CSOUND

Published : 2012-04-25

INTRODUCTION

1. PREFACE
2. HOW TO USE THIS MANUAL
3. ON THIS RELEASE
4. LICENSE

1. PREFACE

Csound is one of the best known and longest established programs in the field of audio-programming. It was developed in the mid-1980s at the Massachusetts Institute of Technology (MIT) by Barry Vercoe.

Csound's history lies deep in the roots of computer music. It is a direct descendant of the oldest computer-program for sound synthesis, 'MusicN' by Max Mathews. Csound is free, distributed under the LGPL licence and is tended and expanded by a core of developers with support from a wider community.

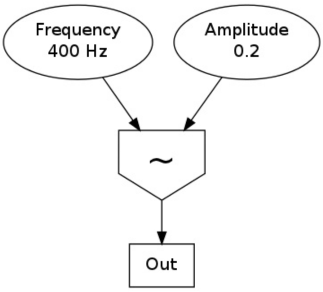
Csound has been growing for more than 25 years. There are few things related to audio that you cannot do with Csound. You can work by rendering offline, or in real-time by processing live audio and synthesizing sound on the fly. You can control Csound via MIDI, OSC, or via the Csound API (Application Programming Interface). In Csound, you will find the widest collection of tools for sound synthesis and sound modification, including special filters and tools for spectral processing.

Is Csound difficult to learn? Generally, graphical audio programming languages like Pure Data (more commonly known as Pd - see the Pure Data FLOSS Manual for further information), Max or Reaktor are easier to learn than text-coded audio programming languages like Csound, SuperCollider or ChucK. You cannot make a typo which produces an error which you do not understand. You program without being aware that you are programming. It feels like patching together different units in a studio. This is a fantastic approach. But when you deal with more complex projects, a text-based programming language is often easier to use and debug, and many people prefer to program by typing words and sentences rather than by wiring symbols together using the mouse.

Thanks to the work of Victor Lazzarini and Davis Pyon, it is also very easy to use Csound as a kind of audio engine inside Pd or Max. See the chapter Csound in other applications and the information on CSound with Pd and CSound in MaxMSP for further information.

Amongst text-based audio programming languages, Csound is arguably the simplest. You do not need to know anything about objects or functions. The basics of the Csound language are a straightforward transfer of the signal flow paradigm to text.

For example, to create a 400 Hz sine oscillator with an amplitude of 0.2, this is the signal flow:



This is a possible transformation of the signal graph into Csound code:

```
    instr Sine
aSig      oscils    0.2, 400, 0
          out       aSig
    endin
```

The oscillator is represented by the opcode *oscils* and gets its input arguments amplitude (0.2), frequency (400) and phase (0) righthand. It produces an audio signal called *aSig* at the left side, which is in turn the input of the second opcode *out*. The first and last lines encase these connections inside an instrument called *Sine*. That's it.

But it is often difficult to find up to date resources that explain all of the things that are possible with Csound. Documentation and tutorials produced by many experienced users tend to be scattered across many different locations. This was one of the main motivations in producing this manual: to facilitate a flow between the knowledge of contemporary Csound users and those wishing to learn more about Csound.

Ten years after the milestone of Richard Boulanger's Csound Book the Csound FLOSS Manual is intended to offer an easy-to-understand introduction and to provide a centre of up to date information about the many features of Csound - not as detailed and in depth as the Csound Book, but including new information and sharing this knowledge with the wider Csound community.

Throughout this manual we will attempt a difficult balancing act: we want to provide users with nearly everything important there is to know about Csound, but we also want to keep things simple and concise to save you from drowning under the multitude of things that we could say about Csound. Frequently this manual will link to other more detailed resources like the Canonical Csound Reference Manual, the primary documentation provided by the Csound developers and associated community over the years, and the Csound Journal (edited by Steven Yi and James Hearon), a quarterly online publication with many great Csound-related articles.

Good luck and happy Csounding!

2. HOW TO USE THIS MANUAL

The goal of this manual is to provide a readable introduction to Csound. In no way is it meant as a replacement for the Canonical Csound Reference Manual. It is intended as an introduction-tutorial-reference hybrid, gathering the most important information you need for working with Csound in a variety of situations. In many places, links are provided to other resources, such as the official manual, the Csound Journal, example collections, and more.

It is not necessary to read each chapter in sequence, feel free to jump to any chapter that interests you, although bear in mind that occasionally a chapter will make reference to a previous one.

If you are new to Csound, the QUICK START chapter will be the best place to go to get started. BASICS provides a general introduction to key concepts about digital sound vital to understanding how Csound deals with audio. The CSOUND LANGUAGE chapter provides greater detail about how Csound works and how to work with Csound.

SOUND SYNTHESIS introduces various methods of creating sound from scratch and SOUND MODIFICATION describes various methods of transforming sounds that already exist within Csound. SAMPLES outlines ways in which to record and play audio samples in Csound, an area that might be of particular interest to those intent on using Csound as a real-time performance instrument. The MIDI and OPEN SOUND CONTROL chapters focus on different methods of controlling Csound using external software or hardware. The final chapters introduce various front-ends that can be used to interface with the Csound engine and Csound's communication with other applications.

If you would like to know more about a topic, and in particular about the use of any opcode, refer first to the Canonical Csound Reference Manual.

All files - examples and audio files - can be downloaded at www.csound-tutorial.net . If you use CsoundQt, you can find all the examples in CsoundQt's examples menu under "Floss Manual Examples".

Like other Audio Tools, Csound can produce extreme dynamic range. Be careful when you run the examples! Start with a low volume setting on your amplifier and take special care when using headphones.

You can help to improve this manual, either in reporting bugs or requests, or in joining as a writer. Just contact one of the maintainers (see the list in ON THIS RELEASE).

Thanks to Alex Hofmann, this manual can be ordered as a print-on-demand at www.lulu.com. Just use the search utility there and look for "Csound". Just the links will not work ...

3. ON THIS RELEASE

We are happy to announce the second release of the Csound Floss Manual. It has been an exciting year for Csound, with many activities and important developments. Thanks to the long and hard work of Steven Yi, John ffitch, Tito Latini and others, a new parser has been written. This opens up many new possibilities for future language adaptations and more flexibility within the Csound syntax. In autumn 2011, the first international Csound Conference took place at HMTM Hannover, with many inspiring workshops, concerts, papers and most notably discussions between developers and users. In early 2012, Jim Aikin's *Csound Power!* was published and it represents a very well written introduction to Csound. In early spring, Victor Lazzarini and Steven Yi published the first release of Csound on Android devices, and all developers are currently pushing towards Csound6.

The first edition of the Csound Floss Manual has been a huge success. We are proud and glad to see it used, linked and quoted in many places. It has come to be regarded as a complement to the Csound Manual. We hope we can continue to reflect Csound's development in this manual. The core writers of the Csound Floss manual would like to extend their thanks to Richard Boulanger, John Clements and others for their support, and to all the writers for their various contributions. Thanks also are due to Adam Hyde and the team at flossmanuals.net for maintaining and developing this important platform for free libre open source software.

What's new in this Release

- New chapters:
    - MACROS (Csound Language)
    - CABBAGE (Csound Frontends)
    - BUILDING CSOUND (Appendix)
    - METHODS OF WRITING CSOUND SCORES (Appendix)
- Chapters now completed:
    - WAVESHAPING (Sound Synthesis)
    - PHYSICAL MODELLING (Sound Synthesis)
    - CONVOLUTION (Sound Modification)
    - CSOUND VIA TERMINAL (Csound Frontends)
    - CSOUND UTILITIES
- Significant amendments and additions to the following chapters:
    - AM / RM / WAVESHAPING (Sound Modification)
    - GRANULAR SYNTHESIS (Sound Modification)
    - CSOUND IN PD (Csound in Other Applications)
    - LINKS (Appendix)
- New chapters as drafts:
    - CSOUND IN ABLETON LIVE (Csound in Other Applications)
    - CSOUND AS A VST PLUGIN (Csound in Other Applications)
    - PYTHON IN CSOUNDQT
    - LUA IN CSOUND
- Slight changes in the structure (the TERMINAL is now considered as a frontend, and THE CSOUND API chapter is now part of the section Csound and other Programming Languages)

Still on the To-Do-List:

- More and better illustrations
- Adding examples for VBAP, Ambisonics etc in PANNING AND SPATIALIZATION (Sound Modification)
- Adding examples and explanations in METHODS OF WRITING CSOUND SCORES (Appendix)
- Update OPCODE GUIDE (and more eyes on it at all)
- Much more should be written in the GLOSSARY
- Except the new drafted chapters PYTHON INSIDE CSOUND and EXTENDING CSOUND are still to write.

Last summer Alex Hofmann put a lot of work into making this manual available as a book on www.lulu.com. Just use the search utility there and look for "Csound", if you would like to obtain a printed version. This second release will be available soon.

Surround Wunderbar Studios, Berlin, 30th March, 2012

Joachim Heintz & Iain McCurdy

Foreword on the First Release

In spring 2010 a group of Csounders decided to start this project. The chapter outline was suggested by Joachim Heintz with suggestions and improvements provided by Richard Boulanger, Oeyvind Brandtsegg, Andrés Cabrera, Alex Hofmann, Jacob Joaquin, Iain McCurdy, Rory Walsh and others. Rory also pointed us to the FLOSS Manuals platform as a possible environment for writing and publishing. Stefano Bonetti, François Pinot, Davis Pyon and Steven Yi joined later and wrote chapters.

In a volunteer project like this, it is not always easy to sustain momentum so in the spring of 2011 some members of the team met in Berlin for a 'book sprint' to achieve a level of completion, and publish a first release.

With heads spinning and square eyes we are happy and proud to offer this manual to you. At the same time we realize that this is a first release with much potential for further improvement. Several chapters have yet to be written, others are not yet complete and the differences between the various authors in terms of the level at which they aim and their degree of detail are perhaps larger than they should be.

This is therefore a beginning. Everyone is invited to improve this book. You can begin to write for one of the empty chapters, contribute to an existing one or insert new examples where you feel they are of benefit. You just need to create an account at http://booki.flossmanuals.net or to let us know of your suggestions.

We hope you enjoy using this manual, we had fun writing it!

Berlin, 31st March, 2011

Joachim Heintz Alex Hofmann Iain McCurdy

jh at joachimheintz.de alex at boomclicks.de i_mccurdy at hotmail.com

You can order a printed version here:

http://www.lulu.com/product/paperback/csound---floss-manual/16265055

4. LICENSE

AUTHORS

Note that this book is a collective effort, so some of the contributors may not have been quoted correctly. If you are one of them, please contact us, or simply put your name at the right place.

INTRODUCTION

PREFACE

Joachim Heintz, Andres Cabrera, Alex Hofmann, Iain McCurdy

HOW TO USE THIS MANUAL
Joachim Heintz, Andres Cabrera, Iain McCurdy

01 BASICS

A. DIGITAL AUDIO
Alex Hofmann, Rory Walsh, Iain McCurdy, Joachim Heintz

B. PITCH AND FREQUENCY
Rory Walsh, Iain McCurdy, Joachim Heintz

C. INTENSITIES
Joachim Heintz

02 QUICK START

A. RUN CSOUND

Alex Hofmann, Joachim Heintz, Andres Cabrera, Iain McCurdy

B. CSOUND SYNTAX
Alex Hofmann, Joachim Heintz, Andres Cabrera, Iain McCurdy

C. CONFIGURING MIDI
Andres Cabrera, Joachim Heintz, Iain McCurdy

D. LIVE AUDIO
Alex Hofmann, Andres Cabrera, Iain McCurdy, Joachim Heintz

E. RENDERING TO FILE
Joachim Heintz, Alex Hofmann, Andres Cabrera, Iain McCurdy

03 CSOUND LANGUAGE

A. INITIALIZATION AND PERFORMANCE PASS

Joachim Heintz

B. LOCAL AND GLOBAL VARIABLES
Joachim Heintz, Andres Cabrera, Iain McCurdy

C. CONTROL STRUCTURES
Joachim Heintz

D. FUNCTION TABLES
Joachim Heintz, Iain McCurdy

E. TRIGGERING INSTRUMENT EVENTS
Joachim Heintz, Iain McCurdy

F. USER DEFINED OPCODES
Joachim Heintz

G. MACROS
Iain McCurdy

04 SOUND SYNTHESIS

A. ADDITIVE SYNTHESIS

Andres Cabrera, Joachim Heintz

B. SUBTRACTIVE SYNTHESIS
Iain McCurdy

C. AMPLITUDE AND RINGMODULATION
Alex Hofman

D. FREQUENCY MODULATION
Alex Hofmann, Bjorn Houdorf

E. WAVESHAPING
Joachim Heintz

F. GRANULAR SYNTHESIS
Iain McCurdy

G. PHYSICAL MODELLING
Joachim Heintz, Iain McCurdy

05 SOUND MODIFICATION

A. ENVELOPES

Iain McCurdy

B. PANNING AND SPATIALIZATION
Iain McCurdy

C. FILTERS
Iain McCurdy

D. DELAY AND FEEDBACK
Iain McCurdy

E. REVERBERATION
Iain McCurdy

F. AM / RM / WAVESHAPING
Alex Hofmann, Joachim Heintz

G. GRANULAR SYNTHESIS
Iain McCurdy

H. CONVOLUTION
Iain McCurdy

I. FOURIER ANALYSIS / SPECTRAL PROCESSING
Joachim Heintz

06 SAMPLES

A. RECORD AND PLAY SOUNDFILES

Iain McCurdy, Joachim Heintz

B. RECORD AND PLAY BUFFERS
Iain McCurdy, Joachim Heintz, Andres Cabrera

07 MIDI

A. RECEIVING EVENTS BY MIDIIN

Iain McCurdy

B. TRIGGERING INSTRUMENT INSTANCES
Joachim Heintz, Iain McCurdy

C. WORKING WITH CONTROLLERS
Iain McCurdy

D. READING MIDI FILES
Iain McCurdy

E. MIDI OUTPUT
Iain McCurdy

08 OSC AND WII

OSC AND WII

Alex Hofmann

09 CSOUND IN OTHER APPLICATIONS

A. CSOUND IN PD

Joachim Heintz, Jim Aikin

B. CSOUND IN MAXMSP
Davis Pyon

C. CSOUND IN ABLETON LIVE
Rory Walsh

D. CSOUND AS A VST PLUGIN
Rory Walsh

10 CSOUND FRONTENDS

CSOUNDQT

Andrés Cabrera

WINXOUND
Stefano Bonetti

BLUE
Steven Yi

CABBAGE
Rory Walsh

CSOUND VIA TERMINAL
Iain McCurdy

11 CSOUND UTILITIES

CSOUND UTILITIES

Iain McCurdy

12 CSOUND AND OTHER PROGRAMMING LANGUAGES

A. THE CSOUND API

François Pinot, Rory Walsh

B. PYTHON INSIDE CSOUND

C. PYTHON IN CSOUNDQT

D. LUA IN CSOUND

13 EXTENDING CSOUND

EXTENDING CSOUND

OPCODE GUIDE

OVERVIEW

Joachim Heintz

SIGNAL PROCESSING I
Joachim Heintz

SIGNAL PROCESSING II
Joachim Heintz

DATA

Joachim Heintz

REALTIME INTERACTION
Joachim Heintz

INSTRUMENT CONTROL
Joachim Heintz

MATH, PYTHON/SYSTEM, PLUGINS
Joachim Heintz

APPENDIX

GLOSSARY

Joachim Heintz

LINKS
Joachim Heintz, Stefano Bonetti

BUILDING CSOUND
Ernesto Illescas, Menno Knevel, Joachim Heintz

METHODS OF WRITING CSOUND SCORES
Iain McCurdy, Joachim Heintz

V.1 - Final Editing Team in March 2011:

Joachim Heintz, Alex Hofmann, Iain McCurdy

V.2 - Final Editing Team in March 2012:

Joachim Heintz, Iain McCurdy

FLOSS MANUALS

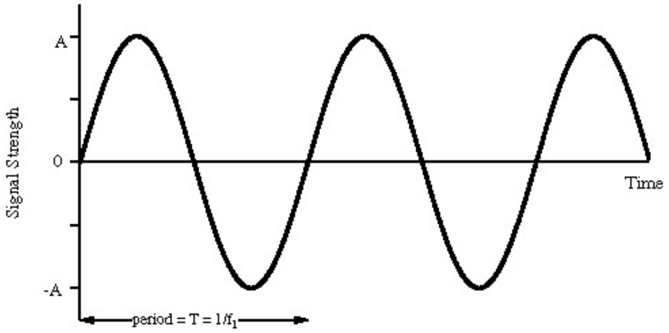
Free manuals for free software

01 BASICS

5. DIGITAL AUDIO
6. FREQUENCIES
7. INTENSITIES

5. DIGITAL AUDIO

At a purely physical level, sound is simply a mechanical disturbance of a medium. The medium in question may be air, solid, liquid, gas or a mixture of several of these. This disturbance to the medium causes molecules to move to and fro in a spring-like manner. As one molecule hits the next, the disturbance moves through the medium causing sound to travel. These so called compressions and rarefactions in the medium can be described as sound waves. The simplest type of waveform, describing what is referred to as 'simple harmonic motion', is a sine wave.



(a) Sine Wave

Each time the waveform signal goes above 0 the molecules are in a state of compression meaning they are pushing towards each other. Every time the waveform signal drops below 0 the molecules are in a state of rarefaction meaning they are pulling away from each other. When a waveform shows a clear repeating pattern, as in the case above, it is said to be periodic. Periodic sounds give rise to the sensation of pitch.

ELEMENTS OF A SOUND WAVE

Periodic waves have four common parameters, and each of the four parameters affects the way we perceive sound.

- **Period**: This is the length of time it takes for a waveform to complete one cycle. This amount of time is referred to as *t*
- **Wavelength()**: the distance it takes for a wave to complete one full period. This is usually measured in meters.
- **Frequency**: the number of cycles or periods per second. Frequency is measured in Hertz. If a sound has a frequency of 440Hz it completes 440 cycles every second. Given a frequency, one can easily calculate the period of any sound. Mathematically, the period is the reciprocal of the frequency (and vice versa). In equation form, this is expressed as follows.

```
Frequency = 1/Period          Period = 1/Frequency
```

Therefore the frequency is the inverse of the period, so a wave of 100 Hz frequency has a period of 1/100 or 0.01 secs, likewise a frequency of 256Hz has a period of 1/256, or 0.004 secs. To calculate the wavelength of a sound in any given medium we can use the following equation:

```
Wavelength = Velocity/Frequency
```

Humans can hear frequencies from 20Hz to 20000Hz (although this can differ dramatically from individual to individual). You can read more about frequency in the next chapter.

- **Phase:** This is the starting point of a waveform. The starting point along the Y-axis of our plotted waveform is not always 0. This can be expressed in degrees or in radians. A complete cycle of a waveform will cover 360 degrees or (2 x pi) radians.

- **Amplitude:** Amplitude is represented by the y-axis of a plotted pressure wave. The strength at which the molecules pull or push away from each other will determine how far above and below 0 the wave fluctuates. The greater the y-value the greater the amplitude of our wave. The greater the compressions and rarefactions the greater the amplitude.

TRANSDUCTION

The analogue sound waves we hear in the world around us need to be converted into an electrical signal in order to be amplified or sent to a soundcard for recording. The process of converting acoustical energy in the form of pressure waves into an electrical signal is carried out by a device known as a a transducer.

A transducer, which is usually found in microphones, produces a changing electrical voltage that mirrors the changing compression and rarefaction of the air molecules caused by the sound wave. The continuous variation of pressure is therefore 'transduced' into continuous variation of voltage. The greater the variation of pressure the greater the variation of voltage that is sent to the computer.

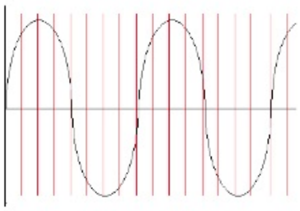
Ideally, the transduction process should be as transparent and clean as possible: i.e., whatever goes in comes out as a perfect voltage representation. In the real world however this is never the case. Noise and distortion are always incorporated into the signal. Every time sound passes through a transducer or is transmitted electrically a change in signal quality will result. When we talk of 'noise' we are talking specifically about any unwanted signal captured during the transduction process. This normally manifests itself as an unwanted 'hiss'.

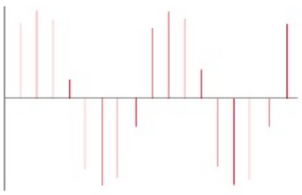
SAMPLING

The analogue voltage that corresponds to an acoustic signal changes continuously so that at each instant in time it will have a different value. It is not possible for a computer to receive the value of the voltage for every instant because of the physical limitations of both the computer and the data converters (remember also that there are an infinite number of instances between every two instances!).

What the soundcard can do however is to measure the power of the analogue voltage at intervals of equal duration. This is how all digital recording works and is known as 'sampling'. The result of this sampling process is a discrete or digital signal which is no more than a sequence of numbers corresponding to the voltage at each successive sample time.

Below left is a diagram showing a sinusoidal waveform. The vertical lines that run through the diagram represents the points in time when a snapshot is taken of the signal. After the sampling has taken place we are left with what is known as a discrete signal consisting of a collection of audio samples, as illustrated in the diagram on the right hand side below. If one is recording using a typical audio editor the incoming samples will be stored in the computer RAM (Random Access Memory). In Csound one can process the incoming audio samples in real time and output a new stream of samples, or write them to disk in the form of a sound file.





It is important to remember that each sample represents the amount of voltage, positive or negative, that was present in the signal at the point in time the sample or snapshot was taken.

The same principle applies to recording of live video. A video camera takes a sequence of pictures of something in motion for example. Most video cameras will take between 30 and 60 still pictures a second. Each picture is called a frame. When these frames are played we no longer perceive them as individual pictures. We perceive them instead as a continuous moving image.

ANALOGUE VERSUS DIGITAL

In general, analogue systems can be quite unreliable when it comes to noise and distortion. Each time something is copied or transmitted, some noise and distortion is introduced into the process. If this is done many times, the cumulative effect can deteriorate a signal quite considerably. It is because of this, the music industry has turned to digital technology, which so far offers the best solution to this problem. As we saw above, in digital systems sound is stored as numbers, so a signal can be effectively "cloned". Mathematical routines can be applied to prevent errors in transmission, which could otherwise introduce noise into the signal.

SAMPLE RATE AND THE SAMPLING THEOREM

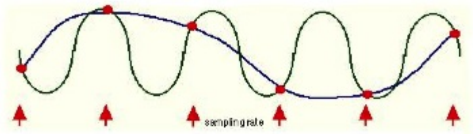
The sample rate describes the number of samples (pictures/snapshots) taken each second. To sample an audio signal correctly it is important to pay attention to the sampling theorem:

"To represent digitally a signal containing frequencies up to X Hz, it is necessary to use a sampling rate of at least 2X samples per second"

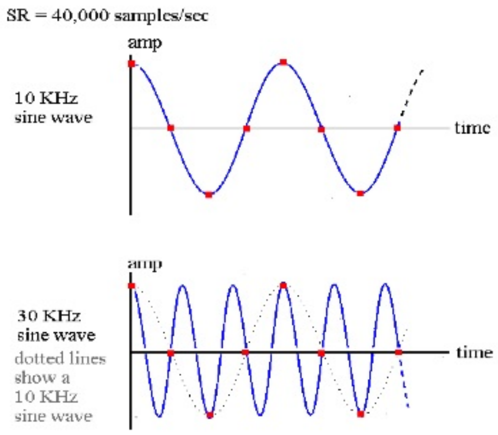
According to this theorem, a soundcard or any other digital recording device will not be able to represent any frequency above 1/2 the sampling rate. Half the sampling rate is also referred to as the Nyquist frequency, after the Swedish physicist Harry Nyquist who formalized the theory in the 1920s. What it all means is that any signal with frequencies above the Nyquist frequency will be misrepresented. Furthermore it will result in a frequency lower than the one being sampled. When this happens it results in what is known as aliasing or foldover.

ALIASING

Here is a graphical representation of aliasing.



The sinusoidal wave form in blue is being sampled at each arrow. The line that joins the red circles together is the captured waveform. As you can see the captured wave form and the original waveform have different frequencies. Here is another example:



We can see that if the sample rate is 40,000 there is no problem sampling a signal that is 10KHz. On the other hand, in the second example it can be seen that a 30kHz waveform is not going to be correctly sampled. In fact we end up with a waveform that is 10kHz, rather than 30kHz.

The following Csound instrument plays a 1000 Hz tone first directly, and then because the frequency is 1000 Hz lower than the sample rate of 44100 Hz:

EXAMPLE 01A01.csd

```
<CsoundSynthesizer>
<CsOptions>
-odac
</CsOptions>
<CsInstruments>
;example by Joachim Heintz
sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

instr 1
asig    oscils  .2, p4, 0
        outs    asig, asig
endin

</CsInstruments>
<CsScore>
i 1 0 2 1000 ;1000 Hz tone
i 1 3 2 43100 ;43100 Hz tone sounds like 1000 Hz because of aliasing
</CsScore>
</CsoundSynthesizer>
```

The same phenomenon takes places in film and video too. You may recall having seen wagon wheels apparently move backwards in old Westerns. Let us say for example that a camera is taking 60 frames per second of a wheel moving. If the wheel is completing one rotation in exactly 1/60th of a second, then every picture looks the same. - as a result the wheel appears to stand still. If the wheel speeds up, i.e., increases frequency, it will appear as if the wheel is slowly turning backwards. This is because the wheel will complete more than a full rotation between each snapshot. This is the most ugly side-effect of aliasing - wrong information.

As an aside, it is worth observing that a lot of modern 'glitch' music intentionally makes a feature of the spectral distortion that aliasing induces in digital audio.

Audio-CD Quality uses a sample rate of 44100Kz (44.1 kHz). This means that CD quality can only represent frequencies up to 22050Hz. Humans typically have an absolute upper limit of hearing of about 20Khz thus making 44.1KHz a reasonable standard sampling rate.

BITS, BYTES AND WORDS. UNDERSTANDING BINARY.

All digital computers represent data as a collection of bits (short for binary digit). A bit is the smallest possible unit of information. One bit can only be one of two states - off or on, 0 or 1. The meaning of the bit, which can represent almost anything, is unimportant at this point. The thing to remember is that all computer data - a text file on disk, a program in memory, a packet on a network - is ultimately a collection of bits.

Bits in groups of eight are called bytes, and one byte usually represents a single character of data in the computer. It's a little used term, but you might be interested in knowing that a nibble is half a byte (usually 4 bits).

THE BINARY SYSTEM

All digital computers work in a environment that has only two variables, 0 and 1. All numbers in our decimal system therefore must be translated into 0's and 1's in the binary system. If you think of
binary numbers in terms of switches. With one switch you can represent up to two different numbers.

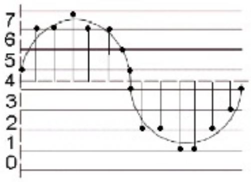
0 (OFF) = Decimal 0
1 (ON) = Decimal 1

Thus, a single bit represents 2 numbers, two bits can represent 4 numbers, three bits represent 8 numbers, four bits represent 16 numbers, and so on up to a byte, or eight bits, which represents 256 numbers. Therefore each added bit doubles the amount of possible numbers that can be represented. Put simply, the more bits you have at your disposal the more information you can store.

BIT-DEPTH RESOLUTION

Apart from the sample rate, another important parameter which can affect the fidelity of a digital signal is the accuracy with which each sample is known, in other words knowing how strong each voltage is. Every sample obtained is set to a specific amplitude (the measure of strength for each voltage) level. The number of levels depends on the precision of the measurement in bits, i.e., how many binary digits are used to store the samples. The number of bits that a system can use is normally referred to as the bit-depth resolution.

If the bit-depth resolution is 3 then there are 8 possible levels of amplitude that we can use for each sample. We can see this in the diagram below. At each sampling period the soundcard plots an amplitude. As we are only using a 3-bit system the resolution is not good enough to plot the correct amplitude of each sample. We can see in the diagram that some vertical lines stop above or below the real signal. This is because our bit-depth is not high enough to plot the amplitude levels with sufficient accuracy at each sampling period.



```
example here for 4, 6, 8, 12, 16 bit of a sine signal ...
... coming in the next release
```

The standard resolution for CDs is 16 bit, which allows for 65536 different possible amplitude levels, 32767 either side of the zero axis. Using bit rates lower than 16 is not a good idea as it will result in noise being added to the signal. This is referred to as quantization noise and is a result of amplitude values being excessively rounded up or down when being digitized. Quantization noise becomes most apparent when trying to represent low amplitude (quiet) sounds. Frequently a tiny amount of noise, known as a dither signal, will be added to digital audio before conversion back into an analogue signal. Adding this dither signal will actually reduce the more noticeable noise created by quantization. As higher bit depth resolutions are employed in the digitizing process the need for dithering is reduced. A general rule is to use the highest bit rate available.

Many electronic musicians make use of deliberately low bit depth quantization in order to add noise to a signal. The effect is commonly known as 'bit-crunching' and is relatively easy to do in Csound.

ADC / DAC

The entire process, as described above, of taking an analogue signal and converting it into a digital signal is referred to as analogue to digital conversion or ADC. Of course digital to analogue conversion, DAC, is also possible. This is how we get to hear our music through our PC's headphones or speakers. For example, if one plays a sound from Media Player or iTunes the software will send a series of numbers to the computer soundcard. In fact it will most likely send 44100 numbers a second. If the audio that is playing is 16 bit then these numbers will range from -32768 to +32767.

When the sound card receives these numbers from the audio stream it will output corresponding voltages to a loudspeaker. When the voltages reach the loudspeaker they cause the loudspeakers magnet to move inwards and outwards. This causes a disturbance in the air around the speaker resulting in what we perceive as sound.

6. FREQUENCIES

As mentioned in the previous section frequency is defined as the number of cycles or periods per second. Frequency is measured in Hertz. If a tone has a frequency of 440Hz it completes 440 cycles every second. Given a tone's frequency, one can easily calculate the period of any sound. Mathematically, the period is the reciprocal of the frequency and vice versa. In equation form, this is expressed as follows.

```
 Frequency = 1/Period         Period = 1/Frequency
```

Therefore the frequency is the inverse of the period, so a wave of 100 Hz frequency has a period of 1/100 or 0.01 seconds, likewise a frequency of 256Hz has a period of 1/256, or 0.004 seconds. To calculate the wavelength of a sound in any given medium we can use the following equation:

$\lambda = Velocity/Frequency$

For instance, a wave of 1000 Hz in air (velocity of diffusion about 340 m/s) has a length of approximately 340/1000 m = 34 cm.

LOWER AND HIGHER BORDERS FOR HEARING

The human ear can generally hear sounds in the range 20 Hz to 20,000 Hz (20 kHz). This upper limit tends to decrease with age due to a condition known as presbyacusis, or age related hearing loss. Most adults can hear to about 16 kHz while most children can hear beyond this. At the lower end of the spectrum the human ear does not respond to frequencies below 20 Hz, with 40 of 50 Hz being the lowest most people can perceive.

So, in the following example, you will not hear the first (10 Hz) tone, and probably not the last (20 kHz) one, but hopefully the other ones (100 Hz, 1000 Hz, 10000 Hz):

EXAMPLE 01B01.csd

```
<CsoundSynthesizer>
<CsOptions>
-odac -m0
</CsOptions>
<CsInstruments>
;example by joachim heintz
sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

instr 1
        prints  "Playing %d Hertz!\n", p4
asig    oscils  .2, p4, 0
        outs    asig, asig
endin

</CsInstruments>
<CsScore>
i 1 0 2 10
i . + . 100
i . + . 1000
i . + . 10000
i . + . 20000
</CsScore>
</CsoundSynthesizer>
```

LOGARITHMS, FREQUENCY RATIOS AND INTERVALS

A lot of basic maths is about simplification of complex equations. Shortcuts are taken all the time to make things easier to read and equate. Multiplication can be seen as a shorthand of addition, for example, 5x10 = 5+5+5+5+5+5+5+5+5+5. Exponents are shorthand for multiplication, 3^5 = 3x3x3x3x3. Logarithms are shorthand for exponents and are used in many areas of science and engineering in which quantities vary over a large range. Examples of logarithmic scales include the decibel scale, the Richter scale for measuring earthquake magnitudes and the astronomical scale of stellar brightnesses. Musical frequencies also work on a logarithmic scale, more on this later.

Intervals in music describe the distance between two notes. When dealing with standard musical notation it is easy to determine an interval between two adjacent notes. For example a perfect 5th is always made up of 7 semitones. When dealing with Hz values things are different. A difference of say 100Hz does not always equate to the same musical interval. This is because musical intervals as we hear them are represented in Hz as frequency ratios. An octave for example is always 2:1. That is to say every time you double a Hz value you will jump up by a musical interval of an octave.

Consider the following. A flute can play the note A at 440 Hz. If the player plays another A an octave above it at 880 Hz the difference in Hz is 440. Now consider the piccolo, the highest pitched instrument of the orchestra. It can play a frequency of 2000 Hz but it can also play an octave above this at 4000 Hz (2 x 2000 Hz). While the difference in Hertz between the two notes on the flute is only 440 Hz, the difference between the two high pitched notes on a piccolo is 1000 Hz yet they are both only playing notes one octave apart.

What all this demonstrates is that the higher two pitches become the greater the difference in Hertz needs to be for us to recognize the difference as the same musical interval. The most common ratios found in the equal temperament scale are the unison: (1:1), the octave: (2:1), the perfect fifth (3:2), the perfect fourth (4:3), the major third (5:4) and the minor third (6:5).

The following example shows the difference between adding a certain frequency and applying a ratio. First, the frequencies of 100, 400 and 800 Hz all get an addition of 100 Hz. This sounds very different, though the added frequency is the same. Second, the ratio 3/2 (perfect fifth) is applied to the same frequencies. This sounds always the same, though the frequency displacement is different each time.

EXAMPLE 01B02.csd

```
<CsoundSynthesizer>
<CsOptions>
-odac -m0
</CsOptions>
<CsInstruments>
;example by joachim heintz
sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

instr 1
        prints  "Playing %d Hertz!\n", p4
asig    oscils  .2, p4, 0
        outs    asig, asig
endin

instr 2
        prints  "Adding %d Hertz to %d Hertz!\n", p5, p4
asig    oscils  .2, p4+p5, 0
        outs    asig, asig
endin

instr 3
        prints  "Applying the ratio of %f (adding %d Hertz)
                 to %d Hertz!\n", p5, p4*p5, p4
asig    oscils  .2, p4*p5, 0
        outs    asig, asig
endin

</CsInstruments>
<CsScore>
;adding a certain frequency (instr 2)
i 1 0 1 100
i 2 1 1 100 100
i 1 3 1 400
i 2 4 1 400 100
i 1 6 1 800
i 2 7 1 800 100
;applying a certain ratio (instr 3)
i 1 10 1 100
i 3 11 1 100 [3/2]
i 1 13 1 400
i 3 14 1 400 [3/2]
i 1 16 1 800
i 3 17 1 800 [3/2]
```

```
</CsScore>
</CsoundSynthesizer>
```

So what of the algorithms mentioned above. As some readers will know the current preferred method of tuning western instruments is based on equal temperament. Essentially this means that all octaves are split into 12 equal intervals. Therefore a semitone has a ratio of $2^{(1/12)}$, which is approximately 1.059463.

So what about the reference to logarithms in the heading above? As stated previously, logarithms are shorthand for exponents. $2^{(1/12)}$= 1.059463 can also be written as log2(1.059463)= 1/12. Therefore musical frequency works on a logarithmic scale.

MIDI NOTES

Csound can easily deal with MIDI notes and comes with functions that will convert MIDI notes to Hertz values and back again. In MIDI speak A440 is equal to A4. You can think of A4 as being the fourth A from the lowest A we can hear, well almost hear.

Caution: like many 'standards' there is occasional disagreement about the mapping between frequency and octave number. You may occasionally encounter A440 being described as A3.

7. INTENSITIES

REAL WORLD INTENSITIES AND AMPLITUDES

There are many ways to describe a sound physically. One of the most common is the Sound Intensity Level (SIL). It describes the amount of power on a certain surface, so its unit is Watt per square meter (W/m^2). The range of human hearing is about $10^{-12} W/m^2$ at the threshold of hearing to $10^0 W/m^2$ at the threshold of pain. For ordering this immense range, and to facilitate the measurement of one sound intensity based upon its ratio with another, a logarithmic scale is used. The unit *Bel* describes the relation of one intensity I to a reference intensity I0 as follows:

$\log_{10} \frac{I}{I_0}$ *Sound Intensity Level in Bel*

If, for instance, the ratio $\frac{I}{I_0}$ is 10, this is 1 Bel. If the ratio is 100, this is 2 Bel.

For real world sounds, it makes sense to set the reference value I_0 to the threshold of hearing which has been fixed as $10^{-12} W/m^2$ at 1000 Hertz. So the range of hearing covers about 12 Bel. Usually 1 Bel is divided into 10 deci Bel, so the common formula for measuring a sound intensity is:

$10 \cdot \log_{10} \frac{I}{I_0}$ ***Sound Intensity Level (SIL) in Decibel (dB)*** with $I_0 = 10^{-12} W/m^2$

While the sound intensity level is useful to describe the way in which the human hearing works, the *measurement* of sound is more closely related to the sound pressure deviations. Sound waves compress and expand the air particles and by this they increase and decrease the localized air pressure. These deviations are measured and transformed by a microphone. So the question arises: what is the relationship between the sound pressure deviations and the sound intensity? The answer is: sound intensity changes I are proportional to the *square* of the sound pressure changes P . As a formula:

$I \approx P^2$ *Relation between Sound Intensity and Sound Pressure*

Let us take an example to see what this means. The sound pressure at the threshold of hearing can be fixed at $2 \cdot 10^{-5} Pa$. This value is the reference value of the Sound Pressure Level (SPL). If we have now a value of $2 \cdot 10^{-4} Pa$, the corresponding sound intensity relation can be calculated as:

$$\left(\frac{2 \cdot 10^4}{2 \cdot 10^5}\right)^2 = 10^2 = 100$$

So, a factor of 10 at the pressure relation yields a factor of 100 at the intensity relation. In general, the dB scale for the pressure P related to the pressure P0 is:

$$10 \cdot \log_{10}\left(\frac{P}{P_0}\right)^2 = 2 \cdot 10 \cdot \log_{10} \frac{P}{P_0} = 20 \cdot \log_{10} \frac{P}{P_0}$$

Sound Pressure Level (SPL) in Decibel (dB) with $P_0 = 2 \cdot 10^{-5} Pa$

Working with Digital Audio basically means working with *amplitudes*. What we are dealing with microphones are amplitudes. Any audio file is a sequence of amplitudes. What you generate in Csound and write either to the DAC in realtime or to a sound file, are again nothing but a sequence of amplitudes. As amplitudes are directly related to the sound pressure deviations, all the relations between sound intensity and sound pressure can be transferred to relations between sound intensity and amplitudes:

$I \approx A^2$ ***Relation between Intensity and Ampltitudes***

$20 \cdot \log_{10} \frac{A}{A_0}$ ***Decibel (dB) Scale of Amplitudes*** with any amplitude A related to an other amplitude A_0

If you drive an oscillator with the amplitude 1, and another oscillator with the amplitude 0.5, and you want to know the difference in dB, you calculate:

$$20 \cdot \log_{10} \frac{1}{0.5} = 20 \cdot \log_{10} 2 = 20 \cdot 0.30103 = 6.0206 dB$$

So, the most useful thing to keep in mind is: when you double the amplitude, you get +6 dB; when you have half of the amplitude as before, you get -6 dB.

WHAT IS 0 DB?

As described in the last section, any dB scale - for intensities, pressures or amplitudes - is just a way to describe a *relationship*. To have any sort of quantitative measurement you will need to know the reference value referred to as "0 dB". For real world sounds, it makes sense to set this level to the threshold of hearing. This is done, as we saw, by setting the SIL to $10^{-12} W/m^2$ and the SPL to $2 \cdot 10^{-5} Pa$.

But for working with digital sound in the computer, this does not make any sense. What you will hear from the sound you produce in the computer, just depends on the amplification, the speakers, and so on. It has nothing, per se, to do with the level in your audio editor or in Csound. Nevertheless, there *is* a rational reference level for the amplitudes. In a digital system, there is a strict limit for the maximum number you can store as amplitude. This maximum possible level is called 0 dB.

Each program connects this maximum possible amplitude with a number. Usually it is '1' which is a good choice, because you know that everything above 1 is clipping, and you have a handy relation for lower values. But actually this value is nothing but a setting, and in Csound you are free to set it to any value you like via the 0dbfs opcode. Usually you should use this statement in the orchestra header:

```
0dbfs = 1
```

This means: "Set the level for zero dB as full scale to 1 as reference value." Note that because of historical reasons the default value in Csound is not 1 but 32768. So you must have this *0dbfs = 1* statement in your header if you want to set Csound to the value probably all other audio applications have.

DB SCALE VERSUS LINEAR AMPLITUDE

Let's see some practical consequences now of what we have discussed so far. One major point is: for getting smooth transitions between intensity levels you must not use a simple linear transition of the amplitudes, but a linear transition of the dB equivalent. The following example shows a linear rise of the amplitudes from 0 to 1, and then a linear rise of the dB's from -80 to 0 dB, both over 10 seconds.

EXAMPLE 01C01.csd

```
<CsoundSynthesizer>
<CsOptions>
-odac
</CsOptions>
<CsInstruments>
;example by joachim heintz
```

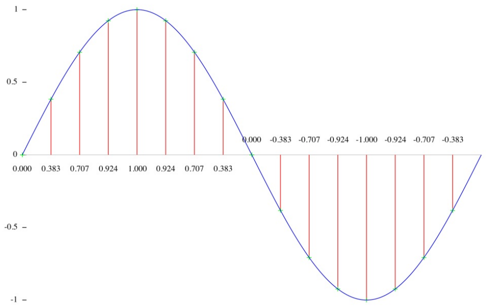
```
sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

instr 1 ;linear amplitude rise
kamp      line    0, p3, 1 ;amp rise 0->1
asig      oscils  1, 1000, 0 ;1000 Hz sine
aout      =       asig * kamp
          outs    aout, aout
endin

instr 2 ;linear rise of dB
kdb       line    -80, p3, 0 ;dB rise -60 -> 0
asig      oscils  1, 1000, 0 ;1000 Hz sine
kamp      =       ampdb(kdb) ;transformation db -> amp
aout      =       asig * kamp
          outs    aout, aout
endin

</CsInstruments>
<CsScore>
i 1 0 10
i 2 11 10
</CsScore>
</CsoundSynthesizer>
```

You will hear how fast the sound intensity increases at the first note with direct amplitude rise, and then stays nearly constant. At the second note you should hear a very smooth and constant increment of intensity.

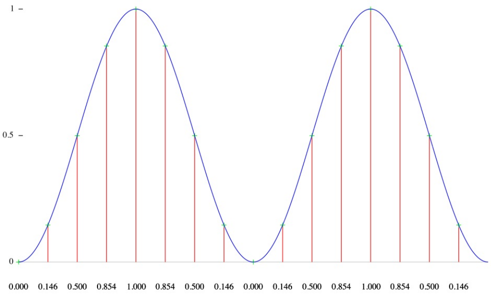
RMS MEASUREMENT

Sound intensity depends on many factors. One of the most important is the effective mean of the amplitudes in a certain time span. This is called the Root Mean Square (RMS) value. To calculate it, you have (1) to calculate the squared amplitudes of number N samples. Then you (2) divide the result by N to calculate the mean of it. Finally (3) take the square root.

Let's see a simple example, and then have a look how getting the rms value works in Csound. Assumeing we have a sine wave which consists of 16 samples, we get these amplitudes:



These are the squared amplitudes:



The mean of these values is:

(0+0.146+0.5+0.854+1+0.854+0.5+0.146+0+0.146+0.5+0.854+1+0.854+0.5+0.146)/16=8/16=0.5

And the resulting RMS value is 0.5=0.707 .

The rms opcode in Csound calculates the RMS power in a certain time span, and smoothes the values in time according to the *ihp* parameter: the higher this value (the default is 10 Hz), the snappier the measurement, and vice versa. This opcode can be used to implement a self-regulating system, in which the rms opcode prevents the system from exploding. Each time the rms value exceeds a certain value, the amount of feedback is reduced. This is an example[1] :

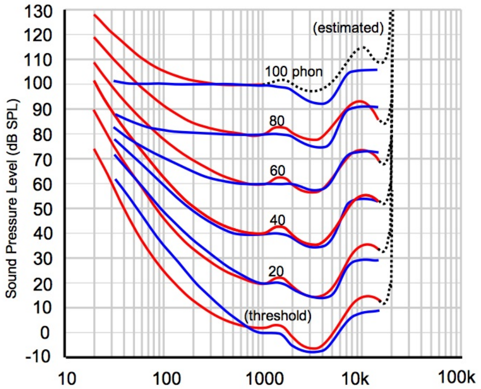
EXAMPLE 01C02.csd

```
<CsoundSynthesizer>
<CsOptions>
-odac
</CsOptions>
<CsInstruments>
;example by Martin Neukom, adapted by Joachim Heintz
sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

giSine    ftgen     0, 0, 2^10, 10, 1 ;table with a sine wave

instr 1
a3        init      0
kamp      linseg    0, 1.5, 0.2, 1.5, 0 ;envelope for initial input
asnd      poscil    kamp, 440, giSine ;initial input
 if p4 == 1 then ;choose between two sines ...
adel1     poscil    0.0523, 0.023, giSine
adel2     poscil    0.073, 0.023, giSine,.5
 else ;or a random movement for the delay lines
adel1     randi     0.05, 0.1, 2
adel2     randi     0.08, 0.2, 2
 endif
a0        delayr    1 ;delay line of 1 second
a1        deltapi   adel1 + 0.1 ;first reading
a2        deltapi   adel2 + 0.1 ;second reading
krms      rms       a3 ;rms measurement
          delayw    asnd + exp(-krms) * a3 ;feedback depending on rms
a3        reson     -(a1+a2), 3000, 7000, 2 ;calculate a3
aout      linen     a1/3, 1, p3, 1 ;apply fade in and fade out
          outs      aout, aout
endin
</CsInstruments>
<CsScore>
i 1 0 60 1 ;two sine movements of delay with feedback
i 1 61 . 2 ;two random movements of delay with feedback
</CsScore>
</CsoundSynthesizer>
```

FLETCHER-MUNSON CURVES

Human hearing is roughly in a range between 20 and 20000 Hz. But inside this range, the hearing is not equally sensitive. The most sensitive region is around 3000 Hz. If you come to the upper or lower border of the range, you need more intensity to perceive a sound as "equally loud".

These curves of equal loudness are mostly called "Fletcher-Munson Curves" because of the paper of H. Fletcher and W. A. Munson in 1933. They look like this:



Equal-loudness contours (red) (from ISO 226:2003 revision)
Fletcher–Munson curves shown (blue) for comparison

Try the following test. In the first 5 seconds you will hear a tone of 3000 Hz. Adjust the level of your amplifier to the lowest possible point at which you still can hear the tone. - Then you hear a tone whose frequency starts at 20 Hertz and ends at 20000 Hertz, over 20 seconds. Try to move the fader or knob of your amplification exactly in a way that you still can hear anything, but as soft as possible. The movement of your fader should roughly be similar to the lowest Fletcher-Munson-Curve: starting relatively high, going down and down until 3000 Hertz, and then up again. (As always, this test depends on your speaker hardware. If your speaker do not provide proper lower frequencies, you will not hear anything in the bass region.)

EXAMPLE 01C03.csd

```
<CsoundSynthesizer>
<CsOptions>
-odac
</CsOptions>
<CsInstruments>
sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

giSine    ftgen     0, 0, 2^10, 10, 1 ;table with a sine wave

instr 1
kfreq     expseg    p4, p3, p5
          printk    1, kfreq ;prints the frequencies once a second
asin      poscil    .2, kfreq, giSine
aout      linen     asin, .01, p3, .01
          outs      aout, aout
endin
</CsInstruments>
<CsScore>
i 1 0 5 1000 1000
i 1 6 20 20  20000
</CsScore>
</CsoundSynthesizer>
```

It is very important to bear in mind that the perceived loudness depends much on the frequencies. You must know that putting out a sine of 30 Hz with a certain amplitude is totally different from a sine of 3000 Hz with the same amplitude - the latter will sound much louder.

1. cf Martin Neukom, Signale Systeme Klangsynthese, Zürich 2003, p. 383^

02 QUICK START

8. MAKE CSOUND RUN

CSOUND AND FRONTENDS

The core element of Csound is an audio engine for the Csound language. It has no graphical elements and it is designed to take Csound text files (like ".csd" files) and produce audio, either in realtime, or by writing to a file. It can still be used in this way, but most users nowadays prefer to use Csound via a frontend. A frontend is an application which assists you in writing code and running Csound. Beyond the functions of a simple text editor, a frontend environment will offer colour coded highlighting of language specific keywords and quick access to an integrated help system. A frontend can also expand possibilities by providing tools to build interactive interfaces as well, sometimes, as advanced compositional tools.

In 2009 the Csound developers decided to include QuteCsound as the standard frontend to be included with the Csound distribution, so you will already have this frontend if you have installed any of the recent pre-built versions of Csound. Conversely if you install a frontend you will require a separate installation of Csound in order for it to function. If you experience any problems with QuteCsound, or simply prefer another frontend design, try WinXound as alternative.

HOW TO DOWNLOAD AND INSTALL CSOUND

To get Csound you first need to download the package for your system from the SourceForge page: http://sourceforge.net/projects/csound/files/csound5/

There are many files here, so here are some guidelines to help you choose the appropriate version.

Windows

Windows installers are the ones ending in *.exe*. Look for the latest version of Csound, and find a file which should be called something like: *Csound5.17-gnu-win32-d.exe*. The important thing to note is the final letter of the installer name, which can be "d" or "f". This specifies the computation precision of the Csound engine. Float precision (32-bit float) is marked with "f" and double precision (64-bit float) is marked "d". This is important to bear in mind, as a frontend which works with the "floats" version, will not run if you have the "doubles" version installed. More recent versions of the pre-built Windows installer have only been released in the 'doubles' version.

After you have downloaded the installer, just run it and follow the instructions. When you are finished, you will find a Csound folder in your start menu containing Csound utilities and the CsoundQt (QuteCsound) frontend.

Mac OS X

The Mac OS X installers are the files ending in *.dmg*. Look for the latest version of Csound for your particular system, for example a Universal binary for 10.7 will be called something like: *csound5.17.3-OSX10.7-Universal.dmg*. When you double click the downloaded file, you will have a disk image on your desktop, with the Csound installer, CsoundQt and a readme file. Double-click the installer and follow the instructions. Csound and the basic Csound utilities will be installed. To install the CsoundQt frontend, you only need to move it to your Applications folder.

Linux and others

Csound is available from the official package repositories for many distributions like Debian, Ubuntu, Fedora, Archlinux and Gentoo. If there are no binary packages for your platform, or you need a more recent version, you can get the source package from the SourceForge page and build from source. Some build instructions can be find in the chapter BUILDING CSOUND in the appendix, and in the Csound Wiki on Sourceforge. Detailed information can also be found in the Building Csound Manual Page.

Note that the Csound repository has moved from cvs to git. After installing git, you can use this command to clone the Csound5 repository, if you like to have access to the latest (perhaps unstable) sources:

```
git clone git://csound.git.sourceforge.net/gitroot/csound/csound5
```

Android and iOS

Recently Csound has been ported to Android and iOS. At the time of writing this release, it is too early for a description. If you are interested, you may have a look at http://sourceforge.net/projects/csound/files/csound5 or at the paper from Victor Lazzarini and Steven Yi at the 2012 Linux Audio Conference.

INSTALL PROBLEMS?

If, for any reason, you can't find the CsoundQt (formerly QuteCsound) frontend on your system after install, or if you want to install the most recent version of CsoundQt, or if you prefer another frontend altogether: see the CSOUND FRONTENDS section of this manual for further information. If you have any install problems, consider joining the Csound Mailing List to report your issues, or write a mail to one of the maintainers (see ON THIS RELEASE).

THE CSOUND REFERENCE MANUAL

The Csound Reference Manual is an indispensable companion to Csound. It is available in various formats from the same place as the Csound installers, and it is installed with the packages for OS X and Windows. It can also be browsed online at The Csound Manual Section at Csounds.com. Many frontends will provide you with direct and easy access to it.

HOW TO EXECUTE A SIMPLE EXAMPLE

Using CsoundQt

Run CsoundQt. Go into the CsoundQt menubar and choose: Examples->Getting started...-> Basics-> HelloWorld

You will see a very basic Csound file (.csd) with a lot of comments in green.

Click on the "RUN" icon in the CsoundQt control bar to start the realtime Csound engine. You should hear a 440 Hz sine wave.

You can also run the Csound engine in the terminal from within QuteCsound. Just click on "Run in Term". A console will pop up and Csound will be executed as an independent process. The result should be the same - the 440 Hz "beep".

Using the Terminal / Console

1. Save the following code in any plain text editor as HelloWorld.csd.

EXAMPLE 02A01.csd

```
<CsoundSynthesizer>
<CsOptions>
-odac
</CsOptions>
<CsInstruments>
;Example by Alex Hofmann
instr 1
aSin      oscils    0dbfs/4, 440, 0
          out       aSin
endin
</CsInstruments>
<CsScore>
i 1 0 1
</CsScore>
</CsoundSynthesizer>
```

2. Open the Terminal / Prompt / Console

3. Type: *csound /full/path/HelloWorld.csd*

where */full/path/HelloWorld.csd* is the complete path to your file. You also execute this file by just typing *csound* then dragging the file into the terminal window and then hitting return.

You should hear a 440 Hz tone.

9. CSOUND SYNTAX

ORCHESTRA AND SCORE

In Csound, you must define "instruments", which are units which "do things", for instance playing a sine wave. These instruments must be called or "turned on" by a "score". The Csound "score" is a list of events which describe how the instruments are to be played in time. It can be thought of as a timeline in text.

A Csound instrument is contained within an Instrument Block, which starts with the keyword instr and ends with the keyword endin. All instruments are given a number (or a name) to identify them.

```
instr 1
... instrument instructions come here...
endin
```

Score events in Csound are individual text lines, which can turn on instruments for a certain time. For example, to turn on instrument 1, at time 0, for 2 seconds you will use:

```
i 1 0 2
```

THE CSOUND DOCUMENT STRUCTURE

A Csound document is structured into three main sections:

- **CsOptions**: Contains the configuration options for Csound. For example using "-o dac" in this section will make Csound run in real-time instead of writing a sound file.
- **CsInstruments**: Contains the instrument definitions and optionally some global settings and definitions like sample rate, etc.
- **CsScore**: Contains the score events which trigger the instruments.

Each of these sections is opened with a <xyz> tag and closed with a </xyz> tag. Every Csound file starts with the <CsoundSynthesizer> tag, and ends with </CsoundSynthesizer>. Only the text in-between will be used by Csound.

EXAMPLE 02B01.csd

```
<CsoundSynthesizer>; START OF A CSOUND FILE

<CsOptions> ; CSOUND CONFIGURATION
-odac
</CsOptions>

<CsInstruments> ; INSTRUMENT DEFINITIONS GO HERE

; Set the audio sample rate to 44100 Hz
sr = 44100

instr 1
; a 440 Hz Sine Wave
aSin      oscils    0dbfs/4, 440, 0
          out       aSin
endin
</CsInstruments>

<CsScore> ; SCORE EVENTS GO HERE
i 1 0 1
</CsScore>

</CsoundSynthesizer> ; END OF THE CSOUND FILE
; Anything after is ignored by Csound
```

Comments, which are lines of text that Csound will ignore, are started with the ";" character. Multi-line comments can be made by encasing them between "/*" and "*/".

OPCODES

"Opcodes" or "Unit generators" are the basic building blocks of Csound. Opcodes can do many things like produce oscillating signals, filter signals, perform mathematical functions or even turn on and off instruments. Opcodes, depending on their function, will take inputs and outputs. Each input or output is called, in programming terms, an "argument". Opcodes always take input arguments on the right and output their results on the left, like this:

```
output    OPCODE    input1, input2, input3, .., inputN
```

For example the oscils opcode has three inputs: amplitude, frequency and phase, and produces a sine wave signal:

```
aSin      oscils    0dbfs/4, 440, 0
```

In this case, a 440 Hertz oscillation starting at phase 0 radians, with an amplitude of *0dbfs/4* (a quarter of 0 dB as full scale) will be created and its output will be stored in a container called *aSin*. The order of the arguments is important: the first input to *oscils* will always be amplitude, the second, frequency and the third, phase.

Many opcodes include optional input arguments and occasionally optional output arguments. These will always be placed after the essential arguments. In the Csound Manual documentation they are indicated using square brackets "[]". If optional input arguments are omitted they are replaced with the default values indicated in the Csound Manual. The addition of optional output arguments normally initiates a different mode of that opcode: for example, a stereo as opposed to mono version of the opcode.

VARIABLES

A "variable" is a named container. It is a place to store things like signals or values from where they can be recalled by using their name. In Csound there are various types of variables. The easiest way to deal with variables when getting to know Csound is to imagine them as cables.

If you want to patch this together: Oscillator->Filter->Output,

you need two cables, one going out from the oscillator into the filter and one from the filter to the output. The cables carry audio signals, which are variables beginning with the letter "a".

```
aSource    buzz       0.8, 200, 10, 1
aFiltered  moogladder aSource, 400, 0.8
           out        aFiltered
```

In the example above, the buzz opcode produces a complex waveform as signal *aSource*. This signal is fed into the moogladder opcode, which in turn produces the signal *aFiltered*. The out opcode takes this signal, and sends it to the output whether that be to the speakers or to a rendered file.

Other common variable types are "k" variables which store control signals, which are updated less frequently than audio signals, and "i" variables which are constants within each instrument note.

You can find more information about variable types here in this manual, or here in the Csound Journal.

USING THE MANUAL

The Csound Reference Manual is a comprehensive source regarding Csound's syntax and opcodes. All opcodes have their own manual entry describing their syntax and behavior, and the manual contains a detailed reference on the Csound language and options.

In CsoundQt you can find the Csound Manual in the Help Menu. You can quickly go to a particular opcode entry in the manual by putting the cursor on the opcode and pressing Shift+F1. WinXsound and Blue also provide easy access to the manual.

10. CONFIGURING MIDI

Csound can receive MIDI events (like MIDI notes and MIDI control changes) from an external MIDI interface or from another program via a virtual MIDI cable. This information can be used to control any aspect of synthesis or performance.

Csound receives MIDI data through MIDI Realtime Modules. These are special Csound plugins which enable MIDI input using different methods according to platform. They are enabled using the *-+rtmidi* command line flag in the *<CsOptions>* section of your .csd file, but can also be set interactively on some front-ends via the configure dialog setups.

There is the universal "portmidi" module. PortMidi is a cross-platform module for MIDI I/O and should be available on all platforms. To enable the "portmidi" module, you can use the flag:

```
-+rtmidi=portmidi
```

After selecting the RT MIDI module from a front-end or the command line, you need to select the MIDI devices for input and output. These are set using the flags -M and -Q respectively followed by the number of the interface. You can usually use:

```
-M999
```

To get a performance error with a listing of available interfaces.

For the PortMidi module (and others like ALSA), you can specify no number to use the default MIDI interface or the 'a' character to use all devices. This will even work when no MIDI devices are present.

```
-Ma
```

So if you want MIDI input using the portmidi module, using device 2 for input and device 1 for output, your *<CsOptions>* section should contain:

```
-+rtmidi=portmidi -M2 -Q1
```

There is a special "virtual" RT MIDI module which enables MIDI input from a virtual keyboard. To enable it, you can use:

```
 -+rtmidi=virtual -M0
```

PLATFORM SPECIFIC MODULES

If the "portmidi" module is not working properly for some reason, you can try other platform specific modules.

Linux

On Linux systems, you might also have an "alsa" module to use the alsa raw MIDI interface. This is different from the more common alsa sequencer interface and will typically require the snd-virmidi module to be loaded.

OS X

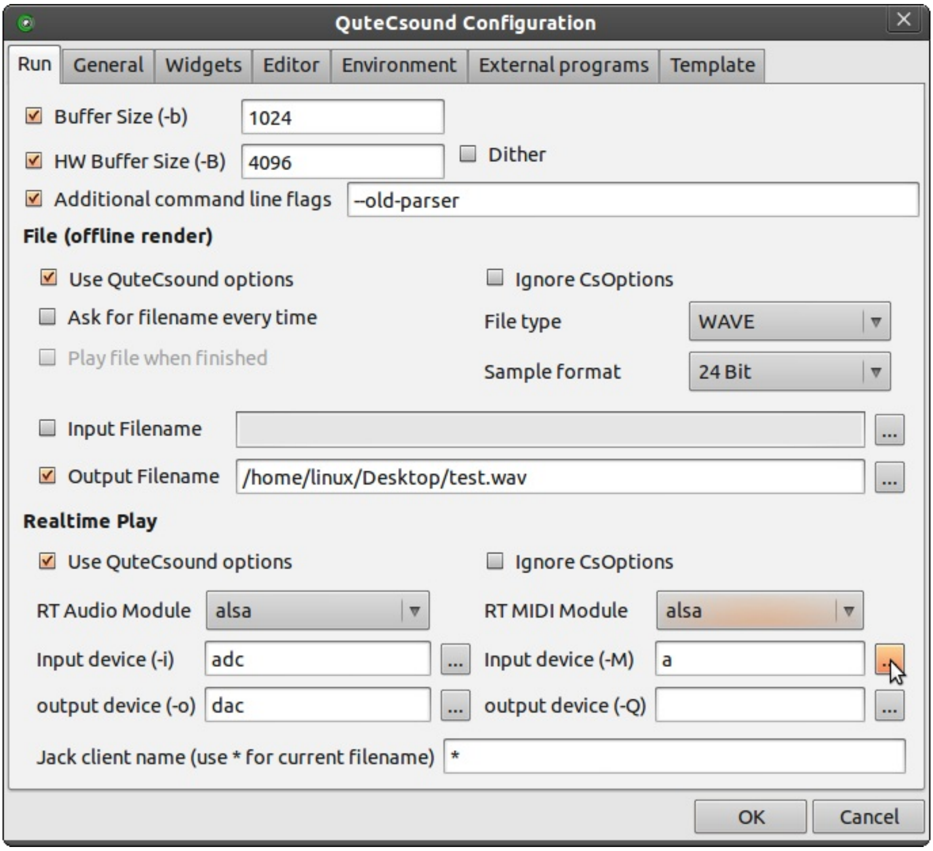
On OS X you may have a "coremidi" module available.

Windows

On Windows, you may have a "winmme" MIDI module.

MIDI I/O IN CSOUNDQT

As with Audio I/O, you can set the MIDI preferences in the configuration dialog. In it you will find a selection box for the RT MIDI module, and text boxes for MIDI input and output devices.



☑ Buffer Size (-b) 1024

☑ HW Buffer Size (-B) 4096 ☐ Dither

☑ Additional command line flags –old-parser

File (offline render)

☑ Use QuteCsound options ☐ Ignore CsOptions

☐ Ask for filename every time

File type WAVE

☐ Play file when finished

Sample format 24 Bit

☐ Input Filename

☑ Output Filename /home/linux/Desktop/test.wav

Realtime Play

☑ Use QuteCsound options ☐ Ignore CsOptions

RT Audio Module alsa

RT MIDI Module alsa

Input device (-i) adc

Input device (-M) a

output device (-o) dac

output device (-Q)

Jack client name (use * for current filename) *

HOW TO USE A MIDI KEYBOARD

Once you've set up the hardware, you are ready to receive MIDI information and interpret it in Csound. By default, when a MIDI note is received, it turns on the Csound instrument corresponding to its channel number, so if a note is received on channel 3, it will turn on instrument 3, if it is received on channel 10, it will turn on instrument 10 and so on.

If you want to change this routing of MIDI channels to instruments, you can use the massign opcode. For instance, this statement lets you route your MIDI channel 1 to instrument 10:

```
 massign 1, 10
```

On the following example, a simple instrument, which plays a sine wave, is defined in instrument 1. There are no score note events, so no sound will be produced unless a MIDI note is received on channel 1.

EXAMPLE 02C01.csd

```
<CsoundSynthesizer>
<CsOptions>
-+rtmidi=portmidi -Ma -odac
</CsOptions>
<CsInstruments>
;Example by Andrés Cabrera

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

        massign   0, 1 ;assign all MIDI channels to instrument 1
giSine  ftgen     0,0,2^10,10,1 ;a function table with a sine wave

instr 1
iCps    cpsmidi   ;get the frequency from the key pressed
iAmp    ampmidi   0dbfs * 0.3 ;get the amplitude
aOut    poscil    iAmp, iCps, giSine ;generate a sine tone
```

```
        outs      aOut, aOut ;write it to the output
endin

</CsInstruments>
<CsScore>
e 3600
</CsScore>
</CsoundSynthesizer>
```

Note that Csound has an unlimited polyphony in this way: each key pressed starts a new instance of instrument 1, and you can have any number of instrument instances at the same time.

HOW TO USE A MIDI CONTROLLER

To receive MIDI controller events, opcodes like ctrl7 can be used. In the following example instrument 1 is turned on for 60 seconds. It will receive controller #1 (modulation wheel) on channel 1 and convert MIDI range (0-127) to a range between 220 and 440. This value is used to set the frequency of a simple sine oscillator.

EXAMPLE 02C02.csd

```
<CsoundSynthesizer>
<CsOptions>
-+rtmidi=virtual -M1 -odac
</CsOptions>
<CsInstruments>
;Example by Andrés Cabrera

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

giSine ftgen 0,0,2^10,10,1

instr 1
; --- receive controller number 1 on channel 1 and scale from 220 to 440
kFreq ctrl7  1, 1, 220, 440
; --- use this value as varying frequency for a sine wave
aOut  poscil 0.2, kFreq, giSine
      outs   aOut, aOut
endin
</CsInstruments>
<CsScore>
i 1 0 60
e
</CsScore>
</CsoundSynthesizer>
```

OTHER TYPE OF MIDI DATA

Csound can receive other type of MIDI, like pitch bend, and aftertouch through the usage of specific opcodes. Generic MIDI Data can be received using the midiin opcode. The example below prints to the console the data received via MIDI.

EXAMPLE 02C03.csd

```
<CsoundSynthesizer>
<CsOptions>
-+rtmidi=portmidi -Ma -odac
</CsOptions>
<CsInstruments>
;Example by Andrés Cabrera

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

instr 1
kStatus, kChan, kData1, kData2 midiin

if kStatus != 0 then ;print if any new MIDI message has been received
    printk 0, kStatus
    printk 0, kChan
    printk 0, kData1
    printk 0, kData2
```

```
endif

endin

</CsInstruments>
<CsScore>
i1 0 3600
e
</CsScore>
</CsoundSynthesizer>
```

11. LIVE AUDIO

CONFIGURING AUDIO & TUNING AUDIO PERFORMANCE

Selecting Audio Devices and Drivers

Csound relates to the various inputs and outputs of sound devices installed on your computer as a numbered list. If you are using a multichannel interface then each stereo pair will most likely be assigned a different number. If you wish to send or receive audio to or from a specific audio connection you will need to know the number by which Csound knows it. If you are not sure of what that is you can trick Csound into providing you with a list of available devices by trying to run Csound using an obviously out of range device number, like this:

EXAMPLE 02D01.csd

```
<CsoundSynthesizer>
<CsOptions>
-iadc999 -odac999
</CsOptions>
<CsInstruments>
;Example by Andrés Cabrera
instr 1
endin
</CsInstruments>
<CsScore>
e
</CsScore>
</CsoundSynthesizer>
```

The input and output devices will be listed seperately. Specify your input device with the -**iadc** flag and the number of your input device, and your output device with the -**odac** flag and the number of your output device. For instance, if you select the "XYZ" device from the list above both, for input and output, you may include something like

```
 -iadc2 -odac3
```

in the <CsOptions> section of you .csd file.

The RT (= real-time) output module can be set with the -**+rtaudio** flag. If you don't use this flag, the PortAudio driver will be used. Other possible drivers are jack and alsa (Linux), mme (Windows) or CoreAudio (Mac). So, this sets your audio driver to mme instead of Port Audio:

```
-+rtaudio=mme
```

Tuning Performance and Latency

Live performance and latency depend mainly on the sizes of the software and the hardware buffers. They can be set in the <CsOptions> using the -B flag for the hardware buffer, and the -b flag for the software buffer. For instance, this statement sets the hardware buffer size to 512 samples and the software buffer size to 128 sample:

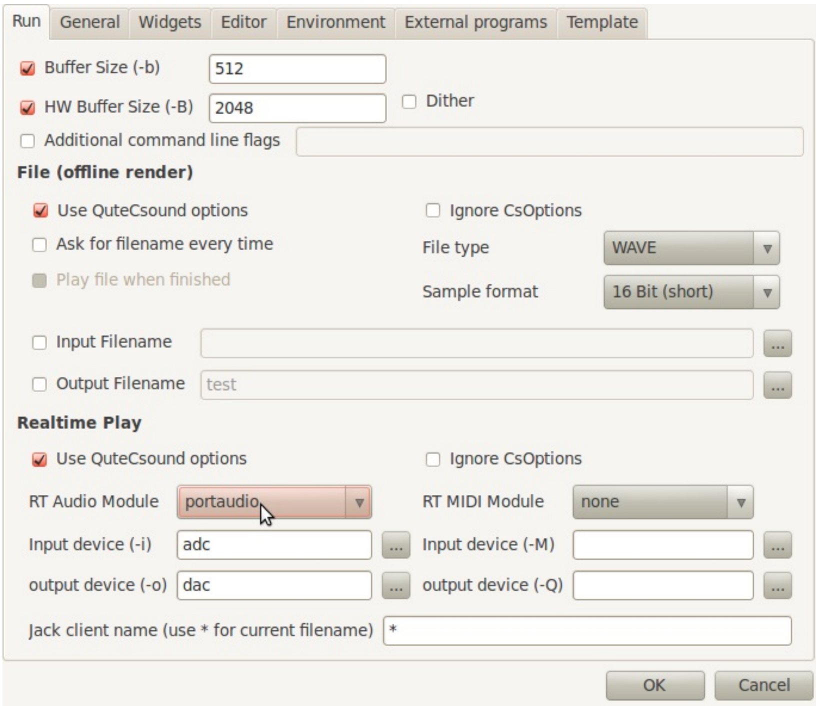
```
-B512 -b128
```

The other factor which affects Csound's live performance is the ksmps value which is set in the header of the <CsInstruments> section. By this value, you define how many samples are processed every Csound control cycle.

Try your realtime performance with -B512, -b128 and ksmps=32. With a software buffer of 128 samples, a hardware buffer of 512 and a sample rate of 44100 you will have around 12ms latency, which is usable for live keyboard playing. If you have problems with either the latency or the performance, tweak the values as described here.

CsoundQt

To define the audio hardware used for realtime performance, open the configuration dialog. In the "Run" Tab, you can choose your audio interface, and the preferred driver. You can select input and output devices from a list if you press the buttons to the right of the text boxes for input and output names. Software and hardware buffer sizes can be set at the top of this dialogue box.



☑ Buffer Size (-b) 512
☑ HW Buffer Size (-B) 2048 ☐ Dither
☐ Additional command line flags

File (offline render)

☑ Use QuteCsound options ☐ Ignore CsOptions
☐ Ask for filename every time File type WAVE
☐ Play file when finished Sample format 16 Bit (short)
☐ Input Filename
☐ Output Filename test

Realtime Play

☑ Use QuteCsound options ☐ Ignore CsOptions
RT Audio Module portaudio RT MIDI Module none
Input device (-i) adc Input device (-M)
output device (-o) dac output device (-Q)
Jack client name (use * for current filename) *

CSOUND CAN PRODUCE EXTREME DYNAMIC RANGE!

Csound can **produce extreme dynamic range**, so keep an eye on the level you are sending to your output. The number which describes the level of 0 dB, can be set in Csound by the 0dbfs assignment in the <CsInstruments> header. There is no limitation, if you set 0dbfs = 1 and send a value of 32000, *this can damage your ears and speakers!*

USING LIVE AUDIO INPUT AND OUTPUT

To process audio from an external source (for example a microphone), use the inch opcode to access any of the inputs of your audio input device. For the output, outch gives you all necessary flexibility. The following example takes a live audio input and transforms its sound using ring modulation. The Csound Console should output five times per second the input amplitude level.

EXAMPLE 02D02.csd

```
<CsoundSynthesizer>
<CsOptions>
;CHANGE YOUR INPUT AND OUTPUT DEVICE NUMBER HERE IF NECESSARY!
-iadc0 -odac0 -B512 -b128
</CsOptions>
<CsInstruments>
;Example by Joachim Heintz
sr = 44100 ;set sample rate to 44100 Hz
ksmps = 32 ;number of samples per control cycle
nchnls = 2 ;use two audio channels
0dbfs = 1 ;set maximum level as 1

giSine    ftgen     0, 0, 2^10, 10, 1 ;table with sine wave

instr 1
aIn       inch      1   ;take input from channel 1
kInLev    downsamp  aIn ;convert audio input to control signal
          printk    .2, abs(kInLev)
;make modulator frequency oscillate 200 to 1000 Hz
```

```
kModFreq  poscil    400, 1/2, giSine
kModFreq  =         kModFreq+600
aMod      poscil    1, kModFreq, giSine ;modulator signal
aRM       =         aIn * aMod ;ring modulation
          outch     1, aRM, 2, aRM ;output to channel 1 and 2
endin
</CsInstruments>
<CsScore>
i 1 0 3600
</CsScore>
</CsoundSynthesizer>
```

Live Audio is frequently used with live devices like widgets or MIDI. In CsoundQt, you can find several examples in Examples -> Getting Started -> Realtime Interaction.

12. RENDERING TO FILE

WHEN TO RENDER TO FILE

Csound can also render audio straight to a sound file stored on your hard drive instead of as live audio sent to the audio hardware. This gives you the possibility to hear the results of very complex processes which your computer can't produce in realtime.

Csound can render to formats like wav, aiff or ogg (and other less popular ones), but not mp3 due to its patent and licencing problems.

RENDERING TO FILE

Save the following code as Render.csd:

EXAMPLE 02E01.csd

```
<CsoundSynthesizer>
<CsOptions>
-o Render.wav
</CsOptions>
<CsInstruments>
;Example by Alex Hofmann
instr 1
aSin      oscils    0dbfs/4, 440, 0
          out       aSin
endin
</CsInstruments>
<CsScore>
i 1 0 1
e
</CsScore>
</CsoundSynthesizer>
```

Open the Terminal / Prompt / Console and type:

```
csound /path/to/Render.csd
```

Now, because you changed the -**o** flag in the <CsOptions> from "-o dac" to "-o *filename*", the audio output is no longer written in realtime to your audio device, but instead to a file. The file will be rendered to the default directory (usually the user home directory). This file can be opened and played in any audio player or editor, e.g. Audacity. (By default, csound is a non-realtime program. So if no command line options are given, it will always render the csd to a file called *test.wav*, and you will hear nothing in realtime.)

The -**o** flag can also be used to write the output file to a certain directory. Something like this for Windows ...

```
<CsOptions>
-o c:/music/samples/Render.wav
</CsOptions>
```

... and this for Linux or Mac OSX:

```
<CsOptions>
-o /Users/JSB/organ/tatata.wav
</CsOptions>
```

Rendering Options

The internal rendering of audio data in Csound is done with 32-bit floating point numbers (or even with 64-bit numbers for the "double" version). Depending on your needs, you should decide the precision of your rendered output file:

- If you want to render 32-bit floats, use the option flag -**f**.
- If you want to render 24-bit, use the flag -**3**.
- If you want to render 16-bit, use the flag -**s** (or nothing, because this is also the default in Csound).

For making sure that the header of your soundfile will be written correctly, you should use the -**W** flag for a WAV file, or the -**A** flag for a AIFF file. So these options will render the file "Wow.wav" as WAV file with 24-bit accuracy:

```
<CsOptions>
-o Wow.wav -W -3
</CsOptions>
```

Realtime and Render-To-File at the Same Time

Sometimes you may want to simultaneously have realtime output and file rendering to disk, like recording your live performance. This can be achieved by using the fout opcode. You just have to specify your output file name. File type and format are given by a number, for instance 18 specifies "wav 24 bit" (see the manual page for more information). The following example creates a random frequency and panning movement of a sine wave, and writes it to the file "live_record.wav" (in the same directory as your .csd file):

EXAMPLE 02E02.csd

```
<CsoundSynthesizer>
<CsOptions>
-odac
</CsOptions>
<CsInstruments>
;Example by Joachim Heintz
sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

          seed      0 ;each time different seed for random
giSine    ftgen     0, 0, 2^10, 10, 1 ;a sine wave

  instr 1
kFreq     randomi   400, 800, 1 ;random frequency
aSig      poscil    .2, kFreq, giSine ;sine with this frequency
kPan      randomi   0, 1, 1 ;random panning
aL, aR    pan2      aSig, kPan ;stereo output signal
          outs      aL, aR ;live output
          fout      "live_record.wav", 18, aL, aR ;write to soundfile
  endin

</CsInstruments>
<CsScore>
i 1 0 10
e
</CsScore>
</CsoundSynthesizer>
```

CsoundQt

All the options which are described in this chapter can be handled very easily in CsoundQt:

- Rendering to file is simply done by clicking the "Render" button, or choosing "Control->Render to File" in the Menu.
- To set file-destination and file-type, you can make your own settings in "CsoundQt Configuration" under the tab "Run -> File (offline render)". The default is a 16-Bit .wav-file.
- To record a live performance, just click the "Record" button. You will find a file with the same name as your .csd file, and a number appended for each record task, in the same folder as your .csd file.

03 CSOUND LANGUAGE

13. INITIALIZATION AND PERFORMANCE PASS

WHAT'S THE DIFFERENCE

A Csound instrument is defined in the <CsInstruments> section of a .csd file. An instrument definition starts with the keyword instr (followed by a number or name to identify the instrument), and ends with the line endin. Each instrument can be called by a score event which starts with the character "i". For instance, this score line

```
i 1 0 3
```

calls instrument 1, starting at time 0, for 3 seconds. It is very important to understand that such a call consists of two different stages: the initialization and the performance pass.

At first, Csound initializes all the variables which begin with a **i** or a **gi**. This initialization pass is done just once.

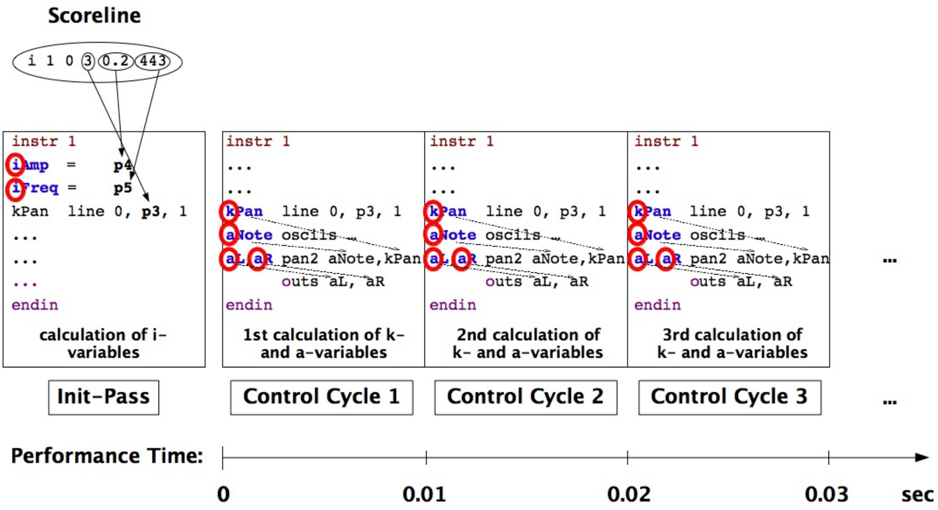
After this, the actual performance begins. During this performance, Csound calculates all the time-varying values in the orchestra again and again. This is called the performance pass, and each of these calculations is called a control cycle (also abbreviated as k-cycle or k-loop). The time for each control cycle depends on the ksmps constant in the orchestra header. If ksmps=10 (which is the default), the performance pass consists of 10 samples. If your sample rate is 44100, with ksmps=10 you will have 4410 control cycles per second (kr=4410), and each of them has a duration of 1/4410 = 0.000227 seconds. On each control cycle, all the variables starting with **k**, **gk**, **a** and **ga** are updated (see the next chapter about variables for more explanations).

This is an example instrument, containing i-, k- and a-variables:

EXAMPLE 03A01.csd

```
<CsoundSynthesizer>
<CsOptions>
-o dac
</CsOptions>
<CsInstruments>
;Example by Joachim Heintz
sr = 44100
ksmps = 441
nchnls = 2
0dbfs = 1
instr 1
iAmp      =       p4 ;amplitude taken from the 4th parameter of the score line
iFreq     =       p5 ;frequency taken from the 5th parameter
; --- move from 0 to 1 in the duration of this instrument call (p3)
kPan      line      0, p3, 1
aNote     oscils  iAmp, iFreq, 0 ;create an audio signal
aL, aR    pan2    aNote, kPan ;let the signal move from left to right
          outs    aL, aR ;write it to the output
endin
</CsInstruments>
<CsScore>
i 1 0 3 0.2 443
</CsScore>
</CsoundSynthesizer>
```

As ksmps=441, each control cycle is 0.01 seconds long (441/44100). So this happens when the instrument call is performed:



Here is another simple example which shows the internal loop at each k-cycle. As we print out the value at each control cycle, ksmps is very high here, so that each k-pass takes 0.1 seconds. The init opcode can be used to set a k-variable to a certain value first (at the init-pass), otherwise it will have the default value of zero until it is assigned something else during the first k-cycle.

EXAMPLE 03A02.csd

```
<CsoundSynthesizer>
<CsInstruments>
;Example by Joachim Heintz
sr = 44100
ksmps = 4410

instr 1
kcount    init      0; set kcount to 0 first
kcount    =         kcount + 1; increase at each k-pass
          printk    0, kcount; print the value
endin

</CsInstruments>
<CsScore>
i 1 0 1
</CsScore>
</CsoundSynthesizer>
```

Your output should contain the lines:

```
i   1 time     0.10000:     1.00000
i   1 time     0.20000:     2.00000
i   1 time     0.30000:     3.00000
i   1 time     0.40000:     4.00000
i   1 time     0.50000:     5.00000
i   1 time     0.60000:     6.00000
i   1 time     0.70000:     7.00000
i   1 time     0.80000:     8.00000
i   1 time     0.90000:     9.00000
i   1 time     1.00000:    10.00000
```

Try changing the ksmps value from 4410 to 44100 and to 2205 and observe the difference.

REINITIALIZATION

If you try the example above with i-variables, you will have no success, because the i-variable is calculated just once:

EXAMPLE 03A03.csd

```
<CsoundSynthesizer>
```

```
<CsInstruments>
;Example by Joachim Heintz
sr = 44100
ksmps = 4410

instr 1
icount    init      0          ;set icount to 0 first
icount    =         icount + 1 ;increase
          print     icount     ;print the value
endin

</CsInstruments>
<CsScore>
i 1 0 1
</CsScore>
</CsoundSynthesizer>
```

The printout is:

```
instr 1:  icount = 1.000
```

Nevertheless it is possible to refresh even an i-rate variable in Csound. This is done with the reinit opcode. You must mark a section by a label (any name followed by a colon). Then the reinit statement will cause the i-variable to refresh. Use rireturn to end the reinit section.

EXAMPLE 03A04.csd

```
<CsoundSynthesizer>
<CsInstruments>
;Example by Joachim Heintz
sr = 44100
ksmps = 4410

instr 1
icount    init      0          ; set icount to 0 first
new:
icount    =         icount + 1 ; increase
          print     icount     ; print the value
          reinit    new        ; reinit the section each k-pass
          rireturn
endin

</CsInstruments>
<CsScore>
i 1 0 1
</CsScore>
</CsoundSynthesizer>
```

This prints now:

```
instr 1:  icount = 1.000
instr 1:  icount = 2.000
instr 1:  icount = 3.000
instr 1:  icount = 4.000
instr 1:  icount = 5.000
instr 1:  icount = 6.000
instr 1:  icount = 7.000
instr 1:  icount = 8.000
instr 1:  icount = 9.000
instr 1:  icount = 10.000
instr 1:  icount = 11.000
```

ORDER OF CALCULATION

Sometimes it is very important to observe the order in which the instruments of a Csound orchestra are evaluated. This order is given by the instrument numbers. So, if you want to use during the same performance pass a value in instrument 10 which is generated by another instrument, you must not give this instrument the number 11 or higher. In the following example, first instrument 10 uses a value of instrument 1, then a value of instrument 100.

EXAMPLE 03A05.csd

```
<CsoundSynthesizer>
<CsInstruments>
;Example by Joachim Heintz
sr = 44100
```

```
ksmps = 4410

instr 1
gkcount   init      0 ;set gkcount to 0 first
gkcount   =         gkcount + 1 ;increase
endin

instr 10
          printk    0, gkcount ;print the value
endin

instr 100
gkcount   init      0 ;set gkcount to 0 first
gkcount   =         gkcount + 1 ;increase
endin


</CsInstruments>
<CsScore>
;first i1 and i10
i 1 0 1
i 10 0 1
;then i100 and i10
i 100 1 1
i 10 1 1
</CsScore>
</CsoundSynthesizer>
```

The output shows the difference:

```
new alloc for instr 1:
new alloc for instr 10:
 i  10 time     0.10000:     1.00000
 i  10 time     0.20000:     2.00000
 i  10 time     0.30000:     3.00000
 i  10 time     0.40000:     4.00000
 i  10 time     0.50000:     5.00000
 i  10 time     0.60000:     6.00000
 i  10 time     0.70000:     7.00000
 i  10 time     0.80000:     8.00000
 i  10 time     0.90000:     9.00000
 i  10 time     1.00000:    10.00000
B  0.000 ..  1.000 T  1.000 TT  1.000 M:      0.0
new alloc for instr 100:
 i  10 time     1.10000:     0.00000
 i  10 time     1.20000:     1.00000
 i  10 time     1.30000:     2.00000
 i  10 time     1.50000:     4.00000
 i  10 time     1.60000:     5.00000
 i  10 time     1.70000:     6.00000
 i  10 time     1.80000:     7.00000
 i  10 time     1.90000:     8.00000
 i  10 time     2.00000:     9.00000
B  1.000 ..  2.000 T  2.000 TT  2.000 M:      0.0
```

ABOUT "I-TIME" AND "K-RATE" OPCODES

It is often confusing for the beginner that there are some opcodes which only work at "i-time" or "i-rate", and others which only work at "k-rate" or "k-time". For instance, if the user wants to print the value of any variable, he thinks: "OK - print it out." But Csound replies: "Please, tell me first if you want to print an i- or a k-variable" (see the following section about the variable types).

For instance, the print opcode just prints variables which are updated at each initialization pass ("i-time" or "i-rate"). If you want to print a variable which is updated at each control cycle ("k-rate" or "k-time"), you need its counterpart printk. (As the performance pass is usually updated some thousands times per second, you have an additional parameter in printk, telling Csound how often you want to print out the k-values.)

So, some opcodes are just for i-rate variables, like filelen or ftgen. Others are just for k-rate variables like metro or max_k. Many opcodes have variants for either i-rate-variables or k-rate-variables, like printf_i and printf, sprintf and sprintfk, strindex and strindexk.

Most of the Csound opcodes are able to work either at i-time or at k-time or at audio-rate, but you have to think carefully what you need, as the behaviour will be very different if you choose the i-, k- or a-variante of an opcode. For example, the random opcode can work at all three rates:

```
ires      random    imin, imax : works at "i-time"
kres      random    kmin, kmax : works at "k-rate"
ares      random    kmin, kmax : works at "audio-rate"
```

If you use the i-rate random generator, you will get one value for each note. For instance, if you want to have a different pitch for each note you are generating, you will use this one.

If you use the k-rate random generator, you will get one new value on every control cycle. If your sample rate is 44100 and your ksmps=10, you will get 4410 new values per second! If you take this as pitch value for a note, you will hear nothing but a noisy jumping. If you want to have a moving pitch, you can use the randomi variant of the k-rate random generator, which can reduce the number of new values per second, and interpolate between them.

If you use the a-rate random generator, you will get as many new values per second as your sample rate is. If you use it in the range of your 0 dB amplitude, you produce white noise.

EXAMPLE 03A06.csd

```
<CsoundSynthesizer>
<CsOptions>
-odac
</CsOptions>
<CsInstruments>
;Example by Joachim Heintz
sr = 44100
ksmps = 32
0dbfs = 1
nchnls = 2

          seed      0 ;each time different seed
giSine    ftgen     0, 0, 2^10, 10, 1 ;sine table

instr 1 ;i-rate random
iPch      random    300, 600
aAmp      linseg    .5, p3, 0
aSine     poscil    aAmp, iPch, giSine
          outs      aSine, aSine
endin

instr 2 ;k-rate random: noisy
kPch      random    300, 600
aAmp      linseg    .5, p3, 0
aSine     poscil    aAmp, kPch, giSine
          outs      aSine, aSine
endin

instr 3 ;k-rate random with interpolation: sliding pitch
kPch      randomi   300, 600, 3
aAmp      linseg    .5, p3, 0
aSine     poscil    aAmp, kPch, giSine
          outs      aSine, aSine
endin

instr 4 ;a-rate random: white noise
aNoise    random    -.1, .1
          outs      aNoise, aNoise
endin

</CsInstruments>
<CsScore>
i 1 0   .5
i 1 .25 .5
i 1 .5  .5
i 1 .75 .5
i 2 2   1
i 3 4   2
i 3 5   2
i 3 6   2
i 4 9   1
</CsScore>
</CsoundSynthesizer>
```

TIMELESSNESS AND TICK SIZE IN CSOUND

In a way it is confusing to speak from "i-time". For Csound, "time" actually begins with the first performance pass. The initalization time is actually the "time zero". Regardless how much human time or CPU time is needed for the initialization pass, the Csound clock does not move at all. This is the reason why you can use any i-time opcode with a zero duration (p3) in the score:

EXAMPLE 03A07.csd

```
<CsoundSynthesizer>
<CsInstruments>
;Example by Joachim Heintz
instr 1
prints "%nHello Eternity!%n%n"
endin
</CsInstruments>
<CsScore>
i 1 0 0 ;let instrument 1 play for zero seconds ...
</CsScore>
</CsoundSynthesizer>
```

Csound's clock is the control cycle. The number of samples in one control cycle - given by the ksmps value - is the smallest possible "tick" in Csound at k-rate. If your sample rate is 44100, and you have 4410 samples in one control cycle (ksmps=4410), you will not be able to start a k-event faster than each 1/10 second, because there is no k-time for Csound "between" two control cycles. Try the following example with larger and smaller ksmps values:

EXAMPLE 03A08.csd

```
<CsoundSynthesizer>
<CsOptions>
</CsOptions>
<CsInstruments>
;Example by Joachim Heintz
sr = 44100
ksmps = 4410; try 44100 or 2205 instead

instr 1; prints the time once in each control cycle
kTimek   timek
kTimes   times
         printks    "Number of control cycles = %d%n", 0, kTimek
         printks    "Time = %f%n%n", 0, kTimes
endin
</CsInstruments>
<CsScore>
i 1 0 10
</CsScore>
</CsoundSynthesizer>
```

Consider typical size of 32 for ksmps. When sample rate is 44100, a single tick will be less than a millisecond. This should be sufficient for in most situations. If you need a more accurate time resolution, just decrease the ksmps value. The cost of this smaller tick size is a smaller computational efficiency. So your choice depends on the situation, and usually a ksmps of 32 represents a good tradeoff.

Of course the precision of writing samples (at a-rate) is in no way affected by the size of the internal k-ticks. Samples are indeed written "in between" control cycles, because they are vectors. So it can be necessary to use a-time variables instead of k-time variables in certain situations. In the following example, the ksmps value is rather high (128). If you use a k-rate variable for a fast moving envelope, you will hear a certain roughness (instrument 1) sometime referred to as 'zipper' noise. If you use an a-rate variable instead, you will have a much cleaner sound (instr 2).

EXAMPLE 03A09.csd

```
<CsoundSynthesizer>
<CsOptions>
-o dac
</CsOptions>
<CsInstruments>
;Example by Joachim Heintz
sr = 44100
;--- increase or decrease to hear the difference more or less evident
ksmps = 128
nchnls = 2
0dbfs = 1
```

```
instr 1 ;envelope at k-time
aSine     oscils    .5, 800, 0
kEnv      transeg   0, .1, 5, 1, .1, -5, 0
aOut      =         aSine * kEnv
          outs      aOut, aOut
endin

instr 2 ;envelope at a-time
aSine     oscils    .5, 800, 0
aEnv      transeg   0, .1, 5, 1, .1, -5, 0
aOut      =         aSine * aEnv
          outs      aOut, aOut
endin

</CsInstruments>
<CsScore>
r 5 ;repeat the following line 5 times
i 1 0 1
s ;end of section
r 5
i 2 0 1
e
</CsScore>
</CsoundSynthesizer>
```

14. LOCAL AND GLOBAL VARIABLES

VARIABLE TYPES

In Csound, there are several types of variables. It is important to understand the differences of these types. There are

- **initialization** variables, which are updated at each initialization pass, i.e. at the beginning of each note or score event. They start with the character **i**. To this group count also the score parameter fields, which always starts with a **p**, followed by any number: *p1* refers to the first parameter field in the score, *p2* to the second one, and so on.
- **control** variables, which are updated at each control cycle (performance pass). They start with the character **k**.
- **audio** variables, which are also updated at each control cycle, but instead of a single number (like control variables) they consist of a vector (a collection of numbers), having in this way one number for each sample. They start with the character **a**.
- **string** variables, which are updated either at i-time or at k-time (depending on the opcode which produces a string). They start with the character **S**.

Except these four standard types, there are two other variable types which are used for spectral processing:

- **f**-variables are used for the streaming phase vocoder opcodes (all starting with the characters **pvs**), which are very important for doing realtime FFT (Fast Fourier Transformation) in Csound. They are updated at k-time, but their values depend also on the FFT parameters like frame size and overlap.
- **w**-variables are used in some older spectral processing opcodes.

The following example exemplifies all the variable types (except the w-type):

EXAMPLE 03B01.csd

```
<CsoundSynthesizer>
<CsOptions>
-o dac
</CsOptions>
<CsInstruments>
;Example by Joachim Heintz
sr = 44100
ksmps = 32
0dbfs = 1
nchnls = 2

          seed      0; random seed each time different

  instr 1; i-time variables
iVar1     =         p2; second parameter in the score
iVar2     random    0, 10; random value between 0 and 10
iVar      =         iVar1 + iVar2; do any math at i-rate
          print     iVar1, iVar2, iVar
  endin

  instr 2; k-time variables
kVar1     line       0, p3, 10; moves from 0 to 10 in p3
kVar2     random     0, 10; new random value each control-cycle
kVar      =          kVar1 + kVar2; do any math at k-rate
; --- print each 0.1 seconds
printks   "kVar1 = %.3f, kVar2 = %.3f, kVar = %.3f%n", 0.1, kVar1, kVar2, kVar
  endin

  instr 3; a-variables
aVar1     oscils     .2, 400, 0; first audio signal: sine
aVar2     rand       1; second audio signal: noise
aVar3     butbp      aVar2, 1200, 12; third audio signal: noise filtered
aVar      =          aVar1 + aVar3; audio variables can also be added
          outs       aVar, aVar; write to sound card
  endin

  instr 4; S-variables
iMyVar    random     0, 10; one random value per note
kMyVar    random     0, 10; one random value per each control-cycle
 ;S-variable updated just at init-time
```

```
SMyVar1   sprintf   "This string is updated just at init-time:
                     kMyVar = %d\n", iMyVar
          printf_i  "%s", 1, SMyVar1
 ;S-variable updates at each control-cycle
          printks   "This string is updated at k-time:
                     kMyVar = %.3f\n", .1, kMyVar
  endin

  instr 5; f-variables
aSig      rand       .2; audio signal (noise)
; f-signal by FFT-analyzing the audio-signal
fSig1     pvsanal    aSig, 1024, 256, 1024, 1
; second f-signal (spectral bandpass filter)
fSig2     pvsbandp   fSig1, 350, 400, 400, 450
aOut      pvsynth    fSig2; change back to audio signal
          outs       aOut*20, aOut*20
  endin

</CsInstruments>
<CsScore>
; p1    p2    p3
i 1     0     0.1
i 1     0.1   0.1
i 2     1     1
i 3     2     1
i 4     3     1
i 5     4     1
</CsScore>
</CsoundSynthesizer>
```

You can think of variables as named connectors between opcodes. You can connect the output from an opcode to the input of another. The type of connector (audio, control, etc.) can be known from the first letter of its name.

