The Incomplete Csound Manual

Risto Holopainen, October 2023

Preface

This manual is written for the impatient sound adventurer who wants to get to know just sufficiently of the Csound language to get started rapidly. The manual is intentionally incomplete, and will remain so. Why? Because Csound has expanded so much in a few decades that no sane person would learn all of it. I have collected what I consider a bare minimum of useful things to know, which is only a fraction of what can be done but still an immensely powerful set of tools. This manual tries to be concise. I will not discuss sound design or the fundamentals of signal processing. If you have any experience with programming or modular synthesizers, you will notice that Csound is located somewhere in the intersection of those two worlds.

The examples have been tested on Csound version 6.16.

Introduction

Csound is a domain specific language for audio. Its heritage goes back to the MUSIC N family of languages developed since the 1960's but it is constantly expanding with the inclusion of more functions and syntactical constructions.

Today Csound has about 1200 different opcodes. These opcodes are similar to the keywords of a programming language, they are the basic vocabulary you need for programming. Or perhaps it would be better to compare opcodes with all the functions available in the standard libraries of a programming language, or the various modules available for a modular synthesizer.

Useful as these opcodes are, it is easy to get lost in the intricacies of all possibilities on offer. Therefore, this guide attempts to present a bare minimum of the Csound language, just enough for a wide range of tasks, if not for everything one might conceivably want to do. My selection of a reduced set of Csound's opcodes reflects my personal interests, other users would surely choose different ones. The point is to find a sufficiently small set of opcodes which can be committed to memory, because nothing hampers productivity like permanently having to consult the manual. This guide introduces a subset of Csound that should be sufficient to get started.

I will focus on two use cases: sound synthesis and processing of an input soundfile. I will assume you have Csound installed. Otherwise, download[1] and install it first so you can try out the examples. You will need to use a simple text editor to write plain text files containing instructions for Csound.

1 https://csound.com/download.html

The interaction with Csound happens on the command line, although there are probably a few graphical front ends if you insist.

Basic structure

Csound is a text based audio programming language. It is functionally separated into two parts: sound synthesis by instruments where sound producing and processing algorithms are defined, and a score which specifies events in time. The first part, called the orchestra file, consists of one or more instrument definitions. Each instrument may or may not receive input signals and should have an audio output signal. The other part is called a score file. It contains a list of note events with an instrument number, a starting time, a duration, and possibly any number of parameters or so-called p-fields that are interpreted by the instrument.

The orchestra file

The orchestra file is a text file with the .orc suffix. It has a header, followed by instrument definitions. The header is where you set the sample rate, the control rate, and the number of output channels. For example, it may look like this:

```
sr = 48000
kr = 12000 ; control rate
nchnls = 2 ; stereo out
```

Anything following a semicolon on a line is a comment. In signal processing literature, there is a ubiquitus expectation that signals be limited to the amplitude range [-1,1]. Various soundfile formats have different maximum amplitudes, but to ensure that we can define signals in the standard range of [-1,1] regardless of output format we always include the following statement next in the orchestra file:

```
0dbfs = 1.0
```

This means that the peak at 0 dB full scale is at the numerical value 1.0. Louder signals will clip.

An instrument definition begins with **instr** followed by a number, and ends with **endin**. Inbetween is the specification of what the instrument does. The syntax of instrument definitions is quite simple; basically it consists of lines with three columns where the first indicates a variable name, the second is an opcode, and the third is a set of parameters used by the opcode. Like this:

```
variable-name(s)   opcode    parameter-list
```

The parameter list is a comma separated list of values pertaining to the opcode, which must be specified in the right order, sometimes with optional parameters at the end of the list. Usually the variable to the left is a single variable name but, depending on the opcode, it may be a comma separated list.

The opcodes may be thought of as similar to the modules of a modular synthesizer: there are oscillators, envelope generators, filters, and much more. The variable names on the left are signals of various types. Think of them as the patch cords of a modular synthesizer; they are connected to the output of the opcode and can be routed to various inputs further down the instrument definition.

And the inputs in this case are the parameters of each opcode. Trying to use a variable that has not yet been declared as a parameter of an opcode produces an error.

A simple oscillator

Let's build an oscillator whose amplitude and frequency can be specified from the score. First we declare the variable iamp, which takes its value from p4, the fourth p-field in the score file, which we'll use to set the amplitude. Similarly, the variable ifrq is assigned a value from p5, the fifth p-field of the score. The variable aos holds the audio signal that we will store in a soundfile.

```
sr = 48000
kr = 16000
nchnls = 1

0dbfs = 1.0

instr1 ; Oscillator routed to output
  iamp     =        p4
  ifrq     =        p5
  aos      oscili   iamp, ifrq, 1
           out      aos
endin
```

Save the above code in a file named sineoscil.orc for later use. Here we have introduced two op-codes: **oscili** and **out**. The oscillator takes three arguments and sends its output signal to the variable aos. The first argument to oscili is the amplitude, which we have defined as an i-rate variable iamp. All i-rate variables remain constant during a note event. The second oscili parameter specifies the frequency in Hz, and the third is the number of a wavetable, which must be defined in the score file. The **out** opcode has no variable name on the left. It routes the signal on its right hand side to the output, which is typically a soundfile. For stereo output the opcode is **outs**, which takes two input argument, one for each channel. Thus, for example,

```
       outs   ax, ay
```

outputs ax to the left channel and ay to the right. The number of channels specified in the header, nchnls, must match the output opcode.

