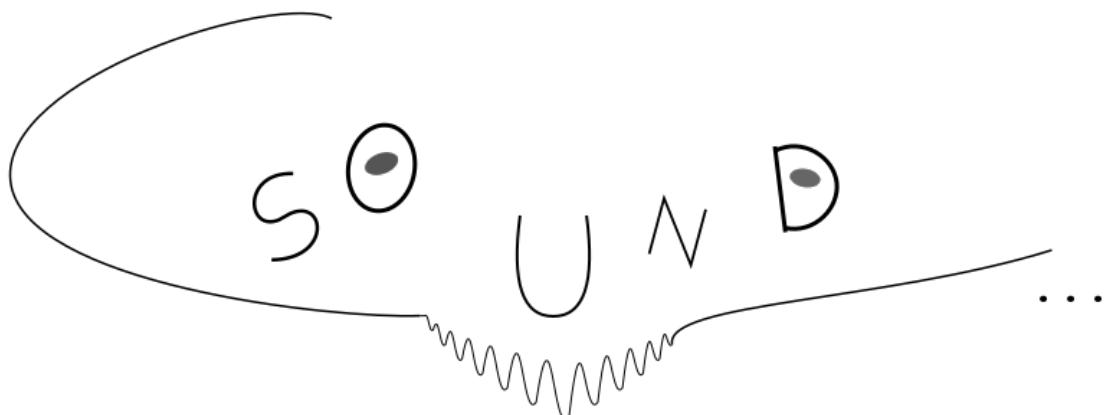


WELCOME TO CSOUND!



... is one of the best known and longest established programs in the field of audio programming. It has been first released in 1986 at the Massachusetts Institute of Technology (MIT) by Barry Vercoe. But Csound's history lies even deeper within the roots of computer music as it is a direct descendant of the oldest computer program for sound synthesis, *MusicN*, by Max Mathews. Csound is free and open source, distributed under the LGPL licence, and it is maintained and expanded by a core of developers with support from a wider global community.

In the past decade, thanks to the work of Victor Lazzarini, Steven Yi, John ffitch, Hlöðver Sigurðsson, Rory Walsh, and many others, Csound has moved from a somehow archaic audio programming language to a modern audio library. It can not only be used from the command line and the classical frontends. It can also be used as a VST plugin. It can be used inside the Unity game engine. It can be used on Android or on any microcomputer like Raspberry Pi or Bela Board. It can be used via its Application Programming Interface (API) in any other programming language, like Python, C++ or Java. And it can now also be used inside any browser as a JavaScript library, just by loading it as Web Assembly module (WASM Csound). Try it here, if you already know some Csound:

```
<CsoundSynthesizer>
<CsOptions>
-o dac
</CsOptions>
<CsInstruments>
sr = 44100
ksmps = 64
nchnls = 2
0dbfs = 1
```

```
instr TryMe
    //some code here ...
endin
schedule( "TryMe" , 0 , -1)

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

This textbook cannot cover all these use cases. Its main goal is:

- To provide an interactive [Getting Started](#) tutorial.
- To collect as many as possible [How to](#) recipes.
- To offer a readable and comprehensive introduction to the Csound language in [Chapter 03](#).
- To discuss some classical and recent methods of sound synthesis in [Chapter 04](#) and of sound modification in [Chapter 05](#).
- To offer an [Opcode Guide](#) as orientation in the overwhelming amount of Csound opcodes.
- To collect different in-depth descriptions and instructions to various subjects, without being complete, and sometimes not up-to-date, in chapters 06-14.

This book is called the *Csound FLOSS Manual* because it has been first released in 2011 at [floss-manuals.net](#). It should not be confused with the *Csound Reference Manual* which can be found [here](#).

Enjoy reading and coding, and please help improve this textbook by feedbacks and suggestions on the Github [discussions](#) page or elsewhere.

ON THIS RELEASE

Csound 7 is finally out!

We have a stable Csound 7 beta since many months, and thanks to work by Hlöðver Sigurðsson and Steven Yi, we have now a [Csound-WASM](#) version of it, too. Hlöðver has applied this also to the Csound FLOSS Manual, so that we can use Csound 7 now here!

As a brief introduction to the exciting new features in Csound 7, I have added a [chapter](#) to the Appendix.

As another good news: Xavier Dole has translated the *Getting Started* to French. It is available at <https://flossmanual.csound.com/fr>, and other parts may be added in future. Other translations are welcome!

Enjoy and thanks in advance for any feedback which is really helpful to improve this book!

Seoul, September 2025

joachim heintz

ON THIS RELEASE

01 Hello Csound

What you learn in this tutorial

- ▶ How to create and output a **sine** tone.
- ▶ What Csound **opcodes** are.
- ▶ What **audio rate** in Csound means, and
- ▶ What an **audio variable** is.
- ▶ How we can draw a **signal flow**.
- ▶ What a Csound **instrument** is, and
- ▶ What a Csound **score** is.

What is a Sine Oscillator

A sine wave can be seen as the most elementary sound in the world. When we draw a sine wave as a graph showing amplitude over time, it looks like this:



Figure 3.1: Sine wave snake

To produce a sine wave, Csound uses an oscillator. An oscillator needs certain input in order to run:

1. A maximum amplitude to output. This results in louder or softer tones.
2. The number of periods (cycles) per second to create. This results in higher or lower pitches. The unit is *Hertz (Hz)*. 1000 Hz would mean that a sine has 1000 periods in each second.

A Sine Oscillator in Csound: Opcode and Arguments

Csound has many different oscillators. (You can find [here](#) some descriptions and comparisons.) In this example, we use the opcode `poscil` which means “precise oscillator”.

An **opcode** is a processing unit in Csound, like an “object” in PureData or Max, or a “UGen” in SuperCollider. If you are familiar with programming languages, you can see an opcode as a built-in function.

The inputs of an opcode are called **arguments** and are written in parentheses immediately after the opcode name. The arguments are separated by commas.

So `poscil(0.2,400)` means: The opcode `poscil` gets two input arguments.

- The first argument is the number `0.2`.
- The second argument is the number `400`.

The meaning of the input arguments depends on how the opcode is implemented. For `poscil`, the first input is the amplitude, and the second input is the frequency. The [Csound Reference Manual](#) contains the information about it. We will give in [Tutorial 08](#) some help how to use it.

This way of writing code is very common in programming languages, like `range(13)` in [Python](#), or `printf("no no")` in [C](#), or `Date.now()` in [JavaScript](#) (in the latter case with empty parentheses which means: no input arguments).

Note: There is also another way of writing Csound code. See [below](#) if you want to learn more about it.

A Signal Flow and its Code

We now create a sine wave of 0.2 amplitude and 400 cycles per second (Hz).

We will call this signal `aSine` because it is an audio signal. The character `a` at the beginning of the variable name signifies exactly this.

An audio signal is a signal which produces a new value every sample. (Learn more [here](#) about samples and sample rate).

This is the code line in Csound to produce the `aSine` signal:

```
aSine = poscil:a(0.2,400)
```

This means: The signal `aSine` is created by the opcode `poscil` at audio rate (`:a`), and the input for `poscil` is `0.2` for the amplitude and `400` for the frequency.

To output the signal (so that we can hear it), we put it in the `outall` opcode. This opcode sends an audio signal to all available output channels.

```
outall(aSine)
```

Note that the signal `aSine` at first was the output of the oscillator, and then became input of the `outall` opcode. This is a typical chain which is well known from modular synthesizers: A cable connects the output of one module with the input of another.

We can draw the program flow like this:

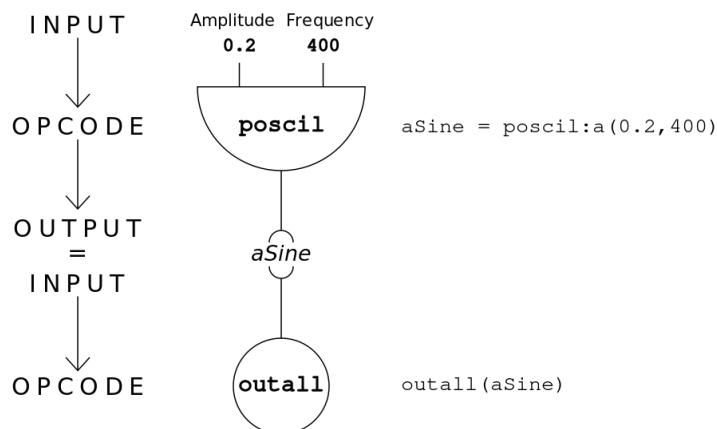


Figure 3.2: Signal flow and Csound code for sine oscillator and output

In the middle you see the signal flow, with symbols for the oscillator and the output. You can imagine them as modules of a synthesizer, connected by a cable called `aSine`.

On the left hand side you see the chain between input, opcode and output. Note that the output of the first chain, contained in the `aSine` variable, becomes the input of the second chain.

On the right hand side you see the Csound code. Each line of code represents one *input -> opcode -> output* chain, in the form `output = opcode(input)`. The line `outall(aSine)` does not have an output in Csound, because it sends audio to the hardware (similar to the “`dac~`” object in PD or Max).

Your first Csound instrument

In Csound, all oscillators, filters, sample players and other processing units are encapsulated in an **instrument**. An instrument has the keyword `instr` at its start, and `endin` at its end.

After the keyword `instr`, separated by a space, we assign a number (1, 2, 3, ...) or a name to the instrument. Let us call our instrument “Hello”, and include the code which we discussed:

```

instr Hello
    aSine = poscil:a(0.2,400)
    outall(aSine)
endin
    
```

Example

We are now ready to run the code. All we have to do is put the instrument in a complete Csound file.

Look at the code in the example. Can you find the instrument code?

Push the “Play” button. You should hear two seconds of a 400 Hz sine tone.

Can you see why it plays for two seconds?

```
<CsoundSynthesizer>
<CsOptions>
-o dac
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 64
nchnls = 2
0dbfs = 1

instr Hello
    aSine = oscil:a(0.2,400)
    outall(aSine)
endin

</CsInstruments>
<CsScore>
i "Hello" 0 2
</CsScore>
</CsoundSynthesizer>
```

The Csound Score

At the bottom of the example code, you find this:

```
<CsScore>
i "Hello" 0 2
</CsScore>
```

This is the **Score** section of the .csd File. it starts with the tag `<CsScore>` and ends with `</CsScore>`. Between these two tags is this score line:

```
i "Hello" 0 2
```

Each column (parameter field) specifies certain information:

i: This is an instrument event.

"Hello": The instrument which this score line refers to.

0: The start time of this instrument, 0 (= start immediately).

2: The duration of this instrument, 2 (seconds).

The parameters are separated from each other by spaces or tabs. (Not by commas as the arguments of an opcode.)

Try it yourself

(You can edit the example just by typing in it.)

- Change the duration of the instrument.
- Change the start time of the instrument.
- Change the frequency of the oscillator.
- Change the amplitude of the oscillator.

Opcodes, Keywords and Terms you have learned in this tutorial

Opcodes

- `poscil:a(Amplitude, Frequency)` oscillator at audio rate, with amplitude and frequency input.
- `outall(aSignal)` outputs `aSignal` to all output channels.

Keywords

- `instr ... endin` are the keywords to begin and end an instrument definition.

Terms

- An *audio rate* or *a-rate* signal is a signal which is updated sample by sample.

Go on now ...

with the next tutorial: [02 Hello Frequency](#).

... or read some more explanations here

Why is a sine “the most elementary sound in the world”?

To be honest, I like sine tones. I know that many people find them boring. I like their simplicity, so against all good advices I will spend the first ten Tutorials with sine tones only. Sorry for being my victim to it ...

But back to the question: What is elementary at sine waves?

From a mathematical point of view it is quite fascinating that we can understand and construct a sine as constant movement of a point on a circle. This is called [simple harmonic motion](#) and is fundamental for many phenomena in the physical world, including sound.

For musical acoustics, there is another meaning of sines being elementary.

A sine tone is the only sound which represents only **one** pitch. All other sounds have two or more pitches in themselves.

This means: All other sounds can be understood as addition of simple sine tones. These sines which are inside a periodic sound, like a sung tone or other natural pitched sounds, are called **partials or harmonics**.

Although the sounding reality is a bit more complex, it shows that sine waves can somehow be understood as most elementary sounds, at least in the world of pitches.

You can find more about this subject in the [Additive Synthesis](#) chapter and in the [Spectral Resynthesis](#) chapter of this book.

Traditional and functional way to write Csound code

Perhaps you are surprised to see Csound code written in the way it is described above. In fact, the classical way to write Csound code is like this:

```
aSine poscil 0.2, 400  
outall aSine
```

You are welcome to continue writing code in this way. The reasons why I use the “functional” way of writing Csound code in these Tutorials are:

1. We are all familiar with this way to declare a left hand side variable y to be the sum of another variable x plus two: $y = x + 2$ Or, to yield a y as function of x : $y = f(x)$ I think it is good to build upon this familiarity. It is hard enough for a musician to learn a programming language, never having heard about variables, signal flow, input arguments, or parameters. What ever let you feel more familiar in this new world is helpful and should be used.
2. As mentioned above, most other programming languages use a similar syntax, in the form

output = function(arguments). So for people who already know any other programming language, it helps learning Csound.

3. The functional style of writing Csound code has always been there in expressions like `ampdb(-10)` or `ftlen(giTTable)`. So it is somehow not completely new but an extension.
4. Whenever we want to use an expression as argument (you will learn more about it in [Tutorial 06](#)) we need this way of writing code. So it is good to use it as consistent as possible.

About these tutorials

This *Getting Started* has been written by Joachim Heintz in 2023. It is based on many experiences in teaching Csound to young composers. Thanks goes to all students and friends, from Hanover University of Music, from Yarava Music Group Tehran, and elsewhere, who gave me feedback and contributed in one or another way to this and other texts and contents. Amin and Parham, Marijana and Betty, Tom and Farhad, Ehsan and Vincent, Julio and Arsalan, to name some of them.

Thanks go also to Wolfgang Fohl, Tarmo Johannes, Rory Walsh and others for their comments and help.

And of course to the Core Developers whose great work made it possible that we all can use Csound in the way we do. I hope this Tutorial can show to some more musicians how admirable and successful the big effort was and is to turn the oldest audio programming language into a modern one, without losing any composition written in the Csound language decades ago.

Each Tutorial has a first part as *must read* (well which *must* is that), followed by an optional part (in which of course the most interesting things reside). To make at least one thing in the world reliable, each *must read* consists of five headings, and each *can read* of three. Actually I planned it to be 4+3, but then I asked Csound, and received this answer:

```
if 4+3 == 7 then
    "Write this Tutorial!\n"
else
    8!
endif
```

So I took this as oracle and decided for 5+3 headings, to fulfill also the 8! requirement. It is always better to satisfy both gods of a conditional branch, in my experience.

Included in the *must read* part is a *must do*. At first a central example, yes very central, substantial, expedient, enjoyable, and of course very instructive. And then a *Try it yourself* which is kind of the dark side of the example: As easy it is to just push the “Run” button, as hard will it perhaps be to solve these damned exercises. But, to quote John ffitch: “We have all been there ...”

I guess, no I have run a long series of tests, no it has been proven by serious studies held by the most notable and prestigious universities in the most important parts of the world, that the average time needed to go through one Tutorial, is one hour. So once I finished the planned number of 24 Tutorials then finally the goal is reached to **learn Csound in one day**.

Nevertheless, I must admit that this *Getting Started* can only be one amongst many. Its focus is on learning the language: How to think and to program in Csound. I believe that for those who

understand this, and enjoy Csound's simplicity and clarity, all doors are open to go anywhere in the endless world of this audio programming language. It be live coding, Bela Board or Raspberry Pi, it be noise music or the most soft and subtle sounds, it be fixed media or the fastest possible real-time application, it be using Csound as standalone or as a plugin to your preferred DAW. As for sounds, please have a look, and in particular an ear, to Iain McCurdy's examples, either on his [website](#) or inside [Cabbage](#). They are an inexhaustible source of inspiration and a yardstick for sound quality.

02 Hello Frequency

What you learn in this tutorial

- How to create a sliding frequency or **glissando**.
- What **k-rate** or **control rate** means and
- What **k-variables** in Csound are.
- How a Csound **.csd document** is structured:
- What **CsOptions** are and
- What the **CsInstruments** or **orchestra** is.

A Line for the Frequency

In natural sounds, the frequency is rarely fixed. Usually, for instance when we speak, the pitch of our voice varies all the time, in a certain range.

The most simple case of such a movement is a line. We need three values to construct a line:

1. A value to start with.
2. A time to move from this value to the target value.
3. The target value itself.

This is a line which moves from 500 to 400 in 0.5 seconds, and then stays at 400 for 1.5 seconds:

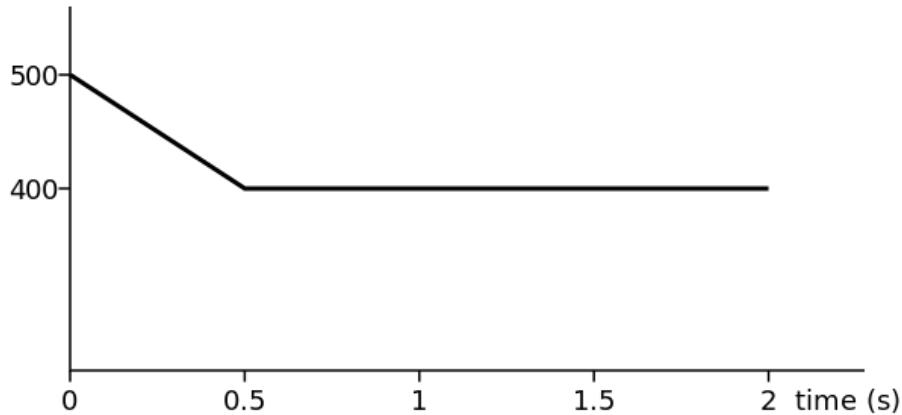


Figure 4.1: Frequency line

Note: Acoustically this way to apply a *glissando* is questionable. We will discuss this in [Tutorial 05](#).

Note: Should we not say: "This is a line which moves from 500 Hz to 400 Hz in 0.5 seconds", rather than: "This is a line which moves from 500 to 400 in 0.5 seconds"? No. The line outputs numbers. These numbers can be used for frequencies, but in another context they can have a different meaning.

A Line Drawn with the 'linseg' Opcode

In Csound, we create a line like this with the opcode `linseg`. This comes from "linear segments". Here we only need one segment which moves from 500 to 400 in 0.5 seconds.

This is the Csound code for this linear movement:

```
kFreq = linseg:k(500,0.5,400)
```

You will recognize the structure `opcode(arguments)` which we already used in the first tutorial. Here, the opcode is `linseg`, and the arguments are 500 as first value, 0.5 as duration to move to the next value, and 400 as target value.

But why is the variable on the left side called `kFreq`, and why is `linseg` written as `linseg:k`?

k-rate Signals

A signal is something whichs values change in time.

But what is time in an audio application like Csound?

The fundamental time is given by the **sample rate**. It determines how many audio samples we have in one second. We saw in the first tutorial the signal `aSine` whichs values change in **audio rate**; calculating a new value for every sample.

The second possible time resolution in Csound is less fine grained. It does not calculate a new value for every sample, but only for a **group of samples**. This time resolution is called **control rate**.

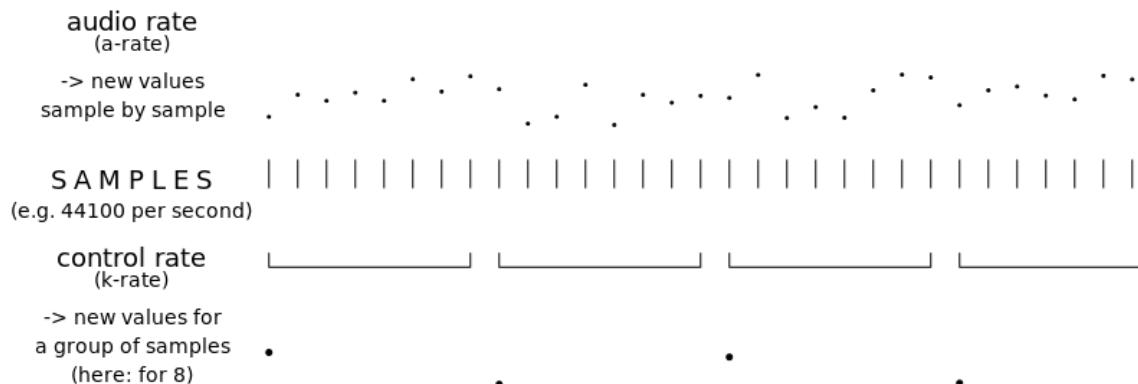


Figure 4.2: audio rate vs control rate

Variables in Csound which have **k** as first character, are using the control rate. Their values are updated in the control rate. This is the reason we write **kFreq**.

After the opcode, before the parenthesis, we put :k to make clear that this opcode uses the control rate as its time resolution.

I recommend always using **k** or **a** at **both** positions:

- Left hand side for the variable, and also
- Right hand side after the opcode.

```
aSine = poscil:a(0.2,400)
kFreq = linseg:k(500,0.5,400)
```

You will learn more about *k-rate* in the next tutorial.

Example

Push the “Play” button. You will hear a tone with a falling pitch.

Can you see how the moving line is fed into the oscillator?

```
<CsoundSynthesizer>
<CsOptions>
-o dac
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 64
nchnls = 2
0dbfs = 1

instr Hello
kFreq = linseg:k(500,0.5,400)
aSine = poscil:a(0.2,kFreq)
```

```

    outall(aSine)
  endin

  </CsInstruments>
  <CsScore>
  i "Hello" 0 2
  </CsScore>
</CsoundSynthesizer>
```

Signal flow

When you look at our instrument code, you see that there is a common scheme:

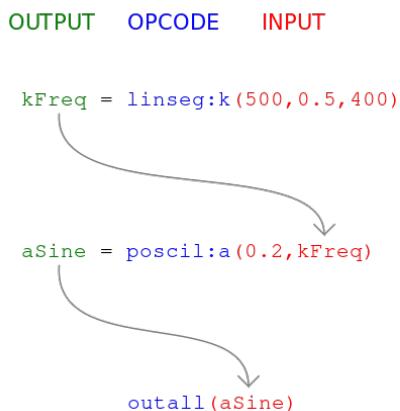


Figure 4.3: Input output insertions

The first line produces a signal and stores it in the variable *kLine*. This variable is then used as input in the second line.

The second line produces a signal and stores it in the variable *aSine*. This variable is then used as input in the third line.

The Csound Document Structure

The whole Csound document consists of three parts:

1. The *<CsOptions>* tag. You see here the statement: -o dac This means: The output (-o) shall be written to the digital-to-analog converter (dac); in other words: to the sound card. Because of this, we listen to the result in real time. Otherwise Csound would write a file as final result of its rendering.
2. The *<CsInstruments>* tag. Here all instruments are collected. This tag is the place for the Csound code. It is also called the Csound “Orchestra”.
3. The *<CsScore>* tag. This we discussed in the previous section.

As you see, all three tags are embraced by another tag:

```
<CsoundSynthesizer> ... </CsoundSynthesizer>
```

This tag defines the boundaries for the Csound program you write. In other words: What you write out of these boundaries will be ignored by Csound.

Try it yourself

- Let the frequency line move upwards instead of downwards.
- Use linseg to create a constant frequency of 400 or 500 Hz.
- Make the time of the *glissando* ramp longer or shorter.
- Add another segment by appending one more value for time, and one more value for the second target value.
- Restore the original arguments of linseg. Then replace linseg by line and hear what is different.

Opcodes, Tags and Terms you have learned in this tutorial

Opcodes

- linseg:k(Value1,Duration1,Value2,...) generates segments of straight lines

Tags

- <CsoundSynthesizer> ... </CsoundSynthesizer> starts and ends a Csound file.
- <CsOptions> ... </CsOptions> starts and ends the [Csound Options](#).
- <CsInstruments> ... </CsInstruments> starts and ends the space for defining Csound instruments.
- <CsScore> ... </CsScore> starts and ends the Csound score.

Terms

- A *control rate* or *k-rate* signal is a signal which is not updated every sample, but for a group or block of samples.

Go on now ...

with the next tutorial: [03 Hello Amplitude](#).

... or read some more explanations here

Linseg versus Line

Csound has a `line` opcode which we could use instead of `linseg`.

We can replace `linseg` in our code with `line`:

```
instr Hello
    kFreq = line:k(500,0.5,400)
    aSine = oscil:a(0.2,kFreq)
    outall(aSine)
endin
```

When you run this code, you will hear that `line` has one important difference to `linseg`: It will not stop at the target value, but continue its movement, in the same way as before:

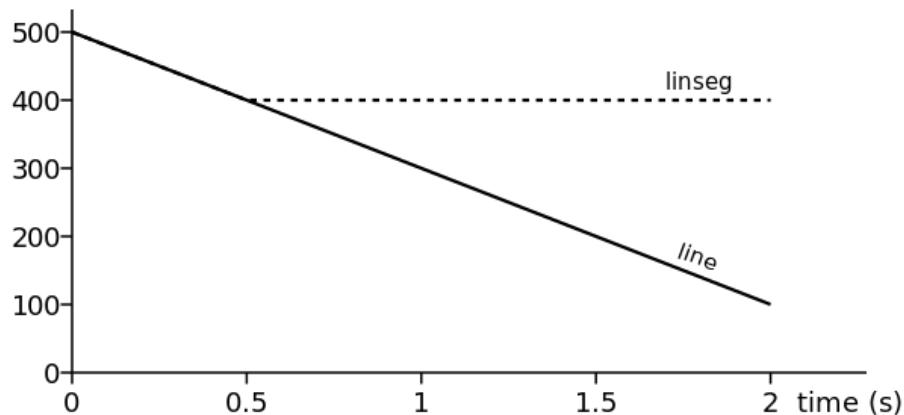


Figure 4.4: line vs linseg

Usually we do not want this, so I'd recommend to always use `linseg`, except some special cases.

Coding conventions

When you press the “Run” button, Csound “reads” the code you have written. Perhaps you already experienced that you wrote something which results in an error, because it is “illegal”.

For instance this code:

```
kFreq = linseg:k(500 0.5 400)
```

What is illegal here? We have separated the three input arguments for `linseg` not by commas, but by spaces. Csound expects commas, and if there is no comma, it returns a syntax error, and the code cannot be compiled:

```
error: syntax error, unexpected NUMBER_TOKEN,
expecting ',' or ')' (token "0.5")
```

This is not a convention, it is a syntax we have to accept when we want our code to be compiled and executed by Csound.

But inside this syntax we have many ways to write code in one way or another.

Let us look to some examples:

```
(1) kFreq = linseg:k(500,0.5,400)
(2) kFreq=linseg:k(500,0.5,400)
(3) kFreq = linseg:k(500, 0.5, 400)
(4) kFreq = linseg:k(500,.5,400)
(5) kFreq      =      linseg:k(500,    0.5,    400)
```

(1) This is the way I write code here in these tutorials. I put a space left and right the `=`, but I put no space after the comma.

(2) This is possible but perhaps you will agree that it is less clear because the `=` sign is somehow hidden.

(3) This is as widely used as (1), or more. I remember when I first read Guido van Rossum's Python tutorial in which he recommends to write as in (1), I did not like it at all. It took twenty years to agree ...

(4) This is a common abbreviation which is possible in Csound and some other programming languages. Rather than `0.5` you can just write `.5`. I use it privately, but will not use it here in these tutorials because it is less clear.

(5) You are allowed to use tabs instead of spaces, and each combination of it, and as many tabs or spaces as you like. But usually we do not want a line to be longer than necessary.

Another convention is to write the keywords `instr` and `endin` at the beginning of the line, and then the code indented by two spaces:

```
instr Hello
  kFreq = linseg:k(500,0.5,400)
  aSine = poscil:a(0.2,kFreq)
  outall(aSine)
endin
```

The reason for this convention is again to format the code in favor of maximum clarity. In the [Csound Book](#) we used one space, but I think two spaces are even better.

To summarize: You have a lot of different options to write Csound code. You can do what you like, but it is wise to accept some conventions which serve for maximal comprehension. The goal is readable and transparent code.

When to choose a-rate or k-rate?

The main reason for introducing *k*-rate was to save CPU power. This was essential in the early years of computer music. Nowadays it is not the same. For usual instruments in electronic music, we can omit *k*-rate variables, and only use *a*-rate variables. In our case, we can write:

```
aFreq = linseg:a(500,0.5,400)  
aSine = poscil:a(0.2,aFreq)  
outall(aSine)
```

As a simple advice:

- Always use *a*-rate when it sounds better.
- Use *k*-rate instead of *a*-rate when you must save CPU power.
- Some opcodes only accept *k*-rate input, but no *a*-rate input. In this case you must use *k*-rate variables.

03 Hello Amplitude

What you learn in this tutorial

- How to create a **time-varying amplitude**.
- What **orchestra header constants** in Csound are:
- The **sample rate**,
- The **ksmps** or **block size**,
- The **nchnls** as number of output channels,
- The **0dbfs** as zero dB full scale equivalent.
- How the **order of execution** works in a Csound program.
- How you can **understand error messages**.

A Line for the Amplitude

Once we understood how to create lines, we can apply a linear movement to the amplitude, too.

This is a signal which moves from 0.3 to 0.1 in half a second:

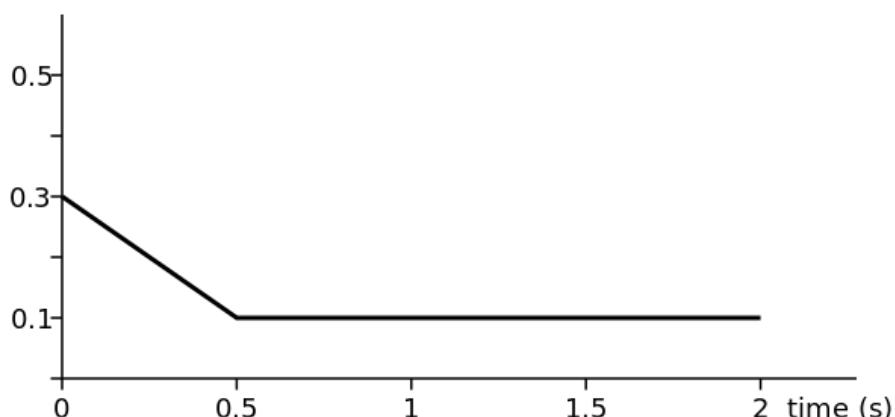


Figure 5.1: Amplitude decay

And this code will already look quite familiar to you:

```
kAmp = linseg:k(0.3,0.5,0.1)
```

Note: Acoustically this way to change volume is questionable. We will discuss this in Tutorial 06.

Back to the Head

Let us now look at some important settings which are written at the beginning of the <CsInstruments> tag:

```
sr = 44100
ksmps = 64
nchnls = 2
0dbfs = 1
```

This is sometimes called the “Orchestra Header”.

We see four constants which should be written explicitly in every Csound file.

They are called **constants** because you cannot change them during one run of Csound. Once you start Csound (by pushing the “Play” button), and Csound compiles, these values remain as they are for this run.

What is the meaning of these constants?

sr

sr means **sample rate**. This is the number of audio samples in one second. Common values are 44100 (the CD standard), 48000 (video standard), or higher values like 96000 or 192000.

As the sample rate is measured per second, it is often expressed in Hertz (Hz), or kilo Hertz (kHz).

You should choose the sample rate which fits to your needs. When you produce audio for an *mpeg* video which needs 48 kHz, you should set:

```
sr = 48000
```

When your sound card runs at 44100 Hz, you should set your **sr** to the same value; or vice versa set your sound card to the value you have as **sr** in Csound.

ksmps

ksmps means: **Number of samples in one control period**.

As you learned in [Tutorial 02](#), the *k*-rate is based on the sample rate. A **group** or **block of samples** is collected in a package. Imagine yourself standing besides a belt. Little statues arrive on the belt in a regular time interval, say once a second. Rather than taking each statue, and throwing it to a colleague, you collect 64 of the cubes in one package. You throw this package once the 64 statues are completed. You will throw one package every 64 seconds.

This “time interval to throw a package” (as packages per second) is the control rate or **k-rate**. The time interval in which the little statues arrive one by one is the audio-rate or **a-rate**. In our example, the *k*-rate is 64 times slower than the *a*-rate.

The **ksmps** constant defines how many audio samples are collected for one *k*-rate package. What we called “package” here, is in technical terms called “block” or “vector”. So *ksmps* is exactly the same as “block size” in PureData or “vector size” in Max.

nchnls

nchnls means **number of channels**.

When you have a stereo sound card, you will set **nchnls = 2**.

When you use eight speakers, you must set **nchnls = 8**. You will need a sound card with at least eight channels for real-time output.

Note: Csound assumes that you have the same number of input channels as you have output channels. If this is not the case, you must use the **nchnls_i** constant to set the number of input channels. For instance, in case you have 8 output channels but only 4 input channels on your audio interface, set:

```
nchnls = 8
nchnls_i = 4
```

0dbfs

0dbfs means: **What is the amplitude which represents 0 dB full scale**.

This sounds a bit more complicated as it is.

We define here a number which represents the maximum possible amplitude.

We should always set this number to 1, as this is what all audio applications reasonably do. 0 is minimum amplitude (meaning “silence”), and 1 is maximum amplitude.

Always set

```
0dbfs = 1
```

and you will be fine.

(We will explain more in [Tutorial 05](#) about 0 dB which is part of this constant.)

Default Values in the Orchestra Header

Perhaps you have seen a Csound file in which the four important constants are not defined.

In this case, Csound will use the “default” values.

The default value for **sr** is 44100. This is acceptable, but it is better to set it explicitly (and perhaps change it then if necessary).

The default value for **ksmps** is 10. This is not good because it is not a power of two. (See [below](#) why this is preferred.)

The default value for **nchnls** is 1. This is not good because usually we want to use at least stereo.

The default value for **0dbfs** is 32767. This is not good at all. Please set it to 1.

It is strongly recommended to always set the four constants explicitly.

Example

Look how both, the *kAmp* and *kFreq* line, are created by the *linseg* opcode.

These two lines are fed via the variable names into the *poscil* oscillator.

```
<CsoundSynthesizer>
<CsOptions>
-o dac
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 64
nchnls = 2
0dbfs = 1

instr Hello
    kAmp = linseg:k(0.3,0.5,0.1)
    kFreq = linseg:k(500,0.5,400)
    aSine = poscil:a(kAmp,kFreq)
    outall(aSine)
endin

</CsInstruments>
<CsScore>
i "Hello" 0 2
</CsScore>
</CsoundSynthesizer>
```

Try it yourself

- Change the duration in the *kAmp* signal from 0.5 to 1 or 2. The frequency and the amplitude line are now moving independently from each other.
- Change the values of the *kAmp* signal so that you get an increasing rather than a decreasing amplitude.
- Change *0dbfs* to 2. You should hear that it sounds softer now because the “full scale” level is twice as high.

- Change 0dbfs to 0.5. You should hear that it sounds louder now because the “full scale” level is set to 0.5 now instead of 1.
- Move the line `aSine = poscil:a(kAmp,kFreq)` from third position in the instrument to the first position. Push the “Play” button and look at the error message. Can you understand why Csound is complaining?

Signal flow and Order of Execution

We can draw the signal flow of our instrument like this:

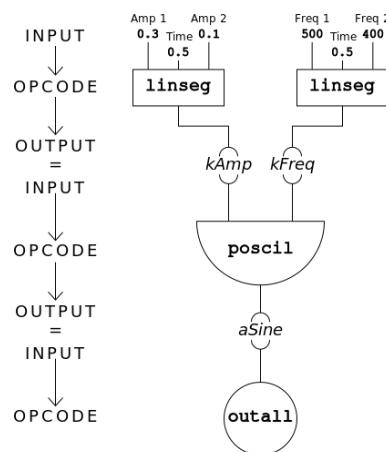


Figure 5.2: Signal flow

Please compare this version to the [signal flow diagram in Tutorial 01](#). The two inputs for the `poscil` oscillator are not any more two numbers, but two signals, as outputs of two `linseg` opcodes.

Csound reads our program code line by line. Whenever something is used as input for an opcode, it must exist at this time.

Read (!!!) the Error Messages

So when we put the third line of the instrument on top, we ask Csound to use something which is not known yet:

```

instr Hello
    aSine = poscil:a(kAmp,kFreq)
    kAmp = linseg:k(0.3,0.5,0.1)
    kFreq = linseg:k(500,0.5,400)
    outall(aSine)
endin

```

The `poscil` oscillator needs `kAmp` and `kFreq` as its input. But at this point of the program code, there is neither `kAmp` nor `kFreq`. So Csound will return a “used before defined” error message:

```
error: Variable 'kAmp' used before defined
```

As you see, Csound stops reading at the first error message. Once you put the `kAmp = linseg:k(0.3,0.5,0.1)` on top of the instrument, Csound will not complain any more about `kAmp`, but move to the next undefined variable:

```
error: Variable 'kFreq' used before defined
```

The error message has changed now. Csound does not complain any more about `kAmp` but about `kFreq`. Once you put the line `kFreq = linseg:k(500,0.5,400)` on first or second position, also this error message will disappear.

To summarize:

1. Learning a programming language is not possible without producing errors. This is an essential part of learning. But:
2. You MUST READ the error messages. Figure out where the “Csound console” is shown in your environment. (It can be hidden or in the background.) And then read until you meet the first error message.
3. The first! And then try to understand this message and follow the hints to solve it. Don’t worry if the next one shows up. You will solve them one by one.

Constants and Terms you have learned in this tutorial

Constants

- `sr` number of samples per second
- `ksmps` number of samples in one control block
- `nchnls` number of channels
- `0dbfs` number (amplitude) for zero dB full scale

Terms

- *Block Size* or *Vector Size* is what Csound sets as `ksmps`: the number of samples in one control cycle

Go on now ...

with the next tutorial: [04 Hello Fade-out](#).

... or read some more explanations here

Some notes about ksmmps

Note 1: It is recommended to use **power-of-two** values for ksmmps. Common values are 32 (= 2^5) or 64 (= 2^6). This is due to input/output handling of audio. You will find the same in other applications.

Note 2: The advantage of a **smaller** ksmmps is the better time resolution for the control rate. If the sample rate is 44100 Hz, we have a time resolution of 1/44100 seconds for each sample. This is about 0.000023 seconds or 0.023 milliseconds between two samples. When we set ksmmps = 64 for this sample rate, we get 64/44100 seconds as time resolution for each control value. This is about 0.00145 seconds or 1.45 milliseconds between two blocks or two control values. When we set ksmmps = 32 for the same sample rate, we get 0.725 milliseconds as time resolution for each new control value.

Note 3: The advantage of a **larger** ksmmps is the better performance in terms of speed. If you have a complex and CPU consuming Csound file, and get dropouts, you can try to increase ksmmps.

Note 4: Although ksmmps is a constant, we can set a **local ksmmps** in one instrument. The opcode for this operation is setksmps. Sometimes we wish to run a krate opcode in one instrument sample by sample. In this case, we can use setksmps(1). We only can split the global ksmmps into smaller parts, not vice versa.

How is this in PD?

Pure Data is another popular audio programming language. Other than Csound it is not text based but uses visual symbols for programming. You can imagine that the program flow which we have drawn [above](#) in symbols is now on your screen, and you connect the object boxes with symbolic cables.

(It is possible, by the way, to embed Csound in PD. Read more [here](#) in this book if you are interested.)

It might be good to have a look how the important constants like sample rate and *ksmps* are set in PD, and how they are called in PD. This comparison may help to understand what is happening in any audio application and audio programming language.

All settings can be found in the *Audio Settings* ... which look like this for my PD version:

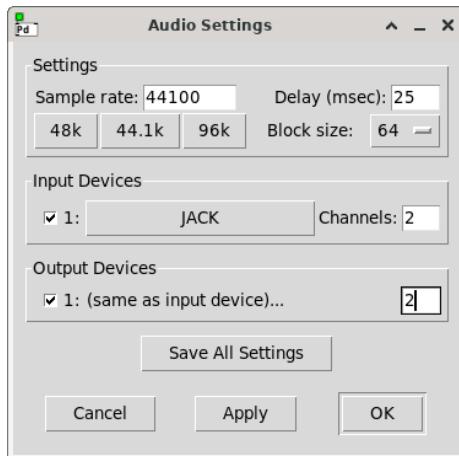


Figure 5.3: PD Audio Settings

Top left the **sample rate** is being set, as we did in Csound via

```
sr = 44100
```

Beneath the **block size** can be selected. This is exactly the same as the **ksmps** in Csound. The selected block size of 64 is what we did as:

```
ksmps = 64
```

Below we can set input and output devices, and the number of input and output channels. The latter is what we did via

```
nchnls = 2
```

(We did not select any audio device because we use Csound in a browser here, so it uses Web Audio. But of course you can select your input and output device in Csound, too.)

And what about **0dbfs**? This is always set to 1 in PD, so no field for it.

What is Open Source?

Both, PD and Csound, are “Open Source Software”. What does this mean?

You can read more about it for instance in the Wikipedia article about [Free and open source software](#). What does it mean for Csound being “free and open source”?

At first, all Csound code is visible to anyone. Currently the [source code](#) is on [Github](#). Anyone can not only read it, but also “fork” it. This means, simply put: To use it, to change it according to own needs and to distribute it under the same conditions as you forked it.

Most forks are used to send “pull requests” to the main Csound developers. This means: “I improved something; please merge it to the Csound code base.”

No one gets paid for developing Csound source code, for contributing to the [Csound website](#), for maintaining [frontends](#), mailing lists, forums, API’s, for writing documentation, examples, tutorials. The motivations are certainly different, but I guess that one is common: To enjoy collaborating and contributing to a software which is our common property and can be used by anyone, to materialize

creative musical ideas. If you like to read more about some of my own experiences and views, have a look [here](#).

Great to see that it works, since nearly four decades, with all highlights and shadows, as in all real things. If you read this and feel that you might contribute anything: Welcome! Have a look at the [Community](#) page of the Csound Website and join us. There is always work to do, and to have fun doing it ...

04 Hello Fade-out

What you learn in this tutorial

- How to apply **linear fades** to an instrument.
- What **p-fields** are.
- What **p3** in an instrument means.

Fade-in and Fade-out

Currently we have a rude end of the tone which is created by our instrument. This is due to a brutal cutoff at the end of the instrument duration.

Imagine a sine wave which is cut anywhere. Perhaps this will happen:

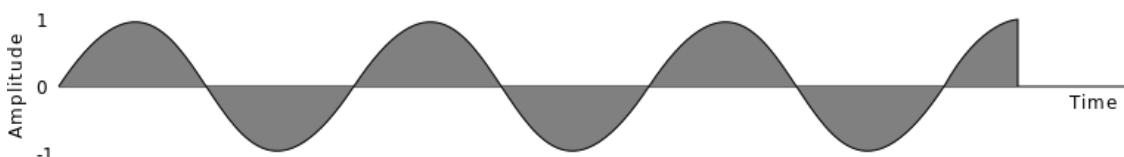


Figure 6.1: Click at the end of a sine wave

In this case we will hear a “click” at the end.

But even without this click: Usually we will prefer a soft end, similar to how natural tones end.

The Csound opcode `linen` is very useful for applying simple fades to an audio signal.

The ‘`linen`’ Opcode

The envelope which is created by `linen` looks like this:

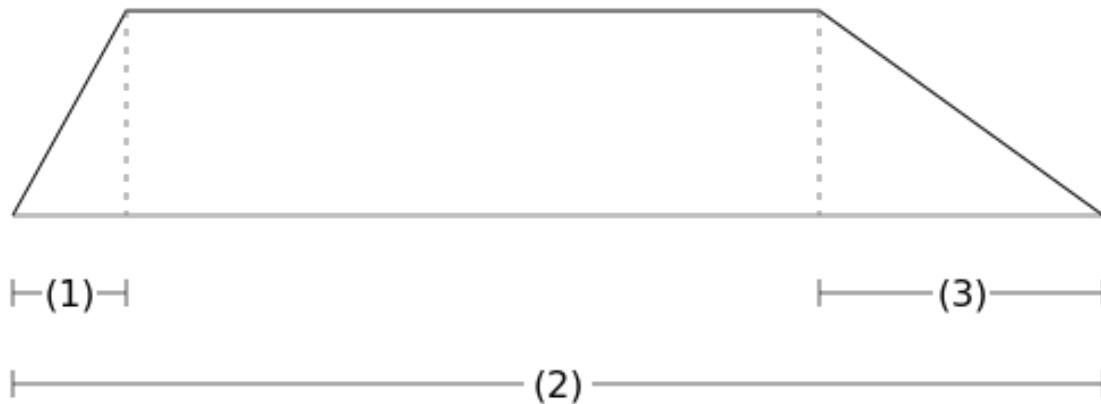


Figure 6.2: The linen opcode

We need three numbers to adjust the fades in `linen`: (1) The time to fade in. (2) The overall duration. (3) The time to fade out.

The Score Parameter Fields

Usually we want to set `linen`'s overall duration to the note's duration.

For instance, if the note's duration is 2 seconds, we will want 2 seconds for `linen`'s overall duration. If the note's duration is 3 seconds, we will want 3 seconds for `linen`'s overall duration.

In [Tutorial 01](#), you have learned how the instrument duration is set in Csound.

You will remember that this is written in the score:

```
i "Hello" 0 2
```

After the starting `i` which indicates an “instrument event”, we have three values:

1. The name of the instrument
2. The start time
3. The duration

We call these values **parameter fields**, or **p-fields**.

We have three p-fields in our score, abbreviated as **p1**, **p2** and **p3**.

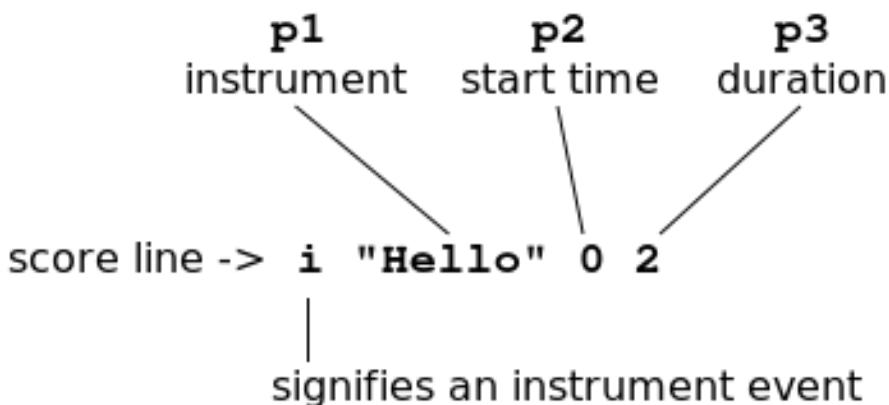


Figure 6.3: Score parameter fields

'p3' in an Instrument

A Csound instrument is instantiated via a score line.

Each score line has at least three parameter fields.

These parameters are passed to the instrument when it is instantiated.

So each instrument “knows” its **p1** (its number or name), its **p2** (its start time), and its **p3** (its duration).

And more:

Each instrument can **refer** to its score parameter fields by just writing **p1** or **p2** or **p3** in the code.

Example

In addition to the three input arguments for **linen** which we have discussed above, we have as first input argument the audio signal which we want to modify.

This is the order of the four input arguments for the **linen** opcode:

1. The audio signal to which the fade-in and fade-out is applied.
2. The fade-in duration (in seconds).
3. The overall duration for **linen**.
4. The fade-out duration.

Look at the code. Can you see how the code uses **p3** to adapt the duration of **linen** to the duration of the instrument?

```
<CsoundSynthesizer>
<CsOptions>
-o dac
</CsOptions>
<CsInstruments>
```

```

sr = 44100
ksmps = 64
nchnls = 2
0dbfs = 1

instr Hello
    kAmp = linseg:k(0.3,0.5,0.1)
    kFreq = linseg:k(500,0.5,400)
    aSine = oscil:a(kAmp,kFreq)
    aOut = linen:a(aSine,0,p3,1)
    outall(aOut)
endin

</CsInstruments>
<CsScore>
i "Hello" 0 2
</CsScore>
</CsoundSynthesizer>

```

Volume Change as Multiplication

How can we apply the signal which we have plotted above, moving as fade-in from 0 to 1, and as fade-out from 1 to 0, to an audio signal?

This is done by multiplication.

We multiply the audio signal with the envelope signal which `linen` generates.

```

aEnv = linen:a(1,0.5,p3,0.5)
aSine = oscil:a(0.2,400)
aOut = aEnv * aSine

```

Actually, the form

```
aOut = linen:a(aSine,0.5,p3,0.5)
```

is a shortcut for this multiplication.

Try it yourself

Change the code so that

- You add a fade-in of 0.5 seconds. (Currently we only have a fade-out.)
- You make the instrument play for 5 seconds, instead of 2 seconds.
- You apply a fade-out for the whole duration of the instrument, regardless what this duration be.
- You apply a fade-in of 1 second, but no fade-out.
- You apply a fade-in of half the instrument's duration, and a fade-out of also half the instrument's duration. (This will result in a triangular envelope shape.)

- You insert **p3** in the *kAmp* line so that the amplitude changes over the whole duration of the instrument.
- You insert **p3** in the *kFreq* line so that the frequency changes over the whole duration of the instrument.

Opcodes and symbols you have learned in this tutorial

Opcodes

- `linen:a(aIn,Fade-in,Duration,Fade-out)` linear fade-in and fade-out

Symbols

- p1 the first parameter of the score line which calls the instrument: number or name of the instrument
- p2 the second parameter of the score line which calls the instrument: the start time of the instrument
- p3 the third parameter of the score line which calls the instrument: the duration of the instrument

Note: p3 refers to the score, but inside the score it has no meaning, and Csound will throw an error if you write **p3** as symbol in the score. It only has a meaning in the **instrument** code.

Go on now ...

with the next tutorial: [05 Hello MIDI Keys](#).

... or read some more explanations here

'linen's amplitude input

The first input for `linen` is the amplitude.

In the most simple case this amplitude is a fixed number. If it is 1, then we have the basic shape which we saw at the beginning of this chapter:



Figure 6.4: linen with amplitude 1

`linen:a(1,0.2,p3,0.5)` will perform a fade-in from 0 to 1 in 0.2 seconds, and a fade-out from 1 to 0 in 0.5 seconds, over the whole duration **p3** of the instrument event.

In the example, we have used the `linen` opcode in directly inserting an audio signal in its first input.

```
linen:a(aSine,0,p3,1)
```

As the first input of `linen` is an amplitude, it can be a constant amplitude, or a signal. This is the case here for `aSine` as amplitude input.

As you already know, in a signal the values change over time, either at *k-rate* or at *a-rate*.

Should we not use a *k*-rate opcode for fades?

Perhaps you wonder why we created an **audio** signal for the envelope signal. Would it not be more efficient to use a **control** signal for the envelope? The code would then read:

```
kFade = linen:k(1,0.5,p3,0.5)
aSine = poscil:a(0.2,400)
aOut = kFade * aSine
```

Indeed this might be slightly more efficient. But for modern computers, this is a neglectable performance gain.

On the other hand, *a-rate* envelopes are preferable because they are really smooth. They have no "staircases" as *k-rate* envelope have.

You can read more [here](#) about this subject. I personally recommend to always use *a-rate* envelopes.

Is it good to have linear fades?

A linear fade is a fade which draws a line for the transition between 0 and 1, or the transition between 1 and 0. This is where the name "linen" comes from: a linear envelope generator.

An alternative is to use curves for fades.

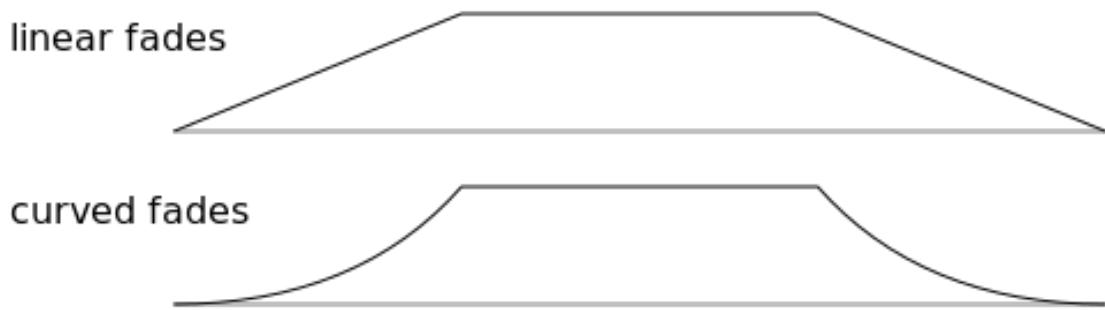


Figure 6.5: Linear versus curved fades

Acoustically linear fades are not the best. We will discuss the reasons in [Tutorial 06](#).

Practically, the linear fades which `linen` generates are sufficient for many cases. But keep in mind that there are other possible shapes for fades, and move to them when you are not satisfied how it sounds. The `transeg` opcode will be your friend then.

05 Hello MIDI Keys

What you learn in this tutorial

- How to work with **MIDI note numbers** instead of raw frequency values.
- How to **convert** MIDI notes **to frequencies**.
- What **i-rate variables** in Csound are.
- How to **print i-rate variables** in the Csound console.

Problems of Using Raw Frequencies

In [Hello Frequency](#) we created a line, or in musical terms, a *glissando* from 500 Hz to 400 Hz:

```
kFreq = linseg:k(500,0.5,400)
```

Actually we have two problems here:

1. When we ask a musician to play a certain pitch, we say “Can you please play D”, or “Can you please play D4”. But what pitch is 500 Hz?
2. When we perform a glissando from 500 Hz to 400 Hz in the way we did, we do it actually not in the linearity we thought we would.

We will discuss the first issue in the next paragraphs. For the second issue, you find an explanation below, in the optional part of this tutorial. You may also have a look at the [Pitches](#) section in the [Digital Audio Basics](#) chapter of this book.

MIDI Note Numbers and the ‘mtof’ Opcode

There are several systems and possibilities to specify pitches like *D4* instead of raw frequencies in Csound.

I recommend using MIDI note numbers, because they are easy to learn and they are used also by other applications and programming languages.

All you must know about MIDI keys or note numbers: *C4* is set to note number 60. And then each semitone, or each next key on a MIDI keyboard, is *plus one* for upwards, and *minus one* for downwards.



Figure 7.1: MIDI Note Numbers

If you want to transform any MIDI key to its related frequency, use the `mtof` (midi to frequency) opcode.

When we want Csound to calculate the frequency which is related to *D4*, and store the result in a variable, we write:

```
iFreq = mtof:i(62)
```

i-rate Variables in Csound

The MIDI key 62 is a number, not a signal. And so is the frequency which relates to this MIDI key. It is a single number, not a signal. This is the reason that we called this variable ***iFreq***.

An **i-rate** variable in Csound is only calculated **once**: when the instrument in which it occurs is initialized. That is where the name comes from.

Remember that **k-rate** and **a-rate** variables are **signals**. A signal varies in time. This means that its values are re-calculated all the time. To summarize:

- **a-rate** variables are signals which are updated every sample.
- **k-rate** variables are signals which are updated every k-cycle. This is less often than the **a-rate**, and determined by the `ksmps` constant.
- **i-rate** variables are no signals but numbers which are only calculated once for each instrument call. They keep this value during the performance of the instrument's note and will not change it.

The 'print' Opcode

Now we would like to know which frequency the *iFreq* variable holds.

For a programming language this means to *print*. Via printing the program shows values in the console. In the console we see the messages from the program.

The print opcode is what we are looking for. Its syntax is simple:

```
print(iVariable)
```

Please run the following code, and look in the console.

```
<CsoundSynthesizer>
<CsOptions>
-o dac
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 64
nchnls = 2
0dbfs = 1

instr Print
    iFreq = mttoi(62)
    print(iFreq)
endin

</CsInstruments>
<CsScore>
i "Print" 0 0
</CsScore>
</CsoundSynthesizer>
```

You should see this message near the end of the console output:

```
instr 1: iFreq = 293.665
```

So we know that the frequency which is related to the MIDI key number 62, or *D4*, is 293.665 Hz.

Note: The print opcode is **for i-rate variables only**. You can **not** use this opcode for printing *k*-rate or *a*-rate variables. We will show opcodes for printing control- or audio-variables later.

Example

We will use now MIDI notes for the glissando, instead of raw frequencies.

We create a line which moves in half a second from MIDI note 72 (C5) to MIDI note 68 (Ab4). This MIDI note line we store in the Variable *kMidi*:

```
kMidi = linseg:k(72,0.5,68)
```

This line is a signal because it changes in time.

Afterwards we convert this line to frequencies by using the *mtof* opcode:

```
kFreq = mttoi:k(kMidi)
```

Note that we use *mtof:k* here because we apply the midi-to-frequency converter to the *k*-rate signal *kMidi*. The result is a *k*-rate signal, too.

```
<CsoundSynthesizer>
<CsOptions>
-o dac
```

```

</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 64
nchnls = 2
0dbfs = 1

instr Hello
    kAmp = linseg:k(0.3,0.5,0.1)
    kMidi = linseg:k(72,0.5,68)
    kFreq = mtot:k(kMidi)
    aSine = oscil:a(kAmp,kFreq)
    aOut = linen:a(aSine,0,p3,1)
    outall(aOut)
endin

</CsInstruments>
<CsScore>
i "Hello" 0 2
</CsScore>
</CsoundSynthesizer>

```

The 'prints' Opcode and Strings

So far we have only dealt with *numbers*.

We have discussed that numbers can only be calculated once, at *i-rate*, or that they are calculated again and again for a block of samples, so being *k-rate*, or even are calculated again and again for each sample, so being *a-rate*.

But all are numbers.

This is somehow natural for an audio application, compared to a text program.

But even in an audio application we need something to write text in, for instance when we point to a sound file as "myfile.wav".

This data type, which starts and ends with double quotes, is called a **string**. (The term is from "character string", so a chain of characters.)

The prints opcode is similar to the print opcode, but it prints a string, and not a number. Try this:

```

<CsoundSynthesizer>
<CsOptions>
-o dac
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 64
nchnls = 2
0dbfs = 1

instr Prints
    prints("Hello String!\n")

```

```

  endin

  </CsInstruments>
  <CsScore>
  i "Prints" 0 0
  </CsScore>
</CsoundSynthesizer>
```

Perhaps you wonder about the \n at the end of the string.

This is a format specifier. It adds a new line.

Please compare: Remove the two characters \n and run the code again. You will see that the console printout is now immediately followed by the next Csound message.

Or add another \n to the existing one, and you will see an empty line after "Hello String!".

In fact, there are much more format specifiers. We will get back to them and to prints in [Tutorial 09](#).

Try it yourself

Change the *kMidi* signal so that

1. the first MIDI key is *E5* rather than *C5*
2. the second MIDI key is *G4* rather than *Ab4*
3. the whole duration of the instrument is used for the *glissando*
4. the glissando goes upwards rather than downwards.

Change also this:

5. Create two variables *iFreqStart* and *iFreqEnd* for the two MIDI notes. (You need to convert the MIDI notes at *i-rate* for it.) Insert then these *i-variables* in the *kFreq = linseg(...)* line. Compare the result to the one in the example.
6. Code a "chromatic scale" (= using always the next MIDI note) which falls from *D5* to *A4*. Each MIDI key stays for one second, and then immediately moves to the next step. You can get this with *linseg* by using zero as duration between two notes. This is the start: *kMidi = linseg:k(74,1,74,0,73,...)*. Don't forget to adjust the overall duration in the score; otherwise you will not hear the series of pitches although you created it ...

Opcodes and terms you have learned in this tutorial

Opcodes

- **mtof:i(MIDI_note)** MIDI-to-frequency converter for one MIDI note
- **mtof:k(kMIDI_notes)** MIDI-to-frequency converter for a *k-rate* signal

- `print(iVariable)` prints *i-rate* variables to the Csound console
- `prints(String)` prints a string to the Csound console

Terms

- *i-rate* is the “time” (as point, not as duration) in which an instrument is initialized
- *i-rate variable* or *i-variable* is a variable which gets a value only once, at the initialization of an instrument.
- A *string* is a sequence of characters, enclosed by double quotes. I’d recommend to only use *ASCII characters* in Csound to avoid problems.

Go on now ...

with the next tutorial: [06 Hello Decibel](#).

... or read some more explanations here

The same is not the same ...

It is worth to look a bit closer to the question of pitch line versus frequency line. We have two possibilities when we create a *glissando* between two MIDI notes:

1. We first create the line between the two MIDI notes. Afterwards we convert this line to the frequencies. This is what we did in the example code:

```
kMidi = linseg:k(72, 0.5, 68)
kFreq = mtotf:k(kMidi)
```

2. We first convert the two MIDI notes to frequencies. Afterwards we create a line. This would be the code:

```
iFreqStart = mtotf:i(72)
iFreqEnd = mtotf:i(68)
kFreq = linseg:k(iFreqStart, 0.5, iFreqEnd)
```

For better comparision, we change the code so that

- we move during 12 seconds instead of 0.5 seconds
- we move two octaves instead of four semitones.

We choose A5 (= 880 Hz or MIDI note 81) and A3 (= 220 Hz or MIDI note 57) as start and end. And we create one variable for each way.

```

kMidiLine_1 = linseg:k(81,12,57)
kFreqLine_1 = mtos:k(kMidi)

iFreqStart = mtos:i(81)
iFreqEnd = mtos:i(57)
kFreqLine_2 = linseg:k(iFreqStart,12,iFreqEnd)

```

The *kFreqLine_1* variable contains the frequency signal which derives from a linear transition in the MIDI note domain. The *kFreqLine_2* variable contains the frequency signal which derives from a linear transition in the frequency domain.

When we use one oscillator for each frequency line, we can listen to both versions at the same time. We will output the *kFreqLine_1* in the left channel, and the *kFreqLine_2* in the right channel.

For more comparison, we also want to look at the MIDI note number version of both signals. For *kFreqLine_1*, this is the *kMidiLine_1* signal which we created. But which MIDI pitches comply with the frequencies of the *kFreqLine_2*?

We can get these pitches via the *ftom* (frequency to midi) opcode. This opcode is the reverse of the *mtos* opcode. For *ftom* we have frequency as input, and get MIDI note numbers as output. So to get the corresponding MIDI pitches to *kFreqLine_2*, we write:

```
kMidiLine_2 = ftom:k(kFreqLine_2)
```

Here comes the code which plays both lines, and prints out MIDI and frequency values of both lines, once a second. Don't worry about the opcodes which you do not know yet. It is mostly about "pretty printing".

```

<CsoundSynthesizer>
<CsOptions>
-o dac
</CsOptions>
<CsInstruments>

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

instr Compare

kMidi = linseg:k(81,12,57)
kFreqLine_1 = mtos:k(kMidi)

iFreqStart = mtos:i(81)
iFreqEnd = mtos:i(57)
kFreqLine_2 = linseg:k(iFreqStart,12,iFreqEnd)
kMidiLine_2 = ftom:k(kFreqLine_2)

prints("Time    Pitches_1   Freqs_1        Freqs_2   Pitches_2\n")
prints("(sec)  (MIDI)     (Hz)          (Hz)     (MIDI)\n")
printks("%2d      %.2f      %.3f      %.3f      %.2f\n", 1,
        timeinsts(), kMidi, kFreqLine_1, kFreqLine_2, kMidiLine_2)

aOut_1 = poscil:a(0.2,kFreqLine_1)
aOut_2 = poscil:a(0.2,kFreqLine_2)
aFadeOut = linen:a(1,0,p3,1)
out(aOut_1*aFadeOut,aOut_2*aFadeOut)

endin

```

```
</CsInstruments>
<CsScore>
i "Compare" 0 13
</CsScore>
</CsoundSynthesizer>
```

In the console, you should see this:

Time (sec)	Pitches_1 (MIDI)	Freqs_1 (Hz)	Freqs_2 (Hz)	Pitches_2 (MIDI)
0	81.00	880.000	880.000	81.00
1	79.00	783.991	825.000	79.88
2	77.00	698.456	770.000	78.69
3	75.00	622.254	715.000	77.41
4	73.00	554.365	660.000	76.02
5	71.00	493.883	605.000	74.51
6	69.00	440.000	550.000	72.86
7	67.00	391.995	495.000	71.04
8	65.00	349.228	440.000	69.00
9	63.00	311.127	385.000	66.69
10	61.00	277.183	330.000	64.02
11	59.00	246.942	275.000	60.86
12	57.00	220.000	220.000	57.00

When we plot the two frequency lines, we see that the first one looks like a concave curve, whilst the second one is a straight line:

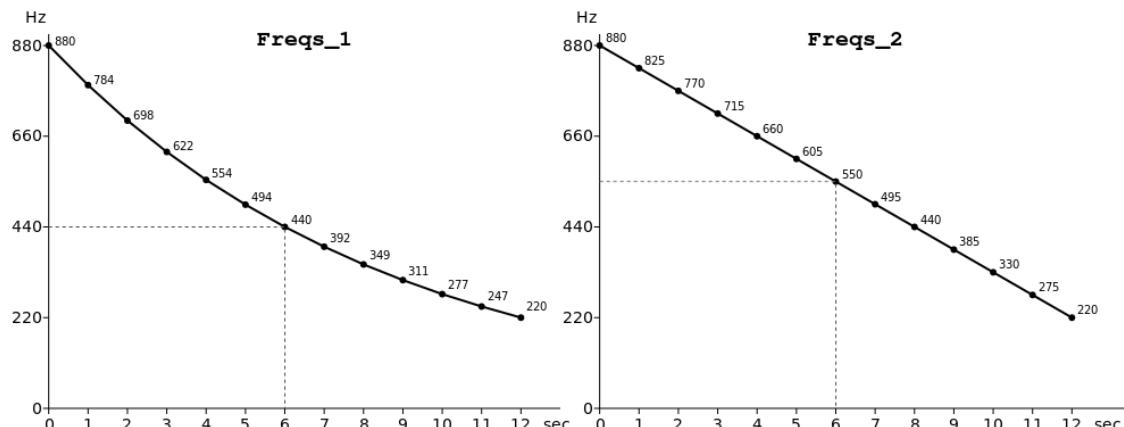


Figure 7.2: Proportional vs. linear frequency transitions

We see that *Freqs_1* reaches 440 Hz after half of the duration, whilst *Freqs_2* reaches 550 Hz after half of the duration.

The *Freqs_1* line **subtracts** 55 Hz in each second of performance.

$$\begin{aligned} 880.000 - 55.000 &= 825.000 \\ 825.000 - 55.000 &= 770.000 \\ \dots \\ 275.000 - 55.000 &= 220.000 \end{aligned}$$

The *Freqs_1* line on the other hand has the same **ratio** between the frequency values of two subsequent seconds:

$$\begin{aligned} 880.000 / 783.991 &= 1.12246\dots \\ 783.991 / 698.456 &= 1.12246\dots \end{aligned}$$

...
246.942 / 220.000 = 1.12246...

Our **perception** follows **ratios**. We will get back to this in the [next tutorial](#).

What we hear is what the *Pitches* printout shows:

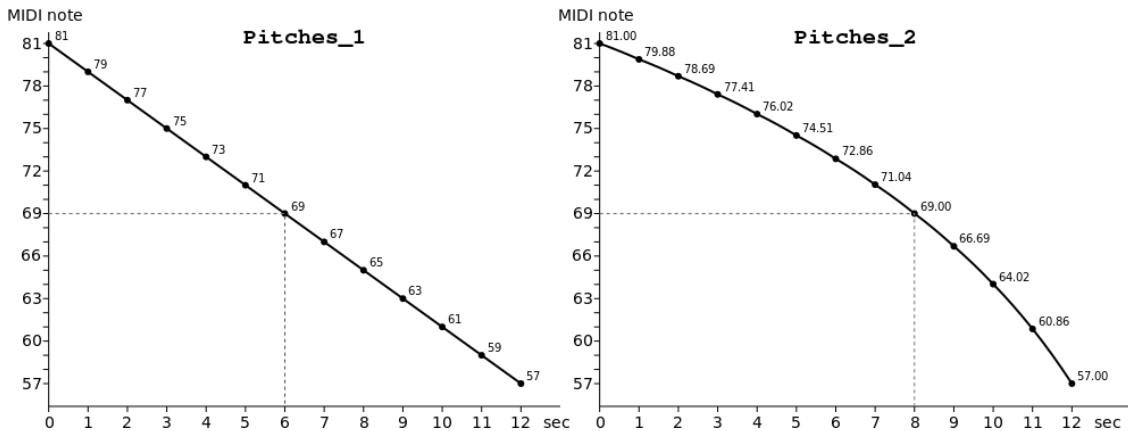


Figure 7.3: Pitch view of proportional vs. linear frequency transitions

We hear that the first line is constantly falling, whilst the second one is too slow at the beginning and too fast at the end.

The dotted lines point to MIDI note 69 which is one octave lower than the note at start. This point is reached after 6 seconds for the *Pitches_1* line. This is correct for a constant pitch decay. We have two octaves which are crossed in 12 seconds, so each octave needs 6 seconds. The *Pitches_2* line, however, reaches MIDI note 69 after 8 seconds, therefore needing 2/3 of the overall time to cross the first octave, and only 1/3 to cross the second octave.

To be written in musical notation, this is how this “too slow” and “too fast” look like:



Figure 7.4: Linear frequency transitions in traditional notation

To adapt the MIDI notes to traditional notation, I have written the cent deviation from the semitones above. The MIDI note number 79.88 for the second note is expressed as A flat minus 12 Cent. If the deviations are larger than 14 Cent, I have added an arrow to the accidentals.

It can be seen clearly how the interval of the first step is only slightly bigger than a semitone, wheras the last step is nearly a major third.

If you enjoy this way of glissando, no problem. Just be conscious that it is not as linear as its frequencies suggest.

Standard pitch for MIDI and equal temperament

Keyboard instruments need one pitch as reference. This pitch is A4, and its frequency is normally set to 440 Hz.

This was not always the case in european music tradition, and as far as I know also in music traditions in other cultures. There was mostly a certain range in which the standard pitch could vary. Even in the 19th century in which scientific standardization prevailed more and more, this process continued. The first international fixation of a standard pitch took place on a conference in Vienna in 1885.

This standard pitch was 435 Hz. But orchestras have a tendency to rise the standard pitch because the sound is more brilliant then. So finally 1939 on the conference of the International Federation of the National Standardizing Association (ISA) in London 440 Hz was fixed. This is valid until today, although most orchestras play a bit higher, in 443 Hz.

Csound offers a nice possibility to change the standard pitch for MIDI. In the orchestra header, you can, for instance, set the standard pitch to 443 Hz via this statement:

```
A4 = 443
```

If you don't set A4, it defaults to 440 Hz.

Once the standard pitch is set, all other pitches are calculated related to it. The tuning system which MIDI uses, is the "equal temperament". This means that from one semitone to the next the frequency ratio is always the same: $\sqrt[12]{2}$ which can also be written as $2^{1/12}$.

So if A4, which is MIDI note number 69, has 440 Hz, note number 70 will have $440 \cdot 2^{1/12}$ Hz. We can use Csound to calculate this:

```
iFreq = 440 * 2^(1/12)
print(iFreq)
```

This prints:

```
iFreq = 466.164
```

And indeed this complies with the result of the `mtof` opcode:

```
iFreq = mtof:i(70)
print(iFreq)
```

Which also prints:

```
iFreq = 466.164
```

"i" in the Score and "i-rate" ...

... have **nothing** to do with each other.

More in general: The score is not Csound code. The score is basically a list of instrument calls, with some simple conventions. No programming language at all.

This is sometimes confusing for beginners. But in modern Csound the score often remains empty. We will show in [Tutorial 07](#) how this works.

06 Hello Decibel

What you learn in this tutorial

- How **human perception** follows **ratios** not only in frequencies but also in **amplitudes**.
- How the **Decibel (dB)** scale works.
- How to **convert Decibel** values to **amplitudes**.
- How to use an **expression as input argument** without creating a variable.

Problems of Using Raw Amplitudes

In the last tutorial, we discussed some issues in working with raw frequencies, and that it is usually preferred to work with MIDI note numbers.

There is a similar issue in working with raw amplitude values. Human perception of both, pitch and volume, follows *ratios*. We hear the frequencies in the left column as octaves because they all have the ratio 2 over 1:

Frequency	Ratio
8000 Hz	> 8000 Hz / 4000 Hz = 2:1
4000 Hz	> 4000 Hz / 2000 Hz = 2:1
2000 Hz	> 2000 Hz / 1000 Hz = 2:1
1000 Hz	> 1000 Hz / 500 Hz = 2:1
500 Hz	> 500 Hz / 250 Hz = 2:1
250 Hz	> 250 Hz / 125 Hz = 2:1
125 Hz	

In the same way, we perceive these amplitudes as having equal loss in volume, because they all follow the same ratio:

Amplitude	Ratio
1	> 1 / 0.5 = 2

```

0.5      > 0.5 / 0.25 = 2
0.25     > 0.25 / 0.125 = 2
0.125    > 0.125 / 0.0625 = 2
0.0625

```

As you see, already after four “intensity octaves” we get to an amplitude smaller than 0.1. But human hearing is capable of about fifteen of these intensity octaves!

Decibel

It is the **Decibel** (dB) scale which reflects this. As you already know from [Tutorial 2](#), we set the amplitude 1 as reference value to zero dB by this statement in the orchestra header:

```
0dbfs = 1
```

Zero dB means here: The highest possible amplitude. Each amplitude ratio of one over two is then a loss of about 6 dB. This yields the following relations between amplitudes and decibels:

Amplitude	dB
1	0
0.5	-6
0.25	-12
0.125	-18
0.063	-24
0.0316	-30
0.01585	-36
0.00794	-42
0.00398	-48
0.001995	-54
0.001	-60

Note 1: To be precise, for an amplitude ratio of 1/2 the difference is -6.0206 dB rather than -6 dB. So the amplitude column is not following precisely the ratio 1/2.

Note 2: You can find more about sound intensities [here](#) in this book.

Note 3: For the general context you may have a look at the [Weber-Fechner law](#).

The ‘ampdb’ Opcode

So we will usually like to work with Decibel rather than with raw amplitudes. As the oscillator requires an amplitude as input argument, we must **convert decibel to amplitudes**. This is done via the ampdb (amplitude from decibel) opcode. Run this, and have a look at the console output:

```

<CsoundSynthesizer>
<CsOptions>
-o dac
</CsOptions>
<CsInstruments>

```

```

sr = 44100
ksmps = 64
nchnls = 2
0dbfs = 1

instr Convert
    iAmp = ampdb:i(-6)
    print(iAmp)
endin

</CsInstruments>
<CsScore>
i "Convert" 0 0
</CsScore>
</CsoundSynthesizer>

```

You should see this message:

```
instr 1:  iAmp = 0.501
```

Similar to `mt of`, the `ampdb` opcode can run at *i-rate* or at *k-rate*. Here we use *i-rate*, so `ampdb : i` because we have a number as input, and not a signal.

We will use `ampdb : k` when we have time varying decibel values as input. In the most simple case, it is a linear rise or decay. We can create this input signal as usual with the `linseg` opcode. Here is a signal which moves from -10 dB to -20 dB in half a second:

```
kDb = linseg:k(-10,0.5,-20)
```

And this is the following conversion to amplitudes:

```
kAmp = ampdb:k(kDb)
```

Inserting an Expression as Input Argument

So far we have always stored the output of an opcode in a variable; The output of an opcode gets a name, and this name is then used as input for the next opcode in the chain. We have currently four chain links. These chain links are written as numbers right hand side in the next figure.

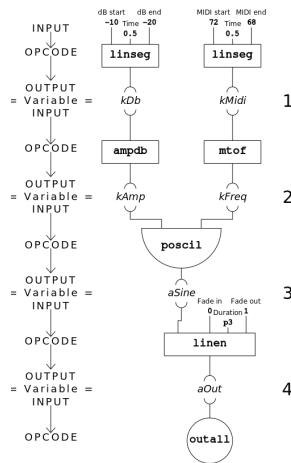


Figure 8.1: Input-output-chain with variables

It is possible to omit the variable names and to directly pass one expression as input argument into the next chain link. This is the code to skip the variable names for chain link 2:

```
aSine = poscil:a(ampdb:k(kDb),mtof:k(kMidi))
```

And the figure can be drawn like this:

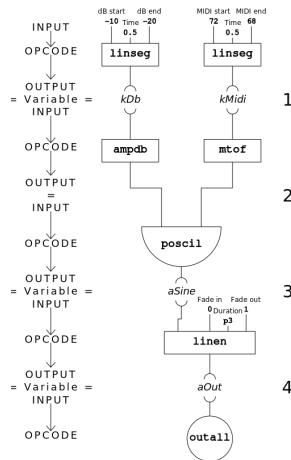


Figure 8.2: Direct connection at chain link 2

Example

This version is used in the example code.

```
<CsoundSynthesizer>
<CsOptions>
-o dac
</CsOptions>
</CsInstruments>

sr = 44100
ksmps = 64
nchnls = 2
```

```

0dbfs = 1

instr Hello
  kDb = linseg:k(-10,0.5,-20)
  kMidi = linseg:k(72,0.5,68)
  aSine = oscil:a(ampdb:k(kDb),mtof:k(kMidi))
  aOut = linen:a(aSine,0,p3,1)
  outall(aOut)
endin

</CsInstruments>
<CsScore>
i "Hello" 0 2
</CsScore>
</CsoundSynthesizer>
```

Short or readable?

The possibility of directly inserting the output of one opcode in another is potentially endless. It leads to shorter code.

On the other hand, if many of these expressions are put inside each other, the code can become a stony desert of (, :, , and)).

Variables are actually not a necessary evil. They can be a big help in understanding what happens in the code, if they carry a meaningful name. This is what I tried as setting *kMidi*, *aSine*, ... Perhaps you will find better names; that is what we should try.

In case we insert all four chain links into each other, the code would look like this:

```
outall(linen:a(poscil:a(ampdb:k(linseg:k(-10,.5,-20),...)
```

I don't think this makes sense; in particular not for beginners. In the Csound FLOSS Manual we have a hard limit because of layout: No code line can be longer than 78 characters. I think this is a good orientation.

The most valuable qualities of code are clarity and readability. If direct insertion of an expression helps for it, then use it. But better to avoid code abbreviations which more derive from laziness than from decision.

You yourself are the best judge. Read your code again after one week, and then make up your mind

...

Try it yourself

- Create a *crescendo* (a rise in volume) rather than a *diminuendo* (a decay in volume) as we did.
- Change the values so that the *crescendo* becomes more extreme.
- Again change to *diminuendo* but also more extreme than in the example.

- Change the code so that you create the variable names `kAmp` and `kFreq` first, as shown in the first figure.
- Play with omitting the variable names also in chain link 1, 3 or 4. Which version do you like most?

Opcodes you have learned in this tutorial

- `ampdb:i(iDecibel)` converts a Decibel number to an amplitude number
- `ampdb:k(kDecibel)` converts a Decibal signal to an amplitude signal

Go on now ...

with the next tutorial: [07 Hello p-Fields](#).

... or read some more explanations here

What is 0 dB?

We have stressed some similarities between working with pitch and with volumes. The similarities derive from the fact that we perceive in a proportional way by our senses. Both, the MIDI scale for pitch and the Decibel scale for volume, reflect this.

But there is one big difference between both. The MIDI scale is an **absolute** scale. MIDI note number 69 **is** 440 Hz. (Or slightly more cautious: MIDI note number 69 is set to the standard pitch which is usually 440 Hz.)

But the Decibel scale is a **relative** scale. It does not mean anything to say "this is -6 dB" unless we have **set** something as **0 dB**.

In **acoustics**, 0 dB is set to a very small value. To put it in a non-scientific way: The softest sound which we can hear.

This means that all common Decibel values are then **positive**, because they are (much) more than this minimum. For instance around 60 dB for a normal conversation.

But as explained above, in **digital audio** it is the other way round. Here our **0 dB** setting points to the **maximum**, to the highest possible amplitude.

In digital audio, we have a certain number of bits for each sample: 16 bit, 24 bit, 32 bit. Whatever it be, there is a maximum. Imagine a 16 bit digital number in which each bit can either be 0 or 1. The maximum possible amplitude is when all bits are 1.

And so for the other resolutions. (They do not add something on top. They offer a finer resolution between the maximum and the minimum.)

So it makes perfect sense to set this maximum possible amplitude as 0 dB. But this means that in digital audio we only have **negative** dB values.

Can I use positive dB values?

There is one exception. Yes it is true that there can be no amplitude larger than 0 dB. But we can use positive dB in digital audio when we **amplify** soft sounds.

As we saw in [Tutorial 04](#), to amplify a signal means to multiply it by a value larger than 1. It makes perfect sense to express this in decibel rather than as a multiplier.

We can say: "I amplify this signal by 6 dB", rather than: "I amplify it by factor 2." And: "I amplify this signal by 12 dB" should be better than "I amplify it by factor 4".

In this context we will use a positive number as input for the ampdb opcode. Here is a simple example:

```
<CsoundSynthesizer>
<CsOptions>
-o dac
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 64
nchnls = 2
0dbfs = 1

instr Amplify
    //get dB value from fourth score parameter
    iDb = p4
    //create a very soft pink noise
    aNoise = pinkish(0.01)
    //amplify
    aOut = aNoise * ampdb(iDb)
    outall(aOut)
endin

</CsInstruments>
<CsScore>
i "Amplify" 0 2 0 //soft sound as it is
i "Amplify" 2 2 10 //amplification by 10 dB
i "Amplify" 4 2 20 //amplification by 20 dB
</CsScore>
</CsoundSynthesizer>
```

A Look at Arithmetic and Geometric series

It is often quite interesting to look back from electronic music to some very old traditions which are inherited in it. I'd like to undertake an excursion to one of these traditions here. Just jump to the next tutorial if you are not interested in it.

What we discussed in this and the previous tutorial about *linear* versus *proportional* transitions in both, frequency and amplitude, has been described by ancient greek mathematicians as *arithmetic* versus *geometric* series.

If we have two numbers, or lengths, and look after the one “in between”, the **arithmetic** mean searches for the equal **distance** between the smaller and the larger one. Or in the words of Archytas of Tarentum (early 4th century B.C.):

The arithmetic mean is when there are three terms showing successively the same excess: the second exceeds the third by the same amount as the first exceeds the second. In this proportion, the ratio of the larger numbers is less, that of the smaller numbers greater.¹

If the first number is 8, and the third number is 2, we look for the second number as arithmetic mean A and the “excess” x like this:

$$A = x + 28 = x + A2 + x + x = 8x + x = 6x = 3$$

So the arithmetic mean A is 5, as $2 + 3 = 5$ and $5 + 3 = 8$.

But as Archytas states, “the ratio of the larger numbers is less, that of the smaller numbers greater”. Here:

$$8/5 = 1.65/2 = 2.5$$

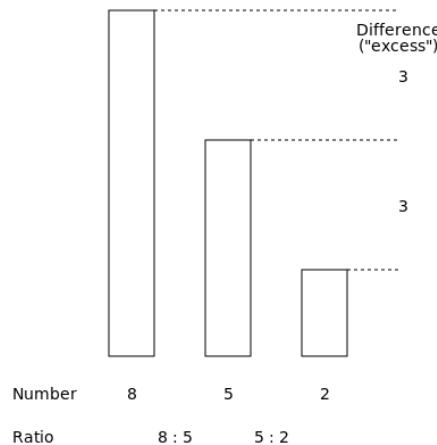


Figure 8.3: Arithmetic mean of 8 and 2

This is what we described as “at first too slow then too fast” in the [previous tutorial](#).

Here comes how Archytas describes the **geometric** mean:

The geometric mean is when the second is to the third as the first is to the second; in this, the greater numbers have the same ratio as the smaller numbers.

¹Ancilla to the Pre-Socratic Philosophers. A complete translation of the Fragments in Diels, Fragmente der Vorsokratiker by Kathleen Freeman. Cambridge, Massachusetts: Harvard University Press [1948], quoted after <http://demonax.info/doku.php?id=text:archytas.fragments>

So if we look for the geometric mean G between 8 and 2 we calculate:

$$8/G = G/28 = G \cdot G/216 = G^2G = 4$$

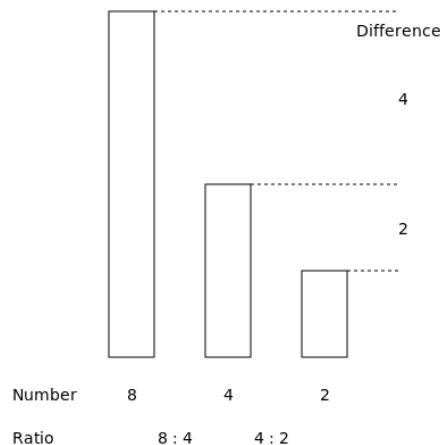


Figure 8.4: Geometric mean of 8 and 2

The geometric mean of 8 and 2 is 4 because the ratio of the larger number to the mean, and the ratio of the mean to the smaller number is the same: $8/4 = 2$ and $4/2 = 2$.

It is quite interesting to look at the “geometric” way to construct this mean which is shown in Euclid’s *Elements* (VI.8):

If, in a right-angled triangle, a (straight-line) is drawn from the right-angle perpendicular to the base then the triangles around the perpendicular are similar to the whole (triangle), and to one another.
²

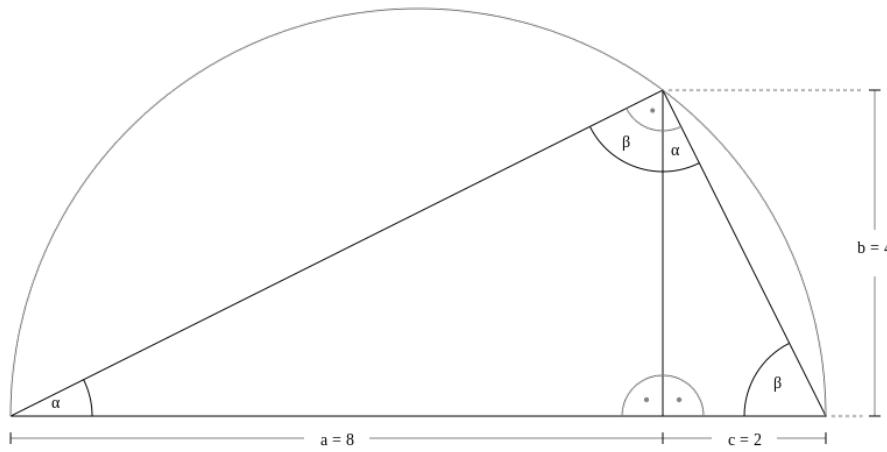


Figure 8.5: Euclid's construction of the geometric mean

Euclid describes how the two triangles which are left and right to this perpendicular have the same angles, and that this is also the case when we look at the large triangle. It establishes the perfect similarity.

²Euclid's Elements of Geometry, Translation R. Fitzpatrick, p. 164, cited after <https://farside.ph.utexas.edu/Books/Euclid/Elements.pdf>

The length of this perpendicular is the geometric mean of the two parts on the base. According to the right triangle altitude theorem, the square of this altitude equals the product of the base parts:

$$b^2 = a \cdot c b = \sqrt{a \cdot c}$$

Exactly this is the formula for the geometric mean.

There is also a close relationship to the **golden ratio** which is famous for its usage in art and nature. In terms of the triangle which Euclid describes it means: Find a triangle so that the smaller base part plus the height equal the larger base part:

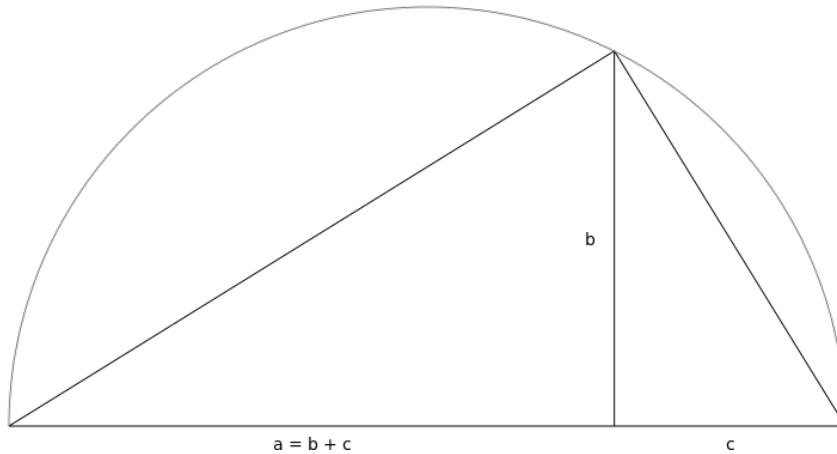


Figure 8.6: Golden ratio as geometric mean

Currently we have $a = 8$, $c = 2$ and $b = 4$. Obviously, $8 = 4 + 2$ is not true.

We can move the separation point a bit to the left, so that $a = 7$ and $c = 3$. This yields $7 = \sqrt{21} + 3$ which is also not true.

Bad luck: The golden ratio can be easily [constructed geometrically](#), but it is an irrational number. We are close though when we choose higher [Fibonacci numbers](#). For instance, for the Fibonacci numbers $b = 88$ and $c = 55$ it yields: $a = b \cdot b/c = 88 \cdot 88/55 = 140.8$ rather than the desired 143.

Good enough for music perhaps which always needs some dirt for its life ...

07 Hello p-Fields

What you learn in this tutorial

- What **instrument instances** are.
- How to **open instruments** for input parameters from the score.
- How to work with **flexible input parameters** inside an instrument.
- How to write **comments** in Csound code.

Instruments are Models which are Instantiated

We have applied until now a lot of useful capacities to our instrument, as sliding pitch and volume, and an automated fade-in and fade-out.

But this all happens **inside** the instrument. Whenever we call our *Hello* instrument, it will play the same pitches in the same volume. All we can decide currently from outside the instrument is to adjust **when** the instrument shall **start**, and **how long** it will play. As you know from [Tutorial 04](#), these informations are submitted to the instrument via parameters in the score:

- The first parameter, abbreviated **p1**, ist the number or name of the instrument which is called.
- The second parameter, abbreviated **p2**, is the start time of this instrument.
- The third parameter, abbreviated **p3**, is the duration of this instrument.

Let us call the instrument *Hello* with these score lines:

```
i "Hello" 0 2  
i "Hello" 3 3  
i "Hello" 9 1
```

What we actually do is to **instantiate** a certain instrument. Each instance is a running object of the instrument model; the concrete “thing” which is there as realization of the model.

In this case, we create and call three instances of the “Hello” instrument which follow each other with some pauses between them:

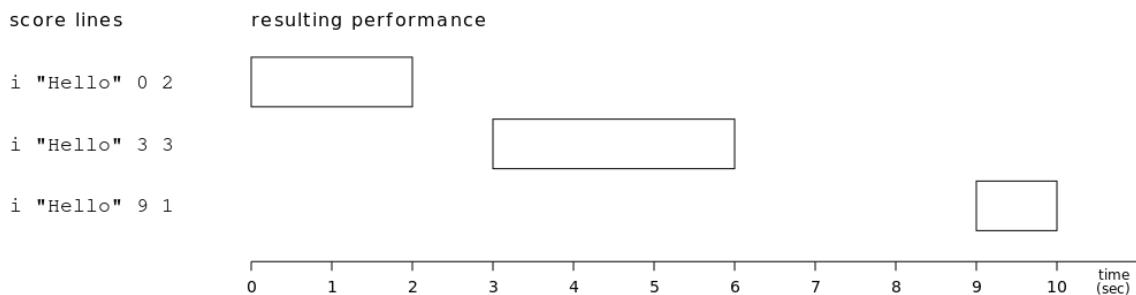


Figure 9.1: Three instrument instances in a sequence

We can create as many instances of an instrument as we like. They can follow in time, as above, or can overlap, as for these score lines:

```
i "Hello" 0 7
i "Hello" 3 6
i "Hello" 5 1
```

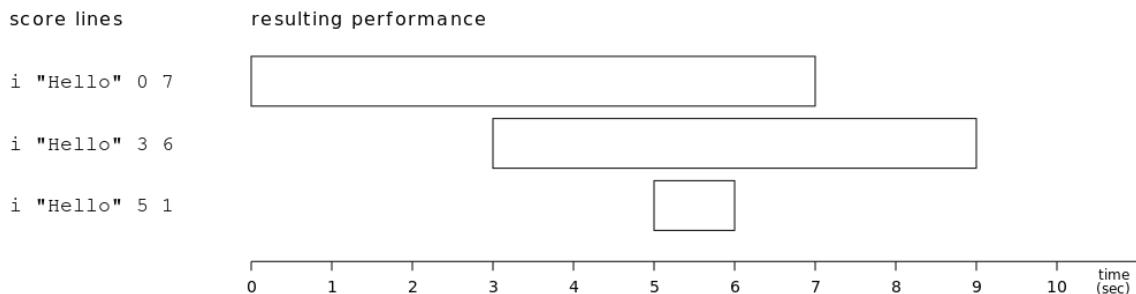


Figure 9.2: Three instrument instances overlapping each other

Make instruments flexible

But back to the characteristics of our instrument, as it is for now. We need it to be more open to the world outside itself. For example, we want to decide in the call of the instrument instance, which MIDI note to play at the beginning, and which MIDI note to play at the end of the duration.

To achieve this, we **add more p-fields** to our score line. We write the first MIDI note number as fourth score parameter **p4**, and the second MIDI number as fifth score parameter **p5**:

```
i "Hello" 0 2 72 68
```

Which means we call:

- **p1** the instrument "Hello"
- **p2** the start time of this instance to be zero
- **p3** a duration of two seconds
- **p4** the MIDI note number 72 at the beginning, and
- **p5** the MIDI note number 68 at the end of the duration.

To make this happen, we insert **p4** and **p5** in the instrument code.

```

instr Hello
  iMidiStart = p4
  iMidiEnd = p5
  kMidi = linseg:k(iMidiStart,p3,iMidiEnd)
  ...
endin

```

The instrument interpretes these values in the same way as it was for **p3** which we already used in our code. For **p4** and **p5**, the instrument instance will look at the score line, and take the fourth parameter as value for **p4**, and the fifth parameter as value for **p5**.

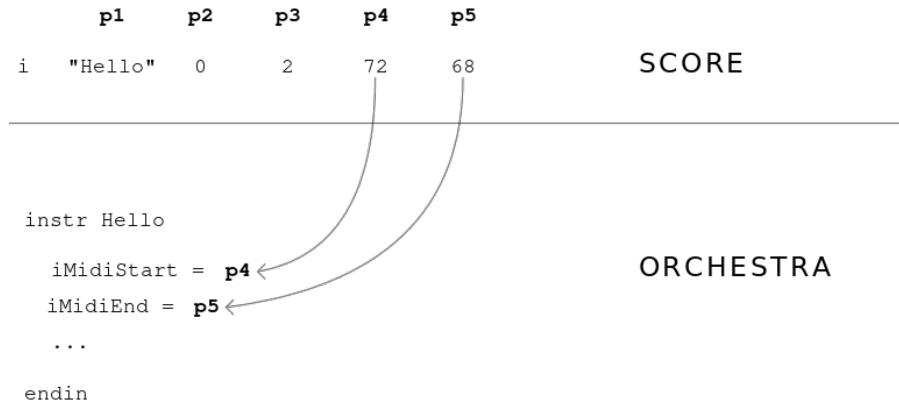


Figure 9.3: Instrument reading p4 and p5 of a sore line

Direct Insertion of p-fields or Variable Definition

We have used **p4** and **p5** in the code above to define the *i*-rate variables, *iMidiStart* and *iMidiEnd*. It would have been perfectly fine, from Csound's side, to set **p4** and **p5** directly in the code, like this:

```
kMidi = linseg:k(p4,p3,p5)
```

I say "from Csound's side" because I think for the readability of code it is better to tell on top of the instrument code, what **p4** and **p5** mean, and connect them with a variable name.

This variable will be *i*-rate variable because the score can only pass fixed values to the instrument.

The variable name should be as meaningful as possible, without becoming too long. Most important, again, is the readability of the code.

Comments in the code

This readability can also be improved by comments; in particular for large code.

Csound offers three possibilities for comments.

- // and ; are the comment signs for one line.

- What is between /* and */ will also be ignored by Csound. It can comprise more than one line.

I suggest that you comment extensively; in particular when you start learning Csound. It will help you to understand what is happening in the code, and it will help you to understand your own code later.

I often even start with comments when I code. Then the comments are there to make clear what I want to do. For instance:

```
instr Dontknowyet
    //generate two random numbers

    //calculate their distance

    //if the distance is larger than x ...
    //... insert some silence ...

    //... otherwise make a lot of crazy noise
endin
```

Example

We will now insert comments in the code. At first extensively; in later tutorials we will reduce it and focus on the new parts of the code.

```
<CsoundSynthesizer>
<CsOptions>
-o dac
</CsOptions>
<CsInstruments>

/* CONSTANTS IN THE INSTRUMENT HEADER*/
sr = 44100
ksmps = 64
nchnls = 2
0dbfs = 1

/* INSTRUMENT CODE */
instr Hello ;Hello is written here without double quotes!
//receive MIDI note at start and at end from the score
iMidiStart = p4 ;this is a MIDI note number
iMidiEnd = p5
//create a glissando over the whole duration (p3)
kMidi = linseg:k(iMidiStart,p3,iMidiEnd)
//create a decay from -10 dB to -20 dB in half of the duration (p3 / 2)
kDb = linseg:k(-10,p3/2,-20)
//sine tone with ampdb and mtof to convert the input to amp and freq
aSine = oscil:a(ampdb:k(kDb),mtof:k(kMidi))
//apply one second of fade out
aOut = linen:a(aSine,0,p3,1)
//output to all channels
outall(aOut)
endin

</CsInstruments>
```

```

<CsScore>
/* SCORE LINES*/
//score parameter fields
//p1    p2 p3    p4 p5
i "Hello" 0 2    72 68 ;here we need "Hello" with double quotes!
i "Hello" 4 3    67 73
i "Hello" 9 5    74 66
i "Hello" 11 .5  72 73
i "Hello" 12.5 .5 73 73.5
</CsScore>
</CsoundSynthesizer>

```

Looking Back to History

"P fields" are an element in which Csound's long history reflects. This is a figure from the 1962 article "Musical Sounds from Digital Computers" by Mathews, Pierce and Guttman, showing a "Computer Card":¹

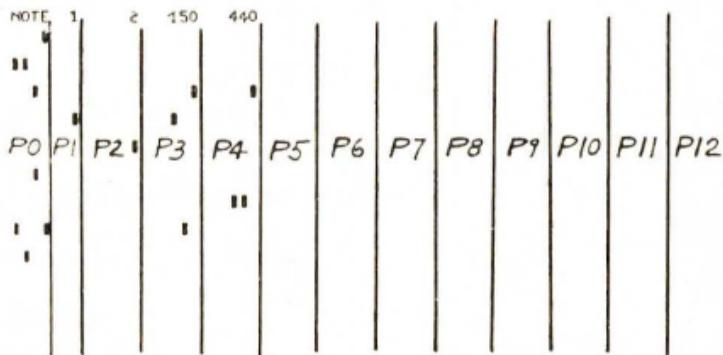


Fig 1 Computer Card

Figure 9.4: p-fields on a computer card in Mathews et.al. 1962

Here the *P0* field contains the information about either "note" or "pause". *P1* is the instrument number. As there is no polyphony here, *P2* is the duration of a chain of events. *P3* is an amplitude here in the range of 0 through 1000, and *P4* is the frequency.

So p-fields were there before computers had a screen and a keyboard ...

It was Mathews' *MUSIC V* which Jean-Claude Risset used to write his epochal "Catalogue of Computer Synthesized Sounds" in 1969. His code can with only slight modifications be transformed into Csound code. Here is a snippet in which you can see p-fields again:²

¹Gravesaner Blätter (Ed. Hermann Scherchen) VI, Heft 23-24 (1962), p. 115, online: <https://soundandscience.de/text/gravesaner-blatter-jahrgang-vi-heft-23-24>

²Jean-Claude Risset, An Introductory Catalogue of Computer Synthesized Sounds, Bell Telephone Laboratories, Murray Hill, New Jersey, 1969, p. 56; online: <https://ia801707.us.archive.org/13/items/an-introductory-catalogue-of-computer-synthesized-sounds/An-Introductory-Catalogue-of-Computer-Synthesized-Sounds.pdf>

#300

This run compares different decays.

Instrument #1 is diagrammed here.

All 4 instruments are similar, they only differ by the function numbers. (For simultaneous voices, one cannot use the same instrument with different functions; for successive notes one could redefine the function or use SET unit generator.)

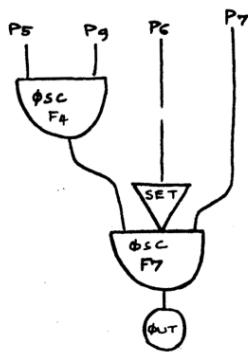


Figure 9.5: Excerpt from Rissets "Catalogue" 1969

The early versions of Max Mathews *MUSIC* program could only run on one particular computer. The “C” in Csound points to the [C Programming Language](#) which was first released in 1972. It made it possible to separate the source code which is written and can be read by humans from a specific machine on which it runs. C is still a very successful language, used for everything which must be fast, like operating systems or audio applications.

P fields are on one hand simple and give a lot of possibilities. There are, on the other hand, restrictions. Basically, a *p-field* carries a number. It took a lot of work from the Csound developers to make it possible to write a string in a *p-field*, too. But it is still not possible to pass a signal via a *p-field* to an instrument.

Fortunately, *p-fields* are only one possibility for an instrument to communicate with the “outer world”. We will discuss other ways later in these Tutorials.

Try it yourself

- Change the values in the score in a way that all directions of the pitch slides are reversed (upwards instead of downwards and vice versa).
- Change the values in the score so that you have not any more a sliding pitch but a constant one.
- Add two p-fields in the score to specify the first and the last volume as *dB*. Refer to these p-fields as **p6** and **p7** in the instrument code. Introduce two new variables and call them *iDbStart* and *iDbEnd*.
- Change the code so that the volume change uses the whole duration of the instrument instance whilst the pitch change only uses half of the instrument’s duration.
- Go back to the first code by reloading the page. Now remove the fifth p-field from the score and change the code in the instrument so that the *iMidiEnd* variable is always 6 MIDI notes lower than *iMidiStart*.
- Introduce **p5** again, but now with a different meaning: 1 in this p-field means that the *iMidiEnd* note will be six MIDI keys higher than *iMidiStart*; -1 will mean that *iMidiEnd* will be six MIDI keys lower than *iMidiStart*.

- Add a p-field which establishes the fade-out duration as ratio to the whole duration. (1 would mean: fade out equals the overall instrument duration; 0.5 would mean: fade out time is half of the instrument duration.)

Terms and symbols you have learned in this tutorial

Terms

- *instance* is the manifestation or realization of an instrument when it is called by a score line

Symbols

- **p4, p5** ... are used in a Csound instrument code to refer to the fourth, fifth ... parameter in the score line which called the instrument instance

Go on now ...

with the next tutorial: [08 Hello schedule.](#)

... or read some more explanations here

MIDI notes and microtonal deviations

You may have noticed that in the last score line of the example this was written:

```
i "Hello" 12.5 .5 73 73.5
```

But what is MIDI note number 73.5? There is obviously no key with this number. There is only key number 73 (C#5) and 74 (D5).

This is true but the conversion from MIDI note numbers to frequencies does not only work for integer MIDI note numbers. It is possible to specify any fraction of the semitone which is between two integer MIDI note numbers. We can split a semitone in two quartertones. This is what we did by referring to MIDI note number 73.5: A quarter tone higher than C#5, or a quarter tone lower than D5.

In the same way we can express any other fraction. Most common is to divide one semitone in one hundred cents. So MIDI note number 60.14 would be C4 plus 14 Cent. Or 68.67 would be A4 minus 33 Cents.

Enumerate instrument instances as fractional part

It is not only possible to call as many instances of an instrument as we like, we can also give each of these instances a certain number. This is done by calling the instrument not as an integer number, but as this integer number plus a fractional part.

In his case, we will not write these score lines ...

```
i 1 0 3  
i 1 2 2  
i 1 5 1
```

... but we will write this score line instead:

```
i 1.1 0 3  
i 1.2 2 2  
i 1.3 5 1
```

In this case, we have called instrument 1.1 as first instance of instrument 1, and 1.2 as its second instance, and 1.3 as its third instance.

The instrument instance gets this information, as we see in this simple example:

```
<CsoundSynthesizer>  
<CsOptions>  
-o dac -m 128  
</CsOptions>  
<CsInstruments>  
sr = 44100  
ksmps = 32  
nchnls = 2  
0dbfs = 1  
  
instr 1  
    print(p1)  
endin  
  
</CsInstruments>  
<CsScore>  
i 1.1 0 3  
i 1.2 2 2  
i 1.3 5 1  
</CsScore>  
</CsoundSynthesizer>
```

You should see this console output:

```
instr 1: p1 = 1.100  
instr 1: p1 = 1.200  
instr 1: p1 = 1.300
```

So the instance which we called as 1.1 "knows" that it is instance 1.1, and so do the other instances. Imagine we want to send a specific message which is broadcasted on a certain software channel

called “radio” only to instance 1.2, then we would write something like: “If my instance is 1.2 then I will receive the message from the radio channel.”

This is the Csound code for it, without explaining about the opcodes you do not know yet. But perhaps you can follow the logic, and please have a look in the console printout.

```
<CsoundSynthesizer>
<CsOptions>
-odac -m128
</CsOptions>
<CsInstruments>
sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

//put the message string "Hello ..." in the channel "radio"
chnset("Hello instrument 1.2!","radio")

instr 1
iMyInstance = p1 ;get instance as 1.1, 1.2 or 1.3
print(iMyInstance) ;print it
//receive the message only if instance is 1.2
if iMyInstance == 1.2 then
    Smessage = chnget:S("radio")
    prints("%s\n",Smessage)
endif
endin

</CsInstruments>
<CsScore>
i 1.1 0 3
i 1.2 2 2
i 1.3 5 1
</CsScore>
</CsoundSynthesizer>
```

You should see this in the conlole printout:

```
instr 1: iMyInstance = 1.100
instr 1: iMyInstance = 1.200
Hello instrument 1.2!
instr 1: iMyInstance = 1.300
```

Instrument names and fractional parts

We can give a name or a number to an instrument. I personally prefer names because the names describes what the instrument does. For instance

```
instr LiveInput
instr RecordToDisk
instr AnalyzeRMS
```

This again improves the readability of the code.

But we cannot call an instrument with a fractional number if we call the instrument by its name. In the score line ...

```
i "LiveInput" 0 1000
```

... the first *p-field* "LiveInput" is a string, not a number. And we cannot extend a string with .1 as we can for the numbers. This will not work:

```
i "LiveInput".1 0 1000
```

We have two options here. The first option uses the fact that Csound internally converts each instrument name to a number. The way Csound assigns numbers to the named instruments is simple: The instrument which is on top gets number 1, the next one gets number 2, and so on.

When we only have one instrument, we can be sure that this is instrument 1 for Csound. And therefore this code works without problems:

```
<CsoundSynthesizer>
<CsOptions>
-odac -m128
</CsOptions>
<CsInstruments>
sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

//put the message string "Hello ..." in the channel "radio"
chnset("Hello instrument 1.2!","radio")

instr TestInstanceSelection
iMyInstance = p1 ;get instance as 1.1, 1.2 or 1.3
print(iMyInstance) ;print it
//receive the message only if instance is 1.2
if iMyInstance == 1.2 then
    Smassage = chnget:S("radio")
    prints("%s\n",Smassage)
endif
endin

</CsInstruments>
<CsScore>
i 1.1 0 3
i 1.2 2 2
i 1.3 5 1
</CsScore>
</CsoundSynthesizer>
```

We have an instrument with the name "TestInstanceSelection" here, but we call it in the score lines as 1.1, 1.2 and 1.3. No problem.

The other way to work with named instruments and fractional parts is to move the instrument call from the score to the actual Csound code. This is the subject of the next Tutorial in which we will introduce the *schedule* opcode.

08 Hello Schedule

What you learn in this tutorial

- How to call instruments **without** using **the score**
- How to get the **number** of a **named instrument**
- How to read the **Csound Reference Manual**

The .csd File Again and a Bit of Csound History

You learned in [Tutorial 02](#) that the Csound .csd file consists of three sections:

- The CsOptions section sets some general options which are used once you pressed the “Play” button. For instance, whether you want to render Csound in real-time to the sound card, or to write a sound file. Or which MIDI device to use, or how many messages to display in the console.
- The CsInstruments section contains the actual Csound code. It is also called the “Orchestra” code because it comprises the Csound instruments.
- The CsScore section collects score lines. Each score line which begins with **i** initiates an instrument instance, at a certain time, for a certain duration, and possibly with additional parameters.

On one hand, this is a reasonable division of jobs. In the CsInstruments we code the instruments, in the CsScore we call them, and in the CsOptions we determine some general settings about the performance. In early times of Csound, the instrument definitions and the score resided in two separated text files. The instrument definitions, or orchestra, were collected in an .orc file, and the score lines were collected in a .sco file. The options were put in so called “flags” which you still see in the CsOptions. The **-o** flag, for example, assigns the audio output.

It is still possible to use Csound in this way from the “command line”, or “Terminal”. A Csound run would be started by a command line like this one:

```
csound -o dac -m 128 my_instruments.orc my_score_lines.sco
```

The call to the Csound executable is done with typing **csound** at first position. In second position we can add as many options as we like. In this case: **-o dac -m 128** for real-time output and

reduced message level. At third and fourth position then follow the .orc and .sco file.

This command line way of running Csound is still a very versatile option. If you enjoy learning Csound, and continue using it for your own audio projects, you will probably at any time in future use it, because it is fast and stable, and you can integrate Csound in any scripting environment.

Using Csound Without the Score Section

We came across this retrospective because the separation between an “orchestra” and a “score” file in early Csound clearly shows the different role of both.

It is not only a different role. It can be said that an instrument knows a bit about the score. As you learned in [Tutorial 04](#) and [Tutorial 07](#), each instrument instance knows about the *p-fields* which created this instance. It knows about its duration, it even knows its start time in the overall Csound performance, and if called by a fractional number, it knows its instance as a unique number.

But the score knows nothing about the instruments. Not even what the sample rate is, or how many channels we set in the orchestra header via `nchnls = 2`. We cannot use any opcode in the score, nor any variable name. The score does not understand the orchestra, and to put it in this way: The score does not understand Csound language.

There are several situations, however, in which we need **one** language to do all. We may want to start instruments from inside another instrument. We may want to pass a variable to an instrument which we calculate during the performance. We may want to trigger instrument instances from live input, like touchscreens, MIDI keyboards or messages from other applications.

Csound offers this flexibility. The most used opcodes for calling instruments from inside the `CsInstruments` section, are `schedule` and `schedulek`. We will introduce at first `schedule`, and then in [Tutorial 11](#) come to `schedulek`.

So from now on you will see the score section mostly empty. But of course there are still many situations in which a traditional score can be used. You will find some hints for score usage in the optional section of this tutorial.

The ‘`schedule`’ Opcode

The `schedule` opcode has the same function as an `i` score line. It calls an instance of a defined instrument. It has at least three input arguments, because each instrument call needs to contain

1. number or name of the called instrument
2. start time of the called instrument
3. duration of the called instrument.

This code calls instrument “Hello” at start time zero for two seconds:

```
schedule("Hello", 0, 2)
```

As schedule is an opcode, its input arguments are **separated by commas**. This is the main difference to the score in which the parameter fields are separated by spaces:

```
i "Hello" 0 2
```

Note 1: As you see the starting **i** in the score is omitted. This is possible because schedule only instantiates *instrument* events whilst a score line can also have other statements. Have a look below in the optional part of this tutorial if you want to know more about it.

Note 2: It is up to you whether you add spaces to the commas in the schedule argument list, or not. For Csound, the commas are the separator. Once you accept this, you can use spaces or tabs in addition, or not. So these two lines are both valid:

```
//arguments are separated by commas only
schedule("Hello",0,2)
//arguments are separated by commas followed by spaces
schedule("Hello", 0, 2)
```

Example

The following example simply transfers the score lines from [Tutorial 07](#) to schedule statements. So it will sound exactly the same as the example in Tutorial 07.

```
<CsoundSynthesizer>
<CsOptions>
-o dac
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 64
nchnls = 2
0dbfs = 1

//same instrument code as in tutorial 07
instr Hello
    iMidiStart = p4
    iMidiEnd = p5
    kDb = linseg:k(-10,p3/2,-20)
    kMidi = linseg:k(iMidiStart,p3,iMidiEnd)
    aSine = oscil:a(ampdb:k(kDb),mtof:k(kMidi))
    aOut = linen:a(aSine,0,p3,1)
    outall(aOut)
endin
//followed by schedule statements rather than score lines
schedule("Hello", 0, 2, 72, 68)
schedule("Hello", 4, 3, 67, 73)
schedule("Hello", 9, 5, 74, 66)
schedule("Hello", 11, .5, 72, 73)
schedule("Hello", 12.5, .5, 73, 73.5)

</CsInstruments>
<CsScore>
//the score is empty here!
</CsScore>
</CsoundSynthesizer>
```

Try it yourself

Make sure that you always stop and restart Csound when you move to the next exercise item.

- You can put the schedule lines anywhere in the `CsInstruments` section. Just make sure it is **outside** the “Hello” instrument. Try putting the lines anywhere by copy-and-paste. You can also scatter them anywhere, and in any order. This will probably not improve the readability of your code, but for Csound it will make no difference.
- Put one of the schedule lines in the score. Csound will report an error, because the score does not understand Csound orchestra code.
- Put **all** schedule lines in the “Hello” instrument. This is not an error, but nothing will happen when you run Csound: There is no statement any more which invokes any instance of instrument “Hello”.
- Internally, Csound converts all instrument names to positive integer numbers. You can get this number via the `nstrnum` opcode. Put the code `iWhatIsYourNumber = nstrnum("Hello")` anywhere in the `CsInstruments` section, for instance below the `schedulelines`. Print this number to the console.
- Once you know this number, replace the string “Hello” in the `schedulelines` by this number. The Csound performance should be identical.
- When we call instrument “Hello” five times, as we do in the example, we call five instances of this instrument. We can assign numbers to these instances in calling the instrument as fractional number. Rather than calling instrument 1, we will call instrument 1.1, 1.2, 1.3, 1.4, and 1.5.

Replace the first argument of `schedule` by these numbers and insert `print(p1)` into the instrument code to prove that the instrument received this information.

- Keep this code, but change the first argument of `schedule` in the way that you still work with the instrument name here. Convert the instrument name “Hello” to a number via `nstrnum` and then add 0.1, 0.2, ... 0.5 for the five lines. Again the console output will be the same.

Csound runs and runs and runs ...

When we use a traditional score, as we did in [Tutorial 07](#), Csound will terminate after the last score event.

But when we leave the score empty, Csound will not terminate. To put it in an anthropomorphic way: Csound waits. Imagine we have established a network connection and can communicate with this Csound instance, then we might call again the “Hello” instrument this way. Or if we have a MIDI keyboard connected, we can do the same.

So usually we want this “endless” performance. (According to the [Csound Manual](#) this is about nine billion years on a 64-bit machine ...) But in case we do want Csound to terminate after a certain amount of time, we can put one line in the score: An `e` followed by a space and then a number. Csound will stop after this number of seconds. Please insert `e 20` in the `CsScore` section of our example above, and run it again. Now it should stop after 20 seconds.

The Csound Reference Manual as Companion and Field for Improvements

All information about the usage of an opcode is collected in the [Csound Reference Manual](#).

Every opcode has an own page here. Please have a look at the page for the `schedule` opcode which is at csound.com/docs/manual/schedule.html.

You will see that the native Csound way of writing code is used here. Rather than

```
schedule(insnum, iwhen, idur [, ip4] [, ip5] [...])
```

it is written

```
schedule insnum, iwhen, idur [, ip4] [, ip5] [...]
```

This is not a big difference. In the native Csound syntax, you will write the input arguments of an opcode at right hand side, and the output of an opcode left hand side; without any = in between.

As the `schedule` opcode has only input arguments, we have nothing at left hand side.

To get used to this way of writing, let us look at the `linseg` manual page. You find it [here](#) in the Reference Manual, and this is its information about the syntax of `linseg`:

```
ares linseg ia, idur1, ib [, idur2] [, ic] [...]
kres linseg ia, idur1, ib [, idur2] [, ic] [...]
```

In the functional style of coding which I use in this tutorial, it would read:

```
ares = linseg:a(ia, idu1, ib [, idur2] [, ic] [...])
kres = linseg:k(ia, idu1, ib [, idur2] [, ic] [...])
```

You will find detailed information in these Reference pages. Some of them may be too technical for you. You will also find a working example for each opcode which is very valuable to get an impression of what this opcode can do.

You may also read something on one of these pages which is outdated. For an Open Source Project it is always a major issue to keep the documentation up to date. We all are asked to contribute, if we can, for instance in opening a ticket at [Github](#) or by suggesting an improvement of the Reference Manual to the [Csound community](#).

Opcodes you have learned in this tutorial

Opcodes

- `schedule` calls an instrument instance similar to an `i` score line
- `nstrnum` returns the internal Csound number of an instrument name

Go on now ...

with the next tutorial: [09 Hello If.](#)

... or read some more explanations here

Score statements and utilities

Before we leave the score, we should at least mention some of its facilities.

It must be said that for the ones who write a fixed media piece, the score offers a lot of useful and proven tools.

So far we only used the `i` statement which calls an instrument instance, and which we somehow replaced in this tutorial by the schedule opcode in the `CsInstruments` section.

We also mentioned the `e` statement which terminates Csound after a certain time.

Another useful statement is the `t` or “Tempo” statement which sets the time for one beat, in metronome units. Per default, this is 60, which means that one beat equals one second. But it can be set to other values; not only once for the whole performance but having different metronome values at different times, and also interpolating between them (resulting in becoming faster or becoming slower).

Here is an overview, with links to detailed descriptions, in the Csound Reference Manual: csound.com/docs/manual/Scorestatements.html

And in this book, the Csound FLOSS Manual, we have a chapter about [methods of writing scores](#), too.

Steven Yi’s [Blue](#) frontend for Csound offers sophisticated possibilities of working with score events as objects.

Triggering other score events than ‘i’ from the orchestra code

If you need to trigger another score event than an instrument event, you can use the `event_i` opcode, and its `k-rate` version `event`. You should not expect everything to work, because the Csound score is preprocessed before the Csound performance starts. This is not possible when we fire a score event from inside the orchestra. So the `t` statements and similar will not work.

The `e` statement, however, will work, and can be used to terminate Csound in a safe way at any time of the performance. In this example, we call at first the `Play` instrument which plays a sine tone for three seconds. Then we call the `Print` instrument which displays its message in the console and

calls the *Terminate* instrument after three seconds. This instrument then terminates the Csound performance.

```
<CsoundSynthesizer>
<CsOptions>
-m128
</CsOptions>
<CsInstruments>
sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

instr Play
    aSine = poscil:a(.2,444)
    outall(linen:a(aSine,p3,p3/2))
endin
schedule("Play",0,3)

instr Print
    puts("I am calling now the 'e' statement after 3 seconds",1)
    schedule("Terminate",3,0)
endin
schedule("Print",3,1)

instr Terminate
    event_i("e",0)
endin

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

Three Particular Durations: 0, -1 and z

We have already used sometimes in examples the duration 0. For instance:

```
<CsoundSynthesizer>
<CsOptions>
-o dac -m128
</CsOptions>
<CsInstruments>
sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

instr Zero
    prints("Look at me!\n")
endin

</CsInstruments>
<CsScore>
i "Zero" 0 0
</CsScore>
</CsoundSynthesizer>
```

When you look at the console output, you see Look at me!.

So the instrument instance has been called, although there is “no duration” in the score line: The third parameter is 0.

For Csound, calling an instrument with duration zero means: Only execute the initialization pass. In other words: All *i*-rate statements will work, but no *k*-rate or *a*-rate one.

The other particular duration is -1. This was introduced for “unlimited” (held) duration and “tied” notes. Important to know if we want to use -1 as duration:

- An instrument called with a negative **p3** will run endlessly.
- But only one instance (!).
- We can turn off this instance by sending a score event with negative **p1**.

This is a simple example:

```
<CsoundSynthesizer>
<CsOptions>
-o dac -m128
</CsOptions>
<CsInstruments>
sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

instr 1
prints( "I am there!\n")
iMidiNote = p4
aSound = poscil:a(.2,mtof:i(iMidiNote))
aOut = linenr:a(aSound,0,1,.01)
outall(aOut)
endin

</CsInstruments>
<CsScore>
i 1 0 -1 70
i 1 1 -1 76
i -1 10 0 0
</CsScore>
</CsoundSynthesizer>
```

We hear that when the second instance starts after one seconds, the first instance is “pushed out” brutally. Csound assumes that we want to continue a *legato* line; so no reason for more than one instance at the same time.

After ten seconds, however, the held note is finished gracefully by the last score event, having -1 as first parameter. The fade-out is done here with the `linenr` opcode. We will explain more about it when we get to realtime MIDI input.

If we want an instrument to play “endlessly”, we can use the z character as special symbol for the duration. According to the [Csound Reference Manual](#), it causes the instrument to run “approximately 25367 years”.

Well, I do not dare to doubt this. But in this case, after ten seconds I started another instrument whichs turns off all instances of instrument “Zett”. We will use `turnoff2` or `turnoff2_i` more later.

```
<CsoundSynthesizer>
<CsOptions>
-o dac -m128
```

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

instr Zett
    prints("I am there!\n")
    iMidiNote = p4
    aSound = oscil:a(.1,mtof:i(iMidiNote))
    aOut = linenr:a(aSound,0,3,.01)
    outall(aOut)
endin

instr Turnoff
    turnoff2_i("Zett",0,1)
endin

</CsInstruments>
<CsScore>
i "Zett" 0 z 70
i "Zett" 1 z 76
i "Zett" 4 z 69
i "Turnoff" 10 0
</CsScore>
</CsoundSynthesizer>
```

Please note that **z** can **only** be used in the **score**. If we use it in a `schedule()` call, it will be interpreted as a variable name.

09 Hello If

What you learn in this tutorial

- How you can work with **if-then** in Csound.
- How you can **print formatted strings**.

One Instance Calls the Next ...

In the last Tutorial we have introduced the `schedule` opcode which calls an instrument instance.

We have put the `schedule` code outside the instrument code. In this case, it works like a score line.

But what will happen if we put a `schedule` statement also inside an instrument?

Try this:

```
<CsoundSynthesizer>
<CsOptions>
-o dac
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 64
nchnls = 2
0dbfs = 1

instr InfiniteCalls
//play a simple tone
aSine = poscil:a(.2,415)
aOut = linen:a(aSine,0,p3,p3)
outall(aOut)
//call the next instance after 3 seconds
schedule("InfiniteCalls",3,2)
endin
schedule("InfiniteCalls",0,2)

</CsInstruments>
<CsScore>
```

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

An infinite chain of calls is triggered.

The first call must be outside the instrument:

```
schedule("InfiniteCalls", 0, 2)
```

This line of code calls the first instance of instrument "InfiniteCalls".

But inside this instrument, again we have an instrument call. Three seconds after the instrument instance is created, the next instance will be there:

```
schedule("InfiniteCalls", 3, 2)
```

Note that the start time of the new instance is very important here. If you set it to 2 seconds instead of 3 seconds, it will create the next instance immediately after the current one. If you set the start time of the next instance to 1 second, two instances will overlap.

Schedule Depending on Conditions

This self-scheduling is a very interesting feature in Csound, but usually we do not want it to be infinite. Instead, we want to make it depending on a certain condition.

We will implement now an instrument which triggers itself again six times. This is what we must do:

- We must pass the number 6 as count variable to the first instrument instance.
- The second instance is then called with number 5 as count variable, and so on.
- If the instance with count variable 1 is reached, no more instances are called.

We can draw this program flow:

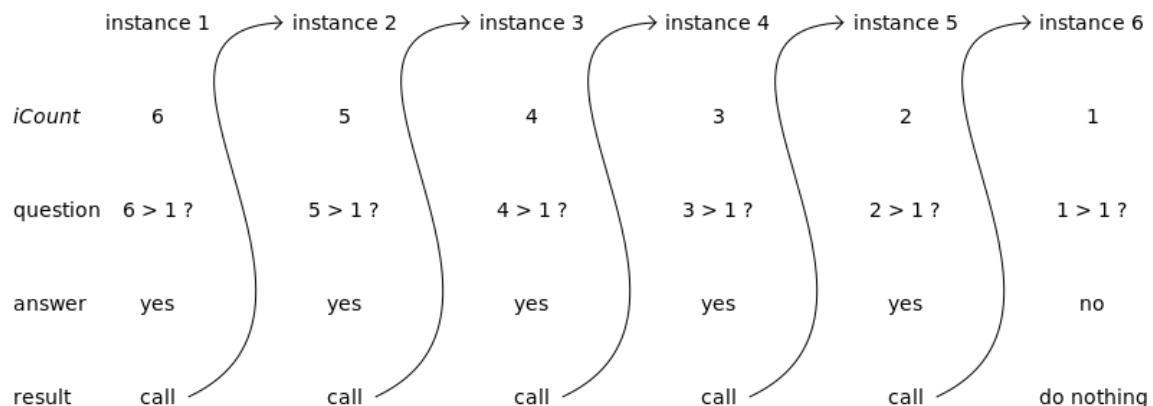


Figure 11.1: Program flow for conditional re-triggering of instrument instances

The 'if' Opcode in Csound

We can implement this limited chain of self-scheduling instances with the help of the `if` opcode. Try out this example by running and changing the `iCount` variable from 6 to 1, and look at the output.

```
<CsoundSynthesizer>
<CsOptions>
-o dac -m 128
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 64
nchnls = 2
0dbfs = 1

instr TryMyIf
iCount = 6
if (iCount > 1) then
    prints("True!\n")
else
    prints("False!\n")
endif
endin

</CsInstruments>
<CsScore>
i "TryMyIf" 0 0
</CsScore>
</CsoundSynthesizer>
```

The keywords here are `if`, `else` and `endif`. `endif` ends the `if` clause in a similar way as `endin` ends an instrument.

You again see the opcode `prints` here which shows a string in the console. We will explain more in the optional part of this tutorial about formatting.

If you want to read more about conditional branching with `if` have a look at [this](#) section in [Chapter 03](#) of this book.

Example

Please run the example and read the code. Can you figure out in which way the `iCount` variable is modified? Which other parameters are changed from instance to instance?

```
<CsoundSynthesizer>
<CsOptions>
-o dac -m 128
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 64
nchnls = 2
```

```

0dbfs = 1

instr Hello
    iMidiStart = p4
    iMidiEnd = p5
    kDb = linseg:k(-10,p3/2,-20)
    kMidi = linseg:k(iMidiStart,p3/3,iMidiEnd)
    aSine = oscil:a(ampdb(kDb),mtof(kMidi))
    aOut = linen:a(aSine,0,p3,1)
    outall(aOut)

    iCount = p6
    print(iCount)
    if (iCount > 1) then
        schedule("Hello",p3,p3+1,iMidiStart-1,iMidiEnd+2,iCount-1)
    endif
endin
schedule("Hello", 0, 2, 72, 68, 6)

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

Try it yourself

Change the code so that from instance to instance

- ▶ the distance in time between two instances increases
- ▶ the distance in time between two instances decreases by the ratio 1/2 (so that instance 3 has only half the distance to instance 2 as instance 2 had to instance 1)
- ▶ the duration increases by the ratio 3/2
- ▶ the first MIDI pitch increases by two semitones whilst
- ▶ the second MIDI pitch decreases by one semitone

More About 'if': If-Else and If-Elseif-Else

`if` is probably the most important word in any programming language. In using `if` we can create branches, and branches of branches, in our program flow. As the usage of `if` is so common for us all in daily life, we can easily transfer it to a programming language, in this case to Csound.

Some example cases following here, based on daily situations.

If - else

"If the sun shines then I will go out, else I will stay at home."

This is the Csound version. Please change the *iSun* variable.

```
<CsoundSynthesizer>
<CsOptions>
-o dac -m 128
</CsOptions>
<CsInstruments>
sr = 44100
ksmps = 64
nchnls = 2
0dbfs = 1

instr If_Then
iSun = 1 // 1 = yes, 0 = no
if (iSun == 1) then
    prints("I go out!\n")
else
    prints("I stay at home!\n")
endif
endin

</CsInstruments>
<CsScore>
i "If_Then" 0 0
</CsScore>
</CsoundSynthesizer>
```

In `if (iSun == 1)` we ask for equality. This is the reason that we use `==` rather than `=`. The double equal sign `==` **asks** for equality between left and right. The simple equal sign `=` **sets** the left hand side variable to the right hand side value, like in `iSun = 1`. Csound also accepts `if (iSun = 1)` but I believe it is better to distinguish these two qualities.

The parentheses in `(iSun == 1)` can be omitted but I prefer to put them for better readability.

Csound does not have an own symbol or keyword for the Boolean “True” and “False”. Usually we use **1** for True/Yes and **0** for False/No.

If - elseif - else

When we add one or more `elseif` questions, we come to a decision between several cases.

“If the pitch is higher than MIDI note 80, then instrument *High* will start. Elseif the pitch is higher than MIDI note 60, then instrument *Middle* will start. Else instrument *Low* will start.”

This is the Csound version. Please change the *iPitch* variable.

```
<CsoundSynthesizer>
<CsOptions>
-o dac -m 128
</CsOptions>
<CsInstruments>
sr = 44100
ksmps = 64
nchnls = 2
0dbfs = 1

instr If_Elseif_Then
iPitch = 90 // change to 70 and 50
if (iPitch > 80) then
```

```

    schedule("High",0,0)
elseif (iPitch > 60) then
    schedule("Middle",0,0)
else
    schedule("Low",0,0)
endif
endin

instr High
    prints("Instrument 'High'!\n")
endin

instr Middle
    prints("Instrument 'Middle'!\n")
endin

instr Low
    prints("Instrument 'Low'!\n")
endin

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

Note that in the second condition `elseif (iPitch > 60)` we grab all pitches which are larger than 60 and lower or equal 80, because we only come to this branch if the first condition `(iPitch > 80)` is not true.

Even More About If: Nested 'if's; AND and OR

Nested 'if's

We can have multiple levels of branching, for instance:

"If the sun shines, then:

- If I need some fruits, then I will go to the market,
- Else I will go in the woods;

Else (= if the sun is not shining):

- If I am hungry then I will cook some food,
- Else (= if I am not hungry):
 - If I am not tired then I will learn more Csound,
 - Else I will have a rest."

```

<CsoundSynthesizer>
<CsOptions>
-o dac -m 128
</CsOptions>
<CsInstruments>
```

```

sr = 44100
ksmps = 64
nchnls = 2
0dbfs = 1

instr Nested_If
    // input situation (1 = yes)
    iSun = 1
    iNeedFruits = 1
    iAmHungry = 1
    iAmTired = 1
    // nested IFs
    if (iSun == 1) then      //sun is shining:
        if (iNeedFruits == 1) then //i need fruits
            prints("I will go to the market\n")
        else                      //i do not need fruits
            prints("I will go to the woods\n")
        endif                     //end of the 'fruits' clause
    else                      //sun is not shining:
        if (iAmHungry == 1) then //i am hungry
            prints("I will cook some food\n")
        else                      //i am not hungry
            if (iAmTired == 0) then //i am not tired
                prints("I will learn more Csound\n")
            else                      //i am tired
                prints("I will have a rest\n")
            endif                     //end of the 'tired' clause
        endif                     //end of the 'hungry' clause
    endif                     //end of the 'sun' clause
endin

</CsInstruments>
<CsScore>
i "Nested_If" 0 0
</CsScore>
</CsoundSynthesizer>

```

Logical AND and OR

Instead of two nested *if* which ought to be true, we can also ask whether both are true: “If the sun is shining and I have finished my work, I will go out.”

In Csound, like in most programming languages, the symbol for this logical AND is `&&`.

The symbol for the logical OR is `||`.

Here comes an example for both. Try changing the values and watch the output.

```

<CsoundSynthesizer>
<CsOptions>
-o dac -m 128
</CsOptions>
<CsInstruments>
sr = 44100
ksmps = 64
nchnls = 2
0dbfs = 1

instr AndOr

```

```
iSunIsShining = 1
iFinishedWork = 1
prints("iSunIsShining = %d\n", iSunIsShining)
prints("iFinishedWork = %d\n", iFinishedWork)
//AND
if (iSunIsShining == 1) && (iFinishedWork == 1) then
    prints("AND = True\n")
else
    prints("AND = False\n")
endif
//OR
if (iSunIsShining == 1) || (iFinishedWork == 1) then
    prints("OR = True\n")
else
    prints("OR = False\n")
endif
endin

</CsInstruments>
<CsScore>
i "AndOr" 0 0
</CsScore>
</CsoundSynthesizer>
```

Opcodes you have learned in this tutorial

- if ... then ... [elseif] ... [else] ... endif conditional branching

Go on now ...

with the next tutorial: [10 Hello Random](#).

... or read some more explanations here

A short form

We have a short form for if in Csound which is quite handy when we set a variable to a certain value depending on a condition.

Instead of ...

```
if (iCondition == 1) then
    iVariable = 10
else
```

```
iVariable = 20
endif
```

... we can write:

```
iVariable = (iCondition == 1) ? 10 : 20
```

You can find another example [here](#) in this book.

Looping with 'if'

It is even possible to build loops with the `if` opcode. This is shown here as a sidenote; not to write code in this style. But it shows that also in looping constructions in programming languages 'if' is behind the scene.

All we need, in addition to the `if` opcode, is:

- A "label" which marks a certain position in the program text. In Csound, these labels end with a colon. We use `start:` here as label.
- A "jump to" mechanism. This is called `goto` in Csound.

This "old fashioned" loop counts from 10 to 1 and then leaves the loop.

```
<CsoundSynthesizer>
<CsOptions>
-o dac -m 128
</CsOptions>
<CsInstruments>
sr = 44100
ksmps = 64
nchnls = 2
0dbfs = 1

instr LoopIf
    iCount = p4
    start:
    if (iCount > 0) then
        print(iCount)
        iCount = iCount-1
        igoto start
    else
        prints("Finished!\n")
    endif
endin

</CsInstruments>
<CsScore>
i "LoopIf" 0 0 10
</CsScore>
</CsoundSynthesizer>
```

In line 16, rather than writing `iCount = iCount-1`, we could also write:

```
iCount -= 1
```

Format strings

The prints opcode prints a string to console.

The string can be a *format string*. This means that it has empty parts or place holders which can be filled by variables.

These place holders always start with % and are followed by a character which signifies a data type. The most common ones are:

- %d for an integer
- %f for a floating point number
- %s for a string

Note that a new line must be assigned via \n. Otherwise after the printout the next message immediately follows, on the same line.

Here comes a simple example:

```
<CsoundSynthesizer>
<CsOptions>
-o dac -m 128
</CsOptions>
<CsInstruments>
sr = 44100
ksmps = 64
nchnls = 2
0dbfs = 1

instr FormatString
prints("This is %d - an integer.\n",4)
prints("This is %f - a float.\n",sqrt(2))
prints("This is a %s.\n","string")
prints("This is a %s","concat")
prints("nated ")
prints("string.\n")
endin

</CsInstruments>
<CsScore>
i "FormatString" 0 0
</CsScore>
</CsoundSynthesizer>
```

More can be found [here](#) in this book. The format specifier are basically the same as in the C programming language. You can find a reference [here](#).

10 Hello Random

What you learn in this tutorial

- How to work with **random numbers** in Csound
- How to set a certain **seed** for random sequences
- How to implement **structural ideas** with random portions
- How a **random walk** can be implemented
- What the **global space** in the Csound orchestra is

Random Numbers and Artistic Decisions

Throwing the dies and inventing games which follow the dies' result has always been enjoyable for people. Most interesting is the relationship between the decided rules and the unpredictability of the next throw.

In modern art and music random choices often have an important role. It can be on a more technical level, for instance when we use random deviations for [granular synthesis](#) to somehow imitate nature in its permanent variety.

But it can also be an essential part of our invention that we create *structures* which can be realized in one or the other way, rather than determining each single note like in a melody.

We will create a simple example for this way of composing here. It will show that "working with random" does not at all mean "withdraw from decisions". In contrary, the decisions are there, and most important for what can happen.

The 'random' Opcode and the 'seed'

For getting a random number, we set one limit for the smallest possible number, and one limit for the largest possible number. Inside this range, a random number is selected.

This is a simple example for a random number between 10 and 20. Please run it three times and watch the console printout.

```
<CsoundSynthesizer>
<CsOptions>
-o dac -m 128
</CsOptions>
<CsInstruments>
sr = 44100
ksmps = 64
nchnls = 2
0dbfs = 1

instr Random
    iRandomNumber = random:i(10,20)
    prints("Random number between 10 and 20: %f\n",iRandomNumber)
endin

</CsInstruments>
<CsScore>
i "Random" 0 0
</CsScore>
</CsoundSynthesizer>
```

You will see that three times the same random number is generated. My printout shows:

Random number between 10 and 20: 18.828730

Why?

Strictly spoken the computer has no random because it can only calculate. A random number is created internally by a calculation. Once the first number is there, all other numbers are determined by a [pseudorandom number generator](#).

This starting point of a random sequence is called *seed*. If we do not set a seed, Csound uses a fixed number. This is the reason why we always got the same number.

The seed opcode offers us two possibilities:

- ▶ For `seed(0)` Csound will seed from the current clock. This is what most other applications do as default. It results in an always different start value, so that is what we usually want to get when we use `random`.
- ▶ For any positive integer number we put in `seed`, for instance `seed(1)` or `seed(65537)`, we get a certain start value of the random sequence. `seed(1)` will yield another result as `seed(65537)`. But once you run your Csound program twice with `seed(1)`, it will result in the same random values. This is a good opportunity to check out different random traces, but being able to reproduce any of them precisely.

Please insert `seed(0)` in the example above. It should be placed below the `0dbfs = 0` line, in the global space of the orchestra. When you run your code several times, it should always print a different *iRandomNumber* output.

Also try to insert `seed(1)` or `seed(2)` etc. instead. You will see that each output is different, but once you run one of them twice, you will get the same result.

The “Global Space” or “instrument 0”

As you know from [Tutorial 02](#) and from [Tutorial 08](#), the space for the actual Csound code in a .csd document is what is enclosed by the <CsInstruments> tag. This is also called the “orchestra”.

Inside the orchestra, we have instrument definitions. Each instrument starts with the keyword `instr` and ends with the keyword `endin`.

But we have also a “global space” in the orchestra. “Global” means here: **Outside** any instrument.

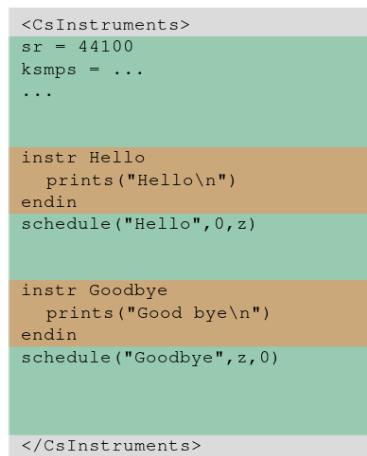


Figure 12.1: Global space (green) and instruments (brown) in CsInstruments section

As you see, the global space is not only on top of the orchestra. In fact, each single empty line outside an instrument is part of the global space.

We have already used this global space. The “orchestra header constants” live in this global space, outside any instrument, when we set:

```

sr = 44100
ksmps = 64
nchnls = 2
0dbfs = 1
  
```

We also used the global space in setting our instrument call via `schedule` below an instrument definition:

```

instr MySpace
    ...
endin
schedule( "MySpace" , 0 , 1 )
  
```

The instrument definition establishes a local space. The `schedule(...)` line resides in the global space.

Sometimes this global space is called *instrument 0*. The reason is that the instruments in the orchestra have 1 as smallest possible number.

This is what we **can** do in the global space:

- We can set global parameters like sample rate etc, and also the seed because it is a global parameter, too.
- We can define own functions or import external code.
- We can create tables (buffers) and assign software channels.
- We can perform *i-rate* expressions. Insert, for instance, `prints("Hello Global Space!\n")` in the global space and look in the console output.

What we can **not** do in the global space:

- We cannot perform any *k-rate* or *a-rate* statement.

The global space is read and executed once we run Csound, even if we do not call any instrument.

Example

Please this time read the code first, and guess how it will sound for each note. What will be the development of this sketch, and how long do you expect it to go?

```
<CsoundSynthesizer>
<CsOptions>
-o dac
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 64
nchnls = 2
0dbfs = 1
seed(12345)

instr Hello
//MIDI notes between 55 and 80 for both start and end
iMidiStart = random:i(55,80)
iMidiEnd = random:i(55,80)
//decibel between -30 and -10 for both start and end
iDbStart = random:i(-30,-10)
iDbEnd = random:i(-30,-10)
//calculate lines depending on the random choice
kDb = linseg:k(iDbStart,p3/2,iDbEnd)
kMidi = linseg:k(iMidiStart,p3/3,iMidiEnd)
//create tone with fade-out and output
aSine = oscil:a(ampdb(kDb),mtof(kMidi))
aOut = linen:a(aSine,0,p3,p3/2)
outall(aOut)

//trigger next instance with random range for start and duration
iCount = p4
if (iCount > 1) then
  iStart = random:i(1,3)
  iDur = p3 + random:i(-p3/2,p3)
  schedule("Hello",iStart,iDur,iCount-1)
endif
endin
schedule("Hello", 0, 2, 15)

</CsInstruments>
<CsScore>
```

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

Structural Decisions

We have a lot of random opcodes in the code. Let us look closer to the decisions inherited in them, and to the effects which result from the decisions.

```
iMidiStart = random:i(55,80)
iMidiEnd = random:i(55,80)
```

Setting both, start and end pitch of the *glissando* line to a range from MIDI note 55 (= F#4) to 80 (= G#6) makes it equally probable that rising and falling lines will appear. Some will have a big range (imagine a line from 78 to 56) whilst others will have a small range (imagine a line from 62 to 64).

The alternative could be, for instance, to set:

```
iMidiStart = random:i(55,67)
iMidiEnd = random:i(68,80)
```

Then the sliding pitch line would always be upwards. Or:

```
iMidiStart = random:i(55,70)
iMidiEnd = random:i(65,80)
```

Then the pitch line would be mostly upwards, but sometimes not.

Similar decisions apply for the volume line which is set to:

```
iDbStart = random:i(-30,-10)
iDbEnd = random:i(-30,-10)
```

The maximum difference is 20 dB which is not too much. So there is some variance between louder and softer tones but all are well perceivable, and there is not much foreground-background effect as it would probably occur with a range of say -50 to -10 dB.

The most important decisions for the form are these, concerning the distance between subsequent notes and the duration of the notes:

```
iStart = random:i(1,3)
iDur = p3 + random:i(-p3/2,p3)
```

The distance between two notes is between one and three seconds. So in average we have a new note every two seconds.

But the duration of the notes is managed in a way that the next note duration has this note's duration (**p3**) plus a random range between

- minus half of this note's duration as minimum, and
- this note's duration as maximum.

(This is the same as a random range between $p3/2$ and $p3*2$. But I personally prefer here thinking the next duration as "this duration plus/minus something".)

For the first note which has a duration of two seconds, this means a random range between one and four seconds. So the tendency of the duration is to become larger and larger. Here is what happens in the example code above for the first seven notes:

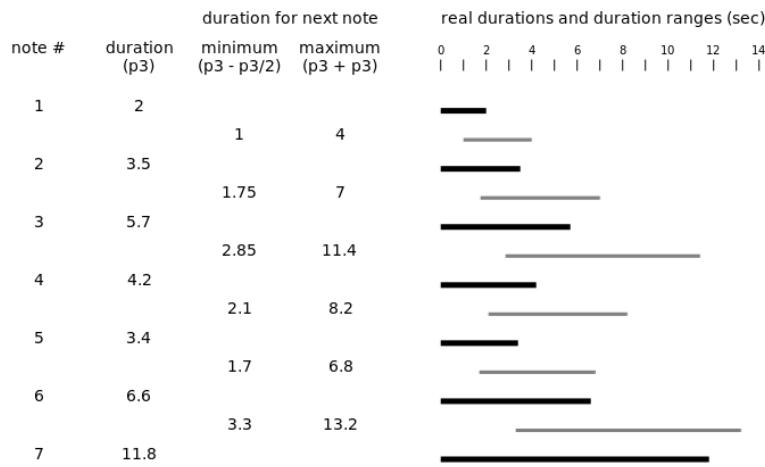


Figure 12.2: Duration development for notes 1-7

It is interesting to see that note 2 and 3 expand their duration as expected, but then note 4 and 5 shrink because they chose their durations close to the minimum.

But on the long run the larger durations prevail so that more and more notes sound at the same time, forming chords or clusters.

Try it yourself

- Set `seed(0)` instead of `seed(12345)` and listen to some of the results.
- Change line 32 `iDur = ...` so that you get equal probability for longer or shorter durations, without any process. Make up your mind about this version.
- Change line 32 `iDur = ...` so that on the long run the durations become shorter and shorter.
- Change the code so that for half of the notes to be played the durations become longer, and for the second half the durations become shorter.
- Change the code so that for the first half of the notes the distance between subsequent notes become shorter, and in the second half again become longer.
- Apply this change also to the pitches and the volume so that in the first half the pitches increase whilst the volume decreases, and then in the second half vice versa.

Random Walks

In a *random walk*, the random values of the next step depend on the previous step.

In the example above, the durations are following a random walk, whilst the other random decisions are independent from a previous step.

As shown in the figure above, note number seven with a duration of 11.8 seconds would not have been possible in one of the previous steps. It depends on the random range which is generated in step six.

The random walk of the note durations is combined with a tendency, leading to a development. But it is also possible to keep the conditions for the next step constant, but nevertheless get surprising patterns. Have a look at the [Wikipedia article](#) or other sources.

This is a random walk for pitch, volume and duration. The conditions for the next step remain unchanged, but nevertheless there can be a direction in each of the three parameters.

```
<CsoundSynthesizer>
<CsOptions>
-o dac -m 128
</CsOptions>
<CsInstruments>
sr = 44100
ksmps = 64
nchnls = 2
0dbfs = 1
seed(54321)

instr RandomWalk
//receive pitch and volume
iMidiPitch = p4
iDecibel = p5
//create tone with fade-out and output
aSine = poscil:a(ampdb:i(iDecibel),mtot:i(iMidiPitch))
aOut = linen:a(aSine,0,p3,p3)
outall(aOut)

//get count
iCount = p6
//only continue if notes are left
if (iCount > 1) then
    //notes are always following each other
    iStart = p3
    //next duration is plusminus half of the current maximum/minimum
    iDur = p3 + random:-p3/2,p3/2)
    //next pitch is plusminus a semitone maximum/minimum
    iNextPitch = iMidiPitch + random:-1,1)
    //next volume is plusminus 2dB maximum/minimum but
    //always in the range -50 ... -6
    iNextDb = iDecibel + random:-3,3)
    if (iNextDb > -6) || (iNextDb < -50) then
        iNextDb = -25
    endif
    //start the next instance
    schedule("RandomWalk",iStart,iDur,iNextPitch,iNextDb,iCount-1)
    //otherwise turn off
else
    event_i("e",0,0)
endif

//print the parameters of this instance to the console
prints("Note # %2d, Duration = %.3f, Pitch = %.2f, Volume = %.1f dB\n",
      50-iCount+1, p3, iMidiPitch, iDecibel)
endin
schedule("RandomWalk",0,2,71,-20,50)
```

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

This is how the result looks like:

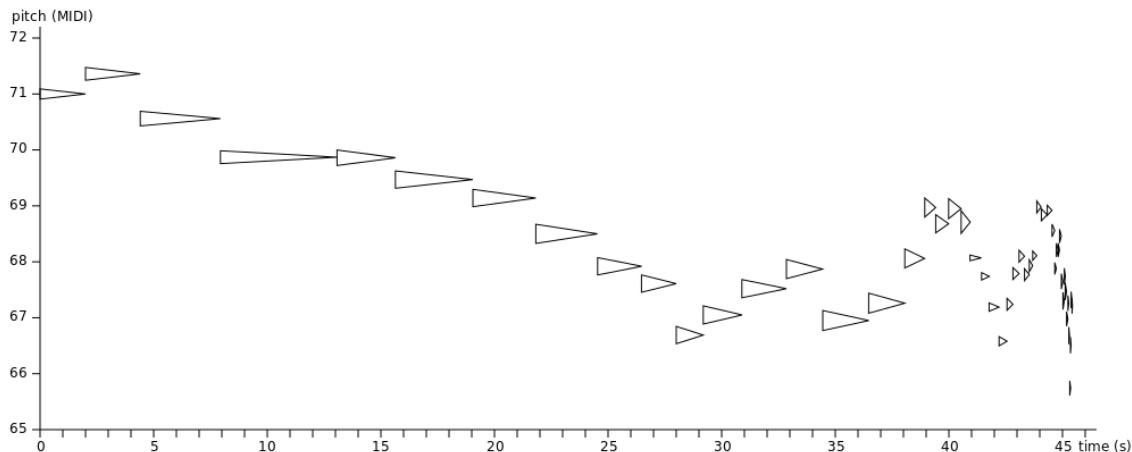


Figure 12.3: Random walk

Without changing the conditions for the random walk, we get an extreme reduction of the durations at the end of this sequence.

In general, random in art is a part of our phantasy and invention. By introducing any "if", we can change conditions in whatever situation. For instance: "If the pitch has been constant in the last three steps, jump to the upper or lower border."

"Random" and "if" together can create crazy realities. Very individual because you yourself came across these ideas. Very rational because it is all based on written rules and conditions. Very unpredictable because the random chain may lead to unforeseen results. And perhaps this is one of the biggest qualities ...

Opcodes and terms you have learned in this tutorial

- `random:i(iMin, iMax)`
- `seed(iNum)`
- *global space* is the part of the `CsInstruments` section which is outside any instrument definition
- *instrument 0* is another word for it

Go on now ...

with the next tutorial: [11 Hello Keys](#).

... or read some more explanations here

Remember i-rate and k-rate ...

Perhaps you noticed that until now we only used random at *i*-rate. As you learned in [Tutorial 05](#) this means that a random value is only generated **once**, at the initialization of the instrument instance.

What happens if we use random at *k*-rate? Like

```
kMidi = random:k(55,80)
```

This will generate a random number between 55 and 80 **in every k-cycle**. As calculated [here](#) this is usually about 1000 times per second.

So this is a massive difference which is only introduced by calling either `random:i` or `random:k`. Although it is consistent and just following what we explained in [Tutorial 02](#) and [Tutorial 05](#), it is often surprising for beginners.

We even can use the `random` opcode at *a*-rate. Then we generate one random value for each sample, so 44100 times per second, if this is our sample rate. When we choose a reasonable range for minimum and maximum, we can listen to it, and call it “white noise”:

```
aNoise = random:a(-0.1,0.1)
```

Random with interpolating or with held values

Having only the choice between one random value for the whole duration of an instrument event (*i*-rate) and about one thousand different random values per second (*k*-rate) is not enough for many use cases. Imagine we want a sound to move between two speakers. We may want it to change its direction about once per second.

This is a typical case which is covered by the `randomi` opcode. The **i** as the end of its name means **interpolating**. This opcode generates random numbers in a certain density, and draws lines between them. These lines are the interpolations.

The input arguments for `randomi` are: 1. minimum, 2. maximum, 3. how many values to generate per second, 4. is about different possibilities at the beginning and should be set to 3 for normal use (consult the reference for more or have a look [here](#)).

This is a random line between 0 and 1, with one new value every second:

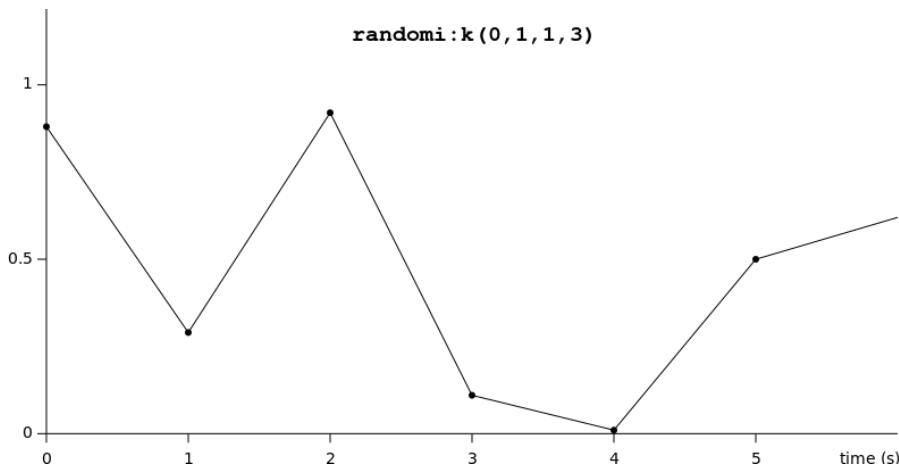


Figure 12.4: Random with interpolation

Sometimes we want the random values to be held until the next one. This is the job of the `randomh` opcode.

The input parameters carry the same meaning as for `randomi`. This is the output for `randomh` with the same input arguments as in the previous plot:

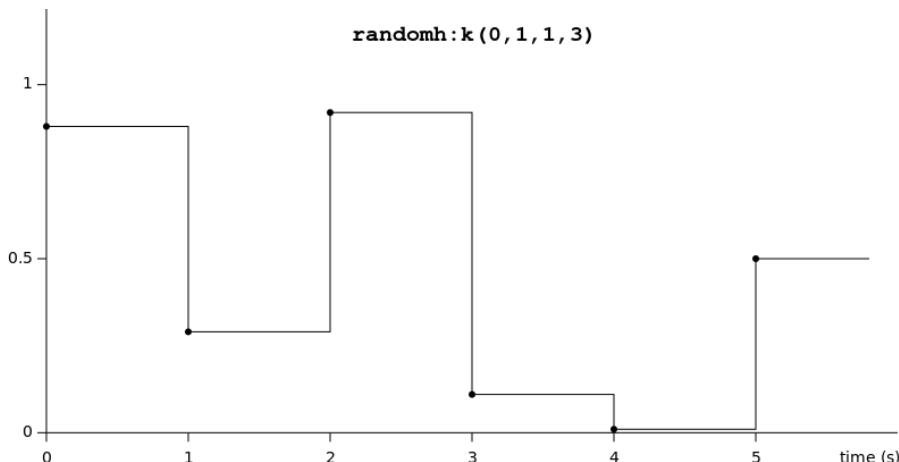


Figure 12.5: Random with held values

All this is just a small selective view on the big world of random.

In all the opcodes we discussed so far we have an equal distribution of the random numbers in a certain range. But in nature, we often have a larger probability in the middle than near the borders of the random range. The [Gaussian distribution](#) is the mathematical formulation for it, and it is implemented in the `gauss` opcode in Csound.

And the *random walk* is only one possibility of a random which depends on a context rather than throwing the dies. The *Markov chain* is another approach. If you want to read more, and run some examples, please have a look at the [Random](#) chapter in this book.

Congratulations!

You have now finished the first ten of these tutorials. This block was meant as a general introduction. To summarize some of the contents which you know now:

- You know how a **.csd file** is structured.
- You know how you set **sample rate**, **block size** and other constants.
- You know how you define **instruments** as main units of any Csound program.
- You know how you call **instances** of instruments via a **score line**.
- You know how you can do the same from inside the orchestra code via **schedule**.
- You know what **i-rate**, **k-rate** and **a-rate** is. (In case you become nervous here, and feel a slight fear: Have a look [here](#) and [here](#)).
- You know how you can use **opcodes** and how to read the [Csound Reference Manual](#) to know about their inputs and outputs.
- You already know some essential opcodes, like
 - outall and out for outputting audio signals
 - poscil as multi-purpose oscillator
 - linseg as generator for whatever lines we need
 - linen as simple tool for fades
 - mtod and ampdb as important converters for pitch and volume
 - if as conditional branching
 - random and derivates as random generators

I think this should be enough to jump anywhere in this textbook, and either dig deeper in the Csound language in [Chapter 03](#), or look at sound synthesis methods in [Chapter 04](#), or to ways to modify existing sounds in [Chapter 05](#).

Of course you are welcome to continue these tutorials which will now move to live input and will try to cover other basic subjects of using Csound in an interactive way.

HOW TO: INSTALLATION

Csound and Frontends

What is a Frontend?

In computer science, we call *frontend* what is close to the user, and *backend* what is close to the hardware.

So a frontend is what you as user see, in which you type text, scroll windows, open and save files by clicking in the menu bar.

Why does Csound not come with a Frontend?

Csound is an “audio engine” or “audio library”. It is a machine to render digital audio data. This machine consists of a bundle of executable or callable files. You can find these files on your computer, but nothing will happen when you try to open these files.

There were a lot of discussions in the Csound community whether one standard frontend should be distributed together with Csound, or not. Sometimes it was the case, but as for today, it is not. The main reasons are:

1. Each frontend has a different design and different ways to use it. There is not the one standard way to build a frontend for Csound.
2. Although it is confusing for beginners, it is important to distinguish between Csound as audio engine and any frontend. It is the big flexibility of Csound that it can be used in many contexts. Each of these contexts somehow establishes a new frontend.

Can I install Csound without a Frontend?

Yes, and this will happen when you go to the [Csound Download Page](#) and click on any of the “install Csound” buttons.

The first step is to install this **plain Csound**.

The second step is to install a frontend.

Which Jobs are done by a Frontend?

A frontend, or IDE (Integrated Development Environment) will **run** Csound. Usually you have a button to **start** and a button to **stop** a Csound performance.

A frontend will provide an **editor** in which you can open, modify and save **.csd files**.

In addition to these essentials, the existing frontends offer different other features, depending on their orientation.

Which Frontends are available?

[CsoundQt](#) has been written in 2008 by Andrés Cabrera to offer a multi-purpose frontend for basic usage of Csound. It is now maintained by Tarmo Johannes and Eduardo Moguillansky.

[Cabbage](#) has been developed since 2007 by Rory Walsh. Its main focus is to export plugins from a Csound file to run in a DAW. Cabbage has grown a lot and is currently the most popular way to work with Csound.

[Blue](#) has been developed since 2004 by Steven Yi. Its main focus is to provide an object-oriented way to work with the Csound score. Around this, many features were added.

[Web IDE](#) is an online IDE which has been started in 2018. Nothing to install here – just create an account and use Csound in your browser.

Which Frontend should I choose?

This depends on your needs, and what you like. All possibilities are good for learning Csound.

How to install on Windows?

Make sure you understood what is mentioned on top of this page about Csound and frontends. Usually you will need to install **both**: plain Csound **and** a frontend.

How to install plain Csound on Windows?

Go to <https://csound.com/download>. Under the Windows icon, choose the "64bit Full Installer". After downloading, follow the usual procedure to install

How to install a frontend on Windows?

- CsoundQt: <https://github.com/CsoundQt/CsoundQt/releases>
- Cabbage: <https://cabbageaudio.com/download/>
- Blue: <https://blue.kunstmusik.com/#download>

How to install on Mac?

Make sure you understood what is mentioned on top of this page about Csound and frontends. Usually you will need to install **both**: plain Csound **and** a frontend.

How to install plain Csound on Mac?

Go to <https://csound.com/download>. Under the Apple icon you will see a *.dmg* (disk image) for download. After downloading, open the *.dmg* and run the installer. If you get a message about "cannot install because it is not from a certified apple developer", go to your System Preferences. In the "Security" panel, allow the Csound can be installed.

How to install a frontend on Mac?

- CsoundQt: <https://github.com/CsoundQt/CsoundQt/releases>
- Cabbage: <https://cabbageaudio.com/download/>
- Blue: <https://blue.kunstmusik.com/#download>

How to install on Linux?

Make sure you understood what is mentioned on top of this page about Csound and frontends. Usually you will need to install **both**: plain Csound **and** a frontend.

How to install plain Csound on Linux?

On <https://csound.com/download> you find links to the repositories of different Linux distributions.

I personally recommend to install Csound from sources. If you have some experiences in how to do it, you get the most recent Csound and can even switch to the development version. Building is not complicated; a good description is [here](#).

How to install a frontend on Linux?

- ▶ CsoundQt: <https://github.com/CsoundQt/CsoundQt/releases>
- ▶ Cabbage: <https://cabbageaudio.com/download/>
- ▶ Blue: <https://blue.kunstmusik.com/#download>

How to install on Android?

On Android, Csound comes with a minimal included frontend. It can be easily found in Google's Play Store or on similar collections.

HOW TO: GET HELP

REFERENCE MANUAL

Where can I find the Reference Manual?

- Online: <https://csound.com/docs/manual/index.html>
- Download: <https://github.com/csound/manual/releases>

ORIENTATION

Where can I find an Opcode Overview?

- In the Reference Manual: <https://csound.com/docs/manual/PartOpcodesOverview.html>
- In the Floss Manual: [Opcode Guide](#)

ERRORS

Where can I find Csound's error messages?

Csound's error messages are displayed in the *Console*.

The console will look different depending on your way to run Csound. Here in this online textbook, the console shows up when you push the *Run* button of an example:

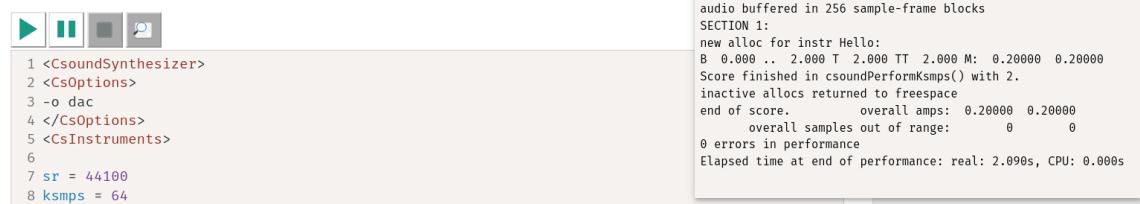
Example

We are now ready to run the code. All we have to do is put the instrument in a complete Csound file.

Look at the code in the example. Can you find the instrument code?

Push the "Play" button. You should hear two seconds of a 400 Hz sine tone.

Can you see why it plays for two seconds?



```

Csound WebAssembly console
instr Hello uses instrument number 1
--Csound version 6.18 (double samples) Feb 17 2023
[commit: HEAD]
libsndfile-1.1.0
graphics suppressed, ascii substituted
sr = 44100.0, kr = 689.062, ksmps = 64
0dBFS level = 1.0, A4 tuning = 440.0
orch now loaded
audio buffered in 256 sample-frame blocks
SECTION 1:
new alloc for instr Hello:
B 0.000 .. 2.000 T 2.000 TT 2.000 M: 0.20000 0.20000
Score finished in csoundPerformKsmmps() with 2.
inactive allocs returned to freespace
end of score.          overall amps: 0.20000 0.20000
          overall samples out of range: 0 0
0 errors in performance
Elapsed time at end of performance: real: 2.090s, CPU: 0.000s

```

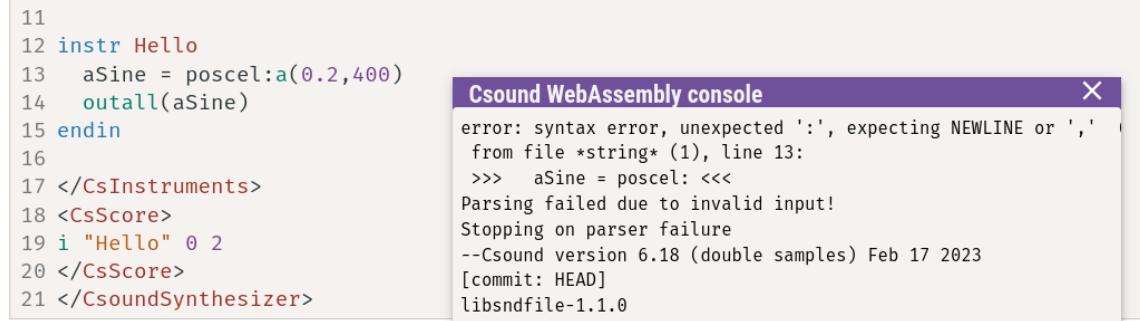
Figure 14.1: Csound console (right) in WebAudio for Floss Manual

In frontends the console may be hidden, but you will be able to bring it to view.

Depending on your way to use Csound, the console may show more or less information. But the error messages should always be visible.

How can I find the related code line for an error message?

Csound tries to point us to the first error in the code. Look at the following output; the error was to write poscel rather than poscil.



```

11
12 instr Hello
13   aSine = poscel:a(0.2,400)
14   outall(aSine)
15 endin
16
17 </CsInstruments>
18 <CsScore>
19 i "Hello" 0 2
20 </CsScore>
21 </CsoundSynthesizer>

Csound WebAssembly console
error: syntax error, unexpected ':', expecting NEWLINE or ',' from file *string* (1), line 13:
>>>   aSine = poscel: <<<
Parsing failed due to invalid input!
Stopping on parser failure
--Csound version 6.18 (double samples) Feb 17 2023
[commit: HEAD]
libsndfile-1.1.0

```

Figure 14.2: Error message with line number

The first line is a bit obscure, but **syntax error** is the important bit here.

The second line reports the line number in the code as 13 which is true. (Sometimes in frontends the line number is plus-minus one.)

The third line is a quotation of the wrong line:

```
>>> aSine = poscel: <<<
```

Note that it does not report the full line, but up to the point at which the syntax error occurred for the Csound parser.

What are the most common error messages?

The following is of course a subjective selection.

1. used before defined

This error message pops up when we use a variable name as input to an opcode, but this variable has no value. Often this is just a typo, for instance we typed `asine` rather than `aSine`.

2. failed to open file (No Error.)

This means that Csound could not find the file, typically as input string for the `diskin` opcode. "No Error" means here that this is not a syntax error. Once the file can be found, the code can be executed.

3. `diskin2`: number of output args inconsistent with number of file channels

This is a frequent message when we have a mismatch between the file input and the number of outputs for the `diskin` opcode. For instance, Csound will complain for `aL,aR diskin "fox.wav"`

because `fox.wav` is a mono sound file, but with `aL,aR` we claimed it to be stereo.

How can I get help if I don't understand an error message?

Use one of the many channels to contact other Csound users: Forum, Chat, or Email. See [here](#) for more.

What can I do when Csound just crashes?

This is somehow the worst case: When Csound crashes, you will not see any error message at all, so you have no clue what went wrong.

If possible, run your Csound file from the command line. The command line itself will not crash, and perhaps you see something which gives more information.

HOW TO: HARDWARE

AUDIO DEVICES

Do I have to handle audio via Csound when I use a frontend?

You will be able to select your audio device via your frontend, for instance [Cabbage](#), [Blue](#) or [CsoundQt](#).

But it is good to know what the [command line options](#) are and what they mean: Good for a deeper understanding of what happens, and for being able to solve problems.

What is an Audio Module in Csound?

An Audio Module handles the real-time input and output exchange of audio between Csound and the sound card (to put it in a simplified way).

Which Audio Modules are available?

The default cross-platform audio module is [PortAudio](#). Another cross-platform audio module is [Jack](#). Other audio modules are platform specific, like Alsa for Linux, Coreaudio for Mac, mme for Windows.

What is the best choice for selecting an audio module?

In my experience, [Jack](#) works best. Although it requires a bit more time to learn how it works, I think it is worth and pays off.

How can I select an audio module?

If you use a frontend, you will find it in the settings / preferences of [Cabbage](#), [CsoundQt](#) or [Blue](#).

(Note that Cabbage uses its own internal audio module instead of the ones provided by Csound. It is also hardwired to 2 channels minimum.)

If you use plain Csound (command line Csound or Csound via API), use the option

```
-+rtaudio=...
```

in the CsOptions tag.

For instance, to use Jack, you will write

```
-+rtaudio=jack
```

How can I connect Csound to a certain audio card?

In case you use a frontend, you will not need to write any options or [command line flags](#) in your .csd file. The following description is only important for using Csound on the command line or via API.

Remember that you also need to choose a real-time audio module. (See above how to do this.)

Csound connects with a specific sound card separately for input and output.

Output

To **output** Csound audio in realtime to a sound card use the option

```
-o dac
```

in your CsOptions tag.

If you use -o dac, Csound will connect with the system default. This is usually desirable.

If you want another audio device than the system default, you need to get the number of this device first. It should be written in the Csound console when Csound starts. If not, run Csound with this option:

```
csound --devices
```

Csound will print out the list of available devices.

Once you picked out the one you want to use, call it for instance as

```
-o dac2
```

Input

To get **input** from your sound card into Csound, use the option

```
-i adc
```

in your CsOptions tag.

If you use -i dac, Csound will connect with the system default. This is usually desirable.

If you want another audio device than the system default, you need to get the number of this device first. It should be written in the Csound console when Csound starts. If not, run Csound with this option:

```
csound --devices
```

Csound will print out the list of available devices.

Once you picked out the one you want to use, call it for instance as

```
-i adc2
```

Input and Output

Of course you can use both options together. To use the system default sound card, you will write:

```
-o dac -i adc
```

Or without spaces:

```
-odac -iadc
```

How can I choose different number of input and output channels?

The nchnls statement in the header of your Csound file will set the number of output channels. If you need another number of input channels, use the opcode nchnls_i. This statement opens 8 channels for output and 4 channels for input:

```
nchnls = 8  
nchnls_i = 4
```

REALTIME AUDIO SETTINGS

How can I synchronize the sample rate in Csound and in my audio card?

Say your system sample rate is 48000, but your Csound file has 44100. The solution is simple, and can be in both ways.

Either change the sample rate in the Csound header:

```
sr = 48000
```

Or change the system's sample rate to 44100. (Some systems will seamlessly handle the sampling rate requested by Csound and will not require any intervention.)

How can I set the audio buffer size?

The audio buffer is a space of memory between Csound and the sound card. It collects a number of audio samples before they really go out to the hardware, or come in from the hardware.

Csound has two options to set the realtime audio buffer size:

- -b followed by a number of samples sets the *Software Buffer Size*.
- -B followed by a number of samples sets the *Hardware Buffer Size*.

The meaning of these numbers depend on the audio module.¹

As a rule of thumb:

1. Set -b to a power-of-two, for instance 256.
2. Set -B to this number times four.

In this case, your CsOptions tag would contain:

```
-b 256 -B 1024
```

Which ksmmps should I use?

Always use a power-of-two. In my experience, 64 or 32 are good values for most cases.

Make sure you never have a larger ksmmps value than the -b buffer size.

Usually your ksmmps will be a fourth of -b. So as a standard configuration for real-time audio in Csound I'd use:

- -b 256 -B 1024 in the CsOptions tag.
- ksmmps = 64 in the Csound orchestra header, at the beginning of the CsInstruments tag.

Can I give realtime audio the priority over other processes in Csound?

When you have instruments that have substantial sections that could block out execution, for instance with code that loads buffers from files or creates big tables, you can try the option --realtime in you CsOptionstag.

¹As Victor Lazzarini explains (mail to Joachim Heintz, 19 march 2013), the role of -b and -B varies between the Audio Modules: "1. For portaudio, -B is only used to suggest a latency to the backend, whereas -b is used to set the actual buffersize. 2. For coreaudio, -B is used as the size of the internal circular buffer, and -b is used for the actual IO buffer size. 3. For jack, -B is used to determine the number of buffers used in conjunction with -b , num = (N + M + 1) / M. -b is the size of each buffer. 4. For alsalib, -B is the size of the buffer size, -b is the period size (a buffer is divided into periods). 5. For pulse, -b is the actual buffersize passed to the device, -B is not used. In other words, -B is not too significant in 1), not used in 5), but has a part to play in 2), 3) and 4), which is functionally similar."

This option will give your audio processing the priority over other tasks to be done. It places all initialisation code on a separate thread, and does not block the audio thread. Instruments start performing only after all the initialisation is done. That can have a side-effect on scheduling if your audio input and output buffers are not small enough, because the audio processing thread may “run ahead” of the initialisation one, taking advantage of any slack in the buffering.

Given that this option is intrinsically linked to low-latency, realtime audio performance, and also to reduce the effect on scheduling these other tasks, it is recommended that small ksmmps and buffer sizes, for example ksmmps=16, 32, or 64, -b32 or 64, and -B256 or 512.

REALTIME AUDIO ISSUES AND ERRORS

Why is my realtime audio distorted?

There is not one reason for it. Many different reasons can lead to distorted audio, for instance:

- A realtime audio module is selected which does not fit to your system.
- The buffer sizes are too large or too small.
- The Csound instruments you run consume too much CPU power.

Why do I have so much latency when I connect a microphone?

Probably you have a too large buffer size. See above about how to set the audio buffer size.

Why do I get a “wrong number of channels” error?

In general it means a mismatch between the nchnls statement in the Csound file and the available number of hardware channels. For instance nchnls = 4 in your Csound file will not work with a stereo sound card and using the portaudio module.

For Mac, this error can also be thrown when you use the internal sound card with nchnls = 2. The reason is that the microphone input is mono, but with nchnls = 2 Csound tries to open two input channels. The solution in this case is to use nchnls_i = 1 in addition:

```
nchnls = 2  
nchnls_i = 1
```

MIDI DEVICES

Do I have to handle MIDI via Csound when I use a frontend?

No. When you use Cabbage, CsoundQt or Blue, your MIDI devices will be connected with Csound via the frontends. You will find descriptions on the pages for [CsoundQt](#), [Cabbage](#) and [Blue](#).

How can I know the MIDI input device number when using plain Csound?

“Plain Csound” means: Using Csound via command line. If you use a frontend, MIDI handling is done by it.

You should see the device numbers in the Csound console once you run Csound.

If not, run Csound with

```
csound --midi-devices
```

Csound will return a list of all available MIDI input devices.

How can I know the MIDI output device number when using plain Csound?

“Plain Csound” means: Using Csound via command line. If you use a frontend, MIDI handling is done by it.

You should see the device numbers in the Csound console once you run Csound.

If not, run Csound with

```
csound --midi-devices
```

Csound will return a list of all available MIDI output devices.

How can I set another MIDI module than PortMidi when using plain Csound?

“Plain Csound” means: Using Csound via command line. If you use a frontend, MIDI handling is done by it.

The default MIDI module in Csound is [PortMidi](#). You can choose another MIDI module by using

```
-+rtmidi=...
```

The available MIDI modules depend on your operating system and on your installation. Usually [alsa](#) is available for Linux, [coremidi](#) for Mac and [winmme](#) for Windows.

How can I select MIDI input and output devices in plain Csound?

“Plain Csound” means: Using Csound via command line. If you use a frontend, MIDI handling is done by it.

Input devices are set via the option -M. Output devices are set with the option -Q. 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
```

Can I use more than one device for input using plain Csound?

“Plain Csound” means: Using Csound via command line. If you use a frontend, MIDI handling is done by it.

You can use multiple MIDI input devices when you use PortMidi. This is done via the option

```
-Ma
```

in the CsOptions tag.

How can I connect a MIDI keyboard with a Csound instrument?

Once you have 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](#) op-code. 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 02D01_Midi_Keypad.csd

```
<CsoundSynthesizer>
<CsOptions>
//it might be necessary to add -Ma here if you use plain Csound
-odac
</CsOptions>
<CsInstruments>
sr = 44100
ksmps = 64
nchnls = 2
```

```

0dbfs = 1

//assign all MIDI channels to instrument "Play"
massign(0,"Play")

instr Play
    //get the frequency from the key pressed
    iCps = cpsmidi()
    //get the amplitude
    iAmp = ampmidi(0dbfs * 0.3)
    //generate a sine tone with these parameters
    aSine = oscil:a(iAmp,iCps)
    //apply fade in and fade out
    aOut = linenr:a(aSine,0.01,0.1,0.01)
    //write it to the output
    outall(aOut)
endin

</CsInstruments>
<CsScore>
</CsScore>
</CsoundSynthesizer>
;Example by Andrés Cabrera and joachim heintz

```

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

How can I use a MIDI controller with plain Csound?

“Plain Csound” means: Using Csound via command line. If you use a frontend, it might route your MIDI controller to a widget so that you will deal with the widget values in your code rather than with the MIDI CC data.

To receive MIDI controller events in plain Csound, opcodes like `ctrl7` can be used. In the following example instrument 1 is turned on for 60 seconds. It will receive controller #1 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 02D02_Midi_Ctlin.csd

```

<CsoundSynthesizer>
<CsOptions>
//it might be necessary to add -Ma here if you use plain Csound
-odac
</CsOptions>
<CsInstruments>
sr = 44100
ksmps = 64
nchnls = 2
0dbfs = 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 = oscil:a(0.2, kFreq)

```

```
//output
outall(aOut)
endin

</CsInstruments>
<CsScore>
i 1 0 60
</CsScore>
</CsoundSynthesizer>
;Example by Andrés Cabrera
```

How can I get a simple printout of all MIDI input with plain Csound?

