Csound

A Manual for the Audio Processing System

and

Supporting Programs

with

Tutorials

Barry Vercoe
Media Lab
M.I.T.

Developed by Barry L. Vercoe at the Experimental Music Studio, Media Laboratory, M.I.T., Cambridge, Massachusetts, with partial support from the System Development Foundation and from National Science Foundation Grant # IRI-8704665.

CONTENTS

Editing by

LUCA PAVAN

pavan@panservice.it

0. PREFACE

Realizing music by digital computer involves synthesizing audio signals with discrete points or samples that are representative of continuous waveforms. There are several ways of doing this, each affording a different manner of control. Direct synthesis generates waveforms by sampling a stored function representing a single cycle; additive synthesis loudness envelope; subtractive synthesis begins with a complex tone and filters it. Non-linear synthesis uses frequency modulation and waveshaping to give simple signals complex characteristics, while sampling and storage of natural sound allows it to be used at will.

Since comprehensive moment-by-moment specification of sound can be tedious, control is gained in two ways: 1) from the instruments in an orchestra, and 2) from the events within a score. An orchestra is really a computer program that can produce sound, while a score is a body of data which that program can react to. Whether a rise-time characteristic is a fixed constant in an instrument, or a variable of each note in the score, depends on how the user wants to control it.

The instruments in a Csound orchestra are defined in a simple syntax that invokes complex audio processing routines. A score passed to this orchestra contains numerically coded pitch and control information, in standard numeric score format. Although many users are content with this format, higher level score processing languages are often convenient.

The programs making up the Csound system have a long history of development, beginning with the Music 4 program written at Bell Telephone Laboratories in the early 1960's by Max Mathews. That initiated the stored table concept and much of the terminology that has since enabled computer music researchers to communicate.
Valuable additions were made at Princeton by the late Godfrey Winham in Music 4B; my own Music 360 (1968) was very indebted to his work. With Music 11 (1973) I took a different tack: the two distinct networks of control and audio signal processing stemmed from my intensive involvement in the preceding years in hardware synthesizer concepts and design. This division has been retained in Csound.

Because it is written entirely in C, Csound is easily installed on any machine running Unix or C. At MIT it runs on VAX/DECstations under Ultrix 4.2, on SUNs under OS 4.1, SGIs under 4.1, and on the Macintosh under ThinkC 4.0. With this single language for audio signal processing, users move easily from machine to machine.

The 1991 version included many new features. I am indebted to others for the contribution of the phase vocoder and FOF synthesis modules. That release also charted a new direction with the addition of a spectral data type, holding much promise for future development. The 1992 release is even more significant for its addition of MIDI converter and control units, enabling Csound to be run from MIDI score-files and from external MIDI keyboards. Since the newest RISC processors bring to computer music an order of magnitude more speed than did those on which it was born, researchers and composers now have access to workstations on which realtime software synthesis with sensing and control is now a reality. This is perhaps the single most important development for people working in the field. This new Csound is designed to take maximum advantage of realtime audio processing, and to encourage interactive experiments in this exciting new domain.

B.V.

1. A BEGINNING TUTORIAL

Introduction

The purpose of this section is to expose the reader to the fundamentals of designing and using computer music instruments in Csound. Only a small portion of the language will be covered here, sufficient to implement some simple instrument examples.

The sections in this primary text are arranged as a Reference manual (not a tutorial), since that is the form the user will eventually find most helpful when inventing instruments. Once the basic concepts are grasped from this beginning tutorial, the reader might let himself into the remainder of the text by locating the information presented here in the Reference entries that follow. More comprehensive tutorials are supplied as Appendices.

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. 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 = 20000      ; audio sampling rate is 20 kHz
kr = 500        ; control rate is 500 Hz
ksmps = 40      ; 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.

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 12
```

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 permissible 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". Tne next section will describe how to invoke a Csound orchestra to perform a Csound score.

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 test 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
```

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-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-variables receive their values at this time, and any other expressions involving just constants and i-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.

2. SYNTAX OF THE ORCHESTRA

An orchestra statement in Csound has the format:

```
label:  result   opcode   argument1, argument2, ... ; comments
```

The label is optional and identifies the basic statement that follows as the potential target of a go-to operation (see Program Control Statements). A label has no effect on the statement per se.

Comments are optional and are for the purpose of letting the user document his orchestra code. Comments always begin with a semicolon (;) and extend to the end of the line.

The remainder (result, opcode, and arguments) form the basic statement. This also is optional, i.e. a line may have only a label or comment or be entirely blank. If present, the basic statement must be complete on one line. The opcode determines the operation to be performed; it usually takes some number of input values (arguments); and it usually has a result field variable to which it sends output values at some fixed rate.
There are four possible rates:

1) once only, at orchestra setup time (effectively a permanent assignment);
2) once at the beginning of each note (at initialization (init) time: I-rate);

3) once every performance-time control loop (perf time control rate, or K-rate);
4) once each sound sample of every control loop (perf time audio rate, or A-rate).

ORCHESTRA STATEMENT TYPES

An orchestra program in Csound is comprised of orchestra header statements which set various global parameters, followed by a number of instrument blocks representing different instrument types. An instrument block, in turn, is comprised of ordinary statements that set values, control the logical flow, or invoke the various signal processing subroutines that lead to audio output.

An orchestra header statement operates once only, at orchestra setup time. It is most commonly an assignment of some value to a global reserved symbol, e.g. sr = 20000. All orchestra header statements belong to a pseudo instrument 0, an init pass of which is run prior to all other instruments at score time 0. Any ordinary statement can serve as an orchestra header statement, eg. gifreq = cpspch(8.09), provided it is an init-time only operation.

An ordinary statement runs at either init time or performance time or both. Operations which produce a result formally run at the rate of that result (that is, at init time for I-rate results; at performance time for K- and A-rate results), with the sole exception of the init opcode (q.v.). Most generators and modifiers, however, produce signals that depend not only on the instantaneous value of their arguments but also on some preserved internal state. These performance-time units therefore have an implicit init-time component to set up that state. The run time of an operation which produces no result is apparent in the
opcode.

Arguments are values that are sent to an operation. Most arguments will accept arithmetic expressions composed of constants, variables, reserved globals, value converters, arithmetic operations and conditional values; these are described below.

CONSTANTS AND VARIABLES

constants are floating point numbers, such as 1, 3.14159, or -73.45. They are available continuously and do not change in value.

variables are named cells containing numbers. They are available continuously and may be updated at one of the four update rates (setup only, I-rate, K-rate, or A-rate). I- and K-rate variables are scalars (i.e. they take on only one value at any given time) and are primarily used to store and recall controlling data, that is, data that changes at the note rate (for I-variables) or at the control rate (for K-variables). I- and K-variables are therefore useful for storing note parameter values, pitches, durations, slow-moving frequencies, vibratos, etc. A-variables, on the other hand, are arrays or vectors of information. Though renewed on the same perf-time control pass as K-variables, these array cells represent a finer resolution of time by dividing the
control period into sample periods (see ksmps below).
A-variables are used to store and recall data changing at the audio sampling rate (e.g. output signals of oscillators, filters, etc.).

A further distinction is that between local and global variables.
local variables are private to a particular instrument, and cannot be read from or written into by any other instrument. Their values are preserved, and they may carry information from pass to pass (e.g. from initialization time to performance time) within a single instrument. Local variable names begin with the letter p, i, k, or a. The same local variable name may appear in two or more different instrument blocks without conflict.

global variables are cells that are accessible by all instruments. The names are either like local names preceded by the letter g, or are special reserved symbols. Global variables are used for broadcasting general values, for communicating between instruments (semaphores), or for sending sound from one instrument to another (e.g. mixing prior to reverberation).

Given these distinctions, there are eight forms of local and global variables:

type	when renewable	Local	Global
reserved symbols	permanent	--	rsymbol
score parameter fields	I-time	pnumber	--
init variables	I-time	iname	giname
control signals	P-time, K-rate	kname	gkname
audio signals	P-time, A-rate	aname	ganame

where rsymbol is a special reserved symbol (e.g. sr, kr), number is a positive integer referring to a score statement pfield, and name is a string of letters and/or digits with local or global meaning. As might be inferred, score parameters are local I-variables whose values are copied from the invoking score statement just prior to the Init pass through an instrument.

VALUE CONVERTERS

ftlen(x)	(init rate args only)
int(x)	(init- or control-rate args only)
frac(x)	(init- or control-rate args only)
dbamp(x)	(init- or control-rate args only)
i(x)	(control-rate args only)
abs(x)	(no rate restriction)
exp(x)	(no rate restriction)
log(x)	(no rate restriction)
sqrt(x)	(no rate restriction)
sin(x)	(no rate restriction)
cos(x)	(no rate restriction)
ampdb(x)	(no rate restriction)

where the argument within the parentheses may be an expression.

Value converters perform arithmetic translation from units of one kind to units of another. The result can then be a term in a further expression.

ftlen(x)	returns the size (no. of points) of stored function table no. x.
int(x)	returns the integer part of x.
frac(x)	returns the fractional part of x.
dbamp(x)	returns the decibel equivalent of the raw amplitude x.
i(x)	returns an Init-type equivalent of the argument, thus permitting a K-time value to be accessed in at init-time or reinit-time, whenever valid.
abs(x)	returns the absolute value of x.
exp(x)	returns e raised to the xth power.
log(x)	returns the natural log of x (x positive only).
sqrt(x)	returns the square root of x (x non-negative).
sin(x)	returns the sine of x (x in radians).
cos(x)	returns the cosine of x (x in radians).

ampdb(x) returns the amplitude equivalent of the decibel value x.
Thus 60 db gives 1000, 66 db gives 2000, 72 db gives 4000, 78 db gives 8000, 84 db gives 16000 and 90 db gives 32000.

Note that for log, sqrt, and ftlen the argument value is restricted. Note also that ftlen will always return a power-of-2 value, i.e. the function table guard point (see F statement) is not included.

PITCH CONVERTERS

octpch(pch)	(init or control rate args only)
pchoct(oct)	(init- or control-rate args only)
cpspch(pch)	(init- or control-rate args only)
octcps(cps)	(init- or control-rate args only)
cpsoct(oct)	(no rate restriction)

where the argument within the parentheses may be a further expression.

These are really value converters with a special function of manipulating pitch data.

Data concerning pitch and frequency can exist in any of the following forms:

name	abbreviation
octave point pitch-class (8ve.pc)	pch
octave point decimal	oct
cycles per second	cps

The first two forms consist of a whole number, representing octave registration, followed by a specially interpreted fractional part. For pch, the fraction is read as two decimal digits representing the 12 equal-tempered pitch classes from .00 for C to.11 for B. For oct, the fraction is interpreted as a true decimal fractional part of an octave. The two fractional forms are thus related by the factor 100/12. In both forms, the fraction is preceded by a whole number octave index such that 8.00 represents Middle C,9.00 the C above, etc. Thus A440 can be represented alternatively by 440 (cps),8.09 (pch), 8.75 (oct), or 7.21 (pch), etc. Microtonal divisions of the pch semitone can be encoded by using more than two decimal places.

The mnemonics of the pitch conversion units are derived from morphemes of the forms involved, the second morpheme describing the source and the first morpheme the object (result). Thus

```
cpspch(8.09)
```

will convert the pitch argument 8.09 to its cps (or Hertz) equivalent, giving the value of 440. Since the argument is constant over the duration of the note, this conversion will take place at I-time, before any samples for the current note are produced. By contrast, the conversion

```
cpsoct(8.75 + K1)
```

which gives the value of A440 transposed by the octave interval K1 will repeat the calculation every, K-period since that is the rate at which K1 varies.

N.B. The conversion from pch or oct into cps is not a linear operation but involves an exponential process that could be time-consuming when executed repeatedly. Csound now uses a built-in table lookup to do this efficiently, even at audio rates.

ARITHMETIC OPERATIONS:

```
- a
+ a
a && b      (logical AND; not audio-rate)
a || b      (logical OR;  not audio-rate)
a + b
a - b
a * b
a / b
```

where the arguments a and b may be further expressions.

Arithmetic operators perform operations of change-sign (negate), don't-change-sign, logical AND logical OR, add, subtract, multiply and divide. Note that a value or an expression may fall between two of these operators, either of which could take it as its left or right argument, as in

```
a + b * c.
```

In such cases three rules apply:

1) * and / bind to their neighbors more strongly than + and -.
   Thus the above expression is taken as
   a + (b * c),
   with * taking b and c and then + taking a and b * c.
2) + and - bind more strongly than &&, which in turn is stronger than ||:
   a && b - c || d is taken as (a && (b-c)) || d
3) When both operators bind equally strongly,
   the operations are done left to right:
   a- b - c is taken as (a - b) - c.

Parentheses may be used as above to force particular groupings.

CONDITIONAL VALUES:

```
(a > b  ? v1 : v2)
(a < b  ? v1 : v2)
(a > = b  ? v1 : v2)
(a < = b  ? v1 : v2)
(a = = b  ? v1 : v2)
(a ! = b  ? v1 : v2)
```

where a, b, v1 and v2 may be expressions, but a, b not audio-rate.

In the above conditionals, a and b are first compared. If the indicated relation is true (a greater than b, a less than b, a greater than or equal to b, a less than or equal to b, a equal to b, a not equal to b), then the conditional expression has the value of v1; if the relation is false, the expression has the value of v2. (For convenience, a sole `=` will function as `= =`.) NB.: If v1 or v2 are expressions, these will be evaluated before the conditional is determined.

In terms of binding strength, all conditional operators (i. e., the relational operators (>,<, etc.), and ?, and :) are weaker than the arithmetic and logical operators (+, -, *, /, && and ||).

Example:

```
(k1 < p5/2 + p6 ? k1 : p7)
```

binds the terms p5/2 and p6. It will return the value k1 below this threshold, else the value p7.

EXPRESSIONS:

Expressions may be composed to any depth from the components shown above. Each part of an expression is evaluated at its own proper rate. For instance, if the terms within a sub-expression all change at the control rate or slower, the sub-expression will be evaluated only at the control rate; that result might then be used in an audio-rate evaluation. For example, in

```
k1 + abs(int(p5) + frac(p5) * 100/12 +  sqrt(k1))
```

the 100/12 would be evaluated at orch init, the p5 expressions evaluated at note I-time, and the remainder of the expression evaluated every k-period. The whole might occur in a unit generator argument position, or be part of an assignment statement .

DIRECTORIES and FILES:

Many generators and the csound command itself specify filenames to be read from or written to. These are optionally full pathnames, whose target directory is fully specified. When not fullpath, filenames are sought in several directories in order, depending on their type and on the setting of certain environment variables. The latter are optional, but they can serve to partition and organize the directories so that source files can be shared rather than duplicated in several user directories.
The environment variables can define directories for soundfiles (SFDIR), sound samples (SSDIR), and sound analysis (SADIR). The search order is:

Soundfiles being written are placed in SFDIR (if it exists), else the current directory.
Soundfiles for reading are sought in the current directory, then SSDIR, then SFDIR.
Analysis control files for reading are sought in the current directory, then SADIR.

NOMENCLATURE:

In Csound there are nine statement types, each of which provides a heading for the descriptive sections that follow in this chapter:

assignment statements	signal generator statements
orchestra header statements	signal modifier statements
instrument block statements	ignal display statements
program control statements	soundfile access statements
duration control statements	

Throughout this document, opcodes are indicated in boldface and their argument and result mnemonics, when mentioned in the text, are given in italics. Argument names are generally mnemonic (amp, phs), and the result is denoted the letter r. Both are preceded by a type qualifier i, k, a or x (e.g. kamp, iphs, ar).
The prefix i denotes scalar values valid at note Init time; prefixes k or a denote control (scalar) and audio (vector) values, modified and referenced continuously throughout performance (i.e. at every control period while the instrument is active). Arguments are used at the prefix-listed times; results are created at their listed times, then remain available for use as inputs elsewhere. The validity of inputs is defined by the following:

arguments with prefix i must be valid at Init time;
arguments with prefix k can be either control or Init values (which remain valid);
arguments with prefix a must be vector inputs;
arguments with prefix x may be either vector or scalar (the compiler will distinguish).

All arguments, unless otherwise stated, can be expressions whose results conform to the above. Most opcodes (such as linen and oscil) can be used in more than one mode, which one being determined by the prefix of the result symbol.

ASSIGNMENT STATEMENTS

```
ir  =    iarg
kr  =    karg
ar  =    xarg
kr  init iarg
ar  init iarg

ir  tival

ir  divz ia, ib, isubst   (these not yet implemented)
kr  divz ka, kb, ksubst
ar  divz xa, xb, ksubst
```

= (simple assignment) - Put the value of the expression iarg (karg, xarg) into the named result. This provides a means of saving an evaluated result for later use.

init - Put the value of the I-time expression iarg into a K- or A-variable, i.e., initialize the result. Note that init provides the only case of an Init-time statement being permitted to write into a Perftime (K- or A-rate) result cell; the statement has no effect at Perf-time.

tival - Put the value of the instrument's internal "tie-in" flag into the named I-variable. Assigns 1 if this note has been 'tied' onto a previously held note (see I Statement); assigns 0 if no tie actually took place. (See also tigoto.)

divz - Whenever b is not zero, set the result to the value a / b; when b is zero, set it to the value of subst instead.

Example:

```
kcps  = i2/3 + cpsoct(k2 + octpch(p5))
```

ORCHESTRA HEADER STATEMENTS

```
sr = n1
kr = n2
ksmps = n3
nchnls = n4
```

These statements are global value assignments, made at the beginning of an orchestra, before any instrument block is defined. Their function is to set certain reserved symbol variables that are required for performance. Once set, these reserved symbols can be used in expressions anywhere in the orchestra.

sr = (optional) - set sampling rate to n1 samples per second per channel. The default value is 10000.

kr = (optional) - set control rate to n2 samples per second. The default value is 1000.

ksmps = (optional) - set the number of samples in a Control Period to n3. This value must equal sr/kr. The default value is 10.

nchnls = (optional) - set number of channels of audio output to n4. (1 = mono, 2 = stereo, 4 = quadraphonic.) The default value is 1 (mono).

In addition, any global variable can be initialized by an init-time assignment anywhere before the first instr statement.

All of the above assignments are run as instrument 0 (i - pass only) at the start of real performance.

Example of header assignments:

```
sr = 10000
kr = 500
ksmps = 20

gi1  =   sr/2.
ga   init 0
gitranspose =  octpch(.0l)
```

INSTRUMENT BLOCK STATEMENTS

```
instr       i, j, ...
.
.   < body
.     of
.     instrument >
.
endin
```

These statements delimit an instrument block. They must always occur in pairs.

instr - begin an instrument block defining instruments i, j, ...

i, j, ... must be numbers, not expressions. Any positive integer is legal, and in any order, but excessively high numbers are best avoided.

endin - end the current instrument block.

Note:

There may be any number of instrument blocks in an orchestra.

Instruments can be defined in any order (but they will always be both initialized and performed in ascending instrument number order).

Instrument blocks cannot be nested (i.e. one block cannot contain another).

PROGRAM CONTROL STATEMENTS

igoto label
tigoto label
kgoto label
goto label
if ia R ib igoto label
if ka R kb kgoto label
if ia R ib goto label
timout istrt, idur, label

where label is in the same instrument block and is not an expression, and where R is one of the Relational operators (>, <, >=, <=, ==, !=) (and = for convenience, see also under Conditional values).

These statements are used to control the order in which statements in an instrument block are to be executed. I-time and P-time passes can be controlled separately as follows:

igoto - During the I-time pass only, unconditionally transfer control to the statement labeled by label.

tigoto - similar to igoto, but effective only during an I-time pass at which a new note is being 'tied' onto a previously held note (see I Statement); no-op when a tie has not taken place.
Allows an instrument to skip initialization of units according to whether a proposed tie was in fact successful (see also tival, delay).

kgoto - During the P-time passes only, unconditionally transfer control to the statement labeled by label.

goto - (combination of igoto and kgoto) Transfer control to label on every pass.

if...igoto - conditional branch at I-time, depending on the truth value of the logical expression "ia R ib". The branch is taken only if the result is true.

if...kgoto - conditional branch during P-time, depending on the truth value of the logical expression "ka R kb". The branch is taken only if the result is true.

if...goto - combination of the above. Condition tested on every pass.

timout - conditional branch during P-time, depending on elapsed note time. istrt and idur specify time in seconds. The branch to label will become effective at time istrt, and will remain so for just idur seconds. Note that timout can be reinitialized for multiple activation within a single note (see example next page).

Example:

```
if k3 > p5 + 10 kgoto next
```

reinit label
rigoto label
rireturn

These statements permit an instrument to reinitialize itself during performance.

reinit - whenever this statement is encountered during a P-time pass, performance is temporarily suspended while a special Initialization pass, beginning at label and continuing to rireturn or endin, is executed. Performance will then be resumed from where it left off.

rigoto - similar to igoto, but effective only during a reinit pass (i.e., No-op at standard I-time). This statement is useful for bypassing units that are not to be reinitialized.

rireturn - terminates a reinit pass (i.e., No-op at standard I-time). This statement, or an endin, will cause normal performance to be resumed.

Example:

The following statements will generate an exponential control signal whose value moves from 440 to 880 exactly ten times over the duration p3.

```
reset:  timout   0, p3 /10, contin   ;after p3/10 seconds,
        reinit   reset               ; reinit both timout
contin: expon    440, p3/10,880      ; and expon
        riretum                      ; then resume perf
```

DURATION CONTROL STATEMENTS

ihold
turnoff

These statements permit the current note to modify its own duration.

ihold - this I-time statement causes a finite-duration note to become a `held' note. It thus has the same effect as a negative p3 (see Score I-statement), except that p3 here remains positive and the instrument reclassifies itself to being held indefinitely. The note can be turned off explicitly with turnoff, or its space taken over by another note of the same instrument number (i.e. it is tied into that note). Effective at I-time only; no-op during a reinit pass.

turnoff - this P-time statement enables an instrument to turn itself off. Whether of finite duration or 'held', the note currently being performed by this instrument is immediately removed from the active note list. No other notes are affected.

Example:

The following statements will cause a note to terminate when a control signal passes a certain threshold (here the Nyquist frequency).

```
k1 expon   440, p3/10,880    ; begin gliss and continue
if  k1 < sr/2 kgoto contin   ; until Nyquist detected
    turnoff                  ; then quit
    contin: a1  oscil  a1, k1, 1
```

MIDI CONVERTERS

ival	**notnum**	
ival	**veloc**	
icps	**cpsmidi**	
icps	**cpsmidib**	
kcps	**cpsmidib**	
ioct	**octmidi**	
ioct	**octmidib**	
koct	**octmidib**	
ipch	**pchmidi**	
ipch	**pchmidib**	
kpch	**pchmidib**	
iamp	**ampmidi**	iscal[, ifn]
kaft	**aftouch**	iscal
kchpr	**chpress**	iscal
kbend	**pchbend**	iscal
ival	**midictrl**	inum
kval	**midictrl**	inum

Get a value from the MIDI event that activated this instrument, or from a continuous MIDI controller, and convert it to a locally useful format.

INITIALIZATION

iscal - I-time scaling factor.

ifn (optional) - function table number of a normalized translation table, by which the incoming value is first interpreted. The default value is 0, denoting no translation.

inum - MIDI controller number.

PERFORMANCE

notnum, **veloc** - get the MIDI byte value (0 - 127) denoting the note number or velocity of the current event.

cpsmidi, **octmidi**, **pchmidi** - get the note number of the current MIDI event, expressed in cps, oct, or pch units for local processing.

cpsmidib, **octmidib**, **pchmidib** - get the note number of the current MIDI event, modify it by the current pitch-bend value, and express the result in cps, oct, or pch units. Available as an I-time value or as a continuous ksig value.

ampmidi - get the velocity of the current MIDI event, optionally pass it through a normalized translation table, and return an amplitude value in the range 0 - iscal.

aftouch, **chpress**, **pchbend** - get the current after-touch, channel pressure, or pitch-bend value for this channel, rescaled to the range 0 - iscal. Note that this access to pitch-bend data is independent of the MIDI pitch, enabling the value here to be used for any arbitrary purpose.

midictrl - get the current value (0 - 127) of a specified MIDI controller.

SIGNAL GENERATORS

kr	**line**	ia, idur1, ib
ar	**line**	ia, idur1, ib
kr	**expon**	ia, idur1, ib
ar	**expon**	ia, idur1, ib
kr	**linseg**	ia, idur1, ib[, idur2, ic[...]]
ar	**linseg**	ia, idur1, ib[, idur2, icI...]]
kr	**expseg**	ia, idur1, ib[, idur2, ic[...]]
ar	**expseg**	ia, idur1, ib[, idur2, ic[...]]

Output values kr or ar trace a straight line (exponential curve) or a series of line segments (exponential segments) between specified points.

INITIALIZATION

ia - starting value. Zero is illegal for exponentials.

ib, ic, etc. - value after dur1 seconds, etc. For exponentials, must be non-zero and must agree in sign with ia.

idur1 - duration in seconds of first segment. A zero or negative value will cause all initialization to be skipped.

idur2, idur3, etc. - duration in seconds of subsequent segments.
A zero or negative value will terminate the initialization process with the preceding point, permitting the last-defined line or curve to be continued indefinitely in performance. The default is zero.

PERFORMANCE

These units generate control or audio signals whose values can pass through 2 or more specified points. The sum of dur values may or may not equal the instrument's performance time: a shorter performance will truncate the specified pattern, while a longer one.will cause the last-defined segment to continue on in the same direction.

Example:

```
k2   expseg 440, p3/2,880, p3/2,440
```

This statement creates a control signal which moves exponentially from 440 to 880 and back, over the duration p3.

kr	**phasor**	kcps[, iphs]
ar	**phasor**	xcps[, iphs]

Produce a normalized moving phase value.

INITIALIZATION

iphs (optional) - initial phase, expressed as a fraction of a cycle (0 to 1). A negative value will cause phase initialization to he skipped. The default value is zero.

PERFORMANCE
An internal phase is successively accumulated in accordance with the cps frequency to produce a moving phase value, normalized to lie in the range 0. <= phs < 1.

When used as the index to a table unit, this phase (multiplied by the desired function table length) will cause it to behave like an oscillator.

Note that **phasor** is a special kind of integrator, accumulating phase increments that represent frequency settings.

Example:

```
phasor       1                    ; cycle once per second
kpch table   k1 * 12, 1           ; through 12-note pch table
a1   oscil   p4, cpspch(kpch), 2  ; with continuous sound
```

ir	**table**	indx, ifn[, ixmode][, ixoff][, iwrap]
ir	**tablei**	indx, ifn[, ixmode][, ixoff][, iwrap]
kr	**table**	kndx, ifn[, ixmode][, ixoff][, iwrap]
kr	**tablei**	kndx, ifn[, ixmode][, ixoff][, iwrap]
ar	**table**	andx, ifn[, ixmode][, ixoff][, iwrap]
ar	**tablei**	andx, ifn[, ixmode][, ixoff][, iwrap]
kr	**oscil1**	idel, kamp, idur, ifn
kr	**oscil1i**	idel, kamp, idur, ifn

Table values are accessed by direct indexing or by incremental sampling.

INITIALIZATION

ifn - function table number. tablei, oscil1i require the extended guard point.

ixmode (optional) - ndx data mode. 0 = raw ndx, 1 = normalized (0 to 1). The default value is 0.

ixoff (optional) - amount by which ndx is to be offset. For a table with origin at center, use tablesize/2 (raw) or .5 (normalized). The default value is 0.

iwrap (optional) - wraparound ndx flag. 0 = nowrap (ndx < 0 treated as ndx=0; ndx > tablesize sticks at ndx=size), 1 = wraparound. The default value is 0.

idel - delay in seconds before oscil1 incremental sampling begins.

idur - duration in seconds to sample through the oscil1 table just once. A zero or negative value will cause all initialization to be skipped.

PERFORMANCE

table invokes table lookup on behalf of init, control or audio indices. These indices can be raw entry numbers (0,l,2...size - 1) or scaled values (0 to 1-e). Indices are first modified by the offset value then checked for range before table lookup (see iwrap). If ndx is likely to be full scale, or if interpolation is being used, the table should have an extended guard point. table indexed by a periodic phasor (see phasor) will simulate an oscillator.

oscil1 accesses values by sampling once through the function table at a rate determined by idur. For the first idel seconds, the point of scan will reside at the first location of the table; it will then begin moving through the table at a constant rate, reaching the end in another idur

seconds; from that time on (i.e. after idel + idur seconds) it will remain pointing at the last location. Each value obtained from sampling is then multiplied by an amplitude factor kamp before being written into the result.

tablei and **oscil1i** are interpolating units in which the fractional part of ndx is used to interpolate between adjacent table entries. The smoothness gained by interpolation is at some small cost in execution time (see also oscili, etc.), but the interpolating and non-interpolating units are otherwise interchangeable. Note that when tablei uses a periodic index whose modulo n is less than the power of 2 table length, the interpolation process requires that there be an (n + 1)th table value that is a repeat of the 1st (see F statement in Score).

kr	**oscil**	kamp, kcps, ifn[, iphs]
kr	**oscili**	kamp, kcps, ifn[, iphs]
ar	**oscil**	xamp, xcps, ifn[, iphs]
ar	**oscili**	xamp, xcps, ifn[, iphs]
ar	**foscil**	xamp, kcps, kcar, kmod, kndx, ifn[, iphs]
ar	**foscili**	xamp, kcps, kcar, kmod, kndx, ifn[, iphs]

Table ifn is incrementally sampled modulo the table length and the value obtained is multiplied by amp.

INITIALIZATION

ifn - function table number. Requires a wrap-around guard point.

iphs (optional) - initial phase of sampling, expressed as a fraction of a cycle (0 to 1). A negative value will cause phase initialization to be skipped. The default value is 0.

PERFORMANCE

The **oscil** units output periodic control (or audio) signals consisting of the value of kamp(xamp)times the value returned from control rate (audio rate) sampling of a stored function table. The internal phase is simultaneously advanced in accordance with the cps input value. While the amplitude and frequency inputs to the K-rate oscils are scalar only, the corresponding inputs to the audio-rate oscils may each be either scalar or vector, thus permitting amplitude and frequency
modulation at either sub-audio or audio frequencies.

foscil is a composite unit that effectively banks two oscils in the familiar Chowning FM setup, wherein the audio-rate output of one generator is used to modulate the frequency input of another (the "carrier"). Effective carrier frequency = kcps * kcar, and modulating frequency = kcps * kmod. For integral values of kcar and kmod, the perceived fundamental will be the minimum positive value of kcps * (kcar - n * kmod), n = 1,1,2,... The input kndx is the index of modulation (usually time-varying and ranging 0 to 4 or so) which determines the spread of acoustic energy over the partial positions given by n = 0,1,2,.., etc. ifn should point to a stored sine wave.

oscili and **foscili** differ from oscil and foscil respectively in that the standard procedure of using a truncated phase as a sampling index is here replaced by a process that interpolates between two successive lookups. Interpolating generators will produce a noticeably cleaner output signal, but they may take as much as twice as long to run. Adequate accuracy can also be gained without the time cost of interpolation by using large stored function tables of 2K, 4K or 8K points if the space is available.

Example:

```
k1  oscil   10, 5, 1              ; 5 cps vibrato
a1  oscil   5000, 440 + k1, 1     ; around A440 +-10 cps
```

ar1 [,ar2] **loscil** xamp, kcps, ifn[, ibas][,imod1,ibeg1,iend1][, imod2,ibeg2,iend2]

Read sampled sound (mono or stereo) from a table, with optional sustain and release looping.

INITIALIZATION

ifn - function table number, typically denoting an AIFF sampled sound segment with prescribed looping points. The source file may be mono or stereo.

ibas (optional) - base frequency in cps of the recorded sound.
This optionally overrides the frequency given in the AIFF file, but is required if the file did not contain one. The default value is 0 (no override).

imod1, mod2 (optional) - play modes for the sustain and release loops. A value of 1 denotes normal looping, 2 denotes forward & backward looping, 0 denotes no looping. The default value (-1) will defer to the mode and the looping points given in the source file.

ibeg1, iend1, ibeg2, iend2 (optional, dependent on mod1, mod2) - begin and end points of the sustain and release loops. These are measured in sample frames from the beginning of the file, so will look the same whether the sound segment is monaural or stereo.

PERFORMANCE

loscil samples the ftable audio at a rate determined by kcps, then multiplies the result by xamp. The sampling increment for kcps is dependent on the table's base-note frequency ibas, and is automatically adjusted if the orchestra sr value differs from that at which the source was recorded. In this unit, ftable is always sampled with interpolation.

If sampling reaches the sustain loop endpoint and looping is in effect, the point of sampling will be modified and loscil will continue reading from within that loop segment. Once the instrument has received a turnoff signal (from the score or from a MIDI noteoff event), the next sustain endpoint encountered will be ignored and sampling will continue towards the release loop end-point, or towards the last sample (henceforth to zeros).

loscil is the basic unit for building a sampling synthesizer.
Given a sufficient set of recorded piano tones, for example, this unit can resample them to simulate the missing tones. Locating the sound source nearest a desired pitch can be done via table lookup. Once a sampling instrument has begun, its turnoff point may be unpredictable and require an external release envelope; this is often done by gating the sampled audio with linenr, which will extend the duration of a turned-off instrument by a specific period while it implements a decay.

Example:

```
inum notnum
icps cpsmidi
iamp ampmidi  3000, 1
ifno table    inum, 2     ;notnum to choose an audio sample
ibas table    inum, 3
kamp linenr   iamp, 0, .05, .01 ;at noteoff, extend by 50 millisecs
asig loscil   kamp, icps, ifno, cpsoct(ibas/12. + 3)
```

ar	**buzz**	xamp, xcps, knh, ifn[, iphs]
ar	**gbuzz**	xamp, xcps, knh, klh, kr, ifn[, iphs]

Output is a set of harmonically related cosine partials.

INITIALIZATION

ifn - table number of a stored function containing (for buzz) a sine wave, or (for gbuzz) a cosine wave. In either case a large table of at least 8192 points is recommended.

iphs (optional) - initial phase of the fundamental frequency, expressed as a fraction of a cycle (0 to 1). A negative value will cause phase initialization to be skipped. The default value is zero.

PERFORMANCE

These units generate an additive set of harmonically related cosine partials of fundamental frequency xcps, and whose amplitudes are scaled so their summation peak equals xamp. The selection and strength of partials is determined by the following control parameters:

knh - total number of harmonics requested. Must be positive.

klh - lowest harmonic present. Can be positive, zero or negative. In gbuzz the set of partials can begin at any partial number and proceeds upwards; if klh is negative, all partials below zero will reflect as positive partials without phase change (since cosine is an even function), and will add constructively to any positive partials in the set.

kr - specifies the multiplier in the series of amplitude coefficients. This is a power series: if the klhth partial has a strength coefficient of A, the (klh + n)th partial will have a coefficient of A * (kr ** n), i.e. strength values trace an exponential curve. kr may be positive, zero or negative, and is not restricted to integers.

buzz and **gbuzz** are useful as complex sound sources in subtractive synthesis. buzz is a special case of the more general gbuzz in which klh = kr = 1; it thus produces a set of knh equal-strength harmonic partials, beginning with the fundamental. (This is a band-limited pulse train; if the partials extend to the Nyquist, i.e. knh = int (sr / 2 / fundamental freq.), the result is a real pulse train of amplitude xamp.) Although both knh and klh may be varied during performance, their internal values are necessarily integer and may cause “pops” due to discontinuities in the output; kr, however, can be varied during performance to good effect. Both buzz and gbuzz can be amplitude- and/or frequency-modulated by either control or audio signals.

N.B. These two units have their analogs in GEN11, in which the same set of cosines can be stored in a function table for sampling by an oscillator. Although computationally more efficient, the stored pulse train has a fixed spectral content, not a time-varying one as above.

ar **adsyn** kamod, kfmod, ksmod, ifilcod
ar **pvoc** ktimpnt, kfmod, ifilcod [, ispecwp]

Output is an additive set of individually controlled sinusoids, using either an oscillator bank or phase vocoder resynthesis.

INITIALIZATION

ifilcod - integer or character-string denoting a control-file derived from analysis of an audio signal. An integer denotes the suffix of a file adsyn.m or pvoc.m; a character-string (in double quotes) gives a filename, optionally a full pathname. If not fullpath, the file is sought first in the current directory, then in the one given by the environment variable SADIR (if defined). adsyn control contains breakpoint amplitude- and frequency-envelope values organized for oscillator resynthesis, while pvoc control contains similar data organized for fft resynthesis. Memory usage depends on the size of the files involved, which are read and held entirely in memory during computation but are shared by multiple calls (see also lpread).

ispecwp (optional) - if non-zero, attempts to preserve the spectral envelope while its frequency content is varied by kfmod.
The default value is zero.

PERFORMANCE

adsyn synthesizes complex time-varying timbres through the method of additive synthesis. Any number of sinusoids, each individually controlled in frequency and amplitude, can be summed by high-speed arithmetic to produce a high-fidelity result.

Component sinusoids are described by a control file describing amplitude and frequency tracks in millisecond breakpoint fashion.
Tracks are defined by sequences of 16-bit binary integers:

-1, time, amp, time, amp,...
-2, time, freq, time, freq,...

such as from hetrodyne filter analysis of an audio file. (For details see the Appendix on hetro.) The instantaneous amplitude and frequency values are used by an internal fixed-point oscillator that adds each active partial into an accumulated output signal. While there is a practical limit (currently 50) on the number of contributing partials, there is no restriction on their behavior over time. Any sound that can be described in terms of the behavior of sinusoids can be synthesized by adsyn alone.

Sound described by an adsyn control file can also be modified during re-synthesis. The signals kamod, kfmod, ksmod will modify the amplitude, frequency, and speed of contributing partials. These are multiplying factors, with kfmod modifying the cps frequency and ksmod modifying the speed with which the millisecond bread-point line-segments are traversed. Thus .7, 1.5, and 2 will give rise to a softer sound, a perfect fifth higher, but only half as long. The values 1,1,1 will leave the sound unmodified. Each of these inputs can be a control signal.

pvoc implements signal reconstruction using an fft-based phase vocoder. The control data stems from a precomputed analysis file with a known frame rate. The passage of time through this file is specified by ktimpnt, which represents the time in seconds.
ktimpnt must always be positive, but can move forwards or backwards in time, be stationary or discontinuous, as a pointer into the analysis file. kfmod is a control-rate transposition factor: a value of 1 incurs no transposition, 1.5 transposes up a perfect fifth, and .5 down an octave.

This implementation of pvoc was written by Dan Ellis. It is based in part on the system of Mark Dolson, but the pre-analysis concept is new.

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

Audio output is a succession of sinusoid bursts initiated at frequency xfund with a spectral peak at xform. For xfund above 25 Hz these bursts produce a speech-like formant with spectral characteristics determined by the k-input parameters. For lower fundamentals this generator provides a special form of granular synthesis.

INITIALIZATION

iolaps - number of preallocated spaces needed to hold overlapping burst data. Overlaps are frequency dependent, and the space required depends on the maximum value of xfund * kdur. Can be over-estimated at no computation cost. Uses less than 50 bytes of memory per iolap.

ifna, ifnb - table numbers of two stored functions. The first is a sine table for sineburst synthesis (size of at least 4096 recommended). The second is a rise shape, used forwards and backwards to shape the sineburst rise and decay; this may be linear (GEN07) or perhaps a sigmoid (GEN19).

itotdur - total time during which this fof will be active.
Normally set to p3. No new sineburst is created if it cannot complete its kdur within the remaining itotdur.

iphs (optional) - initial phase of the fundamental, expressed as a fraction of a cycle (0 to 1). The default value is 0.

ifmode (optional) - formant frequency mode. If zero, each sineburst keeps the xform frequency it was launched with. If non-zero, each is influenced by xform continuously. The default value is 0.

PERFORMANCE

xamp - peak amplitude of each sineburst, observed at the true end of its rise pattern. The rise may exceed this value given a large bandwidth (say, Q < 10) and/or when the bursts are overlapping.

xfund - the fundamental frequency (in Hertz) of the impulses that create new sinebursts.

xform - the formant frequency, i.e. freq of the sinusoid burst induced by each xfund impulse. This frequency can be fixed for each burst or can vary continuously (see ifmode).

koct - octaviation index, normally zero. If greater than zero, lowers the effective xfund frequency by attenuating odd-numbered sinebursts. Whole numbers are full octaves, fractions transitional.

kband - the formant bandwidth (at -6dB), expressed in Hz. The bandwidth determines the rate of exponential decay throughout the sineburst, before the enveloping described below is applied.

kris, kdur, kdec - rise, overall duration, and decay times (in seconds) of the sinusoid burst. These values apply an enveloped duration to each burst, in similar fashion to a Csound linen generator but with rise and decay shapes derived from the ifnb input. kris inversely determines the skirtwidth (at -40 dB) of the induced formant region. kdur affects the density of sineburst overlaps, and thus the speed of computation. Typical values for vocal imitation are .003,.02,.007.

Csound's fof generator is loosely based on Michael Clarke's C-coding of IRCAM's CHANT program (Xavier Rodet et al.). Each fof produces a single formant, and the output of four or more of these can be summed to produce a rich vocal imitation. fof synthesis is a special form of granular synthesis, and this implementation aids transformation between vocal imitation and granular textures. Computation speed depends on kdur, xfund, and the density of any overlaps.

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

Audio output is a naturally decaying plucked string or drum sound based on the Karplus-Strong algorithms.

INITIALIZATION

icps - intended pitch value in cps, used to set up a buffer of 1 cycle of audio samples which will be smoothed over time by a chosen decay method. icps normally anticipates the value of kcps, but may be set artificially high or low to influence the size of the sample buffer.

ifn - table number of a stored function used to initialize the cyclic decay buffer. If ifn = 0, a random sequence will be used instead.

imeth - method of natural decay. There are six, some of which use parameters values that follow.

1 - simple averaging. A simple smoothing process, uninfluenced by parameter values.

2 - stretched averaging. As above, with smoothing time stretched by a factor of iparm1 (>= 1).

3 - simple drum. The range from pitch to noise is controlled by a 'roughness factor' in iparm1 (0 to 1). Zero gives the plucked string effect, while 1 reverses the polarity of every sample (octave down, odd harmonics). The setting .5 gives an optimum snare drum.

4 - stretched drum. Combines both roughness and stretch factors. iparm1 is roughness (0 to 1), and iparm2 the stretch factor (>= 1).

5 - weighted averaging. As method 1, with iparm1 weighting the current sample (the status quo) and iparm2 weighting the previous adjacent one. iparm1 + iparm2 must be <= 1.

6 - 1st order recursive filter, with coefs .5. Unaffected by parameter values.

iparm1, iparm2 (optional) - parameter values for use by the smoothing algorithms (above). The default values are both 0.

PERFORMANCE

An internal audio buffer, filled at I-time according to ifn, is continually resampled with periodicity kcps and the resulting output is multiplied by kamp. Parallel with the sampling, the buffer is smoothed to simulate the effect of natural decay.

Plucked strings (1,2,5,6) are best realized by starting with a random noise source, which is rich in initial harmonics. Drum sounds (methods 3,4) work best with a flat source (wide pulse), which produces a deep noise attack and sharp decay.

The original Karplus-Strong algorithm used a fixed number of samples per cycle, which caused serious quantization of the pitches available and their intonation. This implementation resamples a buffer at the exact pitch given by kcps, which can be varied for vibrato and glissando effects. For low values of the orch sampling rate (e.g. sr = 10000), high frequencies will store only very few samples (sr / icps). Since this may cause noticeable noise in the resampling process, the internal buffer has a minimum size of 64 samples. This can be further enlarged by setting icps to some artificially lower pitch.

kr **rand** xamp[, iseed]
kr **randh** kamp, kcps[, iseed]
kr **randi** kamp, kcps[, iseed]
ar **rand** xamp[, iseed]
ar **randh** xamp, xcps[, iseed]
ar **randi** xamp, xcps[, iseed]

Output is a controlled random number series between +amp and -amp

INITIALIZATION

iseed (optional) - seed value for the recursive psuedo-random formula. A value between 0 and +1 will produce an initial output of kamp * iseed A negative value will cause seed re-initialization to be skipped. The default seed value is .5.

PERFORMANCE

The internal psuedo-random formula produces values which are uniformly distributed over the range kamp to -kamp. rand will thus generate uniform white noise with an R.M.S value of kamp / root 2.

The remaining units produce band-limited noise: the cps parameters permit the user to specify that new random numbers are to be generated at a rate less than the sampling or control frequencies. randh will hold each new number for the period of the specified cycle; randi will produce straightline interpolation between each new number and the next.

Example:

```
i1 = octpch(p5) ; center pitch, to be modified
k1 randh 1,10 ;10 time/sec by random displacements up to 1 octave
   a1 oscil 5000, cpsoct(i1+k1), 1
```

SIGNAL MODIFIERS

kr **linen** kamp, irise, idur, idec
ar **linen** xamp, irise, idur, idec
kr **linenr** kamp, irise, idec, iatdec

ar **linenr** xamp, irise, idec, iatdec
kr **envlpx** kamp, irise, idur, idec, ifn, iatss, iatdec[,ixmod]
ar **envlpx** xamp, irise, idur, idec, ifn, iatss, iatdec[,ixmod]

linen - apply a straight line rise and decay pattern to an input amp signal.
linenr - apply a straight line rise, then an exponential decay while the note is extended in time.
envlpx - apply an envelope consisting of 3 segments: 1) stored function rise shape, 2) modified exponential "pseudo steady state", 3) exponential decay

INITIALIZATION

irise - rise time in seconds. A zero or negative value signifies no rise modification.

idur - overall duration in seconds. A zero or negative value will cause initialization to be skipped.

idec - decay time in seconds. Zero means no decay. An idec > idur will cause a truncated decay.

ifn - function table number of stored rise shape with extended guard point.

iatss - attenuation factor, by which the last value of the envlpx rise is modified during the note's pseudo "steady state." A factor > l causes an exponential growth, and < l an exponential decay. A 1 will maintain a true steady state at the last rise value. Note that this attenuation is not by fixed rate (as in a piano), but is sensitive to a note's duration. However, if iatss is negative (or if "steady state" < 4 k-periods) a fixed attenuation rate of abs(iatss) per second will be used. 0 is illegal.

iatdec - attenuation factor by which the closing "steady state" value is reduced exponentially over the decay period. This value must be positive and is normally of the order of .01. A large or excessively small value is apt to produce a cutoff which is audible. A zero or neg value is illegal.

ixmod (optional, between +- .9 or so) - exponential curve modifier, influencing the "steepness" of the exponential trajectory during the "steady state." Values less than zero will cause an accelerated growth or decay towards the target (e.g. subito piano). Values greater than zero will cause a retarded growth or decay. The default value is zero (unmodified exponential).

PERFORMANCE

Rise modifications are applied for the first irise seconds, and decay from time idur - idec. If these periods are separated in time there will be a "steady state" during which amp will be unmodified (linen) or modified by the first exponential pattern (envlpx). If linen rise and decay periods overlap then both modifications will be in effect for that time; in envlpx that will cause a truncated decay. If the overall duration idur is exceeded in performance, the final decay will continue on in the same direction, going negative for linen but tending
asymptotically to zero in envlpx.

linenr is unique within Csound in containing a note-off sensor and release time extender. When it senses either a score event termination or a MIDI noteoff, it will immediately extend the performance time of the current instrument by idec seconds, then execute an exponential decay towards the factor iatdec. For two or more units in an instrument, extension is by the greatest idec.

kr **port** ksig, ihtim[, isig]
ar **tone** asig, khp[, istor]
ar **atone** asig, khp[, istor]
ar **reson** asig, kcf, kbw[, iscl, istor]
ar **areson** asig, kcf, kbw[, iscl, istor]

A control or audio signal is modified by a low- or band-pass recursive filter with variable frequency response.

INITIALIZATION

isig - initial (i.e. previous) value for internal feedback. The default value is 0.

istor - initial disposition of internal data space. Since filtering incorporates a feedback loop of previous output, the initial status of the storage space used is significant. A zero value will clear the space; a non-zero value will allow previous information to remain. The default value is 0.

iscl - coded scaling factor for resonators. A value of 1 signifies a peak response factor of 1, i.e. all frequencies other than kcf are attenuated in accordance with the (normalized) response curve. A value of 2 raises the response factor so that its overall RMS value equals 1. (This intended equalization of input and output power assumes all frequencies are physically present; hence it is most applicable to white noise.) A zero value signifies no scaling of the signal, leaving that to some later adjustment (e.g. see balance). The default value is 0.

PERFORMANCE

port applies portamento to a step-valued control signal. At each new step value, ksig is low-pass filtered to move towards that value at a rate determined by ihtim. ihtim is the "half-time" of the function (in seconds), during which the curve will traverse half the distance towards the new value, then half as much again, etc., theoretically never reaching its asymptote.

tone implements a first-order recursive low-pass filter in which the variable khp (in cps) determines the response curve's half-power point. Half power is defined as peak power / root 2.
reson is a second-order filter in which kcf controls the center frequency, or cps position of the peak response, and kbw controls
its bandwidth (the cps difference between the upper and lower half - power points).

atone, **areson** are filters whose transfer functions are the complements of tone and reson. atone is thus a form of high-pass filter and areson a notch filter whose transfer functions represent the "filtered out" aspects of their complements. Note, however, that power scaling is not normalized in atone, areson, but remains the true complement of the corresponding unit. Thus an audio signal, filtered by parallel matching reson and areson units, would under addition simply reconstruct the original spectrum. This property is particularly useful for controlled mixing of different sources (e.g., see lpreson).

Complex response curves such as those with multiple peaks can be obtained by using a bank of suitable filters in series. (The resultant response is the product of the component responses.) In such cases, the combined attenuation may result in a serious loss of signal power, but this can be regained by the use of balance.

krmsr,krmso,kerr,kcps **lpread** ktimpnt, ifilcod[, inpoles][,ifrmrate]
ar **lpreson** asig
ar **lpfreson** asig, kfrqratio

These units, used as a read/reson pair, use a control file of time-varying filter coefficients to dynamically modify the spectrum of an audio signal.

INITIALIZATION

ifilcod - integer or character-string denoting a control-file (reflection coefficients and four parameter values) derived from n-pole linear predictive spectral analysis of a source audio signal. An integer denotes the suffix of a file lp.m; a character-string (in double quotes) gives a filename, optionally a full pathname. If not fullpath, the file is sought first in the current directory, then in that of the environment variable SADIR (if defined). Memory usage depends on the size of the file, which is held entirely in memory during computation but
shared by multiple calls (see also adsyn, pvoc).

inpoles, ifrmrate (optional) - number of poles, and frame rate per second in the lpc analysis. These arguments are required only when the control file does not have a header; they are ignored when a header is detected. The default value for both is zero.

PERFORMANCE

lpread accesses a control file of time-ordered information frames, each containing n-pole filter coefficients derived from linear predictive analysis of a source signal at fixed time intervals (e.g. 1/100 of a second), plus four parameter values:

krmsr - root-mean-square (rms) of the residual of analysis,
krmso - rms of the original signal,
kerr - the normalized error signal,
kcps - pitch in cps.

lpread gets its values from the control file according to the input value ktimpnt (in seconds). If ktimpnt proceeds at the analysis rate, time-normal synthesis will result; proceeding at a faster, slower, or variable rate will result in time-warped synthesis. At each K-period, lpread interpolates between adjacent frames to more accurately determine the parameter values (presented as output) and the filter coefficient settings (passed internally to a subsequent lpreson).

The error signal kerr (between 0 and 1) derived during predictive analysis reflects the deterministic/random nature of the analyzed source. This will emerge low for pitched (periodic) material and higher for noisy material. The transition from voiced to unvoiced speech, for example, produces an error signal value of about .3. During synthesis, the error signal value can be used to determine the nature of the **lpreson** driving function: for example, by arbitrating between pitched and non-pitched input, or even by determining a mix of the two. In normal speech resynthesis, the pitched input to lpreson is a wideband periodic signal or pulse train derived from a unit such as buzz, and the nonpitched source is usually derived from rand. However, any audio signal can be used as the driving function, the only assumption of the analysis being that it has a flat response.

lpfreson is a formant shifted lpreson, in which kfrqratio is the (cps) ratio of shifted to original formant positions. This permits synthesis in which the source object changes its apparent acoustic size. lpfreson with kfrqratio = 1 is equivalent to lpreson.

Generally, lpreson provides a means whereby the time-varying content and spectral shaping of a composite audio signal can be controlled by the dynamic spectral content of another. There can be any number of lpread/lpreson (or lpfreson) pairs in an instrument or in an orchestra; they can read from the same or different control files independently.

kr	**rms**	asig[, ihp, istor]
nr	**gain**	asig, krms[, ihp, istor]
ar	**balance**	asig, acomp[, ihp, istor]

The rms power of asig can be interrogated, set, or adjusted to match that of a comparator signal.

INITIALIZATION

ihp (optional) - half-power point (in cps) of a special internal low-pass filter. The default value is 10.

istor (optional) - initial disposition of internal data space (see reson). The default value is 0.

PERFORMANCE

rms output values kr will trace the rms value of the audio input asig. This unit is not a signal modifier, but functions rather as a signal power-guage.

gain provides an amplitude modification of asig so that the output ar has rms power equal to krms. rms and gain used together (and given matching ihp values) will provide the same effect as balance.

balance outputs a version of asig, amplitude-modified so that its rms power is equal to that of a comparator signal acomp. Thus a signal that has suffered loss of power (eg., in passing through a filter bank) can be restored by matching it with, for instance, its own source. It should be noted that gain and balance provide amplitude modification only - output signals are not altered in any other respect.

Example:

```
asrc buzz      10000,440, sr/440, 1  ; band-limited pulse train
a1  reson      asrc, 1000,100        ; sent through
a2  reson      a1,3000,500           ; 2 filters
afin balance   a2, asrc              ; then balanced with
```

source

kr	**downsamp**	asig[, iwlen]
ar	**upsamp**	ksig
ar	**interp**	ksig[, istor]
kr	**integ**	ksig[, istor]
ar	**integ**	asig[, istor]
kr	**diff**	ksig[, istor]
ar	**diff**	asig[, istor]
kr	**samphold**	xsig, kgate[, ival, ivstor]
ar	**samphold**	asig, xgate[, ival, ivstor]

Modify a signal by up- or down-sampling, integration, and differentiation.

INITIALIZATION

iwlen (optional) - window length in samples over which the audio signal is averaged to determine a downsampled value. Maximum length is ksmps; 0 and 1 imply no window averaging. The default value is 0.

istor (optional) - initial disposition of internal save space (see reson). The default value is 0.

ival, ivstor (optional) - controls initial disposition of internal save space. If ivstor is zero the internal "hold" value is set to ival ; else it retains its previous value. Defaults are 0,0 (i.e. init to zero).

PERFORMANCE

downsamp converts an audio signal to a control signal by downsampling. It produces one kval for each audio control period. The optional window invokes a simple averaging process to suppress foldover.

upsamp, **interp** convert a control signal to an audio signal. The first does it by simple repetition of the kval, the second by linear interpolation between successive kvals. upsamp is a slightly more efficient form of the assignment, `asig = ksig'.

integ, **diff** perform integration and differentiation on an input control signal or audio signal. Each is the converse of the other, and applying both will reconstruct the original signal. Since these units are special cases of low-pass and high-pass filters, they produce a scaled (and phase shifted) output that is frequency-dependent. Thus diff of a sine produces a cosine, with amplitude 2 * sin(pi * cps / sr) that of the original (for each component partial); integ will inversely affect the magnitudes of its component inputs. With this understanding, these units can provide useful signal modification.

samphold performs a sample-and-hold operation on its input according to the value of gate. If gate > 0, the input samples are passed to the output; If gate <= 0, the last output value is repeated. The controlling gate can be a constant, a control signal, or an audio signal.

Example:

```
asrc   buzz      10000,440,20, 1   ; band-limited pulse train
adif   diff      asrc              ; emphasize the highs
anew   balance   adif, asrc        ; but retain the power
agate  reson     asrc,0,440        ; use a lowpass of the original
asamp  samphold  anew, agate       ; to gate the new audiosig
```

```
aout    tone    asamp,100       ; smooth out the rough edges
```

ar **delayr** idlt[, istor]
delayw asig
ar **delay** asig, idlt[, istor]
ar **delay1** asig[, istor]

A signal can be read from or written into a delay path, or it can be automatically delayed by some time interval.

INITIALIZATION

idlt - requested delay time in seconds. This can be as large as available memory will permit. The space required for n seconds of delay is 4n * sr bytes. It is allocated at the time the instrument is first initialized, and returned to the pool at the end of a score section.

istor (optional) - initial disposition of delay-loop data space (see reson). The default value is 0.

PERFORMANCE

delayr reads from an automatically established digital delay line, in which the signal retrieved has been resident for idlt seconds. This unit must be paired with and precede an accompanying delayw unit. Any other Csound statements can intervene.

delayw writes asig into the delay area established by the preceding delayr unit. Viewed as a pair, these two units permit the formation of modified feedback loops, etc. However, there is a lower bound on the value of idlt, which must be at least 1 control period (or 1/kr).

delay is a composite of the above two units, both reading from and writing into its own storage area. It can thus accomplish signal time-shift, although modified feedback is not possible.
There is no minimum delay period.

delay1 is a special form of delay that serves to delay the audio signal asig by just one sample. It is thus functionally equivalent to "delay asig, 1/sr" but is more efficient in both time and space. This unit is particularly useful in the fabrication of generalized non-recursive filters.

Example:

```
tigoto  contin          ; except on a tie,
a2  delay   a1, .05, 0  ; begin 50 msec clean delay of sig
contin:
```

ar **deltap** kdlt
ar **deltapi** xdlt

Tap a delay line at variable offset times.

PERFORMANCE

These units can tap into a delayr/delayw pair, extracting delayed audio from the idlt seconds of stored sound. There can be any number of deltap and/or deltapi units between a read/write pair.
Each receives an audio tap with no change of original amplitude.

deltap extracts sound by reading the stored samples directly; deltapi extracts sound by interpolated readout. By interpolating between adjacent stored samples deltapi represents a particular delay time with more accuracy, but it will take about twice as long to run.

The arguments kdlt, xdlt specify the tapped delay time in seconds. Each can range from 1 Control Period to the full delay time of the read/write pair; however, since there is no internal check for adherence to this range, the user is wholly responsible. Each argument can be a constant, a variable, or a time-varying signal; the xdlt argument in deltapi implies that an audio-varying delay is permitted there.

These units can provide multiple delay taps for arbitrary delay path and feedback networks. They can deliver either constant-time or time-varying taps, and are useful for building chorus effects, harmonizers, and doppler shifts. Constant-time delay taps (and some slowly changing ones) do not need interpolated readout; they are well served by deltap.
Medium-paced or fast varying dlt's, however, will need the extra services of deltapi.

N.B. K-rate delay times are not internally interpolated, but rather lay down stepped time-shifts of audio samples; this will be found quite adequate for slowly changing tap times. For medium to fastpaced changes, however, one should provide a higher resolution audio-rate timeshift as input.

Example:

```
asource buzz    1, 440, 20, 1
atime   linseg  1, p3/2,.01, p3/2,1  ; trace a distance in secs
ampfac  =   1/atime/atime            ; and calc an amp factor
adump   delayr  1                    ; set maximum distance
amove   deltapi atime                ; move sound source past
delays  asource                      ; the listener
        out amove * ampfac
```

ar **comb** asig, krvt, ilpt[, istor]
ar **alpass** asig, krvt, ilpt[, istor]
ar **reverb** asig, krvt[, istor]

An input signal is reverberated for krvt seconds with "colored" (comb), flat (alpass), or "natural room" (reverb) frequency response.

INITIALIZATION

ilpt - loop time in seconds, which determines the "echo density" of the reverberation. This in turn characterizes the "color" of the comb filter whose frequency response curve will contain ilpt * sr/2 peaks spaced evenly between 0 and sr/2 (the Nyquist frequency). Loop time can be as large as available memory will permit. The space required for an n second loop is 4n * sr bytes. comb and alpass delay space is allocated and returned as in delay.

istor (optional) - initial disposition of delay-loop data space (cf. reson). The default value is 0.

PERFORMANCE

These filters reiterate input with an echo density determined by loop time ilpt. The attenuation rate is independent and is determined by krvt, the reverberation time (defined as the time in seconds for a signal to decay to 1/1000, or 60dB down from its original amplitude). Output from a comb filter will appear only after ilpt seconds; alpass output will begin to appear immediately.

A standard **reverb** unit is composed of four **comb** filters in parallel followed by two **alpass** units in series. Looptimes are set for optimal "natural room response." Core storage requirements for this unit are proportional only to the sampling rate, each unit requiring approximately 3K words for every 10KC.
The comb, alpass, delay, tone and other Csound units provide the means for experimenting with alternate reverberator designs.

Since output from the standard reverb will begin to appear only after 1/20 second or so of delay, and often with less than three-fourths of the original power, it is normal to output both the source and the reverberated signal. Also, since the reverberated sound will persist long after the cessation of source events, it is normal to put reverb in a separate instrument to which sound is passed via a global variable, and to leave that instrument running throughout the performance.

Example:

```
ga1  init 0          ; init an audio receiver/mixer

instr    1           ; instr (there may be many copies)
a1   oscili  8000, cpspch(p5), 1 ; generate a source signal
     out  a1         ; output the direct sound
ga1  =    ga1 + a1 ; and add to audio receiver
     endin

 instr    99              ; (highest instr number executed last)
a3   reverb    ga1, 1.5  ; reverberate whatever is in ga1
     out  a1             ; and output the result
ga1  =    0              ; empty the receiver for the next pass
     endin
```

OPERATIONS USING SPECTRAL DATA-TYPES

These units generate and process non-standard signal data types, such as down-sampled time-domain control signals and audio signals, and their frequency-domain (spectral) representations. The new data types (d-, w-) are self-defining, and the contents are not processable by any other Csound units. These unit generators are experimental, and subject to change between releases; they will also be joined by others later.

dsig **octdown** xsig, iocts, isamps[, idisprd]
wsig **noctdft** dsig, iprd, ifrqs, iq[, ihann, idbout, idsines]

INITIALIZATION

idisprd (optional) - if non-zero, display the output every idisprd seconds. The default value is 0 (no display).

ihann, idbout, idsines (optional) - if non-zero, then respectively: apply a hanning window to the input; convert the output magnitudes to dB; display the windowed sinusoids used in DFT filtering. The default values are 0,0,0 (rectangular window, magnitude output, no sinusoid display).

