The SNDSYS tutorial

Volker Schatz

October 10, 2007

Contents

1 What's SNDSYS?

SNDSYS is a stream-based digital signal processing system designed for synthesising and processing digitised sound. Its successive processing elements, called "objects" or "sndobj's", read and process each other's output one by one. SNDSYS was written in C, and users wanting to expand it by defining their own sndobj's will not get around knowing that programming language. Even for merely using what is there, you will have to write C programs. But the syntax is fairly basic, and will be introduced step by step in the first chapters of this tutorial.

Rather than a shared library, SNDSYS is simply a set of C functions which can be compiled together with your own program. Due to the fact that I continue to modify it as I see fit, programs which use one version may not be 100% compatible with later versions. For that reason there are no version numbers yet; instead, the `tar` archive in which SNDSYS can be downloaded from my homepage, `www.volkerschatz.com/noise`, is named after the date at which it was created.

This tutorial was written with two goals in mind: The first is to give folks who are new to it and are considering using it an introduction. The following five chapters present usage examples and tell you how to do with SNDSYS what you could probably do with any digital sound synthesizer. The chapters after that are about advanced features of SNDSYS which cannot easily be found elsewhere, at least not in a program designed to generate music. They serve the second aim of this tutorial: To present and interest people in techniques which are not commonly used for sound synthesis, with which I have been experimenting using SNDSYS. Prominent among these are the wavelet methods in Section 8. SNDSYS is primarily a test bench which provides scope for experimentation. This text explains what it is about.

2 Getting started

As I already mentioned in the introductory section, there is at present no way of using SNDSYS without writing a small C program (and there will not be until and unless someone writes a graphical front-end for it, which I unfortunately have no time to do right now). To give non-experts an easier start, a simple framework program is distributed with SNDSYS. Before we have a look at it, let us get over with the boring task of checking your system for requirements.

2.1 Requirements

What you absolutely need are a Linux or UNIX system, a shell, a text editor, a version of the `make` utility and a C compiler. If you are condemned to working on a billionaire crashware platform, you might want to try Cygwin (`www.cygwin.com`), which provides a pretty UNIX-compatible environment without the need to install a separate operating system. Either way, some experience using UNIX is required, as we will be working from a shell command-line. From a graphical window manager, opening a terminal window (xterm, konsole or gnome terminal) will give you access to a shell, usually `bash`.

A `make` utility and a C compiler will usually be available on any Linux/UNIX system. As I use Gnu make, this is probably the safest choice if you have different incarnations of `make` available. For a compiler, I use the Intel compiler, which is free for private use under Linux. The Makefile automatically detects whether it is available and falls back to the Gnu C compiler `gcc` if necessary. If you use `icc` on a newish Intel processor, you will see bragging messages about loops being vectorised. Before I switched them off, `gcc` churned out lots of spurious warning about unreachable code because it is too dumb to comprehend `if` constructs. To each its own.

Lastly, if you want to compile all of the example programs, you will need some libraries and the corresponding header files. An absolute requirement is the math library. It is part of the C library and therefore should be installed if you have a compiler, including header files. Two other libraries can be done without if necessary, but you will have to remove them from the Makefile variable

LIBS, comment out the object sndthread from sndmisc.c and will not be able to compile the piano program: the POSIX threads library (libpthread.so.0, for sndthread) and the X11 library (libX11.so.?, for piano). Neither should be a problem, as the libraries themselves should be present on any UNIX-like system, but possibly you have to install the header files which many Linux distributions store in different packages with names ending on ...-dev or ...-devel.

2.2 Useful supplementary software

As most of our example SNDSYS programs write their output to a file, you are recommended to use a wave file player of your choice. (Though SNDSYS can also output sound via your soundcard, see Section 6.5.) The utility sox [sox] can convert sound formats and comes with a command-line wave file player.

Viewing generated waveforms can be done with a wave file editor. The most sophisticated one I know is snd [snd]; others are sweep [swe] and kwave [kwa].

2.3 A look at a SNDSYS program

Now the arduous technical details have been dealt with, let's have a look at the example source file. It is called frame.c and contains the following lines of code:

```
#include "sndsys.h"

int main(int argc, char *argv[])
{
  sndobj *output;
  double duration;

  output= c(0);
  duration= 2.0;
  sndexecute( 0, duration, writeout("frame.wav", output));
  return 0;
}
```

The first line includes the SNDSYS header file which tells the compiler about all the SNDSYS functions available. You have to include this line whenever you want to use SNDSYS. Next comes the definition of the main program, starting with the declaration of its arguments. If you know C, you will know what they mean, otherwise ignore them. The opening brace which starts the main program's code is followed by the declaration of two variables. The first has the type "sndobj *", which is the type we use to denote SNDSYS objects. The second is a double-precision floating-point variable whose only role is to store, and give a name to, the duration of the sound we want to generate.

The core of the program is really the first line of code after the variable declarations. The variable output is assigned the return value of a function c, which takes a numerical argument we set to zero. This function is the simplest

example of a sndobj, as it outputs the constant which is its argument over and over again. At this point I want to make clear an important distinction in SNDSYS: Its fundamental data type, the `sndobj *`, is really a rather complex thing. You can think of it as a stream of data values, as a waveform, a signal, or as a tone or melody. On the other hand, we will also need plain and simple numbers which we give a certain value but which do not change at all during the runtime of the program, when our sounds are generated. **The constant sound object `c(0)` and the constant number 0 are not at all the same thing!** The former is a data source which outputs zeros ad infinitum, while the latter is just one zero constant. It's the same difference as between a printer and a piece of paper with a number on it.[1] It is very important to be aware of this difference because more complicated objects will have both constants and signals (denoted by the objects which output them) as arguments. If you ignore it, the C compiler will punish you with lots of obscure error messages.

Back to our first simple example program. We have now defined what kind of sound we want to generate by writing the line `output= c(0);`. Some administrational tasks have to be performed for making the generated data available in a file. These are taken care of by the line starting `sndexecute...`. The `sndexecute` function, after some administrational stuff, simply makes a sound object generate output for the given duration, and throws that output away. Why? On the other hand, why not? You might want to observe the signal at different stages in the processing without running the program several times, or write different generated signals to different files. That is why the operation of output to a file is implemented as an object, `writeout`. It has two arguments: the name of the file and the signal which it is to write to that file. The `writeout` object itself serves as an argument to the function `sndexecute`. Its output is in the widespread wave file format, which can be played by many software players. The range of floating point values which SNDSYS uses internally being mapped to the wave file value range is from -1 to +1. Any output outside this interval will be clipped to the maximum, which sounds nasty. So remember that whatever sounds you generate, the maximum value should not exceed ± 1. After the computation is complete, the `writeout` object prints out the maximal and minimal values it received for your information.

Why was the `writeout` object put inside the `sndexecute` function? There was no need to do that. One could equally well have written:

```
...
output= writeout("frame.wav", c(0));
duration= 2.0;
sndexecute( 0, duration, output);
...
```

Now the `output` variable denotes the output of the `writeout` object. This variable is then passed to `sndexecute`. The processing chain is the same

[1] Of course, at some level, the things output by a sndobj are numerical values, but if you want to know about those details, you should not read this tutorial but learn C and have a look at the source code.

as before: “First, generate a stream of zeros. Then, write them to the file frame.wav.” Only the point in the chain where we used the variable is different. Alternatively, we could have done completely without the variable:

```
...
duration= 2.0;
sndexecute( 0, duration, writeout("frame.wav", c(0)));
...
```

The original version of the `frame` program was written in a way that separates the generation of the sound and the writing of the wave file. Down below, we will modify the former while keeping the latter unchanged and will therefore always refer to that version. When you substitute different assignments of the `output` variable, you do not have to delete prior versions. The C programming language allows two different types of comments, which can be used to disable unused code. The first starts with “//” and extends up to the end of the line. To comment out the current assignment to `output`, prepend the comment sign:

```
//  output= c(0);
```

This kind of comment is easiest to use, but rather new, so that there may be compilers in existence which do not support it. Traditional C comments start with “/*” and end with “*/”. Their advantage is that they can extend over several lines. For instance, you could comment out both the assignments to `output` and `duration` (before adding new such assignments):

```
/*
  output= c(0);
  duration= 2.0;
*/
```

2.4 How to run it

To generate the sound described by frame.c, it has to be compiled and run. To do that, first change to the SNDSYS directory in a shell using the `cd` command. Then, execute the following shell command (the dollar sign denotes the shell command prompt):

```
$ make frame.wav
```

This will automatically compile frame.c along with SNDSYS, link the necessary libraries, and run the executable to generate the wave file. (If you create your own program, you have to add its name (without the `.c` extension) to the DISTTARGETS line in the Makefile to be able to use this automatism.) After running `frame`, you will see an output similar to the following:

```
Computation finished.  Time taken: 0.005 seconds for 88200 samples (2 seconds);
16961538 samples per second.
Writeout to 'frame.wav' finished.  Min= 0, max= 0.
```

This is the information output by the `writeout` object when processing has finished. As we have output a signal containing only zeros, both the minimum and the maximum are zero.

3 Generating simple sounds

In the following, we will finally see how to generate something other than constant zero ;). We will continue to use the `frame` program, but will substitute the line `output= c(0);`. I will only give the replacement of that line when presenting code. The rest of the framework program, including the `writeout` object, will always be assumed to be there.

3.1 Guess what – a sine

Let's generate a sine wave at the standard pitch, 440 Hz. This works as follows:

```
output= sine(0, c(440), c(1));
```

The `sine` object takes three arguments: a constant phase, the frequency and the volume. The phase is a constant. Passing it as 0 gives a sine wave, 0.25 would give a cosine. The other two arguments are themselves sndobj's. We just happen to have chosen them to be constant objects `c(...)` (Remember my remarks on the difference between constant numbers and constant sound objects?). Their output is input into the sndobj `sine` and determines its frequency and volume. We will therefore henceforth call them the inputs of the `sine` object.

You can display brief documentation for `sine` (and other sndobj's) with a Perl script which comes with SNDSYS. On a shell prompt in the SNDSYS directory, type

```
$ ./objectdoc.pl sine
```

(In fact, if your shell is any good, you will just have to type "`./o`", press Tab and type "`sine`".) The command generates the following output:

```
sndsource.c:
    sine (phase/2pi) <freq> <ampl>

Generates a sine oscillation.  The float argument phase_2pi determines the
initial phase divided by 2 pi (360 degrees).  The following argument, the first
input, is the frequency, the second the amplitude.

See also: harmonic, rect, saw

sndobj *sine( float phase_2pi, sndobj *freq, sndobj *ampl )
```

The first line gives the SNDSYS source file in which the object is defined. The second line displays a stylised argument list: arguments in parentheses are numerical constants, while arguments in angular brackets are inputs. (Objects can also have array arguments, which are displayed in rectangular brackets, and string arguments, which are put in quotes.) This has nothing to do with the C syntax, it is just my way of documenting the arguments. Then follows a description of what the sndobj does. The last line gives the C declaration of the object (for those who want to know the exact data types).

The sndobj descriptions are embedded in comments in the source code; all the script objectdoc.pl does is extract them and print them. At the moment, it can only be used from within the SNDSYS directory, because it has no way to find the source files otherwise. If you need to access it from any directory, and if you use the shell `bash`, you can define the following shell function (for instance in your .bashrc):

```
$ objectdoc() { ( cd /sndsys/directory && ./objectdoc.pl "$1"; ) }
```

You have to replace `/sndsys/directory` by the absolute path name of your SNDSYS directory.

Next to objectdoc.pl, which is the main documentation tool for SNDSYS objects, there is grepobj.pl. This script functions similarly to the UNIX tool `grep` or the manual reader `man` with the `-k` option. It searches the documentation of all sndobj's for lines containing the word which is its argument and prints the lines prepended by the name of the sndobj. This allows you to search for sndobj's related to a specific task. Remember that I use British spelling.

3.2 Other tones

Two more objects which generate simple tones are `rect` and `saw`. They take the same arguments: A parameter which defines the starting phase and three inputs giving the frequency, amplitude and shape of the output. The "shape" input is called "ratio" and determines the ratio between the positive and negative part of the rectangular signal and the rising and falling part of the sawtooth (though it is not actually equal to that time ratio numerically). For example, to generate a symmetric rectangular waveform, you would write:

```
output= rect(0, c(440), c(1), c(0));
```

For generating a sawtooth with rising slopes and immediate downward jumps, do the following:

```
output= saw(0, c(440), c(1), c(1));
```

The following generates a rectangular wave which is mostly negative with the exception of one positive value per period:

```
output= rect(0, c(440), c(1), c(-1));
```

It contains more high frequencies than the symmetric rectangular wave and sounds sharper.

