AN INSTRUMENT DESIGN TOOTORIAL

by Dr. Richard Boulanger, Berklee College of Music

1.1 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.

1.1.1 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.

1.1.2 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
```

1.1.3 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".

1.1.4 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.

1.2 Toot 1: Play One Note

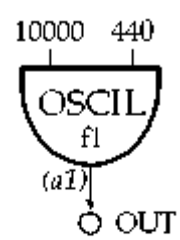
For this and all instrument examples, there exist `orchestra` and `score` files in the Csound subdirectory `tutorfiles` that the user can run to soundtest each feature introduced. The instrument code shown below is actually preceded by an *orchestra header section* similar to that shown above. If you 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**

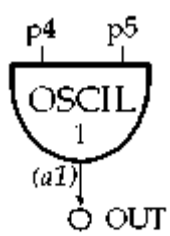
1.3 Toot 2: "P-Fields"

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

1.4 Toot 3: Envelopes

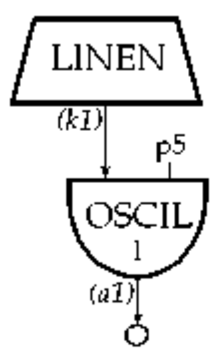
Although in the second instrument we could control and vary the overall amplitude from note to note, it would be more musical if we could contour the loudness during the course of each note. To do this we'll need to employ an additional unit generator **linen**, which the Csound reference manual defines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



