THE PUBLIC CSOUND REFERENCE MANUAL

SUPPLEMENT – TUTORIALS

by Barry Vercoe, Media Lab MIT
and contributors

Edited by John ffitch, Richard Boulanger,
Jean Piché, and David Boothe

Copyright Notice

Contributors

In addition to the core code developed by Barry Vercoe at MIT, a large part of the Csound code was modified, developed and extended by an independent group of programmers, composers and scientists. Copyright to this code is held by the respective authors:

Mike Berry
Eli Breder
Michael Casey
Michael Clark
Perry Cook
Sean Costello
Richard Dobson
Mark Dolson
Rasmus Ekman
Dan Ellis
Tom Erbe
John ffitch
Bill Gardner
Matt Ingalls
Richard Karpen
Victor Lazzarini
Allan Lee
David Macintyre
Peter Neubäcker
Marc Resibois
Gabriel Maldonado
Hans Mikelson
Paris Smaragdis
Greg Sullivan
Robin Whittle

This manual was compiled from the canonical Csound Manual sources maintained by John ffitch, Richard Boulanger, Jean Piche and David Boothe.

Editor’s Preface

Learning Csound can be a daunting experience. The tutorials contained in this book are intended to give you a head start in this process, and were written by people involved in writing, expanding, and maintaining the Csound code itself.

This volume is intended as a companion to The Public Csound Reference Manual, which is indispensable to the Csound user. Refer to the appropriate sections in the Reference Manual as you work through these tutorials. Try the examples yourself. These exercises will give you a solid, fundamental knowledge of the most powerful software synthesis program that is freely and widely available.

The Reference Manual and the example orchestra and scores are available where you obtained this volume.

The definitive source of information on Csound will be *The Csound Book: Perspectives in Software Synthesis, Sound Design, Signal Processing, and Programming*, edited by Richard Boulanger, and to be published by MIT Press in January, 2000.

Another useful source of information is the “MIT Csound FrontPage” at

```
http://mitpress.mit.edu/e-books/csound/frontpage.html.
```

Finally, all Csound users are invited to subscribe to the Csound email list. Here is where there are ongoing discussions of any issues concerning Csound users, bug reports, announcements of new Csound versions, etc. If you are having a problem with Csound you cannot resolve, there is usually someone on the Csound mailing list who can help. For instructions on how to subscribe, send an email to

```
csound-request@maths.ex.ac.uk
```

In the first line of the message, put

```
subscribe name@host
```

After your message has been processed (about 5 minutes), you will receive a message with further instructions. Follow these last few instructions, and your email address will be added to the Csound Mailing list.

Enjoy your Csound experience.

David M. Boothe
Dallas, Texas USA
April, 1999

Table of Contents

This page left blank.

1 A BEGINNING TUTORIAL

by Barry Vercoe, Massachusetts Institute of Technology

1.1 The Orchestra File

Csound runs from two basic files: an *orchestra* file and a *score file*. The orchestra file is a set of *instruments* that tell the computer how to synthesize sound; the score file tells the computer when. An instrument is a collection of modular statements which either *generate* or *modify* a signal; signals are represented by *symbols*, which can be "patched" from one module to another. For example, the following two statements will generate a 440 Hz sine tone and send it to an output channel:

```
 asig    oscil      10000, 440, 1
         out        asig
```

The first line sets up an oscillator whose controlling inputs are an amplitude of 10000, a frequency of 440 Hz, and a waveform number, and whose output is the audio signal *asig*. The second line takes the signal *asig* and sends it to an (implicit) output channel. The two may be encased in another pair of statements that identify the instrument as a whole:

```
instr    1
 asig    oscil      10000, 440, 1
         out        asig
endin
```

In general, an orchestra statement in Csound consists of an action symbol followed by a set of input variables and preceded by a result symbol. Its *action* is to process the inputs and deposit the result where told. The meaning of the input variables depends on the action requested. The 10000 above is interpreted as an amplitude value because it occupies the first input slot of an **oscil** unit; 440 signifies a frequency in Hertz because that is how an **oscil** unit interprets its second input argument; the waveform number is taken to point indirectly to a stored function table, and before we invoke this instrument in a score we must fill function table #1 with some waveform.

The output of Csound computation is not a real audio signal, but a stream of numbers which describe such a signal. When written onto a sound file these can later be converted to sound by an independent program; for now, we will think of variables such as *asig* as tangible audio signals.

Let us now add some extra features to this instrument. First, we will allow the pitch of the tone to be defined as a *parameter* in the score. Score parameters can be represented by orchestra variables which take on their different values on successive notes. These variables are named sequentially: p1, p2, p3, ... The first three have a fixed meaning (see the **Score File**), while the remainder are assignable by the user. Those of significance here are:

p3 - duration of the current note (always in seconds).
p5 - pitch of the current note (in units agreed upon by score and orchestra).

Thus in

```
 asig     oscil        10000, p5, 1
```

the oscillator will take its pitch (presumably in cps) from score parameter 5.

If the score had forwarded pitch values in units other than cycles-per-second (Hertz), then these must first be converted. One convenient score encoding, for instance, combines *pitch class* representation (00 for C, 01 for C#, 02 for D, ... 11 for B) with *octave* representation (8. for middle C, 9. for the C above, etc.) to give pitch values such as 8.00, 9.03, 7.11. The expression

```
          cpspch(8.09)
```

will convert the pitch A (above middle C) to its cps equivalent (440 Hz). Likewise, the expression

```
          cpspch(p5)
```

will first read a value from p5, then convert it from octave.pitch-class units to cps. This expression could be imbedded in our orchestra statement as

```
 asig     oscil        10000, cpspch(p5), 1
```

to give the score-controlled frequency we sought.

Next, suppose we want to shape the amplitude of our tone with a linear rise from 0 to 10000. This can be done with a new orchestra statement

```
 amp      line         0, p3, 10000
```

Here, *amp* will take on values that move from 0 to 10000 over time p3 (the duration of the note in seconds). The instrument will then become

```
instr     1
 amp      line         0, p3, 10000
 asig     oscil        amp, cpspch(p5), 1
          out          asig
endin
```

The signal *amp* is not something we would expect to listen to directly. It is really a variable whose purpose is to control the amplitude of the audio oscillator. Although audio output requires fine resolution in time for good fidelity, a controlling signal often does not need that much resolution. We could use another kind of signal for this amplitude control

```
 kamp     line        0, p3, 10000
```

in which the result is a new kind of signal kamp. Signal names up to this point have always begun with the letter a (signifying an *audio* signal); this one begins with k (for *control*). Control signals are identical to audio signals, differing only in their resolution in time. A control signal changes its value less often than an audio signal, and is thus faster to generate. Using one of these, our instrument would then become

```
instr     1
 kamp     line        0, p3, 10000
 asig     oscil       kamp, cpspch(p5), 1
          out         asig
endin
```

This would likely be indistinguishable in sound from the first version, but would run a little faster. In general, instruments take constants and parameter values, and use calculations and signal processing to move first towards the generation of control signals, then finally audio signals. Remembering this flow will help you write efficient instruments with faster execution times.

We are now ready to create our first orchestra file. Type in the following orchestra using the system editor, and name it "intro.orc".

```
sr        =           44100                 ; audio sampling rate is 44.1 kHz
kr        =           4400                  ; control rate is 4410 Hz
ksmps     =           10                    ; number of samples in a
                                            ; control period (sr/kr)
nchnls    =           1                     ; number of channels of
                                            ; audio output

instr     1
 kctrl    line        0, p3, 10000          ; amplitude envelope
 asig     oscil       kctrl, cpspch(p5), 1  ; audio oscillator
          out         asig                  ; send signal to channel 1
endin
```

It is seen that comments may follow a semi-colon, and extend to the end of a line. There can also be blank lines, or lines with just a comment. Once you have saved your orchestra file on disk, we can next consider the score file that will drive it.

1.2 The Score File

The purpose of the score is to tell the instruments when to play and with what parameter values. The score has a different syntax from that of the orchestra, but similarly permits one statement per line and comments after a semicolon. The first character of a score statement is an **opcode**, determining an action request; the remaining data consists of numeric parameter fields (pfields) to be used by that action.

Suppose we want a sine-tone generator to play a pentatonic scale starting at C-sharp above middle-C, with notes of 1/2 second duration. We would create the following score:

```
;  a sine wave function table
f1 0 256 10 1

;  a pentatonic scale
i1   0    .5   0.   8.01
i1  .5    .    .    8.03
i1 1.0    .    .    8.06
i1 1.5    .    .    8.08
i1 2.0    .    .    8.10
e
```

The first statement creates a stored sine table. The protocol for generating wave tables is simple but powerful. Lines with opcode **f** interpret their parameter fields as follows:

p1 - function table number being created
p2 - creation time, or time at which the table becomes readable
p3 - table size (number of points), which must be a power of two or one greater
p4 - generating subroutine, chosen from a prescribed list.

Here the value 10 in p4 indicates a request for subroutine **GEN10** to fill the table. **GEN10** mixes harmonic sinusoids in phase, with relative strengths of consecutive partials given by the succeeding parameter fields. Our score requests just a single sinusoid. An alternative statement:

```
f1 0 256 10 1 0 3
```

would produce one cycle of a waveform with a third harmonic three times as strong as the first.

The **i** statements, or note statements, will invoke the p1 instrument at time p2, then turn it off after p3 seconds; it will pass all of its p-fields to that instrument. Individual score parameters are separated by any number of spaces or tabs; neat formatting of parameters in columns is nice but unnecessary. The dots in p-fields 3 and 4 of the last four notes invoke a *carry feature*, in which values are simply copied from the immediately preceding note *of the same instrument*. A score normally ends with an **e** statement.

The unit of time in a Csound score is the beat. In the absence of a **tempo** statement, one beat takes one second. To double the speed of the pentatonic scale in the above score, we could either modify p2 and p3 for all the notes in the score, or simply insert the line

```
t 0 120
```

to specify a tempo of 120 beats per minute from beat 0.

Two more points should be noted. First, neither the **f** statements nor the **i** statements need be typed in time order; Csound will sort the score automatically before use. Second, it is permissable to play more than one note at a time with a single instrument. To play the same notes as a three-second pentatonic chord we would create the following:

```
;    a sine wave function
f1   0  256   10    1
;    five notes at once
i1   0    3    0    8.01
i1   0    .    .    8.03
i1   0    .    .    8.06
i1   0    .    .    8.08
i1   0    .    .    8.10
e
```

Now go into the editor once more and create your own score file. Name it "intro.sco". The next section will describe how to invoke a Csound orchestra to perform a Csound score.

