

Synthesis Basics

Digital sound synthesis can be performed in a variety of ways, with:

- Dedicated synthesis hardware, such as keyboard or rack synthesisers and samplers.
- Software synthesis programs, usually designed to mimic existing ‘hard’ (ie. dedicated) synthesisers.
- Computer Music Programming Languages, such as *csound*, PureData, Max/MSP, SuperCollider, etc..

Computer Music Languages

These are computer music systems for sound synthesis and processing with programming-language features.

Characteristics :

- Flexibility
- Generality
- Control
- Programmability
- Comprehensiveness

A Brief History

1957: **MUSIC I**, the first computer program for sound synthesis by **Max Mathews**, written for the IBM704 computer.

1958-1962: A series of **MUSIC N** programs will follow, culminating in **MUSIC IV** (1962), the model for most of the *Computer Music Languages* since the early 60s. This was written for the IBM 7094

1968: **MUSIC V** is developed, written in the FORTRAN language. It becomes the most important computer music language for a decade, mostly because of its portability.

Csound

Csound was created by Barry Vercoe in the 80s as a C-language coded and updated version of **MUSIC11**, which in turn was a version of **MUSIC360**. This was based on Max Mathew's **MUSIC4** of 1962.

Csound is available for most of the modern computer systems and is probably the most used computer music language today.

All computer systems in our lab have their versions of csound. Although the interfaces for Linux, Mac and PC are somewhat different, the options and working logic are similar.

Csound principles

Csound is also known as a *sound compiler* or *sound renderer*, something that describes its working principle: ***transforming textual declarations into sound.***

So the **basic input** to csound is text, and it comes in two forms:

(1) How the sound will be created => “instruments”
(the **orchestra** file)

(2) What is going to be played => “notes” (and function tables)
(the **score** file)

These two elements can either be stored in two separate files (.orc and .sco), or as a single integrated file (.csd)

The orchestra and score files

These files are normal text (ASCII) files, similar to any file created by *Notepad* and *Emacs* (Win) or *Edit* (DOS).

They contain textual declarations in the csound *language* and are the basic input to Csound.

These files can be named anything, but it is common to give them the extensions **.orc** (orchestra) and **.sco** (score) as in *foo.orc* and *foo.sco*, for easy identification.

The two files can be combined into one, an integrated orchestra/score file, which uses html-style tags. Separate files can also be included in orchestras or scores using `#include` directives.

Output

The program output is basically *sound*, but in what form?

There are two:

(1) *Soundfile*: the program can generate soundfiles of several types, including *RIFF-Wave* and *AIFF*. This is the non-realtime (deferred-time) operation mode.

(2) *Realtime*: csound can also send sound directly to the DAC (and read from the ADC). This option is limited by the capabilities of the computer used and complexity of the synthesis algorithms, sampling rate and number of audio channels required.

How does it work?

This is a simple *working loop* for Csound:

(1) Using a text editor, Prepare your **orchestra** and your **score**, calling them, say, **bar.orc** and **bar.sco**

(2) Compile the sound (using the command **csound**)

(a) *either* in realtime to the soundcard:

csound bar.orc bar.sco -o dac

(b) *or* to a soundfile:

csound bar.orc bar.sco -o sound.wav

(3) listen to the result,
possibly go back to (1) for alterations to the orc/sco

A Few Important Details

Some stuff you will *need* to **remember**:

- *Before you start*, you will have to open a **MS-DOS command window**, so you can type the command in.
- The different words in the command line are separated by spaces: ‘**csound**’, ‘**bar.orc**’, ‘**bar.sco**’, etc...
- Anything starting with a minus/dash sign (-) is called an *option*: ‘-o’, ‘-W’, etc. .They control the ways csound behaves.
- For simplicity, the *input text files* need to be in the *current directory*.

What does the command-line mean?

```
csound bar.orc bar.sco -o dac
```

The only two *required arguments* or *parameters* to the command are the basic inputs to the program: the *text files* for the **orchestra** and **score**: **bar.orc bar.sco**

A number of *optional arguments* can follow (for a complete list list see *The Manual*), but they are not strictly required. The only one used above is **-o dac**.