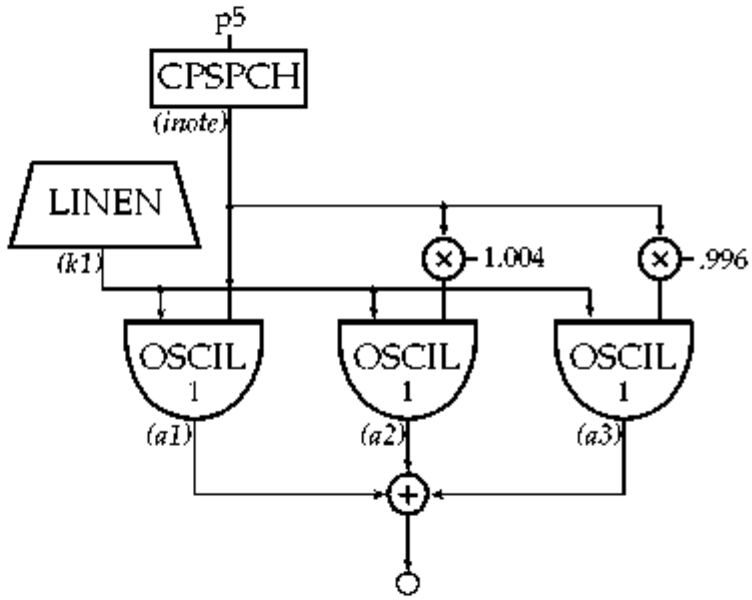
Toot 3: **linen** applied to **oscil**

1.5 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
  a1       =          a1 + a2 + a3
           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

1.6 Toot 5: Vibrato

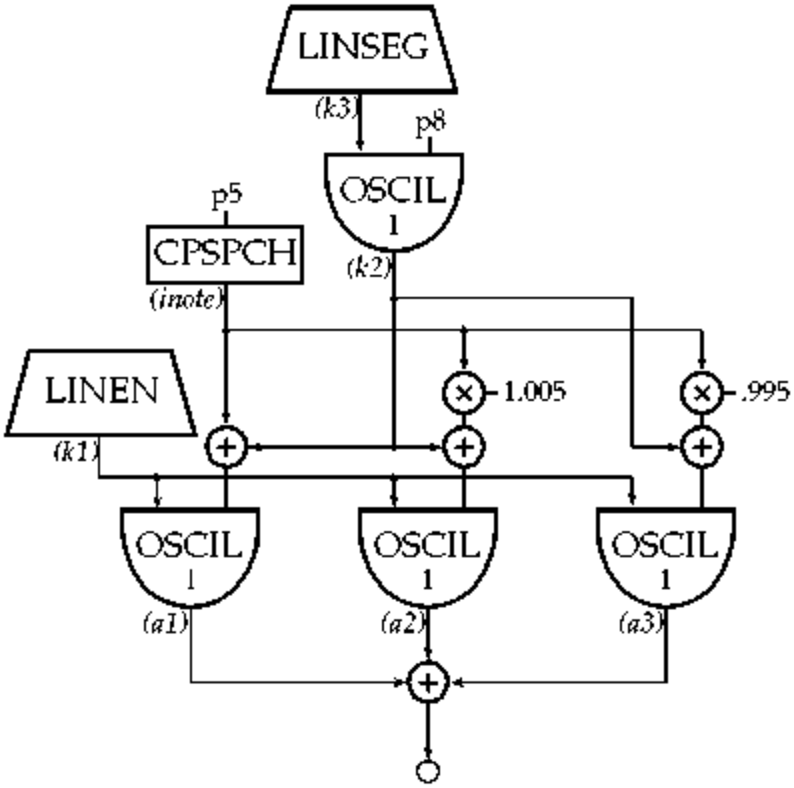
To add some delayed vibrato to our chorusing instrument we use another oscillator for the vibrato and a line segment generator, **linseg**, as a means of controlling the delay. **linseg** is a k-rate or a-rate signal generator which traces a series of straight line segments between any number of specified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

1.7 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:

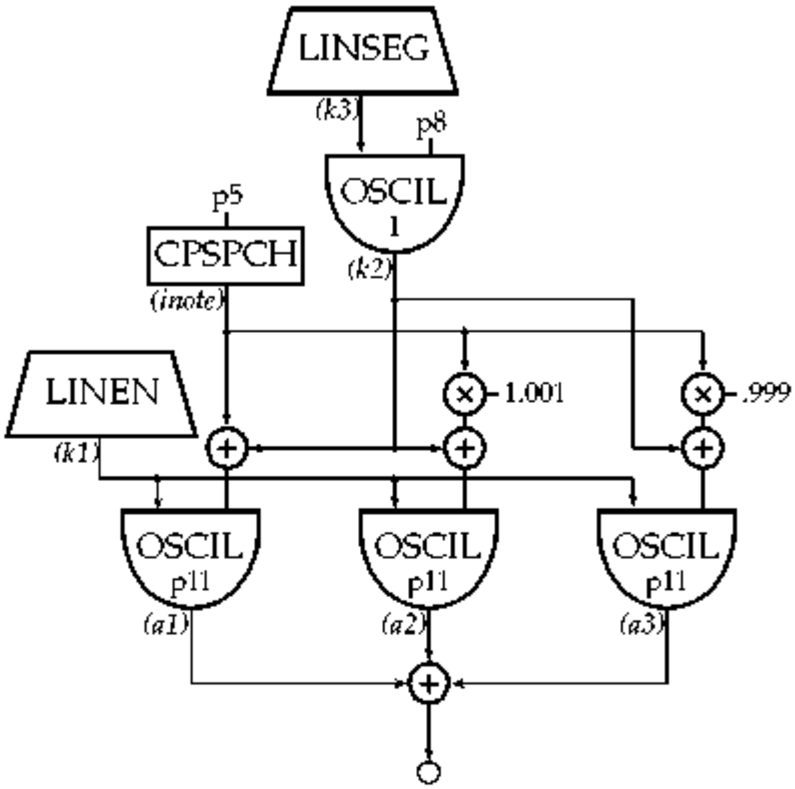
```
p1 - table number - In the orchestra, you reference this table by its
                    number.
p2 - creation time - The time at which the function is generated.
p3 - table size - Number of points in table - must be a power of 2,
                    or that plus 1.
p4 - generating subroutine - 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