PERFORMANCE

octdown - put signal asig or ksig through iocts successive applications of octave decimation and downsampling, and preserve isamps down-sampled values in each octave. Optionally display the composite buffer every idisprd seconds.

noctdft - generate a constant-Q, exponentially-spaced DFT across all octaves of the multiply-downsampled input dsig. Every iprd seconds, each octave of dsig is optionally windowed (ihann non-zero), filtered (using ifrqs parallel filters per octave, exponentially spaced, and with frequency/bandwidth Q of iq), and the output magnitudes optionally converted to dB (idbout non-zero). This unit produces a self-defining spectral datablock wsig, whose characteristics are readable by any units that receive it as input, and for which it becomes the template for output. The information used in producing this wsig (iprd, iocts, ifrqs) is passed via the data block to all derivative wsigs, and is thus available to subsequent spectral operators if needed.

Example:

```
asig    in                              ; get external audio
dsamp octdown asig, 6, 180, 0           ; downsample in 6 octaves
wsig1 noctdft   dsamp, .02, 12, 33, 0, 1, 1; and calc 72-point dft (dB)
```

wsig	**specaddm**	wsig1, wsig2[, imul2]
wsig	**specdiff**	wsigin
wsig	**specscal**	wsigin, ifscale, ifthresh
wsig	**spechist**	wsigin
wsig	**specfilt**	wsigin, ifhtim

INITIALIZATION

imul2 (optional) - if non-zero, scale the wsig2 magnitudes before adding. The default value is 0.

PERFORMANCE

specaddm - do a weighted add of two input spectra. For each channel of the two input spectra, the two magnitudes are combined and written to the output according to: magout = mag1in + mag2in * imul2. The operation is performed whenever the input wsig1 is sensed to be new. This unit will (at Initialization) verify the consistency of the two spectra (equal size, equal period, equal mag types).

specdiff - find the positive difference values between consecutive spectral frames. At each new frame of wsigin, each magnitude value is compared with its predecessor, and the positive changes written to the output spectrum. This unit is useful as an energy onset detector.

specscal - scale an input spectral datablock with spectral envelopes. Function tables ifthresh and ifscale are initially sampled across the (logarithmic) frequency space of the input spectrum; then each time a new input spectrum is sensed the sampled values are used to scale each of its magnitude channels as follows: if ifthresh is non-zero, each magnitude is reduced by its corresponding table-value (to not less than zero); then each magnitude is rescaled by the corresponding ifscale value, and the resulting spectrum written to wsig.

spechist - accumulate the values of successive spectral frames. At each new frame of wsigin, the accumulations-to-date in each magnitude track are written to the output spectrum. This unit thus provides a running histogram of spectral distribution.

specfilt - filter each channel of an input spectrum. At each new frame of wsigin, each magnitude value is injected into a 1st-order lowpass recursive filter, whose half-time constant has been initially set by sampling the ftable ifhtim across the (logarithmic) frequency space of the input spectrum. This unit effectively applies a persistence factor to the data occurring in each spectral channel, and is useful for simulating the energy integration that occurs during auditory perception. It may also be used as a time-attenuated running histogram of the spectral distribution.

Example:

```
wsig2    specdiff  wsig1        ; sense onsets
wsig3    specfilt  wsig2, 2     ; absorb slowly
specdisp  wsig2, .1             ; & display both spectra
specdisp  wsig3, .1
```

koct **specptrk** wsig, inptls, irolloff, iodd[, interp, ifprd, iwtflg]
ksum **specsum** wsig[, interp]
specdisp wsig, iprd[, iwtflg]

INITIALIZATION

interp (optional) - if non-zero, interpolate the output signal (koct or ksum). The default value is 0 (repeat the signal value between changes).

ifprd (optional) - if non-zero, display the internally computed fundamental spectrum. The default value is 0 (no display).

iwtflg (optional) - wait flag. If non-zero, hold each display until released by the user. The default value is 0 (no wait).

PERFORMANCE

specptrk - estimate the pitch of the most prominent complex tone in the spectrum. At note initialization this unit creates a set of inptls harmonically related partials (odd if iodd non-zero) with amplitude rolloff to the fraction irolloff per octave. Then at each new frame of wsig, the spectrum is cross-correlated with this set, and the result at

each point added to an internal copy of the spectrum (optionally displayed). A pitch is then estimated, and the result is released in decimal octave form.
Between frames, the output is either repeated or interpolated at the K-rate.

specsum - sum the magnitudes across all channels of the spectrum.
At each new frame of wsig, the magnitudes are summed and released as a scalar ksum signal. Between frames, the output is either repeated or interpolated at the K-rate. This unit produces a k-signal summation of the magnitudes present in the spectral data, and is thereby a running measure of its moment-to-moment overall strength.

specdisp - display the magnitude values of spectrum wsig every iprd seconds (rounded to some integral number of wsig's originating iprd).

Example:

```
   ksum  specsum   wsig,  1                ; sum the spec bins, and
ksmooth
   if      ksum < 2000  kgoto  zero  ; if sufficient amplitude
   koct  specptrk  wsig              ; pitch-track the signal
   kgoto contin
   zero:  koct =    0                ; else output zero
   contin:
```

SENSING & CONTROL

Ktemp **tempest** kin, iprd, imindur, imemdur, ihp, ithresh, ihtim, ixfdbak, istartempo, ifn[, idisprd, itweek]

Estimate the tempo of beat patterns in a control signal.

INITIALIZATION

iprd - period between analyses (in seconds). Typically about .02 seconds.

imindur - minimum duration (in seconds) to serve as a unit of tempo. Typically about .2 seconds.

imemdur - duration (in seconds) of the kin short-term memory buffer which will be scanned for periodic patterns. Typically about 3 seconds.

ihp - half-power point (in cps) of a low-pass filter used to smooth input kin prior to other processing. This will tend to suppress activity that moves much faster. Typically 2 cps.

ithresh - loudness threshold by which the low-passed kin is center-clipped before being placed in the short-term buffer as tempo-relevant data. Typically at the noise floor of the incoming data.

ihtim - half-time (in seconds) of an internal forward-masking filter that masks new kin data in the presence of recent, louder data. Typically about .005 seconds.

ixfdbak - proportion of this unit's anticipated value to be mixed with the incoming kin prior to all processing. Typically about .3.

istartempo - initial tempo (in beats per minute). Typically 60.

ifn - table number of a stored function (drawn left-to-right) by which the short-term memory data is attenuated over time.

idisprd (optional) - if non-zero, display the short-term past and future buffers every idisprd seconds (normally a multiple of iprd). The default value is 0 (no display).

itweek (optional) - fine-tune adjust this unit so that it is stable when analyzing events controlled by its own output. The default value is 1 (no change).

PERFORMANCE

tempest examines kin for amplitude periodicity, and estimates a current tempo. The input is first low-pass filtered, then center-clipped, and the residue placed in a short-term memory buffer (attenuated over time) where it is analyzed for periodicity using a form of autocorrelation. The period, expressed as a tempo in beats per minute, is output as ktemp.
The period is also used internally to make predictions about future amplitude patterns, and these are placed in a buffer adjacent to that of the input. The two adjacent buffers can be periodically displayed, and the predicted values optionally mixed with the incoming signal to simulate expectation.

This unit is useful for sensing the metric implications of any k-signal (e.g- the RMS of an audio signal, or the second derivative of a conducting gesture), before sending to a tempo statement.

Example:

```
ksum specsum  wsignal, 1          ; sum the amps of a spectrum
ktemp  tempest  ksum, .02, .1, 3, 2, 800, .005, 0, 0, 4, .1, .995 ; and
                                  ;look for beats
```

kx, ky **xyin** iprd, ixmin, ixmax, iymin, iymax[, ixinit, iyinit]
tempo ktempo, istartempo

Sense the cursor position in an input window. Apply tempo control to an uninterpreted score.

INITIALIZATION

iprd - period of cursor sensing (in seconds). Typically .1 seconds.

xmin, xmax, ymin, ymax - edge values for the x-y coordinates of a cursor in the input window.

ixinit, iyinit (optional) - initial x-y coordinates reported; the default values are 0,0. If these values are not within the given min-max range, they will be coerced into that range.

istartempo - initial tempo (in beats per minute). Typically 60.

PERFORMANCE

xyin samples the cursor x-y position in an input window every iprd seconds. Output values are repeated (not interpolated) at the K-rate, and remain fixed until a new change is registered in the window. There may be any number of input windows. This unit is useful for Realtime control, but continuous motion should be avoided if iprd is unusually small.

tempo allows the performance speed of Csound scored events to be controlled from within an orchestra. It operates only in the presence of the csound -t flag. When that flag is set, scored events will be performed from their uninterpreted p2 and p3 (beat) parameters, initially at the given command-line tempo.
When a tempo statement is activated in any instrument (ktempo > 0.), the operating tempo will be adjusted to ktempo beats per minute. There may be any number of tempo statements in an orchestra, but coincident activation is best avoided.

Example:

```
      kx,ky    xyin .05, 30, 0, 120, 0, 75  ; sample the cursor
      tempo    kx,  75                      ; and control the tempo of
performance
```

SOUND INPUT & OUTPUT

a1 **in**
a1, a2 **ins**
a1, a2, a3, a4 **inq**
a1 **soundin** ifilcod[, iskptim][, iformat]
a1, a2 **soundin** ifilcod[, iskptim][, iformat]
a1, a2, a3, a4 **soundin** ifilcod[, iskptim][, iformat]
out asig
outs1 asig
outs2 asig
outs asig1, asig2
outq1 asig
outq2 asig
outq3 asig
outq4 asig
outq asig1, asig2, asig3, asig4

These units read/write audio data from/to an external device or stream.

INITIALIZATION

filcod - integer or character-string denoting the source soundfile name. An integer denotes the file soundin.filcod ; a character-string (in double quotes, spaces permitted) gives the filename itself, optionally a full pathname. If not a full path, the named file is sought first in the current directory, then in that given by the environment variable SSDIR (if defined) then by SFDIR. See also GEN01.

iskptim (optional) - time in seconds of input sound to be skipped. The default value is 0.

iformat (optional) - specifies the audio data file format (1 = 8-bit signed char, 2 = 8-bit A-law bytes, 3 = 8-bit U-law bytes, 4 = 16-bit short integers, 5 = 32bit long integers, 6 = 32-bit floats). If iformat = 0 it is taken from the soundfile header, and if no header from the csound -o command flag. The default value is 0.

PERFORMANCE

in, **ins**, **inq** - copy the current values from the standard audio input buffer. If the command-line flag -i is set, sound is read continuously from the audio input stream (e.g. stdin or a soundfile) into an internal buffer. Any number of these units can read freely from this buffer.

soundin is functionally an audio generator that derives its signal from a pre-existing file. The number of channels read in is set by the number of result cells, a1, a2, etc. A soundin unit opens this file whenever the host instrument is initialized, then closes it again each time the instrument is turned off.
There can be any number of soundin units within a single instrument or orchestra; also, two or more of them can read simultaneously from the same external file.

out, **outs**, **outq** send audio samples to an accumulating output buffer (created at the beginning of performance) which serves to collect the output of all active instruments before the sound is written to disk. There can be any number of these output units in an instrument. The type (mono, stereo, or quad) must agree with nchnls, but units can be chosen to direct sound to any particular channel: outs1 sends to stereo channel 1, outq3 to quad channel 3, etc.

a1, a2, a3, a4 **pan** asig, kx, ky, ifn[, imode][, ioffset]

Distribute an audio signal amongst four channels with localization control.

INITIALIZATION

ifn - function table number of a stored pattern describing the amplitude growth in a speaker channel as sound moves towards it from an adjacent speaker. Requires extended guard-point.

imode (optional) - mode of the kx, ky position values. 0 signifies raw index mode, 1 means the inputs are normalized (0 - 1). The default value is 0.

ioffset (optional) - offset indicator for kx, ky. 0 infers the origin to be at channel 3 (left rear); 1 requests an axis shift to the quadraphonic center. The default value is 0.

PERFORMANCE

pan takes an input signal asig and distributes it amongst four outputs (essentially quad speakers) according to the controls kx and ky. For normalized input (mode=1) and no offset, the four output locations are in order: left-front at (0,1), right-front at (1,1), left-rear at the origin (0,0), and right-rear at (1,0).
In the notation (kx, ky), the coordinates kx and ky, each ranging 0 - 1, thus control the 'rightness' and 'forwardness' of a sound location.

Movement between speakers is by amplitude variation, controlled by the stored function table ifn. As kx goes from 0 to 1, the strength of the right-hand signals will grow from the left-most table value to the right-most, while that of the left-hand signals will progress from the right-most table value to the left-most. For a simple linear pan, the table might contain the linear function 0 - 1. A more correct pan that maintains constant power would be obtained by storing the first quadrant of a sinusoid. Since pan will scale and truncate kx and ky in simple table lookup, a medium-large table (say 8193) should be used. kx, ky values are not restricted to 0 - 1. A circular motion passing through all four speakers (enscribed) would have a diameter of root 2, and might be defined by a circle of radius R = root 1/2 with center at (.5,.5). kx, ky would then come from Rcos(angle), Rsin(angle), with an implicit origin at (.5,.5) (i.e. ioffset = 1). Unscaled raw values operate similarly.
Sounds can thus be located anywhere in the polar or cartesian plane; points lying outside the speaker square are projected correctly onto the square's perimeter as for a listener at the center.

Example:

```
instr 1
k1 phasor 1/p3                  ; fraction of circle
k2 tablei k1, 1, 1              ; sin of angle (sinusoid in f1)
k3 tablei k1, 1, 1, .25, 1      ; cos of angle (sin offset 1/4 circle)
a1 oscili 10000,440, 1          ; audio signal..
a1,a2,a3,a4 pan a1, k2/2, k3/2, 2, 1,1;sent in a circle (f2=1st quad sin)
      outq a1, a2, a3, a4
       endin
```

SIGNAL DISPLAY

print iarg[, iarg,...]
display xsig, iprd[, iwtflg]
dispfft xsig, iprd, iwsiz[, iwtyp][, idbouti][, iwtflg]

These units will print orchestra Init-values, or produce graphic display of orchestra control signals and audio signals. Uses X11 windows if enabled, else (or if -g flag is set) displays are approximated in ascii characters.

INITlALIZATION

iprd - the period of display in seconds.

iwsiz - size of the input window in samples. A window of iwsiz points will produce a Fourier transform of iwsiz/2 points, spread linearly in frequency from 0 to sr/2. iwsiz must be a power of 2. The windows are permitted to overlap.

iwtyp (optional) - window type. 0 = rectangular, 1 = hanning.
The default value is 0 (rectangular).

idbout (optional) - units of output for the Fourier coefficients.
0 = magnitude, 1 = decibels. The default is 0 (magnitude).

iwtflg (optional) - wait flag. If non-zero, each display is held until released by the user. The default value is 0 (no wait).

PERFORMANCE

print - print the current value of the I-time arguments (or expressions) iarg at every I-pass through the instrument.

display - display the audio or control signal xsig every iprd seconds, as an amplitude vs. time graph.

dispfft - display the Fourier Transform of an audio or control signal (asig or ksig) every iprd seconds using the Fast Fourier Transform method.

Example:

```
k1  envlpx   l, .03, p3, .05, l, .5, .0l   ; generate a note envelope
    display  k1, p3                        ; and display entire shape
```

3. THE STANDARD NUMERIC SCORE

A score is a data file that provides information to an orchestra about its performance. Like an orchestra file, a score file is made up of statements in a known format. The Csound orchestra expects to be handed a score comprised mainly of ascii numeric characters. Although most users will prefer a higher level score language such as provided by Cscore, Scot, or another score-generating program, each resulting score must eventually appear in the format expected by the orchestra. A Standard Numeric Score can be created and edited directly by the beginner using standard text editors; indeed, some users continue to prefer it. The purpose of this section is to describe this
format in detail.

The basic format of a standard numeric score statement is:

```
opcode p1 p2 p3 p4...          ;comments
```

The opcode is a single character, always alphabetic. Legal opcodes are f, i, a, t, s, and e, the meanings of which are described in the following pages. The opcode is normally the first character of a line; leading spaces or tabs will be ignored. Spaces or tabs between the opcode and p1 are optional.

p1, p2, p3, etc... are parameter fields (pfields). Each contains a floating point number comprised of an optional sign, digits, and an optional decimal point. Expressions are not permitted in Standard Score files. pfields are separated from each other by one or more spaces or tabs, all but one space of which will be ignored.

Continuation lines are permitted. If the first printing character of a new scoreline is not an opcode, that line will be regarded as a continuation of the pfields from the previous scoreline.

Comments are optional and are for the purpose of permitting the user to document his score file. Comments always begin with a semicolon (;) and extend to the end of the line. Comments will not affect the pfield continuation feature.

Blank lines or comment-only lines are legal (and will be ignored).

Preprocessing of Standard Scores

A Score (a collection of score statements) is divided into time-ordered sections by the s statement. Before being read by the orchestra, a score is preprocessed one section at a time.
Each section is normally processed by 3 routines: Carry, Tempo, and Sort.

1. **Carry** - within a group of consecutive i statements whose p1 whole numbers correspond, any pfield left empty will take its value from the same pfield of the preceding statement. An empty pfield can be enoted by a single point (.) delimited by spaces.
No point is required after the last nonempty pfield. The output of Carry preprocessing will show the carried values explicitly.

The Carry Feature is not affected by intervening comments or blank lines; it is turned off only by a non-i statement or by an i statement with unlike p1 whole number.

An additional feature is available for p2 alone. The symbol + in p2 will be given the value of p2 + p3 from the preceding i statement. This enables note action times to be automatically determined from the sum of preceding durations. The + symbol can itself be carried. It is legal only in p2.

E.g.: the statements

```
i1  0   .5      100
i .  +
i
```

will result in

```
i1  0   .5      100
i1  .5  .5      100
i1  1   .5      100
```

The Carry feature should be used liberally. Its use, especially in large scores, can greatly reduce input typing and will simplify later changes.

2. **Tempo** - this operation time warps a score section according to the information in a t statement. The tempo operation converts p2 (and, for i statements, p3) from original beats into real seconds, since those are the units required by the orchestra. After time warping, score files will be seen to have orchestra-readable format demonstrated by the following:

```
i p1 p2beats p2seconds p3beats p3seconds p4 p5 ....
```

3. **Sort** - this routine sorts all action-time statements into chronological order by p2 value. It also sorts coincident events into precedence order. Whenever an f statement and an i statement have the same p2 value, the f statement will precede.
Whenever two or more i statements have the same p2 value, they will be sorted into ascending p1 value order. If they also have the same p1 value, they will be sorted into ascending p3 value order. Score sorting is done section by section (see s statement). Automatic sorting implies that score statements may appear in any order within a section.

N.B. The operations Carry, Tempo and Sort are combined in a
3-phase single pass over a score file, to produce a new file in orchestra-readable format (see the Tempo example). Processing can be invoked either explicitly by the scsort command, or implicitly by csound which processes the score before calling the orchestra. Source-format files and orchestra-readable files are both in ascii character form, and may be either perused or further modified by standard text editors. Userwritten routines can be used to modify score files before or after the above processes, provided the final orchestra-readable statement format is not violated. Sections of different formats can be sequentially batched; and sections of like format can be merged for automatic sorting.

Next-P and Previous-P Symbols

At the close of any of the above operations, three additional score features are interpreted during file writeout: next-p, previous-p, and ramping.

i statement pfields containing the symbols npx or ppx (where x is some integer) will be replaced by the appropriate pfield value found on the next i statement (or previous i statement) that has the same p1. For example, the symbol np7 will be replaced by the value found in p7 of the next note that is to be played by this instrument. np and pp symbols are recursive and can reference other np and pp symbols which can reference others, etc.
References must eventually terminate in a real number or a ramp symbol (see below). Closed loop references should be avoided.
np and pp symbols are illegal in p1,p2 and p3 (although they may reference these). np and pp symbols may be Carried. np and pp references cannot cross a Section boundary. Any forward or
backward reference to a non-existent note-statement will be given
the value zero.

E.g.: the statements

```
i1  0  1  10  np4  pp5
i1  1  1  20
i1  1  1  30
```

will result in

```
i1  0  1  10  20  0
i1  1  1  20  30  20
i1  2  1  30  0   30
```

np and pp symbols can provide an instrument with contextual knowledge of the score, enabling it to glissando or crescendo, for instance, toward the pitch or dynamic of some future event (which may or may not be immediately adjacent). Note that while the Carry feature will propagate np and pp through unsorted statements, the operation that interprets these symbols is acting on a time-warped and fully sorted version of the score.

Ramping

i statement pfields containing the symbol < will be replaced by values derived from linear interpolation of a time-based ramp. Ramps are anchored at each end by the first real number found in the same pfield of a preceding and following note played by the same instrument.

E.g.: the statements

```
i1  0  1  100
i1  1  1  <
i1  2  1  <
i1  3  1  400
i1  4  1  <
i1  5  1  0
```

will result in

```
i1  0  1  100
i1  1  1  200
i1  2  1  300
i1  3  1  400
i1  4  1  200
i1  5  1  0
```

Ramps cannot cross a Section boundary. Ramps cannot be anchored by an np or pp symbol (although they may be referenced by these). Ramp symbols are illegal in p1, p2 and p3. Ramp symbols may be Carried. Note, however, that while the Carry feature will propagate ramp symbols through unsorted statements, the operation that interprets these symbols is acting on a time-warped and fully sorted version of the score. In fact, time-based linear interpolation is based on warped score-time, so that a ramp which spans a group of accelerating notes will remain linear with respect to strict chronological time.

F STATEMENT (or FUNCTION TABLE STATEMENT)

f p1 p2 p3 p4 ...

This causes a GEN subroutine to place values in a stored function table for use by instruments.

PFIELDS

p1 Table number (from 1 to 200) by which the stored function will be known.
A negative number requests that the table be destroyed.

p2 Action time of function generation (or destruction) in beats.

p3 Size of function table (i.e. number of points).
Must be a power of 2, or a power-of-2 plus 1 (see below).
Maximum table size is 16777216 (2**24) points.

p4 Number of the GEN routine to be called (see GEN ROUTINES).
A negative value will cause rescaling to be omitted.

p5 |
p6 | Parameters whose meaning is determined by the particular GEN routine.
. |
. |

SPECIAL CONSIDERATIONS

Function tables are arrays of floating-point values. Arrays can be of any length in powers of 2; space allocation always provides for 2**n points plus an additional guard point. The guard point value, used during interpolated lookup, can be automatically set to reflect the table's purpose: If size is an exact power of 2, the guard point will be a copy of the first point; this is appropriate for interpolated wrap-around lookup as in oscili, etc., and should even be used for non-interpolating oscil for safe consistency. If size is set to 2**n + 1, the guard point value automatically extends the contour of table values; this is appropriate for single-scan functions such in envlpx, oscil1, oscil1i, etc.

Table space is allocated in primary memory, along with instrument data space. The maximum table number has a soft limit of 200; this can be extended if required.

An existing function table can be removed by an f statement containing a negative p1 and an appropriate action time. A function table can also be removed by the generation of another table with the same p1. Functions are not automatically erased at the end of a score section.

p2 action time is treated in the same way as in i statements with respect to sorting and modification by t statements. If an f statement and an i statement have the same p2, the sorter gives the f statement precedence so that the function table will be available during note initialization.

An f 0 statement (zero p1, positive p2) may be used to create an action time with no associated action. Such time markers are useful for padding out a score section (see s statement).

I STATEMENT (INSTRUMENT or NOTE STATEMENT)

i p1 p2 p3 p4 ...

This statement calls for an instrument to be made active at a specific time and for a certain duration. The parameter field values are passed to that instrument prior to its initialization, and remain valid throughout its Performance.

PFIELDS

p1 Instrument number (from 1 to 200), usually a non-negative integer.
An optional fractional part can provide an additional tag for specifying ties between particular notes of consecutive clusters.
A negative p1 (including tag) can be used to turn off a particular `held' note.

p2 Starting time in arbitrary units called beats.

p3 Duration time in beats (usually positive). A negative value will initiate a held note (see also ihold). A zero value will invoke an initialization pass without performance (see also instr).

p4 |
p5 | Parameters whose significance is determined by the instrument.
. |
. |

SPECIAL CONSIDERATIONS

Beats are evaluated as seconds, unless there is a t statement in this score section or a -t flag in the command line.

Starting or action times are relative to the beginning of a section (see s statement), which is assigned time 0.

Note statements within a section may be placed in any order.
Before being sent to an orchestra, unordered score statements must first be processed by Sorter, which will reorder them by ascending p2 value. Notes with the same p2 value will be ordered by ascending p1; if the same p1, then by ascending p3.

Notes may be stacked, i.e., a single instrument can perform any number of notes simultaneously. (The necessary copies of the instrument's data space will be allocated dynamically by the orchestra loader.) Each note will normally turn off when its p3 duration has expired, or on receipt of a MIDI noteoff signal. An instrument can modify its own duration either by changing its p3 value during note initialization, or by prolonging itself through the action of a linenr unit.

An instrument may be turned on and left to perform indefinitely either by giving it a negative p3 or by including an ihold in its I-time code. If a held note is active, an i statement with matching p1 will not cause a new allocation but will take over the data space of the held note. The new pfields (including p3) will now be in effect, and an I-time pass will be executed in which the units can either be newly initialized or allowed to continue as required for a tied note (see tigoto). A held note may be succeeded either by another held note or by a note of
finite duration. A held note will continue to perform across section endings (see s statement). It is halted only by turnoff or by an i statement with negative matching p1 or by an e statement.

A STATEMENT (or ADVANCE STATEMENT)

a p1 p2 p3

This causes score time to be advanced by a specified amount without producing sound samples.

PFIELDS

p1 carries no meaning. Usually zero
p2 Action time, in beats, at which advance is to begin.
p3 Durational span (distance in beats) of time advance.

p4 |
p5 | These carry no meaning.
.

SPECIAL CONSIDERATIONS

This statement allows the beat count within a score section to be advanced without generating intervening sound samples. This can be of use when a score section is incomplete (the beginning or middle is missing) and the user does not wish to generate and listen to a lot of silence.

p2 action time and p3 distance are treated as in i statements, with respect to sorting and modification by t statements.

An a statement will be temporarily inserted in the score by the Score Extract feature when the extracted segment begins later than the start of a Section. The purpose of this is to preserve the beat count and time count of the original score for the benefit of the peak amplitude messages which are reported on the user console.

Whenever an a statement is encountered by a performing orchestra, its presence and effect will be reported on the user's console.

T STATEMENT (TEMPO STATEMENT)

t p1 p2 p3 p4 ..... (unlimited)

This statement sets the tempo and specifies the accelerations and decelerations for the current section. This is done by converting beats into seconds.

PFIELDS

p1 must be zero
p2 initial tempo in beats per minute
p3, p5, p7, ... times in beats (in non-decreasing order)
p4, p6, p8, ... tempi for the referenced beat times

SPECIAL CONSIDERATIONS

Time and Tempo-for-that-time are given as ordered couples that define points on a "tempo vs time" graph. (The time-axis here is in beats so is not necessarily linear.) The beat-rate of a Section can be thought of as a movement from point to point on that graph: motion between two points of equal height signifies constant tempo, while motion between two points of unequal height will cause an accelarando or ritardando accordingly. The graph can contain discontinuities: two points given equal times but different tempi will cause an immediate tempo change.

Motion between different tempos over non-zero time is inverse linear. That is, an accelerando between two tempos M1 and M2 proceeds by linear interpolation of the single-beat durations from 60/M1 to 60/M2.

The first tempo given must be for beat 0.

A tempo, once assigned, will remain in effect from that time-point unless influenced by a succeeding tempo, i.e. the last specified tempo will be held to the end of the section.

A t statement applies only to the score section in which it appears. Only one t statement is meaningful in a section; it can be placed anywhere within that section. If a score section contains no t statement, then beats are interpreted as seconds (i.e. with an implicit t 0 60 statement).

N.B. If the csound command includes a -t flag, the interpreted tempo of all score t statements will be overridden by the command-line tempo.

S STATEMENT

s anything

The s statement marks the end of a section.

PFIELDS

All pfields are ignored.

SPECIAL CONSIDERATIONS

Sorting of the i, f and a statements by action time is done section by section.

Time warping for the t statement is done section by section.

All action times within a section are relative to its beginning.
A section statement establishes a new relative time of 0, but has no other reinitializing effects (e.g. stored function tables are preserved across section boundaries).

A section is considered complete when all action times and finite durations have been satisfied (i.e., the "length" of a section is determined by the last occurring action or turn-off). A section can be extended by the use of an f 0 statement.

A section ending automatically invokes a Purge of inactive instrument and data spaces.

N.B. Since score statements are processed section by section, the amount of memory required depends on the maximum number of score statements in a section. Memory allocation is dynamic, and the user will be informed as extra memory blocks are requested during score processing.

For the end of the final section of a score, the s statement is optional; the e statement may be used instead.

E STATEMENT

e anything

This statement may be used to mark the end of the last section of the score.

PFIELDS

All pfields are ignored.

SPECIAL CONSIDERATIONS

The e statement is contextually identical to an s statement. Additionally, the e statement terminates all signal generation (including indefinite performance) and closes all input and output files.

If an e statement occurs before the end of a score, all subsequent score lines will be ignored.

The e statement is optional in a score file yet to be sorted. If a score file has no e statement, then Sort processing will supply one.

4. GEN ROUTINES

The GEN subroutines are function-drawing procedures called by f statements to construct stored wavetables. They are available throughout orchestra performance, and can be invoked at any point in the score as given by p2. p1 assigns a table number, and p3 the table size (see f statement). p4 specifies the GEN routine to be called; each GEN routine will assign special meaning to the pfield values that follow.

GEN01

This subroutine transfers data from a soundfile into a function table.

f # time size 1 filcod skiptime format

size - number of points in the table. Ordinarily a power of 2 or a power-of-2 plus 1 (see f statement); the maximum tablesize is 16777216 (2**24) points. If the source soundfile is of type AIFF, allocation of table memory can be deferred by setting this parameter to 0; the size allocated is then the number of points in the file (probably not a power-of-2), and the table is not usable by normal oscillators, but it is usable by a loscil unit.
An AIFF source can also be mono or stereo.

filcod - integer or character-string denoting the source soundfile name. An integer denotes the file soundin.filcod ; a character-string (in double quotes, spaces permitted) gives the filename itself, optionally a full pathname. If not a full path, the file is sought first in the current directory, then in that given by the environment variable SSDIR (if defined) then by SFDIR. See also soundin.

skiptime - begin reading at skiptime seconds into the file.

format - specifies the audio data-file format:

1- 8-bit signed character	4 - 16-bit short integers
2- 8-bit A-law bytes	5 - 32-bit long integers
3- 8-bit U-law bytes	6 - 32-bit floats

If format = 0 the sample format is taken from the soundfile header, or by default from the csound -o command flag.

Reading stops at end-of-file or when the table is full. Table locations not filled will contain zeros.

Note:

If p4 is positive, the table will be post-normalized (rescaled to a maximum absolute value of 1 after generation). A negative p4 will cause rescaling to be skipped.

Examples:

f 1 0 8192 1 23 0 4
f 2 0 0 -1 “trumpet A#5” 0 4

The tables are filled from 2 files, “soundin.23” and “trumpet A#5”, expected in SSDIR or SFDIR. The first table is pre-allocated; the second is allocated dynamically, and its rescaling is inhibited.

GEN02

This subroutine transfers data from immediate pfields into a function table.

f # time size 2 v1 v2 v3 . . .

size - number of points in the table. Must be a power of 2 or a power-of-2 plus 1 (see f statement). The maximum tablesize is 16777216 (2**24) points.

v1, v2, v3, ... - values to be copied directly into the table space. The number of values is limited by the compile-time variable PMAX, which controls the maximum pfields (currently 150). The values copied may include the table guard point; any table locations not filled will contain zeros.

Note:

If p4 is positive, the table will be post-normalized (rescaled to a maximum absolute value of 1 after generation). A negative p4 will cause rescaling to be skipped.

Example:

f 1 0 16 -2 0 1 2 3 4 5 6 7 8 9 10 11 0

This calls upon GEN02 to place 12 values plus an explicit wrap-around guard value into a table of size next-highest power of 2. Rescaling is inhibited.

GEN03

This subroutine generates a stored function table by evaluating a polynomial in x over a fixed inter- val and with specified coefficients.

f # time size 3 xval1 xval2 c0 c1 c2 . . . cn

size - number of points in the table. Must be a power of 2 or a power-of-2 plus 1 (see f statement).

xval1, xval2 - left and right values of the x interval over which the polynomial is defined (xval1 < xval2). These will produce the 1st stored value and the (power-of-2 plus l)th stored value respectively in the generated function table.

c0, c1, c2, ... cn - coefficients of the nth-order polynomial

c0 + c1x + c2x2 + . . . + cnxn

Coefficients may be positive or negative real numbers; a zero denotes a missing term in the polynomial. The coefficient list begins in p7, providing a current upper limit of 144 terms.
Note:

The defined segment [fn(xval1),fn(xval2)] is evenly distributed.

Thus a 512-point table over the interval [-1,1] will have its origin at location 257 (at the start of the 2nd half). Provided the extended guard point is requested, both fn(-1) and fn(1) will exist in the table.

GEN03 is useful in conjunction with table or tablei for audio waveshaping (sound modification by non-linear distortion).
Coefficients to produce a particular formant from a sinusoidal lookup index of known amplitude can be determined at preprocessing time using algorithms such as Chebyshev formulae.
See also GEN13.

Example:

```
f  1 0  1025 3 -1 1 5 4 3 2 2 1
```

This calls GEN03 to fill a table with a 4th order polynomial function over the x-interval -1 to 1. The origin will be at the offset position 512. The function is post-normalized.

GEN04

This subroutine generates a normalizing function by examining the contents of an existing table.

```
f # time size 4 source# sourcemode
```

size - number of points in the table. Should be power-of-2 plus 1. Must not exceed (except by 1) the size of the source table being examined; limited to just half that size if the sourcemode is of type offset (see below).

source # - table number of stored function to be examined.

sourcemode - a coded value, specifying how the source table is to be scanned to obtain the normalizing function. Zero indicates that the source is to be scanned from left to right. Non-zero indicates that the source has a bipolar structure; scanning will begin at the mid-point and progress outwards, looking at pairs of points equidistant from the center.

Note:

The normalizing function derives from the progressive absolute maxima of the source table being scanned. The new table is created left-to-right, with stored values equal to 1/(absolute maximum so far scanned). Stored values will thus begin with 1/(first value scanned), then get progressively smaller as new maxima are encountered. For a source table which is normalized (values <= 1), the derived values will range from 1/(first value scanned) down to 1. If the first value scanned is zero, that inverse will be set to 1.

The normalizing function from GEN04 is not itself normalized.

GEN04 is useful for scaling a table-derived signal so that it has a consistent peak amplitude. A particular application occurs in waveshaping when the carrier (or indexing) signal is less than full amplitude.

Example:

```
f  2  0  512  4  1  1
```

This creates a normalizing function for use in connection with the GEN03 table 1 example. Midpoint bipolar offset is specified.

GEN05, GEN07

These subroutines are used to construct functions from segments of exponential curves (GEN05) or straight lines (GEN07).

```
f #  time  size  5  a  n1  b  n2  c . . .
f #  time  size  7  a  n1  b  n2  c . . .
```

size - number of points in the table. Must be a power of 2 or power-of-2 plus 1 (see f statement).

a, b, c, etc. - ordinate values, in odd-numbered pfields p5, p7, p9, . . . For GEN05 these must be nonzero and must be alike in sign. No such restrictions exist for GEN07.

n1, n2, etc. - length of segment (no. of storage locations), in even-numbered pfields. Cannot be negative, but a zero is meaningful for specifying discontinuous waveforms (e.g. in the example below). The sum n1 + n2 + ... will normally equal size for fully specified functions. If the sum is smaller, the function locations not included will be set to zero; if the sum is greater, only the first size locations will be stored.

Note:

If p4 is positive, functions are post-normalized (rescaled to a maximum absolute value of 1 after generation). A negative p4 will cause rescaling to be skipped.

Discrete-point linear interpolation implies an increase or decrease along a segment by equal differences between adjacent locations; exponential interpolation implies that the progression is by equal ratio. In both forms the interpolation from a to b is such as to assume that the value b will be attained in the n + 1th location. For discontinuous functions, and for the segment encompassing the end location, this value will not actually be reached, although it may eventually appear as a result of final scaling.

Example:

```
f  1  0  256  7  0  128  1  0  -1  128  0
```

This describes a single-cycle sawtooth whose discontinuity is mid-way in the stored function.

GEN06

This subroutine will generate a function comprised of segments of cubic polynomials, spanning specified points just three at a time.

```
f #  time  size  6  a  n1  b  n2  c  n3  d. .
.
```

size - number of points in the table. Must be a power off or power-of-2 plus 1 (see f statement).

a, c, e, ... - local maxima or minima of successive segments, depending on the relation of these points to adjacent inflexions. May be either positive or negative.

b, d, f, ... - ordinate values of points of inflexion at the ends of successive curved segments. May be positive or negative.

n1, n2, n3... - number of stored values between specified points. Cannot be negative, but a zero is meaningful for specifying discontinuities. The sum n1 + n2 + ... will normally equal size for fully specified functions. (for details, see GEN05).

Note:

GEN06 constructs a stored function from segments of cubic polynomial functions. Segments link ordinate values in groups of 3: point of inflexion, maximum/minimum, point of inflexion. The first complete segment encompasses b,c,d and has length n2 + n3, the next encompasses d,e,f and has length n4 + n5, etc. The first segment (a,b with length n1) is partial with only one inflexion; the last segment may be partial too. Although the inflexion points b,d,f ... each figure in two segments (to the left and right), the slope of the two segments remains independent at that common point (i.e. the 1st derivative will
likely be discontinuous). When a,c,e... are alternately maximum and minimum, the inflexion joins will be relatively smooth; for successive maxima or successive minima the inflexions will be comb-like.

Example:

```
f  1  0  65  6  0  16  .5  16  1  16  0  16 -1
```

This creates a curve running 0 to 1 to -1, with a minimum, maximum and minimum at these values respectively. Inflexions are at .5 and 0, and are relatively smooth.

GEN08

This subroutine will generate a piecewise cubic spline curve, the smoothest possible through all specified points.

```
f  #  time  size  8  a  n1  b  n2  c  n3 d . . .
```

size - number of points in the table. Must be a power of 2 or power-of-2 plus 1 (see f statement).

a, b, c ... - ordinate values of the function.

n1, n2, n3 ... - length of each segment measured in stored values. May not be zero, but may be fractional. A particular segment may or may not actually store any values; stored values will be generated at integral points from the beginning of the function. The sum n1 + n2 + ... will normally equal size for fully specified functions.

Note:

GEN08 constructs a stored table from segments of cubic polynomial functions. Each segment runs between two specified points but depends as well on their neighbors on each side. Neighboring segments will agree in both value and slope at their common point. (The common slope is that of a parabola through that point and its two neighbors). The slope at the two ends of the function is constrained to be zero (flat).

Hint: to make a discontinuity in slope or value in the function as stored, arrange a series of points in the interval between two stored values; likewise for a non-zero boundary slope.

Examples:

```
f  1  0  65  8  0  16  0  16  1  16  0 16  0
```

This example creates a curve with a smooth hump in the middle, going briefly negative outside the hump then flat at its ends.

```
f  2  0  65  8  0  16  0 .1  0  15.9 1  15.9  0  .1  0  16  0
```

This example is similar, but does not go negative.

GEN09, **GEN10**, **GEN19**

These subroutines generate composite waveforms made up of weighted sums of simple sinusoids. The specification of each contributing partial requires 3 pfields using GEN09, 1 using GEN10, and 4 using GEN19.

```
f  #  time  size  9  pna  stra  phsa  pnb  strb  phsb . . .
f  #  time  size  10  str1  str2  str3  str4 . . . .
f  #  time  size  19  pna  stra  phsa  dcoa  pnb  strb  phsb  cob . .
.
```

size - number of points in the table. Must be a power of 2 or power-of-2 plus 1 (see f statement).

pna, pnb, etc. - partial no. (relative to a fundamental that would occupy size locations per cycle) of sinusoid a, sinusoid b, etc. Must be positive, but need not be a whole number, i.e., non-harmonic partials are permitted. Partials may be in any order.

stra, strb, etc. - strength of partials pna, pnb, etc. These are relative strengths, since the composite waveform may be rescaled later. Negative values are permitted and imply a 180 degree phase shift.

phsa, phsb, etc. - initial phase of partials pna, pnb, etc., expressed in degrees.

dcoa, dcob, etc. - DC offset of partials pna, pnb, etc. This is applied after strength scaling, i.e. a value of 2 will lift a 2-strength sinusoid from range [-2,2] to range [0,4] (before later rescaling).

str1, str2, str3, etc. - relative strengths of the fixed harmonic partial numbers 1,2,3, etc., beginning in p5. Partials not required should be given a strength of zero.

Note:

These subroutines generate stored functions as sums of sinusoids of different frequencies. The two major restrictions on GEN10Ñthat the partials be harmonic and in phaseÑdo not apply to GEN09 or GEN19.

In each case the composite wave, once drawn, is then rescaled to unity if p4 was positive. A negative p4 will cause rescaling to be skipped.

Examples:

```
f  1  0  1024  9  1  3  0  3  1  0  9  .3333  180
f  2  0  1024  19  .5  1 270  1
```

f 1 combines partials l, 3 and 9 in the relative strengths in which they are found in a square wave, except that partial 9 is upside down. f 2 creates a rising sigmoid [0 - 2]. Both will be rescaled.

GEN11

This subroutine generates an additive set of cosine partials, in the manner of Csound generators buzz and gbuzz.

```
f  #  time  size  11  nh  lh  r
```

size - number of points in the table. Must be a power of 2 or power-of-2 plus 1 (see f statement).

nh - number of harmonics requested. Must be positive.

lh (optional) - lowest harmonic partial present. Can be positive, zero or negative. The set of partials can begin at any partial number and proceeds upwards; if lh is negative, all partials below zero will reflect in zero to produce positive partials without phase change (since

cosine is an even function), and will add constructively to any positive partials in the set. The default value is 1.

r (optional) - multiplier in an amplitude coefficient series. This is a power series: if the lhth partial has a strength coefficient of A the (lh + n)th partial will have a coefficient of A * r**n, i.e. strength values trace an exponential curve. r may be positive, zero or negative, and is not restricted to integers. The default value is 1.

Note:

This subroutine is a non-time-varying version of the csound buzz and gbuzz generators, and is similarly useful as a complex sound source in subtractive synthesis. With lh and r present it parallels gbuzz; with both absent or equal to 1 it reduces to the simpler buzz (i.e. nh equal-strength harmonic partials beginning with the fundamental).

Sampling the stored waveform with an oscillator is more efficient than using dynamic buzz units. However, the spectral content is invariant, and care is necessary lest the higher partials exceed the Nyquist during sampling to produce foldover.

Examples:

```
f 1   0   2049  11   4
f 2   0   2049  11   4   1   1
f 3   0   2049 -11   7   3  .5
```

The first two tables will contain identical band-limited pulse waves of four equal-strength harmonic partials beginning with the fundamental. The third table will sum seven consecutive harmonics, beginning with the third, and at progressively weaker strengths (1, .5, .25, .125 . . .). It will not be post-normalized.

GEN12

This generates the log of a modified Bessel function of the second kind, order 0, suitable for use in amplitude-modulated FM.

```
f   #   time   size   -12   xint
```

size - number of points in the table. Must be a power of 2 or a power-of-2 plus 1 (see f statement). The normal value is power-of-2 plus 1.

xint - specifies the x interval [0 to +int] over which the function is defined.
Note:

This subroutine draws the natural log of a modified Bessel function of the second kind, order 0 (commonly written as I subscript 0), over the x-interval requested. The call should have rescaling inhibited.

The function is useful as an amplitude scaling factor in cycle-synchronous amplitude-modulated FM. (See Palamin & Palamin, J. Audio Eng. Soc., 36/9, Sept. 1988, pp.671-684.) The algorithm is interesting because it permits the normally symmetric FM spectrum to be made asymmetric around a frequency other than the carrier, and is thereby useful for formant positioning. By using a table lookup index of I(r - 1/r), where I is the FM modulation index and r is an exponential parameter affecting partial strengths, the Palamin algorithm becomes relatively efficient, requiring only oscil's, table lookups, and a single exp call.

Example:

```
f   1   0   2049   -12   20
```

This draws an unscaled ln(I0(x)) from 0 to 20.

GEN13, **GEN14**

These subroutines use Chebyshev coefficients to generate stored polynomial functions which, under waveshaping, can be used to split a sinusoid into harmonic partials having a predefinable spectrum.

```
f   #   time   size   13   xint   xamp   h0   h1   h2 . . . hn
f   #   time   size   14   xint   xamp   h0   h1   h2 . . . hn
```

size - number of points in the table. Must be a power of 2 or a power-of-2 plus 1 (see f statement). The normal value is power-of-2 plus 1.

xint - provides the left and right values [-xint, +xint] of the x interval over which the polynomial is to be drawn. These subroutines both call GEN03 to draw their functions; the p5 value here is therefor expanded to a negative-positive p5,p6 pair before GEN03 is actually called. The normal value is 1.

xamp - amplitude scaling factor of the sinusoid input that is expected to produce the following spectrum.

h0, h1, h2, ... hn - relative strength of partials 0 (DC), 1 (fundamental), 2 ... that will result when a sinusoid of amplitude xamp * int(size/2)/xint is waveshaped using this function table. These values thus describe a frequency spectrum associated with a particular factor xamp of the input signal.

Note:

GEN13 is the function generator normally employed in standard waveshaping. It stores a polynomial whose coefficients derive from the Chebyshev polynomials of the first kind, so that a driving sinusoid of strength xamp will exhibit the specified spectrum at output. Note that the evolution of this spectrum is generally not linear with varying xamp. However, it is bandlimited (the only partials to appear will be those specified at generation time); and the partials will tend to occur and to develop in ascending order (the lower partials dominating at low xamp, and the spectral richness increasing for higher values of xamp). A negative hn value implies a 180 degree phase shift of that partial; the requested full-amplitude spectrum will not be affected by this shift, although the evolution of several of its component partials may be. The pattern +,+,-,-,+,+,... for h0,h1,h2... will minimize the normalization problem for low xamp values (see above), but does not necessarily provide the smoothest pattern of evolution.

GEN14 stores a polynomial whose coefficients derive from Chebyshevs of the second kind.

Example:

```
f   1   0  1025   13   1  1   0   5   0   3   0   1
```

This creates a function which, under waveshaping, will split a sinusoid into 3 odd-harmonic partials of relative strength 5:3:1.

GEN15

This subroutine creates two tables of stored polynomial functions, suitable for use in phase quadrature operations.

```
f   #   time  size  15  xint   xamp  h0  phs0   h1  phs1  h2  phs2 . .
.
```

size - number of points in the table. Must be a power of 2 or a power-of-2 plus 1 (see f statement). The normal value is power-of-2 plus 1.

xint - provides the left and right values [-xint, +xint] of the x interval over which the polynomial is to be drawn. This subroutine will eventually call GEN03 to draw both functions; this p5 value is

therefor expanded to a negative-positive p5, p6 pair before GEN03 is actually called. The normal value is 1.

xamp - amplitude scaling factor of the sinusoid input that is expected to produce the following spectrum.

h0, h1, h2, ... hn - relative strength of partials 0 (DC), 1 (fundamental), 2 ... that will result when a sinusoid of amplitude xamp * int(size/2)/xint is waveshaped using this function table. These values thus describe a frequency spectrum associated with a particular factor xamp of the input signal.

phs0, phs1, ... - phase in degrees of desired harmonics h0, h1, ... when the two functions of GEN15 are used with phase quadrature.

Note:

GEN15 creates two tables of equal size, labelled f # and f # + 1. Table # will contain a Chebyshev function of the first kind, drawn using GEN03 with partial strengths h0cos(phs0), h1cos(phs1), ... Table #+1 will contain a Chebyshev function of the 2nd kind by calling GEN14 with partials h1sin(phs1), h2sin(phs2),... (note the harmonic displacement). The two tables can be used in conjunction in a waveshaping network that exploits phase quadrature.

GEN17

This subroutine creates a step function from given x-y pairs.

```
f  #  time  size  17  x1 a x2 b x3 c . . .
```

size - number of points in the table. Must be a power of 2 or a power-of-2 plus 1 (see f statement). The normal value is power-of-2 plus 1.

x1, x2, x3, etc. - x-ordinate values, in ascending order, 0 first.

a, b, c, etc. - y-values at those x-ordinates, held until the next x-ordinate.

This subroutine creates a step function of x-y pairs whose y-values are held to the right. The right-most y-value is then held to the end of the table. The function is useful for mapping one set of data values onto another, such as MIDI note numbers onto sampled sound ftable numbers (see loscil).
Example:

```
f 1 0 128 -17  0 1  12 2  24 3  36 4  48 5 60 6  72 7  84 8
```

This describes a step function with 8 successively increasing levels, each 12 locations wide except the last which extends its value to the end of the table. Rescaling is inhibited. Indexing into this table with a MIDI note-number would retrieve a different value every octave up to the eighth, above which the value returned would remain the same.

5. SCOT: A Score Translator

Scot is a language for describing scores in a fashion that parallels traditional music notation. Scot is also the name of a program which translates scores written in this language into standard numeric score format so that the score can be performed by Csound. The result of this translation is placed in a file called score. A score file written in Scot (named file.sc, say) can be sent through the translator by the command

```
scot file.sc
```

The resulting numeric score can then be examined for errors, edited, or performed by typing

```
csound file.orc score
```

Alternatively, the command

```
csound file.orc -S file.sc
```

would combine both processes by informing Csound of the initial score format.

Internally, a Scot score has at least three parts: a section to define instrument names, a section to define functions, and one or more actual score sections. It is generally advisable to keep score sections short to facilitate finding errors. The overall layout of a Scot score has three main sections:

```
orchestra { .... }
functions { .... }
score { .... }
```

The last two sections may be repeated as many times as desired.
The functions section is also optional. Throughout this Scot document, bear in mind that you are free to break up each of these divisions into as many lines as seem convenient, or to place a carriage return anywhere you are allowed to insert a space, including before and after the curly brackets. Furthermore, you may use as many spaces or tabs as you need to make the score easy to read. Scot imposes no formatting restrictions except that numbers, instrument names, and keywords (for example, orchestra) may not be broken with spaces. You may insert comments (such as measure numbers) anywhere in the score by preceding them with a semicolon. A semicolon causes Scot to ignore the rest of a line.

Orchestra Declaration Section

The orchestra section of a Scot score serves to designate instrument names for use within the score. This is a matter of convenience, since an orchestra knows instruments only by numbers, not names. If you declare three instruments, such as:

```
orchestra { flute=1 cello=2 trumpet=3 }
```

Csound will neither know nor care what you have named the note lists. However, when you use the name $flute, Scot will know you are referring to instr 1 in the orchestra, $cello will refer to instr 2, and $trumpet will be instr 3. You may meaningfully skip numbers or give several instruments the same number. It is up to you to make sure that your orchestra has the correct instruments and that the association between these names and the instruments is what you intend. There is no limit (or a very high one, at least) as to how many instruments you can declare.

Function Declaration Section

The major purpose of this division is to allow you to declare function tables for waveforms, envelopes, etc. These functions are declared exactly as specified for Csound. In fact, everything you type between the brackets in this section will be passed directly to the resulting numeric score with no modification, so that mistakes will not be caught by the Scot program, but rather by the subsequent performance. You can use this section to write notes for instruments for which traditional pitch-rhythm notation is inappropriate. The most common example would be turning on a reverb instrument. Instruments referenced in this way need not appear in the Scot orchestra declaration.
Here is a possible function declaration:

```
functions {
f1 0 256 10 1 0 .5 0 .3
f2 0 256 7 0 64 1 64 .7 64 0
i9 0 -1 3          ; this turns on instr 9
}
```

Score Section

The Scot statements contained inside the braces of each score statement is translated into a numeric score Section (q.v.). It is wise

to keep score sections small, say seven or eight measures of five voices at a time. This avoids overloading the system, and simplifies error checking.

The beginning of the score section is specified by typing:

```
score {
```

Everything which follows until the braces are closed is within a single section. Within this section you write measures of notes in traditional pitch and rhythm notation for any of the instrument names listed in your orchestra declaration. These notes may carry additional information such as slurs, ties and parameter fields. Let us now consider the format for notes entered in a Scot score.

The first thing to do is name the instrument you want and the desired meter. For example, to write some 4/4 measures for the cello, type:

```
$cello
!ti "4/4"
```

The dollar sign and exclamation point tell Scot that a special declarator follows. The time signature declarator is optional; if resent, Scot will check the number of beats in each measure for you.

Pitch and Rhythm

The two basic components of a note statement are the pitch and duration. Pitch is specified using the alphabetic note name, and duration is specified using numeric characters. Duration is indicated at the beginning of the note as a number representing the division of a whole beat. You may always find the number specifying a given duration by thinking of how many times that duration would fit in a 4/4 measure. Also, if the duration is followed by a dot (`.') it is increased by 50%, exactly as in traditional notation. Some sample durations are listed below:

whole note	1
half note	2
double dotted quarter	4..
dotted quarter note	4.
quarter note	4
half note triplet	6
eighth note	8
eighth note triplet	12
sixteenth note	16
thirty-second note	32

Pitch is indicated next by first (optionally) specifying the register and then the note name, followed by an accidental if desired. Normally, the "octave following" feature is in effect. This feature causes any note named to lie within the interval of an augmented fourth of the previous note, unless a new register is chosen. The first note you write will always be within a fourth of middle c unless you choose a different register.

For example, if the first note of an instrument part is notated g flat, the scot program assigns the pitch corresponding to the g flat below middle c. On the other hand, if the first note is f sharp, the pitch assigned will be the f sharp above middle c.
Changes of register are indicated by a preceding apostrophe for each octave displacement upward or a preceding comma for each octave displacement downward. Commas and apostrophes always displace the pitch by the desired number of octaves starting from that note which is within an augmented fourth of the previous pitch.

If you ever get lost, prefacing the pitch specification with an `=' returns the reference to middle c. It is usually wise to use the equals sign in your first note statement and whenever you feel uncertain as to what the current registration is. Let us now write two measures for the cello part, the first starting in the octave below middle c and the second repeating but starting in the octave above middle c:

```
$cello
!ti "4/4"
4=g 4e 4d 4c/ 4='g 4e 4d 4c
```

As you can see, a slash indicates a new measure and we have chosen to use the dummy middle c to indicate the new register. A more convenient way of notating these two measures would be to type the following:

```
$cello
!ti "4/4"
4=g e d c/ ''g e d c
```

You may observe in this example that the quarter note duration carries to the following notes when the following durations are left unspecified. Also, two apostrophes indicate an upward pitch displacement of two octaves from two g's below middle c, where the pitch would have fallen without any modification. It is important to remember three things, then, when specifying pitches:

1) Note pitches specified by letter name only (with or without accidental) will always fall within an interval of a fourth from the preceding pitch.

2) These pitches can be octave displaced upward or downward by preceding the note letter with the desired number of apostrophes or commas.

3) If you are unsure of the current register, you may begin the pitch component of the note with an equals sign which acts as a dummy middle c.

The pitch may be modified by an accidental after the note name:

n	natural
#	sharp
- (hyphen)	flat
##	double sharp
-- (double hyphen)	double flat

Accidentals are carried throughout the measure just as in traditional music notation. However, an accidental specified within a measure will hold for that note in all registers, in contrast with traditional notation. Therefore, make sure to specify n when you no longer want an accidental applied to that pitch-class.

Pitches entered in the Scot score are translated into the appropriate octave point pitch-class value and appear as parameter p5 in the numeric score output. This means you must design your instruments to accept p5 as pitch.

Rests are notated just like notes but using the letter r instead of a pitch name. 4r therefore indicates a quarter rest and 1r a whole rest. Durations carry from rest to rest or to following pitches as mentioned above.

The tempo in beats per minute is specified in each section by choosing a single instrument part and using tempo statements (e.g. t90) at the various points in the score as needed. A quarter note is interpreted as a single beat, and tempi are interpolated between the intervening beats (see score t statement).

Scot Example I

_

A Scot encoding of this score might appear as follows:

```
; A BASIC Tune
orchestra { guitar=1 bass=2 }
functions  {
f1 0 512 10 1 .5 .25 .126
f2 0 256  7 1 120 1 8 0 128 1
}
score    {  ;section 1
$guitar
!ti "4/4"
4=c 8d e- f r 4='c/
8.b- 16a a- g  g- f  4e- c/
```

```
$bass
2=,,c ‘a-/
g  =,c/
}
score    {  ;section 2
$guitar
!ti “4/4”
6=‘c r c 4..c## 16e- /
6f r f 4..f## 16b /
$bass
4=,,c ‘g, c ‘g/
2=a-  g /
}
```

The score resulting from this Scot notation is shown at the end of this chapter.

Groupettes

Duration numbers can have any integral value; for instance,

```
!time “4/4”
5cdefg/
```

would encode a measure of 5 in the time of 4 quarter notes.
However, specification of arbitrary rhythmic groupings in this way is at best awkward. Instead, arbitrary portions of the score section may be enclosed in groupette brackets. The durations of all notes inside groupette brackets will be multiplied by a fraction n/d, where the musical meaning is d in the time of n.
Assuming d and n here are integers, groupette brackets may take these several forms:

```
{d:n:    ....    :}   d in the time of n
{d:: ....    :}       n will be the largest power of 2 less than d
{:  ....    :}        d=3, n=2 (normal triplets)
```

It can be seen that the second and third form are abbreviations for the more common kinds of groupettes. (Observe the punctuation of each form carefully.) Groupettes may be nested to a reasonable depth. Also, groupette factors apply only after the written duration is carried from note to note. Thus, the following example is a correct specification for two measures of 6/8 time:

```
!time “6/8” 8cde {4:3: fgab :} / crc 4.c /
```

The notes inside the groupette are 4 in the space of 3 8th notes, and the written-8th-note duration carries normally into the next measure. This closely parallels the way groupette brackets and note durations interact in standard notation.

Slurs and Ties

Now that you understand part writing in the Scot language, we can start discussing more elaborate features. Immediately following the pitch specification of each note, one may indicate a slur or a tie into the next note (assuming there is one), but not both simultaneously. The slur is typed as a single underscore (`_’) and a tie as a double underscore (`__’). Despite the surface similarity, there is a substantial difference in the effect of these modifiers.

For purposes of Scot, tied notes are notes which, although comprised of several graphic symbols, represent only a single musical event. (Tied notes are necessary in standard music notation for several reasons, the most common being notes which span a measure line and notes with durations not specifiable with a single symbol, such as quarter note tied to a sixteenth). Notes which are tied together are summed by duration and output by Scot as a single event. This means you cannot, for example, change the parameters of a note in the middle of a tie (see below). Two or more notes may be tied together, as in the following example, which plays an f# for eleven beats:

```
!ti “4/4”
1 f#__ / 1 f#__ / 2. f# 4r /
```

By contrast, slurred notes are treated as distinct notes at the Csound level, and may be of arbitrary pitch. The presence of a slur is reflected in parameter p4, but the slur has no other meaning beyond the interpretation of p4 by your instrument.
Since instrument design is beyond the scope of this manual, it will suffice for now to explain that the Scot program gives sets p4 to one of four values depending on the existence of a slur before and after the note in question. This means Scot pays attention not only to the slur attached to a given note, but whether the preceding note specified a slur. The four possibilities are as follows, where the p4 values are taken to apply to the note `d’:

```
4c  d       (no slur)              p4 = 0
4c  d_      (slur after only)      p4 = 1
4c_ d       (slur before only)     p4 = 2
4c_ d_      (before & after)       p4 = 3
```

Parameters

The information contained in the Scot score notation we have considered so far is manifested in the output score in parameters p1 through p5 in the following way:

```
p1:  instrument number
p2:  initialization time of instrument
p3:  duration
p4:  slur information
p5:  pitch information in octave point pitch-class notation
```

Any additional parameters you may want to enter are listed in brackets as the last part of a note specification. These parameters start with p6 and are separated from each other with spaces. Any parameters not specified for a particular note have their value carried from the most recently specified value. You may choose to change some parameters and not others, in which case you can type a dot (`.’) for parameters whose values don’t change, and new values for those that do. Alternatively, the construction N:, where N is an integer, may be used to indicate that the following parameter specifications apply to successive parameters starting with parameter N. For example:

```
4e[15000 3 4 12:100 150] g_ d_[10000 . 5]    c
```

Here, for the first two quarter notes p6, p7, p8, p12, and p13 respectively assume the values 15000, 3, 4, 100, and 150. The values of p9 through p11 are either unchanged, or implicitly zero if they have never been specified in the current section. On the third quarter note, the value of p6 is changed to 10000, and the value of p8 is changed to 5. All others are unchanged.

Normally, parameter values are treated as globalsÑthat is, a value specification will carry to successive notes if no new value is specified. However, if a parameter specification begins with an apostrophe, the value applies locally to the current note only, and will not carry to successive notes either horizontally or vertically (see divisi below).

