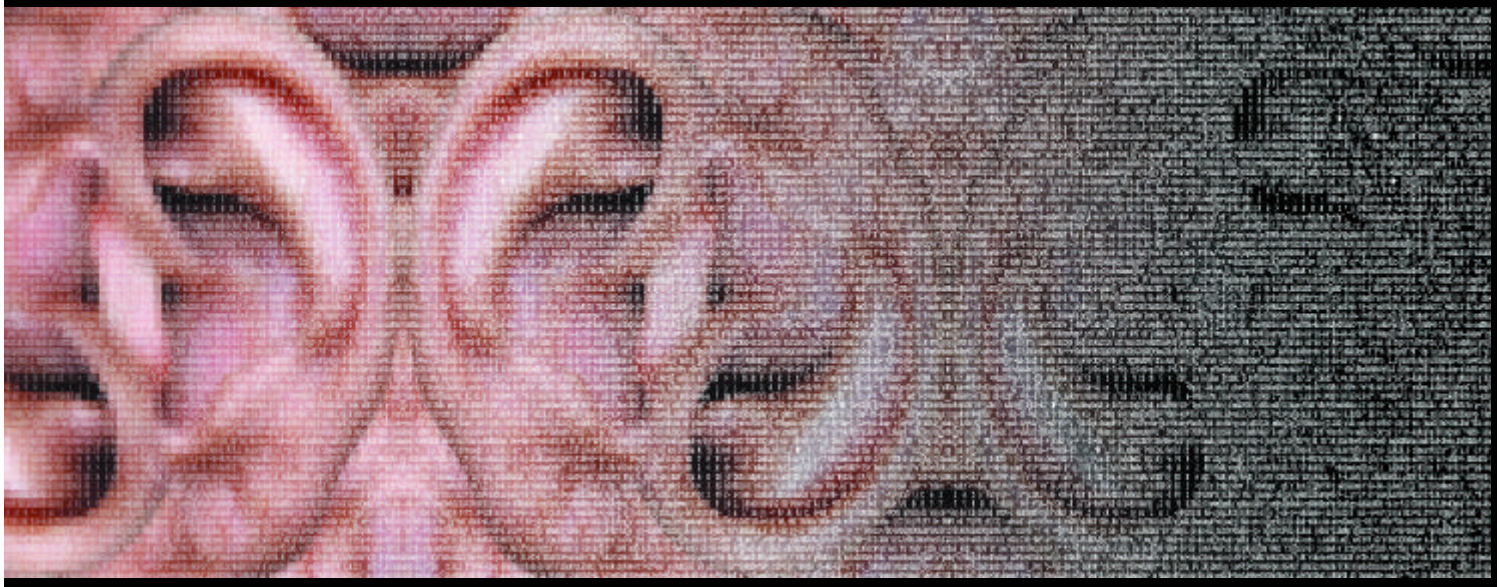
Riccardo Bianchini · Alessandro Cipriani

Virtual Sound

Sound Synthesis and Signal Processing - Theory and Practice with Csound



ConTempo

www.virtual-sound.com

This is a reduced evaluation copy of Virtual Sound.

For the full version go to:

www.contemponet.com/en

or write to

info@virtual-sound.com

Riccardo Bianchini • Alessandro Cipriani

Virtual Sound

Sound Synthesis and Signal Processing
Theory and Practice with Csound

Cinema per l'Orecchio series
Editorial director: Alessandro Cipriani

BIANCHINI R. - CIPRIANI A.
Virtual Sound / by Bianchini R., Cipriani A.
Includes bibliographical references and indexes
ISBN 88-900261-1-1
1. Computer Music-Instruction and study - 2. Computer composition

This edition is an English translation of "Il Suono Virtuale",

Chapters 1-9 by A.Cipriani
Chapters 10-17 and appendixes by R.Bianchini
The authors have revised and upgraded the text for this English edition

Translated from the Italian by Agostino Di Scipio

Cover and CD Design: Alba D'Urbano and Nicolas Reichelt

Manufactured in Italy
Contempo s.a.s., Rome
e-mail info@virtual-sound.com
info@contemponet.com
URL: www.virtual-sound.com
www.contemponet.com
fax +39-06.355.020.25

Contents

FOREWORD

Digital sound synthesis has for some time been at a remarkably high level of sophistication, flexibility and subtlety. This happy situation is due in large part to the remarkable efforts of Barry Vercoe, the author of Csound and its predecessors, MUSIC360 and MUSIC11. Now thanks to the universality of the C programming language, which guarantees portability between computers running on different operating systems, Csound has become the standard synthesis language wherever there is computer music. The flexibility of the language is already evident in this book: it can be studied by both PC oriented musicians as well as by those using Macintosh.

Of even greater value, the composer who learns Csound on his home computer is fully prepared to accept invitations to work elsewhere, for example in a studio equipped with multi-track recording facilities and expensive external sound processors, all built around UNIX machines which run their version of Csound.

Here, then, is the importance of this book: to learn Csound, especially if you have to do it on your own, you need someone who can explain the hows and whys of this language which offers so many possibilities for digital sound synthesis. The student finds himself here in the capable hands of two superb teachers, Riccardo Bianchini and Alessandro Cipriani. The book is infused with their considerable teaching experience in helping musicians with little or no knowledge of informatics to overcome the initial problems and confusions in order to get started on the road towards a full understanding and mastery of computer music. All the basic techniques of Csound are explained using specific sound synthesis models - additive synthesis, modulation synthesis, the use of filters and delay line effects (reverberation, echo, chorus effects), dynamic control over timbral evolution, interfaces with external sound sources (sampling), etc. - all of which allow the student to build his own experiments and more refined variants using the examples as starting points. In short, the approach here is completely hands-on.

Furthermore, each chapter offers suggestions for more advanced use; that is to say, the book is not just for beginners, but is designed to accompany the musician during various learning phases, from the beginning all the way up to the state of the art. Bianchini and Cipriani have obviously absorbed and worked with a vast range of digital synthesis techniques developed by many specialists in the field, and they provide clear explanations and suggestions for their use and adaptations, always in terms of concise Csound programming examples. For example, we find here a clear description of the famous Karplus-Strong plucked string algorithm, with precise indications about how to modify it. These modifications suggest yet other possibilities for experimentation with yet other potentially fascinating alterations. After following the authors' lessons, the

student will be in a position to work with the newest developments in the field, published regularly in such magazines as “Computer Music Journal”, “Interface”, etc.

An original and valuable idea in this book is the insertion of a few pages called “Extensions” following many of the chapters. The musician who has arrived at a reasonably good level of understanding of the main ideas will find here the technical information that allows him to develop original synthesis ideas. An especially pertinent such “Extension” deals with the issue of “complex events” - the concept that one can construct a Csound instrument for the synthesis of several sound objects where the form, synchronization and the character of each single object are under the control of just a few score parameters. Once such an instrument has been constructed, the composer can conceive of his music in terms of complex large scale gestures rather than always composing at the “note for note” level.

And finally, the book concludes with a selection of writings by other experienced Csound users whose unique applications, fully described here, are extremely useful both for immediate execution and, more importantly, as suggestions to the composer for his own personal developments.

So if you are a beginner or even if you already have a good bit of experience with digital sound synthesis, Bianchini and Cipriani offer here a complete manual for understanding Csound at many levels. And since Csound is continually evolving thanks to the work of a small international group of musicians/programmers who are dedicated to upgrading the language with new methods and with specific adaptations of the latest technical discoveries, the Csound user will find himself on the leading edge of computer music.

Welcome to the Csound universe.

James Dashow

CSOUND : WHAT IS IT?

Csound is a digital sound synthesis program created at MIT (*Massachusetts Institute of Technology*) by Barry Vercoe. People from all over the world contribute to develop it further. Indeed, Csound is public domain software: everybody can use it, modify it and contribute to further enhance it. Taking advantage of the power of currently popular processors (such as those inside your PowerMac or PC), Csound performs extremely fast, so fast as to run in "real-time". The program is written in C language, but you don't have to learn C programming to exploit its musical potential. It is sufficient to read this manual through: you will learn how to write Csound orchestras and scores, and how to create any possible sound you can imagine of, right off your personal computer. The important, at the beginning, is not to get scared by the unusual terminology: the learning process, then, will be more natural and quick.

DIRECT SOUND SYNTHESIS BY COMPUTER

What is direct sound synthesis? At the time when analog technology was the mainstream of electronic music, at some point you would have to use, say, nine oscillators and a low-pass filter, and all you could do about that was to go out and purchase nine oscillators and a low-pass. At another point, however, you would need nine low-passes and one oscillator. So you had to go buy eight more low-pass filters. With digital direct synthesis, instead, you can program the available hardware (i.e. your computer) to simulate either nine oscillators and one filter, or nine filters and one oscillator. The process is economic and flexible: you can make your computer simulate any sound generating unit and implement any sound synthesis technique, both old ones and not yet existing ones.

CSOUND : THE BEST SYNTHESIZER IN THE WORLD

The difference between Csound and commercially available audio and music software, is not only that it is free, but also that it doesn't get obsolete: its functionality is such that any new kind of sound synthesis or processing can be implemented. Also, and importantly, such a flexibility allows the musician to create a virtual sound machine that perfectly fits her/his needs. Using Csound you never have to stick to those 30 preset options offered by this or that keyboard or expander: sure, those 30 options are available in your hands right away, and yet they might not give you enough of what is necessary for the sound you have in your mind. On the contrary, Csound is open: as you learn to make competent use of it, you get closer and closer to the sound you're searching for, based on a deep awareness of the synthesis or processing methods involved. Which, by the way, allows you to master any other available tool as well, including commonly used synthesis programs.

This book, in fact, is not only for people already committed to musical research, but also for musicians eager to go deeper into the technological process of their work. What are the basic prerequisites to start reading? Well, nothing more than a basic knowledge of music and acoustics. And, sure, a good familiarity with your computer.

WHAT COMPUTER PRECISELY?

The basic requirement for any sound synthesis language to be successful, is that it is portable to as many types of computer platform as possible. It should not rely on a specific hardware. For that reason, several Csound versions exist, which make the language available on several computers, including PC, Mac, PowerMac, SGI, and others. The more powerful the processor, and the faster the Csound runtime operations. However, by no means the computer speed and power do affect the audio quality of the results you get. Csound doesn't require additional hardware to install on the computer, although you obviously need digital-to-analog converters in order to listen to the actual musical results. Any soundcard on the market will do the job. The soundcard quality will definitely affect the audio quality, especially as in terms of background noise and harmonic distortion. Still, the Csound output is actually stored to some sound file on the hard disk, and as such it is no less than CD quality, and possibly even better. You may want to install, instead of a soundcard, a digital connection to a DAT recorder, using the latter as a high-quality digital-to-analog converter (the only restriction being, then, that only standard sampling rates can be used, namely 32, 44.1 and 48 kHz).

Csound was born for "deferred-time" synthesis. However, as mentioned above, today it can work in real-time provided the computer processor is fast enough (and provided not too complicated sound-generating algorithms are used). To make this notion clear, suppose you want to create one minute of sound: if the synthesis takes more than one minute, such that you have to wait for a sound file to be created on hard disk and play it later, that is "deferred-time". If the synthesis takes one minute or less, such that you listen to the sound while it is being generated, that is "real-time". The time taken by the synthesis process depends on the complexity of the synthesis algorithms, and the latter usually depends, in turn, on the complexity in the generated sound.

WHY "VIRTUAL SOUND"?

The title of this book refers to that obscure moment, in electroacoustic composing, when sound is made "out of nothing", only based on ideas, formulas, desires, methods: prior to listening, it is *virtual*, it has not yet entered the physical world - an experience peculiar to work with digital sound synthesis.

This book was not conceived just as a help to learn Csound and its connection with the MIDI world, or with other tools. It was specially conceived as an introduction to the theory and practice of sound synthesis and processing. The aim was not to survey all of the possibilities Csound offers to you, but to exploit the language to create a bridge for those who are not familiar with direct sound synthesis, making it not too difficult for them to go across the river of knowledge on these matters.

This book comes from afar, rooted as it is in the Electronic Music classes taught by one of the authors in 1977 at the Conservatory of Pescara (Italy). Clearly, having so much time passed since then, and having the matters at issue so radically changed in these years, today very little remains of the original material. At the end of the 1970ies, people at University computing centers around the world were interfacing analog synthesizers and computers. The best-equipped electronic music centers had MOOG or EMS synths among their most advanced sound facilities. To commit oneself to computer music meant to go visit giant University computing facilities and learn about mysterious operating systems. One would then write some computer code and carefully go through a complicated, and quite slow debugging process: launch the program, wait for the results, listen to the sound, make decisions about what was to debug and/or refine, change the code, launch the program again, an so on. Today technological developments tend to hide from users the enormous research work that has been done in order to provide musicians with more handy and effective approaches, possibly based on personal computers. Such enterprise has been stimulated by the enthusiasm raised by the possibilities invented and investigated by computer music research. Csound welcomes everybody to be part of such an adventure, but relieves her/him of the impediments imposed by older technologies.

The authors acknowledge the contribution of Nyssim Lefford, Jon Christopher Nelson, Russell F. Pinkston, Emanuele Casale, Enzo Coco, Gabriel Maldonado, Luca Pavan, Giuseppe Emanuele Rapisarda, Fausto Sebastiani, for their proofreading and clever advice.

Happy Csounding!

1

CSOUND: HOW IT WORKS

1.1 ORCHESTRAS, SCORES, SOUND FILES

In order to generate sound, Csound requires that the user creates two text files. We call these respectively:

1) the **ORCHESTRA** (file extension: *.orc*)
2) the **SCORE** (file extension: *.sco*).[1]

These text files contain everything that is needed to describe the "virtual machine" we want to build and the operations we want it to perform. After saving our orchestra and score files, we use the Csound compiler to *execute* them. The compiler returns a *sound file*, (a file containing a binary code representation of the sound). Once the sound file has been generated, it can be auditioned by merely "playing" the sound file through the computer sound card. The card reads the sound file data and turns it into an electrical signal. By connecting the card's (analog) output to an amplifier, we can send that signal to loudspeakers. At this stage we say that the card performs a "digital-to-analog conversion." It allows us to listen to the sound that is represented digitally in the file.

[1] With the latest Csound versions, starting with release 3.50, we can include these two texts (orchestra and score) within one single file bearing the file extension .csd (see section 1.D.1)

Alternatively, it is possible for Csound to read a previously sampled (digitally recorded) sound and modify it in some way. Imagine we have just sampled a flute tone via a microphone plugged into the sound card input (in this instance the card performs an "analog-to-digital conversion"). With the appropriate orchestra and score files, we can specify how that flute tone is to be modified. Csound will then execute the orchestra and score commands and return a new sound file containing the flute tone duly transformed. Finally, we can listen to the modified flute sound by simply playing it through the computer's sound card.

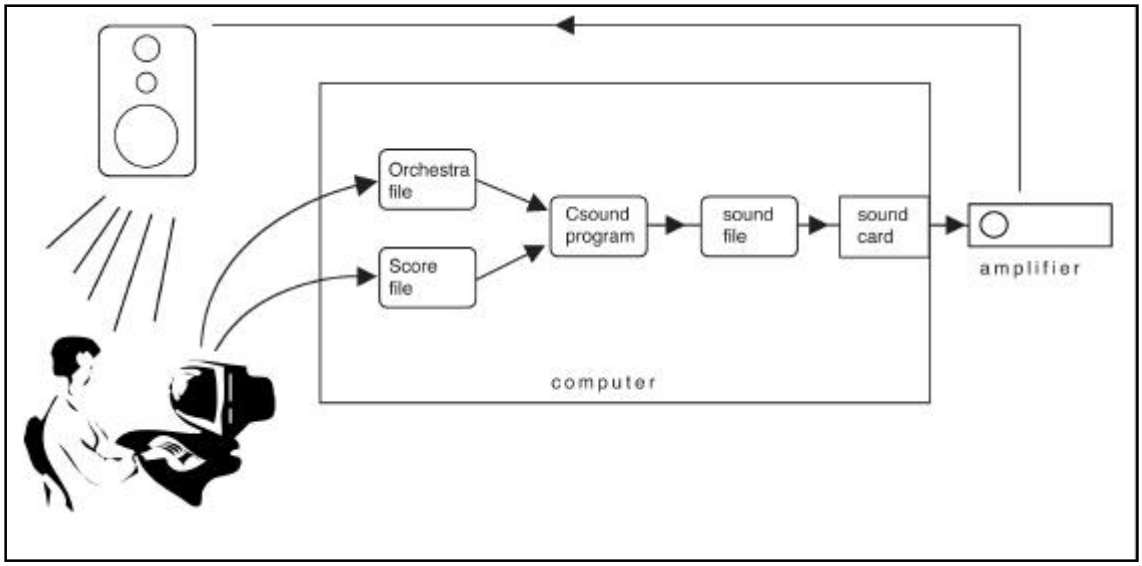


Fig. 1-1

1.2 HOW TO USE WCSHELL FOR WINDOWS [2]

How to run Csound from previously created orchestra and score files, and listen to the result

1. Launch *WCShell* by double-clicking on the *WCShell* icon
2. In the orchestra list, find the file "oscil.orc" and click on it.
3. In the score list, click on "oscil.sco"
4. Click on the *Csound* button to start the synthesis process
5. When the synthesis is complete, close the Csound session by pressing <return>
6. Click on the *PLAY* button to listen

[2] to use Csound without WCShell, see section 1.A.5.

How to create and execute a new orchestra and a new score

1. Launch *WCShell* by double-clicking on the *WCShell* icon
2. Choose *New Orc* from the *Orc* menu. This opens the orchestra editor
3. Type in the orchestra code, and save the file by choosing *Save as...* from the *File* menu
4. Close the orchestra editor by choosing *Exit* from the *File* menu
5. Choose *New Sco* from the *Sco* menu. This opens the score editor
6. Type in the score code, then save the file by choosing *Save as...* from the *File* menu
7. Close the score editor by choosing *Exit* from the *File* menu
8. Click on the two *Update* buttons. Check that the new files are shown in the file browsers
9. Click on the *Csound* button to start the synthesis process
10. When the synthesis is complete, close the Csound session by pressing <return>
11. Click on the *PLAY* button to listen
12. To modify the orchestra, choose *Edit Orc* from the *Orc* menu
13. To modify the score, choose *Edit Sco* from the *Sco* menu

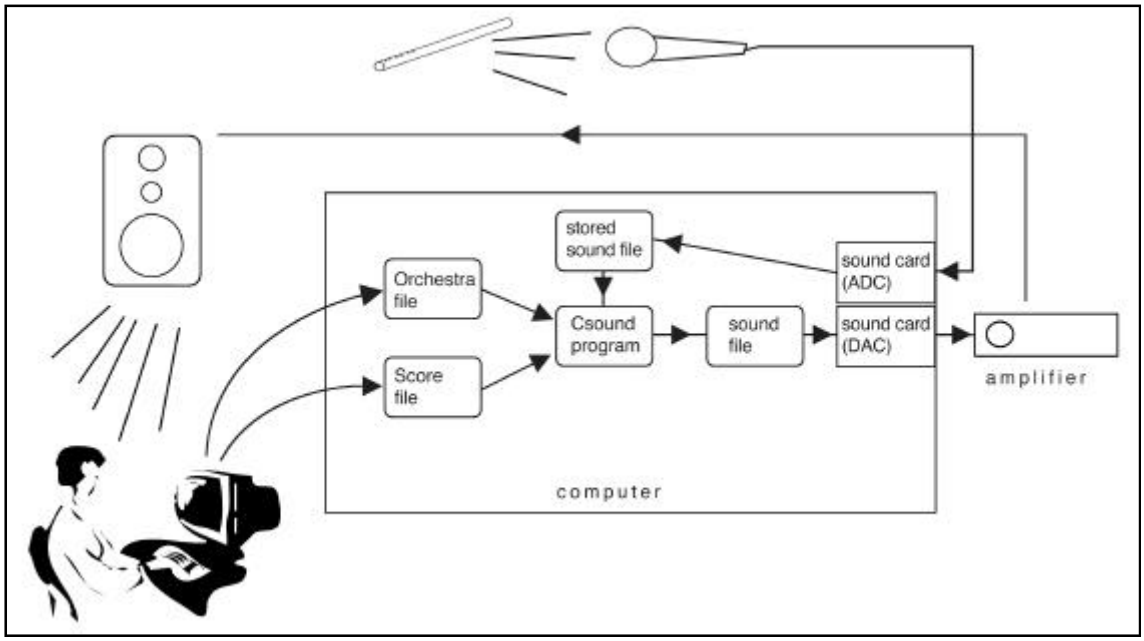


Fig. 1-2

At any time during this process, you may reference either Appendix 1 in this book or the on-line WCShell Help. To start the WCShell Help choose *Help* in the *File* menu or press the F11 key. Please refer to Appendix 1 to learn about WCShell installation and program operation.

1.3 HOW TO USE CSOUND WITH A POWER MAC

How to run Csound from previously created orchestra and score files

1. Find the folder where the Csound program files (*Csound* and *Perf*) are located, double-click on the Csound icon. This starts a graphical interface window.
2. Click the *Select* button next to the orchestra icon. A dialog box will appear where you can browse through folders looking for the desired files. Locate the "oscil.orc" file and double-click on it. The file name will be automatically appear in the orchestra file window, and the "oscil.sco" file name will automatically appear in the score file window.
3. Click on the *Render* button. A message appears to indicate that *perfing* is now running.
4. When *perfing* is complete, a "close" message appears, together with a message box with the text "0 errors in performance". You can listen to the sound by clicking on the *play* arrow (similar to a tape recorder). The play arrow is located on the left side of the message box. You can click on the arrow and listen to the output sound file as many times as you wish. To quit, just click the *Close* button.

How to create and execute a new orchestra and a new score

1. Select *General Preferences* from the *Preferences* menu and select the text editor of your choice (Simple Text is a good start).
2. Double-click on the orchestra file name in the graphical interface. This opens your orchestra file, which can be edited and saved with a different name. You can do the same with the score file. The files that you create and save in the editor must have the extensions ".orc", for "orchestra", or ".sco" for "score".
3. As a test, double-click on "oscil.sco" to open a simple Csound score. The bottom line will read "i1 4 2".
4. Replace "2" (which refers to the duration in seconds) with "10". That creates a tone 10 seconds in duration.
5. Close the window. A dialog box prompts you to save the modified score file. Click *Save*.
6. Now click on *Render*, then listen to the new sound file just created.