3.3 Random noise

Beside tones, random noise is frequently used. There are two basic noise generators in SNDSYS, `flatrand` and `gaussrand`. Both take one argument determining the volume. In the case of `flatrand`, the argument is the actual maximum amplitude of the output signal. For Gaussian noise, the output value can in principle be arbitrarily large, though with exponentially decreasing probability. For estimating the amplitude of Gaussian noise, it is useful to know that only 0.3% of the values will lie outside an interval of three times the mean deviation σ (which is the argument of `gaussrand`). So if you want a Gaussian random output with a certain "maximum" value, choose σ as 1/3.5 to 1/3 times the maximum. For instance, Gaussian noise in the interval $[-1, 1]$ can be produced by:

```
output= gaussrand(0.3);
```

3.4 Modulation – trees of sndobj's

Generating tones with constant pitch and volume is nice, but loses its appeal pretty quickly. Fortunately, varying them is easy. Remember how we discussed, in Section 2.3, the difference between constant arguments and inputs representing data streams? If not, going back and rereading a few sentences around the bold-face text might be a good idea. The frequency and volume arguments of `sine` and other waveform generators are inputs — they can receive the output of any sound object, which may vary with time. To produce a siren with periodically varying pitch, we code the following:

```
output= sine(0, linear(440, 40, sine(0.25, c(1), c(1))), c(1));
```

This code fragment contains three sndobj's: a `sine` which generates the tone, a `linear` object which we will come back to, and another `sine` which generates the variation in pitch. The latter `sine` has a frequency of one Hertz and a starting phase of 1/4 period, which means it is actually a cosine.

The `linear` object is used to solve the problem that a sine or cosine always has zero crossings. We want to vary the generated frequency in a certain range, but certainly do not want to go down to zero Hertz. The `linear` object multiplies its second constant argument with its only input (third argument), adds the first argument, and outputs the result to the frequency input of the `sine` generating the tone. This causes the frequency of the siren to vary between 400 and 480 Hz. Briefly and mathematically put, `linear(`a`, `b`, input)` gives $a + b \cdot$ `input`. `linear` is a very simple object, but is used frequently for similar purposes as here.

Similarly, a tone's amplitude can be modulated to generate a simple tremolo effect. Using a rectangular waveform this time, the source code fragment is:

```
output= rect(0, c(440), linear(0.8, 0.15, sine(0, c(6), c(1))), c(0));
```

Obviously, the only difficulty is getting the matching of the parentheses right. Use an editor intended for editing source code, which will highlight matching parentheses, such as NEdit, emacs or vi (for hackers).

3.5 Chords – adding up

After being content with single tones for so long, let us now learn how to generate chords. The most obvious way to do this is to use the `add` object. Its first argument is the number of inputs to add up, followed by the inputs themselves.[2] You add up three sine waves in the following way:

```
output= add(3, sine(0, c(440), c(0.3)), sine(0, c(550), c(0.3)),
               sine(0, c(660), c(0.3)));
```

There is no harm in continuing the instruction on a second text line, as C is not a line-oriented language. We have added together three sine waves with different pitches to give the triad of A major. The amplitude was reduced so the sum does not exceed 1.

This way of creating a chord works well so long as the pitch is constant. If one wanted to create a simple instrument playing the same chord with a varying root note, however, this would be a little complicated. One would have to compute the other two pitches from the root pitch using `linear` and feed then into the relevant frequency inputs of the `sine`s. There is a simpler way. Because chords are frequently used, there exists an object generating a chord of sine waves with a variable root frequency and amplitude: `harmonic`. Its first argument is a table determining the notes in the chord, followed by the frequency and volume inputs. View the documentation of `harmonic` with the script objectdoc.pl.

We have to learn a new feature of the C programming language to define the table. We will confine ourselves to declaring and initialising a table, which will then remain constant. Below the declaration "`double duration;`" insert the following line into our framework program:

```
float chord[]= { 1, 0.3, 0, 1.25, 0.3, 0, 1.5, 0.3, 0, END };
```

Briefly explained, `float` defines the type of the table's contents (single precision floating point), the brackets indicate that an array rather than a single value is being declared, and the numbers in curly braces are the numbers with which the table is initialised. (Because of this initialisation, the length of the

[2] It is vitally important that the number of input arguments is at least the number given in the first argument. Otherwise the program will probably crash. Program crashes are nothing horrible, no worse from a practical point of view than the program generating something different from what you want or otherwise malfunctioning. To allow the compiler to do this kind of consistency check for you and prevent such crashes, SNDSYS would have to have been written in C++, but it wasn't.

array need not be given between the brackets.) The special value "END" has nothing to do with the C syntax but is defined in the include file sndsys.h. It tells the harmonic object that no more notes follow.

harmonic interprets the note table in the following way: Every set of three successive values defines one of the notes in the chord. The first value gives the frequency of the note relative to the root frequency. The second is the volume of the note relative to the volume which harmonic receives from its input. The third is the starting phase, as in the first argument of sine. Giving the value END for the frequency concludes the table.[3]

Generating the same triad as above with the harmonic object and the table we just defined works like this:

```
output= harmonic(chord, c(440), c(1));
```

The constant object c(440) gives the root pitch of the chord. You can easily vary the pitch or amplitude as we did in Section 3.4.

4 Modifying sounds

This section treats all kinds of filters which influence the sound of waveforms. The explanations will be less verbose than in the previous section, as the general usage of sndobj's should by now have become clear. Some objects will only be mentioned; you can read their documentation with objectdoc.pl to find out more. The code examples given will be fewer and designed mostly to give you an example of reasonable parameter values as a starting-point for playing around yourself.

4.1 Reading a wave file

This section does not yet present a filter; instead, it introduces a sndobj which will allow you to apply them to your favourite song: the file object, which can read a .wav file. Its simplest usage is:

```
output= file("filename.wav", NULL);
```

If you put this line into frame.c, it outputs the first two seconds of the input file. To process a longer interval, edit the value assigned to duration.

The file object can also read waveforms in different formats (such as plain text files containing time-value pairs or simply values) and recognises the file type by the file extension. So your wave file has to have the extension .wav.

The second argument, NULL, tells file that its optional input is unused. This is something of an exception. Most sndobj's have only required inputs and will (you guessed) crash if you replace one with NULL.

[3] It is vitally important that you do not miscount and put the END in place of the volume or phase instead of the frequency. Otherwise the program may crash. If that worries you, add three END values instead of just one.

4.2 Highpass and lowpass

The simplest and most frequently used objects for altering sounds are `highpass` and `lowpass`. They have two inputs: the cutoff frequency which separates the blocked frequencies from those which can pass through; and the signal the filter should be applied to. These two filters have a very smooth frequency response, that is their attenuation does not change steeply even in the vicinity of the cutoff frequency. As a consequence, they do not change sounds radically, merely tint them brighter or darker. Try them out on a wave file of your favourite song:

```
output= lowpass(c(300), file("song.wav", NULL));
```

SNDSYS provides other high- and lowpass filters which allow more precise control and change sounds more strongly. A very useful one is `shelve`, a shelve highpass and lowpass filter rolled into one. It has an additional input which gives the relative amplitude of very high compared to very low frequencies.[4] The filter is normalised in such a way that the output never exceeds the input in amplitude — only the frequencies which should not pass are attenuated. This object has a less smooth frequency response than the simpler `highpass` and `lowpass`. The attenuation is almost constant outside a neighbourhood of the cutoff frequency (hence the name), where it changes comparably rapidly. This makes it suitable for shaping sounds. Consider the following code fragment, which builds a band-stop filter from two `shelve` objects and makes the random noise from `flatrand` sound very different:

```
output= shelve( c(50), c(10), shelve( c(3000), c(0.1), flatrand(1)));
```

Compare it with the unchanged output of `flatrand`!

I would like to mention two other sndobj's which fall into the category of high- or lowpass: `dcblock` is a DC blocker, a very tolerant highpass designed to block only the very lowest frequencies. It is useful when feedback loops cause the long-time average of a signal to drift, as sometimes happens. `avg` is a moving average filter, which has a rather radical lowpass behaviour. It has an input which allows to mix the filtered signal with the unchanged signal, and even create a highpass by subtracting the filtered signal.

4.3 Peak and notch filters

SNDSYS also provides filters which emphasize frequencies close to a given centre frequency. The most versatile is `peaknotch`, which can also attenuate a frequency. Its inputs are the centre frequency of the peak or notch, the amplitude gain (see the footnote in the previous section for how to convert it from dB), the width of the peak and the signal. If the gain factor is < 1, the filter is a notch filter, otherwise a peak filter. Experience shows that the

[4] Because I am a physicist rather than a sound engineer, this input is not in dB ;). One dB amounts to a factor of 1.122 in the amplitude, so to obtain an attenuation of x dB, you have to set the <`ratio`> to $1.122^{\pm x} = \exp(\pm 0.115\, x)$ (+ for a highpass, − for a lowpass).

notches are less pronounced than the peaks for the same bandwidth — so it makes sense to choose larger widths when using `peaknotch` as a notch filter. If you have access to the sound editor `snd`, apply `peaknotch` to white noise generated with `flatrand` and view the frequency spectrum with `snd`'s frequency display. Choose a large window size in the Transform Options dialog to get better frequency resolution.

Two other filters which can emphasize the frequencies close to a given centre frequency are `peak` and `formant`. Besides the signal input, both have inputs for the centre frequency and the width of the peak. Only the kind of bandwidth input differs: `formant` has an input for the actual bandwidth in Hertz, `peak` has an input I called "sharpness", which causes the peak to become narrower as it approaches 1. While their frequency response is similar for corresponding parameters, their range of parameters makes `peak` more suitable for only slight emphasis on a certain frequency, and `formant` better for narrow resonances. (To achieve the respective opposite, you would have to pass `peak` a sharpness above 0.95, and give a bandwidth of 1 kHz or above to `formant`.) For instance, `peak` lends itself well to a "wahwah" effect. Try it out on a wave file of a song:

```
output= peak(c(0.95), linear(800, 200, sine(0, c(2.1), c(1))),
                      file("song.wav",  NULL));
```

`peak` and `formant` differ from `peaknotch` in a similar way as `highpass` and `lowpass` differ from `shelve`. The former filters affect frequencies across the whole spectrum, while the latter's frequency response is constant outside a certain range.

4.4 General filters

This section presents how you can implement any FIR or IIR filter you find in a book or on the Web. For that purpose, SNDSYS provides the `filter` object. It has two array arguments giving the backfeed and forwardfeed. Both have the same format. They contain pairs of values, the first of which is the delay given in terms of sample values and the second the coefficient. The list is concluded with the special value `END`. Suppose you want to use a filter with the following difference equation:

$$y(n) = 0.01\ x(n) + 0.002\ x(n-1) + 0.99\ y(n-1)$$

A difference equation gives the prescription of how to calculate the current output value $y(n)$ from previous output values and current and previous input values $x(\ldots)$. The coefficients of $x(n)$ and $x(n-1)$ are called the forwardfeed coefficients, while in this example the coefficient of $y(n-1)$ is the backfeed coefficient. You program this filter in SNDSYS as follows:

```
float forwfeed[]= { 0, 0.01, 1, 0.002, END };
float backfeed[]= { 1, 0.99, END };

output= filter(forwfeed, backfeed, file("song.wav", NULL));
```

(I am assuming here that you want to apply the filter to a song stored in a wave file.) At this point two issues with minus signs should be mentioned. The delay values in the arrays are always positive; if you have an $x(n-5)$ in the difference equation, the delay value will be 5. The coefficients have to be given including their sign. You may come across difference equations with terms like $-a_3\ y(n-3)$, with a_3 given separately. In this case, the coefficient is $-a_3$ as far as `filter` is concerned.

The `filter` object gives you the opportunity to define filters with constant coefficients. This may mean that you have to recalculate the filter coefficients when you want to change its parameters, such as the cutoff frequency of a low- or highpass. The `filter` object can be used to create filters whose coefficients change depending on inputs — and in fact that is how many of SNDSYS's filters work. But in order to do that, you have to do some "genuine" C programming, and your best guide is the implemenentation of the filters in sndfilt.c.

4.5 Reverberation and basic mixing

Next to high- and lowpasses and other filters modifying the frequency spectrum of a sound signal, reverberation is probably the most widely used kind of sound modification. I have implemented several reverberation objects, most based on algorithms I found on various websites. Probably the most well-known is `freeverb`, which is also implemented in other sound processing tools, such as snd and csound. Unlike the sndobj's we have been dealing with previously, it produces stereo output. However, this does not require any change to our example program, as the `writeout` object automatically knows how to deal with it an writes a stereo wave file. See Section 6.3 for more on stereo signals.

