym cymbal-1 ) ( hit rimshot ) )
         ( 0.00 ( hit 2.0 ) )
         ( 0.10 ( hit ) )
         ( 0.20 ( hit ) )
         ( 0.30 ( hit 2.0 ) )
         ( 0.40 ( hit ) )
         ( 0.50 ( hit ) )
         ( 0.60 ( hit 2.0 ) )
         ( 0.70 ( hit ) )
         ( 0.80 ( hit 2.0 ) )
         ( 0.90 ( hit ) )
         ( 1.00 ( hit ) )
         ( 1.10 ( hit 2.0 ) )
         ( 1.20 ( hit ) )
         ( 1.30 ( hit ) )
         ( 1.40 ( hit 2.0 ) )
         ( 1.50 ( hit ) )
         ( 1.60 ( hit 2.0 ) )
         ( 1.70 ( hit ) )
         ( 1.80 ( hit ) )
         ( 2.00 ( hit 2.0 ) )
         ( 2.10 ( hit ) )
         ( 2.20 ( hit ) )
         ( 2.40 ( cym 1.5 ) )
         ( 2.80 ( cym 1.5 ) )
      )
      ( percphrase ( PLAYPERCSEQ percs ( TEMPO 0.533333 ) ) )
      ( SAVE percphrase percphrase.wav )
   )
)
```

Notice that, for the percussive score, we must specify instruments in the `STOREPERCSEQ` command. In place of "notes" like G and C#, we place the name of the instrument to be struck (or its alias, as given in the `INSTMAP` at the top.) The (optional) second parameter is used to increase the loudness of certain strikes.

Here is a view of the resulting wavedata, percphrase:



Notice, again, that the "natural" sustain length of our "cymbal-1" takes its last strike well beyond the 6.0 second rhythmic length of the phrase. It is instructive to view these wavedata images while listening to them.

Before we move on to the last phase of the tutorial, the MIX commands, we are now at the stage where our wavspec files are getting rather large and unwieldy, and this is the perfect opportunity to see how the use of wavlibs, tonelibs, and perclibs can help us to organize our work.

Here is an improved wavspec file that will allow us to produce all of the phrases we have created thus far. Refer to this file as the "controlling wavspec file":

```
( tutorial
   ( SYNTHINIT
      ( LOADWLIB myInstruments.txt )
      ( LOADWLIB myEnvelopes.txt )
      ( LOADTLIB myMelody.txt )
      ( LOADPLIB myPercussion.txt )
   )
   ( SYNTHCMDS

      ( testbass ( PLAYTONESEQ bassline1 guitar1 ( TEMPO 0.533333 )
            ( KEYMOD -60 ) ( KEYENV picEnv 50 ) ( REVERB ) ) )
      ( SAVE testbass testbass.wav )

      ( percphrase ( PLAYPERCSEQ percs ( TEMPO 0.533333 ) ) )
      ( SAVE percphrase percphrase.wav )
   )
)
```

How does this work? The `SYNTHINIT` section now creates 3 internal "libraries" of instructions, a `WLIB` that holds instructions that produce wavedata, a `TLIB` that can hold tone-sequence definitions, and a `PLIB` that can hold percussion-sequence definitions.

When we execute `PLAYTONESEQ`, it "knows" that `baseline1` should be a tone-sequence, and that `guitar1` and `picEnv` should be wavedata, and will look for them by name in the appropriate libs and execute their instructions as needed.

The formats for the files myInstruments.txt, myEnvelopes.txt, myMelody.txt and myPercussion.txt are rather simple, with the latter two being the simplest of all, so I will start with these.

Here is the file "`myMelody.txt`":

```
( myFirstMelody
   ( STORETONESEQ bassline1
      ( 0.00 ( G2 0.30 1.6 ) )
      ( 0.20 ( G2 0.90 ) )
      ( 0.60 ( G2 0.30 1.2 ) )
      ( 0.80 ( F2 0.30 1.6 ) )
      [etc etc etc ...]
      ( 2.40 ( D2 0.60 1.6 ) ( D3 0.60 1.2 ) )
      ( 2.80 ( C2 0.60 1.6 ) ( C3 0.60 1.2 ) )
   )
)
```

You can store as many tone-sequences as you like in a file, just make sure that you give each sequence a unique name (`baseline1, baseline2, leadline1, etc`).

The percussion lib file, "`myPercussion.txt` ", uses the very same format:

```
( myFirstPercStuff
   ( STOREPERCSEQ percs
      ( INSTMAP ( cym cymbal-1 ) ( hit rimshot ) )
      ( 0.00 ( hit 2.0 ) )
      ( 0.10 ( hit ) )
      [etc etc etc ...]
      ( 2.40 ( D2 0.60 1.6 ) ( D3 0.60 1.2 ) )
      ( 2.80 ( C2 0.60 1.6 ) ( C3 0.60 1.2 ) )
   )
)
```

The wavlib file for our envelopes is also straightforward. Here is "myEnvelopes.txt":

```
( myFirstEnvelopes
   ( env1 ( ENVELOPE ( DURA 2.0 )
         ( COMPONENTS ( HEL 1 0.99 ) ( HEL 49 0.0 ) ) ) )
   ( picEnv ( NEWPOWER env1 0.5 ) )
   ( cymEnv ( NEWPOWER env1 2.0 ) )
   ( hitEnv ( NEWPOWER env1 3.0 ) )
)
```

Now, what about a wavlib file to hold the instruments? Wouldn't a wavlib just like the one above work?

The answer is, yes. But there is an improvement we can make. Examine the wavlib files on the following pages.

```
( myFirstInstruments
   ( guitar1 ( FOURIER INTEGERS 7 400.0 0.999 2.0 ) )
   ( violin1c ( CHORUS guitar1 400.0 ) )
   ( noise ( APPEND ( NEWNOISE 0.01 0.99 2.0 ) ( SILENCE 2.0 ) ) )
   ( rnoise ( EXTRACT ( REVERB noise ) 2.001 4.0 ) )
   ( NORMAMP rnoise 0.9 )
   ( cymbal-1 ( REVERB ( REVERB rnoise ) ) )
   ( tmp-hit ( NEWAMODENV cymbal-1 hitEnv 0.8 0.5 ) )
   ( rimshot ( SMOOTH tmp-hit 0.0001 ) )
)
```

This file won't quite work. Can you see why? Look at "`cymbal-1`", and notice that it requires "`rnoise`". Find the definition for "`rnoise`" and you will see that it can (and will) be produced by modifying the "`noise`" wavedata, but the `NORMAMP` command will never get called. This is because, unlike commands listed in your controlling `SYNTHCMDS` section, these commands do not get executed sequentially, but only by reference. Therefore, there is no way for WavSynth to know, from the definition of "`rnoise`", that `NORMAMP` needs to be applied.

We could fix this, and make the file "work", by replacing

```
   ( rnoise ( EXTRACT ( REVERB noise ) 2.001 4.0 ) )
   ( NORMAMP rnoise 0.9 )
```

with

```
   ( temp ( EXTRACT ( REVERB noise ) 2.001 4.0 ) )
   ( rnoise ( NEWNORMAMP temp 0.9 ) )
```

But this is not the best solution. A more general issue is that this process ends up leaving a lot of "temporary" wavedata in memory. When you begin creating a very long song, you will generate many temporary phrases of wavedata that get MIXED into the final product, and could then be discarded with the `FREE` command. For instance, if only the first part of your song uses the `cymbal-1` wavedata, you could issue (`FREE cymbal-1`) when it is no longer needed, and save system memory. But this would not free up the space taken by `rnoise`, `temp`, `noise`, etc, even though they may no longer be needed as well, and it would be a hassle to have to remember all of this.

Consider, the only actual "instruments" we want are `guitar1`, `violin1c`, `cymbal-1` and `rimshot`. Is there a way to craft an instrument wavlib that only "names" these elements and takes care of freeing unneeded wavedata?

You already suspect the answer is "yes", correct?

The following wavlib fulfills these requirements:

The improved wavlib file, "myInstruments.txt":

```
( myFirstInstruments
   ( guitar1 ( FOURIER INTEGERS 7 400.0 0.999 2.0 ) )
   ( violin1c ( CHORUS guitar1 400.0 ) )
   ( cymbal-1
      ( SYNTH
         ( noise ( APPEND ( NEWNOISE 0.01 0.99 2.0 ) ( SILENCE 2.0 ) ) )
         ( rnoise ( EXTRACT ( REVERB noise ) 2.001 4.0 ) )
         ( FREE noise )
         ( NORMAMP rnoise 0.9 )
         ( cymbal-1 ( REVERB ( REVERB rnoise ) ) )
         ( FREE rnoise )
         ( RETURN cymbal-1 )
      )
   )
   ( rimshot
      ( SYNTH
         ( tmp-hit ( NEWAMODENV cymbal-1 hitEnv 0.8 0.5 ) )
         ( SMOOTH tmp-hit 0.0001 ) )
         ( RETURN tmp-hit )
      )
   )
)
```

Here, we have introduced two new commands, `SYNTH` and `RETURN`.

In essence, any commands listed within a `( SYNTH cmd cmd ... cmd )` block are executed sequentially, just as if they had appeared in the `SYNTHCMDS` section of your controlling wavspec file.

Notice that when "`cymbal-1`" is called upon, we first produce the wavedata called "`noise`", but only hold on to it long enough to form the related wavedata "`rnoise`". Likewise, the moment after we produce `cymbal-1`, we erase `rnoise` as well.

The `RETURN` command simply tells the `SYNTH` command which piece of wavedata to hand upwards to be given the outer instrument name. In the case of "`rimshot`", we return the wavedata "`tmp-hit`", which gets renamed to "`rimshot`" because of the `RETURN` command.

Armed with our instrument, tone-sequence and percussion-sequence files, we are finally ready to revisit our controlling wavspec file, and issue the final MIX commands to produce the first few bars of our song.

Part 3: MIXING

Recall that our previous wavedata phrases, the tonal phrase called "`testbass`" and the percussive phrase called "`percphrase`", were each of rhythmic length 6.0 seconds. What I would like to do is to play the `testbase` phrase twice in a row, first without the percussion phrase, and then with `percphrase` added. The total length of this piece will thus be 12.0 seconds, so we want to create a 12 second piece of "silence" into which we will mix the various phrases as needed.

Here is our final wavspec:

```
( tutorial
   ( SYNTHINIT
      ( LOADWLIB myInstruments.txt )
      ( LOADWLIB myEnvelopes.txt )
      ( LOADTLIB myMelody.txt )
      ( LOADPLIB myPercussion.txt )
   )
   ( SYNTHCMDS

      ( testbass ( PLAYTONESEQ bassline1 guitar1 ( TEMPO 0.533333 )
            ( KEYMOD -60 ) ( KEYENV picEnv 50 ) ( REVERB ) ) )

      ( percphrase ( PLAYPERCSEQ percs ( TEMPO 0.533333 ) ) )

      ( mySong ( SILENCE 12.0 ) )

      ( SELFMIX mysong testbass 0.0 0.0 ADD ABS 0.8 )
      ( SELFMIX mysong testbass 6.0 0.0 ADD ABS 0.6 )
      ( SELFMIX mysong percphrase 6.0 0.0 ADD ABS 0.3 )

      ( SAVE mysong mysong.wav )

   )
)
```

Notice that we add "`testbase`" twice, first at an offset of 0.0 seconds (the beginning of mysong) and then at an offset of 6.0 seconds. The second set of "0.0" tells the MIX command to use the entire length of the parts being added. Also, notice that the first occurrence of `testbass` is added at amplitude 0.8, but the second one with only 0.6 amplitude. This is because the second copy will be mixed with the `percphrase` (at amplitude 0.3) and make that section about as loud as the first.

The "tutorial" folder in the WavSynth package contains a slightly extended version of this tutorial (including a second voice for the violin), all necessary files, including the DOS batch script needed to produce this song and many of the individual sounds and phrases. You might wish to use the contents of that folder as a template for other song-folders, and feel free to modify and experiment at will.

Enjoy!

4 WAV Synthesis Commands

Overview of SYNTHCMDS

The WAV synthesis commands have been divided into 10 major categories

The **File I/O commands** support the loading of pre-existing WAV files, either to be modified themselves or to be mixed with or otherwise modulate the modification of other waveform data, and of course the saving to file of intermediate or finished waveforms, that they may be played through WAV players or examined with related tools.

The **WAV Creation and Destruction** commands support the creation of new waveform data "from scratch", and the elimination from system memory of waveform data that is no longer needed at certain points in an extended process. Commands for generating simple waveforms, Fourier-defined waveforms, piece-wise envelopes, and various colors of noise are supplied.

The **WAV Range Selection and Simple Editing** section provides easy commands for duplicating, repeating, extracting, appending, reversing, and similar operations upon wavedata that do not generally effect the waveshape of the edited waveforms.

The **WAV Transition and Mixing** commands provide functions for blending and layering sounds together, including variable transition, cycling, and stereo panning.

The **WAV Amplitude Adjustment and Modulation** commands provide functions for adjusting wavedata volume and modulating the amplitude in various ways, in particular, the application of envelopes for creating "struck" instrument sounds. Although these commands only affect the "vertical" component of the wavedata, certain amplitude modulation functions (for instance, high frequency `TREMELO`) can generate new timbral sounds and tones.

The **WAV Shape Adjustment and Modulation** commands provide a rich set of functions that affect the shape, and hence the timbre of wavedata, while not affecting the fundamental wavelengths involved.

The **WAV Frequency Adjustment and Modulation** commands can be used to raise or lower the pitch of tonal wavedata, and apply arbitrary frequency modulations in crafting new timbral sounds.

The **Special Effects** commands provide various parameterized `REVERB`, `CHORUS`, and several more curious transformations to wavedata.

The **Melodic Sequence Definition and Application** commands allow tonic melodies to be defined, and applied to one or more waveforms that act as "instruments".

The **Percussion Sequence Definition and Application** commands allow percussion arrangements to be defined, and applied to ensembles of percussion "instruments".

On the following pages you will find the format specifications for each command. Terms that are given in all lowercase (e.g., "tmp1" or "file.wav") imply that the user may write any string token in that position. Terms written in all uppercase (e.g., "`AMODW`" or "`SIN`") are specific commands or function parameters that must be used exactly as specified. Where the user is required to supply an integer, the specification will be in lowercase but preceded by "`I_`", (e.g., `I_count`). Where a real number is allowed, the parameter will be in lowercase but preceded by "`R_`" (e.g., `R_duration`). Where the user must supply one out of several predefined string-token parameters, the term will be specified in lowercase but preceded by "`S_`" (e.g. `S_shape`).

Examples provided should make these representations clear.

4.1 File I/O Commands

4.1.1 LOAD

Format: `( tmpname ( LOAD filename.wav ) )`

Load a preexisting WAV file and make its wavedata available under the wavedata tagname "`tmpname`".

Example: `( weirdSound ( LOAD my\sounds\weird1.wav ) )`

4.1.2 SAVE

Format: `( SAVE tmpname filename.wav )`
Or: `( SAVE tmpname filename.wav mods )`

Save the wavedata currently referenced by the tagname "`tmpname`" and save it to a file in standard WAV format. The second form allows you to specify whether you want the file saved in mono or stereo, in bytes-per-sample of 1 or 2, and at a sample rate between 2000 and 48000 samples per second. The mods are

`( CHAN value )`	value must be 1 for mono, 2 for stereo
`( BPSM value )`	value must be 1 or 2
`( RATE value )`	value must be in range [2000,48000]
`( INFO [tags] )`	(See Appendix B, Embedded Text Information)

Default values are (`CHAN 2 ), ( BPSM 2 ), ( RATE 44100 ).`

Example 1: `( SAVE coolSound my\sounds\coolThing.wav )`
Example 2: `( SAVE wav1 mine.wav ( CHAN 1) ( RATE 8000 ) )`

IMPORTANT! Sorry about this, but NO part of a filename OR path can contain any whitespace. You cannot use (`LOAD/SAVE tmp my\other sounds\cool Thing.wav)` and you CANNOT override this by using quotes.

4.2 Wave Data Creation/Destruction Commands

4.2.1 FREE

Format: `( FREE tmpname [tmpname ...] )`

Disposes of the wavedata currently referenced by the tagname "tmpname" and removes the tagname "tmpname" from the array of in-memory datastores. The tagname can now be assigned new data in subsequent commands, if desired.

4.2.2 SILENCE

Format: `( tmp ( SILENCE R_dura ) )`

Returns a new "blank" segment of two-channel, 44,100 samples/sec wavedata. Of length R_dura in seconds. This command is most useful when one has created several "bars" of a composition, and wants to layer or sequence them into a larger wavedata using the SELFMIX command.

4.2.3 SIMPFORM

Format: `( tmp ( SIMPFORM S_form R_freq R_amp R_dura ) )`

Generate a waveform of the type `S_form`, with the specified frequency, amplitude and duration, and make the wavedata available by the tagname "tmp".

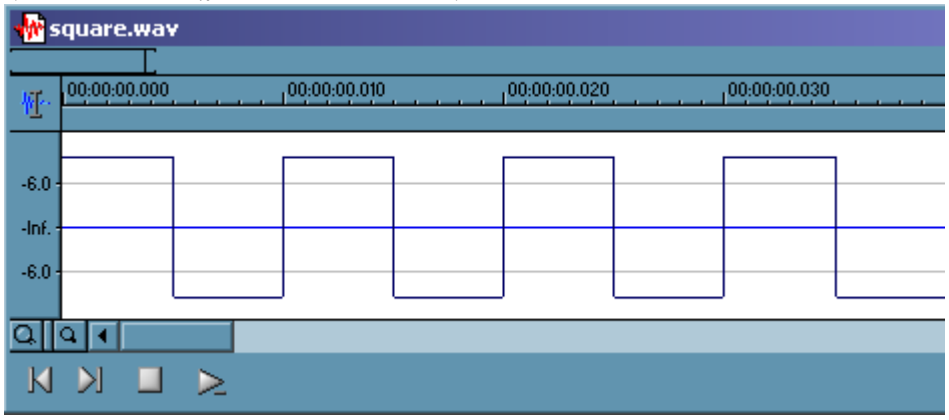
The amplitude is generally given a value between 0.0 and 1.0, where 1.0 will cause the "loudness" to match your output volume, and 0.5 is about 6 Db lower.

The duration `R_dura` is in seconds.

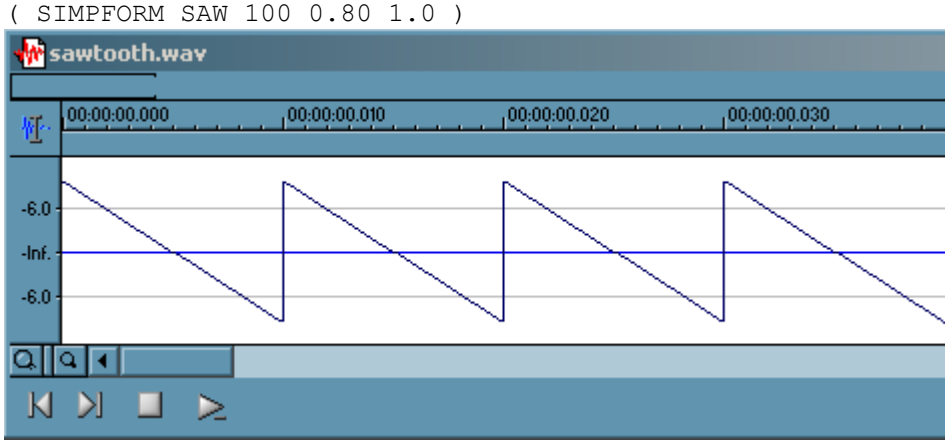
The waveform type must be one of the values `SIN, COS, SQR, SAW, SW2, ELI.`

The following graphics illustrate several examples. In each case, I have generated 1.0 seconds worth of a 100 Hz tone at amplitude 0.80 of maximum. Each graphic displays the first 0.04 seconds of the generated wave data.

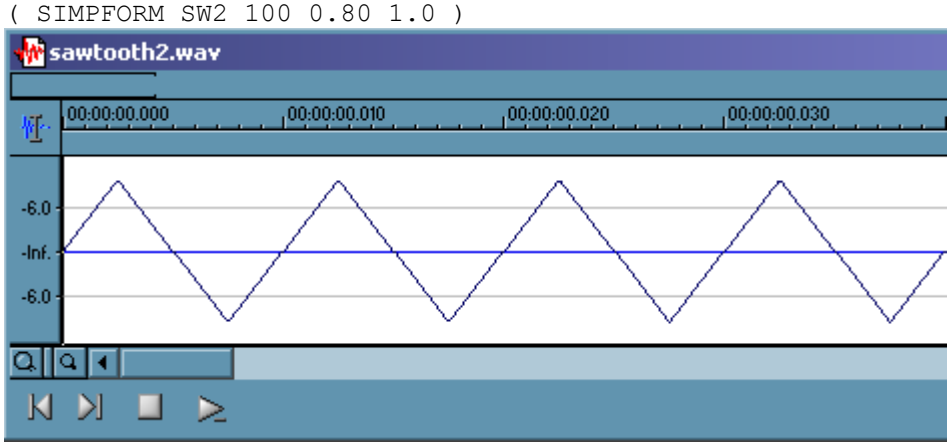
`( SIMPFORM SQR 100 0.80 1.0 )`



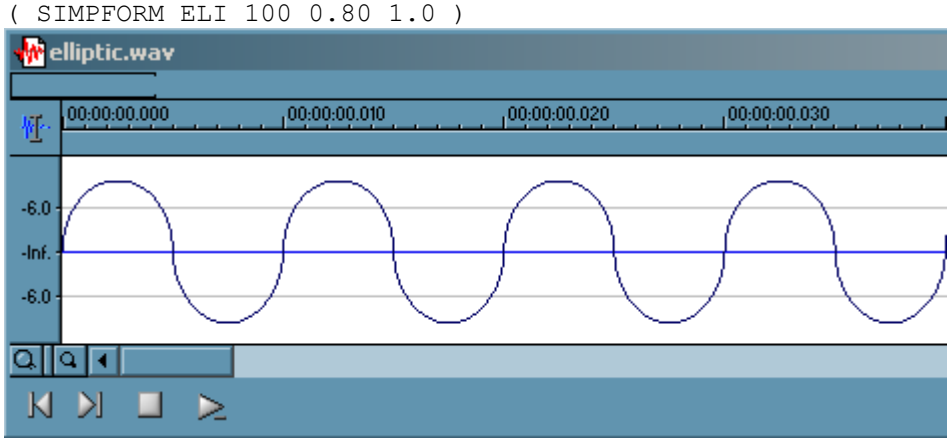
(SIMPFORM SAW 100 0.80 1.0)



(SIMPFORM SW2 100 0.80 1.0)



(SIMPFORM ELI 100 0.80 1.0)

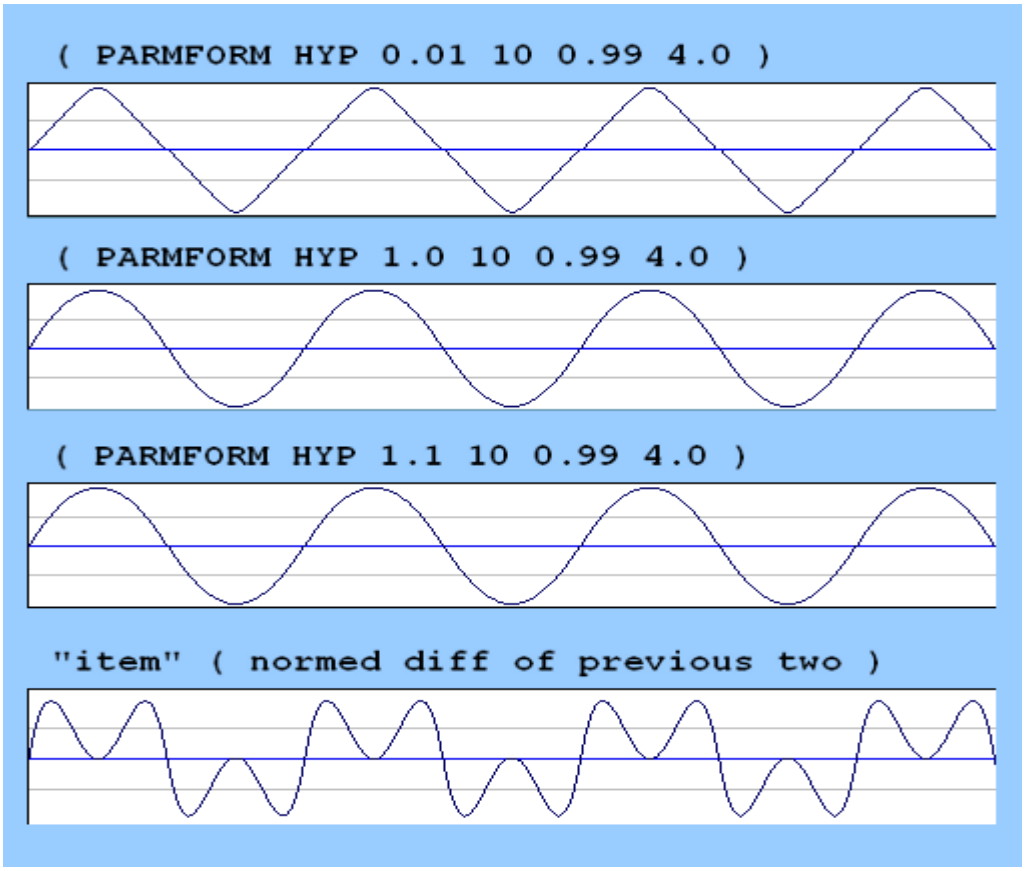


4.2.4 PARMFORM

Format: `( tmp ( PARMFORM HYP R_parm R_freq R_amp R_dura ) )`

In contrast to `SIMPFORM`, where the selection of waveshapes are entirely fixed by the supplied `S_form` (and can only vary in frequency and amplitude,) the `PARMFORM` command begins to introduce waveforms that require an additional "degree of shape" parameter, `R_parm`. At present, the only supported shape is `HYP` (hyperbolic), and for this form, the shape parameter R_parm can be any non-negative real value. At `R_parm = 0`, the waveform becomes the form `SW2` of `SIMPFORM`. As `R_parm` increases toward and beyond 1.0, the waveform looks (superficially) more like the `SIN` form. But looks can be deceiving. An interesting family of forms can be produced by exaggerating the difference between two "similar" `PARMFORM` constructs. For example:

```
( item ( PARMFORM HYP 1.0 200 0.9 4.0 ) )
( tmp2 ( PARMFORM HYP 1.1 200 0.9 4.0 ) )
( INVERT tmp2 )
( SELFMIX item tmp2 0 0 ADD ABS 1.0 )
( FREE tmp2 )
( NORMBT item -0.9 0.9 )
```



4.2.5 FOURIER

Format:

```
( FOURIER S_coeffset I_comps R_freq R_amp R_dura )
```

The `FOURIER` command allows you to create countless different waveforms.

The `FOURIER` function takes a set of coefficients (usually whole numbers), called "Fourier Coefficients", up to a maximum of 20 values, and employs them to produce a sum of many different sine and/or cosine waves, with frequencies and amplitudes derived from the supplied coefficients.

Specifically, if the coefficients are { a, b, c, … }, then the waveform sum will be

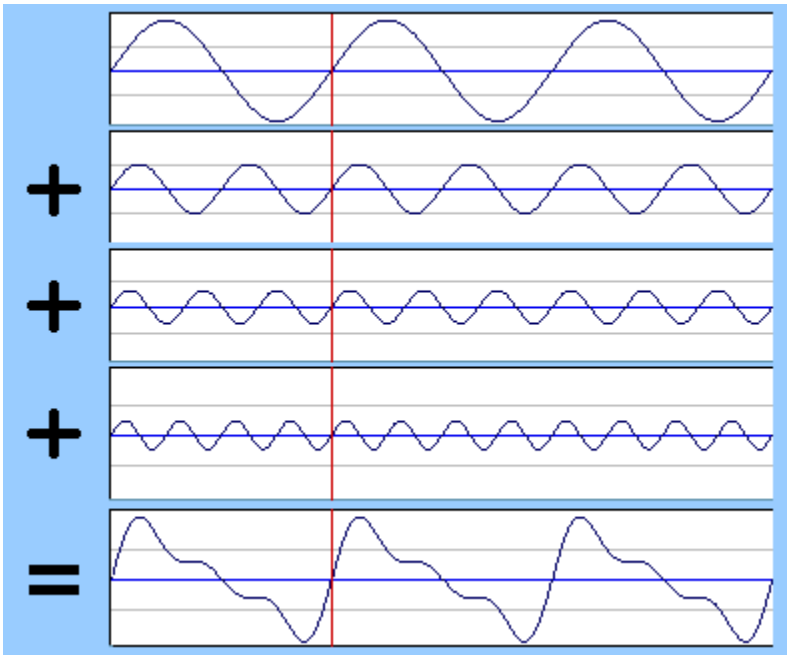
```
(1/a)sin(ax) + (1/b)sin(bx) + (1/c)sin(cx) + ...
```

In general, the first coefficient is 1, and all subsequent coefficients are whole numbers such that each is greater than the previous. The value "`R_freq`" is used as a basis, and corresponds to the coefficient value of 1.

Example: With coefficients { 1, 2, 3, 4 } one is producing the sum

```
sin(x) + (1/2)sin(2x) + (1/3)sin(3x) + (1/4)sin(4x)
```

The net effect of this sum is depicted in the following graphic



Notice that the final waveform has a wavelength corresponding to that of the first sine wave component. Also, although the sum of these components may yield amplitudes that exceed 1.0, the finished waveform is automatically adjusted to the amplitude you specify with the R_amp parameter.

The command I used to produce the final waveform in the previous graphic was

```
( FOURIER INTEGERS 4 100 0.95 1.0 )
```

The `S_coeffset` "`INTEGERS`" is simply the set { 1, 2, 3 , … 20 }, of which I specify using only the first 4 components by the parameter `I_comps`.

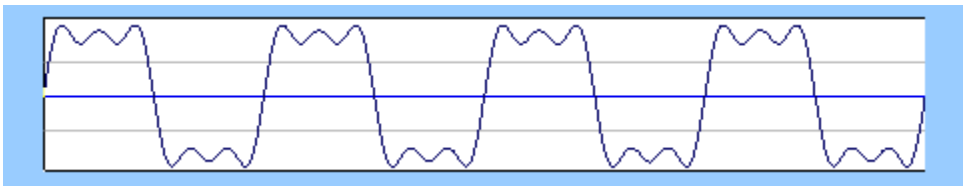
If instead I wanted to use the first 4 (maximum 20) odd numbers, I could use the command

```
( FOURIER ODDS 4 100 0.95 1.0 )
```

This would produce the sum

```
sin(x) + (1/3)sin(3x) + (1/5)sin(5x) + (1/7)sin(7x)
```

and result in the waveform depicted here:



Including INTEGERS and ODDS, WavSynth provides 20 predefined sets of Fourier coefficients that you may reference with the `S_coeffset` parameter.
With each set, you may specify that you wish to use only the first `I_comps` members of that set.

See Appendix D, Predefined Fourier Coefficients for details of these sets.

Importantly, you may define your own sets of Fourier coefficients by creating a file of the form

```
( anyName
   ( myset1 ( coeff[s] ) ( coeff[s] ) … ( coeff[s] ) )
   ( myset2 ( coeff[s] ) ( coeff[s] ) … ( coeff[s] ) )
)
```

and then call them with `( FOURIER myset1 4 100 0.95 1.0 )`, as before.

In order to use your own Fourier coefficients, you must load them using the `LOADFLIB` command, in the `SYNTHINIT` section of your wavspec, as in

```
( LOADFLIB path-to-the-file )
```

The Fourier handler can accept 3 separate ways to specify the coefficients, up to a maximum of 100 components per definition. These are

```
( myset1 f1 f2 … fn )

( myset1 ( cs1 f1 ) ( cs2 f2 ) … ( csn fn ) )

( myset1 ( cs1 cc1 f1 ) ( cs2 cc2 f2 ) … ( csn ccn fn ) )
```

In each case, the values f1, f2, …, fn indicate the relative frequency relationship to the fundamental (f1), and generally satisfy f1 = 1.0, f1 < f2 < … < fn.

With the first and second forms, only "sine" components are employed. When using the first form, no coefficients are supplied, because they are implied to be 1/f for each component f. With the second form, both a coefficient and a frequency multiple is supplied for each component. Finally, with the third form, both coefficients for the "sine" (cs) and "cosine" (cc) components are specified.

Note that supplied coefficients are taken as reciprocals. That is, `( cs1 cc1 f1 )` is taken to mean `(1/cs1)sine(f1*x) + (1/cc1)cosine(f1*x)`.

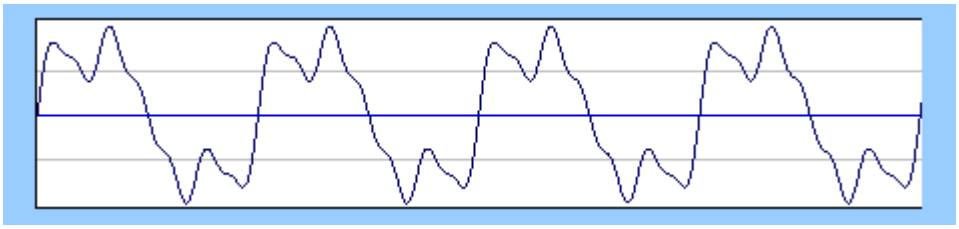
For a trivial example, if you had defined "myset" by

```
( myset 1 3 4 7 11 18 29 47 )
```

then the command

```
( FOURIER myset 8 100 0.95 1.0 )
```

will result in a waveform that looks like this



As with the standard defined Fourier sets, you could apply less than the full 8 components you have defined by specifying a lesser number in the third position.

4.2.6 RANDFOURIER

Format:

```
( RANDFOURIER I_comps I_range R_amp R_basefreq R_dura )
```

The RANDFOURIER function will return a waveform generated by applying random coefficients and frequencies to the Fourier handler. This can be a useful way to explore waveforms in search of peculiar harmonics and timbre.

Each time the function is called, it will ALSO output to the stdout (along with all other run-time messages) the S-expression structure corresponding to the randomly generated values. Thus, if the wavedata produces a sound that you wish to retain, you can simply copy the text from the stdout output capture and place it into your personal file of Fourier coefficients.

The parameter I_comps indicates the number of components to produce. The I_range parameter is used to limit the maximum component frequency, relative to 1. That is, the frequencies f will all satisfy $1 \leq f \leq$ I_range. The minimum accepted value for I_range is 2, the maximum is 10000. The remaining parameters are only employed to produce the wavedata, and have no effect upon the generated coefficients.

4.2.7 QUASIFOURIER

Format:

```
( QUASIFOURIER bwav R_bfreq
         S_coeffset I_comps R_freq R_amp R_dura )
```

The QUASIFOURIER function is an analog to the regular FOURIER function, but where FOURIER assumes that the wave basis are sine (and cosine) components, the QUASIFOURIER function allows you to supply any wavedata as a basis (bwav), specifying its natural frequency with R_bfreq.

Generally, the supplied bwav should represent 1 period of a periodic waveform, and its "natural" frequency should be the reciprocal of its duration in seconds. This wave data is taken analogously to the "sine" function in FOURIER, and (if called upon with a S_coeffset of the third form in FOURIER) will apply a ¼ period left-rotation to the bwav wavedata as analog to the "cosine" component.

All remaining parameters (S_coeffset I_comps R_freq R_amp R_dura) are applied exactly as in the FOURIER command.

4.2.8 HARMOSC

Format:

```
( HARMOSC S_coeffset I_comps R_sfreq R_sdepth R_freq R_amp
      R_dura )
```

The HARMOSC function (short for "Harmonic Oscillator") performs a cyclic mutation upon the parameters supplied to the FOURIER function. The function will only have effect when the Fourier coefficients have the third form, which includes both the sine and cosine coefficients explicitly. Specifically, as the wavedata is generated, it will gradually swap the sine and cosine coefficients, with a frequency and depth given by the R_sfreq and R_sdepth parameters.

This is best illustrated by example.

Suppose S_coeffset defines the set of coefficients

```
( aset ( 1 1 1 ) ( 2 3 2 ) ( 3 5 3 ) … ( n 2n+1 n ) )
```

Under ordinary FOURIER, this would specify that wavedata be generated as the sum of

```
sin(x) + (1/2)sin(2x) + (1/3)sin(3x) + … + (1/n)sin(nx)
+
cos(x) + (1/3)cos(2x) + (1/5)cos(3x) + … + (1/(2n+1))cos(nx).
```

Under the command (HARMOSC aset I_comps 10 0.25 …), the wavedata begins to be generated as the above FOURIER, but as the wavedata progresses, the coefficients (1, 1/2, 1/3 …) being applied to "sine" begin to shift toward those being applied to the "cosine", and vice versa. The frequency with which they swap back and forth is given by the parameter R_sfreq, and the degree they move toward the alternate values is given by R_sdepth, which must be given a value $0.0 \leq$ R_sdepth ≤ 1.0.

For instance, if R_sdepth is given as 0.25, then the second sine-coefficient (initially, 1/2) will shift at most 25% of the way toward (1/3) before moving back to 1/2. Likewise, the second cosine-coefficient (initially 1/3) will shift at most 25% of the way toward (1/2) before returning to 1/3.

The value R_sfreq can be any positive real frequency, and (thus) can be higher than, lower than, or equal to the base frequency R_freq of the generated wavedata. At very low frequency, the timbre will audibly sweep between the two extremes. At high frequencies, the result is an altogether new timbre.

4.2.9 NEWNOISE

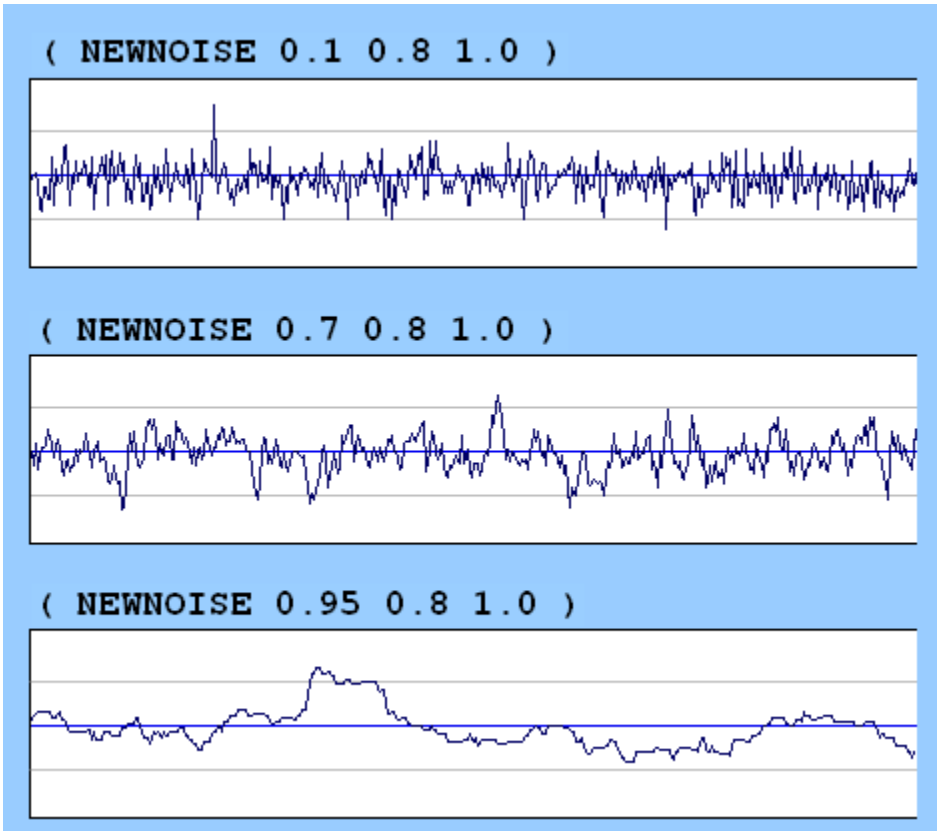
Format: `( NEWNOISE R_color R_amp R_dura )`

This function will produce random noise, tending from Gaussian white noise, through pink noise (1/f noise) toward Brown noise, at the specified amplitude and duration.

The value for `R_color` must satisfy `0 ≤ R_color < 1.0`.

At `R_color` near 0.0, Gaussian white noise is produced. The sequence of sample points in the wavedata are uncorrelated to previous values. At `R_color` near 1.0, the sample points are highly correlated to previous values, tending most often to remain "near" the last few values, resulting in an analog to Brownian motion.

The graphic below depicts 1/100th of a seconds worth of each type of noise.



Noise can be added, at very small amplitudes (under 0.05) to add breathiness to other waveforms, but may also be used as a basis waveform to be "disciplined" by various modulations, smoothings, etc, and may even be used to randomly modulate others. Noise is very useful in crafting certain "natural" sound effects such as the crashing of waves on a beach, the sound of rainfall on a rooftop, the sound of wind, etc.

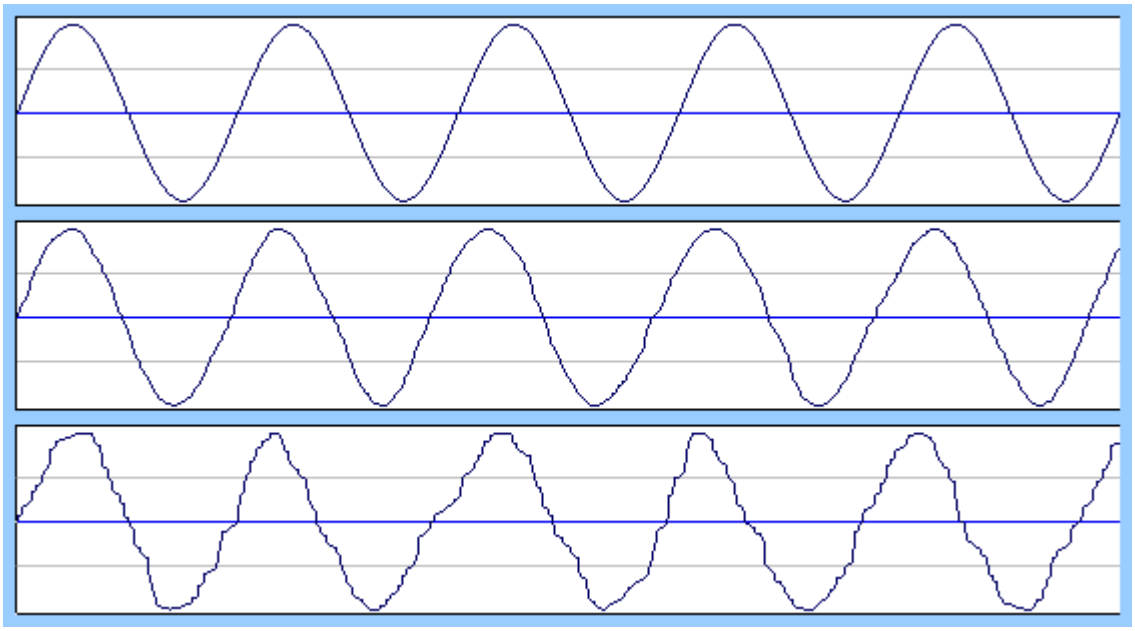
4.2.10 ATIMBRAL

Format: `( ATIMBRAL R_intens R_freq R_amp R_dura )`

The ATIMBRAL command will return a sinusoidal-like waveform, with the specified frequency, amplitude and duration. However, the waveform is temporally distorted to a degree specified by the `R_intens` parameter. In effect, the wave is an ordinary sine wave that is randomly sped up and slowed down at a high-frequency rate. The value of `R_intens` determines how broadly the rate may vary, and must be a value between 0.0 and 1.0.

The following graphic depicts an ordinary sin wave at 100 Hz, and two different intensities of atimbral distortion. The WavSynth commands used were:

```
( SIMPFORM SIN 100 0.99 1.0 )
( ATIMBRAL 0.7 100 0.99 1.0 )
( ATIMBRAL 0.9 100 0.99 1.0 )
```



Notice how the waveform becomes increasingly "lumpy" as `R_intens` approaches 1.0. In general, intensity values below 0.3 are hardly noticeable, while values above 0.8 tend to lose their tonal quality altogether. These effects are useful to produce an airy or breathy quality to a sound.

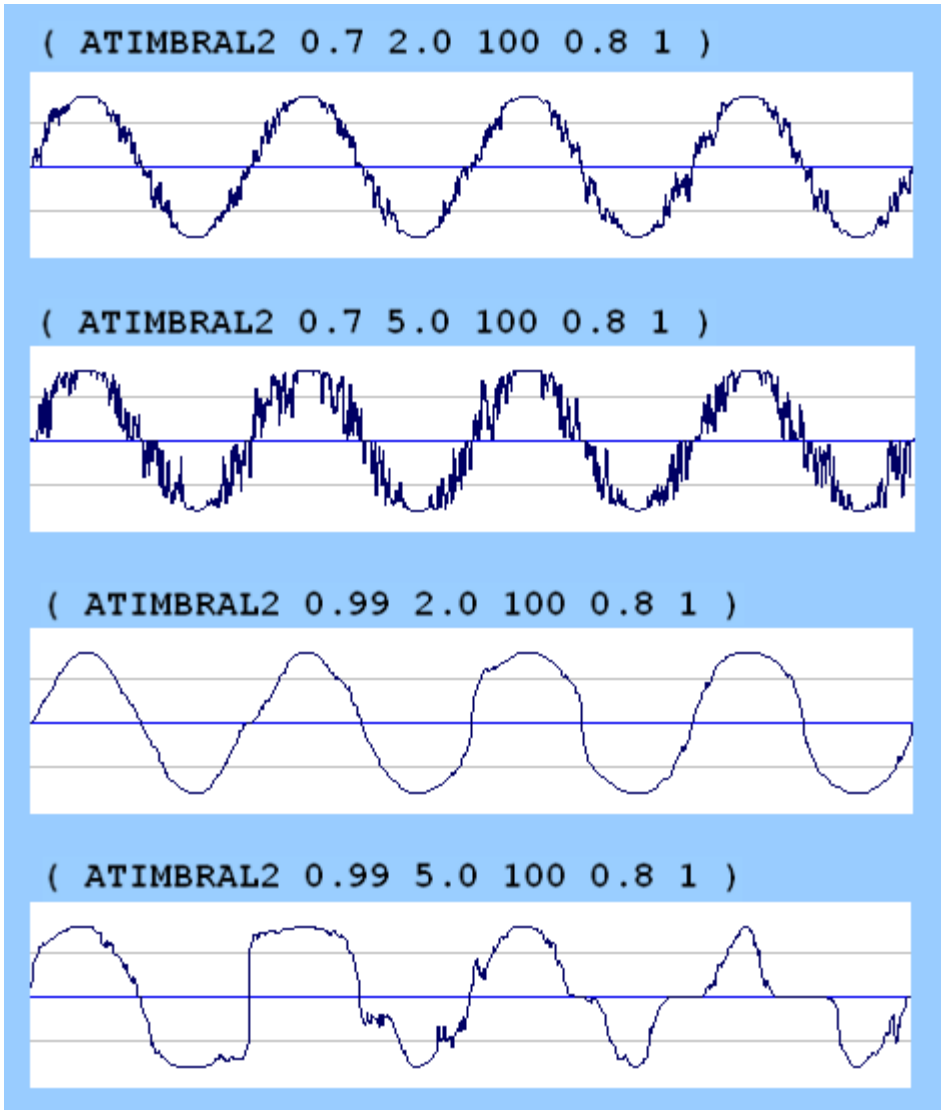
Also, the `ATIMBRAL` function is designed to keep the average wave frequency to the rate specified in the `R_freq` parameter. However, the actual periodicities of the oscillations will vary above and below this value, and this may produce a noticeable "clicking" sound if a short sample is wrapped repeatedly. This can be remedied by applying the `GROUND` command to the wavedata before any repetitions are made.

4.2.11 ATIMBRAL2

Format: `( ATIMBRAL2 R_color R_power R_freq R_amp R_dura )`

The ATIMBRAL2 command provides an alternate scheme for generating an atimbral tone. It operates by first generating color noise of the indicated color, applies the indicated power to the exponentiated noise, and uses the result to power-modulate an ordinary sine wave of the desired frequency. The value of `R_color` must be given between 0.0 and 1.0, and the value for `R_power` may be any value greater than 0.

The following graphic provides samples of `ATIMBRAL2`



Because power-modulation cannot change the sign of its argument, the resulting wav is guaranteed to be positive, negative or zero on the exact periodicity of the basis sine wave.

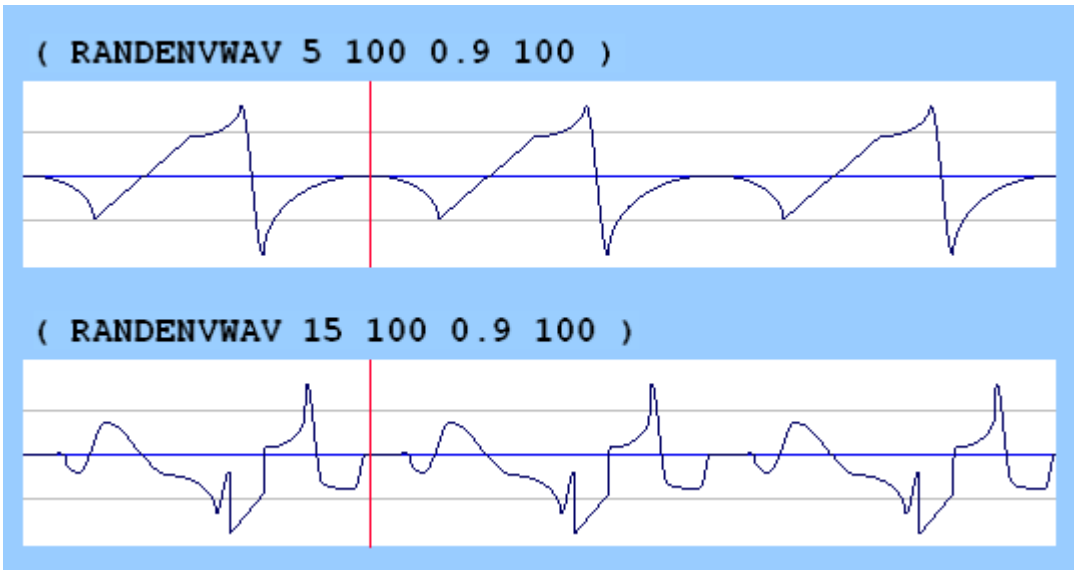
4.2.12 RANDENVWAV

Format: `( RANDENVWAV I_comps R_freq R_amp [I_reps] )`

The `RANDENVWAV` command provides a way to generate a very wild variety of random waveforms, ranging from tonal with distinct timbre through atonal and atimbral. Internally, the `RANDENVWAV` function calls upon the services of the `ENVELOPE` function to generate a random waveform in segments of various shape, length and amplitude. It will form 1 period for the given frequency `R_freq` (with duration 1/freq) using these random segments. If the optional `I_reps` is supplied, this base period is repeated the indicated number of times, resulting in wavedata of duration reps/freq.

The number of components to employ for the base period is specified by `I_comps`, which must be a value between 2 and 500.

The following figure provides a sample for `I_comps = 5`, and `I_comps = 15`.



In each of the above samples, we asked for a 100 hz wavedata (period 0.01 seconds) to be repeated 100 times. Thus, in each case, 1 seconds worth of wavedata is produced. (We are viewing the first 3 cycles, or 0.03 seconds of each above.)

If one specifies a frequency well below the human threshold, such as 5 hz, or 0.1 hz, and specifies a commensurately large number of components like 100 or (up to) 500, the resulting waveform will lack a tonal quality and become an irregular form of noise.

Of course, any `RANDENVWAV` can be used to amplitude, frequency, or power-modulate any other waveform.

4.2.4 ENVELOPE

Format:

```
( ENVELOPE
   ( FREQ R_freq )
   ( DURA R_dura )
   ( INITAMP R_initamp )          [optional, default is 0.0]
   ( REPTYPE S_reptype )          [optional, default is NUL]
   ( REPCOUNT I_repcount )        {optional, default is 1]
   ( COMPONENTS
      ( S_comptype I_portion R_endamp )
      ( S_comptype I_portion R_endamp )
      . . .
   )
)
```

No other WavSynth command provides you more freedom to craft arbitrary waveforms than the ENVELOPE command. Literally, you are defining the point-to-point shape of the waveform, a section at a time. Moreover, you can either stretch your creation "once" over a long duration, to be used as a modulation envelope for another waveform (see section 3, pages 21-22 for examples), or you can make its duration very short and supply REPTYPE and REPCOUNT values, to cause the creation to be variously repeated, resulting in a tonal waveform of peculiar timbre.

The ENVELOPE command is also one of the first for which the major parameters (frequency, duration, initial amplitude, optional reptype and repcount) are listed as subordinate expressions, each of the form (VARNAME value). This structure is required because some of these values are optional, and the WavSynth program must seek them by name in order to know if they have been included.

Also, only one of FREQ or DURA needs to be specified. If both are, then only the first one encountered is used, because they will be reciprocals of one another. The reason is that the value DURA is not *necessarily* the duration of the total construct (it *will* be the total if REPCOUNT is 1, or is omitted.) The DURA value is the time, in seconds, that you wish to allot to ONE PASS over the listed components. If REPCOUNT is greater than 1, meaning that you want the waveform described by the components to be repeated some number of times, the total waveform duration will be DURA*REPCOUNT.

In essence, you can use the ENVELOPE command two distinct ways. Its name suggests that you use it to construct one custom-shaped envelope, perhaps of duration 4.0 seconds, to be used to modulate the amplitude, power, or frequency of some other repeating wave. In such a case, it is more natural to specify (DURA 4.0) than to say (FREQ 0.25). However, if you use this command to create a custom tonal waveform, perhaps a tone of 100 Hz, it might be more natural to specify (FREQ 100) than to say (DURA 0.01). Here, if you wanted a total duration of 2.0 seconds, you would use (REPCOUNT 200).

The COMPONENTS section of the ENVELOPE command is where you specify the various pieces that define your waveform. It looks like:

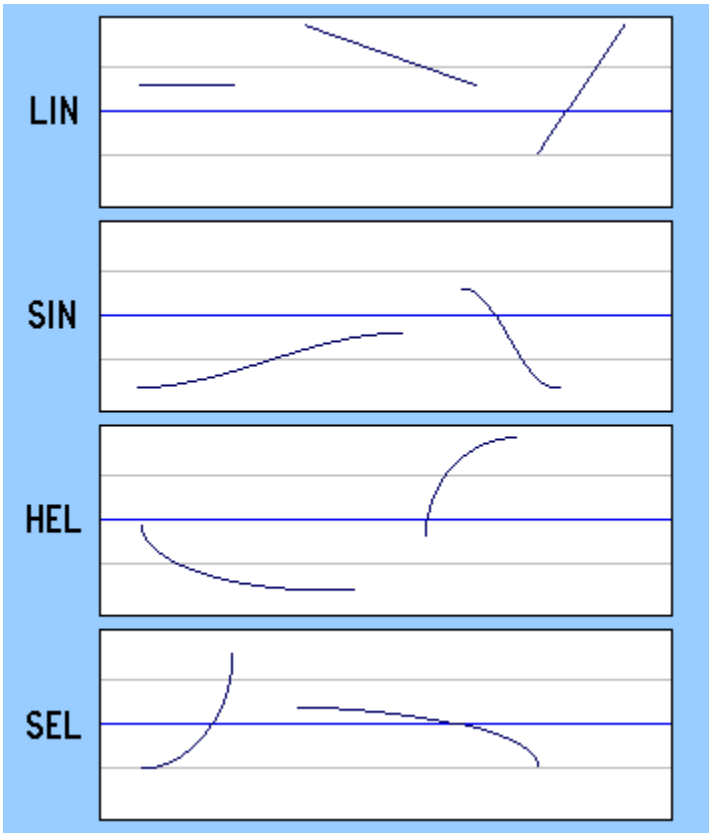
```
( COMPONENTS
   ( S_comptype I_portion R_endamp )
   ( S_comptype I_portion R_endamp )
   . . .
)
```

You must specify at least one component, and can define up to a maximum of 50. Each component is specified by a shape (S_comptype), the portion (I_portion) that it represents of the entire component set, and a terminating amplitude (R_endamp).

The allowable component types are as follows:

LIN	straight segment, rising, falling or horizontal
SIN	sinusoidal transition, rising or falling
HEL	hard-elliptic, rising or falling
SEL	soft-elliptic, rising or falling

The following graphic depicts samples of each shape:



The LIN components are obvious - simply straight segments that transition from one amplitude to another (except for the horizontal case). The SIN components tend to begin horizontally, curve to sweeping upward or downward, and then finish by leveling off to horizontal again. The HEL and SEL components are quadrants of an ellipse. The two that are HEL are called "hard", because as you move from left to right, they are vertically steep at the beginning, producing their greatest change in amplitude at the start, and tend to horizontal at the end. In contrast, the SEL components are "soft" because they begin horizontally, and end vertically.

Envelope Amplitudes

You do not need to specify if a given component is to rise or fall, since WavSynth will figure that out by comparing the `R_endamp` of that component to the `R_endamp` of the previous component (or to the `INITAMP` value, in case of the first component.)

For instance, if the previous component was `( --- --- 0.6 )`, and the next component is given by `( SEL --- 0.8 )` then it will be a rising `SEL`, while `( SEL --- 0.3 )` would be a falling `SEL`.

If you study the example envelope components shown in the previous graphic, you will notice that each begins and ends at an elevation (amplitude) between -1.0 and 1.0, with the blue horizontal line at the middle of each graphic representing 0.0 amplitude. For most envelopes you create, you will want your amplitudes in this range. This is especially true if you intend to use `REPCOUNT` to turn the envelope into a repeating, tonal waveform (although you can always rescale the amplitude of ANY wavedata using one of several `NORM` commands you will learn a bit later.)

The cases where you might use amplitudes beyond [-1.0, 1.0] are those where you may wish to use an envelope as a frequency or power modulator to another waveform. For instance, you might create an envelope with `INITAMP` of 0, and consisting only of the single component `( LIN 1 3.0 )`, with duration 2.0 seconds. This is simply a straight line that begins at amplitude 0, and ends 2 seconds later at amplitude 3. If you were to use it to frequency modulate a 100 Hz tone, the resulting tone would end up starting at 100 Hz, (since 2^0 = 1) and end up at 800 Hz (since 2^3 = 8). The tone would thus "glide" up 3 full octaves over a period of 2 seconds.

But you would not want to try and `SAVE` that envelope as an ordinary WAV file, since it exceeds representability. It would likely "wrap" in weird ways, and you would not recover the original "0.0 to 3.0" linearity of the data upon any reload of that WAV file. (You can, of course, always `SAVE` the text specification of that envelope in your wavspec files or wavlibs, which is another advantage to WavSynth text specifications in addition to file size.)

Envelope Component Portions

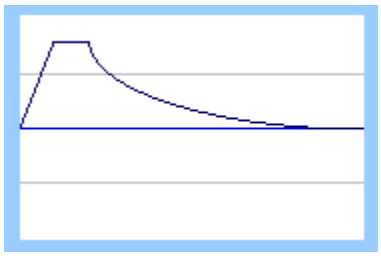
The relative durations allotted to each component is determined by that component's `I_portion` value. Explicitly, WavSynth will add together the `I_portion` values of all of the envelope components, and scale that sum to the to the total envelope duration `R_dura`. Then, each individual component is given a duration corresponding to the contribution their particular `I_portion` makes to the total.

An example will serve us here:

Let us form an envelop of 3 components. The `INITAMP` will be 0, `DURA` will be 1 second, and the three components will be

```
( LIN 1 0.8 )
( LIN 1 0.8 )
( HEL 8 0.0 )
```

The resulting envelope looks like this:



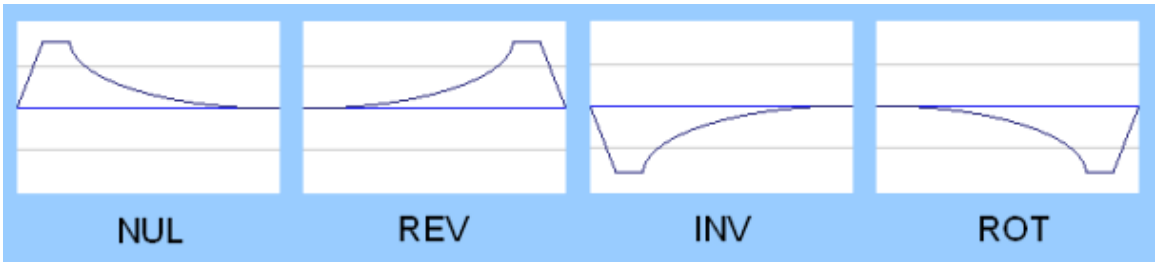
The time-wise (horizontal) length of the three pieces will be 0.1, 0.1 and 0.8 seconds, because I gave the `I_portions` as 1, 1, and 8, which sum to 10. Thus the first component must get 1/10 of `DURA`, as must the second, while the last component must get 8/10 `DURA`. It would have made no difference if I have given the I_portions as 2, 2, 16 or even as 50, 50, 400. Only the relative proportions matter.

This envelope would be natural as an amplitude envelope, and might correspond to the way a stringed instrument would sound when a string is given a quick jab with a bow; the loudness rises quickly as the bow begins to rub the string, sustains for a short moment at that loudness as the bow completes its strike, and then the sound rapidly fades away as the bow leaves the string.

I purposely selected to form this shape, because it is easily distinguished from its reverse, its inverse, and its rotation. These will help to illustrate the other major feature of the `ENVELOPE` command, using `REPTYPE` and `REPCOUNT` to produce a tonal waveform.

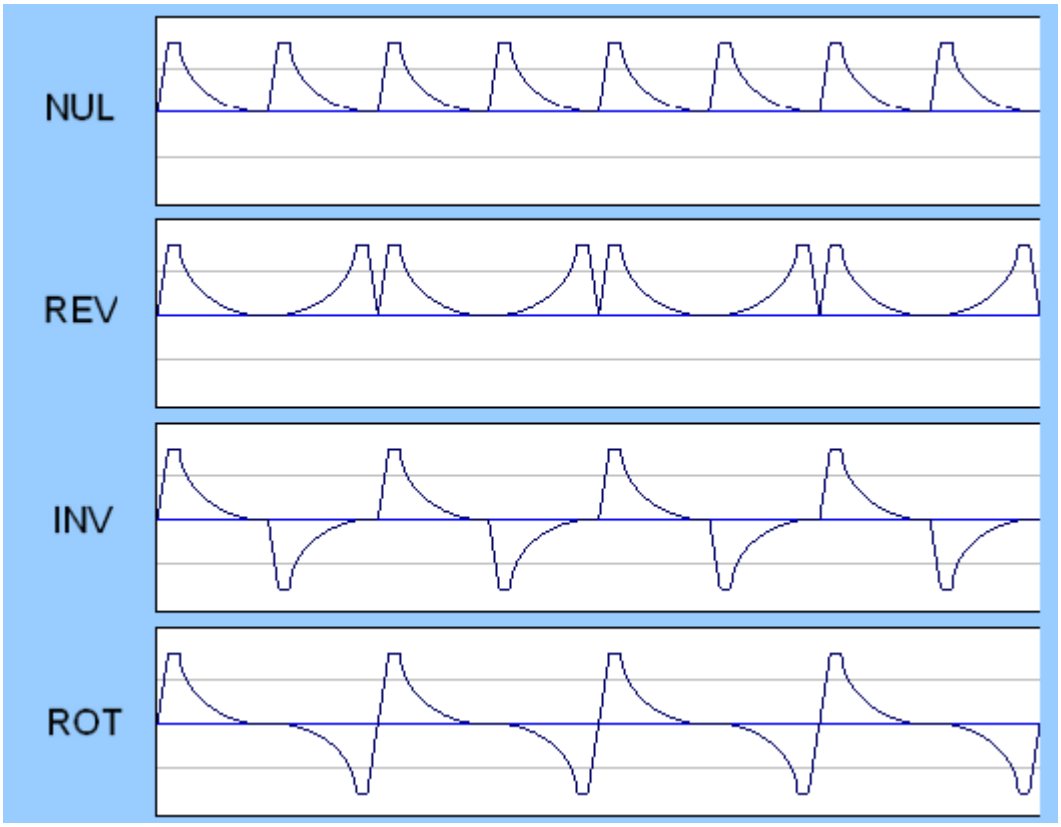
Envelope Reptypes and Repcount

The following graphic depicts the envelope produced in the preceding section, as well as the reverse (REV), the invert (INV), and the rotation (ROT) of that envelope.



The REPTYPE keyword NUL indicates that there is to be no transformation of the defined waveform. The remaining reptypes will affect every *other* repetition of the waveform, when REPCOUNT is given a value greater than 1.

Previously, we defined this envelop to have a DURA of 1.0 seconds, useful as a sweeping modulator of other waveforms. Suppose we instead define the DURA to be 0.01 seconds, corresponding to a repeat frequency of 100 Hz. We can now employ any of the four REPTYPES (NUL, REV, INV, ROT), and produce the following waveforms.



The following wavspec file will actually produce the 4 waveforms shown in the previous graphic, and save the waveforms as WAV files that can be listened to.

```
(
   reptypeExample
   ( SYNTHINIT )
   ( SYNTHCMDS
      ( rep1
         ( ENVELOPE
            ( FREQ 100 ) ( DURA 0.01 ) ( INITAMP 0.0 )
            ( REPTYPE NUL ) ( REPCOUNT 100 )
            ( COMPONENTS ( LIN 1 0.8 ) ( LIN 1 0.8 ) ( HEL 8 0.0 ) )
         )
      )
      ( rep2
         ( ENVELOPE
            ( FREQ 100 ) ( DURA 0.01 ) ( INITAMP 0.0 )
            ( REPTYPE REV ) ( REPCOUNT 100 )
            ( COMPONENTS ( LIN 1 0.8 ) ( LIN 1 0.8 ) ( HEL 8 0.0 ) )
         )
      )
      ( rep3
         ( ENVELOPE
            ( FREQ 100 ) ( DURA 0.01 ) ( INITAMP 0.0 )
            ( REPTYPE INV ) ( REPCOUNT 100 )
            ( COMPONENTS ( LIN 1 0.8 ) ( LIN 1 0.8 ) ( HEL 8 0.0 ) )
         )
      )
      ( rep4
         ( ENVELOPE
            ( FREQ 100 ) ( DURA 0.01 ) ( INITAMP 0.0 )
            ( REPTYPE ROT ) ( REPCOUNT 100 )
            ( COMPONENTS ( LIN 1 0.8 ) ( LIN 1 0.8 ) ( HEL 8 0.0 ) )
         )
      )
      ( SAVE rep1 Rep-NUL.wav )
      ( SAVE rep2 Rep-REV.wav )
      ( SAVE rep3 Rep-INV.wav )
      ( SAVE rep4 Rep-ROT.wav )
   )
)
```

Notice that the ONLY difference between these constructs is the REPTYPE value.

Also, take note that the final WAV files will each have duration 1.0 seconds, even though the DURA value of each envelope definition was only 0.01 seconds. This required using a REPCOUNT of 100.

Finally, because I am using the ENVELOPE command to produce repeating, tonal waveforms, and not simply a one-pass envelope, I chose to name the constructs "rep1", "rep2", etc, instead of "env1", "env2", as they are no longer really envelopes per se.

4.3 Wave Range Selection and Simple Editing Commands

This section explores a wealth of operations you may perform upon wavedata, from the simple to the esoteric. But what actually distinguishes this section from others is that exactly one set of wavedata must be supplied as one of the parameters. Many of these operations come in two separate forms. One form is the **self** form, wherein the desired operations affect the very wavedata you have supplied. The other form is the **new** form, where new wavedata is returned that gives the effect of the modification you requested, but the wavedata you supplied remains unchanged. Where an operation happens to exist in both forms, the form that produces new wavedata will be a command that begins with the prefix NEW. For a detailed exposition on the importance of this distinction in regard to "chained commands", see Appendix C, Chained Commands.

4.3.1 DUP

Format: `( wav2 ( DUP wav1 ) )`

Create new wavedata under the tagname "`wav2`" that is an exact duplicate of the wavedata referenced by tagname "`wav1`".

This command is not really a necessity, but in the case where "wav1" was produced after many commands, or some CPU-intensive operation such as FOURIER synthesis, you may want to try a variety of additional subsequent modifications, and avoid losing the original data. Creating a duplicate of wavedata that may have taken 5 CPU seconds to produce will take only a fraction of a second, and would be preferable to reissuing the originating commands. However (as you will soon see), most of the waveform modification commands return you new wavedata anyway, leaving the input wavedata untouched.

4.3.2 REPEAT

Format: `( wav2 ( REPEAT wav1 I_repcount ) )`
Or: `( wav2 ( REPEAT wav1 I_repcount S_reptype ) )`

The `REPEAT` function always returns a new waveform, leaving the original unchanged. The new wavedata is formed by repeating the input wavedata (`wav1`) as many times as is specified by `I_repcount`.

If `S_reptype` is supplied, it must be one of the predefined tokens `NUL`, `REV`, `INV`, or `ROT`. If `S_reptype` is not supplied, the effect is that same as `S_reptype` of `NUL`. For details of the four different reptypes, see the `ENVELOPE` command (Section 4.2.13, page 47), where reptype and repcount can be supplied as an adjunct to manually crafted wave shapes.

4.3.3 EXTRACT

Format: `( wav2 ( EXTRACT wav1 R_starttime R_finaltime ) )`

The `EXTRACT` function always returns a new waveform, leaving the original unchanged. Given `R_starttime` and `R_finaltime` in seconds, the corresponding portion of `wav1` will be copied into new wavedata named `wav2`.

Note: If either `R_starttime` or `R_finaltime` exceed the bounds of `wav1`, the values are truncated to the existing data. For instance, if wav1 is 10 seconds in duration, and you issue the command

```
( wav2 ( EXTRACT wav1 6.5 9.5 ) )
```

then `wav2` will contain the expected 3 seconds worth of wavedata. But if you issue the command

```
( wav2 ( EXTRACT wav1 6.5 12.5 ) )
```

then `wav2` will contain only the wavedata from 6.5 to 10.0, a duration of 3.5 seconds.

4.3.4 APPEND

Format: `( APPEND wav1 wav2 )`

The `APPEND` function always changes the homed data `wav1` by appending `wav2`. The value for `wav2` maybe either homed data, or homeless data. For instance,

```
( APPEND wav1 ( SILENCE 2.0 ) )
```

will extend the length of `wav1` by adding 2 seconds of silence.

4.3.5 REVERSE and NEWREVERSE

Format: `( REVERSE wav1 )`
Or: `( wav2 ( NEWREVERSE wav1 ) )`

Both of these functions have the effect of producing the timewise reversal of the supplied wavedata. In essence, they play the sound backwards. The difference between these commands is that after you execute `( REVERSE wav1 )`, the wavedata referenced by the name `wav1` is itself reversed. In contrast, if you use `( wav2 ( NEWREVERSE wav1 ) )`, then `wav2` is the reverse of `wav1`, but the data referenced by `wav1` remains unchanged.

Put most simply, the command `( wav2 ( NEWREVERSE wav1 ) )` is just a shortcut for

```
( wav2 ( DUP wav1 ) )
( REVERSE wav2 )
```

4.3.6 INVERT and NEWINVERT

Format: `( INVERT wav1 )`
Or: `( wav2 ( NEWINVERT wav1 ) )`

Both of these functions have the effect of producing the amplitude inversion of the supplied wavedata. In essence, they flip the waveform upside-down. This operation can be useful in preparing a waveform as a modulator of other waveforms, or when one wants to subtract one waveform from another. As in arithmetic, "subtract" is "invert and add".

4.3.7 TIMESHIFT

Format: `( TIMESHIFT wav1 R_secs )`

The `TIMESHIFT` function performs a "time-wise left-shift" of the wavedata by the indicated number of seconds. The result is wavedata that begins `R_secs` into the original data, plays to the end, and then places the original `R_secs` that was skipped at the end.

The duration of the wavedata remains unchanged.

4.4 Wave Transition and Mixing Commands

4.4.1 SELFMIX and NEWMIX

Formats:

```
( SELFMIX wav1 wav2 R_pos R_dura S_mtype S_mmode R_mval )