Csound can receive generic MIDI Data using the `midiin` opcode. The example below prints to the console the data received via MIDI.

EXAMPLE 02C03_Midi_all_in.csd

```
<CsoundSynthesizer>
<CsOptions>
//it might be necessary to add -Ma here if you use plain Csound
-odac
</CsOptions>
<CsInstruments>
sr = 44100
ksmps = 64
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>
i 1 0 -1
</CsScore>
</CsoundSynthesizer>
;Example by Andrés Cabrera
```


HOW TO: OPCODES

OVERVIEW

What are the essential opcodes? Do I need to know all opcodes?

I am sure that there is no expert user who knows all of the 1500+ opcodes. To give some help and advice about essential opcodes, have a look at these overviews:

- The [Opcodes Overview](#) in the Reference Manual.
- The [Opcode Guide](#) in this textbook.

OSCILLATORS

Why are there so many different oscillators in Csound?

In general, the large number of oscillators is an example for how Csound developed. You can call it “liberal”, or “anarchic”, or whatever, but as a matter of fact there was and is not a strict regulation about opcodes to be added. If a developer feels like something cannot be done with an existing opcode, he might rather write a new one instead of extending an existing one.

As to oscillators, the reasons to write new oscillators are: Efficiency and use cases. In the early time of computer music, each and every bit was valuable, so to say. The `oscil` opcode comes from this period and uses a very simple way to read a table, because it was fast and good enough for most use cases in the audible range. For other use cases, for instance LFO, other oscillators were added.

Which oscillator is the best?

I use `poscil` in this book, because it is a very precise oscillator. `oscili` and `oscil3` are good alternatives.

Why should `oscil` be used with caution?

The `oscil` opcode reads a table with `no` interpolation. Here comes a simple example in which the table only consists of eight numbers which are read once a second, so with a frequency of 1 Hz.

```
<CsoundSynthesizer>
<CsOptions>
</CsOptions>
<CsInstruments>
ksmps = 32

//create GEN02 table with numbers [0,1,2,3,4,3,2,1]
giTable = ftgen(0,0,8,-2, 0,1,2,3,4,3,2,1)

instr 1
//let oscil cross the table values once a second
a1 = oscil:a(1,1,giTable)
//print the result every 1/10 second
printks("Time = %.1f sec: Table value = %f\n", 1/10, times:k(), a1[0])
endin

</CsInstruments>
<CsScore>
i 1 0 2.01
</CsScore>
</CsoundSynthesizer>
```

The printout shows that `oscil` only returns the actual table values, without anything “in between”. But this “in between” is required in many cases, for instance when a sine table consists of 1024 points, and is read with a frequency of 100 Hz at a sample rate of 44100 Hz. The number of table values for one second is 102400, but we only have 44100 samples. So in this case 2.32... table values meet one sample, and this requires interpolation.

Whether this lack of interpolation leads to audible artefacts or not depends on the size of the table and the oscillator’s frequency. It can be audible for small tables and low frequencies. To avoid any possible artefacts, `oscili` or `poscil` should be used for linear interpolation, and `oscil3` or `poscil3` for cubic interpolation.

What is the difference between diskin and diskin2?

There is none any more. Just use diskin, and don't worry about messages which mention diskin2. They are the same (nowadays).

HOW TO: PRINT

Csound's print facilities are scattered amongst many different opcodes. You can print anything, but depending on the situation, you will need a particular print opcode. The following list is not complete, but may give some help to figure out the appropriate opcode for printing.

The main distinction is between printing at i-rate and printing at k-rate.

Printing at i-rate means: Printing only once, at the start of an instrument instance.

Printing at k-rate means: Printing during the performance of the instrument instance. Usually we will not print values every k-cycle because this would result in more than thousand printouts per second for a `sr = 44100` and `ksmps = 32`. So the different opcodes offer either a time interval or a trigger.

Printing at i-rate

Which opcodes can I use for basic i-rate printing?

Numbers: `print()`

`print` is for simple printing of i-rate numbers. Note that it rounds to three decimals.

```
iValue = 8.876543
print(iValue)
-> iValue = 8.877
```

Strings: `puts()`

`puts` prints a string on a new line:

```
String1 = "Hello ..."
String2 = "... newline!"
puts(String1,1)
puts(String2,1)
-> Hello ...
... newline!
```

Arrays: printarray()

printarray prints an i-rate array.

```
iArr[] genarray 0,5
printarray(iArr)
-> 0.0000 1.0000 2.0000 3.0000 4.0000 5.0000
```

Which opcodes can I use for formatted i-rate printing?**prints()**

prints offers formatting and can be used for both, numbers and strings.

```
iValue = 8.876543
String = "Hello"
prints("%s %f!\n",String,iValue)
-> Hello 8.876543!
```

printf_i()

printf_i has an additional trigger input. It prints only when the trigger is larger than zero. So this statement will do nothing:

```
iTrigger = 0
String = "Hello"
printf("%s %f!\n",iTrigger,String,iTrigger)
```

But this will print:

```
iTrigger = 1
String = "Hello"
printf("%s %f!\n",iTrigger,String,iTrigger)
-> Hello 1.000000!
```

Printing at k-rate**Which opcodes can I use for basic k-rate printing?****Numbers: printk() or printk2()**

printk is for simple printing of k-rate numbers (rounded to five decimals). The first input parameter specifies the time intervall in seconds between each printout. So this code will print the control cycle count once a second:

```
kValue = timek()
printk(1,kValue)
```

The printout for sr = 44100 and ksmmps = 32 shows:

```
i 1 time 0.00000: 1.00000
i 1 time 1.00063: 1380.00000
i 1 time 2.00127: 2759.00000
```

`printf2` is useful for printing numbers when they change.

```
instr 1
    kValue = randomh(0,1,1,3)
    printf2(kValue)
endin
schedule(1,0,2)
-> i1 0.88287
    i1 0.29134
```

Strings: puts()

`puts` prints a string on a new line. If the trigger input changes its value, the string is printed.

```
instr 1
//initialize string
String = "a"
//initialize trigger for puts()
kTrigger init 1
//printout when kTrigger is larger than zero and changed
puts(String,kTrigger)
//change string randomly and increase trigger
kValue = rnd:k(1)
if kValue > 0.999 then
    kTrigger += 1
    String = sprintfk("%c",random:k(65,90))
endif
endin
schedule(1,0,1)
-> a
    W
    H
    X
    D
    A
    N
```

Arrays: printarray()

The `printarray` opcode can also be used for k-rate arrays. It prints whenever the trigger input changes from 0 to 1.

```
instr 1
//create an array [0,1,2,3,4,5]
kArr[] genarray_i 0,5
//print the array values once a second
printarray(kArr,metro:k(1))
//change the array values randomly
kIndx = 0
while(kIndx < lenarray(kArr)) do
    kArr[kIndx] = rnd:k(6)
    kIndx += 1
od
endin
schedule(1,0,4)
-> 0.0000 1.0000 2.0000 3.0000 4.0000 5.0000
    0.7974 0.4855 3.4391 4.3606 5.9973 5.3945
```

```
5.9479 0.1663 5.6173 1.1320 5.9309 4.3610
5.6611 5.7748 5.8275 5.4023 2.3570 3.7158
```

Which opcodes can I use for formatted k-rate printing?

printks()

Similar to prints we can fill a format string. The time between print intervals is given as first parameter. So this will print once a second:

```
instr 1
    kValue = rnd:k(1)
    printk("Time = %f, kValue = %f\n",1,times:k(),kValue)
endin
schedule(1,0,4)
-> Time = 0.000726, kValue = 0.973500
    Time = 1.001361, kValue = 0.262336
    Time = 2.001995, kValue = 0.858091
    Time = 3.002630, kValue = 0.144621
```

printf()

printf works with a trigger, not with a fixed time interval between printouts.

```
instr 1
    kValue = rnd:k(1)
    if kValue > 0.999 then
        KTrigger = 1
    else
        KTrigger = 0
    endif
    printf("Time = %f, kValue = %f\n",KTrigger,times:k(),kValue)
endin
schedule(1,0,4)
-> Time = 0.380952, kValue = 0.999717
    Time = 0.383129, kValue = 0.999951
    Time = 2.535329, kValue = 0.999191
    Time = 3.274739, kValue = 0.999095
    Time = 3.4443810, kValue = 0.999773
    Time = 3.950295, kValue = 0.999182
```

Can I have simple a-rate printing?

If we want to print every sample, we can create an array of ksmmps length, then write the samples in it and print it via printarray. This is an example for ksmmps = 8 and a printout for every k-cycle over 0.001 seconds.

```
instr 1
    kArr[] init ksmmps
    aSine = poscil:a(1,1000)
    KIdx = 0
    while KIdx < ksmmps do
        kArr[KIdx] = aSine[KIdx]
```

```

    kIndx += 1
    od
    printarray(kArr,-1)
endin
schedule(1,0,.001)
-> 0.0000 0.1420 0.2811 0.4145 0.5396 0.6536 0.7545 0.8400
    0.9086 0.9587 0.9894 1.0000 0.9904 0.9607 0.9115 0.8439
    0.7591 0.6590 0.5455 0.4210 0.2879 0.1490 0.0071 -0.1349
    -0.2743 -0.4080 -0.5335 -0.6482 -0.7498 -0.8361 -0.9056 -0.9566
    -0.9883 -0.9999 -0.9913 -0.9626 -0.9144 -0.8477 -0.7638 -0.6644
    -0.5515 -0.4275 -0.2948 -0.1561 -0.0142 0.1279 0.2674 0.4015

```

Formatting

Which format specifiers can I use?

```

<CsoundSynthesizer>
<CsOptions>
-odac -m0
</CsOptions>
<CsInstruments>

    opcode ToAscii, S, S
        ;returns the ASCII numbers of the input string as string
    Sin      xin ;input string
    ilen     strlen   Sin ;its length
    ipos     =         0 ;set counter to zero
    Sres    =
    loop:   ;for all characters in input string:
    ichr    strchar   Sin, ipos ;get its ascii code number
    Snew    sprintf   "%d ", ichr ;put this number into a new string
    Sres    strcat    Sres, Snew ;append this to the output string
    loop_lt  ipos, 1, ilen, loop ;see comment for 'loop:'
    xout    Sres ;return output string
endop

    instr Integers
printf_i "\nIntegers:\n normal: %d or %d\n", 1, 123, -123
printf_i " signed if positive: %+d\n", 1, 123
printf_i " space left if positive:...% d...% d\n", 1, 123, -123
printf_i " fixed width left ...%-10d...or right...%10d\n", 1, 123, 123
printf_i " starting with zeros if ", 1
printf_i " necessary: %05d %05d %05d %05d %05d\n",
           1, 1, 12, 123, 1234, 12345, 123456
printf_i " floats are rounded: 1.1 -> %d, 1.9 -> %d\n", 1, 1.1, 1.9
    endin

    instr Floats
printf_i "\nFloats:\n normal: %f or %f\n", 1, 1.23, -1.23
printf_i " number of digits after point: %f %.5f %.3f %.1f\n",
           1, 1.23456789, 1.23456789, 1.23456789, 1.23456789
printf_i " space left if positive:...% .3f...% .3f\n", 1, 123, -123
printf_i " signed if positive: %+f\n", 1, 1.23
printf_i " fixed width left ...%-10.3f...or right...%10.3f\n",
           1, 1.23456, 1.23456

```

```
    endin

    instr Strings
printf_i "\nStrings:\n  normal: %s\n", 1, "csound"
printf_i "  fixed width left ...%-10s...or right...%10s\n",
        1, "csound", "csound"
printf_i {{  a string over
            multiple lines in which
                you can insert also quotes: "%s"\n}}, 1, "csound"
    endin

    instr Characters
printf_i "\nCharacters:\n  given as single strings: %s%s%s%s%s\n",
        1, "c", "s", "o", "u", "n", "d"
printf_i "  but can also be given as numbers: %c%c%c%c%c%c\n",
        1, 99, 115, 111, 117, 110, 100
Scsound ToAscii "csound"
printf_i "  in csound, the ASCII code of a character ", 1
printf_i "can be accessed with the opcode strchar.%s", 1, "\n"
printf_i "  the name 'csound' returns the numbers %s\n\n", 1, Scsound
    endin

</CsInstruments>
<CsScore>
i "Integers" 0 0
i "Floats" 0 0
i "Strings" 0 0
i "Characters" 0 0
</CsScore>
</CsoundSynthesizer>
```

03 A. INITIALIZATION AND PERFORMANCE PASS

Not only for beginners, but also for experienced Csound users, many problems result from the misunderstanding of the so-called i-rate and k-rate. You want Csound to do something just once, but Csound does it continuously. You want Csound to do something continuously, but Csound does it just once. If you experience such a case, you will most probably have confused i- and k-rate-variables.

The concept behind this is actually not complicated. But it is something which is more implicitly mentioned when we think of a program flow, whereas Csound wants to know it explicitly. So we tend to forget it when we use Csound, and we do not notice that we ordered a stone to become a wave, and a wave to become a stone. This chapter tries to explicate very carefully the difference between stones and waves, and how you can profit from them, after you understood and accepted both qualities.

Basic Distinction

We will explain at first the difference between *i-rate* and *k-rate*. Then we will look at some properties of k-rate signals, and finally introduce the audio vector.

Init Pass

Whenever a Csound instrument is called, all variables are set to initial values. This is called the initialization pass.

There are certain variables, which stay in the state in which they have been put by the init-pass. These variables start with an **i** if they are local (= only considered inside an instrument), or with a **gi** if they are global (= considered overall in the orchestra). This is a simple example:

EXAMPLE 03A01_Init-pass.csd

```
<CsoundSynthesizer>
<CsInstruments>
```

```
giGlobal    =          1/2
```

```

instr 1
iLocal      =      1/4
                print    giGlobal, iLocal
endin

instr 2
iLocal      =      1/5
                print    giGlobal, iLocal
endin

</CsInstruments>
<CsScore>
i 1 0 0
i 2 0 0
</CsScore>
</CsoundSynthesizer>
;example by joachim heintz

```

The output should include these lines:

```

SECTION 1:
new alloc for instr 1:
instr 1: giGlobal = 0.500  iLocal = 0.250
new alloc for instr 2:
instr 2: giGlobal = 0.500  iLocal = 0.200

```

As you see, the local variables *iLocal* do have different meanings in the context of their instrument, whereas *giGlobal* is known everywhere and in the same way. It is also worth mentioning that the performance time of the instruments (p3) is zero. This makes sense, as the instruments are called, but only the init-pass is performed.¹

Performance Pass

After having assigned initial values to all variables, Csound starts the actual performance. As music is a variation of values in time,² audio signals are producing values which vary in time. In all digital audio, the time unit is given by the sample rate, and one sample is the smallest possible time atom. For a sample rate of 44100 Hz,³ one sample comes up to the duration of $1/44100 = 0.0000227$ seconds.

So, performance for an audio application means basically: calculate all the samples which are finally being written to the output. You can imagine this as the cooperation of a clock and a calculator. For each sample, the clock ticks, and for each tick, the next sample is calculated.

Most audio applications do not perform this calculation sample by sample. It is much more efficient to collect some amount of samples in a *block* or *vector*, and calculate them all together. This means in fact, to introduce another internal clock in your application; a clock which ticks less frequently than the sample clock. For instance, if (always assumed your sample rate is 44100 Hz) your block size consists of 10 samples, your internal calculation time clock ticks every $1/44100 \times 10 = 0.000227 \times 10 = 0.00227$ seconds.

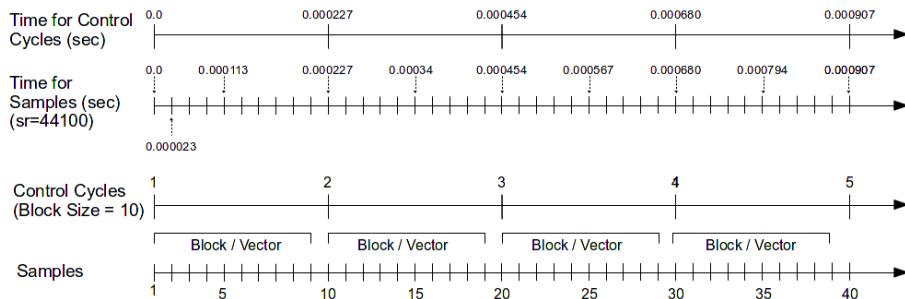
¹You would not get any other result if you set p3 to 1 or any other value, as nothing is done here except initialization.

²For the physical result which comes out of the loudspeakers or headphones, the variation is the variation of air pressure.

³44100 samples per second

(0.000227) seconds. If your block size consists of 441 samples, the clock ticks every 1/100 (0.01) seconds.

The following illustration shows an example for a block size of 10 samples. The samples are shown at the bottom line. Above are the control ticks, one for each ten samples. The top two lines show the times for both clocks in seconds. In the upmost line you see that the first control cycle has been finished at 0.000227 seconds, the second one at 0.000454 seconds, and so on.⁴



The rate (frequency) of these ticks is called the control rate in Csound. By historical reason,⁵ it is called *kontrol rate* instead of control rate, and abbreviated as *kr* instead of *cr*. Each of the calculation cycles is called a *k-cycle*. The block size or vector size is given by the *ksmps* parameter, which means: how many samples (*smps*) are collected for one *k-cycle*.⁶

Let us see some code examples to illustrate these basic contexts.

Implicit Incrementation

EXAMPLE 03A02_Perf-pass_incr.csd

```
<CsoundSynthesizer>
<CsInstruments>
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>
;example by joachim heintz
```

⁴These are by the way the times which Csound reports if you ask for the control cycles. The first control cycle in this example (sr=44100, ksmmps=10) would be reported as 0.00027 seconds, not as 0.00000 seconds.

⁵As Richard Boulanger explains, in early Csound a line starting with c was a comment line. So it was not possible to abbreviate control variables as cAnything (<http://csound.1045644.n5.nabble.com/OT-why-is-control-rate-called-kontrol-rate-td5720858.html#a5720866>).

⁶As the k-rate is directly depending on sample rate (sr) and ksmmps (kr = sr/ksmps), it is probably the best style to specify sr and ksmmps in the header, but not kr.

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
```

A counter (*kCount*) is set here to zero as initial value. Then, in each control cycle, the counter is increased by one. What we see here, is the typical behaviour of a loop. The loop has not been set explicitly, but works implicitly because of the continuous recalculation of all *k*-variables. So we can also speak about the *k*-cycles as an implicit (and time-triggered) *k*-loop.⁷ Try changing the *ksmps* value from 4410 to 8820 and to 2205 and observe the difference.

The next example reads the incrementation of *kCount* as rising frequency. The first instrument, called *Rise*, sets the *k*-rate frequency *kFreq* to the initial value of 100 Hz, and then adds 10 Hz in every new *k*-cycle. As *ksmps*=441, one *k*-cycle takes 1/100 second to perform. So in 3 seconds, the frequency rises from 100 to 3100 Hz. At the last *k*-cycle, the final frequency value is printed out.⁸ The second instrument, *Partials*, increments the counter by one for each *k*-cycle, but only sets this as new frequency for every 100 steps. So the frequency stays at 100Hz for one second, then at 200 Hz for one second, and so on. As the resulting frequencies are in the ratio 1 : 2 : 3 ..., we hear partials based on a 100 Hz fundamental, from the first partial up to the 31st. The opcode *printk2* prints out the frequency value whenever it has changed.

EXAMPLE 03A03_Perf-pass_incr_listen.csd

```
<CsoundSynthesizer>
<CsOptions>
-o dac
</CsOptions>
<CsInstruments>
sr = 44100
ksmps = 441
0dbfs = 1
nchnls = 2

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

instr Rise
kFreq     init      100
aSine     oscil     .2, kFreq, giSine
          outs      aSine, aSine
;increment frequency by 10 Hz for each k-cycle
kFreq     =           kFreq + 10
;print out the frequency for the last k-cycle
kLast     release
if kLast == 1 then
  printk    0, kFreq
```

⁷This must not be confused with a 'real' *k*-loop where inside one single *k*-cycle a loop is performed. See chapter 03C (section Loops) for examples.

⁸The value is 3110 instead of 3100 because it has already been incremented by 10.

```

    endif
  endin

  instr Partials
;initialize kCount
kCount      init      100
;get new frequency if kCount equals 100, 200, ...
  if kCount % 100 == 0 then
    kFreq      =      kCount
  endif
  aSine      poscil     .2, kFreq, giSine
              outs       aSine, aSine
;increment kCount
  kCount      =      kCount + 1
;print out kFreq whenever it has changed
  printk2    kFreq
  endin
</CsInstruments>
<CsScore>
i "Rise" 0 3
i "Partials" 4 31
</CsScore>
</CsoundSynthesizer>
;example by joachim heintz

```

Init versus Equals

A frequently occurring error is that instead of setting the k-variable as *kCount init 0*, it is set as *kCount = 0*. The meaning of both statements has one significant difference. *kCount init 0* sets the value for *kCount* to zero only in the init pass, without affecting it during the performance pass. *kCount = 0* sets the value for *kCount* to zero again and again, in each control cycle. So the increment always starts from the same point, and nothing really happens:

EXAMPLE 03A04_Perf-pass_no_incr.csd

```

<CsoundSynthesizer>
<CsInstruments>
sr = 44100
ksmps = 4410

instr 1
kcount      =      0; sets kcount to 0 at each k-cycle
kcount      =      kcount + 1; does not really increase ...
                printk    0, kcount; print the value
  endin

</CsInstruments>
<CsScore>
i 1 0 1
</CsScore>
</CsoundSynthesizer>
;example by joachim heintz

```

Outputs:

```
i 1 time 0.10000: 1.00000
i 1 time 0.20000: 1.00000
i 1 time 0.30000: 1.00000
i 1 time 0.40000: 1.00000
i 1 time 0.50000: 1.00000
i 1 time 0.60000: 1.00000
i 1 time 0.70000: 1.00000
i 1 time 0.80000: 1.00000
i 1 time 0.90000: 1.00000
i 1 time 1.00000: 1.00000
```

A Look at the Audio Vector

One k-cycle consists of `ksmps` audio samples. The single samples are processed in a block, called audio vector. If `ksmps=32`, for each audio signal 32 samples are processed in every k-cycle.

There are different opcodes to print out k-variables.⁹ There is no opcode in Csound to print out the audio vector directly, but we can use the `vaget` opcode to see what is happening inside one control cycle with the audio samples.

EXAMPLE 03A05_Audio_vector.csd

```
<CsoundSynthesizer>
<CsInstruments>
sr = 44100
ksmps = 5
0dbfs = 1

instr 1
aSine    poscil  1, 2205
kVec1    vaget    0, aSine
kVec2    vaget    1, aSine
kVec3    vaget    2, aSine
kVec4    vaget    3, aSine
kVec5    vaget    4, aSine
printks "kVec1 = %f, kVec2 = %f, kVec3 = %f, kVec4 = %f, kVec5 = %f\n",
        0, kVec1, kVec2, kVec3, kVec4, kVec5
endin
</CsInstruments>
<CsScore>
i 1 0 [1/2205]
</CsScore>
</CsoundSynthesizer>
;example by joachim heintz
```

The output shows these lines:

```
kVec1 = 0.000000, kVec2 = 0.309017, kVec3 = 0.587785, kVec4 =
0.809017, kVec5 = 0.951057
kVec1 = 1.000000, kVec2 = 0.951057, kVec3 = 0.809017, kVec4 =
0.587785, kVec5 = 0.309017
kVec1 = -0.000000, kVec2 = -0.309017, kVec3 = -0.587785, kVec4 =
-0.809017, kVec5 = -0.951057
kVec1 = -1.000000, kVec2 = -0.951057, kVec3 = -0.809017, kVec4 =
-0.587785, kVec5 = -0.309017
```

⁹See the manual page for `printk`, `printk2`, `printks`, `printf` to know more about the differences.

In this example, the number of audio samples in one k-cycle is set to five by the statement `ksmps=5`. The first argument to `vaget` specifies which sample of the block you get. For instance,

```
kVec1      vaget      0, aSine
```

gets the first value of the audio vector and writes it into the variable `kVec1`. For a frequency of 2205 Hz at a sample rate of 44100 Hz, you need 20 samples to write one complete cycle of the sine. So we call the instrument for 1/2205 seconds, and we get 4 k-cycles. The printout shows exactly one period of the sine wave.

At the end of this chapter we will show another and more advanced method to access the audio vector and modify its samples.

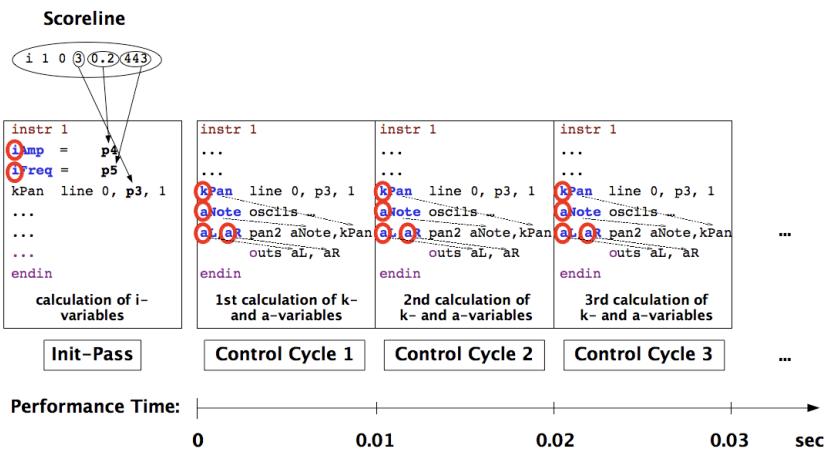
A Summarizing Example

After having put so much attention to the different single aspects of initialization, performance and audio vectors, the next example tries to summarize and illustrate all the aspects in their practical mixture.

EXAMPLE 03A06_Init_perf_audio.cs

```
<CsoundSynthesizer>
<CsOptions>
-o dac
</CsOptions>
<CsInstruments>
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     poscil iAmp, iFreq ;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>
;example by joachim heintz
```

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



Applications and Concepts

We will look now at some applications and consequences of what has been showed. We will see how we can use a k-variable at i-time. Then we will at k-signals in an instrument which is called several times. We will explain the concept of re-initialization and have a look at instruments: in which order they are processed, how named instruments work, and how we can use fractional instrument numbers.

Accessing the Initialization Value of a k-Variable

It has been said that the init pass sets initial values to all variables. It must be emphasized that this indeed concerns all variables, not only the i-variables. It is only the matter that i-variables are not affected by anything which happens later, in the performance. But also k- and a-variables get their initial values.

As we saw, the init opcode is used to set initial values for k- or a-variables explicitly. On the other hand, you can get the initial value of a k-variable which has not been set explicitly, by the `i()` facility. This is a simple example:

EXAMPLE 03A07_Init-values_of_k-variables.csd

```
<CsoundSynthesizer>
<CsOptions>
-o dac
</CsOptions>
<CsInstruments>
instr 1
gkLine line 0, p3, 1
endin
instr 2
iInstr2LineValue = i(gkLine)
print iInstr2LineValue
endin
instr 3
```

```
iInstr3LineValue = i(gkLine)
print iInstr3LineValue
endin
</CsInstruments>
<CsScore>
i 1 0 5
i 2 2 0
i 3 4 0
</CsScore>
</CsoundSynthesizer>
;example by joachim heintz
```

Outputs:

```
new alloc for instr 1:
B 0.000 .. 2.000 T 2.000 TT 2.000 M: 0.0
new alloc for instr 2:
instr 2: iInstr2LineValue = 0.400
B 2.000 .. 4.000 T 4.000 TT 4.000 M: 0.0
new alloc for instr 3:
instr 3: iInstr3LineValue = 0.800
B 4.000 .. 5.000 T 5.000 TT 5.000 M: 0.0
```

Instrument 1 produces a rising k-signal, starting at zero and ending at one, over a time of five seconds. The values of this line rise are written to the global variable *gkLine*. After two seconds, instrument 2 is called, and examines the value of *gkLine* at its init-pass via **i(gkLine)**. The value at this time (0.4), is printed out at init-time as *iInstr2LineValue*. The same happens for instrument 3, which prints out **iInstr3LineValue = 0.800**, as it has been started at 4 seconds.

The *i()* feature is particularly useful if you need to examine the value of any control signal from a widget or from midi, at the time when an instrument starts.

For getting the init value of an element in a k-time array, the syntax *i(kArray,iIndex)* must be used; for instance *i(kArr,0)* will get the first element of array *kArr* at init-time. More about this in the section *Init Values of k-Arrays* in the [Arrays](#) chapter of this book.

k-Values and Initialization in Multiple Triggered Instruments

What happens on a k-variable if an instrument is called multiple times? What is the initialization value of this variable on the first call, and on the subsequent calls?

If this variable is not set explicitly, the init value in the first call of an instrument is zero, as usual. But, for the next calls, the k-variable is initialized to the value which was left when the previous instance of the same instrument turned off.

The following example shows this behaviour. Instrument *Call* simply calls the instrument *Called* once a second, and sends the number of the call to it. Instrument *Called* generates the variable *kRndVal* by a random generator, and reports both:

- the value of *kRndVal* at initialization, and
- the value of *kRndVal* at performance time, i.e. the first control cycle. (After the first k-cycle, the instrument is turned off immediately.)

EXAMPLE 03A08_k-init_in_multiple_calls_1.csd

```

<CsoundSynthesizer>
<CsOptions>
-nm0
</CsOptions>
<CsInstruments>
ksmps = 32

    instr Call
kNumCall init 1
kTrig metro 1
if kTrig == 1 then
    event "i", "Called", 0, 1, kNumCall
    kNumCall += 1
endif
endin

    instr Called
iNumCall = p4
kRndVal random 0, 10
prints "Initialization value of kRnd in call %d = %.3f\n",
    iNumCall, i(kRndVal)
printks " New random value of kRnd generated in call %d = %.3f\n",
    0, iNumCall, kRndVal
turnoff
endin

</CsInstruments>
<CsScore>
i "Call" 0 3
</CsScore>
</CsoundSynthesizer>
;example by joachim heintz

```

The output should show this:

```

Initialization value of kRnd in call 1 = 0.000
    New random value of kRnd generated in call 1 = 8.829
Initialization value of kRnd in call 2 = 8.829
    New random value of kRnd generated in call 2 = 2.913
Initialization value of kRnd in call 3 = 2.913
    New random value of kRnd generated in call 3 = 9.257

```

The printout shows what was stated before: If there is no previous value of a k-variable, this variable is initialized to zero. If there is a previous value, it serves as initialization value.

Note that this is *exactly* the same for User-Defined Opcodes! If you call a UDO twice, it will have the current value of a k-Variable of the first call as init-value of the second call, unless you initialize the k-variable explicitly by an init statement.

The final example shows both possibilities, using explicit initialization or not, and the resulting effect.

EXAMPLE 03A10_k-init_in_multiple_calls_3.csd

```

<CsoundSynthesizer>
<CsOptions>
-nm0
</CsOptions>
<CsInstruments>
ksmps = 32

```

```

instr without_init
prints "instr without_init, call %d:\n", p4
kVal = 1
prints " Value of kVal at initialization = %d\n", i(kVal)
printks " Value of kVal at first k-cycle = %d\n", 0, kVal
kVal = 2
turnoff
endin

instr with_init
prints "instr with_init, call %d:\n", p4
kVal init 1
kVal = 1
prints " Value of kVal at initialization = %d\n", i(kVal)
printks " Value of kVal at first k-cycle = %d\n", 0, kVal
kVal = 2
turnoff
endin

</CsInstruments>
<CsScore>
i "without_init" 0 .1 1
i "without_init" + .1 2
i "with_init" 1 .1 1
i "with_init" + .1 2
</CsScore>
</CsoundSynthesizer>
;example by joachim heintz

```

This is the output:

```

instr without_init, call 1:
    Value of kVal at initialization = 0
    Value of kVal at first k-cycle = 1
instr without_init, call 2:
    Value of kVal at initialization = 2
    Value of kVal at first k-cycle = 1
instr with_init, call 1:
    Value of kVal at initialization = 1
    Value of kVal at first k-cycle = 1
instr with_init, call 2:
    Value of kVal at initialization = 1
    Value of kVal at first k-cycle = 1

```

Note that this characteristics of using *leftovers* from previous instances which may lead to undesired effects, does also occur for audio variables. Similar to k-variables, an audio vector is initialized for the first instance to zero, or to the value which is explicitly set by an *init* statement. In case a previous instance can be re-used, its last state will be the *init* state of the new instance.

The next example shows an undesired side effect in instrument 1. In the third call (*start*=2), the previous values for the *a1* audio vector will be used, because this variable is not set explicitly. This means, though, that 32 amplitudes are repeated in a frequency of *sr/ksmps*, in this case 44100/32 = 1378.125 Hz. The same happens at *start*=4 with audio variable *a2*. Instrument 2 initializes *a1* and *a2* in the cases they need to be, so that the inadvertend tone disappears.

EXAMPLE 03A11_a_inits_in_multiple_calls.csd

```

<CsoundSynthesizer>
<CsOptions>
```

```

-odac
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32 ;try 64 or other values
nchnls = 2
0dbfs = 1

instr 1 ;without explicit init
i1 = p4
if i1 == 0 then
a1 poscil 0.5, 500
endif
if i1 == 1 then
a2 poscil 0.5, 600
endif
outs a1, a2
endin

instr 2 ;with explicit init
i1 = p4
if i1 == 0 then
a1 poscil 0.5, 500
a2 init 0
endif
if i1 == 1 then
a2 poscil 0.5, 600
a1 init 0
endif
outs a1, a2
endin

</CsInstruments>
<CsScore>
i 1 0 .5 0
i . 1 . 0
i . 2 . 1
i . 3 . 1
i . 4 . 0
i . 5 . 0
i . 6 . 1
i . 7 . 1
b 9
i 2 0 .5 0
i . 1 . 0
i . 2 . 1
i . 3 . 1
i . 4 . 0
i . 5 . 0
i . 6 . 1
i . 7 . 1
</CsScore>
</CsoundSynthesizer>
;example by oeyvind brandtsegg and joachim heintz

```

Reinitialization

As we saw above, an i-value is not affected by the performance loop. So you cannot expect this to work as an incrementation:

EXAMPLE 03A12_Init_no_incr.csd

```
<CsoundSynthesizer>
<CsInstruments>
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>
;example by joachim heintz
```

The output is nothing but:

```
instr 1:  iCount = 1.000
```

But you can advise Csound to repeat the initialization of an i-variable. 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 03A13_Re-init.csd

```
<CsoundSynthesizer>
<CsInstruments>
sr = 44100
ksmps = 4410

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

</CsInstruments>
<CsScore>
i 1 0 1
</CsScore>
</CsoundSynthesizer>
;example by joachim heintz
```

Outputs:

```

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

```

What happens here more in detail, is the following. In the actual init-pass, *iCount* is set to zero via *iCount init 0*. Still in this init-pass, it is incremented by one (*iCount = iCount+1*) and the value is printed out as *iCount = 1.000*. Now starts the first performance pass. The statement *reinit new* advises Csound to initialise again the section labeled as *new*. So the statement *iCount = iCount + 1* is executed again. As the current value of *iCount* at this time is 1, the result is 2. So the printout at this first performance pass is *iCount = 2.000*. The same happens in the next nine performance cycles, so the final count is 11.

Order of Calculation

In this context, it can be very important to observe the order in which the instruments of a Csound orchestra are evaluated. This order is determined 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 03A14_Order_of_calc.csd

```

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

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 1.40000: 3.00000
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
```

Instrument 10 can use the values which instrument 1 has produced in the same control cycle, but it can only refer to values of instrument 100 which are produced in the previous control cycle. By this reason, the printout shows values which are one less in the latter case.

Named Instruments

Instead of a number you can also use a name for an instrument. This is mostly preferable, because you can give meaningful names, leading to a better readable code. But what about the order of calculation in named instruments?

The answer is simple: Csound calculates them in the same order as they are written in the orchestra. So if your instrument collection is like this ...

EXAMPLE 03A15_Order_of_calc_named.csd

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

```

instr Grain_machine
prints " Grain_machine\n"
endin

instr Fantastic_FM
prints " Fantastic_FM\n"
endin

instr Random_Filter
prints " Random_Filter\n"
endin

instr Final_Reverb
prints " Final_Reverb\n"
endin

</CsInstruments>
<CsScore>
i "Final_Reverb" 0 1
i "Random_Filter" 0 1
i "Grain_machine" 0 1
i "Fantastic_FM" 0 1
</CsScore>
</CsoundSynthesizer>
;example by joachim heintz

```

... you can count on this output:

```

new alloc for instr Grain_machine:
    Grain_machine
new alloc for instr Fantastic_FM:
    Fantastic_FM
new alloc for instr Random_Filter:
    Random_Filter
new alloc for instr Final_Reverb:
    Final_Reverb

```

Note that the score has not the same order. But internally, Csound transforms all names to numbers, in the order they are written from top to bottom. The numbers are reported on the top of Csound's output.¹⁰

```

instr Grain_machine uses instrument number 1
instr Fantastic_FM uses instrument number 2
instr Random_Filter uses instrument number 3
instr Final_Reverb uses instrument number 4

```

Instruments with Fractional Numbers

Sometimes we want to call several instances of an instrument, but we want to treat each instance different. For this, Csound provides the possibility of fractional note numbers. In the following example, instr 1 shows a basic example, turning on and off certain instances in the score. (Turning off is done here by negative note numbers.) Instr *Play* is a bit more complicated in using the instance number as index to a global array. Instr *Trigger* calls this instrument several times with fractional numbers. It also shows how we can use fractional numbers for named instruments: We

¹⁰If you want to know the number in an instrument, use the nstrnum opcode.

first get the number which Csound appointed to this instrument (using the `nstrnum` opcode), and then add the fractional part (0, 0.1, 0.2 etc) to it.

EXAMPLE 03A16_FractionalInstrNums.csd

```

<CsoundSynthesizer>
<CsOptions>
-odac -m128
</CsOptions>
<CsInstruments>
sr = 44100
nchnls = 2
0dbfs = 1
ksmps = 32
seed 0

giArr[] fillarray 60, 68, 67, 66, 65, 64, 63

instr 1
iMidiNote = p4
iFreq mtof iMidiNote
aPluck pluck .1, iFreq, iFreq, 0, 1
aOut linenr aPluck, 0, 1, .01
out aOut, aOut
endin

instr Trigger
index = 0
while index < lenarray(giArr) do
  iInstrNum = nstrnum("Play") + index/10
  schedule(iInstrNum, index+random:i(0,.5),5)
  index += 1
od
endin

instr Play
iIndx = frac(p1)*10 //index is fractional part of instr number
iFreq = mtof:i(giArr[round(iIndx)])
aPluck pluck .1, iFreq, iFreq, 0, 1
aOut linenr aPluck, 0, 1, .01
out aOut, aOut
endin

</CsInstruments>
<CsScore>
//traditional score
t 0 90
i 1.0 0 -1 60
i 1.1 1 -1 65
i 1.2 2 -1 55
i 1.3 3 -1 70
i 1.4 4 -1 50
i 1.5 5 -1 75

i -1.4 7 1 0
i -1.1 8 1 0
i -1.5 9 1 0
i -1.0 10 1 0
i -1.3 11 1 0
i -1.2 12 1 0

//event generating instrument

```

```
i "Trigger" 15 10
</CsScore>
</CsoundSynthesizer>
;example by joachim heintz
```

Tips for Pratical Use

The last part of this chapter focusses on some situations which are known as stumbling blocks by many users. We will start with a discussion about *i-time* and *k-rate* opcodes, and when to use either of them. In between we will look at some possible issues with the *k-rate* ticks as internal time units. We will have another look at the audio vector, before we try to throw some light in the complicated matter of hidden initialization. Finally, we will give some general suggestions when to choose *i-rate* or *k-rate* opcodes.

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, (s)he thinks: *OK - print it out*. But Csound replies: *Please, tell me first if you want to print an i- or a k-variable.*¹¹

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:

```
iros      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

¹¹See the following section 03B about the variable types for more on this subject.

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 03A17_Random_at_ika.csd

```

<CsoundSynthesizer>
<CsOptions>
-odac
</CsOptions>
<CsInstruments>
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    oscil    aAmp, iPch, giSine
        outs      aSine, aSine
endin

instr 2 ;k-rate random: noisy
kPch     random  300, 600
aAmp    linseg   .5, p3, 0
aSine    oscil    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    oscil    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>
;example by joachim heintz

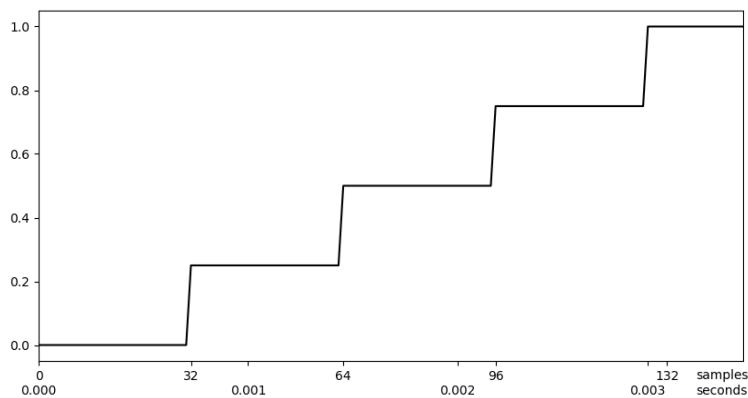
```

Possible Problems with k-Rate Tick Size

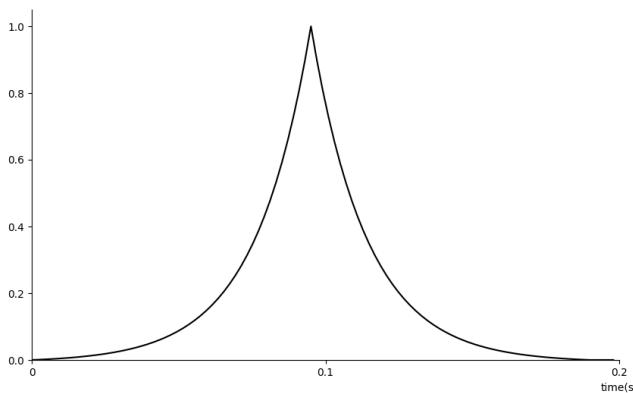
It has been said that usually the k-rate clock ticks much slower than the sample (a-rate) clock. For a common size of `ksmps=32`, one k-value remains the same for 32 samples. This can lead to problems, for instance if you use k-rate envelopes. Let us assume that you want to produce a very short fade-in of 3 milliseconds, and you do it with the following line of code:

```
kFadeIn linseg 0, .003, 1
```

Your envelope will look like this:



Such a *staircase-envelope* is what you hear in the next example as zipper noise. The `transeg` opcode produces a non-linear envelope with a sharp peak:



The rise and the decay are each 1/10 seconds long. If this envelope is produced at k-rate with a blocksize of 128 (instr 1), the noise is clearly audible. Try changing `ksmps` to 64, 32 or 16 and compare the amount of zipper noise. – Instrument 2 uses an envelope at audio-rate instead. Regardless the blocksize, each sample is calculated separately, so the envelope will always be smooth. – Instrument 3 shows a remedy for situations in which a k-rate envelope cannot be avoided: the [a\(\)](#) will convert the k-signal to audio-rate by interpolation thus smoothing the envelope.

EXAMPLE 03A18_Zipper.csd

```

<CsoundSynthesizer>
<CsOptions>
-o dac
</CsOptions>
<CsInstruments>
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    oscil   .5, 800
kEnv     transeg 0, .1, 5, 1, .1, -5, 0
aOut      =       aSine * kEnv
           outs    aOut, aOut
endin

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

instr 3 ;envelope at k-time with a-time interpolation
aSine    oscil   .5, 800
kEnv     transeg 0, .1, 5, 1, .1, -5, 0
aOut      =       aSine * a(kEnv)
           outs    aOut, aOut
endin

</CsInstruments>
<CsScore>
r 3 ;repeat the following line 3 times
i 1 0 1
s ;end of section
r 3
i 2 0 1
s
r 3
i 3 0 1
</CsScore>
</CsoundSynthesizer>
;example by joachim heintz

```

Time Impossible

There are two internal clocks in Csound. The sample rate (sr) determines the audio-rate, whereas the control rate (kr) determines the rate, in which a new control cycle can be started and a new block of samples can be performed. In general, Csound can not start any event in between two control cycles, nor end.

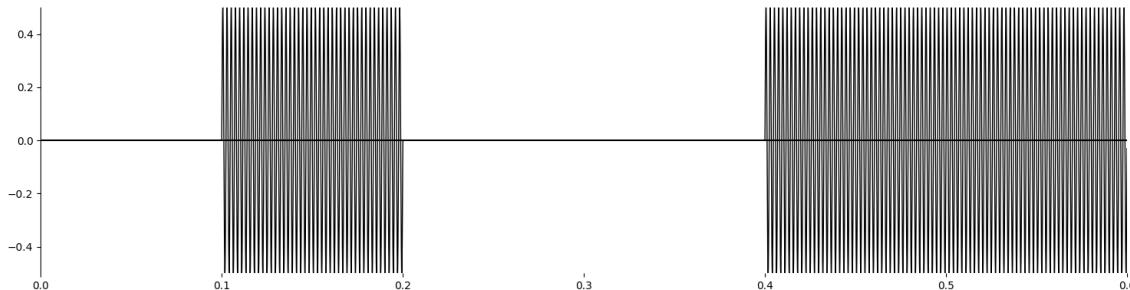
The next example chooses an extreme small control rate (only 10 k-cycles per second) to illustrate this.

EXAMPLE 03A19_Time_Impossible.csd

```
<CsoundSynthesizer>
<CsOptions>
-o test.wav -d
</CsOptions>
<CsInstruments>
sr = 44100
ksmps = 4410
nchnls = 1
0dbfs = 1

instr 1
aPink poscil .5, 430
out aPink
endin
</CsInstruments>
<CsScore>
i 1 0.05 0.1
i 1 0.4 0.15
</CsScore>
</CsoundSynthesizer>
;example by joachim heintz
```

The first call advices instrument 1 to start performance at time 0.05. But this is impossible as it lies between two control cycles. The second call starts at a possible time, but the duration of 0.15 again does not coincide with the control rate. So the result starts the first call at time 0.1 and extends the second call to 0.2 seconds:

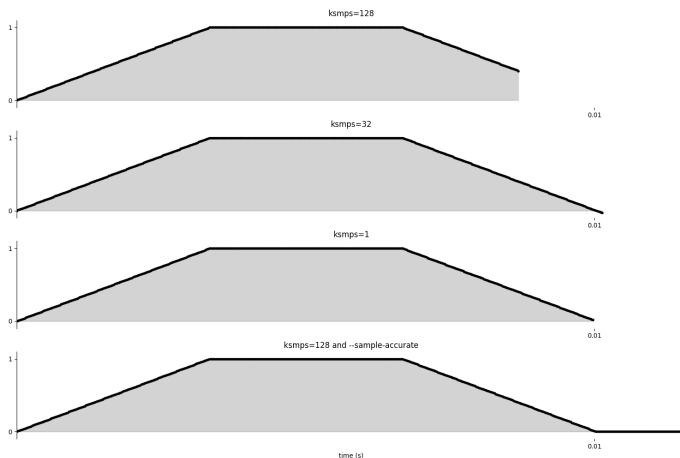


With Csound6, the possibilities of these *in between* are enlarged via the `-sample-accurate` option. The next image shows how a 0.01 second envelope which is generated by the code

```
a1 init 1
a2 linen a1, p3/3, p3, p3/3
out a2
```

(and a call of 0.01 seconds at `sr=44100`) shows up in the following cases:

1. `ksmps=128`
2. `ksmps=32`
3. `ksmps=1`
4. `ksmps=128` and `-sample-accurate` enabled



This is the effect:

1. At `ksmps=128`, the last section of the envelope is missing. The reason is that, at `sr=44100` Hz, 0.01 seconds contain 441 samples. 441 samples divided by the block size (`ksmps`) of 128 samples yield to 3.4453125 blocks. This is rounded to 3. So only $3 * 128 = 384$ Samples are performed. As you see, the envelope itself is calculated correctly in its shape. It *would* end exactly at 0.01 seconds .. but it does not, because the `ksmps` block ends too early. So this envelope might introduce a click at the end of this note.
2. At `ksmps=32`, the number of samples (441) divided by `ksmps` yield to a value of 13.78125. This is rounded to 14, so the rendered audio is slightly longer than 0.01 seconds (448 samples).
3. At `ksmps=1`, the envelope is as expected.
4. At `ksmps=128` and `--sample-accurate` enabled, the envelope is correct, too. Note that the section is now $4 * 128 = 512$ samples long, but the envelope is more accurate than at `ksmps=32`.

So, in case you experience clicks at very short envelopes although you use a-rate envelopes, it might be necessary to set either `ksmps=1`, or to enable the `--sample-accurate` option.

Yet another Look at the Audio Vector

In Csound 6 it is actually possible to access each sample of the audio vector directly, without setting `ksmps=1`. This feature, however, requires some knowledge about arrays and loops, so beginners should skip this paragraph.

The direct access uses the `a...[]` syntax which is common in most programming languages for arrays or lists. As an audio vector is of `ksmps` length, we must iterate in each k-cycle over it. By this, we can both, get and modify the values of the single samples directly. Moreover, we can use control structures which are usually k-rate only also at a-rate, for instance any condition depending on the value of a single sample.

The next example demonstrates three different usages of the sample-by-sample processing. In the *SimpleTest* instrument, every single sample is multiplied by a value (1, 3 and -1). Then the result is added to the previous sample value. This leads to amplification for `iFac=3` and to silence for `iFac=-1` because in this case every sample cancels itself. In the *PrintSamplelf* instrument, each sample which is between 0.99 and 1.00 is printed to the console. Also in the *PlaySamplelf* instrument an if-condition is applied on each sample, but here not for printing rather than playing out only the

samples whichs values are between 0 and 1/10000. They are then multiplied by 10000 so that not only rhythm is irregular but also volume.

EXAMPLE 03A20_Sample_by_sample_processing.csd

```

<CsoundSynthesizer>
<CsOptions>
-odac -m128
</CsOptions>
<CsInstruments>
sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

instr SimpleTest

iFac = p4 ;multiplier for each audio sample

aSinus oscil 0.1, 500

kIndx = 0
while kIndx < ksmpls do
  aSinus[kIndx] = aSinus[kIndx] * iFac + aSinus[kIndx]
  kIndx += 1
od

out aSinus, aSinus

endin

instr PrintSampleIf

aRnd rnd31 1, 0, 1

kBlkCnt init 0
kSmpCnt init 0

kIndx = 0
while kIndx < ksmpls do
  if aRnd[kIndx] > 0.99 then
    printf "Block = %2d, Sample = %4d, Value = %f\n",
           kSmpCnt, kBlkCnt, kSmpCnt, aRnd[kIndx]
  endif
  kIndx += 1
  kSmpCnt += 1
od

kBlkCnt += 1

endin

instr PlaySampleIf

aRnd rnd31 1, 0, 1
aOut init 0

kBlkCnt init 0
kSmpCnt init 0

kIndx = 0
while kIndx < ksmpls do

```

```

if aRnd[kIndx] > 0 && aRnd[kIndx] < 1/10000 then
    aOut[kIndx] = aRnd[kIndx] * 10000
else
    aOut[kIndx] = 0
endif
kIndx += 1
od

out aOut, aOut

endin

</CsInstruments>
<CsScore>
i "SimpleTest" 0 1 1
i "SimpleTest" 2 1 3
i "SimpleTest" 4 1 -1
i "PrintSampleIf" 6 .033
i "PlaySampleIf" 8 10
</CsScore>
</CsoundSynthesizer>
;example by joachim heintz

```

The output should contain these lines, generated by the *PrintSampleIf* instrument, showing that in block 40 there were two subsequent samples which fell under the condition:

```

Block = 2, Sample = 86, Value = 0.998916
Block = 7, Sample = 244, Value = 0.998233
Block = 19, Sample = 638, Value = 0.995197
Block = 27, Sample = 883, Value = 0.990801
Block = 34, Sample = 1106, Value = 0.997471
Block = 40, Sample = 1308, Value = 1.000000
Block = 40, Sample = 1309, Value = 0.998184
Block = 43, Sample = 1382, Value = 0.994353

```

At the end of chapter 03G an example is shown for a more practical use of sample-by-sample processing in Csound: to implement a digital filter as user defined opcode.

Hidden Initialization of k- and S-Variables

Any k-variable can be explicitly initialized by the *init* opcode, as has been shown above. But internally any variable, it be control rate (k), audio rate (a) or string (S), has an initial value. As this initialization can be hidden from the user, it can lead to unexpexted behaviour.

Explicit and implicit initialization

The first case is easy to understand, although some results may be unexpected. Any k-variable which is not explicitly initialized is set to zero as initial value.

EXAMPLE 03A21_Init_explicit_implicit.csd

```

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

```

```

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

instr 1

;explicit initialization
k_Exp init 10
S_Exp init "goodbye"

;implicit initialization
k_Imp linseg 10, 1, 0
S_Imp strcpyk "world"

;print out at init-time
prints "k_Exp -> %d\n", k_Exp
printf_i "S_Exp -> %s\n", 1, S_Exp
prints "k_Imp -> %d\n", k_Imp
printf_i "S_Imp -> %s\n", 1, S_Imp

endin

</CsInstruments>
<CsScore>
i 1 0 1
</CsScore>
</CsoundSynthesizer>
;example by joachim heintz

```

This is the console output:

```

k_Exp -> 10
S_Exp -> goodbye
k_Imp -> 0
S_Imp -> world

```

The implicit output may be of some surprise. The variable *k_Imp* is *not* initialized to 10, although 10 will be the first value during performance. And *S_Imp* carries the *world* already at initialization although the opcode name *strcpyk* may suggest something else. But as the manual page states: *strcpyk does the assignment both at initialization and performance time.*

Order of initialization statements

What happens if there are two init statements, following each other? Usually the second one overwrites the first. But if a k-value is explicitly set via the *init* opcode, the implicit initialization will not take place.

EXAMPLE 03A22_Init_overwrite.csd

```

<CsoundSynthesizer>
<CsOptions>
-nm0
</CsOptions>
</CsInstruments>

sr = 44100
ksmps = 32

```

```

nchnls = 2
0dbfs = 1

instr 1

;k-variables
k_var init 20
k_var linseg 10, 1, 0

;string variables
S_var init "goodbye"
S_var strcpyk "world"

;print out at init-time
prints "k_var -> %d\n", k_var
printf_i "S_var -> %s\n", 1, S_var

endin

</CsInstruments>
<CsScore>
i 1 0 1
</CsScore>
</CsoundSynthesizer>
;example by joachim heintz

```

The output is:

```

k_var -> 20
S_var -> world

```

Both pairs of lines in the code look similar, but do something quite different. For *k_var* the line *k_var linseg 10, 1, 0* will not initialize *k_var* to zero, as this happens only if no init value is assigned. The line *S_var strcpyk "world"* instead does an explicit initialization, and this initialization will overwrite the preceding one. If the lines were swapped, the result would be *goodbye* rather than *world*.

Hidden initialization in k-rate if-clause

If-clauses can be either i-rate or k-rate. A k-rate if-clause initializes nevertheless. Reading the next example may suggest that the variable *String* is only initialized to "yes", because the if-condition will never become true. But regardless it is true or false, any k-rate if-clause initializes its expressions, in this case the *String* variable.

EXAMPLE 03A23_Init_hidden_in_if.csd

```

<CsoundSynthesizer>
<CsOptions>
-m128
</CsOptions>
<CsInstruments>
sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

instr 1

```

```

;a string to be copied at init- and performance-time
String strcpyk "yes!\n"

;print it at init-time
printf_i "INIT 1: %s", 1, String

;a copy assignment that will never become true during performance
kBla = 0
if kBla == 1 then
  String strcpyk "no!\n"
endif

;nevertheless the string variable is initialized by it
printf_i "INIT 2: %s", 1, String

;during performance only "yes!" remains
printf "PERF %d: %s", timeinstk(), timeinstk(), String

;turn off after three k-cycles
if timeinstk() == 3 then
  turnoff
endif

endin

</CsInstruments>
<CsScore>
i 1 0 1
</CsScore>
</CsoundSynthesizer>
;example by joachim heintz

```

Returns:

```

INIT 1: yes!
INIT 2: no!
PERF 1: yes!
PERF 2: yes!
PERF 3: yes!

```

If you want to skip the initialization at this point, you can use an *igoto* statement:

```

if kBla == 1 then
  igoto skip
  String strcpyk "no!\n"
skip:
endif

```

Hidden initialization via UDOs

A user may expect that a UDO will behave identical to a csound native opcode, but in terms of implicit initialization it is not the case. In the following example, we may expect that instrument 2 has the same output as instrument 1.

EXAMPLE 03A24_Init_hidden_in_udo.csd

```

<CsoundSynthesizer>
<CsOptions>
-m128
</CsOptions>

```

```

<CsInstruments>

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

    opcode RndInt, k, kk
kMin, kMax xin
kRnd random kMin, kMax+.999999
kRnd = int(kRnd)
xout kRnd
    endop

instr 1 ;opcode

kBla init 10
kBla random 1, 2
prints "instr 1: kBla initialized to %d\n", i(kBla)
turnoff

endin

instr 2 ;udo has different effect at i-time

kBla init 10
kBla RndInt 1, 2
prints "instr 2: kBla initialized to %d\n", i(kBla)
turnoff

endin

instr 3 ;but the functional syntax makes it different

kBla init 10
kBla = RndInt(1, 2)
prints "instr 3: kBla initialized to %d\n", i(kBla)
turnoff

endin

</CsInstruments>
<CsScore>
i 1 0 .1
i 2 + .
i 3 + .
</CsScore>
</CsoundSynthesizer>
;example by joachim heintz

```

But the output is:

```

instr 1: kBla initialized to 10
instr 2: kBla initialized to 0
instr 3: kBla initialized to 10

```

The reason that instrument 2 does implicit initialization of *kBla* is written in the manual page for *xin / xout*: *These opcodes actually run only at i-time*. In this case, *kBla* is initialized to zero, because the *kRnd* variable inside the UDO is implicitly zero at init-time.

Instrument 3, on the other hand, uses the *=* operator. It works as other native opcodes: if a *k*-variable has an explicit init value, it does not initialize again.

The examples about hidden (implicit) initialization may look somehow sophisticated and far from normal usage. But this is not the case. As users we may think: "I perform a line from 10 to 0 in 1 second, and i write this in the variable kLine. So i(kLine) is 10." It is not, and if you send this value at init-time to another instrument, your program will give wrong output.

When to Use i- or k- Rate

When you code on your Csound instrument, you may sometimes wonder whether you shall use an i-rate or a k-rate opcode. From what is said, the general answer is clear: Use i-rate if something has to be done only once, or in a somehow punctual manner. Use k-rate if something has to be done continuously, or if you must regard what happens during the performance.

03 B. LOCAL AND GLOBAL VARIABLES

Variable Types

In Csound, there are several types of variables. It is important to understand the differences between 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 during the performance of an instrument. 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 Transform) in Csound. They are updated at k-time, but their values depend also on the FFT parameters like frame size and overlap. Examples for using f-sigs can be found in chapter [05 I](#).
- **w**-variables are used in some older spectral processing opcodes.

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

EXAMPLE 03B01_Variable_types.csd

```
<CsoundSynthesizer>
<CsOptions>
--env:SSDIR+=../SourceMaterials -o dac
</CsOptions>
<CsInstruments>
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
printfks   "kVar1 = %.3f, kVar2 = %.3f, kVar = %.3f\n", .1, kVar1, kVar2, kVar
endin

instr 3; a-variables
aVar1      poscil   .2, 400; 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
printfks   "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>
;example by joachim heintz

```

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.) is determined by 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_Local_scope.csd

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

```

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_Global_scope.csd

```

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

;global scalar variables should be initialized 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
printf     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
printf     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>
;example by joachim heintz

```

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.

Introductory Examples

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_Global_audio_intro.csd

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

Of course there is no need to use a global variable in this case. If you do it, you risk your audio will be overwritten by an instrument with a higher number using 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_Global_audio_overwritten.csd

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

Adding Global Variables

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_Global_audio_added.csd

```
<CsoundSynthesizer>
<CsInstruments>
```

```

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>
;example by joachim heintz

```

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_Global_control_added.csd

```

<CsoundSynthesizer>
<CsInstruments>
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>
;example by joachim heintz

```

Adding Audio Vectors

What happens when working 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 the vector's present and previous states. If both previous and present states (with ksmmps=9) are [0 0.1 0.2 0.1 0 -0.1 -0.2 -0.1 0] 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 opposite [0 -0.1 -0.2 -0.1 0 0.1 0.2 0.1 0], you will only get zeros when you add them. 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_Local_audio_add.csd

```
<CsoundSynthesizer>
<CsOptions>
-o dac
</CsOptions>
<CsInstruments>
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 by joachim heintz
```

prints:

i	1	time	0.10000:	0.10000
i	1	time	0.20000:	0.20000
i	1	time	0.30000:	0.30000
i	1	time	0.40000:	0.40000
i	1	time	0.50000:	0.50000
i	1	time	0.60000:	0.60000
i	1	time	0.70000:	0.70000
i	1	time	0.80000:	0.79999
i	1	time	0.90000:	0.89999
i	1	time	1.00000:	0.99999

EXAMPLE 03B09_Global_audio_add.csd

```
<CsoundSynthesizer>
<CsOptions>
```

```

-o dac
</CsOptions>
<CsInstruments>
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>
;example by joachim heintz

```

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 be evenly 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:

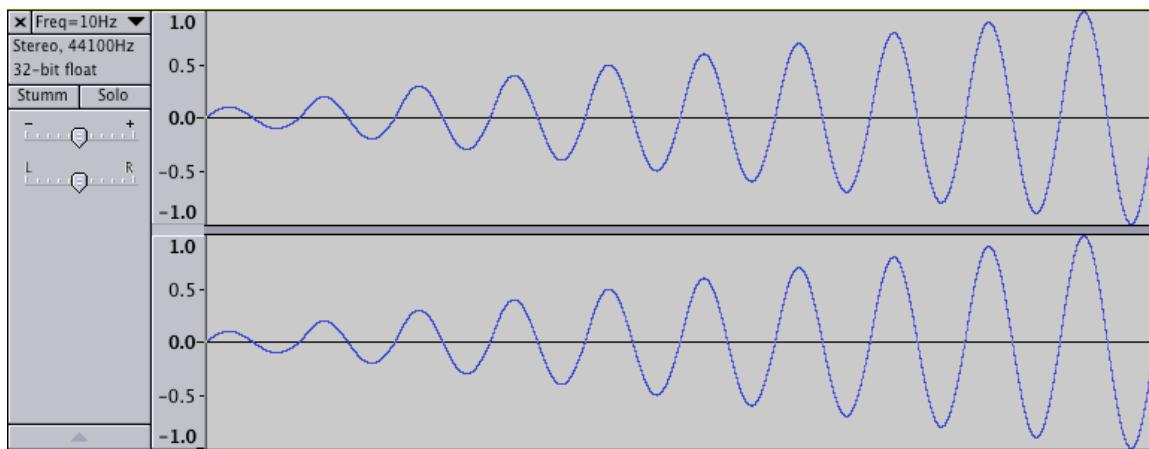


Figure 19.1: Self-reinforcing global audio signal on account of its state in one control cycle being the same as in the previous one

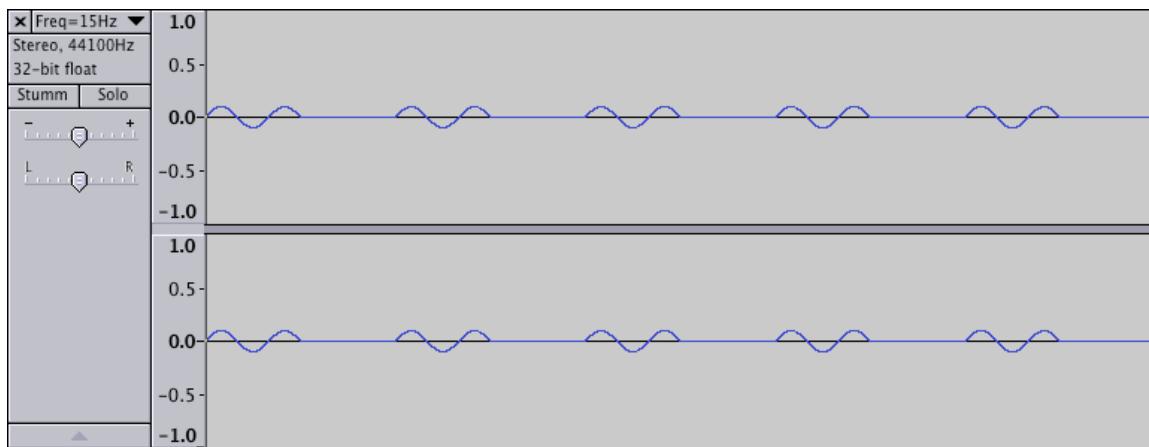


Figure 19.2: Partly self-erasing global audio signal because of phase inversions in two subsequent control cycles

Clear Global Audio After Adding

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_Global_with_clear.csd

```
<CsoundSynthesizer>
<CsOptions>
-o dac
</CsOptions>
<CsInstruments>
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
          timeout  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      oscils3    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
            timeout  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>
;example by joachim heintz

```

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, many users prefer 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_Chn_demo.csd

```

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

03 C. 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 will be covered first. Though all necessary loops can be built just by if-statements, Csound's *while*, *until* and *loop* facility offer a more comfortable way of performing loops. They will be introduced later, in the Loop and the While / Until section of this chapter. Finally, time loops are shown, which are particularly 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 chapter 03A, must be regarded very carefully when working with control structures. If a conditional branch at **i-time** is performed, the condition will be tested **just once for each note**, at the initialization pass. If a conditional branch at **k-time** is performed, the condition will be tested **again and again in each control-cycle**.

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

Thus, **if** and **while** as most used control structures 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 [diskin](#) (soundin / diskin2) with two output arguments. For the mono ones it means: use it with one output argument, and throw it to both output channels:¹

EXAMPLE 03C01_IfThen_i.csd

```
<CsoundSynthesizer>
<CsOptions>
-o dac
</CsOptions>
<CsInstruments>
```

¹The modern way to solve this is to work with an audio array as output of diskin. But nevertheless the example shows a typical usage of the i-rate if branching.

```

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

    instr 1
Sfile      =          "ClassGuit.wav"
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>
;Example by Joachim Heintz

```

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_IfThen_k.csd

```

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

; move between 0 and 1 (3 new values per second)
kGate    randomi  0, 1, 3, 3
; move between 300 and 800 hz (1 new value per sec)
kFreq    randomi  300, 800, 1, 3
; move between -12 and 0 dB (5 new values per sec)
kB       randomi  -12, 0, 5, 3
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>
;example by joachim heintz

```

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)`.² 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_IfThen_short_form.csd

```

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

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)
kB       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(kB) : 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>

```

²Since the release of the new parser (Csound 5.14), the expression can also be written without parentheses.

```
</CsoundSynthesizer>
;example by joachim heintz
```

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 **igoto** or **kgoto**. 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
...
```

In case raw *goto* is used, it is a combination of *igoto* and *kgoto*, so the condition is tested on both, initialization and performance pass.

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_IfGoto_i.csd

```
<CsoundSynthesizer>
<CsOptions>
--env:SSDIR+=../SourceMaterials -odac
```

```

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

    instr 1
Sfile      = "ClassGuit.wav"
ifilchnls filenchnl 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>
;example by joachim heintz

```

But if you want to play the file, you must also use a k-rate if-kgoto, because, not only do you have an event at i-time (initializing the soundin opcode) but also at k-time (producing an audio signal). So goto must be used here, to combine *igoto* and *kgoto*.

EXAMPLE 03C05_IfGoto_ik.csd

```

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

    instr 1
Sfile      =          "ClassGuit.wav"
ifilchnls filenchnl Sfile
    if ifilchnls == 1 goto mono
    goto stereo
mono:
aL      soundin   Sfile
aR      =          aL
    goto      continue
stereo:
aL, aR  soundin   Sfile
continue:
    outs      aL, aR
    endin

</CsInstruments>
<CsScore>
i 1 0 5

```

```
</CsScore>
</CsoundSynthesizer>
;example by joachim heintz
```

k-Rate Examples

This is the same example as above (03C02) in the if-then-else syntax for a moving gate between 0 and 1:

EXAMPLE 03C06_IfGoto_k.csd

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

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_Loops_with_if.csd

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

    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:
        printk     "k-cycle %d reached!%n", 0, ktimek
kcount    =      kcount + 1
        printk2   kcount
if kcount < 11 kgoto loop
        printk    "k-END!%n", 0
noloop:
    endin

</CsInstruments>
<CsScore>
i 1 0 0
i 2 0 1
</CsScore>
</CsoundSynthesizer>
;example by joachim heintz
```

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_Loop_opcodes_i.csd

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

    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>
;example by joachim heintz
```

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

EXAMPLE 03C09_Random_string.cs

```

<CsoundSynthesizer>
<CsOptions>
-nm0
</CsOptions>
<CsInstruments>

    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>
;example by joachim heintz

```

You can also use an i-rate loop to fill a function table (= buffer) with any kind of values. This table can then be read, or manipulated and then be read again. 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_Random_ftable_fill.cs

```

<CsoundSynthesizer>
<CsOptions>
-nm0
</CsOptions>
<CsInstruments>

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>
;example by joachim heintz
```

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 some vectorial opcodes for this task (and in Csound 6 probably array), it can also be done by a k-rate loop like the one shown here:

EXAMPLE 03C11_Table_random_dev.csd

```
<CsoundSynthesizer>
<CsOptions>
-odac
</CsOptions>
<CsInstruments>
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      oscil     .2, 400, giSine
          outs     asig, asig
        endin

</CsInstruments>
<CsScore>
i 1 0 10
</CsScore>
</CsoundSynthesizer>
;example by joachim heintz
```

While / Until

Since the release of Csound 6, it has been possible to write loops in a manner similar to that used by many other programming languages, using the keywords **while** or **until**. The general syntax is:

```
while <condition> do
    ...
od
until <condition> do
    ...
od
```

The body of the **while** loop will be performed again and again, as long as <condition> is **true**. The body of the **until** loop will be performed, as long as <condition> is **false** (not true). This is a simple example at i-rate:

EXAMPLE 03C12_while_until_i-rate.csd

```
<CsoundSynthesizer>
<CsOptions>
-nm0
</CsOptions>
<CsInstruments>
ksmps = 32

instr 1
iCounter = 0
while iCounter < 5 do
    print iCounter
iCounter += 1
od
prints "\n"
endin

instr 2
iCounter = 0
until iCounter >= 5 do
    print iCounter
iCounter += 1
od
endin

</CsInstruments>
<CsScore>
i 1 0 .1
i 2 .1 .1
</CsScore>
</CsoundSynthesizer>
;example by joachim heintz
```

Prints:

```
instr 1: iprint = 0.000
instr 1: iprint = 1.000
instr 1: iprint = 2.000
instr 1: iprint = 3.000
instr 1: iprint = 4.000
```

```
instr 2:  iprint = 0.000
instr 2:  iprint = 1.000
instr 2:  iprint = 2.000
instr 2:  iprint = 3.000
instr 2:  iprint = 4.000
```

The most important thing in using the while/until loop is to **increment** the variable you are using in the loop (here: *iCounter*). This is done by the statement

```
iCounter += 1
```

which is equivalent to the "old" way of writing as

```
iCounter = iCounter + 1
```

If you miss this increment, Csound will perform an endless loop, and you will have to terminate it by the operating system.

The next example shows a similar process at k-rate. It uses a while loop to print the values of an array, and also set new values. As this procedure is repeated in each control cycle, the instrument is being turned off after the third cycle.

EXAMPLE 03C13_while_until_k-rate.csd

```
<CsoundSynthesizer>
<CsOptions>
-nm0
</CsOptions>
<CsInstruments>
ksmps = 32

;create and fill an array
gkArray[] fillarray 1, 2, 3, 4, 5

instr 1
;count performance cycles and print it
kCycle timeinstk
printks "kCycle = %d\n", 0, kCycle
;set index to zero
kIndex = 0
;perform the loop
while kIndex < lenarray(gkArray) do
;print array value
printf " gkArray[%d] = %d\n", kIndex+1, kIndex, gkArray[kIndex]
;square array value
gkArray[kIndex] = gkArray[kIndex] * gkArray[kIndex]
;increment index
kIndex += 1
od
;stop after third control cycle
if kCycle == 3 then
turnoff
endif
endin

</CsInstruments>
<CsScore>
i 1 0 1
</CsScore>
</CsoundSynthesizer>
;example by joachim heintz
```

Prints:

```
kCycle = 1
gkArray[0] = 1
gkArray[1] = 2
gkArray[2] = 3
gkArray[3] = 4
gkArray[4] = 5
kCycle = 2
gkArray[0] = 1
gkArray[1] = 4
gkArray[2] = 9
gkArray[3] = 16
gkArray[4] = 25
kCycle = 3
gkArray[0] = 1
gkArray[1] = 16
gkArray[2] = 81
gkArray[3] = 256
gkArray[4] = 625
```

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 [timeout](#) facility. The use of timeout 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 *timeout* opcode are given so that both methods can be compared.

Timeout Basics

The [timeout](#) 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 *timeout* 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 03C14_Timout_pre.csd

```

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

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

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

</CsInstruments>
<CsScore>
i 1 0 3
i 1 4 5
</CsScore>
</CsoundSynthesizer>
;example by joachim heintz

```

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 five times for the second note:

EXAMPLE 03C15_Timout_basics.csd

```

<CsoundSynthesizer>
<CsOptions>
-odac
</CsOptions>
<CsInstruments>
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      oscil     .2, kFreq, giSine
            outs      aTone, aTone
    endin

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

```

```
;example by joachim heintz
```

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) to mark 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 for 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 [Initialization 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 03C16_Timeloop_and_not.csd

```
<CsoundSynthesizer>
<CsOptions>
-odac
</CsOptions>
<CsInstruments>
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  oscil    .2, kFreq2, giSine

        outs     aTone1+aTone2, aTone1+aTone2
    endin

</CsInstruments>
<CsScore>
i 1 0 3
i 1 4 5
</CsScore>
</CsoundSynthesizer>
;example by joachim heintz
```

Timeout 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 03C17_Timout_different_durations.csd

```
<CsoundSynthesizer>
<CsOptions>
-odac
</CsOptions>
<CsInstruments>
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      oscil     .2, kFreq, giSine
            outs      aTone, aTone
    endin

</CsInstruments>
<CsScore>
i 1 0 3
i 1 4 5
</CsScore>
</CsoundSynthesizer>
;example by joachim heintz
```

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

EXAMPLE 03C18_Timout_random_durations.csd

```
<CsoundSynthesizer>
<CsOptions>
-odac
</CsOptions>
<CsInstruments>
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
play:
kFreq   expseg  400, idur, 600
aTone   poscil  .2, kFreq, giSine
        outs     aTone, aTone
      endin

</CsInstruments>
<CsScore>
i 1 0 20
</CsScore>
</CsoundSynthesizer>
;example by joachim heintz

```

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. To achieve this, the time loop can be used to simply **trigger an event**. This can be done with `schedule`, `event_i` or `scoreline_i`. In the following example, the time loop in instrument 1 triggers a new instance of instrument 2 with a duration of 1 to 5 seconds, every 0.5 to 2 seconds. So in most cases, the previous instance of instrument 2 will still be playing when the new instance is triggered. Random calculations are executed in instrument 2 so that each note will have a different pitch, creating a glissando effect:

EXAMPLE 03C19_Timout_trigger_events.csd

```

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

The last application of a time loop with the *timeout* opcode which is shown here, is a randomly moving envelope. If we 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 *timeout* 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. Here is a possible solution:

EXAMPLE 03C20_Timout_random_envelope.csd

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

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 per second:

EXAMPLE 03C21_Timeloop_metro.csd

```
<CsoundSynthesizer>
<CsOptions>
-odac
</CsOptions>
<CsInstruments>
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 poscil .2, 400
aEnv transeg 1, p3, -10, 0
outs aSig*aEnv, aSig*aEnv
endin

</CsInstruments>
<CsScore>
i 1 0 10
</CsScore>
</CsoundSynthesizer>
;example by joachim heintz
```

The example which is given above (03C19Timout_trigger_events.csd) as a *flexible time loop by _timeout*, can be done with the *metro* opcode in this way:

EXAMPLE 03C22_Metro_trigger_events.csd

```
<CsoundSynthesizer>
<CsOptions>
-odac
</CsOptions>
<CsInstruments>
sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 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   -18, -6; peak randomly between -12 and 0 dB
kAmp    transeg  ampdb(iMaxdb), p3, -10, 0; envelope
aTone    oscil    kAmp, kFreq
          outs     aTone, aTone
endin

</CsInstruments>
<CsScore>
i 1 0 30
</CsScore>
</CsoundSynthesizer>
;example by joachim heintz

```

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

- As *metro* works at k-time, you must use the k-variants of *event* or *scoreline* to call the subinstrument. With *timeout* you must use the i-variants: of *event_i* or *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 *timeout*, you usually must separate the reinitialized from the not reinitialized section by a *rireturn* statement.

Time Loops by Using a Clock Variable

Perhaps both, the most simple and the most *Csoundish* way to perform time loops is to use Csound's internal clock. As explained in [chapter 03A](#), each control cycle in Csound is equivalent to a certain time. This time is calculated as relation between the number of samples per control cycle *ksmps* and the sample rate *sr*: $ksmps/sr$. If, for instance, we have 32 samples per control cycle at a sample rate of 44100, this would be the time for one control cycle: $32/44100 = 0.0007256235827664399$. In other words: Less than one millisecond, so definitely precise enough in the context we are discussing here.

As Csound internally calculates the relation between sample rate and number of samples per control cycle as *control rate* or *kr*, rather than *ksmps/sr* we can also write $1/kr$. This is a bit shorter and more intuitive.

The idea for using this internal time as measurement for time loops is this:

1. We set a variable, say *kTime*, to the desired duration of the time loop.

2. in each control cycle we subtract the internal time from this variable.
3. Once zero has reached, we perform the event we want to perform, and reset the *kTime* variable to the next desired time.

The next example does exactly the same as example 03C21 with the help of the *metro* opcode did, but now by using the internal clock.³

EXAMPLE 03C23_Timeloop_Internal_Clock.csd

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

instr TimeLoop
//set desired time for time loop
kLoopTime = 1/2
//set kTime to zero at start
kTime init 0
//trigger event if zero has reached ...
if kTime <= 0 then
  event "i", "Play", 0, .3
  //... and reset time
  kTime = kLoopTime
endif
//subtract time for each control cycle
kTime -= 1/kr
endin

instr Play
aEnv transeg 1, p3, -10, 0
aSig poscil .2*aEnv, 400
out aSig, aSig
endin

</CsInstruments>
<CsScore>
i "TimeLoop" 0 10
</CsScore>
</CsoundSynthesizer>
;example by joachim heintz
```

So the *trigger events* example which has been showed in using *timeout* (03C19) and *trigger* (03C22) is here again using the internal clock approach.

EXAMPLE 03C24_Internal_clock_trigger_events.csd

```
<CsoundSynthesizer>
<CsOptions>
-odac -m0
</CsOptions>
```

³To say the truth, *metro* is more precise. But this can be neglected for live situations for which this approach is mainly meant to be used.

```

<CsInstruments>
sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1
seed 0

instr TimeLoop
kTime init 0
if kTime <= 0 then
    event "i", "Play", 0, random:k(1,5)
    kTime random .3, 1.5
endif
kTime -= 1/kr
endin

instr Play
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 -18, -6; peak randomly between -12 and 0 dB
kAmp transeg ampdB(iMaxdb), p3, -10, 0; envelope
aTone oscil kAmp, kFreq
        out aTone, aTone
endin

</CsInstruments>
<CsScore>
i "TimeLoop" 0 30
</CsScore>
</CsoundSynthesizer>
;example by joachim heintz

```

Self-Triggering and Recursion

Another surprisingly simple method for a loop in time is self-triggering: When an instrument is called, it calls the next instance, so that an endless chain is created. The following example reproduces the previous one, but without the controlling *TimeLoop* instrument. Instead, at the end of instr *Play*, the next instance is called.

EXAMPLE 03C25_self_triggering.csd

```

<CsoundSynthesizer>
<CsOptions>
-odac -m128
</CsOptions>
<CsInstruments>
sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1
seed 0

instr Play
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  -18, -6; peak randomly between -12 and 0 dB
kAmp       transeg ampdb(iMaxdb), p3, -10, 0; envelope
aTone      oscil   kAmp, kFreq
            out     aTone, aTone
schedule("Play",random:i(.3,1.5),random:i(1,5))
endin

instr Exit
exitnow()
endin

</CsInstruments>
<CsScore>
i "Play" 0 3
i "Exit" 20 1
</CsScore>
</CsoundSynthesizer>
;example by joachim heintz

```

The problem here is: how to stop? The `turnoff2` opcode does not help, because in the moment we turn off the running instance, it has already triggered the next instance.

In our example, this problem has been solved the brutal way: to exit Csound. Much better is to introduce a break condition. This is what is called *base case* in recursion. We can, for instance, give a counter as `p4`, say 20. For each instance, the new call is done with `p4-1` (19, 18, 17, ...). When zero is reached, no self-triggering is done any more.

EXAMPLE 03C26_recursion.csd

```

<CsoundSynthesizer>
<CsOptions>
-odac -m128
</CsOptions>
<CsInstruments>
sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1
seed 0

instr Play
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  -18, -6; peak randomly between -12 and 0 dB
kAmp       transeg ampdb(iMaxdb), p3, -10, 0; envelope
aTone      oscil   kAmp, kFreq
            out     aTone, aTone
if p4 > 0 then
  schedule("Play",random:i(.3,1.5),random:i(1,5), p4-1)
endif
endin

</CsInstruments>
<CsScore>
i "Play" 0 3 20
</CsScore>

```

```
</CsoundSynthesizer>
;example by joachim heintz
```

Recursion is in particular important for User Defined Opcodes. Recursive UDOs will be explained in chapter [03 G](#). They follow the same principles as shown here.

03 D. FUNCTION TABLES

Note: This chapter was written before arrays had been introduced into Csound. Now the usage of arrays is in some situations preferable to using function tables. Have a look in chapter 03 E to see how you can use arrays.

A function table is essentially the same as what other audio programming languages might 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) it can store. 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 7, and the numbers [1.1, 2.2, 3.3, 5.5, 8.8, 13.13, 21.21] in it, this is the relation of value and index:

VALUE	INDEX
1.1	0
2.2	1
3.3	2
5.5	3
8.8	4
13.13	5
21.21	6

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 function tables are stored in RAM) and flexible use of function tables.

How to Generate a Function Table

Each function table must be created **before** it can be used.¹ Even if you want to write values later, you must first create an empty table, because you must initially reserve some space in memory

¹Nevertheless function tables can be created at any time during the Csound performance, for instance via event "f" ...

for it.

Each creation of a function table in Csound is performed by one of the **GEN Routines**. Each GEN Routine generates a function table in a particular way: **GEN01** transfers audio samples from a soundfile into a table, **GEN02** stores values we define explicitly one by one, **GEN10** calculates a waveform using user-defined weightings of harmonically related 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 their general use and provide some simple examples using commonly used GEN routines.

GEN02 and General Parameters for GEN Routines

Let's start with our example described above and write the 7 numbers into a function table with 7 storage locations. For this task 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 use of an "**f statement**" or an "**f score event**" (in a manner similar to the use of "i score events" to call instrument instances). The input parameters after the **f** are as follows:

- **#**: a number (as positive integer) for this function table;
- **time**: at what time, in relation to the passage of the score, the function table is created (usually 0: from the beginning);
- **size**: the size of the function table. A little care is required: in the early days of Csound only power-of-two sizes were possible for function tables (2, 4, 8, 16, ...); nowadays almost all GEN Routines accept other sizes, but these non-power-of-two sizes must be declared as negative numbers!²
- **2**: the number of the GEN Routine which is used to generate the table, and here is another important point which must be borne in mind: **by default, Csound normalizes the table values**. This means that the maximum is scaled to +1 if positive, and to -1 if negative. All other values in the table are then scaled by the same factor that was required to scale the maximum to +1 or -1. To **prevent** Csound from normalizing, a **negative** number can be given as GEN number (in this example, the GEN routine number will be given as -2 instead of 2).
- **v1 v2 v3 ...**: the values which are written into the function table.

The example below demonstrates how the values [1.1 2.2 3.3 5.5 8.8 13.13 21.21] can be stored in a function table using an f-statement in the score. Two versions are created: an unnormalised version (table number 1) and a normalised version (table number 2). The difference in their contents will be demonstrated.

EXAMPLE 03D01_Table_norm_notNorm.csd

```
<CsoundSynthesizer>
<CsOptions>
```

²At least this is still the safest method to declare a non-power-of-two size for the table, although for many GEN routines also positive numbers work.

```

-nm0
</CsOptions>
<CsInstruments>
instr 1 ;prints the values of table 1 or 2
  prints "%nFunction Table %d:%n", p4
  indx init 0
  while indx < 7 do
    ival table indx, p4
    prints "Index %d = %f%n", indx, ival
    indx += 1
  od
endin
</CsInstruments>
<CsScore>
f 1 0 -7 -2 1.1 2.2 3.3 5.5 8.8 13.13 21.21; not normalized
f 2 0 -7 2 1.1 2.2 3.3 5.5 8.8 13.13 21.21; normalized
i 1 0 0 1; prints function table 1
i 1 0 0 2; prints function table 2
</CsScore>
</CsoundSynthesizer>
;example by joachim heintz

```

Prints:

```

Function Table 1:
Index 0 = 1.100000
Index 1 = 2.200000
Index 2 = 3.300000
Index 3 = 5.500000
Index 4 = 8.800000
Index 5 = 13.130000
Index 6 = 21.210000

Function Table 2:
Index 0 = 0.051862
Index 1 = 0.103725
Index 2 = 0.155587
Index 3 = 0.259312
Index 4 = 0.414899
Index 5 = 0.619048
Index 6 = 1.000000

```

Instrument 1 simply reads and prints (to the terminal) the values of the table. Notice the difference in values read, 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 generally preferable to the old way of writing an f-statement in the score.³ The syntax is explained below:

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

- **gir**: 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 0 is given here, Csound will generate the number, which is mostly preferable.

³*ftgen* is preferred mainly because you can refer to the function table by a variable name and must not deal with constant tables numbers. This will enhance the portability of orchestras and better facilitate the combining of multiple orchestras. It can also enhance the readability of an orchestra if a function table is located in the code nearer the instrument that uses it. And, last but not least, variables can be put as arguments into *ftgen* – imagine for instance a size for recording tables which you generate or pass as user input.

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_Table_ftgen.csd

```
<CsoundSynthesizer>
<CsOptions>
-nm0
</CsOptions>
<CsInstruments>

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

instr 1; prints the values of table 1 or 2
    prints "%nFunction Table %d:%n", p4
indx    init     0
while indx < 7 do
    prints "Index %d = %f%n", indx, table:i(indx,p4)
    indx += 1
od
    endin

</CsInstruments>
<CsScore>
i 1 0 0 1; prints function table 1
i 1 0 0 2; prints function table 2
</CsScore>
</CsoundSynthesizer>
;example by joachim heintz
```

Most of the GEN routines offer the possibility to insert the arguments as array rather than as single values. The manual page for ftgen shows this line:

```
gir      ftgen      ifn, itime, isize, igen, iarray
```

The iarray will contain the arguments of the GEN routine as array. Note that you cannot mix here numbers and strings as currently Csound can only have an array of one single type. Usually the array will contain numbers. This is a simple example in which we generate an array containing the numbers from 1 to 7, and then put the array in a GEN02 function table. As in the previous example, we simply print the content of the table.

EXAMPLE 03D03_ftgen_array_arg.csd

```
<CsoundSynthesizer>
<CsOptions>
-nm0
</CsOptions>
<CsInstruments>

// create i-array
iArray[] = genarray(1,7)

// put this array into a function table via GEN02
```

```

giFt ftgen 0, 0, -7, -2, iArray

instr 1; prints the values of giFt
prints("%nFunction Table giFt:%n")
indx = 0
while (indx < 7) do
  prints(" Index %d = %f%n", indx, table:i(indx,giFt))
  indx += 1
od
endin

</CsInstruments>
<CsScore>
i 1 0 0
</CsScore>
</CsoundSynthesizer>
;example by joachim heintz

```

Prints:

```

Function Table giFt:
Index 0 = 1.000000
Index 1 = 2.000000
Index 2 = 3.000000
Index 3 = 4.000000
Index 4 = 5.000000
Index 5 = 6.000000
Index 6 = 7.000000

```

This feature is very powerful, and can be much more than an abbreviation. Have a look at example 03D06 below, at the end of the GEN10 explanation.

GEN01: Importing a Soundfile

GEN01 is used for importing soundfiles stored on disk into the computer's RAM, ready 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. Note that on this occasion the function table number (*ifn*) has been defined using a zero. This means that Csound will automatically assign a unique function table number. This number will also be held by the variable *giFile* which we will normally use to reference the function table anyway so its actual value will not be important to us. If you are interested you can print the value of *giFile* out. If no other tables are defined, it will be 101 and subsequent tables, also using automatically assigned table numbers, will follow accordingly: 102, 103 etc.
- **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** to **0**.
- **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 often 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:SSDIR+=/path/to/sounds` 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:** The format of the amplitude samples in the soundfile, e.g. 16 bit, 24 bit etc. Usually providing 0 here is sufficient, in which case Csound will 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. Note that only certain opcodes are able to properly make use of multichannel audio stored in function tables.

The following example loads a short sample into RAM via a function table and then plays it. Reading the function table here is done using the `poscil3` opcode, as one of many choices in Csound.

EXAMPLE 03D04_Sample_to_table.csd

```
<CsoundSynthesizer>
<CsOptions>
-odac --env:SSDIR=../SourceMaterials
</CsOptions>
<CsInstruments>
sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

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

instr PlayOnce
p3 filelen gS_file ;play whole length of the sound file
aSamp poscil3 .5, 1/p3, giSample
out aSamp, aSamp
endin

</CsInstruments>
<CsScore>
i "PlayOnce" 0 1
</CsScore>
</CsoundSynthesizer>
;example by joachim heintz
```

GEN10: Creating a Waveform

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

There are many GEN Routines which can be used to achieve this. The simplest one is `GEN10`. It produces a waveform by adding sine waves which have the "harmonic" frequency relationship 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, with GEN10 you must specify 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 approximations of the standard harmonic waveforms by the addition of sinusoids. 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 03D05_Standard_waveforms_with_GEN10.csd

```
<CsoundSynthesizer>
<CsOptions>
-odac
</CsOptions>
<CsInstruments>
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 Sine
aSine      oscil    .2, 400, giSine
aEnv       linen    aSine, .01, p3, .05
            outs    aEnv, aEnv
    endin

    instr Saw
aSaw       oscil    .2, 400, giSaw
aEnv       linen    aSaw, .01, p3, .05
            outs    aEnv, aEnv
    endin

    instr Square
aSqu      oscil    .2, 400, giSquare
aEnv       linen    aSqu, .01, p3, .05
            outs    aEnv, aEnv
    endin

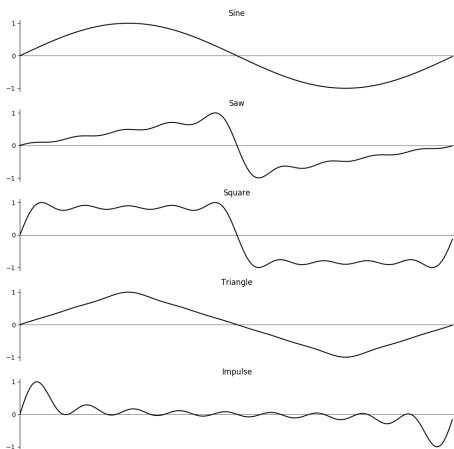
    instr Triangle
aTri      oscil    .2, 400, giTri
aEnv       linen    aTri, .01, p3, .05
            outs    aEnv, aEnv
    endin

    instr Impulse
aImp      oscil    .2, 400, giImp
aEnv       linen    aImp, .01, p3, .05
            outs    aEnv, aEnv
    endin

    instr Sine_with_env
aEnv      oscil    .2, (1/p3)/2, giSine
aSine      oscil    aEnv, 400, giSine
            outs    aSine, aSine
    endin

```

```
</CsInstruments>
<CsScore>
i "Sine" 0 3
i "Saw" 4 3
i "Square" 8 3
i "Triangle" 12 3
i "Impulse" 16 3
i "Sine_with_env" 20 3
</CsScore>
</CsoundSynthesizer>
;Example by Joachim Heintz
```



The last example for GEN10 continues what was explained above (example 03D03) about the usage of arrays. It is rather complex; if you are a beginner in Csound, just skip it.

We use four arrays here to create four random GEN10 tables, representing four different timbres. Each of these arrays are filled with 40 amplitude values for the first 40 harmonics. In a loop a random number between -1 and +1 is generated. If its absolute value is below 0.8, the value is set to zero. If its absolute value is above 0.8, it is kept with a standard attenuation for the higher partials. For the wavetable transformation we imagine a square with the four tables as corners. We imagine a two-dimensional area with a range of 0-1 for both, the horizontal and vertical axis. In the example we move from table 1 (bottom left of the square) to table 2 (bottom right), to table 3 (top right), to table 4 (top left), and back to table 1. Then we move in the diagonal to the middle in which all four tables mix to one sound.

EXAMPLE 03D06_random_GEN10_wavetable.csd

```
<CsoundSynthesizer>
<CsOptions>
-odac
</CsOptions>
<CsInstruments>
sr = 44100
ksmps = 64
0dbfs = 1
nchnls = 2
seed 13 ;try other values here

// create an array containing the number of four tables
gfn[ ] init 4

// create an array for 40 harmonics
```

```

ihar[ ] init 40

// for each table ...
ii = 0
while ii < 4 do

    // ... put 40 random amplitudes for the harmonics in it
    ik = 1
    while ik < 40 do
        // generate random values between 0 and 1
        irnd = abs(random:i(-1,1))
        // use only the ones above 0.8 and scale
        ihar[ik-1] = irnd < 0.8 ? 0 : irnd/ik
        ik += 1
    od
    // here the array is inserted as argument to GEN10
    gifn[ii] = ftgen(0,0,8192,10,ihar)
    ii += 1
od

instr Wavetable

// set volume and base frequency
iAmp = 0.2
iFreq = 133

// go table 1 -> 2 -> 3 -> 4 -> 1, then to a mix of all
kh = linseg:k(0,10,1,10,1,10,0,10,0,10,0.5)
kv = linseg:k(0,10,0,10,1,10,1,10,0,10,0.5)

// read the four tables ...
a1 = poscil:a(iAmp,iFreq,gifn[0])
a2 = poscil:a(iAmp,iFreq,gifn[1])
a3 = poscil:a(iAmp,iFreq,gifn[2])
a4 = poscil:a(iAmp,iFreq,gifn[3])

// ... and mix according to kh and kv
aMix = (a1*(1-kh)+a2*kh)*(1-kv) + (a3*(1-kh)+a4*kh)*kv

// fades and output
aOut = linen:a(aMix,1,p3,10)
outall(aOut)

endin

</CsInstruments>
<CsScore>
i "Wavetable" 0 60
</CsScore>
</CsoundSynthesizer>
;Example by Victor Lazzarini, adapted by joachim heintz

```

How to Write Values to a Function Table

As we have seen, GEN Routines generate function tables, and by doing this, they write values into them according to various methods, but in certain cases you might first want to create an empty table, and then write the values into it later or you might want to alter the default values held in a

function table. The following section demonstrates how to do this.

To be precise, it is not actually correct to talk about an “empty table”. If Csound creates an “empty” table, in fact it writes zeros to the indices which are not specified. Perhaps the easiest method of creating an “empty” table for 100 values is shown below:

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

The simplest to use opcode that writes values to existing function tables during a note’s performance is [tablew](#) and its i-time equivalent is [tableiw](#). 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] [, iwemode]
tablew  ksig, kndx, ifn [, ixmode] [, ixoff] [, iwemode]
tablew  asig, andx, ifn [, ixmode] [, ixoff] [, iwemode]
```

- ***isig, ksig, asig*** is the value (variable) you want to write into a specified location of the table;
- ***indx, kndx, andx*** is the location (index) where you will write the value;
- ***ifn*** is the function table you want to write to;
- ***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 iwemode=0 (default), any index larger than possible is written at the last possible index. If iwemode=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 them to a table. An empty table has first been created in the header (filled with zeros), then instrument 1 calculates the values in an i-time loop and writes them to the table using [tableiw](#). Instrument 2 simply prints all the values in a list to the terminal.

EXAMPLE 03D07_Write_Fibo_to_table.csd

```
<CsoundSynthesizer>
<CsOptions>
-nm0
</CsOptions>
<CsInstruments>

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
while indx < 12 do
  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
  indx += 1
od
endin

instr 2; prints the values of the table
prints "%nContent of Function Table:%n"
indx init 0
while indx < 12 do
  ival table indx, giFt
  prints "Index %d = %f%n", indx, ival
  indx += 1
od
endin

</CsInstruments>
<CsScore>
i 1 0 0
i 2 0 0
</CsScore>
</CsoundSynthesizer>
;example by joachim heintz

```

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 03D08_Record_ksig_to_table.csd

```

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

```

As you see, this typical case of writing k-values to a table requires a changing value for the index, otherwise tablew will continually overwrite at the same table location. This changing value can be created using the `line` or `linseg` opcodes - as was done here - or by using a `phasor`. A phasor moves continuously from 0 to 1 at a user-defined frequency. For example, if you want a phasor to move from 0 to 1 in 5 seconds, you must set the frequency to 1/5. Upon reaching 1, the phasor will wrap-around to zero and begin again. Note that phasor can also be given a negative frequency in which case it moves in reverse from 1 to zero then wrapping around to

1. By setting the `ixmode` argument of `tablew` to 1, you can use the phasor output directly as writing pointer. Below is an alternative version of instrument 1 from the previous example, this time using phasor to generate the index values:

```

instr 1; rec/play of a random frequency movement for 5 seconds
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 to provide input values and also an index that changes at a-rate. The next example first records a

randomly generated audio signal and then plays it back. It then records the live audio input for 5 seconds and subsequently plays it back.

EXAMPLE 03D09_Record_audio_to_table.csd

```

<CsoundSynthesizer>
<CsOptions>
-iadc -odac
</CsOptions>
<CsInstruments>
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 ; record 5 seconds of generated audio to a table
i 2 6 5 ; play back the recording of generated audio
i 3 12 7 ; record 5 seconds of live audio to a table
i 4 20 5 ; play back the recording of live audio
</CsScore>
</CsoundSynthesizer>
;example by joachim heintz

```

How to Retrieve 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` opcodes (which are explained above). It is simply its counterpart for reading values from a function table instead of writing them. Its output can be either an i-, k- or a-rate signal and the value type of the output automatically selects either the a- k- or a-rate version of the opcode. The first input is an index at 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) and when CPU cycles are no object, `tablexkt` can be used for ultimate interpolating quality.⁴

Examples of the use of the `table` opcodes can be found in the earlier examples in the *How to Write Values to a Function Table* section.

⁴For a general introduction about interpolation, see for instance <http://en.wikipedia.org/wiki/Interpolation>

Oscillators

It is normal to read tables that contain a single cycle of an audio waveform using an oscillator but you can actually 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 (the opcode creates the required index internally based on the asked for frequency).

You should bear in mind that some of the oscillators in Csound might 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 03D10_RecPlay_ak_signals.csd

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

```

Saving the Contents of a Function Table to a File

A function table exists only 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 firstly on 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 be easily written to a file using the *ftsave* (i-time) or *ftsavek* (k-time) opcode. Their use is very simple. The first argument specifies the filename (in double quotes), the second argument selects between a text format (non zero) or a binary format (zero) output. Finally you just provide the number of the function table(s) to save.

With 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 03D11_ftsave.csd

```

<CsoundSynthesizer>
<CsInstruments>
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>
;example by joachim heintz

```

The counterpart to `ftsave/ftsavek` are the `ftload/ ftloadk` opcodes. You can use them to 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. In Csound 6.12, `ftaudio` has been introduced. The following example has in a way substituted by this new opcode, but it may show how many low-level routines can be written in Csound code. First instrument 1 records the live input. Then instrument 2 creates a soundfile “`testwrite.wav`” containing this audio in the same folder as the `.csd`. This is done at the first k-cycle of instrument 2, by repeatedly reading 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 03D12_Table_to_soundfile.csd

```

<CsoundSynthesizer>
<CsOptions>
-i adc
</CsOptions>
<CsInstruments>

```

```

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
iDone ftaudio giAudio, "testwrite.wav", -1
if iDone==1 then
prints "FUNCTION TABLE WRITTEN TO FILE 'testwrite.wav'!%n"
endif
endin

</CsInstruments>
<CsScore>
i 1 0 7
i 2 7 0
</CsScore>
</CsoundSynthesizer>
;example by joachim heintz

```

Reading a Text File in a Function Table

Sometimes we have data in a text file which we want to use in a function table. As a practical example, let us assume we have a drawing from PD like this:

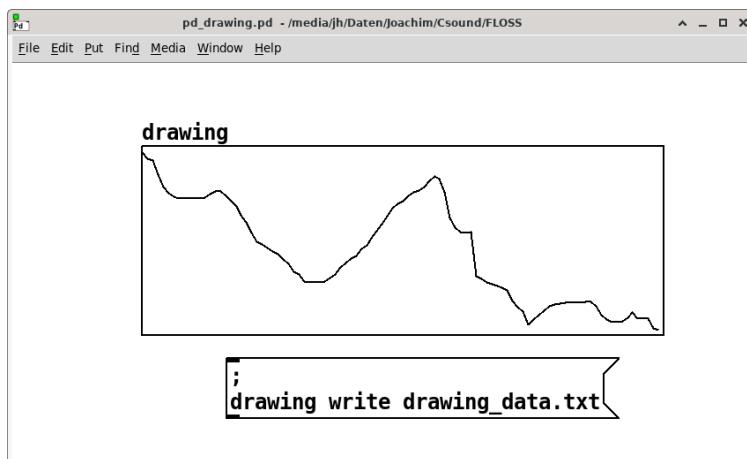


Figure 21.1: PD array as drawing and saved to a text file

The array is scaled from 0 to 10, and the 100 data points are saved in the file *drawing_data.txt*. When we open the file, we see each data point as number on a new line: 9.67289 9.34579 9.25233 8.50468 7.85049 7.52339 ... We can import now this text file, and write the data points in it to a function table, via [GEN23](#), The syntax, as written in the manual:

```
f # time size -23 "filename.txt"
```

The *size* parameter we will set usually to 0 which means: The table will have the same size as the numbers in the file (so 100 here).

-23 tells Csound to use GEN Routine 23 without normalizing the data. (23 instead of -23 would scale our data between 0 and 1 rather than 0 and 10.)

"filename.txt" is the text file with the numerical data to read. Not only newlines can be used, but also spaces, tabs or commas as separators between the numbers. (So we could use a numerical spread sheet when we export it as .csv text file.)

In the following example we apply a simple granular synthesis and we use the function table with the drawing data in two ways in it:

- We interpret the drawing as number of grains per second. So at the beginning we will have a high grain density, and at the end a low grain density.
- We set the grain duration as reciprocal of the grain density. In the simple form, it would mean that for a density of 10 grains per second we have a grain duration of 1/10 seconds. To avoid very long grains at the end, we modify it to half of the reciprocal (so that a grain density of 10 Hz results in grains of 1/20 seconds).

EXAMPLE 03D13_textfile_to_table.csd

```
<CsoundSynthesizer>
<CsOptions>
-odac
</CsOptions>
<CsInstruments>
sr = 44100
ksmps = 64
nchnls = 2
0dbfs = 1

```

```

//create a function table with the drawing data from a file
giDrawing = ftgen(0,0,0,-23,"drawing_data.txt")

//sound file to be played
gS_file = "fox.wav"

instr ReadTable

    //index as pointer from start to end of the table over duration (p3)
    kIndx = linseg:k(0,p3,100)
    //values are read at k-rate with interpolation as a global variable
    gkDrawVals = tablei:k(kIndx,giDrawing)

    endin

instr PlayWithData

    //calculate the skiptime for the sound file compared to the duration
    kFoxSkip = (timeinsts:k() / p3) * filelen(gS_file)

    //trigger the grains in the frequency of the drawing (density)
    kTrig = metro(gkDrawVals)

    //if one grain is triggered
    if kTrig==1 then

        //calculate the grain duration as half the reciprocal of the density
        kGrainDuration = 0.5/gkDrawVals

        //call the grain and send the skiptime
        schedulek("PlayGrain",0,kGrainDuration,kFoxSkip)

    endif

    endin

instr PlayGrain

    //get the skiptime from the calling instrument
    iSkip = p4
    //read the sound from disk at this point
    aSnd diskin gS_file,1,iSkip
    //apply triangular envelope
    aOut = linen:a(aSnd,p3/2,p3,p3/2)
    //output
    outall(aOut)

    endin

</CsInstruments>
<CsScore>
i "ReadTable" 0 10
i "PlayWithData" 0 10
</CsScore>
</CsoundSynthesizer>
;example by joachim heintz

```

There is no need to read the table values as a global variable in a separate instrument. It would be perfectly fine to have it in the PlayWithData instrument, too.

Other GEN Routine Highlights

GEN05, **GEN07**, **GEN25**, **GEN27** and **GEN16** are useful for creating envelopes. **GEN07** and **GEN27** create functions table in the manner of the **linseg** opcode - with **GEN07** the user defines segment duration whereas in **GEN27** the user defines the absolute time for each breakpoint from the beginning of the envelope. **GEN05** and **GEN25** operate similarly to **GEN07** and **GEN27** except that envelope segments are exponential in shape. **GEN16** also create an envelope in breakpoint fashion but it allows the user to specify the curvature of each segment individually (concave - straight - convex).

GEN17, **GEN41** and **GEN42** are used the generate histogram-type functions which may prove useful in algorithmic composition and work with probabilities.

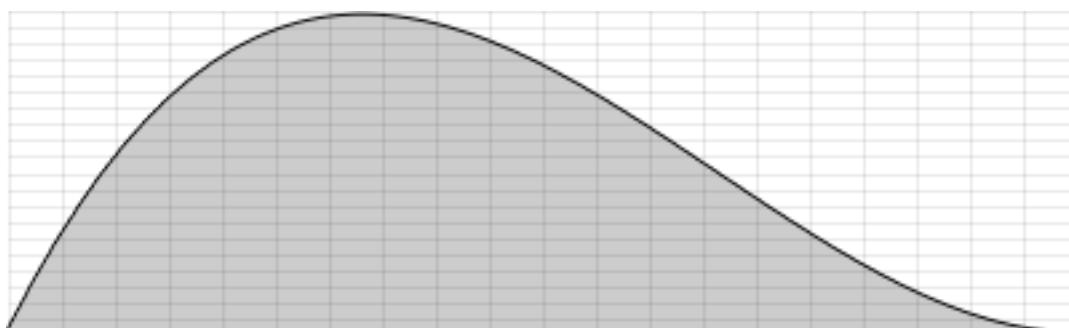
GEN09 and **GEN19** are developments of **GEN10** and are useful in additive synthesis.

GEN11 is a GEN routine version of the **gbuzz** opcode and as it is a fixed waveform (unlike **gbuzz**) it can be a useful and efficient sound source in subtractive synthesis.

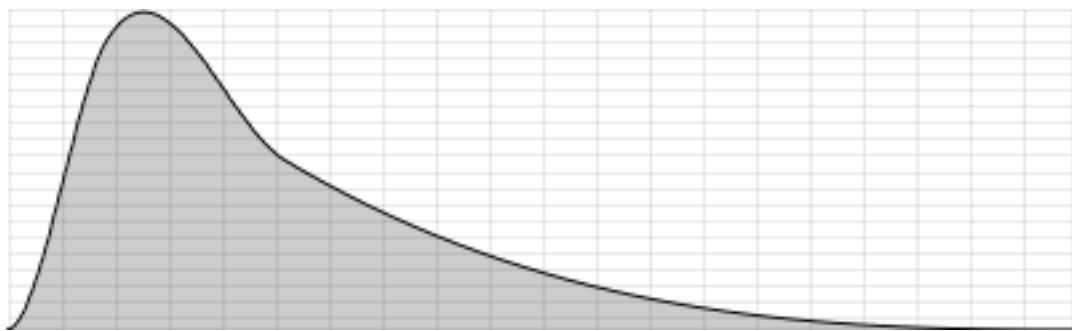
GEN08

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

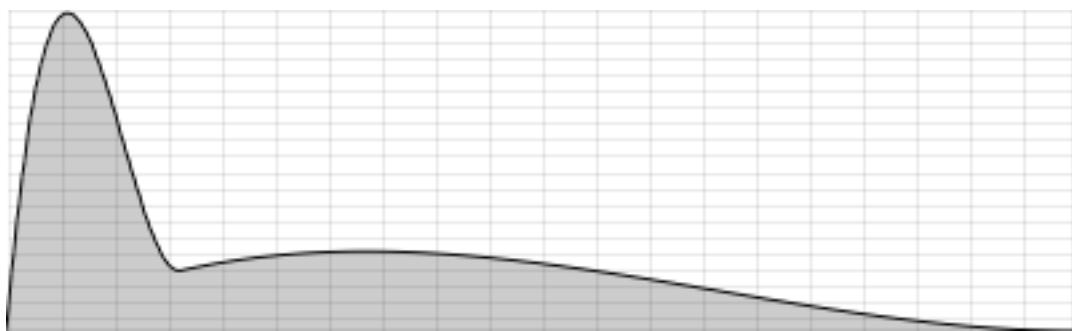
GEN08 creates a curved function that forms the smoothest possible line between a sequence of user defined break-points. This GEN routine can be useful for the creation of window functions for use as envelope shapes or in granular synthesis. In forming a smooth curve, **GEN08** may create apexes that extend well above or below any of the defined values. For this reason **GEN08** is mostly used with post-normalisation turned on, i.e. a minus sign is not added to the GEN number when the function table is defined. Here are some examples of **GEN08** tables:



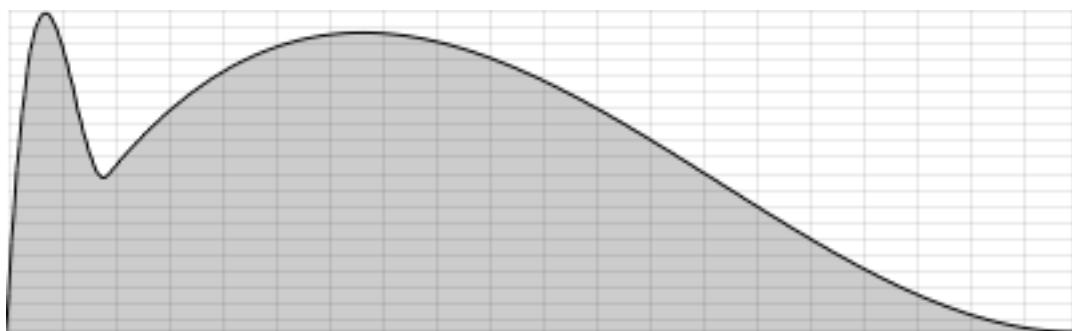
```
f 1 0 1024 8 0 1 1 1023 0
```



```
f 2 0 1024 8 0 97 1 170 0.583 757 0
```



```
f 3 0 1024 8 0 1 0.145 166 0.724 857 0
```

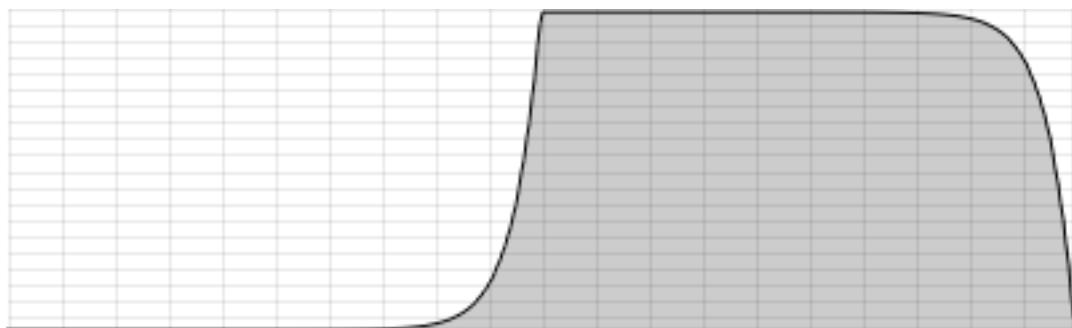


```
f 4 0 1024 8 0 1 0.079 96 0.645 927 0
```

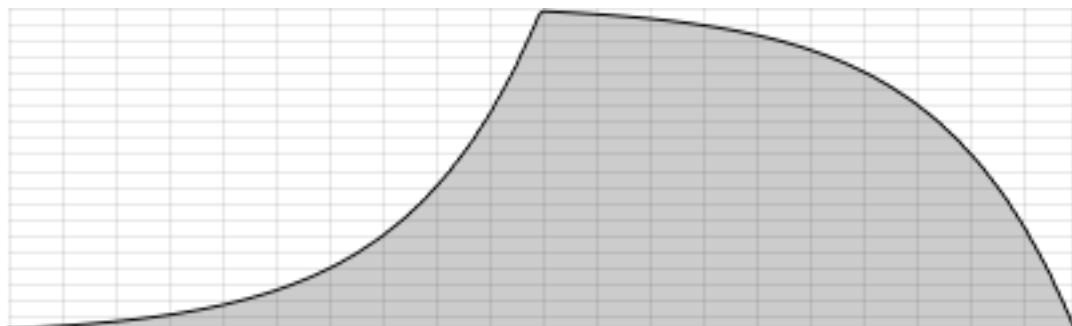
GEN16

```
f # time size 16 val1 dur1 type1 val2 [dur2 type2 val3 ... typeX valN]
```

GEN16 allows the creation of envelope functions using a sequence of user defined breakpoints. Additionally for each segment of the envelope we can define a curvature. The nature of the curvature – concave or convex – will also depend upon the direction of the segment: rising or falling. For example, positive curvature values will result in concave curves in rising segments and convex curves in falling segments. The opposite applies if the curvature value is negative. Below are some examples of GEN16 function tables:



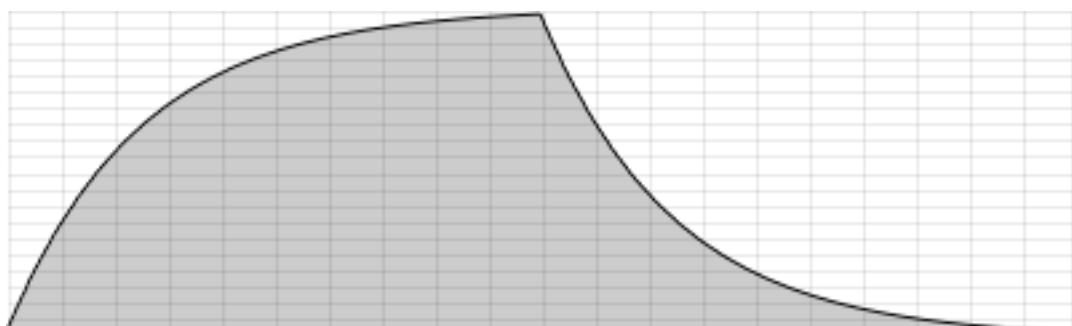
Screenshot_2023-05-31_16-20-42 f 1 0 1024 16 0 512 20 1 512 20 0



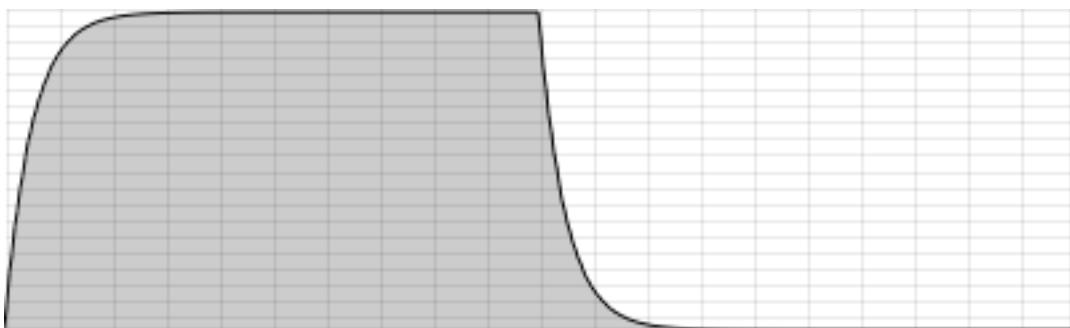
f 2 0 1024 16 0 512 4 1 512 4 0



f 3 0 1024 16 0 512 0 1 512 0 0



f 4 0 1024 16 0 512 -4 1 512 -4 0

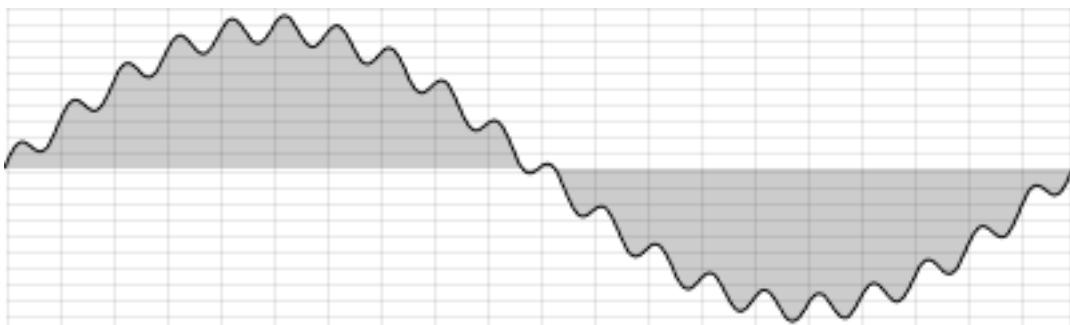


```
f 5 0 1024 16 0 512 -20 1 512 -20 0
```

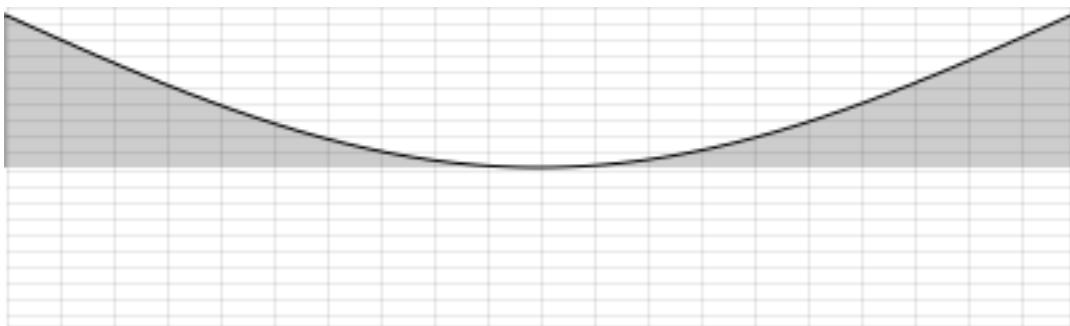
GEN19

```
f # time size 19 pna stra phsa dcoa pnb strb phsb dcob ...
```

GEN19 follows on from GEN10 and GEN09 in terms of complexity and control. It shares the basic concept of generating a harmonic waveform from stacked sinusoids but in addition to control over the strength of each partial (GEN10) and the partial number and phase (GEN09) it offers control over the DC offset of each partial. In addition to the creation of waveforms for use by audio oscillators other applications might be the creation of functions for LFOs and window functions for envelopes in granular synthesis. Below are some examples of GEN19:



```
f 1 0 1024 19 1 1 0 0 20 0.1 0 0
```



```
f 2 0 1024 -19 0.5 1 180 1
```

GEN30

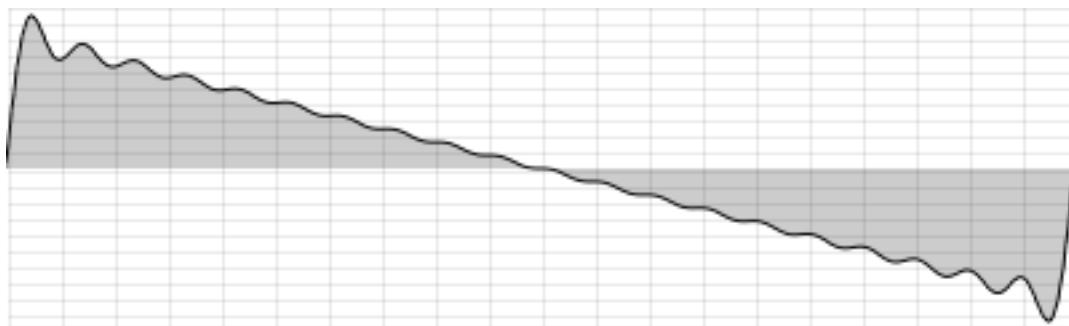
```
f # time size 30 src minh maxh [ref_sr] [interp]
```

GEN30 uses FFT to create a band-limited version of a source waveform without band-limiting. We can create a sawtooth waveform by drawing one explicitly using GEN07 by used as an audio waveform this will create problems as it contains frequencies beyond the Nyquist frequency therefore will cause aliasing, particularly when higher notes are played. GEN30 can analyse this waveform and create a new one with a user defined lowest and highest partial. If we know what note we are going to play we can predict what the highest partial below the Nyquist frequency will be. For a given frequency, freq, the maximum number of harmonics that can be represented without aliasing can be derived using $sr / (2 * freq)$.

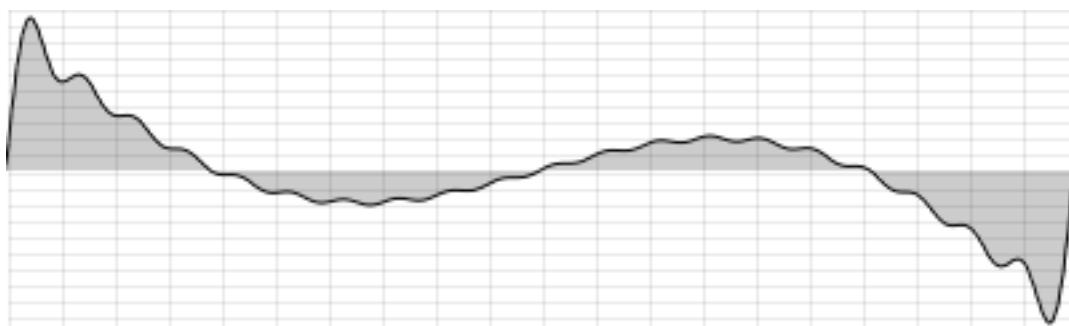
Here are some examples of GEN30 function tables (the first table is actually a GEN07 generated sawtooth, the second two are GEN30 band-limited versions of the first):



```
f 1 0 1024 7 1 1024 -1
```



```
f 2 0 1024 30 1 1 20
```



```
f 3 0 1024 30 1 2 20
```


03 E. ARRAYS

Arrays can be used in Csound since version 6. This chapter first describes the naming conventions and the different possibilities to create an array. After looking more closely to the different types of arrays, the operations on arrays will be explained. Finally examples for the usage of arrays in user-defined opcodes (UDOs) are given.

Naming Conventions

An array is stored in a variable. As usual in Csound, the first character of the variable name declares the array as **i** (numbers, init-time), **k** (numbers, perf-time), **a** (audio vectors, perf-time) or **S** (strings, init- or perf-time). (More on this below, and in chapter [03 A.](#))

At *first* occurrence, the array variable must be followed by *brackets*. The brackets determine the dimensions of the array. So

```
kArr[] init 10
```

creates a one-dimensional k-array of length 10, whereas

```
kArr[][] init 8, 10
```

creates a two-dimensional k-array with 8 rows and 10 columns.

After the first occurrence of the array, referring to it as a whole is done *without* any brackets. Brackets are only used if an element is indexed:

```
kArr[] init 10 ;with brackets: first occurrence  
kLen = lenarray(kArr) ;without brackets: *kArr* not *kArr[]*  
kFirstEl = kArr[0] ;with brackets because of indexing
```

The same syntax is used for a simple copy via the **=** operator:

```
kArr1[] init 10 ;creates kArr1  
kArr2[] = kArr1 ;creates kArr2[] as copy of kArr1
```

Creating an Array

An array can be created by different methods:

- with the `init` opcode,
- with `fillarray`,
- with `genarray`,
- as a copy of an already existing array with the `=` operator,
- implicit as result of some opcodes, e.g. `diskin`.

init

The most general method, which works for arrays of any number of dimensions, is to use the `init` opcode. Each argument for `init` denotes the size of one dimension.

```
kArr[] init 10 ;creates a one-dimensional array with length 10
kArr[][] init 8, 10 ;creates a two-dimensional array (8 lines, 10 columns)
```

fillarray

With the `fillarray` opcode distinct values are assigned to an array. If the array has not been created before, it will be created as result, in the size of elements which are given to `fillarray`. This ...

```
iArr[] fillarray 1, 2, 3, 4
```

... creates an *i*-array of size=4. Note the difference in using the brackets in case the array has been created before, and is filled afterwards:

```
iArr[] init 4
iArr fillarray 1, 2, 3, 4
```

It is also possible to use `functional syntax` for `fillarray`:

```
iArr[] = fillarray(1, 2, 3, 4)
```

In conjunction with a previously defined two-dimensional array, `fillarray` can set the elements, for instance:

```
iArr[][] init 2, 3
iArr fillarray 1, 2, 3, -1, -2, -3
```

This results in a 2D array (matrix) with the elements 1 2 3 as first row, and -1 -2 -3 as second row.¹

¹Another method to fill a matrix is to use the `setrow` opcode. This will be covered later in this chapter.

genarray

This opcode creates an array which is filled by a series of numbers from a start value to an (included) end value. Here are some examples:

```
iArr[] genarray 1, 5 ; creates i-array with [1, 2, 3, 4, 5]
kArr[] genarray_i 1, 5 ; creates k-array at init-time with [1, 2, 3, 4, 5]
iArr[] genarray -1, 1, 0.5 ; i-array with [-1, -0.5, 0, 0.5, 1]
iArr[] genarray 1, -1, -0.5 ; [1, 0.5, 0, -0.5, -1]
iArr[] genarray -1, 1, 0.6 ; [-1, -0.4, 0.2, 0.8]
```

Copy with =

The `=` operator copies any existing array to a new variable. The example shows how a global array is copied into a local one depending on a score p-field: If `p4` is set to 1, `iArr[]` is set to the content of `gi_Arr_1`; if `p4` is 2, it gets the content of `gi_Arr_2`. The content of `iArr[]` is then sent to instr `Play` in a `while` loop.

EXAMPLE 03E01_CopyArray.csd

```
<CsoundSynthesizer>
<CsOptions>
-odac -m128
</CsOptions>
<CsInstruments>
sr = 44100
nchnls = 2
0dbfs = 1
ksmps = 32

gi_Arr_1[] fillarray 1, 2, 3, 4, 5
gi_Arr_2[] fillarray 5, 4, 3, 2, 1

instr Select
if p4==1 then
  iArr[] = gi_Arr_1
else
  iArr[] = gi_Arr_2
endif
index = 0
while index < lenarray(iArr) do
  schedule("Play", index/2, 1, iArr[index])
  index += 1
od
endin

instr Play
aImp mpulse 1, p3
iFreq = mtotf:i(60 + (p4-1)*2)
aTone mode aImp, iFreq, 100
out aTone, aTone
endin

</CsInstruments>
```

```
<CsScore>
i "Select" 0 4 1
i "Select" + . 2
</CsScore>
</CsoundSynthesizer>
;example by joachim heintz
```

Implicit as Opcode Output

Some opcodes generate arrays as output. The size of the array depends on the opcode's input. The `diskin` opcode, for instance, returns an array which has the same size as the number of channels in the audio file. So in the following code, the first array `aRead_A` will have one element (as the audio file is mono), the second array `aRead_B` will have two elements (as the audio file is stereo), the third array `aRead_C` will have four elements (as the audio file is quadro).

```
aRead_A[] diskin "mono.wav"
aRead_B[] diskin "stereo.wav"
aRead_C[] diskin "quadro.wav"
```

Other opcodes which return arrays as output are `vbap`, `bformdec1`, `loscilx` for audio arrays, and `directory` for string arrays.

Types of Arrays

i- and *k*-Rate

Most arrays which are typed by the user to hold data will be either i-rate or k-rate. An i-array can only be modified at init-time, and any operation on it is only performed once, at init-time. A k-array can be modified during the performance, and any (k-) operation on it will be performed in every k-cycle (!).² Here is a simple example showing the difference:

EXAMPLE 03E02_i_k_arrays.csd

```
<CsoundSynthesizer>
<CsOptions>
-nm128 ;no sound and reduced messages
</CsOptions>
<CsInstruments>
sr = 44100
ksmps = 4410 ;10 k-cycles per second

instr 1
iArr[] fillarray 1, 2, 3
iArr[0] = iArr[0] + 10
prints "    iArr[0] = %d\n\n", iArr[0]
```

²More detailed explanation about i- and k-rate can be found in chapter 03 A

```

    endin

instr 2
kArr[] fillarray 1, 2, 3
kArr[0] = kArr[0] + 10
printks "    kArr[0] = %d\n", 0, kArr[0]
    endin

</CsInstruments>
<CsScore>
i 1 0 1
i 2 1 1
</CsScore>
</CsoundSynthesizer>
;example by joachim heintz

```

The printout is:

```

iArr[0] = 11

kArr[0] = 11
kArr[0] = 21
kArr[0] = 31
kArr[0] = 41
kArr[0] = 51
kArr[0] = 61
kArr[0] = 71
kArr[0] = 81
kArr[0] = 91
kArr[0] = 101

```

Although both instruments run for one second, the operation to increment the first array value by ten is executed only once in the *i*-rate version of the array. But in the *k*-rate version, the incrementation is repeated in each *k*-cycle - in this case every 1/10 second, but usually something around every 1/1000 second.

Audio Arrays

An audio array is a collection of audio signals. The size (length) of the audio array denotes the number of audio signals which are hold in it. In the next example, the audio array is created for two audio signals:

```
aArr[] init 2
```

The first audio signal in the array *aArr[0]* carries the output of a sine oscillator with frequency 400 Hz whereas *aArr[1]* gets 500 Hz:

```
aArr[0] poscil .2, 400
aArr[1] poscil .2, 500
```

A percussive envelope *aEnv* is generated with the *transeg* opcode. The last line
*out aArr*aEnv*

multiplies the envelope with each element of the array, and the *out* opcode outputs the result to both channels of the audio output device.

EXAMPLE 03E03_Audio_array.csd

```

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

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

instr AudioArray
aArr[] init 2
aArr[0] oscil .2, 400
aArr[1] oscil .2, 500
aEnv transeg 1, p3, -3, 0
        out aArr*aEnv
endin

</CsInstruments>
<CsScore>
i "AudioArray" 0 3
</CsScore>
</CsoundSynthesizer>
;example by joachim heintz

```

As mentioned above, some opcodes create audio arrays implicitly according to the number of input audio signals:

```
arr[] diskin "7chnls.aiff", 1
```

This code will create an audio array of size 7 according to the seven channel input file.

Strings

Arrays of strings can be very useful in many situations, for instance while working with file paths.³
The array can be filled by one of the ways described above, for instance:

```
S_array[] fillarray "one", "two", "three"
```

In this case, *S_array* is of length 3. The elements can be accessed by indexing as usual, for instance

```
puts S_array[1], 1
```

will return "two".

The [directory](#) opcode looks for all files in a directory and returns an array containing the file names:

EXAMPLE 03E04_Directory.csd

```

<CsoundSynthesizer>
<CsOptions>
```

³You cannot currently have a mixture of numbers and strings in an array, but you can convert a string to a number with the [strtod](#) opcode.

```

-odac -d
</CsOptions>
<CsInstruments>

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

instr Files
S_array[] directory "."
iNumFiles lenarray S_array
prints "Number of files in %s = %d\n", pwd(), iNumFiles
printarray S_array
endin

</CsInstruments>
<CsScore>
i "Files" 0 0
</CsScore>
</CsoundSynthesizer>
;example by joachim heintz

```

Which prints for instance:

```

Number of files in /home/xy/Desktop = 3
"test.cs", "test.wav", "test2.cs"

```

Local or Global

Like any other variable in Csound, an array usually has a local scope. This means that it is only valid in the instrument in which it has been defined. If an array is supposed to be valid across instruments, the variable name must be prefixed with the character **g**, (as is done with other types of global variable in Csound). The next example demonstrates local and global arrays at both *i*- and *k*-rate.

EXAMPLE 03E05_Local_vs_global_arrays.csd

```

<CsoundSynthesizer>
<CsOptions>
-nm128 ;no sound and reduced messages
</CsOptions>
<CsInstruments>
ksmps = 32

instr i_local
iArr[] array 1, 2, 3
    prints " iArr[0] = %d   iArr[1] = %d   iArr[2] = %d\n",
           iArr[0], iArr[1], iArr[2]
endin

instr i_local_diff ;same name, different content
iArr[] array 4, 5, 6
    prints " iArr[0] = %d   iArr[1] = %d   iArr[2] = %d\n",
           iArr[0], iArr[1], iArr[2]
endin

```

```

instr i_global
giArr[] array 11, 12, 13
endin

instr i_global_read ;understands giArr though not defined here
    prints " giArr[0] = %d  giArr[1] = %d  giArr[2] = %d\n",
            giArr[0], giArr[1], giArr[2]
endin

instr k_local
kArr[] array -1, -2, -3
    printks "  kArr[0] = %d  kArr[1] = %d  kArr[2] = %d\n",
            0, kArr[0], kArr[1], kArr[2]
    turnoff
endin

instr k_local_diff
kArr[] array -4, -5, -6
    printks "  kArr[0] = %d  kArr[1] = %d  kArr[2] = %d\n",
            0, kArr[0], kArr[1], kArr[2]
    turnoff
endin

instr k_global
gkArr[] array -11, -12, -13
    turnoff
endin

instr k_global_read
    printks "  gkArr[0] = %d  gkArr[1] = %d  gkArr[2] = %d\n",
            0, gkArr[0], gkArr[1], gkArr[2]
    turnoff
endin
</CsInstruments>
<CsScore>
i "i_local" 0 0
i "i_local_diff" 0 0
i "i_global" 0 0
i "i_global_read" 0 0
i "k_local" 0 1
i "k_local_diff" 0 1
i "k_global" 0 1
i "k_global_read" 0 1
</CsScore>
</CsoundSynthesizer>
;example by joachim heintz

```

Different Rates between Array and Index

Usually the first character of a variable name in Csound shows whether it is *i*-rate or *k*-rate or *a*-rate. But for arrays, we have actually two signifiers: the array variable type, and the index type. If both coincide, it is easy:

- **i_array[i_index]** reads and writes at i-time
- **k_array[k_index]** reads and writes at k-time

For audio arrays, we must distinguish between the audio vector itself which is updated sample by sample, and the array as container which can be updated at k-time. (Imagine an audio array

whichs index switches each control cycle between 0 and 1; thus switching each k-time between the audio vector of both signals.) So the coincidence between variable and index rate is here:

- **a**_array[*k_index*] reads and writes at k-time

But what to do if array type and index type do not coincide? In general, the index type will then determine whether the array is read or written only once (at init-time) or at each *k*-cycle. This is valid in particular for *S* arrays (containing strings). Other cases are:

- **i**_array[*k_index*] reads at k-time; writing is not possible (yields a runtime error)
- **k**_array[*i_index*] reads and writes at k-rate
- **a**_array[*i_index*] reads and writes at k-rate

Init Values of *k*-Arrays

In case we want to retrieve the value of a *k*-array at init time, a special version of the *i()* feature must be used. For usual *k*-variables, a simple *i(kVar)* works, for instance ...

```
instr 1
    gkLine linseg 1, 1, 2
    schedule 2, .5, 0
    endin
    schedule(1,0,1)
instr 2
    iLine = i(gkLine)
    print iLine
    endin
```

... will print: *iLine = 1.499.*

This expression can **not** be used for arrays:

```
kArray[] fillarray 1, 2, 3
iFirst = i(kArray[0])
print iFirst
```

This will return an error. For this purpose, the *i()* expression gets a second argument which signifies the index:

```
kArray[] fillarray 1, 2, 3
iFirst = i(kArray, 0)
print iFirst
```

This will print: *iFirst = 1.000.*

Operations on Arrays

Analyse

lenarray – Array Length

The opcode **lenarray** reports the length of an array.

```
iArr[] fillarray 0, 1, 2, 3, 4
iLen lenarray iArr ; -> 5
aArr[] diskin "my_stereo_file.wav"
iLen lenarray aArr ; -> 2
S_array[] fillarray "foo", "bar"
iLen lenarray S_array ; -> 2
```

For reporting the length of multidimensional arrays, **lenarray** has an additional argument denoting the dimension. The default is 1 for the first dimension.

```
kArr[][] init 9, 5
iLen1 lenarray kArr ; -> 9
iLen2 lenarray kArr, 2 ; -> 5
kArrr[][][] init 7, 9, 5
iLen1 lenarray kArrr, 1 ; -> 7
iLen2 lenarray kArrr, 2 ; -> 9
iLen3 lenarray kArrr, 3 ; -> 5
```

By using functional syntax, **lenarray()** will report the array length at init-time. If the array length is being changed during performance, **lenarray:k()** must be used to report this.

minarray, maxarray – Smallest/Largest Element

The opcodes **minarray** and **maxarray** return the smallest or largest element of a numerical array:

```
iArr[] fillarray 4, -2, 3, 10, 0
print minarray:i(iArr) ; -> -2
print maxarray:i(iArr) ; -> 10
```

sumarray – Sum of all Elements

This is an example for **sumarray**:

```
iArr[] fillarray 4, -2, 3, -10, 0
print sumarray(iArr) ; -> -5
```

cmp – Compare with another Array or with Scalars

The **cmp** opcode offers quite extended possibilities to compare an array to numbers or to another array. The following example investigates in line 18 whether the elements of the array [1,2,3,4,5] are larger or equal 3. Line 20 tests whether the elements are larger than 1 and smaller or equal 4. Line 22 performs an element by element comparison with the array [3,5,1,4,2], asking for larger elements in the original array.

EXAMPLE 03E06_cmp.csd

```

<CsoundSynthesizer>
<CsOptions>
-m0
</CsOptions>
<CsInstruments>

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

giArray[] fillarray 1, 2, 3, 4, 5
giCmpArray[] fillarray 3, 5, 1, 4, 2

instr Compare
printarray giArray, "%d", "Array:"
printarray giCmpArray, "%d", "CmpArray:"
iResult[] cmp giArray, ">=", 3
printarray iResult, "%d", "Array >= 3?"
iResult[] cmp 1, "<", giArray, "<=", 4
printarray iResult, "%d", "1 < Array <= 4?"
iResult[] cmp giArray, ">", giCmpArray
printarray iResult, "%d", "Array > CmpArray?"
endin

</CsInstruments>
<CsScore>
i "Compare" 0 1
</CsScore>
</CsoundSynthesizer>
;example by eduardo moguillansky and joachim heintz

```

The printout is:

```

Array:
1 2 3 4 5
CmpArray:
3 5 1 4 2
Array >= 3?
0 0 1 1 1
1 < Array <= 4?
0 1 1 1 0
Array > CmpArray?
0 0 1 0 1

```

Content Modifications

scalearray – Scale Values

The **scalearray** opcode destructively changes the content of an array according to a new minimum and maximum:

```

iArr[] fillarray 1, 3, 9, 5, 6, -1, 17
scalearray iArr, 1, 3
printarray iArr ; -> 1.2222 1.4444 2.1111 1.6667 1.7778 1.0000 3.0000

```

Optional a range of the array can be selected for the operation; in this example from index 0 to index 4:

```
iArr[] fillarray 1, 3, 9, 5, 6, -1, 17
scalearray iArr, 1, 3, 0, 4
printarray iArr ; -> 1.0000 1.5000 3.0000 2.0000 6.0000 -1.0000 17.0000
```

sorta/sortd – Sort in Ascending/Descending Order

The opcodes **sorta** and **sortd** return an array in which the elements of the input array are sorted in ascending or descending order. The input array is left untouched.

```
iArr[] fillarray 1, 3, 9, 5, 6, -1, 17
iAsc[] sorta iArr
iDesc[] sortd iArr
printarray iAsc, "%d", "Sorted ascending:"
printarray iDesc, "%d", "Sorted descending:"
printarray iArr, "%d", "Original array:"
```

Prints:

```
Sorted ascending:
-1 1 3 5 6 9 17
Sorted descending:
17 9 6 5 3 1 -1
Original array:
1 3 9 5 6 -1 17
```

limit – Limit Values

The **limit** opcode sets a lower and upper limit to which any value off boundaries is restricted.

```
iArr[] fillarray 1, 3, 9, 5, 6, -1, 17
iLimit[] limit iArr, 0, 7
printarray(iLimit, "%d") ; -> 1 3 7 5 6 0 7
```

interleave/deinterleave

As the name suggests, the **interleave** opcode creates a new array in alternating the values of two input arrays. This operation is meant for vectors (one-dimensional arrays) only.

```
iArr1[] genarray 1,5
iArr2[] genarray -1,-5,-1
iArr[] interleave iArr1, iArr2
printarray iArr1, "%d", "array 1:"
printarray iArr2, "%d", "array 2:"
printarray iArr, "%d", "interleaved:"
```

Which prints:

```
array 1:
1 2 3 4 5
array 2:
-1 -2 -3 -4 -5
interleaved:
1 -1 2 -2 3 -3 4 -4 5 -5
```

And vice versa, **deinterleave** returns two arrays from one input array in alternating its values:

```
iArr[] genarray 1,10
iArr1[], iArr2[] deinterleave iArr
printarray iArr, "%d", "input array:"
```

```
printarray iArr1, "%d", "deinterleaved 1:"
printarray iArr2, "%d", "deinterleaved 2:"
```

Which prints:

```
input array:
1 2 3 4 5 6 7 8 9 10
deinterleaved 1:
1 3 5 7 9
deinterleaved 2:
2 4 6 8 10
```

Size Modifications

slicearray – New Array as Slice

The *slicearray* opcode creates a new array from an existing one. In addition to the input array the first and the last (included) index must be specified:

```
iArr[] fillarray 1, 3, 9, 5, 6, -1, 17
iSlice[] slicearray iArr, 1, 3
printarray(iSlice, "%d") ; -> 3 9 5
SArr[] fillarray "bla", "blo", "bli"
Slice[] slicearray SArr, 1, 2
printarray(Slice) ; -> "blo", "bli"
```

An optional argument defines the increment which is one by default:

```
iArr[] fillarray 1, 3, 9, 5, 6, -1, 17
iSlice1[] slicearray iArr, 0, 5
printarray(iSlice1, "%d") ; -> 1 3 9 5 6 -1
iSlice2[] slicearray iArr, 0, 5, 2
printarray(iSlice2, "%d") ; -> 1 9 6
```

trim/trim_i – Lengthen or Shorten Array

Arrays have a fixed length, and it may be needed to shorten or lengthen it. *trim_i* works for any array at i-rate:

```
iArr[] fillarray 1, 3, 9, 5, 6, -1, 17
trim_i iArr, 3
printarray(iArr, "%d") -> 1 3 9
kArr[] fillarray 1, 3, 9, 5, 6, -1, 17
trim_i kArr, 5
printarray(kArr, 1, "%d") ; -> 1 3 9 5 6
aArr[] diskin "fox.wav"
prints "%d\n", lenarray(aArr) ; -> 1
trim_i aArr, 2
prints "%d\n", lenarray(aArr) ; -> 2
SArr[] fillarray "a", "b", "c", "d"
trim_i SArr, 2
printarray(SArr) ; -> "a", "b"
```

If a length bigger than the current array size is required, the additional elements are set to zero. This can only be used for the init-time version *trim_i*:

```
iArr[] fillarray 1, 3, 9
trim_i iArr, 5
printarray(iArr, "%d") ; -> 1 3 9 0 0
```

At performance rather than initialization `trim` can be used. This codes reduces the array size by one for each trigger signal:

```
instr 1
kArr[] fillarray 1, 3, 9, 5, 6, -1, 17
kTrig metro 1
if kTrig==1 then
  trim kArr, lenarray:k(kArr)-1
  printarray kArr,-1,"%d"
endif
endin
schedule(1,0,5)
```

Prints:

```
1 3 9 5 6 -1
1 3 9 5 6
1 3 9 5
1 3 9
1 3
```

Growing an array during performance is not possible in Csound, because memory will only be allocated at initialization. This is the reason that only `trim_i` can be used for this purpose.

Format Interchange

copyf2array – Function Table to Array

As function tables have been the classical way of working with vectors in Csound, switching between them and the array facility introduced in Csound 6 is a basic operation. Copying data from a function table to a vector is done by `copyf2array`. The following example copies a sine function table (8 points) to an array and prints the array content:

```
iFtSine ftgen 0, 0, 8, 10, 1
iArr[] init 8
copyf2array iArr, iFtSine
printarray iArr
; -> 0.0000 0.7071 1.0000 0.7071 0.0000 -0.7071 -1.0000 -0.7071
```

copya2ftab – Array to Function Table

The `copya2ftab` opcode copies an array content to a function table. In the example a function table of size 10 is created, and an array filled with the integers from 1 to 10. The array content is then copied into the function table, and the resulting function table is printed via a `while` loop.

```
iTable ftgen 0, 0, 10, 2, 0
iArr[] genarray 1, 10
copya2ftab iArr, iTable
index = 0
while index < ftlen(iTable) do
  prints "%d ", table:i(index, iTable)
```

```
index += 1
od
```

The printout is:

```
1 2 3 4 5 6 7 8 9 10
```

tab2array – Function Table Slice to Array

The `tab2array` opcode is similar to `copyf2array` but offers more possibilities. One difference is that the resulting array is generated by the opcode, so no need for the user to create the array in advance. This code copies the content of a 16-point saw function table into an array and prints the array:

```
iFtSaw ftgen 0, 0, 8, 10, 1, -1/2, 1/3, -1/4, 1/5, -1/6
iArr[] tab2array iFtSaw
printarray(iArr)
; -> 0.0000 0.4125 0.7638 1.0000 0.0000 -1.0000 -0.7638 -0.4125
```

This will copy the values from index 1 to index 8 (not included):

```
iFtSine ftgen 0, 0, 8, 10, 1, -1/2, 1/3, -1/4, 1/5, -1/6
iArr[] tab2array iFtSine, 1, 7
printarray(iArr)
; -> 0.4125 0.7638 1.0000 0.0000 -1.0000 -0.7638
```

And this will copy the whole array but only every second value:

```
iFtSine ftgen 0, 0, 8, 10, 1, -1/2, 1/3, -1/4, 1/5, -1/6
iArr[] tab2array iFtSine, 0, 0, 2
printarray(iArr)
; -> 0.0000 0.7638 0.0000 -0.7638
```

pvs2array/pvsfromarray – Arrays to/from FFT Data

The data of an f-signal – containing the result of a Fast Fourier Transform – can be copied into an array with the opcode `pvs2array`. The counterpart `pvsfromarray` copies the content of an array to a f-signal.

```
kFrame pvs2array kArr, fSigIn ;from f-signal fSig to array kArr
fSigOut pvsfromarray kArr [,ihopsize, iwinsize, iwintype]
```

Some care is needed to use these opcodes correctly:

- The array `kArr` must be declared in advance to its usage in these opcodes, usually with `init`.
- The size of this array depends on the FFT size of the f-signal `fSigIn`. If the FFT size is N, the f-signal will contain $N/2+1$ amplitude-frequency pairs. For instance, if the FFT size is 1024, the FFT will write out 513 bins, each bin containing one value for amplitude and one value for frequency. So to store all these values, the array must have a size of 1026. In general, the size of `kArr` equals FFT-size plus two.
- The indices 0, 2, 4, ... of `kArr` will contain the amplitudes; the indices 1, 3, 5, ... will contain the frequencies of the bins in a specific frame.
- The number of this frame is reported in the `kFrame` output of `pvs2array`. By this parameter you know when `pvs2array` writes new values to the array `kArr`.
- On the way back, the FFT size of `fSigOut`, which is written by `pvsfromarray`, depends on the size of `kArr`. If the size of `kArr` is 1026, the FFT size will be 1024.

- The default value for *ihopsize* is 4 (= *fftsize*/4); the default value for *inwinsize* is the *fftsize*; and the default value for *iwintype* is 1, which means a hanning window.

Here is an example that implements a spectral high-pass filter. The f-signal is written to an array and the amplitudes of the first 40 bins are then zeroed.⁴ This is only done when a new frame writes its values to the array so as not to waste rendering power.

EXAMPLE 03E07_pvs_to_from_array.csd

```

<CsoundSynthesizer>
<CsOptions>
-o dac
</CsOptions>
<CsInstruments>

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

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

instr FFT_HighPass
iifftsize = 2048 ;fft size set to pvstanal default
fsrc pvstanal 1, 1, 1, gfil ;create fsig stream from function table
kArr[] init iifftsize+2 ;create array for bin data
kflag pvs2array kArr, fsrc ;export data to array

;if kflag has reported a new write action ...
if changed(kflag) == 1 then
; ... set amplitude of first 40 bins to zero:
kndx = 0
while kndx <= 80 do
  kArr[kndx] = 0
  kndx += 2 ;change only even array index = bin amplitude
od
endif

fres pvsfromarray kArr ;read modified data back to fres
aout pvsynth fres ;and resynth
out aout, aout

endin
</CsInstruments>
<CsScore>
i "FFT_HighPass" 0 2.7
</CsScore>
</CsoundSynthesizer>
;example by joachim heintz

```

⁴As sample rate is here 44100, and fftsize is 2048, each bin has a frequency range of $44100 / 2048 = 21.533$ Hz. Bin 0 looks for frequencies around 0 Hz, bin 1 for frequencies around 21.533 Hz, bin 2 around 43.066 Hz, and so on. So setting the first 40 bin amplitudes to 0 means that no frequencies will be resynthesized which are lower than bin 40 which is centered at $40 * 21.533 = 861.328$ Hz.

1D - 2D Interchange

reshapearray – Change Array Dimension

With **reshapearray** a one-dimensional array can be transformed in a two-dimensional one, and vice versa. In the following example, a 1D array of 12 elements is first printed and then transformed in a 2D array with 3 lines and 4 columns:

```
iArr[] genarray 1, 12
printarray iArr, "%d", "1D array:"
reshapearray iArr, 3, 4
printarray iArr, "%d", "2D array:"
```

This is the printout:

```
1D array:
1 2 3 4 5 6 7 8 9 10 11 12
2D array:
0: 1 2 3 4
1: 5 6 7 8
2: 9 10 11 12
```

getrow/getcol – Get Row/Column from a 2D Array

The opcodes **getrow** and **getcol** return the content of a 2D array's row or column as a 1D array:

```
iArr[][] init 3, 4
iArr fillarray 1,2,3,4,5,6,7,8,9,10,11,12
printarray iArr, "%d", "2D array:"
iRow1[] getrow iArr, 0
printarray iRow1, "%d", "First row:"
iCol1[] getcol iArr, 0
printarray iCol1, "%d", "First columns:"
```

Prints:

```
2D array:
0: 1 2 3 4
1: 5 6 7 8
2: 9 10 11 12
First row:
1 2 3 4
First columns:
1 5 9
```

setrow/setcol - Set Row/Column of a 2D Array

The opcodes **setrow** and **setcol** assign a 1D array as row or column of a 2D array:

```
iArr[][] init 3, 4
printarray iArr, "%d", "2D array empty:"
iRow[] fillarray 1, 2, 3, 4
iArr setrow iRow, 0
printarray iArr, "%d", "2D array with first row:"
iCol[] fillarray -1, -2, -3
iArr setcol iCol, 3
printarray iArr, "%d", "2D array with fourth column:"
```

Prints:

```
2D array empty:
 0: 0 0 0 0
 1: 0 0 0 0
 2: 0 0 0 0
2D array with first row:
 0: 1 2 3 4
 1: 0 0 0 0
 2: 0 0 0 0
2D array with fourth column:
 0: 1 2 3 -1
 1: 0 0 0 -2
 2: 0 0 0 -3
```

getrowlin – Get Row from a 2D Array and Interpolate

The `getrowlin` opcode is similar to `getrow` but interpolates between adjacent rows of a matrix if a non-integer number is given.

```
kArr[][] init 3, 4
kArr fillarray 1,2,3,4,5,6,7,8,9,10,11,12
printarray kArr, 1, "%d", "2D array:"
kRow[] getrowlin kArr, 0.5
printarray kRow, 1, "%d", "Row 0.5:"
```

The 0.5th row means an interpolation between first and second row, so this is the output:

```
2D array:
 0: 1 2 3 4
 1: 5 6 7 8
 2: 9 10 11 12
Row 0.5:
 3 4 5 6
```

Functions

Arithmetic Operators

The four basic operators `+`, `-`, `*` and `/` can directly be applied to an array, either with a scalar or a second array as argument.

All operations can be applied to the input array itself (changing its content destructively), or can create a new array as result. This is a simple example for the scalar addition:

```
iArr[] fillarray 1, 2, 3
iNew[] = iArr + 10 ; -> 11 12 13 as new array
iArr += 10 ; iArr is now 11 12 13
```

It also works for a 2D matrix:

```
iArr[][] init 2, 3
iArr fillarray 1, 2, 3, 4, 5, 6
printarray(iArr, "%d", "original array:")
iArr += 10
printarray(iArr, "%d", "modified array:")
```

Which prints:

```
original array:
 0: 1 2 3
 1: 4 5 6
modified array:
 0: 11 12 13
 1: 14 15 16
```

Both possibilities – creating a new array or modifying the existing one – are also valid if a second array is given as argument:

```
iArr[] fillarray 1, 2, 3
iArg[] fillarray 10, 20, 30
iNew[] = iArr + iArg ; -> 11 22 33 as new array
iArr += iArg ; iArr is now 11 22 33
```

Both arrays must have the same size, but it also works for 2D arrays:

```
iArr[][] init 2, 3
iArr fillarray 1, 2, 3, 4, 5, 6
printarray(iArr, "%d", "original array:")
iArg[][] init 2, 3
iArg fillarray 3, 4, 5, 6, 7, 8
printarray(iArg, "%d", "argument array:")
iArr += iArg
printarray(iArr, "%d", "modified array:")
```

Which prints:

```
original array:
 0: 1 2 3
 1: 4 5 6
argument array:
 0: 3 4 5
 1: 6 7 8
modified array:
 0: 4 6 8
 1: 10 12 14
```

Unary Functions

These unary functions accept arrays as input:

- `ceil` – next integer above
- `floor` – next integer below
- `round` – round to next integer
- `int` – integer part
- `frac` – fractional part
- `powoftwo` – power of two
- `abs` – absolute value
- `log2` – logarithm base two
- `log10` – logarithm base ten
- `log` – natural logarithm, optional any base
- `exp` – power of e
- `sqrt` – square root
- `cos` – cosine
- `sin` – sine
- `tan` – tangent

- **cosinv** – arccosine
- **sininv** – arcsine
- **taninv** – arctangent
- **sinh** – hyperbolic sine
- **cosh** – hyperbolic cosine
- **tanh** – hyperbolic tangent
- **cbrt** – cubic root

Some simple examples:

```
iArr[] fillarray 1.1, 2.2, 3.3
iCeil[] ceil iArr ; -> 2 3 4
iInt[] int iArr ; -> 1 2 3
iFrac[] frac iArr ; -> 0.1 0.2 0.3
iPow2[] powoftwo iArr ; -> 2.1435 4.5948 9.8492
```

maparray

The **maparray** opcode was used in early array implementation to apply a unary function to every element of a 1D array. In case a function is not in the list above, this old solution may work.

Binary Functions

These binary functions can take arrays as input:

- **pow** – power of two arguments
- **hypot** – Euclidean distance $\sqrt{a^2 + b^2}$
- **fmod** – remainder (modulo) for arrays value by value
- **fmin** – minimum of two arrays value by value
- **fmax** – maximum of two arrays value by value
- **dor** – dot product of two arrays

For instance:

```
iBase[] fillarray 1.1, 2.2, 3.3
iExp[] fillarray 2, -2, 0
iBasPow2[] pow iBase, 2 ; -> 1.2100 4.8400 10.8900
iBasExp[] pow iBase, iExp ; -> 1.2100 0.2066 1.0000
```

Print

The **printarray** opcode is easy to use and offers all possibilities to print out array contents.

Arrays in UDOs

Input and Output Declaration

Writing a [User Defined Opcode](#) can extend Csound's array facilities to any desired own function. The usage of arrays in the opcode definition is straightforward; most important is to remember that type (*i*, *k*, *a*, or *S*) and dimension of an input array must be declared in two places:

- in the opcode *intypes* list as *i[]*, *i[][]* etc;
- in the *xin* list as variable name, including brackets.

This is a simple UDO definition which returns the first element of a given 1D *k*-array. Note that in the *intype* list it is declared as *k[]*, whereas in the input argument list it is declared as *kArr[]*.

```
opcode FirstEl, k, k[]
kArr[] xin
kOut = kArr[0]
xout kOut
endop
```

The output declaration is done quite similar: abstract type declaration in the *outtypes* list, and variable name in the UDO body. Here the usual naming conventions are valid, as explained at the beginning of this chapter (first occurrence with brackets, then without brackets).

This is an example which creates an *i*-array of *N* elements, applying recursively a given ratio on each element. The output array is declared as *i[]* in the *outtypes* list, and as variable *first* as *iOut[]* then only *iOut* in the body.

```
opcode GeoSer,i[],iii
iStart, iRatio, iSize xin
iOut[] init iSize
indx = 0
while indx < iSize do
  iOut[indx] = iStart
  iStart *= iRatio
  indx += 1
od
xout iOut
```

The call

```
instr 1
iSeries[] GeoSer 2, 3, 5
printarray(iSeries,"%d")
endin
schedule(1,0,0)
```

will print:

```
2 6 18 54 162
```

As an expert note it should be mentioned that UDOs refer to arrays by value. This means that an input array is copied into the UDO, and an output array is copied to the instrument. This can slow down the performance for large arrays and k-rate calls.

Overload

Usually we want to use a UDO for different types of arrays. The best method is to overload the function in defining the different types with the same function name. Csound will then select the appropriate version.

The following example extends the *FirstEl* function from *k*-arrays also to *i*- and *S*-arrays.

EXAMPLE 03E08_array_overload.csd

```
<CsoundSynthesizer>
<CsOptions>
-m0
</CsOptions>
<CsInstruments>
ksmps = 32

opcode FirstEl, k, k[]
kArr[] xin
kOut = kArr[0]
xout kOut
endop

opcode FirstEl, i, i[]
iArr[] xin
iOut = iArr[0]
xout iOut
endop

opcode FirstEl, S, S[]
SArr[] xin
SOut = SArr[0]
xout SOut
endop

instr Test
iTest[] fillarray 1, 2, 3
kTest[] fillarray 4, 5, 6
STest[] fillarray "x", "y", "z"
prints "First element of i-array: %d\n", FirstEl(iTest)
printks "First element of k-array: %d\n", 0, FirstEl(kTest)
printf "First element of S-array: %s\n", 1, FirstEl(STest)
turnoff
endin
</CsInstruments>
<CsScore>
i "Test" 0 1
</CsScore>
</CsoundSynthesizer>
;example by joachim heintz
```

The output is:

```
First element of i-array: 1
First element of k-array: 4
First element of S-array: x
```

Example: Array Shuffle

In composition we sometimes use a list of values and want to get many random permutations of this list. Some programming languages call this *shuffle*. It is not difficult to write it as UDO. First we create the output array, having the same length as the input array. Then we randomly choose one element from the input array. This element is copied into the first position of the output array. Then all elements in the input array right to this element are shifted one position to the left, thus overriding the previously selected element. For instance, if the input array is

```
1 2 3 4 5 6 7
```

and element 4 has been selected randomly, and copied into the output array at first position, the elements 5 6 7 will be shifted one position to the left, so that input array changes to

```
1 2 3 5 6 7
```

This procedure is repeated again and again; in the next run only looking amongst six rather than seven elements.

As Csound has no random opcode for integers, this is first defined as helper function: *RndInt* returns a random integer between *iStart* and *iEnd* (included).⁵

EXAMPLE 03E09_Shuffle.csd

```
<CsoundSynthesizer>
<CsOptions>
-m0
</CsOptions>
<CsInstruments>
ksmps = 32
seed 0

opcode RndInt, i, ii
    iStart, iEnd xin
    iRnd random iStart, iEnd+.999
    iRndInt = int(iRnd)
    xout iRndInt
endop

opcode ArrShuffle, i[], i[]
    iInArr[] xin
    iLen = lenarray(iInArr)
    iOutArr[] init iLen
    iIndx = 0
    iEnd = iLen-1
    while iIndx < iLen do
        ;get one random element and put it in iOutArr
        iRndIndx RndInt 0, iEnd
        iOutArr[iIndx] = iInArr[iRndIndx]
        ;shift the elements after this one to the left
        while iRndIndx < iEnd do
            iInArr[iRndIndx] = iInArr[iRndIndx+1]
            iRndIndx += 1
    od
```

⁵More UDOs can be found at <https://github.com/csudo/csudo/>, <https://github.com/kunstmusik/libsyi> and other places.

```
;reset end and increase counter
iIndx += 1
iEnd -= 1
od
xout iOutArr
endop

instr Test
iValues[] fillarray 1, 2, 3, 4, 5, 6, 7
indx = 0
while indx < 5 do
  iOut[] ArrShuffle iValues
  printarray(iOut,"%d")
  indx += 1
od
endin

</CsInstruments>
<CsScore>
i "Test" 0 0
</CsScore>
</CsoundSynthesizer>
;example by joachim heintz
```

The output is, for instance:

```
7 3 4 5 2 6 1
1 3 2 7 4 5 6
3 5 1 4 7 2 6
6 2 5 1 7 4 3
4 7 2 5 6 1 3
```

03 F. LIVE EVENTS

Note: This chapter is not about live coding. Live coding should be covered in future in an own chapter. For now, have a look at live.csound.com and Steven Yi's related [csound-live-code](#) repository.

The basic concept of Csound from the early days of the program is still valid and useful because it is a musically familiar 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*.

Whenever any Csound code is executed, it has to be compiled first. Since Csound6, you can change the code of any running Csound instance, and recompile it on the fly. There are basically two opcodes for this *live coding*: `compileorc` re-compiles any existing orc file, whereas `compilestr` compiles any string. At the end of this chapter, we will present some simple examples for both methods, followed by a description how to re-compile code on the fly in CsoundQt.

The 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 the widgets in CsoundQt, Cabbage and Blue), from the API. 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 ...

Order of Execution Revisited

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**.

It is worth to have a closer look to what is happening exactly in time if you trigger an instrument from inside another instrument. The first example shows the result when instrument 2 triggers instrument 1 and instrument 3 **at init-time**.

```
EXAMPLE 03F01 OrderOfExc.event.i.csd

<CsoundSynthesizer>
<CsOptions>
-nm0
</CsOptions>
<CsInstruments>
sr = 44100
ksmps = 441

instr 1
kCycle timek
prints "Instrument 1 is here at initialization.\n"
printks "Instrument 1: kCycle = %d\n", 0, kCycle
endin

instr 2
kCycle timek
prints "Instrument 2 is here at initialization.\n"
printks "Instrument 2: kCycle = %d\n", 0, kCycle
event_i "i", 3, 0, .02
event_i "i", 1, 0, .02
endin

instr 3
kCycle timek
prints "Instrument 3 is here at initialization.\n"
printks "Instrument 3: kCycle = %d\n", 0, kCycle
endin

</CsInstruments>
<CsScore>
i 2 0 .02
</CsScore>
</CsoundSynthesizer>
;example by joachim heintz
```

This is the output:

```
Instrument 2 is here at initialization.
Instrument 3 is here at initialization.
Instrument 1 is here at initialization.
Instrument 1: kCycle = 1
Instrument 2: kCycle = 1
Instrument 3: kCycle = 1
Instrument 1: kCycle = 2
Instrument 2: kCycle = 2
Instrument 3: kCycle = 2
```

Instrument 2 is the first one to initialize, because it is the only one which is called by the score. Then instrument 3 is initialized, because it is called first by instrument 2. The last one is instrument 1. All this is done before the actual performance begins. In the performance itself, starting from the first control cycle, all instruments are executed by their order.

Let us compare now what is happening when instrument 2 calls instrument 1 and 3 **during the performance** (= at k-time):

EXAMPLE 03F02_OrderOfExc_event_k.csd

```

<CsoundSynthesizer>
<CsOptions>
-nm0
</CsOptions>
<CsInstruments>
sr = 44100
ksmps = 441
0dbfs = 1
nchnls = 1

instr 1
kCycle timek
prints "Instrument 1 is here at initialization.\n"
printks "Instrument 1: kCycle = %d\n", 0, kCycle
endin

instr 2
kCycle timek
prints " Instrument 2 is here at initialization.\n"
printks " Instrument 2: kCycle = %d\n", 0, kCycle
if kCycle == 1 then
event "i", 3, 0, .02
event "i", 1, 0, .02
endif
printks " Instrument 2: still in kCycle = %d\n", 0, kCycle
endin

instr 3
kCycle timek
prints " Instrument 3 is here at initialization.\n"
printks " Instrument 3: kCycle = %d\n", 0, kCycle
endin

instr 4
kCycle timek
prints " Instrument 4 is here at initialization.\n"
printks " Instrument 4: kCycle = %d\n", 0, kCycle
endin

</CsInstruments>
<CsScore>
i 4 0 .02
i 2 0 .02
</CsScore>
</CsoundSynthesizer>
;example by joachim heintz

```

This is the output:

```

Instrument 2 is here at initialization.
Instrument 4 is here at initialization.
Instrument 2: kCycle = 1
Instrument 2: still in kCycle = 1
Instrument 4: kCycle = 1
Instrument 3 is here at initialization.
Instrument 1 is here at initialization.
Instrument 1: kCycle = 2
Instrument 2: kCycle = 2
Instrument 2: still in kCycle = 2
Instrument 3: kCycle = 2
Instrument 4: kCycle = 2

```

Instrument 2 starts with its init-pass, and then instrument 4 is initialized. As you see, the reverse order of the scorelines has no effect; the instruments which start at the same time are executed in ascending order, depending on their numbers.

In this first cycle, instrument 2 calls instrument 3 and 1. As we see by the output of instrument 4, the whole control cycle is finished first, before instrument 3 and 1 (in this order) are initialized. These both instruments start their performance in cycle number two, where they find themselves in the usual order: instrument 1 before instrument 2, then instrument 3 before instrument 4.

Usually you will not need to know all of this with such precise timing. But in case you experience any problems, a clearer awareness of the process may help.

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 03F03_Score_tricks.csd

```
<CsoundSynthesizer>
<CsOptions>
-odac
</CsOptions>
<CsInstruments>
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 indefinitely (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 again starts 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>
;example by joachim heintz

```

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 Note-On 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 03F04_Midi_triggered_events.csd

```

<CsoundSynthesizer>
<CsOptions>
-Ma -odac
</CsOptions>
<CsInstruments>
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      oscili   .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>
;example by joachim heintz

```

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 CsoundQt'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 03F05_FLTk_triggered_events.csd

```

<CsoundSynthesizer>
<CsOptions>
-odac
</CsOptions>
<CsInstruments>
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   poscil  ampdB(idb), cpsOct(ioct)
aEnv   transeg  1, p3, -10, 0
outs    aSig*aEnv, aSig*aEnv
endin

instr 2
exitnow
endin

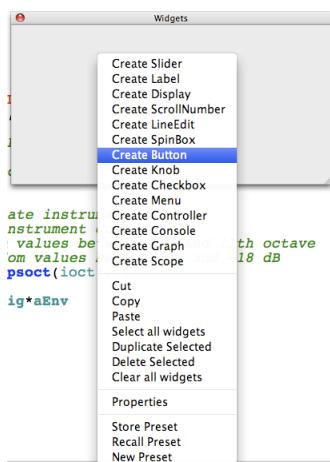
</CsInstruments>
<CsScore>
f 0 36000
e
</CsScore>
</CsoundSynthesizer>
;example by joachim heintz

```

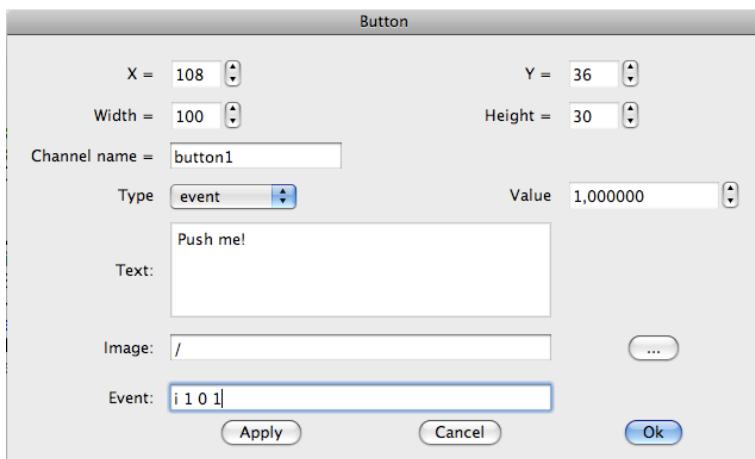
Note that in this example the duration of an instrument event is recalculated when the instrument is initialised. This is done using the statement `p3 = i....`. This can be a useful technique if you want the duration that an instrument plays for to be different each time it is called. In this example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

CsoundQt Button

In CsoundQt, 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.



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
Odbfs = 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), cpscct(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 CsoundQt, read the CsoundQt chapter in the *Frontends* section of this manual.

Using A Realtime Score

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 03F06_Commandline_rt_events.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>

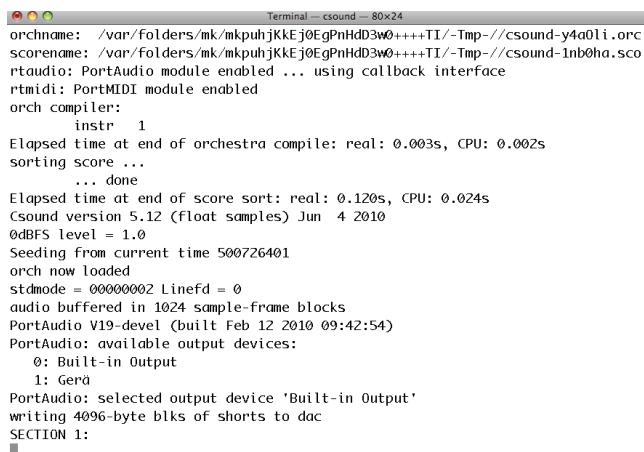
```

If you run it by typing and returning a command line 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:

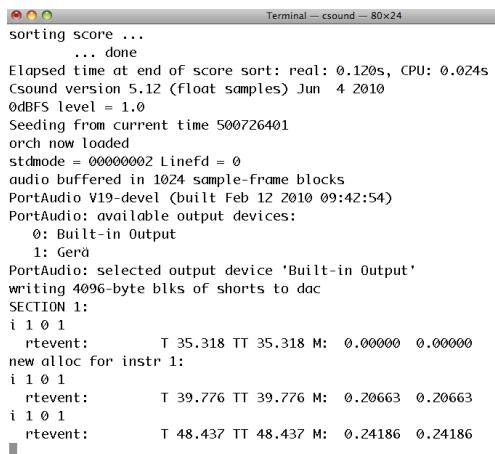


```

Terminal — csound — 80x24
orchname: /var/folders/mk/mkpuhjKEj0EgPnhd3w0++++TI/-Tmp-/csound-y4a0li.orc
scorename: /var/folders/mk/mkpuhjKEj0EgPnhd3w0++++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: Gérard
PortAudio: selected output device 'Built-in Output'
writing 4096-byte blks of shorts to dac
SECTION 1:

```

If you now type the line *i 10 1* and press return, you should hear that instrument 1 has been executed.
After three times your messages may look like this:



```

Terminal — csound — 80x24
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: Gerð
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

```

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 to high the second instrument is triggered, if it crosses from high to low the third instrument is triggered.

EXAMPLE 03F07_Event_by_condition.csd

```

<CsoundSynthesizer>
<CsOptions>
-iadc -odac
</CsOptions>
<CsInstruments>
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      poscil    .2, 500
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      poscil    .2, 400
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>
;example by joachim heintz

```

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 03F08_Generate_event_pool.csd

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

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 03F09_Events sprintf.csd

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

```

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.

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 03F10_Events_collected.csd

```

<CsoundSynthesizer>
<CsOptions>
-odac
</CsOptions>
<CsInstruments>
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 %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        outs    asig*aenv, asig*aenv
  endin

</CsInstruments>
<CsScore>
i 1 0 1
i 2 14 1
e
</CsScore>
</CsoundSynthesizer>
;example by joachim heintz

```

Using Time Loops

As discussed above in the chapter about control structures, a time loop can be built in Csound with the `timeout` 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 instru-

ment 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 03F11_Events_time_loop.csd

```

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

```

Which Opcode Should I Use?

Csound users are often confused about the variety of opcodes available to trigger instrument events. Should I use event, scoreline, schedule or schedkwhen? Should I use event or event_i?

Let us start with the latter, which actually leads to the general question about *i*-rate and *k*-rate opcodes.¹ In short: Using **event_i** (the *i*-rate version) will only trigger an event **once**, when the instrument in which this opcode works is initiated. Using **event** (the *k*-rate version) will trigger an event potentially **again and again**, as long as the instrument runs, in each control cycle. This is a very simple example:

EXAMPLE 03F12_event_i_vs_event.csd

```

<CsoundSynthesizer>
<CsOptions>
-nm0
</CsOptions>
<CsInstruments>
sr=44100
ksmps = 32

;set counters for the instances of Called_i and Called_k
giInstCi init 1
giInstCk init 1

instr Call_i
;call another instrument at i-rate
event_i "i", "Called_i", 0, 1
endin

instr Call_k

```

¹See chapter 03A about Initialization and Performance Pass for a detailed discussion.

```

;call another instrument at k-rate
event "i", "Called_k", 0, 1
endin

instr Called_i
;report that instrument starts and which instance
prints "Instance #%" of Called_i is starting!\n", giInstCi
;increment number of instance for next instance
giInstCi += 1
endin

instr Called_k
;report that instrument starts and which instance
prints " Instance #%" of Called_k is starting!\n", giInstCk
;increment number of instance for next instance
giInstCk += 1
endin

</CsInstruments>
<CsScore>
;run "Call_i" for one second
i "Call_i" 0 1
;run "Call_k" for 1/100 seconds
i "Call_k" 0 0.01
</CsScore>
</CsoundSynthesizer>
;example by joachim heintz

```

Although instrument *Call_i* runs for one second, the call to instrument *Called_i* is only performed once, because it is done with `event_i`: at initialization only. But instrument *Call_k* calls one instance of *Called_k* in each control cycle; so for the duration of 0.01 seconds of running instrument *Call_k*, fourteen instances of instrument *Called_k* are being started.² So this is the output:

```

Instance #1 of Called_i is starting!
Instance #1 of Called_k is starting!
Instance #2 of Called_k is starting!
Instance #3 of Called_k is starting!
Instance #4 of Called_k is starting!
Instance #5 of Called_k is starting!
Instance #6 of Called_k is starting!
Instance #7 of Called_k is starting!
Instance #8 of Called_k is starting!
Instance #9 of Called_k is starting!
Instance #10 of Called_k is starting!
Instance #11 of Called_k is starting!
Instance #12 of Called_k is starting!
Instance #13 of Called_k is starting!
Instance #14 of Called_k is starting!

```

So the first (and probably most important) decision in asking “which opcode should I use”, is the answer to the question: “Do I need an i-rate or a k-rate opcode?”