`freeverb` has four more inputs next to the actual sound signal. They determine the length of the reverberation, the balance of high and low frequencies, the proportion of reverberation which crosses between the left and right channel, and the amount of reverberation to mix into the signal. Read the documentation (to be obtained via objectdoc.pl as always) and start playing around. For instance, try this out on your favourite song:

```
output= freeverb(c(0.4), c(0.5), c(0.1), c(0.2), file("song.wav", NULL));
```

Two other reverberation objects output only the actual reverberation sound, which has to be scaled down and added to the unmodified signal. They are called `nrev` and `josrev`. Let us look at `josrev`[5] first, which has only one input in addition to the sound signal, namely the cutoff frequency for a lowpass used internally. We can get a strong reverberation by mixing its output 50/50 with the original signal. How do we do that? We could write:

```
output= linear(0, 0.5, add(2, file("song.wav", NULL),
                josrev(c(300), file("song.wav", NULL))));
```

[5]It is named after Julius O. Smith because it is based on a suggestion in one of his online books [jos].

There is another way which saves us some typing: We can declare a variable similar to the `output` variable, only for the `file` object's output.

```
sndobj *thefile;

thefile= file("song.wav", NULL);
output= linear(0, 0.5, add(2, thefile, josrev(c(300), thefile)));
```

There is also an internal difference between this formulation and the one above, which makes this one strongly preferable: If we use two `file` objects, the calculations needed to deliver the file's contents will be duplicated, while only the values will be duplicated if we use a variable. That does not matter much in the case of a `file` object, but if you generate a complicated sound to which you later apply a reverberation, the first version may slow you down by almost a factor of two. So if you need to feed the same signal into two different inputs, declare a variable, assign the signal to it and pass that variable to the relevant inputs.

Now what if you want to mix a smaller amount of reverberation into a sound signal? Of course you could scale the reverberation with `linear`, then `add` it together with the original signal, and perhaps scale the result again... but there is a better way. The sndobj for mixing two signals is `mix2`. Its first input <gate> determines how the following two signal inputs are mixed. If <gate> is 1, only the first signal is output, if it is 0, only the second signal — the idea is that <gate> is the coefficient of the first signal. Let us try it out together with `nrev`:

```
sndobj *thefile;

thefile= file("song.wav", NULL);
output= mix2(c(0.9), thefile, nrev(5, 0.7, 0.2, thefile));
```

Note that the parameters of `nrev` are constants, not inputs, and that the middle parameter is a filter coefficient, not a lowpass cutoff frequency.

Two more reverberation objects shall be mentioned. Both are experimental and extremely CPU-hungry. `boxrev` is inspired by the idea of sound waves being multiply reflected on the inside of a box before they reach the listener. All the paths by which the sound waves can reach the listener within a certain cutoff time are computed and the signals with appropriate delay and attenuation are added up. `boxrev`'s first argument is a table containing the delays and attenuation factors corresponding to each dimension of the box, followed by `END`. There may be fewer or more than 3 delay/attenuation pairs if you so choose — corresponding to reflection around a rectangle or a hyper-box. As an example, try the following values:

```
sndobj *thefile;
float boxdims[]= { 0.165, 0.6, 0.114, 0.7, 0.0724, 0.8, END };
```

```
thefile= file("song.wav", NULL);
output= boxrev(boxdims, 2.0, 0.1, thefile);
```

This reverberation sounds unnaturally bright because it does not contain any lowpass. You might want to add a lowpass filter to change that.

The other experimental reverberator is called `convrev`. It generates a filter kernel using several noise generators and processes the sound signal by convoluting it with this kernel. The kernel is composed of three parts: particularly clear direct reflections (echos), indirect reflections (also echos of a sort, but dense enough not to sound echo-ish) and noisy ambient reflections. The duration of those three parts are the first three parameters of `convrev`, their relative volume is determined by the other two. As for the other reverberators, here is an example as a starting-point for your experimentation:

```
sndobj *thefile;

thefile= file("song.wav", NULL);
output= convrev(1.5, 0.7, 0.3, 6, 3, c(0.95), thefile);
```

While we are talking about reverberation, the `convolve` object should not go unmentioned. It allows you to use a reverberation (or other) kernel you have downloaded from somewhere (for instance here) and saved in wave file format. Simply use a `file` object as the <kernel> input of `convolve`. The only downside is the extremely long computation time.

4.6 Distortion and effects

Also already implemented in SNDSYS are several kinds of effects. Distortion of electric guitars is most easily achieved by clipping. The sndobj `clip` clips a signal to be less than another (in absolute terms). In order to use it to distort a signal, choose the clipping input as the squared mean of the signal, provided by the sndobj `fastrms`.[6] By varying the (quench) parameter of `clip`, you can adjust how strongly the signal is clipped when it exceeds the limit.

```
sndobj *thefile;

thefile= file("song.wav", NULL);
output= clip(0.5, fastrms(c(10), thefile), thefile);
```

Other sndobj's which can be used for distortion are `softclip`, `crossoverdist` and `arithmetic` (see Section 6.2; try taking the signal to the power 1/4 for distortion).

Among the effects which sound less nasty than distortion are `flanger`, `phaser` and `leslie`. `flanger` and `phaser` are similar in that both suppress certain discrete frequencies, but differ in the relative location of those

[6] Root of mean square.

frequencies. Their effects are especially audible if one varies the base frequencies from which the others are derived. For `flanger` this is done by varying the <delay> input (which is the inverse of the base frequency), for `phaser` vary the <basefreq> input. `leslie` is another classical effect which is a bit hard to describe — try it out.

5 Give every tone its cue

In the previous sections we have learnt to generate and shape sounds. But that is only one step on the way to creating melodies and music. To that end, we will also need to define the starting time and development with time of a note. This is what we will learn in this chapter.

5.1 Envelope shaping

In this section I will present sndobj's which define the enveloping curve — envelope for short — of a tone. They all have a cue input (which I sometimes also call a "trigger"). This cue input is expected to be zero except when a new note starts. The maximum of the envelopes will be proportional to (usually equal to) the non-zero cue value.

The most well-known envelope shaper is the ADSR (attack, decay, sustain, release) envelope, represented in SNDSYS by the `adsr` object. It is a piecewise linear function which first rises, then falls off to its "sustain" level and finally falls off to zero. The lengths of these four phases and the proportion of the sustain level are the (constant) parameters of `adsr`.

SNDSYS provides several other envelope shapers which have fewer parameters. `expodecay` outputs a falling exponential; `idexp` provides a suitably normalised function proportional to $t \cdot \exp(-t)$ which first grows linearly, then decays exponentially; and `halfgauss` outputs half a Gauss function, starting at its maximum. All three of them have only one argument besides the cue: an input which determines how fast the function decays to zero.

Further, it should be mentioned that piecewise linear envelopes (similar to ADSR) can be generated by creating an ASCII file with the file name extension .dat. It should contain two columns of numbers separated by spaces or tab characters. The first value in every row will be interpreted as a time in seconds, the second as the value of the envelope. The values will be linearly interpolated between the times given. Such .dat files can be read by the `file` object, but that only allows to output the envelope once. The `oneshot` object plays its file every time it receives a non-zero trigger and can therefore be used to generate envelopes from .dat files.

Applying an envelope to a sound is done by multiplying them. The sndobj which does this is called `mul` (no surprise) and has exactly the same usage as `add`. Its first argument is the number of sndobj's to multiply, followed by the objects.

5.2 Cues and ramps

The two most important sndobj's for generating cues are `at` and `cron`.[7] Both have an offset as their first argument, which allows to shift the cues they generate forwards or backwards in time. Both also have an array argument (though in the case of `cron`, it can be replaced by `NULL`). These arrays differ in an important way from the arrays we have encountered previously: These arrays contain double precision floating-point values. The actual initialisation values can be given as for single-precision arrays, but the array has to be declared differently. Consider the following example of an array for `at`:

```
double atlist[]= { 0, 1, 0.5, 1, 0.75, 1, 1, 1, END };
```

The list contains pairs of values, concluded by `END`. The first value of each pair is the time of the cue in seconds; the second is the cue value. `at` outputs zero values except at the samples which are closest to the specified times, when the given cue values are output. (Depending on the sampling rate, the cue times may not hit one sample exactly, but each cue is guaranteed to be output unless there are more of them than sample values.)

Though melodies are still beyond us, we can use `at` to generate a simple rhythm:

```
double atlist[]= { 0, 1, 0.5, 1, 0.875, 1, 1, 1, END };

output= mul(2, sine(0, c(440), c(1)), expodecay(c(0.2), at(0, atlist)));
```

`cron` generates regular cues. Its parameters (interval) and (value) give the interval between cues and their value, respectively. The array argument, if not `NULL`, contains the times when a different value or no cue should be output. It has the same format as the array passed to `at`: pairs of times and values, concluded by `END`.

Another sndobj which can be used to generate regular cues is `interleave`. It interleaves its <signal> input with a certain number of zero samples. If you want to control precisely at which sample values non-zero cues occur, you can feed the cue value into the <signal> input and give the exact number of zero samples by which they will be separated.

So now we can generate cues — fine. To create a melody, we need more than that. We also want to vary the tones' frequencies. We cannot use `at` or `cron` for that, because the waveform generators need a constant frequency input for as long as the tone continues. The simplest way to achieve that is with the `ramp` object. It has the same two arguments as `at`, but the format of its [list] argument is different. It contains triplets of values concluded by `END` (actually, `END` replaces the last value of the last triplet). Of each triplet, the first value is a time in seconds, the second is the value which the output of `ramp` should have at that time, and the third determines the kind of interpolation between this time and the time of the next triplet. The last value of the last triplet is

[7] If you don't get the pun, just ignore the peculiar names.

END because there is no next time to interpolate to. The interpolation may be linear (symbolic value RAMP_L), exponential (RAMP_E), or the output value can remain constant until the next time (RAMP_C). With this knowledge, we can generate a micro-melody in A minor:

```
double freqlist[]= { 0, 440, RAMP_C, 0.5, 528, RAMP_C, 1, 469.333, RAMP_C,
                     1.5, 440, END };

output= mul(2, sine(0, ramp(0, freqlist), c(1)),
               expodecay(c(0.2), cron(0, 0.5, 1, NULL)));
```

Now, if you want to write a song, will you have to write tens of `double` arrays several thousand entries in length which you have to take care to keep consistent? Actually, no. Section 7.7 will show how to read scores from files in certain formats. But the objects in this section are simpler, easier to understand, and can still be useful for all kinds of tasks even if you use a score file, and were therefore presented first.

5.3 Auxiliary triggered objects

In the last section we used the `ramp` object to provide a frequency to the `sine` object which changed with each tone but remained constant during the tone. In this section we will get to know objects which we could have used together with `at` to achieve the same purpose. They are especially necessary when the frequency (or another characteristic of a tone) is determined by a varying signal. Consider the following "melody":

```
output= mul(2, sine(0, linear(440, 110, sine(0.5, c(0.3125), c(1))), c(1)),
                expodecay(c(0.2), cron(0, 0.4, 1, NULL)));
```

What we were trying to do was to change the frequency of the tone with the second `sine` object — with the effect that we got some kind of siren punctuated by cues. We need something to keep the frequency values fixed until the next note starts.

This is the purpose of the `flipflop` object. It has two inputs — the cue and the signal the values of which should be stored. Using this object, our melody sounds much more like a melody (though still somewhat dissonant):

```
output= mul(2, sine(0, linear(440, 110, flipflop(cron(0, 0.4, 1, NULL),
                                  sine(0.5, c(0.3125), c(1)))), c(1)),
                expodecay(c(0.2), cron(0, 0.4, 1, NULL)));
```

Two objects which serve a similar purpose to `flipflop` are `holdtrigger` and `holdnon0`. `holdnon0` could have been used in the previous section to generate a piecewise constant frequency signal from the output of `at`. `holdtrigger` is useful when a cue value has to be present (= non-zero) for a fixed duration of more than one sample value.