There are three types of signals or variables that we need to know about. These are audio rate, control rate, and init rate signals. The audio rate is the sample rate of the output soundfile. The control rate is slower and must divide the sample rate. Its purpose is simply to save computations, and it is used for parameters that may change slower than the audio samples, such as envelopes or vibrato. For those more used to analogue modular synthesizers, control rate signals are similar to cv (control voltage) signals. The i-rate corresponds to note events as defined in the score section. At the start of a note the instrument begins to generate sound, and it keeps on for the duration specified by that note event.

Variable names follow a simple rule: audio rate variables must have a name beginning with **a**, control rate signals must begin with a **k**, and init rate variables begin with **i** (or **p** followed by a number referring to p-fields in the score file). There are also global variables beginning with **g** which will be discussed later, and many other types that we will not discuss. Unfortunately, there exist some opcodes with names beginning with a, i or k, and these names are reserved.

Actually, the naming convention is a clever idea. To declare a variable, its type is automatically inferred from its name, and there is no way of mistaking the type of a variable further down the instrument definition. But trying to name a variable to some reserved keyword will result in possibly cryptic error messages.

The score file

By convention, the suffix .sco is reserved for score files. At a minimum, a score file should have a note event of the form

```
i# p2 p3 [p4 p5 ...]
```

where # is the number of an instrument in the corresponding orchestra file, p2 is the time when the note begins, and p3 is its duration in seconds. In the oscillator example above, p-field 4 is used for amplitude, and p5 for frequency. The third oscillator parameter refers to a wavetable number 1. This wavetable must be defined in the score file. There are an enormous amount of options, of course, but we will stick with one single format:

```
f1 0 4096 10 1
```

This should be the first line in the score file. Here **f** indicates that this is a wavetable, the number immediately after **f** is the number of the wavetable, which the oscillator refers to; the following zero is the time from which the wavetable is made accessible (we want to use it from the beginning); 4096 is the wavetable's size (a power of two should be used), the number 10 specifies one of the fifty or so different versions of the wavetable generating function, in this case one that creates an harmonic tone by summing sinewaves. The last number, 1, specifies the amplitued of the first partial. If more numbers are added to the list, their position after ”10” indicates partial number, and the value is the partial's amplitude. Compound waveforms can be made by summing more harmonics, for instance

```
f2 0 4096 10 1 0 0.5 0 0.25 0 0.125
```

would result in a waveform with the first seven odd partials with amplitudes 1, 0.5, 0.25, and 0.125, respectively, which we have stored in wavetable number 2. By default, the amplitude is normalised to 1. Refering to this wavetable from an oscillator might look something like:

```
asig   oscili   iamp, ifrequency, 2
```

There are often several different ways of acheiving the same result in Csound. The compound spectrum stored in f2 might also be mixed together by playing four instances of one oscillator producing a pure sinewave together, although that would be less efficient. For inharmonic tones, however, there is not option but to use several differently tuned sine oscillators.

Now, let's test the oscillator instrument with a score file. Save the following lines in a file called sineoscil.sco.

```
f1 0 4096 10 1
i1   0    2    0.5    440  ; instr 1, start, dur, amp, freq
i1   2    1    0.9    311  ; at time 2, play one second at 311 Hz
e
```

Note that score files end with the letter **e**. The notes in the score file don't have to follow in consecutive temporal order, Csound will take care of sorting them. Any number of notes can play simultaneously to create a mix, but care has to be taken to avoid excessive amplitudes and clipping.

The csound command

In order to produce an output soundfile, an instrument definition needs to be put in an orchestra file and there needs to be a score file with note events that activate the instrument. Then a command line argument is given in roughly the following form:

```
csound -[options] outfile orchestrafile.orc scorefile.sco
```

If you put the file names in the wrong order you will get an error message. Suppose we have saved the orchestra file as sineoscil.orc and the score as sineoscil.sco, then the command

```
csound -Wgo sinetest.wav sineoscil.orc sineoscil.sco
```

should generate an output file named sinetest.wav with a single sinusoid at 440 Hz, at amplitude 0.5 for two seconds, then a sinusoid at 311 Hz at amplitude 0.9 for one more second. The options in this command are W for output to the .wav format, g to suppress graphics output, and o for the output file.