1.3 The Csound Command

To request your orchestra to perform your score, type the command

```
csound intro.orc  intro.sco
```

The resulting performance will take place in three phases:

1. sort the score file into chronological order. If score syntax errors are encountered they will be reported on your console.
2. translate and load your orchestra. The console will signal the start of translating each *instr* block, and will report any errors. If the error messages are not immediately meaningful, translate again with the *verbose* flag turned on:

   ```
   csound  -v  intro.orc  intro.sco
   ```
3. fill the wave tables and perform the score. Information about this performance will be displayed throughout in messages resembling

   ```
    B  4.000 .. 6.000   T 3.000  TT  3.000  M    7929.    7929.
   ```

A message of this form will appear for every *event* in your score. An event is defined as any change of state (as when a new note begins or an old one ends). The first two numbers refer to beats in your original score, and they delimit the current segment of sound synthesis between successive events (e.g. from beat 4 to beat 6). The second beat value is next restated in real seconds of time, and reflects the *tempo* of the score. That is followed by the Total Time elapsed for all sections of the score so far. The last values on the line show the maximum amplitude of the audio signal, measured over just this segment of time, and reported separately for each channel.

Console messages are printed to assist you in following the orchestra's handling of your score. You should aim at becoming an intelligent reader of your console reports. When you begin working with longer scores and your instruments no longer cause surprises, the above detail may be excessive. You can elect to receive abbreviated messages using the -*m* option of the Csound command.

When your performance goes to completion, it will have created a sound file named *tes*t in your soundfile directory. You can now listen to your sound file by typing

```
play test
```

If your machine is fast enough, and your Csound module includes user access to the audio output device, you can hear your sound as it is being synthesized by using a command like:

```
csound  -o  devaudio  intro.orc  intro.sco
```

1.4 More about the Orchestra

Suppose we next wished to introduce a small vibrato, whose rate is 1/50 the frequency of the note (i.e. A440 is to have a vibrato rate of 8.8 Hz.). To do this we will generate a control signal using a second oscillator, then add this signal to the basic frequency derived from p5. This might result in the instrument

```
instr     1
 kamp     line        0, p3, 10000
 kvib     oscil       2.75, cpspch(p5)/50, 1
 a1       oscil       kamp, cpspch(p5)+kvib, 1
          out         a1
endin
```

Here there are two control signals, one controlling the amplitude and the other modifying the basic pitch of the audio oscillator. For small vibratos, this instrument is quite practical; however it does contain a misconception worth noting. This scheme has added a sine wave deviation to the cps value of an audio oscillator. The value 2.75 determines the *width* of vibrato in cps, and will cause an A440 to be modified about one-tenth of one semitone in each direction (1/160 of the frequency in cps). In reality, a cps deviation produces a different musical interval above than it does below. To see this, consider an exaggerated deviation of 220 cps, which would extend a perfect 5th above A440 but a whole octave below. To be more correct, we should first convert p5 into a *true decimal octave* (not cps), so that an *interval* deviation above is equivalent to that below. In general, pitch modification is best done in true octave units rather than pitch-class or cps units, and there exists a group of pitch converters to make this task easier. The modified instrument would be

```
instr     1
 ioct     =           octpch(p5)
 kamp     line        0, p3, 10000
 kvib     oscil       1/120, cpspch(p5)/50, 1
 asig     oscil       kamp, cpsoct(ioct+kvib), 1
          out         asig
endin
```

This instrument is seen to use a third type of orchestra variable, an i-rate variable. The variable *ioct* receives its value at an *initialization* pass through the instrument, and does not change during the lifespan of this note. There may be many such init time calculations in an instrument. As each note in a score is encountered, the event space is allocated and the instrument is initialized by a special pre-performance pass. i-rate variables receive their values at this time, and any other expressions involving just constants and i-rate variables are evaluated. At this time also, modules such as **line** will set up their target values (such as beginning and end points of the line), and units such as **oscil** will do phase setup and other bookkeeping in preparation for performance. A full description of init-time and performance-time activities, however, must be deferred to a general consideration of the orchestra syntax.

This page left blank.

2 AN INSTRUMENT DESIGN TOOTORIAL

by Richard Boulanger, Berklee College of Music

2.1 Toot Introduction

Csound instruments are created in an *orchestra* file, and the list of notes to play is written in a separate *score* file. Both are created using a standard word processor. When you run Csound on a specific orchestra and score, the score is sorted and ordered in time, the orchestra is translated and loaded, the wavetables are computed and filled, and then the score is performed. The score drives the orchestra by telling the specific instruments when and for how long to play, and what parameters to use during the course of each note event.

Unlike today's commercial hardware synthesizers, which have a limited set of oscillators, envelope generators, filters, and a fixed number of ways in which these can be interconnected, Csound's power is not limited. If you want an instrument with hundreds of oscillators, envelope generators, and filters you just type them in. More important is the freedom to interconnect the modules, and to interrelate the parameters which control them. Like acoustic instruments, Csound instruments can exhibit a sensitivity to the musical context, and display a level of "musical intelligence" to which hardware synthesizers can only aspire.

Because the intent of this tutorial is to familiarize the novice with the syntax of the language, we will design several simple instruments. You will find many instruments of the sophistication described above in various Csound directories, and a study of these will reveal Csound's real power.

The Csound *orchestra file* has two main parts:

1. *the header section* - defining the sample rate, control rate, and number of output channels.

2. *the instrument section* - in which the instruments are designed.

2.1.1 THE HEADER SECTION

A Csound orchestra generates signals at two rates - an audio sample rate and a control sample rate. Each can represent signals with frequencies no higher than half that rate, but the distinction between audio signals and sub-audio control signals is useful since it allows slower moving signals to require less compute time. In the header below, we have specified a sample rate of 44.1 kHz, a control rate of 4410 Hz, and then calculated the number of samples in each control period using the formula: ksmps = sr / kr

```
sr       =           44100
kr       =           4410
ksmps    =           10
nchnls   =           1
```

In Csound orchestras and scores, spacing is arbitrary. It is important to be consistent in laying out your files, and you can use spaces to help this. In the Tutorial Instruments shown below you will see we have adopted one convention. The reader can choose his or her own.

2.1.2 The Instrument Section

All instruments are numbered and are referenced thus in the score. Csound instruments are similar to patches on a hardware synthesizer. Each instrument consists of a set of "unit generators," or software "modules," which are "patched" together with "i/o" blocks – i-, k-, or a-rate variables. Unlike a hardware module, a software module has a number of variable "arguments" which the user sets to determine its behavior. The four types of variables are:

```
setup only
i-rate variables, changed at the note rate
k-rate variables, changed at the control signal rate
a-rate variables, changed at the audio signal rate
```

2.1.3 Orchestra Statements

Each statement occupies a single line and has the same basic format:

```
result   action      arguments
```

To include an oscillator in our orchestra, you might specify it as follows:

a1 **oscil** 10000, 440, 1

The three "arguments" for this oscillator set its amplitude (10000), its frequency (440Hz), and its wave shape (1). The output is put in i/o block *a1*. This output symbol is significant in prescribing the rate at which the oscillator should generate output – here the audio rate. We could have named the result anything (e.g. *asig*) as long as it began with the letter "a".

2.1.4 Comments

To include text in the orchestra or score which will not be interpreted by the program, precede it with a semicolon. This allows you to fully comment your code. On each line, any text which follows a semicolon will be ignored by the orchestra and score translators.

2.2 Toot 1: Play One Note

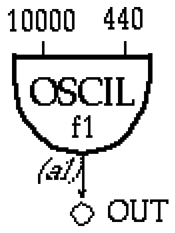
For this and all instrument examples, there exist `orchestra` and `score` files in the Csound subdirectory `tutorfiles` that the user can run to soundtest each feature introduced. The instrument code shown below is actually preceded by an *orchestra header section* similar to that shown above. If you are running on a RISC computer, each example will likely run in realtime. During playback (realtime or otherwise) the audio rate may automatically be modified to suit the local d-a converters.

The first orchestra file, called `toot1.orc` contains a single instrument which uses an **oscil** unit to play a 440Hz sine wave (defined by f1 in the score) at an amplitude of 10000.

```
         instr 1
a1       oscil       10000, 440, 1
         out         a1
         endin
```

Run this with its corresponding score file, *toot1.sco* :

```
f1     0    4096 10   1   ; use "GEN01" to compute a sine wave
i1     0    4             ; run "instr 1" from time 0
                          ; for 4 seconds
e                         ; indicate the "end" of the score
```



Toot 1: **oscil**

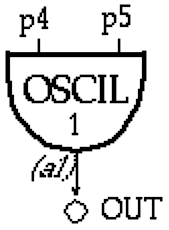
2.3 Toot 2: "P-Fields"

The first instrument was not interesting because it could play only one note at one amplitude level. We can make things more interesting by allowing the pitch and amplitude to be defined by parameters in the score. Each column in the score constitutes a parameter field, numbered from the left. The first three parameter fields of the **i** statement have a reserved function:

```
p1 = instrument number
p2 = start time
p3 = duration
```

All other parameter fields are determined by the way the sound designer defines his instrument. In the instrument below, the oscillator's amplitude argument is replaced by p4 and the frequency argument by p5. Now we can change these values at i-time, i.e. with each note in the score. The orchestra and score files now look like:

```
         instr 2
a1       oscil       p4, p5, 1     ; p4=amp
         out         a1            ; p5=freq
         endin


      f1     0     4096   10    1     ; sine wave
; instrument   start  duration    amp(p4)   freq(p5)
      i2       0      1          2000       880
      i2       1.5    1          4000       440
      i2       3      1          8000       220
      i2       4.5    1         16000       110
      i2       6      1         32000       55
e
```



Toot 2: **oscil** with p-fields

2.4 Toot 3: Envelopes

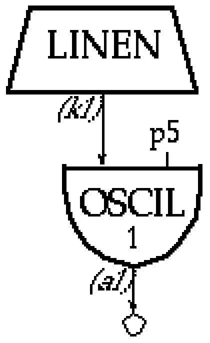
Although in the second instrument we could control and vary the overall amplitude from note to note, it would be more musical if we could contour the loudness during the course of each note. To do this we'll need to employ an additional unit generator **linen**, which the Csound reference manual defines as follows:

```
 kr      linen      kamp, irise, idur, idec
 ar      linen      xamp, irise, idur, idec
```

linen is a signal modifier, capable of computing its output at either control or audio rates. Since we plan to use it to modify the amplitude envelope of the oscillator, we'll choose the latter version. Three of linen's arguments expect **i**-rate variables. The fourth expects in one instance a k-rate variable (or anything slower), and in the other an x-variable (meaning a-rate or anything slower). Our **linen** we will get its amp from p4.