5.4 A humaniser

Computer-generated sounds can sound artificial, a problem which is often mitigated by "humanising" some of the quantities which a computer otherwise treats much more exactly than a human would. There is no "humaniser" in `sndsys` — and yet there is. This is to say that there is no sndobj dedicated to this task, but it is easily accomplished using sndobj's we have already encountered. Let us assume we want to "humanise" the precise frequency of a sine, and that the frequency signal is denoted by `frequency`. To change the frequency slightly in a random way, we can replace `frequency` by:

```
flipflop(cue, mul(2, frequency, linear(1, 0.01, gaussrand(0.3))))
```

`cue` is assumed to be the cue signal for the tone to which the frequency applies. Similarly, the volume or other properties of tones can be humanised. It is important to use the `flipflop` object here. If the frequency were continually changed by a random contribution, this would be audible as noise. As we have been programming it, it will be slightly off but remain constant over the tone.

Humanising the cue itself requires aslightly different approach. It is accomplished by delaying the cue by a small random amount. The `delay` object has a parameter fixing the maximum value the delay input can take, and the delay and signal inputs. Its delay can also be negative. To humanise a precise `cue` signal, replace it by the following:

```
delay(0.1, gaussrand(0.01), cue)
```

6 Miscellaneous sndobj's

This section deals with objects which did not fit neatly into any of the previous sections' categories or are too advanced to be presented in the introductory sections. The first subsection is about delay objects, which are useless on their own but a requirement for complex processing. Multi-channel processing will be introduced, which allows to create stereo sounds and are important for using certain more advanced sndobj's. Mixing follows naturally. SNDSYS's way of accessing your soundcard is presented, and for those with an up-to-date box, multithreading capabilities are explained.

6.1 Delays

SNDSYS offers several objects which simply delay a signal by a certain time. Two of them apply a constant delay: `cdelay` has a time parameter which determines the delay, while `sdelay` has a parameter giving the number of samples by which to delay the signal. In both cases, the delays may also be negative. The delay objects simply read ahead by the requested time and discard the first samples until they read far enough "into the future".

The other two delay objects allow a variable delay given by an input. Because the delayed data have to be stored somewhere, the maximally possible delay

value has to be given in a parameter. If the delay input exceeds this value, the maximum value is used instead. `delay` has just three arguments — maximum delay, delay input and signal input — while `rdelay` has an additional reset input. Whenever it receives a non-zero value on the reset input, it acts as though the delay line has been cleared and outputs zeros for the time of the current delay.

6.2 Arithmetic and comparisons

In earlier sections, we have already learnt how to do the arithmetic operations most frequently needed in sound processing: `linear` for scaling and linear transformation, `add` for summation, and `mul` for multiplication.

The less common arithmetic operations can be performed with the sndobj `arithmetic`. Though such operations should in the interest of efficiency be implemented in C if used frequently or in large number, `arithmetic` allows to play around with them without having to write a sndobj of one's own. Its first argument is a string containing one character defining the arithmetic operation to perform. "+", "−", "*" and "/" are self-explanatory. "^" takes the first operand to the power of the second without changing its sign. The other three are unary operators: "|" takes the modulus, "=" is the negation, and "\" yields the inverse. Note that in C code, "\" has to be given as "\\".

`arithmetic` has two inputs which provide the two operands. When computing unary operations, the second operand is ignored and may be passed as `NULL`. In the case of the inversion "\", a second operand may be given, and is used as a maximum output value. This is important when using this arithmetic operation to compute a delay from a frequency. The length of the delay can then be kept within limits.

`arithmetic` tries to react benevolently to nonsensical input, which often cannot be entirely avoided. Divisions by zero yield the maximal floating-point value with the correct sign, not infinity. Division of zero by zero repeats the previous value. Powers with base zero result in the maximum if the exponent is negative, zero otherwise.

A sndobj vaguely related to arithmetic is `conditional`, which selects one of two inputs based on the comparison of two others. It has two inputs, of which some will usually be the same. If the first input is greater than or equal to the second, the third input is passed through to the output, otherwise the fourth. This allows you to more complex decisions than just limiting the amplitude as for example with `clip`.

6.3 Stereo and multi-channel processing

In the starting chapters, we have been generating sounds in mono, without even caring about the number of channels. This may be acceptable for creating sounds which can later be panned left or right, but for music mono is rather dead. The good news is that you do not have much to learn. Most sndobj's handle stereo signals (or signals with even more channels) transparently, delivering as

many channels of output as you put in. In fact, if you applied the filters from Section 4 to a wave file containing a song in stereo, you may have noticed that the output was in stereo too, automagically.

Multiple channels are harder to handle for sndobj's which combine inputs in some way. Some sndobj's, among them `add`, `mul` and `arithmetic`, output the maximum number of channels of their inputs. The inputs having fewer channels wrap around. This allows to multiply two stereo signals channel by channel, but also allows to multiply a stereo sound with a common (mono) envelope. To find out how a specific sndobj handles multiple channels, consult its documentation (via objectdoc.pl) or, as a last resort, the source code.

How to create or take apart a multi-channel signal? The sndobj for extracting a single channel is called `ch`. Its first argument is a constant denoting the channel to extract. Channel numbers start at zero. Abbreviations are available for the first four channels: `c0`, `c1`, `c2` and `c3`. More complex rearrangement of channels and generation of multi-channel signals can be done with `switchboard`. Its first argument is a string giving the switching to be done and is followed by a variable number of inputs. The string argument specifies which channel of which input to output on each channel, for the first to the last output channel, separated by spaces. The specification of one output channel consists of the input number (starting at zero for the first input), optionally followed by a dot and the channel number (also starting at zero). If the channel number is not given, the first channel is used.

Two rather special sndobj's which reduce the number of channels are `chsum` and `chavg`. They compute the sum and the (arithmetic) average of all channels of their input, respectively. Other sndobj's manipulating channels are the mixing objects presented in the following section.

6.4 Mixing and panning

The simplest mixing object has already been used in Section 4.5: `mix2`. Its first input, <gate>, determines how each of the two other inputs is scaled. If <gate> is 1 or above, only the first input is output, if it is ≤ 0, only the second. (The idea is that <gate> is the coefficient of the first signal input, which it precedes.) To crossfade from one signal to another, you can use a `ramp` as the <gate> input:

```
sndobj *signal1, *signal2;
double faderamp[]={ 0, 1, RAMP_C, 11, 1, RAMP_L, 14, 0, END };

signal1= ...
signal2= ...
output= mix2(ramp(0, faderamp), signal1, signal2);
```

This code fragment cross-fades from `signal1` to `signal2` between 11 and 14 seconds into the song. The output of the `ramp` remains constant after the last time given in `faderamp`, so only the second signal is played after 14 seconds.

Besides crossfading, a frequently used functionality is the creation of a stereo signal from a mono sound. This is done with the sndobj `panpot`. The first argument, a constant parameter, determines how the perceived direction of the signal is created: by a difference in volume between the left and right channel (0) or by a phase shift (1). For values between 0 and 1, both methods are combined. The first input determines where the signal is panned. -1 pans it to the left, 0 to the centre, 1 to the right and intermediate values in between.

A very general and powerful mixer is available in the `mix` object. It resembles `switchboard` in that it can generate an arbitrary number of channels by combining an arbitrary number of multi-channel inputs.[8] Its usage is also quite similar: Its first argument is a string which determines how the inputs are mixed down to the output and is followed by the input signals. The mixing string can have two different formats. The first gives a mixing specification independently for each output channel, separated by semicolons. The second format prescribes the same mixing for all output channels, wrapping input channels around if necessary (the number of output channels is the maximum of the number of the inputs' channels). The mixing specifications consist of a sum of channel numbers (first format) or input numbers (second format) optionally followed by a scaling factor. Always remember that the scaling factor comes second — don't confuse it with the channel number which also looks like a decimal fraction.

If you wanted to use `mix` for panning voices of a song, you could write the following:

```
output= mix("0.0*0.707 + 1.0*0.5 + 2.0*0.707 + 3.0*0.866 ;"
            "0.0*0.707 + 1.0*0.866 + 2.0*0.707 + 3.0*0.5",
            bass, drums, guitar, vocals);
```

Using `mix` for panning is not the most fitting of its uses, but it allows me to demonstrate a couple of things. First, you can see that the parameter string has been split over two lines. The C compiler allows that; it automatically concatenates two strings which are separated only by white space, including line breaks.

Now let's have a look at the mixing specification string. It contains two sums separated by the semicolon at the end of the part of the string which was printed on the top line. The first sum describes the first channel of the output, the second the second channel. It is the convention in wave files, to which the output will finally be written, that the first channel is the left channel, so the first sum defines the left-channel output. Each term in the sum is a product of a channel in the notation input.channel, starting with input 0. I am assuming here that all the inputs are single-channel signals, but nevertheless the ".0" cannot be left out (as it could with `switchboard`).

The scaling factors after the channel numbers do not add up to 1 for the left and right output channel. Why not? Because it is the sound intensity which is

[8] Actually, these numbers cannot be arbitrarily large. The number of inputs and the number of channels per input and of output channels are limited by the constant `PLENTY` which is defined in sndsys.h as 20.

divided up, not the amplitude. The intensity is the square of the amplitude, so the squares of the scaling factors of each input add to 1. There was no need to take care of that when using `panpot`, because it does it automatically. Now we have had a good look at the details, we can say what the above code fragment does: The bass and guitar are panned to the centre, while the drums are panned to the right and the vocals to the left.

When all is said and done, you would probably prefer to use `panpot` for panning each voice and add their stereo signals up with `add`. This also allows to change a voice's location dynamically, as the location is an input of `panpot`.

6.5 Speaker and microphone

So here comes the part you've been waiting for all along: How to make real noise, pardon, music, with SNDSYS. The sndobj's dealing with microphone input and speaker output currently use the Open Sound System API. If you use ALSA, you will also need ALSA's OSS emulation. If you have multiple sound cards, you may have to change the name of the DSP device in the function dspdesc(), which is at the bottom of sndsys.c. It is "`/dev/dsp`" by default; to use the second sound card, you may have to change it to "`/dev/dsp1`" or similar.

The sndobj playing sound over your system's speaker is called `oss`. It has only one input — the signal to play — and a parameter which allows to adjust the volume. If the signal has several channels, the first is played on the left channel of the soundcard, the second on the right, and the others (if present) are ignored. The microphone object is similarly spartan, having a parameter for selecting line input rather than microphone and a parameter for the recording volume.

Rather than code examples, this section comes with actual example programs which are part of the SNDSYS distribution. The first, and perhaps the nicest, is `piano`. It is compiled and started the following way (from a shell prompt in the SNDSYS directory):

```
$ make piano
$ ./piano
```

A small X window with a graphic of a keyboard will pop up and allow you to play piano on your computer keyboard when in focus. The keys are coloured white or black according to which piano keys they represent. Shift keys hold the tone after you release the corresponding key; CapsLock saves you holding Shift down. You quit with the Escape key.

Though nice, `piano` doesn't lend itself well to understanding soundcard access with SNDSYS — too much of the code is dedicated to X programming. `distort` is a much simpler program. It reads a signal from the microphone input of your soundcard, distorts it by clipping, and outputs it again over the speaker. Have a look at distort.c to see `mike` and `oss` in use, and feel free to play around with the clipping parameters or replace the distortion by a filter or a reverberation.

A sndobj which can come in useful occasionally is `voiceact` which is used in the `vrec` program. It discards selected parts of its input signal and can therefore be used to realise voice-activated recording, as in `rec`. Its first input determines whether the (second) signal input is output; if not, the input is *discarded*, that is the program may spend a long time in `voiceact` waiting for output to continue. Unlike other sound objects, `voiceact` does not always generate an output value for every input value. The way SNDSYS is designed, using such sndobj's requires some caution. See Section 9 or README.SNDSYS for more details on this. To see `voiceact` in action, have a look at vrec.c, and use objectdoc.pl to display its documentation.

6.6 Threads

Unlike other sound processing programs, SNDSYS does not apply processing steps to buffers of sound data, but passes each sample value down the processing chain separately. Samples are buffered where necessary for the operation of sndobj's, but the basic idea is that data stream through the successive stages of the calculation. This makes it especially easy to split computations between different processors, cores or even computers.

The only sndobj which implements this at present is `sndthread`, which starts a new thread for computing its input (and the input's inputs and so on). On multi-core and multi-processor systems, the operating system will usually run the new thread on a different core or processor from the main thread, distributing your calculations over two or several of them. `sndthread` just copies the data received from its input and handles the communication between threads. Besides its input, it has a parameter which gives the size of its buffer for the data from the other thread. If it is larger than 1, `sndthread` will read some values in advance, which helps to balance variations in processor usage. I usually set it to 1024.