²As for a sample rate of 44100 Hz (`sr=44100`) and a control period of 32 samples (`ksmps=32`), we have 1378 control periods in one second. So 0.01 seconds will perform 14 control cycles.

i-rate Versions: ***schedule*, *event_i*, *scoreline_i***

If you need an i-rate opcode to trigger an instrument event, ***schedule*** is the most basic choice. You use it actually exactly the same as writing any score event; just separating the parameter fields by commas rather than spaces:

```
schedule iInstrNum (or "InstrName"), iStart, iDur [, ip4] [, ip5] [...]
```

event_i is very similar:

```
event_i "i", iInstrNum (or "InstrName"), iStart, iDur [, ip4] [, ip5] [...]
```

There are two differences between ***schedule*** and ***event_i***. The first is that ***schedule*** can only trigger instruments, whereas ***event_i*** can also trigger ***f*** events (= build function tables).

The second difference is that ***schedule*** can pass strings to the called instrument, but ***event_i*** (and ***event***) can not. So, if you execute this code ...

```
schedule "bla", 0, 1, "blu"
```

... it is allright; but with the same line for ***event_i*** ...

```
event_i "i", "bla", 0, 1, "blu"
```

... you will get this error message in the console:

```
error: Unable to find opcode entry for 'event_i' with matching argument types:  
Found: (null) event_i SccS
```

With ***scoreline_i*** sending strings is also possible. This opcode takes one or more lines of score statements which follow the same conventions as if written in the score section itself.³ If you enclose the line(s) by {{ and }}, you can include as many strings in it as you wish:

```
scoreline_i {{  
    i "bla" 0 1 "blu" "sound"  
    i "bla" 1 1 "brown" "earth"  
}}
```

k-rate versions: ***schedulek*, *event*, *scoreline*, *schedkwhen***

If you need a k-rate opcode to trigger an instrument event, ***schedulek*** is the basic choice as k-variant of ***schedule***:

```
schedulek kInstrNum (or "InstrName"), kStart, kDur [, kp4] [, kp5] [...]
```

The advantage of ***schedulek*** against ***event*** is the possibility to pass strings as p-fields. On the other hand, ***event*** can not only generate instrument events, but also other score events. For instrument events, the syntax is:

```
event "i", kInstrNum (or "InstrName"), kStart, kDur [, kp4] [, kp5] [...]
```

³This means that score parameter fields are separated by spaces, not by commas.

Usually, you will not want to trigger another instrument each control cycle, but based on certain conditions. A very common case is a “ticking” periodic signal, whichs ticks are being used as trigger impulses. The typical code snippet using a metro and the event opcode would be:

```
kTrigger metro 1 ;"ticks" once a second
if KTrigger == 1 then ;if it ticks
  schedulek "my_instr", 0, 1 ;call the instrument
endif
```

In other words: This code would only use one control-cycle per second to call my_instr, and would do nothing in the other control cycles. The `schedkwhen` opcode simplifies such typical use cases, and adds some other useful arguments. This is the syntax:

```
schedkwhen kTrigger, kMinTim, kMaxNum, kInsrNum (or "InstrName"),
kStart, kDur [, kp4] [, kp5] [...]
```

The `kMinTim` parameter specifies the time which has to be spent between two subsequent calls of the subinstrument. This is often quite useful as you may want to state: “Do not call the next instance of the subinstrument unless 0.1 seconds have been passed.” If you set this parameter to zero, there will be no time limit for calling the subinstrument.

The `kMaxNum` parameter specifies the maximum number of instances which run simultaneously. Say, `kMaxNum = 2` and there are indeed two instances of the subinstrument running, no other instance will be initiated. if you set this parameter to zero, there will be no limit for calling new instances.

So, with `schedkwhen`, we can write the above code snippet in two lines instead of four:

```
kTrigger metro 1 ;"ticks" once a second
schedkwhen kTrigger, 0, 0, "my_instr", 0, 1
```

Only, you cannot pass strings as p-fields via `schedkwhen` (and event). So, very much similar as described above for i-rate opcodes, `scoreline` fills this gap (as well as `schedulek`). Usually we will use it with a condition, as we did for the event opcode:

```
kTrigger metro 1 ;"ticks" once a second
if KTrigger == 1 then
  ;if it ticks, call two instruments and pass strings as p-fields
  scoreline {{
    i "bla" 0 1 "blu" "sound"
    i "bla" 1 1 "brown" "earth"
  }}
endif
```

Recompilation

As it has been mentioned at the start of this chapter, since Csound6 you can re-compile any code in an already running Csound instance. Let us first see some simple examples for the general use, and then a more practical approach in CsoundQt.

compileorc / compilestr

The opcode `compileorc` refers to a definition of instruments which has been saved as an `.orc` ("orchestra") file. To see how it works, save this text in a simple text (ASCII) format as "to_recompile.orc":

```
instr 1
iAmp = .2
iFreq = 465
aSig oscils iAmp, iFreq, 0
outs aSig, aSig
endin
```

Then save this csd in the same directory:

EXAMPLE 03F13_compileorc.csd

```
<CsoundSynthesizer>
<CsOptions>
-o dac -d -L stdin -Ma
</CsOptions>
<CsInstruments>
sr = 44100
nchnls = 2
ksmps = 32
0dbfs = 1

massign 0, 9999

instr 9999
ires compileorc "to_recompile.orc"
print ires ; 0 if compiled successfully
event_i "i", 1, 0, 3 ;send event
endin

</CsInstruments>
<CsScore>
i 9999 0 1
</CsScore>
</CsoundSynthesizer>
;example by joachim heintz
```

If you run this csd in the terminal, you should hear a three seconds beep, and the output should be like this:

```
SECTION 1:
new alloc for instr 9999:
instr 9999: ires = 0.000
new alloc for instr 1:
B 0.000 .. 1.000 T 1.000 TT 1.000 M: 0.20000 0.20000
B 1.000 .. 3.000 T 3.000 TT 3.000 M: 0.20000 0.20000
Score finished in csoundPerform().
inactive allocs returned to freespace
end of score.          overall amps: 0.20000 0.20000
                      overall samples out of range:      0      0
0 errors in performance
```

Having understood this, it is easy to do the next step. Remove (or comment out) the score line `i`

9999 0 1 so that the score is empty. If you start the csd now, Csound will run indefinitely. Now call instr 9999 by typing *i* 9999 0 1 in the terminal window (if the option -L stdin works for your setup), or by pressing any MIDI key (if you have connected a keyboard). You should hear the same beep as before. But as the recompile.csd keeps running, you can change now the instrument 1 in file *to_recompile.orc*. Try, for instance, another value for kFreq. Whenever this is done (file is saved) and you call again *instr* 9999 in *recompile.csd*, the new version of this instrument is compiled and then called immediately.

The other possibility to recompile code by using an opcode is *compilestr*. It will compile any instrument definition which is contained in a string. As this will be a string with several lines, you will usually use the {{ delimiter for the start and }} for the end of the string. This is a basic example:

EXAMPLE 03F14_compilestr.csd

```
<CsoundSynthesizer>
<CsOptions>
-o dac -d
</CsOptions>
<CsInstruments>
sr = 44100
nchnls = 1
ksmps = 32
0dbfs = 1

instr 1

;will fail because of wrong code
ires compilestr {{ 
instr 2
a1 oscilb p4, p5, 0
out a1
endin
}}
print ires ; returns -1 because not successfull

;will compile ...
ires compilestr {{ 
instr 2
a1 oscils p4, p5, 0
out a1
endin
}}
print ires ; ... and returns 0

;call the new instrument
;(note that the overall performance is extended)
scoreline_i "i 2 0 3 .2 415"

endin

</CsInstruments>
<CsScore>
i1 0 1
</CsScore>
</CsoundSynthesizer>
;example by joachim heintz
```

Instrument 2 is defined inside instrument 1, and compiled via *compilestr*. In case you can change this string in real-time (for instance in receiving it via OSC), you can add any new definition of

instruments on the fly.

The frontends offer simplified methods for recompilation. In CsoundQt, for instance, you can select any instrument, and choose *Edit > Evaluate Selection*.

03 G. 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 *UDOs*. A *UDO* behaves in the same way as a standard opcode: it has input arguments, and usually one or more output variables. It runs 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*. Finally some examples are shown for different User Defined Opcode definitions and applications.

If you want to write a User Defined Opcode in Csound6 which uses arrays, have a look at the end of chapter [03E](#) to see their usage and naming conventions.

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 us start with the

instrument code:

EXAMPLE 03G01_Pre_UDO.csd

```

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

```

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 03G02_All_to_UDO.csd

```

<CsoundSynthesizer>
<CsOptions>
-odac
</CsOptions>
<CsInstruments>
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
out aOut, aOut
        endop

instr 1
          FiltFb
endin

</CsInstruments>
<CsScore>
i 1 0 60
</CsScore>
</CsoundSynthesizer>
;example by joachim heintz

```

Before we continue the discussion about the quality of this transformation, 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 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 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:

```
out      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 `out` 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 03G03_UDO_more_flex.csd

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

```

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 definition 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 03G04_UDO_calls_UDO.csd

```

<CsoundSynthesizer>
<CsOptions>
-odac
</CsOptions>
<CsInstruments>
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      =
iFb     =
aDelTm randomi .1, .8; 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    =
            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      =
iFb     =
aDelTm randomi .1, .8; 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>
;example by joachim heintz

```

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*).

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 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 03G06_UDO_setksmps.csd

```

<CsoundSynthesizer>
<CsOptions>
-nm0
</CsOptions>
<CsInstruments>
sr = 44100
ksmps = 44100 ;very high because of printing

    opcode Faster, 0, 0
setksmps 4410 ;local ksmmps is 1/10 of global ksmmps
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 ksmmps
    endin

</CsInstruments>
<CsScore>
i 1 0 2
</CsScore>
</CsoundSynthesizer>
;example by joachim heintz

```

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

For k-time arguments, you can use since Csound 5.18 these default values:

- O (instead of k) defaults to 0
- P (instead of k) defaults to 1
- V (instead of k) defaults to 0.5
- J (instead of k) 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 03G07_UDO_default_args.csd

```

<CsoundSynthesizer>
<CsOptions>
-nm0
</CsOptions>
<CsInstruments>

    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>
;example by joachim heintz

```

Overloading

Extending this example a bit shows an important feature of *UDOs*. If we have different input and/or output types, we can use the **same** name for the *UDO*. Csound will choose the appropriate version depending on the context. This is a well-known practice in many programming languages as *overloading a function*.

In the simple example below, the *i*-rate and the *k*-rate version of the UDO are both called *Default*. Depending on the variable type and the number of outputs, the correct version is used by Csound.

EXAMPLE 03G08_UDO_overloading.csd

```

<CsoundSynthesizer>
<CsOptions>
-nm0
</CsOptions>
<CsInstruments>
ksmps  = 32

opcode Defaults, iii, opj
ia, ib, ic xin
xout ia, ib, ic
endop

opcode Defaults, kkkk, OPVJ
k1, k2, k3, k4 xin
xout k1, k2, k3, k4
endop

instr 1
ia, ib, ic Defaults
    prints "ia = %d, ib = %d, ic = %d\n", ia, ib, ic
ia, ib, ic Defaults 10
    prints "ia = %d, ib = %d, ic = %d\n", ia, ib, ic
ia, ib, ic Defaults 10, 100
    prints "ia = %d, ib = %d, ic = %d\n", ia, ib, ic

```

```

ia, ib, ic Defaults 10, 100, 1000
    prints "ia = %d, ib = %d, ic = %d\n", ia, ib, ic
ka1, kb1, kc1, kd1 Defaults
printks "ka = %d, kb = %d, kc = %.1f, kd = %d\n", 0, ka1, kb1, kc1, kd1
ka2, kb2, kc2, kd2 Defaults 2
printks "ka = %d, kb = %d, kc = %.1f, kd = %d\n", 0, ka2, kb2, kc2, kd2
ka3, kb3, kc3, kd3 Defaults 2, 4
printks "ka = %d, kb = %d, kc = %.1f, kd = %d\n", 0, ka3, kb3, kc3, kd3
ka4, kb4, kc4, kd4 Defaults 2, 4, 6
printks "ka = %d, kb = %d, kc = %.1f, kd = %d\n", 0, ka4, kb4, kc4, kd4
ka5, kb5, kc5, kd5 Defaults 2, 4, 6, 8
printks "ka = %d, kb = %d, kc = %.1f, kd = %d\n", 0, ka5, kb5, kc5, kd5
    turnoff
endin

</CsInstruments>
<CsScore>
i 1 0 1
</CsScore>
</CsoundSynthesizer>
;example by joachim heintz

```

Prints:

```

ia = 0, ib = 1, ic = -1
ia = 10, ib = 1, ic = -1
ia = 10, ib = 100, ic = -1
ia = 10, ib = 100, ic = 1000
ka = 0, kb = 1, kc = 0.5, kd = -1
ka = 2, kb = 1, kc = 0.5, kd = -1
ka = 2, kb = 4, kc = 0.5, kd = -1
ka = 2, kb = 4, kc = 6.0, kd = -1
ka = 2, kb = 4, kc = 6.0, kd = 8

```

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 similarly simple way. This is an example of generating seven partials by a recursive UDO. See the last example in the next section for a more musical application of a recursive UDO.

As seen in this example, instruments can do recursion, too. Here we set a number of repetitions in p4. As long as this number has not reached 1, the next instance is called after this instance has finished.

EXAMPLE 03G09_Recursive_UDO.csd

```

<CsoundSynthesizer>
<CsOptions>
-o dac -m 128
</CsOptions>
<CsInstruments>
sr = 44100
ksmps = 32
nchnls = 2

```

```

0dbfs = 1
seed 0

opcode Recursion, a, iip
    //input: frequency, number of partials, this partial (default=1)
    iFreq, iNumParts, iThisPart xin
    //amplitude decreases for higher partials
    iAmp = 1/iNumParts/iThisPart
    //apply small frequency deviation except for the first partial
    iFreqMult = (iThisPart == 1) ? 1 : iThisPart*randomi(0.95,1.05)
    //create this partial
    aOut = poscil:a(iAmp,iFreq*iFreqMult)
    //add the other partials via recursion
    if (iThisPart < iNumParts) then
        aOut += Recursion(iFreq,iNumParts,iThisPart+1)
    endif
    xout(aOut)
endop

instr I_can_do_the_same
    //call a sound with 7 partials based on 400 Hz
    aPartials = Recursion(400,7)
    aOut = linen:a(aPartials,0.01,p3,0.5)
    outall(aOut)
    //call this instrument again for p4 times
    if (p4 > 1) then
        schedule(p1,p3,p3,p4-1)
    endif
endin

</CsInstruments>
<CsScore>
i "I_can_do_the_same" 0 3 3
e 9
</CsScore>
</CsoundSynthesizer>
;example by joachim heintz

```

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.

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 Csound's `diskin` opcode. 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 implemented here by overloading. FilePlay either returns one audio signal (if the file is stereo it uses just the first channel), or it returns 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 a *tolerant diskin* ...

EXAMPLE 03G10_UDO_FilePlay.csd

```
<CsoundSynthesizer>
<CsOptions>
-odac --env:SSDIR+=../SourceMaterials
</CsOptions>
<CsInstruments>
sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

opcode FilePlay, a, SKoo
;gives mono output (if file is stereo, just the first channel is used)
Sfil, kspeed, iskip, iloop xin
  if filenchnls(Sfil) == 1 then
    aout    diskin  Sfil, kspeed, iskip, iloop
  else
    aout, a0  diskin  Sfil, kspeed, iskip, iloop
  endif
    xout      aout
endop

opcode FilePlay, aa, SKoo
;gives stereo output (if file is mono, the channel is duplicated)
Sfil, kspeed, iskip, iloop xin
ichn    filenchnls Sfil
  if filenchnls(Sfil) == 1 then
    aL      diskin  Sfil, kspeed, iskip, iloop
    aR      =        aL
  else
    aL, aR   diskin  Sfil, kspeed, iskip, iloop
  endif
    xout      aL, aR
endop

instr 1
aMono   FilePlay  "fox.wav", 1
          outs     aMono, aMono
  endin

instr 2
aL, aR   FilePlay  "fox.wav", 1
          outs     aL, aR
  endin

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

```
</CsoundSynthesizer>
;example by joachim heintz
```

Change the Content of a Function Table

In example *03C11_Table_random_dev.cs*, 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.

EXAMPLE 03G11_UDO_rand_dev.cs

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

giSine ftgen 0, 0, 256, 10, 1; sine wave

opcode TabDirtk, 0, ikk
;"dirty" 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
  while kndx < ftlen(ift) do
    kval table kndx, ift; read old value
    knewval = kval + rnd31:k(kperc/100,0); calculate new value
    tablew knewval, kndx, giSine; write new value
    kndx += 1
  od
endif
endop

instr 1
kTrig metro 1 ;trigger signal once per second
kPerc linseg 0, p3, 100
TabDirtk giSine, kTrig, kPerc
aSig oscil .2, 400, giSine
out aSig, aSig
  endin

</CsInstruments>
<CsScore>
i 1 0 10
</CsScore>
</CsoundSynthesizer>
;example by joachim heintz
```

The next example permutes a series of numbers randomly each time it is called. For this purpose, one random element of the input array¹ is taken and written to the first position of the output array. Then all elements which are “right of” this one random element are copied one position to the left. As result the previously chosen element is being overwritten, and the number of values to read is shranked by one. This process is done again and again, until each *old* element has placed to a (potentially) *new* position in the resulting output array.

EXAMPLE 03G12_ArrPermRnd.csd

```

<CsoundSynthesizer>
<CsOptions>
-nm0
</CsOptions>
<CsInstruments>
ksmps = 32
seed 0

opcode ArrPermRnd, i[], i[]
iInArr[]      xin
iLen          =      lenarray(iInArr)
;create output array and set index
iOutArr[]     init    iLen
iWriteIdx     =      0
iReadLen      =      iLen
;for all elements:
while iWriteIdx < iLen do
;get one random element and put it in iOutArr
iRndIdx      =      int(random:0, iReadLen-.0001))
iOutArr[iWriteIdx] = iInArr[iRndIdx]
;shift the elements after this one to the left
while iRndIdx < iReadLen-1 do
  iInArr[iRndIdx] = iInArr[iRndIdx+1]
  iRndIdx += 1
od
;decrease length to read in and increase write index
iReadLen -= 1
iWriteIdx += 1
od
          xout    iOutArr
endop

;create i-array as 1, 2, 3, ... 12
giArr[] genarray 1, 12

;permutation of giArr ...
instr Permut
iPermut[] ArrPermRnd giArr
printarray iPermut, "%d"
endin

;... which has not been touched by these operations
instr Print
printarray giArr, "%d"
endin

</CsInstruments>
<CsScore>
i "Permut" 0 .01

```

¹More precisely the random element is taken from a copy of the input array. This copy is always created by the UDO, so the original array is left untouched. This is visible in the last line of the printout.

```
i "Permut" + .
i "Permut" + .
i "Permut" + .
i "Permut" + .
i "Print" .05 .01
</CsScore>
</CsoundSynthesizer>
;example by joachim heintz
```

Prints (for example):

```
8 2 1 7 4 11 5 12 10 6 9 3
7 5 3 2 11 12 9 8 1 10 6 4
7 9 6 2 5 3 12 8 10 1 11 4
1 12 10 11 9 5 4 8 6 7 2 3
7 12 8 2 10 4 5 1 11 3 6 9
1 2 3 4 5 6 7 8 9 10 11 12
```

A Recursive User Defined Opcode for Additive Synthesis

Smilar to example 03F11 we synthesize here a number of partials, and add a random frequency deviation to it. We use a maximum possible deviation of plus/minus three semitones compared to the harmonic frequencies. Also we apply a unique duration for each partial. This is written as a recursive UDO. Each UDO generates one partial, and calls the UDO again until the last partial is generated. The *Main* instrument itself performs the time loop as recursion, calculating the basic values for one note and performing the event. Here the recursive call is limited by time: After 60 seconds no more calls to the *Main* instrument, but instead to another instrument which exits Csound by a score "e" statement.

EXAMPLE 03G13_UDO_Recursive_AddSynth.csd

```
<CsoundSynthesizer>
<CsOptions>
-odac -m128
</CsOptions>
<CsInstruments>
sr = 44100
ksmps = 128
nchnls = 2
0dbfs = 1
seed 0

opcode Partials, aa, iiip
//plays iNumParts partials with frequency deviation and own envelopes and
//own durations for each partial
//iBasFreq: base frequency of sound mixture
//iNumParts: total number of partials
//iPan: panning
//iThisPart: which partial is this (1-N, default=1)
iBasFreq, iNumParts, iPan, iThisPart xin
//this partial's frequency
iFreq = iBasFreq * iThisPart * semitone(random:i(-3,3))
//additional time for this partial
iDur = random:i(p3/3,p3*3)
//volume with random deviation of 6 dB maximum (and the higher the softer)
```

```

iAmp = (1/iNumParts) * ampdb(random:i(-6,0)-iThisPart)
//partial envelope and tone
aEnv = transeg:a(0,0.005,0,iAmp,iDur-0.005,-10,0)
aTone = oscil:a(aEnv,iFreq)
//panning with slight deviations around the iPan input
aL,aR pan2 aTone, iPan+random:i(-0.1,0.1)
//recursion
if (iThisPart < iNumParts) then
    aL2,aR2 Partials iBasFreq,iNumParts,iPan,iThisPart+1
    aL += aL2
    aR += aR2
endif
p3 = iDur
xout(aL,aR)
endop

instr Walk
p3 = random:i(1,5)
iNumPartials = int(random:i(8,14))
iBasFreq = mtof:i(random:i(24,84))
iPan = random:i(0.2,0.8)
aL,aR Partials iBasFreq, iNumPartials, iPan
outs(aL,aR)
//play for 60 seconds
if (times:i() < 60) then
    schedule(p1,random:i(1,4),1)
//then exit gently (= after 10 seconds)
else
    schedule("Turnoff",10,1)
endif
endin
schedule("Walk",0,1)

instr Turnoff
event_i("e",0)
endin

</CsInstruments>
<CsScore>
</CsScore>
</CsoundSynthesizer>
;Example by joachim heintz

```

Filter implementation via Sample-by-Sample Processing

At the end of chapter 03A the ability of sample-by-sample processing has been shown at some basic examples. This feature is really substantial for writing digital filters. This can perfectly be done in the Csound language itself. The next example shows an implementation of the zero delay state variable filter by Steven Yi. In his collection at www.github.com/kunstmusik/libsyi more details and other implementations can be found. — Note also that this code is another example of overloading a UDO definition. The same opcode name is defined here twice; first with the input types aKK (one audio signal and two k-signals with initialization), then with the input types aaa.

EXAMPLE 03G14_UDO_zdf_svf.csd

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

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

opcode zdf_svf,aaa,aKK

ain, kcf, kR      xin

; pre-warp the cutoff- these are bilinear-transform filters
kwd = 2 * $M_PI * kcf
iT = 1/sr
kwa = (2/iT) * tan(kwd * iT/2)
kG = kwa * iT/2

;; output signals
alp init 0
ahp init 0
abp init 0

;; state for integrators
kz1 init 0
kz2 init 0

;;
kindx = 0
while kindx < ksmpls do
    khp = (ain[kindx] - (2*kR+kG) * kz1 - kz2) / (1 + (2*kR*kG) + (kG*kG))
    kbp = kG * khp + kz1
    klp = kG * kbp + kz2

    ; z1 register update
    kz1 = kG * khp + kbp
    kz2 = kG * kbp + klp

    alp[kindx] = klp
    ahp[kindx] = khp
    abp[kindx] = kbp
    kindx += 1
od

xout alp, abp, ahp

endop

opcode zdf_svf,aaa,aaa

ain, acf, aR      xin

iT = 1/sr

;; output signals
alp init 0
ahp init 0
abp init 0
```

```

;; state for integrators
kz1 init 0
kz2 init 0

;;
kindx = 0
while kindx < ksmps do

    ; pre-warp the cutoff- these are bilinear-transform filters
    kwd = 2 * $M_PI * acf[kindx]
    kwa = (2/iT) * tan(kwd * iT/2)
    kG = kwa * iT/2

    kR = aR[kindx]

    khp = (ain[kindx] - (2*kR+kG) * kz1 - kz2) / (1 + (2*kR*kG) + (kG*kG))
    kbp = kG * khp + kz1
    klp = kG * kbp + kz2

    ; z1 register update
    kz1 = kG * khp + kbp
    kz2 = kG * kbp + klp

    alp[kindx] = klp
    ahp[kindx] = khp
    abp[kindx] = kbp
    kindx += 1
od

xout alp, abp, ahp

endop

giSine ftgen 0, 0, 2^14, 10, 1

instr 1

aBuzz buzz 1, 100, 50, giSine
aLp, aBp, aHp zdf_svf aBuzz, 1000, 1

out aHp, aHp

endin

</CsInstruments>
<CsScore>
i 1 0 10
</CsScore>
</CsoundSynthesizer>
;example by steven yi

```


03 H. MACROS

Macros within Csound provide a mechanism whereby a line or a block of code can be referenced using a macro codeword. Whenever the user-defined macro codeword for that block of code 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 extremely 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 - which can be thought of as the main macro containing sub-macros that can be repeated multiple 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. Macro expansion is a feature of the orchestra and score preprocessor and is not part of the compilation itself.

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. A limited range of special characters can be employed in the name. Apostrophes, hash symbols and dollar signs should be avoided. *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. A macro can be defined anywhere within the *<CsInstruments> ... </CsInstruments>* sections of a .csd file. A macro can be redefined or overwritten by reusing the same macro name in another macro definition. Subsequent expansions of the macro will then use the new version.

To 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.

EXAMPLE 03H01_Macros_basic.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 arguments are defined as a list of sub-macro names within brackets after the name of the primary macro with each macro argument being separated using 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 03H02_Macro_6partials.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 fundamental 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 that 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 03H03_Score_macro.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 with equal justification but each offers its own strengths. UDOs strengths lies in their ability to be used just like an opcode with inputs and outputs, 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](#).

03 I. FUNCTIONAL SYNTAX

Functional syntax is very common in many programming languages. It takes the form of `fun()`, where `fun` is any function which encloses its arguments in parentheses. Even in “old” Csound, there existed some rudiments of this functional syntax in some mathematical functions, such as `sqrt()`, `log()`, `int()`, `frac()`. For instance, the following code

```
iNum = 1.234
print int(iNum)
print frac(iNum)
```

would print:

```
instr 1: #i0 = 1.000
instr 1: #i1 = 0.230
```

Here the integer part and the fractional part of the number `1.234` are passed directly as an argument to the `print` opcode, without needing to be stored at any point as a variable.

This alternative way of formulating code can now be used with many opcodes in Csound6.¹ First we shall look at some examples.

The traditional way of applying a fade and a sliding pitch (glissando) to a tone is something like this:

EXAMPLE 03I01_traditional_syntax.csd

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

instr 1
kFade    linseg  0, p3/2, 0.2, p3/2, 0
kSlide   expseg  400, p3/2, 800, p3/2, 600
aTone    oscil    kFade, kSlide
          out      aTone
endin
```

¹The main restriction is that it can only be used by opcodes which have only one output (not two or more).

```
</CsInstruments>
<CsScore>
i 1 0 5
</CsScore>
</CsoundSynthesizer>
;example by joachim heintz
```

In plain English what is happening is:

1. We create a line signal with the opcode *linseg*. It starts at zero, moves to 0.2 in half of the instrument's duration ($p3/2$), and moves back to zero for the second half of the instrument's duration. We store this signal in the variable *kFade*.
2. We create an exponential signal with the opcode *expseg*. It starts at 400, moves to 800 in half the instrument's duration, and moves to 600 for the second half of the instrument's duration. We store this signal in the variable *kSlide*.
3. We create a sine audio signal with the opcode *poscil*. We feed in the signal stored in the variable *kFade* as amplitude, and the signal stored in the variable *kSlide* as frequency input. We store the audio signal in the variable *aTone*.
4. Finally, we write the audio signal to the output with the opcode *out*.

Each of these four lines can be considered as a “function call”, as we call the opcodes (functions) *linseg*, *expseg*, *poscil* and *out* with certain arguments (input parameters). If we now transform this example to functional syntax, we will avoid storing the result of a function call in a variable. Rather we will feed the function and its arguments directly into the appropriate slot, by means of the *fun()* syntax.

If we write the first line in functional syntax, it will look like this:

```
linseg(0, p3/2, 0.2, p3/2, 0)
```

And the second line will look like this:

```
expseg(400, p3/2, 800, p3/2, 600)
```

So we can reduce our code from four lines to two lines:

EXAMPLE 03I02_functional_syntax_1.csd

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

instr 1
aTone poscil linseg(0,p3/2,.2,p3/2,0), expseg(400,p3/2,800,p3/2,600)
out aTone
endin

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

```
</CsoundSynthesizer>
;example by joachim heintz
```

Or if you prefer the “all-in-one” solution:²

EXAMPLE 03I03_functional_syntax_2.csd

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

instr 1
out(poscil(linseg(0,p3/2,.2,p3/2,0),expseg(400,p3/2,800,p3/2,600)))
endin

</CsInstruments>
<CsScore>
i 1 0 5
</CsScore>
</CsoundSynthesizer>
;example by joachim heintz
```

Declare your color: i, k or a?

Most of the Csound opcodes work not only at one rate. You can, for instance, produce random numbers at i-, k- or a-rate:³

```
iros      random    imin, imax
kros      random    kmin, kmax
ares      random    kmin, kmax
```

Let us assume we want to change the highest frequency in our example from 800 to a random value between 700 and 1400 Hz, so that we hear a different movement for each tone. In this case, we can simply write *random(700, 1400)*:

EXAMPLE 03I04_functional_syntax_rate_1.csd

```
<CsoundSynthesizer>
<CsOptions>
-odac
</CsOptions>
<CsInstruments>
sr = 44100
nchnls = 1
```

²Please note that these two examples are not really correct, because the rates of the opcodes are not specified.

³See chapter 03A Initialization and Performance Pass for a more thorough explanation.

```

ksmps = 32
0dbfs = 1

instr 1
    out(poscil(linseg(0,p3/2,.2,p3/2,0),
                expseg(400,p3/2,random(700,1400),p3/2,600)))
endin

</CsInstruments>
<CsScore>
r 5
i 1 0 3
</CsScore>
</CsoundSynthesizer>
;example by joachim heintz

```

But why is the *random* opcode here performing at i-rate, and not at k- or a-rate? This is, so to say, pure random – it happens because in the Csound soruces the i-rate variant of this opcode is written first.⁴ If the k-rate variant were first, the above code failed.

So it is both, clearer and actually required, to explicitly declare at which rate a function is to be performed. This code claims that *poscil* runs at a-rate, *linseg* and *expseg* run at k-rate, and *random* runs at i-rate here:

EXAMPLE 03I05_functional_syntax_rate_2.csd

```

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

instr 1
out(poscil:a(linseg:k(0, p3/2, 1, p3/2, 0),
              expseg:k(400, p3/2, random:i(700, 1400), p3/2, 600)))
endin

</CsInstruments>
<CsScore>
r 5
i 1 0 3
</CsScore>
</CsoundSynthesizer>
;example by joachim heintz

```

Rate declaration is done with simply specifying :a, :k or :i after the function. It would represent good practice to include it all the time, to be clear about what is happening.

⁴See <https://github.com/csound/csound/blob/develop/Opcodes/uggab.c>, line 2085

fun() with UDOs

It should be mentioned that you can use the functional style also with self created opcodes ("User Defined Opcodes"):

EXAMPLE 03I06_functional_syntax_udo.csd

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

opcode FourModes, a, akk[]
    ;kFQ[] contains four frequency-quality pairs
    aIn, kBasFreq, kFQ[] xin
aOut1 mode aIn, kBasFreq*kFQ[0], kFQ[1]
aOut2 mode aIn, kBasFreq*kFQ[2], kFQ[3]
aOut3 mode aIn, kBasFreq*kFQ[4], kFQ[5]
aOut4 mode aIn, kBasFreq*kFQ[6], kFQ[7]
    xout (aOut1+aOut2+aOut3+aOut4) / 4
endop

instr 1
kArr[] fillarray 1, 2000, 2.8, 2000, 5.2, 2000, 8.2, 2000
aImp mpulse .3, 1
        out FourModes(aImp, 200, kArr)
endin

</CsInstruments>
<CsScore>
i 1 0 10
</CsScore>
</CsoundSynthesizer>
;example by joachim heintz, based on an example of iain mccurdy
```

Besides the ability of functional expressions to abbreviate code, this way of writing Csound code allows to coincide with a convention which is shared by many programming languages. This final example is doing exactly the same as the previous one, but for some programmers in a more clear and common way:

EXAMPLE 03I07_functional_syntax_udo_2.csd

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

```
0dbfs = 1

opcode FourModes, a, akk[]
aIn, kBasFreq, kFQ[] xin
aOut1 = mode:a(aIn,kBasFreq*kFQ[0],kFQ[1])
aOut2 = mode:a(aIn,kBasFreq*kFQ[2],kFQ[3])
aOut3 = mode:a(aIn,kBasFreq*kFQ[4],kFQ[5])
aOut4 = mode:a(aIn,kBasFreq*kFQ[6],kFQ[7])
xout (aOut1+aOut2+aOut3+aOut4) / 4
endop

instr 1
kArr[] = fillarray(1, 2000, 2.8, 2000, 5.2, 2000, 8.2, 2000)
aImp   = mpulse:a(.3, 1)
aOut   = FourModes(aImp, randomh:k(200,195,1), kArr)
out(aOut, aOut)
endin

</CsInstruments>
<CsScore>
i 1 0 10
</CsScore>
</CsoundSynthesizer>
;example by joachim heintz, based on an example of iain mccurdy
```

04 A. ADDITIVE SYNTHESIS

Jean Baptiste Joseph Fourier (1768-1830) claimed in this treatise *Théorie analytique de la chaleur* (1822) that any periodic function can be described perfectly as a sum of weighted sine waves. The frequencies of these *harmonics* are integer multiples of the fundamental frequency.

As we can easily produce sine waves of different amplitudes in digital sound synthesis, the *Fourier Synthesis* or *Additive Synthesis* may sound the universal key for creating interesting sounds. But first, not all sounds are periodic. *Noise* is very important part of the sounding world represents the other pole which is essentially non-periodic. And dealing with single sine waves means dealing with a lot of data and requirements.

Nonetheless, additive synthesis can provide unusual and interesting sounds and the power of modern computers and their ability to manage data in a programming language offers new dimensions of working with this old technique. As with most things in Csound there are several ways to go about implementing additive synthesis. We shall endeavour to introduce some of them and to allude to how they relate to different programming paradigms.

Main Parameters of Additive Synthesis

Before examining various methods of implementing additive synthesis in Csound, we shall first consider what parameters might be required. As additive synthesis involves the addition of multiple sine generators, the parameters we use will operate on one of two different levels:

- **For each sine**, there will be a frequency and an amplitude with an envelope.
 - The **frequency** will usually be a constant value, but it can be varied and in fact natural sounds typically exhibit slight modulations of partial frequencies.
 - The **amplitude** must have at least a simple envelope such as the well-known ADSR but more complex methods of continuously altering the amplitude will result in a livelier sound.
- **For the sound as an entirety**, the relevant parameters are:
 - The total **number of sinusoids**. A sound which consists of just three sinusoids will most likely sound poorer than one which employs 100.
 - The **frequency ratios** of the sine generators. For a classic harmonic spectrum, the multipliers of the sinusoids are 1, 2, 3, ... (If your first sine is 100 Hz, the others will be 200,

300, 400, ... Hz.) An inharmonic or noisy spectrum will probably have no simple integer ratios. These frequency ratios are chiefly responsible for our perception of timbre.

- The **base frequency** is the frequency of the first partial. If the partials are exhibiting a 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 in determining the resulting timbre of a sound. If the higher partials are relatively strong, the sound will be perceived as being more “brilliant”; if the higher partials are soft, then the sound will be perceived as being dark and soft.
- The **duration ratios** of the sinusoids. In simple additive synthesis, all single sines have the same duration, but it will be more interesting if they differ - this will usually relate to the durations of the envelopes: if the envelopes of different partials vary, some partials will die away faster than others.

It is not always the aim of additive synthesis to imitate natural sounds, but the task of first analysing and then attempting to imitate a sound can prove to be very useful when studying additive synthesis. This is what a guitar note looks like when spectrally analysed:

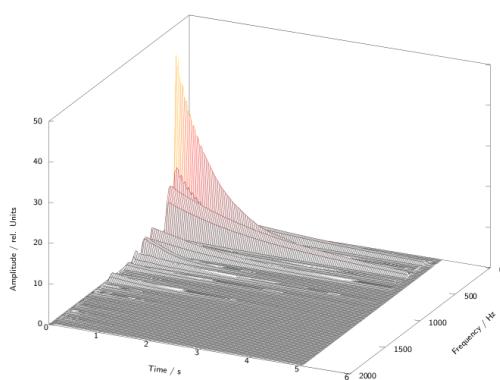


Figure 27.1: Spectral analysis of a guitar tone in time (courtesy of W. Fohl, Hamburg)

Each partial possesses its own frequency movement and duration. We may or may not be able to achieve this successfully using additive synthesis. We will begin with some simple sounds and consider how to go about programming this in Csound. Later we will look at some more complex sounds and the more advanced techniques required to synthesize them.

Different Methods for Additive Synthesis

Simple Additions of Sinusoids Inside an Instrument

If additive synthesis amounts to simply adding together sine generators, it is therefore straightforward to implement this by creating multiple oscillators in a single instrument and adding their

outputs together. In the following example, instrument 1 demonstrates the creation of 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_AddSynth_simple.csd

```

<CsoundSynthesizer>
<CsOptions>
-o dac
</CsOptions>
<CsInstruments>
sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 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
a0sc1    oscil    ibaseamp, ibasefrq
a0sc2    oscil    ibaseamp/2, ibasefrq*2
a0sc3    oscil    ibaseamp/3, ibasefrq*3
a0sc4    oscil    ibaseamp/4, ibasefrq*4
a0sc5    oscil    ibaseamp/5, ibasefrq*5
a0sc6    oscil    ibaseamp/6, ibasefrq*6
a0sc7    oscil    ibaseamp/7, ibasefrq*7
a0sc8    oscil    ibaseamp/8, ibasefrq*8
;apply simple envelope
kenv    linen    1, p3/4, p3, p3/4
;add partials and write to output
aOut = a0sc1 + a0sc2 + a0sc3 + a0sc4 + a0sc5 + a0sc6 + a0sc7 + a0sc8
      outs    aOut*kenv, aOut*kenv
      endin

instr 2 ;inharmonic additive synthesis
ibasefrq = cpspch(p4)
ibaseamp = ampdBFS(p5)
;create 8 inharmonic partials
a0sc1    oscil    ibaseamp, ibasefrq
a0sc2    oscil    ibaseamp/2, ibasefrq*1.02
a0sc3    oscil    ibaseamp/3, ibasefrq*1.1
a0sc4    oscil    ibaseamp/4, ibasefrq*1.23
a0sc5    oscil    ibaseamp/5, ibasefrq*1.26
a0sc6    oscil    ibaseamp/6, ibasefrq*1.31
a0sc7    oscil    ibaseamp/7, ibasefrq*1.39
a0sc8    oscil    ibaseamp/8, ibasefrq*1.41
kenv    linen    1, p3/4, p3, p3/4
aOut = a0sc1 + a0sc2 + a0sc3 + a0sc4 + a0sc5 + a0sc6 + a0sc7 + a0sc8
      outs    aOut*kenv, aOut*kenv
      endin

</CsInstruments>
<CsScore>
;      pch      amp
i 1 0 5    8.00   -13
i 1 3 5    9.00   -17
i 1 5 8    9.02   -15
i 1 6 9    7.01   -15

```

```
i 1 7 10    6.00      -13
s
i 2 0 5    8.00      -13
i 2 3 5    9.00      -17
i 2 5 8    9.02      -15
i 2 6 9    7.01      -15
i 2 7 10   6.00      -13
</CsScore>
</CsoundSynthesizer>
;example by Andrés Cabrera
```

Simple Additions of Sinusoids via the Score

A typical paradigm in programming: if you are repeating lines of code with just minor variations, consider abstracting it in some way. In the Csound language this could mean moving parameter control to the score. In our case, the lines

```
a0sc1    oscil    ibaseamp, ibasefrq
a0sc2    oscil    ibaseamp/2, ibasefrq*2
a0sc3    oscil    ibaseamp/3, ibasefrq*3
a0sc4    oscil    ibaseamp/4, ibasefrq*4
a0sc5    oscil    ibaseamp/5, ibasefrq*5
a0sc6    oscil    ibaseamp/6, ibasefrq*6
a0sc7    oscil    ibaseamp/7, ibasefrq*7
a0sc8    oscil    ibaseamp/8, ibasefrq*8
```

could be abstracted to the form

```
a0sc    oscil    ibaseamp*iampfactor, ibasefrq*ifreqfactor
```

with the parameters *iampfactor* (the relative amplitude of a partial) and *ifreqfactor* (the frequency multiplier) being transferred to the score as *p-fields*.

The next version of the previous instrument, simplifies the instrument code and defines the variable values as score parameters:

EXAMPLE 04A02_AddSynth_score.csd

```
<CsoundSynthesizer>
<CsOptions>
-o dac
</CsOptions>
<CsInstruments>
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
a0sc      poscil  kEnv, iFreq, giSine
          outs    a0sc, a0sc
  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>
;example by Andrés Cabrera and Joachim Heintz

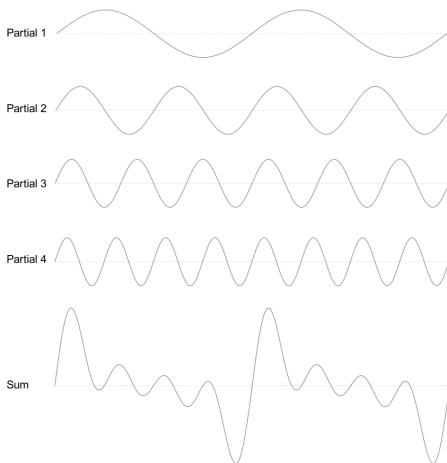
```

You might ask: "Okay, where is the simplification? There are even more lines than before!" This is true, but this still represents better coding practice. The main benefit now is *flexibility*. Now we are able to realise any number of partials using the same instrument, 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 make great use of copy-and-paste, and focus just on what is changing from line to line.

Note that you are now calling **one instrument multiple times** in the creation of a single additive synthesis note, in fact, each instance of the instrument contributes just one partial to the additive tone. Calling multiple instances of one instrument in this way also represents good practice in Csound coding. We will discuss later how this can be achieved in a more elegant way.

Creating Function Tables for Additive Synthesis

Before we continue, let us return to the first example and discuss a classic and abbreviated method for playing a number of partials. As we mentioned at the beginning, Fourier stated that any periodic oscillation can be described using a sum of simple sinusoids. If the single sinusoids are static (with no individual envelopes, durations or frequency fluctuations), the resulting waveform will be similarly static.



Above you see four sine waves, each with fixed frequency and amplitude relationships. These are then mixed together with the resulting waveform illustrated at the bottom (*Sum*). This then begs the question: why not simply calculate this composite waveform first, and then read it with just a single oscillator?

This is what some Csound GEN routines do. They compose the resulting shape of the periodic waveform, and store the values in a function table. [GEN10](#) can be used for creating a waveform consisting of harmonically related partials. Its form begins with the common GEN routine p-fields

```
<table number>, <creation time>, <size in points>, <GEN number>
```

following which you just have to define the relative strengths of the harmonics. [GEN09](#) is more complex and allows you to also control the frequency multiplier and the phase (0-360°) of each partial. Thus we are able to reproduce the first example in a shorter (and computationally faster) form:

EXAMPLE 04A03_AddSynth_GEN.csd

```
<CsoundSynthesizer>
<CsOptions>
-o dac
</CsOptions>
<CsInstruments>
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>
;example by Andrés Cabrera and Joachim Heintz

```

You maybe noticed that to store a waveform in which the partials are not harmonically related, the table must be constructed in a slightly special way (see table *giNois*). If the frequency multipliers in our first example started with 1 and 1.02, the resulting period is actually very long. If the oscillator was playing at 100 Hz, the tone it would produce would actually contain partials at 100 Hz and 102 Hz. So you need 100 cycles from the 1.00 multiplier and 102 cycles from the 1.02 multiplier to complete one period of the composite waveform. In other words, we have to create a table which contains respectively 100 and 102 periods, instead of 1 and 1.02. Therefore the table frequencies will not be related to 1 as usual but instead to 100. This is the reason that we have to introduce a new parameter, *iTabFreq*, for this purpose. (N.B. In this simple example we could actually reduce the ratios to 50 and 51 as 100 and 102 share a common denominator of 2.)

This method of composing waveforms can also be used for generating four standard waveform shapes typically encountered in vintage synthesizers. An **impulse** wave can be created by adding a number of harmonics of the same strength. A **sawtooth** wave has the amplitude multipliers 1, 1/2, 1/3, ... for the harmonics. A **square** wave 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 swapping positive and negative values. The next example creates function tables with just the first ten partials for each of these waveforms.

EXAMPLE 04A04_Standard_waveforms.csd

```

<CsoundSynthesizer>
<CsOptions>
-o dac
</CsOptions>
<CsInstruments>
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>
;example by joachim heintz

```

Triggering Instrument Events 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 (for example by using the `ftmorph` opcode) or by filtering the result. Next we shall consider another approach: triggering one instance of a sub-instrument¹ for each partial, and exploring the possibilities of creating a spectrally dynamic sound using this technique.

Let us return to the second instrument (04A02.csd) which had already made use of 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_Flexible_number_of_partials.csd

```

<CsoundSynthesizer>
<CsOptions>
-o dac
</CsOptions>
<CsInstruments>
sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 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 instances of the subinstrument
loop:
ifreq =         ibasfreq * ipart
iamp =          1/ipart/inumparts
event_i = "i", 10, 0, p3, ifreq, iamp

```

¹This term is used here in a general manner. There is also a Csound opcode `subinstr`, which has some more specific meanings.

```

        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-.1, -10, 0
apart    oscil   aenv, ifreq
          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>
;Example by joachim heintz

```

This instrument can easily be transformed to be played via a midi keyboard. In the next the midi key velocity will map to the number of synthesized partials to implement a brightness control.

EXAMPLE 04A06_Play_it_with_Midi.cs

```

<CsoundSynthesizer>
<CsOptions>
-o dac -Ma
</CsOptions>
<CsInstruments>
sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 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 inparts over the ipart variable
;and trigger inparts instances of the sub-instrument
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    oscil   aenv, ifreq
          outs    apart/3, apart/3
    endin

</CsInstruments>

```

```
<CsScore>
</CsScore>
</CsoundSynthesizer>
;Example by Joachim Heintz
```

Although this instrument is rather primitive it is useful to be able to control the timbre in this way using key velocity. Let us continue to explore some other methods of creating parameter variation in additive synthesis.

Applying User-controlled Random Variations

Natural sounds exhibit constant movement and change in the parameters we have so far discussed. Even the best player or singer will not be able to play a note in the exact same way twice and within a tone, the partials will have some unsteadiness: slight waverings in the amplitudes and slight frequency fluctuations. In an audio programming environment like Csound, we can imitate these movements by employing random deviations. The boundaries of random deviations must be adjusted as carefully. Exaggerate them and the result will be unnatural or like a bad player. The rates or speeds of these fluctuations will also need to be chosen carefully and sometimes we need to modulate the rate of modulation in order to achieve naturalness.

Let us start with some random deviations in our subinstrument. The following 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 relative to its default value. This alteration can be measured in decibels (dB).
- The **duration** of each partial can be made to be longer or shorter than the default value. Let us define this deviation as a percentage. If the expected duration is five seconds, a maximum deviation of 100% will mean a resultant value of between half the duration (2.5 sec) and double the duration (10 sec).

The following example demonstrates the effect of these variations. As a base - and as a reference to its author - we take as our starting point, the *bell-like* sound created by Jean-Claude Risset in his *Sound Catalogue*.²

EXAMPLE 04A07_Risset_variations.csd

```
<CsoundSynthesizer>
<CsOptions>
-o dac -m0
</CsOptions>
<CsInstruments>
sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1
```

²Jean-Claude Risset, Introductory Catalogue of Computer Synthesized Sounds (1969), cited after Dodge/Jerse, Computer Music, New York/London 1985, p.94

```

seed 0

;frequency and amplitude multipliers for 11 partials of Risset's bell
giFqs[] fillarray .56, .563, .92, .923, 1.19, 1.7, 2, 2.74, 3, 3.74, 4.07
giAmps[] fillarray 1, 2/3, 1, 1.8, 8/3, 5/3, 1.46, 4/3, 4/3, 1, 4/3
gSComments[] fillarray "unchanged sound", "slight variations in frequency",
    "slight variations in amplitude", "slight variations in duration",
    "slight variations combined", "heavy variations"
giCommentsIndx[] fillarray 0, 0, 1, 1, 1, 1, 2, 2, 2, 2, 3, 3, 3, 3,
    4, 4, 4, 4, 4, 5, 5, 5, 5, 5, 5
giCommentsCounter init 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
iMsgIndx = giCommentsIndx[giCommentsCounter]
puts gSComments[iMsgIndx], 1
giCommentsCounter += 1
while indx < 11 do
    ifqmult = giFqs[indx] ;get frequency multiplier from array
    ifreq = ibasfreq * ifqmult
    iampmult = giAmps[indx] ;get amp multiplier
    iamp = iampmult / 20 ;scale
    event_i "i", 10, 0, p3, ifreq, iamp, ifqdev, iampdev, idurdev
    indx += 1
od
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
        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>
;Example by joachim heintz

```

In midi-triggered descendant of this instrument, we could - as one of many possible options - vary the amount of possible random variation according to the key velocity so that 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_Risset_played_by_Midi.csd

```

<CsoundSynthesizer>
<CsOptions>
-o dac -m0
</CsOptions>
<CsInstruments>
sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1
seed 0

;frequency and amplitude multipliers for 11 partials of Risset's bell
giFqs[] fillarray .56, .563, .92, .923, 1.19, 1.7, 2, 2.74, 3, 3.74, 4.07
giAmps[] fillarray 1, 2/3, 1, 1.8, 8/3, 5/3, 1.46, 4/3, 4/3, 1, 4/3

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
iamppmid ampmidi 1 ;receive midi-velocity and scale 0-1
;calculate maximum deviations depending on midi-velocity
ifqdev = imxfqdv * iamppmid
iampdev = imxampdv * iamppmid
idurdev = imxdurdv * iamppmid
;trigger subinstruments
indx = 0
while indx < 11 do

```

```

ifqmult    =      giFqs[indx]
ifreq      =      ibasfreq * ifqmult
iampmult   =      giAmps[indx]
iamp       =      iampmult / 20 ;scale
      event_i  "i", 10, 0, 3, ifreq, iamp, ifqdev, iampdev, idurdev
indx      +=      1
od
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
          outs     apart, apart
endin

</CsInstruments>
<CsScore>

</CsScore>
</CsoundSynthesizer>
;example by joachim heintz

```

Whether you can play examples like this in realtime will depend on the power of your computer. Have a look at chapter 2D (Live Audio) for tips on getting the best possible performance from your Csound orchestra.

Using a Recursive UDO

A recursive User-Defines Opcode, as described in chapter 03 G, is an elegant way to accomplish the task of individually controlled partials in an additive synthesis. One instance of the UDO performs one partial. It calls the next instance recursively until the desired number of partials is there. The audio signals are added in the recursion.

The next example demonstrates this in transforming the Risset bell code (04A07) to this approach. The coding style is more condensed here, so some comments are added after the code.

EXAMPLE 04A09_risset_bell_rec_udo.csd

```

<CsoundSynthesizer>
<CsOptions>
-o dac -m128
</CsOptions>
<CsInstruments>
sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1
seed 0

opcode AddSynth,a,i[]i[]iooo
/* iFqs[], iAmps[]: arrays with frequency ratios and amplitude multipliers
iBasFreq: base frequency (hz)
iPtlIndex: partial index (first partial = index 0)
iFreqDev, iAmpDev: maximum frequency (cent) and amplitude (db) deviation */
iFqs[], iAmps[], iBasFreq, iPtlIdx, iFreqDev, iAmpDev xin
iFreq = iBasFreq * iFqs[iPtlIdx] * cent(rnd31:i(iFreqDev,0))
iAmp = iAmps[iPtlIdx] * ampdb(rnd31:i(iAmpDev,0))
aPartial oscil iAmp, iFreq
if iPtlIdx < lenarray(iFqs)-1 then
  aPartial += AddSynth(iFqs,iAmps,iBasFreq,iPtlIdx+1,iFreqDev,iAmpDev)
endif
xout aPartial
endop

;frequency and amplitude multipliers for 11 partials of Risset's bell
giFqs[] fillarray .56, .563, .92, .923, 1.19, 1.7, 2, 2.74, 3, 3.74, 4.07
giAmps[] fillarray 1, 2/3, 1, 1.8, 8/3, 5/3, 1.46, 4/3, 4/3, 1, 4/3

instr Risset_Bell
ibasfreq = p4
iamp = ampdb(p5)
ifqdev = p6 ;maximum freq deviation in cents
iampdev = p7 ;maximum amp deviation in dB
aRisset AddSynth giFqs, giAmps, ibasfreq, 0, ifqdev, iampdev
aRisset *= transeg:a(0, .01, 0, iamp/10, p3-.01, -10, 0)
out aRisset, aRisset
endin

instr PlayTheBells
iMidiPitch random 60,70
schedule("Risset_Bell",0,random:i(2,8),mtof:i(iMidiPitch),
         random:i(-30,-10),30,6)
if p4 > 0 then
  schedule("PlayTheBells",random:i(1/10,1/4),1,p4-1)
endif
endin

</CsInstruments>
<CsScore>
;           base   db    frequency   amplitude
;           freq      deviation   deviation
;                           in cent     in dB
r 2 ;unchanged sound
i 1 0 5    400     -6    0          0
r 2 ;variations in frequency
i 1 0 5    400     -6    50       0
r 2 ;variations in amplitude
i 1 0 5    400     -6    0          10
s

```

```
i "PlayTheBells" 0 1 50 ;perform sequence of 50 bells
</CsScore>
</CsoundSynthesizer>
;example by joachim heintz
```

Some comments:

- Line 12-17: The main inputs are array with the frequencies *iFqs[]*, the array with the amplitudes *iAmps[]*, and the base frequency *iBasFreq*. The partial index *iPtlIndx* is by default zero, as well as the possible frequency and amplitude deviation of each partial.
- Line 18-19: The appropriate frequency and amplitude multiplier is selected from the array as *iFqs[iPtlIndx]* and *iAmps[iPtlIndx]*. The deviations are calculated for each partial by the *rnd31* opcode, a bipolar random generator which by default seeds from system clock.
- Line 21-23: The recursion is done if this is not the last partial. For the Risset bell it means: partials 0, 1, 2, ... are called until partial index 10. As index 10 is not smaller than the length of the frequency array (= 11) minus 1, it will not perform the recursion any more.
- Line 37: The envelope is applied for the sum of all partials (again in functional style, see chapter 03 I), as we don't use individual durations here.
- Line 41-47: The *PlayTheBells* instrument also uses recursion. It starts with p4=50 and calls the next instance of itself with p4=49, which in turn will call the next instance with p4=48, until 0 has been reached. The *Risset_Bell* instrument is scheduled with random values for duration, pitch and volume.

Csound Opcodes for Additive Synthesis

gbuzz, buzz and GEN11

[gbuzz](#) is useful for creating additive tones made of harmonically related cosine waves. Rather than define attributes for every partial individually, *gbuzz* allows us to define parameters that describe the entire additive tone in a more general way: specifically the number of partials in the tone, the partial number of the lowest partial present and an amplitude coefficient multiplier, which shifts the peak of spectral energy in the tone. Although number of harmonics (knh) and lowest harmonic (klh) are k-rate arguments, they are only interpreted as integers by the opcode; therefore changes from integer to integer will result in discontinuities in the output signal. The amplitude coefficient multiplier allows for smooth spectral modulations however. Although we lose some control of individual partials using *gbuzz*, we gain by being able to nimbly sculpt the spectrum of the tone it produces.

In the following example a 100Hz tone is created, in which the number of partials it contains rises from 1 to 20 across its 8 second duration. A spectrogram/sonogram displays how this manifests spectrally. A linear frequency scale is employed in the spectrogram so that harmonic partials appear equally spaced.

EXAMPLE 04A10_gbuzz.csd

```
<CsoundSynthesizer>

<CsOptions>
-o dac
</CsOptions>

<CsInstruments>
sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

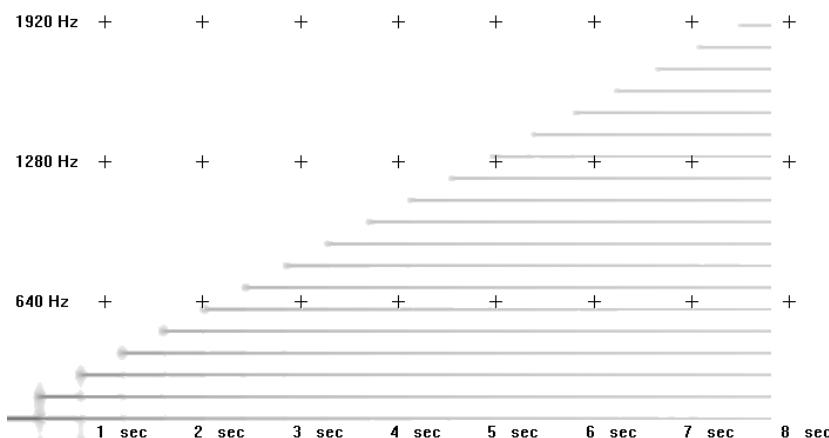
; a cosine wave
gicos ftgen 0, 0, 2^10, 11, 1

instr 1
knh line 1, p3, 20 ; number of harmonics
klh = 1 ; lowest harmonic
kmul = 1 ; amplitude coefficient multiplier
asig gbuzz 1, 100, knh, klh, kmul, gicos
outs asig, asig
endin

</CsInstruments>

<CsScore>
i 1 0 8
e
</CsScore>

</CsoundSynthesizer>
;example by Iain McCurdy
```



The total number of partials only reaches 19 because the `line` function only reaches 20 at the very conclusion of the note.

In the next example the number of partials contained within the tone remains constant but the partial number of the lowest partial rises from 1 to 20.

EXAMPLE 04A11_gbuzz_partials_rise.csd

```
<CsoundSynthesizer>
```

```

<CsOptions>
-o dac
</CsOptions>

<CsInstruments>
sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

; a cosine wave
gicos ftgen 0, 0, 2^10, 11, 1

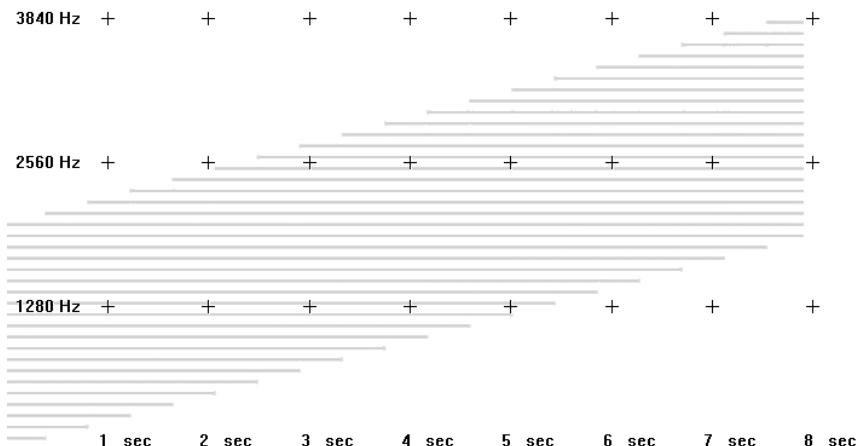
instr 1
knh = 20
klh line 1, p3, 20
kmul = 1
asig gbuzz 1, 100, knh, klh, kmul, gicos
    outs asig, asig
endin

</CsInstruments>

<CsScore>
i 1 0 8
e
</CsScore>

</CsoundSynthesizer>
;example by Iain McCurdy

```



In the spectrogram it can be seen how, as lowermost partials are removed, additional partials are added at the top of the spectrum. This is because the total number of partials remains constant at 20.

In the final *gbuzz* example the amplitude coefficient multiplier rises from 0 to 2. It can be heard (and seen in the spectrogram) how, when this value is zero, emphasis is on the lowermost partial and when this value is 2, emphasis is on the uppermost partial.

EXAMPLE 04A12_gbuzz_amp_coeff_rise.csd

```

<CsoundSynthesizer>

<CsOptions>
```

```

-o dac
</CsOptions>

<CsInstruments>
sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

; a cosine wave
gicos ftgen 0, 0, 2^10, 11, 1

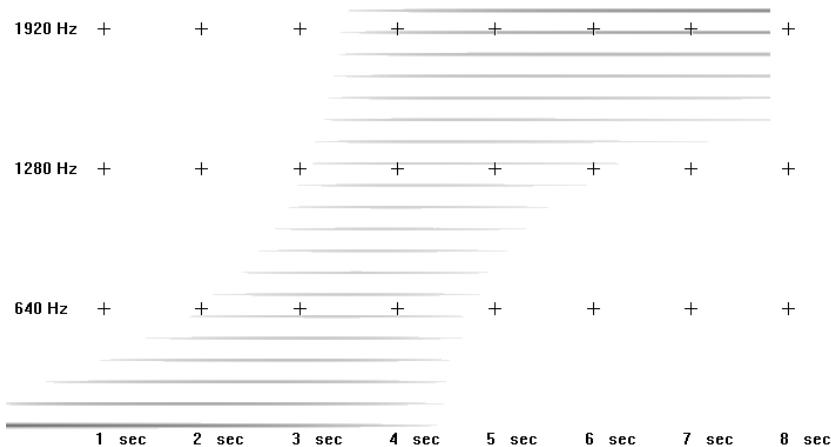
instr 1
knh = 20
klh = 1
kmul line 0, p3, 2
asig gbuzz 1, 100, knh, klh, kmul, gicos
outs asig, asig
endin

</CsInstruments>

<CsScore>
i 1 0 8
e
</CsScore>

</CsoundSynthesizer>
;example by Iain McCurdy

```



buzz is a simplified version of *gbuzz* with fewer parameters – it does not provide for modulation of the lowest partial number and amplitude coefficient multiplier.

GEN11 creates a function table waveform using the same parameters as *gbuzz*. If a *gbuss* tone is required but no performance time modulation of its parameters is needed, *GEN11* may provide a more efficient option. *GEN11* also opens the possibility of using its waveforms in a variety of other opcodes. *gbuzz*, *buzz* and *GEN11* may also prove useful as a source for subtractive synthesis.

hsboscil

The opcode **hsboscil** offers an interesting method of additive synthesis in which all partials are spaced an octave apart. Whilst this may at first seem limiting, it does offer simple means for

morphing the precise make up of its spectrum. It can be thought of as producing a sound spectrum that extends infinitely above and below the base frequency. Rather than sounding all of the resultant partials simultaneously, a window (typically a Hanning window) is placed over the spectrum, masking it so that only one or several of these partials sound at any one time. The user can shift the position of this window up or down the spectrum at k-rate and this introduces the possibility of spectral morphing. *hsbosil* refers to this control as *kbrite*. The width of the window can be specified (but only at i-time) using its *iOctCnt* parameter. The entire spectrum can also be shifted up or down, independent of the location of the masking window using the *ktone* parameter, which can be used to create a *Risset glissando*-type effect. The sense of the interval of an octave between partials tends to dominate but this can be undermined through the use of frequency shifting or by using a waveform other than a sine wave as the source waveform for each partial.

In the next example, instrument 1 demonstrates the basic sound produced by *hsbosil* whilst randomly modulating the location of the masking window (*kbrite*) and the transposition control (*ktone*). Instrument 2 introduces frequency shifting (through the use of the *hilbert* opcode) which adds a frequency value to all partials thereby warping the interval between partials. Instrument 3 employs a more complex waveform (pseudo-inharmonic) as the source waveform for the partials.

EXAMPLE 04A13_hsboscil.csd

```
<CsoundSynthesizer>
<CsOptions>
-o dac -m0
</CsOptions>
<CsInstruments>
sr = 44100
ksmps = 32
0dbfs = 1
nchnls = 2

giSine ftgen 0, 0, 2^10, 10, 1
; hanning window
giWindow ftgen 0, 0, 1024, -19, 1, 0.5, 270, 0.5
;a complex pseudo inharmonic waveform (partials scaled up X 100)
giWave ftgen 0, 0, 262144, 9, 100, 1.000, 0, 278, 0.500, 0, 518, 0.250, 0,
        816, 0.125, 0, 1166, 0.062, 0, 1564, 0.031, 0, 1910, 0.016, 0

instr 1 ; demonstration of hsboscil
kAmp = 0.3
kTone rspline -1,1,0.05,0.2 ; randomly shift spectrum up and down
kBrite rspline -1,3,0.4,2 ; randomly shift masking window up and down
iBasFreq = 200 ; base frequency
iOctCnt = 3 ; width of masking window
aSig hsboscil kAmp, kTone, kBrite, iBasFreq, giSine, giWindow, iOctCnt
out aSig, aSig
endin

instr 2 ; frequency shifting added
kAmp = 0.3
kTone = 0 ; spectrum remains static this time
kBrite rspline -2,5,0.4,2 ; randomly shift masking window up and down
iBasFreq = 75 ; base frequency
iOctCnt = 6 ; width of masking window
aSig hsboscil kAmp, kTone, kBrite, iBasFreq, giSine, giWindow, iOctCnt
; frequency shift the sound
kfshift = -357 ; amount to shift the frequency
areal,aimag hilbert aSig ; hilbert filtering
```

```
asin poscil 1, kfshift, giSine, 0 ; modulating signals
acos poscil 1, kfshift, giSine, 0.25
aSig = (areal*acos) - (aimag*asin) ; frequency shifted signal
out aSig, aSig
endin

instr 3 ; hsboscil using a complex waveform
kAmp = 0.3
kTone rspline -1,1,0.05,0.2 ; randomly shift spectrum up and down
kBrite rspline -3,3,0.1,1 ; randomly shift masking window
iBasFreq = 200
aSig hsboscil kAmp, kTone, kBrite, iBasFreq/100, giWave, giWindow
aSig2 hsboscil kAmp,kTone, kBrite, (iBasFreq*1.001)/100, giWave, giWindow
out aSig+aSig2, aSig+aSig2 ; mix signal with '\detuned\' version
endin

</CsInstruments>
<CsScore>
i 1 0 14
i 2 15 14
i 3 30 14
</CsScore>
</CsoundSynthesizer>
;example by iain mccurdy
```

Additive synthesis can still be an exciting way of producing sounds. It offers the user a level of control that other methods of synthesis simply cannot match. It also provides an essential workbench for learning about acoustics and spectral theory as related to sound.

04 B. SUBTRACTIVE SYNTHESIS

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 modulation synthesis.

A Csound Two-Oscillator Synthesizer

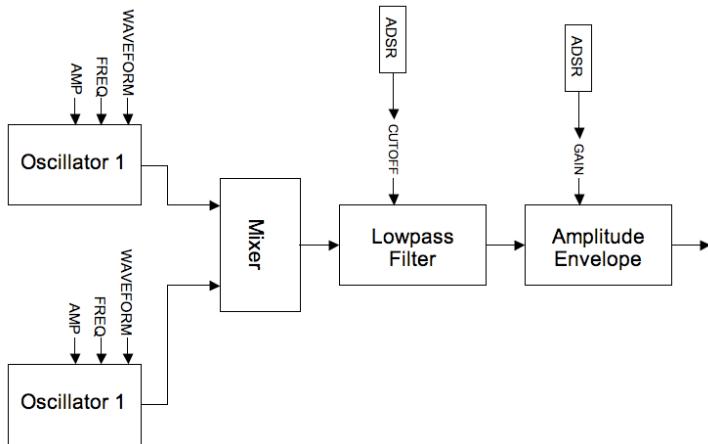
The first example represents perhaps the classic idea of subtractive synthesis: a simple two oscillator synth filtered using a single resonant lowpass filter. Many of the ideas used in this example have been inspired by the design of the [Minimoog](#) synthesizer (1970) and other similar instruments.

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. Waveforms such as sawtooths and square waves offer rich sources for subtractive synthesis as they contain a lot of sound energy across a wide range of frequencies - it could be said that white noise offers the richest sound source containing, as it does, energy at every frequency. A sine wave would offer a very poor source for subtractive synthesis as it contains energy at only one frequency. Other Csound opcodes that might provide rich sources are the [buzz](#) and [gbuzz](#) opcodes and the [GEN09](#), [GEN10](#), [GEN11](#) and [GEN19](#) GEN routines.

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. In the next example MIDI controller 1 will be used to control the global cutoff frequency for the filter.

A schematic for this instrument is shown below:



EXAMPLE 04B01_Subtractive_Midi.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

; assign 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>
;example by Iain McCurdy

```

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 a simple additive synthesis equivalent.¹ 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

¹It has been shown in the chapter about additive synthesis how this quality can be applied to additive synthesis by slight random deviations.

filter, a not insignificant 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](#). Obviously with so much repetition of similar code, some sort of abstraction would be a good idea (perhaps through a UDO or by using a macro), but here, and for the sake of clarity, it is left unabsttracted.

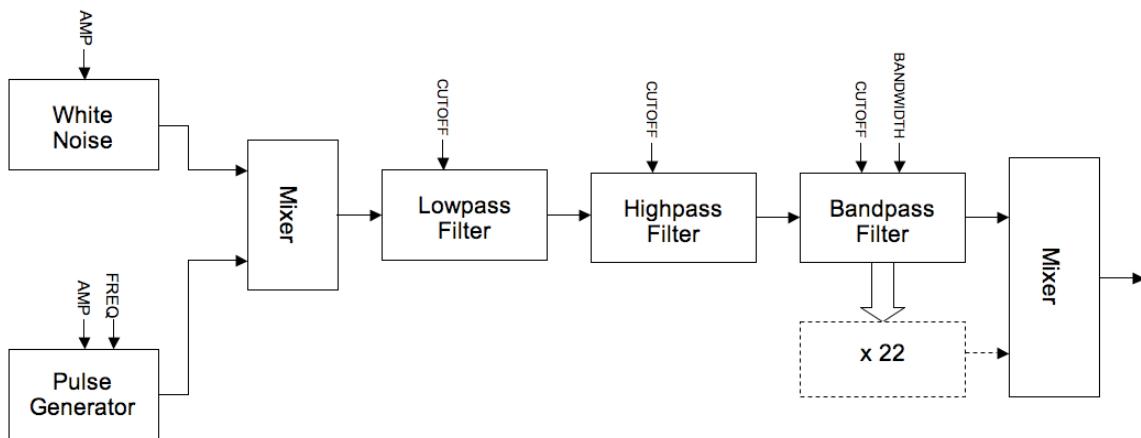
In addition to the white noise as a source, noise impulses are also used as a sound source (via the `mpulse` opcode). The instrument will automatically and randomly slowly crossfade between these two sound sources.

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 minimal ripple at the cutoff frequency.

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_Subtractive_timbres.csd

```

<CsoundSynthesizer>
<CsOptions>
-odac
</CsOptions>
<CsInstruments>
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,cpsocf(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 ; fundamental
;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>
;example written by Iain McCurdy

```

Vowel-Sound Emulation Using Bandpass Filtering

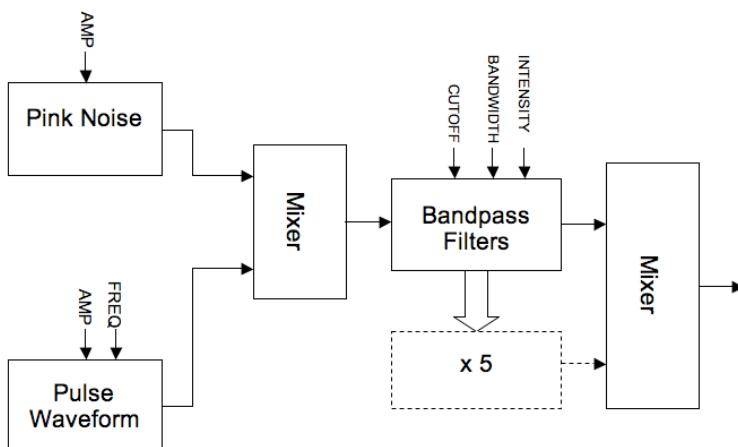
The final example in this section uses precisely tuned bandpass filters, to simulate the sound of the human voice expressing vowel sounds. Spectral resonances in this context are often referred to as [formants](#). Five formants are used to simulate the effect of the human mouth and head as a resonating (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_Subtractive_vowels.csd

```

<CsoundSynthesizer>
<CsOptions>
-odac
</CsOptions>
<CsInstruments>
sr = 44100
ksmps = 16
nchnls = 2
0dbfs = 1

;FUNCTION TABLES STORING DATA FOR VARIOUS VOICE FORMANTS

;BASS
giBF1 ftgen 0, 0, -5, -2, 600, 400, 250, 400, 350
giBF2 ftgen 0, 0, -5, -2, 1040, 1620, 1750, 750, 600
giBF3 ftgen 0, 0, -5, -2, 2250, 2400, 2600, 2400, 2400
giBF4 ftgen 0, 0, -5, -2, 2450, 2800, 3050, 2600, 2675
giBF5 ftgen 0, 0, -5, -2, 2750, 3100, 3340, 2900, 2950

giBDb1 ftgen 0, 0, -5, -2, 0, 0, 0, 0, 0
giBDb2 ftgen 0, 0, -5, -2, -7, -12, -30, -11, -20
giBDb3 ftgen 0, 0, -5, -2, -9, -9, -16, -21, -32
giBDb4 ftgen 0, 0, -5, -2, -9, -12, -22, -20, -28
giBDb5 ftgen 0, 0, -5, -2, -20, -18, -28, -40, -36

giBBW1 ftgen 0, 0, -5, -2, 60, 40, 60, 40, 40
giBBW2 ftgen 0, 0, -5, -2, 70, 80, 90, 80, 80
giBBW3 ftgen 0, 0, -5, -2, 110, 100, 100, 100, 100
giBBW4 ftgen 0, 0, -5, -2, 120, 120, 120, 120, 120
giBBW5 ftgen 0, 0, -5, -2, 130, 120, 120, 120, 120
  
```

```

;TENOR
giTF1 ftgen 0, 0, -5, -2, 650, 400, 290, 400, 350
giTF2 ftgen 0, 0, -5, -2, 1080, 1700, 1870, 800, 600
giTF3 ftgen 0, 0, -5, -2, 2650, 2600, 2800, 2600, 2700
giTF4 ftgen 0, 0, -5, -2, 2900, 3200, 3250, 2800, 2900
giTF5 ftgen 0, 0, -5, -2, 3250, 3580, 3540, 3000, 3300

giTDb1 ftgen 0, 0, -5, -2, 0, 0, 0, 0, 0
giTDb2 ftgen 0, 0, -5, -2, -6, -14, -15, -10, -20
giTDb3 ftgen 0, 0, -5, -2, -7, -12, -18, -12, -17
giTDb4 ftgen 0, 0, -5, -2, -8, -14, -20, -12, -14
giTDb5 ftgen 0, 0, -5, -2, -22, -20, -30, -26, -26

giTBW1 ftgen 0, 0, -5, -2, 80, 70, 40, 40, 40
giTBW2 ftgen 0, 0, -5, -2, 90, 80, 90, 80, 60
giTBW3 ftgen 0, 0, -5, -2, 120, 100, 100, 100, 100
giTBW4 ftgen 0, 0, -5, -2, 130, 120, 120, 120, 120
giTBW5 ftgen 0, 0, -5, -2, 140, 120, 120, 120, 120

;COUNTER TENOR
giCTF1 ftgen 0, 0, -5, -2, 660, 440, 270, 430, 370
giCTF2 ftgen 0, 0, -5, -2, 1120, 1800, 1850, 820, 630
giCTF3 ftgen 0, 0, -5, -2, 2750, 2700, 2900, 2700, 2750
giCTF4 ftgen 0, 0, -5, -2, 3000, 3000, 3350, 3000, 3000
giCTF5 ftgen 0, 0, -5, -2, 3350, 3300, 3590, 3300, 3400

giTBDb1 ftgen 0, 0, -5, -2, 0, 0, 0, 0, 0
giTBDb2 ftgen 0, 0, -5, -2, -6, -14, -24, -10, -20
giTBDb3 ftgen 0, 0, -5, -2, -23, -18, -24, -26, -23
giTBDb4 ftgen 0, 0, -5, -2, -24, -20, -36, -22, -30
giTBDb5 ftgen 0, 0, -5, -2, -38, -20, -36, -34, -30

giTBW1 ftgen 0, 0, -5, -2, 80, 70, 40, 40, 40
giTBW2 ftgen 0, 0, -5, -2, 90, 80, 90, 80, 60
giTBW3 ftgen 0, 0, -5, -2, 120, 100, 100, 100, 100
giTBW4 ftgen 0, 0, -5, -2, 130, 120, 120, 120, 120
giTBW5 ftgen 0, 0, -5, -2, 140, 120, 120, 120, 120

;ALTO
giAF1 ftgen 0, 0, -5, -2, 800, 400, 350, 450, 325
giAF2 ftgen 0, 0, -5, -2, 1150, 1600, 1700, 800, 700
giAF3 ftgen 0, 0, -5, -2, 2800, 2700, 2700, 2830, 2530
giAF4 ftgen 0, 0, -5, -2, 3500, 3300, 3700, 3500, 2500
giAF5 ftgen 0, 0, -5, -2, 4950, 4950, 4950, 4950, 4950

giADb1 ftgen 0, 0, -5, -2, 0, 0, 0, 0, 0
giADb2 ftgen 0, 0, -5, -2, -4, -24, -20, -9, -12
giADb3 ftgen 0, 0, -5, -2, -20, -30, -30, -16, -30
giADb4 ftgen 0, 0, -5, -2, -36, -35, -36, -28, -40
giADb5 ftgen 0, 0, -5, -2, -60, -60, -60, -55, -64

giABW1 ftgen 0, 0, -5, -2, 50, 60, 50, 70, 50
giABW2 ftgen 0, 0, -5, -2, 60, 80, 100, 80, 60
giABW3 ftgen 0, 0, -5, -2, 170, 120, 120, 100, 170
giABW4 ftgen 0, 0, -5, -2, 180, 150, 150, 130, 180
giABW5 ftgen 0, 0, -5, -2, 200, 200, 200, 135, 200

;SOPRANO
giSF1 ftgen 0, 0, -5, -2, 800, 350, 270, 450, 325
giSF2 ftgen 0, 0, -5, -2, 1150, 2000, 2140, 800, 700
giSF3 ftgen 0, 0, -5, -2, 2900, 2800, 2950, 2830, 2700
giSF4 ftgen 0, 0, -5, -2, 3900, 3600, 3900, 3800, 3800
giSF5 ftgen 0, 0, -5, -2, 4950, 4950, 4950, 4950, 4950

```

```

giSDb1 ftgen 0, 0, -5, -2, 0, 0, 0, 0, 0
giSDb2 ftgen 0, 0, -5, -2, -6, -20, -12, -11, -16
giSDb3 ftgen 0, 0, -5, -2, -32, -15, -26, -22, -35
giSDb4 ftgen 0, 0, -5, -2, -20, -40, -26, -22, -40
giSDb5 ftgen 0, 0, -5, -2, -50, -56, -44, -50, -60

giSBW1 ftgen 0, 0, -5, -2, 80, 60, 60, 70, 50
giSBW2 ftgen 0, 0, -5, -2, 90, 90, 90, 80, 60
giSBW3 ftgen 0, 0, -5, -2, 120, 100, 100, 100, 170
giSBW4 ftgen 0, 0, -5, -2, 130, 150, 120, 130, 180
giSBW5 ftgen 0, 0, -5, -2, 140, 200, 120, 135, 200

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 frequencies from tables
kCF1    tablei   kVow*5,giBF1+(iVoice*15)
kCF2    tablei   kVow*5,giBF1+(iVoice*15)+1
kCF3    tablei   kVow*5,giBF1+(iVoice*15)+2
kCF4    tablei   kVow*5,giBF1+(iVoice*15)+3
kCF5    tablei   kVow*5,giBF1+(iVoice*15)+4
; read formant intensity values from tables
kDB1    tablei   kVow*5,giBF1+(iVoice*15)+5
kDB2    tablei   kVow*5,giBF1+(iVoice*15)+6
kDB3    tablei   kVow*5,giBF1+(iVoice*15)+7
kDB4    tablei   kVow*5,giBF1+(iVoice*15)+8
kDB5    tablei   kVow*5,giBF1+(iVoice*15)+9
; read formant bandwidths from tables
kBW1    tablei   kVow*5,giBF1+(iVoice*15)+10
kBW2    tablei   kVow*5,giBF1+(iVoice*15)+11
kBW3    tablei   kVow*5,giBF1+(iVoice*15)+12
kBW4    tablei   kVow*5,giBF1+(iVoice*15)+13
kBW5    tablei   kVow*5,giBF1+(iVoice*15)+14
; create resonant formants by 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), aForm5*ampdbfs(kDB5)
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>
; 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)
; 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>
;example by Iain McCurdy

```

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`, `mode` and the more esoteric `phaser1`, `phaser2` and `resony`.

04 C. AMPLITUDE AND RING MODULATION

In *Amplitude Modulation* (AM) the amplitude of a *Carrier* oscillator is modulated by the output of another oscillator, called *Modulator*. So the carrier amplitude consists of a constant value, by tradition called *DC Offset*, and the modulator output which are added to each other.

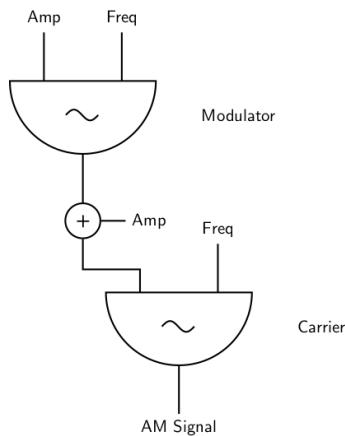


Figure 29.1: Basic Model of Amplitude Modulation

If this modulation is in the sub-audio range (less than 15 Hz), it is perceived as periodic volume modification.¹ Volume-modulation above approximately 15 Hz are perceived as timbre changes. So called *sidebands* appear. This transition is showed in the following example. The modulation frequency starts at 2 Hz and moves over 20 seconds to 100 Hz.

EXAMPLE 04C01_Simple_AM.csd

```
<CsoundSynthesizer>
<CsOptions>
-o dac
</CsOptions>
<CsInstruments>

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

¹For classical string instruments there is a *bow vibrato* which resembles this effect. If the *DC Offset* is weak in comparison to the modulation output, the comparison in classical music is the *tremolo* effect. Also *pulsation* is often used to describe AM with low frequencies.

```

instr 1
aRaise expseg 2, 20, 100
aModulator oscil 0.3, aRaise
iDCOffset = 0.3
aCarrier oscil iDCOffset+aModulator, 440
out aCarrier, aCarrier
endin

</CsInstruments>
<CsScore>
i 1 0 25
</CsScore>
</CsoundSynthesizer>
; example by Alex Hofmann and joachim heintz

```

Sidebands

The sidebands appear on both sides of the carrier frequency f_c . The frequency of the side bands is the sum and the difference between the carrier frequency and the modulator frequency: $f_c - f_m$ and $f_c + f_m$. The amplitude of each sideband is half of the modulator's amplitude.

So the sounding result of the following example can be calculated as this: $f_c = 440$ Hz, $f_m = 40$ Hz, so the result is a sound with 400, 440, and 480 Hz. The sidebands have an amplitude of 0.2. The amplitude of the carrier frequency starts with 0.2, moves to 0.4, and finally moves to 0. Note that we use an alternative way of applying AM here, shown in the AM2 instrument:

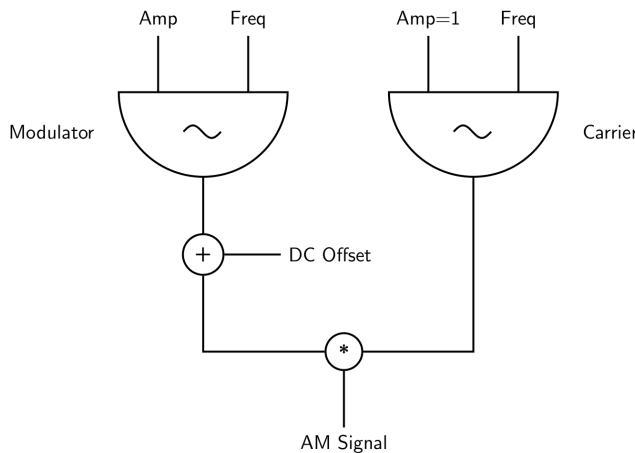


Figure 29.2: Alternative Model of Amplitude Modulation

It is equivalent to the signal flow in the first flow chart (AM1 here). It takes one more line, but now you can substitute any audio signal as carrier, not only an oscillator. So this is the bridge to using AM for the modification of sampled sound as shown in 05F.

EXAMPLE 04C02_Sidebands.csd

```

<CsoundSynthesizer>
<CsOptions>

```

```

-o dac
</CsOptions>
<CsInstruments>

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

instr AM1
aDC_Offset linseg 0.2, 1, 0.2, 5, 0.4, 3, 0
aModulator oscil 0.4, 40
aCarrier oscil aDC_Offset+aModulator, 440
out aCarrier, aCarrier
endin

instr AM2
aDC_Offset linseg 0.2, 1, 0.2, 5, 0.4, 3, 0
aModulator oscil 0.4, 40
aCarrier oscil 1, 440
aAM = aCarrier * (aModulator+aDC_Offset)
out aAM, aAM
endin

</CsInstruments>
<CsScore>
i "AM1" 0 10
i "AM2" 11 10
</CsScore>
</CsoundSynthesizer>
; example by Alex Hofmann and joachim heintz

```

At the end of this example, when the *DC Offset* was zero, we reached Ring Modulation (RM). Ring Modulation can thus be considered as special case of Amplitude Modulation, without any DC Offset. This is the very simple model:²

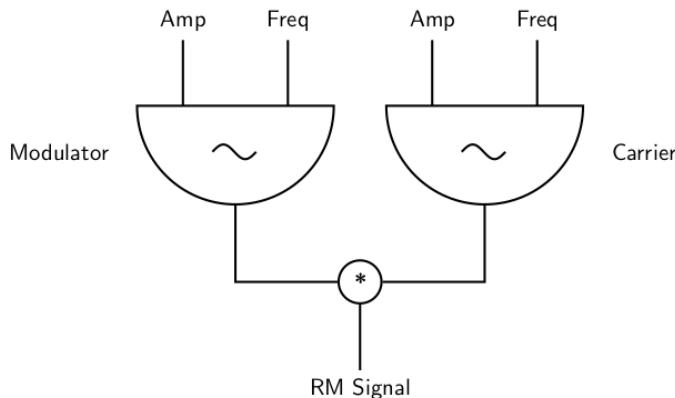


Figure 29.3: Ring Modulation as Multiplication of two Signals

If Ring Modulation happens in the sub-audio domain (less than 10 Hz), it will be perceived as *tremolo*.³ If it happens in the audio-domain, we get a sound with *only* the sidebands.

²Here expressed as multiplication. The alternative would be to feed the modulator's output in the amplitude input of the carrier.

³Note that the frequency of this tremolo in RM will be perceived twice as much as the frequency in AM because every half sine in the modulating signal is perceived as an own period.

AM/RM of Complex Sounds

If either the carrier or the modulator contain more harmonics, the resulting amplitude or ring modulated sound becomes more complex, because of two reasons. First, each partial in the source sound is the origin of two sideband partials in the result. So for three harmonics in the origin we yield six (RM) or nine (AM) partials in the result. And second, the spectrum of the origin is *shifted* by the AM or RM in a characteristic way. This can be demonstrated at a simple example.

Given a carrier signal which consists of three harmonics: 400, 800 and 1200 Hz. The ratio of these partials is 1 : 2 : 3, so our ear will perceive 400 Hz as base frequency. Ring Modulation with a frequency of 100 Hz will result in the frequencies 300, 500, 700, 900, 1100 and 1300 Hz. We have now a frequency every 200 Hz, and 400 Hz is not any more the base of it. (Instead, it will be heard as partials 3, 5, 7, 9, 11 and 13 of 100 Hz as base frequency.) In case we modulate with a frequency of 50 Hz, we get 350, 450, 750, 850, 1150 and 1250 Hz, so again a shifted spectrum, definitely not with 400 Hz as base frequency.

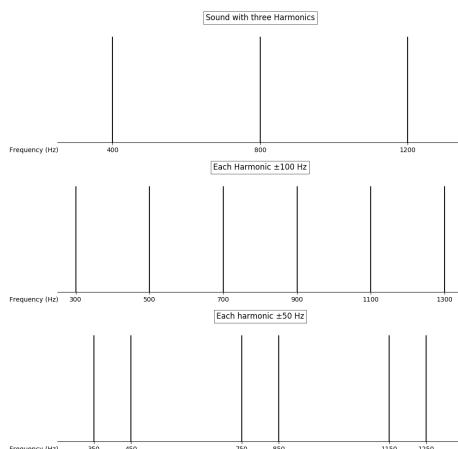


Figure 29.4: Frequency Shifting by Ring Modulation

The next example plays these variants.

EXAMPLE 04C03_RingMod.csd

```
<CsoundSynthesizer>
<CsOptions>
-o dac
</CsOptions>
<CsInstruments>

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

instr Carrier
aPartial_1 poscil .2, 400
aPartial_2 poscil .2, 800
aPartial_3 poscil .2, 1200
```

```
gaCarrier sum aPartial_1, aPartial_2, aPartial_3
;only output this signal if RM is not playing
if (active:k("RM") == 0) then
    out gaCarrier, gaCarrier
endif
endin

instr RM
iModFreq = p4
aRM = gaCarrier * poscil:a(1,iModFreq)
out aRM, aRM
endin

</CsInstruments>
<CsScore>
i "Carrier" 0 14
i "RM" 3 3 100
i "RM" 9 3 50
</CsScore>
</CsoundSynthesizer>
;example by joachim heintz
```


04 D. FREQUENCY MODULATION

Basic Model

In FM synthesis, the frequency of one oscillator (called the carrier) is modulated by the signal from another oscillator (called the modulator). The output of the modulating oscillator is added to the frequency input of the carrier oscillator.

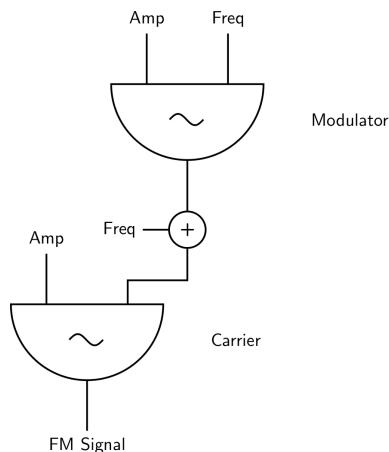


Figure 30.1: Basic Model of Frequency Modulation

The amplitude of the modulator determines the amount of modulation, or the frequency deviation from the fundamental carrier frequency. The frequency of the modulator determines how frequent the deviation will occur in one second. The amplitude of the modulator determines the amount of the deviation. An amplitude of 1 will alter the carrier frequency by ± 1 Hz, whereas an amplitude of 10 will alter the carrier frequency by ± 10 Hz. If the amplitude of the modulating signal is zero, there is no modulation and the output from the carrier oscillator is simply a sine wave with the frequency of the carrier. When modulation occurs, the signal from the modulation oscillator, a sine wave with frequency F_M , drives the frequency of the carrier oscillator both above and below the carrier frequency F_C . If the modulator is running in the sub-audio frequency range (below 20 Hz), the result of Modulation is vibrato. When the modulator's frequency rises in the audio range, we hear it as a change in the timbre of the carrier.

EXAMPLE 04D01_Frequency_modulation.csd

```

<CsoundSynthesizer>
<CsOptions>
-o dac
</CsOptions>
<CsInstruments>

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

instr FM_vibr ;vibrato as the modulator is in the sub-audio range
kModFreq randomi 5, 10, 1
kCarAmp linen 0.5, 0.1, p3, 0.5
aModulator oscil 20, kModFreq
aCarrier oscil kCarAmp, 400 + aModulator
out aCarrier, aCarrier
endin

instr FM_timbr ;timbre change as the modulator is in the audio range
kModAmp linseg 0, p3/2, 212, p3/2, 50
kModFreq line 25, p3, 300
kCarAmp linen 0.5, 0.1, p3, 0.5
aModulator oscil kModAmp, kModFreq
aCarrier oscil kCarAmp, 400 + aModulator
out aCarrier, aCarrier
endin

</CsInstruments>
<CsScore>
i "FM_vibr" 0 10
i "FM_timbr" 10 10
</CsScore>
</CsoundSynthesizer>
;example by marijana janevska

```

Carrier/Modulator Ratio

The position of the frequency components generated by FM depends on the relationship of the carrier frequency to the modulating frequency $F_C:F_M$. This is called the ratio. When $F_C:F_M$ is a simple integer ratio, such as 4:1 (as in the case of two signals at 400 and 100 Hz), FM generates harmonic spectra, that is sidebands that are integer multiples of the carrier and modulator frequencies. When $F_C:F_M$ is not a simple integer ratio, such as 8:2.1 (as in the case of two signals at 800 and 210 Hz), FM generates inharmonic spectra (noninteger multiples of the carrier and modulator).

EXAMPLE 04D02_Ratio.csd

```

<CsoundSynthesizer>
<CsOptions>
-o dac
</CsOptions>
<CsInstruments>

```

```

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

instr Ratio
kRatio = p4
kCarFreq = 400
kModFreq = kCarFreq/kRatio
aModulator oscil 500, kModFreq
aCarrier oscil 0.3, kCarFreq + aModulator
aOut linen aCarrier, .1, p3, 1
out aOut, aOut
endin

</CsInstruments>
<CsScore>
i 1 0 5 2
i . + . 2.1
</CsScore>
</CsoundSynthesizer>
;example written by marijana janevska

```

Index of Modulation

FM of two sinusoids generates a series of sidebands around the carrier frequency F_C . Each sideband spreads out at a distance equal to a multiple of the modulating frequency F_M .

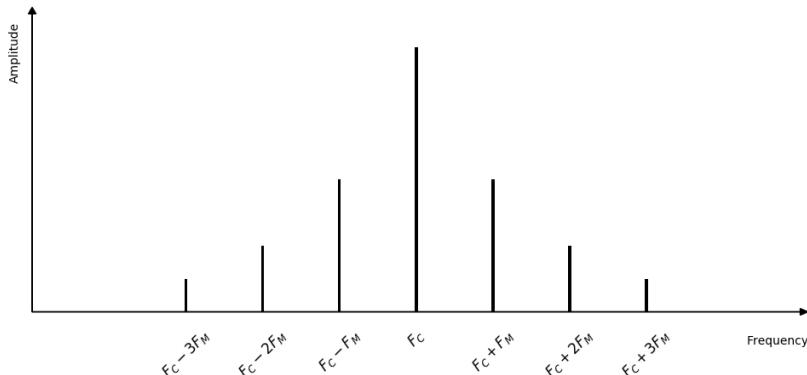


Figure 30.2: FM Sidebands

The bandwidth of the FM spectrum (the number of sidebands) is controlled by the index of modulation I . The Index is defined mathematically according to the following relation:

$$I = A_M : F_M$$

where A_M is the amount of frequency deviation (in Hz) from the carrier frequency. Hence, A_M is a way of expressing the depth or amount of modulation. The amplitude of each sideband depends on the index of modulation. When there is no modulation, the index of modulation is zero and all the signal power resides in the carrier frequency. Increasing the value of the index causes the sidebands to acquire more power at the expense of the power of the carrier frequency. The wider

the deviation, the more widely distributed is the power among the sidebands and the greater the number of sidebands that have significant amplitudes. The number of significant sideband pairs (those that are more than 1/100 the amplitude of the carrier) is approximately $I+1$. For certain values of the carrier and modulator frequencies and Index, extreme sidebands reflect out of the upper and lower ends of the spectrum, causing audible side effects. When the lower sidebands extend below 0 Hz, they reflect back into the spectrum in 180 degree phase inverted form. Negative frequency components add richness to the lower frequency parts of the spectrum, but if negative components overlap exactly with positive components, they can cancel each other. In simple FM, both oscillators use sine waves as their source waveform, although any waveform can be used. The FM can produce such rich spectra, that, when one waveform with a large number of spectral components frequency modulates another, the resulting sound can be so dense that it sounds harsh and undefined. Aliasing can occur easily.

EXAMPLE 04D03_Index.cs

```
<CsoundSynthesizer>
<CsOptions>
-o dac
</CsOptions>
<CsInstruments>

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

instr Rising_index
kModAmp = 400
kIndex linseg 3, p3, 8
kModFreq = kModAmp/kIndex
aModulator oscil kModAmp, kModFreq
aCarrier oscil 0.3, 400 + aModulator
aOut linen aCarrier, .1, p3, 1
out aOut, aOut
endin

</CsInstruments>
<CsScore>
i "Rising_index" 0 10
</CsScore>
</CsoundSynthesizer>
;example by marijana janevska and joachim heintz
```

Standard FM with Ratio and Index

In the basic FM model three variables are given: the frequency of the carrier (F_C or simply C), the frequency of the modulator (F_M or simply M) and the amplitude of the modulator which results in the frequency deviation (so A_M or D). By introducing the Ratio (C:M) and the Index (D:M) as musically meaningful values, it makes sense to transform the previous C, M and D input to C, R and I. C yields the base (or perhaps better: middle) frequency of the sound, R yields the overall characteristic of the timbre, I yields the emergence of the side bands. The three musically meaningful input values can easily be transformed into the basic model:

```
if R = C : M then M = C : R and
if I = D : M then D = I · M.
```

EXAMPLE 04D04_Standard.csd

```
<CsoundSynthesizer>
<CsOptions>
-odac -m128
</CsOptions>
<CsInstruments>

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

instr Standard

//input
iC = 400
iR = p4 ;ratio
iI = p5 ;index
prints "Ratio = %.3f, Index = %.3f\n", iR, ii

//transform
iM = iC / iR
iD = iI * iM

//apply to standard model
aModulator oscil iD, iM
aCarrier oscil 0.3, iC + aModulator
aOut linen aCarrier, .1, p3, 1
out aOut, aOut

endin

instr PlayMess

kC randomi 300, 500, 1, 2, 400
kR randomi 1, 2, 2, 3
kI randomi 1, 5, randomi:k(3,10,1,3), 3

//transform
kM = kC / kR
kD = kI * kM

//apply to standard model
aModulator oscil kD, kM
aCarrier oscil ampdb(port:k(kI*5-30,.1)), kC + aModulator
aOut linen aCarrier, .1, p3, p3/10
out aOut, aOut

endin

</CsInstruments>
<CsScore>
//changing the ratio at constant index=3
i "Standard" 0 3 1 3
i . + . 1.41 .
i . + . 1.75 .
i . + . 2.07 .
s
```

```
//changing the index at constant ratio=3.3
i "Standard" 0 3 3.3 0
i . + . . 1
i . + . . 5
i . + . . 10
s
//let some nonsense happen
i "PlayMess" 0 30
</CsScore>
</CsoundSynthesizer>
;example by joachim heintz
```

Using the `foscil` opcode

Basic FM synthesis can be implemented by using the `foscil` opcode, which effectively connects two oscil opcodes in the familiar Chowning FM setup. In the example below *kDenominator* is a value that when multiplied by the *kCar* parameter, gives the Carrier frequency and when multiplied by the *kMod* parameter, gives the Modulating frequency.

EXAMPLE 04D05_basic_FM_with_foscil.csd

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

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

giSine ftgen 0, 0, 8192, 10, 1

instr 1; basic FM using the foscil opcode
kDenominator = 110
kCar = 3
kMod = 1
kIndex randomi 1, 2, 20
aFM foscil 0.1, kDenominator, kCar, kMod, kIndex, giSine
outs aFM, aFM
endin

instr 2; basic FM with jumping (noisy) Denominator value
kDenominator random 100, 120
kCar = 3
kMod = 1
kIndex randomi 1, 2, 20
aFM foscil 0.1, kDenominator, kCar, kMod, kIndex, giSine
outs aFM, aFM
endin

instr 3; basic FM with jumping Denominator and moving Modulator
kDenominator random 100, 120
kCar = 3
kMod randomi 0, 5, 100, 3
kIndex randomi 1, 2, 20
```

```

aFM foscil 0.1, kDenominator, kCar, kMod, kIndex, giSine
outs aFM, aFM
endin

</CsInstruments>
<CsScore>
i 1 0 10
i 2 12 10
i 3 24 10
</CsInstruments>
</CsScore>
</CsoundSynthesizer>
;example by Marijana Janevska

```

In the example above, in instr 1 the Carrier has a frequency of 330 Hz, the Modulator has a frequency of 110 Hz and the value of the index changes randomly between 1 and 2, 20 times a second. In instr 2, the value of the Denominator is not static. Its value changes randomly between 100 and 120, which makes all the other parameters' values change (Carrier and Modulator frequencies and Index). In instr 3 we add a changing value to the parameter, that when multiplied with the Denominator value, gives the frequency of the Modulator, which gives even more complex spectra because it affects the value of the Index, too.

More Complex FM Algorithms

Combining more than two oscillators (operators) is called complex FM synthesis. Operators can be connected in different combinations: Multiple modulators FM and Multiple carriers FM.

Multiple Modulators (MM FM)

In multiple modulator frequency modulation, more than one oscillator modulates a single carrier oscillator. The carrier is always the last operator in the row. Changing its pitch shifts the whole sound. All other operators are modulators, changing their pitch and especially amplitude alters the sound-spectrum. Two basic configurations are possible: parallel and serial. In parallel MM FM, two sinewaves simultaneously modulate a single carrier oscillator. The principle here is, that (Modulator1:Carrier) and (Modulator2:Carrier) will be separate modulations and later added together.

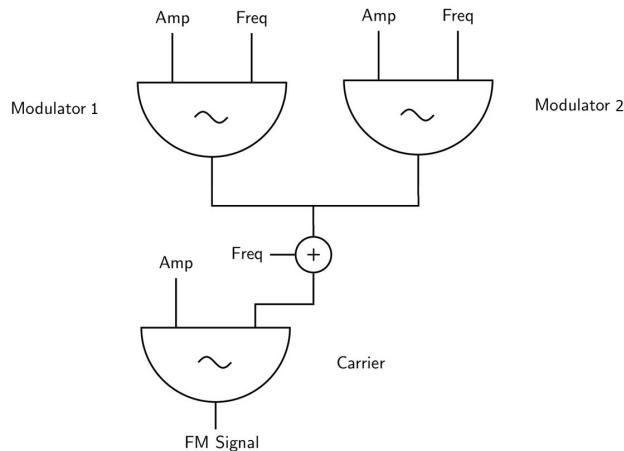


Figure 30.3: Multiple Modulator FM

EXAMPLE CSOUND PARALLEL_MM_FM.CSD

```

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

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

instr parallel_MM_FM
kAmpMod1 randomi 200, 500, 20
aModulator1 poscil kAmpMod1, 700
kAmpMod2 randomi 4, 10, 5
kFreqMod2 randomi 7, 12, 2
aModulator2 poscil kAmpMod2, kFreqMod2
kFreqCar randomi 50, 80, 1, 3
aCarrier poscil 0.2, kFreqCar+aModulator1+aModulator2
out aCarrier, aCarrier
endin

</CsInstruments>
<CsScore>
i "parallel_MM_FM" 0 20
</CsScore>
</CsoundSynthesizer>
;example by Alex Hofmann and Marijana Janevska

```

In serial MM FM, the output of the first modulator is added with a fixed value and then fed to the second modulator, which then is applied to the frequency input of the carrier. This is much more complicated to calculate and the timbre becomes harder to predict, because Modulator1:Modulator2 produces a complex spectrum, which then modulates the carrier.

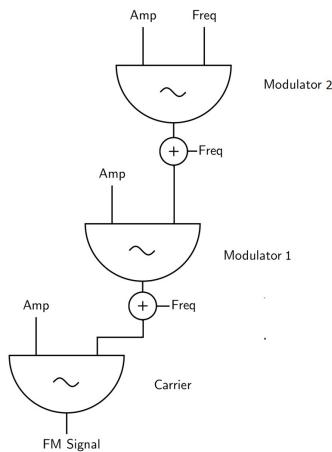


Figure 30.4: Serial Modulator FM

EXAMPLE 04D07_Serial_MM_FM.csd

```

<CsSoundSynthesizer>
<CsOptions>
-odac
</CsOptions>
<CsInstruments>

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

instr serial_MM_FM
kAmpMod2 randomi 200, 1400, .5
aModulator2 poscil kAmpMod2, 700
kAmpMod1 linseg 400, 15, 1800
aModulator1 poscil kAmpMod1, 290+aModulator2
aCarrier poscil 0.2, 440+aModulator1
outs aCarrier, aCarrier
endin

</CsInstruments>
<CsScore>
i "serial_MM_FM" 0 20
</CsScore>
</CsSoundSynthesizer>
;example by Alex Hofmann and Marijana Janevska

```

Multiple Carriers (MC FM)

By multiple carrier frequency modulation, we mean an FM instrument in which one modulator simultaneously modulates two or more carrier oscillators.

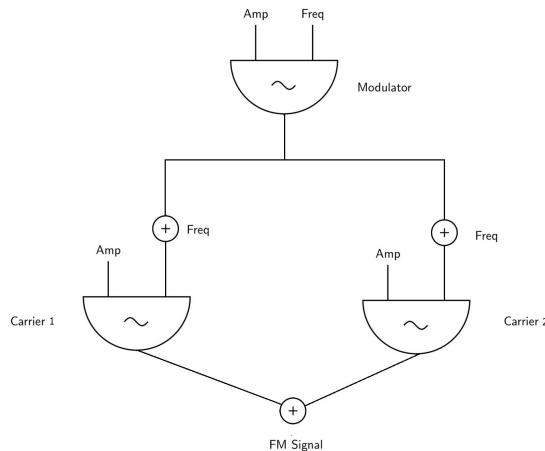


Figure 30.5: Multiple Carrier FM

EXAMPLE 04D08_MC_FM.csd

```

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

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

instr FM_two_carriers
aModulator oscil 100, randomi:k(10,15,1,3)
aCarrier1 oscil 0.3, 700 + aModulator
aCarrier2 oscil 0.1, 701 + aModulator
outs aCarrier1+aCarrier2, aCarrier1+aCarrier2
endin

</CsInstruments>
<CsScore>
i "FM_two_carriers" 0 20
</CsScore>
</CsoundSynthesizer>
;example by Marijana Janevska
  
```

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. By using envelopes to control the modulation index and the overall amplitude evolving sounds with enormous spectral variations can be created. Chowning showed these possibilities in his pieces, in which various sound transformations occur. In the piece Sabelithe a drum sound morphes over the time into a trumpet tone. In the example below, the amplitude of the Modulator has a complex envelope in the attack of the sound, which gives the trumpet-like timbre.

EXAMPLE 04D09_Trumpet.csd

```

<CsoundSynthesizer>
<CsOptions>
-o dac
</CsOptions>
<CsInstruments>

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

instr simple_trumpet
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 oscil aModAmp, kModFreq
aCarrier oscil 0.3*aEnv, kCarFreq+aModulator
outs aCarrier, aCarrier
endin

</CsInstruments>
<CsScore>
i 1 0 2
</CsScore>
</CsoundSynthesizer>
;example by Alex Hofmann

```

The following example uses the same instrument, with different settings to generate a bell-like sound:

EXAMPLE 04D10_Bell.csd

```

<CsoundSynthesizer>
<CsOptions>
-o dac
</CsOptions>
<CsInstruments>

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

instr bell_like
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 oscil aModAmp, kModFreq
aCarrier oscil 0.3*aEnv, kCarFreq+aModulator
outs aCarrier, aCarrier
endin

</CsInstruments>
<CsScore>
i "bell_like" 0 9
</CsScore>
</CsoundSynthesizer>
;example by Alex Hofmann

```

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.

For a feedback FM system, it can happen that the self-modulation comes to a zero point, which would hang the whole system. To avoid this, the carriers table-lookup phase is modulated, instead of its pitch.

Also 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 (cf <http://yala.freeservers.com/t2synths.htm#DX7>).

To build a PM-synth in Csound the tablei opcode substitutes the FM oscillator. In order to step through the f-table, a phasor will output the necessary steps.

EXAMPLE 04D11_Phase modulation and Feedback FM.csd

```

<CsoundSynthesizer>
<CsOptions>
-o dac
</CsOptions>
<CsInstruments>

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

giSine ftgen 0, 0, 8192, 10, 1

instr PM
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 oscil kIndex*aEnv, kModFreq
aPhase phasor kCarFreq
aCarrier tablei aPhase+aModulator, giSine, 1, 0, 1
out aCarrier*aEnv, aCarrier*aEnv
endin

</CsInstruments>

```

```
<CsScore>
i "PM" 0 9
</CsScore>
</CsoundSynthesizer>
;example by Alex Hofmann
```

In the last example we use the possibilities of self-modulation (feedback-modulation) of the oscillator. So here the oscillator is both modulator and carrier. To control the amount of modulation, an envelope scales the feedback.

EXAMPLE 04D12_Feedback modulation.csd

```
<CsoundSynthesizer>
<CsOptions>
-o dac
</CsOptions>
<CsInstruments>

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

giSine ftgen 0, 0, 8192, 10, 1

instr 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), giSine, 1, 0, 1
outs aCarrier*aAmpEnv, aCarrier*aAmpEnv
endin

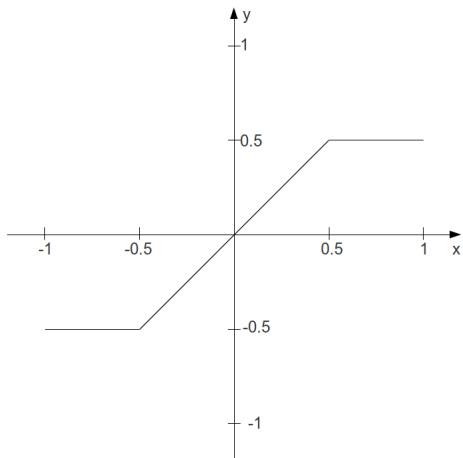
</CsInstruments>
<CsScore>
i "feedback_PM" 0 9
</CsScore>
</CsoundSynthesizer>
;example by Alex Hofmann
```


04 E. WAVESHAPING

Waveshaping is in some ways a relation of modulation techniques such as frequency or phase modulation. Waveshaping can create quite dramatic sound transformations through the application of a very simple process. In FM (frequency modulation) modulation synthesis 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 look-up 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 graph:

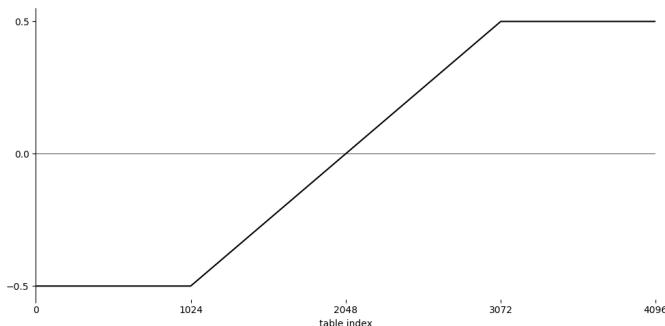


Basic Implementation Model

Although Csound contains several opcodes for waveshaping, implementing waveshaping from first principles as Csound code is fairly straightforward. The x-axis is the amplitude of every single sample, which is in the range of -1 to +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**. This statement will create a table of 4096 points with 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 Csound. 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. 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 we give 4097 as the table size instead of 4096.

This is the code for simple waveshaping using our transfer function which has been discussed previously:

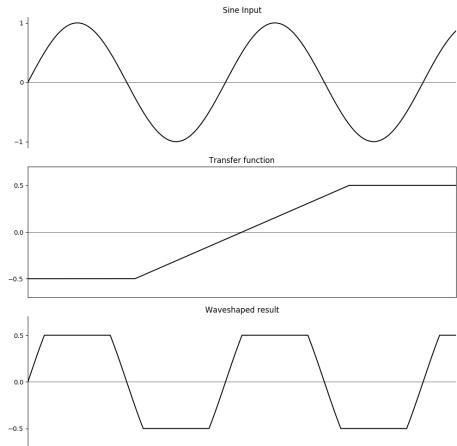
EXAMPLE 04E01_Simple_waveshaping.csd

```
<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
          out        aWavShp, aWavShp
endin
```

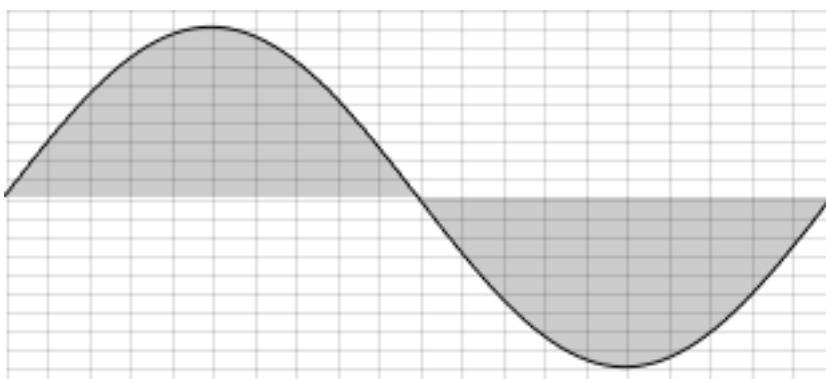
```
</CsInstruments>
<CsScore>
i 1 0 10
</CsScore>
</CsoundSynthesizer>
;example by joachim heintz
```



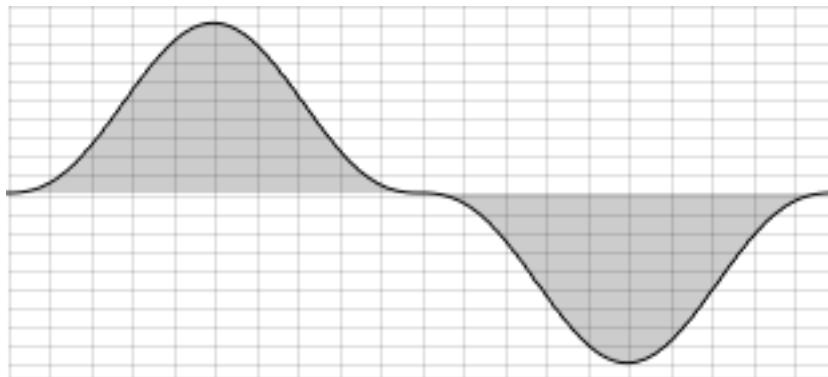
Powershape

The `powershape` opcode performs waveshaping by simply raising all samples to the power of a user given exponent. Its main innovation is that the polarity of samples within the negative domain will be retained. It simply performs the power function on absolute values (negative values made positive) and then reinstates the minus sign if required. It also normalises the input signal between -1 and 1 before shaping and then rescales the output by the inverse of whatever multiple was required to normalise the input. This ensures useful results but does require that the user states the maximum amplitude value expected in the opcode declaration and thereafter abide by that limit. The exponent, which the opcode refers to as *shape amount*, can be varied at k-rate thereby facilitating the creation of dynamic spectra upon a constant spectrum input.

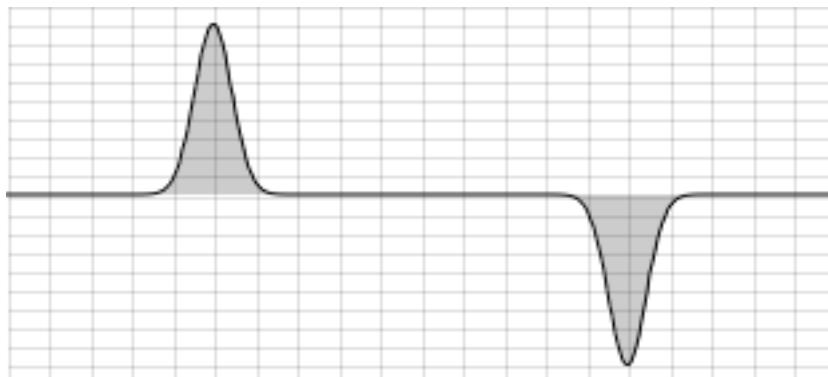
If we consider the simplest possible input - again a sine wave - a shape amount of 1 will produce no change (raising any value to the power of 1 leaves that value unchanged).



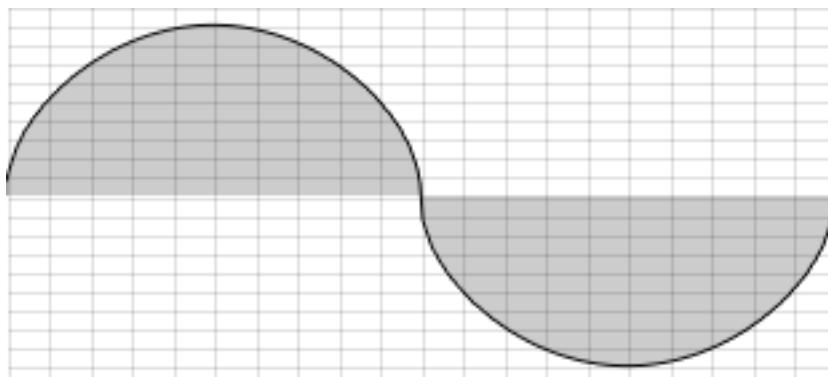
A shaping amount of 2.5 will visibly “squeeze” the waveform as values less than 1 become increasingly biased towards the zero axis.



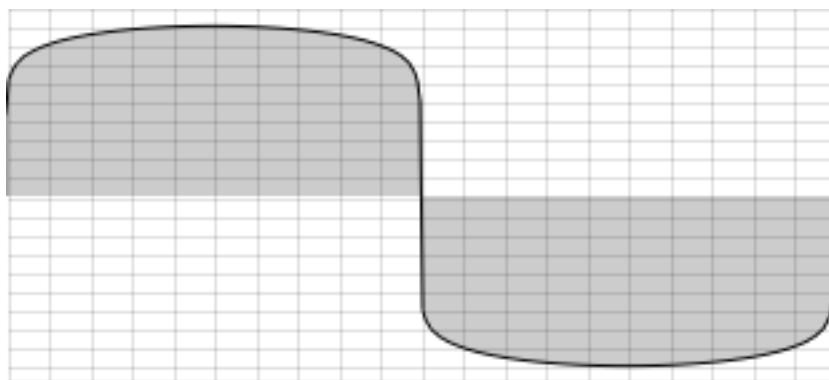
Much higher values will narrow the positive and negative peaks further. Below is the waveform resulting from a shaping amount of 50.



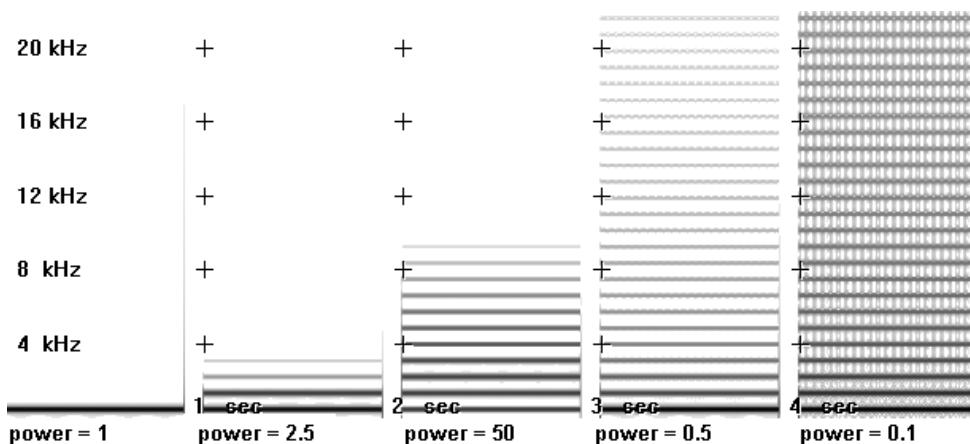
Shape amounts less than 1 (but greater than zero) will give the opposite effect of drawing values closer to -1 or 1. The waveform resulting from a shaping amount of 0.5 shown below is noticeably more rounded than the sine wave input.



Reducing shape amount even closer to zero will start to show squaring of the waveform. The result of a shape amount of 0.1 is shown below.



The sonograms of the five examples shown above are as shown below:



EXAMPLE 04E02_Powershape.csd

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

instr Powershape
    iAmp = 0.2
    iFreq = 300
    aIn poscil iAmp, iFreq
    ifullscale = iAmp
    kShapeAmount linseg 1, 1.5, 1, .5, p4, 1.5, p4, .5, p5
    aOut powershape aIn, kShapeAmount, ifullscale
    out aOut, aOut
endin
</CsInstruments>
<CsScore>
i "Powershape" 0 6 2.5 50
i "Powershape" 7 6 0.5 0.1
</CsScore>
</CsoundSynthesizer>
;example by joachim heintz
```

As power (shape amount) is increased from 1 through 2.5 to 50, it can be observed how harmonic

partials are added. It is worth noting also that when the power exponent is 50 the strength of the fundamental has waned somewhat. What is not clear from the sonogram is that the partials present are only the odd numbered ones. As the power exponent is reduced below 1 through 0.5 and finally 0.1, odd numbered harmonic partials again appear but this time the strength of the fundamental remains constant. It can also be observed that aliasing is becoming a problem as evidenced by the vertical artifacts in the sonograms for 0.5 and in particular 0.1. This is a significant concern when using waveshaping techniques. Raising the sampling rate can provide additional headroom before aliasing manifests but ultimately subtlety in waveshaping's use is paramount.

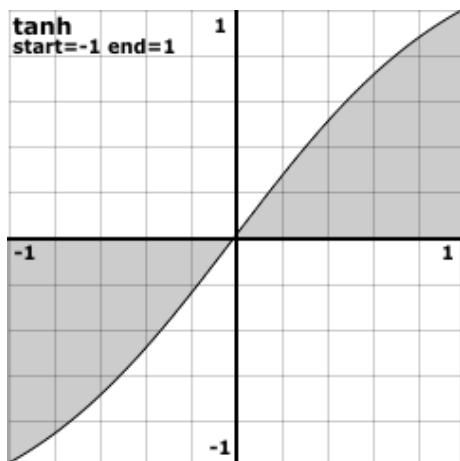
Distort

The [distort](#) opcode, authored by Csound's original creator Barry Vercoe, was originally part of the *Extended Csound* project but was introduced into Canonical Csound in version 5. It waveshapes an input signal according to a transfer function provided by the user using a function table. At first glance this may seem to offer little more than what we have already demonstrated from first principles, but it offers a number of additional features that enhance its usability. The input signal first has soft-knee compression applied before being mapped through the transfer function. Input gain is also provided via the *distortion amount* input argument and this provides dynamic control of the waveshaping transformation. The result of using compression means that spectrally the results are better behaved than is typical with waveshaping. A common transfer function would be the hyperbolic tangent (*tanh*) function. Csound possesses an GEN routine [GENtanh](#) for the creation of tanh functions:

```
GENTanh
f # time size "tanh" start end rescale
```

By adjusting the *start* and *end* values we can modify the shape of the *tanh* transfer function and therefore the aggressiveness of the waveshaping (*start* and *end* values should be the same absolute values and negative and positive respectively if we want the function to pass through the origin from the lower left quadrant to the upper right quadrant).

Start and end values of -1 and 1 will produce a gentle "s" curve.

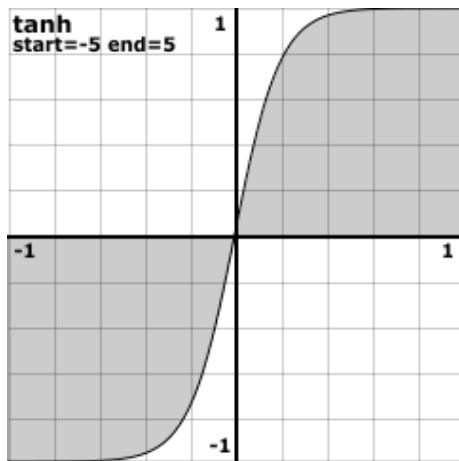


This represents only a very slight deviation from a straight line function from (-1,-1) to (1,1) - which would produce no distortion - therefore the effects of the above used as a transfer function will be

extremely subtle.

Start and end points of -5 and 5 will produce a much more dramatic curve and more dramatic waveshaping:

```
f 1 0 1024 "tanh" -5 5 0
```



Note that the GEN routine's argument p7 for rescaling is set to zero ensuring that the function only ever extends from -1 and 1. The values provided for *start* and *end* only alter the shape.

In the following test example a sine wave at 200 hz is waveshaped using distort and the tanh function shown above.

EXAMPLE 04E03_Distort_1.csd

```
<CsoundSynthesizer>
<CsOptions>
-dm0 -odac
</CsOptions>

<CsInstruments>

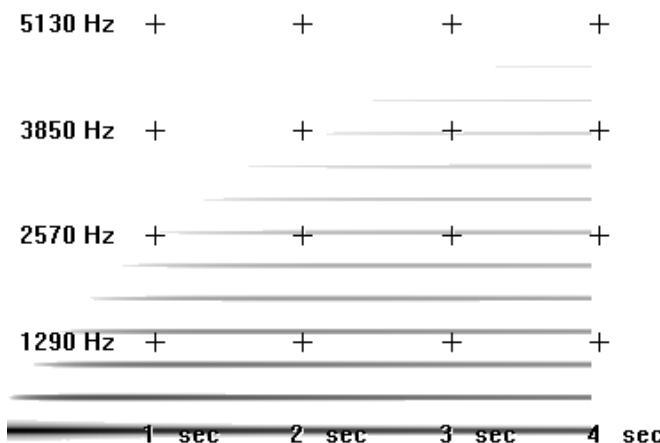
sr = 44100
ksmps =32
nchnls = 1
0dbfs = 1

giSine    ftgen  1,0,1025,10,1      ; sine function
giTanh    ftgen  2,0,257,"tanh",-10,10,0 ; tanh function

instr 1
aSig    oscil   1, 200, giSine        ; a sine wave
kAmt    line    0, p3, 1             ; rising distortion amount
aDst    distort  aSig, kAmt, giTanh   ; distort the sine tone
        out     aDst*0.1
endin

</CsInstruments>
<CsScore>
i 1 0 4
</CsScore>
</CsoundSynthesizer>
;example by Iain McCurdy
```

The resulting sonogram looks like this:



As the distort amount is raised from zero to 1 it can be seen from the sonogram how upper partials emerge and gain in strength. Only the odd numbered partials are produced, therefore over the fundamental at 200 hz partials are present at 600, 1000, 1400 hz and so on. If we want to restore the even numbered partials we can simultaneously waveshape a sine at 400 hz, one octave above the fundamental as in the next example:

EXAMPLE 04E04_Distort_2.csd

```

<CsoundSynthesizer>
<CsOptions>
-dm0 -odac
</CsOptions>
<CsInstruments>

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

giSine    ftgen    1,0,1025,10,1
giTanh    ftgen    2,0,257,"tanh",-10,10,0

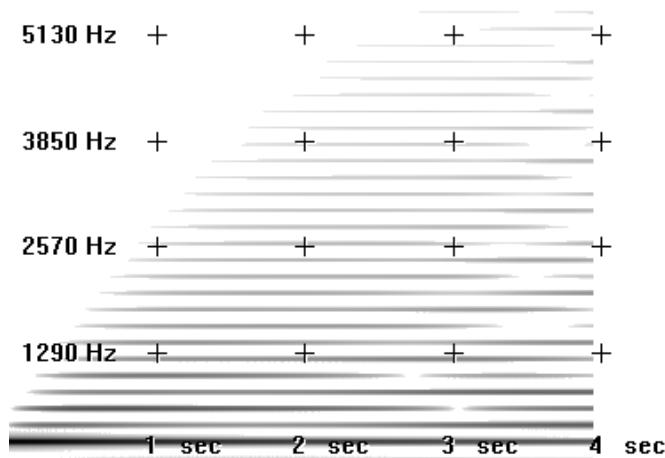
instr 1
kAmt    line    0, p3, 1           ; rising distortion amount
aSig    oscil   1, 200, giSine      ; a sine
aSig2   oscil   kAmt*0.8,400,giSine ; a sine an octave above
aDst    distort  aSig+aSig2, kAmt, giTanh ; distort a mixture of the two sines
        out     aDst*0.1
endin

</CsInstruments>

<CsScore>
i 1 0 4
</CsScore>
</CsoundSynthesizer>
;example by Iain McCurdy

```

The higher of the two sines is faded in using the distortion amount control so that when distortion amount is zero we will be left with only the fundamental. The sonogram looks like this:



What we hear this time is something close to a sawtooth waveform with a rising low-pass filter. The higher of the two input sines at 400 hz will produce overtones at 1200, 2000, 2800 ... thereby filling in the missing partials.

04 F. GRANULAR SYNTHESIS

In his *Computer Music Tutorial*, Curtis Roads gives an interesting introductory model for granular synthesis. A sine as source waveform is modified by a repeating envelope. Each envelope period creates one grain.

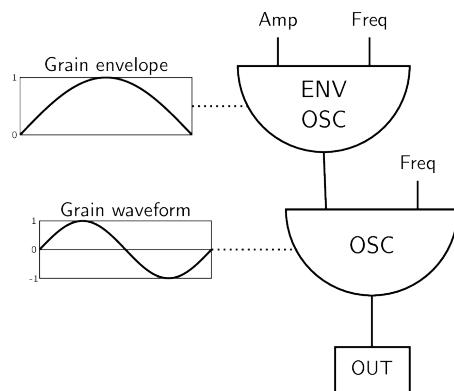


Figure 32.1: After Curtis Roads, Computer Music Tutorial, Fig. 5.11

In our introductory example, we will start with 1 Hz as frequency for the envelope oscillator, then rising to 10 Hz, then to 20, 50 and finally 300 Hz. The grain durations are therefore 1 second, then 1/10 second, then 1/20, 1/50 and 1/300 second. In a second run, we will use the same values, but add a random value to the frequency of the envelope generator, thus avoiding regularities.

EXAMPLE 04F01_GranSynthIntro.csd

```
<CsoundSynthesizer>
<CsOptions>
-o dac -m128
</CsOptions>
<CsInstruments>
sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

giEnv ftgen 0, 0, 8192, 9, 1/2, 1, 0 ;half sine as envelope

instr EnvFreq
printf " Envelope frequency rising from %d to %d Hz\n", 1, p4, p5
gkEnvFreq expseg p4, 3, p4, 2, p5
endin
```

```

instr GrainGenSync
puts "\nSYNCHRONOUS GRANULAR SYNTHESIS", 1
aEnv poscil .2, gkEnvFreq, giEnv
aOsc poscil aEnv, 400
aOut linen aOsc, .1, p3, .5
out aOut, aOut
endin

instr GrainGenAsync
puts "\nA-SYNCHRONOUS GRANULAR SYNTHESIS", 1
aEnv poscil .2, gkEnvFreq+randomi:k(0,gkEnvFreq,gkEnvFreq), giEnv
aOsc poscil aEnv, 400
aOut linen aOsc, .1, p3, .5
out aOut, aOut
endin

</CsInstruments>
<CsScore>
i "GrainGenSync" 0 30
i "EnvFreq" 0 5 1 10
i . + . 10 20
i . + . 20 50
i . + . 50 100
i . + . 100 300
b 31
i "GrainGenAsync" 0 30
i "EnvFreq" 0 5 1 10
i . + . 10 20
i . + . 20 50
i . + . 50 100
i . + . 100 300
</CsScore>
</CsoundSynthesizer>
;example by joachim heintz

```

We hear different characteristics, due to the regular or irregular sequence of the grains. To understand what happens, we will go deeper in this matter and advance to a more flexible model for grain generation.

Concept Behind Granular Synthesis

Granular synthesis can in general be described as 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 repeat a fragment of sound with regularity, there are two principle attributes that we are most concerned with. Firstly the duration of each sound grain is significant: if the grain duration is 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 04F02_GranSynth_basic.csd

```
<CsoundSynthesizer>
<CsOptions>
-odac -m0
</CsOptions>
<CsInstruments>

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
    printk "Rate:%.2F  Dur:%.2F  Formant:%.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  oscil  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>
;example by Iain McCurdy

```

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 formatoire*) 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 04F03_Fof_vowels.csd

```

<CsoundSynthesizer>
<CsOptions>
-odac
</CsOptions>

```

```

<CsInstruments>
sr = 44100
ksmps = 16
nchnls = 2
0dbfs = 1

;FUNCTION TABLES STORING DATA FOR VARIOUS VOICE FORMANTS
;BASS
giBF1 ftgen 0, 0, -5, -2, 600, 400, 250, 400, 350
giBF2 ftgen 0, 0, -5, -2, 1040, 1620, 1750, 750, 600
giBF3 ftgen 0, 0, -5, -2, 2250, 2400, 2600, 2400, 2400
giBF4 ftgen 0, 0, -5, -2, 2450, 2800, 3050, 2600, 2675
giBF5 ftgen 0, 0, -5, -2, 2750, 3100, 3340, 2900, 2950

giBDb1 ftgen 0, 0, -5, -2, 0, 0, 0, 0, 0
giBDb2 ftgen 0, 0, -5, -2, -7, -12, -30, -11, -20
giBDb3 ftgen 0, 0, -5, -2, -9, -9, -16, -21, -32
giBDb4 ftgen 0, 0, -5, -2, -9, -12, -22, -20, -28
giBDb5 ftgen 0, 0, -5, -2, -20, -18, -28, -40, -36

giBBW1 ftgen 0, 0, -5, -2, 60, 40, 60, 40, 40
giBBW2 ftgen 0, 0, -5, -2, 70, 80, 90, 80, 80
giBBW3 ftgen 0, 0, -5, -2, 110, 100, 100, 100, 100
giBBW4 ftgen 0, 0, -5, -2, 120, 120, 120, 120, 120
giBBW5 ftgen 0, 0, -5, -2, 130, 120, 120, 120, 120

;TENOR
giTF1 ftgen 0, 0, -5, -2, 650, 400, 290, 400, 350
giTF2 ftgen 0, 0, -5, -2, 1080, 1700, 1870, 800, 600
giTF3 ftgen 0, 0, -5, -2, 2650, 2600, 2800, 2600, 2700
giTF4 ftgen 0, 0, -5, -2, 2900, 3200, 3250, 2800, 2900
giTF5 ftgen 0, 0, -5, -2, 3250, 3580, 3540, 3000, 3300

giTDb1 ftgen 0, 0, -5, -2, 0, 0, 0, 0, 0
giTDb2 ftgen 0, 0, -5, -2, -6, -14, -15, -10, -20
giTDb3 ftgen 0, 0, -5, -2, -7, -12, -18, -12, -17
giTDb4 ftgen 0, 0, -5, -2, -8, -14, -20, -12, -14
giTDb5 ftgen 0, 0, -5, -2, -22, -20, -30, -26, -26

giTBW1 ftgen 0, 0, -5, -2, 80, 70, 40, 40, 40
giTBW2 ftgen 0, 0, -5, -2, 90, 80, 90, 80, 60
giTBW3 ftgen 0, 0, -5, -2, 120, 100, 100, 100, 100
giTBW4 ftgen 0, 0, -5, -2, 130, 120, 120, 120, 120
giTBW5 ftgen 0, 0, -5, -2, 140, 120, 120, 120, 120

;COUNTER TENOR
giCTF1 ftgen 0, 0, -5, -2, 660, 440, 270, 430, 370
giCTF2 ftgen 0, 0, -5, -2, 1120, 1800, 1850, 820, 630
giCTF3 ftgen 0, 0, -5, -2, 2750, 2700, 2900, 2700, 2750
giCTF4 ftgen 0, 0, -5, -2, 3000, 3000, 3350, 3000, 3000
giCTF5 ftgen 0, 0, -5, -2, 3350, 3300, 3590, 3300, 3400

giTBDb1 ftgen 0, 0, -5, -2, 0, 0, 0, 0, 0
giTBDb2 ftgen 0, 0, -5, -2, -6, -14, -24, -10, -20
giTBDb3 ftgen 0, 0, -5, -2, -23, -18, -24, -26, -23
giTBDb4 ftgen 0, 0, -5, -2, -24, -20, -36, -22, -30
giTBDb5 ftgen 0, 0, -5, -2, -38, -20, -36, -34, -30

giTBW1 ftgen 0, 0, -5, -2, 80, 70, 40, 40, 40
giTBW2 ftgen 0, 0, -5, -2, 90, 80, 90, 80, 60
giTBW3 ftgen 0, 0, -5, -2, 120, 100, 100, 100, 100
giTBW4 ftgen 0, 0, -5, -2, 130, 120, 120, 120, 120
giTBW5 ftgen 0, 0, -5, -2, 140, 120, 120, 120, 120

```

```

;ALTO
giAF1 ftgen 0, 0, -5, -2, 800, 400, 350, 450, 325
giAF2 ftgen 0, 0, -5, -2, 1150, 1600, 1700, 800, 700
giAF3 ftgen 0, 0, -5, -2, 2800, 2700, 2700, 2830, 2530
giAF4 ftgen 0, 0, -5, -2, 3500, 3300, 3700, 3500, 2500
giAF5 ftgen 0, 0, -5, -2, 4950, 4950, 4950, 4950, 4950

giADb1 ftgen 0, 0, -5, -2, 0, 0, 0, 0, 0
giADb2 ftgen 0, 0, -5, -2, -4, -24, -20, -9, -12
giADb3 ftgen 0, 0, -5, -2, -20, -30, -30, -16, -30
giADb4 ftgen 0, 0, -5, -2, -36, -35, -36, -28, -40
giADb5 ftgen 0, 0, -5, -2, -60, -60, -60, -55, -64

giABW1 ftgen 0, 0, -5, -2, 50, 60, 50, 70, 50
giABW2 ftgen 0, 0, -5, -2, 60, 80, 100, 80, 60
giABW3 ftgen 0, 0, -5, -2, 170, 120, 120, 100, 170
giABW4 ftgen 0, 0, -5, -2, 180, 150, 150, 130, 180
giABW5 ftgen 0, 0, -5, -2, 200, 200, 200, 135, 200

;SOPRANO
giSF1 ftgen 0, 0, -5, -2, 800, 350, 270, 450, 325
giSF2 ftgen 0, 0, -5, -2, 1150, 2000, 2140, 800, 700
giSF3 ftgen 0, 0, -5, -2, 2900, 2800, 2950, 2830, 2700
giSF4 ftgen 0, 0, -5, -2, 3900, 3600, 3900, 3800, 3800
giSF5 ftgen 0, 0, -5, -2, 4950, 4950, 4950, 4950, 4950

giSDb1 ftgen 0, 0, -5, -2, 0, 0, 0, 0, 0
giSDb2 ftgen 0, 0, -5, -2, -6, -20, -12, -11, -16
giSDb3 ftgen 0, 0, -5, -2, -32, -15, -26, -22, -35
giSDb4 ftgen 0, 0, -5, -2, -20, -40, -26, -22, -40
giSDb5 ftgen 0, 0, -5, -2, -50, -56, -44, -50, -60

giSBW1 ftgen 0, 0, -5, -2, 80, 60, 60, 70, 50
giSBW2 ftgen 0, 0, -5, -2, 90, 90, 90, 80, 60
giSBW3 ftgen 0, 0, -5, -2, 120, 100, 100, 100, 170
giSBW4 ftgen 0, 0, -5, -2, 130, 150, 120, 130, 180
giSBW5 ftgen 0, 0, -5, -2, 140, 200, 120, 135, 200

gisine ftgen 0, 0, 4096, 10, 1
giexp ftgen 0, 0, 1024, 19, 0.5, 0.5, 270, 0.5

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

    ; read formant cutoff frequencies from tables
    kForm1  tablei   kVow*5,giBF1+(iVoice*15)
    kForm2  tablei   kVow*5,giBF1+(iVoice*15)+1
    kForm3  tablei   kVow*5,giBF1+(iVoice*15)+2
    kForm4  tablei   kVow*5,giBF1+(iVoice*15)+3
    kForm5  tablei   kVow*5,giBF1+(iVoice*15)+4
    ; read formant intensity values from tables
    kDB1    tablei   kVow*5,giBF1+(iVoice*15)+5
    kDB2    tablei   kVow*5,giBF1+(iVoice*15)+6
    kDB3    tablei   kVow*5,giBF1+(iVoice*15)+7
    kDB4    tablei   kVow*5,giBF1+(iVoice*15)+8
    kDB5    tablei   kVow*5,giBF1+(iVoice*15)+9
    ; read formant bandwidths from tables
    kBW1    tablei   kVow*5,giBF1+(iVoice*15)+10
    kBW2    tablei   kVow*5,giBF1+(iVoice*15)+11
    kBW3    tablei   kVow*5,giBF1+(iVoice*15)+12

```

```

kBW4      tablei    kVow*5,giBF1+(iVoice*15)+13
kBW5      tablei    kVow*5,giBF1+(iVoice*15)+14
; create resonant formants using fof opcode
koct      =         1
aForm1    fof       ampdb(kDB1),kFund,kForm1,0,kBW1,0.003,0.02,0.007,\ 
            1000,gisine,giexp,3600
aForm2    fof       ampdb(kDB2),kFund,kForm2,0,kBW2,0.003,0.02,0.007,\ 
            1000,gisine,giexp,3600
aForm3    fof       ampdb(kDB3),kFund,kForm3,0,kBW3,0.003,0.02,0.007,\ 
            1000,gisine,giexp,3600
aForm4    fof       ampdb(kDB4),kFund,kForm4,0,kBW4,0.003,0.02,0.007,\ 
            1000,gisine,giexp,3600
aForm5    fof       ampdb(kDB5),kFund,kForm5,0,kBW5,0.003,0.02,0.007,\ 
            1000,gisine,giexp,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>
; 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   0
i 1  8   .   78  77  1   0   1   0   0   1
i 1  16  .   150 118 0   1   1   0   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>
;example by Iain McCurdy

```

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 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 04F04_Asynchronous_GS.csd

```

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

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
;trigger a note (grain) in instr 3
            schedkwhen kTrig,0,0,3, abs(kDelay), kDur,kForm
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>
;example by Iain McCurdy
```

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 04F05_grain3.csd

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

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
    printk "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>
;example by Iain McCurdy

```

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 04F06_grain3_random.csd

```

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

```

```
    printk "Random Phase:%5.2FPitch Random:%5.2Fn",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>
;example by Iain McCurdy
```

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.

04 G. 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 can then produce sound because of how the system 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.

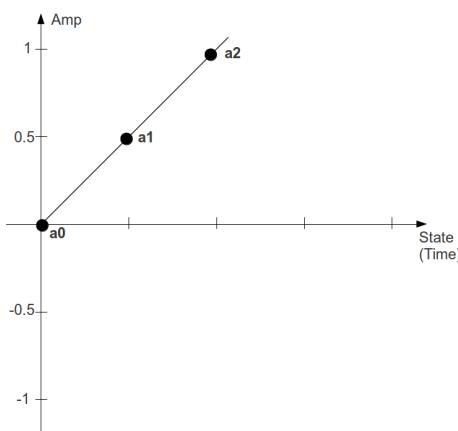
Compared with the complexity of a real-world physically dynamic system a physical model will most likely represent a brutal simplification. Nevertheless, using this technique will demand a lot of formulae, because physical models are described in terms of mathematics. Although designing a model may require 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

(The explanation here follows chapter 8.1.1 of Martin Neukom’s *Signale Systeme Klangsynthese*, Bern 2003)

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 a_0 is 0 and the second state a_1 is 0.5. Without the restricting force of the spring, the mass would continue moving unimpeded following a constant velocity:

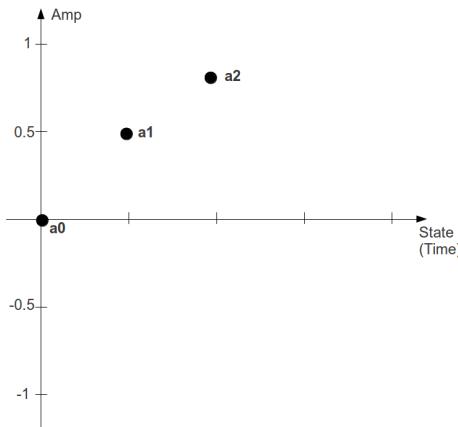


As the velocity between the first two states can be described as $a_1 - a_0$, the value of the third state a_2 will be:

$$a_2 = a_1 + (a_1 - a_0) = 0.5 + 0.5 = 1$$

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 a_1 . This damps the continuous movement of the mass so that for a factor of $c=0.4$ the next position will be:

$$a_2 = (a_1 + (a_1 - a_0)) - c * a_1 = 1 - 0.2 = 0.8$$



Csound can easily calculate the values by simply applying the formulae. For the first k-cycle¹, they are set via the `init` opcode. After calculating the new state, a_1 becomes a_0 and a_2 becomes a_1 for the next k-cycle. In the next csd the new values will be printed five times per second (the states are named here as $k0/k1/k2$ instead of $a0/a1/a2$, because k-rate values are needed for printing instead of audio samples).

EXAMPLE 04G01_Mass_spring_sine.csd

```
<CsoundSynthesizer>
<CsOptions>
-n ;no sound
</CsOptions>
<CsInstruments>
sr = 44100
ksmps = 8820 ;5 steps per second
```

¹See chapter 03A for more information about Csound's performance loops.

```

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
printfs "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

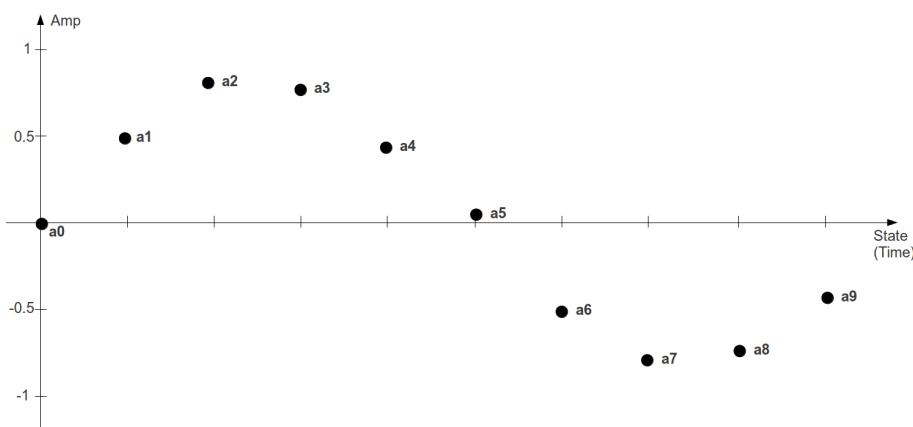
```

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_MS_sine_audible.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, either `ksmps` has to be set to 1, or a `setksmps` statement must be set in the instrument, with the same effect. 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_MS_variable_constant.csd

```

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

instr MassSpring
;set ksmps=1 in this instrument
setksmps 1
;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.

Implementing Simple Physical Systems

This text shows how to get oscillators and filters from simple physical models by recording the position of a point (mass) of a physical system. The behavior of a particle (mass on a spring, mass of a pendulum, etc.) is described by its position, velocity and acceleration. The mathematical equations, which describe the movement of such a point, are *differential equations*. In what follows, we describe how to derive time discrete system equations (also called difference equations) from physical models (described by differential equations). At every time step we first calculate the acceleration of a mass and then its new velocity and position. This procedure is called Euler's method and yields good results for low frequencies compared to the sampling rate (better approximations are achieved with the improved Euler's method or the Runge–Kutta methods).

Integrating the Trajectory of a Point

Velocity v is the difference of positions x per time unit T , acceleration a the difference of velocities v per time unit T :

$$\begin{aligned} v_t &= (x_t - x_{t-1})/T \\ a_t &= (v_t - v_{t-1})/T \end{aligned}$$

We get for $T = 1$

$$\begin{aligned} v_t &= x_t - x_{t-1} \\ a_t &= v_t - v_{t-1} \end{aligned}$$

If we know the position and velocity of a point at time $t-1$ and are able to calculate its acceleration at time t we can calculate the velocity v_t and the position x_t at time t :

$$\begin{aligned} v_t &= v_{t-1} + a_t \text{ and} \\ x_t &= x_{t-1} + v_t \end{aligned}$$

With the following algorithm we calculate a sequence of successive positions x :

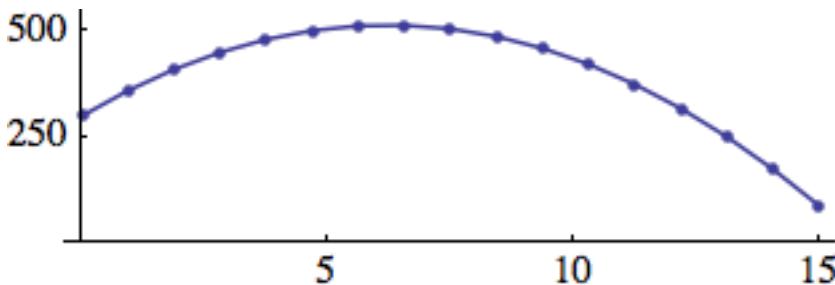
```

1. init x and v
2. calculate a
3. v += a      ; v = v + a
4. x += v      ; x = x + v

```

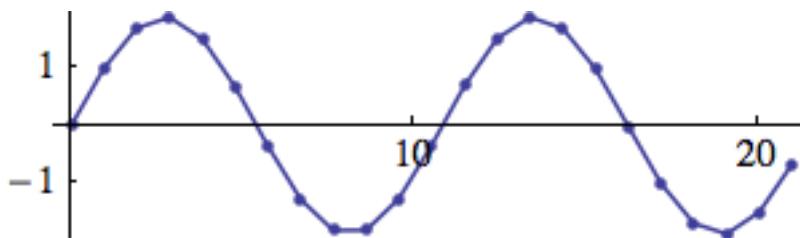
Example 1: The acceleration of gravity is constant ($g = -9.81\text{ms}^{-2}$). For a mass with initial position $x = 300\text{m}$ (above ground) and velocity $v = 70\text{ms}^{-1}$ (upwards) we get the following trajectory (path)

```
g = -9.81; x = 300; v = 70; Table[v += g; x += v, {16}];
```



Example 2: The acceleration a of a mass on a spring is proportional (with factor $-c$) to its position (deflection) x .

```
x = 0; v = 1; c = .3; Table[a = -c*x; v += a; x += v, {22}];
```



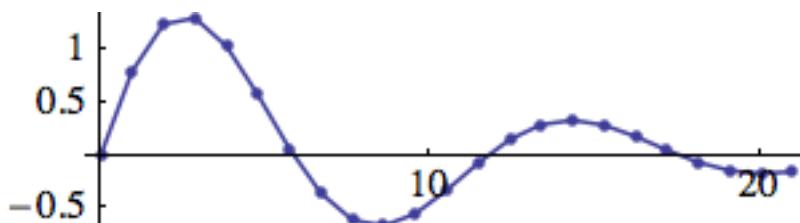
Introducing damping

Since damping is proportional to the velocity we reduce velocity at every time step by a certain amount d :

```
v *= (1 - d)
```

Example 3: Spring with damping (see *lin_reson.csd* below):

```
d = 0.2; c = .3; x = 0; v = 1;
Table[a = -c*x; v += a; v *= (1 - d); x += v, {22}];
```



The factor c can be calculated from the frequency f :

$$c = 2 - \sqrt{4 - d^2} \cos(2\pi f / sr)$$

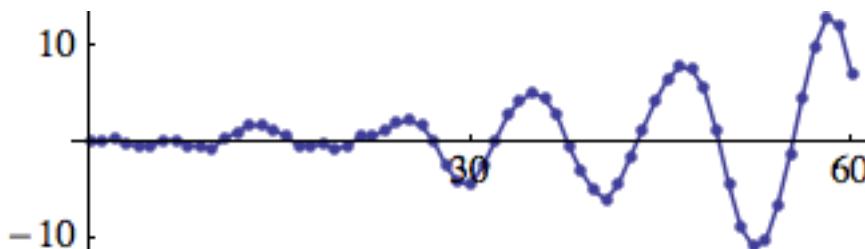
Introducing excitation

In the examples 2 and 3 the systems oscillate because of their initial velocity $v = 1$. The resultant oscillation is the impulse response of the systems. We can excite the systems continuously by adding a value exc to the velocity at every time step.

```
v += exc;
```

Example 4: Damped spring with random excitation (resonator with noise as input)

```
d = .01; s = 0; v = 0;
Table[a = -.3*s; v += a; v += RandomReal[{-1, 1}];
v *= (1 - d); s += v, {61}];
```



EXAMPLE 04G04_lin_reson.csd

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

opcode lin_reson, a, akk
setksmps 1
avel init 0 ;velocity
ax init 0 ;deflection x
ain,kf,kdamp xin
kc = 2-sqrt(4-kdamp^2)*cos(kf*2*$M_PI/sr)
aacel = -kc*ax
avel = avel+aaccel+ain
avel = avel*(1-kdamp)
ax = ax+avel
xout ax
endop

instr 1
aexc rand p4
aout lin_reson aexc,p5,p6
out aout
endin

</CsInstruments>
<CsScore>
; p4 excitaion p5 freq p6 damping
; .0001 440 .0001
i1 0 5

```

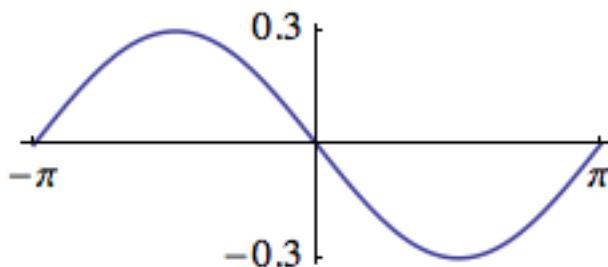
```
</CsScore>
</CsoundSynthesizer>
;example by martin neukom
```

Introducing nonlinear acceleration

Example 5: The acceleration of a pendulum depends on its deflection (angle x).

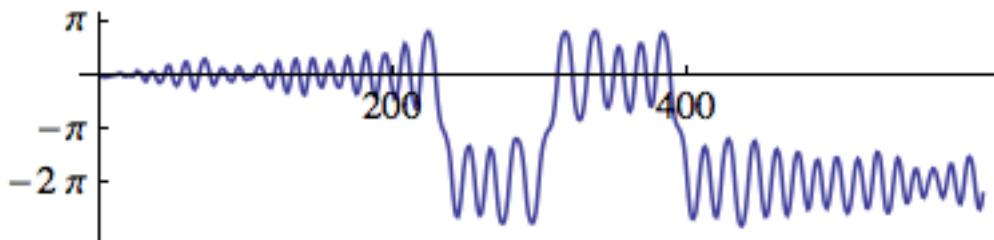
```
a = c*sin(x)
```

This figure shows the function $-0.3\sin(x)$



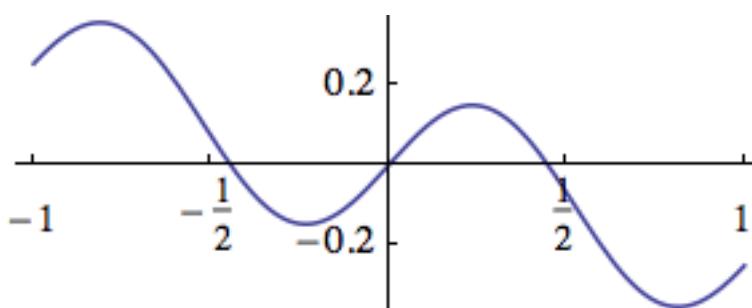
The following trajectory shows that the frequency decreases with increasing amplitude and that the pendulum can turn around.

```
d = .003; s = 0; v = 0;
Table[a = f[s]; v += a; v += RandomReal[{-0.09, .1}]; v *= (1 - d);
s += v, {400}];
```

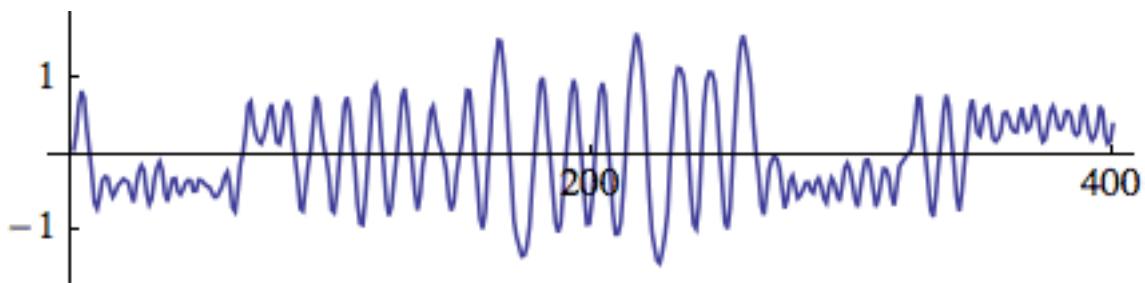


We can implement systems with accelerations that are arbitrary functions of position x .

Example 6: $a = f(x) = -c_1x + c_2\sin(c_3x)$



```
d = .03; x = 0; v = 0; Table[a = f[x]; v += a;
v += RandomReal[{-0.1, .1}]; v *= (1 - d); x += v, {400}];
```

**EXAMPLE 04G05_nonlin_reson.csd**

```

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

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

; simple damped nonlinear resonator
opcode nonlin_reson, a, akki
setksmps 1
avel init 0 ;velocity
adef init 0 ;deflection
ain,kc,kdamp,ifn xin
aacel tablei adef, ifn, 1, .5 ;acceleration = -c1*f1(def)
aacel = -kc*aacel
avel = avel+aacel+ain ;vel += acel + excitation
avel = avel*(1-kdamp)
adef = adef+avel
xout adef
endop

instr 1
kenv oscil p4,.5,1
aexc rand kenv
aout nonlin_reson aexc,p5,p6,p7
out aout
endin

</CsInstruments>
<CsScore>
f1 0 1024 10 1
f2 0 1024 7 -1 510 .15 4 -.15 510 1
f3 0 1024 7 -1 350 .1 100 -.3 100 .2 100 -.1 354 1
; p4 p5 p6 p7
; excitation c1 damping ifn
i1 0 20 .0001 .01 .00001 3
;i1 0 20 .0001 .01 .00001 2
</CsScore>
</CsoundSynthesizer>
;example by martin neukom

```

The Van der Pol Oscillator

While attempting to explain the nonlinear dynamics of vacuum tube circuits, the Dutch electrical engineer Balthasar van der Pol derived the differential equation

$$\frac{d^2x}{dt^2} = -\omega^2x + \mu(1-x^2)\frac{dx}{dt}$$

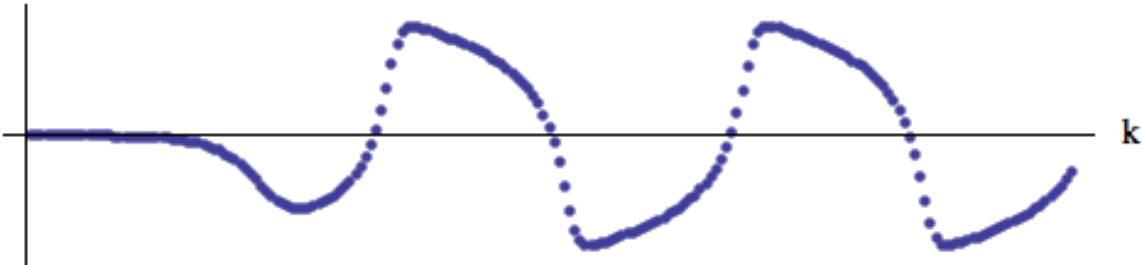
(where d^2x/dt^2 = acceleration and dx/dt = velocity)

The equation describes a linear oscillator $d^2x/dt^2 = -\omega^2x$ with an additional nonlinear term $\mu(1-x^2)dx/dt$. When $|x| > 1$, the nonlinear term results in damping, but when $|x| < 1$, negative damping results, which means that energy is introduced into the system.

Such oscillators compensating for energy loss by an inner energy source are called *self-sustained oscillators*.

```
v = 0; x = .001; ω = 0.1; μ = 0.25;
snd = Table[v += (-ω^2*x + μ*(1 - x^2)*v); x += v, {200}];
```

x[k]



The constant ω is the angular frequency of the linear oscillator ($\mu = 0$). For a simulation with sampling rate sr we calculate the frequency f in Hz as

$$f = \omega \cdot sr / 2\pi$$

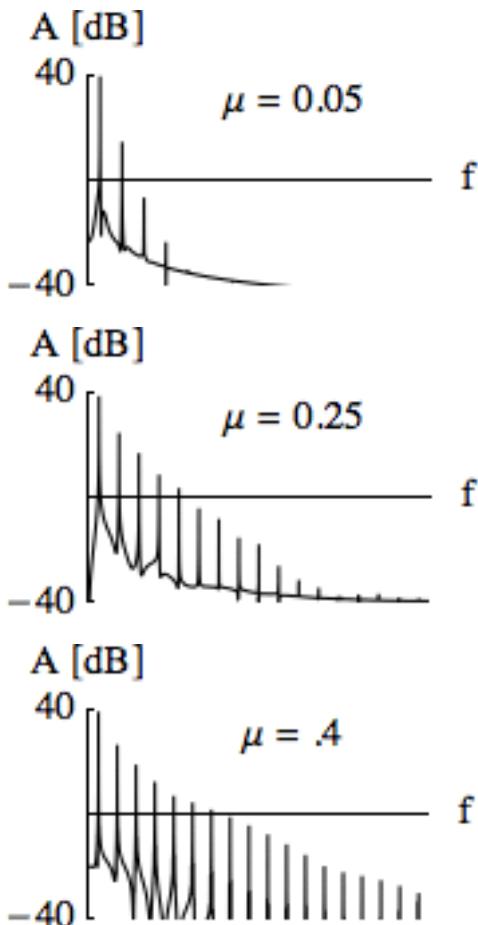
Since the simulation is only an approximation of the oscillation this formula gives good results only for low frequencies. The exact frequency of the simulation is

$$f = \arccos(1 - \omega^2/2) \cdot sr / 2\pi$$

We get ω^2 from frequency f as

$$2 - 2\cos(f \cdot 2\pi / sr)$$

With increasing μ the oscillations nonlinearity becomes stronger and more overtones arise (and at the same time the frequency becomes lower). The following figure shows the spectrum of the oscillation for various values of μ .



Certain oscillators can be synchronized either by an external force or by mutual influence. Examples of synchronization by an external force are the control of cardiac activity by a pace maker and the adjusting of a clock by radio signals. An example for the mutual synchronization of oscillating systems is the coordinated clapping of an audience. These systems have in common that they are not linear and that they oscillate without external excitation (*self-sustained oscillators*).

The UDO *v_d_p* represents a Van der Pol oscillator with a natural frequency *kfr* and a nonlinearity factor *kmu*. It can be excited by a sine wave of frequency *kfex* and amplitude *kaex*. The range of frequency within which the oscillator is synchronized to the exciting frequency increases as *kmu* and *kaex* increase.

EXAMPLE 04G06_van_der_pol.csd

```
<CsoundSynthesizer>
<CsOptions>
-odac
```

```

</CsOptions>
<CsInstruments>

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

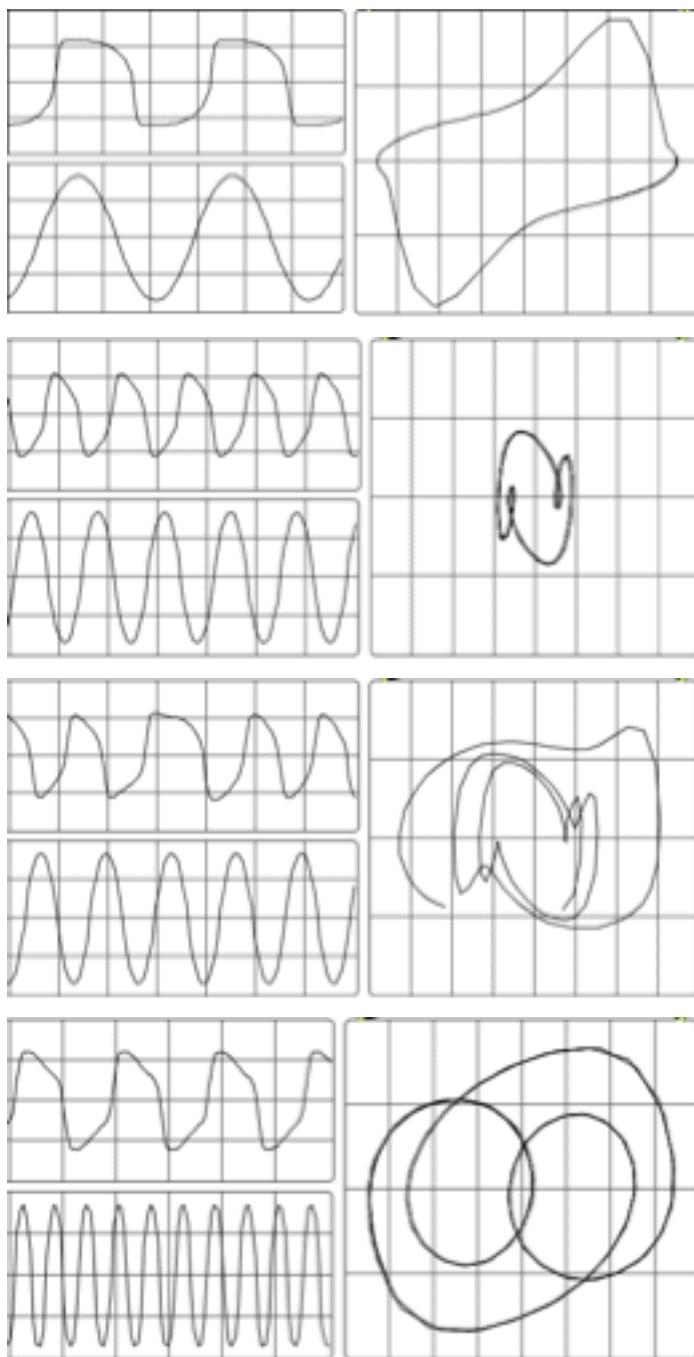
;Van der Pol Oscillator ;outputs a nonlinear oscillation
;inputs: a_excitation, k_frequency in Hz (of the linear part),
;nonlinearity (0 < mu < ca. 0.7)
opcode v_d_p, a, akk
setksmps 1
av init 0
ax init 0
ain,kfr,kmu xin
kc = 2-2*cos(kfr*2*$M_PI/sr)
aa = -kc*ax + kmu*(1-ax*ax)*av
av = av + aa
ax = ax + av + ain
xout ax
endop

instr 1
kaex = .001
kfex = 830
kamp = .15
kf = 455
kmu linseg 0,p3,.7
a1 poscil kaex,kfex
aout v_d_p a1,kf,kmu
out kamp*aout,a1*100
endin

</CsInstruments>
<CsScore>
i1 0 20
</CsScore>
</CsoundSynthesizer>
;example by martin neukom, adapted by joachim heintz

```

The variation of the phase difference between excitation and oscillation, as well as the transitions between synchronous, beating and asynchronous behaviors, can be visualized by showing the sum of the excitation and the oscillation signals in a phase diagram. The following figures show to the upper left the waveform of the Van der Pol oscillator, to the lower left that of the excitation (normalized) and to the right the phase diagram of their sum. For these figures, the same values were always used for kfr , kmu and $kaex$. Comparing the first two figures, one sees that the oscillator adopts the exciting frequency $kfex$ within a large frequency range. When the frequency is low (figure a), the phases of the two waves are nearly the same. Hence there is a large deflection along the x-axis in the phase diagram showing the sum of the waveforms. When the frequency is high, the phases are nearly inverted (figure b) and the phase diagram shows only a small deflection. The figure c shows the transition to asynchronous behavior. If the proportion between the natural frequency of the oscillator kfr and the excitation frequency $kfex$ is approximately simple ($kfex/kfr \cong m/n$), then within a certain range the frequency of the Van der Pol oscillator is synchronized so that $kfex/kfr = m/n$. Here one speaks of higher order synchronization (figure d).



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 04G07_KarplusStrong.csd

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

opcode KS, 0, ii
;performs the karplus-strong algorithm
iTb, iTbSiz xin
;calculate the mean of the last two values
iUlt    tab_i    iTbSiz-1, iTb
iPenUlt tab_i    iTbSiz-2, iTb
iNewVal =        (iUlt + iPenUlt) / 2
;shift values one position to the right
indx    =        iTbSiz-2
loop:
iVal    tab_i    indx, iTb
tabw_i  iVal, indx+1, iTb
loop_ge indx, 1, 0, loop
;fill the new value at the beginning of the table
tabw_i  iNewVal, 0, iTb
endop

opcode PrintTab, 0, iis
;prints table content, with a starting string
iTb, iTbSiz, Sin xin
indx    =      0
Sout    strcpy   Sin
loop:
iVal    tab_i    indx, iTb
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
iTb     ftgen    0, 0, -5, -2, 1, -1, 1, 1, -1
iTbLen  tableng  iTb

```

```

;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>
;example by joachim heintz

```

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 04G08_Plucked.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

```

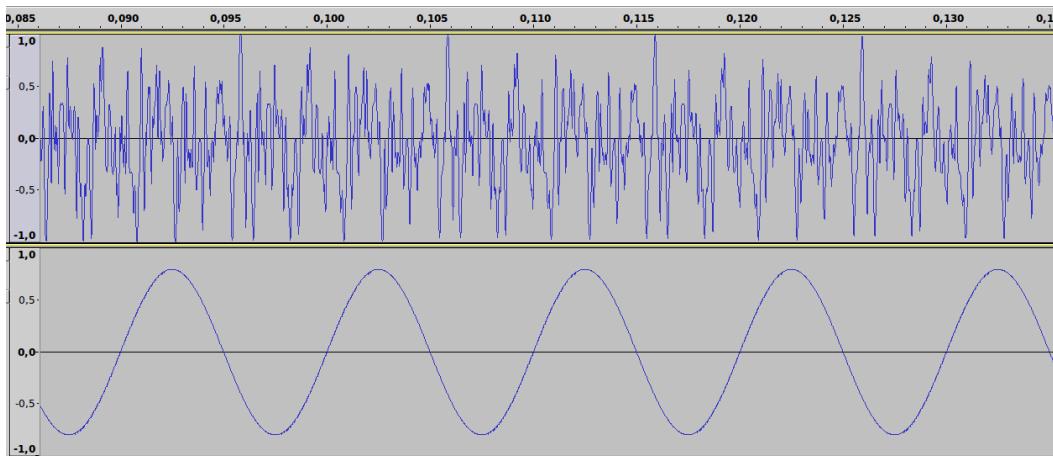
```

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 Opcodes for Physical Modelling

Csound 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 is 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 ability 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 04G09_wgbow.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>
;example by Iain McCurdy

```

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 04G10_wgbow_enhanced.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>
;example by Iain McCurdy

```

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 necessarily 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 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 04G11_barmodel.csd

```
<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

```

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 04G12_PhiSEM.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.

04 H. SCANNED SYNTHESIS

Scanned Synthesis is a relatively new synthesis technique invented by Max Mathews, Rob Shaw and Bill Verplank at Interval Research in 2000. This algorithm uses a combination of a table-lookup oscillator and Issac Newton's mechanical model (equation) of a mass and spring system to dynamically change the values stored in an f-table. The sonic result is a timbral spectrum that changes with time.

Csound has a couple opcodes dedicated to scanned synthesis, and these opcodes can be used not only to make sounds, but also to generate dynamic f-tables for use with other Csound opcodes.

A Quick Scanned Synth

The quickest way to start using scanned synthesis is Matt Gilliard's opcode `scantable`.

```
a1 scantable kamp, kfrq, ipos, imass, istiff, idamp, ivel
```

The arguments *kamp* and *kfrq* should be familiar, amplitude and frequency respectively. The other arguments are f-table numbers containing data known in the scanned synthesis world as **profiles**.

Profiles

Profiles refer to variables in the mass and spring equation. Newton's model describes a string as a finite series of marbles connected to each other with springs.

In this example we will use 128 marbles in our system. To the Csound user, profiles are a series of f-tables that set up the `scantable` opcode. To the opcode, these f-tables influence the dynamic behavior of the table read by a table-lookup oscillator.

```
gipos ftgen 1, 0, 128, 10, 1 ;Position Initial Shape: Sine wave range -1 to 1  
gimass ftgen 2, 0, 128, -7, 1, 128, 1 ;Masses: Constant value 1  
gistiff ftgen 3, 0, 128, -7, 0, 64, 100, 64, 0 ;Stiffness: triangle  
gidamp ftgen 4, 0, 128, -7, 1, 128, 1 ;Damping: Constant value 1  
givel ftgen 5, 0, 128, -2, 0 ;Velocity: Initially constant value 0
```

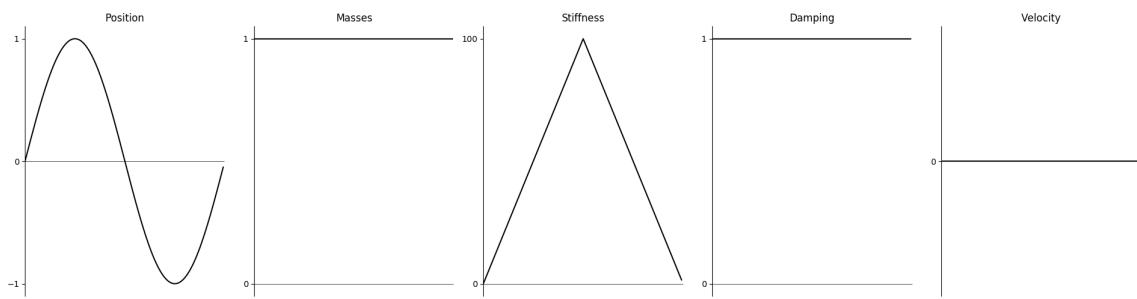


Figure 34.1: Initial function table profiles

All these tables need to be the same size; otherwise Csound will return an error.

Run the following .csd. Notice that the sound starts off sounding like our initial shape (a sine wave) but evolves as if there are filters, distortions or LFO's.