For a more detailed discussion, see the article *An overview Of Csound Variable Types* by Andrés Cabrera in the *Csound Journal*, and the page about *Types, Constants and Variables* in the *Canonical Csound Manual*.

LOCAL SCOPE

The **scope** of these variables is usually the **instrument** in which they are defined. They are **local** variables. In the following example, the variables in instrument 1 and instrument 2 have the same names, but different values.

EXAMPLE 03B02.csd

```
<CsoundSynthesizer>
<CsOptions>
-odac
</CsOptions>
<CsInstruments>
;Example by Joachim Heintz
sr = 44100
ksmps = 4410; very high because of printing
nchnls = 2
0dbfs = 1

  instr 1
;i-variable
iMyVar    init      0
iMyVar    =         iMyVar + 1
          print     iMyVar
;k-variable
kMyVar    init      0
kMyVar    =         kMyVar + 1
          printk    0, kMyVar
;a-variable
aMyVar    oscils    .2, 400, 0
          outs      aMyVar, aMyVar
;S-variable updated just at init-time
SMyVar1   sprintf   "This string is updated just at init-time:
                     kMyVar = %d\n", i(kMyVar)
          printf    "%s", kMyVar, SMyVar1
;S-variable updated at each control-cycle
SMyVar2   sprintfk  "This string is updated at k-time:
                     kMyVar = %d\n", kMyVar
          printf    "%s", kMyVar, SMyVar2
  endin

  instr 2
;i-variable
```

```
iMyVar    init      100
iMyVar    =         iMyVar + 1
          print     iMyVar
;k-variable
kMyVar    init      100
kMyVar    =         kMyVar + 1
          printk    0, kMyVar
;a-variable
aMyVar    oscils    .3, 600, 0
          outs      aMyVar, aMyVar
;S-variable updated just at init-time
SMyVar1   sprintf   "This string is updated just at init-time:
                     kMyVar = %d\n", i(kMyVar)
          printf    "%s", kMyVar, SMyVar1
;S-variable updated at each control-cycle
SMyVar2   sprintfk  "This string is updated at k-time:
                     kMyVar = %d\n", kMyVar
          printf    "%s", kMyVar, SMyVar2
  endin

</CsInstruments>
<CsScore>
i 1 0 .3
i 2 1 .3
</CsScore>
</CsoundSynthesizer>
```

This is the output (first the output at init-time by the print opcode, then at each k-cycle the output of printk and the two printf opcodes):

```
new alloc for instr 1:
instr 1:  iMyVar = 1.000
 i   1 time     0.10000:     1.00000
This string is updated just at init-time: kMyVar = 0
This string is updated at k-time: kMyVar = 1
 i   1 time     0.20000:     2.00000
This string is updated just at init-time: kMyVar = 0
This string is updated at k-time: kMyVar = 2
 i   1 time     0.30000:     3.00000
This string is updated just at init-time: kMyVar = 0
This string is updated at k-time: kMyVar = 3
 B  0.000 ..  1.000 T  1.000 TT  1.000 M:  0.20000  0.20000
new alloc for instr 2:
instr 2:  iMyVar = 101.000
 i   2 time     1.10000:   101.00000
This string is updated just at init-time: kMyVar = 100
This string is updated at k-time: kMyVar = 101
 i   2 time     1.20000:   102.00000
This string is updated just at init-time: kMyVar = 100
This string is updated at k-time: kMyVar = 102
 i   2 time     1.30000:   103.00000
This string is updated just at init-time: kMyVar = 100
This string is updated at k-time: kMyVar = 103
B  1.000 ..  1.300 T  1.300 TT  1.300 M:  0.29998  0.29998
```

GLOBAL SCOPE

If you need variables which are recognized beyond the scope of an instrument, you must define them as **global**. This is done by prefixing the character **g** before the types i, k, a or S. See the following example:

EXAMPLE 03B03.csd

```
<CsoundSynthesizer>
<CsInstruments>
;Example by Joachim Heintz
sr = 44100
ksmps = 4410; very high because of printing
nchnls = 2
0dbfs = 1
```

```
 ;global scalar variables can now be inititalized in the header
giMyVar   init      0
gkMyVar   init      0

  instr 1
 ;global i-variable
giMyVar   =         giMyVar + 1
          print     giMyVar
 ;global k-variable
gkMyVar   =         gkMyVar + 1
          printk    0, gkMyVar
 ;global S-variable updated just at init-time
gSMyVar1  sprintf   "This string is updated just at init-time:
                     gkMyVar = %d\n", i(gkMyVar)
          printf    "%s", gkMyVar, gSMyVar1
 ;global S-variable updated at each control-cycle
gSMyVar2  sprintfk  "This string is updated at k-time:
                     gkMyVar = %d\n", gkMyVar
          printf    "%s", gkMyVar, gSMyVar2
  endin

  instr 2
 ;global i-variable, gets value from instr 1
giMyVar   =         giMyVar + 1
          print     giMyVar
 ;global k-variable, gets value from instr 1
gkMyVar   =         gkMyVar + 1
          printk    0, gkMyVar
 ;global S-variable updated just at init-time, gets value from instr 1
          printf    "Instr 1 tells: '%s'\n", gkMyVar, gSMyVar1
 ;global S-variable updated at each control-cycle, gets value from instr 1
          printf    "Instr 1 tells: '%s'\n\n", gkMyVar, gSMyVar2
  endin

</CsInstruments>
<CsScore>
i 1 0 .3
i 2 0 .3
</CsScore>
</CsoundSynthesizer>
```

The output shows the global scope, as instrument 2 uses the values which have been changed by instrument 1 in the same control cycle:

```
new alloc for instr 1:
instr 1:  giMyVar = 1.000
new alloc for instr 2:
instr 2:  giMyVar = 2.000
 i   1 time     0.10000:     1.00000
This string is updated just at init-time: gkMyVar = 0
This string is updated at k-time: gkMyVar = 1
 i   2 time     0.10000:     2.00000
Instr 1 tells: 'This string is updated just at init-time: gkMyVar = 0'
Instr 1 tells: 'This string is updated at k-time: gkMyVar = 1'

 i   1 time     0.20000:     3.00000
This string is updated just at init-time: gkMyVar = 0
This string is updated at k-time: gkMyVar = 3
 i   2 time     0.20000:     4.00000
Instr 1 tells: 'This string is updated just at init-time: gkMyVar = 0'
Instr 1 tells: 'This string is updated at k-time: gkMyVar = 3'

 i   1 time     0.30000:     5.00000
This string is updated just at init-time: gkMyVar = 0
This string is updated at k-time: gkMyVar = 5
 i   2 time     0.30000:     6.00000
Instr 1 tells: 'This string is updated just at init-time: gkMyVar = 0'
Instr 1 tells: 'This string is updated at k-time: gkMyVar = 5'
```

HOW TO WORK WITH GLOBAL AUDIO VARIABLES

Some special considerations must be taken if you work with global audio variables. Actually, Csound behaves basically the same whether you work with a local or a global audio variable. But usually you work with global audio variables if you want to **add** several audio signals to a global signal, and that makes a difference.

The next few examples are going into a bit more detail. If you just want to see the result (= global audio usually must be cleared), you can skip the next examples and just go to the last one of this section.

It should be understood first that a global audio variable is treated the same by Csound if it is applied like a local audio signal:

EXAMPLE 03B04.csd

```
<CsoundSynthesizer>
<CsOptions>
-odac
</CsOptions>
<CsInstruments>
;Example by Joachim Heintz
sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

  instr 1; produces a 400 Hz sine
gaSig     oscils    .1, 400, 0
  endin

  instr 2; outputs gaSig
          outs      gaSig, gaSig
  endin

</CsInstruments>
<CsScore>
i 1 0 3
i 2 0 3
</CsScore>
</CsoundSynthesizer>
```

Of course, there is absolutely no need to use a global variable in this case. If you do it, you risk that your audio will be overwritten by an instrument with a higher number that uses the same variable name. In the following example, you will just hear a 600 Hz sine tone, because the 400 Hz sine of instrument 1 is overwritten by the 600 Hz sine of instrument 2:

EXAMPLE 03B05.csd

```
<CsoundSynthesizer>
<CsOptions>
-o dac
</CsOptions>
<CsInstruments>
;Example by Joachim Heintz
sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

  instr 1; produces a 400 Hz sine
gaSig     oscils    .1, 400, 0
  endin

  instr 2; overwrites gaSig with 600 Hz sine
gaSig     oscils    .1, 600, 0
  endin

  instr 3; outputs gaSig
          outs      gaSig, gaSig
  endin

</CsInstruments>
<CsScore>
i 1 0 3
i 2 0 3
i 3 0 3
</CsScore>
</CsoundSynthesizer>
```

In general, you will use a global audio variable like a bus to which several local audio signal can be **added**. It's this addition of a global audio signal to its previous state which can cause some trouble. Let's first see a simple example of a control signal to understand what is happening:

EXAMPLE 03B06.csd

```
<CsoundSynthesizer>
<CsInstruments>
;Example by Joachim Heintz
sr = 44100
ksmps = 4410; very high because of printing
nchnls = 2
0dbfs = 1

  instr 1
kSum      init      0; sum is zero at init pass
kAdd      =         1; control signal to add
kSum      =         kSum + kAdd; new sum in each k-cycle
          printk    0, kSum; print the sum
  endin

</CsInstruments>
<CsScore>
i 1 0 1
</CsScore>
</CsoundSynthesizer>
```

In this case, the "sum bus" kSum increases at each control cycle by 1, because it adds the kAdd signal (which is always 1) in each k-pass to its previous state. It is no different if this is done by a local k-signal, like here, or by a global k-signal, like in the next example:

EXAMPLE 03B07.csd

```
<CsoundSynthesizer>
<CsInstruments>
;Example by Joachim Heintz
sr = 44100
ksmps = 4410; very high because of printing
nchnls = 2
0dbfs = 1

gkSum     init      0; sum is zero at init

  instr 1
gkAdd     =         1; control signal to add
  endin

  instr 2
gkSum     =         gkSum + gkAdd; new sum in each k-cycle
          printk    0, gkSum; print the sum
  endin

</CsInstruments>
<CsScore>
i 1 0 1
i 2 0 1
</CsScore>
</CsoundSynthesizer>
```

What is happening now when we work with audio signals instead of control signals in this way, repeatedly adding a signal to its previous state? Audio signals in Csound are a collection of numbers (a vector). The size of this vector is given by the ksmps constant. If your sample rate is 44100, and ksmps=100, you will calculate 441 times in one second a vector which consists of 100 numbers, indicating the amplitude of each sample.

So, if you add an audio signal to its previous state, different things can happen, depending on what is the present state of the vector and what was its previous state. If the previous state (with ksmps=9) has been [0 0.1 0.2 0.1 0 -0.1 -0.2 -0.1 0], and the present state is the same, you will get a signal which is twice as strong: [0 0.2 0.4 0.2 0 -0.2 -0.4 -0.2 0]. But if the present state is [0 -0.1 -0.2 -0.1 0 0.1 0.2 0.1 0], you wil just get zero's if you add it. This is shown in the next example with a local audio variable, and then in the following example with a global audio variable.

EXAMPLE 03B08.csd

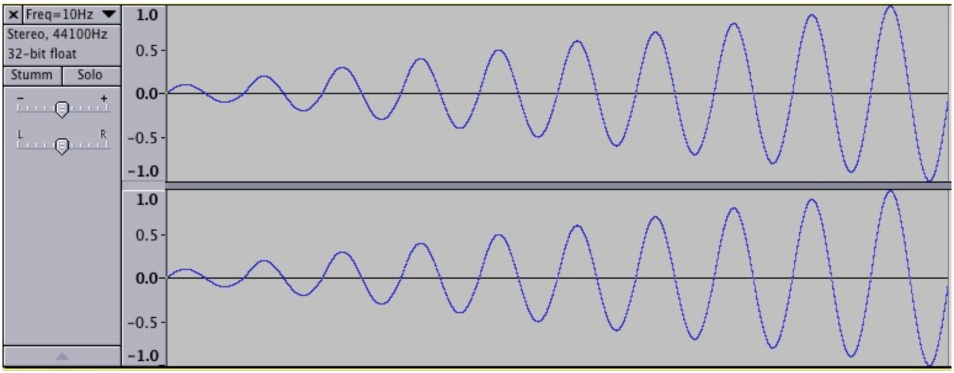
```
<CsoundSynthesizer>
<CsOptions>
-o dac
</CsOptions>
<CsInstruments>
;Example by Joachim Heintz
sr = 44100
ksmps = 4410; very high because of printing
```

```
            ;(change to 441 to see the difference)
nchnls = 2
0dbfs = 1

  instr 1
 ;initialize a general audio variable
aSum      init      0
 ;produce a sine signal (change frequency to 401 to see the difference)
aAdd      oscils    .1, 400, 0
 ;add it to the general audio (= the previous vector)
aSum      =         aSum + aAdd
kmax      max_k     aSum, 1, 1; calculate maximum
          printk    0, kmax; print it out
          outs      aSum, aSum
  endin

</CsInstruments>
<CsScore>
i 1 0 1
</CsScore>
</CsoundSynthesizer>
```

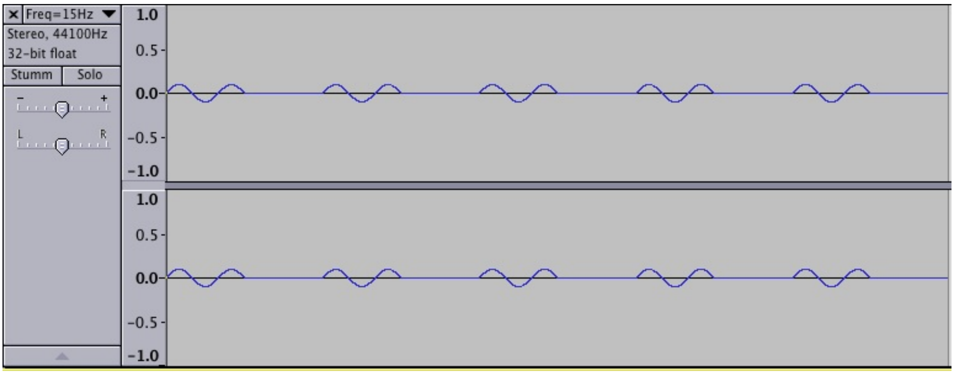
EXAMPLE 03B09.csd

```
<CsoundSynthesizer>
<CsOptions>
-o dac
</CsOptions>
<CsInstruments>
;Example by Joachim Heintz
sr = 44100
ksmps = 4410; very high because of printing
            ;(change to 441 to see the difference)
nchnls = 2
0dbfs = 1

 ;initialize a general audio variable
gaSum     init      0

  instr 1
 ;produce a sine signal (change frequency to 401 to see the difference)
aAdd      oscils    .1, 400, 0
 ;add it to the general audio (= the previous vector)
gaSum     =         gaSum + aAdd
  endin

  instr 2
kmax      max_k     gaSum, 1, 1; calculate maximum
          printk    0, kmax; print it out
          outs      gaSum, gaSum
  endin

</CsInstruments>
<CsScore>
i 1 0 1
i 2 0 1
</CsScore>
</CsoundSynthesizer>
```

In both cases, you get a signal which increases each 1/10 second, because you have 10 control cycles per second (ksmps=4410), and the frequency of 400 Hz can evenly be divided by this. If you change the ksmps value to 441, you will get a signal which increases much faster and is out of range after 1/10 second. If you change the frequency to 401 Hz, you will get a signal which increases first, and then decreases, because each audio vector has 40.1 cycles of the sine wave. So the phases are shifting; first getting stronger and then weaker. If you change the frequency to 10 Hz, and then to 15 Hz (at ksmps=44100), you cannot hear anything, but if you render to file, you can see the whole process of either enforcing or erasing quite clear:



Self-reinforcing global audio signal on account of its state in one control cycle being the same as in the previous one



Partly self-erasing global audio signal because of phase inversions in two subsequent control cycles

So the result of all is: If you work with global audio variables in a way that you add several local audio signals to a global audio variable (which works like a bus), you must **clear** this global bus at each control cycle. As in Csound all the instruments are calculated in ascending order, it should be done either at the beginning of the **first**, or at the end of the **last** instrument. Perhaps it is the best idea to declare all global audio variables in the orchestra header first, and then clear them in an "always on" instrument with the highest number of all the instruments used. This is an example of a typical situation:

EXAMPLE 03B10.csd

```
<CsoundSynthesizer>
<CsOptions>
-o dac
</CsOptions>
<CsInstruments>
;Example by Joachim Heintz
sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

 ;initialize the global audio variables
gaBusL    init      0
gaBusR    init      0
 ;make the seed for random values each time different
          seed      0

  instr 1; produces short signals
 loop:
iDur      random    .3, 1.5
          timout    0, iDur, makenote
          reinit    loop
 makenote:
```

```
iFreq     random    300, 1000
iVol      random    -12, -3; dB
iPan      random    0, 1; random panning for each signal
aSin      oscil3    ampdb(iVol), iFreq, 1
aEnv      transeg   1, iDur, -10, 0; env in a-rate is cleaner
aAdd      =         aSin * aEnv
aL, aR    pan2      aAdd, iPan
gaBusL    =         gaBusL + aL; add to the global audio signals
gaBusR    =         gaBusR + aR
  endin

  instr 2; produces short filtered noise signals (4 partials)
 loop:
iDur      random    .1, .7
          timout    0, iDur, makenote
          reinit    loop
 makenote:
iFreq     random    100, 500
iVol      random    -24, -12; dB
iPan      random    0, 1
aNois     rand      ampdb(iVol)
aFilt     reson     aNois, iFreq, iFreq/10
aRes      balance   aFilt, aNois
aEnv      transeg   1, iDur, -10, 0
aAdd      =         aRes * aEnv
aL, aR    pan2      aAdd, iPan
gaBusL    =         gaBusL + aL; add to the global audio signals
gaBusR    =         gaBusR + aR
  endin

  instr 3; reverb of gaBus and output
aL, aR    freeverb  gaBusL, gaBusR, .8, .5
          outs      aL, aR
  endin

  instr 100; clear global audios at the end
          clear     gaBusL, gaBusR
  endin

</CsInstruments>
<CsScore>
f 1 0 1024 10 1 .5 .3 .1
i 1 0 20
i 2 0 20
i 3 0 20
i 100 0 20
</CsScore>
</CsoundSynthesizer>
```

THE CHN OPCODES FOR GLOBAL VARIABLES

Instead of using the traditional g-variables for any values or signals which are to transfer between several instruments, it is also possible to use the chn opcodes. An i-, k-, a- or S-value or signal can be set by chnset and received by chnget. One advantage is to have strings as names, so that you can choose intuitive names.

For audio variables, instead of performing an addition, you can use the chnmix opcode. For clearing an audio variable, the chnclear opcode can be used.

EXAMPLE 03B11.csd

```
<CsoundSynthesizer>
<CsOptions>
-o dac
</CsOptions>
<CsInstruments>
;Example by Joachim Heintz
sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

  instr 1; send i-values
          chnset    1, "sio"
          chnset    -1, "non"
  endin

  instr 2; send k-values
kfreq     randomi   100, 300, 1
          chnset    kfreq, "cntrfreq"
kbw       =         kfreq/10
          chnset    kbw, "bandw"
  endin

  instr 3; send a-values
anois     rand      .1
          chnset    anois, "noise"
 loop:
idur      random    .3, 1.5
          timout    0, idur, do
          reinit    loop
 do:
ifreq     random    400, 1200
iamp      random    .1, .3
asig      oscils    iamp, ifreq, 0
aenv      transeg   1, idur, -10, 0
asine     =         asig * aenv
          chnset    asine, "sine"
  endin

  instr 11; receive some chn values and send again
ival1     chnget    "sio"
ival2     chnget    "non"
          print     ival1, ival2
kcntfreq  chnget    "cntrfreq"
kbandw    chnget    "bandw"
anoise    chnget    "noise"
afilt     reson     anoise, kcntfreq, kbandw
afilt     balance   afilt, anoise
          chnset    afilt, "filtered"
  endin

  instr 12; mix the two audio signals
amix1     chnget     "sine"
amix2     chnget     "filtered"
          chnmix     amix1, "mix"
          chnmix     amix2, "mix"
  endin

  instr 20; receive and reverb
amix      chnget     "mix"
aL, aR    freeverb   amix, amix, .8, .5
          outs       aL, aR
  endin

  instr 100; clear
          chnclear   "mix"
  endin

</CsInstruments>
<CsScore>
i 1 0 20
i 2 0 20
i 3 0 20
i 11 0 20
i 12 0 20
i 20 0 20
i 100 0 20
</CsScore>
</CsoundSynthesizer>
```

15. CONTROL STRUCTURES

In a way, control structures are the core of a programming language. The fundamental element in each language is the conditional **if** branch. Actually all other control structures like for-, until- or while-loops can be traced back to if-statements.

So, Csound provides mainly the if-statement; either in the usual *if-then-else* form, or in the older way of an *if-goto* statement. These ones will be covered first. Though all necessary loops can be built just by if-statements, Csound's *loop* facility offers a more comfortable way of performing loops. They will be introduced in the Loop section of this chapter. At least, time loops are shown, which are particulary important in audio programming languages.

IF I-TIME THEN NOT K-TIME!

The fundamental difference in Csound between i-time and k-time which has been explained in a previous chapter, must be regarded very carefully when you work with control structures. If you make a conditional branch at **i-time**, the condition will be tested **just once for each note**, at the initialization pass. If you make a conditional branch at **k-time**, the condition will be tested **again and again in each control-cycle**.

For instance, if you test a soundfile whether it is mono or stereo, this is done at init-time. If you test an amplitude value to be below a certain threshold, it is done at performance time (k-time). If you get user-input by a scroll number, this is also a k-value, so you need a k-condition.

Thus, all if and loop opcodes have an "i" and a "k" descendant. In the next few sections, a general introduction into the different control tools is given, followed by examples both at i-time and at k-time for each tool.

IF - THEN - [ELSEIF - THEN -] ELSE

The use of the if-then-else statement is very similar to other programming languages. Note that in Csound, "then" must be written in the same line as "if" and the expression to be tested, and that you must close the if-block with an "endif" statement on a new line:

```
if <condition> then
...
else
...
endif
```

It is also possible to have no "else" statement:

```
if <condition> then
...
endif
```

Or you can have one or more "elseif-then" statements in between:

```
if <condition1> then
...
elseif <condition2> then
...
else
...
endif
```

If statements can also be nested. Each level must be closed with an "endif". This is an example with three levels:

```
if <condition1> then; first condition opened
 if <condition2> then; second condition openend
  if <condition3> then; third condition openend
  ...
  else
  ...
  endif; third condition closed
 elseif <condition2a> then
 ...
 endif; second condition closed
```

```
else
...
endif; first condition closed
```

i-Rate Examples

A typical problem in Csound: You have either mono or stereo files, and want to read both with a stereo output. For the real stereo ones that means: use soundin (diskin / diskin2) with two output arguments. For the mono ones it means: use soundin / diskin / diskin2 with one output argument, and throw it to both output channels:

EXAMPLE 03C01.csd

```
<CsoundSynthesizer>
<CsOptions>
-o dac
</CsOptions>
<CsInstruments>
;Example by Joachim Heintz
sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

  instr 1
Sfile     =          "/my/file.wav" ;your soundfile path here
ifilchnls filenchnls Sfile
 if ifilchnls == 1 then ;mono
aL        soundin    Sfile
aR        =          aL
 else ;stereo
aL, aR    soundin    Sfile
 endif
          outs       aL, aR
  endin

</CsInstruments>
<CsScore>
i 1 0 5
</CsScore>
</CsoundSynthesizer>
```

If you use QuteCsound, you can browse in the widget panel for the soundfile. See the corresponding example in the QuteCsound Example menu.

k-Rate Examples

The following example establishes a moving gate between 0 and 1. If the gate is above 0.5, the gate opens and you hear a tone. If the gate is equal or below 0.5, the gate closes, and you hear nothing.

EXAMPLE 03C02.csd

```
<CsoundSynthesizer>
<CsOptions>
-odac
</CsOptions>
<CsInstruments>
;Example by Joachim Heintz
sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

          seed      0; random values each time different
giTone    ftgen     0, 0, 2^10, 10, 1, .5, .3, .1

  instr 1
kGate     randomi   0, 1, 3; moves between 0 and 1 (3 new values per second)
kFreq     randomi   300, 800, 1; moves between 300 and 800 hz (1 new value per
sec)
kdB       randomi   -12, 0, 5; moves between -12 and 0 dB
                            ;(5 new values per sec)
aSig      oscil3    1, kFreq, giTone
kVol      init      0
 if kGate > 0.5 then; if kGate is larger than 0.5
kVol      =         ampdb(kdB); open gate
 else
kVol      =         0; otherwise close gate
 endif
```

```
kVol      port      kVol, .02; smooth volume curve to avoid clicks
aOut      =         aSig * kVol
          outs      aOut, aOut
  endin

</CsInstruments>
<CsScore>
i 1 0 30
</CsScore>
</CsoundSynthesizer>
```

Short Form: (a v b ? x : y)

If you need an if-statement to give a value to an (i- or k-) variable, you can also use a traditional short form in parentheses: (a v b ? x : y). It asks whether the condition a or b is true. If a, the value is set to x; if b, to y. For instance, the last example could be written in this way:

EXAMPLE 03C03.csd

```
<CsoundSynthesizer>
<CsOptions>
-odac
</CsOptions>
<CsInstruments>
;Example by Joachim Heintz
sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

          seed      0
giTone    ftgen     0, 0, 2^10, 10, 1, .5, .3, .1

  instr 1
kGate     randomi   0, 1, 3; moves between 0 and 1 (3 new values per second)
kFreq     randomi   300, 800, 1; moves between 300 and 800 hz
                               ;(1 new value per sec)
kdB       randomi   -12, 0, 5; moves between -12 and 0 dB
                             ;(5 new values per sec)
aSig      oscil3    1, kFreq, giTone
kVol      init      0
kVol      =         (kGate > 0.5 ? ampdb(kdB) : 0); short form of condition
kVol      port      kVol, .02; smooth volume curve to avoid clicks
aOut      =         aSig * kVol
          outs      aOut, aOut
  endin

</CsInstruments>
<CsScore>
i 1 0 20
</CsScore>
</CsoundSynthesizer>
```

IF - GOTO

An older way of performing a conditional branch - but still useful in certain cases - is an "if" statement which is not followed by a "then", but by a label name. The "else" construction follows (or doesn't follow) in the next line. Like the if-then-else statement, the if-goto works either at i-time or at k-time. You should declare the type by either using **i**goto or **k**goto. Usually you need an additional igoto/kgoto statement for omitting the "else" block if the first condition is true. This is the general syntax:

i-time

```
if <condition> igoto this; same as if-then
 igoto that; same as else
this: ;the label "this" ...
...
igoto continue ;skip the "that" block
that: ; ... and the label "that" must be found
...
continue: ;go on after the conditional branch
...
```

k-time

```
if <condition> kgoto this; same as if-then
 kgoto that; same as else
this: ;the label "this" ...
```

```
...
kgoto continue ;skip the "that" block
that: ; ... and the label "that" must be found
...
continue: ;go on after the conditional branch
...
```

i-Rate Examples

This is the same example as above in the if-then-else syntax for a branch depending on a mono or stereo file. If you just want to know whether a file is mono or stereo, you can use the "pure" if-igoto statement:

EXAMPLE 03C04.csd

```
<CsoundSynthesizer>
<CsInstruments>
;Example by Joachim Heintz
sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

  instr 1
Sfile     = "/Joachim/Materialien/SamplesKlangbearbeitung/Kontrabass.aif"
ifilchnls filenchnls Sfile
if ifilchnls == 1 igoto mono; condition if true
 igoto stereo; else condition
mono:
          prints     "The file is mono!%n"
          igoto      continue
stereo:
          prints     "The file is stereo!%n"
continue:
  endin

</CsInstruments>
<CsScore>
i 1 0 0
</CsScore>
</CsoundSynthesizer>
```

But if you want to play the file, you must also use a k-rate if-kgoto, because you have not just an action at i-time (initializing the soundin opcode) but also at k-time (producing an audio signal). So the code in this case is much more cumbersome than with the if-then-else facility shown previously.

EXAMPLE 03C05.csd

```
<CsoundSynthesizer>
<CsOptions>
-odac
</CsOptions>
<CsInstruments>
;Example by Joachim Heintz
sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

  instr 1
Sfile     =          "my/file.wav"
ifilchnls filenchnls Sfile
 if ifilchnls == 1 kgoto mono
  kgoto stereo
 if ifilchnls == 1 igoto mono; condition if true
  igoto stereo; else condition
mono:
aL        soundin    Sfile
aR        =          aL
          igoto      continue
          kgoto      continue
stereo:
aL, aR    soundin    Sfile
continue:
          outs       aL, aR
  endin

</CsInstruments>
<CsScore>
i 1 0 5
```

```
</CsScore>
</CsoundSynthesizer>
```

k-Rate Examples

This is the same example as above in the if-then-else syntax for a moving gate between 0 and 1:

EXAMPLE 03C06.csd

```
<CsoundSynthesizer>
<CsOptions>
-odac
</CsOptions>
<CsInstruments>
;Example by Joachim Heintz
sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

          seed      0
giTone    ftgen     0, 0, 2^10, 10, 1, .5, .3, .1

  instr 1
kGate     randomi   0, 1, 3; moves between 0 and 1 (3 new values per second)
kFreq     randomi   300, 800, 1; moves between 300 and 800 hz
                              ;(1 new value per sec)
kdB       randomi   -12, 0, 5; moves between -12 and 0 dB
                             ;(5 new values per sec)
aSig      oscil3    1, kFreq, giTone
kVol      init      0
 if kGate > 0.5 kgoto open; if condition is true
  kgoto close; "else" condition
open:
kVol      =         ampdb(kdB)
kgoto continue
close:
kVol      =         0
continue:
kVol      port      kVol, .02; smooth volume curve to avoid clicks
aOut      =         aSig * kVol
          outs      aOut, aOut
  endin

</CsInstruments>
<CsScore>
i 1 0 30
</CsScore>
</CsoundSynthesizer>
```

LOOPS

Loops can be built either at i-time or at k-time just with the "if" facility. The following example shows an i-rate and a k-rate loop created using the if-i/kgoto facility:

EXAMPLE 03C07.csd

```
<CsoundSynthesizer>
<CsInstruments>
;Example by Joachim Heintz

  instr 1 ;i-time loop: counts from 1 until 10 has been reached
icount    =         1
count:
          print     icount
icount    =         icount + 1
 if icount < 11 igoto count
          prints    "i-END!%n"
  endin

  instr 2 ;k-rate loop: counts in the 100th k-cycle from 1 to 11
kcount    init      0
ktimek    timeinstk ;counts k-cycle from the start of this instrument
 if ktimek == 100 kgoto loop
  kgoto noloop
loop:
          printks   "k-cycle %d reached!%n", 0, ktimek
kcount    =         kcount + 1
          printk2   kcount
 if kcount < 11 kgoto loop
          printks   "k-END!%n", 0
```

```
noloop:
  endin

</CsInstruments>
<CsScore>
i 1 0 0
i 2 0 1
</CsScore>
</CsoundSynthesizer>
```

But Csound offers a slightly simpler syntax for this kind of i-rate or k-rate loops. There are four variants of the loop opcode. All four refer to a *label* as the starting point of the loop, an *index variable* as a counter, an *increment* or *decrement*, and finally a *reference value* (maximum or minimum) as comparision:

- loop_lt counts upwards and looks if the index variable is **lower than** the reference value;
- loop_le also counts upwards and looks if the index is **lower than or equal to** the reference value;
- loop_gt counts downwards and looks if the index is **greater than** the reference value;
- loop_ge also counts downwards and looks if the index is **greater than or equal to** the reference value.

As always, all four opcodes can be applied either at i-time or at k-time. Here are some examples, first for i-time loops, and then for k-time loops.

i-Rate Examples

The following .csd provides a simple example for all four loop opcodes:

EXAMPLE 03C08.csd

```
<CsoundSynthesizer>
<CsInstruments>
;Example by Joachim Heintz

  instr 1 ;loop_lt: counts from 1 upwards and checks if < 10
icount    =         1
loop:
          print     icount
          loop_lt   icount, 1, 10, loop
          prints    "Instr 1 terminated!%n"
  endin

  instr 2 ;loop_le: counts from 1 upwards and checks if <= 10
icount    =         1
loop:
          print     icount
          loop_le   icount, 1, 10, loop
          prints    "Instr 2 terminated!%n"
  endin

  instr 3 ;loop_gt: counts from 10 downwards and checks if > 0
icount    =         10
loop:
          print     icount
          loop_gt   icount, 1, 0, loop
          prints    "Instr 3 terminated!%n"
  endin

  instr 4 ;loop_ge: counts from 10 downwards and checks if >= 0
icount    =         10
loop:
          print     icount
          loop_ge   icount, 1, 0, loop
          prints    "Instr 4 terminated!%n"
  endin

</CsInstruments>
<CsScore>
i 1 0 0
i 2 0 0
i 3 0 0
i 4 0 0
</CsScore>
</CsoundSynthesizer>
```

The next example produces a random string of 10 characters and prints it out:

EXAMPLE 03C09.csd

```
<CsoundSynthesizer>
<CsInstruments>
;Example by Joachim Heintz

  instr 1
icount    =         0
Sname     =         ""; starts with an empty string
loop:
ichar     random    65, 90.999
Schar     sprintf   "%c", int(ichar); new character
Sname     strcat    Sname, Schar; append to Sname
          loop_lt   icount, 1, 10, loop; loop construction
          printf_i  "My name is '%s'!\n", 1, Sname; print result
  endin

</CsInstruments>
<CsScore>
; call instr 1 ten times
r 10
i 1 0 0
</CsScore>
</CsoundSynthesizer>
```

You can also use an i-rate loop to fill a function table (= buffer) with any kind of values. In the next example, a function table with 20 positions (indices) is filled with random integers between 0 and 10 by instrument 1. Nearly the same loop construction is used afterwards to read these values by instrument 2.

EXAMPLE 03C10.csd

```
<CsoundSynthesizer>
<CsInstruments>
;Example by Joachim Heintz

giTable   ftgen     0, 0, -20, -2, 0; empty function table with 20 points
          seed      0; each time different seed

  instr 1 ; writes in the table
icount    =         0
loop:
ival      random    0, 10.999 ;random value
; --- write in giTable at first, second, third ... position
          tableiw   int(ival), icount, giTable
          loop_lt   icount, 1, 20, loop; loop construction
  endin

  instr 2; reads from the table
icount    =         0
loop:
; --- read from giTable at first, second, third ... position
ival      tablei    icount, giTable
          print     ival; prints the content
          loop_lt   icount, 1, 20, loop; loop construction
  endin

</CsInstruments>
<CsScore>
i 1 0 0
i 2 0 0
</CsScore>
</CsoundSynthesizer>
```

k-Rate Examples

The next example performs a loop at k-time. Once per second, every value of an existing function table is changed by a random deviation of 10%. Though there are special opcodes for this task, it can also be done by a k-rate loop like the one shown here:

EXAMPLE 03C11.csd

```
<CsoundSynthesizer>
<CsOptions>
-odac
</CsOptions>
<CsInstruments>
;Example by Joachim Heintz
sr = 44100
ksmps = 441
nchnls = 2
0dbfs = 1
```

```
giSine    ftgen     0, 0, 256, 10, 1; sine wave
          seed      0; each time different seed

  instr 1
ktiminstk timeinstk ;time in control-cycles
kcount    init      1
 if ktiminstk == kcount * kr then; once per second table values manipulation:
kndx      =         0
loop:
krand     random    -.1, .1;random factor for deviations
kval      table     kndx, giSine; read old value
knewval   =         kval + (kval * krand); calculate new value
          tablew    knewval, kndx, giSine; write new value
          loop_lt   kndx, 1, 256, loop; loop construction
kcount    =         kcount + 1; increase counter
 endif
asig      poscil    .2, 400, giSine
          outs      asig, asig
  endin

</CsInstruments>
<CsScore>
i 1 0 10
</CsScore>
</CsoundSynthesizer>
```

TIME LOOPS

Until now, we have just discussed loops which are executed "as fast as possible", either at i-time or at k-time. But, in an audio programming language, time loops are of particular interest and importance. A time loop means, repeating any action after a certain amount of time. This amount of time can be equal to or different to the previous time loop. The action can be, for instance: playing a tone, or triggering an instrument, or calculating a new value for the movement of an envelope.

In Csound, the usual way of performing time loops, is the timout facility. The use of timout is a bit intricate, so some examples are given, starting from very simple to more complex ones.

Another way of performing time loops is by using a measurement of time or k-cycles. This method is also discussed and similar examples to those used for the timout opcode are given so that both methods can be compared.

timout Basics

The timout opcode refers to the fact that in the traditional way of working with Csound, each "note" (an "i" score event) has its own time. This is the duration of the note, given in the score by the duration parameter, abbreviated as "p3". A timout statement says: "I am now jumping out of this p3 duration and establishing my own time." This time will be repeated as long as the duration of the note allows it.

Let's see an example. This is a sine tone with a moving frequency, starting at 400 Hz and ending at 600 Hz. The duration of this movement is 3 seconds for the first note, and 5 seconds for the second note:

EXAMPLE 03C12.csd

```
<CsoundSynthesizer>
<CsOptions>
-odac
</CsOptions>
<CsInstruments>
;Example by Joachim Heintz
sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

giSine    ftgen     0, 0, 2^10, 10, 1

  instr 1
kFreq     expseg    400, p3, 600
aTone     poscil    .2, kFreq, giSine
          outs      aTone, aTone
  endin
```

```
</CsInstruments>
<CsScore>
i 1 0 3
i 1 4 5
</CsScore>
</CsoundSynthesizer>
```

Now we perform a time loop with timout which is 1 second long. So, for the first note, it will be repeated three times, and for the second note five times:

EXAMPLE 03C13.csd

```
<CsoundSynthesizer>
<CsOptions>
-odac
</CsOptions>
<CsInstruments>
;Example by Joachim Heintz
sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

giSine    ftgen     0, 0, 2^10, 10, 1

  instr 1
loop:
          timout    0, 1, play
          reinit    loop
play:
kFreq     expseg    400, 1, 600
aTone     poscil    .2, kFreq, giSine
          outs      aTone, aTone
  endin

</CsInstruments>
<CsScore>
i 1 0 3
i 1 4 5
</CsScore>
</CsoundSynthesizer>
```

This is the general syntax of timout:

```
first_label:
          timout    istart, idur, second_label
          reinit    first_label
second_label:
... <any action you want to have here>
```

The **first_label** is an arbitrary word (followed by a colon) for marking the beginning of the time loop section. The **istart** argument for timout tells Csound, when the **second_label** section is to be executed. Usually istart is zero, telling Csound: execute the second_label section immediately, without any delay. The **idur** argument for timout defines how many seconds the second_label section is to be executed before the time loop begins again. Note that the "reinit first_label" is necessary to start the second loop after idur seconds with a resetting of all the values. (See the explanations about reinitialization in the chapter Initilalization And Performance Pass.)

As usual when you work with the reinit opcode, you can use a rireturn statement to constrain the reinit-pass. In this way you can have both, the timeloop section and the non-timeloop section in the body of an instrument:

EXAMPLE 03C14.csd

```
<CsoundSynthesizer>
<CsOptions>
-odac
</CsOptions>
<CsInstruments>
;Example by Joachim Heintz
sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

giSine    ftgen     0, 0, 2^10, 10, 1

  instr 1
loop:
          timout    0, 1, play
          reinit    loop
play:
kFreq1    expseg    400, 1, 600
aTone1    oscil3    .2, kFreq1, giSine
          rireturn  ;end of the time loop
kFreq2    expseg    400, p3, 600
aTone2    poscil    .2, kFreq2, giSine

          outs      aTone1+aTone2, aTone1+aTone2
  endin

</CsInstruments>
<CsScore>
i 1 0 3
i 1 4 5
</CsScore>
</CsoundSynthesizer>
```

timout Applications

In a time loop, it is very important to change the duration of the loop. This can be done either by referring to the duration of this note (p3) ...

EXAMPLE 03C15.csd

```
<CsoundSynthesizer>
<CsOptions>
-odac
</CsOptions>
<CsInstruments>
;Example by Joachim Heintz
sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

giSine    ftgen     0, 0, 2^10, 10, 1

  instr 1
loop:
          timout    0, p3/5, play
          reinit    loop
play:
kFreq     expseg    400, p3/5, 600
aTone     poscil    .2, kFreq, giSine
          outs      aTone, aTone
  endin

</CsInstruments>
<CsScore>
i 1 0 3
i 1 4 5
</CsScore>
</CsoundSynthesizer>
```

... or by calculating new values for the loop duration on each reinit pass, for instance by random values:

EXAMPLE 03C16.csd

```
<CsoundSynthesizer>
<CsOptions>
-odac
</CsOptions>
<CsInstruments>
;Example by Joachim Heintz
sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

giSine    ftgen     0, 0, 2^10, 10, 1

  instr 1
loop:
idur      random    .5, 3 ;new value between 0.5 and 3 seconds each time
          timout    0, idur, play
          reinit    loop
play:
kFreq     expseg    400, idur, 600
aTone     poscil    .2, kFreq, giSine
          outs      aTone, aTone
  endin

</CsInstruments>
<CsScore>
i 1 0 20
</CsScore>
</CsoundSynthesizer>
```

The applications discussed so far have the disadvantage that all the signals inside the time loop must definitely be finished or interrupted, when the next loop begins. In this way it is not possible to have any overlapping of events. For achieving this, the time loop can be used just to **trigger an event**. This can be done with event_i or scoreline_i. In the following example, the time loop in instrument 1 triggers each half to two seconds an instance of instrument 2 for a duration of 1 to 5 seconds. So usually the previous instance of instrument 2 will still play when the new instance is triggered. In instrument 2, some random calculations are executed to make each note different, though having a descending pitch (glissando):

EXAMPLE 03C17.csd

```
<CsoundSynthesizer>
<CsOptions>
-odac
</CsOptions>
<CsInstruments>
;Example by Joachim Heintz
sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

giSine    ftgen     0, 0, 2^10, 10, 1

  instr 1
loop:
idurloop  random    .5, 2 ;duration of each loop
          timout    0, idurloop, play
          reinit    loop
play:
idurins   random    1, 5 ;duration of the triggered instrument
          event_i   "i", 2, 0, idurins ;triggers instrument 2
  endin

  instr 2
ifreq1    random    600, 1000 ;starting frequency
idiff     random    100, 300 ;difference to final frequency
ifreq2    =         ifreq1 - idiff ;final frequency
kFreq     expseg    ifreq1, p3, ifreq2 ;glissando
iMaxdb    random    -12, 0 ;peak randomly between -12 and 0 dB
kAmp      transeg   ampdb(iMaxdb), p3, -10, 0 ;envelope
aTone     poscil    kAmp, kFreq, giSine
          outs      aTone, aTone
  endin

</CsInstruments>
<CsScore>
i 1 0 30
</CsScore>
</CsoundSynthesizer>
```

The last application of a time loop with the *timout* opcode which is shown here, is a **randomly moving envelope**. If you want to create an envelope in Csound which moves between a lower and an upper limit, and has one new random value in a certain time span (for instance, once a second), the time loop with *timout* is one way to achieve it. A line movement must be performed in each time loop, from a given starting value to a new evaluated final value. Then, in the next loop, the previous final value must be set as the new starting value, and so on. This is a possible solution:

EXAMPLE 03C18.csd

```
<CsoundSynthesizer>
<CsOptions>
-odac
</CsOptions>
<CsInstruments>
;Example by Joachim Heintz
sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1
```

```
giSine    ftgen     0, 0, 2^10, 10, 1
          seed      0

  instr 1
iupper    =         0; upper and ...
ilower    =         -24; ... lower limit in dB
ival1     random    ilower, iupper; starting value
loop:
idurloop  random    .5, 2; duration of each loop
          timout    0, idurloop, play
          reinit    loop
play:
ival2     random    ilower, iupper; final value
kdb       linseg    ival1, idurloop, ival2
ival1     =         ival2; let ival2 be ival1 for next loop
          rireturn  ;end reinit section
aTone     poscil    ampdb(kdb), 400, giSine
          outs      aTone, aTone
  endin

</CsInstruments>
<CsScore>
i 1 0 30
</CsScore>
</CsoundSynthesizer>
```

Note that in this case the oscillator has been put after the time loop section (which is terminated by the *rireturn* statement. Otherwise the oscillator would start afresh with zero phase in each time loop, thus producing clicks.

Time Loops by using the *metro* Opcode

The metro opcode outputs a "1" at distinct times, otherwise it outputs a "0". The frequency of this "banging" (which is in some way similar to the metro objects in PD or Max) is given by the *kfreq* input argument. So the output of *metro* offers a simple and intuitive method for controlling time loops, if you use it to trigger a separate instrument which then carries out another job. Below is a simple example for calling a subinstrument twice a second:

EXAMPLE 03C19.csd

```
<CsoundSynthesizer>
<CsOptions>
-odac
</CsOptions>
<CsInstruments>
;Example by Joachim Heintz
sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

  instr 1; triggering instrument
kTrig     metro     2; outputs "1" twice a second
 if kTrig == 1 then
          event     "i", 2, 0, 1
 endif
  endin

  instr 2; triggered instrument
aSig      oscils    .2, 400, 0
aEnv      transeg   1, p3, -10, 0
          outs      aSig*aEnv, aSig*aEnv
  endin

</CsInstruments>
<CsScore>
i 1 0 10
</CsScore>
</CsoundSynthesizer>
```

The example which is given above (03C17.csd) as a flexible time loop by *timout*, can be done with the *metro* opcode in this way:

EXAMPLE 03C20.csd

```
<CsoundSynthesizer>
<CsOptions>
-odac
</CsOptions>
<CsInstruments>
```

```
;Example by Joachim Heintz
sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

giSine    ftgen     0, 0, 2^10, 10, 1
          seed      0

  instr 1
kfreq     init      1; give a start value for the trigger frequency
kTrig     metro     kfreq
 if kTrig == 1 then ;if trigger impulse:
kdur      random    1, 5; random duration for instr 2
          event     "i", 2, 0, kdur; call instr 2
kfreq     random    .5, 2; set new value for trigger frequency
 endif
  endin

  instr 2
ifreq1    random    600, 1000; starting frequency
idiff     random    100, 300; difference to final frequency
ifreq2    =         ifreq1 - idiff; final frequency
kFreq     expseg    ifreq1, p3, ifreq2; glissando
iMaxdb    random    -12, 0; peak randomly between -12 and 0 dB
kAmp      transeg   ampdb(iMaxdb), p3, -10, 0; envelope
aTone     poscil    kAmp, kFreq, giSine
          outs      aTone, aTone
  endin

</CsInstruments>
<CsScore>
i 1 0 30
</CsScore>
</CsoundSynthesizer>
```

Note the differences in working with the *metro* opcode compared to the *timout* feature:

- As *metro* works at k-time, you must use the k-variants of event or scoreline to call the subinstrument. With *timout* you must use the i-variants of *event* or *scoreline* (event_i and scoreline_i), because it uses reinitialization for performing the time loops.
- You must select the one k-cycle where the metro opcode sends a "1". This is done with an if-statement. The rest of the instrument is not affected. If you use *timout*, you usually must seperate the reinitialized from the not reinitialized section by a *rireturn* statement.

LINKS

Steven Yi: Control Flow (Part I = Csound Journal Spring 2006, Part 2 = Csound Journal Summer 2006)

16. FUNCTION TABLES

A function table is essentially the same as what other audio programming languages call a buffer, a table, a list or an array. It is a place where data can be stored in an ordered way. Each function table has a **size**: how much data (in Csound just numbers) can be stored in it. Each value in the table can be accessed by an **index**, counting from 0 to size-1. For instance, if you have a function table with a size of 10, and the numbers [1.1 2.2 3.3 5.5 8.8 13.13 21.21 34.34 55.55 89.89] in it, this is the relation of value and index:

VALUE	1.1	2.2	3.3	5.5	8.8	13.13	21.21	34.34	55.55	89.89
INDEX	0	1	2	3	4	5	6	7	8	9

So, if you want to retrieve the value 13.13, you must point to the value stored under index 5.

The use of function tables is manifold. A function table can contain pitch values to which you may refer using the input of a MIDI keyboard. A function table can contain a model of a waveform which is read periodically by an oscillator. You can record live audio input in a function table, and then play it back. There are many more applications, all using the fast access (because a function table is part of the RAM) and flexible use of function tables.

HOW TO GENERATE A FUNCTION TABLE

Each function table must be created **before** it can be used. Even if you want to write values later, you must first create an empty table, because you must initially reserve some space in memory for it.

Each creation of a function table in Csound is performed by one of the so-called **GEN Routines**. Each GEN Routine generates a function table in a particular way: GEN01 transfers audio samples from a soundfile into a table, with GEN02 we can write values in "by hand" one by one, GEN10 calculates a waveform using information determining a sum of sinusoids, GEN20 generates window functions typically used for granular synthesis, and so on. There is a good overview in the Csound Manual of all existing GEN Routines. Here we will explain the general use and give simple examples for some frequent cases.

GEN02 And General Parameters For GEN Routines

Let's start with our example above and write the 10 numbers into a function table of the same size. For this, use of a GEN02 function table is required. A short description of GEN02 from the manual reads as follows:

```
f # time size 2 v1 v2 v3 ...
```

This is the traditional way of creating a function table by an "**f statement**" or an "**f score event**" (in comparision for instance to "i score events" which call instrument instances). The input parameters after the "f" are the following:

- **#**: a number (as positive integer) for this function table;
- **time**: at which time to be the function table available (usually 0 = from the beginning);
- **size**: the size of the function table. This is a bit tricky, because in the early days of Csound just power-of-two sizes for function tables were possible (2, 4, 8, 16, ...). Nowadays nearly every GEN Routine accepts other sizes, but these **non-power-of-two sizes must be declared as a negative number**!
- **2**: the number of the GEN Routine which is used to generate the table. And here is another important point which must be regarded. **By default, Csound normalizes the table values.** This means that the maximum is scaled to +1 if positive, and to -1 if negative. To **prevent** Csound from normalizing, a **negative** number must be given as GEN number (here -2 instead of 2).
- **v1 v2 v3 ...**: the values which are written into the function table.

So this is the way to put the values [1.1 2.2 3.3 5.5 8.8 13.13 21.21 34.34 55.55 89.89] in a function table with the number 1:

EXAMPLE 03D01.csd

```
<CsoundSynthesizer>
<CsInstruments>
;Example by Joachim Heintz
  instr 1 ;prints the values of table 1 or 2
          prints    "%nFunction Table %d:%n", p4
indx      init      0
loop:
ival      table     indx, p4
          prints    "Index %d = %f%n", indx, ival
          loop_lt   indx, 1, 10, loop
  endin
</CsInstruments>
<CsScore>
f 1 0 -10 -2 1.1 2.2 3.3 5.5 8.8 13.13 21.21 34.34 55.55 89.89; not normalized
f 2 0 -10 2 1.1 2.2 3.3 5.5 8.8 13.13 21.21 34.34 55.55 89.89; normalized
i 1 0 0 1; prints function table 1
i 1 0 0 2; prints function table 2
</CsScore>
</CsoundSynthesizer>
```

Instrument 1 just serves to print the values of the table (the *tablei* opcode will be explained later). See the difference whether the table is normalized (positive GEN number) or not normalized (negative GEN number).

Using the ftgen opcode is a more modern way of creating a function table, which is in some ways preferable to the old way of writing an f-statement in the score. The syntax is explained below:

```
giVar     ftgen     ifn, itime, isize, igen, iarg1 [, iarg2 [, ...]]
```

- **giVar**: a variable name. Each function is stored in an i-variable. Usually you want to have access to it from every instrument, so a gi-variable (global initialization variable) is given.
- **ifn**: a number for the function table. If you type in 0, you give Csound the job to choose a number, which is mostly preferable.

The other parameters (size, GEN number, individual arguments) are the same as in the f-statement in the score. As this GEN call is now a part of the orchestra, each argument is separated from the next by a comma (not by a space or tab like in the score).

So this is the same example as above, but now with the function tables being generated in the orchestra header:

EXAMPLE 03D02.csd

```
<CsoundSynthesizer>
<CsInstruments>
;Example by Joachim Heintz

giFt1 ftgen 1, 0, -10, -2, 1.1, 2.2, 3.3, 5.5, 8.8, 13.13, 21.21, 34.34, 55.55,
89.89
giFt2 ftgen 2, 0, -10, 2, 1.1, 2.2, 3.3, 5.5, 8.8, 13.13, 21.21, 34.34, 55.55,
89.89

  instr 1; prints the values of table 1 or 2
          prints    "%nFunction Table %d:%n", p4
indx      init      0
loop:
ival      table     indx, p4
          prints    "Index %d = %f%n", indx, ival
          loop_lt   indx, 1, 10, loop
  endin

</CsInstruments>
<CsScore>
i 1 0 0 1; prints function table 1
i 1 0 0 2; prints function table 2
</CsScore>
</CsoundSynthesizer>
```

GEN01: Importing a Soundfile

GEN01 is used for importing soundfiles stored on disk into the computer's RAM, ready for for use by a number of Csound's opcodes in the orchestra. A typical ftgen statement for this import might be the following:

```
varname             ifn itime isize igen Sfilnam       iskip iformat ichn
giFile    ftgen     0,  0,    0,    1,   "myfile.wav", 0,    0,      0
```

- **varname**, **ifn**, **itime**: These arguments have the same meaning as explained above in reference to GEN02.
- **isize**: Usually you won't know the length of your soundfile in samples, and want to have a table length which includes exactly all the samples. This is done by setting **isize=0**. (Note that some opcodes may need a power-of-two table. In this case you can not use this option, but must calculate the next larger power-of-two value as size for the function table.)
- **igen**: As explained in the previous subchapter, this is always the place for indicating the number of the GEN Routine which must be used. As always, a positive number means normalizing, which is usually convenient for audio samples.
- **Sfilnam**: The name of the soundfile in double quotes. Similar to other audio programming languages, Csound recognizes just the name if your .csd and the soundfile are in the same folder. Otherwise, give the full path. (You can also include the folder via the "SSDIR" variable, or add the folder via the "--env:NAME+=VALUE" option.)
- **iskip**: The time in seconds you want to skip at the beginning of the soundfile. 0 means reading from the beginning of the file.
- **iformat**: Usually 0, which means: read the sample format from the soundfile header.
- **ichn**: 1 = read the first channel of the soundfile into the table, 2 = read the second channel, etc. 0 means that all channels are read.

The next example plays a short sample. You can download it here. Copy the text below, save it to the same location as the "fox.wav" soundfile, and it should work. Reading the function table is done here with the poscil3 opcode which can deal with non-power-of-two tables.

EXAMPLE 03D03.csd

```
<CsoundSynthesizer>
<CsOptions>
-odac
</CsOptions>
<CsInstruments>
;Example by Joachim Heintz
sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

giSample  ftgen     0, 0, 0, 1, "fox.wav", 0, 0, 1

  instr 1
itablen   =         ftlen(giSample) ;length of the table
idur      =         itablen / sr ;duration
aSamp     poscil3   .5, 1/idur, giSample
          outs      aSamp, aSamp
  endin

</CsInstruments>
<CsScore>
i 1 0 2.757
</CsScore>
</CsoundSynthesizer>
```

GEN10: Creating a Waveform

The third example for generating a function table covers one classical case: building a function table which stores one cycle of a waveform. This waveform is then read by an oscillator to produce a sound.

There are many GEN Routines to achieve this. The simplest one is GEN10. It produces a waveform by adding sine waves which have the "harmonic" frequency relations 1 : 2 : 3 : 4 ... After the usual arguments for function table number, start, size and gen routine number, which are the first four arguments in ftgen for all GEN Routines, you must specify for GEN10 the relative strengths of the harmonics. So, if you just provide one argument, you will end up with a sine wave (1st harmonic). The next argument is the strength of the 2nd harmonic, then the 3rd, and so on. In this way, you can build the standard harmonic waveforms by sums of sinoids. This is done in the next example by instruments 1-5. Instrument 6 uses the sine wavetable twice: for generating both the sound and the envelope.

EXAMPLE 03D04.csd

```
<CsoundSynthesizer>
```

```
<CsOptions>
-odac
</CsOptions>
<CsInstruments>
;Example by Joachim Heintz
sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

giSine    ftgen     0, 0, 2^10, 10, 1
giSaw     ftgen     0, 0, 2^10, 10, 1, 1/2, 1/3, 1/4, 1/5, 1/6, 1/7, 1/8, 1/9
giSquare  ftgen     0, 0, 2^10, 10, 1, 0, 1/3, 0, 1/5, 0, 1/7, 0, 1/9
giTri     ftgen     0, 0, 2^10, 10, 1, 0, -1/9, 0, 1/25, 0, -1/49, 0, 1/81
giImp     ftgen     0, 0, 2^10, 10, 1, 1, 1, 1, 1, 1, 1, 1, 1

  instr 1 ;plays the sine wavetable
aSine     poscil    .2, 400, giSine
aEnv      linen     aSine, .01, p3, .05
          outs      aEnv, aEnv
  endin

  instr 2 ;plays the saw wavetable
aSaw      poscil    .2, 400, giSaw
aEnv      linen     aSaw, .01, p3, .05
          outs      aEnv, aEnv
  endin

  instr 3 ;plays the square wavetable
aSqu      poscil    .2, 400, giSquare
aEnv      linen     aSqu, .01, p3, .05
          outs      aEnv, aEnv
  endin

  instr 4 ;plays the triangular wavetable
aTri      poscil    .2, 400, giTri
aEnv      linen     aTri, .01, p3, .05
          outs      aEnv, aEnv
  endin

  instr 5 ;plays the impulse wavetable
aImp      poscil    .2, 400, giImp
aEnv      linen     aImp, .01, p3, .05
          outs      aEnv, aEnv
  endin

  instr 6 ;plays a sine and uses the first half of its shape as envelope
aEnv      poscil    .2, 1/6, giSine
aSine     poscil    aEnv, 400, giSine
          outs      aSine, aSine
  endin

</CsInstruments>
<CsScore>
i 1 0 3
i 2 4 3
i 3 8 3
i 4 12 3
i 5 16 3
i 6 20 3
</CsScore>
</CsoundSynthesizer>
```

HOW TO WRITE VALUES TO A FUNCTION TABLE

As we saw, each GEN Routine generates a function table, and by doing this, it writes values into it. But in certain cases you might first want to create an empty table, and then write the values into it later. This section is about how to do this.

Actually it is not correct to speak of an "empty table". If Csound creates an "empty" table, in fact it writes zeros to the indices which are not specified. This is perhaps the easiest method of creating an "empty" table for 100 values:

```
giEmpty   ftgen     0, 0, -100, 2, 0
```

The basic opcode which writes values to existing function tables is tablew and its i-time descendant tableiw. Note that you may have problems with some features if your table is not a power-of-two size . In this case, you can also use tabw / tabw_i, but they don't have the offset- and the wraparound-feature. As usual, you must differentiate if your signal (variable) is i-rate, k-rate or a-rate. The usage is simple and differs just in the class of values you want to write to the table (i-, k- or a-variables):

```
          tableiw   isig, indx, ifn [, ixmode] [, ixoff] [, iwgmode]
          tablew    ksig, kndx, ifn [, ixmode] [, ixoff] [, iwgmode]
          tablew    asig, andx, ifn [, ixmode] [, ixoff] [, iwgmode]
```

- **isig**, **ksig**, **asig** is the value (variable) you want to write into specified locations of the table;
- **indx**, **kndx**, **andx** is the location (index) where you write the value;
- **ifn** is the function table you want to write in;
- **ixmode** gives the choice to write by raw indices (counting from 0 to size-1), or by a normalized writing mode in which the start and end of each table are always referred as 0 and 1 (not depending on the length of the table). The default is ixmode=0 which means the raw index mode. A value not equal to zero for ixmode changes to the normalized index mode.
- **ixoff** (default=0) gives an index offset. So, if indx=0 and ixoff=5, you will write at index 5.
- **iwgmode** tells what you want to do if your index is larger than the size of the table. If iwgmode=0 (default), any index larger than possible is written at the last possible index. If iwgmode=1, the indices are wrapped around. For instance, if your table size is 8, and your index is 10, in the wraparound mode the value will be written at index 2.

Here are some examples for i-, k- and a-rate values.

i-Rate Example

The following example calculates the first 12 values of a Fibonacci series and writes it to a table. This table has been created first in the header (filled with zeros). Then instrument 1 calculates the values in an i-time loop and writes them to the table with tableiw. Instrument 2 just serves to print the values.

EXAMPLE 03D05.csd

```
<CsoundSynthesizer>
<CsInstruments>
;Example by Joachim Heintz

giFt      ftgen     0, 0, -12, -2, 0

  instr 1; calculates first 12 fibonacci values and writes them to giFt
istart    =         1
inext     =         2
indx      =         0
loop:
          tableiw   istart, indx, giFt ;writes istart to table
istartold =         istart ;keep previous value of istart
istart    =         inext ;reset istart for next loop
inext     =         istartold + inext ;reset inext for next loop
          loop_lt   indx, 1, 12, loop
  endin

  instr 2; prints the values of the table
          prints    "%nContent of Function Table:%n"
indx      init      0
loop:
ival      table     indx, giFt
          prints    "Index %d = %f%n", indx, ival
          loop_lt   indx, 1, ftlen(giFt), loop
  endin

</CsInstruments>
<CsScore>
i 1 0 0
i 2 0 0
</CsScore>
</CsoundSynthesizer>
```

k-Rate Example

The next example writes a k-signal continuously into a table. This can be used to record any kind of user input, for instance by MIDI or widgets. It can also be used to record random movements of k-signals, like here:

EXAMPLE 03D06.csd

```
<CsoundSynthesizer>
```

```
<CsOptions>
-odac
</CsOptions>
<CsInstruments>
;Example by Joachim Heintz
sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

giFt      ftgen     0, 0, -5*kr, 2, 0; size for 5 seconds of recording
giWave    ftgen     0, 0, 2^10, 10, 1, .5, .3, .1; waveform for oscillator
          seed      0

; - recording of a random frequency movement for 5 seconds, and playing it
  instr 1
kFreq     randomi   400, 1000, 1 ;random frequency
aSnd      poscil    .2, kFreq, giWave ;play it
          outs      aSnd, aSnd
;;record the k-signal
          prints    "RECORDING!%n"
 ;create a writing pointer in the table,
 ;moving in 5 seconds from index 0 to the end
kindx     linseg    0, 5, ftlen(giFt)
 ;write the k-signal
          tablew    kFreq, kindx, giFt
  endin

  instr 2; read the values of the table and play it again
;;read the k-signal
          prints    "PLAYING!%n"
 ;create a reading pointer in the table,
 ;moving in 5 seconds from index 0 to the end
kindx     linseg    0, 5, ftlen(giFt)
 ;read the k-signal
kFreq     table     kindx, giFt
aSnd      oscil3    .2, kFreq, giWave; play it
          outs      aSnd, aSnd
  endin

</CsInstruments>
<CsScore>
i 1 0 5
i 2 6 5
</CsScore>
</CsoundSynthesizer>
```

As you see, in this typical case of writing k-values to a table you need a moving signal for the index. This can be done using the line or linseg opcode like here, or by using a phasor. The *phasor* always moves from 0 to 1 in a certain frequency. So, if you want the *phasor* to move from 0 to 1 in 5 seconds, you must set the frequency to 1/5. By setting the *ixmode* argument of *tablew* to 1, you can use the *phasor* output directly as writing pointer. So this is an alternative version of instrument 1 taken from the previous example:

```
instr 1; recording of a random frequency movement for 5 seconds, and playing it
kFreq     randomi   400, 1000, 1; random frequency
aSnd      oscil3    .2, kFreq, giWave; play it
          outs      aSnd, aSnd
;;record the k-signal with a phasor as index
          prints    "RECORDING!%n"
 ;create a writing pointer in the table,
 ;moving in 5 seconds from index 0 to the end
kindx     phasor    1/5
 ;write the k-signal
          tablew    kFreq, kindx, giFt, 1
endin
```

a-Rate Example

Recording an audio signal is quite similar to recording a control signal. You just need an a-signal as input and also as index. The first example shows first the recording of a random audio signal. If you have live audio input, you can then record your input for 5 seconds.

EXAMPLE 03D07.csd

```
<CsoundSynthesizer>
<CsOptions>
-iadc -odac
</CsOptions>
<CsInstruments>
;Example by Joachim Heintz
```

```
sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

giFt      ftgen     0, 0, -5*sr, 2, 0; size for 5 seconds of recording audio
          seed      0

  instr 1 ;generating a band filtered noise for 5 seconds, and recording it
aNois     rand      .2
kCfreq    randomi   200, 2000, 3; random center frequency
aFilt     butbp     aNois, kCfreq, kCfreq/10; filtered noise
aBal      balance   aFilt, aNois, 1; balance amplitude
          outs      aBal, aBal
;;record the audiosignal with a phasor as index
          prints    "RECORDING FILTERED NOISE!%n"
 ;create a writing pointer in the table,
 ;moving in 5 seconds from index 0 to the end
aindx     phasor    1/5
 ;write the k-signal
          tablew    aBal, aindx, giFt, 1
  endin

  instr 2 ;read the values of the table and play it
          prints    "PLAYING FILTERED NOISE!%n"
aindx     phasor    1/5
aSnd      table3    aindx, giFt, 1
          outs      aSnd, aSnd
  endin

  instr 3 ;record live input
ktim      timeinsts ; playing time of the instrument in seconds
          prints    "PLEASE GIVE YOUR LIVE INPUT AFTER THE BEEP!%n"
kBeepEnv  linseg    0, 1, 0, .01, 1, .5, 1, .01, 0
aBeep     oscils    .2, 600, 0
          outs      aBeep*kBeepEnv, aBeep*kBeepEnv
;;record the audiosignal after 2 seconds
 if ktim > 2 then
ain       inch      1
          printks   "RECORDING LIVE INPUT!%n", 10
 ;create a writing pointer in the table,
 ;moving in 5 seconds from index 0 to the end
aindx     phasor    1/5
 ;write the k-signal
          tablew    ain, aindx, giFt, 1
 endif
  endin

  instr 4 ;read the values from the table and play it
          prints    "PLAYING LIVE INPUT!%n"
aindx     phasor    1/5
aSnd      table3    aindx, giFt, 1
          outs      aSnd, aSnd
  endin

</CsInstruments>
<CsScore>
i 1 0 5
i 2 6 5
i 3 12 7
i 4 20 5
</CsScore>
</CsoundSynthesizer>
```

HOW TO RETREIVE VALUES FROM A FUNCTION TABLE

There are two methods of reading table values. You can either use the table / tab opcodes, which are universally usable, but need an index; or you can use an oscillator for reading a table at k-rate or a-rate.

The table Opcode

The *table* opcode is quite similar in syntax to the *tableiw*/*tablew* opcode (which are explained above). It's just its counterpart in reading values from a function table (instead of writing values to it). So its output is either an i-, k- or a-signal. The main input is an index of the appropriate rate (i-index for i-output, k-index for k-output, a-index for a-output). The other arguments are as explained above for tableiw/tablew:

```
ires      table    indx, ifn [, ixmode] [, ixoff] [, iwrap]
kres      table    kndx, ifn [, ixmode] [, ixoff] [, iwrap]
```

```
ares      table    andx, ifn [, ixmode] [, ixoff] [, iwrap]
```

As table reading often requires interpolation between the table values - for instance if you read k or a-values faster or slower than they have been written in the table - Csound offers two descendants of table for interpolation: tablei interpolates linearly, whilst table3 performs cubic interpolation (which is generally preferable but is computationally slightly more expensive). Another variant is the tab_i / tab opcode which misses some features but may be preferable in some situations. If you have any problems in reading non-power-of-two tables, give them a try. They should also be faster than the table opcode, but you must take care: they include fewer built-in protection measures than *table*, *table*i and *table3* and if they are given index values that exceed the table size Csound will stop and report a performance error.
Examples of the use of the *table* opcodes can be found in the earlier examples in the How-To-Write-Values... section.

Oscillators

Reading table values using an oscillator is standard if you read tables which contain one cycle of a waveform at audio-rate. But actually you can read any table using an oscillator, either at a- or at k-rate. The advantage is that you needn't create an index signal. You can simply specify the frequency of the oscillator.
You should bear in mind that many of the oscillators in Csound will work only with power-of-two table sizes. The poscil/poscil3 opcodes do not have this restriction and offer a high precision, because they work with floating point indices, so in general it is recommended to use them.
Below is an example that demonstrates both reading a k-rate and an a-rate signal from a buffer with poscil3 (an oscillator with a cubic interpolation):

EXAMPLE 03D08.csd

```
<CsoundSynthesizer>
<CsOptions>
-iadc -odac
</CsOptions>
<CsInstruments>
;Example by Joachim Heintz
sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1
; -- size for 5 seconds of recording control data
giControl ftgen     0, 0, -5*kr, 2, 0
; -- size for 5 seconds of recording audio data
giAudio   ftgen     0, 0, -5*sr, 2, 0
giWave    ftgen     0, 0, 2^10, 10, 1, .5, .3, .1; waveform for oscillator
          seed      0

; -- ;recording of a random frequency movement for 5 seconds, and playing it
  instr 1
kFreq     randomi   400, 1000, 1; random frequency
aSnd      poscil    .2, kFreq, giWave; play it
          outs      aSnd, aSnd
;;record the k-signal with a phasor as index
          prints    "RECORDING RANDOM CONTROL SIGNAL!%n"
 ;create a writing pointer in the table,
 ;moving in 5 seconds from index 0 to the end
kindx     phasor    1/5
 ;write the k-signal
          tablew    kFreq, kindx, giControl, 1
  endin

  instr 2; read the values of the table and play it with poscil
          prints    "PLAYING CONTROL SIGNAL!%n"
kFreq     poscil    1, 1/5, giControl
aSnd      poscil    .2, kFreq, giWave; play it
          outs      aSnd, aSnd
  endin

  instr 3; record live input
ktim      timeinsts ; playing time of the instrument in seconds
          prints    "PLEASE GIVE YOUR LIVE INPUT AFTER THE BEEP!%n"
kBeepEnv  linseg    0, 1, 0, .01, 1, .5, 1, .01, 0
aBeep     oscils    .2, 600, 0
          outs      aBeep*kBeepEnv, aBeep*kBeepEnv
;;record the audiosignal after 2 seconds
 if ktim > 2 then
ain       inch      1
          printks   "RECORDING LIVE INPUT!%n", 10
 ;create a writing pointer in the table,
 ;moving in 5 seconds from index 0 to the end
aindx     phasor    1/5
 ;write the k-signal
          tablew    ain, aindx, giAudio, 1
 endif
  endin

  instr 4; read the values from the table and play it with poscil
          prints    "PLAYING LIVE INPUT!%n"
aSnd      poscil    .5, 1/5, giAudio
          outs      aSnd, aSnd
  endin

</CsInstruments>
<CsScore>
i 1 0 5
i 2 6 5
i 3 12 7
i 4 20 5
</CsScore>
</CsoundSynthesizer>
```

SAVING THE CONTENTS OF A FUNCTION TABLE TO A FILE

A function table exists just as long as you run the Csound instance which has created it. If Csound terminates, all the data is lost. If you want to save the data for later use, you must write them to a file. There are several cases, depending on firstly whether you write at i-time or at k-time and secondly on what kind of file you want to write to.

Writing a File in Csound's ftsave Format at i-Time or k-Time

Any function table in Csound can easily be written to a file by the ftsave (i-time) or ftsavek (k-time) opcode. The use is very simple. The first argument specifies the filename (in double quotes), the second argument chooses between a text format (non zero) or a binary format (zero) to write, then you just give the number of the function table(s) to save.
For the following example you should end up with two textfiles in the same folder as your .csd: "i-time_save.txt" saves function table 1 (a sine wave) at i-time; "k-time_save.txt" saves function table 2 (a linear increment produced during the performance) at k-time.

EXAMPLE 03D09.csd

```
<CsoundSynthesizer>
<CsInstruments>
;Example by Joachim Heintz
sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

giWave    ftgen     1, 0, 2^7, 10, 1; sine with 128 points
giControl ftgen     2, 0, -kr, 2, 0; size for 1 second of recording control data
          seed      0

  instr 1; saving giWave at i-time
          ftsave    "i-time_save.txt", 1, 1
  endin

  instr 2; recording of a line transition between 0 and 1 for one second
kline     linseg    0, 1, 1
          tabw      kline, kline, giControl, 1
  endin

  instr 3; saving giWave at k-time
          ftsave    "k-time_save.txt", 1, 2
  endin

</CsInstruments>
<CsScore>
i 1 0 0
i 2 0 1
i 3 1 .1
</CsScore>
</CsoundSynthesizer>
```

The counterpart to ftsave/ftsavek are the opcodes ftload/ftloadk. Using them you can load the saved files into function tables.

Writing a Soundfile from a Recorded Function Table

If you have recorded your live-input to a buffer, you may want to save your buffer as a soundfile. There is no opcode in Csound which does that, but it can be done by using a k-rate loop and the fout opcode. This is shown in the next example, in instrument 2. First instrument 1 records your live input. Then instrument 2 writes the file "testwrite.wav" into the same folder as your .csd. This is done at the first k-cycle of instrument 2, by reading again and again the table values and writing them as an audio signal to disk. After this is done, the instrument is turned off by executing the turnoff statement.

EXAMPLE 03D10.csd

```
<CsoundSynthesizer>
<CsOptions>
-i adc
</CsOptions>
<CsInstruments>
;Example by Joachim Heintz
sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1
; --  size for 5 seconds of recording audio data
giAudio   ftgen     0, 0, -5*sr, 2, 0

  instr 1 ;record live input
ktim      timeinsts ; playing time of the instrument in seconds
          prints    "PLEASE GIVE YOUR LIVE INPUT AFTER THE BEEP!%n"
kBeepEnv  linseg    0, 1, 0, .01, 1, .5, 1, .01, 0
aBeep     oscils    .2, 600, 0
          outs      aBeep*kBeepEnv, aBeep*kBeepEnv
;;record the audiosignal after 2 seconds
 if ktim > 2 then
ain       inch      1
          printks   "RECORDING LIVE INPUT!%n", 10
 ;create a writing pointer in the table,
 ;moving in 5 seconds from index 0 to the end
aindx     phasor    1/5
 ;write the k-signal
          tablew    ain, aindx, giAudio, 1
 endif
  endin

  instr 2; write the giAudio table to a soundfile
Soutname  =         "testwrite.wav"; name of the output file
iformat   =         14; write as 16 bit wav file
itablen   =         ftlen(giAudio); length of the table in samples

kcnt      init      0; set the counter to 0 at start
loop:
kcnt      =         kcnt+ksmps; next value (e.g. 10 if ksmps=10)
andx      interp    kcnt-1; calculate audio index (e.g. from 0 to 9)
asig      tab       andx, giAudio; read the table values as audio signal
          fout      Soutname, iformat, asig; write asig to a file
 if kcnt <= itablen-ksmps kgoto loop; go back as long there is something to do
          turnoff   ; terminate the instrument
  endin

</CsInstruments>
<CsScore>
i 1 0 7
i 2 7 .1
</CsScore>
</CsoundSynthesizer>
```

This code can also be transformed in a User Defined Opcode. It can be found here.

Related Opcodes

ftgen: Creates a function table in the orchestra using any GEN Routine.

table / tablei / table3: Read values from a function table at any rate, either by direct indexing (table), or by linear (tablei) or cubic (table3) interpolation. These opcodes provide many options and are safe because of boundary check, but you may have problems with non-power-of-two tables.

tab_i / tab: Read values from a function table at i-rate (tab_i), k-rate or a-rate (tab). Offer no interpolation and less options than the table opcodes, but they work also for non-power-of-two tables. They do not provide a boundary check, which makes them fast but also give the user the resposability not reading any value off the table boundaries.

tableiw / tablew: Write values to a function table at i-rate (tableiw), k-rate and a-rate (tablew). These opcodes provide many options and are safe because of boundary check, but you may have problems with non-power-of-two tables.

tabw_i / tabw: Write values to a function table at i-rate (tabw_i), k-rate or a-rate (tabw). Offer less options than the tableiw/tablew opcodes, but work also for non-power-of-two tables. They do not provide a boundary check, which makes them fast but also give the user the resposability not writing any value off the table boundaries.

poscil / poscil3: Precise oscillators for reading function tables at k- or a-rate, with linear (poscil) or cubic (poscil3) interpolation. They support also non-power-of-two tables, so it's usually recommended to use them instead of the older oscili/oscil3 opcodes. Poscil has also a-rate input for amplitude and frequency, while poscil3 has just k-rate input.

oscili / oscil3: The standard oscillators in Csound for reading function tables at k- or a-rate, with linear (oscili) or cubic (oscil3) interpolation. They support all rates for the amplitude and frequency input, but are restricted to power-of-two tables. Particularily for long tables and low frequencies they are not as precise as the poscil/poscil3 oscillators.

ftsave / ftsavek: Save a function table as a file, at i-time (ftsave) or k-time (ftsavek). This can be a text file or a binary file, but not a soundfile. If you want to save a soundfile, use the User Defined Opcode TableToSF.

ftload / ftloadk: Load a function table which has been written by ftsave/ftsavek.

line / linseg / phasor: Can be used to create index values which are needed to read/write k- or a-signals with the table/tablew or tab/tabw opcodes.

17. TRIGGERING INSTRUMENT EVENTS

The basic concept of Csound from the early days of the program is still valent and fertile because it is a familiar musical one. You create a set of instruments and instruct them to play at various times. These calls of instrument instances, and their execution, are called "instrument events".

This scheme of instruments and events can be instigated in a number of ways. In the classical approach you think of an "orchestra" with a number of musicians playing from a "score", but you can also trigger instruments using any kind of live input: from MIDI, from OSC, from the command line, from a GUI (such as Csound's FLTK widgets or QuteCsound's widgets), from the API (also used in QuteCsound's Live Event Sheet). Or you can create a kind of "master instrument", which is always on, and triggers other instruments using opcodes designed for this task, perhaps under certain conditions: if the live audio input from a singer has been detected to have a base frequency greater than 1043 Hz, then start an instrument which plays a soundfile of broken glass...

This chapter is about the various ways to trigger instrument events whether that be from the score, by using MIDI, by using widgets, through using conditionals or by using loops.

ORDER OF EXECUTION

Whatever you do in Csound with instrument events, you must bear in mind the order of execution that has been explained in the first chapter of this section about the *Initialization and Performance Pass*: instruments are executed one by one, both in the initialization pass and in each control cycle, and the order is determined **by the instrument number**. So if you have an instrument which triggers another instrument, it should usually have the lower number. If, for instance, instrument 10 calls instrument 20 in a certain control cycle, instrument 20 will execute the event in the same control cycle. But if instrument 20 calls instrument 10, then instrument 10 will execute the event only in the next control cycle.

INSTRUMENT EVENTS FROM THE SCORE

This is the classical way of triggering instrument events: you write a list in the score section of a .csd file. Each line which begins with an "i", is an instrument event. As this is very simple, and examples can be found easily, let us focus instead on some additional features which can be useful when you work in this way. Documentation for these features can be found in the Score Statements section of the Canonical Csound Reference Manual. Here are some examples:

EXAMPLE 03E01.csd

```
<CsoundSynthesizer>
<CsOptions>
-odac
</CsOptions>
<CsInstruments>
;Example by Joachim Heintz
sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

giWav     ftgen     0, 0, 2^10, 10, 1, .5, .3, .1

  instr 1
kFadout   init      1
krel      release   ;returns "1" if last k-cycle
 if krel == 1 && p3 < 0 then ;if so, and negative p3:
          xtratim   .5       ;give 0.5 extra seconds
kFadout   linseg    1, .5, 0 ;and make fade out
 endif
kEnv      linseg    0, .01, p4, abs(p3)-.1, p4, .09, 0; normal fade out
aSig      poscil    kEnv*kFadout, p5, giWav
          outs      aSig, aSig
  endin

</CsInstruments>
<CsScore>
```

```
t 0 120                      ;set tempo to 120 beats per minute
i    1    0    1    .2   400 ;play instr 1 for one second
i    1    2   -10   .5   500 ;play instr 1 indefinetely (negative p3)
i   -1    5    0             ;turn it off (negative p1)
; -- turn on instance 1 of instr 1 one sec after the previous start
i    1.1  ^+1  -10  .2   600
i    1.2  ^+2  -10  .2   700 ;another instance of instr 1
i   -1.2  ^+2  0             ;turn off 1.2
; -- turn off 1.1 (dot = same as the same p-field above)
i   -1.1  ^+1  .
s                            ;end of a section, so time begins from new at zero
i    1    1    1    .2   800
r 5                          ;repeats the following line (until the next "s")
i    1   .25  .25   .2   900
s
v 2                          ;lets time be double as long
i    1    0    2    .2   1000
i    1    1    1    .2   1100
s
v 0.5                        ;lets time be half as long
i    1    0    2    .2   1200
i    1    1    1    .2   1300
s                            ;time is normal now again
i    1    0    2    .2   1000
i    1    1    1    .2   900
s
; -- make a score loop (4 times) with the variable "LOOP"{4 LOOP
i    1    [0 + 4 * $LOOP.]    3    .2   [1200 - $LOOP. * 100]
i    1    [1 + 4 * $LOOP.]    2    .    [1200 - $LOOP. * 200]
i    1    [2 + 4 * $LOOP.]    1    .    [1200 - $LOOP. * 300]
}
e
</CsScore>
</CsoundSynthesizer>
```

Triggering an instrument with an indefinite duration by setting p3 to any negative value, and stopping it by a negative p1 value, can be an important feature for live events. If you turn instruments off in this way you may have to add a fade out segment. One method of doing this is shown in the instrument above with a combination of the release and the xtratim opcodes. Also note that you can start and stop certain instances of an instrument with a floating point number as p1.

USING MIDI NOTEON EVENTS

Csound has a particular feature which makes it very simple to trigger instrument events from a MIDI keyboard. Each MIDI Note-On event can trigger an instrument, and the related Note-Off event of the same key stops the related instrument instance. This is explained more in detail in the chapter *Triggering Instrument Instances* in the MIDI section of this manual. Here, just a small example is shown. Simply connect your MIDI keyboard and it should work.

EXAMPLE 03E02.csd

```
<CsoundSynthesizer>
<CsOptions>
-Ma -odac
</CsOptions>
<CsInstruments>
;Example by Joachim Heintz
sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

giSine    ftgen     0, 0, 2^10, 10, 1
          massign   0, 1; assigns all midi channels to instr 1

  instr 1
iFreq     cpsmidi   ;gets frequency of a pressed key
iAmp      ampmidi   8 ;gets amplitude and scales 0-8
iRatio    random    .9, 1.1 ;ratio randomly between 0.9 and 1.1
aTone     foscili   .1, iFreq, 1, iRatio/5, iAmp+1, giSine ;fm
aEnv      linenr    aTone, 0, .01, .01 ; avoiding clicks at the note-end
          outs      aEnv, aEnv
  endin

</CsInstruments>
<CsScore>
f 0 36000; play for 10 hours
e
</CsScore>
```

```
</CsoundSynthesizer>
```

USING WIDGETS

If you want to trigger an instrument event in realtime with a Graphical User Interface, it is usually a "Button" widget which will do this job. We will see here a simple example; first implemented using Csound's FLTK widgets, and then using QuteCsound's widgets.

FLTK Button

This is a very simple example demonstrating how to trigger an instrument using an FLTK button. A more extended example can be found here.

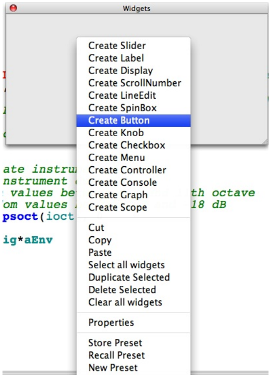
EXAMPLE 03E03.csd

```
<CsoundSynthesizer>
<CsOptions>
-odac
</CsOptions>
<CsInstruments>
;Example by Joachim Heintz
sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

      ; -- create a FLTK panel --
          FLpanel   "Trigger By FLTK Button", 300, 100, 100, 100
      ; -- trigger instr 1 (equivalent to the score line "i 1 0 1")k1, ih1
FLbutton  "Push me!", 0, 0, 1, 150, 40, 10, 25, 0, 1, 0, 1
      ; -- trigger instr 2
k2, ih2   FLbutton  "Quit", 0, 0, 1, 80, 40, 200, 25, 0, 2, 0, 1
          FLpanelEnd; end of the FLTK panel section
          FLrun     ; run FLTK
          seed      0; random seed different each time

  instr 1
idur      random    .5, 3; recalculate instrument duration
p3        =         idur; reset instrument duration
ioct      random    8, 11; random values between 8th and 11th octave
idb       random    -18, -6; random values between -6 and -18 dB
aSig      oscils    ampdb(idb), cpsoct(ioct), 0
aEnv      transeg   1, p3, -10, 0
          outs      aSig*aEnv, aSig*aEnv
  endin

instr 2
          exitnow
endin

</CsInstruments>
<CsScore>
f 0 36000
e
</CsScore>
</CsoundSynthesizer>
```

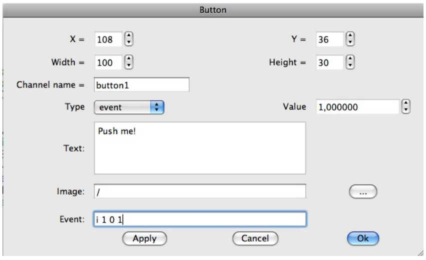
Note that in this example the duration of an instrument event is recalculated when the instrument is inititalized. This is done using the statement "p3 = i...". This can be a useful technique if you want the duration that an instrument plays for to be different each time it is called. In this example duration is the result of a random function'. The duration defined by the FLTK button will be overwriten by any other calculation within the instrument itself at i-time.

QuteCsound Button

In QuteCsound, a button can be created easily from the submenu in a widget panel:

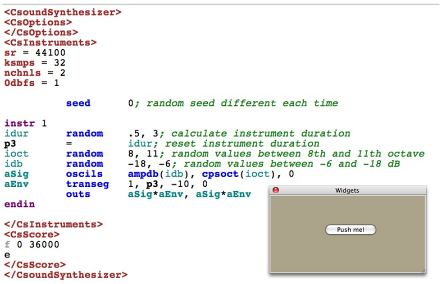


In the Properties Dialog of the button widget, make sure you have selected "event" as Type. Insert a Channel name, and at the bottom type in the event you want to trigger - as you would if writing a line in the score.



X = 108
Y = 36
Width = 100
Height = 30
Channel name = button1
Type: event
Value: 1,000000
Text: Push me!
Image: /
Event: i 1 0 1

In your Csound code, you need nothing more than the instrument you want to trigger:



```
<CsoundSynthesizer>
<CsOptions>
</CsOptions>
<CsInstruments>
sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

          seed      0; random seed different each time

instr 1
idur      random    .5, 3; calculate instrument duration
p3        =         idur; reset instrument duration
ioct      random    8, 11; random values between 8th and 11th octave
idb       random    -18, -6; random values between -6 and -18 dB
aSig      oscils    ampdb(idb), cpsoct(ioct), 0
aEnv      transeg   1, p3, -10, 0
          outs      aSig*aEnv, aSig*aEnv
endin

</CsInstruments>
<CsScore>
f 0 36000
e
</CsScore>
</CsoundSynthesizer>
```

For more information about QuteCsound, read the QuteCsound chapter in the 'Frontends' section of this manual.

USING A REALTIME SCORE (LIVE EVENT SHEET)

Command Line With The -L stdin Option

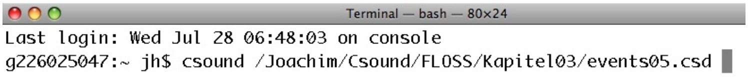
If you use any .csd with the option "-L stdin" (and the -odac option for realtime output), you can type any score line in realtime (sorry, this does not work for Windows). For instance, save this .csd anywhere and run it from the command line:

EXAMPLE 03E04.csd

```
<CsoundSynthesizer>
<CsOptions>
-L stdin -odac
</CsOptions>
<CsInstruments>
;Example by Joachim Heintz
sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

          seed      0; random seed different each time
```

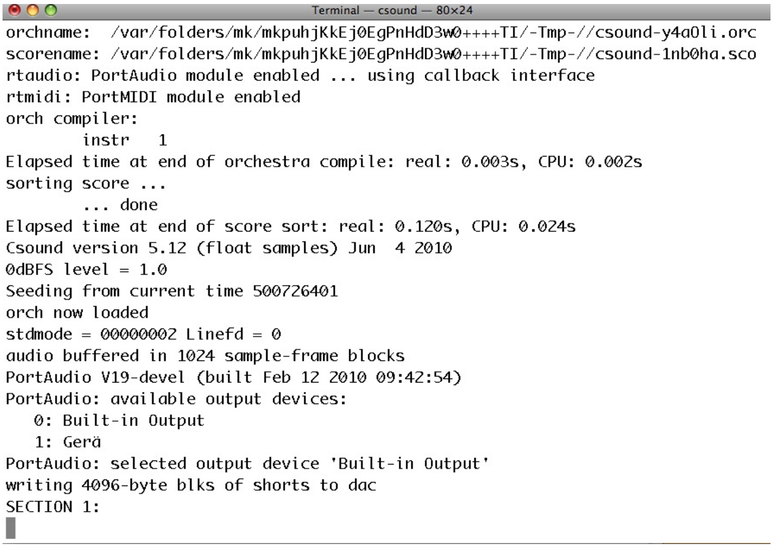
```
  instr 1
idur      random    .5, 3; calculate instrument duration
p3        =         idur; reset instrument duration
ioct      random    8, 11; random values between 8th and 11th octave
idb       random    -18, -6; random values between -6 and -18 dB
aSig      oscils    ampdb(idb), cpsoct(ioct), 0
aEnv      transeg   1, p3, -10, 0
          outs      aSig*aEnv, aSig*aEnv
  endin

</CsInstruments>
<CsScore>
f 0 36000
e
</CsScore>
</CsoundSynthesizer>
```

If you run it by typing and returning a commandline like this ...



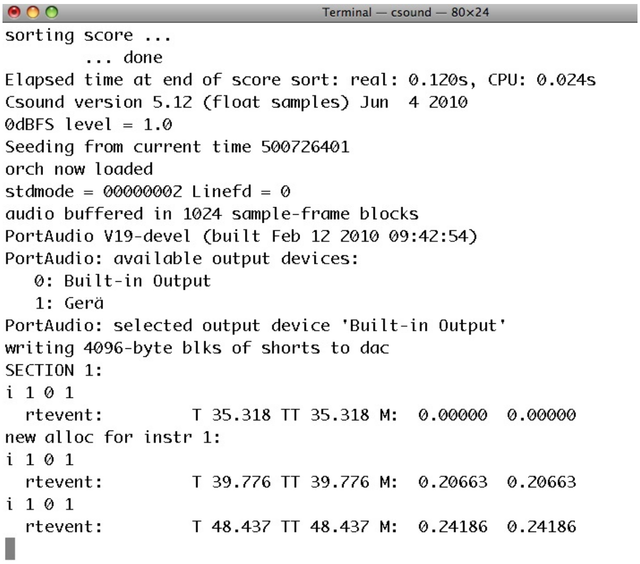
```
Last login: Wed Jul 28 06:48:03 on console
g226025047:~ jh$ csound /Joachim/Csound/FLOSS/Kapitel03/events05.csd
```

... you should get a prompt at the end of the Csound messages:



```
orchname:  /var/folders/mk/mkpuhjKkEj0EgPnHdD3w0++++TI/-Tmp-//csound-y4a0li.orc
scorename: /var/folders/mk/mkpuhjKkEj0EgPnHdD3w0++++TI/-Tmp-//csound-1nb0ha.sco
rtaudio: PortAudio module enabled ... using callback interface
rtmidi: PortMIDI module enabled
orch compiler:
        instr   1
Elapsed time at end of orchestra compile: real: 0.003s, CPU: 0.002s
sorting score ...
        ... done
Elapsed time at end of score sort: real: 0.120s, CPU: 0.024s
Csound version 5.12 (float samples) Jun  4 2010
0dBFS level = 1.0
Seeding from current time 500726401
orch now loaded
stdmode = 00000002 Linefd = 0
audio buffered in 1024 sample-frame blocks
PortAudio V19-devel (built Feb 12 2010 09:42:54)
PortAudio: available output devices:
   0: Built-in Output
   1: Gerä
PortAudio: selected output device 'Built-in Output'
writing 4096-byte blks of shorts to dac
SECTION 1:
```

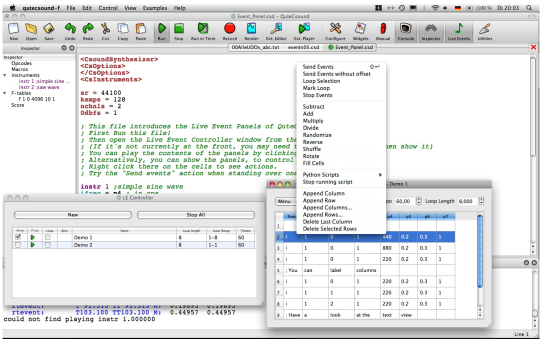
If you now type the line "i 1 0 1" and press return, you should hear that instrument 1 has been executed. After three times your messages may look like this:



```
sorting score ...
        ... done
Elapsed time at end of score sort: real: 0.120s, CPU: 0.024s
Csound version 5.12 (float samples) Jun  4 2010
0dBFS level = 1.0
Seeding from current time 500726401
orch now loaded
stdmode = 00000002 Linefd = 0
audio buffered in 1024 sample-frame blocks
PortAudio V19-devel (built Feb 12 2010 09:42:54)
PortAudio: available output devices:
   0: Built-in Output
   1: Gerä
PortAudio: selected output device 'Built-in Output'
writing 4096-byte blks of shorts to dac
SECTION 1:
i 1 0 1
  rtevent:        T 35.318 TT 35.318 M:  0.00000  0.00000
new alloc for instr 1:
i 1 0 1
  rtevent:        T 39.776 TT 39.776 M:  0.20663  0.20663
i 1 0 1
  rtevent:        T 48.437 TT 48.437 M:  0.24186  0.24186
█
```

QuteCsound's Live Event Sheet

In general, this is the method that QuteCsound uses and it is made available to the user in a flexible environment called the Live Event Sheet. This is just a screenshot of the current (QuteCsound 0.6.0) example of the Live Event Sheet in QuteCsound:



```
<CsoundSynthesizer>
<CsOptions>
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 128
nchnls = 2
0dbfs = 1

; This file introduces the Live Event Panels of QuteC
; First Run this file!
; Then open the Live Event Controller window from the
; (If it's not currently at the front, you may need t                    en show it)
; You can play the contents of the panels by clicking
; Alternatively, you can show the panels, to control
; Right click there on the cells to see actions.
; Try the "Send events" action when standing over one

instr 1 ;simple sine wave
```

Show	Play	Loop	Sync	Name	Loop length	Loop Range	Tempo
☑	▶	☐		Demo 1	8	1-8	60
☐	▶	☐		Demo 2	8	1-1	60

	Even				p4	p5	p6	p7
1								
2	i	1	0	1	440	0.2	0.3	1
3	i	1	0	1	880	0.2	0.3	1
4	i	1	0	1	220	0.2	0.3	1
5	; You	can	label	columns				
6	i	1	0	1	220	0.2	0.3	1
7	i	1	1	1	220	0.2	0.3	1
8	i	1	2	1	220	0.2	0.3	1
9	; Have	a	look	at the	text	view		

```
  rtevent:        T103.100 TT103.100 M:  0.44957  0.44957
could not find playing instr 1.000000
```

Have a look in the QuteCsound frontend to see more of the possibilities of "firing" live instrument events using the Live Event Sheet.

BY CONDITIONS

We have discussed first the classical method of triggering instrument events from the score section of a .csd file, then we went on to look at different methods of triggering real time events using MIDI, by using widgets, and by using score lines inserted live. We will now look at the Csound orchestra itself and to some methods by which an instrument can internally trigger another instrument. The pattern of triggering could be governed by conditionals, or by different kinds of loops. As this "master" instrument can itself be triggered by a realtime event, you have unlimited options available for combining the different methods.

Let's start with conditionals. If we have a realtime input, we may want to define a threshold, and trigger an event

1. if we cross the threshold from below to above;
2. if we cross the threshold from above to below.