How to manage the system folders

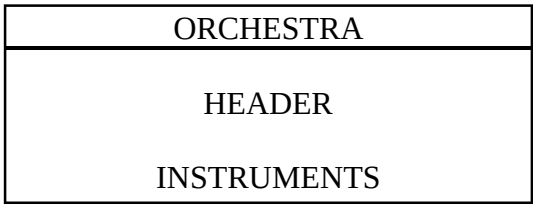
As long as you keep all Csound-readable files (*.orc*, *.sco*, *MIDI* files, analysis files, etc.) in the same folder, you don't need to change anything in the window. If you wish

to change the default folders, click on the *Default directories* button. Here you can choose any folder as the *Sound File Directory* (SFDIR). The one where Csound will save the sound files. Similarly, *Sound Sample Directory* (SSDIR) will designate the folder in which Csound will look for samples to read, and *Analysis Directory* (SADIR) will designate the folder in which Csound will save analysis files. If you don't type in new folder names, Csound searches for the *perf* program in the default directory.

1.4 HOW TO WRITE AN ORCHESTRA

Warning! This section may appear rather complicated. It introduces new terminology and a unique way of thinking about sound. Nonetheless, it is very important that you go through all sections and subsections that are left in this chapter. This information provides the basic foundation and syntax required in all Csound orchestras and scores. When using Csound in the future, you will simply apply the same set of operations described below..

An orchestra file always consists of two parts: header and instruments. [3]



HEADER
The header assigns some value to four fundamental variables that are shared by *all* instruments.

INSTRUMENTS
Instruments are the "virtual machines" constituting your orchestra. An orchestra can include one or more instruments.

HOW TO WRITE THE HEADER
The header determines the following:
sr sampling rate
kr control rate (see section 1.A.1)
ksmps sr/kr ratio (e.g. if sr = 48000 and kr = 4800, then ksmps = 10); it must be an integer
nchnls number of output channels (1=mono, 2=stereo, etc.)

[3] if the header is dropped, Csound will assume the following default values: *sr=44100, kr=4410, ksmps=10, nchnls=1.*

For all the instruments in your orchestra, the range of technical possibilities depends on the information defined in the header. For example, if you set nchnls = 2, then no instrument will be allowed to generate a quadraphonic output stream, only stereo.

A typical header:

```
sr      =  48000
kr      =  4800
ksmps   =  10
nchnls  =  1
```

HOW TO WRITE AN INSTRUMENT

While the header consists of only four assignments, an instrument is usually much more complicated. The level of complication is dependant upon the particular process it is expected to implement.

The first line in the instrument block must contain the instrument “id number”. This is designated by the statement *instr* (instrument) followed by any integer. The statement *endin* marks the end of an instrument. Thus, the general form of a Csound instrument is as follows:

```
          instr 1
…
…         (body of the instrument)
…
          endin
```

For example:

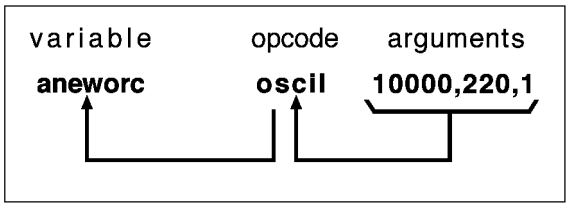
```
          instr   1
aneworc   oscil   10000, 220, 1
          out     aneworc
          endin
```

Here, *aneworc* is the name of a *variable*.

What is a variable?

A variable is like a small case, a drawer with a label (such as *aneworc*) where the result of some operation is temporarily stored. In the example, the variable *aneworc* will contain the result of the oscil operation code (opcode). The *oscil* opcode performs the

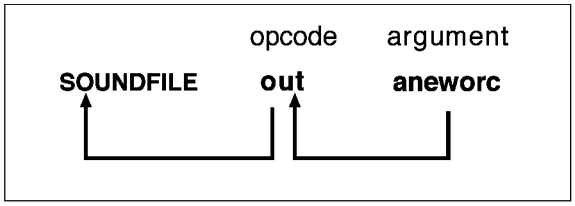
operation of an oscillator to which specific arguments are supplied: amplitude, frequency, and an identification number of a stored waveform function.



In the example above, the oscillator is given an **amplitude of 10000**, a **frequency of 220 Hz**, and a function that is referenced as **function table number 1** (as we'll see, the actual waveform of function 1 is created by some appropriate opcode in the score file). These values are passed to *oscil* to emulate an audio oscillator with determined characteristics of amplitude, frequency and waveform. The result of *oscil* is deposited in the *aneworc* variable.

In this example, the values of *aneworc* can be used to generate a sound with a frequency of 220 Hz. With this application it is good idea to use a sampling rate that allows for high resolution.

To do so, the variable storing the result of *oscil* must be given a name beginning by an *a* (then it will be an *audio* variable). In Csound, all audio variables are updated at the sampling rate specified in the header (e.g. 48000 times per second). Any variable name that begins with *a* (such as *a1*, *agreen*, *asquare*, *ataraxy*, etc.) can be used to define an audio rate, or a-rate variable. After some result gets stored into an audio variable, the variable name itself can be utilized as an argument to another opcode.



In the example above, *aneworc* is the only argument to the opcode *out*. The *out* opcode stores the value of the argument on the hard disk.[4]

The *endin* (end of instrument) statement marks the end of the instrument body. In the present example, it also marks the end of the orchestra.

In summary:

[4] When we launch Csound to run in real-time, *out* writes the result directly to the sound card buffer

```
          instr   1
aneworc   oscil   10000, 220, 1
          out     aneworc
          endin
```

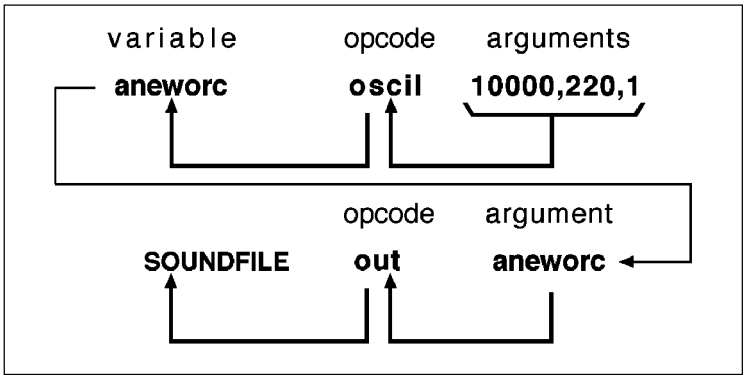
aneworc is an audio **variable**.

oscil is an *opcode* emulating an oscillator. It has 3 arguments: **amplitude**, **frequency** and **function number**.

out writes its argument (*aneworc*) to the sound file we want to create.

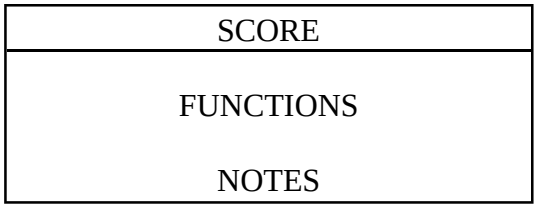
Typically, to ensure predictable performance, an opcode requires that values are given to its arguments (right side of the expression) and that the result of the operation gets stored as a variable (left side of the expression). Once a variable is defined, it can be used as an argument in another opcode, as is the case when *aneworc* is utilized as an input argument for the *out* opcode.

The *out* output is implicitly a hard disk sound file (or the sound card buffer): therefore, the result of *out* does not need to be stored as the value of a variable.



1.5 HOW TO WRITE A SCORE

Like a Csound orchestra file, the score file, usually has two parts: *functions* and *notes*.



FUNCTIONS

Function statements (f statements) are used to create particular waveforms. When using pre-determined waveforms (sampled sounds), the score file may not require the use of function statements and may be comprised solely of notes.

NOTES

Notes are always necessary (well, at least one note!). It is helpful to not think of notes in terms of the notation of music for an acoustic instrument such as a piano. Rather, notes within the context of Csound scores should be thought of more generally as sound events having any duration ranging from a fraction of a second to days, and as sonic events that may or may not have a precise pitch. The actual pitch depends on the particular instrument we implement in the orchestra and the particular functions we create in the score. Finally, notes can be entered in the score following some linear temporal sequence or any random order. Before the synthesis process starts, the program will sort out all of the score events according to their chronological action time

HOW TO CREATE A FUNCTION

In Csound it is possible to create any kind of function. As you recall, our first orchestra included a reference to some waveform function in the score file labeled as function #1.

```
(aneworc  oscil 10000,220,1)
```

The following f statement generates a sinusoidal waveform for use by this orchestra code:

```
f1  0  4096  10  1
```

where:

f1 determines the **function identification number** (1)

0 is the **action time** for the generation of this function (which determines at what time Csound will create function #1). If this had a value of 3, then the function would be created 3 seconds after the beginning of the score timing.

4096 is the **number of points** in the function. In other words, the waveform will consist in a table of values stored in an array of 4096 memory locations. In most cases, the number of points must be equal to a power of two (256, 512, 1024, 4096, etc.), but there are exceptions where it must be, instead, a power-two-plus-one (257, 513, 1025, 4097, etc.). The largest allowable size is 16777216 points (2^{20}).

10 utilizes one of many different methods for generating functions. We will refer to these methods as **GEN** routines, each has its own id number. With the number 10, we invoke the GEN10 function generating subroutine which is, in fact, a good routine to use if we want to create a sine wave. The different types of GEN routines (GEN01, GEN02, etc.), utilize various methods to create function tables

1 means that we expect GEN10 to create a waveform consisting of one single sinusoidal component. If instead we wrote:

```
f1   0   4096   10   1   1   1
```

we would expect three sinusoids in harmonic ratio (fundamental, second harmonic, third harmonic), all with amplitude level = 1.

At this point some may raise the question: “why do all this just to get a sine wave, when I can press a single key of an electronic keyboard and obtain a complex sound, at any pitch and of any duration?” If you want to modify the sound of your keyboard, you won’t be successful unless you have a deep knowledge of the sound-generating process implemented by that keyboard: Csound is an extraordinary tool which allows you to understand and use many existing sound synthesis techniques. Consequently, Csound helps you to better understand your synthesizer and, for that matter, any other electronic music system however complex it may be. Moreover, Csound allows you to do things that no sampler and no synthesizer will ever do. Patience and care are crucial for a composer who is really willing to commit herself/himself to competent exploitation of computer technologies, especially during the early stage of their efforts.

Let’s summarize how we create a sine wave in a Csound score:

Function number	action time	number of points in the function (or “table size”)	GEN type	amplitude of the fundamental
f1	**0**	**4096**	**10**	**1**

HOW TO WRITE NOTES

. Notes are generated through the use of instrument statements (i statements). An i statement turns on an instrument with the corresponding number in the orchestra file.

Csound i statements consist of *parameters* which are located at particular positions, called *p-fields* (*parameter-fields*). The first three p-fields of any note are the only ones

that must always contain values. These three p-fields must contain values for the following pre-defined uses:

p1 (first p-field): determines what instrument in the orchestra is to be played. For example, *i1* selects the instrument #1.

p2 (second p-field): action time. 0 means that the note starts immediately at the beginning of the piece; 3 means three seconds after the beginning, 3.555 means 3 seconds and 555 milliseconds, etc.

p3 (third p-field): duration of the note. The value 2 will generate a note for two seconds, .5 for half a second (you can omit the integer part when it equals zero). Non-american readers should note that a dot (".") must be used as the decimal separator, instead of a comma (","): hence "4.5" is correctly understood by Csound as "four and a half seconds", while "4,5" would not.

One can invent many other parameters (p4, p5, p6, etc.), each for a particular use. How many parameters and for what use, that only depends on the orchestra we ourselves create. In the following example, however, we use only the three pre-defined parameters. Note that it is possible to add comments to orchestra and score codes, e.g. to highlight for ourselves important details, by using semicolons. The Csound compiler ignores anything following a semicolon on any line of code. Lines containing only comments must begin with semicolons.

Example:

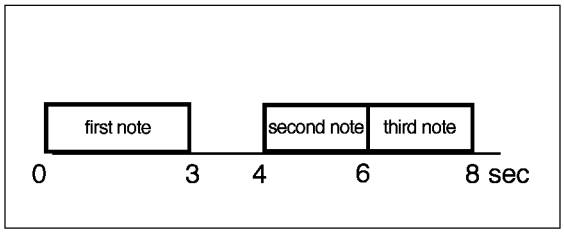
```
out a1                  ; this is a comment and
                        ; when running, Csound won't try to execute it
```

Score example:

```
f1   0   4096  10  1     ; function #1, action time, table size, GEN type, amp. of fundamental
i1   0   3              ; plays instrument 1 starting at the beginning of the piece, lasts 3 seconds
i1   4   2              ; plays instrument 1 starting at 4 seconds, lasts 2 seconds
i1   6   2              ; plays instrument 1, starts at 6 seconds, lasts 2 seconds
```

Notice that between the first and the second note there is a silent rest of 1 second. In Csound you don't have to explicitly declare rests and their durations, as they are automatically determined by the time between the end of one note and action time of another.

Careful! In the score file you separate parameters among them by simply inserting a blank (a single space-bar character or a tab character). Do not use commas or any other punctuation mark. In the orchestra file, commas are used to separate opcode input arguments, but in all other cases you must use blank spaces or tabs.



TIPS & TRICKS: it's good idea to usually space out orchestra and score code within a text file (variables, opcodes, arguments, p-fields) using tabs. This helps keep your code easy to read and scrutinize.

Let's summarize:

```
; instrument number    note action time    note duration
i1                     0                   3
i1                     4                   2
```

EXERCISE *Type in the following code and save it as a new orchestra file and a new score file. Insert your own comments explaining the code. Start reading the next section only when you have clearly understood header statements, instrument statements, functions and notes.*

```
; oscil.orc

          sr      =   44100
          kr      =   4410
          ksmps   =   10
          nchnls  =   1

          instr       1
aneworc   oscil       10000, 220, 1
          out         aneworc
          endin
```

```
; oscil.sco
f1    0    4096   10     1
i1    0    3
i1    4    2
```

Other paragraphs in this chapter:

1.A.2 CONSTANTS AND VARIABLES

1.A.3 THE CSOUND SYNTAX

1.A.4 THE CSOUND BUILDING BLOCKS

1.A.5 USING THE CSOUND COMMAND

1.B.1 A SINGLE FILE INCLUDING ORCHESTRA, SCORE AND FLAGS: THE *CSD* FORMAT

1.C.1 ATTACK AND RELEASE TRANSIENTS

1.D.1 BRIEF HISTORY OF SOUND SYNTHESIS LANGUAGES

LIST OF OPCODES INTRODUCED IN THIS CHAPTER

k1	**oscil**	**amplitude, frequency, function**
a1	**oscil**	**amplitude, frequency, function**
	out	**output_signal**
k1	**line**	**init_level, duration, end_level**
a1	**line**	**init_level, duration, end_level**
k1	**linseg**	**init_level, duration, next_level, duration, next_level, ...**
a1	**linseg**	**init_level, duration, next_level, duration, next_level, ...**
k1	**expon**	**init_level, duration, end_level**
a1	**expon**	**init_level, duration, end_level**
k1	**expseg**	**init_level, duration, next_level, duration, next_level, ...**
a1	**expseg**	**init_level, duration, next_level, duration, next_level, ...**
a1	**linen**	**amplitude, attack_time, duration, release_time**

2

ADDITIVE SYNTHESIS

2.1 CONSTANT SPECTRUM ADDITIVE SYNTHESIS

With additive synthesis we can create complex waveforms of any kind, by adding together simple wave components - usually sine waves (see 2.A.1 and 2.B.1).

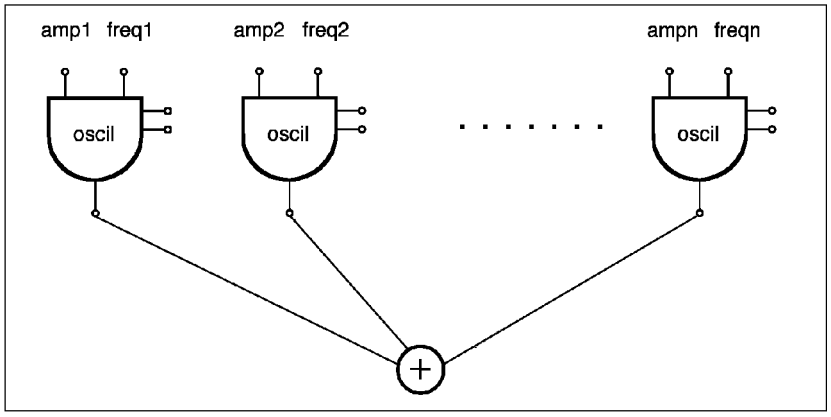


Fig. 2-1

The simplest method dictates a **harmonic relationship between sine waves**. As we have seen, in Csound this can be accomplished using GEN10. Indeed, in order for a complex waveform to be calculated with GEN10, we choose a series of components harmonically related to the fundamental frequency. We also determine the relative

amplitude for each component. It is possible to create a sawtooth-like wave, or a square wave, by modifying the amplitude for each component, accordingly (1 / the order number of the particular component within the harmonic series). The sonic difference between a sawtooth and a square wave results from the former being sum of a full series of harmonic frequencies, while the latter is the sum of the odd-numbered harmonics only:

Example of sawtooth wave (10 harmonics only)

```
f1  0  4096  10  10  5  3.3  2.5  2  1.6  1.4  1.25  1.1  1
```

Example of square wave (9 harmonics only)

```
f1  0  4096  10  10  0  3.3  0  2  0  1.4  0  1.1
```

By changing the relative weight of the harmonic components, it is possible to obtain a vast palette of timbres.

In Section 2.3 we focus on how to get rid of the inherent limitations of perfectly harmonic spectra. As spectrum cannot vary over the course of the note, a harmonic spectrum may not produce an interesting output. Still, starting with this simple model, we can achieve better results with little effort.

Other paragraphs in this chapter:

2.2 VARIABLE SPECTRUM ADDITIVE SYNTHESIS

2.3 PHASE AND *DC OFFSET*: GEN09 AND GEN19

2.4 COMPLEX OSCILLATORS: *BUZZ* AND *GBUZZ*

EXTENSIONS

2.A.1 WAVE SUMMATION

2.A.2 TIMBRE

2.B.1 ADDITIVE SYNTHESIS: HISTORICAL SKETCHES AND THEORY2.C.1 THE DIGITAL OSCILLATOR: HOW IT WORKS

LIST OF OPCODES INTRODUCED IN THIS CHAPTER

a1 buzz amplitude, frequency, number of harmonics, function id number[, initial phase]
a1 gbuzz amplitude, frequency, number of harmonics, order number of the lowest harmonic, amplitude scale factor [, initial phase]
a1 oscili amplitude, frequency, function id number[, initial phase]

3

SUBTRACTIVE SYNTHESIS

3.1 WHITE NOISE AND FILTERS

Subtractive synthesis utilizes the notion that a new sound can be generated by decreasing the amplitude of the components in a spectrally rich sound source. This requires the use of filters. A filter is a device which allows some frequencies to be emphasized over others.

First, let's consider how we create white noise[1] using Csound. For this we use the *rand* opcode, whose only argument is amplitude. Why does *oscili* have three arguments (amplitude, frequency and waveform) while a noise generator such as *rand* has one? White noise is the sum of all audible frequencies, at equal energy at all frequencies. That means that *rand* generates random waveforms (void of periodic patterns), and that it does not need to be assigned any specific waveform function. The only parameter that makes sense for this opcode, then, is the peak amplitude in the random waveforms it generates.

```
;noise.orc
        sr      =   44100
        kr      =   4410
        ksmps   =   10
        nchnls  =   1
```

[1] We call "white noise" a sound made of all audible frequencies, in analogy with the optical phenomenon of the color white, made of all the colors of the visible spectrum.

```
        instr   1
a1      rand    p4
        out     a1
        endin
```

Score example:

```
;noise.sco
i1   0   3   20000
```

Thus we create a white noise lasting three seconds, with a peak level of 20000.

Now, let's see how we submit such a sound to the filters beginning with high- and low-pass filters.

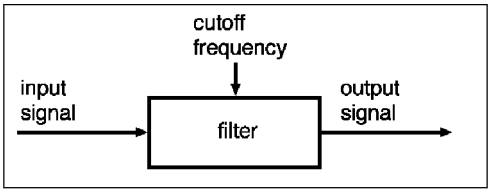
The sound we want to filter is the input signal to the filter. We can determine some of the filter's characteristics such as the frequency bandwidth that will be attenuated, or eliminated, after passing through the filter. Finally, the output from the filter is stored in an audio variable. The following opcodes implement a low-pass and a high-pass, respectively

a. Low-pass filter

a1 tone input_signal, cutoff_frequency [2]

b. High-pass filter

a1 atone input_signal, cutoff_frequency



[2] The opcodes *tone* and *atone*, just like *reson* and *areson* (discussed later) have one more, optional, argument, *istor*. If set to 0, *istor* clears the internal memory storage for the filter at the beginning of the note. If set to 1, it skips the clearing stage during initialization. This stage has minimal effect on the sound.

Other paragraphs in this chapter:

3.2 1st-ORDER LOW-PASS FILTER

3.3 1st-ORDER HIGH-PASS FILTERS

3.4 HIGHER ORDER FILTERS

3.5 BAND-PASS FILTERS

3.6 GAIN UNITS: *RMS*, *GAIN*, *BALANCE*

3.7 MULTI-POLES FILTERS AND RESONANT FILTERS

EXTENSIONS

3.A.1 SUBTRACTIVE SYNTHESIS: HISTORICAL SKETCHES

3.B.1 SUBTRACTIVE SYNTHESIS: THEORY

LIST OF OPCODES INTRODUCED IN THIS CHAPTER

k1	**rand**	**amplitude**
a1	**rand**	**amplitude**
a1	**tone**	**input_signal, cutoff_frequency**
a1	**atone**	**input_signal, cutoff_frequency**
a1	**reson**	**input_signal, cutoff_frequency, bandwidth**
a1	**butterhp**	**input_signal, cutoff_frequency (high-pass)**
a1	**butterlp**	**input_signal, cutoff_frequency (low-pass)**
a1	**butterbp**	**input_signal, cutoff_frequency, bandwidth (band-pass)**
a1	**butterbr**	**input_signal, cutoff_frequency, bandwidth (banda-rejection)**
k1	**rms**	**input_signal**
a1	**gain**	**input_signal, RMS_value**
a1	**balance**	**input_signal, comparator_signal**
ar	**lowres**	**input_signal, cutoff_freq, resonance**

4

FLOW-CHARTS

4.1 GRAPHICAL REPRESENTATION OF PROCEDURES

Any sequence of linked events, or *procedure*, can be represented in several ways: by text, by graphics, or even by acoustical representations. A Csound orchestra is a written description of a set of particular procedures whose result is a generated sound. It is not a very "readable" representation. We can hardly grasp its complete functionality at first glance. Especially, if the orchestra was created by someone else, or if it is complex, you may find it difficult to understand all of its details. What can we do then?