EXAMPLE 04H01_scantable_1.csd

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

gipos ftgen 1, 0, 128, 10, 1 ;position of the masses (initially: sine)
gimass ftgen 2, 0, 128, -7, 1, 128, 1 ;masses: constant value 1
gistiff ftgen 3, 0, 128, -7, 0, 64, 100, 64, 0 ;stiffness; triangle 0->100->0
gidamp ftgen 4, 0, 128, -7, 1, 128, 1 ;damping; constant value 1
givel ftgen 5, 0, 128, -2, 0 ;velocity; initially 0

instr 1
iamp = .2
ifrq = 440
aScan scantable iamp, ifrq, gipos, gimass, gistiff, gidamp, givel
aOut linen aScan, 1, p3, 1
out aOut, aOut
endin

</CsInstruments>
<CsScore>
i 1 0 19
</CsScore>
</CsoundSynthesizer>
;example by Christopher Saunders and joachim heintz
```

What happens in the *scantable* synthesis, is a constant change in the position (table *gipos*) and the velocity (table *givel*) of the mass particles. Here are three snapshots of these tables in the examples above:

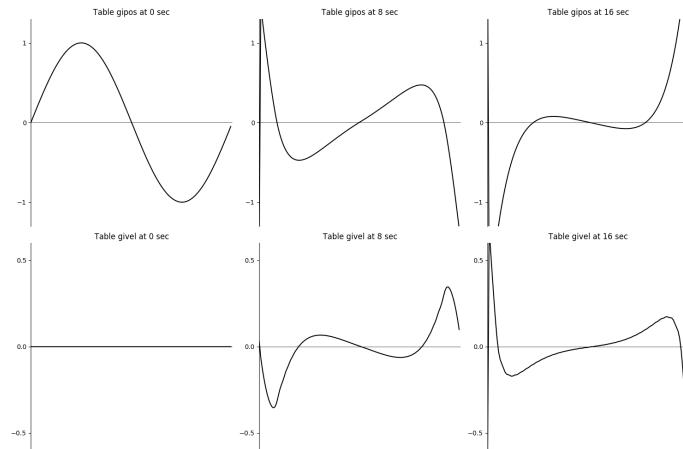


Figure 34.2: Position and Velocity tables at 0, 8, 16 seconds

The audio output of `scantable` is the result of oscillating through the `gipos` table. So we will achieve the same audible result with this code:

EXAMPLE 04H02_scantable_2.csd

```

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

gipos ftgen 1, 0, 128, 10, 1 ;position of the masses (initially: sine)
gimass ftgen 2, 0, 128, -7, 1, 128, 1 ;masses: constant value 1
gistiff ftgen 3, 0, 128, -7, 0, 64, 100, 64, 0 ;stiffness; triangle 0->100->0
gidamp ftgen 4, 0, 128, -7, 1, 128, 1 ;damping; constant value 1
givel ftgen 5, 0, 128, -2, 0 ;velocity; initially 0

instr 1
iamp = .2
ifrq = 440
a0 scantable 0, 0, gipos, gimass, gistiff, gidamp, givel
aScan poscil iamp, ifrq, gipos
aOut linen aScan, 1, p3, 1
out aOut, aOut
endin

</CsInstruments>
<CsScore>
i 1 0 19
</CsScore>
</CsoundSynthesizer>
;example by Christopher Saunders and joachim heintz

```

Dynamic Tables

We can use table which is changed by *scantable* dynamically for any context. Below is an example of using the values of an f-table generated by *scantable* to modify the amplitudes of an fsig, a signal type in csound which represents a spectral signal.

EXAMPLE 04H03_Scantable_pvsmaska.csd

```

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

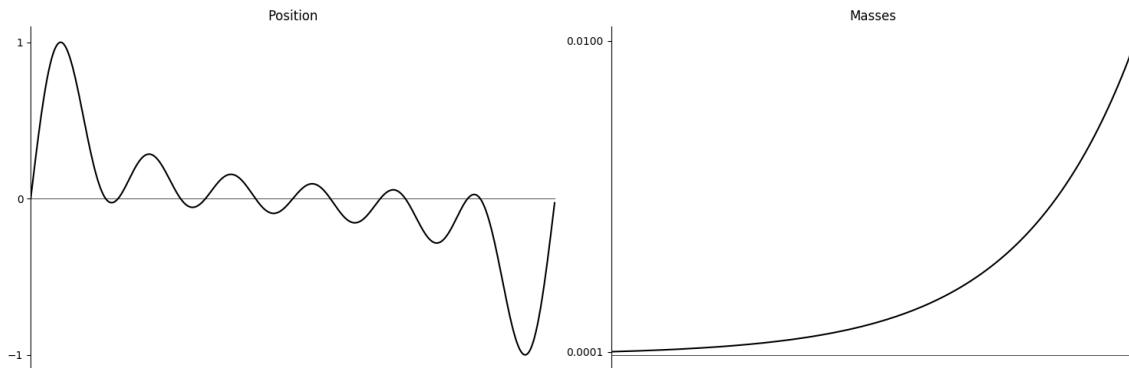
gipos ftgen 0,0,128,10,1,1,1,1,1,1 ;Initial Position Shape: impulse-like
gimass ftgen 0,0,128,-5,0.0001,128,.01 ;Masses: exponential 0.0001 to 0.01
gistiff ftgen 0,0,128,-7,0,64,100,64,0 ;Stiffness; triangle range 0 to 100
gidamp ftgen 0,0,128,-7,1,128,1 ;Damping; constant value 1
givel ftgen 0,0,128,-7,0,128,0 ;Initial Velocity; constant value 0
gisin ftgen 0,0,8192,10,1 ;Sine wave for buzz opcode

instr 1
iamp      =      .2
kfrq      =      110
aBuzz     buzz    iamp, kfrq, 32, gisin
aBuzz     linen   aBuzz, .1, p3, 1
          out     aBuzz, aBuzz
endin
instr 2
iamp      =      .4
kfrq      =      110
a0        scantable 0, 0, gipos, gimass, gistiff, gidamp, givel
ifftsize  =      128
overlap   =      ifftsize / 4
iwinsize  =      ifftsize
iwinshape =      1; von-Hann window
aBuzz     buzz    iamp, kfrq, 32, gisin
fBuzz     pvsanal aBuzz, ifftsize, overlap, iwinsize, iwinshape ;fft
fMask     pvsmaska fBuzz, gipos, 1
aOut      pvsynth  fMask; resynthesize
aOut      linen   aOut, .1, p3, 1
          out     aOut, aOut
endin
</CsInstruments>
<CsScore>
i 1 0 3
i 2 4 20
</CsScore>
</CsoundSynthesizer>
;Example by Christopher Saunders and joachim heintz

```

In this .csd, the score plays instrument 1, a normal buzz sound, and then the score plays instrument 2 – the same buzz sound re-synthesized with amplitudes of each of the 128 frequency bands,

controlled by a dynamic function table which is generated by `scantable`. Compared to the first example, two tables have been changed. The initial positions are an impulse-like wave form, and the masses are between 1/10000 and 1/10 in exponential rise.



A More Flexible Scanned Synth

`Scantable` can do a lot for us, it can synthesize an interesting, time-varying timbre using a table lookup oscillator, or animate an f-table for use in other Csound opcodes. However, there are other scanned synthesis opcodes that can take our expressive use of the algorithm even further.

The opcodes `scans` and `scanu` by Paris Smaragdis give the Csound user one of the most robust and flexible scanned synthesis environments. These opcodes work in tandem to first set up the dynamic wavetable, and then to *scan* the dynamic table in ways a table-lookup oscillator cannot.

`Scanu` takes 18 arguments and sets a table into motion.

```
scanu ipos, irate, ifnvel, ifnmass, ifnstif, ifncentr, ifndamp, kmass,
      kstif, kcentr, kdamp, ileft, iringht, kpos, kstrngth, ain, idisp, id
```

For a detailed description of what each argument does, see the [Csound Reference Manual](#); I will discuss the various types of arguments in the opcode.

The first set of arguments - `ipos`, `ifnvel`, `ifnmass`, `ifnstiff`, `ifncenter`, and `ifndamp` - are f-tables describing the profiles, similar to the profile arguments for `scantable`. Like for `scantable`, the same size is required for each of these tables.

An exception to this size requirement is the `ifnstiff` table. This table is the size of the other profiles squared. If the other f-tables are size 128, then `ifnstiff` should be of size 16384 (or 128*128). To discuss what this table does, I must first introduce the concept of a scanned matrix.

The Scanned Matrix

The scanned matrix is a convention designed to describe the shape of the connections of masses in the mass and spring model.

Going back to our discussion on Newton's mechanical model, the mass and spring model describes the behavior of a string as a finite number of masses connected by springs. As you can

imagine, the masses are connected sequentially, one to another, like beads on a string. Mass #1 is connected to #2, #2 connected to #3 and so on. However, the pioneers of scanned synthesis had the idea to connect the masses in a non-linear way. It's hard to imagine, because as musicians, we have experience with piano or violin strings (one dimensional strings), but not with multi-dimensional strings. Fortunately, the computer has no problem working with this idea, and the flexibility of Newton's equation allows us to use the CPU to model mass #1 being connected with springs not only to #2 but also to #3 and any other mass in the model.

The most direct and useful implementation of this concept is to connect mass #1 to mass #2 and mass #128 – forming a string without endpoints, a circular string, like tying our string with beads to make a necklace. The pioneers of scanned synthesis discovered that this circular string model is more useful than a conventional one-dimensional string model with endpoints. In fact, [scantable](#) uses a circular string.

The matrix is described in a simple ASCII file, imported into Csound via a GEN23 generated f-table.

```
f3 0 16384 -23 "string-128"
```

This text file **must** be located in the same directory as your .csd or csound will give you this error
ftable 3: error opening ASCII file

You can construct your own matrix using Stephen Yi's Scanned Matrix editor included in the Blue frontend for Csound.

To swap out matrices, simply type the name of a different matrix file into the double quotes, i.e.:

```
f3 0 16384 -23 "circularstring_2-128"
```

Different matrices have unique effects on the behavior of the system. Some matrices can make the synth extremely loud, others extremely quiet. Experiment with using different matrices.

Now would be a good time to point out that Csound has other scanned synthesis opcodes preceded with an x, [xscans](#), [xscanu](#), that use a different matrix format than the one used by [scans](#), [scanu](#), and Stephen Yi's Scanned Matrix Editor. The Csound Reference Manual has more information on this.

The Hammer

If the initial shape, an f-table specified by the *ipos* argument determines the shape of the initial contents in our dynamic table. What if we want to “reset” or “pluck” the table, perhaps with a shape of a square wave instead of a sine wave, while the instrument is playing?

With [scantable](#), there is an easy way to do this, send a score event changing the contents of the dynamic f-table. You can do this with the Csound score by adjusting the start time of the f-events in the score.

EXAMPLE 04H04_Hammer.csd

```
<CsoundSynthesizer>
<CsOptions>
-o dac
```

```

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

instr 1
ipos      ftgen    1, 0, 128, 10, 1 ; Initial Shape, sine
imass     ftgen    2, 0, 128, -7, 1, 128, 1 ;Masses(adj.), constant value 1
istiff    ftgen    3, 0, 128, -7, 0, 64, 100, 64, 0 ;Stiffness triangle
idamp     ftgen    4, 0, 128, -7, 1, 128, 1; ;Damping; constant value 1
ivel      ftgen    5, 0, 128, -7, 0, 128, 0 ;Initial Velocity 0
iamp      =         0.2
a1        scantable iamp, 60, ipos, imass, istiff, idamp, ivel
outs      a1, a1
endin
</CsInstruments>
<CsScore>
i 1 0 14
f 1 1 128 10 1 1 1 1 1 1 1 1 1 1 1 1
f 1 2 128 10 1 1 0 0 0 0 0 0 0 0 0 1 1
f 1 3 128 10 1 1 1 1 1 1
f 1 4 128 10 1 0 0 0 0 0 0 0 0 0 0 0 0 1
f 1 5 128 10 1 1
f 1 6 128 13 1 1 0 0 0 -.1 0 .3 0 -.5 0 .7 0 -.9 0 1 0 -1 0
f 1 7 128 21 6 5.745
</CsScore>
</CsoundSynthesizer>
;example by Christopher Saunders

```

You'll get the warning

```
WARNING: replacing previous ftable 1
```

which means this method of hammering the string is working. In fact you could use this method to explore and hammer every possible GEN routine in Csound. [GEN10](#) (sines), [GEN 21](#) (noise) and [GEN 27](#) (breakpoint functions) could keep you occupied for a while.

Unipolar waves have a different sound but a loss in volume can occur. There is a way to do this with [scanu](#), but I do not use this feature and just use these values instead.

```
ileft = 0.  iright = 1.  kpos = 0.  kstrngth = 0.
```

More on Profiles

One of the biggest challenges in understanding scanned synthesis is the concept of profiles.

Setting up the opcode [scanu](#) requires 3 profiles - Centering, Mass and Damping. The pioneers of scanned synthesis discovered early on that the resultant timbre is far more interesting if marble #1 had a different centering force than mass #64.

The farther our model gets away from a physical real-world string that we know and pluck on our guitars and pianos, the more interesting the sounds for synthesis. Therefore, instead of one mass, and damping, and centering value for all 128 of the marbles each marble can have its own

conditions. How the centering, mass, and damping profiles make the system behave is up to the user to discover through experimentation (more on how to experiment safely later in this chapter).

Control Rate Profile Scalars

Profiles are a detailed way to control the behavior of the string, but what if we want to influence the mass or centering or damping of every marble **after** a note has been activated and while its playing?

`Scanu` gives us 4 k-rate arguments `kmass`, `kstif`, `kcentr`, `kdamp`, to scale these forces. One could scale mass to volume, or have an envelope controlling centering.

Caution! These parameters can make the scanned system unstable in ways that could make **extremely** loud sounds come out of your computer. It is best to experiment with small changes in range and keep your headphones off. A good place to start experimenting is with different values for `kcentr` while keeping `kmass`, `kstiff`, and `kdamp` constant. You could also scale mass and stiffness to MIDI velocity.

Audio Injection

Instead of using the hammer method to move the marbles around, we could use audio to add motion to the mass and spring model. `Scanu` lets us do this with a simple audio rate argument. Be careful with the amplitude again.

To bypass audio injection all together, simply assign 0 to an a-rate variable.

```
ain = 0
```

and use this variable as the argument.

Connecting to Scans

The p-field `id` is an arbitrary integer label that tells the scans opcode which `scanu` to read. By making the value of `id` negative, the arbitrary numerical label becomes the number of an f-table that can be used by any other opcode in Csound, like we did with `scantable` earlier in this chapter.

We could then use `poscil` to perform a table lookup algorithm to make sound out of `scanu` (as long as `id` is negative), but `scanu` has a companion opcode, `scans` which has 1 more argument than `oscil`. This argument is the number of an f-table containing the scan trajectory.

Scan Trajectories

One thing we have taken for granted so far with `poscil` is that the wave table is read front to back. If you regard `poscil` as a phasor and table pair, the first index of the table is always read first and the last index is always read last as in the example below:

EXAMPLE 04H05_Scan_trajectories.csd

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

instr 1
andx phasor 440
a1 table andx*8192, 1
outs a1*.2, a1*.2
endin
</CsInstruments>
<CsScore>
f1 0 8192 10 1
i 1 0 4
</CsScore>
</CsoundSynthesizer>
;example by Christopher Saunders
```

But what if we wanted to read the table indices back to front, or even “out of order”? Well we could do something like this:

EXAMPLE 04H06_Scan_trajectories2.csd

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

instr 1
andx phasor 440
andx table andx*8192, 2 ; read the table out of order!
aOut table andx*8192, 1
outs aOut*.2, aOut*.2
endin
</CsInstruments>
<CsScore>
f1 0 8192 10 1
f2 0 8192 -5 .001 8192 1;
```

```
i 1 0 4
</CsScore>
</CsoundSynthesizer>
;example by Christopher Saunders
```

We are still dealing with 1-dimensional arrays, or f-tables as we know them. But if we remember back to our conversation about the scanned matrix, matrices are multi-dimensional.

The opcode `scans` gives us the flexibility of specifying a scan trajectory, analogous to telling the phasor/table combination to read values non-consecutively. We could read these values, not left to right, but in a spiral order, by specifying a table to be the *ifntraj* argument of `scans`.

```
a3 scans iamp, kpch, ifntraj ,id , interp
```

An f-table for the spiral method can generated by reading the ASCII file *spiral-8,16,128,2,1over2* by GEN23

```
f2 0 128 -23 "spiral-8,16,128,2,1over2"
```

The following .csd requires that the files *circularstring-128* and *spiral-8,16,128,2,1over2* be located in the same directory as the .csd.

EXAMPLE 04H07_Scan_matrices.csd

```
<CsoundSynthesizer>
<CsOptions>
-o dac
</CsOptions>
<CsInstruments>
nchnls = 2
sr = 44100
ksmps = 32
0dbfs = 1
instr 1
ipos ftgen 1, 0, 128, 10, 1
irate = .005
ifnvel ftgen 6, 0, 128, -7, 0, 128, 0
ifnmass ftgen 2, 0, 128, -7, 1, 128, 1
ifnstif ftgen 3, 0, 16384,-23,"circularstring-128"
ifncentr ftgen 4, 0, 128, -7, 0, 128, 2
ifndamp ftgen 5, 0, 128, -7, 1, 128, 1
imass = 2
istif = 1.1
icentr = .1
idamp = -0.01
ileft = 0.
iright = .5
ipos = 0.
istrngth = 0.
ain = 0
idisp = 0
id = 8
scanu 1, irate, ifnvel, ifnmass, ifnstif, ifncentr, ifndamp, imass, istif,
      icentr, idamp, ileft, iright, ipos, istrngth, ain, idisp, id
scanu 1,.007,6,2,3,4,5, 2, 1.10 ,.10 ,0 ,.1 ,.5, 0, 0,ain,1,2;
iamp = .2
ifreq = 200
a1 scans iamp, ifreq, 7, id
outs a1, a1
endin
```

```

</CsInstruments>
<CsScore>
f7 0 128 -7 0 128 128
i 1 0 5
f7 5 128 -23 "spiral-8,16,128,2,1over2"
i 1 5 5
f7 10 128 -7 127 64 0 64 127
i 1 10 5
</CsScore>
</CsoundSynthesizer>
;example by Christopher Saunders

```

Notice that the scan trajectory has an FM-like effect on the sound. These are the three different *f7* tables which are started in the score:

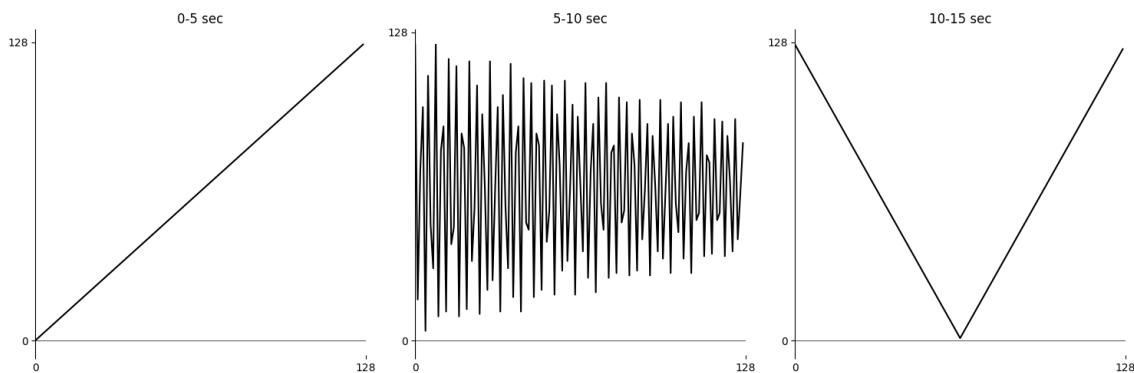


Table Size and Interpolation

Tables used for scan trajectory must be the same size (have the same number of indices) as the mass, centering and damping tables and must also have the same range as the size of these tables. For example, in our .csd we have been using 128 point tables for initial position, mass centering, damping (our stiffness tables have 128 squared). So our trajectory tables must be of size 128, and contain values from 0 to 127.

One can use larger or smaller tables, but their sizes must agree in this way or Csound will give you an error. Larger tables, of course significantly increase CPU usage and slow down real-time performance.

When using smaller size tables it may be necessary to use interpolation to avoid the artifacts of a small table. *scans* gives us this option as a fifth optional argument, *iorder*, detailed in the reference manual and worth experimenting with.

Using the opcodes *scanu* and *scans* require that we fill in 22 arguments and create at least 7 *f*-tables, including at least one external ASCII file (because no one wants to fill in 16,384 arguments to an *f*-statement). This a very challenging pair of opcodes. The beauty of scanned synthesis is that there is no scanned synthesis “sound”.

Using Balance to Tame Amplitudes

However, like this frontier can be a lawless, dangerous place. When experimenting with scanned synthesis parameters, one can illicit extraordinarily loud sounds out of Csound, often by something as simple as a misplaced decimal point.

Warning: the following .csd is hot, it produces massively loud amplitude values. Be very cautious about rendering this .csd, I highly recommend rendering to a file instead of real-time. Only uncomment line 43 when you know what you do!

EXAMPLE 04H08_Scan_extreme_amplitude.csd

```
<CsoundSynthesizer>
<CsOptions>
-o dac
</CsOptions>
<CsInstruments>

nchnls = 2
sr = 44100
ksmps = 32
0dbfs = 1
;NOTE THIS CSD WILL NOT RUN UNLESS
;IT IS IN THE SAME FOLDER AS THE FILE "STRING-128"
instr 1
ipos ftgen 1, 0, 128 , 10, 1
irate = .007
ifnvel ftgen 6, 0, 128 , -7, 0, 128, 0.1
ifnmass ftgen 2, 0, 128 , -7, 1, 128, 1
ifnstif ftgen 3, 0, 16384, -23, "string-128"
ifncentr ftgen 4, 0, 128 , -7, 1, 128, 2
ifndamp ftgen 5, 0, 128 , -7, 1, 128, 1
kmass = 1
kstif = 0.1
kcentr = .01
kdamp = 1
ileft = 0
iright = 1
kpos = 0
kstrngth = 0.
ain = 0
idisp = 1
id = 22
scanu ipos, irate, ifnvel, ifnmass, \
ifnstif, ifncentr, ifndamp, kmass, \
kstif, kcetr, kdamp, ileft, iright, \
kpos, kstrngth, ain, idisp, id
kamp = 0dbfs*.2
kfreq = 200
ifn ftgen 7, 0, 128, -5, .001, 128, 128.
a1 scans kamp, kf freq, ifn, id
a1 dcblock2 a1
iatt = .005
idec = 1
islev = 1
irel = 2
aenv adsr iatt, idec, islev, irel
```

```
;outs a1*aenv,a1*aenv; Uncomment for speaker destruction;
endin
</CsInstruments>
<CsScore>
f8 0 8192 10 1;
i 1 0 5
</CsScore>
</CsoundSynthesizer>
;example by Christopher Saunders
```

The extreme volume of this .csd comes from a value given to scanu

```
kdamp = .1
```

0.1 is not exactly a safe value for this argument, in fact, any value above 0 for this argument can cause chaos.

It would take a skilled mathematician to map out safe possible ranges for all the arguments of scanu. I figured out these values through a mix of trial and error and studying other .csd.

We can use the opcode **balance** to listen to sine wave (a signal with consistent, safe amplitude) and squash down our extremely loud scanned synth output (which is loud only because of our intentional carelessness.)

EXAMPLE 04H09_Scan_balanced_amplitudes.csd

```
<CsoundSynthesizer>
<CsOptions>
-o dac
</CsOptions>
<CsInstruments>

nchnls = 2
sr = 44100
ksmps = 256
0dbfs = 1
;NOTE THIS CSD WILL NOT RUN UNLESS
;IT IS IN THE SAME FOLDER AS THE FILE "STRING-128"

instr 1
ipos ftgen 1, 0, 128 , 10, 1
irate = .007
ifnvel  ftgen 6, 0, 128 , -7, 0, 128, 0.1
ifnmass  ftgen 2, 0, 128 , -7, 1, 128, 1
ifnstif  ftgen 3, 0, 16384, -23, "string-128"
ifncentr ftgen 4, 0, 128 , -7, 1, 128, 2
ifndamp  ftgen 5, 0, 128 , -7, 1, 128, 1
kmass = 1
kstif = 0.1
kcentr = .01
kdamp = -0.01
ileft = 0
iright = 1
kpos = 0
kstrngth = 0.
ain = 0
idisp = 1
id = 22
scanu ipos, irate, ifnvel, ifnmass,
ifnstif, ifncentr, ifndamp, kmass,
```

```
kstif, kcentr, kdamp, ileft, iright,\  
kpos, kstrngth, ain, idisp, id  
kamp = 0dbfs*.2  
kfreq = 200  
ifn ftgen 7, 0, 128, -5, .001, 128, 128.  
a1 scans kamp, kfreq, ifn, id  
a1 dcblock2 a1  
ifnsine ftgen 8, 0, 8192, 10, 1  
a2 poscil kamp, kfreq, ifnsine  
a1 balance a1, a2  
iatt = .005  
idec = 1  
islev = 1  
irel = 2  
aenv adsr iatt, idec, islev, irel  
outs a1*aenv,a1*aenv  
endin  
</CsInstruments>  
<CsScore>  
f8 0 8192 10 1;  
i 1 0 5  
</CsScore>  
</CsoundSynthesizer>  
;example by Christopher Saunders
```

It must be emphasized that this is merely a safeguard. We still get samples out of range when we run this .csd, but many less than if we had not used balance. It is recommended to use balance if you are doing real-time mapping of k-rate profile scalar arguments for `scans`; mass stiffness, damping, and centering.

05 A. ENVELOPES

Envelopes are used to define how a value evolves over time. In early synthesisers, envelopes were used to define the changes in amplitude in a sound across its duration thereby imbuing sounds characteristics such as *percussive*, or *sustaining*. Envelopes are also commonly used to modulate filter cutoff frequencies and the frequencies of oscillators but in reality we are only limited by our imaginations in regard to what they can be used for.

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 types shall be introduced here.

line

The simplest opcode for defining an envelope is `line`. It describes a single envelope segment as a straight line between a start value *ia* and an end value *ib* which has a given duration *idur*.

```
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_line.csd

```
<CsoundSynthesizer>  
<CsOptions>  
-odac  
</CsOptions>  
<CsInstruments>  
sr = 44100  
ksmps = 32  
nchnls = 2  
0dbfs = 1  
  
instr 1  
aEnv    line    0.5, 2, 0      ; amplitude envelope  
aSig     poscil  aEnv, 500      ; audio oscillator  
        out     aSig, aSig      ; audio sent to output
```

```

    endin

  </CsInstruments>
  <CsScore>
  i 1 0 2 ; instrument 1 plays a note for 2 seconds
  </CsScore>
</CsoundSynthesizer>
;example by Iain McCurdy

```

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_line_p3.csd

```

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

instr 1
; A single segment envelope. Time value defined by note duration.
aEnv    line    0.5, p3, 0
aSig    poscil  aEnv, 500
        out     aSig, aSig
    endin

</CsInstruments>
<CsScore>
; p1 p2  p3
i 1 0 1
i 1 2 0.2
i 1 3 4
</CsScore>
</CsoundSynthesizer>
;example by Iain McCurdy

```

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 quantisation 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

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.

```
ares *linseg* ia, idur1, ib [, idur2] [, ic] [...]
kres *linseg* ia, idur1, ib [, idur2] [, ic] [...]
```

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. An 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_linseg.csd

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

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
          out       aSig, aSig
        endin

</CsInstruments>
<CsScore>
i 1 0 1
i 1 2 5
</CsScore>
</CsoundSynthesizer>
;example by Iain McCurdy
```

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 note regardless of duration.

EXAMPLE 05A04_linseg_p3_fractions.csd

```

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

    instr 1
aEnv      linseg   0, p3*0.5, .2, p3*0.5, 0 ; rising then falling envelope
aSig      oscil    aEnv, 500
          out      aSig, aSig
    endin

</CsInstruments>

<CsScore>
; 3 notes of different durations are played
i 1 0  1
i 1 2 0.1
i 1 3  5
</CsScore>
</CsoundSynthesizer>
;example by Iain McCurdy

```

Different behaviour in linear continuation

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 the trajectory defined by its one and only 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. The *linseg* envelope stays at zero, whilst the *line* envelope continues through zero to negative range, thus ending at -0.2.¹

EXAMPLE 05A05_line_vs_linseg.csd

```

<CsoundSynthesizer>
<CsOptions>
-odac
</CsOptions>
<CsInstruments>
sr = 44100
ksmps = 32

```

¹Negative values for the envelope have the same loudness. Only the phase of the signal is inverted.

```

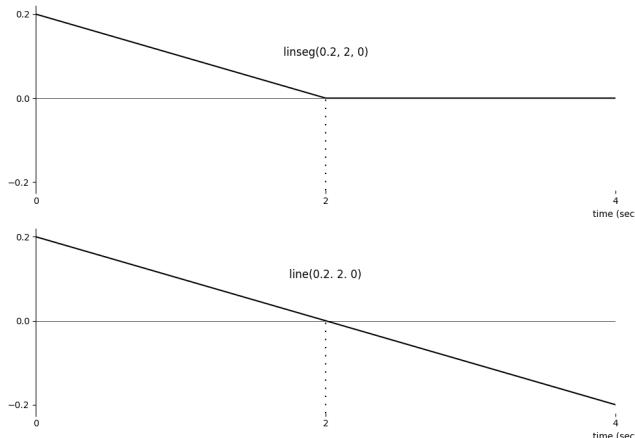
nchnls = 2
0dbfs = 1

instr 1 ; linseg envelope
aEnv    linseg  0.2, 2, 0      ; linseg holds its last value
aSig    oscil   aEnv, 500
        out     aSig, aSig
endin

instr 2 ; line envelope
aEnv    line    0.2, 2, 0      ; line continues its trajectory
aSig    oscil   aEnv, 500
        out     aSig
endin

</CsInstruments>
<CsScore>
i 1 0 4 ; linseg envelope
i 2 5 4 ; line envelope
</CsScore>
</CsoundSynthesizer>
;example by Iain McCurdy and joachim heintz

```



expon and expseg

`expon` and `expseg` are versions of `line` and `linseg` that instead produce envelope segments with concave exponential shapes 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.² On account of the mathematics that are 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.

²See chapter 01 C for some background information.

EXAMPLE 05A06_line_vs_expon.csd

```

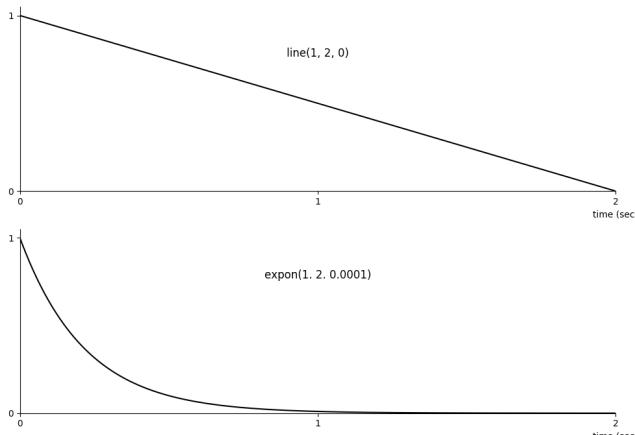
<CsoundSynthesizer>
<CsOptions>
-odac
</CsOptions>
<CsInstruments>
sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

    instr 1 ; line envelope
aEnv    line      1, p3, 0
aSig    oscil    aEnv, 500
        out     aSig, aSig
    endin

    instr 2 ; expon envelope
aEnv    expon    1, p3, 0.0001
aSig    oscil    aEnv, 500
        out     aSig, aSig
    endin

</CsInstruments>
<CsScore>
i 1 0 2 ; line envelope
i 2 2 2 ; expon envelope
</CsScore>
</CsoundSynthesizer>
;example by Iain McCurdy

```



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_expon_pings.csd

```

<CsoundSynthesizer>
<CsOptions>
-odac
</CsOptions>
<CsInstruments>
sr = 44100
ksmps = 32

```

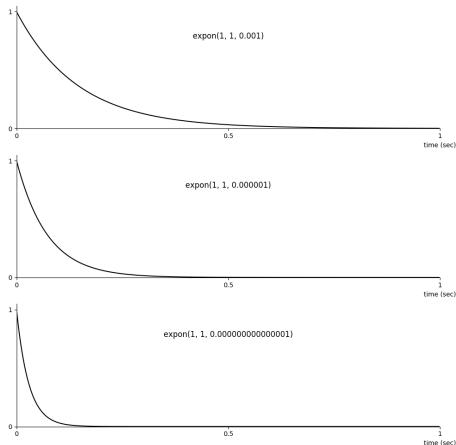
```

nchnls = 2
0dbfs = 1

instr 1; expon envelope
iEndVal = p4 ; variable 'iEndVal' retrieved from score
aEnv    expon 1, p3, iEndVal
aSig    oscil  aEnv, 500
        out    aSig, aSig
endin

</CsInstruments>
<CsScore>
;p1  p2  p3  p4
i 1  0  1  0.001
i 1  1  1  0.000001
i 1  2  1  0.0000000000000001
e
</CsScore>
</CsoundSynthesizer>
;example by Iain McCurdy

```



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.

Envelopes with release segment

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 typical situation for using one of these opcodes is when using a midi keyboard. When a key is released, a fade-out must be applied to avoid clicks. Another typical situation is shown in the next example: a running instrument is turned off by another instrument. Similar to the midi keyboard

usage, the instrument must add in this case a release segment to avoid clicks. The example shows both: how it sounds without release, and how it sounds with release. This is done via the last parameter of the `turnoff2` opcode. If `krelease` is set to zero, it will not allow the instrument it terminates to perform its release segment. This results in audible clicks on the first run. The second run (after creating a new cluster of random notes in the middle range) instead allows the release segment (in setting `krelease` to 1), so each of the notes gets a soft fade-out. (This example can be changed to be used via midi keyboard; follow the comments in the code in this case.)

EXAMPLE 05A08_linsegr.csd

```

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

sr = 44100
ksmps = 64
nchnls = 2
0dbfs = 1
seed 0

instr Generate
//create and start 10 instances of instr Play
indx = 0
while indx < 10 do
  schedule "Play", 0, 15
  indx += 1
od
endin

instr Play
iMidiNote random 60, 72
//use the following line instead when using midi input
;iMidiNote notnum
;aEnv    linsegr 0, 0.01, 0.1, 0.5,0; envelope that senses note releases
;aSig    poscil   aEnv, mtof:i(iMidiNote); audio oscillator
        out      aSig, aSig           ; audio sent to output
endin

instr TurnOff_noRelease
//turn off the ten instances from instr Play starting from the oldest one
//and do not allow the release segment to be performed (result: clicks)
kTrigFreq init 1
kTrig metro kTrigFreq
if kTrig == 1 then
  kRelease = 0 ;no release allowed
  turnoff2 "Play", 1, kRelease
endif
endin

instr TurnOff_withRelease
//turn off the ten instances from instr Play starting from the oldest one
//and do allow the release segment to be performed (no clicks any more)
kTrigFreq init 1
kTrig metro kTrigFreq
if kTrig == 1 then
  kRelease = 1 ;release allowed
  turnoff2 "Play", 1, 1

```

```

        endif
    endin

  
```

`</CsInstruments>
<CsScore>
//for real-time midi input, comment out all score lines
i "Generate" 0 0
i "TurnOff_noRelease" 1 11
i "Generate" 15 0
i "TurnOff_withRelease" 16 11
</CsScore>
</CsoundSynthesizer>
;example by joachim heintz`

Envelopes in Function Tables

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. If the oscillator is given a frequency of 1/p3 then it will read through the envelope just once across the duration of the note.

The following example generates an amplitude envelope which uses the shape of the first half of a sine wave.

EXAMPLE 05A09_sine_env.csd

```

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

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    .5, 1/p3, giEnv
aSig      poscil    aEnv, 500           ; audio oscillator
          out       aSig, 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

```

```
e 7.1
</CsScore>
</CsoundSynthesizer>
;example by Iain McCurdy
```

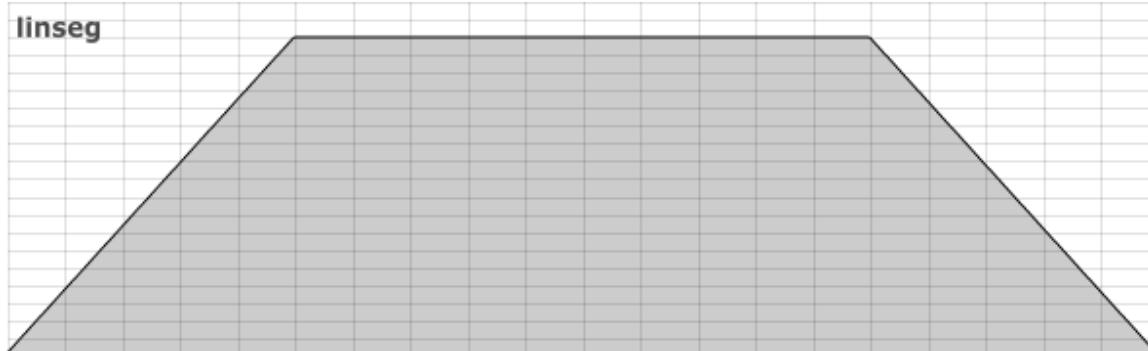
Comparison of the Standard Envelope Opcodes

The precise shape of the envelope of a sound, whether that envelope refers to its amplitude, its pitch or any other parameter, can be incredibly subtle and our ears, in identifying and characterising sounds, are fantastically adept at sensing those subtleties. Csound's original envelope generating opcode `linseg`, whilst capable of emulating the envelope generators of vintage electronic synthesisers, may not produce convincing results in the emulation of acoustic instruments and natural sound. Thus it has, since Csound's creation, been augmented with a number of other envelope generators whose usage is similar to that of `linseg` but whose output function is subtly different in shape.

If we consider a basic envelope that ramps up across $\frac{1}{4}$ of the duration of a note, then sustains for $\frac{1}{2}$ the durations of the and finally ramps down across the remaining $\frac{1}{4}$ duration of the note, we can implement this envelope using `linseg` thus:

```
kEnv linseg 0, p3/4, 0.9, p3/2, 0.9, p3/4, 0
```

The resulting envelope will look like this:

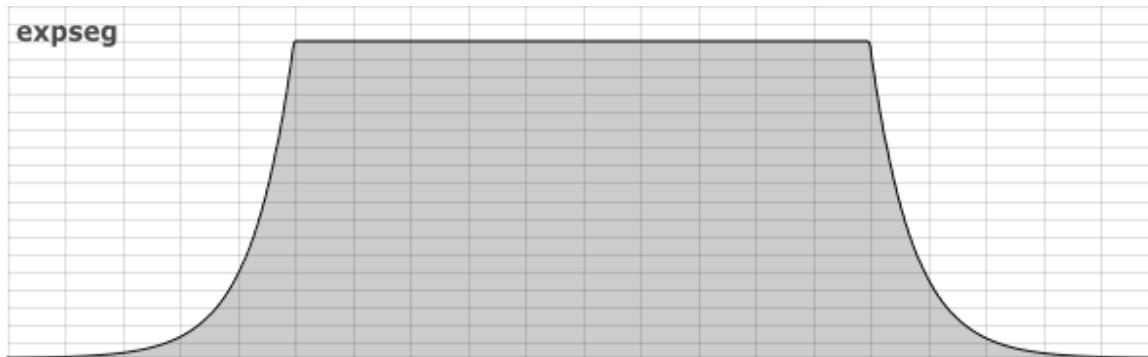


When employed as an amplitude control, the resulting sound may seem to build rather too quickly, then crescendo in a slightly mechanical fashion and finally arrive at its sustain portion with abrupt stop in the crescendo. Similar criticism could be levelled at the latter part of the envelope going from sustain to ramping down.

The `expseg` opcode, introduced sometime after `linseg`, attempted to address the issue of dynamic response when mapping an envelope to amplitude. Two caveats exist in regard to the use of `expseg`: firstly a single `expseg` definition cannot cross from the positive domain to the negative domain (and vice versa), and secondly it cannot reach zero. This second caveat means that an amplitude envelope created using `expseg` cannot express silence unless we remove the offset away from zero that the envelope employs. An envelope with similar input values to the `linseg` envelope above but created with `expseg` could use the following code:

```
kEnv expseg 0.001, p3/4, 0.901, p3/2, 0.901, p3/4, 0.001
kEnv = kEnv - 0.001
```

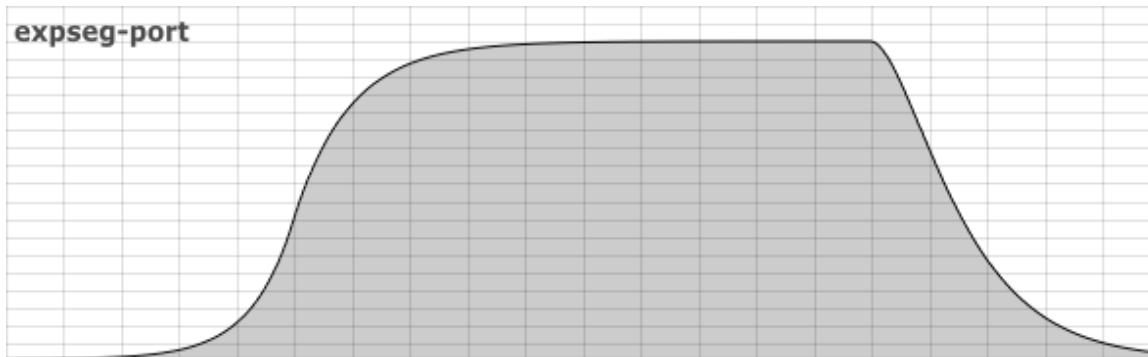
and would look like this:



In this example the offset above zero has been removed. This time we can see that the sound will build in a rather more natural and expressive way, however the change from *crescendo* to *sustain* is even more abrupt this time. Adding some lowpass filtering to the envelope signal can smooth these abrupt changes in direction. This could be done with, for example, the port opcode given a half-point value of 0.05.

```
kEnv port kEnv, 0.05
```

The resulting envelope looks like this:



The changes to and from the sustain portion have clearly been improved but close examination of the end of the envelope reveals that the use of port has prevented the envelope from reaching zero. Extending the duration of the note or overlaying a second *anti-click* envelope should obviate this issue.

```
xtratim 0.1
```

will extend the note by 1/10 of a second.

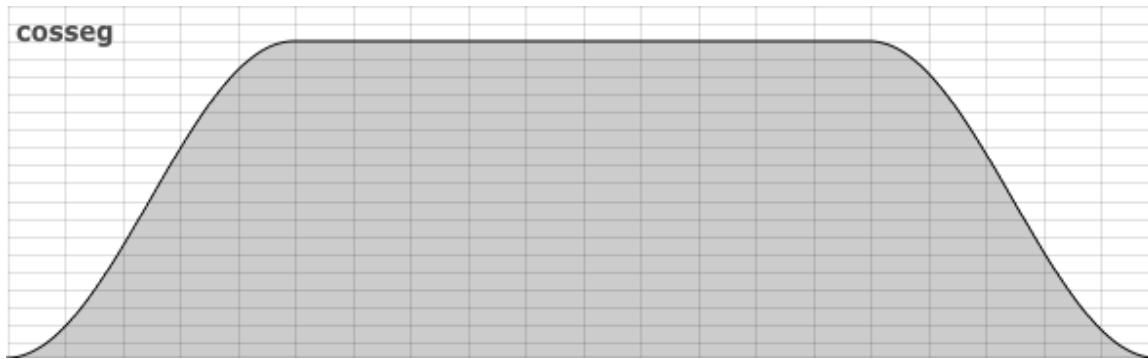
```
aRamp linseg 1, p3-0.1, 1, 0.1, 0
```

will provide a quick ramp down at the note conclusion if multiplied to the previously created envelope.

A more recently introduced alternative is the `cosseg` opcode which applies a cosine transfer function to each segment of the envelope. Using the following code:

```
kEnv cosseg 0, p3/4, 0.9, p3/2, 0.9, p3/4, 0
```

the resulting envelope will look like this:

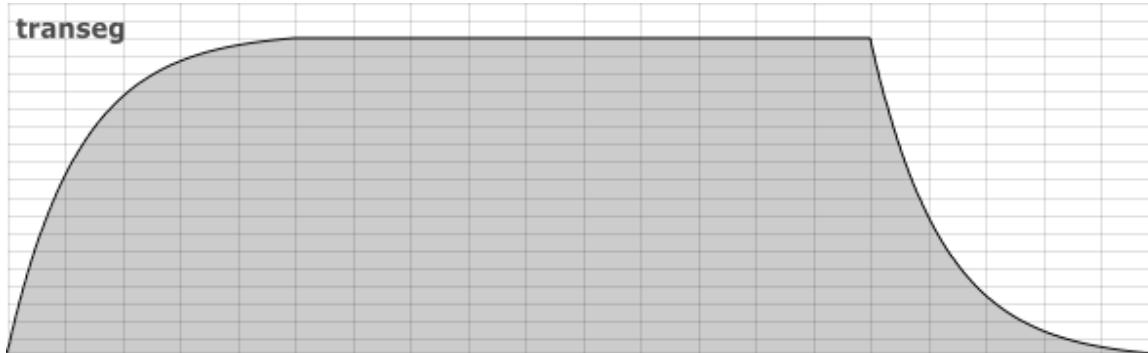


It can be observed that this envelope provides a smooth gradual building up from silence and a gradual arrival at the sustain level. This opcode has no restrictions relating to changing polarity or passing through zero.

Another alternative that offers enhanced user control and that might in many situations provide more natural results is the [transeg](#) opcode. *transeg* allows us to specify the curvature of each segment but it should be noted that the curvature is dependent upon whether the segment is rising or falling. For example a positive curvature will result in a concave segment in a rising segment but a convex segment in a falling segment. The following code:

```
kEnv transeg 0, p3/4, -4, 0.9, p3/2, 0, 0.9, p3/4, -4, 0
```

will produce the following envelope:



This looks perhaps rather lopsided but in emulating acoustic instruments can actually produce more natural results. Considering an instrument such as a clarinet, it is in reality very difficult to fade a note in smoothly from silence. It is more likely that a note will start slightly abruptly in spite of the player's efforts. This aspect is well represented by the attack portion of the envelope above. When the note is stopped, its amplitude will decay quickly and exponentially as reflected in the envelope also. Similar attack and release characteristics can be observed in the slight pitch envelopes expressed by wind instruments.

Ipshold, loopseg and looptseg - A Csound TB303

The next example introduces three of Csound's looping opcodes, [Ipshold](#), [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 provide the added convenience of encapsulating all the required code in one line without the need for [phasors](#), [tables](#) and [ftgens](#). Furthermore all of the input arguments for these opcodes can be modulated at k-rate.

lpshold generates an envelope 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 the notes (as MIDI note numbers) for 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 of the frequency – values for the durations of individual envelope segments are not defining times in seconds but instead 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 segments. In this example it is used to define the amplitude envelope for 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 it allows us to define the shape of each segment individually, whether that be convex, linear or concave. This aspect is defined using the *type* parameters. A type value of 0 denotes a linear segment, 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.

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](#) opcode in order to keep it interesting. It is suggested that these modulations could be replaced by linkages to other controls such as CsoundQt/Cabbage/Blue widgets, FLTK widgets or MIDI controllers. Suggested ranges for each of these values are given in the .csd.

EXAMPLE 05A10_*lpshold_loopseg.csd*

```
<CsoundSynthesizer>
<CsOptions>
-odac ;activates real time sound output
</CsOptions>
<CsInstruments>
sr = 44100
ksmps = 4
nchnls = 2
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
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, 1,0, 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, 1,0, 1,0, 1
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,0,0
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
aAmpEnv interp    kAmpEnv*kOn*kVol
aOut    =           aSig * aAmpEnv
        out       aOut, aOut
    endin

</CsInstruments>
<CsScore>
i 1 0 3600 ; instr 1 plays for 1 hour
</CsScore>
</CsoundSynthesizer>
;example by Iain McCurdy

```

Hopefully this final example has provided some idea as to the extend of parameters that can be controlled using envelopes and also an allusion to their importance in the generation of musical gesture.

05 B. PANNING AND SPATIALIZATION

The location is an important characteristics of real-world sounds. We can sometimes distinguish sounds because we distinguish their different locations. And in music the location can guide our hearing to different meanings.

This is shown at a very simple example. First we hear a percussive sound from both speakers. We will not recognize any pattern. Then we hear one beat from left speaker followed by three beats from right speaker. We will recognize this as 3/4 beats, with the first beat on the left speaker. Finally we hear a random sequence of left and right channel. We will hear this as something like a dialog between two players.

EXAMPLE 05B01_routing.csd

```
<CsoundSynthesizer>
<CsOptions>
-odac -m128
</CsOptions>
<CsInstruments>
sr = 44100
ksmps = 32
0dbfs = 1
nchnls = 2
seed 1

opcode Resonator, a, akk
aSig, kFreq, kQ xin
kRatio[] fillarray 1, 2.89, 4.95, 6.99, 8.01, 9.02
a1 = mode:a(aSig, kFreq*kRatio[0], kQ)
a2 = mode:a(aSig, kFreq*kRatio[1], kQ)
a3 = mode:a(aSig, kFreq*kRatio[2], kQ)
a4 = mode:a(aSig, kFreq*kRatio[3], kQ)
a5 = mode:a(aSig, kFreq*kRatio[4], kQ)
a6 = mode:a(aSig, kFreq*kRatio[5], kQ)
aSum sum a1, a2, a3, a4, a5, a6
aOut = balance:a(aSum, aSig)
xout aOut
endop

instr Equal
kTrig metro 80/60
schedkwhen kTrig, 0, 0, "Perc", 0, 1, .4, 1
schedkwhen kTrig, 0, 0, "Perc", 0, 1, .4, 2
endin

instr Beat
```

```

kRoutArr[] fillarray 1, 2, 2
kIndex init 0
if metro:k(80/60) == 1 then
  event "i", "Perc", 0, 1, .6, kRoutArr[kIndex]
  kIndex = (kIndex+1) % 3
endif
endin

instr Dialog
if metro:k(80/60) == 1 then
  event "i", "Perc", 0, 1, .6, int(random:k(1,2.999))
endif
endin

instr Perc
iAmp = p4
iChannel = p5
aBeat pluck iAmp, 100, 100, 0, 3, .5
aOut Resonator aBeat, 300, 5
outch iChannel, aOut
endin

</CsInstruments>
<CsScore>
i "Equal" 0 9.5
i "Beat" 11 9.5
i "Dialog" 22 9.5
</CsScore>
</CsoundSynthesizer>
;example by joachim heintz and philipp henkel

```

The spatialization technique used in this example is called *routing*. In routing we connect an audio signal directly to one speaker. This is a somehow brutal method which knows only black or white, only right or left. Usually we want to have a more refined way to locate sounds, with different positions between pure left and pure right. This is often compared to a *panorama* - a sound horizon on which certain sounds have a location between left and right. So we look first into this *panning* for a stereo setup. Then we will discuss the extension of panning in a multi-channl setup. The last part of this chapter is dedicated to the Ambisonics technique which offers a different way to locate sound sources.

Simple Stereo Panning

First we will look at some methods of panning a sound between two speakers based on first principles.

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 0 minus the same variable like this:

```

aSigL = aSig * (1 - kPan)
aSigR = aSig * kPan
outs aSigL, aSigR

```

kPan should be a value within the range zero and one. If kPan is 0 all of the signal will be in the left channel, if it is 1, all of the signal will be in the right channel and if it is 0.5 there will be signal

of equal amplitude 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.¹

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((1 - kPan))
aSigR = aSig * sqrt(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 90° 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 * cos(kPan*$M_PI_2)
aSigR = aSig * sin(kPan*$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 * cos((kPan + 0.5) * $M_PI_2)
aSigR = aSig * sin((kPan + 0.5) * $M_PI_2)
outs aSigL, aSigR
```

The following example demonstrates all these methods one after the other for comparison. Panning movement is controlled by a slow moving LFO. The input sound is filtered pink noise.

EXAMPLE 05B02_Pan_stereo.csd

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

instr 1
imethod = p4 ; read panning method variable from score (p4)

;----- generate a source sound -----
a1      pinkish  0.1          ; pink noise
a1      reson    a1, 500, 30, 2 ; bandpass filtered
```

¹The reason has been touched in chapter 01C: The sound intensity is not proportional to the amplitude but to the squared amplitude.

```

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    =          1 - aPan
aPanR    =          aPan
;-----
endif

if imethod=2 then
;----- method 2 -----
aPanL    =          sqrt(1 - aPan)
aPanR    =          sqrt(aPan)
;-----
endif

if imethod=3 then
;----- method 3 -----
aPanL    =          cos(aPan*$M_PI_2)
aPanR    =          sin(aPan*$M_PI_2)
;-----
endif

if imethod=4 then
;----- method 4 -----
aPanL    =          cos((aPan + 0.5) * $M_PI_2)
aPanR    =          sin((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>
;example by Iain McCurdy

```

The opcode `pan2` makes it easier for us to implement various methods of panning. 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.

EXAMPLE 05B03_pan2.csd

```

<CsoundSynthesizer>
<CsOptions>
-odac
</CsOptions>

<CsInstruments>

```

```

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

instr 1
imethod      =      p4 ; read panning method variable from score (p4)
;----- generate a source sound -----
aSig         pinkish  0.1          ; pink noise
aSig         reson    aSig, 500, 30, 2 ; bandpass filtered
;-----

;----- pan the signal -----
aPan         lfo      0.5, 1/2, 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>
;example by Iain McCurdy

```

3D Binaural Encoding

3D binaural encoding is available through 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 oldest one of these is [hrtfer](#). Newer ones are [hrtfmove](#), [hrtfmove2](#) and [hrtfstat](#). The main parameters for control of the opcodes are azimuth (the horizontal direction of the source expressed as an angle formed from the direction in which 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 (although a degree of effect is still audible through speakers).

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 level and directly in front of us. This example uses the files *hrtf-44100-left.dat* and *hrtf-44100-right.dat*. In case they are not loaded, they can be downloaded from the [Csound sources](#).

EXAMPLE 05B04_hrtfmove.csd

```

<CsoundSynthesizer>
<CsOptions>
--env:SADIR+=./SourceMaterials
-odac
</CsOptions>
<CsInstruments>
sr = 44100
ksmps = 32
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

gS_HRTF_left =          "hrtf-44100-left.dat"
gS_HRTF_right =         "hrtf-44100-right.dat"

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, 0.05,0, 0.95,0,0
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 output
aLeft, aRight hrtfmove2 aSig, kAz, kElev, gS_HRTF_left, gS_HRTF_right
                     outs      aLeft, aRight           ; audio to outputs
endin

</CsInstruments>
<CsScore>
i 1 0 24 ; instr 1 plays a note for 24 seconds
</CsScore>
</CsoundSynthesizer>
;example by Iain McCurdy

```

Going Multichannel

So far we have only considered working in 2-channels (stereo), but Csound is extremely flexible at working in more than 2 channels. By changing nchnls in the orchestra header we can specify any number of channels but we also need to ensure that we choose an audio hardware device using the `-odac` option that can handle multichannel audio. Audio channels sent from Csound, that do not address hardware channels, will simply not be reproduced. There may be some need to make adjustments to the software settings of your soundcard using its own software or the operating system's software, but due to the variety of sound hardware options available, it would be impossible to offer further specific advice here.

If you do not use the real-time option `-o dac` but render a file with `-o myfilename.wav`, there are no

restrictions though. Csound will render any multi-channel file independently from your sound card.

Sending Multichannel Sound to the Loudspeakers

In order to send multichannel audio or render a multichannel file we must use opcodes designed for that task. So far we have often used `outs` to send stereo sound to a pair of loudspeakers. The opcode `out` can also be used, and offers any number of output channels up to the maximum of the `nchnls` setting in the header of your .csd file.

So for `nchnls=2` the maximum output is stereo:

```
out aL, aR
```

For `nchnls=4` the maximum output is quadro:

```
out a1, a2, a3, a4
```

And for `nchnls=8` the maximum output is octo:

```
out a1, a2, a3, a4, a5, a6, a7, a8
```

So `out` can replace the opcodes `outs`, `outq`, `outh` and `outo` which were designed for exactly 2, 4, 6 and 8 output channels. `out` can also be used to work with odd channel numbers like 3, 5 or 7 although many soundcards work much better when a channel number of 2, 4 or 8 is used.

The only limitation of `out` is that it always counts from channel number 1. Imagine you have a soundcard with 8 analog outputs (counting 1-8) and 8 digital outputs (counting 9-16), and you want to use only the digital outputs. Here and in similar situations the `outch` opcode is the means of choice. It allows us to direct audio to a specific channel or list of channels and takes the form:

```
outch kchan1, asig1 [, kchan2] [, asig2] [...]
```

So we would write here `nchnls=16` to open the channels on the sound card, and then

```
outch 9,a1, 10,a2, 11,a3, 12,a4, 13,a5, 14,a6, 15,a7, 16,a8
```

to assign the audio signals `a1 ... a8` to the outputs `9 ... 16`.

Note that for `outch` channel numbers can be changed at k-rate thereby opening the possibility of changing the speaker configuration dynamically during performance. Channel numbers do not need to be sequential and unrequired channels can be left out completely. This can make life much easier when working with complex systems employing many channels.

Flexibly Moving Between Stereo and Multichannel

It may be useful to be able to move between working in multichannel (beyond stereo) and then moving back to stereo (when, for example, a multichannel setup is not available). It is useful to work with global variables for the output channels which are set to a hardware channel number on top of the Csound program.

In case we work for a 4-channel setup, we will write this *IO Setup* on top of our program:

```
nchnls = 4
giOutChn_1 = 1
giOutChn_2 = 2
giOutChn_3 = 3
giOutChn_4 = 4
```

And in the *output section* of our program we will use the variable names instead of the numbers, for instance:

```
outch giOutChn_1,a1, giOutChn_2,a2, giOutChn_3,a3, giOutChn_4,a4
```

In case we can at any time only work with a stereo soundcard, all we have to do is to change the *IO Setup* like this:

```
nchnls = 2
giOutChn_1 = 1
giOutChn_2 = 2
giOutChn_3 = 2
giOutChn_4 = 1
```

The *output section* will work as before, so it is a matter of some seconds to connect with another hardware setup.

VBAP

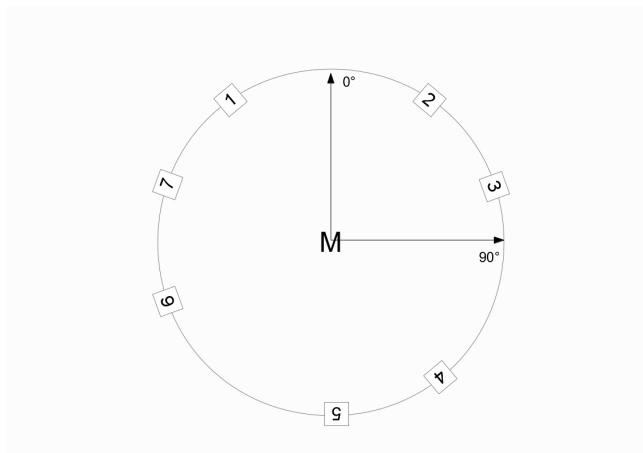
Vector Base Amplitude Panning² can be described as a method which extends stereo panning to more than two speakers. The number of speakers is, in general, arbitrary. Standard layouts such as quadrophonic, octophonic or 5.1 configuration can be used, but in fact any number of speakers can be positioned even in irregular distances from each other. Speakers arranged at different heights can as well be part of an VBAP loudspeaker array.

VBAP is robust and simple, and has proven its flexibility and reliability. Csound offers different op-codes which have evolved from the original implementation to flexibel setups using audio arrays. The introduction here will explain the usage from the first steps on.

Basic Steps

At first the VBAP system needs to know where the loudspeakers are positioned. This job is done with the opcode `vbapsinit`. Let us assume we have seven speakers in the positions and numberings outlined below (M = middle/centre):

²First described by Ville Pulkki in 1997: Ville Pulkki, Virtual source positioning using vector base amplitude panning, in: Journal of the Audio Engeneering Society, 45(6), 456-466



The *vbaplsinit* opcode which is usually placed in the header of a Csound orchestra, defines these positions as follows:

```
vbaplsinit 2, 7, -40, 40, 70, 140, 180, -110, -70
```

The first number determines the number of dimensions (here 2). The second number states the overall number of speakers, then followed by the positions in degrees (clockwise).

All that is required now is to provide *vbap* with a monophonic sound source to be distributed amongst the speakers according to information given about the position. Horizontal position (azimuth) is expressed in degrees clockwise just as the initial locations of the speakers were. The following would be the Csound code to play the sound file *ClassGuit.wav* once while moving it counterclockwise:

EXAMPLE 05B05_VBAP_circle.csd

```
<CsoundSynthesizer>
<CsOptions>
-odac
--env:SSDIR=../SourceMaterials
</CsOptions>
<CsInstruments>
sr = 44100
ksmps = 32
0dbfs = 1
nchnls = 8 ;only channels 1-7 used

vbaplsinit 2, 7, -40, 40, 70, 140, 180, -110, -70

    instr 1
Sfile      =      "ClassGuit.wav"
p3        filelen  Sfile
aSnd[]    diskin   Sfile
kAzim     line      0, p3, -360 ;counterclockwise
aVbap[]   vbap     aSnd[0], kAzim
          out      aVbap ;7 channel output via array
    endin
</CsInstruments>
<CsScore>
i 1 0 1
</CsScore>
</CsoundSynthesizer>
;example by joachim heintz
```

Let us look closer to some parts of this program.

- `-odac` enables realtime output. Choose `-o 05B04_out.wav` if you don't have a multichannel audio card, and Csound will render the output to the file `05B04_out.wav`.
- `-env:SSDIR+=./SourceMaterials` This statement will add the folder `SourceMaterials` which is placed in the top directory to Csound's search path for this file.
- `nchnls = 8` sets the number of channels to 8. `nchnls = 7` would be more consistent as we only use 7 channels. I chose 8 channels because some sound cards have problems to open 7 channels.
- `p3 filelen Sfile` sets the duration of the instrument (`p3`) to the length of the soundfile `Sfile` which in turn has been set to the "ClassGuit.wav" sample (you can use any other file here).
- `aSnd[] diskin Sfile` The opcode `diskin` reads the sound file `Sfile` and creates an audio array. The first channel of the file will be found in `aSnd[0]`, the second (if any) in `aSnd[1]`, and so on.
- `kAzim line 0,p3,-360` This creates an azimuth signal which starts at center (0°) and moves counterclockwise during the whole duration of the instrument call (`p3`) to center again (-360° is also in front).
- `aVbap[] vbap aSnd[0],kAzim` The opcode `vbap` creates here an audio array which contains as many audio signals as are set with the `vbapsinit` statement; in this case seven. These seven signals represent the seven loud speakers. Right hand side, `vbap` gets two inputs: the first channel of the `aSnd` array, and the `kAzim` signal which contains the location of the sound.
- `out aVbap` Note that `aVbap` is an audio array here which contains seven audio signals. The whole array is written to channels 1-7 of the output, either in realtime or as audio file.

The Spread Parameter

As VBAP derives from a panning paradigm, it has one problem which becomes more serious as the number of speakers increases. Panning between two speakers in a stereo configuration means that all speakers are active. Panning between two speakers in a quadro configuration means that half of the speakers are active. Panning between two speakers in an octo configuration means that only a quarter of the speakers are active and so on; so that the actual perceived extent of the sound source becomes unintentionally smaller and smaller.

To alleviate this tendency, Ville Pulkki has introduced an additional parameter, called *spread*, which has a range of zero to hundred percent.³ The "ascetic" form of VBAP we have seen in the previous example, means: no spread (0%). A spread of 100% means that all speakers are active, and the information about where the sound comes from is nearly lost.

The `kspread` input parameter is the second of three optional parameters of the `vbap` opcode:

```
array[] *vbap* asig, kazim [,kelev] [,kspread] [,ilayout]
```

So to set `kspread`, we first have to provide the first one. `kelev` defines the elevation of the sound - it is always zero for two dimensions, as in the speaker configuration in our example. The next example adds a spread movement to the previous one. The spread starts at zero percent, then increases to hundred percent, and then decreases back down to zero.

³Ville Pulkki, Uniform spreading of amplitude panned virtual sources, in: Proceedings of the 1999 IEEE Workshop on Applications of Signal Processing to Audio and Acoustics, Mohonk Mountain House, New Paltz

EXAMPLE 05B06_VBAP_spread.csd

```

<CsoundSynthesizer>
<CsOptions>
-odac
--env:SSDIR=../SourceMaterials
</CsOptions>
<CsInstruments>
sr = 44100
ksmps = 32
0dbfs = 1
nchnls = 8 ;only channels 1-7 used

vbaplsinit 2, 7, -40, 40, 70, 140, 180, -110, -70

    instr 1
Sfile      =      "ClassGuit.wav"
p3         filelen Sfile
aSnd[]     diskin  Sfile
kAzim      line     0, p3, -360 ;counterclockwise
kSpread    linseg   0, p3/2, 100, p3/2, 0
aVbap[]    vbap    aSnd[0], kAzim, 0, kSpread
            out     aVbap
    endin
</CsInstruments>
<CsScore>
i 1 0 1
</CsScore>
</CsoundSynthesizer>
;example by joachim heintz

```

Different VBAP Layouts in one File

The `vbap` opcode which we already used in these examples does also allow to work with different configurations or speaker *layouts*. This layout is added as fractional number to the first parameter (*idim*) of `vbaplsinit`. Setting 2.03 here would declare VBAP layout 3, and an instance of the `vbap` opcode would refer to this layout by the last optional parameter *ilayout*.

By this it is possible to switch between different layouts during performance and to provide more flexibility in the number of output channels used. Here is an example for three different layouts which are called in three different instruments:

EXAMPLE 05B07_VBAP_layouts.csd

```

<CsoundSynthesizer>
<CsOptions>
-odac
</CsOptions>
<CsInstruments>
sr = 44100
ksmps = 32
0dbfs = 1
nchnls = 8

vbaplsinit 2.01, 7, -40, 40, 70, 140, 180, -110, -70
vbaplsinit 2.02, 4, -40, 40, 120, -120

```

```

vbaplsinit 2.03, 3, -70, 180, 70

    instr 1
aNoise    pinkish  0.5
aVbap[]   vbap      aNoise, line:k(0,p3,-360), 0, 0, 1
          out       aVbap ;layout 1: 7 channel output
    endin

    instr 2
aNoise    pinkish  0.5
aVbap[]   vbap      aNoise, line:k(0,p3,-360), 0, 0, 2
          out       aVbap ;layout 2: 4 channel output
    endin

    instr 3
aNoise    pinkish  0.5
aVbap[]   vbap      aNoise, line:k(0,p3,-360), 0, 0, 3
          out       aVbap ;layout 3: 3 channel output
    endin

</CsInstruments>
<CsScore>
i 1 0 6
i 2 6 6
i 3 12 6
</CsScore>
</CsoundSynthesizer>
;example by joachim heintz

```

In addition to the `vbap` opcode, `vbapg` has been written. The idea is to have an opcode which returns the gains (amplitudes) of the speakers instead of the audio signal:

```
k1[, k2...] vbapg kazim [,kelev] [, ksspread] [, ilayout]
```

Ambisonics I: *bformenc1* and *bformdec1*

Ambisonics is another technique to distribute a virtual sound source in space. The main difference to VBAP is that Ambisonics is shaping a *sound field* rather than working with different intensities to locate sounds.

There are excellent sources for the discussion of Ambisonics online which explain its background and parameters.⁴ These topics are also covered later in this chapter when Ambisonics UDOs are introduced. We will focus here first on the basic practicalities of using the Ambisonics opcodes *bformenc1* and *bformdec1* in Csound.

Two steps are required for distributing a sound via Ambisonics. At first the sound source and its localisation are *encoded*. The result of this step is a so-called *B-format*. In the second step this *B-format* is *decoded* to match a certain loudspeaker setup.

It is possible to save the B-format as its own audio file, to preserve the spatial information or you can immediately do the decoding after the encoding thereby dealing directly only with audio signals instead of Ambisonic files. The next example takes the latter approach by implementing

⁴For instance www.ambisonic.net or www.icst.net/research/projects/ambisonics-theory

a transformation of the VBAP circle example to Ambisonics.

EXAMPLE 05B08_Ambi_circle.csd

```
<CsoundSynthesizer>
<CsOptions>
-odac
--env:SSDIR+=../SourceMaterials
</CsOptions>
<CsInstruments>
sr = 44100
ksmps = 32
0dbfs = 1
nchnls = 8

instr 1
Sfile      =      "ClassGuit.wav"
p3         filelen   Sfile
aSnd[]     diskin    Sfile
kAzim      line      0, p3, 360 ;counterclockwise (!)
iSetup     =        4 ;octagon
aw, ax, ay, az bformenc1 aSnd[0], kAzim, 0
a1, a2, a3, a4, a5, a6, a7, a8 bformdec1 iSetup, aw, ax, ay, az
          out       a1, a2, a3, a4, a5, a6, a7, a8
        endin
</CsInstruments>
<CsScore>
i 1 0 1
</CsScore>
</CsoundSynthesizer>
;example by joachim heintz
```

The first thing to note is that for a counterclockwise circle, the azimuth now has the line 0 -> 360, instead of 0 -> -360 as was used in the VBAP example. This is because Ambisonics usually reads the angle in a mathematical way: a positive angle is *counterclockwise*. Next, the encoding process is carried out in the line:

```
aw, ax, ay, az bformenc1 aSnd, kAzim, 0
```

Input arguments are the monophonic sound source *aSnd[0]*, the xy-angle *kAzim*, and the elevation angle which is set to zero. Output signals are the spatial information in x-, y- and z- direction (*ax*, *ay*, *az*), and also an omnidirectional signal called *aw*.

Decoding is performed by the line:

```
a1, a2, a3, a4, a5, a6, a7, a8 bformdec1 iSetup, aw, ax, ay, az
```

The inputs for the decoder are the same *aw*, *ax*, *ay*, *az*, which were the results of the encoding process, and an additional *iSetup* parameter. Currently the Csound decoder only works with some standard setups for the speaker: *iSetup = 4* refers to an octagon.⁵ So the final eight audio signals *a1*, ..., *a8* are being produced using this decoder, and are then sent to the speakers.

⁵See www.csound.com/docs/manual/bformdec1.html for more details.

Different Orders

What we have seen in this example is called *first order* ambisonics. This means that the encoding process leads to the four basic dimensions w, x, y, z as described above. In *second order* ambisonics, there are additional directions called r, s, t, u, v . And in *third order* ambisonics again the additional k, l, m, n, o, p, q directions are applied. The final example in this section shows the three orders, each of them in one instrument. If you have eight speakers in octophonic setup, you can compare the results.

EXAMPLE 05B09_Ambi_orders.csd

```

<CsoundSynthesizer>
<CsOptions>
-odac
--env:SSDIR=../SourceMaterials
</CsOptions>
<CsInstruments>
sr = 44100
ksmps = 32
0dbfs = 1
nchnls = 8

    instr 1 ;first order
aSnd[]    diskin    "ClassGuit.wav"
kAzim     line      0, p3, 360
iSetup    =         4 ;octogon
aw, ax, ay, az bformenc1 aSnd[0], kAzim, 0
a1, a2, a3, a4, a5, a6, a7, a8 bformdec1 iSetup, aw, ax, ay, az
                                out      a1, a2, a3, a4, a5, a6, a7, a8
    endin

    instr 2 ;second order
aSnd[]    diskin    "ClassGuit.wav"
kAzim     line      0, p3, 360
iSetup    =         4 ;octogon
aw, ax, ay, az, ar, as, at, au, av bformenc1 aSnd, kAzim, 0
a1, a2, a3, a4, a5, a6, a7, a8 bformdec1 iSetup,
                                aw, ax, ay, az, ar, as, at, au, av
                                out      a1, a2, a3, a4, a5, a6, a7, a8
    endin

    instr 3 ;third order
aSnd[]    diskin    "ClassGuit.wav"
kAzim     line      0, p3, 360
iSetup    =         4 ;octogon
aw,ax,ay,az,ar,as,at,au,av,ak,al,am,an,ao,ap,aq bformenc1 aSnd, kAzim, 0
a1, a2, a3, a4, a5, a6, a7, a8 bformdec1 iSetup,
                                aw, ax, ay, az, ar, as, at, au, av, ak, al, am, an, ao, ap, aq
                                out      a1, a2, a3, a4, a5, a6, a7, a8
    endin
</CsInstruments>
<CsScore>
i 1 0 6
i 2 7 6
i 3 14 6
</CsScore>
</CsoundSynthesizer>
;example by joachim heintz

```

In theory, first-order ambisonics need at least 4 speakers to be projected correctly. Second-order ambisonics needs at least 6 speakers (9, if 3 dimensions are employed). Third-order ambisonics need at least 8 speakers (or 16 for 3d). So, although higher order should in general lead to a better result in space, you cannot expect it to work unless you have a sufficient number of speakers. Nevertheless practice over theory may prove to be a better judge in many cases.

Ambisonics II: UDOs

In the following section we introduce User Defined OpCodes (UDOs) for Ambisonics. The channels of the B-format are stored in a *zak* space. Call *zakinit* only once and put it outside of any instrument definition in the orchestra file after the header. *zacl* clears the *zak* space and is called after decoding. The *B format* of order n can be decoded in any order.

The text files *ambisonics_udos.txt*, *ambisonics2D_udos.txt*, *AEP_udos.txt* and *utilities.txt* must be located in the same folder as the csd files or included with full path.⁶

Introduction

We will explain here the principles of ambisonics step by step and write a UDO for every step. Since the two-dimensional analogy to Ambisonics is easier to understand and to implement with a simple equipment, we shall fully explain it first.

Ambisonics is a technique of three-dimensional sound projection. The information about the recorded or synthesized sound field is encoded and stored in several channels, taking no account of the arrangement of the loudspeakers for reproduction. The encoding of a signal's spatial information can be more or less precise, depending on the so-called order of the algorithm used. Order zero corresponds to the monophonic signal and requires only one channel for storage and reproduction. In first-order Ambisonics, three further channels are used to encode the portions of the sound field in the three orthogonal directions x, y and z. These four channels constitute the so-called first-order B-format. When Ambisonics is used for artificial spatialisation of recorded or synthesized sound, the encoding can be of an arbitrarily high order. The higher orders cannot be interpreted as easily as orders zero and one.

In a two-dimensional analogy to Ambisonics (called Ambisonics2D in what follows), only sound waves in the horizontal plane are encoded.

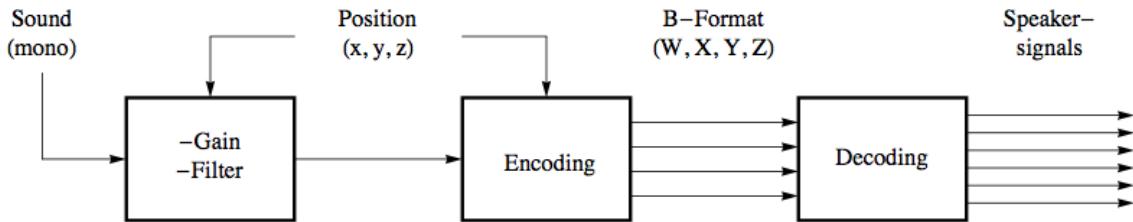
The loudspeaker feeds are obtained by decoding the B-format signal. The resulting panning is amplitude panning, and only the direction to the sound source is taken into account.

The illustration below shows the principle of Ambisonics. First a sound is generated and its position determined. The amplitude and spectrum are adjusted to simulate distance, the latter using a low-pass filter. Then the Ambisonic encoding is computed using the sound's coordinates. Encoding m th order B-format requires $n = (m + 1)^2$ channels ($n = 2m + 1$ channels in Ambisonics2D).

⁶These files can be downloaded together with the entire examples (some of them for CsoundQt) from <https://www.zdhk.ch/5382>

By decoding the B-format, one can obtain the signals for any number ($\geq n$) of loudspeakers in any arrangement. Best results are achieved with symmetrical speaker arrangements.

If the B-format does not need to be recorded the speaker signals can be calculated at low cost and arbitrary order using so-called ambisonics equivalent panning (AEP).



Ambisonics2D

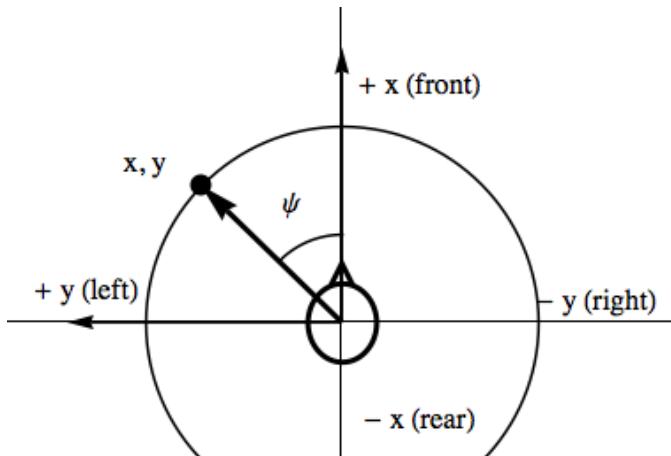
We will first explain the encoding process in Ambisonics2D. The position of a sound source in the horizontal plane is given by two coordinates. In Cartesian coordinates (x, y) the listener is at the origin of the coordinate system $(0, 0)$, and the x -coordinate points to the front, the y -coordinate to the left. The position of the sound source can also be given in polar coordinates by the angle ψ between the line of vision of the listener (front) and the direction to the sound source, and by their distance r .

Cartesian coordinates can be converted to polar coordinates by the formulae:

$$r = \sqrt{x^2 + y^2} \text{ and } \psi = \arctan(x, y),$$

polar to Cartesian coordinates by

$$x = r \cdot \cos(\psi) \text{ and } y = r \cdot \sin(\psi).$$



The 0th order B-Format of a signal S of a sound source on the unit circle is just the mono signal: $W_0 = W = S$. The first order B-Format contains two additional channels: $W_{1,1} = X = S \cdot \cos(\psi) = S \cdot x$ and $W_{1,2} = Y = S \cdot \sin(\psi) = S \cdot y$, i.e. the product of the Signal S with the sine and the cosine of the direction ψ of the sound source. The B-Format higher order contains two additional channels per order m: $W_{m,1} = S \cdot \cos(m\psi)$ and $W_{m,2} = S \cdot \sin(m\psi)$.

$$\begin{aligned} W_0 &= S \\ W_{1,1} &= X = S \cdot \cos(\psi) = S \cdot x \text{ and} \\ W_{1,2} &= Y = S \cdot \sin(\psi) = S \cdot y \\ W_{2,1} &= S \cdot \cos(2\psi) \text{ and} \\ W_{2,2} &= S \cdot \sin(2\psi) \\ &\dots \\ W_{m,1} &= S \cdot \cos(m\psi) \text{ and } W_{m,2} = S \cdot \sin(m\psi) \end{aligned}$$

From the $n = 2m + 1$ B-Format channels the loudspeaker signals p_i of n loudspeakers which are set up symmetrically on a circle (with angle ϕ_i) are:

$$\begin{aligned} p_i &= \frac{1}{n} \cdot (W_0 + 2W_{1,1}\cos(\phi_i) + 2W_{1,2}\sin(\phi_i) + 2W_{2,1}\cos(2\phi_i) + 2W_{2,2}\sin(2\phi_i) + \dots) \\ &= \frac{2}{n} \cdot (\frac{1}{2}W_0 + W_{1,1}\cos(\phi_i) + W_{1,2}\sin(\phi_i) + W_{2,1}\cos(2\phi_i) + W_{2,2}\sin(2\phi_i) + \dots) \end{aligned}$$

(If more than n speakers are used, we can use the same formula.)

In the following Csound example *udo_ambisonics2D_1.csd* the UDO *ambi2D_encode_1a* produces the 3 channels W, X and Y (*a0, a11, a12*) from an input sound and the angle ψ (azimuth *kaz*), the UDO *ambi2D_decode_1_8* decodes them to 8 speaker signals *a1, a2, ..., a8*. The inputs of the decoder are the 3 channels *a0, a11, a12* and the 8 angles of the speakers.

EXAMPLE 05B10_udo_ambisonics2D_1.csd

```
<CsoundSynthesizer>
<CsOptions>
-odac
</CsOptions>
<CsInstruments>
sr      =  44100
ksmps   =  32
nchnls  =  8
0dbfs   =  1
; ambisonics2D first order without distance encoding
; decoding for 8 speakers symmetrically positioned on a circle

```

```

; produces the 3 channels 1st order; input: asound, kazimuth
opcode ambi2D_encode_1a, aaa, ak
asnd,kaz      xin
kaz = $M_PI*kaz/180
a0      =      asnd
a11     =      cos(kaz)*asnd
a12     =      sin(kaz)*asnd
                  xout          a0,a11,a12
endop

; decodes 1st order to a setup of 8 speakers at angles i1, i2, ...
opcode ambi2D_decode_1_8, aaaaaaaaa, aaaiiiiiii
a0,a11,a12,i1,i2,i3,i4,i5,i6,i7,i8      xin
i1 = $M_PI*i1/180
i2 = $M_PI*i2/180
i3 = $M_PI*i3/180
i4 = $M_PI*i4/180
i5 = $M_PI*i5/180
i6 = $M_PI*i6/180
i7 = $M_PI*i7/180
i8 = $M_PI*i8/180
a1      =      (.5*a0 + cos(i1)*a11 + sin(i1)*a12)*2/3
a2      =      (.5*a0 + cos(i2)*a11 + sin(i2)*a12)*2/3
a3      =      (.5*a0 + cos(i3)*a11 + sin(i3)*a12)*2/3
a4      =      (.5*a0 + cos(i4)*a11 + sin(i4)*a12)*2/3
a5      =      (.5*a0 + cos(i5)*a11 + sin(i5)*a12)*2/3
a6      =      (.5*a0 + cos(i6)*a11 + sin(i6)*a12)*2/3
a7      =      (.5*a0 + cos(i7)*a11 + sin(i7)*a12)*2/3
a8      =      (.5*a0 + cos(i8)*a11 + sin(i8)*a12)*2/3
                  xout          a1,a2,a3,a4,a5,a6,a7,a8
endop

instr 1
asnd    rand    .05
kaz     line    0,p3,3*360 ;turns around 3 times in p3 seconds
a0,a11,a12 ambi2D_encode_1a asnd,kaz
a1,a2,a3,a4,a5,a6,a7,a8 \
    ambi2D_decode_1_8 a0,a11,a12,
                      0,45,90,135,180,225,270,315
    outc    a1,a2,a3,a4,a5,a6,a7,a8
endin

</CsInstruments>
<CsScore>
i1 0 40
</CsScore>
</CsoundSynthesizer>
;example by martin neukom

```

The B-format for all signals in all instruments can be summed before decoding. Thus in the next example we create a zak space with 21 channels (zakinit 21, 1) for the 2D B-format up to 10th order where the encoded signals are accumulated. The UDO *ambi2D_encode_3* shows how to produce the 7 B-format channels a0, a11, a12, ..., a32 for third order. The opcode *ambi2D_encode_n* produces the 2(n+1) channels a0, a11, a12, ..., a32 for any order n (needs zakinit 2(n+1), 1). The UDO *ambi2D_decode_basic* is an overloaded function i.e. it decodes to n speaker signals depending on the number of in- and outputs given (in this example only for 1 or 2 speakers). Any number of instruments can be played arbitrarily often. Instrument 10 decodes for the first 4 speakers of an 18 speaker setup.

EXAMPLE 05B11_udo_ambisonics2D_2.csd

```

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

sr      =  44100
ksmps   =  32
nchnls  =  4
0dbfs   =  1

; ambisonics2D encoding fifth order
; decoding for 8 speakers symmetrically positioned on a circle
; all instruments write the B-format into a buffer (zak space)
; instr 10 decodes

; zak space with the 21 channels of the B-format up to 10th order
zakin1t 21, 1

;explicit encoding third order
opcode  ambi2D_encode_3, 0, ak
asnd,kaz      xin

kaz = $M_PI*kaz/180

          zawm      asnd,0
          zawm      cos(kaz)*asnd,1      ;a11
          zawm      sin(kaz)*asnd,2      ;a12
          zawm      cos(2*kaz)*asnd,3    ;a21
          zawm      sin(2*kaz)*asnd,4    ;a22
          zawm      cos(3*kaz)*asnd,5    ;a31
          zawm      sin(3*kaz)*asnd,6    ;a32

endop

; encoding arbitrary order n(zakin1t 2*n+1, 1)
opcode  ambi2D_encode_n, 0, aik
asnd,iorder,kaz xin
kaz = $M_PI*kaz/180
kk = iorder
c1:
          zawm      cos(kk*kaz)*asnd,2*kk-1
          zawm      sin(kk*kaz)*asnd,2*kk
kk =
kk-1

if      kk > 0 goto c1
          zawm      asnd,0
endop

; basic decoding for arbitrary order n for 1 speaker
opcode  ambi2D_decode_basic, a, ii
iorder,iaz      xin
iaz = $M_PI*iaz/180
igain   =      2/(2*iorder+1)
kk = iorder
a1      =      .5*zar(0)
c1:
a1 +=  cos(kk*iaz)*zar(2*kk-1)
a1 +=  sin(kk*iaz)*zar(2*kk)
kk =
kk-1
if      kk > 0 goto c1

```

```

        xout          igain*a1
endop

; decoding for 2 speakers
opcode ambi2D_decode_basic, aa, iii
iorder,iaz1,iaz2      xin
iaz1 = $M_PI*iaz1/180
iaz2 = $M_PI*iaz2/180
igain = 2/(2*iorder+1)
kk = iorder
a1 = .5*zar(0)
c1:
a1 += cos(kk*iaz1)*zar(2*kk-1)
a1 += sin(kk*iaz1)*zar(2*kk)
kk = kk-1
if kk > 0 goto c1

kk = iorder
a2 = .5*zar(0)
c2:
a2 += cos(kk*iaz2)*zar(2*kk-1)
a2 += sin(kk*iaz2)*zar(2*kk)
kk = kk-1
if kk > 0 goto c2
        xout          igain*a1,igain*a2
endop

instr 1
asnd rand      p4
ares reson    asnd,p5,p6,1
kaz  line      0,p3,p7*360 ;turns around p7 times in p3 seconds
                ambi2D_encode_n asnd,10,kaz
endin

instr 2
asnd oscil     p4,p5,1
kaz  line      0,p3,p7*360 ;turns around p7 times in p3 seconds
                ambi2D_encode_n asnd,10,kaz
endin

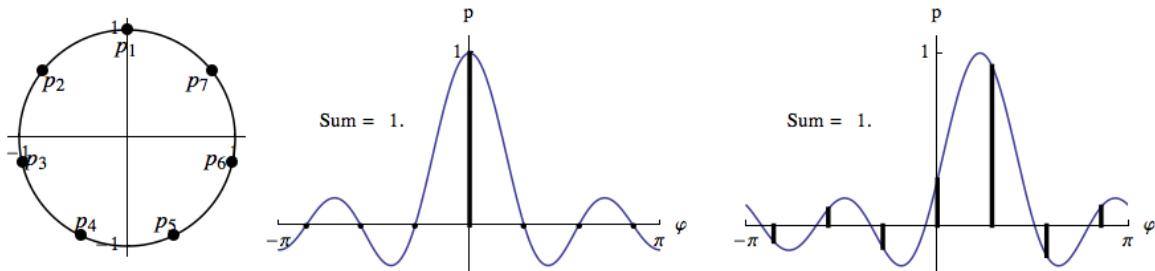
instr 10 ;decode all instruments (the first 4 speakers of a 18 speaker setup)
a1,a2      ambi2D_decode_basic   10,0,20
a3,a4      ambi2D_decode_basic   10,40,60
          outc   a1,a2,a3,a4
          zacl   0,20           ; clear the za variables
endin

</CsInstruments>
<CsScore>
f1 0 32768 10 1
;                                amp      cf      bw
i1 0 3  .7      1500    12      1
i1 2 18       .1  2234    34      -8
;                                amp      fr      0
i2 0 3  .1      440     0      2
i10 0 3
</CsScore>
</CsoundSynthesizer>
;example by martin neukom

```

In-phase Decoding

The left figure below shows a symmetrical arrangement of 7 loudspeakers. If the virtual sound source is precisely in the direction of a loudspeaker, only this loudspeaker gets a signal (center figure). If the virtual sound source is between two loudspeakers, these loudspeakers receive the strongest signals; all other loudspeakers have weaker signals, some with negative amplitude, that is, reversed phase (right figure).



To avoid having loudspeaker sounds that are far away from the virtual sound source and to ensure that negative amplitudes (inverted phase) do not arise, the B-format channels can be weighted before being decoded. The weighting factors depend on the highest order used (M) and the order of the particular channel being decoded (m).

$$g_m = \frac{(M!)^2}{((M+m)!(M-m)!)}$$

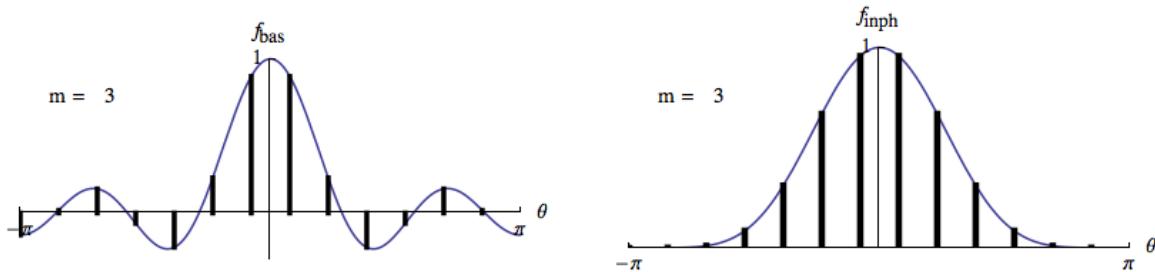
M		g_1	g_2	g_3	g_4	g_5	g_6	g_7	g_8
1	1	0.5							
2	1	0.666667	0.166667						
3	1	0.75	0.3	0.05					
4	1	0.8	0.4	0.114286	0.0142857				
5	1	0.833333	0.47619	0.178571	0.0396825	0.00396825			
6	1	0.857143	0.535714	0.238095	0.0714286	0.012987	0.00108225		
7	1	0.875	0.583333	0.291667	0.1060601	0.0265152	0.00407925	0.000291375	
8	1	0.888889	0.622222	0.339394	0.141414	0.043512	0.009324	0.0012432	0.0000777

The decoded signal can be normalised with the factor

$$g_{norm}(M) = \frac{2 \cdot (2M)!}{4^M \cdot (M!)^2}$$

M	1	2	3	4	5	6	7	8
$g_{norm}(M)$	1	0.75	0.625	0.546875	0.492188	0.451172	0.418945	0.392761

The illustration below shows a third-order B-format signal decoded to 13 loudspeakers first uncorrected (so-called basic decoding, left), then corrected by weighting (so-called in-phase decoding, right).



The following example shows in-phase decoding. The weights and norms up to 12th order are saved in the arrays *iWeight2D[]* and *iNorm2D[]* respectively. Instrument 11 decodes third order for 4 speakers in a square.

EXAMPLE 05B12_udc_ambisonics2D_3.csd

```

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

sr      =  44100
ksmps   =  32
nchnls  =  4
0dbfs   =  1

opcode  ambi2D_encode_n, 0, aik
asnd,iorder,kaz xin
kaz = $M_PI*kaz/180
kk = iorder
c1:
    zawm    cos(kk*kaz)*asnd,2*kk-1
    zawm    sin(kk*kaz)*asnd,2*kk
kk =
    kk-1

if      kk > 0 goto c1
    zawm    asnd,0

endop

;in-phase-decoding
opcode  ambi2D_dec_inph, a, ii
; weights and norms up to 12th order
iNorm2D[] array 1,0.75,0.625,0.546875,0.492188,0.451172,0.418945,
            0.392761,0.370941,0.352394,0.336376,0.322360
iWeight2D[][] init 12,12
iWeight2D   array 0.5,0,0,0,0,0,0,0,0,0,0,0,
            0.666667,0.166667,0,0,0,0,0,0,0,0,0,0,
            0.75,0.3,0.05,0,0,0,0,0,0,0,0,0,
            0.8,0.4,0.114286,0.0142857,0,0,0,0,0,0,0,0,
            0.833333,0.47619,0.178571,0.0396825,0.00396825,0,0,0,0,0,0,0,
            0.857143,0.535714,0.238095,0.0714286,0.012987,0.00108225,0,0,0,0,0,0,
            0.875,0.583333,0.291667,0.1060601,0.0265152,0.00407925,0.000291375,
            0,0,0,0,0,0.888889,0.622222,0.339394,0.141414,0.043512,
            0.009324,0.0012432,0.0000777,0,0,0,0,
            0.9,0.654545,0.381818,0.176224,0.0629371,0.0167832,0.00314685,
            0.000370218,0.0000205677,0,0,0,
            0.909091,0.681818,0.41958,0.20979,0.0839161,0.0262238,0.0061703,
            0.00102838,0.000108251,0.00000541254,0,0,
            0.916667,0.705128,0.453297,0.241758,0.105769,0.0373303,0.0103695,

```

```

0.00218306,0.000327459,0.0000311866,0.00000141757,0,
0.923077,0.725275,0.483516,0.271978,0.12799,0.0497738,0.015718,
0.00392951,0.000748478,0.000102065,0.00000887523,0.000000369801

iorder,iaz1      xin
iaz1 = $M_PI*iaz1/180
kk =    iorder
a1      =      .5*zar(0)
c1:
a1 +=  cos(kk*iaz1)*iWeight2D[iorder-1][kk-1]*zar(2*kk-1)
a1 +=  sin(kk*iaz1)*iWeight2D[iorder-1][kk-1]*zar(2*kk)
kk =      kk-1
if      kk > 0 goto c1
                  xout          iNorm2D[iorder-1]*a1
endop

zakinit 7, 1

instr 1
asnd   rand      p4
ares   reson     asnd,p5,p6,1
kaz    line      0,p3,p7*360 ;turns around p7 times in p3 seconds
                  ambi2D_encode_n asnd,3,kaz
endin

instr 11
a1      ambi2D_dec_inph 3,0
a2      ambi2D_dec_inph 3,90
a3      ambi2D_dec_inph 3,180
a4      ambi2D_dec_inph 3,270
          outc   a1,a2,a3,a4
          zacl   0,6           ; clear the za variables
endin

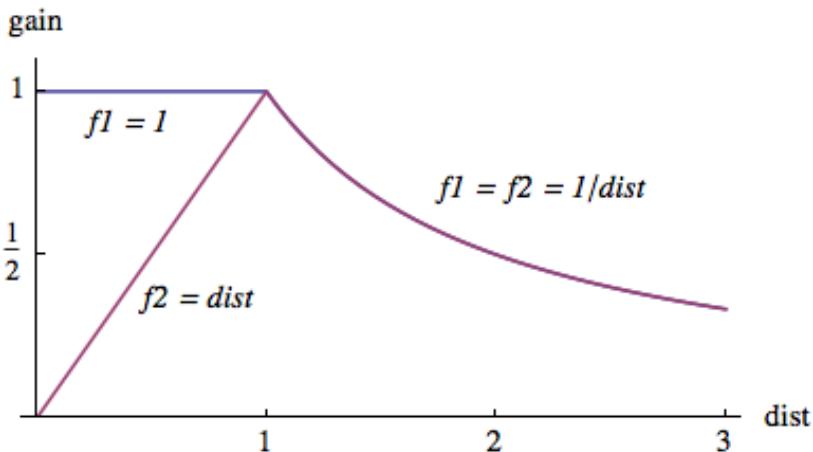
</CsInstruments>
<CsScore>
;           amp      cf      bw
i1 0 3 .1    1500   12      1           turns
i11 0 3
</CsScore>
</CsoundSynthesizer>
;example by martin neukom

```

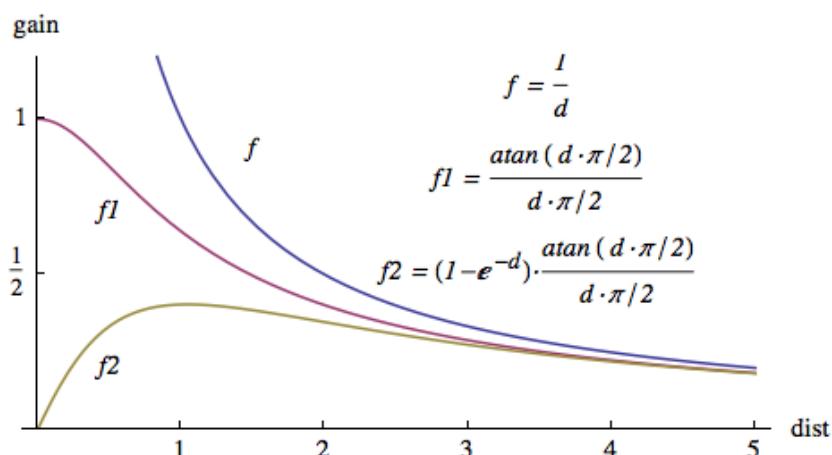
Distance

In order to simulate distances and movements of sound sources, the signals have to be treated before being encoded. The main perceptual cues for the distance of a sound source are reduction of the amplitude, filtering due to the absorption of the air and the relation between direct and indirect sound. We will implement the first two of these cues. The amplitude arriving at a listener is inversely proportional to the distance of the sound source. If the distance is larger than the unit circle (not necessarily the radius of the speaker setup, which does not need to be known when encoding sounds) we can simply divide the sound by the distance. With this calculation inside the unit circle the amplitude is amplified and becomes infinite when the distance becomes zero. Another problem arises when a virtual sound source passes the origin. The amplitude of the speaker signal in the direction of the movement suddenly becomes maximal and the signal of

the opposite speaker suddenly becomes zero. A simple solution for these problems is to limit the gain of the channel W inside the unit circle to 1 ($f1$ in the figure below) and to fade out all other channels ($f2$). By fading out all channels except channel W the information about the direction of the sound source is lost and all speaker signals are the same and the sum of the speaker signals reaches its maximum when the distance is 0.



Now, we are looking for gain functions that are smoother at $d = 1$. The functions should be differentiable and the slope of $f1$ at distance $d = 0$ should be 0. For distances greater than 1 the functions should be approximately $1/d$. In addition the function $f1$ should continuously grow with decreasing distance and reach its maximum at $d = 0$. The maximal gain must be 1. The function $\text{atan}(d \cdot \pi/2)/(d \cdot \pi/2)$ fulfills these constraints. We create a function $f2$ for the fading out of the other channels by multiplying $f1$ by the factor $(1 - e^{-d})$.



In the next example the UDO `ambi2D_enc_dist_n` encodes a sound at any order with distance correction. The inputs of the UDO are `asnd`, `iorder`, `kazimuth` and `kdistance`. If the distance becomes negative the azimuth angle is turned to its opposite ($\text{kaz} \pm \pi$) and the distance taken positive.

EXAMPLE 05B13_udo_ambisonics2D_4.csd

```
<CsoundSynthesizer>
<CsOptions>
--env:SSDIR+=./SourceMaterials -odac -m0
</CsOptions>
<CsInstruments>

sr      =  44100
```

```

ksmps    = 32
nchnls  = 8
0dbfs    = 1

#include "../SourceMaterials/ambisonics2D_udos.txt"

; distance encoding
; with any distance (includes zero and negative distance)

opcode    ambi2D_enc_dist_n, 0, aikk
asnd,iorder,kaz,kdist  xin
kaz = $M_PI*kaz/180
kaz   =      (kdist < 0 ? kaz + $M_PI : kaz)
kdist =      abs(kdist)+0.0001
kgainW  =      taninv(kdist*1.5707963) / (kdist*1.5708)           ;pi/2
kgainH0 =  (1 - exp(-kdist))*kgainW
kk = iorder
asndW   =      kgainW*asnd
asndH0  =      kgainH0*asndW
c1:
      zawm  cos(kk*kaz)*asndH0,2*kk-1
      zawm  sin(kk*kaz)*asndH0,2*kk
kk = kk-1

if      kk > 0 goto c1
      zawm  asndW,0

endop

zakinit 17, 1

instr 1
asnd  rand      p4
;asnd  soundin  "/Users/user/csound/ambisonic/violine.aiff"
kaz   line      0,p3,p5*360      ;turns around p5 times in p3 seconds
kdist  line      p6,p3,p7
      ambi2D_enc_dist_n asnd,8,kaz,kdist
endin

instr 10
a1,a2,a3,a4,
a5,a6,a7,a8      ambi2D_decode     8,0,45,90,135,180,225,270,315
      outc    a1,a2,a3,a4,a5,a6,a7,a8
      zacl    0,16
endin

</CsInstruments>
<CsScore>
f1 0 32768 10 1
;      amp turns dist1 dist2
i1 0 4    1    0    2    -2
;i1 0 4    1    1    1    1
i10 0 4
</CsScore>
</CsoundSynthesizer>
;example by martin neukom

```

In order to simulate the absorption of the air we introduce a very simple lowpass filter with a distance depending cutoff frequency. We produce a Doppler-shift with a distance dependent delay of the sound. Now, we have to determine our unit since the delay of the sound wave is calculated as distance divided by sound velocity. In our example *udo_ambisonics2D_5.csd* we set the unit to 1 meter. These procedures are performed before the encoding. In instrument 1 the movement

of the sound source is defined in Cartesian coordinates. The UDO `xy_to_ad` transforms them into polar coordinates. The B-format channels can be written to a sound file with the opcode `fout`. The UDO `write_ambi2D_2` writes the channels up to second order into a sound file.

EXAMPLE 05B14_udo_ambisonics2D_5.csd

```

<CsoundSynthesizer>
<CsOptions>
--env:SSDIR+=./SourceMaterials -odac -m0
</CsOptions>
<CsInstruments>
sr      = 44100
ksmps   = 32
nchnls  = 8
0dbfs   = 1

#include "../SourceMaterials/ambisonics2D_udos.txt"
#include "../SourceMaterials/ambisonics_utilities.txt" ;Absorb and Doppler

/* these opcodes are included in "ambisonics2D_udos.txt"
opcode xy_to_ad, kk, kk
kx,ky      xin
kdist =    sqrt(kx*kx+ky*ky)
kaz        taninv2   ky,kx
            xout      180*kaz/$M_PI, kdist
endop

opcode Absorb, a, ak
asnd,kdist  xin
aabs       tone      5*asnd,20000*exp(-.1*kdist)
            xout      aabs
endop

opcode Doppler, a, ak
asnd,kdist  xin
abuf       delayr   .5
adop       deltapi   interp(kdist)*0.0029137529 + .01 ; 1/343.2
            delayw   asnd
            xout      adop
endop
*/
opcode write_ambi2D_2, 0,      S
Sname      xin
fout      Sname,12,zar(0),zar(1),zar(2),zar(3),zar(4)
endop

zakinit 17, 1           ; zak space with the 17 channels of the B-format

instr 1
asnd   buzz    p4,p5,50,1
;asnd   soundin "/Users/user/csound/ambisonic/violine.aiff"
kx     line    p7,p3,p8
ky     line    p9,p3,p10
kaz,kdist xy_to_ad kx,ky
aabs   absorb  asnd,kdist
adop   Doppler .2*aabs,kdist
            ambi2D_enc_dist adop,5,kaz,kdist
endin

instr 10          ;decode all instruments
a1,a2,a3,a4,
a5,a6,a7,a8      ambi2D_dec_inph 5,0,45,90,135,180,225,270,315

```

```

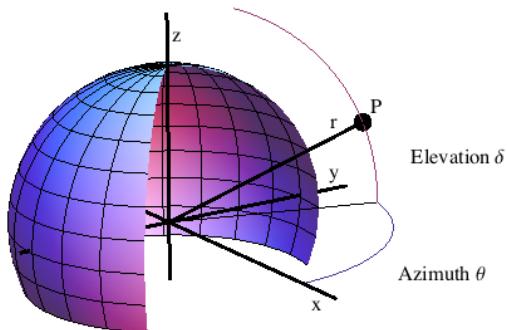
        outc      a1,a2,a3,a4,a5,a6,a7,a8
;        fout "B_format2D.wav",12,zar(0),zar(1),zar(2),zar(3),zar(4),
;                  zar(5),zar(6),zar(7),zar(8),zar(9),zar(10)
        write_ambi2D_2 "ambi_ex5.wav"
        zacl      0,16 ; clear the za variables
endin

</CsInstruments>
<CsScore>
f1 0 32768 10 1
;           amp          f       0      0      x1      x2      y1      y2
i1 0 5     .8   200      0      40    -20      1     .1
i10 0 5
</CsScore>
</CsoundSynthesizer>
;example by martin neukom

```

Adding third dimension

The position of a point in space can be given by its Cartesian coordinates x , y and z or by its spherical coordinates the radial distance r from the origin of the coordinate system, the elevation δ (which lies between $-\pi$ and π) and the azimuth angle θ .



The formulae for transforming coordinates are as follows:

$$x = r \cdot \cos(\delta) \cdot \cos(\theta) \quad y = r \cdot \cos(\delta) \cdot \sin(\theta) \quad z = r \cdot \sin(\delta)$$

$$r = \sqrt{x^2 + y^2 + z^2} \quad \theta = \arctan(y/x) \quad \delta = \arccos\left(\frac{\sqrt{x^2 + y^2}}{z}\right)$$

The channels of the Ambisonic B-format are computed as the product of the sounds themselves and the so-called spherical harmonics representing the direction to the virtual sound sources. The spherical harmonics can be normalised in various ways. We shall use the so-called semi-normalised spherical harmonics. The following table shows the encoding functions up to the third order as function of azimuth and elevation $Y_{mn}(\theta, \delta)$ and as function of x , y and z $Y_{mn}(x, y, z)$ for sound sources on the unit sphere. The decoding formulae for symmetrical speaker setups are the same.

m	n	$Y_{mn}(\theta, \delta)$	$Y_{mn}(x, y, z)$
1	0	$\text{Sin}[\delta]$	z
1	1	$\text{Cos}[\delta] \text{Cos}[\theta]$	x
-1	1	$\text{Cos}[\delta] \text{Sin}[\theta]$	y
2	0	$\frac{1}{2} (-1 + 3 \text{Sin}[\delta]^2)$	$\frac{1}{2} (-1 + 3 z^2)$
1	1	$\frac{1}{2} \sqrt{3} \text{Cos}[\delta] \text{Sin}[2\delta]$	$\sqrt{3} x z$
-1	1	$\frac{1}{2} \sqrt{3} \text{Sin}[2\delta] \text{Sin}[\theta]$	$\sqrt{3} y z$
2	2	$\frac{1}{2} \sqrt{3} \text{Cos}[\delta]^2 \text{Cos}[2\theta]$	$\frac{1}{2} (\sqrt{3} x^2 - \sqrt{3} y^2)$
-2	2	$\sqrt{3} \text{Cos}[\delta]^2 \text{Cos}[\theta] \text{Sin}[\theta]$	$\sqrt{3} x y$
3	0	$\frac{1}{8} (3 \text{Sin}[\delta] - 5 \text{Sin}[3\delta])$	$\frac{1}{2} z (-3 + 5 z^2)$
1	1	$\frac{1}{8} \sqrt{\frac{3}{2}} (\text{Cos}[\delta] - 5 \text{Cos}[3\delta]) \text{Cos}[\theta]$	$\frac{1}{4} (-\sqrt{6} x + 5 \sqrt{6} x z^2)$
-1	1	$\frac{1}{8} \sqrt{\frac{3}{2}} (\text{Cos}[\delta] - 5 \text{Cos}[3\delta]) \text{Sin}[\theta]$	$\frac{1}{4} (-\sqrt{6} y + 5 \sqrt{6} y z^2)$
2	2	$\frac{1}{2} \sqrt{15} \text{Cos}[\delta]^2 \text{Cos}[2\theta] \text{Sin}[\delta]$	$\frac{1}{2} (\sqrt{15} z - 2 \sqrt{15} y^2 z - \sqrt{15} x^2)$
-2	2	$\sqrt{15} \text{Cos}[\delta]^2 \text{Cos}[\theta] \text{Sin}[\delta] \text{Sin}[\theta]$	$\sqrt{15} x y z$
3	1	$\frac{1}{2} \sqrt{\frac{3}{2}} \text{Cos}[\delta]^3 \text{Cos}[3\theta]$	$\frac{1}{4} (\sqrt{10} x^3 - 3 \sqrt{10} x y^2)$
-3	1	$\frac{1}{2} \sqrt{\frac{3}{2}} \text{Cos}[\delta]^3 \text{Sin}[3\theta]$	$\frac{1}{4} (3 \sqrt{10} x^2 y - \sqrt{10} y^3)$

In the first three of the following examples we will not produce sound but display in number boxes (for example using CsoundQt widgets) the amplitude of 3 speakers at positions (1, 0, 0), (0, 1, 0) and (0, 0, 1) in Cartesian coordinates. The position of the sound source can be changed with the two scroll numbers. The example *udo_ambisonics_1.csd* shows encoding up to second order. The decoding is done in two steps. First we decode the B-format for one speaker. In the second step, we create a overloaded opcode for n speakers. The number of output signals determines which version of the opcode is used. The UDOs *ambi_encode* and *ambi_decode* up to 8th order are saved in the text file *ambisonics_udos.txt*.

EXAMPLE 05B15_udo_ambisonics_1.csd

```

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

zakinit 9, 1      ; zak space with the 9 channel B-format second order

opcode  ambi_encode, 0, aikk
asnd,iorder,kaz,kel    xin
kaz = $M_PI*kaz/180
kel = $M_PI*kel/180
kcos_el = cos(kel)
ksin_el = sin(kel)
kcos_az = cos(kaz)
ksin_az = sin(kaz)

zawm    asnd,0
zawm    kcos_el*ksin_az*asnd,1          ; Y      = Y(1,-1) ; W
zawm    ksin_el*asnd,2                  ; Z      = Y(1,0)
zawm    kcos_el*kcos_az*asnd,3          ; X      = Y(1,1)

if           iorder < 2 goto      end

i2      = sqrt(3)/2
kcos_el_p2 = kcos_el*kcos_el
ksin_el_p2 = ksin_el*ksin_el
kcos_2az = cos(2*kaz)

```

```

ksin_2az = sin(2*kaz)
kcos_2el = cos(2*kel)
ksin_2el = sin(2*kel)

zawm i2*kcos_el_p2*ksin_2az*asnd,4 ; V = Y(2,-2)
zawm i2*ksin_2el*ksin_az*asnd,5 ; S = Y(2,-1)
zawm .5*(3*ksin_el_p2 - 1)*asnd,6 ; R = Y(2,0)
zawm i2*ksin_2el*kcos_az*asnd,7 ; S = Y(2,1)
zawm i2*kcos_el_p2*kcos_2az*asnd,8 ; U = Y(2,2)
end:

endop

; decoding of order iorder for 1 speaker at position iaz,iel,idist
opcode aibi_decode1, a, iii
iorder,iaz,iel xin
iaz = $M_PI*iaz/180
iel = $M_PI*iel/180
a0=zar(0)
    if      iorder > 0 goto c0
aout = a0
    goto    end
c0:
a1=zar(1)
a2=zar(2)
a3=zar(3)
icos_el = cos(iel)
isin_el = sin(iel)
icos_az = cos(iaz)
isin_az = sin(iaz)
i1      = icos_el*isin_az ; Y      = Y(1,-1)
i2      = isin_el          ; Z      = Y(1,0)
i3      = icos_el*icos_az ; X      = Y(1,1)
    if iorder > 1 goto c1
aout   = (1/2)*(a0 + i1*a1 + i2*a2 + i3*a3)
    goto end
c1:
a4=zar(4)
a5=zar(5)
a6=zar(6)
a7=zar(7)
a8=zar(8)
ic2    = sqrt(3)/2

icos_el_p2 = icos_el*icos_el
isin_el_p2 = isin_el*isin_el
icos_2az = cos(2*iaz)
isin_2az = sin(2*iaz)
icos_2el = cos(2*iel)
isin_2el = sin(2*iel)

i4 = ic2*icos_el_p2*isin_2az ; V = Y(2,-2)
i5      = ic2*isin_2el*isin_el ; S = Y(2,-1)
i6 = .5*(3*isin_el_p2 - 1) ; R = Y(2,0)
i7 = ic2*isin_2el*icos_az ; S = Y(2,1)
i8 = ic2*icos_el_p2*icos_2az ; U = Y(2,2)

aout = (1/9)*(a0 + 3*i1*a1 + 3*i2*a2 + 3*i3*a3 + 5*i4*a4 + \
            5*i5*a5 + 5*i6*a6 + 5*i7*a7 + 5*i8*a8)

end:
           xout           aout

```

```

endop

; overloaded opcode for decoding of order iorder
; speaker positions in function table ifn
opcode ambi_decode,    a,ii
iorder,ifn xin
  xout ambi_decode1(iorder,table(1,ifn),table(2,ifn))
endop
opcode ambi_decode,    aa,ii
iorder,ifn xin
  xout ambi_decode1(iorder,table(1,ifn),table(2,ifn)),
      ambi_decode1(iorder,table(3,ifn),table(4,ifn))
endop
opcode ambi_decode,    aaa,ii
iorder,ifn xin
  xout ambi_decode1(iorder,table(1,ifn),table(2,ifn)),
      ambi_decode1(iorder,table(3,ifn),table(4,ifn)),
      ambi_decode1(iorder,table(5,ifn),table(6,ifn))
endop

instr 1
asnd    init          1
;kdist  init          1
kaz     invalue "az"
kel     invalue "el"

      ambi_encode asnd,2,kaz,kel

ao1,ao2,ao3    ambi_decode   2,17
                outvalue "sp1", downsample(ao1)
                outvalue "sp2", downsample(ao2)
                outvalue "sp3", downsample(ao3)
                zacl   0,8
endin

</CsInstruments>
<CsScore>
;f1 0 1024 10 1
f17 0 64 -2 0  0 0   90 0   0 90   0 0   0 0   0 0
i1 0 100
</CsScore>
</CsoundSynthesizer>
;example by martin neukom

```

The next example shows in-phase decoding. The weights up to 8th order are stored in the array *iWeight3D[]*.

EXAMPLE 05B16_udo_ambisonics_2.cs

```

<CsoundSynthesizer>
<CsOptions>
--env:SSDIR+=./SourceMaterials -odac -m0
</CsOptions>
<CsInstruments>
sr      = 44100
ksmps   = 32
nchnls  = 1
0dbfs   = 1

zakinit 81, 1 ; zak space for up to 81 channels of the 8th order B-format

```

```

; the opcodes used below are safed in "ambisonics_udos.txt"
#include "../SourceMaterials/ambisonics_udos.txt"

; in-phase decoding up to third order for one speaker
opcode    ambi_dec1_inph3, a, iii
; weights up to 8th order
iWeight3D[][] init  8,8
iWeight3D    array  0.333333,0,0,0,0,0,0,0,
              0.5,0.1,0,0,0,0,0,0,
              0.6,0.2,0.0285714,0,0,0,0,0,
              0.666667,0.285714,0.0714286,0.0079365,0,0,0,0,
              0.714286,0.357143,0.119048,0.0238095,0.0021645,0,0,0,
              0.75,0.416667,0.166667,0.0454545,0.00757576,0.00058275,0,0,
              0.777778,0.466667,0.212121,0.0707071,0.016317,0.002331,0.0001554,0,
              0.8,0.509091,0.254545,0.0979021,0.027972,0.0055944,0.0006993,0.00004114

iorder,iaz,iel    xin
iaz = $M_PI*iaz/180
iel = $M_PI*iel/180
a0=zar(0)
    if    iorder > 0 goto c0
aout = a0
    goto    end
c0:
a1=iWeight3D[iorder-1][0]*zar(1)
a2=iWeight3D[iorder-1][0]*zar(2)
a3=iWeight3D[iorder-1][0]*zar(3)
icos_el = cos(iel)
isin_el = sin(iel)
icos_az = cos(iaz)
isin_az = sin(iaz)
i1    =    icos_el*isin_az           ; Y      = Y(1,-1)
i2    =    isin_el                  ; Z      = Y(1,0)
i3    =    icos_el*icos_az          ; X      = Y(1,1)
    if iorder > 1 goto c1
aout   =  (3/4)*(a0 + i1*a1 + i2*a2 + i3*a3)
    goto end
c1:
a4=iWeight3D[iorder-1][1]*zar(4)
a5=iWeight3D[iorder-1][1]*zar(5)
a6=iWeight3D[iorder-1][1]*zar(6)
a7=iWeight3D[iorder-1][1]*zar(7)
a8=iWeight3D[iorder-1][1]*zar(8)

ic2    = sqrt(3)/2

icos_el_p2 = icos_el*icos_el
isin_el_p2 = isin_el*isin_el
icos_2az = cos(2*iaz)
isin_2az = sin(2*iaz)
icos_2el = cos(2*iel)
isin_2el = sin(2*iel)

i4 = ic2*icos_el_p2*isin_2az    ; V = Y(2,-2)
i5 = ic2*isin_2el*isin_el      ; S = Y(2,-1)
i6 = .5*(3*isin_el_p2 - 1)      ; R = Y(2,0)
i7 = ic2*isin_2el*icos_az      ; S = Y(2,1)
i8 = ic2*icos_el_p2*icos_2az    ; U = Y(2,2)
aout = (1/3)*(a0 + 3*i1*a1 + 3*i2*a2 + 3*i3*a3 + 5*i4*a4 + 5*i5*a5 + \
            5*i6*a6 + 5*i7*a7 + 5*i8*a8)

end:
        xout          aout

```

```

endop

; overloaded opcode for decoding for 1 or 2 speakers
; speaker positions in function table ifn
opcode    ambi_dec2_inph,    a,ii
iorder,ifn xin
        xout      ambi_dec1_inph(iorder,table(1,ifn),table(2,ifn))
endop
opcode    ambi_dec2_inph,    aa,ii
iorder,ifn xin
        xout      ambi_dec1_inph(iorder,table(1,ifn),table(2,ifn)),
                ambi_dec1_inph(iorder,table(3,ifn),table(4,ifn))
endop
opcode    ambi_dec2_inph,    aaa,ii
iorder,ifn xin
        xout      ambi_dec1_inph(iorder,table(1,ifn),table(2,ifn)),
                ambi_dec1_inph(iorder,table(3,ifn),table(4,ifn)),
                ambi_dec1_inph(iorder,table(5,ifn),table(6,ifn))
endop

instr 1
asnd    init      1
kdist   init      1
kaz     invalue   "az"
kel     invalue   "el"

        ambi_encode asnd,8,kaz,kel
ao1,ao2,ao3 ambi_dec_inph 8,17
        outvalue  "sp1", downsample(ao1)
        outvalue  "sp2", downsample(ao2)
        outvalue  "sp3", downsample(ao3)
        zacl      0,80
endin

</CsInstruments>
<CsScore>
f1 0 1024 10 1
f17 0 64 -2 0  0 0   90 0   0 90  0 0   0 0  0 0  0 0  0 0
i1 0 100
</CsScore>
</CsoundSynthesizer>
;example by martin neukom

```

The weighting factors for in-phase decoding of Ambisonics (3D) are:

<i>M</i>		<i>g</i> ₁	<i>g</i> ₂	<i>g</i> ₃	<i>g</i> ₄	<i>g</i> ₅	<i>g</i> ₆	<i>g</i> ₇	<i>g</i> ₈
1	1	0.333333							
2	1	0.5	0.1						
3	1	0.6	0.2	0.0285714					
4	1	0.666667	0.285714	0.0714286	0.00793651				
5	1	0.714286	0.357143	0.119048	0.0238095	0.0021645			
6	1	0.75	0.416667	0.166667	0.0454545	0.00757576	0.000582751		
7	1	0.777778	0.466667	0.212121	0.0707071	0.016317	0.002331	0.0001554	
8	1	0.8	0.509091	0.254545	0.0979021	0.027972	0.00559441	0.000699301	0.000041135

The following example shows distance encoding.

EXAMPLE 05B17_udo_ambisonics_3.csd

```

<CsoundSynthesizer>
<CsOptions>
--env:SSDIR+=./SourceMaterials -odac -m0
</CsOptions>
<CsInstruments>
sr      = 44100
ksmps   = 32
nchnls  = 2
0dbfs   = 1

zakinit 81, 1           ; zak space with the 11 channels of the B-format

#include "../SourceMaterials/ambisonics_udos.txt"

opcode  ambi3D_enc_dist1, 0, aikkk
asnd,iorder,kaz,kel,kdist    xin
kaz = $M_PI*kaz/180
kel = $M_PI*kel/180
kaz   =      (kdist < 0 ? kaz + $M_PI : kaz)
kel   =      (kdist < 0 ? -kel : kel)
kdist = abs(kdist)+0.00001
kgainW  = taninv(kdist*1.5708) / (kdist*1.5708)
kgainHO = (1 - exp(-kdist)) ;*kgainW
    outvalue "kgainHO", kgainHO
    outvalue "kgainW", kgainW
kcos_el = cos(kel)
ksin_el = sin(kel)
kcos_az = cos(kaz)
ksin_az = sin(kaz)
asnd = kgainW*asnd
    zawm   asnd,0                      ; W
asnd = kgainHO*asnd
    zawm   kcos_el*ksin_az*asnd,1      ; Y      = Y(1,-1)
    zawm   ksin_el*asnd,2                ; Z      = Y(1,0)
    zawm   kcos_el*kcos_az*asnd,3      ; X      = Y(1,1)
    if     iorder < 2 goto    end
/*
...
*/
end:
endop

instr 1
asnd   init    1
kaz    invalue "az"
kel    invalue "el"
kdist  invalue "dist"
        ambi_enc_dist asnd,5,kaz,kel,kdist
ao1,ao2,ao3,ao4 ambi_decode 5,17
        outvalue "sp1", downsample(ao1)
        outvalue "sp2", downsample(ao2)
        outvalue "sp3", downsample(ao3)
        outvalue "sp4", downsample(ao4)
        outc   0*ao1,0*ao2;,2*ao3,2*ao4
        zacl   0,80
endin
</CsInstruments>
<CsScore>
f17 0 64 -2 0  0 0  90 0   180 0      0 90  0 0   0 0
i1 0 100

```

```
</CsScore>
</CsoundSynthesizer>
;example by martin neukom
```

In example *udo_ambisonics_4.csd* a buzzer with the three-dimensional trajectory shown below is encoded in third order and decoded for a speaker setup in a cube (f17).

EXAMPLE 05B18_udo_ambisonics_4.csd

```
<CsoundSynthesizer>
<CsOptions>
--env:SSDIR=../SourceMaterials -odac -m0
</CsOptions>
<CsInstruments>
sr      = 44100
ksmps   = 32
nchnls  = 8
0dbfs   = 1

zakinit 16, 1

#include "../SourceMaterials/ambisonics_udos.txt"
#include "../SourceMaterials/ambisonics_utilities.txt"

instr 1
asnd    buzz    p4,p5,p6,1
kt      line    0,p3,p3
kaz,kel,kdist xyz_to_aed 10*sin(kt),10*sin(.78*kt),10*sin(.43*kt)
adop Doppler asnd,kdist
          ambi_enc_dist adop,3,kaz,kel,kdist
a1,a2,a3,a4,a5,a6,a7,a8 ambi_decode 3,17
;k0      ambi_write_B  "B_form.wav",8,14
          outc    a1,a2,a3,a4,a5,a6,a7,a8
          zacl    0,15
endin

</CsInstruments>
<CsScore>
f1 0 32768 10 1
f17 0 64 -2 0 -45 35.2644 45 35.2644 135 35.2644 225 35.2644 \
-45 -35.2644 .7854 -35.2644 135 -35.2644 225 -35.2644
i1 0 40 .5 300 40
</CsScore>
</CsoundSynthesizer>
;example by martin neukom
```

Ambisonics Equivalent Panning (AEP)

If we combine encoding and in-phase decoding, we obtain the following panning function (a gain function for a speaker depending on its distance to a virtual sound source):

$$P(\gamma, m) = \left(\frac{1}{2} + \frac{1}{2} \cos \gamma\right)^m$$

where γ denotes the angle between a sound source and a speaker and m denotes the order. If the speakers are positioned on a unit sphere the cosine of the angle γ is calculated as the scalar product of the vector to the sound source (x, y, z) and the vector to the speaker (x_s, y_s, z_s) .

In contrast to Ambisonics the order indicated in the function does not have to be an integer. This means that the order can be continuously varied during decoding. The function can be used in both Ambisonics and Ambisonics2D.

This system of panning is called Ambisonics Equivalent Panning. It has the disadvantage of not producing a B-format representation, but its implementation is straightforward and the computation time is short and independent of the Ambisonics order simulated. Hence it is particularly useful for real-time applications, for panning in connection with sequencer programs and for experimentation with high and non-integral Ambisonic orders.

The opcode *AEP1* in the next example shows the calculation of ambisonics equivalent panning for one speaker. The opcode *AEP* then uses *AEP1* to produce the signals for several speakers. In the text file *AEP_udos.txt* *AEP* is implemented for up to 16 speakers. The position of the speakers must be written in a function table. As the first parameter in the function table the maximal speaker distance must be given.

EXAMPLE 05B19_udc_AEP.csd

```
<CsoundSynthesizer>
<CsOptions>
-odac
</CsOptions>
<CsInstruments>
sr      =  44100
ksmps   =  32
nchnls  =  4
0dbfs   =  1

;#include "ambisonics_udos.txt"

; opcode AEP1 is the same as in udo_AEP_xyz.csd

opcode  AEP1, a, akiiiikkkkkk
    ; soundin, order, ixs, iys, izs, idsmx, kx, ky, kz
ain,korder,ixs,iys,izs,idsmx,kx,ky,kz,kdist,kfade,kgain xin
idists = sqrt(ixs*ixs+iys*iys+izs*izs)
kpan = kgain*((1-kfade+kfade*(kx*ixs+ky*iys+kz*izs)/(kdist*idists))^korder)
    xout ain*kpan*idists/idsmx
endop

; opcode AEP calculates ambisonics equivalent panning for n speaker
; the number n of output channels defines the number of speakers
; inputs: sound ain, order korder (any real number >= 1)
; ifn = number of the function containing the speaker positions
; position and distance of the sound source kaz,kel,kdist in degrees

opcode AEP, aaaa, akikk
ain,korder,ifn,kaz,kel,kdist      xin
kaz = $M_PI*kaz/180
kel = $M_PI*kel/180
kx = kdist*cos(kel)*cos(kaz)
ky = kdist*cos(kel)*sin(kaz)
kz = kdist*sin(kel)
ispeaker[] array 0,
table(3,ifn)*cos((M_PI/180)*table(2,ifn))*cos((M_PI/180)*table(1,ifn)),
table(3,ifn)*cos((M_PI/180)*table(2,ifn))*sin((M_PI/180)*table(1,ifn)),
table(3,ifn)*sin((M_PI/180)*table(2,ifn)),
table(6,ifn)*cos((M_PI/180)*table(5,ifn))*cos((M_PI/180)*table(4,ifn)),
table(6,ifn)*cos((M_PI/180)*table(5,ifn))*sin((M_PI/180)*table(4,ifn)),
```

```

table(6,ifn)*sin((\$M_PI/180)*table(5,ifn)),
table(9,ifn)*cos((\$M_PI/180)*table(8,ifn))*cos((\$M_PI/180)*table(7,ifn)),
table(9,ifn)*cos((\$M_PI/180)*table(8,ifn))*sin((\$M_PI/180)*table(7,ifn)),
table(9,ifn)*sin((\$M_PI/180)*table(8,ifn)),
table(12,ifn)*cos((\$M_PI/180)*table(11,ifn))*
    cos((\$M_PI/180)*table(10,ifn)),
table(12,ifn)*cos((\$M_PI/180)*table(11,ifn))*
    sin((\$M_PI/180)*table(10,ifn)),
table(12,ifn)*sin((\$M_PI/180)*table(11,ifn))

idsmax  table  0,ifn
kdist   =      kdist+0.000001
kfade   =      .5*(1 - exp(-abs(kdist)))
kgain   =      taninv(kdist*1.5708)/(kdist*1.5708)

a1      AEP1   ain,korder,ispeaker[1],ispeaker[2],ispeaker[3],
          idsmax,kx,ky,kz,kdist,kfade,kgain
a2      AEP1   ain,korder,ispeaker[4],ispeaker[5],ispeaker[6],
          idsmax,kx,ky,kz,kdist,kfade,kgain
a3      AEP1   ain,korder,ispeaker[7],ispeaker[8],ispeaker[9],
          idsmax,kx,ky,kz,kdist,kfade,kgain
a4      AEP1   ain,korder,ispeaker[10],ispeaker[11],ispeaker[12],
          idsmax,kx,ky,kz,kdist,kfade,kgain
xout    a1,a2,a3,a4
endop

instr 1
ain      rand   1
;ain      soundin "/Users/user/csound/ambisonic/violine.aiff"
kt       line   0,p3,360
korder  init   24
;kdist  Dist kx, ky, kz
a1,a2,a3,a4 AEP  ain,korder,17,kt,0,1
          outc   a1,a2,a3,a4
endin

</CsInstruments>
<CsScore>

;function for speaker positions
; GEN -2, parameters: max_speaker_distance, xs1,ys1,zs1,xs2,ys2,zs2, ...
;octahedron
;f17 0 32 -2 1 1 0 0  -1 0 0  0 1 0  0 -1 0  0 0 1  0 0 -1
;cube
;f17 0 32 -2 1,732 1 1 1  1 1 -1  1 -1 1  -1 1 1
;octagon
;f17 0 32 -2 1 0.924 -0.383 0 0.924 0.383 0 0.383 0.924 0 -0.383 0.924 0
;-0.924 0.383 0 -0.924 -0.383 0 -0.383 -0.924 0 0.383 -0.924 0
;f17 0 32 -2 1 0 0 1 45 0 1 90 0 1 135 0 1 180 0 1 225 0 1 270 0 1 315 0 1
;f17 0 32 -2 1 0 -90 1 0 -70 1 0 -50 1 0 -30 1 0 -10 1 0 10 1 0 30 1 0 50 1
f17 0 32 -2 1  -45 0 1  45 0 1  135 0 1  225 0 1
i1 0 2

</CsScore>
<CsoundSynthesizer>
;example by martin neukom

```

Summary of the Ambisonics UDOs

```

zakinits isizea, isizek
  (isizea = (order + 1)^2 in ambisonics (3D);
  isizea = 2·order + 1 in ambi2D; isizek = 1)

```

ambisonics_udos.txt (order <= 8)

```

ambi_encode asnd, iorder, kazimuth, kelevation
  (azimuth, elevation in degrees)
ambi_enc_dist asnd, iorder, kazimuth, kelevation, kdistance
a1 [, a2] ... [, a8] ambi_decode iorder, ifn
a1 [, a2] ... [, a8] ambi_dec_inph iorder, ifn
f ifn 0 n -2 p1 az1 el1 az2 el2 ...
  (n is a power of 2 greater than 3·number_of_spekers + 1)
  (p1 is not used)
ambi_write_B "name", iorder, ifile_format
  (ifile_format see fout in the csound help)
ambi_read_B "name", iorder (only <= 5)
kaz, kel, kdist xyz_to_aed kx, ky, kz

```

ambisonics2D_udos.txt

```

ambi2D_encode asnd, iorder, kazimuth (any order) (azimuth in degrees)
ambi2D_enc_dist asnd, iorder, kazimuth, kdistance
a1 [, a2] ... [, a8] ambi2D_decode iorder, iaz1 [, iaz2] ... [, iaz8]
a1 [, a2] ... [, a8] ambi2D_dec_inph iorder, iaz1 [, iaz2] ... [, iaz8]
  (order <= 12)
ambi2D_write_B "name", iorder, ifile_format
ambi2D_read_B "name", iorder (order <= 19)
kaz, kdist xy_to_ad kx, ky

```

AEP_udos.txt (any order integer or fractional)

```

a1 [, a2] ... [, a16] AEP_xyz asnd, korder, ifn, kx, ky, kz, kdistance
f ifn 0 64 -2 max Speaker Distance x1 y1 z1 x2 y2 z2 ...
a1 [, a2] ... [, a8] AEP asnd, korder, ifn, kazimuth, kelevation,
  kdistance (azimuth, elevation in degrees)
f ifn 0 64 -2 max Speaker Distance az1 el1 dist1 az2 el2 dist2 ...
  (azimuth, elevation in degrees)

```

ambi_utilities.txt

```

kdist dist kx, ky
kdist dist kx, ky, kz
ares Doppler asnd, kdistance
ares absorb asnd, kdistance
kx, ky, kz aed_to_xyz kazimuth, kelevation, kdistance
ix, iy, iz aed_to_xyz iazimuth, ielevation, idistance
a1 [, a2] ... [, a16] dist_corr a1 [, a2] ... [, a16], ifn
f ifn 0 32 -2 max Speaker Distance dist1, dist2, ... (distances in m)
irad radiani idegree
krad radian kdegree
arad radian adegree
idegree degreei irad
kdegree degree krad
adegree degree arad

```


05 C. 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 specialised 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 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_tone_butlp_moogladder.csd

```
<CsoundSynthesizer>
<CsOptions>
-odac ; activates real time sound output
```

```

</CsOptions>
<CsInstruments>
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
</CsScore>
</CsoundSynthesizer>
;example by Iain McCurdy

```

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_atone_buthp_bqrez.csd

```

<CsoundSynthesizer>
<CsOptions>
-odac ; activates real time sound output
</CsOptions>
<CsInstruments>
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)
</CsScore>
</CsoundSynthesizer>
;example by Iain McCurdy

```

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_reson_butbp.csd

```

<CsoundSynthesizer>
<CsOptions>
-odac ; activates real time sound output
</CsOptions>
<CsInstruments>

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

instr 1
    prints      "reson%"          ; 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%"          ; 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%"          ; 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%"          ; 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>
;example by Iain McCurdy
```

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 $1/\text{loop time}$. Loop time and the frequencies of the resonance peaks will be inversely proportional – as loop time gets 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_comb.csd

```
<CsoundSynthesizer>
<CsOptions>
-odac ;activates real time sound output
</CsOptions>
<CsInstruments>

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
</CsScore>
</CsoundSynthesizer>
;example by Iain McCurdy

```

Other Filters Worth Investigating

In addition to a wealth of low and highpass filters, Csound offers several more unique filters. Multimode such as [bqrez](#) provide several different filter types within a single opcode. Filter type is normally chosen using an i-rate input argument that functions like a switch. Another multimode filter, [clfil](#), offers additional filter controls such as *filter design* and *number of poles* to create unusual sound filters. unfortunately some parts of this opcode are not implemented yet.

[eqfil](#) is essentially a parametric equaliser but multiple iterations could be used as modules in a graphic equaliser bank. In addition to the capabilities of eqfil, [pareq](#) adds the possibility of creating low and high shelving filtering which might prove useful in mastering or in spectral adjustment of more developed sounds.

[rbjeq](#) offers a quite comprehensive multimode filter including highpass, lowpass, bandpass, bandreject, peaking, low-shelving and high-shelving, all in a single opcode.

[statevar](#) offers the outputs from four filter types - highpass, lowpass, bandpass and bandreject - simultaneously so that the user can morph between them smoothly. [svfilter](#) does a similar thing but with just highpass, lowpass and bandpass filter types.

[phaser1](#) and [phaser2](#) offer algorithms containing chains of first order and second order allpass filters respectively. These algorithms could conceivably be built from individual allpass filters, but these ready-made versions provide convenience and added efficiency.

[hilbert](#) is a specialist IIR filter that implements the Hilbert transformer.

For those wishing to devise their own filter using coefficients Csound offers [filter2](#) and [zfilter2](#).

Filter Comparision

The following example shows a nice comparision between a number of common used filters.

EXAMPLE 05C05_filter_compar.csd

```

<CsoundSynthesizer>
<CsOptions>

```

```

-odac -m128
</CsOptions>
<CsInstruments>
sr = 44100
ksmps = 64
nchnls = 2
0dbfs = 1

gaOut init 0
giSpb init 0.45

; Filter types
#define MOOG_LADDER #1#
#define MOOG_VCF      #2#
#define LPF18        #3#
#define BQREZ         #4#
#define CLFILT        #5#
#define BUTTERLP      #6#
#define LOWRES        #7#
#define REZZY         #8#
#define SVFILTER      #9#
#define VLOWRES       #10#
#define STATEVAR      #11#
#define MVCLPF1       #12#
#define MVCLPF2       #13#
#define MVCLPF3       #14#

opcode Echo, 0, S
Smsg xin
    printf_i "\n%s\n\n", 1, Smsg
endop

opcode EchoFilterName, 0, i
iType xin

if iType == $MOOG_LADDER then
    Echo "moogladder"
elseif iType == $MOOG_VCF then
    Echo "moogvcf"
elseif iType == $LPF18 then
    Echo "lpf18"
elseif iType == $BQREZ then
    Echo "bqrez"
elseif iType == $CLFILT then
    Echo "clfilt"
elseif iType == $BUTTERLP then
    Echo "butterlp"
elseif iType == $LOWRES then
    Echo "lowres"
elseif iType == $REZZY then
    Echo "rezzy"
elseif iType == $SVFILTER then
    Echo "svfilter"
elseif iType == $VLOWRES then
    Echo "vlowres"
elseif iType == $STATEVAR then
    Echo "statevar"
elseif iType == $MVCLPF1 then
    Echo "mvclpf1"
elseif iType == $MVCLPF2 then
    Echo "mvclpf2"
elseif iType == $MVCLPF3 then
    Echo "mvclpf3"

```

```

else
endif
endop

opcode MultiFilter, a, akki
ain, kcfq, kres, iType xin

kType init iType
if kType == $MOOG_LADDER then
    aout    moogladder ain, kcfq, kres
elseif kType == $MOOG_VCF then
    aout    moogvcf ain, kcfq, kres
elseif kType == $LPF18 then
    aout    lpf18 ain, kcfq, kres, 0.5
elseif kType == $BQREZ then
    aout    bqrez ain, kcfq, 99 * kres + 1
elseif kType == $CLFILT then
    aout    clfilt ain, kcfq, 0, 2
elseif kType == $BUTTERLP then
    aout    butterlp ain, kcfq
elseif kType == $LOWRES then
    aout    lowres ain, kcfq, kres
elseif kType == $REZZY then
    aout    rezzy ain, kcfq, kres
elseif kType == $SVFILTER then
    aout, ahig, aband svfilter ain, kcfq, (499 / 10) * kres + 1 ; rescale
elseif kType == $VLOWRES then
    aout    vlowres ain, kcfq, kres, 2, 0
elseif kType == $STATEVAR then
    ahp, aout, abp, abr statevar ain, kcfq, kres
elseif kType == $MVCLPF1 then
    aout mvclpf1 ain, kcfq, kres
elseif kType == $MVCLPF2 then
    aout mvclpf2 ain, kcfq, kres
elseif kType == $MVCLPF3 then
    aout mvclpf3 ain, kcfq, kres
else
    aout = 0
endif
    xout aout
endop

opcode Wave, a, k
kcps    xin

asqr    vco2 1, kcps * 0.495, 10      ; square
asaw    vco2 1, kcps * 1.005, 0       ; wave
xout    0.5 * (asqr + asaw)
endop

opcode Filter, a, aiii
ain, iFilterType, iCoeff, iCps  xin

iDivision = 1 / (iCoeff * giSpb)
kLfo    loopseg iDivision, 0, 0, 0, 0.5, 1, 0.5, 0
iBase   = iCps
iMod    = iBase * 9

kcfq    = iBase + iMod * kLfo
kres    init 0.6

aout    MultiFilter ain,    kcfq, kres, iFilterType
aout    balance aout, ain

```

```

        xout aout
endop

opcode Reverb, aa, aaii
adryL, adryR, ifeedback, imix xin
awetL, awetR reverbsc adryL, adryR, ifeedback, 10000

aoutL = (1 - imix) * adryL + imix * awetL
aoutR = (1 - imix) * adryR + imix * awetR

        xout aoutL, aoutR
endop

instr Bass
    iCoeff      = p4
    iCps        = p5
    iFilterType = p6

    aWave   Wave iCps
    aOut    Filter aWave, iFilterType, iCoeff, iCps
    aOut    linen aOut, .01, p3, .1

    gaOut   = gaOut + aOut
endin

opcode Note, 0, iiiii
    idt = 2 * giSpb
    iNum, iCoeff, iPch, iFilterType xin
    event_i "i", "Bass", idt * iNum, idt, iCoeff, cpspch(iPch), iFilterType
endop

instr Notes
    iFilterType = p4
    EchoFilterName iFilterType

    Note 0, 2, 6.04, iFilterType
    Note 1, 1/3, 7.04, iFilterType
    Note 2, 2, 6.04, iFilterType
    Note 3, 1/1.5, 7.07, iFilterType

    Note 4, 2, 5.09, iFilterType
    Note 5, 1, 6.09, iFilterType
    Note 6, 1/1.5, 5.09, iFilterType
    Note 7, 1/3, 6.11, iFilterType

    Note 8, 1, 6.04, iFilterType
    Note 9, 1/3, 7.04, iFilterType
    Note 10, 2, 6.04, iFilterType
    Note 11, 1/1.5, 7.07, iFilterType

    Note 12, 2, 6.09, iFilterType
    Note 13, 1, 7.09, iFilterType
    Note 14, 1/1.5, 6.11, iFilterType
    Note 15, 1/3, 6.07, iFilterType

    Note 16, 2, 6.04, iFilterType
    Note 17, 1/3, 7.04, iFilterType
    Note 18, 2, 6.04, iFilterType
    Note 19, 1/1.5, 7.07, iFilterType

    turnoff
endin

```

```

opcode TrigNotes, 0, ii
iNum, iFilterType xin
idt = 20
    event_i "i", "Notes", idt * iNum, 0, iFilterType
endop

instr PlayAll
iMixLevel = p4
event_i "i", "Main", 0, (14 * 20), iMixLevel

TrigNotes 0, $MOOG_LADDER
TrigNotes 1, $MOOG_VCF
TrigNotes 2, $LPF18
TrigNotes 3, $BQREZ
TrigNotes 4, $CLFILT
TrigNotes 5, $BUTTERLP
TrigNotes 6, $LOWRES
TrigNotes 7, $REZZY
TrigNotes 8, $SVFILTER
TrigNotes 9, $VLOWRES
TrigNotes 10, $STATEVAR
TrigNotes 11, $MVCLPF1
TrigNotes 12, $MVCLPF2
TrigNotes 13, $MVCLPF3

turnoff
endin

opcode DumpNotes, 0, iisi
iNum, iFilterType, SFile, iMixLevel xin
idt = 20
Sstr sprintf {{i "%s" %f %f "%s" %f}}, "Dump", idt*iNum, idt, SFile, iMixLevel
    scoreline_i Sstr
        event_i "i", "Notes", idt * iNum, 0, iFilterType
endop

instr DumpAll
iMixLevel = p4

DumpNotes 0, $MOOG_LADDER, "moogladder-dubstep.wav", iMixLevel
DumpNotes 1, $MOOG_VCF, "moogvcf-dubstep.wav", iMixLevel
DumpNotes 2, $LPF18 , "lpf18-dubstep.wav", iMixLevel
DumpNotes 3, $BQREZ, "bqrez-dubstep.wav", iMixLevel
DumpNotes 4, $CLFILT, "clfilt-dubstep.wav", iMixLevel
DumpNotes 5, $BUTTERLP, "butterlp-dubstep.wav", iMixLevel
DumpNotes 6, $LOWRES, "lowres-dubstep.wav", iMixLevel
DumpNotes 7, $REZZY, "rezzy-dubstep.wav", iMixLevel
DumpNotes 8, $SVFILTER, "svfilter-dubstep.wav", iMixLevel
DumpNotes 9, $VLOWRES , "vlowres-dubstep.wav", iMixLevel
DumpNotes 10, $STATEVAR, "statevar-dubstep.wav", iMixLevel
DumpNotes 11, $MVCLPF1 , "mvclpf1-dubstep.wav", iMixLevel
DumpNotes 12, $MVCLPF2 , "mvclpf2-dubstep.wav", iMixLevel
DumpNotes 13, $MVCLPF3 , "mvclpf3-dubstep.wav", iMixLevel

turnoff
endin

instr Dump
Sfile      = p4
iMixLevel  = p5

iVolume     = 0.2
iReverbFeedback = 0.85

```

```
aoutL, aoutR Reverb gaOut, gaOut, iReverbFeedback, iMixLevel
fout SFile, 14, (iVolume * aoutL), (iVolume * aoutR)
endin

instr Main
iVolume = 0.2
iReverbFeedback = 0.3
iMixLevel      = p4

aoutL, aoutR Reverb gaOut, gaOut, iReverbFeedback, iMixLevel
outs (iVolume * aoutL), (iVolume * aoutR)

gaOut = 0
endin

</CsInstruments>
<CsScore>
; the fourth parameter is a reverb mix level
i "PlayAll" 0 1 0.35
; uncomment to save output to wav files
;i "DumpAll" 0 1 0.35
</CsScore>
</CsoundSynthesizer>
;example by Anton Kholomiov
;based on the Jacob Joaquin wobble bass sound
```


05 D. 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.

Basic Delay Line Read-Write Unit

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 establishment 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_delay.csd

```
<CsoundSynthesizer>
<CsOptions>
-odac ; activates real time sound output
</CsOptions>
<CsInstruments>
sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

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, 0
kCps    randomh 400, 600, 0.5
aEnv    interp   kEnv
aSig    oscil    aEnv, kCps

; -- create a delay buffer --
aBufOut delayr  0.3
        delayw  aSig

; -- send audio to output (input and output to the buffer are mixed)
aOut    =         aSig + (aBufOut*0.4)
        out      aOut/2, aOut/2
    endin

</CsInstruments>
<CsScore>
i 1 0 25
</CsScore>
</CsoundSynthesizer>
;example by Iain McCurdy
```

Delay with Feedback

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 cycles 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 to zero this value is. The following example implements a simple delay with feedback.

EXAMPLE 05D02_delay_feedback.csd

```
<CsoundSynthesizer>
<CsOptions>
-odac ;activates real time sound output
```

```

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

    instr 1
    ; -- create an input signal: short 'blip' sounds --
    kEnv    loopseg  0.5,0,0,0,0.0005,1,0.1,0,1.9,0,0 ; repeating envelope
    kCps    randomh 400, 600, 0.5                      ; 'held' random values
    aEnv    interp   kEnv                                ; a-rate envelope
    aSig    oscil    aEnv, kCps                          ; generate audio

    ; -- create a delay buffer --
    iFdbck =      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*iFdbck)

    ; send audio to output (mix the input signal with the delayed signal)
    aOut    =      aSig + (aBufOut*0.4)
    out     aOut/2, aOut/2
    endin

</CsInstruments>
<CsScore>
i 1 0 25
</CsScore>
</CsoundSynthesizer>
;example by Iain McCurdy

```

An alternative for implementing a simple delay-feedback line in Csound would be to use the [delay](#) opcode. This is the same example done in this way:

EXAMPLE 05D03_delay_feedback_2.csd

```

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

    instr 1
    kEnv    loopseg  0.5,0,0,0,0.0005,1,0.1,0,1.9,0,0
    kCps    randomh 400, 600, 0.5
    aSig    oscil    a(kEnv), kCps

    iFdbck =      0.7          ; feedback ratio
    aDelay  init     0          ; initialize delayed signal
    aDelay  delay    aSig+(aDelay*iFdbck), .3 ;delay 0.3 seconds

    aOut    =      aSig + (aDelay*0.4)
    out     aOut/2, aOut/2
    endin

</CsInstruments>

```

```
<CsScore>
i 1 0 25
</CsScore>
</CsoundSynthesizer>
;example by Iain McCurdy and joachim heintz
```

Tap Delay Line

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 Csound's 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 it 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 minimum delay time that will be permissible will be the duration of one k cycle (ksmps/sr).

EXAMPLE 05D04_deltapi.csd

```
<CsoundSynthesizer>
<CsOptions>
-odac ; activates real time sound output
</CsOptions>
<CsInstruments>
sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

instr 1
; -- create an input signal: short 'blip' sounds --
kEnv      loopseg 0.5,0,0,0,0.0005,1,0.1,0,1.9,0,0
aEnv      interp   kEnv
aSig      oscil    aEnv, 500
```

```

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)
aOut         linen    aSig + (aTap*0.4), .1, p3, 1
                out     aOut/2, aOut/2
        endin

</CsInstruments>
<CsScore>
i 1 0 30
</CsScore>
</CsoundSynthesizer>
;example by Iain McCurdy

```

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 05D05_multi-tap_delay.csd

```

<CsoundSynthesizer>
<CsOptions>
-odac ; activates real time sound output
</CsOptions>
<CsInstruments>
sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

instr 1
; -- create an input signal: short 'blip' sounds --
kEnv    loopseg  0.5,0,0,0,0.0005,1,0.1,0,1.9,0,0; repeating envelope
kCps    randomh  400, 1000, 0.5                   ; 'held' random values
aEnv    interp    kEnv                            ; a-rate envelope
aSig    oscil    aEnv, kCps                      ; 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)
aOut     linen    aSig + ((aTap1+aTap2+aTap3)*0.4), .1, p3, 1
                out     aOut/2, aOut/2
        endin

</CsInstruments>
<CsScore>
i 1 0 25
</CsScore>

```

```
</CsoundSynthesizer>
;example by Iain McCurdy
```

Flanger

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. Instead a stack of harmonically related resonances are perceived whichs frequencies are in simple ratio with $1/delay_time$. This effect is known as a comb filter and is explained in the previous chapter. 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 an U-shaped parabola as its waveform as this seems to provide the smoothest comb filter modulations.

EXAMPLE 05D06_flanger.cs

```
<CsoundSynthesizer>
<CsOptions>
-odac ; activates real time sound output
</CsOptions>
<CsInstruments>

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

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
kFdbck 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*kFdbck) ; write audio into buffer

; send audio to the output (mix the input signal with the delayed signal)
aOut linen (aSig + aTap)/2, .1, p3, 1
      out aOut, aOut
      endin

</CsInstruments>
<CsScore>
i 1 0 25
</CsScore>
</CsoundSynthesizer>
;example by Iain McCurdy
```

As alternative to using the *deltap* group of opcodes, Csound provides opcodes which start with *vdel* (for *variable delay line*). They establish one single delay line per opcode. This may be easier to write for one or few taps, whereas for a large number of taps the method which has been described in the previous examples is preferable.

Basically all these opcode have three main arguments:

1. The audio input signal.
2. The delay time as audio signal.
3. The maximum possible delay time.

Some caution must be given to the unit in argument 2 and 3: *vdelay* and *vdelay3* use *milliseconds* here, whereas *vdelayx* uses seconds (as nearly every other opcode in Csound).

This is an identical version of the previous *flanger* example which uses *vdelayx* instead of *deltap3*. The *vdelayx* opcode has an additional parameter which allows the user to set the number of samples to be used for interpolation between 4 and 1024. The higher the number, the better the quality, requiring yet more rendering power.

EXAMPLE 05D07_flanger_2.csd

```
<CsoundSynthesizer>
<CsOptions>
-odac ; activates real time sound output
</CsOptions>
<CsInstruments>

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

giLFOShape ftgen 0, 0, 2^12, 19, 0.5, 1, 180, 1

    instr 1
aSig    pinkish 0.1

aMod    poscil 0.005, 0.05, giLFOShape
iOffset = ksmpls/sr
kFdbck linseg 0.8,(p3/2)-0.5,0.95,1,-0.95

aDelay init 0
aDelay vdelayx aSig+aDelay*kFdbck, aMod+iOffset, 0.5, 128

aOut    linen (aSig+aDelay)/2, .1, p3, 1
        out    aOut, aOut
    endin

</CsInstruments>
<CsScore>
i 1 0 25
</CsScore>
</CsoundSynthesizer>
;example by Iain McCurdy and joachim heintz
```

Custom Delay Line

As an advanced insight into sample-by-sample processing in Csound, we end here with an intriguing example by Steven Yi (showed on the Csound mailing list 2019/12/11). It demonstrates how a delay line can be created as Csound array which is written and read as circular buffer. Here are some comments:

- Line 15: The array is created with the size *delay-time times sample-rate*, in our case $0.25 * 44100 = 11025$. So 11025 samples can be stored in this array.
- Line 16-17: The read pointer *kread_ptr* is set to the second element (*index=1*), the write pointer *kwrite_ptr* is set to the first element (*index=0*) at beginning.
- Line 19-20: The audio signal as input for the delay line – it can be anything.
- Line 22-23, 30-31: The *while* loop iterates through each sample of the audio vector: from *kindx=0* to *kindx=31* if *ksmps* is 32.
- Line 24: Each element of the audio vector is copied into the appropriate position of the array. At the beginning, the first element of the audio vector is copied to position 0, the second element to position 1, and so on.
- Line 25: The element in the array to which the read index “*kread_ptr** points is copied to the appropriate element of the delayed audio signal. As **kread_ptr** starts with 1 (not 0), at first it can only copy zeros.
- Line 27-28: Both pointers are incremented by one and then the *modulo* is taken. This ensures that the array is not read or written beyond its boundaries, but used as a circular buffer.

EXAMPLE 05D08_custom_delay_line.csd

```
<CsoundSynthesizer>
<CsOptions>
-odac ; activates real time sound output
</CsOptions>
<CsInstruments>
sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

instr CustomDelayLine

;; 0.25 second delay
idel_size = 0.25 * sr
kdelay_line[] init idel_size
kread_ptr init 1
kwrite_ptr init 0

asig = vco2(0.3, 220 * (1 + int(lfo:k(3, 2, 2))) * expon(1, p3, 4), 10)
asig = zdf_ladder(asig, 2000, 4)

kindx = 0
while (kindx < ksmpls) do
    kdelay_line[kwrite_ptr] = asig[kindx]
    adel[kindx] = kdelay_line[kread_ptr]
```

```
kwrite_ptr = (kwrite_ptr + 1) % idel_size
kread_ptr = (kread_ptr + 1) % idel_size

kindx += 1
od

out(linen:a(asig,0,p3,1),linen:a(adel,0,p3,1))

endin

</CsInstruments>
<CsScore>
i "CustomDelayLine" 0 10
</CsScore>
</CsoundSynthesizer>
;example by Steven Yi
```


05 E. 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 opcodes for reverberation

Csound's earliest reverb opcodes are *reverb* and *nreverb*. By today's standards they sound rather crude and as a consequence modern Csound users tend to prefer the more recent opcodes *freeverb* and *reverbsc*.

General considerations about using reverb in Csound

The typical way to use a reverb is to run it as an effect throughout the entire Csound performance and to send its 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.

Method I: Global variables

The first method uses Csound's *global variables*, so that an audio variable created in one instrument can be read in another instrument. There are several points to highlight here. First the global audio variable that is used to send audio to 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 than 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 *freverb* 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 these 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_freeverb.csd

```

<CsoundSynthesizer>
<CsOptions>
-odac ; activates real time sound output
</CsOptions>
<CsInstruments>
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 0.5,0,0,1,0.003,1,0.0001,0,0.9969,0,0; 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 freverb 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
</CsScore>
</CsoundSynthesizer>
;example by Iain McCurdy
```

Method II: zak opcodes

The second method 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 `zakinits` opcode. This opcode initializes both a and k rate channels; we must initialize at least one of each even if we don't require both.

```
zakinits 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 Nyquist frequency (sr/2) - it controls the cutoff frequencies of low pass filters within the algorithm.

EXAMPLE 05E02_reverbsc.csd

```
<CsoundSynthesizer>
<CsOptions>
-odac ; activates real time sound output
</CsOptions>
<CsInstruments>
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.
zakinits 1, 1

instr 1 ; sound generating instrument - sparse noise bursts
kEnv      loopseg 0.5,0,0,1,0.003,1,0.0001,0,0.9969,0,0; 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
</CsScore>
</CsoundSynthesizer>
;example by Iain McCurdy

```

`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 `freverb`'s rather static reverberant tail. On the other hand `reverbsc`'s tail is not as smooth as that of `freverb`, individual 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.

Method III: chn opcodes

As *third method*, 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_reverb_with_chn.csd

```

<CsoundSynthesizer>
<CsOptions>
-odac ; activates real time sound output
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2

```

```

0dbfs = 1

instr 1 ; sound generating instrument - sparse noise bursts
kEnv      loopseg  0.5,0, 0,1,0.003,1,0.0001,0,0.9969,0,0 ; 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

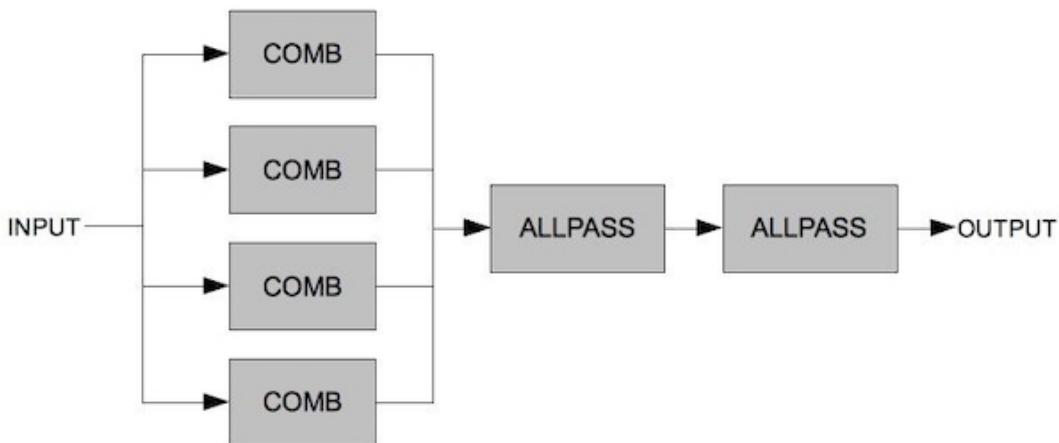
instr 5 ; reverb (always on)
aInSig    chnget    "ReverbSend"   ; read audio from the named channel
kTime     init      4           ; reverb time
kHDif     init      0.5         ; 'high frequency diffusion' (0 - 1)
aRvb      nreverb   aInSig, kTime, kHDif ; create reverb signal
outs      aRvb, aRvb        ; 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
</CsScore>
</CsoundSynthesizer>
;example by Iain McCurdy

```

The Schroeder Reverb Design

Many reverb algorithms including Csound's *freverb*, *reverb* and *nreverb* 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 all-pass 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 *freverb* 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 *freverb* 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 certainly 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_schroeder_reverb.csd

```

<CsoundSynthesizer>
<CsOptions>
-odac -m128
; activate real time sound output and suppress note printing
</CsOptions>
<CsInstruments>
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    expandr   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   expandr   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
aSig   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>
;example by Iain McCurdy
```

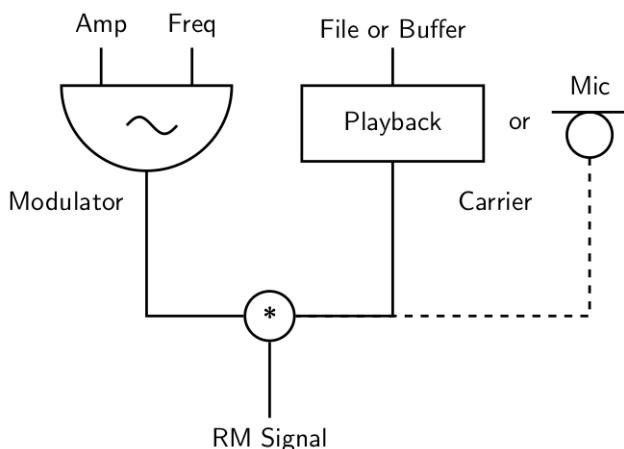
This chapter has introduced some of the more recent Csound opcodes for delay-line based reverb algorithms which in most situations can be used to provide high quality and efficient reverberation. Convolution offers a whole new approach for the creation of realistic reverbs that imitate actual spaces - this technique is demonstrated in the [Convolution](#) chapter.

05 F. AM / RM / WAVESHAPING

An introduction as well as some background theory of amplitude modulation, ring modulation and waveshaping is given in chapters [04 C](#) and [04 E](#). 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 AND RING MODULATION

As shown in chapter [04 C](#), ring modulation in digital domain can be implemented as multiplication of a carrier audio signal with a modulator signal. If adapted to the modification of samples or live input, the carrier signal now changes to a playback unit or a microphone. The modulator usually remains a sine oscillator.



The spectrum of the carrier sound is shifted by plus and minus the modulator frequency. As this is happening for each part of the spectrum, the source sound often seems to lose its center. A piano sound easily becomes bell-like, and a voice can become gonomic.

In the following example, first three static modulating frequencies are applied. As the voice itself has a somehow floating pitch, we already hear an always moving artificial spectrum component. This effect is emphasized in the second instrument which applies a random glissando for the

¹This is the same for Granular Synthesis which can either be “pure” synthesis or applied on sampled sound.

modulating frequency. If the random movements are slow (first with 1 Hz, then 10 Hz), the pitch movements are still recognizable. If they are fast (100 Hz in the last call), the sound becomes noisy.

EXAMPLE 05F01_RM_modification.csd

```
<CsoundSynthesizer>
<CsOptions>
-o dac --env:SSDIR+=../SourceMaterials
</CsOptions>
<CsInstruments>

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

instr RM_static
aMod    oscil      1, p4
aCar    diskin    "fox.wav"
aRM     =          aMod * aCar
        out        aRM, aRM
endin

instr RM_moving
aMod    oscil      1, randomi:k(400,1000,p4,3)
aCar    diskin    "fox.wav"
aRM     =          aMod * aCar
        out        aRM, aRM
endin

</CsInstruments>
<CsScore>
i "RM_static" 0 3 400
i .           + . 800
i .           + . 1600
i "RM_moving" 10 3 1
i .           + . 10
i .           + . 100
</CsScore>
</CsoundSynthesizer>
;written by Alex Hofmann and joachim heintz
```

In instrument *RM_static*, the fourth parameter of the score line (p4) directly yields the frequency of the modulator. In instrument *RM_moving*, this frequency is a random movement between 400 and 1000 Hz, and p4 here yields the rate in which new random values are generated.

For amplitude modulation, a constant part - the *DC offset* - is added to the modulating signal. The result is a mixture of unchanged and ring modulated sound, in different weights. The most simple way to implement this is to add a part of the source signal to the ring modulated signal.

WAVESHAPING

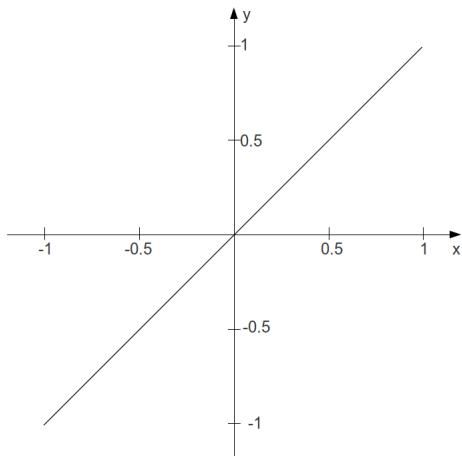
In chapter 04E waveshaping has been described as a method of applying a transfer function to an incoming signal. It has been discussed that the table which stores the transfer function must

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.

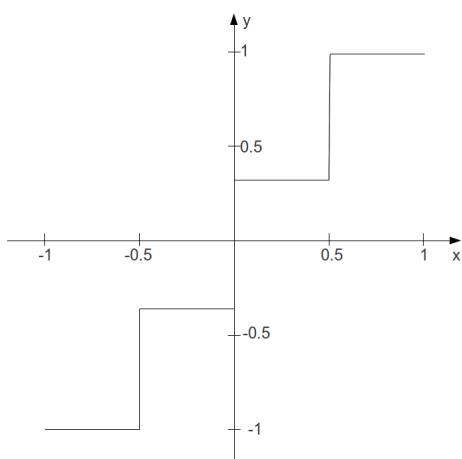
Bit Depth Reduction

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 05F02_Wvshp_bit_crunch.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
           out      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 already to a heavy distortion, because 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 ...

EXAMPLE 05F03_Wvshp_different_transfer_funcs.csd

```

<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
iTtrsFnc = p4
kEnv     linseg 0, .01, 1, p3-.2, 1, .01, 0
aL, aR   soundin "ClassGuit.wav"
aIndxL   = (aL + 1) / 2
aWavShpL tablei aIndxL, iTtrsFnc, 1
aIndxR   = (aR + 1) / 2

```

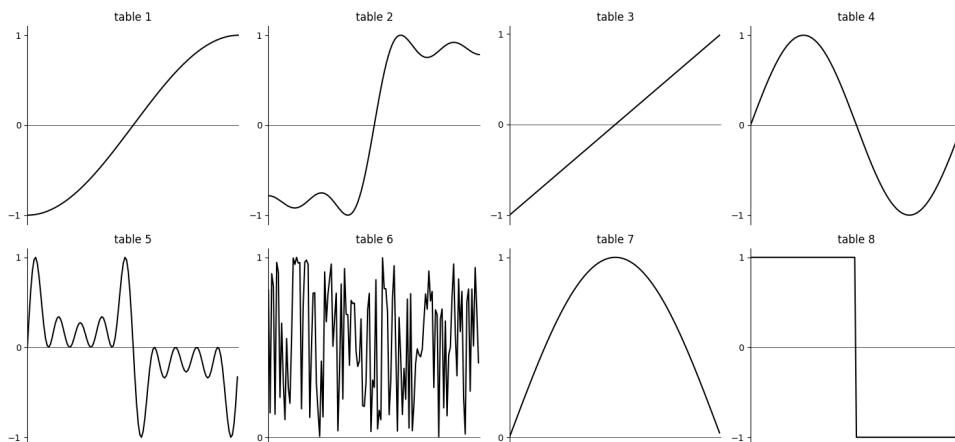
```

aWavShpR  tablei      aIndxR, iTnsFnc, 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, using rather different tables:



EXAMPLE 05F04_distort.csd

```

<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
```

05 G. GRANULAR SYNTHESIS

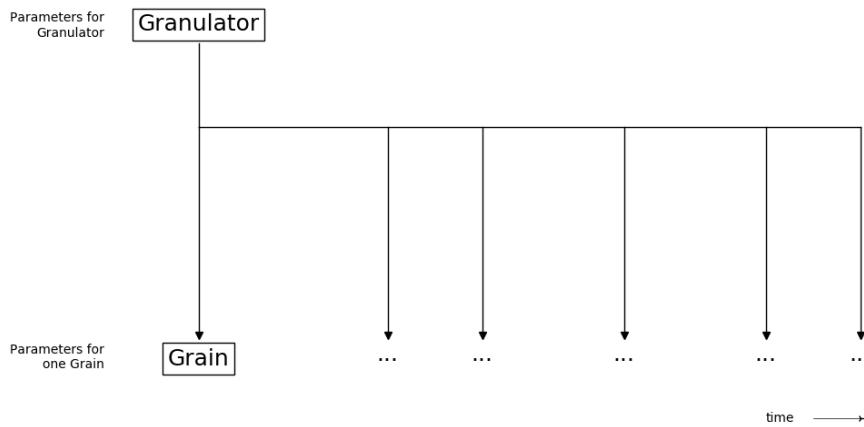
This chapter will focus upon granular synthesis used as a DSP technique upon recorded sound files and will introduce techniques including time stretching, time compressing and pitch shifting. The emphasis will be upon asynchronous granulation. For an introduction to synchronous granular synthesis using simple waveforms please refer to chapter [04 F](#).

We will start with a self-made granulator which we build step by step. It may help to understand the main parameters, and to see upon which decisions the different opcode designs are built. In the second part of this chapter we will introduce some of the many Csound opcodes for granular synthesis, in typical use cases.

A Self-Made Granulator

It is perfectly possible to build one's own granular machine in Csound code, without using one of the many opcodes for granular synthesis. This machine will certainly run slower than a native opcode. But for understanding what is happening, and being able to implement own ideas, this is a very instructive approach.

Granular synthesis can be described as a sequence of small sound snippets. So we can think of two units: One unit is managing the sequence, the other unit is performing one grain. Let us call the first unit *Granulator*, and the second unit *Grain*. The *Granulator* will manage the sequence of grains in calling the *Grain* unit again and again, with different parameters:



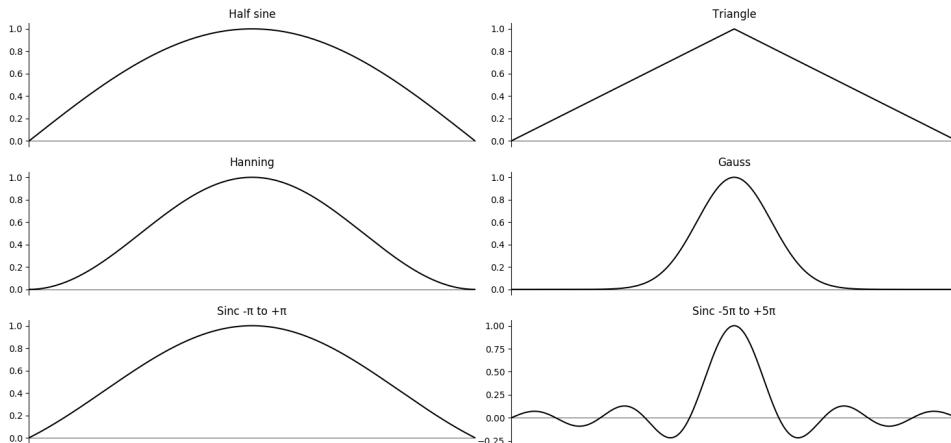
In Csound, we implement this architecture as two instruments. We will start with the instrument which performs one grain.

The Grain Unit

Parameters for One Grain

The *Grain* instrument needs the following information in order to play back a single grain:

1. **Sound.** In the most simple version this is a sound file on the hard disk. More flexible and fast is a sample which has been stored in a buffer (function table). We can also record this buffer in real time and through this perform live granular synthesis.
2. **Point in Sound to start playback.** In the most simple version, this is the same as the *skiptime* for playing back sound from hard disk via *diskin*. Usually we will choose seconds as unit for this parameter.
3. **Duration.** The duration for one grain is usually in the range 20-50 ms, but can be smaller or bigger for special effects. In Csound this parameter is passed to the instrument as *p3* in its call, measured in seconds.
4. **Speed of Playback.** This parameter is used by *diskin* and similar opcodes: 1 means the normal speed, 2 means double speed, 1/2 means half speed. This would result in no pitch change (1), octave higher (2) and octave lower(1/2). Negative numbers mean reverse playback.
5. **Volume.** We will measure it in *dB*, where 0 dB means to play back the sound as it is recorded.
6. **Envelope.** Each grain needs an envelope which starts and ends at zero, to ensure that there will be no clicks. These are some frequently used envelopes:¹



7. **Spatial Position.** Each grain will be send to a certain point in space. For stereo, it will be a panning position between 0 (left) and 1 (right).

¹The function tables have been created with this code:

```
i0 ftgen 1, 0, 8192, 20, 3, 1
i0 ftgen 2, 0, 8192, 9, 1/2, 1, 0
i0 ftgen 3, 0, 8192, 20, 2, 1
i0 ftgen 4, 0, 8192, 20, 6, 1
i0 ftgen 5, 0, 8192, 20, 9, 1
i0 ftgen 6, 0, 8192, 20, 9, 1, 5
```

Simple Grain Implementation

We start with the most simple implementation. We play back the sound with `diskin` and apply a triangular envelope with the `linen` opcode. We pass the *grain duration* as *p3*, the *playback start* as *p4* and the *playback speed* as *p5*. We choose a constant grain duration of 50 ms, but in the first five examples different starting points, then in the other five examples from one starting point different playback speeds.

EXAMPLE 05G01_simple_grain.csd

```

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

instr Grain
//input parameters
Sound = "fox.wav"
iStart = p4 ;position in sec to read the sound
iSpeed = p5
iVolume = -3 ;dB
iPan = .5 ;0=left, 1=right
//perform
aSound = diskin:a(Sound,iSpeed,iStart,1)
aOut = linen:a(aSound,p3/2,p3,p3/2)
aL, aR pan2 aOut*ampdb(iVolume), iPan
out(aL,aR)
endin

</CsInstruments>
<CsScore>
;          start speed
i "Grain" 0 .05 .05   1
i .      1 .    .2     .
i .      2 .    .42    .
i .      3 .    .78    .
i .      4 .    1.2    .
i .      6 .    .2     1
i .      7 .    .       2
i .      8 .    .       0.5
i .      9 .    .       10
i .     10 .    .25   -1
</CsScore>
</CsoundSynthesizer>
;example by joachim heintz

```

It is a tiring job to write a score line for each grain ... – no one will do this. But with but a small change we can read through the whole sound file by calling our *Grain* instrument only once! The technique we use in the next example is to start a new instance of the *Grain* instrument by the running instance, as long as the end of the sound file has not yet been reached. (This technique has been described in paragraph *Self-Triggering and Recursion* of chapter 03 C.)

EXAMPLE 05G02_simple_grain_continuous.cs

```

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

instr Grain
//input parameters
Sound = "fox.wav"
iStart = p4 ;position in sec to read the sound
iSpeed = 1
iVolume = -3 ;dB
iPan = .5 ;0=left, 1=right
//perform
aSound = diskin:a(Sound,iSpeed,iStart,1)
aOut = linen:a(aSound,p3/2,p3,p3/2)
aL, aR pan2 aOut*ampdb(iVolume), iPan
out(aL,aR)
//call next grain until sound file has reached its end
if iStart < filelen(Sound) then
  schedule("Grain",p3,p3,iStart+p3)
endif
endin
schedule("Grain",0,50/1000,0)

</CsInstruments>
<CsScore>
e 5 ;stops performance after 5 seconds
</CsScore>
</CsoundSynthesizer>
;example by joachim heintz

```

Improvements

The *Grain* instrument works but it has some weaknesses:

- Rather than being played back from disk, the sound should be put in a buffer (function table) and played back from there. This is faster and gives more flexibility, for instance in filling the buffer with real-time recording.
- The envelope should also be read from a function table. Again, this is faster and offers more flexibility. In case we want to change the envelope, we simply use another function table, without changing any code of the instrument.

Table reading can be done by different methods in Csound. Have a look at chapter 03 D for details. We will use reading the tables with the [poscil3](#) oscillator here. This should give a very good result in sound quality.

In the next example we reproduce the first example above to check the new code to the *Grain* instrument.

EXAMPLE 05G03_simple_grain_optimized.cs

```

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

//load sample to table and create triangular shape
giSample ftgen 0, 0, 0, -1, "fox.wav", 0, 0, 1
giEnv ftgen 0, 0, 8192, 20, 3, 1
giSampleLen = ftlen(giSample)/sr

instr Grain
//input parameters
iStart = p4 ;position in sec to read the sound
iSpeed = p5
iVolume = -3 ;dB
iPan = .5 ;0=left, 1=right
//perform
aEnv = poscil3:a(ampdb(iVolume),1/p3,giEnv)
aSound = poscil3:a(aEnv,iSpeed/giSampleLen,giSample,iStart/giSampleLen)
aL, aR pan2 aSound, iPan
out(aL,aR)
endin

</CsInstruments>
<CsScore>
;          start speed
i "Grain" 0 .05 .05   1
i .        1 .    .2    .
i .        2 .    .42   .
i .        3 .    .78   .
i .        4 .    1.2   .
i .        6 .    .2    1
i .        7 .    .      2
i .        8 .    .      0.5
i .        9 .    .      10
i .       10 .    .25  -1
</CsScore>
</CsoundSynthesizer>
;example by joachim heintz

```

Some comments to this code:

- Line 12-13: The sample *fox.wav* is loaded into function table *giSample* via GEN routine 01. Note that we are using here -1 instead of 1 because we don't want to normalize the sound.² The triangular shape is loaded via GEN 20 which offers a good selection of different envelope shapes.
- Line 14: *giSampleLen = ftlen(giSample)/sr*. This calculates the length of the sample in seconds, as length of the function table divided by the sample rate. It makes sense to store this in a global variable because we are using it in the *Grain* instrument again and again.
- Line 23:*aEnv = poscil3:a(ampdb(iVolume),1/p3,giEnv)*. The envelope (as audio

²This decision is completely up to the user.

signal) is reading the table *giEnv* in which a triangular shape is stored. We set the amplitude of the oscillator to *ampdb(iVolume)*, so to the amplitude equivalent of the *iVolume* decibel value. The frequency of the oscillator is $1/p3$ because we want to read the envelope exactly once during the performance time of this instrument instance.

- Line 24: *aSound = poscil3:a(aEnv, iSpeed/giSampleLen, \\ giSample, iStart/giSampleLen)*
Again this is a *poscil3* oscillator reading a table. The table is here *giSample*; the amplitude of the oscillator is the *aEnv* signal we produced. The frequency of reading the table in normal speed is $1/giSampleLen$; if we include the speed changes, it is $iSpeed/giSampleLen$. The starting point to read the table is given to the oscillator as phase value (0=start to 1=end of the table). So we must divide *iStart* by *giSampleLen* to get this value.

The Granulator Unit

The main job for the *Granulator* is to call the *Grain* unit again and again over time. The *grain density* and the *grain distribution* direct this process. The other parameters are basically the same as in the *Grain* unit.

As granular synthesis is a mass structure, one of its main features is to deal with variations. Each parameter usually deviates in a given range. We will at first build the *Granulator* in the most simple way, without these deviations, and then proceed to more interesting, variative structures.

Parameters for the Granulator

The first seven parameters are similar to the parameters for the *Grain* unit. Grain density and grain distribution are added at the end of the list.

1. **Sound.** The sound must be loaded in a function table. We pass the variable name or number of this table to the *Grain* instrument.
2. **Pointer in Sound.** Usually we will have a moving pointer position here. We will use a simple line in the next example, moving from start to end of the sound in a certain duration. Later we will implement a moving pointer driven by speed.
3. **Duration.** We will use milliseconds as unit here and then change it to seconds when we call the *Grain* instrument.
4. **Pitch Shift (Transposition).** This is the speed of reading the sound in the *Grain* units, resulting in a pitch shift or transposition. We will use Cent as unit here, and change the value internally to the corresponding speed: cent=0 -> speed=1, cent=1200 -> speed=2, cent=-1200 -> speed=0.5.
5. **Volume.** We will measure it in *dB* as for the *Grain* unit. But the resulting volume will also depend on the grain density, as overlapping grains will add their amplitudes.
6. **Envelope.** The grain envelope must be stored in a function table. We will pass the name or number of the table to the *Grain* instrument.
7. **Spatial Position.** For now, we will use a fixed pan position between 0 (left) and 1 (right), as we did for the *Grain* instrument.
8. **Density.** This is the number of grains per second, so the unit is Hz.
9. **Distribution.** This is a continuum between synchronous granular synthesis, in which all grains are equally distributed, and asynchronous, in which the distribution is irregular or scattered.³

³As maximum irregularity we will consider a random position between the regular position of a grain and the regular position

We will use 0 for synchronous and 1 for asynchronous granular synthesis.

Simple Granulator Implementation

For triggering the single grains, we use the `metro` opcode. We call a grain on each trigger tick of the `metro`. This is a basic example; the code will be condensed later, but is kept here more explicit to show the functionality.

EXAMPLE 05G04_simple_granulator.csd

```
<CsoundSynthesizer>
<CsOptions>
-odac -m128
</CsOptions>
<CsInstruments>
sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

//load sample to table and create half sine envelope
giSample ftgen 0, 0, 0, -1, "fox.wav", 0, 0, 1
giHalfSine ftgen 0, 0, 1024, 9, .5, 1, 0

instr Granulator
//input parameters as described in text
iSndTab = giSample ;table with sample
iSampleLen = ftlen(iSndTab)/sr
kPointer = linseg:k(0,iSampleLen,iSampleLen)
iGrainDur = 30 ;milliseconds
iTranspos = -100 ;cent
iVolume = -6 ;dB
iEnv = giHalfSine ;table with envelope
iPan = .5 ;panning 0-1
iDensity = 50 ;Hz (grains per second)
iDistribution = .5 ;0-1
//perform: call grains over time
kTrig = metro(iDensity)
if kTrig==1 then
  kOffset = random:k(0,iDistribution/iDensity)
  schedulek("Grain", kOffset, iGrainDur/1000, iSndTab, iSampleLen,
            kPointer, cent(iTranspos), iVolume, iEnv, iPan)
endif
endin

instr Grain
//input parameters
iSndTab = p4
iSampleLen = p5
iStart = p6
iSpeed = p7
iVolume = p8 ;dB
iEnvTab = p9
iPan = p10
//perform
aEnv = poscil3:a(ampdb(iVolume),1/p3,iEnvTab)
aSound = poscil3:a(aEnv,iSpeed/iSampleLen,iSndTab,iStart/iSampleLen)
aL, aR pan2 aSound, iPan
```

of the next neighbouring grain. (Half of this irregularity will be a random position between own regular and half of the distance to the neighbouring regular position.)