In Csound, this could be implemented using an orchestra of three instruments. The first instrument is the master instrument. It receives the input signal and investigates whether that signal is crossing the threshold and if it does whether it is crossing from low to high or from high to low. If it crosses the threshold from low ot high the second instrument is triggered, if it crosses from high to low the third instrument is triggered.

EXAMPLE 03E05.csd

```
<CsoundSynthesizer>
<CsOptions>
-iadc -odac
</CsOptions>
<CsInstruments>
;Example by Joachim Heintz
sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

          seed      0; random seed different each time

  instr 1; master instrument
ichoose   =         p4; 1 = real time audio, 2 = random amplitude movement
ithresh   =         -12; threshold in dB
kstat     init      1; 1 = under the threshold, 2 = over the threshold
;;CHOOSE INPUT SIGNAL
 if ichoose == 1 then
ain       inch      1
 else
kdB       randomi   -18, -6, 1
ain       pinkish   ampdb(kdB)
 endif
;;MEASURE AMPLITUDE AND TRIGGER SUBINSTRUMENTS IF THRESHOLD IS CROSSED
afoll     follow    ain, .1; measure mean amplitude each 1/10 second
kfoll     downsamp  afoll
 if kstat == 1 && dbamp(kfoll) > ithresh then; transition down->up
          event     "i", 2, 0, 1; call instr 2
          printks   "Amplitude = %.3f dB%n", 0, dbamp(kfoll)
kstat     =         2; change status to "up"
 elseif kstat == 2 && dbamp(kfoll) < ithresh then; transition up->down
          event     "i", 3, 0, 1; call instr 3
          printks   "Amplitude = %.3f dB%n", 0, dbamp(kfoll)
kstat     =         1; change status to "down"
 endif
  endin

  instr 2; triggered if threshold has been crossed from down to up
asig      oscils    .2, 500, 0
aenv      transeg   1, p3, -10, 0
          outs      asig*aenv, asig*aenv
  endin

  instr 3; triggered if threshold has been crossed from up to down
asig      oscils    .2, 400, 0
aenv      transeg   1, p3, -10, 0
          outs      asig*aenv, asig*aenv
  endin

</CsInstruments>
<CsScore>
i 1 0 1000 2 ;change p4 to "1" for live input
e
</CsScore>
</CsoundSynthesizer>
```

USING I-RATE LOOPS FOR CALCULATING A POOL OF INSTRUMENT EVENTS

You can perform a number of calculations at init-time which lead to a list of instrument events. In this way you are producing a score, but inside an instrument. The score events are then executed later.

Using this opportunity we can introduce the scoreline / scoreline_i opcode. It is quite similar to the event / event_i opcode but has two major benefits:

- You can write more than one scoreline by using "{{" at the beginning and "}}" at the end.
- You can send a string to the subinstrument (which is not possible with the event opcode).

Let's look at a simple example for executing score events from an instrument using the scoreline opcode:

EXAMPLE 03E06.csd

```
<CsoundSynthesizer>
<CsOptions>
-odac
</CsOptions>
<CsInstruments>
;Example by Joachim Heintz
sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

          seed      0; random seed different each time

  instr 1 ;master instrument with event pool
          scoreline_i {{i 2 0 2 7.09
                        i 2 2 2 8.04
                        i 2 4 2 8.03
                        i 2 6 1 8.04}}
  endin

  instr 2 ;plays the notes
asig      pluck     .2, cpspch(p4), cpspch(p4), 0, 1
aenv      transeg   1, p3, 0, 0
          outs      asig*aenv, asig*aenv
  endin

</CsInstruments>
<CsScore>
i 1 0 7
e
</CsScore>
</CsoundSynthesizer>
```

With good right, you might say: "OK, that's nice, but I can also write scorelines in the score itself!" That's right, but the advantage with the *scoreline_i* method is that you can **render** the score events in an instrument, and **then** send them out to one or more instruments to execute them. This can be done with the sprintf opcode, which produces the string for scoreline in an i-time loop (see the chapter about control structures).

EXAMPLE 03E07.csd

```
<CsoundSynthesizer>
<CsOptions>
-odac
</CsOptions>
<CsInstruments>
;Example by Joachim Heintz
sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

giPch     ftgen     0, 0, 4, -2, 7.09, 8.04, 8.03, 8.04
          seed      0; random seed different each time

  instr 1 ; master instrument with event pool
itimes    =         7 ;number of events to produce
icnt      =         0 ;counter
istart    =         0
Slines    =         ""
loop:               ;start of the i-time loop
idur      random    1, 2.9999 ;duration of each note:
idur      =         int(idur) ;either 1 or 2
itabndx   random    0, 3.9999 ;index for the giPch table:
itabndx   =         int(itabndx) ;0-3
ipch      table     itabndx, giPch ;random pitch value from the table
Sline     sprintf   "i 2 %d %d %.2f\n", istart, idur, ipch ;new scoreline
Slines    strcat    Slines, Sline ;append to previous scorelines
istart    =         istart + idur ;recalculate start for next scoreline
          loop_lt   icnt, 1, itimes, loop ;end of the i-time loop
          puts      Slines, 1 ;print the scorelines
          scoreline_i Slines ;execute them
iend      =         istart + idur ;calculate the total duration
p3        =         iend ;set p3 to the sum of all durations
          print     p3 ;print it
  endin

  instr 2 ;plays the notes
asig      pluck     .2, cpspch(p4), cpspch(p4), 0, 1
aenv      transeg   1, p3, 0, 0
          outs      asig*aenv, asig*aenv
  endin
```

```
</CsInstruments>
<CsScore>
i 1 0 1 ;p3 is automatically set to the total duration
e
</CsScore>
</CsoundSynthesizer>
```

In this example, seven events have been rendered in an i-time loop in instrument 1. The result is stored in the string variable *Slines*. This string is given at i-time to scoreline_i, which executes them then one by one according to their starting times (p2), durations (p3) and other parameters.

If you have many scorelines which are added in this way, you may run to Csound's maximal string length. By default, it is 255 characters. It can be extended by adding the option "-+max_str_len=10000" to Csound's maximum string length of 9999 characters. Instead of collecting all score lines in a single string, you can also execute them inside the i-time loop. Also in this way all the single score lines are added to Csound's event pool. The next example shows an alternative version of the previous one by adding the instrument events one by one in the i-time loop, either with event_i (instr 1) or with scoreline_i (instr 2):

EXAMPLE 03E08.csd

```
<CsoundSynthesizer>
<CsOptions>
-odac
</CsOptions>
<CsInstruments>
;Example by Joachim Heintz
sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

giPch     ftgen     0, 0, 4, -2, 7.09, 8.04, 8.03, 8.04
          seed      0; random seed different each time

  instr 1; master instrument with event_i
itimes    =         7; number of events to produce
icnt      =         0; counter
istart    =         0
loop:               ;start of the i-time loop
idur      random    1, 2.9999; duration of each note:
idur      =         int(idur); either 1 or 2
itabndx   random    0, 3.9999; index for the giPch table:
itabndx   =         int(itabndx); 0-3
ipch      table     itabndx, giPch; random pitch value from the table
          event_i   "i", 3, istart, idur, ipch; new instrument event
istart    =         istart + idur; recalculate start for next scoreline
          loop_lt   icnt, 1, itimes, loop; end of the i-time loop
iend      =         istart + idur; calculate the total duration
p3        =         iend; set p3 to the sum of all durations
          print     p3; print it
  endin

  instr 2; master instrument with scoreline_i
itimes    =         7; number of events to produce
icnt      =         0; counter
istart    =         0
loop:               ;start of the i-time loop
idur      random    1, 2.9999; duration of each note:
idur      =         int(idur); either 1 or 2
itabndx   random    0, 3.9999; index for the giPch table:
itabndx   =         int(itabndx); 0-3
ipch      table     itabndx, giPch; random pitch value from the table
Sline     sprintf   "i 3 %d %d %.2f", istart, idur, ipch; new scoreline
          scoreline_i Sline; execute it
          puts      Sline, 1; print it
istart    =         istart + idur; recalculate start for next scoreline
          loop_lt   icnt, 1, itimes, loop; end of the i-time loop
iend      =         istart + idur; calculate the total duration
p3        =         iend; set p3 to the sum of all durations
          print     p3; print it
  endin

  instr 3; plays the notes
asig      pluck     .2, cpspch(p4), cpspch(p4), 0, 1
aenv      transeg   1, p3, 0, 0
          outs      asig*aenv, asig*aenv
  endin

</CsInstruments>
```

```
<CsScore>
i 1 0 1
i 2 14 1
e
</CsScore>
</CsoundSynthesizer>
```

USING TIME LOOPS

As discussed above in the chapter about control structures, a time loop can be built in Csound either with the timout opcode or with the metro opcode. There were also simple examples for triggering instrument events using both methods. Here, a more complex example is given: A master instrument performs a time loop (choose either instr 1 for the timout method or instr 2 for the metro method) and triggers once in a loop a subinstrument. The subinstrument itself (instr 10) performs an i-time loop and triggers several instances of a sub-subinstrument (instr 100). Each instance performs a partial with an independent envelope for a bell-like additive synthesis.

EXAMPLE 03E09.csd

```
<CsoundSynthesizer>
<CsOptions>
-odac
</CsOptions>
<CsInstruments>
;Example by Joachim Heintz
sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

giSine    ftgen     0, 0, 2^10, 10, 1
          seed      0

  instr 1; time loop with timout. events are triggered by event_i (i-rate)
loop:
idurloop  random    1, 4; duration of each loop
          timout    0, idurloop, play
          reinit    loop
play:
idurins   random    1, 5; duration of the triggered instrument
          event_i   "i", 10, 0, idurins; triggers instrument 10
  endin

  instr 2; time loop with metro. events are triggered by event (k-rate)
kfreq     init      1; give a start value for the trigger frequency
kTrig     metro     kfreq
 if kTrig == 1 then ;if trigger impulse:
kdur      random    1, 5; random duration for instr 10
          event     "i", 10, 0, kdur; call instr 10
kfreq     random    .25, 1; set new value for trigger frequency
 endif
  endin

  instr 10; triggers 8-13 partials
inumparts random    8, 14
inumparts =         int(inumparts); 8-13 as integer
ibasoct   random    5, 10; base pitch in octave values
ibasfreq  =         cpsoct(ibasoct)
ipan      random    .2, .8; random panning between left (0) and right (1)
icnt      =         0; counter
loop:
          event_i   "i", 100, 0, p3, ibasfreq, icnt+1, inumparts, ipan
          loop_lt   icnt, 1, inumparts, loop
  endin

  instr 100; plays one partial
ibasfreq  =         p4; base frequency of sound mixture
ipartnum  =         p5; which partial is this (1 - N)
inumparts =         p6; total number of partials
ipan      =         p7; panning
ifreqgen  =         ibasfreq * ipartnum; general frequency of this partial
ifreqdev  random    -10, 10; frequency deviation between -10% and +10%
; -- real frequency regarding deviation
ifreq     =         ifreqgen + (ifreqdev*ifreqgen)/100
ixtratim  random    0, p3; calculate additional time for this partial
p3        =         p3 + ixtratim; new duration of this partial
imaxamp   =         1/inumparts; maximum amplitude
idbdev    random    -6, 0; random deviation in dB for this partial
iamp      =   imaxamp * ampdb(idbdev-ipartnum); higher partials are softer
ipandev   random    -.1, .1; panning deviation
```

```
ipan      =         ipan + ipandev
aEnv      transeg   0, .005, 0, iamp, p3-.005, -10, 0
aSine     poscil    aEnv, ifreq, giSine
aL, aR    pan2      aSine, ipan
          outs      aL, aR
          prints    "ibasfreq = %d, ipartial = %d, ifreq = %d%n",
                     ibasfreq, ipartnum, ifreq
  endin

</CsInstruments>
<CsScore>
i 1 0 300 ;try this, or the next line (or both)
;i 2 0 300
</CsScore>
</CsoundSynthesizer>
```

LINKS AND RELATED OPCODES

Links

A great collection of interactive examples with FLTK widgets by Iain McCurdy can be found here. See particularily the "Realtime Score Generation" section. Recently, the collection has been ported to QuteCsound by René Jopi, and is part of QuteCsound's example menu.

An extended example for calculating score events at i-time can be found in the Re-Generation of Stockhausen's "Studie II" by Joachim Heintz (also included in the QuteCsound Examples menu).

Related Opcodes

event_i / event: Generate an instrument event at i-time (event_i) or at k-time (event). Easy to use, but you cannot send a string to the subinstrument.

scoreline_i / scoreline: Generate an instrument at i-time (scoreline_i) or at k-time (scoreline). Like event_i/event, but you can send to more than one instrument but unlike event_i/event you can send strings. On the other hand, you must usually preformat your scoreline-string using sprintf.

sprintf / sprintfk: Generate a formatted string at i-time (sprintf) or k-time (sprintfk), and store it as a string-variable.

-+max_str_len=10000: Option in the "CsOptions" tag of a .csd file which extend the maximum string length to 9999 characters.

massign: Assigns the incoming MIDI events to a particular instrument. It is also possible to prevent any assigment by this opcode.

cpsmidi / ampmidi: Returns the frequency / velocity of a pressed MIDI key.

release: Returns "1" if the last k-cycle of an instrument has begun.

xtratim: Adds an additional time to the duration (p3) of an instrument.

turnoff / turnoff2: Turns an instrument off; either by the instrument itself (turnoff), or from another instrument and with several options (turnoff2).

-p3 / -p1: A negative duration (p3) turns an instrument on "indefinitely"; a negative instrument number (p1) turns this instrument off. See the examples at the beginning of this chapter.

-L stdin: Option in the "CsOptions" tag of a .csd file which lets you type in realtime score events.

timout: Allows you to perform time loops at i-time with reinitalization passes.

metro: Outputs momentary 1s with a definable (and variable) frequency. Can be used to perform a time loop at k-rate.

follow: Envelope follower.

18. USER DEFINED OPCODES

Opcodes are the core units of everything that Csound does. They are like little machines that do a job, and programming is akin to connecting these little machines to perform a larger job. An opcode usually has something which goes into it: the inputs or arguments, and usually it has something which comes out of it: the output which is stored in one or more variables. Opcodes are written in the programming language C (that is where the name "Csound" comes from). If you want to create a new opcode in Csound, you must write it in C. How to do this is described in the Extending Csound chapter of this manual, and is also described in the relevant chapter of the Canonical Csound Reference Manual.

There is, however, a way of writing your own opcodes in the Csound Language itself. The opcodes which are written in this way, are called User Defined Opcodes or "UDO"s. A UDO behaves in the same way as a standard opcode: it has input arguments, and usually one or more output variables. They run at i-time or at k-time. You use them as part of the Csound Language after you have defined and loaded them.

User Defined Opcodes have many valuable properties. They make your instrument code clearer because they allow you to create abstractions of blocks of code. Once a UDO has been defined it can be recalled and repeated many times within an orchestra, each repetition requiring only a single line of code. UDOs allow you to build up your own library of functions you need and return to frequently in your work. In this way, you build your own Csound dialect within the Csound Language. UDOs also represent a convenient format with which to share your work in Csound with other users.

This chapter explains, initially with a very basic example, how you can build your own UDOs, and what options they offer. Following this, the practice of loading UDOs in your .csd file is shown, followed by some tips in regard to some unique capabilities of UDOs. Before the "Links And Related Opcodes" section at the end, some examples are shown for different User Defined Opcode definitions and applications.

TRANSFORMING CSOUND INSTRUMENT CODE TO A USER DEFINED OPCODE

Writing a User Defined Opcode is actually very easy and straightforward. It mainly means to extract a portion of usual Csound instrument code, and put it in the frame of a UDO. Let's start with the instrument code:

EXAMPLE 03F01.csd

```
<CsoundSynthesizer>
<CsOptions>
-odac
</CsOptions>
<CsInstruments>
;Example by Joachim Heintz
sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

giSine    ftgen     0, 0, 2^10, 10, 1
          seed      0

  instr 1
aDel      init      0; initialize delay signal
iFb       =         .7; feedback multiplier
aSnd      rand      .2; white noise
kdB       randomi   -18, -6, .4; random movement between -18 and -6
aSnd      =         aSnd * ampdb(kdB); applied as dB to noise
kFiltFq   randomi   100, 1000, 1; random movement between 100 and 1000
aFilt     reson    aSnd, kFiltFq, kFiltFq/5; applied as filter center frequency
aFilt     balance   aFilt, aSnd; bring aFilt to the volume of aSnd
aDelTm    randomi   .1, .8, .2; random movement between .1 and .8 as delay time
aDel      vdelayx   aFilt + iFb*aDel, aDelTm, 1, 128; variable delay
kdbFilt   randomi   -12, 0, 1; two random movements between -12 and 0 (dB) ...
kdbDel    randomi   -12, 0, 1; ... for the filtered and the delayed signal
aOut      =         aFilt*ampdb(kdbFilt) + aDel*ampdb(kdbDel); mix it
```

```
          outs      aOut, aOut
  endin

</CsInstruments>
<CsScore>
i 1 0 60
</CsScore>
</CsoundSynthesizer>
```

This is a filtered noise, and its delay, which is fed back again into the delay line at a certain ratio iFb. The filter is moving as kFiltFq randomly between 100 and 1000 Hz. The volume of the filtered noise is moving as kdB randomly between -18 dB and -6 dB. The delay time moves between 0.1 and 0.8 seconds, and then both signals are mixed together.

Basic Example

If this signal processing unit is to be transformed into a User Defined Opcode, the first question is about the extend of the code that will be encapsulated: where the UDO code will begin and end? The first solution could be a radical, and possibly bad, approach: to transform the whole instrument into a UDO.

EXAMPLE 03F02.csd

```
<CsoundSynthesizer>
<CsOptions>
</CsOptions>
<CsInstruments>
;Example by Joachim Heintz
sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

giSine    ftgen     0, 0, 2^10, 10, 1
          seed      0

  opcode FiltFb, 0, 0
aDel      init      0; initialize delay signal
iFb       =         .7; feedback multiplier
aSnd      rand      .2; white noise
kdB       randomi   -18, -6, .4; random movement between -18 and -6
aSnd      =         aSnd * ampdb(kdB); applied as dB to noise
kFiltFq   randomi   100, 1000, 1; random movement between 100 and 1000
aFilt     reson    aSnd, kFiltFq, kFiltFq/5; applied as filter center frequency
aFilt     balance   aFilt, aSnd; bring aFilt to the volume of aSnd
aDelTm    randomi   .1, .8, .2; random movement between .1 and .8 as delay time
aDel      vdelayx   aFilt + iFb*aDel, aDelTm, 1, 128; variable delay
kdbFilt   randomi   -12, 0, 1; two random movements between -12 and 0 (dB) ...
kdbDel    randomi   -12, 0, 1; ... for the filtered and the delayed signal
aOut      =         aFilt*ampdb(kdbFilt) + aDel*ampdb(kdbDel); mix it
          outs      aOut, aOut
  endop

instr 1
          FiltFb
endin

</CsInstruments>
<CsScore>
i 1 0 60
</CsScore>
</CsoundSynthesizer>
```

Before we continue the discussion about the quality of this transormation, we should have a look at the syntax first. The general syntax for a User Defined Opcode is:

```
opcode name, outtypes, intypes
...
endop
```

Here, the **name** of the UDO is **FiltFb**. You are free to use any name, but it is suggested that you begin the name with a capital letter. By doing this, you avoid duplicating the name of most of the pre-existing opcodes (FLTK and STK opcodes begin with capital letters) which normally start with a lower case letter. As we have no input arguments and no output arguments for this first version of FiltFb, both **outtypes** and **intypes** are set to zero. Similar to the instr ... endin block of a normal instrument definition, for a UDO the **opcode** ... **endop** keywords begin and end the UDO definition block. In the instrument, the UDO is called like a normal opcode by using its name, and in the same line the input arguments are listed on the right and the output arguments on the left. In the previous a example, 'FiltFb' has no input and output arguments so it is called by just using its name:

```
instr 1
          FiltFb
endin
```

Now - why is this UDO more or less useless? It achieves nothing, when compared to the original non UDO version, and in fact loses some of the advantages of the instrument defined version. Firstly, it is not advisable to include this line in the UDO:

```
          outs      aOut, aOut
```

This statement writes the audio signal aOut from inside the UDO to the output device. Imagine you want to change the output channels, or you want to add any signal modifier after the opcode. This would be impossible with this statement. So instead of including the 'outs' opcode, we give the FiltFb UDO an audio output:

```
          xout      aOut
```

The xout statement of a UDO definition works like the "outlets" in PD or Max, sending the result(s) of an opcode back to the caller instrument.

Now let us consider the UDO's input arguments, choose which processes should be carried out within the FiltFb unit, and what aspects would offer greater flexibility if controllable from outside the UDO. First, the **aSnd** parameter should not be restricted to a white noise with amplitude 0.2, but should be an input (like a "signal inlet" in PD/Max). This is implemented using the line:

```
aSnd      xin
```

Both the output and the input type must be declared in the first line of the UDO definition, whether they are i-, k- or a-variables. So instead of "opcode FiltFb, 0, 0" the statement has changed now to "opcode FiltFb, a, a", because we have both input and output as a-variable.

The UDO is now much more flexible and logical: it takes any audio input, it performs the filtered delay and feedback processing, and returns the result as another audio signal. In the next example, instrument 1 does exactly the same as before. Instrument 2 has live input instead.

EXAMPLE 03F03.csd

```
<CsoundSynthesizer>
<CsOptions>
</CsOptions>
<CsInstruments>
;Example by Joachim Heintz
sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

giSine    ftgen     0, 0, 2^10, 10, 1
          seed      0

  opcode FiltFb, a, a
aSnd      xin
aDel      init      0; initialize delay signal
iFb       =         .7; feedback multiplier
kdB       randomi   -18, -6, .4; random movement between -18 and -6
aSnd      =         aSnd * ampdb(kdB); applied as dB to noise
kFiltFq   randomi   100, 1000, 1; random movement between 100 and 1000
aFilt     reson    aSnd, kFiltFq, kFiltFq/5; applied as filter center frequency
aFilt     balance   aFilt, aSnd; bring aFilt to the volume of aSnd
aDelTm    randomi   .1, .8, .2; random movement between .1 and .8 as delay time
aDel      vdelayx   aFilt + iFb*aDel, aDelTm, 1, 128; variable delay
kdbFilt   randomi   -12, 0, 1; two random movements between -12 and 0 (dB) ...
kdbDel    randomi   -12, 0, 1; ... for the filtered and the delayed signal
aOut      =         aFilt*ampdb(kdbFilt) + aDel*ampdb(kdbDel); mix it
          xout      aOut
  endop

  instr 1; white noise input
aSnd      rand      .2
aOut      FiltFb    aSnd
          outs      aOut, aOut
  endin

  instr 2; live audio input
aSnd      inch      1; input from channel 1
```

```
aOut      FiltFb    aSnd
          outs      aOut, aOut
  endin

</CsInstruments>
<CsScore>
i 1 0 60 ;change to i 2 for live audio input
</CsScore>
</CsoundSynthesizer>
```

Is There An Optimal Design For A User Defined Opcode?

Is this now the optimal version of the *FiltFb* User Defined Opcode? Obviously there are other parts of the opcode definiton which could be controllable from outside: the feedback multiplier **iFb**, the random movement of the input signal **kdB**, the random movement of the filter frequency **kFiltFq**, and the random movements of the output mix **kdbSnd** and **kdbDel**. Is it better to put them outside of the opcode definition, or is it better to leave them inside?

There is no general answer. It depends on the degree of abstraction you desire or you prefer to relinquish. If you are working on a piece for which all of the parameters settings are already defined as required in the UDO, then control from the caller instrument may not be necessary . The advantage of minimizing the number of input and output arguments is the simplification in using the UDO. The more flexibility you require from your UDO however, the greater the number of input arguments that will be required. Providing more control is better for a later reusability, but may be unnecessarily complicated.

Perhaps it is the best solution to have one abstract definition which performs one task, and to create a derivative - also as UDO - fine tuned for the particular project you are working on. The final example demonstrates the definition of a general and more abstract UDO *FiltFb*, and its various applications: instrument 1 defines the specifications in the instrument itself; instrument 2 uses a second UDO *Opus123_FiltFb* for this purpose; instrument 3 sets the general *FiltFb* in a new context of two varying delay lines with a buzz sound as input signal.

EXAMPLE 03F04.csd

```
<CsoundSynthesizer>
<CsOptions>
-odac
</CsOptions>
<CsInstruments>
;Example by Joachim Heintz
sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

giSine    ftgen     0, 0, 2^10, 10, 1
          seed      0

  opcode FiltFb, aa, akkkia
; -- DELAY AND FEEDBACK OF A BAND FILTERED INPUT SIGNAL --
;input: aSnd = input sound
; kFb = feedback multiplier (0-1)
; kFiltFq: center frequency for the reson band filter (Hz)
; kQ = band width of reson filter as kFiltFq/kQ
; iMaxDel = maximum delay time in seconds
; aDelTm = delay time
;output: aFilt = filtered and balanced aSnd
; aDel = delay and feedback of aFilt

aSnd, kFb, kFiltFq, kQ, iMaxDel, aDelTm xin
aDel      init      0
aFilt     reson     aSnd, kFiltFq, kFiltFq/kQ
aFilt     balance   aFilt, aSnd
aDel      vdelayx   aFilt + kFb*aDel, aDelTm, iMaxDel, 128; variable delay
          xout      aFilt, aDel
  endop

  opcode Opus123_FiltFb, a, a
;;the udo FiltFb here in my opus 123 :)
;input = aSnd
;output = filtered and delayed aSnd in different mixtures
aSnd      xin
kdB       randomi   -18, -6, .4; random movement between -18 and -6
aSnd      =         aSnd * ampdb(kdB); applied as dB to noise
kFiltFq   randomi   100, 1000, 1; random movement between 100 and 1000
iQ        =         5
iFb       =         .7; feedback multiplier
```

```
aDelTm    randomi   .1, .8, .2; random movement between .1 and .8 as delay time
aFilt, aDel FiltFb    aSnd, iFb, kFiltFq, iQ, 1, aDelTm
kdbFilt   randomi   -12, 0, 1; two random movements between -12 and 0 (dB) ...
kdbDel    randomi   -12, 0, 1; ... for the noise and the delay signal
aOut      =         aFilt*ampdb(kdbFilt) + aDel*ampdb(kdbDel); mix it
          xout      aOut
  endop

  instr 1; well known context as instrument
aSnd      rand      .2
kdB       randomi   -18, -6, .4; random movement between -18 and -6
aSnd      =         aSnd * ampdb(kdB); applied as dB to noise
kFiltFq   randomi   100, 1000, 1; random movement between 100 and 1000
iQ        =         5
iFb       =         .7; feedback multiplier
aDelTm    randomi   .1, .8, .2; random movement between .1 and .8 as delay time
aFilt, aDel FiltFb    aSnd, iFb, kFiltFq, iQ, 1, aDelTm
kdbFilt   randomi   -12, 0, 1; two random movements between -12 and 0 (dB) ...
kdbDel    randomi   -12, 0, 1; ... for the noise and the delay signal
aOut      =         aFilt*ampdb(kdbFilt) + aDel*ampdb(kdbDel); mix it
aOut      linen     aOut, .1, p3, 3
          outs      aOut, aOut
  endin

  instr 2; well known context UDO which embeds another UDO
aSnd      rand      .2
aOut      Opus123_FiltFb aSnd
aOut      linen     aOut, .1, p3, 3
          outs      aOut, aOut
  endin

  instr 3; other context: two delay lines with buzz
kFreq     randomh   200, 400, .08; frequency for buzzer
aSnd      buzz      .2, kFreq, 100, giSine; buzzer as aSnd
kFiltFq   randomi   100, 1000, .2; center frequency
aDelTm1   randomi   .1, .8, .2; time for first delay line
aDelTm2   randomi   .1, .8, .2; time for second delay line
kFb1      randomi   .8, 1, .1; feedback for first delay line
kFb2      randomi   .8, 1, .1; feedback for second delay line
a0, aDel1 FiltFb    aSnd, kFb1, kFiltFq, 1, 1, aDelTm1; delay signal 1
a0, aDel2 FiltFb    aSnd, kFb2, kFiltFq, 1, 1, aDelTm2; delay signal 2
aDel1     linen     aDel1, .1, p3, 3
aDel2     linen     aDel2, .1, p3, 3
          outs      aDel1, aDel2
  endin

</CsInstruments>
<CsScore>
i 1 0 30
i 2 31 30
i 3 62 120
</CsScore>
</CsoundSynthesizer>
```

The good thing about the different possibilities of writing a more specified UDO, or a more generalized: You needn't decide this at the beginning of your work. Just start with any formulation you find useful in a certain situation. If you continue and see that you should have some more parameters accessible, it should be easy to rewrite the UDO. Just be careful not to confuse the different versions you create. Use names like Faulty1, Faulty2 etc. instead of overwriting Faulty. Making use of extensive commenting when you initially create the UDO will make it easier to adapt the UDO at a later time. What are the inputs (including the measurement units they use such as Hertz or seconds)? What are the outputs? - How you do this, is up to you and depends on your style and your preference.

HOW TO USE THE USER DEFINED OPCODE FACILITY IN PRACTICE

In this section, we will address the main points of using UDOs: what you must bear in mind when loading them, what special features they offer, what restrictions you must be aware of and how you can build your own language with them.

Loading User Defined Opcodes In The Orchestra Header

As can be seen from the examples above, User Defined Opcodes must be defined in the orchestra header (which is sometimes called "instrument 0"). Note that your opcode definitions must be the **last** part of all your orchestra header statements. The following usage results in an error, even though it is probably fair to regard Csound as intolerant in doing so - this intolerance may be removed in future versions of Csound.

EXAMPLE 03F05.csd

```
<CsInstruments>
;Example by Joachim Heintz
sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

  opcode FiltFb, aa, akkkia
; -- DELAY AND FEEDBACK OF A BAND FILTERED INPUT SIGNAL --
;input: aSnd = input sound
; kFb = feedback multiplier (0-1)
; kFiltFq: center frequency for the reson band filter (Hz)
; kQ = band width of reson filter as kFiltFq/kQ
; iMaxDel = maximum delay time in seconds
; aDelTm = delay time
;output: aFilt = filtered and balanced aSnd
; aDel = delay and feedback of aFilt

aSnd, kFb, kFiltFq, kQ, iMaxDel, aDelTm xin
aDel      init      0
aFilt     reson     aSnd, kFiltFq, kFiltFq/kQ
aFilt     balance   aFilt, aSnd
aDel      vdelayx   aFilt + kFb*aDel, aDelTm, iMaxDel, 128; variable delay
          xout      aFilt, aDel
  endop

giSine    ftgen     0, 0, 2^10, 10, 1
          seed      0

instr 1
...
```

Csound will complain about "misplaced opcodes", which means that the *ftgen* and the *seed* statement must be **before** the opcode definitions.

Loading A Set Of User Defined Opcodes

You can load as many User Defined Opcodes into a Csound orchestra as you wish. As long as they do not depend on each other, their order is arbitrarily. If UDO *Opus123_FiltFb* uses the UDO *FiltFb* for its definition (see the example above), you must first load *FiltFb*, and then *Opus123_FiltFb*. If not, you will get an error like this:

```
orch compiler:
 opcode Opus123_FiltFb a a
error:  no legal opcode, line 25:
aFilt, aDel FiltFb    aSnd, iFb, kFiltFq, iQ, 1, aDelTm
```

Loading By An #include File

Definitions of User Defined Opcodes can also be loaded into a .csd file by an "#include" statement. What you must do is the following:

1. Save your opcode definitions in a plain text file, for instance "MyOpcodes.txt".
2. If this file is in the same directory as your .csd file, you can just call it by the statement:

   ```
   #include "MyOpcodes.txt"
   ```

3. If "MyOpcodes.txt" is in a different directory, you must call it by the full path name, for instance:

   ```
   #include "/Users/me/Documents/Csound/UDO/MyOpcodes.txt"
   ```

As always, make sure that the "#include" statement is the last one in the orchestra header, and that the logical order is accepted if one opcode depends on another.

If you work with User Defined Opcodes a lot, and build up a collection of them, the #include feature allows you easily import several or all of them to your .csd file.

The setksmps Feature

The ksmps assignment in the orchestra header cannot be changed during the performance of a .csd file. But in a User Defined Opcode you have the unique possibility of changing this value by a local assignment. If you use a setksmps statement in your UDO, you can have a locally smaller value for the number of samples per control cycle in the UDO. In the following example, the print statement in the UDO prints ten times compared to one time in the instrument, because ksmps in the UDO is 10 times smaller:

EXAMPLE 03F06.csd

```
<CsoundSynthesizer>
<CsInstruments>
;Example by Joachim Heintz
sr = 44100
ksmps = 44100 ;very high because of printing

  opcode Faster, 0, 0
setksmps 4410 ;local ksmps is 1/10 of global ksmps
printks "UDO print!%n", 0
  endop

  instr 1
printks "Instr print!%n", 0 ;print each control period (once per second)
Faster ;print 10 times per second because of local ksmps
  endin

</CsInstruments>
<CsScore>
i 1 0 2
</CsScore>
</CsoundSynthesizer>
```

Default Arguments

For i-time arguments, you can use a simple feature to set default values:

- "o" (instead of "i") defaults to 0
- "p" (instead of "i") defaults to 1
- "j" (instead of "i") defaults to -1

So you can omit these arguments - in this case the default values will be used. If you give an input argument instead, the default value will be overwritten:

EXAMPLE 03F07.csd

```
<CsoundSynthesizer>
<CsInstruments>
;Example by Joachim Heintz

  opcode Defaults, iii, opj
ia, ib, ic xin
xout ia, ib, ic
  endop

instr 1
ia, ib, ic Defaults
           print     ia, ib, ic
ia, ib, ic Defaults  10
           print     ia, ib, ic
ia, ib, ic Defaults  10, 100
           print     ia, ib, ic
ia, ib, ic Defaults  10, 100, 1000
           print     ia, ib, ic
endin

</CsInstruments>
<CsScore>
i 1 0 0
</CsScore>
</CsoundSynthesizer>
```

Recursive User Defined Opcodes

Recursion means that a function can call itself. This is a feature which can be useful in many situations. Also User Defined Opcodes can be recursive. You can do many things with a recursive UDO which you cannot do in any other way; at least not in a simliarly simple way. This is an example of generating eight partials by a recursive UDO. See the last example in the next section for a more musical application of a recursive UDO.

EXAMPLE 03F08.csd

```
<CsoundSynthesizer>
<CsOptions>
</CsOptions>
<CsInstruments>
;Example by Joachim Heintz
sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

  opcode Recursion, a, iip
;input: frequency, number of partials, first partial (default=1)
ifreq, inparts, istart xin
iamp      =         1/inparts/istart ;decreasing amplitudes for higher partials
 if istart < inparts then ;if inparts have not yet reached
acall     Recursion ifreq, inparts, istart+1 ;call another instance of this UDO
 endif
aout      oscils    iamp, ifreq*istart, 0 ;execute this partial
aout      =         aout + acall ;add the audio signals
          xout      aout
  endop

  instr 1
amix      Recursion 400, 8 ;8 partials with a base frequency of 400 Hz
aout      linen     amix, .01, p3, .1
          outs      aout, aout
  endin

</CsInstruments>
<CsScore>
i 1 0 1
</CsScore>
</CsoundSynthesizer>
```

EXAMPLES

We will focus here on some examples which will hopefully show the wide range of User Defined Opcodes. Some of them are adaptions of examples from previous chapters about the Csound Syntax. Much more examples can be found in the User-Defined Opcode Database, editied by Steven Yi.

Play A Mono Or Stereo Soundfile

Csound is often very strict and gives errors where other applications might 'turn a blind eye'. This is also the case if you read a soundfile using one of Csound's opcodes: soundin, diskin or diskin2. If your soundfile is mono, you must use the mono version, which has one audio signal as output. If your soundfile is stereo, you must use the stereo version, which outputs two audio signals. If you want a stereo output, but you happen to have a mono soundfile as input, you will get the error message:

```
INIT ERROR in ...: number of output args inconsistent with number
of file channels
```

It may be more useful to have an opcode which works for both, mono and stereo files as input. This is a ideal job for a UDO. Two versions are possible: FilePlay1 returns always one audio signal (if the file is stereo it uses just the first channel), FilePlay2 returns always two audio signals (if the file is mono it duplicates this to both channels). We can use the default arguments to make this opcode behave exactly as diskin2:

EXAMPLE 03F09.csd

```
<CsoundSynthesizer>
<CsOptions>
-odac
</CsOptions>
<CsInstruments>
;Example by Joachim Heintz
sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

  opcode FilePlay1, a, Skoooooo
;gives mono output regardless your soundfile is mono or stereo
;(if stereo, just the first channel is used)
;see diskin2 page of the csound manual for information about the input arguments
Sfil, kspeed, iskip, iloop, iformat, iwsize, ibufsize, iskipinit xin
```

```
ichn      filenchnls Sfil
 if ichn == 1 then
aout      diskin2   Sfil, kspeed, iskip, iloop, iformat, iwsize,
                    ibufsize, iskipinit
 else
aout, a0  diskin2   Sfil, kspeed, iskip, iloop, iformat, iwsize,
                    ibufsize, iskipinit
 endif
          xout      aout
  endop

  opcode FilePlay2, aa, Skoooooo
;gives stereo output regardless your soundfile is mono or stereo
;see diskin2 page of the csound manual for information about the input arguments
Sfil, kspeed, iskip, iloop, iformat, iwsize, ibufsize, iskipinit xin
ichn      filenchnls Sfil
 if ichn == 1 then
aL        diskin2    Sfil, kspeed, iskip, iloop, iformat, iwsize,
                     ibufsize, iskipinit
aR        =          aL
 else
aL, aR     diskin2    Sfil, kspeed, iskip, iloop, iformat, iwsize,
                      ibufsize, iskipinit
 endif
          xout       aL, aR
  endop

  instr 1
aMono     FilePlay1  "fox.wav", 1
          outs       aMono, aMono
  endin

  instr 2
aL, aR    FilePlay2  "fox.wav", 1
          outs       aL, aR
  endin

</CsInstruments>
<CsScore>
i 1 0 4
i 2 4 4
</CsScore>
</CsoundSynthesizer>
```

Change The Content Of A Function Table

In example *03C11.csd*, a function table has been changed at performance time, once a second, by random deviations. This can be easily transformed to a User Defined Opcode. It takes the function table variable, a trigger signal, and the random deviation in percent as input. In each control cycle where the trigger signal is "1", the table values are read. The random deviation is applied, and the changed values are written again into the table. Here, the tab/tabw opcodes are used to make sure that also non-power-of-two tables can be used.

EXAMPLE 03F10.csd

```
<CsoundSynthesizer>
<CsOptions>
-odac
</CsOptions>
<CsInstruments>
;Example by Joachim Heintz
sr = 44100
ksmps = 441
nchnls = 2
0dbfs = 1

giSine    ftgen     0, 0, 256, 10, 1; sine wave
          seed      0; each time different seed

  opcode TabDirtk, 0, ikk
;"dirties" a function table by applying random deviations at a k-rate trigger
;input: function table, trigger (1 = perform manipulation),
;deviation as percentage
ift, ktrig, kperc xin
 if ktrig == 1 then ;just work if you get a trigger signal
kndx      =         0
loop:
krand     random    -kperc/100, kperc/100
kval      tab       kndx, ift; read old value
knewval   =         kval + (kval * krand); calculate new value
          tabw      knewval, kndx, giSine; write new value
          loop_lt   kndx, 1, ftlen(ift), loop; loop construction
 endif
  endop

  instr 1
kTrig     metro     1, .00001 ;trigger signal once per second
          TabDirtk  giSine, kTrig, 10
aSig      poscil    .2, 400, giSine
          outs      aSig, aSig
  endin

</CsInstruments>
<CsScore>
i 1 0 10
</CsScore>
</CsoundSynthesizer>
```

Of course you can also change the content of a function table at init-time. The next example permutes a series of numbers randomly each time it is called. For this purpose, first the input function table **iTabin** is copied as **iCopy**. This is necessary because we do not want to change iTabin in any way. Next a random index in iCopy is created and the value at this location in iTabin is written at the beginning of **iTabout**, which contains the permuted results. At the end of this cycle, each value in iCopy which has a larger index than the one which has just been read, is shifted one position to the left. So now iCopy has become one position smaller - not in table size but in the number of values to read. This procedure is continued until all values from iCopy are reflected in iTabout:

EXAMPLE 03F11.csd

```
<CsoundSynthesizer>
<CsInstruments>
;Example by Joachim Heintz

giVals    ftgen     0, 0, -12, -2, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12
          seed      0; each time different seed

  opcode TabPermRand_i, i, i
;permuts randomly the values of the input table
;and creates an output table for the result
iTabin    xin
itablen   =         ftlen(iTabin)
iTabout   ftgen     0, 0, -itablen, 2, 0 ;create empty output table
iCopy     ftgen     0, 0, -itablen, 2, 0 ;create empty copy of input table
          tableicopy iCopy, iTabin ;write values of iTabin into iCopy
icplen    init      itablen ;number of values in iCopy
indxwt    init      0 ;index of writing in iTabout
loop:
indxrd    random    0, icplen - .0001; random read index in iCopy
indxrd    =         int(indxrd)
ival      tab_i     indxrd, iCopy; read the value
          tabw_i    ival, indxwt, iTabout; write it to iTabout
; -- shift values in iCopy larger than indxrd one position to the left
 shift:
 if indxrd < icplen-1 then ;if indxrd has not been the last table value
ivalshft  tab_i     indxrd+1, iCopy ;take the value to the right ...
          tabw_i    ivalshft, indxrd, iCopy ;...and write it to indxrd position
indxrd    =         indxrd + 1 ;then go to the next position
          igoto     shift ;return to shift and see if there is anything left to
do
 endif
indxwt    =         indxwt + 1 ;increase the index of writing in iTabout
          loop_gt   icplen, 1, 0, loop ;loop as long as there is ;
                                       ;a value in iCopy
          ftfree    iCopy, 0 ;delete the copy table
          xout      iTabout ;return the number of iTabout
  endop

instr 1
iPerm     TabPermRand_i giVals ;perform permutation
;print the result
indx      =         0
Sres      =         "Result:"
print:
ival      tab_i     indx, iPerm
Sprint    sprintf   "%s %d", Sres, ival
Sres      =         Sprint
          loop_lt   indx, 1, 12, print
          puts      Sres, 1
endin

instr 2; the same but performed ten times
icnt      =         0
loop:
```

```
iPerm     TabPermRand_i giVals ;perform permutation
;print the result
indx      =         0
Sres      =         "Result:"
print:
ival      tab_i     indx, iPerm
Sprint    sprintf   "%s %d", Sres, ival
Sres      =         Sprint
          loop_lt   indx, 1, 12, print
          puts      Sres, 1
          loop_lt   icnt, 1, 10, loop
endin

</CsInstruments>
<CsScore>
i 1 0 0
i 2 0 0
</CsScore>
</CsoundSynthesizer>
```

Print The Content Of A Function Table

There is no opcode in Csound for printing the contents of a function table, but one can be created as a UDO. Again a loop is needed for checking the values and putting them into a string which can then be printed. In addition, some options can be given for the print precision and for the number of elements in a line.

EXAMPLE 03F12.csd

```
<CsoundSynthesizer>
<CsOptions>
-ndm0 -+max_str_len=10000
</CsOptions>
<CsInstruments>
;Example by Joachim Heintz

gitab     ftgen     1, 0, -7, -2, 0, 1, 2, 3, 4, 5, 6
gisin     ftgen     2, 0, 128, 10, 1


  opcode TableDumpSimp, 0, ijo
;prints the content of a table in a simple way
;input: function table, float precision while printing (default = 3),
;parameters per row (default = 10, maximum = 32)
ifn, iprec, ippr xin
iprec     =         (iprec == -1 ? 3 : iprec)
ippr      =         (ippr == 0 ? 10 : ippr)
iend      =         ftlen(ifn)
indx      =         0
Sformat   sprintf   "%%.%df\t", iprec
Sdump     =         ""
loop:
ival      tab_i     indx, ifn
Snew      sprintf   Sformat, ival
Sdump     strcat    Sdump, Snew
indx      =         indx + 1
imod      =         indx % ippr
 if imod == 0 then
          puts      Sdump, 1
Sdump     =         ""
 endif
 if indx < iend igoto loop
          puts      Sdump, 1
  endop


instr 1
          TableDumpSimp p4, p5, p6
          prints    "%n"
endin

</CsInstruments>
<CsScore>
;i1   st   dur   ftab   prec   ppr
i1    0    0     1      -1
i1    .    .     1       0
i1    .    .     2       3     10
i1    .    .     2       6     32
</CsScore>
</CsoundSynthesizer>
```

A Recursive User Defined Opcode For Additive Synthesis

In the last example of the chapter about Triggering Instrument Events a number of partials were synthesized, each with a random frequency deviation of up to 10% compared to precise harmonic spectrum frequencies and a unique duration for each partial. This can also be written as a recursive UDO. Each UDO generates one partial, and calls the UDO again until the last partial is generated. Now the code can be reduced to two instruments: instrument 1 performs the time loop, calculates the basic values for one note, and triggers the event. Then instrument 11 is called which feeds the UDO with the values and passes the audio signals to the output.

EXAMPLE 03F13.csd

```
<CsoundSynthesizer>
<CsOptions>
-odac
</CsOptions>
<CsInstruments>
;Example by Joachim Heintz
sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

giSine    ftgen     0, 0, 2^10, 10, 1
          seed      0

  opcode PlayPartials, aa, iiipo
;plays inumparts partials with frequency deviation and own envelopes and
;durations for each partial
ibasfreq  \ ; base frequency of sound mixture
inumparts \ ; total number of partials
ipan      \ ; panning
ipartnum  \ ; which partial is this (1 - N, default=1)
ixtratim  \ ; extra time in addition to p3 needed for this partial (default=0)
          xin
ifreqgen  =         ibasfreq * ipartnum; general frequency of this partial
ifreqdev  random    -10, 10; frequency deviation between -10% and +10%
ifreq     =         ifreqgen + (ifreqdev*ifreqgen)/100; real frequency
ixtratim1 random    0, p3; calculate additional time for this partial
imaxamp   =         1/inumparts; maximum amplitude
idbdev    random    -6, 0; random deviation in dB for this partial
iamp      =        imaxamp * ampdb(idbdev-ipartnum); higher partials are softer
ipandev   random    -.1, .1; panning deviation
ipan      =         ipan + ipandev
aEnv      transeg   0, .005, 0, iamp, p3+ixtratim1-.005, -10, 0; envelope
aSine     poscil    aEnv, ifreq, giSine
aL1, aR1  pan2      aSine, ipan
 if ixtratim1 > ixtratim then
ixtratim  =  ixtratim1 ;set ixtratim to the ixtratim1 if the latter is larger
 endif
 if ipartnum < inumparts then ;if this is not the last partial
; -- call the next one
aL2, aR2  PlayPartials ibasfreq, inumparts, ipan, ipartnum+1, ixtratim
 else               ;if this is the last partial
p3        =         p3 + ixtratim; reset p3 to the longest ixtratim value
 endif
          xout      aL1+aL2, aR1+aR2
  endop

  instr 1; time loop with metro
kfreq     init      1; give a start value for the trigger frequency
kTrig     metro     kfreq
 if kTrig == 1 then ;if trigger impulse:
kdur      random    1, 5; random duration for instr 10
knumparts random    8, 14
knumparts =         int(knumparts); 8-13 partials
kbasoct   random    5, 10; base pitch in octave values
kbasfreq  =         cpsoct(kbasoct) ;base frequency
kpan      random    .2, .8; random panning between left (0) and right (1)
          event     "i", 11, 0, kdur, kbasfreq, knumparts, kpan; call instr 11
kfreq     random    .25, 1; set new value for trigger frequency
 endif
  endin

  instr 11; plays one mixture with 8-13 partials
aL, aR    PlayPartials p4, p5, p6
          outs      aL, aR
  endin

</CsInstruments>
<CsScore>
i 1 0 300
</CsScore>
</CsoundSynthesizer>
```

Using Strings as Arrays

For some situations it can be very useful to use strings in Csound as a collection of single strings or numbers. This is what programming languages call a list or an array. Csound does not provide opcodes for this purpose, but you can define these opcodes as UDOs. A set of these UDOs can then be used like this:

```
ilen       StrayLen     "a b c d e"
 ilen -> 5
Sel        StrayGetEl   "a b c d e", 0
 Sel -> "a"
inum       StrayGetNum  "1 2 3 4 5", 0
 inum -> 1
ipos       StrayElMem   "a b c d e", "c"
 ipos -> 2
ipos       StrayNumMem  "1 2 3 4 5", 3
 ipos -> 2
Sres       StraySetEl   "a b c d e", "go", 0
 Sres -> "go a b c d e"
Sres       StraySetNum  "1 2 3 4 5", 0, 0
 Sres -> "0 1 2 3 4 5"
Srev       StrayRev     "a b c d e"
 Srev -> "e d c b a"
Sub        StraySub     "a b c d e", 1, 3
 Sub -> "b c"
Sout       StrayRmv     "a b c d e", "b d"
 Sout -> "a c e"
Srem       StrayRemDup  "a b a c c d e e"
 Srem -> "a b c d e"
ift,iftlen StrayNumToFt "1 2 3 4 5", 1
 ift -> 1 (same as f 1 0 -5 -2 1 2 3 4 5)
 iftlen -> 5
```

You can find an article about defining such a sub-language here, and the up to date UDO code here (or at the UDO repository).

LINKS AND RELATED OPCODES

Links

This is the page in the Canonical Csound Reference Manual about the definition of UDOs.

The most important resource of User Defined Opcodes is the User-Defined Opcode Database, editied by Steven Yi.

Also by Steven Yi, read the second part of his article about control flow in Csound in the Csound Journal (summer 2006).

Related Opcodes

opcode: The opcode used to begin a User Defined Opcode definition.

#include: Useful to include any loadable Csound code, in this case definitions of User Defined Opcodes.

setksmps: Lets you set a smaller ksmps value locally in a User Defined Opcode.

19. MACROS

Macros within Csound is a mechanism whereby a line or a block of text can be referenced using a macro codeword. Whenever the codeword is subsequently encountered in a Csound orchestra or score it will be replaced by the code text contained within the macro. This mechanism can be useful in situations where a line or a block of code will be repeated many times - if a change is required in the code that will be repeated, it need only be altered once in the macro definition rather than having to be edited in each of the repetitions.

Csound utilises a subtly different mechanism for orchestra and score macros so each will be considered in turn. There are also additional features offered by the macro system such as the ability to create a macro that accepts arguments - a little like the main macro containing sub-macros that can be repeated several times within the main macro - the inclusion of a block of text contained within a completely separate file and other macro refinements.

It is important to realise that a macro can contain any text, including carriage returns, and that Csound will be ignorant to its use of syntax until the macro is actually used and expanded elsewhere in the orchestra or score.

ORCHESTRA MACROS

Macros are defined using the syntax:

```
#define NAME # replacement text #
```

'NAME' is the user-defined name that will be used to call the macro at some point later in the orchestra; it must begin with a letter but can then contain any combination of numbers and letters. 'replacement text', bounded by hash symbols will be the text that will replace the macro name when later called. Remember that the replacement text can stretch over several lines. One syntactical aspect to note is that '#define' needs to be right at the beginning of a line, i.e. the Csound parser will be intolerant toward the initial '#' being preceded by any white space, whether that be spaces or tabs. A macro can be defined anywhere within the <CsInstruments> </CsInstruments> sections of a .csd file.

When it is desired to use and expand the macro later in the orchestra the macro name needs to be preceded with a '$' symbol thus:

```
  $NAME
```

The following example illustrates the basic syntax needed to employ macros. The name of a sound file is referenced twice in the score so it is defined as a macro just after the header statements. Instrument 1 derives the duration of the sound file and instructs instrument 2 to play a note for this duration. instrument 2 plays the sound file. The score as defined in the <CsScore> </CsScore> section only lasts for 0.01 seconds but the event_i statement in instrument 1 will extend this for the required duration. The sound file is a mono file so you can replace it with any other mono file or use the original one.

EXAMPLE 03G01.csd

```
<CsoundSynthesizer>

<CsOptions>
-odac
</CsOptions>

<CsInstruments>
sr  =  44100
ksmps  =  16
nchnls  =  1
0dbfs = 1

; define the macro
#define SOUNDFILE # "loop.wav" #

 instr  1
; use an expansion of the macro in deriving the duration of the sound file
idur  filelen   $SOUNDFILE
```

```
      event_i   "i",2,0,idur
 endin

 instr  2
; use another expansion of the macro in playing the sound file
a1  diskin2  $SOUNDFILE,1
    out      a1
 endin

</CsInstruments>

<CsScore>
i 1 0 0.01
e
</CsScore>
</CsoundSynthesizer>
; example written by Iain McCurdy
```

In more complex situations where we require slight variations, such as different constant values or different sound files in each reuse of the macro, we can use a macro with arguments. A macro's argument are defined as a list of sub-macro names within brackets after the name of the primary macro and each macro argument is separated by an apostrophe as shown below.

```
#define NAME(Arg1'Arg2'Arg3...) # replacement text #
```

Arguments can be any text string permitted as Csound code, they should not be likened to opcode arguments where each must conform to a certain type such as i, k, a etc. Macro arguments are subsequently referenced in the macro text using their names preceded by a '$' symbol. When the main macro is called later in the orchestra its arguments are then replaced with the values or strings required. The Csound Reference Manual states that up to five arguments are permitted but this still refers to an earlier implementation and in fact many more are actually permitted.

In the following example a 6 partial additive synthesis engine with a percussive character is defined within a macro. Its fundamental frequency and the ratios of its six partials to this fundamental frequency are prescribed as macro arguments. The macro is reused within the orchestra twice to create two different timbres, it could be reused many more times however. The fundamental frequency argument is passed to the macro as p4 from the score.

EXAMPLE 03G02.csd

```
<CsoundSynthesizer>

<CsOptions>
-odac
</CsOptions>

<CsInstruments>
sr  =  44100
ksmps  =  16
nchnls  =  1
0dbfs = 1

gisine  ftgen  0,0,2^10,10,1

; define the macro
#define ADDITIVE_TONE(Frq'Ratio1'Ratio2'Ratio3'Ratio4'Ratio5'Ratio6) #
iamp =      0.1
aenv expseg  1,p3*(1/$Ratio1),0.001,1,0.001
a1  poscil  iamp*aenv,$Frq*$Ratio1,gisine
aenv expseg  1,p3*(1/$Ratio2),0.001,1,0.001
a2  poscil  iamp*aenv,$Frq*$Ratio2,gisine
aenv expseg  1,p3*(1/$Ratio3),0.001,1,0.001
a3  poscil  iamp*aenv,$Frq*$Ratio3,gisine
aenv expseg  1,p3*(1/$Ratio4),0.001,1,0.001
a4  poscil  iamp*aenv,$Frq*$Ratio4,gisine
aenv expseg  1,p3*(1/$Ratio5),0.001,1,0.001
a5  poscil  iamp*aenv,$Frq*$Ratio5,gisine
aenv expseg  1,p3*(1/$Ratio6),0.001,1,0.001
a6  poscil  iamp*aenv,$Frq*$Ratio6,gisine
a7  sum     a1,a2,a3,a4,a5,a6
    out     a7
#

 instr  1 ; xylophone
; expand the macro with partial ratios that reflect those of a xylophone
; the fundemental frequency macro argument (the first argument -
; - is passed as p4 from the score
$ADDITIVE_TONE(p4'1'3.932'9.538'16.688'24.566'31.147)
```

```
 endin

 instr  2 ; vibraphone
$ADDITIVE_TONE(p4'1'3.997'9.469'15.566'20.863'29.440)
 endin

</CsInstruments>

<CsScore>
i 1 0  1 200
i 1 1  2 150
i 1 2  4 100
i 2 3  7 800
i 2 4  4 700
i 2 5  7 600
e
</CsScore>
</CsoundSynthesizer>
; example written by Iain McCurdy
```

SCORE MACROS

Score macros employ a similar syntax. Macros in the score can be used in situations where a long string of p-fields are likely to be repeated or, as in the next example, to define a palette of score patterns than repeat but with some variation such as transposition. In this example two 'riffs' are defined which each employ two macro arguments: the first to define when the riff will begin and the second to define a transposition factor in semitones. These riffs are played back using a bass guitar-like instrument using the wgpluck2 opcode. Remember that mathematical expressions within the Csound score must be bound within square brackets [].

EXAMPLE 03G02.csd

```
<CsoundSynthesizer>

<CsOptions>
-odac
</CsOptions>

<CsInstruments>
sr  =  44100
ksmps  =  16
nchnls  =  1
0dbfs = 1


 instr  1 ; bass guitar
a1   wgpluck2 0.98, 0.4, cpsmidinn(p4), 0.1, 0.6
aenv linseg   1,p3-0.1,1,0.1,0
 out a1*aenv
 endin

</CsInstruments>

<CsScore>
; p4 = pitch as a midi note number
#define RIFF_1(Start'Trans)
#
i 1 [$Start     ]  1     [36+$Trans]
i 1 [$Start+1   ]  0.25  [43+$Trans]
i 1 [$Start+1.25]  0.25  [43+$Trans]
i 1 [$Start+1.75]  0.25  [41+$Trans]
i 1 [$Start+2.5 ]  1     [46+$Trans]
i 1 [$Start+3.25]  1     [48+$Trans]
#
#define RIFF_2(Start'Trans)
#
i 1 [$Start     ]  1     [34+$Trans]
i 1 [$Start+1.25]  0.25  [41+$Trans]
i 1 [$Start+1.5 ]  0.25  [43+$Trans]
i 1 [$Start+1.75]  0.25  [46+$Trans]
i 1 [$Start+2.25]  0.25  [43+$Trans]
i 1 [$Start+2.75]  0.25  [41+$Trans]
i 1 [$Start+3   ]  0.5   [43+$Trans]
i 1 [$Start+3.5 ]  0.25  [46+$Trans]
#
t 0 90
$RIFF_1(0 ' 0)
$RIFF_1(4 ' 0)
$RIFF_2(8 ' 0)
$RIFF_2(12'-5)
```

```
$RIFF_1(16'-5)
$RIFF_2(20'-7)
$RIFF_2(24' 0)
$RIFF_2(28' 5)
e
</CsScore>
</CsoundSynthesizer>
; example written by Iain McCurdy
```

Score macros can themselves contain macros so that, for example, the above example could be further expanded so that a verse, chorus structure could be employed where verses and choruses, defined using macros, were themselves constructed from a series of riff macros.

UDOs and macros can both be used to reduce code repetition and there are many situations where either could be used but each offers its own strengths. UDOs strengths lies in their ability to be used just like an opcode with inputs and output, the ease with which they can be shared - between Csound projects and between Csound users - their ability to operate at a different k-rate to the rest of the orchestra and in how they facilitate recursion. The fact that macro arguments are merely blocks of text, however, offers up new possibilities and unlike UDOs, macros can span several instruments. Of course UDOs have no use in the Csound score unlike macros. Macros can also be used to simplify the creation of complex FLTK GUI where panel sections might be repeated with variations of output variable names and location.

Csound's orchestra and score macro system offers many additional refinements and this chapter serves merely as an introduction to their basic use. To learn more it is recommended to refer to the relevant sections of the Csound Reference Manual.

04 SOUND SYNTHESIS

20. ADDITIVE SYNTHESIS
21. SUBTRACTIVE SYNTHESIS
22. AMPLITUDE AND RING MODULATION
23. FREQUENCY MODULATION
24. WAVESHAPING
25. GRANULAR SYNTHESIS
26. PHYSICAL MODELLING

20. ADDITIVE SYNTHESIS

Jean Baptiste Joseph Fourier demonstrated around 1800 that any continuous function can be perfectly described as a sum of sine waves. This in fact means that you can create any sound, no matter how complex, if you know which sine waves to add together.

This concept really excited the early pioneers of electronic music, who imagined that sine waves would give them the power to create any sound imaginable and previously unimagined. Unfortunately, they soon realized that while adding sine waves is easy, interesting sounds must have a large number of sine waves which are constantly varying in frequency and amplitude, which turns out to be a hugely impractical task.

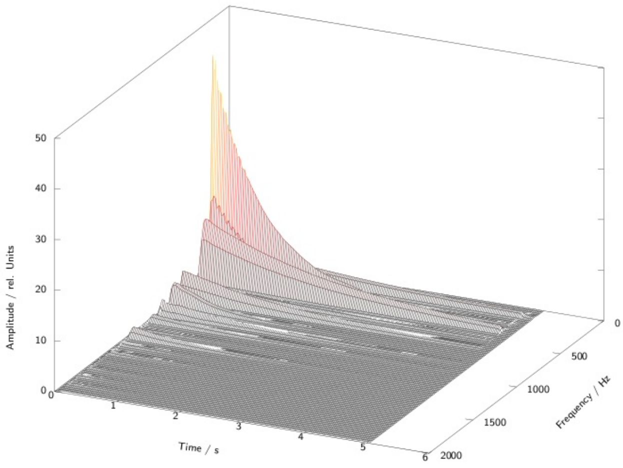
However, additive synthesis can provide unusual and interesting sounds. Moreover both, the power of modern computers, and the ability of managing data in a programming language offer new dimensions of working with this old tool. As with most things in Csound there are several ways to go about it. We will try to show some of them, and see how they are connected with different programming paradigms.

WHAT ARE THE MAIN PARAMETERS OF ADDITIVE SYNTHESIS?

Before going into different ways of implementing additive synthesis in Csound, we shall think about the parameters to consider. As additive synthesis is the addition of several sine generators, the parameters are on two different levels:

- **For each sine**, there is a frequency and an amplitude with an envelope.
  - The **frequency** is usually a constant value. But it can be varied, though. Natural sounds usually have very slight changes of partial frequencies.
  - The **amplitude** must at least have a simple envelope like the well-known ADSR. But more complex ways of continuously altering the amplitude will make the sound much more lively.
- **For the sound as a whole**, these are the relevant parameters:
  - The total **number of sinusoids**. A sound which consists of just three sinusoids is of course "poorer" than a sound which consists of 100 sinusoids.
  - The **frequency ratios** of the sine generators. For a classical harmonic spectrum, the multipliers of the sinusoids are 1, 2, 3, ... (If your first sine is 100 Hz, the others are 200, 300, 400, ... Hz.) For an inharmonic or noisy spectrum, there are probably no simple integer ratios. This frequency ratio is mainly responsible for our perception of timbre.
  - The **base frequency** is the frequency of the first partial. If the partials are showing an harmonic ratio, this frequency (in the example given 100 Hz) is also the overall perceived pitch.
  - The **amplitude ratios** of the sinusoids. This is also very important for the resulting timbre of a sound. If the higher partials are relatively strong, the sound appears more brilliant; if the higher partials are soft, the sound appears dark and soft.
  - The **duration ratios** of the sinusoids. In simple additive synthesis, all single sines have the same duration, but they may also differ. This usually relates to the envelopes: if the envelopes of different partials vary, some partials may die away faster than others.

It is not always the aim of additive synthesis to imitate natural sounds, but it can definitely be learned a lot through the task of first analyzing and then attempting to imitate a sound using additive synthesis techniques. This is what a guitar note looks like when spectrally analyzed:



Spectral analysis of a guitar tone in time (courtesy of W. Fohl, Hamburg)

Each partial has its own movement and duration. We may or may not be able to achieve this successfully in additive synthesis. Let us begin with some simple sounds and consider ways of programming this with Csound; later we will look at some more complex sounds and advanced ways of programming this.

SIMPLE ADDITIONS OF SINUSOIDS INSIDE AN INSTRUMENT

If additive synthesis amounts to the adding sine generators, it is straightforward to create multiple oscillators in a single instrument and to add the resulting audio signals together. In the following example, instrument 1 shows a harmonic spectrum, and instrument 2 an inharmonic one. Both instruments share the same amplitude multipliers: 1, 1/2, 1/3, 1/4, ... and receive the base frequency in Csound's pitch notation (octave.semitone) and the main amplitude in dB.

EXAMPLE 04A01.csd

```
<CsoundSynthesizer>
<CsOptions>
-o dac
</CsOptions>
<CsInstruments>
;example by Andrés Cabrera
sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

giSine    ftgen     0, 0, 2^10, 10, 1

    instr 1 ;harmonic additive synthesis
;receive general pitch and volume from the score
ibasefrq  =         cpspch(p4) ;convert pitch values to frequency
ibaseamp  =         ampdbfs(p5) ;convert dB to amplitude
;create 8 harmonic partials
aOsc1     poscil    ibaseamp, ibasefrq, giSine
aOsc2     poscil    ibaseamp/2, ibasefrq*2, giSine
aOsc3     poscil    ibaseamp/3, ibasefrq*3, giSine
aOsc4     poscil    ibaseamp/4, ibasefrq*4, giSine
aOsc5     poscil    ibaseamp/5, ibasefrq*5, giSine
aOsc6     poscil    ibaseamp/6, ibasefrq*6, giSine
aOsc7     poscil    ibaseamp/7, ibasefrq*7, giSine
aOsc8     poscil    ibaseamp/8, ibasefrq*8, giSine
;apply simple envelope
kenv      linen     1, p3/4, p3, p3/4
;add partials and write to output
aOut = aOsc1 + aOsc2 + aOsc3 + aOsc4 + aOsc5 + aOsc6 + aOsc7 + aOsc8
          outs      aOut*kenv, aOut*kenv
    endin

    instr 2 ;inharmonic additive synthesis
```

```
ibasefrq  =         cpspch(p4)
ibaseamp  =         ampdbfs(p5)
;create 8 inharmonic partials
aOsc1     poscil    ibaseamp, ibasefrq, giSine
aOsc2     poscil    ibaseamp/2, ibasefrq*1.02, giSine
aOsc3     poscil    ibaseamp/3, ibasefrq*1.1, giSine
aOsc4     poscil    ibaseamp/4, ibasefrq*1.23, giSine
aOsc5     poscil    ibaseamp/5, ibasefrq*1.26, giSine
aOsc6     poscil    ibaseamp/6, ibasefrq*1.31, giSine
aOsc7     poscil    ibaseamp/7, ibasefrq*1.39, giSine
aOsc8     poscil    ibaseamp/8, ibasefrq*1.41, giSine
kenv      linen     1, p3/4, p3, p3/4
aOut = aOsc1 + aOsc2 + aOsc3 + aOsc4 + aOsc5 + aOsc6 + aOsc7 + aOsc8
          outs aOut*kenv, aOut*kenv
    endin

</CsInstruments>
<CsScore>
;          pch       amp
i 1 0 5    8.00      -10
i 1 3 5    9.00      -14
i 1 5 8    9.02      -12
i 1 6 9    7.01      -12
i 1 7 10   6.00      -10
s
i 2 0 5    8.00      -10
i 2 3 5    9.00      -14
i 2 5 8    9.02      -12
i 2 6 9    7.01      -12
i 2 7 10   6.00      -10
</CsScore>
</CsoundSynthesizer>
```

SIMPLE ADDITIONS OF SINUSOIDS VIA THE SCORE

A typical paradigm in programming: If you find some almost identical lines in your code, consider to abstract it. For the Csound Language this can mean, to move parameter control to the score. In our case, the lines

```
aOsc1     poscil    ibaseamp, ibasefrq, giSine
aOsc2     poscil    ibaseamp/2, ibasefrq*2, giSine
aOsc3     poscil    ibaseamp/3, ibasefrq*3, giSine
aOsc4     poscil    ibaseamp/4, ibasefrq*4, giSine
aOsc5     poscil    ibaseamp/5, ibasefrq*5, giSine
aOsc6     poscil    ibaseamp/6, ibasefrq*6, giSine
aOsc7     poscil    ibaseamp/7, ibasefrq*7, giSine
aOsc8     poscil    ibaseamp/8, ibasefrq*8, giSine
```

can be abstracted to the form

```
aOsc     poscil    ibaseamp*iampfactor, ibasefrq*ifreqfactor, giSine
```

with the parameters *iampfactor* (the relative amplitude of a partial) and *ifreqfactor* (the frequency multiplier) transferred to the score.

The next version simplifies the instrument code and defines the variable values as score parameters:

EXAMPLE 04A02.csd

```
<CsoundSynthesizer>
<CsOptions>
-o dac
</CsOptions>
<CsInstruments>
;example by Andrés Cabrera and Joachim Heintz
sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

giSine    ftgen     0, 0, 2^10, 10, 1

    instr 1
iBaseFreq =         cpspch(p4)
iFreqMult =         p5 ;frequency multiplier
iBaseAmp  =         ampdbfs(p6)
iAmpMult  =         p7 ;amplitude multiplier
iFreq     =         iBaseFreq * iFreqMult
iAmp      =         iBaseAmp * iAmpMult
kEnv      linen     iAmp, p3/4, p3, p3/4
```

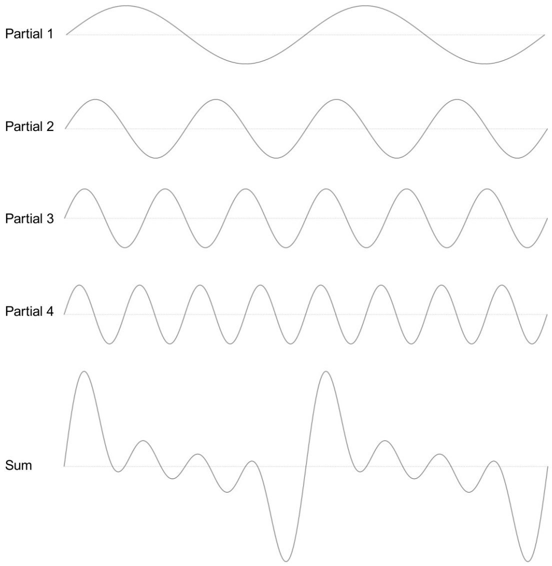
```
aOsc      poscil    kEnv, iFreq, giSine
          outs      aOsc, aOsc
    endin

</CsInstruments>
<CsScore>
;          freq      freqmult  amp       ampmult
i 1 0 7    8.09      1         -10       1
i . . 6    .         2         .         [1/2]
i . . 5    .         3         .         [1/3]
i . . 4    .         4         .         [1/4]
i . . 3    .         5         .         [1/5]
i . . 3    .         6         .         [1/6]
i . . 3    .         7         .         [1/7]
s
i 1 0 6    8.09      1.5       -10       1
i . . 4    .         3.1       .         [1/3]
i . . 3    .         3.4       .         [1/6]
i . . 4    .         4.2       .         [1/9]
i . . 5    .         6.1       .         [1/12]
i . . 6    .         6.3       .         [1/15]
</CsScore>
</CsoundSynthesizer>
```

You might say: Okay, where is the simplification? There are even more lines than before! - This is true, and this is certainly just a step on the way to a better code. The main benefit now is *flexibility*. Now our code is capable of realizing any number of partials, with any amplitude, frequency and duration ratios. Using the Csound score abbreviations (for instance a dot for repeating the previous value in the same p-field), you can do a lot of copy-and-paste, and focus on what is changing from line to line.

Note also that you are now calling **one instrument in multiple instances** at the same time for performing additive synthesis. In fact, each instance of the instrument contributes just one partial for the additive synthesis. This call of multiple and simultaneous instances of one instrument is also a typical procedure for situations like this, and for writing clean and effective Csound code. We will discuss later how this can be done in a more elegant way than in the last example.

CREATING FUNCTION TABLES FOR ADDITIVE SYNTHESIS

Before we continue on this road, let us go back to the first example and discuss a classical and abbreviated method of playing a number of partials. As we mentioned at the beginning, Fourier stated that any periodic oscillation can be described as a sum of simple sinusoids. If the single sinusoids are static (no individual envelope or duration), the resulting waveform will always be the same.



You see four sine generators, each with fixed frequency and amplitude relations, and mixed together. At the bottom of the illustration you see the composite waveform which repeats itself at each period. So - why not just calculate this composite waveform first, and then read it with just one oscillator?

This is what some Csound GEN routines do. They compose the resulting shape of the periodic wave, and store the values in a function table. GEN10 can be used for creating a waveform consisting of harmonically related partials. After the common GEN routine p-fields

```
<table number>, <creation time>, <size in points>, <GEN number>
```

you have just to determine the relative strength of the harmonics. GEN09 is more complex and allows you to also control the frequency multiplier and the phase (0-360°) of each partial. We are able to reproduce the first example in a shorter (and computational faster) form:

EXAMPLE 04A03.csd

```
<CsoundSynthesizer>
<CsOptions>
-o dac
</CsOptions>
<CsInstruments>
;example by Andrés Cabrera and Joachim Heintz
sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

giSine    ftgen     0, 0, 2^10, 10, 1
giHarm    ftgen     1, 0, 2^12, 10, 1, 1/2, 1/3, 1/4, 1/5, 1/6, 1/7, 1/8
giNois    ftgen     2, 0, 2^12, 9, 100,1,0,  102,1/2,0,  110,1/3,0,
                 123,1/4,0,  126,1/5,0,  131,1/6,0,  139,1/7,0,  141,1/8,0

    instr 1
iBasFreq  =         cpspch(p4)
iTabFreq  =         p7 ;base frequency of the table
iBasFreq  =         iBasFreq / iTabFreq
iBaseAmp  =         ampdb(p5)
iFtNum    =         p6
aOsc      poscil    iBaseAmp, iBasFreq, iFtNum
aEnv      linen     aOsc, p3/4, p3, p3/4
          outs      aEnv, aEnv
    endin

</CsInstruments>
<CsScore>
;          pch       amp       table      table base (Hz)
i 1 0 5    8.00      -10       1          1
i . 3 5    9.00      -14       .          .
i . 5 8    9.02      -12       .          .
i . 6 9    7.01      -12       .          .
i . 7 10   6.00      -10       .          .
s
i 1 0 5    8.00      -10       2          100
i . 3 5    9.00      -14       .          .
i . 5 8    9.02      -12       .          .
i . 6 9    7.01      -12       .          .
i . 7 10   6.00      -10       .          .
</CsScore>
</CsoundSynthesizer>
```

As you can see, for non-harmonically related partials, the construction of a table must be done with a special care. If the frequency multipliers in our first example started with 1 and 1.02, the resulting period is acually very long. For a base frequency of 100 Hz, you will have the frequencies of 100 Hz and 102 Hz overlapping each other. So you need 100 cycles from the 1.00 multiplier and 102 cycles from the 1.02 multiplier to complete one period and to start again both together from zero. In other words, we have to create a table which contains 100 respectively 102 periods, instead of 1 and 1.02. Then the table values are not related to 1 - as usual - but to 100. That is the reason we have to introduce a new parameter *iTabFreq* for this purpose.

This method of composing waveforms can also be used for generating the four standard historical shapes used in a synthesizer. An **impulse** wave can be created by adding a number of harmonics of the same strength. A **sawtooth** has the amplitude multipliers 1, 1/2, 1/3, ... for the harmonics. A **square** has the same multipliers, but just for the odd harmonics. A **triangle** can be calculated as 1 divided by the square of the odd partials, with swaping positive and negative values. The next example creates function tables with just ten partials for each standard form.

EXAMPLE 04A04.csd

```
<CsoundSynthesizer>
<CsOptions>
-o dac
</CsOptions>
<CsInstruments>
;example by Joachim Heintz
sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

giImp  ftgen  1, 0, 4096, 10, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1
giSaw  ftgen  2, 0, 4096, 10, 1,1/2,1/3,1/4,1/5,1/6,1/7,1/8,1/9,1/10
giSqu  ftgen  3, 0, 4096, 10, 1, 0, 1/3, 0, 1/5, 0, 1/7, 0, 1/9, 0
giTri  ftgen  4, 0, 4096, 10, 1, 0, -1/9, 0, 1/25, 0, -1/49, 0, 1/81, 0

instr 1
asig   poscil .2, 457, p4
       outs   asig, asig
endin

</CsInstruments>
<CsScore>
i 1 0 3 1
i 1 4 3 2
i 1 8 3 3
i 1 12 3 4
</CsScore>
</CsoundSynthesizer>
```

TRIGGERING SUBINSTRUMENTS FOR THE PARTIALS

Performing additive synthesis by designing partial strengths into function tables has the disadvantage that once a note has begun there is no way of varying the relative strengths of individual partials. There are various methods to circumvent the inflexibility of table-based additive synthesis such as morphing between several tables (using for example the ftmorf opcode). Next we will consider another approach: triggering one instance of a subinstrument for each partial, and exploring the possibilities of creating a spectrally dynamic sound using this technique.

Let us return to the second instrument (05A02.csd) which already made some abstractions and triggered one instrument instance for each partial. This was done in the score; but now we will trigger one complete note in one score line, not just one partial. The first step is to assign the desired number of partials via a score parameter. The next example triggers any number of partials using this one value:

EXAMPLE 04A05.csd

```
<CsoundSynthesizer>
<CsOptions>
-o dac
</CsOptions>
<CsInstruments>
;Example by Joachim Heintz
sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

giSine    ftgen     0, 0, 2^10, 10, 1

instr 1 ;master instrument
inumparts =         p4 ;number of partials
ibasfreq  =         200 ;base frequency
ipart     =         1 ;count variable for loop
;loop for inumparts over the ipart variable
;and trigger inumpartss instanes of the subinstrument
loop:
ifreq     =         ibasfreq * ipart
iamp      =         1/ipart/inumparts
          event_i   "i", 10, 0, p3, ifreq, iamp
          loop_le   ipart, 1, inumparts, loop
endin

instr 10 ;subinstrument for playing one partial
ifreq     =         p4 ;frequency of this partial
iamp      =         p5 ;amplitude of this partial
```

```
aenv      transeg   0, .01, 0, iamp, p3-0.1, -10, 0
apart     poscil    aenv, ifreq, giSine
          outs      apart, apart
endin

</CsInstruments>
<CsScore>
;         number of partials
i 1 0 3   10
i 1 3 3   20
i 1 6 3   2
</CsScore>
</CsoundSynthesizer>
```

This instrument can easily be transformed to be played via a midi keyboard. The next example connects the number of synthesized partials with the midi velocity. So if you play softly, the sound will have fewer partials than if a key is struck with force.

EXAMPLE 04A06.csd

```
<CsoundSynthesizer>
<CsOptions>
-o dac
</CsOptions>
<CsInstruments>
;Example by Joachim Heintz
sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

giSine    ftgen     0, 0, 2^10, 10, 1
          massign   0, 1 ;all midi channels to instr 1

instr 1 ;master instrument
ibasfreq  cpsmidi ;base frequency
iampmid   ampmidi   20 ;receive midi-velocity and scale 0-20
inparts   =         int(iampmid)+1 ;exclude zero
ipart     =         1 ;count variable for loop
;loop for inumparts over the ipart variable
;and trigger inumpartss instanes of the subinstrument
loop:
ifreq     =         ibasfreq * ipart
iamp      =         1/ipart/inparts
          event_i   "i", 10, 0, 1, ifreq, iamp
          loop_le   ipart, 1, inparts, loop
endin

instr 10 ;subinstrument for playing one partial
ifreq     =         p4 ;frequency of this partial
iamp      =         p5 ;amplitude of this partial
aenv      transeg   0, .01, 0, iamp, p3-.01, -3, 0
apart     poscil    aenv, ifreq, giSine
          outs      apart/3, apart/3
endin

</CsInstruments>
<CsScore>
f 0 3600
</CsScore>
</CsoundSynthesizer>
```

Although this instrument is rather primitive it is useful to be able to control the timbre in this using key velocity. Let us continue to explore other methods of creating parameter variations in additive synthesis.

USER-CONTROLLED RANDOM VARIATIONS IN ADDITIVE SYNTHESIS

In natural sounds, there is movement and change all the time. Even the best player or singer will not be able to play a note in the exact same way twice. And inside a tone, the partials have some unsteadiness all the time: slight excitations of the amplitudes, uneven durations, slight frequency movements. In an audio programming language like Csound, we can achieve these movements with random deviations. It is not so important whether we use randomness or not, rather in which way. The boundaries of random deviations must be adjusted as carefully as with any other parameter in electronic composition. If sounds using random deviations begin to sound like mistakes then it is probably less to do with actually using random functions but instead more to do with some poorly chosen boundaries.

Let us start with some random deviations in our subinstrument. These parameters can be affected:

- The **frequency** of each partial can be slightly detuned. The range of this possible maximum detuning can be set in cents (100 cent = 1 semitone).
- The **amplitude** of each partial can be altered, compared to its standard value. The alteration can be measured in Decibel (dB).
- The **duration** of each partial can be shorter or longer than the standard value. Let us define this deviation as a percentage. If the expected duration is five seconds, a maximum deviation of 100% means getting a value between half the duration (2.5 sec) and the double duration (10 sec).

The following example shows the effect of these variations. As a base - and as a reference to its author - we take the "bell-like sound" which Jean-Claude Risset created in his Sound Catalogue.[1]

EXAMPLE 04A07.csd

```
<CsoundSynthesizer>
<CsOptions>
-o dac
</CsOptions>
<CsInstruments>
;Example by Joachim Heintz
sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

;frequency and amplitude multipliers for 11 partials of Risset's bell
giFqs     ftgen     0, 0, -11,-2,.56,.563,.92, .923,1.19,1.7,2,2.74,
                     3,3.74,4.07
giAmps    ftgen     0, 0, -11, -2, 1, 2/3, 1, 1.8, 8/3, 1.46, 4/3, 4/3, 1, 4/3
giSine    ftgen     0, 0, 2^10, 10, 1
          seed      0

instr 1 ;master instrument
ibasfreq  =         400
ifqdev    =         p4 ;maximum freq deviation in cents
iampdev   =         p5 ;maximum amp deviation in dB
idurdev   =         p6 ;maximum duration deviation in %
indx      =         0 ;count variable for loop
loop:
ifqmult   tab_i     indx, giFqs ;get frequency multiplier from table
ifreq     =         ibasfreq * ifqmult
iampmult  tab_i     indx, giAmps ;get amp multiplier
iamp      =         iampmult / 20 ;scale
          event_i   "i", 10, 0, p3, ifreq, iamp, ifqdev, iampdev, idurdev
          loop_lt   indx, 1, 11, loop
endin

instr 10 ;subinstrument for playing one partial
;receive the parameters from the master instrument
ifreqnorm =         p4 ;standard frequency of this partial
iampnorm  =         p5 ;standard amplitude of this partial
ifqdev    =         p6 ;maximum freq deviation in cents
iampdev   =         p7 ;maximum amp deviation in dB
idurdev   =         p8 ;maximum duration deviation in %
;calculate frequency
icent     random    -ifqdev, ifqdev ;cent deviation
ifreq     =         ifreqnorm * cent(icent)
;calculate amplitude
idb       random    -iampdev, iampdev ;dB deviation
iamp      =         iampnorm * ampdb(idb)
;calculate duration
idurperc  random    -idurdev, idurdev ;duration deviation (%)
iptdur    =         p3 * 2^(idurperc/100)
p3        =         iptdur ;set p3 to the calculated value
;play partial
aenv      transeg   0, .01, 0, iamp, p3-.01, -10, 0
apart     poscil    aenv, ifreq, giSine
          outs      apart, apart
endin

</CsInstruments>
<CsScore>
;         frequency   amplitude   duration
;         deviation   deviation   deviation
;         in cent     in dB       in %
```

```
;;unchanged sound (twice)
r 2
i 1 0 5   0           0           0
s
;;slight variations in frequency
r 4
i 1 0 5   25          0           0
;;slight variations in amplitude
r 4
i 1 0 5   0           6           0
;;slight variations in duration
r 4
i 1 0 5   0           0           30
;;slight variations combined
r 6
i 1 0 5   25          6           30
;;heavy variations
r 6
i 1 0 5   50          9           100
</CsScore>
</CsoundSynthesizer>
```

For a midi-triggered descendant of the instrument, we can - as one of many possible choices - vary the amount of possible random variation on the key velocity. So a key pressed softly plays the bell-like sound as described by Risset but as a key is struck with increasing force the sound produced will be increasingly altered.

EXAMPLE 04A08.csd

```
<CsoundSynthesizer>
<CsOptions>
-o dac
</CsOptions>
<CsInstruments>
;Example by Joachim Heintz
sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

;frequency and amplitude multipliers for 11 partials of Risset's bell
giFqs     ftgen     0, 0, -11, -2, .56,.563,.92,.923,1.19,1.7,2,2.74,3,
                    3.74,4.07
giAmps    ftgen     0, 0, -11, -2, 1, 2/3, 1, 1.8, 8/3, 1.46, 4/3, 4/3, 1, 4/3
giSine    ftgen     0, 0, 2^10, 10, 1
          seed      0
          massign   0, 1 ;all midi channels to instr 1

instr 1 ;master instrument
;;scale desired deviations for maximum velocity
;frequency (cent)
imxfqdv   =         100
;amplitude (dB)
imxampdv  =         12
;duration (%)
imxdurdv  =         100
;;get midi values
ibasfreq  cpsmidi ;base frequency
iampmid   ampmidi   1 ;receive midi-velocity and scale 0-1
;;calculate maximum deviations depending on midi-velocity
ifqdev    =         imxfqdv * iampmid
iampdev   =         imxampdv * iampmid
idurdev   =         imxdurdv * iampmid
;;trigger subinstruments
indx      =         0 ;count variable for loop
loop:
ifqmult   tab_i     indx, giFqs ;get frequency multiplier from table
ifreq     =         ibasfreq * ifqmult
iampmult  tab_i     indx, giAmps ;get amp multiplier
iamp      =         iampmult / 20 ;scale
          event_i   "i", 10, 0, 3, ifreq, iamp, ifqdev, iampdev, idurdev
          loop_lt   indx, 1, 11, loop
endin

instr 10 ;subinstrument for playing one partial
;receive the parameters from the master instrument
ifreqnorm =         p4 ;standard frequency of this partial
iampnorm  =         p5 ;standard amplitude of this partial
ifqdev    =         p6 ;maximum freq deviation in cents
iampdev   =         p7 ;maximum amp deviation in dB
idurdev   =         p8 ;maximum duration deviation in %
;calculate frequency
icent     random    -ifqdev, ifqdev ;cent deviation
ifreq     =         ifreqnorm * cent(icent)
```

```
;calculate amplitude
idb       random    -iampdev, iampdev ;dB deviation
iamp      =         iampnorm * ampdb(idb)
;calculate duration
idurperc  random    -idurdev, idurdev ;duration deviation (%)
iptdur    =         p3 * 2^(idurperc/100)
p3        =         iptdur ;set p3 to the calculated value
;play partial
aenv      transeg   0, .01, 0, iamp, p3-.01, -10, 0
apart     poscil    aenv, ifreq, giSine
          outs      apart, apart
endin

</CsInstruments>
<CsScore>
f 0 3600
</CsScore>
</CsoundSynthesizer>
```

It will depend on the power of your computer whether you can play examples like this in realtime. Have a look at chapter 2D (Live Audio) for tips on getting the best possible performance from your Csound orchestra.

Additive synthesis can still be an exciting way of producing sounds. The nowadays computational power and programming structures open the way for new discoverings and ideas. The later examples were intended to show some of these potentials of additive synthesis in Csound.

1. Jean-Claude Risset, Introductory Catalogue of Computer Synthesized Sounds (1969), cited after Dodge/Jerse, Computer Music, New York / London 1985, p.94^

21. SUBTRACTIVE SYNTHESIS

INTRODUCTION

Subtractive synthesis is, at least conceptually, the inverse of additive synthesis in that instead of building complex sound through the addition of simple cellular materials such as sine waves, subtractive synthesis begins with a complex sound source, such as white noise or a recorded sample, or a rich waveform, such as a sawtooth or pulse, and proceeds to refine that sound by removing partials or entire sections of the frequency spectrum through the use of audio filters.

The creation of dynamic spectra (an arduous task in additive synthesis) is relatively simple in subtractive synthesis as all that will be required will be to modulate a few parameters pertaining to any filters being used. Working with the intricate precision that is possible with additive synthesis may not be as easy with subtractive synthesis but sounds can be created much more instinctively than is possible with additive or FM synthesis.

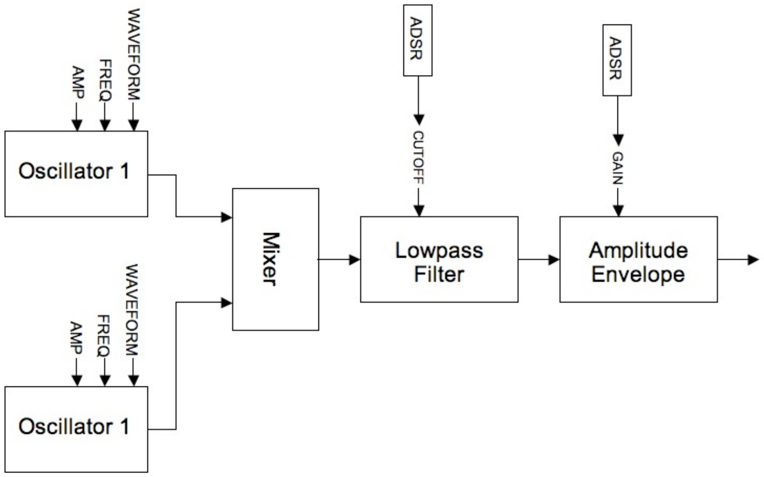
A CSOUND TWO-OSCILLATOR SYNTHESIZER

The first example represents perhaps the classic idea of subtractive synthesis: a simple two oscillator synth filtered using a single resonant lowpass filter. Many of the ideas used in this example have been inspired by the design of the Minimoog synthesizer (1970) and other similar instruments.

Each oscillator can describe either a sawtooth, PWM waveform (i.e. square - pulse etc.) or white noise and each oscillator can be transposed in octaves or in cents with respect to a fundamental pitch. The two oscillators are mixed and then passed through a 4-pole / 24dB per octave resonant lowpass filter. The opcode 'moogladder' is chosen on account of its authentic vintage character. The cutoff frequency of the filter is modulated using an ADSR-style (attack-decay-sustain-release) envelope facilitating the creation of dynamic, evolving spectra. Finally the sound output of the filter is shaped by an ADSR amplitude envelope.

As this instrument is suggestive of a performance instrument controlled via MIDI, this has been partially implemented. Through the use of Csound's MIDI interoperability opcode, mididefault, the instrument can be operated from the score or from a MIDI keyboard. If a MIDI note is received, suitable default p-field values are substituted for the missing p-fields. MIDI controller 1 can be used to control the global cutoff frequency for the filter.

A schematic for this instrument is shown below:



EXAMPLE 04B01.csd

```
<CsoundSynthesizer>
```

```
<CsOptions>
-odac -Ma
</CsOptions>

<CsInstruments>
sr = 44100
ksmps = 4
nchnls = 2
0dbfs = 1

initc7 1,1,0.8                 ;set initial controller position

prealloc 1, 10

   instr 1
iNum   notnum                  ;read in midi note number
iCF    ctrl7        1,1,0.1,14 ;read in midi controller 1

; set up default p-field values for midi activated notes
       mididefault  iNum, p4   ;pitch (note number)
       mididefault  0.3, p5    ;amplitude 1
       mididefault  2, p6      ;type 1
       mididefault  0.5, p7    ;pulse width 1
       mididefault  0, p8      ;octave disp. 1
       mididefault  0, p9      ;tuning disp. 1
       mididefault  0.3, p10   ;amplitude 2
       mididefault  1, p11     ;type 2
       mididefault  0.5, p12   ;pulse width 2
       mididefault  -1, p13    ;octave displacement 2
       mididefault  20, p14    ;tuning disp. 2
       mididefault  iCF, p15   ;filter cutoff freq
       mididefault  0.01, p16  ;filter env. attack time
       mididefault  1, p17     ;filter env. decay time
       mididefault  0.01, p18  ;filter env. sustain level
       mididefault  0.1, p19   ;filter release time
       mididefault  0.3, p20   ;filter resonance
       mididefault  0.01, p21  ;amp. env. attack
       mididefault  0.1, p22   ;amp. env. decay.
       mididefault  1, p23     ;amp. env. sustain
       mididefault  0.01, p24  ;amp. env. release

; asign p-fields to variables
iCPS   =            cpsmidinn(p4) ;convert from note number to cps
kAmp1  =            p5
iType1 =            p6
kPW1   =            p7
kOct1  =            octave(p8) ;convert from octave displacement to multiplier
kTune1 =            cent(p9)   ;convert from cents displacement to multiplier
kAmp2  =            p10
iType2 =            p11
kPW2   =            p12
kOct2  =            octave(p13)
kTune2 =            cent(p14)
iCF    =            p15
iFAtt  =            p16
iFDec  =            p17
iFSus  =            p18
iFRel  =            p19
kRes   =            p20
iAAtt  =            p21
iADec  =            p22
iASus  =            p23
iARel  =            p24

;oscillator 1
;if type is sawtooth or square...
if iType1==1||iType1==2 then
 ;...derive vco2 'mode' from waveform type
 iMode1 = (iType1=1?0:2)
 aSig1  vco2   kAmp1,iCPS*kOct1*kTune1,iMode1,kPW1;VCO audio oscillator
else                                   ;otherwise...
 aSig1  noise  kAmp1, 0.5              ;...generate white noise
endif

;oscillator 2 (identical in design to oscillator 1)
if iType2==1||iType2==2 then
 iMode2  =  (iType2=1?0:2)
 aSig2  vco2   kAmp2,iCPS*kOct2*kTune2,iMode2,kPW2
else
  aSig2 noise  kAmp2,0.5
endif

;mix oscillators
aMix       sum          aSig1,aSig2
;lowpass filter
kFiltEnv   expsegr      0.0001,iFAtt,iCPS*iCF,iFDec,iCPS*iCF*iFSus,iFRel,0.0001
```

```
aOut       moogladder   aMix, kFiltEnv, kRes

;amplitude envelope
aAmpEnv    expsegr      0.0001,iAAtt,1,iADec,iASus,iARel,0.0001
aOut       =            aOut*aAmpEnv
           outs         aOut,aOut
  endin
</CsInstruments>

<CsScore>
;p4  = oscillator frequency
;oscillator 1
;p5  = amplitude
;p6  = type (1=sawtooth,2=square-PWM,3=noise)
;p7  = PWM (square wave only)
;p8  = octave displacement
;p9  = tuning displacement (cents)
;oscillator 2
;p10 = amplitude
;p11 = type (1=sawtooth,2=square-PWM,3=noise)
;p12 = pwm (square wave only)
;p13 = octave displacement
;p14 = tuning displacement (cents)
;global filter envelope
;p15 = cutoff
;p16 = attack time
;p17 = decay time
;p18 = sustain level (fraction of cutoff)
;p19 = release time
;p20 = resonance
;global amplitude envelope
;p21 = attack time
;p22 = decay time
;p23 = sustain level
;p24 = release time
; p1 p2 p3  p4 p5  p6 p7   p8 p9  p10 p11 p12 p13
;p14 p15 p16  p17  p18  p19 p20 p21  p22 p23 p24
i 1  0  1   50 0   2  .5   0  -5  0   2   0.5 0   \
 5   12  .01  2    .01  .1  0   .005 .01 1   .05
i 1  +  1   50 .2  2  .5   0  -5  .2  2   0.5 0   \
 5   1   .01  1    .1   .1  .5  .005 .01 1   .05
i 1  +  1   50 .2  2  .5   0  -8  .2  2   0.5 0   \
 8   3   .01  1    .1   .1  .5  .005 .01 1   .05
i 1  +  1   50 .2  2  .5   0  -8  .2  2   0.5 -1  \
 8   7  .01   1    .1   .1  .5  .005 .01 1   .05
i 1  +  3   50 .2  1  .5   0  -10 .2  1   0.5 -2  \
 10  40  .01  3    .001 .1  .5  .005 .01 1   .05
i 1  +  10  50 1   2  .01  -2 0   .2  3   0.5 0   \
 0   40  5    5    .001 1.5 .1  .005 .01 1   .05

f 0 3600
e
</CsScore>

</CsoundSynthesizer>
```

SIMULATION OF TIMBRES FROM A NOISE SOURCE

The next example makes extensive use of bandpass filters arranged in parallel to filter white noise. The bandpass filter bandwidths are narrowed to the point where almost pure tones are audible. The crucial difference is that the noise source always induces instability in the amplitude and frequency of tones produced - it is this quality that makes this sort of subtractive synthesis sound much more organic than an additive synthesis equivalent. If the bandwidths are widened then more of the characteristic of the noise source comes through and the tone becomes 'airier' and less distinct; if the bandwidths are narrowed the resonating tones become clearer and steadier. By varying the bandwidths interesting metamorphoses of the resultant sound are possible.