( wav3 ( NEWMIX wav1 wav2 R_pos R_dura S_mtype S_mmode R_mval ) )
```

These **MIX** commands are used to hand-craft sequences, layer sound effects, etc. Most specifically, they are each used to add sounds on top of one another or to overwrite portions of a sound.

The `SELFMIX` command will modify `wav1` by mixing in `wav2` according to the supplied parameters. The `wav2` data will not be affected. The `wav1` data will be changed.

The `NEWMIX` command has the same effect, except that the mixing occurs to a copy of `wav1` and is returned as new wavedata `wav3`. Neither `wav1` nor `wav2` are affected.

For simplicity in the following discussions, I will just use the term `MIX` to indicate either of these two commands.

The `MIX` commands take five additional parameters, besides the two waveforms that are to be mixed. These parameters (`R_pos R_dura S_mtype S_mmode` and `R_mval`) are best explored individually.

Parameter R_pos:

This parameter indicates, in seconds, the initial position in `wav1` where `wav2` is to be mixed. E.g, you might want to add sound `wav2` **1.5** seconds beyond the start of `wav1`.

Parameter R_dura:

This parameter indicates, in seconds, the amount of `wav2` to be mixed. For instance, suppose `wav1` is 20 seconds long, and `wav2` is 5 seconds long, but you only want 3 seconds worth of `wav2` to be added. Just use `R_dura` of **3**. If you want to add 8.4 seconds worth instead, use `R_dura` of **8.4**.

NOTE: `R_dura` does NOT "squeeze" or "stretch" `wav2` to the indicated duration. If you specify an `R_dura` that is longer than wav2, the `MIX` command will just repeat all or some of `wav2`, sufficient to cover the duration.

As a special case, if you want to add ALL of `wav2` (or whatever amount of it will fit, after the `R_pos` start position), you can specify `R_dura` of **0**, and the `MIX` command will determine the duration of `wav2` automatically.

The last three parameters, `S_mtype S_mmode` and `R_mval`, affect the form and amplitude of the mixing.

Parameter S_mtype: (mixtype)

OVR (overwrite)
ADD (add)

This parameter controls whether `wav2` is to overwrite (fully replace) `wav1` in the region where it is mixed, or whether the two sounds are to be blended in some fashion. Examples after introduction of `S_mmode` will clarify.

Parameter S_mmode: (mixmode)

OVR ABS (overwrite absolute, use amplitude R_mval)
OVR REL (overwrite relative to destination amplitude by factor R_mval)
OVR TRN (overwrite transition, natural amplitude, phase time R_mval)
ADD ABS (add absolute, after adjusting amplitude of wav2 by R_mval)
ADD CEL (add relative to remaining ceiling by fraction R_mval)
ADD REL (add relative to destination amplitude by factor R_mval)
ADD RAT (add in ratio R_mval to destination, adjusting both)

DISCUSSION:

Whenever two waveforms (sets of wavedata) are "added" or "blended" together, the issue of relative "loudness" of the resulting addition become a potential problem. To avoid "clipping" (unintended distortion in saved WAV file data), the final amplitudes of any composition MUST be within the limits [-1.0,1.0]. Although many of the WavSynth operations can produce (and utilize) wavedata where these limits are intentionally exceeded (e.g., `( NORMBT wav 0.5 2.5 )`, for use as a frequency modulator), the final waveforms to be output as WAV files should be treated to `( SETAMP wav 1.0 )`, or to be really safe, `( SETAMP wav 0.99 )`. However, even this broadstroke uniform amplitude adjustment can have unintended consequences.

Suppose that you have layered many waveforms together, and it just so happens that at some point of your composition, many loud sounds occurred at the same time, resulting in an amplitude of 8.5. Applying `( SETAMP wav 0.99 )` will effectively reduce the amplitude of the entire composition so that the loudest sounds are at amplitude 0.99, but this means that the more "quiet" parts of your composition are now "super-quiet".

The various `S_mixmode` parameters allow you to control some of these issues before they become such a problem. Depending upon the parameters used, they take into account both the amplitude of what is being added in, as well as the amplitude of the wav that is receiving the addition, *in the region where the addition is occurring*.

These modes are reviewed below.

In the foregoing discussion, where one is mixing a waveform `wav2` into a waveform `wav1` at a particular location in `wav1`, that region of `wav1` will be termed the "target region" of the destination waveform.

OVR ABS (overwrite absolute, use amplitude R_mval)
ADD ABS (add absolute, after adjusting amplitude of wav2 to R_mval)

Whether overwriting or adding, the **ABS** mixmode will ignore the amplitude of the target region of the destination waveform, and only pre-adjust the amplitude of `wav2` by the supplied value `R_mval` before overwriting or adding it to the target region.

OVR REL (overwrite relative to destination amplitude by factor R_mval)
ADD REL (add relative to destination amplitude by factor R_mval)

The **REL** mixmode will first measure the maximum amplitude of the target region of the destination waveform. The waveform to be added is then rescaled in amplitude to a value given by the product of the destination amplitude and the supplied `R_mval` factor.

OVR TRN (overwrite transition, natural amplitude, phase time R_mval)

When employing the **TRN** mixmode, the `R_mval` is a short transition period, in seconds, during which the destination region is phased-out and the sound to be added is phased-in, (and at the end of the R_dura, phased-out again) at whatever amplitudes they possess.

ADD CEL (add relative to remaining ceiling by fraction R_mval)

If the maximum amplitude of the target region is 0.7, the ceiling is defined to be 1 – 0.7, or 0.3, the amount of amplitude space remaining. The **CEL** mixmode act exactly like the **REL** mixmode, except the `R_mval` is taken as a portion relative to the ceiling of the target region amplitude, rather than to that amplitude itself.

ADD RAT (add in ratio R_mval to destination, adjusting both)

The **RAT** (ratio) mixmode is one of the most useful and convenient. In particular, if the target waveform is everywhere well-behaved (has amplitudes not exceeding [-1,1]), and the supplied `R_mval` is a value in the range [0,1], the resulting mix will also be well-behaved. The overall "before and after" amplitude shape of the entire destination waveform remains unchanged. Instead, the value of `R_mval` is used to specify the portion of the amplitude of a target region to be "handed over" to the wave to be added.

For example, suppose you supply `R_mval` of 0.25, indicating that the wave to be added should represent 25% of the total. If the current amplitude of the destination target region is 0.5, then it will be reduced to 0.375, and the wave to be added will be scaled to 0.125, resulting in a sum amplitude of 0.5, as before.

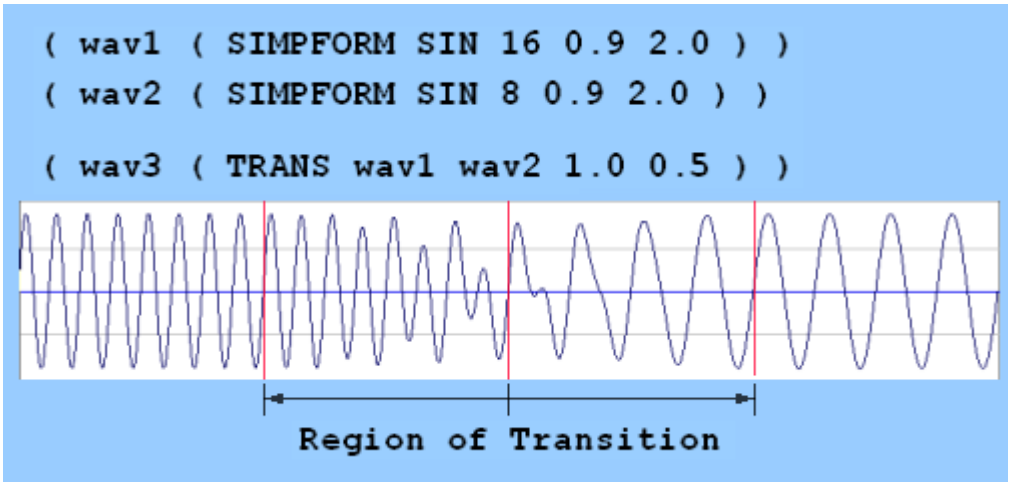
4.4.2 TRANS

Format: `( wav3 ( TRANS wav1 wav2 R_centm R_radtm [R_depth] ) )`

The `TRANS` command returns new wavedata formed by creating a transition from `wav1` to `wav2`, centered about the point `R_centm` seconds into each wav, with a radius of transition `R_radtm` seconds on either side. The result is a `wav3` that is purely `wav1` up to the point `R_centm - R_radtm`, then becomes a mix reaching 50% of each at the point `R_centm`, reaching 100% `wav2` at and beyond `R_centm + R_radtm`, through the duration of the *longer* of the two wavs. If the end of `one` is reached first, it is recycled.

If the optional value `R_depth` is given, `0 ≤ R_depth ≤ 1`, the transition only reaches that indicated portion of mixing toward `wav2`, rather than the default 1.0 (100 %).

The transition curve used is sinusoidal. The following figure depicts a transition between a sin wave of 16 hz and a sin wave of 8 hz, centered at 1.0 seconds with a radius of 0.5 seconds.



A typical use of the `TRANS` function might be to simultaneously fade out one part of a musical composition (or its arrangement or instrumentation) while fading in another.

Note: WavSynth employs `TRANS` internally when applying the `SELFMIX` or `NEWMIX` commands in the mode `OVR TRN` (see section 4.1.1). It can also used as the method for fading between successive notes or chords composed with the `PLAYTONESEQ` command when `MIXTYPE TRN` is specified.

The following functions, `LOCAL` and `CYCLE`, perform operations related to `TRANS`. The first transitions once from wav1 to wav2 and then back to wav1, at a particular location and radius of effect. The second transitions repeatedly between two wavs with a given periodicity.

4.4.3 LOCAL

Format: `( wav3 ( LOCAL wav1 wav2 R_centm R_radtm [R_depth] ) )`

The `LOCAL` command returns new wavedata formed by creating a transition from `wav1` to `wav2`, and then returning to `wav1`, centered about the point in time `R_centm` seconds into each wav, with a radius of transition `R_radtm` seconds on either side. The result is a `wav3` that is purely `wav1` up to the point `R_centm - R_radtm`, then becomes a mix reaching 100% `wav2` at the point `R_centm`, then reverting back to `wav1` at and beyond `R_centm + R_radtm`, through the duration of the *longer* of the two wavs. If the end of `one` is reached first, it is recycled.

If the optional value `R_depth` is given, $0 \leq$ `R_depth` ≤ 1, the transition only reaches that indicated portion of mixing toward `wav2`, rather than the default 1.0 (100 %).

4.4.4 CYCLE

Format: `( wav3 ( CYCLE wav1 wav2 R_freq [R_depth] ) )`

The `CYCLE` command returns new wavedata formed by sinusoidally transitioning back and forth between `wav1` and `wav2` at the indicated frequency. The result will begin at 100% `wav1`, become 100% `wav2` after `(0.5)/R_freq` seconds, and return to 100% `wav2` at `1/R_freq` seconds, and repeat this behavior through the duration of the *longer* of the two wavs. If the end of `one` is reached first, it is recycled.

If the optional value `R_depth` is given, $0 \leq$ `R_depth` ≤ 1, the transition only reaches that indicated portion of mixing toward `wav2`, rather than the default 1.0 (100 %).

4.4.5 CYCBW

Format: `( wav4 ( CYCBW wav1 wav2 wav3 ) )`

The `CYCBW` command creates new wavedata representing a variable mixture of `wav1` and `wav2`, in proportions dictated by the values of the supplied data `wav3` (a copy of which is normalized into the range [0.0,1.0] for this purpose). Thus, the output will reflect `wav1` wherever `wav3` is maximal, and reflect `wav2` where `wav3` is minimal. None of `wav1`, `wav2` or `wav3` are affected by these operations. The length of the result will be the larger of `wav1` or `wav2`, with the shorter (and `wav3`) recycled as needed.

4.4.6 CROSSCHAN

Format: `( wav2 ( CROSSCHAN wav1 R_degree ) )`

The `CROSSCHAN` command applied a constant degree of mixing between the left and right channels of stereo wav data. (If the input `wav1` is monophonic, a copy is returned.) The value of `R_degree` must satisfy `0.0 ≤ R_degree ≤ 1.0`, and will affect the mixing as follows:

If `R_degree = 0.0`, no mixing occurs (exact duplicate of wav1 is returned.)
If `R_degree = 0.5`, each channel receives 50% of the other (monophonic.)
If `R_degree = 1.0`, the left and right channels are effectively swapped.

NOTE: If the left and right channels already contain identical wavedata, the `CROSSCHAN` command will have no effect.

4.4.7 PAN

Format: `( wav2 ( PAN wav1 R_balance ) )`
Or: `( wav2 ( PAN wav1 ( ENV env1 ) ) )`
Or: `( wav2 ( PAN wav1 ( ENV env1 R_min R_max ) ) )`

The `PAN` command is used to adjust the relative volume between the left and right channels of stereo wavedata, in either a constant way, or according to the dictates of a supplied envelope wavedata `env1`.

NOTE: If `wav1` is single-channel wavedata, the result `wav2` will first be made stereo, and then the indicated `PAN` will be applied.

For each of the `PAN` commands, the total volume (left plus right channels combined) is not affected. Only the relative weight between the two is adjusted.

With the first format, the relative volume of the channels is adjusted such that the left channel gets all the weight when `R_balance = -1`, the right channel gets all of the weight when `R_balance = 1`, and both channels get equal weight when `R_balance = 0`.

With the second format, the relative weigh between the channels is made according to the amplitude of the supplied envelope wav, which must take values only between -1 and 1.

In the third format, the supplied envelope `env1` can take arbitrary values, because the `PAN` command will modify a temporary copy of `env1`, renormalizing its existing minimal and maximal values to the indicated values `R_min` and `R_max`, which must satisfy

`-1.0 ≤ R_min ≤ R_max ≤ 1.0`

4.5 Amplitude Adjustment and Modulation Commands

4.5.1 SETAMP and NEWSETAMP

Format: `( SETAMP wav1 R_amp [I_chanspec] )`
Or: `( wav2 ( NEWSETAMP wav1 R_amp [I_chanspec] ) )`

These operations uniformly normalize the amplitude (loudness) to the specified value. The wavedata is NOT shifted vertically to equalize amplitude above or below 0. Thus, wavedata that begins or ends at zero amplitude will remain so. An `R_amp` value of 1.0 produces the maximal loudness, while an `R_amp` value of 0.0 reduces the waveform to a flatline of silence.

By default, `SETAMP` and `NEWSETAMP` will apply the amplitude to both channels (if both exist). You can specify application to only the left or right channel by supplying the optional `I_chanspec` with 1 for left only, 2 for right only, or 3 for both.

4.5.2 NORMAMP and NEWNORMAMP

Format: `( NORMAMP wav1 R_amp [G] )`
Or: `( wav2 ( NEWNORMAMP wav1 R_amp [G] ) )`

These operations uniformly normalize the amplitude (loudness) to the specified value, and in addition shift the waveform vertically so that equal weight of the waveform occurs above and below the zero-amplitude value. An `R_amp` value of 1.0 produces the maximal loudness, while an `R_amp` value of 0.0 reduces the waveform to a flatline of silence.

NOTE: Due to the vertical shift to equalize the positive and negative weighing, it is likely that the endpoints of the data will be shifted AWAY from zero amplitude. This can be remedied by supplying **`G`** as the last parameter, which will apply the `GROUND` effect (see 4.5.4).

4.5.3 NORMBT and NEWNORMBT

Format: `( NORMBT wav1 R_ampmin R_ampmax [G] )`
Or: `( wav2 ( NEWNORMBT wav1 R_ampmin R_ampmax [G] ) )`

This operation identifies the lowest and highest value of the waveform, and then uniformly rescales and vertically shifts the amplitude so that the waveform lies between the two supplied amplitude values `R_ampmin` and `R_ampmax`.

This function is very useful when you wish to prepare a waveform for use as an amplitude or power modulator to another waveform. Your target waveform "`targ`" may have a frequency of 400 Hz, but you wish to modulate the power characteristics of that

data between the values of (say) 3 and 6, on a frequency of 80 Hz. In such a case, you could perform the following operations:

```
( tmp1 ( SIMPFORM SIN 80 1.0 1.0 ) )
( NORMBT tmp1 3.0 6.0 )
( PMODW targ tmp1 1.0 )
```

After the NORMBT command, the wavedata tmp1 is a sine wave that oscillates between 3.0 and 6.0, rather than between -1.0 and 1.0. This wavedata (tmp1) would not "save well" if output as a WAV format file to be listened to, but it is perfectly fine as a power or frequency modulator to more well-behaved data, such as we assume "targ" to be.

NOTE: Due to the vertical shift effected by NORMBT and NEWNORMBT, it is likely that the endpoints of the data will be shifted AWAY from zero amplitude. This can be remedied by supplying **G** as the last parameter, which will apply the GROUND effect (see 4.5.4 below). This can be important if the wavedata being normed is intended to be heard, as opposed to being used for a modulation envelope or other intermediate operations.

4.5.4 GROUND

Format: `( GROUND wav1 [R_dura] )`

The GROUND command can be used to force the ends of a waveform toward 0 amplitude (without affecting the volume.) By default, it will locate the midpoint of the wavedata, and linearly "tilt" the leading half and trailing half of the data as needed to bring its endpoints to 0. The command has no effect if the data is already "grounded".

If the optional R_dura is specified, and is less than the duration of the wavedata, then the "tilting" is applied only to the leading and trailing R_dura/2 portions of the wavedata.

Discussion: Why apply the GROUND command?

In a great many cases, it is perfectly useful to produce and apply wavedata whose endpoints are non-zero, such as when used as modulator envelopes, etc. However, it is often the case that wavedata intended to be "heard", perhaps used in the construction of instrument sounds, or in layering parts of a composition, will fail to begin or end at zero due to the effects of certain wav operations. This is especially a problem when a short section of wavedata is (say) looped using the REPEAT command (or several other commands that "recycle" a piece of wavedata in applying an effect to another wavedata.) If the piece being recycled does not end as it began (usually at zero), there can be a noticeable "clicking" sound produced at the crossing point.

4.5.5 AMODW and NEWAMODW

Format: `( AMODW wav1 wav2 R_amp [I_reps] )`
Or: `( wav3 ( NEWAMODW wav1 wav2 R_amp [I_reps] ) )`

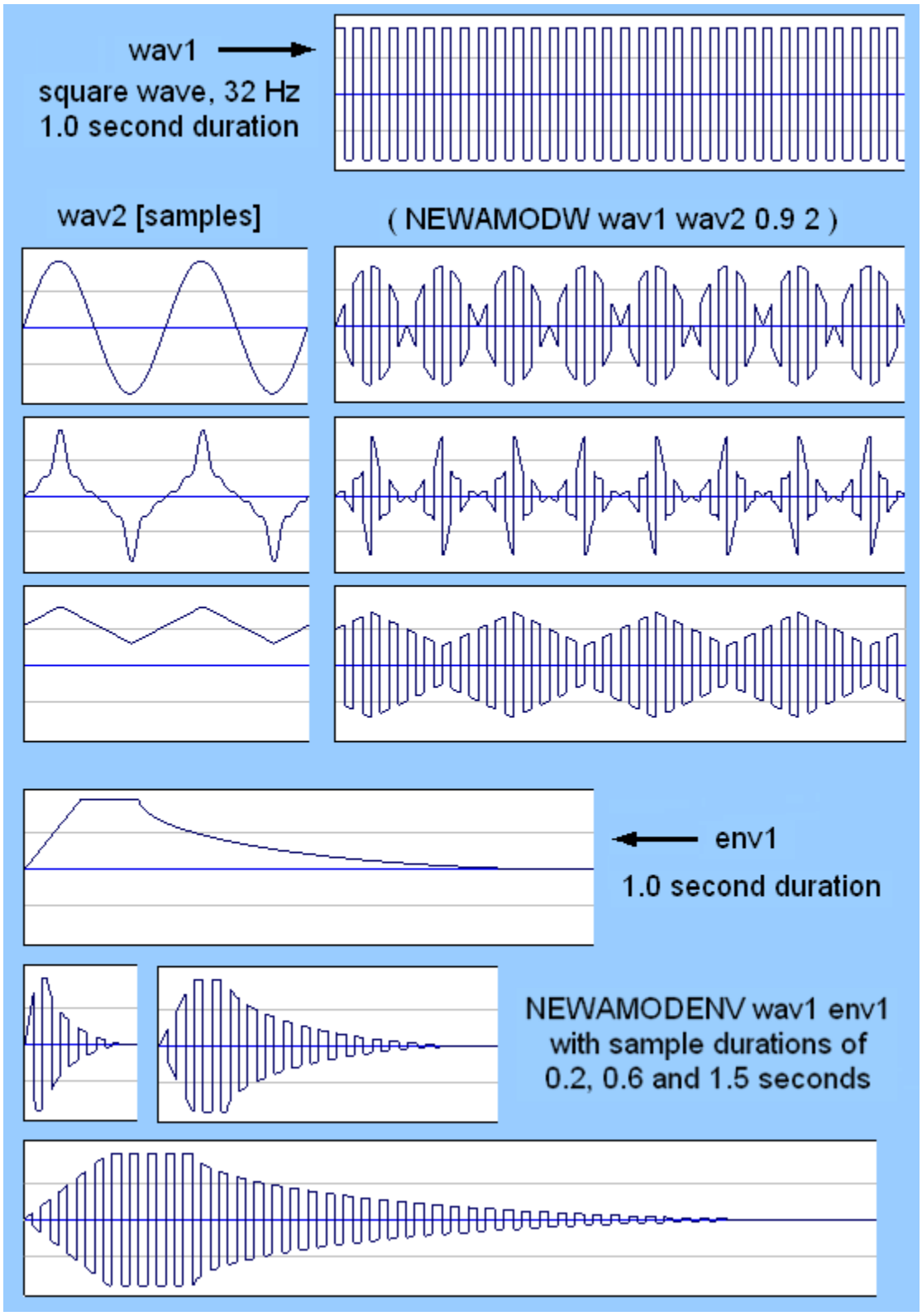
These functions modulate the amplitude of `wav1` according to the values in `wav2`, and scale the final result such that its maximal amplitude is `R_amp`. The resulting waveform will have the same duration as `wav1`, and the modulator `wav2` will be resized so that the desired number of `I_reps` exactly match the length of `wav1`. (If `I_reps` is omitted, it is taken to be 1). We have already seen this command employed where `wav2` was an envelope (see section 3 tutorial, pages 21-22) but any waveform data may be used as an amplitude modulator.

4.5.6 NEWAMODENV

Format: `( wav3 ( NEWAMODENV wav1 wav2 R_amp R_dura ) )`

This function is similar to `NEWAMODW`, but is especially useful when the amplitude modulator, `wav2`, is an envelope. The form `wav2` is used only once, but is stretched in time according to the supplied `R_dura` value. This saves one from having to craft many different envelopes that would differ only in length. The returned wavedata `wav3` will (thus) have the duration `R_dura`. Omitting `R_dura` applies the duration of `wav1`.

The following graphic shows several examples of both `NEWAMODW` and `NEWAMODENV`.


wav1
square wave, 32 Hz
1.0 second duration
wav2 [samples]
(NEWAMODW wav1 wav2 0.9 2)
env1
1.0 second duration
NEWAMODENV wav1 env1
with sample durations of
0.2, 0.6 and 1.5 seconds

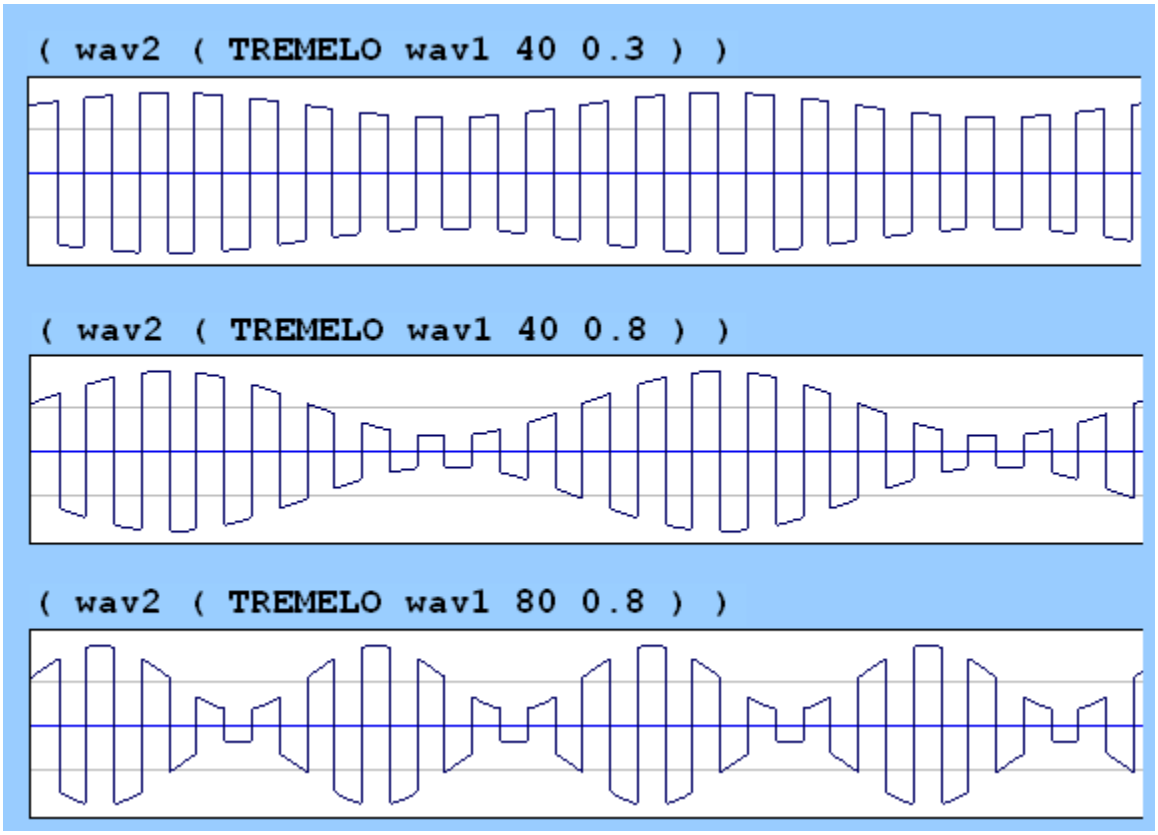
4.5.7 TREMELO

Format: `( wav2 ( TREMELO wav1 R_freq R_ampdepth ) )`

This function will sinusoidally modulate the amplitude of the supplied waveform (`wav1`) at the specified frequency and amplitude depth. For illustration, I chose to begin with a square wave of frequency 400 Hz, using