-o is used to specify the **output**. If not present, a soundfile called *test.wav* will be created (in the Riff-WAVE format). This is created in the defined soundfile directory (in our PCs this is **d:\audio**)

How do we interpret the options in the other command line?

```
csound bar.orc bar.sco -o sound.wav
```

csound to creates a RIFF-Wave format file (the default)
-o sound.wav tells csound to name the output file *sound.wav*

Where is the output soundfile?

Csound will write to the defined *soundfile* directory, defined by the environment variable SFDIR.

In our PCs this is **d:\audio**. Other relevant environment variables are SSDIR and SADIR (sound-sample directory and sound-analysis directory). They are also set to **d:\audio**.

More on Csound output

We have seen that the `-o` flag (‘option’) specifies the output. If the words ‘dac’ or ‘devaudio’ are used, the output goes to the soundcard directly.

If any other word is used, it will be taken to be a soundfile name. Output formats supported are defined by the following flags:

[-W --wave] RIFF-Wave (default, no need for flag)

[-A --aiff] AIFF

[-J or --ircam] IRCAM format

The default output type is defined by the environment variable `SFOUTYP` and is set to `WAV`.

Csound is non-graphical

It is important to note that the interface to csound is **not graphical** (although there are GUI versions, which we will not use in this course).

The simplest and most complete way of learning how to use csound requires that we become familiar with its command-line interface. There is no other way around it.

Csound also outputs some text on to the screen, which gives very useful information about the synthesis process. In due time, we will learn how to read that information

Csound orchestra basics

An orchestra file will typically contain two elements:

1. *One header*: with information about the system state:

sampling rate (sr)

control rate (kr)

control samples (DSP vectorsize) (ksmps)

number of channels (nchnls)

2. *One or more instrument definition sections*: where the details of sound production are defined, between the words *instr N* and *endin*.

Comments are accepted. They start with a semicolon (;) and go to the *end of the line*. C-style comments are also accepted, these are written between /* and */. They can be *multi-line*.

Opcodes and Variables

Instrument code will contain two basic syntax elements:

(1) **opcodes**: operator codes, defining an ‘operator’ used in a code line. This is generally an unit generator (oscillator, envelope, etc.), thus the terms are generally equivalent.

Ex.: `osci1` defines a *truncating oscillator*

(2) **variables**: variables are ‘memory slots’, where a numerical value can be stored. These are given names so that they can be used anywhere in the instrument.

Ex.: `afdb` is a variable named *afdb*

`kenv` is another variable named *kenv*

Examples

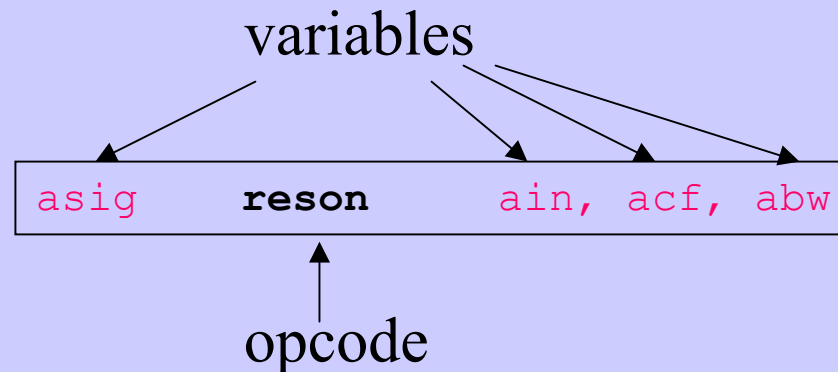
```
; header (comments start with a ";")
sr=44100      ; sampling rate (samples/sec)
kr=100        ; control rate (blocks/sec)
ksmps= 441    ; number of samples in one block
nchnls=1      ; number of channels

instr 1      ; instrument 1 code

                /* C-style comments are also accepted.
                  In this section you will put your
                  opcodes and variables */

endin        ; end instrument 1 code
```

an instrument code line:



Csound score basics

Scores define **control** information to run instruments. Comments are also accepted, as in the orchestra. The score is composed of:

1. ‘Note’ lists: statements starting an action on an instrument
I-statements (start with an ‘i’)
2. Function tables: statements defining tabulated function tables
F-statements (start with an ‘f’)
3. Other less used statements: section definition, tempo definition, etc...:
S-, T-statements, etc.