A good solution is to adopt a representation other than text, like a graphical representation, for example. The kind of graphical representation we are going to use here is based on *flow-charts*. Flow-charts are useful in many fields, because they make it easy to grasp the structure of a set of linked events in an intuitive manner. Consider the analogy of a city map. Draw yourself a little map, that’s usually more handy than remembering directions like "take the third right turn, when you get to the pay-phone turn left, go straight to the gasoline station...".

4.2 LET'S GO TO THE SEASIDE

Imagine a situation like this: a friend suggests that you go with him/her to the seaside, in your car early Sunday morning provided it doesn’t rain. What are the operations and choices to make in such a situation? Here's a list of some choices including those which are banal and self-evident:

1. Set the alarm clock - then wake up.
2. Look outside. Is it raining? If it is, go back to bed.
3. Pack a seaside bag, go outside and get in the car.
4. Start the engine. Does it start? If it doesn't, go back to bed.
5. Drive to your friend's place.
6. Is she/he ready? If he/she's not, wait.
7. Drive to the seaside.

Let's see how this simple procedure, called "Go to the seaside", can be represented in a flow-chart. The diagram is in figure 4-1. In the graph, time moves from top to bottom, and also from left to right sometimes. Each rectangular box is labeled (e.g. "Set alarm clock and wake up). Diamond boxes have question marks (e.g. "Watch outside, is it raining?"). Rectangular boxes denote *actions*, and have a single output. Diamonds denote *tests*, and have two outputs, one for tests evaluated as true, one for those evaluated as false. Thus, if we answer the question "is it raining?" with a "yes" (true), the diagram flows to "go back to bed" and the procedure ends with the action "STOP". If, instead, the answer is "no" (false), we can continue the process.

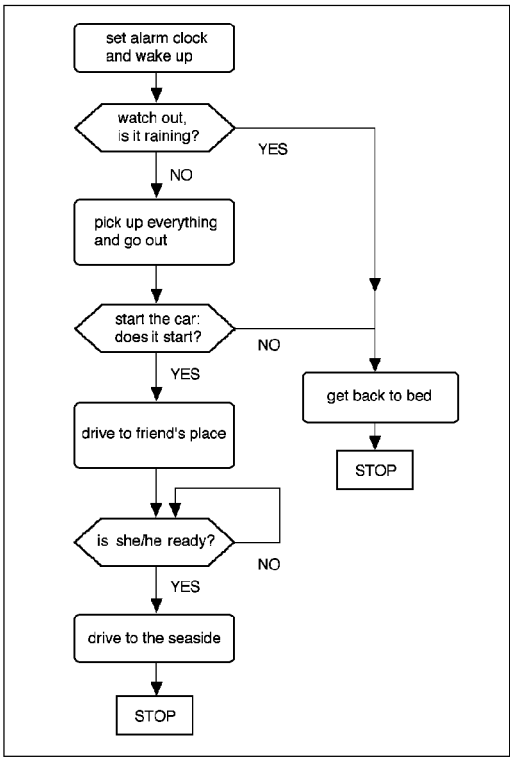


Fig. 4-1

Today, the graphical representation of musical procedures is rather common. In fact, much computer music *software* includes a graphical user interface. As in MAX, KYMA, Patchwork, etc. Therefore, it is useful to learn how this kind of graphical representation works.

Other paragraphs in this chapter:

4.3 SYMBOLS

4.4 COMPLEX DIAGRAMS

5

STEREO, CONTROL SIGNALS, VIBRATO, TREMOLO, 3-D SOUND

5.1 STEREOPHONIC ORCHESTRAS

So far, your orchestra headers always included the assignment *nchnls = 1*, and thus all instruments were generating mono signals. Let's now introduce the possibility of choosing among mono and stereo instruments.

For stereo output, use *outs* rather than *out*. The syntax is as follows

outs left_out_signal, right_out_signal

Observe the following example:

```
;stereo.orc
          sr      =   44100
          kr      =   4410
          ksmps   =   10
          nchnls  =   2               ;notice nchnls=2
          instr       1
asine     oscil       10000, 1000, 1
```

```
asquare   oscil   10000, 220, 2
          outs    asine, asquare  ;asine on left channel, asquare on right
          endin
;
          instr   2
awn       rand    10000
          outs    awn, awn        ;awn on both left and right channels
          endin
```

A score for this orchestra would be:

```
;stereo.sco
f1   0   4096   10   1
f2   0   4096   7    1   2048   1    0   -1   2048   -1
i1   0   5
i2   6   5
```

Here we have a stereo orchestra (*nchnls=2*) which includes two distinct instruments

instr1: generates two signals, *asine*, heard in the left channel, and *asquare*, heard in the right channel
instr2: generates only one signal heard in both output channels, for monophonic effect.

Let's now create a new orchestra which positions notes in three distinct locations in the stereo field, left, center and right.

```
;stereo1.orc
        sr      =   44100
        kr      =   4410
        ksmps   =   10
        nchnls  =   2
        instr       1
ast     oscili      p4, p5, 1
        outs        ast*(1-p6), ast*p6
        endin
```

In this example, the first argument to *outs* (left output) is assigned the *ast* signal multiplied by 1 minus the p6 value from the score (notice that multiplication is denoted by an asterisk, *); the second argument (right output) is assigned again *ast*, but this time multiplied by p6.

Here's a score for this new instrument

```
;stereo1.sco
f1   0    4096    10    9     8     7     6     5     4     3     2     1
i1   0    1             10000             200                 1          ;right stereo output
i1   1.1  1             10000             200                 .5         ;center stereo output
i1   2.2  1             10000             200                 0          ;left stereo output
```

How come the first note is audible in the right channel? Notice that for that particular note we set p6 = 1, and according to the orchestra code, the left output is *ast* multiplied by 1-p6, in this particular case, *ast**(1-1), equals *ast**0. All audio samples in the *ast* signal are set to zero. The right output, on the other hand, is *ast* multiplied by p6, i.e. *ast* * 1. Thus, all audio samples remain untouched (any number multiplied by 1 gives the number itself). Therefore, the sound will come to our ears via the right channel only.

Let's take a look at other stereo panning possibilities. As a general procedure, consider the following

```
outs [audio variable]*(1-p6), [audio variable]*p6
```

Now consider some specific cases. For p6=1

outs [audio variable]*(1-1) = 0, [audio variable]*1 = audio variable

which means: no signal on the left, the whole signal on the right.

For p6=0

outs [audio variable]*(1-0) = audio variable, [audio variable]*0 = 0

hence we get no signal on the right, and the whole signal on the left.

For p6=.5

outs [audio variable]*(1-.5) = audio variable / 2, [audio variable]*.5 = audio variable / 2

which means: The signal appears at equal amplitude in both speakers (monophonic effect).

For p6=.75

outs [audio variable]*(1-.75) = audio variable * 1/4, [audio variable]*.75 = audio variable * 3/4

we get the signal at 1/4 amplitude on the left, and three quarters amplitude on the right.

This way, we are able to allocate a specific position for each note across the stereo front.

If we want a sound to move from, left to right and back, over the course of the note duration, we have to set some control variable with linear values going from 0 to 1 and then back to 0. Let's see an example of this dynamic stereo panning

```
;dynamic stereo panning programmed from the orchestra

          instr   1
kstereo   linseg  0, p3/2, 1, p3/2, 0
ast       oscili  p4, p5, 1
          outs    ast*(1-kstereo), ast*kstereo
          endin

;dynamic stereo panning programmed from the score

          instr   2
kstereo   linseg  p6, p3/2, p7, p3/2, p8
ast       oscili  p4, p5, 1
          outs    ast*(1-kstereo), ast*kstereo
          endin
```

Observe that the second instrument exploits three stereo positions: p6 = initial position, p7 = middle position, p8 = final position (values in the range [0,1]).

Here's a score for this orchestra

```
f1    0    4096    10      1
i2    0    5       20000   500   .5    1     0     ;from center to right, then to left
i2    6    5       20000   500   0     .5    .5    ;from left to center, then remains there
i2    12   4       20000   500   0     .5    1     ;from left to right
```

This method is quite simple, but it is not completely rewarding. If you listen carefully to the third note (shifting from left to right), you may notice that the sound is weaker at the center of the stereo field, and louder when it moves across the left

(beginning) or the right (end) channel. That happens because the perceived sound level is proportional to the signal power which is itself proportional to the square of the amplitude.

There are several ways to solve this problem. Perhaps the most effective one was suggested by Charles Dodge, and consists in defining the gain factors for the two channels (i.e. *kstereo* and *1-kstereo*, in our orchestra) as the square roots of the stereo panning control signal.

Here's that solution:

```
; stereo panning with control signal square root

          instr   1
kstereo   linseg  0, p3/2, 1, p3/2, 0
ast       oscili  p4, p5, 1
kleft     =       sqrt(1-kstereo)       ; square root of 1-kstereo
kright    =       sqrt(kstereo)         ; square root of kstereo
          outs    ast*kleft, ast*kright
          endin
```

Besides *outs*, two more opcodes exist for modifying stereo output, *outs1* and *outs2*. The former sends the signal to the left channel only, the latter to the right channel only.

For quadraphonic sound, the opcode is *outq*, but we can also use *outq1*, *outq2*, *outq3*, and *outq4* to determine the four outputs separately. (Clearly, a quadraphonic sound card is needed to listen to quad sound output).

Let's summarize the syntax of the Csound output signal opcodes

```
out       asig                    ; one output channel (mono output)
outs      asig1, asig2            ; left channel, right channel (stereo output)
outs1     asig                    ; left channel only
outs2     asig                    ; right channel only
outq      asig1, asig2, asig3, asig4  ; 1st chan., 2nd chan., 3rd chan., 4th chan.,
                                  ; (quad output)
outq1     asig                    ; 1st channel only
outq2     asig                    ; 2nd channel only
outq3     asig                    ; 3rd channel only
outq4     asig                    ; 4th channel only
```

Other paragraphs in this chapter:

6

DIGITAL AUDIO

6.1 DIGITAL SOUND

Music played back from vinyl discs (LPs), open-reel tapes, cassette tapes, radio, and television, consists of electrical signals converted to sound by the loudspeakers. Such signals are “analog” signals, meaning that their voltage variations are proportional to the pressure variations in the represented sound signal.

Music played back from CDs, DATs, MiniDiscs, and computers, are comprised of “digital” signals, which represent the sound pressure variations by means of a sequence of binary digits, or information units (bits).

As is well-known, any sound is completely defined by its instant amplitude values. A digital device generates a sequence of numbers, each corresponding to a single instant amplitude.

In digital media (either based on mechanical technology, such as the CD player, or magnetic technology, such as the DAT recorder) signals are written and read as sequences of binary digits, 1 and 0. Correction mechanisms are usually applied during the playback process, to prevent moderate flaws and defects in the physical material (e.g. small scratches or dust on the CD surface) and even minute demagnetization (of the DAT tape) from causing serious problems. Digital copies are always perfectly identical to the original but analog copies, no matter how good, introduce signal degradation.

Finally, the signal-to-noise ratio in digital media is much better than analog media It is approximately 96 dB for CD and DAT (the two having more or less the same audio quality) while it is only 60 or 70 dB for analog tape recorders not using noise-reduction systems.

6.2 ANALOG-TO-DIGITAL AND DIGITAL-TO-ANALOG CONVERSION

Digital audio systems include special circuits capable of converting signals from analog to digital and from digital to analog. These are called A-D converters (ADC) and D-A converters (DAC).

Let's briefly examine the functionality of the A-D process. The process is to translate a series of electrical magnitudes into a sequence of numbers, so that each number in the sequence captures a particular voltage at a specific time. In fig.6-1, the continuous line is an analog signal, i.e. the unfolding in time of an electrical voltage. We divide the time-axis into equally-spaced, shorter segments and register the corresponding amplitude values from the analog signal. Each value, or "sample", remains unchanged until the next is registered. We obtain a step-valued signal which is a rough representation of the original.

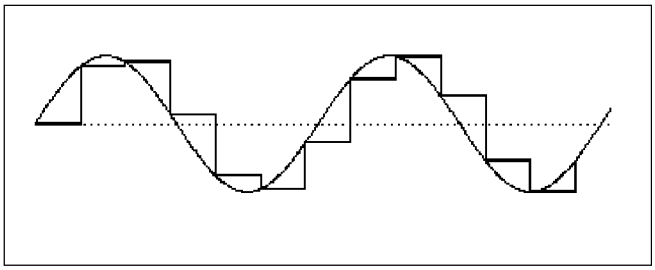


Fig. 6-1

Clearly, the smaller the time between successive samples the more accurate the representation. In theory, if we could divide the time-axis into infinitely small portions, the two signals would be identical.

The time between successive samples is called the *sampling time*. Its inverse, 1/*sampling time*, is the *sampling frequency*, or *sampling rate* (*sr*). To correctly perform an analog to digital conversion, the sampling frequency must be at least twice as high as the frequency of the highest component in the recorded signal. Thus, to convert a sound including components up to 20000 Hz, for example, we need a sampling rate of at least 40000 Hz. A lower sampling rate causes the phenomenon known as *foldover* where components with a frequency higher than half the sampling rate are folded back into the audible range.

With *sr* = 20000 Hz, an analog signal with a frequency of 11000 Hz would result, after the conversion process, in a frequency of 9000 Hz (see section 6.B.1).

Another crucial factor in the digital domain is the number of binary digits available to store the amplitude values after the conversion process itself. Of course, we are bound to a finite number of digits, typically 16 bits, that allows for the coding of integer numbers within the range [-32768, +32767], and corresponds to 65535 different amplitude values. With 7 bits we would be limited to as few as 127 different values, and a very poor audio quality (for details, see section 6.A.1). Indeed, 16-bit numerical resolution is a basic pre-requisite for high quality digital audio. Recently some proposals were put forth, for better audio quality: the DVD (Digital Versatile Disc) allows 5 audio channels with 24-bit resolution, and at sampling rates as high as 48 or even 96 kHz.

To conclude, we should notice that a digital signal converted to analog remains a step-valued signal, and its spectrum includes many replicas, or *alias* images, of the expected spectrum (fig.6-2).

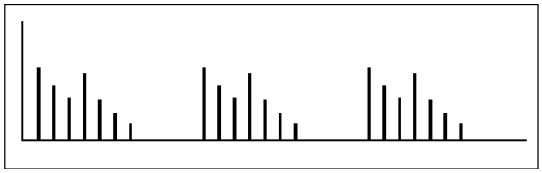


Fig. 6-2

This phenomenon is caused by harmonic distortion introduced by the discrete steps between any two successive samples. To avoid the audible artifacts of this phenomenon, D-A converters have a built-in analog filter to remove the alias spectra (*anti-aliasing* filter). This filter is a low-pass with cutoff frequency (*fc*) set to half the sampling rate

fc = sr / 2

The frequency response of anti-aliasing filters is far from ideal, and no such filter is capable of completely removing the unwanted frequencies. Even a sharp cutoff would be of little use, as it would cause new artifacts in the frequency response curve (such as a rippled response curve) and other side-effects (phase distortion). To get rid of this problem, today D-A converters work at much higher sampling rates (*oversampling*), of at least 4 times the nominal value, such that the aliased components are shifted far above the pass band of the anti-aliasing filter (fig. 6-3).

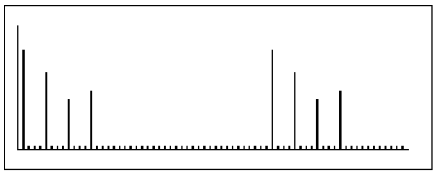


Fig. 6-3

Other paragraphs in this chapter:

6.3 SOUND CARDS AND AUDIO FILE FORMATS

6.4 SOME ANNOTATIONS ON DIGITAL AUDIO FOR MULTIMEDIA APPLICATIONS

EXTENSIONS

6.A.1 DIGITAL-TO-ANALOG AND ANALOG-TO-DIGITAL CONVERSION

6.B.1 FOLDOVER

7

SAMPLING AND PROCESSING

7.1 THE *SOUNDIN* AND *DISKIN* OPCODES

In Csound we can play back sampled sounds by using the *soundin* opcode. It provides no processing possibilities, but, on the other hand, it is fairly simple to use.

Basically, it is an audio signal generator whose output samples are taken directly off a pre-existing sound file. This requires no function table, as the generated waveform is that of the played back sound file itself. Also, it requires no frequency and amplitude arguments, as Csound uses the sound file default playback.

The syntax is the same for mono, stereo and quad output signals:

audio variable	opcode	filename	initial time to skip	sound file format	comment
a1	**soundin**	**ifilcod**	**[, iskptim]**	**[, iformat]**	**; mono**
a1, a2	**soundin**	**ifilcod**	**[, iskptim]**	**[, iformat]**	**; stereo**
a1,..., a4	**soundin**	**ifilcod**	**[, iskptim]**	**[, iformat]**	**; quad**

As you see, only the first argument is required, and its meaning is self-evident. It is the name of the sound file that we want to play back.

For example, we may write:

```
a1      soundin "cork.wav"              ;the filename must be within double quotes
```

Csound looks for the required sound file in the current directory first, then in the SSDIR and finally in the SFDIR directory (if specified in the Csound system preferences). You can also specify the entire path for the sound file:

```
a1      soundin "c:/corks/cork.wav"     ;path and filename within double quotes
```

Notice that, starting with the Windows 3.50 Csound release, to write pathnames you should use a regular slash ("/"), not the back-slash ("\"). The latter is now used as a line continuation symbol - very useful when you type in very long lines of Csound code. For example, the following line:

```
f1   0   4096   10   1   2   3   4   5   6   7   8   9   10
```

can be written as:

```
f1   0   4096   10   1   2   3   \
4   5   6   7   8   9   10
```

To reference the sound file, we can also call it with the special name "soundin" plus the number of the particular extension. For example: *soundin.1*. That implies that the file to be read had been previously saved to hard disk with that particular name and extension:

```
a1      soundin.1               ; in such case quotes are omitted
```

Let's now consider the *skiptime* argument (optional). Suppose the file we want to play contains the sound of a voice speaking the word "beetle". If we want to play back the entire file, we do not need to specify anything, as the *skiptime* default value is 0 (Csound reads the file through, starting from the beginning). However, if we want to play only the final portion of the word, "tle", omitting the "bee", the file being read will have to start at some point other than the beginning, say at 0.1 seconds.

Then we write:

```
a1      soundin "beetle.wav", .1        ; generates "tle", omitting the "bee"
```

The third argument, *iformat*, can often be dropped. It is necessary only when the sound file was created with some non-standard format lacking a readable header.

A Csound instrument can feature as many *soundin* opcodes as desired, limited only by the operating system setup options and by the particular Csound release being used. Within an instrument, *soundin* opcodes can refer either to a single sound file or to several sound files. When many *soundin* opcodes refer to the same sound file, they can have either the same or a different *skiptime*.

Be aware that the *soundin* opcode cannot be re-initialized (this is explained in section 17.3).

EXERCISE 1 *Create a new instrument with a sound file submitted to some filtering.*

EXERCISE 2*: Create a new instrument playing back a sound file. The soundin skiptime values are assigned from the score (p4), such that the file is read starting from a different point at each note. After that, try notes with durations shorter than the actual sound file duration such that the playback is interrupted at some random point before the end of sound file is reached.*

diskin is similar to *soundin*, but it also allows changes in the playback speed and direction, and for creating simple sound loops. The syntax is as follows:

a1[,a2[,a3,a4]] diskin ifilcod, kpitch[,iskiptim][, iwraparound] [,iformat]

ifilcod name of the sound file; works just as in *soundin*.
kpitch ratio between the output frequency and the sound file original frequency.

e.g.:

kpitch = 1 the sound file is read through and played back with no modification;
kpitch = 2 double playback speed, resultant pitch is one octave higher;
kpitch = .5 half the normal speed, resultant pitch is one octave lower;
kpitch = 3 three times the normal speed, an octave-plus-a-fifth (= a 12th) higher

See the ratio-to-interval chart in the appendix to Chap.8. If *kpitch* is negative, the sound file is played in the reverse direction, from end to beginning.

For example:

kpitch = -2 double speed (one octave higher), from the last to the first sample (reverse playback).

iskiptim (optional) *skiptime*; works just as with *soundin*.

iwraparound (optional) useful for implementing a sound loop. It must be 1 (= on) or 0 (= off). When on, if note duration is longer than the sound file duration, playback returns to the beginning everytime the end of sound file is reached (the reverse is true when *kpitch* is negative).

iformat (optional) works as with *soundin*.

Like *soundin*, *diskin* cannot be re-initialized (see 17.3).

Other paragraphs in this chapter:

7.2 COPYING SOUND FILES TO FUNCTION TABLES (GEN01)

7.3 READING A SOUND FILE TABLE WITH *LOSCIL*

7.4 RELEASE LOOP WITH *LINENR*

7.5 THE *FOLLOW* OPCODE

7.6 *LIMIT* AND *ILIMIT*

LIST OF OPCODES INTRODUCED IN THIS SECTION

a1[,a2][,a3,a4]	**soundin**	**filename[, initial_skip_time] [, format]**
a1[,a2][,a3,a4]	**diskin**	**filename, freq., [, initial_skip_time] [, loop_flag][, format]**
a1	**loscil**	**amp, freq, func_table_no. [, base_frequency] [sustain_loop_flag, loop_start_time, loop_end_time] [decay_loop_flag, loop_start_time, loop_end_time]**
k1	**linenr**	**amplitude, rise_time, decay_time, decay_bias_value**
a1	**linenr**	**amplitude, rise_time, decay_time, decay_bias_value**
a1	**follow**	**input_sig, time_window**
a1	**limit**	**input_audio_sig, lower_limit, upper_limit**
k1	**limit**	**input_control_sig, lower_limit, upper_limit**
i1	**ilimit**	**input_init_variable, lower_limit, upper_limit**

8

SOUND ANALYSIS AND RESYNTHESIS

8.1 INTRODUCTION

Analysis/resynthesis is one of the most interesting approaches to sound processing. The following is an overview of the approach:

1) A sound file is separated into its spectral components using an some particular computer program. The analysis data are stored in an analysis file.

2) The analysis file can be edited, and its data modified. The new data may differ from the original sound.

3) The modified data is used to synthesize a new sound file.

This way it is possible to stretch the sound duration while leaving the frequency unchanged or change the frequency leaving the duration unchanged. Such transformations may be dynamic (accelerando, glissando, etc.). Depending on the modifications to the analysis data, a vast palette of very interesting sound effects can be created.

Several analysis/resynthesis methods exist. We will focus on the following ones in Csound

kind of analysis technique	analysis program	analysis file type	resynthesis method	Csound resynthesis opcode
phase vocoder	*pvanal*	fft (**Win**) or pv (**Mac**)	inverse FFT phase vocoder	*pvoc*
heterodyne analysis	*hetro*	het	oscillator banks or additive synthesis	*adsyn*
LPC (linear prediction code)	*lpanal*	lpc	filter banks	*lpread/lpreson*

Phase vocoding is widely utilized today even by proprietary, commercial programs. Such programs, however, only allow limited modification of a number of parameters, and provide little or no explanation regarding the effected output. We'll discuss the theory behind the phase vocoder in next section. It will help you to fully exploit not only the potential of Csound phase vocoding, but any other phase vocoder as well.