22 reson filters are used for the bandpass filters on account of their ability to ring and resonate as their bandwidth narrows. Another reason for this choice is the relative CPU economy of the reson filter, a not inconsiderable concern as so many of them are used. The frequency ratios between the 22 parallel filters are derived from analysis of a hand bell, the data was found in the appendix of the Csound manual here.

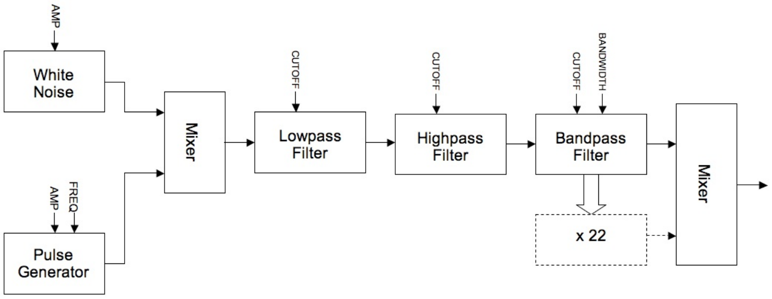
In addition to the white noise as a source, noise impulses are also used as a sound source (via the 'mpulse' opcode). The instrument will automatically and randomly slowly crossfade between these two sound sources.

A lowpass and highpass filter are inserted in series before the parallel bandpass filters to shape the frequency spectrum of the source sound. Csound's butterworth filters butlp and buthp are chosen for this task on account of their steep cutoff slopes and lack of ripple at the cutoff point.

The outputs of the reson filters are sent alternately to the left and right outputs in order to create a broad stereo effect.

This example makes extensive use of the 'rspline' opcode, a generator of random spline functions, to slowly undulate the many input parameters. The orchestra is self generative in that instrument 1 repeatedly triggers note events in instrument 2 and the extensive use of random functions means that the results will continually evolve as the orchestra is allowed to perform.

A flow diagram for this instrument is shown below:



EXAMPLE 04B02.csd

```
<CsoundSynthesizer>

<CsOptions>
-odac
</CsOptions>

<CsInstruments>
;Example written by Iain McCurdy

sr = 44100
ksmps = 16
nchnls = 2
0dbfs = 1

  instr 1 ; triggers notes in instrument 2 with randomised p-fields
krate  randomi 0.2,0.4,0.1   ;rate of note generation
ktrig  metro  krate          ;triggers used by schedkwhen
koct   random 5,12           ;fundemental pitch of synth note
kdur   random 15,30          ;duration of note
schedkwhen ktrig,0,0,2,0,kdur,cpsoct(koct) ;trigger a note in instrument 2
  endin

  instr 2 ; subtractive synthesis instrument
aNoise  pinkish  1                  ;a noise source sound: pink noise
kGap    rspline  0.3,0.05,0.2,2     ;time gap between impulses
aPulse  mpulse   15, kGap           ;a train of impulses
kCFade  rspline  0,1,0.1,1          ;crossfade point between noise and impulses
aInput  ntrpol   aPulse,aNoise,kCFade;implement crossfade

; cutoff frequencies for low and highpass filters
kLPF_CF  rspline  13,8,0.1,0.4
kHPF_CF  rspline  5,10,0.1,0.4
; filter input sound with low and highpass filters in series -
; - done twice per filter in order to sharpen cutoff slopes
aInput    butlp    aInput, cpsoct(kLPF_CF)
aInput    butlp    aInput, cpsoct(kLPF_CF)
aInput    buthp    aInput, cpsoct(kHPF_CF)
aInput    buthp    aInput, cpsoct(kHPF_CF)

kcf     rspline  p4*1.05,p4*0.95,0.01,0.1 ; fundemental
; bandwidth for each filter is created individually as a random spline function
kbw1    rspline  0.00001,10,0.2,1
kbw2    rspline  0.00001,10,0.2,1
kbw3    rspline  0.00001,10,0.2,1
kbw4    rspline  0.00001,10,0.2,1
kbw5    rspline  0.00001,10,0.2,1
kbw6    rspline  0.00001,10,0.2,1
kbw7    rspline  0.00001,10,0.2,1
kbw8    rspline  0.00001,10,0.2,1
```

```
kbw9    rspline  0.00001,10,0.2,1
kbw10   rspline  0.00001,10,0.2,1
kbw11   rspline  0.00001,10,0.2,1
kbw12   rspline  0.00001,10,0.2,1
kbw13   rspline  0.00001,10,0.2,1
kbw14   rspline  0.00001,10,0.2,1
kbw15   rspline  0.00001,10,0.2,1
kbw16   rspline  0.00001,10,0.2,1
kbw17   rspline  0.00001,10,0.2,1
kbw18   rspline  0.00001,10,0.2,1
kbw19   rspline  0.00001,10,0.2,1
kbw20   rspline  0.00001,10,0.2,1
kbw21   rspline  0.00001,10,0.2,1
kbw22   rspline  0.00001,10,0.2,1

imode   =        0 ; amplitude balancing method used by the reson filters
a1      reson    aInput, kcf*1,               kbw1, imode
a2      reson    aInput, kcf*1.0019054878049, kbw2, imode
a3      reson    aInput, kcf*1.7936737804878, kbw3, imode
a4      reson    aInput, kcf*1.8009908536585, kbw4, imode
a5      reson    aInput, kcf*2.5201981707317, kbw5, imode
a6      reson    aInput, kcf*2.5224085365854, kbw6, imode
a7      reson    aInput, kcf*2.9907012195122, kbw7, imode
a8      reson    aInput, kcf*2.9940548780488, kbw8, imode
a9      reson    aInput, kcf*3.7855182926829, kbw9, imode
a10     reson    aInput, kcf*3.8061737804878, kbw10,imode
a11     reson    aInput, kcf*4.5689024390244, kbw11,imode
a12     reson    aInput, kcf*4.5754573170732, kbw12,imode
a13     reson    aInput, kcf*5.0296493902439, kbw13,imode
a14     reson    aInput, kcf*5.0455030487805, kbw14,imode
a15     reson    aInput, kcf*6.0759908536585, kbw15,imode
a16     reson    aInput, kcf*5.9094512195122, kbw16,imode
a17     reson    aInput, kcf*6.4124237804878, kbw17,imode
a18     reson    aInput, kcf*6.4430640243902, kbw18,imode
a19     reson    aInput, kcf*7.0826219512195, kbw19,imode
a20     reson    aInput, kcf*7.0923780487805, kbw20,imode
a21     reson    aInput, kcf*7.3188262195122, kbw21,imode
a22     reson    aInput, kcf*7.5551829268293, kbw22,imode

; amplitude control for each filter output
kAmp1    rspline  0, 1, 0.3, 1
kAmp2    rspline  0, 1, 0.3, 1
kAmp3    rspline  0, 1, 0.3, 1
kAmp4    rspline  0, 1, 0.3, 1
kAmp5    rspline  0, 1, 0.3, 1
kAmp6    rspline  0, 1, 0.3, 1
kAmp7    rspline  0, 1, 0.3, 1
kAmp8    rspline  0, 1, 0.3, 1
kAmp9    rspline  0, 1, 0.3, 1
kAmp10   rspline  0, 1, 0.3, 1
kAmp11   rspline  0, 1, 0.3, 1
kAmp12   rspline  0, 1, 0.3, 1
kAmp13   rspline  0, 1, 0.3, 1
kAmp14   rspline  0, 1, 0.3, 1
kAmp15   rspline  0, 1, 0.3, 1
kAmp16   rspline  0, 1, 0.3, 1
kAmp17   rspline  0, 1, 0.3, 1
kAmp18   rspline  0, 1, 0.3, 1
kAmp19   rspline  0, 1, 0.3, 1
kAmp20   rspline  0, 1, 0.3, 1
kAmp21   rspline  0, 1, 0.3, 1
kAmp22   rspline  0, 1, 0.3, 1

; left and right channel mixes are created using alternate filter outputs.
; This shall create a stereo effect.
aMixL    sum      a1*kAmp1,a3*kAmp3,a5*kAmp5,a7*kAmp7,a9*kAmp9,a11*kAmp11,\
                        a13*kAmp13,a15*kAmp15,a17*kAmp17,a19*kAmp19,a21*kAmp21
aMixR    sum      a2*kAmp2,a4*kAmp4,a6*kAmp6,a8*kAmp8,a10*kAmp10,a12*kAmp12,\
                        a14*kAmp14,a16*kAmp16,a18*kAmp18,a20*kAmp20,a22*kAmp22

kEnv     linseg   0, p3*0.5, 1,p3*0.5,0,1,0       ; global amplitude envelope
outs   (aMixL*kEnv*0.00008), (aMixR*kEnv*0.00008) ; audio sent to outputs
  endin

</CsInstruments>

<CsScore>
i 1 0 3600  ; instrument 1 (note generator) plays for 1 hour
e
</CsScore>

</CsoundSynthesizer>
```

VOWEL-SOUND EMULATION USING BANDPASS FILTERING

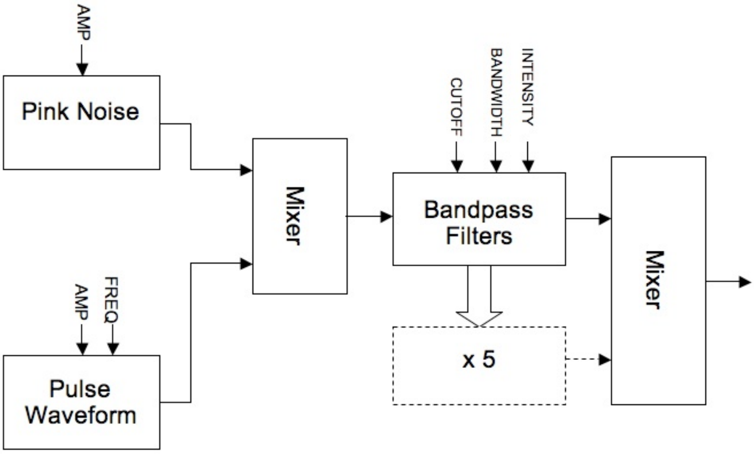
The final example in this section uses precisely tuned bandpass filters, to simulate the sound of the human voice expressing vowel sounds. Spectral resonances in this context are often referred to as 'formants'. Five formants are used to simulate the effect of the human mouth and head as a resonating (and therefore filtering) body. The filter data for simulating the vowel sounds A,E,I,O and U as expressed by a bass, tenor, counter-tenor, alto and soprano voice were found in the appendix of the Csound manual here. Bandwidth and intensity (dB) information is also needed to accurately simulate the various vowel sounds.

reson filters are again used but butbp and others could be equally valid choices.

Data is stored in GEN07 linear break point function tables, as this data is read by k-rate line functions we can interpolate and therefore morph between different vowel sounds during a note.

The source sound for the filters comes from either a pink noise generator or a pulse waveform. The pink noise source could be used if the emulation is to be that of just the breath whereas the pulse waveform provides a decent approximation of the human vocal chords buzzing. This instrument can however morph continuously between these two sources.

A flow diagram for this instrument is shown below:



EXAMPLE 04B03.csd

```
<CsoundSynthesizer>

<CsOptions>
-odac
</CsOptions>

<CsInstruments>
;example by Iain McCurdy

sr = 44100
ksmps = 16
nchnls = 2
0dbfs = 1

instr 1
  kFund    expon     p4,p3,p5               ; fundamental
  kVow     line      p6,p3,p7               ; vowel select
  kBW      line      p8,p3,p9               ; bandwidth factor
  iVoice   =         p10                    ; voice select
  kSrc     line      p11,p3,p12             ; source mix

  aNoise   pinkish   3                      ; pink noise
  aVCO     vco2      1.2,kFund,2,0.02       ; pulse tone
  aInput   ntrpol    aVCO,aNoise,kSrc       ; input mix

  ; read formant cutoff frequenies from tables
  kCF1     table     kVow,1+(iVoice*15),1
  kCF2     table     kVow,2+(iVoice*15),1
  kCF3     table     kVow,3+(iVoice*15),1
  kCF4     table     kVow,4+(iVoice*15),1
  kCF5     table     kVow,5+(iVoice*15),1
  ; read formant intensity values from tables
  kDB1     table     kVow,6+(iVoice*15),1
  kDB2     table     kVow,7+(iVoice*15),1
  kDB3     table     kVow,8+(iVoice*15),1
  kDB4     table     kVow,9+(iVoice*15),1
  kDB5     table     kVow,10+(iVoice*15),1
  ; read formant bandwidths from tables
  kBW1     table     kVow,11+(iVoice*15),1
  kBW2     table     kVow,12+(iVoice*15),1
  kBW3     table     kVow,13+(iVoice*15),1
  kBW4     table     kVow,14+(iVoice*15),1
  kBW5     table     kVow,15+(iVoice*15),1
  ; create resonant formants byt filtering source sound
  aForm1   reson     aInput, kCF1, kBW1*kBW, 1     ; formant 1
  aForm2   reson     aInput, kCF2, kBW2*kBW, 1     ; formant 2
  aForm3   reson     aInput, kCF3, kBW3*kBW, 1     ; formant 3
  aForm4   reson     aInput, kCF4, kBW4*kBW, 1     ; formant 4
  aForm5   reson     aInput, kCF5, kBW5*kBW, 1     ; formant 5

  ; formants are mixed and multiplied both by intensity values derived from
tables and by the on-screen gain controls for each formant
  aMix     sum
aForm1*ampdbfs(kDB1),aForm2*ampdbfs(kDB2),aForm3*ampdbfs(kDB3),aForm4*ampdbfs(kDB4

  kEnv     linseg    0,3,1,p3-6,1,3,0     ; an amplitude envelope
           outs      aMix*kEnv, aMix*kEnv ; send audio to outputs
endin

</CsInstruments>

<CsScore>
f 0 3600 ;DUMMY SCORE EVENT - PERMITS REALTIME PERFORMANCE FOR UP TO 1 HOUR

;FUNCTION TABLES STORING FORMANT DATA FOR EACH OF THE FIVE VOICE TYPES
REPRESENTED
;BASS
f 1  0 32768 -7 600 10922 400 10922 250 10924 350 ;FREQ
f 2  0 32768 -7 1040 10922 1620 10922 1750 10924 600 ;FREQ
f 3  0 32768 -7 2250 10922 2400 10922 2600 10924 2400 ;FREQ
f 4  0 32768 -7 2450 10922 2800 10922 3050 10924 2675 ;FREQ
f 5  0 32768 -7 2750 10922 3100 10922 3340 10924 2950 ;FREQ
f 6  0 32768 -7 0 10922 0 10922 0 10924 0 ;dB
f 7  0 32768 -7 -7 10922 -12 10922 -30 10924 -20 ;dB
f 8  0 32768 -7 -9 10922 -9 10922 -16 10924 -32 ;dB
f 9  0 32768 -7 -9 10922 -12 10922 -22 10924 -28 ;dB
f 10 0 32768 -7 -20 10922 -18 10922 -28 10924 -36 ;dB
f 11 0 32768 -7 60 10922 40 10922 60 10924 40 ;BAND WIDTH
f 12 0 32768 -7 70 10922 80 10922 90 10924 80 ;BAND WIDTH
f 13 0 32768 -7 110 10922 100 10922 100 10924 100 ;BAND WIDTH
f 14 0 32768 -7 120 10922 120 10922 120 10924 120 ;BAND WIDTH
f 15 0 32768 -7 130 10922 120 10922 120 10924 120 ;BAND WIDTH
;TENOR
f 16 0 32768 -7 650  8192 400  8192 290 8192 400 8192 350 ;FREQ
f 17 0 32768 -7 1080  8192 1700    8192 1870 8192 800 8192 600 ;FREQ
f 18 0 32768 -7 2650 8192 2600    8192 2800 8192 2600 8192 2700 ;FREQ
f 19 0 32768 -7 2900 8192 3200    8192 3250 8192 2800 8192 2900 ;FREQ
f 20 0 32768 -7 3250 8192 3580    8192 3540 8192 3000 8192 3300 ;FREQ
f 21 0 32768 -7 0 8192 0 8192 0 8192 0 8192 0 ;dB
f 22 0 32768 -7 -6 8192 -14 8192 -15 8192 -10 8192 -20 ;dB
f 23 0 32768 -7 -7 8192 -12 8192 -18 8192 -12 8192 -17 ;dB
f 24 0 32768 -7 -8 8192 -14 8192 -20 8192 -12 8192 -14 ;dB
f 25 0 32768 -7 -22 8192 -20 8192 -30 8192 -26 8192 -26 ;dB
f 26 0 32768 -7 80 8192 70 8192 40 8192 40 8192 40 ;BAND WIDTH
f 27 0 32768 -7 90 8192 80 8192 90 8192 80 8192 60 ;BAND WIDTH
f 28 0 32768 -7 120 8192 100 8192 100 8192 100 8192 100 ;BAND WIDTH
f 29 0 32768 -7 130 8192 120 8192 120 8192 120 8192 120 ;BAND WIDTH
f 30 0 32768 -7 140 8192 120 8192 120 8192 120 8192 120 ;BAND WIDTH
;COUNTER TENOR
f 31 0 32768 -7 660 8192 440 8192 270 8192 430 8192 370 ;FREQ
f 32 0 32768 -7 1120 8192 1800 8192 1850 8192 820 8192 630 ;FREQ
f 33 0 32768 -7 2750 8192 2700 8192 2900 8192 2700 8192 2750 ;FREQ
f 34 0 32768 -7 3000 8192 3000 8192 3350 8192 3000 8192 3000 ;FREQ
f 35 0 32768 -7 3350 8192 3300 8192 3590 8192 3300 8192 3400 ;FREQ
f 36 0 32768 -7 0 8192 0 8192 0 8192 0 8192 0 ;dB
f 37 0 32768 -7 -6 8192 -14 8192 -24 8192 -10 8192 -20 ;dB
f 38 0 32768 -7 -23 8192 -18 8192 -24 8192 -26 8192 -23 ;dB
f 39 0 32768 -7 -24 8192 -20 8192 -36 8192 -22 8192 -30 ;dB
f 40 0 32768 -7 -38 8192 -20 8192 -36 8192 -34 8192 -30 ;dB
f 41 0 32768 -7 80 8192 70 8192 40 8192 40 8192 40 ;BAND WIDTH
f 42 0 32768 -7 90 8192 80 8192 90 8192 80 8192 60 ;BAND WIDTH
f 43 0 32768 -7 120 8192 100 8192 100 8192 100 8192 100 ;BAND WIDTH
f 44 0 32768 -7 130 8192 120 8192 120 8192 120 8192 120 ;BAND WIDTH
f 45 0 32768 -7 140 8192 120 8192 120 8192 120 8192 120 ;BAND WIDTH
;ALTO
f 46 0 32768 -7 800 8192 400 8192 350 8192 450 8192 325 ;FREQ
f 47 0 32768 -7 1150 8192 1600 8192 1700 8192 800 8192 700 ;FREQ
f 48 0 32768 -7 2800 8192 2700 8192 2700 8192 2830 8192 2530 ;FREQ
f 49 0 32768 -7 3500 8192 3300 8192 3700 8192 3500 8192 2500 ;FREQ
```

```
f 50 0 32768 -7 4950 8192 4950 8192 4950 8192 4950 8192 4950 ;FREQ
f 51 0 32768 -7 0 8192 0 8192 0 8192 0 8192 0 ;dB
f 52 0 32768 -7 -4 8192 -24 8192 -20 8192 -9 8192 -12 ;dB
f 53 0 32768 -7 -20 8192 -30 8192 -30 8192 -16 8192 -30 ;dB
f 54 0 32768 -7 -36 8192 -35 8192 -36 8192 -28 8192 -40 ;dB
f 55 0 32768 -7 -60 8192 -60 8192 -60 8192 -55 8192 -64 ;dB
f 56 0 32768 -7 50 8192 60 8192 50 8192 70 8192 50 ;BAND WIDTH
f 57 0 32768 -7 60 8192 80 8192 100 8192 80 8192 60 ;BAND WIDTH
f 58 0 32768 -7 170 8192 120 8192 120 8192 100 8192 170 ;BAND WIDTH
f 59 0 32768 -7 180 8192 150 8192 150 8192 130 8192 180 ;BAND WIDTH
f 60 0 32768 -7 200 8192 200 8192 200 8192 135 8192 200 ;BAND WIDTH
;SOPRANO
f 61 0 32768 -7 800 8192 350 8192 270 8192 450 8192 325 ;FREQ
f 62 0 32768 -7 1150 8192 2000 8192 2140 8192 800 8192 700 ;FREQ
f 63 0 32768 -7 2900 8192 2800 8192 2950 8192 2830 8192 2700 ;FREQ
f 64 0 32768 -7 3900 8192 3600 8192 3900 8192 3800 8192 3800 ;FREQ
f 65 0 32768 -7 4950 8192 4950 8192 4950 8192 4950 8192 4950 ;FREQ
f 66 0 32768 -7 0 8192 0 8192 0 8192 0 8192 0 ;dB
f 67 0 32768 -7 -6 8192 -20 8192 -12 8192 -11 8192 -16 ;dB
f 68 0 32768 -7 -32 8192 -15 8192 -26 8192 -22 8192 -35 ;dB
f 69 0 32768 -7 -20 8192 -40 8192 -26 8192 -22 8192 -40 ;dB
f 70 0 32768 -7 -50 8192 -56 8192 -44 8192 -50 8192 -60 ;dB
f 71 0 32768 -7 80 8192 60 8192 60 8192 70 8192 50 ;BAND WIDTH
f 72 0 32768 -7 90 8192 90 8192 90 8192 80 8192 60 ;BAND WIDTH
f 73 0 32768 -7 120 8192 100 8192 100 8192 100 8192 170 ;BAND WIDTH
f 74 0 32768 -7 130 8192 150 8192 120 8192 130 8192 180 ;BAND WIDTH
f 75 0 32768 -7 140 8192 200 8192 120 8192 135 8192 200 ;BAND WIDTH

; p4 = fundemental begin value (c.p.s.)
; p5 = fundemental end value
; p6 = vowel begin value (0 - 1 : a e i o u)
; p7 = vowel end value
; p8 = bandwidth factor begin (suggested range 0 - 2)
; p9 = bandwidth factor end
; p10 = voice (0=bass; 1=tenor; 2=counter_tenor; 3=alto; 4=soprano)
; p11 = input source begin (0 - 1 : VCO - noise)
; p12 = input source end

;         p4  p5  p6  p7  p8  p9 p10 p11  p12
i 1 0  10 50  100 0   1   2   0  0   0    0
i 1 8  .  78  77  1   0   1   0  1   0    0
i 1 16 .  150 118 0   1   1   0  2   1    1
i 1 24 .  200 220 1   0   0.2 0  3   1    0
i 1 32 .  400 800 0   1   0.2 0  4   0    1
e
</CsScore>

</CsoundSynthesizer>
```

CONCLUSION

These examples have hopefully demonstrated the strengths of subtractive synthesis in its simplicity, intuitive operation and its ability to create organic sounding timbres. Further research could explore Csound's other filter opcodes including vcomb, wguide1, wguide2 and the more esoteric phaser1, phaser2 and resony.

22. AMPLITUDE AND RING MODULATION

INTRODUCTION

Amplitude-modulation (AM) means, that one oscillator varies the volume/amplitude of an other. If this modulation is done very slowly (1 Hz to 10 Hz) it is recognised as tremolo. Volume-modulation above 10 Hz leads to the effect, that the sound changes its timbre. So called side-bands appear.

Example 04C01.csd

```
<CsoundSynthesizer>
<CsOptions>
-o dac
</CsOptions>
<CsInstruments>

sr = 48000
ksmps = 32
nchnls = 1
0dbfs = 1

instr 1
aRaise expseg 2, 20, 100
aModSine poscil 0.5, aRaise, 1
aDCOffset = 0.5    ; we want amplitude-modulation
aCarSine poscil 0.3, 440, 1
out aCarSine*(aModSine + aDCOffset)
endin

</CsInstruments>
<CsScore>
f 1 0 1024 10 1
i 1 0 25
e
</CsScore>
</CsoundSynthesizer>
; written by Alex Hofmann (Mar. 2011)
```

THEORY, MATHEMATICS AND SIDEBANDS

The side-bands appear on both sides of the main frequency. This means (freq1-freq2) and (freq1+freq2) appear.

The sounding result of the following example can be calculated as this: freq1 = 440Hz, freq2 = 40 Hz -> The result is a sound with [400, 440, 480] Hz.

The amount of the sidebands can be controlled by a DC-offset of the modulator.

Example 04C02.csd

```
<CsoundSynthesizer>
<CsOptions>
-o dac
</CsOptions>
<CsInstruments>

sr = 48000
ksmps = 32
nchnls = 1
0dbfs = 1

instr 1
aOffset linseg 0, 1, 0, 5, 0.6, 3, 0
aSine1 poscil 0.3, 40 , 1
aSine2 poscil 0.3, 440, 1
out (aSine1+aOffset)*aSine2
endin


</CsInstruments>
<CsScore>
f 1 0 1024 10 1
i 1 0 10
```

```
e
</CsScore>
</CsoundSynthesizer>
; written by Alex Hofmann (Mar. 2011)
```

Ring-modulation is a special-case of AM, without DC-offset (DC-Offset = 0). That means the modulator varies between -1 and +1 like the carrier. The sounding difference to AM is, that RM doesn't contain the carrier frequency.

(If the modulator is unipolar (oscillates between 0 and +1) the effect is called AM.)

MORE COMPLEX SYNTHESIS USING RING MODULATION AND AMPLITUDE MODULATION

If the modulator itself has more harmonics, the result becomes easily more complex.

Carrier freq: 600 Hz
Modulator freqs: 200Hz with 3 harmonics = [200, 400, 600] Hz
Resulting freqs: [0, 200, 400, <-600->, 800, 1000, 1200]

Example 04C03.csd

```
<CsoundSynthesizer>
<CsOptions>
-o dac
</CsOptions>
<CsInstruments>

sr = 48000
ksmps = 32
nchnls = 1
0dbfs = 1

instr 1   ; Ring-Modulation (no DC-Offset)
aSine1 poscil 0.3, 200, 2 ; -> [200, 400, 600] Hz
aSine2 poscil 0.3, 600, 1
out aSine1*aSine2
endin

</CsInstruments>
<CsScore>
f 1 0 1024 10 1 ; sine
f 2 0 1024 10 1 1 1; 3 harmonics
i 1 0 5
e
</CsScore>
</CsoundSynthesizer>
; written by Alex Hofmann (Mar. 2011)
```

Using an inharmonic modulator frequency also makes the result sound inharmonic. Varying the DC-offset makes the sound-spectrum evolve over time.
Modulator freqs: [230, 460, 690]
Resulting freqs: [(-)90, 140, 370, <-600->, 830, 1060, 1290]
(negative frequencies become mirrowed, but phase inverted)

Example 04C04.csd

```
<CsoundSynthesizer>
<CsOptions>
-o dac
</CsOptions>
<CsInstruments>

sr = 48000
ksmps = 32
nchnls = 1
0dbfs = 1

instr 1   ; Amplitude-Modulation
aOffset linseg 0, 1, 0, 5, 1, 3, 0
aSine1 poscil 0.3, 230, 2 ; -> [230, 460, 690] Hz
aSine2 poscil 0.3, 600, 1
out (aSine1+aOffset)*aSine2
endin

</CsInstruments>
<CsScore>
f 1 0 1024 10 1 ; sine
```

```
f 2 0 1024 10 1 1 1; 3 harmonics
i 1 0 10
e
</CsScore>
</CsoundSynthesizer>
```

23. FREQUENCY MODULATION

FROM VIBRATO TO THE EMERGENCE OF SIDEBANDS

A vibrato is a periodical change of pitch, normally less than a halftone and with a slow changing-rate (around 5Hz). Frequency modulation is usually done with sine-wave oscillators.

Example 04D01.csd

```
<CsoundSynthesizer>
<CsOptions>
-o dac
</CsOptions>
<CsInstruments>
sr = 48000
ksmps = 32
nchnls = 2
0dbfs = 1

instr 1
aMod poscil 10, 5 , 1  ; 5 Hz vibrato with 10 Hz modulation-width
aCar poscil 0.3, 440+aMod, 1  ; -> vibrato between 430-450 Hz
outs aCar, aCar
endin

</CsInstruments>
<CsScore>
f 1 0 1024 10 1   ;Sine wave for table 1
i 1 0 2
</CsScore>
</CsoundSynthesizer>
; written by Alex Hofmann (Mar. 2011)
```

When the modulation-width becomes increased, it becomes harder to describe the base-frequency, but it is still a vibrato.

Example 04D02.csd

```
<CsoundSynthesizer>
<CsOptions>
-o dac
</CsOptions>
<CsInstruments>
sr = 48000
ksmps = 32
nchnls = 2
0dbfs = 1

instr 1
aMod poscil 90, 5 , 1 ; modulate 90Hz ->vibrato from 350 to 530 hz
aCar poscil 0.3, 440+aMod, 1
outs aCar, aCar
endin

</CsInstruments>
<CsScore>
f 1 0 1024 10 1   ;Sine wave for table 1
i 1 0 2
</CsScore>
</CsoundSynthesizer>
; written by Alex Hofmann (Mar. 2011)
```

THE SIMPLE MODULATOR->CARRIER PAIRING

Increasing the modulation-rate leads to a different effect. Frequency-modulation with more than 20Hz is no longer recognized as vibrato. The main-oscillator frequency lays in the middle of the sound and sidebands appear above and below. The number of sidebands is related to the modulation amplitude, later this is controlled by the so called *modulation-index*.

Example 04D03.csd

```
<CsoundSynthesizer>
<CsOptions>
```

```
-o dac
</CsOptions>
<CsInstruments>
sr = 48000
ksmps = 32
nchnls = 2
0dbfs = 1

instr 1
aRaise linseg 2, 10, 100    ;increase modulation from 2Hz to 100Hz
aMod poscil 10, aRaise , 1
aCar poscil 0.3, 440+aMod, 1
outs aCar, aCar
endin

</CsInstruments>
<CsScore>
f 1 0 1024 10 1   ;Sine wave for table 1
i 1 0 12
</CsScore>
</CsoundSynthesizer>
; written by Alex Hofmann (Mar. 2011
```

Hereby the main-oscillator is called *carrier* and the one changing the carriers frequency is the *modulator*. The *modulation-index*: **I = mod-amp/mod-freq**. Making changes to the modulation-index, changes the amount of overtones, but not the overall volume. That gives the possibility produce drastic timbre-changes without the risk of distortion.

When *carrier* and *modulator* frequency have integer ratios like 1:1, 2:1, 3:2, 5:4.. the sidebands build a harmonic series, which leads to a sound with clear fundamental pitch.

Example 04D04.csd

```
<CsoundSynthesizer>
<CsOptions>
-o dac
</CsOptions>
<CsInstruments>
sr = 48000
ksmps = 32
nchnls = 2
0dbfs = 1

instr 1
kCarFreq = 660     ; 660:440 = 3:2 -> harmonic spectrum
kModFreq = 440
kIndex = 15        ; high Index.. try lower values like 1, 2, 3..
kIndexM = 0
kMaxDev = kIndex*kModFreq
kMinDev = kIndexM*kModFreq
kVarDev = kMaxDev-kMinDev
kModAmp = kMinDev+kVarDev
aModulator poscil kModAmp, kModFreq, 1
aCarrier poscil 0.3, kCarFreq+aModulator, 1
outs aCarrier, aCarrier
endin


</CsInstruments>
<CsScore>
f 1 0 1024 10 1   ;Sine wave for table 1
i 1 0 15
</CsScore>
</CsoundSynthesizer>
; written by Alex Hofmann (Mar. 2011)
```

Otherwise the spectrum of the sound is inharmonic, which makes it metallic or noisy.
Raising the *modulation-index*, shifts the energy into the side-bands. The side-bands distance is:
Distance in Hz = (carrierFreq)-(k*modFreq) | k = {1, 2, 3, 4 ..}

This calculation can result in negative frequencies. Those become reflected at zero, but with inverted phase! So negative frequencies can erase existing ones. Frequencies over Nyquist-frequency (half of samplingrate) "fold over" (aliasing).

THE JOHN CHOWNING FM MODEL OF A TRUMPET

Composer and researcher Jown Chowning worked on the first digital implementation of FM in the 1970's.

Using envelopes to control the *modulation index* and the overall amplitude gives you the possibility to create evolving sounds with enormous spectral variations. Chowning showed these possibilities in his pieces, where he let the sounds transform. In the piece *Sabelithe* a drum sound morphes over the time into a trumpet tone.

Example 04D05.csd

```
<CsoundSynthesizer>
<CsOptions>
-o dac
</CsOptions>
<CsInstruments>
sr = 48000
ksmps = 32
nchnls = 2
0dbfs = 1

instr 1  ; simple way to generate a trumpet-like sound
kCarFreq = 440
kModFreq = 440
kIndex = 5
kIndexM = 0
kMaxDev = kIndex*kModFreq
kMinDev = kIndexM * kModFreq
kVarDev = kMaxDev-kMinDev
aEnv expseg .001, 0.2, 1, p3-0.3, 1, 0.2, 0.001
aModAmp = kMinDev+kVarDev*aEnv
aModulator poscil aModAmp, kModFreq, 1
aCarrier poscil 0.3*aEnv, kCarFreq+aModulator, 1
outs aCarrier, aCarrier
endin


</CsInstruments>
<CsScore>
f 1 0 1024 10 1   ;Sine wave for table 1
i 1 0 2
</CsScore>
</CsoundSynthesizer>
; written by Alex Hofmann (Mar. 2011)
```

The following example uses the same instrument, with different settings to generate a bell-like sound:

Example 04D06.csd

```
<CsoundSynthesizer>
<CsOptions>
-o dac
</CsOptions>
<CsInstruments>
sr = 48000
ksmps = 32
nchnls = 2
0dbfs = 1

instr 1  ; bell-like sound
kCarFreq = 200  ; 200/280 = 5:7 -> inharmonic spectrum
kModFreq = 280
kIndex = 12
kIndexM = 0
kMaxDev = kIndex*kModFreq
kMinDev = kIndexM * kModFreq
kVarDev = kMaxDev-kMinDev
aEnv expseg .001, 0.001, 1, 0.3, 0.5, 8.5, .001
aModAmp = kMinDev+kVarDev*aEnv
aModulator poscil aModAmp, kModFreq, 1
aCarrier poscil 0.3*aEnv, kCarFreq+aModulator, 1
outs aCarrier, aCarrier
endin

</CsInstruments>
<CsScore>
f 1 0 1024 10 1   ;Sine wave for table 1
i 1 0 9
</CsScore>
</CsoundSynthesizer>
; written by Alex Hofmann (Mar. 2011)
```

MORE COMPLEX FM ALGORITHMS

Combining more than two oscillators (operators) is called complex FM synthesis. Operators can be connected in different combinations often 4-6 operators are used. The carrier is always the last operator in the row. Changing it's pitch, shifts the whole sound. All other operators are modulators, changing their pitch alters the sound-spectrum.

Two into One: M1+M2 -> C

The principle here is, that (M1:C) and (M2:C) will be separate modulations and later added together.

Example 04D07.csd

```
<CsoundSynthesizer>
<CsOptions>
-o dac
</CsOptions>
<CsInstruments>
sr = 48000
ksmps = 32
nchnls = 2
0dbfs = 1

instr 1
aMod1 poscil 200, 700, 1
aMod2 poscil 1800, 290, 1
aSig poscil 0.3, 440+aMod1+aMod2, 1
outs aSig, aSig
endin


</CsInstruments>
<CsScore>
f 1 0 1024 10 1   ;Sine wave for table 1
i 1 0 3
</CsScore>
</CsoundSynthesizer>
; written by Alex Hofmann (Mar. 2011)
```

In series: M1->M2->C

This is much more complicated to calculate and sound-timbre becomes harder to predict, because M1:M2 produces a complex spectrum (W), which then modulates the carrier (W:C).

Example 04D08.csd

```
<CsoundSynthesizer>
<CsOptions>
-o dac
</CsOptions>
<CsInstruments>
sr = 48000
ksmps = 32
nchnls = 2
0dbfs = 1

instr 1
aMod1 poscil 200, 700, 1
aMod2 poscil 1800, 290+aMod1, 1
aSig poscil 0.3, 440+aMod2, 1
outs aSig, aSig
endin

</CsInstruments>
<CsScore>
f 1 0 1024 10 1   ;Sine wave for table 1
i 1 0 3
</CsScore>
</CsoundSynthesizer>
; written by Alex Hofmann (Mar. 2011)
```

PHASE MODULATION - THE YAMAHA DX7 AND FEEDBACK FM

There is a strong relation between frequency modulation and phase modulation, as both techniques influence the oscillator's pitch, and the resulting timbre modifications are the same.

If you'd like to build a feedbacking FM system, it will happen that the self-modulation comes to a zero point, which stops the oscillator forever. To avoid this, it is more practical to modulate the carriers table-lookup phase, instead of its pitch.

Even the most famous FM-synthesizer Yamaha DX7 is based on the phase-modulation (PM) technique, because this allows feedback. The DX7 provides 7 operators, and offers 32 routing combinations of these. (http://yala.freeservers.com/t2synths.htm#DX7)

To build a PM-synth in Csound *tablei* opcode needs to be used as oscillator. In order to step through the f-table, a *phasor* will output the necessary steps.

Example 04D09.csd

```
<CsoundSynthesizer>
<CsOptions>
-o dac
</CsOptions>
<CsInstruments>
sr = 48000
ksmps = 32
nchnls = 2
0dbfs = 1

instr 1  ; simple PM-Synth
kCarFreq = 200
kModFreq = 280
kModFactor = kCarFreq/kModFreq
kIndex = 12/6.28   ;  12/2pi to convert from radians to norm. table index
aEnv expseg .001, 0.001, 1, 0.3, 0.5, 8.5, .001
aModulator poscil kIndex*aEnv, kModFreq, 1
aPhase phasor kCarFreq
aCarrier tablei aPhase+aModulator, 1, 1, 0, 1
outs (aCarrier*aEnv), (aCarrier*aEnv)
endin

</CsInstruments>
<CsScore>
f 1 0 1024 10 1   ;Sine wave for table 1
i 1 0 9
</CsScore>
</CsoundSynthesizer>
; written by Alex Hofmann (Mar. 2011)
```

Let's use the possibilities of self-modulation (feedback-modulation) of the oscillator. So in the following example, the oscillator is both *modulator* and *carrier*. To control the amount of modulation, an envelope scales the feedback.

Example 04D10.csd

```
<CsoundSynthesizer>
<CsOptions>
-o dac
</CsOptions>
<CsInstruments>
sr = 48000
ksmps = 32
nchnls = 2
0dbfs = 1

instr 1  ; feedback PM
kCarFreq = 200
kFeedbackAmountEnv linseg 0, 2, 0.2, 0.1, 0.3, 0.8, 0.2, 1.5, 0
aAmpEnv expseg .001, 0.001, 1, 0.3, 0.5, 8.5, .001
aPhase phasor kCarFreq
aCarrier init 0 ; init for feedback
aCarrier tablei aPhase+(aCarrier*kFeedbackAmountEnv), 1, 1, 0, 1
outs aCarrier*aAmpEnv, aCarrier*aAmpEnv
endin

</CsInstruments>
<CsScore>
f 1 0 1024 10 1   ;Sine wave for table 1
i 1 0 9
</CsScore>
</CsoundSynthesizer>
; written by Alex Hofmann (Mar. 2011)
```

The last example features modulation of the buzz opcode. The buzz opcode can have a lot of harmonic overtones and frequency modulation of the buzz opcode gives even more overtones. Four different voices play at the same time, forming strange chords that use glissando/portamento to move from one chord to the next. This .csd file is regenerative, everytime you run it, it should show a different performance.

EXAMPLE 04D11.csd

```
<CsoundSynthesizer>
<CsOptions>
-odac
</CsOptions>
<CsInstruments>

; By Bjørn Houdorf, April 2012

sr = 44100
ksmps = 8
nchnls = 2
0dbfs = 1



; Global initializations ("Instrument 0")

          seed      0;  New pitches,

gkfreq1   init      0;  every time you

gkfreq2   init      0;  run this file

gkfreq3   init      0

gkfreq4   init      0



gimidia1  init      60; The 4 voices start

gimidib1  init      60; at different

gimidia2  init      64; MIDI frequencies

gimidib2  init      64

gimidia3  init      67

gimidib3  init      67

gimidia4  init      70

gimidib4  init      70



; Function Table

giFt1     ftgen     0, 0, 16384, 10, 1; Sine wave



instr 1; Master control pitch for instrument 2



test:

idurtest  poisson   20; Duration of each test loop

          timout    0, idurtest, execute



          reinit    test



execute:

gimidia1  =         gimidib1

ital1     random    -4, 4

gimidib1  =         gimidib1 + ital1

gimidia2  =         gimidib2
```

```
ital2     random    -4, 4
gimidib2  =         gimidib2 + ital2
gimidia3  =         gimidib3
ital3     random    -4, 4
gimidib3  =         gimidib3 + ital3
gimidia4  =         gimidib4
ital4     random    -4, 4
gimidib4  =         gimidib4 + ital4

idiv      poisson   4

idurx     =         0.01; Micro end segment to create
            ;a held final frequency value

ifreq1a   =         cpsmidinn(gimidia1)
ifreq1b   =         cpsmidinn(gimidib1)
; Portamento frequency ramp:
gkfreq1   linseg    ifreq1a, idurtest/idiv, ifreq1b, idurx, ifreq1b
ifreq2a   =         cpsmidinn(gimidia2)
ifreq2b   =         cpsmidinn(gimidib2)
gkfreq2   linseg    ifreq2a, idurtest/idiv, ifreq2b, idurx, ifreq2b
ifreq3a   =         cpsmidinn(gimidia3)
ifreq3b   =         cpsmidinn(gimidib3)
gkfreq3   linseg    ifreq3a, idurtest/idiv, ifreq3b, idurx, ifreq3b
ifreq4a   =         cpsmidinn(gimidia4)
ifreq4b   =         cpsmidinn(gimidib4)
gkfreq4   linseg    ifreq4a, idurtest/idiv, ifreq4b, idurx, ifreq4b
endin

instr 2 ; Oscillators
iamp      =         p4
irise     =         p5
idur      =         p3
idec      =         p6
kamp      =         p7
imodfrq   =         p8
iharm     =         p9 ; Number of harmonics

ky        linen     iamp, irise, idur, idec

kampfreq  =         2
kampa     oscili    kamp, kampfreq, giFt1
```

```
; Different phase for the 4 voices
klfo1     oscili    kampa, imodfrq, giFt1, 0
klfo2     oscili    kampa, imodfrq, giFt1, 0.25
klfo3     oscili    kampa, imodfrq, giFt1, 0.50
klfo4     oscili    kampa, imodfrq, giFt1, 0.75

kzfrq     =         0.1; Velocity of amplitude oscillation
kampvoice =         0.5; Amplitude of each voice

; Amplitude between -0.5 and 0.5
kx1       oscili    0.5, kzfrq, giFt1, 0
kx2       oscili    0.5, kzfrq, giFt1, 0.25
kx3       oscili    0.5, kzfrq, giFt1, 0.50
kx4       oscili    0.5, kzfrq, giFt1, 0.75

; Add 0.5 so amplitude oscillates between 0 and 1
k1        =         kx1+0.5
k2        =         kx2+0.5
k3        =         kx3+0.5
k4        =         kx4+0.5

; Minimize interference between chorus oscillators
itilf     random    -5, 5

asig11    buzz      ky*k1, (2.02*gkfreq1)+itilf+klfo1, iharm, giFt1
asig12    buzz      ky*k1, gkfreq1 +klfo1, iharm, giFt1; Voice 1
asig13    buzz      ky*k1, (1.51*gkfreq1)+itilf+klfo1, iharm, giFt1
aa1       =         asig11+asig12+asig13

asig21    buzz      ky*k2, (2.01*gkfreq2)+itilf+klfo2, iharm, giFt1
asig22    buzz      ky*k2, gkfreq2 +klfo2, iharm, giFt1; Voice 2
asig23    buzz      ky*k2, (1.51*gkfreq2)+itilf+klfo2, iharm, giFt1
aa2       =         asig21+asig22+asig23

asig31    buzz      ky*k3, (2.01*gkfreq3)+itilf+klfo3, iharm, giFt1
asig32    buzz      ky*k3, gkfreq3 +klfo3, iharm, giFt1; Voice 3
asig33    buzz      ky*k3, (1.51*gkfreq3)+itilf+klfo3, iharm, giFt1
aa3       =         asig31+asig32+asig33

asig41    buzz      ky*k4, (2.01*gkfreq4)+itilf+klfo4, iharm, giFt1
asig42    buzz      ky*k4, gkfreq4 +klfo4, iharm, giFt1; Voice 4
asig43    buzz      ky*k4, (1.51*gkfreq4)+itilf+klfo4, iharm, giFt1
aa4       =         asig41+asig42+asig43
```

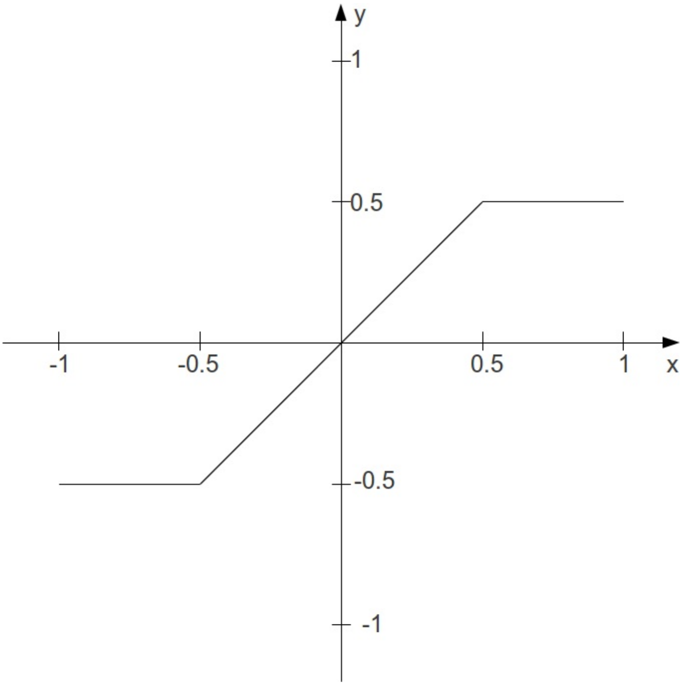
```
          outs      aa1+aa3, aa2+aa4

endin
</CsInstruments>
<CsScore>

; Master control instrument
; Inst start dur
i1       0   3600

; Oscillators
; inst start idur iamp irise idec kamp imodfrq iharm
i2      0    3600  0.3   4    20  0.10    7     16

</CsScore>
</CsoundSynthesizer>
```

24. WAVESHAPING

Waveshaping can in some ways be thought of as a relation to modulation techniques such as frequency or phase modulation. Waveshaping can achieve quite dramatic sound tranformations through the application of a very simple process. In FM (frequency modulation) synthesis modulation occurs between two oscillators, waveshaping is implemented using a single oscillator (usually a simple sine oscillator) and a so-called 'transfer function'. The transfer function transforms and shapes the incoming amplitude values using a simple lookup process: if the incoming value is x, the outgoing value becomes y. This can be written as a table with two columns. Here is a simple example:

Incoming (x) Value	Outgoing (y) Value
-0.5 or lower	-1
between -0.5 and 0.5	remain unchanged
0.5 or higher	1

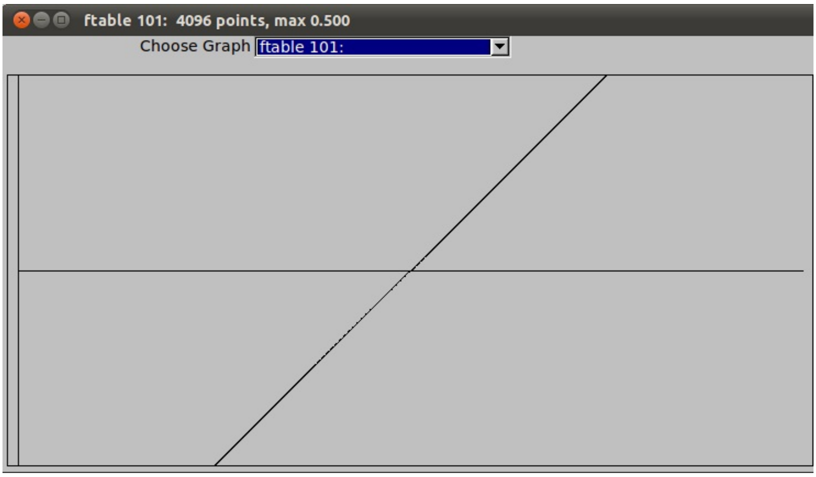
Illustrating this in an x/y coordinate system results in the following image:



BASIC IMPLEMENTATION MODEL

Implementing this as Csound code is pretty straightforward. The x-axis is the amplitude of every single sample, which is in the range of -1 to +1.[1] This number has to be used as index to a table which stores the transfer function. To create a table like the one above, you can use Csound's sub-routine GEN07[2] . This statement will create a table of 4096 points in the desired shape:

```
giTrnsFnc ftgen 0, 0, 4096, -7, -0.5, 1024, -0.5, 2048, 0.5, 1024, 0.5
```



Now, two problems must be solved. First, the index of the function table is not -1 to +1. Rather, it is either 0 to 4095 in the raw index mode, or 0 to 1 in the normalized mode. The simplest solution is to use the normalized index and scale the incoming amplitudes, so that an amplitude of -1 becomes an index of 0, and an amplitude of 1 becomes an index of 1:

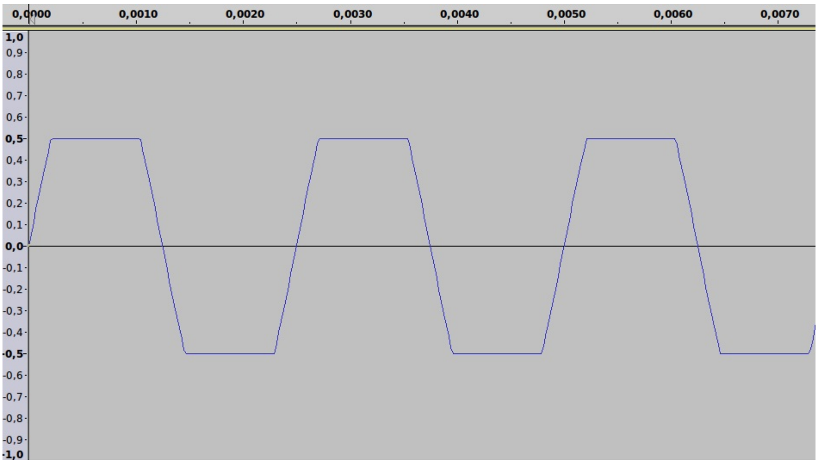
```
aIndx = (aAmp + 1) / 2
```

The other problem stems from the difference in the accuracy of possible values in a sample and in a function table. Every single sample is encoded in a 32-bit floating point number in standard audio applications - or even in a 64-bit float in recent Csound.[3] A table with 4096 points results in a 12-bit number, so you will have a serious loss of accuracy (= sound quality) if you use the table values directly.[4] Here, the solution is to use an interpolating table reader. The opcode tablei (instead of table) does this job. This opcode then needs an extra point in the table for interpolating, so it is wise to use 4097 as size instead of 4096.[5]

This is the code for the simple waveshaping with the transfer function which has been discussed so far:

EXAMPLE 04E01.csd

```
<CsoundSynthesizer>
<CsOptions>
-odac
</CsOptions>
<CsInstruments>
sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

giTrnsFnc ftgen 0, 0, 4097, -7, -0.5, 1024, -0.5, 2048, 0.5, 1024, 0.5
giSine    ftgen 0, 0, 1024, 10, 1

instr 1
aAmp      poscil    1, 400, giSine
aIndx     =         (aAmp + 1) / 2
aWavShp   tablei    aIndx, giTrnsFnc, 1
          outs      aWavShp, aWavShp
endin

</CsInstruments>
<CsScore>
i 1 0 10
</CsScore>
</CsoundSynthesizer>
```



CHEBYCHEV POLYNOMIALS AS TRANSFER FUNCTIONS

1. Use the statement 0dbfs=1 in the orchestra header to ensure this.^
2. See chapter 03D:FUNCTION TABLES to find more information about creating tables.^
3. This is the 'd' in some abbreviations like Csound5.17-gnu-win32-d.exe (d = double precision floats).^
4. Of course you can use an even smaller table if your goal is the degradation of the incoming sound ("distortion"). See chapter 05F for some examples.^
5. A table size of a power-of-two plus one inserts the "extended guard point" as an extension of the last table value, instead of copying the first index to this location. See http://www.csounds.com/manual/html/f.html for more information.^

25. GRANULAR SYNTHESIS

CONCEPT BEHIND GRANULAR SYNTHESIS

Granular synthesis is a technique in which a source sound or waveform is broken into many fragments, often of very short duration, which are then restructured and rearranged according to various patterning and indeterminacy functions.

If we imagine the simplest possible granular synthesis algorithm in which a precise fragment of sound is repeated with regularity, there are two principle attributes of this process that we are most concerned with. Firstly the duration of each sound grain is significant: if the grain duration if very small, typically less than 0.02 seconds, then less of the characteristics of the source sound will be evident. If the grain duration is greater than 0.02 then more of the character of the source sound or waveform will be evident. Secondly the rate at which grains are generated will be significant: if grain generation is below 20 hertz, i.e. less than 20 grains per second, then the stream of grains will be perceived as a rhythmic pulsation; if rate of grain generation increases beyond 20 Hz then individual grains will be harder to distinguish and instead we will begin to perceive a buzzing tone, the fundamental of which will correspond to the frequency of grain generation. Any pitch contained within the source material is not normally perceived as the fundamental of the tone whenever grain generation is periodic, instead the pitch of the source material or waveform will be perceived as a resonance peak (sometimes referred to as a formant); therefore transposition of the source material will result in the shifting of this resonance peak.

GRANULAR SYNTHESIS DEMONSTRATED USING FIRST PRINCIPLES

The following example exemplifies the concepts discussed above. None of Csound's built-in granular synthesis opcodes are used, instead schedkwhen in instrument 1 is used to precisely control the triggering of grains in instrument 2. Three notes in instrument 1 are called from the score one after the other which in turn generate three streams of grains in instrument 2. The first note demonstrates the transition from pulsation to the perception of a tone as the rate of grain generation extends beyond 20 Hz. The second note demonstrates the loss of influence of the source material as the grain duration is reduced below 0.02 seconds. The third note demonstrates how shifting the pitch of the source material for the grains results in the shifting of a resonance peak in the output tone. In each case information regarding rate of grain generation, duration and fundamental (source material pitch) is output to the terminal every 1/2 second so that the user can observe the changing parameters.

It should also be noted how the amplitude of each grain is enveloped in instrument 2. If grains were left unenveloped they would likely produce clicks on account of discontinuities in the waveform produced at the beginning and ending of each grain.

Granular synthesis in which grain generation occurs with perceivable periodicity is referred to as synchronous granular synthesis. granular synthesis in which this periodicity is not evident is referred to as asynchronous granular synthesis.

EXAMPLE 04F01.csd

```
<CsoundSynthesizer>

<CsOptions>
-odac -m0
</CsOptions>

<CsInstruments>
;Example by Iain McCurdy

sr = 44100
ksmps = 1
nchnls = 1
0dbfs = 1

giSine  ftgen  0,0,4096,10,1
```

```
instr 1
  kRate  expon  p4,p3,p5   ; rate of grain generation
  kTrig  metro  kRate      ; a trigger to generate grains
  kDur   expon  p6,p3,p7   ; grain duration
  kForm  expon  p8,p3,p9   ; formant (spectral centroid)
   ;                      p1 p2 p3   p4
  schedkwhen    kTrig,0,0,2, 0, kDur,kForm ;trigger a note(grain) in instr 2
  ;print data to terminal every 1/2 second
  printks "Rate:%5.2F  Dur:%5.2F  Formant:%5.2F%n", 0.5, kRate , kDur, kForm
endin

instr 2
  iForm =       p4
  aEnv  linseg  0,0.005,0.2,p3-0.01,0.2,0.005,0
  aSig  poscil  aEnv, iForm, giSine
        out     aSig
endin

</CsInstruments>

<CsScore>
;p4 = rate begin
;p5 = rate end
;p6 = duration begin
;p7 = duration end
;p8 = formant begin
;p9 = formant end
; p1 p2 p3 p4 p5  p6   p7    p8  p9
i 1  0  30 1  100 0.02 0.02  400 400  ;demo of grain generation rate
i 1  31 10 10 10  0.4  0.01  400 400  ;demo of grain size
i 1  42 20 50 50  0.02 0.02  100 5000 ;demo of changing formant
e
</CsScore>

</CsoundSynthesizer>
```

GRANULAR SYNTHESIS OF VOWELS: FOF

The principles outlined in the previous example can be extended to imitate vowel sounds produced by the human voice. This type of granular synthesis is referred to as FOF (fonction d'onde formatique) synthesis and is based on work by Xavier Rodet on his CHANT program at IRCAM. Typically five synchronous granular synthesis streams will be used to create five different resonant peaks in a fundamental tone in order to imitate different vowel sounds expressible by the human voice. The most crucial element in defining a vowel imitation is the degree to which the source material within each of the five grain streams is transposed. Bandwidth (essentially grain duration) and intensity (loudness) of each grain stream are also important indicators in defining the resultant sound.

Csound has a number of opcodes that make working with FOF synthesis easier. We will be using fof.

Information regarding frequency, bandwidth and intensity values that will produce various vowel sounds for different voice types can be found in the appendix of the Csound manual here. These values are stored in function tables in the FOF synthesis example. GEN07, which produces linear break point envelopes, is chosen as we will then be able to morph continuously between vowels.

EXAMPLE 04F02.csd

```
<CsoundSynthesizer>

<CsOptions>
-odac
</CsOptions>

<CsInstruments>
;example by Iain McCurdy

sr = 44100
ksmps = 16
nchnls = 2
0dbfs = 1

instr 1
  kFund    expon     p4,p3,p5               ; fundemental
  kVow     line      p6,p3,p7               ; vowel select
  kBW      line      p8,p3,p9               ; bandwidth factor
  iVoice   =         p10                    ; voice select
  ; read formant cutoff frequenies from tables
  kForm1     table     kVow,1+(iVoice*15),1
  kForm2     table     kVow,2+(iVoice*15),1
  kForm3     table     kVow,3+(iVoice*15),1
  kForm4     table     kVow,4+(iVoice*15),1
  kForm5     table     kVow,5+(iVoice*15),1
  ; read formant intensity values from tables
  kDB1     table     kVow,6+(iVoice*15),1
  kDB2     table     kVow,7+(iVoice*15),1
  kDB3     table     kVow,8+(iVoice*15),1
  kDB4     table     kVow,9+(iVoice*15),1
  kDB5     table     kVow,10+(iVoice*15),1
  ; read formant bandwidths from tables
  kBW1     table     kVow,11+(iVoice*15),1
  kBW2     table     kVow,12+(iVoice*15),1
  kBW3     table     kVow,13+(iVoice*15),1
  kBW4     table     kVow,14+(iVoice*15),1
  kBW5     table     kVow,15+(iVoice*15),1
  ; create resonant formants by filtering source sound
  koct     =         1
  aForm1   fof       ampdb(kDB1),kFund,kForm1,0,kBW1,0.003,0.02,0.007,\
                       1000,101,102,3600
  aForm2   fof       ampdb(kDB2),kFund,kForm2,0,kBW2,0.003,0.02,0.007,\
                       1000,101,102,3600
  aForm3   fof       ampdb(kDB3),kFund,kForm3,0,kBW3,0.003,0.02,0.007,\
                       1000,101,102,3600
  aForm4   fof       ampdb(kDB4),kFund,kForm4,0,kBW4,0.003,0.02,0.007,\
                       1000,101,102,3600
  aForm5   fof       ampdb(kDB5),kFund,kForm5,0,kBW5,0.003,0.02,0.007,\
                       1000,101,102,3600

  ; formants are mixed
  aMix     sum       aForm1,aForm2,aForm3,aForm4,aForm5
  kEnv     linseg    0,3,1,p3-6,1,3,0     ; an amplitude envelope
           outs      aMix*kEnv*0.3, aMix*kEnv*0.3 ; send audio to outputs
endin

</CsInstruments>

<CsScore>
;FUNCTION TABLES STORING FORMANT DATA FOR EACH OF THE FIVE VOICE TYPES
REPRESENTED
;BASS
f 1  0 32768 -7 600 10922 400 10922 250 10924 350 ;FREQ
f 2  0 32768 -7 1040 10922 1620 10922 1750 10924 600 ;FREQ
f 3  0 32768 -7 2250 10922 2400 10922 2600 10924 2400 ;FREQ
f 4  0 32768 -7 2450 10922 2800 10922 3050 10924 2675 ;FREQ
f 5  0 32768 -7 2750 10922 3100 10922 3340 10924 2950 ;FREQ
f 6  0 32768 -7 0 10922 0 10922 0 10924 0 ;dB
f 7  0 32768 -7 -7 10922 -12 10922 -30 10924 -20 ;dB
f 8  0 32768 -7 -9 10922 -9 10922 -16 10924 -32 ;dB
f 9  0 32768 -7 -9 10922 -12 10922 -22 10924 -28 ;dB
f 10 0 32768 -7 -20 10922 -18 10922 -28 10924 -36 ;dB
f 11 0 32768 -7 60 10922 40 10922 60 10924 40 ;BAND WIDTH
f 12 0 32768 -7 70 10922 80 10922 90 10924 80 ;BAND WIDTH
f 13 0 32768 -7 110 10922 100 10922 100 10924 100 ;BAND WIDTH
f 14 0 32768 -7 120 10922 120 10922 120 10924 120 ;BAND WIDTH
f 15 0 32768 -7 130 10922 120 10922 120 10924 120 ;BAND WIDTH
;TENOR
f 16 0 32768 -7 650  8192 400  8192 290 8192 400 8192 350 ;FREQ
f 17 0 32768 -7 1080  8192 1700    8192 1870 8192 800 8192 600 ;FREQ
f 18 0 32768 -7 2650 8192 2600    8192 2800 8192 2600 8192 2700 ;FREQ
f 19 0 32768 -7 2900 8192 3200    8192 3250 8192 2800 8192 2900 ;FREQ
f 20 0 32768 -7 3250 8192 3580    8192 3540 8192 3000 8192 3300 ;FREQ
f 21 0 32768 -7 0 8192 0 8192 0 8192 0 8192 0 ;dB
f 22 0 32768 -7 -6 8192 -14 8192 -15 8192 -10 8192 -20 ;dB
f 23 0 32768 -7 -7 8192 -12 8192 -18 8192 -12 8192 -17 ;dB
f 24 0 32768 -7 -8 8192 -14 8192 -20 8192 -12 8192 -14 ;dB
f 25 0 32768 -7 -22 8192 -20 8192 -30 8192 -26 8192 -26 ;dB
f 26 0 32768 -7 80 8192 70 8192 40 8192 40 8192 40 ;BAND WIDTH
f 27 0 32768 -7 90 8192 80 8192 90 8192 80 8192 60 ;BAND WIDTH
f 28 0 32768 -7 120 8192 100 8192 100 8192 100 8192 100 ;BAND WIDTH
f 29 0 32768 -7 130 8192 120 8192 120 8192 120 8192 120 ;BAND WIDTH
f 30 0 32768 -7 140 8192 120 8192 120 8192 120 8192 120 ;BAND WIDTH
;COUNTER TENOR
f 31 0 32768 -7 660 8192 440 8192 270 8192 430 8192 370 ;FREQ
f 32 0 32768 -7 1120 8192 1800 8192 1850 8192 820 8192 630 ;FREQ
f 33 0 32768 -7 2750 8192 2700 8192 2900 8192 2700 8192 2750 ;FREQ
f 34 0 32768 -7 3000 8192 3000 8192 3350 8192 3000 8192 3000 ;FREQ
f 35 0 32768 -7 3350 8192 3300 8192 3590 8192 3300 8192 3400 ;FREQ
f 36 0 32768 -7 0 8192 0 8192 0 8192 0 8192 0 ;dB
f 37 0 32768 -7 -6 8192 -14 8192 -24 8192 -10 8192 -20 ;dB
f 38 0 32768 -7 -23 8192 -18 8192 -24 8192 -26 8192 -23 ;dB
f 39 0 32768 -7 -24 8192 -20 8192 -36 8192 -22 8192 -30 ;dB
f 40 0 32768 -7 -38 8192 -20 8192 -36 8192 -34 8192 -30 ;dB
```

```
f 41 0 32768 -7 80 8192 70 8192 40 8192 40 8192 40 ;BAND WIDTH
f 42 0 32768 -7 90 8192 80 8192 90 8192 80 8192 60 ;BAND WIDTH
f 43 0 32768 -7 120 8192 100 8192 100 8192 100 8192 100 ;BAND WIDTH
f 44 0 32768 -7 130 8192 120 8192 120 8192 120 8192 120 ;BAND WIDTH
f 45 0 32768 -7 140 8192 120 8192 120 8192 120 8192 120 ;BAND WIDTH
;ALTO
f 46 0 32768 -7 800 8192 400 8192 350 8192 450 8192 325 ;FREQ
f 47 0 32768 -7 1150 8192 1600 8192 1700 8192 800 8192 700 ;FREQ
f 48 0 32768 -7 2800 8192 2700 8192 2700 8192 2830 8192 2530 ;FREQ
f 49 0 32768 -7 3500 8192 3300 8192 3700 8192 3500 8192 2500 ;FREQ
f 50 0 32768 -7 4950 8192 4950 8192 4950 8192 4950 8192 4950 ;FREQ
f 51 0 32768 -7 0 8192 0 8192 0 8192 0 8192 0 ;dB
f 52 0 32768 -7 -4 8192 -24 8192 -20 8192 -9 8192 -12 ;dB
f 53 0 32768 -7 -20 8192 -30 8192 -30 8192 -16 8192 -30 ;dB
f 54 0 32768 -7 -36 8192 -35 8192 -36 8192 -28 8192 -40 ;dB
f 55 0 32768 -7 -60 8192 -60 8192 -60 8192 -55 8192 -64 ;dB
f 56 0 32768 -7 50 8192 60 8192 50 8192 70 8192 50 ;BAND WIDTH
f 57 0 32768 -7 60 8192 80 8192 100 8192 80 8192 60 ;BAND WIDTH
f 58 0 32768 -7 170 8192 120 8192 120 8192 100 8192 170 ;BAND WIDTH
f 59 0 32768 -7 180 8192 150 8192 150 8192 130 8192 180 ;BAND WIDTH
f 60 0 32768 -7 200 8192 200 8192 200 8192 135 8192 200 ;BAND WIDTH
;SOPRANO
f 61 0 32768 -7 800 8192 350 8192 270 8192 450 8192 325 ;FREQ
f 62 0 32768 -7 1150 8192 2000 8192 2140 8192 800 8192 700 ;FREQ
f 63 0 32768 -7 2900 8192 2800 8192 2950 8192 2830 8192 2700 ;FREQ
f 64 0 32768 -7 3900 8192 3600 8192 3900 8192 3800 8192 3800 ;FREQ
f 65 0 32768 -7 4950 8192 4950 8192 4950 8192 4950 8192 4950 ;FREQ
f 66 0 32768 -7 0 8192 0 8192 0 8192 0 8192 0 ;dB
f 67 0 32768 -7 -6 8192 -20 8192 -12 8192 -11 8192 -16 ;dB
f 68 0 32768 -7 -32 8192 -15 8192 -26 8192 -22 8192 -35 ;dB
f 69 0 32768 -7 -20 8192 -40 8192 -26 8192 -22 8192 -40 ;dB
f 70 0 32768 -7 -50 8192 -56 8192 -44 8192 -50 8192 -60 ;dB
f 71 0 32768 -7 80 8192 60 8192 60 8192 70 8192 50 ;BAND WIDTH
f 72 0 32768 -7 90 8192 90 8192 90 8192 80 8192 60 ;BAND WIDTH
f 73 0 32768 -7 120 8192 100 8192 100 8192 100 8192 170 ;BAND WIDTH
f 74 0 32768 -7 130 8192 150 8192 120 8192 130 8192 180 ;BAND WIDTH
f 75 0 32768 -7 140 8192 200 8192 120 8192 135 8192 200 ;BAND WIDTH

f 101 0 4096 10 1   ;SINE WAVE
;EXPONENTIAL CURVE USED TO DEFINE THE ENVELOPE SHAPE OF FOF PULSES:
f 102 0 1024 19 0.5 0.5 270 0.5
; p4 = fundamental begin value (c.p.s.)
; p5 = fundamental end value
; p6 = vowel begin value (0 - 1 : a e i o u)
; p7 = vowel end value
; p8 = bandwidth factor begin (suggested range 0 - 2)
; p9 = bandwidth factor end
; p10 = voice (0=bass; 1=tenor; 2=counter_tenor; 3=alto; 4=soprano)

; p1 p2  p3  p4  p5  p6  p7  p8  p9 p10
i 1  0   10  50  100 0   1   2   0  0
i 1  8   .   78  77  1   0   1   0  1
i 1  16  .   150 118 0   1   1   0  2
i 1  24  .   200 220 1   0   0.2 0  3
i 1  32  .   400 800 0   1   0.2 0  4
e
</CsScore>
</CsoundSynthesizer>
```

ASYNCHRONOUS GRANULAR SYNTHESIS

The previous two examples have played psychoacoustic phenomena associated with the perception of granular textures that exhibit periodicity and patterns. If we introduce indeterminacy into some of the parameters of granular synthesis we begin to lose the coherence of some of these harmonic structures.

The next example is based on the design of example 04F01.csd. Two streams of grains are generated. The first stream begins as a synchronous stream but as the note progresses the periodicity of grain generation is eroded through the addition of an increasing degree of gaussian noise. It will be heard how the tone metamorphosizes from one characterized by steady purity to one of fuzzy airiness. The second the applies a similar process of increasing indeterminacy to the formant parameter (frequency of material within each grain).

Other parameters of granular synthesis such as the amplitude of each grain, grain duration, spatial location etc. can be similarly modulated with random functions to offset the psychoacoustic effects of synchronicity when using constant values.

EXAMPLE 04F03.csd

```
<CsoundSynthesizer>

<CsOptions>
-odac
</CsOptions>

<CsInstruments>
;Example by Iain McCurdy

sr = 44100
ksmps = 1
nchnls = 1
0dbfs = 1

giWave  ftgen  0,0,2^10,10,1,1/2,1/4,1/8,1/16,1/32,1/64

instr 1 ;grain generating instrument 1
  kRate         =          p4
  kTrig         metro      kRate      ; a trigger to generate grains
  kDur          =          p5
  kForm         =          p6
  ;note delay time (p2) is defined using a random function -
  ;- beginning with no randomization but then gradually increasing
  kDelayRange   transeg    0,1,0,0,  p3-1,4,0.03
  kDelay        gauss      kDelayRange
  ;                                  p1 p2 p3   p4
                schedkwhen kTrig,0,0,3, abs(kDelay), kDur,kForm ;trigger a note
(grain) in instr 3
endin

instr 2 ;grain generating instrument 2
  kRate          =          p4
  kTrig          metro      kRate      ; a trigger to generate grains
  kDur           =          p5
  ;formant frequency (p4) is multiplied by a random function -
  ;- beginning with no randomization but then gradually increasing
  kForm          =          p6
  kFormOSRange  transeg     0,1,0,0,  p3-1,2,12 ;range defined in semitones
  kFormOS       gauss       kFormOSRange
  ;                                    p1 p2 p3   p4
                schedkwhen  kTrig,0,0,3, 0, kDur,kForm*semitone(kFormOS)
endin

instr 3 ;grain sounding instrument
  iForm =       p4
  aEnv  linseg  0,0.005,0.2,p3-0.01,0.2,0.005,0
  aSig  poscil  aEnv, iForm, giWave
        out     aSig
endin

</CsInstruments>

<CsScore>
;p4 = rate
;p5 = duration
;p6 = formant
; p1 p2   p3 p4  p5   p6
i 1  0    12 200 0.02 400
i 2  12.5 12 200 0.02 400
e
</CsScore>

</CsoundSynthesizer>
```

SYNTHESIS OF DYNAMIC SOUND SPECTRA: GRAIN3

The next example introduces another of Csound's built-in granular synthesis opcodes to demonstrate the range of dynamic sound spectra that are possible with granular synthesis.

Several parameters are modulated slowly using Csound's random spline generator rspline. These parameters are formant frequency, grain duration and grain density (rate of grain generation). The waveform used in generating the content for each grain is randomly chosen using a slow sample and hold random function - a new waveform will be selected every 10 seconds. Five waveforms are provided: a sawtooth, a square wave, a triangle wave, a pulse wave and a band limited buzz-like waveform. Some of these waveforms, particularly the sawtooth, square and pulse waveforms, can generate very high overtones, for this reason a high sample rate is recommended to reduce the risk of aliasing (see chapter 01A).

Current values for formant (cps), grain duration, density and waveform are printed to the terminal every second. The key for waveforms is: 1:sawtooth; 2:square; 3:triangle; 4:pulse;

5:buzz.

EXAMPLE 04F04.csd

```
<CsoundSynthesizer>

<CsOptions>
-odac
</CsOptions>

<CsInstruments>
;example by Iain McCurdy

sr = 96000
ksmps = 16
nchnls = 1
0dbfs = 1

;waveforms used for granulation
giSaw   ftgen 1,0,4096,7,0,4096,1
giSq    ftgen 2,0,4096,7,0,2046,0,0,1,2046,1
giTri   ftgen 3,0,4096,7,0,2046,1,2046,0
giPls   ftgen 4,0,4096,7,1,200,1,0,0,4096-200,0
giBuzz  ftgen 5,0,4096,11,20,1,1

;window function - used as an amplitude envelope for each grain
;(hanning window)
giWFn   ftgen 7,0,16384,20,2,1

instr 1
  ;random spline generates formant values in oct format
  kOct    rspline 4,8,0.1,0.5
  ;oct format values converted to cps format
  kCPS    =       cpsoct(kOct)
  ;phase location is left at 0 (the beginning of the waveform)
  kPhs    =       0
  ;frequency (formant) randomization and phase randomization are not used
  kFmd    =       0
  kPmd    =       0
  ;grain duration and density (rate of grain generation)
  kGDur   rspline 0.01,0.2,0.05,0.2
  kDens   rspline 10,200,0.05,0.5
  ;maximum number of grain overlaps allowed. This is used as a CPU brake
  iMaxOvr =       1000
  ;function table for source waveform for content of the grain
  ;a different waveform chosen once every 10 seconds
  kFn     randomh 1,5.99,0.1
  ;print info. to the terminal
          printks "CPS:%5.2F%TDur:%5.2F%TDensity:%5.2F%TWaveform:%1.0F%n",1,\
                     kCPS,kGDur,kDens,kFn
  aSig    grain3  kCPS, kPhs, kFmd, kPmd, kGDur, kDens, iMaxOvr, kFn, giWFn, \
                    0, 0
          out     aSig*0.06
endin

</CsInstruments>

<CsScore>
i 1 0 300
e
</CsScore>

</CsoundSynthesizer>
```

The final example introduces grain3's two built-in randomizing functions for phase and pitch. Phase refers to the location in the source waveform from which a grain will be read, pitch refers to the pitch of the material within grains. In this example a long note is played, initially no randomization is employed but gradually phase randomization is increased and then reduced back to zero. The same process is applied to the pitch randomization amount parameter. This time grain size is relatively large:0.8 seconds and density correspondingly low: 20 Hz.

EXAMPLE 04F05.csd

```
<CsoundSynthesizer>

<CsOptions>
-odac
</CsOptions>

<CsInstruments>
```

```
;example by Iain McCurdy

sr = 44100
ksmps = 16
nchnls = 1
0dbfs = 1

;waveforms used for granulation
giBuzz  ftgen 1,0,4096,11,40,1,0.9

;window function - used as an amplitude envelope for each grain
;(bartlett window)
giWFn   ftgen 2,0,16384,20,3,1

instr 1
  kCPS    =       100
  kPhs    =       0
  kFmd    transeg 0,21,0,0, 10,4,15, 10,-4,0
  kPmd    transeg 0,1,0,0,  10,4,1,  10,-4,0
  kGDur   =       0.8
  kDens   =       20
  iMaxOvr =       1000
  kFn     =       1
  ;print info. to the terminal
          printks "Random Phase:%5.2F%TPitch Random:%5.2F%n",1,kPmd,kFmd
  aSig    grain3  kCPS, kPhs, kFmd, kPmd, kGDur, kDens, iMaxOvr, kFn, giWFn, 0, 0
          out     aSig*0.06
endin

</CsInstruments>

<CsScore>
i 1 0 51
e
</CsScore>

</CsoundSynthesizer>
```

CONCLUSION

This chapter has introduced some of the concepts behind the synthesis of new sounds based on simple waveforms by using granular synthesis techniques. Only two of Csound's built-in opcodes for granular synthesis, fof and grain3, have been used; it is beyond the scope of this work to cover all of the many opcodes for granulation that Csound provides. This chapter has focused mainly on synchronous granular synthesis; chapter 05G, which introduces granulation of recorded sound files, makes greater use of asynchronous granular synthesis for time-stretching and pitch shifting. This chapter will also introduce some of Csound's other opcodes for granular synthesis.

26. PHYSICAL MODELLING

With physical modelling we employ a completely different approach to synthesis than we do with all other standard techniques. Unusually the focus is not primarily to produce a sound, but to model a physical process and if this process exhibits certain features such as periodic oscillation within a frequency range of 20 to 20000 Hz, it will produce sound.

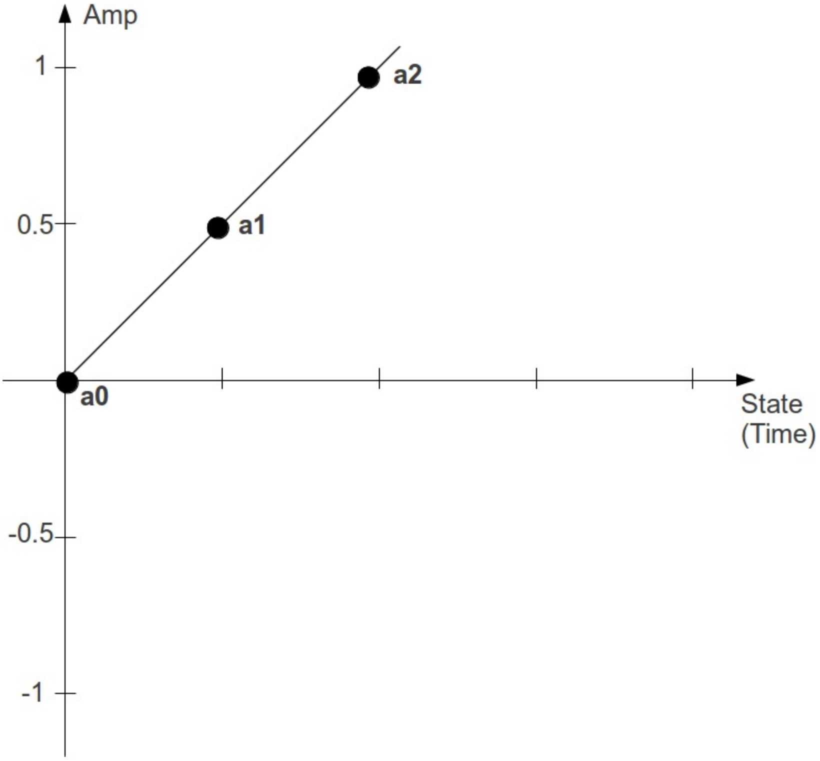
Physical modelling synthesis techniques do not build sound using wave tables, oscillators and audio signal generators, instead they attempt to establish a model, as a system in itself, which which can then produce sound because of how the function it producers time varies with time. A physical model usually derives from the real physical world, but could be any time-varying system. Physical modelling is an exciting area for the production of new sounds.

Compared with the complexity of a real-world physically dynamic system a physical model will most likely represent a brutal simplification. Nevertheless, using this technique will demand a lot of formulae, because physical models are described in terms of mathematics. Although designing a model may require some considerable work, once established the results commonly exhibit a lively tone with time-varying partials and a "natural" difference between attack and release by their very design - features that other synthesis techniques will demand more from the end user in order to establish.

Csound already contains many ready-made physical models as opcodes but you can still build your own from scratch. This chapter will look at how to implement two classical models from first principles and then introduce a number of Csound's ready made physical modelling opcodes.

THE MASS-SPRING MODEL[1]

Many oscillating processes in nature can be modelled as connections of masses and springs. Imagine one mass-spring unit which has been set into motion. This system can be described as a sequence of states, where every new state results from the two preceding ones. Assumed the first state *a0* is 0 and the second state *a1* is 0.5. Without the restricting force of the spring, the mass would continue moving unimpeded following a constant velocity:

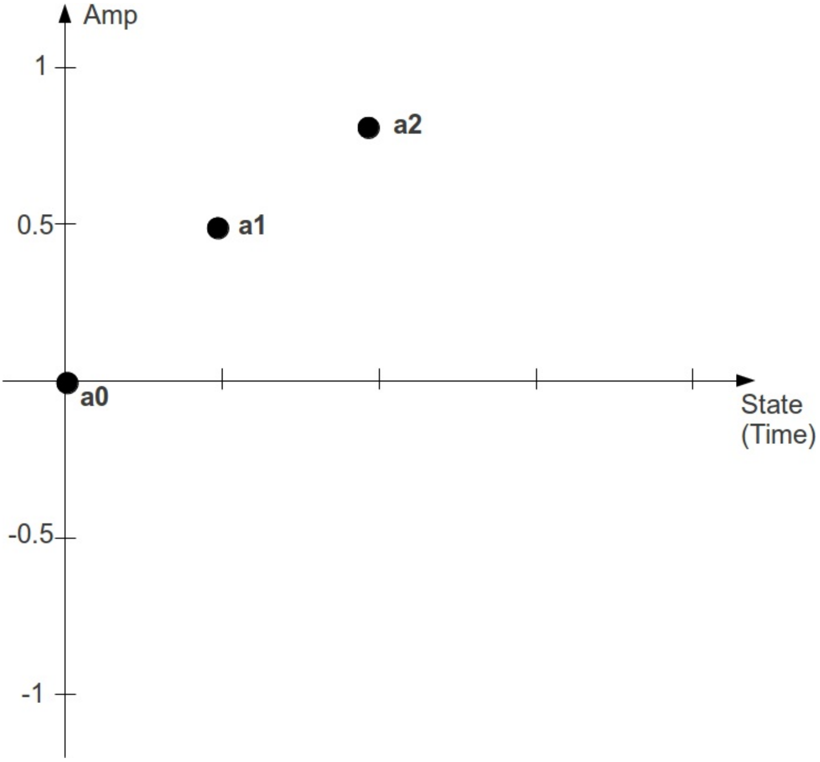


As the velocity between the first two states can be described as *a1-a0*, the value of the third state *a2* will be:

a2 = a1 + (a1 - a0) = 0.5 + 0.5 = 1

But, the spring pulls the mass back with a force which increases the further the mass moves away from the point of equilibrium. Therefore the masses movement can be described as the product of a constant factor *c* and the last position *a1*. This damps the continuous movement of the mass so that for a factor of c=0.4 the next position will be:

*a2 = (a1 + (a1 - a0)) - c * a1 = 1 - 0.2 = 0.8*



Csound can easily calculate the values by simply applying the formulae. For the first k-cycle[2] , they are set via the init opcode. After calculating the new state, *a1* becomes *a0* and *a2* becomes *a1* for the next k-cycle. This is a csd which prints the new values five times per second. (The states are named here as *k0/k1/k2* instead of *a0/a1/a2*, because k-rate values are needed here for printing instead of audio samples.)

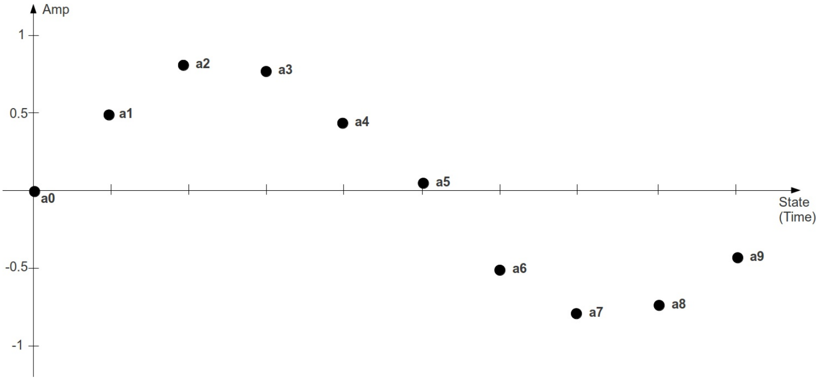
EXAMPLE 04G01.csd

```
<CsoundSynthesizer>
<CsOptions>
-n ;no sound
</CsOptions>
<CsInstruments>
sr = 44100
ksmps = 8820 ;5 steps per second

instr PrintVals
;initial values
kstep init 0
k0 init 0
k1 init 0.5
kc init 0.4
;calculation of the next value
k2 = k1 + (k1 - k0) - kc * k1
printks "Sample=%d: k0 = %.3f, k1 = %.3f, k2 = %.3f\n", 0, kstep, k0, k1, k2
;actualize values for the next step
kstep = kstep+1
k0 = k1
k1 = k2
endin
```

```
</CsInstruments>
<CsScore>
i "PrintVals" 0 10
</CsScore>
</CsoundSynthesizer>
;example by joachim heintz
```

The output starts with:

```
State=0:  k0 =  0.000,  k1 =  0.500,  k2 =  0.800
State=1:  k0 =  0.500,  k1 =  0.800,  k2 =  0.780
State=2:  k0 =  0.800,  k1 =  0.780,  k2 =  0.448
State=3:  k0 =  0.780,  k1 =  0.448,  k2 = -0.063
State=4:  k0 =  0.448,  k1 = -0.063,  k2 = -0.549
State=5:  k0 = -0.063,  k1 = -0.549,  k2 = -0.815
State=6:  k0 = -0.549,  k1 = -0.815,  k2 = -0.756
State=7:  k0 = -0.815,  k1 = -0.756,  k2 = -0.393
State=8:  k0 = -0.756,  k1 = -0.393,  k2 =  0.126
State=9:  k0 = -0.393,  k1 =  0.126,  k2 =  0.595
State=10: k0 =  0.126,  k1 =  0.595,  k2 =  0.826
State=11: k0 =  0.595,  k1 =  0.826,  k2 =  0.727
State=12: k0 =  0.826,  k1 =  0.727,  k2 =  0.337
```



So, a sine wave has been created, without the use of any of Csound's oscillators...

Here is the audible proof:

EXAMPLE 04G02.csd

```
<CsoundSynthesizer>
<CsOptions>
-odac
</CsOptions>
<CsInstruments>
sr = 44100
ksmps = 1
nchnls = 2
0dbfs = 1

instr MassSpring
;initial values
a0        init      0
a1        init      0.05
ic        =         0.01 ;spring constant
;calculation of the next value
a2        =         a1+(a1-a0) - ic*a1
          outs      a0, a0
;actualize values for the next step
a0        =         a1
a1        =         a2
endin
</CsInstruments>
<CsScore>
i "MassSpring" 0 10
</CsScore>
</CsoundSynthesizer>
;example by joachim heintz, after martin neukom
```

As the next sample is calculated in the next control cycle, ksmps has to be set to 1.[3] The resulting frequency depends on the spring constant: the higher the constant, the higher the frequency. The resulting amplitude depends on both, the starting value and the spring constant.

This simple model shows the basic principle of a physical modelling synthesis: creating a system which produces sound because it varies in time. Certainly it is not the goal of physical modelling synthesis to reinvent the wheel of a sine wave. But modulating the parameters of a model may lead to interesting results. The next example varies the spring constant, which is now no longer a constant:

EXAMPLE 04G03.csd

```
<CsoundSynthesizer>
<CsOptions>
-odac
</CsOptions>
<CsInstruments>
sr = 44100
ksmps = 1
nchnls = 2
0dbfs = 1

instr MassSpring
;initial values
a0        init      0
a1        init      0.05
kc        randomi   .001, .05, 8, 3
;calculation of the next value
a2        =         a1+(a1-a0) - kc*a1
          outs      a0, a0
;actualize values for the next step
a0        =         a1
a1        =         a2
endin
</CsInstruments>
<CsScore>
i "MassSpring" 0 10
</CsScore>
</CsoundSynthesizer>
;example by joachim heintz
```

Working with physical modelling demands thought in more physical or mathematical terms: examples of this might be if you were to change the formula when a certain value of *c* had been reached, or combine more than one spring.

THE KARPLUS-STRONG ALGORITHM: PLUCKED STRING

The Karplus-Strong algorithm provides another simple yet interesting example of how physical modelling can be used to synthesized sound. A buffer is filled with random values of either +1 or -1. At the end of the buffer, the mean of the first and the second value to come out of the buffer is calculated. This value is then put back at the beginning of the buffer, and all the values in the buffer are shifted by one position.

This is what happens for a buffer of five values, for the first five steps:

initial state	1	-1	1	1	-1
step 1	0	1	-1	1	1
step 2	1	0	1	-1	1
step 3	0	1	0	1	-1
step 4	0	0	1	0	1
step 5	0.5	0	0	1	0

The next Csound example represents the content of the buffer in a function table, implements and executes the algorithm, and prints the result after each five steps which here is referred to as one cycle:

EXAMPLE 04G04.csd

```
<CsoundSynthesizer>
<CsOptions>
-n
</CsOptions>
<CsInstruments>
```

```
sr = 44100
ksmps = 32
nchnls = 1
0dbfs = 1

  opcode KS, 0, ii
  ;performs the karplus-strong algorithm
iTab, iTbSiz xin
;calculate the mean of the last two values
iUlt      tab_i     iTbSiz-1, iTab
iPenUlt   tab_i     iTbSiz-2, iTab
iNewVal   =         (iUlt + iPenUlt) / 2
;shift values one position to the right
indx      =         iTbSiz-2
loop:
iVal      tab_i     indx, iTab
          tabw_i    iVal, indx+1, iTab
          loop_ge   indx, 1, 0, loop
;fill the new value at the beginning of the table
          tabw_i    iNewVal, 0, iTab
  endop

  opcode PrintTab, 0, iiS
  ;prints table content, with a starting string
iTab, iTbSiz, Sout xin
indx      =         0
loop:
iVal      tab_i     indx, iTab
Snew      sprintf   "%8.3f", iVal
Sout      strcat    Sout, Snew
          loop_lt   indx, 1, iTbSiz, loop
          puts      Sout, 1
  endop

instr ShowBuffer
;fill the function table
iTab      ftgen     0, 0, -5, -2, 1, -1, 1, 1, -1
iTbLen    tableng   iTab
;loop cycles (five states)
iCycle    =         0
cycle:
Scycle    sprintf   "Cycle %d:", iCycle
          PrintTab  iTab, iTbLen, Scycle
;loop states
iState    =         0
state:
          KS        iTab, iTbLen
          loop_lt   iState, 1, iTbLen, state
          loop_lt   iCycle, 1, 10, cycle
endin

</CsInstruments>
<CsScore>
i "ShowBuffer" 0 1
</CsScore>
</CsoundSynthesizer>
```

This is the output:

```
Cycle 0:   1.000  -1.000   1.000   1.000  -1.000
Cycle 1:   0.500   0.000   0.000   1.000   0.000
Cycle 2:   0.500   0.250   0.000   0.500   0.500
Cycle 3:   0.500   0.375   0.125   0.250   0.500
Cycle 4:   0.438   0.438   0.250   0.188   0.375
Cycle 5:   0.359   0.438   0.344   0.219   0.281
Cycle 6:   0.305   0.398   0.391   0.281   0.250
Cycle 7:   0.285   0.352   0.395   0.336   0.266
Cycle 8:   0.293   0.318   0.373   0.365   0.301
Cycle 9:   0.313   0.306   0.346   0.369   0.333
```

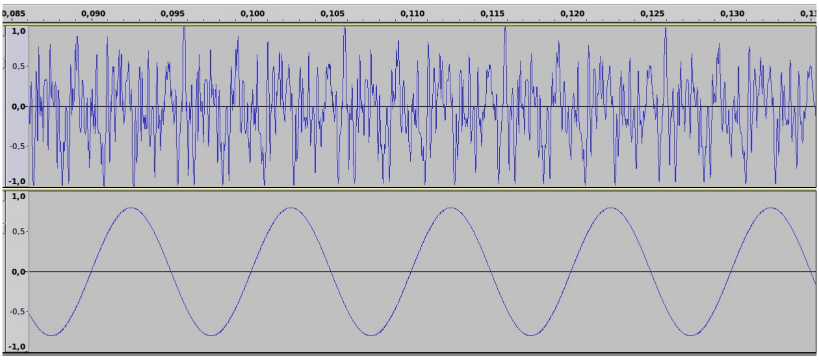
It can be seen clearly that the values get smoothed more and more from cycle to cycle. As the buffer size is very small here, the values tend to come to a constant level; in this case 0.333. But for larger buffer sizes, after some cycles the buffer content has the effect of a period which is repeated with a slight loss of amplitude. This is how it sounds, if the buffer size is 1/100 second (or 441 samples at sr=44100):

EXAMPLE 04G05.csd

```
<CsoundSynthesizer>
<CsOptions>
-odac
</CsOptions>
<CsInstruments>
```

```
sr = 44100
ksmps =  1
nchnls = 2
0dbfs = 1

instr 1
;delay time
iDelTm    =         0.01
;fill the delay line with either -1 or 1 randomly
kDur      timeinsts
 if kDur < iDelTm then
aFill     rand      1, 2, 1, 1 ;values 0-2
aFill     =         floor(aFill)*2 - 1 ;just -1 or +1
          else
aFill     =         0
 endif
;delay and feedback
aUlt      init      0 ;last sample in the delay line
aUlt1     init      0 ;delayed by one sample
aMean     =         (aUlt+aUlt1)/2 ;mean of these two
aUlt      delay     aFill+aMean, iDelTm
aUlt1     delay1    aUlt
          outs      aUlt, aUlt
endin

</CsInstruments>
<CsScore>
i 1 0 60
</CsScore>
</CsoundSynthesizer>
;example by joachim heintz, after martin neukom
```

This sound resembles a plucked string: at the beginning the sound is noisy but after a short period of time it exhibits periodicity. As can be heard, unless a natural string, the steady state is virtually endless, so for practical use it needs some fade-out. The frequency the listener perceives is related to the length of the delay line. If the delay line is 1/100 of a second, the perceived frequency is 100 Hz. Compared with a sine wave of similar frequency, the inherent periodicity can be seen, and also the rich overtone structure:



Csound also contains over forty opcodes which provide a wide variety of ready-made physical models and emulations. A small number of them will be introduced here to give a brief overview of the sort of things available.

WGBOW - A WAVEGUIDE EMULATION OF A BOWED STRING BY PERRY COOK

Perry Cook is a prolific author of physical models and a lot of his work has been converted into Csound opcodes. A number of these models wgbow, wgflute, wgclar wgbowedbar and wgbrass are based on waveguides. A waveguide, in its broadest sense, is some sort of mechanism that limits the extend of oscillations, such as a vibrating string fixed at both ends or a pipe. In these sorts of physical model a delay is used to emulate these limits. One of these, wgbow, implements an emulation of a bowed string. Perhaps the most interesting aspect of many physical models in not specifically whether they emulate the target instrument played in a conventional way accurately but the facilities they provide for extending the physical limits of the instrument and how it is played - there are already vast sample libraries and software samplers for emulating conventional instruments played conventionally. wgbow offers several interesting options for experimentation including the ability to modulate the bow pressure and the bowing position at k-rate. Varying bow pressure will change the tone of the sound produced by changing the harmonic emphasis. As bow pressure reduces, the fundamental of the tone becomes weaker and overtones become more prominent. If the bow pressure is reduced further the abilty of the system to produce a resonance at all collapse. This boundary between tone production and the inability to produce a tone can provide some interesting new sound effect. The following example explores this sound area by modulating the bow pressure parameter around this threshold. Some additional features to enhance the example are that 7 different notes are played simultaneously, the bow pressure modulations in the right channel are delayed by a varying amount with respect top the left channel in order to create a stereo effect and a reverb has been added.