What `sndthread` does *not* do is automatically determine where your SNDSYS program should best be split between processors. You have to decide that yourself. Start by dividing it into roughly equal parts in terms of the number of sndobj's and watch CPU usage with the programs `xosview` or `top`. If there is a significant difference, one part of your program takes up much more computation time than the other, and you should insert the `sndthread` object at a different point. Note that you cannot insert a `sndthread` just anywhere; it has to be the only connection between the sndobj's connected to its input and those to which its output is passed on. For instance consider the variable `thefile` which we declared in Section 4.5. If you replaced one of the two places in which it is used by `sndthread(1024, thefile)`, the other would still try to obtain data from a sndobj which would now be part of the other thread, creating chaos. The correct solution in such cases is to insert `sndthread` into the *assignment* of the variable, in this case: `thefile= sndthread(1024, file("song.wav", NULL));`.

6.7 A curiosity: Verhulst dynamics

To conclude this rather dull chapter, I will present a funny example of what you can abuse SNDSYS to do. The Verhulst process is a classic in the dynamics of iterative function systems. It consists of iteratively computing $c \cdot x \cdot (1 - x)$, the result becoming the new x. Depending on the value of the parameter c, x may converge to a constant, oscillate between the same 2, 4, or more values, or become entirely chaotic. The sndobj `bifurcation` outputs the x values of a cycle corresponding to c given by its input. By hooking it up to a sine generator and varying c, one can actually hear how the limit value becomes a cycle of successively larger length.

Put the following lines into our example framework program frame.c:

```
double bifurcramp[]= {0, 2.8, RAMP_L, 5, 3.3, RAMP_L, 10, 3.48, RAMP_L,
                      15, 3.53, RAMP_L, 20, 3.7, END };

output= mul(2, c0(smoothstep(SMST_LINEAR, 0.5, c(0.005),
                  expodecay(c(0.1), cron(0, 0.25, 1, NULL)))),
        sine( 0.0, linear( 440, 440,
           bifurcation(0.25, ramp(0.0, bifurcramp))), c(1)));
duration= 20.0;
```

After compiling and running the program, you can hear how a slowly changing single tone becomes two alternating tones, then four, and finally a chaotic succession of tones. This is how the limit of the Verhulst process changes as you change its parameter c along the `ramp` in the program.

Funny melody or no, let's still have a brief look at the code. The `double`-type array tells the `ramp` object how the Verhulst parameter should change with time. The first two line of the assignment to the `output` variable contain the first factor of the multiplication, which provides the envelope for the tones. The output of an `expodecay` object triggered every quarter of a second is fed through a `smoothstep` object, which helps to prevent clicks and is further explained in Section 7.2. The next two lines generate a sine wave with a frequency corresponding to an x value in the limit cycle of the Verhulst process. `bifurcation` outputs a number between 0 and 1, which is then scaled to give a frequency between 440 and 880 Hz. Its parameter input comes from a `ramp` which is designed to increase the parameter more slowly as bifurcations become more frequent.

7 Advanced sound synthesis

This section presents sound generation possibilities of SNDSYS which go beyond simple waveform generators and filters. Some skill in C programming, or the willingness to acquire it, will be required at least in Sections 7.1 and 7.7. The code examples will be more complex than previously, and sometimes incomplete and template-like, leaving you to fill in the blanks. Due to the great complexity

of the objects and methods presented in this section, the only way to realise their potential is to experiment with them at length.

7.1 Waveforms

We have already seen above how to use a wave file in SNDSYS. In addition, in Section 5.1, the `oneshot` object was briefly mentioned. These two are the basic sndobj's for generating an arbitrary waveform and will be explained in depth now, before we move on to something more advanced.

The `file` object can do much more than just play wave files. One of its features is that it repeats its file over and over again. While this may not seem overly sensible for a wave file containing a song, it opens up the possibility of storing a waveform in a wave file and using `file` as a waveform generator. `file`'s optional input, together with a constant resampling factor, can be used to set the frequency. Another constant parameter can be used to set the starting phase if desired (similarly as with the `sine` object). These additional constants work as follows: The file name argument of `file` is replaced by a string containing the characters '*' and/or '+' followed by the actual file name. The numerical constants then follow the frequency input in the argument list. Because the frequency input acts as a transposition factor with wave files, rather than an actual frequency, the resampling factor should be set to the inverse of the frequency of the waveform stored in the wave file. If it contains just one period, this is equivalent to its length in seconds. So if you have a very detailed waveform sampled with 88.2 kHz, which is 0.5 seconds long, you play it with the frequency `freq` by coding the following: (Differences in the sampling rate are automatically recognised and compensated by `file`.)

```
file("*waveform.wav", freq, 0.5)
```

But `file` can also read waveforms in different formats. Two plain text formats are available: .asc files contain successive sample values, one to each line. .dat files contain two values per text line, a time in seconds and a sample value. The times need not be equidistant, but have to be ascending, and the last value has to equal the first. In both cases, the sample values are floating-point values. As both .asc and .dat files are assumed to contain single waveforms, `file`'s input acts as a frequency and the constant resampling factor need not be used.

Finally, `file` can also play waveforms defined by polylines drawn with `xfig` [xfig]. This is a bit of a joke, but allows to create "zigzag" waveforms with lots of straight lines quickly. The polyline should be drawn from left to right, and the vertical coordinate of the last point should match that of the first (otherwise it will be replaced by that value). The vertical coordinate values will be scaled to fill the interval $[-1, 1]$.

`oneshot` and `rapidfire` are two objects which play a file only once at a time, in response to a cue input. They have two primary applications: drum sounds and envelopes. Drum sounds can be recorded or generated, stored in a wave file,

and played back with one of these sndobj's. The possibility to define an envelope in a .dat file was already mentioned in Section 5.1. (Note that the role of their input <speed> is different from that of `file`. Consult their documentation if you consider using that feature.) `oneshot` and `rapidfire` differ in how they react to triggers while a playback is still in progress. `oneshot` will abort and restart the playback, while `rapidfire` is able to perform several playbacks in parallel and add them up. One could say that `oneshot` simulates a single drum being beaten repeatedly, and `rapidfire` several drums being beaten in turn.

The sndobj's which have been presented so far are only the simplest case of waveform synthesis. The sound real-life instruments changes depending on pitch, volume, on how they are played, and to some extent on random chance. The sndobj `multiwave` is designed with this in mind. It assumes you have stored several waveforms of an instrument stored in .asc files and interpolates linearly between them depending on its inputs. This allows you to generate waveforms which change according to several different factors.

To handle this multitude, however, a little bit of theory is required. Ideally, you should have a waveform available for several pitches, several volumes, several values of other parameters which you may have in mind, and all combinations of them. You can obtain such waveforms from instrument sound samples, for instance those provided free by the University of Iowa [iow]. `multiwave` obtains the value of the parameters on which the waveform depends from its inputs. To obtain the right waveform, it interpolates linearly between the given waveforms, one parameter at a time.

How many and what kind of parameters the waveform depends on is what you tell `multiwave` in its many parameters. Use objectdoc.pl to display its documentation. Its first argument, (dimension), is the number of parameters, the others which are not inputs are arrays of different types. The first, [indextypes], contains two integer values for each waveform parameter, its type and the number of the input providing its value. The other two, [ascfiles] and [positions], describe the .asc waveform files. The former contains the names of those files, followed by an additional entry which is the empty string `""` or NULL, to tell `multiwave` where the list ends. The latter defines which sets of parameter values the waveforms correspond to. It contains all parameter values for all files, so its length is the number of parameters multiplied with the number of files.

`multiwave`'s constant arguments describing the waveforms and how to interpolate them are followed by its inputs. First is the frequency input, which is always required and which is used to decide how "fast" to play the waveform. If one of the entries in [indextypes] has type `MWTYPE_FREQ`, it will also be used for interpolation. After the frequency input follow the inputs corresponding to the other parameters, starting with the one identified as input index 0 in [indextypes].

As you will have noticed, the use of `multiwave` requires some more advanced C programming skills as well as some spacial imagination of multi-dimensional parameter spaces. I regret to say that I have not yet created an example, which would require a largish number of waveform files. I have used `multiwave` in

the little piece "Capricious Bass" which is available in OGG Vorbis format on my web site, and extracted multiple waveforms for plucked bass and saxophone from the University of Iowa's instrument samples for the purpose [iow]. But at the moment I do not yet want to make them public.

7.2 Fix the clicks

If you have played around a bit with the envelope shapers `expodecay` and `halfgauss` and with drum sounds using `oneshot`, you will have noticed that they can cause nasty clicks at some or every cue. In the case of the envelope shapers, this is due to the shaped wave signal being suddenly switched on when a cue occurs. If you have been shaping sine waves with zeros approximately coinciding with the cues, you will have heard nothing unusual, but one you start creating sounds with random elements in them, this will become a problem.

The object to address the clicks due to envelopes is `smoothstep`. It does what its name suggests. It smoothes steps in its input signal by replacing them with slopes or part of a sine wave, depending on its mode of operation. `smoothstep` compares successive sample values to determine whether a "step" occurred; to decide what constitutes a "step", you have to give it the minimum height of a step to be corrected. The right value to choose depends on the context in which it is being used. The usual procedure is to choose a largish value of 0.1 or so and decrease it if necessary until the clicks are gone. A second input besides the signal to smooth is the time to take for smoothing each step, which is usually chose as a constant.

`smoothstep` doubles the number of channels of a signal. Every other output channel gives the difference between the original signal and the smoothed signal. This can be used to implement "attack" sounds which occur when an instrument's tone starts. If you do not want these channels, you have to select the one (or ones) you want.

A typical usage of `smoothstep` is the following:

```
c0(smoothstep(SMST_LINEAR, 0.05, c(0.01), envelope))
```

It may have occurred to you that smoothing an envelope could have been achieved much easier with a simple lowpass. That is true, but `smoothstep` acts differently and has other uses. It only acts locally, not affecting the signal at all when there are no steps. This means it can also be used on signals which are not pure envelopes but already contain the tone. I have also used it to smooth transitions between pitches output by score objects (see Section 7.7), which an instrument couldn't handle. Advanced modes of operation allow it to smooth only upward or downward steps. See its documentation for details.

`smoothstep` could also be used to smother clicks at the start of a waveform played with `oneshot`. But there is a simpler option: `smoothstart`. Its only parameter is the number of samples it is allowed to modify before a tone starts. It constructs a third-order polynomial which matches both the silence before the tone and the first two samples played. As it reacts to the transition from

absolute silence to a sound, it is best used to modify a saved sample before it is used with `oneshot`. Otherwise the start of the sample may not be recognised when several cues follow closely on each other.

7.3 Fractional Brownian motion — a noise generator

The random number generators presented in Section 3.3 are nice, but they go only so far. You can generate white noise with them, which you can then filter to give noise with a certain frequency distribution. While useful in many cases, this does not generate noise which sounds "natural".

Natural noise often obeys scaling laws associated with fractals. This allows it to be generated by the following recursive procedure. It starts with complete silence, which appears as a straight horizontal line when you plot the sound signal. Imagine an elastic string fastened to two pins at the left and right end of the time interval you view. Now you take the string at regular intervals, move it up or down from the horizontal line, and fix it with further pins. Then you shorten the interval by a factor and do the same thing again. And again. And so on. The amount by which you move the string at every step is random but also diminishes, so at the end you get a signal which moves up and down randomly, but is much more correlated than white noise. This way of constructing a fractal, and some mathematical implications, are well described in a book about fractal images [sfi].

The sndobj `fbm` generates noise using the method explained above. It has three inputs which determine its output (and a reset input which may be passed as NULL and is rarely used). The first, <lacunarity> is the factor by which the interval used at successive steps of the iteration is shrunk. The second, simply called <H>, is the exponent of the power law relating the size of the interval with the size of the displacement. It determines the fractal dimension of the resulting curve, which is $2 - 2\,H$.

For non-mathematicians: <lacunarity> determines how "full" the noise sounds. Small lacunarity values cause the result to sound "thin", while large values (up to the maximum of 0.9) give a "full" sound. <H> determines the frequencies present in the noise. Small values cause all frequencies to be equally prevalent and generate nearly (but not quite) white noise. Large values, up to but smaller than 1, cause low frequencies to predominate. `fbm`'s third input is a cutoff frequency which sets the lowest frequency the noise contains.