The output of the **linen** (*k1*) is patched into the *kamp* argument of an **oscil**. This applies an envelope to the **oscil**. The orchestra and score files now appear as:

```
         instr 3
 k1      linen      p4, p6, p3, p7      ; p4=amp
 a1      oscil      k1, p5, 1           ; p5=freq
         out        a1                  ; p6=attack time
         endin                          ; p7=release time


 f1  0  4096  10  1                     ; sine wave
;instr  start duration  amp(p4)  freq(p5)  attack(p6)  release(p7)
 i3     0     1        10000     440       .05         .7
 i3     1.5   1        10000     440       .9          .1
 i3     3     1         5000     880       .02         .99
 i3     4.5   1         5000     880       .7          .01
 i3     6     2        20000     220       .5          .5
 e
```



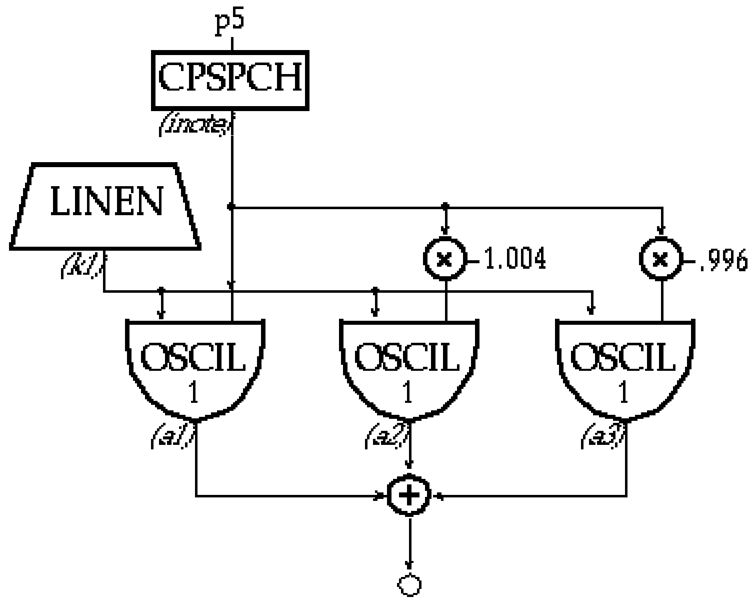
Toot 3: **linen** applied to **oscil**

2.5 Toot 4: Chorusing

Next we'll animate the basic sound by mixing it with two slightly de-tuned copies of itself. We'll employ Csound's **cpspch** value converter which will allow us to specify the pitches by octave and pitch-class rather than by frequency, and we'll use the **ampdb** converter to specify loudness in dB rather than linearly.

Since we are adding the outputs of three oscillators, each with the same amplitude envelope, we'll scale the amplitude before we mix them. Both *iscale* and *inote* are arbitrary names to make the design a bit easier to read. Each is an i-rate variable, evaluated when the instrument is initialized.

```
          instr 4                            ; toot4.orc
 iamp     =         ampdb(p4)                ; convert decibels to linear amp
 iscale   =         iamp * .333              ; scale the amp at initialization
 inote    =         cpspch(p5)               ; convert "octave.pitch" to cps
 k1       linen     iscale, p6, p3, p7       ; p4=amp
 a3       oscil     k1, inote*.996, 1        ; p5=freq
 a2       oscil     k1, inote*1.004, 1       ; p6=attack time
 a1       oscil     k1, inote, 1             ; p7=release time
          out       a1
          endin

 f1 0 4096 10 1                              ; sine wave
;instr  start  duration  amp(p4)   freq(p5)  attack(p6) release(p7)
 i4       0     1          75        8.04     .1           .7
 i4       1     1          70        8.02     .07          .6
 i4       2     1          75        8.00     .05          .5
 i4       3     1          70        8.02     .05          .4
 i4       4     1          85        8.04     .1           .5
 i4       5     1          80        8.04     .05          .5
 i4       6     2          90        8.04     .03           1
e
```



Toot 4: multiple **oscil**s with value converters

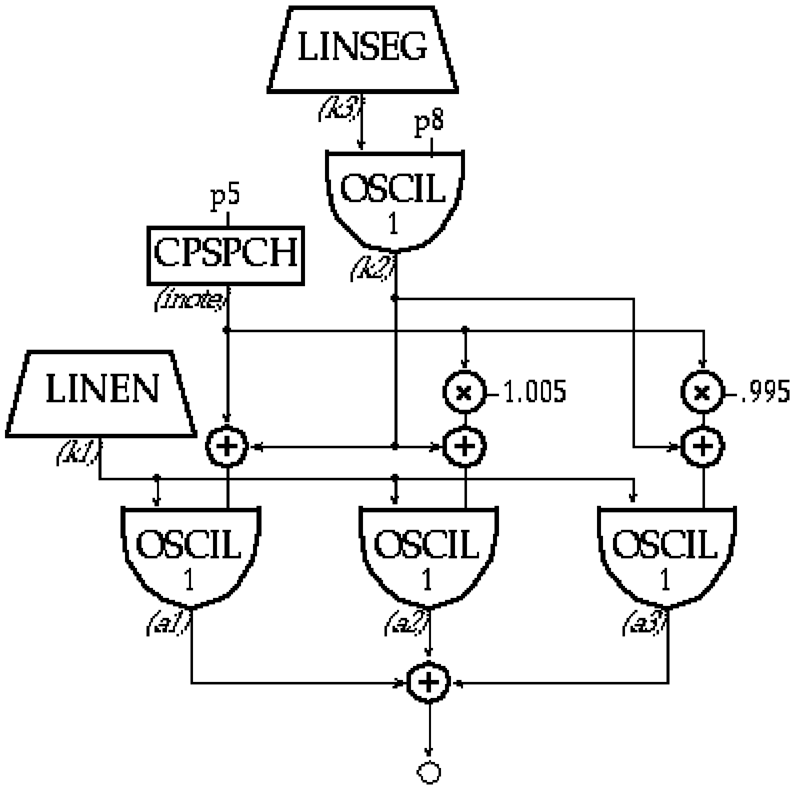
2.6 Toot 5: Vibrato

To add some delayed vibrato to our chorusing instrument we use another oscillator for the vibrato and a line segment generator, **linseg**, as a means of controlling the delay. **linseg** is a k-rate or a-rate signal generator which traces a series of straight line segments between any number of specified points. The Csound manual describes it as:

```
 kr      linseg      ia, idur1, ib[, idur2, ic[...]]
 ar      linseg      ia, idur1, ib[, idur2, ic[...]]
```

Since we intend to use this to slowly scale the amount of signal coming from our vibrato oscillator, we'll choose the k-rate version. The i-rate variables: *ia*, *ib*, i*c*, etc., are the values for the points. The i-rate variables: *idur1*, *idur2*, *idur3*, etc., set the duration, in seconds, between segments.

```
         instr 5                                       ; toot5.orc
 irel     =          .01                               ; set vibrato release
                                                       ;   time
 idel1    =          p3 - (p10 * p3)                   ; calculate initial
                                                       ;   delay (% of dur)
 isus     =          p3 - (idel1- irel)                ; calculate remaining
                                                       ;   duration
 iamp     =          ampdb(p4)                         ; p4=amp
 iscale   =          iamp * .333
 inote    =          cpspch(p5)                        ; p5=freq
 k3       linseg     0, idel1, p9, isus, p9, irel, 0   ; p6=attack time
 k2       oscil      k3, p8, 1                         ; p7=release time
 k1       linen      iscale, p6, p3, p7                ; p8=vib rate
 a3       oscil      k1, inote*.995+k2, 1              ; p9=vib depth
 a2       oscil      k1, inote*1.005+k2, 1             ; p10=vib delay (0-1)
 a1       oscil      k1, inote+k2, 1
          out        a1+a2+a3
          endin

                                                       ;toot5.sco
 f 1  0  4096  10  1
;ins strt dur  amp  frq      atk  rel  vibrt   vibdpth  vibdel
 i5  0    3    86   10.00     .1  .7   7        6       .4
 i5  4    3    86   10.02     1   .2   6        6       .4
 i5  8    4    86   10.04     2   1    5        6       .4
 e
```



Toot 5: Vibrato

2.7 Toot 6: Gens

The first character in a score statement is an **opcode**, determining an action request; the remaining data consists of numeric parameter fields (p-fields) to be used by that action. So far we have been dealing with two different opcodes in our score: f and i. i statements, or note statements, invoke the p1 instrument at time p2 and turn it off after p3 seconds; all remaining p-fields are passed to the instrument.

f statements, or lines with an opcode of f, invoke function-drawing subroutines called **GENS**. In Csound there are currently twenty-three GEN routines which fill wavetables in a variety of ways. For example, **GEN01** transfers data from a soundfile; **GEN07** allows you to construct functions from segments of straight lines; and **GEN10**, which we've been using in our scores so far, generates composite waveforms made up of a weighted sum of simple sinusoids. We have named the function "f1," invoked it at time 0, defined it to contain 512 points, and instructed **GEN10** to fill that wavetable with a single sinusoid whose amplitude is 1. **GEN10** can in fact be used to *approximate* a variety of other waveforms, as illustrated by the following:

```
f1 0  2048 10  1                                          ; Sine
f2 0  2048 10  1   .5   .3   .25  .2   .167   .14  .125 .111    ; Sawtooth
f3 0  2048 10  1   0    .3    0   .2    0     .14   0   .111    ; Square
f4 0  2048 10  1   1    1     1   .7    .5    .3    .1          ; Pulse
```

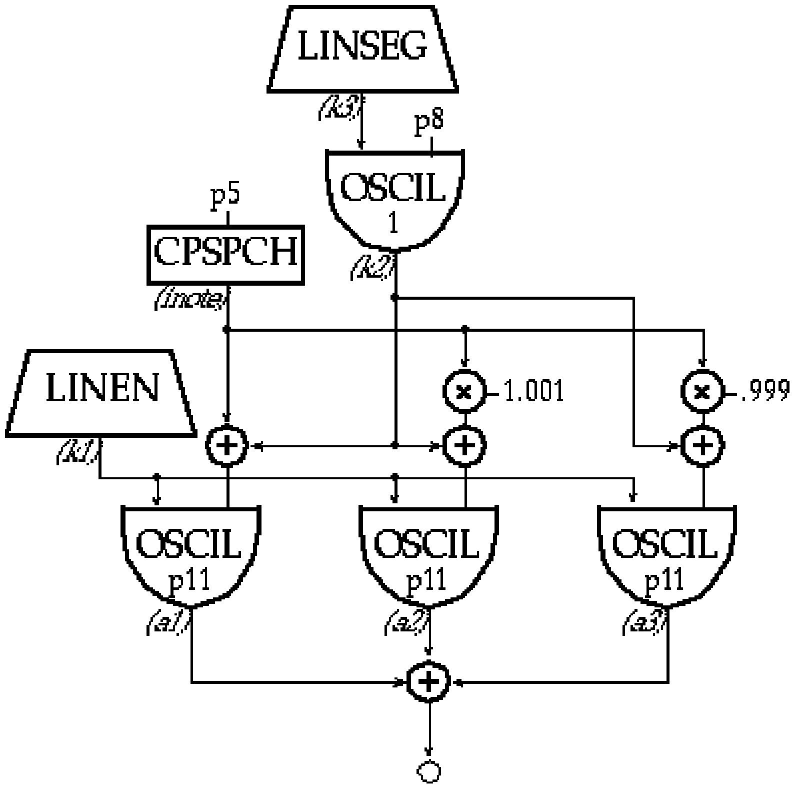
For the opcode f, the first four p-fields are interpreted as follows:

```
p1 - table number - In the orchestra, you reference this table by its
                    number.
p2 - creation time - The time at which the function is generated.
p3 - table size - Number of points in table - must be a power of 2,  or
                    that plus 1.
p4 - generating subroutine - Which of the 17 GENS will you employ.
p5 -> p?  -  meaning determined by the particular GEN subroutine.
```