Sometimes it would be practical to put the orchestra and score in the same file. This is known as the *unified file format*. There is a special syntax to specify what part of the single file makes up the orchestra and score parts. The main advantage of the split format, apart from having fewer syntactical structures to remember, is that score files may be generated algorithmically by other programs without having to think about outputting the orchestra file as well. In this guide we will only use the split format.

All available opcodes can be listed by typing:

```
csound -z
```

A help message comes up with the flag -h.

Practical advice

Csound is extremely flexible in how things can be done. Efficiency considerations are the reason why there is such a thing as a control rate. Otherwise everything could run at the sample rate, and if you really care about sound quality that might be preferable. Likewise, efficiency is the motivation for using wavetables instead of, say, calling a sine function. But the sine function is available and can be called when necessary.

The interface between the instrument and the score p-fields is also very flexible. There can be any number of p-fields (there probably is an upper limit, but it is unlikely that you would ever approach it), and the p-fields may represent almost whatever one can think of. To avoid confusion, it is good practice to follow these two simple principles whenever possible:

Try to be consistent in the use of p-fields.

Keep the number of p-fields small, make design decisions in the instrument.

As for consistency, say you have several instruments with amplitude and frequency given by p-fields in the score. It would be very confusing to use p4 for amplitude and p5 for frequency in some instruments, and to do the opposite in some other instruments. Sticking with a habit regarding the first few p-fields will lessen the potential confusion as you work with many different instruments.

Also, just because you *can* have twenty p-fields and specify all sorts of obscure parameters of a sound doesn't mean that you should. There is a significant cognitive load of having to recall what all those parameters do. Besides, there will be a lot of typing to do, because you have to fill in all those p-fields for each note.

Case studies

Type in these examples and run them yourself. Try to understand what is going on, and experiment with modifications.

Additive synthesis

Should we use a wavetable with a complex waveform for additive synthesis? This is efficient, but restricted to harmonic spectra. Another option is to use an oscillator that produces a single sinusoid and sum a few of them by writing several notes in the score file. With the latter alternative, the spectrum may be inharmonic and the partials may each have their own amplitude envelope. Or we could use several oscillators in the same instrument, each tuned to a different frequency.

In this new version of a sine oscillator, we introduce one new opcode: **linen**. This is a simple envelope generator with four parameters: peak amplitude, rise or attack time, total duration (which is always p-field 3), and final decay. The time unit defaults to seconds. By multiplying the oscillator's output aos with the control rate envelope kenv in the last line we avoid clicks at the beginning and end of notes.

```
instr 2 ; single sine oscillator

  iamp   =         p4
  ifrq   =         p5
  iatk   =         0.05          ; atack time of envelope
  idec   =         0.15          ; decay time
  aos    oscili    1, ifrq, 1    ; f1 should hold a sinewave
  kenv   linen     iamp, iatk, p3, idec

         out    kenv * aos
endin
```

In the score file, we will specify three inharmonic notes to sound simultaneously for five seconds. Their amplitude is set to 0.3 to ensure the sum will not clip.

```
f1 0 4096 10 1 ; sinewave

i2   0    5    0.3    396.0
i2   0    5    0.3    560.0
i2   0    5    0.3    762.1
e
```

At this point it might be interesting to experiment with different spectral mixtures. Or try changing the wavetable to the previously defined f2. As mentioned, the same spectrum with three partials could be created by one note and an instrument using three oscillators.

```
instr 3

  iamp      =       p4/3
  ifrq1     =       p5
  ifrq2     =       1.4141*ifrq1
  ifrq3     =       1.9245*ifrq1

  a1      oscili    1, ifrq1, 1
  a2      oscili    1, ifrq2, 1
  a3      oscili    1, ifrq3, 1

  amix    =         a1 + a2 + a3
  kenv    linen     iamp, 0.05, p3, 0.15
          out       kenv * amix

endin

; score file:
f1 0 4096 10 1
i1   0    5    0.9  396
e
```

With instrument 3 we have a transposable mix of three oscillators, so complex tones with the same intervals between their three partials may be transposed to any frequency. Or we may superimpose several of these notes in the score, and even change the wavetable to something containing a few higher partials.

Notice the line beginning with amix. The equals sign assigns the expression on the right hand side to the variable amix. All standard arithmetic operations can be used, although division by zero should be avoided. Variables of different rate can be mixed freely, but the result will be converted to the fastest rate. This is what happens in the multiplication kenv*amix, where the output is a-rate.