Other paragraphs in this chapter:

8.2 PHASE VOCODER ANALYSIS

8.3 PHASE VOCODER RE-SYNTHESIS

8.4 HETERODYNE ANALYSIS (*HETRO*)

8.5 *ADSYN* RESYNTHESIS

8.6 MODELING THE VOCAL TRACT WITH *LPANAL*

8.7 MODELING THE VOCAL TRACT: LPREAD/LPREASON RESYNTHESIS

EXTENSIONS

8.A.1 FAST FOURIER TRANSFORM (FFT)

LIST OF OPCODES INTRODUCED IN THIS SECTION

a1 pvoc look-up_pointer, freq_scale_factor, analysis_file [, spectral_envelope_preservation_code]
a1 adsyn amp_scale_factor, freq_scale_factor, increment_scale_factor, analysis_file

krmsr, krmso, kerr, kcps lpread ktimpnt, ifilcod
krmsr average amplitude of analysis residual
krmso average amplitude of input signal
kerr estimated error in tracking fundamental frequency
kcps tracked fundamental frequency

a1 lpreson excitation_source_signal

9

USING MIDI FILES

9.1 STANDARD MIDI FILES

In 1982, the biggest music instrument manufacturers came to an agreement concerning a standard communication protocol for electronic instruments known as MIDI (Musical Instrument Digital Interface). In 1993, a standard file format was established, the Standard MIDI File (SMF) is used for storing MIDI data in a format common to many commercial software products (sequencers, notation programs, etc.). Under Windows operating systems, for example, standard MIDI files have the extension .MID and contain information needed to drive any MIDI device.

There exist two types of SMF, called Type 0 and Type 1. The only difference between them is that Type 0 has one track of MIDI data, while Type 1 can contain as many as 256 tracks. The single track in Type 0 can, however, address MIDI messages through any of the 16 MIDI channels.

Type 0 SMFs include the following information:

General header (with data such as the id file format, time patterning, metronome, tempo, musical key, etc.)
Track header (track id, etc.)
Track data (executable MIDI messages, such as note *ons* and *offs*, *program changes*, etc., separated by time lags).

Type 1 SMFs include the following:

General header
Track 1 header
Track 1 data
Track 2 header
Track 2 data
...
Track 19 header
Track 19 data.
etc.

Nearly all computer music applications can save data either in a proprietary format or the SMF format.

Other paragraphs in this chapter:

9.2 USING STANDARD MIDI FILES IN CSOUND

9.3 INSTRUMENT ASSIGNMENT

9.4 MIDI VALUE CONVERTERS

9.5 CONVERTING SCORE FILES TO SMF FILES AND VICE-VERSA

EXTENSIONS

9.A.1 THE MIDI STANDARD

9.A.2 FEATURES OF MIDI DEVICES

9.A.3 MIDI NUMBERS

9.A.4 THE MIDI PROTOCOL

LIST OF OPCODES INTRODUCED IN THIS SECTION

k1 linenr amp, rise_time, decay_time, attenuation_factor_of_decay_curve
a1 linenr amp, rise_time, decay_time, attenuation_factor_of_decay_curve

kval midictrlsc controller_number [,max_value] [, min_value] [,initial_value]

See section 9.4 for other opcodes.

10

REAL TIME MIDI CONTROLS

10.1 USING CSOUND IN REAL TIME

As the power of personal computers increases, it becomes possible to synthesize sounds in real time. The degree of complexity of the generated sounds depends on the orchestra and the number of separate musical lines we want to create (polyphony), and, to a larger extent, on the power of the computer used.[1]

Real time sound synthesis presents many musical possibilities for interaction and synchronization with vocalists and instrumentalists. It is possible to trigger some process or event with an external event or message. It is possible to adjust the synthesis timing with the timing of live instrumentalists, etc. The point here is that the machine operations now follow from human interactions, rather than the other way round (as with pre-recorded tape).

This new situation calls for real time signal processing, a possibility that in the past was reserved solely for specialized hardware and expensive computers.

Presently, the only means we have to control Csound in real time is through MIDI messages. We can send MIDI messages from any MIDI device, such as master keyboards, MIDI controllers (also known as “MIDI mixers”, due to the fact that you operate them with faders, like a mixing console). But we can also send MIDI messages

[1] With “power”, here we mean mainly the effective speed of floating point operations, a crucial factor for synthesis engines like Csound.

from another computer, and even from the same computer running Csound! There's no limit to MIDI connections except in our own imagination.

Clearly, if we are to take such an approach, it behooves us to exploit the computer resources in the most efficient way possible. That implies, for example, that we define the orchestra code with due attention to avoiding redundant operations (especially operations on audio- and control-rate variables) and use the slowest applicable sampling and control rates.

As an example, the line

```
a2      =       a1/2
```

should be replaced with

```
a2      =       a1*.5
```

as in fact multiplies are performed faster than divides. The line

```
aout    =    a1*kvol/4+a2*kvol/4+a3*kvol/4
```

should be replaced with the following:

```
k1      =       kvol*.25
aout    =       (a1+a2+a3)*k1
```

thereby using one variable assignment and two multiplies instead of three divides and three multiplies.

To use Csound in real time, it is necessary to replace the output file name with words such as *devaudio* or *dac* (depending on the particular computer platform and the particular Csound release), referencing the soundcard or other output audio device available on your computer.

Therefore, instead of

csound -W -oyourfile.wav yourorc.orc yourscore.sco

you want to write

csound -odevaudio yourorc.orc yourscore.sco

How to launch real time Csound (*Win*)

In the synthesis dialog box, beside the output sound file window is the *Realtime Out* button. Click on it, and you should see the sound file name change to *devaudio*. The button itself has a different label on it now, *Audio File*. Hitting this button, you can switch between real time and deferred time (sound file) output. We recommend you change the Csound buffer size when creating sounds in real time.
See section 10.3.

How to launch real time Csound (*Mac*)

It is sufficient to hit the *Audio out* button. The sound samples will be sent, then, to the computer's *Sound Manager*, which routes the audio samples to the available output device. To change the output buffer size, select *Set Buffers* from the *Preferences* menu.
See section 10.3.

Other paragraphs in this chapter:

10.2 REAL TIME ORCHESTRAS AND SCORES

10.3 SOME CAUTIONS

11

AMPLITUDE MODULATION AND RING MODULATION

11.1 INTRODUCTION

"Modulation is the alteration of the amplitude, phase, or frequency of an oscillator in accordance with another signal".[1] The modulated oscillator is called the *carrier*, the modulating oscillator is called the *modulator*.

Do you remember the way in which we created tremolo?

What we did was to introduce a slight **amplitude modulation** by means of a modulating signal (a control variable), and that changed the amplitude of a carrier signal. In that case, the modulator caused limited amplitude variations, and had a very low frequency in the sub-audio range (*infrasonic* frequencies). But, what happens if we use a modulator with a frequency higher than 20 Hz? The resulting sound is different, and includes new frequency components to the side of the carrier frequency. These new components are usually called *sidebands*, as they are symmetrically placed below and above the carrier frequency as we shall see in a moment.

Amplitude modulation (AM) and *ring* modulation (RM) both are based on this simple process. The difference is that the former involves a *unipolar* modulator signal while the second involves a *bipolar* one (see fig.11-1).

---

[1] C.Dodge & T.Jerse, *Computer Music*, Schirmer, New York, 1985, p.80.

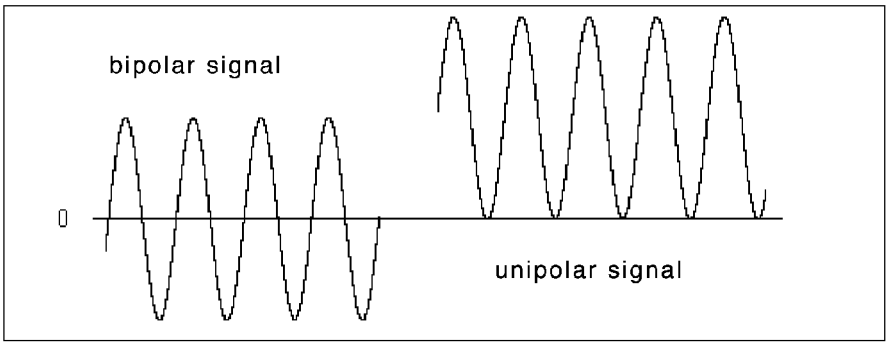


Fig. 11-1

A **bipolar** signal oscillates between positive and negative peaks:

```
a1 oscili 10000, 220, 1          ;signal oscillates between 10000 and -10000
```

A **unipolar** signal oscillates only in the positive (or negative) field. To create a unipolar signal, we have to add some constant value to a bipolar oscillation. The constant value is usually referred to as *DC offset* (direct current offset, see section 2.7). Let's consider an example of a unipolar signal oscillating in the positive field:

```
abimod      oscili 1 , 220, 1      ;abimod oscillates between 1 and -1
aunimod  = abimod+1                ;aunimod oscillates between 0 e 2, i.e.
                                   ;in the positive field
```

As you see, we simply add a constant value, 1, to the signal *abimod*, and make it oscillate only in the positive field.

In chap.5, we used both uni- and bipolar modulators with a very low, infrasonic, frequency.

Other paragraphs in this chapter:

11.2 AMPLITUDE MODULATION (AM)

11.3 RING MODULATION (RM)

EXTENSIONS

11.A.1 THE AM AND RM FORMULAS

11.B.1 HISTORICAL SKETCHES ON RING MODULATION

12

FREQUENCY MODULATION (FM)

12.1 BASIC FM THEORY

Like amplitude modulation, frequency modulation (FM), too, involves a modulating oscillator and a carrier oscillator (at least in the simplest setup). In this technique, however, the modulator drives the frequency of the carrier, not its amplitude. The flow-chart of a simple FM instrument is illustrated in fig.12-1. As you see, this is similar to the flow-chart for the vibrato instrument discussed in section 5.3.

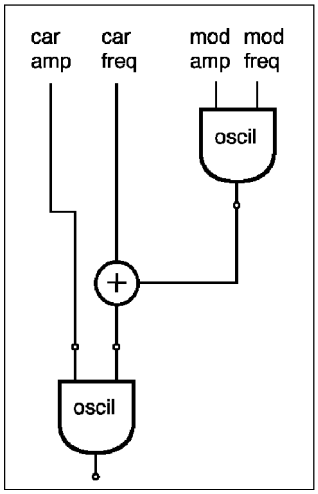


Fig. 12-1

So we have two sine wave oscillators, one called the carrier, the other called the modulator. If the modulator amplitude is 0, there will be no modulation and the output signal will be identical with the carrier signal. By increasing the modulator amplitude, we introduce frequency modulation. In effect the carrier frequency will go up and down following the waveform pattern of the modulator. When the modulating oscillation is in the positive field, the carrier frequency will raise above the base frequency. When it moves to the negative, the carrier frequency shifts below the base frequency. As we increase the modulator amplitude, the amount of deviation gets larger. The maximum deviation in the carrier is called the *peak frequency deviation*, and is measured in Hz.

In AM, a sine wave modulator + carrier coupled results in a spectrum with three components (the carrier frequency plus two sidebands). In FM, theoretically the number of sidebands can be infinite. However, the number of audible sidebands depends on the peak frequency deviation. The higher the peak, the more numerous the sidebands in the output sound spectrum.

With a carrier frequency (**C)** of 1000 Hz and a modulating frequency (**M**) of 3 Hz, the following sidebands are obtained:

1003 (C+M)	997 (C-M)
1006 (C+2*M)	994 (C-2*M)
1009 (C+3*M)	991 (C-3*M)
1012 (C+4*M)	988 (C-4*M)
1015 (C+5*M)	985 (C-5*M)
.....	.....

In theory, the FM sidebands are *always* infinite in number. In practice, however, the higher-order sidebands often have too weak an amplitude level to be heard, especially when the modulation index is small and the modulator frequency is in the sub-audio range. The peak frequency deviation is calculated multiplying a constant, called the modulation index (**I**) by the modulator frequency:

D = I * M

Consider the following score. We will use this score in conjunction with the orchestra illustrated in next section, *fm.orc*:

```
;fm.sco
f1   0    4096   10    1
;    start    dur      car amp    car frq      mod frq      mod indx
i1   0        2.9      10000      1000         3            10
```

```
i1    3     .     10000    1000    3       30
i1    6.    .     10000    1000    3       50
i1    9.    .     10000    1000    3       1000
```

With the first three notes, we perceive a sine tone sweeping higher and lower across the frequency range, in a glissando effect. Here the sidebands are so close to the carrier frequency (1000 Hz) that they fall in the same critical band.[1] As they are not perceptible they do not affect the overall timbre.

Things change with the fourth note. Although the modulator has the same 3 Hz as the preceding notes, the modulation index is much larger and causes the frequency deviation to be wider. The amplitude of the sidebands is sufficiently large, and we will hear something that is not quite the same as a sine wave with glissando. Let's calculate the peak deviation for this note:

D = 3*1000 = 3000 Hz.

The audible components of this sound fall within a frequency band as wide as 6000 Hz, and range from -2000 Hz (C - D = carrier frequency minus peak deviation = 1000 - 3000) to 4000 Hz (C + D = carrier frequency plus peak deviation = 1000 + 3000). Things seem to be getting more and more complicated. What happens when frequencies are negative?

Negative frequency components are identical with positive frequency components, but with inverted phase. They sum algebrically with positive components and are destructive to those components that happen to have the same frequency. The phenomenon is illustrated fig.12-2.

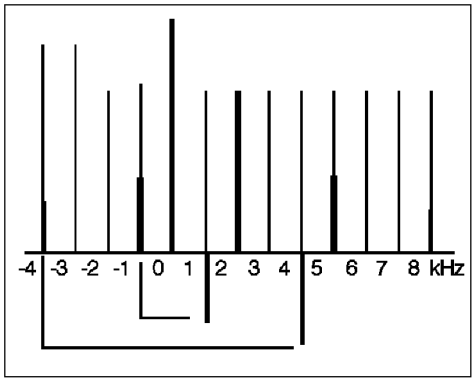


Fig. 12-2

[1] The "critical band" is defined as the smallest frequency difference between two components that allows the human ear to hear those components as separate and distinct sounds, not as a single sound.

There is a 2000 Hz carrier and a 3000 Hz modulator. In the resulting sound spectrum, the -1000 Hz and -4000 Hz components fold back into the positive frequency range and are phase-shifted, causing a loss of amplitude in the 1000 Hz and 4000 Hz components.

This is similar to the foldover phenomenon which takes place when we try to generate frequencies higher than the Nyquist frequency (*sr/2*). For example, suppose a sampling rate of 22050 Hz is used (Nyquist frequency = 11025 Hz), and a FM synthesis instrument is used with C = 5000 Hz and M = 5000 Hz and with I = 3. The sideband frequencies sweep between -10000 Hz and +20000 Hz, and that causes the folding of both the higher components (higher than 11025 Hz) and the negative components. This is illustrated in fig.12-3. You see that there are at least four components higher than the Nyquist frequency. Consider the 15 kHz one. It folds back into the audible frequency range, as a frequency of 15000-11025=3975 Hz.

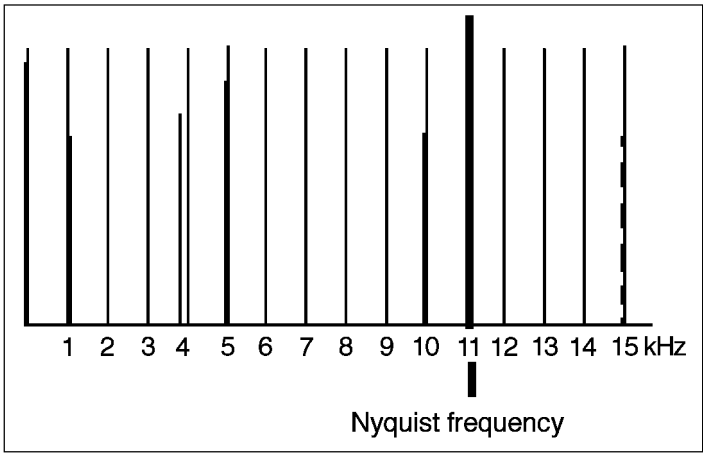


Fig. 12-3

Other paragraphs in this chapter:

12.2 SIMPLE FM ORCHESTRAS

12.3 SOUND SPECTRA FAMILIES

12.4 MULTIPLE-CARRIER FM

12.5 MULTIPLE-MODULATOR FM

EXTENSIONS

12.A.1 FM FORMULAS

12.A.2 SIMULATION OF INSTRUMENTAL SOUNDS

LIST OF OPCODES INTRODUCED IN THIS CHAPTER

ar foscil **amplitude, nominal_frequency, carrier freq, modulating freq, index, function_number[,phase]**

ar foscili **amplitude, nominal_frequency, carrier freq, modulating freq, index, function_number [,phase]**

13

GLOBAL VARIABLES, ECHO, REVERB, CHORUS, FLANGER, PHASER, CONVOLUTION

13.1 ECHO AND REVERB

Echo and reverb are well-known “effects” utilized in sound synthesis and processing. The echo effect simulates the reflection of a sound against a surface. Itës only audible when the reflection is heard at least 1/20th of a second after the direct sound (see fig.13-1, top). If there are several reflecting surfaces, as is the case in a cube shaped room, we hear multiple echoes (fig.13-1, bottom).

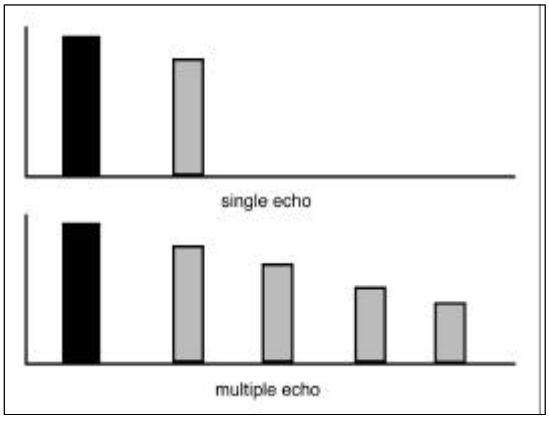


Fig. 13-1

A true reverberation is achieved when multiple echoes merge together (fig.13-2). In this case, we hear the early reflections first, a few milliseconds after the direct sound. Then we hear several echoes that fuse together and slowly decay in amplitude.

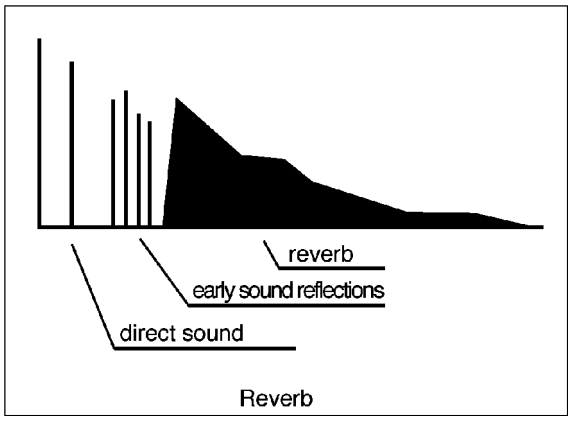


Fig. 13-2

In fig.13-3 we see the map of a rectangular room. The first sound that reaches the listener's ear is the direct signal from the sound source itself, followed next by several sound reflections. These sounds travel a longer distance to the ear, and therefore are delayed. The delays differ in the number of reflections. First we hear the echoes of a single reflection, then those of two, then three reflections, and so on. The higher the number of reflections, the softer the amplitude level of the echo, as in fact each reflection implies some loss of energy.

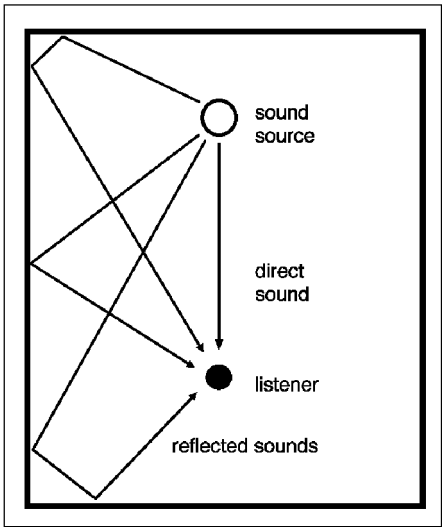


Fig. 13-3

We call *reverberation time* the time it takes for the echoes to decrease in amplitude by 60 dB. It is one of the main acoustical characteristics of a room or hall. In theory, to really understand the reverberation of a particular room, several tests are required to analyze the surface materials which have different reflecting properties at different frequency regions.

Other paragraphs in this chapter:

13.2 THE *DELAY* OPCODE

13.3 REVERB

13.4 LOCAL AND GLOBAL VARIABLES

13.5 MORE EFFECTS USING DELAY: FLANGING, PHASING AND CHORUS

13.6 CONVOLUTION

EXTENSIONS

13.A.1 HOW TO BUILD UP A REVERB UNIT

LIST OF OPCODES INTRODUCED IN THIS CHAPTER

ar delayr delay_time[, internal_memory_storage]
delayw input_signal
a1 deltap delay_time
a1 deltapi delay_time
ar delay input_signal, delay_time[, internal_memory_storage]
ar delay1 input_signal [,internal_memory_storage]
ar reverb input_signal, reverb_time[, internal_memory_storage]
ar reverb2 input_signal, reverb_time, high_freq_reverb_time[, internal_memory_storage]
ar vdelay input_signal, delay_time, max_delay_time[,internal_memory_storage]
ar1[,...[,ar4]]] convolve input_signal, filename, channel
ar comb input_signal, reverb_time, loop_time [,internal_memory_storage]
ar alpass input_signal, reverb_time, loop_time [,internal_memory_storage]

14

THE TABLE OPCODE. WAVESHAPING SYNTHESIS, VECTOR SYNTHESIS

To use function tables of any kind and shape for generation of signals (with *table* opcode) it is necessary to use the appropriate GEN routines introduced in this chapter.

14.1 GEN02 AND SOME OBSERVATIONS ON FUNCTIONS

GEN02 transfers the required parameter values (p-field values other than p1, p2, p3 and p4) into a memory table.