In the instrument and score below, we have added three additional functions to the score, and modified the orchestra so that the instrument can call them via p11.

```
         instr 6                                     ; toot6.orc
ifunc    =          p11                              ; select basic
                                                     ;   waveform
irel     =          .01                              ; set vibrato release
idel1    =          p3 - (p10 * p3)                  ; calculate initial
                                                     ;   delay
isus     =          p3 - (idel1- irel)               ; calculate remaining
                                                     ;   dur
iamp     =          ampdb(p4)
iscale   =          iamp * .333                      ; p4=amp
inote    =          cpspch(p5)                       ; p5=freq
k3       linseg     0, idel1, p9, isus, p9, irel, 0  ; p6=attack time
k2       oscil      k3, p8, 1                        ; p7=release time
k1       linen      iscale, p6, p3, p7               ; p8=vib rate
a3       oscil      k1, inote*.999+k2, ifunc         ; p9=vib depth
a2       oscil      k1, inote*1.001+k2, ifunc        ; p10=vib delay (0-1)
a1       oscil      k1, inote+k2, ifunc
         out        a1 + a2 + a3
         endin
```

```
                                                                 ;toot6.sco
 f1 0 2048 10 1                                                  ; Sine
 f2 0 2048 10 1 .5 .3 .25 .2 .167 .14 .125 .111                  ; Sawtooth
 f3 0 2048 10 1 0  .3  0   .2  0  .14  0   .111                  ; Square
 f4 0 2048 10 1 1 1 1 .7 .5 .3 .1                                ; Pulse
;ins  strt  dur  amp  frq   atk  rel  vibrt vibdpth  vibdel  waveform(f)
 i6   0     2    86   8.00 .03   .7   6     9       .8       1
 i6   3     2    86   8.02 .03   .7   6     9       .8       2
 i6   6     2    86   8.04 .03   .7   6     9       .8       3
 i6   9     3    86   8.05 .03   .7   6     9       .8       4
 e
```



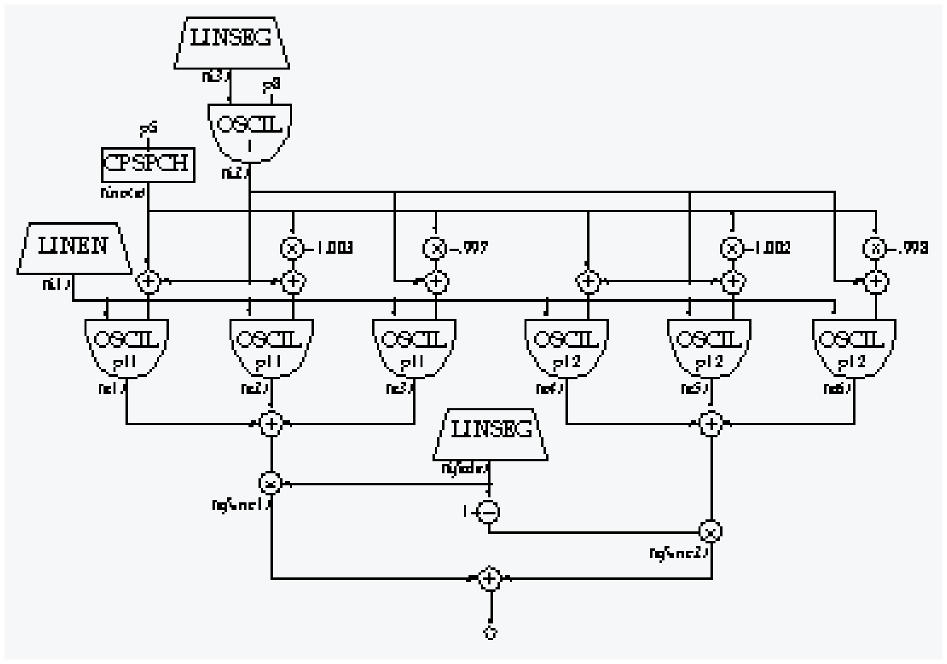
Toot 6: GENs

2.8 Toot 7: Crossfade

Now we will add the ability to do a linear crossfade between any two of our four basic waveforms. We will employ our delayed vibrato scheme to regulate the speed of the crossfade.

```
          instr 7                                       ; toot7.orc
 ifunc1   =         p11                                 ; initial waveform
 ifunc2   =         p12                                 ; crossfade waveform
 ifad1    =         p3 - (p13 * p3)                     ; calculate initial
                                                        ;   fade
 ifad2    =         p3 - ifad1                          ; calculate remaining
                                                        ;   dur
 irel     =         .01                                 ; set vibrato release
 idel1    =         p3 - (p10 * p3)                     ; calculate initial
                                                        ;   delay
 isus     =         p3 - (idel1- irel)                  ; calculate remaining
                                                        ;   dur
 iamp     =         ampdb(p4)
 iscale   =         iamp * .166                         ; p4=amp
 inote    =         cpspch(p5)                          ; p5=freq
 k3       linseg    0, idel1, p9, isus, p9, irel, 0     ; p6=attack time
 k2       oscil     k3, p8, 1                           ; p7=release time
 k1       linen     iscale, p6, p3, p7                  ; p8=vib rate
 a6       oscil     k1, inote*.998+k2, ifunc2           ; p9=vib depth
 a5       oscil     k1, inote*1.002+k2, ifunc2          ; p10=vib delay (0-1)
 a4       oscil     k1, inote+k2, ifunc2                ; p11=initial wave
 a3       oscil     k1, inote*.997+k2, ifunc1           ; p12=cross wave
 a2       oscil     k1, inote*1.003+k2, ifunc1          ; p13=fade time
 a1       oscil     k1, inote+k2, ifunc1
 kfade    linseg    1, ifad1, 0, ifad2, 1
 afunc1   =         kfade * (a1+a2+a3)
 afunc2   =         (1 - kfade) * (a4+a5+a6)
          out       afunc1 + afunc2
          endin
```

```
                                                        ; toot7.sco
 f1 0 2048 10 1                                         ; Sine
 f2 0 2048 10 1 .5 .3 .25 .2 .167 .14 .125 .111         ; Sawtooth
 f3 0 2048 10 1 0  .3  0   .2  0  .14  0   .111         ; Square
 f4 0 2048 10 1 1 1 1 .7 .5 .3 .1                       ; Pulse

;ins strt dur  amp  frq   atk rel vibrt vbdpt vibdel strtwav endwav crosstime
 i7   0   5    96  8.07  .03  .1   5      6    .99     1       2     .1
 i7   6   5    96  8.09  .03  .1   5      6    .99     1       3     .1
 i7  12   8    96  8.07  .03  .1   5      6    .99     1       4     .1
e
```



Toot 7: Crossfade

2.9 Toot 8: Soundin

Now instead of continuing to enhance the same instrument, let us design a totally different one. We'll read a soundfile into the orchestra, apply an amplitude envelope to it, and add some reverb. To do this we will employ Csound's **soundin** and **reverb** generators. The first is described as:

```
 a1       soundin     ifilcod[, iskiptime[, iformat]]
```

soundin derives its signal from a pre-existing file. *ifilcod* is either the filename in double quotes, or an integer suffix (.n) to the name "soundin". Thus the file `soundin.5` could be referenced either by the quoted name or by the integer 5. To read from 500ms into this file we might say:

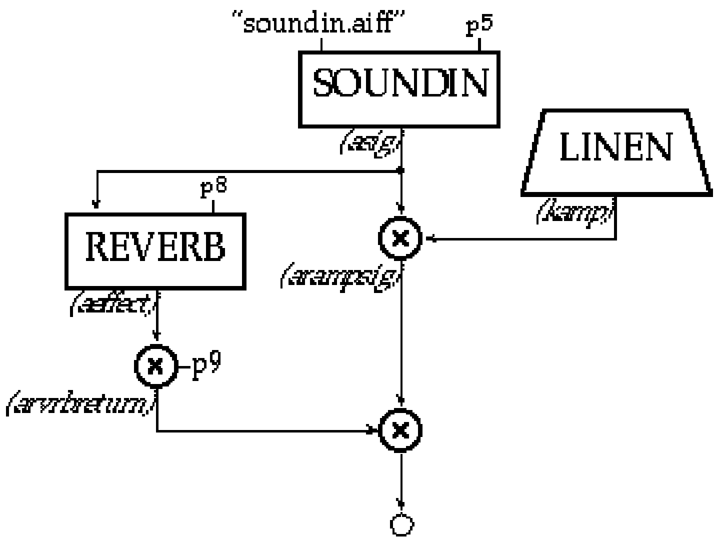
```
 a1       soundin     "soundin.5",  .5
```

The Csound **reverb** generator is actually composed of four parallel **comb** filters plus two **allpass** filters in series. Although we could design a variant of our own using these same primitives, the preset reverb is convenient, and simulates a natural room response via internal parameter values. Only two arguments are required the input (*asig*) and the reverb time (*krvt*)

```
 ar       reverb      asig, krvt
```

The soundfile instrument with artificial envelope and a reverb (included directly) is as follows:

```
             instr 8                                ; toot8.orc
 idur        =       p3
 iamp        =       p4
 iskiptime   =       p5
 iattack     =       p6
 irelease    =       p7
 irvbtime    =       p8
 irvbgain    =       p9
 kamp        linen   iamp, iattack, idur, irelease
 asig        soundin "soundin.aiff", iskiptime
 arampsig    =       kamp * asig
 aeffect     reverb  asig, irvbtime
 arvbreturn  =       aeffect * irvbgain
             out     arampsig + arvbreturn
             endin

                                                   ;toot8.sco
;ins  strt dur  amp   skip   atk  re  rvbtime rvbgain
 i8   0    1    .3    0     .03  .1   1.5    .2
 i8   2    1    .3    0     .1   .1   1.3    .2
 i8   3.5  2.25 .3    0     .5   .1   2.1    .2
 i8   4.5  2.25 .3    0     .01  .1   1.1    .2
 i8   5    2.25 .3   .1     .01  .1   1.1    .1
e
```



Toot 8: **soundin**

2.10 Toot 9: Global Stereo Reverb

In the previous example you may have noticed the soundin source being “cut off” at ends of notes, because the reverb was *inside* the instrument itself. It is better to create a companion instrument, a global reverb instrument, to which the source signal can be sent. Let's also make this stereo.

Variables are named cells which store numbers. In Csound, they can be either *local* or *global*, are available continuously, and can be updated at one of four rates - setup, i-rate, k-rate, or a-rate.