The documentation of `fbm` contains several example parameter values with descriptions. Here, I will give a code example only for the last and most complicated, which involves varying H and sounds somewhat like a large heap of gravel sliding off a metal surface. H will be varied with a sine, and a ramp will be used to fade the sound in and out and adjust the volume.

```
double slideramp[]= { 0, 0, RAMP_L, 0.1, 1, RAMP_L, 1.2, 1, RAMP_L,
                      1.4, 0.7, RAMP_L, 1.5, 0, END };

output= arithmetic( "*", fbm(c(0.8), linear( 0.9, -0.9,
```

```
        sine( 0.25, c(0.14), c(1) )), c(150), NULL), ramp(0.0, slideramp) );
duration= 1.5;
```

`fbm`'s immediate purpose is to generate realistic noise, but its uses are many. It can be used wherever a random number generator is required and will often yield more realistic sounds than if `flatrand` had been used. One funny effect occurs when `fbm` is used for phase modulation:

```
output= delay( 1.0, linear(0.0, 0.00005, fbm( c(0.5), c(0.8), c(600), NULL)),
               sine(0.0, c(600), c(0.8)));
```

This generates a tone which is clearly noisy, but which appears quite smooth in a waveform editor. Due to the highly correlated nature of the noise, the changes in phase cause a barely perceivable ripple in the shape of the sine. The human ear, however, is not deceived.

7.4 More noise and randomised cues

The sndobj which will be presented in this section resembles `fbm` in that it is based on the idea of self-similarity which also underlie fractals. It was inspired by investigations of a block which is only just sliding on a vibrating sloped surface. Such a block will slide down in jerks, with the strength of the jerks correlated with their frequency through a power law.

The `slide` object generates one-sample peaks at regular intervals, with heights which depend on the interval. Like `fbm`, `slide` repeats this for successively smaller intervals (and peak heights). Its first two inputs, <minfreq> and <maxfreq>, give the inverse of the largest and smallest interval size, and the <lacunarity> input determines the factor between the interval sizes applied at successive steps. The <exponent> input determines how the height of the peaks depends on the interval size. For an exponent of 0, the pulse height will remain constant. A positive exponent will cause the height to decrease for a smaller interval, making smaller peaks more frequent. The last input, <meandev>, allows a randomisation of both the intervals and the pulse heights. It should be smaller than 1, and larger values cause a stronger randomisation while 0 causes none at all.

Unlike `fbm`, `slide` is not primarily intended to be listened to directly. However, its output can be used to simulate crackling fire and similar noises. Here is a simple example:

```
output= linear(0, 0.7, slide( c(10), c(500), c(0.7), c(1.5), c(0.3) ));
```

`slide` can also be used to provide random cues. Many natural processes — rain drumming on a roof, large numbers of stones or beads hitting a plate — are composed of single events which occur with varying intensity. `slide` can be used as a trigger for sounds of one raindrop or one bead hitting to reproduce the process as a whole. Another application is to provide more-or-less regular but randomised reflections for a reverberator. It is used internally by `convrev` for that purpose.

7.5 Waveform manipulations

This section covers three sndobj's which allow you to manipulate the waveform of any signal. They are more likely to be useful as an effect; for synthesis, you could probably generate the right waveform in the first place, though perhaps not in all cases. I have to admit I have not worked very much with these objects myself, so they may have applications which I do not know of.

`reshape` replaces the shape of each "half-wave" of a signal with that of a different signal. A "half-wave" is half a waveform. The precise definition of this term depends on `reshape`'s mode of operation. If it is `RESHAPE_SLOPE`, each half-wave extends from one (local) minimum or maximum of the signal to the next maximum or minimum. For the other modes, it extends from one zero-crossing to the next. (If your signal does not touch the zero line, you should shift it with `linear` before feeding it into `reshape`.) `reshape` scales the half-waves of its <shapesig> input so that their size matches the half-waves of the signal, and substitutes them.

`reshape` is an "asynchronous" sndobj, because its <shapesig> input may have to provide more or fewer samples than it outputs in time interval. For this reason the <shapesig> and <signal> inputs should not have anything to do with each other. If there is a signal on which both of them depend, you should do the exact opposite of what I recommended in Section 4.5: The signal should be reproduced for both inputs, rather than assigning it to a variable used with them both. Then they will be sufficiently independent for `reshape` to work.

The following example uses `reshape` in its operation mode `RESHAPE_SLOPE` to replace a song's waveforms with sine half-waves. This creates a fine distortion which neither leaves parts of the signal unaffected nor becomes too nasty. (How exactly it sounds may depend on the wave file you apply it to.)

```
output= reshape(1.0, RESHAPE_SLOPE, sine(0.0, c(440), c(1)),
                     file("song.wav", NULL) );
```

The second sndobj affecting waveforms, `repeathw`, works on just one signal. Rather than modifying the waveforms, it repeats every half-wave of the signal several times. Unlike `reshape`, `repeathw` always defines a half-wave as the part of the signal between two zero-crossings. If your signal has fewer zero-crossings than you would like, applying `dcblock` or a different highpass may help.

Obviously, the output sound of `repeathw` will take several times as long to play as the input. Besides, `repeathw` causes strange effects which differ depending on its parameter inputs. After its constant parameter, which sets a maximum length for the half-waves, two parameter inputs follow before the signal input. The first determines the number of repetitions for the half-waves. (It is always rounded to an integer.) The second allows repeating sets of several half-waves rather than each on its own.

The effect of `repeathw` is so strong that it must count more as a curiosity than as a serious tool for generating smooth sounds. As described in its documentation (use objectdoc.pl), it may generate bubbling sounds or simply slow down its input signal (though unfortunately not without artifacts). Its

applications range from generating outright weird sounds to encryption of sound data - though admittedly there is currently no corresponding decryption sndobj. Have a look at the example parameter values given in the documentation and apply `repeathw` to a wave file of a song!

The third object modifying a signal's half-waves is called `avghw` and averages the shapes of a number of successive half-waves. Unlike the two others, it is not asynchronous, meaning that it reads a sample from its input for every sample it outputs. This sndobj is a bit of a disappointment — it simply generates a distortion. I had hoped for something more interesting when I programmed it, but all the same I wanted to mention it here.

7.6 A string model — backfeed loops

So far we have learnt that objects can be arranged in lines (when all of them have only one input) or trees (when one or several of them have more than one input). This section introduces a new topology: loops. Because creating each sndobj requires references to its inputs, loops are created in two stages. When building up a processing chain of objects, two objects `defloop` and `loop` are inserted. Later a function `closeloops()` has to be called which replaces the references to each `loop` object with the input of the corresponding `defloop` object. If you use the function `sndexecute()` to execute your SNDSYS program, you need not perform the second step, as `sndexecute()` calls `closeloops()`.

Both `defloop` and `loop` have arguments "name" and (id) which denote a specific loop. (The integer (id) makes it easier to generate a series of loops; it would be harder to increment a string of characters.) As its third argument, `defloop` has the signal which is to be used instead of the corresponding `loop` object. The `loop` object takes the number of channels as its third argument. This is necessary when the sndobj of which it is an input needs to know the number of channels of its input.

A simple example for a backfeed loop is provided by programming the Karplus-Strong string model using SNDSYS. This model is used in the example program model.c to produce a single tone. The code looks like this:

```
double trigger0[]= { 0.0, 1, END };

output= defloop("karplus-strong", 1, cdelay( 1.0/440,
              add( 2, at( 0.0, trigger0 ), _12_0pole( 1, 0.5, 0.5, 0.0,
                loop("karplus-strong", 1, 1 )))));
```

The `defloop` and `loop` objects feed the output of the `delay` back into the `_12_0pole` filter. This filter is a low-level filter object which has the filter coefficients as its argument and is here used as a one-zero filter which averages two successive samples. The remaining object is the `add` which adds up the backfeed signal and the excitation, the plucking of the string provided by the `at` object.[9]

[9]Since writing that example, I have created two special-purpose sndobj's which facilitate

7.7 Scores and instruments

Finally we are coming to the features of SNDSYS which allow you to compose and play tunes. As in many other parts of SNDSYS, the emphasis was on power and versatility rather than suitability for casual use. Unlike for instance Sapphire [sap], SNDSYS does not distinguish between scores and other processing elements — all output one or more channels of data. Instruments again are not artificially distinguished from other sndobj's — they just happen to understand the output of score objects. This brings about much freedom and also responsibility — to some extent it is up to you how scores tell instruments what to do, and it is up to you to ensure they understand each other.

7.7.1 abc

SNDSYS currently has two different objects to interpret score files. Let us first have a look at the score format which gives you less latitude. The sndobj's name is `abc`, which parses the music notation language abc [abc]. Describing all of abc is beyond the scope of this tutorial — but then, all of abc is not actually implemented in the `abc` object. You are encouraged to read about abc on its home page [abc], become enthusiastic, read the documentation of the `abc` object, and become disappointed. Nevertheless, here is a brief overview:

abc files contain one or several tunes, each of which is composed of a header and the actual music notation. The header consists of lines starting with a single letter followed by a colon. It gives basic information about the song — its title ("T:song"), metre (for example "M:4/4"), tempo ("Q:1/4=120"), key ("K:C") and unit note length ("L:1/4"). SNDSYS's `abc` object interprets only six header fields, the five just mentioned and a field ("u:") defining accents. For the purpose of playing tunes, the most important are Q:, K: and L:. Unlike some other programs interpreting abc files, SNDSYS assumes an abc file to contain only one tune. To select one of several melodies from the same file, you can only use the non-standard V: (voice) field, which has a number as its value which is a parameter of the `abc` object.

The header is followed by the notated music itself. Notes are represented by the letters "a" to "g", with the octave determined by capitalisation and appended puctuation (a comma for lowering, an apostrophe or several for raising by an or several octaves). The note length is proportional to the unit note length given in the L: header field. Notes of different lengths can have a factor appended to them which can be an integer or a fraction of integers (for instance "3/4"). In the latter case, the numerator can be omitted if it is one. Simply appending a slash "/" divides the note length by two, which can be repeated several times. Rests are denoted by "z" with the same syntax. More advanced features in a nutshell: chords are notated as several notes between square brackets; bars are delimited by vertical bars "|"; ties are notated with hyphens or by enclosing

programming the Karplus-Strong model. `karplusfilt` implements the zero filter and allows an additional decay and a negation for generating drum sounds. `stringfeedback` replaces the `add` object and has the additional feature of being able to initialise the delay line.

notes in parentheses; dynamics marks and special accents are given as text enclosed in exclamation marks preceding the (first) note they apply to; staccato is notated by a "." preceding the note.

Now you have a basic grasp of the syntax of abc files, let us come to the really important things. How does the `abc` object output the data fromteh abc file, and how can you write instruments which play a tune using its output? `abc` has at least three output channels. The first channel delivers a cue at the start of each note. The second channel contains the amplitude of the note. The remaining channels contain the frequency or frequencies of the note or of all notes in the chord.

How does `abc` know which dynamics signs correspond to which amplitudes? Easy, you tell it. The same applies to the cue values which depend on which accent(s), if any, are set on the note. This information is contained in the `abcplayopts` struct a pointer to which is passed to `abc`. The struct element `mfvolume` is the amplitude which is output for mezzoforte notes. Forte and piano notes differ by the factor `fortefactor` or 1/`fortefactor`, respectively. Fortissimo differs by two `fortefactor`s and so on. The amplitude channel of `abc` also contains the information of when to end a note: it will fall to zero at that point. Which proportion of the nominal note length the tone will be held is determined by the struct members `len`, `stacclen` and `tenutolen`, depending on whether and which accent is present. The value of the cues which `abc` delivers are given by `cue` (normal notes), `stacccue` (dotted, "staccato" notes), `legatocue` (the second or later of a group of tied notes) and `legstacccue` (dotted tied notes). If the element `endcue` is non-zero, a cue is also sent at the end of each tone.

Further members of the `abcplayopts` struct are: `maxchord` gives the maximum number of notes in a chord and thereby determines the number of output channels. If `jazzynotes` is non-zero, it causes the length ratio of broken rhythms to be 2:1 rather than 3:1. `silentfreq` determines whether the value of the frequency channel(s) remains the same after the end of a note (1) or is set to 1 (0). The element `transposefactor` gives a factor the frequencies are multiplied with — instant transposition. Finally, two arrays are part of the `abcplayopts` structure. Their entries correspond to each other. `accentstring[]` contains accent names including the exclamation marks by which they are limited, and `accentcue[]` the corresponding cues to be output for such notes. If the accent is negative, the corresponding positive value is added to the usual cue instead of replacing it.