Waveshaping and ring modulation

One of the classic synthesis techniques uses nonlinear transfer functions to produce harmonic partials assuming the input is a sinusoid. Waveshaping is simply the application of a polynomial, often with the sinusoid as its argument:

$$y(t)=f(x(t))$$
$$x(t)=\sin(\omega t)$$

where the polynomial is

$$f(x)=a_0+a_1 x+a_2 x^2+a_3 x^3+\ldots$$

The polynomial's order is the highest term with nonzero exponent. When the input is a single sinusoid at frequency *F* Hz, the highest partial produced by a transfer function of order *n* is *nF*. Even powers will cause a DC offset, which can be removed by subtracting a constant or by highpass filtering.

```
instr 4; waveshaper

  ax   oscili    1, p5, 1           ; put a sine in table 1
  aw   =         ax^2               ; same as ax*ax
  aw   atone     aw, 100            ; highpass filter, cutoff 100 Hz
  ke   linen     p4, 0.1, p3, 0.2   ; envelope
       out       ke*aw

endin

; score file
f1 0 4096 10 1
i4   0    2    0.7    220
e
```

The opcode **atone** is a one-pole highpass filter which can be used to remove the DC offset caused by squaring the sine signal. Notice that it is one of the few opcodes beginning with an *a*. Never try to name a variable atone!

When creating higher order polynomials, it is a good idea to introduce new variables and factor higher powers into lower ones. In the next instrument we also introduce three related modulation signals, k1, k2, and k3, which are used to animate the sound by taking turns emphasising different spectral components.

```
instr 5

  ax     oscili   1, p5, 1
  ax2    =        ax^2
  ax3    =        ax*ax2
  ax5    =        ax*ax2*ax2

  imod   =        1/3
  k1     oscili   1, imod, 1
  k2     =        .5*(1 + k1)   ; unipolar modulation signal
  k3     =        1 – k2        ; and its inversion
  aw     =        k1*ax + k2*ax2 + k3*ax3 + k1*k1*ax5   ; mixture

  ke     linen    p4, 0.05, p3, 0.2
         out      .25 * ke * aw ;  divide by 4 because we mixed four signals
endin
```

Instrument 5 creates a mixture by waveshaping with a fifth order polynomial. We have to take care with the amplitudes of all the signals, and to err on the cautious side the mixture is multiplied with 0.25 (because we mixed four signals) before sending it to the output. Often one needs to do some simple calculations or use a graph plotting program to visualise functions. No opcode is needed to transform the bipolar sinusoid k1 to the unipolar sinusoid k2, we simply use arithmetic.

Apart from polynomials, some other functions work well for waveshaping, such as sin, cos, and tanh. In particular, these three functions all have the range [-1,1], which makes it easy to reason about the output amplitude. On the other hand, these functions have an infinite Taylor expansion, which means that they do not produce bandlimited output. Used with input signals of moderate amplitude that should not pose a great problem. A few things to try:

```
ax     oscili   k1, p5, 1   ; make the amplitude time-variable
aw1    =        cos(ax)
aw2    =        sin(ax)
aw3    =        tanh(ax)
```

Notice that this example deviates from the usual syntax where the opcode goes in the second column. Standard mathematical functions are available as inline operators.

Ring modulation is simply the multiplication of two audio rate signals. If the signals are pure sinusoids at f_1 and f_2 Hz, their ring modulated product has components at $f_1 \pm f_2$ Hz. Wave-shaping and ring modulation can be combined in many ways.

```
instr 6
; waveshaper—ring modulator
; inharmonic : two oscillators --> waveshaper * rm

  iamp   =        p4
  ax     oscili   iamp, p5, 1
  ay     oscili   iamp, p6, 1
  az     oscili   1, p7, 1

  aw     =        .125*(ax+ay)^3; mix ax+ay, then waveshape
  arm    =        az*aw         ; multiplication is ring modulation
  kxp    expon    1, p3, 0.001  ; exponential envelope

         out      kxp * arm
endin

; score file
f1 0 4096 10 1

i6   0 1    0.85   400  325  110
i6   1 2    0.64   220  500  311
i6   3 1.2  0.75   244  344  433
i6   4 1.5  0.90   177  646  535
e
```

Instrument 6 uses three oscillators with frequencies specified in p5, p6, and p7 in the score file. The first two oscillators are mixed together, then we take the third power of the sum. Waveshaping of a compound tone, such as two sinusoids, produces not only higher harmonic partials of each of the sinusoids, but also some intermodulation products such as sums and differences of the frequencies. Finally, ring modulation of this waveshaped tone with a third frequency produces an even denser spectrum. Rich inharmonic timbres can be made if the three frequencies are mutually inharmonic.

The **expon** opcode generates an exponential envelope suitable for percussive sounds. Its first parameter is the amplitude at the outset, the second parameter is a duration, and the third parameter is the final amplitude after this duration has elapsed. The third parameter should be small but positive. Here we have used p3 as the duration parameter, so the envelope declines to the value 0.001 during one complete note, regardless of the note's length. When note lengths differ in the score, this means that the envelope will shrink or stretch accordingly. One might want to use a fixed duration as the second parameter, and perhaps also avoid clicks in the beginning and end by multiplying with a linen envelope.