Local variables (which begin with the letters p, i, k, or a) are private to a particular instrument. They cannot be read from, or written to, by any other instrument.

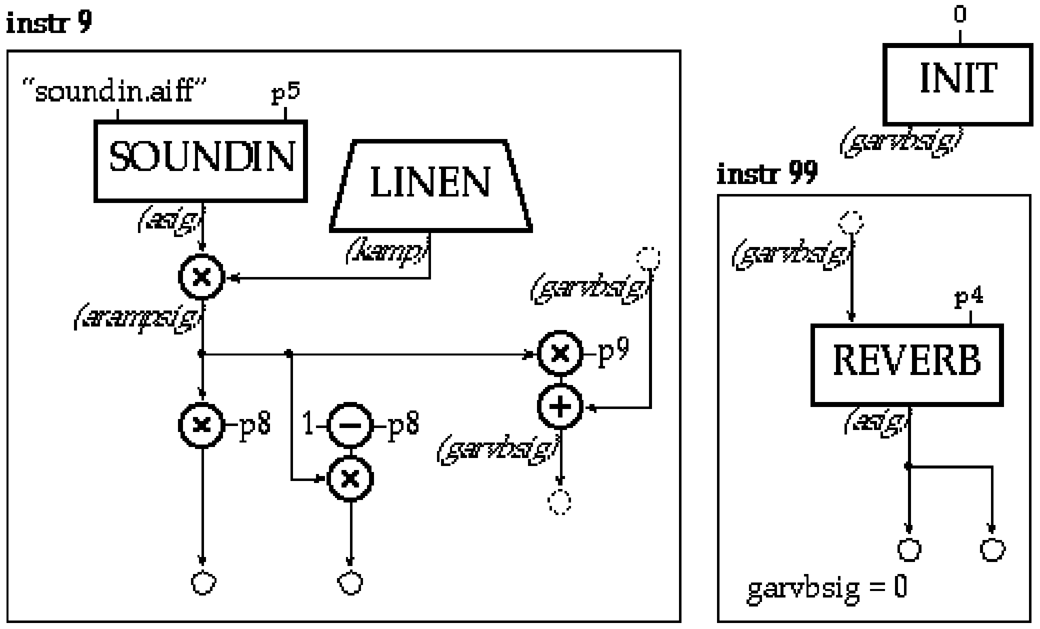
Global Variables are cells which are accessible by all instruments. Three of the same four variable types are supported (i, k, and a**)**, but these letters are preceded by the letter “g” to identify them as “global.” Global variables are used for “broadcasting” general values, for communicating between instruments, and for sending sound from one instrument to another.

The reverb instr99 below receives input from instr9 via the global a-rate variable *garvbsig*. Since instr9 *adds into* this global, several copies of instr9 can do this without losing any data. The addition requires *garvbsig* to be cleared before each k-rate pass through any active instruments. This is accomplished first with an **init** statement in the orchestra header, giving the reverb instrument a higher number than any other (instruments are performed in numerical order), and then clearing *garvbsig* within instr99 once its data has been placed into the reverb.

```
sr          =           44100                   ; toot9.orc
kr          =           4410
ksmps       =           10
nchnls      =           2                       ; stereo
garvbsig    init        0                       ; make zero at orch init time

            instr 9
  idur      =           p3
  iamp      =           p4
  iskiptime =           p5
  iattack   =           p6
  irelease  =           p7
  ibalance  =           p8                 ; panning: 1=left, .5=center, 0=right
  irvbgain  =           p9
  kamp      linen       iamp, iattack, idur, irelease
  asig      soundin     "soundin.aiff", iskiptime
  arampsig  =           kamp * asig
            outs        arampsig * ibalance,  arampsig * (1 - ibalance)
  garvbsig  =           garvbsig + arampsig * irvbgain
            endin

            instr 99                            ; global reverb
  irvbtime  =           p4
  sig       reverb      garvbsig,  irvbtime     ; put global signal into reverb
            outs        sig, asig
  garvbsig  =           0                       ; then clear it
            endin
```

In the score we turn the global reverb on at time 0 and keep it on until *irvbtime* after the last note.

```
; ins     strt dur  rvbtime                    ; toot9.sco
 i99     0    9.85 2.6

;ins strt  dur   amp  skip   atk  rel  balance(0-1) rvbsend
 i9  0     1     .5    0     .02  .1    1             .2
 i9  2     2     .5    0     .03  .1    0             .3
 i9  3.5   2.25  .5    0     .9   .1    .5            .1
 i9  4.5   2.25  .5    0     1.2  .1    0             .2
 i9  5     2.25  .5    0     .2   .1    1             .3
 e
```

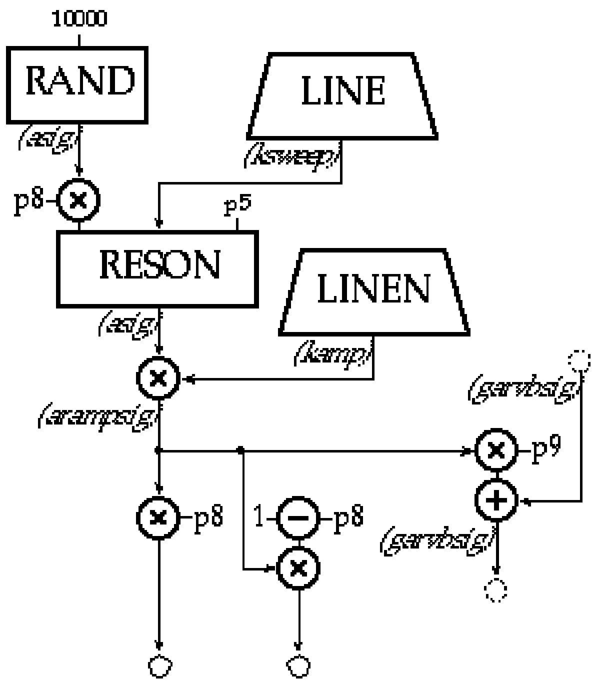


Toot 9: Global Stereo Reverb

2.11 Toot 10: Filtered Noise

The following instrument uses the Csound **rand** unit to produce noise, and a **reson** unit to filter it. The bandwidth of **reson** will be set at i-time, but its center frequency will be swept via a **line** unit through a wide range of frequencies during each note. We add reverb as in Toot 9.

```
garvbsig   init       0

           instr 10                         ; toot10.orc
 iattack   =          .01
 irelease  =          .2
 iwhite    =          10000
 idur      =          p3
 iamp      =          p4
 isweepstar =         p5
 isweepend =          p6
 ibandwidth =         p7
 ibalance  =          p8                    ; pan: 1 = left, .5 = center,
                                            ;    0 = right
 irvbgain  =          p9
 kamp      linen      iamp, iattack, idur, irelease
 ksweep    line       isweepstart, idur, isweepend
 asig      rand       iwhite
 afilt     reson      asig, ksweep, ibandwidth
 arampsig  =          kamp * afilt
           outs       arampsig * ibalance, arampsig * (1 - ibalance)
 garvbsig  =          garvbsig  +  arampsig * irvbgain
           endin

           instr 100
 irvbtime  =          p4
 asig      reverb     garvbsig,  irvbtime
           outs       asig, asig
 garvbsig  =          0
           endin

                                            ;toot10.sco
;ins   strt  dur  rvbtime
 i100    0    15   1.1
 i100    15   10   5

;ins   strt  dur   amp  stswp  ndswp  bndwth  balance(0-1)  rvbsend
 i10    0    2    .05  5000   500     20       .5            .1
 i10    3    1    .05  1500   5000    30       .5            .1
 i10    5    2    .05  850    1100    40       .5            .1
 i10    8    2    .05  1100   8000    50       .5            .1
 i10    8    .5   .05  5000   1000    30       .5            .2
 i10    9    .5   .05  1000   8000    40       .5            .1
 i10    11   .5   .05  500    2100    50       .4            .2
 i10    12   .5   .05  2100   1220    75       .6            .1
 i10    13   .5   .05  1700   3500    100      .5            .2
 i10    15   5    .01  8000   800     60       .5            .15
 e
```



Toot 10: Filtered Noise

2.12 Toot 11: Carry, Tempo & Sort

We now use a plucked string instrument to explore some of Csound's score preprocessing capabilities. Since the focus here is on the score, the instrument is presented without explanation.

```
        instr 11
asig1   pluck      ampdb(p4)/2, p5, p5, 0, 1
asig2   pluck      ampdb(p4)/2, p5 * 1.003,  p5 * 1.003, 0, 1
        out        asig1+asig2
        endin
```

The score can be divided into time-ordered sections by the **s** statement**.** Prior to performance, each section is processed by three routines: **Carry**, **Tempo**, and **Sort**. The score `toot11.sco` has multiple sections containing each of the examples below, in both of the forms listed.

2.12.1 CARRY

The carry feature allows a dot (".") in a p-field to indicate that the value is the same as above, provided the instrument is the same. Thus the following two examples are identical:

```
;ins start dur  amp  freq   |    ; ins start dur   amp  freq
 i11    0    1   90   200   |      i11    0    1    90   200
 i11    1    .    .   300   |      i11    1    1    90   300
 i11    2    .    .   400   |      i11    2    1    90   400
```

A special form of the carry feature applies to p2 only. A "+" in p2 will be given the value of p2+p3 from the previous **i** statement. The "+" can also be carried with a dot:

```
;ins start dur  amp  freq   |   ;  ins start  dur  amp  freq
 i11    0    1   90   200   |      i11    0    1    90   200
 i.     +    .    .   300   |      i11    1    1    90   300
 i.     .    .    .   500   |      i11    2    1    90   500
```

The carrying dot may be omitted when there are no more explicit pfields on that line:

```
;ins start dur  amp  freq   |   ;  ins start  dur  amp  freq
 i11    0    1   90   200   |      i11    0    1    90   200
 i11    +    2              |      i11    1    2    90   200
 i11                        |      i11    3    2    90   200
```

2.12.2 RAMPING

A variant of the carry feature is ramping, which substitutes a sequence of linearly interpolated values for a ramp symbol (<) spanning any two values of a pfield. Ramps work only on consecutive calls to the same instrument, and they cannot be applied to the first three p-fields.

```
;ins start dur  amp  freq   |    ; ins start  dur  amp  freq
 i11    0    1   90   200   |      i11     0    1   90   200
 i .    +    .   <    <     |      i11     1    1   85   300
 i .    .    .   <    400   |      i11     2    1   80   400
 i .    .    .   <    <     |      i11     3    1   75   300
 i .    .    4   70   200   |      i11     4    4   70   200
```

2.12.3 TEMPO

The unit of time in a Csound score is the beat - normally one beat per second. This can be modified by a tempo statement which enables the score to be arbitrarily time-warped. Beats are converted to their equivalent in seconds during score pre-processing of each Section. In the absence of a Tempo statement in any Section, the following tempo statement is inserted:

```
 t   0   60
```

It means that at beat 0 the tempo of the Csound beat is 60 (1 beat per second). To hear the Section at twice the speed, we have two options: 1) cut all p2 and p3 in half and adjust the start times, or 2) insert the statement `t 0 120` within the Section.