The syntax is:

fn t s 2 v1 v2 v3 ...

n	function table number
t	action time
s	table size
2	GEN routine number (if positive, the function values are normalized to peak at an amplitude of 1; if negative, normalization is omitted)
v1, v2, ...	values

Let’s see an example:

```
f1 0 16  2  0 1 2 3 4 5 6 7 8 9 10 9 8 7 6 5
; normalized values, range = 0-1

f1 0 16 -2  0 1 2 3 4 5 6 7 8 9 10 9 8 7 6 5
; non-normalized values, range = 0-10
```

Both tables are made of 16 points. They include 16 positions, each with a value of its own. The contents of the second table, with non-normalized values, are as follows:

Index	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Table values	0	1	2	3	4	5	6	7	8	9	10	9	8	7	6	5

The first table includes normalized values. The contents are as follows:

Index	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Table values	.0	.1	.2	.3	.4	.5	.6	.7	.8	.9	1	.9	.8	.7	.6	.5

The values we entered in the second table do not get normalized, due to the fact that we set a negative GEN number. Here's another example, using GEN10:

```
f1 0 4096 10 2        ;values normalized (-1 to 1)
f1 0 4096 -10 2       ;values in the required range (-2 to 2)
```

Before we go through the remainder of this chapter, let's define some of the terms and concepts relative to their functions.

a. Functions

A function is a mathematical method for generating the values of one variable from another variable. The former is called the *dependent* variable. The latter is called the *independent* variable.

The general notation for a function is:

y = f(x)

where:

x=independent variable (can assume any value)
y=dependent variable (values vary with x according to the law expressed by *f*)

(consider these examples: y=2*x, y=sin(x), $y=4*x^2+3$...)

b. GEN

In Csound, a **GEN** routine represents a method, labeled with a number, for generating a series of values to be stored into a series of memory locations (= a table). Each GEN represents the implementation of a separate function.

c. Tables

A **table** is a 1-dimensional set of values (= an array), which you access by specifying some **index**. For example, given the following table, the **index** 4 returns a **value** of 7:

Index	0	1	2	3	4	5	6	7	8
Table values	1	5	7	3	7	9	56	3	12

Csound stores different tables in different memory areas. Each table is identified by a number (the total number of tables is dependent on the particular Csound release). In each table, new values can be entered to replace previously entered values. At some predetermined time, new values (perhaps generated by a new function) fill the table and replace older ones.

Here's an example:

```
f 1 0  4096 10 1
;this table is valid from 0 to 10 secs
;at 10 seconds, new values replace the older, as from the table function
;below
f 1 10 4096 10 1 .5 .4 .3 .2 .1
;this new table remains valid until it is cancelled out by the statement
;below (a negative function number causes the table function to be
;destroyed)
f -1 20
```

Other paragraphs in this chapter:

14.2 THE *TABLE* OPCODE

14.3 LINE-SEGMENTS, EXPONENTIALS, CUBIC SPLINES: GEN05, GEN07 AND GEN08

14.4 WAVESHAPING SYNTHESIS (NONLINEAR DISTORTION)

14.5 USING CHEBYCHEV POLYNOMIALS (GEN13)

14.6 USING *TABLE* FOR DYNAMIC RANGE PROCESSING (COMPRESSORS AND EXPANDERS)

14.7 GEN03

14.8 TABLE CROSSFADE: VECTOR SYNTHESIS

LIST OF OPCODES INTRODUCED IN THIS SECTION

i1	**table**	**index, function_table [, index_mode][, offset][, wrap]**
k1	**table**	**index, function_table [, index_mode][, offset][, wrap]**
a1	**table**	**index, function_table [, index_mode][, offset][, wrap]**
i1	**tablei**	**index, function_table [, index_mode][, offset][, wrap]**
k1	**tablei**	**index, function_table [, index_mode][, offset][, wrap]**

15

GRANULAR SYNTHESIS AND FORMANT SYNTHESIS

15.1 WHAT IS GRANULAR SYNTHESIS

"When the slow variations in the sound are thought of as discrete-time functions" (as is the case with Csound control variables) "the generated sound should be described as a chain of elementary sounds, each having its own, constant characteristics. [...] The elementary sounds are called *grains*, and the technique exploiting this facility is *granular synthesis*" [De Poli, 1981].

Granular synthesis involves creating large masses of small acoustical events, called "grains", of duration ranging from 10 to 100 milliseconds. A sound grain is usually characterized by a symmetrical envelope shape. Typical grain envelopes include the bell-like Gaussian curve, the 3-segment trapezoid envelope, and the 2-segment attack-and-decay envelope (see figures 15-1 and 15-2).

In Csound, granular synthesis usually implies that a single "note" (a single *i* statement in the score) gives rise to a very complex event, sometimes including the synthesis of a few thousands grains per second. Starting with very short and simple sonic units, granular synthesis creates extremely rich and articulated sound structures. Another

approach involves generating thousands of *i* statements (via some preprocessing software), each of which produces a single grain.

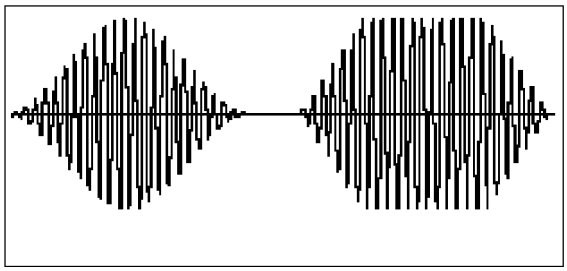


Fig. 15-1

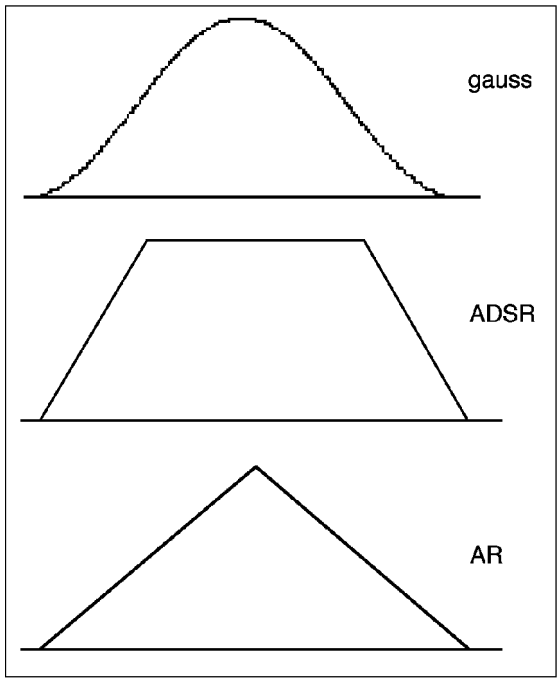


Fig. 15-2

In 1947, Dennis Gabor first discussed a granular approach in his article "Acoustical Quanta and the Theory of Hearing". Later, Norbert Wiener (1964) and Abraham Moles (1969) contributed to the approach. Iannis Xenakis, in his *Musiques Formelles* (1971), was the first composer to define a compositional theory based on sound grains. In 1975, Curtis Roads implemented a simple form of digital granular synthesis, experimenting with his own automated methods for generating massive streams of grains, and also composing his *prototype* (1975). Later he published two important articles on granular synthesis, in 1978 and 1985.

During its early stages, granular synthesis was difficult to use, due to the enormous amount of calculations required. In 1986 Barry Truax first proposed a real-time implementation using a dedicated digital signal processor called the DMX-1000. He also composed *Riverrun* (1988), a tape piece where sound grains are likened to water drops, which little by little accumulate and give rise to the powerful image of large river flow (for the implementation of Truax's method, see the discussion in Eugenio Giordani's lecture at the end of this book).

The waveform of a sound grain can be either extracted from a sampled sound or generated with another synthesis method, like FM or additive synthesis. In any case, the sound grain waveform is conveniently stored in a table. This method differs, however, in that the sound grain waveform, being so short, is already enveloped and entirely encapsulated in a function table.

A grain represents a kind of cell, a microscopic sound unit with its own parameters. Parameters include: duration, frequency, envelope, rise time, stereo location, waveform. Time delay between successive grains (inter-grain delay) is another crucial parameter.

As is clear, the generation of thousands of grains per second requires a huge number of calculations, that can be difficult to handle. A practical solution is achieved by implementing higher-level control structures, through which you adjust all data according to a global approach. Typical controls including preset values, maybe ramped or randomly scattered over a set range, according to "tendency masks" or any other approach that can generate a very large number of control values. For example, we can set the grain frequency to 220 Hz and have each successive grain change in a random manner. The amount of "randomness" can be given set boundaries. If the variation range is +/- 10 Hz (20 Hz), the frequency will change, randomly, between 210 and 230 Hz. Other grain parameters could be randomized as well, including duration (we can specify an average duration and a random variation range) and inter-grain delay. As an alternative to inter-grain delay, we could define a *density* parameter. For example, if we specify a fixed grain duration of 50 milliseconds, and a density of 15 grains per second, that automatically determines an inter-grain delay of 16.7 milliseconds (fig.15-3, top). A density of 20 grains per second would give a null (0) delay (fig.15-3, center), while a density of 30 would give a negative delay, resulting in overlapping grains (fig.15-3, bottom).

To start, let's create a granular synthesis orchestra using sine wave grains. The grain can be generated using...

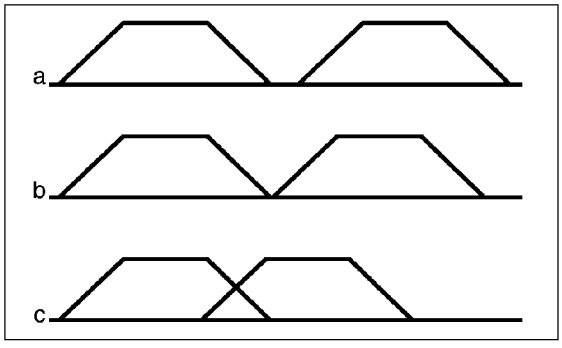


Fig. 15-3

Other paragraphs in this chapter:

15.2 THE *GRAIN* OPCODE

15.3 THE *GRANULE* OPCODE

15.4 FORMANT WAVE SYNTHESIS (*FOF*)

15.5 SNDWARP

LIST OF OPCODES INTRODUCED IN THIS CHAPTER

ar grain amp, freq, density, amp_offset, freq_offset,
waveform_table_number, envelope_table_number, max_grain_dur

ar granule amp, number_of_voices, speed_ratio, speed_mode, amp_thrshld,
function_table_no, pitch_transpose , initial_skip, random_skip,
duration, inter-grain_delay, random_delay, grain_dur,
random_grain_dur, rise_time, decay_time [,seed] [, pitch_shift1]
[,pitch_shift2] [,pitch_shift3] [,pitch_shift4] [, envelope_func_table]

ar fof amp, fundamental, formant_freq, octave_index, formant_bandwidth,
fof_rise_time, fof_dur, fof_decay_time, overlap_factor, fof_function_no,
envelope_table_no, duration[, initial_phase] [, frequency_mode]

asig[, acmp] sndwarp amp, dur_scale [or time_index], freq_scale, sampled_sound_table_no,
start_point, size, random_size_range, overlap_windows,
envelope_window, itime_mode (=switch between dur_scale and time_index)

16

PHYSICAL MODELING SYNTHESIS

16.1 INTRODUCTION

Physical modeling is a powerful tool for the production (or reproduction) of sounds that closely resemble real musical instruments. Even when it is used for the synthesis of non-imitative sounds, it retains a realistic quality. It differs from other digital synthesis techniques in that physical modeling attempts to simulate mechanical sound-generating systems, not the sounds themselves. The approach embodies the mathematical reconstruction of the physical properties of the modeled system.

There are several modeling styles:

1. Mass/spring paradigm (Hiller and Ruiz, 1971). This requires a precise description of the physical characteristics of vibrating objects: length, width, thickness, mass, elasticity, etc. Furthermore, it requires that we stipulate what physicists call *boundary conditions* which are the limits to which a vibrating object is constrained. Finally, it requires that we have a mathematical description of the excitation mechanism (a force impressed on the elastic object), so that we can study its effect on the system. This is usually represented by *difference equations*, thereby obtaining a *wave equation* that describes the resulting sound signal.

2. Modal synthesis (Calvet, Laurens and Adrien, 1990). This approach is based on the premise that a vibrating object can be represented as a collection of simpler elements

(also called *substructures*), including the bridge and the bow on the violin, or the drum heads and the drum body on percussion instruments. Each substructure is characterized by its own *modes of vibration*, which are captured as a collection of *modal data*. Usually, modal data include the frequencies and the damping coefficients for the modes of vibration of the substructure under consideration. Modal data also include a set of coordinates representing the vibrating mode's shape. The peculiar advantage of this approach is the modularity of the substructures. For example, we could experiment with a drum head striking a cello string, and other such "unnatural" situations. The method was incorporated into synthesis programs like MOSAIC and MODALYSE.

3. MSW synthesis (McIntyre, Schumacher, and Woodhouse, 1983). This approach is based on a precise study of the birth and propagation of waves, and on an exact characterization of the physical mechanisms behind the sound phemomenon. The physical model is usually partitioned into two constituent elements: a *non-linear excitation* mechanism (an oscillator with a waveshaper block simulating the behavior typical of, perhaps, a reed, whose output sound features peculiar distortions) and a linear resonator (a filter of variable complexity).

4. Waveguide model (Smith, Cook, et al. 1982-1993). This is the only physical modeling approach that has been used in commercial applications. It is the basis of synthesizers manufactured by Yamaha, Korg, and Roland. The waveguide theory is quite complicated, and is far beyond the scope of this short introductory chapter. For an in-depth explanation, please refer to the scientific literature in the references.

Other paragraphs in this chapter:

17

CSOUND AS A PROGRAMMING LANGUAGE

17.1 CSOUND IS A PROGRAMMING LANGUAGE

In every respect, Csound is a computer programming language. Although it is heavily oriented towards sound synthesis (it was specially designed for that purpose), nothing really prevents us from using it for any other kind of calculation. Try running the program from the following orchestra and score:

```
;calcul.orc
        sr      =   100
        kr      =   1
        ksmps   =   100
        nchnls  =   1

        instr       1
        i1      =   log(10) ; assign the natural logarithm of 10 to the i1 variable
        print       i1      ; display i1 (see section 17.5 on print)
        endin

;calcul.sco
i1      0   .01
```

This example simply prints the value 2.303 (the logarithm of 10) on screen. In this piece of code, you will have surely noticed some oddities, like *sr* = 100 Hz. In truth, that sampling rate is too large. No sound file is generated here. The control rate is quite low (1 Hz!). As a matter of fact, no control variable is featured in the orchestra, so any control rate would be fine. The "duration" is 1 centisecond, but could be any other.

Just like any other programming language, Csound is very powerful for certain tasks (sound synthesis) while not particularly suited to others (data management and manipulation, etc.). However, we should also add that it allows us to create "intelligent" programs, capable of changing their behavior upon input data changes.

17.2 PROGRAM FLOW MODIFICATIONS. VARIABLE TYPE CONVERTERS

Normally, Csound opcodes are executed sequentially, from the first to the last one. However, it is possible to change the program flow. For example, some opcodes could be executed only when some condition is satisfied.

The most important program control statements include the *unconditioned branch* statements:

igoto label
kgoto label
goto label

igoto is an unconditioned branch to the statement labeled by *label* (initialization time only, see section 1.A.1).
kgoto is identical, but works at control rate statements.
goto works both at initialization time and control rate.

Notice that no program control statements apply at audio rate.

Special statements allow for the modification of the program flow dependent upon the occurrence of some other event.

These are called *conditioned branches*, including:

if ia COND ib igoto label
if ka COND kb kgoto label
if ia COND ib goto label

Here *ia* and *ib* are expressions, while COND is a general name for relational operators such as:

>	(greater than)
<	(lesser than)
>=	(either greater than and equal to)
<=	(either lesser than and equal to)
==	(equal to)
!=	(other than)

Notice that the symbol "=" means a value assignment (e.g. *a1 = 12*), while "==" is a relational operator (*13==13* is TRUE, *12==13* is FALSE).

As an example, consider the following code:

```
...
        if      i1>i2   goto jump
a1      rand    iamp
        goto    alright

jump:
a1      oscil   iamp, ifrq, 1

alright:
...
```

There the program control branches to the line labeled *jump*, and continues thereafter, only if *i1* is greater than *i2*. The branch causes the *a1* variable to be generated with *oscil* rather than *rand*. If the conditional branch does not occur, the program executes *rand* and skips to the *alright* label, such that *oscil* is not executed. The program flow is illustrated in figure 17-1.

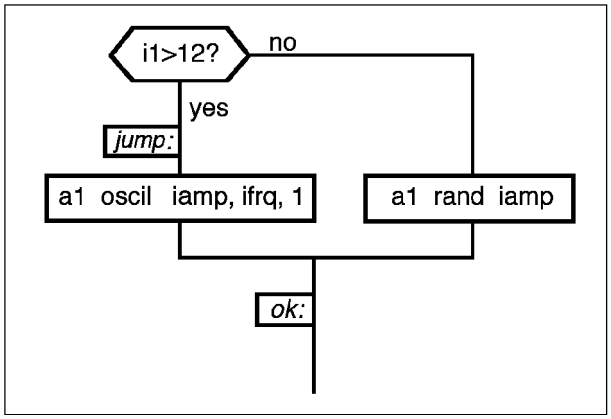


Fig. 17-1

For efficiency reasons, no conditional branches are available at audio rate. Therefore, in Csound a line such as the following is not permitted:

```
if      a1<a2       goto  joe
```

But what if we really need a branch conditioned by audio rate variables? The only way out of that problem is to set the control rate as high as the audio sampling rate, and use special opcodes for variable type conversions:

```
        sr      =   44100
        kr      =   44100
        ksmps   =   1
        nchnls  =   1
        instr       1
a1      oscil       10000, 440, 1
a2      oscil       10000, 458, 1
k1      downsamp    a1
k2      downsamp    a2
        if          k1<k2   goto    joe
        ...
```

Csound variable type converters include:

```
i1      =           i(ksig)
i1      =           i(asig)
k1      downsamp    asig[, iwlen]
a1      upsamp      ksig
a1      interp      ksig[, istor]
```

i(ksig) and *i(asig)* return init-time variables equal to, respectively, control rate and audio rate variables. In a sense, they take a “snapshot” of either control or audio signals and hold it for the entire note duration.

downsamp converts an audio rate variable, *asig*, into a control rate variable, *k1*. *upsamp* and *interp* do the opposite. But, while *upsamp* samples and holds the current *ksig* value, *interp* linearly interpolates between the current *ksig* value and the previous one.

iwlen is the length in samples of an internal window over which the signal is averaged to determine a downsampled value. The maximum length is *ksmps*. Values 0 and 1 imply no averaging. The default is 0.

Other paragraphs in this chapter:

17.3 RE-INITIALIZATION

17.4 PROLONGING THE NOTE DURATION

17.5 DEBUGGING

17.6 MATHEMATICAL AND TRIGONOMETRIC FUNCTIONS

17.7 CONDITIONAL VALUES

EXTENSIONS

17.A.1 DESIGNING COMPLEX EVENTS

LIST OF OPCODES INTRODUCED IN THIS SECTION

	igoto	**label**
	kgoto	**label**
	goto	**label**
	if	**ia COND ib igoto label**
	if	**ka COND kb kgoto label**
	if	**ia COND ib goto label**
i1	**=**	**i(control-rate_variable)**
i1	**=**	**i(audio_rate_variable)**
k1	**downsamp**	**audio_variable [, window_size]**
a1	**upsamp**	**control_variable**
a1	**interp**	**control_variable [, memory_initialization_flag]**
	reinit	**label**
	rireturn	
	timout	**init_time, duration, label**
	ihold	
	turnoff	
	print	**init-time_variable_1 [,init-time_variable_2,...]**
	display	**variable, time [, wait_flag]**
	dispfft	**variable, time, window_size [,window_type][, amp_units] [, wait_flag]**
	printk	**time, control_variable [, number_of_blanks]**
	printk2	**control_variable [, number_of_balnks]**

ir	**pow**	**basis, exponent**
kr	**pow**	**basis, exponent, [scale_factor]**
ar	**pow**	**basis, exponent, [scale_factor]**

APPENDIX 1

WCSHELL

A1.1 WHAT IS WCSHELL?

WCShell is a program created by Riccardo Bianchini to simplify using Csound on Windows95/98/2000/NT computers. It is the next generation of a previously designed MS-DOS application called CShell. The present WCShell release (5.4.1) was developed from a number of preceding releases created for Windows 3.x.

WCShell includes a command console enabling you to select among available disk units and folders, and to load orchestra, score and sound files. It also includes a text editor to write the orchestra and score code, and a number of buttons to launch Csound and listen to sound files. Also included with WCShell is a number of utility programs.

WCShell was written in Microsoft™ VisualBasic© 5.00. The current release works under Windows95/98/2000 and WindowsNT. It is a shareware product, and can be freely copied and circulated.