```

    out(aL,aR)
  endin

  </CsInstruments>
  <CsScore>
  i "Granulator" 0 3
  </CsScore>
  </CsoundSynthesizer>
;example by joachim heintz

```

Some comments:

- Line 13: We use a half sine here, generated by [GEN09](#). Other envelopes are available via [GEN20](#). As we used a high-quality interpolating oscillator in the *Grain* instrument for reading the envelope, the table size is kept rather small.
- Line 18: The length of the sample is calculated here not as a global variable, but once in this instrument. This would allow to pass the *iSndTab* as *p*-field without changing the code.
- Line 30: The irregularity is applied as random offset to the regular position of the grain. The range of the offset is from zero to *iDistribution/iDensity*. For *iDensity*=50Hz and *iDistribution*=1, for instance, the maximum offset would be 1/50 seconds.
- Line 31-32: The [schedulek](#) opcode is used here. It has been introduced in Csound 6.14; for older versions of Csound, [event](#) can be used as well: `event("i","Grain",kOffset, ...)` would be the code then. Note that we divide the *iGrainDur* by 1000, because it was given in milliseconds. For the transformation of the *cent* input to a multiplier, we simply use the [cent](#) opcode.

It is suggested to change some values in this example, and to listen to the result; for instance:

1. Change `kPointer = linseg:k(0,iSampleLen,iSampleLen)` (line 19) to `kPointer = linseg:k(0,iSampleLen*2,iSampleLen)` or to `kPointer = linseg:k(0,iSampleLen*5,iSampleLen)` (increase *p3* in the score then, too). This change will increase the time in which the pointer moves from start to end of the sound file. This is called **time stretching**, and is one of the main features of granular synthesis. If a smaller duration for the pointer is used (e.g. *iSampleLen*/2 or *iSampleLen*/5) we apply *time compression*.
2. Change *iGrainDur* (line 20) from 30 ms to a bigger or smaller value. For very small values (below 10 ms) artifacts arise.
3. Set *iDensity* (line 29) to 10 Hz or less and change the *iDistribution* (line 26). A distribution of 0 should give a perfectly regular sequence of grains, whereas 1 should result in irregularity.

Improvements and Random Deviations

The preferred method for the moving pointer in the *Granulator* instrument is a [phasor](#). It is the best approach for real-time use. It can run for an unlimited time and can easily move backwards. As input for the phasor, technically its frequency, we will put the speed in the usual way: 1 means normal speed, 0 is freeze, -1 is backwards reading in normal speed. As optional parameter we can set a start position of the pointer.

All we have to do for implementing this in Csound is to take the sound file length in account for both, the pointer position and the start position:

```
iFileLen = 2 ;sec
iStart = 1 ;sec
kSpeed = 1
kPhasor = phasor:a(kSpeed/iFileLen,iStart/iFileLen)
kPointer = kPhasor*iFileLen
```

In this example, the phasor will start with an initial phase of $iStart/iFileLen = 0.5$. The $kPhasor$ signal which is always 0-1, will move in the frequency $kSpeed/iFileLen$, here 1/2. The $kPhasor$ will then be multiplied by two, so will become 0-2 for $kPointer$.

It is very useful to add **random deviations** to some of the parameters for granular synthesis. This opens the space for many different structures and possibilities. We will apply here random deviations to these parameters of the *Granulator*:

► *Pointer*. The pointer will “tremble” or “jump” depending on the range of the random deviation.

The range is given in seconds. It is implemented in line 36 of the next example as

```
kPointer = kPhasor*iSampleLen + rnd31:k(iPointerRndDev,0)
```

The opcode `rnd31` is a bipolar random generator which will output values between $-iPointerRndDev$ and $+iPointerRndDev$. This is then added to the normal pointer position.

- *Duration*. We will define here a maximum deviation in percent, related to the medium grain duration. 100% would mean that a grain duration can deviate between half and twice the medium duration. A medium duration of 20 ms would yield a random range of 10-40 ms in this case.
- *Transposition*. We can add to the main transposition a bipolar random range. If, for example, the main transposition is 500 cent and the maximum random transposition is 300 cent, each grain will choose a value between 200 and 800 cent.
- *Volume*. A maximum decibel deviation (also bipolar) can be added to the main volume.
- *Spatial Position*. In addition to the main spatial position (in the stereo field 0-1), we can add a bipolar maximum deviation. If the main position is 0.5 and the maximum deviation is 0.2, each grain will have a panning position between 0.3 and 0.7.

The next example demonstrates the five possibilities one by one, each parameter in three steps: at first with no random deviations, then with slight deviations, then with big ones.

EXAMPLE 05G05_random_deviations.csd

```
<CsoundSynthesizer>
<CsOptions>
-odac -m128
</CsOptions>
<CsInstruments>
sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

giSample ftgen 0, 0, 0, -1, "fox.wav", 0, 0, 1
giHalfSine ftgen 0, 0, 1024, 9, .5, 1, 0

instr Granulator
//standard input parameter
iSndTab = giSample
iSampleLen = ftlen(iSndTab)/sr
```

```

iStart = 0 ;sec
kPointerSpeed = 2/3
iGrainDur = 30 ;milliseconds
iTanspos = -100 ;cent
iVolume = -6 ;dB
iEnv = giHalfSine ;table with envelope
iPan = .5 ;panning 0-1
iDensity = 50 ;Hz (grains per second)
iDistribution = .5 ;0-1
//random deviations (for demonstration set to p-fields)
iPointerRndDev = p4 ;sec
iGrainDurRndDev = p5 ;percent
iTansposRndDev = p6 ;cent
iVolumeRndDev = p7 ;dB
iPanRndDev = p8 ;as in iPan
//perform
kPhasor = phasor:k(kPointerSpeed/iSampleLen,iStart/iSampleLen)
kTrig = metro(iDensity)
if KTrig==1 then
    kPointer = kPhasor*iSampleLen + rnd31:k(iPointerRndDev,0)
    kOffset = random:k(0,iDistribution/iDensity)
    kGrainDurDiff = rnd31:k(iGrainDurRndDev,0) ;percent
    kGrainDur = iGrainDur*2^(kGrainDurDiff/100) ;ms
    kTranspos = cent(iTanspos+rnd31:k(iTansposRndDev,0))
    kVol = iVolume+rnd31:k(iVolumeRndDev,0)
    kPan = iPan+rnd31:k(iPanRndDev,0)
    schedulek("Grain",kOffset,kGrainDur/1000,iSndTab,
              iSampleLen,kPointer,kTranspos,kVol,iEnv,kPan)
endif
endin

instr Grain
//input parameters
iSndTab = p4
iSampleLen = p5
iStart = p6
iSpeed = p7
iVolume = p8 ;dB
iEnvTab = p9
iPan = p10
//perform
aEnv = poscil3:a(ampdb(iVolume),1/p3,iEnvTab)
aSound = poscil3:a(aEnv,iSpeed/iSampleLen,iSndTab,iStart/iSampleLen)
aL, aR pan2 aSound, iPan
out(aL,aR)
endin

</CsInstruments>
<CsScore>
t 0 40
; Random Deviations: Pointer GrainDur Transp Vol Pan
;RANDOM POINTER DEVIATIONS
i "Granulator" 0 2.7 0 0 0 0 0 ;normal pointer
i . 3 . 0.1 0 0 0 0 ;slight trembling
i . 6 . 1 0 0 0 0 ;chaotic jumps
;RANDOM GRAIN DURATION DEVIATIONS
i . 10 . 0 0 0 0 0 ;no deviation
i . 13 . 0 100 0 0 0 ;100%
i . 16 . 0 200 0 0 0 ;200%
;RANDOM TRANSPOSITION DEVIATIONS
i . 20 . 0 0 0 0 0 ;no deviation
i . 23 . 0 0 300 0 0 ;±300 cent maximum
i . 26 . 0 0 1200 0 0 ;±1200 cent maximum

```

```

;RANDOM VOLUME DEVIATIONS
i .      30 .    0     0     0     0     0 ;no deviation
i .      33 .    0     0     0     6     0 ;±6 dB maximum
i .      36 .    0     0     0     12    0 ;±12 dB maximum
;RANDOM PAN DEVIATIONS
i .      40 .    0     0     0     0     0 ;no deviation
i .      43 .    0     0     0     0     .1 ;±0.1 maximum
i .      46 .    0     0     0     0     .5 ;±0.5 maximum
</CsScore>
</CsoundSynthesizer>
;example by joachim heintz

```

It sounds like for normal use, the pointer, transposition and pan deviation are most interesting to apply.

Final Example

After first presenting the more instructional examples, this final one shows some of the potential applications for granular sounds. It uses the same parts of *The quick brown fox* as in the first example of this chapter, each which different sounds and combination of the parameters.

EXAMPLE 05G06_the_fox_universe.csd

```

<CsoundSynthesizer>
<CsOptions>
-odac -m128
</CsOptions>
<CsInstruments>
sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1
seed 0

opcode Chan,S,Si
Sname, id xin
Sout sprintf "%s_%d", Sname, id
xout Sout
endop

giSample ftgen 0, 0, 0, -1, "fox.wav", 0, 0, 1
giSinc ftgen 0, 0, 1024, 20, 9, 1
gi_ID init 1

instr Quick
id = gi_ID
gi_ID += 1
iStart = .2
chnset(1/100, Chan("PointerSpeed",id))
chnset(linseg:k(10,p3,1), Chan("GrainDur",id))
chnset(randomi:k(15,20,1/3,3), Chan("Density",id))
chnset(linseg:k(7000,p3/2,6000), Chan("Transpos",id))
chnset(600,Chan("TransposRndDev",id))
chnset(linseg:k(-10,p3-3,-10,3,-30), Chan("Volume",id))
chnset(randomi:k(.2,.8,1,3), Chan("Pan",id))
chnset(.2,Chan("PanRndDev",id))
schedule("Granulator",0,p3,id,iStart)
schedule("Output",0,p3,id,0)
endin

```

```

instr Brown
id = gi_ID
gi_ID += 1
iStart = .42
chnset(1/100, Chan("PointerSpeed",id))
chnset(50, Chan("GrainDur",id))
chnset(50, Chan("Density",id))
chnset(100,Chan("TransposRndDev",id))
chnset(linseg:k(-50,3,-10,12,-10,3,-50), Chan("Volume",id))
chnset(.5, Chan("Pan",id))
schedule("Granulator",0,p3,id,iStart)
schedule("Output",0,p3+3,id,.3)
endin

instr F
id = gi_ID
gi_ID += 1
iStart = .68
chnset(50, Chan("GrainDur",id))
chnset(40, Chan("Density",id))
chnset(100,Chan("TransposRndDev",id))
chnset(linseg:k(-30,3,-10,p3-6,-10,3,-30)+randomi:k(-10,10,1/3),
      Chan("Volume",id))
chnset(.5, Chan("Pan",id))
chnset(.5, Chan("PanRndDev",id))
schedule("Granulator",0,p3,id,iStart)
schedule("Output",0,p3+3,id,.9)
endin

instr Ox
id = gi_ID
gi_ID += 1
iStart = .72
chnset(1/100,Chan("PointerSpeed",id))
chnset(50, Chan("GrainDur",id))
chnset(40, Chan("Density",id))
chnset(-2000,Chan("Transpos",id))
chnset(linseg:k(-20,3,-10,p3-6,-10,3,-30)+randomi:k(-10,0,1/3),
      Chan("Volume",id))
chnset(randomi:k(.2,.8,1/5,2,.8), Chan("Pan",id))
schedule("Granulator",0,p3,id,iStart)
schedule("Output",0,p3+3,id,.9)
endin

instr Jum
id = gi_ID
gi_ID += 1
iStart = 1.3
chnset(0.01,Chan("PointerRndDev",id))
chnset(50, Chan("GrainDur",id))
chnset(40, Chan("Density",id))
chnset(transeg:k(p4,p3/3,0,p4,p3/2,5,3*p4),Chan("Transpos",id))
chnset(linseg:k(0,1,-10,p3-7,-10,6,-50)+randomi:k(-10,0,1,3),
      Chan("Volume",id))
chnset(p5, Chan("Pan",id))
schedule("Granulator",0,p3,id,iStart)
schedule("Output",0,p3+3,id,.7)
if p4 < 300 then
  schedule("Jum",0,p3,p4+500,p5+.3)
endif
endin

instr Whole

```

```

id = gi_ID
gi_ID += 1
iStart = 0
chnset(1/2, Chan("PointerSpeed", id))
chnset(5, Chan("GrainDur", id))
chnset(20, Chan("Density", id))
chnset(.5, Chan("Pan", id))
chnset(.3, Chan("PanRndDev", id))
schedule("Granulator", 0, p3, id, iStart)
schedule("Output", 0, p3+1, id, 0)
endin

instr Granulator
//get ID for resolving string channels
id = p4
//standard input parameter
iSndTab = giSample
iSampleLen = ftlen(iSndTab)/sr
iStart = p5
kPointerSpeed = chnget:k(Chan("PointerSpeed", id))
kGrainDur = chnget:k(Chan("GrainDur", id))
kTranspos = chnget:k(Chan("Transpos", id))
kVolume = chnget:k(Chan("Volume", id))
iEnv = giSinc
kPan = chnget:k(Chan("Pan", id))
kDensity = chnget:k(Chan("Density", id))
iDistribution = 1
//random deviations
kPointerRndDev = chnget:k(Chan("PointerRndDev", id))
kTransposRndDev = chnget:k(Chan("TransposRndDev", id))
kPanRndDev = chnget:k(Chan("PanRndDev", id))
//perform
kPhasor = phasor:k(kPointerSpeed/iSampleLen, iStart/iSampleLen)
kTrig = metro(kDensity)
if kTrig==1 then
  kPointer = kPhasor*iSampleLen + rnd31:k(kPointerRndDev, 0)
  kOffset = random:k(0, iDistribution/kDensity)
  kTranspos = cent(kTranspos+rnd31:k(kTransposRndDev, 0))
  kPan = kPan+rnd31:k(kPanRndDev, 0)
  schedulek("Grain", kOffset, kGrainDur/1000, iSndTab, iSampleLen,
            kPointer, kTranspos, kVolume, iEnv, kPan, id)
endif
endin

instr Grain
//input parameters
iSndTab = p4
iSampleLen = p5
iStart = p6
iSpeed = p7
iVolume = p8
iEnvTab = p9
iPan = p10
id = p11
//perform
aEnv = poscil3:a(ampdb(iVolume), 1/p3, iEnvTab)
aSound = poscil3:a(aEnv, iSpeed/iSampleLen, iSndTab, iStart/iSampleLen)
aL, aR pan2 aSound, iPan
//write audio to channels for id
chnmix(aL, Chan("L", id))
chnmix(aR, Chan("R", id))
endin

```

```

instr Output
id = p4
iRverbTim = p5
aL_dry = chnget:a(Chan("L",id))
aR_dry = chnget:a(Chan("R",id))
aL_wet, aR_wet reverbsc aL_dry, aR_dry, iRverbTim,sr/2
out(aL_dry+aL_wet,aR_dry+aR_wet)
chnclear(Chan("L",id),Chan("R",id))
endin

</CsInstruments>
<CsScore>
i "Quick" 0 20
i "Brown" 10 20
i "F" 20 50
i "Ox" 30 40
i "Jum" 72 30 -800 .2
i "Quick" 105 10
i "Whole" 118 5.4
</CsScore>
</CsoundSynthesizer>
;example by joachim heintz

```

Some comments:

- Line 12-16: The User-Defined Opcode (UDO) *Chan* puts a string and an ID together to a combined string: *Chan("PointerSpeed", 1)* returns "PointerSpeed_1". This is nothing but a more readable version of *sprintf("%s_%d", "PointerSpeed", 1)*.
- Line 20-24: The whole architecture of this example is based on software channels. The *instr Quick* schedules one instance of *instr Granulator*. While this instance is still running, the *instr Brown* schedules another instance of *instr Granulator*. Both, *Quick* and *Brown* want to send their specific values to their instance of *instr Granulator*. This is done by an ID which is added to the channel name. For the pointer speed, *instr Quick* uses the channel "PointerSpeed*1" whereas *instr Brown* uses the channel "PointerSpeed*2". So each of the instruments *Quick*, *Brown* etc. have to get a unique ID. This is done with the global variable *gi_ID*. When *instr Quick* starts, it sets its own variable *id* to the value of *gi_ID* (which is 1 in this moment), and then sets *gi_ID* to 2. So when *instr Brown* starts, it sets its own *id* as 2 and sets *gi_ID* to 3 for future use by instrument *F*.
- Line 34: Each of the instruments which provide the different parameters, like *instr Quick* here, call an instance of *instr Granulator* and pass the ID to it, as well as the pointer start in the sample: *schedule("Granulator", 0, p3, id, iStart)*. The *id* is passed here as fourth parameter, so *instr Granulator* will read *id = p4* in line 112 to receive the ID, and *iStart = p5* in line 116, to receive the pointer start.
- Line 35: As we want to add some reverb, but with different reverb time for each structure, we start one instance of *instr Output* here. Again it will pass the own ID to the instance of *instr Output*, and also the reverb time. In line 162-163 we see how these values are received: *id = p4* and *iRverbTim = p5*
- Line 157-158: *Insr Grain* does not output the audio signal directly, but sends it via *chnmix* to the instance of *instr Output* with the same ID. See line 164-165 for the complementary code in *instr Output*. Note that we must use *chnmix* not *chnset* here because we must add all audio in the overlapping grains (try to substitute *chnmix* by *chnset* to hear the difference). The zeroing of each audio channel at the end of the chain by *chnclear* is also important (comment out line 168 to hear the difference).

Live Input

Instead of using prerecorded samples, granular synthesis can also be applied to live input. Basically what we have to do is to add an instrument which writes the live input continuously to a table. When we ensure that writing and reading the table is done in a circular way, the table can be very short.

The time interval between writing and reading can be very short. If we do not transpose, or only downwards, we can read immediately. Only if we transpose upwards, we must wait. Imagine a grain duration of 50 ms, a delay between writing and reading of 20 ms, and a pitch shift of one octave upwards. The reading pointer will move twice as fast as the writing pointer, so after 40 ms of the grain, it will get ahead of the writing pointer.

So, in the following example, we will set the desired delay time to a small value. It has to be adjusted by the user depending on maximal transposition and grain size.

EXAMPLE 05G07 *live_granular.csd*

```
<CsoundSynthesizer>
<CsOptions>
-odac -iadc -m128
</CsOptions>
<CsInstruments>
sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

giTable ftgen 0, 0, sr, 2, 0 ;for one second of recording
giHalfSine ftgen 0, 0, 1024, 9, .5, 1, 0
giDelay = 1 ;ms

instr Record
aIn = inch(1)
gaWritePointer = phasor(1)
tablew(aIn,gaWritePointer,giTable,1)
endin
schedule("Record",0,-1)

instr Granulator
kGrainDur = 30 ;milliseconds
kTranspos = -300 ;cent
kDensity = 50 ;Hz (grains per second)
kDistribution = .5 ;0-1
kTrig = metro(kDensity)
if kTrig==1 then
  kPointer = k(gaWritePointer)-giDelay/1000
  kOffset = random:k(0,kDistribution/kDensity)
  schedulek("Grain",kOffset,kGrainDur/1000,kPointer,cent(kTranspos))
endif
endin
schedule("Granulator",giDelay/1000,-1)

instr Grain
iStart = p4
iSpeed = p5
aOut = poscil3:a(poscil3:a(.3,1/p3,giHalfSine),iSpeed,giTable,iStart)
out(aOut,aOut)
endin
```

```
</CsInstruments>
<CsScore>
</CsScore>
</CsoundSynthesizer>
;example by joachim heintz
```

We only use some of the many parameters here; others can be added easily. As we chose one second for the table, we can simplify some calculations. Most important is to know for instr *Granulator* the current position of the write pointer, and to start playback *giDelay* milliseconds (here 1 ms) after it. For this, we write the current write pointer position to a global variable *gaWritePointer* in instr *Record* and get the start for one grain by

```
kPointer = k(gaWritePointer)-giDelay/1000
```

After having built this self-made granulator step by step, we will look now into some Csound opcodes for sample-based granular synthesis.

Csound Opcodes for Granular Synthesis

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 [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.

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 05G08_sndwarp.csd

```

<CsoundSynthesizer>
<CsOptions>
-odac -m128
--env:SSDIR+=../SourceMaterials
</CsOptions>
<CsInstruments>
sr = 44100
ksmps = 16
nchnls = 2
0dbfs = 1

; waveform used for granulation
giSound ftgen 1, 0, 0, 1, "ClassGuit.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       =
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
                     aSigL,aSigR
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>
;example written by Iain McCurdy
```

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 and 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 whereas 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 05G09_selfmade_grain.csd

```
<CsoundSynthesizer>
<CsOptions>
-odac --env:SSDIR+=../SourceMaterials
</CsOptions>
<CsInstruments>

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 = "ClassGuit.wav"

; waveform used for granulation
giSound ftgen 1,0,0,1,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      1/12.5 ;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
</CsScore>
</CsoundSynthesizer>
;example written by Iain McCurdy

```

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 completely 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 of 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 it 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 its 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 is printed to the terminal as each note begins.

EXAMPLE 05G10_granule.csd

```
<CsoundSynthesizer>
<CsOptions>
-odac -m128
--env:SSDIR+=../SourceMaterials
</CsOptions>
<CsInstruments>
sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1
```

```

;waveforms used for granulation
giSoundL ftgen 1,0,1048576,1,"ClassGuit.wav",0,0,1
giSoundR ftgen 2,0,1048576,1,"ClassGuit.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
igsizsize_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,kgsizsize,igsizsize_os,iatt,idec,iseedL,\
    ipitch1,ipitch2,ipitch3,ipitch4
aSigR  granule kamp,ivoice,iratio,imode,ithd,giSoundR,ipshift,igskip,\
    igskip_os,ilength,kgap,igap_os,kgsizsize,igsizsize_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
</CsScore>
</CsoundSynthesizer>
;example written by Iain McCurdy

```

Grain delay effect with fof2

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.

EXAMPLE 05G11_grain_delay.csd

```

<CsoundSynthesizer>
<CsOptions>
--env:SSDIR+=./SourceMaterials
-odac -m128
</CsOptions>
<CsInstruments>
sr = 44100
ksmps = 128
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 0,0,0,1,"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/giTabelen) ; 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/giTabelen)-(kDelTim/(giTabelen/sr)) ;grain phase
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
</CsScore>
</CsoundSynthesizer>
;example by Oeyvind Brandtsegg

```

In the last example we will use the `grain` opcode. This opcode is part of a little group of opcodes which also includes `grain2` and `grain3`. `grain` is the oldest opcode, `Grain2` is a more easy-to-use opcode, while `Grain3` offers more control.

EXAMPLE 05G12_grain.csd

```

<CsoundSynthesizer>
<CsOptions>
-o dac --env:SSDIR=../SourceMaterials
</CsOptions>
<CsInstruments>
sr      = 44100
ksmps  = 128
nchnls = 2
0dbfs  = 1

; First we hear each grain, but later on it sounds more like a drum roll.
gareverbL init      0
gareverbR init      0

```

```

giFt1 ftgen 0, 0, 1025, 20, 2, 1 ; GEN20, Hanning window for grain envelope
giFt2      ftgen      0, 0, 0, 1, "fox.wav", 0, 0, 0

instr 1 ; Granular synthesis of soundfile
ipitch    =           sr/ftlen(giFt2) ; Original frequency of the input sound
kdens1   expon     3, p3, 500
kdens2   expon     4, p3, 400
kdens3   expon     5, p3, 300
kamp     line      1, p3, 0.05
a1       grain     1, ipitch, kdens1, 0, 0, 1, giFt2, giFt1, 1
a2       grain     1, ipitch, kdens2, 0, 0, 1, giFt2, giFt1, 1
a3       grain     1, ipitch, kdens3, 0, 0, 1, giFt2, giFt1, 1
aleft   =
aright   =
outs     aleft, aright ; Output granulation
gareverbL =
gareverbR =
endin

instr 2 ; Reverb
kkamp   line     0, p3, 0.08
aL      reverb   gareverbL, 10*kkamp ; reverberate what is in gareverbL
aR      reverb   gareverbR, 10*kkamp ; and garaverbR
outs    kkamp*aL, kkamp*aR ; and output the result
gareverbL =
gareverbR =
endin
</CsInstruments>
<CsScore>
i1 0 20 ; Granulation
i2 0 21 ; Reverb
</CsScore>
</CsoundSynthesizer>
;example by Bjørn Houdorf

```

Several 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*.

05 H. 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 44100×44100 (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 all-pass 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 soft-

ware 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](#), [ftmorph](#) and [pconvolve](#). [convolve](#) and [dconv](#) are earlier implementations and are less suited to real-time operation, [cross2](#) relates to FFT-based cross synthesis and [ftmorph](#) 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 *ipartitionsiz*e 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 response 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 unit. 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, so about 12ms at a sample rate of 44100 Hz.

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 terracotta bowl sharply. You can, of course, replace them with ones of your own but remain mindful of mono/stereo/multichannel integrity.

EXAMPLE 05H01_pconvolve.csd

```
<CsoundSynthesizer>
<CsOptions>
--env:SSDIR+=../SourceMaterials -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 partition size.
; Larger values require less CPU but result in more latency.
; Smaller values produce lower latency but may cause
; realtime performance issues
ipartitionsize = 256
aconv          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
aMix          ntrpol   adel, aconv*0.1, p5
              outs     aMix ,aMix
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>
;example by Iain McCurdy

```

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_ftconv.csd

```
<CsoundSynthesizer>
<CsOptions>
--env:SSDIR+=../SourceMaterials -odac
</CsOptions>
<CsInstruments>

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

; impulse responses stored as mono 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       =
iTableBuffer  ftgentmp
icount        =
loop:
ival          table
              tableiw
              loop_lt
icount        =
loop2:
ival          table
              tableiw
              loop_lt
endop

instr 3 ; reverse the contents of a function table
        tab_reverse p4
endin

instr 1 ; sound file player
```

```

gasig          diskin    p4,1,0,1
endin

instr 2 ; convolution reverb
; buffer length
iplen =      1024
; derive the length of the impulse response
iirlen =     nsamp(p4)
aconv 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
aMix   ntrpol    adel,aconv*0.1,p5
        outs      aMix, aMix
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
</CsScore>
</CsoundSynthesizer>
;example by Iain McCurdy

```

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 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.

liveconv

The `liveconv` opcode is an interesting extension of the `ftconv` opcode. Its main purpose is to make dynamical reloading of the table with the impulse response not only possible, but give an option to avoid artefacts in this reloading. This is possible as reloading can be done partition by partition.

The following example mimics the live input by short snippets of the `fox.wav` sound file. Once the new sound starts to fill the table (each time instr `Record_IR` is called), it sends the number 1 via software channel `conv_update` to the `kupdate` parameter of the `liveconv` opcode in instr `Convolver`. This will start the process of applying the new impulse response.

EXAMPLE 05H03_liveconv.csd

```
<CsoundSynthesizer>
<CsOptions>
-odac -m128
</CsOptions>
<CsInstruments>

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

;create IR table
giIR_record ftgen 0, 0, 131072, 2, 0

instr Input

ain diskin "beats.wav", 1, 0, 1
chnset ain, "input"
if timeinsts() < 2 then
  outch 2, ain/2
endif
endin

instr Record_IR

;set p3 to table duration
p3 = ftlen(giIR_record)/sr
iskip = p4
irlen = p5

;mimic live input for impulse response
asnd diskin "fox.wav", 1, iskip
amp linseg 0, 0.01, 1, irlen, 1, 0.01, 0
asnd *= amp

;fill IR table
andx_IR line 0, p3, ftlen(giIR_record)
tablew asnd, andx_IR, giIR_record

;send 1 at first k-cycle, otherwise 0
ktrig init 1
chnset ktrig, "conv_update"
ktrig = 0
```

```

;output the IR for reference
outch 1, asnd

endin

instr Convolver

;receive information about updating the table
kupdate chnget "conv_update"

;different dB values for the different IR
kB[ ] fillarray -34, -35, -40, -28, -40, -40, -40
kindx init -1
if kupdate==1 then
  kindx += 1
endif

;apply live convolution
ain chnget "input"
aconv liveconv ain, giIR_record, 2048, kupdate, 0
  outch 2, aconv*ampdb(kdB[kindx])

endin

</CsInstruments>
<CsScore>
;play input sound alone first
i "Input" 0 15.65

;record impulse response multiple times
;           skip   IR_dur
i "Record_IR" 2 1 0.17 0.093
i .          4 . 0.50 0.13
i .          6 . 0.76 0.19
i .          8 . 0.97 0.12
i .         10 . 1.72 0.12
i .         12 . 2.06 0.12
i .         14 . 2.37 0.27

;convolve continuously
i "Convolver" 2 13.65
</CsScore>
</CsoundSynthesizer>
;example by Oeyvind Brandtsegg and Sigurd Saupe

```

Some comments to the code of this example:

- Line 13: A function table is created in which the impulse responses can be recorded in real-time. A power of two size (here $2^{17} = 131072$) is preferred as the partition size will then be an integer multiple of the table size.
- Line 15-24: This instrument mimics the audio source on which the convolution will be applied. Here it is *beats.wav*, a short sound file which is looped.
- Line 27: Whenever instr *Record_IR* is called, it will record an impulse response to table *giIR_record*. The impulse response can be very small, but the whole table must be recorded anyway. So the duration of the instrument (*p3*) must be set to the time it takes for this recording. This is the length of the table divided by the sample rate: *ftlen(giIR_record)/sr*, here $131072 / 44100 = 2.972$ seconds.
- Line 32-24: The second live input which is used for the impulse response, is mimicked here by the file *fox.wav* which is played back with different skip times in the different calls of the

instrument. The envelope *amp* applies a short fade in and fade out to the short portion of the sample which we want to use. (*asnd *= amp* is a short form for *asnd = asnd*amp.*)

- ▶ Line 56-60: Depending on the intensity and the spectral content of the impulse response, the convolution will have rather different volume. The code in these lines is to balance it. The *kdB[]* array has seven different dB values for the seven calls of *instr Record_IR*. Each new update message (when *kupdate* gets 1) will increase the *kindx* pointer in the array so that these seven dB values are being applied in line 54 as *ampdb(kdB[kindx])* to the convolution *aconv*.

05 I. FOURIER ANALYSIS / SPECTRAL PROCESSING

An audio signal can be described as continuous changes of amplitudes in time.¹ This is what we call *time-domain*. With a Fourier Transform (FT), we can transfer this time-domain signal to the *frequency domain*. This can, for instance, be used to analyze and visualize the spectrum of the signal. Fourier transform and subsequent manipulations in the frequency domain open a wide area of far-reaching sound transformations, like time stretching, pitch shifting, cross synthesis and any kind of spectral modification.

General Aspects

Fourier Transform is a complex method. We will describe here in short what is most important to know about it for the user.

FT, STFT, DFT and FFT

As described in chapter 04 A, the mathematician J.B. Fourier (1768-1830) developed a method to approximate periodic functions by weighted sums of the trigonometric functions *sine* and *cosine*. As many sounds, for instance a violin or a flute tone, can be described as *periodic functions*,² we should be able to analyse their spectral components by means of the Fourier Transform.

As continuous changes are inherent to sounds, the FT used in musical applications follows a principle which is well known from film or video. The continuous flow of time is divided into a number of fixed *frames*. If this number is big enough (at least 20 frames per second), the continuous flow can reasonably be divided to this sequence of FT *snapshots*. This is called the *Short Time Fourier Transform (STFT)*.

Some care has to be taken to minimise the side effects of cutting the time into snippets. Firstly an *envelope* for the analysis frame is applied. As one analysis frame is often called *window*, the

¹Silence in the digital domain is not only when the amplitudes are always zero. Silence is any constant amplitude, it be 0 or 1 or -0.2.

²To put this simply: If we zoom into recordings of any pitched sound, we will see periodic repetitions. If a flute is playing a 440 Hz (A4) tone, we will see every 2.27 milliseconds (1/440 second) the same shape.

envelope shapes are called *window function*, *window shape* or *window type*. Most common are the *Hamming* and the *von Hann* (or *Hanning*) window functions:

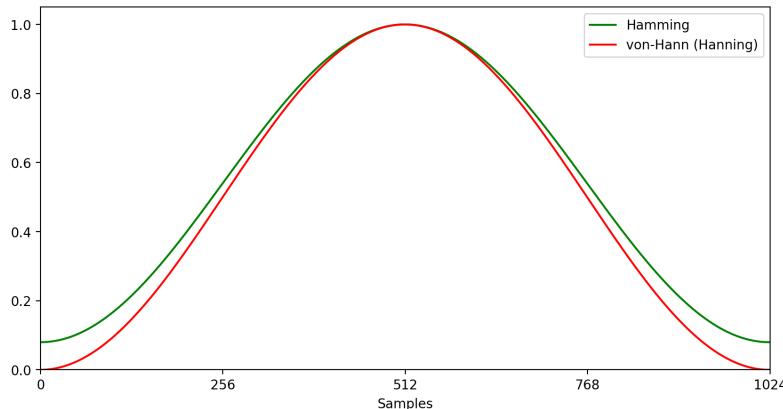


Figure 43.1: Hamming and Hanning window (1024 samples)

Secondly the analysis windows are not put side by side but as *overlapping* each other. The minimal overlap would be to start the next window at the middle of the previous one. More common is to have four overlaps which would result in this image:³

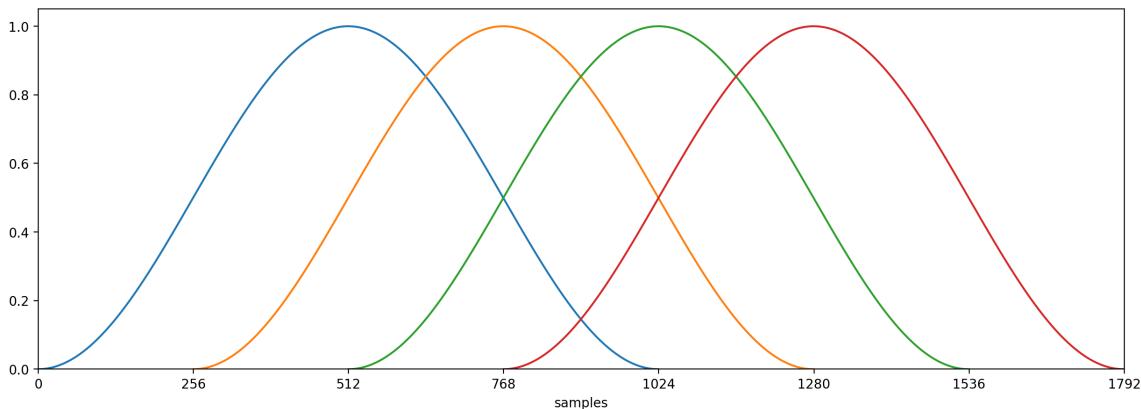


Figure 43.2: Four overlapping Hanning windows (each of size=1024 samples)

We already measured the size of the analysis window in these figures in samples rather than in milliseconds. As we are dealing with *digital* audio, the Fourier Transform has become a *Digital Fourier Transform (DFT)*. It offers some simplifications compared to the analogue FT as the number of amplitudes in one frame is finite. And moreover, there is a considerable gain of speed in the calculation if the window size is a power of two. This version of the DFT is called *Fast Fourier Transform (FFT)* and is implemented in all audio programming languages.

Window Size, Bins and Time-Frequency-Tradeoff

Given that one FFT analysis window size should last about 10-50 ms and that a power-of-two number of samples must be matched, for $sr=44100$ the sizes 512, 1024 or 2048 samples would

³It can be a good choice to have 8 overlaps if CPU speed allows it.

be most suitable for one FFT window, thus resulting in a window length of about 11, 23 and 46 milliseconds respectively. Whether a smaller or larger window size is better, depends on different decisions.

First thing to know about this is that the frequency resolution in a FFT analysis window directly relates to its size. This is based on two aspects: the fundamental frequency and the number of potential harmonics which are analysed and weighted via the Fourier Transform.

The *fundamental frequency* of one given FFT window is the inverse of its size in seconds related to the sample rate. For $sr=44100$ Hz, the fundamental frequencies are:

- 86.13 Hz for a window size of 512 samples
- 43.07 Hz for a window size of 1024 samples
- 21.53 Hz for a window size of 2048 sample.

It is obvious that a larger window is better for frequency analysis at least for low frequencies. This is even more the case as the estimated harmonics which are scanned by the Fourier Transform are *integer multiples* of the fundamental frequency.⁴ These estimated harmonics or partials are usually called *bins* in FT terminology. So, again for $sr=44100$ Hz, the bins are:

- bin 1 = 86.13 Hz, bin 2 = 172.26 Hz, bin 3 = 258.40 Hz for size=512
- bin 1 = 43.07 Hz, bin 2 = 86.13 Hz, bin 3 = 129.20 Hz for size=1024
- bin 1 = 21.53 Hz, bin 2 = 43.07 Hz, bin 3 = 64.60 Hz for size=2048

This means that a larger window is not only better to analyse low frequencies, it also has a better frequency resolution in general. In fact, the window of size 2048 samples has 1024 analysis bins from the fundamental frequency 21.53 Hz to the Nyquist frequency 22050 Hz, each of them covering a frequency range of 21.53 Hz, whilst the window of size 512 samples has 256 analysis bins from the fundamental frequency 86.13 Hz to the Nyquist frequency 22050 Hz, each of them covering a frequency range of 86.13 Hz.⁵

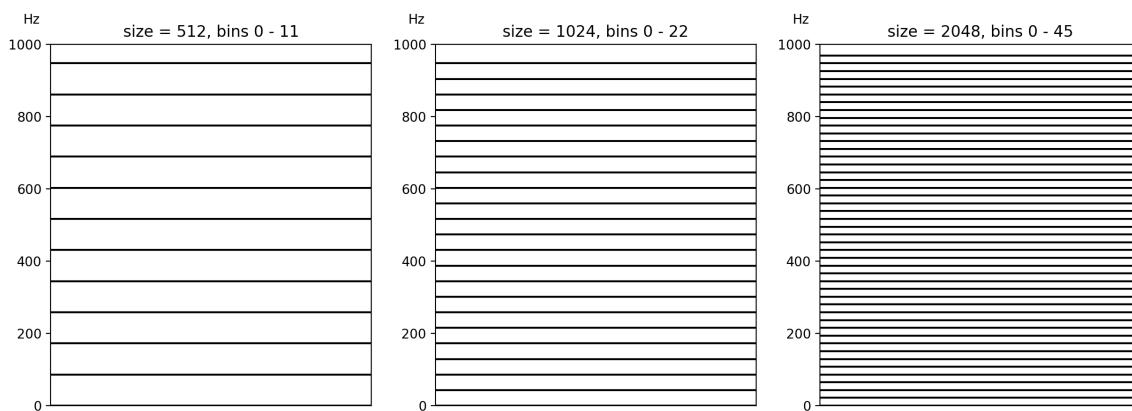


Figure 43.3: Bins up to 1000 Hz for different window sizes

Why then not always use the larger window? — Because a larger window needs more time, or in other words: the time resolution is worse for a window size of 2048, fair for a window size of 1024 and is better for a window size of 512.

⁴Remember that FT is based on the assumption that the signal to be analysed is a periodic function.

⁵For both, the *bin 0* is to be added which analyses the energy at 0 Hz. So in general the number of bins is half of the window size plus one: 257 bins for size 512, 513 bins for size 1024, 1025 bins for size 2048.

This dilemma is known as *time-frequency tradeoff*. We must decide for each FFT situation whether the frequency resolution or the time resolution is more important. If, for instance, we have long piano chords with low frequencies, we may use the bigger window size. If we analyse spoken words of a female voice, we may use the smaller window size. Or to put it very pragmatic: We will use the medium FFT size (1024 samples) first, and in case we experience unsatisfying results (bad frequency response or smearing time resolution) we will change the window size.

FFT in Csound

The raw output of a Fourier Transform is a number of *amplitude-phase* pairs per analysis window frame. Most Csound opcodes use another format which transforms the *phase* values to *frequencies*. This format is related to the *phase vocoder* implementation, so the Csound opcodes of this class are called *phase vocoder opcodes* and start with **pv** or **pvs**.

The **pv** opcodes belong to the early implementation of FFT in Csound. This group comprises the opcodes **pvadd**, **pbufread**, **pvcross**, **pvinterp**, **pvoc**, **pvread** and **vpvoc**. Note that these **pv** opcodes are **not designed to work in real-time**.

The opcodes which **are** designed for *real-time spectral processing* are called *phase vocoder streaming* opcodes. They all start with **pvs**; a rather complete list can be found on the [Spectral Processing](#) site in the Csound Manual. They are fast and easy to use. Because of their power and diversity they are one of the biggest strengths in using Csound.

We will focus on these **pvs** opcodes here, which for most use cases offer all what is desirable to work in the spectral domain. There is, however, a group of opcodes which allow to go back to the *raw FFT output* (without the phase vocoder format). They are listed as [array-based spectral opcodes](#) in the Csound Manual.

From Time Domain to Frequency Domain: **pvsanal**

For dealing with signals in the frequency domain, the **pvs** opcodes implement a new signal type, the *frequency- or f-signal*. If we start with an audio signal in time-domain as **aSig**, it will become **fSig** as result of the Fourier Transform.

There are several opcodes to perform this transform. The most simple one is **pvsanal**. It performs on-the-fly transformation of an input audio signal **aSig** to a frequency signal **fSig**. In addition to the audio signal input it requires some basic FFT settings:

- **ifftsize** is the size of the FFT. As explained above, 512, 1024 or 2048 samples are reasonable values here.
- **overlap** is the number of samples after which the next (overlapping) FFT frame starts (often referred to as *hop size*). Usually it is 1/4 of the FFT size, so for instance 256 samples for a FFT size of 1024.
- **iwinsize** is the size of the analysis window. Usually this is set to the same size as **ifftsize**.⁶

⁶It can be an integral multiple of **ifftsize**, so a window twice as large as the FFT size would be possible and may improve the quality of the analysis. But it also induces more latency which usually is not desirable.

- *iwintype* is the shape of the analysis window. 0 will use a Hamming window, 1 will use a von-Hann (or Hanning) window.

The first example covers two typical situations:

- The audio signal derives from playing back a soundfile from the hard disk (instr 1).
- The audio signal is the live input (instr 2).

(Caution - this example can quickly start feeding back. Best results are with headphones.)

EXAMPLE 05I01_pvsanal.csd

```
<CsoundSynthesizer>
<CsOptions>
-i adc -o dac
--env:SSDIR+=/home/runner/work/csound-floss/csound-floss/resources/SourceMaterials
</CsOptions>
<CsInstruments>
sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

;general values for fourier transform
gifftsz = 1024
gioverlap = 256
giwintyp = 1 ;von hann window

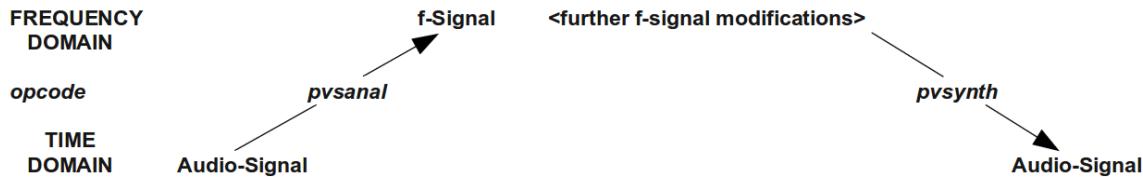
instr 1 ;soundfile to fsig
asig    soundin  "fox.wav"
fsig    pvsanal  asig, gifftsz, gioverlap, gifftsz*2, giwintyp
aback   pvsynth  fsig
        outs     aback, aback
endin

instr 2 ;live input to fsig
        prints  "LIVE INPUT NOW!\n"
ain     inch     1 ;live input from channel 1
fsig    pvsanal  ain, gifftsz, gioverlap, gifftsz, giwintyp
alisten pvsynth  fsig
        outs     alisten, alisten
endin

</CsInstruments>
<CsScore>
i 1 0 3
i 2 3 10
</CsScore>
</CsoundSynthesizer>
;example by joachim heintz
```

You should hear first the *fox.wav* sample, and then the slightly delayed live input signal. The delay (or latency) that you will observe will depend first of all on the general settings for realtime input (*ksmps*, *-b* and *-B*: see chapter 02 D), but it will also be added to by the FFT process. The window size here is 1024 samples, so the additional delay is $1024/44100 = 0.023$ seconds. If you change the window size *gifftsz* to 2048 or to 512 samples, you should notice a larger or shorter delay. For realtime applications, the decision about the FFT size is not only a question of better time resolution versus better frequency resolution, but it will also be a question concerning tolerable latency.

What happens in the example above? Firstly, the audio signal (*asig* or *ain*) is being analyzed and transformed to an f-signal. This is done via the opcode *pvsanal*. Then nothing more happens than the f-signal being transformed from the frequency domain signal back into the time domain (an audio signal). This is called inverse Fourier transformation (IFT or IFFT) and is carried out by the opcode *pvsynth*. In this case, it is just a test: to see if everything works, to hear the results of different window sizes and to check the latency, but potentially you can insert any other pvs opcode(s) in between this analysis and resynthesis:



Alternatives and Time Stretching: *pvstanal* / *pvsbufread*

Working with *pvsanal* to create an f-signal is easy and straightforward. But if we are using an already existing sound file, we are missing one of the interesting possibilities in working with FFT: time stretching. This we can obtain most simple when we use *pvstanal* instead. The *t* in *pvstanal* stands for *table*. This opcode performs FFT on a sound which has been loaded in a table.⁷ These are the main parameters:

- *ktimescal* is the time scaling ratio. 1 means normal speed, 0.5 means half speed, 2 means double speed.
- *kpitch* is the pitch scaling ratio. We will keep this here at 1 which means that the pitch is not altered.
- *ktab* is the function table which is being read.

pvstanal offers some more and quite interesting parameters but we will use it here only a simple way to demonstrate time stretching.

EXAMPLE 05I02_pvstanal.csd

```

<CsoundSynthesizer>
<CsOptions>
-o dac
</CsOptions>
<CsInstruments>

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

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

instr 1
iTimeScal =      p4
fsig      pvstanal  iTimeScal, 1, 1, gfil
aout      pvsynth   fsig
          outs      aout, aout
endin
  
```

⁷The table can also be recorded in live performance.

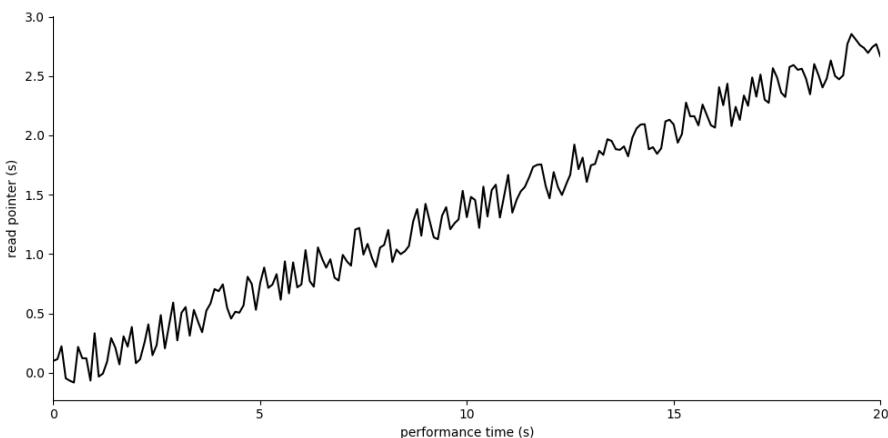
```
</CsInstruments>
<CsScore>
i 1 0 2.7 1 ;normal speed
i 1 3 1.3 2 ;double speed
i 1 6 4.5 0.5 ; half speed
i 1 12 17 0.1 ; 1/10 speed
</CsScore>
</CsoundSynthesizer>
;example by joachim heintz
```

We hear that for extreme time stretching artifacts arise. This is expected and a result of the FFT resynthesis. Later in this chapter we will discuss how to avoid these artifacts.

The other possibility to work with a table (buffer) and get the f-signal by reading it is to use [pvsbuf-read](#). This opcode does not read from an audio buffer but needs a buffer which is filled with FFT data already. This job is done by the related opcode [pvsbuffer](#). In the next example, we wrap this procedure in the User Defined Opcode *FileToPvsBuf*. This *UDO* is called at the first control cycle of instrument *simple_time_stretch*, when *timeinstk()* (which counts the control cycles in an instrument) outputs 1. After this job is done, the pvs-buffer is ready and stored in the global variable *gibuffer*.

Time stretching is then done in the first instrument in a similar way we performed above with *pvtanal*; only that we do not control directly the speed of reading but the real time position (in seconds) in the buffer. In the example, we start in the middle of the sound file and read the words "over the lazy dog" with a time stretch factor of about 10.

The second instrument can still use the buffer. Here a time stretch line is superimposed by a *trembling* random movement. It changes 10 times a second and interpolates to a point which is between - 0.2 seconds and + 0.2 seconds from the current position of the slow moving time pointer created by the expression *linseg:k(0,p3,gilen)*.



So although a bit harder to use, *pvsbufread* offers some nice possibilities. And it is reported to have a very good performance, for instance when playing back a lot of files triggered by a MIDI keyboard.

EXAMPLE 05I03_pvsbufread.csd

```
<CsoundSynthesizer>
<CsOptions>
-o dac --env:SSDIR+=../SourceMaterials
```

```

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

opcode FileToPvsBuf, iik, kSooop
;writes an audio file to a fft-buffer if trigger is 1
kTrig, Sfile, iFFTsize, iOverlap, iWinSize, iWinshape xin
;default values
iFFTsize = (iFFTsize == 0) ? 1024 : iFFTsize
iOverlap = (iOverlap == 0) ? 256 : iOverlap
iWinSize = (iWinSize == 0) ? iFFTsize : iWinSize
;fill buffer
if kTrig == 1 then
  ilen filelen Sfile
  kNumCycles = ilen * kr
  kcycle init 0
  while kcycle < kNumCycles do
    ain soundin Sfile
    fftin pvsanal ain, iFFTsize, iOverlap, iWinSize, iWinshape
    ibuf, ktim pvsbuffer fftin, ilen + (iFFTsize / sr)
    kcycle += 1
  od
endif
xout ibuf, ilen, ktim
endop

instr simple_time_stretch
gibuffer, gilen, k0 FileToPvsBuf timeinstk(), "fox.wav"
ktmpnt linseg 1.6, p3, gilen
fread pvsbufread ktmpnt, gibuffer
aout pvsynth fread
out aout, aout
endin

instr tremor_time_stretch
ktmpnt = linseg:k(0,p3,gilen) + randi:k(1/5,10)
fread pvsbufread ktmpnt, gibuffer
aout pvsynth fread
out aout, aout
endin

</CsInstruments>
<CsScore>
i 1 0 10
i 2 11 20
</CsScore>
</CsoundSynthesizer>
;example by joachim heintz

```

The `mincer` opcode also provides a high-quality time- and pitch-shifting algorithm. Other than `pvsanal` and `pvsbufread` it already transforms the f-signal back to time domain, thus outputting an audio signal.

Pitch shifting

Simple pitch shifting can be carried out by the opcode `pvscale`. All the frequency data in the f-signal are scaled by a certain value. Multiplying by 2 results in transposing by an octave upwards; multiplying by 0.5 in transposing by an octave downwards. For accepting cent values instead of ratios as input, the `cent` opcode can be used.

EXAMPLE 05I04_pvscale.csd

```
<CsoundSynthesizer>
<CsOptions>
-odac --env:SSDIR+=../SourceMaterials
</CsOptions>
<CsInstruments>
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>
;example by joachim heintz
```

Pitch shifting via FFT resynthesis is very simple in general, but rather more complicated in detail. With speech for instance, there is a problem because of the formants. If we simply scale the frequencies, the formants are shifted, too, and the sound gets the typical *helium voice* effect. There are some parameters in the `pvscale` opcode, and some other pvs-opcodes which can help to avoid

this, but the quality of the results will always depend to an extend upon the nature of the input sound.

As mentioned above, simple pitch shifting can also be performed via [pvstanal](#) or [mincer](#).

Spectral Shifting

Rather than multiplying the bin frequencies by a scaling factor, which results in pitch shifting, it is also possible to *add* a certain amount to the single bin frequencies. This results in an effect which is called frequency shifting. It resembles the shifted spectra in ring modulation which has been described at the end of chapter [04C](#).

The frequency-domain spectral shifting which is performed by the [pvshift](#) opcode has some important differences compared to the time-domain ring modulation:

- Frequencies are only added or subtracted, not both.
- A lowest frequency can be given. Below this frequency the spectral content will be left untouched; only above addition/subtraction will be processed.
- Some options are implemented which try to preserve formants. This is of particular interest when working with human voice.

The following example performs some different shifts on a single viola tone.

EXAMPLE 05I05_pvshift.csd

```

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

instr Shift
aSig diskin "BratscheMono.wav"
fSig pvstanal aSig, 1024, 256, 1024, 1
fShift pvshift fSig, p4, p5
aShift pvsynth fShift
out aShift, aShift
endin

</CsInstruments>
<CsScore>
i "Shift" 0 9 0 0 ;no shift (base freq is 218)
i . + . 50 0 ;shift all by 50 Hz
i . + . 150 0 ;shift all by 150 Hz
i . + . 500 0 ;shift all by 500 Hz
i . + . 150 230 ;only above 230 Hz by 150 Hz
i . + . . 460 ;only above 460 Hz
i . + . . 920 ;only above 920 Hz
</CsScore>
</CsoundSynthesizer>
;example by joachim heintz

```

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 represent an interpolation between A and B, the cross synthesis is more flexible and adjustable. This is what [pvs voc](#) does.
- Combine the frequencies of sound A with the amplitudes of sound B. Give user 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 of 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 05I06_phase_vocoder.csd

```
<CsoundSynthesizer>
<CsOptions>
-odac --env:SSDIR+=../SourceMaterials
</CsOptions>
<CsInstruments>
sr = 44100
ksmps = 32
nchnls = 1
0dbfs = 1

;store the samples in function tables (buffers)
gifulA    ftgen    0, 0, 0, 1, "fox.wav", 0, 0, 1
gifulB    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, gifulA, gidet, giwrap, giskip,\ 
                  gifftsiz, giovlp, githresh
fsigB    pvstanal 1, giamp, gipitch, gifulB, gidet, giwrap, giskip,\ 
                  gifftsiz, giovlp, githresh
fvoc     pvs voc fsigA, fsigB, 1, 1
aout     pvsynth fvoc
aenv     linen    aout, .1, p3, .5

```

```

        out      aenv
endin

</CsInstruments>
<CsScore>
i 1 0 11
</CsScore>
</CsoundSynthesizer>
;example by joachim heintz

```

The next example introduces *pvcross*:

EXAMPLE 05I07_pvcross.csd

```

<CsoundSynthesizer>
<CsOptions>
-odac --env:SSDIR+=../SourceMaterials
</CsOptions>
<CsInstruments>
sr = 44100
ksmps = 32
nchnls = 1
0dbfs = 1

;store the samples in function tables (buffers)
gfilA    ftgen    0, 0, 0, 1, "BratscheMono.wav", 0, 0, 1
gfilB    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, gfilA, gidet, giwrap, giskip,\ 
                  gifftsiz, giovlp, githresh
fsigB    pvstanal  .5, giamp, gipitch, gfilB, gidet, giwrap, giskip,\ 
                  gifftsiz, giovlp, githresh
fcross   pvcross   fsigA, fsigB, 0, 1
aout     pvsynth   fcross
aenv     linen     aout, .1, p3, .5
        out      aenv
endin

</CsInstruments>
<CsScore>
i 1 0 11
</CsScore>
</CsoundSynthesizer>
;example by joachim heintz

```

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 is dependent upon the amplitudes of sound B, and if the amplitudes are strong enough, you will hear a resonating effect:

EXAMPLE 05I08_pvsfilter.csd

```

<CsoundSynthesizer>
<CsOptions>
-odac --env:SSDIR+=../SourceMaterials
</CsOptions>
<CsInstruments>
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      aenv
endin

</CsInstruments>
<CsScore>
i 1 0 11
</CsScore>
</CsoundSynthesizer>
;example by joachim heintz

```

Sound Quality in FFT Signals

Artifacts can easily occur in several situations of applying FFT. In example 05I02 we have seen how it is a side effect in extreme time stretching of spoken word. The opcodes `pvsSmooth` and `pvsBlur` can be a remedy against it, or at least a relief. The adjustment of the parameters are crucial here:

- For `pvsSmooth`, the `kacf` and the `kfcf` parameter apply a low pass filter on the amplitudes and the frequencies of the *f*-signal. The range is 0-1 each, where 0 is the lowest and 1 the highest cutoff frequency. Lower values will smooth more, so the effect will be stronger.
- For `pvsBlur`, the `kblurtime` depicts the time in seconds during which the single FFT windows will be averaged.

This is a trial to reduce the amount of artefacts. Note that `pvsanal` actually has the best method to reduce artifacts in spoken word, as it can leave onsets unstretched (`kdetect` which is on by default).

EXAMPLE 05I09_pvsSmooth_pvsblur.csd

```

<CsoundSynthesizer>
<CsOptions>
-m 128
</CsOptions>
<CsInstruments>
sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

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

instr Raw
fStretch pvstanal 1/10, 1, 1, gifil, 0 ;kdetect is turned off
aStretch pvsynth fStretch
out aStretch, aStretch
endin

instr Smooth
iAmpCutoff = p4 ;0-1
iFreqCutoff = p5 ;0-1
fStretch pvstanal 1/10, 1, 1, gifil, 0
fSmooth pvsSmooth fStretch, iAmpCutoff, iFreqCutoff
aSmooth pvsynth fSmooth
out aSmooth, aSmooth
endin

instr Blur
iBlurtime = p4 ;sec
fStretch pvstanal 1/10, 1, 1, gifil, 0
fBlur pvsblur fStretch, iBlurtime, 1
aSmooth pvsynth fBlur
out aSmooth, aSmooth
endin

instr Smooth_var
fStretch pvstanal 1/10, 1, 1, gifil, 0
kAmpCut randomi .001, .1, 10, 3
kFreqCut randomi .05, .5, 50, 3
fSmooth pvsSmooth fStretch, kAmpCut, kFreqCut
aSmooth pvsynth fSmooth
out aSmooth, aSmooth
endin

instr Blur_var
kBBlurtime randomi .005, .5, 200, 3
fStretch pvstanal 1/10, 1, 1, gifil, 0
fBlur pvsblur fStretch, kBBlurtime, 1
aSmooth pvsynth fBlur
out aSmooth, aSmooth
endin

instr SmoothBlur
iacf = p4
ifcf = p5
iblurtime = p6

```

```

fanal pvstanal 1/10, 1, 1, gifil, 0
fsmot pvsSmooth fanal, iacf, ifcf
fblur pvsBlur fsmot, iblurtime, 1
a_smt pvsynth fblur
aOut linenr a_smt, 0, iblurtime*2, .01
    out aOut, aOut
endin

</CsInstruments>
<CsScore>
i "Raw" 0 16
i "Smooth" 17 16 .01 .1
i "Blur" 34 16 .2
i "Smooth_var" 51 16
i "Blur_var" 68 16
i "SmoothBlur" 85 16 1 1 0
i . 102 . .1 1 .25
i . 119 . .01 .1 .5
i . 136 . .001 .01 .75
i . 153 . .0001 .001 1
</CsScore>
</CsoundSynthesizer>
;example by joachim heintz and farhad ilaghi hosseini

```

Retrieving Single Bins from FFT

It is not only possible to work with the full data set of the Fourier Transform, but to select single bins (amplitude-frequency pairs) from it. This can be useful for specialized resynthesis or for using the analysis data in any way.

pvsbin

The most fundamental extraction of single bins can be done with the `pvsbin` opcode. It takes the f-signal and the bin number as input, and returns the amplitude and the frequency of the bin. These values can be used to drive an oscillator which resynthesizes this bin.

The next example shows three different applications. At first, instr `SingleBin` is called four times, performing bin 10, 20, 30 and 40. Then instr `FourBins` calls the four instances of `SingleBin` at the same time, so we hear the four bins together. Finally, instr `SlidingBins` uses the fact that the bin number can be given to `pvsbin` as k-rate variable. The line `kBin randomi 1,50,200,3` produces changing bins with a rate of 200 Hz, between bin 1 and 50.

Note that we are always smoothing the bin amplitudes `kAmp` by applying `port(kAmp,.01)`. Raw `kAmp` instead will get clicks, whereas `port(kAmp,.1)` would remove the small attacks.

EXAMPLE 05I10_pvsbin.csd

```

<CsoundSynthesizer>
<CsOptions>
-odac -m128
</CsOptions>
<CsInstruments>

sr = 44100

```

```

ksmps = 32
nchnls = 2
0dbfs = 1

instr SingleBin
iBin = p4 //bin number
aSig diskin "fox.wav"
fSig pvsanal aSig, 1024, 256, 1024, 1
kAmp, kFreq pvsbin fSig, iBin
aBin poscil port(kAmp,.01), kFreq
aBin *= iBin/10
out aBin, aBin
endin

instr FourBins
iCount = 1
while iCount < 5 do
    schedule("SingleBin",0,3,iCount*10)
    iCount += 1
od
endin

instr SlidingBins
kBin randomi 1,50,200,3
aSig diskin "fox.wav"
fSig pvsanal aSig, 1024, 256, 1024, 1
kAmp, kFreq pvsbin fSig, int(kBin)
aBin poscil port(kAmp,.01), kFreq
aBin *= kBin/10
out aBin, aBin
endin

</CsInstruments>
<CsScore>
i "SingleBin" 0 3 10
i . + . 20
i . + . 30
i . + . 40
i "FourBins" 13 3
i "SlidingBins" 17 3
</CsScore>
</CsoundSynthesizer>
;example by joachim heintz

```

pvtrace

Another approach to retrieve a selection of bins is done by the opcode **pvtrace**. Here, only the N loudest bins are written in the *f* signal which this opcode outputs.

This is a simple example first which lets *pvtrace* play in sequence the 1, 2, 4, 8 and 16 loudest bins.

EXAMPLE 05I11_pvtrace_simple.csd

```

<CsoundSynthesizer>
<CsOptions>
-o dac
</CsOptions>
<CsInstruments>
sr = 44100

```

```

ksmps = 32
nchnls = 2
0dbfs = 1

instr Simple
aSig diskin "fox.wav"
fSig pvsanal aSig, 1024, 256, 1024, 1
fTrace pvstrace fSig, p4
aTrace pvsynth fTrace
out aTrace, aTrace
endin

</CsInstruments>
<CsScore>
i "Simple" 0 3 1
i . + . 2
i . + . 4
i . + . 8
i . + . 16
</CsScore>
</CsoundSynthesizer>
;example by joachim heintz

```

An optional second output of `pvstrace` returns an array with the *kn* bin numbers which are most prominent. As a demonstration, this example passes only the loudest bin to `pvsbin` and resynthesizes it with an oscillator unit.

EXAMPLE 05I12_pvstrace_array.csd

```

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

instr LoudestBin
aSig diskin "fox.wav"
fSig pvsanal aSig, 1024, 256, 1024, 1
fTrace, kBins[] pvstrace fSig, 1, 1
kAmp, kFreq pvsbin fSig, kBins[0]
aLoudestBin poscil port(kAmp,.01), kFreq
out aLoudestBin, aLoudestBin
endin

</CsInstruments>
<CsScore>
i "LoudestBin" 0 3
</CsScore>
</CsoundSynthesizer>
;example by joachim heintz

```


05 K. ATS RESYNTHESIS

The ATS Technique

General overview

The ATS technique (*Analysis-Transformation-Synthesis*) was developed by Juan Pampin. A comprehensive explanation of this technique can be found in his *ATS Theory*¹ but, essentially, it may be said that it represents two aspects of the analyzed signal: the deterministic part and the stochastic or residual part. This model was initially conceived by Julius Orion Smith and Xavier Serra,² but ATS refines certain aspects of it, such as the weighting of the spectral components on the basis of their *Signal-to-Mask-Ratio (SMR)*.³

The deterministic part consists in sinusoidal trajectories with varying amplitude, frequency and phase. It is achieved by means of the depuration of the spectral data obtained using *STFT (Short-Time Fourier Transform)* analysis.

The stochastic part is also termed *residual*, because it is achieved by subtracting the deterministic signal from the original signal. For such purposes, the deterministic part is synthesized preserving the phase alignment of its components in the second step of the analysis. The residual part is represented with noise variable energy values along the 25 critical bands.⁴

The ATS technique has the following advantages:

1. The splitting between deterministic and stochastic parts allows an independent treatment of two different qualitative aspects of an audio signal.
2. The representation of the deterministic part by means of sinusoidal trajectories improves the information and presents it on a way that is much closer to the way that musicians think of sound. Therefore, it allows many *classical* spectral transformations (such as the suppression of partials or their frequency warping) in a more flexible and conceptually clearer

¹Juan Pampin, 2011, [ATS_theory](#)

²Xavier Serra and Julius O. Smith III, 1990, A Sound Analysis/Synthesis System Based on a Deterministic plus Stochastic Decomposition, Computer Music Journal, Vol.14, 4, MIT Press, USA

³Eberhard Zwicker and Hugo Fastl, 1990, Psychoacoustics, Facts and Models. Springer, Berlin, Heidelberg

⁴Cf. Zwicker/Fastl (above footnote)

way.

3. The representation of the residual part by means of noise values among the 25 critical bands simplifies the information and its further reconstruction. Namely, it is possible to overcome the common artifacts that arise in synthesis using oscillator banks or *IDFT*, when the time of a noisy signal analyzed using a *FFT* is warped.

The ATS File Format

Instead of storing the *crude* data of the *FFT* analysis, the ATS files store a representation of a digital sound signal in terms of sinusoidal trajectories (called *partials*) with instantaneous frequency, amplitude, and phase changing along temporal frames. Each frame has a set of partials, each having (at least) amplitude and frequency values (phase information might be discarded from the analysis). Each frame might also contain noise information, modeled as time-varying energy in the 25 critical bands of the analysis residual. All the data is stored as 64 bits floats in the host's byte order.

The ATS files start with a header at which their description is stored (such as frame rate, duration, number of sinusoidal trajectories, etc.). The header of the ATS files contains the following information:

1. ats-magic-number (just the arbitrary number 123. for consistency checking)
2. sampling-rate (samples/sec)
3. frame-size (samples)
4. window-size (samples)
5. partials (number of partials)
6. frames (number of frames)
7. ampmax (max. amplitude)
8. frqmax (max. frequency)
9. dur (duration in sec.)
10. type (frame type, see below)

The ATS frame type may be, at present, one of the four following:

Type 1: only sinusoidal trajectories with amplitude and frequency data. Type 2: only sinusoidal trajectories with amplitude, frequency and phase data. Type 3: sinusoidal trajectories with amplitude, and frequency data as well as residual data. Type 4: sinusoidal trajectories with amplitude, frequency and phase data as well as residual data.

So, after the header, an ATS file with frame type 4, *np* number of partials and *nf* frames will have:

```
Frame 1:
  time tag
  Amp.of partial 1,   Freq. of partial 1, Phase of partial 1
  .....
  .....
  Amp.of partial np,   Freq. of partial np, Phase of partial np

  Residual energy value for critical band 1
  .....
  .....
  Residual energy value for critical band 25
```

```
.....  

Frame nf:  

    time tag  

    Amp.of partial 1,   Freq. of partial 1, Phase of partial 1  

    .....  

    .....  

    Amp.of partial np,   Freq. of partial np, Phase of partial np  

    .....  

    Residual energy  value for  critical band 1  

    .....  

    .....  

    Residual energy  value for  critical band 25
```

As an example, an ATS file of frame type 4, with 100 frames and 10 partials will need:

- A header with 10 double floats values.
- 100_10_3 double floats for storing the Amplitude, Frequency and Phase values of 10 partials along 100 frames.
- 25 * 100 double floats for storing the noise information of the 25 critical bands along 100 frames.
- 100 double floats for storing the time tag information for each frame

Header:	10 * 8	=	80 bytes
Deterministic data:	3000 * 8	=	24000 bytes
Residual data:	2500 * 8	=	20000 bytes
Time tags data	100	=	800 bytes
 Total:		80 + 24000 + 20000 + 800	= 44880 bytes

The following Csound code shows how to retrieve the data of the header of an ATS file.

EXAMPLE 05K01_ats_header.csd

```
<CsoundSynthesizer>
<CsOptions>
-n -m0
</CsOptions>
<CsInstruments>
sr = 44100
ksmps = 32
nchnls = 1
0dbfs = 1

;Some macros
#define ATS_SR # 0 # ;sample rate      (Hz)
#define ATS_FS # 1 # ;frame size       (samples)
#define ATS_WS # 2 # ;window Size     (samples)
#define ATS_NP # 3 # ;number of Partials
#define ATS_NF # 4 # ;number of Frames
#define ATS_AM # 5 # ;maximum Amplitude
#define ATS_FM # 6 # ;maximum Frequency (Hz)
#define ATS_DU # 7 # ;duration        (seconds)
#define ATS_TY # 8 # ;ATS file Type

instr 1
iats_file=p4
;instr1 just reads the file header and loads its data into several variables
;and prints the result in the Csound prompt.
```

```

i_sampling_rate      ATSinfo iats_file,    $ATS_SR
i_frame_size         ATSinfo iats_file,    $ATS_FS
i_window_size        ATSinfo iats_file,    $ATS_WS
i_number_of_partials ATSinfo iats_file,    $ATS_NP
i_number_of_frames   ATSinfo iats_file,    $ATS_NF
i_max_amp            ATSinfo iats_file,    $ATS_AM
i_max_freq           ATSinfo iats_file,    $ATS_FM
i_duration           ATSinfo iats_file,    $ATS_DU
i_ats_file_type      ATSinfo iats_file,    $ATS_TY

print i_sampling_rate
print i_frame_size
print i_window_size
print i_number_of_partials
print i_number_of_frames
print i_max_amp
print i_max_freq
print i_duration
print i_ats_file_type

endin

</CsInstruments>
<CsScore>
;change to put any ATS file you like
#define ats_file #"basoon-C4.ats"#
;      st      dur      atsfile
i1    0       0       $ats_file
e
</CsScore>
</CsoundSynthesizer>
;Example by Oscar Pablo Di Liscia

```

Performing ATS Analysis with the ATSA Command-line Utility of Csound

All the Csound Opcodes devoted to ATS Synthesis need to read an ATS Analysis file. ATS was initially developed for the *CLM* environment (*Common Lisp Music*), but at present there exist several GNU applications that can perform ATS analysis, among them the *Csound* Package command-line utility ATSA which is based on the ATSA program (Di Liscia, Pampin, Moss) and was ported to Csound by Istvan Varga. The ATSA program (Di Liscia, Pampin, Moss) may be obtained at <https://github.com/jamezilla/ats/tree/master/ats>

Graphical Resources for Displaying ATS Analysis Files

If a plot of the ATS files is required, the ATSH software (Di Liscia, Pampin, Moss) may be used. ATSH is a C program that uses the GTK graphic environment. The source code and compilation directives can be obtained at <https://github.com/jamezilla/ats/tree/master/ats>

Another very good GUI program that can be used for such purposes is Qatsh, a Qt 4 port

by Jean-Philippe Meuret. This one can be obtained at <https://sourceforge.net/p/speed-dreams/code/HEAD/tree/subprojects/soundeditor/>

Parameters Explanation and Proper Analysis Settings

The analysis parameters are somewhat numerous, and must be carefully tuned in order to obtain good results. A detailed explanation of the meaning of these parameters can be found at <https://csound.com/docs/manual/UtilityAtsa.html>

In order to get a good analysis, the sound to be analysed should meet the following requirements:

1. The ATS analysis was meant to analyse isolated, individual sounds. This means that the analysis of sequences and/or superpositions of sounds, though possible, is not likely to render optimal results.
2. Must have been recorded with a good signal-to-noise ratio, and should not contain unwanted noises.
3. Must have been recorded without reverberation and/or echoes.

A good ATS analysis should meet the following requirements:

1. Must have a good temporal resolution of the frequency, amplitude, phase and noise (if any) data. The tradeoff between temporal and frequency resolution is a very well known issue in FFT based spectral analysis.
2. The Deterministic and Stochastic (also termed *residual) data must be reasonably separated in their respective ways of representation. This means that, if a sound has both, deterministic and stochastic data, the former must be represented by sinusoidal trajectories, whilst the latter must be represented by energy values among the 25 critical bands. This allows a more effective treatment of both types of data in the synthesis and transformation processes.
3. If the analysed sound is pitched, the sinusoidal trajectories (Deterministic) should be as stable as possible and ordered according the original sound harmonics. This means that the first trajectory should represent the first (fundamental) harmonic, the second trajectory should represent the second harmonic, and so on. This allow to perform easily further transformation processes during resynthesis (such as, for example, selecting the odd harmonics to give them a different treatment than the others).

Whilst the first requirement is unavoidable, in order to get a useful analysis, the second and third ones are sometimes almost impossible to meet in full and their accomplishment depends often on the user objectives.

Synthesizing ATS Analysis Files

Synthesis Techniques Applied to ATS.

The synthesis techniques that are usually applied in order to get a synthesized sound that resembles the original sound as much as possible are detailed explained in Pampin 2011⁵ and di Liscia 2013⁶. However, it is worth pointing out that once the proper data is stored in an analysis file, the user is free to read and apply to this data any reasonable transformation/synthesis technique/s, thereby facilitating the creation of new and interesting sounds that need not be similar nor resemble the original sound.

Csound Opcodes for Reading ATS Data Files

The opcodes `ATSread`, `ATSreadnz`, `ATSBufread`, `ATSInterpread` and `ATSPartialtap` were essentially developed to read ATS data from ATS files.

`ATSread`

This opcode reads the deterministic ATS data from an ATS file. It outputs frequency/amplitude pairs of a sinusoidal trajectory corresponding to a specific partial number, according to a time pointer that must be delivered. As the unit works at *k*-rate, the frequency and amplitude data must be interpolated in order to avoid unwanted clicks in the resynthesis.

The following example reads and synthesizes the 10 partials of an ATS analysis corresponding to a steady 440 cps flute sound. Since the instrument is designed to synthesize only one partial of the ATS file, the mixing of several of them must be obtained performing several notes in the score (the use of Csound's macros is strongly recommended in this case). Though not the most practical way of synthesizing ATS data, this method facilitates individual control of the frequency and amplitude values of each one of the partials, which is not possible any other way. In the example that follows, even numbered partials are attenuated in amplitude, resulting in a sound that resembles a clarinet. Amplitude and frequency envelopes could also be used in order to affect a time changing weighting of the partials. Finally, the amplitude and frequency values could be used to drive other synthesis units, such as filters or FM synthesis networks of oscillators.

EXAMPLE 05K02_atsread.csd

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

⁵Juan Pampin, 2011, ATS_theory (see footnote 1)

⁶Oscar Pablo Di Liscia, 2013, A Pure Data toolkit for real-time synthesis of ATS spectral data <http://lac.linuxaudio.org/2013/papers/26.pdf>

```

instr 1
iamp = p4                      ;amplitude scaler
ifreq = p5                       ;frequency scaler
ipar = p6                        ;partial required
itab = p7                         ;audio table
iatstsfile = p8                  ;ats file

idur ATSinfo iatsfile, 7        ;get duration

ktime line 0, p3, idur          ;time pointer

kfreq, kamp ATSread ktime, iatsfile, ipar ;get frequency and amplitude values
aamp      interp  kamp                ;interpolate amplitude values
afreq      interp  kfreq               ;interpolate frequency values
aout oscil3 aamp*iamp, afreq*ifreq, itab ;synthesize with amp and freq scaling

        out      aout
endin

</CsInstruments>
<CsScore>
; sine wave table
f 1 0 16384 10 1
#define atstsfile #"/flute-A5.ats"#

;    start   dur     amp     freq    par    tab    atstsfile
i1    0       3      1       1       1      1      $atsfile
i1    0       .      .1      .       2      .      $atsfile
i1    0       .      1       .       3      .      $atsfile
i1    0       .      .1      .       4      .      $atsfile
i1    0       .      1       .       5      .      $atsfile
i1    0       .      .1      .       6      .      $atsfile
i1    0       .      1       .       7      .      $atsfile
i1    0       .      .1      .       8      .      $atsfile
i1    0       .      1       .       9      .      $atsfile
i1    0       .      .1      .      10      .      $atsfile
e
</CsScore>
</CsoundSynthesizer>
;example by Oscar Pablo Di Liscia

```

We can use arrays to simplify the code in this example, and to choose different numbers of partials:

EXAMPLE 05K03_atsread2.csd

```

<CsoundSynthesizer>
<CsOptions>
-o dac
</CsOptions>
<CsInstruments>
sr      = 44100
ksmps   = 32
nchnls  = 1
0dbfs   = 1

gS_ATS_file =      "flute-A5.ats" ;ats file
giSine     ftgen      0, 0, 16384, 10, 1 ; sine wave table

instr Master ;call instr "Play" for each partial
iNumParts =          p4 ;how many partials to synthesize
idur      ATSinfo    gS_ATS_file, 7 ;get ats file duration

```

```

iAmps[]    array      1, .1 ;array for even and odd partials
iParts[]   genarray   1,iNumParts ;creates array [1, 2, ..., iNumParts]

indx       =          0 ;initialize index
;loop for number of elements in iParts array
until indx == iNumParts do
;call an instance of instr "Play" for each partial
  event_i   "i", "Play", 0, p3, iAmps[indx%2], iParts[indx], idur
indx     +=          1 ;increment index
od ;end of do ... od block

  turnoff ;turn this instrument off as job has been done
endin

instr Play
iamp      =          p4 ;amplitude scaler
ipar       =          p5 ;partial required
idur       =          p6 ;ats file duration

ktime     line      0, p3, idur ;time pointer

kfreq, kamp ATSread ktime, gS_ATS_file, ipar
aamp      interp   kamp ;interpolate amplitude values
afreq      interp   kfreq ;interpolate frequency values
aout      oscil3  aamp*iamp, afreq, giSine ;synthesize with amp scaling

  out      aout
endin
</CsInstruments>
<CsScore>
;           strt dur number of partials
i "Master" 0    3    1
i .         +    .    3
i .         +    .    10
</CsScore>
</CsoundSynthesizer>
;example by Oscar Pablo Di Liscia and Joachim Heintz

```

ATSreadnz

This opcode is similar to *ATSread* in the sense that it reads the noise data of an ATS file, delivering k-rate energy values for the requested critical band. In order to this Opcode to work, the input ATS file must be either type 3 or 4 (types 1 and 2 do not contain noise data). *ATSreadnz* is simpler than *ATSread*, because whilst the number of partials of an ATS file is variable, the noise data (if any) is stored always as 25 values per analysis frame each value corresponding to the energy of the noise in each one of the critical bands. The three required arguments are: a time pointer, an ATS file name and the number of critical band required (which, of course, must have a value between 1 and 25).

The following example is similar to the previous. The instrument is designed to synthesize only one noise band of the ATS file, the mixing of several of them must be obtained performing several notes in the score. In this example the synthesis of the noise band is done using Gaussian noise filtered with a resonator (i.e., band-pass) filter. This is not the method used by the ATS synthesis Opcodes that will be further shown, but its use in this example is meant to lay stress again on the fact that the use of the ATS analysis data may be completely independent of its generation. In this case, also, a macro that performs the synthesis of the 25 critical bands was programmed. The ATS file used correspond to a female speech sound that lasts for 3.633 seconds, and in the

examples is stretched to 10.899 seconds, that is three times its original duration. This shows one of the advantages of the Deterministic plus Stochastic data representation of ATS: the stochastic ("noisy") part of a signal may be stretched in the resynthesis without the artifacts that arise commonly when the same data is represented by cosine components (as in the FFT based resynthesis). Note that, because the Stochastic noise values correspond to energy (i.e., intensity), in order to get the proper amplitude values, the square root of them must be computed.

EXAMPLE 05K04_atsreadnz.csd

```

<CsoundSynthesizer>
<CsOptions>
-o dac
</CsOptions>
<CsInstruments>
sr = 44100
ksmps = 32
nchnls = 1
0dbfs = 1

instr 1
itabc = p7           ;table with the 25 critical band frequency edges
iscal = 1             ;reson filter scaling factor
iamp = p4              ;amplitude scaler
iband = p5              ;energy band required
if1    table  iband-1, itabc ;lower edge
if2    table  iband, itabc   ;upper edge
idif   = if2-if1
icf    = if1 + idif*.5      ;center frequency value
ibw    = icf*p6            ;bandwidth
iatsfile = p8            ;ats file name

idur    ATSinfo iatsfile, 7      ;get duration

ktime   line    0, p3, idur      ;time pointer

ken     ATSreadnz ktime, iatsfile, iband ;get frequency and amplitude values
anoise  gauss 1
aout    reson anoise*sqrt(ken), icf, ibw, iscal ;synthesize with scaling

        out aout*iamp
endin

</CsInstruments>
<CsScore>
; sine wave table
f1 0 16384 10 1
;the 25 critical bands edge's frequencies
f2 0 32 -2 0 100 200 300 400 510 630 770 920 1080 1270 1480 1720 2000 2320 \
    2700 3150 3700 4400 5300 6400 7700 9500 12000 15500 20000

;an ats file name
#define atsfile #"female-speech.ats"#

;a macro that synthesize the noise data along all the 25 critical bands
#define all_bands(start'dur'amp'bw'file)
#
i1    $start $dur   $amp   1      $bw    2      $file
i1    .       .     .      2      .       .      $file
i1    .       .     .      3      .       .      .
i1    .       .     .      4      .       .      .
i1    .       .     .      5      .       .      .

```

```
i1 . . . . 6 .
i1 . . . . 7 .
i1 . . . . 8 .
i1 . . . . 9 .
i1 . . . . 10 .
i1 . . . . 11 .
i1 . . . . 12 .
i1 . . . . 13 .
i1 . . . . 14 .
i1 . . . . 15 .
i1 . . . . 16 .
i1 . . . . 17 .
i1 . . . . 18 .
i1 . . . . 19 .
i1 . . . . 20 .
i1 . . . . 21 .
i1 . . . . 22 .
i1 . . . . 23 .
i1 . . . . 24 .
i1 . . . . 25 .
#
;ditto...original sound duration is 3.633 secs.
;stretched 300%
$all_bands(0'10.899'1'.05'$atsfile)

e
</CsScore>
</CsoundSynthesizer>
;example by Oscar Pablo Di Liscia
```

ATSBufread, ATSInterpread, ATSPartialtap.

The **ATSBufread** opcode reads an ATS file and stores its frequency and amplitude data into an internal table. The first and third input arguments are the same as in the **ATSread** and the **ATSreadnz** Opcodes: a time pointer and an ATS file name. The second input argument is a frequency scaler. The fourth argument is the number of partials to be stored. Finally, this Opcode may take two optional arguments: the first partial and the increment of partials to be read, which default to 0 and 1 respectively.

Although this opcode does not have any output, the ATS frequency and amplitude data is available to be used by other opcode. In this case, two examples are provided, the first one uses the **ATSInterpread** opcode and the second one uses the **ATSPartialtap** opcode.

The **ATSInterpread** opcode reads an ATS table generated by the **ATSBufread** opcode and outputs amplitude values interpolating them between the two amplitude values of the two frequency trajectories that are closer to a given frequency value. The only argument that this opcode takes is the desired frequency value.

The following example synthesizes five sounds. All the data is taken from the ATS file *test.ats*. The first and final sounds match the two frequencies closer to the first and the second partials of the analysis file and have their amplitude values closer to the ones in the original ATS file. The other three sounds (second, third and fourth), have frequencies that are in-between the ones of the first and second partials of the ATS file, and their amplitudes are scaled by an interpolation between the amplitudes of the first and second partials. The more the frequency requested approaches the one of a partial, the more the amplitude envelope rendered by **ATSInterpread** is similar to the

one of this partial. So, the example shows a gradual morphing between the amplitude envelope of the first partial to the amplitude envelope of the second according to their frequency values.

EXAMPLE 05K05_atsinterpread.csd

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

instr 1

iamp =      p4          ;amplitude scaler
ifreq =      p5          ;frequency scaler
iatfile =   p7          ;atsfile
itab =       p6          ;audio table
ifreqscal = 1           ;frequency scaler
ipars    ATSinfo iatsfile, 3 ;how many partials
idur     ATSinfo iatsfile, 7 ;get duration
ktime    line   0, p3, idur ;time pointer

        ATSbufread ktime, ifreqscal, iatsfile, ipars ;reads an ATS buffer
kamp    ATSinterpread ifreq           ;get the amp values according to freq
aamp    interp kamp                 ;interpolate amp values
aout    oscil3 aamp, ifreq, itab     ;synthesize

        out aout*iamp
endin

</CsInstruments>
<CsScore>
; sine wave table
f 1 0 16384 10 1
#define atsfile #"/test.ats"#

; start dur amp freq atab atsfile
i1 0    3   1    440  1    $atsfile    ;first partial
i1 +   3   1    550  1    $atsfile    ;closer to first partial
i1 +   3   1    660  1    $atsfile    ;half way between both
i1 +   3   1    770  1    $atsfile    ;closer to second partial
i1 +   3   1    880  1    $atsfile    ;second partial
e
</CsScore>
</CsoundSynthesizer>
;example by Oscar Pablo Di Liscia
```

The `ATSPartialtap` opcode reads an ATS table generated by the `ATSbufread` Opcode and outputs the frequency and amplitude k-rate values of a specific partial number. The example presented here uses four of these opcodes that read from a single ATS buffer obtained using `ATSbufread` in order to drive the frequency and amplitude of four oscillators. This allows the mixing of different combinations of partials, as shown by the three notes triggered by the designed instrument.

EXAMPLE 05K06_atspartialtap.csd

```

<CsoundSynthesizer>
<CsOptions>
-o dac
</CsOptions>
<CsInstruments>
sr = 44100
ksmps = 32
nchnls = 1
0dbfs = 1

instr 1
iamp = p4/4          ;amplitude scaler
ifreq = p5            ;frequency scaler
itab = p6             ;audio table
ip1 = p7              ;first partial to be synthesized
ip2 = p8              ;second partial to be synthesized
ip3 = p9              ;third partial to be synthesized
ip4 = p10             ;fourth partial to be synthesized
iatfile = p11         ;atsfile

ipars    ATSinfo iatfile, 3      ;get how many partials
idur     ATSinfo iatfile, 7      ;get duration

ktime   line   0, p3, idur      ;time pointer

        ATSbufread ktime, ifreq, iatfile, ipars ;reads an ATS buffer

kf1,ka1 ATSPartialtap ip1 ;get the amp values according each partial number
af1      interp kf1
aa1      interp ka1
kf2,ka2 ATSPartialtap ip2      ;ditto
af2      interp kf2
aa2      interp ka2
kf3,ka3 ATSPartialtap ip3      ;ditto
af3      interp kf3
aa3      interp ka3
kf4,ka4 ATSPartialtap ip4      ;ditto
af4      interp kf4
aa4      interp ka4

a1      oscil3 aa1, af1*ifreq, itab    ;synthesize each partial
a2      oscil3 aa2, af2*ifreq, itab    ;ditto
a3      oscil3 aa3, af3*ifreq, itab    ;ditto
a4      oscil3 aa4, af4*ifreq, itab    ;ditto

        out (a1+a2+a3+a4)*iamp
endin

</CsInstruments>
<CsScore>
; sine wave table
f 1 0 16384 10 1
#define atsfile #"/boe-A5.ats"

; start dur amp freq atab part#1 part#2 part#3 part#4 atsfile
i1 0 3 10 1 1 1 5 11 13 $atsfile
i1 + 3 7 1 1 1 6 14 17 $atsfile
i1 + 3 400 1 1 15 16 17 18 $atsfile

e
</CsScore>

```

```
</CsoundSynthesizer>
;example by Oscar Pablo Di Liscia
```

Synthesizing ATS data: ATSadd, ATSandnz, ATSsinnoi, ATScross.

The four opcodes that will be presented in this section synthesize ATS analysis data internally and allow for some modifications of these data as well. A significant difference to the preceding opcodes is that the synthesis method cannot be chosen by the user. The synthesis methods used by all of these opcodes are fully explained in *Juan Pampin 2011* and *Oscar Pablo Di Liscia 2013* (see footnotes 1 and 6).

The **ATSadd** opcode synthesizes deterministic data from an ATS file using an array of table lookup oscillators whose amplitude and frequency values are obtained by linear interpolation of the ones in the ATS file according to the time of the analysis requested by a time pointer. The frequency of all the partials may be modified at k-rate, allowing shifting and/or frequency modulation. An ATS file, a time pointer and a function table are required. The table is supposed to contain either a cosine or a sine function, but nothing prevents the user from experimenting with other functions. Some care must be taken in the last case, so as not to produce foldover (frequency aliasing). The user may also request a number of partials smaller than the number of partials of the ATS file (by means of the *inpars* variable in the example below). There are also two optional arguments: a partial offset (i.e., the first partial that will be taken into account for the synthesis, by means of the *ipofst* variable in the example below) and a step to select the partials (by means of the *inpincr* variable in the example below). Default values for these arguments are 0 and 1 respectively. Finally, the user may define a final optional argument that references a function table that will be used to rescale the amplitude values during the resynthesis. The amplitude values of all the partials along all the frames are rescaled to the table length and used as indexes to lookup a scaling amplitude value in the table. For example, in a table of size 1024, the scaling amplitude of all the 0.5 amplitude values (-6 dBFS) that are found in the ATS file is in the position 512 (1024/2). Very complex filtering effects can be obtained by carefully setting these gating tables according to the amplitude values of a particular ATS analysis.

EXAMPLE 05K07_atsadd.csd

```
<CsoundSynthesizer>
<CsOptions>
-o dac
</CsOptions>
<CsInstruments>

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

;Some macros
#define ATS_NP # 3 #      ;number of Partial
#define ATS_DU # 7 #      ;duration

instr 1
```

```

/*read some ATS data from the file header*/
iatsfile = p11
i_number_of_partials    ATSinfo iatsfile,  $ATS_NP
i_duration              ATSinfo iatsfile,  $ATS_DU

iamp      =      p4          ;amplitude scaler
ifreqdev =      2^(p5/12)    ;frequency deviation (p5=semitones up or down)
itable   =      p6          ;audio table

/*here we deal with number of partials, offset and increment issues*/
inpars  = (p7 < 1 ? i_number_of_partials : p7) ;inpars can not be <=0
ipofst  =      (p8 < 0 ? 0 : p8)      ;partial offset can not be < 0
ipincr  =      (p9 < 1 ? 1 : p9)      ;partial increment can not be <= 0
imax     =      ipofst + inpars*ipincr  ;max. partials allowed

if imax <= i_number_of_partials igoto OK
;if we are here, something is wrong!
;set npars to zero, so as the output will be zero and the user knows
print imax, i_number_of_partials
inpars  = 0
ipofst  = 0
ipincr  = 1
OK: ;data is OK
/*********************************************
igatefn =      p10           ;amplitude scaling table

ktime  linseg 0, p3, i_duration
asig ATSadn $ats_file, ifreqdev, iatsfile, itable, inpars, ipofst, ipincr, igatefn

out    asig*iamp
endin

</CsInstruments>
<CsScore>

;change to put any ATS file you like
#define ats_file #"basoon-C4.ats"#

;audio table (sine)
f1      0      16384  10      1
;some tables to test amplitude gating
;f2 reduce progressively partials with amplitudes from 0.5 to 1
;and eliminate partials with amplitudes below 0.5 (-6dBFS)
f2      0      1024    7      0 512 0 512 1
;f3 boost partials with amplitudes from 0 to 0.125 (-12dBFS)
;and attenuate partials with amplitudes from 0.125 to 1 (-12dBFS to 0dBFS)
f3      0      1024    -5     8 128 8 896 .001

; start dur amp freq atable npars offset pincr gatefn atsfile
i1  0      2.82  1      0      1      0      0      1      0      $ats_file
i1  +      .      1      0      1      0      0      1      2      $ats_file
i1  +      .      .8     0      1      0      0      1      3      $ats_file

e
</CsScore>
</CsoundSynthesizer>
;example by Oscar Pablo Di Liscia

```

The `ATSadn` opcode synthesizes residual (“noise”) data from an ATS file using the method explained above. This opcode works in a similar fashion to `ATSad` except that frequency warping of the noise bands is not permitted and the maximum number of noise bands will always be 25 (the 25 critical bands, see Zwicker/Fastl, footnote 3). The optional arguments `offset` and `increment`

work in a similar fashion to that in ATSadd. The `ATSaddnz` opcode allows the synthesis of several combinations of noise bands, but individual amplitude scaling of them is not possible.

EXAMPLE 05K08_atsaddnz.cs

```

<CsoundSynthesizer>
<CsOptions>
-o dac
</CsOptions>
<CsInstruments>

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

;Some macros
#define NB      # 25 # ;number noise bands
#define ATS_DU  # 7 # ;duration

instr 1
/*read some ATS data from the file header*/
iatsfile = p8
i_duration ATSinfo iatsfile, $ATS_DU

iamp      =      p4          ;amplitude scaler

/*here we deal with number of partials, offset and increment issues*/
inb      =      (p5 < 1 ? $NB : p5)      ;inb can not be <=0
ibofst   =      (p6 < 0 ? 0 : p6)      ;band offset cannot be < 0
ibincr   =      (p7 < 1 ? 1 : p7)      ;band increment cannot be <= 0
imax     =      ibofst + inb*ibincr      ;max. bands allowed

if imax <= $NB igoto OK
;if we are here, something is wrong!
;set nb to zero, so as the output will be zero and the user knows
print imax, $NB
inb = 0
ibofst = 0
ibincr = 1
OK: ;data is OK
/*********************************************
ktme    linseg  0, p3, i_duration
asig    ATSaddnz ktime, iatsfile, inb, ibofst, ibincr

        out      asig*iamp
endin

</CsInstruments>
<CsScore>

;change to put any ATS file you like
#define ats_file #"female-speech.ats"#

;  start dur  amp nbands bands_offset bands_incr atsfile
i1  0    7.32  1    25    0            1      $ats_file ;all bands
i1  +    .    .    15    10           1      $ats_file ;from 10 to 25 step 1
i1  +    .    .    8     1            3      $ats_file ;from 1 to 24 step 3
i1  +    .    .    5     15           1      $ats_file ;from 15 to 20 step 1

e
</CsScore>
```

```
</CsoundSynthesizer>
;example by Oscar Pablo Di Liscia
```

The `ATSSinnoi` opcode synthesizes both deterministic and residual (“noise”) data from an ATS file. This opcode may be regarded as a combination of the two previous opcodes but with the allowance of individual amplitude scaling of the mixes of deterministic and residual parts. All the arguments of `ATSSinnoi` are the same as those for the two previous opcodes, except for the two k-rate variables `ksinlev` and `knoislev` that allow individual, and possibly time-changing, scaling of the deterministic and residual parts of the synthesis.

EXAMPLE 05K09_atssinnoi.csd

```
<CsoundSynthesizer>
<CsOptions>
-o dac
</CsOptions>
<CsInstruments>

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

;Some macros
#define ATS_NP # 3 # ;number of Partials
#define ATS_DU # 7 # ;duration

instr 1
iatsfile = p11
/*read some ATS data from the file header*/
i_number_of_partials    ATSinfo iatsfile, $ATS_NP
i_duration              ATSinfo iatsfile, $ATS_DU
print i_number_of_partials

iamp      =      p4          ;amplitude scaler
ifreqdev =      2^(p5/12)    ;frequency deviation (p5=semitones up or down)
isinlev   =      p6          ;deterministic part gain
inoislev  =      p7          ;residual part gain

/*here we deal with number of partials, offset and increment issues*/
inpars   =      (p8 < 1 ? i_number_of_partials : p8) ;inpars can not be <=0
ipofst   =      (p9 < 0 ? 0 : p9)           ;partial offset can not be < 0
ipincr   =      (p10 < 1 ? 1 : p10)         ;partial increment can not be <= 0
imax     =      ipofst + inpars*ipincr ;max. partials allowed

if imax <= i_number_of_partials igoto OK
;if we are here, something is wrong!
;set npars to zero, so as the output will be zero and the user knows
prints "wrong number of partials requested", imax, i_number_of_partials
inpars = 0
ipofst = 0
ipincr = 1
OK: ;data is OK
/********************************************/

ktime linseg 0, p3, i_duration
asig  ATSSinnoi ktime, isinlev, inoislev, ifreqdev, iatsfile,
      inpars, ipofst, ipincr

out    asig*iamp
```

```

    endin

  </CsInstruments>
  <CsScore>
;change to put any ATS file you like
#define ats_file #"female-speech.ats"#

; start  dur   amp   freqdev sinlev  noislev npars   offset  pincr  atsfile
i1  0     3.66 .79   0       1       0       0       0       1       $ats_file
;deterministic only
i1 + 3.66 .79   0     0       1       0       0       0       1       $ats_file
;residual only
i1 + 3.66 .79   0     1       1       0       0       0       1       $ats_file
;deterministic and residual
;  start dur   amp   freqdev sinlev  noislev npars   offset  pincr  atsfile
i1 + 3.66 2.5   0     1       0       80      60      1       $ats_file
;from partial 60 to partial 140, deterministic only
i1 + 3.66 2.5   0     0       1       80      60      1       $ats_file
;from partial 60 to partial 140, residual only
i1 + 3.66 2.5   0     1       1       80      60      1       $ats_file
;from partial 60 to partial 140, deterministic and residual
e
</CsScore>
</CsoundSynthesizer>
example by Oscar Pablo Di Liscia

```

`ATScross` is an opcode that performs some kind of “interpolation” of the amplitude data between two ATS analyses. One of these two ATS analyses must be obtained using the `ATScross` opcode (see above) and the other is to be loaded by an `ATScross` instance. Only the deterministic data of both analyses is used. The ATS file, time pointer, frequency scaling, number of partials, partial offset and partial increment arguments work the same way as usages in previously described opcodes. Using the arguments `kmylev` and `kbuflev` the user may define how much of the amplitude values of the file read by `ATScross` is to be used to scale the amplitude values corresponding to the frequency values of the analysis read by `ATScross`. So, a value of 0 for `kbuflev` and 1 for `kmylev` will retain the original ATS analysis read by `ATScross` unchanged whilst the converse (`kbuflev=1` and `kmylev=0`) will retain the frequency values of the `ATScross` analysis but scaled by the amplitude values of the `ATScross` analysis. As the time pointers of both units need not be the same, and frequency warping and number of partials may also be changed, very complex cross synthesis and sound hybridation can be obtained using this opcode.

EXAMPLE 05K10_atscross.csd

```

<CsoundSynthesizer>
<CsOptions>
-o dac
</CsOptions>
<CsInstruments>

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

;ATS files
#define ats1 #"flute-A5.ats"#
#define ats2 #"oboe-A5.ats"#

```

```

instr 1
iamp    = p4          ;general amplitude scaler
ilev1   = p5          ;level of iats1 partials
ifd1    = 2^(p6/12)    ;frequency deviation for iats1 partials
ilev2   = p7          ;level of ats2 partials
ifd2    = 2^(p8/12)    ;frequency deviation for iats2 partials
itaue   = p9          ;audio table

/*get ats file data*/
inp1  ATSinfo $ats1, 3
inp2  ATSinfo $ats2, 3
idur1 ATSinfo $ats1, 7
idur2 ATSinfo $ats2, 7

ktime  line    0, p3, idur1
ktime2 line    0, p3, idur2

        ATSBufread ktime, ifd1, $ats1, inp1
aout   ATSCross   ktime2, ifd2, $ats2, itau, ilev2, ilev1, inp2

        out      aout*iamp

endin

</CsInstruments>
<CsScore>

; sine wave for the audio table
f1      0      16384  10      1

; start dur amp lev1 f1  lev2 f2 table
i1 0    2.3 .75 0  0  1  0  1 ;original oboe
i1 +   .  . 0.25 .  .75  .  . ;oboe 75%, flute 25%
i1 +   .  . 0.5  . 0.5  .  . ;oboe 50%, flute 50%
i1 +   .  . .75  .  .25  .  . ;oboe 25%, flute 75%
i1 +   .  . 1  . 0  .  . ;oboe partials with flute's amplitudes

e
</CsScore>
</CsoundSynthesizer>
;example by Oscar Pablo Di Liscia

```

06 A. RECORD AND PLAY SOUNDFILES

Playing Soundfiles from Disk - *diskin*

The simplest way of playing a sound file from Csound is to use the `diskin` 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 (buffer) is preferable.¹

Sound File Name, Absolute or Relative Path

The first input argument for *diskin* denotes the file which is to be read. It is called *ifilcod* which stems from old Csound times where a sound file should be named "soundin.1" and could then be read by the opcode `soundin` as 1 for *ifilcod*. For now we usually give a string here as input, so *Sfilcod* would be a better name for the variable. The string can be either a *name* like "loop.wav", or it can contain the full *path* to the sound file like "/home/me/Desktop/loop.wav". Usually we will prefer to give a name instead of full path – not only because it is shorter, but also because it makes our csd more portable. Csound will recognize a sound file by its *name* in these cases:

1. The *csd* file and the sound file are in the same directory (folder). This is the most simple way and gives full flexibility to run the same *csd* from any other computer, just by copying the whole folder.
2. The folder which contains the sound file is known to Csound. This can be done with the option `-env:SSDIR+=/path/to/sound/folder`. Csound will then add this folder to the *Sound Sample Directory* (SSDIR) in which it will look for sound samples.

A path to look for sound files can not only be given as *absolute* path but also as *relative* path. Let us assume we have this structure for the *csd* file and the *sound* file:

¹As this is a matter of speed, it depends both, on the complexity of the csound file(s) you are running, and the speed of the hard disk. A Solid State Disk is much faster than a traditional HDD, so a Csound file with a lot of *diskin* processes may run fine on a SSD which did not run on a HDD.

```
| -home
| -me
  superloop.csd
  |-Desktop
    loop.wav
```

The *superloop.csd* Csound file is not in the same directory as the *loop.wav* sound file. But relative to the *csd* file, the sound is in *Desktop*, and *Desktop* is indeed in the same folder as the *superloop.csd* file. So we could write this:

```
aSound diskin "Desktop/loop.wav"
```

Or we could use this in the *CsOptions* tag:

```
--env:SSDIR+=Desktop
```

And then again just give the raw name to *diskin*:

```
aSound diskin "loop.wav"
```

This is another example for a possible file structure:

```
| -home
| -me
  |-samples
    loop.wav
  |-Desktop
    superloop.csd
```

Now the *loop.wav* is relative to the *csd* file not in a subfolder, but on the higher level folder called *me*, and then in the folder *samples*. So we have to specify the relative path like this: “Go up, then look into the folder *samples*.” *Going up* is specified as two dots, so this would be relative path for *diskin*:

```
aSound diskin "../samples/loop.wav"
```

Again, we could alternatively use *-env:SSDIR+=..../samples* in the *CsOptions* and then simple refer to “*loop.wav*”.

Diskins Output Arguments: Single or Array

In the [Csound Manual](#) we see two different options for outputs, left hand side of the *diskin* opcode:

```
ar1 [, ar2 [, ar3 [, ... arN]]] diskin      ...
ar1[]                      diskin      ...
```

The first line is the traditional way. We will output here as many audio signals as the sound file has channels. Many Csound user will have read this message:

```
INIT ERROR in instr 1 line 17: diskin2:
number of output args inconsistent with number of file channels
```

This *inconsistency of the number of output arguments* and the *number of file channels* happens, if we use the stereo file “*magic.wav*” but write:

```
aSample diskin "magic.wav"
```

Or vice versa, we use the *mono* file "nice.wav" but write:

```
aLeft, aRight diskin "nice.wav"
```

Since Csound6, however, we have the second option mentioned on Csound's manual page for *diskin*:

```
ar1[] diskin ...
```

If the output variable name is followed by square brackets, *diskin* will write its output in an audio array.² The size (*length*) of this array mirrors the number of channels in the audio file: 1 for a mono file, 2 for a stereo file, 4 for a quadro file, etc.

This is a very convenient method to avoid the mismatch error between output arguments and file channels. In the example below we will use this method. We write the audio in an array and will only use the first element for the output. So this will work with any number of channels for the input file.

Speed, Skiptime and Loop

After the mandatory file name or path string, we can pass some optional input arguments:

- *kpitch* specifies the speed of reading the sound file. The *default* is 1 here, which means normal speed. 2 would result in double speed (octave higher and half time to read through the sound file), 0.5 would result in half speed (octave lower and twice as much time needed for reading). Negative values read backwards. As this is a *k-rate* parameter, it offers a lot of possibilities for modification already.
- *iskiptim* specifies the point in the sound file where reading starts. The default is 0 (= from the beginning); 2 would mean to skip the first two seconds of the sound file.
- *iwraparound* answers the question what *diskin* will do when reading reaches the end of the file. The default is here 0 which means that reading stops. If we put 1 here, *diskin* will loop the sound file.

EXAMPLE 06A01_Play_soundfile.csd

```
<CsoundSynthesizer>
<CsOptions>
-odac --env:SSDIR+=../SourceMaterials
</CsOptions>
<CsInstruments>
sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

;file should be found in the 'SourceMaterials' folder
gS_file = "fox.wav"
```

²Chapter 03 E gives more explanations about arrays in Csound.

```

instr Defaults
kSpeed = p4 ; playback speed
iSkip = p5 ; inskip into file (in seconds)
iLoop = p6 ; looping switch (0=off 1=on)
aRead[] diskin gS_file, kSpeed, iSkip, iLoop
out aRead[0], aRead[0] ;output first channel twice
endin

instr Scratch
kSpeed randomi -1, 1.5, 5, 3
aRead[] diskin gS_file, kSpeed, 1, 1
out aRead[0], aRead[0]
endin
</CsInstruments>
<CsScore>
;      dur speed skip loop
i 1 0 4 1    0 0      ;default values
i . 4 3 1    1.7 0    ;skiptime
i . 7 6 0.5   0 0     ;speed
i . 13 6 1    0 1     ;loop
i 2 20 20
</CsScore>
</CsoundSynthesizer>
;example written by Iain McCurdy and joachim heintz

```

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* and will be written to this file. The number of channels that the file will contain 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_Write_soundfile.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>
sr      = 44100
ksmps  = 32
nchnls = 1
0dbfs  = 1

instr 1 ; a simple tone generator
aEnv   expon  0.2, p3, 0.001          ; a percussive envelope
aSig    oscil   aEnv, cpsmidinn(p4)    ; 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
</CsScore>
</CsoundSynthesizer>
; example written by Iain McCurdy

```

Both Audio to Disk and RTAudio Output - *fout* with *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 read 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_Write_RT.csd

```

<CsoundSynthesizer>
<CsOptions>
-odac ; activate real-time audio output
</CsOptions>
<CsInstruments>
sr      =      44100
ksmps   =      32
nchnls  =      1
0dbfs   =      1

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)     ; 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
</CsScore>
</CsoundSynthesizer>
;example written by Iain McCurdy
```

06 B. 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*. The opcode 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. As always, you can replace the sound file with one of your own. Note that *GEN01* accepts mono or stereo files; the number of output arguments for *flooper2* must correspond with the mono or stereo table.

EXAMPLE 06B01_flooper2.csd

```
<CsoundSynthesizer>
<CsOptions>
-odac ; activate real-time audio
</CsOptions>
<CsInstruments>
sr      =      44100
ksmps   =      32
nchnls  =      2
0dbfs   =      1

; STORE AUDIO IN RAM USING GEN01 FUNCTION TABLE
giSoundFile  ftgen  0, 0, 0, 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, 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>
; example written by Iain McCurdy

```

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_sndloop.csd

```

<CsoundSynthesizer>
<CsOptions>
; real-time audio in and out are both activated
-iadc -odac
</CsOptions>
<CsInstruments>
sr      =      44100
ksmps   =      32
nchnls  =      2
0dbfs   =      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 = 45 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, aOut      ; send audio to output
endin

</CsInstruments>
<CsScore>
i 1 0 3600 ; instr 1 plays for 1 hour
</CsScore>
</CsoundSynthesizer>
;example written by Iain McCurdy

```

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 to the next 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 in seconds is multiplied by the sample rate to calculate the proper table size.

EXAMPLE 06B03_RecPlayToTable.csd

```

<CsoundSynthesizer>
<CsOptions>
; real-time audio in and out are both activated
-iadc -odac -m128
</CsOptions>
<CsInstruments>
sr      =      44100
ksmps   =      32
nchnls  =      2
0dbfs   =      1

```

```

giTabLenSec = 3 ;table duration in seconds
giBuffer ftgen 0, 0, giTabLenSec*sr, 2, 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"
    kKey sensekey ; sense activity on the computer keyboard
    if kKey==114 then ; if ASCII value of 114 ('r') is output
    event "i", 2, 0, giTabLenSec ; activate recording instrument (2)
    endif
    if kKey==112 then ; if ASCII value of 112 ('p') is output
    event "i", 3, 0, giTabLenSec ; activate playback instrument
    endif
    endin

    instr 2 ; record to buffer
    ; -- 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,ftlen(giBuffer); create an index for writing to table
tablew   ain,ndx,giBuffer ; write audio to function table
endin

    instr 3 ; playback from buffer
    ; -- 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, ftlen(giBuffer) ;create an index for reading from table
aRead    table   aNdx, giBuffer ; read audio to audio storage table
        out     aRead, aRead ; send audio to output
    endin

</CsInstruments>
<CsScore>
i 1 0 3600 ; Sense keyboard activity. Start recording - playback.
</CsScore>
</CsoundSynthesizer>
;example written by Iain McCurdy

```

Encapsulating Record and Play Buffer Functionality to a UDO

Recording and playing back of buffers can also be encapsulated into a User Defined Opcode (UDO).¹ We will show here a version which in a way *re-invents the wheel* as it creates an own sample-by-sample increment for reading and writing the buffer rather than using a pointer. This

¹See Chapter 03 G for more information about writing UDOs in Csound.

is mostly meant as example how open this field is for different user implementations, and how easy it is to create own applications based on the fundamental functionalities of table reading and writing.

One way to write compact Csound code is to follow the principle *one job per line* (of code). For defining *one job* of a good size, we will mostly need a UDO which combines some low-level tasks and also allows us to apply a memorizable name for this job. So often the principle *one job per line* results in *one UDO per line*.

The *jobs* in the previous example can be described as follows:

1. Create a buffer of a certain length.
2. Watch keyboard input.
3. Record input channel 1 to table if 'r' key is pressed.
4. Play back table if 'p' key is pressed and output.

Let us go step by step through this list, before we finally write this instrument in four lines of code. Step **1** we already did in the previous example; we only wrap the GEN routine in a UDO which gets the time as input and returns the buffer variable as output. Anything else is hidden.

```
opcode createBuffer, i, i
  ilen xin
  ift ftgen 0, 0, ilen*sr, 2, 0
  xout ift
endop
```

Step **2** is the only one which is a normal Csound code line, consisting of the `sensekey` opcode. Due to the implementation of `sensekey`, there should only be one `sensekey` in a Csound orchestra.

```
kKey, kDown sensekey
```

Step **3** consists of two parts. We will write one UDO for both. The first UDO writes to a buffer if it gets a signal to do so. We choose here a very low-level way of writing an audio signal to a buffer. Instead of creating an index, we just increment the single index numbers. To continue the process at the end of the buffer, we apply the *modulo* operation to the incremented numbers.²

```
opcode recordBuffer, 0, aik
  ain, ift, krec xin
  setksmps 1 ;k=a here in this UDO
  kndx init 0 ;initialize index
  if krec == 1 then
    tablew ain, a(kndx), ift
    kndx = (kndx+1) % ftlen(ift)
  endif
endop
```

The second UDO outputs 1 as long as a key is pressed. Its input consists of the ASCII key which is selected, and of the output of the `sensekey` opcode.

```
opcode keyPressed, k, kki
  kKey, kDown, iAscii xin
  kPrev init 0 ;previous key value
  kOut = (kKey == iAscii || (kKey == -1 && kPrev == iAscii) ? 1 : 0)
  kPrev = (kKey > 0 ? kKey : kPrev)
  kPrev = (kPrev == kKey && kDown == 0 ? 0 : kPrev)
  xout kOut
endop
```

²The symbol for the *modulo operation* is %. The result is the *remainder* in a division: $1 \% 3 = 1$, $4 \% 3 = 1$, $7 \% 3 = 1$ etc.

The reading procedure in step 4 is in fact the same as was used for writing. We only have to replace the opcode for writing *tablew* with the opcode for reading *table*.

```
opcode playBuffer, a, ik
ift, kplay xin
setksmps 1 ;k=a here in this UDO
kndx init 0 ;initialize index
if kplay == 1 then
  aRead table a(kndx), ift
  kndx = (kndx+1) % ftlen(ift)
endif
xout aRead
endop
```

Note that you must disable the key repeats on your computer keyboard for the following example (in CsoundQt, disable “Allow key repeats” in *Configuration -> General*). Press the *r* key as long as you want to record, and the *p* key for playing back. Both, record and playback, is done circular.

EXAMPLE 06B04_BufRecPlay_UDO.csd

```
<CsoundSynthesizer>
<CsOptions>
-i adc -o dac -m128
</CsOptions>
<CsInstruments>
sr = 44100
ksmps = 32
nchnls = 1
0dbfs = 1

***** UDO definitions *****/
opcode createBuffer, i, i
ilen xin
ift ftgen 0, 0, ilen*sr, 2, 0
xout ift
endop
opcode recordBuffer, 0, aik
ain, ift, krec xin
setksmps 1 ;k=a here in this UDO
kndx init 0 ;initialize index
if krec == 1 then
  tablew ain, a(kndx), ift
  kndx = (kndx+1) % ftlen(ift)
endif
endop
opcode keyPressed, k, kki
kKey, kDown, iAscii xin
kPrev init 0 ;previous key value
kOut = (kKey == iAscii || (kKey == -1 && kPrev == iAscii) ? 1 : 0)
kPrev = (kKey > 0 ? kKey : kPrev)
kPrev = (kPrev == kKey && kDown == 0 ? 0 : kPrev)
xout kOut
endop
opcode playBuffer, a, ik
ift, kplay xin
setksmps 1 ;k=a here in this UDO
kndx init 0 ;initialize index
if kplay == 1 then
  aRead table a(kndx), ift
```

```

kndx = (kndx+1) % ftlen(ift)
endif
endop

instr RecPlay
iBuffer = createBuffer(3) ;buffer for 3 seconds of recording
kKey, kDown sensekey
recordBuffer(inch(1), iBuffer, keyPressed(kKey,kDown,114))
out playBuffer(iBuffer, keyPressed(kKey,kDown,112))
endin

</CsInstruments>
<CsScore>
i 1 0 1000
</CsScore>
</CsoundSynthesizer>
;example written by joachim heintz

```

We use mostly the functional style of writing Csound code here. Instead of

```
iBuffer = createBuffer(3)
```

we could also write:

```
iBuffer createBuffer 3
```

To plug the audio signal from channel 1 directly into the *recordBuffer* UDO, we plug the *inch(1)* directly into the first input. Similar the output of the *keyPressed* UDO as third input. For more information about functional style coding, see chapter 03 I.

Further Opcodes for Investigation

Csound contains a wide range of opcodes that offer a variety of *ready-made* methods of playing back audio held in a function table. The oldest group of these opcodes are *loscil* and *loscil3*. Despite their age they offer some unique features such as the ability implement both sustain and release stage looping (in a variety of looping modes), their ability to read from stereo as well as mono function tables and their ability to read looping and base frequency data from the sound file stored in the function table. *loscil* and *loscil3* were originally intended as the kernel mechanism for building a sampler.

For reading multichannel files of more than two channels, the more recent *loscilmx* exists as an excellent option. It can also be used for mono or stereo, and it can – similar to *diskin* – write its output in an audio array.

loscil and *loscil3* will only allow looping points to be defined at i-time. *Iposcil*, *Iposcil3*, *Iposcila*, *Iposcilsa* and *Iposcilsa2* will allow looping points to be changed a k-rate, while the note is playing.

It is worth not forgetting Csound's more exotic methods of playback of sample stored in function tables. *mincer* and *temposcals* use streaming vocoder techniques to facilitate independent pitch and time-stretch control during playback (this area is covered more fully in chapter 05 I. *sndwarp* and *sndwarpst* similarly facilitate independent pitch and playback speed control but through the technique of granular synthesis this area is covered in detail in chapter 05 G).

07 A. RECEIVING EVENTS BY MIDIIN

Csound provides a variety of opcodes, such as `cpsmidi`, `ampmidi` and `ctrl7`, which facilitate the reading of incoming midi data into Csound with minimal fuss. These opcodes allow us to read in midi information without us having to worry about parsing status bytes and so on. Occasionally though when more complex midi interaction is required, 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 to use) by writing 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 by using the `-M flag`. You will also need to make sure that you have a midi keyboard or controller connected. You may also want to include the `-m128 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_midiin_print.csd

```
<CsoundSynthesizer>
<CsOptions>
-Ma -m128
; activates all midi devices, suppress note printings
</CsOptions>
<CsInstruments>
; 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 change
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>
;example by Iain McCurdy
```

The principle advantage of using the *midiin* opcode is that, unlike opcodes such as *cpsmidi*, *amp-midi* and *ctrl7* which only receive specific midi data types on a specific channel, *midiin* “listens” to all incoming data including system exclusive messages. In situations where elaborate Csound instrument triggering mappings that are beyond the capabilities of the default triggering mechanism are required, then the use of *midiin* might be beneficial.

07 B. 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 command line flags for midi input have been set (see [configuring midi](#) for further information), 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 transmitting. 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 corresponding instruments do not exist, notes on any of these channels will be directed to instrument 1.

EXAMPLE 07B01_MidiInstrTrigger.csd

```
<CsoundSynthesizer>
<CsOptions>
-Ma -odac -m128
;activates all midi devices, real time sound output, suppress note printings
</CsOptions>
<CsInstruments>
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:
    timeout 0, 1, impulse
    reinit reset
impulse:
aenv expon    1, 0.3, 0.0001
aSig oscil    aenv, 500, gisine
    out      aSig
    endin

instr 2 ; 2 impulses (midi channel 2)
prints "instrument/midi channel: %d%n",p1
reset:
    timeout 0, 1, impulse
    reinit reset
impulse:
aenv expon    1, 0.3, 0.0001
aSig oscil    aenv, 500, gisine
a2 delay     aSig, 0.15           ; short delay adds another impulse
    out      aSig+a2
    endin

instr 3 ; 3 impulses (midi channel 3)
prints "instrument/midi channel: %d%n",p1
reset:
    timeout 0, 1, impulse
    reinit reset
impulse:
aenv expon    1, 0.3, 0.0001
aSig oscil    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
    endin

</CsInstruments>
<CsScore>
f 0 300
</CsScore>
<CsoundSynthesizer>
;example by Iain McCurdy

```

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 defines 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 jumbles 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_massign.csd

```

<CsoundSynthesizer>
<CsOptions>
-Ma -odac -m128
; activate all midi devices, real time sound output, suppress note printing
</CsOptions>
<CsInstruments>
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 and midi channel
reset:                                         ; label 'reset'
    timeout 0, 1, impulse                      ; jump to 'impulse' for 1 second
    reinit reset                               ; reinitialize 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:
    timeout 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:
    timeout 0, 1, impulse
    reinit reset
impulse:
aenv expon      1, 0.3, 0.0001
aSig poscil    aenv, 500, gisine
a2 delay       aSig, 0.15
a3 delay       a2, 0.15                       ; delay generates a 2nd impulse
        out      aSig+a2+a3                  ; delay generates a 3rd impulse
    endin

</CsInstruments>

<CsScore>
f 0 300
</CsScore>

```

```
<CsoundSynthesizer>
;example by Iain McCurdy
```

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 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. Below is an example of using fractional instrument numbers.

EXAMPLE 07B03_MidiTriggerChain.csd

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

```

        printk "instr %f playing!\n", 1, p1
    endin

</CsInstruments>
<CsScore>
</CsScore>
</CsoundSynthesizer>
;Example by Joachim Heintz, using code of Victor Lazzarini

```

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 for 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 for any key below C3 instrument 2 will be called, for any key between C3 and B4 instrument 3, and for any higher key instrument 4.

Using this multiple triggering you are also able to trigger more than one instrument at the same time (which is not possible using the *massign* opcode). Here is an example using a User Defined Opcode (see the [UDO chapter](#) of this manual):

EXAMPLE 07B04_MidiMultiTrigg.csd

```

<CsoundSynthesizer>
<CsOptions>
-Ma
</CsOptions>
<CsInstruments>
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 triggered
ifn, 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>
</CsScore>
</CsoundSynthesizer>
;Example by Joachim Heintz, using code of Victor Lazzarini

```

07 C. 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 of rescaling 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 controller's output response to something possibly 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_ctrl7_print.csd

```
<CsoundSynthesizer>
<CsOptions>
-Ma -odac
; activate all MIDI devices
</CsOptions>
<CsInstruments>
; '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
</CsScore>
```

```
<CsoundSynthesizer>
;example by Iain McCurdy
```

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 by using the opcodes `pchbend` and `aftouch`. Once again we can specify minimum and maximum values with which to rescale 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 by 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 they change. 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, i.e. you will need to play a midi note on your keyboard and then move the pitch bend wheel.

EXAMPLE 07C02_pchbend_aftouch.csd

```
<CsoundSynthesizer>
<CsOptions>
-odac -Ma
</CsOptions>
<CsInstruments>
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>
</CsScore>
<CsoundSynthesizer>
;example by Iain McCurdy

```

Initialising MIDI Controllers

It may be useful to be able to define the initial value of a midi controller, that is, the value any *ctrl7*s will adopt until their corresponding hardware controls have been moved. Midi hardware controls only send messages when they change so until this happens their values in Csound defaults to their minimum settings unless additional initialisation has been carried out. As an example, if we imagine we have a 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 utilised. It would be more useful to be able to define the starting value for each of these controllers. The *initc7* opcode allows us to do this. If *initc7* is placed within the instrument itself it will be reinitialised each time the instrument is called, if it is placed in instrument 0 (just after the header statements) then it will only be initialised when the orchestra is first launched. The latter case is probably most useful.

In the following example a simple synthesizer is created. 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 will set this controller to its maximum value (midi value 127), and a value of 0.5 will set it to its halfway value (midi value 64), and so on.

Additionally this example uses the *cpsmidi* opcode to scan midi pitch (basically converting midi note numbers to cycles-per-second) and the *ampmidi* opcode to scan and rescale key velocity.

EXAMPLE 07C03_cpsmidi_ampmidi.csd

```

<CsoundSynthesizer>
<CsOptions>
-Ma -odac
; activate all midi inputs and real-time audio output
</CsOptions>
<CsInstruments>
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 key velocity. Rescale to be from 0 to 1
kVol ctrl7 1,1,0,1      ; read in CC 1, chan 1. Rescale to be from 0 to 1
aSig oscil iAmp*kVol, iCps, giSine ; an audio oscillator
        out   aSig       ; send audio to output
        endin

</CsInstruments>
<CsScore>
</CsScore>
<CsoundSynthesizer>
;example by Iain McCurdy

```

You will maybe hear that this instrument produces *clicks* as notes begin and end. To find out how to prevent this see the section on envelopes with release sensing in chapter [05 A.](#)

Smoothing 7-bit Quantisation 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 2^7 possible values; therefore 128 possible values, which is rather inadequate for defining, for example, the frequency of an oscillator over a number of octaves, the cutoff frequency of a filter or a quickly moving volume control. We soon become aware of the parameter that is being changed moving in steps - so not really a *continuous* controller. We may also experience clicking artefacts, sometimes called *zipper noise*, as the value changes. The extent of this will depend upon the parameter being controlled. 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 at k-rate 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 ensures that when a new note begins the volume control immediately jumps to its required value rather than gliding up from zero as would otherwise be affected by 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_smoothing.csd

```

<CsoundSynthesizer>
<CsOptions>
-Ma -odac
</CsOptions>
<CsInstruments>
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 .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>
</CsScore>
<CsoundSynthesizer>
;example by Iain McCurdy

```

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_MidiControlComplex.csd

```

<CsoundSynthesizer>
<CsOptions>
-Ma -odac
</CsOptions>
<CsInstruments>
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, chn 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>
</CsScore>
<CsoundSynthesizer>
;example by Iain McCurdy

```

RECORDING CONTROLLER DATA

Data performed on a controller or controllers can be recorded into GEN tables or arrays so that a real-time interaction with a Csound instrument can be replayed at a later time. This can be preferable to recording the audio output, as this will allow the controller data to be modified. The simplest approach is to simply store each controller value every k-cycle into sequential locations in a function table but this is rather wasteful as controllers will frequently remain unchanged from k-cycle to k-cycle.

A more efficient approach is to store values only when they change and to time stamp those events so that they can be replayed later on in the right order and at the right speed. In this case data will be written to a function table in pairs: time-stamp followed by a value for each new event (event refers to when a controller changes). This method does not store durations of each event, merely when they happen, therefore it will not record how long the final event lasts until recording stopped. This may or may not be critical depending on how the recorded controller data is used later on but in order to get around this, the following example stores the duration of the complete recording at index location 0 so that we can derive the duration of the last event. Additionally the first event stored at index location 1 is simply a value: the initial value of the controller (the time stamp for this would always be zero anyway). Thereafter events are stored as time-stamped pairs of data: index 2=time stamp, index 3=associated value and so on.

To use the following example, activate *Record*, move the slider around and then deactivate *Record*. This gesture can now be replayed using the *Play* button. As well as moving the GUI slider, a tone is produced, the pitch of which is controlled by the slider.

Recorded data in the GEN table can also be backed up onto the hard drive using *ftsave* and recalled in a later session using *ftload*. Note that *ftsave* also has the capability of storing multiple function tables in a single file.

EXAMPLE 07C06_RecordController.csd

```

<CsoundSynthesizer>
<CsOptions>
-odac -dm0

```

```

</CsOptions>
<CsInstruments>
sr      = 44100
ksmps   = 8
nchnls = 1
0dbfs   = 1

FLpanel "Record Gesture",500,90,0,0
gkRecord,gihRecord FLbutton "Rec/Stop",1,0,22,100,25, 5, 5,-1
gkPlay,gihPlay   FLbutton "Play",    1,0,22,100,25,110, 5,-1
gksave,ihsave   FLbutton "Save to HD", 1,0,21,100,25,290,5,0,4,0,0
gkload,ihload   FLbutton "Load from HD", 1,0,21,100,25,395,5,0,5,0,0
gkval, gihval   FLslider "Control", 0,1, 0,23, -1,490,25, 5,35
FLpanel_end
FLrun

gidata ftgen 1,0,1048576,-2,0 ; Table for controller data.

opcode RecordController,0,Ki
  kval,ifn      xin
  i     ftgen 1,0,ftlen(ifn),-2,0           ; erase table
  tableiw i(kval),1,ifn          ; write initial value at index 1.
  ;(Index 0 will be used be storing the complete gesture duration.)
  kndx  init  2        ; Initialise index
  kTime  timeinsts       ; time since this instrument started in seconds
; Write a data event only when the input value changes
if changed(kval)==1 && kndx<=(ftlen(ifn)-2) && kTime>0 then
; Write timestamp to table location defined by current index.
  tablew kTime, kndx, ifn
; Write slider value to table location defined by current index.
  tablew kval, kndx + 1, ifn
; Increment index 2 steps (one for time, one for value).
  kndx  =      kndx + 2
endif
; sense note release
  krel  release
; if we are in the final k-cycle before the note ends
  if(krel==1) then
; write total gesture duration into the table at index 0
  tablew kTime,0,ifn
  endif
endop

opcode PlaybackController,k,i
  ifn      xin
; read first value
; initial controller value read from index 1
  ival  table  1,ifn
; initial value for k-rate output
  kval  init  ival
; Initialise index to first non-zero timestamp
  kndx  init  2
; time in seconds since this note started
  kTime  timeinsts
; first non-zero timestamp
  iTimeStamp  tablei 2,ifn
; initialise k-variable for first non-zero timestamp
  kTimeStamp  init  iTimeStamp
; if we have reached the timestamp value...
  if kTime>=kTimeStamp && kTimeStamp>0 then
; ...Read value from table defined by current index.
  kval  table  kndx+1,ifn
  kTimeStamp  table  kndx+2,ifn           ; Read next timestamp

```

```

; Increment index. (Always 2 steps: timestamp and value.)
kndx    limit    kndx+2, 0, ftlen(ifn)-2
endif
        xout    kval
endop

; cleaner way to start instruments than using FLbutton built-in mechanism
instr  1
; trigger when button value goes from off to on
kOnTrig trigger gkRecord,0.5,0
; start instrument with a held note when trigger received
schedkwhen      kOnTrig,0,0,2,0,-1
; trigger when button value goes from off to on
kOnTrig trigger gkPlay,0.5,0
; start instrument with a held note when trigger received
schedkwhen      kOnTrig,0,0,3,0,-1
endin

instr  2      ; Record gesture
if gkRecord==0 then           ; If record button is deactivated...
  turnoff                  ; ...turn this instrument off.
endif
; call UDO
  RecordController      gkval,gidata
; Generate a sound.
kporttime    linseg  0,0.001,0.02
kval        portk   gkval,kporttime
asig        oscil   0.2,cpsoct((kval*2)+7)
out         asig
endin

instr  3      ; Playback recorded gesture
if gkPlay==0 then           ; if play button is deactivated...
  turnoff                  ; ...turn this instrument off.
endif
kval      PlaybackController  gidata
; send initial value to controller
  FLsetVal_i   i(kval),gihval
; Send values to slider when needed.
  FLsetVal     changed(kval),kval,gihval
; Generate a sound.
kporttime    linseg  0,0.001,0.02
kval        portk   gkval,kporttime
asig        oscil   0.2,cpsoct((kval*2)+7)
out         asig
; stop note when end of table reached
kTime     timeinsts       ; time in seconds since this note began
; read complete gesture duration from index zero
iRecTime    tablei  0,gidata
; if we have reach complete duration of gesture...
if kTime>=iRecTime then
; deactivate play button (which will in turn, turn off this note.)
  FLsetVal     1,0,gihPlay
endif
endin

instr  4      ; save table
  ftsave "ControllerData.txt", 0, gidata
endin

instr  5      ; load table
  ftload "ControllerData.txt", 0, gidata

```

```
endin

</CsInstruments>
<CsScore>
i 1 0 3600
</CsScore>
</CsoundSynthesizer>
;example by Iain McCurdy
```


07 D. READING MIDI FILES

Instead of using either the standard Csound score or live midi events as input for an 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 interpreted as desired using the techniques discussed in [Working with Controllers](#).

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_ReadMidiFile.csd

```
<CsoundSynthesizer>
<CsOptions>
;-F' flag reads in a midi file
-F AnyMIDIfile.mid
</CsOptions>
<CsInstruments>
sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

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
</CsScore>
<CsoundSynthesizer>
;Example by Iain McCurdy

```

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_MidiToScore.csd

```

<CsoundSynthesizer>
<CsOptions>
; enter name of input midi file
-F InputMidiFile.mid

```

```
</CsOptions>
<CsInstruments>

;ksmps needs to be small to ensure accurate rendering of timings
ksmps = 8

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 than duration of midi file
</CsScore>
</CsoundSynthesizer>
;example by Iain McCurdy
```

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.

07 E. 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 to desired 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`. It 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 restrain its output. 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*)¹ 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

¹See chapter 03 C for details.

instruction on the same midi channel and with the same note number before stopping the note. We must be mindful of this when working with *midfout*. 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_midiout.csd

```
<CsoundSynthesizer>
<CsOptions>
; amend device number accordingly
-Q999
</CsOptions>
<CsInstruments>
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>
;example by Iain McCurdy
```

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 utilising a release flag generated by the [release](#) opcode whenever a note ends and sending the note-off then.

EXAMPLE 07E02_score_to_midiout.csd

```

<CsoundSynthesizer>
<CsOptions>
; amend device number accordingly
-Q999
</CsOptions>
<CsInstruments>
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>
;example by Iain McCurdy

```

Obviously *midfout* 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 unnecessary midi data is not sent out, the output of the *line* function is first converted into integers, and *midfout* 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_midiout_cc.csd

```

<CsoundSynthesizer>
<CsOptions>
; amend device number accordingly
-Q999
</CsOptions>
<CsInstruments>
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>
;example by Iain McCurdy

```

midion - Outputting MIDI Notes Made Easier

`midout` 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_midion.csd

```

<CsoundSynthesizer>
<CsOptions>
; amend device number accordingly
-Q999
</CsOptions>
<CsInstruments>

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>
;example by Iain McCurdy

```

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_midion2.csd

```

<CsoundSynthesizer>
<CsOptions>
; amend device number accordingly
-Q999
</CsOptions>
<CsInstruments>

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>
;example by Iain McCurdy

```

`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_moscil.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>
;example by Iain McCurdy

```

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 A. OPEN SOUND CONTROL

Open Sound Control (OSC) offers a flexible and dynamic alternative to MIDI. It uses modern network communications, usually based on the user datagram transport layer protocol (UDP), and allows not only the communication between synthesizers but also between applications and remote computers.

Data Types and Csound Signifiers

The basic unit of OSC data is a *message*. This is being sent to an *address* which follows the UNIX path convention, starting with a slash and creating branches at every following slash. The names inside this structure are free, but the convention is that it should fit to the content, for instance /filter/rudi/cutoff or /Processing/sketch/RGB. So, in contrast to MIDI, the address space is not predefined and can be changed dynamically.

The OSC message must specify the type(s) of its argument(s). This is a list of all types which are available in Csound, and the signifier which Csound uses for this type:

Data Type	Csound Signifier
audio	a
character	c
double	d
float	f
long integer 64-bit	h
integer 32-bit	i
string	s
array (scalar)	A
table	G

Once data types are declared, messages can be sent and received. In OSC terminology, anything that sends a message is a client, and anything that receives a message is a server. Csound can be both. Usually it will communicate with another application either as client or as server. It can, for instance, receive data from [Processing](#), or it can send data to [Inscore](#).

Sending and Receiving Different Data Types

For this introduction we will keep both functions in Csound, one instrument will send an OSC message, another instrument will receive this message. We will start with sending and receiving nothing but one integer, to study the basic functionality.

Send/Receive an integer

EXAMPLE 08A01_OSC_send_recv_int.csd

```

<CsoundSynthesizer>
<CsOptions>
-m 128
</CsOptions>
<CsInstruments>

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

giPortHandle OSCinit 47120

instr Send
kSendTrigger = 1
kSendValue = 17
OSCsend kSendTrigger, "", 47120, "/exmp_1/int", "i", kSendValue
endin

instr Receive
kReceiveValue init 0
kGotIt OSClisten giPortHandle, "/exmp_1/int", "i", kReceiveValue
if kGotIt == 1 then
printf "Message Received for '%s' at time %f: kReceiveValue = %d\n",
      1, "/exmp_1/int", times:k(), kReceiveValue
endif
endin

</CsInstruments>
<CsScore>
i "Receive" 0 3 ;start listening process first
i "Send" 1 1    ;then after one second send message
</CsScore>
</CsoundSynthesizer>
;example by joachim heintz

```

To understand the main functionalities to use OSC in Csound, we will look more closely to what happens in the code.

OSCinit

```
giPortHandle OSCinit 47120
```

The `OSCinit` statement is necessary for the `OSClisten` opcode. It takes a port number as input argument and returns a handle, called `giHandle` in this case. This statement should usually be done in the global space.

OSCsend

```
kSendTrigger = 1
kSendValue = 17
OSCsend kSendTrigger, "", 47120, "/exmp_1/int", "i", kSendValue
```

The `OSCsend` opcode will send a message whenever the `kSendTrigger` will change its value. As this variable is set here to a fixed number, only one message will be sent. The second input for `OSCsend` is the host to which the message is being sent. An empty string means “localhost” or “127.0.0.1”. Third argument is the port number, here 47120, followed by the destination address string, here “/exmp_1/int”. As we are sending an integer here, the type specifier is “i” as fifth argument, followed by the value itself.

OSClisten

```
kReceiveValue init 0
kGotIt OSClisten giPortHandle, "/exmp_1/int", "i", kReceiveValue
```

On the receiver side, we find the `giPortHandle` which was returned by `OSCinit`, and the address string again, as well as the expected type, here “i” for integer. Note that the value which is received is on the *input* side of the opcode. So `kReceiveValue` must be initialized before, which is done in line 21. Whenever `OSClisten` receives a message, the `kGotIt` output variable will become 1 (otherwise it is zero).

```
if kGotIt == 1 then
  printf "Message Received for '/exmp_1/int' at time %f: \
  kReceiveValue = %d\n", 1, times:k(), kReceiveValue
endif
```

Here we catch this point, and get a printout with the time at which the message has been received. As our listening instrument starts first, and the sending instrument after one second, we will see a message like this one in the console:

```
Message Received for '/exmp_1/int' at time 1.002086: kReceiveValue = 17
```

Note that the time at which the message is received is necessarily slightly later than the time at which it is being sent. The time difference is usually around some milliseconds; it depends on the UDP transmission.

Send/Receive more than one data type in a message

The string which specifies the data types which are being sent, can consist of more than one character. It was “i” in the previous example, as we sent an integer. When we want to send a float and a string, it will become “fs”. This is the case in the next example; anything else is very similar to what was shown before.

EXAMPLE 08A02_OSC_more_data.csd

```

<CsoundSynthesizer>
<CsOptions>
-m 128
</CsOptions>
<CsInstruments>

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

giPortHandle OSCinit 47120

instr Send
kSendTrigger = 1
kFloat = 1.23456789
Sstring = "bla bla"
OSCsend kSendTrigger, "", 47120, "/exmp_2/more", "fs", kFloat, Sstring
endin

instr Receive
kReceiveFloat init 0
SReceiveString init ""
kGotIt OSClisten giPortHandle, "/exmp_2/more", "fs",
          kReceiveFloat, SReceiveString
if kGotIt == 1 then
  printf "kReceiveFloat = %f\nSReceiveString = '%s'\n",
         1, kReceiveFloat, SReceiveString
endif
endin

</CsInstruments>
<CsScore>
i "Receive" 0 3 ;start listening process first
i "Send" 1 1    ;then after one second send message
</CsScore>
</CsoundSynthesizer>
;example by joachim heintz

```

The printout is here:

```

kReceiveFloat = 1.234568
SReceiveString = 'bla bla'

```

Send/Receive arrays

Instead of single data, OSC can also send and receive collections of data. The next example shows how an array is being sent once a second, and is being transformed for each [metro](#) tick.

EXAMPLE 08A03_Send_receive_array.csd

```

<CsoundSynthesizer>
<CsOptions>
-m 128
</CsOptions>

```

```

<CsInstruments>

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

giPortHandle OSCinit 47120

instr Send
kSendTrigger init 0
kArray[] fillarray 1, 2, 3, 4, 5, 6, 7
if metro(1)==1 then
  kSendTrigger += 1
  kArray *= 2
endif
OSCsend kSendTrigger, "", 47120, "/exmp_3/array", "A", kArray
endin

instr Receive
kReceiveArray[] init 7
kGotIt OSClisten giPortHandle, "/exmp_3/array", "A", kReceiveArray
if kGotIt == 1 then
  printarray kReceiveArray
endif
endin

</CsInstruments>
<CsScore>
i "Receive" 0 3
i "Send" 0 3
</CsScore>
</CsoundSynthesizer>
;example by joachim heintz

```

Each time the metro ticks, the array values are multiplied by two. So the printout is:

```

2.0000 4.0000 6.0000 8.0000 10.0000 12.0000 14.0000
4.0000 8.0000 12.0000 16.0000 20.0000 24.0000 28.0000
8.0000 16.0000 24.0000 32.0000 40.0000 48.0000 56.0000

```

Send/Receive function tables

The next example shows a similar approach for function tables. Three different tables are being sent once a second, and received in the second instrument. Imagine two Csound instances running on two different computers for a more realistic situation.

EXAMPLE 08A04_Send_receive_table.csd

```

<CsoundSynthesizer>
<CsOptions>
-m 128
</CsOptions>
<CsInstruments>

sr  = 44100
ksmps = 32

```

```

nchnls  = 2
0dbfs   = 1

giPortHandle OSCinit 47120

giTable_1 ftgen 0, 0, 1024, 10, 1
giTable_2 ftgen 0, 0, 1024, 10, 1, 1, 1, 1, 1
giTable_3 ftgen 0, 0, 1024, 10, 1, .5, 3, .1

instr Send
kSendTrigger init 1
kTable init giTable_1
kTime init 0
OSCsend kSendTrigger, "", 47120, "/exmp_4/table", "G", kTable
if timeinsts() >= kTime+1 then
  kSendTrigger += 1
  kTable += 1
  kTime = timeinsts()
endif
endin

instr Receive
iReceiveTable ftgen 0, 0, 1024, 2, 0
kGotIt OSClisten giPortHandle, "/exmp_4/table", "G", iReceiveTable
aOut poscil .2, 400, iReceiveTable
out aOut, aOut
endin

</CsInstruments>
<CsScore>
i "Receive" 0 3
i "Send" 0 3
</CsScore>
</CsoundSynthesizer>
;example by joachim heintz

```

Send/Receive audio

It is also possible to send and receive an audio signal via OSC. In this case, a OSC message must be sent on each k-cycle. Remember though that OSC is not optimized for this task. Most probably you will hear some dropouts in the next example. (Larger ksmmps values should give better result.)

EXAMPLE 08A05_send_receive_audio.csd

```

<CsoundSynthesizer>
<CsOptions>
-m 128
</CsOptions>
<CsInstruments>

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

giPortHandle OSCinit 47120

```

```

instr Send
kSendTrigger init 1
aSend oscil .2, 400
OSCsend kSendTrigger, "", 47120, "/exmp_5/audio", "a", aSend
kSendTrigger += 1
endin

instr Receive
aReceive init 0
kGotIt OSClisten giPortHandle, "/exmp_5/audio", "a", aReceive
out aReceive, aReceive
endin

</CsInstruments>
<CsScore>
i "Receive" 1 3
i "Send" 0 5
</CsScore>
</CsoundSynthesizer>
;example by joachim heintz

```

Other OSC Opcodes

The examples in this chapter were simple demonstrations of how different data types can be sent and received via OSC. The real usage requires a different application as partner for Csound, instead of the soliloquy we performed here. It should be added that there are some OSC opcodes which extend the basic functionality of `OSCsend` and `OSClisten`:

- `OSCcount` returns the count of OSC messages currently unread.
- `OSCraw` listens for all messages at a given port.
- `OSCbundle` sends OSC messages in a bundle rather than single messages.
- There is a variant of `OSCsend` called `OSCsend_lo` which uses the liblo library.

Practical Examples with Processing

We will show here some examples for the communication between Csound and [Processing](#). Processing is a well-established programming language for any kind of image processing, including video recording and playback. The OSC library for Processing is called `oscP5`. After installing this library, it can be used for both, sending and receiving Open Sound Control messages in any way.

Csound to Processing: Video Playback

We often want to use visuals in connection with audio. A simple case is that we have a live electronic setup and at a certain point we want to start a video. We may want to start the video in Processing with a Csound message like this:

```
instr StartVideo
    OSCsend(1, "", 12000, "/launch2/start", "i", 1)
endin
```

This means that we send the integer 1 to the address “launch2/start” on port 12000.

To receive this message in Processing, we import the oscP5 library and create a new OscP5 instance which listens to port 12000:

```
import oscP5.*;
OscP5 oscP5;
oscP5 = new OscP5(this, 12000);
```

Then we use the pluck() method which passes the OSC messages of a certain address (here “/launch2/start”) to a method with a user-defined name. We call it “startVideo” here:

```
oscP5.plug(this, "startVideo", "/launch2/start");
```

All we have to do now is to wrap Processing’s video play() message in this startVideo method. This is the full example code, referring to the video “launch2.mp4” which can be found in the Processing examples:

```
//import the video and osc library
import processing.video.*;
import oscP5.*;
//create objects
OscP5 oscP5;
Movie movie;

void setup() {
    size(560, 406);
    background(0);
    // load the video
    movie = new Movie(this, "launch2.mp4");
    //receive OSC
    oscP5 = new OscP5(this, 12000);
    //pass the message to the startVideo method
    oscP5.plug(this, "startVideo", "/launch2/start");
}

//activate video playback when startVideo receives 1
void startVideo(int onOff) {
    if (onOff == 1) {
        movie.play();
    }
}

//callback function to read new frames
void movieEvent(Movie m) {
    m.read();
}

//show it
void draw() {
    image(movie, 0, 0, width, height);
}
```

To start the video by hitting any key we can write something like this on the Csound side:

```
instr ReceiveKey
    kKey, kPressed sensekey
    if kPressed==1 then
```

```

schedulek("StartVideo",0,1)
endif
endin
schedule("ReceiveKey",0,-1)

instr StartVideo
OSCsend(1,"",12000,"/launch2/start","i",1)
endin

```

Processing to Csound: Mouse Pressed

We start with a simple example for swapped roles: Processing is now the sender of the OSC message, and Csound the receiver.

For processing, sending OSC is recommended by using this method:

```
oscP5.send(OscMessage, myRemoteLocation)
```

where `myRemoteLocation` is a `NetAddress` which is created by the library `netP5`. This is the code for sending an integer count whenever the mouse is pressed via OSC. The message is sent on port 12002 to the address “/P5/pressed”.

```

//import oscP5 and netP5 libraries
import oscP5.*;
import netP5.*;
//initialize objects
OscP5 oscP5;
NetAddress myRemoteLocation;

int count;

void setup(){
  size(600, 400);
  //create OSC object, listening at port 12001
  oscP5 = new OscP5(this,12001);
  //create NetAddress for sending: localhost at port 12002
  myRemoteLocation = new NetAddress("127.0.0.1",12002);
  //initialize variable for count
  count = 0;
}

void mousePressed(){
  //create new OSC message when mouse is pressed
  OscMessage pressed = new OscMessage("/P5/pressed");
  //add count to the message
  count += 1;
  pressed.add(count);
  //send it
  oscP5.send(pressed, myRemoteLocation);
}

void draw(){}

```

The following Csound code receives the messages and prints the count numbers:

EXAMPLE 08A06_receive_mouse_pressed.csd

```

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

sr = 44100
ksmps = 64
nchnls = 2
0dbfs = 1

instr ReceiveOSC
iPort = OSCinit(12002)
kAns, kMess[] OSClisten iPort, "/P5/pressed", "i"
printf "Mouse in Processing pressed %d times!\n", kAns, kMess[0]
endin
schedule("ReceiveOSC",0,-1)

</CsInstruments>
<CsScore>
</CsScore>
</CsoundSynthesizer>
;example by joachim heintz

```

Processing to Csound: Moving Lines

For showing one simple example for the many possibilities to connect Processing's interactive visuals with Csound, we will use the *Distance 1D* example as basic. It shows two thin ans two thick lines which move on the screen in a speed which depends on the mouse position. We slightly modify the speed so that the four lines have four different speeds. Rather than ...

```

xpos1 += mx/16;
xpos2 += mx/64;
xpos3 -= mx/16;
xpos4 -= mx/64;

```

... we write:

```

xpos1 += mx/16;
xpos2 += mx/60;
xpos3 -= mx/18;
xpos4 -= mx/64;

```

And we send the x-positions of the four lines via the address "/P5/xpos" in this way:

```

//create OSC message and add the four x-positions
OscMessage xposMessage = new OscMessage("/P5/xpos");
xposMessage.add(xpos1);
xposMessage.add(xpos2);
xposMessage.add(xpos3);
xposMessage.add(xpos4);
//send it
oscP5.send(xposMessage, myRemoteLocation);

```

This is the complete Processing code:

```

//import oscP5 and netP5 libraries
import oscP5.*;
import netP5.*;
//initialize objects
OscP5 oscP5;
NetAddress myRemoteLocation;

//variables for the x position of the four lines
//(adapted from processings Basics>Math>Distance 1D example)
float xpos1;
float xpos2;
float xpos3;
float xpos4;
int thin = 8;
int thick = 36;

void setup(){
    size(640, 360);
    noStroke();
    xpos1 = width/2;
    xpos2 = width/2;
    xpos3 = width/2;
    xpos4 = width/2;
    //create OSC object, listening at port 12001
    oscP5 = new OscP5(this,12001);
    //create NetAddress for sending: localhost at port 12002
    myRemoteLocation = new NetAddress("127.0.0.1",12002);
}

void draw(){
    //create movement depending on mouse position
    background(0);
    float mx = mouseX * 0.4 - width/5.0;
    fill(102);
    rect(xpos2, 0, thick, height/2);
    rect(xpos4, height/2, thick, height/2);
    fill(204);
    rect(xpos1, 0, thin, height/2);
    rect(xpos3, height/2, thin, height/2);
    xpos1 += mx/16;
    xpos2 += mx/60;
    xpos3 -= mx/18;
    xpos4 -= mx/64;
    if(xpos1 < -thin) { xpos1 = width; }
    if(xpos1 > width) { xpos1 = -thin; }
    if(xpos2 < -thick) { xpos2 = width; }
    if(xpos2 > width) { xpos2 = -thick; }
    if(xpos3 < -thin) { xpos3 = width; }
    if(xpos3 > width) { xpos3 = -thin; }
    if(xpos4 < -thick) { xpos4 = width; }
    if(xpos4 > width) { xpos4 = -thick; }

    //create OSC message and add the four x-positions
    OscMessage xposMessage = new OscMessage("/P5/xpos");
    xposMessage.add(xpos1);
    xposMessage.add(xpos2);
    xposMessage.add(xpos3);
    xposMessage.add(xpos4);
    //send it
    oscP5.send(xposMessage, myRemoteLocation);
}

```

On the Csound side, we receive the four x-positions on “/p5/xpos” as floating point numbers and

write it to the global array `gkPos`:

```
kAns,gkPos[] OSClisten iPort, "/P5/xpos", "ffff"
```

Each line in the Processing sketch is played by one instance of instrument “Line” in this way:

- the thick lines have a lower pitch than the thin lines
- in the middle of the canvas the sounds are louder (-20 dB compared to -40 dB at borders)
- in the middle of the canvas the pitch is higher (one octave compared to the left/right)

To achieve this, we build a function table `iTriangle` with 640 points (as much as the Processing canvas has x-points), containing a straight line from zero at left and right, to one in the middle:

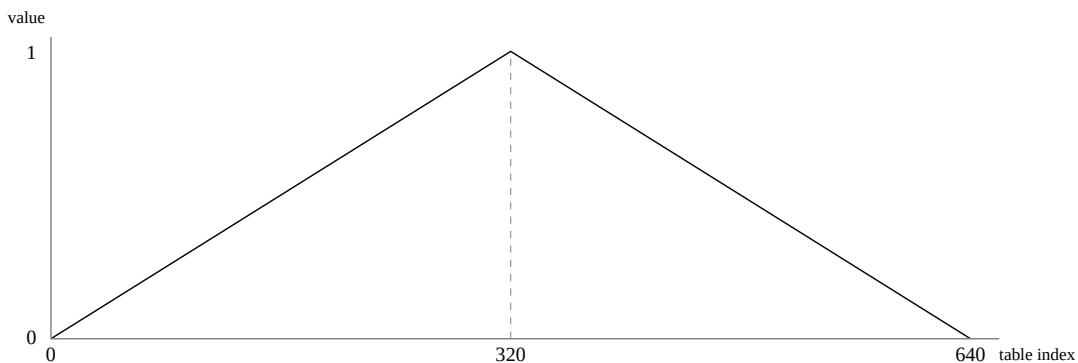


Figure 52.1: `iTriangle` function table

The incoming x position is used for both, the volume (dB) and the pitch (MIDI). A vibrato is added to the sine waves (smaller but faster for higher pitches) to the sine waves, and the panning reflects the position of the lines between left and right.

EXAMPLE 08A07_P5_Csound OSC.csd

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

sr = 44100
ksmps = 64
nchnls = 2
0dbfs = 1

instr GetLinePositions
//initialize port to receive OSC messages
iPort = OSCinit(12002)
//write the four x-positions in the global array gkPos
kAns,gkPos[] OSClisten iPort, "/P5/xpos", "ffff"
//call four instances of the instrument Line
indx = 0
while indx < 4 do
  schedule("Line", 0, 9999, indx+1)
  indx += 1
od
endin
```

```

schedule( "GetLinePositions" , 0,-1)

instr Line
  iLine = p4 //line 1-4 in processing
  kPos = gkPos[iLine-1]
  //table for volume (db) and pitch (midi)
  iTriangle = ftgen(0,0,641,-7,0,320,1,320,0)
  //volume is -40 db left/right and -20 in the middle of the screen
  kVolDb = tablei:k(kPos,iTriangle)*30 - 40
  //reduce by 4 db for the higher pitches (line 1 and 3)
  kVolDb = (iLine % 2 != 0) ? kVolDb-4 : kVolDb
  //base pitch is midi 50 for thick and 62 for thin lines
  iMidiBasePitch = (iLine % 2 == 0) ? 50 : 62
  //pitch is one octave plus iLine higher in the middle
  kMidiPitch = table:k(kPos,iTriangle)*12 + iMidiBasePitch+iLine
  //vibrato depending on line number
  kVibr = randi:k(iLine/4, 100/iLine,2)
  //generate sound and apply gentle fade in
  aSnd = poscil:a(ampdb(kVolDb),mtof:k(kMidiPitch+kVibr))
  aSnd *= linseg:a(0,1,1)
  //panning follows position on screen
  aL, aR pan2 aSnd, kPos/640
  out(aL,aR)
endin

</CsInstruments>
<CsScore>
</CsScore>
</CsoundSynthesizer>
;example by joachim heintz

```

Open Sound Control is the way to communicate between Processing and Csound as independent applications. Another way for connecting these extensive libraries for image and audio processing is by using JavaScript, and run both in a browser. Have a look at chapter [10 F](#) and [12 G](#) for more information.

08 B. CSOUND AND ARDUINO

It is the intention of this chapter to suggest a number of ways in which Csound can be paired with an Arduino prototyping circuit board. It is not the intention of this chapter to go into any detail about how to use an Arduino, there is already a wealth of information available elsewhere online about this. It is common to use an Arduino and Csound with another program functioning as an interpreter, so therefore some time is spent discussing these other programs.

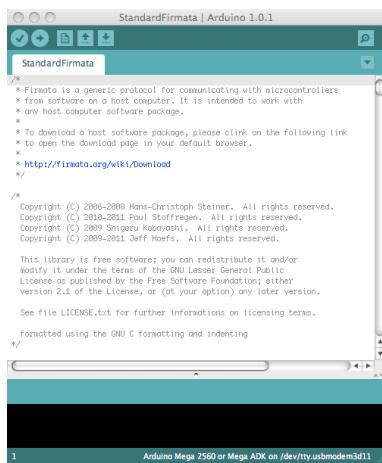
An Arduino is a simple microcontroller circuit board that has become enormously popular as a component in multidisciplinary and interactive projects for musicians and artists since its introduction in 2005. An Arduino board can be programmed to do many things and to send and receive data to and from a wide variety of other components and devices. As such it is impossible to specifically define its function here. An Arduino is normally programmed using its own development environment (IDE). A program is written on a computer which is then uploaded to the Arduino; the Arduino then runs this program, independent of the computer if necessary. Arduino's IDE is based on that used by [Processing](#) and [Wiring](#). Arduino programs are often referred to as *sketches*. There now exists a plethora of Arduino variants and even a number of derivatives and clones but all function in more or less the same way.

Interaction between an Arduino and Csound is essentially a question of communication and as such a number of possible solutions exist. This chapter will suggest several possibilities and it will then be up to the user to choose the one most suitable for their requirements. Most Arduino boards communicate using serial communication (normally via a USB cable). A number of Arduino programs, called *Firmata*, exist that are intended to simplify and standardise communication between Arduinos and software. Through the use of a Firmata the complexity of Arduino's serial communication is shielded from the user and a number of simpler objects, ugens or opcodes (depending on what the secondary software is) can instead be used to establish communication. Unfortunately Csound is rather poorly served with facilities to communicate using the *Firmata* (although this will hopefully improve in the future) so it might prove easiest to use another program (such as Pd or Processing) as an intermediary between the Arduino and Csound.

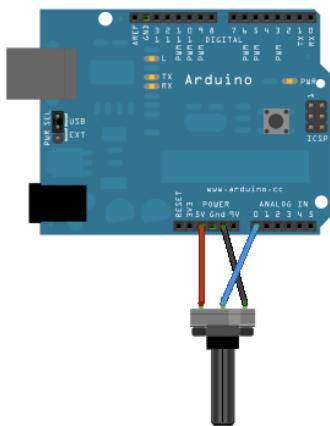
Arduino - Pd - Csound

First we will consider communication between an Arduino (running a Standard Firmata) and Pd. Later we can consider the options for further communication from Pd to Csound.

Assuming that the [Arduino IDE](#) (integrated development environment) has been installed and that the Arduino has been connected, we should then open and upload a Firmata sketch. One can normally be found by going to *File* -> *Examples* -> *Firmata* -> ... There will be a variety of flavours from which to choose but *StandardFirmata* should be a good place to start. Choose the appropriate Arduino board type under *Tools* -> *Board* -> ... and then choose the relevant serial port under *Tools* -> *Serial Port* -> ... Choosing the appropriate serial port may require some trial and error but if you have chosen the wrong one this will become apparent when you attempt to upload the sketch. Once you have established the correct serial port to use, it is worth taking a note of which number on the list (counting from zero) this corresponds to as this number will be used by Pd to communicate with the Arduino. Finally upload the sketch by clicking on the right-pointing arrow button.

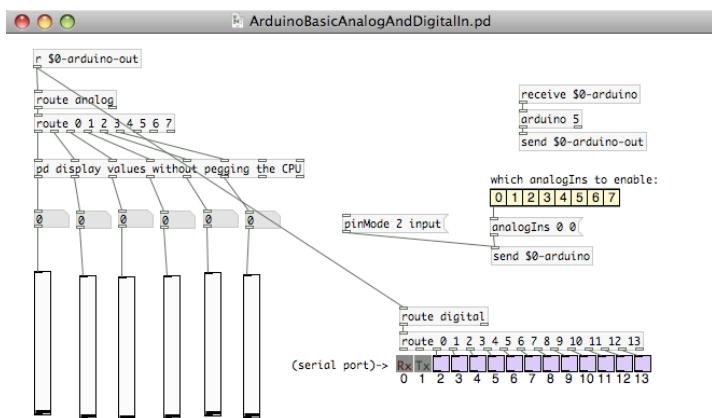


Assuming that [Pd](#) is already installed, it will also be necessary to install an add-on library for Pd called [Pduino](#). Follow its included instructions about where to place this library on your platform and then reopen Pd. You will now have access to a set of Pd objects for communicating with your Arduino. The Pduino download will also have included a number of examples Pd. *arduino-test.pd* will probably be the best patch to start. First set the appropriate serial port number to establish communication and then set Arduino pins as *input*, *output* etc. as you desire. It is beyond the scope of this chapter to go into further detail regarding setting up an Arduino with sensors and auxiliary components, suffice to say that communication to an Arduino is normally tested by *blinking* digital pin 13 and communication from an Arduino is normally tested by connecting a 10 kilo-ohm (10k) potentiometer to analog pin zero. For the sake of argument, we shall assume in this tutorial that we are setting the Arduino as a hardware controller and have a potentiometer connected to pin 0.

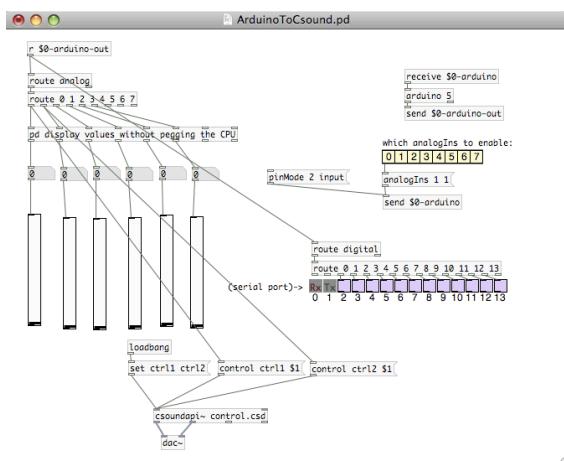


This picture below demonstrates a simple Pd patch that uses Pduino's objects to receive com-

munication from Arduino's analog and digital inputs. (Note that digital pins 0 and 1 are normally reserved for serial communication if the USB serial communication is unavailable.) In this example serial port 5 has been chosen. Once the analogIns enable box for pin 0 is checked, moving the potentiometer will change the values in the left-most number box (and move the slider connected to it). Arduino's analog inputs generate integers with 10-bit resolution (0 - 1023) but these values will often be rescaled as floats within the range 0 - 1 in the host program for convenience.



Having established communication between the Arduino and Pd we can now consider the options available to us for communicating between Pd and Csound. The most obvious (but not necessarily the best or most flexible) method is to use Pd's `csound6~` object. The above example could be modified to employ `csound6~` as shown below.



The outputs from the first two Arduino analog controls are passed into Csound using its API. Note that we should use the unpegged (not quantised in time) values directly from the `route` object. The Csound file `control.csd` is called upon by Pd and it should reside in the same directory as the Pd patch. Establishing communication to and from Pd could employ code such as that shown below. Data from controller one (Arduino analog 0) is used to modulate the amplitude of an oscillator and data from controller two (Arduino analog 1) varies its pitch across a four octave range.

EXAMPLE 08B01_Pd_to_Csound.csd

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

```

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

instr 1
; read in controller data from Pd via the API using 'invalue'
kctrl1 invalue "ctrl1"
kctrl2 invalue "ctrl2"
; re-range controller values from 0 - 1 to 7 - 11
koct = (kctrl2*4)+7
; create an oscillator
a1 vco2 kctrl1,cpsoct(koct),4,0.1
outs a1,a1
endin
</CsInstruments>
<CsScore>
i 1 0 10000
</CsScore>
</CsoundSynthesizer>
;example by Iain McCurdy

```

Communication from Pd into Csound is established using the `invalue` opcodes and audio is passed back to Pd from Csound using `outs`. Note that Csound does not address the computer's audio hardware itself but merely passes audio signals back to Pd. Greater detail about using Csound within Pd can be found in the chapter [Csound in Pd](#).

A disadvantage of using the method is that in order to modify the Csound patch it will require being edited in an external editor, re-saved, and then the Pd patch will need to be reloaded to reflect these changes. This workflow might be considered rather inefficient.

Another method of data communication between PD and Csound could be to use MIDI. In this case some sort of MIDI connection node or virtual patchbay will need to be employed. On Mac this could be the IAC driver, on Windows this could be [MIDI Yoke](#) and on Linux this could be [Jack](#). This method will have the disadvantage that the Arduino's signal might have to be quantised in order to match the 7-bit MIDI controller format but the advantage is that Csound's audio engine will be used (not Pd's; in fact audio can be disabled in Pd) so that making modifications to the Csound file and hearing the changes should require fewer steps.

A final method for communication between Pd and Csound is to use OSC. This method would have the advantage that analog 10 bit signal would not have to be quantised. Again workflow should be good with this method as Pd's interaction will effectively be transparent to the user and once started it can reside in the background during working. Communication using OSC is also used between Processing and Csound so is described in greater detail below.

Arduino - Processing - Csound

It is easy to communicate with an Arduino using a Processing sketch and any data within Processing can be passed to Csound using OSC.

The following method makes use of the [Arduino](#) and [P5](#) (glove) libraries for processing. Again these need to be copied into the appropriate directory for your chosen platform in order for Pro-

cessing to be able to use them. Once again there is no requirement to actually know very much about Processing beyond installing it and running a patch (sketch). The following [sketch](#) will read all Arduino inputs and output them as OSC.



Start the Processing sketch by simply clicking the triangle button at the top-left of the GUI. Processing is now reading serial data from the Arduino and transmitting this as OSC data within the computer.

The OSC data sent by Processing can be read by Csound using its own OSC opcodes. The following example simply reads in data transmitted by Arduino's analog pin 0 and prints changed values to the terminal. To read in data from all analog and digital inputs you can use Iain McCurdy's [Arduino_Processing_OSC_Csound.csd](#).

EXAMPLE 08B02_Processing_to_Csound.csd

```
<CsoundSynthesizer>
<CsOptions>
-o dac
</CsOptions>
<CsInstruments>
sr = 44100
ksmps = 8
nchnls = 1
0dbfs = 1

; handle used to reference osc stream
gihandle OSCinit 12001

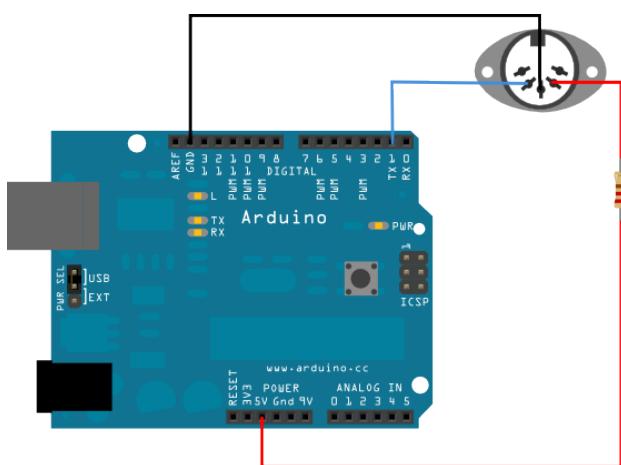
instr 1
; initialise variable used for analog values
gkana0    init      0
; read in OSC channel '/analog/0'
gktrigana0 OSCListen gihandle, "/analog/0", "i", gkana0
; print changed values to terminal
        printk2    gkana0
endin

</CsInstruments>
<CsScore>
i 1 0 3600
</CsScore>
</CsoundSynthesizer>
;example by Iain McCurdy
```

Also worth investigating is Jacob Joaquin's [Csoundo](#) - a Csound library for Processing. This library will allow closer interaction between Processing and Csound in the manner of the `csound6~` object in Pd. This project has more recently been developed by Rory Walsh.

Arduino as a MIDI Device

Some users might find it most useful to simply set the Arduino up as a MIDI device and to use that protocol for communication. In order to do this all that is required is to connect MIDI pin 4 to the Arduino's 5v via a 200k resistor, to connect MIDI pin 5 to the Arduino's TX (serial transmit) pin/pin 1 and to connect MIDI pin 2 to ground, as shown below. In order to program the Arduino it will be necessary to install Arduino's [MIDI library](#).



Programming an Arduino to generate a MIDI controller signal from analog pin 0 could be done using the following code:

```
// example written by Iain McCurdy
// import midi library
#include <MIDI.h>

const int analogInPin = A0; // choose analog input pin
int sensorValue = 0; // sensor value variable
int oldSensorValue = 0; // sensor value from previous pass
int midiChannel = 1; // set MIDI channel

void setup()
{
  MIDI.begin(1);
}

void loop()
{
  sensorValue = analogRead(analogInPin);

  // only send out a MIDI message if controller has changed
  if (sensorValue!=oldSensorValue)
  {
    // controller 1, rescale value from 0-1023 (Arduino) to 0-127 (MIDI)
    MIDI.sendControlChange(1,sensorValue/8,midiChannel);
    oldSensorValue = sensorValue; // set old sensor value to current
  }
}
```

```

    }
    delay(10);
}

```

Data from the Arduino can now be read using Csound's [ctrl7](#) opcodes for reading MIDI controller data.

The Serial Opcodes

Serial data can also be read directly from the Arduino by Csound by using Matt Ingalls' opcodes for serial communication: [serialBegin](#) and [serialRead](#).

An example Arduino sketch for serial communication could be as simple as this:

```

// Example written by Matt Ingalls
// ARDUINO CODE:

void setup() {
    // enable serial communication
    Serial.begin(9600);

    // declare pin 9 to be an output:
    pinMode(9, OUTPUT);
}

void loop()
{
    // only do something if we received something
    // (this should be at csound's k-rate)
    if (Serial.available())
    {

        // set the brightness of LED (connected to pin 9) to our input value
        int brightness = Serial.read();
        analogWrite(9, brightness);

        // while we are here, get our knob value and send it to csound
        int sensorValue = analogRead(A0);
        Serial.write(sensorValue/4); // scale to 1-byte range (0-255)
    }
}

```

It will be necessary to provide the correct address of the serial port to which the Arduino is connected (in the given example the Windows platform was being used and the port address was [/COM4](#)).

It will be necessary to scale the value to correspond to the range provided by a single byte (0-255) so therefore the Arduino's 10 bit analog input range (0-1023) will have to be divided by four.

EXAMPLE 08B03_Serial_Read.csd

```

<CsoundSynthesizer>
; Example written by Matt Ingalls
; run with a commandline something like:

```

```

; csound --opcode-lib=serialOpcodes.dylib serialdemo.csd -odac -iadc

<CsOptions>
</CsOptions>
;--opcode-lib=serialOpcodes.dylib -odac
<CsInstruments>

ksmps = 500 ; the default krate can be too fast for the arduino to handle
0dbfs = 1

instr 1
    iPort    serialBegin      "/COM4", 9600
    kVal     serialRead       iPort
            printk2           kVal
endin

</CsInstruments>
<CsScore>
i 1 0 3600
</CsScore>
</CsoundSynthesizer>

```

This example will read serial data from the Arduino and print it to the terminal. Reading output streams from several of Arduino's sensor inputs simultaneously will require more complex parsing of data within Csound as well as more complex packaging of data from the Arduino. This is demonstrated in the following example which also shows how to handle serial transmission of integers larger than 255 (the Arduino analog inputs have 10 bit resolution).

First the Arduino sketch, in this case reading and transmitting two analog and one digital input:

```

// Example written by Sigurd Saupe
// ARDUINO CODE:

// Analog pins
int potPin = 0;
int lightPin = 1;

// Digital pin
int buttonPin = 2;

// Value IDs (must be between 128 and 255)
byte potID = 128;
byte lightID = 129;
byte buttonID = 130;

// Value to toggle between inputs
int select;

/*
** Two functions that handles serial send of numbers of varying length
*/

// Recursive function that sends the bytes in the right order
void serial_send_recursive(int number, int bytePart)
{
    if (number < 128) {          // End of recursion
        Serial.write(bytePart); // Send the number of bytes first
    }
    else {
        serial_send_recursive((number >> 7), (bytePart + 1));
    }
    Serial.write(number % 128); // Sends one byte

```

```

}

void serial_send(byte id, int number)
{
    Serial.write(id);
    serial_send_recursive(number, 1);
}

void setup() {
    // enable serial communication
    Serial.begin(9600);
    pinMode(buttonPin, INPUT);
}

void loop()
{
    // Only do something if we received something (at csound's k-rate)
    if (Serial.available())
    {
        // Read the value (to empty the buffer)
        int csound_val = Serial.read();

        // Read one value at the time (determined by the select variable)
        switch (select) {
            case 0: {
                int potVal = analogRead(potPin);
                serial_send(potID, potVal);
            }
            break;
            case 1: {
                int lightVal = analogRead(lightPin);
                serial_send(lightID, lightVal);
            }
            break;
            case 2: {
                int buttonVal = digitalRead(buttonPin);
                serial_send(buttonID, buttonVal);
            }
            break;
        }

        // Update the select (0, 1 and 2)
        select = (select+1)%3;
    }
}

```

The solution is similar to MIDI messages. You have to define an ID (a unique number ≥ 128) for every sensor. The ID behaves as a status byte that clearly marks the beginning of a message received by Csound. The remaining bytes of the message will all have a most significant bit equal to zero (value < 128). The sensor values are transmitted as ID, length (number of data bytes), and the data itself. The recursive function `serial_send_recursive` counts the number of data bytes necessary and sends the bytes in the correct order. Only one sensor value is transmitted for each run through the Arduino loop.

The Csound code receives the values with the ID first. Of course you have to make sure that the IDs in the Csound code matches the ones in the Arduino sketch. Here's an example of a Csound orchestra that handles the messages sent from the Arduino sketch:

EXAMPLE 08B04_Serial_Read_multiple.csd

```

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

sr  = 44100
ksmps = 500 ; the default krate can be too fast for the arduino to handle
nchnls = 2
0dbfs = 1

giSaw  ftgen 0, 0, 4096, 10, 1, 1/2, 1/3, 1/4, 1/5, 1/6, 1/7, 1/8

instr 1

; Initialize the three variables to read
kPot    init 0
kLight  init 0
kButton init 0

iPort    serialBegin "/COM5", 9600 ;connect to the arduino with baudrate = 9600
           serialWrite iPort, 1      ;Triggering the Arduino (k-rate)

kValue   = 0
kType    serialRead iPort          ; Read type of data (pot, light, button)

if (kType >= 128) then

    kIndex = 0
    kSize   serialRead iPort

    loopStart:
        kValue      = kValue << 7
        kByte       serialRead iPort
        kValue      = kValue + kByte
        loop_lt kIndex, 1, kSize, loopStart
endif

if (kValue < 0) kgoto continue

if (kType == 128) then          ; This is the potmeter
    kPot    = kValue
elseif (kType == 129) then      ; This is the light
    kLight  = kValue
elseif (kType == 130) then      ; This is the button (on/off)
    kButton = kValue
endif

continue:

; Here you can do something with the variables kPot, kLight and kButton
; printk "Pot %f\n", 1, kPot
; printk "Light %f\n", 1, kLight
; printk "Button %d\n", 1, kButton

; Example: A simple oscillator controlled by the three parameters
kAmp    port    kPot/1024, 0.1
kFreq   port    (kLight > 100 ? kLight : 100), 0.1
aOut    oscil   kAmp, kFreq, giSaw

if (kButton == 0) then

```

```
        out      a0out
endif

endin

</CsInstruments>
<CsScore>
i 1 0 60      ; Duration one minute
e
</CsScore>
</CsoundSynthesizer>
;example written by Sigurd Saupe
```

Remember to provide the correct address of the serial port to which the Arduino is connected (the example uses `/COM5`).

HID

Another option for communication has been made available by a new Arduino board called *Leonardo*. It pairs with a computer as if it were an HID (Human Interface Device) such as a mouse, keyboard or a gamepad. Sensor data can therefore be used to imitate the actions of a mouse connected to the computer or keystrokes on a keyboard. Csound is already equipped with opcodes to make use of this data. Gamepad-like data is perhaps the most useful option though and there exist opcodes (at least in the Linux version) for reading gamepad data. It is also possible to read in data from a gamepad using [pygame](#) and Csound's python opcodes.

08 C. CSOUND VIA UDP

Using Csound via UDP with the `-port` Option

The `-port=N` option allows users to send orchestras to be compiled on-the-fly by Csound via UDP connection. This way, Csound can be started with no instruments, and will listen to messages sent to it. Many programs are capable of sending UDP messages, and scripting languages, such as Python, can also be used for this purpose. The simplest way of trying out this option is via the netcat program, which can be used in the terminal via the nc command.

Let's explore this as an example of the `-port` option.

First, Csound is started with the following command:

```
$ csound -odac --port=1234
```

Alternatively, if using a frontend such as CsoundQt, it is possible run an empty CSD, with the `-port` in its CsOptions field:

EXAMPLE 10F01_csound_udp.csd

```
<CsoundSynthesizer>
<CsOptions>
--port=1234
</CsOptions>
<CsInstruments>
</CsInstruments>
<CsScore>
</CsScore>
</CsoundSynthesizer>
```

This will start Csound in a daemon mode, waiting for any UDP messages in port 1234. Now with netcat, orchestra code can be sent to Csound. A basic option is to use it interactively in the terminal, with a heredocument command (`<<`) to indicate the end of the orchestra we are sending:

```
$ nc -u 127.0.0.1 1234 << EOF
> instr 1
> a1 oscili p4*0dbfs,p5
> out a1
> endin
> schedule 1,0,1,0.5,440
> EOF
```

Csound will respond with a 440Hz sinewave. The **ctl-c** key combination can be used to close nc and go back to the shell prompt. Alternatively, we could write our orchestra code to a file and then send it to Csound via the following command (orch is the name of our file):

```
$ nc -u 127.0.0.1 1234 < orch
```

Csound performance can be stopped in the usual way via **ctl-c** in the terminal, or through the dedicated transport controls in a frontend. We can also close the server it via a special UDP message:

```
ERROR WITH MACRO close
```

However, this will not close Csound, but just stop the UDP server.

09 A. CSOUND IN PD

Installing

You can embed Csound in PD via the external object **csound6~** which has been written by Victor Lazzarini. This external is either part of the Csound distribution or can be built from the sources at https://github.com/csound/csound_pd. In the examples folder of this repository you can also find all the .csd and .pd files of this chapter.

On **Ubuntu Linux**, you can install the `csound6~` via the Synaptic Package Manager. Just look for `csound6~` or `pd-csound`, check *install*, and your system will install the library at the appropriate location. If you build Csound from sources, go to the [csound_pd repository](#) and follow the build instructions. Once it is compiled, the object will appear as `csound6~.pd_linux` and should be copied (together with `csound6~.help.pd`) to `/usr/lib/pd/extr`a, so that PD can find it. If not, add it to PD's search path (*File->Path...*).

On **Mac OSX**, you find the `csound6~` external, help file and examples in the `release` directory of the `csound_pd` repository. (Prior to 6.11, the `csound6~` was in `/Library/Frameworks/CsoundLib64.framework/Versions/6.0/Resources/PD` after installing Csound.)

Put these files in a folder which is in PD's search path. For PD-extended, it is 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**, you find the `csound6~` external, help file and examples in the `release` directory of the `csound_pd` repository, too.

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_pdcs_control_in.csd

```

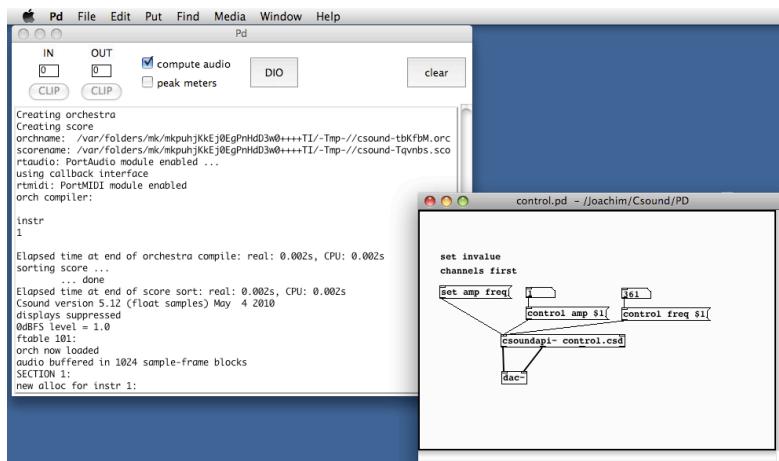
<CsoundSynthesizer>
<CsOptions>
</CsOptions>
<CsInstruments>
sr = 44100
nchnls = 2
0dbfs = 1
ksmps = 8

instr 1
kFreq    invalue  "freq"
kAmp     invalue  "amp"
aSin      oscil    kAmp, kFreq
                out      aSin, aSin
endin

</CsInstruments>
<CsScore>
i 1 0 10000
</CsScore>
</CsoundSynthesizer>
;example by joachim heintz

```

Save this file under the name *control.csd*. Save a PD window in the same folder and create the following patch:



Note that for *invalue* channels, you first must register these channels by a *set* message. The usage of *chnget* is easier; a simple example can be found in [this](#) example in the *csound6~* repository.

As you see, the first two outlets of the *csound6~* 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. The number of audio inlets created in the *csound6~* object will depend on the number of input channels used in the Csound orchestra. The following .csd uses two audio inputs:

EXAMPLE 09A02_pdcs_live_in.csd

```

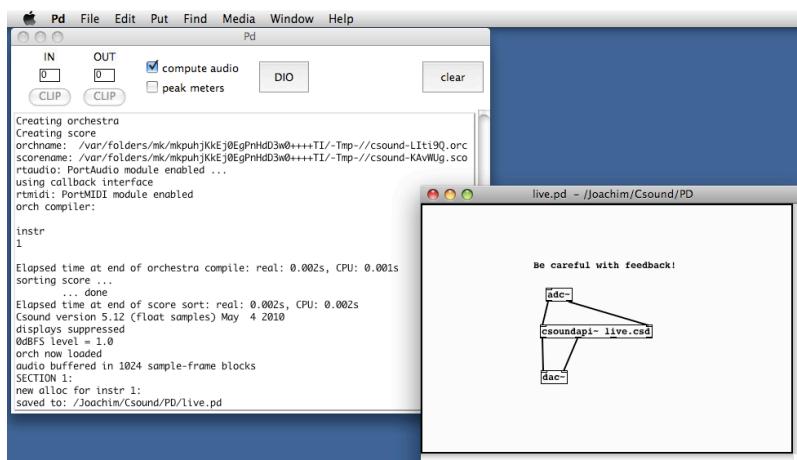
<CsoundSynthesizer>
<CsOptions>
</CsOptions>
<CsInstruments>
sr = 44100
0dbfs = 1
ksmps = 8
nchnls = 2

instr 1
aL     inch    1
aR     inch    2
kcfl    randomi 100, 1000, 1; center frequency
kcfr    randomi 100, 1000, 1; 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

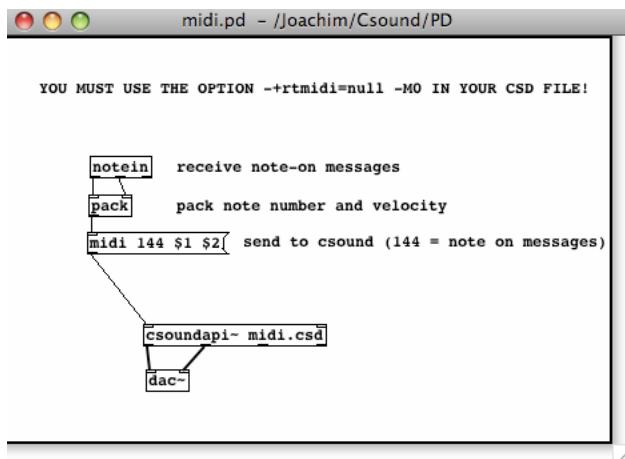
</CsInstruments>
<CsScore>
i 1 0 10000
</CsScore>
</CsoundSynthesizer>
;example by joachim heintz

```

The corresponding PD patch is extremely simple:

**MIDI**

The `csound6~` 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:



This is the corresponding midi.csd. It must contain the options `-+rtmidi=null -M0` in the `<CsOptions>` tag. It is an FM synth in which the modulation index is defined according to the note velocity. The harder a key is truck, the higher the index of modulation will be; and therefore a greater number of stronger partials will be created. The ratio is calculated randomly between two limits, which can be adjusted.

EXAMPLE 09A03_pdcs_midi.csd

```

<CsoundSynthesizer>
<CsOptions>
-+rtmidi=null -M0
</CsOptions>
<CsInstruments>
sr      =  44100
ksmps   =  8
nchnls  =  2
0dbfs  =  1

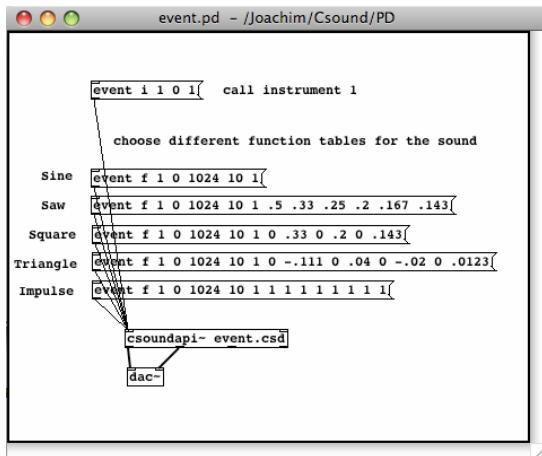
giSine    ftgen    0, 0, 2^10, 10, 1

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>
;example by joachim heintz
  
```

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.



EXAMPLE 09A04_pdcs_events.csd

```

<CsoundSynthesizer>
<CsOptions>
</CsOptions>
<CsInstruments>
sr = 44100
ksmps = 8
nchnls = 2
0dbfs = 1
seed 0; each time different seed

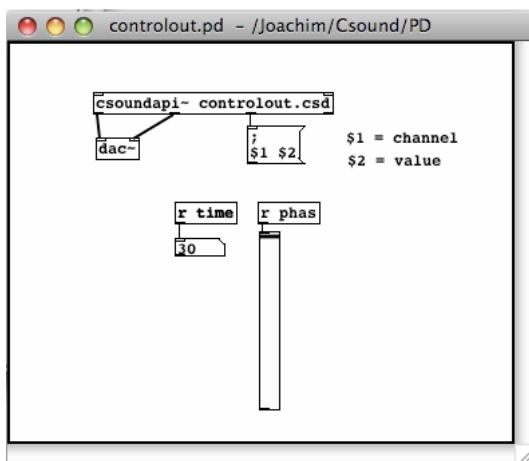
instr 1
iDur    random  0.5, 3
p3      =        iDur
iFreq1  random  400, 1200
iFreq2  random  400, 1200
idB    random -18, -6
kFreq   linseg  iFreq1, iDur, iFreq2
kEnv    transeg ampdb(idB), p3, -10, 0
aTone   oscil   kEnv, kFreq
        outs    aTone, aTone
endin

</CsInstruments>
<CsScore>
</CsScore>
</CsoundSynthesizer>
;example by joachim heintz

```

Control Output

If you want Csound to pass any control data to PD, you can use the opcode `outvalue`. You will receive this data at the second outlet from the right of the `csound6~` 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_pdcs_control_out.csd

```

<CsoundSynthesizer>
<CsOptions>
</CsOptions>
<CsInstruments>
sr = 44100
nchnls = 2
0dbfs = 1
ksmps = 8

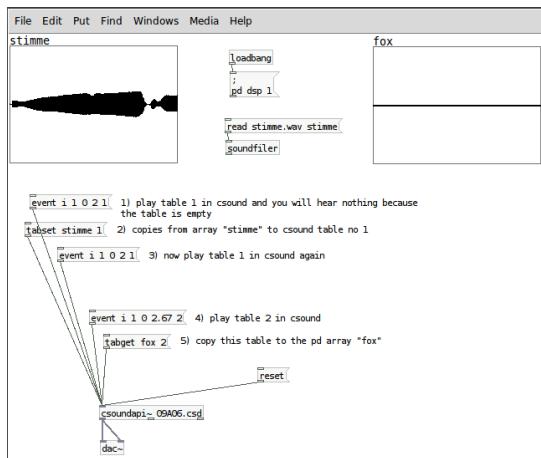
instr 1
ktim    times
kphas   phasor  1
          outvalue "time", ktim
          outvalue "phas", kphas*127
endin

</CsInstruments>
<CsScore>
i 1 0 30
</CsScore>
</CsoundSynthesizer>
;example by joachim heintz

```

Send/Receive Buffers from PD to Csound and back

A PD array can be sent directly to Csound, and a Csound function table to PD. The message `tabset array-name ftable-number` copies a PD array into a Csound function table. The message `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`.



EXAMPLE 06A06_pdcs_tabset_tabget.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)
iphc      =      iskip / (ftlen(ifn) / sr)
asig      poscils  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 would recommend starting with `ksmps=8`. If there are performance problems, try to increase this value to 16, 32, or 64, i.e. ascending powers of 2.

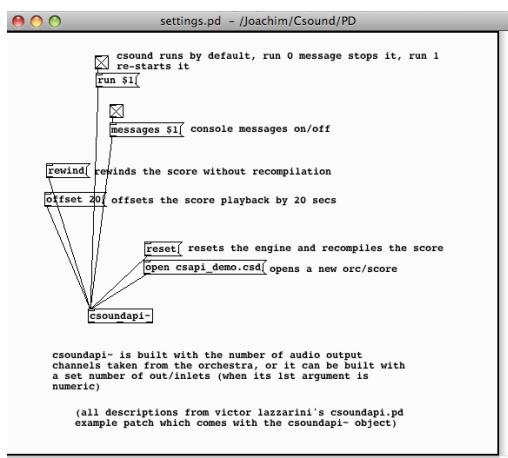
The `csound6~` object runs by default if you turn on audio in PD. You can stop it by sending a `run 0` message, and start it again with a `run 1` message.

You can recompile the `csd` file of a `csound6~` object by sending a `reset` message.

By default, you see all the messages of Csound in the PD window. If you do not want to see them, send a `message 0` message. `message 1` re-enables message printing.

If you want to open a new `.csd` file in the `csound6~` 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 that number of seconds.



09 B. CSOUND IN MAXMSP

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.

Note: Most of the descriptions below have been written years ago by Davis Pyon. They may be outdated and will need to be updated.

Installing

The `csound~` requires an installation of Csound. The external can be downloaded on [Csound's download page](#) (under *Other*).

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_maxcs_helloworld.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.

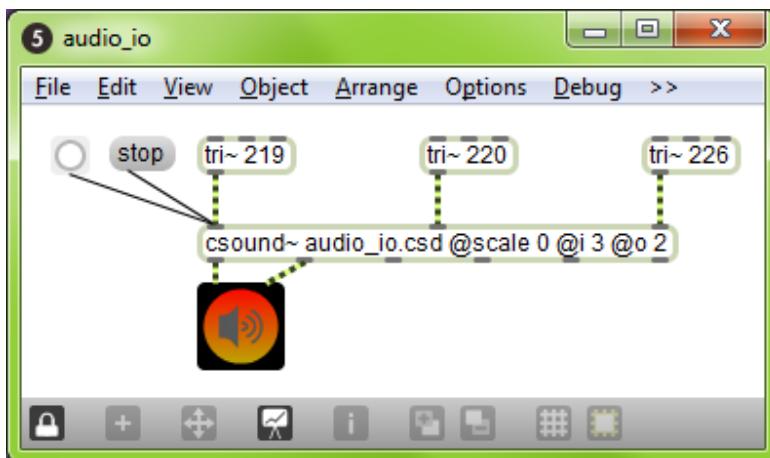
Audio I/O

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 outlets. 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_maxcs_audio_io.csd

```
<CsoundSynthesizer>
<CsInstruments>
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>
;example by Davis Pyon
```

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

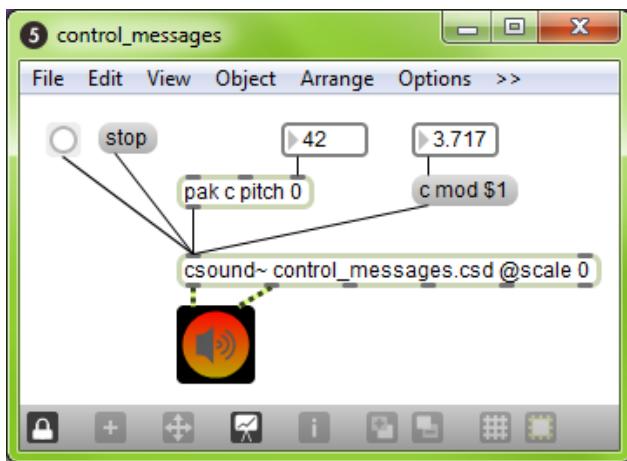
Control messages allow you to send numbers to Csound. It is the primary way to control Csound parameters at i-rate or k-rate. To control a-rate (audio) parameters, you must use and audio inlet.

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 Csound file demonstrates control messages:



EXAMPLE 09B03_maxcs_control_in.csd

```
<CsoundSynthesizer>
<CsInstruments>
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    oscil .2, cpsmidinn(kPitch), 2, kMod, 1.5, giSine
       outch 1, aFM, 2, aFM
endin
</CsInstruments>
<CsScore>
f0 86400
i1 0 86400
e
</CsScore>
</CsoundSynthesizer>
;example by Davis Pyon
```

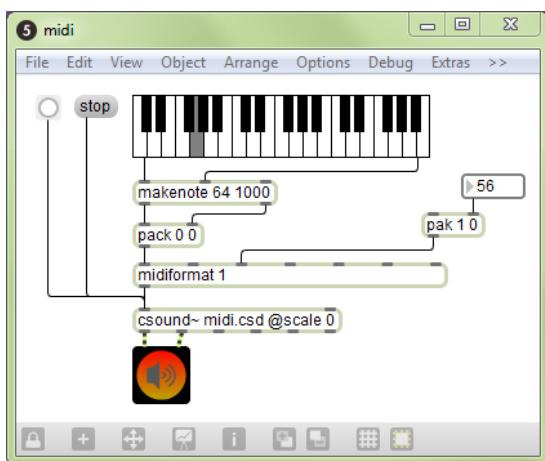
In the patch, notice that we use two different methods to construct control messages. The *pak*

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 Csound 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 its 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_maxcs_midi.csd

```
<CsoundSynthesizer>
<CsInstruments>
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     oscil .2, iPitch, 2, kMod, 1.5, giSine
        outch 1, aFM, 2, aFM
endin
</CsInstruments>
```

```
<CsScore>
</CsScore>
</CsoundSynthesizer>
;example by Davis Pyon
```

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.

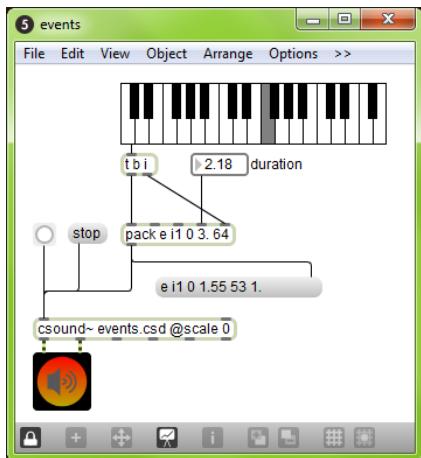
In the Csound file, notice the *massign* statements in the header of the orchestra. *massign 0, 0* tells Csound to clear all mappings between MIDI channels and Csound instrument numbers. This is highly recommended because forgetting to add this statement may cause confusion somewhere down the road. The 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]).¹

Notice that in the score section of the Csound file, we no longer have the statement *i1 0 86400* as we had in earlier examples. The score section is left empty here, so that instrument 1 is compiled but not activated. Activation is done via MIDI here.

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_maxcs_events.csd

```
<CsoundSynthesizer>
<CsInstruments>
sr      = 44100
ksmps  = 32
nchnls = 2
```

¹Csound's MIDI options and opcodes are described in detail in section 7 of this manual.