Frequency modulation

```
instr 7 ; basic 2op FM

 imod    =        p7
 amod    oscili   1, p5, 1
 acar    oscili   1, p6*(1+imod*amod), 1
 ke      linen    p4, 0.05, p3, 0.15

         out      ke*acar
endin

; score
f1 0 4096 10 1

i7   1    1    0.75 330  330  1.5
i7   2    1    0.81 444  515  0.7
i7   3    1    0.64 275  523  1.0
e
```

In the simplest form, frequency modulation (FM) uses two oscillators, a modulator and a carrier. The modulator in this example comes from an oscillator with a fixed frequency, given by p5, and it modulates the carrier frequency p6 by an amount given by the modulation index, imod, which comes from p7. The expression p6*(1+imod*amo) specifies an instantaneous frequency which varies around the carrier p6 by the relative amount imod.

Next we introduce a morphing FM instrument using two modulators and one carrier. In the beginning of the note, the modulation is taken from the first oscillator (*ax*), and at the end it is taken from the second oscillator (*ay*). During the note, the first oscillator linearly fades out while the second fades in.

```
instr 8
; morphing dual modulator fm

 ax    oscili   1, p5, 1
 ay    oscili   1, p6, 1
 kl    line     1, p3, 0
 am    =        kl*ax + (1-kl)*ay
 ind   =        1.5  ; modulation index
 az    oscili   1, p7 * (1 + ind*am), 1
 ke    linen    p4, 0.05, p3, 0.08
       out      ke*az
endin
```

The new opcode **line** is similar to expon, it takes three parameters which are the starting value, the duration, and the finishing value. The control rate signal kl ramps from 1 to 0 during each note. A cross-fade is made by the expression

```
kl*ax + (1-kl)*ay.
```

Smoother transitions between two different modulation frequencies can be achieved this way than using a single modulator with sliding frequency.

Score file syntax

If any instrument initiated from the score file refers to a function table, it needs to be available at the time when it is used. Therefore, a statement such as

```
f1 0 4096 10 1
```

should always go on the first line of the score file. If no function table is needed, this line can be omitted. Then follows instrument statements, which are the letter **i** followed by the instrument number, followed by at least two more p-fields. The first two have a fixed meaning: p2 is the start time of a note, p3 is its duration. More p-fields can be used, and must correspond to the same number of references to p-fields in the instrument. The score ends with the letter **e**.

The overall tempo of a score may be changed by putting a line such as

```
t 0 72
```

before any note events. This means that at time 0, the tempo will be set to 72 beats per minute. Tempo changes such as ritardandi can be achieved by adding another beat location and a new tempo, for instance

```
t 0 54 11 80
```

will create an accelerando from 54 BPM at the beginning, to 80 BPM 11 beats into the score.

New sections can be introduced with the letter **s**. When a new section is introduced the time (p2) begins from 0 again, but note events are written to the sound file after the end of the previous section.

Several lines may be commented out by putting them between /* and */, as in a c-style comment. This works also in the orchestra file.

A special bracket notation may be used for specifying rational numbers or simple arithmetic expressions. Instead of writing a decimal number, let's say 1.33333, you may write it as the fraction [1+1/3]. This is particularly handy for pure intonation and tuplet rhythms.

To repeat a value in a certain p-field from the note above, a single period (.) may be inserted in that p-field. For this to work, all notes must refer to the same instrument.

```
f1 0 4096 10 1
t  0  48   ; tempo 48 BPM

i1   0    3    0.7    [3/5]     ; pure intonation melody
i1   3    2    0.2    [8/9]
i1   5    2    0.8    [1/1]
s                                ; begin new section
i1   0    1    0.6    [5/4]
i1   1    2    .      [9/8]     ; dot in p4 means repeat last value
i1   3    1    .      [4/3]
e
```

When using ratio notation for frequencies, the instrument should define a reference frequency, such as [1/1] = 440 Hz. Then the frequency ifrq = 440*p5 can be used by the oscillators.

Subtractive synthesis

There are lots of different filters in Csound. We have already encountered the one-pole highpass **atone**. There is a corresponding one-pole lowpass **tone** filter. Both take the input signal as the first argument and cutoff frequency as second argument.

Four Butterworth filters are available: **butlp** for lowpass, **buthp** for highpass, **butbp** for bandpass, and **butbr** for band-reject.

```
ay   butlp    ax, kfreq   ; cutoff frequency may also be i-rate or a-rate
ay   buthp    ax, kfreq
ay   butbp    ax, kfreq, kband   ; centre frequency and bandwidth in Hz
ay   butbr    ax, kfreq, kband   ; band-reject
```

For a simple demonstration we could use white noise as input. The **rand** opcode takes one argument, specifying the maximum amplitude of the noise.

```
instr 9
  ano    rand     1.0  ; random noise, amplitude 1
  abr    butbr    ano, 1400, 500
  abp    butbp    abr, 600, 300
         out      abp
endin
```

Sound input

Csound can be used as a sampler by reading soundfiles and playing them at different speeds and starting the playback from an arbitrary position. It can also be used as a customisable effects processor.