EXAMPLE 04G06.csd

```
<CsoundSynthesizer>

<CsOptions>
-odac
</CsOptions>

<CsInstruments>

sr      =       44100
ksmps   =       32
nchnls  =       2
0dbfs   =       1
        seed    0

gisine  ftgen 0,0,4096,10,1

gaSendL,gaSendR init 0

 instr 1 ; wgbow instrument
kamp     =        0.3
kfreq    =        p4
ipres1   =        p5
ipres2   =        p6
; kpres (bow pressure) defined using a random spline
kpres    rspline  p5,p6,0.5,2
krat     =        0.127236
kvibf    =        4.5
kvibamp  =        0
iminfreq =        20
; call the wgbow opcode
aSigL  wgbow    kamp,kfreq,kpres,krat,kvibf,kvibamp,gisine,iminfreq
; modulating delay time
kdel     rspline  0.01,0.1,0.1,0.5
; bow pressure parameter delayed by a varying time in the right channel
kpres    vdel_k   kpres,kdel,0.2,2
aSigR  wgbow   kamp,kfreq,kpres,krat,kvibf,kvibamp,gisine,iminfreq
         outs     aSigL,aSigR
; send some audio to the reverb
gaSendL  =        gaSendL + aSigL/3
gaSendR  =        gaSendR + aSigR/3
 endin

 instr 2 ; reverb
aRvbL,aRvbR reverbsc gaSendL,gaSendR,0.9,7000
            outs     aRvbL,aRvbR
            clear    gaSendL,gaSendR
 endin

</CsInstruments>

<CsScore>
; instr. 1
;  p4 = pitch (hz.)
;  p5 = minimum bow pressure
;  p6 = maximum bow pressure
; 7 notes played by the wgbow instrument
i 1  0 480  70 0.03 0.1
i 1  0 480  85 0.03 0.1
i 1  0 480 100 0.03 0.09
i 1  0 480 135 0.03 0.09
i 1  0 480 170 0.02 0.09
i 1  0 480 202 0.04 0.1
i 1  0 480 233 0.05 0.11
; reverb instrument
i 2 0 480
</CsScore>

</CsoundSynthesizer>
```

This time a stack of eight sustaining notes, each separated by an octave, vary their 'bowing position' randomly and independently. You will hear how different bowing positions accentuates and attenuates different partials of the bowing tone. To enhance the sound produced some filtering with tone and pareq is employed and some reverb is added.

EXAMPLE 04G07.csd

```
<CsoundSynthesizer>

<CsOptions>
-odac
</CsOptions>

<CsInstruments>

sr      =       44100
ksmps   =       32
nchnls  =       2
0dbfs   =       1
        seed    0

gisine  ftgen 0,0,4096,10,1

gaSend init 0

 instr 1 ; wgbow instrument
kamp     =        0.1
kfreq    =        p4
kpres    =        0.2
krat     rspline  0.006,0.988,0.1,0.4
kvibf    =        4.5
kvibamp  =        0
iminfreq =        20
aSig  wgbow    kamp,kfreq,kpres,krat,kvibf,kvibamp,gisine,iminfreq
aSig     butlp     aSig,2000
aSig     pareq    aSig,80,6,0.707
         outs     aSig,aSig
gaSend   =        gaSend + aSig/3
 endin

 instr 2 ; reverb
aRvbL,aRvbR reverbsc gaSend,gaSend,0.9,7000
            outs     aRvbL,aRvbR
            clear    gaSend
 endin

</CsInstruments>

<CsScore>
; instr. 1 (wgbow instrument)
;  p4 = pitch (hertz)
; wgbow instrument
i 1  0 480  20
i 1  0 480  40
i 1  0 480  80
i 1  0 480  160
i 1  0 480  320
i 1  0 480  640
i 1  0 480  1280
i 1  0 480  2460
; reverb instrument
i 2 0 480
</CsScore>

</CsoundSynthesizer>
```

All of the wg- family of opcodes are worth exploring and often the approach taken here - exploring each input parameter in isolation whilst the others retain constant values - sets the path to understanding the model better. Tone production with wgbrass is very much dependent upon the relationship between intended pitch and lip tension, random experimentation with this opcode is as likely to result in silence as it is in sound and in this way is perhaps a reflection of the experience of learning a brass instrument when the student spends most time push air silently through the instrument. With patience it is capable of some interesting sounds however. In its case, I would recommend building a realtime GUI and exploring the interaction of its input arguments that way. wgbowedbar, like a number of physical modelling algorithms, is rather unstable. This is not necessary a design flaw in the algorithm but instead perhaps an indication that the algorithm has been left quite open for out experimentation - or abuse. In these situation caution is advised in order to protect ears and loudspeakers. Positive feedback within the model can result in signals of enormous amplitude very quickly. Employment of the clip opcode as a means of some protection is recommended when experimenting in realtime.

BARMODEL - A MODEL OF A STRUCK METAL BAR BY STEFAN BILBAO

barmodel can also imitate wooden bars, tubular bells, chimes and other resonant inharmonic objects. barmodel is a model that can easily be abused to produce ear shreddingly loud sounds therefore precautions are advised when experimenting with it in realtime. We are presented with a wealth of input arguments such as 'stiffness', 'strike position' and 'strike velocity', which relate in an easily understandable way to the physical process we are emulating. Some parameters will evidently have more of a dramatic effect on the sound produced than other and again it is recommended to create a realtime GUI for exploration. Nonetheless, a fixed example is provided below that should offer some insight into the kinds of sounds possible.

Probably the most important parameter for us is the stiffness of the bar. This actually provides us with our pitch control and is not in cycle-per-second so some experimentation will be required to find a desired pitch. There is a relationship between stiffness and the parameter used to define the width of the strike - when the stiffness coefficient is higher a wider strike may be required in order for the note to sound. Strike width also impacts upon the tone produced, narrower strikes generating emphasis upon upper partials (provided a tone is still produced) whilst wider strikes tend to emphasize the fundamental).

The parameter for strike position also has some impact upon the spectral balance. This effect may be more subtle and may be dependent upon some other parameter settings, for example, when strike width is particularly wide, its effect may be imperceptible. A general rule of thumb here is that is that in order to achieve the greatest effect from strike position, strike width should be as low as will still produce a tone. This kind of interdependency between input parameters is the essence of working with a physical model that can be both intriguing and frustrating.

An important parameter that will vary the impression of the bar from metal to wood is

An interesting feature incorporated into the model in the ability to modulate the point along the bar at which vibrations are read. This could also be described as pick-up position. Moving this scanning location results in tonal and amplitude variations. We just have control over the frequency at which the scanning location is modulated.

EXAMPLE 04G07.csd [4]

```
<CsoundSynthesizer>
<CsOptions>
-odac
</CsOptions>
<CsInstruments>
sr     = 44100
ksmps  = 32
nchnls = 2
0dbfs  = 1

 instr   1
; boundary conditions 1=fixed 2=pivot 3=free
kbcL    =               1
kbcR    =               1
; stiffness
iK      =               p4
; high freq. loss (damping)
ib      =               p5
; scanning frequency
kscan   rspline         p6,p7,0.2,0.8
; time to reach 30db decay
iT30    =               p3
; strike position
ipos    random          0,1
; strike velocity
ivel    =               1000
; width of strike
iwid    =               0.1156
aSig    barmodel        kbcL,kbcR,iK,ib,kscan,iT30,ipos,ivel,iwid
kPan rspline         0.1,0.9,0.5,2
aL,aR   pan2            aSig,kPan
 outs             aL,aR
 endin

</CsInstruments>
```

```
<CsScore>
;t 0 90 1 30 2 60 5 90 7 30
; p4 = stiffness (pitch)

#define gliss(dur'Kstrt'Kend'b'scan1'scan2)
#
i 1 0     20 $Kstrt $b $scan1 $scan2
i 1 ^+0.05 $dur >     $b $scan1 $scan2
i 1 ^+0.05 $dur >     $b $scan1 $scan2
i 1 ^+0.05 $dur >     $b $scan1 $scan2
i 1 ^+0.05 $dur >     $b $scan1 $scan2
i 1 ^+0.05 $dur >     $b $scan1 $scan2
i 1 ^+0.05 $dur >     $b $scan1 $scan2
i 1 ^+0.05 $dur >     $b $scan1 $scan2
i 1 ^+0.05 $dur >     $b $scan1 $scan2
i 1 ^+0.05 $dur >     $b $scan1 $scan2
i 1 ^+0.05 $dur >     $b $scan1 $scan2
i 1 ^+0.05 $dur >     $b $scan1 $scan2
i 1 ^+0.05 $dur >     $b $scan1 $scan2
i 1 ^+0.05 $dur >     $b $scan1 $scan2
i 1 ^+0.05 $dur >     $b $scan1 $scan2
i 1 ^+0.05 $dur >     $b $scan1 $scan2
i 1 ^+0.05 $dur >     $b $scan1 $scan2
i 1 ^+0.05 $dur $Kend $b $scan1 $scan2
#
$gliss(15'40'400'0.0755'0.1'2)
b 5
$gliss(2'80'800'0.755'0'0.1)
b 10
$gliss(3'10'100'0.1'0'0)
b 15
$gliss(40'40'433'0'0.2'5)
e
</CsScore>
</CsoundSynthesizer>
; example written by Iain McCurdy
```

PHISEM - PHYSICALLY INSPIRED STOCHASTIC EVENT MODELING

The PhiSEM set of models in Csound, again based on the work of Perry Cook, imitate instruments that rely on collisions between smaller sound producing object to produce their sounds. These models include a tambourine, a set of bamboo windchimes and sleighbells. These models algorithmically mimic these multiple collisions internally so that we only need to define elements such as the number of internal elements (timbrels, beans, bells etc.) internal damping and resonances. Once again the most interesting aspect of working with a model is to stretch the physical limits so that we can hear the results from, for example, a maraca with an impossible number of beans, a tambourine with so little internal damping that it never decays. In the following example I explore tambourine, bamboo and sleighbells each in turn, first in a state that mimics the source instrument and then with some more extreme conditions.

EXAMPLE 04G08.csd

```
<CsoundSynthesizer>

<CsOptions>
-odac
</CsOptions>

<CsInstruments>

sr     = 44100
ksmps  = 32
nchnls = 1
0dbfs  = 1

 instr 1 ; tambourine
iAmp      =           p4
iDettack  =           0.01
iNum      =           p5
iDamp     =           p6
iMaxShake =           0
iFreq     =           p7
iFreq1    =           p8
iFreq2    =           p9
aSig      tambourine  iAmp,iDettack,iNum,iDamp,iMaxShake,iFreq,iFreq1,iFreq2
          out         aSig
 endin
 instr 2 ; bamboo
iAmp      =           p4
iDettack  =           0.01
iNum      =           p5
iDamp     =           p6
iMaxShake =           0
iFreq     =           p7
iFreq1    =           p8
iFreq2    =           p9
aSig      bamboo      iAmp,iDettack,iNum,iDamp,iMaxShake,iFreq,iFreq1,iFreq2
          out         aSig
 endin

 instr 3 ; sleighbells
iAmp      =           p4
iDettack  =           0.01
iNum      =           p5
iDamp     =           p6
iMaxShake =           0
iFreq     =           p7
iFreq1    =           p8
iFreq2    =           p9
aSig      sleighbells iAmp,iDettack,iNum,iDamp,iMaxShake,iFreq,iFreq1,iFreq2
          out         aSig
 endin

</CsInstruments>

<CsScore>
; p4 = amp.
; p5 = number of timbrels
; p6 = damping
; p7 = freq (main)
; p8 = freq 1
; p9 = freq 2

; tambourine
i 1 0 1 0.1  32 0.47 2300 5600 8100
i 1 + 1 0.1  32 0.47 2300 5600 8100
i 1 + 2 0.1  32 0.75 2300 5600 8100
i 1 + 2 0.05  2 0.75 2300 5600 8100
i 1 + 1 0.1  16 0.65 2000 4000 8000
i 1 + 1 0.1  16 0.65 1000 2000 3000
i 1 8 2 0.01  1 0.75 1257 2653 6245
i 1 8 2 0.01  1 0.75  673 3256 9102
i 1 8 2 0.01  1 0.75  314 1629 4756

b 10

; bamboo
i 2 0 1 0.4 1.25 0.0  2800 2240 3360
i 2 + 1 0.4 1.25 0.0  2800 2240 3360
i 2 + 2 0.4 1.25 0.05 2800 2240 3360
i 2 + 2 0.2   10 0.05 2800 2240 3360
i 2 + 1 0.3   16 0.01 2000 4000 8000
i 2 + 1 0.3   16 0.01 1000 2000 3000
i 2 8 2 0.1    1 0.05 1257 2653 6245
i 2 8 2 0.1    1 0.05 1073 3256 8102
i 2 8 2 0.1    1 0.05  514 6629 9756

b 20

; sleighbells
i 3 0 1 0.7 1.25 0.17 2500 5300 6500
i 3 + 1 0.7 1.25 0.17 2500 5300 6500
i 3 + 2 0.7 1.25 0.3  2500 5300 6500
i 3 + 2 0.4   10 0.3  2500 5300 6500
i 3 + 1 0.5   16 0.2  2000 4000 8000
i 3 + 1 0.5   16 0.2  1000 2000 3000
i 3 8 2 0.3    1 0.3  1257 2653 6245
i 3 8 2 0.3    1 0.3  1073 3256 8102
i 3 8 2 0.3    1 0.3   514 6629 9756
e
</CsScore>

</CsoundSynthesizer>
; example written by Iain McCurdy
```

Physical modelling can produce rich, spectrally dynamic sounds with user manipulation usually abstracted to a small number of descriptive parameters. Csound offers a wealth of other opcodes for physical modelling which cannot all be introduced here so the user is encouraged to explore based on the approaches exemplified here. You can find lists in the chapters Models and Emulations, Scanned Synthesis and Waveguide Physical Modeling of the Csound Manual.

1. The explanation here follows chapter 8.1.1 of Martin Neukom's *Signale Systeme Klangsynthese* (Bern 2003)^
2. See chapter 03A INITIALIZATION AND PERFORMANCE PASS for more information about Csound's performance loops.^
3. If defining this as a UDO, a local ksmps=1 could be set without affecting the general ksmps. See chapter 03F USER DEFINED OPCODES and the Csound Manual for setksmps for more information.^
4. See chapter 03G MACROS about the use of macros in the score.^

05 SOUND MODIFICATION

27. ENVELOPES
28. PANNING AND SPATIALIZATION
29. FILTERS
30. DELAY AND FEEDBACK
31. REVERBERATION
32. AM / RM / WAVESHAPING
33. GRANULAR SYNTHESIS
34. CONVOLUTION
35. FOURIER TRANSFORMATION / SPECTRAL PROCESSING

27. ENVELOPES

Envelopes are used to define how a value changes over time. In early synthesizers, envelopes were used to define the changes in amplitude in a sound across its duration thereby imbuing sounds characteristics such as 'percussive', or 'sustaining'. Of course envelopes can be applied to any parameter and not just amplitude.

Csound offers a wide array of opcodes for generating envelopes including ones which emulate the classic ADSR (attack-decay-sustain-release) envelopes found on hardware and commercial software synthesizers. A selection of these opcodes, which represent the basic types, shall be introduced here

The simplest opcode for defining an envelope is line. *line* describes a single envelope segment as a straight line between a start value and an end value which has a given duration.

```
ares line ia, idur, ib
kres line ia, idur, ib
```

In the following example *line* is used to create a simple envelope which is then used as the amplitude control of a *poscil* oscillator. This envelope starts with a value of 0.5 then over the course of 2 seconds descends in linear fashion to zero.

EXAMPLE 05A01.csd

```
<CsoundSynthesizer>

<CsOptions>
-odac ; activates real time sound output
</CsOptions>

<CsInstruments>
; Example by Iain McCurdy
sr = 44100
ksmps = 32
nchnls = 1
0dbfs = 1

giSine   ftgen    0, 0, 2^12, 10, 1 ; a sine wave

  instr 1
aEnv     line     0.5, 2, 0         ; amplitude envelope
aSig     poscil   aEnv, 500, giSine ; audio oscillator
         out      aSig              ; audio sent to output
  endin

</CsInstruments>
<CsScore>
i 1 0 2 ; instrument 1 plays a note for 2 seconds
e
</CsScore>
</CsoundSynthesizer>
```

The envelope in the above example assumes that all notes played by this instrument will be 2 seconds long. In practice it is often beneficial to relate the duration of the envelope to the duration of the note (p3) in some way. In the next example the duration of the envelope is replaced with the value of p3 retrieved from the score, whatever that may be. The envelope will be stretched or contracted accordingly.

EXAMPLE 05A02.csd

```
<CsoundSynthesizer>

<CsOptions>
-odac ;activates real time sound output
</CsOptions>

<CsInstruments>
;Example by Iain McCurdy
sr = 44100
ksmps = 32
nchnls = 1
0dbfs = 1

giSine   ftgen    0, 0, 2^12, 10, 1 ; a sine wave
```

```
  instr 1
; A single segment envelope. Time value defined by note duration.
aEnv     line     0.5, p3, 0
aSig     poscil   aEnv, 500, giSine ; an audio oscillator
         out      aSig              ; audio sent to output
  endin

</CsInstruments>
<CsScore>
; p1 p2  p3
i 1  0    1
i 1  2  0.2
i 1  3    4
e
</CsScore>
</CsoundSynthesizer>
```

It may not be disastrous if a envelope's duration does not match p3 and indeed there are many occasions when we want an envelope duration to be independent of p3 but we need to remain aware that if p3 is shorter than an envelope's duration then that envelope will be truncated before it is allowed to complete and if p3 is longer than an envelope's duration then the envelope will complete before the note ends (the consequences of this latter situation will be looked at in more detail later on in this section).

line (and most of Csound's envelope generators) can output either k or a-rate variables. k-rate envelopes are computationally cheaper than a-rate envelopes but in envelopes with fast moving segments quantization can occur if they output a k-rate variable, particularly when the control rate is low, which in the case of amplitude envelopes can lead to clicking artefacts or distortion.

linseg is an elaboration of *line* and allows us to add an arbitrary number of segments by adding further pairs of time durations followed envelope values. Provided we always end with a value and not a duration we can make this envelope as long as we like.

In the next example a more complex amplitude envelope is employed by using the *linseg* opcode. This envelope is also note duration (p3) dependent but in a more elaborate way. A attack-decay stage is defined using explicitly declared time durations. A release stage is also defined with an explicitly declared duration. The sustain stage is the p3 dependent stage but to ensure that the duration of the entire envelope still adds up to p3, the explicitly defined durations of the attack, decay and release stages are subtracted from the p3 dependent sustain stage duration. For this envelope to function correctly it is important that p3 is not less than the sum of all explicitly defined envelope segment durations. If necessary, additional code could be employed to circumvent this from happening.

EXAMPLE 05A03.csd

```
<CsoundSynthesizer>

<CsOptions>
-odac ; activates real time sound output
</CsOptions>

<CsInstruments>
; Example by Iain McCurdy

sr = 44100
ksmps = 32
nchnls = 1
0dbfs = 1

giSine   ftgen    0, 0, 2^12, 10, 1 ; a sine wave

  instr 1
; a more complex amplitude envelope:
;                 |-attack-|-decay--|---sustain---|-release-|
aEnv     linseg   0, 0.01, 1, 0.1, 0.1, p3-0.21, 0.1, 0.1, 0
aSig     poscil   aEnv, 500, giSine
         out      aSig
  endin

</CsInstruments>

<CsScore>
i 1 0 1
i 1 2 5
e
</CsScore>
```

```
</CsoundSynthesizer>
```

The next example illustrates an approach that can be taken whenever it is required that more than one envelope segment duration be p3 dependent. This time each segment is a fraction of p3. The sum of all segments still adds up to p3 so the envelope will complete across the duration of each each note regardless of duration.

EXAMPLE 05A04.csd

```
<CsoundSynthesizer>

<CsOptions>
-odac ;activates real time sound output
</CsOptions>

<CsInstruments>
;Example by Iain McCurdy

sr = 44100
ksmps = 32
nchnls = 1
0dbfs = 1

giSine   ftgen    0, 0, 2^12, 10, 1; a sine wave

  instr 1
aEnv     linseg   0, p3*0.5, 1, p3*0.5, 0 ; rising then falling envelope
aSig     poscil   aEnv, 500, giSine
         out      aSig
  endin

</CsInstruments>

<CsScore>
; 3 notes of different durations are played
i 1 0   1
i 1 2 0.1
i 1 3   5
e
</CsScore>

</CsoundSynthesizer>
```

The next example highlights an important difference in the behaviours of *line* and *linseg* when p3 exceeds the duration of an envelope.

When a note continues beyond the end of the final value of a *linseg* defined envelope the final value of that envelope is held. A *line* defined envelope behaves differently in that instead of holding its final value it continues in a trajectory defined by the last segment.

This difference is illustrated in the following example. The *linseg* and *line* envelopes of instruments 1 and 2 appear to be the same but the difference in their behaviour as described above when they continue beyond the end of their final segment is clear when listening to the example.

Note that information given in the Csound Manual in regard to this matter is incorrect at the time of writing.

EXAMPLE 05A05.csd

```
<CsoundSynthesizer>

<CsOptions>
-odac ; activates real time sound output
</CsOptions>

<CsInstruments>
; Example by Iain McCurdy

sr = 44100
ksmps = 32
nchnls = 1
0dbfs = 1

giSine   ftgen    0, 0, 2^12, 10, 1 ; a sine wave

  instr 1 ; linseg envelope
aCps     linseg   300, 1, 600       ; linseg holds its last value
aSig     poscil   0.2, aCps, giSine
         out      aSig
  endin

  instr 2 ; line envelope
aCps     line     300, 1, 600       ; line continues its trajectory
aSig     poscil   0.2, aCps, giSine
         out      aSig
  endin

</CsInstruments>

<CsScore>
i 1 0 5 ; linseg envelope
i 2 6 5 ; line envelope
e
</CsScore>

</CsoundSynthesizer>
```

expon and expseg are versions of *line* and *linseg* that instead produce envelope segments with concave exponential rather than linear shapes. *expon* and *expseg* can often be more musically useful for envelopes that define amplitude or frequency as they will reflect the logarithmic nature of how these parameters are perceived. On account of the mathematics that is used to define these curves, we cannot define a value of zero at any node in the envelope and an envelope cannot cross the zero axis. If we require a value of zero we can instead provide a value very close to zero. If we still really need zero we can always subtract the offset value from the entire envelope in a subsequent line of code.

The following example illustrates the difference between *line* and *expon* when applied as amplitude envelopes.

EXAMPLE 05A06.csd

```
<CsoundSynthesizer>

<CsOptions>
-odac ; activates real time sound output
</CsOptions>

<CsInstruments>
; Example by Iain McCurdy

sr = 44100
ksmps = 32
nchnls = 1
0dbfs = 1

giSine   ftgen    0, 0, 2^12, 10, 1 ; a sine wave

  instr 1 ; line envelope
aEnv     line     1, p3, 0
aSig     poscil   aEnv, 500, giSine
         out      aSig
  endin

  instr 2 ; expon envelope
aEnv     expon    1, p3, 0.0001
aSig     poscil   aEnv, 500, giSine
         out      aSig
  endin

</CsInstruments>

<CsScore>
i 1 0 2 ; line envelope
i 2 2 1 ; expon envelope
e
</CsScore>

</CsoundSynthesizer>
```

The nearer our 'near-zero' values are to zero the quicker the curve will appear to reach 'zero'. In the next example smaller and smaller envelope end values are passed to the expon opcode using p4 values in the score. The percussive 'ping' sounds are perceived to be increasingly short.

EXAMPLE 05A07.csd

```
<CsoundSynthesizer>

<CsOptions>
```

```
-odac ; activates real time sound output
</CsOptions>

<CsInstruments>
; Example by Iain McCurdy

sr = 44100
ksmps = 32
nchnls = 1
0dbfs = 1

giSine   ftgen    0, 0, 2^12, 10, 1 ; a sine wave

  instr 1; expon envelope
iEndVal  =        p4 ; variable 'iEndVal' retrieved from score
aEnv     expon    1, p3, iEndVal
aSig     poscil   aEnv, 500, giSine
         out      aSig
  endin

</CsInstruments>

<CsScore>
;p1  p2 p3 p4
i 1  0  1  0.001
i 1  1  1  0.000001
i 1  2  1  0.000000000000001
e
</CsScore>

</CsoundSynthesizer>
```

Note that *expseg* does not behave like linseg in that it will not hold its last final value if p3 exceeds its entire duration, instead it continues its curving trajectory in a manner similar to *line* (and *expon*). This could have dangerous results if used as an amplitude envelope.

When dealing with notes with an indefinite duration at the time of initiation (such as midi activated notes or score activated notes with a negative p3 value), we do not have the option of using p3 in a meaningful way. Instead we can use one of Csound's envelopes that sense the ending of a note when it arrives and adjust their behaviour according to this. The opcodes in question are *linenr, linsegr, expsegr, madsr, mxadsr* and *envlpxr*. These opcodes wait until a held note is turned off before executing their final envelope segment. To facilitate this mechanism they extend the duration of the note so that this final envelope segment can complete.

The following example uses midi input (either hardware or virtual) to activate notes. The use of the *linsegr* envelope means that after the short attack stage lasting 0.1 seconds, the penultimate value of 1 will be held as long as the note is sustained but as soon as the note is released the note will be extended by 0.5 seconds in order to allow the final envelope segment to decay to zero.

EXAMPLE 05A08.csd

```
<CsoundSynthesizer>

<CsOptions>
-odac -+rtmidi=virtual -M0
; activate real time audio and MIDI (virtual midi device)
</CsOptions>

<CsInstruments>
; Example by Iain McCurdy

sr = 44100
ksmps = 32
nchnls = 1
0dbfs = 1

giSine   ftgen    0, 0, 2^12, 10, 1        ; a sine wave

  instr 1
icps     cpsmidi
;                 attack-|sustain-|-release
aEnv     linsegr  0, 0.01,  0.1,     0.5,0 ; envelope that senses note releases
aSig     poscil   aEnv, icps, giSine       ; audio oscillator
         out      aSig                     ; audio sent to output
  endin

</CsInstruments>

<CsScore>
```

```
f 0 240 ; csound performance for 4 minutes
e
</CsScore>

</CsoundSynthesizer>
```

Sometimes designing our envelope shape in a function table can provide us with shapes that are not possible using Csound's envelope generating opcodes. In this case the envelope can be read from the function table using an oscillator and if the oscillator is given a frequency of 1/p3 then it will read though the envelope just once across the duration of the note.

The following example generates an amplitude envelope which is the shape of the first half of a sine wave.

EXAMPLE 05A09.csd

```
<CsoundSynthesizer>

<CsOptions>
-odac ; activate real time sound output
</CsOptions>

<CsInstruments>
; Example by Iain McCurdy

sr = 44100
ksmps = 32
nchnls = 1
0dbfs = 1

giSine   ftgen    0, 0, 2^12, 10, 1        ; a sine wave
giEnv    ftgen    0, 0, 2^12, 9, 0.5, 1, 0 ; envelope shape: a half sine

  instr 1
; read the envelope once during the note's duration:
aEnv     poscil   1, 1/p3, giEnv
aSig     poscil   aEnv, 500, giSine        ; audio oscillator
         out      aSig                     ; audio sent to output
  endin

</CsInstruments>

<CsScore>
; 7 notes, increasingly short
i 1 0 2
i 1 2 1
i 1 3 0.5
i 1 4 0.25
i 1 5 0.125
i 1 6 0.0625
i 1 7 0.03125
f 0 7.1
e
</CsScore>

</CsoundSynthesizer>
```

LPSHOLD, LOOPSEG AND LOOPTSEG - A CSOUND TB303

The next example introduces three of Csound's looping opcodes, lpshold, loopseg and looptseg.

These opcodes generate envelopes which are looped at a rate corresponding to a defined frequency. What they each do could also be accomplished using the 'envelope from table' technique outlined in an earlier example but these opcodes provides the added convenience of encapsulating all the required code in one line without the need of any function tables. Furthermore all of the input arguments for these opcodes can be modulated at k-rate.

lpshold generates an envelope with in which each break point is held constant until a new break point is encountered. The resulting envelope will contain horizontal line segments. In our example this opcode will be used to generate a looping bassline in the fashion of a Roland TB303. Because the duration of the entire envelope is wholly dependent upon the frequency with which the envelope repeats - in fact it is the reciprocal – values for the durations of individual envelope segments are defining times in seconds but represent proportions of the entire envelope duration. The values given for all these segments do not need to add up to any specific value as Csound rescales the proportionality according to the sum of all segment durations. You might find it convenient to contrive to have them all add up to 1, or to 100 – either is equally valid. The other looping envelope opcodes discussed here use the same method for defining segment durations.

loopseg allows us to define a looping envelope with linear segements. In this example it is used to define the amplitude envelope of each individual note. Take note that whereas the *lpshold* envelope used to define the pitches of the melody repeats once per phrase the amplitude envelope repeats once for each note of the melody therefore its frequency is 16 times that of the melody envelope (there are 16 notes in our melodic phrase).

looptseg is an elaboration of *loopseg* in that is allows us to define the shape of each segment individually whether that be convex, linear of concave. This aspect is defined using the 'type' parameters. A 'type' value of 0 denotes a linear segement, a positive value denotes a convex segment with higher positive values resulting in increasingly convex curves. Negative values denote concave segments with increasing negative values resulting in increasingly concave curves. In this example *looptseg* is used to define a filter envelope which, like the amplitude envelope, repeats for every note. The addition of the 'type' parameter allows us to modulate the sharpness of the decay of the filter envelope. This is a crucial element of the TB303 design. Note that *looptseg* is only available in Csound 5.12 or later.

Other crucial features of this instrument such as 'note on/off' and 'hold' for each step are also implemented using *lpshold*.

A number of the input parameters of this example are modulated automatically using the randomi opcodes in order to keep it interesting. It is suggested that these modulations could be replaced by linkages to other controls such as QuteCsound widgets, FLTK widgets or MIDI controllers. Suggested ranges for each of these values are given in the .csd.

EXAMPLE 05A10.csd

```
<CsoundSynthesizer>
<CsOptions>
-odac ;activates real time sound output
</CsOptions>
<CsInstruments>
; Example by Iain McCurdy

sr = 44100
ksmps = 4
nchnls = 1
0dbfs = 1

seed 0; seed random number generators from system clock

  instr 1; Bassline instrument
kTempo    =            90          ; tempo in beats per minute
kCfBase   randomi      1,4, 0.2    ; base filter frequency (oct format)
kCfEnv    randomi      0,4,0.2     ; filter envelope depth
kRes      randomi      0.5,0.9,0.2 ; filter resonance
kVol      =            0.5         ; volume control
kDecay    randomi      -10,10,0.2  ; decay shape of the filter.
kWaveform =            0           ; oscillator waveform. 0=sawtooth 2=square
kDist     randomi      0,1,0.1     ; amount of distortion
kPhFreq   =            kTempo/240  ; freq. to repeat the entire phrase
kBtFreq   =            (kTempo)/15 ; frequency of each 1/16th note
; -- Envelopes with held segments  --
; The first value of each pair defines the relative duration of that segment,
; the second, the value itself.
; Note numbers (kNum) are defined as MIDI note numbers.
; Note On/Off (kOn) and hold (kHold) are defined as on/off switches, 1 or zero
;                    note:1      2     3     4     5     6     7     8
;                         9     10    11    12    13    14    15    16    0
kNum  lpshold kPhFreq, 0, 0,40,  1,42, 1,50, 1,49, 1,60, 1,54, 1,39, 1,40, \
                       1,46, 1,36, 1,40, 1,46, 1,50, 1,56, 1,44, 1,47,1,45
kOn   lpshold kPhFreq, 0, 0,1,   1,1,  1,1,  1,1,  1,1,  1,1,  1,0,  1,1,  \
                       1,1,  1,1,  1,1,  1,1,  1,1,  1,1,  1,0,  1,1,  1,1
kHold lpshold kPhFreq, 0, 0,0,   1,1,  1,1,  1,0,  1,0,  1,0,  1,0,  1,1,  \
                       1,0,  1,0,  1,1,  1,1,  1,1,  1,1,  1,0,  1,0,  1,0
kHold     vdel_k       kHold, 1/kBtFreq, 1 ; offset hold by 1/2 note duration
kNum      portk        kNum, (0.01*kHold)  ; apply portamento to pitch changes
                                           ; if note is not held: no portamento
kCps      =            cpsmidinn(kNum)     ; convert note number to cps
kOct      =            octcps(kCps)        ; convert cps to oct format
; amplitude envelope                  attack    sustain       decay  gap
kAmpEnv   loopseg      kBtFreq, 0, 0, 0,0.1, 1, 55/kTempo, 1, 0.1,0, 5/kTempo,0
kAmpEnv   =            (kHold=0?kAmpEnv:1)  ; if a held note, ignore envelope
kAmpEnv   port         kAmpEnv,0.001

; filter envelope
```

```
kCfOct    looptseg      kBtFreq,0,0,kCfBase+kCfEnv+kOct,kDecay,1,kCfBase+kOct
; if hold is off, use filter envelope, otherwise use steady state value:
kCfOct    =             (kHold=0?kCfOct:kCfBase+kOct)
kCfOct    limit        kCfOct, 4, 14 ; limit the cutoff frequency (oct format)
aSig      vco2         0.4, kCps, i(kWaveform)*2, 0.5 ; VCO-style oscillator
aFilt      lpf18        aSig, cpsoct(kCfOct), kRes, (kDist^2)*10 ; filter audio
aSig      balance       aFilt,aSig             ; balance levels
kOn       port         kOn, 0.006              ; smooth on/off switching
; audio sent to output, apply amp. envelope,
; volume control and note On/Off status
          out          aSig * kAmpEnv * kVol * kOn
  endin

</CsInstruments>
<CsScore>
i 1 0 3600 ; instr 1 plays for 1 hour
e
</CsScore>
</CsoundSynthesizer>
```

28. PANNING AND SPATIALIZATION

SIMPLE STEREO PANNING

Csound provides a large number of opcodes designed to assist in the distribution of sound amongst two or more speakers. These range from opcodes that merely balance a sound between two channel to ones that include algorithms to simulate the doppler shift that occurs when sound moves, algorithms that simulate the filtering and inter-aural delay that occurs as sound reaches both our ears and algorithms that simulate distance in an acoustic space.

First we will look at some 'first principles' methods of panning a sound between two speakers.

The simplest method that is typically encountered is to multiply one channel of audio (aSig) by a panning variable (kPan) and to multiply the other side by 1 minus the same variable like this:

```
aSigL  =  aSig * kPan
aSigR  =  aSig * (1 – kPan)
          outs aSigL, aSigR
```

where kPan is within the range zero to 1. If kPan is 1 all the signal will be in the left channel, if it is zero all the signal will be in the right channel and if it is 0.5 there will be signal of equal amplitide in both the left and the right channels. This way the signal can be continuously panned between the left and right channels.

The problem with this method is that the overall power drops as the sound is panned to the middle.

One possible solution to this problem is to take the square root of the panning variable for each channel before multiplying it to the audio signal like this:

```
aSigL  =     aSig * sqrt(kPan)
aSigR  =     aSig * sqrt((1 – kPan))
       outs  aSigL, aSigR
```

By doing this, the straight line function of the input panning variable becomes a convex curve so that less power is lost as the sound is panned centrally.

Using 90º sections of a sine wave for the mapping produces a more convex curve and a less immediate drop in power as the sound is panned away from the extremities. This can be implemented using the code shown below.

```
aSigL  =     aSig * sin(kPan*$M_PI_2)
aSigR  =     aSig * cos(kPan*$M_PI_2)
       outs  aSigL, aSigR
```

(Note that '$M_PI_2' is one of Csound's built in macros and is equivalent to pi/2.)

A fourth method, devised by Michael Gogins, places the point of maximum power for each channel slightly before the panning variable reaches its extremity. The result of this is that when the sound is panned dynamically it appears to move beyond the point of the speaker it is addressing. This method is an elaboration of the previous one and makes use of a different 90 degree section of a sine wave. It is implemented using the following code:

```
aSigL  =     aSig * sin((kPan + 0.5) * $M_PI_2)
aSigR  =     aSig * cos((kPan + 0.5) * $M_PI_2)
       outs  aSigL, aSigR
```

The following example demonstrates all three methods one after the other for comparison. Panning movement is controlled by a slow moving LFO. The input sound is filtered pink noise.

EXAMPLE 05B01.csd

```
<CsoundSynthesizer>

<CsOptions>
-odac ; activates real time sound output
</CsOptions>
```

```
<CsInstruments>
; Example by Iain McCurdy

sr = 44100
ksmps = 10
nchnls = 2
0dbfs = 1

  instr 1
imethod  =         p4; read panning method variable from score (p4)

;---------------- generate a source sound -------------------
a1       pinkish   0.3; pink noise
a1       reson     a1, 500, 30, 1; bandpass filtered
aPan     lfo       0.5, 1, 1; panning controlled by an lfo
aPan     =         aPan + 0.5; offset shifted +0.5
;------------------------------------------------------------

 if imethod=1 then
;------------------------ method 1 --------------------------
aPanL    =         aPan
aPanR    =         1 - aPan
;------------------------------------------------------------
 endif

 if imethod=2 then
;------------------------ method 2 --------------------------
aPanL    =       sqrt(aPan)
aPanR    =       sqrt(1 - aPan)
;------------------------------------------------------------
 endif

 if imethod=3 then
;------------------------ method 3 --------------------------
aPanL    =       sin(aPan*$M_PI_2)
aPanR    =       cos(aPan*$M_PI_2)
;------------------------------------------------------------
 endif

 if imethod=4 then
;------------------------ method 4 --------------------------
aPanL   =  sin ((aPan + 0.5) * $M_PI_2)
aPanR   =  cos ((aPan + 0.5) * $M_PI_2)
;------------------------------------------------------------
 endif

         outs    a1*aPanL, a1*aPanR ; audio sent to outputs
  endin

</CsInstruments>

<CsScore>
; 4 notes one after the other to demonstrate 4 different methods of panning
;p1 p2  p3   p4(method)
i 1 0   4.5  1
i 1 5   4.5  2
i 1 10  4.5  3
i 1 15  4.5  4
e
</CsScore>

</CsoundSynthesizer>
```

An opcode called pan2 exist which makes panning slightly easier for us to implement simple panning employing various methods. The following example demonstrates the three methods that this opcode offers one after the other. The first is the 'equal power' method, the second 'square root' and the third is simple linear. The Csound Manual alludes to fourth method but this does not seem to function currently.

EXAMPLE 05B02.csd

```
<CsoundSynthesizer>

<CsOptions>
-odac ; activates real time sound output
</CsOptions>

<CsInstruments>
; Example by Iain McCurdy

sr = 44100
ksmps = 10
nchnls = 2
```

```
0dbfs = 1

  instr 1
imethod        =         p4 ; read panning method variable from score (p4)
;----------------------- generate a source sound ------------------------
aSig           pinkish   0.5               ; pink noise
aSig           reson     aSig, 500, 30, 1 ; bandpass filtered
;-----------------------------------------------------------------------

;--------------------------- pan the signal ---------------------------
aPan           lfo       0.5, 1, 1        ; panning controlled by an lfo
aPan           =         aPan + 0.5       ; DC shifted + 0.5
aSigL, aSigR   pan2      aSig, aPan, imethod; create stereo panned output
;-----------------------------------------------------------------------

               outs      aSigL, aSigR     ; audio sent to outputs
  endin

</CsInstruments>

<CsScore>
; 3 notes one after the other to demonstrate 3 methods used by pan2
;p1 p2  p3   p4
i 1  0  4.5   0 ; equal power (harmonic)
i 1  5  4.5   1 ; square root method
i 1 10  4.5   2 ; linear
e
</CsScore>

</CsoundSynthesizer>
```

3-D BINAURAL ENCODING

3-D binaural simulation is availalable in a number of opcodes that make use of spectral data files that provide information about the filtering and inter-aural delay effects of the human head. The older one of these is hrtfer. The newer ones are hrtfmove, hrtfmove2 and hrftstat. The main parameters for controlfor the opcodes are azimuth (where the sound source in the horizontal plane relative to the direction we are facing) and elevation (the angle by which the sound deviates from this horizontal plane, either above or below). Both these parameters are defined in degrees. 'Binaural' infers that the stereo output of this opcode should be listened to using headphones so that no mixing in the air of the two channels occurs before they reach our ears.

The following example take a monophonic source sound of noise impulses and processes it using the *hrtfmove2* opcode. First of all the sound is rotated around us in the horizontal plane then it is raised above our head then dropped below us and finally returned to be straight and level in front of us.For this example to work you will need to download the files hrtf-44100-left.dat and hrtf-44100-right.dat and place them in your SADIR (see setting environment variables) or in the same directory as the .csd.

EXAMPLE 05B03.csd

```
<CsoundSynthesizer>

<CsOptions>
-odac ; activates real time sound output
</CsOptions>

<CsInstruments>
; Example by Iain McCurdy

sr = 44100
ksmps = 10
nchnls = 2
0dbfs = 1

giSine         ftgen       0, 0, 2^12, 10, 1                 ; sine wave
giLFOShape     ftgen       0, 0, 131072, 19, 0.5, 1, 180, 1 ; U-shape parabola

  instr 1
; create an audio signal (noise impulses)
krate          oscil       30,0.2,giLFOShape                 ; rate of impulses
; amplitude envelope: a repeating pulse
kEnv           loopseg     krate+3,0, 0,1, 0.1,0, 0.9,0
aSig           pinkish     kEnv                              ; noise pulses

; -- apply binaural 3d processing --
; azimuth (direction in the horizontal plane)
kAz            linseg      0, 8, 360
; elevation (held horizontal for 8 seconds then up, then down, then horizontal
kElev          linseg      0, 8,   0, 4, 90, 8, -40, 4, 0
; apply hrtfmove2 opcode to audio source - create stereo ouput
aLeft, aRight  hrtfmove2   aSig, kAz, kElev, \
                                "hrtf-44100-left.dat","hrtf-44100-right.dat"
               outs        aLeft, aRight                    ; audio to outputs
endin

</CsInstruments>

<CsScore>
i 1 0 60 ; instr 1 plays a note for 60 seconds
e
</CsScore>

</CsoundSynthesizer>
```

29. FILTERS

Audio filters can range from devices that subtly shape the tonal characteristics of a sound to ones that dramatically remove whole portions of a sound spectrum to create new sounds. Csound includes several versions of each of the commonest types of filters and some more esoteric ones also. The full list of Csound's standard filters can be found here. A list of the more specialized filters can be found here.

LOWPASS FILTERS

The first type of filter encountered is normally the lowpass filter. As its name suggests it allows lower frequencies to pass through unimpeded and therefore filters higher frequencies. The crossover frequency is normally referred to as the 'cutoff' frequency. Filters of this type do not really cut frequencies off at the cutoff point like a brick wall but instead attenuate increasingly according to a cutoff slope. Different filters offer cutoff slopes of different of steepness. Another aspect of a lowpass filter that we may be concerned with is a ripple that might emerge at the cutoff point. If this is exaggerated intentionally it is referred to as resonance or 'Q'.

In the following example, three lowpass filters filters are demonstrated: tone, butlp and moogladder. *tone* offers a quite gentle cutoff slope and therefore is better suited to subtle spectral enhancement tasks. *butlp* is based on the Butterworth filter design and produces a much sharper cutoff slope at the expense of a slightly greater CPU overhead. *moogladder* is an interpretation of an analogue filter found in a moog synthesizer – it includes a resonance control.

In the example a sawtooth waveform is played in turn through each filter. Each time the cutoff frequency is modulated using an envelope, starting high and descending low so that more and more of the spectral content of the sound is removed as the note progresses. A sawtooth waveform has been chosen as it contains strong higher frequencies and therefore demonstrates the filters characteristics well; a sine wave would be a poor choice of source sound on account of its lack of spectral richness.

EXAMPLE 05C01.csd

```
<CsoundSynthesizer>

<CsOptions>
-odac ; activates real time sound output
</CsOptions>

<CsInstruments>
; Example by Iain McCurdy

sr = 44100
ksmps = 32
nchnls = 1
0dbfs = 1

  instr 1
        prints       "tone%n"    ; indicate filter type in console
aSig    vco2         0.5, 150    ; input signal is a sawtooth waveform
kcf     expon        10000,p3,20 ; descending cutoff frequency
aSig    tone         aSig, kcf   ; filter audio signal
        out          aSig        ; filtered audio sent to output
  endin

  instr 2
        prints       "butlp%n"   ; indicate filter type in console
aSig    vco2         0.5, 150    ; input signal is a sawtooth waveform
kcf     expon        10000,p3,20 ; descending cutoff frequency
aSig    butlp        aSig, kcf   ; filter audio signal
        out          aSig        ; filtered audio sent to output
  endin

  instr 3
        prints       "moogladder%n" ; indicate filter type in console
aSig    vco2         0.5, 150       ; input signal is a sawtooth waveform
kcf     expon        10000,p3,20    ; descending cutoff frequency
aSig    moogladder   aSig, kcf, 0.9 ; filter audio signal
        out          aSig           ; filtered audio sent to output
  endin
```

```
</CsInstruments>

<CsScore>
; 3 notes to demonstrate each filter in turn
i 1 0  3; tone
i 2 4  3; butlp
i 3 8  3; moogladder
e
</CsScore>

</CsoundSynthesizer>
```

HIGHPASS FILTERS

A highpass filter is the converse of a lowpass filter; frequencies higher than the cutoff point are allowed to pass whilst those lower are attenuated. atone and buthp are the analogues of *tone* and *butlp*. Resonant highpass filters are harder to find but Csound has one in bqrez. *bqrez* is actually a multi-mode filter and could also be used as a resonant lowpass filter amongst other things. We can choose which mode we want by setting one of its input arguments appropriately. Resonant highpass is mode 1. In this example a sawtooth waveform is again played through each of the filters in turn but this time the cutoff frequency moves from low to high. Spectral content is increasingly removed but from the opposite spectral direction.

EXAMPLE 05C02.csd

```
<CsoundSynthesizer>

<CsOptions>
-odac ; activates real time sound output
</CsOptions>

<CsInstruments>
; Example by Iain McCurdy

sr = 44100
ksmps = 32
nchnls = 1
0dbfs = 1

  instr 1
        prints       "atone%n"     ; indicate filter type in console
aSig    vco2         0.2, 150      ; input signal is a sawtooth waveform
kcf     expon        20, p3, 20000 ; define envelope for cutoff frequency
aSig    atone        aSig, kcf     ; filter audio signal
        out          aSig          ; filtered audio sent to output
  endin

  instr 2
        prints       "buthp%n"     ; indicate filter type in console
aSig    vco2         0.2, 150      ; input signal is a sawtooth waveform
kcf     expon        20, p3, 20000 ; define envelope for cutoff frequency
aSig    buthp        aSig, kcf     ; filter audio signal
        out          aSig          ; filtered audio sent to output
  endin

  instr 3
        prints       "bqrez(mode:1)%n" ; indicate filter type in console
aSig    vco2         0.03, 150         ; input signal is a sawtooth waveform
kcf     expon        20, p3, 20000     ; define envelope for cutoff frequency
aSig    bqrez        aSig, kcf, 30, 1  ; filter audio signal
        out          aSig              ; filtered audio sent to output
  endin

</CsInstruments>

<CsScore>
; 3 notes to demonstrate each filter in turn
i 1 0  3 ; atone
i 2 5  3 ; buthp
i 3 10 3 ; bqrez(mode 1)
e
</CsScore>

</CsoundSynthesizer>
```

BANDPASS FILTERS

A bandpass filter allows just a narrow band of sound to pass through unimpeded and as such is a little bit like a combination of a lowpass and highpass filter connected in series. We normally expect at least one additional parameter of control: control over the width of the band of frequencies allowed to pass through, or 'bandwidth'.

In the next example cutoff frequency and bandwidth are demonstrated independently for two different bandpass filters offered by Csound. First of all a sawtooth waveform is passed through a reson filter and a butbp filter in turn while the cutoff frequency rises (bandwidth remains static). Then pink noise is passed through *reson* and *butbp* in turn again but this time the cutoff frequency remains static at 5000Hz while the bandwidth expands from 8 to 5000Hz. In the latter two notes it will be heard how the resultant sound moves from almost a pure sine tone to unpitched noise. *butbp* is obviously the Butterworth based bandpass filter. *reson* can produce dramatic variations in amplitude depending on the bandwidth value and therefore some balancing of amplitude in the output signal may be necessary if out of range samples and distortion are to be avoided. Fortunately the opcode itself includes two modes of amplitude balancing built in but by default neither of these methods are active and in this case the use of the balance opcode may be required. Mode 1 seems to work well with spectrally sparse sounds like harmonic tones while mode 2 works well with spectrally dense sounds such as white or pink noise.

EXAMPLE 05C03.csd

```
<CsoundSynthesizer>

<CsOptions>
-odac ; activates real time sound output
</CsOptions>

<CsInstruments>
; Example by Iain McCurdy

sr = 44100
ksmps = 32
nchnls = 1
0dbfs = 1

  instr 1
        prints       "reson%n"          ; indicate filter type in console
aSig    vco2         0.5, 150           ; input signal: sawtooth waveform
kcf     expon        20,p3,10000        ; rising cutoff frequency
aSig    reson        aSig,kcf,kcf*0.1,1 ; filter audio signal
        out          aSig               ; send filtered audio to output
  endin

  instr 2
        prints       "butbp%n"          ; indicate filter type in console
aSig    vco2         0.5, 150           ; input signal: sawtooth waveform
kcf     expon        20,p3,10000        ; rising cutoff frequency
aSig    butbp        aSig, kcf, kcf*0.1 ; filter audio signal
        out          aSig               ; send filtered audio to output
  endin

  instr 3
        prints       "reson%n"          ; indicate filter type in console
aSig    pinkish      0.5                ; input signal: pink noise
kbw     expon        10000,p3,8         ; contracting bandwidth
aSig    reson        aSig, 5000, kbw, 2 ; filter audio signal
        out          aSig               ; send filtered audio to output
  endin

  instr 4
        prints       "butbp%n"          ; indicate filter type in console
aSig    pinkish      0.5                ; input signal: pink noise
kbw     expon        10000,p3,8         ; contracting bandwidth
aSig    butbp        aSig, 5000, kbw    ; filter audio signal
        out          aSig               ; send filtered audio to output
  endin

</CsInstruments>

<CsScore>
i 1 0  3 ; reson - cutoff frequency rising
i 2 4  3 ; butbp - cutoff frequency rising
i 3 8  6 ; reson - bandwidth increasing
i 4 15 6 ; butbp - bandwidth increasing
e
</CsScore>

</CsoundSynthesizer>
```

COMB FILTERING

A comb filter is a special type of filter that creates a harmonically related stack of resonance peaks on an input sound file. A comb filter is really just a very short delay effect with feedback. Typically the delay times involved would be less than 0.05 seconds. Many of the comb filters documented in the Csound Manual term this delay time, 'loop time'. The fundamental of the harmonic stack of resonances produced will be 1/loop time. Loop time and the frequencies of the resonance peaks will be inversely proportionsl – as loop time get smaller, the frequencies rise. For a loop time of 0.02 seconds the fundamental resonance peak will be 50Hz, the next peak 100Hz, the next 150Hz and so on. Feedback is normally implemented as reverb time – the time taken for amplitude to drop to 1/1000 of its original level or by 60dB. This use of reverb time as opposed to feedback alludes to the use of comb filters in the design of reverb algorithms. Negative reverb times will result in only the odd numbered partials of the harmonic stack being present.

The following example demonstrates a comb filter using the vcomb opcode. This opcode allows for performance time modulation of the loop time parameter. For the first 5 seconds of the demonstration the reverb time increases from 0.1 seconds to 2 while the loop time remains constant at 0.005 seconds. Then the loop time decreases to 0.0005 seconds over 6 seconds (the resonant peaks rise in frequency), finally over the course of 10 seconds the loop time rises to 0.1 seconds (the resonant peaks fall in frequency). A repeating noise impulse is used as a source sound to best demonstrate the qualities of a comb filter.

EXAMPLE 05C04.csd

```
<CsoundSynthesizer>

<CsOptions>
-odac ;activates real time sound output
</CsOptions>

<CsInstruments>
;Example by Iain McCurdy

sr = 44100
ksmps = 32
nchnls = 1
0dbfs = 1

  instr 1
; -- generate an input audio signal (noise impulses) --
; repeating amplitude envelope:
kEnv         loopseg   1,0, 0,1,0.005,1,0.0001,0,0.9949,0
aSig         pinkish   kEnv*0.6                     ; pink noise pulses

; apply comb filter to input signal
krvt    linseg  0.1, 5, 2                           ; reverb time
alpt    expseg  0.005,5,0.005,6,0.0005,10,0.1,1,0.1 ; loop time
aRes    vcomb   aSig, krvt, alpt, 0.1               ; comb filter
        out     aRes                                ; audio to output
  endin

</CsInstruments>

<CsScore>
i 1 0 25
e
</CsScore>

</CsoundSynthesizer>
```

30. DELAY AND FEEDBACK

A delay in DSP is a special kind of buffer sometimes called a circular buffer. The length of this buffer is finite and must be declared upon initialization as it is stored in RAM. One way to think of the circular buffer is that as new items are added at the beginning of the buffer the oldest items at the end of the buffer are being 'shoved' out.

Besides their typical application for creating echo effects, delays can also be used to implement chorus, flanging, pitch shifting and filtering effects.

Csound offers many opcodes for implementing delays. Some of these offer varying degrees of quality - often balanced against varying degrees of efficiency whilst some are for quite specialized purposes.

To begin with this section is going to focus upon a pair of opcodes, delayr and delayw. Whilst not the most efficient to use in terms of the number of lines of code required, the use of *delayr* and *delayw* helps to clearly illustrate how a delay buffer works. Besides this, *delayr* and *delayw* actually offer a lot more flexibility and versatility than many of the other delay opcodes.

When using *delayr* and *delayw* the establishement of a delay buffer is broken down into two steps: reading from the end of the buffer using *delayr* (and by doing this defining the length or duration of the buffer) and then writing into the beginning of the buffer using *delayw*.

The code employed might look like this:

```
aSigOut  delayr  1
         delayw  aSigIn
```

where 'aSigIn' is the input signal written into the beginning of the buffer and 'aSigOut' is the output signal read from the end of the buffer. The fact that we declare reading from the buffer before writing to it is sometimes initially confusing but, as alluded to before, one reason this is done is to declare the length of the buffer. The buffer length in this case is 1 second and this will be the apparent time delay between the input audio signal and audio read from the end of the buffer.

The following example implements the delay described above in a .csd file. An input sound of sparse sine tone pulses is created. This is written into the delay buffer from which a new audio signal is created by read from the end of this buffer. The input signal (sometimes referred to as the dry signal) and the delay output signal (sometimes referred to as the wet signal) are mixed and set to the output. The delayed signal is attenuated with respect to the input signal.

EXAMPLE 05D01.csd

```
<CsoundSynthesizer>

<CsOptions>
-odac ; activates real time sound output
</CsOptions>

<CsInstruments>
; Example by Iain McCurdy

sr = 44100
ksmps = 32
nchnls = 1
0dbfs = 1
giSine   ftgen   0, 0, 2^12, 10, 1 ; a sine wave

  instr 1
; -- create an input signal: short 'blip' sounds --
kEnv    loopseg  0.5, 0, 0, 0,0.0005, 1 , 0.1, 0, 1.9, 0
kCps    randomh  400, 600, 0.5
aEnv    interp   kEnv
aSig    poscil   aEnv, kCps, giSine

; -- create a delay buffer --
aBufOut delayr   0.3
        delayw   aSig

; -- send audio to output (input and output to the buffer are mixed)
        out      aSig + (aBufOut*0.4)
  endin

</CsInstruments>

<CsScore>
i 1 0 25
e
</CsScore>
</CsoundSynthesizer>
```

If we mix some of the delayed signal into the input signal that is written into the buffer then we will delay some of the delayed signal thus creating more than a single echo from each input sound. Typically the sound that is fed back into the delay input is attenuated so that sound cycle through the buffer indefinitely but instead will eventually die away. We can attenuate the feedback signal by multiplying it by a value in the range zero to 1. The rapidity with which echoes will die away is defined by how close the zero this value is. The following example implements a simple delay with feedback.

EXAMPLE 05D02.csd

```
<CsoundSynthesizer>

<CsOptions>
-odac ;activates real time sound output
</CsOptions>

<CsInstruments>
;Example by Iain McCurdy

sr = 44100
ksmps = 32
nchnls = 1
0dbfs = 1

giSine   ftgen   0, 0, 2^12, 10, 1  ; a sine wave

  instr 1
; -- create an input signal: short 'blip' sounds --
kEnv    loopseg  0.5,0,0,0,0.0005,1,0.1,0,1.9,0 ; repeating envelope
kCps    randomh  400, 600, 0.5                  ; 'held' random values
aEnv    interp   kEnv                           ; a-rate envelope
aSig    poscil   aEnv, kCps, giSine             ; generate audio

; -- create a delay buffer --
iFdback =        0.7                    ; feedback ratio
aBufOut delayr   0.3                    ; read audio from end of buffer
; write audio into buffer (mix in feedback signal)
        delayw   aSig+(aBufOut*iFdback)

; send audio to output (mix the input signal with the delayed signal)
        out      aSig + (aBufOut*0.4)
  endin

</CsInstruments>

<CsScore>
i 1 0 25
e
</CsScore>

</CsoundSynthesizer>
```

Constructing a delay effect in this way is rather limited as the delay time is static. If we want to change the delay time we need to reinitialise the code that implements the delay buffer. A more flexible approach is to read audio from within the buffer using one of Csounds opcodes for 'tapping' a delay buffer, *deltap*, *deltapi*, *deltap3 or deltapx*. The opcodes are listed in order of increasing quality which also reflects an increase in computational expense. In the next example a delay tap is inserted within the delay buffer (between the *delayr* and the *delayw*) opcodes. As our delay time is modulating quite quickly we will use *deltapi* which uses linear interpolation as it rebuilds the audio signal whenever the delay time is moving. Note that this time we are not using the audio output from the *delayr* opcode as we are using the audio output from *deltapi* instead. The delay time used by *deltapi* is created by *randomi* which creates a random function of straight line segments. A-rate is used for the delay time to improve the accuracy of its values, use of k-rate would result in a noticeably poorer sound quality. You will notice that as well as modulating the time gap between echoes, this example also modulates the pitch of the echoes – if the delay tap is static within the buffer there would be no change in pitch, if is moving towards the beginning of the buffer then pitch will rise and if it is moving towards the end of the buffer then pitch will drop. This side effect has led to digital delay buffers being used in the design of many pitch shifting effects.

The user must take care that the delay time demanded from the delay tap does not exceed the length of the buffer as defined in the *delayr* line. If it does it will attempt to read data beyond the end of the RAM buffer – the results of this are unpredictable. The user must also take care that the delay time does not go below zero, in fact the minumum delay time that will be permissible will be the duration of one k cycle (ksmps/sr).

EXAMPLE 05D03.csd

```
<CsoundSynthesizer>

<CsOptions>
-odac ; activates real time sound output
</CsOptions>

<CsInstruments>
; Example by Iain McCurdy

sr = 44100
ksmps = 32
nchnls = 1
0dbfs = 1

giSine   ftgen   0, 0, 2^12, 10, 1  ; a sine wave

  instr 1
; -- create an input signal: short 'blip' sounds --
kEnv          loopseg  0.5,0,0,0,0.0005,1,0.1,0,1.9,0
aEnv          interp   kEnv
aSig          poscil   aEnv, 500, giSine

aDelayTime    randomi  0.05, 0.2, 1      ; modulating delay time
; -- create a delay buffer --
aBufOut       delayr   0.2               ; read audio from end of buffer
aTap          deltapi  aDelayTime        ; 'tap' the delay buffer
              delayw   aSig + (aTap*0.9) ; write audio into buffer

; send audio to the output (mix the input signal with the delayed signal)
              out      aSig + (aTap*0.4)
  endin

</CsInstruments>

<CsScore>
i 1 0 30
e
</CsScore>

</CsoundSynthesizer>
```

We are not limited to inserting only a single delay tap within the buffer. If we add further taps we create what is known as a multi-tap delay. The following example implements a multi-tap delay with three delay taps. Note that only the final delay (the one closest to the end of the buffer) is fed back into the input in order to create feedback but all three taps are mixed and sent to the output. There is no reason not to experiment with arrangements other than this but this one is most typical.

EXAMPLE 05D04.csd

```
<CsoundSynthesizer>

<CsOptions>
-odac ; activates real time sound output
</CsOptions>

<CsInstruments>
; Example by Iain McCurdy

sr = 44100
ksmps = 32
nchnls = 1
0dbfs = 1

giSine   ftgen   0, 0, 2^12, 10, 1 ; a sine wave

  instr 1
; -- create an input signal: short 'blip' sounds --
kEnv    loopseg  0.5,0,0,0,0.0005,1,0.1,0,1.9,0 ; repeating envelope
```

```
kCps    randomh  400, 1000, 0.5                 ; 'held' random values
aEnv    interp   kEnv                           ; a-rate envelope
aSig    poscil   aEnv, kCps, giSine             ; generate audio

; -- create a delay buffer --
aBufOut delayr   0.5                    ; read audio end buffer
aTap1   deltap   0.1373                 ; delay tap 1
aTap2   deltap   0.2197                 ; delay tap 2
aTap3   deltap   0.4139                 ; delay tap 3
        delayw   aSig + (aTap3*0.4)     ; write audio into buffer

; send audio to the output (mix the input signal with the delayed signals)
        out      aSig + ((aTap1+aTap2+aTap3)*0.4)
  endin

</CsInstruments>

<CsScore>
i 1 0 25
e
</CsScore>

</CsoundSynthesizer>
```

As mentioned at the top of this section many familiar effects are actually created from using delay buffers in various ways. We will briefly look at one of these effects: the flanger. Flanging derives from a phenomenon which occurs when the delay time becomes so short that we begin to no longer perceive individual echoes but instead a stack of harmonically related resonances are perceived the frequencies of which are in simple ratio with 1/delay_time. This effect is known as a comb filter. When the delay time is slowly modulated and the resonances shifting up and down in sympathy the effect becomes known as a flanger. In this example the delay time of the flanger is modulated using an LFO that employs a U-shaped parabola as its waveform as this seems to provide the smoothest comb filter modulations.

EXAMPLE 05D05.csd

```
<CsoundSynthesizer>

<CsOptions>
-odac ; activates real time sound output
</CsOptions>

<CsInstruments>
;Example by Iain McCurdy

sr = 44100
ksmps = 32
nchnls = 1
0dbfs = 1

giSine   ftgen   0, 0, 2^12, 10, 1                 ; a sine wave
giLFOShape  ftgen   0, 0, 2^12, 19, 0.5, 1, 180, 1 ; u-shaped parabola

  instr 1
aSig    pinkish  0.1                               ; pink noise

aMod    poscil   0.005, 0.05, giLFOShape           ; delay time LFO
iOffset =        ksmps/sr                          ; minimum delay time
kFdback linseg   0.8,(p3/2)-0.5,0.95,1,-0.95       ; feedback

; -- create a delay buffer --
aBufOut delayr   0.5                   ; read audio from end buffer
aTap    deltap3  aMod + iOffset        ; tap audio from within buffer
        delayw   aSig + (aTap*kFdback) ; write audio into buffer

; send audio to the output (mix the input signal with the delayed signal)
        out      aSig + aTap
  endin

</CsInstruments>

<CsScore>
i 1 0 25
e
</CsScore>

</CsoundSynthesizer>
```

31. REVERBERATION

Reverb is the effect a room or space has on a sound where the sound we perceive is a mixture of the direct sound and the dense overlapping echoes of that sound reflecting off walls and objects within the space.

Csound's earliest reverb opcodes are *reverb* and *nreverb*. By today's standards these sound rather crude and as a consequence modern Csound users tend to prefer the more recent opcodes *freeverb* and *reverbsc*.

The typical way to use a reverb is to run as a effect throughout the entire Csound performance and to send it audio from other instruments to which it adds reverb. This is more efficient than initiating a new reverb effect for every note that is played. This arrangement is a reflection of how a reverb effect would be used with a mixing desk in a conventional studio. There are several methods of sending audio from sound producing instruments to the reverb instrument, three of which will be introduced in the coming examples

The first method uses Csound's global variables so that an audio variable created in one instrument and be read in another instrument. There are several points to highlight here. First the global audio variable that is use to send audio the reverb instrument is initialized to zero (silence) in the header area of the orchestra.

This is done so that if no sound generating instruments are playing at the beginning of the performance this variable still exists and has a value. An error would result otherwise and Csound would not run. When audio is written into this variable in the sound generating instrument it is added to the current value of the global variable.

This is done in order to permit polyphony and so that the state of this variable created by other sound producing instruments is not overwritten. Finally it is important that the global variable is cleared (assigned a value of zero) when it is finished with at the end of the reverb instrument. If this were not done then the variable would quickly 'explode' (get astronomically high) as all previous instruments are merely adding values to it rather that redeclaring it. Clearing could be done simply by setting to zero but the *clear* opcode might prove useful in the future as it provides us with the opportunity to clear many variables simultaneously.

This example uses the freeverb opcode and is based on a plugin of the same name. Freeverb has a smooth reverberant tail and is perhaps similar in sound to a plate reverb. It provides us with two main parameters of control: 'room size' which is essentially a control of the amount of internal feedback and therefore reverb time, and 'high frequency damping' which controls the amount of attenuation of high frequencies. Both there parameters should be set within the range 0 to 1. For room size a value of zero results in a very short reverb and a value of 1 results in a very long reverb. For high frequency damping a value of zero provides minimum damping of higher frequencies giving the impression of a space with hard walls, a value of 1 provides maximum high frequency damping thereby giving the impression of a space with soft surfaces such as thick carpets and heavy curtains.

EXAMPLE 05E01.csd

```
<CsoundSynthesizer>

<CsOptions>
-odac ; activates real time sound output
</CsOptions>

<CsInstruments>
;Example by Iain McCurdy

sr =  44100
ksmps = 32
nchnls = 2
0dbfs = 1

gaRvbSend    init      0 ; global audio variable initialized to zero

  instr 1 ; sound generating instrument (sparse noise bursts)
kEnv         loopseg   0.5,0,0,1,0.003,1,0.0001,0,0.9969,0 ; amp. env.
aSig         pinkish   kEnv              ; noise pulses
             outs      aSig, aSig        ; audio to outs
```

```
iRvbSendAmt  =         0.8               ; reverb send amount (0 - 1)
; add some of the audio from this instrument to the global reverb send variable
gaRvbSend    =         gaRvbSend + (aSig * iRvbSendAmt)
  endin

  instr 5 ; reverb - always on
kroomsize    init      0.85          ; room size (range 0 to 1)
kHFDamp      init      0.5           ; high freq. damping (range 0 to 1)
; create reverberated version of input signal (note stereo input and output)
aRvbL,aRvbR  freeverb  gaRvbSend, gaRvbSend,kroomsize,kHFDamp
             outs      aRvbL, aRvbR ; send audio to outputs
             clear     gaRvbSend    ; clear global audio variable
  endin

</CsInstruments>

<CsScore>
i 1 0 300 ; noise pulses (input sound)
i 5 0 300 ; start reverb
e
</CsScore>

</CsoundSynthesizer>
```

The next example uses Csound's zak patching system to send audio from one instrument to another. The zak system is a little like a patch bay you might find in a recording studio. Zak channels can be a, k or i-rate. These channels will be addressed using numbers so it will be important to keep track of what each numbered channel is used for. Our example will be very simple in that we will only be using one zak audio channel. Before using any of the zak opcodes for reading and writing data we must initialize zak storage space. This is done in the orchestra header area using the zakinit opcode. This opcode initializes both a and k rate channels; we must intialize at least one of each even if we don't require both.

```
zakinit    1, 1
```

The audio from the sound generating instrument is mixed into a zak audio channel the zawm opcode like this:

```
zawm    aSig * iRvbSendAmt, 1
```

This channel is read from in the reverb instrument using the zar opcode like this:

```
aInSig  zar   1
```

Because audio is begin mixed into our zak channel but it is never redefined (only mixed into) it needs to be cleared after we have finished with it. This is accomplished at the bottom of the reverb instrument using the zacl opcode like this:

```
zacl      0, 1
```

This example uses the reverbsc opcode. It too has a stereo input and output. The arguments that define its character are feedback level and cutoff frequency. Feedback level should be in the range zero to 1 and controls reverb time. Cutoff frequency should be within the range of human hearing (20Hz -20kHz) and less than the Nyqvist frequency (sr/2) - it controls the cutoff frequencies of low pass filters within the algorithm.

EXAMPLE 05E02.csd

```
<CsoundSynthesizer>

<CsOptions>
-odac ; activates real time sound output
</CsOptions>

<CsInstruments>
; Example by Iain McCurdy

sr =  44100
ksmps = 32
nchnls = 2
0dbfs = 1

; initialize zak space  - one a-rate and one k-rate variable.
; We will only be using the a-rate variable.
             zakinit   1, 1

  instr 1 ; sound generating instrument - sparse noise bursts
kEnv         loopseg   0.5,0, 0,1,0.003,1,0.0001,0,0.9969,0 ; amp. env.
```

```
aSig         pinkish   kEnv       ; pink noise pulses
             outs      aSig, aSig ; send audio to outputs
iRvbSendAmt  =         0.8        ; reverb send amount (0 - 1)
; write to zak audio channel 1 with mixing
             zawm      aSig*iRvbSendAmt, 1
  endin

  instr 5 ; reverb - always on
aInSig       zar       1    ; read first zak audio channel
kFblvl       init      0.88 ; feedback level - i.e. reverb time
kFco         init      8000 ; cutoff freq. of a filter within the reverb
; create reverberated version of input signal (note stereo input and output)
aRvbL,aRvbR  reverbsc  aInSig, aInSig, kFblvl, kFco
             outs      aRvbL, aRvbR ; send audio to outputs
             zacl      0, 1         ; clear zak audio channels
  endin

</CsInstruments>

<CsScore>
i 1 0 10 ; noise pulses (input sound)
i 5 0 12 ; start reverb
e
</CsScore>

</CsoundSynthesizer>
```

reverbsc contains a mechanism to modulate delay times internally which has the effect of harmonically blurring sounds the longer they are reverberated. This contrasts with *freeverb*'s rather static reverberant tail. On the other hand *screverb*'s tail is not as smooth as that of *freeverb,* inidividual echoes are sometimes discernible so it may not be as well suited to the reverberation of percussive sounds. Also be aware that as well as reducing the reverb time, the feedback level parameter reduces the overall amplitude of the effect to the point where a setting of 1 will result in silence from the opcode.

A more recent option for sending sound from instrument to instrument in Csound is to use the *chn...* opcodes. These opcodes can also be used to allow Csound to interface with external programs using the software bus and the Csound API.

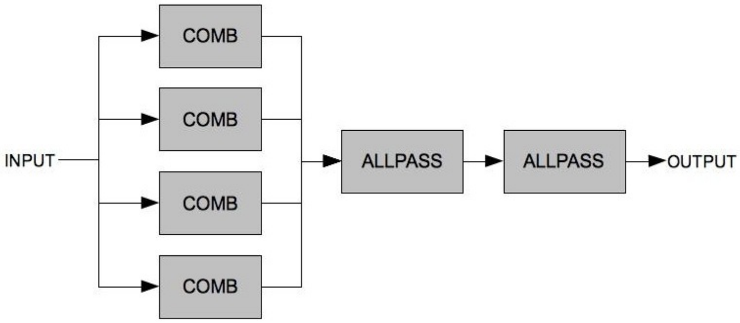
EXAMPLE 05E03.csd

```
<CsoundSynthesizer>

<CsOptions>
-odac ; activates real time sound output
</CsOptions>

<CsInstruments>
; Example by Iain McCurdy

sr =  44100
ksmps = 32
nchnls = 2
0dbfs = 1

  instr 1 ; sound generating instrument - sparse noise bursts
kEnv         loopseg   0.5,0, 0,1,0.003,1,0.0001,0,0.9969,0 ; amp. envelope
aSig         pinkish   kEnv                                 ; noise pulses
             outs      aSig, aSig                           ; audio to outs
iRvbSendAmt  =         0.4                        ; reverb send amount (0 - 1)
;write audio into the named software channel:
             chnmix    aSig*iRvbSendAmt, "ReverbSend"
  endin

  instr 5 ; reverb (always on)
aInSig       chnget    "ReverbSend"   ; read audio from the named channel
kTime        init      4              ; reverb time
kHDif        init      0.5            ; 'high frequency diffusion' (0 - 1)
aRvb         nreverb   aInSig, kTime, kHDif ; create reverb signal
outs         aRvb, aRvb               ; send audio to outputs
             chnclear  "ReverbSend"   ; clear the named channel
endin

</CsInstruments>

<CsScore>
i 1 0 10 ; noise pulses (input sound)
i 5 0 12 ; start reverb
e
```

```
</CsScore>

</CsoundSynthesizer>
```

THE SCHROEDER REVERB DESIGN

Many reverb algorithms including Csound's freeverb, reverb and reverbn are based on what is known as the Schroeder reverb design. This was a design proposed in the early 1960s by the physicist Manfred Schroeder. In the Schroeder reverb a signal is passed into four parallel comb filters the outputs of which are summed and then passed through two allpass filters as shown in the diagram below. Essentially the comb filters provide the body of the reverb effect and the allpass filters smear their resultant sound to reduce ringing artefacts the comb filters might produce. More modern designs might extent the number of filters used in an attempt to create smoother results. The freeverb opcode employs eight parallel comb filters followed by four series allpass filters on each channel. The two main indicators of poor implementations of the Schoeder reverb are individual echoes being excessively apparent and ringing artefacts. The results produced by the freeverb opcode are very smooth but a criticism might be that it is lacking in character and is more suggestive of a plate reverb than of a real room.



The next example implements the basic Schroeder reverb with four parallel comb filters followed by three series allpass filters. This also proves a useful exercise in routing audio signals within Csound. Perhaps the most crucial element of the Schroeder reverb is the choice of loop times for the comb and allpass filters – careful choices here should obviate the undesirable artefacts mentioned in the previous paragraph. If loop times are too long individual echoes will become apparent, if they are too short the characteristic ringing of comb filters will become apparent. If loop times between filters differ too much the outputs from the various filters will not fuse. It is also important that the loop times are prime numbers so that echoes between different filters do not reinforce each other. It may also be necessary to adjust loop times when implementing very short reverbs or very long reverbs. The duration of the reverb is effectively determined by the reverb times for the comb filters. There is ceratinly scope for experimentation with the design of this example and exploration of settings other than the ones suggested here.

This example consists of five instruments. The fifth instrument implements the reverb algorithm described above. The first four instruments act as a kind of generative drum machine to provide source material for the reverb. Generally sharp percussive sounds provide the sternest test of a reverb effect. Instrument 1 triggers the various synthesized drum sounds (bass drum, snare and closed hi-hat) produced by instruments 2 to 4.

EXAMPLE 05E04.csd

```
<CsoundSynthesizer>

<CsOptions>
-odac -m0
; activate real time sound output and suppress note printing
</CsOptions>

<CsInstruments>
;Example by Iain McCurdy

sr =  44100
ksmps = 1
nchnls = 2
0dbfs = 1

giSine       ftgen       0, 0, 2^12, 10, 1 ; a sine wave
```

```
gaRvbSend    init        0                 ; global audio variable initialized
giRvbSendAmt init        0.4               ; reverb send amount (range 0 - 1)

  instr 1 ; trigger drum hits
ktrigger    metro       5                  ; rate of drum strikes
kdrum       random      2, 4.999           ; randomly choose which drum to hit
            schedkwhen  ktrigger, 0, 0, kdrum, 0, 0.1 ; strike a drum
  endin

  instr 2 ; sound 1 - bass drum
iamp        random      0, 0.5               ; amplitude randomly chosen
p3          =           0.2                  ; define duration for this sound
aenv        line        1,p3,0.001           ; amplitude envelope (percussive)
icps        exprand     30                   ; cycles-per-second offset
kcps        expon       icps+120,p3,20       ; pitch glissando
aSig        oscil       aenv*0.5*iamp,kcps,giSine  ; oscillator
            outs        aSig, aSig           ; send audio to outputs
gaRvbSend   =           gaRvbSend + (aSig * giRvbSendAmt) ; add to send
  endin

  instr 3 ; sound 3 - snare
iAmp        random      0, 0.5                   ; amplitude randomly chosen
p3          =           0.3                      ; define duration
aEnv        expon       1, p3, 0.001             ; amp. envelope (percussive)
aNse        noise       1, 0                     ; create noise component
iCps        exprand     20                       ; cps offset
kCps        expon       250 + iCps, p3, 200+iCps ; create tone component gliss.
aJit        randomi     0.2, 1.8, 10000          ; jitter on freq.
aTne        oscil       aEnv, kCps*aJit, giSine  ; create tone component
aSig        sum         aNse*0.1, aTne           ; mix noise and tone components
aRes        comb        aSig, 0.02, 0.0035       ; comb creates a 'ring'
aSig        =           aRes * aEnv * iAmp       ; apply env. and amp. factor
            outs        aSig, aSig               ; send audio to outputs
gaRvbSend   =           gaRvbSend + (aSig * giRvbSendAmt); add to send
  endin

  instr 4 ; sound 4 - closed hi-hat
iAmp        random      0, 1.5               ; amplitude randomly chosen
p3          =           0.1                  ; define duration for this sound
aEnv        expon       1,p3,0.001           ; amplitude envelope (percussive)
aSig        noise       aEnv, 0              ; create sound for closed hi-hat
aSig        buthp       aSig*0.5*iAmp, 12000 ; highpass filter sound
aSig        buthp       aSig,          12000 ; -and again to sharpen cutoff
            outs        aSig, aSig           ; send audio to outputs
gaRvbSend   =           gaRvbSend + (aSig * giRvbSendAmt) ; add to send
  endin


  instr 5 ; schroeder reverb - always on
; read in variables from the score
kRvt        =           p4
kMix        =           p5

; print some information about current settings gleaned from the score
            prints      "Type:"
            prints      p6
            prints      "\\nReverb Time:%2.1f\\nDry/Wet Mix:%2.1f\\n\\n",p4,p5

; four parallel comb filters
a1          comb        gaRvbSend, kRvt, 0.0297; comb filter 1
a2          comb        gaRvbSend, kRvt, 0.0371; comb filter 2
a3          comb        gaRvbSend, kRvt, 0.0411; comb filter 3
a4          comb        gaRvbSend, kRvt, 0.0437; comb filter 4
asum        sum         a1,a2,a3,a4 ; sum (mix) the outputs of all comb filters

; two allpass filters in series
a5          alpass      asum, 0.1, 0.005 ; send mix through first allpass filter
aOut        alpass      a5, 0.1, 0.02291 ; send 1st allpass through 2nd allpass

amix        ntrpol      gaRvbSend, aOut, kMix  ; create a dry/wet mix
            outs        amix, amix             ; send audio to outputs
            clear       gaRvbSend              ; clear global audio variable
  endin

</CsInstruments>

<CsScore>
; room reverb
i 1  0 10                     ; start drum machine trigger instr
i 5  0 11 1 0.5 "Room Reverb" ; start reverb

; tight ambience
i 1 11 10                          ; start drum machine trigger instr
i 5 11 11 0.3 0.9 "Tight Ambience" ; start reverb

; long reverb (low in the mix)
```

```
i 1 22 10                                      ; start drum machine
i 5 22 15 5 0.1 "Long Reverb (Low In the Mix)" ; start reverb

; very long reverb (high in the mix)
i 1 37 10                                           ; start drum machine
i 5 37 25 8 0.9 "Very Long Reverb (High in the Mix)" ; start reverb
e
</CsScore>

</CsoundSynthesizer>
```

This chapter has introduced some of the more recent Csound opcodes for delay-line based reverb algorithms which in most situations can be used to provide high quality and efficient reverberation. Convolution offers a whole new approach for the creation of realistic reverbs that imitate actual spaces - this technique is demonstrated in the Convolution chapter.

32. AM / RM / WAVESHAPING

An introduction as well as some background theory of amplitude modulation, ring modulation and waveshaping is given in the fourth chapter entitled "sound-synthesis". As all of these techniques merely modulate the amplitude of a signal in a variety of ways, they can also be used for the modification of non-synthesized sound. In this chapter we will explore amplitude modulation, ring modulation and waveshaping as applied to non-synthesized sound.[1]

AMPLITUDE MODULATION

With "sound-synthesis", the principle of AM was shown as a amplitude multiplication of two sine oscillators. Later we've used a more complex modulators, to generate more complex spectrums. The principle also works very well with sound-files (samples) or live-audio-input.

Karlheinz Stockhausens *"Mixtur für Orchester, vier Sinusgeneratoren und vier Ringmodulatoren "* (1964) was the first piece which used analog ringmodulation (AM without DC-offset) to alter the acoustic instruments pitch in realtime during a live-performance. The word ringmodulation inherites from the analog *four-diode circuit* which was arranged in a "ring".

In the following example shows how this can be done digitally in Csound. In this case a sound-file works as the *carrier* which is modulated by a *sine-wave-osc*. The result sounds like old 'Harald Bode' pitch-shifters from the 1960's.

Example: 05F01.csd

```
<CsoundSynthesizer>
<CsOptions>
-o dac
</CsOptions>
<CsInstruments>

sr = 48000
ksmps = 32
nchnls = 1
0dbfs = 1


instr 1   ; Ringmodulation
aSine1 poscil 0.8, p4, 1
aSample diskin2 "fox.wav", 1, 0, 1, 0, 32
out aSine1*aSample
endin

</CsInstruments>
<CsScore>
f 1 0 1024 10 1 ; sine

i 1 0 2 400
i 1 2 2 800
i 1 4 2 1600
i 1 6 2 200
i 1 8 2 2400
e
</CsScore>
</CsoundSynthesizer>
; written by Alex Hofmann (Mar. 2011)
```

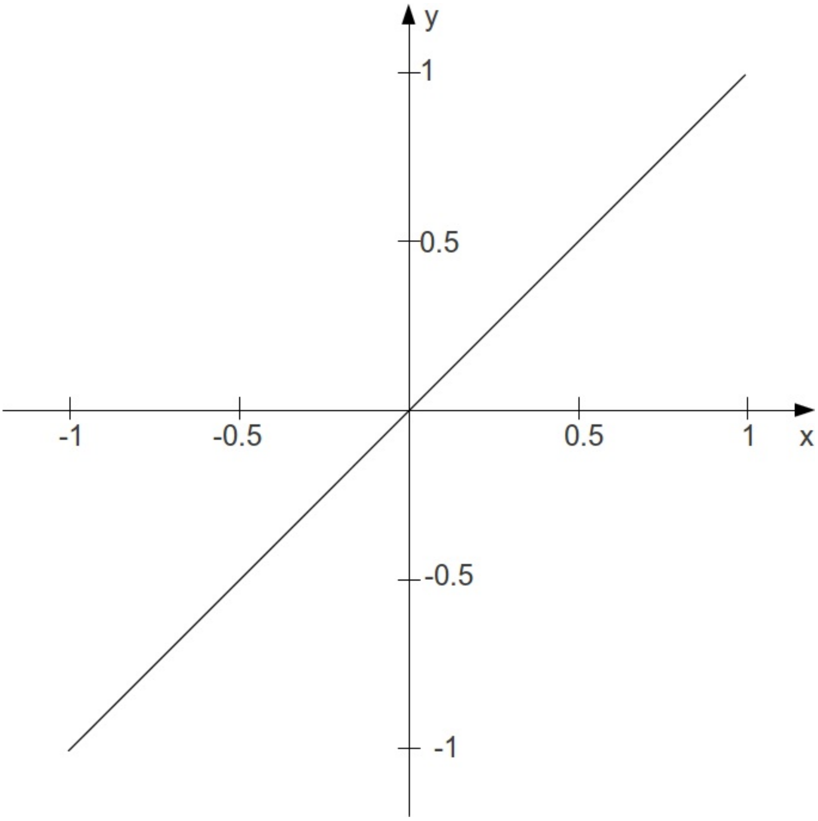
WAVESHAPING

In chapter 04E waveshaping has been described as a method of applying a transfer function to an incoming signal. It has been discussed that the table which stores the transfer function must be read with an interpolating table reader to avoid degradation of the signal. On the other hand, degradation can be a nice thing for sound modification. So let us start with this branch here.

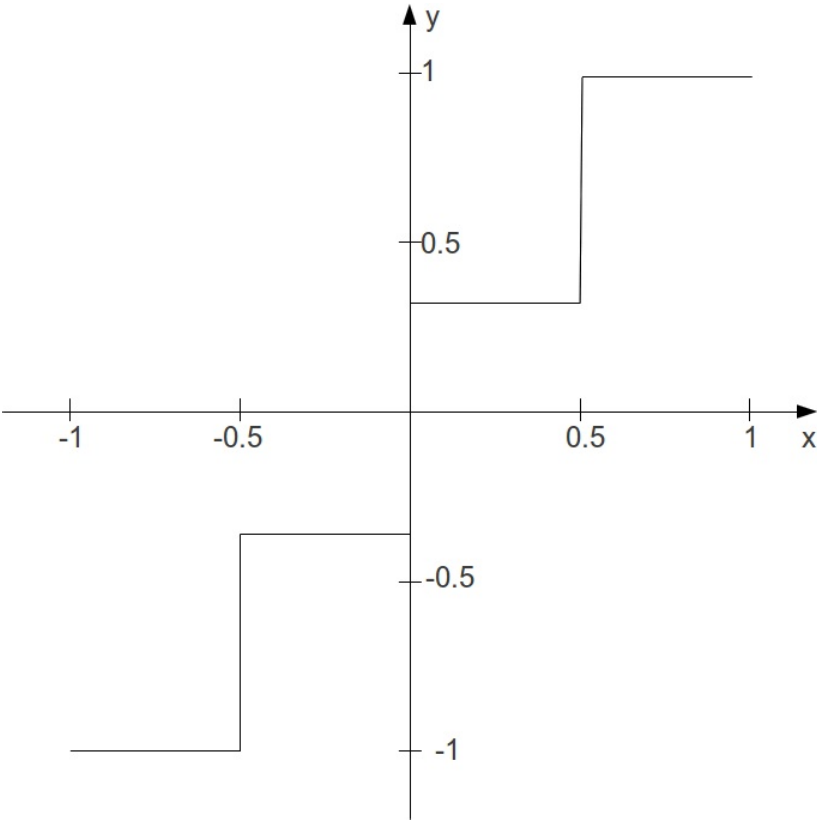
Bit Depth Reduction

If the transfer function itself is linear, but the table of the function is small, and no interpolation is applied to the amplitude as index to the table, in effect the bit depth is reduced. For a function table of size 4, a line becomes a staircase:

Bit Depth = high



Bit Depth = 2



This is the sounding result:

EXAMPLE 05G01.csd

```
<CsoundSynthesizer>
<CsOptions>
-odac
</CsOptions>
<CsInstruments>
sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

giTrnsFnc ftgen 0, 0, 4, -7, -1, 3, 1

instr 1
aAmp      soundin   "fox.wav"
aIndx     =         (aAmp + 1) / 2
aWavShp   table     aIndx, giTrnsFnc, 1
          outs      aWavShp, aWavShp
endin

</CsInstruments>
<CsScore>
i 1 0 2.767
</CsScore>
</CsoundSynthesizer>
;example by joachim heintz
```

Transformation and Distortion

In general, the transformation of sound in applying waveshaping depends on the transfer function. The following example applies at first a table which does not change the sound at all, because the function just says $y = x$. The second one leads aready to a heavy distortion, though "just" the samples between an amplitude of -0.1 and +0.1 are erased. Tables 3 to 7 apply some chebychev functions which are well known from waveshaping synthesis. Finally, tables 8 and 9 approve that even a meaningful sentence and a nice music can regarded as noise ...

```
<CsoundSynthesizer>
<CsOptions>
```

```
-odac
</CsOptions>
<CsInstruments>
sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

giNat   ftgen 1, 0, 2049, -7, -1, 2048, 1
giDist  ftgen 2, 0, 2049, -7, -1, 1024, -.1, 0, .1, 1024, 1
giCheb1 ftgen 3, 0, 513, 3, -1, 1, 0, 1
giCheb2 ftgen 4, 0, 513, 3, -1, 1, -1, 0, 2
giCheb3 ftgen 5, 0, 513, 3, -1, 1, 0, 3, 0, 4
giCheb4 ftgen 6, 0, 513, 3, -1, 1, 1, 0, 8, 0, 4
giCheb5 ftgen 7, 0, 513, 3, -1, 1, 3, 20, -30, -60, 32, 48
giFox   ftgen 8, 0, -121569, 1, "fox.wav", 0, 0, 1
giGuit  ftgen 9, 0, -235612, 1, "ClassGuit.wav", 0, 0, 1

instr 1
iTrnsFnc  =         p4
kEnv      linseg    0, .01, 1, p3-.2, 1, .01, 0
aL, aR    soundin   "ClassGuit.wav"
aIndxL    =         (aL + 1) / 2
aWavShpL  tablei    aIndxL, iTrnsFnc, 1
aIndxR    =         (aR + 1) / 2
aWavShpR  tablei    aIndxR, iTrnsFnc, 1
          outs      aWavShpL*kEnv, aWavShpR*kEnv
endin

</CsInstruments>
<CsScore>
i 1 0 7 1 ;natural though waveshaping
i 1 + . 2 ;rather heavy distortion
i 1 + . 3 ;chebychev for 1st partial
i 1 + . 4 ;chebychev for 2nd partial
i 1 + . 5 ;chebychev for 3rd partial
i 1 + . 6 ;chebychev for 4th partial
i 1 + . 7 ;after dodge/jerse p.136
i 1 + . 8 ;fox
i 1 + . 9 ;guitar
</CsScore>
</CsoundSynthesizer>
;example by joachim heintz
```

Instead of using the "self-built" method which has been described here, you can use the Csound opcode distort. It performs the actual waveshaping process and gives a nice control about the amount of distortion in the *kdist* parameter. Here is a simple example:[2]

```
<CsoundSynthesizer>
<CsOptions>
-odac
</CsOptions>
<CsInstruments>
sr     = 44100
ksmps  = 32
nchnls = 2
0dbfs  = 1

gi1 ftgen 1,0,257,9,.5,1,270 ;sinoid (also the next)
gi2 ftgen 2,0,257,9,.5,1,270,1.5,.33,90,2.5,.2,270,3.5,.143,90
gi3 ftgen 3,0,129,7,-1,128,1 ;actually natural
gi4 ftgen 4,0,129,10,1 ;sine
gi5 ftgen 5,0,129,10,1,0,1,0,1,0,1,0,1 ;odd partials
gi6 ftgen 6,0,129,21,1 ;white noise
gi7 ftgen 7,0,129,9,.5,1,0 ;half sine
gi8 ftgen 8,0,129,7,1,64,1,0,-1,64,-1 ;square wave

instr 1
ifn       =         p4
ivol      =         p5
kdist     line      0, p3, 1 ;increase the distortion over p3
aL, aR    soundin   "ClassGuit.wav"
aout1     distort   aL, kdist, ifn
aout2     distort   aR, kdist, ifn
          outs      aout1*ivol, aout2*ivol
endin
</CsInstruments>
<CsScore>
i 1 0 7 1 1
i . + . 2 .3
i . + . 3 1
i . + . 4 .5
i . + . 5 .15
i . + . 6 .04
```

```
i . + . 7 .02
i . + . 8 .02
</CsScore>
</CsoundSynthesizer>
;example by joachim heintz
```

---

1. This is the same for Granular Synthesis which can either be "pure" synthesis or applied so sampled sound.^
2. Have a look at Iain McCurdy's Realtime example (which has also been ported to CsoundQt by René Jopi) for 'distort' for a more interactive exploration of the opcode.^

33. GRANULAR SYNTHESIS

This chapter will focus upon granular synthesis used as a DSP technique upon recorded sound files and will introduce techniques including time stretching, time compressing and pitch shifting. The emphasis will be upon asynchronous granulation. For an introduction to synchronous granular synthesis using simple waveforms please refer to chapter 04F.

Csound offers a wide range of opcodes for sound granulation. Each has its own strengths and weaknesses and suitability for a particular task. Some are easier to use than others, some, such as granule and partikkel, are extremely complex and are, at least in terms of the number of input arguments they demand, amongst Csound's most complex opcodes.

SNDWARP - TIME STRETCHING AND PITCH SHIFTING

sndwarp may not be Csound's newest or most advanced opcode for sound granulation but it is quite easy to use and is certainly up to the task of time stretching and pitch shifting. sndwarp has two modes by which we can modulate time stretching characteristics, one in which we define a 'stretch factor', a value of 2 defining a stretch to twice the normal length, and the other in which we directly control a pointer into the file. The following example uses sndwarp's first mode to produce a sequence of time stretches and pitch shifts. An overview of each procedure will be printed to the terminal as it occurs. sndwarp does not allow for k-rate modulation of grain size or density so for this level we need to look elsewhere.

You will need to make sure that a sound file is available to sndwarp via a GEN01 function table. You can replace the one used in this example with one of your own by replacing the reference to 'ClassicalGuitar.wav'. This sound file is stereo therefore instrument 1 uses the stereo version of sndwarp. 'sndwarpst'. A mismatch between the number of channels in the sound file and the version of sndwarp used will result in playback at an unexpected pitch. You will also need to give GEN01 an appropriate size that will be able to contain your chosen sound file. You can calculate the table size you will need by multiplying the duration of the sound file (in seconds) by the sample rate - for stereo files this value should be doubled - and then choose the next power of 2 above this value. You can download the sample used in the example at http://www.iainmccurdy.org/csoundrealtimeexamples/sourcematerials/ClassicalGuitar.wav.

sndwarp describes grain size as 'window size' and it is defined in samples so therefore a window size of 44100 means that grains will last for 1s each (when sample rate is set at 44100). Window size randomization (irandw) adds a random number within that range to the duration of each grain. As these two parameters are closely related it is sometime useful to set irandw to be a fraction of window size. If irandw is set to zero we will get artefacts associated with synchronous granular synthesis.

sndwarp (along with many of Csound's other granular synthesis opcodes) requires us to supply it with a window function in the form of a function table according to which it will apply an amplitude envelope to each grain. By using different function tables we can alternatively create softer grains with gradual attacks and decays (as in this example), with more of a percussive character (short attack, long decay) or 'gate'-like (short attack, long sustain, short decay).

EXAMPLE 05G01.csd

```
<CsoundSynthesizer>

<CsOptions>
-odac -m0
; activate real-time audio output and suppress printing
</CsOptions>

<CsInstruments>
; example written by Iain McCurdy

sr = 44100
ksmps = 16
nchnls = 2
0dbfs = 1

; waveform used for granulation
giSound  ftgen 1,0,2097152,1,"ClassicalGuitar.wav",0,0,0
```

```
; window function - used as an amplitude envelope for each grain
; (first half of a sine wave)
giWFn   ftgen 2,0,16384,9,0.5,1,0

  instr 1
kamp        =          0.1
ktimewarp   expon      p4,p3,p5  ; amount of time stretch, 1=none 2=double
kresample   line       p6,p3,p7  ; pitch change 1=none 2=+1oct
ifn1        =          giSound   ; sound file to be granulated
ifn2        =          giWFn     ; window shaped used to envelope every grain
ibeg        =          0
iwsize      =          3000      ; grain size (in sample)
irandw      =          3000      ; randomization of grain size range
ioverlap    =          50        ; density
itimemode   =          0         ; 0=stretch factor 1=pointer
            prints     p8        ; print a description
aSigL,aSigR sndwarpst  kamp,ktimewarp,kresample,ifn1,ibeg, \
                                 iwsize,irandw,ioverlap,ifn2,itimemode
            outs       aSigL,aSigR
  endin

</CsInstruments>

<CsScore>
;p3 = stretch factor begin / pointer location begin
;p4 = stretch factor end / pointer location end
;p5 = resample begin (transposition)
;p6 = resample end (transposition)
;p7 = procedure description
;p8 = description string
; p1 p2   p3 p4 p5  p6    p7    p8
i 1  0    10 1  1   1     1     "No time stretch. No pitch shift."
i 1  10.5 10 2  2   1     1     "%nTime stretch x 2."
i 1  21   20 1  20  1     1     \
                 "%nGradually increasing time stretch factor from x 1 to x 20."
i 1  41.5 10 1  1   2     2     "%nPitch shift x 2 (up 1 octave)."
i 1  52   10 1  1   0.5   0.5   "%nPitch shift x 0.5 (down 1 octave)."
i 1  62.5 10 1  1   4     0.25  \
 "%nPitch shift glides smoothly from 4 (up 2 octaves) to 0.25 (down 2 octaves)."
i 1  73   15 4  4   1     1     \
"%nA chord containing three transpositions: \
                                   unison, +5th, +10th. (x4 time stretch.)"
i 1  73   15 4  4   [3/2] [3/2] ""
i 1  73   15 4  4   3     3     ""
e
</CsScore>

</CsoundSynthesizer>
```

The next example uses sndwarp's other timestretch mode with which we explicitly define a pointer position from where in the source file grains shall begin. This method allows us much greater freedom with how a sound will be time warped; we can even freeze movement an go backwards in time - something that is not possible with timestretching mode.