A1.2 THE MAIN PAGE

The main page of WCShell is printed in figure A-1-1. It includes:

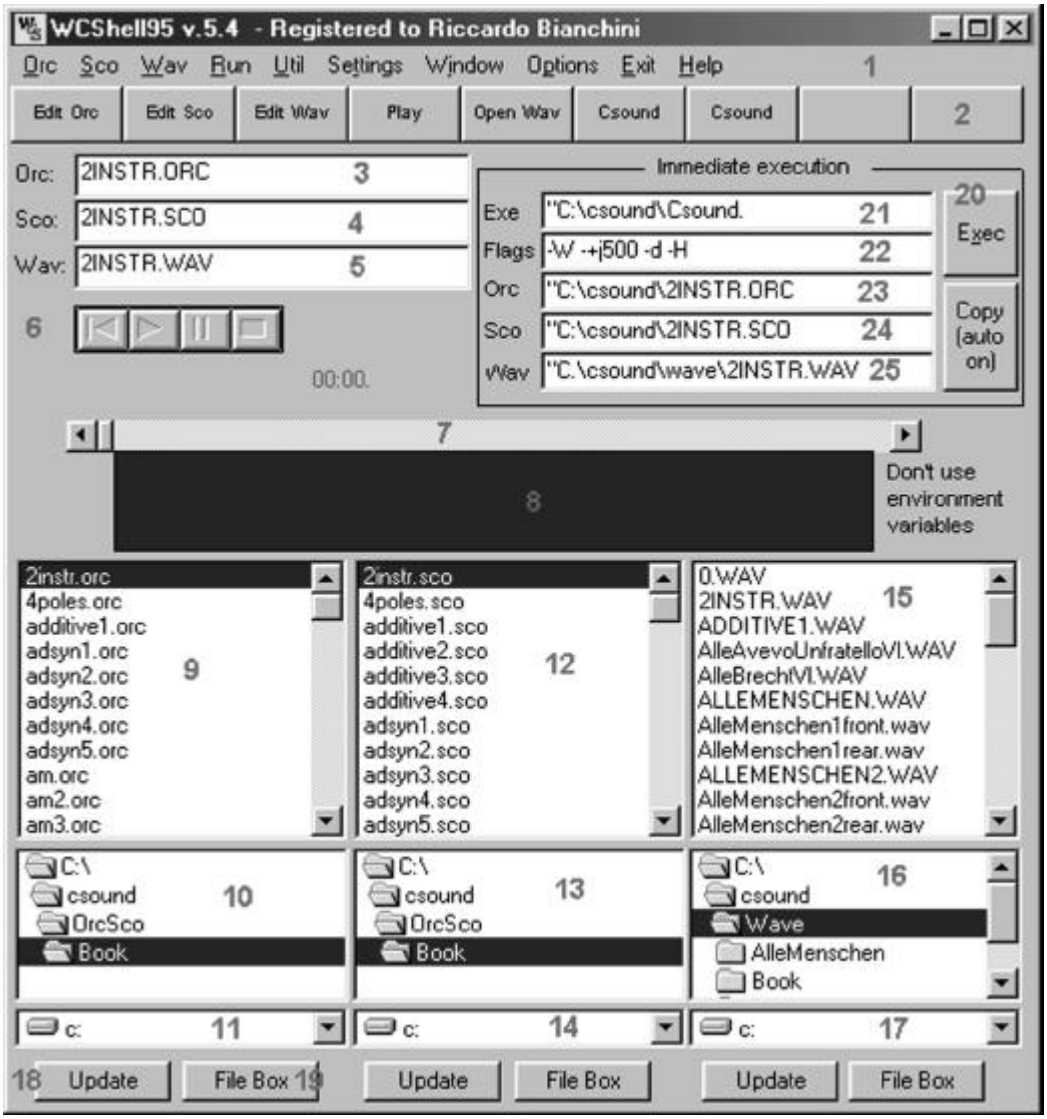


Fig. A-1-1

1. menu bar
2. command buttons
3. filename of currently selected orchestra
4. filename of currently selected score
5. filename for the output sound file
6. tape-recorder buttons (play, stop, rewind, pause)
7. a scroll bar to browse across the current sound file

8. the waveform (amplitude/time graph) of the currently selected sound file
9. orchestra file list-box (available orchestras in current orchestra folder)
10. orchestra folder list-box (available folders on current disk drive)
11. orchestra disk drive
12. score file list-box (available scores in current score folder)
13. score folder list-box (available folders on current disk drive)
14. score disk drive
15. sound file list-box (available sound files in current sound file folder)
16. sound file folder list-box (available folders on current disk drive)
17. sound file disk drive
18. update buttons (update contents of file, directory and disk drive list-boxes, as relative to orchestras, scores or sound files)
19. explore resource buttons (to launch the file manager facility currently installed on your computer)
20. start button (to run Csound from selected files)
21. currently selected Csound executable file
22. Csound command line flags
23. filename for the Csound sound output
24. orchestra file pathname
25. score file pathname

Most important are the main menu bar, the command buttons (for quick access to commands also accessible from the menu), and three text boxes with the filenames of currently selected orchestra, score and sound file. Also important are the playback buttons (similar to a cassette tape recorder), a scroll-bar to move across the sound file, three separate list boxes (for orchestra, score and sound files) with the respective folder and disk-drive. These latter allow you to browse the hard disks on your computer and locate the files.

A.1.3 INSTALLATION FROM CD

It is quite simple to install WCShell from the free CD-ROM packaged with this book. Before you proceed, however, we recommend that you read the readme.txt file for information concerning updates and changes. To install WCShell, do the following:

1. Close all applications currently in use
2. from the Start button, choose the “Run” dialog box and enter the following path and file name: “D:\WCSHELL\SETUP.EXE” (replace drive D with any other drive letter, as necessary). As an alternative, you can browse the CD-ROM drive

and locate the *setup.exe* file. Now run the setup program. During the installation, we recommend that you choose "C:\CSOUND" as the destination folder.

3. When you're done with the installation, run WCShell and select the "Settings". That opens a dialog box where you can enter the settings for correct use of the program. See figure A-1-2.

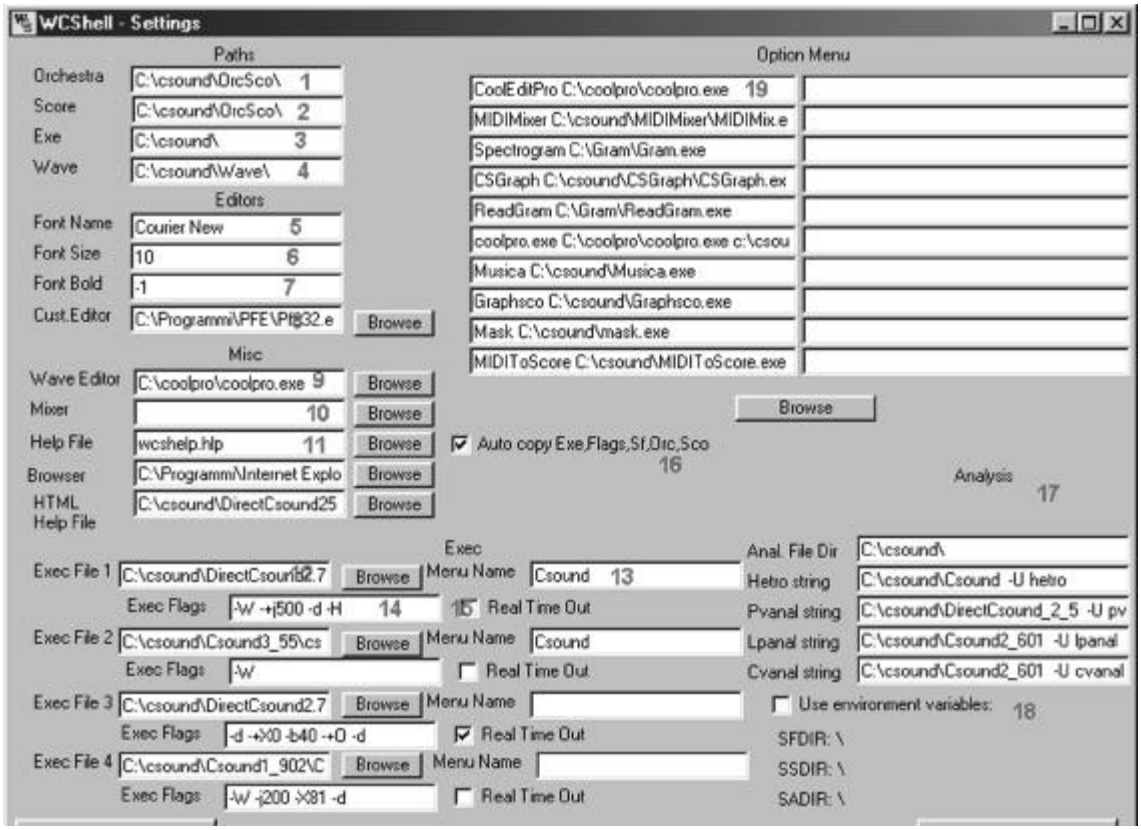


Fig. A-1-2

A.1.4 INSTALLATION FROM INTERNET

To install WCShell from your Internet browser, follow these steps:

1. create a temporary folder, such as C:\TEMP on your hard drive
2. download the *wscshell.zip* file, or *wcshellXX.zip* if available (for example *wcshell51_a.zip*, or *wcshell51_b.zip* would be subsequent versions of the 5.1 release).
3. unzip the downloaded file, and save the extracted files to the same temporary folder as the zip file

4. locate and run *setup.exe*, then follow the same instructions as in the preceding section (2 and 3).

A.1.5 WCSHELL STARTER

The first thing to do is to tell WCShell the folder where it should load and save files during your work sessions. Select "Settings" from the main menu. In the appropriate text boxes, enter the folder pathname for orchestra files (1), score files (2), executable files (3) and sound files (4). Enter the folder where WCShell should locate the Csound executable (normally c:\csound\csound.exe) (12). Enter the string "Csound" (13) that will then appear on the start synthesis button. Enter the flag "-W" as the sole *Exec Flag* (14). Output sound files will be saved in Wave format.

Now click *Write Settings* and *Exit*.

At this point, we can really start a WCShell session. Select Orc and then *New Orc* to launch the orchestra text editor. Write your orchestra code and save it with *File/Save as...*, then close the editor with the *File/Exit* command. Create your score code first selecting *Sco/New Sco*, then following the same procedure you used to create the orchestra file.

As soon as you exit the score editor, you can launch Csound by clicking the start button (now indicated by the button name "Csound"). You'll see a dialog box where you can enter more command line flags, change the output filename, etc. If no change is necessary, click the *OK* button. Csound is now running. Provided no syntax errors are found, the synthesis terminates without problems. Close the synthesis monitor window by pressing Enter. Click the *Play* button (or press F3) to listen to the newly generated sound file. Quite simple, isn't it?

To modify the orchestra, click the *Edit Orc* button or press F1. To modify the score, click the *Edit Sco* button or press F2.

A.1.6 THE ORCHESTRA EDITOR

The orchestra editor option menu includes the following:

FILE

New	remove the current contents from the editor window. If the file was not saved, the program asks for your confirmation.
Open	open an existing file
Save	save file
Save as...	save file with a new name
Insert	load the contents of another file and paste them to the cursor position

Append load the contents of another file and paste them to the end of current file
Print send the editor contents to the printer
Exit leave editor. If the file was not saved, the program asks for your confirmation.

EDIT

Copy copy selection to the clipboard
Paste paste the contents of the clipboard to the cursor position
Goto line... move cursor to line number...
Goto Tag <CsOptions> jump to the first command line flag in a csd file
Goto Tag <CsInstruments> jumps to the beginning of the orchestra code in a *csd* file
Goto Tag <CsScore> jump to the beginning of the score code in a *csd* file

FONTS open a dialog box to choose a character type, size and attributes

FIND

Find find a text string
Find next find next occurrence of the text string
Find Insno… find instrument number... (for this to properly work, the instr opcode and the instrument number must be separated by a tab)

OPTIONS

Header allows you to automatically insert any of the following standard headers:
22050 Hz Mono
22050 Hz Stereo
44100 Hz Mono
44100 Hz Stereo
48000 Hz Mono
48000 Hz Stereo
Conversions allows you to insert any of standard value converters

WINDOWS

Strip Windows strip the edit windows one on top of the other, horizontally
Tile Windows arrange the edit windows one next to the other, vertically

Close Editors close editors in use
Main bring the main WCShell window in foreground

CSD TAGS

<CsoundSynthesizer> insert the <CsoundSynthesizer> tag
</CsoundSynthesizer> insert the </CsoundSynthesizer> tag
<CsOptions> insert the <CsOptions> tag
</CsOptions> insert the </CsOptions> tag
<CsInstruments> insert the <CsInstruments> tag
</CsInstruments> insert the </CsInstruments> tag
<CsScore> insert the <CsScore> tag
</CsScore> insert the </CsScore> tag

SPECIAL

Reread the line feed reread a UNIX file (*line feed* without *carriage return*)
OVERWRITE ON/OFF switch between "insert" and "cancel" editing modes

A.1.7 SCORE EDITOR

The score editor option menu includes the following options:

FILE

New remove the current contents from the editor window. If the file was not saved, the program asks for your confirmation.
Open open an existing file
Save save file
Save as... save file with a new name
Insert load the contents of another file and paste them to the cursor position
Append load the contents of another file and paste them to the end of current file
Print send the editor contents to the printer
Exit leave editor. If the file was not saved, the program asks for your confirmation.

EDIT

Copy copy selection to the clipboard

Paste paste the contents of the clipboard to the cursor position
Comment selected allows you to insert a comment string for the selected text (semicolons ";" are automatically inserted to separate code lines from comments)
Uncomment selected cancels all semicolons ";" from selected text

FONTS open a dialog box to choose a character type, size and attributes

FIND

Find find a text string
Find next find next occurrence of the text string

OPTIONS

Insert Sine Function automatically insert the line *f1 0 4096 10 1* at the beginning of score
Draw Function open a special window where you can draw functions (the function table will be generated with either GEN07 or GEN08, see next section)
View Selected Function allows you to view a function table created with GEN07; the corresponding *f* statement line must be previously selected and copied into the clipboard

WINDOWS

Strip Windows strip windows one on top of the other, horizontally
Tile Windows arrange windows one next to the other, vertically
Close Editors close editors in use
Main bring the main WCShell window in foreground

SPECIAL

Reread the line feed reread a UNIX file (*line feed* without *carriage return*)
OVERWRITE ON/OFF switch between "insert" and "cancel" editing modes

A.1.8 DRAWING FUNCTIONS

The *Draw Function* option in the Score Editor opens a windows like the one in fig.A-1-3. It allows you to graphically render a function with GEN07 and GEN08.

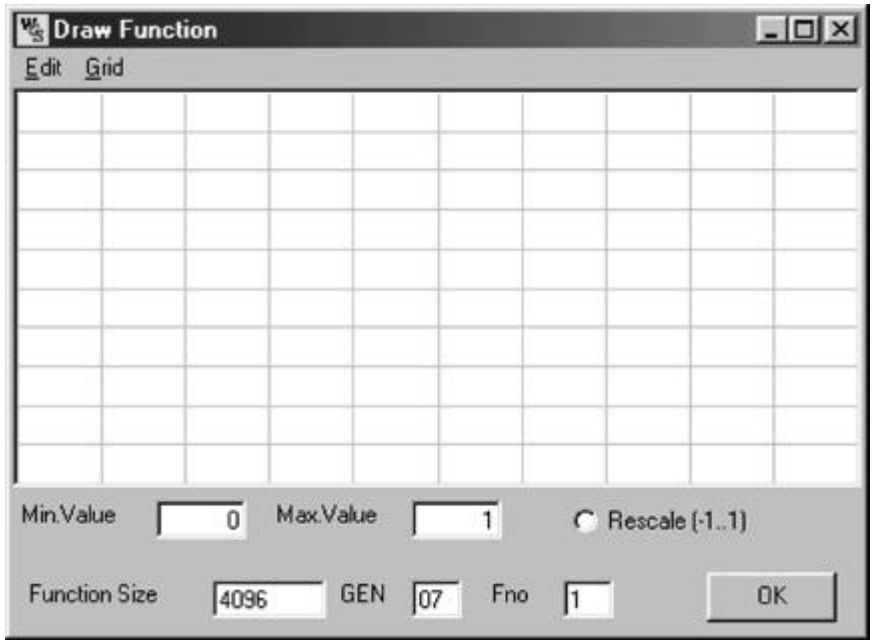


Fig. A-1-3

You can define a value range by entering the appropriate values in the *Min Value* and *Max Value* text boxes (the default is 0 and 1). Also, you can define a table size (default is 4096), the particular GEN to use (default is GEN07), and the table id number (default is 1). Finally, you can decide whether the function table should or should not be normalized (re-scaled).

To draw a function, you move the cursor to the desired horizontal/vertical coordinates and click. That creates the first break point in your function. Each new click will create a straight line linking the new break point to the previous one. Trying to move back in the horizontal direction makes no sense.

From the option menu you can choose *Grid* to create a grid with any number of vertical and horizontal lines, which provide a visual aid to help you draw functions.

The *Edit* option allows you to undo previously drawn line segments (you can undo as many segments as were created since you started drawing).

A.1.9 SCORE PROCESSING

From within WCShell you can call up and run *Scorex*, an application designed to convert a Csound score (with p-fields separated either by tabs or blanks) to a spreadsheet, similar to the old *Visicalc* ™ or the newer *Excel* ™ applications. In the spreadsheet editor, rows correspond to notes or functions, columns correspond to p-fields. It is then possible

to perform calculations or to modify a single cell in a row or any number of selected cells in rows and columns.

Featured operations are:

Add	add a given quantity to selected cells
Mult	multiply the selected cells by a given quantity
Lin Interp	perform linear interpolation between first and last selected cells
Exp Interp	perform exponential interpolation between first and last selected cells
Func	apply a given function (exp, log, int, frac, sine, cosine, tangent, abs) to selected cells
Fill	enter a text string in selected cells
Rand Add	add a random quantity, in a specified range, to selected cells
Rand Mult	multiply the selected cells by a random quantity, in a specified range.

When you call any of these operation, a dialog box prompts you and asks you to specify the required quantity or ranges.

WCShell and *Scorex* provide many other useful options, that you can learn on your own by consulting the on-line help.

APPENDIX 2

MATHEMATICS AND TRIGONOMETRY

A.2.1 FREQUENCY VALUES FOR THE EQUALLY-TEMPERED CHROMATIC SCALE

octaves	0	1	2	3	4	5	6	7
C	32.7032	65.4064	130.8128	261.6256	523.2511	1046.5023	2093.0045	4186.0090
C#	34.6478	69.2957	138.5913	277.1826	554.3653	1108.7305	2217.4610	4434.9221
D	36.7081	73.4162	146.8324	293.6648	587.3295	1174.6591	2349.3181	4698.6363
D#	38.8909	77.7817	155.5635	311.1270	622.2540	1244.5079	2489.0159	4978.0317
E	41.2034	82.4069	164.8138	329.6276	659.2551	1318.5102	2637.0205	5274.0409
F	43.6535	87.3071	174.6141	349.2282	698.4565	1396.9129	2793.8259	5587.6517
F#	46.2493	92.4986	184.9972	369.9944	739.9888	1479.9777	2959.9554	5919.9108
G	48.9994	97.9989	195.9977	391.9954	783.9909	1567.9817	3135.9635	6271.9270
G#	51.9131	103.8262	207.6523	415.3047	830.6094	1661.2188	3322.4376	6644.8752
A	55.0000	110.0000	220.0000	440.0000	880.0000	1760.0000	3520.0000	7040.0000
A#	58.2705	116.5409	233.0819	466.1638	932.3275	1864.6550	3729.3101	7458.6202
B	61.7354	123.4708	246.9417	493.8833	987.7666	1975.5332	3951.0664	7902.1328

Other paragraphs in this chapter:

READINGS

CSOUND AND LINUX

by Nicola Bernardini

GENERATING AND MODIFYING SCORES WITH GENERAL PURPOSE PROGRAMMING LANGUAGES

by Riccardo Bianchini

DYAD CONTROLLED ADDITIVE SYNTHESIS

by James Dashow

SOUND SYNTHESIS BY ITERATED NONLINEAR FUNCTIONS

by Agostino Di Scipio

1. INTRODUCTION

The iteration of nonlinear functions is part of the branch of mathematics called “chaos theory”. In 1991, I set out to investigate the possibility of using it in the field of digital sound synthesis. It was soon clear that the approach was to open up a world of unique sounds of its own, many with turbulent and noisy components (aperiodicity), and some with more familiar harmonic sound spectra (periodicity).

This present overview is a survey of iterated function synthesis based on Csound examples that the reader can analyze and modify for him/herself (readers willing to go into the theoretical and mathematical details are referred to the references). If you try to extend the examples provided here, it will be evident that even slight changes in the synthesis parameters give rise to dramatic differences in the sound, such that often the audible result can hardly be foreseen at the outset. Which suggests that the best way to deal with iterated nonlinear functions is with a kind of empirical, explorative and open-ended attitude. Indeed, the sound synthesis approach described here is a “non-standard” approach. It is not grounded on any model of scientifically proven and verified acoustical relevance.

Rather, it represents a somewhat arbitrary, procedural model which is properly understood as an interesting way of creating sequences of numbers such that, by sending these numbers to a digital-to-analog converter, sounds may eventually emerge having properties that are useful for music and sound-design.

2. GENERAL DESCRIPTION

To iterate a function is to apply some transformation, *f*, to a datum, *x(0*), and to apply the transformation again to the first resulting value, and then again to the second resulting value, ... and so on, *n* times again:

x(n) = f (x(n-1))

We call *x(n*) the *n*th “iterate” obtained by applying *f* to *x(0*). If *f* is nonlinear (e.g. a sine, a line broken in several segments, or any high-order polynomial) the process will determine different sequences of results. The particular sequence is dependent on the initial datum, *x(0*) and on the parameters of the function *f*. In most cases, it is impossible to predict the output series of values.

To translate this general procedure into a digital sound synthesis method, we can follow these steps:

a - initialize *x(0)* and *f'* parameters
b - take the *n*th iterate, *x(n)*, and save it as the current digital sample
c - update *x(0)* and *f'* parameters
d - repeat b and c as many times as the samples required.

In other words, the output stream of samples is the series of the *n*th iterations of *f* upon changing values in *x(0)* and *f*. If we call *i* the sample order index (discrete time index), the synthesis technique can be represented as a simple recursive formula:

x(n,i) = f(i) (x(n-1,i))

This model framework represents a class of synthesis techniques rather than a single technique. The aspect that is peculiar to all particular cases in the class is the fact that the stream of samples is calculated as the sequence of the *n*th iterations of some function. To implement a particular technique, we must select a particular nonlinear function and run a given number of iterations.

As an example, think of the technique known as *waveshaping* (often called *nonlinear distortion* in Europe). It involves the transformation of an input signal by a waveshaping function (usually a Chebychev polynomial, but can be a sine wave or something else). If the operation is repeated, feeding the output sample back into the waveshaper, we get a special case of iterated function synthesis.

Obviously, every nonlinear function determines a peculiar process of its own. However, in the literature on theory of deterministic chaos, many have stressed that the numerical sequences obtained with iterated functions are more heavily dependent on the iteration itself, rather than on the function. It is the iteration that allows coherent or chaotic patterns to emerge, not the nonlinear function being iterated.

Other paragraphs in this reading:

3. IMPLEMENTATION

BIBLIOGRAPHICAL REFERENCES

GSC4: A CSOUND PROGRAM FOR GRANULAR SYNTHESIS

by Eugenio Giordani

1. Introduction

The notion of a Granular Synthesis Csound patch was initially stimulated by the work of Barry Truax. During the Italian seminar, Musica/Complessità, held in the summer 1988, he presented material concerning his research and musical compositions based on real-time granular synthesis. One of the most didactically inexpensive yet powerful synthesis software packages available at the time was Csound. So, I decided to challenge the Music-N dualism of orchestra and score that is the typical feature of this type of programming language, try to write a simple program for this complex sound synthesis.

Since Granular Synthesis creates many acoustic events (grains) per unit of time, it is not a practical to generate each grain using one note score statements. My goal was to realize an automated process for grain generation processed at the micro-level while preserving control over the synthesis process and overall parameters, as well.

The recent releases of Csound, now include the granulation process (see the opcodes *grain* and *granule*) but at the time this patch was written, no granulation unit was available in the Csound opcodes. The goal of this lecture is to serve as a practical exercise for implementation of a complete sound synthesis algorithm in this language - from the conception of an idea to a working program.

2. General structure of the algorithm and synthesis parameters description

Using GSC4 is possible to generate four independent stereo streams of sonic grains. The number four derives from the demand of minimum vertical grain density and real-time capabilities of personal computers.