```
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>
</CsScore>
</CsoundSynthesizer>
;example by Davis Pyon
```

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 Csound 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.

09 C. 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.

The best choice currently is to use [Cabbage](#) to create Csound based plugins. See the [Cabbage chapter](#) in part 10 of this manual.

10 A. CSOUNDQT

CsoundQt (named *QuteCsound* until autumn 2011) is a free, cross-platform graphical frontend to Csound. It has been written by Andrés Cabrera and is maintained since 2016 by Tarmo Johannes. 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.

Installing

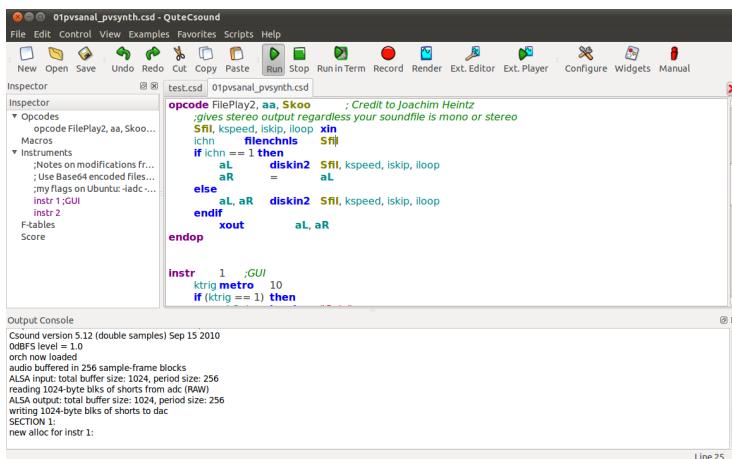
CsoundQt is a *frontend* for Csound, so **Csound needs to be installed first**. Make sure you have installed Csound before you install CsoundQt; otherwise it will not work at all.

CsoundQt is **included** in the Csound installers **for Mac OSX and Windows**. It is recommended to use the CsoundQt version which is shipped with the installer for compatibility between CsoundQt and Csound. The Windows installer will probably install CsoundQt automatically. For OSX, first install the Csound package, then open the CsoundQt disk image and copy the CsoundQt Application into the *Applications* folder.

For **Linux** there is a Debian/Ubuntu package. Unfortunately it is built without *RtMidi* support, so you will not be able to connect CsoundQt's widgets directly with your midi controllers. The alternative is to build CsoundQt with [QtCreator](#) which is not too hard and gives you all options, including the PythonQt connection. You will find instructions how to build in the [CsoundQt Wiki](#).

General Usage and Configuration

CsoundQt 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.

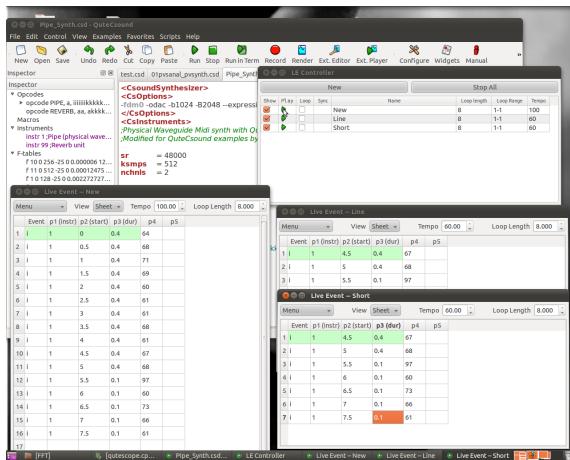


In the widget editor panel, you can create a variety of widgets to control Csound. To link the value from a widget, you first need to set its channel, and then use the Csound opcodes `invalue` or `chnget`. To send values to widgets, e.g. for data display, you need to use the `outvalue` or `chnset` opcode.



CsoundQt also implements the use of HTML and JavaScript code embedded in the optional `<html>` element of the CSD file. If this element is detected, CsoundQt will parse it out as a Web page, compile it, and display it in the *HTML5 Gui* window. HTML code in this window can control Csound via a selected part of the Csound API that is exposed in JavaScript. This can be used to define custom user interfaces, display video and 3D graphics, generate Csound scores, and much more. See chapter [Csound and Html](#) for more information.

CsoundQt also offers convenient facilities for score editing in a spreadsheet like environment which can be transformed using Python scripting (see also the chapter about [Python in CsoundQt](#)).



You will find more detailed information at [CsoundQt's home page](#).

Configuring CsoundQt

CsoundQt gives easy access to the most important [Csound options](#) and to many specific CsoundQt settings via its Configuration Panel. In particular the *Run* tab offers many choices which have to be understood and set carefully.

To open the configuration panel simply push the *Configure* button. The configuration panel comprises seven tabs. The available configurable parameters in each tab are described below for each tab.

The single options, their meaning and tips of how to set them are listed at the [Configuring CsoundQt](#) page of CsoundQt's website.

10 B. CABBAGE



Cabbage is a software for prototyping and developing audio instruments with the Csound audio synthesis language. Instrument development and prototyping is carried out with the main Cabbage IDE. Users write and compile Csound code in a code editor. If one wishes, they can also create a graphical frontend, although this is not essential. Any Csound file can be run with Cabbage, regardless of whether or not it has a graphical interface. Cabbage is designed for realtime processing in mind. While it is possible to use Cabbage to run Csound in the more traditional score-driven way, but your success may vary.

Cabbage is a ‘host’ application. It treats each and every Csound instruments as a unique native plugin, which gets added to a digital audio graph (DAG) once it is compiled. The graph can be opened and edited at any point during a performance. If one wishes to use one of their Csound instruments in another audio plugin host; such as Reaper, Live, Bitwig, Ardour, QTractor, etc. They can export the instrument through the ‘Export instrument’ option.

Download and Install

Cabbage is hosted on GitHub, and **pre-compiled binaries** for **Windows** and **OSX** can be found on the [release](#) section of Cabbage’s home page. If you run **Linux** you will need to build Cabbage yourself, but instructions are included with the source code. The main platform installers for Cabbage include an option of installing the latest version of Csound. If you already have a version of Csound

installed, you can skip this step. Note that you will **need to have Csound installed** one way or another in order to run Cabbage.

Using Cabbage

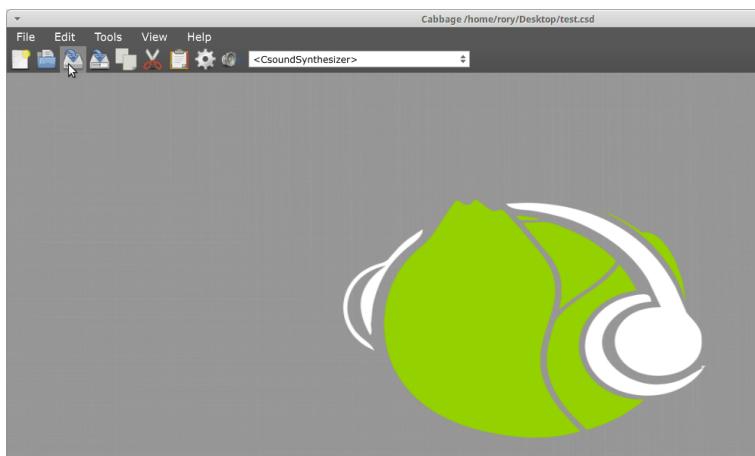
Instrument development and prototyping is carried out with the main Cabbage IDE. Users write and compile their Csound code in a code editor. Each Csound file opened with have a corresponding editor. If one wishes one can also create a graphical frontend, although this is no longer a requirement for Cabbage. Any Csound file can be run with Cabbage, regardless of whether or not it has a graphical interface. Each Csound files that is compiled by Cabbage will be added to an underlying digital audio graph. Through this graph users can manage and configure instrument to create patches of complex processing chains.

Opening files

User can open any .csd files by clicking on the Open File menu command, or toolbar button. Users can also browse the Examples menu from the main File menu. Cabbage ships with over 100 high-end instruments that can be modified, hacked, and adapted in any way you wish. Note that if you wish to modify the examples, use the Save-as option first. Although this is only required on Windows, it's a good habit to form. You don't want to constantly overwrite the examples with your own code. Cabbage can load and perform non-Cabbage score-driven .csd files. However, it also uses its own audio IO, so it will overwrite any -odac options set in the CsOptions section of a .csd file.

Creating a new file

New files can be created by clicking the New file button in the toolbar, or by clicking File->New Csound file from the main menu. When a new file is requested, Cabbage will give you the choice of 3 basic templates, as shown below.



The choices are:

- A new synth.

When this option is selected Cabbage will generate a simple synthesiser with an ADSR envelope and MIDI keyboard widget. In the world of VST, these instruments are referred to as VSTi's.

- A new effect.

When this option is selected Cabbage will create a simple audio effect. It will generate a simple Csound instrument that provides access to an incoming audio stream. It also generates code that will control the gain of the output.

- A new Csound file.

This will generate a basic Csound file without any graphical frontend.

Note that these templates are provided for quickly creating new instruments. One can modify any of the template code to convert it from a synth to an effect or vice versa.

Building/exporting instruments

To run an instrument users can use the controls at the top of the file's editor. Alternatively one can go to the 'Tools' menu and hit 'Build Instrument'. If you wish to export a plugin go to 'Export' and choose the type of plugin you wish to export. To use the plugin in a host, you will need to let the host know where your plugin file is located. On Windows and Linux, the corresponding .csd should be located in the same directory as the plugin dll. The situation is different on MacOS as the .csd file is automatically packaged into the plugin bundle.

```

1 <Cabbage>
2 form caption("Untitled") size(400, 300), colour(58, 110, 182), pluginID("def1")
3 keyboard bounds(8, 158, 381, 95)
4 </Cabbage>
5 <Csoundsynthesizer>
6 <CsOptions>
7 -n -d +rtmidi=NULL -M0 -m0d --midi-key-cps=4 --midi-velocity-amp=5
8 </CsOptions>
9 <CsInstruments>
10 ; Initialize the global variables.
11 sr = 44100
12 ksmps = 32
13 nchnls = 2
14 0dbfs = 1
15
16 ;instrument will be triggered by keyboard widget
17 instr 1
18  kEnv madsr .1, .2, .6, .4
19  aOut vco2 p5, p4
20  outs aOut*kEnv, aOut*kEnv
21 endin
22
23 </CsInstruments>
24 <CsScore>
25

```

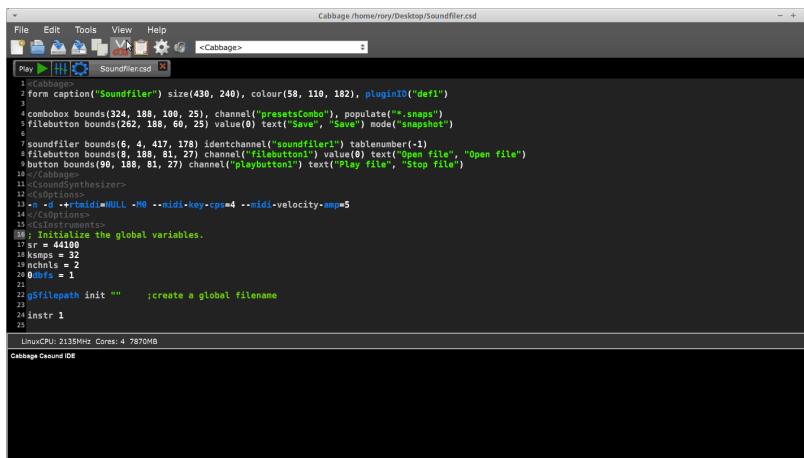
LinuxCPU: 3001MHz Cores: 4 7870MB
midi channel 16 using Instr 1

Closing a file will not stop it from performing. To stop a file from performing you must hit the Stop button.

Creating GUI interfaces for instruments

To create a GUI for your instrument you must enter edit mode for that instrument. You can do this by hitting the Edit mode button at the top of the file's editor, or by hitting Ctrl+e when the editor for that instrument have focus. Once in edit mode, each widget will have a thin white border around

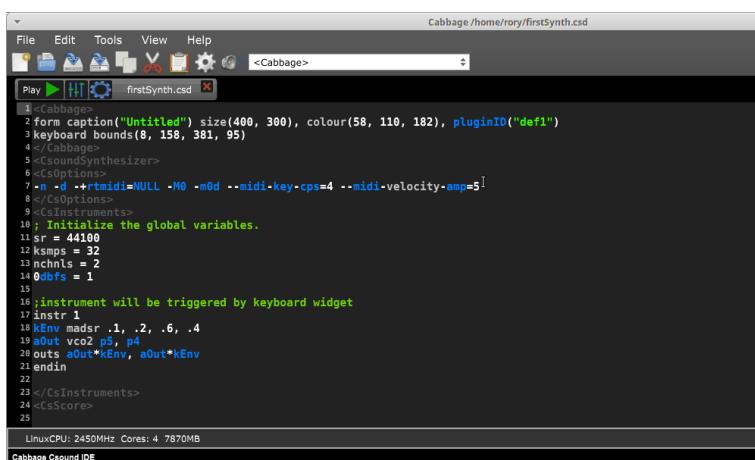
it. You can move widgets around whilst in edit. You can also right-click and insert new widgets, as well as modify their appearance using the GUI properties editor on the right-hand side of the screen.



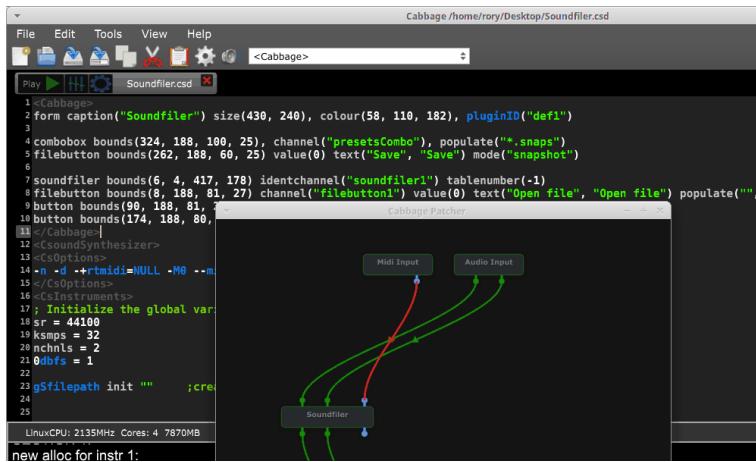
You will notice that when you select a widget whilst in edit mode, Cabbage will highlight the corresponding line of text in your source code. When updating GUI properties, hit ‘Enter’ when you are editing single line text or numeric properties, ‘Tab’ when you are editing multi-line text properties, and ‘Escape’ when you are editing colours.

Editing the audio graph

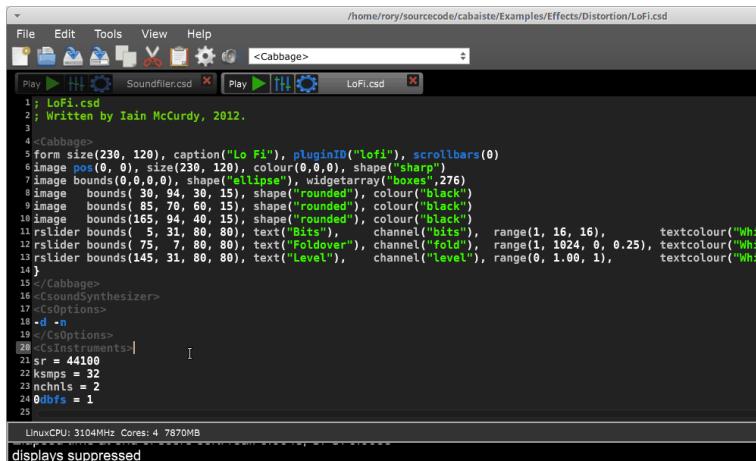
Each and every instrument that you compile in Cabbage will be added to an underlying audio graph. This is true for both Cabbage files, and traditional Csound files. To edit the graph one can launch the Cabbage patcher from the view menu.



Instruments can also be added directly through the graph by right-clicking and adding them from the context menu. The context menu will show all the examples files, along with a selection of files from a user-defined folder. See the section below on *Settings* to learn how to set this folder.



Instruments can also be deleted by right-clicking the instrument node. Users can delete/modify connections by clicking on the connections themselves. They can also connect node by clicking and dragging from an output to an input.

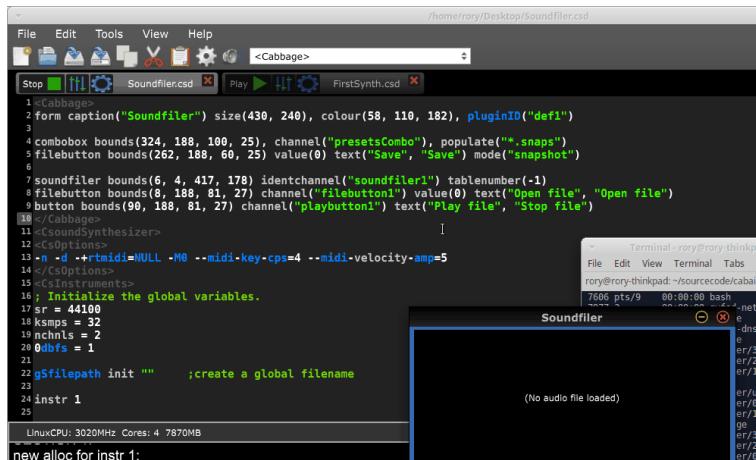


Once an instrument node has been added, Cabbage will automatically open the corresponding code. Each time you update the corresponding source code, the node will also be updated.

As mentioned above, closing a file will not stop it from performing. It is possible to have many instruments running even though their code is not showing. To stop an instrument you must hit the Stop button at the top of its editor, or delete the plugin from the graph.

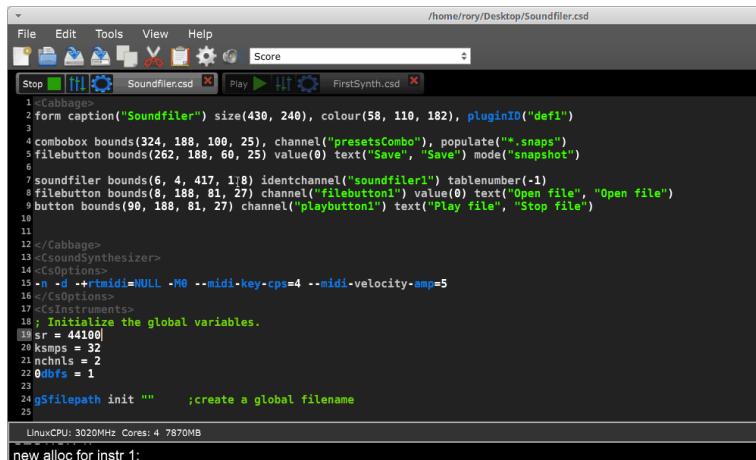
Navigating large source files

It can become quite tricky to navigate very long text files. For this reason Cabbage provides a means of quickly jumping to instrument definitions. It is also possible to create a special ; - Region: tag. Any text that appears after this tag will be appended to the drop-down combo box in the Cabbage tool bar.



Using the code repository

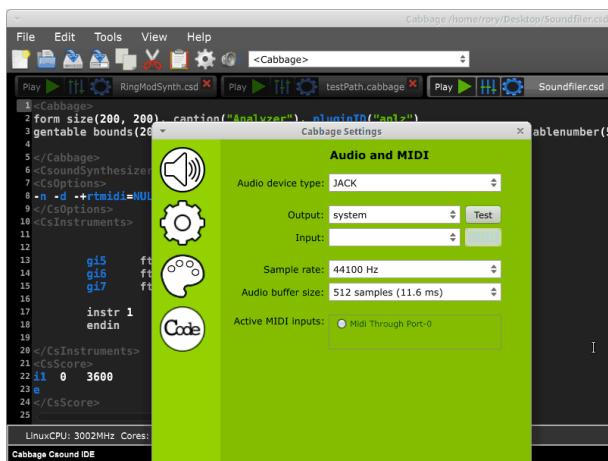
Cabbage provides a quick way of saving and recalling blocks of code. To save code to the repository simple select the code you want, right-click and hit *Add to code repository*. To insert code later from the repository, right-click the place you wish to insert the code and hit *Add from code repository*.



Code can be modified, edited or deleted at a later stage in the Settings dialogue.

Settings

The settings dialogue can be opened by going to the *Edit->Setting* menu command, or pressing the Settings cog in the main toolbar.



Audio and MIDI settings

These settings are used to 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. Note the buffer sizes selected here are only relevant when using the Cabbage IDE. Plugins will have their buffer sizes set by the host. The last known audio and MIDI settings will automatically be saved and recalled for the next session.

Editor

The following settings provide control for various aspects of Cabbage and how it runs its instruments.

- Auto-load: Enabling this will cause Cabbage to automatically load the last files that were open.
- Plugin Window: Enable this checkbox to ensure that the plugin window is always on top and does not disappear behind the main editor when it loses focus.
- Graph Window: Same as above only for the Cabbage patcher window.
- Auto-complete: provides a rough auto-complete of variable names Editor lines to scroll with MouseWheel: Sets the number of lines to jump on each movement of the mouse wheel.

Directories

These directory fields are given default directories that rarely, if ever, need to be changed.

- Csound manual directory: Sets the path to index.html in the Csound help manual. The default directories will be the standard location of the Csound help manual after installing Csound.
- Cabbage manual directory: Sets the path to index.html in the Cabbage help manual.
- Cabbage examples directory: Set the path to the Cabbage examples folder. This should never need to be modified.

- User files directory: Sets the path to a folder containing user files that can be inserted by right-clicking in the patcher. Only files stored in this, and the examples path will be accessible in the Cabbage patcher context menu.

Colours

- Interface: Allows user to set custom colours for various elements of the main graphical interface
- Editor: Allows users to modify the syntax highlighting in the Csound editor
- Console: Allows users to changes various colours in the Csound output console.

Code Repository

This tab shows the various blocks of code that have been saved to the repository. You can edit or delete any of the code blocks. Hit Save/Update to update any changes.

First Synth

As mentioned in the previous section, each Cabbage instrument is defined in a simple text file with a .csd extension. The syntax used to create GUI widgets is quite straightforward and should be provided within special xml-style tags <Cabbage> and </Cabbage> which can appear either above or below Csound's own <CsoundSynthesizer> tags. Each line of Cabbage specific code relates to one GUI widget only. The attributes of each widget are set using different identifiers such as colour(), channel(), size() etc. Where identifiers are not used, Cabbage will use the default values. Long lines can be broken up with a \ placed at the end of a line.

Each and every Cabbage widget has 4 common parameters: position on screen(x, y) and size(width, height). Apart from position and size all other parameters are optional and if left out default values will be assigned. To set widget parameters you will need to use an appropriate identifier after the widget name. More information on the various widgets and identifiers available in Cabbage can be found in the Widget reference section of these docs.

Getting started

Now that the basics of the Csound language have been outlined, let's create a simple instrument. The opcodes used in this simple walk through are vco2, madsr, moogladder and outs.

The vco2 opcode models a voltage controlled oscillator. It provides users with an effective way of generating band-limited waveforms and can be the building blocks of many a synthesiser. Its syntax, taken from the Csound [reference](#) manual, is given below. It is important to become au fait with the way opcodes are presented in the Csound reference manual. It, along with the the

Cabbage widget reference are two documents that you will end up referencing time and time again as you start developing Cabbage instruments.

```
ares vco2 kamp, kcps [, imode] [, kpw] [, kphs] [, inyx]
```

vco2 outputs an a-rate signal and accepts several different input argument. The types of input parameters are given by the first letter in their names. We see above that the kamp argument needs to be k-rate. Square brackets around an input argument means that argument is optional and can be left out. Although not seen above, whenever an input argument start with x, it can be an *i*, *k* or a-rate variable.

kamp determines the amplitude of the signal, while *kcps* set the frequency of the signal. The default type of waveform created by a vco2 is a sawtooth waveform. The simplest instrument that can be written to use a vco2 is given below. The out opcode is used to output an a-rate signal as audio.

```
instr 1
    aOut vco2 1, 440
    out aOut
endin
```

In the traditional Csound context, we would start this instrument using a score statement. We'll learn about score statements later, but because we are building a synthesiser that will be played with a MIDI keyboard, our score section will not be very complex. In fact, it will only contain one line of code. *f0 z* is a special score statement that instructs Csound to listen for events for an extremely long time. Below is the entire source code, including a simple Cabbage section for the instrument presented above.

EXAMPLE 10B01_cabbage_1.csd

```
<Cabbage>
form caption("Untitled") size(400, 300), \
    colour(58, 110, 182), \
    pluginID("def1")
keyboard bounds(8, 158, 381, 95)
</Cabbage>
<CsoundSynthesizer>
<CsOptions>
    -+rtmidi=NULL -M0 -m0d --midi-key-cps=4 --midi-velocity-amp=5
</CsOptions>
<CsInstruments>
    ; Initialize the global variables.
    sr = 44100
    ksmmps = 32
    nchnls = 2
    0dbfs = 1

    ;instrument will be triggered by keyboard widget
    instr 1
        iFreq = p4
        iAmp = p5
        aOut vco2 iAmp, iFreq
        outs aOut, aOut
    endin

</CsInstruments>
<CsScore>
    ;causes Csound to run for about 7000 years...

```

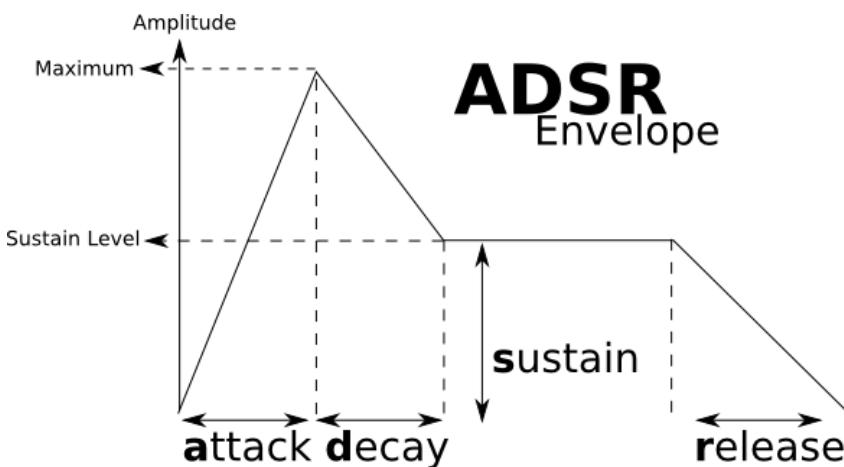
```
f0 z
</CsScore>
</CsoundSynthesizer>
```

You'll notice that the pitch and frequency for the vco2 opcode has been replaced with two *i*-rate variables, *iFreq* and *iAmp*, who in turn get their value from *p5*, and *p4*. *p5* and *p4* are *p*-variables. Their values will be assigned based on incoming MIDI data. If you look at the code in the section you'll see the text '-midi-key-cps=4 -midi-velocity-amp=5'. This instructs Csound to pass the current note's velocity to *p5*, and the current note's frequency, in cycle per second(Hz.) to *p4*. *p4* and *p5* were chosen arbitrarily. *p7*, *p8*, *p*-whatever could have been used, so long as we accessed those same *p* variables in our instrument.

Another important piece of text from the section is the '*M0*'. This tells Csound to send MIDI data to our instruments. Whenever a note is pressed using the on-screen MIDI keyboard, or a hardware keyboard if one is connected, it will trigger instrument 1 to play. Not only will it trigger instrument one to play, it will also pass the current note's amplitude and frequency to our two *p* variables.

Don't be a click head!

If you've tried out the instrument above you'll notice the notes will click each time they sound. To avoid this, an amplitude envelope should be applied to the output signal. The most common envelope is the ubiquitous ADSR envelope. ADSR stands for Attack, Decay, Sustain and Release. The attack, decay and sustain sections are given in seconds as they relate to time values. The sustain value describes the sustain level which kicks in after the attack and decay times have passed. The note's amplitude will rest at this sustain level until it is released.



Csound offers several ADSR envelopes. The simplest one to use, and the one that will work out of the box with MIDI based instruments is `madsr`. Its syntax, as listed in the Csound [reference](#) manual, is given as:

```
kres madsr iatt, idec, islev, irel
```

Note that the inputs to `madsr` are *i*-rate. They cannot change over the duration of a note. There are several places in the instrument code where the output of this opcode can be used. It could be applied directly to the first input argument of the `vco2` opcode, or it can be placed in the line with the `out` opcode. Both are valid approaches.

EXAMPLE 10B02_cabbage_2.csd

```

<Cabbage>
form caption("Untitled") size(400, 300), \
    colour(58, 110, 182), \
    pluginID("def1")
keyboard bounds(8, 158, 381, 95)
</Cabbage>
<CsSynthesizer>
<CsOptions>
-n -d --rtmidi=NULL -m0d --midi-key-cps=4 --midi-velocity-amp=5
</CsOptions>
<CsInstruments>
; Initialize the global variables.
sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

;instrument will be triggered by keyboard widget
instr 1
iFreq = p4
iAmp = p5
iAtt = 0.1
iDec = 0.4
iSus = 0.6
iRel = 0.7
kEnv madsr iAtt, iDec, iSus, iRel
aOut vco2 iAmp, iFreq
outs aOut*kEnv, aOut*kEnv
endin

</CsInstruments>
<CsScore>
;causes Csound to run for about 7000 years...
f0 z
</CsScore>
</CsSynthesizer>

```

Controlling ADSR parameters.

The values of the ADSR parameters can be set using widgets. A typical widget for such control is a slider of some sort. They all behave in more or less the same way. Their job is to send numbers to Csound on a fixed channel. Each widget that is capable of controlling some aspect of an instrument must have a channel set using the channel() identifier. In the following code 4 sliders are created. Each one has a unique channel() set, and they all have the same range. More details on sliders can be found in the widget reference section.

```

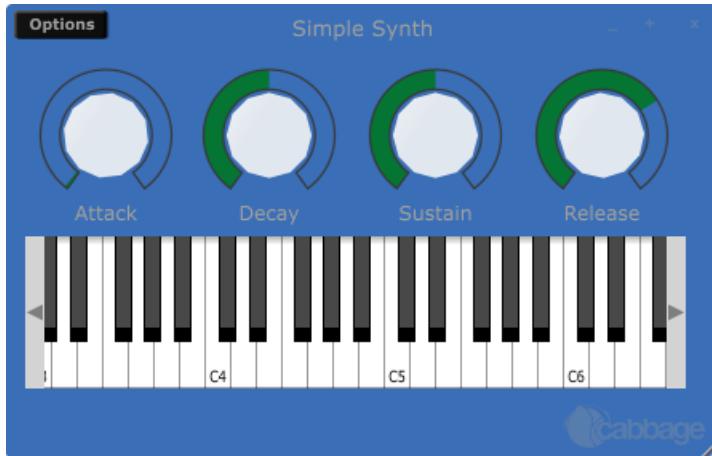
form caption("Simple Synth") \
    size(450, 260), \
    colour(58, 110, 182), \
    pluginID("def1")
keyboard bounds(14, 120, 413, 95)
rslider bounds(12, 14, 105, 101), \
    channel("att"), range(0, 1, 0.01, 1, .01), \
    text("Attack")
rslider bounds(114, 14, 105, 101), channel("dec"), \

```

```

range(0, 1, 0.5, 1, .01), text("Decay")
rslider bounds(218, 14, 105, 101), channel("sus"), \
range(0, 1, 0.5, 1, .01), text("Sustain")
rslider bounds(322, 14, 105, 101), channel("rel"), \
range(0, 1, 0.7, 1, .01), text("Release")

```



It can't be stated enough that each widget responsible for controlling an aspect of your instrument **MUST** have a channel set using the `channel()` identifier. Why? Because Csound can access these channels using its `chnget` opcode. The syntax for `chnget` is very simple:

```
kRes chnget "channel"
```

The `chnget` opcode will create a variable that will contain the current value of the named channel. The rate at which the `chnget` opcode will operate at is determined by the first letter of its output variable. The simple instrument shown in the complete example above can now be modified so that it accesses the values of each of the sliders.

```

instr 1
iFreq = p4
iAmp = p5
iAtt chnget "att"
iDec chnget "dec"
iSus chnget "sus"
iRel chnget "rel"
kEnv madsr iAtt, iDec, iSus, iRel
aOut vco2 iAmp, iFreq
outs aOut*kEnv, aOut*kEnv
endin

```

Every time a user plays a note, the instrument will grab the current value of each slider and use that value to set its ADSR envelop. Note that the `chnget` opcodes listed above all operate at *i*-time only. This is important because the `madsr` opcode expects *i*-rate variable.

Low-pass me the Cabbage please...

ADSR envelopes are often used to control the cut-off frequency of low-pass filters. Low-pass filters block high frequency components of a sound, while letting lower frequencies pass. A popular low-pass filter found in Csound is the moogladder filter which is modeled on the famous filters found in Moog synthesisers. Its syntax, as listed in the Csound reference manual is given as:

```
asig moogladder ain, kcf, kres
```

Its first input argument is an a-rate variable. The next two arguments set the filter cut-off frequency and the amount of resonance to be added to the signal. Both of these can be k-rate variables, thus allowing them to be changed during the note. Cut-off and resonance controls can easily be added to our instrument. To do so we need to add two more sliders to our Cabbage section of code. We'll also need to add two more chnget opcodes and a moogladder to our Csound code. One thing to note about the cut-off slider is that it should be exponential. As the user increases the slider, it should increment in larger and larger steps. We can do this by setting the sliders skew value to .5. More details about this can be found in the slider widget reference page.

EXAMPLE 10B03_cabbage_3.csd

```
<Cabbage>
form caption("Simple Synth") size(450, 220), \
    colour(58, 110, 182), \
    pluginID("def1")
keyboard bounds(14, 88, 413, 95)
rslider bounds(12, 14, 70, 70), \
    channel("att"), \
    range(0, 1, 0.01, 1, .01), \
    text("Attack")
rslider bounds(82, 14, 70, 70), \
    channel("dec"), \
    range(0, 1, 0.5, 1, .01), \
    text("Decay")
rslider bounds(152, 14, 70, 70), \
    channel("sus"), \
    range(0, 1, 0.5, 1, .01), \
    text("Sustain")
rslider bounds(222, 14, 70, 70), \
    channel("rel"), \
    range(0, 1, 0.7, 1, .01), \
    text("Release")
rslider bounds(292, 14, 70, 70), \
    channel("cutoff"), \
    range(0, 22000, 2000, .5, .01), \
    text("Cut-Off")
rslider bounds(360, 14, 70, 70), \
    channel("res"), range(0, 1, 0.7, 1, .01), \
    text("Resonance")
</Cabbage>
<CsoundSynthesizer>
<CsOptions>
-n -d --rtmidi=NULL -M0 -m0d --midi-key-cps=4 --midi-velocity-amp=5
</CsOptions>
<CsInstruments>
; Initialize the global variables.
sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

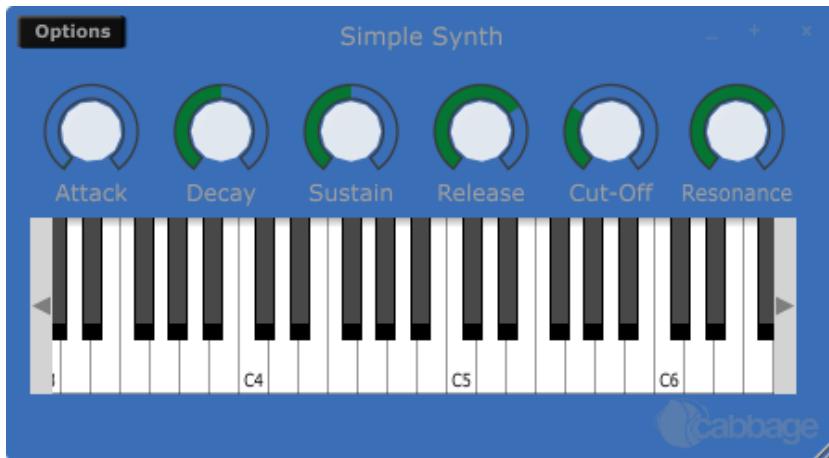
;instrument will be triggered by keyboard widget
instr 1
iFreq = p4
iAmp = p5

iAtt chnget "att"
```

```
iDec chnget "dec"
iSus chnget "sus"
iRel chnget "rel"
kRes chnget "res"
kCutOff chnget "cutoff"

kEnv madsr iAtt, iDec, iSus, iRel
aOut vco2 iAmp, iFreq
aLP moogladder aOut, kCutOff, kRes
outs aLP*kEnv, aLP*kEnv
endin

</CsInstruments>
<CsScore>
;causes Csound to run for about 7000 years...
f0 z
</CsScore>
</CsoundSynthesizer>
```



Sightings of LFOs!

Many synths use some kind of automation to control filter parameters. Sometimes an ADSR is used to control the cut-off frequency of a filter, and in other cases low frequency oscillators, or LFOs are used. As we have already seen how ADSRs work, let's look at implementing an LFO to control the filters cut-off frequency. Csound comes with a standard LFO opcode that provides several different type of waveforms to use. Its syntax, as listed in the Csound [reference](#) manual is given as:

```
kres lfo kamp, kcps [, itype]
```

Type can be one of the following:

- itype = 0 sine
- itype = 1 triangles
- itype = 2 square (bipolar)
- itype = 3 square (unipolar)
- itype = 4 saw-tooth
- itype = 5 saw-tooth(down)

In our example we will use a downward moving saw-tooth wave form. A basic implementation

would look like this.

```
(...)
kEnv madsr iAtt, iDec, iSus, iRel
aOut vco2 iAmp, iFreq
kLFO lfo 1, 1, 5
aLP moogladder aOut, kLFO*kCutOff, kRes
outs aLP*kEnv, aLP*kEnv
endin
(...)
```

The output of the LFO is multiplied by the value of *kCutOff*. The frequency of the LFO is set to 1 which means the cut-off frequency will move from *kCutOff* to 0, once every second. This will create a simple rhythmical effect. Of course it doesn't make much sense to have the frequency fixed at 1. Instead, it is better to give the user control over the frequency using another slider. Finally, an amplitude control slider will also be added, allowing users to control the over amplitude of their synth.

There are many further improvements that could be made to the simple instrument. For example, a second vco2 could be added to create a detune effect which will add some depth to the synth's sound. One could also an ADSR to control the filter envelope, allowing the user an option to switch between modes. If you do end up with something special why not share it on the Cabbage recipes forum!

EXAMPLE 10B04_cabbage_4.csd

```
<Cabbage>
form caption("Simple Synth") size(310, 310), \
  colour(58, 110, 182), \
  pluginID("def1")
keyboard bounds(12, 164, 281, 95)
rslider bounds(12, 14, 70, 70), \
  channel("att"), \
  range(0, 1, 0.01, 1, .01), \
  text("Attack")
rslider bounds(82, 14, 70, 70), \
  channel("dec"), \
  range(0, 1, 0.5, 1, .01), \
  text("Decay")
rslider bounds(152, 14, 70, 70), \
  channel("sus"), \
  range(0, 1, 0.5, 1, .01), \
  text("Sustain")
rslider bounds(222, 14, 70, 70), \
  channel("rel"), \
  range(0, 1, 0.7, 1, .01), \
  text("Release")
rslider bounds(12, 84, 70, 70), \
  channel("cutoff"), \
  range(0, 22000, 2000, .5, .01), \
  text("Cut-Off")
rslider bounds(82, 84, 70, 70), \
  channel("res"), \
  range(0, 1, 0.7, 1, .01), \
  text("Resonance")
rslider bounds(152, 84, 70, 70), \
  channel("LFOFreq"), \
  range(0, 10, 0, 1, .01), \
  text("LFO Freq")
```

```

rslider bounds(222, 84, 70, 70), \
    channel("amp"), \
    range(0, 1, 0.7, 1, .01), \
    text("Amp")
</Cabbage>
<CsOptions>
-n -d --rtmidi=NULL -M0 --midi-key-cps=4 --midi-velocity-amp=5
</CsOptions>
<CsInstruments>
; Initialize the global variables.
sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

;instrument will be triggered by keyboard widget
instr 1
iFreq = p4
iAmp = p5

iAtt chnget "att"
iDec chnget "dec"
iSus chnget "sus"
iRel chnget "rel"
kRes chnget "res"
kCutOff chnget "cutoff"
kLFOFreq chnget "LFOFreq"
kAmp chnget "amp"

KEnv madsr iAtt, iDec, iSus, iRel
aOut vco2 iAmp, iFreq
kLFO lfo 1, kLFOFreq
aLP moogladder aOut, kLFO*kCutOff, kRes
outs kAmp*(aLP*kEnv), kAmp*(aLP*kEnv)
endin

</CsInstruments>
<CsScore>
;causes Csound to run for about 7000 years...
f0 z
</CsScore>
</CsOptions>
</CsInstruments>
</CsoundSynthesizer>

```

A basic Cabbage effect

Cabbage effects are used to process incoming audio. To do this we make use of the signal input opcodes. One can use either ins or inch. To create a new effect click the new file button and select audio effect.

After you have named the new effect Cabage will generate a very simple instrument that takes an incoming stream of audio and outputs directly, without any modification or further processing. In order to do some processing we can add some Csound code the instrument. The code presented below is for a simple reverb unit. We assign the incoming sample data to two variables, i.e., alnL and alnR. We then process the incoming signal through the reverbsc opcode. Some GUI widgets have also been added to provide users with access to various parameter. See the previous section

on creating your first synth if you are not sure about how to add GUI widgets.

Example

EXAMPLE 10B05_cabbage_5.csd

```

<Cabbage>
form size(280, 160), \
    caption("Simple Reverb"), \
    pluginID("plu1")
groupbox bounds(20, 12, 233, 112), text("groupbox")
rslider bounds(32, 40, 68, 70), \
    channel("size"), \
    range(0, 1, .2, 1, 0.001), \
    text("Size"), \
    colour(2, 132, 0, 255),
rslider bounds(102, 40, 68, 70), \
    channel("fco"), \
    range(1, 22000, 10000, 1, 0.001), \
    text("Cut-Off"), \
    colour(2, 132, 0, 255),
rslider bounds(172, 40, 68, 70), \
    channel("gain"), \
    range(0, 1, .5, 1, 0.001), \
    text("Gain"), \
    colour(2, 132, 0, 255),
</Cabbage>
<CsOptions>
<CsOptions>
-n -d
</CsOptions>
<CsInstruments>
sr = 44100
ksmps = 64
nchnls = 2
0dbfs=1

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 1024 10 1
i1 0 z
</CsScore>
</CsOptions>
</CsInstruments>
</CsScore>
</CsoundSynthesizer>
```

The above instrument uses 3 rsliders to control the reverb size(feedback level), the cut-off frequency, and 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 synth and effect instruments, you'll notice that the

synth instrument doesn't use any i-statement. Instead it uses an `f0 z`. This tells Csound to wait for incoming events until the user kills it. Because the instrument is to be controlled via MIDI we don't need to use an i-statement in the score. In the second example we use an i-statement with a long duration so that the instrument runs without stopping for a long time.



Learning More

To learn more about Cabbage, please visit the [Cabbage website](#). There you will find links to more tutorials, video links, and the user forum.

10 C. BLUE

General Overview

Blue is a graphical computer music environment for composition, a versatile front-end to Csound. It is written in Java, platform-independent, and uses Csound as its audio engine. It provides higher level abstractions such as a graphical timeline for composition, GUI-based instruments, score generating SoundObjects like PianoRolls, python scripting, Cmask, Jmask and more. It is available at: <http://blue.kunstmusik.com>

Organization of Tabs and Windows

Blue organizes all tasks that may arise while working with Csound within a single environment. Each task, be it score generation, instrument design, or composition is done in its own window. All the different windows are organized in tabs so that you can flip through easily and access them quickly.

In several places you will find lists and trees: All of your instruments used in a composition are numbered, named and listed in the Orchestra-window.

You will find the same for UDOs (User Defined Opcodes). From this list you may export or import Instruments and UDOs from a library to the piece and vice versa. You may also bind several UDOs to a particular Instrument and export this instrument along with the UDOs it needs.

Editor

Blue holds several windows where you can enter code in an editor-like window. The editor-like windows are found for example in the Orchestra-window, the window to enter global score or the Tables-window to collect all the functions. There you may type in, import or paste text-based information. It gets displayed with syntax highlighting of Csound code.

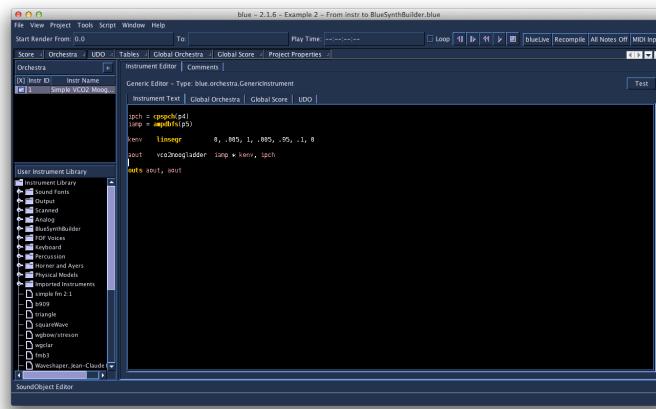


Figure 60.1: The Orchestra-window

The Score Timeline as a Graphical Representation of the Composition

The Score timeline allows for visual organization of all the used *SoundObjects* in a composition.

In the score window, which is the main graphical window that represents the composition, you may arrange the composition by arranging the various *SoundObjects* in the timeline. A *SoundObject* is an object that holds or even generates a certain amount of score-events. *SoundObjects* are the building blocks within Blue's score timeline. *SoundObjects* can be lists of notes, algorithmic generators, python script code, Csound instrument definitions, PianoRolls, Pattern Editors, Tracker interfaces, and more. These *SoundObjects* may be text based or GUI-based as well, depending on their facilities and purposes.

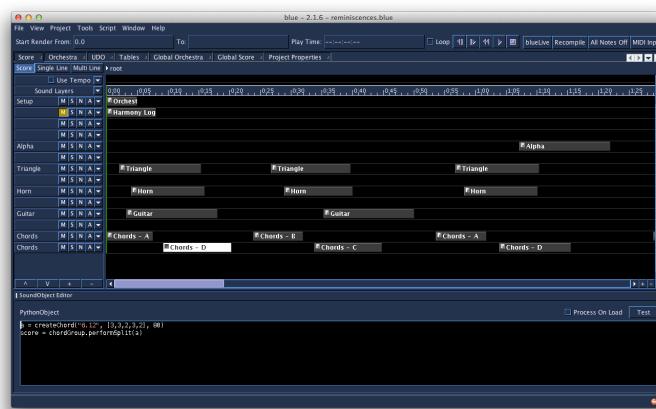


Figure 60.2: Timeline holding several Sound Objects, one selected and opened in the SoundObject editor window

SoundObjects

To enable every kind of music production style and thus every kind of electronic music, blue holds a set of different SoundObjects. SoundObjects in blue can represent many things, whether it is a single sound, a melody, a rhythm, a phrase, a section involving phrases and multiple lines, a gesture, or anything else that is a perceived sound idea.

Just as there are many ways to think about music, each with their own model for describing sound and vocabulary for explaining music, there are a number of different SoundObjects in blue. Each SoundObject in blue is useful for different purposes, with some being more appropriate for expressing certain musical ideas than others. For example, using a scripting object like the *PythonObject* or *RhinoObject* would serve a user who is trying to express a musical idea that may require an algorithmic basis, while the *PianoRoll* would be useful for those interested in notating melodic and harmonic ideas. The variety of different SoundObjects allows for users to choose what tool will be the most appropriate to express their musical ideas.

Since there are many ways to express musical ideas, to fully allow the range of expression that Csound offers, Blue's SoundObjects are capable of generating different things that Csound will use. Although most often they are used for generating Csound SCO text, SoundObjects may also generate ftables, instruments, user-defined opcodes, and everything else that would be needed to express a musical idea in Csound.

Modifying a SoundObject

First, you may set the start time and duration of every SoundObject by hand by typing in precise numbers or drag it more intuitively back and fourth on the timeline. This modifies the position in time of a SoundObject, while stretching it modifies the outer boundaries of it and may even change the density of events it generates inside.

If you want to enter information into a SoundObject, you can open and edit it in a SoundObject editor-window. But there is also a way to modify the "output" of a SoundObject, without having to change its content. The way to do this is using *NoteProcessors*.

By using NoteProcessors, several operations may be applied onto the parameters of a SoundObject. NoteProcessors allow for modifying the SoundObjects score results, i.e. adding 2 to all p4 values, multiplying all p5 values by 6, etc. These NoteProcessors can be chained together to manipulate and modify objects to achieve things like transposition, serial processing of scores, and more.

Finally the SoundObjects may be grouped together and organized in larger-scale hierarchy by combining them to *PolyObjects*. Polyobject are objects, which hold other SoundObjects, and have timelines in themselves. Working within them on their timelines and outside of them on the parent timeline helps organize and understand the concepts of objective time and relative time between different objects.

Instruments with a graphical interface

Instruments and effects with a graphical interface may help to increase musical workflow. Among the instruments with a graphical user interface there are *BlueSynthBuilder* (BSB)-Instruments, *Blue-Effects* and the *Blue Mixer*.

BlueSynthBuilder (BSB)-Instruments

The BlueSynthBuilder (BSB)-Instruments and the BlueEffects work like conventional Csound instruments, but there is an additional opportunity to add and design a GUI that may contain sliders, knobs, textfields, pull-down menus and more. You may convert any conventional Csound Instrument automatically to a BSB-Instrument and then add and design a GUI.



Figure 60.3: The interface of a BSB-Instrument

Blue Mixer

Blue's graphical mixer system allows signals generated by instruments to be mixed together and further processed by Blue Effects. The GUI follows a paradigm commonly found in music sequencers and digital audio workstations.

The mixer UI is divided into channels, sub-channels, and the master channel. Each channel has a fader for applying level adjustments to the channel's signal, as well as bins pre- and post-fader for adding effects. Effects can be created on the mixer, or added from the Effects Library.

Users can modify the values of widgets by manipulating them in real-time, but they can also draw automation curves to compose value changes over time.

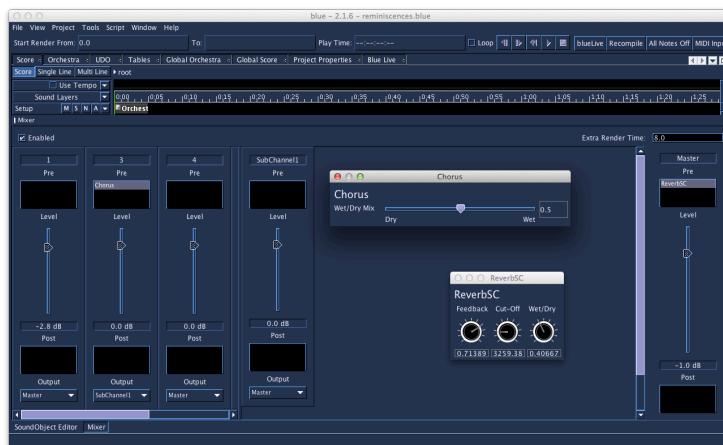


Figure 60.4: The BlueMixer

Automation

For *BSB-Instruments*, *blueMixer* and *blueEffects* it is possible to use Lines and Graphs within the score timeline to enter and edit parameters via a line. In Blue, most widgets in *BlueSynthBuilder* and Effects can have automation enabled. Faders in the Mixer can also be automated.

Editing automation is done in the Score timeline. This is done by first selecting a parameter for automation from the SoundLayer's "A" button's popup menu, then selecting the Single Line mode in the Score for editing individual line values.

Using Multi-Line mode in the score allows the user to select blocks of SoundObjects and automations and move them as a whole to other parts of the Score.

Thus the parameters of these instruments with a GUI may be automatized and controlled via an editable graph in the Score-window.

Libraries

blue features also *libraries for instruments*, *SoundObjects*, *UDOs*, *Effects* (for the *blueMixer*) and the *CodeRepository* for code snippets. All these libraries are organized as lists or trees. Items of the library may be imported to the current composition or exported from it to be used later in other pieces.

The *SoundObject* library allows for instantiating multiple copies of a *SoundObject*, which allows for editing the original object and updating all copies. If NoteProcessors are applied to the instances in the composition representing the general structure of the composition you may edit the content of a *SoundObject* in the library while the structure of the composition remains unchanged. That way you may work on a *SoundObject* while all the occurrences in the composition of that very *SoundObject* are updated automatically according the changes done in the library.

The Orchestra manager organizes instruments and functions as an instrument librarian. There is also an Effects Library and a Library for the UDOs.

Other Features

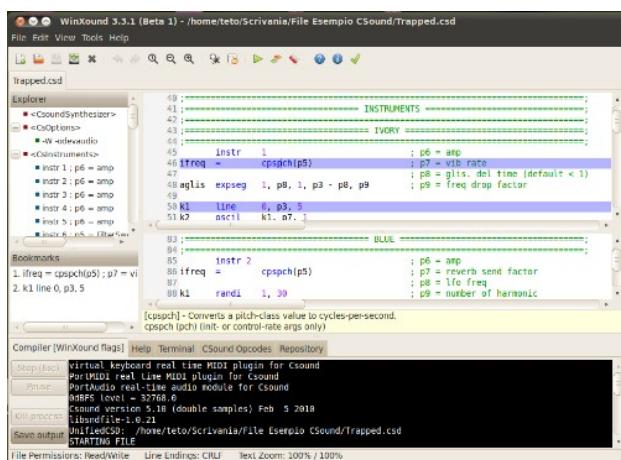
- **blueLive** - work with SoundObjects in realtime to experiment with musical ideas or performance.
- **SoundObject freezing** - frees up CPU cycles by pre-rendering SoundObjects
- **Microtonal support** using scales defined in the Scala scale format, including a microtonal PianoRoll, Tracker, NoteProcessors, and more.

10 D. WINXOUND

It seems that WinXound is not developed any more, but it might still be useful to run older Csound versions; so we keep this chapter for now.

WinXound is a free and open-source Front-End GUI Editor for Csound 6, 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 Csound 6 compiler.

See [WinXound's Website](#) for more information.



10 E. 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 to 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, from 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 essentially the

same information is stored might seem excessive but it is really just to allow flexibility 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 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 our 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.

10 F. WEB BASED CSOUND

Csound to be used in a browser via javascript is currently one of the most exciting developments. This chapter will be rewritten soon. For now, have a look at these introductions and tutorials:

1. Victor Lazzarini has written an extensive tutorial as [Vanilla Guide to Webaudio Csound](#)
2. Steven Yi and Hlöðver Sigurðsson showed an extended tutorial on the ICSC 2022: [Csound on the Web](#)
3. Rory Walsh has contributed a tutorial about his [p5.csound](#), a wrapper to use Csound inside p5.js sketches (the javascript version of Processing: <https://rorywalsh.github.io/p5.Csound/#/>).

The WebAudio Csound Documentation can be found [here](#).

11 A. ANALYSIS

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 command 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)]
```

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](#), [cveral](#), [hetro](#), [lpanal](#) and [pvanal](#).

We will explain in the following paragraphs the background and usage of these five different sound analysis utilities.

atsa

Chapter 05 K gives some background about the *Analysis-Transformation-Synthesis* (ATS) method of spectral resynthesis. It requires the preceding analysis of a sound file. This is the job of the *atsa* utility.

The basic usage is simple:

```
atsa [flags] infilename outfilename
```

where *infilename* is the sound file to be analyzed, and *outfilename* is the .ats file which is written as result of the *atsa* utility.

It can be said that the default values of the various *flags* are reasonable for a first usage. For a refinement of the analysis the [atsa manual page](#) provides all necessary information.

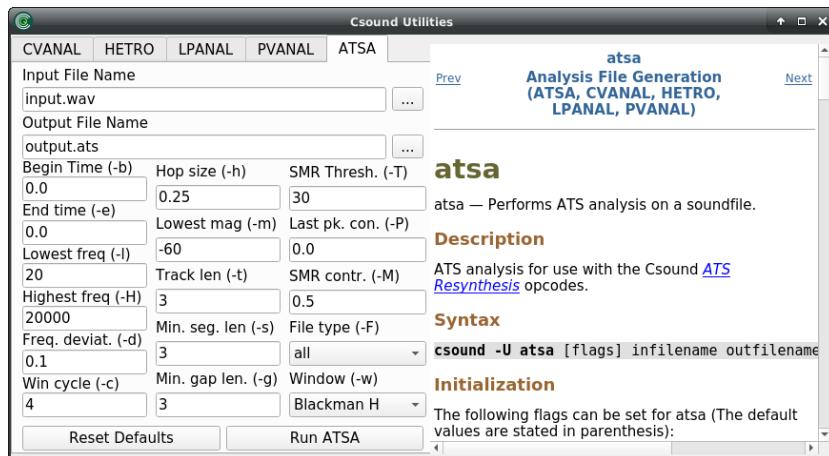


Figure 64.1: ATSA Utility in CsoundQt

cvanal

The *cvanal* utility analyses an impulse response for usage in the old [concolve](#) opcode. Nowadays, convolution in Csound is mostly done with other opcodes which are described in the [Convolution](#) chapter of this book. More information about the *cvanal* utility can be found [here](#) in the Csound Manual.

hetro

The heterodyne filter analysis can be understood as one way of applying the Fourier Transform.¹ Its attempt is to reconstruct a number of partial tracks in a time-breakpoint manner. The breakpoints

¹Cf. Curtis Roads, *The Computer Music Tutorial*, Cambridge MA: MIT Press 1996, 548-549; James Beauchamp, *Analysis, Synthesis and Perception of Musical Sounds*, New York:Springer 2007, 5-12

are measured in milliseconds. Although this utility is originally designed for the usage in the [adsyn](#) opcode, it can be used to get data from any harmonic sound for additive synthesis.

The usage of *hetro* follows the general utility standard:

```
hetro [flags] infilename outfilename
```

But the adjustment of some *flags* is crucial here depending on the desired usage of the analysis:

- **-f begfreq**: This is the estimated frequency of the fundamental. The default is 100 Hz, but it should be adjusted as good as possible to the real fundamental frequency of the input sound.
- **-h partials**: This is the number of partials the utility will analyze and write in the output file. The default number of 10 is quite low and will usually result in a dull sound in the resynthesis.
- **-n brkpts**: This is the number of breakpoints for the analysis. These breakpoints are initially evenly spread over the duration, and then reduced and adjusted by the algorithm. The default number of 256 is reasonable for most usage, but can be massively reduced for some sounds and usages.
- **-m minamp**: The *hetro* utility uses the old Csound amplitude convention where 0 dB is set to 32767. This has to be considered in this option, in which a minimal amplitude is set. Below this amplitude a partial is considered dormant. So the default 64 corresponds to -54 dB; other common values are 128 (-48 dB), 32 (-60 dB) or 0 (no thresholding).

As an example, we start the utility with these parameters:

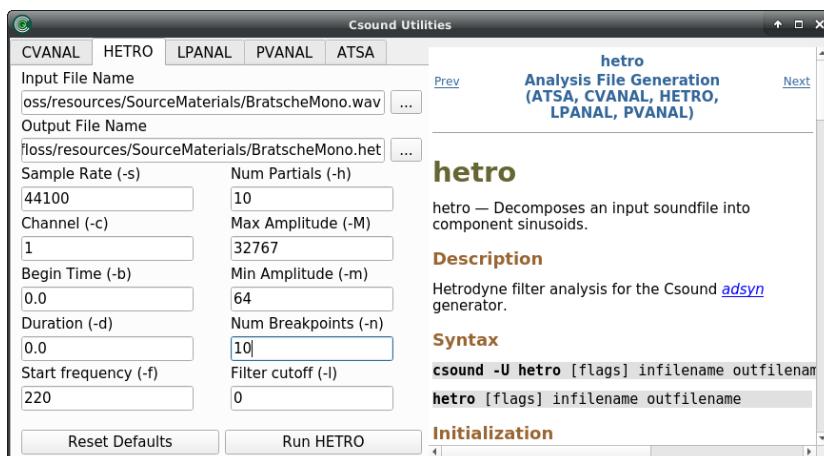


Figure 64.2: HETRO Utility in CsoundQt

This is the output of the analysis in the Csound console:

```
util hetro:
audio sr = 44100, monaural
opening WAV infile resources/SourceMaterials/BratscheMono.wav
analysing 359837 sample frames (8.2 secs)
analyzing harmonic #0
freq estimate 220.0, max found 220.0, rel amp 1039228.8
analyzing harmonic #1
freq estimate 440.0, max found 443.5, rel amp 358754.9
analyzing harmonic #2
freq estimate 660.0, max found 660.0, rel amp 329786.0
analyzing harmonic #3
freq estimate 880.0, max found 880.0, rel amp 253682.2
analyzing harmonic #4
```

```

freq estimate 1100.0, max found 1147.5, rel amp 188708.2
analyzing harmonic #5
freq estimate 1320.0, max found 1320.0, rel amp 51153.9
analyzing harmonic #6
freq estimate 1540.0, max found 1564.8, rel amp 52575.5
analyzing harmonic #7
freq estimate 1760.0, max found 1760.0, rel amp 149709.0
analyzing harmonic #8
freq estimate 1980.0, max found 1980.0, rel amp 162766.8
analyzing harmonic #9
freq estimate 2200.0, max found 2200.0, rel amp 71892.1
scale = 0.013184
harmonic #0:    amp points 10,   frq points 10,   peakamp 13701
harmonic #1:    amp points 10,   frq points 10,   peakamp 4730
harmonic #2:    amp points 10,   frq points 10,   peakamp 4348
harmonic #3:    amp points 10,   frq points 10,   peakamp 3344
harmonic #4:    amp points 9,    frq points 9,    peakamp 2488
harmonic #5:    amp points 10,   frq points 10,   peakamp 674
harmonic #6:    amp points 9,    frq points 9,    peakamp 693
harmonic #7:    amp points 10,   frq points 10,   peakamp 1974
harmonic #8:    amp points 9,    frq points 9,    peakamp 2146
harmonic #9:    amp points 9,    frq points 9,    peakamp 948
wrote 848 bytes to resources/SourceMaterials/BratscheMono.het

```

The file *BratscheMono.het* starts with HETRO 10 as first line, showing that 10 partial track data will follow. The amplitude data lines begin with -1, the frequency data lines begin with -2. This is start and end of the first two lines, slightly formatted to show the breakpoints:

```

-1, 0,0, 815,3409, 1631,11614, 2447,12857, ... , 7343,0, 32767
-2, 0,220, 815,217, 1631,218, 2447,219, ... , 7343,217, 32767

```

After the starting -1 or -2, the time-value pairs are written. Here we have at 0 ms an amplitude of 0 and a frequency of 220. At 815 ms we have amplitude of 3409 and frequency of 217. At 7343 ms, near the end of this file, we have amplitude of 0 and frequency of 217, followed in both cases by 32767 (as additional line ending signifier).

Ipanal

Linear Prediction Coding has been developed for the analysis and resynthesis of speech.² The [Ipanal](#) utility performs the analysis, which will then be used by the [LPC Resynthesis Opcodes](#). The defaults can be seen in the following screenshot:

²Cf. Curtis Roads, The Computer Music Tutorial, Cambridge MA: MIT Press 1996, 200-210

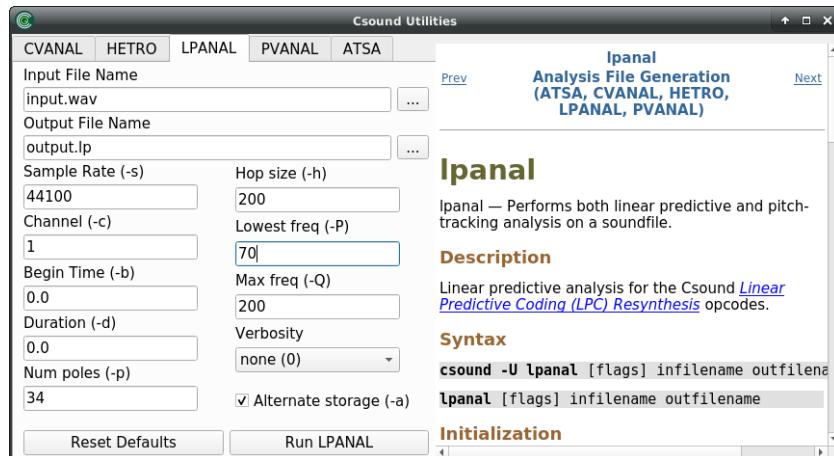


Figure 64.3: LPANAL Utility in CsoundQt

It should be mentioned that in 2020 Victor Lazzarini wrote a bunch of opcodes which apply real-time (streaming) linear prediction analysis. The complement of the old Ipanal utility is the [lpcanal](#) opcode.

pvanal

The [pvanal](#) utility performs a Short-Time Fourier Transform over a sound file. It will produce a [.pxv](#) file which can be used by the old [pv](#)-opcodes. Nowadays the [pvs](#)-opcodes are mostly in use; see chapter [05 I](#) of this book. Nevertheless, the [pvanal](#) utility provides a simple option to perform FFT and write the result in a file.

The main parameter are few; the defaults can be seen here:

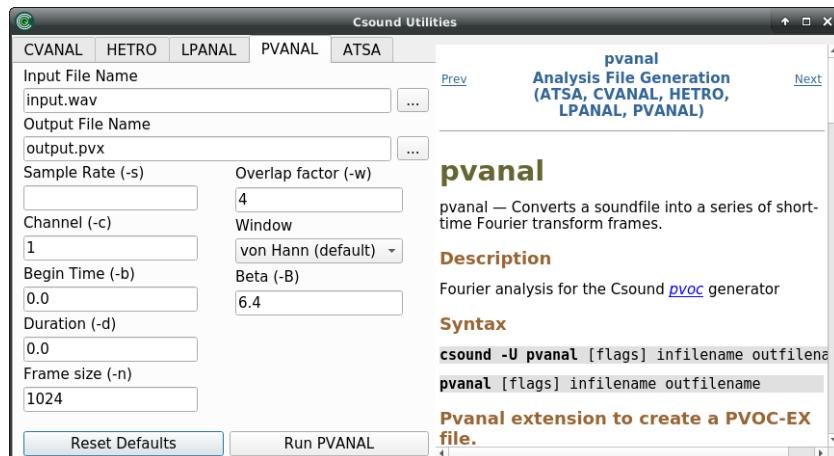


Figure 64.4: PVANAL Utility in CsoundQt

The binary data of a [.pxv](#) file can be converted in a text file via the [pvlook](#) utility.

11 B. FILE INFO AND CONVERSION

sndinfo

The utility *sndinfo* (sound information) provides the user with some information about one or more sound files. *sndinfo* is invoked and provided with a file name:

```
sndinfo ./SourceMaterials/fox.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:  
./SourceMaterials/fox.wav:  
srate 44100, monaural, 16 bit WAV, 2.757 seconds  
(121569 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 gleaned the same information from a GUI based sample editor. We also have the advantage of being able to copy and paste from the terminal window into a .csd file.

File Conversion Utilities

het_import / het_export

The utilities *het_import* and *het_export* are marked as deprecated because the files generated by *hetro* are text files nowadays.

pvlook

The *pvlook* utility shows the output of a STFT analysis files created with *pvanal*. The invocation is:

```
pvlook [flags] infilename
```

As these files contain a big amount of information, the *flags* contain some options to select a range of bins and frames:

- **-bb** and **-eb** set the begin and end of the bin number for the output (defaulting to lowest and highest bin)
- **-bf** and **-ef** set the begin and end of the analysis frames to be printed (defaulting to first and last frame).

If we want to look at the fifth bin only in the frames 100-110 or the file “fox.pvx”, we run:

```
pvlook -bb 5 -eb 5 -bf 100 -ef 110 fox.pvx
```

The output is:

```
util pvlook:
; File name /home/me/csound-manual-git/examples/fox.pvx
; Channels 1
; Word Format float
; Frame Type Amplitude/Frequency
; Source format 16bit
; Window Type Kaiser(0.000000)
; FFT Size 1024
; Window length 2048
; Overlap 256
; Frame align 4104
; Analysis Rate 172.265625
; First Bin Shown: 5
; Number of Bins Shown: 1
; First Frame Shown: 100
; Number of Data Frames Shown: 11

Bin 5 Freqs.
131.728 134.213 135.257 133.603 133.640 131.737 135.581 147.809 176.199
211.347 149.678

Bin 5 Amps.
0.018 0.020 0.020 0.019 0.018 0.017 0.020 0.016 0.002 0.011 0.010
```

pvexport / pvimport

Another method of transforming a .pxv analysis file created by [pvanal](#) is done with the [pv_export](#) utility. It converts the binary file to a text file. After some general information about the source file in the header, each line contains amp-freq pairs of the bins.

The text file can be re-converted to a binary .pxv file with the [pv_import](#) utility.

sdif2ad

The [hetro](#) utility will create an [sdif](#) file if the extension *.sdif* is given for the outfile. This file can be converted by the [sdif2ad](#) utility to a file which can be used by the [adsyn](#) opcode.

src_conv

Sample rate conversion is an everyday's situation in electronic music production. The `src_conv` utility is based on Eric de Castro Lopo's `libsamplerate`. It offers five quality levels where 1 is the worst and 5 the best. The general syntax is here:

```
src_conv [flags] infile
```

The most important flags are:

- `-Q` conversion quality (1-5, default=3)
- `-o` name of the output file (default is `test.wav`)
- `-r` output sample rate
- `-s` or `-3` or `-f` output bit depth (16 (=default) / 24 / 32 bit)
- `-W` for `.wav` as output format (other options are `-A` = aiff and `-J` = ircam)

To convert the sample rate of `fox.wav` in best quality to 48 kHz and writing a 32 bit output file as `best_fox.wav` we write:

```
src_conv -r 48000 -o best_fox.wav -W -Q5 -f fox.wav
```


11 C. MISCELLANEOUS

A final group 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.

12 A. 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.

Though it is written in C, the Csound API uses an object structure. This is achieved through an opaque pointer representing a Csound instance. This opaque pointer is passed as the first argument when an API function is called from the host program.

To use the Csound C API, you have to include `csound.h` in your source file and to link your code with `libcsound64` (or `libcsound` if using the 32 bit version of the library). Here is an example of the `csound` command line application written in C, 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 the 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, using `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 (notice the use of the `-DUSE_DOUBLE` flag to signify that we compile against the 64 bit version of the `csound` library):

```
gcc -DUSE_DOUBLE -o csoundCommand csoundCommand.c -lcsound64
```

The command for building with a 32 bit version of the library would be:

```
gcc -o csoundCommand csoundCommand.c -lcsound
```

Within the C or C++ examples of this chapter, we will use the MYFLT type for the audio samples. Doing so, the same source files can be used for both development (32 bit or 64 bit), the compiler knowing how to interpret MYFLT as double if the macro USE_DOUBLE is defined, or as float if the macro is not defined.

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
```

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 you 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-thread function. 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 {
    int result;          /* result of csoundCompile() */
    CSOUND *csound;      /* instance of csound */
    bool PERF_STATUS;   /* performance 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 ((csoundPerformKsmpls(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/csound.h>

uintptr_t csThread(void *clientData);

typedef struct {
    int result;
    CSOUND *csound;
    int PERF_STATUS;
} userData;

int main(int argc, char *argv[])
{
    int finish;
    void *ThreadID;
    userData *ud;
    ud = (userData *)malloc(sizeof(userData));
    MYFLT *pvalue;
    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 1;
}

/* keep performing until user types a number and presses enter */
scanf("%d", &finish);
ud->PERF_STATUS = 0;
csoundDestroy(ud->csound);
free(ud);
return 0;
}

/* performance thread function */
uintptr_t csThread(void *data)
{
    userData *udata = (userData *)data;
    if (!udata->result) {
        while ((csoundPerformKsmpls(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
kfreq chnget "pitch"
asig oscil 10000, kfreq, 1

```

```
    out    asig
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:

```
# control data (one MYFLT value)
CSOUND_CONTROL_CHANNEL

# audio data (ksmps MYFLT values)
CSOUND_AUDIO_CHANNEL

# string data (MYFLT values with
# enough space to store
# csoundGetChannelDatasize())
CSOUND_STRING_CHANNEL

#characters, including the NULL character at the end of the string)
#and at least one of these:

# when you need Csound to accept incoming values from a host
CSOUND_INPUT_CHANNEL
# when you need Csound to send outgoing values to a host
CSOUND_OUTPUT_CHANNEL
```

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/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;
    ud->csound = csoundCreate(NULL);
    ud->result = csoundCompile(ud->csound, argc, argv);
    if (csoundGetChannelPtr(ud->csound, &pvalue, "pitch",
```

```

    CSOUND_INPUT_CHANNEL | CSOUND_CONTROL_CHANNEL) != 0) {
    printf("csoundGetChannelPtr could not get the \"pitch\" channel");
    return 1;
}
if (!ud->result) {
    ud->PERF_STATUS = 1;
    ThreadID = csoundCreateThread(csThread, (void*)ud);
}
else {
    printf("csoundCompiled returned an error");
    return 1;
}
printf("\nEnter a pitch in Hz(0 to Exit) and type return\n");
while (userInput != 0) {
    *pvalue = (MYFLT)userInput;
    scanf("%d", &userInput);
}
ud->PERF_STATUS = 0;
csoundDestroy(ud->csound);
free(ud);
return 0;
}

/*-----
 * definition of our performance thread function
-----*/
intptr_t csThread(void *data)
{
    userData *udata = (userData *)data;
    if (!udata->result) {
        while ((csoundPerformKsmpls(udata->csound) == 0) &&
               (udata->PERF_STATUS == 1));
        csoundDestroy(udata->csound);
    }
    udata->PERF_STATUS = 0;
    return 1;
}

```

There are several ways of sending data to and from Csound through software buses. They are divided in two categories:

Named Channels with no Callback

This category uses `csoundGetChannelPtr()` to get a pointer to the data of the named channel. There are also six functions to send data to and from a named channel in a thread safe way:

```

MYFLT csoundGetControlChannel(CSOUND *csound, const char *name, int *err)
void csoundSetControlChannel(CSOUND *csound, const char *name, MYFLT val)
void csoundGetAudioChannel(CSOUND *csound, const char *name, MYFLT *samples)
void csoundSetAudioChannel(CSOUND *csound, const char *name, MYFLT *samples)
void csoundGetStringChannel(CSOUND *csound, const char *name, char *string)
void csoundSetStringChannel(CSOUND *csound, const char *name, char *string)

```

The opcodes concerned are `chani`, `chano`, `chnget` and `chnset`. When using numbered channels with `chani` and `chano`, the API sees those channels as named channels, the name being derived from the channel number (i.e. 1 gives "1", 17 gives "17", etc).

There is also a helper function returning the data size of a named channel:

```
int csoundGetChannelDatasize(CSOUND *csound, const char *name)
```

It is particularly useful when dealing with string channels.

Named Channels with Callback

Each time a named channel with callback is used (opcodes invalue, outvalue, chnrecv, and chnsend), the corresponding callback registered by one of those functions will be called:

```
void csoundSetInputChannelCallback(
    CSOUND *csound, channelCallback_t inputChannelCallback
);
void csoundSetOutputChannelCallback(
    CSOUND *csound, channelCallback_t outputChannelCallback
);
```

Other Channel Functions

```
int csoundSetPvsChannel(
    CSOUND *csound, const PVSDATEXT *fin, const char *name
);
int csoundGetPvsChannel(
    CSOUND *csound, PVSDATEXT *fout, const char *name
);

int csoundSetControlChannelHints(
    CSOUND *csound, const char *name, controlChannelHints_t hints
);
int csoundGetControlChannelHints(
    CSOUND *csound, const char *name, controlChannelHints_t *hints
);

int *csoundGetChannelLock(CSOUND *csound, const char *name);
# kills off one or more running instances of an instrument
int csoundKillInstance(
    CSOUND *csound, MYFLT instr, char *instrName, int mode, int allow_release
);

int csoundRegisterKeyboardCallback(
    CSOUND *csound,
    int (*func)(void *userData, void *p, unsigned int type),
    void *userData, unsigned int type
);
# replace csoundSetCallback() and csoundRemoveCallback()
void csoundRemoveKeyboardCallback(
    CSOUND *csound,
    int (*func)(void *, void *, unsigned int)
);
```

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

Csound can call subroutines declared in the host program when some special events occur. This is done through the callback mechanism. One has to declare to Csound the existence of a callback routine using an API setter function. Then when a corresponding event occurs during performance, Csound will call the host callback routine, eventually passing some arguments to it.

The example below shows a very simple command line application allowing the user to rewind the score or to abort the performance. This is achieved by reading characters from the keyboard: 'r' for rewind and 'q' for quit. During performance, Csound executes a loop. Each pass in the loop yields ksmmps audio frames. Using the API `csoundSetYieldCallback()` function, we can tell to Csound to call our own routine after each pass in its internal loop.

The yieldCallback routine must be non-blocking. That's why it is a bit tricky to force the C `getchar` function to be non-blocking. To enter a character, you have to type the character and then hit the return key.

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

int yieldCallback(CSOUND *csound)
{
    int fd, oldstat, dummy;
    char ch;

    fd = fileno(stdin);
    oldstat = fcntl(fd, F_GETFL, dummy);
```

```

fcntl(fd, F_SETFL, oldstat | O_NDELAY);
ch = getc(stdin);
fcntl(fd, F_SETFL, oldstat);
if (ch == -1)
    return 1;
switch (ch) {
case 'r':
    csoundRewindScore(csound);
    break;
case 'q':
    csoundStop(csound);
    break;
}
return 1;
}

int main(int argc, char **argv)
{
    CSOUND *csound = csoundCreate(NULL);
    csoundSetYieldCallback(csound, yieldCallback);
    int result = csoundCompile(csound, argc, argv);
    if (result == 0) {
        result = csoundPerform(csound);
    }
    csoundDestroy(csound);
    return (result >= 0 ? 0 : result);
}

```

The user can also set callback routines for file open events, real-time audio events, real-time MIDI events, message events, keyboards events, graph events, and channel invalue and outvalue events.

CsoundPerformanceThread: A Swiss Knife for the API

Beside the API, Csound provides a helper C++ class to facilitate threading issues: `CsoundPerformanceThread`. This class performs a score in a separate thread, allowing the host program to do its own processing in its main thread during the score performance. The host program will communicate with the `CsoundPerformanceThread` class by sending messages to it, calling `CsoundPerformanceThread` methods. Those messages are queued inside `CsoundPerformanceThread` and are treated in a first in first out (FIFO) manner.

The example below is equivalent to the example in the callback section. But this time, as the characters are read in a different thread, there is no need to have a non-blocking character reading routine.

```

#include <csound/csound.hpp>
#include <csound/csPerfThread.hpp>

#include <iostream>
using namespace std;

int main(int argc, char **argv)
{
    Csound *cs = new Csound();
    int result = cs->Compile(argc, argv);
    if (result == 0) {

```

```
CsoundPerformanceThread *pt = new CsoundPerformanceThread(cs);
pt->Play();
while (pt->GetStatus() == 0) {
    char c = cin.get();
    switch (c) {
    case 'r':
        cs->RewindScore();
        break;
    case 'q':
        pt->Stop();
        pt->Join();
        break;
    }
}
return (result >= 0 ? 0 : result);
}
```

Because CsoundPerformanceThread is not part of the API, we have to link to libcsnd6 to get it working:

```
g++ -DUSE_DOUBLE -o perfThread perfThread.cpp -lcsound64 -lcsnd6
```

When using this class from Python or Java, this is not an issue because the ctcsound.py module and the csnd6.jar package include the API functions and classes, and the CsoundPerformanceThread class as well (see below).

Here is a more complete example which could be the base of a frontal application to run Csound. The host application is modeled through the CsoundSession class which has its own event loop (mainLoop). CsoundSession inherits from the API Csound class and it embeds an object of type CsoundPerformanceThread. Most of the CsoundPerformanceThread class methods are used.

```
#include <csound/csound.hpp>
#include <csound/csPerfThread.hpp>
#include <iostream>
#include <string>

using namespace std;

class CsoundSession : public Csound {
public:
    CsoundSession(string const &csdFileName = "") : Csound() {
        m_pt = NULL;
        m_csd = "";
        if (!csdFileName.empty()) {
            m_csd = csdFileName;
            startThread();
        }
    };
    void startThread() {
        if (Compile((char *)m_csd.c_str()) == 0) {
            m_pt = new CsoundPerformanceThread(this);
            m_pt -> Play();
        }
    };
    void resetSession(string const &csdFileName) {
        if (!csdFileName.empty())
            m_csd = csdFileName;
    }
};
```

```

    if (!m_csd.empty()) {
        stopPerformance();
        startThread();
    }

void stopPerformance() {
    if (m_pt) {
        if (m_pt -> GetStatus() == 0)
            m_pt -> Stop();
        m_pt -> Join();
        m_pt = NULL;
    }
    Reset();
};

void mainLoop() {
    string s;
    bool loop = true;
    while (loop) {
        cout
            << endl
            << "l)oad csd; " +
            "e(vent; " +
            "r(ewind; " +
            "t(oggle pause; " +
            "s(top; " +
            "p(lay; " +
            "q(uit: ";
        char c = cin.get();
        switch (c) {
            case 'l':
                cout << "Enter the name of csd file:";
                cin >> s;
                resetSession(s);
                break;
            case 'e':
                cout << "Enter a score event:";
                cin.ignore(1000, '\n'); // a bit tricky, but well, this is C++!
                getline(cin, s);
                m_pt -> InputMessage(s.c_str());
                break;
            case 'r':
                RewindScore();
                break;
            case 't':
                if (m_pt)
                    m_pt -> TogglePause();
                break;
            case 's':
                stopPerformance();
                break;
            case 'p':
                resetSession("");
                break;
            case 'q':
                if (m_pt) {
                    m_pt -> Stop();
                    m_pt -> Join();
                }
                loop = false;
                break;
        }
    }
}

```

```

        cout << endl;
    }

private:
    string m_csd;
    CsoundPerformanceThread *m_pt;
};

int main(int argc, char **argv) {
    string csdName = "";
    if (argc > 1)
        csdName = argv[1];
    CsoundSession *session = new CsoundSession(csdName);
    session -> mainLoop();
}

```

The application is built with the following command:

```
g++ -o csoundSession csoundSession.cpp -lcsound64 -lcsnd6
```

There are also methods in `CsoundPerformanceThread` for sending score events (`ScoreEvent`), for moving the time pointer (`SetScoreOffsetSeconds`), for setting a callback function (`SetProcessCallback`) to be called at the end of each pass in the process loop, and for flushing the message queue (`FlushMessageQueue`).

As an exercise, the user should complete this example using the methods above and then try to rewrite the example in Python and/or in Java (see below).

Csound API Review

The best source of information is the `csound.h` header file. Let us review some important API functions in a C++ example:

```

#include <csound/csound.hpp>
#include <csound/csPerfThread.hpp>

#include <iostream>
#include <string>
#include <vector>
using namespace std;

string orc1 = "instr 1          \n"
              "idur = p3      \n"
              "iamp = p4      \n"
              "ipch = cpspch(p5) \n"
              "kenv linen iamp, 0.05, idur, 0.1 \n"
              "a1  oscil kenv, ipch \n"
              "      out     a1      \n"
              "endin"; 

string orc2 = "instr 1      \n"
              "idur = p3  \n"
              "iamp = p4  \n"
              "ipch = cpspch(p5) \n"
              "a1  oscili iamp, ipch, 1, 1.5, 1.25 \n"
              "      out     a1      \n"

```

```

        "endin\n";

string orc3 = "instr 1      \n"
    "idur = p3      \n"
    "iamp = p4      \n"
    "ipch = cpspch(p5-1)      \n"
    "kenv  linen   iamp, 0.05, idur, 0.1  \n"
    "asig  rand     0.45      \n"
    "afilt moogvcf2 asig, ipch*4, ipch/(ipch * 1.085)  \n"
    "asig  balance  afilt, asig  \n"
    "       out      kenv*asig      \n"
"endin\n";

string sco1 = "i 1 0 1      0.5 8.00\n"
    "i 1 + 1      0.5 8.04\n"
    "i 1 + 1.5    0.5 8.07\n"
    "i 1 + 0.25   0.5 8.09\n"
    "i 1 + 0.25   0.5 8.11\n"
    "i 1 + 0.5    0.8 9.00\n";

string sco2 = "i 1 0 1      0.5 9.00\n"
    "i 1 + 1      0.5 8.07\n"
    "i 1 + 1      0.5 8.04\n"
    "i 1 + 1      0.5 8.02\n"
    "i 1 + 1      0.5 8.00\n";

string sco3 = "i 1 0 0.5   0.5 8.00\n"
    "i 1 + 0.5   0.5 8.04\n"
    "i 1 + 0.5   0.5 8.00\n"
    "i 1 + 0.5   0.5 8.04\n"
    "i 1 + 0.5   0.5 8.00\n"
    "i 1 + 0.5   0.5 8.04\n"
    "i 1 + 1.0    0.8 8.00\n";

void noMessageCallback(CSOUND *cs, int attr, const char *format,
                      va_list valist) {
    // Do nothing so that Csound will not print any message,
    // leaving a clean console for our app
    return;
}

class CsoundSession : public Csound {
public:
    CsoundSession(vector<string> &orc, vector<string> &sco) : Csound() {
        m_orc = orc;
        m_sco = sco;
        m_pt = NULL;
    };

    void mainLoop() {
        SetMessageCallback(noMessageCallback);
        SetOutput((char *)"dac", NULL, NULL);
        GetParams(&m_csParams);
        m_csParams.sample_rate_override = 48000;
        m_csParams.control_rate_override = 480;
        m_csParams.e0dbfs_override = 1.0;
        // Note that setParams is called before first compilation
        SetParams(&m_csParams);
        if (CompileOrc(orc1.c_str()) == 0) {
            Start(this->GetCsound());
            // Just to be sure...
            cout << GetSr() << ", " << GetKr() << ", ";
            cout << GetNchnls() << ", " << Get0dBFS() << endl;
    }
}

```

```

        m_pt = new CsoundPerformanceThread(this);
        m_pt->Play();
    } else {
        return;
    }

    string s;
    TREE *tree;
    bool loop = true;
    while (loop) {
        cout << endl << "1) 2) 3): orchestras, 4) 5) 6): scores; q(uit: ";
        char c = cin.get();
        cin.ignore(1, '\n');
        switch (c) {
        case '1':
            tree = ParseOrc(m_orc[0].c_str());
            CompileTree(tree);
            DeleteTree(tree);
            break;
        case '2':
            CompileOrc(m_orc[1].c_str());
            break;
        case '3':
            EvalCode(m_orc[2].c_str());
            break;
        case '4':
            ReadScore((char *)m_sco[0].c_str());
            break;
        case '5':
            ReadScore((char *)m_sco[1].c_str());
            break;
        case '6':
            ReadScore((char *)m_sco[2].c_str());
            break;
        case 'q':
            if (m_pt) {
                m_pt->Stop();
                m_pt->Join();
            }
            loop = false;
            break;
        }
    }
};

private:
    CsoundPerformanceThread *m_pt;
    CSOUND_PARAMS m_csParams;
    vector<string> m_orc;
    vector<string> m_sco;
};

int main(int argc, char **argv) {
    vector<string> orc;
    orc.push_back(orc1);
    orc.push_back(orc2);
    orc.push_back(orc3);
    vector<string> sco;
    sco.push_back(sco1);
    sco.push_back(sco2);
    sco.push_back(sco3);
    CsoundSession *session = new CsoundSession(orc, sco);
    session->mainLoop();
}

```

```
}
```

Deprecated Functions

```
csoundQueryInterface()
csoundSetInputValueCallback()
csoundSetOutputValueCallback()
csoundSetChannelIOCallback()
csoundPerformKsmpsAbsolute()
```

are still in the header file but are now deprecated.

Builtin Wrappers

The Csound API has also been wrapped to other languages. Usually Csound is built and distributed including a wrapper for Python and a wrapper for Java.

To use the Python Csound API wrapper, you have to import the `ctcsound` module. The `ctcsound` module is normally installed in the `site-packages` or `dist-packages` directory of your python distribution as a `ctcsound.py` file. Our `csound` command example becomes:

```
import sys
import ctcsound

cs = ctcsound.Csound()
result = cs.compile_(sys.argv)
if result == 0:
    result = cs.perform()
cs.cleanup()
del cs
sys.exit(result)
```

We use a Csound object (remember Python has OOp features). Note the use of the `sys.argv` list to get the program input arguments.

This example would be launched with the following command:

```
python csoundCommand.py myexample.csd
```

To use the Java Csound API wrapper, you have to import the `csnd6` package. The `csnd6` package is located in the `csnd6.jar` archive which has to be known from your Java path. Our `csound` command example becomes:

```
import csnd6.*;

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.argv(), 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.

On our linux system, with csnd.jar located in /usr/local/lib/, our Java Program would be compiled and run with the following commands:

```

javac -cp /usr/local/lib/csnd6.jar CsoundCommand.java
java -cp /usr/local/lib/csnd6.jar:. CsoundCommand

```

There is a drawback using the java wrappers: as it is built during the Csound build, the host system on which Csound will be used must have the same version of Java than the one which were on the system used to build Csound. The mechanism presented in the next section can solve this problem.

Foreign Function Interfaces

Modern programming languages often propose a mechanism called Foreign Function Interface (FFI) which allows the user to write an interface to shared libraries written in C.

Python provides the ctypes module which is used by the ctcsound.py module.

Lua proposes the same functionality through the LuaJIT project. Here is a version of the csound command using LuaJIT FFI:

```

-- This is the wrapper part defining our LuaJIT interface to
-- the Csound API functions that we will use, and a helper function
-- called csoundCompile, which makes a pair of C argc, argv arguments from
-- the script input args and calls the API csoundCompile function
-- This wrapper could be written in a separate file and imported

```

```
-- in the main program.

local ffi = require("ffi")
ffi.cdef[[
typedef void CSOUND;
CSOUND *csoundCreate(void *hostData);
int csoundCompile(CSOUND *, int argc, const char *argv[]);
int csoundPerform(CSOUND *);
void csoundDestroy(CSOUND *);
]]

csoundAPI = ffi.load("csound64.so")

string_array_t = ffi.typeof("const char *[?]" )

function csoundCompile(csound, args)
    local argv = {"dummy"}
    for i, v in ipairs(args) do
        argv[i+1] = v
    end
    local argv = string_array_t(#argv + 1, argv)
    argv[#argv] = nil
    return csoundAPI.csoundCompile(csound, #argv, argv)
end

-- This is the Csound commandline program using the wrapper interface
csound = csoundAPI.csoundCreate(nil)
result = csoundCompile(csound, {...})
if result == 0 then
    csoundAPI.csoundPerform(csound)
end
csoundAPI.csoundDestroy(csound)
```

The FFI package of the Google Go programming language is called cgo. Here is a version of the csound command using cgo:

```
package main

/* This is the wrapper part defining our Go interface to
   the Csound API functions that we will use. It uses the go object
   model building methods that will call the corresponding API functions.
   This wrapper could be written in a separate file and imported
   in the main program.
*/

/*
#cgo CFLAGS: -DUSE_DOUBLE=1
#cgo CFLAGS: -I /usr/local/include
#cgo linux CFLAGS: -DLINUX=1
#cgo LDFLAGS: -lcsound64

#include <csound/csound.h>
*/
import "C"

import (
    "os"
    "unsafe"
)

type CSOUND struct {
    Cs (*C.CSOUND)
}
```

```

type MYFLT float64

func CsoundCreate(hostData unsafe.Pointer) CSOUND {
    var cs (*C.CSOUND)
    if hostData != nil {
        cs = C.csoundCreate(hostData)
    } else {
        cs = C.csoundCreate(nil)
    }
    return CSOUND{cs}
}

func (csound CSOUND) Compile(args []string) int {
    argc := C.int(len(args))
    argv := make([]*C.char, argc)
    for i, arg := range args {
        argv[i] = C.CString(arg)
    }
    result := C.csoundCompile(csound.Cs, argc, &argv[0])
    for _, arg := range argv {
        C.free(unsafe.Pointer(arg))
    }
    return int(result)
}

func (csound CSOUND) Perform() int {
    return int(C.csoundPerform(csound.Cs))
}

func (csound *CSOUND) Destroy() {
    C.csoundDestroy(csound.Cs)
    csound.Cs = nil
}

// This is the Csound commandline program using the wrapper interface
func main() {
    csound := CsoundCreate(nil)
    if result := csound.Compile(os.Args); result == 0 {
        csound.Perform()
    }
    csound.Destroy()
}

```

A complete wrapper to the Csound API written in Go is available at the [Go-Csnd project](#) on github.

The different examples in this section are written for Linux. For other operating systems, some adaptations are needed: for example, for Windows the library name suffix is .dll instead of .so.

The advantage of FFI over Builtin Wrappers is that as long as the signatures of the functions in the interface are the same than the ones in the API, it will work without caring about the version number of the foreign programming language used to write the host program. Moreover, one needs to include in the interface only the functions used in the host program. However a good understanding of the C language low level features is needed to write the helper functions needed to adapt the foreign language data structures to the C pointer system.

References & Links

[Csound API Docs](#)

[Csound API Examples](#)

[ctcsound Docs](#)

Rory Walsh 2006, Developing standalone applications using the Csound Host API and wxWidgets,
[Csound Journal Volume 1 Issue 4 - Summer 2006](#)

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](#)

François Pinot 2014, "Go Binding for Csound6", <https://github.com/fggp/go-csnd>

12 B. PYTHON AND CSOUND

The connection between Csound and Python has a long history. Already in 2002 Maurizio Umberto Puxeddu contributed the [Python Opcodes](#) which allowed the execution of Python code inside Csound. Because of Csound's confession to keep backwards compatibility, this possibility to run **Python inside Csound** will stay as long as the Python code can be executed.

With the Csound API however, which has been explained in the [previous chapter](#), a more flexible and versatile communication between Python and Csound can be established. Now it is **Csound** which runs **inside Python**. This Csound Python API was first generated by [SWIG](#) from the Csound's C API. This version was called `csnd6.py`. In 2015, François Pinot wrote a new version of the Csound Python API. It is based on Python's [ctypes](#) from which its name `ctcsound` (as ctypes csound) originates. This version is better adopted to native Python code and has some useful additional features, as the integration into [Jupyter Notebooks](#) and the new implementation of Andrés Cabrera's *iCsound*.

We will describe in the first part of this chapter some features of using Csound inside Python via `ctcsound`. In the second part we will describe some use cases of the old Python Opcodes in Csound. The possibility to use Python in the score section of a `.csd` file is described in [chapter 14 A](#).

Csound in Python using `ctcsound`

We will focus here on some examples of using `ctcsound` in the Jupyter Notebooks. More can be found in `ctcsound`'s [repository](#).

Installing

Install `ctcsound.py`

The file `ctcsound.py` is distributed with the Csound installer. This version must be used, to avoid incompatibilities between the installed Csound version and the `ctcsound` version. In case it cannot be found, it can be installed from the [Csound sources](#).

To make the `ctcsound.py` working in Python, it must be copied to a directory which Python uses to load external libraries. This folder is usually called `site-packages`. In case there are more than one versions of Python on your computer, make sure you copy to the one which you use to launch the Jupyter Notebooks. On OSX, for instance, when using Anaconda Python, copy `ctcsound.py` from `/Library/Frameworks/CsoundLib64.framework/Versions/6.0/Resources/Python/Current` to `anaconda3/lib/python3.X/site-packages`.

Once this is done, open a Jupyter Notebook and type

```
import ctcsound
```

to see if the installation was successful.

Install csoundmagics

The `csoundmagics` offer some nice features to work with `ctcsound` in the Jupyter Notebooks, including syntax highlighting. They also contain the `iCsound` class. To install the `csoundmagics`, the files at <https://github.com/csound/ctcsound> should be downloaded first. They contain the `cookbook` with a lot of Jupyter Notebook files. The files to be installed can be found in the `csoundmagics` folder in the `cookbook` directory. A description how to install can be found in the `fifth` example of the `cookbook`.

Once this is done, open a Jupyter Notebook and type

```
%load_ext csoundmagics
```

to see if the installation was successful.

iCsound

The `iCsound` class is loaded as part of `ctcsound` with the same command we just used to check the installation:

```
%load_ext csoundmagics
```

After this command has loaded all the libraries, we create an instance of `iCsound`:

```
cs = ICsound()
```

Usually we will get the message: Csound engine started at slot#: 1. Now we can write some simple code and send it to this instance of Csound:

```
orc = """
instr 1
    aOut poscil .2, 400
    out aOut, aOut
endin
"""
cs.sendCode(orc)
cs.sendScore('i 1 0 -1')
```

Csound runs now and plays a sine tone. To turn off the instrument and delete this instance of `iCsound`, we use:

```
cs.sendScore('i -1 0 1')
del cs
```

Some features

As a short survey of some *csoundmagics* and *iCsound* features, we start again with loading the library and creating an instance:

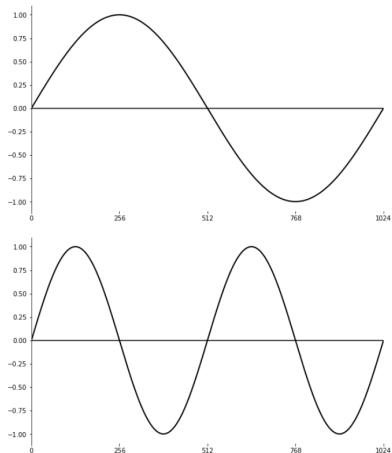
```
%load_ext csoundmagics
cs = ICsound()
```

Now we can use the `%%csound` magics to communicate directly with the running *csound* instance:

```
%%csound
iSine1 ftgen 1, 0, 1024, 10, 1
iSine2 ftgen 2, 0, 1024, 10, 0, 1
```

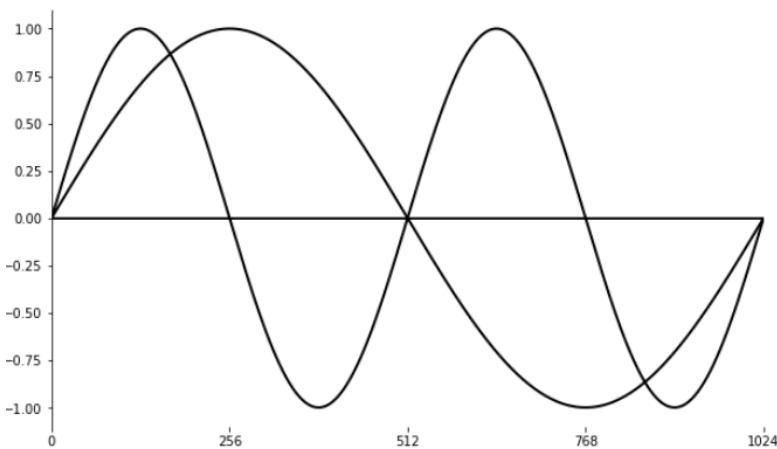
The `plotTable()` method displays now both tables by an internal call to the `matplotlib`:

```
cs.plotTable(1)
cs.plotTable(2)
```



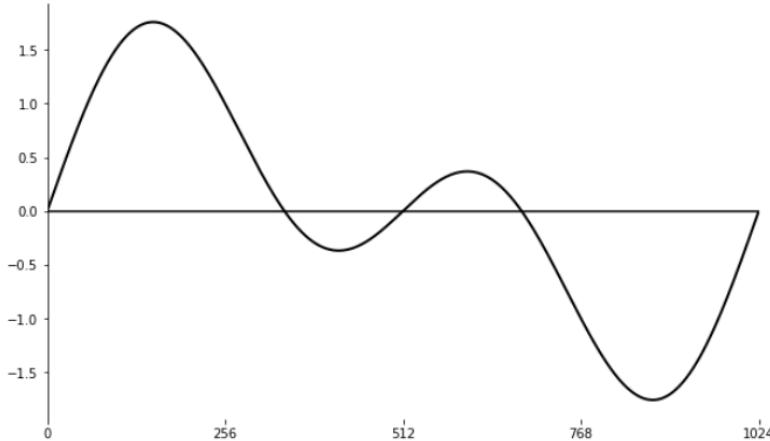
If we want to see both tables in the same plot, we use the option `reuse=True`:

```
cs.plotTable(1)
cs.plotTable(2,reuse=True)
```



Now we add both tables and display the result:

```
both = cs.table(1) + cs.table(2)
cs.fillTable(3,both)
cs.plotTable(3)
```



Building GUI with PySimpleGUI

It is fairly easy to build an own GUI in this way. We use [PySimpleGUI](#) here. After installing, it can be loaded into Python with `import PySimpleGUI as sg`.

Generally spoken, a GUI can have two functions. It can control Csound, for instance start/stop Csound, browse files or change control values. The second function is to use a GUI to display Csound values. We will give one simple example for each case.

GUI controls Csound

This is a code which creates a GUI which lets the user browse an audio file, start and stop playback in a loop, with a volume slider, and deletes the Csound instance when closing. Comments are below.

```
import PySimpleGUI as sg
%load_ext csoundmagics
cs = ICsound()

orc = """
instr 1
    Sfile chnget "file"
    kVol chnget "vol"
    aSound[] diskin Sfile, 1, 0, 1
    kFadeOut linenr 1, .01, 1, .01
    out aSound[0]*kFadeOut*ampdb(kVol), aSound[1]*kFadeOut*ampdb(kVol)
endin
"""
cs.sendCode(orc)

layout = [
    [sg.Text('Select File, then Start/Stop')],
    [sg.FileBrowse(key='FILE', enable_events=True),
     sg.Button('Start'),
     sg.Button('Stop')],
```

```
[sg.Slider(key='VOL',
            range=(-20,6),
            default_value=0,
            orientation='h',
            enable_events=True)]]

window = sg.Window('GUI -> Csound', layout)

while True:
    event, values = window.read()
    if event is None:
        cs.sendScore('i -1 0 1')
        del cs
        break
    cs.setStringChannel('file',values['FILE'])
    cs.setControlChannel('vol',values['VOL'])
    if event is 'Start':
        cs.sendScore('i 1 0 -1')
    if event is 'Stop':
        cs.sendScore('i -1 0 1')

window.close()
```

In the first section, we see the usual way to load the modules, create a Csound instance and send an instrument to it. The *layout* section defines the widgets which will be present in the GUI. The *key* parameter is particularly important here, as this is the way a widget can be identified.

The interaction between the GUI and Csound happens in the *while* loop. Here we send the values of the browse button and the slider to Csound:

```
cs.setStringChannel('file',values['FILE'])
cs.setControlChannel('vol',values['VOL'])
```

Also we start and stop the Csound instrument when the Start/Stop buttons are pressed:

```
if event is 'Start':
    cs.sendScore('i 1 0 -1')
if event is 'Stop':
    cs.sendScore('i -1 0 1')
```

And finally, if the window is being closed, we turn off the instrument, delete the Csound instance and leave the while-loop:

```
if event is None:
    cs.sendScore('i -1 0 1')
    del cs
    break
```

GUI displays Csound values

In the previous example, Csound received values from the GUI via chnget, and the Python code sent these values via *setStringChannel* and *setControlChannel*. Considering now the other way round, we find chnset on the Csound side, and channel on the Python side. The following code shows a moving line in Csound which is displayed by a slider and a text box.

```
import PySimpleGUI as sg
%load_ext csoundmagics
cs = ICsound()
orc = """
seed 0
```

```

instr 1
  kLine randomi -1,1,1,3
  chnset kLine, "line"
endin
"""
cs.sendCode(orc)
cs.sendScore('i 1 0 -1')

layout = [[sg.Slider(range=(-1,1),
                      orientation='h',
                      key='LINE',
                      resolution=.01)],
           [sg.Text(size=(6,1),
                   key='LINET',
                   text_color='black',
                   background_color='white',
                   justification = 'right',
                   font=('Courier',16,'bold'))]
          ]
window = sg.Window('Csound -> GUI', layout)

while True:
    event, values = window.read(timeout=100)
    if event is None:
        cs.sendScore('i -1 0 1')
        del cs
        break
    window['LINE'].update(cs.channel('line')[0])
    window['LINET'].update('%+.3f' % cs.channel('line')[0])
window.close()

```

Python in Csound using the Python Opcodes

The second part of this chapter, discussing the old Python opcodes in Csound, is based on Andrés Cabrera's article *Using Python inside Csound, An introduction to the Python opcodes*.¹ All examples below are to be executed in a Terminal. If using CsoundQt, choose *Run in Term* instead of *Run*. It should be noted that all examples here are using Python2. There is a [plugin](#) to port the Python opcodes to Python 3. There is also an [example](#) how to embed ctcsound in the Python opcodes.

Starting the Python Interpreter and Running Python Code at i-Time: `pyinit` and `pyruni`

To use the Python opcodes inside Csound, you must first start the Python interpreter. This is done using the `pyinit` opcode. The `pyinit` opcode must be put in the header before any other Python opcode is used, otherwise, since the interpreter is not running, all Python opcodes will return an error. You can run any Python code by placing it within quotes as argument to the opcode `pyruni`.

¹Csound Journal Issue 6, Spring 2007: <http://csoundjournal.com/issue6/python0pcodes.html>

This opcode executes the Python code at init time² and can be put in the header. The example below shows a simple csd file which prints the text “Hello Csound world!” to the terminal.

EXAMPLE 12B01_pyinit.csd

```
<CsoundSynthesizer>
<CsOptions>
-ndm0
</CsOptions>
<CsInstruments>

;start python interpreter
pyinit

;run python code at init-time
pyruni "print '*****'"
pyruni "print '*Hello Csound world!*'"
pyruni "print '*****'"

</CsInstruments>
<CsScore>
e 0
</CsScore>
</CsoundSynthesizer>
;Example by Andrés Cabrera and Joachim Heintz
```

Python Variables are usually Global

The Python interpreter maintains its state for the length of the Csound run. This means that any variables declared will be available on all calls to the Python interpreter. In other words, they are global. The code below shows variables ‘c’ and ‘d’ being calculated both in the header (c) and in instrument 2 (d), and that they are available in all instruments (here printed out in instrument 1 and 3). A multi-line string can be written in Csound with the {{...}} delimiters. This can be useful for longer Python code snippets.

EXAMPLE 12B02_python_global.csd

```
<CsoundSynthesizer>
<CsOptions>
-ndm0
</CsOptions>
<CsInstruments>

pyinit

;Execute a python script in the header
pyruni {{ 
a = 2
b = 3
c = a + b
}}
```

²See chapter 03 A for more about init- and k-time in Csound.

```

instr 1 ;print the value of c
prints "Instrument %d reports:\n", p1
pyruni "print 'a + b = c = %d' % c"
endin

instr 2 ;calculate d
prints "Instrument %d calculates the value of d!\n", p1
pyruni "d = c**2"
endin

instr 3 ;print the value of d
prints "Instrument %d reports:\n", p1
pyruni "print 'c squared = d = %d' % d"
endin

</CsInstruments>
<CsScore>
i 1 1 0
i 2 3 0
i 3 5 0
</CsScore>
</CsoundSynthesizer>
;Example by Andrés Cabrera and Joachim Heintz

```

Prints:

```

Instrument 1 reports:
a + b = c = 5
Instrument 2 calculates the value of d!
Instrument 3 reports:
c squared = d = 25

```

Running Python Code at k-Time

Python scripts can also be executed at k-rate using `pyrun`. When `pyrun` is used, the script will be executed on every k-pass for the instrument, which means it will be executed `kr` times per second. The example below shows a simple example of `pyrun`. The number of control cycles per second is set here to 100 via the statement `kr=100`. After setting the value of variable 'a' in the header to zero, instrument 1 runs for one second, thus incrementing the value of 'a' to 100 by the Python statement `a = a + 1`. Instrument 2, starting after the first second, prints the value. Instrument 1 is then called again for another two seconds, so the value of variable 'a' is 300 afterwards. Then instrument 3 is called which performs both, incrementing (in the `+=` short form) and printing, for the first two k-cycles.

EXAMPLE 12B03_pyrun.csd

```

<CsoundSynthesizer>
<CsOptions>
-ndm0
</CsOptions>
<CsInstruments>

kr=100

;start the python interpreter

```

```

pyinit
;set variable a to zero at init-time
pyruni "a = 0"

instr 1
;increment variable a by one in each k-cycle
pyrun "a = a + 1"
endin

instr 2
;print out the state of a at this instrument's initialization
pyruni "print 'instr 2: a = %d' % a"
endin

instr 3
;perform two more increments and print out immediately
kCount timeinstk
pyrun "a += 1"
pyrun "print 'instr 3: a = %d' % a"
;;turnoff after k-cycle number two
if kCount == 2 then
turnoff
endif
endin
</CsInstruments>
<CsScore>
i 1 0 1 ;Adds to a for 1 second
i 2 1 0 ;Prints a
i 1 2 2 ;Adds to a for another two seconds
i 3 4 1 ;Prints a again
</CsScore>
</CsoundSynthesizer>
;Example by Andrés Cabrera and Joachim Heintz

```

Prints:

```

instr 2: a = 100
instr 3: a = 301
instr 3: a = 302

```

Running External Python Scripts: *pyexec*

Csound allows you to run Python script files that exist outside your *csd* file. This is done using *pyexec*. The *pyexec* opcode will run the script indicated, like this:

```
pyexec "/home/python/myscript.py"
```

In this case, the script *myscript.py* will be executed at k-rate. You can give full or relative path names.

There are other versions of the *pyexec* opcode, which run at initialization only (*pyexeci*) and others that include an additional trigger argument (*pyexect*).

Passing values from Python to Csound: *pyeval(i)*

The opcode *pyeval* and its relatives allow you to pass to Csound the value of a Python expression. As usual, the expression is given as a string. So we expect this to work:

Not Working Example!

```
<CsoundSynthesizer>
<CsOptions>
-ndm0
</CsOptions>
<CsInstruments>

pyinit
pyruni "a = 1"
pyruni "b = 2"

instr 1
ival pyevali "a + b"
prints "a + b = %d\n", ival
endin

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

Running this code results in an error with this message:

```
INIT ERROR in instr 1: pyevali: expression must evaluate in a float
```

What happens is that Python has delivered an integer to Csound, which expects a floating-point number. Csound always works with numbers which are not integers (to represent a 1, Csound actually uses 1.0). This is equivalent mathematically, but in computer memory these two numbers are stored in a different way. So what you need to do is tell Python to deliver a floating-point number to Csound. This can be done by Python's *float()* facility. So this code should work:

EXAMPLE 12B04_pyevali.csd

```
<CsoundSynthesizer>
<CsOptions>
-ndm0
</CsOptions>
<CsInstruments>

pyinit
pyruni "a = 1"
pyruni "b = 2"

instr 1
ival pyevali "float(a + b)"
prints "a + b = %d\n", ival
endin

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

```
</CsScore>
</CsoundSynthesizer>
;Example by Andrés Cabrera and Joachim Heintz
```

Prints:

```
a + b = 3
```

Passing Values from Csound to Python: *pyassign(i)*

You can pass values from Csound to Python via the *pyassign* opcodes. This is a very simple example which calculates the cent distance of the proportion 3/2:

EXAMPLE 12B05_pyassigni.csd

```
<CsoundSynthesizer>
<CsOptions>
-ndm0
</CsOptions>
<CsInstruments>

pyinit

instr 1 ;assign 3/2 to the python variable "x"
pyassigni "x", 3/2
endin

instr 2 ;calculate cent distance of this proportion
pyruni {{
from math import log
cent = log(x,2)*1200
print cent
}}
endin

</CsInstruments>
<CsScore>
i 1 0 0
i 2 0 0
</CsScore>
</CsoundSynthesizer>
;example by joachim heintz
```

Unfortunately, you can neither pass strings from Csound to Python via *pyassign*, nor from Python to Csound via *pyeval*. So the interchange between both worlds is actually limited to numbers.

Calling Python Functions with Csound Variables

Apart from reading and setting variables directly with an opcode, you can also call Python functions from Csound and have the function return values directly to Csound. This is the purpose of the *pycall* opcodes. With these opcodes you specify the function to call and the function arguments as arguments to the opcode. You can have the function return values (up to 8 return values

are allowed) directly to Csound i- or k-rate variables. You must choose the appropriate opcode depending on the number of return values from the function, and the Csound rate (i- or k-rate) at which you want to run the Python function. Just add a number from 1 to 8 after to pycall, to select the number of outputs for the opcode. If you just want to execute a function without return value simply use pycall. For example, the function average defined above, can be called directly from Csound using:

```
kave    pycall1 "average", ka, kb
```

The output variable kave, will calculate the average of the variable ka and kb at k-rate.

As you may have noticed, the Python opcodes run at k-rate, but also have i-rate versions if an *i* is added to the opcode name. This is also true for pycall. You can use *pycall1i*, *pycall2i*, etc. if you want the function to be evaluated at instrument initialization, or in the header. The following csd shows a simple usage of the pycall opcodes:

EXAMPLE 12B06_pycall.csd

```
<CsoundSynthesizer>
<CsOptions>
-dnm0
</CsOptions>
<CsInstruments>

pyinit

pyruni {{
def average(a,b):
    ave = (a + b)/2
    return ave
}} ;Define function "average"

instr 1 ;call it
iave    pycall1i "average", p4, p5
prints "a = %i\n", iave
endin

</CsInstruments>
<CsScore>
i 1 0 1 100 200
i 1 1 1 1000 2000
</CsScore>
</CsoundSynthesizer>
;example by andrés cabrera and joachim heintz
```

This csd will print the following output:

```
a = 150
a = 1500
```

Local Instrument Scope

Sometimes you want Python variables to be global, and sometimes you may want Python variables to be local to the instrument instance. This is possible using the local Python opcodes. These opcodes are the same as the ones shown above, but have the prefix pyl instead of py. There are

opcodes like `pylruni`, `pylcall1t` and `pylassigni`, which will behave just like their global counterparts, but they will affect local Python variables only. It is important to have in mind that this locality applies to instrument instances, not instrument numbers. The next example shows both, local and global behaviour.

EXAMPLE 12B07_local_vs_global.csd

```

<CsoundSynthesizer>
<CsOptions>
-dnm0
</CsOptions>
<CsInstruments>
ksmps=32

pyinit

instr 1 ;local python variable 'value'
pylassigni "value", p4
if timeinstk() == 1 then
    kvalue pyleval "value"
    printk "Python variable 'value' in instr %d, instance %d, at start = %d\n",
          0, p1, frac(p1)*10, kvalue
elseif release() == 1 then
    kvalue pyleval "value"
    printk "Python variable 'value' in instr %d, instance %d, at end = %d\n",
          0, p1, frac(p1)*10, kvalue
endif
endin

instr 2 ;global python variable 'value'
pyassigni "value", p4
if timeinstk() == 1 then
    kvalue pyeval "value"
    printk "Python variable 'value' in instr %d, instance %d, at start = %d\n",
          0, p1, frac(p1)*10, kvalue
elseif release() == 1 then
    kvalue pyeval "value"
    printk "Python variable 'value' in instr %d, instance %d, at end = %d\n",
          0, p1, frac(p1)*10, kvalue
endif
endin

</CsInstruments>
<CsScore>
;           p4
i 1.1 0.0  1  100
i 1.2 0.1  1  200
i 1.3 0.2  1  300
i 1.4 0.3  1  400

i 2.1 2.0  1  100
i 2.2 2.1  1  200
i 2.3 2.2  1  300
i 2.4 2.3  1  400
</CsScore>
</CsoundSynthesizer>
;Example by Andrés Cabrera and Joachim Heintz

```

Prints:

```

Python variable 'value' in instr 1, instance 1, at start = 100
Python variable 'value' in instr 1, instance 2, at start = 200
Python variable 'value' in instr 1, instance 3, at start = 300
Python variable 'value' in instr 1, instance 4, at start = 400
Python variable 'value' in instr 1, instance 1, at end = 100
Python variable 'value' in instr 1, instance 2, at end = 200
Python variable 'value' in instr 1, instance 3, at end = 300
Python variable 'value' in instr 1, instance 4, at end = 400
Python variable 'value' in instr 2, instance 1, at start = 100
Python variable 'value' in instr 2, instance 2, at start = 200
Python variable 'value' in instr 2, instance 3, at start = 300
Python variable 'value' in instr 2, instance 4, at start = 400
Python variable 'value' in instr 2, instance 1, at end = 400
Python variable 'value' in instr 2, instance 2, at end = 400
Python variable 'value' in instr 2, instance 3, at end = 400
Python variable 'value' in instr 2, instance 4, at end = 400

```

Both instruments pass the value of the score parameter field p4 to the python variable *value*. The only difference is that instrument 1 does this local (with `pylassign` and `pyleval`) and instrument 2 does it global (with `pyasssign` and `pyeval`). Four instances of instrument 1 are called with 0.1 seconds time offset, for the duration of one second. Printout is done in the first and the last k-cycle of the instrument.

At start, all instruments show that they have set the python variable *value* correctly to the p4 value. This does not change in instrument 1, because the settings are local here. In instrument 2, however, the now *global* python variable *value* is being reset by each of the four instances. At start of the first instance (Csound time 2.0), it is 100. At start of instance 2 (time 2.1), it is 200. It is set to 400 at Csound time 2.3. So at time 2.999, when the first instance finishes its performance, the value is not any more 100, but 400. This is reported in the *at end* printout.

Triggered Versions of Python Opcodes

All of the python opcodes have a “triggered” version, which will only execute when its trigger value is different to 0. The names of these opcodes have a “t” added at the end of them (e.g. `pycallt` or `pylassignt`), and all have an additional parameter called `ktrig` for triggering purposes.

Simple Markov Chains Using the Python Opcodes

Python opcodes can simplify the creation of complex data structures for algorithmic composition. Below you will find a simple example of using the Python opcodes to generate Markov chains for a pentatonic scale. Markov chains require in practice building matrices, which start becoming unwieldy in Csound, especially for more than two dimensions. In Python multi-dimensional matrices can be handled as nested lists very easily. Another advantage is that the size of matrices (or lists) need not be known in advance, since it is not necessary in python to declare the sizes of lists.

EXAMPLE 12B08_markov.csd

```

<CsoundSynthesizer>
<CsOptions>
-odac -dm0
</CsOptions>
<CsInstruments>

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

pyinit

; Python script to define probabilities for each note as lists within a list
; Definition of the get_new_note function which randomly generates a new
; note based on the probabilities of each note occurring.
; Each note list must total 1, or there will be problems!

pyruni {{
c = [0.1, 0.2, 0.05, 0.4, 0.25]
d = [0.4, 0.1, 0.1, 0.2, 0.2]
e = [0.2, 0.35, 0.05, 0.4, 0]
g = [0.7, 0.1, 0.2, 0, 0]
a = [0.1, 0.2, 0.05, 0.4, 0.25]

markov = [c, d, e, g, a]

from random import random, seed

seed()

def get_new_note(previous_note):
    number = random()
    accum = 0
    i = 0
    while accum < number:
        accum = accum + markov[int(previous_note)][int(i)]
        i = i + 1
    return i - 1.0
}}


giSine ftgen 0, 0, 2048, 10, 1 ;sine wave
giPenta ftgen 0, 0, -6, -2, 0, 2, 4, 7, 9 ;Pitch classes for pentatonic scale

instr 1 ;Markov chain reader and note spawner
;p4 = frequency of note generation
;p5 = octave
ioct init p5
klastnote init 0 ;Used to remember last note played
ktrig metro p4 ;generate a trigger with frequency p4
knewnote pycall1t ktrig, "get_new_note", klastnote ;get new note from chain
schedkwhen ktrig, 0, 10, 2, 0, 0.2, knewnote, ioct ;launch note on instr 2
klastnote = knewnote ;New note is now the old note
endin

instr 2 ;A simple sine wave instrument
;p4 = note to be played
;p5 = octave
ioct init p5
ipclass table p4, giPenta
ipclass = ioct + (ipclass / 100) ; Pitch class of the note

```

```
ifreq = cpspch(ipclass) ;Note frequency in Hertz
aenv linen .2, 0.05, p3, 0.1 ;Amplitude envelope
aout poscil aenv, ifreq , giSine ;Simple oscillator
outs aout, aout
endin

</CsInstruments>
<CsScore>
;          frequency of      Octave of
;          note generation    melody
i 1 0 30      3           7
i 1 5 25      6           9
i 1 10 20     7.5         10
i 1 15 15     1           8
</CsScore>
</CsoundSynthesizer>
;Example by Andrés Cabrera
```

12 C. LUA AND CSOUND

The Lua programming language originated in Brazil in 1993. It became a flexible and popular scripting language in the 2000s, especially for people working in Game and app design. The key characteristics of Lua derive from its simplicity, including a fairly small size and good performance. Compared to Python and similar languages, Lua is faster.

So running Csound in Lua is a good option if someone is building an app which needs to be both fast and also simple to code. (Compared to the potential complexity of writing an application in C or C++) Throughout the 2010s, Csounders from all over the world built interesting and rich audio applications with Lua and Csound. For Indie Developers working with the [Löve 2D Engine](#), which is based on Lua, Csound can be a sophisticated option for controlling the sound, as well as creating/controlling the sounds in Csound.

Installing

In order to run Csound Code in Lua, the *luaCsnd6* shared object is needed. Currently (Csound 6.14) it is not available in the Windows and Mac installer. In other words, it requires an own build of Csound on these platforms.

On Linux, the *luaCsnd.so* should be found in */usr/lib*, if you install Csound via the package manager. For own builds of Csound, it should be found in */usr/local/lib* or in your build directory.

Setting the Lua Path

Once the *luaCsnd6.so* is there, it needs to be added to the Lua Path. Either put the *luaCsnd* object in a directory where Lua is searching by default, or add these lines to the configuration file of your shell:

Using Csound build directory: LUA_CPATH="/home/user/csound/buildluaCsnd6.so"

Or using installed Csound (depending on the installation path):

```
LUA_CPATH="/usr/local/lib/luaCsnd6.so"
```

```
LUA_CPATH="/usr/lib/luaCsnd6.so"
```

Running Csound in Lua

This is a test code to be executed in your Lua environment. It will show whether Csound can be run via the Lua Csound API.

```
require "luaCsnd6"

-- Defining our Csound ORC code within a multiline String
orc = [[
sr=44100
ksmps=32
nchnls=2
0dbfs=1
instr 1
aout vco2 0.5, 440
outs aout, aout
endin
]]

-- Defining our Csound SCO code
sco = "i1 0 1"

local c = luaCsnd6.Csound()
c:SetOption("-odac") -- Using SetOption() to configure Csound
-- Note: use only one commandline flag at a time

c:CompileOrc(orc) -- Compile the Csound Orchestra string
c:ReadScore(sco) -- Compile the Csound SCO String
c:Start() -- When compiling from strings, call Start() prior to Perform()
c:Perform() -- Run Csound to completion
c:Stop()
```

Future Applications for Lua and Csound

Concerning the future of Csound and Lua, it might be beneficial in the wake of the AI revolution to look into the possibility of using the Torch Framework in Lua. (Which is comparable to the Tensorflow Framework in Python to create different kinds of Deep Learning Application for Audio, Composition or Sound Synthesis). Although the Torch Framework is not as widely used as Tensorflow in Python, having even a dedicated Audio Library with the magenta projects applications, the DSP capabilities of Lua and Python are quite limited and are lacking the a sophisticated environment that Csound offers.

12 D. CSOUND IN iOS

The first part of this chapter is a guide which aims to introduce and illustrate some of the power that the Csound language offers to iOS Developers. It assumes that the reader has a rudimentary background in Csound, and some experience and understanding of iOS development with either Swift or Objective-C. The most recent Csound iOS SDK can be downloaded on Csound's [download](#) page. Older versions can be found [here](#). The Csound for iOS Manual (Lazzarini, Yi, Boulanger) that ships with the Csound for iOS API is intended to serve as a lighter reference for developers. This guide is distinct from it in that it is intended to be a more thorough, step-by-step approach to learning the API for the first time.

The second part of this chapter is a detailed discussion of the full integration of Csound into the iOS Core Audio system.

I. Features of Csound in iOS

Getting Started

There are a number of ways in which one might begin to learn to work with the Csound for iOS API. Here, to aid in exploring it, we first describe how the project of examples that ships with the API is structured. We then talk about how to go about configuring a new iOS Xcode project to work with Csound from scratch.

Csound for iOS Examples

The Csound for iOS Examples project contains a number of simple examples (in both Objective-C and Swift) of how one might use Csound's synthesis and signal processing capabilities, and the communicative functionality of the API. It is available both in the download bundle or online in the [Csound sources](#).

In the `ViewControllers` group, a number of subgroups exist to organize the various individual examples into a single application. This is done using the Master-Detail application layout paradigm, wherein a set of options, all of them listed in a *master* table, correlates to a single *detail* View-Controller. Familiar examples of this design model, employed by Apple and provided with every

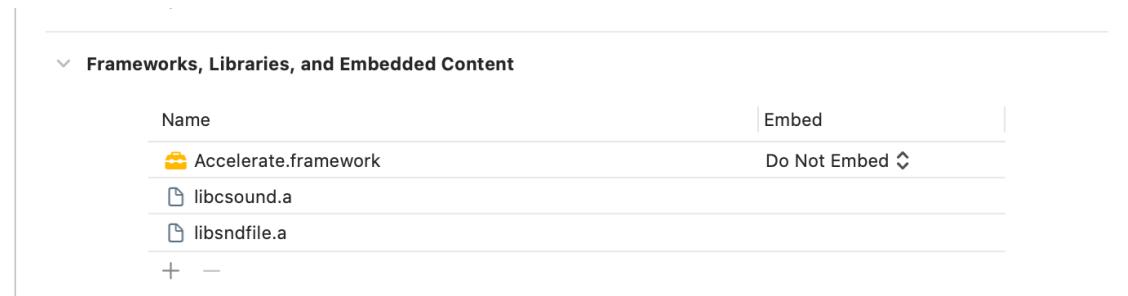
iOS device, are the Settings app, and the Notes app – each of these contains a master table upon which the detail ViewController's content is predicated.

In each of these folders, you will find a unique example showcasing how one might use some of the features of the Csound for iOS API to communicate with Csound to produce and process sounds and make and play music. These are designed to introduce you to these features in a practical setting, and each of these has a unifying theme that informs its content, interactions, and structure.

Adding Csound to Your Project

If you are working in Objective-C, adding Csound for iOS to your project is as simple as dragging the csound-iOS folder into your project. You should select *Groups* rather than *Folder References*, and it is recommended that you elect to copy the csound-iOS folder into your project folder ("Copy Items if Needed").

You need to add Accelerate framework to your project to make it build. In your Xcode project settings select "General", scroll down to Frameworks, Libraries and Embedded Content" and add Accelerate.framework with default option "Do Not Embed".



Once you have successfully added this folder, including the CsoundObj class (the class that manages Csound on iOS) is as simple as adding an import statement to the class. For example:

```
//  
// ViewController.h  
//  
#import "CsoundObj.h"
```

Note that this only makes the CsoundObj class available, which provides an interface for Csound. There are other objects containing UI and CoreMotion bindings, as well as MIDI handling. These are discussed later in this document, and other files will need to be imported in order to access them.

For Swift users, the process is slightly different: you will need to first create a *bridging header*: a .h header file that can import the Objective-C API for access in Swift. The naming convention is *[YourProjectName]-Bridging Header.h* and this file can be easily created manually in Xcode by choosing *File > New > File > Header File* (under Source), and using the naming convention described above. After this, you will need to navigate to your project build settings and add the path to this file (relative to your project's .xcodeproj project file).

Once this is done, navigate to the bridging header in Xcode and add your Objective-C `#import` statements here. For example:

```
//  
// CsoundsiOS_ExampleSwift-Bridging-Header.h
```

```
// CsoundiOS_ExampleSwift
//

#ifndef CsoundiOS_ExampleSwift_Bridging_Header_h
#define CsoundiOS_ExampleSwift_Bridging_Header_h

#import "CsoundObj.h"

#endif /* CsoundiOS_ExampleSwift_Bridging_Header_h */
```

You do not need to add any individual import statements to Swift files, CsoundObj's functionality should be accessible in your .swift files after this process is complete.

Playing a .csd File

The first thing we will do so that we can play a .csd file is add our .csd file to our project. In this case, we will add a simple .csd (in this case named test.csd) that plays a sine tone with a frequency of 440Hz for ten seconds. Sample Csound code for this is:

EXAMPLE 12D01_iOS_simple.csd

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

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

instr 1
asig poscil 0.5 , 440
outs asig , asig
endin

</CsInstruments>
<CsScore>
i1 0 10
</CsScore>
</CsoundSynthesizer>
```

We will add this to our Xcode project by dragging and dropping it into our project's main folder, making sure to select *Copy items if needed* and to add it to our main target.

In order to play this .csd file, we must first create an instance of the CsoundObj class. We can do this by creating a property of our class as follows, in our .h file (for example, in ViewController.h):

```
//
// ViewController.h
// CsoundiOS_ExampleProject
//

#import <UIKit/UIKit.h>
#import "CsoundObj.h"

@interface ViewController : UIViewController
```

```
@property CsoundObj *csound;
@end
```

Once we've done this, we can move over to the corresponding .m file (in this case, ViewController.m) and instantiate our Csound object. Here we will do this in our `viewDidLoad` method, that is called when our ViewController's view loads.

```
//
// ViewController.m
// CsoundiOS_ExampleProject
//

@interface ViewController()
@end
@implementation ViewController

- (void)viewDidLoad {
    [super viewDidLoad];
    // Allocate memory for and initialize a CsoundObj
    self.csound = [[CsoundObj alloc] init];
}

}
```

Note: in order to play our .csd file, we must first get a path to it that we can give Csound. Because part of this path can vary depending on certain factors (for example, the user's native language setting), we cannot pass a static or "hard-coded" path. Instead, we will access the file using the NSBundle class (or 'Bundle' in Swift).

The .csd file is copied as a resource (you can see this under the *Build Phases* tab in your target's settings), and so we will access it and tell Csound to play it as follows:

```
- (void) viewDidLoad {
    [super viewDidLoad];
    self.csound = [[CsoundObj alloc] init];
    // CsoundObj *csound is declared as a property in .h
    NSString *pathToCsd =
        [[NSBundle mainBundle] pathForResource:@"test" ofType:@"csd"];
    [self.csound play:pathToCsd];
}
```

Note that in Swift, this is a little easier and we can simply use:

```
import UIKit
class ViewController: UIViewController {
    var csound = CsoundObj()

    override func viewDidLoad() {
        super.viewDidLoad()
        let pathToCsd = Bundle.main.path(forResource: "test", ofType: "csd")
        self.csound.play(pathToCsd)
    }
}
```

With this, the test.csd file should load and play, and we should hear a ten-second long sine tone shortly after the application runs (i.e. when the main ViewController's main view loads).

Recording and Rendering

Recording (Real-Time)

To record the output of Csound in real-time, instead of the play method, use:

```
// Objective-C
NSURL *docsDirURL = [[[NSFileManager defaultManager]
    URLsForDirectory:NSDocumentDirectory
    inDomains:NSUTFUserDomainMask] lastObject];
NSURL *file = [docsDirURL URLByAppendingPathComponent:@"outputFile.aif"];
NSString *csdPath =
    [[NSBundle mainBundle] pathForResource:@"csdToRecord" ofType:@"csd"];
[csound record:csdPath toURL:file];

// Swift
let docsDirURL =
    FileManager.default.urls(for: .documentDirectory, in: .userDomainMask)[0]
let file = docsDirURL.appendingPathComponent("outFile.aif")
let csdPath = Bundle.main.path(forResource: "csdFile", ofType: "csd")
csound.record(csdPath, to: file)
```

Alternatively, the recordToURL method can be used while Csound is already running to begin recording:

```
// Objective-C
NSURL *docsDirURL = [[[NSFileManager defaultManager]
    URLsForDirectory:NSDocumentDirectory
    inDomains:NSUTFUserDomainMask] lastObject];
NSURL *file = [docsDirURL URLByAppendingPathComponent:@"outputFile.aif"];
[csound recordToURL:file];

// Swift
let docsDirURL =
    FileManager.default.urls(for: .documentDirectory, in: .userDomainMask)[0]
let file = docsDirURL.appendingPathComponent("outFile.aif")
csound.record(to: file)
```

Note: the stopRecording method is used to stop recording without also stopping Csound's real-time rendering.

Rendering (Offline)

You can also render a .csd to an audio file offline. To render Csound offline to disk, use the record:toFile: method, which takes a path rather than a URL as its second argument. For example:

```
// Objective-C
NSString *docsDir = NSSearchPathForDirectoriesInDomains(
    NSDocumentDirectory, NSUserDomainMask, YES)[0];
NSString *file = [docsDir stringByAppendingPathComponent:@"outFile.aif"];
NSString *csdPath =
    [[NSBundle mainBundle] pathForResource:@"csdFile" ofType:@"csd"];
[csound record:csdPath toFile:file];

// Swift
let docsDir = NSSearchPathForDirectoriesInDomains(
    .documentDirectory, .userDomainMask, true
)[0]
```

```
let file = docsDir.appending("/outFile.aif")
let csdPath = Bundle.main.path(forResource: "csdFile", ofType: "csd")
csound.record(csdPath, toFile: file)
```

These demonstrations above save the audio files in the app's documents directory, which allows write access for file and subdirectory storage on iOS. Note that the -W and -A flags behave as usual on iOS: they will decide whether the file rendered is a WAV or an AIFF file. In the event that neither is provided, the latter will be used as a default.

The CsoundUI Class

The CsoundUI class provides for direct bindings between named Csound channels and commonly used objects from the UIKit iOS framework. While it is not necessary to use a CsoundUI object for this communication between iOS and Csound, it can, in many cases, abstract the process of setting up a UI object binding to a single line of code. To initialize a CsoundUI object, we must give it a reference to our Csound object:

```
//Objective-C
CsoundUI *csoundUI = [[CsoundUI alloc] initWithCsoundObj: self.csound];

// Swift
var csoundUI = CsoundUI(csoundObj: csound)
```

Normally, however, these objects are declared as properties rather than locally in methods. As mentioned, CsoundUI uses named channels for communicating to and from Csound. Once set-up, values passed to these named channels are normally accessed through the chnget opcode, for example:

```
instr 1
kfreq chnget "frequency"
asig oscil 0.5 , kfreq
outs asig , asig
endin
```

Conversely, in order to pass values from Csound, the chnset opcode is normally used with two arguments. The first is the variable, and it is followed by the channel name:

```
instr 1
krand randomi 300 , 2000 , 1 , 3
asig poscil 0.5 , krand
outs asig , asig
chnset krand , "randFreq"
endin
```

UIButton Binding

The UIButton binding is predominantly contained within the CsoundButtonBinding class, which CsoundUI uses to create individual button bindings. To add a button binding, use:

```
//Objective-C
[self.csoundUI addButton:self.button forChannelName:@"channelName"];
```

```
// Swift
csoundUI.addButton(forChannelName: "channelName")
```

Where `self.button` is the button you would like to bind to, and the string `channelName` contains the name of the channel referenced by `chnget` in Csound.

The corresponding value in Csound will be equal to 1 while the button is touched, and reset to 0 when it is released. A simple example of how this might be used in Csound, based on the `pvsCross` example by Joachim Heintz, is shown below:

```
instr 1
kpermut chnget "crossToggle"
ain1 soundin "fox .wav"
ain2 soundin "wave .wav"

;fft - analysis of file 1
fftin1 pvsanal ain1 , 1024 , 256 , 1024 , 1
;fft - analysis of file 2
fftin2 pvsanal ain2 , 1024 , 256 , 1024 , 1

if kpermut == 1 then
fcross pvsCross fftin2 , fftin1 , .5 , .5
else
fcross pvsCross fftin1 , fftin2 , .5 , .5
endif

aout pvsynth fcross
out aout
endin
```

UISwitch Binding

The UISwitch binding provides a connection between the UISwitch object and a named channel in Csound. This binding is managed in the `CsoundSwitchBinding` class and you can create a UISwitch binding by using:

```
//Objective-C
[self.csoundUI addSwitch:self.uiSwitch forChannelName:"channelName"];
```



```
// Swift
csoundUI.add(switch, forChannelName: "channelName")
```

As in the case of the UIButton binding, the UISwitch binding provides an on-off state value (1 or 0 respectively) to Csound. Below we use it to turn on or off a simple note generator:

```
; Triggering instrument
instr 1
kTrigFreq randomi gkTrigFreqMin , gkTrigFreqMax , 5
ktrigger metro kTrigFreq
kdur randomh .1 , 2 , 5
konoff chnget "instrToggle"
if konoff == 1 then
schedkwhen ktrigger , 0 , 0 , 2 , 0 , kdur
endif
endin

; Sound generating instrument
instr 2
iamp random 0.03 ,0.5
ipan random 0 , 1
```

```

ipdx random 0 ,13
ipch table ipdx , 2+i( gkscale )
aenv expseg 1 , ( p3 ) , .001
asig oscil iamp * aenv , cpspch(ipch) , 1
outs asig * ipan , asig * (1 - ipan)
endin

```

UILabel Binding

The UILabel binding allows you to display any value from Csound in a UILabel object. This can often be a helpful way of providing feedback to the user. You can add a label binding with:

```

//Objective-C
[self.csoundUI addLabel:self.label forChannelName:@"channelName"];

// Swift
csoundUI.add(label, forChannelName: "channelName")

```

However, in this case the channel is an output channel. To demonstrate, let us add an output channel in Csound to display the frequency of the sound generating instrument's oscillator from the previous example (for UISwitch):

```

; Triggering instrument
instr 1
kTrigFreq randomi gkTrigFreqMin , gkTrigFreqMax , 5
ktrigger metro kTrigFreq
kdur randomh .1 , 2 , 5
konoff chnget " instrToggle "
if konoff == 1 then
  schedkwhen ktrigger , 0 , 0 , 2 , 0 , kdur
endif
endin

; Sound generating instrument
instr 2
iamp random 0.03 ,0.5
ipan random 0 , 1
ipdx random 0 ,13
ipch table ipdx , 2+i( gkscale )
aenv expseg 1 , ( p3 ) , .001
asig oscil iamp * aenv , cpspch(ipch) , 1
chnset cpspch(ipch) , " pitchOut "
outs asig * ipan , asig * (1 - ipan)
endin

```

Note additionally that the desired precision of the value display can be set beforehand using the labelPrecision property of the CsoundUI object. For example:

```
self.csoundUI.labelPrecision = 4;
```

UISlider Binding

The UISlider binding is possibly the most commonly used UI binding - it allows the value of a UISlider object to be passed to Csound whenever it changes. This is set up in the CsoundSliderBinding class and we access it via CsoundUI using:

```

// Objective-C
[self.csoundUI addSlider:self.slider
forChannelName:@"channelName"];

```

```
// Swift
csoundUI.addSlider(forChannelName: "channelName")
```

Note that this restricts you to using the slider's actual value, rather than a rounded version of it or some other variation, which would normally be best suited to a manual value binding, which is addressed later in this guide. An example is provided below of two simple such UISlider-bound values in Csound:

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

instr 1
kfreq chnget "frequency" ; input 0 - 1
kfreq expcurve kfreq , 500 ; exponential distribution
kfreq *= 19980 ; scale to range
kfreq += 20 ;add offset
kamp chnget "amplitude"
kamp port kamp , .001 ; smooth values
asig oscil kamp , kfreq
outs asig , asig
endin
```

Above we get around being restricted to the value of the UISlider by creating an exponential distribution in Csound. Of course we could simply make the minimum and maximum values of the UISlider 20 and 20000 respectively, but that would be a linear distribution by default. In both cases here, the UISlider's range of floating point values is set to be from 0 to 1.

Momentary Button Binding

The momentary button binding is similar to the normal UIButton binding in that it uses a UIButton, however it differs in how it uses this object. The UIButton binding passes a channel value of 1 for as long as the UIButton is held, whereas the momentary button binding sets the channel value to 1 for one Csound k-period (i.e. one k-rate sample). It does this by setting an intermediate value to 1 when the button is touched, passing this to Csound on the next k-cycle, and immediately resetting it to 0 after passing it. This is all occurring predominantly in the `CsoundMomentaryButtonBinding` class, which we access using:

```
// Objective-C
[self.csoundUI
    addMomentaryButton:self.triggerButton
    forChannelName:@"channelName"]
];

// Swift
csoundUI.addMomentaryButton(triggerButton, forChannelName: "channelName")
```

Here's a simple usage example:

```
; Triggering instrument
instr 1
ktrigger chnget "noteTrigger"
schedkwhen ktrigger , 0 , 0 , 2 , 0 , kdur
endin

; Sound generating instrument
instr 2
```

```
iamp random 0.03 ,0.5
ipan random 0 , 1
ipdx random 0 ,13
ipch table ipdx , 2+i( gkscale )
aenv expseg 1 , ( p3 ) , .001
asig oscil iamp * aenv , cpspch(ipch) , 1
chnset cpspch(ipch) , " pitchOut "
outs asig * ipan , asig * (1 - ipan)
endin
```

This replaces the automatic instrument triggering with a manual trigger. Every time the UIButton is touched, a note (by way of an instance of instr 2) will be triggered. This may seem like a more esoteric binding, but there are a variety of potential uses.

The CsoundMotion Class

The CsoundMotion class and its associated bindings allow information to be passed from a device's motion sensors, via the CoreMotion framework, to Csound. As with CsoundUI, it is possible to pass this data indirectly by writing code to mediate between CoreMotion and Csound, but CsoundMotion simplifies and greatly abstracts this process. Subsection 4.4 shows an example of how these values are accessed and might be used in Csound. Note that with CsoundMotion, you do not assign channel names: they are pre-assigned by the relevant objects (e.g. "AccelerometerX").

To declare and initialize a CsoundMotion object, use:

```
// Objective-C
CsoundMotion *csoundMotion =
    [[CsoundMotion alloc] initWithCsoundObj:self.csound];

// Swift
var csoundMotion = CsoundMotion(csoundObj: csound)
```

As with CsoundUI, it may often be advantageous to declare the CsoundMotion object as a property rather than locally.

Accelerometer Binding

The accelerometer binding, implemented in the CsoundAccelerometerBinding class and enabled through the CsoundMotion class, allows access to an iOS device's accelerometer data along its three axes (X, Y, Z). The accelerometer is a device that measures acceleration, aiding in several basic interactions. To enable it, use:

```
// Objective-C
[csoundMotion enableAccelerometer];

// Swift
csoundMotion.enableAccelerometer()
```

Gyroscope Binding

The gyroscope binding, implemented in the CsoundGyroscopeBinding class and enabled through the CsoundMotion class, allows access to an iOS device's gyroscope data along its three axes (X,

Y, Z). The accelerometer is a device that allows rotational velocity to be determined, and together with the accelerometer forms a system with six degrees of freedom. To enable it, use:

```
// Objective-C
[csoundMotion enableGyroscope];

// Swift
csoundMotion.enableGyroscope()
```

Attitude Binding

Finally, the attitude binding, implemented in `CsoundAttitudeBinding` and enabled through `CsoundMotion`, allows access to an iOS device's *attitude data*. As the Apple reference notes, *attitude* refers to the orientation of a body relative to a given frame of reference. `CsoundMotion` enables this as three Euler angle values: *roll*, *pitch*, and *yaw* (rotation around X, Y, and Z respectively). To enable the attitude binding, use:

```
// Objective-C
[csoundMotion enableAttitude];

// Swift
csoundMotion.enableAttitude()
```

Together, these bindings enable control of Csound parameters with device motion in ways that are very simple and straightforward. In the following subsection, an example demonstrating each of the pre-set channel names as well as how some of this information might be used is provided.

Motion Csound Example

Here is an example of a Csound instrument that accesses all of the data, and demonstrates uses for some of it. This example is taken from the [Csound for iOS Examples](#) project.

```
instr 1
kaccelX chnget " accelerometerX "
kaccelY chnget " accelerometerY "
kaccelZ chnget " accelerometerZ "

kgyroX chnget " gyroX "
kgyroY chnget " gyroY "
kgyroZ chnget " gyroZ "

kattRoll chnget " attitudeRoll "
kattPitch chnget " attitudePitch "
kattYaw chnget " attitudeYaw "

kcutoff = 5000 + (4000 * kattYaw )
kresonance = .3 + (.3 * kattRoll )
kpch = 880 + ( kaccelX * 220)
a1 vco2 ( kattPitch + .5) * .2 , kpch
a1 moogladder a1 , kcutoff , kresonance
aL , aR reverbsc a1 , a1 , .72 , 5000
outs aL , aR
endin
```

Each of the channel names is shown here, and each corresponds to what is automatically set in the relevant binding. A little experimenting can be very helpful in determining what to use these values for in your particular application, and of course one is never under any obligation to use

all of them. Regardless, they can be helpful and very straightforward ways to add now-familiar interactions.

The *CsoundBinding* Protocol

The *CsoundBinding* protocol allows you to read values from and write values to Csound using named channels that can be referenced in your .csd file using opcodes like chnget and chnset, as described in the earlier section on CsoundUI. The protocol definition from CsoundObj is:

```
@protocol CsoundBinding <NSObject>
- (void)setup:(CsoundObj*)csoundObj;
@optional
- (void)cleanup;
- (void)updateValuesFromCsound;
- (void)updateValuesToCsound;
@end
```

In order to add a binding object to Csound, use CsoundObj's addBinding method:

```
// Objective-C
[self.csound addBinding:self];

// Swift
csound.addBinding(self)
```

Note that you will need to conform to the CsoundBinding protocol, and implement, at minimum, the required setup method. The CsoundBinding setup method will be called on every object added as a binding, and the remaining methods, marked with the @optional directive will be called on any bindings that implement them.

Channels and Channel Types

Named channels allow us to pass data to and from Csound while it is running. These channels refer to memory locations that we can write to and Csound can read from, and vice-versa. The two most common channel types are: CSOUND_CONTROL_CHANNEL refers to a floating point control channel, normally associated with a k-rate variable in Csound. CSOUND_AUDIO_CHANNEL refers to an array of floating point audio samples of length *ksmps*.

Each of these can be an input or output channel depending on whether values are being passed to or from Csound.

Given below is an example of using named channels in a simplified Csound instrument. The polymorphic *chnget* and *chnset* opcodes are used, and the context here implies that *kverb* received its value from an input control channel named *verbMix*, and that *asig* outputs to an audio channel named *samples*.

```
giSqr ftgen 2, 0, 8192, 10, 1, 0, .33, 0, .2, 0, .14, 0, .11, 0, .09

instr 1
kfreq = p4
kverb chnget " verbMix "
aosc poscil .5, kfreq, 2
arvb reverb aosc, 1.5
asig = (aosc * (1 - kverb)) + (arvb * kverb)
```

```
chnset asig , " samples "
outs asig , asig
endin
```

The section that follows will describe how to set up and pass values to and from this instrument's channels in an iOS application.

The Setup Method

The *setup* method is called before Csound's first performance pass, and this is typically where channel references are created. For example:

```
// Objective-C
// verbPtr and samplesPtr are instance variables of type float*
- (void) setup : (CsoundObj * )csoundObj {
    verbPtr = [csoundObj getInputChannelPtr:@"verbMix"
                                         channelType:CSOUND_CONTROL_CHANNEL];
    samplesPtr = [csoundObj getOutputChannelPtr:@"samples"
                                         channelType:CSOUND_AUDIO_CHANNEL];
}

// Swift
var verbPtr: UnsafeMutablePointer<Float>?
var samplesPtr: UnsafeMutablePointer<Float>?

func setup(_ csoundObj: CsoundObj) {
    verbPtr = csoundObj.getInputChannelPtr(
        "verbMix", channelType: CSOUND_CONTROL_CHANNEL
    )
    samplesPtr = csoundObj.getOutputChannelPtr(
        "samples", channelType: CSOUND_AUDIO_CHANNEL
    )
}
```

The *cleanup* method from *CsoundBinding*, also optional, is intended for use in removing bindings once they are no longer active. This can be done using *CsoundObj*'s *removeBinding* method:

```
// Objective-C
// verbPtr and samplesPtr are instance variables of type float*
-(void) cleanup {
    [self.csound removeBinding:self];
}

// Swift
func cleanup() {
    csound.removeBinding(self)
}
```

Communicating Values To and From Csound

Communicating values to Csound is normally handled in the *updateValuesToCsound* method. This method is called once per performance pass (i.e. at the k-rate). For example:

```
// Objective-C
-(void) updateValuesToCsound {
    *verbPtr = self.verbSlider.value;
}
```

```
// Swift
func updateValuesToCsound() {
    verbPtr?.pointee = verbSlider.value
}
```

This updates the value at a memory location that Csound has already associated with a named channel (in the setup method). This process has essentially replicated the functionality of the CsoundUI API's slider binding. The advantage here is that we could perform any transformation on the slider value, or associate another value (that might not be associated with a UI object) with the channel altogether. To pass values back from Csound, we use the updateValuesFromCsound method.

```
// Objective-C
-(void)updateValuesFromCsound {
    float *samps = samplesPtr;
}
```

Note that in Swift, we have do a little extra work in order to get an array of samples that we can easily index into:

```
// Swift
func updateValuesFromCsound() {
    let samps = samplesPtr?.pointee
    let sampsArray = [Float](UnsafeBufferPointer(start: audioPtr,
        count: Int(csound.getKsmpls())))
}
```

Note also that updateValuesToCsound is called before updateValuesFromCsound during each performance pass, with the Csound engine performance call in between the two.

The *CsoundObjListener* Protocol

The *CsoundObjListener* protocol allows objects in your program to be notified when Csound begins running, and when it completes running. The protocol definition from CsoundObj is:

```
@protocol CsoundObjListener <NSObject>
@optional
- (void)csoundObjStarted:(CsoundObj *)csoundObj;
- (void)csoundObjCompleted:(CsoundObj *)csoundObj;
@end
```

Note that there are no methods that an object is required to adopt in order to conform to this protocol. These methods simply allow an object to elect to be notified when Csound either begins, completes running, or both. Note that these methods are not called on the main thread, so any UI work must be explicitly run on the main thread. For example:

```
// Objective-C
- (void)viewDidLoad {
    [super viewDidLoad];
    [self.csound addListener:self];
}
- (void)csoundObjStarted:(CsoundObj *)csoundObj {
    [self.runningLabel performSelectorOnMainThread:@selector(setText:)
        withObject:@"Csound Running"
        waitUntilDone:NO];
```

```

}

// Swift
override func viewDidLoad() {
    super.viewDidLoad()
    csound.add(self)
}
func csoundObjCompleted(_ csoundObj: CsoundObj) {
    DispatchQueue.main.async { [unowned self] in
        self.runningLabel.text = "Csound Stopped"
    }
}

```

Console Output

Console output from Csound is handled via a callback. You can set the method that handles console info using CsoundObj's setMessageCallbackSelector method, and passing in an appropriate selector, for instance:

```

// Objective-C
[self.csound setMessageCallbackSelector:@selector(printMessage:)];

```



```

// Swift
csound.setMessageCallbackSelector(#selector(printMessage(_:)))

```

An object of type NSValue will be passed in. This object is acting as a wrapper for a C struct of type Message. The definition for Message in *CsoundObj.h* is:

```

typedef struct {
    CSOUND *cs;
    int attr;
    const char *format;
    va_list valist;
} Message;

```

The two fields of interest to us for the purposes of console output are format and valist. The former is a format string, and the latter represents a list of arguments to match its format specifiers.

The process demonstrated in the code examples below can be described as:

1. Declare an instance of a Message struct.
2. Unwrap the NSValue to store its contained Message value at the address of this instance.
3. Declare an empty C string, to act as a buffer.
4. Use the vsnprintf function to populate the buffer with the formatted output string.
5. Wrap this C string in an Objective-C NSString or Swift String.

```

// Objective-C
- (void)printMessage:(NSValue *)infoObj
{
    Message info;
    [infoObj getValue:&info];
    char message[1024];
    vsnprintf(message, 1024, info.format, info.valist);
    NSString *messageStr = [NSString stringWithFormat:@"%@", message];
    NSLog(@"%@", messageStr);
}

```

```
// Swift
func messageCallback(_ infoObj: NSValue) {
    var info = Message()
    infoObj.getValue(&info)
    let message = UnsafeMutablePointer<Int8>.allocate(capacity: 1024)

    vsnprintf(message, 1024, info.format, info.valist)
    let messageStr = String(cString: message)
    print(messageStr)
}
```

In both cases above, we are printing the resulting string objects to Xcode's console. This can be very useful for finding and addressing issues that have to do with Csound or with a .csd file you might be using.

We could also pass the resulting string object around in our program; for example, we could insert the contents of this string object into a UITextView for a simulated Csound console output.

Csound-iOS and MIDI

The Csound iOS API provides two possible ways of passing MIDI information to Csound. CsoundObj can receive MIDI events from CoreMIDI directly. By default, this functionality is disabled, but setting CsoundObj's midiInEnabled property to true (or YES on Objective-C) enables it. This must, however be done before Csound is run.

Note that you must also set the appropriate command-line flag in your csd, under CsOptions. For example, -M0. Additionally, the MIDI device must be connected before the application is started.

MidiWidgetsManager

The second way that is provided to communicate MIDI information to Csound is indirect, via the use of UI widgets and CsoundUI. In this case, the MidiWidgetsManager uses a MidiWidgetsWrapper to connect a MIDI CC to a UI object, and then CsoundUI can be used to connect this UI object's value to a named channel in Csound. For instance:

```
// Objective-C
MidiWidgetsManager *widgetsManager = [[MidiWidgetsManager alloc] init];
[widgetsManager addSlider:self.cutoffSlider forControllerNumber:5];
[csoundUI addSlider:self.cutoffSlider forChannelName:@"cutoff"];
[widgetsManager openMidiIn];

// Swift
let widgetsManager = MidiWidgetsManager()
widgetsManager.add(cutoffSlider, forControllerNumber: 5)
csoundUI?.add(cutoffSlider, forChannelName: "cutoff")
widgetsManager.openMidiIn()
```

An advantage of this variant is that MIDI connections to the UI widgets are active even when Csound is not running, so visual feedback can still be provided, for example. At the time of writing, support is only built-in for UISliders.

Other Functionality

This section describes a few methods of CsoundObj that are potentially helpful for more complex applications.

getCsound

```
(CSOUND *)getCsound;
```

The getCsound method returns a pointer to a struct of type CSOUND, the underlying Csound instance in the C API that the iOS API wraps. Because the iOS API only wraps the most commonly needed functionality from the Csound C API, this method can be helpful for accessing it directly without needing to modify the Csound iOS API to do so.

Note that this returns an opaque pointer because the declaration of this struct type is not directly accessible. This should, however, still allow you to pass it into Csound C API functions in either Objective-C or Swift if you would like to access them.

getAudioUnit

```
(AudioUnit *)getAudioUnit;
```

The getAudioUnit method returns a pointer to a CsoundObj instance's I/O AudioUnit, which provides audio input and output to Csound from iOS.

This can have several potential purposes. As a simple example, you can use the AudioOutputUnitStop() function with the returned value's pointee to pause rendering, and AudioOutputUnitStart() to resume.

updateOrchestra

```
(void)updateOrchestra:(NSString *)orchestraString;
```

The updateOrchestra method allows you to supply a new Csound orchestra as a string.

Other

Additionally, getKsmmps returns the current *ksmps* value, and getNumChannels returns the number of audio channels in use by the current Csound instance. These both act directly as wrappers to Csound C API functions.

II. How to Fully Integrate Csound into Apples iOS CoreAudio

In the second part of this chapter we will study some strategies for best integration of Csound with CoreAudio, in order to aid the development of iOS applications. There are some important issues

to be considered for a professional audio application, such as the native Inter-App Audio routing, buffer frame etc. We will examine in detail the relevant code (Csound and Objective-C) taken from few audio apps based on Csound. We will learn how to manage the buffer frame and sampling rate; how to draw a Waveform in CoreGraphics from a Csound GEN Routine; how to write a Csound GEN table and much more.

Getting Started

The development of professional audio applications involves to consider some important aspects of iOS in order to maximize the compatibility and versatility of the app.

The approach should focus on these five important points:

1. Implement Background Audio
2. Switch on/off your Audio Engine
3. Implement Core MIDI
4. Do not waste System Resources
5. Set up the Sampling Rate and Buffer Frame according to applications running in the system.

The code for the User Interface (UI) is written in Objective-C, whilst the Csound API (i.e. Application Programming Interface) is written in C. This duality allows us to understand in detail the interaction between both. As we will see in the next section, the control unit is based on the *callback* mechanism rather than the *pull* mechanism.

No Objective-C code was deliberately written in the C audio *callback*, since it is not recommended as well it is not recommended to allocate/de-allocate memory.

Since often we will refer to the tutorials (XCode projects), it would be useful to have on hand the xCode environment. These files can be downloaded [here](#).

Setup for an Audio App

In the first XCode project (*01_csSetup*) we configure a Single View Application to work with audio and Csound. The project dependencies are only **libcsound.a** and **libsndfile.a** with the header (.h) files and *CsoundMIDI.h* as well as *CsoundMIDI.m*.

The code of *initializeAudio* function will enable the input/output audio:

```
- (void)initializeAudio {
    /* Audio Session handler */
    AVAudioSession *session = [AVAudioSession sharedInstance];

    NSError *error = nil;
    BOOL success = NO;

    success = [session setCategory:AVAudioSessionCategoryPlayAndRecord
                    withOptions:(AVAudioSessionCategoryOptionMixWithOthers |
                               AVAudioSessionCategoryOptionDefaultToSpeaker)
                    error:&error];
}
```

```

success = [session setActive:YES error:&error];

/* Sets Interruption Listener */
[[NSNotificationCenter defaultCenter]
    addObserver:self
        selector:@selector(InterruptionListener:)
            name:AVAudioSessionInterruptionNotification
        object:session];

AudioComponentDescription defaultOutputDescription;
defaultOutputDescription.componentType = kAudioUnitType_Output;
defaultOutputDescription.componentSubType = kAudioUnitSubType_RemoteIO;
defaultOutputDescription.componentManufacturer =
    kAudioUnitManufacturer_Apple;
defaultOutputDescription.componentFlags = 0;
defaultOutputDescription.componentFlagsMask = 0;

// Get the default playback output unit
AudioComponent HALOutput =
    AudioComponentFindNext(NULL, &defaultOutputDescription);
NSAssert(HALOutput, @"Can't find default output");

// Create a new unit based on this that we will use for output
err = AudioComponentInstanceNew(HALOutput, &csAUHAL);

// Enable IO for recording
UInt32 flag = 1;
err = AudioUnitSetProperty(csAUHAL, kAudioOutputUnitProperty_EnableIO,
                           kAudioUnitScope_Input, 1, &flag, sizeof(flag));
// Enable IO for playback
err = AudioUnitSetProperty(csAUHAL, kAudioOutputUnitProperty_EnableIO,
                           kAudioUnitScope_Output, 0, &flag, sizeof(flag));

err = AudioUnitInitialize(csAUHAL);

/* AUDIOPORT and IAA */
[self initializeAB_IAA];
}

```

This code is common to many audio applications, easily available online or from the Apple documentation. Basically, we setup the app as *PlayAndRecord* category, then we create the AudioUnit. The *PlayAndRecord* category allows receiving audio from the system and simultaneously produce audio.

IMPORTANT:

For proper operation with Audiobus (AB) and Inter-App Audio (IAA), we must instantiate and initialize one Audio Unit (AU) only once for the entire life cycle of the app. To destroy and recreate the AU would involve to require more memory (for each instance). If the app is connected to IAA or AB it will stop responding and we will experience unpredictable behavior, which may lead to an unexpected crash.

Actually there is no way to tell at runtime AB and / or IAA that the AU address has changed. The *InitializeAudio* function should be called only once, unlike the run/stop functions of Csound.

These aspects will become more clear in the following paragraphs.

Initialize Csound and Communicate with it

The ***AudioDSP.m*** class implements the entire audio structure and manages the user interface interaction with Csound. *AudioDSP* is a subclass of *NSObject* that is instantiated on the ***Main.storyboard***. A reference to this class on the storyboard greatly facilitates connections between the GUI (*IBOutlet* and *IBAction*) and the DSP i.e. Csound.

As we will see in the next section, all links are established graphically with the *Interface Builder*.

The main CSOUND structure is allocated in the *AudioDSP* constructor and initializes the audio system. This approach foresees that the *_cs* (CSOUND*) class variable persists for the entire life cycle of the app. As mentioned, the *initializeAudio* function should be called only once.