This example is self generative in that instrument 2, the instrument that actually creates the granular synthesis textures, is repeatedly triggered by instrument 1. Instrument 2 is triggered once every 12.5s and these notes then last for 40s each so will overlap. Instrument 1 is played from the score for 1 hour so this entire process will last that length of time. Many of the parameters of granulation are chosen randomly when a note begins so that each note will have unique characteristics. The timestretch is created by a line function: the start and end points of which are defined randomly when the note begins. Grain/window size and window size randomization are defined randomly when a note begins - notes with smaller window sizes will have a fuzzy airy quality wheres notes with a larger window size will produce a clearer tone. Each note will be randomly transposed (within a range of +/- 2 octaves) but that transposition will be quantized to a rounded number of semitones - this is done as a response to the equally tempered nature of source sound material used.

Each entire note is enveloped by an amplitude envelope and a resonant lowpass filter in each case encasing each note under a smooth arc. Finally a small amount of reverb is added to smooth the overall texture slightly

EXAMPLE 05G02.csd

```
<CsoundSynthesizer>

<CsOptions>
-odac
```

```
</CsOptions>

<CsInstruments>
;example written by Iain McCurdy

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

; the name of the sound file used is defined as a string variable -
; - as it will be used twice in the code.
; This simplifies adapting the orchestra to use a different sound file
gSfile = "ClassicalGuitar.wav"

; waveform used for granulation
giSound  ftgen 1,0,2097152,1,gSfile,0,0,0

; window function - used as an amplitude envelope for each grain
giWFn   ftgen 2,0,16384,9,0.5,1,0

seed 0 ; seed the random generators from the system clock
gaSendL init 0  ; initialize global audio variables
gaSendR init 0

  instr 1 ; triggers instrument 2
ktrigger  metro   0.08         ;metronome of triggers. One every 12.5s
schedkwhen ktrigger,0,0,2,0,40 ;trigger instr. 2 for 40s
  endin

  instr 2 ; generates granular synthesis textures
;define the input variables
ifn1        =          giSound
ilen        =          nsamp(ifn1)/sr
iPtrStart   random     1,ilen-1
iPtrTrav    random     -1,1
ktimewarp   line       iPtrStart,p3,iPtrStart+iPtrTrav
kamp        linseg     0,p3/2,0.2,p3/2,0
iresample   random     -24,24.99
iresample   =          semitone(int(iresample))
ifn2        =          giWFn
ibeg        =          0
iwsize      random     400,10000
irandw      =          iwsize/3
ioverlap    =          50
itimemode   =          1
; create a stereo granular synthesis texture using sndwarp
aSigL,aSigR sndwarpst  kamp,ktimewarp,iresample,ifn1,ibeg,\
                              iwsize,irandw,ioverlap,ifn2,itimemode
; envelope the signal with a lowpass filter
kcf         expseg     50,p3/2,12000,p3/2,50
aSigL       moogvcf2    aSigL, kcf, 0.5
aSigR       moogvcf2    aSigR, kcf, 0.5
; add a little of our audio signals to the global send variables -
; - these will be sent to the reverb instrument (2)
gaSendL     =          gaSendL+(aSigL*0.4)
gaSendR     =          gaSendR+(aSigR*0.4)
            outs       aSigL,aSigR
  endin

  instr 3 ; reverb (always on)
aRvbL,aRvbR reverbsc   gaSendL,gaSendR,0.85,8000
            outs       aRvbL,aRvbR
;clear variables to prevent out of control accumulation
            clear      gaSendL,gaSendR
  endin

</CsInstruments>

<CsScore>
; p1 p2 p3
i 1  0  3600 ; triggers instr 2
i 3  0  3600 ; reverb instrument
e
</CsScore>

</CsoundSynthesizer>
```

GRANULE - CLOUDS OF SOUND

The granule opcode is one of Csound's most complex opcodes requiring up to 22 input arguments in order to function. Only a few of these arguments are available during performance (k-rate) so it is less well suited for real-time modulation, for real-time a more nimble implementation such as syncgrain, fog, or grain3 would be recommended. For more complex realtime granular techniques, the partikkel opcode can be used. The granule opcode as used here, proves itself ideally suited at the production of massive clouds of granulated sound in which individual grains are often completed indistinguishable. There are still two important k-rate variables that have a powerful effect on the texture created when they are modulated during a note, they are: grain gap - effectively density - and grain size which will affect the clarity of the texture - textures with smaller grains will sound fuzzier and airier, textures with larger grains will sound clearer. In the following example transeg envelopes move the grain gap and grain size parameters through a variety of different states across the duration of each note.

With granule we define a number a grain streams for the opcode using its 'ivoice' input argument. This will also have an effect on the density of the texture produced. Like sndwarp's first timestretching mode, granule also has a stretch ratio parameter. Confusingly it works the other way around though, a value of 0.5 will slow movement through the file by 1/2, 2 will double is and so on. Increasing grain gap will also slow progress through the sound file. granule also provides up to four pitch shift voices so that we can create chord-like structures without having to use more than one iteration of the opcode. We define the number of pitch shifting voices we would like to use using the 'ipshift' parameter. If this is given a value of zero, all pitch shifting intervals will be ignored and grain-by-grain transpositions will be chosen randomly within the range +/-1 octave. granule contains built-in randomizing for several of it parameters in order to easier facilitate asynchronous granular synthesis. In the case of grain gap and grain size randomization these are defined as percentages by which to randomize the fixed values.

Unlike Csound's other granular synthesis opcodes, granule does not use a function table to define the amplitude envelope for each grain, instead attack and decay times are defined as percentages of the total grain duration using input arguments. The sum of these two values should total less than 100.

Five notes are played by this example. While each note explores grain gap and grain size in the same way each time, different permutations for the four pitch transpositions are explored in each note. Information about what these transpositions are, are printed to the terminal as each note begins.

EXAMPLE 05G03.csd

```
<CsoundSynthesizer>

<CsOptions>
-odac -m0
; activate real-time audio output and suppress note printing
</CsOptions>

<CsInstruments>
; example written by Iain McCurdy

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

;waveforms used for granulation
giSoundL ftgen 1,0,1048576,1,"ClassicalGuitar.wav",0,0,1
giSoundR ftgen 2,0,1048576,1,"ClassicalGuitar.wav",0,0,2

seed 0; seed the random generators from the system clock
gaSendL init 0
gaSendR init 0

  instr 1 ; generates granular synthesis textures
            prints     p9
;define the input variables
kamp        linseg     0,1,0.1,p3-1.2,0.1,0.2,0
ivoice      =          64
iratio      =          0.5
imode       =          1
ithd        =          0
ipshift     =          p8
igskip      =          0.1
igskip_os   =          0.5
ilength     =          nsamp(giSoundL)/sr
kgap        transeg    0,20,14,4,       5,8,8,     8,-10,0,    15,0,0.1
igap_os     =          50
kgsize      transeg    0.04,20,0,0.04,  5,-4,0.01, 8,0,0.01,   15,5,0.4
igsize_os   =          50
```

```
iatt        =          30
idec        =          30
iseedL      =          0
iseedR      =          0.21768
ipitch1     =          p4
ipitch2     =          p5
ipitch3     =          p6
ipitch4     =          p7
;create the granular synthesis textures; one for each channel
aSigL  granule  kamp,ivoice,iratio,imode,ithd,giSoundL,ipshift,igskip,\
     igskip_os,ilength,kgap,igap_os,kgsize,igsize_os,iatt,idec,iseedL,\
     ipitch1,ipitch2,ipitch3,ipitch4
aSigR  granule  kamp,ivoice,iratio,imode,ithd,giSoundR,ipshift,igskip,\
     igskip_os,ilength,kgap,igap_os,kgsize,igsize_os,iatt,idec,iseedR,\
     ipitch1,ipitch2,ipitch3,ipitch4
;send a little to the reverb effect
gaSendL     =          gaSendL+(aSigL*0.3)
gaSendR     =          gaSendR+(aSigR*0.3)
            outs       aSigL,aSigR
  endin

  instr 2 ; global reverb instrument (always on)
; use reverbsc opcode for creating reverb signal
aRvbL,aRvbR reverbsc   gaSendL,gaSendR,0.85,8000
            outs       aRvbL,aRvbR
;clear variables to prevent out of control accumulation
            clear      gaSendL,gaSendR
  endin

</CsInstruments>

<CsScore>
; p4 = pitch 1
; p5 = pitch 2
; p6 = pitch 3
; p7 = pitch 4
; p8 = number of pitch shift voices (0=random pitch)
; p1 p2  p3   p4  p5    p6    p7    p8    p9
i 1  0   48   1   1     1     1     4    "pitches: all unison"
i 1  +   .    1   0.5   0.25  2     4    \
  "%npitches: 1(unison) 0.5(down 1 octave) 0.25(down 2 octaves) 2(up 1 octave)"
i 1  +   .    1   2     4     8     4    "%npitches: 1 2 4 8"
i 1  +   .    1   [3/4] [5/6] [4/3] 4    "%npitches: 1 3/4 5/6 4/3"
i 1  +   .    1   1     1     1     0    "%npitches: all random"

i 2 0 [48*5+2]; reverb instrument
e
</CsScore>

</CsoundSynthesizer>
```

GRAIN DELAY EFFECT

Granular techniques can be used to implement a flexible delay effect, where we can do transposition, time modification and disintegration of the sound into small particles, all within the delay effect itself. To implement this effect, we record live audio into a buffer (Csound table), and let the granular synthesizer/generator read sound for the grains from this buffer. We need a granular synthesizer that allows manual control over the read start point for each grain, since the relationship between the write position and the read position in the buffer determines the delay time. We've used the fof2 opcode for this purpose here.

```
<CsoundSynthesizer>
<CsOptions>
; activate real-time audio output and suppress note printing
</CsOptions>

<CsInstruments>
;example by Oeyvind Brandtsegg

sr = 44100
ksmps = 512
nchnls = 2
0dbfs = 1

; empty table, live audio input buffer used for granulation
giTablen  = 131072
giLive    ftgen 0,0,giTablen,2,0

; sigmoid rise/decay shape for fof2, half cycle from bottom to top
giSigRise ftgen 0,0,8192,19,0.5,1,270,1

; test sound
giSample  ftgen 0,0,524288,1,"fox.wav", 0,0,0

instr 1
; test sound, replace with live input
  a1      loscil 1, 1, giSample, 1
     outch 1, a1
          chnmix a1, "liveAudio"
```

```
endin

instr 2
; write live input to buffer (table)
  a1      chnget "liveAudio"
  gkstart tablewa giLive, a1, 0
  if gkstart < giTablen goto end
  gkstart = 0
  end:
  a0      = 0
          chnset a0, "liveAudio"
endin

instr 3
; delay parameters
  kDelTim = 0.5   ; delay time in seconds (max 2.8 seconds)
  kFeed   = 0.8
; delay time random dev
  kTmod   = 0.2
  kTmod   rnd31 kTmod, 1
  kDelTim = kDelTim+kTmod
; delay pitch random dev
  kFmod   linseg 0, 1, 0, 1, 0.1, 2, 0, 1, 0
  kFmod   rnd31 kFmod, 1
 ; grain delay processing
  kamp   = ampdbfs(-8)
  kfund   = 25 ; grain rate
  kform   = (1+kFmod)*(sr/giTablen) ; grain pitch transposition
  koct    = 0
  kband   = 0
  kdur    = 2.5 / kfund ; duration relative to grain rate
  kris    = 0.5*kdur
  kdec    = 0.5*kdur
  kphs    = (gkstart/giTablen)-(kDelTim/(giTablen/sr)) ; calculate grain phase based on delay time
  kgliss  = 0
  a1     fof2 1, kfund, kform, koct, kband, kris, kdur, kdec, 100, \
      giLive, giSigRise, 86400, kphs, kgliss
          outch     2, a1*kamp
          chnset a1*kFeed, "liveAudio"
endin

</CsInstruments>
<CsScore>
i 1 0 20
i 2 0 20
i 3 0 20
e
</CsScore>
</CsoundSynthesizer>
```

CONCLUSION

Two contrasting opcodes for granular synthesis have been considered in this chapter but this is in no way meant to suggest that these are the best, in fact it is strongly recommended to explore all of Csound's other opcodes as they each have their own unique character. The syncgrain family of opcodes (including also syncloop and diskgrain) are deceptively simple as their k-rate controls encourages further abstractions of grain manipulation, fog is designed for FOF synthesis type synchronous granulation but with sound files and partikkel offers a comprehensive control of grain characteristics on a grain-by-grain basis inspired by Curtis Roads' encyclopedic book on granular synthesis 'Microsound'.

34. CONVOLUTION

Convolution is a mathematical procedure whereby one function is modified by another. Applied to audio, one of these functions might be a sound file or a stream of live audio whilst the other will be, what is referred to as, an impulse response file; this could actually just be another shorter sound file. The longer sound file or live audio stream will be modified by the impulse response so that the sound file will be imbued with certain qualities of the impulse response. It is important to be aware that convolution is a far from trivial process and that realtime performance may be a frequent consideration. Effectively every sample in the sound file to be processed will be multiplied in turn by every sample contained within the impulse response file. Therefore, for a 1 second impulse response at a sampling frequency of 44100 hertz, each and every sample of the input sound file or sound stream will undergo 44100 multiplication operations. Expanding upon this even further, for 1 second's worth of a convolution procedure this will result in 44100 x 44100 (or 1,944,810,000) multiplications. This should provide some insight into the processing demands of a convolution procedure and also draw attention to the efficiency cost of using longer impulse response files.

The most common application of convolution in audio processing is reverberation but convolution is equally adept at, for example, imitating the filtering and time smearing characteristics of vintage microphones, valve amplifiers and speakers. It is also used sometimes to create more unusual special effects. The strength of convolution based reverbs is that they implement acoustic imitations of actual spaces based upon 'recordings' of those spaces. All the quirks and nuances of the original space will be retained. Reverberation algorithms based upon networks of comb and allpass filters create only idealised reverb responses imitating spaces that don't actually exist. The impulse response is a little like a 'fingerprint' of the space. It is perhaps easier to manipulate characteristics such as reverb time and high frequency diffusion (i.e. lowpass filtering) of the reverb effect when using a Schroeder derived algorithm using comb and allpass filters but most of these modification are still possible, if not immediately apparent, when implementing reverb using convolution. The quality of a convolution reverb is largely dependent upon the quality of the impulse response used. An impulse response recording is typically achieved by recording the reverberant tail that follows a burst of white noise. People often employ techniques such as bursting balloons to achieve something approaching a short burst of noise. Crucially the impulse sound should not excessively favour any particular frequency or exhibit any sort of resonance. More modern techniques employ a sine wave sweep through all the audible frequencies when recording an impulse response. Recorded results using this technique will normally require further processing in order to provide a usable impulse response file and this approach will normally be beyond the means of a beginner.

Many commercial, often expensive, implementations of convolution exist both in the form of software and hardware but fortunately Csound provides easy access to convolution for free. Csound currently lists six different opcodes for convolution, convolve (convle), cross2, dconv, ftconv, ftmorf and pconvolve. convolve (convle) and dconv are earlier implementations and are less suited to realtime operation, cross2 relates to FFT-based cross synthesis and ftmorf is used to morph between similar sized function table and is less related to what has been discussed so far, therefore in this chapter we shall focus upon just two opcodes, pconvolve and ftconv.

PCONVOLVE

pconvolve is perhaps the easiest of Csound's convolution opcodes to use and the most useful in a realtime application. It uses the uniformly partitioned (hence the 'p') overlap-save algorithm which permits convolution with very little delay (latency) in the output signal. The impulse response file that it uses is referenced directly, i.e. it does not have to be previously loaded into a function table, and multichannel files are permitted. The impulse response file can be any standard sound file acceptable to Csound and does not need to be pre-analysed as is required by convolve. Convolution procedures through their very nature introduce a delay in the output signal but pconvolve minimises this using the algorithm mentioned above. It will still introduce some delay but we can control this using the opcode's 'ipartitionsize' input argument. What value we give this will require some consideration and perhaps some experimentation as choosing a high partition size will result in excessively long delays (only an issue in realtime work) whereas very low partition sizes demand more from the CPU and too low a size may result in buffer under-runs and interrupted realtime audio. Bear in mind still that realtime CPU performance will depend heavily on the length of the impulse file. The partition size argument is actually an optional argument and if omitted it will default to whatever the software buffer size is as defined by the -b command line flag. If we specify the partition size explicitly however, we can use this information to delay the input audio (after it has been used by pconvolve) so that it can be realigned in time with the latency affected audio output from pconvolve - this will be essential in creating a 'wet/dry' mix in a reverb effect. Partition size is defined in sample frames therefore if we specify a partition size of 512, the delay resulting from the convolution procedure will be 512/sr (sample rate).

In the following example a monophonic drum loop sample undergoes processing through a convolution reverb implemented using pconvolve which in turn uses two different impulse files. The first file is a more conventional reverb impulse file taken in a stairwell whereas the second is a recording of the resonance created by striking a terracota bowl sharply. If you wish to use the three sound files I have used in creating this example the mono input sound file is here and the two stereo sound files used as impulse responses are here and here. You can, of course, replace them with ones of your own but remain mindful of mono/stereo/multichannel integrity.

EXAMPLE 05H01.csd

```
<CsoundSynthesizer>

<CsOptions>
-odac
</CsOptions>

<CsInstruments>

sr     =  44100
ksmps  =  512
nchnls =  2
0dbfs  =  1

gasig init 0

 instr 1 ; sound file player
gasig           diskin2   p4,1,0,1
 endin

 instr 2 ; convolution reverb
; Define partion size.
; Larger values require less CPU but result in more latency.
; Smaller values produce lower latency but may cause -
; - realtime performance issues
ipartitionsize =   256
ar1,ar2         pconvolve gasig, p4,ipartitionsize
; create a delayed version of the input signal that will sync -
; - with convolution output
adel            delay     gasig,ipartitionsize/sr
; create a dry/wet mix
aMixL           ntrpol    adel,ar1*0.1,p5
aMixR           ntrpol    adel,ar2*0.1,p5
                outs      aMixL,aMixR
gasig         =         0
 endin

</CsInstruments>

<CsScore>
; instr 1. sound file player
;    p4=input soundfile
; instr 2. convolution reverb
;    p4=impulse response file
;    p5=dry/wet mix (0 - 1)

i 1 0 8.6 "loop.wav"
i 2 0 10 "Stairwell.wav" 0.3

i 1 10 8.6 "loop.wav"
i 2 10 10 "Dish.wav" 0.8
e
</CsScore>

</CsoundSynthesizer>
```

FTCONV

ftconv (abbreviated from 'function table convolution) is perhaps slightly more complicated to use than pconvolve but offers additional options. The fact that ftconv utilises an impulse response that we must first store in a function table rather than directly referencing a sound file stored on disk means that we have the option of performing transformations upon the audio stored in the function table before it is employed by ftconv for convolution. This example begins just as the previous example: a mono drum loop sample is convolved first with a typical reverb impulse response and then with an impulse response derived from a terracotta bowl. After twenty seconds the contents of the function tables containing the two impulse responses are reversed by calling a UDO (instrument 3) and the convolution procedure is repeated, this time with a 'backwards reverb' effect. When the reversed version is performed the dry signal is delayed further before being sent to the speakers so that it appears that the reverb impulse sound occurs at the culmination of the reverb build-up. This additional delay is switched on or off via p6 from the score. As with pconvolve, ftconv performs the convolution process in overlapping partitions to minimise latency. Again we can minimise the size of these partitions and therefore the latency but at the cost of CPU efficiency. ftconv's documentation refers to this partition size as 'iplen' (partition length). ftconv offers further facilities to work with multichannel files beyond stereo. When doing this it is suggested that you use GEN52 which is designed for this purpose. GEN01 seems to work fine, at least up to stereo, provided that you do not defer the table size definition (size=0). With ftconv we can specify the actual length of the impulse response - it will probably be shorter than the power-of-2 sized function table used to store it - and this action will improve realtime efficiency. This optional argument is defined in sample frames and defaults to the size of the impulse response function table.

EXAMPLE 05H02.csd

```
<CsoundSynthesizer>

<CsOptions>
-odac
</CsOptions>

<CsInstruments>

sr     =  44100
ksmps  =  512
nchnls =  2
0dbfs  =  1

; impulse responses stored as stereo GEN01 function tables
giStairwell ftgen 1,0,131072,1,"Stairwell.wav",0,0,0
giDish  ftgen 2,0,131072,1,"Dish.wav",0,0,0

gasig init 0

; reverse function table UDO
 opcode tab_reverse,0,i
ifn             xin
iTabLen         =               ftlen(ifn)
iTableBuffer    ftgentmp        0,0,-iTabLen,-2, 0
icount          =               0
loop:
ival            table           iTabLen-icount-1, ifn
                tableiw         ival,icount,iTableBuffer
                loop_lt         icount,1,iTabLen,loop
icount          =               0
loop2:
ival            table           icount,iTableBuffer
                tableiw  ival,icount,ifn
                loop_lt         icount,1,iTabLen,loop2
 endop

 instr 3 ; reverse the contents of a function table
          tab_reverse p4
 endin

 instr 1 ; sound file player
gasig           diskin2   p4,1,0,1
 endin

 instr 2 ; convolution reverb
; buffer length
iplen = 1024
; derive the length of the impulse response
iirlen = nsamp(p4)
ar1,ar2 ftconv gasig, p4, iplen,0, iirlen
; delay compensation. Add extra delay if reverse reverb is used.
adel            delay     gasig,(iplen/sr) + ((iirlen/sr)*p6)
; create a dry/wet mix
aMixL   ntrpol    adel,ar1*0.1,p5
```

```
aMixR   ntrpol    adel,ar2*0.1,p5
        outs      aMixL,aMixR
gasig         =         0
 endin

</CsInstruments>

<CsScore>
; instr 1. sound file player
;    p4=input soundfile
; instr 2. convolution reverb
;    p4=impulse response file
;    p5=dry/wet mix (0 - 1)
;    p6=reverse reverb switch (0=off,1=on)
; instr 3. reverse table contents
;    p4=function table number

; 'stairwell' impulse response
i 1 0 8.5 "loop.wav"
i 2 0 10 1 0.3 0

; 'dish' impulse response
i 1 10 8.5 "loop.wav"
i 2 10 10 2 0.8 0

; reverse the impulse responses
i 3 20 0 1
i 3 20 0 2

; 'stairwell' impulse response (reversed)
i 1 21 8.5 "loop.wav"
i 2 21 10 1 0.5 1

; 'dish' impulse response (reversed)
i 1 31 8.5 "loop.wav"
i 2 31 10 2 0.5 1

e
</CsScore>

</CsoundSynthesizer
```

Suggested avenues for further exploration with ftconv could be applying envelopes to, filtering and time stretching and compressing the function table stored impulse files before use in convolution.

The impulse responses I have used here are admittedly of rather low quality and whilst it is always recommended to maintain as high standards of sound quality as possible the user should not feel restricted from exploring the sound transformation possibilities possible form whatever source material they may have lying around. Many commercial convolution algorithms demand a proprietary impulse response format inevitably limiting the user to using the impulse responses provided by the software manufacturers but with Csound we have the freedom to use any sound we like.

35. FOURIER TRANSFORMATION / SPECTRAL PROCESSING

A fourier transformation (FT) is used to transfer an audio-signal from time-domain to the frequency-domain. This can, for instance, be used to analyze and visualize the spectrum of the signal appearing in a certain time span. Fourier transform and subsequent manipulations in the frequency domain open a wide area of interesting sound transformations, like time stretching, pitch shifting and much more.

HOW DOES IT WORK?

The mathematician J.B. Fourier (1768-1830) developed a method to approximate unknown functions by using trigonometric functions. The advantage of this was, that the properties of the trigonometric functions (sin & cos) were well-known and helped to describe the properties of the unknown function.

In music, a fourier transformed signal is decomposed into its sum of sinoids. In easy words: Fourier transform is the opposite of additive synthesis. Ideally, a sound can be splitted by Fourier transformation into its partial components, and resynthesized again by adding these components.

Because of sound beeing represented as discrete samples in the computer, the computer implementation calculates a discrete Fourier transform (DFT). As each transformation needs a certain number of samples, one main decision in performing DFT is about the number of samples used. The analysis of the frequency components is better the more samples are used for it. But as samples are progression in time, a caveat must be found for each FT in music between either better time resolution (fewer samples) or better frequency resolution (more samples). A typical value for FT in music is to have about 20-100 "snapshots" per second (which can be compared to the single frames in a film or video).

At a sample rate of 48000 samples per second, these are about 500-2500 samples for one frame or window. The standard method for DFT in computer music works with window sizes which are power-of-two samples long, for instance 512, 1024 or 2048 samples. The reason for this restriction is that DFT for these power-of-two sized frames can be calculated much faster. So it is called Fast Fourier Transform (FFT), and this is the standard implementation of the Fourier transform in audio applications.

HOW TO DO IT IN CSOUND?

As usual, there is not just one way to work with FFT and spectral processing in Csound. There are several families of opcodes. Each family can be very useful for a specific approach of working in the frequency domain. Have a look at the Spectral Processing overview in the Csound Manual. This introduction will focus on the so-called "Phase Vocoder Streaming" opcodes (all these opcodes begin with the charcters "pvs") which came into Csound by the work of Richard Dobson, Victor Lazzarini and others. They are designed to work in realtime in the frequency domain in Csound; and indeed they are not just very fast but also easier to use than FFT implementations in some other applications.

CHANGING FROM TIME-DOMAIN TO FREQUENCY-DOMAIN

For dealing with signals in the frequency domain, the pvs opcodes implement a new signal type, the **f-signals**. Csound shows the type of a variable in the first letter of its name. Each audio signal starts with an **a**, each control signal with a **k**, and so each signal in the frequency domain used by the pvs-opcodes starts with an **f**.

There are several ways to create an f-signal. The most common way is to convert an audio signal to a frequency signal. The first example covers two typical situations:

- the audio signal derives from playing back a soundfile from the hard disc (instr 1)
- the audio signal is the live input (instr 2)

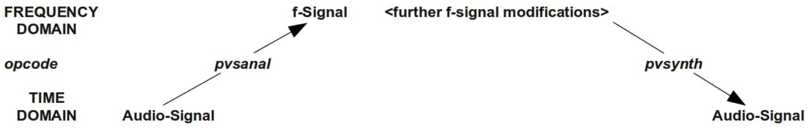
(Be careful - the example can produce a feedback three seconds after the start. Best results are with headphones.)

EXAMPLE 05I01.csd [1]

```
<CsoundSynthesizer>
<CsOptions>
-i adc -o dac
</CsOptions>
<CsInstruments>
;Example by Joachim Heintz
;uses the file "fox.wav" (distributed with the Csound Manual)
sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

;general values for fourier transform
gifftsiz  =         1024
gioverlap =         256
giwintyp  =         1 ;von hann window

instr 1 ;soundfile to fsig
asig      soundin   "fox.wav"
fsig      pvsanal   asig, gifftsiz, gioverlap, gifftsiz*2, giwintyp
aback     pvsynth   fsig
          outs      aback, aback
endin

instr 2 ;live input to fsig
          prints    "LIVE INPUT NOW!%n"
ain       inch      1 ;live input from channel 1
fsig      pvsanal   ain, gifftsiz, gioverlap, gifftsiz, giwintyp
alisten   pvsynth   fsig
          outs      alisten, alisten
endin

</CsInstruments>
<CsScore>
i 1 0 3
i 2 3 10
</CsScore>
</CsoundSynthesizer>
```

You should hear first the "fox.wav" sample, and then, the slightly delayed live input signal. The delay depends first on the general settings for realtime input (ksmps, -b and -B: see chapter 2D). But second, there is also a delay added by the FFT. The window size here is 1024 samples, so the additional delay is 1024/44100 = 0.023 seconds. If you change the window size *gifftsiz* to 2048 or to 512 samples, you should get a larger or shorter delay. - So for realtime applications, the decision about the FFT size is not only a question "better time resolution versus better frequency resolution", but it is also a question of tolerable latency.

What happens in the example above? At first, the audio signal (*asig, ain*) is being analyzed and transformed in an f-signal. This is done via the opcode pvsanal. Then nothing happens but transforming the frequency domain signal back into an audio signal. This is called inverse Fourier transformation (IFT or IFFT) and is done by the opcode pvsynth.[2] In this case, it is just a test: to see if everything works, to hear the results of different window sizes, to check the latency. But potentially you can insert any other pvs opcode(s) in between this entrance and exit:



PITCH SHIFTING

Simple pitch shifting can be done by the opcode pvscale. All the frequency data in the f-signal are scaled by a certain value. Multiplying by 2 results in transposing an octave upwards; multiplying by 0.5 in transposing an octave downwards. For accepting cent values instead of ratios as input, the cent opcode can be used.

EXAMPLE 05I02.csd

```
<CsoundSynthesizer>
<CsOptions>
-odac
</CsOptions>
<CsInstruments>
;example by joachim heintz
sr = 44100
ksmps = 32
nchnls = 1
0dbfs = 1

gifftsize =         1024
gioverlap =         gifftsize / 4
giwinsize =         gifftsize
giwinshape =        1; von-Hann window

instr 1 ;scaling by a factor
ain       soundin  "fox.wav"
fftin     pvsanal  ain, gifftsize, gioverlap, giwinsize, giwinshape
fftscal   pvscale  fftin, p4
aout      pvsynth  fftscal
          out      aout
endin

instr 2 ;scaling by a cent value
ain       soundin  "fox.wav"
fftin     pvsanal  ain, gifftsize, gioverlap, giwinsize, giwinshape
fftscal   pvscale  fftin, cent(p4)
aout      pvsynth  fftscal
          out      aout/3
endin

</CsInstruments>
<CsScore>
i 1 0 3 1; original pitch
i 1 3 3 .5; octave lower
i 1 6 3 2 ;octave higher
i 2 9 3 0
i 2 9 3 400 ;major third
i 2 9 3 700 ;fifth
e
</CsScore>
</CsoundSynthesizer>
```

Pitch shifting via FFT resynthesis is very simple in general, but more or less complicated in detail. With speech for instance, there is a problem because of the formants. If you simply scale the frequencies, the formants are shifted, too, and the sound gets the typical "Mickey-Mousing" effect. There are some parameters in the *pvscale* opcode, and some other pvs-opcodes which can help to avoid this, but the result always depends on the individual sounds and on your ideas.

TIME STRETCH/COMPRESS

As the Fourier transformation seperates the spectral information from the progression in time, both elements can be varied independently. Pitch shifting via the *pvscale* opcode, as in the previous example, is independent from the speed of reading the audio data. The complement is changing the time without changing the pitch: time stretching or time compression.

The simplest way to alter the speed of a sampled sound is using pvstanal (which is new in Csound 5.13). This opcode transforms a sound which is stored in a function table, in an f-signal, and time manipulations are simply done by altering the *ktimescal* parameter.

Example 05I03.csd

```
<CsoundSynthesizer>
<CsOptions>
-odac
</CsOptions>
<CsInstruments>
;example by joachim heintz
sr = 44100
ksmps = 32
nchnls = 1
0dbfs = 1

;store the sample "fox.wav" in a function table (buffer)
```

```
gifil     ftgen     0, 0, 0, 1, "fox.wav", 0, 0, 1

;general values for the pvstanal opcode
giamp     =         1 ;amplitude scaling
gipitch   =         1 ;pitch scaling
gidet     =         0 ;onset detection
giwrap    =         0 ;no loop reading
giskip    =         0 ;start at the beginning
gifftsiz  =         1024 ;fft size
giovlp    =         gifftsiz/8 ;overlap size
githresh  =         0 ;threshold

instr 1 ;simple time stretching / compressing
fsig      pvstanal  p4, giamp, gipitch, gifil, gidet, giwrap, giskip,
                    gifftsiz, giovlp, githresh
aout      pvsynth   fsig
          out       aout
endin

instr 2 ;automatic scratching
kspeed    randi     2, 2, 2 ;speed randomly between -2 and 2
kpitch    randi     p4, 2, 2 ;pitch between 2 octaves lower or higher
fsig      pvstanal  kspeed, 1, octave(kpitch), gifil
aout      pvsynth   fsig
aenv      linen     aout, .003, p3, .1
          out       aout
endin

</CsInstruments>
<CsScore>
;         speed
i 1 0 3   1
i . + 10   .33
i . + 2   3
s
i 2 0 10 0;random scratching without ...
i . 11 10 2 ;... and with pitch changes
</CsScore>
</CsoundSynthesizer>
```

CROSS SYNTHESIS

Working in the frequency domain makes it possible to combine or "cross" the spectra of two sounds. As the Fourier transform of an analysis frame results in a frequency and an amplitude value for each frequency "bin", there are many different ways of performing cross synthesis. The most common methods are:

- Combine the amplitudes of sound A with the frequencies of sound B. This is the classical phase vocoder approach. If the frequencies are not completely from sound B, but can be scaled between A and B, the crossing is more flexible and adjustable to the sounds being used. This is what pvsvoc does.
- Combine the frequencies of sound A with the amplitudes of sound B. Give more flexibility by scaling the amplitudes between A and B: pvscross.
- Get the frequencies from sound A. Multiply the amplitudes of A and B. This can be described as spectral filtering. pvsfilter gives a flexible portion of this filtering effect.

This is an example for phase vocoding. It is nice to have speech as sound A, and a rich sound, like classical music, as sound B. Here the "fox" sample is being played at half speed and "sings" through the music of sound B:

EXAMPLE 05I04.csd

```
<CsoundSynthesizer>
<CsOptions>
-odac
</CsOptions>
<CsInstruments>
;example by joachim heintz
sr = 44100
ksmps = 32
nchnls = 1
0dbfs = 1

;store the samples in function tables (buffers)
gifilA    ftgen     0, 0, 0, 1, "fox.wav", 0, 0, 1
gifilB    ftgen     0, 0, 0, 1, "ClassGuit.wav", 0, 0, 1


;general values for the pvstanal opcode
```

```
giamp     =         1 ;amplitude scaling
gipitch   =         1 ;pitch scaling
gidet     =         0 ;onset detection
giwrap    =         1 ;loop reading
giskip    =         0 ;start at the beginning
gifftsiz  =         1024 ;fft size
giovlp    =         gifftsiz/8 ;overlap size
githresh  =         0 ;threshold

instr 1
;read "fox.wav" in half speed and cross with classical guitar sample
fsigA     pvstanal  .5, giamp, gipitch, gifilA, gidet, giwrap, giskip,
                     gifftsiz, giovlp, githresh
fsigB     pvstanal  1, giamp, gipitch, gifilB, gidet, giwrap, giskip,
                     gifftsiz, giovlp, githresh
fvoc      pvsvoc    fsigA, fsigB, 1, 1
aout      pvsynth   fvoc
aenv      linen     aout, .1, p3, .5
          out       aout
endin

</CsInstruments>
<CsScore>
i 1 0 11
</CsScore>
</CsoundSynthesizer>
```

The next example introduces *pvscross*:

EXAMPLE 05I05.csd

```
<CsoundSynthesizer>
<CsOptions>
-odac
</CsOptions>
<CsInstruments>
;example by joachim heintz
sr = 44100
ksmps = 32
nchnls = 1
0dbfs = 1

;store the samples in function tables (buffers)
gifilA    ftgen     0, 0, 0, 1, "BratscheMono.wav", 0, 0, 1
gifilB    ftgen     0, 0, 0, 1, "fox.wav", 0, 0, 1

;general values for the pvstanal opcode
giamp     =         1 ;amplitude scaling
gipitch   =         1 ;pitch scaling
gidet     =         0 ;onset detection
giwrap    =         1 ;loop reading
giskip    =         0 ;start at the beginning
gifftsiz  =         1024 ;fft size
giovlp    =         gifftsiz/8 ;overlap size
githresh  =         0 ;threshold

instr 1
;cross viola with "fox.wav" in half speed
fsigA     pvstanal  1, giamp, gipitch, gifilA, gidet, giwrap, giskip,
                    gifftsiz, giovlp, githresh
fsigB     pvstanal  .5, giamp, gipitch, gifilB, gidet, giwrap, giskip,
                     gifftsiz, giovlp, githresh
fcross    pvscross  fsigA, fsigB, 0, 1
aout      pvsynth   fcross
aenv      linen     aout, .1, p3, .5
          out       aout
endin

</CsInstruments>
<CsScore>
i 1 0 11
</CsScore>
</CsoundSynthesizer>
```

The last example shows spectral filtering via *pvsfilter*. The well-known "fox" (sound A) is now filtered by the viola (sound B). Its resulting intensity depends on the amplitudes of sound B, and if the amplitudes are strong enough, you hear a resonating effect:

EXAMPLE 05I06.csd

```
<CsoundSynthesizer>
<CsOptions>
```

```
-odac
</CsOptions>
<CsInstruments>
;example by joachim heintz
sr = 44100
ksmps = 32
nchnls = 1
0dbfs = 1

;store the samples in function tables (buffers)
gifilA    ftgen     0, 0, 0, 1, "fox.wav", 0, 0, 1
gifilB    ftgen     0, 0, 0, 1, "BratscheMono.wav", 0, 0, 1

;general values for the pvstanal opcode
giamp     =         1 ;amplitude scaling
gipitch   =         1 ;pitch scaling
gidet     =         0 ;onset detection
giwrap    =         1 ;loop reading
giskip    =         0 ;start at the beginning
gifftsiz  =         1024 ;fft size
giovlp    =         gifftsiz/4 ;overlap size
githresh  =         0 ;threshold

instr 1
;filters "fox.wav" (half speed) by the spectrum of the viola (double speed)
fsigA     pvstanal  .5, giamp, gipitch, gifilA, gidet, giwrap, giskip,
                     gifftsiz, giovlp, githresh
fsigB     pvstanal  2, 5, gipitch, gifilB, gidet, giwrap, giskip,
                     gifftsiz, giovlp, githresh
ffilt     pvsfilter fsigA, fsigB, 1
aout      pvsynth   ffilt
aenv      linen     aout, .1, p3, .5
          out       aout
endin

</CsInstruments>
<CsScore>
i 1 0 11
</CsScore>
</CsoundSynthesizer>
```

There are much more ways of working with the pvs opcodes. Have a look at the *Signal Processing II* section of the *Opcodes Overview* to find some hints.

1. All soundfiles used in this manual are free and can be downloaded at www.csound-tutorial.net^
2. For some cases it is good to have pvsadsyn as an alternative, which is using a bank of oscillators for resynthesis.^

06 SAMPLES

36. RECORD AND PLAY SOUNDFILES

PLAYING SOUNDFILES FROM DISK - DISKIN2

The simplest way of playing a sound file from Csound is to use the diskin2 opcode. This opcode reads audio directly from the hard drive location where it is stored, i.e. it does not pre-load the sound file at initialisation time. This method of sound file playback is therefore good for playing back very long, or parts of very long, sound files. It is perhaps less well suited to playing back sound files where dense polyphony, multiple iterations and rapid random access to the file is required. In these situations reading from a function table or buffer is preferable.

diskin2 has additional parameters for speed of playback, and interpolation.

EXAMPLE 06A01.csd

```
<CsoundSynthesizer>

<CsOptions>
-odac ; activate real-time audio output
</CsOptions>

<CsInstruments>
; example written by Iain McCurdy

sr  =  44100
ksmps  =  32
nchnls  =  1

  instr 1 ; play audio from disk
kSpeed  init     1           ; playback speed
iSkip   init     0           ; inskip into file (in seconds)
iLoop   init     0           ; looping switch (0=off 1=on)
; read audio from disk using diskin2 opcode
a1      diskin2  "loop.wav", kSpeed, iSkip, iLoop
        out      a1          ; send audio to outputs
  endin

</CsInstruments>

<CsScore>
i 1 0 6
e
</CsScore>

</CsoundSynthesizer>
```

WRITING AUDIO TO DISK

The traditional method of rendering Csound's audio to disk is to specify a sound file as the audio destination in the Csound command or under <CsOptions>, in fact before real-time performance became a possibility this was the only way in which Csound was used. With this method, all audio that is piped to the output using *out, outs* etc. will be written to this file. The number of channels that the file will conatain will be determined by the number of channels specified in the orchestra header using 'nchnls'. The disadvantage of this method is that we cannot simultaneously listen to the audio in real-time.

EXAMPLE 06A02.csd

```
<CsoundSynthesizer>

<CsOptions>
; audio output destination is given as a sound file (wav format specified)
; this method is for deferred time performance,
; simultaneous real-time audio will not be possible
-oWriteToDisk1.wav -W
</CsOptions>

<CsInstruments>
; example written by Iain McCurdy

sr     =  44100
ksmps  =  32
```

```
nchnls =  1
0dbfs  =  1

giSine  ftgen  0, 0, 4096, 10, 1            ; a sine wave

  instr 1 ; a simple tone generator
aEnv    expon    0.2, p3, 0.001              ; a percussive envelope
aSig    poscil   aEnv, cpsmidinn(p4), giSine ; audio oscillator
        out      aSig                        ; send audio to output
  endin

</CsInstruments>

<CsScore>
; two chords
i 1   0 5 60
i 1 0.1 5 65
i 1 0.2 5 67
i 1 0.3 5 71

i 1   3 5 65
i 1 3.1 5 67
i 1 3.2 5 73
i 1 3.3 5 78
e
</CsScore>

</CsoundSynthesizer>
```

WRITING AUDIO TO DISK WITH SIMULTANEOUS REAL-TIME AUDIO OUTPUT - FOUT AND MONITOR

Recording audio output to disk whilst simultaneously monitoring in real-time is best achieved through combining the opcodes monitor and fout. 'monitor' can be used to create an audio signal that consists of a mix of all audio output from all instruments. This audio signal can then be rendered to a sound file on disk using 'fout'. 'monitor' can read multi-channel outputs but its number of outputs should correspond to the number of channels defined in the header using 'nchnls'. In this example it is reading just in mono. 'fout' can write audio in a number of formats and bit depths and it can also write multi-channel sound files.

EXAMPLE 06A03.csd

```
<CsoundSynthesizer>

<CsOptions>
-odac ; activate real-time audio output
</CsOptions>

<CsInstruments>
;example written by Iain McCurdy

sr      =       44100
ksmps   =       32
nchnls  =       1
0dbfs   =       1

giSine  ftgen  0, 0, 4096, 10, 1 ; a sine wave
gaSig   init   0; set initial value for global audio variable (silence)

  instr 1 ; a simple tone generator
aEnv    expon    0.2, p3, 0.001              ; percussive amplitude envelope
aSig    poscil   aEnv, cpsmidinn(p4), giSine ; audio oscillator
        out      aSig
  endin

  instr 2 ; write to a file (always on in order to record everything)
aSig    monitor                              ; read audio from output bus
        fout     "WriteToDisk2.wav",4,aSig   ; write audio to file (16bit mono)
  endin

</CsInstruments>

<CsScore>
; activate recording instrument to encapsulate the entire performance
i 2 0 8.3

; two chords
i 1   0 5 60
i 1 0.1 5 65
i 1 0.2 5 67
```

```
i 1 0.3 5 71

i 1   3 5 65
i 1 3.1 5 67
i 1 3.2 5 73
i 1 3.3 5 78
e
</CsScore>

</CsoundSynthesizer
```

37. RECORD AND PLAY BUFFERS

PLAYING AUDIO FROM RAM - FLOOPER2

Csound offers many opcodes for playing back sound files that have first been loaded into a function table (and therefore are loaded into RAM). Some of these offer higher quality at the expense of computation speed some are simpler and less fully featured.

One of the newer and easier to use opcodes for this task is flooper2. As its name might suggest it is intended for the playback of files with looping. 'flooper2' can also apply a cross-fade between the end and the beginning of the loop in order to smooth the transition where looping takes place.

In the following example a sound file that has been loaded into a GEN01 function table is played back using 'flooper2'. 'flooper2' also includes a parameter for modulating playback speed/pitch. There is also the option of modulating the loop points at k-rate. In this example the entire file is simply played and looped. You can replace the sound file with one of your own or you can download the one used in the example from here:

Some notes about GEN01 and function table sizes:

When storing sound files in GEN01 function tables we must ensure that we define a table of sufficient size to store our sound file. Normally function table sizes should be powers of 2 (2, 4, 8, 16, 32 etc.). If we know the duration of our sound file we can derive the required table size by multiplying this duration by the sample rate and then choosing the next power of 2 larger than this. For example when the sampling rate is 44100, we will require 44100 table locations to store 1 second of audio; but 44100 is not a power of 2 so we must choose the next power of 2 larger than this which is 65536. (Hint: you can discover a sound file's duration by using Csound's 'sndinfo' utility.)

There are some 'lazy' options however: if we underestimate the table size, when we then run Csound it will warn us that this table size is too small and conveniently inform us via the terminal what the minimum size required to store the entire file would be - we can then substitute this value in our GEN01 table. We can also overestimate the table size in which case Csound won't complain at all, but this is a rather inefficient approach.

If we give table size a value of zero we have what is referred to as 'deferred table size'. This means that Csound will calculate the exact table size needed to store our sound file and use this as the table size but this will probably not be a power of 2. Many of Csound's opcodes will work quite happily with non-power of 2 function table sizes, but not all! It is a good idea to know how to deal with power of 2 table sizes. We can also explicitly define non-power of 2 table sizes by prefacing the table size with a minus sign '-'.

All of the above discussion about required table sizes assumed that the sound file was mono, to store a stereo sound file will naturally require twice the storage space, for example, 1 second of stereo audio will require 88200 storage locations. GEN01 will indeed store stereo sound files and many of Csound's opcodes will read from stereo GEN01 function tables, but again not all! We must be prepared to split stereo sound files, either to two sound files on disk or into two function tables using GEN01's 'channel' parameter (p8), depending on the opcodes we are using.

Storing audio in GEN01 tables as mono channels with non-deferred and power of 2 table sizes will ensure maximum compatibility.

EXAMPLE 06B01.csd

```
<CsoundSynthesizer>

<CsOptions>
-odac ; activate real-time audio
</CsOptions>

<CsInstruments>
; example written by Iain McCurdy

sr  =  44100
```

```
ksmps  =  32
nchnls  =  1
0dbfs   =       1

; STORE AUDIO IN RAM USING GEN01 FUNCTION TABLE
giSoundFile   ftgen   0, 0, 262144, 1, "loop.wav", 0, 0, 0

  instr 1 ; play audio from function table using flooper2 opcode
kAmp         =         1   ; amplitude
kPitch       =         p4  ; pitch/speed
kLoopStart   =         0   ; point where looping begins (in seconds)
kLoopEnd     =         nsamp(giSoundFile)/sr; loop end (end of file)
kCrossFade   =         0   ; cross-fade time
; read audio from the function table using the flooper2 opcode
aSig         flooper2  kAmp,kPitch,kLoopStart,kLoopEnd,kCrossFade,giSoundFile
             out       aSig ; send audio to output
  endin

</CsInstruments>

<CsScore>
; p4 = pitch
; (sound file duration is 4.224)
i 1 0 [4.224*2] 1
i 1 + [4.224*2] 0.5
i 1 + [4.224*1] 2
e
</CsScore>

</CsoundSynthesizer>
```

CSOUND'S BUILT-IN RECORD-PLAY BUFFER - SNDLOOP

Csound has an opcode called sndloop which provides a simple method of recording some audio into a buffer and then playing it back immediately. The duration of audio storage required is defined when the opcode is initialized. In the following example two seconds is provided. Once activated, as soon as two seconds of live audio has been recorded by 'sndloop', it immediately begins playing it back in a loop. 'sndloop' allows us to modulate the speed/pitch of the played back audio as well as providing the option of defining a crossfade time between the end and the beginning of the loop. In the example pressing 'r' on the computer keyboard activates record followed by looped playback, pressing 's' stops record or playback, pressing '+' increases the speed and therefore the pitch of playback and pressing '-' decreases the speed/pitch of playback. If playback speed is reduced below zero it enters the negative domain in which case playback will be reversed.

You will need to have a microphone connected to your computer in order to use this example.

EXAMPLE 06B02.csd

```
<CsoundSynthesizer>

<CsOptions>
; real-time audio in and out are both activated
-iadc -odac
</CsOptions>

<CsInstruments>
;example written by Iain McCurdy

sr  =  44100
ksmps  =  32
nchnls  =  1

  instr 1
; PRINT INSTRUCTIONS
           prints  "Press 'r' to record, 's' to stop playback, "
           prints  "'+' to increase pitch, '-' to decrease pitch.\\n"
; SENSE KEYBOARD ACTIVITY
kKey sensekey; sense activity on the computer keyboard
aIn        inch    1             ; read audio from first input channel
kPitch     init    1             ; initialize pitch parameter
iDur       init    2             ; inititialize duration of loop parameter
iFade      init    0.05          ; initialize crossfade time parameter
 if kKey = 114 then              ; if 'r' has been pressed...
kTrig      =       1             ; set trigger to begin record-playback
 elseif kKey = 115 then          ; if 's' has been pressed...
kTrig      =       0             ; set trigger to turn off record-playback
 elseif kKey = 43 then           ; if '+' has been pressed...
kPitch     =       kPitch + 0.02 ; increment pitch parameter
 elseif kKey = 95 then           ; if '-' has been pressed
```

```
kPitch     =       kPitch - 0.02 ; decrement pitch parameter
 endif                           ; end of conditional branches
; CREATE SNDLOOP INSTANCE
aOut, kRec sndloop aIn, kPitch, kTrig, iDur, iFade ; (kRec output is not used)
           out     aOut          ; send audio to output
  endin

</CsInstruments>

<CsScore>
i 1 0 3600 ; instr 1 plays for 1 hour
</CsScore>

</CsoundSynthesizer>
```

RECORDING TO AND PLAYBACK FROM A FUNCTION TABLE

Writing to and reading from buffers can also be achieved through the use of Csound's opcodes for table reading and writing operations. Although the procedure is a little more complicated than that required for 'sndloop' it is ultimately more flexible. In the next example separate instruments are used for recording to the table and for playing back from the table. Another instrument which runs constantly scans for activity on the computer keyboard and activates the record or playback instruments accordingly. For writing to the table we will use the tablew opcode and for reading from the table we will use the table opcode (if we were to modulate the playback speed it would be better to use one of Csound's interpolating variations of 'table' such as tablei or table3. Csound writes individual values to table locations, the exact table locations being defined by an 'index'. For writing continuous audio to a table this index will need to be continuously moving 1 location for every sample. This moving index (or 'pointer') can be created with an a-rate line or a phasor. The next example uses 'line'. When using Csound's table operation opcodes we first need to create that table, either in the orchestra header or in the score. The duration of the audio buffer can be calculated from the size of the table. In this example the table is 2^17 points long, that is 131072 points. The duration in seconds is this number divided by the sample rate which in our example is 44100Hz. Therefore maximum storage duration for this example is 131072/44100 which is around 2.9 seconds.

EXAMPLE 06B03.csd

```
<CsoundSynthesizer>

<CsOptions>
; real-time audio in and out are both activated
-iadc -odac -d -m0
</CsOptions>

<CsInstruments>
; example written by Iain McCurdy

sr  =  44100
ksmps  =  32
nchnls  =  1

giBuffer ftgen  0, 0, 2^17, 7, 0; table for audio data storage
maxalloc 2,1 ; allow only one instance of the recording instrument at a time!

  instr 1 ; Sense keyboard activity. Trigger record or playback accordingly.
           prints  "Press 'r' to record, 'p' for playback.\\n"
iTableLen  =       ftlen(giBuffer)  ; derive buffer function table length
idur       =       iTableLen / sr   ; derive storage time in seconds
kKey sensekey                       ; sense activity on the computer keyboard
  if kKey=114 then                  ; if ASCCI value of 114 ('r') is output
event "i", 2, 0, idur, iTableLen  ; activate recording instrument (2)
  endif
 if kKey=112 then                   ; if ASCCI value of 112 ('p) is output
event "i", 3, 0, idur, iTableLen  ; activate playback instrument
 endif
  endin

  instr 2 ; record to buffer
iTableLen  =        p4              ; table/recording length in samples
; -- print progress information to terminal --
           prints   "recording"
           printks  ".", 0.25       ; print '.' every quarter of a second
krelease   release                  ; sense when note is in final k-rate pass...
 if krelease=1 then                 ; then ..
           printks  "\\ndone\\n", 0 ; ... print a message
 endif
; -- write audio to table --
ain        inch     1               ; read audio from live input channel 1
```

```
andx       line     0,p3,iTableLen  ; create an index for writing to table
           tablew   ain,andx,giBuffer ; write audio to function table
endin

  instr 3 ; playback from buffer
iTableLen  =        p4              ; table/recording length in samples
; -- print progress information to terminal --
           prints   "playback"
           printks  ".", 0.25       ; print '.' every quarter of a second
krelease   release                  ; sense when note is in final k-rate pass
 if krelease=1 then                 ; then ...
           printks  "\\ndone\\n", 0 ; ... print a message
 endif; end of conditional branch
; -- read audio from table --
aNdx       line     0, p3, iTableLen; create an index for reading from table
a1         table    aNdx, giBuffer  ; read audio to audio storage table
           out      a1              ; send audio to output
  endin

</CsInstruments>

<CsScore>
i 1 0 3600 ; Sense keyboard activity. Start recording - playback.
</CsScore>

</CsoundSynthesizer>
```

ENCAPSULATING RECORD AND PLAY BUFFER FUNCTIONALITY TO A UDO

Recording and playing of buffers can also be encapsulated into a User Defined Opcode. For being flexible in the size of the buffer, the *tabw* opcode will be used for writing audio data to a buffer. *tabw* writes to a table of any size and does not need a power-of-two table size like *tablew*. An empty table (buffer) of any size can be created with a negative number as size. A table for recording 10 seconds of audio data can be created in this way:

```
giBuf1    ftgen    0, 0, -(10*sr), 2, 0
```

The used can decide whether he wants to assign a certain number to the table, or whether he lets Csound do this job, calling the table via its variable, in this case giBuf1. This is a UDO for creating a mono buffer, and another UDO for creating a stereo buffer:

```
 opcode BufCrt1, i, io
ilen, inum xin
ift       ftgen     inum, 0, -(ilen*sr), 2, 0
          xout      ift
 endop

 opcode BufCrt2, ii, io
ilen, inum xin
iftL      ftgen     inum, 0, -(ilen*sr), 2, 0
iftR      ftgen     inum, 0, -(ilen*sr), 2, 0
          xout      iftL, iftR
 endop
```

This simplifies the procedure of creating a record/play buffer, because the user is just asked for the length of the buffer. A number can be given, but by default Csound will assign this number. This statement will create an empty stereo table for 5 seconds of recording:

```
iBufL,iBufR BufCrt2   5
```

A first, simple version of a UDO for recording will just write the incoming audio to sequential locations of the table. This can be done by setting the *ksmps* value to 1 inside this UDO (setksmps 1), so that each audio sample has its own discrete k-value. In this way the write index for the table can be assigned via the statement andx=kndx, and increased by one for the next k-cycle. An additional k-input turns recording on and of:

```
 opcode BufRec1, 0, aik
ain, ift, krec  xin
          setksmps  1
if krec == 1 then ;record as long as krec=1
kndx      init      0
andx      =         kndx
          tabw      ain, andx, ift
kndx      =         kndx+1
endif
 endop
```

The reading procedure is simple, too. Actually the same code can be used; it is sufficient just to replace the opcode for writing (*tabw*) with the opcode for reading (*tab*):

```
 opcode BufPlay1, a, ik
ift, kplay  xin
          setksmps  1
if kplay == 1 then ;play as long as kplay=1
kndx      init      0
andx      =         kndx
aout      tab       andx, ift
kndx      =         kndx+1
endif
 endop
```

So - let's use these first simple UDOs in a Csound instrument. Press the "r" key as long as you want to record, and the "p" key for playing back. Note that you must disable the key repeats on your computer keyboard for this example (in QuteCsound, disable "Allow key repeats" in Configuration -> General).

EXAMPLE 06B04.csd

```
<CsoundSynthesizer>
<CsOptions>
-i adc -o dac -d -m0
</CsOptions>
<CsInstruments>
;example written by Joachim Heintz
sr = 44100
ksmps = 32
nchnls = 1
0dbfs = 1

  opcode BufCrt1, i, io
ilen, inum xin
ift       ftgen     inum, 0, -(ilen*sr), 2, 0
          xout      ift
  endop

  opcode BufRec1, 0, aik
ain, ift, krec  xin
          setksmps  1
imaxindx  =         ftlen(ift)-1 ;max index to write
knew      changed   krec
if krec == 1 then ;record as long as krec=1
 if knew == 1 then ;reset index if restarted
kndx      =         0
 endif
kndx      =         (kndx > imaxindx ? imaxindx : kndx)
andx      =         kndx
          tabw      ain, andx, ift
kndx      =         kndx+1
endif
  endop

  opcode BufPlay1, a, ik
ift, kplay  xin
          setksmps  1
imaxindx  =         ftlen(ift)-1 ;max index to read
knew      changed   kplay
if kplay == 1 then ;play as long as kplay=1
 if knew == 1 then ;reset index if restarted
kndx      =         0
 endif
kndx      =         (kndx > imaxindx ? imaxindx : kndx)
andx      =         kndx
aout      tab       andx, ift
kndx      =         kndx+1
endif
          xout      aout
  endop

  opcode KeyStay, k, kkk
;returns 1 as long as a certain key is pressed
key, k0, kascii    xin ;ascii code of the key (e.g. 32 for space)
kprev     init      0 ;previous key value
kout      =         (key == kascii || (key == -1 && kprev == kascii) ? 1 : 0)
kprev     =         (key > 0 ? key : kprev)
kprev     =         (kprev == key && k0 == 0 ? 0 : kprev)
          xout      kout
  endop

  opcode KeyStay2, kk, kk
;combines two KeyStay UDO's (this way is necessary
;because just one sensekey opcode is possible in an orchestra)
kasci1, kasci2 xin ;two ascii codes as input
key,k0    sensekey
kout1     KeyStay   key, k0, kasci1
kout2     KeyStay   key, k0, kasci2
          xout      kout1, kout2
  endop


instr 1
ain        inch      1 ;audio input on channel 1
iBuf       BufCrt1   3 ;buffer for 3 seconds of recording
kRec,kPlay KeyStay2  114, 112 ;define keys for record and play
           BufRec1   ain, iBuf, kRec ;record if kRec=1
aout       BufPlay1  iBuf, kPlay ;play if kPlay=1
           out       aout ;send out
endin

</CsInstruments>
<CsScore>
i 1 0 1000
</CsScore>
</CsoundSynthesizer>
```

Let's realize now a more extended and easy to operate version of these two UDO's for recording and playing a buffer. The wishes of a user might be the following:

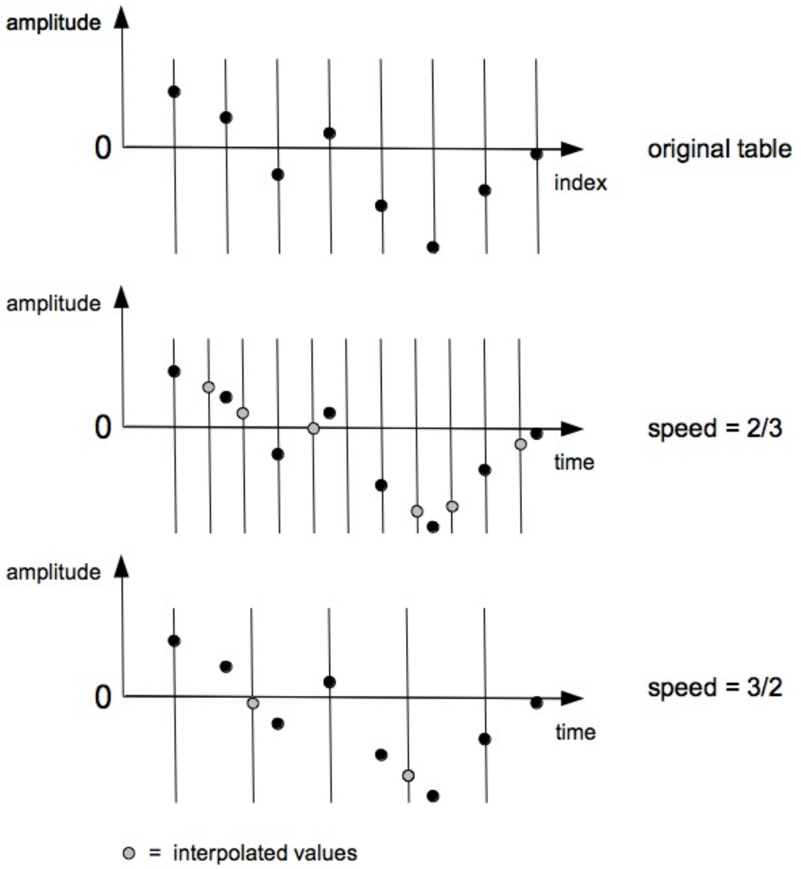
Recording:

- allow recording not just from the beginning of the buffer, but also from any arbitrary starting point *kstart*
- allow circular recording (wrap around) if the end of the buffer has been reached: *kwrap=1*

Playing:

- play back with different speed *kspeed* (negaitve speed means playing backwards)
- start playback at any point of the buffer *kstart*
- end playback at any point of the buffer *kend*
- allow certain modes of wraparound *kwrap* while playing:
    - kwrap=0 stops at the defined end point of the buffer
    - kwrap=1 repeats playback between defined end and start points
    - kwrap=2 starts at a the defined starting point but wraps between end point and beginning of the buffer
    - kwrap=3 wraps between *kstart* and the end of the table

The following example provides versions of *BufRec* and *BufPlay* which do this job. We will use the table3 opcode instead of the simple tab or table opcodes in this case, because we want to translate any number of samples in the table to any number of output samples by different speed values:



For higher or lower speed values than the original record speed, interpolation must be used in between certain sample values if the original shape of the wave is to be reproduced as accurately as possible. This job is performed with high quality by [table3]() which employs cubic interpolation.

In a typical application of recording and playing buffer buffers, the ability to interact with the process will be paramount. We can benefit from having interactive access to the following:

- starting and stopping record

- adjusting the start and end points of recording
- use or prevent wraparound while recording
- starting and stopping playback
- adjusting the start and end points of playback
- adjusting wraparound in playback at one of the specified modes (1 - 4)
- applying volume at playback

These interactions could be carried out via widgets, MIDI, OSC or something else. As we want to provide examples which can be used with any Csound frontend here, we are restricted to triggering the record and play events by hitting the space bar of the computer keyboard. (See the QuteCsound version of this example for a more interactive version.)

EXAMPLE 06B05.csd

```
<CsoundSynthesizer>
<CsOptions>
-i adc -o dac -d
</CsOptions>
<CsInstruments>
;example written by joachim heintz
sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1
```

```
  opcode BufCrt2, ii, io ;creates a stereo buffer
ilen, inum xin ;ilen = length of the buffer (table) in seconds
iftL      ftgen     inum, 0, -(ilen*sr), 2, 0
iftR      ftgen     inum, 0, -(ilen*sr), 2, 0
          xout      iftL, iftR
  endop

  opcode BufRec1, k, aikkkk ;records to a buffer
ain, ift, krec, kstart, kend, kwrap xin
  setksmps 1
kendsmps =  kend*sr ;end point in samples
kendsmps =  (kendsmps == 0 || kendsmps > ftlen(ift) ? ftlen(ift) : kendsmps)
kfinished =  0
knew  changed krec ;1 if record just started
 if krec == 1 then
  if knew == 1 then
kndx  =  kstart * sr - 1 ;first index to write
  endif
  if kndx >= kendsmps-1 && kwrap == 1 then
kndx  =  -1
  endif
  if kndx < kendsmps-1 then
kndx  =  kndx + 1
andx  =  kndx
  tabw  ain, andx, ift
  else
kfinished =  1
  endif
 endif
   xout  kfinished
  endop

  opcode BufRec2, k, aaiikkkk ;records to a stereo buffer
ainL, ainR, iftL, iftR, krec, kstart, kend, kwrap xin
kfin      BufRec1     ainL, iftL, krec, kstart, kend, kwrap
kfin      BufRec1     ainR, iftR, krec, kstart, kend, kwrap
          xout        kfin
  endop

  opcode BufPlay1, ak, ikkkkkk
ift, kplay, kspeed, kvol, kstart, kend, kwrap xin
;kstart = begin of playing the buffer in seconds
;kend = end of playing in seconds. 0 means the end of the table
;kwrap = 0: no wrapping. stops at kend (positive speed) or kstart
;  (negative speed).this makes just sense if the direction does not
;  change and you just want to play the table once
;kwrap = 1: wraps between kstart and kend
;kwrap = 2: wraps between 0 and kend
;kwrap = 3: wraps between kstart and end of table
;CALCULATE BASIC VALUES
kfin  init  0
iftlen  =  ftlen(ift)/sr ;ftlength in seconds
kend  =  (kend == 0 ? iftlen : kend) ;kend=0 means end of table
kstart01 =  kstart/iftlen ;start in 0-1 range
kend01  =  kend/iftlen ;end in 0-1 range
kfqbas  =  (1/iftlen) * kspeed ;basic phasor frequency
;DIFFERENT BEHAVIOUR DEPENDING ON WRAP:
if kplay == 1 && kfin == 0 then
 ;1. STOP AT START- OR ENDPOINT IF NO WRAPPING REQUIRED (kwrap=0)
 if kwrap == 0 then
; -- phasor freq so that 0-1 values match distance start-end
kfqrel  =  kfqbas / (kend01-kstart01)
andxrel phasor  kfqrel ;index 0-1 for distance start-end
; -- final index for reading the table (0-1)
andx  =  andxrel * (kend01-kstart01) + (kstart01)
kfirst  init  1 ;don't check condition below at the first k-cycle (always true)
kndx  downsamp andx
kprevndx init  0
 ;end of table check:
  ;for positive speed, check if this index is lower than the previous one
  if kfirst == 0 && kspeed > 0 && kndx < kprevndx then
kfin  =  1
 ;for negative speed, check if this index is higher than the previous one
  else
kprevndx =  (kprevndx == kstart01 ? kend01 : kprevndx)
   if kfirst == 0 && kspeed < 0 && kndx > kprevndx then
kfin  =  1
   endif
kfirst  =  0 ;end of first cycle in wrap = 0
  endif
 ;sound out if end of table has not yet reached
asig  table3  andx, ift, 1
kprevndx =  kndx ;next previous is this index
 ;2. WRAP BETWEEN START AND END (kwrap=1)
 elseif kwrap == 1 then
kfqrel  =  kfqbas / (kend01-kstart01) ;same as for kwarp=0
```

```
andxrel phasor  kfqrel
andx  =  andxrel * (kend01-kstart01) + (kstart01)
asig  table3  andx, ift, 1 ;sound out
 ;3. START AT kstart BUT WRAP BETWEEN 0 AND END (kwrap=2)
 elseif kwrap == 2 then
kw2first init  1
  if kw2first == 1 then ;at first k-cycle:
  reinit  wrap3phs ;reinitialize for getting the correct start phase
kw2first =  0
  endif
kfqrel  =  kfqbas / kend01 ;phasor freq so that 0-1 values match distance start-
end
wrap3phs:
andxrel phasor  kfqrel, i(kstart01) ;index 0-1 for distance start-end
  rireturn ;end of reinitialization
andx  =  andxrel * kend01 ;final index for reading the table
asig  table3  andx, ift, 1 ;sound out
 ;4. WRAP BETWEEN kstart AND END OF TABLE(kwrap=3)
 elseif kwrap == 3 then
kfqrel  =  kfqbas / (1-kstart01) ;phasor freq so that 0-1 values match distance
start-end
andxrel phasor  kfqrel ;index 0-1 for distance start-end
andx  =  andxrel * (1-kstart01) + kstart01 ;final index for reading the table
asig  table3  andx, ift, 1
 endif
else ;if either not started or finished at wrap=0
asig  =  0 ;don't produce any sound
endif
    xout  asig*kvol, kfin
  endop

  opcode BufPlay2, aak, iikkkkkk ;plays a stereo buffer
iftL, iftR, kplay, kspeed, kvol, kstart, kend, kwrap xin
aL,kfin   BufPlay1     iftL, kplay, kspeed, kvol, kstart, kend, kwrap
aR,kfin   BufPlay1     iftR, kplay, kspeed, kvol, kstart, kend, kwrap
          xout         aL, aR, kfin
  endop

  opcode In2, aa, kk ;stereo audio input
kchn1, kchn2 xin
ain1      inch      kchn1
ain2      inch      kchn2
          xout      ain1, ain2
  endop

  opcode Key, kk, k
;returns '1' just in the k-cycle a certain key has been pressed (kdown)
;  or released (kup)
kascii    xin ;ascii code of the key (e.g. 32 for space)
key,k0    sensekey
knew      changed   key
kdown     =         (key == kascii && knew == 1 && k0 == 1 ? 1 : 0)
kup       =         (key == kascii && knew == 1 && k0 == 0 ? 1 : 0)
          xout      kdown, kup
  endop

instr 1
giftL,giftR BufCrt2   3 ;creates a stereo buffer for 3 seconds
gainL,gainR In2     1,2 ;read input channels 1 and 2 and write as global audio
          prints    "PLEASE PRESS THE SPACE BAR ONCE AND GIVE AUDIO INPUT
                     ON CHANNELS 1 AND 2.\n"
          prints    "AUDIO WILL BE RECORDED AND THEN AUTOMATICALLY PLAYED
                     BACK IN SEVERAL MANNERS.\n"
krec,k0   Key       32
 if krec == 1 then
          event     "i", 2, 0, 10
 endif
endin

instr 2
; -- records the whole buffer and returns 1 at the end
kfin      BufRec2   gainL, gainR, giftL, giftR, 1, 0, 0, 0
  if kfin == 0 then
          printks   "Recording!\n", 1
  endif
 if kfin == 1 then
ispeed    random    -2, 2
istart    random    0, 1
iend      random    2, 3
iwrap     random    0, 1.999
iwrap     =         int(iwrap)
printks "Playing back with speed = %.3f, start = %.3f, end = %.3f,
                    wrap = %d\n", p3, ispeed, istart, iend, iwrap
aL,aR,kf  BufPlay2  giftL, giftR, 1, ispeed, 1, istart, iend, iwrap
  if kf == 0 then
          printks   "Playing!\n", 1
  endif
 endif
krel      release
 if kfin == 1 && kf == 1 || krel == 1 then
          printks   "PRESS SPACE BAR AGAIN!\n", p3
          turnoff
 endif
          outs      aL, aR
endin

</CsInstruments>
<CsScore>
i 1 0 1000
e
</CsScore>
</CsoundSynthesizer>
```

07 MIDI

38. RECEIVING EVENTS BY MIDIIN
39. TRIGGERING INSTRUMENT INSTANCES
40. WORKING WITH CONTROLLERS
41. READING MIDI FILES
42. MIDI OUTPUT

38. RECEIVING EVENTS BY MIDIIN

Csound provides a variety of opcodes, such as cpsmidi, ampmidi and ctrl7 which allow for transparent interpretation of incoming midi data. These opcodes allow us to read in midi information without us having to worry about parsing status bytes and so on. Occasionally when we are involved in more complex midi interaction, it might be advantageous for us to scan all raw midi information that is coming into Csound. The midiin opcode allows us to do this.

In the next example a simple midi monitor is constructed. Incoming midi events are printed to the terminal with some formatting to make them readable. We can disable Csound's default instrument triggering mechanism (which in this example we don't want) by giving the line:

```
massign 0,0
```

just after the header statement (sometimes referred to as instrument 0).

For this example to work you will need to ensure that you have activated live midi input within Csound, either by using the -M flag or from within the QuteCsound configuration menu, and that you have a midi keyboard or controller connected. You may also want to include the -m0 flag which will disable some of Csound's additional messaging output and therefore allow our midi printout to be presented more clearly.

The status byte tells us what sort of midi information has been received. For example, a value of 144 tells us that a midi note event has been received, a value of 176 tells us that a midi controller event has been received, a value of 224 tells us that pitch bend has been received and so on.

The meaning of the two data bytes depends on what sort of status byte has been received. For example if a midi note event has been received then data byte 1 gives us the note velocity and data byte 2 gives us the note number, if a midi controller event has been received then data byte 1 gives us the controller number and data byte 2 gives us the controller value.

EXAMPLE 07A01.csd

```
<CsoundSynthesizer>

<CsOptions>
-Ma -m0
; activates all midi devices, suppress note printings
</CsOptions>

<CsInstruments>
; Example by Iain McCurdy

; no audio so 'sr' or 'nchnls' aren't relevant
ksmps = 32

; using massign with these arguments disables default instrument triggering
massign 0,0

  instr 1
kstatus, kchan, kdata1, kdata2  midiin            ;read in midi
ktrigger  changed  kstatus, kchan, kdata1, kdata2 ;trigger if midi data changes
 if  ktrigger=1&&kstatus!=0 then              ;if status byte is non-zero...
; -- print midi data to the terminal with formatting --
 printks "status:%d%tchannel:%d%tdata1:%d%tdata2:%d%n"\
                                    ,0,kstatus,kchan,kdata1,kdata2
 endif
  endin

</CsInstruments>

<CsScore>
i 1 0 3600 ; instr 1 plays for 1 hour
</CsScore>

</CsoundSynthesizer>
```

The principle advantage of the *midiin* opcode is that, unlike opcodes such as *cpsmidi*, *ampmidi* and *ctrl7* which only receive specific midi data types on a specific channel, *midiin* 'listens' to all incoming data including system exclusive. In situations where elaborate Csound instrument triggering mappings that are beyond the default triggering mechanism's capabilities, are required then the use for *midiin* might be beneficial.

39. TRIGGERING INSTRUMENT INSTANCES

CSOUND'S DEFAULT SYSTEM OF INSTRUMENT TRIGGERING VIA MIDI

Csound has a default system for instrument triggering via midi. Provided a midi keyboard has been connected and the appropriate commmand line flags for midi input have been set (see configuring midi for further information) or the appropriate settings have been made in QuteCsound's configuration menu, then midi notes received on midi channel 1 will trigger instrument 1, notes on channel 2 will trigger instrument 2 and so on. Instruments will turn on and off in sympathy with notes being pressed and released on the midi keyboard and Csound will correctly unravel polyphonic layering and turn on and off only the correct layer of the same instrument begin played. Midi activated notes can be thought of as 'held' notes, similar to notes activated in the score with a negative duration (p3). Midi activated notes will sustain indefinitely as long as the performance time will allow until a corresponding note off has been received - this is unless this infinite p3 duration is overwritten within the instrument itself by p3 begin explicitly defined.

The following example confirms this default mapping of midi channels to instruments. You will need a midi keyboard that allows you to change the midi channel on which it is transmmitting. Besides a written confirmation to the console of which instrument is begin triggered, there is an audible confirmation in that instrument 1 plays single pulses, instrument 2 plays sets of two pulses and instrument 3 plays sets of three pulses. The example does not go beyond three instruments. If notes are received on midi channel 4 and above, because corresonding instruments do not exist, notes on any of these channels will be directed to instrument 1.

EXAMPLE 07B01.csd

```
<CsoundSynthesizer>

<CsOptions>
-Ma -odac -m0
;activates all midi devices, real time sound output, and suppress note printings
</CsOptions>

<CsInstruments>
; Example by Iain McCurdy

sr = 44100
ksmps = 32
nchnls = 1
0dbfs = 1

gisine ftgen 0,0,2^12,10,1

  instr 1 ; 1 impulse (midi channel 1)
prints "instrument/midi channel: %d%n",p1 ; print instrument number to terminal
reset:                                    ; label 'reset'
     timout 0, 1, impulse                 ; jump to 'impulse' for 1 second
     reinit reset                         ; reninitialize pass from 'reset'
impulse:                                  ; label 'impulse'
aenv expon     1, 0.3, 0.0001             ; a short percussive envelope
aSig poscil    aenv, 500, gisine          ; audio oscillator
     out       aSig                       ; audio to output
  endin

  instr 2 ; 2 impulses (midi channel 2)
prints "instrument/midi channel: %d%n",p1
reset:
     timout 0, 1, impulse
     reinit reset
impulse:
aenv expon     1, 0.3, 0.0001
aSig poscil    aenv, 500, gisine
a2   delay     aSig, 0.15                 ; short delay adds another impulse
     out       aSig+a2                    ; mix two impulses at output
  endin

  instr 3 ; 3 impulses (midi channel 3)
```

```
prints "instrument/midi channel: %d%n",p1
reset:
     timout 0, 1, impulse
     reinit reset
impulse:
aenv expon     1, 0.3, 0.0001
aSig poscil    aenv, 500, gisine
a2   delay     aSig, 0.15                 ; delay adds a 2nd impulse
a3   delay     a2, 0.15                   ; delay adds a 3rd impulse
     out       aSig+a2+a3                 ; mix the three impulses at output
  endin

</CsInstruments>
<CsScore>
f 0 300
e
</CsScore>
<CsoundSynthesizer>
```

USING MASSIGN TO MAP MIDI CHANNELS TO INSTRUMENTS

We can use the massign opcode, which is used just after the header statement, to explicitly map midi channels to specific instruments and thereby overrule Csound's default mappings. *massign* takes two input arguments, the first defines the midi channel to be redirected and the second stipulates which instrument it should be directed to. The following example is identical to the previous one except that the *massign* statements near the top of the orchestra jumble up the default mappings. Midi notes on channel 1 will be mapped to instrument 3, notes on channel 2 to instrument 1 and notes on channel 3 to instrument 2. Undefined channel mappings will be mapped according to the default arrangement and once again midi notes on channels for which an instrument does not exist will be mapped to instrument 1.

EXAMPLE 07B02.csd

```
<CsoundSynthesizer>

<CsOptions>
-Ma -odac -m0
; activate all midi devices, real time sound output, and suppress note printing
</CsOptions>

<CsInstruments>
; Example by Iain McCurdy

sr = 44100
ksmps = 32
nchnls = 1
0dbfs = 1

gisine ftgen 0,0,2^12,10,1

massign 1,3  ; channel 1 notes directed to instr 3
massign 2,1  ; channel 2 notes directed to instr 1
massign 3,2  ; channel 3 notes directed to instr 2

  instr 1 ; 1 impulse (midi channel 1)
iChn midichn                                 ; discern what midi channel
prints "channel:%d%tinstrument: %d%n",iChn,p1 ; print instr num and midi channel
reset:                                       ; label 'reset'
     timout 0, 1, impulse                    ; jump to 'impulse' for 1 second
     reinit reset                            ; reninitialize pass from 'reset'
impulse:                                     ; label 'impulse'
aenv expon     1, 0.3, 0.0001                ; a short percussive envelope
aSig poscil    aenv, 500, gisine             ; audio oscillator
     out       aSig                          ; send audio to output
  endin

  instr 2 ; 2 impulses (midi channel 2)
iChn midichn
prints "channel:%d%tinstrument: %d%n",iChn,p1
reset:
     timout 0, 1, impulse
     reinit reset
impulse:
aenv expon     1, 0.3, 0.0001
aSig poscil    aenv, 500, gisine
a2   delay     aSig, 0.15                     ; delay generates a 2nd impulse
     out       aSig+a2                        ; mix two impulses at the output
  endin
  instr 3 ; 3 impulses (midi channel 3)
iChn midichn
prints "channel:%d%tinstrument: %d%n",iChn,p1
reset:
     timout 0, 1, impulse
     reinit reset
impulse:
aenv expon     1, 0.3, 0.0001
aSig poscil    aenv, 500, gisine
a2   delay     aSig, 0.15                     ; delay generates a 2nd impulse
a3   delay     a2, 0.15                       ; delay generates a 3rd impulse
     out       aSig+a2+a3                     ; mix three impulses at output
  endin

</CsInstruments>

<CsScore>
f 0 300
e
</CsScore>

<CsoundSynthesizer>
```

massign also has a couple of additional functions that may come in useful. A channel number of zero is interpreted as meaning 'any'. The following instruction will map notes on any and all channels to instrument 1.

```
massign 0,1
```

An instrument number of zero is interpreted as meaning 'none' so the following instruction will instruct Csound to ignore triggering for notes received on any and all channels.

```
massign 0,0
```

The above feature is useful when we want to scan midi data from an already active instrument using the midiin opcode, as we did in EXAMPLE 0701.csd.

USING MULTIPLE TRIGGERING

Csound's event/event_i opcode (see the Triggering Instrument Events chapter) makes it possible to trigger any other instrument from a midi-triggered one. As you can assign a fractional number to an instrument, you can distinguish the single instances from each other. This is an example for using fractional instrument numbers.

EXAMPLE 07B03.csd

```
<CsoundSynthesizer>
<CsOptions>
-Ma
</CsOptions>
<CsInstruments>
;Example by Joachim Heintz, using code of Victor Lazzarini
sr = 44100
ksmps = 32
nchnls = 1
0dbfs = 1

          massign   0, 1 ;assign all incoming midi to instr 1

  instr 1 ;global midi instrument, calling instr 2.cc.nnn (c=channel, n=note
number)
inote     notnum    ;get midi note number
ichn      midichn   ;get midi channel
instrnum  =         2 + ichn/100 + inote/100000 ;make fractional instr number
     ; -- call with indefinite duration
           event_i   "i", instrnum, 0, -1, ichn, inote
kend      release   ;get a "1" if instrument is turned off
 if kend == 1 then
          event     "i", -instrnum, 0, 1 ;then turn this instance off
 endif
  endin

  instr 2
ichn      =         int(frac(p1)*100)
inote     =         round(frac(frac(p1)*100)*1000)
          prints    "instr %f: ichn = %f, inote = %f%n", p1, ichn, inote
          printks   "instr %f playing!%n", 1, p1
  endin

</CsInstruments>
```

```
<CsScore>
f 0 36000
e
</CsScore>
</CsoundSynthesizer>
```

This example merely demonstrates a technique for passing information about MIDI channel and note number from the directly triggered instrument to a sub-instrument. A practical application for this would be in creating keygroups - triggering different instruments by playing in different regions of the keyboard. In this case you could change just the line:

```
instrnum  =         2 + ichn/100 + inote/100000
```

to this:

```
 if inote < 48 then
instrnum  =         2
 elseif inote < 72 then
instrnum  =         3
 else
instrnum  =         4
 endif
instrnum  =         instrnum + ichn/100 + inote/100000
```

In this case you will call for any key below C3 instrument 2, for any key between C3 and B4 instrument 3, and for any higher key instrument 4.

By this multiple triggering you are also able to trigger more than one instrument at the same time (which is not possible by the *massign* opcode). This is an example using a User Defined Opcode (see the UDO chapter of this manual):

EXAMPLE 07B04.csd

```
<CsoundSynthesizer>
<CsOptions>
-Ma
</CsOptions>
<CsInstruments>
;Example by Joachim Heintz, using code of Victor Lazzarini
sr = 44100
ksmps = 32
nchnls = 1
0dbfs = 1

          massign   0, 1 ;assign all incoming midi to instr 1
giInstrs  ftgen     0, 0, -5, -2, 2, 3, 4, 10, 100 ;instruments to be triggered

 opcode MidiTrig, 0, io
;triggers the first inum instruments in the function table ifn by a midi event,
; with fractional numbers containing channel and note number information

; -- if inum=0 or not given, all instrument numbers in ifn are triggeredifn, inum
xin
inum      =         (inum == 0 ? ftlen(ifn) : inum)
inote     notnum
ichn      midichn
iturnon   =         0
turnon:
iinstrnum tab_i     iturnon, ifn
if iinstrnum > 0 then
ifracnum  =         iinstrnum + ichn/100 + inote/100000
         event_i   "i", ifracnum, 0, -1
endif
         loop_lt   iturnon, 1, inum, turnon
kend      release
if kend == 1 then
kturnoff  =         0
turnoff:
kinstrnum tab       kturnoff, ifn
 if kinstrnum > 0 then
kfracnum  =         kinstrnum + ichn/100 + inote/100000
         event     "i", -kfracnum, 0, 1
         loop_lt   kturnoff, 1, inum, turnoff
 endif
endif
 endop

 instr 1 ;global midi instrument
; -- trigger the first two instruments in the giInstrs table
         MidiTrig  giInstrs, 2
 endin
 instr 2
ichn      =         int(frac(p1)*100)
inote     =         round(frac(frac(p1)*100)*1000)
         prints    "instr %f: ichn = %f, inote = %f%n", p1, ichn, inote
         printks   "instr %f playing!%n", 1, p1
 endin

 instr 3
ichn      =         int(frac(p1)*100)
inote     =         round(frac(frac(p1)*100)*1000)
         prints    "instr %f: ichn = %f, inote = %f%n", p1, ichn, inote
         printks   "instr %f playing!%n", 1, p1
 endin


</CsInstruments>
<CsScore>
f 0 36000
e
</CsScore>
</CsoundSynthesizer>
```

40. WORKING WITH CONTROLLERS

SCANNING MIDI CONTINUOUS CONTROLLERS

The most useful opcode for reading in midi continuous controllers is ctrl7. 'ctrl7's input arguments allow us to specify midi channel and controller number of the controller to be scanned in addition to giving us the option to rescale the received midi values between a new minimum and maximum value as defined by the 3rd and 4th input arguments. Further possibilities for modifying the data output are provided by the 5th (optional) argument which is used to point to a function table that reshapes the controllers output response to something other than linear. This can be useful when working with parameters which are normally expressed on a logarithmic scale such as frequency.

The following example scans midi controller 1 on channel 1 and prints values received to the console. The minimum and maximum values are given as 0 and 127 therefore they are not rescaled at all. (Controller 1 is also the modulation wheel on a midi keyboard.)

EXAMPLE 07C01.csd

```
<CsoundSynthesizer>

<CsOptions>
-Ma -odac
; activate all MIDI devices
</CsOptions>

<CsInstruments>
; Example by Iain McCurdy

; 'sr' and 'nchnls' are irrelevant so are omitted
ksmps = 32

  instr 1
kCtrl    ctrl7    1,1,0,127    ; read in controller 1 on channel 1
kTrigger changed  kCtrl        ; if 'kCtrl' changes generate a trigger ('bang')
 if kTrigger=1 then
; Print kCtrl to console with formatting, but only when its value changes.
printks "Controller Value: %d%n", 0, kCtrl
 endif
  endin

</CsInstruments>

<CsScore>
i 1 0 3600
e
</CsScore>

<CsoundSynthesizer>
```

There are also 14 bit and 21 bit versions of *ctrl7* (ctrl14 and ctrl21) which improve upon the 7 bit resolution of 'ctrl7' but hardware that outputs 14 or 21 bit controller information is rare so these opcodes are seldom used.

SCANNING PITCH BEND AND AFTERTOUCH

We can scan pitch bend and aftertouch in a similar way using the opcodes pchbend and aftouch. Once again we can specify minimum and maximum values with which to re-range the output. In the case of 'pchbend' we specify the value it outputs when the pitch bend wheel is at rest followed by a value which defines the entire range from when it is pulled to its minimum to when it is pushed to its maximum. In this example playing a key on the keyboard will play a note, the pitch of which can be bent up or down two semitones using the pitch bend wheel. Aftertouch can be used to modify the amplitude of the note while it is playing. Pitch bend and aftertouch data is also printed at the terminal whenever it changes. One thing to bear in mind is that for 'pchbend' to function the Csound instrument that contains it needs to have been activated by a MIDI event: you will need to play a midi note on your keyboard and then move the pitch bend wheel.

EXAMPLE 07C02.csd

```
<CsoundSynthesizer>

<CsOptions>
-odac -Ma
</CsOptions>

<CsInstruments>
;Example by Iain McCurdy

sr = 44100
ksmps = 32
nchnls = 1
0dbfs = 1

giSine  ftgen  0,0,2^10,10,1  ; a sine wave

  instr 1
; -- pitch bend --
kPchBnd  pchbend  0,4                ; read in pitch bend (range -2 to 2)
kTrig1   changed  kPchBnd            ; if 'kPchBnd' changes generate a trigger
 if kTrig1=1 then
printks "Pitch Bend:%f%n",0,kPchBnd  ; print kPchBnd to console when it changes
 endif

; -- aftertouch --
kAfttch  aftouch 0,0.9               ; read in aftertouch (range 0 to 0.9)
kTrig2   changed kAfttch             ; if 'kAfttch' changes generate a trigger
 if kTrig2=1 then
printks "Aftertouch:%d%n",0,kAfttch  ; print kAfttch to console when it changes
 endif

; -- create a sound --
iNum     notnum                      ; read in MIDI note number
; MIDI note number + pitch bend are converted to cycles per seconds
aSig     poscil   0.1,cpsmidinn(iNum+kPchBnd),giSine
         out      aSig               ; audio to output
  endin

</CsInstruments>

<CsScore>
f 0 300
e
</CsScore>

<CsoundSynthesizer>
```

INITIALIZING MIDI CONTROLLERS

It may be useful to be able to define the beginning value of a midi controller that will be used in an orchestra - that is, the value it will adopt until its corresponding hardware control has been moved. Until a controller has been moved its value in Csound defaults to its minimum setting unless additional initialization has been carried out. It is important to be aware that midi controllers only send out information when they are moved, when lying idle they send out no information. As an example, if we imagine we have an Csound instrument in which the output volume is controlled by a midi controller it might prove to be slightly frustrating that each time the orchestra is launched, this instrument will remain silent until the volume control is moved. This frustration might become greater when many midi controllers are begin utilized. It would be more useful to be able to define the starting value for each of these controllers. The initc7 opcode allows us to define the starting value of a midi controller until its hardware control has been moved. If 'initc7' is placed within the instrument itself it will be re-initialized each time the instrument is called, if it is placed in instrument 0 (just after the header statements) then it will only be initialized when the orchestra is first launched. The latter case is probably most useful.

In the following example a simple synthesizer is implemented. Midi controller 1 controls the output volume of this instrument but the 'initc7' statement near the top of the orchestra ensures that this control does not default to its minimum setting. The arguments that 'initc7' takes are for midi channel, controller number and initial value. Initial value is defined within the range 0-1, therefore a value of 1 set this controller to its maximum value (midi value 127), and a value of 0.5 sets it to its halfway value (midi value 64) and so on.

Additionally this example uses the cpsmidi opcode to scan in midi pitch and the ampmidi opcode to scan in note velocity.

EXAMPLE 07C03.csd

```
<CsoundSynthesizer>

<CsOptions>
-Ma -odac
; activate all midi inputs and real-time audio output
</CsOptions>

<CsInstruments>
; Example by Iain McCurdy

sr = 44100
ksmps = 32
nchnls = 1
0dbfs = 1

giSine ftgen 0,0,2^12,10,1 ; a sine wave
initc7 1,1,1               ; initialize CC 1 on chan. 1 to its max level

  instr 1
iCps cpsmidi               ; read in midi pitch in cycles-per-second
iAmp ampmidi 1             ; read in note velocity - re-range to be from 0 to 1
kVol ctrl7   1,1,0,1       ; read in CC 1, chan. 1. Re-range to be from 0 to 1
aSig poscil  iAmp*kVol, iCps, giSine ; an audio oscillator
     out     aSig          ; send audio to output
  endin

</CsInstruments>

<CsScore>
f 0 3600
e
</CsScore>

<CsoundSynthesizer>
```

You will maybe hear that this instrument produces 'clicks' as notes begin and end. To find out how to prevent this please see the section on envelopes with release sensing in the chapter Sound Modification: Envelopes.

SMOOTHING 7-BIT QUANTIZATION IN MIDI CONTROLLERS

A problem we encounter with 7 bit midi controllers is the poor resolution that they offer us. 7 bit means that we have 2 to the power of 7 possible values; therefore 128 possible values, which is rather inadequate for defining the frequency of an oscillator over a number of octaves, the cutoff frequency of a filter or a volume control. We quickly become aware of the parameter that is being controlled moving up in steps - not so much of a 'continuous' control. We may also experience clicking artefacts, sometimes called 'zipper noise', as the value changes. There are some things we can do to address this problem. We can filter the controller signal within Csound so that the sudden changes that occur between steps along the controller's travel are smoothed using additional interpolating values - we must be careful not to smooth excessively otherwise the response of the controller will become sluggish. Any k-rate compatible lowpass filter can be used for this task but the portk opcode is particularly useful as it allows us to define the amount of smoothing as a time taken to glide to half the required value rather than having to specify a cutoff frequency. Additionally this 'half time' value can be varied as a k-rate value which provides an advantage availed of in the following example.

This example takes the simple synthesizer of the previous example as its starting point. The volume control which is controlled by midi controller 1 on channel 1 is passed through a 'portk' filter. The 'half time' for 'portk' ramps quickly up to its required value of 0.01 through the use of a linseg statement in the previous line. This is done so that when a new note begins the volume control jumps immediately to its required value rather than gliding up from zero on account of the effect of the 'portk' filter. Try this example with the 'portk' half time defined as a constant to hear the difference. To further smooth the volume control, it is converted to an a-rate variable through the use of the interp opcode which, as well as performing this conversion, interpolates values in the gaps between k-cycles.

EXAMPLE 07C04.csd

```
<CsoundSynthesizer>
<CsOptions>
-Ma -odac
</CsOptions>
<CsInstruments>
;Example by Iain McCurdy
```

```
sr = 44100
ksmps = 32
nchnls = 1
0dbfs = 1

giSine   ftgen    0,0,2^12,10,1
         initc7   1,1,1          ; initialize CC 1 to its max. level

  instr 1
iCps      cpsmidi                ; read in midi pitch in cycles-per-second
iAmp      ampmidi 1              ; read in note velocity - re-range 0 to 1
kVol      ctrl7   1,1,0,1        ; read in CC 1, chan. 1. Re-range from 0 to 1
kPortTime linseg  0,0.001,0.01   ; create a value that quickly ramps up to 0.01
kVol      portk   kVol,kPortTime ; create a filtered version of kVol
aVol      interp  kVol           ; create an a-rate version of kVol
aSig      poscil  iAmp*aVol,iCps,giSine
          out     aSig
  endin

</CsInstruments>
<CsScore>
f 0 300
e
</CsScore>
<CsoundSynthesizer>
```

All of the techniques introduced in this section are combined in the final example which includes a 2-semitone pitch bend and tone control which is controlled by aftertouch. For tone generation this example uses the gbuzz opcode.

EXAMPLE 07C05.csd

```
<CsoundSynthesizer>

<CsOptions>
-Ma -odac
</CsOptions>

<CsInstruments>
;Example by Iain McCurdy

sr = 44100
ksmps = 32
nchnls = 1
0dbfs = 1

giCos   ftgen    0,0,2^12,11,1 ; a cosine wave
         initc7   1,1,1        ; initialize controller to its maximum level

  instr 1
iNum      notnum                   ; read in midi note number
iAmp      ampmidi 0.1              ; read in note velocity - range 0 to 0.2
kVol      ctrl7   1,1,0,1          ; read in CC 1, chan. 1. Re-range from 0 to 1
kPortTime linseg  0,0.001,0.01     ; create a value that quickly ramps up to 0.01
kVol      portk   kVol, kPortTime  ; create filtered version of kVol
aVol      interp  kVol             ; create an a-rate version of kVol.
iRange    =       2                ; pitch bend range in semitones
iMin      =       0                ; equilibrium position
kPchBnd   pchbend iMin, 2*iRange   ; pitch bend in semitones (range -2 to 2)
kPchBnd   portk   kPchBnd,kPortTime; create a filtered version of kPchBnd
aEnv      linsegr 0,0.005,1,0.1,0  ; amplitude envelope with release stage
kMul      aftouch 0.4,0.85         ; read in aftertouch
kMul      portk   kMul,kPortTime   ; create a filtered version of kMul
; create an audio signal using the 'gbuzz' additive synthesis opcode
aSig      gbuzz   iAmp*aVol*aEnv,cpsmidinn(iNum+kPchBnd),70,0,kMul,giCos
          out     aSig             ; audio to output
  endin

</CsInstruments>

<CsScore>
f 0 300
e
</CsScore>

<CsoundSynthesizer>
```

41. READING MIDI FILES

Instead of using either the standard Csound score or live midi events as input for a orchestra Csound can read a midi file and use the data contained within it as if it were a live midi input.

The command line flag to instigate reading from a midi file is '-F' followed by the name of the file or the complete path to the file if it is not in the same directory as the .csd file. Midi channels will be mapped to instrument according to the rules and options discussed in Triggering Instrument Instances and all controllers can be interpretted as desired using the techniques discussed in Working with Controllers. One thing we need to be concerned with is that without any events in our standard Csound score our performance will terminate immedately. To circumvent this problem we need some sort of dummy event in our score to fool Csound into keeping going until our midi file has completed. Something like the following, placed in the score, is often used.

```
f 0 3600
```

This dummy 'f' event will force Csound to wait for 3600 second (1 hour) before terminating performance. It doesn't really matter what number of seconds we put in here, as long as it is more than the number of seconds duration of the midi file. Alternatively a conventional 'i' score event can also keep performance going; sometimes we will have, for example, a reverb effect running throughout the performance which can also prevent Csound from terminating. Performance can be interrupted at any time by typing ctrl+c in the terminal window.