The simplest kind of instrument playing the output of `abc` would just ignore the cue channel. It would simply generate a tone with the requested amplitude and frequency and work somewhat like this:

```
sndobj *score, *freq, *vol;

score= abc(.....);
vol= c0(smoothstep(SMST_LINEAR, 0.05, c(0.003), c1(score)));
freq= c2(score);
```

```
output= sine(0, freq, vol);
```

The next most complicated possibility is suitable for simulating a plucked string instrument. Here we will all but ignore the volume and only multiply it with the cue, which we feed into an envelope generator:

```
sndobj *score, *freq, *vol;

score= abc(.....);
vol= c0(smoothstep(SMST_LINEAR, 0.05, c(0.003),
             expodecay(c(0.2), mul(2, c0(score), c1(score))) ));
freq= c2(score);
output= sine(0, freq, vol);
```

A more complicated type of instrument would combine both approaches. The tone itself could be generated as in the first example, while an "attack" noise could be prompted by the cue and added to the tone. The attack noise would then be influenced by a note's accents (via the cue value) while the actual tone remained the same. Both would be influenced by the volume derived from dynamics signs, possibly by modifying the sound using a `multiwave`.

An example for an abc file is contained in the SNDSYS distribution in melody2.abc, which is read by model2.c. The instrument used in this source file is the Karplus-Strong string model, which is controlled similarly to our second example above.

7.7.2 xyz

The second score object which is currently implemented is called `xyz`, which is a pun referring to abc. Its basic idea is reading a comma-separated list of numerical values from an external file and delivering those values on its output channels. But it can do more than that: Cue values can be generated, frequencies can be notated similarly to abc pitches, prefactors can be applied to output channels, and bar lengths are enforced as in abc. `xyz` was created to allow more flexible control of instruments without having to encode too much information in obscure accents, as would have to be done in abc.

`xyz` has only two arguments, the .xyz file name and the name of the voice. It can do without the many settings which are delivered to the `abc` object in the `abcplayopts` structure, because such information is directly embedded in the .xyz score file. This file contains one or several voices which consist of their name followed by the voice's notes enclosed in curly braces.

The notes for each voice are organised in rows of comma-separated values. These are not necessarily equivalent to text lines — xyz is not a line-oriented format. Each row is concluded by a semicolon, a vertical bar, or a colon. A vertical bar marks the end of a bar, and the end of every other note or rest is denoted by a semicolon. The colon ends the first row which defines the type of each column of numbers. "Normal" numerical columns do not require a type keyword in the top row; their place in the first row may be empty. Other column

types are the following: “length” denotes the column giving the note length in seconds. This column has to be present.[10] The type keyword “net” denotes the column (if present) which contains the proportion of the note length for which the tone is to be played. It corresponds to the `len` element of the `abcplayopts` structure for `abc`. “cue” columns deliver cues; the value given in that column is output only once, followed by all zeros. Finally, “abc” columns allow to give a frequency by an abc note pitch. The syntax is slightly modified: The “,” lowering the pitch by an octave is replaced by a “.”, and the apostrohpe “’” is replaced by a backtick “‘”.

All column type designations except “length” and “net” can be optionally followed by a factor, the word “mute” and a further number. The factor is multiplied with all numbers in that column. The mute value is output on the corresponding output channel when a note has ended (as defined by the “net” column) or during rests. The “length” keyword can be followed by a unit length, by which all note lengths will be multiplied, and the number of unit notes forming a bar. If the bar length is given, the length of each bar will be checked and a warning will be printed if the length deviates.

After so much theory, let us now look at a practical example. The following excerpt of an xyz file shows the first rows of one voice:

```
voice1
{
  length 0.5 4, net, abc 1.189, 1 mute 0:
  2, 0.8, a, 1;
  0.5, 1, c, 0.5;
  0.5, , d, ;
  0.5, , e;           // no typo!
  0.5, , d, |
  4, 0.8, c‘, |
  // ....
}
```

First, a look at the top row. The note length is given in the first column, with a unit note half a second long and four unit notes to a bar. It is up to you which notated note length you associate with the unit note. In this case one would probably interpret it as a quarter, and the metre as common time. The following column gives the actual length of a tone compared to the nominal note length. The third column is the pitch in abc notation, and the last column is an ordinary numerical column giving the amplitude. The pitch is transposed by a minor third upwards, which gives a factor of $\exp(-3 \cdot \log(2)/12) \approx 0.841$. The last column has prefactor 1 and shall be output as 0 when the tone is muted. This gives us an output channel which represents the correct amplitude, even for gaps between notes and for rests.

[10] If it is not there, `xyz` will arbitrarily interpret the first column as the note length column, which is usually not a good idea.

Let us now try to read the melody. According to the first row the first note is two unit notes long. Assuming the unit note is a quarter, this is a half note. The following four notes are then eighth notes and conclude the first bar. The last note of our excerpt is a whole note filling the second bar. A look at the third column reveals the melody: The first note is one octave below the standard pitch, therefore has a frequency of 220 Hz, and is called A3 in piano notation. The following eighths live near the bottom of the same octave, and the final whole note is a middle C.

Now have a look at the second column. For the first and last note, the "net" length is smaller than one, meaning that they are separated from the respective following note. The first eighth note has a net length of 1, while for the othe three it is not specified. This causes `xyz` to repeat the previous value and give them, too, a net length of 1. This value causes no pause between the eighth notes and therefore causes them to be played legato. The fourth column gives the notes' volume. If we assume the value 1 for the first note to be mezzoforte, the other notes are piano or so. Again, the value is left out where it remains constant. In the fourth note, even the comma is left out and the row is ended prematurely. This is allowed and causes all omitted columns to keep their previous values.

Finally, a speciality of xyz files is that `xyz` applies the C preprocessor to them.[11] This allows you to use C comments in them. The example above shows two single-line comments starting with "//", but the longer form delimited with "/*" and "*/" is also allowed. The preprocessor also allows you to define constants which you can use in your scores. For instance, you could have written the above excerpt like this:

```
#define mf 1.0
#define p  0.5

voice1
{
  length 0.5 4, net, abc 1.189, 1 mute 0:
  2, 0.8, a, mf;
  0.5, 1, c, p;
  0.5, , d, p;
  0.5, , e, p;
  0.5, , d, p|
  4, 0.8, c', p|
}
```

If you use this feature in the future, beware of defining a constant `f` for forte! The preprocessor is a dumb program and will also replace f notes in the abc columns with the numerical value. Better call forte `ft` or similar. If you are

[11] That means that the preprocessor has to be installed and available with the command `cpp` on a system where you use a compiled SNDSYS program which uses `xyz`, not only where yo compile it.

getting strange error messages from `xyz` and suspect the preprocessor is messing things up, look at the file /tmp/*name*.xyz.cpp. This is where the `xyz` object has the C preprocessor dump its output and what itself subsequently parses. The `#define` directive also lets you abbreviate riffs or parts of a melody. `#include` allows to include other xyz files. See the documentation of your C preprocessor for details.

How do you play an xyz file? Here is a short example program. (As before, it is a fragment to be inserted into frame.c.) It assumes that you have written the short xyz file from above to frame.xyz.

```
sndobj *score, *freq, *vol;

score= xyz("frame.xyz", "voice1");
freq= c0(score);
vol= c0(smoothstep(SMST_LINEAR, 0.05, c(0.003),
                   linear(0, 0.3, mul(2, invA(freq), c1(score)))));
output= sine(0, freq, vol);
duration= 4.0;
```

The length and net length columns are not output by `xyz`, so the first output channel is the frequency. The second channel contains the amplitude. It is scaled and smoothed before being fed into the sine generator. The only sndobj we have not seen before is `invA`. It outputs the inverse of the A-weighting for the frequency which is its argument and is used to adjust the amplitude so that tones of different pitches will sound equally loud.[12]

8 Wavelets

This section is about using wavelet transforms to generate and modify sounds, and the objects SNDSYS provides for this purpose. Wavelet transforms are an advanced topic of which we will only touch the surface. But even the bits we will need, and their implementation in SNDSYS, are not easy. If the explanations bore you, don't hesitate to skip to the examples and play around with them. Sometimes it is best to learn from one's own experience.

8.1 Wavelet transforms

What are wavelet transforms? They[13] are transforms which allow analysing a signal both in terms of the frequencies it contains and in terms of how it changes with time. That this is possible is not trivial and entails several peculiarities, some of which we will encounter shortly. You may have heard of the Fourier

[12] Though the A-weighting strictly speaking applies to the intensity (squared amplitude) rather than the amplitude, `invA` is written to be used as shown here — to be multiplied with an amplitude. `invA` provides the square root of the inverse intensity A-weighting.

[13] I am using the plural because, as we will see, there are several different types of wavelets, which yield different types of transforms.

transform, which transforms a signal into its frequency spectrum. Wavelet transform go farther in that they can say which frequencies a signal contains at a given time, at least to some extent.

If a wavelet transform truly could give a full frequency spectrum at every time, the amount of data would be huge. (A single Fourier spectrum alone has as much data as the signal itself.) Somehow it is intuitively clear that this amount of data would be highly redundant, or that most data sets would fit no corresponding signal. In fact, a wavelet transform generates exactly as many data as the signal provides. Unlike the Fourier transform, it does not decompose the signal into sine waves with multiples of a certain base frequency. It merely divides the signal into bands between frequencies which are factors of two apart, the highest of which is half the sampling rate. The lowest boundary frequency is a constant parameter of all wavelet-related objects available in SNDSYS.

Beside the "rough" frequency resolution in octaves, a further peculiarity of wavelet transforms is the frequency-dependent time resolution. Every second value of a wavelet-transformed signal belongs to the highest frequency band, every fourth to the next lower and so on. This can be understood if one realises that higher frequency oscillations have shorter periods — so their time resolution is correspondingly more accurate.

But what are wavelets *really*? As their name suggests, they are small waves. As we will see below, they are oscillations which have a certain amplitude at a point in time but which fall off rapidly to zero both before and afterwards. The different frequency bands analysed by a wavelet transform correspond to wavelets scaled by different powers of two. Since different (though not just any) wavelet shapes can serve to generate a transform, there are several different wavelet transforms. SNDSYS provides the most widespread ones and offers the possibility to perform custom ones as well.

The simplest kind of wavelet is the Haar wavelet. Even though it is not really suitable for generating and manipulating sounds (unless you favour industrial-grade distortion), SNDSYS provides two implementations of it: `wthaarnn` is the very fast un-normalised version, and `wthaar` the conventionally normalised variety. `iwthaarnn` and `iwthaar` are the corresponding inverse transforms which restore a signal from its wavelet decomposition. I programmed these transforms because they are easy to do, and they may have applications in encryption and data compression.

The other wavelet transforms come in series. Besides the bottom boundary frequency and the signal input, they have as an additional parameter the order of the transform. This is an even number determining the number of samples the highest frequency wavelet is long. The wavelet transform objects `wtdaub`, `wtdaubla` and `wtdaubbl` are variants of the Daubechies wavelet transform. (The inverse transforms have the same name with an `i` in front.) `wtdaub` uses the classical Daubechies wavelets. The other two are modified to enhance certain properties of the wavelets which cannot all be achieved at once.

`wtdaubbl` gives the best relative localisation for coefficients in different wavelet bands — for other wavelets, the non-zero transformed values making up a short bump may be far apart. The downside of this optimisation is that

the frequency bands are not as clearly separated as they could be, but rather washed out. Another trade-off is that these wavelets are more likely to give rise to artifacts when the wavelet-transformed data are manipulated. The "least asymmetric" Daubechies wavelets implemented in `wtdaubla` are a compromise between all three properties. They are the ones I use most frequently. The orders of all variants of Daubechies wavelets implemented in SNDSYS are between 2 and 20 (only even numbers).

Finally, the last series of wavelets provided by SNDSYS is the so-called Coiflets. Implemented by the sndobj `wtcoif`, their order has to be divisible by six and ranges from 6 to 30. Though the small-order coiflet transforms are artifact-prone, I have found the higher-order ones as useful as the least asymmetric Daubechies wavelets for sound applications.

For performing a wavelet transform based on any quadrature mirror filter, the sndobj `wtany` and its inverse `iwtany` are provided.[14] All the above-mentioned transforms (except `wthaarnn`) are in fact special cases of `wtany`. It has the *scaling function* or *smoothing filter* as its argument rather than the corresponding wavelet filter; the scaling function is given by a float array concluded by the magic value `END`. An additional parameter (llap) is used to shift the filters in time and thereby adjust the phase behaviour, or temporal localisation, of the transform. `wtany` was programmed to make the transforms as free of artifacts as possible. The filters which separate the frequency bands were programmed as true moving filters without folding some coefficients back into a fixed-size interval, as is sometimes done. This has the disadvantage that for high-order wavelets the transformed data depend on sample values from a long time interval, but you can't have everything.