P-fields

Score statements are based on a series of parameters fields or **p-fields**. These are numbered from the left from 1: **p1**, **p2**, **p3**, **p4**... **pN** and they appear after the letter defining the statement (e.g. 'i').

Examples:

i1 0 10 3000 40 5.2

This is read as: **i-statement**, six p-fields,
 $p1 = 1$, $p2 = 0$, $p3 = 10$, $p4 = 3000$, $p5 = 40$, $p6 = 5.2$

f1 0 1024 10 1

Read as: **f-statement**, five p-fields,
 $p1 = 1$, $p2 = 0$, $p3 = 1024$, $p4 = 10$, $p5 = 1$

I-statements

The **i-statements**, all score lines starting with **i**, are a list of values used to “play a note” in an instrument (ie. activate it). Here’s one of them:

```
i1 0 10
```

The first **3 p-fields** have a **fixed** meaning:

p1 is instrument number (**instr 1**, in the example),

p2 is the start time (in secs, **0**);

p3 is the duration (also in secs, **10**).

All **other p-fields** (if present) are assigned *any meaning* by the instrument definition. They can be frequency, amplitude, bandwidth, attack or *whatever parameter* the instrument requires.

An instrument example

```
sr = 44100      ; sampling rate (samples/sec)
kr = 4410       ; control rate (sample blocks/sec)
ksmps = 10     ; samples in a block (sr/kr)
nchnls = 1     ; channels

instr 1
ifrequency = p5    ; frequency is p-field 5
iamplitude = p4    ; amplitude is p-field 4
iattack = p6      ; attack is p-field 6
idecay = p7       ; decay is p-field 7

/* an envelope generator, defined by the opcode linen
   note that p-field 3 (duration) is used here,
   to provide the envelope duration */
kenvelope linen iamplitude, iattack, p3, idecay

/* an oscillator, defined by the opcode oscil:
   this generates a pitched sound with
   frequency determined by the variable ifrequency and
   amplitude controlled by the variable kfrequency */
asignal oscil kenvelope, ifrequency, 1

/* the instrument output, defined by the opcode out
   which is taking the audio from the variable asignal */
out asignal

endin
```

A score example

```
f1 0 1024 10 1 /* this defines a function table that
                 holds a single cycle of a sine wave
                 which will be used by the oscillator */

i1 0 2 16000 440 0.01 0.05 /* makes instr 1 active
                             for 2 secs, starting at 0
                             secs. The peak amplitude is
                             16000; the frequency, 440;
                             the attack lasts for 0.01 secs;
                             the decay, 0.05 secs */

i1 1 5 4000 500 0.1 0.9 ; similarly to above
i1 1 3 6000 230 0.03 0.2 ; these two sounds are produced
                           ; at the same time (1 sec)
                           ; but one lasts longer than
                           ; the other
```

Producing the sound

This is how you can hear the synthesised sound:

- Save the two text files in *first.orc* and *first.sco*
- Open the command-line window and make sure that you are in the same directory as the two csound files (if not, use the command **cd** to change directories!)
- Run csound: **csound first.orc first.sco -o first.wav**
- Open the soundfile *first.wav* in Soundforge (or any other soundfile editor).