Pfield Macros

Scot has a macro text substitution facility which operates only on the pfield specification text within brackets. It allows control values to be specified symbolically rather than numerically. Macro definitions appear inside brackets in the orchestra section. A single bracketed list of macro definitions preceding the first instrument declaration defines macros which apply to all instruments. An additional bracketed list of
definitions may follow each instrument to specify macros that apply to that particular instrument.

```
orchestra {
    [ pp=2000 p=4000 mp=7000 mf=10000 f=20000 ff=30000
      modi = 11: w = 1 x = 2 y = 3 z= 4
      vib = “10:1 “ novib = “10:0 1”
    ]
```

```
violin = 1     [ pizz = " 20:1" arco = " 20:0" ]
serpent = 3    [ ff = 25000 sfz = 'f sffz = 'ff]
}
score {
 $violin = 4c[mf modi z.y novib] d e a['f vib3] /
        8 b[pizz]c 4d[f] 2c[ff arco] /
 $serpent =, 4.c[mp modi y.x] 8b 2c /
        'g[f ], c[ff] /
}
```

As can be seen from this example, a macro definition consists of the macro name, which is a string of alphabetic characters, followed by an equal sign, followed by the macro value. As usual, spaces, tabs, and newlines may be used freely. The macro value may contain arbitrary characters, and may be quoted if spacing characters need to be included.

When a macro name is encountered in bracketed pfield lists in a score section, that name is replaced with the macro text with no additional punctuation supplied. The macro text may itself invoke other macros, although it is a serious error for a macro to contain itself, directly or indirectly. Since macro names are identified as strings of alphabetic characters, and no additional spaces are provided when a macro is expanded, macros may easily perform such concatenations as found in the first serpent note above, where the integer and fractional parts of a single pfield are constructed. Also, a macro may do no more than define a symbolic pfield, as in the definition of modi. The primary intention of macros is in fact not only to reduce the number of characters required, but also to enable symbolic definitions of parameter numbers and parameter values. For instance, a particular instrument's interpretation of the dynamic ff can be changed merely by changing a macro value, rather than changing all occurrences of that particular value in the score.

Divisi

Notes may be stacked on top of each other by using a back arrow (`<') between the notes of the divisi. Each time Scot encounters a back arrow, it understands that the following note is to start at the same time as the note to the left of the back arrow.
Durations, accidentals and parameters carry from left to right through the divisi. Each time these are given new values, the notes to the right of the back arrows also take on the new values unless they are specified again.

When writing divisi you can stack compound events by enclosing them in parentheses. Also, divisi which occur at the end of the measure must have the proper durations or the scot program will mis-interpret the measure duration length.

Scot Example II

Scot encoding:

```
orchestra { right=1 left=2 }
functions { f1 0 256 10 1}
score {
$right !key "-b"
; since p5 is pitch, p7 is set to the pitch of next note
!ti "2/4"
!next p5 "p7"  ;since p5 is pitch, p7 refers to pitch of next note
!next p6 "p8"  ;If p6 is vol, say, then p8 refers to vol of next note
t90
8r c[3 np5]<e<='g r c<f<a / t90 r d-<g<b r =c[5]<f<a__ /
!ti "4/4"
t80
4d_<f__<(8a g__) 4c<(8fe)<4g 4.c<f<f 8r/

$left  !key "-b"
!next p5 "p7"
!next p6 "p8"
!ti "2/4"
8=,c[3 np5] r f r/ e r f r/
!ti "4/4"
2b_[5]<(4=,b_c) 4.a<f 8r/
}
```

Notice in this example that tempo statements occurred in instrument `right' only. Also, all notes had p6=3 until the third measure, at which point p6 took on the value 5 for all notes. The next parameter option used is described in Additional Features. The output score is given at the end.

Additional Features

Several options can be included in any of the individual instrument parts within a section. A sample statement follows the description of each option. The keyword which follows the `!' in these statements may be abbreviated to the first two characters.

Key Signatures

Any desired key signature is specified by listing the accidentals as they occur in a key signature statement. Thereafter, all notes of that instrument part are sharpened or flattened accordingly. For example, for the key of D, type

```
!key "#fc"
```

Accidental Following

Accidental following may be turned on or off as needed. When turned off, accidentals no longer carry throughout the measure as in traditional notation. This convention is sometimes used in contemporary scores.

```
!accidentals "off"
```

Octave Following

Turning off octave following indicates that pitches stay in the same absolute octave register until explicitly moved. An absolute octave starts at pitch c and ends at the b above it. The octave middle-c-to-b is indicated with an equals sign (`=') and octave displacement is indicated with the appropriate number of commas or apostrophes. These displacements are cummulative. For example,

```
!octaves "off"
4='c g b 'c
```

starts at the c above middle c and ends at two c's above middle c.

Vertical Following

Turning off vertical following means that durations, register, and parameters only carry horizontally from note to note and not vertically as described in the section on divisi.

```
!vertical "off"
```

Transposition

Any instrument part can be transposed to another key by indicating the intervalic difference between the notated key and the desired key. This difference is always taken with reference to middle c - to transpose a whole step upward, for example, type

```
!transpose "d"
```

This indicates that the part is transposed by the interval difference between middle c and d.

Next-value and Previous-value Parameteres

In order to play a note, it is sometimes necessary for an instrument to know what value one or more parameters will have for the next note. For instance, an instrument might be designed which glisses during the last portion of its performance (perhaps only when a slur is indicated) from its written pitch to the pitch of the next note. This can only be done, of course, if the instrument can know what the pitch of the next note will be.

The necessary information can be provided using next-value parameters. A next value parameter might be declared by

```
!next p5 "p6"
```

which is interpreted to mean that for the current instrument, p6 will contain the next note's p5 value. This holds true globally for all occurences of this instrument until further modifications. If for any reason you wish to override this value, p6 may be filled in explicitly. This is sometimes useful for the last note of a section, for which p6 will otherwise assume the p5 value for the current note. The next-value feature is illustrated in the Scot example II.

The necessary information may also be provided using standard numeric score next-value parameters. A parameter argument containing the symbol npx (where x is an integer) will substitute parameter number x of the following note for that instrument.
Similarly, the value of a parameter occurring during the previous note may be referenced with the symbol ppx (where x is an integer). Details of the next- and previous-value parameter feature may be found in the Numeric Scores section.

Ramping

Pfields containing the symbol < will be replaced by values derived from linear interpolation of a time-based ramp. Ramp endpoints are defined by the first real number found in the same pfield of a preceding and following note played by the same instrument. Details of the ramping feature are likewise found in the Numeric Scores section.

f0 Statements

In each score section, Scot automatically produces an f0 statement with a p2 equal to the ending time of the last note or rest in the section. Thus, `dead time' at the end of a section for reverberation decay or whatever purpose may be specified musically by rests in one or more parts. See the eighth rest at the end of Scot example II and its output score shown below.

Output Scores

Output file score from Scot Example I.

```
f1 0 512 10 1 .5 .25 .126
f2 0 256  7 1 120 1 8 0 128 1
i1.01 0 1 0 8.00
i1.01 1 0.5 0 8.02
i1.01 1.5 0.5 0 8.03
i1.01 2 0.5 0 8.05
i1.01 3 1 0 9.00
i1.01 4 0.75 0 8.10
i1.01 4.75 0.25 0 8.09
i1.01 5 0.25 0 8.08
i1.01 5.25 0.25 0 8.07
i1.01 5.5 0.25 0 8.06
i1.01 5.75 0.25 0 8.05
i1.01 6 1 0 8.03
i1.01 7 1 0 8.00
i2.01 0 2 0 6.00
i2.01 2 2 0 6.08
i2.01 4 2 0 6.07
i2.01 6 2 0 7.00
t0 60
f0 8
s
i1.01 0 0.6667 0 9.00
i1.01 1.3333 0.6667 0 9.00
i1.01 2 1.75 0 9.02
i1.01 3.75 0.25 0 9.03
i1.01 4 0.6667 0 9.05
i1.01 5.3333 0.6667 0 9.05
i1.01 6 1.75 0 9.07
i1.01 7.75 0.25 0 9.09
i2.01 0 1 0 6.00
i2.01 1 1 0 6.07
i2.01 2 1 0 6.00
i2.01 3 1 0 6.07
i2.01 4 2 0 7.08
i2.01 6 2 0 7.07
t0 60
f0 8
s
```

Output file score from Scot Example II.

```
f1 0 256 10 1
c r1 n 7 5
c r1 n 8 6
i1.01 0.5000 0.5000 0 8.00 3 8.00 3
i1.02 0.5000 0.5000 0 8.04 3 8.05 3
i1.03 0.5000 0.5000 0 8.07 3 8.09 3
i1.01 1.5000 0.5000 0 8.00 3 8.01 3
i1.02 1.5000 0.5000 0 8.05 3 8.07 3
i1.03 1.5000 0.5000 0 8.09 3 8.10 3
i1.01 2.5000 0.5000 0 8.01 3 8.00 5
i1.02 2.5000 0.5000 0 8.07 3 8.05 5
i1.03 2.5000 0.5000 0 8.10 3 8.09 5
i1.01 3.5000 0.5000 0 8.00 5 8.02 5
i1.02 3.5000 0.5000 0 8.05 5 8.05 5
i1.01 4.0000 1.0000 1 8.02 5 8.00 5
i1.03 3.5000 1.0000 0 8.09 5 8.07 5
i1.01 5.0000 1.0000 2 8.00 5 8.00 5
i1.02 4.0000 1.5000 0 8.05 5 8.04 5
i1.02 5.5000 0.5000 0 8.04 5 8.05 5
i1.03 4.5000 1.5000 0 8.07 5 8.05 5
i1.01 6.0000 1.5000 0 8.00 5 8.00 5
i1.02 6.0000 1.5000 0 8.05 5 8.05 5
i1.03 6.0000 1.5000 0 8.05 5 8.05 5
c r2 n 7 5
c r2 n 8 6
i2.01 0.0000 0.5000 0 7.00 3 7.05 3
i2.01 1.0000 0.5000 0 7.05 3 7.04 3
i2.01 2.0000 0.5000 0 7.04 3 7.05 3
i2.01 3.0000 0.5000 0 7.05 3 7.10 5
i2.01 4.0000 2.0000 1 7.10 5 7.09 5
i2.02 4.0000 1.0000 1 6.10 5 7.00 5
i2.02 5.0000 1.0000 2 7.00 5 7.05 5
i2.01 6.0000 1.5000 2 7.09 5 7.09 5
i2.02 6.0000 1.5000 0 7.05 5 7.05 5
t0 60 0.0000 90.0000 2.0000 90.0000 4.0000 80.0000 4.0000
90.0000
f0 8.0000
s
e
```

6. The Unix CSOUND Command

csound is a command for passing an orchestra file and score file to Csound to generate a soundfile. The score file can be in one of many different formats, according to user preference.
Translation, sorting, and formatting into orchestra-readable numeric text is handled by various preprocessors; all or part of the score is then sent on to the orchestra. Orchestra performance is influenced by command flags, which set the level of displays and console reports, specify 1/0 filenames and sample formats, and declare the nature of realtime sensing and control.

The format of a command is:

```
csound [-flags] orchname scorename
```

where the arguments are of 2 types: flag arguments (beginning with a "-"), and name arguments (such as filenames). Certain flag arguments take a following name or numeric argument. The available flags are:

-I, -n	sound output inhibitors
-iName, -oName	sound I/O filenames
-bNumb, -BNumb, -h	audio buffers & header control

-A, -c, -a, -u, -s, -1, -f audio output formats
-v, -mNumb, -d, -g message & display levels
-S, -xName, -tNumb score formats & tempo control
-LName line-oriented realtime event stream
-MName, -FName, -PNumb MIDI event streams
-N, -T notify or terminate when done

Flags may appear anywhere in the command line, either separately or bundled together. A flag taking a Name or Number will find it in that argument, or in the immediately subsequent one. The following are thus equivalent commands:

csound -nm3 orchname -Sxxfilename scorename
csound -n -m 3 orchname -x xfilename -S scorename

All flags and names are optional. The default values are:

csound -s -otest -b1024 -B1024 -m7 -P128 orchname scorename

where orchname is a file containing Csound orchestra code, and scorename is a file of score data in standard numeric score format, optionally presorted and time-warped. If scorename is omitted, there are two default options: 1) if realtime input is expected (-L, -M or -F), a dummy scorefile is substituted consisting of the single statement 'f 0 3600' (i.e. listen for RT input for one hour); 2) else csound uses the previously processed score.srt in the current directory.

Csound reports on the various stages of score and orchestra processing as it goes, doing various syntax and error checks along the way. Once the actual performance has begun, any error messages will derive from either the instrument loader or the unit generators themselves. A csound command may include any rational combination of the following flag arguments, with default values as described:

csound -I
I-time only. Allocate and initialize all instruments as per the score, but skip all P-time processing (no k-signals or a-signals, and thus no amplitudes and no sound). Provides a fast validity check of the score pfields and orchestra i-variables.

csound -n
Nosound. Do all processing, but bypass writing of sound to disk.
This flag does not change the execution in any other way.

csound -i isfname
Input soundfile name. If not a full pathname, the file will be ought first in the current directory, then in that given by the environment variable SSDIR (if defined), then by SFDIR. The name stdin will cause audio to be read from standard input. If RTAUDIO is enabled, the name devaudio will request sound from the host audio input device.

csound -o osfname
Output soundfile name. If not a full pathname, the soundfile will be placed in the directory given by the environment variable SFDIR (if defined), else in the current directory. The name stdout will cause audio to be written to standard output. If no name is given, the default name will be test. If RTAUDIO is enabled, the name devaudio will send to the host audio output device.

csound -b Numb
Number of audio sample-frames per soundio software buffer. Large is efficient, but small will reduce audio I/O delay. The default is 1024. In realtime performance, Csound waits on audio I/O on Numb boundaries. It also processes audio (and polls for other input like MIDI) on orchestra ksmps boundaries. The two can be made synchronous. For convenience, if Numb = -N (is negative) the effective value is ksmps * N (audio synchronous with k-period boundaries). With N small (e.g. 1) polling is then frequent and also locked to fixed DAC sample boundaries.

csound -B Numb
Number of audio sample-frames held in the DAC hardware buffer. This is a threshold on which software audio I/O (above) will wait before returning. A small number reduces audio I/O delay; but the value is often hardware limited, and small values will risk data lates. The default is 1024.

csound -h
No header on output soundfile. Don't write a file header, just binary samples.

csound {-c, -a, -u, -s, -l, -f}
Audio sample format of the output soundfile. One of:
c 8-bit signed character
a 8-bit a-law
u 8-bit u-law
s short integer
l long integer
f single-precision float (not playable, but can be read by -i, soundin and GEN01)

csound -A
Write an AIFF output soundfile. Restricts the above formats to c, s, or l.

csound -v
Verbose translate and run. Prints details of orch translation and performance, enabling errors to be more clearly located.

csound -m Numb
Message level for standard (terminal) output. Takes the sum of 3 print control flags, turned on by the following values: 1 = note amplitude messages, 2 = samples out of range message, 4 = warning messages. The default value is m7 (all messages on).

csound -d
Suppress all displays.

csound -g
Recast graphic displays into ascii characters, suitable for any terminal.

csound -S
Interpret scorename as a Scot file and create a standard score file (named "score") from it, then sort and perform that.

csound -x xfile
Extract a portion of the sorted score score.srt, according to xfile (see extract below).

csound -t Numb
Use the uninterpreted beats of score.srt for this performance, and set the initial tempo at Numb beats per minute. When this flag is set, the tempo of score performance is also controllable from within the orchestra (see the tempo unit).

csound -L devname
Read Line-oriented realtime score events from device devname. The name stdin will permit score events to be typed at your terminal, or piped from another process. Each line-event is terminated by a carriage-return. Events are coded just like those in a standard numeric score, except that an event with p2=0 will be performed immediately, and an event with p2=T will be performed T seconds after arrival. Events can arrive at any time, and in any order. The score carry feature is legal here, as are held notes (p3 negative) and string arguments, but ramps and pp or np references are not.

csound -M devname
Read MIDI events from device devname.

csound -F mfname
Read MIDI events from midifile mfname.

csound -P Numb
Set MIDI sustain pedal threshold (0 - 128). The official switch value of 64 is normally too low, and is more realistic above 100.
The default value of 128 will block all pedal info.

csound -N
Notify (ring the bell) when score or miditrack is done.

csound -T
Terminate the performance when miditrack is done.

The EXTRACT feature

This feature will extract a segment of a sorted numeric score file according to instructions taken from a control file. The control file contains an instrument list and two time points, from and to, in the form:

instruments 1 2 from 1:27.5 to 2:2

The component labels may be abbreviated as i, f and t. The time points denote the beginning and end of the extract in terms of:

[section no.] : [beat no.].

each of the three parts is also optional. The default values for missing i, f or t are:

all instruments, beginning of score, end of score.

extract reads an orchestra-readable score file and produces an orchestra-readable result. Comments, tabs and extra spaces are flushed, w and a statements are added and an f0 reflecting the extract length is appended to the output. Following an extract process, the abbreviated score will contain all function table statements, together with just those note statements that occur in the from-to interval specified. Notes lying completely in the interval will be unmodified; notes that lie only partly within will have their p3 durations truncated as necessary.

Independent Preprocessing

Although the result of all score preprocessing is retained in the file score.srt after orchestra performance (it exists as soon as score preprocessing has completed), the user may sometimes want to run these phases independently. The command

scot filename

will process the Scot formatted filename, and leave a standard numeric score result in a file named score for perusal or later processing.

The command
scscort < infile > outfile

will put a numeric score infile through Carry, Tempo, and Sort preprocessing, leaving the result in outfile.

Likewise extract, also normally invoked as part of the csound command, can be invoked as a standalone program:

extract xfile < score.sort > score.extract

This command expects an already sorted score. An unsorted score should first be sent through scsort then piped to the extract program:

scsort < scorefile | extract xfile > score.extract

Appendix 1: The Soundfile Utility Programs

The Csound Utilities are soundfile preprocessing programs that return information on a soundfile or create some analyzed version of it for use by certain Csound generators. Though different in goals, they share a common soundfile access mechanism and are describable as a set. The Soundfile Utility programs can be invoked in two equivalent forms:

csound -U utilname [flags] filenames . . .
utilname [flags] filenames . . .

In the first, the utility is invoked as part of the Csound executable, while in the second it is called as a standalone program. The second is smaller by about 200K, but the two forms are identical in function. The first is convenient in not requiring the maintenance and use of several independent programsÑone program does all. When using this form, a -U flag detected in the command line will cause all subsequent flags and names to be interpreted as per the named utility; i.e. Csound
generation will not occur, and the program will terminate at the end of utility processing.

Directories. Filenames are of two kinds, source soundfiles and resultant analysis files. Each has a hierarchical naming convention, influenced by the directory from which the Utility is invoked. Source soundfiles with a full pathname (begins with dot (.), slash (/), or for ThinkC includes a colon (:)), will be sought only in the directory named. Soundfiles without a path will be sought first in the current directory, then in the directory named by the SSDIR environment variable (if defined), then in the directory named by SFDIR. An unsuccessful search will return a “cannot open” error.

Resultant analysis files are written into the current directory, or to the named directory if a path is included. It is tidy to keep analysis files separate from sound files, usually in a separate directory known to the SADIR variable. Analysis is conveniently run from within the SADIR directory. When an analysis file is later invoked by a Csound generator (adsyn, lpread, pvoc) it is sought first in the current directory, then in the directory defined by SADIR.
Soundfile Formats. Csound can read and write audio files in a variety of formats. Write formats are described by Csound command flags. On reading, the format is determined from the soundfile header, and the data automatically converted to floating-point during internal processing. When Csound is installed on a host with local soundfile conventions (SUN, NeXT, Macintosh) it may conditionally include local packaging code which creates soundfiles not portable to other hosts. However, Csound on any host can always generate and read AIFF files, which is thus a portable format. Sampled sound libraries are typically AIFF, and the variable SSDIR usually points to a directory of such sounds. If defined, the SSDIR directory is in the search path during soundfile access. Note that some AIFF sampled sounds have an audio looping feature for sustained performance; the analysis programs will traverse any loop segment once only.

For soundfiles without headers, an SR value may be supplied by a command flag (or its default). If both header and flag are present, the flag value will over-ride.

When sound is accessed by the audio Analysis programs (below), only a single channel is read. For stereo or quad files, the default is channel one; alternate channels may be obtained on request.

SNDINFO - get basic information about one or more soundfiles.

csound -U sndinfo soundfilenames . . .
or
sndinfo soundfilenames . . .

sndinfo will attempt to find each named file, open it for reading, read in the soundfile header, then print a report on the basic information it finds. The order of search across soundfile directories is as described above. If the file is of type AIFF, some further details are listed first.

EXAMPLE

csound -U sndinfo test Bosendorfer/”BOSEN mf A0 st” foo foo2

where the environment variables SFDIR = /u/bv/sound, and SSDIR = /so/bv/Samples, might produce the following:

util SNDINFO:

/u/bv/sound/test:
srate 22050, monaural, 16 bit shorts, 1.10 seconds
headersiz 1024, datasiz 48500 (24250 sample frames)

/so/bv/Samples/Bosendorfer/BOSEN mf A0 st: AIFF, 197586 stereo samples, base Frq 261.6 (midi 60), sustnLp: mode 1, 121642 to 197454, relesLp: mode 0
AIFF soundfile, looping with modes 1, 0
srate 44100, stereo, 16 bit shorts, 4.48 seconds
headersiz 402, datasiz 790344 (197586 sample frames)

/u/bv/sound/foo:
no recognizable soundfile header

/u/bv/sound/foo2:
couldn't find

HETRO - hetrodyne filter analysis for the Csound adsyn generator.

csound -U hetro [flags] infilename outfilename
or hetro [flags] infilename outfilename

hetro takes an input soundfile, decomposes it into component sinusoids, and outputs a description of the components in the form of breakpoint amplitude and frequency tracks. Analysis is conditioned by the control flags below. A space is optional between flag and value.
-s<srate> sampling rate of the audio input file. This will over-ride the srate of the soundfile header, which otherwise applies. If neither is present, the default is 10000. Note that for adsyn synthesis the srate of the source file and the generating orchestra need not be the same.

-c<channel> channel number sought. The default is 1.

-b<begin> beginning time (in seconds) of the audio segment to be analyzed. The default is 0.0

-d<duration> duration (in seconds) of the audio segment to be analyzed. The default of 0.0 means to the end of the file.
Maximum length is 32.766 seconds.

-f<begfreq> estimated starting frequency of the fundamental, necessary to initialize the filter analysis. The default is 100
(cps).

-h<partials> number of harmonic partials sought in the audio file. Default is 10, maximum 50.

-M<maxamp> maximum amplitude summed across all concurrent tracks.
The default is 32767.

-m<minamp> amplitude threshold below which a single pair of amplitude/frequency tracks is considered dormant and will not contribute to output summation. Typical values: 128 (48 db down
from full scale), 64 (54 db down), 32 (60 db down), 0 (no thresholding). The default threshold is 64 (54 db down).

-n<brkpts> initial number of analysis breakpoints in each amplitude and frequency track, prior to thresholding (-m) and linear breakpoint consolidation. The initial points are spread evenly over the duration. The default is 256.

-l<cutfreq> substitute a 3rd order Butterworth low-pass filter with cutoff frequency cutfreq (in cps), in place of the default averaging comb filter. The default is 0 (don't use).

EXAMPLE
hetro -s44100 -b.5 -d2.5 -h16 -M24000 audiofile.test adsynfile7

This will analyze 2.5 seconds of channel 1 of a file "audiofile.test", recorded at 44.1 KHz, beginning .5 seconds from the start, and place the result in a file "adsynfile7". We request just the first 16 harmonics of the sound, with 256 initial breakpoint values per amplitude or frequency track, and a peak summation amplitude of 24000. The fundamental is estimated to begin at 100 Hz. Amplitude thresholding is at 54 db down.
The Butterworth LPF is not enabled.

FILE FORMAT
The output file contains time-sequenced amplitude and frequency values for each partial of an additive complex audio source. The information is in the form of breakpoints (time, value, time, value, ...) using 16-bit integers in the range 0 - 32767. Time is given in milliseconds, and frequency in Hertz (cps). The breakpoint data is exclusively non-negative, and the values -1 and -2 uniquely signify the start of new amplitude and frequency tracks. A track is terminated by the value 32767.
Before being written out, each track is data-reduced by amplitude thresholding and linear breakpoint consolidation.

A component partial is defined by two breakpoint sets: an amplitude set, and a frequency set. Within a composite file these sets may appear in any order (amplitude, frequency, amplitude ....; or amplitude, amplitude..., then frequency, frequency,...). During adsyn resynthesis the sets are automatically paired (amplitude, frequency) from the order in which they were found. There should be an equal number of each.

A legal adsyn control file could have following format:

```
-1 time1 value1 ... timeK valueK 32767 ; amplitude breakpoints
                                        ; for partial 1
-2 time1 value1 ... timeL valueL 32767 ; frequency breakpoints
                                        ; for partial 1
-1 time1 value1 ... timeM valueM 32767 ; amplitude breakpoints
                                        ; for partial 2
-2 time1 value1 ... timeN valueN 32767 ; frequency breakpoints
                                        ; for partial 2
-2 time1 value1 .....
-2 time1 value1 .....          ; pairable tracks for partials 3 and 4
-1 time1 value1 .....
-1 time2 value1 .....
```

LPANAL - linear predictive analysis for the Csound lp generators

csound -U lpanal [flags] infilename outfilename
or
lpanal [flags] infilename outfilename

lpanal performs both lpc and pitch-tracking analysis on a soundfile to produce a time-ordered sequence of frames of control information suitable for Csound resynthesis. Analysis is conditioned by the control flags below. A space is optional between the flag and its value.

-s<srate> sampling rate of the audio input file. This will over-ride the srate of the soundfile header, which otherwise applies. If neither is present, the default is 10000.

-c<channel> channel number sought. The default is 1.

-b<begin> beginning time (in seconds) of the audio segment to be analyzed. The default is 0.0

-d<duration> duration (in seconds) of the audio segment to be analyzed. The default of 0.0 means to the end of the file.

-p<npoles> number of poles for analysis. The default is 34, the maximum 50.

-h<hopsize> hop size (in samples) between frames of analysis. This determines the number of frames per second (srate / hopsize) in the output control file. The analysis framesize is hopsize * 2 samples. The default is 200, the maximum 500.

-C<string> text for the comments field of the lpfile header.
The default is the null string.

-P<mincps> lowest frequency (in cps) of pitch tracking. -P0 means no pitch tracking.

-Q<maxcps> highest frequency (in cps) of pitch tracking. The narrower the pitch range, the more accurate the pitch estimate.
The defaults are -P70, -Q200.

-v<verbosity> level of terminal information during analysis. 0 = none, 1 = verbose, 2 = debug. The default is 0.

EXAMPLE
lpanal -p26 -d2.5 -P100 -Q400 audiofile.test lpfil22

will analyze the first 2.5 seconds of file "audiofile.test", producing srate/200 frames per second, each containing 26-pole filter coefficients and a pitch estimate between 100 and 400 Hertz. Output will be placed in "lpfil22" in the current directory.

FILE FORMAT
Output is a file comprised of an identifiable header plus a set of frames of floating point analysis data. Each frame contains four values of pitch and gain information, followed by npoles filter coefficients. The file is readable by Csound's lpread.
lpanal is an extensive modification of Paul Lanksy's lpc analysis programs.

PVANAL - Fourier analysis for the Csound pvoc generator

csound -U pvanal [flags] infilename outfilenam
or pvanal [flags] infilename outfilename

pvanal converts a soundfile into a series of short-time Fourier transform (STFT) frames at regular timepoints (a frequency-domain representation). The output file can be used by pvoc to generate audio fragments based on the original sample, with timescales and pitches arbitrarily and dynamically modified. Analysis is conditioned by the flags below. A space is optional between the flag and its argument.

-s<srate> sampling rate of the audio input file. This will over-ride the srate of the soundfile header, which otherwise applies. If neither is present, the default is 10000.

-c<channel> channel number sought. The default is 1.

-b<begin> beginning time (in seconds) of the audio segment to be analyzed. The default is 0.0

-d<duration> duration (in seconds) of the audio segment to be analyzed. The default of 0.0 means to the end of the file.

-n<frmsiz> STFT frame size, the number of samples in each Fourier analysis frame. Must be a power of two, in the range 16 to 16384. For clean results, a frame must be larger than the longest pitch period of the sample. However, very long frames result in temporal "smearing" or reverberation. The bandwidth of each STFT bin is determined by sampling rate / frame size. The default framesize is the smallest power of two that corresponds to more than 20 milliseconds of the source (e.g. 256 points at 10 kHz sampling, giving a 25.6 ms frame).

-w<windfact> Window overlap factor. This controls the number of Fourier transform frames per second. Csound's pvoc will interpolate between frames, but too few frames will generate audible distortion; too many frames will result in a huge analysis file. A good compromise for windfact is 4, meaning that each input point occurs in 4 output windows, or conversely that the offset between successive STFT frames is framesize/4. The default value is 4. Do not use this flag with -h.

-h<hopsize> STFT frame offset. Converse of above, specifying the increment in samples between successive frames of analysis (see also lpcanal). Do not use with -w.

EXAMPLE
pvanal asound pvfile

will analyze the soundfile "asound" using the default frmsiz and windfact to produce the file "pvfile" suitable for use with pvoc.

FILES
The output file has a special pvoc header containing details of the source audio file, the analysis frame rate and overlap.
Frames of analysis data are stored as float, with the magnitude and 'frequency' (in Hz) for the first N/2 + 1 Fourier bins of each frame in turn. 'Frequency' encodes the phase increment in such a way that for strong harmonics it gives a good indication of the true frequency. For low amplitude or rapidly moving harmonics it is less meaningful.

DIAGNOSTICS
Prints total number of frames, and frames completed on every 20th.

AUTHOR: Dan Ellis, dpwe@media-lab.media.mit.edu

Appendix 2: CSCORE

Cscore is a standalone program for generating and manipulating numeric score files. It comprises a number of function subprograms, called into operation by a user-written main program. The function programs augment the C language library functions; they can optionally read standard numeric score files, can massage and expand the data in various ways, then write the data out as a new score file to be read by a Csound orchestra.

The user-written main program is also in C. It is not essential to know the C language well to write a main program, since the function calls have a simple syntax, and are powerful enough to do most of the complicated work. Additional power can come from C later as the need arises.

Events, Lists, and Operations

An event in Cscore is equivalent to one statement of a standard numeric score. It is either created or read in from an existing score file. An event is comprised of an opcode and an array of pfield values stored somewhere in memory. Storage is organized by the following structure:

```
struct event {
    char op;          /* opcode */
    char tnum;
    short pcnt;
    float p[PMAX+1];      /* pfields */
};
```

Any function subprogram that creates, reads, or copies an event function will return a pointer to the storage structure holding the event data. The event pointer can be used to access any component of the structure, in the form of e->op or e->p[n].
Each newly stored event will give rise to a new pointer, and a sequence of new events will generate a sequence of distinct pointers that must themselves be stored. Groups of event pointers are stored in a list, which has its own structure:

```
struct evlist {
    int nslots;      /* size of this list     */
    struct event *e[1];    /* list of event pointers */
};
```

Any function that creates or modifies a list will return a pointer to the new list. The list pointer can be used to access any of its component event pointers, in the form of a->e[n].
Event pointers and list pointers are thus primary tools for manipulating the data of a score file.

Pointers and lists of pointers can be copied and reordered without modifying the data values they refer to. This means that notes and phrases can be copied and manipulated from a high level of control. Alternatively, the data within an event or group of events can be modified without changing the event or list pointers. Cscore

provides a library of programming methods or function subprograms by which scores can be created and manipulated in this way.

In the following summary of Cscore function calls, some simple naming conventions are used:

the symbols e, f are pointers to events (notes);
the symbols a, b are pointers to lists (arrays) of such events;
the letters ev at the end of a function name signify operation on an event;
the letter l at the start of a function name signifies operation on a list.

calling syntax	description
e = createv(n);	create a blank event with n pfields
e = defev("...");	defines an event as per the character string ...
e = copyev(f);	make a new copy of event f
e = getev();	read the next event in the score input file
putev(e);	write event e to the score output file
putstr("...");	write the character string ... to score output
a = lcreat(n);	create an empty event list with n slots
a = lappev(a,e);	append event e to list a
n = lcount(a);	count the events now in list a
a = lcopy(b);	copy the list b (but not the events)
a = lcopyev(b);	copy the events of b, making a new list
a = lget();	read events from score input (to next s or e)
lput(a);	write the events of list a to score output
a = lsepf(b);	separate the f statements from list b into list a
a = lcat(a,b);	concatenate (append) the list b onto the list a
lsort(a);	sort the list a into chronological order by p[2]
a = lxins(b,"...");	extract notes of instruments ... (no new events)
a = lxtimev(b,from,to) ;	extract notes of time-span, creating new events
relev(e);	release the space of event e
lrel(a);	release the space of list a (but not events)
lrelev(a);	release the eventsof list a, and the list space

Writing a Main program.

The general format for a main program is:

```
#include  <csound/cscore.h>
main()
{
   /*  VARIABLE DECLARATIONS     */

   /*  PROGRAM BODY         */
}
```

The include statement will define the event and list structures for the program. The following C program will read from a standard numeric score, up to (but not including) the first s or e statement, then write that data (unaltered) as output.

```
#include  <csound/cscore.h>
main()
{
    struct evlist *a;   /* a is allowed to point to an event list */

    a = lget();        /* read events in, return the list pointer */
    lput(a);      /*  write these events out (unchanged)*/
    putstr("e");        /* write the string e to output */
}
```

After execution of lget(), the variable a points to a list of event addresses, each of which points to a stored event. We have used that same pointer to enable another list function (lput) to access and write out all of the events that were read. If we now define another symbol e to be an event pointer, then the statement

```
e = a->e[4];
```

will set it to the contents of the 4th slot in the evlist structure. The contents is a pointer to an event, which is itself comprised of an array of parameter field values. Thus the term e->p[5] will mean the value of parameter field 5 of the 4th event in the evlist denoted by a. The program below will multiply the value of that pfield by 2 before writing it out.

```
#include  <csound/cscore.h>
main()
{
    struct event  *e;       /* a pointer to an event          */
    struct evlist *a;

    a = lget();         /* read a score as a list of events    */
    e = a->e[4];        /* point to event 4 in event list\fIa\fR */
    e->p[5] *= 2;       /* find pfield 5, multiply its value by 2 */
    lput(a);          /* write out the list of events         */
    putstr("e");        /* add a "score end" statement          */
}
```

Now consider the following score, in which p[5] contains frequency in cps.

```
f 1 0 257 10 1
f 2 0 257 7 0 300 1 212 .8
i 1 1 3 0 440 10000
i 1 4 3 0 256 10000
i 1 7 3 0 880 10000
e
```

If this score were given to the preceding main program, the resulting output would look like this:

```
f 1 0 257 10 1
f 2 0 257 7 0 300 1 212 .8
i 1 1 3 0 440 10000
i 1 4 3 0 512 10000      ; p[5] has become 512 instead of 256.
i 1 7 3 0 880 10000
e
```

Note that the 4th event is in fact the second note of the score.
So far we have not distinguished between notes and function table setup in a numeric score. Both can be classed as events. Also note that our 4th event has been stored in e[4] of the structure.
For compatibility with Csound pfield notation, we will ignore p[0] and e[0] of the event and list structures, storing p1 in p[1], event 1 in e[1], etc. The Cscore functions all adopt this convention.

As an extension to the above, we could decide to use a and e to examine each of the events in the list. Note that e has not preserved the numeral 4, but the contents of that slot. To inspect p5 of the previous listed event we need only redefine e with the assignment

```
e = a->e[3];
```

More generally, if we declare a new variable f to be a pointer to a pointer to an event, the statement

```
f = &a->e[4];
```

will set f to the address of the fourth event in the event list a, and *f will signify the contents of the slot, namely the event pointer itself. The expression

```
(*f)->p[5],
```

like e->p[5], signifies the fifth pfield of the selected event. However, we can advance to the next slot in the evlist by advancing the pointer f. In C this is denoted by f++.
In the following program we will use the same input score. This time we will separate the ftable statements from the note statements. We will next write the three note-events stored in the list a, then create a second score section consisting of the original pitch set and a transposed version of itself. This will bring about an octave doubling.

By pointing the variable f to the first note-event and incrementing f inside a while block which iterates n times (the number of events in the list), one statement can be made to act upon the same pfield of each successive event.

```
#include  <csound/cscore.h>
main()
{
    struct event *e,**f;        /* declarations. see pp.89 in the */
    struct evlist *a,*b;        /* C language programming manual */
    int n;

    a = lget();             /* read score into event list "a" */
    b = lsepf(a);           /* separate f statements */
    lput(b);                /* write f statements out to score */
    lrelev(b);              /* and release the spaces used     */
    e = defev("t 0 120");   /* define event for tempo statement */
    putev(e);               /* write tempo statement to score */
    lput(a);                /* write the notes */
    putstr("s");            /* section end */
    putev(e);               /* write tempo statement again */
    b = lcopyev(a);         /* make a copy of the notes in "a" */
    n = lcount(b);          /* and count the number copied */
    f = &a->e[1];
    while (n--)             /* iterate the following line n times: */
      (*f++)->p[5] *= .5;  /*   transpose pitch down one octave */
      a = lcat(b,a);            /* now add these notes to original pitches
*/
    lput(a);
    putstr("e");
}
```

The output of this program is:

```
f 1 0 257 10 1
f 2 0 257 7 0 300 1 212 .8
t 0 120
i 1 1 3 0 440 10000
i 1 4 3 0 256 10000
i 1 7 3 0 880 10000
s
t 0 120
i 1 1 3 0 440 10000
i 1 4 3 0 256 10000
i 1 7 3 0 880 10000
i 1 1 3 0 220 10000
i 1 4 3 0 128 10000
i 1 7 3 0 440 10000
e
```

Next we extend the above program by using the while statement to look at p[5] and p[6]. In the original score p[6] denotes amplitude. To create a diminuendo in the added lower octave, which is independent from the original set of notes, a variable called dim will be used.

```
#include <csound/cscore.h>
main()
{
    struct event *e,**f;
    struct evlist *a,*b;
    int n, dim;             /* declare new variable as integer */

    a = lget();
    b = lsepf(a);
    lput(b);
    lrelev(b);
    e = defev("t 0 120");
    putev(e);
    lput(a);
    putstr("s");
    putev(e);           /* write out another tempo statement */
    b = lcopyev(a);
    n = lcount(b);
    dim = 0;            /* initialize dim to 0 */
    f = &a->e[1];
    while (n--){
        (*f)->p[6] -= dim;  /* subtract current value of dim */
        (*f++)->p[5] *= .5; /* transpose, move f to next event */
        dim += 2000;        /* increase dim for each note */
    }
    a = lcat(b,a);
    lput(a);
    putstr("e");
}
```

The increment of f in the above programs has depended on certain precedence rules of C. Although this keeps the code tight, the practice can be dangerous for beginners. Incrementing may alternately be written as a separate statement to make it moreclear.

```
    while (n--){
        (*f)->p[6] -= dim;
        (*f)->p[5] *= .5;
        dim += 2000;
        f++;
    }
```

Using the same input score again, the output from this program is:

```
f 1 0 257 10 1
f 2 0 257 7 0 300 1 212 .8
t 0 120
i 1 1 3 0 440 10000
i 1 4 3 0 256 10000
i 1 7 3 0 880 10000
s
t 0 120
i 1 1 3 0 440 10000      ; Three original notes at
i 1 4 3 0 256 10000       ; beats 1,4 and 7 with no dim.
i 1 7 3 0 880 10000
i 1 1 3 0 220 10000      ; three notes transposed down one octave
i 1 4 3 0 128 8000       ; also at beats 1,4 and 7 with dim.
i 1 7 3 0 440 6000
e
```

In the following program the same three-note sequence will be repeated at various time intervals. The starting time of each group is determined by the values of the array cue. This time the dim will occur for each group of notes rather than each note.
Note the position of the statement which increments the variable dim outside the inner while block.

```
#include  <csound/cscore.h>

int cue[3]={0,10,17};           /* declare array of 3 integers */

main()
{
    struct event *e, **f;
    struct evlist *a, *b;
    int n, dim, cuecount, holdn;    /* declare new variables */

    a = lget();
    b = lsepf(a);
    lput(b);
    lrelev(b);
    e = defev("t 0 120");
    putev(e);
    n = lcount(a);
    holdn = n;                  /* hold the value of "n" to reset below
*/
    cuecount = 0;               /* initilize cuecount to "0" */
    dim = 0;
    while (cuecount <= 2) {         /* count 3 iterations of inner
"while" */
        f = &a->e[1];           /* reset pointer to first event of list "a"
*/
        n = holdn;              /* reset value of "n" to original note
        count */
        while (n--) {
        (*f)->p[6] -= dim;
        (*f)->p[2] += cue[cuecount];    /* add values of cue */
        f++;
        }
        printf("; diagnostic:  cue = %d\n", cue[cuecount]);
        cuecount++;
```

```
        dim += 2000;
        lput(a);
    }
    putstr("e");
}
```

Here the inner while block looks at the events of list a (the notes) and the outer while block looks at each repetition of the events of list a (the pitch group repetitions). This program also demonstrates a useful trouble-shooting device with the printf function. The semi-colon is first in the character string to produce a comment statement in the resulting score file. In this case the value of cue is being printed in the output to insure that the program is taking the proper array member at the proper time. When output data is wrong or error messages are
encountered, the printf function can help to pinpoint the problem.

Using the identical input file, the C program above will generate:

```
f 1 0 257 10 1
f 2 0 257 7 0 300 1 212 .8
t 0 120

; diagnostic:  cue = 0
i 1 1 3 0 440 10000
i 1 4 3 0 256 10000
i 1 7 3 0 880 10000

; diagnostic:  cue = 10
i 1 11 3 0 440 8000
i 1 14 3 0 256 8000
i 1 17 3 0 880 8000

; diagnostic:  cue = 17
i 1 28 3 0 440 4000
i 1 31 3 0 256 4000
i 1 34 3 0 880 4000
e
```

Further development of these scores will lead the composer to techniques of score manipulation which are similar to serial techniques of composition. Pitch sets may be altered with regard to any of the parameter fields. The programing allows transpositions, time warping, pitch retrograding and dynamic controls, to name a few.

Compiling a Cscore program

A Cscore program example.c is compiled and linked with its library modules by the command:

```
$ cc -o myprog example.c -lcscore
```

The resulting executable file myprog is run by typing:

```
$ myprog              (no input, output printed onthe screen)
$ myprog < scorin     (input score named \fIscorin\fR, output on screen)
$ myprog < scorin > scorout   (input as above, output into a file)
```

Appendix 3: An Instrument Design Tutorial

by
Richard Boulanger
Berklee College of Music

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.

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 16kHz, a control rate of 1kHz, and then calculated the number of samples in each control period using the formula: ksmps = sr / kr.

```
sr   =      16000
kr   =      1000
ksmps  =      16
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.

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 Ñ i, k, or a 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
```

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 waveshape (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 outputÑhere the audio rate. We could have named the result anything (e.g. "asig") as long as it began with the letter "a".

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.

Tutorial Instruments

Toot 1: Play One Note

For this and all instrument examples below, 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 "gen1" 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 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
           e ndin

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 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 4: Chorusing

Next we'll animate the basic sound by mixing it with two slightly detuned 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 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)
      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*.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

    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 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 seventeen 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

  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 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              ; calculateremaining 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

  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  str  dur  amp  frq     atk  rel  vibrt  vbdpt  vibdel  startwav  endwav
crosst
  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
```

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 requiredÑthe 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

; ins  strt   dur     amp   skip   atk   rel   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 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 ratesÑsetup, 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       =  18900      ; toot9.orc
kr       =  945
ksmps    =  20
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

 asig      reverb      garvbsig,    irvbtime  ; put global signal into
reverb
     outs asig, 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 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 above.

```
garvbsig  init   0

instr 10                ; toot10.orc
 iattack   =  .01
 irelease  =  .2
 iwhite    =  10000
 idur      =  p3
 iamp      =  p4
 isweepstart  =  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
```

```
; ins    strt    dur  rvbtime                ;
toot10.sco
  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 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.

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
```

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
```

Tempo. The unit of time in a Csound score is the beatÑnormally 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 (see the Csound manual). 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 soundfiles:

```
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
```

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.

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
```

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  vibrt  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 12: Tables & 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
        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

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
```

Toot 13: Spectral Fusion

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

```
   a1    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.

```
   a2    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.

```
a1   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

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

;ins 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
```

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

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 this chapter, 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.

The directory 'morefiles' contains examples of the classical designs of Risset and Chowning. Detailed discussions of these instruments can be found in Charles Dodge's and Thomas Jerse's Computer Music textbook. This text is the key to getting the most out of these instrumental models and their innovative approaches to signal processing. Also recommended are the designs of Russell Pinkston. They demonstrate techniques for legato phrasing, portamento, random vibrato, and random sequence generation. His instrument representing Dx7 OpCode[a] Editor/Librarian patches is a model for bringing many wonderful sounds into your orchestra.

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 - March 1991 - Boston, Massachusetts - USA

Appendix 4: An 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 main Csound manual) is:

```
ar fof xamp xfund xform koct kband kris kdur kdeciolaps 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 to0.

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 = 22050
    kr = 441
    ksmps = 50

    instr    1
a1  fof  15000, 200, 650, 0, 40, .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 = risetime, duration and decaytime (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, 2, 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 skirtwidth at -40dB. Increasing kband increases the local envelope's exponential decay rate, thus shortening it and increasing the -6dbB spectral region. Increasing kris (the
envelope attack time) inversely makes the -40dB spectral region smaller.

The next example changes first the bandwidth then the skirtwidth.
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 in the manual proper).

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.

Appendix 5: Csound for the Macintosh

by
Bill Gardner
MIT Media Lab

Introduction

This document describes the Macintosh verson of the Csound program and assumes the reader is already familiar with the Csound program as described in the Csound Users Manual. Csound is primarily intended for the UNIX operating system and hence its operation is specified through command line arguments. The Macintosh version of Csound surrounds this mechanism with the standard Macintosh user interface primitives, e.g. menus and dialog boxes. After the user specifies the Csound input files and options using the Macintosh user interface, Macintosh Csound creates the UNIX command line and invokes Csound appropriately.
All subsequent Csound output is directed to a console window (or optionally to a listing file). Output sound files are created in Digidesign's Sound Designer II format or optionally in AIFF format.

Orchestra and Score file selection dialog

When Csound is launched, it automatically brings up the main file selection dialog. This dialog has fields for the required input orchestra and score files, the output sample file, and optionally a MIDI file and output listing file. Only the output file name may be explicitly typed in, the other fields must be filled by clicking on the corresponding Select button, which will bring up a standard Macintosh file selection dialog. Because the orchestra and score file names usually differ only in the filename extension (".orc" for orchestra files and ".sco" for score files), Csound only requires that you select one of the two; the other file name will be automatically formed by changing the extension appropriately. If an output file is selected (in the Options dialog), then the output file name is similarly created with the extension ".snd".
Users familiar with Csound will recall that Csound expects all sound files to live in a single directory called the SFDir (for sound file directory). On UNIX, this directory is specified via a UNIX shell environment variable. On the Macintosh, the user must select this directory. This can be done by clicking on the SFDir button next to the output file text item. This brings up the Sound File Directory selection dialog which shows the current sound file directory. When shipped, this will be blank. The sound file directory must be set up properly in order that output files may be created. Clicking on the Select button brings up a standard file selection dialog. Navigate to the directory you want to use for sound files, and then click on the Save button.
Finally, click on the Save Settings button so that Csound will remember this directory when it is run in the future.

If an input MIDI file is desired, click on the MIDI file checkbox. This brings up a file selection dialog which can be used to select a MIDI file.

If a listing file is desired, click on the listing file checkbox.
This bring up a file selection dialog which can be used to specify an output listing file. If a listing file is selected, Csound will route almost all output to the listing file. Certain messages will still appear in the console window.

The Smp Fmt and Options buttons bring up other dialogs for setting the Csound output sample format and options, respectively. These are described later in this document.

After all the files and options have been set up, click on the OK button to run Csound. If the files and options are incorrectly set up, Csound will report an error by printing an appropriate message to the console window. After the score file has been processed, Csound will display the message "*** PRESS MOUSE BUTTON TO EXIT ***". Pressing the mouse button will cause Csound to exit back to the Macintosh Finder. It is not possible to process multiple files without relaunching Csound.

Clicking on the Cancel button causes the file selection dialog to disappear. The dialog can be brought back by selecting the Choose Orchestra and Score menu item in the Csound menu. Note that cancelling the dialog does not cause Csound to forget the settings of the various options. This is particularly useful if you want to select extra options explicity by using the "Enter command line..." menu item.

Csound menu

This section describes the menu items available in the Csound menu. To access the menu items the file selection dialog must be cancelled as described above.

Choose Orchestra and Score...

Selecting this menu item brings up the main file selection dialog described above.

Options...

Selecting this menu item brings up the options dialog containing checkboxes for each Csound option. These are each described below:

Note amplitudes

When checked, causes information regarding note amplitudes to be displayed. Corresponds to the -m1 option in UNIX Csound.

Samples out of range

When checked, causes information regarding out of range samples to be displayed. Corresponds to the -m2 option in UNIX Csound.

Warnings

When checked, causes information regarding out of range samples to be displayed. Corresponds to the -m4 option in UNIX Csound.

No table graphics

When checked, suppresses the display of Csound wavetables. Corresponds to the -d option in UNIX Csound.

Diagnostic messages

When checked, causes diagnostic messages to be displayed. Corresponds to the -v option in UNIX Csound.

Initialize processing only

When checked, causes Csound to only perform initialization of orchestras and no other processing. Corresponds to the -I option in UNIX Csound.

No sound output

When checked, suppresses the output of a sound file. Corresponds to the -n option in UNIX Csound.

Change file types

When checked, Csound will change the file creator field of the selected orchestra and score files. This lets the Macintosh Finder know that the files are associated with the Csound application. Thus, if you subsequently double-click on one of these files from the Finder, Csound will be executed using the selected file as input.

Output Sample Format...

Selecting this menu item brings up the output sample format dialog which controls the format of the output sample file. All sound files are created as Digidesign Sound Designer II format files, unless the -A option is specified in the command line, which causes Csound to create AIFF (Audio Interchange File Format) files. The radio buttons select the sample format and default to 16-bit integer. The 8-bit integer, 8-bit a-law, 8-bit µ-law, 16-bit integer, 32-bit integer, and 32-bit float formats correspond to the -c, -a, -u, -s, -l, and -f options, respectively, in UNIX Csound. Note that 8-bit a-law format is not supported. If the no header checkbox is checked, this suppresses the output of a sound file header; only the raw samples are output. This corresponds to the -h option in UNIX Csound. The blocksize controls how many samples are accumulated before writing to the output sample file. This corresponds to the -b option in UNIX Csound. Note that all input files must be either Digidesign Sound Designer II format, AIFF format or raw16-bit samples.

Sound File Directory...

Selecting this menu item brings up the sound file directory dialog. This dialog is described above in the file selection dialog section.

Sampled Sound Directory...

Selecting this menu item brings up the sampled sound directory dialog. This dialog allows the user to specify the directory where Csound will look for sampled sound files to be loaded into function tables. This corresponds to the SSDir shell environment variable in UNIX Csound. If no directory is specified, Csound will look for sampled sounds in the sound file directory, described earlier.

Enter Command Line...

Selecting this menu item brings up the command line dialog. The command line dialog is used for directly entering a UNIX command line to invoke Csound. This is useful for specifying obscure arguments to Csound which are not otherwise supported in the Macintosh interface. (The AIFF file option -A is one such option). The dialog comes up with a command line that corresponds to all currently selected files and options. The command line appears in a Macintosh text edit field for editing.
When specifying file name arguments that contain spaces, enclose the argument in double-quotes. Click on OK to invoke Csound with the specified arguments, or click on Cancel to exit the dialog without executing Csound.

Save Settings

Selecting this menu item causes all option settings to be remembered for the next time Csound is executed.

Quit

Selecting this menu item causes Csound to exit to the Macintosh Finder.

This documentation written on February 10, 1992 by Bill Gardner, MIT Media Laboratory, Music and Cognition Group, 20 Ames Street, Cambridge MA 02139.
internet: billg@media-lab.media.mit.edu

Appendix 6: Adding your own Cmodules to Csound

If the existing Csound generators do not suit your needs, you can write your own modules in C and add them to the run-time system.

When you invoke Csound on an orchestra and score file, the orchestra is first read by a table-driven translator 'otran' and the instrument blocks converted to coded templates ready for loading into memory by 'oload' on request by the score reader.
To use your own C-modules within a standard orchestra you need only add an entry in otran's table and relink Csound with your own code.

The translator, loader, and run-time monitor will treat your module just like any other provided you follow some conventions.
You need a structure defining the inputs, outputs and workspace, plus some initialization code and some perf-time code. Let's put an example of these in two new files, newgen.h and newgen.c:

```
typedef struct {                /*  newgen.h - define a structure */
    OPDS    h;                  /* required header */
    float       *result, *istrt, *incr, *itime, *icontin; /* addr outarg,
inargs  */
    float       curval, vincr;          /* private dataspace */
    long        countdown;              /* ditto       */
} RMP;

#include "cs.h"             /*  newgen.c  -  init and perf code */
#include "newgen.h"

void rampset(p)             /* at note initialization:     */
 register RMP  *p;
{
    if (*p->icontin == 0.)
    p->curval = *p->istrt;  /* optionally get new start value */
    p->vincr = *p->incr / esr;  /* set s-rate increment per sec.   */
    p->countdown = *p->itime * esr; /* counter for itime seconds
*/
}

void ramp(p)                    /* during note performance:   */
 register RMP  *p;
{
    register float *rsltp = p->result;  /* init an output array pointer
*/
    register int nn = ksmps;            /* array size from orchestra */
    do {
        *rsltp++ = p->curval;           /* copy current value to ouput
*/
        if (--p->countdown >= 0)        /* for the first itime seconds,
*/
            p->curval += p->vincr;  /*   ramp the value  */
    } while (--nn);
}
```

Now we add this module to the translator table entry.c, under the opcode name rampt:

```
#include "newgen.h"
void rampset(), ramp();

/*  opcode  dspace  thread  outarg  inargs  isub  ksub
asub  */
{ "rampt",  S(RMP),  5,  "a", "iiio",  rampset,  NULL,
ramp },
```

Finally we relink Csound to include the new module. Under Unix this means changing the Makefile in three places:

1. Add the name newgen.o to the variable OBJS.
2. Add the name newgen.h as a dependency for entry.o
3. Create a new dependency, newgen.o: newgen.h

Now run 'make csound'. If your host is a Macintosh, simply add newgen.h and newgen.c to one of the Csound segments and invoke the C compiler.

The above actions have added a new generator to the Csound language. It is an audio-rate linear ramp function which modifies an input value at a user-defined slope for some period.
A ramp can optionally continue from the previous note's last value. The Csound manual entry would look like:

ar rampt istart, islope, itime [, icontin]

istart - beginning value of an audio-rate linear ramp.
Optionally overridden by a continue flag.

islope - slope of ramp, expressed as the y-interval change per second.

itime - ramp time in seconds, after which the value is held for the remainder of the note.

icontin (optional) - continue flag. If zero, ramping will proceed from input istart . If non-zero, ramping will proceed from the last value of the previous note. The default value is zero.

The file newgen.h includes a one-line list of output and input parameters. These are the ports through which the new generator will communicate with the other generators in an instrument.
Communication is by address, not value, and this is a list of pointers to floats. There are no restrictions on names, but the input-output argument types are further defined by character strings in entry.c (inargs, outargs). Inarg types are commonly x, a, k, and i, in the normal Csound manual conventions; also available are o (optional, defaulting to 0), p (optional, defaulting to 1). Outarg types include a, k, i and s (asig or ksig). It is important that all listed argument names be assigned a corresponding argument type in entry.c. Also, i-type args are valid only at initialization time, and other-type args are available only at perf time. Subsequent lines in the RMP structure declare the work space needed to keep the code re-entrant. These enable the module to be used multiple times in multiple instrument copies while preserving all data.

The file newgen.c contains two subroutines, each called with a pointer to the uniquely allocated RMP structure and its data.
The subroutines can be of three types: note initialization, k-rate signal generation, a-rate signal generation. A module normally requires two of theseÑinitialization, and either k-rate or a-rate subroutinesÑwhich become inserted in various threaded lists of runnable tasks when an instrument is activated. The thread-types appear in entry.c in two forms: isub, ksub and asub names; and a threading index which is the sum of isub=1, ksub=2, asub=4. The code itself may reference global variables defined in cs.h and oload.c, the most useful of which are:

```
float   esr       user-defined sampling rate
float   ekr       user-defined control rate
float   ensmps    user-defined ksmps
int  ksmps        user-defined ksmps
int  nchnls       user-defined nchnls
int  odebug       command-line -v flag
int  initonly     command-line -I flag
int  msglevel     command-line -m level
float   pi, twopi obvious constants
float   tpidsr       twopi / esr
float   sstrcod   special code for string arguments
```

Function tables

To access stored function tables, special help is available. The newly defined structure should include a pointer

```
FUNC    *ftp;
```

initialized by the statement

```
ftp = ftpfind(p->ifuncno);
```

where float *ifuncno is an i-type input argument containing the ftable number. The stored table is then at ftp->ftable, and other data such as length, phase masks, cps-to-incr converters, are also accessed from this pointer. See the FUNC structure in cs.h, the ftfind() code in fgens.c, and the code for oscset() and koscil() in ugens2.c.

Additional space

Sometimes the space requirement of a module is too large to be part of a structure (upper limit 65535 bytes), or it is dependent on an i-arg value which is not known until initialization.

Additional space can be dynamically allocated and properly managed by including the line

```
AUX CH      auxch;
```

in the defined structure (*p), then using the following style of code in the init module:

```
if (p->auxch.auxp == NULL)
auxalloc(npoints * sizeof(float), &p->auxch);
```

The address of this auxilliary space is kept in a chain of such spaces belonging to this instrument, and is automatically managed while the instrument is being duplicated or garbage-collected during performance. The assignment

```
char *auxp = p->auxch.auxp;
```

will find the allocated space for init-time and perf-time use.
See the LINSEG structure in ugens1.h and the code for lsgset() and klnseg() in ugens1.c.

File sharing

When accessing an external file often, or doing it from multiple places, it is often efficient to read the entire file into memory. This is accomplished by including the line

```
MEMFIL    *mfp;
```

in the defined structure (*p), then using the following style of code in the init module:

```
if (p->mfp == NULL)
   p ->mfp = ldmemfile(filname);
```

where char *filname is a string name of the file requested. The data read will be found between

```
(char *)  p->mfp->beginp;        and       (char *) p->mfp->endp;
```

Loaded files do not belong to a particular instrument, but are automatically shared for multiple access. See the ADSYN structure in ugens3.h and the code for adset() and adsyn() in ugens3.c.

String arguments

To permit a quoted string input argument (float *ifilnam, say) in our defined structure (*p), assign it the argtype S in entry.c, include another member char *strarg in the structure, insert a line

```
TSTRARG( "rampt",  RMP) \
```

in the file oload.h, and include the following code in the init module:

```
if (*p->ifilnam == sstrcod)
   strcpy(filename, unquote(p->strarg));
```

See the code for adset() in ugens3.c, lprdset() in ugens5.c, and pvset() in ugens8.c.

Appendix 7: A CSOUND QUICK REFERENCE

VALUE CONVERTERS

ftlen(x)	(init-rate args only)
int(x)	(init- or control-rate args only)
frac(x)	" "
dbamp(x)	" "
i(x)	(control-rate arg; only)
abs(x)	(no rate restriction)
exp(x)	" "
log(x)	" "
sqrt(x)	" "
sin(x)	" "
cos(x)	" "
ampdb(x)	" "

PITCH CONVERTERS

octpch(pch)	(init- or control-rate args only)
pchoct(oct)	" "
cpspch(pch)	" "
octcps(cps)	" "
cpsoct(oct)	(no rate restriction)

PROGRAM CONTROL

igoto	label
tigoto	label
kgoto	label
goto	label
if	ia R ib igoto label
if	ka R kb kgoto label
if	ia R ib goto label
timout	istrt, idur, label

MIDI CONVERTERS

iamp	**ampmidi**	iscal[, ifn]
kaft	**aftouch**	iscal
kchpr	**chpress**	iscal
kbend	**pchbend**	iscal
ival	**midictrl**	inum
kval	**midictrl**	inum

SIGNAL GENERATORS

kr	**line**	ia, idur1, ib
ar	**line**	ia, idur1, ib
kr	**expon**	ia, idur1, ib
ar	**expon**	ia, idur1, ib
kr	**linseg**	ia, idur1, ib[, idur2, ic[...]]
ar	**linseg**	ia, idur1, ib[, idur2, ic[...]]
kr	**expseg**	ia, idur1, ib[, idur2, ic[...]]
ar	**expseg**	ia, idur1, ib[, idur2, ic[...]]
kr	**phasor**	kcps[, iphs]
ar	**phasor**	xcps[, iphs]
ir	**table**	indx, ifn[, ixmode][, ixoff][, iwrap]
ir	**tablei**	indx, ifn[, ixmode][, ixoff][, iwrap]
kr	**table**	kndx, ifn[, ixmode][, ixoff][, iwrap]
kr	**tablei**	kndx, ifn[, ixmode][, ixoff][, iwrap]
ar	**table**	andx, ifn[, ixmode][, ixoff][, iwrap]
ar	**tablei**	andx, ifn[, ixmode][, ixoff][, iwrap]
kr	**oscil1**	idel, kamp, idur, ifn
kr	**oscil1i**	idel, kamp, idur, ifn
kr	**oscil**	kamp, kcps, ifn[, iphs]
kr	**oscili**	kamp, kcps, ifn[, iphs]
ar	**oscil**	xamp, xcps, ifn[, iphs]
ar	**oscili**	xamp, xcps, ifn[, iphs]
ar	**foscil**	xamp, kcps, kcar, kmod, kndx, ifn[, iphs]
ar	**foscili**	xamp, kcps, kcar, kmod, kndx, ifn[, iphs]
ar1 [,ar2]	**loscil**	xamp, kcps, ifn[, ibas][, imod1,ibeg1,iend1] [,imod2,ibeg2,iend2]
ar	**buzz**	xamp, xcps, knh, ifn[, iphs]
ar	**gbuzz**	xamp, xcps, knh, kih, kr, ifn[, iphs]
ar	**adsyn**	kamod, kfmod, ksmod, ifilcod
ar	**pvoc**	ktimpnt, kfmod, ifilcod[, ispecwp]
ar	**fof**	xamp, xfund, xform, koct, kband, kris, kdur, kdec, iolaps, ifna, ifnb, itotdur[, iphs][, ifmode]
ar	**pluck**	kamp, kcps, icps, ifn, imeth [, iparm1, iparm2]
kr	**rand**	xamp[, iseed]

kr	**randh**	kamp, kcps[, iseed]
kr	**randi**	kamp, kcps[, iseed]
ar	**rand**	xamp[, iseed]
ar	**randh**	xamp, xcps[, iseed]
ar	**randi**	xamp, xcps[, iseed]

SIGNAL MODIFIERS

kr	**linen**	kamp, irise, idur, idec
ar	**linen**	xamp, irise, idur, idec
kr	**linenr**	kamp, irise, idec, iatdec
ar	**linenr**	xamp, irise, idec, iatdec
kr	**envlpx**	kamp, irise, idur, idec, ifn, iatss, iatdec[, ixmod]
ar	**envlpx**	xamp, irise, idur, idec, ifn, iatss, iatdec[, ixmod]
kr	**port**	ksig, ihtim[, isig]
ar	**tone**	asig, khp[, istor]
ar	**atone**	asig, khp[, istor]
arreson		asig, kcf, kbw[, iscl, istor]
ar	**areson**	asig, kcf, kbw[, iscl, istor]
krmsr, krmso, kerr,	**kcpslpread**	ktimpnt,ifilcod[, inpoles][,ifrmrate]
ar	**lpreson**	asig
ar	**lpfreson**	asig, kfrqratio
kr	**rms**	asig[, ihp, istor]
nr	**gain**	asig, krms[, ihp, istor]
ar	**balance**	asig, acomp[, ihp, istor]
kr	**downsamp**	asig[, iwlen]
ar	**upsamp**	ksig
ar	**interp**	ksig[, istor]
kr	**integ**	ksig[, istor]
ar	**integ**	asig[, istor]
kr	**diff**	ksig[, istor]
ar	**diff**	asig[, istor]
kr	**samphold**	xsig, kgate[, ival, ivstor]
ar	**samphold**	asig, xgate[, ival, ivstor]
ar	**delayr**	idlt[, istor]
	delayw	asig
ar	**delay**	asig, idlt[, istor]
ar	**delay1**	asig[, istor]
ar	**deltap**	kdlt
ar	**deltapi**	xdlt
ar	**comb**	asig, krvt, ilpt[, istor]
ar	**alpass**	asig, krvt, ilpt[, istor]
ar	**reverb**	asig, krvt[, istor]

OPERATIONS USING SPECTRAL DATA TYPES

dsig	**octdown**	xsig, iocts, isamps[, idisprd]
wsig	**noctdft**	dsig, iprd, ifrqs, iq[, ihann, idbout, idsines]
wsig	**specscal**	wsigin, ifscale, ifthresh
wsig	**specaddm**	wsig1, wsig2[, imul2]
wsig	**specdiff**	wsigin
wsig	**specaccm**	wsigin
wsig	**specfilt**	wsigin, ifhtim
	specdisp	wsig, iprd[, iwtflg]
ksum	**specsum**	wsig[, interp]

SENSING & CONTROL

ktemp		kin, iprd, imindur, imemdur, ihp, ithresh,
tempest		ihtim, ixfdbak, istartempo, ifn[, idisprd, itweek]
kx, ky	**xyin**	iprd, ixmin, ixmax, iymin, iymax[,ixinit, iyinit]
	tempo	ktempo, istartempo

SOUND INPUT & OUTPUT

a1	**in**	
a1, a2	**ins**	
a1, a2, a3, a4	**inq**	
a1	**soundin**	ifilcod[,iskptim][, iformat]
a1, a2	**soundin**	ifilcod[,iskptim][, iformat]
a1, a2, a3, a4	**soundin**	ifilcod[,iskptim][, iformat]
	out	asig
	outs1	asig
	outs2	asig
	outs	asig1, asig2
	outq1	asig
	outq2	asig
	outq3	asig
	outq4	asig
	outq	asig1, asig2, asig3, asig4
a1, a2, a3, a4	**pan**	asig, kx, ky, ifn[, imode][,ioffset]

SIGNAL DISPLAY

print	iarg[, iarg,...]
display	xsig, iprd[, iwtflg]
dispfft	xsig, iprd, iwsiz[, iwtyp][, idbouti][,iwtflg]

END OF Csound MANUAL

Log of changes introduced from 3.15.10

CSOUND for MSDOS
CSOUND for ATARI ST
John Fitch
School of Mathematical Sciences
University of Bath
Bath BA2 7AY
England
Tel: +44-1225-826820
FAX: +44-1225-826492
E-mail: jpff@maths.bath.ac.uk
or J.P.Fitch@bath.ac.uk

(also Codemist Ltd, Tel/FAX: +44-1225-837430)

csound_286.zip
csound_fpt_286.zip
csound_386.zip
csound_fpt_386.zip
csound_486.zip
csound_src.zip

These files are the executables for CSound for 286/386/486 machines running MS-DOS. There are versions built for a plain machine and for a machine with a floating point co-processor.

Also, the files *.ttp are Csound for the Atari ST. See below for ATARI notes.

NOTE: The 486 version does not seem to work on an 486SX, for which the 386 version should be used.

Note: I have not tried all these versions myself, as I have a 386 without co-processor.

There is a mailing list kept for this version; to join send mail to pcsound-request@maths.bath.ac.uk. There is also a Csound mailing list for discussion of any aspect of the system, and a WWW page at http://www.leeds.ac.uk/music/Man/c_front.html

Local Changes:
==========
It attempts to do graphics. It is supposed to adjust to your graphics system. See below for notes on how to set screen types explicitly. I also have graphics in PVANAL and LPANAL with a -g option. In CSOUND itself there is a pause before and after each graph. This can be turned off if the environment variable CSNOSTOP is set to YES.

The WAV file format was all new for the PC, but is now in the main sources. To ensure you get WAV sound files either use the -W option, or set an environment variable SFOUTYP to WAV in your AUTOEXEC.BAT
The system on a 386 or 486 will recognise the output file devaudio as an attempt to us a sound card for direct output (SoundBlaster or compatible). This is not yet finished, and seems to be limited to less than 14KHz sampling, and 8 bit samples in mono. I am attempting to improve that. On my slow 386 the machine cannot keep up with generating in time, so there is a chopping effect. Your mileage may vary.

I have introduced a local version number; currently I have v3.20.10.
The major number refers to MIT's current version (3 is the beta), the 20 is my sequence number for the PC, and the 10 is the version of the "real time" support.

There are three new utilities to scale for amplitude, a mixer for mixing sound files together, and a mkgraph program to write envelopes for these two.

Version 3.20.10:
A number of changes in the MIDI area. A number of new generators added, including Butterworth filters, vdelay, multitap and reverb2.
Also granular synthesis generator and stochastic generators. There is an envelope-following generator as well. See Appendix 7 for details of these generators. These are largely the work of Paris Smaragdis.

Version 3.19.10:
Not released

Version 3.18.10:
Not released

Version 3.17.10:
Created new utility mkgraph which creates envelope files using the mouse for drawing. These files can be used by mixer and scale to provide more flexible gain control on sound files.
New utility ENVEXT for create an envelope file from a sound file.

Version 3.16.10:
Mixer and scale can take envelope files

Version 3.15.10:
Experimentally I have attempted to read the device sbmidi as a MIDI input for use with the -M option. I have no idea if this works.
When using -o devaudio (also can use -o dac or -o sblst) it will force the format to be -c or -s.
The mixer can now take varying numbers of channels as input and can
include some or all channels, and can direct input channel n to output channel m. As the scaling can be negative as well as positive this incorporates removal of information as well. The syntax is not good, but inspiration is not with me this weekend.

Copyright:
==========
The systems are the product of the MIT Media Laboratory, and this is their copyright notice:

Interested parties should note that CSound is a system for creation of sound, and is not a MIDI sequencer.

The systems built are described briefly below.

CSOUND EXE
digital audio processing and sound synthesis

csound [flags] orchfile scorefile

Csound is an environment in which a "scorefile" or external event sequence can invoke arbitrarily complex signal-processing "instruments" to produce sound. Audio may be displayed during its creation, and the resulting sound sent to an on-line audio device or to an intermediate soundfile for later playback. Flags include
-C use Cscore processing of scorefile
-I I-time only orch run
-n no sound onto disk
-i fnam sound input filename
-o fnam sound output filename (if fnam is devaudio, dac or sblst use directly)
-b N sample frames (or -kprds) per software sound I/O buffer
-B N samples per hardware sound I/O buffer
-A create an AIFF format output soundfile
-W create a WAV format output soundfile
-h no header on output soundfile
-c 8-bit signed_char sound samples
-a alaw sound samples
-u ulaw sound samples
-s short_int sound samples
-l long_int sound samples
-f float sound samples
-r N orchestra srate override
-v verbose orch translation
-m N tty message level. Sum of: 1=note amps, 2=out-of-range msg, 4=warnings, 8=SB mesages
-d suppress all displays
-g suppress graphics, use ascii displays
-S score is in Scot format
-t N use uninterpreted beats of the score, initially at tempo N
-L dnam read Line-oriented realtime score events from device 'dnam'
-M dnam read MIDI realtime events from device 'dnam' (must be sbmidi)
-F fnam read MIDIfile event stream from file 'fnam'
-P N MIDI sustain pedal threshold (0 - 128)
-R continually rewrite header while writing soundfile (WAV/AIFF)
-H print a heartbeat character at each soundfile write
-N notify (ring the bell) when score or miditrack is done
-T terminate the performance when miditrack is done

flag defaults: csound -s -otest -b1024 -B1024 -m7 -P128

CSCORE.LIB
Cscore is a program for generating and manipulating numeric score files.
It comprises a number of function subprograms, called into operation by a user-written main program.
The function programs augment the C language library functions; they can optionally read standard numeric score files, can massage and expand the data in various ways, then write the data out as a new score file to be read by a Csound orchestra.

EXTRACT.EXE
Program for extracting parts of a work. Not tested in PC version

HETRO.EXE

hetrodyne filter analysis for Csound adsyn module
hetro [flags] [fundamental] [filename]

hetro takes as input a file containing amplitude samples of some sound over time (it is assumed that the samples are evenly spaced in time) and decomposes that sound into a set of harmonically related sine waves with time varying amplitude and phase.

LPCANAL.EXE

Paul Lansky's software for linear predictive analysis and pitch tracking, adapted for Csound.

lpcanal [-p<n> -i<n> -s<t> -d<t> -o<file> -C<str> -P<frq> -Q<frq>] soundfile

lpcanal is the new experimental combination of the old anallpc and ptrack. It performs linear predictive analysis and pitch tracking on monaural 16bit fixed point soundfiles. If a -g flag is used then a graphical display is given of some of the output as it is being computed.

PVANAL.EXE

Fourier analysis module for Csound PVOC unit generator

pvanal [-n frame-size] [-o overlap | -i increment] \
inputSoundFile outputFFTFile

pvanal converts a playable sample (a time-domain representation) into
a series of short-time Fourier transform (STFT) frames centred at regular points throughout the file (a frequency-domain representation). The output file can then be used as the data for the PVOC unit generator in Csound to generate notes based on the original sample, but with their timescales and pitches arbitrarily and dynamically modified. If a -g flag is used then a graphical display is given of some of the output as it is being computed.

SCOT.EXE

Scot is a scoring program to prepare input for CSound. It is rather complex and initial testing on the PC suggests that I have not got it correct yet.

SCSORT.EXE

Stand-alone sorting of sound files
Not tested on PC

SNDINFO.EXE

Reads the header of a sound file to identify type, duration etc

sndinfo soundfile

SCALE.EXE

As well as doing the same as SNDINFO this utility reports on the maximum amplitude, and can generate a new soundfile with the amplitude scaled by a floating point value.

scale [-flags] soundfile

Legal flags are:

-o fnam	sound output filename
-A	create an AIFF format output soundfile
-W	create a WAV format output soundfile
-h	no header on output soundfile
-c	8bit signed_char sound samples
-a	alaw sound samples
-u	ulaw sound samples
-s	short_int sound samples
-l	long_int sound samples
-f	float sound samples
-F fpnum	amount to scale amplitude
-F fname	envelope file for scaling
-R	continually rewrite header while writing soundfile (WAV/AIFF)
-H	print a heartbeat character at each soundfile write
-N	notify (ring the bell) when score or miditrack is done

flag defaults: scale -s -otest -F 0.0

If scale is 0.0 then reports maximum possible scaling; otherwise scale
and generate a new soundfile

MIXER.EXE

This utility can mix together a number of sound files (up to 20 at present) with different starting times and with scaling on each file.

mixer [-flags] soundfile [-flags] soundfile ...

Legal flags are:

-o	fnam sound output filename
-A	create an AIFF format output soundfile
-W	create a WAV format output soundfile
-h	no header on output soundfile
-c	8-bit signed_char sound samples
-a	alaw sound samples
-u	ulaw sound samples
-s	short_int sound samples
-l	long_int sound samples
-f	float sound samples
-F fpnum	amount to scale amplitude of next sound file
-F fname	an envelope file for scaling
-R	continually rewrite header while writing soundfile (WAV/AIFF)
-H	print a heartbeat character at each soundfile write
-N	notify (ring the bell) when score or miditrack is done
-S int	Sample at which to insert next sound file
-T fpnum	Time at which to insert next sound file
-1 -2 -3 -4	include named channel
-^ n m	include channel n and output as channel m

Defaults are: mixer -s -otest -F1.0 -S0

MIXER can also be used for some echo effects.

MKGRAPH.EXE

A small utility to creat envelope files for MIXER and SCALE. Type ? when the program is running to get all the controls.

mkgraph [-v] [envfile] [-o outname]

Default is mkgraph -o newgraph. If an envfile is given it is loaded and can be editted. envfile and outname can be the same.

ENVEXT.EXE

Given a sound file it creates an envelope file with an approximation to the envelope of the sound file.

envext [-w time] [-o file] soundfile

Defaults are

envext -w 0.25 -o newenv

Graphics:
========

The graphics is just for the display of waveforms. The full specification of the graphics used says that it tries auto-determining the graphics on the machine. This can sometimes fail, so it reads the environment variable FG_DISPLAY, and if set as below it uses that kind of graphics.

Value	Type
=====	====
GCAHIRES	GCA 640 x 200 x 2
GCAMEDRES	GCA 320 x 200 x 4
EGACOLOR	EGA 640 x 200 x 16
EGAECD	Enhanced EGA 640 x 350 x 16
EGALOWRES	
EGAMONO	
EVGAHIRES	Everest EVGA board

HERCFULL	Hercules 2 pages 2 colour
HERCHALF	Hercules 1 page 2 colour
ORCHIDPROHIRES	VGA type
PARADISEHIRES	VGA type
TOSHIBA	Toshiba 3100-- 640 x 400 x 2
TRIDENTHIRES	Trident 800 x 600 x 16
VEGAVGAHIRES	Video 7 vega VGA board
VESA6A	VESA mode 0x6a
VESA2	VESA mode 0x102
VGA11	IBM VGA mode 0x11
VGA12	IBM VGA mode 0x12
VGA13	IBM VGA mode 0x13
8514A	IBM 8514A display adapter

Virtual Memory:
==========

The system uses virtual memory on 386/486. The limits on memory size are the minimum of

1. Free disk space + code size
2. 256 times your extended memory
3. 3.5Gbytes (!)

You should set up the environment variable TMP or TEMP to the disk to use for swap space. If this is not set it looks at disks C:, D:, ... looking for the largest free space. That gives the limit of space.

Reporting Bugs:
==========

Please mail (or possibly FAX) me reports on any bugs and shortcomings of the PC version. I will endeavour to fix or assist, but it is only fair to warn you that this is not either of my jobs, and so it may be lower in priorities. But I am interested in widening the availability of CSound.

The system has been built with Zortech's C++ Compiler, with its royalty-free DOS extender, x and z modes, and FlashTech's virtual memory and graphics. We (as Codemist) use this system for a commercial product, and it seems satisfactory, and reasonably trouble
free.

DOS6:
====

It is known that the DOS6 memory manager does not obey the full rules, and so interferes with Csound. I now have a fix for this, and the
corrected version is now on the server, but there do still seem to be problems. The old fix was to ensure that in your AUTOEXEC.BAT or
CONFIG.SYS that if there is a call to

emm386 -noems

in it that you change this to read

emm386

This should fix things for now. Or remove the line!

80286 - version
==========

The files in the 286 versions also need the program ZPM.EXE, which is provided, in your search path.

John Fitch
School of Mathematical Sciences
University of Bath
Bath BA2 7AY
5QR
United Kingdom
Tel: +44-1225-826820
FAX: +44-1225-826492

Codemist Ltd
"Alta", Horsecombe Vale
Combe Down, Bath BA2
United Kingdom
Tel: +441225-837430
FAX: +44-1225-837430

Appendix 7 : Newest Csound opcodes

by
Paris Smaragdis
Berklee College of Music

This appendix describes recent additions to Csound. These additions include a granular synthesis synthesizer, a new set of filters, a new variable delay, a multitap delay, a new reverb, an envelope follower, various noise generators, a power function generator and two gen routines, GEN20 and GEN21.

1) Granular synthesizer.

ar **grain** xamp, xpitch, xdens, kampoff, kpitchoff, kgdur, igfn, iwfn, imgdur

Generates granular synthesis textures.

INITIALIZATION

igfn, igdur - igfn is the ftable number of the grain waveform. This can be just a sine wave or a sampled sound of any length. Each grain will start from a random table position and sustain for igdur seconds.

iwfn - Ftable number of the amplitude envelope used for the grains (see also GEN20).

imgdur - Maximum grain duration in seconds. This the biggest value to be assigned on kgdur.

PERFORMANCE

xamp - Total amplitude of the sound.

xpitch - Grain frequency in cps.

xdens - Density of grains measured in grains per second. If this is constant then the output is synchronous granular synthesis, very similar to fof. If xdens has a random element (like added noise), then the result is more like asynchronous granular synthesis.

kampoff - Maximum amplitude deviation from kamp. This means that the maximum amplitude a grain can have is kamp + kampoff and the minimum is kamp. If kampoff is set to zero then there is no random amplitude for each grain.

kpitchoff - Maximum pitch deviation from kpitch in cps. Similar to kampoff.

kgdur - Grain duration in seconds. The maximum value for this should be declared in imgdur. If kgdur at any point becomes greater than imgdur, it will be truncated to imgdur.

2) Butterworth filters.

ar	**butterhp**	asig, kfreq
ar	**butterlp**	asig, kfreq
ar	**butterbp**	asig, kfreq, kband
ar	**butterbr**	asig, kfreq, kband

Implementations of second-order hipass, lopass, bandpass and bandreject Butterworth filters.

PERFORMANCE

These new filters are butterworth second-order IIR filters. They are slightly slower than the original filters in Csound, but they offer an almost flat passband and very good precision and stopband attenuation.

asig - Input signal to be filtered.

kfreq - Cuttoff or center frequency for each of the filters.

kband - Bandwidth of the bandpass and bandreject filters.

EXAMPLE

```
asig    rand        10000           ; White noise signal
alpf    butterlp    asig, 1000      ; cutting frequencies above1K
ahpf    butterhp    asig, 500       ; passing frequencies above
500Hz
abpf    butterbp    asig, 2000, 100 ; passing only 1950 to 2050 Hz
abrf    butterbr    asig, 4500, 200 ; cutting only 4400 to 4600 Hz
```

3) Vdelay

ar **vdelay** asig, adel, imaxdel

This is an interpolating variable time delay, it is not very different from the existing implementation (deltapi), it is only easier to use.

INITIALIZATION

imaxdel - Maximum value of delay in samples. If adel gains a value greater than imaxdel it is folded around imaxdel. This should not happen.

PERFORMANCE

With this unit generator it is possible to do Doppler effects or chorusing and flanging.

asig - Input signal.

adel - Current value of delay in samples. Note that linear functions have no pitch change effects. Fast changing values of adel will cause discontinuities in the waveform resulting noise.

Example

```
 f1 0 8192 10 1

ims = 100                       ; Maximum delay time in msec
a1    oscil   10000, 1737, 1    ; Make a signal
a2    oscil   ims/2, 1/p3, 1    ; Make an LFO
a2  = a2 + ims/2                ; Offset the LFO so that it is positive
a3    vdelay  a1, a2, ims       ; Use the LFO to control delay time
   out    a3
```

Two important points here. First, the delay time must be always positive. And second, even though the delay time can be controlled in
k-rate, it is not advised to do so, since sudden time changes will create clicks.

4) Multitap delay

ar **multitap** asig, itime1, igain1, itime2, igain2 . . .

Multitap delay line implementation.

INITIALIZATION

The arguments itime and igain set the position and gain of each tap. The delay line is fed by asig.

Example:

```
a1    oscil       1000, 100, 1
a2    multitap    a1, 1.2, .5, 1.4, .2
out   a2
```

This results in two delays, one with length of 1.2 and gain of .5, and one with length of 1.4 and gain of .2.

5) Reverb2

ar **reverb2** asig, ktime, khdif

This is a reverberator consisting of 6 parallel comb-lowpass filters being fed into a series of 5 allpass filters.

PERFORMANCE

The input signal asig is reverberated for ktime seconds. The parameter khdif controls the high frequency diffusion amount. The values of khdif should be from 0 to 1. If khdif is set to 0 the all the frequencies decay with the same speed. If khdif is 1, high frequencies decay faster that lower ones.

Example:

```
a1    oscil       10000, 100, 1
a2    reverb2     a1, 2.5, .3
out   a1 + a2 * .2
```

This results in a 2.5 sec reverb with faster high frequency attenuation.

6) Envelope follower

kr **follow** asig, idt

Envelope follower unit generator.

INITIALIZATION

idt - This is the period, in seconds, that the average amplitude of asig is reported. If the frequency of asig is low then idt must be large (more than half the period of asig).

PERFORMANCE
asig - This is the signal from which to extract the envelope.

Example

```
k1    line    0, p3, 30000   ; Make k1 a simple envelope
a1    oscil   k1, 1000, 1    ; Make a simple signal using k1
ak1   follow  a1, .02        ; ak1 is now like k1
a2    oscil   ak1, 1000, 1   ; Make a simple signal using   ak1
   out   a2                  ; Both a1 and a2 are the same
```

To avoid zipper noise, by discontinuities produced from complex envelope tracking, a lowpass filter could be used, to smooth the estimated envelope.

7) Noise generators

All of the following opcodes operate in i-, k- and a-rate. The output rate depends on the first letter of the opcode, a for a-rate, k for k-rate and i for i-rate.

xlinrand krange - Linear distribution random number generator. Krange is the range of the random numbers [0 - krange). Outputs only
positive numbers.

xtrirand krange - Same as above only ouputs both negative and positive numbers.

xexprand krange - Exponential distribution random number generator.
krange is the range of the random numbers [0 - krange). Outputs only positive numbers.

xbexprnd krange - Same as above, only extends to negative numbers too with an exponential distribution.

xcauchy kalpha - Cauchy distribution random number generator. Kalpha controls the spread from zero (big kalpha => big spread). Outputs both positive and negative numbers.

xpcauchy kalpha - Same as above, ouputs positive numbers only.

xpoisson klambda - Poisson distribution random number generator. klambda is the mean of the distribution. Outputs only positive numbers.

xgauss krange - Gaussian distribution random number generator. Krange is the range of the random numbers (-krange - 0 - krange). Outputs both positive and negative numbers.

xweibull ksigma, ktau - Weibull distribution random number generator.
ksigma scales the spread of the distribution and ktau, if greater than one numbers near ksigma are favored, if smaller than one small values are favored and if t equals 1 the distribution is exponential.
Outputs only positive numbers.

xbeta krange, kalpha, kbeta - Beta distribution random number generator. krange is the range of the random numbers [0 - krange). If kalpha is smaller than one, smaller values favor values near 0. If kbeta is smaller than one, smaller values favor values near krange.
If both kalpha and kbeta equal one we have uniform distribution. If both kalpha and kbeta are greater than one we have a sort of gaussian distribution. Outputs only positive numbers.

For more detailed explanation of these distributions, see:

1. C. Dodge - T.A. Jerse 1985. Computer music. Schirmer books. pp.265 - 286

2. D. Lorrain. “A panoply of stochastic cannons”. In C. Roads, ed. 1989. Music machine . Cambridge, Massachusetts: MIT press, pp. 351 - 379.

Examples:

```
a1  atrirand  32000              ; Audio noise with triangle distribution
k1  kcauchy  10000               ; Control noise with Cauchy dist.
i1  ibetarand  30000, .5, .5     ; i-time random value, beta dist.
```

8) Power functions

```
ir    ipow    iarg, kpow
kr    kpow    karg, kpow, [inorm]
ar    apow    aarg, kpow, [inorm]
```

Computes xarg to the power of kpow and scales the result by inorm.

INITIALIZATION

inorm - The number to divide the result (default to 1). This is especially useful if you are doing powers of a- or k- signals where samples out of range are extremely common!

iarg - If you are using ipow this is the base.

PERFORMANCE

karg - If you are using kpow this is the base.

aarg - If you are using apow this is the base.

EXAMPLES:

```
1.   i2t2   ipow        2,2          ; Computes 2^2.

2.  kline  line        0, 1, 4
    kexp   kpow   kline, 2, 4
```

This feeds a linear function to kpow and scales that to the line’s peak value. The output will be an exponential curve with the same range as the input line.

```
3.   iamp ipow          10, 2
     a1  oscil    iamp, 100, 1
     a2  apow     a1, 2, iamp
       out a2
```

This will output a sine with its negative part folded over the amp axis. The peak value will be iamp = 10^2 = 100.

9) GEN20

This subroutine generates functions of different windows. These windows are usually used for spectrum analysis or for grain envelopes.

f # time size 20 window max opt

size - number of points in the table. Must be a power of 2 (+ 1).

window - Type of window to generate.

1 - Hamming
2 - Hanning
3 - Bartlett (triangle)
4 - Blackman (3 - term)
5 - Blackman - Harris (4 - term)
6 - Gaussian
7 - Kaiser
8 - Rectangle
9 - Sinc

max - For negative p4 this will be the absolute value at window peak point. If p4 is positive or p4 is negative and p6 is missing the table will be post-rescaled to a maximum value of 1.

opt - Optional argument required by the Kaiser window.

Examples:

f 1 0 1024 20 5

This creates a function which contains a 4 - term Blackman - Harris window with maximum value of 1.

f 1 0 1024 -20 2 456

This creates a function that contains a Hanning window with a maximum value of 456.

f 1 0 1024 -20 1

This creates a function that contains a Hamming window with a maximum value of 1.

f 1 0 1024 20 7 1 2

This creates a function that contains a Kaiser window with a maximum
value of 1. The extra argument specifies how `open’ the window is, for example a value of 0 results in a rectangular window and a value of 10 in a Hamming like window.

10) GEN21

This generates tables of different random distributions. (see also noise
generators, above).

f # time size 21 type lvl arg1 arg2

Time and size are the usual Gen function arguments. Type defines the distribution to be used.

1 - Uniform
2 - Linear
3 - Triangular

4 - Exponential
5 - Biexponential
6 - Gaussian
7 - Cauchy
8 - Positive Cauchy
9 - Beta
10 - Weibull
11 - Poison

Of all these cases only 9 (Beta) and 10 (Weibull) need extra arguments. Beta needs two arguments and Weibull one.

Examples:

```
f1 0 1024 21 1          ; Uniform (white noise)
f1 0 1024 21 6          ; Gaussian
f1 0 1024 21 9 1 1 2    ; Beta (note that level precedes arguments)
f1 0 1024 21 10 1 2     ; Weibull
```

All of the above additions were designed by the author between May and December 1994, under the supervision of Dr. Richard Boulanger.

This appendix was written on 20 December 1994 by Paris Smaragdis, Berklee College of Music.
Internet: psmaragdis@aol.com

AUTHOR: Greg Sullivan, sullivan@aussie.enet.dec.com
(Based on algorithm given in 'Elements Of Computer Music', by F. Richard Moore).

CVANAL - Impulse Response Fourier Analysis for CONVOLVE operator

csound -U cvanal [flags] infilename outfilename

cvanal converts a soundfile into a single Fourier transform frame. The
output file can be used by the CONVOLVE operator to perform Fast Convolution between an input signal and the original impulse response.
Analysis is conditioned by the flags below. A space is optional between the flag and its argument.

-s<rate> sampling rate of the audio input file. This will over-ride the srate of the soundfile header, which otherwise applies. If neither is present, the default is 10000.

-c<channel> channel number sought. If omitted, the default is to process all channels. Ifa value is given, only the selected channel will be processed.

-b<begin> beginning time (in seconds) of the audio segment to be analysed. The default is 0.0

-d<duration> duration (in seconds) of the audio segment to be analysed.
The default of 0.0 means to the end of the file.

EXAMPLE

cvanal asound cvfile

will analyse the soundfile "asound" to produce the file "cvfile" for the
use with CONVOLVE.

HINT: To use data that is not already contained in a soundfile, a soundfile converter that accepts text files may be used to create a standard audio file. E.g, the .DAT format for SOX. This is useful for implementing FIR filters.

FILES

The output file has a special CONVOLVE header, containing details of the source audio file. The analysis data is stored as 'float', in rectangular (real/imaginary) form.

NOTE: The analysis file is NOT system independent! Ensure that the original impulse recording/data is retained. If/when required, the analysis file can be recreated.

AUTHOR: Greg Sullivan, sullivan@aussie.enet.dec.com
(Based on algorithm given in 'Elements Of Computer Music', by F. Richard Moore.

CONVOLVE unit generator:

ar1[,ar2[,ar3[,ar4]]] **convolve** ain,ifilcod,channel

Output is the convolution of signal ain and the impulse response contained in ifilcod. Note that it is considerably more efficient to use one instance of the operator when processing a mono input to create stereo, or quad, outputs.

INITIALISATION

ifilcod - integer or character-string denoting an impulse response data file. An integer denotes the suffix of a file convolve.m; a character string (in double quotes) gives a filename, optionally a full pathname.
If not a fullpath, the file is sought first in the the current directory, then in the one given by the environment variable SADIR (if defined).
The data file contains the Fourier transform of an impulse response.
Memory usage depends on the size of the data file, which is read and held entirely in memory during computation, but which is shared by multiple calls.
channel - integer denoting the channel of the impulse response to be used for the convolution. 0 (the default) means to use all channels.
For multi-channel output, the number of channels in the impulse response must match the number of output signals.

PERFORMANCE

convolve implements Fast Convolution. The output of this operator is delayed with respect to the input. The following formulas should be used to calculate the delay:

For (1/kr) <= IRdur:
Delay = ceil(IRdur * kr) / kr
For (1/kr) > IRdur:
Delay = IRdur * ceil(1/(kr*IRdur))
Where:
kr = Csound control rate
IRdur = duration, in seconds, of impulse response
ceil(n) = smallest integer not smaller than n

One should be careful to also take into account the initial delay, if any, of the impulse response. For example, if an impulse response is created from a recording, the soundfile may not have the initial delay included. Thus, one should either ensure that the soundfile has the correct amount of zero padding at the start, or, preferably, compensate for this delay in the orchestra. (the latter method is more efficient). To compensate for the delay in the orchestra, _subtract_ the initial delay from the result calculated using the above formula(s), when calculating the required delay to introduce into the 'dry' audio path.
For typical applications, such as reverb, the delay will be in the order of 0.5 to 1.5 seconds, or even longer. This renders the current implementation unsuitable for real time applications.
It could conceivably be used for real time filtering however, if the number of taps is is small enough.

Example:
- Create frequency domain impulse response file:

```
c:\> csound -Ucvanal l1_44.wav l1_44.cv
```

- Determine duration of impulse response. For high accuracy, determine the number of sample frames in the impulse response soundfile, and then compute the duration with:
duration = (sample frames)/(sample rate of soundfile)

This is due to the fact that the SNDINFO utility only reports the duration to the nearest 10ms. If you have a utility that reports the duration to the required accuracy, then you can simply use the reported value directly.

```
c:\> sndinfo l1_44.wav
length = 60822 samples, sample rate = 44100

Duration = 60822/44100 = 1.379s.
```

- Determine initial delay, if any, of impulse response.
If the impulse response has not had the initial delay removed, then you can skip this step. If it has been removed, then the only way you will know the initial delay is if the information has been provided separately.
For this example, let's assume that the initial delay is 60ms. (0.06s)

- Determine the required delay to apply to the dry signal, to align it with the convolved signal:

```
If kr = 441:
1/kr = 0.0023, which is <= IRdur (1.379s), so:
Delay1 = ceil(IRdur * kr) / kr
    = ceil(608.14) / 441
    = 609/441
    = 1.38s

Accounting for the initial delay:

delay2 = 0.06s
Total delay = delay1 - delay2
            = 1.38 - 0.06
            = 1.32s
```

- Create .orc file, e.g:

```
----CUT----

;Simple demonstration of CONVOLVE operator, to apply reverb.
      sr = 44100
      kr = 441
      ksmps = 100
      nchnls = 2
      instr   1
imix = 0.22 ; Wet/dry mix. Vary as desired.
          ; NB: 'Small' reverbs often require a much higher
          ; percentage of wet signal to sound interesting. 'Large'
          ; reverbs seem require less. Experiment! The wet/dry mix is
           ; very important- a small change can make a large
difference.

ivol = 0.9  ; Overall volume level of reverb. May need to adjust
           ; when wet/dry mix is changed, to avoid clipping.

idel = 1.32 ; Required delay to align dry audio with output of
            ; convolve.
            ; This can be automatically calculated within the orc file,
            ; if desired.
adry        soundin "anechoic.wav"          ; input (dry) audio
awet1,awet2   convolve adry,"l1_44.cv"    ; stereo convolved (wet)
                                            ; audio
adrydel       delay   (1-imix)*adry,idel ; Delay dry signal, to align it
with
                          ; convolved signal. Apply level
                          ; adjustment here too.
   outs    ivol*(adrydel+imix*awet1),ivol*(adrydel+imix*awet2)
                          ; Mix wet & dry signals, and output
      endin
```

---CUT----
The granule unit generator (Allan Lee) is more complex than grain (above), but does add new possibilities. This is a shorten manual, without the pictures,
NAME
granule - Granular synthesis unit generator for Csound.

SYNOPSIS

granule xamp ivoice iratio imode ithd ifn ipshift igskip igskip_os ilength kgap igap_os kgsize igsize_os iatt idec [iseed] [ipitch1] [ipitch2] [ipitch3] [ipitch4] [ifnenv]

DESCRIPTION
granule is a Csound unit generator which employs a wavetable as input to produce granularly synthesized audio output. Wavetable data may be generated by any of the gen subroutines such as gen01 which reads an audio data file into a wavetable. This enable a sampled sound to be used as the source for the grains. Up to 128 voices are implemented internally.The maximum number of voices can be increased by redefining the variable MAXVOICE in the grain4.h file. granule has a build-in random number generator to handle all the random offset parameters.
Thresholding is also implemented to scan the source function table at initialization stage. This facilitates features such as skipping silence passage between sentences.
The characteristics of the synthesis are controlled by 22 parameters. xamp is the amplitude of the output and it can be either audio rate or control rate variable. All parameters with prefix i must be valid at Init time, parameters with prefix k can be either control or Init values.

SUMMARY OF PARAMETERS

xamp - amplitude.
ivoice - number of voices.
iratio - ratio of the speed of the gskip pointer relative to output audio sample rate. eg. 0.5 will be half speed.
imode - +1 grain pointer move forward (same direction of the gskip pointer), -1 backward (oppose direction to the gskip pointer) or 0 for random.
ithd - threshold, if the sampled signal in the wavetable is smaller then ithd, it will be skipped.
ifn - function table number of sound source.
ipshift - pitch shift control. If ipshift is 0, pitch will be set randomly up and down an octave. If ipshift is 1, 2, 3 or 4, up to four different pitches can be set amount the number of voices definded in ivoice. The optional parameters ipitch1, ipitch2, ipitch3 and ipitch4 are used to quantify the pitch shifts.
igskip - initial skip from the beginning of the function table in sec.
igskip_os - gskip pointer random offset in sec, 0 will be no offset.
ilength - length of the table to be used starting from igskip in sec.
kgap - gap between grains in sec.
igap_os - gap random offset in % of the gap size, 0 gives no offset.
kgsize - grain size in sec.
igsize_os - grain size random offset in % of grain size, 0 gives no offset.
iatt - attack of the grain envelope in % of grain size.
idec - decade of the grain envelope in % of grain size.
[iseed] - optional, seed for the random number generator, default is 0.5.
[ipitch1], [ipitch2], [ipitch3], [ipitch4] - optional, pitch shift parameter, used when ipshift is set to 1, 2, 3 or 4. Time scaling technique is used in pitch shift with linear interpolation between data points. Default value is 1, the original pitch.

EXAMPLE

Listing of orchestra file:

```
sr = 44100
kr = 4410
ksmps = 10
nchnls = 2
instr 1
;
k1 linseg 0,0.5,1,(p3-p2-1),1,0.5,0
```

```
a1 granule
p4*k1,p5,p6,p7,p8,p9,p10,p11,p12,p13,p14,p15,p16,p17,p18,p19,
p20,p21,p22,p23,p24
a2                                                    granule
p4*k1,p5,p6,p7,p8,p9,p10,p11,p12,p13,p14,p15,p16,p17,p18,p19,
p20+0.17,p21,p22,p23,p24
outs a1,a2
endin
```

Listing of score file:

```
;f statement read sound file sine.aiff in SFDIR directory into f-table 1
f 1 0 524288 1 "sine.a iff" 1 0
i1 0 10 2000 64 0.5 0 0 1 4 0 0.005 10 0.01 50 0.02 50 30 30 0.39 1
1.42 0.29 2
e
```

The above example reads a sound file called sine.aiff into wavetable number 1 with 524,288 samples. It generates 10 seconds of stereo audio output using the wavetable. In the orchestra file, all parameters required to control the synthesis are passed from the score file. A linseg function generator is used to generate an envelope with 0.5 second of linear attack and decade. Stereo effect is generated by using different seeds for the two granule function calls. In the example, 0.17 is added to p20 before passing into the second granule call to ensure that all of the random offset events are different from the first one.
In the score file, the parameters are interpreted as:
p5 (ivoice) the number of voices is set to 64
p6 (iratio) is set to 0.5, it scan the wavetable at half of the speed of the audio output rate
p7 (imode) is set to 0, the grain pointer only move forward
p8 (ithd) is set to 0, skipping the thresholding process
p9 (ifn) is set to 1, function table number 1 is used
p10 (ipshift) is set to 4, four different pitches are going to be generated
p11 (igskip) is set to 0 and p12 (igskip_os) is set to 0.005, no skipping into the wavetable and a 5 mSec random offset is used
p13 (ilength) is set to 10, 10 seconds of the wavetable is to be used
p14 (kgap) is set to 0.01 and p15 (igap_os) is set to 50, 10 mSec gap with 50% random offset is to be used
p16 (kgsize) is set to 0.02 and p17 (igsize_os) is set to 50, 20 mSec grain with 50% random offset is used
p18 (iatt) and p19 (idec) are set to 30, 30% of linear attack and decade is applied to the grain
p20 (iseed) seed for the random number generator is set to 0.39
p21 - p 24 are pitches set to 1 which is the original pitch, 1.42 which is a 5th up, 0.29 which is a 7th down and finally 2 which is an octave up.

Csound is developed by Barry L. Vercoe at the Experimental Music Studio, Media Laboratory, M.I.T., Cambridge, Massachusetts.

ATARI Csound
==========
The files are
___ csound.ttp
___ extract.ttp
___ hetro.ttp
___ lpanal.ttp
___ pvanal.ttp
___ scale.ttp
___ scsort.ttp
___ sndinfo.ttp

I do not have any of the standard compression programs for the Atari at present, so these are raw binary files.
This is an initial port of 3.14, with no support for graphics or MIDI. The code is not yet optimised, and has only been subjected to limited testing. If there is sufficient interest I will optimise and extend, but my Atari is currently failing to boot. The code does not assume the existence of a floating point co- processor, and is built for the 68000 (lowest common denominator). Please report any bugs or comments to me. I am using LatticeC and the sources and scripts are available from me, assuming I can read the Atari disk...

There is a better-supported commercially available Atari Csound from CDP.
They also have a large suite of music synthesis programs on Atari and PC.

Contact Tom Endrich:
_tendrich@cix.compulink.co.uk
_Tel: +44-1904-613299
_Composers' Desktop Project_11 Kilburn Road_York YO1 4DF_England

---

Release Notes for v3.44
==========

Version 3.44 is mainly a collection of new opcodes, together with a few small fixes.

Summary
=======

a) Phase Vocoding Opcodes:
ktableseg, ktablexseg, voscili, vpvoc, pvread, pvcross, pvbufread, pvinterp

b) Tuning Opcodes:
cps2pch, cpsxpch

c) 3-D Sound Opcode:
hrtfer

d) Time Stretching Opcode:
sndwarp

e) Non-linear Filter Opcode:
nlfilt

f) New format for LPC analysis:
Allows for pole stabalisation, and two new opcodes, lpslot, lpinterp
for interpolation between different analyses

g) Bug Fixes:
Reading of numbers in Event reading fixed
Comments allowed after e in scores
Obscure bad case in opening files
Minor bug in vdelay fixed

h) Permanent Graphs (Fabio Bertolotti):
A new option (-G) ensures that the graphs are saved as a PostScript file, with the same name as the soundfile with .eps as the extention. This has been in use by teachers for a while.

h) Code Changes (PC):
Improvements in output to Multimedia-implemented DAC
Recognise file names starting a: etc
Non-stop feature in Windows

and a number of internal code improvements which I doubt you care about.

DOCUMENTATION ON NEW OPCODES
==========

a) PVOC related unit generators added by Richard Karpen, 1992-1996

kfreq, kamp **pvread** ktimpnt, ifile, ibin

pvbufread ktimpnt, ifile
ar **pvinterp** ktimpnt, kfmod, ifile, kfreqscale1, kfreqscale2, ampscale1, kampscale2, kfreqinterp, kampinterp

ar **pvcross** ktimpnt, kfmod, ifile, kamp1, kamp2, [ispecwp]
tableseg ifn1, idur1, ifn2[, idur2, ifn3[...]]
tablexseg ifn1, idur1, ifn2[, idur2, ifn3[...]]
ar **vpvoc** ktimpnt, kfmod, ifile, [ispecwp]

DESCRIPTION

pvread reads from a pvoc file and returns the frequency and amplitude from a single anaylsis channel or bin. The eturned values can be used anywhere else in the csound instrument. For example, one can use them as arguments to an oscillator to synthesize a single component from an analyzed signal or a bank of pvreads can be used to resynthesize the analyzed sound using additive synthesis by passing the frequency and magnitude values to a bank of oscillators.

pvbufread reads from a pvoc file and makes the retrieved data available to any following pvinterp and pvcross units that appear in an instrument before a subsequent pvbufread (just as lpread and lpreson work together). The data is passed internally and the unit has no output of its own. pvinterp and pvcross allow the interprocessing of two phase vocoder analysis files prior to the resynthesis which these units do also. Both of these units receive data from one of the files from a previously called pvbufread unit. The other file is read by the pvinterp and/or pvcross units. Since each of these units has its own time-pointer the analysis files can be read at different speeds and directions from one another. pvinterp does not allow for the use of the ispecwp process as with the pvoc and vpvoc units.

pvinterp interpolates between the amplitudes and frequencies, on a bin by bin basis, of two phase vocoder analysis files (one from a previously called pvbufread unit and the other from within its own argument list), allowing for user defined transitions between analyzed sounds. It also allows for general scaling of the amplitudes and frequencies of each file separately before the interpolated values are calculated and sent to the resynthesis routines. The kfmod argument in pvinterp performs its frequency scaling on the frequency values after their derivation from the separate scaling and subsequent interpolation is performed so that this acts as an overall scaling value of the new frequency components.

pvcross applies the amplitudes from one phase vocoder analysis file to the data from a second file and then performs the resynthesis. The data is passed, as described above, from a previously called pvbufread unit. The two k-rate amplitude arguments are used to scale the amplitudes of each files separately before they are added together and used in the resynthesis (see below for further explanation). The frequencies of the first file are not used at all in this process. This unit simply allows for cross-synthesis through the application of the amplitudes of the spectra of one signal to the frequencies of a second signal. Unlike pvinterp, pvcross does allow for the use of the ispecwp as in pvoc and vpvoc.

tableseg and **tablexseg** are like linseg and expseg but interpolate between values in a stored function tables. The result is a new function table passed internally to any following vpvoc or voscili which occurs before a subsequent tableseg or tablexseg (much like lpread/lpreson pairs work). The uses of these are described below under vpvoc and (see also voscili which allows the use of these units in an interpolating oscillator).

vpvoc is identical to pvoc except that it takes the result of a previous tableseg or tablexseg and uses the resulting function table (passed internally to the vpvoc), as an envelope over the magnitudes of the analysis data channels. The result is spectral enveloping. The function size used in the tableseg should be framesize/2, where framesize is the number of bins in the phase vocoder analysis file that is being used by the vpvoc. Each location in the table will be used to scale a single analysis bin. By using different functions for ifn1, ifn2, etc.. in the tableseg, the spectral envelope becomes a dynamically changing one.

ARGUMENTS

ifile is the pvoc number (n in pvoc.n) or the name in quotes of the analysis file made using pvanal. See pvoc documentation in Csound manual.

kfreq and kamp are the outputs of the pvread unit. These values, retrieved from a phase vocoder analysis file, represent the values of frequency and amplitude from a single analysis channel specified in the ibin argument. Interpolation between analysis frames is performed at k-rate resolution and dependent of course upon the rate and direction of ktimpnt.

ktimpnt, kfmod, and ispecwp used in pvread and vpvoc are exactly the same as for pvoc (see above description of pvinterp for its special use of kfmod).

ibin is the number of the analysis channel from which to return frequency in cps and magnitude.

kfreqscale1, kfreqscale2, kampscale1, kampscale2 are used in pvinterp to scale the frequencies and amplitudes stored in each frame of the phase vocoder analysis file. kfreqscale1 and kampscale1 scale the frequencies and amplitudes of the data from the file read by the previously called pvbufread (this data is passed internally to the pvinterp unit). kfreqscale2 and kampscale2 scale the frequencies and amplitudes of the file named by ifile in the pvinterp argument list and read within the pvinterp unit. By using these arguments it is possible to adjust these values before applying the interpolation.
For example, if file1 is much louder than file2, it might be desirable to scale down the amplitudes of file1 or scale up those of file2 before interpolating. Likewise one can adjust the frequencies of each to bring them more in accord with one another (or just the opposite of course!) before the interpolation is performed.

kfreqinterp and kampinterp, used in pvinterp, determine the interpolation distance between the values of one phase vocoder file and the values of a second file. When the value of kfreqinterp is 0, the frequency values will be entirely those from the first file (read by the pvbufread), post scaling by the kfreqscale1 argument. When the value of kfreqinterp is 1 the frequency values will be those of the second file (read by the pvinterp unit itself), post scaling by kfreqscale2. When kfreqinterp is between 0 and 1 the frequency values will be calculated, on a bin, by bin basis, as the percentage between each pair of frequencies (in other words, kfreqinterp=.5 will cause the frequencies values to be half way between the values in the set of data from the first file and the set of data from the second file). kampinterp1 and kampinterp2 work in the same way upon the amplitudes of the two files. Since these are k-rate arguments, the percentages can change over time making it possible to create many kinds of transitions between sounds.

ifn1, ifn2, ifn3, etc... in tableseg and tablexseg are stored functions, created from an f-card in the numeric notelist. ifn1, ifn2, and so on, MUST be the same size.

idur1, idur2, etc...in tableseg and tablexseg are the durations during which interpolation from one table to the next will take place.

SIMPLE EXAMPLES

The example below shows the use pvread to synthesize a single component from a phase vocoder analysis file. It should be noted that the kfreq and kamp outputs can be used for any kind of synthesis, filtering, processing, and so on.

```
ktime   line 0, p3, 3 kfreq, kamp pvread ktime, “pvoc.file”, 7 ; read
                                        ; data from 7th analysis bin.
asig    oscili kamp, kfreq, 1           ; function 1 is a stored sine
```

The below shows an example using pvbufread with pvinterp to interpolate between the sound of an oboe and the sound of a clarinet. The value of kinterp returned by a linseg is used to determine the timing of the transitions between the two sounds. The interpolation of frequencies and amplitudes are controlled by the same factor in this example, but for other effects it might be interesting to not have them sychnronized in this way. In this example the sound will begin as a clarinet, transform into the oboe and then return again to the clarinet sound. The value of kfreqscale2 is 1.065 because the oboe in this case is a semitone higher in pitch than the clarinet and this brings them approximately to the same pitch. The value of kampscale2 is .75 because the analyzed clarinet was somewhat louder than the

analyzed oboe. The setting of these two parameters make the transition quite smooth in this case, but such adjustments are by no means necessary or even advocated.

```
ktime1      line  0, p3, 3.5 ; used as index in the “oboe.pvoc” data file
ktime2      line  0, p3, 4.5 ; used as index in the “clar.pvoc” data file
kinterp     linseg 1, p3*.15, 1, p3*.35, 0, p3*.25, 0, p3*.15, 1, p3*.1,
1
pvbufread ktime1, “oboe.pvoc”
apv  pvinterp  ktime2,1,”clar.pvoc”,1,1.065,1,.75,1-kinterp,1-kinterp
```

Below is an example using pvbufread with pvcross. In this example the amplitudes used in the resynthesis gradually change from those of the oboe to those of the clarinet. The frequencies, of course, remain those of the clarinet throughout the process since pvcross does not use the frequency data from the file read by pvbufread.

```
ktime1      line 0, p3, 3.5 ; used as index in the “oboe.pvoc” data file
ktime2      line 0, p3, 4.5 ; used as index in the “clar.pvoc” data file
kcross      expon .001, p3, 1
pvbufread  ktime1, “oboe.pvoc”
apv         pvcross ktime2, 1, “clar.pvoc”, 1-kcross, kcross
```

In following example using vpvoc shows the use of functions such as

```
        f 1 0 256 5 .001 128 1 128 .001
        f 2 0 256  5 1 128 .001 128 1
        f 3 0 256  7 1 256 1
```

to scale the amplitudes of the separate analysis bins.

```
ktime       line 0, p3,3 ; time pointer, in seconds, into data file
            ktablexseg 1, p3*.5, 2, p3*.5, 3
apv         vpvoc ktime,1, “pvoc.file”
```

The result would be a time-varying “spectral envelope” applied to the phase vocoder analysis data. Since this amplifies or attenuates the amount of signal at the frequencies that are paired with the amplitudes which are scaled by these functions, it has the effect of applying very accurate filters to the signal. In this example the first table would have the effect of a band- pass filter , gradually be band-rejected over half the note’s duration, and then go towards no modification of the magnitudes over the second half.

---

b) Tuning Opcodes (John Fitch)

```
 icps      cps2pch   ipch, iequal
 icps      cpsxpch   ipch, iequal, irepeat, ibase
```

Converts a pitch-class notation into cycles-per-second for equal divisions of the octave (for cps2pch) or for equal divisions of any interval. There is a restriction of no more than 100 equal divisions.

INITIALISATION

ipch - Input number of the form 8ve.pc, indicating an `octave’ and which note in the octave.

iequal - if positive, the number of equal intervals into which the `octave’ is divided. Must be less than or equal to 100, if negative is the number of a table of frequency multipliers

irepeat -Number indicating the interval which is the `octave’. The integer 2 corresponds to octave divisions, 3 a twelveth, 4 is two octaves, and so on. This need not be an integer, but must be positive.

ibase - The frequency which corresponds to pitch 0.0

Note: 1. The following are essentially the same

```
ia=         cpspch(8.02)
ib          cps2pch          8.02, 12
ic          cpsxpch          8.02, 12, 2, 1.02197503906
```

2. These are opcodes not functions.

3. Negative values of ipch are allowed, but not negative irepeat, iequal or ibase.

EXAMPLES

```
inote cps2pch p5, 19 ; convert oct.pch to cps in 19ET
inote cpsxpch p5, 12, 3, 261.62561;Pierce scale centered on middle
A
inote cpsxpch p5, 21, 4, 16.35160062496 ;10.5ET scale
```

The use of a table allows exotic scales by mapping frequencies in a table
For example the table

```
        f2 0 16 -2 1 1.1 1.2 1.3 1.4 1.6 1.7 1.8 1.9
```

can be used with

```
        ip      cps2pch      p4, -2
```

to get a 10 note scale of unequal divisions

---

c) Use of HRTF data

```
    aLeft, aRight  hrtfer   asig, kAz, kElev, “HRTFcompact”
```

Output is binaural (headphone) 3D audio.

INITIALIZATION
kAz - azimuth value in degrees. Positive values represent position on the right, negative values are positions on the left.
kElev - elevation value in degrees. Positive values represent position on the right, negative values are positions on the left.

At present, the only file which can be used with hrtferxk is HRTFcompact. It must be passed to the u.g. as the last argument within qoutes as shown above.

PERFORMANCE
These unit generators place a mono input signal in a virtual 3D space around the listner by convolving the input with the appropriate HRTF data specified by the opcode’s azimuth and elevation values. Hrtferxk allows these values to be k-values, allowing for dynamic spatialization. hrtferi can only place the input at the reqeusted position because the HRTF is loaded in at i-time (remember that currently, csound has a limit of 20 files it can hold in memory, otherwise it causes a segmentation fault). The output will need to be scaled either by using balance or by multiplying the output by some scaling constant.

Note - the sampling rate of the orchestra must be 44.1kHz. This is because 44.1kHz is the sampling rate at which the HRTFs were measured. In order to be used at a different rate, the HRTFs need to be resampled at the desired rate.

Example:

```
kaz  linseg 0, p3, -360  ; move the sound in circle
kel  linseg -40, p3, 45 ; around the listener, changing
                        ;elevation as its turning
asrc soundin “soundin.1”
aleft,aright hrtfer asrc, kaz, kel, “HRTFcompact”
aleftscale  = aleft * 200
arightscale = aright * 200

outs aleftscale, arightscale
```

---

d) Time Stretch -- Written by Richard Karpen, 1992.

ar **sndwarp** xamp, xtimewarp, xresample, ifn1, ibeg, iwsize, irandw, ioverlap, ifn2, [itimemode]

DESCRIPTION
sndwarp reads sound samples from a table (see under ifn1 for information about using this with the NeXT operating system) and applies time-stretching and/or pitch modification. Time and frequency modification are indepentant from one another. For example, a sound can be stretched while raising the pitch. The window size and overlap arguments are important to the result and should be experimented with. In general they should be as small as possible. For example,
start with iwsize=sr/20 and ioverlap=5. Try irandw=iwsize*.2. The algorithm reacts differently depending upon the input sound.

ARGUMENTS
ifn1 is the number of the table holding the sound samples which will be subjected to the sndwarp processing. GEN01 is the appropriate function generator to use to store the sound samples (a version of sndwarp for the NeXT operating system reads soundfiles directly instead of using function tables. In the NeXT version there is no practical limit to the length of the sound and stereo soundfiles can be processed. In this version, the maximum length of input sound is limited to the maximum table size allowable and/or to the amount of memory available to the program. Only the table-lookup version is available with this release). xamp is the amplitude by which to scale the signal (post time and frequency scaling). xtimewarp determines how the input signal will be stretched or shrunk in time. There are two ways to use this argument dependent upon the value given for itimemode. When the value of itimemode is 0 (or if no value is given; 0 is the default), xitimewarp will scale the time of the sound. For example, a value of 2 will stretch the sound by 2 times. When itimemode is any non-zero value then xtimewarp is used as a time pointer in a similar way in which the time pointer works in lpread and pvoc. An example below illustrates this. In both cases, the pitch will NOT be altered by this process. Pitch shifting is done independently using xresample. xresample is the factor by which to change the pitch of the sound. For example, a value of 2 will produce a sound one octave higher than the original. The "speed" of the sound, however, will NOT be altered. ibeg is the time in seconds to begin reading in the table (or soundfile). When itimemode is non-zero, the value of itemwarp is offset by ibeg. iwsize is the window size in samples used in the time warping algorithm. irandw is the bandwidth of a random number generator. The random numbers will be added to iwsize. ioverlap determines the density of overlapping windows. ifn2 is a function used to shape the window. It is usually something like a half a sine (ie: f1 0 16384 9 .5 1 0).

EXAMPLE
The below example shows a slowing down or stretching of the sound stored in the stored table (ifn1). Over the duration of the note, the stretching will grow from no change from the original to a sound which is ten times "slower" than the original. At the same time the overall pitch will move upward over the duration by an octave.

```
iwindfun=1
isampfun=2
ibeg=0
iwindsize=2000
iwindrand=400
ioverlap=10

awarp    line    1, p3, 10
aresamp  line    1, p3, 2
kenv     line    1, p3, .1
asig     sndwarp kenv, awarp, aresamp, isampfun, ibeg,
iwindsize, iwindrand, overlap, iwindfun
```

Here is an example using xtimewarp as a time pointer

```
itimemode=1
atime line 0, p3, 10
asig  sndwarp kenv, atime, aresamp, sampfun, ibeg, iwindsize,
      iwindrand, ioverlap, iwindfun, itimemode
```

In the above, atime advances the time pointer used in the sndwarp from 0 to 10 over the duration of the note. If p3 is 20 then the sound will be two times slower than the original. Of course you can use a more complex function than just a single straight line to control the time factor.

---

e) Non-linear filter

ar **nlfilt** ain, ka, kb, kd, kL, kC

Implements the filter Y{n} =a Y{n-1} + b Y{n-2} + d Y^2{n-L} + X{n} - C described in Dobson and Fitch (ICMC'96).

Examples:

i) Non-linear effect:

With a=b=0 and a delay (L) of 20 samples. The other parameter range

d = 0.8, 0.9, 0.7
C = 0.4, 0.5, 0.6

This affects the lower register most but there are audible effects over the whole range. We suggest that it may be useful for colouring drums, and for adding arbitrary highlights to notes

ii) Low Pass with non-linear:

a = 0.4
b = 0.2
d = 0.7
C = 0.11
L = 20, ... 200

There are instability problems with this variant but the effect is more pronounced of the lower register, but is otherwise much like the pure comb. Short values of L can add attack to a sound.

iii) High Pass with non-linear:

The range of parameters are
a = 0.35
b = -0.3
d = 0.95
C = 0,2, ... 0.4
L = 200

iv) High Pass with non-linear:

The range of parameters are
a = 0.7
b = -0.2, ... 0.5
d = 0.9
C = 0.12, ... 0.24
L = 500, 10

The high pass version is less likely to oscillate. It adds scintillation to medium-high registers. With a large delay L it is a little like a reverberation, while with small values there appear to be formant-like regions. There are arbitrary colour changes and resonances as the pitch changes. Works well with individual notes.

Warning: The "useful" ranges of parameters are not yet mapped.
---

g) Chanhes to Linear Predictive Coding

1) LPC Pole stabilization.

It is done through a new option in the analysis stage.

csound -U lpanal -a [other options]
The -a flag [alternate storage] asks lpanal to write a file with filter poles values rather than the usual filter coefficient files.

When lpread / lpreson are used with pole files, automatic stabilization is performed and the filter should not get wild anymore. I've implemented only one stabilization algorithm, more tune could be implemented later.

(This is the default in the Windows GUI)

2) LPC Interpolation.

Two new opcodes are available to perform interpolation between pole files of two analysis.

---

lpslot islot

islot: number of slot to be selected [0<islot<20]

lpslot selects the slot to be use by further lp opcodes. This is the way to load and reference several analysis at the same time.

---

lpinterpol islot1,islot2,kmix

islot1: slot where the first analysis was stored
islot2: slot where the second analysis was stored
kmix : mix value between the two analysis. Should be between 0 and 1. 0 means analysis 1 only. 1 means analysis 2 only. Any value inbetween will produce interpolation between the filters.

lpinterpol computes a new set of poles from the interpolation between two analysis. The poles will be stored in the current lpslot and used by the next lpreson opcode.

Here is a typical orc using the opcodes:

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

instr 1
; Define sound generator
ipower  init    50000
ifreq   init    440
asrc    buzz  ipower,ifreq,10,1                 ; Define time line
ktime     line    0,p3,p3                               ; Read square data
poles
lpslot  0
krmsr,krmso,kerr,kcps           lpread     ktime,"square.pol" ; Read
triangle data
                                                        ; poles
lpslot  1
krmsr,krmso,kerr,kcps           lpread          ktime,"triangle.pol"  ;
Compute result of
                                        ;  mixing  and  balance
power
kmix    line   0,p3,1
lpinterp 0,1,kmix

ares        lpreson asrc
aout        balance ares,asrc
      out   aout
       endin
```

Release Notes for v3.45
==========

Version 3.45 is a small collection of corrections and additions following the large changes of 3.44

Summary
=======

a) An additional optional argument has been added to all the Butterworth filters, vdelay, and reverb2, which ifnon-zero
skips the initialisation stage.

b) Return code corrected after no graphics

c) Allow arguments like -m6W

d) Added new opcodes uniform to complete the random generators

e) Added fof2 opcode

f) New command option -z to give a list of opcodes. -z0 or -z prints a list; -z1 prints a list with answer/argument descriptions.

g) Adjust writing of scores to allow longer tables (as described by Richard Karpen).

h) Adjusted sndwarp to give (optional) stereo, and new opcode sndwarpst (Described below)

DOCUMENTATION ON NEW/REVISED OPCODES
==========

_SNDWARP (Written by Richard Karpen, 1992. Most recent revision, 1997)

asig [, acmp] **sndwarp** xamp, xtimewarp, xresample, ifn1, ibeg, iwsize, irandw, ioverlap, ifn2, itimemode

asig1, asig2 [, acmp1, acmp2] **sndwarpst** xamp, xtimewarp, xresample, ifn1, ibeg, iwsize, irandw, ioverlap, ifn2, timemode

DESCRIPTION
sndwarp reads sound samples from a table and applies time-stretching and/or pitch modification. Time and frequency modification are indepentant from one another. For example, a sound can be stretched in time while raising the pitch! The window size and overlap arguments are important to the result and should be experimented with. In general they should be as small as possible. For example, start with iwsize=sr/10 and ioverlap=15. Try irandw=iwsize*.2. If you can get away with less overlaps, the program will be faster. But too few may cause an audible flutter in the amplitude. The algorithm reacts differently depending upon the input sound and there are no fixed rules for the best use in all circumstances. But with proper tuning, excellent results can be achieved.
OUTPUTS
asig is the single channel of output from the sndwarp unit generator while asig1 and asig2 are the stereo (left and right) outputs from sndwarpst. sndwarp assumes that the function table holding the sampled signal is a mono one while sndwarpst assumes that it is stereo. This simply means that sndwarp will index the table by single-sample frame increments and sndwarpst will index the table by a two-sample frame increment. The user must be aware then that if a mono signal is used with sndwarst or a stereo one with sndwarp, time and pitch will be altered accordingly.

acmp in sndwarp and acmp1, acmp2 in sndwarpst, are single layer (no overlaps), unwindowed versions of the time and/or pitch altered signal. They are supplied in order to be able to balance the amplitude of the signal output, which typically contains many overlapping and windowed versions of the signal, with a clean version of the time-scaled and pitch-shifted signal. The sndwarp process can cause noticeable changes in amplitude, (up and down), due to a time differential between the overlaps when time-shifting is being done.