According to this scheme, the *orchestra* set up includes four grain generation instruments plus one instrument for the control and one instrument for sound mix and scale:

instr 1, 2, 3, 4 : grain generators instruments
instr 11 : control instrument
instr 21 : sound mix, out and scale instrument

In order to generate a complex sound event, we need to switch on six instruments at the same start time (p2) with the same duration (p3).

The majority of the parameters (up to p13) are utilized by the control instrument, whereas the others instruments each contains only four parameters (p1 to p4).

Control of the granulation process is determined by control functions that describe the evolution in time of the synthesis parameters. Those parameters refer to instrument 11 and are:

1)	center grains duration in ms	p4
2)	random grains duration in ms	p5
3)	center inter-grain delay in ms	p6
4)	random inter-grain delay in ms	p7
5)	grain envelope ramp scale factor in non-dimensional units	p8
6)	center waveform-file frequency in Hz	p9
7)	random waveform-file frequency in Hz	p10
8)	center waveform-file phase or file pointer (normalized)	p11
9)	random waveform-file phase or file pointer (normalized)	p12
10)	overall amplitude (normalized)	p13

Each of these parameters states the function number assigned for the relative parameter. So, during the instrument activation, ten functions (created by some GENi method) must exist inside the score. Referring to the score file included in the Appendix, we can summarize the meaning of those synthesis parameters:

f11 :
the average (center) grains duration is defined by a linear function (GEN 7) with initial value of 10 ms that after 256/512 of p3 (the total event duration) reaches the value of 20 ms, keeps costant for 128/512 and moves to the final value of 16 ms after 128/512 of p3.

f12:
the peak random duration value of the grains is defined by a linear function (GEN 7) with initial value of 4 ms that after 256/512 of p3 reduces itself to 1 ms and after 256/512 goes to the final value of 0 ms (no random deviation).

f13:
the inter-grain delay is defined by a linear function (GEN 7) with initial value of 10 ms that after 256/512 of p3 raises to 20 ms and after 256/512 reaches the final value of 5 ms.

f14:
the peak random inter-grain delay value of the grains is defined by a linear function (GEN 7) with initial value of 0 ms (no random deviation) that after 128/512 of p3 stays constant to 0 ms; after 256/512 raises to the value of 2 ms and after 128/512 reaches the final value of 0 ms (no random deviation).

f15:

grain envelope ramp scale factor is defined by a linear function (GEN 7) with initial value of 2 that after 256/512 of p3 rises to 4 and after 256/512 reaches the final value of 2. Practically, the grain envelope shape change gradually from a triangle at the beginning towards a trapezium and back to triangle. When the envelope shape looks like a triangle, the duration of both the attack and decay ramp is equal to half duration of the whole envelope (no sustain). When the envelope shape looks like a trapezium, the duration of both the attack and decay ramp is equal to one quarter of the whole envelope, so the sustain duration is equal to two quarter of the whole envelope.

f16:

the frequency of the granulated waveform is defined by a linear function (GEN 7) with initial value of 220 Hz that stays constant during the whole event. Ins the score there is a comment line referring to the values of control frequency (from 1.345 to 3.345 Hz) in the case of a granulated sampled waveform (sample.wav) of 32768 samples size instead of a single cycle wave. In this case, the initial value of 1.345 derives from the ratio 44100/32768 and represents the original pitch of the waveform.

f17:

the peak random frequency of the granulated waveform is defined by a linear function (GEN 7) with initial value of 0 hz (no random deviation) that after p3 reaches the final value of 110 hz (50% of frequency modulation).

f18:

the phase (or file pointer) of the granulated waveform is defined by linear function (GEN 7) with initial value of 0 that stays constant over the whole event.

f19:

the peak random phase (or file pointer) of the granulated waveform is defined by linear function (GEN 7) with initial value of 0 that stays constant over the whole event.

f20:

the overall amplitude waveform is defined by a linear function (GEN 7) with initial value of 0 that after 128/512 of p3 rises to 1, stays constant for 256/512 and reaches 0 after 128/512 of p3.

f1:

the granulated waveform is defined by a simple Fourier additive function (GEN 10).

It is important to notice that, except in functions f1 and f20, the parameter p4 is always negative because the function breakpoints must represent their values in an absolute scale. For the four generation instruments (i1,i2,i3,i4), is sufficient to specify (besides the 3 obligatory parameters p1,p2 and p3) the function number (f1 in this case) and for instrument 21 the output scale factor.

For the audio waveform is possible to specify a single cycle wave or a sampled sound using a GEN1 function. In the first case, the value of the parameter p4 of the instrument 21 must contain the maximum output value (in the range 0 , 32767) whereas in the second one, the value is defined in the range 0, 1. The reason of this fact is that the sampled audio signal is not post-normalized during the table reading (GEN -1).

As stated before, is important to point out that the audio signal may be both a single cicle wave or a sampled sound.

Although there is no functional differences in the two cases, it is better to pay attention in the frequency specification.

In the first case, the frequency value and the relative random deviations are simply the desired values.

In the second case, the nominal value of the *natural frequency* (Fn) of the oscillator (by that we mean the frequency value to reproduce the audio signal at the original pitch) is derived from the ratio of the sampling rate (*sr*) and the length (in samples) of the table that contains the sampled waveform (Fn = sr / table length).

For example, if the length of an audio waveform sampled at 44.1 kHz is 64k samples (about 1.486 seconds) , the natural frequency will be 44100 / 65536 = 0.672 Hz.

In general, since the effective duration of the audio signal hardly ever equivalent to a powers of two, we have to provide a table to store the values in excess of the waveform size minus the nearest power of two.

If the total duration of the audio signal sampled at 44.1 kHz is 1.2 seconds, to be effective, the table should be 44.1 x 1.2 = 52920 samples. This implies one should choose a table size of 64 k-samples. However, the natural frequency will still be 0.672 Hz because the Csound oscillator modules work with tables whose length is equivalent to a power of two. The only difference, in this case, is that the phase parameter ranges from 0 and 0.807. The value 0.807 is obtained from the ratio 52920/65536.

It is also important that the random frequency fluctuation must be congruent with the corresponding deterministic value.

For example, with a natural frequency value of 0.672 Hz and a fluctuation of 10 percent the corresponding fluctualtion is about 0.06 Hz.

The concept of *natural frequency* is very important in this context because we can granulate a single cycle of a waveform or a whole sample of sound. From the point of view of the oscillator there is no difference. We can generalize this with the equation:

Fn = I x SR/ L

where :

Fn = oscillator natural frequency
SR = sampling rate
I = increment (table reading step)
L = table length (samples)

With respect to the previous example, if we want to granulate a 1.2 sec stored sound sampled at 44.1 kHz and reproduce it at the original pitch, we have to specify something like

```
f16   0   512  -7   0.672   512   0.672
```

When the granulation is applied only to sampled sounds, it is a good idea to multiply the variable *ifreq* with the expression *sr/ftlen(ifun)*, where *ftlen(x)* is a function that returns the table length in samples. Hence, the score line controlling the frequency will be:

```
f16   0   512  -7   1   512   1
```

In this way, the sound's pitch is handled as a ratio with respect to the natural frequency, and we avoid computing the real values of the frequency itself. For example, to create a continuos pitch glide of the granulated sound, starting from its original frequency and moving up to the natural fifth above (interval ratio of 3:2=1.5), the following score line is required

```
f16   0   512  -7   1   512   1.5
```

The same control is possible for random frequency deviation.

If the audio function is a single cycle of a periodic wave, the phase parameter of the oscillator has an influence on the acoustic result. But if the audio function is a sampled sound, its role is crucial. In fact, in this case, the phase parameter becomes the *table pointer*, allowing us to granulate different sections of the sampled sound.

When you want to granulate the whole sample, from the beginning to the end of sound, we can use a linear function (GEN 7) that moves from 0 to 0.999:

```
f18   0   512  -7   0   512   0.999
```

Here, the sampled sound is granulated without time warp. By exchanging two values, it is possible to reverse the sound. Following this approach, we can achieve different and interesting acoustic results.

For example, in the next score line, during the first half of the event (p3/2), the granulated sound is reversed starting from its middle point (0.5) to the origin, and in the second half, the sampled sound is forward time compressed:

```
f18  0  512  -7  0.5  256  0  256  0.999
```

In any case, the total event duration (p3) may be kept the same as the duration of the original sampled sound or it maybe increased or reduced.

To increase the time transformation of the original sound, one can use a more complex control function (i.e. non-linear) and add an additional random control with the function 19 (see code listing).

3. Origins of the synthesis algorithm

The basic algorithm is based on the model proposed by Barry Truax and implemented on the DMX-1000 real time processor controlled by a host microcomputer. Fig. 1 shows the basic process of the granulation technique in this early implementation. Here, the synthesis generation probably used two programming sections: a background section for the envelope generator bank running on the processor and managed by an interrupt logic (i.e. using a 1 ms timer), and a control section running on the host computer.

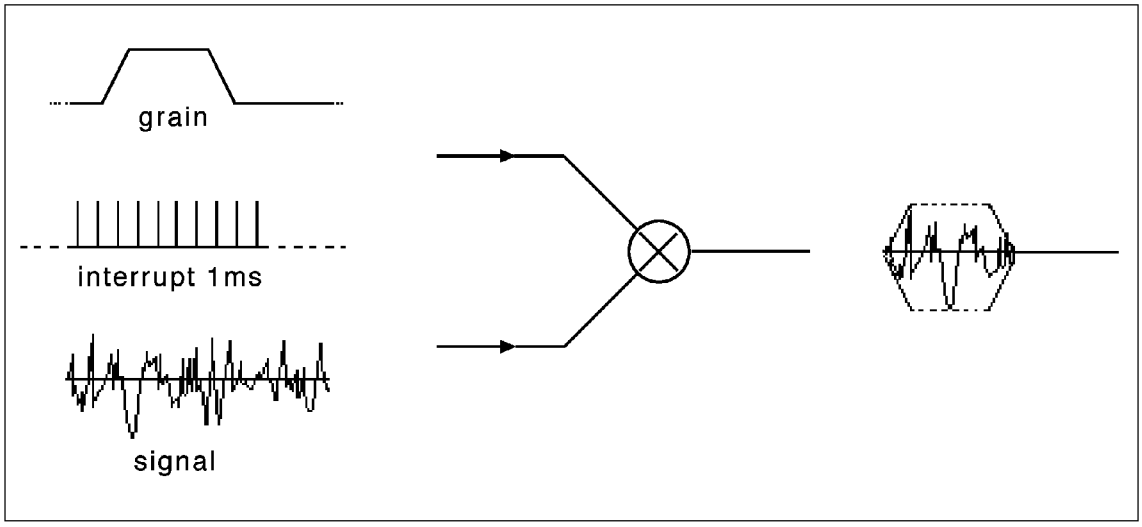


Fig. 1

Since this Csound instrument for the granulation process was implemented in 1989 (an era with few real-time applications for general purpose computers) the main goal of its development was to make a basic strategy for implementing this innovative synthesis technique in a non real-time environment.

Given the inherent necessity to generate a great amount of micro-events per units of time, it was not practical to pursue a note by note generation approach. An alternative solution would have been the utilization of a pre-processing front-end program for the generation of the innumerable lines of notes, each one corresponding to a single grain.

Other paragraphs in this reading:

DIRECTCSOUND AND VMCI:
THE PARADIGM OF INTERACTIVITY

by Gabriel Maldonado

REFERENCES

Backus, J. *The Acoustical Foundation of Music*. New York, New York: Norton

Balena F., De Poli, G. 1987. 'Un modello semplificato del clarinetto mediante oscillatore non lineare', in *Musica e tecnologia: industria e cultura per lo sviluppo del Mezzogiorno*, Quaderni di Musica/Realtà Milano: UNICOPLI

Berry, R.W. 1988. 'Experiments in Computer Controlled Acoustical Modelling (A Step Backwards?)', in *Proceedings of the 14th Computer Music Conference*. Köln: Feedback Papers

Bianchini, R. 1987. 'Composizione automatica di strutture musicali', in *I profili del suono*. Salerno: Musica Verticale-Galzerano

Bianchini, R. 1996. 'WCShell e i suoi software tools per la generazione e la modifica di partiture in formato Csound', in *La terra fertile, Atti del Convegno*. L'Aquila

Cott, J. 1973. *Stockhausen. Conversations with the Composer.* New York, New York

De Poli, G. 1981. 'Tecniche numeriche di sintesi della musica', in *Bollettino LIMB n.1*. Venezia: La Biennale di Venezia

Dodge, C., and Jerse, T. A. 1985. *Computer Music*. New York, New York: Schirmer Books

Forin, A. 1981. 'Spettri dinamici prodotti mediante distorsione con polinomi equivalenti in un punto', in *Bollettino LIMB n.2*, Venezia: La Biennale di Venezia

Gabor, D. 1947. "Acoustical Quanta and the Theory of Hearing", in *Nature*, 159(4044)

IMUG. 1983. *MIDI Specifications*, Los Altos, California: MIDI International User's Group

Moles, A. 1969. "The Sonic Object", in *Information Theory And Esthetic Perception*. Urbana, Illinois: University of Illinois Press

Morresi, N. 1967. *Dispense di acustica applicata*, unpublished

Pousseur, H. (ed.), 1976. *La musica elettronica*. Milano: Feltrinelli

Prieberg, Fred K. 1963. *Musica ex machina*. Torino: Einaudi

Risset, J.C. 1981. 'Tecniche digitali del suono: influenza attuale e prospettive future in campo musicale', in *Bollettino LIMB n.2*. Venezia: La Biennale di Venezia

Risset, J.C., and Wessel, D. 1981. 'Indagine sul timbro mediante analisi e sintesi', in *Bollettino LIMB n.2*, Venezia: La Biennale di Venezia

Roads, C. 1978 'Automated granular synthesis of sound' *Computer Music Journal* 2(2): 61-62

Roads, C. 1985 'Granular synthesis of sound' in C.Roads and J.Strawn, eds.1985. *Foundations of Computer Music*. Cambridge, Massachusetts: The MIT Press

Roads, C., and Strawn, J. (ed.). 1985. Foundations of Computer Music. Cambridge, Massachusetts: The MIT Press

Roads, C. 1995. *The Computer Music Tutorial*. Cambridge, Massachusetts: The MIT Press

Seto, W.W. 1978. *Teoria ed applicazioni di Acustica*, Milano: ETAS Libri

Spiegel, M. R. 1974. *Manuale di matematica*, Milano: ETAS Libri

Smith, J. O. III. 1996. "Discrete-Time Modeling of Acoustic Systems with Applications to Sound Synthesis of Musical Instruments", in *Proceedings of the Nordic Acoustical Meeting*, Helsinki

Strawn, J. (ed.). 1985. *Digital Audio Signal Processing*, Los Altos, California: William Kaufmann Inc.

Tisato, G. 1987. 'Sintesi dei suoni vocali e del canto in modulazione di frequenza', in *Musica e tecnologia: industria e cultura per lo sviluppo del Mezzogiorno*, Milano: Quaderni di Musica/Realtà - UNICOPLI

Truax, B. 1988. 'Real-Time Granular Synthesis with a Digital Signal Processor', in *Computer Music Journal*, 12(2): 14-26. Cambridge, Massachusets: Summer M.I.T Press

Vercoe, B. 1986-1992. *Csound Manual*. Cambridge, Massachusetts: Media Lab - MIT

Wiener, N. 1964. "Spatio-Temporal Continuity, Quantum Theory, and Music". printed in *The Concepts Of Space And Time*, Boston Studies XXII, M.Capek (Ed.). 1975. Boston, Mass: D.Riedel

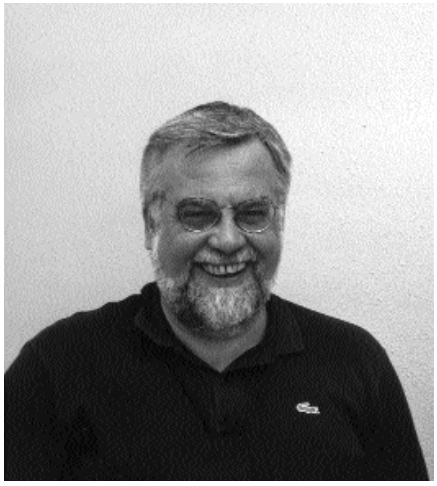
Xenakis, I. 1971. *Formalized Music*, Bloomington, Indiana: Indiana U.Press

CSOUND WEB SITES

1. MAIN SITES

2. SOFTWARE

3. UNIVERSITIES, RESEARCH CENTERS AND ASSOCIATIONS



Riccardo Bianchini
Milan, 1946

e-mail: rb@fabaris.it
www.fabaris.it/bianchini

Bianchini studied Piano, Composition, Electroacoustic Music, and Engineering at the Politecnico University. Since 1974 he has been Professor of Electroacoustic Music in the Conservatories of Pescara, Milan, and since 1987 in Conservatorio "S.Cecilia" in Rome.

He translated in Italian many important books on music, such as C.Rosen's "The Classical Style" and "Sonata Forms", His articles have been published in music reviews (Rivista IBM, Musica, Perspectives of New Music, etc.).

Since 1995 he is visiting professor in Universities of Uruguay and Argentina.

As a software writer, he is the author of many music related programs.

Since 1983 to 1991 he collaborated with RAI (Italian State Radio).

His compositions (orchestral, vocal, instrumental, electroacoustic, incidental), published and recorded by Edipan and BMG-Ariola, have been performed and/or broadcast in Europe, USA, Cuba, Argentina, Uruguay and Australia.

BIBLIOGRAPHY

Bianchini,R. 1973. "La nuova musica in Italia", in *Storia della musica Oxford-Feltrinelli, Vol.X*. Milan: Feltrinelli
Bianchini, R. 1976. "La musica elettronica", in *Rivista IBM*, 12. Milano
Bianchini, R. 1976. "Musica e letteratura (II)", in *Enciclopedia Feltrinelli Fischer.* Milano: Feltrinelli
Bianchini, R. 1976. "La musica contemporanea", in *Musica 1,2,3,4,5*. Milano
Bianchini, R. 1976. "La musica informatica", in *Musica Domani, 22*. Milano
Bianchini, R. 1985. *Computer Music: manuale di informatica musicale*. Unpublished
Bianchini, R. 1987. "Composizione automatica di strutture musicali", in *I profili del suono*. Salerno: Galzerano

Bianchini, R. 1996. “WCShell e i suoi software tools per la generazione e la modifica di partiture in formato Csound”, in *La terra fertile, Proceedings*. L’Aquila
Bianchini, R, ed. 1996. Cristiano, C. *I territori di Montopoli e Bocchignano*. Montopoli di Sabina.
Bianchini, R. 1999. “La musica elettronica in Italia”, in *Azzurra, 4*. Córdoba: Istituto Italiano di Cultura

TRANSLATIONS

1973. *Storia della musica Oxford-Feltrinelli, Vol.X*. Milan: Feltrinelli (translation of The Oxford History of Music)
Pousseur H., ed. 1975. *La musica elettronica*. Milan: Feltrinelli (translation)
Rosen, C. 1979. *Lo stile classico*. Milan: Feltrinelli (translation of The Classical Style)
Dick, R. 1979. *L’altro flauto*. Milan: Ricordi (translation of The Other Flute)
Rosen, C. 1984. *Le forme sonata*. Milan: Feltrinelli (translation of Sonata Forms)

DISCOGRAPHY

10 storie Zen, for flute, clarinet, vibraphone, piano, viola and cello.
WNC Ensemble. PAN PRC S2062
Roèn, for flute, clarinet, bassoon, horn, violin, viola, cello and piano.
L’Artisanat Furieux Ensemble, dir. T. Battista. PAN CDC 3010
Klimt, for flute, oboe, clarinet, violin, viola, cello, piano and tape.
Romensemble, dir. F.E.Scogna RCA CCD 3001
Machu Picchu, for flute, oboe, clarinet, bassoon, 2 trumpets, french horn, trombone, live electronics and tape.
Farfensemble, dir. R.Bianchini. ED0009

COMPOSITIONS

haiku, (1976, 10:00), piano and tape
Mirror, (1976, 05:00), flute and piano. C.A.S.
Collettivo II, (1976, 05:00), flute, oboe, clarinet, bassoon, violin, cello and piano. EDIPAN
Due racconti, (1979, 05:00), 2 clarinets, basoon, viola, piano. EDIPAN
La nave bianca (Preludio), (1980, 05:00), chamber orchestra. EDIPAN
La nave bianca (incidental music), (1980, 25:00), chamber orchestra and male choir
Roèn, (1982, 05:00), flute, clarinet, bassoon, french horn, violin, viola, cello and piano. EDIPAN

Riyàr, (1982, 05:00), piccolo, flute in C, flute in G, bass flute (1 flutist). EDIPAN
Sedrùna, (1982, 5:00), four hands piano. EDIPAN
Tre quadri immaginari, (1983, 10:00), harp. EDIPAN
Quattro canti: 1. "Di più cupi sentieri" (D.Villatico), 2. "La tierra que era mía" (J.G,.Durán), 3. "I have done" (J.London), 4. "Im wunderschönen Monat Mai" (R.M.Rilke), (1980-1988, 12:00), soprano and piano. EDIPAN
6 Preludi, (1980-1984, 11:00), piano. EDIPAN
Due fogli d'album, (1985, 02:00), flute and piano
Foglio d'album, (1985, 1:30), violin and piano
La principessa senza tempo, (1985, 14:10), flute and tape. EDIPAN
Alias, (1985, 5:00), 2 oboes and basson
Chanson d'aube, (1986, 6:00), 4 trumpets, 4 horns, 4 trombones
Rosengarten, (1986, 05:00) violin and orchestra
Our Faust, (1986, 17:00), clarinet, trombone, double bass and live electronics. BMG-Ariola
Arsól, (1987, 10:00), real time computer and quad tape
Divertimento, (1988, 8:20), for 13 instruments. EDIPAN
Somanón, (1989, 8:00), 11 strings. EDIPAN
Preuss (1989, 16:20), violin, cello and tape. BMG-Ariola
Fànes (1989, 8:00), flute in C and flute in G (1 flutist)
Alberi (1990, 6:00), sax quartet. BMG-Ariola
Saluto a Pablo (1990, 2:00), soprano, fute and clarinet
Cuando sonó la trompeta (1990, 7:30), soprano and tape. BMG-Ariola
Klimt (1991,10:50), flute, oboe, clarinet, violin, viola, cello, piano and tape. BMG-Ariola
Chanson d'aube II (1991,5:00), 2 oboes, 2 clarinets, 2 horns and 2 bassoons
Tre ricercari (1993, 5:00), 2 trunpets, horn and trombone
Machu Picchu (1993, 14:00), flute, oboe, clarinet, bassoon, 2 trumpets, horn, trombone, live electronics and tape
Poche note… per Enzo Porta, (1994, 2:00), violin
6 Preludi (II quaderno), (1994, 11:00), piano
Il contrabbasso poteva non esserci, (1995, 2:30), 2 flutes, oboe, piano and string quartet
Naste, (1995, 2:00), flute and cello
Howl, (1995, 6:30), male or female voice and tape
I dannati della terra, (1996, 28:00), actor, soprano, flute, percussion, tape and image projection
Ghe Xe, (1997, 5:00), flute in G and piano
Aria di Albertine (da "Doppio sogno"), (1997, 4:30), soprano, flute, oboe, clarinet, basoon, piano and string quartet