The tempo statement can also be used to move between different tempi during the score, thus enabling ritardandi and accelerandi. Changes are linear by beat size. The following statement will cause the score to begin at tempo 120, slow to tempo 80 by beat 4, then accelerate to 220 by beat 7:

```
 t    0   120   4   80    7   220
```

The following will produce identical sound files:

```
t  0  120                                    ; Double-time via Tempo
;ins  start  dur  amp  freq       |          ; ins start  dur  amp  freq
 i11     0    .5   90   200       |            i11     0    1   90   200
 i .     +    .    <    <         |            i .     +    .   <    <
 i .     .    .    <    400       |            i .     .    .   <    400
 i .     .    .    <    <         |            i .     .    .   <    <
 i .     .    2    70   200       |            i .     .    4   70   200
```

The following includes an accelerando and ritard. It should be noted, however, that the ramping feature is applied *after* time-warping, and is thus proportional to elapsed chronological time. While this is perfect for amplitude ramps, frequency ramps will not result in harmonically related pitches during tempo changes. The frequencies needed here are thus made explicit.

```
      t     0   60   4    400  8    60      ; Time-warping via Tempo
     ;    ins start dur  amp  freq
          i11  0    1    70   200
          i .  +    .    <    500
          i .  .    .    90   800
          i .  .    .    <    500
          i .  .    .    70   200
          i .  .    .    90   1000
          i .  .    .    <    600
          i .  .    .    70   200
          i .  .    8    90   100
```

2.12.4 SCORE SECTIONS

Three additional score features are extremely useful in Csound. The **s** statement was used above to divide a score into Sections for individual pre-processing. Since each s statement establishes a new relative time of 0, and all actions within a section are relative to that, it is convenient to develop the score one section at a time, then link the sections into a whole later.

Suppose we wish to combine the six above examples (call them `toot11a - toot11f`) into one score. One way is to start with `toot11a.sco`, calculate its total duration and add that value to every starting time of `toot11b.sco`, then add the composite duration to the start times of `toot11c.sco`, etc. Alternatively, we could insert an **s** statement between each of the sections and run the entire score. The file `toot11.sco`, which contains a sequence of all of the above score examples, did just that.

2.12.5 ADDING EXTRA TIME

The **f0** statement, which creates an "action time" with no associated action, is useful in extending the duration of a section. Two seconds of silence are added to the first two sections below.

```
;    ins start dur  amp  freq          ; toot11g.sco
     i11  0    2    90   100
     f 0  4                            ; The f0 Statement
     s                                 ; The Section Statement
     i11  0    1    90   800
     i .  +    .    .    400
     i .  .    .    .    100
     f 0  5
     s
     i11  0         4    90   50
     e
```

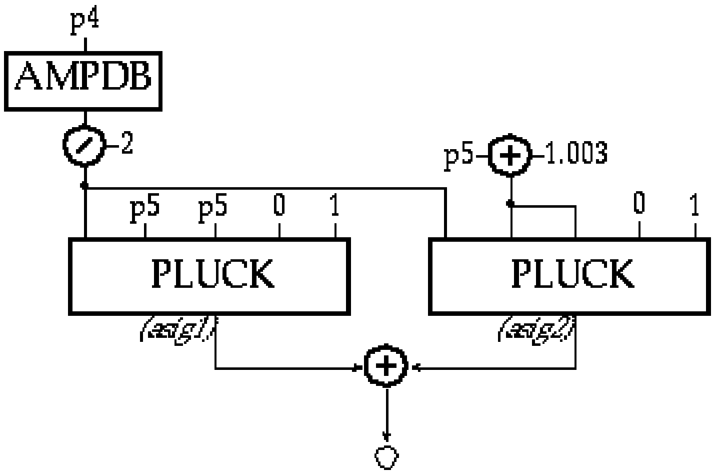
2.12.6 SORT

During preprocessing of a score section, all action-time statements are sorted into chronological order by p2 value. This means that notes can be entered in any order, that you can merge files, or work on instruments as temporarily separate sections, then have them sorted automatically when you run Csound on the file.

The file below contains excerpts from this section of the rehearsal chapter and from instr6 of the tutorial, and combines them as follows:

```
;   ins start dur  amp  freq          ; toot11h.sco
    i11  0    1    70   100           ; Score Sorting
    i .  +    .    <    <
    i .  .    .    <    <
    i .  .    .    90   800
    i .  .    .    <    <
    i .  .    .    <    <
    i .  .    .    70   100
    i .  .    .    90   1000
    i .  .    .    <    <
    i .  .    .    <    <
    i .  .    .    <    <
    i .  .    .    70   <
    i .  .    8    90   50
```

```
   f1 0 2048 10 1                                      ; Sine
   f2 0 2048 10 1 .5 .3 .25 .2 .167 .14 .125 .111      ; Sawtooth
   f3 0 2048 10 1 0  .3  0   .2  0 .14  0   .111       ; Square
   f4 0 2048 10 1 1 1 1 .7 .5 .3 .1                    ; Pulse

; ins  strt  dur   amp   frq  atk  rel  vibr  vibdpth  vibdel    waveform
   i6     0    2    86   9.00 .03  .1   6    5          .4       1
   i6     2    2    86   9.02 .03  .1   6    5          .4       2
   i6     4    2    86   9.04 .03  .1   6    5          .4       3
   i6     6    4    86   9.05 .05  .1   6    5          .4       4
```