```
- (instancetype)init {
    self = [super init];
    if (self) {

        // Creates an instance of Csound
        _cs = csoundCreate(NULL);

        // Setup CoreAudio
        [self initializeAudio];
    }
    return self;
}
```

Since we have the CSOUND structure allocated and the CoreAudio properly configured, we can manage Csound asynchronously.

The main purpose of this simple example is to study how the user interface (UI) interacts with Csound. All connections have been established and managed graphically through the Interface Builder.

The *UISwitch* object is connected with the *toggleOnOff*, which has the task to toggle on/off Csound in this way:

```
- (IBAction)toggleOnOff:(id)component {
    UISwitch *uiswitch = (UISwitch *)component;

    if (uiswitch.on) {

        NSString *tempFile =
            [[NSBundle mainBundle] pathForResource:@"test" ofType:@".csd"];

        [self stopCsound];
        [self startCsound:tempFile];

    } else {
        [self stopCsound];
    }
}
```

In the example the *test.cs* is performed which implements a simple sinusoidal oscillator. The frequency of the oscillator is controlled by the *UISlider* object. This is linked with the *sliderAction* callback.

As anticipated, the mechanism adopted is driven by events (callback). This means that the function associated with the event is called only when the user performs an action on the UI slider.

In this case the action is of type *Value Changed*. The Apple documentation concerning the `UIControl` framework should be consulted, for further clarification in this regard.

```
- (IBAction)sliderAction:(id)sender {
    UISlider *sld = sender;
    if (!cs || !running)
        return;

    NSString *channelName = @“freq”;
    float *value;
    csoundGetChannelPtr(
        _cs, &value,
        [channelName cStringUsingEncoding:NSUTF8StringEncoding],
        CSOUND_CONTROL_CHANNEL | CSOUND_INPUT_CHANNEL
    );
    *value = (float)sld.value;
}
```

As we can see, we get the pointer through `csoundGetChannelPtr`, this is relative to incoming control signals. From the point of view of Csound, the signals in the input (CSOUND_INPUT_CHANNEL) are sampled from the software bus via `chnget`, while in the output (CSOUND_OUTPUT_CHANNEL) `chnset` is used.

The allocation is done by dereferencing the pointer in this way:

```
*value = (float) sld.value;
```

or

```
value[0] = (float) sld.value;
```

The `channelName` string `freq` is the reference text used by the `chnget` opcode in the `instr 1` of the Csound Orchestra.

```
kfr chnget “freq”
```

Since the control architecture is based on the callback mechanism and therefore depends on the user actions, we must send all values when Csound starts. We can use Csound's delegate:

```
-(void)csoundObjDidStart {
    [_freq sendActionsForControlEvents:UIControlEventTouchUpInsideAllEvents];
}
```

This operation must be repeated for all UI widgets in practice. Immediately after Csound is running we send an `UIControlEventsAllEvents` message to all widgets. So we are sure that Csound receives properly the current state of the UI's widgets values.

In this case `_freq` is the reference (IBOutlet) of the `UISlider` in the **Main.storyboard**.

Enabling Audiobus and Inter-App Audio

The last line of code in the `initializeAudio` function calls the `initializeAB_IAA` for initialize and configure the Inter-App Audio and Audiobus.

The XCode tutorials do not include the Audiobus SDK since it is covered by license, see the website for more information and to consult the official documentation [here](#).

However, the existing code to Audiobus should ensure proper functioning after the inclusion of the library.

In the file `AudioDSP.h` there are two macros: `AB` and `IAA`. These are used to include or exclude the needed code. The first step is to configure the two `AudioComponentDescriptions` for the types: `kAudioUnitType_RemoteInstrument` and `kAudioUnitType_RemoteEffect`.

```
/* Create Sender and Filter ports */
AudioComponentDescription desc_instr = {
    kAudioUnitType_RemoteInstrument,
    'icso',
    'iyou', 0, 0
};

AudioComponentDescription desc_fx = {
    kAudioUnitType_RemoteEffect,
    'xcso',
    'xyou', 0, 0
};
```

This point is crucial because you have to enter the same information in the file `Info.plist`

▼ Information Property List	Dictionary	(19 items)
Bundle display name	String	\$(PRODUCT_NAME)
▼ Required background modes	Array	(1 item)
Item 0	String	App plays audio or streams audio/video using AirPlay
▼ AudioComponents	Array	(2 items)
▼ Item 0	Dictionary	(5 items)
manufacturer	String	iyou
name	String	cs4dev (Instr)
subtype	String	icso
type	String	auri
version	Number	1
▼ Item 1	Dictionary	(5 items)
manufacturer	String	xyou
name	String	cs4dev (Fx)
subtype	String	xcso
type	String	aurx
version	Number	1

In the `Info.plist` (i.e. Information Property List), the `Bundle display name` key and `Required background modes` must absolutely be defined to enable the audio in the background.

The app must continue to play audio even when it is not in the foreground. Here we configure the `Audio Components` (i.e. AU).

```
typedef struct AudioComponentDescription {
    OSType componentType;
    OSType componentSubType;
    OSType componentManufacturer;
    UInt32 componentFlags;
    UInt32 componentFlagsMask;
} AudioComponentDescription;
```

As said, the *AudioComponentDescription* structure used for the configuration of the AU, must necessarily coincide in the *Info.plist*,

The structure fields (*OSType*) are of *FourCharCode*, so they must consist of four characters.

IMPORTANT: it is recommended to use different names for both *componentSubType* and *componentManufacturer* of each *AudioComponent*. In the example the characters 'i' and 'x' refer to *Instrument* and *Fx*.

Only for the first field (*componentType*) of the *AudioComponentDescription* structure we can use the enumerator

```
enum {
    kAudioUnitType_RemoteEffect      = 'aurx',
    kAudioUnitType_RemoteGenerator   = 'aurg',
    kAudioUnitType_RemoteInstrument = 'auri',
    kAudioUnitType_RemoteMusicEffect = 'aurm'
};
```

where *auri* identifies the *Instrument* (*Instr*) and *aurx* the effect (*Fx*), at which point the app will appear on the lists of the various *IAA Host* as *Instr* and *Fx* and in *Audiobus* as Sender or Receiver.

At this point we are able to:

1. Perform Audio in the background
2. Get IAA and AB support for input/output
3. Toggle DSP (Csound) on and off
4. Control Csound through the *callback* mechanism
5. Record the output audio

In the following sections we will see how to manage the advanced settings for Csound's *ksmps*, according to the system BufferFrame.

Buffer Frame vs *ksmps*

In IOS, the first audio app which is running (in foreground or in background), imposes its own Sampling Rate and BufferFrame to the whole iOS (i.e. for all audio apps).

iOS allows power-of-two BufferFrame values in the range 64, 128, 256, 512, 1024, etc ...

It is not recommended to use values bigger than 1024 or smaller than 64. A good compromise is 256, as suggests the default value of *GarageBand* and Other similar applications.

In the Csound language, the vector size (i.e. BufferFrame in the example) is expressed as *ksmps*. So it is necessary to manage appropriately the values of *BufferFrame* and *ksmps*.

There are three main possible solutions:

1. Keep the *ksmps* static with a very low value, such as 32 or 64
2. Dynamically manage the *ksmps* depending on *BufferFrame*
3. *BufferFrame* and *ksmps* decorrelation

All three cases have advantages and disadvantages. In the first case the BufferFrame must be always \geq *ksmps*, and in the second case we must implement a spartan workaround tp synchronize

ksmps with *BufferFrame*. The third and more complex case requires a control at run-time on the audio callback and we must manage an accumulation buffer. Thanks to this, the *BufferFrame* can be bigger than *ksmps* or vice versa. However there are some limitations. In fact, this approach does not always lead to the benefits hoped for in terms of performance.

Static *ksmps*

To keep the *ksmps* static with a very low value, such as 32 or 64, we will set *ksmps* in the Csound Orchestra to this value. As mentioned, the *BufferFrame* of iOS is always greater or equal than 64. The operation is assured thanks to the *for* statement in the *Csound_Render*:

```
OSStatus Csound_Render(void *inRefCon,
                      AudioUnitRenderActionFlags *ioActionFlags,
                      const AudioTimeStamp *inTimeStamp,
                      UInt32 dump,
                      UInt32 inNumberFrames,
                      AudioBufferList *ioData) {
    //...

    /* inNumberFrames => ksmmps */
    for(int i = 0; i < (int)slices; ++i){
        ret = csoundPerformKsmmps(cs);
    }
    //...
}
```

This C routine is called from the CoreAudio every *inNumberFrames* (i.e. *BufferFrame*). The *ioData* pointer contains *inNumberFrames* of audio samples incoming from the input (mic/line). Csound reads this data and returns *ksmps* processed samples.

When the *inNumberFrames* and *ksmps* are identical, we can simply copy out the processed buffer, this is done by calling the *csoundPerformKsmmps()* procedure. Since that *ksmps* is less or equal to *inNumberFrames*, we need to call *N slices* the *csoundPerformKsmmps()*. This is safe, as *ksmps* will in this situation never be greater than *inNumberFrames*.

Example:

```
ksmps = 64
inNumberFrames = 512
```

slices is calculated as follows:

```
int slices = inNumberFrames / csoundGetKsmmps(cs);
slices is 8
```

In other words, every *Csound_Render* call involves eight sub-calls to *csoundPerformKsmmps()*, for every sub-call we fill the *ioData* with *ksmps* samples.

Dynamic ksmmps, Buffer Frame and Sampling Rate Synchronization

The *ksmps* value should be chosen according to the Csound Orchestra operating logic. If the Orchestra is particularly heavy in terms of k-rate operations, which however do not require low-latency, we can use higher values as 512 or 1024. This second case is adoptable for many apps developed so far. In fact, except in specific cases, it is always convenient to set the *ksmps* to the same value as the system's *BufferFrame*.

The following steps are required to change *sr* and *ksmps*:

1. Stop and Clean the *Csound* Object
2. Replace the Orchestra Code in the .csd file with new *sr* and *ksmps*
3. Initialize and Run the *Csound* Object with these new values

This is a workaround but it works properly; we just have to set placeholders in the Orchestra header.

```
<CsInstruments>
sr = 44100
ksmps = 512

;;;;SR;;;; //strings replaced from Objective-C
;;;;KSMPS;;;;
nchnls = 2
0dbfs = 1

...
```

The two univocal strings are the placeholders for *sr* and *ksmps*. They begin with the semicolon character so that Csound recognizes it as a comment. The following function in Objective-C looks for the placeholders in the *myOrchestra.csd* and replaces them with new *sr* and *ksmps* values.

```
- (void)csoundApplySrAndKsmmpsSettings:(Float64)sr withBuffer:(Float64)ksmps
{
    NSString* pathAndName = [[[NSBundle mainBundle] resourcePath]
        stringByAppendingPathComponent:@"/myOrchestra.csd"];

    if ([[NSFileManager defaultManager] fileExistsAtPath:pathAndName]) {
        NSString* myString =
            [[NSString alloc] initWithContentsOfFile:pathAndName
                encoding:NSUTF8StringEncoding
                error:NULL];

        myString = [myString
            stringByReplacingOccurrencesOfString:@";;;SR;;;;"
            withString:[NSString
                stringWithFormat:@"sr = %f", sr]];

        myString = [myString
            stringByReplacingOccurrencesOfString:@";;;KSMPS;;;;"
            withString:[NSString
                stringWithFormat:@"ksmps = %f", ksmmps]];

        NSString* pathAndNameRUN =
            [NSString stringWithFormat:@"%@dspRUN.csd", NSTemporaryDirectory()];

        NSError* error = nil;
```

```

// save copy of dspRUN.csd in library directory
[myString writeToFile:pathAndNameRUN
    atomically:NO
    encoding:NSUTF8StringEncoding
    error:&error];

// Run Csound
[self startCsound:pathAndNameRUN];

} else
NSLog(@"file %@ Does Not Exist At Path!!!", pathAndName);
}

```

The NSString *pathAndName* contains the file path of *myOrchestra.csd* in the Resources folder. This path is used to copy in *myString* the entire file (as NSString). Subsequently the *stringByReplacingOccurrencesOfString* method replaces the placeholders with the valid strings.

Since iOS does not allow to edit files in the application *Resources* folder (i.e. *pathAndName*), we need to save the modified version in the new file *dspRUN.csd* that is saved in the temporary folder (i.e. *pathAndNameRUN*). This is achieved through the *writeToFile* method.

As a final step it is necessary to re-initialise Csound by calling the *runCsound* function which runs Csound and sends the appropriate values of *sr* and *ksmps*.

###BufferFrame and *ksmps* decorrelation

As seen the second case is a good compromise, however it is not suitable in some particular conditions. So far we have only considered the aspect in which the app works on the main audio thread, with a *BufferFrame* imposed by iOS. But there are special cases in which the app is called to work on a different thread and with a different *BufferFrame*.

For instance the *freeze track* feature implemented by major Host IAA apps (such as Cubasis, Auria etc ...) bypasses the current setup of iOS and imposes an arbitrary *BufferFrame* (usually 64).

Since Csound is still configured with the iOS *BufferFrame* (the main audio thread), but during the freeze track process the *Csound_Perform* routine is called with a different BufferFrame, Csound cannot work properly.

In order to solve this limitation we need a run-time control on the audio callback and handle the exception.

On the *Csound_Render* we will evaluate the condition for which *slices* is < 1:

```

OSStatus Csound_Perform(void *inRefCon,
                        AudioUnitRenderActionFlags *ioActionFlags,
                        const AudioTimeStamp *inTimeStamp, UInt32 dump,
                        UInt32 inNumberFrames, AudioBufferList *ioData) {

//...

/* CSOUND PERFORM */
if (slices < 1.0) {
    /* inNumberFrames < ksmmps */
    Csound_Perform_DOWNSAMP(inRefCon, ioActionFlags, inTimeStamp, dump,
                            inNumberFrames, ioData);
} else {
    /* inNumberFrames => ksmmps */
    for (int i = 0; i < (int)slices; ++i) {
        ret = csoundPerformKsmmps(cs);
}

```

```

    }
}

//...
}
}

```

Please note that *slices* is calculated as follows:

```
int slices = inNumberFrames / csoundGetKsmmps(cs);
```

Every time the *ksmps* (for some reason) is greater than *BufferFrame*, we will perform the *Csound_Perform_DOWNSAMP* procedure.

```

// Called when inNumberFrames < ksmmps
OSStatus Csound_Perform_DOWNSAMP(
    void *inRefCon,
    AudioUnitRenderActionFlags *ioActionFlags,
    const AudioTimeStamp *inTimeStamp, UInt32 dump,
    UInt32 inNumberFrames,
    AudioBufferList *ioData
) {
    AudioDSP *cdata = (_bridge AudioDSP *)inRefCon;

    int ret = cdata->ret, nchnls = cdata->nchnls;
    CSOUND *cs = cdata->_cs;

    MYFLT *spin = csoundGetSpin(cs);
    MYFLT *spout = csoundGetSpout(cs);
    MYFLT *buffer;

    /* DOWNSAMPLING FACTOR */
    int UNSAMPLING = csoundGetKsmmps(cs) / inNumberFrames;

    if (cdata->counter < UNSAMPLING - 1) {
        cdata->counter++;
    } else {
        cdata->counter = 0;
    }

    /* CSOUND PROCESS KSMPS */
    if (!cdata->ret) {
        /* PERFORM CSOUND */
        cdata->ret = csoundPerformKsmmps(cs);
    } else {
        cdata->running = false;
    }
}

/* INCREMENTS DOWNSAMPLING COUNTER */
int slice_downsamp = inNumberFrames * cdata->counter;

/* COPY IN CSOUND SYSTEM SLICE INPUT */
for (int k = 0; k < nchnls; ++k) {
    buffer = (MYFLT *)ioData->mBuffers[k].mData;
    for (int j = 0; j < inNumberFrames; ++j) {
        spin[(j + slice_downsamp) * nchnls + k] = buffer[j];
    }
}

/* COPY OUT CSOUND KSMPS SLICE */
for (int k = 0; k < nchnls; ++k) {
    buffer = (MYFLT *)ioData->mBuffers[k].mData;
    for (int j = 0; j < inNumberFrames; ++j) {
        buffer[j] = (MYFLT)spout[(j + slice_downsamp) * nchnls + k];
    }
}

```

```

    }

cdata->ret = ret;
return noErr;
}

```

As mentioned we need a buffer for the accumulation. It is, however, not necessary to create a new one since you can directly use Csound's *spin* and *spout* buffer.

First we have to evaluate what is the level of under-sampling, for example:

```

Csound ksmps = 512
iOS inNumberFrames = 64

/* DOWNSAMPLING FACTOR */
int UNSAMPLING = csoundGetKsmps(cs)/inNumberFrames;

UNSAMPLING is 8

```

This value represents the required steps to accumulate the input signal in *spin* for every call of *csoundPerformKsmps()*.

```

if (cdata->counter < UNSAMPLING-1) {
    cdata->counter++;
}
else {
    cdata->counter = 0;

    /* CSOUND PROCESS KSMPS */
    if(!cdata->ret) {
        cdata->ret = csoundPerformKsmps(cs);
    }
}

```

The *Csound_Perform_DOWNSAMP* routine is called by iOS every 64 samples, while we must call *csoundPerformKsmps()* after 512 samples. This means we need to skip eight times (i.e. UNSAMPLING) until we have collected the input buffer.

From another point of view, before calling *csoundPerformKsmps()* we must accumulate eight *inNumberFrames* in *spin*, and for every call of *Csound_Perform_DOWNSAMP* we must return *inNumberFrames* from *spout*.

In the next example, the iOS audio is in *buffer* which is a pointer of the *ioData* structure.

```

/* INCREMENTS DOWNSAMPLING COUNTER */
int slice_downsamp = inNumberFrames * cdata->counter;

/* COPY IN CSOUND SYSTEM SLICE INPUT */
for (int k = 0; k < nchnls; ++k){
    buffer = (MYFLT *) ioData->mBuffers[k].mData;
    for(int j = 0; j < inNumberFrames; ++j){
        spin[(j+slice_downsamp)*nchnls+k] = buffer[j];
    }
}

/* COPY OUT CSOUND KSMPS SLICE */
for (int k = 0; k < nchnls; ++k) {
    buffer = (MYFLT *) ioData->mBuffers[k].mData;
    for(int j = 0; j < inNumberFrames; ++j) {
        buffer[j] = (MYFLT) spout[(j+slice_downsamp)*nchnls+k];
    }
}

```

```

    }
}

```

Ignoring the implementation details regarding the de-interlacing of the audio, we can focus on the ***slice_downsamp*** which serves as offset-index for the arrays *spin* and *spout*.

The implementation of both second and third cases guarantees that the app works properly in every situation.

Plot a Waveform

In this section we will see a more complex example to access memory of Csound and display the contents on a *UIView*.

The *waveDrawView* class interacts with the *waveLoopPointsView*, the *loopoints* allow us to select a portion of the file via the zoom on the waveform (pinch in / out). These values (*loopoints*) are managed by Csound which ensures the correct reading of the file and returns the normalized value of the instantaneous phase of reading.

The two classes are instantiated in ***Main.storyboard***. Please note the hierarchy that must be respected for the setup of other projects as well as the three *UIView* must have the same size (frame) and cannot be dynamically resized.



In the score of the file *csound_waveform.csd*, two *GEN Routines* are declared to load WAV files in memory:

```

f2 0 0 1 "TimeAgo.wav" 0 0 1
f3 0 0 1 "Density_Sample08.wav" 0 0 1
  
```

In order to access the audio files in the app Resources folder, we need to setup some environment variables for Csound. This is done in the *runCsound* function. Here we set the SFDIR (Sound File Directory) and the SADIR (Sound analysis directory):

```

// Set Environment Sound Files Dir
NSString *resourcesPath = [[NSBundle mainBundle] resourcePath];
NSString *envFlag = @"--env:SFDIR+=";

char *SFDIR = (char *)[envFlag stringByAppendingString:resourcesPath]
cStringUsingEncoding:NSUTF8StringEncoding];

envFlag = @"--env:SADIR+=";
char *SADIR = (char *)[envFlag stringByAppendingString:resourcesPath]
cStringUsingEncoding:NSUTF8StringEncoding;

char *argv[4] = {
    "csound",
    SFDIR,
    SADIR,
    (char *) [csdFilePath cStringUsingEncoding:NSUTF8StringEncoding];
};

ret = csoundCompile(_cs, 4, argv);
  
```

The interaction between Csound and the UI is two-way, the class method `drawWaveForm` draws the contents of the `genNum`.

```
[waveView drawWaveFromCsoundGen:_cs genNumber:genNum];
```

After calling this method, we need to enable an `NSTimer` object in order to read continuously (pull) the phase value returned by Csound. Please examine the `loadSample_1` function code for insights.

The timer is disabled when the DSP is switched off, in the timer-callback we get the pointer, this time from `CSOUND_OUTPUT_CHANNEL`, finally we use this value to synchronize the graphics cursor on the waveform (scrub) in the GUI.

```
- (void)updateScrubPositionFromTimer {
    if (!running) return;

    MYFLT* channelPtr_file_position = nil;
    csoundGetChannelPtr(_cs, &channelPtr_file_position,
        @"file_position_from_csound"
        cStringUsingEncoding:NSUTF8StringEncoding],
        CSOUND_CONTROL_CHANNEL | CSOUND_OUTPUT_CHANNEL);

    if (channelPtr_file_position) {
        [waveView updateScrubPosition:@*channelPtr_file_position];
    }
}
```

In the Orchestra we find the corresponding code for writing in the software bus.

```
chnset kfilposphas, "file_position_from_csound"
```

Write into a Csound GEN table

We already have seen how to read from the Csound's GEN memory. Now we will focus on the write operation with two possible ways.

The goal is to modify a table in realtime while being read (played) by an oscillator LUT (i.e. look-up table). A **Pad XY**, to the left in the UI, manages the interpolation on the four prototypes and, to the right of the interface, a **16-slider** surface controls the harmonic content of a wave.

Concerning the first example (pad morph), the waveform interpolations are implemented in the Orchestra file and performed by Csound. The UI communicates with Csound, by activating an instrument (`instr 53`) through a score message. Instead, in the second example (16-slider surface) the code is implemented in the `AudioDSP.m` file and, precisely, in the `didValueChanged` delegate. The architecture of this second example is based on `addArm` procedure that write in a temporary array. The resulting waveform is then copied to the GEN-table, via the `csoundTableCopyIn` API.

In the first example, `instr 53` is activated via a score message for every action on the pad, this is performed in `ui_wavesMorphPad`:

```
NSString* score = [NSString stringWithFormat:
    @"i53 0 %f %f %f",
    UPDATE_RES,
    pad.xValue,
    pad.yValue];
```

```
csoundInputMessage(_cs, [score cStringUsingEncoding:NSUTFStringEncoding]);
```

The *instr* 53 is kept active for UPDATE_RES sec (0.1), the *maxalloc* opcode limits the number of simultaneous instances (notes). Thus, any score events which fall inside UPDATE_RES time, are ignored.

```
maxalloc 53, 1 ;iPad UI Waveforms morphing only 1 instance
```

This results in a sub-sampling of Csound's *instr* 53, compared to the UI pad-callback. The waveform display process is done by the Waveview class, it is a simplified version of the WaveDrawView class, introduced in the tutorial (**04_plotWaveForm**), that does not deserve particular investigation. As mentioned, the waveforms's interpolations are performed by Csound, followed by the *instr* 53 code:

```
tableimix giWaveTMP1, 0, giWaveSize, giSine, \
    0, 1.-p4, giTri, 0, p4
tableimix giWaveTMP2, 0, giWaveSize, giSawSmooth, \
    0, 1.-p4, giSquareSmooth, 0, p4
tableimix giWaveMORPH, 0, giWaveSize, giWaveTMP2, \
    0, 1.-p5, giWaveTMP1, 0, p5
chnset giWaveMORPH , "wave_func_table"
```

The p4 and p5 p-fields are the XY pad axes used as weights for the three vector-interpolations which are required. The *tablemix* opcode mixes two tables with different weights into the *giWaveTMP1* destination table. In this case we interpolate a Sine Wave (i.e. *giSine*) with a triangular (i.e. *giTri*), then in the second line between *giSawSmooth* and *giSquareSmooth*, mixing the result in *giWaveTMP2*. At the end of the process, ***giWaveMORPH*** contains the interpolated values of the two *giWaveTMP1* and *giWaveTMP2* arrays.

The global *ftgen-tables*, deliberately have been declared with the first argument set to zero. This means that the GEN_-table number is assigned dynamically from Csound at compile time. Since we do not know the number assigned, we must return the number of the table through *chnset* at runtime.

In the ***AudioDSP.m*** class is the implementation code of the second example.

The **APE_MULTISLIDER** class returns, through its own delegate method *didValueChanged*, an array with the indexed values of the sliders. These are used as amplitude-weights for the generation of the harmonic additive waveform. Let us leave out the code about the wave's amplitude normalization and we focus on this code:

```
MYFLT *tableNumFloat;
csoundGetChannelPtr(_cs, &tableNumFloat,
    @"harm_func_table"
    cStringUsingEncoding:NSUTFStringEncoding],
    CSOUND_CONTROL_CHANNEL | CSOUND_INPUT_CHANNEL);

/* Contain the table num (i.e. giWaveHARM) */
int tableNum = (int)*tableNumFloat;

/* Contain the table (giWaveHARM) Pointer */
MYFLT *tablePtr;
int tableLength = csoundGetTable(_cs, &tablePtr, tableNum);

/* Is invalid? Return */
if (tableLength <= 0 || tableNum <= 0 || !tablePtr)
```

```

    return;

/* Clear temporary array */
memset(srcHarmonic, 0, tableLength * sizeof(MYFLT));

/* Generate an additive sinusoidal waveform with 16 harmonics */
for (int i = 0; i < maxnum; ++i) {
    [self appendHarm:i + 1
        Amp:(powf(value[i], 2.0)) * average
        SIZE:tableLength
        DEST:srcHarmonic];
}

/* Write array in the Csound Gen Memory (i.e. giWaveHARM) */
csoundTableCopyIn(_cs, tableNum, srcHarmonic);

```

This function also can be sub-sampled by de-commenting the DOWNSAMP_FUNC macro. This code is purely for purpose of example as it can be significantly optimized, in the case of vectors's operations, the Apple vDSP framework could be an excellent solution.

Optimize performance and add a custom opcode

In this final section we will understand how to use the programming environment to implement an opcode directly in the **AudioDSP** class and add it to the list of Csound opcodes without re-compiling Csound. This is fundamental in order to optimize some audio processing, in particular heavy ones regarding CPU cost. In fact, outside of Csound it will be possible to use a series of instruments such as the highly powerful vDSP of Apple, especially for the implementation of FFT routines.

The steps involved are three:

1. add custom opcode to the Csound opcode list
2. declare opcode structure
3. implement functions

The first step must be done in the *runCsound* process, before calling *csoundCompile*.

```

csoundAppendOpcode(
    cs, "MOOGLADDER", sizeof(MOOGLADDER_OPCODE),
    0, 3, "a", "akk", iMOOGLADDER, kMOOGLADDER, aMOOGLADDER
);

```

This appends an opcode implemented by external software to Csound's internal opcode list. The opcode list is extended by one slot, and the parameters are copied into the new slot.

Basically, what we have done is declaring three pointers to functions (iMOOGLADDER, kMOOGLADDER and aMOOGLADDER) implemented in the class **AudioDSP**.

The second step is to declare the data structure used by the opcode in the **AudioDSP.h**. So the header file csdl.h must be included according to the documentation:

Plugin opcodes can extend the functionality of Csound, providing new functionality that is exposed as opcodes in the Csound language. Plugins need to include this header file only, as it will bring all necessary data structures to interact with Csound. It is not necessary for plugins to link to the libcsound library, as plugin opcodes will always receive a CSOUND pointer (to the CSOUND_struct) which contains all the API functions inside. This is the basic template for a plugin opcode. See the*

manual for further details on accepted types and function call rates. The use of the LINKAGE macro is highly recommended, rather than calling the functions directly.

```
typedef struct {
    OPDS      h;
    MYFLT    *ar, *asig, *kcutoff, *kresonance;
    //...
} MOOGLADDER_OPCODE;
```

Finally, the implementation of the three required functions in **AudioDSP.m**:

```
int iMOOGLADDER(CSOUND *csound, void *p_) {
    //...
}

int kMOOGLADDER(CSOUND *csound, void *p_)
{
    //...
}

int aMOOGLADDER(CSOUND *csound, void *p_)
{
    //...
}
```

In the Orchestra code, we can call MOOGLADDER in the same way as the native opcodes compiled:

```
aOutput MOOGLADDER aInput, kcutoff, kres
```

The MOOGLADDER is a simplified and optimized implementation of the opcode moogladder by Victor Lazzarini. The iVCS3 app uses this mechanism for the Envelope and Filter implementation—that also allows a fine control of the cutoff.

Conclusion

1. The descriptions here describe the essential audio integrations in iOS. Some of the topics will be soon out of date, like the Inter-Audio App (IAA) which is deprecated from Apple since iOS 13, or the Audiobus which is replaced as well from the modern AUv3 technology.
2. This approach covers the inalienable features for the audio integration using Csound for professional software audio applications and presents some workarounds to solve some intrinsic idiosyncratic issues related to the Csound world.
3. A separate study deserves the integration of Csound for the Av3 architecture, meanwhile in the [tutorial repository](#) you can download an Xcode project template using Csound as audio engine for an AUv3 plugins extension. The template is self-explanatory.
4. All the tutorials are using the latest Csound 6.14 compiled for Apple Catalyst SDK it means that the app can run as universal in both iOS and macOS (since Catalina >= 10.15).

Links

[Csound for iOS](#) (look for the iOS-zip file)

Online Tutorial

apeSoft

Audiobus

A Tasty

The Open Music App Collaboration Manifesto

12 E. CSOUND ON ANDROID

There is no essential difference between running Csound on a computer and running it on a smartphone. Csound has been available on the Android platform since 2012 (Csound 5.19), thanks to the work of Victor Lazzarini and Steven Yi. Csound 6 was ported to Android, and enhanced, by Michael Gogins and Steven Yi in the summer of 2013.

The following packages are available for Android:

1. The *CsoundAndroid library*, which is intended to be used by developers for creating apps based on Csound. This is available for download at [Csound's download page](#).
2. The *Csound for Android app*, which is a self-contained environment for creating, editing, debugging, and performing Csound pieces on Android. The app includes a number of built-in example pieces. This is available from the [Google Play store](#), or for download from the [csound-extended repository releases page](#).

For more information about these packages, download them and consult the documentation contained therein.

This chapter is about the Csound for Android app.

The Csound for Android app

The Csound for Android app permits the user, on any Android device that is powerful enough, including most tablets and the most powerful smartphones, to do most things that can be done with Csound on any other platform such as OS X, Windows, or Linux. This includes creating Csound pieces, editing them in the built-in text editor, debugging them, and performing them, either in real time to audio output or to a soundfile for later playback.

The app has a built-in, pre-configured user interface with nine sliders, five push buttons, one trackpad, and a 3-dimensional accelerometer that are pre-assigned to control channels which can be read using Csound's [chnget](#) opcode.

The app also contains an embedded Web browser, based on WebKit, that implements most features of the HTML5 standard. This embedded browser can run Csound pieces written as *.html* files. In addition, the app can render HTML and JavaScript code that is contained in an optional `<html>` element of a regular *.csd* file.

In both cases, the JavaScript context of the Web page will contain a global Csound object with a JavaScript interface that implements useful functions of the Csound API. This can be used to control Csound from JavaScript, handle events from HTML user interfaces, generate scores, and do many other things. For a more complete introduction to the use of HTML with Csound, see [12 G.](#)

The app has some limitations and missing features compared with the longer-established platforms:

1. There is no real-time MIDI input or output.
2. Audio input is not always accurately synchronized with audio output.
3. Some plugin opcodes are missing, including most opcodes involved with using other plugin formats or inter-process communications.

However, some of the more useful plugins are indeed available on Android:

1. The signal flow graph opcodes for routing audio from instruments to effects, etc.
2. The FluidSynth opcodes for playing SoundFonts.
3. The Open Sound Control (OSC) opcodes.
4. The libstdutil library, which enables Csound to be used for various time/frequency analysis and resynthesis tasks, and for other purposes.

Installing the App

There are several ways to install the Csound for Android app. You can download it using your device, or you can download it to a computer and transfer it to your device. These methods are presented below.

Google Play Store

The most straightforward way to install the Csound for Android app is to get it from the [Google Play Store](#).

Install from Another Source

Preparing Your Device

Using the *Csound for Android* app is similar to using an application on a regular computer. You need to be able to browse the file system.

There are a number of free and paid apps that give users the ability to browse the Linux file system that exists on all Android devices. If you don't already have such a utility, you should install a file browser that provides access to as much as possible of the file system on your device, including

system storage and external store such as an SD card. The free [AndroZip](#) app can do this, for instance.

If you render soundfiles, they take up a lot of space. For example, CD-quality stereo soundfiles (44.1 KHz, 16 bit) take up about 1 megabytes per minute of sound. Higher quality or more channels take up even more room. But even without extra storage, a modern smartphone should have gigabytes, thousands of megabytes, of free storage. This is actually enough to make an entire album of pieces.

On most devices, installing extra storage is easy and not very expensive. Obtain the largest possible SD card, if your device supports them. This will vastly expand the amount of available space, up to 32 or 64 gigabytes or even more.

Download to Device

To download the Csound for Android app to your device, go online using Google Search or a Web browser. You can find the application package file, CsoundApplication-release.apk, on the [csound-extended releases page](#) (you may first have to allow your Android device to install an app which is not in Google Play).

Click on the filename to download the package. The download will happen in the background. You can then go to the notifications bar of your device and click on the downloaded file. You will be presented with one or more options for how to install it. The installer will ask for certain permissions, which you need to grant.

Transfer from a Computer

It's also easy to download the CsoundApplication-release.apk file to a personal computer. Once you have downloaded the file from GitHub, connect your device to the computer with a USB cable. The file system of the device should then automatically be mounted on the file system of the computer. Find the CsoundApplication-release.apk in the computer's download directory, and copy the CsoundApplication-release.apk file. Find your device's download directory, and paste the CsoundApplication-release.apk file there.

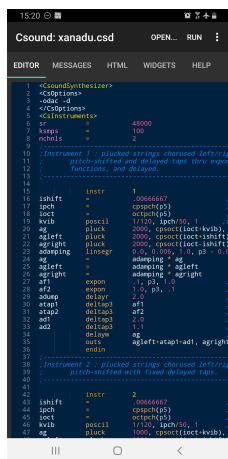
Then you will need to use a file browser that is actually on your device, such as AndroZip. Browse to your Download directory, select the CsoundApplication-release.apk file, and you should be presented with a choice of actions. Select the Install action. The installer will ask for certain permissions, which you should give.

User Interface

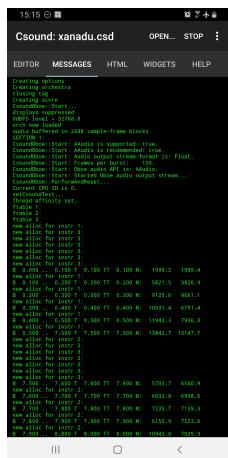
Tabs

The Csound for Android app has a tabbed user interface. The tabs include:

EDITOR – Built-in text editor for .csd and .html files.



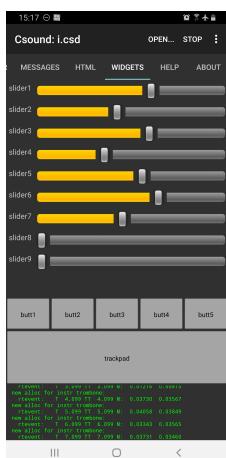
MESSAGES – Displays runtime messages from Csound in a scrolling display.



HTML – Displays the Web page specified by HTML code in the piece, may include interactive widgets, 3-dimensional graphics, etc., etc.



WIDGETS – Displays built-in widgets bound to control channels with predefined names.



HELP – Displays the online Csound Reference Manual in an embedded Web browser. **ABOUT** – Displays the Csound home page in an embedded Web browser.

Main Menu

The app also has a top-level menu with the following commands:

NEW... creates a blank template CSD file in the root directory of the user's storage for the user to edit. The CSD file will be remembered and performed by Csound.

OPEN... – opens an existing CSD file in the root directory of the user's storage. The user's storage filesystem can be navigated to find other files.

SAVE – saves the current contents of the editor to its file.

RUN/STOP – if a CSD file has been loaded, pushing the button starts running Csound; if Csound is running, pushing the button stops Csound. If the `<CsOptions>` element of the CSD file contains `-odac`, Csound's audio output will go to the device audio output. If the element contains `-osoundfilename`, Csound's audio output will go to the file `soundfilename`, which should be a valid Linux pathname in the user's storage filesystem.

Save as ... – saves the current contents of the editor to a new file.

Examples – shows a number of example pieces that may be loaded.

User guide – a minimal guide to setting up and using the app.

Privacy policy – presents the Csound for Android app's privacy policy.

The widgets are assigned control channel names `slider1` through `slider9`, `butt1` through `butt5`, `trackpad.x`, and `trackpad.y`. In addition, the accelerometer on the Android device is available as `accelerometerX`, `accelerometerY`, and `accelerometerZ`.

The values of these widgets are normalized between 0 and 1, and can be read into Csound during performance using the `chnget` opcode, like this:

```
kslider1_value chnget "slider1"
```

The area below the trackpad prints messages output by Csound as it runs.

Settings Menu

The Settings menu on your device offers the following choices:

Audio driver – selects an *Automatic* choice of the optimal audio driver for your device (this is the default), the older *OpenSL ES* driver which supports both audio input and audio output, and the newer *AAudio* driver that provides lower audio output latency on Oreo or later. **Plugins** – an (additional) directory for plugin opcodes. **Output** – overrides the default soundfile output directory. **Samples** – overrides the default directory from which load sound samples. **Analysis** – overrides the default directory from which to load analysis files. **Include** – overrides the default directory from which to load Csound #include files.

These settings are not required, but they can make using Csound easier and faster to use.

Path for Samples

To read a sample from the internal storage, add `/storage/emulated/0/locationofyoursample` to Csound's search path via the `--env:SSDIR` flag in the `<CsOptions>` tag of your `.csd` file, for instance:

```
<CsOptions>
--env:SSDIR+=/storage/emulated/0/SAMPLES
</CsOptions>
```

Loading and Performing a Piece

Example Pieces

From the app's menu, select the *Examples* command, then select one of the listed examples, for example *Xanadu* by Joseph Kung. You may then click on the *RUN* button to perform the example, or the *EDITOR* tab to view the code for the piece. If you want to experiment with the piece, you can use the *Save as...* command to save a copy on your device's file system under a different name. You can then edit the piece and save your changes.

Running an Existing Piece

If you have access to a mixer and monitor speakers, or even a home stereo system, or even a boom box, you can hook up your device's headphone jack to your sound system with an adapter cable. Most devices have reasonably high quality audio playback capabilities, so this can work quite well.

Just to prove that everything is working, start the Csound for Android app. Go to the app menu, select the *Examples* item, select the *Xanadu* example, and it will be loaded into Csound. Then click on the *RUN* command. Its name should change to *STOP*, and Csound's runtime messages should begin to scroll down the *MESSAGES* tab. At the same time, you should hear the piece play. You can stop the performance at any time by selecting the *STOP* command, or you can let the performance complete on its own.

That's all there is to it. You can scroll up and down in the messages pane if you need to find a particular message, such as an error or warning.

If you want to look at the text of the piece, or edit it, select the *Edit* button. If you have installed Jota, that editor should open with the text of the piece, which you can save, or not. You can edit the piece with this editor, and any changes you make and save will be performed the next time you start the piece.

Creating a New Piece

This example will take you through the process of creating a new Csound piece, step by step. Obviously, this piece is not going to reveal anything like the full power of Csound. It is only intended to get you to the point of being able to create, edit, and run a Csound piece that will actually make sound on your Android device – from scratch.

Run the *Csound for Android* app and select the *NEW...* command. You should be presented with a file dialog asking you for a filename for your piece. Type in *toot.csd*, and select the *SAVE* button. The file will be stored in the root directory of your user storage on your device. You can save the file to another place if you like.

The text editor should open with a *template* CSD file. Your job is to fill out this template to hear something.

Create a blank line between *<CsOptions>* and *</CsOptions>*, and type *-odac -d -m3*. This means send audio to the real-time output (*-odac*), do not display any function tables (*-d*), and log some informative messages during Csound's performance (*-m3*).

Create a blank line between *<CsInstruments>* and *</CsInstruments>* and type the following text:

```
sr = 44100
ksmps = 32
nchnls = 1
0dbfs = 1
instr 1
    asignal oscil 0.2, 440
    out asignal
endin
```

This is just about the simplest possible Csound orchestra. The orchestra header specifies an audio signal sampling rate of 44,100 frames per second, with 32 audio frames per control signal sample, and one channel of audio output. The instrument is just a simple sine oscillator. It plays a tone at concert A.

Create a blank line between *<CsScore>* and *</CsScore>* and type:

```
i1 0 5
```

This means play instrument 1 starting at time 0 for 5 seconds.

Select the app's **SAVE** button.

Select the Csound app's **RUN** button. You should hear a loud sine tone for 5 seconds. If you don't hear anything, perhaps your device doesn't support audio at 44100 Hertz, so try `sr = 48000` instead.

If you want to save your audio output to a soundfile named `test.wav`, change `-odac` above to, for example, `-o/storage/emulated/0/Music/test.wav`. Android is fussy about writing to device storage, so you may need to use exactly the directory printed in the **MESSAGES** tab when the app starts.

That's it!

Using the Widgets

This section shows how to use the built-in widgets of the Csound for Android app for controlling Csound in performance. For instructions on how to use the `<html>` element of the CSD file to create custom user interfaces, see the [Csound and HTML](#) chapter of this book.

The Csound for Android app provides access to a set of predefined on-screen widgets, as well as to the accelerometer on the device. All of these controllers are permanently assigned to pre-defined control channels with pre-defined names, and mapped to a pre-defined range of values, from 0 to 1.

You should be able to cut and paste this code into your own pieces without many changes.

The first step is to declare one global variable for each of the control channels, with the same name as the control channel, at the top of the orchestra header, initialized to a value of zero:

```
gkslider1 init 0
gkslider2 init 0
gkslider3 init 0
gkslider4 init 0
gkslider5 init 0
gkslider6 init 0
gkslider7 init 0
gkslider8 init 0
gkslider9 init 0
gkbutt1 init 0
gkbutt2 init 0
gkbutt3 init 0
gkbutt4 init 0
gkbutt5 init 0
gktrackpadx init 0
gktrackpady init 0
gkaccelerometerx init 0
gkaccelerometry init 0
gkaccelerometerz init 0
```

Then write an *always-on* instrument that reads each of these control channels into each of those global variables. At the top of the orchestra header:

```
alwayson "Controls"
```

As the next to last instrument in your orchestra:

```
instr Controls
gkslider1 chnget "slider1"
gkslider2 chnget "slider2"
gkslider3 chnget "slider3"
gkslider4 chnget "slider4"
gkslider5 chnget "slider5"
gkslider6 chnget "slider6"
gkslider7 chnget "slider7"
gkslider8 chnget "slider8"
gkslider9 chnget "slider9"
gkbutt1 chnget "butt1"
gkbutt2 chnget "butt2"
gkbutt3 chnget "butt3"
gkbutt4 chnget "butt4"
gkbutt5 chnget "butt5"
gktrackpadx chnget "trackpad.x"
gktrackpady chnget "trackpad.y"
gkaccelerometerx chnget "accelerometerX"
gkaccelerometry chnget "accelerometerY"
gkaccelerometerz chnget "accelerometerZ"
endin
```

So far, everything is common to all pieces. Now, for each specific piece and specific set of instruments, write another *always-on* instrument that will map the controller values to the names and ranges required for your actual instruments. This code, in addition, can make use of the peculiar button widgets, which only signal changes of state and do not report continuously whether they are *on* or *off*. These examples are from *Gogins/Drone-IV.csd*.

At the top of the orchestra header:

```
alwayson "VariablesForControls"
```

As the very last instrument in your orchestra:

```
a instr VariablesForControls
if gkslider1 > 0 then
    gkFirstHarmonic = gkslider1 * 2
    gkgrainDensity = gkslider1 * 400
    gkratio2 = gkslider1 ;1/3
endif
if gkslider2 > 0 then
    gkDistortFactor = gkslider2 * 2
    gkgrainDuration = 0.005 + gkslider2 / 2
    gkindex1 = gkslider2 * 4
endif
if gkslider3 > 0 then
    gkVolume = gkslider3 * 5
    gkgrainAmplitudeRange = gkslider3 * 300
    gkindex2 = gkslider3 ;0.0125
endif
if gkslider4 > 0 then
    gkgrainFrequencyRange = gkslider4 / 10
endif
if gktrackpady > 0 then
    gkDelayModulation = gktrackpady * 2
    ; gkGain = gktrackpady * 2 - 1
endif
if gktrackpadx > 0 then
```

```

gkReverbFeedback = (3/4) + (gktrackpadx / 4)
; gkCenterHz = 100 + gktrackpadx * 3000
endif
kbutt1 trigger gkbutt1, .5, 0
if kbutt1 > 0 then
  gkbritels = gkbritels / 1.5
  gkbritehs = gkbritehs / 1.5
  ; gkQ = gkQ / 2
endif
kbutt2 trigger gkbutt2, .5, 0
if kbutt2 > 0 then
  gkbritels = gkbritels * 1.5
  gkbritehs = gkbritehs * 1.5
  ; gkQ = gkQ * 2
endif
endin

```

Now, the controllers are re-mapped to sensible ranges, and have names that make sense for your instruments. They can be used as follows. Note particularly that, just above the instrument definition, in other words actually in the orchestra header, these global variables are initialized with values that will work in performance, in case the user does not set up the widgets in appropriate positions before starting Csound. This is necessary because the widgets in the Csound for Android app, unlike say the widgets in CsoundQt, do not “remember” their positions and values from performance to performance.

```

gkratio1 init 1
gkratio2 init 1/3
gkindex1 init 1
gkindex2 init 0.0125
instr Phaser
  insno = p1
  istart = p2
  iduration = p3
  ikey = p4
  ivelocity = p5
  iphase = p6
  ipan = p7
  iamp = ampdb(ivelocity) * 8
  iattack = gioverlap
  idecay = gioverlap
  isustain = p3 - gioverlap
  p3 = iattack + isustain + idecay
  kenvelope transeg 0.0, iattack / 2.0, 1.5, iamp / 2.0, iattack / 2.0,
    -1.5, iamp, isustain, 0.0, iamp, idecay / 2.0, 1.5, iamp / 2.0,
    idecay / 2.0, -1.5, 0
  ihertz = cpsmidinn(ikey)
  print insno, istart, iduration, ikey, ihertz, ivelocity, iamp, iphase, ipan
  isine ftgenonce 0,0,65536,10,1
  khertz = ihertz
  ifunction1 = isine
  ifunction2 = isine
  a1,a2 crosspm gkratio1, gkratio2, gkindex1, gkindex2,
    khertz, ifunction1, ifunction2
  aleft, aright pan2 a1+a2, ipan
  adamping linseg 0, 0.03, 1, p3 - 0.1, 1, 0.07, 0
  aleft = adamping * aleft * kenvelope
  aright = adamping * aright * kenvelope
  outleta "outleft", aleft
  outleta "outright", aright
endin

```

12 F. CSOUND AND HASKELL

Csound-expression

Csound-expression is a framework for creation of computer music. It is a Haskell library to ease the use of Csound. It generates Csound files out of Haskell code.

With the help of the library Csound instruments can be created on the fly. A few lines in the interpreter is enough to get cool sound. Some of the features of the library are heavily inspired by reactive programming. Instruments can be evoked with event streams. Event streams can be combined in the manner of reactive programming. The GUI-widgets are producing the event streams as control messages. Moreover with Haskell all standard types and functions like lists, maps and trees can be used. By this, code and data can be organized easily.

One of the great features that comes with the library is a big collection of solid patches which are predefined synthesizers with high quality sound. They are provided with the library csound-catalog.

Csound-expression is an open source library. It's available on Hackage (the main base of Haskell projects).

Key principles

Here is an overview of the features and principles:

- Keep it simple and compact.
- Support for interactive music coding. We can create our sounds in the REPL. So we can chat with our audio engine and can quickly test ideas. It greatly speeds up development comparing to traditional compile-listen style.
- With the library we can create our own libraries. We can create a palette of instruments and use it as a library. It means we can just import the instruments and there is no need for copy and paste and worry for collision of names while pasting. In fact there is a library on hackage that is called csound-catalog. It defines great high quality instruments from the Csound Catalog and other sources.

- Try to hide low level Csound's wiring as much as we can (no IDs for ftables, instruments, global variables). Haskell is a modern language with a rich set of abstractions. The author tried to keep the Csound primitives as close to the Haskell as possible. For example, invocation of the instrument is just an application of the function.
- No distinction between audio and control rates on the type level. Derive all rates from the context. If the user plugs signal to an opcode that expects an audio rate signal the argument is converted to the right rate. Though user can force signal to be of desired type.
- Less typing, more music. Use short names for all types. Make a library so that all expressions can be built without type annotations. Make it simple for the compiler to derive all types. Don't use complex type classes or brainy language concepts.
- Ensure that output signal is limited by amplitude. Csound can produce signals with HUGE amplitudes. Little typo can damage your ears and your speakers. In generated code all signals are clipped by 0dbfs value. 0dbfs is set to 1. Just as in Pure Data. So 1 is absolute maximum value for amplitude.
- Remove score/instrument barrier. Let instrument play a score within a note and trigger other instruments. Triggering the instrument is just an application of the function. It produces the signal as output which can be used in another instrument and so on.
- Set Csound flags with meaningful (well-typed) values. Derive as much as you can from the context. This principle let us start for very simple expressions. We can create our audio signal, apply the function dac to it and we are ready to hear the result in the speakers. No need for XML copy and paste form. It's as easy as typing the line

```
> dac (osc 440)
```

 in the interpreter.
- The standard functions for musical needs. We often need standard waveforms and filters and adsr's. Some functions are not so easy to use in the Csound. So there are a lot of predefined functions that capture lots of musical ideas. the library strives to defines audio DSP primitives in the most basic easiest form.
 - There are audio waves: osc, saw, tri, sqr, pw, ramp, and their unipolar friends (usefull for LFOs).
 - There are filters: lp, hp, bp, br, mlp (moog low pass), filt (for packing several filters in chain), formant filters with ppredefined vowels.
 - There are handy envelopes: fades, fadeOut, fadeln, linseg (with held last value).
 - There noisy functions: white, pink.
 - There are step sequencers: sqrSeq, sawSeq, adsrSeq, and many more. Step sequencer can produce the sequence of unipolar shapes for a given wave-form. The scale factors are defined as the list of values.
- Composable GUIs. Interactive instruments should be easy to make. The GUI widget is a container for signal. It carries an output alongside with visual representation. There are standard ways of composition for the visuals (like horizontal or vertical grouping). It gives us the easy way to combine GUIs. That's how we can create a filtered saw-tooth that is controlled with sliders:

```
> dac $ vlift2 (\cps q -> mlp (100 + 5000 * cps) q (saw 110))  
(uslider 0.5) (uslider 0.5)
```

 The function *uslider* produces slider which outputs a unipolar signal (ranges from 0 to 1).

The single argument is an initial value. The function `vlift2` groups visuals vertically and applies a function of two arguments to the outputs of the sliders. This way we get a new widget that produces the filtered sawtooth wave and contains two sliders. It can become a part of another expression. No need for separate declarations.

- Event streams inspired with FRP (functional reactive programming). Event stream can produce values over time. It can be a metronome click or a push of the button, switch of the toggle button and so on. We have rich set of functions to combine events. We can map over events and filter the stream of events, we can merge two streams, accumulate the result.

That's how we can count the number of clicks:

```
let clicks = lift1 (\evt -> appendE (0 :: D) (+) $ fmap (const 1)
evt) $ button "Click me!"
```

- There is a library that greatly simplifies the creation of the music that is based on samples. It's called `csound-sampler`. With it we can easily create patterns out of wav-files, we can reverse files or play random segments of files.

How to try out the library

To try out the library you need:

- `ghc` - Haskell compiler
- `cabal` – Haskell tool to install open source libraries
- `Csound` - to run the audio

As you install all those tools you can type in the terminal:

```
cabal install csound-catalog --lib
```

It will install `csound-expression` and batteries. If you want just the main library use `csound-expression` instead of `csound-catalog`.

If your `cabal` version is lower than 3.0 version you can skip the flag `--lib`. The version of `cabal` can be checked with:

```
cabal --version
```

After that library is installed and is ready to be used. You can try in the haskell interpreter to import the library and hear the greeting test sound:

```
> ghci
> import Csound.Base
> dac (testDrone3 220)
```

It works and you can hear the sound if you have installed evrything and the system audio is properly configured to work with default `Csound` settings.

Next step to go would be to read through the [tutorial](#). The library covers almost all features of `Csound` so it is as huge as `Csound` but most concepts are easy to grasp and it is driven by compositions of small parts.

Links

The library tutorial: <https://github.com/spell-music/csound-expression/blob/master/tutorial/Index.md>

The library homepage on hackage (it's haskell stock of open source projects): <http://hackage.haskell.org/package/csound-expression>

The library homepage on github: <http://github.com/anton-k/csound-expression/blob/master/tutorial/Index.md>

The csound-sampler library: <http://github.com/anton-k/csound-sampler>

The csound-catalog library homepage on hackage: <http://hackage.haskell.org/package/csound-catalog>

Music made with Haskell and Csound: <http://soundcloud.com/anton-kho>

12 G. CSOUND IN HTML AND JAVASCRIPT

Introduction

Currently it is possible to use Csound together with HTML and JavaScript in at least the following environments:

1. [CsoundQt](#), described in [10 A](#).
2. The Csound for Android app, described in [12 E](#).
3. The [csound.node](#) extension for [NW.js](#).
4. Csound built for WebAssembly, which has two slightly different forms:
 1. The canonical build, described in [10 F](#).
 2. The [csound-extended](#) build.

For instructions on installing any of these environments, please consult the documentation provided in the links mentioned above.

All of these environments provide a JavaScript interface to Csound, which appears as a global Csound object in the JavaScript context of a Web page. Please note, there may be minor differences in the JavaScript interface to Csound between these environments.

With HTML and JavaScript it is possible to define user interfaces, to control Csound, and to generate Csound scores and even orchestras.

In all of these environments, a piece may be written in the form of a Web page (an .html file), with access to a global instance of Csound that exists in the JavaScript context of that Web page. In such pieces, it is common to embed the entire .orc or .csd file for Csound into the .html code as a JavaScript multiline string literal or an invisible TextArea widget.

In CsoundQt and Csound for Android, the HTML code may be embedded in an optional <html> element of the Csound Structured Data (.csd) file. This element essentially defines a Web page that contains Csound, but the host application is responsible for editing the Csound orchestra and running it.

This chapter is organized as follows:

1. Introduction (this section)
2. Tutorial User's Guide

3. Conclusion

HTML must be understood here to represent not only Hyper Text Markup Language, but also all of the other Web standards that currently are supported by Web browsers, Web servers, and the Internet, including cascading style sheets (CSS), HTML5 features such as drawing on a graphics canvas visible in the page, producing animated 3-dimensional graphics with WebGL including shaders and GPU acceleration, Web Audio, various forms of local data storage, Web Sockets, and so on and so on. This whole conglomeration of standards is currently defined and maintained under the non-governmental leadership of the [World Wide Web Consortium](#) (W3C) which in turn is primarily driven by commercial interests belonging to the [Web Hypertext Application Technology Working Group](#) (WHATWG). Most modern Web browsers implement almost all of the W3C standards up to and including HTML5 at an impressive level of performance and consistency. To see what features are available in your own Web browser, go to this [test page](#). All of this stuff is now usable in Csound pieces.

An Example of Use

For an example of a few of the things are possible with HTML in Csound, take a look at the following piece, **Scrim**s, which runs in contemporary Web browsers using a WebAssembly build of Csound and JavaScript code. In fact, it's running right here on this page!

Scrims is a demanding piece, and may not run without dropouts unless you have a rather fast computer. However, it demonstrates a number of ways to use HTML and JavaScript with Csound:

1. Use of the [Three.js](#) library to generate a 3-dimensional animated image of the popcorn fractal.
2. Use of an external JavaScript library, [silencio](#), to sample the moving image and to generate Csound notes from it, that are sent to Csound in real time with the Csound API `csound.readScore` function.
3. Use of a complex Csound orchestra that is embedded in a hidden TextArea on the page.
4. Use of the [dat.gui](#) library to easily create sliders and buttons for controlling the piece in real time.
5. Use of the [jQuery](#) library to simplify handling events from sliders, buttons, and other HTML elements.
6. Use of a TextArea widget as a scrolling display for Csound's runtime messages.

To see this code in action, you can right-click on the piece and select the **Inspect** command. Then you can browse the source code, set breakpoints, print values of variables, and so on.

It is true that LaTeX can do a better job of typesetting than HTML and CSS. It is true that game engines can do a better job for interactive, 3-dimensional computer animation with scene graphs than WebGL. It is true that compiled C or C++ code runs faster than JavaScript. It is true that Haskell is a more fully-featured functional programming language than JavaScript. It is true that MySQL is a more powerful database than HTML5 storage.

But the fact is, there is no single program except for a Web browser that manages to be quite as functional in all of these categories in a way that beginning to intermediate programmers can use, and for which the only required runtime is the Web browser itself.

For this reason alone, HTML makes a very good front end for Csound. Furthermore, the Web standards are maintained in a stable form by a large community of competent developers representing diverse interests. So I believe HTML as a front end for Csound should be quite stable and remain backwardly compatible, just as Csound itself remains backwardly compatible with old pieces.

How it Works

The Web browser embedded into CsoundQt is the [Qt WebEngine](#). The Web browser embedded into Csound for Android is the [WebView](#) available in the [Android SDK](#).

For a .html piece, the front end renders the HTML as a Web page and displays it in an embedded Web browser. The front end injects an instance of Csound into the JavaScript context of the Web.

For a .csd piece, the front end parses the <html> element out of the .csd file. The front end then loads this Web page into its embedded browser, and injects the same instance of Csound that is running the .csd into the JavaScript context of the Web page.

It is important to understand that *any* valid HTML code can be used in Csound's <html> element. It is just a Web page like any other Web page.

In general, the different Web standards are either defined as JavaScript classes and libraries, or glued together using JavaScript. In other words, HTML without JavaScript is dead, but HTML with JavaScript handlers for HTML events and attached to the document elements in the HTML code, comes alive. Indeed, JavaScript can itself define HTML documents by programmatically creating Document Object Model objects.

JavaScript is the engine and the major programming language of the World Wide Web in general, and of code that runs in Web browsers in particular. JavaScript is a standardized language, and it is a functional programming language similar to Scheme. JavaScript also allows classes to be defined by prototypes.

The JavaScript execution context of a Csound Web page contains Csound itself as a *csound* JavaScript object that has at least the following methods:

```
;; [returns a number]
getVersion()
;; [returns the numeric result of the evaluation]
compileOrc(orchestra_code)
evalCode(orchestra_code)
readScore(score_lines)
setControlChannel(channel_name,number)
;; [returns a number representing the channel value]
getControlChannel(channel_name)
message(text)
;; [returns a number]
getSr()
;; [returns a number]
getKsmmps()
;; [returns a number]
getNchnls()
# [returns 1 if Csound is playing, 0 if not]
isPlaying()
```

The front end contains a mechanism for forwarding JavaScript calls in the Web page's JavaScript context to native functions that are defined in the front end, which passes them on to Csound. This

involves a small amount of C++ glue code that the user does not need to know about. In CsoundQt, the glue code uses some JavaScript proxy generator that is injected into the JavaScript context of the Web page, but again, the user does not need to know anything about this.

In the future, more functions from the Csound API will be added to this JavaScript interface, including, at least in some front ends, the ability for Csound to appear as a Node in a Web Audio graph (this already is possible in the Emscripten built of Csound).

Tutorial User Guide

Here we will use CsoundQt to run Csound with HTML.

Let's get started and do a few things in the simplest possible way, in a series of *toots*. All of these pieces are completely contained in unfolding boxes here, from which they can be copied and then pasted into the CsoundQt editor, and some pieces are included as HTML examples in CsoundQt.

1. Display "Hello, World, this is Csound!" in HTML.
2. Create a button that will generate a series of notes based on the logistic equation.
3. Create a slider to set the value of the parameter that controls the degree of chaos produced by iterating the logistic equation, and two other sliders to control the frequency ratio and modulation index of the FM instrument that plays the notes from the logistic equation.
4. Style the HTML elements using a style sheet.

HelloWorld.csd

This is about the shortest CSD that shows some HTML output.

EXAMPLE 12G01_Hello_HTML_World.csd

```
<CsoundSynthesizer>
<CsOptions>
</CsOptions>
<CsInstruments>
sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1
</CsInstruments>
<html>
Hello, World, this is Csound!
</html>
<CsScore>
e 1
</CsScore>
</CsoundSynthesizer>
;example by Michael Gogins
```

Minimal_HTML_Example.csd

This is a simple example that shows how to control Csound using an HTML slider.

EXAMPLE 12G02_Minimal_HTML.csd

```

<CsoundSynthesizer>
<CsOptions>
-odac -d
</CsOptions>
<html>
    <head> </head>
    <body bgcolor="lightblue">
        <script>
            function onGetControlChannel(value) {
                document.getElementById(
                    'testChannel'
                ).innerHTML = value;
            } // to test csound.getControlChannel with QtWebEngine
        </script>
        <h2>Minimal Csound-HTML5 example</h2>
        <br />
        <br />
        Frequency:
        <input
            type="range"
            id="slider"
            oninput='csound.setControlChannel("testChannel",this.value/100.0); '
        />
        <br />
        <button
            id="button"
            onclick='csound.readScore("i 1 0 3")'
        >
            Event
        </button>
        <br /><br />
        Get channel from csound with callback (QtWebchannel):
        <label id="getchannel"></label>
        <button
            onclick='csound.getControlChannel("testChannel", onGetControlChannel)'
        >
            Get</button>
        <br />
        Value from channel "testChannel":
        <label id="testChannel"></label><br />
        <br />
        Get as return value (QtWebkit)
        <button
            onclick='alert("TestChannel: "+csound.getControlChannel("testChannel"))'
        >
            Get as return value
        </button>

        <br />
    </body>
</html>

<CsInstruments>
```

```

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

chnset 0.5, "testChannel" ; to test chnget in the host

instr 1
    kfreq= 200+chnget:k("testChannel")*500
    printk2 kfreq
    aenv linen 1,0.1,p3,0.25
    out poscil(0.5,kfreq)*aenv
endin

; schedule 1,0,0.1, 1

</CsInstruments>
<CsScore>
i 1 0 0.5 ; to hear if Csound is loaded
f 0 3600
</CsScore>
</CsoundSynthesizer>
;example by Tarmo Johannes
;reformatted for flossmanual by Hlödver Sigurdsson

```

Styled_Sliders.csd

And now a more complete example where the user controls both the compositional algorithm, the logistic equation, and the sounds of the instruments. In addition, HTML styles are used to create a more pleasing user interface.

First the entire piece is presented, then the parts are discussed separately.

EXAMPLE 12G03_Extended_HTML.csd

```

<CsoundSynthesizer>
; Example about using CSS in html section of CSD
; By Michael Gogins 2016
; Reformatted for flossmanual by Hlödver Sigurdsson

<CsOptions>
-odac
</CsOptions>
<CsInstruments>

sr = 44100
ksmps = 32
nchnls = 2

iampdbfs init 32768
prints "Default amplitude at 0 dBFS: %9.4f\n", iampdbfs
idbafs init dbamp(iampdbfs)
prints "dbA at 0 dBFS: %9.4f\n", idbafs
iheadroom init 6
prints "Headroom (dB): %9.4f\n", iheadroom
idbaheadroom init idbafs - iheadroom
prints "dbA at headroom: %9.4f\n", idbaheadroom

```

```

iamphedroom init ampdb(idbaheadroom)
prints "Amplitude at headroom:      %9.4f\n", iamphedroom
prints "Balance so the overall amps at the end of performance -6 dbfs.\n"

connect "ModerateFM", "outleft", "Reverberation", "inleft"
connect "ModerateFM", "outright", "Reverberation", "inright"
connect "Reverberation", "outleft", "MasterOutput", "inleft"
connect "Reverberation", "outright", "MasterOutput", "inright"

alwayson "Reverberation"
alwayson "MasterOutput"
alwayson "Controls"

gk_FmIndex init 0.5
gk_FmCarrier init 1

///////////////////////////////
// By Michael Gogins.
/////////////////////////////
instr ModerateFM
    i_instrument = p1
    i_time = p2
    i_duration = p3
    i_midikey = p4
    i_midivelocity = p5
    i_phase = p6
    i_pan = p7
    i_depth = p8
    i_height = p9
    i_pitchclassset = p10
    i_homogeneity = p11
    iattack = 0.002
    isustain = p3
    idecay = 8
    irelease = 0.05
    iHz = cpsmidinn(i_midikey)
    idB = i_midivelocity
    iamplitude = ampdb(idB) * 4.0
    kcarrier = gk_FmCarrier
    imodulator = 0.5
    ifmamplitude = 0.25
    kindx = gk_FmIndex * 20
    ifrequencyb = iHz * 1.003
    kcarrierb = kcarrier * 1.004
    aindenv transeg 0.0, iattack, -11.0, 1.0, idecay, -7.0, 0.025, isustain, \
        0.0, 0.025, irelease, -7.0, 0.0
    aindex = aindenv * kindx * ifmamplitude
    isinetable ftgenonce 0, 0, 65536, 10, 1, 0, .02

    ; ares foscili xamp, kcps, xcar, xmod, kndx, ifn [, iphs]
    aouta foscili 1.0, iHz, kcarrier, imodulator, kindx / 4., isinetable
    aoutb foscili 1.0, ifrequencyb, kcarrierb, imodulator, kindx, isinetable

    ; Plus amplitude correction.
    asignal = (aouta + aoutb) * aindenv
    adeclick linsegr 0, iattack, 1, isustain, 1, irelease, 0
    asignal = asignal * iamplitude
    aoutleft, aoutright pan2 asignal * adeclick, i_pan
    outleta "outleft", aoutleft
    outleta "outright", aoutright
    prints "instr %d t %9.4f d %9.4f k %9.4f v %9.4f p %9.4f\n", \
        p1, p2, p3, p4, p5, p7
endin

```

```

gkReverberationWet init .5
gk_ReverberationDelay init .6

instr Reverberation
ainleft inleta "inleft"
ainright inleta "inright"
aoutleft = ainleft
aoutright = ainright
kdry = 1.0 - gkReverberationWet
awetleft, awetright reverbsc ainleft, ainright, gk_ReverberationDelay, 18000
aoutleft = ainleft * kdry + awetleft * gkReverberationWet
aoutright = ainright * kdry + awetright * gkReverberationWet
outleta "outleft", aoutleft
outleta "outright", aoutright
prints "instr %4d t %9.4f d %9.4f k %9.4f v %9.4f p %9.4f\n", \
      p1, p2, p3, p4, p5, p7
endin

gk_MasterLevel init 1

instr MasterOutput
ainleft inleta "inleft"
ainright inleta "inright"
aoutleft = gk_MasterLevel * ainleft
aoutright = gk_MasterLevel * ainright
outs aoutleft, aoutright
prints "instr %4d t %9.4f d %9.4f k %9.4f v %9.4f p %9.4f\n", \
      p1, p2, p3, p4, p5, p7
endin

instr Controls
gk_FmIndex_ chnget "gk_FmIndex"
if gk_FmIndex_ != 0 then
  gk_FmIndex = gk_FmIndex_
endif

gk_FmCarrier_ chnget "gk_FmCarrier"
if gk_FmCarrier_ != 0 then
  gk_FmCarrier = gk_FmCarrier_
endif

gk_ReverberationDelay_ chnget "gk_ReverberationDelay"
if gk_ReverberationDelay_ != 0 then
  gk_ReverberationDelay = gk_ReverberationDelay_
endif

gk_MasterLevel_ chnget "gk_MasterLevel"
if gk_MasterLevel_ != 0 then
  gk_MasterLevel = gk_MasterLevel_
endif
endin

</CsInstruments>
<CsScore>
</CsScore>
</CsoundSynthesizer>

```

```

<html>
<head> </head>
<body>
  <style type="text/css">
    input[type="range"] {
      -webkit-appearance: none;

```

```

border-radius: 5px;
box-shadow: inset 0 0 5px #333;
background-color: #999;
height: 10px;
width: 100%;
vertical-align: middle;
}
input[type="range"]::-webkit-slider-thumb {
-webkit-appearance: none;
border: none;
height: 16px;
width: 16px;
border-radius: 50%;
background: yellow;
margin-top: -4px;
border-radius: 10px;
}
table td {
border-width: 2px;
padding: 8px;
border-style: solid;
border-color: transparent;
color: yellow;
background-color: teal;
font-family: sans-serif;
}
</style>

<h1>Score Generator</h1>

<script>
var c = 0.99;
var y = 0.5;
function generate() {
csound.message("generate()...\n");
for (i = 0; i < 50; i++) {
var t = i * (1.0 / 3.0);
var y1 = 4.0 * c * y * (1.0 - y);
y = y1;
var key = Math.round(36.0 + y * 60.0);
var note = "i 1 " + t + " 2.0 " + key + " 60 0.0 0.5\n";
csound.readScore(note);
}
}

function on_sliderC(value) {
c = parseFloat(value);
document.querySelector("#sliderCOutput").value = c;
}

function on_sliderFmIndex(value) {
var numberValue = parseFloat(value);
document.querySelector("#sliderFmIndexOutput").value = numberValue;
csound.setControlChannel("gk_FmIndex", numberValue);
}

function on_sliderFmRatio(value) {
var numberValue = parseFloat(value);
document.querySelector("#sliderFmRatioOutput").value = numberValue;
csound.setControlChannel("gk_FmCarrier", numberValue);
}

function on_sliderReverberationDelay(value) {
}

```

```
var numberValue = parseFloat(value);
document.querySelector("#sliderReverberationDelayOutput").value =
    numberValue;
csound.setControlChannel("gk_ReverberationDelay", numberValue);
}

function on_sliderMasterLevel(value) {
    var numberValue = parseFloat(value);
    document.querySelector("#sliderMasterLevelOutput").value = \
        numberValue;
    csound.setControlChannel("gk_MasterLevel", numberValue);
}
</script>

<table>
    <col width="2*" />
    <col width="5*" />
    <col width="100px" />
    <tr>
        <td>
            <label for="sliderC">c</label>
        </td>
        <td>
            <input
                type="range"
                min="0"
                max="1"
                value=".5"
                id="sliderC"
                step="0.001"
                oninput="on_sliderC(value)"
            />
        </td>
        <td>
            <output for="sliderC" id="sliderCOutput">.5</output>
        </td>
    </tr>
    <tr>
        <td>
            <label for="sliderFmIndex">Frequency modulation index</label>
        </td>
        <td>
            <input
                type="range"
                min="0"
                max="1"
                value=".5"
                id="sliderC"
                step="0.001"
                oninput="on_sliderFmIndex(value)"
            />
        </td>
        <td>
            <output for="sliderFmIndex" id="sliderFmIndexOutput">.5</output>
        </td>
    </tr>
    <tr>
        <td>
            <label for="sliderFmRatio">Frequency modulation ratio</label>
        </td>
        <td>
            <input
                type="range"

```

```
        min="0"
        max="1"
        value=".5"
        id="sliderFmRatio"
        step="0.001"
        oninput="on_sliderFmRatio(value)"
    />
</td>
<td>
    <output for="sliderFmRatio" id="sliderFmRatioOutput">.5</output>
</td>
</tr>
<tr>
    <td>
        <label for="sliderReverberationDelay">Reverberation delay</label>
    </td>
    <td>
        <input
            type="range"
            min="0"
            max="1"
            value=".5"
            id="sliderReverberationDelay"
            step="0.001"
            oninput="on_sliderReverberationDelay(value)"
        />
    </td>
    <td>
        <output
            for="sliderReverberationDelay"
            id="sliderReverberationDelayOutput"
            >.5</output>
        >
    </td>
</tr>
<tr>
    <td>
        <label for="sliderMasterLevel">Master output level</label>
    </td>
    <td>
        <input
            type="range"
            min="0"
            max="1"
            value=".5"
            id="sliderMasterLevel"
            step="0.001"
            oninput="on_sliderMasterLevel(value)"
        />
    </td>
    <td>
        <output for="sliderMasterLevel" id="sliderMasterLevelOutput"
            >.5
        </output>
    </td>
</tr>
<tr>
    <td>
        <button onclick="generate()">Generate score</button>
    </td>
</tr>
</table>
```

```
</body>
</html>
```

Here I have introduced a simple Csound orchestra consisting of a single frequency modulation instrument feeding first into a reverberation effect, and then into a master output unit. These are connected using the signal flow graph opcodes. The actual orchestra is of little interest here.

Generating the Score

This piece has no score, because the score will be generated at run time. In the `<html>` element, I also have added this button:

```
<button onclick="generate()"> Generate score </button>
```

When this button is clicked, it calls a JavaScript function that uses the logistic equation, which is a simple quadratic dynamical system, to generate a Csound score from a chaotic attractor of the system. This function also is quite simple. Its main job, aside from iterating the logistic equation a few hundred times, is to translate each iteration of the system into a musical note and send that note to Csound to be played using the Csound API function `readScore()`. So the following `<script>` element is added to the body of the `<html>` element:

```
<script>
  var c = 0.99;
  var y = 0.5;
  function generate() {
    csound.message("generate()...\n");
    for (i = 0; i < 200; i++) {
      var t = i * (1.0 / 3.0);
      var y1 = 4.0 * c * y * (1.0 - y);
      y = y1;
      var key = Math.round(36.0 + y * 60.0);
      var note = "i 1 " + t + " 2.0 " + key + " 60 0.0 0.5\n";
      csound.readScore(note);
    }
  }
</script>
```

Adding Sliders

The next step is to add more user control to this piece. We will enable the user to control the attractor of the piece by varying the constant `c`, and we will enable the user to control the sound of the Csound orchestra by varying the frequency modulation index, frequency modulation carrier ratio, reverberation time, and master output level.

This code is demonstrated on a low level, so that you can see all of the details and understand exactly what is going on. A real piece would most likely be written at a higher level of abstraction, for example by using a third party widget toolkit, such as jQuery UI.

A slider in HTML is just an `input` element like this:

```
<input
  id="sliderC"
  type="range"
  min="0"
  max="1"
  value=".5"
  step="0.001"
```

```
    oninput="on_sliderC(value)"
  />
```

This element has attributes of minimum value 0, maximum value 1, which normalizes the user's possible values between 0 and 1. This could be anything, but in many musical contexts, for example VST plugins, user control values are always normalized between 0 and 1. The tiny step attribute simply approximates a continuous range of values.

The most important thing is the oninput attribute, which sets the value of a JavaScript event handler for the oninput event. This function is called whenever the user changes the value of the slider.

For ease of understanding, a naming convention is used here, with *sliderC* being the basic name and other names of objects associated with this slider taking names built up by adding prefixes or suffixes to this basic name.

Normally a slider has a label, and it is convenient to show the actual numerical value of the slider. This can be done like so:

```
<table>
  <col width="2*" />
  <col width="5*" />
  <col width="100px" />
  <tr>
    <td>
      <label for="sliderC">c</label>
    </td>

    <td>
      <input
        type="range"
        min="0"
        max="1"
        value=".5"
        id="sliderC"
        step="0.001"
        oninput="on_sliderC(value)"
      />
    </td>

    <td>
      <output for="sliderC" id="sliderCOutput">.5</output>
    </td>
  </tr>
</table>
```

If the slider, its label, and its numeric display are put into an HTML table, that table will act like a layout manager in a standard widget toolkit, and will resize the contained elements as required to get them to line up.

For this slider, the JavaScript handler is:

```
function on_sliderC(value) {
  c = parseFloat(value);
  document.querySelector("#sliderCOutput").value = c;
}
```

The variable *c* was declared at global scope just above the generate() function, so that variable is accessible within the *on_sliderC* function.

Keep in mind, if you are playing with this code, that a new value of c will only be heard when a new score is generated.

Very similar logic can be used to control variables in the Csound orchestra. The value of the slider has to be sent to Csound using the channel API, like this:

```
function on_sliderFmIndex(value) {
    var numberValue = parseFloat(value);
    document.querySelector("#sliderFmIndexOutput").value = numberValue;
    csound.setControlChannel("gk_FmIndex", numberValue);
}
```

Then, in the Csound orchestra, that value has to be retrieved using the chnget opcode and applied to the instrument to which it pertains. It is most efficient if the variables controlled by channels are global variables declared just above their respective instrument definitions. The normalized values can be rescaled as required in the Csound instrument code.

```
gk_FmIndex init 0.5
instr ModerateFM
...
kindex = gk_FmIndex * 20
...
endin
```

Also for the sake of efficiency, a global, always-on instrument can be used to read the control channels and assign their values to these global variables:

```
instr Controls
gk_FmIndex_ chnget "gk_FmIndex"
if gk_FmIndex_ != 0 then
    gk_FmIndex = gk_FmIndex_
endif
gk_FmCarrier_ chnget "gk_FmCarrier"
if gk_FmCarrier_ != 0 then
    gk_FmCarrier = gk_FmCarrier_
endif
gk_ReverberationDelay_ chnget "gk_ReverberationDelay"
if gk_ReverberationDelay_ != 0 then
    gk_ReverberationDelay = gk_ReverberationDelay_
endif
gk_MasterLevel_ chnget "gk_MasterLevel"
if gk_MasterLevel_ != 0 then
    gk_MasterLevel = gk_MasterLevel_
endif
endin
```

Note that each actual global variable has a default value, which is only overridden if the user actually operates its slider.

Customizing the Style

The default appearance of HTML elements is brutally simple. But each element has attributes that can be used to change its appearance, and these offer a great deal of control.

Of course, setting for example the font attribute for each label on a complex HTML layout is tedious. Therefore, this example shows how to use a style sheet. We don't need much style to get a much improved appearance:

```

<style type="text/css">
  input[type="range"] {
    -webkit-appearance: none;
    border-radius: 5px;
    box-shadow: inset 0 0 5px #333;
    background-color: #999;
    height: 10px;
    width: 100%;
    vertical-align: middle;
  }
  input[type="range"]::-webkit-slider-thumb {
    -webkit-appearance: none;
    border: none;
    height: 16px;
    width: 16px;
    border-radius: 50%;
    background: yellow;
    margin-top: -4px;
    border-radius: 10px;
  }
  table td {
    border-width: 2px;
    padding: 8px;
    border-style: solid;
    border-color: transparent;
    color: yellow;
    background-color: teal;
    font-family: sans-serif;
  }
</style>

```

This little style sheet is generic, that is, it applies to every element on the HTML page. It says, for example, that `table td` (table cells) are to have a yellow sans-serif font on a teal background, and this will apply to every table cell on the page. Style sheets can be made more specialized by giving them names. But for this kind of application, that is not usually necessary.

Conclusion

Most, if not all all, of the functions performed by other Csound front ends could be encompassed by HTML and JavaScript. However, there are a few gotchas. For CsoundQt and other front ends based on Chrome, there may be extra latency and processing overhead required by inter-process communications. For Emscripten and other applications that use Web Audio, there may also be additional latency.

Obviously, much *more* can be done with HTML, JavaScript, and other Web standards found in contemporary Web browsers. Full-fledged, three-dimensional, interactive, multi-player computer games are now being written with HTML and JavaScript. Other sorts of Web applications also are being written this way.

Sometimes, JavaScript is embedded into an application for use as a scripting language. The Csound front ends discussed here are examples, but there are others. For example, Max for Live can be programmed in JavaScript, and so can the open source score editor MuseScore. In fact, in MuseScore, JavaScript can be used to algorithmically generate notated scores.

13 A. DEVELOPING PLUGIN OPCODES

Csound is possibly one of the most easily extensible of all modern music programming languages. The addition of unit generators (opcodes) and function tables is generally the most common type of extension to the language. This is possible through two basic mechanisms: user-defined opcodes (UDOs), written in the Csound language itself and pre-compiled/binary opcodes, written in C or C++.¹

To facilitate the latter case, Csound offers a simple opcode development API, from which dynamically-loadable, or *plugin* unit generators can be built. A similar mechanism for function tables is also available. For this we can use either the C++ or the C languages. C++ opcodes are written as classes derived from a template (“pseudo-virtual”) base class OpcodeBase. In the case of C opcodes, we normally supply a module according to a basic description. The sections on plugin opcodes will use the C language. For those interested in object-oriented programming, alternative C++ class implementations for the examples discussed in this text can be extrapolated from the original C code.

You may find additional information and examples at Csound’s [Opcode SDK repository](#).

Csound data types and signals

The Csound language provides four basic data types: i-, k-, a- and f-types (there is also a fifth type, w, which will not be discussed here). These are used to pass the data between opcodes, each opcode input or output parameter relating to one of these types. The Csound i-type variable is used for initialisation variables, which will assume only one value in performance. Once set, they will remain constant throughout the instrument or UDO code, unless there is a reinitialisation pass. In a plugin opcode, parameters that receive i-type variables are set inside the initialisation part of the code, because they will not change during processing.

The other types are used to hold scalar (k-type), vectorial (a-type) and spectral-frame (f) signal variables. These will change in performance, so parameters assigned to these variables are set and modified in the opcode processing function. Scalars will hold a single value, whereas vectors hold an array of values (a vector). These values are floating-point numbers, either 32- or 64-bit, depending on the executable version used, defined in C/C++ as a custom MYFLT type.

¹For UDOs compare chapter [03 G](#)

Plugin opcodes will use pointers to input and output parameters to read and write their input/output. The Csound engine will take care of allocating the memory used for its variables, so the opcodes only need to manipulate the pointers to the addresses of these variables.

A Csound instrument code can use any of these variables, but opcodes will have to accept specific types as input and will generate data in one of those types. Certain opcodes, known as polymorphic opcodes, will be able to cope with more than one type for a specific parameter (input or output). This generally implies that more than one version of the opcode will have to be implemented, which will be called depending on the parameter types used.

Plugin opcodes

Originally, Csound opcodes could only be added to the system as statically-linked code. This required that the user recompiled the whole Csound code with the added C module. The introduction of a dynamic-loading mechanism has provided a simpler way for opcode addition, which only requires the C code to be compiled and built as a shared, dynamic library. These are known in Csound parlance as plugin opcodes and the following sections are dedicated to their development process.

Anatomy of an opcode

The C code for a Csound opcode has three main programming components: a *data structure* to hold the internal data, an *initialising function* and a *processing function*. From an object-oriented perspective, an opcode is a simple class, with its attributes, constructor and perform methods. The data structure will hold the attributes of the class: input/output parameters and internal variables (such as delays, coefficients, counters, indices etc.), which make up its dataspace.

The constructor method is the initialising function, which sets some attributes to certain values, allocates memory (if necessary) and anything that is needed for an opcode to be ready for use. This method is called by the Csound engine when an instrument with its opcodes is allocated in memory, just before performance, or when a reinitialisation is required.

Performance is implemented by the processing function, or perform method, which is called when new output is to be generated. This happens at every control period, or *ksmps* samples. This implies that signals are generated at two different rates: the control rate, *kr*, and the audio rate, *sr*, which is $kr * ksmgs$ samples/sec. What is actually generated by the opcode, and how its perform method is implemented, will depend on its input and output Csound language data types.

Opcoding basics

C-language opcodes normally obey a few basic rules and their development require very little in terms of knowledge of the actual processes involved in Csound. Plugin opcodes will have to provide the three main programming components outlined above: a data structure plus the initialisa-

tion and processing functions. Once these elements are supplied, all we need to do is to add a line telling Csound what type of opcode it is, whether it is an i-, k- or a-rate based unit generator and what arguments it takes.

The data structure will be organised in the following fashion:

1. The OPDS data structure, holding the common components of all opcodes.
2. The output pointers (one MYFLT pointer for each output)
3. The input pointers (as above)
4. Any other internal dataspace member.

The Csound opcode API is defined by *csdl.h*, which should be included at the top of the source file. The example below shows a simple data structure for an opcode with one output and three inputs, plus a couple of private internal variables:

```
#include "csdl.h"

typedef struct _newopc {
    OPDS h;
    MYFLT *out; /* output pointer */
    MYFLT *in1,*in2,*in3; /* input pointers */
    MYFLT var1; /* internal variables */
    MYFLT var2;
} newopc;
```

Initialisation

The initialisation function is only there to initialise any data, such as the internal variables, or allocate memory, if needed. The plugin opcode model in Csound 6 expects both the initialisation function and the perform function to return an int value, either OK or NOTOK. Both methods take two arguments: pointers to the CSOUND data structure and the opcode dataspace. The following example shows an example initialisation function. It initialises one of the variables to 0 and the other to the third opcode input parameter.

```
int newopc_init(CSOUND *csound, newopc *p){
    p->var1 = (MYFLT) 0;
    p->var2 = *p->in3;
    return OK;
}
```

Control-rate performance

The processing function implementation will depend on the type of opcode that is being created. For control rate opcodes, with k- or i-type input parameters, we will be generating one output value at a time. The example below shows an example of this type of processing function. This simple

example just keeps ramping up or down depending on the value of the second input. The output is offset by the first input and the ramping is reset if it reaches the value of `var2` (which is set to the third input argument in the constructor above).

```
int newopc_process_control(CSOUND *csound, newopc *p){
    MYFLT cnt = p->var1 + *(p->in2);
    if(cnt > p->var2) cnt = (MYFLT) 0; /* check bounds */
    *(p->out) = *(p->in1) + cnt; /* generate output */
    p->var1 = cnt; /* keep the value of cnt */
    return OK;
}
```

Audio-rate performance

For audio rate opcodes, because it will be generating audio signal vectors, it will require an internal loop to process the vector samples. This is not necessary with k-rate opcodes, because, as we are dealing with scalar inputs and outputs, the function has to process only one sample at a time. If we were to make an audio version of the control opcode above (disregarding its usefulness), we would have to change the code slightly. The basic difference is that we have an audio rate output instead of control rate. In this case, our output is a whole vector (a MYFLT array) with `ksmps` samples, so we have to write a loop to fill it. It is important to point out that the control rate and audio rate processing functions will produce exactly the same result. The difference here is that in the audio case, we will produce `ksmps` samples, instead of just one sample. However, all the vector samples will have the same value (which actually makes the audio rate function redundant, but we will use it just to illustrate our point).

Another important thing to consider is to support the `-sample-accurate` mode introduced in Csound 6. For this we will need to add code to start processing at an offset (when this is given), and finish early (if that is required). The opcode will then lookup these two variables (called `offset` and `early`) that are passed to it from the container instrument, and act to ensure these are taken into account. Without this, the opcode would still work, but not support the sample-accurate mode.

```
int newopc_process_audio(CSOUND *csound, newopc *p){
    int i, n = CS_KSMPS;
    MYFLT *aout = p->out; /* output signal */
    MYFLT cnt = p->var1 + *(p->in2);
    uint32_t offset = p->h.insdshead->ksmps_offset;
    uint32_t early = p->h.insdshead->ksmps_no_end;

    /* sample-accurate mode mechanism */
    if(offset) memset(aout, '\0', offset*sizeof(MYFLT));
    if(early) {
        n -= early;
        memset(&aout[n], '\0', early*sizeof(MYFLT));
    }

    if(cnt > p->var2) cnt = (MYFLT) 0; /* check bounds */

    /* processing loop */
    for(i=offset; i < n; i++) aout[i] = *(p->in1) + cnt;
    p->var1 = cnt; /* keep the value of cnt */
    return OK;
}
```

```
}
```

In order for Csound to be aware of the new opcode, we will have to register it. This is done by filling an opcode registration structure OENTRY array called *localops* (which is static, meaning that only one such array exists in memory at a time):

```
static OENTRY localops[] = {
{ "newopc", sizeof(newopc), 0, 7, "s", "kki", (SUBR) newopc_init,
(SUBR) newopc_process_control, (SUBR) newopc_process_audio }
};
```

Linkage

The OENTRY structure defines the details of the new opcode:

1. the opcode name (a string without any spaces).
2. the size of the opcode dataspace, set using the macro S(struct_name), in most cases; otherwise this is a code indicating that the opcode will have more than one implementation, depending on the type of input arguments (a polymorphic opcode).
3. Flags to control multicore operation (0 for most cases).
4. An int code defining when the opcode is active: 1 is for i-time, 2 is for k-rate and 4 is for a-rate. The actual value is a combination of one or more of those. The value of 7 means active at i-time (1), k-rate (2) and a-rate (4). This means that the opcode has an init function, plus a k-rate and an a-rate processing functions.
5. String definition the output type(s): a, k, s (either a or k), i, m (multiple output arguments), w or f (spectral signals).
6. Same as above, for input types: a, k, s, i, w, f, o (optional i-rate, default to 0), p (opt, default to 1), q (opt, 10), v(opt, 0.5), j(opt, -1), h(opt, 127), y (multiple inputs, a-type), z (multiple inputs, k-type), Z (multiple inputs, alternating k- and a-types), m (multiple inputs, i-type), M (multiple inputs, any type) and n (multiple inputs, odd number of inputs, i-type).
7. I-time function (init), cast to (SUBR).
8. K-rate function.
9. A-rate function.

Since we have defined our output as "s", the actual processing function called by csound will depend on the output type. For instance

```
k1 newopc kin1, kin2, i1
```

will use *newopc_process_control()*, whereas

```
a1 newopc kin1, kin2, i1
```

will use *newopc_process_audio()*. This type of code is found for instance in the oscillator opcodes, which can generate control or audio rate (but in that case, they actually produce a different output for each type of signal, unlike our example).

Finally, it is necessary to add, at the end of the opcode C code the LINKAGE macro, which defines some functions needed for the dynamic loading of the opcode.

Building opcodes

The plugin opcode is build as a dynamic module. All we need is to build the opcode as a dynamic library, as demonstrated by the examples below.

On OSX:

```
gcc -O2 -dynamiclib -o myopc.dylib opsrc.c -DUSE_DOUBLE  
-I/Library/Frameworks/CsoundLib64.framework/Headers
```

Linux:

```
gcc -O2 -shared -o myopc.so -fPIC opsrc.c -DUSE_DOUBLE  
-I<path to Csound headers>
```

Windows (MinGW+MSYS):

```
gcc -O2 -shared -o myopc.dll opsrc.c -DUSE_DOUBLE  
-I<path to Csound headers>
```

CSD Example

To run Csound with the new opcodes, we can use the `--opcode-lib=libname` option.

EXAMPLE 13A01_newop.csd

```
<CsoundSynthesizer>  
<CsOptions>  
--opcode-lib=newopc.so ; OSX: newopc.dylib; Windows: newopc.dll  
</CsOptions>  
<CsInstruments>  
  
schedule 1,0,100,440  
  
instr 1  
  
asig  newopc  0, 0.001, 1  
ksig  newopc  1, 0.001, 1.5  
aosc   oscili 1000, p4*ksig  
       out aosc*asig  
  
endin  
  
</CsInstruments>  
</CsoundSynthesizer>  
;example by victor lazzarini
```

14 A. OPCODE GUIDE

If Csound is called from the command line with the option -z, a list of all opcodes is printed. The total number of all opcodes is more than 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 guide is another attempt to provide some orientation within Csound's wealth of opcodes — a wealth which is often frightening for beginners and still overwhelming for experienced users.

Three selections are given here, each larger than the other:

1. The **33 Most Essential Opcodes**. This selection might be useful for beginners. Learning ten opcodes a day, Csound can be learned in three days, and many full-featured Csound programs can be written with these 33 opcodes.
2. The **Top 100 Opcodes**. Adding 67 more opcodes to the first collection pushes the csound programmer to the next level. This should be sufficient for doing most of the jobs in Csound.
3. The third overview is rather extended already, and follows mostly the classification in the Csound Manual. It comprises nearly **500** opcodes.

Although these selections come from some experience in using and teaching Csound, they must remain subjective, as working in Csound can go in quite different directions.

33 ESSENTIAL OPCODES

Oscillators

[poscil\(3\)](#) — high precision oscillator with linear (cubic) interpolation

[vco\(2\)](#) — analog modelled oscillator

Noise and Random

[rand](#) — standard random (noise) generator

[random](#) — random numbers between min/max

`randomi/randomh` – random numbers between min/max with interpolating or hold segments
`seed` – set the global seed

Envelopes

`linen(r)` – linear fade in/out

Line Generators

`linseg(r)` – one or more linear segments
`transeg(r)` – one or more user-definable segments

Line Smooth

`sc_lag(ud)` – exponential lag (with different smoothing times) (traditional alternatives are `port(k)` and `tonek`)

Sound Files / Samples

`diskin` – sound file read/playback with different options

Audio I/O

`inch` – read audio from one or more input channel
`out` – write audio to one or more output channels (starting from first hardware output)

Control

`if` – if clause
`changed(2)` – k-rate signal change detector

Instrument Control

`schedule(k)` – perform instrument event
`turnoff(2)` – turn off this or another instrument

Time

`metro(2)` – trigger metronome

Software Channels

`chnset/chnget` – set/get value in channel

MIDI

`massign` – assign MIDI channel to Csound instrument

`notnum` – note number received

`veloc` – velocity received

Key

`sensekey` – sense computer keyboard

Panning

`pan2` – stereo panning with different options

Reverb

`reverbsc` – stereo reverb after Sean Costello

Delay

`vdelayx` – variable delay with highest quality interpolation

Distortion

`distort(1)` – distortion via waveshaping

Filter

`butbp(hp/lp)` – second order butterworth filter

Level

`rms` – RMS measurement

`balance(2)` – adjust audio signal level according to comparator

Math / Conversion

`ampdb/dbamp` – dB to/from amplitude

`mtof/ftom` – MIDI note number to/from frequency

Print

`print(k)` – print i/k-values

TOP 100 OPCODES

Oscillators / Phasors

`poscil(3)` – high precision oscillator with linear (cubic) interpolation

`vco(2)` – analog modelled oscillator

`(g)buzz` – buzzer

`mpulse` – single sample impulses

`phasor` – standard phasor

Noise and Random

`rand` – standard random (noise) generator

`(bi)rnd` – simple unipolar (bipolar) random generator

`random` – random numbers between min/max

`randomi/randomh` – random numbers between min/max with interpolation or hold numbers

`seed` – set the global seed

Envelopes

`linen(r)` – linear fade in/out
`(m)adsr` – traditional ADSR envelope

Line Generators

`linseg(r)` – one or more linear segments
`expseg(r)` – one or more exponential segments
`cosseg` – one or more cosine segments
`transeg(r)` – one or more user-definable segments

Line Smooth

`sc_lag(ud)` – exponential lag (with different smoothing times)

Sound Files / Samples

`diskin` – sound file read/playback with different options
`mp3in` – mp3 read/playback
`loscil(3/x)` – read sampled sound from a table
`flooper(2)` – crossfading looper
`filescal/mincer` – phase-locked vocoder processing with time and pitch scale
`filelen` – length of sound file

Audio I/O

`inch` – read audio from one or more input channels
`out` – write audio to one or more output channels (starting from first hardware output)
`outch` – write audio to arbitrary output channel(s)
`monitor` – monitor audio output channels

Tables (Buffers)

`ftgen` – create any table with a GEN subroutine
`table(i/3)` – read from table (with linear/cubic interpolation)

`tablew` – write data to a table

`ftsamplebank` – load files in a directory to tables

Arrays

`fillarray` – fill array with values

`lenarray` – length of array

`getrow/getcol` – get a row/column from a two-dimensional array

`setrow/setcol` – set a row/column of a two-dimensional array

Program Control

`if` – if clause

`while` – while loop

`changed(2)` – k-rate signal change detector

`trigger` – threshold trigger

Instrument Control

`active` – number of active instrument instances

`maxalloc` – set maximum number of instrument instances

`schedule(k)` – perform instrument event

`turnoff(2)` – turn off this or another instrument

`nstrnum` – number of a named instrument

Time

`metro(2)` – trigger metronome

`timeinsts` – time of instrument instance in seconds

Software Channels

`chnget/chnset` – get/set value from/to channel

`chnmix/chnclear` – mix value to channel / clear channel

MIDI

`massign` – assign MIDI channel to Csound instrument
`notnum` – note number received
`veloc` – velocity received
`ctrl7(14/21)` – receive controller

OSC

`OSClisten` – receive messages
`OSCraw` – listens to all messages
`OSCsend` – send messages

Key

`sensekey` – sense computer keyboard

Panning / Spatialization

`pan2` – stereo panning with different options
`vbap` – vector base amplitude panning for multichannel (also 3d)
`bformenc1/bformdec1` – B-format encoding/decoding

Reverb

`freeverb` – stereo reverb after Jezar
`reverbsc` – stereo reverb after Sean Costello

Spectral Processing

`pvsanal` – spectral analysis with audio signal input
`pvstanal` – spectral analysis from sampled sound
`pvsynth` – spectral resynthesis
`pvscale` – scale frequency components (pitch shift)
`pvsmorph` – morphing between two f-signals
`pvsftw/pvsftr` – write/read amplitude and/or frequency data to/from tables
`pvs2array/pvsfromarray` – write/read spectral data to/from arrays

Convolution

pconvolve – partitioned convolution

Granular Synthesis

partikkel – complete granular synthesis

Physical Models

pluck – plucked string (Karplus-Strong) algorithm

Delay

vdelayx – variable delay with highest quality interpolation

(v)comb – comb filter

Distortion

distort(1) – distortion via waveshaping

powershape – waveshaping by raising to a variable exponent

Filter

(a)tone – first order IIR low (high) pass filter

reson – second order resonant filter

butbp(hp/lp) – second order butterworth filter

mode – mass-spring system modelled

zdf_ladder – zero delay feedback implementation of 4 pole ladder filter

Level

rms – RMS measurement

balance(2) – adjust audio signal level according to comparator

Math / Conversion

`ampdb/dbamp` – dB to/from amplitude
`mtof/ftom` – MIDI note number to/from frequency
`cent` – cent to scaling factor
`log2` – return 2 base log
`abs` – absolute value
`int/frac` – integer/fractional part
`linlin` – signal scaling

Amplitude / Pitch Tracking

`follow(2)` – envelope follower
`ptrack` – pitch tracking using STFT

Print

`print(k)` – print i/k-values
`printarray` – print array
`ftprint` – print table

File IO

`fout` – write out real-time audio output (*for rendered audio file output see chapter 02E and 06A*)
`ftsave(k)` – save table(s) to text file or binary
`fprint(k)s` – formatted printing to file
`readf(i)` – reads an external file line by line
`directory` – files in a directory as string array

Signal Type Conversion

`i(k) / k(a) / a(k)` – i-value from k-signal / k-signal from a-signal / a-signal from k-signal

EXTENDED OPCODE OVERVIEW IN CATEGORIES

I. AUDIO I/O AND SOUND FILES

AUDIO I/O

General Settings and Queries Note that modern *Csound* frontends handle most of the Audio I/O settings. For command line usage, see [this](#) section in the *Csound Options*.

`sr` – set sample rate (default=44100)
`ksmps` – set block (audio vector) size (default=10) (*setting to power-of-two (e.g. 32/64/128) is recommended*)
`nchnls` – set number of I/O channels (default=1)
`nchnls_i` – set number of input channels if different from output
`0dbfs` – set zero dB full scale (default=32767) (*setting 0dbfs=1 is strongly recommended*)
`nchnls_hw` – report number of channels in hardware
`setksmps` – set local ksmgs in User-Defined-Opcodes or instruments

Signal Input and Output `inch` – read audio from one or more input channels
`out` – write audio to one or more output channels (starting from first hardware output)
`outch` – write audio to arbitrary output channel(s)
`monitor` – monitor audio output channels

SOUND FILES AND SAMPLES

Sound File Playback `diskin` – sound file read/playback with different options
`mp3in` – mp3 read/playback

Sample Playback (GEN01) – load file into table
`loscil(3/x)` – read sampled sound from a table
`lposcil` – read sampled sound with loops
`flooper(2)` – crossfading looper

Time Stretch and Pitch Shift `filescal` – phase-locked vocoder processing with time and pitch scale
`mincer` – phase-locked vocoder processing on table loaded sound
`mp3scal` – tempo scaling of mp3 files
`paulstretch` – extreme time stretch
`sndwarp(st)` – granular-based time and pitch modification

NOTE that any granular synthesis opcode and some of the pvs opcodes (pvstanal, pvsbufred) can also be used for this approach

Soundfonts and Fluid Opcodes see overview [here](#)

Sound File Queries `filelen` – length of sound file
`filesr` – sample rate of sound file
`filenchnls` – number of channels in sound file

`filepeak` – peak in sound file
`filebit` – bit depth in sound file
`filevalid` – check whether file exists
`mp3len` – length of mp3 file

Directories `directory` – files in a directory as string array
`ftsamplebank` – load files in a directory to tables

Sound File Output `fout` – write out real-time audio output (*for rendered audio file output see chapter 02E and 06A*)

II. SIGNAL GENERATORS

OSCILLATORS AND PHASORS

Standard Oscillators `poscil(3)` – high precision oscillator with linear (cubic) interpolation
`oscili(3)` – standard oscillator with linear (cubic) interpolation
`lfo` – low frequency oscillator of various shapes
`oscilikt` – interpolating oscillator with k-rate changeable tables
`more ...` – more standard oscillators ...

Note: `oscil` is not recommended as it has integer indexing which can result in low quality

Dynamic Spectrum Oscillators `(g)buzz` – buzzer
`mpulse` – single sample impulses
`vco(2)` – analog modelled oscillator
`squinewave` – shape-shifting oscillator with hardsync

Phasors `phasor` – standard phasor
`syncphasor` – phasor with sync I/O
`ephasor` – phasor with additional exponential decay output
`sc_phasor` – resettable phasor

RANDOM AND NOISE GENERATORS

Seed `seed` – set the global seed
`getseed` – get the global seed

Noise Generators `rand` – standard random (noise) generator
`pinker` – pink noise after Stefan Stenzel
`pinkish` – pink noise generator
`fractalnoise` – fractal noise generator
`gauss(i)` – Gaussian distribution random generator
`gendy(c/x)` – dynamic stochastic waveform synthesis conceived by Iannis Xenakis

General Random Generators `rnd` – simple unipolar random generator
`birnd` – simple bipolar random generator
`random` – random numbers between min/max
`rnd31` – random generator with controllable distributions
`dust(2)` – random impulses
`gausstrig` – random impulses around a frequency
`lorenz` – implements lorenz system of equations
`urd` – user-defined random distributions

Random Generators with Interpolating or Hold Numbers `randi(c)` – bipolar random generator with linear (cubic) interpolation
`randh` – bipolar random generator with hold numbers
`randomi` – random numbers between min/max with interpolation
`randomh` – random numbers between min/max with hold numbers
`more ...` – more random generators ...

ENVELOPES AND LINES

Simple Standard Envelopes `linen` – linear fade in/out
`linenr` – fade out at release
`(x)adsr` – ADSR envelope with linear (exponential) lines
`m(x)adsr` – ADSR for MIDI notes with linear (exponential) lines
`more` – more standard envelopes ...

Envelopes by Linear and Exponential Generators

`linseg` – one or more linear segments
`expseg` – one or more exponential segments
`transeg` – one or more user-definable segments
`linsegr` – linear segments with final release segment
`expsegr` – exponential segments with release
`transegr` – user-definable segments with release
`bpf` – break point function with linear interpolation
`jitter(2)` – randomly segmented line
`jspline` – jitter-spline generated line
`loopseg` – loops linear segments
`rspline` – random spline curves
`more` – more envelope generators ...

Signal Smooth `port(k)` – portamento-like smoothing for control signals (with variable half-time)
`sc_lag(u)` – exponential lag (with different smoothing times)
`(t)lineto` – generate glissando from control signal

PHYSICAL MODELS AND FM INSTRUMENTS

Waveguide Physical Modelling see [here](#)
and [here](#)

Frequency Modulation `foscili` – basic FM oscillator
`cross(p/f)m(i)` – two mutually frequency and/or phase modulated oscillators (see also chapter 04D)

FM Instrument Models see [here](#)

III. SIGNAL MODIFIERS

DELAYS

Audio Delays `delay` – simple constant audio delay
`vdelay(3)` – variable delay with linear (cubic) interpolation
`vdelayx` – variable delay with highest quality interpolation
`vdelayxw` – variable delay changing write rather than read position
`delayr` – establishes delay line and read from it
`delayw` – write into delay line
`deltapxw` – write into a delay line with high quality interpolation
`deltap(i/3)` – tap a delay line with linear (cubic) interpolation
`deltapx` – tap a delay line with highest quality interpolation
`deltapn` – tap a delay line at variable offsets
`multitap` – multiple tap delays with different gains
`(v)comb` – comb filter

Control Signal Delays `delayk` – simple constant delay for k-signals
`vdel_k` – variable delay for k-signals

FILTERS

Compare the extensive [Standard Filters](#) and [Specialized Filters](#) overviews in the Csound Manual.

Low Pass Filters `tone` – first order IIR filter
`tonex` – serial connection of several tone filters
`butlp` – second order IIR filter
`clfilt` – adjustable types and poles

High Pass Filters `atone` – first order IIR filter
`atonex` – serial connection of several atone filters
`buthp` – second order IIR filer
`clfilt` – adjustable types and poles
`dcblock(2)` – removes DC offset

Band Pass And Resonant Filters `reson` – second order resonant filter
`resonx/resony` – serial/parallel connection of several reson filters
`resonr/resonz` – variants of the reson filter
`butbp` – second order butterworth filter

mode – mass-spring system modelled
fofilter – formant filter

Band Reject Filters **areson** – first order IIR filter
butbr – second order IIR filter

Equalizer **eqfil** – equalizer filter
rbjeq – parametric equalizer and filter
exciter – non-linear filter to add brilliance

REVERB

freeverb – stereo reverb after Jezar
reverbsc – stereo reverb after Sean Costello
reverb – simple reverb
nreverb – reverb with adjustable number of units
babo – physical model reverberator
(v)alpass – reverberates with a flat frequency response Note: Convolution reverb can be performed with
pconvolve and similar opcodes.

DISTORTION AND SIMILAR MODIFICATIONS

Distortion and Wave Shaping **distort(1)** – distortion via waveshaping
powershape – waveshaping by raising to a variable exponent
polynomial – polynominal over audio input signal
chebyshevpoly – chebyshev polynomials over audio input signal
fold – adds artificial foldover to an audio signal
pdclip – linear clipping of audio signal

Flanging, Phasing, Phase Shaping **flanger** – flanger
phaser1(2) – first/second order allpass filters in series
pdhalf(y) – phase distortion synthesis

Sample Level Operations **samphold** – performs sample-and-hold
vaget – audio vector read access
vaset – audio vector write access
framebuffer – reads/writes audio to/from array
shiftin/out – writes/reads the content of an audio variable to/from array

Other **doppler** – doppler shift
diff – modify a signal by differentiation
integ – modify a signal by integration
mirror – reflects a signal which exceeds boundaries
select – select sample value based on audio-rate comparisons
wrap – wraps around a signal which exceeds boundaries
waveset – repeating cycles of input audio signal

`sndloop` – looping on audio input signal
`mandel` – Mandelbrot set formula for complex plane

SIGNAL MEASUREMENT AND DYNAMIC PROCESSING

Amplitude Measurement and Envelope Following `rms` – RMS measurement
`peak` – maintains highest value received
`max_k` – local maximum/minimum of audio signal
`follow(2)` – envelope follower
`vactrol` – envelope follower

Pitch Estimation (Pitch Tracking) `ptrack` – pitch tracking using STFT
`pitch` – pitch tracking using constant-Q DFT
`pvspeech` – pitch/amplitude tracking of a PVS signal
`pvscent` – spectral centroid of a PVS signal

Dynamic Processing `balance(2)` – adjust audio signal level according to comparator
`compress(2)` – compress audio signal
`dam` – dynamic compressor/expander
`clip` – clips a signal to a predefined limit
`limit(1)` – sets lower and upper limit

SPATIALIZATION

Amplitude Panning `pan2` – stereo panning with different options
`vbap` – vector base amplitude panning for multichannel (also 3d)

Ambisonics `bformenc1` – B-format encoding
`bformdec1` – B-format decoding

Binaural / HRTF `hrtfstat` – static 3d binaural audio for headphones
`hrtfmove(2)` – dynamic 3d binaural audio
`hrtfearly` – early reflections in a HRTF room
`hrtfreverb` – binaural diffuse-field reverberator

Other `spat3d` – positioning in 3d space with optional simulation of room acoustics

IV. GRANULAR SYNTHESIS AND SPECTRAL PROCESSING

GRANULAR SYNTHESIS

`partikkel` – complete granular synthesis
`fof(2)` – formant orientated granular synthesis
`fog` – fof synthesis with samples sound
`diskgrain` – synchronous granular synthesis with sound file
`grain(2/3)` – granular textures

granule – complex granular textures
syncgrain/syncloop – synchronous granular synthesis
 others ... – other granular synthesis opcodes ... (see also chapter [05G](#))

SPECTRAL PROCESSING WITH PVS OPCODES

Environment **pvsinit** – initializes f-signal to zero
pvsinfo – get information about f-sig
pvsin – retrieve f-signal from input software bus
pvsout – writing f-signal to output software bus

Real-time Analysis and Resynthesis **pvsanal** – spectral analysis with audio signal input
pvtanal – spectral analysis from sampled sound
pvstrace – retain only N loudest bins
pvsynth – spectral resynthesis
pvsadsyn – spectral resynthesis using fast oscillator bank

Writing Spectral Data to a File and Reading from it **pvsfwrite** – writing f-sig to file
pvsfread – read f-sig data from a file loaded into memory
pvsdiskin – read f-sig data directly from disk

Writing Spectral Data to a Buffer or Array and Reading from it **pvsbuffer** – create and write f-sig to circular buffer
pvsbufread(2) – read f-sig from pvsbuffer
pvsftw – write amplitude and/or frequency data to tables
pvsftr – read amplitude and/or frequency data from table
pvs2array(pvs2tab) – write spectral data to arrays
pvsfromarray(tab2pvs) – read spectral data from arrays

Processing Spectral Signals **pvsbin** – obtain amp/freq from one bin
pvscent – spectral centroid of f-signal
pvsceps – cepstrum of f-signal
pvscale – scale frequency components (pitch shift)
pvshift – shift frequency components
pvsbandp – spectral band pass filter
pvsbandr – spectral band reject filter
pvsmix – mix two f-signals
pvcross – cross synthesis
pvsfilter – another cross synthesis
pvs voc – phase vocoder
pvsmorph – morphing between two f-signals
pvsfreeze – freeze amp/freq time functions
pvsmaska – modify amplitudes using table
pvcstencil – transform f-sig according to masking table
pvsarp – arpeggiate spectral components of f-sig
pvsblur – average amp/freq time functions
pvssmooth – smooth amp/freq time functions

`pvslock` – frequency lock input f-signal
`pvswrap` – warp the spectral envelope of an f-signal

OTHER SPECTRAL TRANSFORM

`dct(inv)` – (inverse) discrete cosine transformation
`fft(inv)` – (inverse) complex-to-complex FFT
`r2c` – real to complex conversion
`mags` – magnitudes of a complex-numbered array
`phs` – obtains phases of a complex-numbered array
`pol2rect` – polar to rectangular conversion of arrays
`rect2pol` – rectangular to polar format conversion
`rfft` – FFT of real-value array
`riFFT` – complex-to-real inverse FFT
`unwrap` – unwraps phase values array
`fmanal` – AM/FM analysis from quadrature signal
`hilbert(2)` – Hilbert transform
`mfb` – mel scale filterbank for spectral magnitudes

CONVOLUTION

`pconvolve` – partitioned convolution
`ftconv` – table-based partitioned convolution
`dconv` – direct convolution
`tvconv` – time-varying convolution

V. DATA

BUFFERS / FUNCTION TABLES

Creating/Deleting Function Tables (Buffers) `ftgen` – create any table with a GEN subroutine
`GEN Routines` – overview of subroutines
`ftfree` – delete function table
`ftgenonce` – create table inside an instrument
`ftgentmp` – create table bound to instrument instance
`tablecopy` – copy table from other table
`copya2ftab` – copy array to a function table

Writing to Tables `tablew` – write data to a table
`tablewkt` – write to k-rate changeable tables
`ftslice` – copy a slice from one table to another table
`modmatrix` – modulation matrix reading from and writing to tables
`ftmorf` – morph between tables and write the result

Reading From Tables `table(i/3)` – read from table (with linear/cubic interpolation)
`tablexkt` – reads function tables with linear/cubic/sinc interpolation

Saving Tables to Files `ftsave(k)` – save table(s) to text file or binary
`ftaudio` – save table data to audio file

Loading Tables From Files `ftload(k)` – load table(s) from file written with `ftsave`
`GEN23` – read numeric values from a text file
`GEN01` – load audio file into table
`GEN49` – load mp3 sound file into table

Writing Tables to Arrays `copyf2array` – copy function table to an array
`tab2array` – copy a slice from a table to an array

Table Queries `ftlen` – length of a table
`ftchnls` – number of channels of a stored sound
`ftsr` – sample rate of a stored sound
`nsamp` – number of sample frames in a table
`tabsum` – sum of table values
`getftargs` – get arguments of table creation

ARRAYS

Creation `init` – initialise array
`fillarray` – fill array with values
`genarray(_i)` – create array with arithmetic sequence
`=` – create or reset array as copy of another array

Analyse `lenarray` – length of array
`minarray` – minimum value in array
`maxarray` – maximum value in array
`sumarray` – sum of values in array
`cmp` – compare two arrays

Content Modification `scalearray` – scale values in an array
`sorta(d)` – sort an array in ascending (descending) order
`limit(1)` – limit array values
`(de)interleave` – combine/split arrays

Size Modification `slicearray` – take slice of an array
`trim(_i)` – adjust size of one-dimensional array

Format Interchange `copya2ftab` – copy array to a function table
`copyf2array` – copy function table to an array
`tab2array` – copy a slice from a table to an array
`pvs2array(pvs2tab)` – write spectral data to arrays
`pvsfromarray(tab2pvs)` – read spectral data from arrays

Dimension Interchange `reshapearray` – change dimensions of an array
`getrow` – get a row from a two-dimensional array
`getcol` – get a column from a two-dimensional array
`setrow` – set a row of a two-dimensional array
`setcol` – set a column of a two-dimensional array
`getrowlin` – copy a row from a 2D array and interpolate between rows

Functions See chapter 03E for a list of mathematical function which can directly be applied to arrays.

STRINGS

Creation `=` – direct assignment
`sprintf(k)` – string variable from format string
`strget` – string variable from strset number or p-field
`strcpy(k)` – string copy at i- or k-time
`strsub` – string as part of another string

String Queries `strcmp(k)` – compare strings
`strlen(k)` – length of string
`strindex(k)` – first occurrence of string1 in string2
`strrindex(k)` – last occurrence of string1 in string2
`strchar(k)` – return ASCII code of character in string

String Manipulation `strcat(k)` – concatenate strings
`strstrip` – removes white space from both ends of a string
`strlower(k)` – convert string to lower case
`strupper(k)` – convert string to upper case

Conversion and Assignment `S` – number to string
`strtod(k)` – string to number
`strset` – link string with a numeric value

FILES

Note: for sound files see SOUND FILES AND SAMPLES

`fprint(k)s` – formatted printing to file
`dumpk` – write k-signal to file
`hdf5write` – write signals and arrays to hdf5 file
`hdf5read` – read signals and arrays from hdf5 file
`readf(i)` – reads an external file line by line
`readk` – read k-signal from file

VI. PROGRAM FLOW

INSTRUMENTS AND VARIABLES

Instances and Allocation `active` — number of active instrument instances
`maxalloc` — set maximum number of instrument instances
`prealloc` — allocate memory before running an instrument
`subinstr` — instrument to be used as opcode

Variable Initialization and Conversion `init` — initialize variables
`reinit` — re-initialize i-variable
`i(k)` — i-value from k-signal
`k(a)` — k-signal from a-signal
`a(k)` — a-signal from k-signal

On-the-fly Evaluation and Compilation `evalstr` — evaluate Csound code as string
`compilecsd` — compile new instruments from csd file
`compileorc` — compile new instruments from raw orc file
`compilestr` — compile new instruments from string

Named Instruments `nstrnum` — number of a named instrument
`nstrstr` — name of an instrument

TIME, CONDITIONS, LOOPS, SCORE ACCESS

Time Reading `times` — absolute time in seconds
`timek` — absolute time in k-cycles
`timeinsts` — time of instrument instance in seconds
`timeinstk` — time of instrument instance in k-cycles

Conditions and Loops `if` — if clause
`(i/k)goto` — jump in code
`while` — while loop
`changed(2)` — k-rate signal change detector

Score Parameter Access `p(index)` — value in given p-field
`pset` — initialize p-field values
`passign` — assign p-field values to variables or array
`pcount` — number of p-fields in instrument

EVENTS AND TRIGGERS

Events `schedule(k)` — perform instrument event
`event(_i)` — perform any score event
`scoreline(_i)` — perform score lines
`sched(k)when(named)` — perform score event by trigger

`readscore` – read and process score from input string
`rewindscore` – rewind playback position of current score
`setscorepos` – set score playback position

Trigger Generators `metro(2)` – trigger metronome
`trigger` – threshold trigger
`sc_trig` – timed trigger
`seqtime(2)` – generates trigger according to values stored in a table
`timedseq` – time-variant sequencer

Terminate `turnoff` – turn off this instrument instance
`turnoff2` – turn off another instrument
`mute` – mute future instances of an instrument
`remove` – remove instrument definition
`exitnow` – exit Csound

PRINTING

Simple Printing `print` – print i-values
`printfk` – print k-values
`printfk2` – print k-values when changed
`puts` – print string

Formatted Printing `print(k)s` – formatted printing
`printf(_i)` – formatted printing with trigger

Arrays and Tables `printarray` – print array
`ftprint` – print table

SOFTWARE CHANNELS

Chn OpCodes `chn_k` – declare k-signal channel
`chn_a` – declare a-signal channel
`chn_S` – declare string channel
`chnset` – set value in channel
`chnget` – get value from channel
`chnmix` – mix value to channel
`chnclear` – clear channel
`chnseti/k/a/s` – array based chnset
`chngeti/k/a/s` – array based chnget

Invalue / Outvalue `invalue` – get value from channel
`outvalue` – set value to channel

Zak Patch System see [overview](#) in the Csound Manual

Signal Flow Graph and Mixer see [here](#)
and [here](#) in the Csound Manual

MATHEMATICAL CALCULATIONS

Arithmetic Operations `+` — addition

`-` — subtraction

`*` — multiplication

`/` — division

`^` — power of

`%` — modulo

`divz` — safe division (avoids division by zero) `exp` — e raised to x-th power

`log(2/10)` — logarithm (natural, 2, 10)

`sqrt` — square root

`abs` — absolute value

`int` — integer part

`frac` — fractional part

`signum` — signum function

`round` — round to nearest integer

`ceil` — round upwards

`floor` — round downwards

Trigonometric Functions `sin` — sine

`cos` — cosine

`tan` — tangent

`sinh` — hyperbolic sine

`cosh` — hyperbolic cosine

`tanh` — hyperbolic tangent

`sininv` — arcsine

`cosinv` — arccosine

`taninv(2)` — arctangent

Comparisons `min` — minimum of different i/k/a values

`max` — maximum of different i/k/a values

`minabs` — minimum of different absolute values/signals

`maxabs` — maximum of different absolute values/signals

`ntropol` — weighted mean of two values/signals

Logic Operators `&&` — logical and

`||` — logical or

Tests `qinf` — question whether argument is infinite number

`qnan` — question whether argument is not a number

CONVERTERS

MIDI to/from Frequency `mtof` — MIDI note number to frequency

`futm` — frequency to MIDI note number

`cpsmidinn` – MIDI note number to frequency
`mton` – midi number to note name
`ntom` – note name to midi number
`ntof` – note name to frequency

Other Pitch Converters `cent` – cent to scaling factor
`octave` – octave to scaling factor
`octcps` – frequency to octave-point-decimal
`cpsoct` – octave-point-decimal to frequency
`cspch` – pitch-class to frequency

Amplitude Converters `ampdb(fs)` – dB to amplitude (full scale)
`db(fs)amp` – amplitude to dB

Scaling `linlin` – linear signal scaling

VII. PERIPHERALS AND CONNECTIONS

MIDI

Note: Modern frontends now usually handle MIDI input.

Assignments `massign` – assign MIDI channel to Csound instrument
`pgmassign` – assign MIDI program to Csound instrument

Opcodes for Use in MIDI-Triggered Instruments `notnum` – note number received
`cpsmidi` – frequency of note received
`veloc` – velocity received
`ampmidi` – velocity with scaling options
`midichn` – MIDI channel received
`pchbend` – pitch bend received
`aftouch` – after-touch received
`polyaft` – polyphonic after-touch received

Opcodes For Use In All Instruments `ctrl7(14/21)` – receive controller
`initc7(14/21)` – initialize controller input
`mclock` – sends a MIDI clock message
`mdelay` – MIDI delay

MIDI Input and Output `midin` – generic MIDI messages received
`midout(_i)` – MIDI message to MIDI Out port
`midifilestatus` – status of MIDI input file
`midion` – sends note on/off messages to MIDI Out port
`more` MIDI out opcodes

OPEN SOUND CONTROL AND NETWORK

Open Sound Control `OSCinit` – initialize OSClisten port

`OSCinitM` – initializes multicast OSC listener

`OSClisten` – receive messages

`OSCraw` – listens to all messages

`OSCsend` – send messages

`OSCbundle` – send data in a bundle

`OSCcount` – report messages received but unread

Network Audio `socksend` – send data

`sockrecv` – receive data

`websocket` – read and write signals and arrays

OTHER

Widgets Note that *GUI elements are provided by all frontends.*

*Usage of the built-in FLTK widgets has limitations and is in general **not recommended**.*

FLTK overview [here](#)

Keyboard `sensekey` – sense computer keyboard

WII `wiiconnect` – connect

`wiidata` – read

`wiirange` – set scaling and range limits

`wiisend` – send data

P5 Glove `p5gconnect` – connect

`p5gdata` – read

Serial Opcodes `serialBegin` – open a serial port

`serialEnd` – close a serial port

`serialFlush` – flush data from a serial port

`serialPrint` – print data from a serial port

`serialRead` – read data from a serial port

`serialWrite(_i)` – write data to a serial port

SYSTEM

`getcfg` – get configuration

`system(_i)` – call external program via system

`pwd` – print working directory

`rtclock` – read real time clock

`date(s)` – return date and time

PLUGINS

Python `pyinit` – initialize Python interpreter

`pyrun` – run Python code

`pyexec` – execute script from file

`pycall` – write Python call to Csound variable

`pyeval` – evaluate Python expression and write to Csound variable

`pyassign` – assign Csound variable to Python variable

Faust `faustaudio` – instantiate and run a faust program

`faustcompile` – invoke compiler

`faustdsp` – instantiate a faust program

`faustctl` – adjust a given control

`faustgen` – compile, instantiate and run a faust program

`faustplay` – run a faust program

Ladspa `dssiinit` – load plugin

`dssiactivate` – (de)activate plugin

`dssilist` – list available plugins

`dssiaudio` – process audio using a plugin

`dssictls` – send control to plugin

This overview was compiled by Joachim Heintz in may 2020, based on Csound 6.14.

Thanks to Tarmo Johannes, Victor Lazzarini, Gleb Rogozinsky, Steven Yi, Oeyvind Brandtsegg, Richard Boulanger, John ffitch, Luis Jure, Rory Walsh, Eduardo Moguillansky and others for their feedback which made the selections at least a tiny bit less subjective.

14 B. METHODS OF WRITING CSOUND SCORES

Although the use of Csound real-time has become more prevalent and arguably more important whilst the use of 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, then with the definition of a user-defined score language, and concluding several external Csound score generating programs.

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 even a 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 interpreted 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:
 1. A copy of the user written score. This file has the suffix `.ext`
 2. 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 installed on your computer, you should be able to run the following examples. They do actually nothing but print the arguments (= file names).

Calling a binary without a script

EXAMPLE 14A01_Score_bin.csd

```
<CsoundSynthesizer>
<CsInstruments>
instr 1
endin
</CsInstruments>
<CsScore bin="python3">
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
```

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 this csd:

EXAMPLE 14A02_Score_bin_script.csd

```
<CsoundSynthesizer>
<CsInstruments>
instr 1
endin
</CsInstruments>
<CsScore bin="python3 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
```

To get rid of the complaints we must write the score file. This is a simple example, saved as *test.py*:

```
from sys import argv
print("Script = '%s'" % argv[0])
print("File to read = '%s'" % argv[1])
print("File to write = '%s'" % argv[2])
with open(argv[2], 'w') as out:
    out.write('i 1 0 1')
```

Which is called in this Csound file:

```
<CsoundSynthesizer>
<CsInstruments>
instr 1
    outall(oscil(.2,expon(1500,p3,400)))
endin
</CsInstruments>
<CsScore bin="python3 test.py">
</CsScore>
</CsoundSynthesizer>
```

CsBeats

As an alternative to the classical Csound score, [CsBeats](#) is included with Csound. This is a domain specific language tailored to the concepts of beats, rhythm and standard western notation. To use Csbeat, specify “csbeats” as the CsScore bin option in a Csound unified score file.

```
<CsScore bin="csbeats">
```

For more information, refer to the [Csound Manual](#).

Scripting Language Examples

The following example 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.

EXAMPLE 14A03_Score_perlscript.csd

```
<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);When you execute this .csd file in the terminal, your
open(SCO, ">", $out);

while (<EXT>) {
  s/SEED\s+(\d+)/srand($1);$&/e;
  s/rand\(\d*\)/eval $&/ge;
  print SCO;
}
```

Pysco

Pysco is a modular Csound score environment for event generation, event processing, and the fashioning musical structures in time. Pysco is non-imposing and does not force composers into any one particular compositional model; Composers design their own score frameworks by importing from existing Python libraries, or fabricate their own functions as needed. It fully supports the existing classical Csound score, and runs inside a unified CSD file. The sources are [on github](#), so although the code is still using Python2, it can certainly serve as an example about the possibilities of using Python as score scripting language.

Pysco is designed to be a giant leap forward from the classical Csound score by leveraging Python, a highly extensible general-purpose scripting language. While the classical Csound score does feature a small handful of score tricks, it lacks common computer programming paradigms, offering little in terms of alleviating the tedious process of writing scores by hand. Python plus the Pysco interface transforms the limited classical score into highly flexible and modular text-based compositional environment.

Transitioning away from the Classical Csound Score

Only two changes are necessary to get started. First, the optional *bin* argument for the CsScore tag needs to specify “python pysco.py”. Second, all existing classical Csound score code works when placed inside the *score()* function.

```
<CsScore bin="python pysco.py">

score('''
f 1 0 8192 10 1
t 0 144
i 1 0.0 1.0 0.7 8.02
i 1 1.0 1.5 0.4 8.05
i 1 2.5 0.5 0.3 8.09
i 1 3.0 1.0 0.4 9.00
''')

</CsScore>
```

Boiler plate code that is often associated with scripting and scoring, such as file management and string concatenation, has been conveniently factored out.

The last step in transitioning is to learn a few of Python or Pysco features. While Pysco and Python offers an incredibly vast set of tools and features, one can supercharge their scores with only a small handful.

Managing Time with the *cue()*

The *cue()* object is the Pysco [context manager](#) for controlling and manipulating time in a score. Time is a fundamental concept in music, and the *cue()* object elevates the role of time to that of

other control such as *if* and *for* statements, synthesizing time into the form of the code.

In the classical Csound score model, there is only the concept of beats. This forces composers to place events into the global timeline, which requires an extra added inconvenience of calculating start times for individual events. Consider the following code in which measure 1 starts at time 0.0 and measure 2 starts at time 4.0.

```
; Measure 1
i 1 0.0 1.0 0.7 8.02
i 1 1.0 1.5 0.4 8.05
i 1 2.5 0.5 0.3 8.09
i 1 3.0 1.0 0.4 9.00

; Measure 2
i 1 4.0 1.0 0.7 8.07
i 1 5.0 1.5 0.4 8.10
i 1 6.5 0.5 0.3 9.02
i 1 7.0 1.0 0.4 9.07
```

In an ideal situation, the start times for each measure would be normalized to zero, allowing composers to think local to the current measure rather than the global timeline. This is the role of Pysco's *cue()* context manager. The same two measures in Pysco are rewritten as follows:

```
# Measure 1
with cue(0):
    score('''
        i 1 0.0 1.0 0.7 8.02
        i 1 1.0 1.5 0.4 8.05
        i 1 2.5 0.5 0.3 8.09
        i 1 3.0 1.0 0.4 9.00
    ''')

# Measure 2
with cue(4):
    score('''
        i 1 0.0 1.0 0.7 8.07
        i 1 1.0 1.5 0.4 8.10
        i 1 2.5 0.5 0.3 9.02
        i 1 3.0 1.0 0.4 9.07
    ''')
```

The start of measure 2 is now 0.0, as opposed to 4.0 in the classical score environment. The physical layout of these time-based block structure also adds visual cues for the composer, as indentation and *with cue()* statements adds clarity when scanning a score for a particular event.

Moving events in time, regardless of how many there are, is nearly effortless. In the classical score, this often involves manually recalculating entire columns of start times. Since the *cue()* supports nesting, it's possible and rather quite easy, to move these two measures any where in the score with a new *with cue()* statement.

```
# Movement 2
with cue(330):
    # Measure 1
    with cue(0):
        i 1 0.0 1.0 0.7 8.02
        i 1 1.0 1.5 0.4 8.05
        i 1 2.5 0.5 0.3 8.09
        i 1 3.0 1.0 0.4 9.00

    #Measure 2
    with cue(4):
```

```
i 1 0.0 1.0 0.7 8.07
i 1 1.0 1.5 0.4 8.10
i 1 2.5 0.5 0.3 9.02
i 1 3.0 1.0 0.4 9.07
```

These two measures now start at beat 330 in the piece. With the exception of adding an extra level of indentation, the score code for these two measures are unchanged.

Generating Events

Pysco includes two functions for generating a Csound score event. The `score()` function simply accepts any and all classical Csound score events as a string. The second is `event_i()`, which generates a properly formatted Csound score event. Take the following Pysco event for example:

```
event_i(1, 0, 1.5, 0.707 8.02)
```

The `event_i()` function transforms the input, outputting the following Csound score code:

```
i 1 0 1.5 0.707 8.02
```

These event score functions combined with Python's extensive set of features aid in generating multiple events. The following example uses three of these features: the `for statement`, `range()`, and `random()`.

```
from random import random

score('t 0 160')

for time in range(8):
    with cue(time):
        frequency = 100 + random() * 900
        event_i(1, 0, 1, 0.707, frequency)
```

Python's `for` statement combined with `range()` loops through the proceeding code block eight times by iterating through the list of values created with the `range()` function. The list generated by `range(8)` is:

```
[0, 1, 2, 3, 4, 5, 6, 7]
```

As the script iterates through the list, variable `time` assumes the next value in the list; The `time` variable is also the start time of each event. A hint of algorithmic flair is added by importing the `random()` function from Python's `random library` and using it to create a random frequency between 100 and 1000 Hz. The script produces this classical Csound score:

```
t 0 160
i 1 0 1 0.707 211.936363038
i 1 1 1 0.707 206.021046104
i 1 2 1 0.707 587.07781543
i 1 3 1 0.707 265.13585797
i 1 4 1 0.707 124.548796225
i 1 5 1 0.707 288.184408335
i 1 6 1 0.707 396.36805871
i 1 7 1 0.707 859.030151952
```

Processing Events

Pysco includes two functions for processing score event data called *p_callback()* and *pmap()*. The *p_callback()* is a pre-processor that changes event data before it's inserted into the score object while *pmap()* is a post-processor that transforms event data that already exists in the score.

```
p_callback(event_type, instr_number, pfield, function, *args)
pmap(event_type, instr_number, pfield, function, *args)
```

The following examples demonstrates a use case for both functions. The *p_callback()* function pre-processes all the values in the pfield 5 column for instrument 1 from conventional notation (D5, G4, A4, etc) to hertz. The *pmap()* post-processes all pfield 4 values for instrument 1, converting from decibels to standard amplitudes.

```
p_callback('i', 1, 5, conv_to_hz)

score('''
t 0 120
i 1 0 0.5 -3 D5
i 1 + . . G4
i 1 + . . A4
i 1 + . . B4
i 1 + . . C5
i 1 + . . A4
i 1 + . . B4
i 1 + . . G5
''')

pmap('i', 1, 4, dB)
```

The final output is:

```
f 1 0 8192 10 1
t 0 120
i 1 0 0.5 0.707945784384 587.329535835
i 1 + . . 391.995435982
i 1 + . . 440.0
i 1 + . . 493.883301256
i 1 + . . 523.251130601
i 1 + . . 440.0
i 1 + . . 493.883301256
i 1 + . . 783.990871963
```

CMask

CMask is an application that produces score files for Csound, i.e. lists of notes or rather events. Its main application is the generation of events to create a texture or granular sounds. The program takes a parameter file as input and makes a score file that can be used immediately with Csound.

The basic concept in CMask is the tendency mask. This is an area that is limited by two time variant boundaries. This area describes a space of possible values for a score parameter, for example amplitude, pitch, pan, duration etc. For every parameter of an event (a note statement

pfield in Csound) a random value will be selected from the range that is valid at this time.

There are also other means in CMask for the parameter generation, for example cyclic lists, oscillators, polygons and random walks. Each parameter of an event can be generated by a different method. A set of notes / events generated by a set of methods lasting for a certain time span is called a field.

A CMask example: creation of a dynamic texture

```
{
f1 0 8193 10 1 ;sine wave
}

f 0 20 ;field duration: 20 secs

p1 const 1
p2 ;decreasing density
rnd uni ;from .03 - .08 sec to .5 - 1 sec
mask [.03 .5 ipl 3] [.08 1 ipl 3] map 1
prec 2
p3 ;increasing duration
rnd uni mask [.2 3 ipl 1] [.4 5 ipl 1]
prec 2

p4 ;narrowing frequency grid
rnd uni mask [3000 90 ipl 1] [5000 150 ipl 1] map 1
quant [400 50] .95
prec 2
p5 ;FM index gets higher from 2-4 to 4-7
rnd uni mask [2 4] [4 7]
prec 2

p6 range 0 1 ;panorama position uniform distributed
prec 2 ;between left and right
```

The output is:

```
f1 0 8193 10 1 ;sine wave

; ----- begin of field 1 --- seconds: 0.00 - 20.00 -----
;ins    time      dur      p4      p5      p6

i1      0        0.37    3205.55 3.57    0.8
i1      0.07    0.24    3190.83 3.55    0.28
i1      0.12    0.3     3589.39 2.74    0.51
i1      0.2     0.38    3576.81 3.46    0.14
i1      0.25    0.2     3158.89 2.3     0.8
i1      0.28    0.28    2775.01 2.25    1
.....
.....
.....
i1      18.71   4.32    145.64   5.75    0.27
i1      19.12   3.27    129.68   5.27    0.3
i1      19.69   4.62    110.64   6.87    0.65

; ----- end of field 1 --- number of events: 241 -----
```

Cmask can be downloaded for [MacOS9](#), [Win](#), [Linux](#) (by André Bartetzki) and is ported to [OSX](#)(by Anthony Kozar).

nGen

nGen is a free multi-platform generation tool for creating Csound event-lists (score files) and standard MIDI files. It is written in C and runs on a variety of platforms (version 2.1.2 is currently available for OSX 10.5 and above, Linux Intel and Windows 7+). All versions run in the UNIX command-line style (at a command-line shell prompt). nGen was designed and written by composer Mikel Kuehn and was inspired in part by the basic syntax of Aleck Brinkman's Score11 note list pre-processor (Score11 is available for Linux Intel from the Eastman Computer Music Center) and Leland Smith's Score program.

nGen will allow you to do several things with ease that are either difficult or not possible using Csound and/or MIDI sequencing programs; nGen is a powerful front-end for creating Csound score-files and basic standard MIDI files. Some of the basic strengths of nGen are:

- Event-based granular textures can be generated quickly. Huge streams of values can be generated with specific random-number distributions (e.g., Gaussian, flat, beta, exponential, etc.).
- Note-names and rhythms can be entered in intuitive formats (e.g., pitches: C4, Df3; rhythms: 4, 8, 16, 32).
- “Chords” can be specified as a single unit (e.g., C4:Df:E:Fs). Textual and numeric macros are available.

Additionally, nGen supplies a host of conversion routines that allow p-field data to be converted to different formats in the resulting Csound score file (e.g., octave.pitch-class can be formatted to Hz values, etc.). A variety of formatting routines are also supplied (such as the ability to output floating-point numbers with a certain precision width).

nGen is a portable text-based application. It runs on most platforms (Windows, Mac, Linux, Irix, UNIX, etc.) and allows for macro- and micro-level generation of event-list data by providing many dynamic functions for dealing with statistical generation (such as interpolation between values over the course of many events, varieties of pseudo-random data generation, p-field extraction and filtering, 1/f data, the use of “sets” of values, etc.) as well as special modes of input (such as note-name/octave-number, reciprocal duration code, etc.). Its memory allocation is dynamic, making it useful for macro-level control over huge score-files. In addition, nGen contains a flexible text-based macro pre-processor (identical to that found in recent versions of Csound), numeric macros and expressions, and also allows for many varieties of data conversion and special output formatting. nGen is command-line based and accepts an ASCII formatted text-file which is expanded into a Csound score-file or a standard MIDI file. It is easy to use and is extremely flexible making it suitable for use by those not experienced with high-level computer programming languages.

An example of simple granular synthesis with wave forms

```
;These lines go directly to the output file
>f1    0    16384   10    1                                ;sine wave
>f2    0    16384   10    1 0 .5 0 .25 0 .125 0 .0625 ;odd partials (dec.)
>f3    0    16384   10    1 .5 .25 .125 .0625        ;decreasing strength
>f4    0    16384   10    1 1 1 1 1                   ;pulse
>f5    0    16384   10    1 0 1 0 1                   ;odd
>f82   0    16385   20    2    1                      ;grain envelope
```

```
#define MAX #16000#                                ;a macro for the maximum amplitude

i1 = 7 0 10 {                                     ;intervalic start time
    p2 .01

    /* The duration of each event slowly changes over time starting at 20 the
     initial start time interval to 1x the ending start-time interval. The "T"
     variable is used to control the duration of both move statements (50% of
     the entire i-block duration). */
    p3 mo(T*.5 1. 20 1)   mo(T*.5 1. 1 10)

    /* Amplitude gets greater in the center to compensate for shorter grains
     the MAX macro (see above) is used to set the high range anchor. */
    p4 rd(.1) mo(T*.5, 1. E 0 $MAX)   mo(T*.5 1. E $MAX 0)

    /* Frequency: moves logarithmically from 3000 to a range between 100 and
     200 then exponentially up to a range between 1000 and 4000. The "T"
     variable is again used to specify a percentage of the iblock's total
     duration. If you try to compile this as a MIDI file, all of the Herz
     values will turn into MIDI note numbers through VALUE % 128 -- rapidly
     skimming over the entire keyboard... */
    p5 rd(0) mo(T*.4 1. l 3000 [100 200]) \
        mo(T*.6 1. e [100 200] [1000 4000])

    /* Spatial placement: 25% hard-left 25% hard-right 50% a Gaussian value
     (near the middle). */
    p6(re2) ra(10 .25 0 .25 1 .5 [g 0 1])
    p7(in)  se(T 1. [1 2 3 4 5]) ;select different wave-form function #
}
```

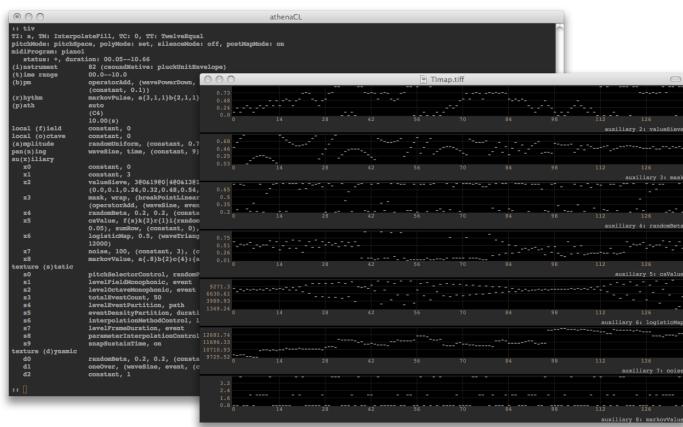
The output is:

```
f1  0  16384  10  1                               ;sine wave
f2  0  16384  10  1 0 .5 0 .25 0 .125 0 .0625 ;odd partials (dec.)
f3  0  16384  10  1 .5 .25 .125 .0625       ;decreasing strength
f4  0  16384  10  1 1 1 1 1                     ;pulse
f5  0  16384  10  1 0 1 0 1                     ;odd
f82 0  16385  20  2 1                           ;grain envelope
;I-block #1 (i1):
i1  0.000  0.200      0.000  3000.000      0.00  3
i1  0.010  0.200      0.063  2673.011      0.79  3
i1  0.020  0.199      0.253  2468.545      1.00  2
i1  0.030  0.199      0.553  2329.545      1.00  5
i1  0.040  0.198      1.033  2223.527      1.00  2
i1  0.050  0.198      1.550  2160.397      0.50  4
.....
.....
.....
i1  9.970  0.100      127.785  2342.706      0.48  1
i1  9.980  0.100      64.851  3200.637      1.00  1
i1  9.990  0.100      0.000  3847.285      1.00  2
e
```

nGen for Mac, Windows and Linux can be downloaded [here](#)

AthenaCL

The athenaCL system is a software tool for creating musical structures. Music is rendered as a polyphonic event list, or an EventSequence object. This EventSequence can be converted into diverse forms, or OutputFormats, including scores for the Csound synthesis language, Musical Instrument Digital Interface (MIDI) files, and other specialized formats. Within athenaCL, Orchestra and Instrument models provide control of and integration with diverse OutputFormats. Orchestra models may include complete specification, at the code level, of external sound sources that are created in the process of OutputFormat generation.



The athenaCL system features specialized objects for creating and manipulating pitch structures, including the Pitch, the Multiset (a collection of Pitches), and the Path (a collection of Multisets). Paths define reusable pitch groups. When used as a compositional resource, a Path is interpreted by a Texture object (described below).

The athenaCL system features three levels of algorithmic design. The first two levels are provided by the ParameterObject and the Texture. The ParameterObject is a model of a low-level one-dimensional parameter generator and transformer. The Texture is a model of a multi-dimensional generative musical part. A Texture is controlled and configured by numerous embedded ParameterObjects. Each ParameterObject is assigned to either event parameters, such as amplitude and rhythm, or Texture configuration parameters. The Texture interprets ParameterObject values to create EventSequences. The number of ParameterObjects in a Texture, as well as their function and interaction, is determined by the Texture's parent type (TextureModule) and Instrument model. Each Texture is an instance of a TextureModule. TextureModules encode diverse approaches to multi-dimensional algorithmic generation. The TextureModule manages the deployment and interaction of lower level ParameterObjects, as well as linear or non-linear event generation. Specialized TextureModules may be designed to create a wide variety of musical structures.

The third layer of algorithmic design is provided by the Clone, a model of the multi-dimensional transformative part. The Clone transforms EventSequences generated by a Texture. Similar to Textures, Clones are controlled and configured by numerous embedded ParameterObjects.

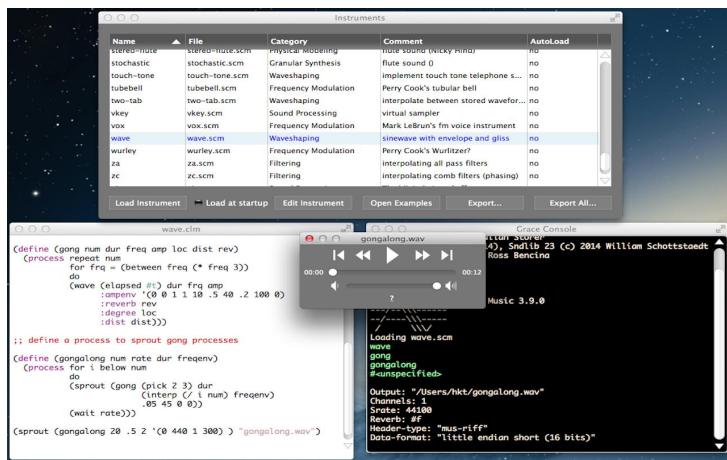
Each Texture and Clone creates a collection of Events. Each Event is a rich data representation that includes detailed timing, pitch, rhythm, and parameter data. Events are stored in EventSequence objects. The collection all Texture and Clone EventSequences is the complete output of

athenaCL. These EventSequences are transformed into various OutputFormats for compositional deployment.

AthenaCL can be downloaded [here](#).

Common Music

Common Music is a music composition system that transforms high-level algorithmic representations of musical processes and structure into a variety of control protocols for sound synthesis and display. It generates musical output via MIDI, OSC, CLM, FOMUS and CSOUND. Its main user application is Grace (Graphical Realtime Algorithmic Composition Environment) a drag-and-drop, cross-platform app implemented in JUCE (C++) and S7 Scheme. In Grace musical algorithms can run in real time, or faster-than-real time when doing file-based composition. Grace provides two coding languages for designing musical algorithms: S7 Scheme, and SAL, an easy-to-learn but expressive algol-like language.



Some of the features:

- Runs on Mac, Windows and Linux
- Two coding languages for designing algorithms: S7 Scheme and SAL (an easy-to-learn alternate)
- Data visualization

Common Music 3 can be downloaded [here](#).

14 C. AMPLITUDE AND PITCH TRACKING

Tracking the amplitude of an audio signal is a relatively simple procedure but simply following the amplitude values of the waveform is unlikely to be useful. An audio waveform will be bipolar, expressing both positive and negative values, so to start with, some sort of rectifying of the negative part of the signal will be required. The most common method of achieving this is to square it (raise to the power of 2) and then to take the square root. Squaring any negative values will provide positive results (-2 squared equals 4). Taking the square root will restore the absolute values.

An audio signal is an oscillating signal, periodically passing through amplitude zero but these zero amplitudes do not necessarily imply that the signal has decayed to silence as our brain perceives it. Some sort of averaging will be required so that a tracked amplitude of close to zero will only be output when the signal has settled close to zero for some time. Sampling a set of values and outputting their mean will produce a more acceptable sequence of values over time for a signal's change in amplitude. Sample group size will be important: too small a sample group may result in some residual ripple in the output signal, particularly in signals with only low frequency content, whereas too large a group may result in a sluggish response to sudden changes in amplitude. Some judgement and compromise is required.

The procedure described above is implemented in the following example. A simple audio note is created that ramps up and down according to a linseg envelope. In order to track its amplitude, audio values are converted to k-rate values and are then squared, then square rooted and then written into sequential locations of an array 31 values long. The mean is calculated by summing all values in the array and divided by the length of the array. This procedure is repeated every k-cycle. The length of the array will be critical in fine tuning the response for the reasons described in the preceding paragraph. Control rate (kr) will also be a factor therefore is taken into consideration when calculating the size of the array. Changing control rate (kr) or number of audio samples in a control period (ksmps) will then no longer alter response behaviour.

EXAMPLE 14C01_Amplitude_Tracking_First_Principles.csd

```
<CsoundSynthesizer>
<CsOptions>
-dm128 -odac
</CsOptions>
<CsInstruments>
sr = 44100
ksmps = 16
nchnls = 2
0dbfs = 1
```

```

; a rich waveform
giwave ftgen 1,0, 512, 10, 1,1/2,1/3,1/4,1/5

instr 1
; create an audio signal
aenv    linseg    0,p3/2,1,p3/2,0 ; triangle shaped envelope
aSig    poscil    aenv,300,giwave ; audio oscillator
        out       aSig, aSig      ; send audio to output

; track amplitude
kArr[] init 500 / ksmps    ; initialise an array
kNdx   init 0             ; initialise index for writing to array
kSig    downsample aSig    ; create k-rate version of audio signal
kSq     =     kSig ^ 2      ; square it (negatives become positive)
kRoot   =     kSq ^ 0.5     ; square root it (restore absolute values)
kArr[kNdx] = kRoot        ; write result to array
kMean   =     sumarray(kArr) / lenarray(kArr); calculate mean of array
        printk 0.1,kMean    ; print mean to console
; increment index and wrap-around if end of the array is met
kNdx   wrap    kNdx+1, 0, lenarray(kArr)
endin

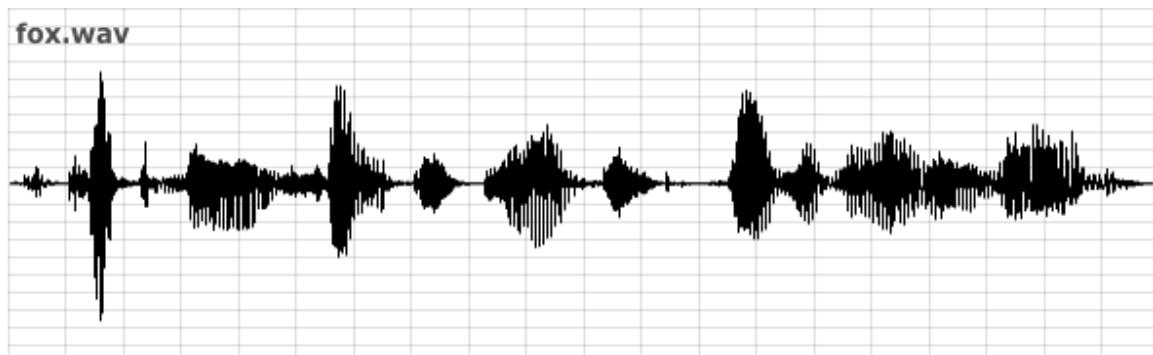
</CsInstruments>
<CsScore>
i 1 0 5
</CsScore>
</CsoundSynthesizer>
;example by Iain McCurdy

```

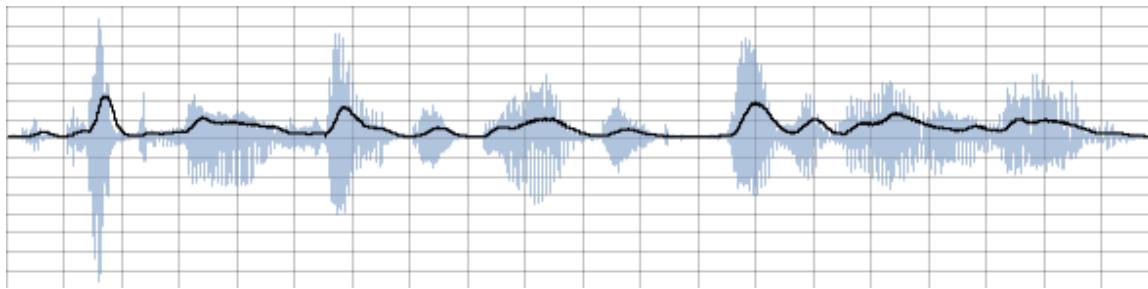
In practice it is not necessary for us to build our own amplitude tracker as Csound already offers several opcodes for the task. `rms` outputs a k-rate amplitude tracking signal by employing mathematics similar to those described above. `follow` outputs at a-rate and uses a sample and hold method as it outputs data, probably necessitating some sort of low-pass filtering of the output signal. `follow2` also outputs at a-rate but smooths the output signal by different amounts depending on whether the amplitude is rising or falling.

A quick comparison of these three opcodes and the original method from first principles is given below:

The sound file used in all three comparisons is `fox.wav` which can be found as part of the Csound HTML Manual download. This sound is someone saying: “the quick brown fox jumps over the lazy dog”.

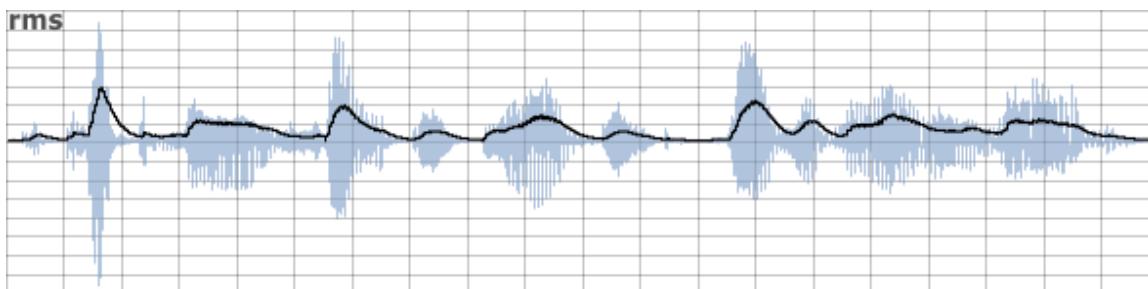


First of all by employing the technique exemplified in example 14C01, the amplitude following signal is overlaid upon the source signal:



It can be observed that the amplitude tracking signal follows the amplitudes of the input signal reasonably well. A slight delay in response at sound onsets can be observed as the array of values used by the averaging mechanism fills with appropriately high values. As discussed earlier, reducing the size of the array will improve response at the risk of introducing ripple. Another approach to dealing with the issue of ripple is to low-pass filter the signal output by the amplitude follower. This is an approach employed by the [follow2](#) opcode. The second thing that is apparent is that the amplitude following signal does not attain the peak value of the input signal. At its peaks, the amplitude following signal is roughly 1/3 of the absolute peak value of the input signal. How close it gets to the absolute peak amplitude depends somewhat on the dynamic nature of the input signal. If an input signal sustains a peak amplitude for some time then the amplitude following signal will tend to this peak value.

The [rms](#) opcode employs a method similar to that used in the previous example but with the convenience of an encapsulated opcode. Its output superimposed upon the waveform is shown below:



Its method of averaging uses filtering rather than simply taking a mean of a buffer of amplitude values. [rms](#) allows us to set the cutoff frequency (kCf) of its internal filter:

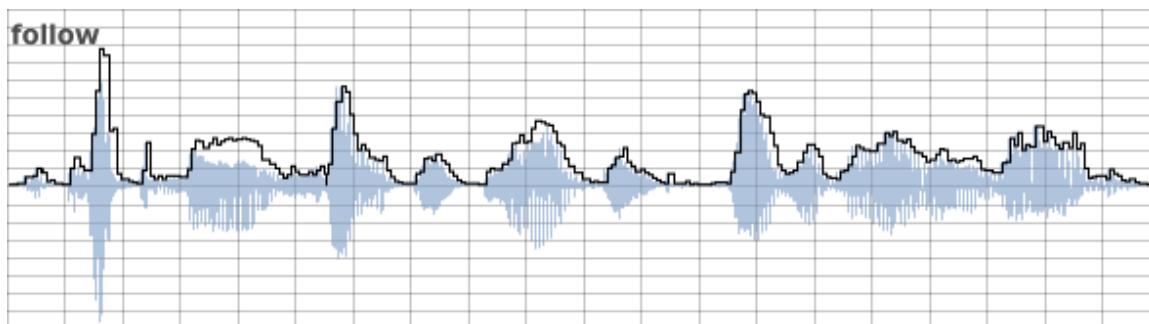
```
kRms    rms    aSig, kCf
```

This is an optional argument which defaults to 10. Lowering this value will dampen changes in rms and smooth out ripple, raising it will improve the response but increase the audibility of ripple. A choice can be made based on some foreknowledge of the input audio signal: dynamic percussive input audio might demand faster response whereas audio that dynamically evolves gradually might demand greater smoothing.

The [follow](#) opcode uses a sample-and-hold mechanism when outputting the tracked amplitude. This can result in a stepped output that might require addition lowpass filtering before use. We actually defined the period, the duration for which values are held, using its second input argument. The update rate will be one over the period. In the following example the audio is amplitude tracked using the following line:

```
aRms    follow    aSig, 0.01
```

with the following result:

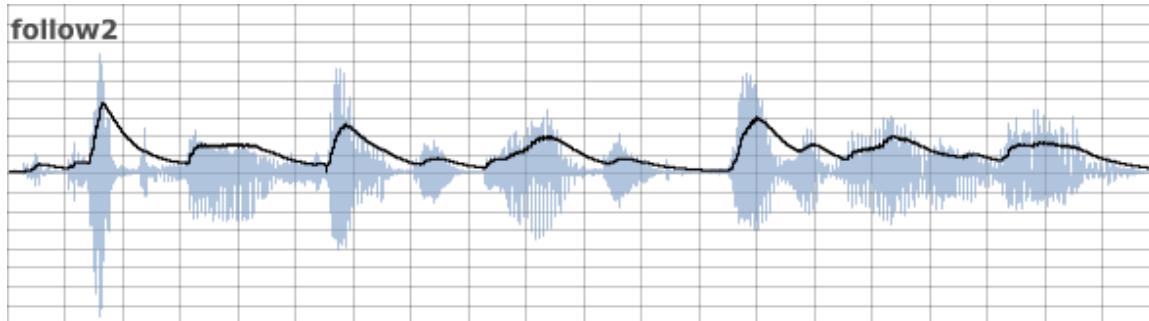


The hump over the word spoken during the third and fourth time divisions initially seem erroneous but it is a result of greater amplitude excursion into the negative domain. `follow` provides a better reflection of absolute peak amplitude.

`follow2` uses a different algorithm with smoothing on both upward and downward slopes of the tracked amplitude. We can define different values for attack and decay time. In the following example the decay time is much longer than the attack time. The relevant line of code is:

```
iAtt = 0.04
iRel = 0.5
aTrk follow2 aSig, 0.04, 0.5
```

and the result of amplitude tracking is:



This technique can be used to extend the duration of short input sound events or triggers. Note that the attack and release times for `follow2` can also be modulated at k-rate.

Dynamic Gating and Amplitude Triggering

Once we have traced the changing amplitude of an audio signal it is straightforward to use specific changes in that function to trigger other events within Csound. The simplest technique would be to simply define a threshold above which one thing happens and below which something else happens. A crude dynamic gating of the signal above could be implemented thus:

EXAMPLE 14C02_Simple_Dynamic_Gate.csd

```
<CsoundSynthesizer>
<CsOptions>
-dm128 -odac
</CsOptions>
<CsInstruments>
```

```

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1
; this is a necessary definition,
; otherwise amplitude will be -32768 to 32767

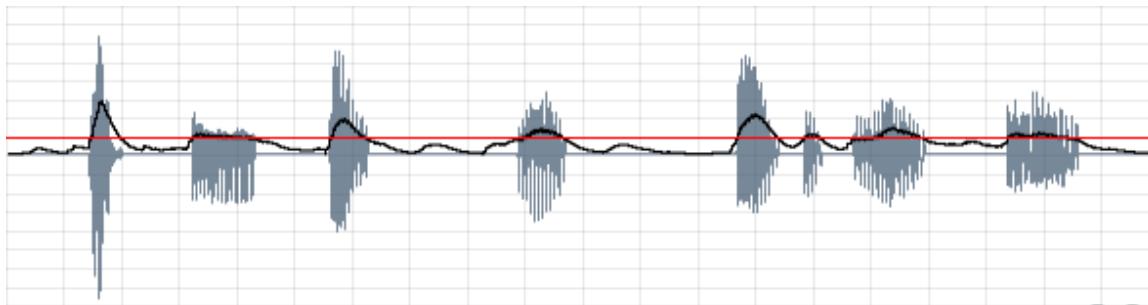
instr 1
aSig diskin "fox.wav", 1           ; read sound file
kRms rms aSig                      ; scan rms
iThreshold = 0.1                   ; rms threshold
kGate = kRms > iThreshold ? 1 : 0 ; gate either 1 or zero
aGate interp kGate ; interpolate to create smoother on->off->on switching
aSig = aSig * aGate                ; multiply signal by gate
out aSig, aSig                     ; send to output
endin

</CsInstruments>
<CsScore>
i 1 0 10
</CsScore>
</CsoundSynthesizer>
;example by Iain McCurdy

```

Once a dynamic threshold has been defined, in this case 0.1, the RMS value is interrogated every k-cycle as to whether it is above or below this value. If it is above, then the variable kGate adopts a value of 1 (open) or if below, kGate is zero (closed). This on/off switch could just be multiplied to the audio signal to turn it on or off according to the status of the gate but clicks would manifest each time the gates opens or closes so some sort of smoothing or ramping of the gate signal is required. In this example I have simply interpolated it using the *interp* opcode to create an a-rate signal which is then multiplied to the original audio signal. This means that a linear ramp with be added across the duration of a k-cycle in audio samples – in this case 32 samples. A more elaborate approach might involve portamento and low-pass filtering.

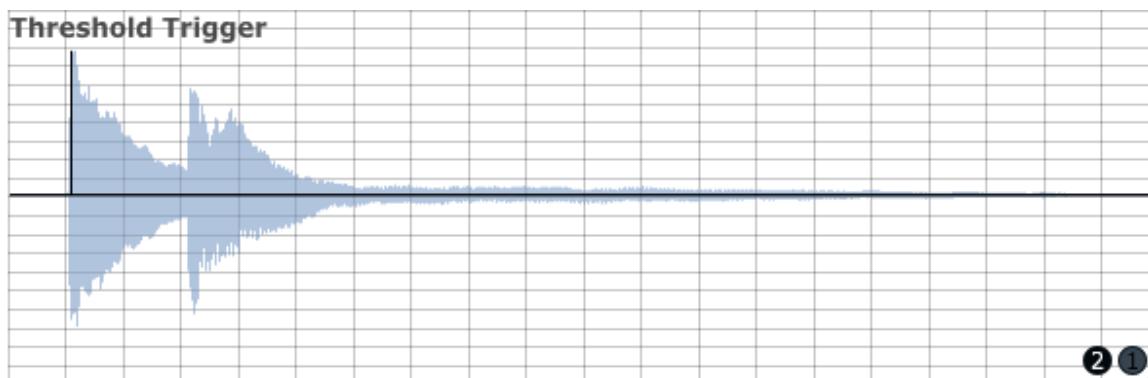
The results of this dynamic gate are shown below:



The threshold is depicted as a red line. It can be seen that each time the RMS value (the black line) drops below the threshold the audio signal (blue waveform) is muted.

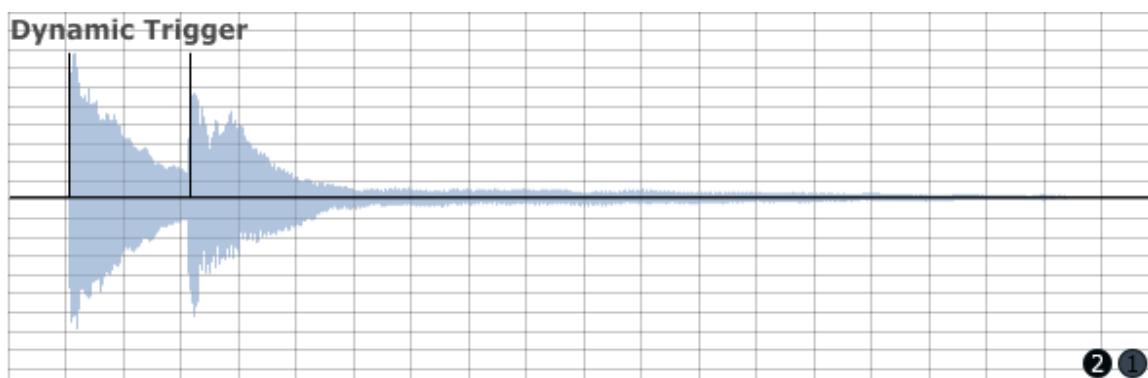
The simple solution described above can prove adequate in applications where the user wishes to sense sound event onsets and convert them to triggers but in more complex situations, in particular when a new sound event occurs whilst the previous event is still sounding and pushing the RMS above the threshold, this mechanism will fail. In these cases triggering needs to depend upon dynamic *change* rather than absolute RMS values. If we consider a two-event sound file where two notes sound on a piano, the second note sounding while the first is still decaying, triggers generated using the RMS threshold mechanism from the previous example will only sense the first note onset. (In the diagram below this sole trigger is illustrated by the vertical black line.) Raising the

threshold might seem to be remedial action but is not ideal as this will prevent quietly played notes from generating triggers.



It will often be more successful to use magnitudes of amplitude increase to decide whether to generate a trigger or not. The two critical values in implementing such a mechanism are the time across which a change will be judged (*iSampTim* in the example) and the amount of amplitude increase that will be required to generate a trigger (*iThresh*). An additional mechanism to prevent double triggerings if an amplitude continues to increase beyond the time span of a single sample period will also be necessary. What this mechanism will do is to bypass the amplitude change interrogation code for a user-definable time period immediately after a trigger has been generated (*iWait*). A timer which counts elapsed audio samples (*kTimer*) is used to time how long to wait before retesting amplitude changes.

If we pass our piano sound file through this instrument, the results look like this:



This time we correctly receive two triggers, one at the onset of each note.

The example below tracks audio from the sound-card input channel 1 using this mechanism.

EXAMPLE 14C03_Dynamic_Trigger.csd

```
<CsoundSynthesizer>
<CsOptions>
-dm0 -iadc -odac
</CsOptions>
<CsInstruments>
sr      =  44100
ksmps  =  32
nchnls =  2
0dbfs  =  1

instr  1
iThresh  =        0.1          ; change threshold

```

```

aSig      inch    1           ; live audio in
iWait     =      1000        ; prevent repeats wait time (in samples)
kTimer   init    1001        ; initial timer value
kRms     rms    aSig, 20      ; track amplitude
iSampTim =      0.01        ; time across which change in RMS will be measured
kRmsPrev delayk kRms, iSampTim ; delayed RMS (previous)
kChange   =      kRms - kRmsPrev ; change
if(kTimer>iWait) then          ; if we are beyond the wait time...
  kTrig   =      kChange > iThresh ? 1 : 0 ; trigger if threshold exceeded
  kTimer   =      kTrig == 1 ? 0 : kTimer ; reset timer when a trigger generated
else                           ; otherwise (we are within the wait time buffer)
  kTimer   +=      ksmpls       ; increment timer
  kTrig   =      0             ; cancel trigger
endif
  schedkwhen kTrig,0,0,2,0,0.1 ; trigger a note event
endin

instr  2
aEnv    transg  0.2, p3, -4, 0 ; decay envelope
aSig    oscil   aEnv, 400       ; 'ping' sound indicator
        out      aSig         ; send audio to output
endin

</CsInstruments>
<CsScore>
i 1 0 [3600*24*7]
</CsScore>
</CsoundSynthesizer>
;example by Iain McCurdy

```

Pitch Tracking

Csound currently provides five opcode options for pitch tracking. In ascending order of newness they are: [pitch](#), [pitchamdf](#), [pvspitch](#), [ptrack](#) and [plltrack](#). Related to these opcodes are [pvcent](#) and [centroid](#) but rather than track the harmonic fundamental, they track the spectral centroid of a signal. An example and suggested application for centroid is given a little later on in this chapter.

Each offers a slightly different set of features – some offer simultaneous tracking of both amplitude and pitch, some only pitch tracking. None of these opcodes provide more than one output for tracked frequency therefore none offer polyphonic tracking although in a polyphonic tone the fundamental of the strongest tone will most likely be tracked. Pitch tracking presents many more challenges than amplitude tracking therefore a degree of error can be expected and will be an issue that demands addressing. To get the best from any pitch tracker it is important to consider preparation of the input signal – either through gating or filtering – and also processing of the output tracking data, for example smoothing changes through the use of filtering opcode such as [port](#), [median](#) filtering to remove erratic and erroneous data and a filter to simply ignore obviously incorrect data. Parameters for these procedures will rely upon some prior knowledge of the input signal, the pitch range of an instrument for instance. A particularly noisy environment or a distant microphone placement might demand more aggressive noise gating. In general some low-pass filtering of the input signal will always help in providing a more stable frequency tracking signal. Something worth considering is that the attack portion of a note played on an acoustic instrument generally contains a lot of noisy, harmonically chaotic material. This will tend to result in slightly

chaotic movement in the pitch tracking signal, we may therefore wish to sense the onset of a note and only begin tracking pitch once the sustain portion has begun. This may be around 0.05 seconds after the note has begun but will vary from instrument to instrument and from note to note. In general lower notes will have a longer attack. However we do not really want to overestimate the duration of this attack stage as this will result in a sluggish pitch tracker. Another specialised situation is the tracking of pitch in singing – we may want to gate sibilant elements (sss, t etc.). *pvscent* can be useful in detecting the difference between vowels and sibilants.

pitch is the oldest of the pitch tracking opcodes on offer and provides the widest range of input parameters.

```
koct, kamp pitch asig, iupdte, ilo, ihi, idbthresh [, ifrqs] [, iconf]
[, istrt] [, iocts] [, iq] [, inptls] [, irolloff] [, iskip]
```

This makes it somewhat more awkward to use initially (although many of its input parameters are optional) but some of its options facilitate quite specialised effects. Firstly it outputs its tracking signal in oct format. This might prove to be a useful format but conversion to other formats is easy anyway. Apart from a number of parameters intended to fine tune the production of an accurate signal it allows us to specify the number of octave divisions used in quantising the output. For example if we give this a value of 12 we have created the basis of a simple chromatic *autotune* device. We can also quantise the procedure in the time domain using its *update period* input. Material with quickly changing pitch or vibrato will require a shorter update period (which will demand more from the CPU). It has an input control for *threshold of detection* which can be used to filter out and disregard pitch and amplitude tracking data beneath this limit. Pitch is capable of very good pitch and amplitude tracking results in real-time.

pitchamdf uses the so-called *Average Magnitude Difference Function* method. It is perhaps slightly more accurate than *pitch* as a general purpose pitch tracker but its CPU demand is higher.

pvspitch uses streaming FFT technology to track pitch. It takes an f-signal as input which will have to be created using the *pvsanal* opcode. At this step the choice of FFT size will have a bearing upon the performance of the *pvspitch* pitch tracker. Smaller FFT sizes will allow for faster tracking but with perhaps some inaccuracies, particularly with lower pitches whereas larger FFT sizes are likely to provide for more accurate pitch tracking at the expense of some time resolution. *pvspitch* tries to mimic certain functions of the human ear in how it tries to discern pitch. *pvspitch* works well in real-time but it does have a tendency to jump its output to the wrong octave – an octave too high – particularly when encountering vibrato.

ptrack also makes uses of streaming FFT but takes an normal audio signal as input, performing the FFT analysis internally. We still have to provide a value for FFT size with the same considerations mentioned above. *ptrack* is based on an algorithm by Miller Puckette, the co-creator of MaxMSP and creator of PD. *ptrack* also works well in real-time but it does have a tendency to jump to erroneous pitch tracking values when pitch is changing quickly or when encountering vibrato. Median filtering (using the *mediank* opcode) and filtering of outlying values might improve the results.

plltrack uses a phase-locked loop algorithm in detecting pitch. *plltrack* is another efficient real-time option for pitch tracking. It has a tendency to gliss up and down from very low frequency values at the start and end of notes, i.e. when encountering silence. This effect can be minimised by increasing its *feedback* parameter but this can also make pitch tracking unstable over sustained notes.

In conclusion, *pitch* is probably still the best choice as a general purpose pitch tracker, *pitchamdf*

is also a good choice. *pvs pitch*, *ptrack* and *plctrack* all work well in real-time but might demand additional processing to remove errors.

pvscent and *centroid* are a little different to the other pitch trackers in that, rather than try to discern the fundamental of a harmonic tone, they assess what the centre of gravity of a spectrum is. An application for this is in the identification of different instruments playing the same note. Softer, darker instruments, such as the french horn, will be characterised by a lower centroid to that of more shrill instruments, such as the violin.

Both opcodes use FFT. Centroid works directly with an audio signal input whereas *pvscent* requires an f-sig input. Centroid also features a trigger input which allows us to manually trigger it to update its output. In the following example we use centroid to detect individual drums sounds – bass drum, snare drum, cymbal – within a drum loop. We will use the dynamic amplitude trigger from earlier on in this chapter to detect when sound onsets are occurring and use this trigger to activate centroid and also then to trigger another instrument with a replacement sound. Each percussion instrument in the original drum loop will be replaced with a different sound: bass drums will be replaced with a kalimba/thumb piano sound, snare drums will be replaced by hand claps (a la TR-808), and cymbal sounds will be replaced with tambourine sounds. The drum loop used is *beats.wav* which can be found with the download of the Csound HTML manual (and within the Csound download itself). This loop is not ideal as some of the instruments coincide with one another – for example, the first consists of a bass drum and a snare drum played together. The *beat replacer* will inevitably make a decision one way or the other but is not advanced enough to detect both instruments playing simultaneously. The critical stage is the series of *if ... elseifs ...* at the bottom of instrument 1 where decisions are made about instruments' identities according to what centroid band they fall into. The user can fine tune the boundary division values to modify the decision making process. *centroid* values are also printed to the terminal when onsets are detected which might assist in this fine tuning.

EXAMPLE 14C04_Drum_Replacement.csd

```
<CsoundSynthesizer>
<CsOptions>
-dm0 -odac
</CsOptions>
<CsInstruments>

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

instr 1
asig diskin "beats.wav",1

iThreshold = 0.05
iWait      = 0.1*sr
kTimer     init iWait+1
iSampTim = 0.02           ; time across which RMS change is measured
kRms    rms    asig ,20
kRmsPrev delayk kRms,iSampTim ; rms from earlier
kChange = kRms - kRmsPrev   ; change (+ve or -ve)

if kTimer > iWait then          ; prevent double triggerings
; generate a trigger
kTrigger = kChange > iThreshold ? 1 : 0

```

```

; if trigger is generated, reset timer
kTimer = kTrigger == 1 ? 0 : kTimer
else
  kTimer += ksmps                      ; increment timer
  kTrigger = 0                          ; clear trigger
endif

ifftsize = 1024
; centroid triggered 0.02 after sound onset to avoid noisy attack
kDelTrig delayk kTrigger,0.02
kcenf centroid asig, kDelTrig, ifftsize ; scan centroid
  printk2 kcenf                  ; print centroid values
if kDelTrig==1 then
  if kcenf>0 && kcenf<2500 then    ; first freq. band
    event "i","Cowbell",0,0.1
  elseif kcenf<8000 then          ; second freq. band
    event "i","Clap",0,0.1
  else
    event "i","Tambourine",0,0.5
  endif
endif
endin

instr Cowbell
kenv1 transeg 1,p3*0.3,-30,0.2, p3*0.7,-30,0.2
kenv2 expon 1,p3,0.0005
kenv = kenv1*kenv2
ipw = 0.5
a1 vco2 0.65,562,2,0.5
a2 vco2 0.65,845,2,0.5
amix = a1+a2
iLPF2 = 10000
kcf expseg 12000,0.07,iLPF2,1,iLPF2
alpf butlp amix,kcf
abpf reson amix, 845, 25
amix dcblock2 (abpf*0.06*kenv1)+(alpf*0.5)+(amix*0.9)
amix buthp amix,700
amix = amix*0.5*kenv
out amix
endin

instr Clap
if frac(p1)==0 then
  event_i "i", p1+0.1, 0, 0.02
  event_i "i", p1+0.1, 0.01, 0.02
  event_i "i", p1+0.1, 0.02, 0.02
  event_i "i", p1+0.1, 0.03, 2
else
  kenv transeg 1,p3,-25,0
  iamp random 0.7,1
  anoise dust2 kenv*iamp, 8000
  iBPF = 1100
  ibw = 2000
  iHPF = 1000
  iLPF = 1
  kcf expseg 8000,0.07,1700,1,800,2,500,1,500
  asig butlp anoise,kcf*iLPF
  asig buthp asig,iHPF
  ares reson asig,iBPF,ibw,1
  asig dcblock2 (asig*0.5)+ares
  out asig
endif
endin

```

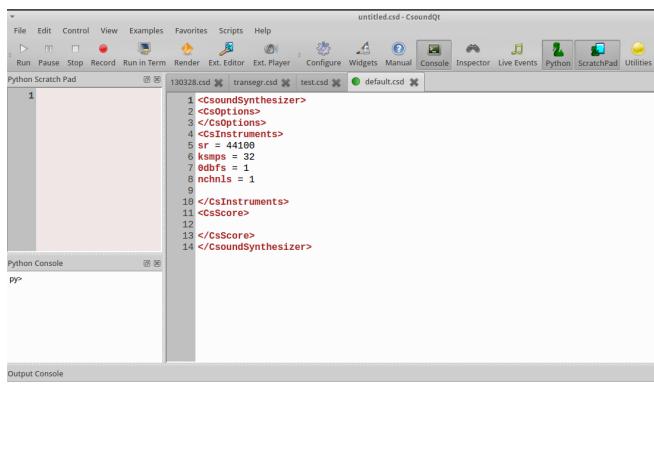
```
instr Tambourine
    asig tambourine    0.3,0.01 ,32, 0.47, 0, 2300 , 5600, 8000
                out    asig ;SEND AUDIO TO OUTPUTS
endin

</CsInstruments>
<CsScore>
i 1 0 10
</CsScore>
</CsoundSynthesizer>
;example by Iain McCurdy
```


14 D. PYTHON IN CSOUNDQT

If CsoundQt is built with PythonQt support,¹ it enables a lot of new possibilities, mostly in three main fields: interaction with the CsoundQt interface, interaction with widgets and using classes from Qt libraries to build custom interfaces in python.

If you start CsoundQt and can open the panels *Python Console* and *Python Scratch Pad*, you are ready to go.



The CsoundQt Python Object

As CsoundQt has formerly been called *QuteCsound*, this name can still be found in the sources. The *QuteCsound object* (called *PyQcsObject* in the sources) is the interface for scripting CsoundQt. All declarations of the class can be found in the file *pyqcsobject.h* in the sources.

It enables the control of a large part of CsoundQt's possibilities from the python interpreter, the python scratchpad, from scripts or from inside of a running Csound file via Csound's python opcodes.²

By default, a *PyQcsObject* is already available in the python interpreter of CsoundQt called "q". To use any of its methods, we can use a form like

¹If not, have a look at the [releases](#) page. Python 2.7 must be installed, too. For building CsoundQt with Python support, have a look at the descriptions in [CsoundQt's Wiki](#).

²See chapter 12 B for more information on the python opcodes and ctcsound.