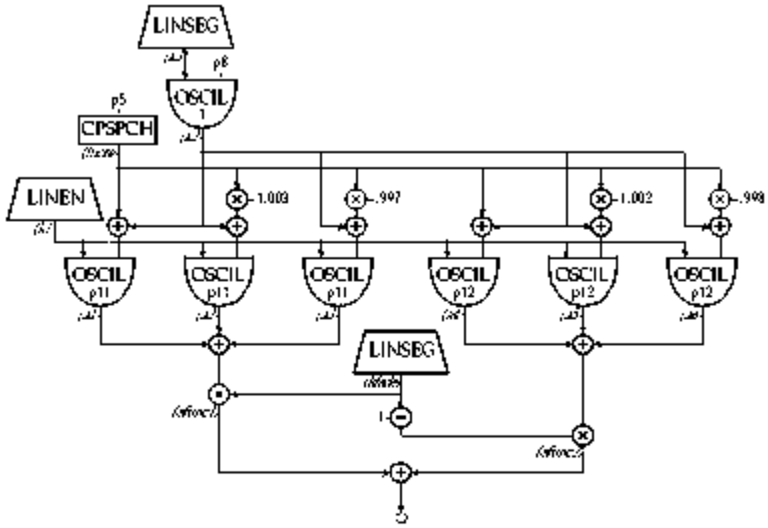
1.8 Toot 7: Crossfade

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

1.9 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:

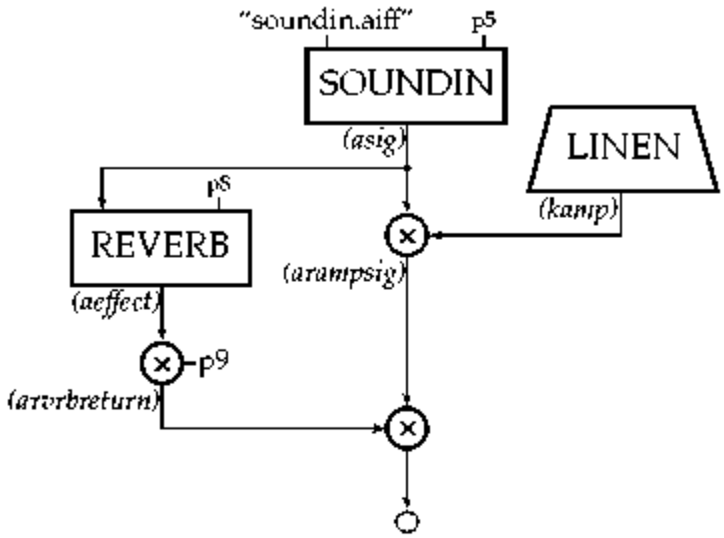
```
a1        soundin     "soundin.5",  .5
```

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**

1.10 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.