Input from a soundfile is possible by several opcodes. The **diskin** opcode can read most common soundfile formats with up to 40 channels. The first and mandatory parameter is the file name, usually given as a text string. There are several optional parameters, the first of which is the playback speed where 1 is the original speed, and the second parameter is a skip time in seconds, setting the point in the input soundfile from which to begin reading. The speed parameter may be k-rate, so it's possible to dynamically warp the playback speed.

When reading a mono soundfile, diskin sends the read samples to a single a-rate variable, but if there are more than one input channel, there must be a comma separated list of input variables matching the number of channels.

```
ax       diskin   ”toot.wav”, kspeed, iskip ; mono input
al, ar   diskin   ”teet.wav”                ; stereo input
```

Here it is assumed that the soundfiles toot.wav and teet.wav exist in the same folder as the orchestra file. Instead of just reading one fixed soundfile, diskin may be given a p-field containing a text string with the file name. Then the score may specify different soundfiles to read for each note. It might look something like this:

```
; inside an instrument:
ax       diskin   p4
; in the score file:
i1 0 3 ”toot.wav”
```

Looping soundfiles is possible by setting the fourth parameter of diskin to 1. The soundfile then plays to the end, and then starts over from the beginning. If kspeed is negative, the playback will be backwards. Note that you cannot play a soundfile backwards starting from the beginning unless it is set to looping, but you could set iskip to the end of the file and play it backwards.

```
kspeed   =        -1/2                          ; backwards at half speed
ax       diskin   ”toot.wav”, kspeed, iskip, 1  ; last 1: looping
```

Delay and feedback

One of the most important components in audio effects is delay lines. A wide range of effects can be built using delays and various other filters. The **delay** opcode takes a mono audiorate variable as its first parameter and a fixed delay length in seconds as its second parameter.

```
instr 10
  asig   diskin     ”toot.wav”
  ad     delay      asig, 0.5 ; delay asig by half a second
         out        .5 * (asig + ad)
endin
```

In this example the delay is added to the original signal. This is the same as a FIR (finite impulse response) comb filter. If the delay time is set much shorter, less than 1/20 of a second, the repetition will colour the sound at the frequency which is the inverse of the delay length. For more pronounced effects we will add feedback.

In Csound, variables have to be declared before being used, but feedback is circular or self-referential. The way to solve it is to introduce a feedback variable using the opcode **init**. It takes one parameter, the initial value of the variable at the start of a note. Then we can do feedback like this:

```
ay   init   0
ay   =      ax + ay
```

However, there is no such thing as instantaneous feedback in a digital system, so the last line could be written in mathematical notation as $y_n = x_n + y_{n-1}$. Now we may combine feedback with delay:

```
instr 11
  ain    diskin   ”toot.wav”
  adl    init     0                    ; feedback variable
  adl    delay    ain + 0.5*adl, 1/3   ; add previous value of adl to itself

         out      ain + adl
endin
```

This instrument delays the input by a third of a second, adds the delay to itself with a gain factor of 1/2, meaning that each repetition is half as loud as the previous one, and the mix of the dry signal and the delay are sent to the output.

The character of a delay effect can be further shaped by inserting a filter in the feedback loop. Try experimenting with tone, atone, butlp, buphp, butbp, or butbr.

There is also a **vdelay** that permits variable delay time. Modulation of delay time is the basis of chorus and flanger effects, and it can be used to impose a vibrato on a sound. For no particular reason, the opcode **vdelay** differs from the fixed length version by using milliseconds as time unit instead of seconds. All delays need to allocate a certain amount of memory, one memory location for each sample of delay. In the fixed delay version the memory allocation is handled automatically, but in the variable delay the maximum delay length one intends to use must be provided as the third parameter:

```
adl       vdelay    ax, ktime, imax ; ktime may vary between 0 and imax [ms]
```

A vibrato can be created by modulating the delay time around an average value.

```
instr 12 ; vibrato
  ain     diskin    ”toot.wav”
  kdel    oscili    1, p4                  ; sinusoidal modulation at rate p4
  imax    =         200                    ; longest delay 200 ms
  kdel    =         1 + 0.8*kdel
  adl     vdelay    ain, 100*kdel, imax    ; kdel varies between 20 and 180 ms
          out       adl
endin
```

A time-warping sampler

Using **diskin** we can dynamically change the playback speed of a soundfile. Now we will try something interesting: the amplitude envelope of the soundfile will determine the playback speed. Although there is an opcode for virtually anything, including extracting the RMS amplitude, it is easy to build an envelope follower from simpler components. We will square the input signal and average it using a lowpass filter with very low cutoff. Lastly, we take the square root and use that as the amplitude envelope kamp.