```
q.stopAll()
```

The methods can be divided into four groups:

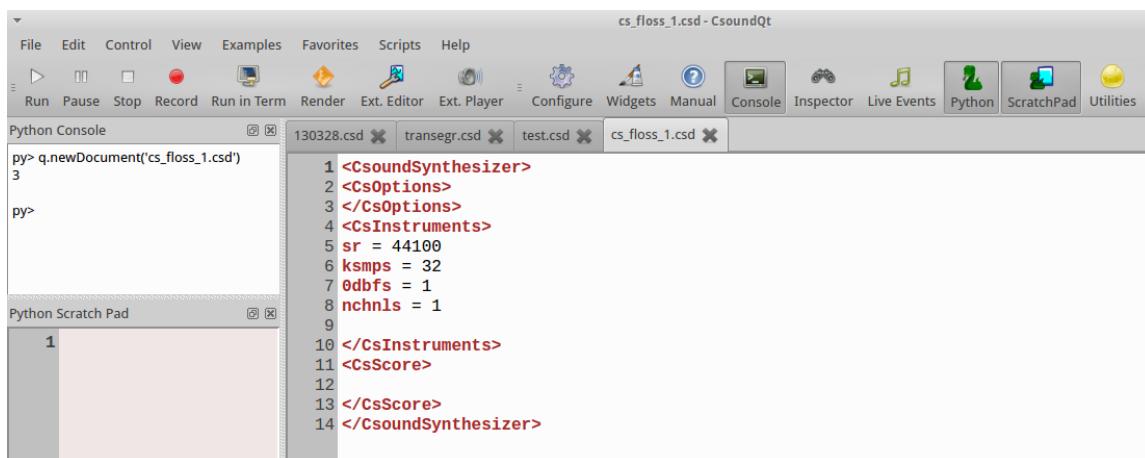
- access CsoundQt's interface (open or close files, start or stop performance etc)
- edit Csound files which has already been opened as tabs in CsoundQt
- manage CsoundQt's widgets
- interface with the running Csound engine

File and Control Access

If you have CsoundQt running on your computer, you should type the following code examples in the Python Console (if only one line) or the Python Scratch Pad (if more than one line of code).³

Create or Load a csd File

Type `q.newDocument('cs_floss_1.csd')` in your Python Console and hit the Return key. This will create a new csd file named `cs_floss_1.csd` in your working directory. And it also returns an integer (in the screenshot below: 3) as index for this file.



If you close this file and then execute the line `q.loadDocument('cs_floss_1.csd')`, you should see the file again as tab in CsoundQt.

Let us have a look how these two methods `newDocument` and `loadDocument` are described in the sources:

```
int newDocument(QString name)
int loadDocument(QString name, bool runNow = false)
```

The method `newDocument` needs a name as string ("QString") as argument, and returns an integer. The method `loadDocument` also takes a name as input string and returns an integer as index for this csd. The additional argument `runNow` is optional. It expects a boolean

³To evaluate multiple lines of Python code in the Scratch Pad, choose either Edit->Evaluate Section (Alt+E), or select and choose Edit->Evaluate Selection (Alt+Shift+E).

value (True/False or 1/0). The default is *false* which means “do not run immediately after loading”. So if you type instead `q.loadDocument('cs_floss_1.csd', True)` or `q.loadDocument('cs_floss_1.csd', 1)`, the csd file should start immediately.

Run, Pause or Stop a csd File

For the next methods, we first need some more code in our csd. So let your `cs_floss_1.csd` look like this:

EXAMPLE 14B01_run_pause_stop.csd

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

giSine      ftgen      0, 0, 1024, 10, 1

instr 1
kPitch      expseg    500, p3, 1000
aSine       oscil     .2, kPitch, giSine
              out        aSine
endin
</CsInstruments>
<CsScore>
i 1 0 10
</CsScore>
</CsoundSynthesizer>
```

This instrument performs a simple pitch glissando from 500 to 1000 Hz in ten seconds. Now make sure that this csd is the currently active tab in CsoundQt, and execute this:

```
q.play()
```

This starts the performance. If you do nothing, the performance will stop after ten seconds. If you type instead after some seconds

```
q.pause()
```

the performance will pause. The same task `q.pause()` will resume the performance. Note that this is different from executing `q.play()` after `q.pause()` ; this will start a new performance. With

```
q.stop()
```

you can stop the current performance.

Access to Different csd Tabs via Indices

The `play()`, `pause()` and `stop()` method, as well as other methods in CsoundQt's integrated Python, allow also to access csd file tabs which are not currently active. As we saw in the creation of a new csd file by `q.newDocument('cs_floss_1.csd')`, each of them gets an index. This index allows universal access to all csd files in a running CsoundQt instance.

First, create a new file `cs_floss_2.csd`, for instance with this code:

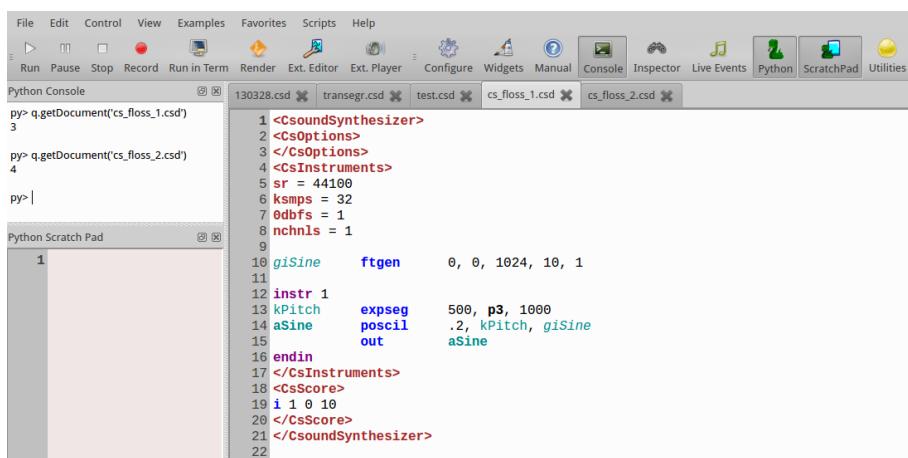
EXAMPLE 14B02_tabs.csd

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

giSine      ftgen      0, 0, 1024, 10, 1

instr 1
kPitch      expseg     500, p3, 1000
aSine        poscil     .2, kPitch, giSine
                out        aSine
endin
</CsInstruments>
<CsScore>
i 1 0 10
</CsScore>
</CsoundSynthesizer>
```

Now get the index of these two tabs in executing `q.getDocument('cs_floss_1.csd')` and `q.getDocument('cs_floss_2.csd')`. This will show something like this:

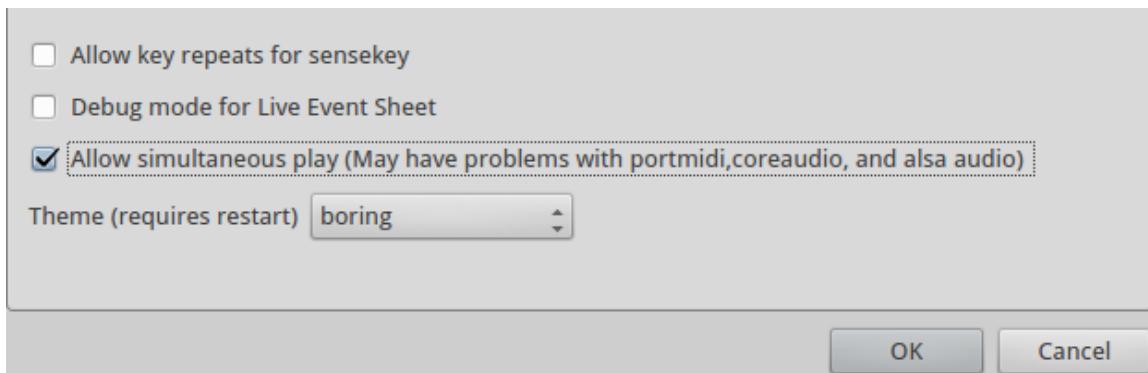


So in my case the indices are 3 and 4.⁴ Now you can start, pause and stop any of these files with tasks like these:

⁴If you have less or more csd tabs already while creating the new files, the index will be lower or higher.

```
q.play(3)
q.play(4)
q.stop(3)
q.stop(4)
```

If you have checked *Allow simultaneous play* in CsoundQt's *Configure->General ...*



.. you should be able to run both csds simultaneously. To stop all running files, use:

```
q.stopAll()
```

To set a csd as active, use `setDocument(index)`. This will have the same effect as clicking on the tab.

Send Score Events

Now comment out the score line in the file `cs_floss_2.csd`, or simply remove it. When you now start Csound, this tab should run. Now execute this command:

```
q.sendEvent('i 1 0 2')
```

This should trigger instrument 1 for two seconds.

Query File Name or Path

In case you need to know the name⁵ or the path of a csd file, you have these functions:

```
getFileName()
getFilePath()
```

Calling the method without any arguments, it refers to the currently active csd. An index as argument links to a specific tab. Here is a Python code snippet which returns indices, file names and file paths of all tabs in CsoundQt:

```
index = 0
while q.getFileName(index):
    print 'index = %d' % index
    print ' File Name = %s' % q.getFileName(index)
```

⁵Different to most usages, *name* means here the full path including the file name.

```
print ' File Path = %s' % q.getFilePath(index)
index += 1
```

Which returns for instance:

```
index = 0
File Name = /home/jh/Joachim/Stuecke/30Carin/csound/130328.csd
File Path = /home/jh/Joachim/Stuecke/30Carin/csound
index = 1
File Name = /home/jh/src/csoundmanual/examples/transegr.csd
File Path = /home/jh/src/csoundmanual/examples
index = 2
File Name = /home/jh/Desktop/test.csd
File Path = /home/jh/Desktop
```

Get and Set csd Text

One of the main features of Python scripting in CsoundQt is the ability to edit any section of a csd file. There are several get functions, to query text, and also set functions to change or insert text.

Get Text from a csd File

Make sure your `cs_floss_2.csd` is the active tab, and execute the following python code lines:

```
q.getCsd()
q.getOrc()
q.getSco()
```

You will get the full visible csd, the orc or the sco part as a unicode string.

You can also get the text for the `<CsOptions>`, the text for CsoundQt's widgets and presets, or the full text of this csd:

```
q.getOptionsText()
q.getWidgetsText()
q.getPresetsText()
q.getFullText()
```

If you select some text or some widgets, you will get the selection with these commands:

```
q.getSelectedText()
q.getSelectedWidgetsText()
```

As usual, you can specify any of the loaded csds via its index. So calling `q.getOrc(3)` instead of `q.getOrc()` will return the orc text of the csd with index 3, instead of the orc text of the currently active csd.

Set Text in a csd File

Set the cursor anywhere in your active csd, and execute the following line in the Python Console:

```
q.insertText('my nice insertion')
```

You will see your nice insertion in the csd file. In case you do not like it, you can choose Edit->Undo. It does not make a difference for the CsoundQt editor whether the text has been typed by hand, or by the internal Python script facility.

Text can also be inserted to individual sections using the functions:

```
setCsd(text)
setFullText(text)
setOrc(text)
setSco(text)
setWidgetsText(text)
setPresetsText(text)
setOptionsText(text)
```

Note that the whole section will be overwritten with the string *text*.

Opcode Exists

You can ask whether a string is an opcode name, or not, with the function *opcodeExists*, for instance:

```
py> q.opcodeExists('line')
True
py> q.opcodeExists('OSCsend')
True
py> q.opcodeExists('Line')
False
py> q.opcodeExists('Joe')
NotYet
```

Example: Score Generation

A typical application for setting text in a csd is to generate a score. There have been numerous tools and programs to do this, and it can be very pleasant to use CsoundQt's Python scripting for this task. Let us modify our previous instrument first to make it more flexible:

EXAMPLE 14B03_score_generated.csd

```
<CsoundSynthesizer>
<CsOptions>
</CsOptions>
</CsInstruments>
sr = 44100
ksmps = 32
```

```

0dbfs = 1
nchnls = 1

giSine      ftgen      0, 0, 1024, 10, 1

instr 1
iOctStart =           p4 ;pitch in octave notation at start
iOctEnd   =           p5 ;and end
iDbStart  =           p6 ;dB at start
iDbEnd    =           p7 ;and end
kPitch    expseg      cpsoct(iOctStart), p3, cpsoct(iOctEnd)
kEnv      linseg      iDbStart, p3, iDbEnd
aSine     poscil      ampdb(kEnv), kPitch, giSine
iFad      random      p3/20, p3/5
aOut      linen       aSine, iFad, p3, iFad
          out        aOut
endin
</CsInstruments>
<CsScore>
i 1 0 10 ;will be overwritten by the python score generator
</CsScore>
</CsoundSynthesizer>

```

The following code will now insert 30 score events in the score section:

```

from random import uniform
numScoEvents = 30
sco = ''
for ScoEvent in range(numScoEvents):
    start = uniform(0, 40)
    dur = 2**uniform(-5, 3)
    db1, db2 = [uniform(-36, -12) for x in range(2)]
    oct1, oct2 = [uniform(6, 10) for x in range(2)]
    scoLine = 'i 1 %f %f %f %d %d\n' % (start,dur,oct1,oct2,db1,db2)
    sco = sco + scoLine
q.setSco(sco)

```

This generates a texture with either falling or rising gliding pitches. The durations are set in a way that shorter durations have a bigger probability than larger ones. The volume and pitch ranges allow many variations in the simple shape.

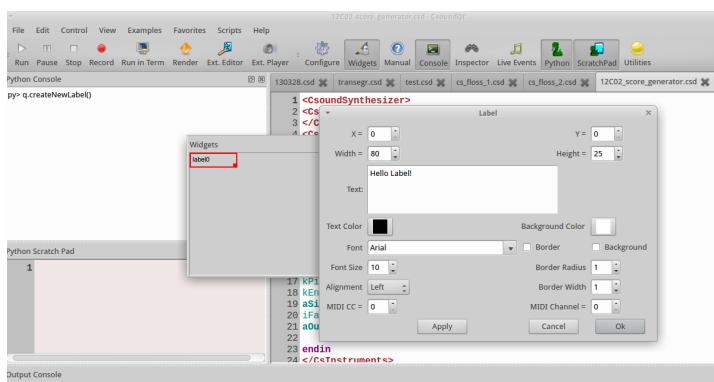
Widgets

Creating a Label

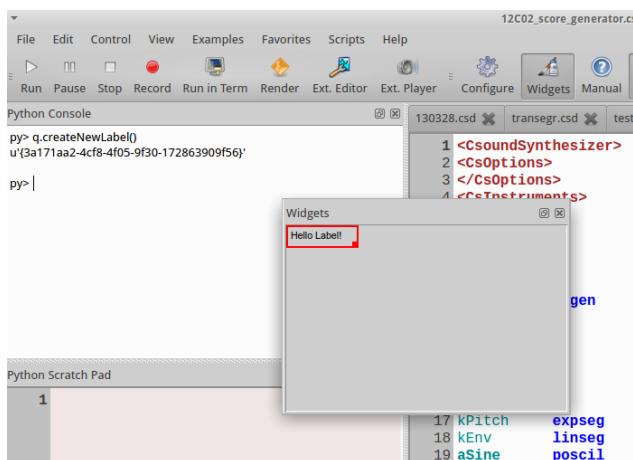
Click on the *Widgets* button to see the widgets panel. Then execute this command in the Python Console:

```
q.createNewLabel()
```

The properties dialog of the label pops up. Type *Hello Label!* or something like this as text.



When you click *Ok*, you will see the label widget in the panel, and a strange unicode string as return value in the Python Console:



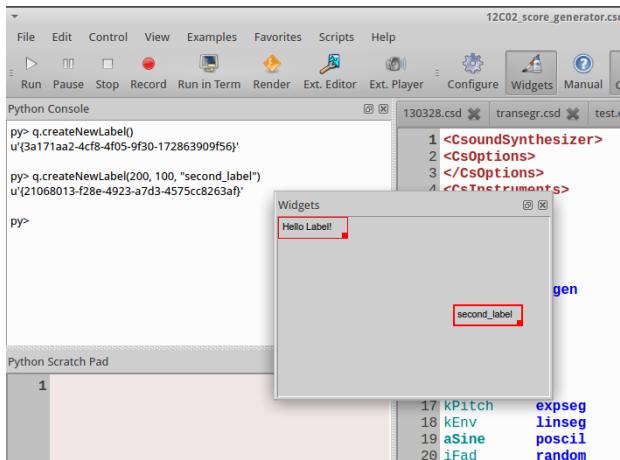
The string `u'3a171aa2-4cf8-4f05-9f30-172863909f56'` is a “universally unique identifier” (uuid). Each widget can be accessed by this ID.

Specifying the Common Properties as Arguments

Instead of having a live talk with the properties dialog, we can specify all properties as arguments for the `createNewLabel` method:

```
q.createNewLabel(200, 100, "second_label")
```

This should be the result:



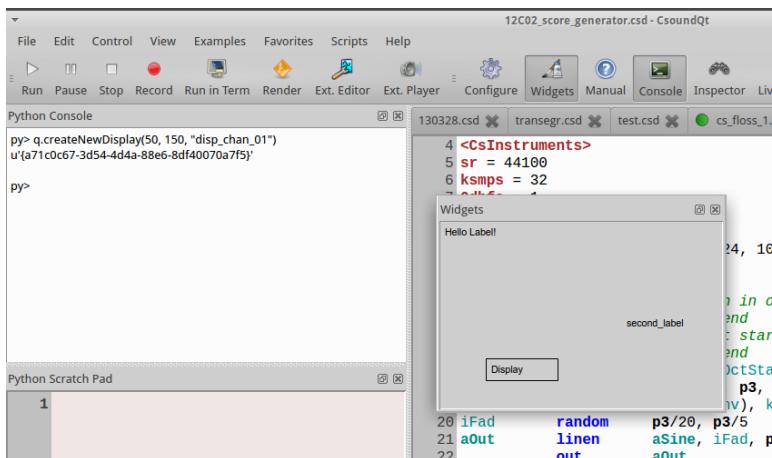
A new label has been created—without opening the properties dialog—at position x=200 y=100⁶ with the name `second_label`. If you want to create a widget not in the active document, but in another tab, you can also specify the tab index. The following command will create a widget at the same position and with the same name in the first tab:

```
q.createNewLabel(200, 100, "second_label", 0)
```

Setting the Specific Properties

Each widget has a xy position and a channel name.⁷ But the other properties depend on the type of widget. A Display has name, width and height, but no resolution like a SpinBox. The function `setWidgetProperty` refers to a widget via its ID and sets a property. Let us try this for a Display widget. This command creates a Display widget with channel name "disp_chan_01" at position x=50 y=150:

```
q.createNewDisplay(50, 150, "disp_chan_01")
```



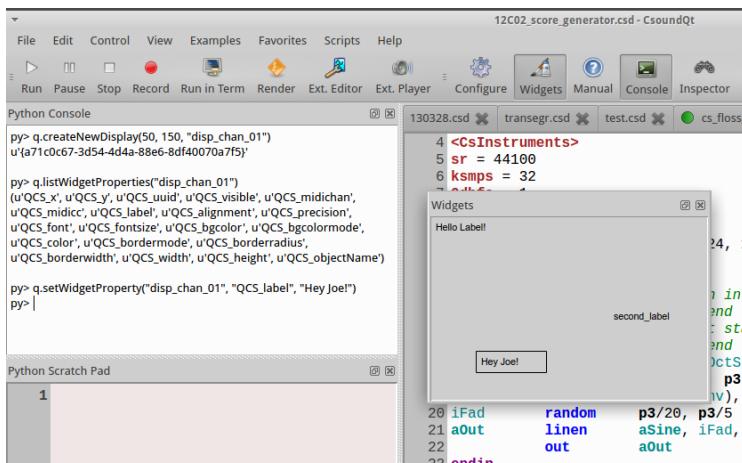
And this sets the text to a new string:⁸

⁶Pixels from left and from top.

⁷Only a label does not have a channel name. So as we saw, in case of a label the name is its displayed text.

⁸For the main property of a widget (text for a Display, number for Sliders, SpinBoxes etc) you can also use the `setChannelString` and `setChannelValue` method. See below at *Getting and Setting Channel Values*

```
q.setProperty("disp_chan_01", "QCS_label", "Hey Joe!")
```

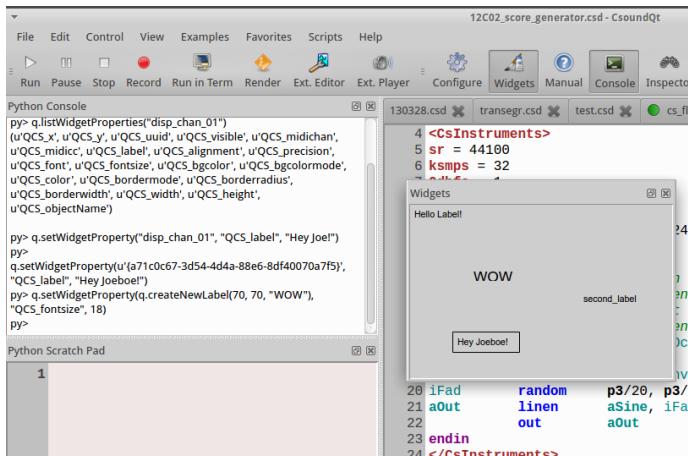


The `setProperty` method needs the ID of a widget first. This can be expressed either as channel name (`disp_chan_01`) as in the command above, or as uuid. As I got the string `u'{a71c0c67-3d54-4d4a-88e6-8df40070a7f5}'` as uuid, I can also write:

```
q.setProperty(u'{a71c0c67-3d54-4d4a-88e6-8df40070a7f5}',  
             'QCS_label', 'Hey Joeboe!')
```

For humans, referring to the channel name as ID is certainly preferable.⁹ But as the `createNew...` method returns the uuid, you can use it implicitly, for instance in this command:

```
q.setProperty(q.createNewLabel(70, 70, "WOW"), "QCS_fontsize", 18)
```



Getting the Property Names and Values

How can we know that the visible text of a Display widget is called `QCS_label` and the fontsize `QCS_fontsize`? If we do not know the name of a property, we can ask CsoundQt for it via the function `listWidgetProperties`:

⁹Note that two widgets can share the same channel name (for instance a slider and a spinbox). In this case, referring to a widget via its channel name is not possible at all.

```
py> q.listWidgetProperties("disp_chan_01")
(u'QCS_x', u'QCS_y', u'QCS_uuid', u'QCS_visible', u'QCS_midichan',
 u'QCS_midicc', u'QCS_label', u'QCS_alignment', u'QCS_precision',
 u'QCS_font', u'QCS_fontsize', u'QCS_bgcolor', u'QCS_bgcolormode',
 u'QCS_color', u'QCS_bordermode', u'QCS_borderradius', u'QCS_borderwidth',
 u'QCS_width', u'QCS_height', u'QCS_objectName')
```

listWidgetProperties returns all properties in a tuple. We can query the value of a single property with the function *getWidgetProperty*, which takes the uuid and the property as inputs, and returns the property value. So this code snippet asks for all property values of our Display widget:

```
widgetID = "disp_chan_01"
properties = q.listWidgetProperties(widgetID)
for property in properties:
    propVal = q.getWidgetProperty(widgetID, property)
    print property + ' = ' + str(propVal)
```

Returns:

```
QCS_x = 50
QCS_y = 150
QCS_uuid = {a71c0c67-3d54-4d4a-88e6-8df40070a7f5}
QCS_visible = True
QCS_midichan = 0
QCS_midicc = -3
QCS_label = Hey Joeboe!
QCS_alignment = left
QCS_precision = 3
QCS_font = Arial
QCS_fontsize = 10
QCS_bgcolor = #ffffff
QCS_bgcolormode = False
QCS_color = #000000
QCS_bordermode = border
QCS_borderradius = 1
QCS_borderwidth = 1
QCS_width = 80
QCS_height = 25
QCS_objectName = disp_chan_01
```

Get the UUIDs of all Widgets

For getting the uuid strings of all widgets in the active csd tab, type

```
q.getWidgetUuids()
```

As always, the uuid strings of other csd tabs can be accessed via the index.

Some Examples for Creating and Modifying Widgets

Create a new slider with the channel name *level* at position 10,10 in the (already open but not necessarily active) document *test.csd*:

```
q.createNewSlider(10, 10, "level", q.getDocument("test.csd"))
```

Create ten knobs with the channel names *partial_1*, *partial_2* etc, and the according labels *amp_part_1*, *amp_part_2* etc in the currently active document:

```
for no in range(10):
    q.createNewKnob(100*no, 5, "partial_"+str(no+1))
    q.createNewLabel(100*no+5, 90, "amp_part_"+str(no+1))
```

Alternatively, you can store the uuid strings while creating:

```
knobs, labels = [], []
for no in range(10):
    knobs.append(q.createNewKnob(100*no, 5, "partial_"+str(no+1)))
    labels.append(q.createNewLabel(100*no+5, 90, "amp_part_"+str(no+1)))
```

The variables *knobs* and *labels* now contain the IDs:

```
py> knobs
[u'{8d10f9e3-70ce-4953-94b5-24cf8d6f6adb}', 
 u'{d1c98b52-a0a1-4f48-9bca-bac55dad0de7}', 
 u'{b7bf4b76-baff-493f-bc1f-43d61c4318ac}', 
 u'{1332208d-e479-4152-85a8-0f4e6e589d9d}', 
 u'{428cc329-df4a-4d04-9cea-9be3e3c2a41c}', 
 u'{1e691299-3e24-46cc-a3b6-85fdd40eac15}', 
 u'{a93c2b27-89a8-41b2-befb-6768cae6f645}', 
 u'{26931ed6-4c28-4819-9b31-4b9e0d9d0a68}', 
 u'{874beb70-b619-4706-a465-12421c6c8a85}', 
 u'{3da687a9-2794-4519-880b-53c2f3b67b1f}']

py> labels
[u'{9715ee01-57d5-407d-b89a-bae2fc6acecf}', 
 u'{71295982-b5e7-4d64-9ac5-b8fbcffbd254}', 
 u'{09e924fa-2a7c-47c6-9e17-e710c94bd2d1}', 
 u'{2e31dbfb-f3c2-43ab-ab6a-f47abb4875a3}', 
 u'{adfe3aef-4499-4c29-b94a-a9543e54e8a3}', 
 u'{b5760819-f750-411d-884c-0bad16d68d09}', 
 u'{c3884e9e-f0d8-4718-8fcb-66e82456f0b5}', 
 u'{c1401878-e7f7-4e71-a097-e92ada42e653}', 
 u'{a7d14879-1601-4789-9877-f636105b552c}', 
 u'{ec5526c4-0fda-4963-8f18-1c7490b0a667}'
```

Move the first knob 200 pixels downwards:

```
q.setProperty(knobs[0], "QCS_y", q.getProperty(knobs[0],
"QCS_y") + 200)
```

Modify the maximum of each knob so that the higher partials have less amplitude range (set maximum to 1, 0.9, 0.8, ... 0.1):

```
for knob in range(10):
    q.setProperty(knobs[knob], "QCS_maximum", 1-knob/10.0)
```

Deleting widgets

You can delete a widget using the method *destroyWidget*. You have to pass the widget's ID, again either as channel name or (better) as uuid string. This will remove the first knob in the example above:

```
q.destroyWidget("partial_1")
```

This will delete all knobs:

```
for w in knobs:
    q.destroyWidget(w)
```

And this will delete all widgets of the active document:

```
for w in q.getWidgetUuids():
    q.destroyWidget(w)
```

Getting and Setting Channel Names and Values

After this cruel act of destruction, let us again create a slider and a display:

```
py> q.createNewSlider(10, 10, "level")
u'{b0294b09-5c87-4607-afda-2e55a8c7526e}'
py> q.createNewDisplay(50, 10, "message")
u'{a51b438f-f671-4108-8cdb-982387074e4d}'
```

Now we will ask for the values of these widgets¹⁰ with the methods `getChannelValue` and `getChannelString`:

```
py> q.getChannelValue('level')
0.0
py> q.getChannelString("level")
u''
py> q.getChannelValue('message')
0.0
py> q.getChannelString('message')
u'Display'
```

As you see, it depends on the type of the widget whether to query its value by `getChannelValue` or `getChannelString`. Although CsoundQt will not return an error, it makes no sense to ask a slider for its string (as its value is a number), and a display for its number (as its value is a string).

With the methods `setChannelValue` and `setChannelString` we can change the main content of a widget very easily:

```
py> q.setChannelValue("level", 0.5)
py> q.setChannelString("message", "Hey Joe again!")
```

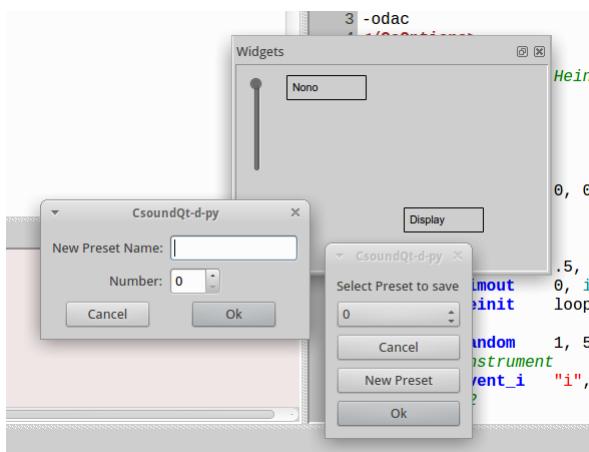
This is much more handy than the general method using `setWidgetProperty`:

```
py> q.setWidgetProperty("level", "QCS_value", 1)
py> q.setWidgetProperty("message", "QCS_label", "Nono")
```

Presets

Now right-click in the widget panel and choose *Store Preset -> New Preset*:

¹⁰Here again accessed by the channel name. Of course accessing by uid would also be possible (and more safe, as explained above).



You can (but need not) enter a name for the preset. The important thing here is the number of the preset (here 0). - Now change the value of the slider and the text of the display widget. Save again as preset, now being preset 1. - Now execute this:

```
q.loadPreset(0)
```

You will see the content of the widgets reloaded to the first preset. Again, with

```
q.loadPreset(1)
```

you can switch to the second one.

Like all python scripting functions in CsoundQt, you can not only use these methods from the Python Console or the Python Scratch Pad, but also from inside any csd. This is an example how to switch all the widgets to other predefined states, in this case controlled by the score. You will see the widgets for the first three seconds in Preset 0, then for the next three seconds in Preset 1, and finally again in Preset 0:

EXAMPLE 14B04_presets.csd

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

pyinit

instr loadPreset
    index = p4
    pycalli "q.loadPreset", index
endin

</CsInstruments>
<CsScore>
i "loadPreset" 0 3 0
i "loadPreset" + . 1
i "loadPreset" + . 0
</CsScore>
</CsoundSynthesizer>
;example by tarmo johannes and joachim heintz
```

Csound Functions

Several functions can interact with the Csound engine, for example to query information about it. Note that the functions `getSampleRate`, `getKsmmps`, `getNumChannels` and `getCurrentCsound` refer to a *running* instance of Csound.

```
py> q.getVersion() # CsoundQt API version
u'1.0'
py> q.getSampleRate()
44100.0
py> q.getKsmmps()
32
py> q.getNumChannels()
1
py> q.getCurrentCsound()
CSOUND (C++ object at: 0x2fb5670)
```

With `getCsChannel`, `getCsStringChannel` and `setCsChannel` you can access csound channels directly, independently from widgets. They are useful when testing a csd for use with the Csound API (in another application, a csLapds or Cabbage plugin, Android application) or similar. These are some examples, executed on a running csd instance:

```
py> q.getCsChannel('my_num_chn')
0.0
py> q.getCsStringChannel('my_str_chn')
u''

py> q.setCsChannel('my_num_chn', 1.1)
py> q.setCsChannel('my_str_chn', 'Hey Csound')

py> q.getCsChannel('my_num_chn')
1.1
py> q.getCsStringChannel('my_str_chn')
u'Hey Csound'
```

If you have a function table in your running Csound instance which has for instance been created with the line `giSine ftgen 1, 0, 1024, 10, 1`, you can query `getTableArray` like this:

```
py> q.getTableArray(1)
MYFLT (C++ object at: 0x35d1c58)
```

Finally, you can register a Python function as a callback to be executed in between processing blocks for Csound. The first argument should be the text that should be called on every pass. It can include arguments or variables which will be evaluated every time. You can also set a number of periods to skip to avoid.

```
registerProcessCallback(QString func, int skipPeriods = 0)
```

You can register the python text to be executed on every Csound control block callback, so you can execute a block of code, or call any function which is already defined.

Creating Own GUIs with PythonQt

One of the very powerful features of using Python inside CsoundQt is the ability to build own GUIs. This is done via the [PythonQt](#) library which gives you access to the Qt toolkit via Python. We will show some examples here. Have a look in the *Scripts* menu in CsoundQt to find much more (you will find the code in the *Editor* submenu).

Dialog Box

Sometimes it is practical to ask from user just one question - number or name of something and then execute the rest of the code (it can be done also inside a csd with python opcodes). In Qt, the class to create a dialog for one question is called [QInputDialog](#).

To use this or any other Qt classes, it is necessary to import the PythonQt and its Qt submodules. In most cases it is enough to add this line:

```
from PythonQt.Qt import *
```

or

```
from PythonQt.QtGui import *
```

At first an object of [QInputDialog](#) must be defined, then you can use its methods *getInt*, *getDouble*, *getItem* or *getText* to read the input in the form you need. This is a basic example:

```
from PythonQt.Qt import *

inpdia = QInputDialog()
myInt = inpdia.getInt(inpdia, "Example 1", "How many?")
print myInt
# example by tarmo johannes
```

Note that the variable *myInt* is now set to a value which remains in your Python interpreter. Your Python Console may look like this when executing the code above, and then ask for the value of *myInt*:

```
py>
12
Evaluated 5 lines.
py> myInt
12
```

Depending on the value of *myInt*, you can do funny or serious things. This code re-creates the Dialog Box whenever the user enters the number 1:

```
from PythonQt.Qt import *

def again():
    inpdia = QInputDialog()
    myInt = inpdia.getInt(inpdia, "Example 1", "How many?")
    if myInt == 1:
        print "If you continue to enter '1'"
        print "I will come back again and again."
```

```

    again()
else:
    print "Thanks - Leaving now."
again()
# example by joachim heintz

```

A simple example follows showing how an own GUI can be embedded in your Csound code. Here, Csound waits for the user input, and then prints out the entered value as the Csound variable giNumber:

EXAMPLE 14B05_dialog.csd

```

<CsoundSynthesizer>
<CsOptions>
-n
</CsOptions>
<CsInstruments>
ksmps = 32

pyinit
pyruni {{
from PythonQt.Qt import *
dia = QInputDialog()
dia.setDoubleDecimals(4)
} }

giNumber pyevali {{
dia.getDouble(dia,"CS question","Enter number: ")
}} ; get the number from Qt dialog

instr 1
    print giNumber
endin

</CsInstruments>
<CsScore>
i 1 0 0
</CsScore>
</CsoundSynthesizer>
;example by tarmo johannes

```

More complex examples can be found in CsoundQt's *Scripts* menu.

List of PyQcsObject Methods in CsoundQt

Load/Create/Activate a csd File

```

int loadDocument(QString name, bool runNow = false)
int getDocument(QString name = "")
int newDocument(QString name)
void setDocument(int index)

```

Play/Pause/Stop a csd File

```
void play(int index = -1, bool realtime = true)
void pause(int index = -1)
void stop(int index = -1)
void stopAll()
```

Send Score Events

```
void sendEvent(int index, QString events)
void sendEvent(QString events)
void schedule(QVariant time, QVariant event)
```

Query File Name/Path

```
QString getFileName(int index = -1)
QString getPath(int index = -1)
```

Get csd Text

```
QString getSelectedText(int index = -1, int section = -1)
QString getCsd(int index = -1)
QString getFullText(int index = -1)
QString getOrc(int index = -1)
QString getSco(int index = -1)
QString getWidgetsText(int index = -1)
QString getSelectedWidgetsText(int index = -1)
QString getPresetsText(int index = -1)
QString getOptionsText(int index = -1)
```

Set csd Text

```
void insertText(QString text, int index = -1, int section = -1)
void setCsd(QString text, int index = -1)
void setFullText(QString text, int index = -1)
void setOrc(QString text, int index = -1)
void setSco(QString text, int index = -1)
void setWidgetsText(QString text, int index = -1)
void setPresetsText(QString text, int index = -1)
void setOptionsText(QString text, int index = -1)
```

Opcode Exists

```
bool opcodeExists(QString opcodeName)
```

Create Widgets

```
QString createNewLabel(
    int x = 0, int y = 0, QString channel = QString(), int index = -1
)
QString createNewDisplay(
    int x = 0, int y = 0, QString channel = QString(), int index = -1
)
QString createNewScrollNumber(
    int x = 0, int y = 0, QString channel = QString(), int index = -1
)
QString createNewLineEdit(
    int x = 0, int y = 0, QString channel = QString(), int index = -1
)
QString createNewSpinBox(
    int x = 0, int y = 0, QString channel = QString(), int index = -1
)
QString createNewSlider(
    QString channel, int index = -1
)
QString createNewSlider(
    int x = 0, int y = 0, QString channel = QString(), int index = -1
)
QString createNewButton(
    int x = 0, int y = 0, QString channel = QString(), int index = -1
)
QString createNewKnob(
    int x = 0, int y = 0, QString channel = QString(), int index = -1
)
QString createNewCheckBox(
    int x = 0, int y = 0, QString channel = QString(), int index = -1
)
QString createNewMenu(
    int x = 0, int y = 0, QString channel = QString(), int index = -1
)
QString createNewMeter(
    int x = 0, int y = 0, QString channel = QString(), int index = -1
)
QString createNewConsole(
    int x = 0, int y = 0, QString channel = QString(), int index = -1
)
QString createNewGraph(
    int x = 0, int y = 0, QString channel = QString(), int index = -1
)
QString createNewScope(
    int x = 0, int y = 0, QString channel = QString(), int index = -1
)
```

Query Widgets

```
QVariant getWidgetProperty(QString widgetid, QString property, int index = -1)
double getChannelValue(QString channel, int index = -1)
QString getChannelString(QString channel, int index = -1)
QStringList listWidgetProperties(QString widgetid, int index = -1)
QStringList getWidgetUuids(int index = -1)
```

Modify Widgets

```
void setWidgetProperty(
    QString widgetid, QString property, QVariant value, int index = -1
)
void setChannelValue(QString channel, double value, int index = -1)
void setChannelString(QString channel, QString value, int index = -1)
```

Delete Widgets

```
bool destroyWidget(QString widgetid)
```

Presets

```
void loadPreset(int presetIndex, int index = -1)
```

Live Event Sheet

```
QuteSheet* getSheet(int index = -1, int sheetIndex = -1)
QuteSheet* getSheet(int index, QString sheetName)
```

Csound / API

```
QString getVersion()
void refresh()
void setCsChannel(QString channel, double value, int index = -1)
void setCsChannel(QString channel, QString value, int index = -1)
double getCsChannel(QString channel, int index = -1)
QString getCsStringChannel(QString channel, int index = -1)
```

```
CSOUND* getCurrentCsound()
double getSampleRate(int index = -1)
int getKsmprs(int index = -1)
int getNumChannels(int index = -1)
MYFLT *getTableArray(int ftable, int index = -1)
void registerProcessCallback(
    QString func, int skipPeriods = 0, int index = -1
)
```

14 E. GLOSSARY

Math Symbols

Multiplication in formulars is usually denoted with the dot operator:

$$2 \cdot 3 = 6$$

In text, the `*` is also used (as in Csound and other programming languages), or with the cross `x`.

Proportionality is written as: \propto .

Csound Terms

block size is the number of samples which are processed as vector or “block”. In Csound, we usually speak of `ksmps`: The number of samples in one control period.

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. For a sample rate of 44100 Hz and a `ksmps` value of 32, the time for one control cycle is $32/44100 = 0.000726$ seconds. See the chapter about [Initialization And Performance Pass](#) for a detailed discussion.

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`. For a sample rate of 44100 Hz and a `ksmps` value of 32, the control rate is 1378.125, so 1378.125 control cycles will be performed in one second. (Note that this value is not necessarily an integer, whilst `ksmps` is always an integer.)

.csd file is a text file containing a Csound program to be compiled and run by Csound. This file format contains several sections or *tags* (similar to XML or HTML), amongst them the *CsOptions* (Csound options), the *CsInstruments* (a collection of the Csound instruments) and the *CsScore* (the Csound score).

DSP means *Digital Signal Processing* and is used as a general term to describe any modification we apply on sounds in the digital domain.

f-statement or **function table statement** is a score line which starts with "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). (This is usually not any more required since Csound now runs "endless" with empty score.)

frequency domain means to look at a signal considering its frequency components. The mathematical procedure to transform a time-domain signal into frequency-domain is called ""Fourier Transform**. See the chapters about [Additive Synthesis](#) and about [Spectral Processing](#).

functional style is a way of coding where a function is written and the arguments of the function following in parentheses behind. Traditionally, Csound uses another convention to write code, but since Csound 6 functional style can be used as well. See the [functional syntax](#) chapter for more information.

GEN routine is a subroutine which generates a **function table** (mostly called *buffer* in other audio programming languages). GEN Routines are very different; they can load a sound file (GEN01), create segmented lines (GEN05 and others), composite waveforms (GEN10 and others), window functions (GEN20) or random distributions (GEN40). See the chapter about [function tables](#) and the [Gen Routines Overview](#) in the Csound Manual.

GUI Graphical User Interface refers to a system of on-screen sliders, buttons etc. used to interact with Csound, normally in real-time.

i-time or **init-time** or **i-rate** denotes the moment in which an instrument instance is initialized. In this initialization all 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 is a basic unit in Csound to perform any job, for instance generate noise, read an audio file, create an envelope or oscillate through a table. In other audio programming languages it is called UGen (Unit Generator) or object. An opcode can also be compared to a build-in function (e.g. in Python), whereas a User Defined Opcode (UDO) can be compared to a function which is written by the user. For an overview, see the [Opcode Guide](#).

options comprised as **csound options** and also called **command line flags** contain important decisions about how Csound has to run a .csd file. The -o option, for instance, tells Csound whether to output audio in realtime to the audio card, or to a sound file instead. See the [overview in the Csound Manual](#) for a detailed list of these options. Options are usually specified in the *CsOptions* tag of a .csd file. Modern frontends mostly pass the options to Csound via their settings.

orchestra is a collection of Csound instruments in a program, or referring to the .csd file, the *CsInstruments* tag. The term is somehow outdated, as it points to the early years of Csound where an .orc file was separated from the .sco (score) file.

p-field refers to the score section of a .csd file. A *p-field* can be compared to a column in a spread

sheet or table. An instrument, called by a line of score, receives any *p-field* parameter as *p1*, *p2* etc: *p1* will receive the parameter of the first column, *p2* the parameter of the second column, and so on.

performance pass see **control cycle**

score as in the Csound score, is the section of Csound code where events are written in the score language (which is completely different from the Csound orchestra language). The main events are *instrument* event, where each line starts with the character **i**. Another type of events is the **f** event which creates a function table. In modern Csound usage the score can be omitted, as all score jobs can also be done from inside the Csound instruments. See the [score chapter](#) for more information.

time domain means to look at a signal considering the changes of amplitudes over time. It is the common way to plot audio signals (time as x-axis, amplitudes as y-axis).

time stretching can be done in various ways in Csound. See [filescal](#), [sndwarp](#), [waveset](#), [pvstanal](#), [mincer](#), [pvsfread](#), [pvsdiskin](#) and the Granular Synthesis opcodes.

UDO or User-Defined Opcode is the definition of an opcode written in the Csound language itself. See the [UDO chapter](#) for more information.

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 of options are available for the creation of widgets usable by Csound, from its own built-in FLTK widgets to those provided by front-ends such as CsoundQt, Cabbage and Blue.

14 F. LINKS

Downloads

Csound FLOSS Manual Files: <https://csound-flossmanual.github.io/>

Csound: <http://csound.com/download.html>

Csound source code: <http://github.com/csound/csound>

User Defined Opcodes for Csound: <https://github.com/kunstmusik/libsyi>, <http://github.com/csudo/csudo> and others

CsoundQt: <http://github.com/CsoundQt/CsoundQt/releases>

Cabbage: <http://cabbageaudio.com/>

Blue: <http://blue.kunstmusik.com/>

WinXound: <http://mnt.conts.it/winxound/>

Community

The [Csound community home page](#) is the main place for news, basic infos, links and more.

The [Csound Journal](#) is a main source for different aspects of working with Csound.

Forums and Mailing Lists

The main forum to put questions concerning the usage of Csound is the [Csound Forum](#). It has categories and different possibilities how to put questions (or help with answers).

The traditional place for questions and answers is the Csound Mailing List. To subscribe to the [Csound User Discussion List](#), go to <https://listserv.heanet.ie/cgi-bin/wa?A0=CSOUND-LIST>

CSOUND. After subscribing, put questions at csound@listserv.heanet.ie. You can search in the list archive at nabble.com.

There is a **Csound slack** at <https://csound.slack.com>. Users can subscribe at <https://csound-slack.herokuapp.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://listserv.heanet.ie/cgi-bin/wa?A0=CSOUND-DEV>

Blue: <https://lists.sourceforge.net/mailman/listinfo/bluemusic-users>

Cabbage <http://forum.cabbageaudio.com/>

Bug Tracker

It should be distinguished whether an issue belongs to Csound or to one of the frontends.

Rule of thumb: If an issue has nothing to do with a graphical element or any frontend-specific feature, it belongs to Csound. Otherwise it belongs to CsoundQt, Cabbage or Blue.

Links for the issue trackers:

- [Core Csound](#)
- [CsoundQt](#)
- [Cabbage](#)
- [Blue](#)

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 (some years ago):

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=P50OyFyNaCA>

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=09WU7DzdUmE>

<http://www.youtube.com/watch?v=Hs3e07o349k>

<http://www.youtube.com/watch?v=yUMzp6556Kw>

New editing features in 0.6.0:

<http://www.youtube.com/watch?v=Hk1qPlnyv88>

New features in 0.7.0:

<https://www.youtube.com/watch?v=iytVlxMILyw>

New features in 0.9.2:

https://youtu.be/q_1SUT1wD6o

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

Csound and Inscore

<http://vimeo.com/54160283> (installation)

<http://vimeo.com/54160405> (examples)

german versions:

<http://vimeo.com/54159567> (installation)

<http://vimeo.com/54159964> (beispiele)

Csound Conferences

See the list at <https://csound.com/conferences.html>

Example Collections

Csound Realtime Examples by Iain McCurdy is certainly the most extended, approved and up-to-date collection.

The [Amsterdam Catalog](#) by John-Philipp Gathen is particularly interesting because of the adaption of Jean-Claude Risset's famous "Introductory Catalogue of Computer Synthesized Sounds" from 1969.

Books

Victor Lazzarini's [Computer Music Instruments](#) (2017) has a lot of Csound examples, in conjunction with Python and Faust.

[Csound – A Sound and Music Computing System](#) (2016) by Victor Lazzarini and others is the new Csound Standard Book, covering all parts of the Csound audio programming language in depth.

Martin Neukom's [Signals, Systems and Sound Synthesis](#) (2013) is a comprehensive computer music tutorial with a lot of Csound in it.

[Csound Power!](#) by Jim Aikin (2012) is a perfect up-to-date introduction for beginners.

[The Audio Programming Book](#) edited by Richard Boulanger and Victor Lazzarini (2011) is a major source with many references to Csound.

[Virtual Sound](#) by Alessandro Cipriani and Maurizio Giri (2000)

[The Csound Book](#) (2000) edited by Richard Boulanger is still the compendium for anyone who really wants to go in depth with Csound.

14 G. CREDITS

This textbook is a collective effort. If someone's name is missing in the list below, please contact us.

The goal of this work is to share knowledge about the Free (Libre) Open Source Software Csound for anyone. So any usage which contributes to this goal is welcome. The license below is to serve this goal. If you are not sure about a usage, please contact us, and we will find a solution.

LICENSE

copyright (c) 2011-2023 by Joachim Heintz and contributors

The content of this book is released under creative commons license **CC-BY 4.0**. In short, anyone is allowed to copy, redistribute and transform the content, as long as appropriate credit is given to the source. For more information see <https://creativecommons.org/licenses/by/4.0/>.

AUTHORS

00 INTRODUCTION

PREFACE Joachim Heintz, Andres Cabrera, Alex Hofmann, Iain McCurdy, Alexandre Abrioux

ON THIS RELEASE Joachim Heintz, Iain McCurdy

01 GETTING STARTED

Joachim Heintz

02 HOW TO

Joachim Heintz

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. *ARRAYS* Joachim Heintz
- F. *LIVE CSOUND* Joachim Heintz, Iain McCurdy
- G. *USER DEFINED OPCODES* Joachim Heintz
- H. *MACROS* Iain McCurdy
- I. *FUNCTIONAL SYNTAX* Joachim Heintz

04 SOUND SYNTHESIS

- A. *ADDITIVE SYNTHESIS* Andres Cabrera, Joachim Heintz, Bjorn Houdorf
- B. *SUBTRACTIVE SYNTHESIS* Iain McCurdy
- C. *AMPLITUDE AND RINGMODULATION* Alex Hofmann
- D. *FREQUENCY MODULATION* Alex Hofmann, Bjorn Houdorf, Marijana Janevska, Joachim Heintz
- E. *WAVESHAPING* Joachim Heintz, Iain McCurdy
- F. *GRANULAR SYNTHESIS* Iain McCurdy
- G. *PHYSICAL MODELLING* Joachim Heintz, Iain McCurdy, Martin Neukom
- H. *SCANNED SYNTHESIS* Christopher Saunders, Joachim Heintz

05 SOUND MODIFICATION

- A. *ENVELOPES* Iain McCurdy
- B. *PANNING AND SPATIALIZATION* Iain McCurdy, Joachim Heintz
- 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, Oeyvind Brandtsegg, Bjorn Houdorf, Joachim Heintz

H. CONVOLUTION Iain McCurdy

I. FOURIER ANALYSIS / SPECTRAL PROCESSING Joachim Heintz

K. ANALYSIS TRANSFORMATION SYNTHESIS Oscar Pablo di Liscia

L. AMPLITUDE AND PITCH TRACKING Iain McCurdy

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 OTHER COMMUNICATION

A. OPEN SOUND CONTROL Alex Hofmann, Joachim Heintz

B. CSOUND AND ARDUINO Iain McCurdy

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

- A. *CSOUNDQT* Andrés Cabrera, Joachim Heintz, Peiman Khosravi
- B. *CABBAGE* Rory Walsh, Menno Knevel, Iain McCurdy
- C. *BLUE* Steven Yi, Jan Jacob Hofmann
- D. *WINXOUND* Stefano Bonetti, Menno Knevel
- E. *CSOUND VIA TERMINAL* Iain McCurdy
- F. *WEB BASED CSOUND* Victor Lazzarini, Iain McCurdy, Ed Costello

11 CSOUND UTILITIES

- A. *ANALYSIS* Iain McCurdy
- B. *FILE INFO AND CONVERSION* Iain McCurdy
- C. *MISCELLANEOUS* Iain McCurdy

12 CSOUND AND OTHER PROGRAMMING LANGUAGES

- A. *THE CSOUND API* François Pinot, Rory Walsh
- B. *PYTHON AND CSOUND* Andrés Cabrera, Joachim Heintz, Jim Aikin
- C. *LUA AND CSOUND* Michael Gogins, Philipp Henkel
- D. *CSOUND IN IOS* Nicholas Arner, Nikhil Singh, Richard Boulanger, Alessandro Petrolati
- E. *CSOUND ON ANDROID* Michael Gogins
- F. *CSOUND AND HASKELL* Anton Kholomiov
- G. *CSOUND AND HTML* Michael Gogins

13 EXTENDING CSOUND

- A. *DEVELOPING PLUGIN OPCODES* Victor Lazzarini

14 MISCELLANEA

- A. *OPCODE GUIDE* Joachim Heintz, Iain McCurdy

- B. METHODS OF WRITING CSOUND SCORES Iain McCurdy, Joachim Heintz, Jacob Joaquin, Menno Knevel
- C. AMPLITUDE AND PITCH TRACKING Iain McCurdy
- D. PYTHON IN CSOUNDQT Tarmo Johannes, Joachim Heintz
- E. GLOSSARY Joachim Heintz, Iain McCurdy
- F. LINKS Joachim Heintz, Stefano Bonetti

15 DIGITAL AUDIO 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
- D. RANDOM Joachim Heintz, Martin Neukom, Iain McCurdy

EDITING TEAMS FOR VERSIONS

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

V.3 - Final Editing Team in March 2013:

Joachim Heintz, Iain McCurdy

V.4 - Final Editing Team in September 2013:

Joachim Heintz, Alexandre Abrioux

V.5 - Final Editing Team in March 2014:

Joachim Heintz, Iain McCurdy

V.6 - Final Editing Team March-June 2015:

Joachim Heintz, Iain McCurdy

V.7 - Final Editing and Programming Team 2019/20:

Joachim Heintz, Hlöðver Sigurðsson

V.8 - Final Editing and Programming Team 2023:

Joachim Heintz, Hlöðver Sigurðsson

15 A. 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 combination of several of these. This disturbance in the medium causes molecules to move back and forth 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.

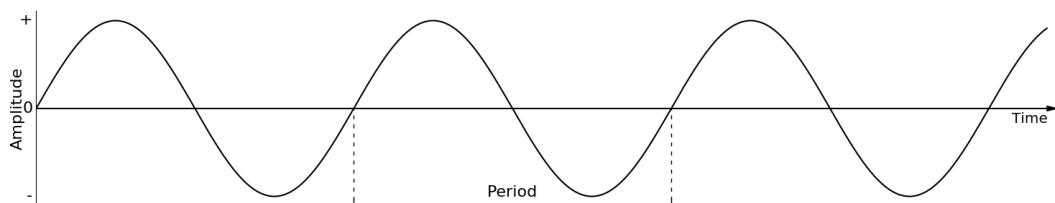


Figure 82.1: Sine wave

Each time the waveform signal goes above zero the molecules are in a state of compression meaning that each molecule within the waveform disturbance is pushing into its neighbour. Each time the waveform signal drops below zero the molecules are in a state of rarefaction meaning the molecules are pulling away from their neighbours. 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 some main parameters:

- **Period:** The time it takes for a waveform to complete one cycle, measured in seconds.
- **Frequency:** The number of cycles or periods per second, measured in Hertz (Hz). If a sound has a frequency of 440 Hz it completes 440 cycles every second. Read more about frequency in the [next chapter](#).
- **Phase:** This is the starting point of a waveform. It can be expressed in degrees or in radians. A complete cycle of a waveform will cover 360 degrees or 2π radians. A sine with a phase of 90° or $\pi/2$ results in a cosine.

- **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, which will also depend upon the resistance offered by the medium, will determine how far above and below zero - the point of equilibrium - 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 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 as possible: whatever goes in should come out as a perfect analogy in a voltage representation. In reality, however, this will not be the case. Low quality devices add noise and deformation. High quality devices add certain characteristics like warmth or transparency.

Sampling

The analogue voltage that corresponds to an acoustic signal changes continuously, so that at each point 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 this 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 moment of sampling.

Below is a diagram showing a sinusoidal waveform. The vertical lines that run through the diagram represent 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 bottom half of the diagram.

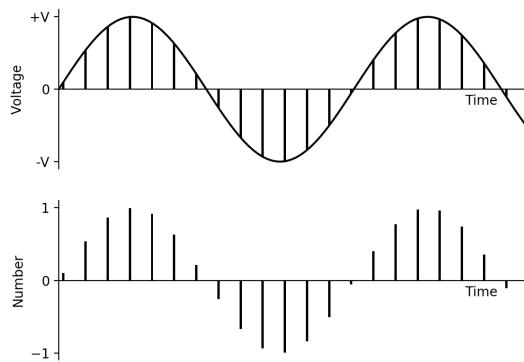


Figure 82.2: Sampling of an analog signal

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 at which the sample or snapshot was taken.

The same principle applies to recording of live video: a video camera takes a sequence of pictures of motion and most video cameras will take between 30 and 60 still pictures a second. Each picture is called a frame and when these frames are played in sequence at a rate corresponding to that at which they were taken we no longer perceive them as individual pictures, we perceive them instead as a continuous moving image.

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 and will actually produce 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.

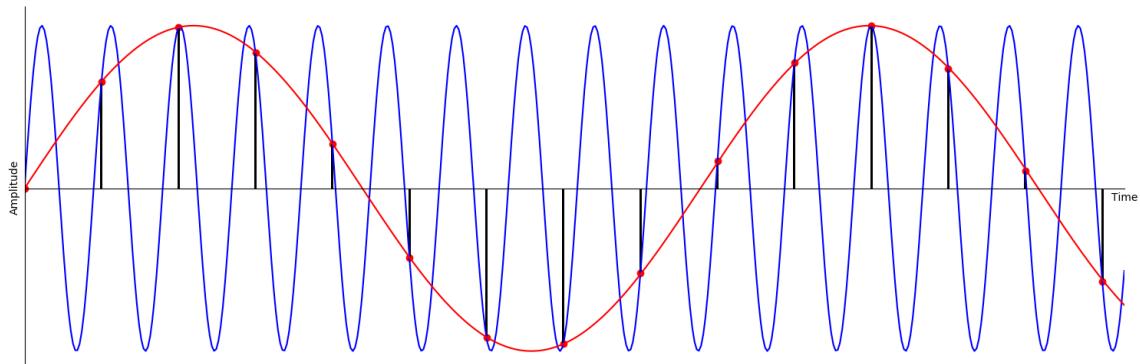


Figure 82.3: Aliasing (red) of a high frequency (blue)

The sinusoidal waveform in blue is being sampled at the vertical black lines. The line that joins the red circles together is the captured waveform. As you can see, the captured waveform and the original waveform express different frequencies.

Here is another example, showing for a sample rate of 40 kHz in the upper section a sine of 10 kHz, and in the lower section a sine of 30 kHz:

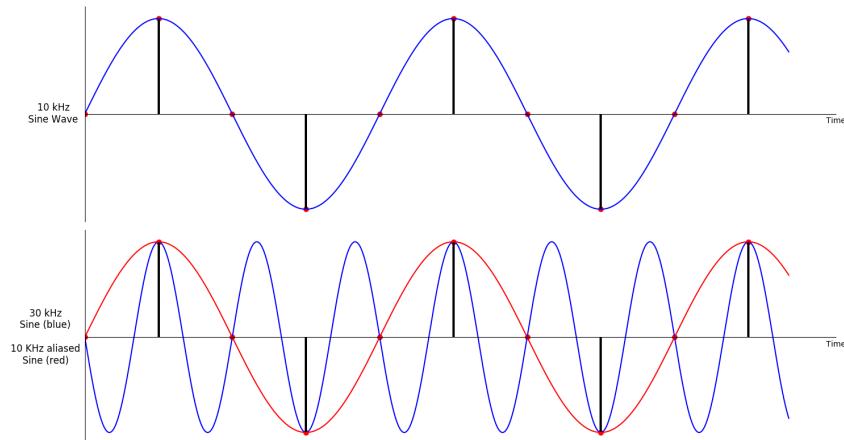


Figure 82.4: Aliasing of a 30 kHz sine at 40 kHz sample rate

We can see that if the sample rate is 40 kHz there is no problem with sampling a signal that is 10 kHz. On the other hand, in the second example it can be seen that a 30 kHz waveform is not going to be correctly sampled. In fact we end up with a waveform that is 10 kHz, rather than 30 kHz. This may seem like an academic proposition in that we will never be able to hear a 30 kHz waveform anyway but some synthesis and DSP techniques procedures will produce these frequencies as unavoidable by-products and we need to ensure that they do not result in unwanted artifacts.

In computer music we can produce any frequency internally, much higher than we can hear, and much higher than the Nyquist frequency. This may occur intentionally, or by accident, for instance when we multiply a frequency of 2000 Hz by the 22nd harmonic, resulting in 44000 Hz. In the following example, instrument 1 plays a 1000 Hz tone first directly, and then as a result of 43100 Hz input which is 1000 Hz lower than the sample rate of 44100 Hz. Instrument 2 demonstrates unwanted aliasing as a result of harmonics beyond Nyquist: the 22nd partial of 1990 Hz is 43780 Hz which sounds as $44100 - 43780 = 320$ Hz.

EXAMPLE 15A01_Aliasing.cs

```

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

//wave form with harmonics 1, 11 and 22
giHarmonics ftgen 0, 0, 8192, 9, 1,.1,0, 11,.1,0, 22,1,0

instr 1
    asig poscil .1, p4
    out asig, asig
endin

instr 2
    asig poscil .2, p4, giHarmonics
    out asig, asig
endin

</CsInstruments>
<CsScore>
i 1 0 2 1000 ;1000 Hz sine
i 1 3 2 43100 ;43100 Hz sine sounds like 1000 Hz because of aliasing
i 2 6 4 1990 ;1990 Hz with harmonics 1, 11 and 22
                ;results in 1990*22=43780 Hz so aliased 320 Hz
                ;for the highest harmonic
</CsScore>
</CsSoundSynthesizer>
;example by joachim heintz

```

The same phenomenon happens in film and video, too. You may recall having seen wagon wheels apparently turn in the wrong direction in old Westerns. Let us say for example that a camera is taking 30 frames per second of a wheel moving. In one example, if the wheel is completing one rotation in exactly 1/30th of a second, then every picture looks the same and as a result the wheel appears to be motionless. If the wheel speeds up, i.e. it increases its rotational 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.

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. Csound is perfectly capable of imitating the effects of aliasing while being run at any sample rate - if that is what you desire.

Audio-CD Quality uses a sample rate of 44100 Hz (44.1 kHz). This means that CD quality can only represent frequencies up to 22050 Hz. Humans typically have an absolute upper limit of hearing of about 20 KHz thus making 44.1 KHz a reasonable standard sampling rate. Higher sample rates offer better time resolution and the Nyquist frequency is not that close to the limit of hearing. But on the other hand twice the sample rate creates twice as much data. The choice has to be made depending on the situation; in this book we stick on the sample rate of 44100 Hz for the examples.

Bits, Bytes and Words

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 in one of two states: off or on, 0 or 1. 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 historically represented a single character of data in the computer memory. Mostly one byte is the smallest unit of data, and bigger units will be created by using two, three or more bytes. A good example is the number of bytes which is used to store the number for one audio sample. In early games it was 1 byte (8 bit), on a CD it is 2 bytes (16 bit), in sound cards it is often 3 bytes (24 bit), in most audio software it is internally 4 bytes (32 bit), and in Csound 8 bytes (64 bit).

The [word length](#) of a computer is the number of bits which is handled as a unit by the processor. The transition from 32-bit to 64-bit word length around 2010 in the most commonly used processors required new compilations of Csound and other applications, in particular for the Windows installers. To put it simple: A 32-bit machine needs an application compiled for 32-bit, a 64-bit machine needs an application compiled for 64-bit.

Bit-depth Resolution

The sample rate determines the finer or rougher resolution in time. The number of bits for each single sample determines the finer or rougher resolution in amplitude. The standard resolution for CDs is 16 bit, which allows for 65536 different possible amplitude levels, 32767 on 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.

The figure below shows the quantization issue in simplified version, assuming a depth of only 3 bit. This is like a grid of $2^3 = 8$ possible levels which can be used for each sample. At each sampling period the soundcard plots an amplitude which is adjusted to the next possible vertical position. For a signal with lower amplitude the distortion would even be stronger.

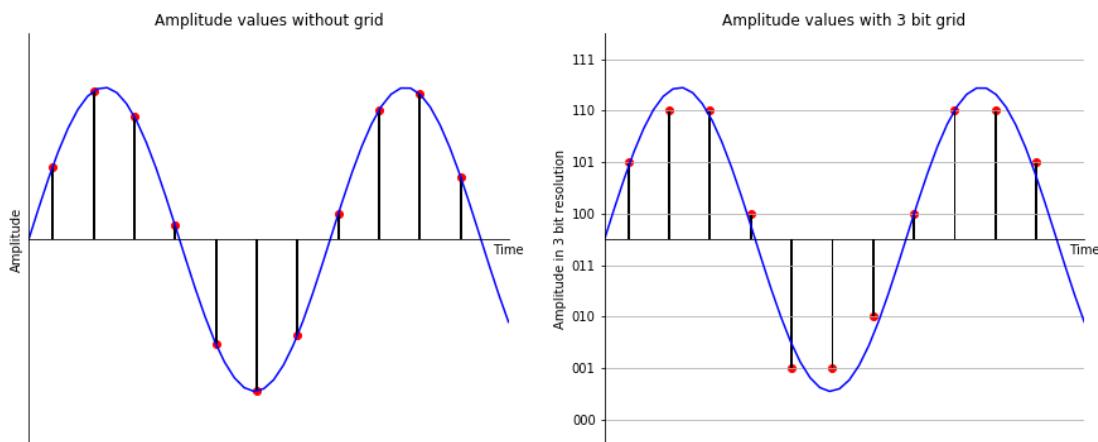


Figure 82.5: Inaccurate amplitude values due to insufficient bit depth resolution

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 easy to implement in Csound. Example 05F02 in chapter 05F shows one possibility.

ADC / DAC

The entire process, as described above, of taking an analogue signal and converting it to 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. If a sound is played back or streamed, the software will send a series of numbers to the soundcard. The soundcard converts these numbers back to voltage. When the voltages reaches the loudspeaker they cause the loudspeaker's membrane to move inwards and outwards. This induces a disturbance in the air around the speaker – compressions and rarefactions as described at the beginning of this chapter – resulting in what we perceive as sound.

15 B. PITCH AND FREQUENCY

Pitch and frequency are related but different terms.¹ *Pitch* is used by musicians to describe the “height” of a tone, most obvious on a keyboard. *Frequency* is a technical term. We will start with the latter and then return to pitch in some of its numerous aspects, including intervals, tuning systems and different conversions between pitch and frequency in Csound.

Frequencies

As mentioned in the previous chapter, *frequency* is defined as the *number of cycles or periods per second*. The SI unit is *Hertz* where 1 Hertz means 1 period per second. If a tone has a frequency of 100 Hz it completes 100 cycles every second. If a tone has a frequency of 200 Hz it completes 200 cycles every second.

Given a tone’s frequency, the time for one period can be calculated straightforwardly. For 100 periods per seconds (100 Hz), the time for one period is 1/100 or 0.01 seconds. For 200 periods per second (200 Hz), the time for each period is only half as much: 1/200 or 0.005 seconds. Mathematically, the period is the reciprocal of the frequency and vice versa. In equation form, this is expressed as follows:

$$\text{Frequency} = \frac{1}{\text{Period}}$$

$$\text{Period} = \frac{1}{\text{Frequency}}$$

Wavelength

In physical reality, one cycle of a periodic sound can not only be measured in time, but also as extension in space. This is called the wavelength. It is usually abbreviated with the greek letter λ (lambda). It can be calculated as the ratio between the velocity and the frequency of the wave.

$$\lambda = \frac{\text{Velocity}}{\text{Frequency}}$$

As the velocity of a sound in air (at 20° Celsius) is about 340 m/s, we can calculate the wavelength of a sound as

¹Similar to *volume* and *amplitude* – see [next chapter](#).

$$\lambda = \frac{\frac{340m}{s}}{\frac{s}{Number\ of\ Cycles}} = \frac{340}{Number\ of\ Cycles}m$$

For instance, a sine wave of 1000 Hz has a length of approximately $340/1000\text{ m} = 34\text{ cm}$, whereas a wave of 100 Hz has a length of $340/100\text{ m} = 3.4\text{ m}$.

Periodic and Nonperiodic Sounds

Not all sounds are periodic. In fact, periodic sounds are only one end of a range. The other end is noise. In between is a continuum which can be described from both points of view: a periodic sound which has noisy parts, or a noise which has periodic parts. The following example shows these aspects in one of their numerous possibilities. It starts with a sine tone of 1000 Hz and slowly adds aperiodicity. This is done by changing the frequency of the sine oscillator faster and faster, and in a wider and wider range. At the end noise is reached. The other way, from noise to a periodic tone, is shown with a band filter. Its band width is at first 10000 Hz around a center frequency of 1000 Hz, i.e. essentially not altering the white noise. Then the band width decreases dramatically (from 10000 Hz to 0.1 Hz) so that at the end a sine tone is nearly reached.

EXAMPLE 15B01_PeriodicAperiodic.csd

```

<CsoundSynthesizer>
<CsOptions>
-o dac
</CsOptions>
<CsInstruments>

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

instr SineToNoise
kMinFreq = expseg:k(1000, p3*1/5, 1000, p3*3/5, 20, p3*1/5, 20)
kMaxFreq = expseg:k(1000, p3*1/5, 1000, p3*3/5, 20000, p3*1/5, 20000)
kRndFreq = expseg:k(1, p3*1/5, 1, p3*3/5, 10000, p3*1/5, 10000)
aFreq = randomi:a(kMinFreq, kMaxFreq, kRndFreq)
aSine = oscil:a(.1, aFreq)
aOut = linen:a(aSine, .5, p3, 1)
out(aOut, aOut)
endin

instr NoiseToSine
aNoise = rand:a(.1, 2, 1)
kBw = expseg:k(10000, p3*1/5, 10000, p3*3/5, .1, p3*1/5, .1)
aFilt = reson:a(aNoise, 1000, kBw, 2)
aOut = linen:a(aFilt, .5, p3, 1)
out(aOut, aOut)
endin

</CsInstruments>
<CsScore>
i "SineToNoise" 0 10
i "NoiseToSine" 11 10
</CsScore>
</CsoundSynthesizer>
;example by joachim heintz

```

This is what the signal looks like at the start and the end of the *SineToNoise* process:

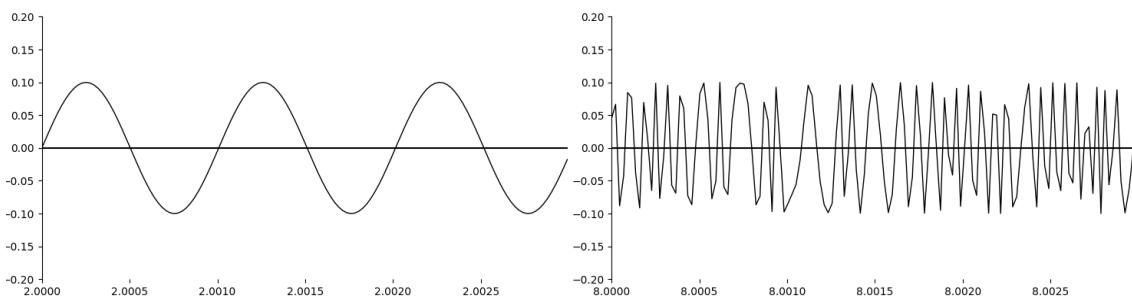


Figure 83.1: Sine to noise

And this is what the signal looks like at the start and the end of the *NoiseToSine* process:

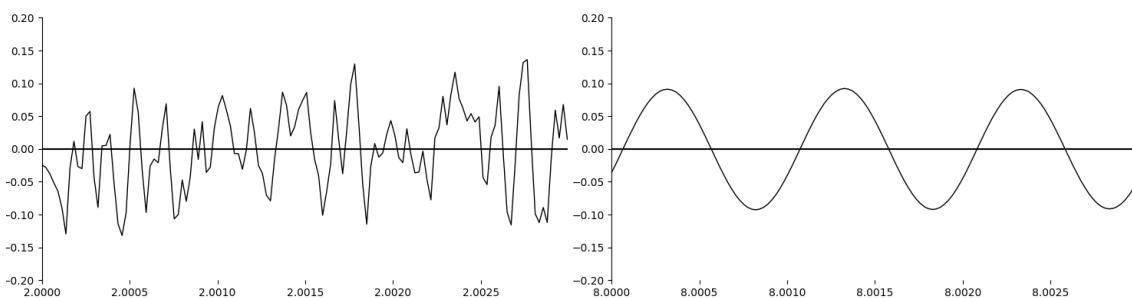


Figure 83.2: Noise to sine

Only when a sound is periodic, we perceive a pitch. But the human ear is very sensitive, and it is quite fascinating to observe how little periodicity is needed to sense some pitch.

Upper and Lower Limits of Hearing

It is generally stated that the human ear can hear sounds in the range 20 Hz to 20,000 Hz (20kHz). This upper limit tends to decrease with age due to a condition known as presbyacusis, or age related hearing loss. Most adults can hear frequencies up 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, and very low frequencies need more power to be heard than medium or high frequencies. (This is explained more in detail in the paragraph about the *Fletcher-Munson-Curves* in the [next chapter](#).)

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 15B02_LimitsOfHearing.csd

```
<CsoundSynthesizer>
<CsOptions>
-odac -m0
</CsOptions>
```

```

<CsInstruments>
sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1

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

</CsInstruments>
<CsScore>
i 1 0 2 10
i . + . 100
i . + . 1000
i . + . 10000
i . + . 20000
</CsScore>
</CsoundSynthesizer>
;example by joachim heintz

```

Pitches

Musicians tune their instruments, and theorists concern themselves with the rationale, describing intervals and scales. This has happened in different cultures, for ages, long before the term frequency was invented and long before it was possible to measure a certain frequency by technical devices. What is the relationship between musical terms like octave, major third, semitone and the frequency we have to specify for an oscillator? And why are frequencies often described as being on a “logarithmic scale”?

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 for repeated additions, for example, $5 \times 10 = 5+5+5+5+5$. Exponents are shorthand for repeated multiplications, $3^5 = 3 \times 3 \times 3 \times 3 \times 3$. 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.

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 seven semitones, so seven adjacent keys on a keyboard. When dealing with Hz values things are different. A difference of say 100 Hz does not always equate to the same musical interval. This is because musical intervals are represented as *ratios* between two frequencies. An octave for example is always defined by the ratio 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 A4 at 440 Hz. If the player plays A5 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 A6 with a frequency of 1760 Hz but it can also play A7 an octave above this at 3520 Hz (2×1760 Hz). While the difference in Hertz between A4 and A5 on the flute is only 440 Hz, the difference between A6 and A7 on a piccolo is 1760 Hz yet they are both only playing notes one octave apart.

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 spacing sounds constant, although the frequency displacement is different each time.

EXAMPLE 15B03 Adding_vs_ratio.csd

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

instr 1
prints "Playing %d Hertz!\n", p4
asig oscil .2, p4
aout linen asig, 0, p3, p3
outs aout, aout
endin

instr 2
prints "Adding %d Hertz to %d Hertz!\n", p5, p4
asig oscil .2, p4+p5
aout linen asig, 0, p3, p3
outs aout, aout
endin

instr 3
prints "Applying the ratio of %f (adding %d Hertz) to %d Hertz!\n",
p5, p4*p5, p4
asig oscil .2, p4*p5
aout linen asig, 0, p3, p3
outs aout, aout
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>
;example by joachim heintz
```

Equal tempered scale

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, called semitones. Therefore a semitone has a ratio of $2^{1/12}$, which is approximately 1.059463.² The next semitone will have the ratio $2^{2/12}$ (1.122462...), the third one $2^{3/12}$ (1.189207...), and so on. The exponents increase linear (1/12, 2/12, 3/12, ...), thus yielding the same proportion between each subsequent semitone.

So what about the reference to logarithms? As stated previously, logarithms are shorthand for exponents. $2^{1/12} = 1.059463$ can also be written as $\log_2(1.059463) = 1/12$. Therefore, frequencies representing musical scales or intervals can be described on a logarithmic scale. The linear progression of the exponents (with base 2) as 1/12, 2/12, 3/12 ... represent the linear progression of semitones.

MIDI Notes

The equal-tempered scale is present on each MIDI keyboard. So the most common way to work with pitches is to use MIDI note numbers. In MIDI speak A4 (= 440 Hz) is MIDI note 69.³ The semitone below, called A flat or G sharp, is MIDI note 68, and so on. The MIDI notes 1-127 cover the frequency range from 9 Hz to 12544 Hz which is pretty well suited to the human hearing (and to a usual grand piano which would correspond to MIDI keys 21-108).

Csound can easily deal with MIDI notes and comes with functions that will convert MIDI notes to Hertz values (*mtof*) and back again (*fton*). The next example shows a small chromatic melody which is given as MIDI notes in the array *iMidiKeys[]*, and then converted to the corresponding frequencies, related to the definition of A4 (440 Hz as default). The opcode *mton* returns the note names.

EXAMPLE 15B04_Midi_to_frequency.csd

```
<CsoundSynthesizer>
<CsOptions>
-o dac -m128
</CsOptions>
<CsInstruments>
```

² $2^{1/12}$ is the same as $\sqrt[12]{2}$ thus the number which yields 2 if multiplied by itself 12 times.

³Caution: like many standards there is occasional disagreement about the mapping between frequency and octave number. You may occasionally encounter A 440 Hz being described as A3.

```

sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1
A4 = 457

instr LetPlay

iMidiKeys[] fillarray 69, 69, 69, 68, 67, 66, 65, 64
iDurations[] fillarray 2, 1, 1, 1, 1, 1, 1, 4
iIndex = 0
iStart = 0
while iIndex < lenarray(iMidiKeys) do
    schedule "Play", iStart, iDurations[iIndex]*3/2, iMidiKeys[iIndex]
    iStart += iDurations[iIndex]
    iIndex += 1
od

endin

instr Play

iMidiKey = p4
iFreq mtos iMidiKey
S_name mton iMidiKey
printf_i "Midi Note = %d, Frequency = %f, Note name = %s\n",
    1, iMidiKey, iFreq, S_name
aPluck pluck .2, iFreq, iFreq, 0, 1
aOut linen aPluck, 0, p3, p3/2
aL, aR pan2 aOut, (iMidiKey-61)/10
out aL, aR

endin

</CsInstruments>
<CsScore>
i "LetPlay" 0 1
</CsScore>
</CsoundSynthesizer>
;example by joachim heintz

```

As A4 is set in the header to 457 Hz (overwriting the default 440 Hz), this is the printout:

```

Midi Note = 69, Frequency = 457.000000, Note name = 4A
Midi Note = 69, Frequency = 457.000000, Note name = 4A
Midi Note = 69, Frequency = 457.000000, Note name = 4A
Midi Note = 68, Frequency = 431.350561, Note name = 4G#
Midi Note = 67, Frequency = 407.140714, Note name = 4G
Midi Note = 66, Frequency = 384.289662, Note name = 4F#
Midi Note = 65, Frequency = 362.721140, Note name = 4F
Midi Note = 64, Frequency = 342.363167, Note name = 4E

```

Other Pitch Representation

In addition to raw frequency input and MIDI note numbers, Csound offers two more possibilities to specify a certain pitch. The *pch* notation is a floating point number, in which the integer part denotes the octave number and the fractional part denotes the semitones. The octave numbers are not the same as in the common system – the middle octave is number 8 rather than 4. So C4,

the “middle c” on a piano, has the number 8.00. Semitones upwards are then 8.01, 8.02 and so on, reaching A4 as 8.09. B4 is 8.11 and C5 is 9.00.

The *oct* notation also uses floating point numbers. The integer part has the same meaning as in the *pch* notation. The fractional part divides one octave in acoustically equal steps. For 8.00 as C4 and 9.00 as C5, 8.5 denotes a pitch which is acoustically in the middle between C4 and C5, which means that the proportion between this frequency and the C4 frequency is the same as the proportion between the C5 frequency and this tone’s frequency. Csound calculates this as:

```
instr 1
    iC4 = cpsoct(8)
    iC5 = cpsoct(9)
    iNew = cpsoct(8.5)
    prints "C4 = %.3f Hz, C5 = %.3f Hz, oct(8.5) = %.3f Hz.\n",
           iC4, iC5, iNew
    prints "Proportion New:C4 = %.3f, C5:New = %.3f\n",
           iNew/iC4, iC5/iNew
    endin
    schedule(1,0,0)
```

And the output is:

```
C4 = 261.626 Hz, C5 = 523.251 Hz, oct(8.5) = 369.994 Hz.
Proportion New:C4 = 1.414, C5:New = 1.414
```

On a keyboard, this pitch which divides the octave in two acoustically equal halves, is F#4. It can be notated in *pch* notation as 8.06, or in MIDI notation as key number 66. So why was *oct* notation added? – The reason is that by this notation it becomes very simple to introduce for instance the division of an octave into 10 equal steps: 8.1, 8.2, ..., or in 8 equal steps as 8.125, 8.25, 8.375, ...

The following code shows that things like these can also be achieved with a bit of math, but for simple cases it is quite convenient to use the *oct* notation. A scale consisting of ten equal steps based on A3 (= 220 Hz) is constructed.

```
instr 1
    puts "Calculation with octpch():", 1
    iOctDiff = 0
    while iOctDiff < 1 do
        prints "oct(%2f)=%3f ", 7.75+iOctDiff, cpsoct(7.75+iOctDiff)
        iOctDiff += 1/10
    od
    puts "",1
    puts "Calculation with math:", 1
    iExp = 0
    while iExp < 1 do
        prints "pow(2,%1f)=%3f ", pow(2,iExp), pow(2,iExp) * 220
        iExp += 1/10
    od
    puts "",1
    endin
    schedule(1,0,0)
```

Cent

One semitone in the equal-tempered tuning system can be divided into 100 Cent. It is a common way to denote small or “microtonal” deviations. It can be used in Csound’s MIDI notation as frac-

tional part. MIDI note number 69.5 is a quarter tone (50 Cent) above A4; 68.75 is an eighth tone (25 Cent) below A4. In the *pch* notation we would write 8.095 for the first and 8.0875 for the second pitch.

All musical intervals can be described as ratios or multipliers. The ratio for the perfect fifth is 3:2, or 1.5 when used as multiplier. Also one Cent is a multiplier. As one octave consists of 12 semitones, and each semitone consists of 100 Cent, one octave consists of 1200 Cent. So one Cent, described as multiplier, is $2^{1/1200}$ (1.000577...), and 50 Cent is $2^{50/1200}$ (1.0293022...). To return this multiplier, Csound offers the *cent* converter. So *cent(50)* returns the number by which we must multiply a certain frequency to get a quarter tone higher, and *cent(-25)* returns the multiplier for calculating an eighth tone lower.

```
instr 1
    prints "A quarter tone above A4 (440 Hz):\n"
    prints " 1. as mtot:i(69.5) = %f\n", mtot:i(69.5)
    prints " 2. as cpspch(8.095) = %f\n", cpspch(8.095)
    prints " 3. as 2^(50/1200)*440 = %f\n", 2^(50/1200)*440
    prints " 4. as cent(50)*440 = %f\n", cent(50)*440
    endin
    schedule(1,0,0)
```

The result of this comparison is:

```
A quarter tone above A4 (440 Hz):
 1. as mtot:i(69.5) = 452.892984
 2. as cpspch(8.095) = 452.880211
 3. as 2^(50/1200)*440 = 452.892984
 4. as cent(50)*440 = 452.892984
```

Tuning Systems

The equal-tempered tuning system which can be found on each MIDI keyboard is not the only tuning system in existence. For many musical contexts it is not appropriate. In European history there were many different systems, for instance the Pythagorean and the Meantone tuning. Each of the countless traditional music cultures all over the world, for instance Arabic Maqam, Iranian Dastgah, Indian Raga, has its own tuning system. And in contemporary music we find also numerous different tuning systems.

Audio programming languages like Csound, which can synthesize sounds with any frequency, are particularly suited for this approach. It is even simple to “tune” a MIDI keyboard in quarter tones or to any historical tuning using Csound. The following example shows the fundamentals. It plays the five notes C D E F G (= MIDI 60 62 64 65 67) first in Pythagorean tuning, then in Meantone, then as quarter tones, then as partials 1-5.

EXAMPLE 15B05_Tuning_Systems.csd

```
<CsoundSynthesizer>
<CsOptions>
-o dac -m128
</CsOptions>
<CsInstruments>
```

```

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

instr Pythagorean
giScale[] fillarray 1, 9/8, 81/64, 4/3, 3/2
schedule("LetPlay",0,0)
puts "Pythagorean scale",1
endin

instr Meantone
giScale[] fillarray 1, 10/9, 5/4, 4/3, 3/2
schedule("LetPlay",0,0)
puts "Meantone scale",1
endin

instr Quartertone
giScale[] fillarray 1, 2^(1/24), 2^(2/24), 2^(3/24), 2^(4/24)
schedule("LetPlay",0,0)
puts "Quartertone scale",1
endin

instr Partials
giScale[] fillarray 1, 2, 3, 4, 5
schedule("LetPlay",0,0)
puts "Partials scale",1
endin

instr LetPlay
indx = 0
while indx < 5 do
  schedule("Play",indx,2,giScale[indx])
  indx += 1
od
endin

instr Play
iFreq = mttoi(60) * p4
print iFreq
aSnd vco2 .2, iFreq, 8
aOut linen aSnd, .1, p3, p3/2
out aOut, aOut
endin

</CsInstruments>
<CsScore>
i "Pythagorean" 0 10
i "Meantone" 10 10
i "Quartertone" 20 10
i "Partials" 30 10
</CsScore>
</CsoundSynthesizer>
;example by joachim heintz

```

Frequently Used Formulas

New Frequency from Frequency and Proportion

Given:

- Frequency f
- Proportion p

Searched:

- New Frequency f_{new}

Solution: $f_{new} = f \cdot p$

Example: Which frequency is in 5/4 proportion to 440 Hz? $\rightarrow f_{new} = 440 \text{ Hz} \cdot 5/4 = 550 \text{ Hz}$

Csound code: `iFreq_new = 440 * 5/4`

New Frequency from Frequency and Cent Difference

Given:

- Frequency f
- Cent difference c

Searched:

- New Frequency f_{new}

Solution: $f_{new} = f \cdot 2^{c/1200}$

Example: Which frequency is 50 Cent below 440 Hz? $f_{new} = 440 \cdot 2^{-50/1200} = 427.474 \text{ Hz}$

Csound code: `iFreq_new = 440 * 2^(-50/1200)`

Cent Difference of two Frequencies

Given:

- Frequency_1 f_1
- Frequency_2 f_2

Searched:

- Cent difference c

Solution: $c = \log_2 \frac{f_1}{f_2} \cdot 1200$

Example: What is the Cent difference between 550 Hz and 440 Hz? $\rightarrow c = \log_2 \frac{550}{440} \cdot 1200 = 386.314 \text{ Cent}$

Csound code: `iCent = log2(550/440) * 1200`

15 C. INTENSITIES

As musicians we are dealing with volume, loudness, sound intensity. (In classical western music called *dynamics*, designated as *forte*, *piano* and its variants.) In digital audio, however, we are dealing with *amplitudes*. We are asked, for instance, to set the amplitude of an oscillator. Or we see this message at the end of a Csound performance in the console telling us the “overall amps” (= amplitudes):

```
SECTION 1:  
Score finished in csoundPerformKsmmps() with 2.  
inactive allocs returned to freespace  
end of score.          overall amps: 0.49927 0.49927  
                      overall samples out of range:      0      0  
0 errors in performance
```

Figure 84.1: Csound console printout

Amplitudes are related to sound intensities, but in a more complicated way than we may think. This chapter starts with some essentials about measuring intensities and the *decibel* (*dB*) scale. It continues with *rms* measurement and ends with the *Fletcher-Munson* curves.

Real World Intensities and Amplitudes

SIL – Sound Intensity Level

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 Watts 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 I_0 as follows:

$$\log_{10} \frac{I}{I_0}$$

Sound Intensity Level in Bel

If, for example, the ratio 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 human hearing covers about 12 Bel. Usually 1 Bel is divided into 10 decibel, so the common formula for measuring a sound intensity is:

$$10 \cdot \log_{10} \frac{I}{I_0}$$

Sound Intensity Level (SIL) in deci Bel (dB) with $I_0 = 10^{-12} \text{ W/m}^2$

SPL – Sound Pressure Level

While the sound intensity level is useful in describing the way in which 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. But 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:

$$I \propto P^2$$

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 $2 \cdot 10^{-5} \text{ Pa}$. This value is the reference value of the Sound Pressure Level (SPL). If we now have a value of $2 \cdot 10^{-4} \text{ Pa}$, the corresponding sound intensity relationship can be calculated as

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

Therefore a factor of 10 in a pressure relationship yields a factor of 100 in the intensity relationship. In general, the dB scale for the pressure P related to the pressure P_0 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 Decibels (dB) with $P_0 = 2 \cdot 10^{-5} \text{ Pa}$

Sound Intensity and Amplitudes

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

$$I \propto A^2$$

Relationship between Intensity and Amplitude

This yields the same transformation as described above for the sound pressure; so finally the relation in Decibel of any amplitude A to a reference amplitude A_0 is:

$$20 \cdot \log_{10} \frac{A}{A_0}$$

Decibel (dB) Scale of Amplitudes

If we drive an oscillator with an amplitude of 1, and another oscillator with an amplitude of 0.5 and we want to know the difference in dB, this is the calculation:

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

The most useful thing to bear in mind is that when we double an amplitude this will provide a change of +6 dB, or when we halve an amplitude this will provide a change of -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 $2 \cdot 10^{-5} \text{ Pa}$.

When working with digital sound within a computer, this method for defining 0 dB will not make any sense. The loudness of the sound produced in the computer will ultimately depend on the amplification and the speakers, and the amplitude level set in your audio editor or in Csound will only apply an additional, and not an absolute, sound level control. 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 normally used as the reference point for 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 for 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 use the amplitude convention used by most modern audio programming environments.

dB Scale Versus Linear Amplitude

Now we will consider some practical consequences of what we have discussed so far. One major point is that for achieving perceptually smooth changes across 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 15C01_db_vs_linear.csd

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

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

instr 2 ;linear rise of dB
kdb     line    -80, p3, 0   ;dB rise -80 -> 0
asig    oscil   1, 1000      ;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>
;example by joachim heintz
```

The first note, which employs a linear rise in amplitude, is perceived as rising quickly in intensity

with the rate of increase slowing quickly. The second note, which employs a linear rise in decibels, is perceived as a more constant rise in 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 N samples. Then you (2) divide the result by N to calculate the mean of it. Finally (3) take the square root.

Let us consider a simple example and then look at how to derive rms values within Csound. Assuming we have a sine wave which consists of 16 samples, we get these amplitudes:

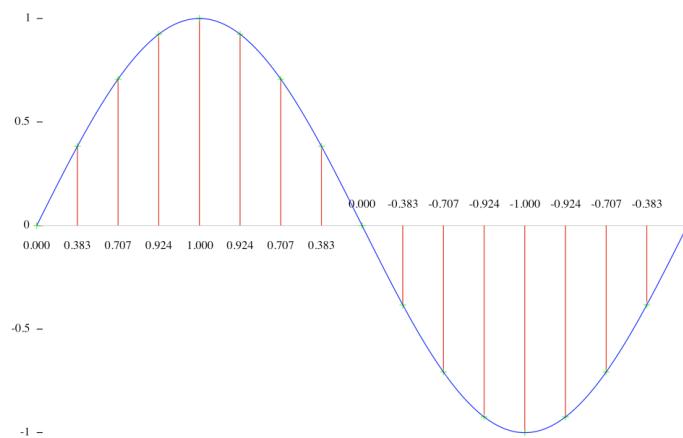


Figure 84.2: 16 times sampled Sine Wave

These are the squared amplitudes:

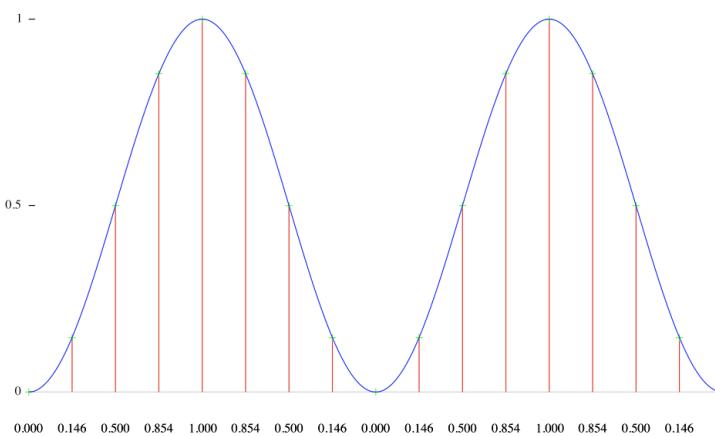


Figure 84.3: Squared Amplitudes of Sine

The mean of these values is:

$$\frac{0+0.146+0.5+0.854+1+0.854+0.5+0.146+0+0.146+0.5+0.854+1+0.854+0.5+0.146}{16} = \frac{8}{16} = 0.5$$

And the resulting RMS value is $\sqrt{0.5} = 0.707$.

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 is (the default is 10 Hz), the quicker this measurement will respond to changes, 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 example¹:

EXAMPLE 15C02_rms_feedback_system.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

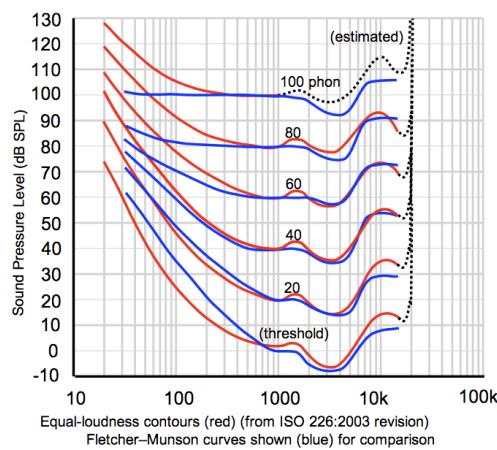
instr 1
a3      init      0
kamp    linseg   0, 1.5, 0.2, 1.5, 0      ;envelope for initial input
asnd    oscil    kamp, 440, giSine        ;initial input
if p4 == 1 then
  adel1  oscil    0.0523, 0.023, giSine
  adel2  oscil    0.073, 0.023, giSine,.5
else
  adel1  randi    0.05, 0.1, 2
  adel2  randi    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
       delayw  asnd + exp(-krms) * a3
a3      reson    -(a1+a2), 3000, 7000, 2
aout   linen    a1/3, 1, p3, 1          ;calculate a3
       outs     aout, aout             ;apply fade in and fade out
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>
;example by Martin Neukom, adapted by Joachim Heintz
```

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

Fletcher-Munson Curves

The range of human hearing is roughly from 20 to 20000 Hz, but within this range, the hearing is not equally sensitive to intensity. The most sensitive region is around 3000 Hz. If a sound is operating in the upper or lower limits of this range, it will need greater intensity in order to be perceived 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:



Try the following test. During the first 5 seconds you will hear a tone of 3000 Hz. Adjust the level of your amplifier to the lowest possible level at which you still can hear the tone. Next 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. Of course, this effectiveness of this test will also depend upon the quality of your speaker hardware. If your speakers do not provide adequate low frequency response, you will not hear anything in the bass region.

EXAMPLE 15C03_FletcherMunson.csd

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

instr 1
kfreq    expseg   p4, p3, p5
          printk    1, kfreq ;prints the frequencies once a second
          asin     oscil    .2, kfreq
          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>
;example by joachim heintz
```

It is very important to bear in mind when designing instruments that the perceived loudness of a sound will depend upon its frequency content. You must remain aware that projecting a 30 Hz sine at a certain amplitude will be perceived differently to a 3000 Hz sine at the same amplitude; the latter will sound much louder.

15 D. RANDOM

This chapter is in three parts. Part I provides a general introduction to the concepts behind random numbers and how to work with them in Csound. Part II focusses on a more mathematical approach. Part III introduces a number of opcodes for generating random numbers, functions and distributions and demonstrates their use in musical examples.

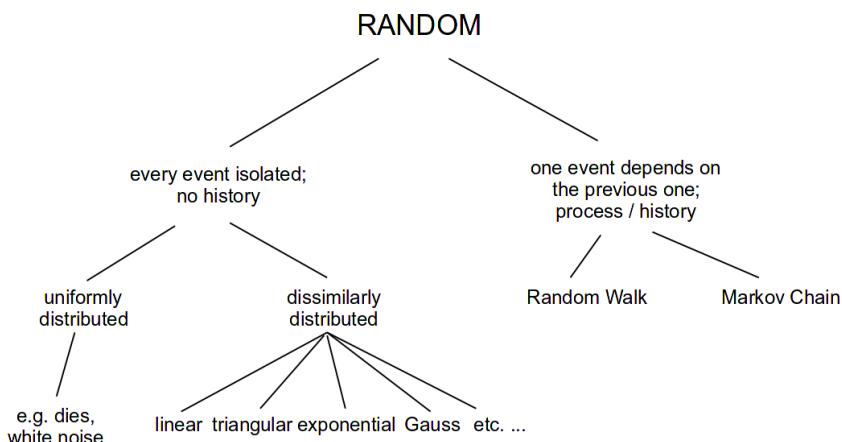
I. GENERAL INTRODUCTION

Random is Different

The term *random* derives from the idea of a horse that is running so fast it becomes *out of control* or *beyond predictability*.¹ Yet there are different ways in which to run fast and to be out of control; therefore there are different types of randomness.

We can divide types of randomness into two classes. The first contains random events that are independent of previous events. The most common example for this is throwing a die. Even if you have just thrown three One's in a row, when thrown again, a One has the same probability as before (and as any other number). The second class of random number involves random events which depend in some way upon previous numbers or states. Examples here are Markov chains and random walks.

¹<http://www.etymonline.com/index.php?term=random>



The use of randomness in electronic music is widespread. In this chapter, we shall try to explain how the different random horses are moving, and how you can create and modify them on your own. Moreover, there are many pre-built random opcodes in Csound which can be used out of the box (see the [overview](#) in the Csound Manual and the [Opcode Guide](#)). The final section of this chapter introduces some musically interesting applications of them.

Random Without History

A computer is typically only capable of computation. Computations are *deterministic* processes: one input will always generate the same output, but a random event is not predictable. To generate something which *looks like* a random event, the computer uses a pseudo-random generator.

The pseudo-random generator takes one number as input, and generates another number as output. This output is then the input for the next generation. For a huge amount of numbers, they look as if they are randomly distributed, although everything depends on the first input: the *seed*. For one given seed, the next values can be predicted.

Uniform Distribution and Seed

The output of a classical pseudo-random generator is uniformly distributed: each value in a given range has the same likelihood of occurrence. The first example shows the influence of a fixed seed (using the same chain of numbers and beginning from the same location in the chain each time) in contrast to a seed being taken from the system clock (the usual way of imitating unpredictability). The first three groups of four notes will always be the same because of the use of the same seed whereas the last three groups should always have a different pitch.

EXAMPLE 15D01_different_seed.csd

```

<CsoundSynthesizer>
<CsOptions>
-odac -m0
</CsOptions>
<CsInstruments>

```

```

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

instr generate
;get seed: 0 = seeding from system clock
;          otherwise = fixed seed
    seed      p4
;generate four notes to be played from subinstrument
iNoteCount =
0
while iNoteCount < 4 do
iFreq    random 400, 800
        schedule "play", iNoteCount, 2, iFreq
iNoteCount += 1 ;increase note count
od
endin

instr play
iFreq      =      p4
            print iFreq
aImp      mpulse .5, p3
aMode     mode    aImp, iFreq, 1000
aEnv      linen   aMode, 0.01, p3, p3-0.01
            outs    aEnv, aEnv
endin
</CsInstruments>
<CsScore>
;repeat three times with fixed seed
r 3
i "generate" 0 2 1
;repeat three times with seed from the system clock
r 3
i "generate" 0 1 0
</CsScore>
</CsoundSynthesizer>
;example by joachim heintz

```

Note that a pseudo-random generator will repeat its series of numbers after as many steps as are given by the size of the generator. If a 16-bit number is generated, the series will be repeated after 65536 steps. If you listen carefully to the following example, you will hear a repetition in the structure of the white noise (which is the result of uniformly distributed amplitudes) after about 1.5 seconds in the first note.² In the second note, there is no perceivable repetition as the random generator now works with a 31-bit number.

EXAMPLE 15D02_white_noises.csd

```

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

```

²Because the sample rate is 44100 samples per second. So a repetition after 65536 samples will lead to a repetition after $65536/44100 = 1.486$ seconds.

```

instr white_noise
iBit      =          p4 ;0 = 16 bit, 1 = 31 bit
;input of rand: amplitude, fixed seed (0.5), bit size
aNoise    rand       .1, 0.5, iBit
           outs      aNoise, aNoise
endin

</CsInstruments>
<CsScore>
i "white_noise" 0 10 0
i "white_noise" 11 10 1
</CsScore>
</CsoundSynthesizer>
;example by joachim heintz

```

Two more general notes about this:

1. The way to set the seed differs from opcode to opcode. There are several opcodes such as `rand` featured above, which offer the choice of setting a seed as input parameter. For others, such as the frequently used `random` family, the seed can only be set globally via the `seed` statement. This is usually done in the header so a typical statement would be:

```

<CsInstruments>
sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1
seed 0 ;seeding from current time

```

2. Random number generation in Csound can be done at any rate. The type of the output variable tells you whether you are generating random values at i-, k- or a-rate. Many random opcodes can work at all these rates, for instance `random`:

```

1) ires  random  imin, imax
2) kres  random  kmin, kmax
3) ares  random  kmin, kmax

```

In the first case, a random value is generated only once, when an instrument is called, at initialisation. The generated value is then stored in the variable `ires`. In the second case, a random value is generated at each k-cycle, and stored in `kres`. In the third case, in each k-cycle as many random values are stored as the audio vector has in size, and stored in the variable `ares`. Have a look at example `03A16_Random_at_ika.csd` to see this at work. Chapter 03A tries to explain the background of the different rates in depth, and how to work with them.

Other Distributions

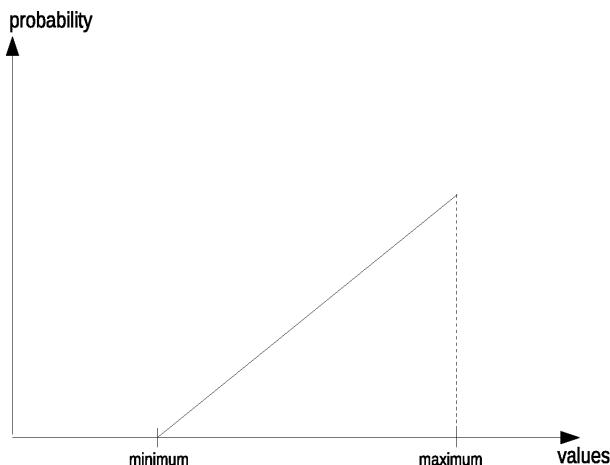
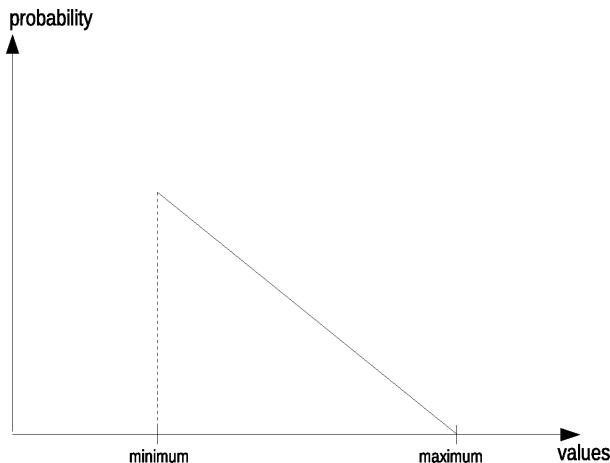
The uniform distribution is the one each computer can output via its pseudo-random generator. But there are many situations you will not want a uniformly distributed random, but any other shape. Some of these shapes are quite common, but you can actually build your own shapes quite easily in Csound. The next examples demonstrate how to do this. They are based on the chapter in Dodge/Jerse³ which also served as a model for many random number generator opcodes in

³Charles Dodge and Thomas A. Jerse, Computer Music, New York 1985, Chapter 8.1, in particular page 269-278.

Csound.⁴

Linear

A linear distribution means that either lower or higher values in a given range are more likely:



To get this behaviour, two uniform random numbers are generated, and the lower is taken for the first shape. If the second shape with the precedence of higher values is needed, the higher one of the two generated numbers is taken. The next example implements these random generators as User Defined Opcodes. First we hear a uniform distribution, then a linear distribution with precedence of lower pitches (but longer durations), at least a linear distribution with precedence of higher pitches (but shorter durations).

EXAMPLE 15D03_linrand.csd

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

```

⁴Most of them have been written by Paris Smaragdis in 1995: betarnd, bexprnd, cauchy, exprnd, gauss, linrand, pcauchy, poisson, trirand, unirand and weibull.

```

seed 0

;*****DEFINE OPCODES FOR LINEAR DISTRIBUTION****

opcode linrnd_low, i, ii
;linear random with precedence of lower values
iMin, iMax xin
;generate two random values with the random opcode
iOne      random    iMin, iMax
iTTwo     random    iMin, iMax
;compare and get the lower one
iRnd      =         iOne < iTwo ? iOne : iTwo
            xout     iRnd
endop

opcode linrnd_high, i, ii
;linear random with precedence of higher values
iMin, iMax xin
;generate two random values with the random opcode
iOne      random    iMin, iMax
iTTwo     random    iMin, iMax
;compare and get the higher one
iRnd      =         iOne > iTwo ? iOne : iTwo
            xout     iRnd
endop

;*****INSTRUMENTS FOR THE DIFFERENT DISTRIBUTIONS****

instr notes_uniform
    prints "... instr notes_uniform playing:\n"
    prints "EQUAL LIKELINESS OF ALL PITCHES AND DURATIONS\n"
;how many notes to be played
iHowMany  =          p4
;trigger as many instances of instr play as needed
iThisNote =          0
iStart    =          0
until iThisNote == iHowMany do
iMidiPch  random    36, 84 ;midi note
iDur      random    .5, 1 ;duration
            event_i  "i", "play", iStart, iDur, int(iMidiPch)
iStart    +=        iDur ;increase start
iThisNote +=        1 ;increase counter
enduntil
;reset the duration of this instr to make all events happen
p3        =          iStart + 2
;trigger next instrument two seconds after the last note
            event_i  "i", "notes_linrnd_low", p3, 1, iHowMany
endin

instr notes_linrnd_low
    prints "... instr notes_linrnd_low playing:\n"
    prints "LOWER NOTES AND LONGER DURATIONS PREFERRED\n"
iHowMany  =          p4
iThisNote =          0
iStart    =          0
until iThisNote == iHowMany do
iMidiPch  linrnd_low 36, 84 ;lower pitches preferred
iDur      linrnd_high .5, 1 ;longer durations preferred
            event_i  "i", "play", iStart, iDur, int(iMidiPch)
iStart    +=        iDur
iThisNote +=        1
enduntil
;reset the duration of this instr to make all events happen

```

```

p3          =           iStart + 2
;trigger next instrument two seconds after the last note
    event_i    "i", "notes_linrnd_high", p3, 1, iHowMany
endin

instr notes_linrnd_high
    prints    "... instr notes_linrnd_high playing:\n"
    prints    "HIGHER NOTES AND SHORTER DURATIONS PREFERRED\n"
iHowMany   =           p4
iThisNote  =           0
iStart     =           0
until iThisNote == iHowMany do
iMidiPch   linrnd_high 36, 84 ;higher pitches preferred
iDur       linrnd_low .3, 1.2 ;shorter durations preferred
    event_i    "i", "play", iStart, iDur, int(iMidiPch)
iStart     +=           iDur
iThisNote  +=           1
enduntil
;reset the duration of this instr to make all events happen
p3          =           iStart + 2
;call instr to exit csound
    event_i    "i", "exit", p3+1, 1
endin

;****INSTRUMENTS TO PLAY THE SOUNDS AND TO EXIT CSOUND****

instr play
;increase duration in random range
iDur       random      p3, p3*1.5
p3          =           iDur
;get midi note and convert to frequency
iMidiNote  =           p4
iFreq      cpsmidinn iMidiNote
;generate note with karplus-strong algorithm
aPluck     pluck       .2, iFreq, iFreq, 0, 1
aPluck     linen       aPluck, 0, p3, p3
;filter
aFilter    mode        aPluck, iFreq, .1
;mix aPluck and aFilter according to MidiNote
;(high notes will be filtered more)
aMix       ntrpol     aPluck, aFilter, iMidiNote, 36, 84
;panning also according to MidiNote
;(low = left, high = right)
iPan       =           (iMidiNote-36) / 48
aL, aR     pan2       aMix, iPan
            outs       aL, aR
endin

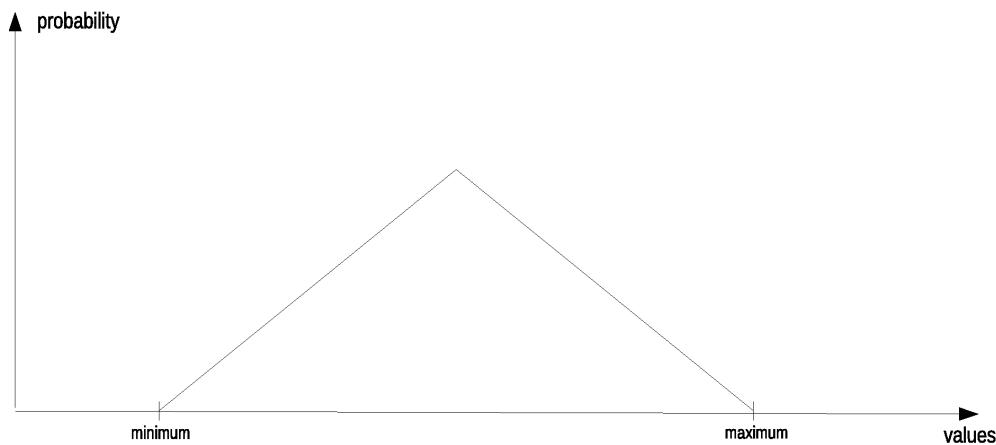
instr exit
            exitnow
endin

</CsInstruments>
<CsScore>
i "notes_uniform" 0 1 23 ;set number of notes per instr here
;instruments linrnd_low and linrnd_high are triggered automatically
e 99999 ;make possible to perform long (exit will be automatically)
</CsScore>
</CsoundSynthesizer>
;example by joachim heintz

```

Triangular

In a triangular distribution the values in the middle of the given range are more likely than those at the borders. The probability transition between the middle and the extrema are linear:



The algorithm for getting this distribution is very simple as well. Generate two uniform random numbers and take the mean of them. The next example shows the difference between uniform and triangular distribution in the same environment as the previous example.

EXAMPLE 15D04_trirand.csd

```
<CsoundSynthesizer>
<CsOptions>
-odac -m0
</CsOptions>
<CsInstruments>
sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1
seed 0

;*****UDO FOR TRIANGULAR DISTRIBUTION*****
opcode trirnd, i, ii
iMin, iMax xin
;generate two random values with the random opcode
iOne    random   iMin, iMax
iTwo    random   iMin, iMax
;get the mean and output
iRnd     =        (iOne+iTwo) / 2
          xout    iRnd
endop

;*****INSTRUMENTS FOR UNIFORM AND TRIANGULAR DISTRIBUTION****

instr notes_uniform
    prints "... instr notes_uniform playing:\n"
    prints "EQUAL LIKELINESS OF ALL PITCHES AND DURATIONS\n"
;how many notes to be played
iHowMany = p4
;trigger as many instances of instr play as needed
iThisNote = 0
iStart = 0
until iThisNote == iHowMany do
iMidiPch random 36, 84 ;midi note
```

```

iDur      random    .25, 1.75 ;duration
          event_i   "i", "play", iStart, iDur, int(iMidiPch)
iStart    +=        iDur ;increase start
iThisNote +=        1 ;increase counter
enduntil
;reset the duration of this instr to make all events happen
p3        =        iStart + 2
;trigger next instrument two seconds after the last note
          event_i   "i", "notes_trirnd", p3, 1, iHowMany
endin

instr notes_trirnd
          prints   "... instr notes_trirnd playing:\n"
          prints   "MEDIUM NOTES AND DURATIONS PREFERRED\n"
iHowMany  =        p4
iThisNote =        0
iStart    =        0
until iThisNote == iHowMany do
iMidiPch  trirnd  36, 84 ;medium pitches preferred
iDur      trirnd  .25, 1.75 ;medium durations preferred
          event_i   "i", "play", iStart, iDur, int(iMidiPch)
iStart    +=        iDur
iThisNote +=        1
enduntil
;reset the duration of this instr to make all events happen
p3        =        iStart + 2
;call instr to exit csound
          event_i   "i", "exit", p3+1, 1
endin

;*****INSTRUMENTS TO PLAY THE SOUNDS AND EXIT CSOUND****

instr play
;increase duration in random range
iDur      random    p3, p3*1.5
p3        =        iDur
;get midi note and convert to frequency
iMidiNote =        p4
iFreq     cpsmidinn iMidiNote
;generate note with karplus-strong algorithm
aPluck    pluck    .2, iFreq, iFreq, 0, 1
aPluck    linen    aPluck, 0, p3, p3
;filter
aFilter   mode     aPluck, iFreq, .1
;mix aPluck and aFilter according to MidiNote
;(high notes will be filtered more)
aMix      ntrpol   aPluck, aFilter, iMidiNote, 36, 84
;panning also according to MidiNote
;(low = left, high = right)
iPan      =        (iMidiNote-36) / 48
aL, aR    pan2    aMix, iPan
          outs    aL, aR
endin

instr exit
          exitnow
endin

</CsInstruments>
<CsScore>
i "notes_uniform" 0 1 23 ;set number of notes per instr here
;instr trirnd will be triggered automatically
e 99999 ;make possible to perform long (exit will be automatically)

```

```
</CsScore>
</CsoundSynthesizer>
;example by joachim heintz
```

More Linear and Triangular

Having written this with some very simple UDOs, it is easy to emphasise the probability peaks of the distributions by generating more than two random numbers. If you generate three numbers and choose the smallest of them, you will get many more numbers near the minimum in total for the linear distribution. If you generate three random numbers and take the mean of them, you will end up with more numbers near the middle in total for the triangular distribution.

If we want to write UDOs with a flexible number of sub-generated numbers, we have to write the code in a slightly different way. Instead of having one line of code for each random generator, we will use a loop, which calls the generator as many times as we wish to have units. A variable will store the results of the accumulation. Re-writing the above code for the UDO *trirnd* would lead to this formulation:

```
opcode trirnd, i, ii
iMin, iMax xin
    ;set a counter and a maximum count
iCount      =          0
iMaxCount   =          2
    ;set the accumulator to zero as initial value
iAccum     =          0
    ;perform loop and accumulate
until iCount == iMaxCount do
iUniRnd    random    iMin, iMax
iAccum   +=        iUniRnd
iCount   +=          1
enduntil
    ;get the mean and output
iRnd       =        iAccum / 2
            xout     iRnd
endop
```

To get this completely flexible, you only have to get *iMaxCount* as input argument. The code for the linear distribution UDOs is quite similar. – The next example shows these steps:

1. Uniform distribution.
2. Linear distribution with the precedence of lower pitches and longer durations, generated with two units.
3. The same but with four units.
4. Linear distribution with the precedence of higher pitches and shorter durations, generated with two units.
5. The same but with four units.
6. Triangular distribution with the precedence of both medium pitches and durations, generated with two units.
7. The same but with six units.

Rather than using different instruments for the different distributions, the next example combines all possibilities in one single instrument. Inside the loop which generates as many notes as desired by the *iHowMany* argument, an if-branch calculates the pitch and duration of one note depending on the distribution type and the number of sub-units used. The whole sequence (which type first,

which next, etc) is stored in the global array *giSequence*. Each instance of instrument *notes* increases the pointer *giSeqIdx*, so that for the next run the next element in the array is being read. If the pointer has reached the end of the array, the instrument which exits Csound is called instead of a new instance of *notes*.

EXAMPLE 15D05_more_lin_tri_units.csd

```

<CsoundSynthesizer>
<CsOptions>
-odac -m0
</CsOptions>
<CsInstruments>
sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1
seed 0

;*****SEQUENCE OF UNITS AS ARRAY*****
giSequence[] array 0, 1.2, 1.4, 2.2, 2.4, 3.2, 3.6
giSeqIdx = 0 ;startindex

;*****UDO DEFINITIONS*****
opcode linrnd_low, i, iii
;linear random with precedence of lower values
iMin, iMax, iMaxCount xin
;set counter and initial (absurd) result
iCount = 0
iRnd = iMax
;loop and reset iRnd
until iCount == iMaxCount do
iUniRnd random iMin, iMax
iRnd = iUniRnd < iRnd ? iUniRnd : iRnd
iCount += 1
enduntil
xout iRnd
endop

opcode linrnd_high, i, iii
;linear random with precedence of higher values
iMin, iMax, iMaxCount xin
;set counter and initial (absurd) result
iCount = 0
iRnd = iMin
;loop and reset iRnd
until iCount == iMaxCount do
iUniRnd random iMin, iMax
iRnd = iUniRnd > iRnd ? iUniRnd : iRnd
iCount += 1
enduntil
xout iRnd
endop

opcode trirnd, i, iii
iMin, iMax, iMaxCount xin
;set a counter and accumulator
iCount = 0
iAccum = 0
;perform loop and accumulate
until iCount == iMaxCount do
iUniRnd random iMin, iMax

```

```

iAccum    +=      iUniRnd
iCount    +=      1
enduntil
;get the mean and output
iRnd      =      iAccum / iMaxCount
xout      =      iRnd
endop

;*****ONE INSTRUMENT TO PERFORM ALL DISTRIBUTIONS*****
;0 = uniform, 1 = linrnd_low, 2 = linrnd_high, 3 = trirnd
;the fractional part denotes the number of units, e.g.
;3.4 = triangular distribution with four sub-units

instr notes
;how many notes to be played
iHowMany  =      p4
;by which distribution with how many units
iWhich   =      giSequence[giSeqIdx]
iDistrib  =      int(iWhich)
iUnits   =      round(frac(iWhich) * 10)
;set min and max duration
iMinDur  =      .1
iMaxDur  =      2
;set min and max pitch
iMinPch  =      36
iMaxPch  =      84

;trigger as many instances of instr play as needed
iThisNote =      0
iStart    =      0
iPrint    =      1

;for each note to be played
until iThisNote == iHowMany do

;calculate iMidiPch and iDur depending on type
if iDistrib == 0 then
printf_i  "%s", iPrint, "... uniform distribution:\n"
printf_i  "%s", iPrint, "EQUAL LIKELIHOOD OF ALL PITCHES AND DURATIONS\n"
iMidiPch random iMinPch, iMaxPch ;midi note
iDur     random iMinDur, iMaxDur ;duration
elseif iDistrib == 1 then
printf_i  "... linear low distribution with %d units:\n", iPrint, iUnits
printf_i  "%s", iPrint, "LOWER NOTES AND LONGER DURATIONS PREFERRED\n"
iMidiPch linrnd_low iMinPch, iMaxPch, iUnits
iDur     linrnd_high iMinDur, iMaxDur, iUnits
elseif iDistrib == 2 then
printf_i  "... linear high distribution with %d units:\n", iPrint, iUnits
printf_i  "%s", iPrint, "HIGHER NOTES AND SHORTER DURATIONS PREFERRED\n"
iMidiPch linrnd_high iMinPch, iMaxPch, iUnits
iDur     linrnd_low iMinDur, iMaxDur, iUnits
else
printf_i  "... triangular distribution with %d units:\n", iPrint, iUnits
printf_i  "%s", iPrint, "MEDIUM NOTES AND DURATIONS PREFERRED\n"
iMidiPch trirnd iMinPch, iMaxPch, iUnits
iDur     trirnd iMinDur, iMaxDur, iUnits
endif

;call subinstrument to play note
event_i  "i", "play", iStart, iDur, int(iMidiPch)

;increase start time and counter
iStart    +=      iDur

```

```

iThisNote += 1
;avoid continuous printing
iPrint = 0
enduntil

;reset the duration of this instr to make all events happen
p3 = iStart + 2

;increase index for sequence
giSeqIdx += 1
;call instr again if sequence has not been ended
if giSeqIdx < lenarray(giSequence) then
    event_i "i", "notes", p3, 1, iHowMany
;or exit
else
    event_i "i", "exit", p3, 1
endif
endin

;*****INSTRUMENTS TO PLAY THE SOUNDS AND EXIT CSOUND*****
instr play
;increase duration in random range
iDur random p3, p3*1.5
p3 = iDur
;get midi note and convert to frequency
iMidiNote = p4
iFreq cpsmidinn iMidiNote
;generate note with karplus-strong algorithm
aPluck pluck .2, iFreq, iFreq, 0, 1
aPluck linen aPluck, 0, p3, p3
;filter
aFilter mode aPluck, iFreq, .1
;mix aPluck and aFilter according to MidiNote
;(high notes will be filtered more)
aMix ntrpol aPluck, aFilter, iMidiNote, 36, 84
;panning also according to MidiNote
;(low = left, high = right)
iPan = (iMidiNote-36) / 48
aL, aR pan2 aMix, iPan
        outs aL, aR
endin

instr exit
    exitnow
endin

</CsInstruments>
<CsScore>
i "notes" 0 1 23 ;set number of notes per instr here
e 99999 ;make possible to perform long (exit will be automatically)
</CsScore>
</CsoundSynthesizer>
;example by joachim heintz

```

With this method we can build probability distributions which are very similar to exponential or gaussian distributions.⁵ Their shape can easily be formed by the number of sub-units used.

⁵According to Dodge/Jerse, the usual algorithms for exponential and gaussian are: Exponential: Generate a uniformly distributed number between 0 and 1 and take its natural logarithm. Gauss: Take the mean of uniformly distributed numbers and scale them by the standard deviation.

Scalings

Random is a complex and sensible context. There are so many ways to let the horse go, run, or dance – the conditions you set for this *way of moving* are much more important than the fact that one single move is not predictable. What are the conditions of this randomness?

- *Which Way.* This is what has already been described: random with or without history, which probability distribution, etc.
- *Which Range.* This is a decision which comes from the composer/programmer. In the example above I have chosen pitches from Midi Note 36 to 84 (C2 to C6), and durations between 0.1 and 2 seconds. Imagine how it would have been sounded with pitches from 60 to 67, and durations from 0.9 to 1.1 seconds, or from 0.1 to 0.2 seconds. There is no range which is “correct”, everything depends on the musical idea.
- *Which Development.* Usually the boundaries will change in the run of a piece. The pitch range may move from low to high, or from narrow to wide; the durations may become shorter, etc.
- *Which Scalings.* Let us think about this more in detail.

In the example above we used two implicit scalings. The pitches have been scaled to the keys of a piano or keyboard. Why? We do not play piano here obviously ... – What other possibilities might have been instead? One would be: no scaling at all. This is the easiest way to go – whether it is really the best, or simple laziness, can only be decided by the composer or the listener.

Instead of using the equal tempered chromatic scale, or no scale at all, you can use any other ways of selecting or quantising pitches. Be it any which has been, or is still, used in any part of the world, or be it your own invention, by whatever fantasy or invention or system.

As regards the durations, the example above has shown no scaling at all. This was definitely laziness...

The next example is essentially the same as the previous one, but it uses a pitch scale which represents the overtone scale, starting at the second partial extending upwards to the 32nd partial. This scale is written into an array by a statement in instrument 0. The durations have fixed possible values which are written into an array (from the longest to the shortest) by hand. The values in both arrays are then called according to their position in the array.

EXAMPLE 15D06_scalings.csd

```
<CsoundSynthesizer>
<CsOptions>
-odac -m0
</CsOptions>
<CsInstruments>
sr = 44100
ksmps = 32
nchnls = 2
0dbfs = 1
seed 0

;*****POSSIBLE DURATIONS AS ARRAY*****
giDurs[]    array    3/2, 1, 2/3, 1/2, 1/3, 1/4
giLenDurs   lenarray  giDurs
```

```

;*****POSSIBLE PITCHES AS ARRAY*****
;initialize array with 31 steps
giScale[] init 31
giLenScale lenarray giScale
;iterate to fill from 65 hz onwards
iStart = 65
iDenom = 3 ;start with 3/2
iCnt = 0
until iCnt = giLenScale do
giScale[iCnt] = iStart
iStart = iStart * iDenom / (iDenom-1)
iDenom += 1 ;next proportion is 4/3 etc
iCnt += 1
enduntil

;*****SEQUENCE OF UNITS AS ARRAY*****
giSequence[] array 0, 1.2, 1.4, 2.2, 2.4, 3.2, 3.6
giSeqIdx = 0 ;startindex

;*****UDO DEFINITIONS*****
opcode linrnd_low, i, iii
;linear random with precedence of lower values
iMin, iMax, iMaxCount xin
;set counter and initial (absurd) result
iCount = 0
iRnd = iMax
;loop and reset iRnd
until iCount == iMaxCount do
iUniRnd random iMin, iMax
iRnd = iUniRnd < iRnd ? iUniRnd : iRnd
iCount += 1
enduntil
xout iRnd
endop

opcode linrnd_high, i, iii
;linear random with precedence of higher values
iMin, iMax, iMaxCount xin
;set counter and initial (absurd) result
iCount = 0
iRnd = iMin
;loop and reset iRnd
until iCount == iMaxCount do
iUniRnd random iMin, iMax
iRnd = iUniRnd > iRnd ? iUniRnd : iRnd
iCount += 1
enduntil
xout iRnd
endop

opcode trirnd, i, iii
iMin, iMax, iMaxCount xin
;set a counter and accumulator
iCount = 0
iAccum = 0
;perform loop and accumulate
until iCount == iMaxCount do
iUniRnd random iMin, iMax
iAccum += iUniRnd
iCount += 1
enduntil
;get the mean and output
iRnd = iAccum / iMaxCount

```

```

        xout      iRnd
endop

;*****ONE INSTRUMENT TO PERFORM ALL DISTRIBUTIONS*****
;0 = uniform, 1 = linrnd_low, 2 = linrnd_high, 3 = trirnd
;the fractional part denotes the number of units, e.g.
;3.4 = triangular distribution with four sub-units

instr notes
;how many notes to be played
iHowMany    =          p4
;by which distribution with how many units
iWhich     =          giSequence[giSeqIdx]
iDistrib    =          int(iWhich)
iUnits     =          round(frac(iWhich) * 10)

;trigger as many instances of instr play as needed
iThisNote   =          0
iStart      =          0
iPrint      =          1

;for each note to be played
until iThisNote == iHowMany do

;calculate iMidiPch and iDur depending on type
if iDistrib == 0 then
printf_i  "%s", iPrint, "... uniform distribution:\n"
printf_i  "%s", iPrint, "EQUAL LIKELINESS OF ALL PITCHES AND DURATIONS\n"
iScaleIndx random  0, giLenScale-.0001 ;midi note
iDurIndx  random  0, giLenDurs-.0001 ;duration
elseif iDistrib == 1 then
printf_i  "... linear low distribution with %d units:\n", iPrint, iUnits
printf_i  "%s", iPrint, "LOWER NOTES AND LONGER DURATIONS PREFERRED\n"
iScaleIndx linrnd_low 0, giLenScale-.0001, iUnits
iDurIndx  linrnd_low 0, giLenDurs-.0001, iUnits
elseif iDistrib == 2 then
printf_i  "... linear high distribution with %d units:\n", iPrint, iUnits
printf_i  "%s", iPrint, "HIGHER NOTES AND SHORTER DURATIONS PREFERRED\n"
iScaleIndx linrnd_high 0, giLenScale-.0001, iUnits
iDurIndx  linrnd_high 0, giLenDurs-.0001, iUnits
else
printf_i  "... triangular distribution with %d units:\n", iPrint, iUnits
printf_i  "%s", iPrint, "MEDIUM NOTES AND DURATIONS PREFERRED\n"
iScaleIndx trirnd  0, giLenScale-.0001, iUnits
iDurIndx  trirnd  0, giLenDurs-.0001, iUnits
endif

;call subinstrument to play note
iDur       =          giDurs[int(iDurIndx)]
iPch       =          giScale[int(iScaleIndx)]
event_i    "i", "play", iStart, iDur, iPch

;increase start time and counter
iStart    +=          iDur
iThisNote +=          1
;avoid continuous printing
iPrint    =          0
enduntil

;reset the duration of this instr to make all events happen
p3        =          iStart + 2

;increase index for sequence

```

```

giSeqIdx += 1
;call instr again if sequence has not been ended
if giSeqIdx < lenarray(giSequence) then
    event_i    "i", "notes", p3, 1, iHowMany
;or exit
    else
        event_i    "i", "exit", p3, 1
endif
endin

;*****INSTRUMENTS TO PLAY THE SOUNDS AND EXIT CSOUND*****
instr play
    ;increase duration in random range
    iDur      random    p3*2, p3*5
    p3       =          iDur
    ;get frequency
    iFreq     =          p4
    ;generate note with karplus-strong algorithm
    aPluck   pluck     .2, iFreq, iFreq, 0, 1
    aPluck   linen     aPluck, 0, p3, p3
    ;filter
    aFilter   mode      aPluck, iFreq, .1
    ;mix aPluck and aFilter according to freq
    ;(high notes will be filtered more)
    aMix     ntrpol    aPluck, aFilter, iFreq, 65, 65*16
    ;panning also according to freq
    ;(low = left, high = right)
    iPan     =          (iFreq-65) / (65*16)
    aL, aR   pan2     aMix, iPAn
            outs     aL, aR
endin

instr exit
    exitnow
endin
</CsInstruments>
<CsScore>
i "notes" 0 1 23 ;set number of notes per instr here
e 99999 ;make possible to perform long (exit will be automatically)
</CsScore>
</CsoundSynthesizer>
;example by joachim heintz

```

Random With History

There are many ways a current value in a random number progression can influence the next. Two of them are used frequently. A Markov chain is based on a number of possible states, and defines a different probability for each of these states. A random walk looks at the last state as a position in a range or field, and allows only certain deviations from this position.

Markov Chains

A typical case for a Markov chain in music is a sequence of certain pitches or notes. For each note, the probability of the following note is written in a table like this:

		next element	
		a	b
previous element	a	0.2	0.5
	b	0.5	0.0
	c	0.1	0.8

This means: the probability that element *a* is repeated, is 0.2; the probability that *b* follows *a* is 0.5; the probability that *c* follows *a* is 0.3. The sum of all probabilities must, by convention, add up to 1. The following example shows the basic algorithm which evaluates the first line of the Markov table above, in the case, the previous element has been *a*.

EXAMPLE 15D07_markov_basics.csd

```
<CsoundSynthesizer>
<CsOptions>
-ndm0
</CsOptions>
<CsInstruments>
sr = 44100
ksmps = 32
0dbfs = 1
nchnls = 1
seed 0

instr 1
iLine[]    array      .2, .5, .3
iVal        random     0, 1
iAccum     =           iLine[0]
iIndex     =
0
until iAccum >= iVal do
iIndex += 1
iAccum += iLine[iIndex]
enduntil
printf_i "Random number = %.3f, next element = %c!\n", 1, iVal, iIndex+97
endin
</CsInstruments>
<CsScore>
r 10
i 1 0 0
</CsScore>
</CsoundSynthesizer>
;example by joachim heintz
```

The probabilities are 0.2 0.5 0.3. First a uniformly distributed random number between 0 and 1 is generated. An accumulator is set to the first element of the line (here 0.2). It is interrogated as to whether it is larger than the random number. If so then the index is returned, if not, the second element is added (0.2+0.5=0.7), and the process is repeated, until the accumulator is greater or equal the random value. The output of the example should show something like this:

```
Random number = 0.850, next element = c!
Random number = 0.010, next element = a!
Random number = 0.805, next element = c!
Random number = 0.696, next element = b!
Random number = 0.626, next element = b!
Random number = 0.476, next element = b!
Random number = 0.420, next element = b!
Random number = 0.627, next element = b!
```

```
Random number = 0.065, next element = a!
Random number = 0.782, next element = c!
```

The next example puts this algorithm in an User Defined Opcode. Its input is a Markov table as a two-dimensional array, and the previous line as index (starting with 0). Its output is the next element, also as index. – There are two Markov chains in this example: seven pitches, and three durations. Both are defined in two-dimensional arrays: *giProbNotes* and *giProbDurs*. Both Markov chains are running independently from each other.

EXAMPLE 15D08_markov_music.csd

```
<CsoundSynthesizer>
<CsOptions>
-m128 -odac
</CsOptions>
<CsInstruments>
sr = 44100
ksmps = 32
0dbfs = 1
nchnls = 2
seed 0

;*****USER DEFINED OPCODES FOR MARKOV CHAINS*****
opcode Markov, i, i[][]i
iMarkovTable[], iPrevEl xin
iRandom    random    0, 1
iNextEl    =
iAccum     =         iMarkovTable[iPrevEl][iNextEl]
until iAccum >= iRandom do
iNextEl    +=
iAccum     +=         iMarkovTable[iPrevEl][iNextEl]
enduntil
        xout      iNextEl
endop
opcode Markovk, k, k[][]k
kMarkovTable[], kPrevEl xin
kRandom    random    0, 1
kNextEl    =
kAccum     =         kMarkovTable[kPrevEl][kNextEl]
until kAccum >= kRandom do
kNextEl    +=
kAccum     +=         kMarkovTable[kPrevEl][kNextEl]
enduntil
        xout      kNextEl
endop

;*****DEFINITIONS FOR NOTES*****
;notes as proportions and a base frequency
giNotes[] array 1, 9/8, 6/5, 5/4, 4/3, 3/2, 5/3
giBasFreq = 330
;probability of notes as markov matrix:
;first -> only to third and fourth
;second -> anywhere without self
;third -> strong probability for repetitions
;fourth -> idem
;fifth -> anywhere without third and fourth
;sixth -> mostly to seventh
;seventh -> mostly to sixth
giProbNotes[][] init 7, 7
giProbNotes fillarray 0.0, 0.0, 0.5, 0.5, 0.0, 0.0, 0.0,
```

```

0.2, 0.0, 0.2, 0.2, 0.2, 0.1, 0.1,
0.1, 0.1, 0.5, 0.1, 0.1, 0.1, 0.0,
0.0, 0.1, 0.1, 0.5, 0.1, 0.1, 0.1,
0.2, 0.2, 0.0, 0.0, 0.2, 0.2, 0.2,
0.1, 0.1, 0.0, 0.0, 0.1, 0.1, 0.6,
0.1, 0.1, 0.0, 0.0, 0.1, 0.6, 0.1

;*****DEFINITIONS FOR DURATIONS*****
;possible durations
gkDurs[] array 1, 1/2, 1/3
;probability of durations as markov matrix:
;first -> anything
;second -> mostly self
;third -> mostly second
gkProbDurs[][] init 3, 3
gkProbDurs array 1/3, 1/3, 1/3,
0.2, 0.6, 0.3,
0.1, 0.5, 0.4

;*****SET FIRST NOTE AND DURATION FOR MARKOV PROCESS*****
giPrevNote init 1
gkPrevDur init 1

;*****INSTRUMENT FOR DURATIONS*****
instr trigger_note
kTrig metro 1/gkDurs[gkPrevDur]
if kTrig == 1 then
    event "i", "select_note", 0, 1
gkPrevDur Markovk gkProbDurs, gkPrevDur
endif
endin

;*****INSTRUMENT FOR PITCHES*****
instr select_note
;choose next note according to markov matrix and previous note
;and write it to the global variable for (next) previous note
giPrevNote Markovv giProbNotes, giPrevNote
;call instr to play this note
    event_i "i", "play_note", 0, 2, giPrevNote
;turn off this instrument
    turnoff
endin

;*****INSTRUMENT TO PERFORM ONE NOTE*****
instr play_note
;get note as index in ginotes array and calculate frequency
iNote = p4
iFreq = giBasFreq * giNotes[iNote]
;random choice for mode filter quality and panning
iQ random 10, 200
iPan random 0.1, .9
;generate tone and put out
aImp mpulse 1, p3
aOut mode aImp, iFreq, iQ
aL, aR pan2 aOut, iPan
outs aL, aR
endin

</CsInstruments>
<CsScore>
i "trigger_note" 0 100
</CsScore>
</CsoundSynthesizer>
```

```
;example by joachim heintz
```

Random Walk

In the context of movement between random values, *walk* can be thought of as the opposite of *jump*. If you jump within the boundaries A and B, you can end up anywhere between these boundaries, but if you walk between A and B you will be limited by the extent of your step - each step applies a deviation to the previous one. If the deviation range is slightly more positive (say from -0.1 to +0.2), the general trajectory of your walk will be in the positive direction (but individual steps will not necessarily be in the positive direction). If the deviation range is weighted negative (say from -0.2 to 0.1), then the walk will express a generally negative trajectory.

One way of implementing a random walk will be to take the current state, derive a random deviation, and derive the next state by adding this deviation to the current state. The next example shows two ways of doing this.

The *pitch* random walk starts at pitch 8 in octave notation. The general pitch deviation *gkPitchDev* is set to 0.2, so that the next pitch could be between 7.8 and 8.2. But there is also a pitch direction *gkPitchDir* which is set to 0.1 as initial value. This means that the upper limit of the next random pitch is 8.3 instead of 8.2, so that the pitch will move upwards in a greater number of steps. When the upper limit *giHighestPitch* has been crossed, the *gkPitchDir* variable changes from +0.1 to -0.1, so after a number of steps, the pitch will have become lower. Whenever such a direction change happens, the console reports this with a message printed to the terminal.

The *density* of the notes is defined as notes per second, and is applied as frequency to the *metro* opcode in instrument *walk*. The lowest possible density *giLowestDens* is set to 1, the highest to 8 notes per second, and the first density *giStartDens* is set to 3. The possible random deviation for the next density is defined in a range from zero to one: zero means no deviation at all, one means that the next density can alter the current density in a range from half the current value to twice the current value. For instance, if the current density is 4, for *gkDensDev*=1 you would get a density between 2 and 8. The direction of the densities *gkDensDir* in this random walk follows the same range 0..1. Assumed you have no deviation of densities at all (*gkDensDev*=0), *gkDensDir*=0 will produce ticks in always the same speed, whilst *gkDensDir*=1 will produce a very rapid increase in speed. Similar to the pitch walk, the direction parameter changes from plus to minus if the upper border has crossed, and vice versa.

EXAMPLE 15D09_random_walk.csd

```
<CsoundSynthesizer>
<CsOptions>
-m128 -odac
</CsOptions>
<CsInstruments>
sr = 44100
ksmps = 32
0dbfs = 1
nchnls = 2
seed 1 ;change to zero for always changing results

;*****SETTINGS FOR PITCHES*****
;define the pitch street in octave notation
giLowestPitch =    7
```

```

giHighestPitch = 9
;set pitch startpoint, deviation range and the first direction
giStartPitch = 8
gkPitchDev init 0.2 ;random range for next pitch
gkPitchDir init 0.1 ;positive = upwards

;*****SETTINGS FOR DENSITY*****
;define the maximum and minimum density (notes per second)
giLowestDens = 1
giHighestDens = 8
;set first density
giStartDens = 3
;set possible deviation in range 0..1
;0 = no deviation at all
;1 = possible deviation is between half and twice the current density
gkDensDev init 0.5
;set direction in the same range 0..1
;(positive = more dense, shorter notes)
gkDensDir init 0.1

;*****INSTRUMENT FOR RANDOM WALK*****
instr walk
;set initial values
kPitch init giStartPitch
kDens init giStartDens
;trigger impulses according to density
kTrig metro kDens
;if the metro ticks
if kTrig == 1 then
;1) play current note
    event "i", "play", 0, 1.5/kDens, kPitch
;2) calculate next pitch
;define boundaries according to direction
kLowPchBound = gkPitchDir < 0 ? -gkPitchDev+gkPitchDir : -gkPitchDev
kHighPchBound = gkPitchDir > 0 ? gkPitchDev+gkPitchDir : gkPitchDev
;get random value in these boundaries
kPchRnd random kLowPchBound, kHighPchBound
;add to current pitch
kPitch += kPchRnd
;change direction if maxima are crossed, and report
if kPitch > giHighestPitch && gkPitchDir > 0 then
gkPitchDir = -gkPitchDir
    printk " Pitch touched maximum - now moving down.\n", 0
elseif kPitch < giLowestPitch && gkPitchDir < 0 then
gkPitchDir = -gkPitchDir
    printk "Pitch touched minimum - now moving up.\n", 0
endif
;3) calculate next density (= metro frequency)
;define boundaries according to direction
kLowDensBound = gkDensDir < 0 ? -gkDensDev+gkDensDir : -gkDensDev
kHighDensBound = gkDensDir > 0 ? gkDensDev+gkDensDir : gkDensDev
;get random value in these boundaries
kDensRnd random kLowDensBound, kHighDensBound
;get multiplier (so that kDensRnd=1 yields to 2, and kDens=-1 to 1/2)
kDensMult = 2 ^ kDensRnd
;multiply with current duration
kDens *= kDensMult
;avoid too high values and too low values
kDens = kDens > giHighestDens*1.5 ? giHighestDens*1.5 : kDens
kDens = kDens < giLowestDens/1.5 ? giLowestDens/1.5 : kDens
;change direction if maxima are crossed
if (kDens > giHighestDens && gkDensDir > 0) ||
(kDens < giLowestDens && gkDensDir < 0) then

```

```

gkDensDir = -gkDensDir
if kDens > giHighestDens then
printks " Density touched upper border - now becoming less dense.\n", 0
else
printks " Density touched lower border - now becoming more dense.\n", 0
endif
endif
endif
endin

;*****INSTRUMENT TO PLAY ONE NOTE*****
instr play
;get note as octave and calculate frequency and panning
iOct      =      p4
iFreq     =      cpsoct(iOct)
iPan      ntrpol  0, 1, iOct, giLowestPitch, giHighestPitch
;calculate mode filter quality according to duration
iQ         ntrpol  10, 400, p3, .15, 1.5
;generate tone and throw out
aImp      mpulse   1, p3
aMode     mode     aImp, iFreq, iQ
aOut      linen    aMode, 0, p3, p3/4
aL, aR    pan2    aOut, iPan
          outs    aL, aR
endin

</CsInstruments>
<CsScore>
i "walk" 0 999
</CsScore>
</CsoundSynthesizer>
;example by joachim heintz

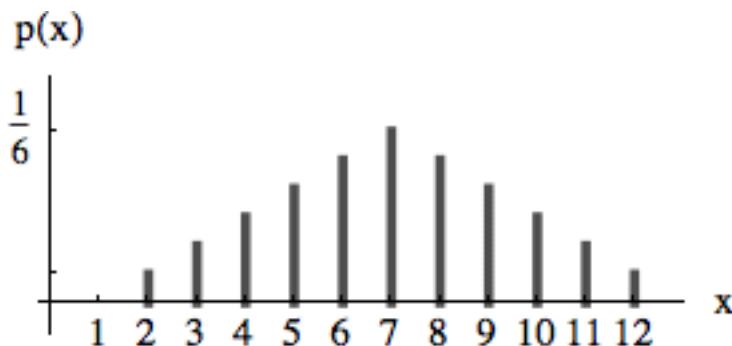
```

II. SOME MATHS PERSPECTIVES ON RANDOM

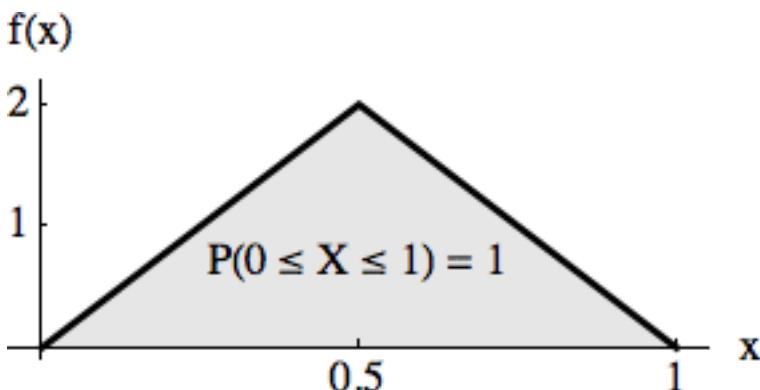
Random Processes

The relative frequency of occurrence of a random variable can be described by a probability function (for discrete random variables) or by density functions (for continuous random variables).

When two dice are thrown simultaneously, the sum x of their numbers can be 2, 3, ...12. The following figure shows the probability function $p(x)$ of these possible outcomes. $p(x)$ is always less than or equal to 1. The sum of the probabilities of all possible outcomes is 1.



For continuous random variables the probability of getting a specific value x is 0. But the probability of getting a value within a certain interval can be indicated by an area that corresponds to this probability. The function $f(x)$ over these areas is called the density function. With the following density the chance of getting a number smaller than 0 is 0, to get a number between 0 and 0.5 is 0.5, to get a number between 0.5 and 1 is 0.5 etc. Density functions $f(x)$ can reach values greater than 1 but the area under the function is 1.



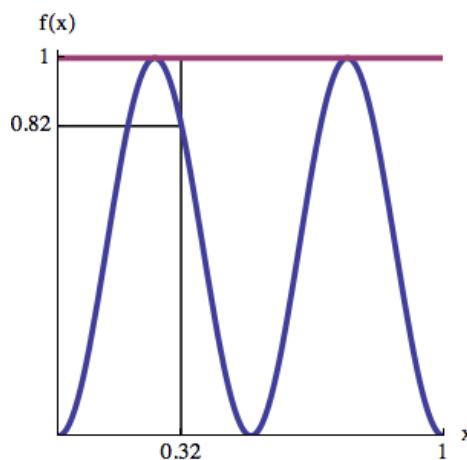
Generating Random Numbers With a Given Probability or Density

Csound provides opcodes for some specific densities but no means to produce random numbers with user defined probability or density functions. The opcodes *rand_density* and *rand_probability* (see below) generate random numbers with probabilities or densities given by tables. They are realized by using the so-called *rejection sampling method*.

Rejection Sampling

The principle of *rejection sampling* is to first generate uniformly distributed random numbers in the range required and to then accept these values corresponding to a given density function (or otherwise reject them). Let us demonstrate this method using the density function shown in the next figure. (Since the rejection sampling method uses only the *shape* of the function, the area under the function need not be 1). We first generate uniformly distributed random numbers *rnd1* over the interval $[0, 1]$. Of these we accept a proportion corresponding to $f(rnd1)$. For example, the value 0.32 will only be accepted in the proportion of $f(0.32) = 0.82$. We do this by generating a new random number *rnd2* between 0 and 1 and accept *rnd1* only if $rnd2 < f(rnd1)$; otherwise we reject it. (see *Signals, Systems and Sound Synthesis*⁶ chapter 10.1.4.4)

⁶Neukom, Martin. Signals, systems and sound synthesis. Bern: Peter Lang, 2013. Print.

**EXAMPLE 15D10_Rejection_Sampling.csd**

```

<CsoundSynthesizer>
<CsOptions>
-odac
</CsOptions>
<CsInstruments>
sr = 44100
ksmps = 10
nchnls = 1
0dbfs = 1

; random number generator to a given density function
; kout random number; k_minimum,k_maximum,i_fn for a density function

opcode rand_density, k, kki
kmin,kmax,ifn  xin
loop:
krnd1    random      0,1
krnd2    random      0,1
k2        table       krnd1,ifn,1
            if         krnd2 > k2  kgoto loop
            xout      kmin+krnd1*(kmax-kmin)
endop

; random number generator to a given probability function
; kout random number
; in: i_nr number of possible values
; i_fn1 function for random values
; i_fn2 probability functionExponential: Generate a uniformly distributed
; number between 0 and 1 and take its natural logarithm.

opcode rand_probability, k, iii
inr,ifn1,ifn2  xin
loop:
krnd1    random      0,inr
krnd2    random      0,1
k2        table       int(krnd1),ifn2,0
            if         krnd2 > k2  kgoto loop
kout     table       krnd1,ifn1,0
            xout      kout
endop

instr 1
krnd    rand_density  400,800,2
aout    poscil        .1,krnd,1

```

```

        out          aout
endin

instr 2
krnd      rand_probability p4,p5,p6
aout      poscil      .1,krnd,1
          out         aout
endin

</CsInstruments>
<CsScore>
;sine
f1 0 32768 10 1
;density function
f2 0 1024 6 1 112 0 800 0 112 1
;random values and their relative probability (two dice)
f3 0 16 -2 2 3 4 5 6 7 8 9 10 11 12
f4 0 16 2 1 2 3 4 5 6 5 4 3 2 1
;random values and their relative probability
f5 0 8 -2 400 500 600 800
f6 0 8 2 .3 .8 .3 .1

i1 0 10
i2 0 10 4 5 6
</CsScore>
</CsoundSynthesizer>
;example by martin neukom

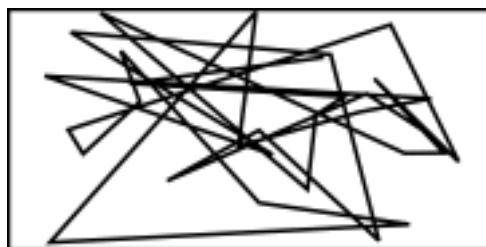
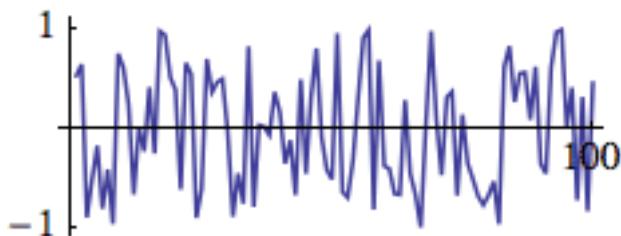
```

Random Walk

In a series of random numbers the single numbers are independent upon each other. Parameter (left figure) or paths in the room (two-dimensional trajectory in the right figure) created by random numbers wildly jump around.

Example 1

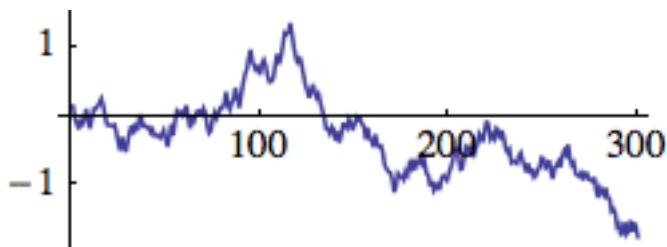
```
Table[RandomReal[{-1, 1}], {100}];
```



We get a smoother path, a so-called random walk, by adding at every time step a random number r to the actual position x ($x += r$).

Example 2

```
x = 0; walk = Table[x += RandomReal[{-0.2, 0.2}], {300}];
```



The path becomes even smoother by adding a random number r to the actual velocity v .

```
v += r
x += v
```

The path can be bounded to an area (figure to the right) by inverting the velocity if the path exceeds the limits (\min, \max):

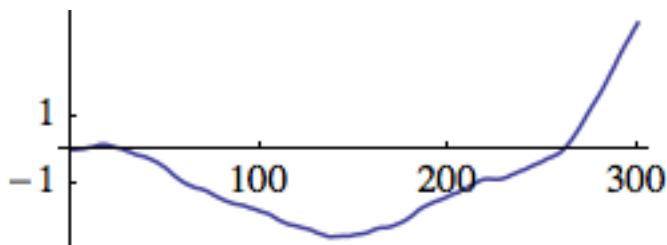
```
vif(x < min || x > max) v *= -1
```

The movement can be damped by decreasing the velocity at every time step by a small factor d

```
v *= (1-d)
```

Example 3

```
x = 0; v = 0; walk = Table[x += v += RandomReal[{-0.01, 0.01}], {300}];
```

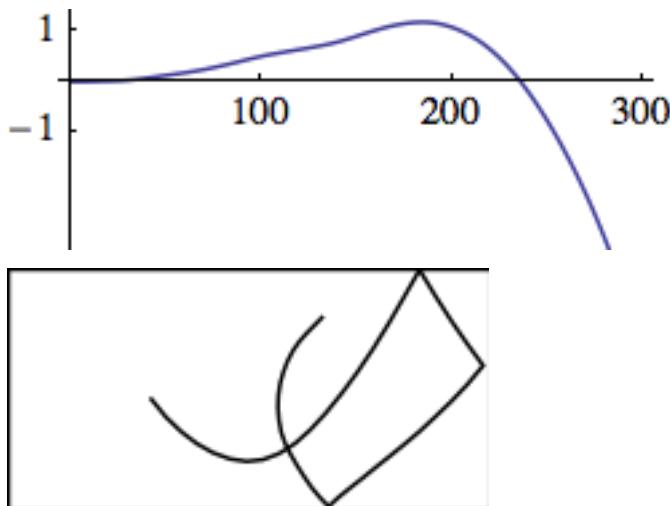


The path becomes again smoother by adding a random number r to the actual acceleration a , the change of the acceleration, etc.

```
a += r
v += a
x += v
```

Example 4

```
x = 0; v = 0; a = 0;
Table[x += v += a += RandomReal[{-0.0001, .0001}], {300}];
```



(see Martin Neukom, *Signals, Systems and Sound Synthesis* chapter 10.2.3.2)

EXAMPLE 15D11_Random_Walk2.csd

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

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

; random frequency
instr 1
kx    random  -p6, p6
kfreq = p5*2^kx
aout  oscil  p4, kfreq, 1
out   aout
endin

; random change of frequency
instr 2
kx    init    .5
kfreq = p5*2^kx
kv    random  -p6, p6
kv    =      kv*(1 - p7)
kx    =      kx + kv
aout  oscil  p4, kfreq, 1
out   aout
endin

; random change of change of frequency
instr 3
kv    init    0
kx    init    .5
kfreq = p5*2^kx
ka    random  -p7, p7
```

```

kv      =      kv + ka
kv      =      kv*(1 - p8)
kx      =      kx + kv
kv      =      (kx < -p6 || kx > p6?-kv : kv)
aout    oscili p4, kfreq, 1
out     aout

endin

</CsInstruments>
<CsScore>

f1 0 32768 10 1
; i1    p4      p5      p6
; i2    p4      p5      p6      p7
;      amp    c_fr    rand   damp
; i2 0 20 .1      600    0.01   0.001
;      amp    c_fr    d_fr    rand   damp
;      amp    c_fr    rand
; i1 0 20 .1      600    0.5
; i3    p4      p5      p6      p7      p8
i3 0 20 .1      600    1       0.001  0.001
</CsScore>
</CsoundSynthesizer>
;example by martin neukom

```

III. MISCELLANEOUS EXAMPLES

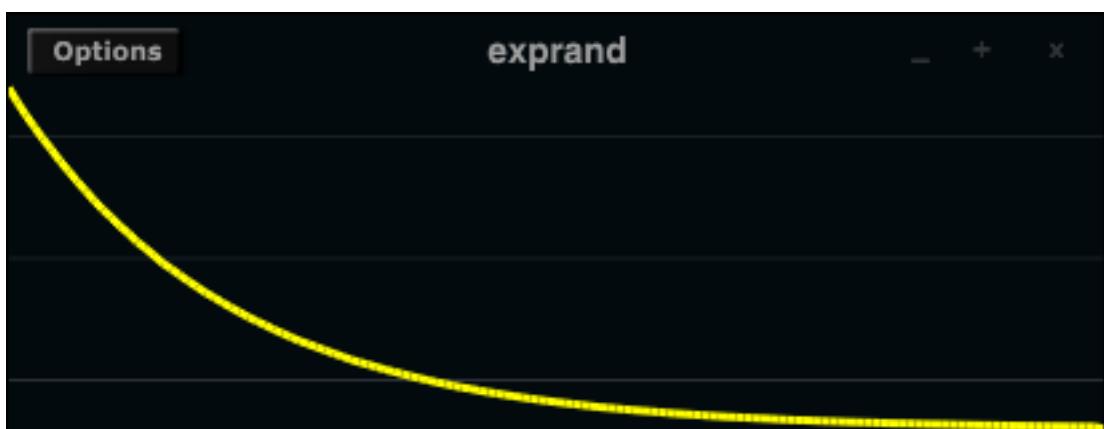
Csound has a range of opcodes and GEN routine for the creation of various random functions and distributions. Perhaps the simplest of these is [random](#) which simply generates a random value within user defined minimum and maximum limit and at i-time, k-rate or a-rate according to the variable type of its output:

```

ires random imin, imax
kres random kmin, kmax
ares random kmin, kmax

```

Values are generated according to a uniform random distribution, meaning that any value within the limits has equal chance of occurrence. Non-uniform distributions in which certain values have greater chance of occurrence over others are often more useful and musical. For these purposes, Csound includes the [betarand](#), [bexprand](#), [cauchy](#), [exprand](#), [gauss](#), [linrand](#), [pcauchy](#), [poisson](#), [trirand](#), [unirand](#) and [weibull](#) random number generator opcodes. The distributions generated by several of these opcodes are illustrated below.





In addition to these so called *x-class noise generators* Csound provides random function generators, providing values that change over time at various ways. Remember that most of these random generators will need to have `seed` set to zero if the user wants to get always different random values.

`randomh` generates new random numbers at a user defined rate. The previous value is held until a new value is generated, and then the output immediately assumes that value.

The instruction:

```
kmin    =      -1
kmax    =       1
kfreq   =       2
kout    randomh  kmin,kmax,kfreq
```

will produce and output a random line which changes its value every half second between the minimum of -1 and the maximum of 1. Special care should be given to the fourth parameter *imode* which is by default 0, but can be set to 1, 2, or 3. For *imode*=0 and *imode*=1 the random lines will start at the minimum (here -1) and will hold this value until the first period has been finished. For *imode*=2 it will start at a value set by the user (by default 0), whereas for *imode*=3 it will start at a random value between minimum und maximum. This is a generation for five seconds:

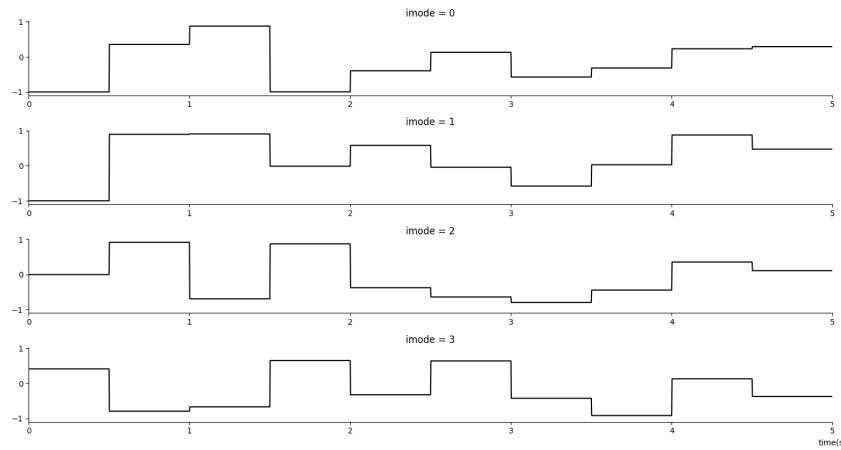


Figure 85.1: Opcode randomh with different values for imode

Usually we will use *imode*=3, as we want the random line to start immediately at a random value. The same options are valid for `randomi` which is an interpolating version of `randomh`. Rather than jump to new values when they are generated, `randomi` interpolates linearly to the new value, reaching it just as a new random value is generated. Now we see the difference between *imode*=0 and *imode*=1. The former remains one whole period on the minimum, and begins its first interpolation after it; the latter also starts on the minimum but begins interpolation immediately. Replacing `randomh` with `randomi` in the above code snippet would result in the following output:

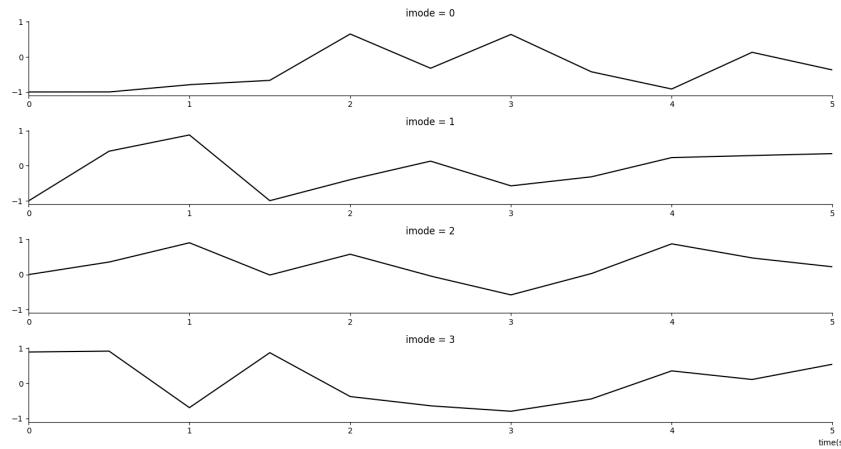
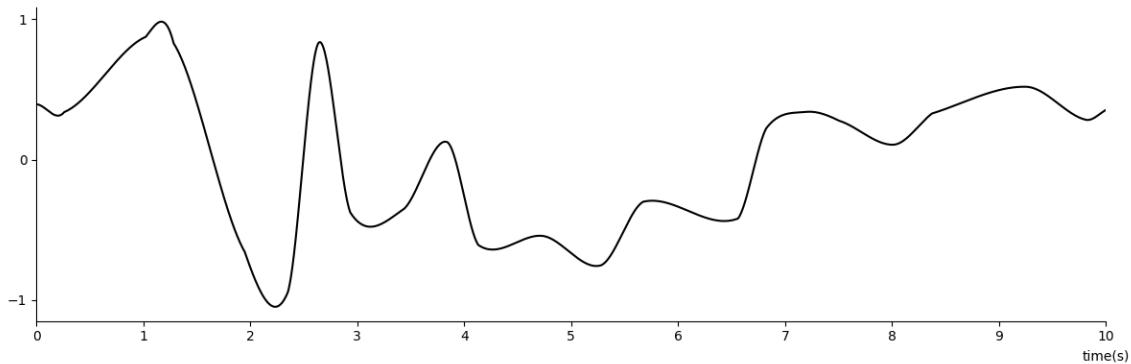


Figure 85.2: Opcode randomi with different values for imode

In practice `randomi`'s angular changes in direction as new random values are generated might be audible depending on the how it is used. `rspline` (or the simpler `jspline`) allows us to specify not just a single frequency but a minimum and a maximum frequency, and the resulting function is a smooth spline between the minimum and maximum values and these minimum and maximum frequencies. The following input:

```
kmin      =      -0.95
kmax      =       0.95
kminfrq   =        1
kmaxfrq   =        4
asig      rspline   kmin, kmax, kminfrq, kmaxfrq
```

would generate an output something like:



We need to be careful with what we do with rspline's output as it can exceed the limits set by *kmin* and *kmax*. Minimum and maximum values can be set conservatively or the *limit* opcode could be used to prevent out of range values that could cause problems.

The following example uses rspline to *humanise* a simple synthesiser. A short melody is played, first without any humanising and then with humanising. rspline random variation is added to the amplitude and pitch of each note in addition to an i-time random offset.

EXAMPLE 15D12_humanising.csd

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

giWave ftgen 0, 0, 2^10, 10, 1,0,1/4,0,1/16,0,1/64,0,1/256,0,1/1024

    instr 1 ; an instrument with no 'humanising'
inote =      p4
aEnv  linen  0.1,0.01,p3,0.01
aSig  poscil aEnv,cpsmidinn(inote),giWave
      outs   aSig,aSig
    endin

    instr 2 ; an instrument with 'humanising'
inote =      p4

; generate some i-time 'static' random parameters
iRndAmp random -3,3 ; amp. will be offset by a random number of decibels
iRndNte random -5,5 ; note will be offset by a random number of cents

; generate some k-rate random functions
kAmpWob rspline -1,1,1,10 ; amplitude 'wobble' (in decibels)
kNteWob rspline -5,5,0.3,10 ; note 'wobble' (in cents)

; calculate final note function (in CPS)
kcps =      cpsmidinn(inote+(iRndNte*0.01)+(kNteWob*0.01))

; amplitude envelope (randomisation of attack time)
aEnv  linen  0.1*ampdb(iRndAmp+kAmpWob),0.01+rnd(0.03),p3,0.01
aSig  poscil aEnv,kcps,giWave
```

```

    outs      aSig,aSig
  endin

</CsInstruments>
<CsScore>
t 0 80
\#define SCORE(i) \#
i $i 0 1   60
i . + 2.5 69
i . + 0.5 67
i . + 0.5 65
i . + 0.5 64
i . + 3   62
i . + 1   62
i . + 2.5 70
i . + 0.5 69
i . + 0.5 67
i . + 0.5 65
i . + 3   64 \#
$$SCORE(1) ; play melody without humanising
b 17
$$SCORE(2) ; play melody with humanising
e
</CsScore>
</CsoundSynthesizer>
;example by Iain McCurdy

```

The final example implements a simple algorithmic note generator. It makes use of GEN17 to generate histograms which define the probabilities of certain notes and certain rhythmic gaps occurring.

EXAMPLE 15D13_simple_algorithmic_note_generator.cs

```

<CsoundSynthesizer>
<CsOptions>
-odac -m0
</CsOptions>
<CsInstruments>
sr = 44100
ksmps = 32
nchnls = 1
0dbfs = 1

giNotes ftgen  0,0,-100,-17,0,48, 15,53, 30,55, 40,60, 50,63,
              60,65, 79,67, 85,70, 90,72, 96,75
giDurs  ftgen 0,0,-100,-17,0,2, 30,0.5, 75,1, 90,1.5

instr 1
kDur  init      0.5          ; initial rhythmic duration
kTrig metro      2/kDur       ; metronome freq. 2 times inverse of duration
kNdx  trandom    kTrig,0,1   ; create a random index upon each metro 'click'
kDur  table      kNdx,giDurs,1 ; read a note duration value
        schedkwhen kTrig,0,0,2,0,1 ; trigger a note!
        endin

instr 2
iNote table rnd(1),giNotes,1 ; read a random value from the function table
aEnv  linsegr   0, 0.005, 1, p3-0.105, 1, 0.1, 0 ; amplitude envelope
iPlk  random    0.1, 0.3 ; point at which to pluck the string
iDtn  random    -0.05, 0.05 ; random detune
aSig  wgpluck2  0.98, 0.2, cpsmidinn(iNote+iDtn), iPlk, 0.06

```

```
    out      aSig * aEnv
  endin
</CsInstruments>

<CsScore>
i 1 0    300 ; start 3 long notes close after one another
i 1 0.01 300
i 1 0.02 300
e
</CsScore>
</CsoundSynthesizer>
;example by Iain McCurdy
```


15 E. NEW FEATURES IN CSOUND 7

Csound 7 is perhaps the biggest step in Csound development ever, at least from the user's perspective. Full flexibility of naming variables, extended array features, for-loops and much more really brings a new experience of using Csound. We cover here some of these features with small examples; you may compare to [explanations in the Csound Reference Manual](#).

Variable Names

Variable names no longer have to start with i, k, a etc. to declare their data type implicitly. Instead any name can be used, and the type is then declared explicitly with a colon and the type abbreviation.

EXAMPLE 15E01_Explicit_Types.csd

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

instr Locals

    freq:i = 500
    amp:i = 0.2

    sine:a = poscil(amp,freq)
    outall(sine)

endin

instr Globals

    dry@global:a init 0
    dry += diskin:a("fox.wav") / 3
    dry += pinkish(.05)
```

```

    schedule(GlobalsReverb, 0, p3+3)

  endin

instr GlobalsReverb

wet:a = reverb2(dry, 2, .5)
outall(dry/5+wet/2)
dry = 0

  endin

</CsInstruments>
<CsScore>
i 1 0 2
i 2 3 3
</CsScore>
</CsoundSynthesizer>
;example by joachim heintz

```

Note that instrument names are now also variables. So instead of referring to an instrument as name via a string, we can now refer to it as variable:

schedule("GlobalsReverb", 0, p3+3) is still ok but
 schedule(GlobalsReverb, 0, p3+3 is the new way to go.

Arrays

The equal sign can now be used to declare an array. An array variable uses the same explicit types as discussed above, for instance i[] for a numeric array at i-rate. There is a nice shortcut for the genarray opcode: [1 ... 5] is resolved to [1,2,3,4,5].

EXAMPLE 15E02_Arrays.csd

```

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

instr ArrayDef

freq:i[] = [500,600,700]
vals:k[] = [3,5,7,8]
ser:i[] = [1 ... 10] //shortcut
files:S[] = ["bla.wav","blu.flac"]
sound:a[] = diskin("fox.wav")
printarray(ser)
turnoff

  endin

```

```

instr ArrayExample

// two random k-rate envelopes in an array
env:k[] = [randomi:k(-20,0,15,3),randomi:k(-20,0,15,3)]

// two mono sound streams
a1 = diskin:a("fox.wav",randomi:k(.7,1.1,1,3),0,1)
a2 = diskin:a("ClassGuitMono.wav",randomi:k(.8,1.3,1,3),0,1)

// apply envelopes and put into array
audio:a[] = [a1*ampdb(env[0]), a2*ampdb(env[1])]

// sometimes swap the audio channels (clicks may occur)
if (randomh:k(0,2,1,3) > 1) then
    audio = [audio[1], audio[0]]
endif

// output
out(audio)

endin

</CsInstruments>
<CsScore>
i 1 0 1
i 2 0 10
</CsScore>
</CsoundSynthesizer>
;example by joachim heintz

```

For-Loops

For-loops have been implemented in two variants.

The first one uses the syntax `for var in [array] do` where `var` uses the own given type to determine whether the loop should be executed at i-time or at k-time. If the type of `var` is not given, it uses the type of the array.

The second variant of for-loop has a counter as second variable: `for var,i in [array] do`. This counter starts from zero and counts the number of loops.

EXAMPLE 15E03_For-loops.csd

```

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

instr K_loop_due_to_var

```

```

// num is k so the loop runs at k-rate
num:k init 0
for num in [1,2,3,5,8] do
    printk(0,num)
od
turnoff

endin

instr I_Loop_Simple

// num is not specified: i-rate because of the i-rate array
for num in [1,2,3,5,8] do
    print(num)
od

endin

instr I_Loop_With_Counter

// the same but with loop counter
for num,i in [1,2,3,5,8] do
    print(num,i)
od

endin

instr Short

strt:i init 0
for freq in [200 ... 2000, 100] do // = [200, 300, ..., 2000]
    schedule(PlaySine,strt,5-strt,freq)
    strt += 1/8
od

endin

instr PlaySine

env:a = transeg(.1,p3,-3,0)
outall(poscila:(env,p4))

endin

</CsInstruments>
<CsScore>
i 1 0 1
i 2 0.1 0
i 3 0.2 0
i 4 0.3 1
</CsScore>
</CsoundSynthesizer>
;example by joachim heintz

```

New UDO Syntax and Pass-by-reference

User Defined Opcodes now follow the syntax `opcode name(inargs):(outargs)`. When this syntax is used, the arguments are called by reference, and not by copy. This gives new possibilities,

but also allows destructive changes as the example shows.

EXAMPLE 15E04_UDOs.csd

```

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

// new UDO definition which implicitly changes the input array

opcode Destructive(arr:i[])::()
  for v,i in arr do
    arr[i] = -v
  od
endop

instr TestDestructive

myarr:i[] = [1,2,3,4,5]
printarray(myarr)
Destructive(myarr)
printarray(myarr)
turnoff
endin

// old UDO definition does not change the input array as it is copied

opcode NonDestructive, 0, i[]
  arr:i[] xin
  for v,i in arr do
    arr[i] = -v
  od
endop

instr TestNonDestructive

myarr:i[] = [1,2,3,4,5]
printarray(myarr)
NonDestructive(myarr)
printarray(myarr)
turnoff
endin

// routing an array of audio signals to hardware outputs
// /practical use case is for multichannel not stereo)

opcode Route(asigs:a[],hwouts:k[]):()
  for h,i in hwouts do
    outch(h,asigs[i])
  od
endop

instr ArrayRouting

// array of hardware output channels (starting at 1)

```

```
hw_out_chnls:k[] = [2,1]
// array of audio signals
audio_sigs:a[] = [poscil:a(.2,500),poscil:a(.2,400)]
// output
Route(audio_sigs,hw_out_chnls)

endin

</CsInstruments>
<CsScore>
i 1 0 1
i 2 0.1 1
i 3 0.2 3
</CsScore>
</CsoundSynthesizer>
;example by joachim heintz
```