So what does a wavelet actually look like? We can use SNDSYS to generate some single wavelets well separated from each other:

```
output= iwtdaubla(20, 20, linear(0, 0.5,
                  slide( c(20), c(20), c(0.5), c(1), c(0.5))));
duration= 5.0;
```

View the result in a waveform editor. You will see several short waves appearing and decaying, the wavelets. (That the lower-frequency wavelets have a smaller amplitude is a consequence of their different normalisation and of our simple way of generating them.) We used the `slide` object to generate a series of single-sample cues approximately 0.05 seconds apart but strongly randomised ($\pm$50%). The random intervals cause the cues to hit wavelet data representing different wavelet bands, with high-frequency bands more probable due to their data being more frequent. The single wavelets sound like clicks and "bop" sounds of varying frequency.

For more information about wavelets, look into wikipedia or your favourite search engine. For the mathematically interested reader, I can recommend the book [wm] which includes proofs and exercises. [nr] also has a section about

[14] If you don't know what a quadrature mirror filter is and don't want to go and find out, you won't be creating your own wavelet transform and can skip this rather technical paragraph.

wavelets, which is available here. If you are interested how SNDSYS represents wavelet-transformed data, read the description at the beginning of sndwt.c and the documentation of `wtany`.

8.2 Manipulating wavelet bands

Judging from my experience so far, wavelets are more useful for changing and shaping sounds than generating them "from nothing". To that end, SNDSYS offers a number of objects allowing to manipulate wavelet data.

`wshift` shifts the wavelet data upward or downward in frequency. Because every higher wavelet band has a factor of two more data, this involves interpolating the data. `wshift`'s (mode) argument determines how this is done. The mode `WSHIFT_SELECT` simply picks out or repeats data and creates the worst artifacts (after reverse transforming the signal). `WSHIFT_SPREAD` performs some interpolation but still causes artifacts. These artifacts are in fact similar to those resulting from `repeathw` (see Section 7.5), probably because wavelets and half-waves behave similarly. In my experience the artifacts of `wshift` are less strong, at least for shifting downwards in frequency; though that may depend on the sound signal. The third mode of operation, `WSHIFT_SUCC`, interpolates between successive data of the some wavelet band. Far from generating a smooth output, this results in a strong metallic distortion. `wshift`'s third parameter is the number of wavelet bands (octaves) to shift the signal upward (>0) or downward (<0).

Rather than give an example for `wshift`, I will proceed right away to the next object affecting wavelet-transformed data, `wspread`. What this sndobj does amounts to executing `wspread` for all upward and downward shifts and adding the results to the original signal, with smaller and smaller prefactors the larger the shift. It has a parameter giving the `wshift` mode and two inputs determining the prefactors of the signal shifted up or down by one band. The prefactors of the multiply shifted signals are the squares or higher powers of these inputs.

`wspread` generates a full, choir-like sound for modes `WSHIFT_SELECT` and `WSHIFT_SPREAD`. Interestingly, the distortion of `WSHIFT_SELECT` is hardly audible (it seems to cancel out between the different amounts of shift), so this is the mode to prefer of those two. The mode `WSHIFT_SUCC` again creates a metallic distortion, a fuller sound but a worse distortion than `wshift` for a single shift alone. Let's now apply this to a wave file so you can hear the effect for yourself.

```
output= iwtdaubla(20, 20., wspread(20., WSHIFT_SELECT, c(0.25), c(0.4),
                  wtdaubla(20, 20., file("song.wav", NULL))));
```

The sound signal is first wavelet-transformed, then put through `wspread`, and transformed back. To try out the metallic distortion which is a frequent feature of manually manipulated wavelets, substitute the operation mode `WSHIFT_SUCC` or replace `wspread(20., WSHIFT_SELECT, c(0.25), c(0.4),...` by `wshift(20., WSHIFT_SUCC, 1, ...`. The metallic distortion

makes wavelets great for generating sounds reminiscent of metal objects being hit. For instance, try the following:

```
double trigger0[]= { 0.0, 1, END };

output= iwtdaubla(8, 20., wshift(20., WSHIFT_SUCC, -2, wtdaubla(8, 20.,
                  mul(2, rect(0., c(14000.), c(1.), c(0)),
                      expodecay(c(0.2), at(0, trigger0))))));
```

As usual, this processing chain has to be read from right to left. A rectangular wave is generated and made to decay exponentially by multiplication with an `expodecay` object. Its frequency is four times as high as the output we want, because `wshift` will shift it down by two octaves. This decaying wave is wavelet-transformed, `wshift` is applied, and the result is transformed back. Note that we use low-order wavelets here because we want a certain amount of distortion. You are encouraged to experiment with this code example: Choosing higher-order wavelets will diminish the metallic sound. You can replace the `rect` with a `saw` which will sound different and less metallic. Try varying the number and direction of the octave shifts, adapting the original frequency in the opposite direction for a better comparison. This will create completely different sounds, though a certain metallic quality will always be there.

The next wavelet sndobj we will have a look at is `wselect`. It allows to suppress specific wavelet bands. The bands to be kept are either chosen explicitly (operation mode `WSELECT_INDEX`) or by the size of the corresponding coefficients (the other modes). Unlike for other sndobj's, `wselect`'s mode is an input and can therefore be changed on the fly. The second input is a bit mask in which a bit is set for each wavelet band to keep. Bit 0, which has the value 1, represents the highest-frequency band for mode `WSELECT_INDEX` or the largest in magnitude for the other modes. The bit values 2^{n_i} have to be added up to give the mask selecting the n_i^{th} wavelet bands.[15]

`wselect` can be used to generate fine cymbal sounds. To that end, we select a single wavelet band from decaying white noise:

```
double trigger0[]= { 0.0, 1, END };

output= iwtcoif(30, 20., wselect(20., c(WSELECT_INDEX), c(0x2),
                wtcoif(30, 20., mul(2, gaussrand(.3),
                       expodecay(c(0.3), at(0, trigger0))))));
```

Here, the highest-order "coiflets" were used, and `gaussrand` was preferred over `flatrand`, because in my personal opinion this combination sounds best. Compare that to the bland sound of just `mul(2, gaussrand(.3), expodecay(c(0.3), at(0, trigger0)))`! Of course, all we have done is created a narrow band-pass filter which we apply to the white noise. But

[15] Never mind that sound data are really floating-point. The <mask> input is rounded to the nearest integer and treated as such.

wavelets give us ready access to such band-passes. In fact, the above code could be simplified by leaving out the `wtcoif` object, because the wavelet transform of uncorrelated noise is also uncorrelated noise. Only the argument of `gaussrand` should be made larger to match the normalisation of the wavelet band in question.

The constant object `c(0x2)` selects the second-highest wavelet band, with the number 2 given in hexadecimal. Other bands also give nice noises. For instance, selecting two of the lower bands with the hexadecimal constant 0x300 generates the rumbling of far-away thunder. To avoid having to adapt the volume of the Gaussian noise again, you can use a feature of `wselect`: use "`WSELECT_INDEX|WSELECT_SAMERMS`" as the mode instead of just `WSELECT_INDEX`, and `wselect` will try to compensate differences between the wavelet bands' normalisation itself.

The other modes of `wselect`, which keep wavelet bands depending on how large their respective data are, give strange or outright silly effects. These modes differ only in details. You can best test them by applying `wselect` to a wavelet-transformed digitised song. Choose a mask of 0xF or 0x1F to keep about half the bands. You will hear abrupt changes in the sound as specific bands are switched on and off.

Other sndobj's which work on wavelet-transformed data are `wtrunc`, `wthresh` and `wlinear`. `wtrunc` suppresses all wavelet bands above or below a given frequency. For most wavelets, this is a lowpass or highpass, but for low-order wavelets special effects can result. `wthresh` suppresses wavelet bands which do not exceed a certain magnitude. `wlinear` is in some sense a generalisation of `wspread`. It allows you to do a general linear transformation between wavelet bands. For instance, you could create a variant of `wspread` which only spreads wavelet bands to their immediate neighbours, or to their second and fourth neighbours and so on. You can also mirror wavelet bands, swapping high and low frequencies. Just because it sounds so outrageous, here is an example you can apply to a digitised song:

```
float wlinmirror[110]= {
  0,0,0,0,0,0,0,0,0,1,0,
  0,0,0,0,0,0,0,0,1,0,0,
  0,0,0,0,0,0,0,1,0,0,0,
  0,0,0,0,0,0,1,0,0,0,0,
  0,0,0,0,0,1,0,0,0,0,0,
  0,0,0,0,1,0,0,0,0,0,0,
  0,0,0,1,0,0,0,0,0,0,0,
  0,0,1,0,0,0,0,0,0,0,0,
  0,1,0,0,0,0,0,0,0,0,0,
  1,0,0,0,0,0,0,0,0,0,0,
};

output= iwtdaubla(20, 20., wlinear(20., WSHIFT_SELECT, 10,
           wlinmirror, wtdaubla(20, 20., file("song.wav", NULL))));
```

8.3 Controlled manipulation

The observant reader will have noticed that I have kept the parameter inputs of the wavelet objects in the previous section constant. The curious may have tried to vary them and met unexpected results. The reason for this is that the wavelet-transformed signals are delayed relative to those which are not transformed (to which the signals changing parameters usually belong). This delay can be compensated with the objects `wdelay` and `iwdelay` which should always be used with non-constant parameter inputs of `wspread`, `wselect`, `wtrunc` and `wthresh`.

`wdelay` creates the same delay as a wavelet-transformed signal relative to the untransformed signal, while `iwdelay` has the opposite (negative) delay. Both have as parameters the lower boundary frequency, the order of the wavelet transform and its "overlap" parameter. The latter is 0 for Haar and ordinary Daubechies wavelets and should be passed as `WDELAY_CENTRE` for the optimised Daubechies wavelets and Coiflets. Note that the first two arguments of `wdelay` and `iwdelay` are reversed compared to the wavelet transform objects. (That is so for "historical" reasons, or in other words for no good reason.)

As an example, consider you might want to slowly fade in the choir effect created by `wspread`. This can be done by using a `ramp` to adjust its parameter inputs and delaying them with `wdelay`.

```
sndobj *tmp;
double fadein[]= { 0, 0, RAMP_L, 5, 1, END };

tmp= wdelay(20., 20, WDELAY_CENTRE, ramp(0, fadein));
output= iwtdaubla(20, 20., wspread(20., WSHIFT_SELECT,
                  linear(0, 0.25, tmp), linear(0, 0.4, tmp),
                  wtdaubla(20, 20., file("song.wav", NULL))));
```

9 More information

This tutorial, even the advanced chapters, is by no means exhaustive. There are a number of sndobj's which have not been mentioned. Some are highly experimental (SNDSYS is a test bench first and foremost, after all), others didn't work out the way I hoped and were abandoned. If you are missing something, look through sndsys.h and use objectdoc.pl to view the documentation of anything that might be it (or might help to create it). If you find something, test it in a simple short program. If you don't find anything, or if it does not work as you expect, you are stumped unless you know C.

This brings us to the second thing which is missing from this tutorial. If you know the C programming language, you can create sndobj's of your own. The file README.SNDSYS contains terse information on how to do this, as well as a description of how SNDSYS really works internally. It is required reading for anybody aiming to extend, or merely to understand, SNDSYS. The document of last resort is of course the source code itself, some of which is even documented ;).

It contains a lot of sndobj's which can serve as examples, as well as some nasty hand-optimised hacks.

References

[abc] abc music notation
`http://abcnotation.org.uk/`

[iow] University of Iowa Electronic Music Studios
`http://theremin.music.uiowa.edu/`

[jos] Julius O. Smith's online books
`http://ccrma.stanford.edu/~jos/`

[kwa] Kwave, a sound editor for KDE
`http://kwave.sourceforge.net/`

[nr] W. H. Press, S. A. Teukolsky, W. T. Vetterling and B. P. Flannery: Numerical Recipes in C
Cambridge University Press 1988, ISBN 0-512-43108-5
`http://www.nrbook.com/b/bookcpdf.php`

[sap] Sapphire — an acoustic compiler
`http://www.pale.org/sapphire/index.html`

[sfi] H.-O. Peitgen and D. Saupe (Eds.): The Science of Fractal Images
Springer 1988, ISBN 0-387-96608-0 or 3-540-96608-0

[snd] Snd, a sound editor
`http://ccrma.stanford.edu/software/snd/`

[sox] SOX — Sound eXchange
`http://sox.sourceforge.net/`

[swe] Sweep, an audio editor
`http://www.metadecks.org/software/sweep/index.html`

[wm] D. B. Percival and A. T. Walden: Wavelet Methods for Time Series Analysis
Cambridge University Press 2000, ISBN 0-521-64068-7

[xfig] Xfig, a drawing program for X windows
`www.xfig.org/`