The following example plays back a midi file using Csound's 'fluidsynth' family of opcodes to facilitate playing soundfonts (sample libraries). For more information on these opcodes please consult the Csound Reference Manual. In order to run the example you will need to download a midi file and two (ideally contrasting) soundfonts. Adjust the references to these files in the example accordingly. Free midi files and soundfonts are readily available on the internet. I am suggesting that you use contrasting soundfonts, such as a marimba and a trumpet, so that you can easily hear the parsing of midi channels in the midi file to different Csound instruments. In the example channels 1,3,5,7,9,11,13 and 15 play back using soundfont 1 and channels 2,4,6,8,10,12,14 and 16 play back using soundfont 2. When using fluidsynth in Csound we normally use an 'always on' instrument to gather all the audio from the various soundfonts (in this example instrument 99) which also conveniently keeps performance going while our midi file plays back.

EXAMPLE 07D01.csd

```
<CsoundSynthesizer>

<CsOptions>
;'-F' flag reads in a midi file
-F AnyMIDIfile.mid
</CsOptions>

<CsInstruments>
;Example by Iain McCurdy

sr = 44100
ksmps = 32
nchnls = 1
0dbfs = 1

sr = 44100
ksmps = 32
nchnls = 2

giEngine     fluidEngine; start fluidsynth engine
; load a soundfont
iSfNum1      fluidLoad          "ASoundfont.sf2", giEngine, 1
; load a different soundfont
iSfNum2      fluidLoad          "ADifferentSoundfont.sf2", giEngine, 1
; direct each midi channels to a particular soundfonts
             fluidProgramSelect giEngine, 1, iSfNum1, 0, 0
             fluidProgramSelect giEngine, 3, iSfNum1, 0, 0
             fluidProgramSelect giEngine, 5, iSfNum1, 0, 0
             fluidProgramSelect giEngine, 7, iSfNum1, 0, 0
             fluidProgramSelect giEngine, 9, iSfNum1, 0, 0
             fluidProgramSelect giEngine, 11, iSfNum1, 0, 0
             fluidProgramSelect giEngine, 13, iSfNum1, 0, 0
             fluidProgramSelect giEngine, 15, iSfNum1, 0, 0
             fluidProgramSelect giEngine, 2, iSfNum2, 0, 0
             fluidProgramSelect giEngine, 4, iSfNum2, 0, 0
             fluidProgramSelect giEngine, 6, iSfNum2, 0, 0
             fluidProgramSelect giEngine, 8, iSfNum2, 0, 0
             fluidProgramSelect giEngine, 10, iSfNum2, 0, 0
             fluidProgramSelect giEngine, 12, iSfNum2, 0, 0
             fluidProgramSelect giEngine, 14, iSfNum2, 0, 0
             fluidProgramSelect giEngine, 16, iSfNum2, 0, 0

  instr 1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16 ; fluid synths for channels 1-16
iKey         notnum                 ; read in midi note number
iVel         ampmidi            127 ; read in key velocity
; create a note played by the soundfont for this instrument
             fluidNote          giEngine, p1, iKey, iVel
  endin

  instr 99 ; gathering of fluidsynth audio and audio output
aSigL, aSigR fluidOut           giEngine      ; read all audio from soundfont
             outs               aSigL, aSigR  ; send audio to outputs
  endin
</CsInstruments>

<CsScore>
i 99 0 3600 ; audio output instrument also keeps performance going
e
</CsScore>

<CsoundSynthesizer>
```

Midi file input can be combined with other Csound inputs from the score or from live midi and also bear in mind that a midi file doesn't need to contain midi note events, it could instead contain, for example, a sequence of controller data used to automate parameters of effects during a live performance.

Rather than to directly play back a midi file using Csound instruments it might be useful to import midi note events as a standard Csound score. This way events could be edited within the Csound editor or several scores could be combined. The following example takes a midi file as input and outputs standard Csound .sco files of the events contained therein. For convenience each midi channel is output to a separate .sco file, therefore up to 16 .sco files will be created. Multiple .sco files can be later recombined by using #include... statements or simply by using copy and paste.

The only tricky aspect of this example is that note-ons followed by note-offs need to be sensed and calculated as p3 duration values. This is implemented by sensing the note-off by using the release opcode and at that moment triggering a note in another instrument with the required score data. It is this second instrument that is responsible for writing this data to a score file. Midi channels are rendered as p1 values, midi note numbers as p4 and velocity values as p5.

EXAMPLE 07D02.csd

```
<CsoundSynthesizer>

<CsOptions>
; enter name of input midi file
-F InputMidiFile.mid
</CsOptions>

<CsInstruments>
; Example by Iain McCurdy

;ksmps needs to be 10 to ensure accurate rendering of timings
ksmps = 10

massign 0,1

  instr 1
iChan       midichn
iCps        cpsmidi            ; read pitch in frequency from midi notes
iVel        veloc 0, 127 ; read in velocity from midi notes
kDur        timeinsts          ; running total of duration of this note
kRelease    release            ; sense when note is ending
 if kRelease=1 then            ; if note is about to end
;           p1  p2  p3    p4     p5    p6
event "i",  2,  0, kDur, iChan, iCps, iVel ; send full note data to instr 2
 endif
  endin

  instr 2
iDur        =        p3
```

```
iChan       =        p4
iCps        =        p5
iVel        =        p6
iStartTime  times        ; read current time since the start of performance
; form file name for this channel (1-16) as a string variable
SFileName   sprintf  "Channel%d.sco",iChan
; write a line to the score for this channel's .sco file
            fprints  SFileName, "i%d\\t%f\\t%f\\t%f\\t%d\\n",\
                          iChan,iStartTime-iDur,iDur,iCps,iVel
  endin

</CsInstruments>

<CsScore>
f 0 480 ; ensure this duration is as long or longer that duration of midi file
e
</CsScore>

</CsoundSynthesizer>
```

The example above ignores continuous controller data, pitch bend and aftertouch. The second example on the page in the Csound Manual for the opcode fprintks renders all midi data to a score file.

42. MIDI OUTPUT

Csound's ability to output midi data in real-time can open up many possibilities. We can relay the Csound score to a hardware synthesizer so that it plays the notes in our score instead of a Csound instrument. We can algorithmically generate streams of notes within the orchestra and have these played by the external device. We could even route midi data internally to another piece of software. Csound could be used as a device to transform incoming midi data, transforming, transposing or arpeggiating incoming notes before they are output again. Midi output could also be used to preset faders on a motorized fader box (such as the Behringer BCF 2000) to their correct initial locations.

INITIATING REALTIME MIDI OUTPUT

The command line flag for realtime midi output is -Q. Just as when setting up an audio input or output device or a midi input device we must define the desired device number after the flag. When in doubt what midi output devices we have on our system we can always specify an 'out of range' device number (e.g. -Q999) in which case Csound will not run but will instead give an error and provide us with a list of available devices and their corresponding numbers. We can then insert an appropriate device number.

MIDIOUT - OUTPUTTING RAW MIDI DATA

The analog of the opcode for the input of raw midi data, midiin, is midiout. midiout will output a midi message with its given input arguments once every k period - this could very quickly lead to clogging of incoming midi data in the device to which midi is begin sent unless measures are taken to prevent the *midiout* code from begin executed on every k pass. In the following example this is dealt with by turning off the instrument as soon as the *midiout* line has been executed just once by using the turnoff opcode. Alternative approaches would be to set the status byte to zero after the first k pass or to embed the *midiout* within a conditional (*if... then...*) so that its rate of execution can be controlled in some way.

Another thing we need to be aware of is that midi notes do not contain any information about note duration; instead the device playing the note waits until it receives a corresponding note-off instruction on the same midi channel and with the same note number before stopping the note. When working with *midiout* we must also be aware of this. The status byte for a midi note-off is 128 but it is more common for note-offs to be expressed as a note-on (status byte 144) with zero velocity. In the following example two notes (and corresponding note offs) are send to the midi output - the first note-off makes use of the zero velocity convention whereas the second makes use of the note-off status byte. Hardware and software synths should respond similarly to both. One advantage of the note-off message using status byte 128 is that we can also send a note-off velocity, i.e. how forcefully we release the key. Only more expensive midi keyboards actually sense and send note-off velocity and it is even rarer for hardware to respond to received note-off velocities in a meaningful way. Using Csound as a sound engine we could respond to this data in a creative way however.

In order for the following example to work you must connect a midi sound module or keyboard receiving on channel 1 to the midi output of your computer. You will also need to set the appropriate device number after the '-Q' flag.

No use is made of audio so sample rate (sr), and number of channels (nchnls) are left undefined - nonetheless they will assume default values.

EXAMPLE 07E01.csd

```
<CsoundSynthesizer>

<CsOptions>
; amend device number accordingly
-Q999
</CsOptions>

<CsInstruments>
;Example by Iain McCurdy
```

```
ksmps = 32 ;no audio so sr and nchnls irrelevant

  instr 1
; arguments for midiout are read from p-fields
istatus   init      p4
ichan     init      p5
idata1    init      p6
idata2    init      p7
          midiout   istatus, ichan, idata1, idata2; send raw midi data
          turnoff   ; turn instrument off to prevent reiterations of midiout
  endin

</CsInstruments>

<CsScore>
;p1 p2 p3   p4 p5 p6 p7
i 1 0 0.01 144 1  60 100 ; note on
i 1 2 0.01 144 1  60   0 ; note off (using velocity zero)

i 1 3 0.01 144 1  60 100 ; note on
i 1 5 0.01 128 1  60 100 ; note off (using 'note off' status byte)
</CsScore>

</CsoundSynthesizer>
```

The use of separate score events for note-ons and note-offs is rather cumbersome. It would be more sensible to use the Csound note duration (p3) to define when the midi note-off is sent. The next example does this by utilizing a release flag generated by the release opcode whenever a note ends and sending the note-off then.

EXAMPLE 07E02.csd

```
<CsoundSynthesizer>

<CsOptions>
; amend device number accordingly
-Q999
</CsOptions>

<CsInstruments>
;Example by Iain McCurdy

ksmps = 32 ;no audio so sr and nchnls omitted

  instr 1
;arguments for midiout are read from p-fields
istatus   init      p4
ichan     init      p5
idata1    init      p6
idata2    init      p7
kskip     init      0
 if kskip=0 then
          midiout   istatus, ichan, idata1, idata2; send raw midi data (note on)
kskip     =         1; ensure that the note on will only be executed once
 endif
krelease  release; normally output is zero, on final k pass output is 1
 if krelease=1 then; i.e. if we are on the final k pass...
       midiout   istatus, ichan, idata1, 0; send raw midi data (note off)
 endif
  endin

</CsInstruments>

<CsScore>
;p1 p2 p3   p4 p5 p6 p7
i 1 0    4 144 1  60 100
i 1 1    3 144 1  64 100
i 1 2    2 144 1  67 100
f 0 5; extending performance time prevents note-offs from being lost
</CsScore>

</CsoundSynthesizer>
```

Obviously *midiout* is not limited to only sending only midi note information but instead this information could include continuous controller information, pitch bend, system exclusive data and so on. The next example, as well as playing a note, sends controller 1 (modulation) data which rises from zero to maximum (127) across the duration of the note. To ensure that unnessessary midi data is not sent out, the output of the *line* function is first converted into integers, and *midiout* for the continuous controller data is only executed whenever this integer value changes. The function that creates this stream of data goes slightly above this maximum value (it finishes at a value of 127.1) to ensure that a rounded value of 127 is actually achieved.

In practice it may be necessary to start sending the continuous controller data slightly before the note-on to allow the hardware time to respond.

EXAMPLE 07E03.csd

```
<CsoundSynthesizer>

<CsOptions>
; amend device number accordingly
-Q999
</CsOptions>

<CsInstruments>
; Example by Iain McCurdy

ksmps = 32 ; no audio so sr and nchnls irrelevant

  instr 1
; play a midi note
; read in values from p-fields
ichan     init      p4
inote     init      p5
iveloc    init      p6
kskip     init      0 ; 'skip' flag ensures that note-on is executed just once
 if kskip=0 then
          midiout   144, ichan, inote, iveloc; send raw midi data (note on)
kskip     =         1   ; flip flag to prevent repeating the above line
 endif
krelease  release       ; normally zero, on final k pass this will output 1
 if krelease=1 then     ; if we are on the final k pass...
          midiout   144, ichan, inote, 0  ; send a note off
 endif

; send continuous controller data
iCCnum    =         p7
kCCval    line      0, p3, 127.1  ; continuous controller data function
kCCval    =         int(kCCval)   ; convert data function to integers
ktrig     changed   kCCval        ; generate a trigger each time kCCval changes
 if ktrig=1 then                  ; if kCCval has changed...
          midiout   176, ichan, iCCnum, kCCval  ; ...send a controller message
 endif
  endin

</CsInstruments>

<CsScore>
;p1 p2 p3   p4 p5 p6  p7
i 1 0  5    1  60 100 1
f 0 7 ; extending performance time prevents note-offs from being lost
</CsScore>

</CsoundSynthesizer>
```

MIDION - OUTPUTTING MIDI NOTES MADE EASIER

midiout is the most powerful opcode for midi output but if we are only interested in sending out midi notes from an instrument then the midion opcode simplifies the procedure as the following example demonstrates by playing a simple major arpeggio.

EXAMPLE 07E04.csd

```
<CsoundSynthesizer>

<CsOptions>
; amend device number accordingly
-Q999
</CsOptions>

<CsInstruments>
; Example by Iain McCurdy

ksmps = 32 ;no audio so sr and nchnls irrelevant

  instr 1
; read values in from p-fields
kchn    =       p4
knum    =       p5
kvel    =       p6
        midion  kchn, knum, kvel ; send a midi note
  endin
```

```
</CsInstruments>

<CsScore>
;p1 p2  p3  p4 p5 p6
i 1 0   2.5 1 60  100
i 1 0.5 2   1 64  100
i 1 1   1.5 1 67  100
i 1 1.5 1   1 72  100
f 0 30 ; extending performance time prevents note-offs from being missed
</CsScore>

</CsoundSynthesizer>
```

Changing any of 'midion's k-rate input arguments in realtime will force it to stop the current midi note and send out a new one with the new parameters.

midion2 allows us to control when new notes are sent (and the current note is stopped) through the use of a trigger input. The next example uses 'midion2' to algorithmically generate a melodic line. New note generation is controlled by a metro, the rate of which undulates slowly through the use of a randomi function.

EXAMPLE 07E05.csd

```
<CsoundSynthesizer>

<CsOptions>
; amend device number accordingly
-Q999
</CsOptions>

<CsInstruments>
; Example by Iain McCurdy

ksmps = 32 ; no audio so sr and nchnls irrelevant

  instr 1
; read values in from p-fields
kchn    =        p4
knum    random   48,72.99  ; note numbers chosen randomly across a 2 octaves
kvel    random   40, 115   ; velocities are chosen randomly
krate   randomi  1,2,1     ; rate at which new notes will be output
ktrig   metro    krate^2   ; 'new note' trigger
        midion2  kchn, int(knum), int(kvel), ktrig ; send midi note if ktrig=1
  endin

</CsInstruments>

<CsScore>
i 1 0 20 1
f 0 21 ; extending performance time prevents the final note-off being lost
</CsScore>

</CsoundSynthesizer>
```

'midion' and 'midion2' generate monophonic melody lines with no gaps between notes.

moscil works in a slightly different way and allows us to explicitly define note durations as well as the pauses between notes thereby permitting the generation of more staccato melodic lines. Like 'midion' and 'midion2', 'moscil' will not generate overlapping notes (unless two or more instances of it are concurrent). The next example algorithmically generates a melodic line using 'moscil'.

EXAMPLE 07E06.csd

```
<CsoundSynthesizer>

<CsOptions>
; amend device number accordingly
-Q999
</CsOptions>

<CsInstruments>
; Example by Iain McCurdy

ksmps = 32 ;no audio so sr and nchnls omitted

seed 0; random number generators seeded by system clock

  instr 1
; read value in from p-field
```

```
kchn    =         p4
knum    random    48,72.99  ; note numbers chosen randomly across a 2 octaves
kvel    random    40, 115   ; velocities are chosen randomly
kdur    random    0.2, 1    ; note durations chosen randomly from 0.2 to 1
kpause  random    0, 0.4    ; pauses betw. notes chosen randomly from 0 to 0.4
        moscil    kchn, knum, kvel, kdur, kpause ; send a stream of midi notes
  endin

</CsInstruments>

<CsScore>
;p1 p2 p3 p4
i 1 0  20 1
f 0 21 ; extending performance time prevents final note-off from being lost
</CsScore>

</CsoundSynthesizer>
```

MIDI FILE OUTPUT

As well as (or instead of) outputting midi in realtime, Csound can render data from all of its midi output opcodes to a midi file. To do this we use the '--midioutfile=' flag followed by the desired name for our file. For example:

```
<CsOptions>
-Q2 --midioutfile=midiout.mid
</CsOptions>
```

will simultaneously stream realtime midi to midi output device number 2 and render to a file named 'midiout.mid' which will be saved in our home directory.

08 OPEN SOUND CONTROL

43. OPEN SOUND CONTROL - NETWORK COMMUNICATION

Open Sound Control (OSC) is a network protocol format for musical control data communication. A few of its advantages compared to MIDI are, that it's more accurate, quicker and much more flexible. With OSC you can easily send messages to other software independent if it's running on the same machine or over network. There is OSC support in software like PD, Max/Msp, Chuck or SuperCollider. A nice screencast of Andrés Cabrera shows communication between PD and Csound via OSC.

OSC messages contain an IP adress with port information and the data-package which will be send over network. In Csound, there are two opcodes, which provide access to network communication called OSCsend, OSClisten.

Example 08A01.csd

```
<CsoundSynthesizer>
<CsOptions>
-o dac
</CsOptions>
<CsInstruments>
sr = 48000
ksmps = 32
nchnls = 2
0dbfs = 1

; localhost means communication on the same machine, otherwise you need
; an IP adress
#define IPADDRESS # "localhost" #
#define S_PORT   # 47120 #
#define R_PORT   # 47120 #

turnon 1000  ; starts instrument 1000 immediately
turnon 1001  ; starts instrument 1001 immediately


instr 1000  ; this instrument sends OSC-values
 kValue1 randomh 0, 0.8, 4
 kNum randomh 0, 8, 8
 kMidiKey tab (int(kNum)), 2
 kOctave randomh 0, 7, 4
 kValue2 = cpsmidinn (kMidiKey*kOctave+33)
 kValue3 randomh 0.4, 1, 4
 Stext sprintf "%i", $S_PORT
 OSCsend   kValue1+kValue2, $IPADDRESS, $S_PORT, "/QuteCsound",
                  "fff", kValue1, kValue2, kValue3
endin


instr 1001  ; this instrument receives OSC-values
 kValue1Received init 0.0
 kValue2Received init 0.0
 kValue3Received init 0.0
 Stext sprintf "%i", $R_PORT
 ihandle OSCinit $R_PORT
 kAction  OSClisten ihandle, "/QuteCsound", "fff",
                 kValue1Received, kValue2Received, kValue3Received
  if (kAction == 1) then
   printk2 kValue2Received
   printk2 kValue1Received

  endif
 aSine poscil3 kValue1Received, kValue2Received, 1
 ; a bit reverbration
 aInVerb = aSine*kValue3Received
 aWetL, aWetR freeverb aInVerb, aInVerb, 0.4, 0.8
outs aWetL+aSine, aWetR+aSine
endin

</CsInstruments>
<CsScore>
f 1 0 1024 10 1
f 2 0 8 -2      0 2 4 7 9 11 0 2
e 3600
</CsScore>
</CsoundSynthesizer>
```

```
; example by Alex Hofmann (Mar. 2011)
```

09 CSOUND IN OTHER APPLICATIONS

44. CSOUND IN PD

INSTALLING

You can embed Csound in PD via the external **csoundapi~**, which has been written by Victor Lazzarini. This external is part of the Csound distribution.

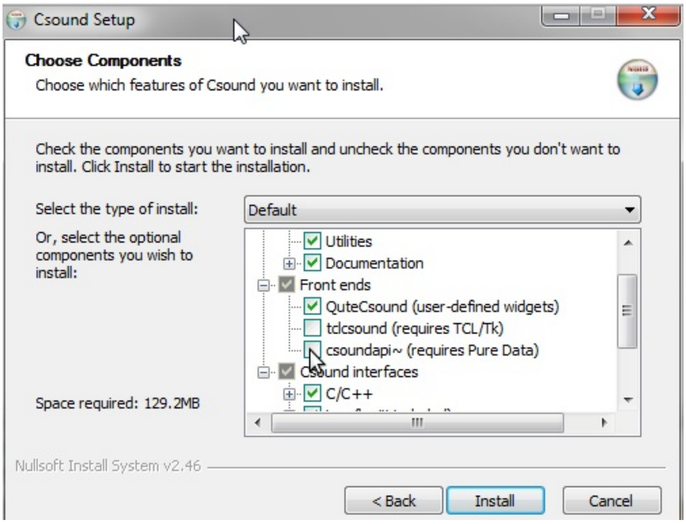
On **Ubuntu Linux**, you can install the csoundapi~ via the Synaptic package manager. Just look for "csoundapi~" or "pd-csound", check "install", and your system will install the library at the appropriate location. If you build Csound from sources, you should also be able to get the csoundapi~ via the scons option buildPDClass=1. It will be put as csoundapi~.pd_linux in /usr/lib/pd/extra, so that PD should be able to find it. If not, add it to PD's search path (File->Path...).

On **Mac OSX**, you find the csoundapi~ in the following path:

/Library/Frameworks/CsoundLib.framework/Versions/5.2/Resources/PD/csoundapi~.pd_darwin

Put this file in a folder which is in PD's search path. For PD-extended, it's by default ~/Library/Pd. But you can put it anywhere. Just make sure that the location is specified in PD's Preferences > Path... menu.

On **Windows**, while installing Csound, open up the "Front ends" component in the Installer box and make sure the item "csoundapi~" is checked:



Choose Components

Choose which features of Csound you want to install.

Check the components you want to install and uncheck the components you don't want to install. Click Install to start the installation.

Select the type of install: Default

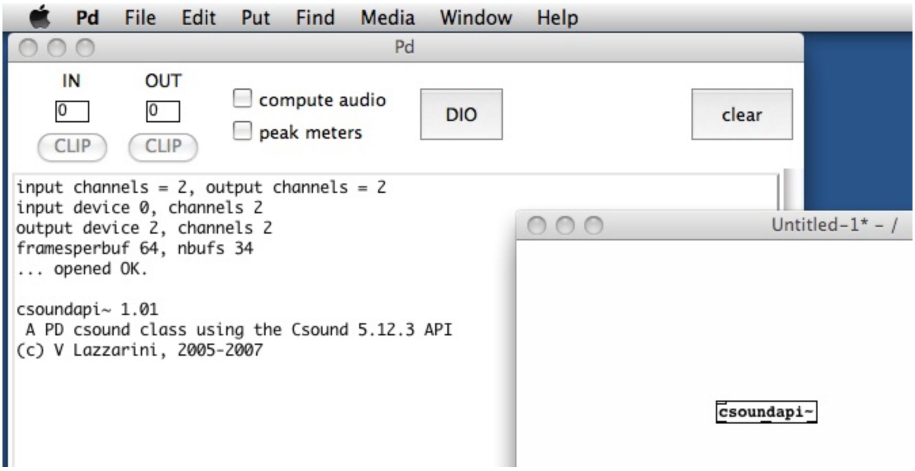
Or, select the optional components you wish to install:

- ☑ Utilities
- ☑ Documentation
- ☑ Front ends
  - ☑ QuteCsound (user-defined widgets)
  - ☐ tclcsound (requires TCL/Tk)
  - ☐ csoundapi~ (requires Pure Data)
- ☑ Csound interfaces
  - ☑ C/C++

Space required: 129.2MB

After having finished the installation, you will find csoundapi~.dll in the csound/bin folder. Copy this file into the pd/extra folder, or in any other location in PD's search path.

When you have installed the "csoundapi~" extension on any platform, and included the file in PD's search path if necessary, you should be able to call the csoundapi~ object in PD. Just open a PD window, put a new object, and type in "csoundapi~":



```
input channels = 2, output channels = 2
input device 0, channels 2
output device 2, channels 2
framesperbuf 64, nbufs 34
... opened OK.

csoundapi~ 1.01
 A PD csound class using the Csound 5.12.3 API
(c) V Lazzarini, 2005-2007
```

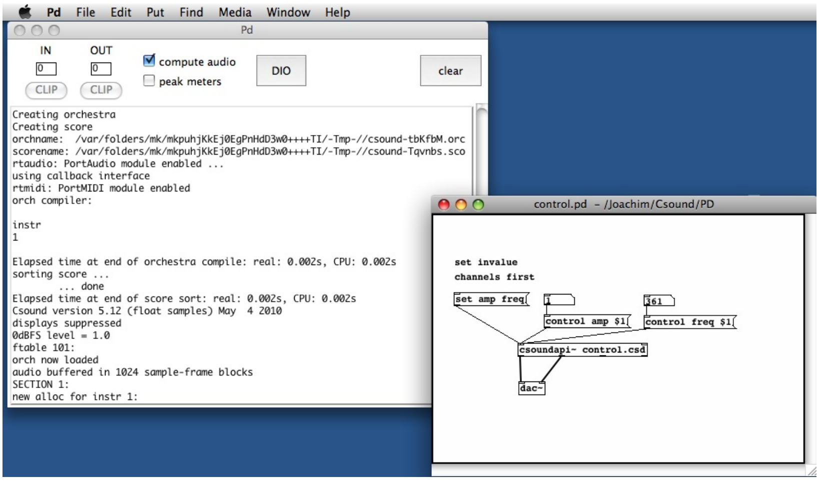
CONTROL DATA

You can send control data from PD to your Csound instrument via the keyword "control" in a message box. In your Csound code, you must receive the data via **invalue** or **chnget**. This is a simple example:

EXAMPLE 09A01.csd

```
<CsoundSynthesizer>
<CsOptions>
</CsOptions>
<CsInstruments>
;Example by Joachim Heintz

sr = 44100
nchnls = 2
0dbfs = 1
ksmps = 8

giSine    ftgen     0, 0, 2^10, 10, 1

instr 1
kFreq     invalue   "freq"
kAmp      invalue   "amp"
aSin      oscili    kAmp, kFreq, giSine
          outs      aSin, aSin
endin

</CsInstruments>
<CsScore>
i 1 0 10000
</CsScore>
</CsoundSynthesizer>
```

Save this file under the name "control.csd". Save a PD window in the same folder and create the following patch:



```
Creating orchestra
Creating score
orchname:  /var/folders/mk/mkpuhjKkEj0EgPnHdD3w0++++TI/-Tmp-//csound-tbKfbM.orc
scorename: /var/folders/mk/mkpuhjKkEj0EgPnHdD3w0++++TI/-Tmp-//csound-Tqvnbs.sco
rtaudio: PortAudio module enabled ...
using callback interface
rtmidi: PortMIDI module enabled
orch compiler:

instr
1

Elapsed time at end of orchestra compile: real: 0.002s, CPU: 0.002s
sorting score ...
        ... done
Elapsed time at end of score sort: real: 0.002s, CPU: 0.002s
Csound version 5.12 (float samples) May  4 2010
displays suppressed
0dBFS level = 1.0
ftable 101:
orch now loaded
audio buffered in 1024 sample-frame blocks
SECTION 1:
new alloc for instr 1:
```

Note that for invalue channels, you first must register these channels by a "set" message.

As you see, the first two outlets of the csoundapi~ object are the signal outlets for the audio channels 1 and 2. The third outlet is an outlet for control data (not used here, see below). The rightmost outlet sends a bang when the score has been finished.

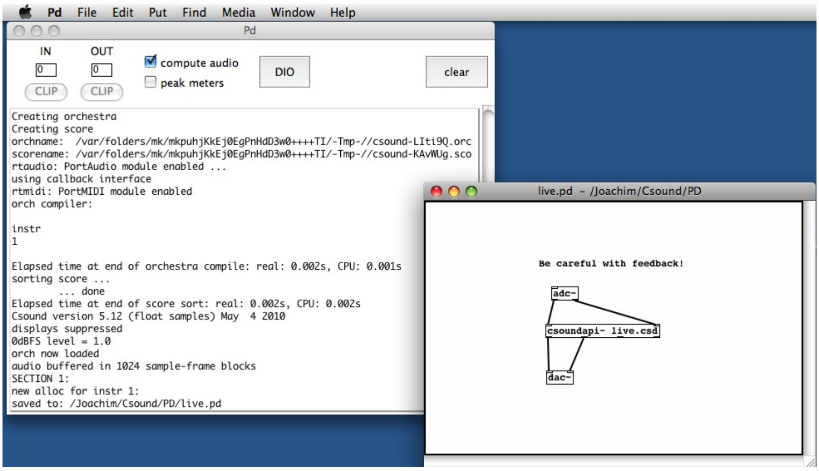
LIVE INPUT

Audio streams from PD can be received in Csound via the **inch** opcode. As many input channels there are, as many audio inlets are created in the csoundapi~ object. The following CSD uses two audio inputs:

EXAMPLE 09A02.csd

```
<CsoundSynthesizer>
<CsOptions>
</CsOptions>
<CsInstruments>
;Example by Joachim Heintz
sr = 44100
0dbfs = 1
ksmps = 8
nchnls = 2

instr 1
aL        inch      1
aR        inch      2
kcfL      randomi   100, 1000; center frequency
kcfR      randomi   100, 1000; for band pass filter
aFiltL    butterbp  aL, kcfL, kcfL/10
aoutL     balance   aFiltL, aL
aFiltR    butterbp  aR, kcfR, kcfR/10
aoutR     balance   aFiltR, aR
          outch     1, aoutL
          outch     2, aoutR
endin

</CsInstruments>
<CsScore>
i 1 0 10000
</CsScore>
</CsoundSynthesizer>
```

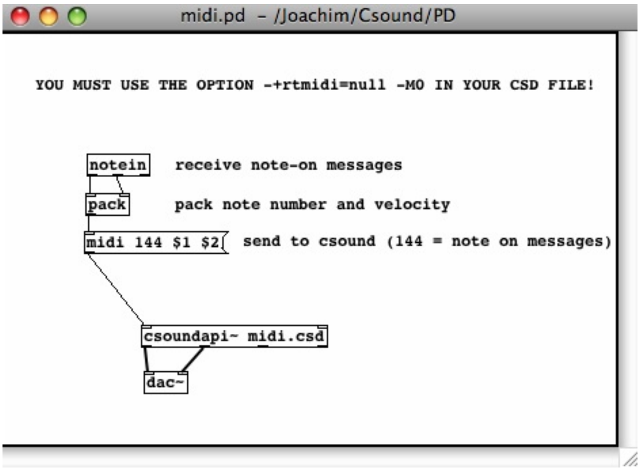
The corresponding PD patch is extremely simple:



```
Creating orchestra
Creating score
orchname:  /var/folders/mk/mkpuhjKkEj0EgPnHdD3w0++++TI/-Tmp-//csound-LIti9Q.orc
scorename: /var/folders/mk/mkpuhjKkEj0EgPnHdD3w0++++TI/-Tmp-//csound-KAvWUg.sco
rtaudio: PortAudio module enabled ...
using callback interface
rtmidi: PortMIDI module enabled
orch compiler:

instr
1

Elapsed time at end of orchestra compile: real: 0.002s, CPU: 0.001s
sorting score ...
        ... done
Elapsed time at end of score sort: real: 0.002s, CPU: 0.002s
Csound version 5.12 (float samples) May  4 2010
displays suppressed
0dBFS level = 1.0
orch now loaded
audio buffered in 1024 sample-frame blocks
SECTION 1:
new alloc for instr 1:
saved to: /Joachim/Csound/PD/live.pd
```

MIDI

The csoundapi~ object receives MIDI data via the keyword "midi". Csound is able to trigger instrument instances in receiving a "note on" message, and turning them off in receiving a "note off" message (or a note-on message with velocity=0). So this is a very simple way to build a synthesizer with arbitrary polyphonic output:


midi.pd - /Joachim/Csound/PD
YOU MUST USE THE OPTION -+rtmidi=null -M0 IN YOUR CSD FILE!
notein
receive note-on messages
pack
pack note number and velocity
midi 144 $1 $2
send to csound (144 = note on messages)
csoundapi~ midi.csd
dac~

This is the corresponding midi.csd. It must contain the options -+rtmidi=null -M0 in the <CsOptions> tag. It's an FM synth which changes the modulation index according to the verlocity: the more you press a key, the higher the index, and the more partials you get. The ratio is calculated randomly between two limits which can be adjusted.

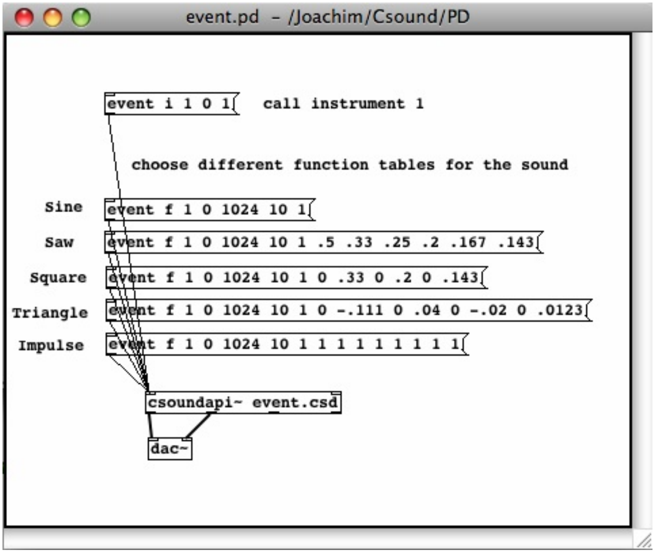
EXAMPLE 09A03.csd

```
<CsOptions>
-+rtmidi=null -M0
</CsOptions>
<CsoundSynthesizer>
<CsInstruments>
;Example by Joachim Heintz
sr      =  44100
ksmps   =  8
nchnls  =  2
0dbfs = 1

giSine    ftgen     0, 0, 2^10, 10, 1

instr 1
iFreq     cpsmidi   ;gets frequency of a pressed key
iAmp      ampmidi   8;gets amplitude and scales 0-8
iRatio    random    .9, 1.1; ratio randomly between 0.9 and 1.1
aTone     foscili   .1, iFreq, 1, iRatio/5, iAmp+1, giSine; fm
aEnv      linenr    aTone, 0, .01, .01; avoiding clicks at the end of a note
```

```
          outs      aEnv, aEnv
endin

</CsInstruments>
<CsScore>
f 0 36000; play for 10 hours
e
</CsScore>
</CsoundSynthesizer>
```

SCORE EVENTS

Score events can be sent from PD to Csound by a message with the keyword **event**. You can send any kind of score events, like instrument calls or function table statements. The following example triggers Csound's instrument 1 whenever you press the message box on the top. Different sounds can be selected by sending f events (building/replacing a function table) to Csound.


event.pd - /Joachim/Csound/PD
call instrument 1
choose different function tables for the sound
Sine
Saw
Square
Triangle
Impulse

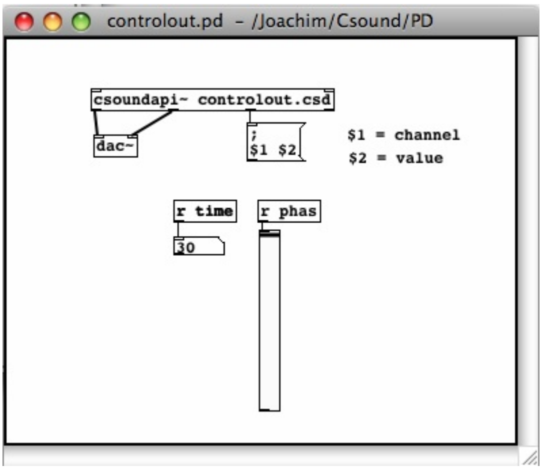
```
event i 1 0 1
event f 1 0 1024 10 1
event f 1 0 1024 10 1 .5 .33 .25 .2 .167 .143
event f 1 0 1024 10 1 0 .33 0 .2 0 .143
event f 1 0 1024 10 1 0 -.111 0 .04 0 -.02 0 .0123
event f 1 0 1024 10 1 1 1 1 1 1 1 1 1
csoundapi~ event.csd
dac~
```

EXAMPLE 09A04.csd

```
<CsoundSynthesizer>
<CsOptions>
</CsOptions>
<CsInstruments>
;Example by Joachim Heintz
sr = 44100
ksmps = 8
nchnls = 2
0dbfs = 1

          seed      0; each time different seed
giSine    ftgen     1, 0, 2^10, 10, 1; function table 1

instr 1
iDur      random    0.5, 3
p3        =         iDur
iFreq1    random    400, 1200
iFreq2    random    400, 1200
idB       random    -18, -6
kFreq     linseg    iFreq1, iDur, iFreq2
kEnv      transeg   ampdb(idB), p3, -10, 0
aTone     oscili    kEnv, kFreq, 1
          outs      aTone, aTone
endin

</CsInstruments>
<CsScore>
f 0 36000; play for 10 hours
e
</CsScore>
</CsoundSynthesizer>
```

CONTROL OUTPUT

If you want Csound to give any sort of control data to PD, you can use the opcodes **outvalue** or **chnset**. You will receive this data at the second outlet from the right of the csoundapi~ object. The data are sent as a list with two elements. The name of the control channel is the first element, and the value is the second element. You can get the values by a *route* object or by a *send/receive* chain. This is a simple example:

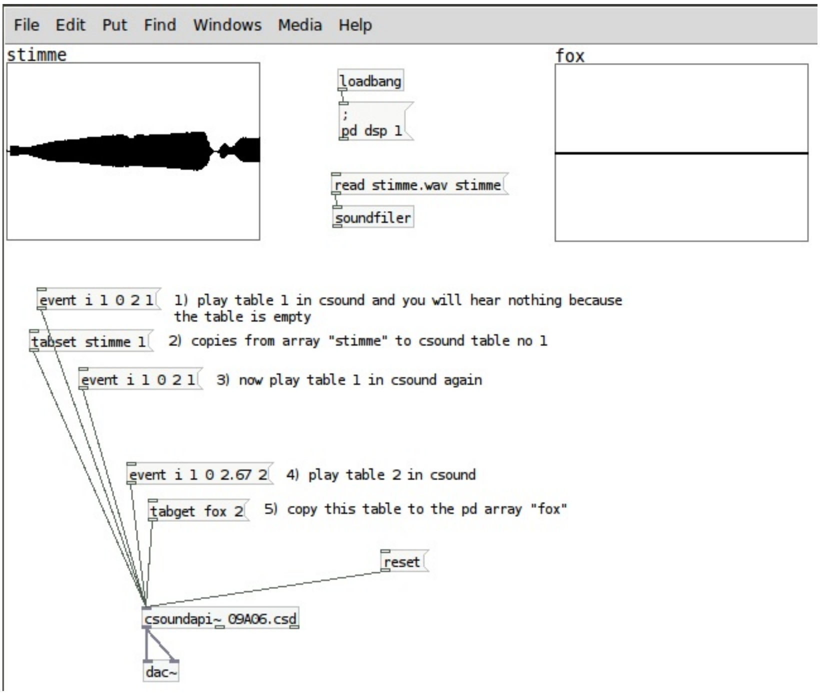


EXAMPLE 09A05.csd

```
<CsoundSynthesizer>
<CsOptions>
</CsOptions>
<CsInstruments>
;Example by Joachim Heintz

sr = 44100
nchnls = 2
0dbfs = 1
ksmps = 8

instr 1
ktim      times
kphas     phasor    1
          outvalue  "time", ktim
          outvalue  "phas", kphas*127
endin

</CsInstruments>
<CsScore>
i 1 0 30
</CsScore>
</CsoundSynthesizer>
```

SEND/RECEIVE BUFFERS FROM PD TO CSOUND AND BACK

Recently (January 2012) Victor Lazzarini has introduced a new feature which makes it possible to send a PD array to Csound, and a Csound function table to PD. The message *tabset* [tabset array-name ftable-number] copies a PD array into a Csound function table. The message *tabget* [tabget array-name ftable-number] copies a Csound function table into a PD array. The example below should explain everything. Just choose another soundfile instead of "stimme.wav".



1) play table 1 in csound and you will hear nothing because the table is empty

2) copies from array "stimme" to csound table no 1

3) now play table 1 in csound again

4) play table 2 in csound

5) copy this table to the pd array "fox"

EXAMPLE 06A06.csd

```
<CsoundSynthesizer>
<CsOptions>
-odac
</CsOptions>
<CsInstruments>
sr = 44100
ksmps = 8
nchnls = 1
0dbfs = 1

giCopy ftgen 1, 0, -88200, 2, 0 ;"empty" table
giFox  ftgen 2, 0, 0, 1, "fox.wav", 0, 0, 1

  opcode BufPlay1, a, ipop
ifn, ispeed, iskip, ivol xin
icps      =         ispeed / (ftlen(ifn) / sr)
iphs      =         iskip / (ftlen(ifn) / sr)
asig      poscil3   ivol, icps, ifn, iphs
          xout      asig
  endop

  instr 1
itable    =         p4
aout      BufPlay1  itable
          out       aout
  endin

</CsInstruments>
<CsScore>
f 0 99999
</CsScore>
</CsoundSynthesizer>
;example by joachim heintz
```

SETTINGS

Make sure that the Csound vector size given by the ksmps value, is not larger than the internal PD vector size. It should be a power of 2. I'd recommend to start with ksmps=8. If there are performance problems, try to increase this value to 16, 32, or 64.

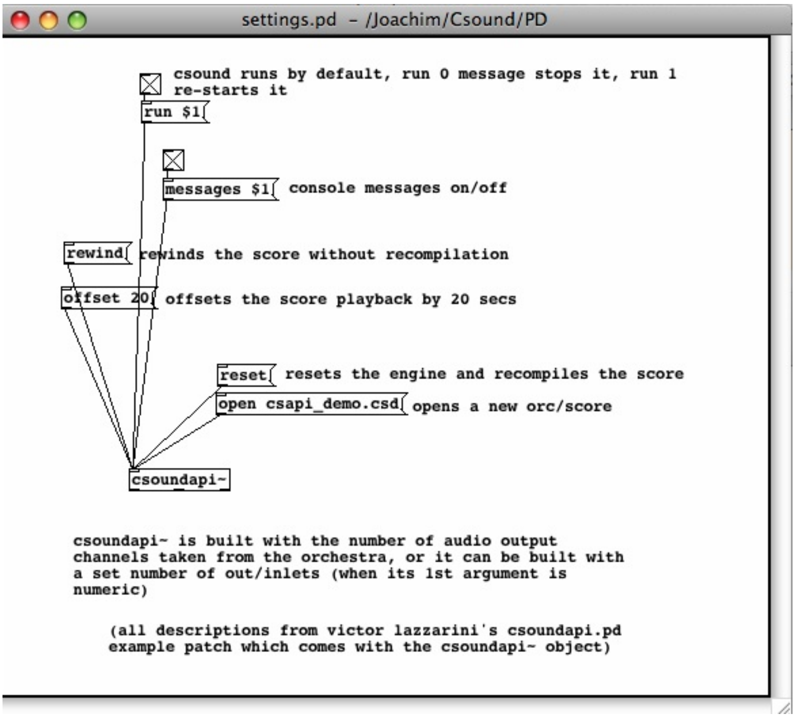
The csoundapi~ object runs by default if you turn on audio in PD. You can stop it by sending a "run 0" message, and start it again with a "run 1" message.

You can recompile the .csd file of a csoundapi~ object by sending a "reset" message.

By default, you see all the messages of Csound in the PD window. If you don't want to see them, send a "message 0" message. "message 1" prints the output again.

If you want to open a new .csd file in the csoundapi~ object, send the message "open", followed by the path of the .csd file you want to load.

A "rewind" message rewinds the score without recompilation. The message "offset", followed by a number, offsets the score playback by an amount of seconds.



csound runs by default, run 0 message stops it, run 1 re-starts it

console messages on/off

rewinds the score without recompilation

offsets the score playback by 20 secs

resets the engine and recompiles the score

opens a new orc/score

csoundapi~ is built with the number of audio output channels taken from the orchestra, or it can be built with a set number of out/inlets (when its 1st argument is numeric)

(all descriptions from victor lazzarini's csoundapi.pd example patch which comes with the csoundapi~ object)

CSOUND IN MAXMSP

The information contained within this document pertains to csound~ v1.0.7.

INTRODUCTION

Csound can be embedded in a Max patch using the csound~ object. This allows you to synthesize and process audio, MIDI, or control data with Csound.

INSTALLING

Before installing csound~, install Csound5. csound~ needs a normal Csound5 installation in order to work. You can download Csound5 from here.

Once Csound5 is installed, download the csound~ zip file from here.

INSTALLING ON MAC OS X

1. Expand the zip file and navigate to binaries/MacOSX/.
2. Choose an mxo file based on what kind of CPU you have (intel or ppc) and which type of floating point numbers are used in your Csound5 version (double or float). The name of the Csound5 installer may give a hint with the letters "f" or "d" or explicitly with the words "double" or "float". However, if you do not see a hint, then that means the installer contains both, in which case you only have to match your CPU type.
3. Copy the mxo file to:
    - *Max 4.5*: /Library/Application Support/Cycling '74/externals/
    - *Max 4.6*: /Applications/MaxMSP 4.6/Cycling'74/externals/
    - *Max 5*: /Applications/Max5/Cycling '74/msp-externals/
4. Rename the mxo file to "csound~.mxo".
5. If you would like to install the help patches, navigate to the help_files folder and copy all files to:
    - *Max 4.5*: /Applications/MaxMSP 4.5/max-help/
    - *Max 4.6*: /Applications/MaxMSP 4.6/max-help/
    - *Max 5*: /Applications/Max5/Cycling '74/msp-help/

INSTALLING ON WINDOWS

1. Expand the zip file and navigate to binaries\Windows\.
2. Choose an mxe file based on the type of floating point numbers used in your Csound5 version (double or float). The name of the Csound5 installer may give a hint with the letters "f" or "d" or explicitly with the words "double" or "float".
3. Copy the mxe file to:
    - *Max 4.5*: C:\Program Files\Common Files\Cycling '74\externals\
    - *Max 4.6*: C:\Program Files\Cycling '74\MaxMSP 4.6\Cycling '74\externals\
    - *Max 5*: C:\Program Files\Cycling '74\Max 5.0\Cycling '74\msp-externals\
4. Rename the mxe file to "csound~.mxe".
5. If you would like to install the help patches, navigate to the help_files folder and copy all files to:
    - *Max 4.5*: C:\Program Files\Cycling '74\MaxMSP 4.5\max-help\
    - *Max 4.6*: C:\Program Files\Cycling '74\MaxMSP 4.6\max-help\
    - *Max 5*: C:\Program Files\Cycling '74\Max 5.0\Cycling '74\msp-help\

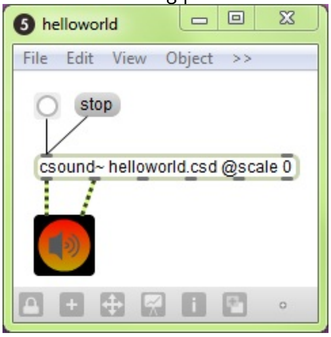
KNOWN ISSUES

On Windows (only), various versions of Csound5 have a known incompatibility with csound~ that has to do with the fluid opcodes. How can you tell if you're affected? Here's how: if you stop a Csound performance (or it stops by itself) and you click on a non-MaxMSP or non-Live window and it crashes, then you are affected. Until this is fixed, an easy solution is to remove/delete fluidOpcodes.dll from your plugins or plugins64 folder. Here are some common locations for that folder:

- C:\Program Files\Csound\plugins
- C:\Program Files\Csound\plugins64

CREATING A CSOUND~ PATCH

1. Create the following patch:



2. Save as "helloworld.maxpat" and close it.
3. Create a text file called "helloworld.csd" within the same folder as your patch.
4. Add the following to the text file:

   EXAMPLE 09B01.csd

```
<CsoundSynthesizer>
<CsInstruments>
;Example by Davis Pyon
sr     = 44100
ksmps  = 32
nchnls = 2
0dbfs  = 1

instr 1
aNoise noise .1, 0
       outch 1, aNoise, 2, aNoise
endin

</CsInstruments>
<CsScore>
f0 86400
i1 0 86400
e
</CsScore>
</CsoundSynthesizer>
```

5. Open the patch, press the bang button, then press the speaker icon.

At this point, you should hear some noise. Congratulations! You created your first csound~ patch.

You may be wondering why we had to save, close, and reopen the patch. This is needed in order for csound~ to find the csd file. In effect, saving and opening the patch allows csound~ to "know" where the patch is. Using this information, csound~ can then find csd files specified using a relative pathname (e.g. "helloworld.csd"). Keep in mind that this is only necessary for newly created patches that have not been saved yet. By the way, had we specified an absolute pathname (e.g. "C:/Mystuff/helloworld.csd"), the process of saving and reopening would have been unnecessary.

The "@scale 0" argument tells csound~ not to scale audio data between Max and Csound. By default, csound~ will scale audio to match 0dB levels. Max uses a 0dB level equal to one, while Csound uses a 0dB level equal to 32768. Using "@scale 0" and adding the statement "**0dbfs** = 1" within the csd file allows you to work with a 0dB level equal to one everywhere. This is highly recommended.

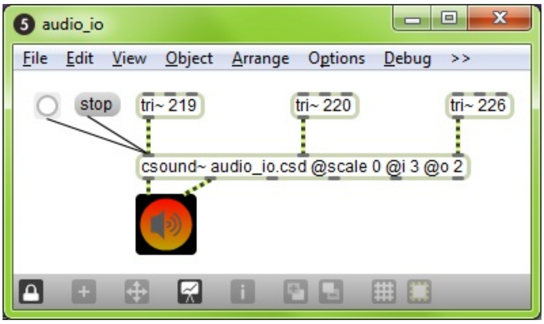
AUDIO I/O

All csound~ inlets accept an audio signal and some outlets send an audio signal. The number of audio outlets is determined by the arguments to the csound~ object. Here are four ways to specify the number of inlets and outlets:

- [csound~ @io 3]

- [csound~ @i 4 @o 7]
- [csound~ 3]
- [csound~ 4 7]

"@io 3" creates 3 audio inlets and 3 audio outlets. "@i 4 @o 7" creates 4 audio inlets and 7 audio outlets. The third and fourth lines accomplish the same thing as the first two. If you don't specify the number of audio inlets or outlets, then csound~ will have two audio inlets and two audio oulets. By the way, audio outlets always appear to the left of non-audio outlets. Let's create a patch called audio_io.maxpat that demonstrates audio i/o:



Here is the corresponding text file (let's call it audio_io.csd):

EXAMPLE 09B02.csd

```
<CsoundSynthesizer>
<CsInstruments>
;Example by Davis Pyon
sr     = 44100
ksmps  = 32
nchnls = 3
0dbfs  = 1

instr 1
aTri1 inch 1
aTri2 inch 2
aTri3 inch 3
aMix  = (aTri1 + aTri2 + aTri3) * .2
      outch 1, aMix, 2, aMix
endin

</CsInstruments>
<⁞CsScore>
f0 86400
i1 0 86400
e
</CsScore>
</CsoundSynthesizer>
```

In audio_io.maxpat, we are mixing three triangle waves into a stereo pair of outlets. In audio_io.csd, we use **inch** and **outch** to receive and send audio from and to csound~. **inch** and **outch** both use a numbering system that starts with one (the left-most inlet or outlet).

Notice the statement "**nchnls** = 3" in the orchestra header. This tells the Csound compiler to create three audio input channels and three audio output channels. Naturally, this means that our csound~ object should have no more than three audio inlets or outlets.

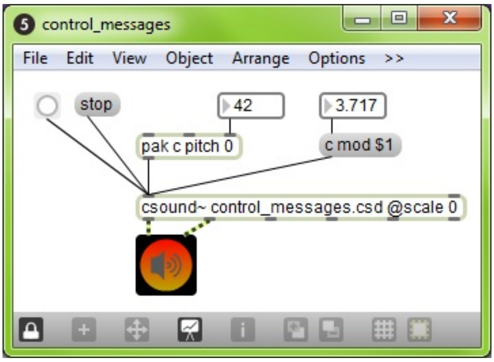
CONTROL MESSAGES

Control messages allow you to send numbers to Csound. It is the primary way to control Csound parameters at i-rate or k-rate. To control a-rate (audio) parameters, you must use and audio inlet. Here are two examples:

- control frequency 2000
- c resonance .8

Notice that you can use either "control" or "c" to indicate a control message. The second argument specifies the name of the channel you want to control and the third argument specifies the value.

The following patch and text file demonstrates control messages:



EXAMPLE 09B03.csd

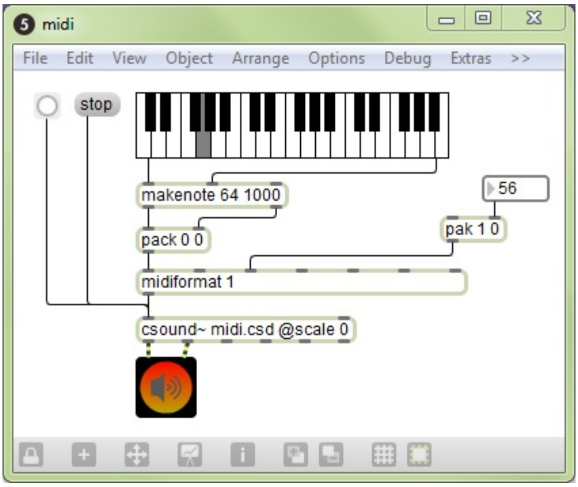
```
<CsoundSynthesizer>
<CsInstruments>
;Example by Davis Pyon
sr     = 44100
ksmps  = 32
nchnls = 2
0dbfs  = 1

giSine ftgen 1, 0, 16384, 10, 1 ; Generate a sine wave table.

instr 1
kPitch chnget "pitch"
kMod   invalue "mod"
aFM    foscil .2, cpsmidinn(kPitch), 2, kMod, 1.5, giSine
       outch 1, aFM, 2, aFM
endin
</CsInstruments>
<CsScore>
f0 86400
i1 0 86400
e
</CsScore>
</CsoundSynthesizer>
```

In the patch, notice that we use two different methods to construct control messages. The "pak" method is a little faster than the message box method, but do whatever looks best to you. You may be wondering how we can send messages to an audio inlet (remember, all inlets are audio inlets). Don't worry about it. In fact, we can send a message to any inlet and it will work.

In the text file, notice that we use two different opcodes to receive the values sent in the control messages: chnget and invalue. chnget is more versatile (it works at i-rate and k-rate, and it accepts strings) and is a tiny bit faster than invalue. On the other hand, the limited nature of invalue (only works at k-rate, never requires any declarations in the header section of the orchestra) may be easier for newcomers to Csound.

MIDI

csound~ accepts raw MIDI numbers in it's first inlet. This allows you to create Csound instrument instances with MIDI notes and also control parameters using MIDI Control Change. csound~ accepts all types of MIDI messages, except for: sysex, time code, and sync. Let's look at a patch and text file that uses MIDI:



EXAMPLE 09B04.csd

```
<CsoundSynthesizer>
<CsInstruments>
;Example by Davis Pyon
sr     = 44100
ksmps  = 32
nchnls = 2
0dbfs  = 1

massign 0, 0 ; Disable default MIDI assignments.
massign 1, 1 ; Assign MIDI channel 1 to instr 1.

giSine ftgen 1, 0, 16384, 10, 1 ; Generate a sine wave table.

instr 1
iPitch cpsmidi
kMod   midic7 1, 0, 10
aFM    foscil .2, iPitch, 2, kMod, 1.5, giSine
       outch 1, aFM, 2, aFM
endin
</CsInstruments>
<CsScore>
f0 86400
e
</CsScore>
</CsoundSynthesizer>
```

In the patch, notice how we're using midiformat to format note and control change lists into raw MIDI bytes. The "1" argument for midiformat specifies that all MIDI messages will be on channel one.

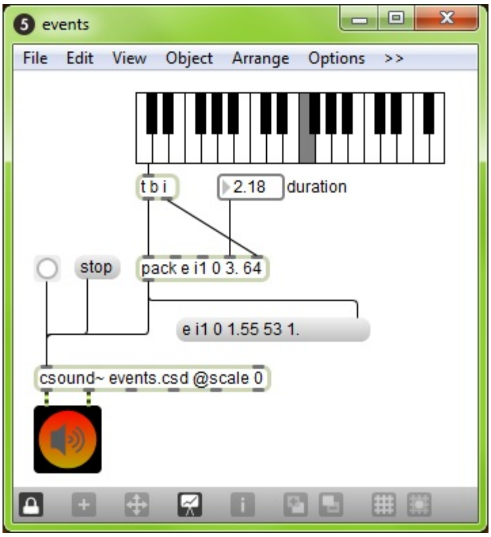
In the text file, notice the massign statements in the header of the orchestra. "massign 0,0" tells Csound to clear all mappings between MIDI channels and Csound instrument numbers. This is highly recommended because forgetting to add this statement may cause confusion somewhere down the road. The next statement "massign 1,1" tells Csound to map MIDI channel one to instrument one.

To get the MIDI pitch, we use the opcode cpsmidi. To get the FM modulation factor, we use midic7 in order to read the last known value of MIDI CC number one (mapped to the range [0,10]).

Notice that in the score section of the text file, we no longer have the statement "i1 0 86400" as we had in earlier examples. This is a good thing as you should never instantiate an instrument via both MIDI and score events (at least that has been this writer's experience).

EVENTS

To send Csound events (i.e. score statements), use the "event" or "e" message. You can send any type of event that Csound understands. The following patch and text file demonstrates how to send events:



EXAMPLE 09B05.csd

```
<CsoundSynthesizer>
<CsInstruments>
;Example by Davis Pyon
sr     = 44100
ksmps  = 32
nchnls = 2
0dbfs  = 1

instr 1
  iDur = p3
  iCps = cpsmidinn(p4)
 iMeth = 1
       print iDur, iCps, iMeth
aPluck pluck .2, iCps, iCps, 0, iMeth
       outch 1, aPluck, 2, aPluck
endin
</CsInstruments>
<CsScore>
f0 86400
e
</CsScore>
</CsoundSynthesizer>
```

In the patch, notice how the arguments to the pack object are declared. The "i1" statement tells Csound that we want to create an instance of instrument one. There is no space between "i" and "1" because pack considers "i" as a special symbol signifying an integer. The next number specifies the start time. Here, we use "0" because we want the event to start right now. The duration "3." is specified as a floating point number so that we can have non-integer durations. Finally, the number "64" determines the MIDI pitch. You might be wondering why the pack object output is being sent to a message box. This is good practice as it will reveal any mistakes you made in constructing an event message.

In the text file, we access the event parameters using p-statements. We never access **p1** (instrument number) or **p2** (start time) because they are not important within the context of our instrument. Although **p3** (duration) is not used for anything here, it is often used to create audio envelopes. Finally, **p4** (MIDI pitch) is converted to cycles-per-second. The print statement is there so that we can verify the parameter values.

46. CSOUND IN ABLETON LIVE

Csound can be used in Ableton Live through Max4Live. Max4Live is a toolkit which allows users to build devices for Live using Max/MSP. Please see the previous section on using Csound in Max/MSP for more details on how to use Csound in Live.

Cabbage can also be used to run Csound in Live, or any other audio plugin host. Please refer to the section titled 'Cabbage' in chapter 10.

47. D. CSOUND AS A VST PLUGIN

Csound can be built into a VST or AU plugin through the use of the Csound host API. Refer to the section on using the Csound API for more details.

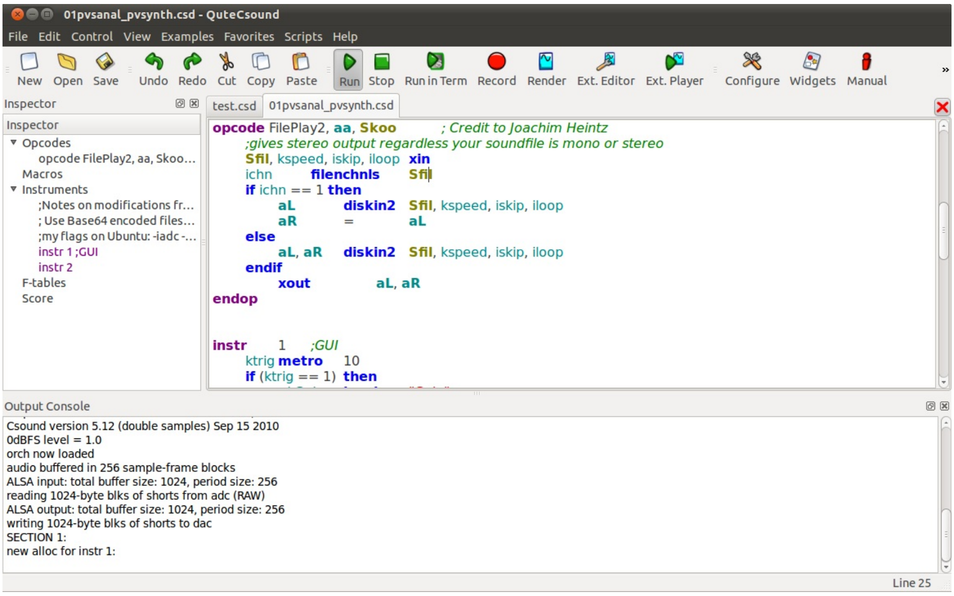
If you are not well versed in low level computer programming you can just use Cabbage to create Csound based plugins. See the section titled 'Cabbage' in Chapter 10.

10 CSOUND FRONTENDS

48. CSOUNDQT

CsoundQt is a free, cross-platform graphical frontend to Csound. It features syntax highlighting, code completion and a graphical widget editor for realtime control of Csound. It comes with many useful code examples, from basic tutorials to complex synthesizers and pieces written in Csound. It also features an integrated Csound language help display.

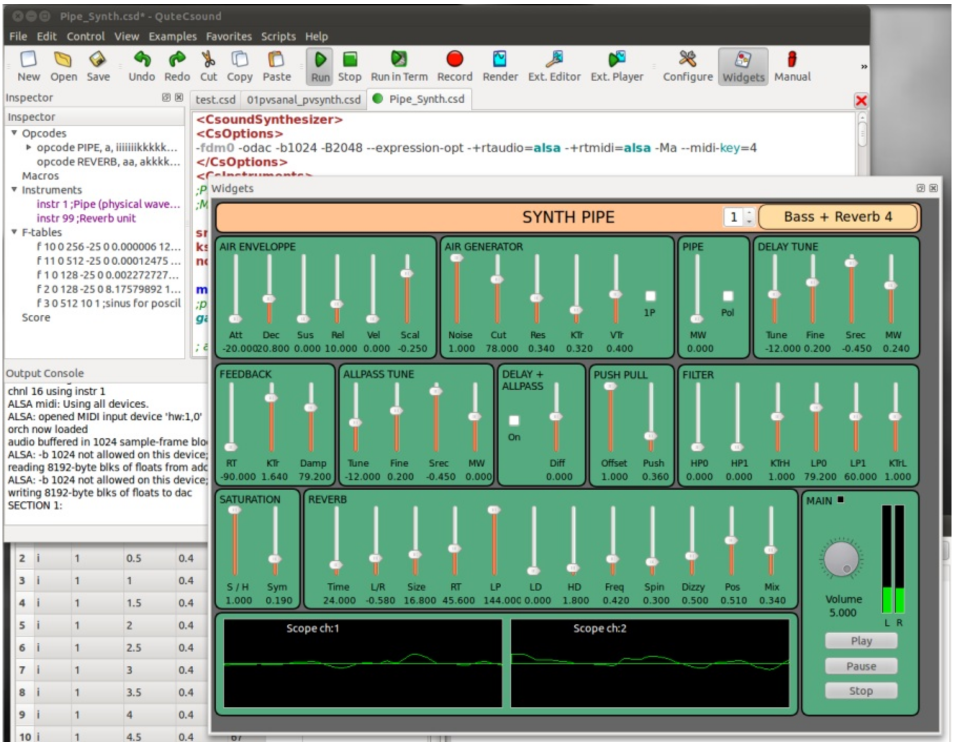
CsoundQt (named QuteCsound until automn 2011) can be used as a code editor tailored for Csound, as it facilitates running and rendering Csound files without the need of typing on the command line using the Run and Render buttons.



```
opcode FilePlay2, aa, Skoo          ; Credit to Joachim Heintz
       ;gives stereo output regardless your soundfile is mono or stereo
       Sfil, kspeed, iskip, iloop  xin
       ichn         filenchnls      Sfil
       if ichn == 1 then
              aL         diskin2   Sfil, kspeed, iskip, iloop
              aR         =         aL
       else
              aL, aR     diskin2   Sfil, kspeed, iskip, iloop
       endif
              xout              aL, aR
endop


instr       1      ;GUI
       ktrig metro      10
       if (ktrig == 1)  then
```

```
Csound version 5.12 (double samples) Sep 15 2010
0dBFS level = 1.0
orch now loaded
audio buffered in 256 sample-frame blocks
ALSA input: total buffer size: 1024, period size: 256
reading 1024-byte blks of shorts from adc (RAW)
ALSA output: total buffer size: 1024, period size: 256
writing 1024-byte blks of shorts to dac
SECTION 1:
new alloc for instr 1:
```

In the widget editor panel, you can create a variety of widgets to control Csound. To link the value from a widget, you first need to set its channel, and then use the Csound opcode invalue. To send values to widgets, e.g. for data display, you need to use the outvalue opcode.


Pipe_Synth.csd* - QuteCsound
File Edit Control View Examples Favorites Scripts Help
New Open Save Undo Redo Cut Copy Paste Run Stop Run in Term Record Render Ext. Editor Ext. Player Configure Widgets Manual
Inspector
Opcodes
opcode PIPE, a, iiiiiiikkkkk...
opcode REVERB, aa, akkkk...
Macros
Instruments
instr 1 ;Pipe (physical wave...
instr 99 ;Reverb unit
F-tables
f 10 0 256 -25 0 0.000006 12...
f 11 0 512 -25 0 0.00012475 ...
f 1 0 128 -25 0 0.002272727...
f 2 0 128 -25 0 8.17579892 1...
f 3 0 512 10 1 ;sinus for poscil
Score
test.csd 01pvsanal_pvsynth.csd Pipe_Synth.csd
Widgets
SYNTH PIPE
Bass + Reverb 4
Scope ch:1
Scope ch:2
Play
Pause
Stop

```
<CsoundSynthesizer>
<CsOptions>
-fdm0 -odac -b1024 -B2048 --expression-opt -+rtaudio=alsa -+rtmidi=alsa -Ma --midi-key=4
</CsOptions>
```

Output Console

```
chnl 16 using instr 1
ALSA midi: Using all devices.
ALSA: opened MIDI input device 'hw:1,0'
orch now loaded
audio buffered in 1024 sample-frame blo
ALSA: -b 1024 not allowed on this device;
reading 8192-byte blks of floats from adc
ALSA: -b 1024 not allowed on this device;
writing 8192-byte blks of floats to dac
SECTION 1:
```

SYNTH PIPE 1

AIR ENVELOPPE: Att -20.000 Dec 20.800 Sus 0.000 Rel 10.000 Vel 0.000 Scal -0.250

AIR GENERATOR: Noise 1.000 Cut 78.000 Res 0.340 KTr 0.320 VTr 0.400 1P

PIPE: MW 0.000 Pol

DELAY TUNE: Tune -12.000 Fine 0.200 Srec -0.450 MW 0.240

FEEDBACK: RT -90.000 KTr 1.640 Damp 79.200

ALLPASS TUNE: Tune -12.000 Fine 0.200 Srec -0.450 MW 0.000

DELAY + ALLPASS: On Diff 0.000

PUSH PULL: Offset 1.000 Push 0.360

FILTER: HP0 0.000 HP1 0.000 KTrH 1.000 LP0 79.200 LP1 60.000 KTrL 1.000

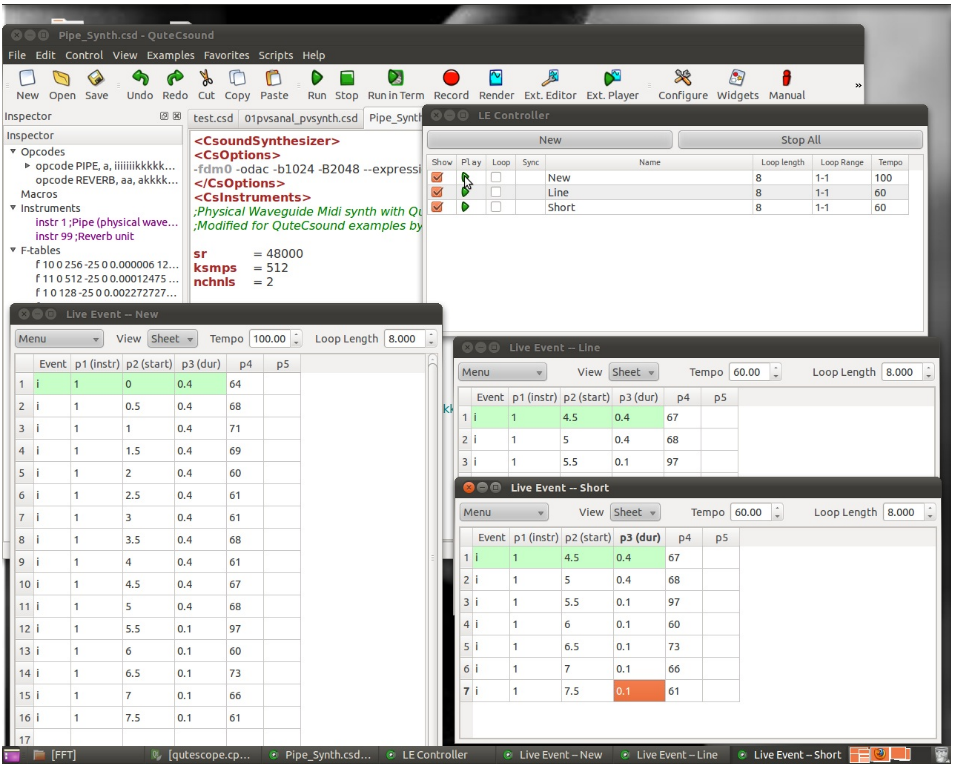
SATURATION: S / H 1.000 Sym 0.190

REVERB: Time 24.000 L/R -0.580 Size 16.800 RT 45.600 LP 144.000 LD 0.000 HD 1.800 Freq 0.420 Spin 0.300 Dizzy 0.500 Pos 0.510 Mix 0.340

MAIN: Volume 5.000 L R

2	i	1	0.5	0.4
3	i	1	1	0.4
4	i	1	1.5	0.4
5	i	1	2	0.4
6	i	1	2.5	0.4
7	i	1	3	0.4
8	i	1	3.5	0.4
9	i	1	4	0.4
10	i	1	4.5	0.4

CsoundQt also offers convenient facilities for score editing in a spreadsheet like environment which can be transformed using Python scripting.


Pipe_Synth.csd - QuteCsound
File Edit Control View Examples Favorites Scripts Help
New Open Save Undo Redo Cut Copy Paste Run Stop Run in Term Record Render Ext. Editor Ext. Player Configure Widgets Manual
Inspector
Opcodes
opcode PIPE, a, iiiiiiikkkkk...
opcode REVERB, aa, akkkk...
Macros
Instruments
instr 1 ;Pipe (physical wave...
instr 99 ;Reverb unit
F-tables
f 10 0 256 -25 0 0.000006 12...
f 11 0 512 -25 0 0.00012475 ...
f 1 0 128 -25 0 0.002272727...
test.csd 01pvsanal_pvsynth.csd Pipe_Synth
LE Controller
New
Stop All
[FFT] [qutescope.cp... Pipe_Synth.csd... LE Controller Live Event – New Live Event – Line Live Event – Short

```
<CsoundSynthesizer>
<CsOptions>
-fdm0 -odac -b1024 -B2048 --expressi
</CsOptions>
<CsInstruments>
;Physical Waveguide Midi synth with Qu
;Modified for QuteCsound examples by

sr      = 48000
ksmps   = 512
nchnls  = 2
```

LE Controller

Show	Play	Loop	Sync	Name	Loop length	Loop Range	Tempo
☑	▶	☐		New	8	1-1	100
☑	▶	☐		Line	8	1-1	60
☑	▶	☐		Short	8	1-1	60

Live Event -- New

Menu View Sheet Tempo 100.00 Loop Length 8.000

	Event	p1 (instr)	p2 (start)	p3 (dur)	p4	p5
1	i	1	0	0.4	64	
2	i	1	0.5	0.4	68	
3	i	1	1	0.4	71	
4	i	1	1.5	0.4	69	
5	i	1	2	0.4	60	
6	i	1	2.5	0.4	61	
7	i	1	3	0.4	61	
8	i	1	3.5	0.4	68	
9	i	1	4	0.4	61	
10	i	1	4.5	0.4	67	
11	i	1	5	0.4	68	
12	i	1	5.5	0.1	97	
13	i	1	6	0.1	60	
14	i	1	6.5	0.1	73	
15	i	1	7	0.1	66	
16	i	1	7.5	0.1	61	
17						

Live Event -- Line

Menu View Sheet Tempo 60.00 Loop Length 8.000

	Event	p1 (instr)	p2 (start)	p3 (dur)	p4	p5
1	i	1	4.5	0.4	67	
2	i	1	5	0.4	68	
3	i	1	5.5	0.1	97	

Live Event -- Short

Menu View Sheet Tempo 60.00 Loop Length 8.000

	Event	p1 (instr)	p2 (start)	p3 (dur)	p4	p5
1	i	1	4.5	0.4	67	
2	i	1	5	0.4	68	
3	i	1	5.5	0.1	97	
4	i	1	6	0.1	60	
5	i	1	6.5	0.1	73	
6	i	1	7	0.1	66	
7	i	1	7.5	0.1	61	

You will find more detailed information and video tutorials in the CsoundQt home page at http://qutecsound.sourceforge.net.

49. WINXOUND

WinXound Description:
WinXound is a free and open-source Front-End GUI Editor for CSound 5, CSoundAV, CSoundAC, with Python and Lua support, developed by Stefano Bonetti.
It runs on Microsoft Windows, Apple Mac OsX and Linux.
WinXound is optimized to work with the new CSound 5 compiler.

WinXound Features:

- Edit CSound, Python and Lua files (csd, orc, sco, py, lua) with Syntax Highlight and Rectangular Selection;
- Run CSound, CSoundAV, CSoundAC, Python and Lua compilers;
- Run external language tools (QuteCsound, Idle, or other GUI Editors);
- CSound analysis user friendly GUI;
- Integrated CSound manual help;
- Possibilities to set personal colors for the syntax highlighter;
- Convert orc/sco to csd or csd to orc/sco;
- Split code into two windows horizontally or vertically;
- CSound csd explorer (File structure for Tags and Instruments);
- CSound Opcodes autocompletion menu;
- Line numbers;
- Bookmarks;
- ...and much more ... (Download it!)

Web Site and Contacts:
- Web: winxound.codeplex.com
- Email: stefano_bonetti@tin.it (or stefano_bonetti@alice.it)

REQUIREMENTS

System requirements for Microsoft Windows:

- Supported: Xp, Vista, Seven (32/64 bit versions);
- (Note: For Windows Xp you also need the Microsoft Framework .Net version 2.0 or major. You can download it from www.microsoft.com site);
- CSound 5: http://sourceforge.net/projects/csound - (needed for CSound and LuaJit compilers);
- Not requested but suggested: CSoundAV by Gabriel Maldonado (http://www.csounds.com/maldonado/);
- Requested to work with Python: Python compiler (http://www.python.org/download/)

System requirements for Apple Mac OsX:

- Osx 10.5 or major;
- CSound 5: http://sourceforge.net/projects/csound - (needed for CSound compiler);

System requirements for Linux:

- Gnome environment or libraries;
- Please, read carefully the "ReadMe" file in the source code.

INSTALLATION AND USAGE

Microsoft Windows Installation and Usage:

- Download and install the Microsoft Framework .Net version 2.0 or major (only for Windows Xp);

- Download and install the latest version of CSound 5 (http://sourceforge.net/projects/csound);
- Download the WinXound zipped file, decompress it where you want (see the (*)note below), and double-click on "WinXound_Net" executable;
- (*)note: THE WINXOUND FOLDER MUST BE LOCATED IN A PATH WHERE YOU HAVE FULL READ AND WRITE PERMISSION (for example in your User Personal folder).

Apple Mac OsX Installation and Usage:

- Download and install the latest version of CSound 5 (http://sourceforge.net/projects/csound);
- Download the WinXound zipped file, decompress it and drag WinXound.app to your Applications folder (or where you want). Launch it from there.

Linux Installation and Usage:

- Download and install the latest version of CSound 5 for your distribution;
- Ubuntu (32/64 bit): Download the WinXound zipped file, decompress it in a location where you have the full read and write permissions;
- To compile the source code:
  1) Before to compile WinXound you need to install:
  - gtkmm-2.4 (libgtkmm-2.4-dev) >= 2.12
  - vte (libvte-dev)
  - webkit-1.0 (libwebkit-dev)

  2) To compile WinXound open the terminal window, go into the uncompressed "winxound_gtkmm" directory and type:
  ```
  ./configure
  make
  ```

  3) To use WinXound without installing it:
  ```
  make standalone
  ./bin/winxound
  ```
  [Note: WinXound folder must be located in a path where you have full read and write permission.]

  4) To install WinXound:
  ```
  make install
  ```

Source Code:

- Windows: The source code is written in C# using Microsoft Visual Studio C# Express Edition 2008.
- OsX: The source code is written in Cocoa and Objective-C using XCode 3.2 version.
- Linux: The source code is written in C++ (Gtkmm) using Anjuta.

Note: *The TextEditor is entirely based on the wonderful SCINTILLA text control by Neil Hodgson (http://www.scintilla.org).*

Screenshots:

Look at: winxound.codeplex.com

Credits:
Many thanks for suggestions and debugging help to Roberto Doati, Gabriel Maldonado, Mark Jamerson, Andreas Bergsland, Oeyvind Brandtsegg, Francesco Biasiol, Giorgio Klauer, Paolo Girol,

Francesco Porta, Eric Dexter, Menno Knevel, Joseph Alford, Panos Katergiathis, James Mobberley, Fabio Macelloni, Giuseppe Silvi, Maurizio Goina, Andrés Cabrera, Peiman Khosravi, Rory Walsh and Luis Jure.

50. BLUE

blue is a Java-based music composition environment for use with Csound. It provides higher level abstractions such as a timeline, GUI-based instruments, score generating soundObjects like pianoRolls, scripting, and more. It is available at:

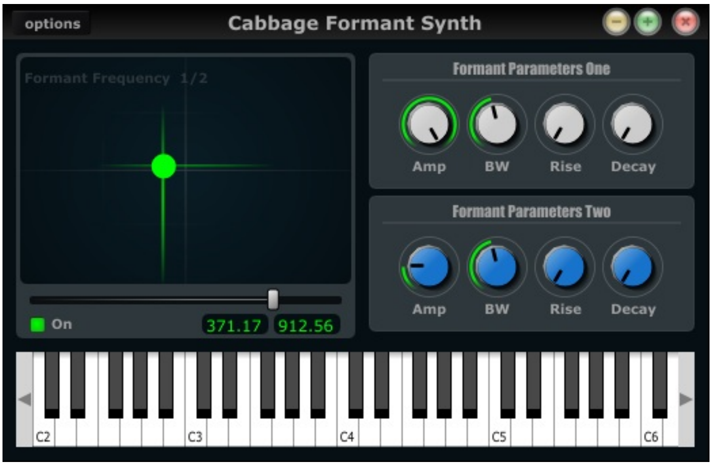
http://blue.kunstmusik.com

Cabbage is a software for prototyping and developing audio plugins with the Csound audio synthesis language. It provides Csound programmers with a simple albeit powerful toolkit for the development of cross-platform audio software. Pre-built binaries for Microsoft Windows and Apple OSX(Built on OSX 10.6) are available from the Cabbage google code homepage. You will also find a zipped archive of sample instruments.

This document will take you through the basics of using Cabbage. It starts with a look at features provided by the host and then moves on to some simple examples. The text concludes with a reference section for the various GUI controls available in Cabbage. It's assumed that the reader has some prior knowledge of Csound.

In order to use Cabbage you MUST have Csound installed. Cabbage is only available for the doubles version of Csound. This is the version that comes with the various installers so there shouldn't be any problems. If however you build your own version of Csound and don't build with the 'useDouble=1' options Cabbage will not work properly.

THE CABBAGE STANDALONE PLAYER



Most prototyping will be done in the Cabbage standalone host. This host lets you load and run Cabbage instruments, as seen in the screenshot above. Clicking on the options button will give you access to the following commands:

Open Cabbage Instrument

Use this command to open a cabbage instrument(Unified Csound file with a dedicated <Cabbage></Cabbage> section). You may open any .csd file you wish and add a Cabbage section yourself once it's open. If opening existing Csound instrument you will need to use the***-n*** command line options to tell Csound not to open any audio devices, as these are handled directly by Cabbage.

> On OSX users can open .csd files contained within plugins. Just select a .vst file instaed of a .csd file when opening. See the sections on exporting plugins for more information.

New Cabbage…

This command will help you create a new Cabbage instrument/effect. Cabbage instruments are synthesisers capable of creating sounds from scratch while effects process incoming audio. Effects can access the incoming audio by using the ***inch*** or ***ins*** opcodes. All effects have stereo inputs and stereo outputs. Instruments can access the incoming MIDI data in a host of different ways but the easiest is to pipe the MIDI data directly to instrument p-fields using the MIDI inter-op command line flags. Examples can be found in the examples folder.

The ***ctrl7*** opcode doesn't currently work so you should avoid using it in your instruments. ***ctrl7*** will be available in future versions

View Source Editor

This command will launch the integrated text editor. The text editor will always contain the text which corresponds to the instrument that is currently open. Each time a file is saved in the editor(Ctrl+S), Cabbage will automatically recompile the underlying Csound instrument and update any changes that have been made to the instruments GUI. The editor also features a Csound message console that can prove useful when debugging instruments.

Audio Settings

Clicking on the audio settings command will open the audio settings window. Here you can choose your audio/MIDI input/output devices. You can also select the sampling rate and audio buffer sizes. Small buffer sizes will reduce latency but might cause some clicks in the audio. Keep testing buffer sizes until you find a setting that works best for your PC.

> Cabbage hosts Csound instruments. It uses its own audio IO callbacks which will override any IO settings specified in the <CsOptions> sections of your Csound file.

Export…

This command will export your Cabbage instrument as a plugin.
Clicking ***synth*** or ***plugin*** will cause Cabbage to create a plugin file(with a .dll file extension) into teh same directory as teh csd file you are using. When ***exporting as*** Cabbage will prompt you to save your plugin in a set location, under a specific name. Once Cabbage has created the plugin it will make a copy of the current .csd file and locate it in the same folder as the plugin. This new .csd file will have the same name as the plugin and should ALWAYS be in the same directory as the plugin.

> You do not need to keep exporting instruments as plugins every time you modify them. You need only modify the associated source code. To simplify this task, Cabbage will automatically load the associated .csd file whenever you export as a plugin. On OSX Cabbage can open a plugin's .csd file directly by selecting the plugin when prompted to select a file to open.

Always on Top

This command lets you toggle ***Always on top*** mode. By default it is turned on. This means your Cabbage instrument will always appear on top of any other applications that are currently open.

Update Instrument

This command updates Cabbage. This is useful if you decide to use another editor rather the one provided. Just remember to save any changes made to your Cabbage instrument before hitting update.

Batch Convert

This command will let you convert a selection of Cabbage .csd files into plugins so you don't have to manually open and export each one.

> This feature is currently only available on Windows.

YOUR FIRST CABBAGE INSTRUMENTS

The following section illustrates the steps involved in building a simple Cabbage instrument. It's assumed that the user has some prior knowledge of Csound. When creating a Cabbage patch users must provide special xml-style tags at the top of a unified Csound file. The Cabbage specific code should be placed between an opening <Cabbage> and a closing </Cabbage> tag. You can create a new instrument by using the ***New Cabbage Instrument*** menu command. Select either a synth or an effect and Cabbage will automatically generate a basic template for you to work with.

Each line of Cabbage specific code relates to one graphical user interface(GUI) control only. Lines must start with the type of GUI control you wish to use, i.e, vslider, button, xypad, etc. Users then add identifiers to indicate how the control will look and behave. All parameters passed to identifiers are either strings denoted with double quotes or numerical values. Information on different identifiers and their parameters is given below in the reference section. Long lines can be broken up with a \ placed at the end of a line.

> This section does not go into details about each Cabbage control, nor does it show all available identifiers. Details about the various Cabbage controls can be found in reference section below.

A basic Cabbage synthesiser

Code to create the most basic of Cabbage synthesisers is presented below. This instrument uses the MIDI interop command line flags to pipe MIDI data directly to p-fields in instrument

1. In this case all MIDI pitch data is sent directly to p4, and all MIDI amplitude data is sent to p5. MIDI data been sent on channel 1 will cause instrument 1 to play. Data being sent on channel 2 will cause instrument 2 to play. It has been reported that the ***massign*** opcode does not work as expected with Cabbage. This is currently under investigation.

```
<Cabbage>
form size(400, 120), caption("Simple Synth"), pluginID("plu1")
keyboard bounds(0, 0, 380, 100)
</Cabbage>
<CsoundSynthesizer>
<CsOptions>
-n -d -+rtmidi=NULL -M0 --midi-key-cps=4 --midi-velocity-amp=5
</CsOptions>
<CsInstruments>
sr = 44100
ksmps = 64
nchnls = 2
0dbfs=1

instr 1
kenv linenr p5, 0.1, .25, 0.01
a1 oscil kenv*k1, p4, 1
outs a1, a1
endin

</CsInstruments>
<CsScore>
f1 0 1024 10 1
f0 3600
</CsScore>
</CsoundSynthesizer>
```

> You’ll notice that a ***-n*** and ***-d*** are passed to Csound in the CsOptions section. -n stops Csound from writing audio to disk. This must be used as Cabbage manages its own audio IO callbacks. The ***-d*** prevents any FLTK widgets from displaying. You will also notice that our instrument is stereo. ALL Cabbage instruments operate in stereo.

Controlling your Cabbage patch

The most obvious limitation to the above instrument is that users cannot interact directly with Csound. In order to do this one can use a Csound channel opcode and a Cabbage control such as a slider. Any control that is to interact with Csound must have a channel identifier.

When one supplies a channel name to the channel() identifier Csound will listen for data being sent on that channel through the use of the named channel opcodes. There are a few ways of retrieving data from the named channel bus in Csound, the most straightforward one being the chnget opcode. It’s defined in the Csound reference manual as:

```
kval chnget Sname
```

Sname is the name of the channel. This same name must be passed to the ***channel()*** identifier in the corresponding <Cabbage> section.

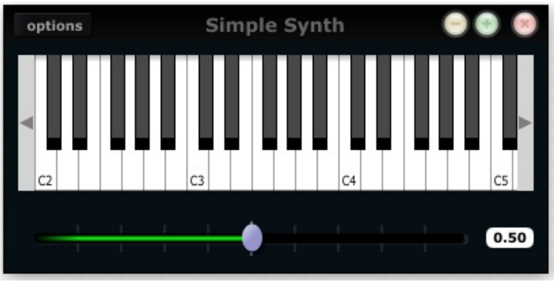
> At present Cabbage only works with the chnget/chnset method of sending and receiving channel data. invalue and outvalue won't work.

Our previous example can be modified so that a slider now controls the volume of our oscillator.

```
<Cabbage>
form size(400, 170), caption("Simple Synth"), pluginID("plu1")
hslider bounds(0, 110, 380, 50), channel("gain"), range(0, 1, .5), textBox(1)
keyboard bounds(0, 0, 380, 100)
</Cabbage>
<CsoundSynthesizer>
<CsOptions>
-n -d -+rtmidi=NULL -M0 --midi-key-cps=4 --midi-velocity-amp=5
</CsOptions>
<CsInstruments>
sr = 44100
ksmps = 64
nchnls = 2
0dbfs=1

instr 1
k1 chnget "gain"
kenv linenr p5, 0.1, 1, 0.1
a1 oscil kenv*k1, p4, 1
outs a1, a1
endin

</CsInstruments>
<CsScore>
```

```
f1 0 1024 10 1
f0 3600
</CsScore>
</CsoundSynthesizer>
```

In the example above we use a ***hslider*** control which is a horizontal slider. The bounds() identifier sets up the position and size of the widget. The most important identifier is ***channel("gain")***. It is passed a string called ***gain.*** This is the same string we pass to ***chnget*** in our Csound code. When a user moves the slider, the current position of the slider is sent to Csound on a channel named "gain". Without the channel() identifier no communication would take place between the Cabbage control and Csound. The above example also uses a MIDI keyboard that can be used en lieu of a real MIDI keyboard when testing plugins.



A basic Cabbage effect

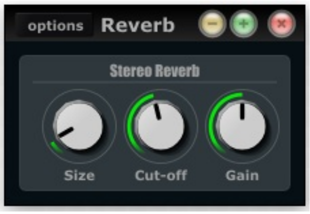
Cabbage effects are used to process incoming audio. To do so one must make sure they can access the incoming audio stream. Any of Csound's signal input opcodes can be used for this. The examples that come with Cabbage use both the ***ins*** and ***inch*** opcodes to retreive the incoming audio signal. The following code is for a simple reverb unit. It accepts a stereo input and outputs a stereo signal.

```
<Cabbage>
form caption("Reverb") size(230, 130)
groupbox text("Stereo Reverb"), bounds(0, 0, 200, 100)
rslider channel("size"), bounds(10, 25, 70, 70), text("Size"), range(0, 2, 0.2)
rslider channel("fco"), bounds(70, 25, 70, 70), text("Cut-off"), range(0, 22000, 10000)
rslider channel("gain"), bounds(130, 25, 70, 70), text("Gain"), range(0, 1, 0.5)
</Cabbage>
<CsoundSynthesizer>
<CsOptions>
-d -n
</CsOptions>
<CsInstruments>
; Initialize the global variables.
sr = 44100
ksmps = 32
nchnls = 2

instr 1
kfdback chnget "size"
kfco chnget "fco"
kgain chnget "gain"
ainL inch 1
ainR inch 2
aoutL, aoutR reverbsc ainL, ainR, kfdback, kfco
outs aoutL*kgain, aoutR*kgain
endin

</CsInstruments>
<CsScore>
f1 0 4096 10 1
i1 0 1000
</CsScore>
</CsoundSynthesizer>
```

The above instrument uses 3 sliders to control

- the reverb size
- the cut-off frequency for the internal low-pass filters set up on the different delay lines
- overall gain.

The range() identifier is used with each slider to specify the min, max and starting value of the sliders.



> If you compare the two score sections in the above instruments you'll notice that the synth instrument doesn't use any i-statement. Instead it uses an ***f0 3600***. This tells Csound to wait for 3600 seconds before exiting. Because the instrument is to be controlled via MIDI we don't need to use an i-statement in the score. In the other example we use an i-statement with a long duration so that the effect runs without stopping for a long time.

Exporting your instruments as plugins

Once you have created your instruments you will need to export them as plugins if you want them to be seen by other host applications. When you export in Cabbage it will create a plugin file that will have the same name as the csd file you are currently working on. In your plugin host you will need to add the directory that contains your Cabbage plugins and csd files.

> In order to make future changes to the instrument you only need to edit the associated .csd file. For instance, if you have a plugin called "SavageCabbage.dll" and you wish to make some changes, you only have to edit the corresponding "SavageCabbage.csd" file. In order to see the changes in your plugin host you will need to delete and re-instantiate the plugin from the track. Your changes will be seen once you re-instantiate the plugin.

CABBAGE REFERENCE

Each and every Cabbage control has a numbers of possible identifiers that can be used to tell Cabbage how it will look and behave. Identifiers with parameters enclosed in quote marks must be passed a quoted string. Identifiers containing parameters without quotes must be passed numerical values. All parameters except ***pos()*** have default values and are therefore optional. In the reference tables below any identifiers enclosed in square brackets are optional.

As pos() and size() are used so often they can be set in one go using the bounds() identifier:

bounds(x, y, width, height): bounds takes integer values that set position and size on screen(in pixels)

Below is a list of the different GUI controls currently available in Cabbage. Controls can be split into two groups, interactive controls and non-interactive controls. The non-interactive controls such as group boxes and images don't interact in any way with either Csound or plugin hosts. The interactive controls such as sliders and buttons do interact with Csound. Each interactive control that one inserts into a Cabbage instrument will be accessible in a plugin host if the instrument has been exported as a plugin. The name that appears beside each native slider in the plugin host will be the assigned channel name for that control.

> In order to save space in the following reference section ***bounds()*** will be used instead of ***pos()*** and ***size()*** wherever applicable.

Form

```
form caption("title"), size(Width, Height), pluginID("plug")
```

Form creates the main application window. pluginID() is the only required identifier. The default values for size are 600x300.

`caption`: The string passed to caption will be the string that appears on the main application window.

`size(Width, Height)`: integer values denoted the width and height of the form.

`pluginID("plug")`: this unique string must be four characters long. It is the ID given to your plugin when loaded by plugin hosts.

> Every plugin must have a unique pluginID. If two plugins share the same ID there will be conflicts when trying to load them into a plugin host.

Example:

```
form  caption("Simple Synth"), pluginID("plu1")
```

GroupBox

```
groupbox bounds(x, y, width, height), text("Caption")
```

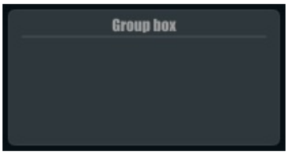
Groupbox creates a container for other GUI controls. They do not communicate with Csound but can be useful for organising widgets into panels.

bounds(x, y, width, height): integer values that set position and size on screen(in pixels)

text("caption"): "caption" will be the string to appear on the group box

Example:

```
groupbox bounds(0, 0, 200, 100), text("Group box")
```



Keyboard

```
keyboard bounds(x, y, width, height)
```

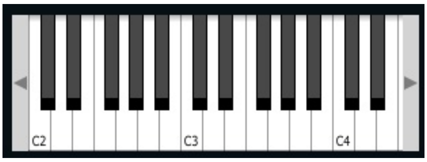
Keyboard create a piano keyboard that will send MIDI information to your Csound instrument. This component can be used together with a hardware controller. Pressing keys on the actual MIDI keyboard will cause the on-screen keys to light up.

bounds(x, y, width, height): integer values that set position and size on screen(in pixels)

> you can only use one MIDI keyboard component with each Cabbage instrument. Also note that the keyboard can be played at different velocities depending on where you click on the key with your mouse. Clicking at the top of the key will cause a smaller velocity while clicking on the bottom will cause the note to sound with full velocity. The keyboard control is only provided as a quick and easy means of testing plugins in Cabbage. Treating it as anything more than that could result in severe disappointment!

Example:

```
keyboard bounds(0, 0, 200, 100)
```



CsoundOutput

```
csoundoutput bounds(x, y, width, height), text("name")
```

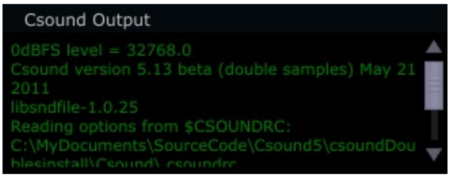
csoundoutput will let you view the Csound output console within your instrument's GUI, useful when 'de-slugging'(debugging in Cabbage is known as de-slugging!) Cabbage instruments.

bounds(x, y, width, height): integer values that set position and size on screen(in pixels)

text("name"): "name" will be the text that appears on the top of the check box.