```
( wav1 ( SIMPFORM SQR 400 0.9 1.0 ) )
```

and then I applied various `TREMELO` settings. The result is depicted below.



If the `TREMELO` frequency `R_freq` is a value below about 20 Hz (too low to be perceived as a tone in itself), the supplied waveform will tend to retain its original timbral quality, but will audibly "flutter" in loudness at the rate specified. If the `TREMELO` frequency is above 20 or 30 Hz, the effect will tend to produce an entirely new timbral quality.

The `TREMELO` function always returns new wavedata.

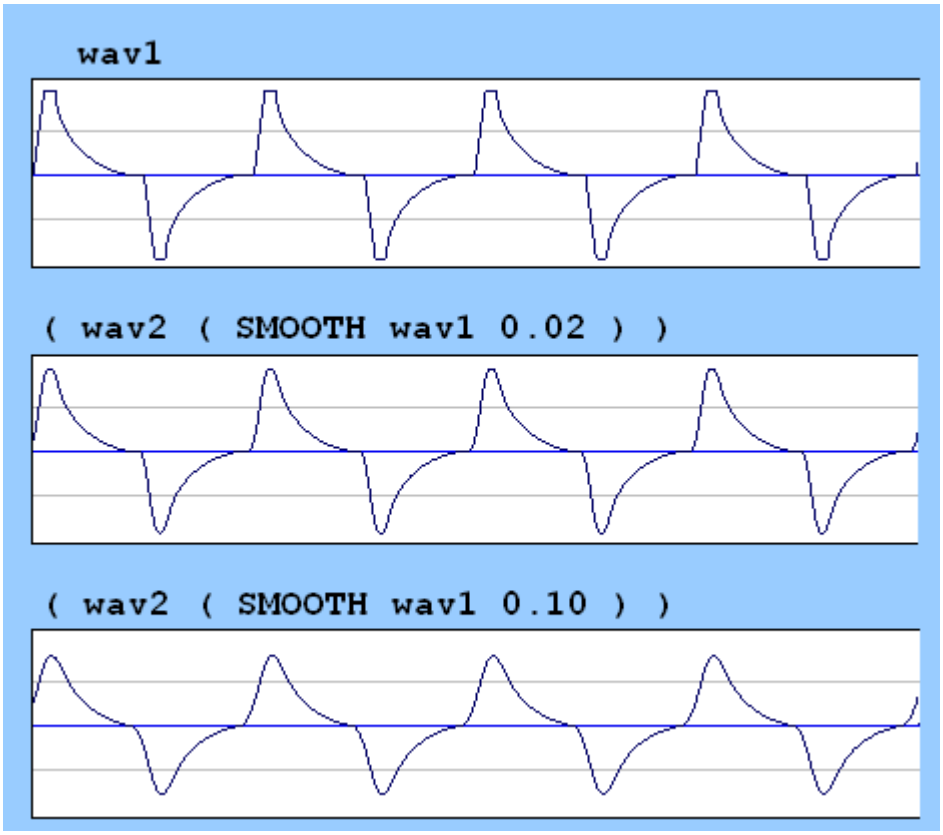
4.6 WaveShape Adjustment and Modulation Commands

4.6.1 SMOOTH

Format: `( wav2 ( SMOOTH wav1 R_radius ) )`

This function performs a Gaussian smoothing of each point of the waveform data with neighboring points, to a radius of `R_radius` seconds. A new waveform is returned, leaving the original unaffected.

The following graphic depicts one of the waveforms (`wav1`) we produced under the `ENVELOPE` command (see Section 4.2.4, page 51), and various degrees of smoothing.



Beware: Smoothing can take substantial time when applied to a waveform of long duration, or when the radius of smoothing is more than a small fraction of a second.

4.6.2 VSMOOTH

Format: `( wav2 ( VSMOOTH wav1 R_size [R_power] ) )`

The VSMOOTH command (short for Variable Smoothing) is actually designed as a form of noise reduction. In particular, regions of large amplitude receive minimal smoothing, and the "quiet" regions of minimal amplitude receive the greatest amount of smoothing. The R_size parameter is applied exactly as in the SMOOTH command (4.6.1). If `R_power` is omitted, 2.0 is used. Otherwise, `R_power > 1.0` will affect the degree of smoothing.

4.6.3 TSMOOTH

Format: `( wav2 ( TSMOOTH wav1 R_halftm R_maxrad ) )`

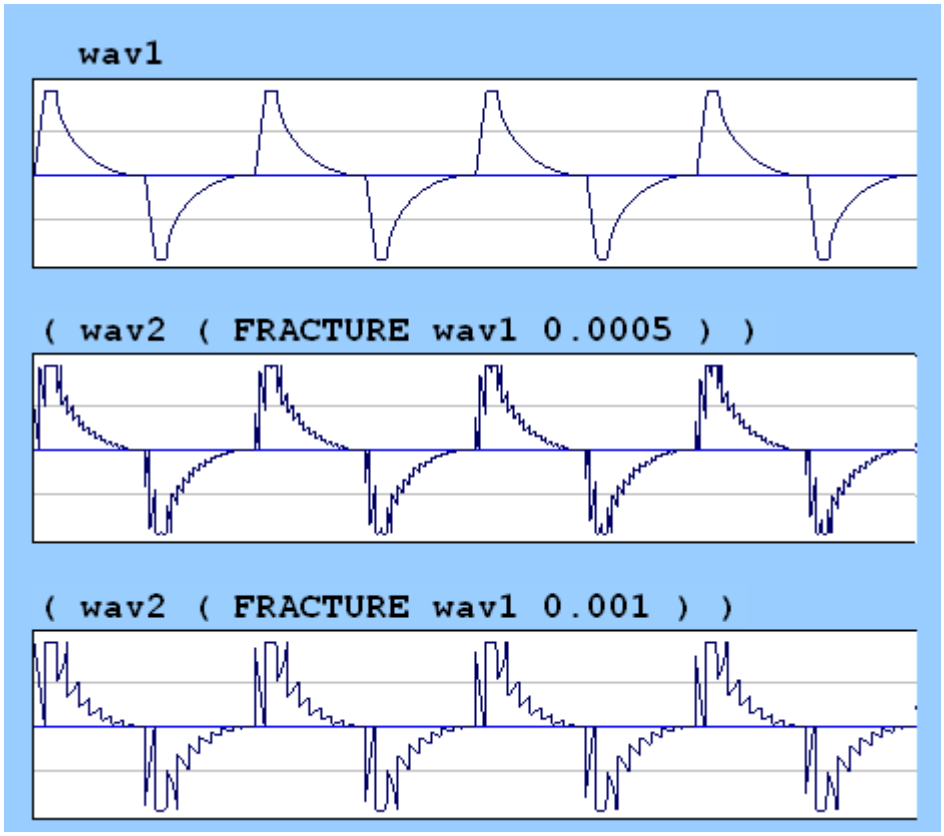
The TSMOOTH command (short for Terminal Smoothing) is actually designed to increasingly filter out high-pitched ringing from strike-decays. The smoothing (sample averaging) has no effect near the strike point, but increasingly adds samples according to the limit for `nsamps = (tm*maxsamps)/(tm+halftm)`,where maxsamps is the maxrad * sample_rate.

4.6.3 FRACTURE

Format: `( wav2 ( FRACTURE wav1 R_size ) )`

This function divides the waveform data into blocks of size `R_size` seconds, and reverses each block. For small block size (less than 0.05 seconds) this can be used to introduce a degree of peculiar noise to the waveform.

The following graphic depicts one of the waveforms (`wav1`) we produced under the ENVELOPE command (see Section 4.2.4, page 51), and various degrees of FRACTURE.

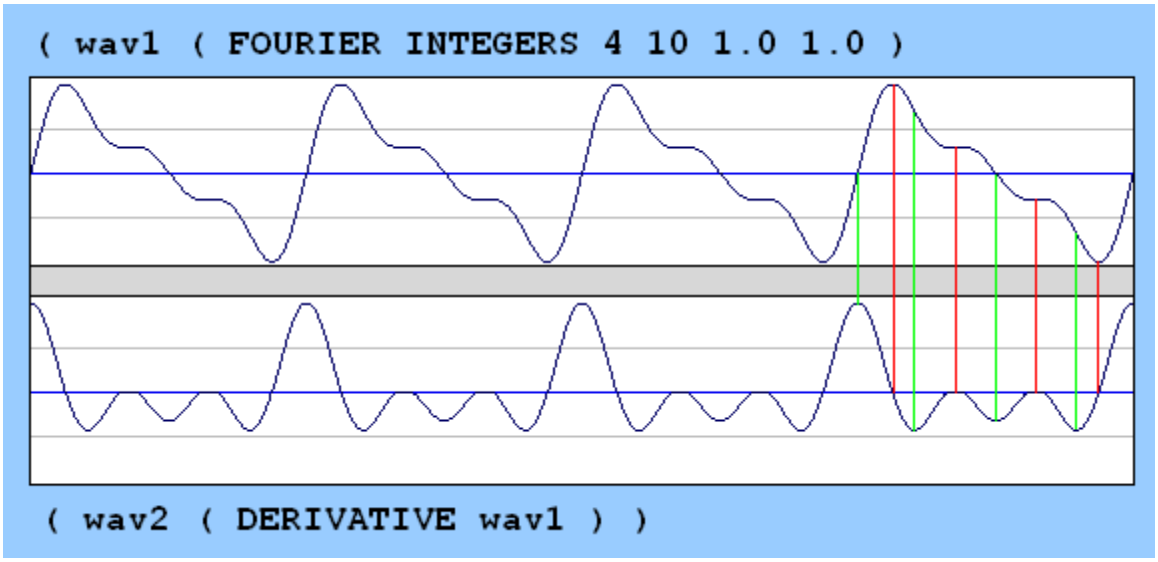

wav1
(wav2 (FRACTURE wav1 0.0005))
(wav2 (FRACTURE wav1 0.001))

4.6.4 DERIVATIVE

Format: `( wav2 ( DERIVATIVE wav1 ) )`

This function returns new wavedata representing the derivative of the supplied wavedata. In short, the derivative reflects the slope, or "steepness" of the input waveform, and scales the resulting values back to the amplitudes of the original.

In the following example, wav1 is produced by FOURIER coefficients (see section 4.2.5) and wav2 is its derivative.



The green lines show how the derivative reaches its most extreme (positive or negative) values where the original had the greatest positive or negative slope. The red lines show how the derivative will have zero values where the original had zero slope.

4.6.5 INTEGRAL

Format: `( wav2 ( INTEGRAL wav1 ) )`

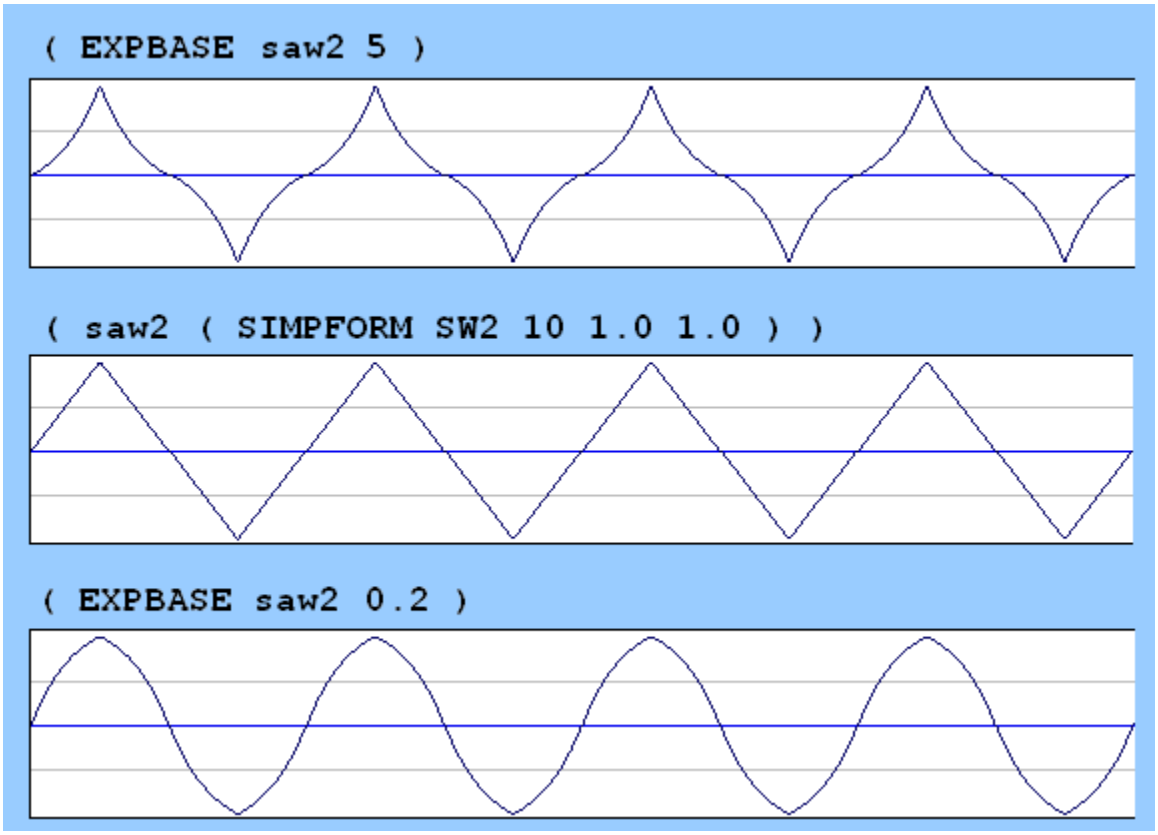
This function, the complement to the `DERIVATIVE` function, produces wavedata whose slopes match the amplitude values of the supplied wavedata `wav1`. In order to attempt to return a waveform that begins and ends at zero (i.e., the amplitude does not stray without bound), a copy of the supplied `wav1` is weight-normalized – it is shifted vertically so that an equal amount of total area lies above and below zero. Nevertheless, some drift can occur due to rounding errors, and one can correct for this by applying the `GROUND` command to the returned wavedata. In general, the `INTEGRAL` function should be applied only to relatively short wavedata samples.

4.6.6 EXPBASE

Format: `( wav2 ( EXPBASE wav1 R_base ) )`
or: `( wav2 ( EXPBASE wav1 R_base I_degree ) )`

These commands raise the supplied value R_base to the power indicated by the wavedata values, rescaling the result back to the amplitude of the original wavedata. As with the POWER function, the positive and negative sections of the waveform are treated independently, thus negative portions remain negative. Any `R_base` greater than 0.0 is allowed. If an integer `I_degree` greater than 1 is supplied, the entire process is repeated `I_degree` times.

The following graphic depicts a symmetric sawtooth wave (`saw2`), and the effect of using it as an exponent to a base of 5 and to a base of 0.2.



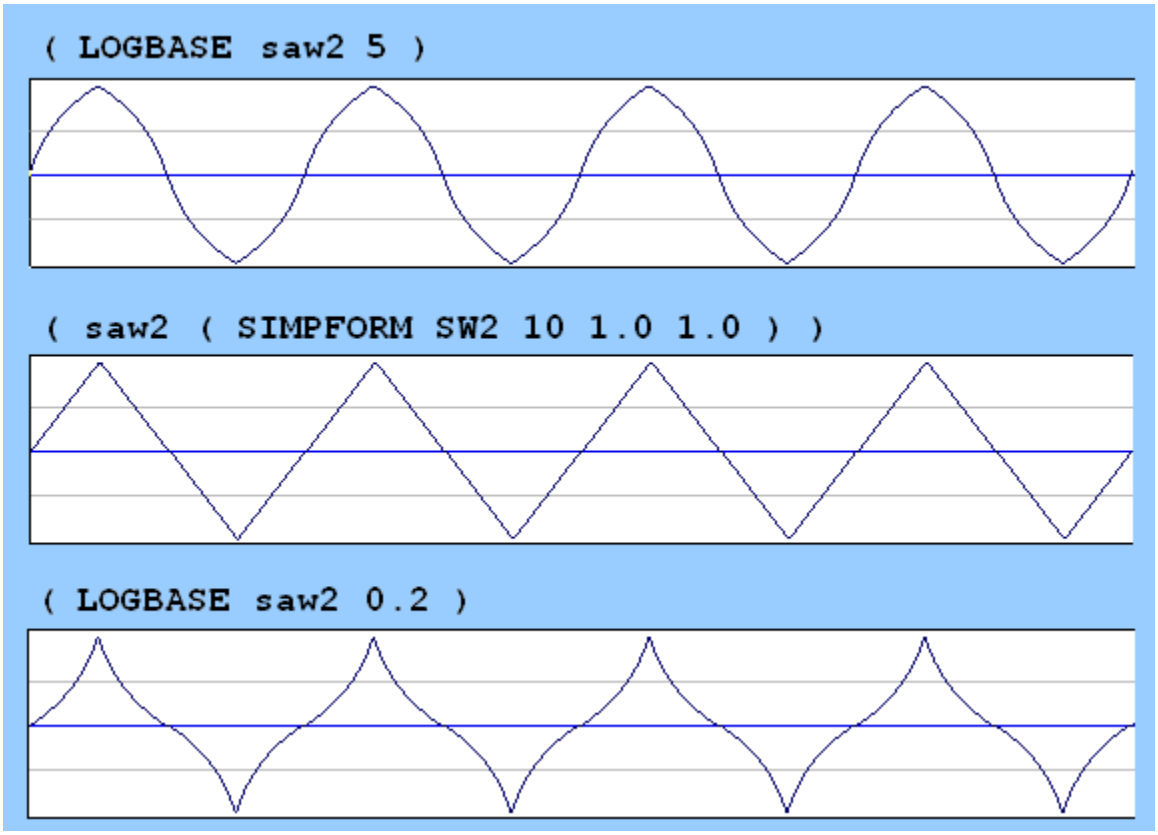
At present, the EXPBASE function always returns new wavedata. It will not affect the values of the supplied wavedata.

4.6.7 LOGBASE

Format: `( wav2 ( LOGBASE wav1 R_base ) )`
or: `( wav2 ( LOGBASE wav1 R_base I_degree ) )`

These commands take the log, base R_base, of the supplied wavedata values, rescaling the result back to the amplitude of the original wavedata. As with the POWER function, the positive and negative sections of the waveform are treated independently, thus negative portions remain negative. Any `R_base` greater than 0.0 is allowed. If an integer `I_degree` greater than 1 is supplied, the entire process is repeated `I_degree` times.

The following graphic depicts a symmetric sawtooth wave (`saw2`), and the effect of taking the log base 5, and the log base 0.2, of the `saw2` wavedata.



As with `EXPBASE`, the `LOGBASE` function always produces new wavedata and leaves the supplied wave data untouched.

NOTE: Although (for instance) `EXPBASE` with `R_base` 5, and `LOGBASE` with `R_base` 0.2 look similar, this is not the case, and the waveforms will exhibit noticeable tonal differences.

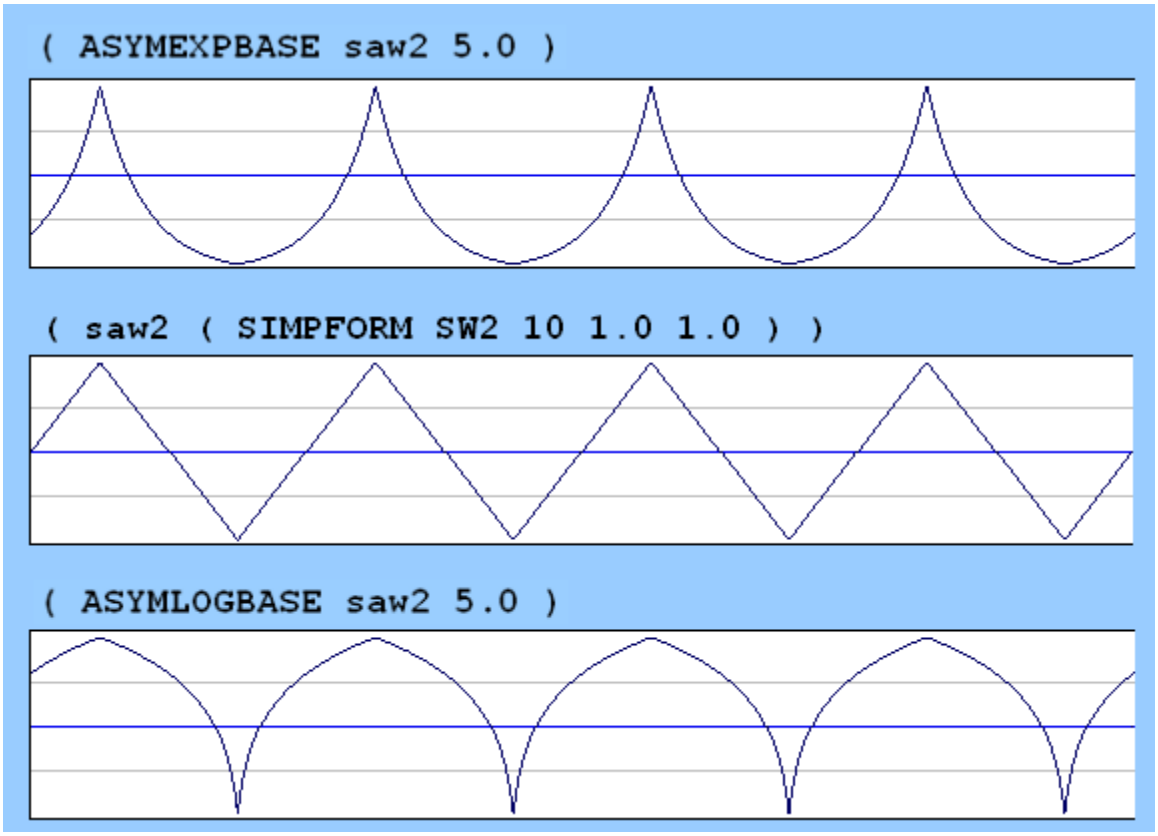
4.6.8 ASYMEXPBASE and ASYMLOGBASE

Format: `( wav2 ( ASYMEXPBASE wav1 R_base ) )`
or: `( wav2 ( ASYMEXPBASE wav1 R_base I_degree ) )`

Format: `( wav2 ( ASYMLOGBASE wav1 R_base ) )`
or: `( wav2 ( ASYMLOGBASE wav1 R_base I_degree ) )`

These functions are similar to `EXPBASE` and `LOGBASE`, except that the positive and negative values are not treated independently. Rather, the entire amplitude span is transformed, resulting in waveforms that will not be symmetrical above and below 0.0, even though the supplied wavedata may be symmetric.

The following graphic depicts a symmetric sawtooth wave (`saw2`), and the effect of `ASYMEXPBASE` and `ASYMLOGBASE`, with `R_base` of 5.0.



If an integer `I_degree` greater than 1 is supplied, the entire process is repeated `I_degree` times.

As with `EXPBASE` and `LOGBASE`, these functions always return new wavedata, and leave the supplied wavedata untouched.

4.6.9 HYPERDN

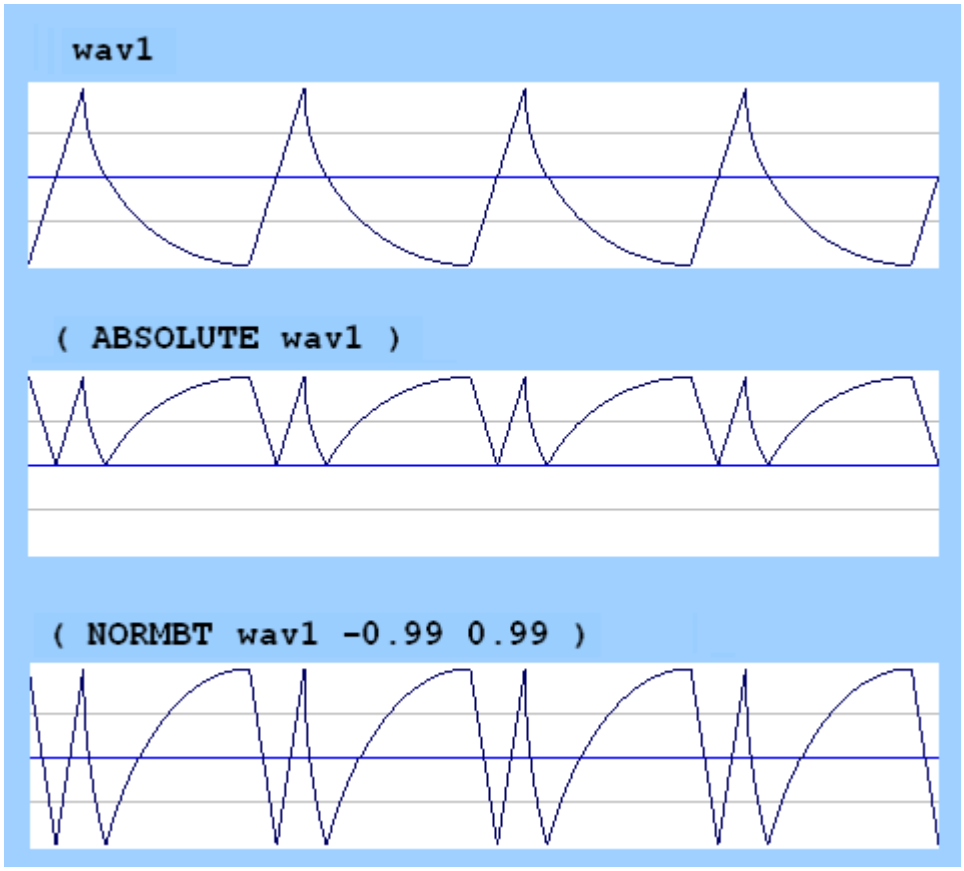
Format: `( wav2 ( HYPERDN wav1 R_degree R_size) )`

The `HYPERDN` command (Hyperbolic Suppression) is another form of background noise reduction. As with the VSMOOTH command (see 4.6.2) it is designed to apply the greatest degree of smoothing to regions of minimal amplitude. It employs a parametric hyperbolic curve whose "sharpness" is controlled by the `R_degree` parameter.

4.6.10 ABSOLUTE and NEWABSOLUTE

Format: `( ABSOLUTE wav1 )`
Or: `( wav2 ( NEWABSOLUTE wav1 ) )`

These commands simply "reflect" any portion of the waveform that appears below zero into the positive region (ala, the mathematical absolute value). While often useful in crafting modulation envelopes (some of which must take on only non-negative values) one can also re-normalize the absolute result of a waveform back to equal positive and negative range, to produce new tonal waveforms.

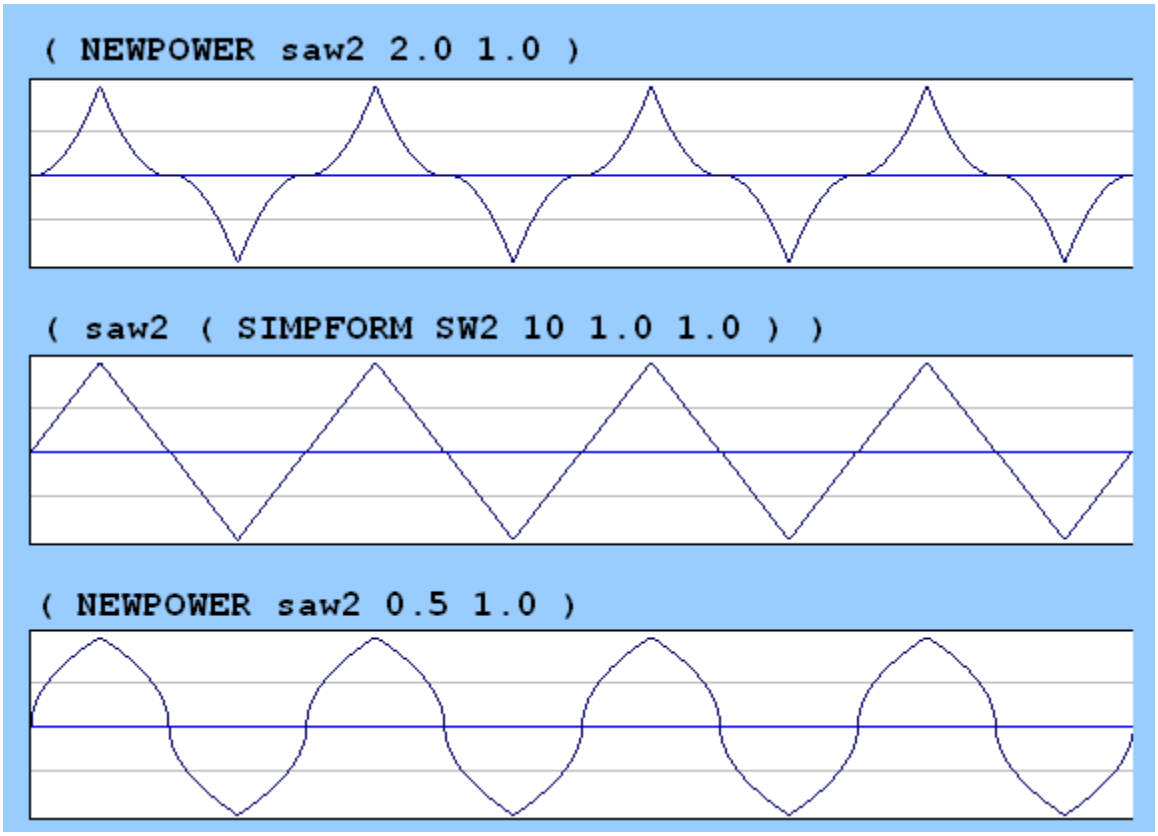


4.6.11 POWER and NEWPOWER

Format: `( POWER wav1 R_power R_amp )`
Or: `( wav2 ( NEWPOWER wav1 R_power R_amp ) )`

These commands uniformly raise or lower the supplied wavedata to the indicated power. Since (normal) wavedata lies between -1.0 and +1.0 in amplitude, raising to a power greater than 1.0 draws the wave toward zero amplitude, while raising to a power less than 1.0 draws the wave toward maximal (positive or negative) amplitude. In both cases, portions of the wavedata that have small amplitude to begin with are affected most dramatically, while portions that lie very near to maximal amplitude (1.0 or -1.0) are affected most slowly.

The following graphic depicts a symmetric sawtooth wave (`saw2`), and the effect of raising it to the power 2, and to the power 0.5.



Notice that POWER modulation does not affect the frequency of the waveform. Also, the positive and negative sections of the waveform are treated independently, thus negative portions remain negative.

Any power `R_power` greater than 0.0 is allowed.

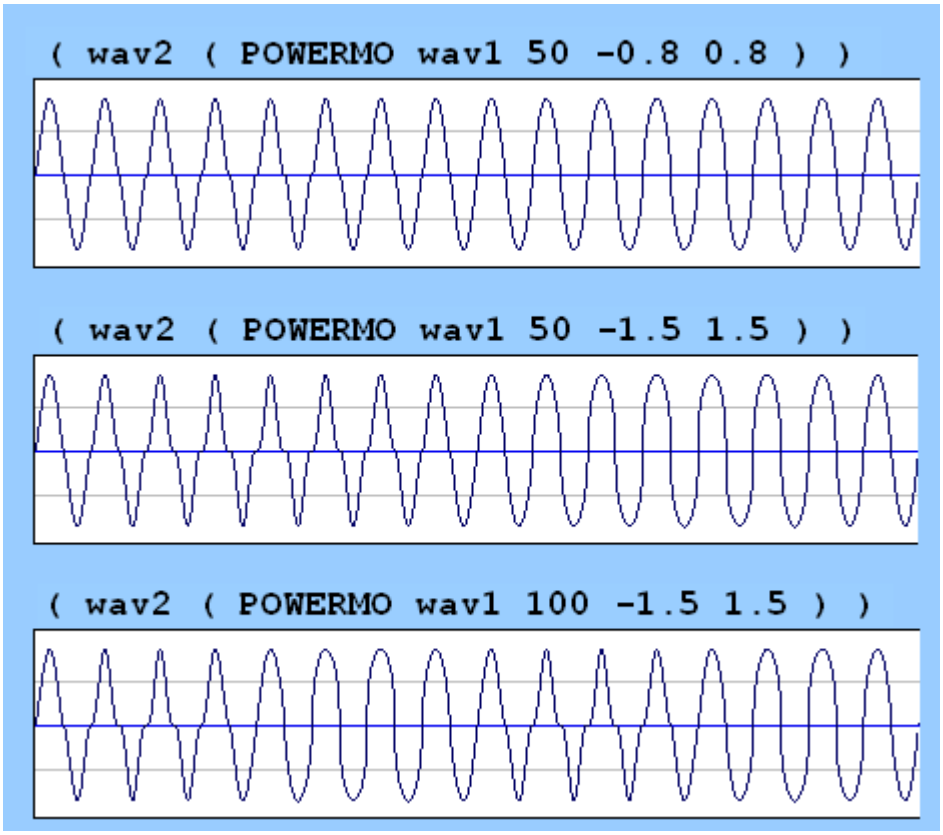
4.6.12 POWERMO

Format: `( wav2 ( POWERMO wav1 R_freq R_pmin R_pmax ) )`

This function will sinusoidally modulate the power to which the supplied waveform (`wav1`) is raised, at the specified modulation frequency and power span. For illustration, I chose to begin with an ordinary sine wave of frequency 400 Hz, using