When used with a balance unit, acmp, acmp1, acmp2 can greatly enhance the quality of sound. They are optional, but note that in sndwarpst they must both be present in the syntax (use both or neither). An example of how to use this is given below.

INPUT ARGUMENTS
ifn1 is the number of the table holding the sound samples which will be subjected to the sndwarp processing. GEN01 is the appropriate function generator to use to store the sound samples from a pre-existing soundfile.

xamp is the value by which to scale the amplitude (see note on the use of this when using acmp, acmp1, amcp2).

xtimewarp determines how the input signal will be stretched or shrunk in time. There are two ways to use this argument depending upon the value given for itimemode. When the value of itimemode is 0, xitimewarp will scale the time of the sound. For example, a value of 2 will stretch the sound by 2 times. When itimemode is any non-zero value then xtimewarp is used as a time pointer in a similar way in which the time pointer works in lpread and pvoc. An example below illustrates this. In both cases, the pitch will NOT be altered by this process. Pitch shifting is done independently using xresample.

xresample is the factor by which to change the pitch of the sound. For example, a value of 2 will produce a sound one octave higher than the original. The timing of the sound, however, will NOT be altered.

ibeg is the time in seconds to begin reading in the table (or soundfile). When itimemode is non- zero, the value of itimewarp is offset by ibeg.

iwsize is the window size in samples used in the time scaling algorithm.

irandw is the bandwidth of a random number generator. The random numbers will be added to iwsize.

ioverlap determines the density of overlapping windows.

ifn2 is a function used to shape the window. It is usually used to create a ramp of some kind from zero at the beginning and back down to zero at the end of each window. Try using a half a sine (ie: f1 0 16384 9 .5 1 0) which works quite well. Other shapes can also be used.

EXAMPLES

The below example shows a slowing down or stretching of the sound stored in the stored table (ifn1). Over the duration of the note, the stretching will grow from no change from the original to a sound which is ten times “slower” than the original. At the same time the overall pitch will move upward over the duration by an octave.

```
iwindfun=1
isampfun=2_ibeg=0_iwindsize=2000_iwindrand=400_ioverlap=10

awarp     line     1, p3, 1
aresamp   line     1, p3, 2
kenv      line     1, p3, .1

asig                                                    sndwarp
kenv,awarp,aresamp,isampfun,ibeg,iwindsize,iwindrand,
ioverlap,iwindfun,0
```

Now, here’s an example using xtimewarp as a time pointer and using stereo

```
itimemode=1
atime line 0, p3, 10
asig1, asig2   sndwarpst kenv, atime, aresamp, sampfun, ibeg,
iwindsize, iwindrand, ioverlap, iwindfun, itimemode
```

In the above, atime advances the time pointer used in the sndwarp from 0 to 10 over the duration of the note. If p3 is 20 then the sound will be two times slower than the original. Of course you can use a more complex function than just a single straight line to control the time factor.

Now the same as above but using the balance function with the optional outputs:

```
asig,acmp  sndwarp
1,awarp,aresamp,isampfun,ibeg,iwindsize,iwindrand,
ioverlap,iwindfun,itimemode

abal balance asig, acmp

asig1,asig2,acmp1,acmp2   sndwarpst 1,  atime,  aresamp,
sampfun, ibeg, iwindsize, iwindrand, ioverlap, iwindfun,  itimemode

abal1 balance asig1, acmp1
abal2 balance asig2, acmp2
```

In the above two examples notice the use of the balance unit. The output of balance can then be scaled, enveloped, sent to an out or outs, and so on. Notice that the amplitude arguments to sndwarp and sndwarpst are “1” in these examples. By scaling the signal after the sndwarp process, abal, abal1, and abal2 should contain signals that have nearly the same amplitude as the original input signal to the sndwarp process. This makes it much easier to predict the levels and avoid samples out of range or sample values that are too small.

More advice: Only use the stereo version when you really need to be processing a stereo file. It is someone slower than the mono version and if you use the balance function it is slower again. There is nothing wrong with using a mono sndwarp in a stereo orchestra and sending the result to one or both channels of the stereo output!

FOF2

DESCRIPTION:

Get rid of the last argument to fof, “ifmode”. Instead we’ll have “kgliss”, internal grain glissandi. (This is certainly not as generally useful as kphs, but it does brighten up any fof instrument, allowing more spectral variation.)

Usage:
kgliss - sets the end pitch of each grain relative to the initial pitch, in octaves. Thus kgliss = 2 means that the grain ends two octaves above its initial pitch, while kgliss = -5/3 has the grain ending a perfect major sixth below.

Rationale:
The toggle switch “ifmode” always struck me as of rather limited use. When zero (default), the initial pitch of a grain (given by kform at the inception of a new grain) is kept steady throughout its lifetime, if one, every grain’s pitch follows kform. This may certainly be good for vocal synthesis, but for granular synthesis i’d like more control over the internal grain pitch. Eg grain glissandi.

There have been requests for a way of telling what opcodes are available in a given version. To assist with that there is a new command line option

-z	Give a list of opcodes and exit
-z0	The same
-z1	Give a list of opcodes, together with answer and argument types; then exit

---

Butterworth filters (Revised)

ar	**butterhp**	asig, kfreq[, iskip]
ar	**butterlp**	asig, kfreq[, iskip]
ar	**butterbp**	asig, kfreq, kband[, iskip]

ar **butterbr** asig, kfreq, kband[, iskip]

Implementations of second-order hipass, lopass, bandpass and bandreject Butterworth filters.

PERFORMANCE

These new filters are butterworth second-order IIR filters. They are slightly slower than the original filters in Csound, but they offer an almost flat passband and very good precision and stopband attenuation.

asig - Input signal to be filtered.

kfreq - Cuttoff or center frequency for each of the filters.

kband - Bandwidth of the bandpass and bandreject filters.

iskip - Skip initialisation if present and non zero

EXAMPLE

```
asigrand        10000           ; White noise signal
alpf   butterlp asig, 1000      ; cutting frequencies above1K
ahpf   butterhp asig, 500       ; passing frequencies above 500Hz
abpf   butterbp asig, 2000, 100 ; passing only 1950 to 2050 Hz
abrf   butterbr asig, 4500, 200 ; cutting only 4400 to 4600 Hz
```

Vdelay (revised)

ar **vdelay** asig, adel, imaxdel[, iskip]

This is an interpolating variable time delay, it is not very different from the existing implementation (deltapi), it is only easier to use.

INITIALIZATION

imaxdel - Maximum value of delay in samples. If adel gains a value greater than imaxdel it is folded around imaxdel. This should not happen.

iskip - Skip initialisation if present and non zero

PERFORMANCE

With this unit generator it is possible to do Doppler effects or chorusing and flanging.

asig - Input signal.

adel - Current value of delay in samples. Note that linear functions have no pitch change effects. Fast changing values of adel will cause discontinuities in the waveform resulting noise.

Example

```
f1 0 8192 10 1
ims = 100                       ; Maximum delay time in msec
a1   oscil  10000, 1737, 1      ; Make a signal
a2   oscil  ims/2, 1/p3, 1      ; Make an LFO
a2 = a2 + ims/2                 ; Offset the LFO so that it is
positive
a3   vdelay         a1, a2, ims ; Use the LFO to control delay time
     out    a3
```

Two important points here. First, the delay time must be always positive. And second, even though the delay time can be controlled in k-rate, it is not advised to do so, since sudden time changes will create clicks.

Reverb2 (Revised)

ar **reverb2** asig, ktime, khdif[, iskip]

This is a reverberator consisting of 6 parallel comb-lowpass filters being fed into a series of 5 allpass filters.

iskip - Skip initialisation if present and non zero

PERFORMANCE

The input signal asig is reverberated for ktime seconds. The parameter khdif controls the high frequency diffusion amount. The values of khdif should be from 0 to 1. If khdif is set to 0 the all the frequencies decay with the same speed. If khdif is 1, high frequencies decay faster that lower ones.

Example:

```
a1      oscil 10000, 100, 1
a2      reverb2   a1, 2.5, .3
out     a1 + a2 * .2
```

This results in a 2.5 sec reverb with faster high frequency attenuation.

Noise Generator

xuniform krange - Uniform distribution random number generator. Krange is the range of the random numbers [0 - krange).

Release Notes for v3.46

Version 3.46 is a large collection of corrections, and some additions including the Whittle table opcodes, FOG and a more flexible version of soundin.

Summary

a) Changes in a number of MIDI opcodes (ipchmidib, ioctmidib, icpsmidib, kpchmidib, koctmidib, kcpsmidib, imidictrl, kmidictrl) to add additional optional argument, mainly for scaled pitchbend; new opcode midisetb (Mike Berry)

b) Large number of bug fixes; mainly minor and to do with igoto (many people). Corrected writing to log file on PC (mega dyslexia!)

c) Negative p3 in score nolonger confuses other references to that note in the score (JPff/RWD)

d) Attempt to fix LINUX precision problem in large table sizes etc

e) Correction of my fiasco in opcode listing

f) Correction of HRTF code on PCs and other machines with that byte
order.

g) New opcode, diskin. Like soundin except allows variable rate of reading sound file. (Mike Berry)

h) Fixes to -o dac on Windows95

i) Improved pvoc with optional last argument (Richard Karpen)

j) Defaults for sr/kr/ksmps such that one can omit one. Defaults if all omitted made CD level.

k) Robin Whittle's table reading opcodes included

l) Missing endin now noticed

m) New opcode fog from Michael Clarke (still needs a little weaking)

n) Fixed cscore main program for dribble files

o) Number of tables is no longer fixed, but expands as required.

DOCUMENTATION ON NEW/REVISED OPCODES
==========

MIDI CONVERTERS

ival **notnum**
ival **veloc**
icps **cpsmidi**
icps **cpsmidib**
kcps **cpsmidib** [irange]
ioct **octmidi**
ioct **octmidib**
koct **octmidib** [irange]
ipch **pchmidi**
ipch **pchmidib**
kpch **pchmidib** [irange]
iamp **ampmidi** iscal[, ifn]
kaft **aftouch** iscal
kchpr **chpress** iscal
kbend **pchbend** iscal
ival **midictrl** inum[, initial]
kval **midictrl** inum[, initial]
kval **midictrlsc** inum[,iscal] [, ioffset] [, initial]

Get a value from the MIDI event that activated this instrument, or from a continuous MIDI controller, and convert it to a locally useful format.

INITIALIZATION

iscal - I-time scaling factor.

ifn (optional) - function table number of a normalized translation table, by which the incoming value is first interpreted. The default value is 0, denoting no translation.

inum - MIDI controller number.

initial - the initial value of the controller.

irange - the pitch bend range in semitones.

PERFORMANCE

notnum, veloc - get the MIDI byte value (0 - 127) denoting the note number or velocity of the current event.

cpsmidi, octmidi, pchmidi - get the note number of the current MIDI event, expressed in cps, oct, or pch units for local processing.

cpsmidib, octmidib, pchmidib - get the note number of the current MIDI event, modify it by the current pitch-bend value, and express the
result in cps, oct, or pch units. Available as an I-time value or as a continuous ksig value.

ampmidi - get the velocity of the current MIDI event, optionally pass it through a normalized translation table, and return an amplitude value in the range 0 - iscal.

aftouch, chpress, pchbend - get the current after-touch, channel pressure, or pitch-bend value for this channel, rescaled to the range 0 - iscal. Note that this access to pitch-bend data is independent of the MIDI pitch, enabling the value here to be used for any arbitrary purpose.

midictrl - get the current value (0 - 127) of a specified MIDI controller.

midictrlsc - get a scaled and offset value of a controller.

SIGNAL INPUT & OUTPUT

a1 **in**
a1, a2 **ins**
a1, a2, a3, a4 **inq**
a1 **soundin** ifilcod[,iskptim][, iformat]
a1, a2 **soundin** ifilcod[,iskptim][, iformat]
a1, a2, a3, a4 **soundin** ifilcod[,iskptim][, iformat]
a1[,a2[,a3,a4] **diskin** ifilcod, kpitch[,iskiptim][, iwraparound] [,iformat]
out asig
outs1 asig
outs2 asig
outs asig1, asig2

outq1 asig
outq2 asig
outq3 asig
outq4 asig
outq asig1, asig2, asig3, asig4

These units read/write audio data from/to an external device or stream.

INITIALIZATION

filcod - integer or character-string denoting the source soundfile name. An integer denotes the file soundin.filcod ; a character-string (in double quotes, spaces permitted) gives the filename itself, optionally a full pathname. If not a full path, the named file is sought first in the current directory, then in that given by the environment variable SSDIR (if defined) then by SFDIR. See also GEN01.

iskptim (optional) - time in seconds of input sound to be skipped.
The default value is 0.

iformat (optional) - specifies the audio data file format: 1 = 8-bit signed char (high-order 8 bits of a 16-bit integer), 2 = 8-bit A-law bytes, 3 = 8-bit U-law bytes, 4 = 16-bit short integers, 5 = 32-bit
long integers, 6 = 32-bit floats). If iformat = 0 it is taken from the soundfile header, and if no header from the csound -o command flag. The default value is 0.

kpitch - can be any real number. a negative number signifies backwards playback. The given number is a pitch ratio, where: 1= norm pitch, 2=oct higher, 3=12th higher,etc; .5= oct lower, .25=2oct lowr, etc; -1= norm pitch backwards,-2=oct higher backwrds,etc..

iwraparound - 1=on, 0=off (wraps around to end of file either direction)

PERFORMANCE

in, ins, inq - copy the current values from the standard audio input buffer. If the command-line flag -i is set, sound is read continuously from the audio input stream (e.g. stdin or a soundfile) into an internal buffer. Any number of these units can read freely from this buffer.

soundin is functionally an audio generator that derives its signal from a pre-existing file. The number of channels read in is controlled by the number of result cells, a1, a2, etc., which must match that of the input file. A soundin unit opens this file whenever the host instrument is initialized, then closes it again each time the instrument is turned off. There can be any number of soundin units within a single instrument or orchestra; also, two or more of them can read simultaneously from the same external file.

diskin is identical to soundin, except that it can alter the pitch of the sound that is being read.

out, outs, outq send audio samples to an accumulating output buffer (created at the beginning of performance) which serves to collect the output of all active instruments before the sound is written to disk.
There can be any number of these output units in an instrument. The type (mono, stereo, or quad) must agree with nchnls, but units can be

chosen to direct sound to any particular channel: outs1 sends to stereo channel 1, outq3 to quad channel 3, etc.

---

ar **fog** xamp, xdens, xtrans, xspd, koct, kband, kris, kdur, kdec, iolaps, ifna, ifnb, itotdur[, iphs][, itmode]

Audio output is a succession of grains derived from data in a stored function table ifna. The local envelope of these grains and their timing is based on the model of fof synthesis and permits detailed control of the granular synthesis.

INITIALIZATION

iolaps - number of pre-located spaces needed to hold overlapping rain
data. Overlaps are density dependent, and the space required depends on the maximum value of xdens* kdur. Can be over-estimated at no computation cost. Uses less than 50 bytes of memory per iolaps.

ifna, ifnb - table numbers of two stored functions. The first is the data used for granulation, usually from a soundfile (GEN01). The second is a rise shape, used forwards and backwards to shape the grain rise and decay; this is normally a sigmoid (GEN19) but may be linear (GEN07).

itotdur - total time during which this fog will be active. Normally set to p3. No new grain is created if it cannot complete its kdur within the remaining itotdur.

iphs (optional) - initial phase of the fundamental, expressed as a fraction of a cycle (0 to 1). The default value is 0.

itmode (optional) - transposition mode. If zero, each grain keeps the xtrans value it was launched with. if non-zero, each is influenced by xtrans continuously. The default value is 0.

PERFORMANCE

xamp - amplitude factor. Amplitude is also dependent on the number of overlapping grains, the interaction of the rise shape (ifnb) and the exponential decay (kband), and the scaling of the grain waveform (ifna). The actual amplitude may therefore exceed xamp.

xdens - density. The frequency of grains per second.

xtrans - transposition factor. The rate at which data from the stored function table ifna is read within each grain. This has the effect of transposing the original material. A value of 1 produces the original pitch. Higher values transpose upwards, lower values downwards. Negative values result in the function table being read backwards.

xspd - speed. The rate at which successive grains advance through the stored function table ifna. xspd is in the form of an index (0 to 1) to ifna. This determines the movement of a pointer used as the starting point for reading data within each grain. (xtrans determines the rate at which data is read starting from this pointer.)

koct - octaviation index. The operation of this parameter is identical to that in fof.

kband, kris, kdur, kdec - grain envelope shape. These parameters determine the exponential decay (kband), and the rise (kris), overall duration (kdur,) and decay (kdec) times of the grain envelope. Their operation is identical to that of the local envelope parameters in fof.

The Csound fog generator is by Michael Clarke, extending his earlier work based on IRCAM's fof algorithm.

Example:

```
;p4 = transposition factor
;p5 = speed factor
;p6 = function table for grain data
i1      =       sr/ftlen(p6) ;scaling to reflect sample rate and table length
a1      phasor  i1*p5        ;index for speed
a2      fog     5000, 100, p4, a1, 0, 0, ,.01, .02, .01, 2, p6, 1, p3, 0, 1
```

>>> 1 - TABLE WRITE
==========

This works on existing function tables, changing their contents. There could be all sorts of uses for this. Assuming that users (.orc and .sco programmers) know what they are doing, then there should be no more trouble than the use of global variables.

As when using global variables, the user must consider how the code is run.

In each k cycle, instruments are executed, in order of instrument number, and within instruments, in order of the instances of the instrument. I presume the instance order depends on their starting time.

As execution proceeds, each ugen is run once at k time. For k type ugens, they do their job once. For a rate ugens, they process one or more arrays of a rate variables. For instance a table read at a rate, with ksmps = 7, uses a 7 long array of indexes to read into a table, retrieving 7 different values and writing them to a 7 long array for the output.

So

ablah table azot, 5

will read from table 5, a set of values pointed to by an array of indexes pointed to by azot, and write them to an array pointed to by ablah.

We may conceive of an idea of writing successive a rate values to a single table location, and subsequently reading them from that location. This would not work with ksmps = 7 - only the last written value would remain by the time execution passed to the next ugen.

So table write is a means of patching i or k rate signals to particular locations in function tables, where they can be read by table read ugens. However this does not work for a rate signals, unless you conspire to use a range of the table, and organise your indexes very carefully.

Patching of i, k and a rate signals under .orc program control is best achieved with arrays - which do not yet exist. See the zak system for a next best solution with ugens.

So the main purpose of a table write ugen is to refashion function tables on the fly under program control. tablemix, tablecopy, tablera and tablewa can also be used for such purposes.

Applications are diverse. One is to generate a waveshaping table with .orc code. A loop could be created and an instrument could spend some time with k rate operations looping to address each table location - rewriting the table, before letting performance proceed. This would probably be too slow to work with real-time music production.

Another application is to continuously sculpt tables while they are being used. Each k or a cycle, one or a few locations are changed a little.

With these applications in mind, lets look at the tablew and itablew code.

Firstly , itabelw is just the same as k rate tablew, except it only happens once at the initialisation of the ugen. (I must investigate what happens if an i rate ugen is executed first, via an if goto, some time after the instrument is initialised.)

The Csound orchestra loader decides whether this is k or a rate operation, and fires up the appropriate subroutine in the unit generator code.

There is no output variable.

itablew, tablew and tablewkt
---

itablew isig, indx, ifn [,ixmode] [,ixoff] [,iwgmode]

Use itablew when all inputs are init time variables or constants and you only want to run it at the initialisation of the instrument.

tablew is for writing at k or at a rates, with the table number being specified at init time.

tablewkt is the same, but uses a k rate variable for selecting the table number. The valid combinations of variable types are shown by the first letter of the variable names:

itablew	isig, indx, ifn [,ixmode] [,ixoff] [,iwgmode]
tablew	ksig, kndx, ifn [,ixmode] [,ixoff] [,iwgmode]
tablew	asig, andx, ifn [,ixmode] [,ixoff] [,iwgmode]
tablewkt	ksig, kndx, kfn [,ixmode] [,ixoff] [,iwgmode]
tablewkt	asig, andx, kfn [,ixmode] [,ixoff] [,iwgmode]

isig, ksig, asig — The value to be written into the table.

indx, kndx, andx — Index into table, either a positive number range matching the table length (ixmode = 0) or a 0 to 1 range (ixmode != 0)

ifn, kfn — Table number. Must be >= 1. Floats are rounded down to an integer. If a table number does not point to a valid table, or the table has not yet been loaded (gen01) then an error will result and the instrument will be de-activated.

ixmode — Default 0 ==0 xndx and ixoff ranges match the length of the table.

!=0 xndx and ixoff have a 0 to 1 range.

ixoff — Default 0 ==0 Total index is controlled directly by xndx. ie. the indexing starts from the start of the table.

!=0 Start indexing from somewhere else in the table. Value must be positive and less than the table length (ixmode = 0) or less than 1 (ixmode !=0

iwgmode — Default 0 ==0 Limit mode } See below
==1 Wrap mode }
==2 Guardpoint mode }

0 = Limit mode

Limit the total index (ndx + ixoff) to between 0 and the guard point.

For a table of length 5, this means that locations 0 to 3 and location 4 (the guard point) can be written. A negative total index writes to location 0. Total indexes > 4 write to location 4.

1 = Wrap mode

Wrap total index value into locations 0 to E, where E is one less than either the table length or the factor of 2 number which is one less than the table length. For example, wrap into a 0 to 3 range - so that total index 6 writes to location 2.

2 = Guardpoint mode

The guardpoint is written at the same time as location 0 is written - with the same value.

This facilitates writing to tables which are intended to be read with interpolation for producing smooth cyclic waveforms. In addition, before it is used, the total index is incremented by half the range between one location and the next, before being rounded down to the integer address of a table location.

Normally (igwmode = 0 or 1) for a table of length 5 - which has locations 0 to 3 as the main table and location 4 as the guard point, a total index in the range of 0 to 0.999 will write to location 0. ("0.999" means just less than 1.0.) 1.0 to 1.999 will write to location 1 etc.

A similar pattern holds for all total indexes 0 to 4.999 (igwmode = 0) or to 3.999 (igwmode = 1). igwmode = 0 enables locations 0 to 4 to be written - with the guardpoint (4) being written with a potentially different value from location 0.

With a table of length 5 and the iwgmode = 2, then when the total index is in the range 0 to 0.499, it will write to locations 0 and 4.
Range 0.5 to 1.499 will write to location 1 etc. 3.5 to 4.0 will _also_ write to locations 0 and 4.

This way, the writing operation most closely approximates the results of interpolated reading. Guard point mode should only be used with tables that have a guardpoint.

Guardpoint mode is accomplished by adding 0.5 to the total index, rounding to the next lowest integer, wrapping it modulo the factor of two which is one less than the table length, writing the the table (locations 0 to 3 in our example) and then writing to the guard point if index == 0.

tablew has no output value. The last three parameters are optional and
have default values of 0.

Caution with k rate table numbers

The following notes also apply to the tablekt and tableikt ugens which
can now have their table number changed at k rate.

At k rate or a rate, if a table number of < 1 is given, or the table number points to a non-existent table, or to one which has a length of 0 (it is to be loaded from a file later) then an error will result and the instrument will be deactivated.

>>> 2 - tablegpw, tableleng, tablemix and tablecopy
==========

tableleng
---

ir **itableng** ifn
kr **tableng** kfn

ifn i rate number of function table
kfn k rate number of function table

These return the length of the specified table. This will be a power of two number in most circumstances - it will not show whether a table has a guardpoint or not - it seems this information is not available in the table's data structure. If table is not found, then 0 will be returned.

Likely to be useful for setting up code for table manipulation operations, such as tablemix and tablecopy.

tablgpw
---

itablegpw ifn
tablegpw kfn

For writing the table's guard point, with the value which is in location 0. Does nothing if table does not exist.

Likely to be useful after manipulating a table with tablemix or tablecopy.

tablemix
---

tablemix kdft, kdoff, klen, ks1ft, ks1off, ks1g, ks2ft, ks2off, ks2g
itablemix idft, idoff, ilen, is1ft, is1off, is1g, is2ft, is2off, is2g

This ugen mixes from two tables, with separate gains into the destination table. Writing is done for klen locations, usually stepping forward through the table - if klen is positive.
If it is negative, then the writing and reading order is backwards - towards lower indexes in the tables. This bidirectional option makes it easy to shift the contents of a table sideways by reading from it and writing back to it with a different offset.

If klen is 0, no writing occurs. Note that the internal integer value of klen is derived from the ANSI C floor() function - which returns the next most negative integer. Hence a fractional negative klen value of -2.3 would create an internal length of 3, and cause the copying to start from the offset locations and proceed for two locations to the left.

The total index for table reading and writing is calculated from the starting offset for each table, plus the index value, which starts at 0 and then increments (or decrements) by 1 as mixing proceeds.

These total indexes can potentially be very large, since there is no restriction on the offset or the klen. However each total index for each table is ANDed with a length mask (such as 0000 0111 for a table of length 8) to form a final index which is actually used for reading or writing. So no reading or writing can occur outside the tables.

This is the same as "wrap" mode in table read and write. These ugens do not read or write the guardpoint.

If a table has been rewritten with one of these, then if it has a guardpoint which is supposed to contain the same value as the location 0, then call tablegpw afterwards.

The indexes and offsets are all in table steps - they are not normalised to 0 - 1. So for a table of length 256, klen should be set to 256 if all the table was to be read or written.

The tables do not need to be the same length - wrapping occurs individually for each table.

kdft Destination function table.

kdoff Offset to start writing from. Can be negative.

klen Number of write operations to perform. Negative means work backwards.

ks1ft ks2ft Source function tables. These can be the same as the destination table, if care is exercised about direction of copying data.

ks1off ks2off Offsets to start reading from in source tables.

ks1g ks2g Gains to apply when reading from the source tables. The results are added and the sum is written to the destination table.

tablecopy
---

tablecopy kdft, ksft
itablecopy idft, isft

Simple, fast table copy ugens. Takes the table length from the destination table, and reads from the start of the source table. For speed reasons, does not check the source length - just copiesregardless - in "wrap" mode. This may read through the source table several times. A source table with length 1 will causeall values in the destination table to be written to its value.

Table copy cannot read or write the guardpoint. To read it use table read, with ndx = the table length. Likewise use table write to write it.

To write the guardpoint to the value in location 0, use tablegpw.

This is primarily to change function tables quickly in a real-time situation.

kdft Number of destination function table.

ksft Number of source function table.

>>> 3 - tablera and tablewa
==========

These ugens read and write tables in sequential locations to and from an a rate variable. Some thought is required before using them. They
have at least two major, and quite different, applications which are discussed below.

ar **tablera** kfn, kstart, koff

kstart **tablewa** kfn, asig, koff

ar a rate distination for reading ksmps values from a table.

kfn i or k rate number of the table to read or write.

kstart Where in table to read or write.

asig a rate signal to read from when writing to the table.

koff i or k rate offset into table. Range unlimited - see explanation at end of this section.

In one application, these are intended to be used in pairs, or with several tablera ugens before a tablewa - all sharing the same kstart variable.

These read from and write to sequential locations in a table at audiorates, with ksmps floats being written and read each cycle.

tablera starts reading from location kstart.
tablewa starts writing to location kstart, and then writes to kstart with the number of the location one more than the one it last wrote.
(Note that for tablewa, kstart is both an input and output variable.)
If the writing index reaches the end of the table, then no further writing occurs and zero is written to kstart.

For instance, if the table's length was 16 (locations 0 to 15), and ksmps was 5. Then the following steps would occur with repetitive runs of the tablewa ugen, assuming that kstart started at 0.

Run no.	Initial kstart	Final kstart	locations written
1	0	5	0 1 2 3 4

```
2          5     10    5   6   7   8   9

3          10    15    10  11  12  13  14

4          15    0     15
```

This is to facilitate processing table data using standard a rate orchestra code between the tablera and tablewa ugens:

```
;---------------------------------
                    ;
  kstart = 0        ;
                    ; Read 5 values from table into an
                    ; a rate variable.

lab1:   atemp   tablera ktabsource, kstart, 0  ; Process the values using a
                                                ; rate code.
atemp = log(atemp)  ;
                    ; Write itback to the table

kstart  tablewa ktabdest, atemp, 0    ; Loop until all table locations have
                                      ; been processed.
if ktemp > 0 goto lab1                ;
                                      ;
;---------------------------------
```

The above example shows a processing loop, which runs every k cycle, reading each location in the table ktabsource, and writing the log of those values into the same locations of table ktabdest.

This enables whole tables, parts of tables (with offsets and different control loops) and data from several tables at once to be manipulated with a rate code and written back to another (or to the same) table.
This is a bit of a fudge, but it is faster than doing it with k rate table read and write code.

Another application is:

```
;---------------------------------
                          ;
kzero = 0                 ;
kloop = 0                 ;
                          ;
kzero tablewa 23, asignal, 0    ; ksmps a rate samples written into
                                ; locations 0 to (ksmps-1) of table 23.
                                ;
lab1: ktemp table kloop, 23     ; Start a loop which runs ksmps times,
                                ; in which each cycle processes one of
 [ Some code to manipulate ]    ; table 23's values with k rate orchestra
 [ the value of ktemp.     ]    ; code.
                    ;
                    ;
      tablew ktemp, kloop, 23 ; Write the processed value to the table.
                    ;
kloop = kloop + 1             ; Increment the kloop, which is both the
                              ; pointer into the table and the loop
if kloop < ksmps goto lab1    ; counter. Keep looping until all values
                              ; in the table have beenprocessed.
                              ;
asignal tablera 23, 0, 0      ; Copy the table contents back to an a rate
                              ; variable.
;---------------------------------
```

koff This is an offset which is added to the sum of kstart and the internal index variable which steps through the table. The result is then ANDed with the lengthmask (000 0111 for a table of length 8 - or 9 with guardpoint) and that final index is used to read or write to the table. koff can be any value. It is converted into a long using the ANSI floor() function so that -4.3 becomes -5. This is what we would want when using offsets which range above and below zero.

Ideally this would be an optional variable, defaulting to 0, however with the existing Csount orchestra read code, such default parameters must be init time only. We want k rate here, so we cannot have a default.

Notes on tablera and tablewa

These are a fudge, but they allow all Csounds k rate operators to be used (with caution) on a rate variables - something that would only be possible otherwise by ksmps = 1, downsamp and upsamp.

Several cautions:

1 - The k rate code in the processing loop is really running at a rate, so time dependant functions like port and oscil work faster than normal - their code is expecting to be running at k rate.

2 - This system will produce undesirable results unless the ksmps fits within the table length. For instance a table of length 16 will accomodate 1 to 16 samples, so this example will work with ksmps = 1 to 16.

Both these ugens generate an error and deactivate the instrument if a table with length < ksmps is selected. Likewise an error occurs if kstart is below 0 or greater than the highest entry in the table - if kstart >= table length.

3 - kstart is intended to contain integer values between 0 and (table length - 1). Fractional values above this should not affect operation but do not achieve anything useful.

4 - These ugens do not do interpolation and the kstart and koff parameters always have a range of 0 to (table length - 1) - not 0 to 1 as is available in other table read/write ugens. koff can be outside this range but it is wrapped around by the final AND operation.

5 - These ugens are permanently in wrap mode. When koff is 0, no wrapping needs to occur, since the kstart++ index will always be within the table's normal range. koff != 0 can lead to wrapping.

6 - The offset does not affect the number of read/write cycles performed, or the value written to kstart by tablewa.

7 - These ugens cannot read or write the guardpoint. Use tablegpw to write the guardpoint after manipulations have been done with tablewa.

>>> 4 - The "zak" system for patching signals
==========

"zak" means a or k rate patching, (i rate too), with a z at the start of the names of the ugens.

This is a fudge to do the work until arrays are implemented. I want to use such facilities and will use zak for the time being.

The zak system uses one area of memory as a global i or k rate patching area, and another for audio rate patching.

These are establised by a ugen which must be called once only:

zakinit isizea, isizek

isizea The number of audio rate "locations" for a rate patching. Each "location" is actually an array which is ksmps long.

isizek The number of locations we want to reserve for floats in the zk space. These can be written and read at i and k rates.

eg. zakinit 10 30 reserves memory for locations 0 to 30 of zk space and for locations 0 to 10 of a rate za space. With ksmps = 8, this would take 31 floats for zk and 80 floats for za space.

At least one location is always allocated for both za and zk spaces.
There is nothing wrong with having za and zk ranges thousands or tens
of thousands, but most pieces probably only need a few dozen to patch
their signals around.

These patching locations can be referred to by number with the following ugens.

The easiest way to run zakinit just once is to put it outside any instrument definition. Typically this would be at the start of the orchestra file, with the sr etc. definitions. All code outside the instrument definitions is treated as instrument one and is given an init run at time = 0.

zir, zkr, zkw

There are two short, simple, fast opcodes which read a location in zk space, at either i time or at the k rate.

ir **zir** indx
kr **zkr** kndx

Likewise, two write to a location in zk space at i time or at the k rate.

ziw isig, indx
zkw ksig, kndx

These are fast and always check that the index is within the range of zk space. If it is out of range, an error is reported and 0 is returned, or no writing takes place.

isig i rate } Value to write to the zk
ksig i or k rate } location.

indx i rate } Which zk location to write it to.
kndx i or k rate }

For instance,

zkw kzoom, p8

can be used so that parameter 8 of the instrument's command line could control where in zk space the output is written.

zkw kzoom, 7

This will always write it to zk location 7.

kxxx phasor 1

kdest = 40 + kxxx * 16
zkw kzoom, kdest

This will write kzoom to locations 40 to 55 on a one second scan cycle.

zar, zaw

For a rate reading and writing, we use similar opcodes:

ar **zar** kndx

Reads number kndx array of floats which are the ksmps number of audio rate floats to be processed in a k cycle.

zaw asig, kndx

Writes into the array specified by kndx.

In both cases, the ugen figures out where the array is and auto indexes through it to get each of the ksmps number of samples.

The za space is separate from the zk space.

These are the basic zk and za read and write ugens. However there are
a number of luxuriant variants:

ziwm, zkwm
---

ziwm isig, indx [,imix]
zkwm ksig, kndx [,imix]

Like ziw and zkw above, except that they can mix - add the sig to the current value of the variable. If no imix is specified, they mix, but if imix is used, then 0 will cause writing (like ziw and zkw) any other value will cause mixing.

zkmod
---

kr **zkmod** ksig, kzkmod

zkmod is a unit generator intended to facilitate the modulation of one signal by another, where the modulating signal comes from a zk variable. Either additive or mulitiplicative modulation is provided.

ksig is the input signal, to be modulated and sent to the output of the zkmod unit generator.

kzkmod controls which zk variable is used for modulation. A positive value means additive modulation, a negative value means multiplicative modulation. A value of 0 means no change to ksig - it is transferred directly to the output.

For instance kzkmod = 23 will read from zk variable 23, and add the value it finds there to ksig. If kzkmod = -402, then ksig is multiplied by the value read from zk location 402.

kskmod can be an i or a k rate value.

zkcl
---

zkcl kfirst, klast

This will clear to zero one or more variables in the zk space. Useful for those variables which are accumulators for mixing things during the processing for each cycle, but which must be cleared to zero before the next set of calculations.

zar, zarg, zaw, zawm

For a rate reading and writing, in the za space, we use similar opcodes:

ar **zar** kndx

kndx Points to which za variable to read.

This reads the number kndx array of floats in za space which are the ksmps number of audio rate floats to be processed in a k cycle.

ar **zarg** kndx, kgain

Similar to zar, but multiplies the a rate signal by a k rate value kgain.

zaw asig, kndx

Writes asig into the za variable specified by kndx.

zawm asig, kndx [,imix]

Like zaw above, except that it can mix - add the asig to the current value of the destination za variable. If no imix is specified, it mixes, but if imix is used, then 0 will cause a simple write (like zaw) and any other value will cause mixing.

zamod
---

zamod asig, kzamod

Modulation of one audio rate signal by a second one - which comes from a za variable. The location of the modulating variable is controlled by the i or k rate variable kzamod. This is the audio rate version of zkmod described above.

zacl
---

zacl kfirst, klast

This will clear to zero one or more variables in the za space. Useful for those variables which are accumulators for mixing things during the processing for each cycle, but which must be cleared to zero before the next set of calculations.

Summary of zak ugens

What types of input variables are used?

			Runs at time
ir	zir	indx	i
kr	zkr	kndx	k
	ziw	isig, indx	i
	zkw	ksig, kndx	k
	ziwm	isig, indx, imix	i
	zkwm	ksig, kndx, kmix	k
	zkcl	kfirst, klast	k
ar	zar	kndx	k but does arrays
ar	zarg	kndx, kgain	k but does arrays
	zaw	asig, kndx	k but doesarrays
	zawm	asig, kndx, kmix	k but does arrays
	zacl	kfirst, klast	k but does arrays

isig }
indx } Known at init time
imix }

ksig }
kndx }
kmix } k rate variables
kfirst }
klast }
kgain }

asig } a rate variable - an array of floats.

Known bugs in zak system

When using the mix function of zkwm or zawm, care must be taken that
the variables mixed to are zeroed at the end (or start) of each k cycle. The same applies to any variables to which signals are mixed.
If you keep adding signals to them, their values can drift to astronomical figures - which is probably not what you want.

My intention is to have certain ranges of za and zk variables used for mixing - I use zkcl and zacl in the last instrument to clear those ranges.

>>> 5 - Six simple time reading ugens
==========

timek, timek, times, itimes

These read absolute time since the start of the performance - in two formats.

One is timek or itimek for time in krate cycles. So with:

sr = 44100
kr = 6300
ksmps = 7

then after half a second, the timek or itimek ugen would report 3150. It will always report an integer.

Time in seconds is available with times or itimes.

These would return 0.5 after half a second.

kr **timek**
kr **times**

Both the above expect a k rate variable for output.

There are no input parameters.

For similar ugens which only operate at the start of the instance of the instrument:

ir **itimek**
ir **itimes**

Both these expect an i rate variable (starting with i or gi) as their output.

instimek, instimes
---

kr **instimek**
kr **instimes**

These are similar to timek and times, except they return the time since the start of this instance of the instrument.

6 - Printing k rate variables on the screen as numbers
==========

I hate debugging - these ugens are intended to facilitate the debugging of orchestra code.

printk
---

printk prints one k rate value on every k cycle, every second or at intervals specified. First the instrument number is printed, then the absolute time in seconds, then a specified number of spaces, then the value. The variable number of spaces enables different values to be spaced out across the screen - so they are easier to view.

printk kval, ispace [, itime]

kval The number to be printed.

ispace How many spaces to insert before it is printed. (Max 130.)

itime How much time in seconds is to elapse between printings.
(Default 1 second.)

The first print is on the first k cycle of the instance of the instrument. This may not be 0.000 seconds, but the first k cycle afterwards. I want to investigate this - I thought that k rate code should run from time 0.

printks
---

printks is a completely different ugen - similar to printf() in C.

It is highly flexible, and if used together with cursor positioning codes, could be used to write specific values to locations in the screen as the Csound processing proceeds. With MSDOS, a colour screen and ANSI.SYS, it would be possible to have multiple colours, flashing displays - looking like NASA mission control, with k rate variables controlling the values displayed, the location on the screen where they are displayed, their colour etc.

There is also a special mode where a float variable is rounded to the next lowest integer, and ANDed with 0 1111 1111 to produce a character between 0 and 255 to be sent to be printed.

This elaborate use is a bit over the top - a hacker's paradise. But printks can be used simply, just to print variables for debugging.

printks prints numbers and text, with up to four printable numbers - which can be i or k rate values.

printks "txtstring", itime, kval1, kval2, kval3, kval4

txtstring Text to be printed first - can be up to 130 characters at least. _Must_ be in double quotes.

The string is printed as is, but standard printf %f etc. codes are interpreted to print the four parameters.

However (at least with DJGPP) the\n style of character codes are not interpreted by printf. This ugen therefore provides certain specific codes which are expanded:

\n or \N Newline

\t or \T Tab

^ Escape character

^^ ^

~ Escape and '[' These are the lead in codes for MSDOS ANSI.SYS screen control characters.

~~ ~

An init error is generated if the first parameter is not a string of length > 0 enclosed in double quotes.

[For some reason (at least with the DJGPP version, the program crashes if a null string - "" - is given. This seems not to be due to this ugen. This should be tidied up sometime.]

A special mode of operation allows this ugen to convert kval1 input parameter into a 0 to 255 value and to use it as the first character to be printed.

This enables a Csound program to send arbitrary characters to the console - albeit with a little awkwardness.

[printf() does not have a format specifier to read a float and turn it into a byte for direct output. We could add extra code to do this if we really wanted to put arbitrary characters out with ease.]

To acheive this, make the first character of the string a # and then, if desired continue with normal text and format specifiers. Three more format specifers may be used - they access kval2, kval3 and kval4.

itime How much time in seconds is to elapse between printings. (Default 1 second.)

kvalx The k rate values to be printed. Use 0 for those which are not used.

For instance:

printks "Volume = %6.2f Freq = %8.3f\n", 0.1, kvol, kfreq, 0, 0

This would print:

Volume = 1234.56 Freq = 12345.678

printks "#x\\y = %6.2\n", 0.1, kxy, 0, 0, 0

This would print a tab character followed by:

x\y = 1234.56

Discussion

Both these printing ugens can be made to run on every k cycle - or at least every k cycle they are run in the instrument. Conditional goto statements can be used to run them only at certain times or when something goes wrong. To make them run on every k cycle like this, set itime to 0.

When itime is not 0, then (if the orchestra code runs the ugen on every k cycle) then the ugen will decide when to print. It will always print on the first k cycle it is called. This means that if you set one of these to print only every 10 seconds, and conditional code in the instrument causes it to be run for the very first time at 3 seconds, then it will print at 3 seconds.

Subsequent runs of the ugen at between 3 and 9.999 seconds would not
cause it to print. This could be very useful - set the time to longer than the piece and conditional code in the instrument can be used to report a bug just once, on its first occurrence. You almost certainly do not want a print operation happening every k cycle - it slows the program down too much.

Staying with the 10 second cycle example, if such a printk or printks ugen was called every k cycle, then it would print at 0 seconds

(actually the first k cycle after 0), at 10.0 seconds, at 20.0 seconds etc.

The time cycles start from the time the ugen is initialized - typically the initialisation of the instrument.

Damien Miller pointed out an interesting application of these ugens - get the output of the program and sort the lines with a line sorter.
The result would be the printed lines sorted first by instrument number, and then by time - for printk. However printks can be made to produce almost anything. The instrument is available as p1 and the time can easily be found and made available as a printks parameter.

One option I have considered but not implemented is for these printed lines to be written to a file as well as to the screen. Let me know if you like this idea, or have any other ideas about debugging.

printf() style %f formatting

One of the less enjoyable parts of C programming is trying to figure out what magic incantations to offer to printf()

All the parameters are floats, so this reduces our decisions to two main issues:

1 - How many decimal points of precision do we want? (0 means no decimal point.)

2 - How many digits (or spaces) do we want printed in total - _including_ those after the decimal point?

%f Just prints with full precision - 123.123456

%6.2f Prints 1234.12

%5.0f Prints 12345

There is more to the printf() codes than this - see an ANSI C library reference. Instead of 'f', you can use 'e' to get scientific notation. Using any other format specifiers than f, e, g, E and G will cause unpredictable results, because the parameters are always floating point numbers.

>>> 7 - Why arrays or "zak" are so important for some applications
==========

A major theme of my approach to making music is to set up processes
and let them interact and be affected by random occurrences. This can
be expensive in analog hardware - but a load of fun too.

Setting up a garden of interacting processes and then tweaking them to
whatever state of control or chaos I like is my idea of fun!

Lets say I want to set up a musical cellular automata - with 100 similar cells.

Each one produces sound and has various internal states stored as i, k or a rate variables. The behaviour of each cell is at least partially dependant on that of its neighbours. Typically, each cell would make some of its own internal state - including sound output - _readable_ by its neigbours or other things.

There could be a global matron function who tries to control the cells' level of friskiness if they individually or collectively incur her wrath by becoming too obstreperous.

So I have a 10 x 10 array of cells, and their internal state is made available as global variables - with different names for the same variable in different cells.

This could be done with 100 carefully written instruments, but life is too short.
The only alternative is to use one instrument and have each instance decide where its interal states are written to for others to read.
It should decide which of the 99 other instances it will read the states of.

The ideal way is if we could write global variables as:

gahuey[p7] = afoo * abar

or

gahuey[kdest] = afoo * abar

In either case, one element of an array huey[] of a rate variables is written. (Actually each variable is an array of ksmps floats.)

[Interlude 1 - from what are the popular C variables foo and bar
[derived? See the end of the file.

Likewise we want to be able to write these array specifications in the right hand of equations.

gaduey[kdest] = huey[ksource] * (ablah + p4)

So that is the first thing about arrays - make them easy and direct to use with i or k rate indexing.

Secondly, make them multidimensional:

galouey[4, 10]
Is a two dimensional array of global audio rate variables.

gkblah[2, 4, 10]
Is a three dimensional array of global k rate variables.

Thirdly, we want them to be either global or local to the instance of the instrument.

This is quite a tall order, since the core of Csound is not perfect and is largely devoid of comments. Such facilities are obviously beyond what Csound was originally conceived to do, but now that CPUs are so much faster, many people will be writing more sophisticated programs. Since PCs with dual Pentiums exist today, and in a year or two will be available with up to four P6 processors, lets think big!

In principle, the global aspect of arrays can be acheived with the zak system, but it is trickier.

zak ugens do not go on the left or right of equations, they have their own line. They must write to normal variables and be fed by normal variables. Arrays, and multi dimensional arrays can all be done with offsets and multiplications to arrive at the final number of the location in za or zk space - but it this involves bulky, hard do debug and understand .orc code, and there is no prospect for building mnemonic names into the way these variables are accessed.

I intend to do some cellular automatata or use multiple reverb and sound source instruments with varying delay times between them, all mixed with my binaural model - with the instruments, reverb points (and hence their connecting time delays) potentially moving around.

There are great prospects for many hours of programming work, bogging down the CPU, and probably horrible results - but I am intrigued.

Release Notes for 3.47
==========

These are the release notes for version 3.47.
Many internal changes made to remove compiler warnings. Mainly declarations and prototypes. Anyone who works at source level should beware as structures have new fields, fields have been removed and so on. Some variables have type changed.
The Windows GUI has been revised as well.

Language Changes:

1. Comments allowed in score in more places
2. Treat \ at end of line as continuation in orchestra
3. Maximum number of an instrument is dynamic, and expands as needed.
4. Removed limit on number of labels
5. Introduced ^ syntax into score files
6. Two new GENs, numbered 25 and 27
7. No limit to number of partials in hetro/adsyn

Opcode Fixes

1. Some fixes in fog
2. Internal tidying in granule
3. Bug fix in cpsxpch
4. Fix problem with tables 300, 600, 900,...

New Opcodes

The following have been added.

acos	**asin**	**atan**	**birrnd**
chanctrl	**cosh**	**cross2**	**ctlinit**
ctrl14	**ctrl21**	**ctrl7**	**dam**
expsegr	**filter2**	**ftgen**	**ftlptim**
harmon	**ictrl14**	**ictrl21**	**ictrl7**
imidic14	**imidic21**	**imidic7**	**initc14**
initc21	**initc7**	**ioff**	**ion**
iondur	**iondur2**	**ioutat**	**ioutc**
ioutc14	**ioutpat**	**ioutpb**	**ioutpc**
ipchbend	**kfilter2**	**kon**	**koutat**
koutc	**koutc14**	**koutpat**	**koutpb**
koutpc	**kpchbend**	**linsegr**	**massign**
mclock	**midic14**	**midic21**	**midic7**
moscil	**mrtmsg**	**osciln**	**release**
repluck	**rnd**	**sinh**	**tanh**
turnon	**wgpluck**	**wgpuck2**	**xtratim**

and the modelled opcodes (following Perry Cook)

wgclar	**wgflute**	**wgbow**	**wgbrass**
marimba	**vibes**	**agogobel**	**shaker**
fmbell	**fmrhode**	**fmwurlie**	**fmmetal**
fmb3	**fmvoice**	**fmpercfl**	**moog**
mandol	**voice**		

Revised Opcodes:

The opcodes here have had their specification changed, generally in a compatible way. Usually to add scale factors and the like

imidictrl	**kmidictrl**	**linseg**	**pchbend**
printks	**veloc**		

Other Changes:

Revised realtime audio on Windows (again!)
Additional features on Windows GUI to includes access to orchestra and score editing and post-calculation soundfile editing.
Emacs orchestra mode expanded for new opcodes, and fixed a little ksmps constrained to be integer.

Details on Opcodes
==========

atan(x), acos(x), asin(x), tanh(x), sinh(x), cosh(x)

Functions to calculate the arctangent, etc. Available in i, k and a forms.
(Author JPff)

irnd(x), krnd(x), ibirnd(x), kbirnd()

Functions, return random values in the range [0,x) or (-x,x)

chanctrl
---
ival chanctrl ichnl, ictlno[,ilow,ihigh]
kval chanctrl ichnl, ictlno[,ilow,ihigh]

Get the current value of a controller and optionally map it onto specified range. ichnl is the MIDI channel and ictlno is the MIDI controller number.
(Author BV)

cross2
---
asig cross2 ain1, ain2, ilen, iovl, iwin, kbias

Cross synthesis between the two audio signals
(Author PS)

ctrtlinit
---

ctrlinit ichnkm, ictlno1, ival1[, ictlno2, ival2[, ictlno3, ival3[,..ival32]]

Sets initial values for a set of MIDI controllers.
(Author BV)

dam

Dynamic amplitude modifier
ar dam ain, kthresh, icomp1, icomp2, irtme, iftme

(Author MR)

filter, **kfilter**, **zfilter**
---

a1 filter2 asig, iM, iN, ib0, ib1, ..., ibM, ia1, ia2, ..., iaN
k1 kfilter2 ksig, iM, iN, ib0, ib1, ..., ibM, ia1, ia2, ..., iaN
a1 zfilter2 asig, kdamp, kfreq, iM, iN, ib0, ib1,..,ibM, ia1, ia2, .,iaN

General purpose custom filter with time-varying pole control. The filter coefficients implement the following difference equation:

(1)*y(n) = b0*x[n] + b1*x[n-1] + ... + bM*x[n-M] - a1*y[n-1] - ... - aN*y[n-N]

the system function for which is represented by:

$$H(Z) = \frac{B(Z)}{A(Z)} = \frac{b0 + b1*Z^{-1} + ... + bM*Z^{-M}}{1 + a1*Z^{-1} + ... + aN*Z^{-N}}$$

INITIALIZATION

At initialization the number of zeros and poles of the filter are specified along with the corresponding zero and pole coefficients. The coefficients must be obtained by an external filter-design application such as Matlab and specified directly or loaded into a table via gen01. With zfilter2, the roots of the characteristic polynomials are solved at initialization so that the pole-control operations can be implemented efficiently.

PERFORMANCE

The filter2 and kfilter2 opcodes perform filtering using a transposed form-II digital filter lattice with no time-varying control. zfilter2 uses the additional operations of radial pole-shearing and angular pole-warping in the Z plane.

Pole shearing increases the magnitude of poles along radial lines in the Z-plane. This has the affect of altering filter ring times. The k-rate variable kdamp is the damping parameter. Positive values (0.01 to 0.99) increase the ring-time of the filter (hi-Q), negative values (-0.01 to -0.99) decrease the ring-time of the filter, (lo-Q).

Pole warping changes the frequency of poles by moving them along angular paths in the Z plane. This operation leaves the shape of the magnitude response unchanged but alters the frequencies by a constant factor (preserving 0 and p). The k-rate variable k-freq determines the frequency warp factor. Positive values (0.01 to 0.99) increase frequencies toward p and negative values (-0.01 to -0.99) decrease frequencies toward 0.

Since filter2 implements generalized recursive filters, it can be used to specify a large range of general DSP algorithms. For example, a digital waveguide can be implemented for musical instrument modeling using a pair of delayr and delayw opcodes in conjunction with the filter2 opcode.

Examples:

A first-order linear-phase lowpass linear-phase FIR filter operating on a k-rate signal:

```
k1  kfilter2  ksig, 2, 0, 0.5, 0.5 ;; k-rate FIR filter
```

A controllable second-order IIR filter operating on an a-rate signal:

```
a1  zfilter2  asig, kdamp, kfreq, 1, 2, 1, ia1, ia2 ;; controllable IIR
```

(Author MKC)

ftgen
---

iafno ftgen ifno,itime,isize, igen, iarga[,...iargz]

iafno is either a requested or automatically assigned table number above 100. If ifno is zero the number is assigned automatically and the value placed in iafno, Any other value is used as the table. itime is ignored, but otherwise this is as the table generation in the score with the f statement.
(Author BV)

ftlptim

Function. Returns the loop segment start-time in seconds of a stored table.
(Author BV)

harmon
---

ar harmon asig,kestfrq,kmaxvar, kgenfrq1, kgenfrq2, imode, iminfrq, iprd

Analyse an audio input and generate harmonising voices in synchrony.

imode=0 is to treat the 2 generated frequencies as ratios=1 they are cps iminfrq is the lowest expects frequency in cps
iprd is the period of analysis

kestfrq is an estimate of the input frequency, and kmaxvar is a ratio to limit the search.
Only one voice may be higher that the the signal, and a zero frequency silences the sound

(Author BV)

ictrl7, **ictrl21**, **ictrl14**, **ctrl7**, **ctrl21**, **ctrl14**
imidic7, **imidic14**, **imidic21**, **midic7**, **midic14**, **midic21**

```
idest       imidic7     ictlno, imin, imax [, ifn]
kdest       midic7      ictlno, kmin, kmax [, ifn]

idest       imidic14    ictlno1, ictlno2, imin, imax [, ifn]
kdest       midic14     ictlno1, ictlno2, kmin, kmax [, ifn]

idest       imidic21    ictlno1, ictlno2, ictlno3, imin, imax [, ifn]
kdest       midic21     ictlno1, ictlno2, ictlno3, kmin, kmax [, ifn]


idest    ictrl7  ichan, ictlno, imin, imax [,ifn]
kdest     ctrl7   ichan, ictlno, kmin, kmax [,ifn]

idest    ictrl14  ichan, ictlno1, ictlno2, imin, imax [,ifn]
kdest     ctrl14   ichan, ictlno1, ictlno2, kmin, kmax [,ifn]

idest    ictrl21  ichan, ictlno1, ictlno2, ictlno3, imin, imax [,ifn]
kdest     ctrl21  ichan, ictlno1, ictlno2, ictlno3, kmin, kmax [,ifn]
```

DESCRIPTION
Allow precise MIDI input controller signal.

INITIALIZATION
idest - output signal
ichan - MIDI channel (in (i)ctrl14 and (i)ctrl21 all the controllers used in an opcode instance must be of the same channel)

ictlno - midi controller number (1-127)
ictlno1 - most-significant byte controller number (1-127)
ictlno2 - in midic14: less-significant byte controller number (1-127); in midic21: mid-significant byte controller number (1-127)
ictlno3 - less-significant byte controller number (1-127)
imi - user-defined minimum floating-point value of output
imax - user-defined maximum floating-point value of output
ifn (optional) - table to be read when indexing is required. Table must be normalized. Output is scaled according to max and min val.

PERFORMANCE
kdest - output signal
kmin - user-defined minimum floating-point value of output
kmax - user-defined maximum floating-point value of output

imidic7 and midic7 (i and k rate 7 bit midi control) allow floating point 7 bit midi signal scaled with a minimum and a maximum range. They also allow optional non-interpolated table indexing.
In midic7 minimum and maximum values can be varied at krate.

imidic14 and midic14 (i and k-rate 14 bit midi control) do the same as the above with 14 bit precision.

imidic21 and midic21 (i and k rate 21 bit midi control) do the same as the above with 21 bit precision.

imidic14, midic14, imidic21 and midic21 can use optional interpolated table indexing. They require two or three midi controllers as input.

Warning! Don't use (i)midicXX opcodes within a sco-activated i-statement or Csound will crash. Instruments containing (i)midicXX opcodes can be only activated by a MIDI note-on message. Use (i)ctrlXX opcodes if you need to include them in a sco-oriented instrument instead.

ictrl7, ctrl7, ictrl14, ctrl14, ictrl21, ctrl21 are very similar to (i)midicXX opcodes the only differences are:

1) (i)ctrlXX UGs can be included in sco-oriented instruments without Csound crashes.
2) They need the additional parameter ichan containing the MIDI channel of the controller. MIDI channel is the same for all the controller used in a single (i)ctrl14 or (i)ctrl21 opcode.

initc7, **initc14**, **initc21**
---
initc7 ichan, ictlno, ivalue
initc14 ichan, ictlno1, ictlno2, ivalue
initc21 ichan, ictlno1, ictlno2, ictlno3, ivalue

DESCRIPTION
Initializes MIDI controller ictlno with ivalue

INITIALIZATION
ichan - midi channel
ictlno - controller number (initc7)
ictlno1 - MSB controller number
ictlno2 - in initc14 LSB controller number; in initc21 Medium Significant Byte controller number
ictlno3 - LSB controller number
ivalue - floating point value (must be within 0 to 1)

initc7, initc14, initc21 can be used together with both (i)midicXX and
(i)ctrlXX opcodes for initializing the first controllers' value.
Ivalue argument must be set with a number within 0 to 1. An error occurs if it is not.
Use the following formula to set ivalue according with (i)midicXX and (i)ctrlXX min and max range:

ivalue = (initial_value - min) / (max - min)

ion, **ioff**, **iondur**, **iondur2**
---
ion ichn, inum, ivel
ioff ichn, inum, ivel
iondur ichn, inum, ivel, idur
iondur2 ichn, inum, ivel, idur

DESCRIPTION
send note-on and note-off messages to the MIDI OUT port.

INITIALIZATION

ichn - MIDI channel number (0-15)
inum - note number (0-127)
ivel - velocity (0-127)

PERFORMANCE

ion (i-rate note on) and ioff (i-rate note off) are the simplest MIDI OUT opcodes.
ion sends a MIDI noteon message to MIDI OUT port, and ioff sends a
noteoff message.
A ion opcode must always be followed by an ioff with the same channel and number inside the same instrument, otherwise the note will play endlessly.
These ion and ioff are useful only when introducing a timout statement to play a non zero duration MIDI note.
For most purposes it is better to use iondur and iondur2.

iondur and iondur2 (i-rate note on with duration) send a noteon and a noteoff MIDI message both with the same channel, number and velocity. Noteoff message is sent after idur seconds are elapsed by the time iondur was activated.

iondur differs from iondur2 in that iondur truncates note duration when current instrument is deactivated by score or by realtime playing, while iondur2 will extend performance time of current instrument until idur seconds have elapsed. In realtime playing it is suggested to use iondur also for undefined durations, giving a large idur value.

Any number of iondur or iondur2 opcodes can appear in the same Csound instrument, allowing chords to be played by a single instr.

ioutc, **ioutc14**, **koutc**, **koutc14**, **ioutpb**, **koutpb**, **ioutat**, **koutat**, **ioutpc**, **koutpc**, **ioutpat**, **koutpat**
---
ioutc ichn, inum, ivalue, imin, imax
koutc kchn, knum, kvalue, kmin, kmax
ioutc14 ichn, imsb, ilsb, ivalue, imin, imax
koutc14 kchn, kmsb, klsb, kvalue, kmin, kmax

ioutpb ichn, ivalue, imin, imax
koutpb kchn, kvalue, kmin, kmax
ioutat ichn, ivalue, imin, imax
koutat kchn, kvalue, kmin, kmax
ioutpc ichn, iprog, imin, imax
koutpc kchn, kprog, kmin, kmax

ioutpat ichn, inotenum, ivalue, imin, imax
koutpat kchn, knotenum, kvalue, kmin, kmax

DESCRIPTION
Send a single Channel message to the MIDI OUT port.