Toot 11: Carry, Tempo, and Sort

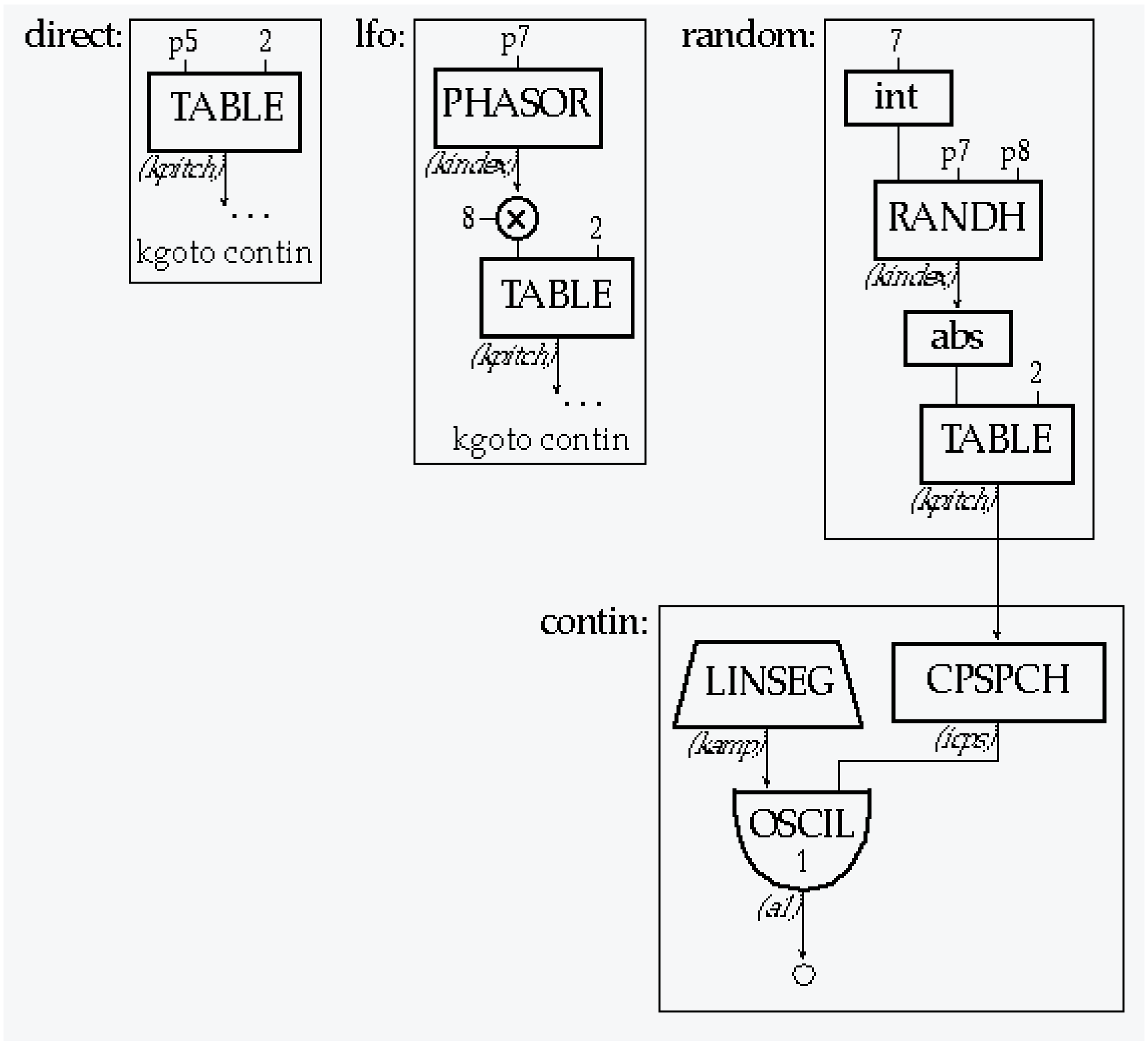
2.13 Toot 12: Tables and Labels

This is by far our most complex instrument. In it we have designed the ability to store pitches in a table, and then index them in three different ways: 1) directly, 2) via an lfo, and 3) randomly. As a means of switching between these three methods, we will use Csound's *program control* statements and logical and conditional operations.

```
                    instr 12                          ;toot12.orc
           iseed    =    p8
           iamp     =    ampdb(p4)
           kdirect  =    p5
           imeth    =    p6
           ilforate =    p7                           ; lfo and random
                                                      ;   index rate
           itab     =    2
           itablesize =  8

 if (imeth == 1)    igoto direct
 if (imeth == 2)    kgoto lfo
 if (imeth == 3)    kgoto random

direct:   kpitch    table   kdirect, itab             ; index "f2" via p5
                    kgoto   contin

lfo:      kindex    phasor  ilforate
          kpitch    table   kindex  *  itablesize, itab
                    kgoto   contin

random:   kindex    randh   int(7), ilforate, iseed
          kpitch    table   abs(kindex), itab

contin:   kamp      linseg  0, p3 * .1, iamp, p3 * .9, 0  ; amp envelope
          asig      oscil   kamp, cpspch(kpitch), 1       ; audio osc
                    out     asig
                    endin

                                                      ;toot12.sco
f1 0 2048 10 1                                        ; sine
f2 0 8 -2  8.00 8.02 8.04 8.05 8.07 8.09 8.11 9.00     ; cpspch C major scale

; method 1 - direct index of table values
; ins start   dur  amp index method  lforate   rndseed
  i12     0    .5   86   7    1        0        0
  i12    .5    .5   86   6    1        0
  i12     1    .5   86   5    1        0
  i12     1.5  .5   86   4    1        0
  i12     2    .5   86   3    1        0
  i12     2.5  .5   86   2    1        0
  i12     3    .5   86   1    1        0
  i12     3.5  .5   86   0    1        0
  i12     4    .5   86   0    1        0
  i12     4.5  .5   86   2    1        0
  i12     5    .5   86   4    1        0
  i12     5.5  2.5  86   7    1        0
s
```

```
; method 2 - lfo index of table values
; ins  start dur  amp index method lforate   rndseed
  i12     0    2    86   0    2      1         0
  i12     3    2    86   0    2      2
  i12     6    2    86   0    2      4
  i12     9    2    86   0    2      8
  i12     12   2    86   0    2      16
s

; method 3 - random index of table values
; ins  start dur  amp  index method  rndrate  rndseed
  i12     0    2    86   0    3      2       .1
  i12     3    2    86   0    3      3       .2
  i12     6    2    86   0    3      4       .3
  i12     9    2    86   0    3      7       .4
  i12     12   2    86   0    3      11      .5
  i12     15   2    86   0    3      18      .6
  i12     18   2    86   0    3      29      .7
  i12     21   2    86   0    3      47      .8
  i12     24   2    86   0    3      76      .9
  i12     27   2    86   0    3      123     .9
  i12     30   5    86   0    3      199     .1
e
```

```
if (imeth == 1) igoto direct
if (imeth == 2) igoto lfo
if (imeth == 3) igoto random
```



Toot 12: Tables and Labels

2.14 Toot 13: Spectral Fusion

For our final instrument, we will employ three unique synthesis methods: Physical Modeling, Formant-Wave Synthesis, and Non-linear Distortion. Three of Csound's most powerful unit generators - **pluck**, **fof**, and **foscil**, make this complex task a fairly simple one. The Reference Manual describes these as follows:

```
ar       pluck       kamp, kcps, icps, ifn, imeth [, iparm1, iparm2]
```

pluck simulates the sound of naturally decaying plucked strings by filling a cyclic decay buffer with noise and then smoothing it over time according to one of several methods. The unit is based on the Karplus-Strong algorithm.

```
ar       fof         xamp, xfund, xform, koct, kband, kris, kdur kdec,\\
                     iolaps, ifna, ifnb, itotdur[, iphs[, ifmode]]
```

fof simulates the sound of the male voice by producing a set of harmonically related partials (a formant region) whose spectral envelope can be controlled over time. It is a special form of granular synthesis, based on the CHANT program from IRCAM by Xavier Rodet et al.

```
ar       foscil      xamp, kcps, kcar, kmod, kndx, ifn [, iphs]
```

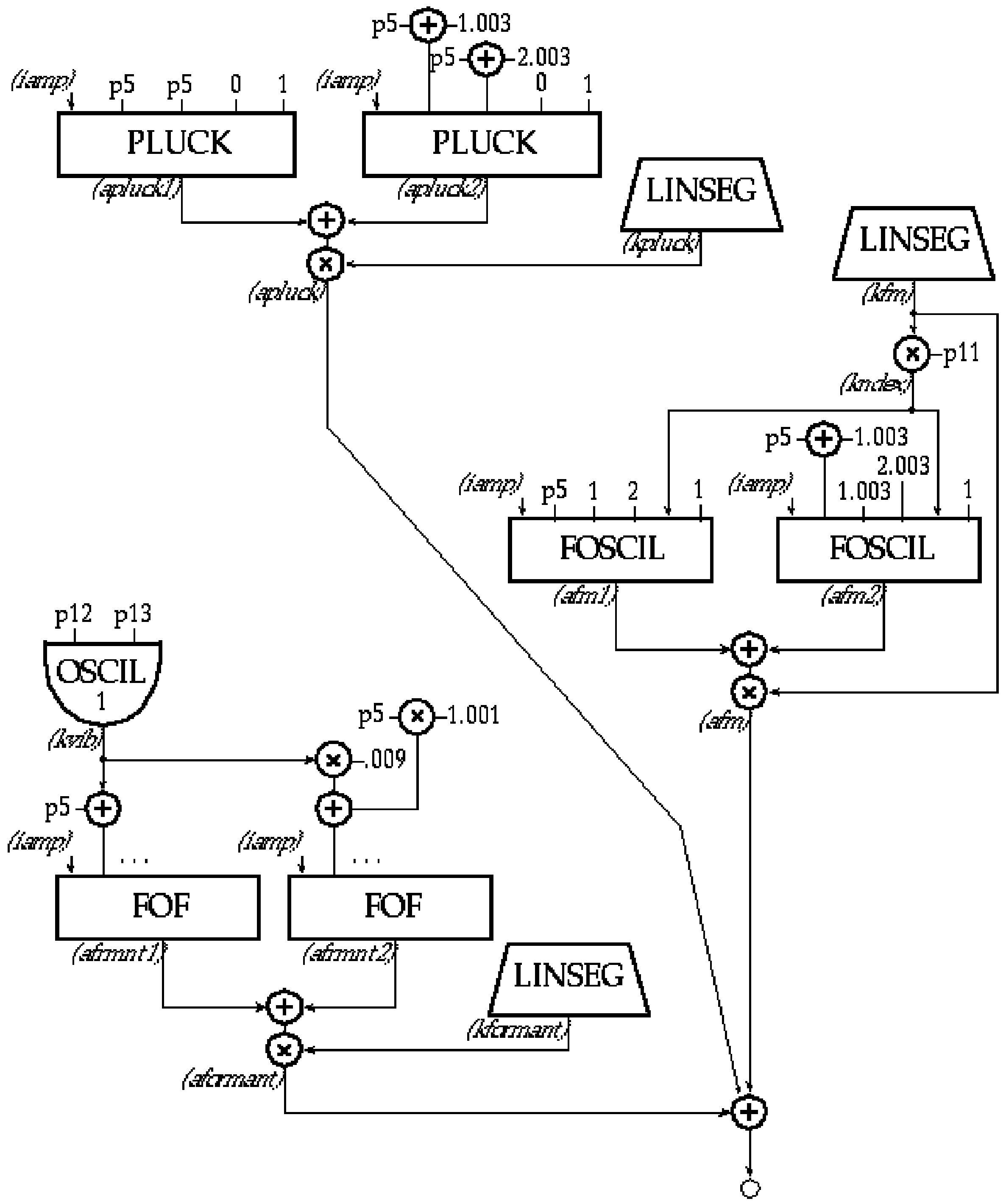
foscil is a composite unit which banks two oscillators in a simple FM configuration, wherein the audio-rate output of one (the "modulator") is used to modulate the frequency input of another (the "carrier.")

The plan for our instrument is to have the plucked string attack dissolve into an FM sustain which transforms into a vocal release. The orchestra and score are as follows:

```
             instr 13              ; toot13.orc
iamp         =      ampdb(p4) / 2  ; amplitude, scaled for two sources
ipluckamp    =      p6             ; % of total amp, 1=dB amp as in p4
ipluckdur    =      p7*p3          ; % of total dur, 1=entire dur of note
ipluckoff    =      p3 - ipluckdur
ifmamp       =      p8             ; % of total amp, 1=dB amp as in p4
ifmrise      =      p9*p3          ; % of total dur, 1=entire dur of note
ifmdec       =      p10*p3         ; % of total duration
ifmoff       =      p3 - (ifmrise + ifmdec)
index        =      p11
ivibdepth    =      p12
ivibrate     =      p13
iformantamp  =      p14            ; % of total amp, 1=dB amp as in p4
iformantrise =      p15*p3         ; % of total dur, 1=entire dur of note
iformantdec  =      p3 - iformantrise
```

```
kpluck   linseg     ipluckamp, ipluckdur, 0, ipluckoff, 0
apluck1  pluck      iamp, p5, p5, 0, 1
apluck2  pluck      iamp, p5*1.003, p5*1.003, 0, 1
apluck   =          kpluck * (apluck1+apluck2)

kfm      linseg     0, ifmrise, ifmamp, ifmdec, 0, ifmoff, 0
kndx     =          kfm * index
afm1     foscil     iamp, p5, 1, 2, kndx, 1
afm2     foscil     iamp, p5*1.003, 1.003, 2.003, kndx, 1
afm      =          kfm * (afm1+afm2)

kfrmnt   linseg     0, iformantrise, iformantamp, iformantdec, 0
kvib     oscil      ivibdepth, ivibrate, 1
afrmnt1  fof        iamp, p5+kvib, 650, 0, 40, .003, .017, .007, 4, 1,\\ 2,
                    p3
afrmnt2  fof        iamp, (p5*1.001)+kvib*.009, 650, 0, 40, .003, .017,\\
                    .007, 10, 1, 2, p3
aformnt  =          kfrmnt * (afrmnt1+afrmnt2)

         out        apluck + afm + aformnt
         endin
```

```
                                              ; toot13.sco
f1  0  8192   10   1                          ; sine wave
f2  0  2048   19  .5  1  270  1               ; sigmoid rise

;i  st  dr  mp   frq  plkmp plkdr fmp fmris fmdec indx vbdp vbrt  frmp fris
i13 0    5  80   200   .8    .3   .7   .2   .35    8    1    5     3    .5
i13 +    8  80   100   .     .4   .7   .35  .35    7    1    6     3    .7
i13 .   13  80   50    .     .3   .7   .2   .4     6    1    4     3    .6
```



Toot 13: Spectral Fusion

2.15 When Things Sound Wrong

When you design your own Csound instruments you may occasionally be surprised by the results. There will be times when you've computed a file for hours and your playback is just silence, while at other times you may get error messages which prevent the score from running, or you may hang the computer and nothing happens at all.

In general, Csound has a comprehensive error-checking facility that reports to your console at various stages of your run: at score sorting, orchestra translation, initializing each call of every instrument, and during performance. However, if your error was syntactically permissable, or it generated only a warning message, Csound could faithfully give you results you don't expect. Here is a list of the things you might check in your score and orchestra files:

1. You typed the letter ‘l’ instead of the number ‘1.’

2. You forgot to precede your comment with a semi-colon.

3. You forgot an opcode or a required parameter.

4. Your amplitudes are not loud enough, or they are too loud.

5. Your frequencies are not in the audio range - 20Hz to 20kHz.

6. You placed the value of one parameter in the p-field of another.

7. You left out some crucial information like a function definition.

8. You didn't meet the GEN specifications.

2.16 Suggestions for Further Study

Csound is such a powerful tool that we have touched on only a few of its many features and uses. You are encouraged to take apart the instruments in the tutorials, rebuild them, modify them, and integrate the features of one into the design of another. To understand their capabilities you should compose short etudes with each. You may be surprised to find yourself merging these little studies into the fabric of your first Csound compositions.