In keeping with Csound's usual quirkiness k-rate variables have their own set of filters. The **port** opcode is designed to create a portamento, which works well for the task of envelope following. The first argument to port is the input signal, and the second argument is not a cutoff frequency, but a ”half-time” or the time it takes the signal to reach half-ways to its goal. In the world of modular synthesizers, this would be called a slew rate limiter.

Suppose we want the playback speed kspeed to vary inversely with amplitude; that is, when the input signal is loud it plays slowly, and when it is soft it runs faster. We might try something like

```
kspeed = 1.2 - kamp
```

which gives a fastest speed of 1.2 when the input is silent, and a theoretical slowest speed of 0.2 if kamp approaches 1. But we don't know how the input signal's amplitude varies over time, and how much it will affect the playback speed. It may be necessary to progress by trial and error. There is a convenient opcode, **printk**, for printing out variables at regular time intervals. By printing out the values of the amplitude envelope we can better adjust the values.

```
instr 13
  kspeed   init   1
  ain      diskin "toot.wav", kspeed
  kamp     =      k(ain)^2      ; convert ain to k-rate and square it
  kamp     port   kamp, 0.125   ; slew rate limit kamp
  kamp     =      sqrt(kamp)
  kspeed   =      1.2 - 2.0*kamp; set parameters by trial and error
           printk 0.3, kamp     ; print out kamp every 0.3 sec
           out    ain
endin

; score
i13  0  8
e
```

Control structures

Looping is a fundamental programming construct. Csound instruments already implicitly loop from the first to the last line of the instrument definition as each sample is calculated. Explicit loops can be introduced using labels and **goto** statements. There are also conditional statements and comparisons. These control structures work only with i-rate and k-rate variables. Using them with a-rate variables requires an embarrassing workaround: the k-rate has to be set equal to the a-rate in the orchestra header, then a-rate variables have to be converted to k-rate before being used in logical expressions, and then converted back again.

```
; .orc
sr = 48000
kr = 48000
nchnls = 1

0dbfs = 1.0

instr 14
  ax   oscili 0.7, 244
  kp   =      k(ax)             ; convert ax to k-rate
  kp   =      (kp > 0 ? kp : 0)  ; if kp > 0, keep value, otherwise make it 0
  ap   =      a(kp)             ; convert kp to a-rate
       out    ap
endin

; .sco
i14    0    2
e
```

This instrument does half-wave rectification of a sine at 244 Hz, just for the purpose of illustration. As always, there are other ways to do half-wave rectification.

The opcode cmp allows a-rate arguments and a text string indicating which comparison operator to use. In the next example,

```
cmp ax, "<", ay
```

outputs 1 at each sample such that ax < ay, and a 0 otherwise. However, this doesn't help us with branching at audio rate, we still have to convert a-rate to k-rate. Here we use **if**, **goto**, and labels to transfer control.

```
instr 15

  ax       oscili   1, 400
  ay       oscili   1, 507
  az       =        ax
  ac       cmp      ax, "<", ay     ; ac=1 if ax<ay, ac=0 otherwise
  kg       =        k(ac)           ; convert ac to k-rate

  if       kg > 0   goto summation  ; because if doesn't do a-rate
  goto output

summation:
  az   =    .5*(ax + ay)

output:
         out    az

endin
```

This instrument compares two sinusoids, ax and ay, and if ax < ay their scaled sum is output, otherwise only ax is output.

Global variables

Sometimes it is useful to share a variable between instruments. For example, one instrument makes a dry percussive sound, and we wish to add reverberation. If we put the reverb inside the percussive instrument it will turn off at the end of each note. The reverb must be its own instrument which is turned on for as long as the percussion instrument is playing, and a few more seconds at the end to allow the reverb to ring out. To communicate between the instruments we need global variables.

A global variable can be i-rate, k-rate, or a-rate just like local variables, and its name must begin with gi, gk, or ga. If there is any feedback the global variable should be initiated before and outside the instruments that use it.

```
sr = 48000
kr = 48000
nchnls = 1

0dbfs  =    1.0
gadel  init 0      ; global a-rate signal for delay with feedback


instr 16; perk

  ke     expon    p4, 0.25, 0.01
  ax     oscili   ke, p5
  gadel  =        gadel + ax    ; add input ax to delay gadel
         out      ax            ; dry signal
endin


instr 99
  gadel    delay    0.75*gadel, 0.145  ; make each iteration decay by 0.75
           out      gadel              ; we want to hear it
endin
```

```
; score
i99  0    8  ; turn on and let it stay on while i3 plays

i3   0    1    0.85   165
i3   1    1    0.75   175
i3   3    1    0.78   243
e
```

Another use of a global variable might be to impose a slow pitch drift on an instrument persistently across individual notes. In the next example, the always-on instrument has a global variable gkpitch which slopes linearly from 1 to 0.9 during the length of the piece. The oscillator in the main instrument multiplies gkpitch with its pitch value.

```
gkpitch  init 1

instr 17
  ax   oscili   p4, p5 * gkpitch
       out    ax
endin

instr 98
  gkpitch  line 1, p3, 0.9
endin

; score
i98  0    60   ; assuming a one minute piece

i17    0    1    0.6    440
i17    1    1    0.7    330
; ... and a lot more i17 notes
e
```

This brings us to another point. Entering lots of notes and p-fields into score files is a tedious endeavour. On the other hand, the simple format of a score file makes it suitable for automated generation by algorithmic procedures. Any programming or scripting language can be put to service.