INITIALIZATION AND PERFORMANCE

ichn, kchn - MIDI channel number (0-15)
inum, knum - controller number (0-127 for example. 1 = ModWheel; 2 = BreathControl etc.)
ivalue, kvalue - floating point value
imin, kmin - minimum floating point value (converted in midi integer value 0)
imax, kmax - maximum floating point value (converted in midi integer value 127 (7 bit) or 16383 (14 bit))
imsb, kmsb - most significant byte controller number when using 14 bit parameters
ilsb, klsb - less significant byte controller number when using 14 bit parameters
iprog, kprog - program change number in floating point
inotenum, knotenum - MIDI note number (used in polyphonic aftertouch messages)

ioutc and koutc (i and k-rate midi controller output) send controller messages to MIDI OUT device.

iout14 and kout14 (i and k-rate midi 14 bit controller output) send a pair of controller messages. These opcodes can drive 14 bit parameters on MIDI instruments that recognize them. The first control message contains the most significant byte of i(k)value argument while the second message contains the less significant byte. i(k)msb and i(k)lsb are the number of the most and less significant controller.

ioutpb and koutpb (i and k-rate pitch bend output) send pitch bend messages.
ioutat and koutat (i and k-rate aftertouch output) send aftertouch messages.
ioutat and koutat (i and k-rate aftertouch output) send aftertouch messages.
ioutpc and koutpc (i and k-rate program change output) send program change messages.

ioutpat and koutpat (i and k-rate polyphonic aftertouch output) send polyphonic aftertouch messages. These opcodes can drive a different value of a parameter for each note currently active. They work only with MIDI instruments which recognize them.

N.B. All these opcodes can scale the i(k)value floating-point argument according with i(k)max and i(k)min values. For example, setting i(k)min = 1.0 and i(k)max = 2.0, when i(k)value argument receives a 2.0 value, the opcode will send a 127 value to MIDI OUT

device, while when receiving a 1.0 it will send a 0 value. I-rate opcodes send their message once during instrument initialization.
K-rate opcodes send a message each time the MIDI converted value of
argument i(k)value changes.

ipchbend, **kpchbend**
---

ibend ipchbend [ilow, ihigh]
kbend kpchbend [ilow, ihigh]

Get the current pitchbend value from a MIDI channel, and map it to the specified range
(Author BV)

kon, **moscil**
---
moscil kchn, knum, kvel, kdur, kpause
kon kchn, knum, kvel

DESCRIPTION
Send stream of note-on and note-off messages to the MIDI OUT port.

INITIALIZATION

PERFORMANCE
kchn - MIDI channel number (0-15)
knum - note number (0-127)
kvel - velocity (0-127)
kdur - note duration in seconds
kpause - pause duration after each noteoff and before new note in seconds

moscil and kon are the most powerful MIDI OUT opcodes.
moscil (midi oscil) plays a stream of notes of kdur duration. Channel, pitch, velocity, duration and pause can be controlled at k-rate, allowing very complex algorithmically generated melodic lines.
When current instrument is deactivated, the note played by current instance of moscil is forcely truncated.

kon (k-rate note on) plays MIDI notes with current kchn, knum and kvel.
These arguments can be varied at k-rate. Each time the MIDI converted value of any of these arguments changes, last MIDI note played by current instance of kon is immediately turned off and a new note with the new argument values is activated.

This opcode, as well as moscil, can generate very complex melodic textures if controlled by complex k-rate signals.

Any number of moscil or kon opcodes can appear in the same Csound instrument, allowing a counterpoint-style polyphony within a single instrument.

(Author GM)

linsegr,**expsegr**
---

kr linsegr ia, idur1, ib[,idur2, ic[..]]. irel, iz
ar linsegr ia, idur1, ib[,idur2, ic[..]]. irel, iz
kr expsegr ia, idur1, ib[,idur2, ic[..]]. irel, iz
akr expsegr ia, idur1, ib[,idur2, ic[..]]. irel, iz

Like linseg except that on a MIDI note off event the release sectin is used, extending the performance by irel seconds, during which the value of the opcode changes to iz.
(Author BV)

massign
---

massign ichnl, insno

Assign MIDI channel to a Csound instrument. This is an orchestral header statement
(Author BV)

mclock, **mrtmsg**
---
mclock ifreq
mrtmsg imsgtype

DESCRIPTION
Send System Realtime messages to the MIDI OUT port.

INITIALIZATION

ifreq - clock message frequency rate in Hz
imsgtype - type of real-time message:
1 sends a START message (0xFA);
2 sends a CONTINUE message (0xFB);
0 sends a STOP message (0xFC);
-1 sends a SYSTEM RESET message (0xFF);
-2 sends an ACTIVE SENSING message (0xFE)

PERFORMANCE

mclock (midi clock) sends a MIDI CLOCK message (0xF8) every 1/ifreq seconds. So ifreq is the frequency rate of CLOCK message in Hz.
mrtmsg (midi realtime message) sends a realtime message once, in init stage of current instrument. imsgtype parameter is a flag to indicate the message type (see above, in ARGUMENTS description).

(Author GM)

osciln
---

ar osciln kamp, ifrq, ifn, itimes

Like oscil1, but makes a total of itimes passes over the data, after which it is silent
(Author BV)

repluck, **wgpluck2**
---

wgpluck2 is an implementation of the physical model of the plucked string, with control over the pluck point, the pickup point and the filter. repluck is the same operation, but with an additional audio signal used to excite the 'string'

ar wgpluck2 iplk, xam, icps, kpick, krefl
ar repluck iplk, xam, icps, kpick, krefl, axcite

The string plays at icps pitch. The point of pluck is iplk, which is a fraction of the way up the string (0 to 1). A pluck point of zero means no initial pluck. xamp is the gain. and kpick is what proportion of the way along the string to sample the output. The relection at the bridge is contrlled by the refleaction coefficient, where 1 means total reflection and 0 is totally dead.

(Author JPff)

turnon
---

turnon insno[,itime]

Activate an instrument, for an indefinite time, after a delay of itime seconds.
(Author BV)

wgpluck

(Author MKC)

xtratim, **release**

```
---------------
          xtratim     iextradur
kflag     release
```

DESCRIPTION
Extend the duration of realtime generated events and handle their extra life.

INITIALIZATION

iextradur - additional duration of current instrument instance.

PERFORMANCE

xtratim exetends current MIDI-activated note duration of iextradur seconds after the corresponding note-off message has deactivated current note itself. This opcode has no output arguments.

release outputs current note state. If current note is in the release stage (i.e. if its duration has been exetended with xtratim opcode and if it has only just deactivated), kflag output argument is set to 1, else (in sustain stage of current note) is set to 0.

These two opcodes are useful for implementing complex release-oriented envelopes.

Example:

```
instr     1    ;allows complex ADSR envelope with MIDI events
inum      notnum
icps      cpsmidi
iamp      ampmidi   4000
;
;############### complex envelope section ##############
          xtratim   1           ;extra-time, i.e. release dur
krel      init      0
krel      release   ;outputs release-stage flag (0 or 1 values)
if        (krel > .5) kgoto  rel    ;if in relase-stage goto relase section
;
;************* attack and sustain section ************
kmp1      linseg    0,.03,1,.05,1,.07,0,.08,.5,4,1,50,1
kmp       =         kmp1*iamp
          kgoto     done
;
;************* release section ***********************
rel:
kmp2      linseg    1,.3,.2,.7,0
kmp       = kmp1*kmp2*iamp
done:
;#################################################
;
a1        oscili    kmp, icps, 1
          out       a1
          endin
```

ar **wgclar** kamp, kfreq, kstiff, iatt, idetk, kngain, kvibf, kvamp, ifn[, iminfreq]

Audio output is a tone similar to a clarinet, using a physical model developed from Perry Cook, but re-coded for Csound.

Initialisation

iatt - time in seconds to reach full blowing pressure. 0.1 seems to correspond to reasonable playing. A longer time gives a definite initial wind sound.

idetk - time in seconds taken to stop blowing. 0.1 is a smooth ending

ifn - table of shape of vibrato, usually a sine table, created by a function

iminfreq - lowest frequency at which the instrument will play. If it is omitted it is taken to be the same as the initial kfreq.

Performance

A note is played on a clarinet-like instrument, with the arguments as below.

kamp - Amplitude of note.

kfreq - Frequency of note played. While it can be varied in performance, I have not tried it.

kstiff - a stiffness parameter for the reed. Values should be negative, and about -0.3. The useful range is approximately -0.44 to -0.18.

kngain - amplitude of the noise component, about 0 to 0.5

kvibf - frequency of vibrato in Hertz. Suggested range is 0 to 12

kvamp - amplitude of the vibrato

Example:

```
a1        wgclar    31129.60, 440, -0.3, 0.1, 0.1, 0.2, 5.735, 0.1, 1
          out       a1
```

---

ar **wgflute** kamp, kfreq, kjet, iatt, idetk, kngain, kvibf, kvamp, ifn[, iminfreq]

Audio output is a tone similar to a flute, using a physical model developed from Perry Cook, but re-coded for Csound.

Initialisation

iatt - time in seconds to reach full blowing pressure. 0.1 seems to correspond to reasonable playing.

idetk - time in seconds taken to stop blowing. 0.1 is a smooth ending.

ifn - table of shape of vibrato, usually a sine table, created by a function.

iminfreq - lowest frequency at which the instrument will play. If it is omitted it is taken to be the same as the initial kfreq.

Performance

A note is played on a flute-like instrument, with the arguments as below.

kamp - Amplitude of note.

kfreq - Frequency of note played. While it can be varied in performance, I have not tried it.

kjet - a parameter controlling the air jet. Values should be positive, and about 0.3. The useful range is approximately 0.08 to 0.56.

kngain - amplitude of the noise component, about 0 to 0.5

kvibf - frequency of vibrato in Hertz. Suggested range is 0 to 12

kvamp - amplitude of the vibrato.

Example:

```
a1        wgflute   31129.60, 440, 0.32, 0.1, 0.1, 0.15, 5.925, 0.05, 1
          out       a1
```

---

ar **wgbow** kamp, kfreq, kpres, krat, kvibf, kvamp, ifn[, iminfreq]

Audio output is a tone similar to a bowed string, using a physical model developed from Perry Cook, but re-coded for Csound.

Initialisation

ifn - table of shape of vibrato, usually a sine table, created by a function.

iminfreq - lowest frequency at which the instrument will play. If it is omitted it is taken to be the same as the initial kfreq.

Performance

A note is played on a bowed string-like instrument, with the arguments as below.

kamp - Amplitude of note.

kfreq - Frequency of note played. While it can be varied in performance, I have not tried it.

kpres - a parameter controlling the pressure of the bow on the string. Values should be about 3. The useful range is approximately 1 to 5.

kratio - the position of the bow along the string. Usual playing is about 0.127236. The suggested range is 0.025 to 0.23.

kvibf - frequency of vibrato in Hertz. Suggested range is 0 to 12.

kvamp - amplitude of the vibrato.

Example:
A bowing with vibrato setting in after a short time.

```
kv   linseg     0, 0.5, 0, 1, 1, p3-0.5, 1
a1   wgbowed    31129.60, 440, 3.0, 0.127236, 6.12723, kv*0.01, 1
     out        a1
```

---

ar **wgbrass** kamp, kfreq, klipt, idatt, kvibf, kvamp, ifn[, iminfreq]

Audio output is a tone related to a brass instrument, using a physical model developed from Perry Cook, but re-coded for Csound.
[NOTE: This is rather poor, and at present uncontrolled. Needs revision, and possibly more parameters]

Initialisation

idatt -- time taken to stop blowing.

ifn - table of shape of vibrato, usually a sine table, created by a function.

iminfreq - lowest frequency at which the instrument will play. If it is omitted it is taken to be the same as the initial kfreq.

Performance

A note is played on a bowed string-like instrument, with the arguments as below.

kamp - Amplitude of note.

kfreq - Frequency of note played. While it can be varied in performance, I have not tried it.

klibt - tension of lips, in range 0 to 1.

kvibf - frequency of vibrato in Hertz. Suggested range is 0 to 12.

kvamp - amplitude of the vibrato.

Example:

```
a1   wgbrass    31129.60, 440, 0.4, 0.1, 6.137, 0.05, 1
```

---

ar **marimba** kamp, kfreq, ihrd, ipos, imp, kvibf, kvamp, ivibfn, idec
ar **vibes** kamp, kfreq, ihrd, ipos, imp, kvibf, kvamp, ivibfn, idec

Audio output is a tone related to the striking of a wooden or metal block as found in a marimba or vibraphone. The method is a physical model developed from Perry Cook, but re-coded for Csound.

Initialisation

ihrd -- the hardness of the stick used in the strike. A range of 0 to 1 is used. 0.5 is a suitable value.

ipos -- where the block is hit, in the range 0 to 1.

imp - a table of the strike impulses. The file "marmstk1.wav" is a suitable function from measurements, and can be loaded with a GEN1 table.

ivfn - shape of vibrato, usually a sine table, created by a function.

idec - time before end of note when damping is introduced.

Performance

A note is played on a marimba-like instrument, with the arguments as below.

kamp - Amplitude of note.

kfreq - Frequency of note played. While it can be varied in performance, I have not tried it.

kvibf - frequency of vibrato in Hertz. Suggested range is 0 to 12.

kvamp - amplitude of the vibrato.

Example:

```
a1   marimba 31129.60, 440, 0.5, 0.561, 2, 6.0, 0.05, 1, 0.1
a2   vibes   31129.60, 440, 0.5, 0.561, 2, 4.0, 0.2, 1, 0.1
```

---

ar **agogobel** kamp, kfreq, ihrd, ipos, imp, kvibf, kvamp, ivibfn

Audio output is a tone related to the striking of a cow bell or similar. The method is a physical model developed from Perry Cook, but re-coded for Csound.

Initialisation

ihrd -- the hardness of the stick used in the strike. A range of 0 to 1 is used. 0.5 is a suitable value.

ipos -- where the block is hit, in the range 0 to 1.

imp - a table of the strike impulses. The file "britestk.wav" is a suitable function from measurements, and can be loaded with a GEN1 table.

ivfn - shape of vibrato, usually a sine table, created by a function.

Performance

A note is played on a cowbell or agogobell-like instrument, with the arguments as below.

kamp - Amplitude of note.

kfreq - Frequency of note played. While it can be varied in performance, I have not tried it.

kvibf - frequency of vibrato in Hertz. Suggested range is 0 to 12.

kvamp - amplitude of the vibrato.

Example:
a1 agogobel 31129.60, 440, p4, 0.561, 3, 6.0, 0.3, 1

---

ar **shaker** kamp, kfreq, kbeans, kdamp, knum, ktimes[, idecay]

Audio output is a tone related to the shaking of a maraca or similar gourd instrument. The method is a physically inspired model developed from Perry Cook, but re-coded for Csound.

Initialisation

idecay - If present indicates for how long at the end of the note the shaker is to be damped. The default value is zero.

Performance

A note is played on a cowbell or agogobell-like instrument, with the arguments as below.

kamp - Amplitude of note.

kfreq - Frequency of note played, that is the frequency of the gourd. It can be varied in performance, I have not tried it.

kbeans - The number of beans in the gourd. A value of 8 seems suitable.

kdamp -- The damping vsalue of the shaker. Values of 0.98 to 1 seems suitable, with 0.99 a reasonable default.

knum -- The number of shakes of the gourd. Values over 64 are considered infinite.

ktimes -- Number of times shaken.

[RICK: Not sure all these are useful -- not clear in code]

Example:
a1 shaker 31129.60, 440, 8, 0.999, 0, 100, 0

---

a1 **fmtbell** kamp, kfreq, kc1, kc2, kvdepth, kvrate, ifn1, ifn2, ifn3, ifn4, ivfn
a1 **fmrhode** kamp, kfreq, kc1, kc2, kvdepth, kvrate, ifn1, ifn2, ifn3, ifn4, ivfn
a1 **fmwurlie** kamp, kfreq, kc1, kc2, kvdepth, kvrate, ifn1, ifn2, ifn3, ifn4, ivfn
a1 **fmmetal** kamp, kfreq, kc1, kc2, kvdepth, kvrate, ifn1, ifn2, ifn3, ifn4, ivfn
a1 **fmb3** kamp, kfreq, kc1, kc2, kvdepth, kvrate, ifn1, ifn2, ifn3, ifn4, ivfn
a1 **fmpercfl** kamp, kfreq, kc1, kc2, kvdepth, kvrate, ifn1, ifn2, ifn3, ifn4, ivfn

A family of FM sounds, all using 4 basic oscilators and various architectures, as used in the TX81Z synthesiser.

Initialisation

All these opcodes take 5 tables for initialisation. The first 4 are the basic inputs and the last is the low frequency oscillator (LFO) used for vibrato. The last table should usually be a sine wave.

For the other opcodes the initial waves should be as in the table

	ifn1	ifn2	ifn3	ifn4
fmtbell	sinewave	sinewave	sinewave	sinewave
fmrhode	sinewave	sinewave	sinewave	fwavblnk
fmwurlie	sinewave	sinewave	sinewave	fwavblnk
fmmetal	sinewave	twopeaks	twopeaks	sinewave
fmb3	sinewave	sinewave	sinewave	sinewave
fmpercfl	sinewave	sinewave	sinewave	sinewave

The sounds produced are then

fmtbell Tubular Bell
fmrhode Fender Rhodes Electric Piano
fmwurlie Wurlitzer Electric Piano
fmmetal "Heavy Metal"
fmb3 Hammond B3 organ
fmpercfl Percussive Flute

Performance

kamp – Amplitude.

kfreq – frequency.

kc1, kc2 -- Controls for the syntheser, as in the table

	kc1	kc2	Algorithm
fmtbell	Mod index 1	Crossfase of two outputs	5
fmrhode	Mod index 1	Crossfase of two outputs	5
fmwurlie	Mod index 1	Crossfase of two outputs	5
fmmetal	Total mod index	Crossfade of two modulators	3
fmb3	Total mod index	Crossfade of two modulators	4
fmpercfl	Total mod index	Crossfade of two modulators	4

kvdepth -- Vibrator depth.

kvrate -- Vibrator rate.

Examples:
a1 fmtbell 31129.60, 440, 1, 1.2, 0.2, 6, 1,1,1,1, 1
a1 fmrhode 31129.60, 440, 1, 1.2, 0.2, 12, 1,1,1,4, 1
a1 fmwurlie 31129.60, 440, 1, 1.2, 0.2, 8, 1,1,1,4, 1
a1 fmmetal 31129.60, 110, 1, 1.2, 0.2, 5.5, 1,5,5,1, 1
a1 fmb3 31129.60, 440, 1, 1.2, 0.2, 8, 1,1,1,1, 1
a1 fmpercfl 31129.60, 440, 0.1, 0.1, 0.5, 12, 1,1,1,1, 1

---

a1 **fmvoice** kamp, kfreq, kvowel, ktilt, kvibamt, kvibrate, ifn1, ifn2, ifn3, ifn4, ivibfn

FM Singing Voice Synthesis,
Initialisation:

ifn1, ifn2, ifn3, ifn3 -- Tables, usually of sinewaves.

Performance

kamp -- Amplitude control.

kfreq -- Base frequency of sound.

kvowel -- the vowel being sung, in the range 0-64; it is rounded to the nearest integer.

ktilt -- the spectral tilt of the sound in the range 0 to 99.

kvibamt -- Depth of vibrato.

kvibrate -- Rate of vibrato.

Example

k1 line 0, p3, 64
a1 fmvoice 31129.60, 110, k1, 0, 0.005, 6, 1,1,1,1,1

---

a1 **moog** kamp, kfreq, kfiltq, kfiltrate, kvibf, kvamp, iafn, iwfn, ivfn

An emulation of a mini-Moog syntheser.

Initialisation.

iafn, iwfn, ivfn -- three table numbers containing the attack wave form (unlooped), the main looping wave form, and the vibrato waveform.
The files mandpluk.aiff and impuls20.aiff are suitable for the first two, and a sine wave for the last.

Performance.

kamp - Amplitude of note.

kfreq - Frequency of note played. It can be varied in performance.

kfiltq - Q of the filter, in the range 0.8 to 0.9

kfiltrate - rate control for the filter in the range 0 to 0.0002

kvibf - frequency of vibrato in Hertz. Suggested range is 0 to 12.

kvamp - amplitude of the vibrato.

---
a1 **mandol** kamp, kfreq, kpluck, kdetune, kgain, ksize, ifn[, iminfreq]

An emulation of a mandolin.

Initialisation.

ifn -- table number containing the pluck wave form. The file mandpluk.aiff is suitable for this.

iminfreq -- Lowest frequency to be played on the note. If it is omitted it is taken to be the same as the initial kfreq.

Performance.

kamp - Amplitude of note.

kfreq - Frequency of note played. It can be varied in performance.

kpluck - The pluck position, in range 0 to 1. Suggested value is 0.4

kgain - the loopgain of the model, in the range 0.97 to 1.

kdetune - The proportional detuning between the two strings. Suggested range 1 and 0.9

ksize - The size of the body of the mandolin. Range 0 to 2.

---
a1 **voice** kamp, kfreq, kphoneme, kform, kvibf, kvamp, ifn, ivfn

An emulation of a human voice.

Initialisation.

ifn, ivfn -- two table numbers containing the carrier wave form and the vibrato waveform.
The files impuls20.aiff, ahh.aiff eee.aiff or ooo.aiff are suitable for the first of these, and a sine wave for the second.

Performance.

kamp - Amplitude of note.

kfreq - Frequency of note played. It can be varied in performance.

kphoneme - an integer in the range 0 to 16, which select the formants for the sounds "eee","ihh","ehh","aaa",
"ahh","aww","ohh","uhh",
"uuu","ooo","rrr","lll",
"mmm","nnn","nng","ngg".

At present the phonemes
"fff","sss","thh","shh",
"xxx","hee","hoo","hah",
"bbb","ddd","jjj","ggg",
"vvv","zzz","thz","zhh"

are not available.

kform - Gain on the phoneme. values 0.0 to 1.2 recommended.

kvibf - frequency of vibrato in Hertz. Suggested range is 0 to 12.

kvamp - amplitude of the vibrato.

GEN25, GEN27
These subroutines are used to construct functions from segments of exponential curves (GEN25) or straight lines (GEN27) in breakpoint fashion.

f # time size 25 x1 y1 x2 y2 x3 y3 . . .
f # time size 27 x1 y1 x2 y2 x3 y3 . . .

size - number of points in the table. Must be a power of 2 or power-of-2
plus 1 (see f statement).

x1, x2, x3, etc. - locations in table at which to attain the following y value. Must be in increasing order. If the last value is less than size, then the rest will be set to zero. Should not be negative but can be zero.

y1, y2, y3, etc. - Breakpoint values attained at the location specified by the preceding x value. For GEN25 these must be non-zero and must be alike in sign. No such restrictions exist for GEN27.

Note: If p4 is positive, functions are post-normalized (rescaled to a maximum absolute value of 1 after generation). A negative p4 will cause rescaling to be skipped.

Example: f 1 0 257 27 0 0 100 1 200 -1 256 0

This describes a function which begins at 0, rises to 1 at the 100th table location, falls to -1, by the 200th location, and returns to 0 by the end of the table. The interpolation is linear.

f 1 0 257 25 0 0.001 100 1 200 .001 256 0.001

Similar to above, but creates exponential curve. No values <= 0 are allowed in Gen25.

Windows GUI Changes
==========

The treatment of graphs has changed significantly. In particular there are two new menu items, to display the previous or next graph.
These work during the run, and also at the end. It can remember up to
40 graphs.

Instead of a dialog asking for a click at the end, the system waits for any character. During this time the menus can be used, so the graphs can be redisplayed, or the output scrolled back. Also the text buffer size has been increased.

Also for Windows, the output form devaudio0, devaudio1,.. or adc0, adc1,... can be used to select which of many audio devices are to be used. (devaudio1 would refer to device 1 etc). The form devaudio refers to device zero.

==John ff
1998 Jan 2

Release Notes for 3.48
==========

These are the release notes for version 3.48, which is mainly a bug-fix release. These notes should be read in conjunction with earlier

release notes. The main non-bug-fix material is described in the language changes.

Language Changes

The input or output file in -i and -o can start with a | to indicate a process which is started to create or process audio files. This works on Windows and Unix, but not (yet) DOS.

WAV format now supports floating samples correctly. (Richard Dobson)

Macros in orchestra and scores [Notes 1 and 2]

Repeat sections in Scores [Note 1]

#include available in orchestra and score [Notes 1 and 2]

/* */ comments allowed in orchestra and score

Removal of limit on orchestra size (ARGSPACE and ORTEXT problems)

The opcodes asin, acos and atan renamed as sininv, cosinv and taninv so as to avoid name-space polution.

_ allowed as part of a word. Words case significant.

Limited evaluation of expressions in the score [Note 6]

Opcode Fixes
---

reverb2 was tuned to one particular sampling rate. Replaced by nreverb opcode. (Richard Karpen)

wgbow, flute and brass fixed in various ways

Fixes in fof2 and fog (Ekman)

shaker used to ignore krequency; now fixed

New Opcodes
---
pvadd (Richard Karpen) [Note 3]

taninv2 in kk and aa contexts (equivalent to atan2 in C)

printk2 which prints when a k-value changes (Gabriel Maldonado) [Note 0]

locsig, **locsend**, **space**, **spsend**, **spdist** (Richard Karpen) for locating sound [Notes 7 and 8]

New GEN function 28 to read x,y values direct from a file.

Other Changes:

On Windows permissions of output files could be wrong.

Problem on SUN fixed, which gave silence sometimes and other errors.

emacs modes have new opcodes added, and some support for macros.

Pipes allowed in -L inputs, using a | as the first character of the 'filename'.

-H2 and -H3 options to display heartbeat in a different way. -H1 is equivalent to -H and -H0 is equivalent to no -H option. [Note 5]

Windows GUI Changes

Output device selectable by menu (Richard Dobson). No equivalent code for input yet.

Reading MIDI files fixed in interface.

xyin implemented in windows. [Note 4]

Heartbeat option on Extras dialog allows numbers now. [Note 5]

==John ff
1998 Apr 14
==========
Note 0:
printk2

printk2 kvar [, numspaces]

INITIALIZATION

numspaces - number of space characters printed before the value of kvar

PERFORMANCE

kvar - signal to be printed

Derived from Robin Whittle's printk, prints a new value of kvar each time kvar changes. Useful for monitoring MIDI control changes when
using sliders. Warning! don't use this opcode with normal, continuously variant k-signals, because it can hang the computer, as the rate of printing is too fast.

Note 1:

Changes to the Score Language

John ffitch April 1998

The 3.48 version of Csound introduces a number of changes in the language in which scores are presented to the system. These are all upward compatible, and so do not require any changes in existing scores. These changes should allow for simpler score writing, and provide an elementary alternative to the full score-generation systems. Similar changes have been made in the orchestra language.

Simple Macros

Macros are textual replacements which are made in the score as it is being read. The macro system in Csound is a very simple one, and uses two special characters to indicate the presence of macros, the characters # and $.

To define a macro one uses the # character.

#define NAME # replacement text#

The name of the macro can be any made from letters, upper or lower case. Digits are not allowed. The replacement text is any character string (not containing a #) and can extend over more than one line. The replacement text is enclosed within the # characters, which ensures that additional characters are not inadvertently captured.

To use a macro the name is used following a $ character. The name is
terminated by the next non-letter. If the need is to have the name without a space a period can be used to terminate the name, which is ignored. The string $NAME. is replaced by the replacement text from
the definition. Of course the replacement text can also include macro calls.

If a macro is not required any longer it can be undefined with

#undef NAME

Example:
If a note-event has a set of p-fields which are repeated

```
#define ARGS # 1.01 2.33 138#
i1 0 1 8.001000 $ARGS
i1 0 1 8.011500 $ARGS
i1 0 1 8.021200 $ARGS
i1 0 1 8.031000 $ARGS
```

This will get expanded before sorting into

```
i1 0 1 8.001000 1.01 2.33 138
i1 0 1 8.011500 1.01 2.33 138
i1 0 1 8.021200 1.01 2.33 138
i1 0 1 8.031000 1.01 2.33 138
```

This can save typing and is easier to change if for example one needed to change one of the parameters. If there were two sets of p-fields one could have a second macro (there is no real limit on the number of macros one can define).

```
#define ARGS1 # 1.01 2.33 138#
#define ARGS2 # 1.41 10.33 1.00#
i1 0 1 8.001000 $ARGS1
i1 0 1 8.011500 $ARGS2
i1 0 1 8.021200 $ARGS1
i1 0 1 8.031000 $ARGS2
```

An alternative would be to use the second form of the macro, described below.

Note: some care is needed with textual macros as they can sometimes do strange things. They take no notice of any meaning, and so spaces are significant, which is why the definition has the replacement text surrounded by # characters, unlike that in the C programming language.
Used carefully simply macros are a powerful concept, but they can be abused.

Advanced Macros

Macros can also be defined with parameters. This can be used in more complex situations. In order to define a macro with arguments the syntax is

#define NAME(A#B#C) #replacement text#

Within the replacement text the arguments can be substituted by the form $A. In fact the implementation defines the arguments as simple macros. There may be up to 5 arguments, and the names can be any choice of letters. Case is significant in macro names.

In use the argument form for example

```
#define ARG(A) # 2.345  1.03  $A  234.9#
i1 0 1 8.00 1000 $ARG(2.0)
i1  + 1 8.01 1200 $ARG(3.0)
```

which expands to

```
i1 0  1 8.00 1000 2.345  1.03  2.0  234.9
i1 + 1 8.01 1200 2.345  1.03  3.0  234.9
```

As with the simple macros, these macros can also be undefined with

#undef NAME

Another Use For Macros

When writing a complex score it is sometimes all too easy to forget to what the various instrument numbers refer. One can use macros to give names to the numbers. For example

```
#define Flute       #i1#
#define Whoop       #i2#

$Flute.   0       10       4000      440
$Whoop.   5       1
```

Multiple File Score

It is sometimes convenient to have the score in more than one file. This use is supported by the #include facility which is part of the macro system. A line containing the text

#include :filename:

where the character : can be replaced by any suitable character. For most uses the double quote symbol will probably be the most convenient.

This takes input from the named file until it ends, when input reverts to the previous input. There is currently a limit of 20 on the depth of included files and macros.

A suggested use of #include would be to define a set of macros which
are part of the composer's style. It could also be used to provide repeated sections.

```
s
#include :section1:
;; Repeat that
s
#include :section1:
```

However there is an alternative way of doing repeats, described below.

Repeated Sections

Sections can be repeated by using #include or by editing the text. An alternative is the new r directive in the score language.

r3 NN

starts a repeated section, which lasts until the next s, r or e directive. The section is repeated 3 times in this example. In order that the sections may be more flexible than simple editing, the macro NN is given the value of 1 for the first time through the section, 2 for the second, and 3 for the third. This can be used to change p-field parameters, or indeed ignored.

Warning: because of serious problems of interaction with macro expansion, sections must start and end in the same file, and not in a macro.

Evaluation of Expressions

In earlier versions of Csound the numbers presented in a score were used as given. There are occasions when some simple evaluation would be easier. This need is increased when there are macros. To assist in this area the syntax of an arithmetic expressions within square brackets [ ] has been introduced. Expressions built from the operations +, -, *, and / are allowed, together with grouping with ( ). The expressions can include numbers, and naturally macros whose values are numeric or arithmetic strings. All calculations are made in floating point numbers. Note that unary minus is not yet supported.

Example:

```
r3      CNT

i1      0       [0.3*$CNT.]
i1      +       [($CNT./3)+0.2]

e
```

As the three copies of the section have the macro $CNT. with the different values of 1, 2 and 3, this expands to

```
s
i1      0       0.3
i1      0.3     0.533333
s
i1      0       0.6
i1      0.6     0.866667
s
i1      0       0.9
i1      0.9     1.2
e
```

This is an extreme form, but the evaluation system can be used to ensure that repeated sections are subtly different.

Note 2:
Changes to the Orchestra Language

John ffitch April 1998

In version 3.48 a macro and multiple file system has been incorporated into the orchestra language. This is similar to the macro system in the score language, but is independent.

Simple Macros

Macros are textual replacements which are made in the orchestra as it is being read. The macro system in Csound is a very simple one, and uses two special characters to indicate the presence of macros, the characters # and $.

To define a macro one uses the # character.

#define NAME # replacement text#

The name of the macro can be any made from letters, upper or lower case. Digits are not allowed. The replacement text is any character string (not containing a #) and can extend over more than one line.
The replacement text is enclosed within the # characters, which ensures that additional characters are not inadvertently captured.

To use a macro the name is used following a $ character. The name is terminated by the next non-letter. If the need is to have the name without a space a period can be used to terminate the name, which is ignored. The string $NAME. is replaced by the replacement text from the definition. Of course the replacement text can also include macro calls.

If a macro is not required any longer it can be undefined with

#undef NAME

Example:

```
#define REVERB #ga = ga+a1
            out a1#

instr 1
 a1                 oscil
 $REVERB.
endin

instr 2
 a1                 repluck
 $REVERB.
endin
```

This will get expanded before compilation into

```
instr 1
 a1                 oscil
 ga = ga+a1
            out a1
endin

instr 2
 a1                 repluck
 ga = ga+a1
            out a1
endin
```

This can save typing, and in the case, for example, of a general effects processing sequence, it can lead to a coherent and consistent use.

This form is limiting in at least having the variable names fixed. An alternative would be to use the second form of the macro, described below.

Note: some care is needed with textual macros as they can sometimes do strange things. They take no notice of any meaning, and so spaces are significant, which is why the definition has the replacement text surrounded by # characters, unlike that in the C programming language.
Used carefully simply macros are a powerful concept, but they can be abused.

Advanced Macros

Macros can also be defined with parameters. This can be used in more complex situations. In order to define a macro with arguments the syntax is

#define NAME(A#B#C) #replacement text#

Within the replacement text the arguments can be substituted by the form $A. In fact the implementation defines the arguments as simple macros. There may be up to 5 arguments, and the names can be any choice of letters. Case is significant in macro names.

In use the argument form for example

```
#define REVERB(A) #ga = ga+$A.
            out $A.#
instr 1
 a1                 oscil
 $REVERB(a1)
endin

instr 2
 a2                 repluck
 $REVERB(a2)
endin
```

to which expands

```
instr 1
 a1                 oscil
 ga = ga+a1
            out a1
endin

instr 2
 a2                 repluck
 ga = ga+a2
            out a2
endin
```

As with the simple macros, these macros can also be undefined with

#undef NAME

Multiple File Orchestras

It is sometimes convenient to have the orchestra arranged in a number of files, for example with each instrument in a separate file. This style is supported by the #include facility which is part of the macro system. A line containing the text

#include :filename:

where the character : can be replaced by any suitable character. For most uses the double quote symbol will probably be the most convenient.

This takes input from the named file until it ends, when input reverts to the previous input. There is currently a limit of 20 on the depth of included files and macros.

Another suggested use of #include would be to define a set of macros which are part of the composer's style.

An extreme form would be to have each instrument defines as a macro, with the instrument number as a parameter. Then an entire orchestra could be constructed from a number of #include statements followed by macro calls.

#include :clarinet:
#include :flute:
#include :bassoon:
$CLARINET(1)
$FLUTE(2)
$BASSOON(3)

It must be stressed that these changes are at the textual level and so take no cognisance of any meaning.

Note 3:
pvadd
Created by Richard Karpen, 1998

a2 **pvadd** ktimpnt, kfmod, ifile, ifn, ibins [, ibinoffset, ibinincr]

DESCRIPTION

pvadd reads from a pvoc file and uses the data to perform additive synthesis using an internal array of interpolating oscillators. The user supplies the wave table (usually one period of a sine wave), and can choose which analysis bins will be used in the re-synthesis.

PERFORMANCE

ktimpnt, kfmod, and ifile are used in the same way as in pvoc.

ifn is the table number of a stored function containing a sine wave.

ibins is the number of bins that will be used in the resynthesis (each bin counts as one oscillator in the re-synthesis).

ibinoffset is the first bin used (it is optional and defaults to 0).

ibinincr sets an increment by which pvadd counts up from ibinoffset for ibins components in the re-synthesis (see below for a further explanation).

EXAMPLE:

```
ktime  line   0, p3, p3
asig   pvadd  ktime, 1, "oboe.pvoc", 1, 100, 2
```

In the above, ibins is 100 and ibinoffset is 2. Using these settings the resynthesis will contain 100 components beginning with bin #2 (bins are counted starting with 0). That is, resynthesis will be done using bins 2-101 inclusive. It is usually a good idea to begin with bin 1 or 2 since the 0th and often 1st bin have data that is neither necessary nor even helpful for creating good clean resynthesis.

```
ktime  line   0, p3, p3
asig   pvadd  ktime, 1, "oboe.pvoc", 1, 100, 2, 2
```

The above is the same as the previous example with the addition of thevalue 2 used for the optional ibinincr argument. This result will still result in 100 components in the resynthesis, but pvadd will count through the bins by 2 instead of by 1. It will use bins 2, 4, 6, 8, 10, and so on. For ibins=10, ibinoffset=10, and ibinincr=10, pvadd would use bins 10, 20, 30, 40, up to and including 100.

USEFUL HINTS:
By using several pvadd units together, one can gradually fade in different parts of the resynthesis, creating various "filtering" effects. The author uses pvadd to synthesis one bin at a time to have control over each separate component of the re-synthesis.

If any combination of ibins, ibinoffset, and ibinincr, creates a situation where pvadd is asked to used a bin number greater than the number of bins in the analysis, it will just use all of the available bins and give no complain. So to use every bin just make ibins a big number (ie. 2000).

Expect to have to scale up the amplitudes by factors of 10-100 by the way.

Note 4:

When xyin is called the position of the mouse within the output window is used to reply to the request. This simple mechanism does mean that only one xyin can be used accurately at once. The position of the mouse is reported in the output window

---

Note 5:

-H1 generates a 'rotating line' progress report.
-H2 generates a . everytime a buffer is written.
-H3 reports the size in seconds of the output.
-H4 sounds a bell for every buffer of the output written.

Note 6:

Expressions enclosed in square brackets [ ] are evaluated at read-time for scores. This allows 4-function arithmetic and brackets (no unary minus yet) on numbers and macros whose values are numbers. It can be used with repeats to change timing or dynamics.

Note 7:

```
a1, a2              locsig     asig, kdegree, kdistance, kreverbsend
a1, a2, a3, a4      locsig     asig, kdegree, kdistance, kreverbsend
a1, a2              locsend
a1, a2, a3, a4      locsend
```

DESCRIPTION

locsig takes an input signal and distributes it among 2 or 4 channels using values in degrees to calculate the balance between adjacent channels. It also takes arguments for distance (used to attenuate signals that are to sound as if they are some distance further than the loudspeaker itself), and for the amount the signal that will be sent to reverberators. This unit is based upon the example in the Charles Dodge/Thomas Jerse book, "Computer Music," page 320.

locsend depends upon the existence of a previously defined locsig. The number of output signals must match the number in the previous locsig. The output signals from locsend are derived from the values given for distance and reverb in the locsig and are ready to be sent to local or global reverb units (see example below). The reverb amount and the balance between the 2 or 4 channels are calculated in the same way as described in the Dodge book (an essential text!).

PERFORMANCE

kdegree - value between 0 and 360 for placement of the signal in a 2 or 4 channel space configured as: a1=0, a2=90, a3=180, a4=270 (kdegree=45 would balanced the signal equally between a1 and a2).

locsig maps kdegree to sin and cos functions to derive the signal balances (ie.: asig=1, kdegree=45, a1=a2=.707).

kdistance - value >= 1 used to attenuate the signal and to calculate reverb level to simulate distance cues. As kdistance gets larger the sound should get softer and somewhat more reverberant (assuming the use of locsend in this case).

kreverbsend - the percentage of the direct signal that will be factored along with the distance and degree values to derive signal amounts that can be sent to a reverb unit such as reverb, or reverb2.

EXAMPLE:

```
asig some audio signal
kdegree line 0, p3, 360
kdistance line 1, p3, 10
a1, a2, a3, a4 locsig asig, kdegree, kdistance, .1
ar1, ar2, ar3, ar4 locsend

ga1 = ga1+ar1
ga2 = ga2+ar2
ga3 = ga3+ar3
ga4 = ga4+ar4

outq a1, a2, a3, a4
endin

instr 99 ; reverb instrument

a1 reverb2 ga1, 2.5, .5
a2 reverb2 ga2, 2.5, .5
a3 reverb2 ga3, 2.5, .5
a4 reverb2 ga4, 2.5, .5

outq a1, a2, a3, a4
ga1=0
ga2=0
ga3=0
ga4=0
```

In the above example, the signal, asig, is sent around a complete circle once during the duration of a note while at the same time it becomes more and more "distant" from the listeners' location. Locsig sends the appropriate amount of the signal internally to locsend. The outputs of the locsend are added to global accumulators in a common Csound style and the global signals are used as inputs to the reverb units in a separate instrument.

locsig is useful for quad and stereo panning as well as fixed placed of sounds anywhere between two loudspeakers. Below is an example of the fixed placement of sounds in a stereo field.

```
instr 1
...
a1, a2 locsig asig, p4, p5, .1
ar1, ar2 locsend

ga1=ga1+ar1
ga2=ga2+ar2
outs a1, a2
endin

instr 99 ; reverb....
....
endin

A few notes
;place the sound in the left speaker and near
i1 0 1 0 1
;place the sound in the right speaker and far
i1 1 1 90 25
;place the sound equally between left and right and in the middle ground distance
i1 2 1 45 12
e
```

The next example shows a simple intuitive use of the distance value to simulate doppler shift. The same value is used to scale the frequency as is used as the distance input to locsig.

```
kdistance line 1, p3, 10
kfreq = (ifreq * 340) / (340 + kdistance)
asig oscili iamp, kfreq, 1
kdegree line 0, p3, 360

a1, a2, a3, a4 locsig asig, kdegree, kdistance, .1
ar1, ar2, ar3, ar4 locsend
```

Note 8:

a1, a2, a3, a4 **space** asig, ifn, ktime, kreverbsend [,kx, ky]
a1, a2, a3, a4 **spsend**
k1 **spdist** ifn, ktime, [,kx, ky]

DESCRIPTION

space takes an input signal and distributes it among 4 channels using cartesian xy coordinates to calculate the balance of the outputs. The xy coordinates can be defined in a separate text file and accessed through a Function statement in the score using Gen28 (description of Gen28 given below), or they can be specified using the optional kx, ky arguments. There advantages to the former are: 1. A graphic user interface can be used to draw and edit the trajectory through the cartesian plane; 2. The file format is in the form time1 X1 Y1 time2 X2 Y2 time3 X3 Y3 allowing the user to define a time-tagged trajectory. space then allows the user to specify a time pointer (much as is used for pvoc, lpread and some other units) to have detailed control over the final speed of movement.

spsend depends upon the existence of a previously defined space. The output signals from spsend are derived from the values given for XY and reverb in the space and are ready to be sent to local or global reverb units (see example below).

spdist uses the same xy data as space, also either from a text file using Gen28 or from x and y arguments given to the unit directly. The purpose of this unit is to make available the values for distance that are calculated from the xy coordinates. In the case of space the xy values are used to determine a distance which is used to attenuate the signal and prepare it for use in spsend. But it is also useful to have these values for distance available to scale the frequency of the signal before it is sent to the space unit.

PERFORMANCE

The configuration of the XY coordinates in space places the signal in the following way: a1 is -1, 1; a2 is 1, 1; a3 is -1, -1; a4 is 1, -1. This assumes a loudspeaker set up as a1 is left front, a2 is right front, a3 is left back, a4 is right back. Values greater than 1 will result in sounds being attenuated as if in the distance. Space considers the speakers to be at a distance of 1; smaller values of XY can be used, but space will not amplify the signal in this case. It will, however balance the signal so that it can sound as if it were within the 4 speaker space. x=0, y=1, will place the signal equally balanced between left and right front channels, x=y=0 will place the signal equally in all 4 channels, and so on. Although there must be 4 output signal from space, it can be used in a 2 channel orchestra. If the XY's are kept so that Y>=1, it should work well to do panning and fixed localization in a stereo field.

ifn - number of the stored function created using Gen28. This function generator reads a text file which contains sets of three values representing the xy coordinates and a time-tag for when the signal should be placed at that location. The file should look like:

0	-1	1
1	1	1
2	4	4
2.1	-4	-4
3	10	-10
5	-40	0

If that file were named "move" then the Gen28 call in the score would like:

```
f1 0 0 "move"
```

Gen28 takes 0 as the size and automatically allocates memory. It creates values to 10 milliseconds of resolution. So in this case there will be 500 values created by interpolating X1 to X2 to X3 and so on, and Y1 to Y2 to Y3 and so on, over the appropriate number of values that are stored in the function table. In the above example, the sound will begin in the left front, over 1 second it will move to the right front, over another second it move further into the distance but still in the left front, then in just 1/10th of a second it moves to the left rear, a bit distant. Finally over the last .9 seconds the sound will move to the right rear, moderately distant, and it comes to rest between the two left channels (due west!), quite distant. Since the values in the table are accessed through the use of a time-ponter in the space unit, the actual timing can be made to follow the file's timing exactly or it can be made to go faster or slower through the same trajectory. If you have access to the GUI that allows one to draw and edit the files, there is no need to create the text files manually. But as long as the file is ASCII and in the format shown above, it doesn't matter how it is made! IMPORTANT: If ifn is 0 then space will take its values for the xy coordinates from kx and ky.

ktime - index into the table containing the xy coordinates. If used like:

```
ktime line 0, 5, 5
a1, a2, a3, a4 space asig, 1, ktime, ...
```

with the file "move" described above, the speed of the signal's movement will be exactly as desribed in that file. However:

```
ktime line 0, 10, 5
```

the signal will move at half the speed specified. Or in the case of:

```
ktime line 5, 15, 0
```

the signal will move in the reverse direction as specified and 3 times slower! Finally:

```
ktime line 2, 10, 3
```

will cause the signal to move only from the place specified in line 3 of the text file to the place specified in line 5 of the text file, and it will take 10 seconds to do it.

kreverbsend - the percentage of the direct signal that will be factored along with the distance as derived from the XY coordinates to calculate signal amounts that can be sent to reverb units such as reverb, or reverb2.

kx, ky - when ifn is 0, space and spdist will use these values as the XY coordinates to localize the signal. They are optional and both default to 0.

EXAMPLE:

```
asig some audio signal
ktime line 0, p3, p10
a1, a2, a3, a4 space asig,1, ktime, .1
ar1, ar2, ar3, ar4 spsend

ga1 = ga1+ar1
ga2 = ga2+ar2
ga3 = ga3+ar3
ga4 = ga4+ar4

outq a1, a2, a3, a4
endin

instr 99 ; reverb instrument

a1 reverb2 ga1, 2.5, .5
a2 reverb2 ga2, 2.5, .5
a3 reverb2 ga3, 2.5, .5
a4 reverb2 ga4, 2.5, .5

outq a1, a2, a3, a4
ga1=0
ga2=0
ga3=0
ga4=0
```

In the above example, the signal, asig, is moved according to the data in Function #1 indexed by ktime. space sends the appropriate amount of the signal internally to spsend. The outputs of the spsend are added to global accumulators in a common Csound style and the global signals are used as inputs to the reverb units in a separate instrument.

space can useful for quad and stereo panning as well as fixed placed of sounds anywhere between two loudspeakers. Below is an example of the fixed placement of sounds in a stereo field using XY values from the score instead of a function table.

```
instr 1
...
a1, a2, a3, a4 space asig, 0, 0, .1, p4, p5
ar1, ar2, ar3, ar4 spsend

ga1=ga1+ar1
ga2=ga2+ar2
outs a1, a2
endin

instr 99 ; reverb....
....
endin

A few notes: p4 and p5 are the X and Y values
;place the sound in the left speaker and near
i1 0 1 -1 1
;place the sound in the right speaker and far
i1 1 1 45 45
;place the sound equally between left and right and in the middle
ground distance
i1 2 1 0 12
e
```

The next example shows a simple intuitive use of the distance values returned by spdist to simulate doppler shift.

```
ktime line 0, p3, 10
kdist spdist 1, ktime
kfreq = (ifreq * 340) / (340 + kdist)
asig oscili iamp, kfreq, 1

a1, a2, a3, a4 space asig, 1, ktime, .1
ar1, ar2, ar3, ar4 spsend
```

The same function and time values are used for both spdist and space. This insures that the distance values used internally in the space unit will be the same as those returned by spdist to give the impression of a doppler shift!

Release Notes for 3.49

==========

These are the release notes for version 3.49, which is a large collection of bug-fixes and new code. These notes should be read in conjunction with earlier release notes. Note that this incorporates all changes since 3.48, including sub-releases.

Language Changes

-J option selects IRCAM format in the same was as -W and -A

Improved diagnostics in orchestra reading

b opcode in score to reset the clock

Increase number of arguments to about 800 (still not dynamic)

Improved recognition of # at start of line

Stop redrawing of graphs in some circumstances

strset now works, and unlimited in number; can ne used in pv, lpc, adsyn amd convolve cases as well.

Removed a large number of 'namespace polution' opcodes to other names

eg itable is now table
kgauss is now gauss
ktableseg and ktablexseg renamed as tableseg and tablexseg

Use of large instrument numbers now correct

Corrected use of [ ] in scores

Freed space problem in GEN20

Digits allowed in macros names except at start

-z option does not report internal opcodes

AIFC supported at least for floats

Included files in orchestra us a pathname look-up

v opcode in scores for local textual varying of time

Allow Mac, Unix or PC files to be read on other platforms

Ouput file null is thrown away (ie no sound file generated)

MIDI control message PROGRAM_TYPE recognised

Rewrite us of \ as line continuation in orchestra

New ramp functions in score introduced by { and } give ramps driven by
expon rather than line

The ramp function ~ gives a random value (uniform distribution) in range on the ramp

Opcode Fixes

Internal bug in cross2 fixed which could confuse a second note

wgflute improved so as not to reinitialise so much

diskin and soundin fixed a little

aftertouch had wrong arguments

shaker has argument removed which was not used

Skip initialisation in physical model instruments if lowest frequency is negative (for legato sounds)

Arguments to specptrk and specdisp now agree with manual

envlpr code included -- omitted by mistake earlier

New Opcodes
---

dcblockr -- DC Blocking filter

flanger -- as it says

lowres, lowresx and vlowres -- lowpass resonent filters

tonex atonex resonx -- more multiple filters

spectrum -- calculate w variables

mirror, wrap -- actions on large amplitudes

ntrpol -- interpolation

trigger -- trigger events

ftsr -- sample rate of a f-table

wguide1, wguide2 -- primitive wave guides

GEN23 -- read a table of numbers

adsr and madsr -- classical ADSR envelope

biquad -- a new filter

moogvcf -- another one

rezzy -- and another

Other Changes:

Solaris audio corrected

Bug in line events for score fixed

voscili opcode removed as did not work well and the functionality exists elsewhere

Scot removed

Windows GUI Changes

Made buffer sizes in extras window independednt and remembered

Stop redrawing of graphs

OK button renamed Render

Remove references to Pedal

Added project button to set orc/sco/wav in one go

Experimental control opcodes with non-MIDI sliders

==John ff
1998 Oct 18
==========

dcblockr

aout dcblockr ain[, igain]

INITIALISATION

igain -- the gain of teh filter, which defaults to 0.99

PERFORMANCE

Implements the DC blocking filter
Y[i] = X[i] - X[i-1] + (igain * Y[i=1])
This is due to P.Cook, and coded by JPff
---

flanger

ar **flanger** asig, adel, kfeedback, imaxd

DESCRIPTION

a user controlled flanger

INITALIZATION

imaxd - maximum delay in seconds (needed for inital memory allocation)

PERFORMANCE

ar - output signal
asig - input signal
adel - delay in seconds
kfeedback - feedback amount (in normal tasks this should not exceed 1,
even if bigger values are allowed)

This unit is useful for generating chourses and flangers. The delay must be varied at a-rate connecting adel to an oscillator output. Also the feedback can vary at k-rate. This opcode is implemented to allow kr different than sr (else delay could not be lower than ksmps) enhancing realtime performance. (BtW: this unit is very similar to wguide1, the only difference is flanger does not have the lowpass filter.).

---

lowres, lowresx

ar **lowres** asig, kcutoff, kresonance [,istor]
ar **lowresx** asig, kcutoff, kresonance [, inumlayer, istor]

DESCRIPTION

lowres is a resonant lowpass filter.
lowresx is equivalent to more layer of lowres, with the same arguments, serially connected.

INITIALIZATION

inumlayer - number of elements of lowresx stack. Default value is 4. There is no maximum.

istor - initial disposition of internal data space.

A zero value will clear the space; a non-zero value will allow previous information to remain. The default value is 0.

PERFORMANCE

ar - output signal
asig - input signal
kcutoff - filter cutoff frequency point
kresonance - resonance amount

lowres is a resonant lowpass filter derived from a Hans Mikelsons orchestra. This implementation is very much faster than implementing it in Csound language, and it allows kr lower than sr. kcutoff is not in cps and kresonance is not in dB, so experiment for finding best results.

lowresx is equivalent to more layer of lowres, with the same arguments, serially connected. Using a stack of more filters allows a sharper freqency cutoff. It is very faster than using more lowres instances in Csound orchestra, because only one initialization and 'k' cycle are needed at time, and the audio loop falls enterely inside the cache memory of processor.

---

vlowres

ar **vlowres** asig, kfco, kres, iord, ksep;

DESCRIPTION

a bank of filters in which freqency cutoff can be separated under user control

INITIALIZATION

iord - total nuber of filter (1 to 10)

PERFORMANCE

ar - output signal
asig - input signal
kfco - frequency cutoff (not in cps)
ksep - frequency cutoff separation for each filter

vlowres (variable resonant lowpass filter) allow a variable response curve in resonant filters. It can be thinked as a bank of lowpass resonant filters with the same resonance, serially connected. The frequency cutoff of each filter can vary with the kcutoff and ksep parameters.

---

tonex atonex resonx

ar **tonex** asig, khp[, inumlayer, istor]
ar **atonex** asig, khp[, inumalayer, istor]
ar **resonx** asig, kcf, kbw[, inumlayer, iscl, istor]

INITALIZATION

inumlayer - number of elements of filter stack. Default value is 4.

isig - some as tone, atone and reson
istor - some as tone, atone and reson
iscl - some as reson

PERFORMANCE

ar - output signal
asig - input signal
khp - some as tone, atone
kcf - some as reson
kbw - some as reson

tonex, atonex and resonx are equivalent to more layer of tone, atone and reson, with the same arguments, serially connected. Using a stack of more filters allows a sharper frequency cutoff. They are very faster than using more instances in Csound orchestra of old opcodes, because only one initialization and 'k' cycle are needed at time, and the audio loop falls enterely inside the cache memory of processor.

---

spectrum

wsig **spectrum** xsig, iprd, iocts, ifrqs, iq[,ihann, idbout, idsprd, idsinrs]

Generate a constant-Q, exponentially-spaced DFT across al octaves of a multiply-downsampled control or audio input signal.
INITIALIZATION

ihann (optional) - apply a hamming or hanning window to the input. The default is 0 (hamming window)

idbout (optional) - coded conversion of the DFT output: 0 = magnitude, 1 = dB, 2 = mag squared, 3 = root magnitude. The default value is 0 (magnitude).

idisprd (optional) - if non-zero, display the composite downsampling buffer every idisprd seconds. The default value is 0 (no display).

idsines (optional) - if non-zero, display the hamming or hanning windowed sinusoids used in DFT filtering. The default value is 0 (no

sinusoid display).

PERFORMANCE

This unit first puts signal asig or ksig through iocts of successive octave decimation and downsampling, and preserves a buffer of down-sampled values in each octave (optionally displayed as a composite buffer every idisprd seconds). Then at every iprd seconds, the preserved samples are passed through a filter bank (ifrqs parallel filters per octave, exponentially spaced, with frequency/bandwidth Q of iq), and the output magnitudes optionally converted (idbout) to produce a band-limited spectrum that can be read by other units.

The stages in this process are computationally intensive, and computation time varies directly with iocts, ifrqs, iq, and inversely with iprd. Settings of ifrqs = 12, iq = 10, idbout = 3, and iprd = .02 will normally be adequate, but experimentation is encouraged.
ifrqs currently has a maximum of 120 divisions per octave. For audio input, the frequency bins are tuned to coincide with A440.

This unit produces a self-defining spectral datablock wsig, whose characteristics used (iprd, iocts, ifrqs, idbout) are passed via the data block itself to all derivative wsigs. There can be any number of spectrum units in an instrument or orchestra, but all wsig names must be unique.

Example:

```
asig  in                                    ; get external audio
wsig  spectrum  asig,.02,6,12,33,0,1,1      ; downsample in 6 octs
                                            ;& calc a 72 pt
                                  ; dft (Q 33, dB out) every 20 msecs
```

---

mirror, wrap --

idest **wrap** isig, ilow, ihigh
kdest **wrap** ksig, klow, khigh
adest **wrap** asig, klow, khigh

idest **mirror** isig, ilow, ihigh
kdest **mirror** ksig, klow, khigh
adest **mirror** asig, klow, khigh

DESCRIPTION
Wraps the signal in various ways (similar to limit opcode by Robin Whittle).

INITIALIZATION - PERFORMANCE

xdest - output signal
xsig - input ignal
xlow - low threshold
xhigh - high threshold

mirror "reflects" the signal that exceeds low and high thresholds.
wrap wraps-around the signal that exceeds low and high thresholds.

These opcodes are useful in several situations, such as for table indexing and for clipping and modeling irate, krate or arate signals.
wrap is also useful for wrapping-around tables data when maximum index is not a power of two (see table and tablei).
Another use of wrap is in cyclical event repeating with arbirary cycle length.

---

ntrpol

ir **ntrpol** isig1, isig2, ipoint [, imin, imax]
kr **ntrpol** ksig1, ksig2, kpoint [, imin, imax]
ar **ntrpol** asig1, asig2, kpoint [, imin, imax]

DESCRIPTION

calculates the weighted mean value (i.e. linear interpolation) of two input signals

INITALIZATION
imin - minimum xpoint value (optional, default 0)
imax - maximum xpoint value (optional, default 1)

PERFORMANCE
xr - output signal
xsig1, xsig2 - input signals
xpoint - interpolation point beetween the two values

nterpol opcode outputs the linear interpolation beetween two input values. xpoint is the distance of evaluation point from the first value. With the default values of imin and imax, (0 and 1) a zero value indicates no distance from the first value and the maximum distance from the second one. With a 0.5 ntrpol value will output the mean value of the two inputs, indicating the exact half point between xsig1 and xsig2. A 1 value indicates the maximum distance from the first value and no distance from the second one.
The range of xpoint can be also defined with imin and imax to make easier its management.
These opcodes are useful for crossfading two signals.

---

trigger

kout **trigger** ksig, kthreshold, kmode

DESCRIPTION

informs when a krate signal crosses a threshold

PERFORMANCE
kout - output signal (a stream of zeroes with some 1)
ksig - input signal
kthreshold - trigger threshold
kmode - can be 0 , 1 or 2

Normally trigger outputs zeroes: only each time ksig crosses kthreshold 'trig' outputs a 1. There are three modes of using ktrig:
kmode = 0 - (down-up) ktrig outputs a 1 when current value of ksig is higher than kthreshold while old value of ksig was equal or lower than kthreshold
kmode = 1 - (up-down) ktrig outputs a 1 when current value of ksig is lower than kthreshold while old value of ksig was equal or higher than kthreshold
kmode = 2 - (both) ktrig outputs a 1 in both the two previous cases.

---

ftsr(x)

DESCRIPTION

this function returns the sampling-rate of a GEN01 or GEN22 generated table. The sampling-rate is determined from the header of the original file. If the original file has no header, or the table was not created by these two GENs ftsr returns 0.

---

wguide1, wguide2

DESCRIPTION

simple waveguide blocks

ar **wguide1** asig, kfreq, kcutoff, kfeedback;
ar **wguide2** asig, kfreq1, kfreq2, kcutoff1, kcutoff2, kfeedback1, kfeedback2

PERFORMANCE

wguide1 is the most elemental waveguide model consisting of one delay line and one first-order lowpass filter.
wguide2 is a model of beaten plate consisting of two parallel delay lines and two first-order lowpass filters. The two feedabak lines are mixed and sent to the delay again each cycle.

asig is the input of excitation noise, kfreq the frequency (i.e. the inverse of delay time), kcutoff is the filter cutoff frequency in Hz and kfeedback is the feedback factor.
Implementing waveguide algoritms as opcodes, instead of as orc instr, allows the user to set kr different than sr, allowing better performance particulary when using real-time.

GEN23

This subroutine reads numeric values from an external ascii file

f# time size -23 "filename.txt"

The numeric values contained in "filename.txt" (which indicates the complete pathname of the character file to be read), can be separated by spaces, tabs, newline characters or commas. Also words that contains non-numeric characters can be used as comments since they are ignored.

All characters following ';' (comment) are ignored until next line (numbers too).

---

adsr, madsr

kr **adsr** iatt, idec, islev, irel[, idelay]
ar **adsr** iatt, idec, islev, irel[, idelay]

DESCRIPTION

Calculates the classical ADSR envelope

INITALIZATION

iatt - duration of attack phase
idec - duration of decay
islev - level for sustain phase
irel - duration of release phase
idel - period of zero before the envelope starts

PERFORMANCE
The envelope is the range 0 to 1 and may need to be scaled further.

The length of the sustain is calculated from then length of the note. This means adsr is not suitable for use with MIDI events. The opcode madsr uses the linsegr mechanism and so can be used in MIDI applications

---

Sweepable Filters

ar **biquad** asig, kb0, kb1, kb2, ka0, ka1, ka2

ar **rezzy** asig, kfco, kres

ar **moogvcf** asig, kfco, kres

Implementation of a sweepable general purpose filter and two sweepable resonant low-pass filters.