There are many sources of information on Csound and software synthesis. The ultimate sourcebook for Csound is *The Csound Book: Perspectives in Software Synthesis, Sound Design, Signal Processing, and Programming*, edited by Richard Boulanger, and published by MIT Press.

Nothing will increase your understanding more than actually making music with Csound. The best way to discover the full capability of these tools is to create your own music with them. As you negotiate the new and uncharted terrain you will make many discoveries. It is my hope that through Csound you discover as much about music as I have, and that this experience brings you great personal satisfaction and joy.

Richard Boulanger
Boston, Massachusetts USA
March, 1991

3 A FOF SYNTHESIS TUTORIAL

by J. M. Clarke, University of Huddersfield

The **fof** synthesis generator in Csound has more parameter fields than other modules. To help the user become familiar with these parameters this tutorial will take a simple orchestra file using just one **fof** unit-generator and demonstrate the effect of each parameter in turn. To produce a good vocal imitation, or a sound of similar sophistication, an orchestra containing five or more **fof** generators is required and other refinements (use of random variation of pitch etc.) must be made. The sounds produced in these initial explorations will be much simpler and consequently less interesting but they will help to show clearly the basic elements of **fof** synthesis. This tutorial assumes a basic working knowledge of Csound itself. The specification of the **fof** unit-generator (as found in the reference section of this manual) is:

```
 ar      fof        xamp, xfund, xform, koct, kband, kris, kdur,\\ kdec,
                    iolaps, ifna, ifnb, itotdur, [iphs,\\ [ifmode]]
```

where:

xamp, xfund, xform – can receive any rate (constant, control or audio)

koct, kband, kdris, kdur, kdec – can receive only constants or control rates

iolaps, ifna, ifnb, itotdur – must be given a fixed value at initialization

iphs, ifmode – are optional, defaulting to 0.

The following orchestra contains a simple instrument we will use for exploring each parameter in turn. On the faster machines (DECstation, SparcStation, SGI Indigo) it will run in real time.

```
sr        =        44100
kr        =        4410
ksmps     =        10

instr 1
 a1       fof      15000, 200, 650, 0, 0, .003, .02, .007, 5, 1, 2, p3
          out      a1
endin
```

It should be run with the following score:

```
f1  0  4096  10  1
f2  0  1024  19  .5  .5  270  .5
i1  0  3
e
```

The result is very basic. This is not surprising since we have created only one formant region (a vocal imitation would need at least five) and have no vibrato or random variation of the parameters. By varying one parameter at a time we will help the reader learn how the unit-generator works. Each of the following “variations” starts from the model. Parameters not specified remain as given.

xamp – amplitude

The first input parameter controls the amplitude of the generator. At present our model uses a constant amplitude, this can be changed so that the amplitude varies according to a line function:

```
a2       linseg   0,  p3*.3,  20000,  p3*.4,  15000,  p3*.3,  0
a1       fof      a2, ......(as before)...
```

The amplitude of a **fof** generator needs care. *xamp* does not necessarily indicate the maximum output, which can also depend on the rise pattern, bandwidth, and the presence of any “overlaps.”

xfund – fundamental frequency

This parameter controls the pitch of the fundamental of the unit generator. Starting again from the original model this example demonstrates an exaggerated vibrato:

```
a2       oscil    20,  5,  1
a1       fof      15000,  200+a2,  etc........
```

fof synthesis produces a rapid succession of (normally) overlapping excitations or granules. The fundamental is in fact the speed at which new excitations are formed and if the fundamental is very low, these excitations are heard as separate granules. In this case the fundamental is not so much a pitch as a pulse speed. The possibility of moving between pitch and pulse, between timbre and granular texture is one of the most interesting aspects of **fof**. For a simple demonstration try the following variation. It will be especially clear if the score note is lengthened to about 10 seconds.

```
a2       expseg   5,  p3*.8,  200,  p3*.2,  150
a1       fof      15000,  a2  etc........
```

koct – octaviation coefficient

Skipping a parameter, we come to an unusual means of controlling the fundamental: *octaviation*. This parameter is normally set to 0. For each unit increase in *koct* the fundamental pitch will drop by one octave. The change of pitch is **not** by the normal means of glissando, but by gradually fading out alternate excitations (leaving half the original number). Try the following (again with the longer note duration):

```
k1       linseg   0, p3*.1, 0, p3*.8, 6, p3*.1, 6
a1       fof      15000, 200, 650, k1, etc........
```

This produces a drop of six octaves; if the note is sufficiently long you should be able to hear the fading out of alternate excitations towards the end.

xform – formant frequency

ifmode – formant mode (0 = striated, non-0 = smooth)

The spectral output of a **fof** unit-generator resembles that of an impulse generator filtered by a band pass filter. It is a set of partials above a fundamental *xfund* with a spectral peak at the formant frequency *xform*. Motion of the formant can be implemented in two ways. If *ifmode* = 0, data sent to *xform* has effect only at the start of a new excitation. That is, each excitation gets the current value of this parameter at the time of creation and holds it until the excitation ends. Successive overlapping excitations can have different formant frequencies, creating a richly varied sound. This is the mode of the original CHANT program. If *ifmode* is non-zero, the frequency of each excitation varies continuously with *xform*. This allows glissandi of the formant frequency. To demonstrate these differences we take a very low fundamental, so that the granules can be heard separately and the formant frequency is audible not as the center frequency of a "band" but as a pitch in its own right. Compare the following in which only *ifmode* is changed:

```
a2       line     400,  p3,  800
a1       fof      15000, 5, a2, 0, 1, .003, .5, .1, 3, 1, 2, p3, 0, 0

a2       line     400,  p3,  800
a1       fof      15000, 5, a2, 0, 1, .003, .5, .1, 3, 1, 2, p3, 0, 1
```

In the first case, the formant frequency moves by step at the start of each excitation, whereas in the second it changes smoothly. A more subtle difference is perceived with higher fundamental frequencies. (Note that the later **fof** parameters were changed in this example to lengthen the excitations so that their pitch could be heard easily.)

xform also permits frequency modulation of the formant frequency. Applying FM to an already complex sound can lead to strange results, but here is a simple example:

```
acarr    line     400, p3, 800
index    =        2.0
imodfr   =        400
idev     =        index * imodfr
amodsig  oscil    idev, imodfr, 1
a1       fof      15000, 5, acarr+amodsig, 0, 1, .003, .5, .1, 3, 1,  2,
                  p3,  0,  1
```

kband – formant bandwidth

kris, kdur, kdec – rise time, duration and decay time (in seconds) of the excitation envelope

These parameters control the shape and length of the **fof** granules. They are shaped in three segments: a rise, a middle decay, and a terminating decay. For very low fundamentals, these are perceived as an amplitude envelope, but with higher fundamentals (above 30 Hz), the granules merge together and these parameters effect the timbre of the sound. Note that these four parameters influence a new granule only at the time of its initialization and are fixed for its duration; later changes will affect only subsequent granules. We begin our examination with low frequencies.

```
k1       line     .003, p3, .1                          ; kris
a1       fof      15000, 2, 300, 0, 0, k1, .5, .1, 2, 1, 2, p3
```

Run this with a note length of 10 seconds. Notice how the attack of the envelope of the granules lengthens. The shape of this attack is determined by the forward shape of *ifnb* (here a sigmoid).

Now try changing *kband*:

```
k1       linseg   0, p3, 10                             ; kband
a1       fof      15000, 2, 300, 0, k1, .003, .5, .1, 2, 1, 2, p3
```

Following its rise, an excitation has a built-in exponential decay and *kband* determines its rate. The bigger *kband* the steeper the decay; zero means no decay. In the above example, the successive granules had increasingly fast decays.

```
k1       linseg   3, p3, .003
a1       fof      15000, 2, 300, 0, 0, .003, .4, k1, 2, 1, 2, p3
```

This demonstrates the operation of *kdec*. Because an exponential decay never reaches zero, it must be terminated gracefully. *kdur* is the overall duration (in seconds from the start of the excitation), and *kdec* is the length of the terminating decay. In the above example, the terminating decay starts very early in the first granules and then becomes progressively later. Note that *kband* is set to zero so that only the terminating decay is evident.

In the next example, the start time of the termination remains constant, but its length gets shorter:

```
k1       expon    .3, p3, .003
a1       fof      15000, 2, 300, 0, 0, .003, .01 + k1, k1, 2, 1, , p3
```

It may be surprising to find that, for higher fundamentals, the local envelope determines the spectral shape of the sound. Electronic and computer music has often shown how features of music we normally consider independent, such as pitch, timbre, rhythm, are, in fact, different aspects of the same thing. In general, the longer the local envelope segment, the narrower the band of partials around that frequency. *kband* determines the bandwidth of the formant region at -6dB, and *kris* controls the skirt width at -40dB. Increasing *kband* increases the local envelope's exponential decay rate, thus shortening it and increasing the –6 dB spectral region. Increasing *kris* (the envelope attack time)inversely makes the –40 dB spectral region smaller.

The next example changes first the bandwidth, then the skirt width. You should be able to hear the difference.

```
k1      linseg   100, p3/4, 0, p3/4, 100, p3/2, 100        ; kband
k2      linseg   .003, p3/2, .003, p3/4, .01, p3/4, .003   ; kris
a1      fof      15000, 100, 440, 0, k1, k2, .02, .007, 3, 1, 2, p3
```

In the first half of the note, *kris* remains constant while *kband* broadens, then narrows again. In the second half, *kband* is fixed while *kris* lengthens (narrowing the spectrum), then returns again.

Note that *kdur* and *kdec* don't really shape the spectrum, they simply tidy up the decay so as to prevent unwanted discontinuities which would distort the sound. For vocal imitations these parameters are typically set at .017 and .007 and left unchanged. With high ("soprano") fundamentals it is possible to shorten these values and save computation time (reduce overlaps).

iolaps – number of overlap spaces

Granules are created at the rate of the fundamental frequency, and new granules are often created before earlier ones have finished, resulting in overlaps. The number of overlaps at any one time is given by *xfund* * *kdur*. For a typical bass note the calculation might be 200 * .018 = 3.6, and for a soprano note 660 * .015 = 9.9. **fof** needs at least this number (rounded up) of spaces in which to operate. The number can be over-estimated at no computation cost, and at only a small space cost. If there are insufficient overlap spaces during operation, the note will terminate.

ifna, *ifnb* – stored function tables

Identification numbers of two function tables (see the **fof** entry).

itotdur – total duration within which all granules in a note must be completed

So that incomplete granules are not cut off at the end of a note **fof** will not create new granules if they will not be completed by the time specified. Normally, given the value p3 (the note length), this parameter can be changed for special effect; **fof** will output zero after time *itotdur*.

iphs – initial phase (optional, defaulting to 0).

Specifies the initial phase of the fundamental. Normally zero, but giving different **fof** generators different initial phases can be helpful in avoiding “zeros” in the spectrum.