```
( wav1 ( SIMPFORM SIN 400 0.9 1.0 ) )
```

and then I applied various `POWERMO` settings. The result is depicted below.



The actual power to which the waveform is raised will vary between 2^{pmin} and 2^{pmax}, hence for pmin = -1 and pmax = 1, the power will oscillate between ½ (square root) and power 2.0.

As with `TREMELO` and `VIBRATO`, a `POWERMO` modulation frequency (`R_freq`) below 20 Hz will result in an audible fluttering of the original timbral quality. Above 20 Hz, the effect will result in an entirely new timbral quality, with tonality (pitch) taken over by the modulation frequency itself.

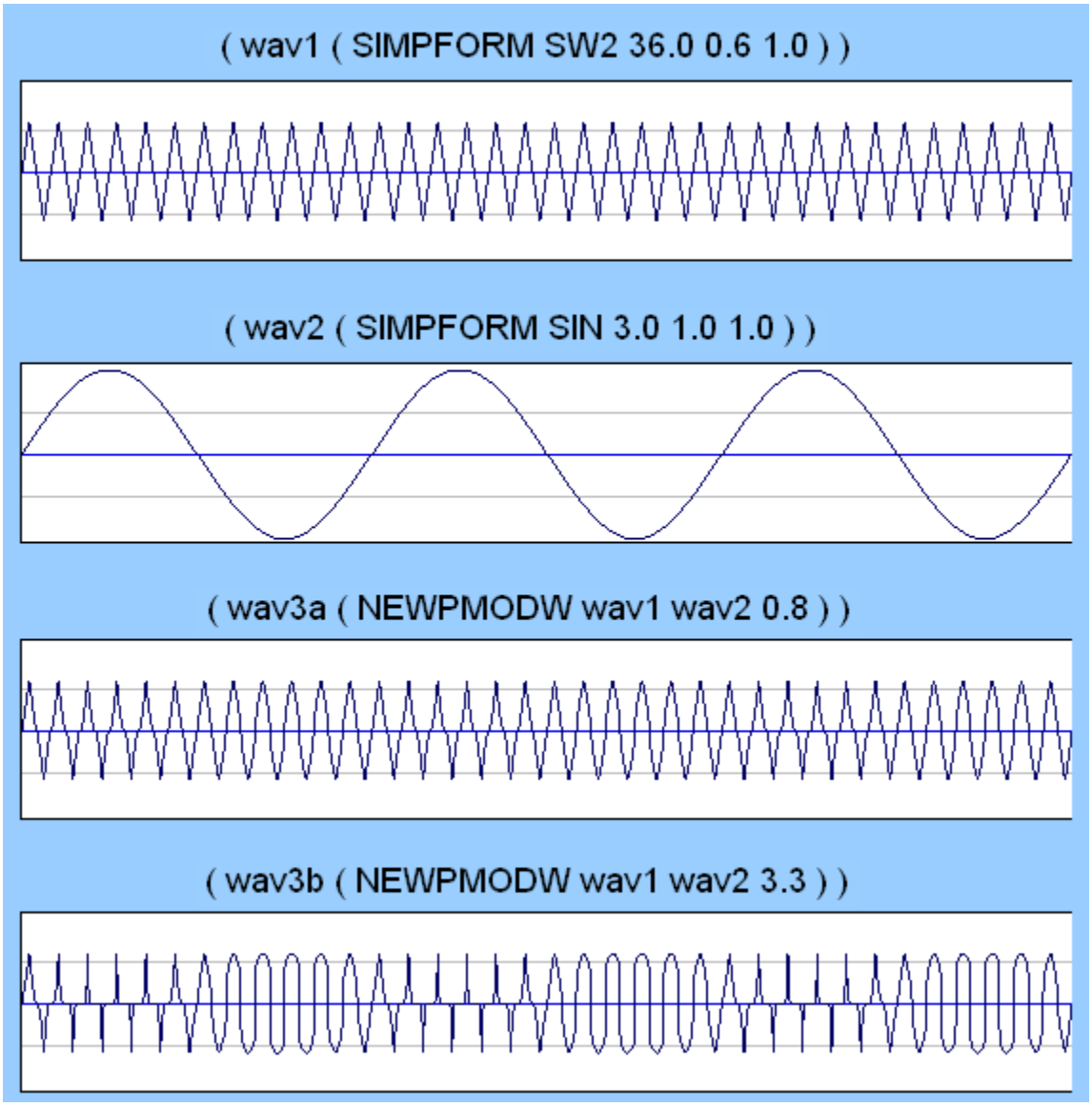
POWERMO returns new wavedata, and does not modify the original input wavedata.

4.6.13 PMODW and NEWPMODW

Format: `( PMODW wav1 wav2 R_factor )`
Or: `( wav3 ( NEWPMODW wav1 wav2 R_factor ) )`

These functions are related to `POWER` and `NEWPOWER` (see Section 4.6.11), which raise the values of the target waveform `wav1` to a fixed power. Here, `PMODW` and `NEWPMODW` raise the values of `wav1` to powers that vary according to the amplitudes of the second waveform `wav2` (actually, to 2-to-the-power of the `R_factor` scaling of the amplitudes in `wav2`.) Power modulation affects the "shape" of a waveform, but not its frequency.

In the graphic below, we take wav1 to be a sawtooth waveform at 36 Hz, and power modulate it with wav2, a leisurely 3 Hz sine wave.



4.6.14 PMODENV and NEWPMODENV

Format: `( PMODENV wav1 env R_minp R_maxp )`
Or: `( wav2 ( NEWPMODENV wav1 env R_minp R_maxp ) )`

The `PMODENV` commands (power modulate by envelope) perform the same function as `PMODW`, except that the supplied envelope `env` is stretched or compressed in duration, so as to be applied exactly once to the supplied wavedata `wav1`.

The values `R_minp` and `R_maxp` allow you map the lowest and highest amplitudes of the supplied envelope to corresponding minimum and maximum power modulation, and must satisfy $0 \leq$ `R_minp` $\leq$ `R_maxp`.

4.6.15 SMOOTHENV

Format: `( SMOOTHENV wav1 env R_maxradius )`

The `SMOOTHENV` command will smooth the waveform `wav1` to a degree dictated by the values of waveform `env`. Specifically, where env has zero value, no smoothing occurs, and where `env` approaches a value of 1.0, smoothing to a radius of `R_maxradius` (in seconds) occurs. The waveform `env` is stretched or compressed so to apply exactly once to the entire length of the data `wav1`.

4.7 Frequency Adjustment and Modulation Commands

4.7.1 NEWPITCH

Format(s): `( wav2 ( NEWPITCH wav1 R_factor ) )`
`( wav2 ( NEWPITCH wav1 R_factor R_dura ) )`

This operation adjusts the pitch (frequency) of the waveform by the indicated factor. A factor of 1.0 has no effect. A factor of 2.0 raises the tonal quality by a full octave, while a factor of 0.5 lowers the tonal quality by a full octave. Any value greater than 0 is allowed.

If the first form is used (the `R_dura` parameter is omitted) then the duration of the resulting waveform will be (1/`R_factor`) of the input `wav1`, since `wav1` is effectively being "played" faster or slower. For instance, doubling the frequency will result in a waveform of half the duration.

If the second form is used, a waveform of duration `R_dura` is produced. However, note that this is accomplished by truncating or wrapping of the input wavedata by means of interpolative wrapping. If the input wavedata "sounds different " at the end than it does at the beginning, and the requested `R_dura` is greater than (1/`R_factor`) times the original, there will be a noticeable artifact at the wrapping point. This behavior is exemplified by the graphic on the next page.

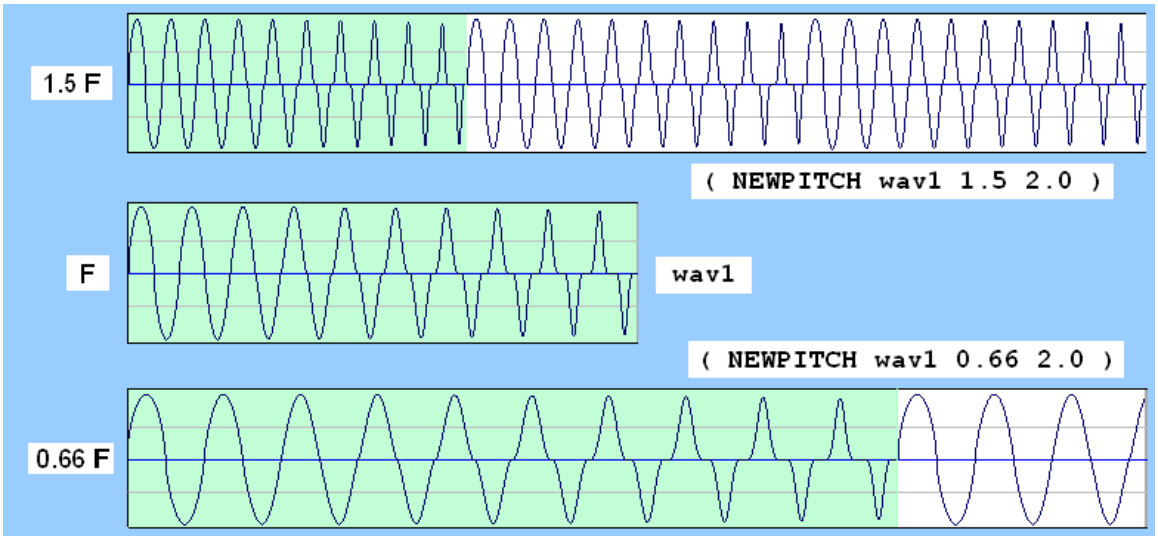
As a convenience, if `R_dura` is explicitly given as 0, the output wavedata will have exactly the duration of the supplied input wavedata.

Examples of NEWPITCH

In this example, I purposely crafted a waveform (wav1) of duration 1.0 seconds that does not have a constant timbral quality. It begins as an ordinary sine wave, but is increasingly power-modulated toward the end, resulting in a "spiky" quality. The commands I employed to produce it are these:

```
( env1 ( ENVELOPE
   ( FREQ 1.0 ) ( DURA 1.0 ) ( INITAMP 0.0 )
   ( COMPONENTS ( LIN 1 1.0 ) ) )
)

( tmp1 ( SIMPFORM SIN 10 0.99 1.0 ) )
( wav1 ( NEWPMODW tmp1 env1 3.0 ) )
```

I gave `wav1` a frequency of only 10 Hz, so that its changing shape could be discerned over it full 1.0 second duration. I label this frequency "F" in the graphic below, and depict the 1.0 seconds of `wav1` entirely in green (see middle row of image).



Using the `NEWPITCH` command, I create two new waveforms, depicted above and below `wav1`. The upper one is 1.5 x 10 Hz, or 15 Hz, and is thus "squeezed" to a shorter wavelength (higher pitch). The lower one is 0.66 x 10 Hz, or 6.6 Hz, and is thus "stretched" to a longer wavelength (lower pitch.)

Notice that in each of the `NEWPITCH` commands, I demanded a 2.0 seconds duration be produced, and that the graphics above and below `wav1` are indeed twice as long. Had I left off the "2.0" entirely, the resulting waveforms would have been returned with durations depicted by the portions shaded only in green. And had I specified 0 for the duration, each would have been returned with duration identical to `wav1` (1.0 seconds).

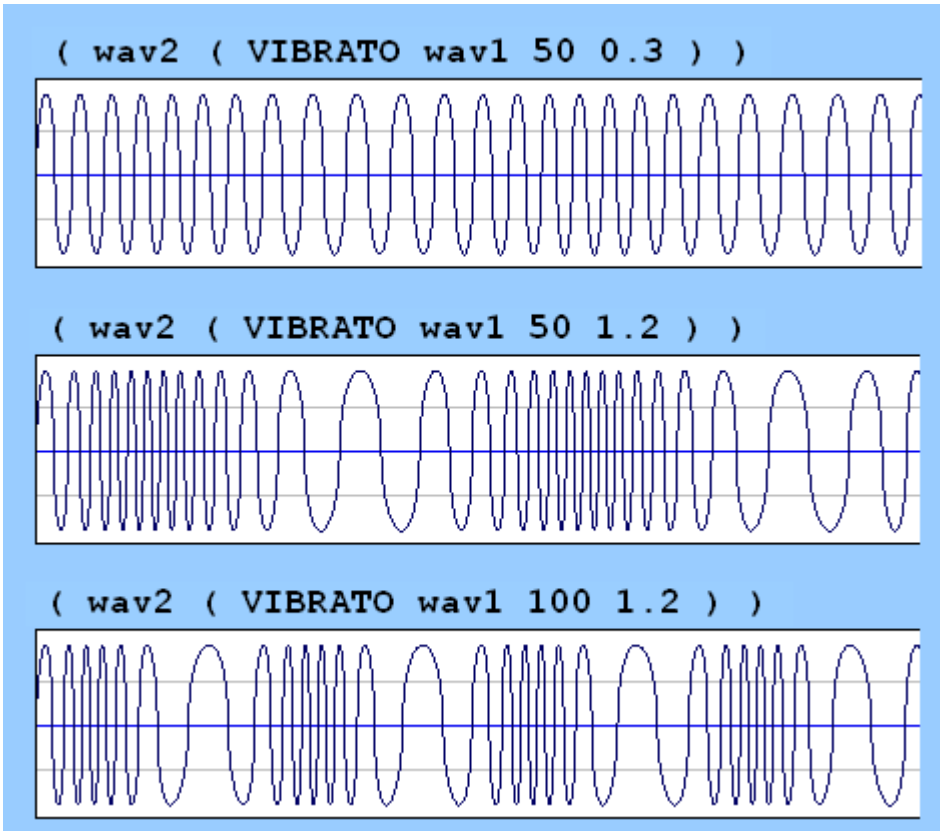
4.7.2 VIBRATO

Format: `( wav2 ( VIBRATO wav1 R_freq R_freqspan ) )`

This function will sinusoidally modulate the frequency of the supplied waveform (`wav1`) at the specified modulation frequency and frequency span. For illustration, I chose to begin with an elliptic wave of frequency 600 Hz, using

```
( wav1 ( SIMPFORM ELI 600 0.9 1.0 ) )
```

and then I applied various `VIBRATO` settings. The result is depicted below.



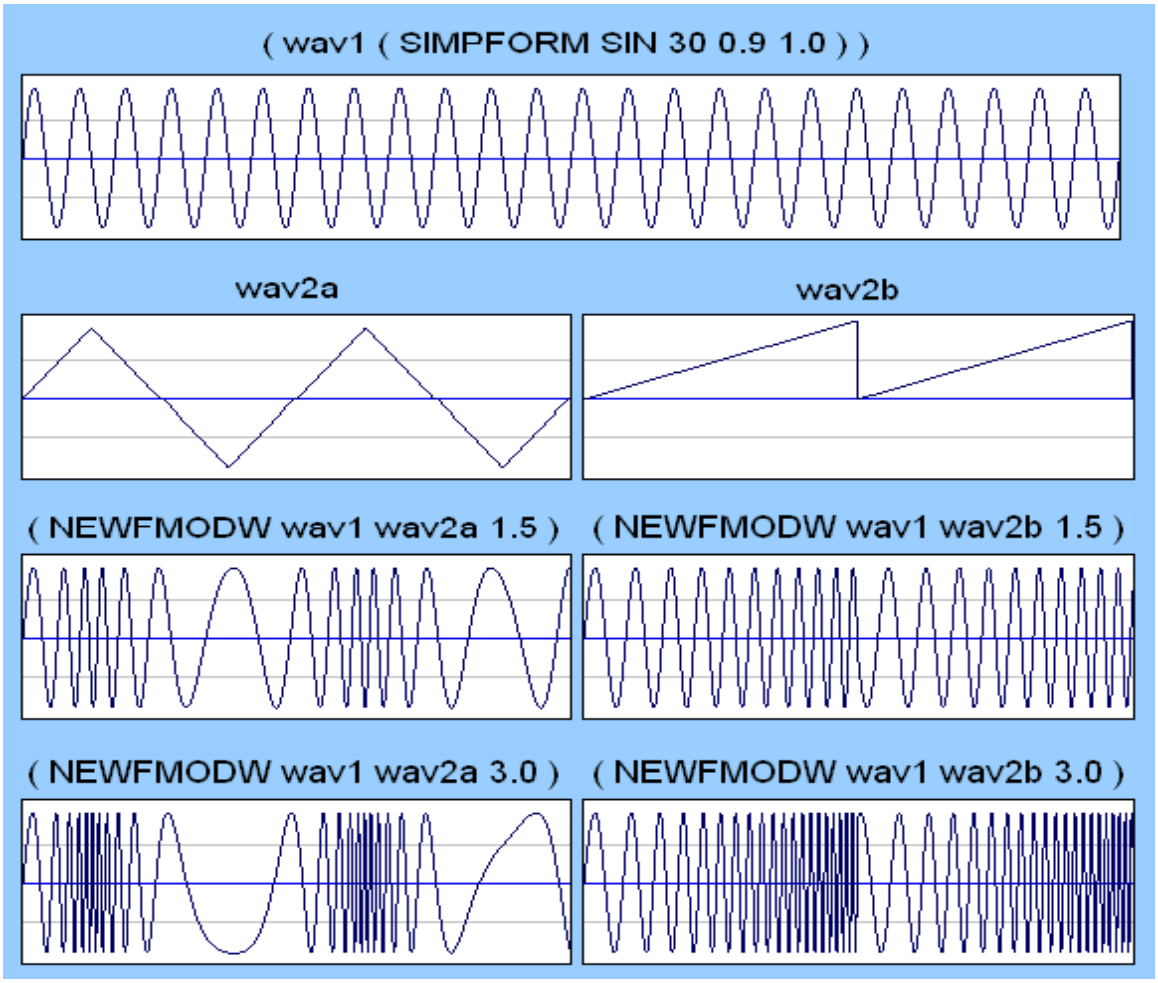
The value of `R_freqspan` gives the degree to which the original frequency of `wav1` is to be raised or lowered. An `R_freqspan` of 1.0 means that the pitch will be modulated plus and minus 1 full octave (i.e., between 2f and 0.5f). The rate at which the frequency will be varied is given by the `R_freq` value.

If the `VIBRATO` frequency `R_freq` is a value below about 20 Hz (too low to be perceived as a tone in itself), one will audibly hear the tone quavering in pitch, as a guitarist might produce by wiggling a string after it is played. Above 20 Hz, the pitch begins to become that of the modulation frequency itself, with the original waveform and frequency acting mostly to determine the timbral quality of the result. `VIBRATO` returns new wavedata.

4.7.3 NEWFMODW and NEWFMODENV

Format: `( wav3 ( NEWFMODW wav1 wav2 R_factor ) )`
`( wav3 ( NEWFMODENV wav1 env R_factor ) )`

This function modulates the frequency (pitch) of `wav1`, according to the amplitude values of `wav2` or an envelope. The `R_factor` is used to adjust the amplitude of `wav2` before it is used for the modulation. Examples are depicted in the illustration below.



On the left side of the above figure, note that the modulator `wav2a` has an amplitude range of about [-1.0,1.0]. The `R_factor` value of 3.0 used for the lower-left modulation acts to vertically expand that range to [-3.0,3.0]. Then since $2^{-3} = 1/8$ and $2^3 = 8$, the modulation will cause the original frequency of `wav1` to vary between 8 times slower and 8 times faster. One the right side, the amplitude range [0,1] of `wav2b` becomes [0,3], and since $2^0 = 1$ and $2^3 = 8$, `wav1` gets modulated between 1 times and 8 times its original rate.

With `NEWFMODW`, the modulating wav is expected to wrap, and its endpoints are smoothly joined in this anticipation. With `NEWFMODENV`, the envelope is left unmodified.

4.7.4 DEFREQ and ENFREQ

Format: `( wav2 ( DEFREQ wav1 R_lofreq R_hifreq ) )`
`( wav2 ( ENFREQ wav1 R_lofreq R_hifreq ) )`

The `DEFREQ` and `ENFREQ` commands act to de-emphasize or enhance frequency components at or near `R_lofreq` and of diminishing effect tending toward `R_hifreq`. Precisely, at each point x in the supplied wav1, a long-range and short-range mean of samples, centered on x and corresponding in length to the wavelengths of lofreq and hifreq are calculated. The wavedata returned replaces the value v at x with v – (shortmean – longmean) in the case of `DEFREQ`, and with v + (shortmean – longmean) in the case of `ENFREQ`.

4.7.5 FSWEEP

Format: `( wav2 ( FSWEEP wav1 env R_lofreq R_hifreq ) )`

Apply `ENFREQ` to `wav1`, applying the corresponding value of `env` as a factor to the frequency basis `R_lofreq` and `R_hifreq`. The envelope `env` is cycled over if shorter than `wav1`.

4.8 Special Effects Commands

4.8.1 REVERB and AFTREVERB

Format:

```
( wav2 ( REVERB wav1 [R_intens R_esize I_comps R_life] ) )
( wav2 ( AFTREVERB wav1 [R_intens R_esize I_comps R_life] ) )
```

The REVERB and AFTREVERB effects provide a decaying echoic quality to a waveform. To understand the effect, consider how one would simulate a simple echo. Naturally, one would take a copy of the sound and lay it down on top of itself after a small delay, and with reduced amplitude, as in "HELLO … … hello …". The sound of the echo would itself echo after another small delay at a yet reduced amplitude, as the series of echoes die out. A reverb is "like" such an echo-series, except that each "echo" is increasingly smeared-out (and typically, the series of "echoes" occur rather quickly and thus are usually not perceived as distinct echoes). This "smearing" is accomplished by using a number of slightly-displaced copies of the original sound at the first echo-point, and likewise "smearing" the resulting echo itself when applied to the next echo point, etc. Essentially, each echo-point is the locus of a small "echo-cluster".

When music is played with ordinary reverb effect, the original sounds appear as well as their reverberant series. The AFTREVERB effect produces a particularly haunting quality by subtracting-out the original sound, leaving only the reverberance.

Four parameters are employed to control the nature of the REVERB and AFTREVERB effects. By default, when using `( wav2 ( REVERB wav1 ) )` with no additional specifications, these parameters are set as follows:

```
R_intens = 0.05
R_esize  = 0.08
I_comps  = 9
R_life   = 1.4
```

The R_intens value determines the amplitude decay applied to each successive echo-cluster. Smaller values cause the reverb to decay more quickly.

The R_esize value determines the spacing, in seconds, between the successive echoes (actually, between the echo-clusters), while I_comps determines the number of mini-echoes comprising each cluster.

The R_life and R_esize together determines the number of "echoes" employed, by the formula `echoCount = R_life/R_esize` (thus, default is 17 echoes).

NOTE: Because these parameters are positionally defined, you must specify all up to the last one you wish to modify. Thus, to modify R_life, you must supply all parameters.

4.8.2 CHORUS

Format: `( wav2 ( CHORUS wav R_wfreq [R_qfreq R_qdepth R_intens I_size I_patchends] ) )`

The `CHORUS` command produces a "choir-like" effect from an otherwise singular tonal "voice" or instrument sound. To understand how this is accomplished, imagine a single human voice attempting to sing a sustained pure tone. As hard as they try, the note they are attempting to maintain will drift very slightly above and below the target frequency, by a small fraction of a semitone, and do so in a slightly random but regular, or "quasi-periodic" manner. The `CHORUS` effect induces these quasi-periodic frequency modulations, independently, to several copies of the supplied wav data, and then adds these together to produce the resulting "chorus".

Minimally, one must supply a value for `R_wfreq`, representing the frequency of the supplied wavedata `wav1`. If `wav1` is a complex of tones, then an approximate guess at the most fundamental frequency will suffice.

All remaining parameters are optional, and if omitted will assume the values

```
R_qfreq  = 7.0
R_qdepth = 0.01
R_intens = 0.8
I_size   = 5
I_patchends = 0
```

The `R_qfreq` value corresponds to the average frequency with which each voice "quavers" above and below its original pitch.

The `R_qdepth` value limits the degree to which the frequency quavering can vary above or below the original pitch.

The `R_intens` value determines the relative amplitude each successive "voice" contributes to the total chorus.

The `I_size` value represents the total number of voices present in the chorus.

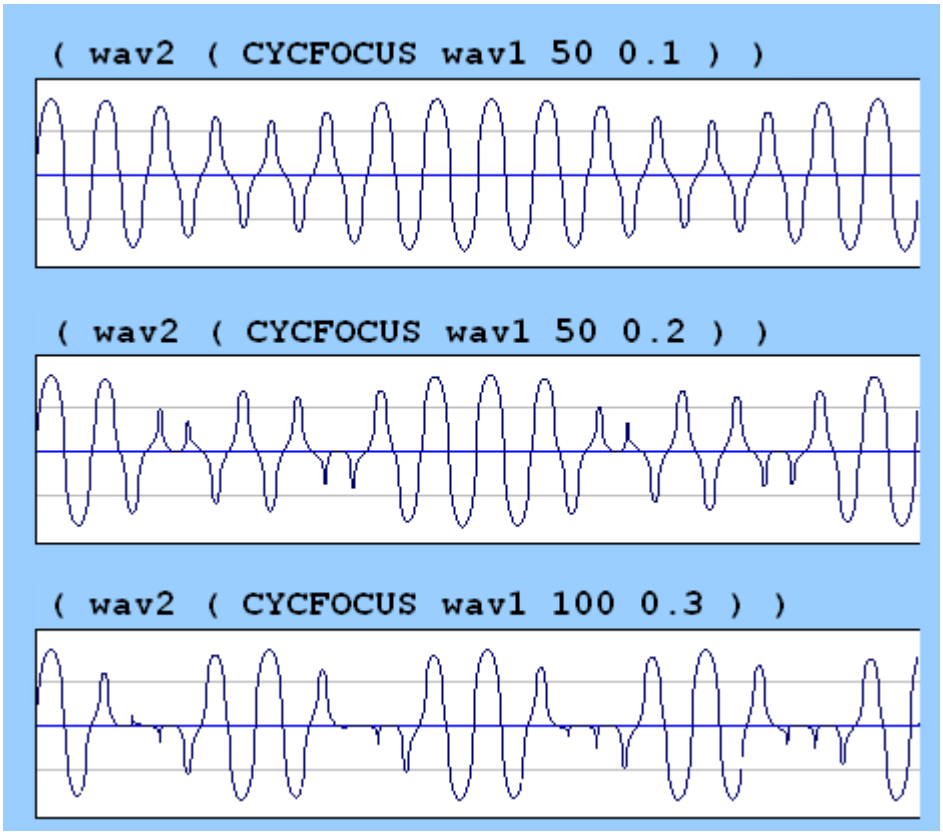
The `I_patchends` parameter is somewhat overloaded. By default, in addition to the independent pitch-quavering applied to each voice, the voices are time-shifted randomly throughout the base wavelength in order to avoid strong coincidence at the start of the assembly. To avoid this, and force all voices to start together, use `I_patchends = -1`. This is best when the sound being "chorused" changes moment to moment, or is a one-time "strike" at a note or instrument. However, the `CHORUS` function is very useful when supplied a near-constant tone (in formulating the basis for instrument sounds), in which case it is likely that one intends to "loop" the result repeatedly to create long sustained notes. In this case, it is best to use `I_patchends = 1`, which time-shifts the voices, AND properly "melds" the beginning and ending of the wavedata for smooth looping.

4.8.3 CYCFOCUS

Format: `( wav2 ( CYCFOCUS wav1 R_freq R_freqspan ) )`

This function takes the same parameters as `VIBRATO`, and performs a related operation. Recall that `VIBRATO` acts to sinusoidally modulate the pitch of the supplied wavedata, at a frequency and depth specified by `R_freq` and `R_freqspan`. This resulted in a waveform that was alternately "stretched" and "compressed". The `CYCFOCUS` operation will perform the `VIBRATO` operation twice, to produce two sets of wavedata, with the difference that one set is "stretched" where the other is "compressed", and vice versa. These two sets of wavedata are then averaged, and the result is returned.

The following graphic depicts these effects upon `wav1`, a plain elliptic wave of 400 Hz.



Notice that the original elliptic waveform comes "into focus" according to the desired periodicity, and becomes distortedly out of focus between these regions. These effects can be very peculiar at both low intensity (`R_freqspan`) or high intensity, and will produce interesting long-cycle drifting of timbral quality when the supplied `R_freq` is very close to the natural frequency of the supplied waveform `wav1`.

4.8.4 BEATWAVE

Format: `( wav2 ( BEATWAVE wav1 R_wfreq R_beatfreq ) )`

Discussion: Suppose you have two waveforms, w1, and w2, that are identical except in their frequency. For instance, let frequency w1 = 50 Hz, while frequency w2 = 49 Hz. Add these waveforms together, and they are initially in sync. But after 0.5 seconds have passed, the waveforms are at a point where w1 has completed exactly 25 of its 50 cycles, while w2 has completed exactly 24.5 cycles of its 49 cycles. The two waves are now exactly out of phase, and (if they had identical amplitudes) are canceling each other out. Yet by the time a full 1.0 seconds have passed, they are once again in sync, having completed exactly 50 and 49 full cycles, respectively. This "phasing in and out" has the effect of causing the tone to go from loud to soft and back to loud again, exactly once every second. This "amplitude oscillation" is called a "beat", and we can say that the effect of adding these two waveforms results in a "beat-frequency" of 1 Hz.

If the frequencies of w1 and w2 were 50 Hz and 48 Hz respectively, it is not hard to see that the resulting beat-frequency would then be 2 Hz. Clearly, the first opportunity for each to be simultaneously "in-phase" would be after 25 cycles of w1, and 24 cycles of w2, thus a "beat" every half-second. At 50 Hz and 40 Hz, the beats would occur every 5 cycles of one and 4 of the other, giving a beat-frequency of 10 Hz.

As w1 and w2 diverge in frequency, the beat-frequency continues to rise, becoming a tonal component of the sum above beat-frequency 20 Hz. One can even have the resulting beat-frequency "harmonize" with the original w1 component.

Suppose you WANTED a particular beat-frequency, and had only w1 to start with? Rather than have to figure out the proper frequency of a new waveform w2 that would be required, and have to create that new wave just to add it to the first, the `BEATWAVE` function will perform these operations for you, and return the consequent sum.

Supply `BEATWAVE` with a waveform `wav1`, and its actual frequency `R_wfreq`, along with the desired beat-frequency `R_beatfreq`. `BEATWAVE` will calculate the necessary frequency that a "`temp1`" copy of `wav1` would require, produce that `temp1` (via a `NEWPITCH` operation), and then return the wave sum of `wav1` and `temp1`. The input wavedata `wav1` is not affected, and the temporary `temp1` is disposed of automatically.

4.8.5 DEPHASE and ANTIDEPHASE

Format: `( wav2 ( DEPHASE wav1 R_wfreq R_decayrate ) )`
Format: `( wav2 ( ANTIDEPHASE wav1 R_wfreq R_decayrate ) )`

The `DEPHASE` command takes a waveform `wav1` and adds copies of it, each a small degree out of phase with the one before it, and with diminishing amplitude, such that the last copy has only 1/1000th the original amplitude. The number of such copies added depends upon the given decay rate. A decay rate of 0.1 will reach 0.001 amplitude after only 3 copies have been added, with phases distributed uniformly between 0 and maxphase, while a decay rate of 0.5 will require about 10 copies. The degree of phase-shift each copy receives is such that the last copy is shifted ½ wavelength from the original, and all intermediate copies are uniformly spaced between these.

The `ANTIDEPHASE` command performs the same function, except that instead of adding each diminished copy, copies are alternately subtracted and added.

4.8.5 FSMEAR and ANTIFSMEAR

Format: `( wav2 ( FSMEAR wav1 R_maxmult R_decayrate ) )`
Format: `( wav2 ( ANTIFSMEAR wav1 R_maxmult R_decayrate ) )`

The `FSMEAR` command takes a waveform `wav1` and adds copies of it, each incrementally (geometrically) of higher pitch than the one before it, up to a limit multiple `R_maxmult`, and each with diminishing amplitude, such that the last copy has only 1/1000th the original amplitude. The number of such copies added depends upon the given decay rate. A decay rate of 0.1 will reach 0.001 amplitude after only 3 copies have been added, with frequencies distributed geometrically between 1 and maxmult of the input wav's rate, while a decay rate of 0.5 will require about 10 copies.

The `ANTIFSMEAR` command performs the same function, except that instead of adding each diminished copy, copies are alternately subtracted and added.

4.8.6 INTRAPHASE and INTRAPHASEVENV

Format: `( wav2 ( INTRAPHASE wav1 R_wfreq R_pfreq R_pspan ) )`

The `INTRAPHASE` command takes a waveform `wav1` and a frequency value `R_wfreq`, from which it assumes a `wav1` wavelength of `1/wfreq`. Within this wavelength, it performs a positive to negative frequency modulation, such that the earlier part of the waveform is "compressed" toward its beginning and the latter part stretched away from its ending. The degree of this deformation increases and decreases, with an independent periodicity given by `R_pfreq`, and to a maximal deformation given by 0 < `R_pspan < 1`.

4.8.7 INTRAPHASEVENV

Format: `( wav2 ( INTRAPHASEVENV wav1 R_wfreq env ) )`

The `INTRAPHASE` command takes a waveform `wav1` and a frequency value `R_wfreq`, from which it assumes a `wav1` wavelength of L =`1/wfreq`. Within this wavelength, it performs a positive to negative frequency modulation, according to the value of env at each point.

Specifically: To calculate wav2(t), the value r = (1-env(t))/(1+env(t)) is calculated, and then wav2(t) = wav1(L*[INT(t/L) + (t/L - INT(t/L))^r])

4.8.8 INTRAPHASERAND

Format: `( wav2 ( INTRAPHASERAND wav1 R_wfreq R_min R_max ) )`

The `INTRAPHASERAND` command is similar to `INTRAPHASE`, except that there is no periodicity to the degree of deformation applied to each wave-period of `wav1`. Instead, each successive period receives a random degree of deformation selected from the range `R_min` and `R_max`, which must satisfy `0 < R_min < R_max < 1`.

The general effect of `INTRAPHASERAND` is to create another form of atimbral sound.

4.8.9 TSUNAMI

Format: `( wav2 ( TSUNAMI wav1 S_coeffset [I_comps] ) )`

The `TSUNAMI` command is similar to the `QUASIFOURIER`, (see 4.2.7). Recall that the `QUASIFOURIER` command would accept a piece of wavedata wav1, representing a periodic fundamental form, and apply the given set of coefficients in a Quasi-Fourier synthesis.

With the `TSUNAMI` command, the synthesis begins as in `QUASIFOURIER`, but over the entire length of the supplied data wav1, the coefficients are gradually (proportionally) swapped, with coefficient 1 becoming coefficient n, coefficient 2 becoming coefficient n-1, etc. The swapping reaches 100% at the midpoint of wav1, and then returns the coefficients gradually back to their original values. The effect is to swap the high-frequency harmonics with the lower fundamentals, and then return, exactly once over the length of wav1. The name "Tsunami" was selected, short for "Harmonic Tsunami".

4.9 Melodic Sequence Definition and Application Commands

4.9.1 STORETONESEQ

Format:

```
( STORETONESEQ seqname
   ( R_stm ( S_key R_dura [R_vol] )* )
   ( R_stm ( S_key R_dura [R_vol] )* )
   [ etc ]
)
```

The `STORETONESEQ` command is your primary "song writing" function. It does not itself produce wavedata, but rather allows you to define a sequence of notes and chords that can be applied to an instrument with the corresponding `PLAYTONESEQ` command.

The values `R_stm` are the "start-time" offsets into the sequence where you want the listed note or notes to be played. Each note is specified by supplying a parenthetical expression of the form `( S_key R_dura [R_vol] )` where

`S_key` is one of { `C0, C#0, D0, … C1, C1#, D1, … Cn,C#n,Dn, …` }
`R_dura` is the desired duration of that note
`R_vol` is an (optional) modifier to the loudness of that note

A chord is specified simply by listing multiple `( key dura [vol] )` expressions within the same encompassing `R_stm` expression. Notes within a chord may be given their own intensity (vol) and their own durations, and those durations may be shorter than, or exceed, the `R_stm` of subsequent note or chord expressions. The default "vol" assigned to each note is 1.0, and you can make any note relatively softer or louder by providing `R_vol` values such as 0.8 or 1.3, etc.

Here is an example, using the first "Row, row, row your boat" measure from the well-known melody, except that I have replaced the initial "C" note with a C-major chord:

```
( STORETONESEQ seqname
   ( 0.0 ( C3 0.6 ) ( E3 0.6 ) ( G3 0.6 ) )
   ( 0.3 ( C3 0.3 1.2 ) )
   ( 0.6 ( C3 0.6 ) ( E3 0.6 ) ( G3 0.6 ) )
   ( 0.8 ( D3 0.1 1.2 ) )
   ( 0.9 ( E3 0.3 1.2 ) )
)
```

Note that I have made the "C,E,G" chord sustain for 0.6 seconds, beyond the "C" note that sounds after only 0.3 seconds, and have made that lone "C" note louder to give it emphasis between the chords. Also, notice that the entire phase lasts a (nominal) 1.2 seconds (the "E" note sounds after 0.9 seconds, and lasts for 0.3 seconds.)

It may seem rather cumbersome to have to specify note offset start-times and durations this way, but realize that these specifications are all "relative to the phrase". When it comes time to apply this phrase to a tonal sound or instrument, you will be able to transpose the entire phrase to a new key, override the entire volume, modify ALL sustains (`R_dura`) by any proportion, and most importantly, adjust the overall tempo by any proportion. For instance, if you supply `( TEMPO 0.5 )` at `PLAYTONESEQ` time, all of the `R_stm` and `R_dura` values get divided by 0.5 (i.e., multiplied by 2.0), and the phrase will then play at half the original tempo, and be 2.4 seconds in length.

QUESTION: (IMPORTANT)

When you specify note keys like C3 or G#4, are you really getting the note that corresponds to the same note on (say) a real piano keyboard?

The answer is "depends". You can force this to occur if you are careful in crafting the base wavedata sounds that will comprise your "orchestra" of instruments. When crafting basic wavedata sounds, you often specify the desired frequency in hz (hertz, or cycles per second). Since a lot of the subsequent modulations are mathematical, it is "easiest" to comprehend these operations when dealing with familiar numbers like 100 hz, rather than (say) A-440 (440 hz), or the C below A-400, whose actual frequency is about 261.626 hz.

Since the most common scale when learning tonic music is the C major scale, WavSynth performs as follows:

> When wavedata is created, it is produced with exactly the specified frequency, and will play this way when sounded by itself, when mixed with other wavedata, etc.
>
> However, when `PLAYTONESEQ` is invoked, the applied instrument sound is shifted upwards in frequency by the factor 1.308127827, and this has the following effect

```
Wavefreq    ShiftFreq   KeyboardNote
   25.0       32.7032      C0
   50.0       65.4064      C1
  100.0      130.8128      C2
  200.0      261.6256      C3
  400.0      523.2511      C4
  800.0     1046.5023      C5
  [ etc ]
```

In short, any wavedata whose fundamental frequency is a "power of 2" multiple of 100hz, such as 25, 50, 100, 200, 400, 800, 1600, etc, will map exactly to a standard keyboard C, in a given octave. If you have created wavedata whose frequency is not such a multiple, you can always convert it to a new frequency with the `REPITCH` command.

4.9.2 PLAYTONESEQ

Format:

```
( wav ( PLAYTONESEQ seqname instrument [options] ) )
```

where "options" may be any one or more of

```
( TEMPO R_tfactor )
( KEYMOD I_semitones )
( KEYENV env1 [R_attackpart] )
( SUSTAIN R_sfactor )
( MIXTYPE TRN R_trntime )
( SEQENV env2 )
( REVERB or AFTREVERB or NOREVERB )
( FINAMP R_amp )
```

The PLAYTONESEQ command will return new wavedata "wav", by applying the indicated wav sound "instrument" to any tone-sequence "seqname" specified previously with the STORETONESEQ command. The result is a "solo-performance" of the indicated tone-sequence, which may be mixed and sequenced with others using the "MIX" commands to produce larger arrangements and songs.

The listed options provide the following additional capabilities:

TEMPO:
: Adjust the tempo (timing) of the melody by the factor R_tfactor > 0. For instance, a factor or 2.0 will double the pace, etc. If omitted, the effect is equivalent to (TEMPO 1.0), which has no effect.

KEYMOD:
: Transpose the supplied melody upwards or downwards by the indicated number of semitones. Allowed range of values is -288 to 287.

SUSTAIN:
: Extend or reduce ALL relative sustains (dura) in the tone sequence by the indicated factor R_sfactor. For instance, (SUSTAIN 1.5) will cause each note to sustain 50% longer than its original specification. The tempo and timing of the overall phrase is not affected.

SEQENV:
: Apply the given envelope env2 to affect the amplitude over the whole length of the returned wavedata. For instance, an envelope that rises from 0.5 to 1.0 will cause the piece to gradually double is volume over its length, effecting a "crescendo", while an envelope that drops from 1.0 down to 0.0 will cause the piece to gradually fade to silence over its length.

REVERB or AFTERREVERB or NOREVERB: (added as a convenience)

FINAMP:
: Adjust the final maximal amplitude of the returned wavedata.

The remaining options, KEYENV, and MIXTYPE require a bit of extended explanation.

KEYENV:
Apply the given envelope env1 to each individual note in the tone-sequence, as it is laid down into the resulting wavedata. The envelope is stretched or compressed in order to accommodate the duration of each note. If the optional attackpart parameter is supplied, then the initial portion of the key envelope given by 1.0/attackpart is left unchanged, and only the remaining portion of the envelope is stretched or compressed.

MIXTYPE TRN:
Ignore the note durations given in the melody. Instead, phase-out whatever volume remains of each note or chord, as the next time-sequenced not or chord is encountered. The supplied R_trntime value determines the interval, in seconds, over which the transition occurs.

Discussion:

When wavedata you have created as an "instrument" sound is supplied to PLAYTONESEQ, there are several issues to consider. If your instrument sound has already been "shaped" by a modulation envelope to produce a particular attack and decay, then you will probably want to omit the KEYENV entirely. However, this is only wise if you have ensured two other things, (1) that the attack portion of your instrument envelope is very brief, and (2) the "sample" representing the instrument sound is rather long. The reason is that, in order for PLAYTONESEQ to produce various tonal notes, it must "play" the sound faster or slower to change the pitch. This will have the unfortunate effect of "hardening" or "dulling" the attack part of the envelope, which MUST be stretched or compressed in its entirety to effect the desired pitch change. Worst of all, if your instrument sound is naturally at a low frequency, and you ask for a "high note" (hence the sample must be "played faster"), it will end up reaching the end of your sample sound and repeat again from the beginning, causing a repeat of the attack. (This can be addressed by applying "MIXTYPE TRN", but at the expense of the original note durations – the notes will be artificially transitioned-out whenever a subsequently-start-timed note is encountered.)

In contrast, it is better to have your instrument sound prepared "un-shaped", and able to be looped smoothly (or, be at least 5-10 seconds in length, and formed at a fundamental frequency like 400 or 800 hz.). This "instrument" wavedata can be sampled/interpolated fast or slow to produce different "notes", for any duration. Then, the final "touch" that defines the instrument sound, the KEYENV, will be applied to the tonal note *after* the pitch operation has been performed, resulting in a cleaner application of the envelope.

Also, You can have the note-durations vary without hardening or dulling the attack part of the envelope. For instance, if the attack part of your KEYENV is 5% of the entire envelope, use (KEYENV env 20), to express that the attack is 1/20th of the total.

4.9.3 PLAYGLISSANDO

Format:

```
( w2 ( PLAYGLISSANDO seqname w1 I_key R_tempo R_tranrate [env]) )
```

The PLAYGLISSANDO command is an alternate way to apply a stored tone sequence to an instrument. With PLAYGLISSANDO, the sequence of notes are "pitch-glided" from one to the next. This acts to produce an effect like a slide guitar or slide trombone, where notes rise and fall. Note durations are ignored, as each note will last until the next scheduled note is due to sound. The amount of time over which the glide takes place is determined by the R_tranrate parameter.

Because of the pitch-glide nature of processing, PLAYGLISSANDO will play the supplied instrument sound "w1" continuously, using the indicated tone sequence to specify the timing of frequency modulations. For this reason, it is important to ensure that the supplied instrument sound is relatively continuous and "loops smoothly" when wrapping from the end back to the beginning.

As with PLAYTONESEQ, the overall key and tempo of the composition can be adjusted using the I_key and R_tempo values.

If the (optional) key-shaping amplitude envelope is specified, it is applied after the glissando is produced, and rescaled in size according to the effective duration of each tone played.

4.10 Percussion Sequence Definition and Application Commands

4.10.1 STOREPERCSEQ

Format:

```
( STOREPERCSEQ seqname
   [ ( INSTMAP ( ptag inst ) ( ptag inst ) ... ) ]
   ( R_stm ( inst [R_vol] [R_dura] [R_fmod] )* )
   ( R_stm ( inst [R_vol] [R_dura] [R_fmod] )* )
   [ etc ]
)
```

The `STOREPERCSEQ` command allows you to specify an entire percussion sequence involving multiple percussion instruments. At each point R_stm seconds into the sequence, one can specify one or more instruments be struck by describing each with a tuple of the form

```
( instrument [R_vol] [R_dura] [R_fmod] )
```

Unlike the instruments normally played with the `PLAYTONESEQ` command, percussion instruments are almost always pre-shaped, having their own unique percussive attack and decay envelopes applied at the time the instrument sound itself is formed.

The importance (and effect) of modifying strike duration and/or pitch for such instruments is rather diminished. In contrast, the most important aspect that defines a percussion sequence is the relative timing and intensity of the various strikes. For this reason, the order of parameters in the tuple has been arranged so that the modifications to strike volume, duration or pitch are optional, and ordered such that the most often used parameter, `R_vol`, comes first.

Note: If `R_dura` is omitted (or is 0) then the instrument sounds for its "natural" length.

Lastly, percussion sequences can be rather long. Imagine a long sequence like

```
( STOREPERCSEQ myriff
   ( 0.000 ( cymbal1 1.2 ) ( snare1 1.4 ) )
   ( 0.005 ( snare1 1.4 ) )
   ( 0.010 ( snare1 1.4 ) )
   ( 0.000 ( cymbal1 1.2 ) ( snare1 1.4 ) )
   [ etc ]
)
```

Suppose you decide you would like to try it using the instrument "tomtom1" in place of "snare1". It would be very tedious to have to edit each line to make such a change.

Instead, you could begin with the optional INSTMAP section, for instance

```
   ( INSTMAP ( inst1 cymbal1 ) ( inst2 snare1 ) )
```

Then, using `inst1` and `inst2` in the "score", you need only change "`snare1`" in one spot.

4.10.2 PLAYPERCSEQ

Format:

```
( wav ( PLAYPERCSEQ seqname [options] ) )
```

where "options" may be any one or more of

```
( TEMPO R_tfactor )
( SUSTAIN R_sfactor )
( REVERB or AFTREVERB or NOREVERB )
( SEQENV env2 )
( FINAMP R_amp )
```

The PLAYPERCSEQ command will return new wavedata "wav", by realizing the schedule of instrument strikes defined by STOREPERCSEQ in the named sequence "seqname". The set of options to this command a very much like those for PLAYTONESEQ, except that there is no "instrument" assigned, there are no global pitch modifications and there is no strike envelop applied. As explained in the section for STOREPERCSEQ, all instruments have been specified in the sequence itself, and those instruments (their wavedata) is expected to have already been "shaped" by attack-decay envelopes unique to each instrument.

5 Global Process Commands

5.1 Global Playback Settings

5.1.1 GLOBAL_WAVALIAS
5.1.2 GLOBAL_SETTEMPO
5.1.3 GLOBAL_KEYSHIFT
5.1.4 GLOBAL_KEYSUSTAIN
5.1.5 GLOBAL_PLAYEFFECT
5.1.6 GLOBAL_FINALAMP

The six commands listed above may appear at any point in the SYNTHCMDS section, and are intended to affect the behavior of all subsequent PLAYTONESEQ or PLAYPERCSEQ commands. To motivate their usage, imagine you have already defined a number of instrument sounds, and a series of musical phrases or measures. In the SYNTHCMDS section, you issue several PLAY commands to produce the wavedata for each phrase or measure, in preparation for the final mixing phase. Your commands might appear as follows:

```
( SYNTHCMDS
   ( part1 ( PLAYTONESEQ phrase1 guitar-06
      ( KEYMOD -12 ) ( TEMPO 0.32 ) ( KEYENV env22 )
      ( SUSTAIN 1.8 ) ( FINAMP 0.75 ) )
   )
   ( part2 ( PLAYTONESEQ phrase2 guitar-06
      ( KEYMOD -12 ) ( TEMPO 0.32 ) ( KEYENV env22 )
      ( SUSTAIN 1.2 ) ( FINAMP 0.75 ) )
   )
   ( part3 ( PLAYTONESEQ phrase3 guitar-06
      ( KEYMOD -24 ) ( TEMPO 0.32 ) ( KEYENV env22 )
      ( SUSTAIN 1.2 ) ( REVERB ) ( FINAMP 0.75 ) )
   )
   [ ... etc ... ]
   ( part9 ( PLAYTONESEQ phrase9 guitar-06
      ( KEYMOD -12 ) ( TEMPO 0.64 ) ( KEYENV env22 )
      ( SUSTAIN 1.8 ) ( FINAMP 0.75 ) )
   )

   [ mixing goes here, plus final saves ]
)
```

You execute the wavspec, listen to the result, and decide (for instance) that the whole piece is a little too slow. Maybe TEMPO 0.40 would be better? Also, that guitar-06 sounds a little funky – maybe guitar-07 would be brighter.

You could make the desired changes by manually editing ALL of these PLAYTONESEQ commands – but the above GLOBAL setting commands make the job much easier. Consider the equivalent SYNTHCMDS section on the following page:

```
( SYNTHCMDS

   ( GLOBAL_WAVALIAS guitar guitar-06 )
   ( GLOBAL_SETTEMPO 0.32 )

   ( part1 ( PLAYTONESEQ phrase1 guitar
      ( KEYMOD -12 ) ( KEYENV env22 )
      ( SUSTAIN 1.8 ) ( FINAMP 0.75 ) )
   )
   ( part2 ( PLAYTONESEQ phrase2 guitar
      ( KEYMOD -12 ) ( KEYENV env22 )
      ( SUSTAIN 1.2 ) ( FINAMP 0.75 ) )
   )
   ( part3 ( PLAYTONESEQ phrase3 guitar
      ( KEYMOD -24 ) ( KEYENV env22 )
      ( SUSTAIN 1.2 ) ( REVERB ) ( FINAMP 0.75 ) )
   )
   [ ... etc ... ]
   ( part9 ( PLAYTONESEQ phrase9 guitar
      ( KEYMOD -12 ) ( TEMPO 2.0 ) ( KEYENV env22 )
      ( SUSTAIN 1.8 ) ( FINAMP 0.75 ) )
   )

   [ mixing goes here, plus final saves ]
)
```

Notice that the result is the same as before. The "`guitar`" is still "`guitar-06`", and the `TEMPO` is still 0.32 for all phases except "`part9`", which was always twice as fast (0.64).

To effect the desired change, you need only edit the first two lines to read

```
( GLOBAL_WAVALIAS guitar guitar-07 )
( GLOBAL_SETTEMPO 0.40 )
```

Similarly, you can transpose the entire song up or down (in semitones) by issuing

```
( GLOBAL_KEYSHIFT I_semitones )
```

and you can lengthen or shorten the note-sustains in these phrases, without altering the tempo, by issuing

```
( GLOBAL_KEYSUSTAIN R_sustainfactor )
```

Take note that the commands `GLOBAL_SETTEMPO`, `GLOBAL_KEYSHIFT`, and `GLOBAL_KEYSUSTAIN` each have an effect that applies together with whatever specification (if any) was included with the subsequent `PLAYTONESEQ` commands.

In contrast, the `GLOBAL_PLAYEFFECT` command, which can be assigned to any of the values `REVERB`, `AFTREVERB`, or `NOREVERB`, only affect affects the subsequent commands that do not themselves specify one of these play effects.

Each of these commands can be issued as many times as you like in the SYNTHCMDS section, and all subsequent PLAYTONESEQ and PLAYPERCSEQ commands will be affected.

Also, you can have multiple GLOBAL_WAVALIAS commands in effect at the same time. This is especially useful if you want to change "guitar-06" to "violin-04", but the violin requires a different KEYENV (to simulate a "bowed" effect, rather than a plucked or strummed effect). For instance, you might specify

```
( GLOBAL_WAVALIAS mainInstrument guitar-06 )
( GLOBAL_WAVALIAS mainKeyEnv env22 )
```

and issue PLAYTONESEQ commands like

```
( part1 ( PLAYTONESEQ phrase1 mainInstrument
   ( KEYMOD -12 ) ( KEYENV mainKeyEnv )
   ( SUSTAIN 1.8 ) ( FINAMP 0.75 ) )
)
```

Now, if you want to try the violin, you need only edit the GLOBAL aliases to read

```
( GLOBAL_WAVALIAS mainInstrument violin-04 )
( GLOBAL_WAVALIAS mainKeyEnv bowenv19 )
```

5.1 Process Status and Information Commands

5.2.1 TELLWAV

Format: (`TELLWAV wavtag [I_mode] )`

This command describes the characteristics of the loaded wavedata indicated by the supplied `wavtag` name. The values of `I_mode` may be 1, 2, 3, 4, or INFO. If omitted, a value of 1 is assumed. The result is written to stdout (and captured by the "`runlog.txt`" file described in Section 1 – Overview of WavSynth).

```
( TELLWAV wavtag )
( TELLWAV wavtag 1 )
```

This mode will report the bitdepth (8 or 16 bit samples), whether the wavedata is mono or stereo, the duration in seconds, and sample rate in samples per second.

```
( TELLWAV wavtag 2 )
```

In addition to the report of mode 1, this mode will yield several statistics about the wavedata, notably the mean amplitude, the standard deviation in amplitude, and the minimal and maximal values of the amplitude.

```
( TELLWAV wavtag 3 ) and ( TELLWAV wavtag 4 )
```

In addition to the report of modes 1 and 2, these modes will output the actual sample values, in hexadecimal bytes (mode 3) or in decimal integers (mode 4). **BEWARE** – these modes will produce very large output files!

```
( TELLWAV wavtag INFO )
```

This mode will only report the embedded text info of the WAV file (if any). See Appendix B, Embedded Test Information for details.

5.2.2 TELLWAVBANK

Format: (`TELLWAVBANK )`

This command will list the index number of each occupied wavbank entry, and where (as it should) each entry has associated wavedata and an associated tagname. In addition, for each entry that has associated wavedata, the hex memory address of that data is listed. This is useful for determining if one has inadvertently pointed two different names to the same wavedata location.

5.2.3 TESTPARENS

This is not actually a WavSynth command to be placed into a wavspec file, but rather an option you can place on the commandline to the WavSynth program itself.

If you edit the batch file "runWavSynth.bat" and add a "-D" option to the command line,

```
wavsynth –D –i samplewavspec.txt > runlog.txt
```

then, rather than actually process your wavspec file, the WavSynth program will analyze the occurrences of parentheses, tell you if there is a miscount of matching parentheses.

5.3 EXIT

Format: `( EXIT )`

This command is optional. If placed among a sequence of WavSynth commands in the `SYNTHCMDS` section, it will cause all subsequent commands to be ignored and the program to exit. This can be useful when debugging problems, or if you wish to experiment with the earlier commands, and not have to wait for all of the other commands to execute each time you save and process your project.

Alphabetic Index of WavSynth Commands and Terms

APPENDIX A: Background on WAV File Structure and Detail

1. Background on WAV files

Standard WAV files specify the values of a waveform precisely as it unfolds in time, digitally accurate to a resolution specified by two fundamental parameters:

1. Sample Rate: The number of samples recorded per second
2. Bit Depth: The range of amplitude values a sample may assume

The sample rate is a measure of the frequency resolution of the waveform storage. Human hearing can detect sounds as high as 20,000 Hz (oscillations per second), however above 10,000 Hz, the ability to distinguish the waveform (say, flute versus horn) tends to disappear. Thus, only about 4 numbers per oscillation are needed at the highest frequency in order to perceive the oscillation rate (effectively, the "note") itself. For this reason, high-fidelity WAV files most often use a sample rate of 44,100 samples per second.

In other words, one second's worth of sound is represented as a stream of 44,100 separate numbers, each number specifying a particular amplitude of the ensemble waveform at a particular moment during that second.

The bit depth specifies the size of number that is used to represent each sample value. With a bit depth of 8, one is using "one byte" per sample, and is limited to representing only one of 256 possible amplitude values at each sample point. For high fidelity sound, it is common to use two bytes (16 bits) per sample. This allows each sample point to specify one of 65536 possible amplitude values.

Most interesting musical compositions and sound effects require a stereo separation. Thus a WAV file holding stereo information must contain samples for both the left and right channel.

Taken together, each second of high fidelity stereo WAV storage requires 44,100 2-byte samples for each of the two stereo channels, resulting in an approximate file size of

176,400 bytes per second of stereo sound.

Or about 6 seconds of high fidelity sound per megabyte.

Irrespective of the Sample Rate and Bit Depth of an existing WAV file, the WavSynth program will load the file and convert ALL samples to floating point decimal "real" values, with amplitudes normalized (initially) to [-1.0,1.0]. As well, wave forms that are created in WavSynth are produced and managed in this internal format in order to minimize rounding errors and unwanted artifacts when many effects are employed.

WavSynth will output finished WAV files as you require, defaulting to 44,100 samples per second, 16-bit stereo if otherwise unspecified.

1. WAV file Structure Detail

In the foregoing description, I will indicate the byte-positions in the binary WAV file as decimal integer addresses, with 0 being the address of the first byte. For those positions where the value in the file is fixed by standard, I will write the literal value in square braces, as in [value]. If the value is a variable integer value, I will write [int4] to represent a 4-byte integer, [int2] for a 2-byte integer, and [byte] for a single byte value.

NOTE: Other than literal strings embedded at certain points in the binary WAV file, all multi-byte integer values are stored in reverse-byte order. The following table illustrates this for several numbers, represented in ordinary decimal value, in "natural" multi-byte hexadecimal (base 16) form, and in reverse-byte format, for a 4-byte (32-bit) integer.

Decimal	Natural Hexadecimal	Reverse-Byte Hexadecimal
1	[00 00 00 01]	[01 00 00 00]
2	[00 00 00 02]	[02 00 00 00]
10	[00 00 00 0A]	[0A 00 00 00]
15	[00 00 00 0F]	[0F 00 00 00]
16	[00 00 00 10]	[10 00 00 00]
255	[00 00 00 FF]	[FF 00 00 00]
256	[00 00 01 00]	[00 01 00 00]
65535	[00 00 FF FF]	[FF FF 00 00]
65536	[00 01 00 00]	[00 00 01 00]

Pos	Description	Value and/or Format	
0	File Type Identifier	[RIFF]	(literal string "RIFF")
4	Total File Size in bytes	[int4]	
8	More Identity Standard	[WAVEfmt]	(note the trailing space)
16	Remaining Header bytes	[int4]	(bytes until "data" marker)
20	????	[int2]	(always seen it as value 1)
22	Number of channels	[int2]	(either 1 or 2)
24	Sample Rate	[int4]	(8000, 44100, etc)
28	Bytes per Second	[int4]	(chan*SampRate*BPSamp)
32	BPSamp * chan	[int2]	(BPSamp*chan)
34	Bit Depth	[int2] or [int4]	(BPSamp*8, thus 8 or 16)
36 or 38	Data Marker, DM	[data]	literal string "data"
DM+4	Actual Data Length	[int4]	number of bytes of wavedata
DM+8	The wavedata	(see description below)	

Note: One is guaranteed to locate the start of the actual wavedata by reading the 4-byte integer at byte-position 16 (Remaining Header Bytes), and moving to file position 20 + (Remaining Header bytes) + 8. The 4-byte integer immediately preceding this position gives the total number of wavedata bytes to expect.

The bytes of the wavedate section express the actual amplitude values of the waveform as it unfolds in time, with amplitude specified for each sample. In the case of low-fidelity 8-bit (1 byte) samples, each amplitude is expressed by a single byte, whose value can range from 0 to 255, with 0 representing max-negative amplitude, 128 representing 0 amplitude and 255 representing max-positive amplitude. Thus, if a sample is loaded as a real floating-point decimal number, subtraction of 128.0 and division by 128.0 will result in A real amplitude between -1.0 and 1.0. The following figure depicts the relationship

```
Normalized Amplitude:  -1.0 - - - - - - - - - 0.0 - - - - - - - - - 1.0
In Byte Storage:       [00][01] ...       [7F][80][81] ...          [FF]
```

Using high-fidelity 16-bit samples, each sample is represented by a 2-byte integer, whose value can range from 0 to 65535. Here, the WAV standard is to represent 0 amplitude with a 16-bit 0 value, max-positive amplitude with 32,767, and max-negative amplitude with 32,768. The following figure depicts the relationship to normalized amplitudes.

```
Normalized:  -1.0 - - - - - - - - - - - - 0.0 - - - - - - - - - - - - 1.0
ByteStorage: [80 00] . . .    [FF FF] [00 00] [00 01] . . .     [7F FF]
```

The format of the actual wavedata bytes is as follows. Note that when I now write byte-position address, it references only the wavedata section. Thus, position '0' is actually file position DM+8, etc. Each row represents one "sample" of time.

IF 16-bit Stereo:	Pos	4 Bytes	Description
	0	[L L][R R]	2-byte left chan, 2-byte right chan
	4	[L L][R R]	2-byte left chan, 2-byte right chan
	8	(etc)	

IF 8-bit Stereo:	Pos	2 Bytes	Description
	0	[L][R]	1-byte left chan, 1-byte right chan
	2	[L][R]	1-byte left chan, 1-byte right chan
	4	(etc)	

IF 16-bit Mono:	Pos	2 Bytes	Description
	0	[B B]	2-byte sample
	2	[B B]	2-byte sample
	4	(etc)	

IF 8-bit Mono:	Pos	1 Byte	Description
	0	[B]	1-byte sample

1	[B]	1-byte sample
2	(etc)	

The byte-position immediately following the end of the wavedata bytes is what I call the position "tail". From experience, this can be left out entirely, although I have also seen the following:

tail	Tail section of wave file	[LIST]	string indicating tail section?
tail+4	Length of tail contents	[int4]	Length of remaining file

The tail often holds printable strings such as AuthorName, Copyright, etc.
See **APPENDIX B: Embedded Text Information** for details.

APPENDIX B: Embedded Text Information

When using the SAVE command, in addition to (optionally) specifying the number of channels, bytes-per-sample, and the sample rate, as given by the example

Example: `( SAVE wav1 mine.wav ( CHAN 1) ( RATE 8000 ) )`

you may also (optionally) add a structured section named "INFO", supplying informational tags and their associated values. For clarity, I will show an example in a multi-line format, extending the above example:

```
( SAVE wav1 mine.wav
      ( CHAN 1)
      ( RATE 8000 )
      ( INFO
            ( IART John Doe Sr. )
            ( ICRD 2004-04-30 )
            ( INAM The Battle of Wits in E Minor )
            ( ISFT WavSynth Version 1.0 )
      )
)
```

NOTE: Only the informational tags that appear will overwrite those that existed in the file originally. Thus, if having saved this file, you decide that you need to change the creation date to 2004-05-18, you need only write

```
( SAVE wav1 mine.wav
      ( CHAN 1)
      ( RATE 8000 )
      ( INFO ( ICRD 2004-04-30 ) )
)
```

To avoid having the new SAVE promote the file to stereo, 44,100 samples per second, 2-bytes per sample format, you do need to reiterate CHAN, BPSM, and RATE, or you may write

```
( SAVE wav1 mine.wav
      ( ORIG )
      ( INFO ( ICRD 2004-04-30 ) )
)
```

Recall from the explanation of the SAVE command, the "ORIG" parameter tells WavSynth to attempt to SAVE the file with CHAN, BPSM and RATE as they were when the file was loaded. This is only possible if the wavedata being saved is not the product of any intervening "NEW" commands.

The following page describes all of the WavSynth-recognized informational tags.

WavSynth Informational Tags

IARL	Subject archival location
IART	Identify the original artist
ICMS	Identify the commissioning agency
ICMT	Provide general comments (no newline characters)
ICOP	Identify copyright holder[; copyright holder]...
ICRD	Subject creation date in format YYYY-MM-DD
IENG	Identify the engineer(s) [name; name; name; ...]
IGNR	Describe the subject genre
IKEY	Embedded keywords [word; word; ...]
INAM	Title of the subject file
ISBJ	Subject of the file
ISFT	Identify the authoring software
ISRC	Identify provider of original source
ISRF	Describe original source medium [e.g., wax recording...]
ITCH	Identify the digitizing technician

APPENDIX C: WavSynth Efficiency and Chained Commands

Consider the difference between:

```
( wav1 ( SIMPFORM SAW 100 0.9 1.0 ) )
( wav2 ( SMOOTH wav1 0.3 ) )
( SAVE wav2 sample.wav )
```

and

```
( wav2 ( SMOOTH ( SIMPFORM SAW 100 0.9 1.0 ) 0.3 )
( SAVE wav2 sample.wav )
```

Both set of commands will produce exactly the same wavedata. But each form has its advantages and disadvantages. To explain, let us introduce the terms "homed" data and "homeless" data.

The command `( wav1 ( SIMPFORM SAW 100 0.9 1.0 ) )` calls upon the `SIMPFORM` function to produce some wavedata, and then gives it a **home** in memory under the name "wav1". The command `( wav2 ( SMOOTH wav1 0.3 ) )` calls upon the `SMOOTH` function, which seeks and finds the **homed** data "wav1", produces a copy of it which it then smooths and returns. That "smoothed" copy is then given a **home** under the name "wav2". The data which the `SMOOTH` function had acted upon (wav1) is left in memory for use by later commands.

In contrast, when `( wav2 ( SMOOTH ( SIMPFORM SAW 100 0.9 1.0 ) 0.3 )` is executed, the `SMOOTH` command sees that it must call upon a data-creation command (`SIMPFORM`) in order to begin processing. The wavedata it receives from `SIMPFORM` is thus **homeless** data. It is in memory only temporarily. It takes this data, and produces a smoothed copy of it. That copy gets a **home** named "wav2", but the **homeless** input data (received directly from `SIMPFORM`) is automatically freed from memory.

In some cases, the efficiency is further enhanced, depending upon the operation. While `SMOOTH` is an operation that MUST produce and operate upon new wavedata, for those that have both a "SELF" and a "NEW" form (e.g., `NORMBT` and `NEWNORMBT`), the "NEW" forms are "smarter" in this regard. Compare:

Case 1: `( wav9 ( NEWNORMBT wav1 0.25 0.75 ) )`
Case 2: `( wav9 ( NEWNORMBT ( SIMPFORM . . . ) 0.25 0.75 ) )`

In Case 1, `NEWNORMBT` knows it is operating upon **homed** data (wav1), and must not modify that data, so it norms a copy, and passes that copy **homeless** up to the top of the command, where it receives a **home** as wav9. In Case 2, it knows it is operating upon **homeless** data (via `SIMPFORM`), so it behaves like `NORMBT` and norms the **homeless** data itself, and indicates that it is returning **homeless** data to the top of the command (where it finally receives a **home** as wav9). This saves a malloc, a free, and system memory.

APPENDIX D: Predefined Fourier Coefficients

The following named Fourier Coefficients may be supplied to the `FOURIER` command. In all cases, coefficient "0" has value "0" and is not listed. Even the user-defined Fourier coefficients will take the supplied values to begin at index 1.

Where there are less than 20 defined coefficients, you may still supply the command to ask for more than appear, but only the actual number that are available will be used. Most are defined to 20 values, even though not all are represented in this table.

Set_Name	Coefficients
INTEGERS	1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, …
ODDS	1, 3, 5, 7, 9, 11, 13, 15, 17, 19, 21, 23, 25, 27, …
HALVES	1, 1.5, 2, 2.5, 3, 3.5, 4, 4.5, 5, 5.5, 6, 6.5, 7, …
PRIMES	1, 2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, …
SIMPLEX2	1, 3, 6, 10, 15, 21, 28, 36, 45, 55, 66, 78, 91, …
SQUARES	1, 4, 9, 16, 25, 36, 49, 64, 81, 100, 121, 144, …
CUBES	1, 8, 27, 64, 125, 216, 343, 512, 729, 1000, 1331, …
POWEROFRT2	1, 1.41421456, 2, 2.8284271, 4, 5.65685425, 8, …
POWEROF2	1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024, 2048, …
POWEROF3	1, 3, 9, 27, 81, 243, 729, 2187, 6561.
FIBONNACCI	1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, …
UNITPATHSR1	1, 2, 2, 3, 4, 5, 7, 8, 13, 13, 22, 21, 39, 34, …
FACTORIAL	1, 1, 2, 6, 24, 120, 720, 5040.
TRIPOLENETS	1, 1, 2, 2, 4, 6, 11, 18, 36, 60, 135.
MAXCYCPERMS	1, 2, 3, 4, 6, 6, 12, 15, 20, 30, 30, 60, 60, 84, …
USPCU	1, 2, 3, 4, 6, 9, 11, 17, 23, 30, 44, 60, 80.
MPRODINTP	1, 2, 3, 4, 6, 9, 12, 18, 27, 36, 54, 81, 108, 162, …
ALTODDS	1, -3, 5, -7, 9, -11, 13, -15, 17, -19, 21, -23, …
TWOOFTHREE1	1, 3, 4, 6, 7, 9, 10, 12, 13, 15, 16, 18, 19, 21, …
TWOOFTHREE2	1, 2, 4, 5, 7, 8, 10, 11, 13, 14, 16, 17, 19, 20, …

CLOSURE: Contact the author of WavSynth

Send mail to psytek@psychotechnica.org

I will do my best to answer any questions that may arise.