Utilities

Csound comes equipped with several utility programs, accessed from the command line:

```
csound -U [utility program name] [flags] [filenames]
```

The utilities include lpc analysis (linear predictive coding, typically used with vocal sounds), phase vocoder and other kinds of spectral representation and processing, and convolution. For documentation, see: https://csound.com/docs/manual/UtilitySoundfile.html

There are also utility programs for denoising (dnoise), mixing (mixer), extracting the amplitude envelope of a soundfile to a text file (envext), and others.

Miscellaneous

Be aware that much more can be done with Csound than what is covered in this short guide, including realtime sound input and output, building custom GUIs, and communicating with MIDI devices. Newer versions allow declaring arrays of variables instead of just scalars. User defined opcodes can be created, and more inline functional syntax has been added in recent versions.

Further reading

Quick references:
https://csound.com/docs/manual/MiscQuickref.html
https://flossmanual.csound.com/introduction/preface

For anyone:
Boulanger, R. (2000). The Csound Book. The MIT Press.

For programmers:
Boulanger, R. and Lazzarini, V. (2011): The Audio Programming Book. The MIT Press.

Essential opcodes reference

Variable and argument names begin with i, k, or a depending on which rate they apply to. Argument names beginning with an x may be i, k, or a-rate. This is a significantly abbreviated reference; many opcodes exist both in a-rate and k-rate versions, and some also as i-rate. Some optional arguments have been left out.

Orchestra header

```
sr = n                ; sample rate
kr = n                ; control rate
nchnls = n            ; number of channels in output
0dbfs = x             ; numerical value of highest representable amplitude value
instr n               ; begin instrument definition
endin                 ; end instrument definition
```

Signal generators

```
asig   oscili    xamp, xfreq [, ifunction, iphase]
ksig   oscili    kamp, kfreq [, ifunction, iphase]
asig   rand      xamp                    ; can also output k-rate

ksig   linen     kamp, irise, idur, idecay      ; max. amplitude, times in sec.
ksig   line      ia, idur, ib  ; line segment from ia to ib over idur seconds
ksig   expon     ia, idur, ib  ; exponential segment from ia to ib over idur
```

Filters, signal modifiers

```
asig   tone        ain, kcutoff              ; one-pole lowpass
aisg   atone       ain, kcutoff              ; one-pole highpass
asig   butlp       ain, kcutoff              ; Butterworth lowpass
asig   buthp       ain, kcutoff              ; Butterworth highpass
asig   butbp       ain, xfreq, xband         ; bandpass, frequency and bandwidth
asig   butbr       ain, xfreq, xband         ; band-reject

asig   delay       ain, idel                 ; delay ain by idel seconds
asig   vdelay      ain, kdel, imax           ; delay ain by kdel (<imax) msec

ksig   port        kx, ihalf                 ; portamento of kx, ihalf in sec
```

Input and output

```
asig      diskin   infile, kspeed [, iskiptime [, iwrap]]
al, ar    diskin   infile, kspeed [ ...]     ; a-variables on the left match
                                             ; number of channels in infile

          out      asig                      ; mono output
          outs     aleft, aright             ; stereo output

xvar      init     init_value                ; for feedback variables
          printk   itime, ksig               ; print ksig every itime seconds
```

Logic, control structures

```
if [condition] goto label                    ; i or k-rate condition
asig   cmp      ax, "rel", ay                ; rel is one of <, <=, ==, >, >=
kz     =        (kx <rel> ky ? ka : kb)      ; <rel> is one of <, <=, ==, >, >=
```

Mathematical operators and functions

```
xsig   =        [expression]
asig   =        ax^ipow       ; raise ax to ipow, k-rate exponents also allowed
asig   =        ax <op> ay    ; <op> is one of + - * / or %, all rates

asig   =        funct(x)              ; funct is one of the following:
sqrt(x), sin(x), cos(x),
tan(x), tanh(x), log(x)               ; x may be any rate
int(x)                                ; integer part
frac(x)                               ; fractional part
abs(x)                                ; absolute value

kx     =        k(ax)                 ; rate conversion from a to k
ax     =        a(kx)                 ; rate conversion from k to a
```