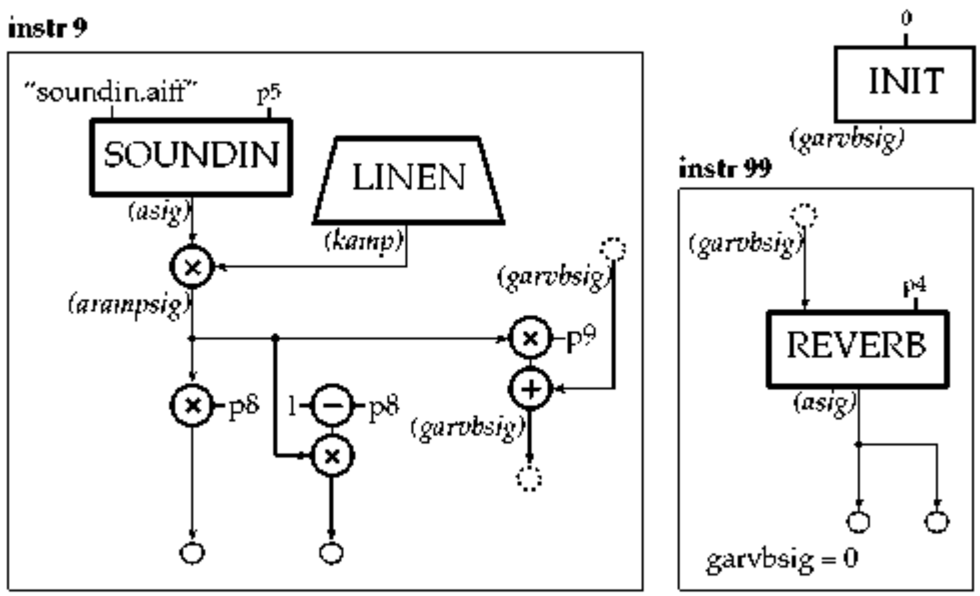
The reverb instr99 below receives input from instr9 via the global a-rate variable *garvbsig*. Since instr9 *adds into* this global, several copies of instr9 can do this without losing any data. The addition requires *garvbsig* to be cleared before each k-rate pass through any active instruments. This 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
```



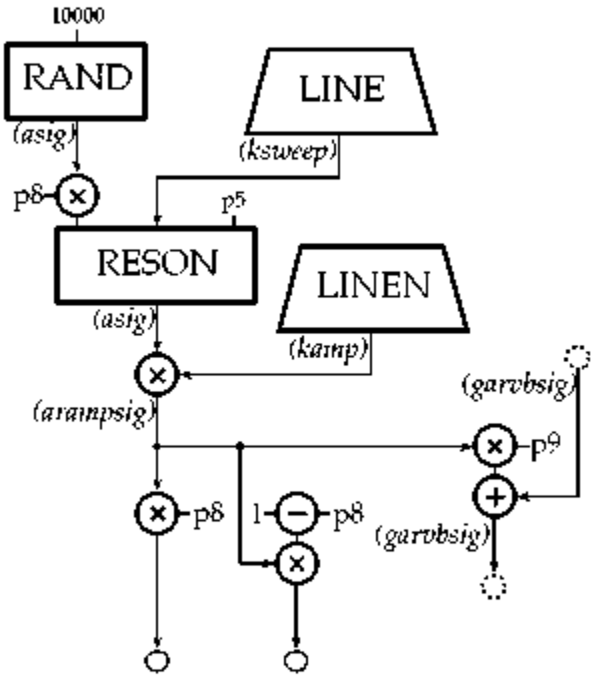
Toot 9: Global Stereo Reverb

1.11 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

1.12 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.

1.12.1 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
```

1.12.2 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
```

1.12.3 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
```

1.12.4 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.