PERFORMANCE

biquad is a general purpose biquadratic digital filter of the form:

a0*y(n) + a1*y[n-1] + a2*y[n-2] = b0*x[n] + b1*x[n-1] + b2*x[n-2]

This type of filter is often encountered in digital signal processing literature. It allows six user defined k-rate coefficients.

rezzy is a resonant low-pass filter created empirically by Hans Mikelson.

kfco is the filter cut-off frequency in Hz

kres is the amount of resonance. Values of 1 to 100 are typical. Resonance should be one or greater.

moogvcf is a digital emulation of the Moog diode ladder filter configuration. This emulation is based loosely on the paper "Analyzing the Moog VCF with Considerations for Digital Implemnetation" by Stilson and Smith (CCRMA). This version was originally coded in Csound by Josep Comajuncosas. Some modifications and conversion to C were done by Hans Mikelson.

Note: This filter requires that the input signal be normalized to one.

kfco is the filter cut-off frequency in Hz.

kres is the amount of resonance with self oscillation occurring when kres is approximately one.

Examples

```
;biquad example
kfcon   =     2*3.14159265*kfco/sr
kalpha                =                                                 1-
2*krez*cos(kfcon)*cos(kfcon)+krez*krez*cos(2*kfcon)
kbeta   =     krez*krez*sin(2*kfcon)-2*krez*cos(kfcon)*sin(kfcon)
kgama   =     1+cos(kfcon)
km1     =     kalpha*kgama+kbeta*sin(kfcon)
km2     =     kalpha*kgama-kbeta*sin(kfcon)
kden    =     sqrt(km1*km1+km2*km2)
kb0     =     1.5*(kalpha*kalpha+kbeta*kbeta)/kden
kb1     =     kb0
kb2     =     0
ka0     =     1
ka1     =     -2*krez*cos(kfcon)
ka2     =     krez*krez
ayn     biquad axn, kb0, kb1, kb2, ka0, ka1, ka2
        outs   ayn*iamp/2, ayn*iamp/2

;  Sta  Dur  Amp   Pitch Fco   Rez
i14 8.0  1.0  20000  6.00  1000  .8
i14 +    1.0  20000  6.03  2000  .95

;rezzy example
kfco      expseg 100+.01*ifco, .2*idur, ifco+100, .5*idur, ifco*.1+100,
.3*idur, .001*ifco+100
apulse1 buzz   1,ifqc, sr/2/ifqc, 1 ; Avoid aliasing
asaw    integ  apulse1
axn     =      asaw-.5
ayn     rezzy  axn, kfco, krez
        outs   ayn*iamp, ayn*iamp

;   Sta  Dur  Amp   Pitch  Fco   Rez
i10  0.0  1.0  20000  6.00   1000  2
i10  +    1.0  20000  6.03   2000  10

;moogvcf example
apulse1 buzz    1,ifqc, sr/2/ifqc, 1 ; Avoid aliasing
asaw    integ   apulse1
ax      =       asaw-.5
ayn     moogvcf ax, kfco, krez
        outs    ayn*iamp, ayn*iamp

;    Sta  Dur  Amp   Pitch  Fco   Rez
i11   4.0  1.0  20000  6.00   1000  .4
i11   +    1.0  20000  6.03   2000  .7
```

Author

Hans Mikelson

October 1998

Release Notes for 3.493
==========

These are the release notes for version 3.493, which will eventually becone 3.50

Bug Fixes

Pow now available again.
Internal changes to parser to make fewer calls to strcmp
Corrections to rand in a-rate case and 16 bit randoms

Language Changes
---

hetro had a wrong constant which would give rise to a little noise.

If the incorrect out opcode is used it now attempts to correct to the correct one, which is not necessarily correct.

new names dumpk rather than kdump introduced.

kon renamed midion

kfilter2 renamed filter2 (still not sure it works though)

The opcodes rand randi and randh take an additional, optional argument which if non zero gives a 31bit random nuber rather than the 16bit one.

Rising to a power is available in expressions with the ^ operator.
use with some caution as I am not sure that the precidence is correct.

An internal changes has changed the conditional compilation flag for the Ingalls' port from __MWERKS__ to macintosh; this should help the BeOS port.

Opcode Fixes
---

sndwarp had bugs on Linux

ramnd, randh and randi now take an additional operand, which if non-zero use a better randon number generator

bug in ntrpol fixed

MIDI on Linux may work.

New Opcodes
---

schedule -- schedule an instrument event

schedwhen -- conditional scheduling

lfo -- Low Frequency Oscilator with 6 shapes

midion2 -- MIDI turnon (G.Maldonado)

midiin -- (G.Maldonado)

midiout -- (G.Maldonado)

nrpn -- (G.Maldonado)

cpstmid -- (G.Maldonado)

streson -- string resonator (V.Lazzarini)

Other Changes:

Windows GUI Changes

==John ff
1998 Nov 1

==========

schedule, schedwhen

schedule	inst, iwhen, idur, ...
schedwhen	ktrigger, kinst, kwhen, kdur, ...

PERFORMANCE

schedule adds a new score event. The arguments are the same as in a score. The when time (p2) is measured from the time of this event.

If the duration is zero or negative the new event is of MIDI type, and inherits the release sub-event from the scheduling instruction.

In the case of schedwhen the event is only scheduled when the krate value ktrigger is first non-zero.

Examples:

```
;; Double hit and 1sec separation
instr 1
    schedule        2,  1, 0.5, p4, p5
a1  shaker          p4, 60, 0.999, 0, 100, 0
    out     a1
endin

instr 2
a1  marimba  p4, cpspch(p5), p6, p7, 2, 6.0, 0.05, 1, 0.1
    out     a1
endin

instr 3
kr  table  kr, 1
    schedwhen       kr, 1, 0.25, 1, p4, p5
endin
```

---

lfo

kr	**lfo**	kamp, kcps[, itype]
ar	**lfo**	kamp, kcps[, itype]

DESCRIPTION

A LFO of various shapes

INITALIZATION

itype -- determine the form of the oscilator
(default) 0: sine
1: triangles
2: square (biplar)
3: square (unipolar)
4: saw-tooth
5: saw-tooth(down)

The sine wave is implemented as a 4096 table and linear interpolation. The others are calculated.

PERFORMANCE

ar, kr - output signal
kamp - amplitude
kcps - frequency of oscilator

EXAMPLE:

```
instr 1
kp  lfo    10, 5, 4
ar  oscil  p4, p5+kp, 1
    out    ar
endin
```

---

minion2

midion2 kchn, knum, kvel, ktrig

DESCRIPTION

sends note on and off messages to the midi out port when triggered by a value different than zero.

PERFORMANCE

kchn - midi channel
knum - midi note number
kvel - note velocity
ktrig - trigger input signal (normally 0)

Similary to 'midion', this opcode sends note-on and note-off messages to the midi out port, but only when ktrig is different than zero. This opcode is thinked to work toghether with the output of the 'trigger' opcode.

(G.Maldonado)

---

pctrlinit

(G.Maldonado)

---

dpctrlinit

(G.Maldonado)

---

midiin

kstatus, kchan, kdata1, kdata2 **midiin**

DESCRIPTION
returns a generic midi message received by the midi in port

PERFORMANCE
kstatus - the type of midi message. Can be:
128 (note off),
144 (note on),
160 (polyphonic aftertouch),
176 (control change),
192 (program change),
208 (channel aftertouch),
224 (pitch bend)
or 0 if no midi message are pending in theMIDI IN buffer.

kchan - midi channel (1-16)
kdata1, kdata2 - message-dependent data values

midiin has no input arguments, because it reads at the midi in port implicitly. It works at k-rate. Normally (i.e. when no messages are pending) kstatus is zero, only each time midi data are present in the midi in buffer, kstatus is set to the type of the relative messages.

(G.Maldonado)

---

midiout

midiout kstatus, kchan, kdata1, kdata2

DESCRIPTION
sends a generic midi message to the midi out port

PERFORMANCE
kstatus - the type of midi message. Can be:
128 (note off),
144 (note on),
160 (polyphonic aftertouch),
176 (control change),
192 (program change),
208 (channel aftertouch),
224 (pitch bend)
or 0 when no midi messages must be sent to the MIDI OUT port.

kchan - midi channel (1-16)
kdata1, kdata2 - message-dependent data values

midiout has not output arguments, because it sends the message to the midi out port implicitly. It works at k-rate. It sends a midi message only when kstatus is different than zero.
Warning! Normally kstatus should be set to 0, only when the user intend to send a midi message, it can be set to the corresponding message type number.

(G.Maldonado)

---

nrpn

nrpn kchan, kparmnum, kparmvalue

DESCRIPTION

sends a nprn (Non Registered Parameter Number) message to the midi out port each time one of the input arguments changes.

PERFORMANCE

kchan - midi channel
kparmnum - number of NRPN parameter
kparmvalue - value of NRPN parameter

This opcode sends new message when the MIDI translated value of one of the input arguments changes. It operates at k-rate. Useful with the midi instruments that recognize NRPNs (for example with the newest sound-cards with internal midi synthesizer such as SB AWE32, AWE64, GUS etc. in which each patch parameter can be changed during the performance via NRPN)

(G.Maldonado)

---

cpstmid

icps **cpstmid** ifn

INITIALIZATION

ifn - function table containing the parameters
(numgrades, interval, basefreq, basekeymidi) and
the tuning ratios.

(init rate only)

This unit is similar to cpsmidi, but allows fully customized micro-tuning scales. It requires five parameters, the first ifn is the function table number of the tuning ratios, and the other parameters must be stored in the function tables itself. The function table ifn should be generated by the GEN2 and the first four values stored in this function are: numgrades (the number of grades of the micro-tuning scale), interval (the frequency range covered before repeating the grade ratios, for example 2 for one octave, 1.5 for a fift etcetera), basefreq (the base frequency of the scale in cps), basekeymidi (the midi-note-number to which to assign the basefreq unmodified).

After these four values, the user can begin to insert the tuning ratios. For example, for a standard 12-grade scale with the base-frequency of 261 cps assigned to the key-number 60, the corresponding f-statement in the score to generate the table should be:

; numgrades basefreq tuning-ratios (eq.temp) .....
; interval basekeymidi
f1 0 64 -2 12 2 261 60 1 1.059463094359 1.122462048309 1.189207115003 ..etc...

Another example with a 24-grade scale with a base frequency of 440 assigned to the key-number 48, and a repetition interval of 1.5:

```
;               numgrades    basefreq    tuning-ratios .....
;               interval          basekeymidi
f1 0 64 -2   24      1.5     440        48        1   1.01  1.02  1.03   ..etc...
```

(G.Maldonado)

---

ar **streson** asig, kfr, ifdbgain

An audio signal is modified by an string resonator with variable fundamental frequency.

INITIALIZATION

ifdbgain - feedback gain, between 0 and 1, of the internal delay line. A value close to 1 creates a slower decay and a more pronounced resonance. Small values may leave the input signal unaffected. Depending on the filter frequency, typical values are > .9.

PERFORMANCE

streson passes the input asig through a network composed of comb, low-pass and all-pass filters, similar to the one used in some versions of the Karplus-Strong algorythm, creating a string resonator effect. The fundamental frequency of the "string" is controled by the k-rate variable kfr.This opcode can be used to simulate sympathetic resonances to an input signal.

streson is an adaptation of the StringFlt object of the SndObj Sound Object Library developed by the author.

Victor Lazzarini
Music Department
National University of Ireland, Maynooth
Maynooth Co.Kildare
Ireland

Release Notes for 3.50

==========

These are the release notes for version 3.50. This accumulates a number of changes which have been released in bits, but there are even more here than previously released.

It incorporates significant bodies of code from Gabriel Maldonado and hans Mikelson, with contributions from Richard Boulanger, V.Lazzarini, Greg Sulliven, rasmus ekman, matt ingalls, Ed Hall, and many others who assisted in identifying bugs etc. (I really should maintain records of them all, but they know who they are I hope).

Bug Fixes

Pow now available again.
Internal changes to parser to make fewer calls to strcmp
Corrections to rand in a-rate case and 16 bit randoms
Two bugs in extending labels and goto tables corrected
Minor bug in extending instrument numbers fixed

Language Changes

---

hetro had a wrong constant which would give rise to a little noise.

If the incorrect out opcode is used it now attempts to correct to the correct one, which is not necessarily correct.

new names dumpk rather than kdump introduced.

kon renamed midion

kfilter2 renamed filter2 (still not sure it works though)

The opcodes rand randi and randh take an additional, optional argument which if non zero gives a 31bit random nuber rather than the 16bit one.

Rising to a power is available in expressions with the ^ operator.
use with some caution as I am not sure that the precidence is correct.

An internal changes has changed the conditional compilation flag for the Ingalls' port from __MWERKS__ to macintosh; this should help the BeOS port.

The single file .csd input has been extended for all command-line versions, and possibly for Windows. It can not decode additional parameters.

Id a file .csoundrc exists, it is read to set parameters first, which can be overridden. It used the .csd form so options are written as on the command line, with optional newlines at appropriate places. It does not set orc/sco names (as far as i can understand it)

Opcode Fixes

---

sndwarp had bugs on Linux

rand, randh and randi now take an additional operand, which if non-zero use a better randon number generator

bug in ntrpol fixed

MIDI on Linux may work, or may not....

Many changes to the pitchbend opcodes

moogvcf and rezzy can accept a-rate parameters, and moogvcf takes an optional scaling factor

foscil/foscili can take a-rate amplitude and frequency

biquad has an additional optional argument, which if non zero skips initialisation.

New Opcodes

---

schedule -- schedule an instrument event

schedwhen -- conditional scheduling

lfo -- Low Frequency Oscilator with 6 shapes

midion2 -- MIDI turnon (G.Maldonado)

midiin -- (G.Maldonado)

midiout -- (G.Maldonado)

nrpn -- (G.Maldonado)

cpstmid -- (G.Maldonado)

streson -- string resonator (V.Lazzarini)

mod opcodes -- to complete arithmetic operations

slider8, slider8f, islider8
slider16, slider16f, islider16
slider32, slider32f, islider32
slider64, slider64f, islider64
s16b14, is16b14, s32b14, is32b14 -- MIDI slider controls (G.Maldonado)

vco -- (Hans Mikelson)

planet -- (Hans Mikelson)

distort1 -- (Hans Mikelson)

pareq -- Implementation of Zoelzer's Parmentric Equalizer Filters (Hans Mikelson)

deltapn -- (Hans Mikelson)

Experimental opcodes:
---
oscil3 -- Just like oscili but with cubic interpolation
foscil3
losil3
table3
itable3
deltap3
vdelay3

Other Changes:
---

use of kdump or kon, while still allowed gives a message about deprecated opcodes.

Windows GUI Changes

None i think

==John ff
1999 Orthodox Christmas
==========

schedule, schedwhen

schedule inst, iwhen, idur, ...
schedwhen ktrigger, kinst, kwhen, kdur, ...

PERFORMANCE

schedule adds a new score event. The arguments are the same as in a score. The when time (p2) is measured from the time of this event.

If the duration is zero or negative the new event is of MIDI type, and inherits the release sub-event from the scheduling instruction.

In the case of schedwhen the event is only scheduled when the krate value ktrigger is first non-zero.

Examples:

```
;; Double hit and 1sec separation
instr 1
    schedule        2,  1, 0.5, p4, p5
a1  shaker          p4, 60, 0.999, 0, 100, 0
    out     a1
endin

instr 2
a1  marimba  p4, cpspch(p5), p6, p7, 2, 6.0, 0.05, 1, 0.1
    out     a1
endin

instr 3
kr  table  kr, 1
    schedwhen       kr, 1, 0.25, 1, p4, p5
endin
```

---

lfo

kr **lfo** kamp, kcps[, itype]
ar **lfo** kamp, kcps[, itype]

DESCRIPTION

A LFO of various shapes

INITALIZATION

itype -- determine the form of the oscilator
(default) 0: sine
1: triangles
2: square (biplar)
3: square (unipolar)
4: saw-tooth
5: saw-tooth(down)

The sine wave is implemented as a 4096 table and linear interpolation. The others are calculated.

PERFORMANCE

ar, kr - output signal
kamp - amplitude
kcps - frequency of oscilator

EXAMPLE:

```
instr 1
kp  lfo     10, 5, 4
ar  oscil   p4, p5+kp, 1
    out     ar
endin
```

---

minion2

midion2 kchn, knum, kvel, ktrig

DESCRIPTION

sends note on and off messages to the midi out port when triggered by a value different than zero.

PERFORMANCE

kchn - midi channel
knum - midi note number
kvel - note velocity
ktrig - trigger input signal (normally 0)

Similary to 'midion', this opcode sends note-on and note-off messages to the midi out port, but only when ktrig is different than zero. This opcode is thinked to work toghether with the output of the 'trigger' opcode.

(G.Maldonado)

---

midiin

kstatus, kchan, kdata1, kdata2 **midiin**

DESCRIPTION
returns a generic midi message received by the midi in port

PERFORMANCE
kstatus - the type of midi message. Can be:
128 (note off),
144 (note on),
160 (polyphonic aftertouch),
176 (control change),
192 (program change),
208 (channel aftertouch),
224 (pitch bend)
or 0 if nomidi message are pending in the MIDI IN buffer.

kchan - midi channel (1-16)
kdata1, kdata2 - message-dependent data values

midiin has no input arguments, because it reads at the midi in port implicitly. It works at k-rate. Normally (i.e. when no messages are pending) kstatus is zero, only each time midi data are present in the midi in buffer, kstatus is set to the type of the relative messages.

(G.Maldonado)

---

midiout

midiout kstatus, kchan, kdata1, kdata2

DESCRIPTION
sends a generic midi message to the midi out port

PERFORMANCE
kstatus - the type of midi message. Can be:
128 (note off),
144 (note on),
160 (polyphonic aftertouch),
176 (control change),
192 (program change),
208 (channel aftertouch),
224 (pitch bend)
or 0 when no midi messages must be sent to the MIDI OUT port.
kchan - midi channel (1-16)
kdata1, kdata2 - message-dependent data values

midiout has not output arguments, because it sends the message to the midi out port implicitly. It works at k-rate. It sends a midi message only when kstatus is different than zero.
Warning! Normally kstatus should be set to 0, only when the user intend to send a midi message, it can be set to the corresponding message type number.

(G.Maldonado)

---

nrpn

nrpn kchan, kparmnum, kparmvalue

DESCRIPTION

sends a nprn (Non Registered Parameter Number) message to the midi out port each time one of the input arguments changes.

PERFORMANCE

kchan - midi channel
kparmnum - number of NRPN parameter
kparmvalue - value of NRPN parameter

This opcode sends new message when the MIDI translated value of one of the input arguments changes. It operates at k-rate. Useful with the midi instruments that recognize NRPNs (for example with the newest sound-cards with internal midi synthesizer such as SB AWE32, AWE64, GUS etc. in which each patch parameter can be changed during the performance via NRPN)

(G.Maldonado)

---

cpstmid

icps **cpstmid** ifn

INITIALIZATION

ifn - function table containing the parameters
(numgrades, interval, basefreq, basekeymidi) and
the tuning ratios.
(init rate only)

This unit is similar to cpsmidi, but allows fully customized micro-tuning scales. It requires five parameters, the first ifn is the function table number of the tuning ratios, and the other parameters must be stored in the function tables itself. The function table ifn should be generated by the GEN2 and the first four values stored in this function are: numgrades (the number of grades of the micro-tuning scale), interval (the frequency range covered before repeating the grade ratios, for example 2 for one octave, 1.5 for a fift etcetera), basefreq (the base frequency of the scale in cps), basekeymidi (the midi-note-number to which to assign the basefreq unmodified).

After these four values, the user can begin to insert the tuning ratios. For example, for a standard 12-grade scale with the base-frequency of 261 cps assigned to the key-number 60, the corresponding f-statement in the score to generate the table should be:

```
;     numgrades    basefreq          tuning-ratios (eq.temp) ..
.....
;          nterval       basekeymidi
f1 0 64 -2 12 2    261     60    1  1.059463094359  1.122462048309
1.189207115003 ..etc...
```

Another example with a 24-grade scale with a base frequency
of 440 assigned to the key-number 48, and a repetition interval of 1.5:

```
;     numgrades      basefreq          tuning-ratios .......
;              interval       basekeymidi
f1 0 64 -2   24      1.5     440       48       1   1.01  1.02  1.03   ..etc...
```

(G.Maldonado)

---

ar **streson** asig, kfr, ifdbgain

An audio signal is modified by an string resonator with variable fundamental frequency.

INITIALIZATION

ifdbgain - feedback gain, between 0 and 1, of the internal delay line. A value close to 1 creates a slower decay and a more pronounced resonance. Small values may leave the input signal unaffected. Depending on the filter frequency, typical values are > .9.

PERFORMANCE

streson passes the input asig through a network composed of comb, low-pass and all-pass filters, similar to the one used in some versions of the Karplus-Strong algorythm, creating a string resonator effect. The fundamental frequency of the "string" is controled by the k-rate variable kfr.This opcode can be used to simulate sympathetic resonances to an input signal.

streson is an adaptation of the StringFlt object of the SndObj Sound Object Library developed by the author.

Victor Lazzarini
Music Department
National University of Ireland, Maynooth
Maynooth Co.Kildare
Ireland

Expression:

```
kr =      ka % kb
ar =      ka % ab
ar =      aa % kb
ar =      aa % ab
```

PERFORMANCE

Returns the valus a reduced by b, so the result in absolute value that the absoute value of b, by repeated subtraction. This is the same as a modulus function in the integer case.

---

ar **vco** kamp, kfqc, iwave, kpw, isine, imaxd

Implementation of an band limited analog modeled oscillator based on integration of band limited impulses.

Performance

vco can be used to simulate a variety of analog wave forms.

kamp determines the amplitude, kfqc is the frequency of the wave,

iwave determines the waveform 1 = sawtooth, 2 = Square/PWM, 3 = triangle/Saw Ramp

kpw determines the pulse width when iwave is set to 2 and determines Saw/Ramp character when iwave is set to 3. The value of kpw should be between 0 and 1. A value of .5 will generate a square wave or a triangle wave depending on iwave.

isine should be the number of a stored sine wave table.

imaxd is the maximum delay time. A time of 1/ifqc may be required for the pwm and triangle waveform. To bend the pitch down this value must be as large as 1/(minimum frequency).

Example

```
instr 10
  idur = p3 ; Duration
  iamp = p4 ; Amplitude
  ifqc = cpspch(p5) ; Frequency
  iwave = p6 ; Selected wave form 1=Saw, 2=Square/PWM, 3=Tri/Saw-Ramp-Mod
  isine = 1
  imaxd = 1/ifqc*2 ; Allows pitch bend down of two octaves
  kpw1 oscil .25, ifqc/200, 1
  kpw = kpw1 + .5
  asig vco iamp, ifqc, iwave, kpw, 1, imaxd
  outs asig, asig ; Ouput and amplification
endin

f1 0 65536 10 1
; Sta Dur Amp Pitch Wave
i10 0 2 20000 5.00 1
i10 + . . . 2
i10 . . . . 3
i10 . 2 20000 7.00 1
i10 . . . . 2
i10 . . . . 3
i10 . 2 20000 9.00 1
i10 . . . . 2
i10 . . . . 3
i10 . 2 20000 11.00 1
i10 . . . . 2
i10 . . . . 3
```

Author

Hans Mikelson

December 1998

---

aout **distort1** asig, kpregain, kpostgain, kshape1, kshape2

Implementation of modified hyperbolic tangent distortion.

PERFORMANCE

distort1 can be used to generate wave shaping distortion based on a modification of the tanh function.

$$aout = \frac{exp(asig * (pregain + shape1)) - exp(asig*(pregain+shape2))}{exp(asig*pregain) + exp(-asig*pregain)}$$

asig is the input signal.

kpregain determines the amount of gain applied to the signal before waveshaping. A value of 1 gives slight distortion.

kpostgain determines the amount of gain applied to the signal after waveshaping.

kshape1 determines the shape of the positive part of the curve. A value of zero gives a flat clip, small positive values give sloped shaping.

kshape2 determines the shape of the negative part of the curve.

Example

```
gadist  init    0

        instr   1
iamp    =       p4
ifqc    =       cpspch(p5)
asig    pluck   iamp, ifqc, ifqc, 0, 1
gadist  =       gadist + asig
        endin

        instr   50
kpre    init    p4
kpost   init    p5
kshap1  init    p6
kshap2  init    p7
aout    distort1 gadist, kpre, kpost, kshap1, kshap2
        outs    aout, aout
gadist  =       0
        endin

;   Sta  Dur  Amp    Pitch
i1  0.0  3.0  10000  6.00
i1  0.5  2.5  10000  7.00
i1  1.0  2.0  10000  7.07
i1  1.5  1.5  10000  8.00

;   Sta  Dur  PreGain PostGain Shape1 Shape2
i50 0    3    2      1        0      0
```

Author

Hans Mikelson
December 1998

PS Name chosen to avoid clash with XTC's distort opcode

---

outx, outy, outz **planet** kmass1, kmass2, ksep, ix, iy, iz, ivx, ivy, ivz, idelta, ifriction

Signal generator which loosely simulates a planet orbiting in a binary star system.

PERFORMANCE

planet simulates a planet orbiting in a binary star system. The outputs are the x, y and z coordinates of the orbiting planet. It is possible for the planet to achieve escape velocity by a close encounter with a star. This makes this system somewhat unstable.

kmass1 is the mass of the first star,

kmass2 is the mass of the second star,

ksep determines the distance between the two stars,

ix, iy, iz are the initial x, y and z coordinates of the planet,

ivx, ivy, ivz are the initial velocity vector components for the planet.

idelta is the step size used to approximate the differential equation.

ifriction is a value for friction which can used to keep the system from blowing up.

Example

```
        instr 1

idur    =   p3
iamp    =   p4
km1     =   p5
km2     =   p6
ksep    =   p7
ix      =   p8
iy      =   p9
iz      =   p10
ivx     =   p11
ivy     =   p12
ivz     =   p13
ih      =   p14
ifric   =   p15

kamp   linseg    0, .002, iamp, idur-.004, iamp, .002, 0

ax, ay, az planet km1, km2, ksep, ix, iy, iz, ivx, ivy, ivz, ih, ifric

        outs   ax*kamp, ay*kamp

        endin

;  Sta  Dur  Amp   M1  M2  Sep  X   Y  Z  VX  VY  VZ   h   Frict
i1  0    1    5000  .5  .35 2.2  0  .1  0  .5  .6 -.1  .5  -0.1
i1  +    .    .     .5  0   0    0  .1  0  .5  .6 -.1  .5   0.1
i1  .    .    .     .4  .3  2    0  .1  0  .5  .6 -.1  .5   0.0
i1  .    .    .     .3  .3  2    0  .1  0  .5  .6-.1  .5   0.1
i1  .    .    .     .25 .3  2    0  .1  0  .5  .6-.1  .5   1.0
i1  .    .    .     .2  .5  2    0  .1  0  .5  .6-.1  .1   1.0
```

Author

Hans Mikelson
December 1998

Banks of sliders

slider8,slider16,slider32,slider64
slider8f, slider16f,slider32f,slider64f
islider8,islider16,islider32,islider64
s16b14,is16b14,s32b14,is32b14

SYNTAX

k1,k2,k3,k4,k5,k6,k7,k8 **slider8** ichan, ictlnum1, imin1, imax1, init1, ifn1, ..., ictlnum8, imin8, imax8, init8, ifn8

k1, ... , k16 **slider16** ichan, ictlnum1, imin1, imax1, init1, ifn1, ..., ictlnum16, imin16, imax16, init16, ifn16

k1, ... , k32 **slider32** ichan, ictlnum1, imin1, imax1, init1, ifn1, ..., ictlnum32, imin32, imax32, init32, ifn32

k1, ... , k64 **slider64** ichan, ictlnum1, imin1, imax1, init1, ifn1, ..., ictlnum64, imin64, imax64, init64, ifn64

k1,k2,k3,k4,k5,k6,k7,k8 **slider8f** ichan, ictlnum1, imin1, imax1, init1, ifn1, icutoff1, ..., ictlnum8, imin8, imax8, init8, ifn8, icutoff8

k1, ... , k16 **slider16f** ichan, ictlnum1, imin1, imax1, init1, ifn1, icutoff1, ... , ictlnum16, imin16, imax16, init16, ifn16, icutoff16

k1, ... , k32 **slider32f** ichan, ictlnum1, imin1, imax1, init1, ifn1, icutoff1, ... , ictlnum32, imin32, imax32, init32, ifn32, icutoff32

k1, ... , k64 **slider64f** ichan, ictlnum1, imin1, imax1, init1, ifn1, icutoff1, ... , ictlnum64, imin64, imax64, init64, ifn64, icutoff64

i1, ... , i8 **islider8** ichan, ictlnum1, imin1, imax1, ifn1, ..., ictlnum8, imin8, imax8, ifn8

i1, ... , i16 **islider16** ichan, ictlnum1, imin1, imax1, ifn1, ..., ictlnum16, imin16, imax16, ifn16

i1, ... , i32 **islider32** ichan, ictlnum1, imin1, imax1, ifn1, ..., ictlnum32, imin32, imax32, ifn32

i1, ... , i64 **islider64** ichan, ictlnum1, imin1, imax1, ifn1, ..., ictlnum64, imin64, imax64, ifn64

i1, ... , i16 **s16b14** ichan, ictlno_msb1, ictlno_lsb1, imin1, imax1, initvalue1, ifn1, ..., ictlno_msb16, ictlno_lsb16, imin16, imax16, initvalue16, ifn16

i1, ... , i16 **is16b14** ichan, ictlno_msb1, ictlno_lsb1, imin1, imax1, ifn1, ... ictlno_msb16, ictlno_lsb16, imin16, imax16, ifn16

i1, ... , i32 **s32b14** ichan, ictlno_msb1, ictlno_lsb1, imin1, imax1, initvalue1, ifn1, ..., ictlno_msb32, ictlno_lsb32, imin32, imax32, initvalue32, ifn32

i1, ... , i32 **is32b14** ichan, ictlno_msb1, ictlno_lsb1, imin1, imax1, ifn1, ... ictlno_msb32, ictlno_lsb32, imin32, imax32, ifn32

DESCRIPTION

MIDI slider control banks

INITIALIZATION

i1 ... i64 - output values

ichan - midi channel (1-16)

ictlnum1 ... ictlnum64 - midi control number
ictlno_msb1 .... ictlno_msb32 - midi control number (most significant byte)
ictlno_lsb1 .... ictlno_lsb32 - midi control number (less significant byte)
imin1 ... imin64 - minimum values for each controller
imax1 ... imax64 - maximum values for each controller
init1 ... init64 - inital value for each controller
ifn1 ... ifn64 - function table for conversion for each controller
icutoff1 ... icutoff64 - low pass filter frequency cutoff for each controller

PERFORMANCE

k1 ... k64 - output values

isliderN, sliderN and sliderNf are banks of MIDI controller (useful when using midi mixer such as KAWAI MM-16 or others for changing whatever sound parameter in realtime. A software slider bank will be avalaible within short time).
The raw midi control messages at the input port are converted to agree with iminN and imaxN, and an initial value can be set. Also an optional non-interpolated function table with a custom translation curve is allowed, useful for enabling exponential response curves.

When no function table translation is required, set the ifnN value to 0, else set ifnN to a valid function table number. When table translation is enabled (i.e. setting ifnN value to a non-zero number referring to an already allocated function table), initN value should be set equal to iminN or imaxN value, else the initial output value will not be the same as specified in initN argument.
slider8 allows a bank of 8 different midi control message numbers, slider16 does the same with a bank of 16 controls, and so on.

sliderNf filter the signal before output for eliminating discontinuities due to the low resolution of the MIDI (7 bit); the cutoff frequency can be set separately for each controller (suggested range: .1 to 5 cps). Warning! sliderNf opcodes do not output the required initial

value immediately, but only after some k-cycle because the filter slightly delays the output.

As the input and output arguments are many, you can split the line using '\' (backslash) character (new in 3.47 version) to improve the readability. Using these opcodes is quite more efficient than using the separate ones (ctrl7 and ktone) when more controllers are required.

In isliderN there is not an initial-value input argument because the output is get directly from current status of internal controller array of Csound.

isNb14 and sNb14 opcodes are the 14-bit versions of these banks of controllers.

---
ar **pareq** asig, kc, iv, iq, imode

iv is volume boost or cut
iq is the quality factor (sqrt(.5) is no resonance)
imode is 0=Peaking EQ, 1=Low Shelf, 2=High Shelf

Performance

kc is the centre of shelf value
asig is the incoming signal

Eqample:

```
      instr 15
ifc           =     p4      ; Center / Shelf
iq            =     p5      ; Quality factor sqrt(.5) is no resonance
iv            =     ampdb(p6)   ; Volume Boost/Cut
imode  =     p7     ;Mode 0=Peaking EQ, 1=Low Shelf, 2=High Shelf
kfc    linseg  ifc*2, p3, ifc/2
asig   rand  5000                ; Random number source for testing
aout   pareq asig, kfc, iv, iq, imode ; Parmetric equalization
      outs  aout, aout            ; Output the results
endin

; SCORE:
;   Sta  Dur  Fcenter  Q       Boost/Cut(dB)  Mode
i15         0    1    10000          .2        12          1
i15         +    .    5000           .2        12          1
i15         .    .    1000           .707      -12         2
i15         .    .    5000           .1        -12         0
```

Hans Mikelson

---
oscil3, **foscil3**, **loscil3**, **vdelay3**, **table3**, **itable3**, **deltap3**

These are experimental opcodes which use cubic interpolation rather than teh linear interpolation of oscili, foscili, loscili, vdelay, tablei, itablei and deltapi. Testing so far has shown that oscil3 works and gives a better sound (on a 32 point sine wave). The others have not been tested and so shoudl be used with some care. Feedback on these is most acceptable

JPff

Release Notes for 3.51
==========

These are the release notes for version 3.51. This is mainly a small number of bug fixes from 3.50, but rather significant ones.

Bug Fixes

Use of C-style comment /* .. */ now works on both orchestra and score

Another attempt to get line continations working
Language Changes

Lines starting # or ; in .csoundrc or in the options part of a .csd file are treated as comments. Comments can also start where an option is expected.

Opcode Fixes
---

wgbow -- the pitch control was all wrong and has been rewritten. Also the bow slope had been removed; now restored.

oscil3 at k-rate was totally wrong; fixed

New Opcodes
---

envlpxr -- inadvertently lost; exponential, MIDI controlled envelope

xadsr -- ADSR opcode with exponential lines rather than linear

mxadsr -- ADSR with exponential curves and MIDI sensitive to release

Other Changes:

Code for follow recast

Windows GUI Changes

Can look for .csd files in orchestra field

==John ff
1999 in time for Luigi Nono's Birthday
==========

Release Notes for 3.53
==========

These are the release notes for version 3.53.

Bug Fixes

In vdelay it was possible for an error on wrapping the delay

(PC only) the shaker opcode did not work due to a file transfer failure.

envlpxr could cause a crash due to a typing error

Bug in wgflute which caused silent notes eliminated

Bug in diskin/soundin fixed

cpsmidi nolonger attempts to track pitchbend

Language Changes

Internal changes to NeXT added in many places (thanks to Stephen Brandon)

Strings are now recognised in scores for a large number fo opcodes (convolve, adsyn, diskin, soundin, pvoc etc.

ftlen upgraded so it works with deferred function tables (it loads the file)

opcode ondur/ondur2 renamed to noteondur/noteondur2.

peakk renamed peak (with internal discrimination)

Inside [ ] in the score the form ~ will give a randon number in the range 0 to 1.

Opcode Fixes
---

ftsr -- this opcode/function got lost at some stage, mea culpa

mandol -- not accepts a negative base-frequency to skip initialisation

In various wg opcodes, if minimum frequency is not given and the frequency is a k-rate value, instead of an error, a minimum of 50Hz is assumed with a warning

New Opcodes
---

nestedap -- nested allpass filters

lorenz -- ode generator

pitch -- a spectrum-based pitch-tracker

Other Changes:

Windows GUI Changes

==John ff
1999 Budget Day
==========

nestedap implements three different nested all-pass filters useful for implementing reverbs.

aout **nestedap** asig, imode, imaxdelay, idelay1, igain1 [, idelay2, igain2, idelay3, igain3]

Mode 1 is a simple all-pass filter.
Mode 2 is a single nested all-pass filter
Mode 3 is a double nested all-pass filter.

Note imaxdelay is not currently used but will be necessary if k-rate delay is implemented.

Example:

```
instr 5
insnd       =           p4
gasig       diskin      insnd, 1
endin

instr 10
  imax      =           1
  idel1     =           p4
  igain1    =           p5
  idel2     =           p6
  igain2    =           p7
  idel3     =           p8
  igain3    =           p9
  idel4     =           p10
  igain4    =           p11
  idel5     =           p12
  igain5    =           p13
  idel6     =           p14
  igain6    =           p15
  afdbk     init        0

aout1       nestedap    gasig+afdbk*.4, 3, imax, idel1, igain1, idel2,
            igain2, idel3,    igain3
aout2       nestedap    aout1, 2, imax, idel4, igain4, idel5, igain5
            aout nestedap    aout2, 1, imax, idel6, igain6
afdbk       butterlp    aout, 1000
            outs        gasig+(aout+aout1)/2, gasig-(aout+aout1)/2
  gasig     =           0
endin
f1 0 8192 10 1

; Diskin
;   Sta  Dur  Soundin
i5  0    3    1

; Reverb
;    Sta   Dur   Del1 Gain1 Del2 Gain2   Del3 Gain3 Del4 Gain4 Del5
Gain5 Del6 Gain6
i10  0    4    97   .11   23   .07    43   .09   72   .2    53   .2    119  .3
```

---
lorenz implements the lorenz system of equations:

ax, ay, az **lorenz** ksv, krv, kbv, kh, ix, iy, iz, iskip

```
instr 20
  ksv       =           p4
  krv       =           p5
  kbv       =           p6

  ax, ay,az lorenz      ksv, krv, kbv, .01, .6, .6, .6, 1
            outs        ax*1000, ay*1000
endin

; Lorenz system
;   Sta  Dur  S    R   V
i20 5    1    10   28  2.667
```

---
pitch is a spectrum-based pitch tracker

koct, kamp **pitch** asig, iupdte, ilo, ihi, idbthresh[, ifrqs, iconf, istrt, iocts, ifrqs, iq, inptls, irolloff, istor]

The input signal is analysed to give a pitch/amplitude pair for the strongest pitch in the signal. The value is updates every iupdte seconds.

INITIALISATION

ilo, ihi -- range in which pitch is detected (as decimal octaves)

idbthresh -- energy level in decibells necessary for pith to be detected. Once started it continues until it is 6bd down

iconf -- the number of conformations needed for an octave jump. Default value is 10

istrt -- starting pitch for tracker, defaults to average of ilo and ihi.

iocts -- number of octave decimations in spectrum, defaulting to 6

ifrqs -- number of divisons of an octave, defaults to 12 and is limited to 120

iq -- Q rate of analysis, dafaulting to 10

inptls, irolloff -- number of harmonic partials used in matching. Defaulst to 4 and 0.6

istor -- is none zero skips initialisation

PERFORMANCE

Using the same techniques as spectrum and specptrk estimates the pitch of the signal. Pitch is reported in decimal octave form, and amplitude in db

While the default settings are reasonable for general use, some experimentation may be necessary for complex sounds.

Release Notes for 3.54
==========

These are the release notes for version 3.54

Bug Fixes
---

in -o there were some bracketing difficulties.

Arguments in macros are now checked for length overflowing internal buffer

fm4op opcodes could give rubbish die to uninitialised array.

Function nsamp made usable

Language Changes

For piped output to work there must not be a WAV or AIFF header (they require a rewind). This is not checked.

The default sound file is test, test.wav or test.aif depending on selected format.

There are now y and z type arguments in entry.c (from Gabriel)

When using line events the e event is now accepted

Both .csd and .CSD files are accepted as description files

The system expects to have a file csound.txt for strings. This allows for languages other than American.

Opcode Fixes

An error message in pvread said it was from pvoc. Changed to correct opcode.

pareq, rezzy, moogvcf and biquad optimised a little

New Opcodes
---

sum -- add together arbitrary number of arguments

product -- multiply arbitrary number of arguments

Other Changes:

Internal changes to optimise all irate random opcodes; not much though.

Internally the variable PMASK has been renamed PHMASK as (a) that is a better description and (b) it caused problems on Solaris

Windows GUI Changes

Automatic adding of .wav or .aif on sound files

==John ff
1999 May 17

==========

aout **sum** a1, a2, a3, ...
aout **product** a1, a2, a3, ...

DESCRIPTION

The signals are added or multiplied together to give the output signal.

PERFORMANCE

a1, a2,.. -- audio inputs

Release Notes for 3.55
==========

These are the release notes for version 3.55

Bug Fixes
---

only in opcodes (below)

Language Changes

The environment variable CSSTRNGS is used to identify the string database. If it is not present it looks in SSDIR SADIR etc and finally /usr/local/lib

This can be overridden with a -j filename option

Opcode Fixes
---

linseg, linsegr -- an off-by-one error corrected in all cases

buzz, gbuzz -- error case reported only once per note instead of every k-cycle in error.

loscil3 -- ignored the amplitude leading to usually quiet output

mandolin -- Bug fixed which stop the initial pluck, and also rescaled

New Opcodes
---

svfilter -- Implementation of a resonant second order filter, with simultaneous lowpass, highpass and bandpass outputs.

hilbert -- An IIR implementation of a Hilbert transformer.

resonr, resonz -- Implementations of a second-order, two-pole two-zero bandpass filter with variable frequency response.

mac, maca -- Multiply and Accumulate instructions

Other Changes:

Windows GUI Changes
---

devaudio as an output device was incorrectly changed to devaudio.wav

---

==John ff
1999 June 20

==========

hilbert

areal, aimag **hilbert** asig

DESCRIPTION

An IIR implementation of a Hilbert transformer.

PERFORMANCE

hilbert is an IIR filter based implementation of a broad-band 90 degree phase difference network. The input to hilbert is an audio signal, with a frequency range from 15 Hz to 15 kHz. The outputs of hilbert have an identical frequency response to the input (i.e. they sound the same), but the two outputs have a constant phase difference of 90 degrees, plus or minus some small amount of error, throughout the entire frequency range - the outputs are in quadrature. hilbert is useful in the implementation of many digital signal processing techniques that require a signal in phase quadrature. areal corresponds to the cosine output of hilbert, while aimag corresponds to the sine output; the two outputs have a constant phase difference throughout the audio range that corresponds to the phase relationship between cosine and sine waves.

Internally, hilbert is based on two parallel 6th-order allpass filters. Each allpass filter implements a phase lag that increases with frequency; the difference between the phase lags of the parallel allpass filters at any given point is approximately 90 degrees. Unlike an FIR-based Hilbert transformer, the output of hilbert does not have a linear phase response. However, the IIR structure used in hilbert is far more efficient to compute, and the nonlinear phase response can be used in the creation of interesting audio effects, as in the second example below.

AUTHOR

Sean Costello
Seattle, Washington
1999

---

svfilter

alow, ahigh, aband **svfilter** asig, kcf, kq[, iscl]

DESCRIPTION

Implementation of a resonant second order filter, with simultaneous lowpass, highpass and bandpass outputs.

INITIALIZATION

iscl - coded scaling factor, similar to that in reson. A non-zero value signifies a peak response factor of 1, i.e. all frequencies other than kcf are attenuated in accordance with the (normalized) response curve. A zero value signifies no scaling of the signal, leaving that to some later adjustment (see balance). The default value is 0.

PERFORMANCE

svfilter is a second order state-variable filter, with k-rate controls for cutoff frequency and Q. As Q is increased, a resonant peak forms around the cutoff frequency. svfilter has simultaneous lowpass, highpass, and bandpass filter outputs; by mixing the outputs together, a variety of frequency responses can be generated. The state-variable filter, or "multimode" filter was a common feature in early analog synthesizers, due to the wide variety of sounds available from the interaction between cutoff, resonance, and output mix ratios. Svfilter is well suited to the emulation of "analog" sounds, as well as other applications where resonant filters are called for.

asig - Input signal to be filtered.

kcf - Cutoff or resonant frequency of the filter, measured in cps.

kq - Q of the filter, which is defined (for bandpass filters) as bandwidth/cutoff. kq should be in a range between 1 and 500. As kq is increased, the resonance of the filter increases, which corresponds to an increase in the magnitude and "sharpness" of the resonant peak. When using svfilter without any scaling of the signal (where iscl is either absent or 0), the volume of the resonant peak increases as Q increases. For high values of Q, it is recommended that iscl be set to a non-zero value, or that an external scaling function such as balance is used.

svfilter is based upon an algorithm in Hal Chamberlin's Musical Applications of Microprocessors (Hayden Books, 1985).

AUTHOR

Sean Costello
Seattle, Washington
1999

---

resonr, resonz

ar **resonr** asig, kcf, kbw[,iscl, istor]
ar **resonz** asig, kcf, kbw[,iscl, istor]

DESCRIPTION

Implementations of a second-order, two-pole two-zero bandpass filter
with variable frequency response.

INITIALIZATION

The optional initialization variables for resonr and resonz are identical to the i-time variables for reson.

istor - initial disposition of internal data space. Since filtering incorporates a feedback loop of previous output, the initial status of the storage space used is significant. A zero value will clear the space; a non-zero value will allow previous information to remain. The default value is 0.

iscl - coded scaling factor for resonators. A value of 1 signifies a peak response factor of 1, i.e. all frequencies other than kcf are attenuated in accordance with the (normalized) response curve. A value of 2 raises the response factor so that its overall RMS value equals 1. (This intended equalization of input and output power assumes all frequencies are physically present; hence it is most applicable to white noise.) A zero value signifies no scaling of the signal, leaving that to some later adjustment (see balance). The default value is 0.

PERFORMANCE

resonr and resonz are variations of the classic two-pole bandpass resonator (reson). Both resonr and resonz have two zeroes in their transfer functions, in addition to the two poles. resonz has its zeroes located at z = 1 and z = -1. resonr has its zeroes located at +sqrt(R) and -sqrt(R), where R is the radius of the poles in the complex z-plane. The addition of zeroes to resonr and resonz results in the improved selectivity of the magnitude response of these filters at cutoff frequencies close to 0, at the expense of less selectivity of frequencies above the cutoff peak. resonr and resonz have very close to constant-gain as the center frequency is swept, resulting in a more efficient control of the magnitude response than with traditional two-pole resonators such as reson. resonr and resonz produce a sound that is considerably different from reson, especially for lower center frequencies; trial and error is the best way of determining which resonator is best suited for a particular application.

asig - Input signal to be filtered.

kcf - Cutoff or resonant frequency of the filter, measured in cps.

kbw - Bandwidth of the filter (the cps difference between the upper and lower half-power points).

AUTHOR

Sean Costello
Seattle, Washington
1999

---

mac and maca

ar **mac** ksig1, asig2, ksig3, asig4, ...
ar **maca** asig1, asig2, asig3, asig4, ...

DESCRIPTION

Multiplies the arguments in pairs and accumulates their sum
ar = ksig1*asig2 + ksig3*asig4 + ...
ar = asig1*asig2 + asig3*asig4 + ...

INITIALIZATION

none

PERFORMANCE

ksign - multipliers (scales) of signals

asign - Audio signals to be summed/scaled

Release Notes for 3.56
==========

These are the release notes for version 3.56

Bug Fixes
---

pset opcode was ignored.

The ~ operator within [] in a score was wrong and did not work

Language Changes

There are two new operators in scores, within arithmetic contexts [ ].
@ followed by a number yields the power of two equal or greater than the number given. The operator @@ gives the power-of-two-plus1 equal or greater than the number given.

Opcode Fixes
---

follow had an off-by-one error which meant it increased but never decreased

New Opcodes
---

clockon
clockoff
readclock -- Performance timing opcodes

resony -- A bank of second-order bandpass filters, connected in parallel.

fold -- Adds artificial foldover to an audio signal

vincr -- incretment an audio variable
clear -- Clear audio variables [Note: these opcodes have results on right so may lead to incorrect warnings]

fout
foutk
fouti
foutir -- Outout to audio files

fiopen
fin
fink
fini -- Input from audio files

Other Changes:

Some internal reorganisation.

Windows GUI Changes

New button and dialog box to set SSDIR, SADIR and SFDIR. Also csound.txt name cached.

Editors are spawned in NOWAIT mode so can exist while setting options

Playback can be interrupted after "Play at End"

==John ff
1999 July 20
==========

resony

ar **resony** asig, kbf, kbw, inum, ksep [, iscl, istor]

DESCRIPTION

A bank of second-order bandpass filters, connected in parallel.

INITIALIZATION

inum - number of filters. Defaults to 4
iscl - coded scaling factor for resonators. A value of 1 signifies a peak response factor of 1, i.e. all frequencies other than kcf are attenuated in accordance with the (normalized) response curve. A value of 2 raises the response factor so that its overall RMS value equals 1. (This intended equalization of input and output power assumes all frequencies are physically present; hence it is most applicable to white noise.) A zero value signifies no scaling of the signal, leaving that to some later adjustment (e.g. see balance). The default value is 0.
istor - initial disposition of internal data space. Since filtering incorporates a feedback loop of previous output, the initial status of the storage space used is significant. A zero value will clear the space; a non-zero value will allow previous information to remain. The default value is 0.

PERFORMANCE

asig - audio input signal
kbf - base frequency, i.e. center frequency of lowest filter in Hz
kbw - bandwidth in Hz
ksep - separation of the center frequency of filters in octaves

resony is a bank of second-order bandpass filters, with k-rate variant frequency separation, base frequency and bandwidth, connected in parallel (i.e. the resulting signal is a mixing of the output of each filter). The center frequency of each filter depends of kbf and ksep variables.

EXAMPLE:
asig, kbf, kbw, inum, ksep [, iscl, istor]

In this example the global variable gk1 modifies kbf, gk2 modifies kbw, gk3 inum, gk4 ksep and gk5 the main volume.

```
      instr  1
a1          soundin "myfile.aif"
a2          resony  a1,  gk1 , gk2 ,i(gk3),gk4 ,2
      out    a2 * gk5
      endin
```

---

fold

ar **fold** asig, kincr

DESCRIPTION

Adds artificial foldover to an audio signal

PERFORMANCE

asig - input signal
kincr - amount of foldover expressed in multiple of sampling rate. Must be >= 1

fold is an opcode which creates artificial foldover. For example, when kincr is equal to 1 with sr=44100, no foldover is added, when kincr is set to 2 the foldover is equivalent to a downsampling to 22050, when it is set to 4 to 11025 etc. Fractional values of kincr are possible, allowing a continuous variation of foldover amount. This can be used for a wide range of special effects.

EXAMPLE:

```
      instr  1
kfreq       line   1,p3,200
a1          oscili  10000, 100, 1
k1          init   8.5
a1          fold   a1, kfreq
      out    a1
      endin
```

---

vincr, clear

vincr asig, aincr
clear avar1 [,avar2, avar3,...,avarN]

DESCRIPTION

vincr increments an audio variable of another signal, i.e. accumulates output.
clear zeroes a list of audio signals.

PERFORMANCE

asig - audio variable to be incremented
aincr - incrementation signal
avar1 [,avar2, avar3,...,avarN] - signals to be zeroed

vincr (variable increment) and clear are thinked to be used togheter. vincr stores the result of the sum of two audio variables into the first variable itself (which is thinked to be used as accumulator in case of polyphony). The accumulator-variable can be used for output signal by means of fout opcode. After the disk writing operation, the accumulator-variable should be set to zero by means of clear opcode (or it will explode).

---

fout, foutk, fouti, foutir, fiopen

fout "ifilename", iformat, aout1 [, aout2, aout3,.... ,aoutN]
foutk "ifilename", iformat, kout1 [, kout2, kout3,....,koutN]
fouti ihandle, iformat, iflag, iout1 [, iout2, iout3,....,ioutN]
foutir ihandle, iformat, iflag, iout1 [, iout2, iout3,....,ioutN]
ihandle **fiopen** "ifilename",imode

DESCRIPTION

fout, foutk, fouti and foutir output N audio, k or i-rate signals to a specified file of N channels.
fiopen can be used to open a file in one of the specified modes.

INITIALIZATION

ifilename - a double-quote delimited string file name iformat - a flag to choose output file format:
for fout and foutk only:
0 - 32-bit floating point samples without header (binary PCM multichannel file)
1 - 16-bit integers without header (binary PCM multichannel file)
2 - 16-bit integers with type header from -W -A or -J (mono or stereo file)

for fouti and foutir only:
0 - floating point in text format
1 - 32-bit floating point in binary format

iflag - choose the mode of writing to the ascii file (valid only in ascii mode; in binary mode iflag has no meaning, but it must be present anyway).

iflag can be a value choosen among the following:
0 - line of text without instrument prefix
1 - line of text with instrument prefix (see below)
2 - reset the time of instrument prefixes to zero (to be used only in some particular cases. See below)

iout,... ioutN - values to be written to the file.

imode - choose the mode of opening the file.
imode can be a value choosen among the following:
0 - open a text file for writing
1 - open a text file for reading
2 - open a binary file for writing
3 - open a binary file for reading

PERFORMANCE

aout1,... aoutN - signals to be written to the file.
kout1,...koutN - signals to be written to the file.

fout (file output) writes samples of audio signals to a file with any number of channels. Channel number depends by the number of aoutN variables (i.e. a mono signal with only an a-rate argument, a stereo signal with two a-rate arguments etc.) Maximum number of channels is fixed to 64.
More fout opcodes can be present in the same instrument, referring to different files.
Notice that, differently by out, outs and outq, fout does not zeroes the audio variable, so you must provide a zeroing after calling fout if poliphony is used. You can use incr and clear opcodes for this task.

foutk operates in the same way of fout, but with k-rate signals. iformat can be set only to 0 or 1.

fouti and foutir write i-rate values to a file. The main use of these opcodes is to generate a score file during a realtime session. For this purpose the user should set iformat to 0 (text file output) and iflag to 1, which enable the output of a prefix consisting of the following strings:

i num actiontime duration

before the values of iout1...ioutN arguments. Prefix is referring to instrument number, action time and duration of current note.

The difference of fouti and foutir is that, in the case of fouti, when iflag is set to 1, the duration of the first opcode is undefined (so it is replaced by a dot) wheras in the case of foutir is defined at the end of note, so the corresponding text line is written only at the end of the current note (in order to recognize its duration). The corresponding file is linked by the ihandle value generated by fiopen opcode (see below). So fouti and foutir can be used to generate a Csound score while playing a realtime session.

fiopen opens a file to be used by the foutX opcodes. It must be defined externally by any instruments, in the header section. It returns a number ihandle, which is univocally referring to the opened file.

Notice that fout and foutk can use both a string containing a file pathname or a handle-number generated by fiopen, wheras in the case of fouti and foutir, the target file can be only specified by means of a handle-number.

---

fin, fink, fini

fin "ifilename", iskipframes, iformat, ain1 [, ain2, ain3,.... ,ainN]
fink "ifilename", iskipframes, iformat, kin1 [, kin2, kin3,.... ,kinN]
fini "ifilename", iskipframes, iformat, in1 [, in2, in3,.... ,inN]

DESCRIPTION

read signals from a file (at a, k, and i-rate)

INITIALIZATION

ifilename - input file name (can be a string or a handle number generated by fiopen)
iskipframes - number of frames to skip at the start (every frame contains a sample of each channel)
iformat - a number specifying the input file format: for fin and fink:
0 - 32 bit floating points without header
1 - 16 bit integers without header

for fini:
0 - floating points in text format (loop; see below)
1 - floating points in text format (no loop; see below)
2 - 32 bit floating points in binary format (no loop)

fin (file input) is the complement of fout: it reads a multi channel file to generate audio rate signals. At present time no header is supported for file format. The user must be sure that the number of channel of the input file is the same of the number of ainX arguments

fink is the same as fin, but operates at k-rate.

fini is the complement of fouti and foutir, it reads the values each time the corresponding instrument note is activated. When iformat is set to 0, if the end of file is reached the file pointer is zeroed, restarting the scanning from the beginning. When iformat is set to 1 or 2 no loop is enabled, so at the end of file the corresponding variables will be filled with zeroes.

---

clockon, clockoff, readclock

clockon inum
clockoff inum
ival readclock inum

DESCRIPTION

Starts and stops one of a number of internal clocks, and read value of a clock.

INITIALIZATION

inum is the number of a clock. There are 32 clocks numbered 0 through 31; all other values are mapped to clock number 32.
[Note: in 3.56 a bug means that xloxk zero is always used -- fixed in source!]

PERFORMANCE

Between a clockon and a clockoff the CPU time used is accumulated in the clock. The precision is machine dependent, but is milliseconds on UNIX and Windows.

readclock reads the current value of a clock at initialisation time.

Note there is no way to zero a clock.

Release Notes for 3.57
==========

These are the release notes for version 3.57

Bug Fixes
---

clock opcodes all mapped to clock 0 -- fixed

divz was decoded incorrectly in parsing

The triple strike in marimba never happened due to programming error.
The percentage of doubles and singles are variable as two optional arguments, both defaulting to 20%.

Some error and warning strings were wrong. Extensively reviewed and fixed

Language Changes

In GEN23 (read ASCII file of numbers it is now possible to have a length of 0 and have the generator calculated from the number of numbers in the file.

Opcode Fixes
---

in buzz and gbuzz at least 1 harmonic is always used, and the absolute value of the number is used rather than giving a warning bug in wgbrass fixed which could lead to crashes

New Opcodes
---

active -- tell how many active instances there are of an instrument
cpuprc -- limit number of allocations of an instrument by load
maxalloc -- limit number of allocations of an instrument count
prealloc -- create a pool of unactive instances

expsega -- a-rate exponential segments

logbtwo
powoftwo -- fast versions of pow and log in both i and k position

```
ilen        filelen     ifilcod     ; returns length of "ifilcod" in seconds
isr         filesr      ifilcod     ; returns the sample rate of "ifilcod"
inchnls     filenchnls ifilcod      ; returns the number of chnls of "ifilcod"
ipeak       filepeak ifilcod, [ichnl]  ; returns peak absolute value of
                                                ; "ifilcod"
                                    ; if ichnl=0, returns peak out of all channels
                                    ; if ichnl>0, returns the peak of ichnl
                                    ; if ichnl is not specified, returns
                                    ; peak of the entire file.
                                    ; currently only supports AIFF_C float files
```

Other Changes:

The pvtool utility has been included in utils2, and as a -U option

Windows GUI Changes

New dialog for pvlook utility

==John ff
1999 August 3
==========

active

inum **active** ins

DESCRIPTION

Returns the number of active instances of instrument number ins

---
expsega

asig **expsega** ia, idur1, ib[, idur2, ic[...]]

DESCRIPTION

An exponential segment generator operating at a-rate. This unit is almost identical to expseg, but very more precise when defining segments with very short duration (i.e. in percussive attack phase) at audio-rate.

Note that expseg opcode does not operate correctly at audio rate when segments are shorter than a k-period.

INITIALISATION

ia - starting value. Zero is illegal.
ib, ic, etc. - value after idur1 seconds, etc. must be non-zero and must agree in sign with ia.
idur1 - duration in seconds of first segment.

A zero or negative value will cause all initialization to be skipped.

idur2, idur3, etc. - duration in seconds of subsequent segments.
A zero or negative value will terminate the initialization process with the preceding point, permitting the last-defined line or curve to be continued indefinitely in performance. The default is zero.

PERFORMANCE

These units generate control or audio signals whose values can pass through 2 or more specified points. The sum of dur values may or may not equal the instrument's performance time: a shorter performance will truncate the specified pattern, while a longer one will cause the last-defined segment to continue on in the same direction.
---
powoftwo(x)
logbtwo(x)

powoftwo( ) function returns 2 ^ x and allows positive and negatives numbers as argument.

logbtwo( ) returns the logarithm base two of x.
If the argument is in the range [-5,+5] for powoftwo( ) or [0.25,4] for logbatwo( ) then an internal table is used, allowing a precision more fine than one cent in a range of ten octaves. Outside those ranges the value is calculated afresh and will be as slow as use of pow or log.

logbtwo( ) returns the logarithm base two of x.

Also they are very useful when working with tuning ratios. They work at i and k-rate.
---

cpuprc
maxalloc
cpuprc instrnum, ipercent
maxalloc instrnum, icount
prealloc instrnum, icount

DESCRIPTION
cpuprc sets the cpu processing-time percent usage of an instrument in order to avoid buffer underrun in realtime performances maxalloc limits the number of allocations of an instrument.
prealloc creates space for instruments but does not run them

INITIALIZATION

instrnum - instrument number
ipercent - percent of cpu processing-time to assign
icount -- number of instruments instances that can be allocated

cpuprc is an opcode that enables a sort of polyphony theshold. The user must set ipercent value for each instrument he want to activate in realtime. It is assumed that the total theorical processing time of the cpu of the computer is 100%, but note that this percent value can only be defined empirically.

For example if ipercent is set to 5% for instrument 1, the maximum number of voices that can be allocated at any time will be 20 (as 5% X20 = 100%). If the user attempts to play a further note while the 20 previous notes are still playing, Csound inhibits the allocation of that note and will display a warning message.

In order to avoid audio buffer underruns, it is suggested to set the maximum number of voices a bit below the real processing power of the computer, because sometimes an instrument can require more processing time than normal (for example, if the instrument contains an oscillator which reads a table that doesn't fit in cache memory, it will be slower than normal; also, any concurrent program which run in multitasking, can subtract more processing power in some cases, less power in other cases etc.)

Initially all instruments are set to a default value of ipercent = 0.0% (i.e. zero processing time or rather infinite cpu processing-speed). Note that this opcode can be used either at instrument 0 time or dynamically, when it only affects later instruments. Any active instuenmt whose load is changed may lead to incorrect or anomolous results.