Csound messages

The following will be printed to the screen:

```
Csound Version 4.19 (Apr 16 2002)
graphics not supported on this terminal, ascii substituted
orch now loaded
audio buffered in 16384 sample-frame blocks
writing 32768-byte blks of shorts to d:\audio\first.wav
(WAV)
```

This tells us the version of csound and other messages, of which the most important is where the audio will be written:

```
writing 32768-byte blks of shorts to d:\audio\first.wav
```

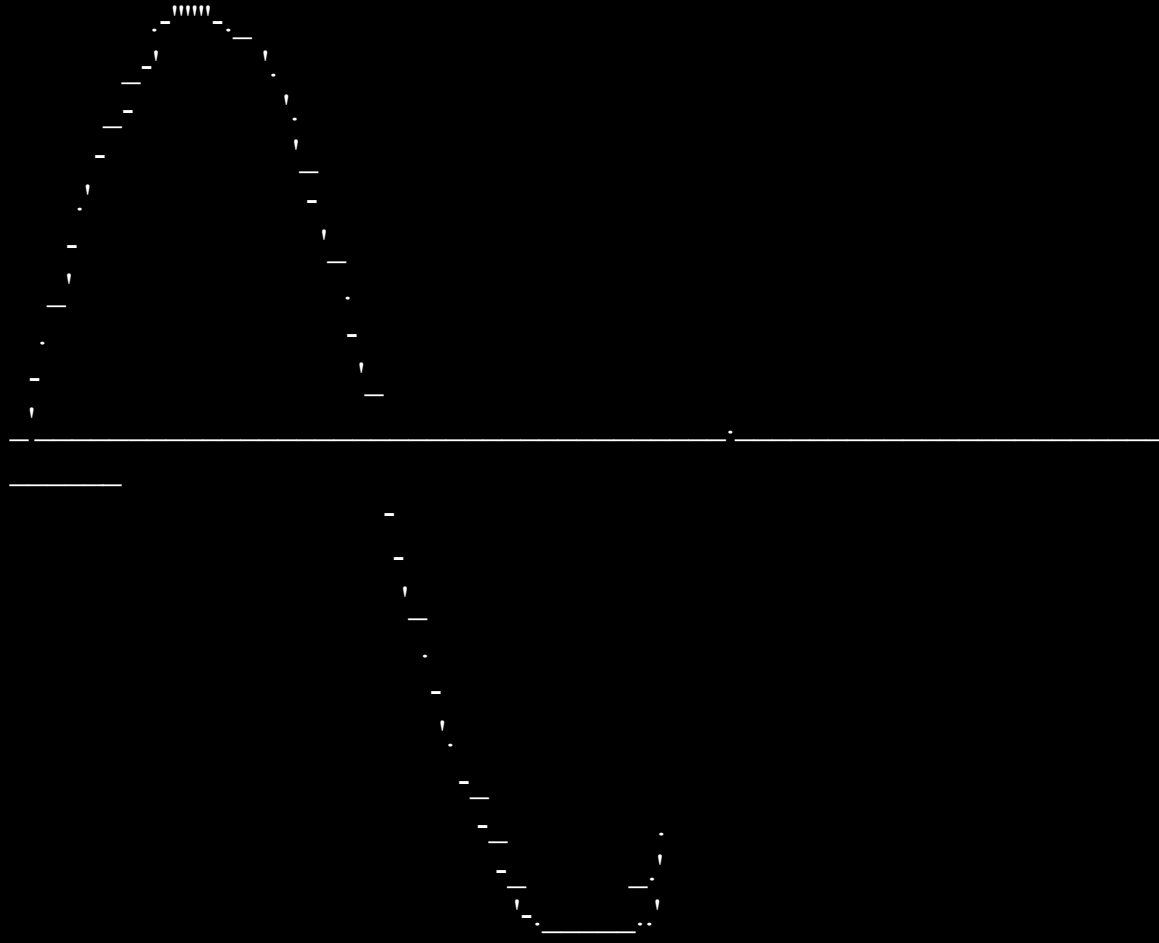
‘shorts’ mean 16-bit samples and the destination is the file we chose.

The function Table

SECTION 1:

f_{table} 1:

f_{table} 1: 1024 points, scalemax 1.000



csound prints an ASCII representation of the function table defined in the score.