1.12.5 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
```

1.12.6 SORT

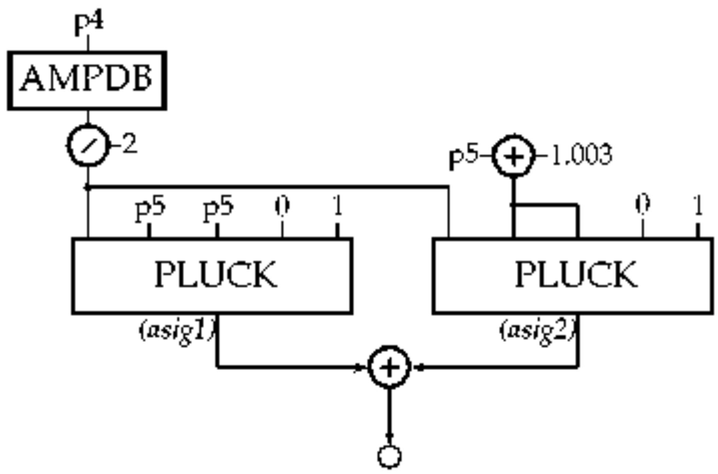
During preprocessing of a score section, all action-time statements are sorted into chronological order by p2 value. This means that notes can be entered in any order, that you can merge files, or work on instruments as temporarily separate sections, then have them sorted automatically when you run Csound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

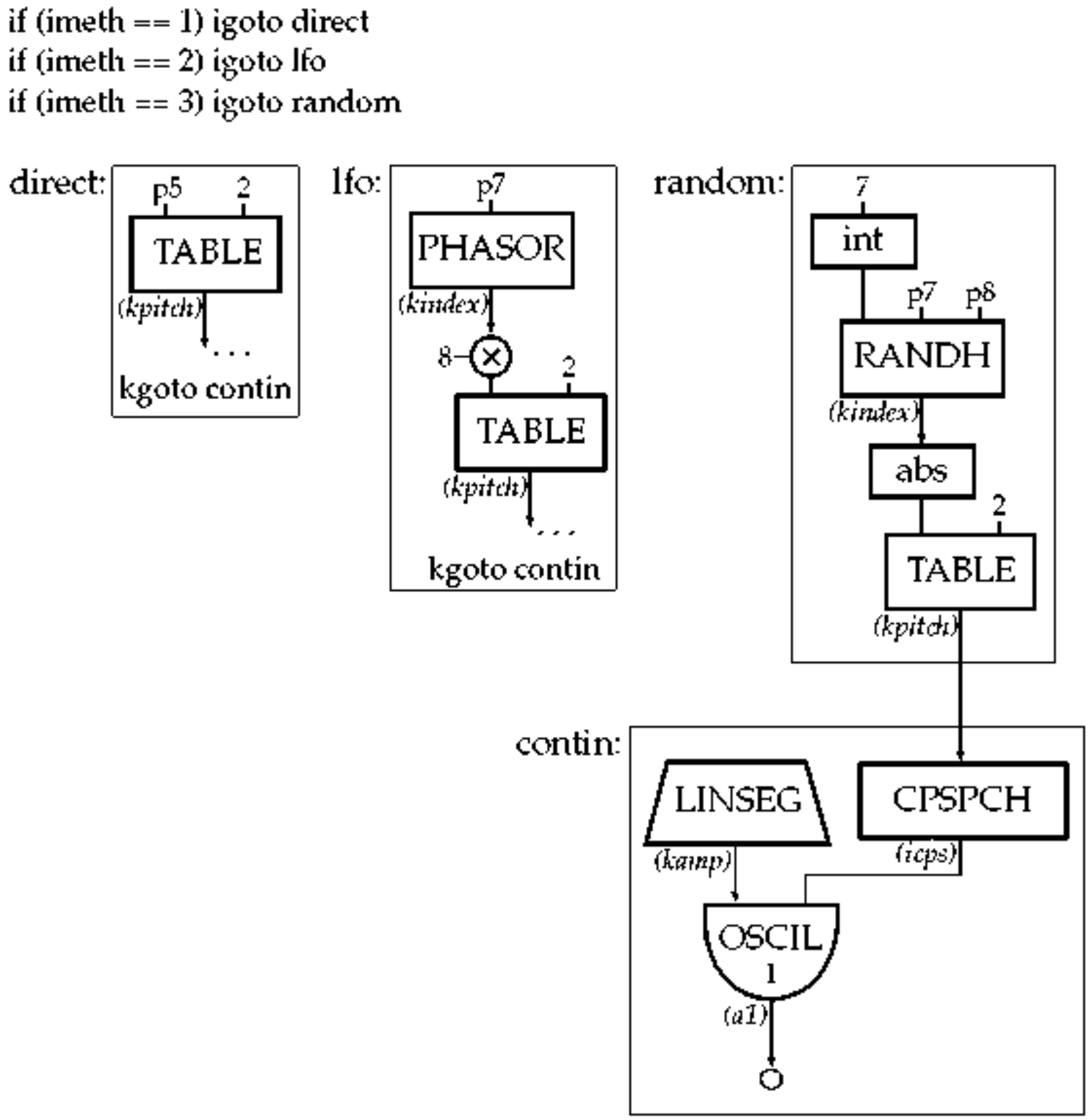
1.13 Toot 12: Tables and Labels

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
```