In maxalloc setting the number of maximum allocation to 0 means unlimited allocations are allowed. A negative allocation disallows any allocation.

example:

```
sr = 44100
kr = 441
ksmps = 100
nchnls = 2

cpuprc 1, 2.5          ;** set instr 1 to 2.5% of cpu, max 40 voices
cpuprc 2, 33.333       ;** set instr 2 to 33.333% of cpu, max 3 voices

instr 1
...body...
endin

instr 2
....body...
endin
```

Release Notes for 3.58
==========

These are the release notes for version 3.58

Bug Fixes

A file in .csd was left open which stopped some things working

Language Changes

Number of arguments to macros in both score and orchestra is unrestricted, and spaces are now allowed in argument lists

Blank lines and comments in .csd files allowed

Opcode Fixes
---

readk opcodes could not have worked as they were.

fof/fog only allocate space if phs is positive, to allow for legato

some improvement in streson (but not yet correct)

Avoid some crashes when using MIDI in non-midi context

New Opcodes
---

adsynt -- Additive synthesis

hsboscil -- Oscilator with brightness and tonality control

pitchamdf -- Pitch following

Other Changes:

Windows GUI Changes

Windows GUI Changes:
The xyin opcode should now read the mouse at the requested rate.

---
xyin

Windows Implementation Note

In this implementation, mouse input is read from the full screen rather than clipped to the Winsound output window. If you use more instances of xyin in an orchestra, they will only do different scaling of the mouse cursor position (this was also true in the earlier version).
The bottom left screen position is minimum for x and y.
Note that the graphics display option must be set to Full for the xyin operator to be functional.

Example:

```
sr = 22050
kr =  294
ksmps = 75
nchnls = 2

instr 1 ; Simple xyin test
        ; Let oscillators range 20 - 2000 Hz
        kcps1, kcps2 xyin .03, 20, 2000, 20, 2000, 500, 300
        ; Smooth input
        kcps1 port kcps1, .01
        kcps2 port kcps2, .01

        ; Use input
        kamp linseg 0, .5, 20000, p3-1, 20000, .5, 0
        kndx oscil 4, kcps1 / 50, 1
        kndx = kndx+5
        ar1 foscil kamp, 1, kcps1, kcps2, kndx, 1
        ar2 foscil kamp, 1, kcps2, kcps1, kndx, 1

                outs ar1, ar2
endin

; Score:
f1 0 4096 10 1      ; sine

i1 0 10
e

; End score
```

---

ar **adsynt** kamp, kcps, ifn, ifreqtbl, iamptbl, icnt [, iphs]

DESCRIPTION
This opcode performs additive synthesis with an arbitrary number of partials (not necessarily harmonic). Frequency and amplitude of each partial is given in the two tables provided. The purpose of this opcode is to have an instrument generate synthesis parameters at k-time and write them to the global parameter tables with the tablew opcode.

INITIALIZATION
ifn - a function table, usually a sine. Table values are not interpolated for performance reasons, so you better provide a larger table for better quality.
ifreqtbl - an arbitrary function table. Size has to be at least icnt. Table can contain frequency values for each partial at start, but is usually used for generating parameters at runtime with tablew. Frequencies must be relative to kcps.
iamptbl - same as ifreqtbl for relative partial amplitudes.
icnt - number of partials to be generated.
iphs - initial phase if each oscillator, if -1 initialization is skipped. If > 1 all phases will be initialized with a random value.

PERFORMANCE
kamp - Amplitude of note.
kcps - Base frequency of note. Partial frequencies will be relative to kcps.
==========

hsboscil
---

ar **hsboscil** kamp, ktona, kbrite, ibasef, ifn, imixtbl [, ioctcnt] [, iphs]

DESCRIPTION
This oscillator takes tonality and brightness as arguments, relative to a base frequency (ibasef). Tonality is a cyclic parameter in the logarithmic octave, brightness is realized by mixing multiple weighted octaves.
It is useful when tone space is understood in a concept of polar coordinates.
If you run ktona as a line and keep kbrite constant, you get Risset's glissando.
Oscillator table ifn is always read interpolated.
Performance time requires about ioctcnt * oscili.

INITIALIZATION

ibasef - a base frequency to which tonality and brighness are relative.
ifn - a function table, usually a sine.
imixtbl - a function table used for weighting the octaves, usually something like: f n 0 1024 -19 1 0.5 270 0.5
ioctcnt - number of octaves used for brightness blending, default is 3, minimum 2, maximum 10.
iphs - initial phase if the oscillator, if -1 initialisation is skipped.

PERFORMANCE

kamp - Amplitude of note.
ktona - Cyclic tonality parameter relative to ibasef in logarithmic octave, range 0 - 1, values > 1 can be used and are internally reduced to frac(ktona).
kbrite - brightness parameter relative to ibasef achieved by weighting ioctcnt octaves. It is scaled in a way that a value of 0

corresponds to orignal ibasef, 1 one octave above, -2 two octaves below ibasef etc. and any fractional value in between.
==========

pitchamdf
---

kcps, krms **pitchamdf** asig, imincps, imaxcps [, icps] [, imedi] [, idowns] [, iexcps]

DESCRIPTION
This opcode follows the pitch of signal asig based on the amdf method (Average Magnitude Difference Function) and outputs it to kcps. Additionally it outputs the energy of the signal to krms. The method is quite fast and should run in realtime. Techniques like that usually only work for monophonic
signals.

INITIALIZATION
imincps - estimated minimum frequency (expressed in cps) present in the signal.
imaxcps - estimated maximum frequency present in the signal.
icps - estimated initial frequency of the signal.
If 0, (imincps+imaxcps) / 2 is assumed. (Default = 0)
imedi - size of median filter applied to kcps output.
In fact, the resulting size of the filter will be imedi*2+1.
If 0, no median filtering will be applied. (Default = 1)
idowns - downsampling factor for asig. A factor of idowns>1 results in faster performance but may result in worse pitch detection.
Useful range is 1...4 (integer values). (Default = 1)
iexcps - how frequently pitch analysis is executed, expressed in cps.
If 0, iexcps is set to imincps which is usually reasonable,
but you can experiment with other values. (Default = 0)

PERFORMANCE
Pitch is detected quite reliably in monophonic signals if you select fitting init values. imincps and imaxcps should be as narrow as possible to the range of the signal's pitch - this results in better performance and better detection.
Setting icps close to the signal's real initial pitch prevents garbage at start, as the process can only detect pitch after some periods. The median filter prevents the pitch from jumping - experiment what size is best for the given signal.
The other init values can usually be left at their default.
It can be useful to lowpass-filter asig before giving it to pitchamdf.

EXAMPLE

```
asig       loscil     1, 1, input, 1       ; get input signal with original freq
asig       tone       asig, 1000           ; lowpass-filter
kcps, krms  pitchamdf  asig, 150, 500, 200  ; extract pitch and
                                                envelope
asig1      oscil      krms, kcps, iwave    ; "resynthesize" with some
                                                waveform
           out        asig1
```

==========
==John ff
1999 August 30
==========

Release Notes for 3.59
==========
These are the release notes for version 3.59

Bug Fixes

Fixed a typing error in fgens

MIDI file sin .csd files now work

Language Changes
---

a-rate^p-rate expressions allowed

Opcode Fixes
---

pluck: Error check for kcps exceeding sample rate

posc family: allow negative frequencies

Phasor: use double sinternally for better precision

poweroftwo -- also works at a-rate
logbasetwo -- also works at a-rate

repluck, nreverb, grain, cross2, nlfilt -- no longer change constants

linseg -- h-rate version rewritten to remove various bugs

tone, tonex, atone, atonex -- better intialisation

mxdsr, madsr -- new optional arguemnt to give release time.

linesegr, expsegr -- bugs corrected

vpvoc -- new optional argument to give a table for controsl rather than previous tableseg/tablexseg

slider* -- fixed so work

New Opcodes
---

phasorbnk -- bank of phasors

schedkwhen -- k-rate adding of score events

Other Changes:

Better treatment of score events

Windows GUI Changes

Correction in MIDI files selected

---

phasorbnk
---

kr **phasorbnk** kcps, kindx, icnt [, iphs]
ar p**hasorbnk** xcps, kindx, icnt [, iphs]

DESCRIPTION

This opcode works like the phasor opcode, except that there is an array of an arbitrary number of phasors that can be accessed by index.

INITIALIZATION
icnt - maximum number of phasors to be used.
iphs - initial phase if each phasor, if -1 initialization is skipped.
If > 1 all phases will be initialized with a random value.

PERFORMANCE
For each independent phasor an internal phase is successively accumulated in accordance with the cps frequency to produce a moving phase value, normalized to lie in the range 0 <= phs < 1.
Each individual phasor is accessed by index kindx.
This phasor bank can be used inside a k-time loop to generate multiple independent voices, or together with the adsynt opcode to change parameters in the tables used by adsynt.

EXAMPLE
Generate multiple voices with independent partials.
(In fact this example would better be done with adsynt.)

See also example for k-rate use of phasorbnk under adsynt.

```
giwave ftgen  1, 0, 1024, 10, 1          ; generate a sinewave table

      instr 1
icnt          =         10                ; generate 10 voices
asum          =         0                 ; empty output buffer
kindex        =         0                 ; reset loop index

loop:         ;loop executed every k-cycle

kcps          =         (kindex+1)*100 + 30     ; non-harmonic partials
aphas  phasorbnk  kcps, kindex, icnt      ; get phase for each voice
asig   table      aphas, giwave, 1        ; and read wave from table
asum   =          asum + asig             ; accumulate output

kindex        =         kindex + 1
if (kindex < icnt) kgoto loop             ; do loop

      out         asum*3000
      endin
```

==========
==John ff
1999 August 30
==========

Release Notes for 4.01
==========

These are the release notes for version 4.01. This is a set of small changes against version 4.00, which was only slightly different from the 3.59 (v4.0 beta) release. These notes incorporate all changes since v3.591

Bug Fixes

None

Language Changes

Csound no longer creates score.srt as a fixed file unless the option -t0 is given.

Opcode Fixes
---

wguide1 and wguide2 -- fixed code so both a-rate and k-rate variables can be used.

pvinterp and pvread -- now allow small frame sizes

space -- bug fixed which would lead to inconsistent results

schedule and schedwhen -- should now work if the event is in the future

New Opcodes

(opcode change)

resony -- now has a new optional argument (not at the end ***INCOMPATIBLE CHANGE**) which controls logarithmic or linear spread.
See new documentation below.

Other Changes:

Version number now printed in x.xx format

Windows GUI Changes

Changes to how often the screen is repainted; should maintain the graphs better.

New check box in Extras dialog for keeping score.srt

==John ff
1999 Thanksgiving Week (USA)
==========
resony

ar **resony** asig, kbf, kbw, inum, ksep [, isepmode, iscl, istor]

DESCRIPTION

A bank of second-order bandpass filters, connected in parallel.

INITIALIZATION

inum - number of filters.
isepmode - determines if the separation center frequencies of each filter must be generated in logarithmically (using octave as unit of measure) or linearly (using Hertz). Default value is 0, corresponding to logarithmic mode.

iscl - coded scaling factor for resonators. A value of 1 signifies a peak response factor of 1, i.e. all frequencies other than kcf are attenuated in accordance with the (normalized) response curve. A value of 2 raises the response factor so that its overall RMS value equals 1. (This intended equalization of input and output power assumes all frequencies are physically present; hence it is most applicable to white noise.) A zero value signifies no scaling of the signal, leaving that to some later adjustment (e.g. see balance). The default value is 0.

istor - initial disposition of internal data space. Since filtering incorporates a feedback loop of previous output, the initial status of the storage space used is significant. A zero value will clear the space; a non-zero value will allow previous information to remain. The default value is 0.

PERFORMANCE

asig - audio input signal
kbf - base frequency, i.e. center frequency of lowest filter in Hz
kbw - bandwidth in Hz
ksep - separation of the center frequency of filters (in octaves or in Hertz, depending by isepmode flag)

resony is a bank of second-order bandpass filters, with k-rate variant frequency separation, base frequency and bandwidth, connected in parallel (i.e. the resulting signal is a mixing of the output of each filter). The center frequency of each filter depends of kbf and ksep variables. The maximum number of filters is set to 100.

EXAMPLE:

In this example the global variable gk1 modifies kbf, gk2 modifies kbw, gk3 inum, gk4 ksep and gk5 the main volume.

```
          instr       1
a1        soundin "myfile.aif"
a2        resony      a1,   gk1 , gk2 ,i(gk3),gk4
          out         a2 * gk5
          endin
```

==========

Release Notes for 4.02
==========

These are the release notes for version 4.02. This is a set of small changes against version 4.01.

Bug Fixes

On Windows, Macintoshes and BeOS any silent section of audio was possibly too long by up to 127 k-cycles.

Coding error in tablew code fixed -- may have not given any errors. Similar error in Windows interface fixed.

Temporary files are removed (was OK on Unix)

^ in scores refers to previous event always

Language Changes

Gen23 treats negative numbers correctly, and is more forgiving in some cases

Opcode Fixes

All the slider code had potential problems (order of evaluation)

New Opcodes

NONE

Other Changes:

Slightly improved performance on Windows.

Windows GUI Changes

Improved code for play_at_end and similar. Fixes some longstanding oddities

Stupid coding errors in sound input fixed

==John ff
2000 Jan 29
==========

Release Notes for 4.03
==========

These are the release notes for version 4.03. This is a set of small changes against version 4.01/4.02.

Bug Fixes
---

pvl_main.c had a small coding error

Removed stdout and stdin as values in top-level assignment as some C systems do not allow it.

Coding error in getstring fixed

Language Changes

In loscil and loscil3 the base frequency of a sample defaults to middle C if it is missing from the sample and the opcode

The opcodes rand, randh and randi all accept an additional optional argument which is a base value added to the random result. This value can vary at k-rate.

Opcode Fixes
---

lfo in a-rate form was broken except for sinusoidal case.

New Opcodes
---

follow2 -- a different envelope extractor with controllable response to rise and fail

Other Changes:

The scale program can now take new arguments -M num or -P num which give a maximum value to which to scale or a maximum percentage of full range (32767 or 1.0 for floats). This uses two passes over the sound file.

On Windows machines the temporary files are made in the temporary directory, or SFDIR or HOME based on environment variables.

Experimentally I have arranged that an AIFF sample read which has no looping information is adjusted to be treated as a single loop the length of the sample.

WAV and AIFF files generated by Csound now contain a PEAK chunk.

Windows GUI Changes

==John ff
2000 Feb
==========

follow2

ar **follow2** asig, katt, krel

DESCRIPTION

A controllable envelope extractor using the algorithm attributed to Jean-Marc Jot.

PERFORMANCE

asig -- the input signal whose envelope is followed

katt -- the attack rate (60dB attack time in seconds)

krel -- the decay rate (60dB decay time in seconds)
The output tracks the amplitude envelope of the input signal. The rate at which the output grows to follow the signal is controlled by the attack rate, and the rate at which is decreases in response to a lower amplitude is controlled by the release rate. This gives a smoother envelope that the follow opcode at a little more expense.

EXAMPLE

a1 follow2 ain, 0.01, .1

JPff

==========

Release Notes for 4.05
==========

These are the release notes for version 4.05. There were no notes for 4.04 which was only released for Linux. This version contains two new families of opcodes, and some significant fixes.

Bug Fixes

Calculation of kr (if omitted) was wrong

On some systems (notable recent Linux) the double closing of the file scfp let to crashes.

Temporary files are cleaned up in more circumstances

Problems with large numbers of labels fixed

Language Changes

Added a new option, -Z, which switches on dithering of audio conversion from internal fpt to 32bit, 16bit and 8bit formats. This is not properly tested

Opcode Fixes
---

atone and atonex failed if the input and output were the same variable

Simpler tests in midiops3 family

New Opcodes

Added two opcodes for scanned synthesis (Interval's copyright):
scanu
scans

Added family of Sound Font opcodes:
sfload, sfpreset, sfplay, sfplaym,
sfplist, sfilist, sfpassign, sfinstrm, sfinstr

Other Changes:

Integration of BeOs makefiles and audio

Windows GUI Changes

None

==John ff
2000 March
==========

SoundFont2-related opcodes
ifilhandle **sfload** "filename"
sfplist ifilhandle
sfilist ifilhandle
sfpassign istartindex, ifilhandle
ipreindex **sfpreset** iprog, ibank, ifilhandle, ipreindex

a1, a2 **sfplay** ivel, inotnum, xamp, xfreq, ipreindex [, iflag]
a1 **sfplaym** ivel, inotnum, xamp, xfreq, ipreindex [, iflag]

a1,a2 **sfinstr** ivel, inotnum, xamp, xfreq, instrNum, ifilhandle [, iflag]
a1 **sfinstrm** ivel, inotnum, xamp, xfreq, instrNum, ifilhandle[, iflag]

DESCRIPTION

Csound now supports SoundFont2 format. These opcodes allow to manage the sample-structure of SoundFont2 files.

INITIALIZATION

filename - name of the SoundFont2 file (complete pathname). You must use "/" to separate directories even under Windows. It must be typed within double-quotes.
ifilhandle - unique number generated by sfload opcode to be used as an identifier of a SoundFont2 file, since several SoundFont2 files can be loaded and activated at the same time.
istartindex - starting preset index set by the user in bulk preset assignments (see below).
ipreindex - preset index
iprog - program number of a bank of presets of a SoundFont2 file
ibank - number of a specific bank of a SoundFont2 file
ivel - velocity value
inotnum - note number value
iflag - flag regarding the behaviour of xfreq (see below).
instrNum - number of an instrument of a SoundFont2 file.

PERFORMANCE

xamp - amplitude correction factor
xfreq - frequency value or frequency correction factor (depending by iflag, see below)

SoundFont2 is a widespread standard which allow to embed banks of wavetable-based sounds into a binary file. In order to understand the usage of these opcodes, the user must know some notion about SF2 format. So a brief description of this format follows.

The SoundFont2 format is made by generator and modulator objects. All current Csound opcodes regarding SF2 support generator section only, so we will only deal with the generator-related structure of SF2 format, omitting the modulators.

There are several levels of generators having a hierarchical structure. The most basic kind of generator object is a sample. Samples can or can't be be looped and are associated to a MIDI note number, called base-key. When a sample is associated with a range of MIDI note numbers, with a range of velocities, with a transposition (coarse and fine tuning), with a scale tuning, and with a level scaling factor, such sample makes up a split. A set of splits, together with a name, makes up an instrument. When an instrument is associated with a key range, with a velocity range, with a level scaling factor, and with a transposition, it makes up a layer. A set of layers, together with a name, makes up a preset. Presets are normally the final sound-generating structures ready for the user. They generate sound according to the settings of their lower-level components.

Both sample data and structure data is embedded in the same SoundFont2 binary file. A single SF2 file can contain up to a maximum of 128 banks of 128 preset programs, for a total of 16384 presets each one. Maximum number of layers, instruments, splits and samples is not defined, and probably is only limited by the computer memory.

sfload opcode loads an entire SF2 file in memory. It returns a file handle to be used by other opcodes. Several instances of sfload can placed in the header section of an orchestra, allowing to work with more-than-one SF2 files at the same time.

sfplist prints a list of all presets of a previously loaded SF2 file to the console.

sfilist prints a list of all instruments of a previously loaded SF2 file to the console.
sfpassign assigns all presets of a previously loaded SF2 file to a sequence of progressive index numbers, to be used later with the opcodes sfplay and sfplaym. The user can establish the first index number by setting startindex argument. Any number of sfpassign instances can be placed in the header section of an orchestra, each one assigning presets belonging to different SF2 files. The user must

take care that preset index numbers of different SF2 files don't cross themselves.

sfpreset assigns an existing preset of a previously-loaded SF2 file to an index number, to be used later with the opcodes sfplay and sfplaym. The user must previously know the program and the bank numbers of the preset in order to fill the corresponding arguments. Any number of sfpreset instances can be placed in the header section of an orchestra, each one assigning a different preset belonging to the same (or different) SF2 file to different index numbers.

sfplay plays a preset generating a stereo sound.
ivel argument does not directly affect the amplitude of the output, but inform sfplay opcode about what sample has to be chosen in multi-sample velocity-splitted presets. inotnum argument sets the frequency of the output when iflag = 0. When iflag == 1, inotnum doesn't directly affect the frequency of the output (see below). Adjustment of amplitude can be done by varying the xamp argument, that actually is a multiplier factor. xfreq argument have a two different behaviour depending by the value of iflag argument:

when iflag = 0 (or missing as this value is the default)
xfreq argument is a multiplier of a the default frequency assigned by SF2 preset to the inotenum value. This can correct the default frequency (for example to obtain vibrato or some other frequency -shift effect).

when iflag = 1 xfreq argument should contain the actual frequency of the output sound in cps. This allow the user to use any kind of micro-tuning based scales. However this flag is designed to work correctly only with presets tuned to the default equal temperament. Don't try to use this flag value with preset already having non-standard tunings or with drum-kit-based presets, since unexpected results could occur.

Notice that both xamp and xfreq arguments can contain k-rate signals as well as a-rate signals, but the user must be sure that both rguments are filled with variables of the same rate, or sfplay will ot work correctly. The user must be sure that ipreindex argument is illed with a number containing a previously assigned preset, therwise Csound will crash.

sfplaym opcode is a mono version of sfplay. It should be used with ono preset, or with the stereo presets in which stereo output is not equired, because is a bit faster than sfplay.

sfinstr plays an SF2 instrument instead of a preset (an SF2 instrument is the base of a preset layer). instrnum argument contains the nstrument number, and the user must be sure that such number belongs o an existent instrument of a determinate soundfont bank. Notice that oth xamp and xfreq arguments can contain k-rate signals as well as -rate signals, but, also in this case, the user must be sure that oth arguments are filled with variables of the same rate, or sfinstr will not work correctly.

sfinstrm plays is a mono version of sfinstr. This is the fastest opcode of the SF2 family.
These Csound opcodes only handle sampling structure of SF2 files, because support of modulator objects (amplitude envelopes, frequency modulation, filter envelopes and modulation) is very basic and trivial in SF2 standard; so, adding any kind of modulation or processing to the sample data is completely left to the Csound user, bypassing all restrictions forced by the SF2 standard.

Gabriel Maldonado

---

scanu iinit, irate, ivel, im, if, ic, id, km, kf, kc, kd, il, ir, kx, ky, ain, idisp, iid

iinit: the initial position of the masses. If this is a negative number, then the absolute of iinit signifies the table to use as a hammer shape. If iinit > 0, the length of it should be the same as the intended mass number, otherwise it can be anything.

irate: the amount of time between successive updates of the mass state. Kind of like the sample period of the system. If the number is big the string will update at a slow rate showing little timbral variability, otherwise it will change rapidly resulting in a more dynamic sound.

ivel: The number of the ftable that contains the initial velocity for each mass. It should have the same size as the indented mass number.

im: The number of the ftable that contains the mass of each mass. It should have the same size as the indented mass number.

if: The number of the ftable that contains the spring stiffness of each connection. It should have the same size as the square of the indented mass number. The data ordering is a row after row dump of the connection matrix of the system.

ic: The number of the ftable that contains the centering force of each mass. It should have the same size as the indented mass number.

id: The number of the ftable that contains the damping factor of each mass. It should have the same size as the indented mass number.

km: A parameter that scales the masses.

kf: a parameter that scales the spring stiffness.

kc; a parameter that scales the centering force.

kd: a parameter that scales damping.

il: If iinit < 0, the position of the left hammer (il = 0 is hit at leftmost, il = 1 is hit at rightmost).

ir: If iinit < 0, the position of the right hammer (ir = 0 is hit at leftmost, ir = 1 is hit at rightmost).

ix: This is the position of an active hammer along the string (0 leftmost,1 rightmost). The shape of the hammer is determined by iinit and the power it pushes with is iy.

iy: The power that the active hammer uses.

ain: The audio input that adds to the velocity of the masses (don't make it too loud).

idisp: If 0, no display of the masses is provided. Otherwise you get to see them wiggle.

iid: For scanu: the ID of the opcode. This will be used to point the scanning opcode to the proper waveform maker. If this value is negative it is minus the wavetable on which to write the waveshape.
That wavetable can be used later from an other opcode to generate sound. The initial contents of this table will be destroyed, so do not rely on them being there.

The syntax for scans is:

scans kamp, kfreq, itrj, iid

kamp: The output amplitude. Note that the resulting amplitude is also dependent to the state of the wavetable. This number is effectively the scaling factor of the wavetable.

kfreq: The frequency of the scan rate.

itraj: The number of the ftable that contains the scanning trajectory.
This is a series of numbers that contains addresses of masses. The order of these addresses is used as the scan path. It shouldn't contain values more that the number of masses, as well as negative numbers.

iid: The ID number of the scanu opcode's waveform to use. To produce the matrices, the file format is straightforward. For example for 4 masses we have the following grid describing the connections:

	1	2	3	4
1				
2				
3				
4				

Whenever two masses are connected then the point they define is 1, so for a unidirectional string we would have the following connections, (1,2), (2,3), (3,4) (if it was bidirectional we would also have (2,1), (3,2), (4,3)). So I fill these out with ones and the rest with zeros and I get:

	1	2	3	4
1	0	1	0	0
2	0	0	1	0
3	0	0	0	1
4	0	0	0	0

Similarly for the other shapes, I find the connections and fill them out. This gets saved in an ASCII file column by column, so the string up there would be saved as:

0.
1.
0.
0.
0.
0.
1.
0.
0.
0.
0.
1.
0.
0.
0.
0.

Paris Smaragdis

Release Notes for 4.06

==========

These are the release notes for version 4.06. A lot has changed, and in places my notes are less than explicit. Major change is in multiple channel audio. The maximum number of channels is increased to 256, and there are opcodes for reading and writing m any channels.
Related there are the VBAP family of opcodes which allow for positioning and moving of sound between members of an array of speakers.

I have been playing with Tcl/Tk having had to teach it this last term, and I have a set of on-screen sliders which can control an instrument, not through MIDI. As this is a first attempt there may be opportunities for better versions. The interface is such that any Python fans, or indeed any other system could be used instead. At present it assumes the existence of wish and the TK sources are hardwired. This will change when i have thought about it.

Bug Fixes

In reading scores it was possible to get an overflow condition which gave really odd errors.

Language Changes

In scansys opcodes it is now possible to select the interpolation order with a new optional i-rate argument. The default is 4 (as it was previously) but there are reports that cubic (3) or quadratic (2) sounds better, and is certainly faster.

Maximum number of audio channels in now 256

File names in FGENS 23 and 28 are now expanded relative to a number of directories.

Opcode Fixes

Bug in ADSR fixed.

vpvoc now checks things more carefully

schedule now behaves with negative triggers

New Opcodes

---

outx, out32 and outch, outz for multi-channel output

inx, in32 and inc, inz similar for input

vbap family of opcodes added (vbap4, vbap8, vbap16, vbapz, vbaplsinit,
vbap4move, vbap8move, vbap16move, vbapzmove)

control, setcntrl now available for UNIX, and any operating system with Tcl/Tk (perhaps)

pinkish to generate pink noise

seqtime, trigseq -- Handle timed-sequences of groups of values stored into tables.

Other Changes:

OS2 code now incorporated into sources

Soundfont code reworked

There is a small Tcl/Tk program to build matrices for the scanned synthesis opcodes -- matrix.tk

GUI Changes

In Windows, Heartbeat option 3 writes information to title bar

In Unix implementations there are now on-screen sliders for real-time control of Csound, using the control opcode.

==John ff
2000 June 10

==========

ar **pinkish** xin[, imethod, inumbands, iseed, iskip]

Generates approximately pink noise (-3dB/oct response) by two different methods: multirate noise generator due to Moore, coded by Martin Gardner, or a filter bank designed by Paul Kellet.

PERFORMANCE
ar - pink noise.
xin - For Gardner method: k- or a-rate amplitude.
For Kellet filters: normally a-rate uniform random noise from rand (31-bit) or unirand, but can be any a-rate signal.
The output peak value varies widely (15%) even over long runs, and will usually be well below the input amplitude. Peak

values may also occasionally overshoot input amp/noise.
imethod - (optional) selects filter method.
=0 Gardner method (default).
=1 Kellet filter bank.
=2 A somewhat faster filter bank by Kellet, with less accurate response.
inumbands - (optional) only effective with Gardner method. The number of noise bands to generate. Maximum is 32, minimum is 4. Higher levels give smoother spectrum, but above 20 bands there will be almost DC-like slow fluctuations. Default value is 20.
iseed - (optional) only effective with Gardner method. If non-zero, seeds the random generator. If zero, the generator will be seeded from current time. Default is 0.
iskip - (optional) if non-zero, skip (re)initialisation of internal state (useful for tied notes). Default is 0.

pinkish attempts to generates pink noise (ie noise with equal energy in each octave), by either of two different methods.

The first method, by Moore/Gardner, adds several (up to 32) signals of white noise, generated at octave rates (sr, sr/2, sr/4 etc). It gets pseudo-random values from an internal 32-bit generator, which is local to each opcode instance and seedable (similarly to rand).

The second method is a lowpass filter with hardcoded response approximating -3dB/oct. If input is uniform white noise, it outputs pink noise. Any signal may be used as input for this method. The high quality filter is slower, but has less ripple and slightly wider operating frequency range than the "economy" version. With the Kellet filters, seeding is not used.

The Gardner method output has some bumps and dips in the low-mid and mid-high frequency ranges. It can be set to generate more low-frequency energy by increasing the number of bands. It is also a bit faster. The Kellet filter (refined) has very smooth spectrum, but a more limited effective range, and the level increases slightly at the high end of the spectrum.

EXAMPLE
Kellet-filtered noise for a tied note (iskip is non-zero).

```
  awhite unirand 2.0
  awhite = awhite - 1.0       ; Normalize to +/-1.0
  apink pinkish awhite, 1, 0, 0, 1
    out apink * 30000
```

---

outx a1, a2, a3, a4, a5, a6, a7, a8, a9, aa, ab, ac, ad, ae, af
out32 a1, a2, a3, a4, a5, a6, a7, a8, a9, aa, ab, ac, ad, ae, af, ag, ah, ai, aj, ak, al, am, an, ao, ap, aq, ar, as, at, au
outc a1[, a2,....]
outch k1, a1, k2, a2, ...
outz k1

outx and out32 output 16 and 32 channels of audio.
outc outputs as many channels as provided. Any channels greater than nchnls are ignores, and zeros are added as necessary outch outputs a1 on channel k1, a2 on channel k2 and so on.
outz outputs from a ZAK array, for nchnls of audio

---

a1, a2, a3, a4, a5, a6, a7, a8, a9, aa, ab, ac, ad, ae, af **inx** a1, a2, a3, a4, a5, a6, a7, a8, a9 , aa, ab, ac, ad, ae, af, ag, ah, ai, aj, ak, al, am, an, ao, ap, aq, ar, as, at, au **in32**
a1 **inch** k1
inz k1

inx and in32 read 16 and 32 channel inputs
inch reads from a numbered channel k1 into a1
inz reads audio samples in nchnls into a ZA array starting at k1

---

vbaplsinit, vbap4, vbap8, vbap16
vbap4move, vbap8move, vbap16move

vbaplsinit idim, ils_amount, idir1, idir2,...

a1, a2, a3, a4 **vbap4** asig, iazi,iele, ispread
a1, a2, a3, a4, a5, a6, a7, a8 **vbap8** asig, iazi,iele, ispread

a1, a2, a3, a4, a5, a6, a7, a8, a9, a10, a11, a12, a13, a14, a15, a16 **vbap16** asig, iazi,iele, ispread

a1, a2, a3, a4 **vbap4move** asig, ispread, ifld_amount, ifld1, ifld2, ...

a1, a2, a3, a4, a5, a6, a7, a8 **vbap8move** asig, ispread, ifld_amount, ifld1, ifld2, ...

a1, a2, a3, a4, a5, a6, a7, a8, a9, a10, a11, a12, a13, a14, a15, a16 **vbap16move** asig, ispread, ifld_amount, ifld1, ifld2, ...

Distribute an audio signal amongst 2 to 16 channels with localization control.

INITIALIZATION

idim - dimensionality, 2 or 3.

ils_amount - number of loudspeakers. In two dimensions the number can vary between two to 16. In three dimensions the number can vary between three and 16.

idirn - directions of loudspeakers, number of directions must be less or equal than 16. In two-dimensional loudspeaker positioning idirn is the azimuth angle respective to nth channel. In three-dimensional loudspeaker positioning fields are the azimuth and elevation angles of each loudspeaker consequently (azi1, ele1, azi2, ele2,...).

asig - audio signal to be panned.

iazi - azimuth angle of the virtual source.

iele - elevation angle of the virtual source

ispread - spreading of the virtual source (range 0 - 100). If value is zero, conventional amplitude panning is used. When value is increased, the amount of loudspeakers used in panning gets larger. If value is 100, the sound is applied to all loudspeakers.

ifld_amount - number of fields (absolute value must be 2 or larger). If ifld_amount is positive, the virtual source movement is a polyline specified by given directions, each transition is performed in an equal time interval. If ifld_amount is negative, specified angular velocities are applied to the virtual source during specified relative time intervals (see below).

ifldn - Azimuth angles or angular velocities, and relative durations of movement phases (see below).

PERFORMANCE

vbap4, vbap8 and vbap16 take an input signal asig and distribute it amongst at two to 16 outputs according to the controls iazi and iele and configured loudspeaker placement. If idim = 2, iele is set to zero. The distribution is performed using Vector Base Amplitude Panning (VBAP) [1]. VBAP distributes the signal using loudspeaker data configured with vbaplsinit. The signal is applied at most to two loudspeakers in 2-D loudspeaker configurations and to three loudspeakers in 3-D loudspeaker configurations. If the virtual source is panned outside the region spanned by loudspeakers, nearmost loudspeakers are used in panning.

vbap4move, vbap8move and vbap16move allow moving virtual sources to be applied. If ifld_amount is positive, the fields represent directions of virtual sources and equal times, iazi1, [iele1,] iazi2, [iele2,]....
The position of the virtual source is interpolated between directions starting from first direction and ending to last. Each interval is interpolated in time that is fraction total_time / number_of_intervals of the duration of the sound event.

If ifld_amount is negative, the fields represent angular velocities and equal times. The first field is however the starting direction, iazi1, [iele1,] iazi_vel1, [iele_vel1,] iazi_vel2, [iele_vel2,]....
Each velocity is applied to the note that is fraction total_time/ number_of_velocities of the duration of the sound event. If the elevation of the virtual source becomes greater than 90 degrees or less than 0 degrees, the polarity of angular velocity is changed. Thus the elevational angular velocity produces a virtual source that moves up and down between 0 and 90 degrees.

EXAMPLE

```
2-D panning example with stationary virtual sources
        sr = 44100
        kr = 441
        ksmps = 100
        nchnls = 4
        vbaplsinit 2, 6,  0, 45, 90, 135, 200, 315,
        instr     1                                     ;parameter
                asig oscil 20000, 440, 1              ; p4 = azimuth
a1, a2, a3, a4, a5, a6, a7, a8          vbap8         asig, p4, 0, 20
                outq     a1,a2,a3,a4
;       outq     a5,a6,a7,a8
        endin
```

References
[1] Ville Pulkki: Virtual Sound Source Positioning Using Vector Base Amplitude Panning. Journal of the Audio Engineering Society, 1997 June, Vol. 45/6, p. 456.

Implementation by Ville Pulkki
Sibelius Academy Computer Music Studio
Laboratory of Acoustics and Audio Signal Processing
Helsinki University of Technology
May 2000

The opcode vbapz and vbabzmove are the multiple channel analogs of the above opcodes, working an nchnls and using a ZAK array for output.
The limit on the number of channels is 256.

(Coded by JPff from material of Ville Pulkii)

Sequence-related opcodes (seqtime and trigseq)

ktrig_out **seqtime** ktime_unit, kstart, kloop, initndx, kfn_times
trigseq ktrig_in, kstart, kloop, initndx, kfn_values, kout1 [, kout2, kout3, ..., koutN]

DESCRIPTION

Handle timed-sequences of groups of values stored into tables.

INITIALIZATION

initndx - initial index

PERFORMANCE

ktrig_out - output trigger signal
ktime_unit - unit of measure of time, related to seconds.
kstart - start index of looped section
kloop - end index of looped section
kfn_times - number of table containing a sequence of times
kfn_values - numer of a table containing a sequence of groups of values
ktrig_in - input tirgger signal
kout1 [, kout2, kout3, ..., koutN] - output values

These opcodes handle timed-sequences of groups of values stored into tables.

seqtime generates a trigger signal (a sequence of impulses, see also trigger opcode), according to the values stored in kfn_times table. This table should contain a series of delta-times (i.e. times beetween to adiacent events). The time units stored into table are expressed in seconds, but can be rescaled by means of ktime_unit argument. The table can be filled with GEN02 or by means of an external text-file containing numbers, with GEN23. It is possible to start the sequence from a value different than the first, by assigning to initndx an index different than zero (which corresponds to the first value of the table). Normally the sequence is looped, and the start and end of loop can be adjusted by modifying kstart and kloop arguments. User must be sure that values of these arguments (as well as initndx) correspond to valid table numbers, otherwise Csound will crash (because no range-checking is implementeted). It is possible to disable loop (one-shot mode) by assigning the same value both to kstart and kloop arguments. In this case, the last read element will be the one corresponding to the value of such arguments. Table can be read backward by assigning a negative kloop value. It is possible to trigger two events almost at the same time (actually separated by a k-cycle) by giving a zero value to the corresponding delta-time.
First element contained in the table should be zero, if the user intend to send a trigger impulse it immediately after the orchestra instrument containing seqtime opcode

trigseq accepts a trigger signal (ktrig_in) as input and outputs group of values (contained into kfn_values table) each time ktrig_in assumes a non-zero value. Each time a group of values is triggered, table pointer is advanced of a number of positions corresponding to the number of group-elements, in order to point to the next group of values. The number of elements of groups is determined by the number of koutX arguments. It is possible to start the sequence from a value different than the first, by assigning to initndx an index different than zero (which corresponds to the first value of the table). Normally the sequence is looped, and the start and end of loop can be adjusted by modifying kstart and kloop arguments. User must be sure that values of these arguments (as well as initndx) correspond to valid table numbers, otherwise Csound will crash (because no range-checking is implementeted). It is possible to disable loop (one-shot mode) by assigning the same value both to kstart and kloop arguments. In this case, the last read element will be the one corresponding to the value of such arguments. Table can be read backward by assigning a negative kloop value.

trigseq is designed to be used together with seqtime or trigger opcodes.

Example:

```
                instr       1
icps            cpsmidi
iamp            ampmidi     5000
ktrig           seqtime     1,   1,  10,  0, 1
trigseq         ktrig, 0, 10, 0, 2, kdur, kampratio, kfreqratio
schedkwhen ktrig, -1, -1, 2, 0, kdur, kampratio*iamp, kfreqratio*icps
                endin

                instr  2
**** put here your intrument code *******
                out         a1
                endin
```

Release Notes for 4.07
==========

These are the release notes for version 4.07. Note that there are four new files in the sources, bowedbar.c, bowedbar.h, phisem.c and phisem.h.

Bug Fixes

Error in message in extract functions fixed

Typing error in fileopen fixed

Fixed bad message in AIFF headers

Minor fix in WAV format files used for input

Initial value in midi controllers changed in one case

Language Changes

New tags added to .csd files to allow for Base64 encoded MIDI files <CsMidifileB filename=...>, and for Base64 encoded samples <CsSampleB filename=...>.

Macro names can now include _ as a character

Exponential format numbers in scores allowed (finishes earlier attempt)
Opcode Fixes

Minor bug in bowed fixed related to length of delay line

The physical model opcodes have been revised in line with P.Cook's STK3.1. This effects filter values in marimba, gogobel in particular. Strike position on vibraphone now used, and in gogobell.

The reverb and nreverb opcodes could have a zero delay time, which gives rise to an infinite gain. Attempts to set non-positive delay has the value changed to 0.01s

New Opcodes
---

clip -- apply soft clipping to a signal using a variety of algorithms. Current version has two working algorithms

wgbowedbar -- physical model of a bowed bar

PhISem family of opcodes: cabasa, crunch, sekere, sandpaper, stix, guiro, tambourine, bamboo, dripwater, sleighbells. These are all percussion sounds.

Other Changes:

Some support for OS/2 in sources

Some support for rpm format distribution
In vbap a large array has been moved off stack, which should help platforms with stack limitations,

Windows GUI Changes

Phase Vocoder dialog had its check for illegal hopsize all wrong

PVLook dialog extended to allow a log file

==John ff
2000 August
==========

ar **clip** ain, imethod, ilimit[, iarg]

Clips an input audio signal to a limit in a 'soft' fashion rather than a straight cutoff. There are three methods at present, and the argument is used in each case to control the abruptness of the clip.

PERFORMANCE

ar - clipped audio
ain - an input a-rate signal
imethod - selects clipping method.
=0 Bram de Jong method (default).
=1 sine clipping
=2 tanh clipping

iarg (optional)-- Method 0 in the range 0 to 1 indicating the fraction at which the clipping starts. Default value is 0.5. This argument is not used in methods 1 or 2

The first method, by Bram de Jong, applies the algorithm (assuming a
signal normalised to 1).
|x| > a: f(x) = sign(x) * (a + (x-a)/(1+((x-a)/(1-a))^2))
|x| > 1: f(x) = sign(x)*(a+1)/2

The second method is a sine clip:
|x|<limit f(x) = limit * sin(pi x/(2*limit)
f(x) = limit * sign(x)

The third method is a tanh clip:
|x|<limit f(x) = limit * tanh(x/limit)/tanh(1)
f(x) = limit * sign(x)

Note: Method 1 seems to be non-functional

EXAMPLE

```
a1    in
a2    oscil   25000, 1
asig  clip    a1+a2, 0, 30000, 0.75
      out     asig
```

---
ar **wgbowedbar** kamp, kfreq, kpos, kbowpres, kgain[, kconst, ktVel, ibowpos, ilow]

A physical model of a bowed bar, belonging to the Perry Cook family of waveguide instruments.

kamp -- amplitude of signal

kfreq -- frequency of signal

kpos -- where on bar the bow is used in the range 0 to 1

kbowpres -- pressure of the bow 9as in wgbowed)
kgain -- gain of filter; suggested to have values about 0.809.

kconst -- an integration constant, defaulting to zero.

ktVel -- either 0 or 1; with zero the bow velocity follows an ADSR style trajectory; when 1 the value of the bow velocity decays in an exponential way.

kbowpos -- the position on the bow, which affects the bow velocity trajectory.

ilow -- lowest frequency required

Example

```
        instr     1
;;;     pos = [0, 1]
;;;     bowpress = [1, 10]
;;;     GAIN = [0.8, 1]
;;;     intr = [0,1]
;;;     trackvel = {0, 1}
;;;     bowpos = [0, 1]

;; amp,freq, pos,bowPr, GAIN, int,trackVel,bowpos,lowest
Freq;
kb      line      0.5, p3, 0.1
kp      line      0.6, p3, 0.7
kc      line      1, p3, 1
a1        wgbowedbar  p4, cpspch(p5), kb, kp,   0.995, p6, 0,      kc,
50
        out       a1
        endin

i1 0   3 32000 7.00 0
```

PhiSem::

ar **cabasa** iamp, idettack[, knum, kdamp, kmaxshake]

ar **crunch** iamp, idettack[, knum, kdamp, kmaxshake]
ar **sekere** iamp, idettack[, knum, kdamp, kmaxshake]
ar **sandpaper** iamp, idettack[, knum, kdamp, kmaxshake]
ar **stix** iamp, idettack[, knum, kdamp, kmaxshake]
ar **guiro** iamp, idettack[, knum, kdamp, kmaxshake, kfreq, kfreq1]
ar **tambourine** iamp, idettack[, knum, kdamp, kmaxshake, kfreq, kfreq1, kfreq2]
ar **bamboo** iamp, idettack[, knum, kdamp, kmaxshake, kfreq, kfreq1, kfreq2]
ar **dripwater** iamp, idettack[, knum, kdamp, kmaxshake, kfreq, kfreq1, kfreq2]
ar **sleighbells** iamp, idettack[, knum, kdamp, kmaxshake, kfreq, kfreq1, kfreq2]

Semi-physical models of various percussion sounds.

iamp -- Amplitude of output. Note that as these instruments are stochastic, this is only a rough guide.

idettack -- period of time over which all sound is stopped

knum -- The number of beads, teeth, bells, timbrels etc. If zero the default value is used

cabasa	512
crunch	7
sekere	64
sandpaper	128
stix	30
guiro	128
tambourine	32
bamboo	1.25
dripwater	10
sleighbells	32

kdamp -- the damping factor of the instrument. The value is used as an adjustment close to the defaults below, with 1 being no damping. If zero the default values are:

cabasa	0.997
crunch	0.99806
sekere	0.999
sandpaper	0.999
stix	0.998
guiro	1.0
tambourine	0.9985
bamboo	0.9999
dripwater	0.995
sleighbells	0.9994

kmaxshake -- amount of energy to add back into the system, in range 0 to 1.

kfreq -- Setting the main resonant frequency; default values are:

guiro	2500
tambourine	2300
bamboo	2800
dripwater	450
sleighbells	2500

kfreq1 -- setting the first resonant frequency; defaults are

guiro	
tambourine	5600
bamboo	2240
dripwater	600
sleighbells	5300

kfreq2 -- setting the second resonant frequency; defaults are

tambourine	8100
bamboo	3360
dripwater	750
sleighbells	6500

Examples

```
asig cabasa  p4, 0.01, 0, 0, 0
asig sekere  p4, 0.01, 0, 0, 0
asig sandpaper  p4, 0.01, 0, 0, 0
asig stix  p4, 0.01, 0, 0, 0
asig tambourine p4, 0.01
asig bamboo p4, 0.01
asig dripwater p4, 0.01
asig sleighbells p4, 0.01
```

Release Notes for 4.08
==========

These are the release notes for version 4.08. Note that there are new files in the sources, sdif.c, sdif.h, sdif-mem.c, sdif-mem.h and sdif2adsyn.c

This release is mainly a number of bug fixes, but there are a couple of new opcodes, and a major internal reorganisation to allow creation of a double-based Csound.

Bug Fixes

Bug in score macros fixed

Dithering message was overlaid with a Scansys message

Language Changes
---

hetro can generate SDIF files, and a new utility can translate SDIF to adsyn

Opcode Fixes
---

guiro had an argument missing which was dangerous

The damp parameter of guiro was documented as being the damping, but it was not in the code. It is now, and should have a value less than 1.

New Opcodes
---

mpulse -- generate a stream of impulses
button -- buttonpush control
checkbox -- checkbox control

Other Changes:

Internal changes to make FreeBSD build easier

There has been a major source change so it is now possible to build Csound using doubles rather than floats internally (*). This is (in general) slower and bigger, but more accurate. We have only tested on Windows and Linux so far. Could be other on other platforms. Note that this changed nearly every file as the previous attempt fell over a Windows/Micro$oft special.

(*) In fact I will distribute 32 and 64 bit builds

Windows GUI Changes

On screen controls for buttons and checks may work.

==John ff
2000 August
==========

SDIF support in Csound.

For detailed information on the Sound Description Interchange Format, refer to the CNMAT website:

http://cnmat.CNMAT.Berkeley.EDU/SDIF

If the filename passed to HETRO has the extension .sdif, data will be written in SDIF format as 1TRC frames of additive synthesis data. The accompanying utility program "sdif2ads" can be used to convert any SDIF file containing a stream of 1TRC data to the Csound 'adsyn' format. 'sdif2ads' allows the user to limit the number of partials retained, and to apply an amplitude scaling factor. This is often necessary, as the SDIF specification does not, to date, require amplitudes to be within a particular range. 'sdif2ads' reports information about the file to the console, including the frequency range.

The main advantages of SDIF over the adsyn format, for Csound users, is that SDIF files are fully portable across platforms (data is 'big-endian'), and do not have the duration limit of 32.76 seconds imposed by the 16bit adsyn format.This limit is necessarily imposed by 'sdif2ads'. It is planned to incorporate sdif reading directly into adsyn, thus enabling files of any length (currently memory-permitting) to be analysed and processed.

It is important to note that the SDIF formats are still under development, and that while the 1TRC format is now fairly well established, it can still change.

Some other SDIF resources (including a viewer) are available via the NC_DREAM website:

http://www.bath.ac.uk/~masjpf/NCD/dreamhome.html

Richard Dobson 5th August 2000
rwd@cableinet.co.uk

---

aout **mpulse** kamp, kfreq[, ioffset]

Generate a set of impulses of amplitude kamp at frequency kfreq. The first impulse is after a delay of ioffset seconds (defaulting to zero). The value of kfreq is read only after an impulse, so it is the interval to the next impulse at the time of an impulse.

INITIALISATION

ioffset -- defaults to zero, is the delay before the first impulse.
If it is negative the value is taken as the negation of the number of samples; otherwise it is in seconds.

PERFORMANCE

kamp -- amplitude of the impulses generated

kfreq -- frequency of the impulse train

After the initial delay an impulse of kamp amplitude is generated as a single sample. Immediately after generating the impulse the time of the next one is calculated. If kfreq is zero there is an infinite wait to the next impulse. If kfreq is negative the frequency is counted in samples rather than seconds.

Example:

Generate a set of impulses at 10 a second, after a delay of 0.05s

```
instr 1
      a1 mpulse 32000, 0.1, 0.05
        out a1
endin
```

JPff: 2000 Sept 16

---

kans **button** inum
kans **checkbox** inum

Sense on-screen controls (cf control opcode) [Needs Windows or TCL/TK]

INITIALISATION

inum -- the number of the button or checkbox. If it does not exist it is made on-screen at initialisation time.

PERFORMANCE

If the button has been pushed since the last k-period then return 1; otherwise return 0
If the checkbox is set (pushed) then return 1; if it is not pushed return 0

Example:
increase pitch while a checkbox is set, and extend duration for each push of a button

```
instr 1
      kcps    =       cpsoct(p5)
      k1      check   1
      if (k1 == 1) kcps = kcps * 1.1
      a1      oscil   p4, kcps, 1
              out     a1
      k2      button  1
      if (k2 == 1) p3 = p3 + 0.1
endin
```

JPff: 2000 Sept 16

Release Notes for 4.09
==========

Bug Fixes

Some internal strings had been changed by mistake, confusing some operating systems.

On Windows there was a bug in sfont stuff, now fixed

Language Changes

It is more likely that // and /* */ comments will work in .csd files
Peak chunks can be switched off with a -K option

There is a new form in a .csd file which allows version checking. One can police whether the version of Csound can run a particular piece.
Format is
<CsVersion>
Before #.# or After #.# or #.#
</CsVersion>
The last two forms are equivalent

There was a simple but devastating bug in reading AIFF files

Opcode Fixes

The 31-bit pseudo random number generators are now bipolar as they should have been.
The sliders can now have text labels, which can be set with setctrl opcode, which is extended to allow case 4 (label)

In sfont opcode there is a filter to stop teh printing of unprintable characters which was upsetting xterms on some unixes.

There was a bug in expseg which I had never seen but could occur if a structure was reused internally.

There was a fence-post problem in looping oscilators.

The whole of wgpluck has been reworked. The bug whereby teh first use was quiet has been fixed, and the excitation of the string moved to the correct place (it used to be added to outout of string not theinput). The loop filter has been reworked, for simpler and shorter

code, but i am still not convinced that it is correct. It is at least no worse.

New Opcodes
---

babo -- Ball in a Box resonator (note copyright on this)

sense -- Check is a (computer) key has been pressed [Unix only at present]

transeg -- a mixed linear and exponential envelope opcode, rather like in cmusic.

GEN16 -- new gen to do the same as transeg

Other Changes:
---

vreverb revised significantly to allow a more flexible structure

The entry table has been split into two to make it more manageable, and in particular to allow a M68K system to be created.
Incorporation of BeOS patches

Windows GUI Changes

==John ff
2000 October
==========

en ar **nreverb** asig, krvt, khdif [, iskip] [,inumCombs, ifnCombs] [,inumAlpas, ifnAlpas]

INITIALIZATION

inumCombs - number of filter constants in comb filter. If omitted, the values default to the nreverb constants.

ifnCombs - function table with inumCombs comb filter time values directly followed the same number of comb gain values. The ftable should not be rescaled (use negative fgen number).
Positive time values are in seconds. The given time is converted internally into number of samples, and then set to the nearest greater prime number.
If time is negative, it is interpreted directly as time in sample frames, and no processing is done (except negation).

inumAlpas, ifnAlpas - same as inumCombs/ifnCombs, for allpass filter.

PERFORMANCE

This is a revision of nreverb which adds the possibility of reading any number of comb and allpass filter constants from a ftable.

EXAMPLES

Orchestra:
```
a1          soundin    "neopren.wav"
a2          nreverb    a1, 1.5, .75, 0, 8, 71, 4, 72
            out        a1 + a2 * .4
```

Score:
```
; freeverb time constants, as direct (negative) sample, with arbitrary ;
; gains
f71 0 16   -2  -1116 -1188 -1277 -1356 -1422 -1491 -1557 -1617  0.8
0.79  0.78  0.77  0.76  0.75  0.74  0.73
f72 0 16   -2  -556 -441 -341 -225  0.7  0.72  0.74  0.76

i1  0    7
e
```

BABO(Csound)

Babo

ar,al **babo** asig,ksrcx,ksrcy,ksrcz,irx,iry,irz[,idiff[,ifno]]

DESCRIPTION
Babo stands for BAll-within-the-BOx. It is a physical model reverberator based on the paper by Davide Rocchesso "The Ball within the Box: a sound-processing metaphor", Computer Music Journal, Vol 19, N.4, pp.45-47, Winter 1995.

A short description of the opcode should mention that it allows to define the resonator geometry along with some of its response characteristics, the position of the listener within the resonator, and the position of the source of sound. Babo then calculates early and later reflections by means of a tapped delay line and a circulant feedback delay network.

INITIALIZATION

irx, iry, irz – the coordinates of the geometry of the resonator (length of the edges in meters)

idiff - is the coefficient of diffusion at the walls, which regulates the amount of diffusion (0-1, where 0 = no diffusion, 1 = maximum diffusion - default: 1)

ifno - expert values function: a function number that holds all the additional parameters of the resonator

INITIALIZATION (Expert Values)
These values are contained in a function, typically a GEN2--type function used in non-rescaling mode.

decay - main decay of the resonator (default: 0.99)

hydecay - high frequency decay of the resonator (default: 0.1)

rcvx,rcvy,rcvz - the coordinates of the position of the receiver (=the listener) (in meters; 0,0,0 is the resonator center)

rdistance - is the distance in meters between the two pickups (i.e. your ears, for example... - default: 0.3)

direct - is the attenuation of the direct signal (0-1, default: 0.5)

early_diff - is the attenuation coefficient of the early reflections (0-1, default: 0.8)

PERFORMANCE

ar,al - the stereo output signal

asig - the input signal

ksrcx,ksrcy,ksrcz - the virtual coordinates of the source of sound (i.e. the input signal); these are allowed to move at k-rate and provide all the necessary variations in terms of response of the resonator

EXAMPLES

Orchestra File - Simple usage
```
; minimal babo instrument
instr 1
 ix       = p5      ; x position of source
 iy       = p6      ; y position of source
 iz       = p7      ; z position of source
 ixsize   = p8      ; width  of the resonator
 iysize   = p9      ; depth  of the resonator
 izsize   = p10     ; height of the resonator
```

```
    ainput      soundin p4

    al,ar       babo    ainput*0.9, ix, iy, iz, ixsize, iysize, izsize

                outs    al,ar

                endin
```

Score File - Simple Usage

```
    ; simple babo usage:
    ;p4     : soundin number
    ;p5     : x position of source
    ;p6     : y position of source
    ;p7     : z position of source
    ;p1     : width  of the resonator
    ;p12    : depth  of the resonator
    ;p13    : height of the resonator

    i1  0 10 1  6  4  3     14 .39  11 .86  10
    ;           ^^^^^^    ^^^^^^^^^^^^^^
    ;                       ++++++++++++: optimal room dims
    ;                                     according to
    ;                       Milner and Bernard JASA 85(2), 1989
    ;           +++++++: source position
    e
```

Orchestra File - Expert usage

```
                ; full blown babo instrument with movement
                instr 2
                ixstart     = p5    ; start x position of source (left-right)
                ixend       = p8    ; end   x position of source
                iystart     = p6    ; start y position of source (front-back)
                iyend       = p9    ; end   y position of source
                izstart     = p7    ; start z position of source (up-down)
                izend       = p10   ; end   z position of source
                ixsize      = p11   ; width  of the resonator
                iysize      = p12   ; depth  of the resonator
                izsize      = p13   ; height of the resonator
                idiff       = p14   ; diffusion coefficient
                iexpert     = p15;power user values stored in this function

    ainput      soundin p4
    ksource_x   line    ixstart, p3, ixend
    ksource_y   line    iystart, p3, iyend
    ksource_z   line    izstart, p3, izend
    al,ar   babo    ainput*0.9,     ksource_x,      ksource_y,      ksource_z,
ixsize, iysize, izsize, idiff, iexpert
                outs    al,ar

                endin
```

Score File - Expert Usage

```
    ; full blown instrument
    ;p5      : start x position of source (left-right)
    ;p6      : end   x position of source
    ;p7      : start y position of source (front-back)
    ;p8      : end   y position of source
    ;p9      : start z position of source (up-down)
    ;p10     : end   z position of source
    ;p11     : width  of the resonator
    ;p12     : depth  of the resonator
    ;p13     : height of the resonator
    ;p14     : diffusion coefficient
    ;p15     : power user values stored in this function

    ;       decay  hidecay rx ry rz rdistance direct early_diff
f1  0 8 -2  0.95   0.95  0  0  0    0.3     0.5    0.8  ; brighter
f2  0 8 -2  0.95   0.5   0  0  0    0.3     0.5    0.8  ; default (to be set as)
f3  0 8 -2  0.95   0.01   0  0  0    0.3     0.5    0.8  ; darker
f4  0 8 -2  0.95   0.7  0  0  0  0.3   0.1  0.4  ; to hear the effect of
                                                    ;diffusion
f5  0 8 -2  0.9    0.5    0  0  0    0.3     2.0   0.98 ; to hear the movement
f6  0 8 -2  0.99   0.1    0  0  0    0.3     0.5     0.8  ; default vals
;       ----- gen. number: negative to avoid rescaling

i2   10 10 1  6  4  3   6   4  3  14.39  11.86  10   1   6 ; defaults
i2   +   4 2  6  4  3   6   4  3  14.39  11.86  10   1   1 ; hear brightness 1
i2   +   4 2  6  4  3  -6  -4  3  14.39  11.86  10   1   2 ; hear brightness 2
i2   +   4 2  6  4  3  -6  -4  3  14.39  11.86  10   1   3 ; hear brightness 3
i2   +   3 2  .6 .4 .3 -.6 -.4 .3  1.439  1.186  1.0 0.0 4 ; hear diffusion 1
i2   +   3 2  .6 .4 .3 -.6 -.4 .3  1.439  1.186  1.0 1.0 4 ; hear diffusion 2
i2   +   4 2 12  4  3 -12  -4 -3  24.39  21.86  20   1   5 ; hear movement
i2   +   4 1  6  4  3   6   4  3  14.39  11.86   10  1   1 ; hear brightness 1
i2   +   4 1  6  4  3  -6  -4  3  14.39  11.86   10  1   2 ; hear brightness 2
i2   +   4 1  6  4  3  -6  -4  3  14.39  11.86   10  1   3 ; hear brightness 3
i2   +   3 1  .6 .4 .3 -.6 -.4 .3  1.439  1.186  1.0 0.0 4 ; hear diffusion 1
i2   +   3 1  .6 .4 .3 -.6 -.4 .3  1.439  1.186  1.0 1.0 4 ; hear diffusion 2
i2   +   4 1 12  4  3 -12  -4 -3 24 .39  21.86   20  1   5 ; hear movement
;           ^^^^^^^^^^^^^^^^     ^^^^^^^^^^^^^ ^   ^
;           ||||||||||||||||||||||||    ||||||||||||||||||||||| |  --: expert values
;                                                                  function
;           ||||||||||||||||||||||||    ||||||||||||||||||||||| +-- : diffusion
;           ||||||||||||||||||||||||          ----------------: optimal room dims
;           ||||||||||||||||||||||||  according to Milner and Bernard JASA
;                                                                  85(2), 1989
;           ----------------------: source position start and end
    e
```

AUTHORS

Davide Rocchesso (rocchesso@sci.univr.it) invented Babo, Padova 1994.

Paolo Filippi (paolfili@tiscalinet.it) coded the csound implementation, Padova 1999

Nicola Bernardini (nicb@axnet.it) wrote the manual page and cleaned up the code, Rome 2000.

---

ksig **sense**

Returns the ascii code of one of the keys that has been pressed, or -1 if no key.

Note that this is not properly verified, and seems not to work at all on Windows.

(JPff)

---

ar **transeg** istart, idur, itype, ivalue, [idur, itype, ivalue,]*
kr **transeg** istart, idur, itype, ivalue, [idur, itype, ivalue,]*

Constructs an envelope between istart and ivalue for a duration of idur seconds. If itype is 0 then a straight line is produced; otherwise is creates the curve

istart + (ivalue - istart) * (1 - exp(i*itype/(n-1))) / (1 - exp(itype))
for n steps

Thus if itype > 0 there is a slowly rising, fast decaying (convex) curve, while is itype < 0 the curve is fast rising, slowly decaying (concave).

(JPff -- with assistance from a number of people)

GEN16

f # time size 16 start dur type end [dur type end ....]

Creates a table from start to end of dur steps. It typoe is 0 this is a straight line. Otherwise it is

start + (end - start)*(1 - exp(i*type/(N-1)))/(1 - exp(type))

(JPff -- with assistance from a number of people)