We can clearly see the single-cycle sine wave.

The 'Notes'

```
new alloc for instr 1:  
B 0.000 .. 1.000 T 1.000 TT 1.000 M: 16000.0  
new alloc for instr 1:  
new alloc for instr 1:  
B 1.000 .. 2.000 T 2.000 TT 2.000 M: 25817.2  
B 2.000 .. 4.000 T 4.000 TT 4.000 M: 9998.3  
B 4.000 .. 6.000 T 6.000 TT 6.000 M: 4000.0  
end of score.          overall amps: 25817.2  
          overall samples out of range: 0  
0 errors in performance  
peak CH 1: 25817.173828 (written: 0.787878) at 85270  
17 32768-byte soundblks of shorts written to d:\audio/first.wav (WAV)
```

Csound prints information on the synthesis performance, '*new alloc for instr 1*' means that the instrument was activated at a certain time. The last values are peak amplitudes output by the program.

Performance messages

You can see that the print-out messages relate to the score i-statements. This is basically a report on the allocation of new instruments, as well as temporary max amplitude values.

```
new alloc for instr 1:  
B 0.000 .. 1.000 T 1.000 TT 1.000 M: 16000.0  
new alloc for instr 1:  
new alloc for instr 1:  
B 1.000 .. 2.000 T 2.000 TT 2.000 M: 25817.2  
B 2.000 .. 4.000 T 4.000 TT 4.000 M: 9998.3  
B 4.000 .. 6.000 T 6.000 TT 6.000 M: 4000.0
```

score i-statements

```
i1 0 2 16000 440 0.01 0.05  
  
i1 1 5 4000 500 0.1 0.9  
i1 1 3 6000 230 0.03 0.2
```

When new instruments are allocated in memory to perform a sound, a message is displayed. In certain cases, they do not need to be allocated, but simply use free space that is already allocated. Here there is a need to allocate space three times, as at one point three copies of instrument 1 are active.

Samples out-of-range messages

One common error message displayed by csound is '*samples out-of-range*'.

This message simply states that amplitudes higher than the maximum possible for the precision used have been produced. This results in clipping distortion of the signal and possible clicks and other undesirable effects.

In the usual 16-bit precision, the maximum absolute amplitude is 32767. If it is surpassed, the number of samples above it will be reported.

In most cases, reducing the amplitude is enough to resolve a clipping problem.

P-field warning messages

If an instrument requires a certain p-field that is not defined in an i-statement, the following message will be displayed:

```
WARNING: instr 1 pmax = 4 note pcnt = 3
```

This just tells us that instrument 1 uses up to **p4** and the i-statement only includes pfields up to **p3**.

Warning messages for a converse situation are similar

```
WARNING: instr 3 pmax = 5 note pcnt = 7
```

Instrument 3 here has only defined 5 pfields, but the score uses up to p7 in one of its i-statements.

Fatal Messages

Fatal messages are also printed to give the user an idea of what has gone wrong in the performance. They will be used when csound has failed to run properly.

Common errors are:

- (a) variables used before defined
- (b) unknown opcodes (probably resulting from typographic errors)
- (c) wrong number of opcode arguments
- (d) non-existing function tables (not defined)
- (e) inconsistent sr, kr, ksmps (ksmps has to be sr/kr and integral)

Summary: key concepts

- *Music Programming Languages* can be used for digital audio synthesis; **csound** is one such system, a *sound compiler*.
- *csound* takes two text files: an orchestra and a score.
- *orchestras* contain instrument definitions.
- *scores* contain performance and control data.
- *csound* is non-graphical and runs in a command-line window.
- *csound* can generate audio to **soundfiles** or to the **dac (soundcard)**.
- *orchestras* contain a header plus instrument definition blocks.
- *scores* contain f- and i-statements (among others) with a list of parameter fields
 - in an *i-statement*, pfields-1 to 3 (p1,p2,p3) have fixed meaning, other p-fields are assignable to any parameter.
 - *f-statements* define function tables.