Example:

```
csoundoutput bounds(210, 00, 340, 145), text("Csound Output")
```



```
Csound Output
0dBFS level = 32768.0
Csound version 5.13 beta (double samples) May 21
2011
libsndfile-1.0.25
Reading options from $CSOUNDRC:
C:\MyDocuments\SourceCode\Csound5\csoundDou
blesinstall\Csound\.csoundrc
```

Image

```
image bounds(x, y, width, height), file("file name"), shape("type"), colour("colour")\
        outline("colour"), line(thickness)
```

Image creates a static shape or graphic. It can be used to show pictures or it can be used to draw simple shapes. If you wish to display a picture you must pass the file name to the file() identifier. The file MUST be in the same directory as your Cabbage instrument. If you simply wish to draw a shape you can choose a background colour with colour() and an outline colour with outline(). line() will let you determine the thickness of the outline.

bounds(x, y, width, height): integer values that set position and size on screen(in pixels)

file("filename"): "filename" is the name of the image to be displayed on the control

shape("type");: "shape" must be either "round"(with rounded corners, default), "sharp"(with sharp corners), or "ellipse"(an elliptical shape)

colour("colour"): This sets the colour of the image if no file name is given with the file identifier. Any CSS or HTML colour string can be passed to this identifier.

outline("colour"): This sets the outline colour of the image/shape. Any CSS or HTML colour string can be passed to this identifier.

line(thickness): This sets the line thickness in pixels.

Example:

```
image bounds(0, 10, 260, 190), colour("white")
image bounds(5, 15, 250, 180), colour("brown")
image bounds(30, 30, 200, 150), file("logo_cabbage_sw_no_text.png")
```

Sliders

```
hslider bounds(x, y, width, height), channel("chanName")[, caption("caption") \
        text("name"), textBox(on/off), range(min, max, value, skew, incr) \
        midCtrl(Channel, Ctrlnum), colour("colour")]
```

Slider can be used to create an on-screen slider. Data can be sent to Csound on the channel specified through the chanName string. Presented above is the syntax for a horizontal slider, i.e., ***hslider***. In order to change it to another slider type simple substitute hslider with the appropriate identifier as outlined below.

bounds(x, y, width, height): integer values that set position and size on screen(in pixels)

channel("chanName"): "chanName" is the name of the channel upon which to communicate with Csound(see examples above).

caption("caption"): This identifier lets you place your control within a groupbox. "caption" is the text that will appear on groupbox. This identifier is useful for naming and containing controls.

range(min, max, value, skew, incr): the first 2 parameters are required. The rest are optional. The first two parameters let you set the minimum value and the maximum value. The next parameter determines the initial value of the slider. The next allows you to adjust the skew factor. Tweaking the skew factor can cause the slider to output values in a non linear fashion. A skew of 0.5 will cause the slider to output values in an exponential fashion. A skew of 1 is the default value, which causes the slider to behave is a typical linear form.

> For the moment ***min*** must be less than ***max***. In other words you can't invert the slider. Also note that skew defaults to 1 when the slider is being controlled by MIDI.

text("name"): The string passed in for "name" will appear on a label beside the slider. This is useful for naming sliders.

textBox(on/off): textbox takes a 0 or a 1. 1 will cause a text box to appear with the sliders values. Leaving this out will result in the numbers appearing automatically when you hover over the sliders with your mouse.

midCtrl(channel, Ctrlnum) : channel must be a valid midi channel, while controller num should be the number of the controller you wish to use. This identifier only works when running your instruments within the Cabbage standalone player.

colour("colour"): This sets the colour of the image if a file name is not passed to file. Any CSS or HTML colour string can be passed to this identifier.

Slider types:

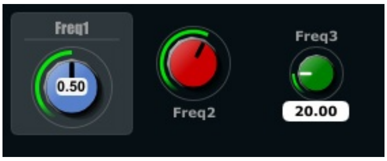
hslider: horizontal slider

vslider: vertical slider

rslider: rotary slider

Example:

```
rslider bounds(0, 110, 90, 90), caption("Freq1"), channel("freq2"), colour("cornflowerblue")\
        range(0, 1, .5), midictrl(0, 1)
rslider bounds(100, 120, 70, 70), text("Freq2"), channel("freq2"), colour("red")\
         range(0, 1, .5), midictrl(0, 1)
rslider bounds(190, 120, 70, 70), text("Freq3"), channel("freq2"), colour("green")\
        text("Freq3"), textbox(1)
```



Button

```
button bounds(x, y, width, height), channel("chanName")[, text("offCaption","onCaption")\
        caption("caption"), value(val)]
```

Button creates a button that can be used for a whole range of different tasks. The "channel" string identifies the channel on which the host will communicate with Csound. "OnCaption" and "OffCaption" determine the strings that will appear on the button as users toggle between two states, i.e., 0 or 1. By default these captions are set to "On" and "Off" but the user can specify any strings they wish. Button will constantly toggle between 0 and 1.

bounds(x, y, width, height): integer values that set position and size on screen(in pixels)

channel("chanName"): "chanName" is the name of the channel upon which to communicate with Csound(see examples above).

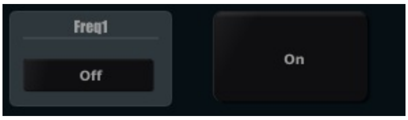
caption("caption"): This identifier lets you place your control within a groupbox. "caption" is the text that will appear on group box. This identifier is useful for naming and containing controls.

text("offCaption", "onCaption"): The text identifier must be passed at least one string argument. This string will be the one that will appear on the button. If you pass two strings to text() the button will toggle between the two string each time it is pushed.

value(val): val sets the initial state of the control

Example:

```
button bounds(0, 110, 120, 70), caption("Freq1"), text("On", "Off"), channel("freq2"), value(1)
button bounds(150, 110, 120, 70), text("On", "Off"), channel("freq2"), value(0)
```



CheckBox

```
checkbox bounds(x, y, width, height), channel("chanName")[, text("name"), value(val), caption("Caption")]
```

Checkbox creates a checkbox which functions like a button only the associated caption will not change when the user checks it. As with all controls capable of sending data to an instance of Csound the channel string is the channel on which the control will communicate with Csound.

channel("chanName"): "chanName" is the name of the channel upon which to communicate with Csound(see examples above).

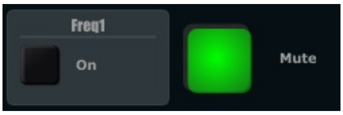
caption("caption"): This identifier lets you place your control within a groupbox. "caption" is the text that will appear on groupbox. This identifier is useful for naming and containing controls.

text("name"): "name" will be the text that appears beside the checkbox.

value(val): val sets the initial state of the control

Example:

```
checkbox bounds(0, 110, 120, 70), caption("Freq1"), text("On"), channel("freq2")
checkbox bounds(130, 110, 120, 70), text("Mute"), channel("freq2"), value(1)
```



ComboBox

```
combobox bounds(x, y, width, height), channel("chanName")[, value(val), items("item1", "item2", ...)\
        caption("caption")]
```

Combobox creates a drop-down list of items which users can choose from. Once the user selects an item, the index of their selection will be sent to Csound on a channel named by the channel string. The default value is 0.

bounds(x, y, width, height): integer values that set position and size on screen(in pixels)

channel("chanName"): "chanName" is the name of the channel upon which to communicate with Csound(see examples above).

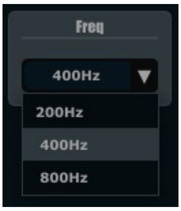
items("item1", "item2", etc): list of items that will populate the combobox. Each item has a corresponding index value. The first item when selected will send a 1, the second item a 2, the third a 3 etc.

value(val): val sets the initial state of the control

caption("caption"): This identifier lets you place your control within a groupbox. "caption" is the text that will appear on groupbox. This identifier is useful for naming and containing controls.

Example:

```
combobox bounds(0, 110, 120, 70), channel"freq"), caption("Freq"), items("200Hz", "400Hz", "800Hz"),
value(2)
```



> Combo boxes are proving a little troublesome when used in plugin hosts. We hope to resolve this issue shortly. In the mean time one can use a slider and split it into different regions. Note that all GUI controls appear as sliders when shown as native controls in a plugin host.

XYPad

```
xypad bounds(x, y, width, height), channel("chanName")[, rangex(min, max, val)\
        rangey(min, max, val), text("name")]
```

xypad is an x/y controller that sends data to Csound on two named channels. The first channel transmits the current position of the ball on the X axis, while the second transmits the position of the ball on the Y axis. If you turn on automation via the checkbox located on the bottom left of the xypad you can throw the ball from edge to edge. Once the ball is in full flight you can control the speed of the ball using the XYpad slider.

bounds(x, y, width, height): integer values that set position and size on screen(in pixels)

channel("chanName"): "chanName" is the name of the channel in which to communicate with Csound(see examples above).

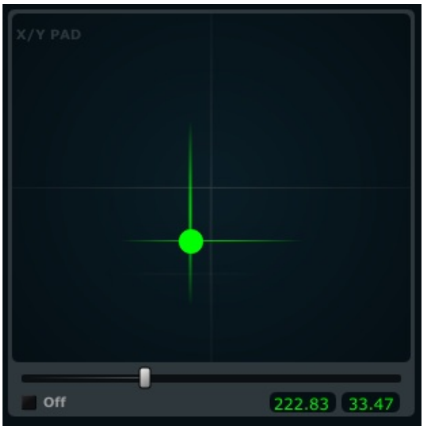
text("name"): "name" will be the text that appears on the top right hand side of the XYpad surface.

rangex(min, max, value): sets the range of the X axis. The first 2 parameters are required. The third is optional. The first two parameters let you set the minimum value and the maximum value. The next parameter determines the initial value.

rangey(min, max, value): sets the range of the Y axis. The first 2 parameters are required. The third is optional. The first two parameters let you set the minimum value and the maximum value. The next parameter determines the initial value.

Example:

```
xypad bounds(0, 0, 300, 300), text("X/Y PAD"), rangex(0, 500, 250), rangey(0, 100, 25)
```



QUICK REFERENCE

This quick reference table table gives a list of the valid identifiers for each Cabbage control.

Parameter	form	slider	button	checkbox	groupbox	combobox	xypad	image	csoundoutput

TROUBLESHOOTING, FAQS, TIPS AND TRICKS

- ***Why doesn’t my VST host see my Cabbage plugins?*** The most likely reason is that you have not added the directory containing your plugins to your host’s preferences. Most hosts will allow you to choose the folders that contain plugins. If you don’t set the Cabbage plugin directory then the host has no idea where your Cabbage plugins are located.
- ***Why doesn’t my Cabbage plugin load?*** The most likely reason a plugin will not load is because there are errors in the Csound code. Cabbage plugins will load regardless of errors in the Cabbage code, but errors in the Csound code will stop Csound from compiling successfully and prevent the plugin from loading. Always make sure that the Csound code is error free before exporting.
- ***One mega plugin or several smaller ones?*** It’s a good idea to split multi-effects instruments into separate plugins. This allows greater modularity within you plugin host and can often lead to less demand on your PC’s CPU.
- ***Mixing effects and instruments?*** Adding an effect processor to a plugin instrument might seem like a good idea. For instance you might add some reverb to the output of your FM synth to create some nice presence. In general however it is best to keep them separate. Plugin instruments demand a whole lot more CPU than their effects siblings. Performance will be a lot smoother if you split the two processes up and simply send the output of your synthesiser into an instance of a Cabbage reverb effect plugin.
- ***What’s up? My plugin makes a load of noise?*** If you have nchnls set to 1 thre will be noise sent to the second, or right channel. Make sure that nchnls is ALWAYS set to 2! Also be careful when dealing with stereo input. If you try to access the incoming signal on the right channel but you don't have any audio going to the right channel you may experience some noise.
- ***I can’t tell whether my sliders are controlling anything?!*** There will be times when moving sliders or other interactive controls just doesn’t do what you might expect. The best way to de-slug Cabbage instruments is to use the ***printk2*** opcode in Csound. For instance if a slider is not behaving as expected make sure that Csound is receiving data from the slider on the correct channel. Using the code below should print the values of the slider to the Csound output console each time you move it. If not, then you most likely have the wrong channel name set.

```
(...)
k1 chnget "slider1"
printk2 k1
(...)
```

- ***What gives? I’ve checked my channels and they are consistent, yet moving my sliders does nothing?*** Believe it or not we have come across some cases of this happening! In all cases it was due to the fact that the chosen channel name contained a /. Please try to use plain old letters for your channel names. Avoid using any kind of mathematical operators or fancy symbols and everything should be Ok.
- ***Can I use nchnls to determine the number of output channels in my plugin?*** Currently all Cabbage plugins are stereo by default. We are looking into ways of making plugins multichannel but limitations in the VST SDK are proving to be a stumbling block. It is something we are committed to finding a fix for.
- ***Can I use Csound MACROs in the <Cabbage> section of my csd file?*** I’m afraid not. The Cabbage section of your csd file is parsed by Cabbage’s own parser therefore it will not understand any Csound syntax whatsoever.
- ***I’ve built some amazing instruments, how do I share them with the world?!*** Easy. Send me(rory walsh at ear dot ie), your instruments and I will add them to the Cabbage examples so that other Cabbage users can have a go.

52. CSOUND VIA TERMINAL

Whilst many of us now interact with Csound through one of its many front-ends which provide us with an experience more akin the that of mainstream software, new-comers to Csound should bear in mind that there was a time when the only way running Csound was from the command line using the Csound command. In fact we must still run Csound in this way but front-ends do this for us usually via some toolbar button or widget. Many people still prefer to interact with Csound from a terminal window and feel this provides a more 'naked' and honest interfacing with the program. Very often these people come from the group of users who have been using Csound for many years, form the time before front-ends. It is still important for all users to be aware of how to run Csound from the terminal as it provides a useful backup if problems develop with a preferred front-end.

THE CSOUND COMMAND

The Csound command follows the format:

```
csound [performance_flags] [input_orc/sco/csd]
```

Executing 'csound' with no additional arguments will run the program but after a variety of configuration information is printed to the terminal we will be informed that we provided "insufficient arguments" for Csound to do anything useful. This action can still be valid for first testing if Csound is installed and configured for terminal use, for checking what version is installed and for finding out what performance flags are available without having to refer to the manual.

Performance flags are controls that can be used to define how Csound will run. All of these flags have defaults but we can make explicitly use flags and change these defaults to do useful things like controlling the amount of information that Csound displays for us while running, activating a MIDI device for input, or altering buffer sizes for fine tuning realtime audio performance. Even if you are using a front-end, command line flags can be manipulated in a familiar format usually in 'settings' or 'preferences' menu. Adding flags here will have the same effect as adding them as part of the Csound command. To learn more about Csound's command line flags it is best to start on the page in the reference manual where they are listed and described by category.

Command line flags can also be defined within the <CsOptions> </CsOptions> part of a .csd file and also in a file called .csoundrc which can be located in the Csound home program directory and/or in the current working directory. Having all these different options for where esentially the same information is stored might seem excessive but it is really just to allow flexibiliy in how users can make changes to how Csound runs, depending on the situation and in the most efficient way possible. This does however bring up one one issue in that if a particular command line flag has been set in two different places, how does Csound know which one to choose? There is an order of precedence that allows us to find out.

Beginning from its own defaults the first place Csound looks for additional flag options is in the .csoundrc file in Csound's home directory, the next is in a .csoundrc file in the current working directory (if it exists), the next is in the <CsOptions> of the .csd and finally the Csound command itself. Flags that are read later in this list will overwrite earlier ones. Where flags have been set within a front-end's options, these will normally overwrite any previous instructions for that flag as they form part of the Csound command. Often a front-end will incorporate a check-box for disabling its own inclusion of flag (without actually having to delete them from the dialogue window).

After the command line flags (if any) have been declared in the Csound command, we provide the name(s) of out input file(s) - originally this would have been the orchestra (.orc) and score (.sco) file but this arrangement has now all but been replaced by the more recently introduced .csd (unified orchestra and score) file. The facility to use a separate orchestra and score file remains however.

For example:

```
Csound -d -W -osoundoutput.wav inputfile.csd
```

will run Csound and render the input .csd 'inputfile.csd' as a wav file ('-W' flag) to the file 'soundoutput.wav' ('-o' flag). Additionally displays will be suppressed as dictated by the '-d' flag. The input .csd file will need to be in the current working directory as no full path has been provided. the output file will be written to the current working directory of SFDIR if specified.

11 CSOUND UTILITIES

53. CSOUND UTILITIES

Csound comes bundled with a variety of additional utility applications. These are small programs that perform a single function, very often with a sound file, that might be useful just before or just after working with the main Csound program. Originally these were programs that were run from the command line but many of Csound front-ends now offer direct access to many of these utilities through their own utilities menus. It is useful to still have access to these programs via the command line though, if all else fails.

The standard syntax for using these programs from the command line is to type the name of the utility followed optionally by one or more command line flags which control various performance options of the program - all of these will have useable defaults anyway - and finally the name of the sound file upon which the utility will operate.

```
utility_name [flag(s)] [file_name(s)]
```

If we require some help or information about a utility and don't want to be bothered hunting through the Csound Manual we can just type the the utility's name with no additional arguments, hit enter and the commmand line response will give us some information about that utility and what command line flags it offers. We can also run the utility through Csound - perhaps useful if there are problems running the utility directly - by calling Csound with the -U flag. The -U flag will instruct Csound to run the utility and to interpret subsequent flags as those of the utility and not its own.

```
Csound -U utility_name [flag(s)] [file_name(s)]
```

SNDINFO

As an example of invoking one of these utilities form the command line we shall look at the utility 'sndinfo' (sound information) which provides the user with some information about one or more sound files. 'sndinfo' is invoked and provided with a file name thus:

```
sndinfo /Users/iainmccurdy/sounds/mysound.wav
```

If you are unsure of the file address of your sound file you can always just drag and drop it into the terminal window. The output should be something like:

```
util sndinfo:
/Users/iainmccurdy/sounds/mysound.wav:
 srate 44100, stereo, 24 bit WAV, 3.335 seconds
 (147078 sample frames)
```

'sndinfo' will accept a list of file names and provide information on all of them in one go so it may prove more efficient gleaning the same information from a GUI based sample editor. We also have the advantage of begin able to copy and paste from the terminal window into a .csd file.

ANALYSIS UTILITIES

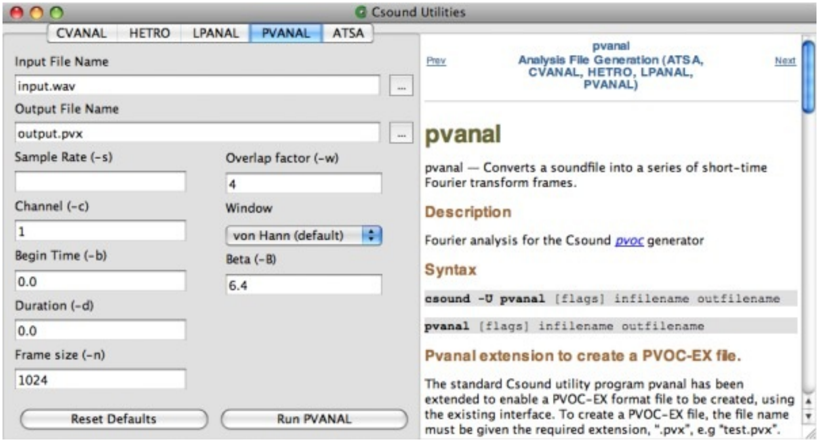
Although many of Csound's opcodes already operate upon commonly encountered sound file formats such as 'wav' and 'aiff', a number of them require sound information in more specialised and pre-analysed formats and for this Csound provides the sound analysis utilities atsa, cvanal, hetro, lpanal and pvanal. By far the most commonly used of these is pvanal which, although originally written to provide analysis files for pvoc and its generation of opcodes, has now been extended to be able to generate files in the pvoc-ex (.pvx) format for use with the newer 'pvs' streaming pvoc opcodes.

This time as well as requiring an input sound file for analysis we will need to provide a name (and optionally the full address) for the output file. Using pvanal's command flags we can have full control over typical FFT conversion parameters such as FFT size, overlap, window type etc. as well as additional options that may prove useful such as the ability to select a fragment of a larger sound file for the analysis. In the following illustration we shall make use of just one flag, -s, for selecting which channel of the input sound file to analyse, all other flag values shall assume their default values which should work fine in most situations.

```
pvanal -s1 mysound.wav myanalysis.pvx
```

pvanal will analyse the first (left if stereo) channel of the input sound file 'mysound.wav' (and in this case as no full address has been provided it will need to be in either the current working directory or SSDIR), and a name has been provided for the output file 'myanalysis.pvx', which, as no full address has been given, will be placed in the current working directory. While pvanal is running it will print a running momentary and finally inform us once the process is complete.

If you use CsoundQT you can have direct access to pvanal with all its options through the 'utilities' button in the toolbar. Once opened it will reveal a dialogue window looking something like this:



Input File Name: input.wav

Output File Name: output.pvx

Sample Rate (-s):

Channel (-c): 1

Begin Time (-b): 0.0

Duration (-d): 0.0

Frame size (-n): 1024

Overlap factor (-w): 4

Window: von Hann (default)

Beta (-B): 6.4

pvanal
Analysis File Generation (ATSA, CVANAL, HETRO, LPANAL, PVANAL)

pvanal

pvanal — Converts a soundfile into a series of short-time Fourier transform frames.

Description

Fourier analysis for the Csound *pvoc* generator

Syntax

```
csound -U pvanal [flags] infilename outfilename
```

```
pvanal [flags] infilename outfilename
```

Pvanal extension to create a PVOC-EX file.

The standard Csound utility program pvanal has been extended to enable a PVOC-EX format file to be created, using the existing interface. To create a PVOC-EX file, the file name must be given the required extension, ".pvx", e.g "test.pvx".

Especially helpful is the fact that we are also automatically provided with pvanal's manual page.

FILE CONVERSION UTILITIES

The next group of utilities, het_import, het_export, pvlook, pv_export, pv_import, sdif2ad and srconv facilitate file conversions between various types. Perhaps the most interesting of these are pvlook, which prints to the terminal a formatted text version of a pvanal file - useful to finding out exactly what is going on inside individual analysis bins, something that may be of use when working with the more advanced resynthesis opcodes such as pvadd or pvsbin. srconv can be used to convert the sample rate of a sound file.

MISCELLANEOUS UTILITIES

A final grouping gathers together various unsorted utilities: cs, csb64enc, envext, extractor, makecsd, mixer, scale and mkdb. Most interesting of these are perhaps extractor which will extract a user defined fragment of a sound file which it will then write to a new file, mixer which mixes together any number of sound files and with gain control over each file and scale which will scale the amplitude of an individual sound file.

It has been seen that the Csound utilities offer a wealth of useful, but often overlooked, tools to augment our work with Csound. Whilst some of these utilities may seem redundant now that most of us have access to fully featured 3rd-party sound editing software, it should be borne in mind that many of these utilities were written in the 1980s and early 90s when such tools were less readily available.

12 CSOUND AND OTHER PROGRAMMING LANGUAGES

54. THE CSOUND API

An application programming interface (API) is an interface provided by a computer system, library or application that allows users to access functions and routines for a particular task. It gives developers a way to harness the functionality of existing software within a host application. The Csound API can be used to control an instance of Csound through a series of different functions thus making it possible to harness all the power of Csound in one's own applications. In other words, almost anything that can be done within Csound can be done with the API. The API is written in C, but there are interfaces to other languages as well, such as Python, C++ and Java.

To use the Csound C API, you have to include csound.h in your source file and to link your code with libcsound. Here is an example of the csound command line application written using the Csound C API:

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

int main(int argc, char **argv)
{
  CSOUND *csound = csoundCreate(NULL);
  int result = csoundCompile(csound, argc, argv);
  if (result == 0) {
    result = csoundPerform(csound);
  }
  csoundDestroy(csound);
  return (result >= 0 ? 0 : result);
}
```

First we create an instance of Csound. To do this we call `csoundCreate`() which returns an opaque pointer that will be passed to most Csound API functions. Then we compile the orc/sco files or the csd file given as input arguments through the argv parameter of the main function. If the compilation is successful (result == 0), we call the `csoundPerform`() function. `csoundPerform`() will cause Csound to perform until the end of the score is reached. When this happens csoundPerform() returns a non-zero value and we destroy our instance before ending the program.

On a linux system, with libcsound named libcsound64 (double version of the csound library), supposing that all include and library paths are set correctly, we would build the above example with the following command:

```
gcc -DUSE_DOUBLE -o csoundCommand csoundCommand.c -lcsound64
```

The C API has been wrapped in a C++ class for convenience. This gives the Csound basic C++ API. With this API, the above example would become:

```
#include <csound/csound.hpp>

int main(int argc, char **argv)
{
  Csound *cs = new Csound();
  int result = cs->Compile(argc, argv);
  if (result == 0) {
    result = cs->Perform();
  }
  return (result >= 0 ? 0 : result);
}
```

Here, we get a pointer to a Csound object instead of the csound opaque pointer. We call methods of this object instead of C functions, and we don't need to call csoundDestroy in the end of the program, because the C++ object destruction mechanism takes care of this. On our linux system, the example would be built with the following command:

```
g++ -DUSE_DOUBLE -o csoundCommandCpp csoundCommand.cpp -lcsound64
```

The Csound API has also been wrapped to other languages. The Csound Python API wraps the Csound API to the Python language. To use this API, you have to import the csnd module. The csnd module is normally installed in the site-packages or dist-packages directory of your python distribution as a csnd.py file. Our csound command example becomes:

```
import sys
import csnd

def csoundCommand(args):
    csound = csnd.Csound()
    arguments = csnd.CsoundArgVList()
    for s in args:
        arguments.Append(s)
    result = csound.Compile(arguments.argc(), arguments.argv())
    if result == 0:
        result = csound.Perform()
    return result

def main():
    csoundCommand(sys.argv)

if __name__ =='__main__':
    main()
```

We use a Csound object (remember Python has OOp features). Note the use of the CsoundArgVList helper class to wrap the program input arguments into a C++ manageable object. In fact, the Csound class has syntactic sugar (thanks to method overloading) for the Compile method. If you have less than six string arguments to pass to this method, you can pass them directly. But here, as we don't know the number of arguments to our csound command, we use the more general mechanism of the CsoundArgVList helper class.

The Csound Java API wraps the Csound API to the Java language. To use this API, you have to import the csnd package. The csnd package is located in the csnd.jar archive which has to be known from your Java path. Our csound command example becomes:

```
import csnd.*;

public class CsoundCommand
{
  private Csound csound = null;
  private CsoundArgVList arguments = null;

  public CsoundCommand(String[] args) {
    csound = new Csound();
    arguments = new CsoundArgVList();
    arguments.Append("dummy");
    for (int i = 0; i < args.length; i++) {
      arguments.Append(args[i]);
    }
    int result = csound.Compile(arguments.argc(), arguments.argv());
    if (result == 0) {
      result = csound.Perform();
    }
    System.out.println(result);
  }


  public static void main(String[] args) {
    CsoundCommand csCmd = new CsoundCommand(args);
  }
}
```

Note the "dummy" string as first argument in the arguments list. C, C++ and Python expect that the first argument in a program argv input array is implicitly the name of the calling program. This is not the case in Java: the first location in the program argv input array contains the first command line argument if any. So we have to had this "dummy" string value in the first location of the arguments array so that the C API function called by our csound.Compile method is happy.

This illustrates a fundamental point about the Csound API. Whichever API wrapper is used (C++, Python, Java, etc), it is the C API which is working under the hood. So a thorough knowledge of the Csound C API is highly recommended if you plan to use the Csound API in any of its different flavours. The main source of information about the Csound C API is the csound.h header file which is fully commented.

On our linux system, with csnd.jar located in /usr/local/lib/csound/java, our Java Program would be compiled and run with the following commands:

```
javac -cp /usr/local/lib/csound/java/csnd.jar CsoundCommand.java
java -cp /usr/local/lib/csound/java/csnd.jar:. CsoundCommand
```

There also exists an extended Csound C++ API, which adds to the Csound C++ API a CsoundFile class, the CsoundAC C++ API, which provides a class hierarchy for doing algorithmic composition using Michael Gogins' concept of music graphs, and API wrappers for the LISP, LUA and HASKELL languages.

For now, this chapter chapter we will focus on the basic C/C++ API, and the Python and Java API.

THREADING

Before we begin to look at how to control Csound in real time we need to look at threads. Threads are used so that a program can split itself into two or more simultaneously running tasks. Multiple threads can be executed in parallel on many computer systems. The advantage of running threads is that you do not have to wait for one part of your software to finish executing before you start another.

In order to control aspects of your instruments in real time your will need to employ the use of threads. If you run the first example found on this page you will see that the host will run for as long as `csoundPerform`() returns 0. As soon as it returns non-zero it will exit the loop and cause the application to quit. Once called, `csoundPerform`() will cause the program to hang until it is finished. In order to interact with Csound while it is performing you will need to call csoundPerform() in a separate unique thread.

When implementing threads using the Csound API, we must define a special performance function thread. We then pass the name of this performance function to `csoundCreateThread`(), thus registering our performance-thread function with Csound. When defining a Csound performance-thread routine you must declare it to have a return type uintptr_t, hence it will need to return a value when called. The thread function will take only one parameter, a pointer to void. This pointer to void is quite important as it allows us to pass important data from the main thread to the performance thread. As several variables are needed in our thread function the best approach is to create a user defined data structure that will hold all the information your performance thread will need. For example:

```
typedef struct{
/*result of csoundCompile()*/
int result;
/*instance of csound*/
CSOUND* csound;
/*performance status*/
bool PERF_STATUS;
}userData;
```

Below is a basic performance-thread routine. `*data` is cast as a `userData` data type so that we can access its members.

```
uintptr_t csThread(void *data)
{
userData* udata = (userData*)data;
if(!udata->result)
  {
  while((csoundPerformKsmps(udata->csound) == 0)&&
     (udata->PERF_STATUS==1));
  csoundDestroy(udata->csound);
  }
udata->PERF_STATUS = 0;
return 1;
}
```

In order to start this thread we must call the csoundCreateThread() API function which is declared in csound.h as:

```
void *csoundCreateThread(uintptr_t (*threadRoutine) (void *),void *userdata);
```

If you are building a command line program you will need to use some kind of mechanism to prevent int main() from returning until after the performance has taken place. A simple while loop will suffice.

The first example presented above can now be rewritten to include a unique performance thread:

```
#include <stdio.h>
#include "csound.h"

uintptr_t csThread(void *clientData);
```

```
typedef struct {
int result;
CSOUND* csound;
int PERF_STATUS;
}userData;

int main(int argc, char *argv[])
{
void* ThreadID;
userData* ud;
ud = (userData *)malloc(sizeof(userData));
MYFLT* pvalue;
csoundInitialize(&argc, &argv, 0);
ud->csound=csoundCreate(NULL);
ud->result=csoundCompile(ud->csound,argc,argv);

if(!ud->result)  {
ud->PERF_STATUS=1;
ThreadID = csoundCreateThread(csThread, (void*)ud);
}
else{
return 0;
}

//keep performing until user presses enter
scanf("%d", &finish);
ud->PERF_STATUS=0;
csoundDestroy(ud->csound);
free(ud);
return 1;
}

//performance thread function
uintptr_t csThread(void *data)
{
  userData* udata = (userData*)data;
      if(!udata->result)
        {
        while((csoundPerformKsmps(udata->csound) == 0) &&(udata->PERF_STATUS==1));
     csoundDestroy(udata->csound);
        }
  udata->PERF_STATUS = 0;
  return 1;
}
```

The application above might not appear all that interesting. In fact it's almost the exact same as the first example presented except that users can now stop Csound by hitting 'enter'. The real worth of threads can only be appreciated when you start to control your instrument in real time.

Channel I/O

The big advantage to using the API is that it allows a host to control your Csound instruments in real time. There are several mechanisms provided by the API that allow us to do this. The simplest mechanism makes use of a 'software bus'.

The term bus is usually used to describe a means of communication between hardware components. Buses are used in mixing consoles to route signals out of the mixing desk into external devices. Signals get sent through the sends and are taken back into the console through the returns. The same thing happens in a software bus, only instead of sending analog signals to different hardware devices we send data to and from different software.

Using one of the software bus opcodes in Csound we can provide an interface for communication with a host application. An example of one such opcode is chnget. The *chnget* opcode reads data that is being sent from a host Csound API application on a particular named channel, and assigns it to an output variable. In the following example instrument 1 retrieves any data the host may be sending on a channel named "pitch":

```
instr 1
```

```
kval chnget "pitch"
a1 oscil 10000, kval, 1
out a1
endin
```

One way in which data can be sent from a host application to an instance of Csound is through the use of the csoundGetChannelPtr() API function which is defined in csound.h as:

```
int csoundGetChannelPtr(CSOUND *, MYFLT **p, const char *name,
 int type);
```

CsoundGetChannelPtr() stores a pointer to the specified channel of the bus in p. The channel pointer p is of type MYFLT. The argument name is the name of the channel and the argument type is a bitwise OR of exactly one of the following values:

`CSOUND_CONTROL_CHANNEL` - control data (one MYFLT value)
`CSOUND_AUDIO_CHANNEL` - audio data (ksmps MYFLT values)
`CSOUND_STRING_CHANNEL` - string data (MYFLT values with enough space to store csoundGetStrVarMaxLen(CSOUND*) characters, including the NULL character at the end of the string)

and at least one of these:

`CSOUND_INPUT_CHANNEL` - when you need Csound to accept incoming values from a host
`CSOUND_OUTPUT_CHANNEL` - when you need Csound to send outgoing values to a host

If the call to csoundGetChannelPtr() is successful the function will return zero. If not, it will return a negative error code. We can now modify our previous code in order to send data from our application on a named software bus to an instance of Csound using csoundGetChannelPtr().

```
#include <stdio.h>
#include "csound.h"

//performance thread function prototype
uintptr_t csThread(void* clientData);

//userData structure declaration
typedef struct {
int result;
CSOUND* csound;
int PERF_STATUS;
}userData;



//-----------------------------------------------------------
// main function
//-----------------------------------------------------------
int main(int argc, char *argv[])
{
int userInput=200;
void* ThreadID;
userData* ud;
ud = (userData*)malloc(sizeof(userData));
MYFLT* pvalue;
csoundInitialize(&argc, &argv, 0);
ud->csound=csoundCreate(NULL);
ud->result=csoundCompile(ud->csound,argc,argv);
if(!ud->result)
{
ud->PERF_STATUS=1;
ThreadID = csoundCreateThread(csThread, (void*)ud);
}
else{
printf("csoundCompiled returned an error");
return 0;
}
printf("\nEnter a pitch in
Hz(0 to Exit) and type return\n");
while(userInput!=0)
{
if(csoundGetChannelPtr(ud->csound,
&pvalue, "pitch",
CSOUND_INPUT_CHANNEL
| CSOUND_CONTROL_CHANNEL)==0);
*pvalue =
(MYFLT)userInput;
scanf("%d",
&userInput);
}
ud->PERF_STATUS=0;
csoundDestroy(ud->csound);
free(ud);
return 1;
}
//-----------------------------------------------------------
//definition of our performance thread function
//-----------------------------------------------------------
uintptr_t csThread(void *data)
{
userData* udata =(userData*)data;
if(!udata->result)
{
while((csoundPerformKsmps(udata->csound)== 0)
 &&(udata->PERF_STATUS==1));
 csoundDestroy(udata->csound);
}
udata->PERF_STATUS = 0;
return 1;
}
```

SCORE EVENTS

Adding score events to the csound instance is easy to do. It requires that csound has its threading done, see the paragraph above on threading. To enter a score event into csound, one calls the following function:

```
void myInputMessageFunction( void* data, const char* message)
{
 userData* udata = (userData*) data;
 csoundInputMessage( udata->csound , message );
}
```

Now we can call that function to insert Score events into a running csound instance. The formatting of the message should be the same as one would normally have in the Score part of the .csd file. The example shows the format for the message. Note that if you're allowing csound to print its error messages, if you send a malformed message, it will warn you. Good for debugging. There's an example with the csound source code that allows you to type in a message, and then it will send it.

```
                       instrNum    start    duration   p4   p5  p6   ...   pN
const char* message = "i1          0        1         0.5  0.3 0.1"   ;
myInputMessageFunction( (void*) udata , message);
```

Callbacks

CONCLUSION

REFERENCES & LINKS

Michael Gogins 2006, "Csound and CsoundVST API Reference Manual", http://csound.sourceforge.net/refman.pdf

Rory Walsh 2006, "Developing standalone applications using the Csound Host API and wxWidgets", Csound Journal Volume 1 Issue 4 - Summer 2006, http://www.csounds.com/journal/2006summer/wxCsound.html

Rory Walsh 2010, "Developing Audio Software with the Csound Host API", The Audio Programming Book, DVD Chapter 35, The MIT Press

François Pinot 2011, "Real-time Coding Using the Python API: Score Events", Csound Journal Issue 14 - Winter 2011, http://www.csounds.com/journal/issue14/realtimeCsoundPython.html

55. USING PYTHON INSIDE CSOUND

coming in the next release ...

For now, have a look at Andrés Cabrera, Using Python inside Csound, An introduction to the Python opcodes, Csound Journal Issue 6, Spring 2007:
http://www.csounds.com/journal/issue6/pythonOpcodes.html

56. C. PYTHON IN CSOUNDQT

coming in the next release ...

For now, you may want to have a look at Andrés Cabrera's paper Python Scripting in QuteCsound at the Csound Conference in Hannover.

57. D. LUA IN CSOUND

coming in the next release ...

For now, have a look at Michael Gogins' paper Writing Csound Opcodes in Lua at the Csound Conference in Hannover (there is also a video from the workshop at www.youtube.com/user/csconf2011).

13 EXTENDING CSOUND

58. EXTENDING CSOUND

coming in the next release ...

OPCODE GUIDE

59. OPCODE GUIDE: OVERVIEW
60. OPCODE GUIDE: BASIC SIGNAL PROCESSING
61. OPCODE GUIDE: ADVANCED SIGNAL PROCESSING
62. OPCODE GUIDE: DATA
63. OPCODE GUIDE: REALTIME INTERACTION
64. OPCODE GUIDE: INSTRUMENT CONTROL
65. OPCODE GUIDE: MATH, PYTHON/ SYSTEM, PLUGINS

59. OPCODE GUIDE: OVERVIEW

If you run Csound from the command line with the option -z, you get a list of all opcodes. Currently (Csound 5.13), the total number of all opcodes is about 1500. There are already overviews of all of Csound's opcodes in the Opcodes Overview and the Opcode Quick Reference of the Canonical Csound Manual.

This chapter is another attempt to provide some orientation within Csound's wealth of opcodes. Unlike to the references mentioned above not all opcodes are listed, but the ones listed are commented briefly. Some opcodes appear more than once, which is done intentionally, for example, there are different contexts within which you might use the *ftgen* opcode and the layout here reflects this multipurpose nature of a number of opcodes. This guide may also provide insights into the opcodes listed that the other sources do not.

BASIC SIGNAL PROCESSING

- OSCILLATORS AND PHASORS
    - **Standard Oscillators**

      (oscils) poscil poscil3 oscili oscil3 more

    - **Dynamic Sprectrum Oscillators**

      buzz gbuzz mpulse vco vco2

    - **Phasors**

      phasor syncphasor

- RANDOM AND NOISE GENERATORS

  (seed) rand randi randh rnd31 random (randomi /randomh) pinkish more

- ENVELOPES
    - **Simple Standard Envelopes**

      linen linenr adsr madsr more

    - **Envelopes By Linear And Exponential Generators**

      linseg expseg transeg (linsegr expsegr transegr) more

    - **Envelopes By Function Tables**

- DELAYS
    - **Audio Delays**

      vdelay vdelayx vdelayw

      delayr delayw deltap deltapi deltap3 deltapx deltapxw deltapn

    - **Control Delays**

      delk vdel_k

- FILTERS

Compare Standard Filters and Specialized Filters overviews.

- **Low Pass Filters**

  tone tonex butlp clfilt

- **High Pass Filters**

  atone atonex buthp clfilt

- **Band Pass And Resonant Filters**

  reson resonx resony resonr resonz butbp

- **Band Reject Filters**

  areson butbr

- **Filters For Smoothing Control Signals**

  port portk

- REVERB

(pconvolve) freeverb reverbsc reverb nreverb babo

- SIGNAL MEASUREMENT, DYNAMIC PROCESSING, SAMPLE LEVEL OPERATIONS
  - **Amplitude Measurement And Following**

    rms balance follow follow2 peak max_k

  - **Pitch Estimation**

    ptrack pitch pitchamdf pvscent

  - **Tempo Estimation**

    tempest

  - **Dynamic Processing**

    compress dam clip

  - **Sample Level Operations**

    limit samphold vaget vaset

- SPATIALIZATION
  - **Panning**

    pan2 pan

  - **VBAP**

    vbaplsinit vbap4 vbap8 vbap16

  - **Ambisonics**

    bformenc1 bformdec1

- Binaural / HRTF

  hrtfstat hrtfmove hrtfmove2 hrtfer

ADVANCED SIGNAL PROCESSING

- MODULATION AND DISTORTION
  - Frequency Modulation

    foscil foscili

    crossfm crossfmi crosspm crosspmi crossfmpm crossfmpmi

  - Distortion And Wave Shaping

    distort distort1 powershape polynomial chebyshevpoly

  - Flanging, Phasing, Phase Shaping

    flanger harmon phaser1 phaser2 pdclip pdhalf pdhalfy

  - Doppler Shift

    doppler

- GRANULAR SYNTHESIS

  partikkel others sndwarp

- CONVOLUTION

  pconvolve ftconv dconv

- FFT AND SPECTRAL PROCESSING
  - Realtime Analysis And Resynthesis

    pvsanal pvstanal pvsynth pvsadsyn

  - Writing FFT Data To A File And Reading From It

    pvsfwrite pvanal pvsfread pvsdiskin

  - Writing FFT Data To A Buffer And Reading From It

    pvsbuffer pvsbufread pvsftw pvsftr

  - FFT Info

    pvsinfo pvsbin pvscent

  - Manipulating FFT Signals

    pvscale pvshift pvsbandp pvsbandr pvsmix pvscross pvsfilter pvsvoc pvsmorph pvsfreeze pvsmaska pvsblur pvstencil pvsarp pvsmooth

- PHYSICAL MODELS AND FM INSTRUMENTS
  - Waveguide Physical Modelling

    see here and here

- **FM Instrument Models**

  see here

DATA

- BUFFER / FUNCTION TABLES
  - **Creating Function Tables (Buffers)**

    ftgen GEN Routines

  - **Writing To Tables**

    tableiw / tablew tabw_i / tabw

  - **Reading From Tables**

    table / tablei / table3 tab_i / tab

  - **Saving Tables To Files**

    ftsave / ftsavek TableToSF

  - **Reading Tables From Files**

    ftload / ftloadk GEN23

- SIGNAL INPUT/OUTPUT, SAMPLE AND LOOP PLAYBACK, SOUNDFONTS
  - **Signal Input And Output**

    inch ; outch out outs ; monitor

  - **Sample Playback With Optional Looping**

    flooper2 sndloop

  - **Soundfonts And Fluid Opcodes**

    fluidEngine fluidSetInterpMethod fluidLoad fluidProgramSelect fluidNote fluidCCi fluidCCk fluidControl fluidOut fluidAllOut

- FILE INPUT AND OUTPUT
  - **Sound File Input**

    soundin diskin diskin2 mp3in (GEN01)

  - **Sound File Queries**

    filelen filesr filenchnls filepeak filebit

  - **Sound File Output**

    fout

  - **Non-Soundfile Input And Output**

    readk GEN23 dumpk fprints / fprintks ftsave / ftsavek ftload / ftloadk

- CONVERTERS OF DATA TYPES
    - i <- k

      i(k)

    - k <- a

      downsamp max_k

    - a <- k

      upsamp interp

- PRINTING AND STRINGS
    - Simple Printing

      print printk printk2 puts

    - Formatted Printing

      prints printf_i printks printf

    - String Variables

      sprintf sprintfk strset strget

    - String Manipulation And Conversion

      see here and here

REALTIME INTERACTION

- MIDI
    - Opcodes For Use In MIDI-Triggered Instruments

      massign pgmassign notnum cpsmidi veloc ampmidi midichn pchbend aftouch polyaft

    - Opcodes For Use In All Instruments

      ctrl7 (ctrl14/ctrl21) initc7 ctrlinit (initc14/initc21) midiin midiout

- OPEN SOUND CONTROL AND NETWORK
    - Open Sound Control

      OSCinit OSClisten OSCsend

    - Remote Instruments

      remoteport insremot insglobal midiremot midiglobal

    - Network Audio

      socksend sockrecv

- HUMAN INTERFACES
    - Widgets

FLTK overview here

  - **Keys**

    sensekey

  - **Mouse**

    xyin

  - **WII**

    wiiconnect wiidata wiirange wiisend

  - **P5 Glove**

    p5gconnect p5gdata

INSTRUMENT CONTROL

- SCORE PARAMETER ACCESS

  p(x) pindex pset passign pcount

- TIME AND TEMPO

  - Time Reading

    times/timek timeinsts/timeinstk date/dates setscorepos

  - Tempo Reading

    tempo miditempo tempoval

  - Duration Modifications

    ihold xtratim

  - Time Signal Generators

    metro mpulse

- CONDITIONS AND LOOPS

  changed trigger if loop_lt/loop_le/loop_gt/loop_ge

- PROGRAM FLOW

  init igoto kgoto timout reinit/rigoto/rireturn

- EVENT TRIGGERING

  event_i / event scoreline_i / scoreline schedkwhen seqtime /seqtime2 timedseq

- INSTRUMENT SUPERVISION

  - Instances And Allocation

    active maxalloc prealloc

  - Turning On And Off

    turnon turnoff/turnoff2 mute remove exitnow

  - Named Instruments

    nstrnum

- SIGNAL EXCHANGE AND MIXING
  - chn opcodes

    chn_k / chn_a / chn_S chnset chnget chnmix chnclear

  - zak?

MATHS

- MATHEMATICAL CALCULATIONS
  - Arithmetic Operations

    + - * / ^ %

    exp(x) log(x) log10(x) sqrt(x)

    abs(x) int(x) frac(x)

    round(x) ceil(x) floor(x)

  - Trigonometric Functions

    sin(x) cos(x) tan(x)

    sinh(x) cosh(x) tanh(x)

    sininv(x) cosinv(x) taninv(x) taninv2(x)

  - Logic Operators

    && ||

- CONVERTERS
  - MIDI To Frequency

    cpsmidi cpsmidinn more

  - Frequency To MIDI

    F2M F2MC (UDO's)

  - Cent Values To Frequency

    cent

  - Amplitude Converters

    ampdb ampdbfs dbamp dbfsamp

  - Scaling

    Scali Scalk Scala (UDO's)

PYTHON AND SYSTEM

- PYTHON OPCODES

  pyinit pyrun pyexec pycall pyeval pyassign

- SYSTEM OPCODES

  getcfg system/system_i

PLUGINS

- PLUGIN HOSTING
  - LADSPA

    dssiinit dssiactivate dssilist dssiaudio dssictls

  - VST

    vstinit vstaudio/vstaudiog vstmidiout vstparamset/vstparamget vstnote
    vstinfo vstbankload vstprogset vstedit

- EXPORTING CSOUND FILES TO PLUGINS

60. OPCODE GUIDE: BASIC SIGNAL PROCESSING

- ## OSCILLATORS AND PHASORS

  - ### Standard Oscillators

    oscils is a very **simple sine oscillator** which can be used for quick tests. It needs no function table, but provides just i-rate arguments.

    ftgen generates a function table, which is needed by any oscillator except oscils. The GEN Routines fill the function table with any desired waveform, either a sine wave or any other curve. Compare the function table chapter of this manual for more information.

    poscil can be recommended as **standard oscillator** because it is very precise also for long tables and low frequencies. It provides linear interpolation, any rate for the input arguments, and works also for non-power-of-two tables. poscil3 provides cubic interpolation, but has just k-rate input. **Other common oscillators** are oscili and oscil3. They are less precise than poscil/poscili, but you can skip the initialization which can be useful in certain situations. The oscil opcode does not provide any interpolation, so it should usually be avoided. **More** Csound oscillators can be found here.

  - ### Dynamic Spectrum Oscillators

    buzz and **gbuzz** generate a set of harmonically related sine resp. cosine partials.

    mpulse generates a set of impulses.

    vco and **vco2** implement band-limited, analog modeled oscillators with different standard waveforms.

  - ### Phasors

    phasor produces the typical moving phase values between 0 and 1. The more complex syncphasor lets you synchronize more than one phasor precisely.

- ## RANDOM AND NOISE GENERATORS

  seed sets the seed value for the majority of the Csound random generators (seed 0 generates each time another random output, while any other seed value generates the same random chain on each new run).

  rand is the usual opcodes for bipolar random values. If you give 1 as input argument (called "amp"), you will get values between -1 and +1. **randi** interpolates between values which are generated in a (variable) frequency. **randh** holds the value until the next one is generated. You can control the seed value by an input argument (a value greater than 1 seeds from current time), you can decide whether to use a 16bit or a 31bit random number, and you can add an offset.

  rnd31 can be used for alle rates of variables (i-rate variables are not supported by rand). It gives the user also control over the random distribution, but has no offset parameter.

  random is often very convenient to use, because you have a minimum and a maximum value as input argument, instead of a range like *rand* and *rnd31*. It can also be used for all rates, but you have no direct seed input, and the randomi/randomh variants always start from the lower border, instead anywhere between the borders.

pinkish produces pink noise at audio-rate (white noise is produced by *rand*).

There are much more random opcodes. Here is an overview. It is also possible to use some GEN Routines for generating random distributions. They can be found in the GEN Routines overview.

- # ENVELOPES

  - ## Simple Standard Envelopes

    linen applies a linear rise (fade in) and decay (fade out) to a signal. It is very easy to use, as you put the raw audio signal in and get the enveloped signal out.

    linenr does the same for any note which's duration is not fixed at the beginning, like MIDI notes or any real time events. linenr begins to fade out exactly when the instrument is turned off, adding an extra time after this turnoff.

    adsr calculates the classical attack-decay-sustain-release envelope. The result is to be multiplied with the audio signal to get the enveloped signal.

    madsr does the same for a realtime note (like explained above for linenr).

    Other standard envelope generators can be found in the Envelope Generators overview of the Canonical Csound Manual.

  - ## Envelopes By Linear And Exponential Generators

    linseg creates one or more segments of lines between specified points.

    expseg does the same with exponential segments. Note that zero values are illegal.

    transeg is very flexible to use, because you can specify the shape of the curve for each segment (continuous transitions from convex to linear to concave).

    All these opcodes have a -r variant (linsegr, expsegr, transegr) for MIDI or other live events.

    More opcodes can be found in this overview.

  - ## Envelopes By Function Tables

    Any curve, or parts of it, of any function table, can be used as envelope. Just create a function table by ftgen resp. by a GEN Routine. Then read the function table, or a part of it, by an oscillator, and multiply the result with the audio signal you want to envelope.

- # DELAYS

  - ## Audio Delays

    The **vdelay familiy** of opcodes is easy to use and implement all necessary features to work with delays:

    vdelay implements a variable delay at audio rate with linear interpolation.

    vdelay3 offers cubic interpolation.

    vdelayx has an even higher quality interpolation (and is by this reason slower). vdelayxs lets you input and output two channels, and vdelayxq four.

    vdelayw changes the position of the write tap in the delay line instead of the read tap. vdelayws is for stereo, and vdelaywq for quadro.

    The **delayr/delayw** opcodes establishes a delay line in a more complicated way. The advantage is that you can have as many taps in one delay line as you need.

    delayr establishes a delay line and reads from it.

    delayw writes an audio signal to the delay line.

    deltap, **deltapi**, **deltap3**, **deltapx** and **deltapxw** are working similar to the relevant opcodes of the vdelay family (see above).

    deltapn offers a tap delay measured in samples, not seconds.

  - ### Control Delays

    delk and **vdel_k** let you delay any k-signal by some time interval (usable for instance as a kind of *wait* mode).

- ## FILTERS

  Csound has an extremely rich collection of filters and they are good available on the Csound Manual pages for Standard Filters and Specialized Filters. So here some most frequently used filters are mentioned, and some tips are given. Note that filters usually change the signal level, so you will need the balance opcode.

  - ### Low Pass Filters

    tone is a first order recursive low pass filter. tonex implements a series of tone filters.

    butlp is a seond order low pass Butterworth filter.

    clfilt lets you choose between different types and poles numbers.

  - ### High Pass Filters

    atone is a first order recursive high pass filter. atonex implements a series of atone filters.

    buthp is a second order high pass Butterworth filter.

    clfilt lets you choose between different types and poles numbers.

  - ### Band Pass And Resonant Filters

    reson is a second order resonant filter. resonx implements a series of reson filters, while resony emulates a bank of second order bandpass filters in parallel. resonr and resonz are variants of reson with variable frequency response.

    butbp is a second order band-pass Butterworth filter.

  - ### Band Reject Filters

    areson is the complement of the reson filter.

    butbr is a band-reject butterworth filter.

  - ### Filters For Smoothing Control Signals

    port and **portk** are very frequently used to smooth control signals which are received by MIDI or widgets.

- ## REVERB

  Note that you can work easily in Csound with convolution reverbs based on impulse response files, for instance with pconvolve.

  freeverb is the implementation of Jezar's well-known free (stereo) reverb.

reverbsc is a stereo FDN reverb, based on work of Sean Costello.

reverb and **nreverb** are the traditional Csound reverb units.

babo is a physical model reverberator ("ball within the box").

- # SIGNAL MEASUREMENT, DYNAMIC PROCESSING, SAMPLE LEVEL OPERATIONS

  - ## Amplitude Measurement And Following

    rms determines the root-mean-square amplitude of an audio signal.

    balance adjusts the amplitudes of an audio signal according to the rms amplitudes of another audio signal.

    follow / **follow2** are envelope followers which report the average amplitude in a certain time span (follow) or according to an attack/decay rate (follow2).

    peak reports the highest absolute amplitude value received.

    max_k outputs the local maximum or minimum value of an incoming audio signal, checked in a certain time interval.

  - ## Pitch Estimation

    ptrack, **pitch** and **pitchamdf** track the pitch of an incoming audio signal, using different methods.

    pvscent calculates the spectral centroid for FFT streaming signals (see below under "FFT And Spectral Processing")

  - ## Tempo Estimation

    tempest estimates the tempo of beat patterns in a control signal.

  - ## Dynamic Processing

    compress compresses, limits, expands, ducks or gates an audio signal.

    dam is a dynamic compressor/expander.

    clip clips an a-rate signal to a predefined limit, in a “soft” manner.

  - ## Sample Level Operations

    limit sets the lower and upper limits of an incoming value (all rates).

    samphold performs a sample-and-hold operation on its a- or k-input.

    vaget / **vaset** allow getting and setting certain samples of an audio vector at k-rate.

- # SPATIALIZATION

  - ## Panning

    pan2 distributes a mono audio signal across two channels, with different envelope options.

    pan distributes a mono audio signal amongst four channels.

  - ## VBAP

    vbaplsinit configures VBAP output according to loudspeaker parameters for a 2- or 3-dimensional space.

vbap4 / **vbap8** / **vbap16** distributes an audio signal among up to 16 channels, with k-rate control over azimut, elevation and spread.

- ### Ambisonics

  bformenc1 encodes an audio signal to the Ambisonics B format.

  bformdec1 decodes Ambisonics B format signals to loudspeaker signals in different possible configurations.

- ### Binaural / HRTF

  hrtfstat, **hrtfmove** and **hrtfmove2** are opcodes for creating 3d binaural audio for headphones. hrtfer is an older implementation, using an external file.

61. OPCODE GUIDE: ADVANCED SIGNAL PROCESSING

- ## MODULATION AND DISTORTION

  - ### Frequency Modulation

    foscil and **foscili** implement composite units for FM in the Chowning setup.

    crossfm, **crossfmi**, **crosspm**, **crosspmi**, **crossfmpm** and **crossfmpmi** are different units for frequency and/or phase modulation.

  - ### Distortion And Wave Shaping

    distort and **distort1** perform waveshaping by a function table (distort) or by modified hyperbolic tangent distortion (distort1).

    powershape waveshapes a signal by raising it to a variable exponent.

    polynomial efficiently evaluates a polynomial of arbitrary order.

    chebyshevpoly efficiently evaluates the sum of Chebyshev polynomials of arbitrary order.

    GEN03, GEN13, GEN14 and GEN15 are also used for Waveshaping.

  - ### Flanging, Phasing, Phase Shaping

    flanger implements a user controllable flanger.

    harmon analyzes an audio input and generates harmonizing voices in synchrony.

    phaser1 and **phaser2** implement first- or second-order allpass filters arranged in a series.

    pdclip, **pdhalf** and **pdhalfy** are useful for phase distortion synthesis.

  - ### Doppler Shift

    doppler lets you calculate the doppler shift depending on the position of the sound source and the microphone.

- ## GRANULAR SYNTHESIS

  partikkel is the most flexible opcode for granular synthesis. You should be able to do everything you like in this field. The only drawback is the large number of input arguments, so you may want to use other opcodes for certain purposes.

  You can find a list of other relevant opcodes here.

  sndwarp focusses granular synthesis on time stretching and/or pitch modifications. Compare waveset and the pvs-opcodes pvsfread, pvsdiskin, pvscale, pvshift for other implementations of time and/or pitch modifications.

- ## CONVOLUTION

  pconvolve performs convolution based on a uniformly partitioned overlap-save algorithm.

  ftconv is similar to pconvolve, but you can also use parts of the impulse response file, instead of reading the whole file.

dconv performs direct convolution.

- # FFT AND SPECTRAL PROCESSING

  - ## Realtime Analysis And Resynthesis

    pvsanal performs a Fast Fourier Transformation of an audio stream (a-signal) and stores the result in an f-variable.

    pvstanal creates an f-signal directly from a sound file which is stored in a function table (usually via GEN01).

    pvsynth performs an Inverse FFT (takes a f-signal and returns an audio-signal).

    pvsadsyn is similar to pvsynth, but resynthesizes with a bank of oscillators, instead of direct IFFT.

  - ## Writing FFT Data To A File And Reading From It

    pvsfwrite writes an f-signal (= the FFT data) from inside Csound to a file. This file has the PVOCEX format and its name ends on .pvx.

    pvanal does actually the same as Csound Utility (a seperate program which can be called in QuteCsound or via the Terminal). In this case, the input is an audio file.

    pvsfread reads the FFT data from an extisting .pvx file. This file can be generated by the Csound Utility pvanal. Reading the file is done by a time pointer.

    pvsdiskin is similar to pvsfread, but reading is done by a speed argument.

  - ## Writing FFT Data To A Buffer And Reading From It

    pvsbuffer writes a f-signal to a circular buffer (and creates it).

    pvsbufread reads a f-signal from a buffer which was created by pvsbuffer.

    pvsftw writes amplitude and/or frequency data from a f-signal to a function table.

    pvsftr transforms amplitude and/or frequency data from a function table to a f-signal.

  - ## FFT Info

    pvsinfo gets info either from a realtime f-signal or from a .pvx file.

    pvsbin gets the amplitude and frequency values from a single bin of a f-signal.

    pvscent calculates the spectral centroid of a signal.

  - ## Manipulating FFT Signals

    pvscale transposes the frequency components of a f-stream by simple multiplication.

    pvshift changes the frequency components of a f-stream by adding a shift value, starting at a certain bin.

    pvsbandp and **pvsbandr** applies a band pass and band reject filter to the frequency components of a f-signal.

    pvsmix, **pvscross**, **pvsfilter**, **pvsvoc** and **pvsmorph** perform different methods of cross synthesis between two f-signals.

    pvsfreeze freezes the amplitude and/or frequency of a f-signal according to a k-rate trigger.

pvsmaska, **pvsblur**, **pvstencil**, **pvsarp**, **pvsmooth** perform other manipulations on a stream of FFT data.

- # PHYSICAL MODELS AND FM INSTRUMENTS

  - ### Waveguide Physical Modelling

    see here and here

  - ### FM Instrument Models

    see here

62. OPCODE GUIDE: DATA

- ## BUFFER / FUNCTION TABLES

  See the chapter about function tables for more detailled information.

  - ### Creating Function Tables (Buffers)

    ftgen generates any function table. The GEN Routines are used to fill a function table with different kind of data, like soundfiles, envelopes, window functions and much more.

  - ### Writing To Tables

    tableiw /
    tablew: Write values to a function table at i-rate (tableiw), k-rate and a-rate (tablew). These opcodes provide many options and are safe because of boundary check, but you may have problems with non-power-of-two tables.

    tabw_i / **tabw**: Write values to a function table at i-rate (tabw_i), k-rate or a-rate (tabw). Offer less options than the tableiw/tablew opcodes, but work also for non-power-of-two tables. They do not provide a boundary check, which makes them fast but also give the user the resposability not writing any value off the table boundaries.

  - ### Reading From Tables

    table / **tablei** / **table3**: Read values from a function table at any rate, either by direct indexing (table), or by linear (tablei) or cubic (table3) interpolation. These opcodes provide many options and are safe because of boundary check, but you may have problems with non-power-of-two tables.

    tab_i / **tab**: Read values from a function table at i-rate (tab_i), k-rate or a-rate (tab). Offer no interpolation and less options than the table opcodes, but they work also for non-power-of-two tables. They do not provide a boundary check, which makes them fast but also give the user the resposability not reading any value off the table boundaries.

  - ### Saving Tables To Files

    ftsave / **ftsavek**: Save a function table as a file, at i-time (ftsave) or k-time (ftsavek). This can be a text file or a binary file, but not a soundfile. If you want to save a soundfile, use the User Defined Opcode TableToSF.

  - ### Reading Tables From Files

    ftload / **ftloadk**: Load a function table which has been written by ftsave/ftsavek.

    GEN23 transfers a text file into a function table.

- ## SIGNAL INPUT/OUTPUT, SAMPLE AND LOOP PLAYBACK, SOUNDFONTS

  - ### Signal Input And Output

    inch read the audio input from any channel of your audio device. Make sure you have the nchnls value in the orchestra header set properly.

    outch writes any audio signal(s) to any output channel(s). If Csound is in realtime mode (by the flag '-o dac' or by the 'Render in Realtime' mode of a frontend like

QuteCsound), the output channels are the channels of your output device. If Csound is in 'Render to file' mode (by the flag '-o mysoundfile.wav' or the the frontend's choice), the output channels are the channels of the soundfile which is being written. Make sure you have the nchnls value in the orchestra header set properly to get the number of channels you wish to have.

out and **outs** are frequently used for mono and stereo output. They always write to channel 1 (out) resp. 1 and 2 (outs).

monitor can be used (in an instrument with the highest number) to get the sum of all audio on the different output channels.

Sample Playback With Optional Looping

flooper2 is a function-table-based crossfading looper.

sndloop records input audio and plays it back in a loop with user-defined duration and crossfade time.

Note that there are also User Defined Opcodes for sample playback of buffers / function tables.

Soundfonts And Fluid Opcodes

fluidEngine instantiates a FluidSynth engine.

fluidSetInterpMethod sets an interpolation method for a channel in a FluidSynth engine.

fluidLoad loads SoundFonts.

fluidProgramSelect assigns presets from a SoundFont to a FluidSynth engine's MIDI channel.

fluidNote plays a note on a FluidSynth engine's MIDI channel.

fluidCCi sends a controller message at i-time to a FluidSynth engine's MIDI channel.

fluidCCk sends a controller message at k-rate to a FluidSynth engine's MIDI channel.

fluidControl plays and controls loaded Soundfonts (using 'raw' MIDI messages).

fluidOut receives audio from a single FluidSynth engine.

fluidAllOut receives audio from all FluidSynth engines.

FILE INPUT AND OUTPUT

Sound File Input

soundin reads from a soundfile (up to 24 channels). Make sure that the sr value in the orchestra header matches the sample rate of your soundfile, or you will get higher or lower pitched sound.

diskin is like soundin, but can also alter the speed of reading (resulting in higher or lower pitches) and you have an option to loop the file.

diskin2 is like diskin, but automatically converts the sample rate of the soundfile if it does not match the sample rate of the orchestra, and it offers different interpolation methods for reading the soundfile at altered speed.

GEN01 reads soundfile into a function table (buffer).

mp3in lets you play mp3 sound files.

Sound File Queries

filelen returns the length of a soundfile in seconds.

filesr returns the sample rate of a soundfile.

filenchnls returns the number of channels of a soundfile.

filepeak returns the peak absolute value of a soundfile, either of one specified channel, or from all channels. Make sure you have set 0dbfs to 1; otherwise you will get values relative to Csound's default 0dbfs value of 32768.

filebit returns the bit depth of a soundfile.

- Sound File Output

  Keep in mind that Csound always writes output to a file if you have set the '-o' flag to the name of a soundfile (or if you choose 'render to file' in a frontend like QuteCound).

  fout writes any audio signal(s) to a file, regardless Csound is in realtime or render-to-file mode. So you can record your live performance with this opcode.

- Non-Soundfile Input And Output

  readk can read data from external files (for instance a text file) and transform them to k-rate values.

  GEN23 transfers a text file into a function table.

  dumpk writes k-rate signals to a text file.

  fprints / **fprintks** write any formatted string to a file. If you call this opcode several times during one performance, the strings are appended. If you write to an already existing file, the file will be overwritten.

  ftsave / **ftsavek**: Save a function table as a binary or text file, in a specific format.

  ftload / **ftloadk**: Load a function table which has been written by ftsave/ftsavek.

CONVERTERS OF DATA TYPES

- i <- k

  i(k) returns the value of a k-variable at init-time. This can be useful to get the value of GUI controllers, or when using the reinit feature.

- k <- a

  downsamp converts an a-rate signal to a k-rate signal, with optional averaging.

  max_k returns the maximum of an a-rate signal in a certain time span, with different options of calculation

- a <- k

  upsamp converts a k-rate signal to an a-rate signal by simple repetitions. It is the same as the statement asig=ksig.

  interp converts a k-rate signal to an a-rate signal by interpolation.

PRINTING AND STRINGS

- Simple Printing

  print is a simple opcode for printing i-variables. Note that the printed numbers are rounded to 3 decimal places.

printk is its counterpart for k-variables. The *itime* argument specifies the time in seconds between printings (*itime=0* means one printout in each k-cycle which is usually some thousand printings per second).

printk2 prints a k-variable whenever it has changed.

puts prints S-variables. The *ktrig* argument lets you print either at i-time or at k-time.

- ### Formatted Printing

  prints lets you print a format string at i-time. The format is similar to the C-style syntax (verweis). There is no %s format, therefore no string variables can be printed.

  printf_i is very similar to prints. It also works at init-time. The advantage in comparision to prints is the ability of printing string variables. On the other hand, you need a trigger and at least one input argument.

  printks is like prints, but takes k-variables, and like at printk you must specify a time between printing.

  printf is like printf_i, but works at k-rate.

- ### String Variables

  sprintf works like printf_i, but stores the output in a string variable, instead of printing it out.

  sprintfk is the same for k-rate arguments.

  strset links any string with a numeric value.

  strget transforms a strset number back to a string.

- ### String Manipulation And Conversion

  There are many opcodes for analysing, manipulating and conversing strings. There is a good overview in the Canonical Csound Manual on this and that page.

63. OPCODE GUIDE: REALTIME INTERACTION

- ## MIDI

  - ### Opcodes For Use In MIDI-Triggered Instruments

    massign assigns certain midi channels to instrument numbers. See the Triggering Instrument Instances chapter for more information.

    pgmassign assigns certain program changes to instrument numbers.

    notnum gets the midi number of the key which has been pressed and activated this instrument instance.

    cpsmidi converts this note number to the frequency in cycles per second (Hertz).

    veloc and **ampmidi** get the velocity of the key which has been pressed and activated this instrument instance.

    midichn returns the midi channel number from which the note was activated.

    pchbend gets the pitch bend information.

    aftouch and **polyaft** get the aftertouch information.

  - ### Opcodes For Use In All Instruments

    ctrl7 gets the values of a usual (7bit) controller and scales it. ctrl14 and ctrl21 can be used for high definition controllers.

    initc7 or **ctrlinit** set the initial value of 7bit controllers. Use initc14 and initc21 for high definition devices.

    midiin gives access to all incoming midi events.

    midiout writes any event to the midi out port.

- ## OPEN SOUND CONTROL AND NETWORK

  - ### Open Sound Control

    OSCinit initializes a port for later use of the OSClisten opcode.

    OSClisten receives messages of the port which was initialized by OSCinit.

    OSCsend sends messages to a port.

  - ### Remote Instruments

    remoteport defines the port for use with the remote system.

    insremot will send note events from a source machine to one destination.

    insglobal will send note events from a source machine to many destinations.

    midiremot will send midi events from a source machine to one destination.

    midiglobal will broadcast the midi events to all the machines involved in the remote concert.

  - ### Network Audio

socksend sends audio data to other processes using the low-level UDP or TCP protocols.

sockrecv receives audio data from other processes using the low-level UDP or TCP protocols.

- ## HUMAN INTERFACES

    - ### Widgets

      The FLTK Widgets are integrated in Csound. Information and examples can be found here.

      QuteCsound implements a more modern and easy-to-use system for widgets. The communication between the widgets and Csound is done via invalue (or chnget) and outvalue (or chnset).

    - ### Keys

      sensekey gets the input of your computer keys.

    - ### Mouse

      xyin can get the mouse position if your front-end does not provide this sensing otherwise.

    - ### WII

      wiiconnect reads data from a number of external Nintendo Wiimote controllers.

      wiidata reads data fields from a number of external Nintendo Wiimote controllers.

      wiirange sets scaling and range limits for certain Wiimote fields.

      wiisend sends data to one of a number of external Wii controllers.

    - ### P5 Glove

      p5gconnect reads data from an external P5 Glove controller.

      p5gdata reads data fields from an external P5 Glove controller.

64. OPCODE GUIDE: INSTRUMENT CONTROL

- ## SCORE PARAMETER ACCESS

p(x) gets the value of a specified p-field. (So, 'p(5)' and 'p5' both return the value of the fifth parameter in a certain score line, but in the former case you can insert a variable to specify the p-field.

pindex does actually the same, but as an opcode instead of an expression.

pset sets p-field values in case there is no value from a scoreline.

passign assigns a range of p-fields to i-variables.

pcount returns the number of p-fields belonging to a note event.

- ## TIME AND TEMPO

  - ### Time Reading

times / **timek** return the time in seconds (times) or in control cycles (timek) since the start of the current Csound performance.

timeinsts / **timeinstk** return the time in seconds (timeinsts) or in control cycles (timeinstk) since the start of the instrument in which they are defined.

date / **dates** return the number of seconds since 1 January 1970, using the operating system's clock; either as a number (date) or as a string (dates).

setscorepos sets the playback position of the current score performance to a given position.

  - ### Tempo Reading

tempo allows the performance speed of Csound scored events to be controlled from within an orchestra.

miditempo returns the current tempo at k-rate, of either the midi file (if available) or the score.

tempoval reads the current value of the tempo.

  - ### Duration Modifications

ihold causes a finite-duration note to become a 'held' note.

xtratim extend the duration of the current instrument instance.

  - ### Time Signal Generators

metro outputs a metronome-like control signal in a variable frequency.

mpulse generates an impulse for one sample (as audio-signal), followed by a variable time span.

- ## CONDITIONS AND LOOPS

changed reports whether a k-variable (or at least one of some k-variables) has changed.

trigger informs whether a k-rate signal crosses a certain threshold.

if branches conditionally at initialization or during performance time.

loop_lt, **loop_le**, **loop_gt** and **loop_ge** perform loops either at i- or k-time.

- # PROGRAM FLOW

init initializes a k- or a-variable (assigns a value to a k- or a-variable which is valid at i-time).

igoto jumps to a label at i-time.

kgoto jumps to a label at k-time.

timout jumps to a label for a given time. Can be used in conjunction with reinit to perform time loops (see the chapter about Control Structures for more information).

reinit / **rigoto** / **rireturn** forces a certain section of code to be reinitialized (= i-rate variables are renewed).

- # EVENT TRIGGERING

event_i / **event**: Generate an instrument event at i-time (event_i) or at k-time (event). Easy to use, but you cannot send a string to the subinstrument.

scoreline_i / **scoreline**: Generate an instrument at i-time (scoreline_i) or at k-time (scoreline). Like event_i/event, but you can send to more than one instrument but unlike event_i/event you can send strings. On the other hand, you must usually preformat your scoreline-string using sprintf.

schedkwhen triggers an instrument event at k-time if a certain condition is given.

seqtime / **seqtime2** can be used to generate a trigger signal according to time values in a function table.

timedseq is an event-sequencer in which time can be controlled by a time-pointer. Sequence data are stored into a table.

- # INSTRUMENT SUPERVISION

  - ## Instances And Allocation

    active returns the number of active instances of an instrument.

    maxalloc limits the number of allocations (instances) of an instrument.

    prealloc creates space for instruments but does not run them.

  - ## Turning On And Off

    turnon activates an instrument for an indefinite time.

    turnoff / **turnoff2** enables an instrument to turn itself, or another instrument, off.

    mute mutes/unmutes new instances of a given instrument.

    remove removes the definition of an instrument as long as it is not in use.

    exitnow exits csound as fast as possible, with no cleaning up.

  - ## Named Instruments

    nstrnum returns the number of a named instrument.

- # SIGNAL EXCHANGE AND MIXING

- **chn opcodes**

  chn_k, **chn_a**, and **chn_S** declare a control, audio, or string channel. Note that this can be done implicitely in most cases by chnset/chnget.

  chnset writes a value (i, k, S or a) to a software channel (which is identified by a string as its name).

  chnget gets the value of a named software channel.

  chnmix writes audio data to an named audio channel, mixing to the previous output.

  chnclear clears an audio channel of the named software bus to zero.

- **zak**

65. OPCODE GUIDE: MATH, PYTHON/ SYSTEM, PLUGINS

MATH

- ### MATHEMATICAL CALCULATIONS

  - #### Arithmetic Operations

    +, **-**, *****, **/**, **^**, **%** are the usual signs for addition, subtraction, multiplication, division, raising to a power and modulo. The precedence is like in common mathematics (a "*" binds stronger than "+" etc.), but you can change this behaviour with parentheses: 2^(1/12) returns 2 raised by 1/12 (= the 12st root of 2), while 2^1/12 returns 2 raised by 1, and the result divided by 12.

    exp(x), **log(x)**, **log10(x)** and **sqrt(x)** return e raised to the xth power, the natural log of x, the base 10 log of x, and the square root of x.

    abs(x) returns the absolute value of a number.

    int(x) and **frac(x)** return the integer respective the fractional part of a number.

    round(x), **ceil(x)**, **floor(x)** round a number to the nearest, the next higher or the next lower integer.

  - #### Trigonometric Functions

    sin(x), **cos(x)**, **tan(x)** perform a sine, cosine or tangent function.

    sinh(x), **cosh(x)**, **tanh(x)** perform a hyperbolic sine, cosine or tangent function.

    sininv(x), **cosinv(x)**, **taninv(x)** and **taninv2(x)** perform the arcsine, arccosine and arctangent functions.

  - #### Logic Operators

    && and **||** are the symbols for a logical "and" respective "or". Note that you can use here parentheses for defining the precedence, too, for instance: if (ival1 < 10 && ival2 > 5) || (ival1 > 20 && ival2 < 0) then ...

- ### CONVERTERS

  - #### MIDI To Frequency

    cpsmidi converts a MIDI note number from a triggered instrument to the frequency in Hertz.

    cpsmidinn does the same for any input values (i- or k-rate).

    Other opcodes convert to Csonund's pitch- or octave-class system. They can be found here.

  - #### Frequency To MIDI

    Csound has no own opcode for the conversion of a frequency to a midi note number, because this is a rather simple calculation. You can find a User Defined Opcode for rounding to the next possible midi note number or for the exact translation to a midi note number and a cent value as fractional part.

  - #### Cent Values To Frequency

cent converts a cent value to a multiplier. For instance, *cent(1200)* returns 2, *cent(100)* returns 1.059403. If you multiply this with the frequency you reference to, you get frequency of the note which corresponds to the cent interval.

- Amplitude Converters

  ampdb returns the amplitude equivalent of the dB value. *ampdb(0)* returns 1, *ampdb(-6)* returns 0.501187, and so on.

  ampdbfs returns the amplitude equivalent of the dB value, according to what has been set as 0dbfs (1 is recommended, the default is 15bit = 32768). So ampdbfs(-6) returns 0.501187 for 0dbfs=1, but 16422.904297 for 0dbfs=32768.

  dbamp returns the decibel equivalent of the amplitude value, where an amplitude of 1 is the maximum. So dbamp(1) -> 0 and dbamp(0.5) -> -6.020600.

  dbfsamp returns the decibel equivalent of the amplitude value set by the 0dbfs statement. So dbfsamp(10) is 20.000002 for 0dbfs=0 but -70.308998 for 0dbfs=32768.

- Scaling

  Scaling of signals from an input range to an output range, like the "scale" object in Max/MSP, is not implemented in Csound, because it is a rather simple calculation. It is available as User Defined Opcode: Scali (i-rate), Scalk (k-rate) or Scala (a-rate).

PYTHON AND SYSTEM

PYTHON OPCODES

pyinit initializes the Python interpreter.

pyrun runs a Python statement or block of statements.

pyexec executes a script from a file at k-time, i-time or if a trigger has been received.

pycall invokes the specified Python callable at k-time or i-time.

pyeval evaluates a generic Python expression and stores the result in a Csound k- or i-variable, with optional trigger.

pyassign assigns the value of the given Csound variable to a Python variable possibly destroying its previous content.

SYSTEM OPCODES

getcfg returns various Csound configuration settings as a string at init time.

system / **system_i** call an external program via the system call.

PLUGINS

PLUGIN HOSTING

LADSPA

dssiinit loads a plugin.

dssiactivate activates or deactivates a plugin if it has this facility.

dssilist lists all available plugins found in the LADSPA_PATH and DSSI_PATH global variables.

dssiaudio processes audio using a plugin.

dssictls sends control information to a plugin's control port.

- VST

  vstinit loads a plugin.

  vstaudio / **vstaudiog** return a plugin's output.

  vstmidiout sends midi data to a plugin.

  vstparamset / **vstparamget** sends and receives automation data to and from the plugin.

  vstnote sends a midi note with a definite duration.

  vstinfo outputs the parameter and program names for a plugin.

  vstbankload loads an .fxb bank.

  vstprogset sets the program in a .fxb bank.

  vstedit opens the GUI editor for the plugin, when available.

APPENDIX

66. GLOSSARY

control cycle, **control period** or **k-loop** is a pass during the performance of an instrument, in which all k- and a-variables are renewed. The time for one control cycle is measured in samples and determined by the ksmps constant in the orchestra header. If your sample rate is 44100 and your ksmps value is 10, the time for one control cycle is 1/4410 = 0.000227 seconds. See the chapter about Initialization And Performance Pass for more information.

control rate or **k-rate** (kr) is the number of control cycles per second. It can be calculated as the relationship of the sample rate sr and the number of samples in one control period ksmps. If your sample rate is 44100 and your ksmps value is 10, your control rate is 4410, so you have 4410 control cycles per second.

dummy f-statement see **f-statement**

f-statement or **function table statement** is a score line which starts with a "f" and generates a function table. See the chapter about function tables for more information. A **dummy f-statement** is a statement like "f 0 3600" which looks like a function table statement, but instead of generating any table, it serves just for running Csound for a certain time (here 3600 seconds = 1 hour).

FFT Fast Fourier Transform is a system whereby audio data is stored or represented in the frequency domain as opposed to the time domain as amplitude values as is more typical. Working with FFT data facilitates transformations and manipulations that are not possible, or are at least more difficult, with audio data stored in other formats.

GEN rountine a GEN (generation) routine is a mechanism within Csound used to create function tables of data that will be held in RAM for all or part of the performance. A GEN routine could be a waveform, a stored sound sample, a list of explicitly defined number such as tunings for a special musical scale or an amplitude envelope. In the past function tables could only be created only in the Csound score but now they can also be created (and deleted and over-written) within the orchestra.

GUI Graphical User Interface refers to a system of on-screen sliders, buttons etc. used to interact with Csound, normally in realtime.

i-time or **init-time** or **i-rate** signify the time in which all the variables starting with an "i" get their values. These values are just given once for an instrument call. See the chapter about Initialization And Performance Pass for more information.

k-loop see **control cycle**

k-time is the time during the performance of an instrument, after the initialization. Variables starting with a "k" can alter their values in each ->control cycle. See the chapter about Initialization And Performance Pass for more information.

k-rate see **control rate**

opcode the code word of a basic building block with which Csound code is written. As well as the opcode code word an opcode will commonly provide output arguments (variables), listed to the left of the opcode, and input arguments (variables). listed to the right of the opcode. An opcode is equivalent to a 'ugen' (unit generator) in other languages.

orchestra as in the Csound orchestra, is the section of Csound code where traditionally the instruments are written. In the past the 'orchestra' was one of two text files along with the 'score' that were needed to run Csound. Most people nowadays combine these two sections, along with other optional sections in a .csd (unified) Csound file. The orchestra will also normally contain header statements which will define global aspects of the Csound performance such as sampling rate.

p-field a 'p' (parameter) field normally refers to a value contained within the list of values after an event item with the Csound score.

performance pass see **control cycle**

score as in the Csound score, is the section of Csound code where note events are written that will instruct instruments within the Csound orchestra to play. The score can also contain function tables. In the past the 'score' was one of two text files along with the 'orchestra' that were needed to run Csound. Most people nowadays combine these two sections, along with other optional sections in a .csd (unified) Csound file.

time stretching can be done in various ways in Csound. See sndwarp, waveset, pvstanal and the Granular Synthesis opcodes. In the frequency domain, you can use the pvs-opcodes pvsfread, pvsdiskin, pvscale, pvshift.

widget normally refers to some sort of standard GUI element such as a slider or a button. GUI widgets normally permit some user modifications such as size, positioning colours etc. A variety options are available for the creation of widgets usable by Csound, from it own built-in FLTK widgets to those provided by front-ends such as CsoundQT, Cabbage and Blue.

67. LINKS

DOWNLOADS

Csound: http://sourceforge.net/projects/csound/files/

Csound's User Defined Opcodes: http://www.csounds.com/udo/

CsoundQt: http://sourceforge.net/projects/qutecsound/files/

WinXound:http://winxound.codeplex.com

Blue: http://sourceforge.net/projects/bluemusic/files/

Cabbage: http://code.google.com/p/cabbage

COMMUNITY

Csound's info page on sourceforge is a good collection of links and basic infos.

csounds.com is the main page for the Csound community, including news, online tutorial, forums and many links.

The Csound Journal is a main source for different aspects of working with Csound.

MAILING LISTS AND BUG TRACKER

To subscribe to the **Csound User** Discussion List, send a message with "subscribe csound <your name>" in the message body to sympa@lists.bath.ac.uk. To post, send messages to csound@lists.bath.ac.uk. You can search in the list archive at nabble.com.

To subscribe to the **CsoundQt User** Discussion List, go to https://lists.sourceforge.net/lists/listinfo/qutecsound-users. You can browse the list archive here.

Csound Developer Discussions: https://lists.sourceforge.net/lists/listinfo/csound-devel

Blue: http://sourceforge.net/mail/?group_id=74382

Please report any **bug** you experienced in **Csound** at http://sourceforge.net/tracker/?group_id=81968&atid=564599, and a **CsoundQt** related bug at http://sourceforge.net/tracker/?func=browse&group_id=227265&atid=1070588. Every bug report is an important contribution.

TUTORIALS

A Beginning Tutorial is a short introduction from Barry Vercoe, the "father of Csound".

An Instrument Design TOOTorial by Richard Boulanger (1991) is another classical introduction, still very worth to read.

Introduction to Sound Design in Csound also by Richard Boulanger, is the first chapter of the famous Csound Book (2000).

Virtual Sound by Alessandro Cipriani and Maurizio Giri (2000)

A Csound Tutorial by Michael Gogins (2009), one of the main Csound Developers.

VIDEO TUTORIALS

A playlist as overview by Alex Hofmann:

http://www.youtube.com/view_play_list?p=3EE3219702D17FD3

CsoundQt (QuteCsound)

QuteCsound: Where to start?
http://www.youtube.com/watch?v=0XcQ3ReqJTM

First instrument:
http://www.youtube.com/watch?v=P5OOyFyNaCA

Using MIDI:
http://www.youtube.com/watch?v=8zszIN_N3bQ

About configuration:
http://www.youtube.com/watch?v=KgYea5s8tFs

Presets tutorial:
http://www.youtube.com/watch?v=KKlCTxmzcS0
http://www.youtube.com/watch?v=aES-ZfanF3c

Live Events tutorial:
http://www.youtube.com/watch?v=O9WU7DzdUmE
http://www.youtube.com/watch?v=Hs3eO7o349k
http://www.youtube.com/watch?v=yUMzp6556Kw

New editing features in 0.6.0:
http://www.youtube.com/watch?v=Hk1qPlnyv88

Csoundo (Csound and Processing)

http://csoundblog.com/2010/08/csound-processing-experiment-i/

Open Sound Control in Csound

http://www.youtube.com/watch?v=JX1C3TqP_9Y

THE CSOUND CONFERENCE IN HANNOVER (2011)

Web page with papers and program.

All Videos can be found via the YoutTube channel csconf2011.

EXAMPLE COLLECTIONS

Csound Realtime Examples by Iain McCurdy is one of the most inspiring and up-to-date collections.

The Amsterdam Catalog by John-Philipp Gather is particularily interesting because of the adaption of Jean-Claude Risset's famous "Introductory Catalogue of Computer Synthesized Sounds" from 1969.

BOOKS

The Csound Book (2000) edited by Richard Boulanger is still the compendium for anyone who really wants to go in depth with Csound.

Virtual Sound by Alessandro Cipriani and Maurizio Giri (2000)

Signale, Systeme, und Klangsysteme by Martin Neukom (2003, german) has many interesting examples in Csound.

The Audio Programming Book edited by Richard Boulanger and Victor Lazzarini (2011) is a major source with many references to Csound.

Csound Power! by Jim Aikin (2012) is a perfect up-to-date introduction for beginners.

68. BUILDING CSOUND

Currently (April 2012) a collection of build instructions has been started at the Csound Media Wiki at Sourceforge. Please have a look there if you have problems in building Csound.

LINUX

Debian

On Wheezy with an amd64 architecture.

Download a copy of the Csound sources from the Sourceforge. To do so, in the terminal type:

```
git clone --depth 1 git://csound.git.sourceforge.net/gitroot/csound/csound5
```

Use aptitude to get (at least) the dependencies for a basic build, which are: libsndfile1-dev, python2.6-dev, scons. To do so, use the following command (with sudo or as root):

```
aptitude install libsndfile1-dev python2.6-dev scons
```

There are many more optional dependencies, which are recommended to get in most cases (some are already part of Debian), and which are documented here. I built with the following libraries installed: libportaudiocpp0, alsa, libportmidi0, libfltk1.1, swig2.0, libfluidsynth1 and liblo7. To install them (some might already be in your sistem), type:

```
aptitude install libportaudiocpp0 alsa libportmidi0 libfltk1.1 swig2.0 libfluidsynth1 liblo7
```

Go inside the csound5/ folder you downloaded from sourceforge, and edit build-linux-double.sh in order to meet your building needs, once again, read about the options in the Build Csound section of the manual.

On amd64 architectures, it is IMPORTANT to change gcc4opt=atom to gcc4opt=generic (otherwise it will build for single processor). I also used buildNewParser=0, since I could not get to compile with the new parser. To finally build, run the script:

```
./build-linux-double.sh
```

If the installation was successful, use the following command to install:

```
./install.py
```

Make sure that the following environment variables are set:

```
OPCODEDIR64=/usr/local/lib/csound/plugins64
CSSTRNGS=/usr/local/share/locale
```

If you built the python interface, move the csnd.py and -csnd.so from /usr/lib/python2.6/site-packages/ to /usr/lib/python2.6/dist-packages/ (the standard place for external Python modules since version 2.6). You can do so with the following commands:

```
/usr/lib/python2.6/site-packages/csnd.py /usr/lib/python2.6/dist-packages/

/usr/lib/python2.6/site-packages/_csnd.so /usr/lib/python2.6/dist-packages/
```

If you want to un-install, you can do so by running the following command:

```
/usr/local/bin/uninstall-csound5
```

Good luck!

Ubuntu

1. Download the sources. Either the last stable release from http://sourceforge.net/projects/csound/files/csound5/ or the latest (possible unstable) sources from git (running the command git clone git://csound.git.sourceforge.net/gitroot/csound/csound5).

2. Open a Terminal window and run the command

```
 sudo apt-get install csound
```

This should install all the dependencies which are needed to build Csound.

3. Change the directory to the folder you have downloaded in step 1, using the command cd.

4. Run the command scons. You can start with

```
scons -h
```

to check the configuration and choose your options. See the Build Csound section of the manual for more information about the options. If you want to build the standard configuration, just run scons without any options.

If you get an error, these are possible reasons:

- You must install bison and flex to use the new parser.
- If there is a complaint about not finding a file called custom.py, copy the file custom-linux-jpff.py and rename it as custom.py.

There is also a detailed instruction by Menno Knevel at csounds.com which may help.

5. Run

```
sudo python install.py
```

You should now be able to run csound by the command /usr/local/bin/csound, or simply by the command csound.

OSX

As mentioned above, have a look at http://sourceforge.net/apps/mediawiki/csound/index.php?title=Main_Page.

WINDOWS

There is a detailed description of Michael Gogins, entitled *How to Build Csound on Windows* in the Csound Sources. You can either download the Csound Sources at http://sourceforge.net/projects/csound/files/csound5 or get the latest version at the Csound Git Repository.

69. METHODS OF WRITING CSOUND SCORES

Although the use of Csound real-time has become more prevalent and arguably more important whilst the use if the score has diminished and become less important, composing using score events within the Csound score remains an important bedrock to working with Csound. There are many methods for writing Csound score several of which are covered here, starting with the classical method of writing scores by hand, and concluding with the definition of a user-defined score language.

WRITING SCORE BY HAND

In Csound's original incarnation the orchestra and score existed as separate text files. This arrangement existed partly in an attempt to appeal to composers who had come from a background of writing for conventional instruments by providing a more familiar paradigm. The three unavoidable attributes of a note event - which instrument plays it, when, and for how long - were hardwired into the structure of a note event through its first three attributes or 'p-fields'. All additional attributes (p4 and beyond), for example: dynamic, pitch, timbre, were left to the discretion of the composer, much as they would be when writing for conventional instruments. It is often overlooked that when writing score events in Csound we define start times and durations in 'beats'. It just so happens that 1 beat defaults to a duration of 1 second leading to the consequence that many Csound users spend years thinking that they are specifying note events in terms of seconds rather than beats. This default setting can easily be modified and manipulated as shown later on.

The most basic score event as described above might be something like this:

```
 i 1 0 5
```

which would demand that instrument number '1' play a note at time zero (beats) for 5 beats. After time of constructing a score in this manner it quickly becomes apparent that certain patterns and repetitions recur. Frequently a single instrument will be called repeatedly to play the notes that form a longer phrase therefore diminishing the worth of repeatedly typing the same instrument number for p1, an instrument may play a long sequence of notes of the same duration as in a phrase of running semiquavers rendering the task of inputting the same value for p3 over and over again slightly tedious and often a note will follow on immediately after the previous one as in a legato phrase intimating that the p2 start-time of that note might better be derived from the duration and start-time of the previous note by the computer than to be figured out by the composer. Inevitably short-cuts were added to the syntax to simplify these kinds of tasks:

```
i 1 0 1 60
i 1 1 1 61
i 1 2 1 62
i 1 3 1 63
i 1 4 1 64
```

could now be expressed as:

```
i 1 0 1 60
i . + 1 >
i . + 1 >
i . + 1 >
i . + 1 64
```

where '.' would indicate that that p-field would reuse the same p-field value from the previous score event, where '+', unique for p2, would indicate that the start time would follow on immediately after the previous note had ended and '>' would create a linear ramp from the first explicitly defined value (60) to the next explicitly defined value (64) in that p-field column (p4).

A more recent refinement of the p2 shortcut allows for staccato notes where the rhythm and timing remain unaffected. Each note lasts for 1/10 of a beat and each follows one second after the previous.

```
i 1 0   .1 60
i . ^+1 .  >
i . ^+1 .  >
i . ^+1 .  >
i . ^+1 .  64
```

The benefits offered by these short cuts quickly becomes apparent when working on longer scores. In particular the editing of critical values once, rather than many times is soon appreciated.

Taking a step further back, a myriad of score tools, mostly also identified by a single letter, exist to manipulate entire sections of score. As previously mentioned Csound defaults to giving each beat a duration of 1 second which corresponds to this 't' statement at the beginning of a score:

```
t 0 60
```

"At time (beat) zero set tempo to 60 beats per minute"; but this could easily be anything else or evena string of tempo change events following the format of a linsegb statement.

```
t 0 120 5 120 5 90 10 60
```

This time tempo begins at 120bpm and remains steady until the 5th beat, whereupon there is an immediate change to 90bpm; thereafter the tempo declines in linear fashion until the 10th beat when the tempo has reached 60bpm.

'm' statements allow us to define sections of the score that might be repeated ('s' statements marking the end of that section). 'n' statements referencing the name given to the original 'm' statement via their first parameter field will call for a repetition of that section.

```
m verse
i 1 0   1 60
i . ^+1 .  >
i . ^+1 .  >
i . ^+1 .  >
i . ^+1 . 64
s
n verse
n verse
n verse
```

Here a 'verse' section is first defined using an 'm' section (the section is also played at this stage). 's' marks the end of the section definition and 'n' recalls this section three more times.

Just a selection of the techniques and shortcuts available for hand-writing scores have been introduced here (refer to the Csound Reference Manual for a more encyclopedic overview). It has hopefully become clear however that with a full knowledge and implementation of these techniques the user can adeptly and efficiently write and manipulate scores by hand.

EXTENSION OF THE SCORE LANGUAGE: BIN="..."

It is possible to pass the score as written through a pre-processor before it is used by Csound to play notes. instead it can be first interpretted by a binary (application), which produces a usual csound score as a result. This is done by the statement bin="..." in the <CsScore> tag. What happens?

1. If just a binary is specified, this binary is called and two files are passed to it:
   a. A copy of the user written score. This file has the suffix *.ext*
   b. An empty file which will be read after the interpretation by Csound. This file has the usual score suffix *.sco*
2. If a binary and a script is specified, the binary calls the script and passes the two files to the script.

If you have Python[1] installed on your computer, you should be able to run the following examples. They do actually nothing but print the arguments (= file names).

EXAMPLE ...csd: Calling a binary without a script

```
<CsoundSynthesizer>
<CsInstruments>
instr 1
endin
```

```
</CsInstruments>
<CsScore bin="python">
from sys import argv
print "File to read = '%s'" % argv[0]
print "File to write = '%s'" % argv[1]
</CsScore>
</CsoundSynthesizer>
```

When you execute this .csd file in the terminal, your output should include something like this:

```
File to read = '/tmp/csound-idWDwO.ext'
File to write = '/tmp/csound-EdvgYC.sco'
```

And there should be a complaint because the empty .sco file has not been written:

```
cannot open scorefile /tmp/csound-EdvgYC.sco
```

EXAMPLE .... csd: Calling a binary and a script

To test this, first save this file as *print.py* in the same folder where your .csd examples are:

```
from sys import argv
print "Script = '%s'" % argv[0]
print "File to read = '%s'" % argv[1]
print "File to write = '%s'" % argv[2]
```

Then run the ....csd:

```
<CsoundSynthesizer>
<CsInstruments>
instr 1
endin
</CsInstruments>
<CsScore bin="python print.py">
</CsScore>
</CsoundSynthesizer>
```

The output should include these lines:

```
Script = 'print.py'
File to read = '/tmp/csound-jwZ9Uy.ext'
File to write = '/tmp/csound-NbMTfJ.sco'
```

And again a complaint about the invalid score file:

```
cannot open scorefile /tmp/csound-NbMTfJ.sco
```

csbeats

Scripts

Scripting Language Examples

The following script uses a perl script to allow seeding options in the score. A random seed can be set as a comment; like ";;SEED 123". If no seed has been set, the current system clock is used. So there will be a different value for the first three random statements, while the last two statements will always generate the same values.

```
<CsoundSynthesizer>
<CsInstruments>
;example by tito latini

instr 1
  prints "amp = %f, freq = %f\n", p4, p5;
endin

</CsInstruments>
<CsScore bin="perl cs_sco_rand.pl">
```

```
i1  0  .01  rand()   [200 + rand(30)]
i1  +  .    rand()   [400 + rand(80)]
i1  +  .    rand()   [600 + rand(160)]
;; SEED 123
i1  +  .    rand()   [750 + rand(200)]
i1  +  .    rand()   [210 + rand(20)]
e

</CsScore>
</CsoundSynthesizer>
```

```
# cs_sco_rand.pl
my ($in, $out) = @ARGV;
open(EXT, "<", $in);
open(SCO, ">", $out);

while (<EXT>) {
  s/SEED\s+(\d+)/srand($1);$&/e;
  s/rand\(\d*\)/eval $&/ge;
  print SCO;
}
```

1. www.python.org^