Toot 12: Tables and Labels

1.14 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]
```

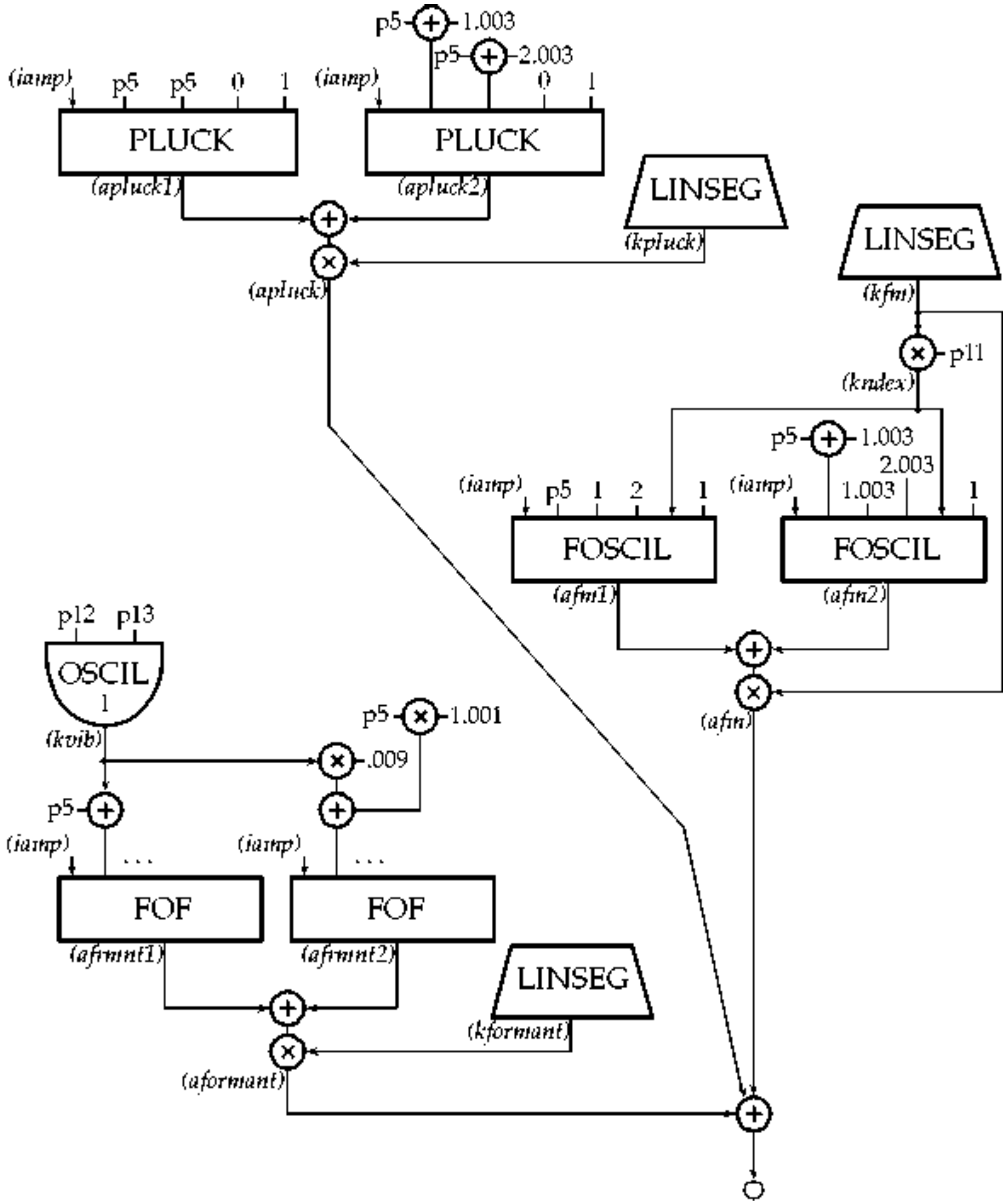
foscil is a composite unit which banks two oscillators in a simple FM configuration, wherein the audio-rate output of one (the "modulator") is used to modulate the frequency input of another (the "carrier.")

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

1.15 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.

1.16 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