Canciones para las estrellas, (1997, 6:30), tape
Canciones para las estrellas, (1997, 8:00), piano and tape
Los pájaros del sueño, (1998, 9:00), clarinet and tape
Montevideana, (1999, 5:00), tape
How Deep the Sea, (1999, 5:30), Big Band (4 saxophones, 3 trumpets, 2 trombones, piano, bass and drums)
Alle Menschen werden Brüder, (1999, 7:40), violin, speaker and quad tape

TRANSCRIPTIONS AND REVISIONS

A.Gabrieli, *Ricercar nel duodecimo tono,* (1977), fl,ob,cl,fg,tr,cor,trbn
G.Gabrieli, *Quattro canzoni per sonar a quattro,* (1977), fl,ob,cl,fg,tr,cor,trbn
A.Willaert, *Ricercar X*, (1977), flute, oboe, clarinet, bassoon, trumpet, horn and trombone
J. da Modena, *Ricercar III*, (1977), trumpet, horn, trombone
Anonimi Francesi, *Suite di danze,* (1977), flute, oboe, clarinet, bassoon, trumpet, horn and trombone
H.Pousseur, *Icare apprenti*, (1977), flute, oboe, clarinet, bassoon, trumpet, horn and trombone
F.Schubert, *4 Ländler,* (1993), wind quintet
F.Schubert, *4 Ländler*, (1994), flute, oboe, clarinet, bassoon, string quartet
F.Schubert, *Deutsche Tänze,* (1994), flute, oboe, clarinet, bassoon, 2 trumpets, horn and trombone
W.A.Mozart, *Musiche di palcoscenico da "Don Giovanni"*, (1994), flute, oboe, clarinet, bassoon, trumpet, horn and trombone
H.Purcell, *Suite,* (1995), flute, oboe, clarinet, bassoon, trumpet, horn and trombone
D.Auber, *"Fra Diavolo"*, pot pourri dall'opera, (1995), flute, oboe, clarinet, bassoon, trumpet, horn and trombone
J.Lennon, P.McCartney, *Eleanor Rigby,* (1995), flute, oboe, clarinet, bassoon, 2 trumpets, horn and trombone
J.Lennon, P.McCartney, *Penny Lane,* (1995), flute, oboe, clarinet, bassoon, 2 trumpets, horn and trombone
J.Lennon, P.McCartney, *Yesterday,* (1995), flute, oboe, clarinet, bassoon, 2 trumpets, horn and trombone
J.Lennon, P.McCartney, *Girl*, (1995), flute, oboe, clarinet, bassoon, 2 trumpets, horn and trombone
J.Lennon, P.McCartney, *Lady Madonna*, (1995), 2 trumpets, horn and trombone
Bela Bartók, *Danze Rumene,* (1993), wind quartet
Bela Bartók, *Danze Rumene,* (1994), flute, oboe, clarinet, bassoon and string quartet

photo by Chris Bitten

Alessandro Cipriani
Tivoli (Rome), 1959

e-mail a.cipriani@virtual-sound.com
www.edisonstudio.it

Cipriani completed his studies in music composition and electroacoustic music at the Conservatorio S.Cecilia in Rome. He studied for a time with Barry Truax. Since 1989 he has worked on intermedia pieces, often in collaboration with visual artist Alba D'Urbano. More recently he has composed electroacoustic pieces with traditional religious singers for a new CD.
His works have received honors and/or have been selected for performance by government arts commissions, professional organizations and festival organizers including Bourges, Government of Canada Award, International Computer Music Conference, International Symposium on Electronic Arts, Musica Nova, Newcomp, etc. He has taught electroacoustic music at the Ist. Mus. V. Bellini in Catania since 1995. He has lectured about his music and his theory of 'electroacoustic tradition' at several Universities in Italy, Canada and the U.S. He has published analytical and theoretical papers in several journals and publications of the proceedings of various conferences.
His CD "Il Pensiero Magmatico" in collaboration with StefanoTaglietti is available on the Edipan label. Other pieces can be found in the ICMC95 and ICMC99 CDs. His music has been broadcast by RAI, CBC and other national radio networks as well as performed at festivals in Europe, Canada, South-America and the U.S.A.
He is one of the co-founders of Edison Studio in Rome.

BIBLIOGRAPHY

Cipriani, A. 1993 "Visibili..." in Atti del X Colloquio di Informatica Musicale, LIM-DSI Univ. degli Studi di Milano, Milano, pp.404-6

Cipriani A. 1993 Due tesi complementari sulle due versioni di Kontakte di K.Stockhausen - Tesi di Diploma in Musica Elettronica - Conservatorio di Musica S.Cecilia - Roma

Cipriani A. 1995 "Towards an electroacoustic tradition?" in Proceedings of the International Computer Music Conference, ICMA, Banff, pp. 5-8

Cipriani A. 1995 "Problems of methodology: the analysis of Kontakte" in Atti del X Colloquio di Informatica Musicale, AIMI , Bologna, pp. 41-44

Cipriani A. 1996 "Verso una tradizione elettroacustica? Appunti per una ricerca" in Musica/Realtà N°49 Marzo LIM Lucca pp.18-24

Cipriani A. 1996 "Tradizione orale, tradizione scritta, tradizione elettroacustica" in Atti del II Convegno La Terra Fertile - Incontro Nazionale di Musica Elettronica - Conservatorio di Musica "A.Casella", L'Aquila

Cipriani A. 1998 “Kontakte (Elektronische Musik) di K.Stockhausen: genesi, metodi, forma” in Bollettino G.A.T.M. anno V, n.1 GATM Univ. Studi Bologna

Cipriani A. 1998 “Musica e Internet: arte come esperienza, arte come codice” in Aperture n.5, Roma

Bianchini R.- Cipriani A. 1998 Il Suono Virtuale, Contempo, Roma

DISCOGRAPHY

A.Cipriani
QUADRO
for string quartet and tape
International Computer Music Conference ’95 – PRCD1600

A.Cipriani-S.Taglietti
IL PENSIERO MAGMATICO
for tape, piano, percussions and mixed choir
EDIPAN – PAN CD 3059

A.Cipriani
AL NUR (La Luce)
for islamic chant, zarb, daf and tape
International Computer Music Conference ’99 – PRCD2000

COMPOSITIONS

#1, #2, #3
Kreis
three works for video and electroacoustic music
Video by Alba D'Urbano
Music by Alessandro Cipriani
(1987-1991)
finalist at Locarno Video Festival (Switzerland)

#4
Kreis: la piazza
sound-video installation
(in collaboration with Alba D'Urbano)
presented at EASA - Berlin Kulturstadt Europas - Esplanade Berlin (Germany) Ago 1988

#5
Circolo Vizioso
sound-video installation
(in collaboration with Alba D'Urbano)
presented for the first time at
Internationaal Audio Visueel Experimenteel Festival 1989

#6
Circoscritto
sound-video installation
(in collaboration with Alba D'Urbano)
presented at
Centro di Video Arte (Palazzo dei Diamanti) - Ferrara within the exhibition "POLISET"
Dic.1991

#7
Luce di due soli
for piano, vibraphone and tape(dur. 25'25")
(1991)
(in collaboration with Giovanni Bietti)
Finalist at 1991 Newcomp Computer Music Competition
(USA) - First Performance
29th Festival of "Nuova Consonanza" - Roma

#8
Visibili
for two violins and tape (dur.8'30")
(1992)
MENTION AT "21e CONCOURS INTERNATIONAL DE MUSIQUE ELECTROACOUSTIQUE" - BOURGES (FRANCE) 1993
First Performance XV Festival Musica Verticale - Roma

#9
Quadro
for string quartet and tape (dur.10'30")
(1993)
INCLUDED IN THE INTERNATIONAL COMPUTER MUSIC CONFERENCE '95 CD
First Performance 18° Cantiere Internazionale d'Arte di Montepulciano

#10
Terra Fluida
for video and electroacoustic music
Video by Alba D'Urbano Music by A.Cipriani
(1991-1994)
MAIN PRIZE AT MUSICA NOVA ELECTROACOUSTIC MUSIC COMPETITION CZECH REPUBLIC RADIO-TV
World Premiere Musica Nova Praha Dec 1996

#11
Recordare
for bass and double-bass recorders and tape (dur.12'24")
(1994-rev.1997)
FINALIST AT 25e CONCOURS INTERNATIONAL DE MUSIQUE ELECTROACOUSTIQUE" - BOURGES (France) 1997
First Performance 3 Nov '94 - Musica Verticale/Progetto Musica '94
Goethe Institut - Roma

#12
L'Acqua, il Musico, Lo Specchio
intermedia work for 5 musicians, two actors, video projections and tape (dur. 1h. 15 ca.')
in collaboration with Giovanni Bietti
(1993-94)
First Performance Dec 3rd '94 - Progetto Musica '94 (Spazi Aperti) Acquario - Roma

#13
In Memory of a Recorder
for tape(15' 45")
(1993-94)
First Performance Festival Animato 1995 Sala Uno - Roma 19 maggio '95

#14
Il Pensiero Magmatico
for magnetic tape, piano, percussions and mixed choir (dur. 53:00)
(1995-96)
in collaboration with Stefano Taglietti, texts by Bizhan Bassiri
World Premiere Oct. 18th, 1996 Musée FRC Le Creux de l'Enfer Centre d'Art Contemporain Thiers Francia

#15
Still Blue
Homage to Derek Jarman
(1996)
for magnetic tape, piano, cello,sax soprano, mime and projected video
First Performance Freon Ensemble - Progetto Dyonisos
Nov. 29th '96 - Acquario Romano - Progetto Musica '96

Religious Chant Trilogy (#16 - #19 - #21)

#16
Angelus Domini
(1996)
for gregorian chant and magnetic tape
First Performance Musica Verticale/Progetto Musica '96
Dec. 9th '96

#19
Al Nur (La Luce)
(1997)
for islamic chant, persian percussions, magnetic tape and projected video
First Performance Nov. 12th, 1997
Musica e Scienza /Progetto Musica '97 Goethe Institut – Roma
INCLUDED IN THE INTERNATIONAL COMPUTER MUSIC CONFERENCE'99 CD

#21
Mimahamakim
(1999)
for jewish chant and tape

#17
Pensiero Magmatico
(1997)
for video and magnetic tape
Video by Bizhan Bassiri
Music by Alessandro Cipriani
First performance Galleria Miscetti
March 1997 Roma

#18
Quem quaeritis non est hic
(1997)
Sound installation for quadraphonic magnetic tape
First Performance - Installation by Alba D'Urbano "Quem quaeritis..."
Kassel (Germany)
Martinskirche Jun-Sep 1997

#20
Still Blue
(Homage to Derek Jarman)
The Video
(1998)
Video by A.Cipriani, G.Latini and S. Di Domenico
Music by Alessandro Cipriani
First Performance Nuova Consonanza - Rome
SELECTED AT "CORTO CIRCUITO '98" EUROPEAN FESTIVAL OF AUDIO-VISUAL COMMUNICATION - NAPLES

#.22
Reflection of the Moon over two springs
(2000)
for tape

INDEX

B

C

D

E

F

G

H

I

K

L

M

N

O

P

Q

R

S

T

U

V

W

OPCODE LIST

VALUE CONVERTERS

int(x) **(init- or control-rate args only)**
frac(x) **“ “**
dbamp(x) **“ “**
i(x) **(control-rate arg; only)**
abs(x) **(no rate restriction)**
exp(x) **“ “**
log(x) **“ “**
log10(x) **“ “**
sqrt(x) **“ “**
sin(x) **“ “**
cos(x) **“ “**
ampdb(x) **“ “**

sininv(x) **“ “**
sinh(x) **“ “**
cosinv(x) **“ “**
cosh(x) **“ “**
taninv(x) **“ “**
tanh(x) **“ “**
taninv2(x, y) **“ “**

xr **pow iarg, kpow**

PITCH CONVERTERS

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

Tuning opcodes:

icps **cps2pch ipch, iequal**
icps **cpsxpch ipch, iequal, irepeat, ibase**

SIGNAL DISPLAY AND FILE OUTPUT

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

printk itime, kval [, ispace]
printk2 kval [,ispace]
printks "txtstring", itime, kval1, kval2, kval3, kval4

dumpk ksig, ifilname, iformat, iprd
dumpk2 ksig1, ksig2, ifilname, iformat, iprd
dumpk3 ksig1, ksig2, ksig3, ifilname, iformat, iprd
dumpk4 ksig1, ksig2, ksig3, ksig4, ifilname, iformat, iprd

SENSING & CONTROL

ktemp tempest kin, iprd, imindur, imemdur, ihp,ithresh, ihtim, ixfdbak, istartempo, ifn [,idisprd, itweek]
kx, ky xyin iprd, ixmin, ixmax, iymin, iymax [,ixinit, iyinit]
tempo ktempo, istartempo

TIME READING

kr timek
kr times
ir itimek
ir itimes
kr instimek
kr instimes

PROGRAM CONTROL

Jumps: igoto label

tigoto label
kgoto label
goto label

if ia R ib igoto label

if ka R kb kgoto label
if ia R ib goto label

timout istrt, idur, label

Reinitialisation:

reinit label
rigoto label
rireturn

MIDI SUPPORT

MIDI converters

ival notnum
ival veloc
icps cpsmidi
icps cpsmidib
kcps cpsmidib [irange]
kval cpstmid ifn
ioct octmidi
ioct octmidib
koct octmidib [irange]
ipch pchmidi
ipch pchmidib
kpch pchmidib [irange]
iamp ampmidi iscal[, ifn]
kaft aftouch iscal

kchpr chpress iscal
kbend pchbend iscal
ival midictrl inum [, initial]
kval midictrl inum [, initial]

MIDI controller input:

initc7/14/21 ichan, ictlno, ivalue

idest imidic7/14/21 ictlno, imin, imax [, ifn]
kdest midic7/14/21 ictlno, kmin, kmax [, ifn]

idest ictrl7/14/21 ichan, ictlno, imin, imax [,ifn]
kdest ctrl7/14/21 ichan, ictlno, kmin, kmax [,ifn]

nval chanctrl ichan, ictlno [,ilow,ihigh]
kbend pchbend [ilow, ihigh]

Misc MIDI input messages:

**kstatus, kchan, kdata1, kdata2 **

midiin

MIDI note output:

ion ichn, inum, ivel
ioff ichn, inum, ivel
iondur ichn, inum, ivel, idur
iondur2 ichn, inum, ivel, idur

moscil kchn, knum, kvel, kdur, kpause
midion kchn, knum, kvel
midion2 kchn, knum, kvel, ktrig

Output MIDI channel messages:

ioutc(14) ichn, inum, ivalue, imin, imax
koutc(14) kchn, knum, kvalue, kmin, kmax

ioutpb ichn, ivalue, imin, imax
koutpb kchn, kvalue, kmin, kmax
ioutat ichn, ivalue, imin, imax
koutat kchn, kvalue, kmin, kmax
ioutpc ichn, iprog, imin, imax
koutpc kchn, kprog, kmin, kmax

ioutpat ichn, inotenum, ivalue, imin, imax

koutpat kchn, knotenum, kvalue, kmin, kmax

Output MIDI system realtime messages:

mclock ifreq
mrtmsg imsgtype

Misc MIDI output messages:

nrpn kchan, kparmnum, kparmvalu
midiout kstatus, kchan, kdata1, kdata2

Extend MIDI events:

xtratim iextradur
kflag release

INSTRUMENT CONTROL

turnon insno[,itime]
ihold
turnoff

schedule inst, iwhen, idur, ...
schedwhen ktrigger, kinst, kwhen, kdur, ...

FUNCTION TABLE CONTROL

Table manipulation overview

Get information about function tables:

iafno ftgen ifno,itime,isize, igen, iarga[,...iargz]
ftlen(x) (init-rate args only)
ftlptim(x) (init-rate args only)
ftsr(x) (init-rate args only)
nsamp(x) (init-rate args only)

kr tablekt kndx, kfn [, ixmode] [,ixoff] [,iwrap]

ar tableikt andx, kfn [, ixmode] [,ixoff] [,iwrap]
ir itableng ifn
kr tableng kfn

Read/write function tables:

itablew isig, indx, ifn [,ixmode] [,ixoff] [,iwgmode]
tablew ksig, kndx, ifn [,ixmode] [,ixoff] [,iwgmode]
tablew asig, andx, ifn [,ixmode] [,ixoff] [,iwgmode]
tablewk ksig, kndx, kfn [,ixmode] [,ixoff] [,iwgmode]
tablewkt asig, andx, kfn [,ixmode] [,ixoff] [,iwgmode]

ar tablera kfn, kstart, koff
kstart tablewa kfn, asig, koff

itablegpwifn
tablegpw kfn

tablecopy kdft, ksft
itablecopy idft, isft
tablemix kdft, kdoff, klen, ks1ft, ks1off, ks1g, ks2ft, ks2off, ks2g
itablemix idft, idoff, ilen, is1ft, is1off, is1g, is2ft, is2off, is2g

SIGNAL GENERATORS

Linear signal generators:

nr line ia, idur1, ib
nr expon ia, idur1, ib
nr linseg ia, idur1, ib [,idur2, ic[Ö]]
nr expsegia, idur1, ib [,idur2, ic[Ö]]
nr linsegr ia, idur1, ib [,idur2, ic[Ö]], irel, iz
nr expsegr Ia, Idur1, Ib [,Idur2, Ic[...]], Irel, iz

nr adsr iatt, idec, islev, irel[, idelay]
nr madsr iatt, idec, islev, irel[, idelay]

Phase generator:

nr phasor kcps [,iphs]

Table access:

xr table indx, ifn [,ixmode] [,ixoff] [,iwrap]
xr tablei indx, ifn [,ixmode] [,ixoff] [,iwrap]
kr oscil1 idel, kamp, idur, ifn
kr oscil1i idel, kamp, idur, ifn
kr osciln kamp, idur, ifrq, ifn, itimes

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

kr lfo kamp, kcps[, itype]

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

Buzzers:

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

Granular synthesis:

ar fof xamp, xfund, xform, koct, kband, kris, kdur, kdec, iolaps, ifna, ifnb, itotdur [,iphs] [,ifmode]
ar fof2 xamp, xfund, xform, koct, kband, kris, kdur, kdec, iolaps, ifna, ifnb, itotdur, kphs, kgliss
ar fog xamp, xdens, xtrans, xspd, koct, kband, kris, kdur, kdec, iolaps, ifna, ifnb, itotdur[, iphs][, itmode]
ar grain xamp, xpitch, xdens, kampoff, kpitchoff, kgdur, igfn, iwfn, imgdur
ar granule xamp, ivoice, iratio, imode, ithd, ifn, ipshift, igskip, igskip_os, ilength, kgap, igap_os, kgsize, igsize_os, iatt, idec [,iseed] [,ipitch1] [,ipitch2] [,ipitch3] [,ipitch4] [,ifnenv]

Time-streching:

ar [,acmp] sndwarp xamp, xtimewarp, xresample, ifn1, ibeg, iwsize, irandw, ioverlap, ifn2, [itimemode]

**ar1, ar2 [, acmp1, acmp2] **

sndwarpst xamp, xtimewarp, xresample, ifn1, ibeg, iwsize, irandw, ioverlap, ifn2, itimemode

Waveguide physical modeling:

ar wguide1 asig, kfreq, kcutoff, kfeedback;
ar wguide2 asig, kfreq1, kfreq2, kcutoff1, kcutoff2, kfeedback1, kfeedback2

ar pluck kamp, kcps, icps, ifn, imeth [,iparm1, iparm2]
ar wgpluck
ar wgpluck2 iplk, xamp, icps, kpick, krefl
ar repluck iplk, xamp, icps, kpick, krefl, axcite

ar wgbow kamp, kfreq, kpres, kratio, kvibf, kvamp, ifn[, iminfreq]
ar wgbrass kamp, kfreq, kliptens, idetk, kvibf, kvamp, ifn[, iminfreq]
ar wgclar kamp, kfreq, kstiff, iatt, idetk, kngain, kvibf, kvamp, ifn[, iminfreq]
ar wgflute kamp, kfreq, kjet, iatt, idetk, kngain, kvibf, kvamp, ifn[, iminfreq]

More physical models:

ar agogobel kamp, kfreq, ihrd, ipos, imp, kvibf, kvamp, ivibfn
ar marimba kamp, kfreq, ihrd, ipos, imp, kvibf, kvamp, ivibfn, idec
ar vibes kamp, kfreq, ihrd, ipos, imp, kvibf, kvamp, ivibfn, idec
ar shaker kamp, kfreq, kbeans, kdamp, knum, ktimes[, idecay]

4-operator FM instruments:

ar fmtbell kamp, kfreq, kc1, kc2, kvdepth, kvrate, ifn1, ifn2, ifn3, ifn4, ivfn
ar fmrhode kamp, kfreq, kc1, kc2, kvdepth, kvrate, ifn1, ifn2, ifn3, ifn4, ivfn
ar fmwurlie kamp, kfreq, kc1, kc2, kvdepth, kvrate, ifn1, ifn2, ifn3, ifn4, ivfn
ar fmmetal kamp, kfreq, kc1, kc2, kvdepth, kvrate, ifn1, ifn2, ifn3, ifn4, ivfn
ar fmb3 kamp, kfreq, kc1, kc2, kvdepth, kvrate, ifn1, ifn2, ifn3, ifn4, ivfn
ar fmpercfl kamp, kfreq, kc1, kc2, kvdepth, kvrate, ifn1, ifn2, ifn3, ifn4, ivfn

ar fmvoice kamp, kfreq, kvowel, ktilt, kvibamt, kvibrate, ifn1, ifn2, ifn3, ifn4, ivibfn

Emulation instruments:

ar moog kamp, kfreq, kfiltq, kfiltrate, kvibf, kvamp, iafn, iwfn, ivfn
ar mandol kamp, kfreq, kpluck, kdetune, kgain, ksize, ifn[, iminfreq]

ar voice kamp, kfreq, kphoneme, kform, kvibf, kvamp, ifn, ivfn

Random generators (uniform distribution):

nr rand xamp [,iseed, iuse31]
nr randh kamp, kcps [,iseed, iuse31]
nr randi kamp, kcps [,iseed, iuse31]

rnd(x)
birnd(x)

Random generators (various distributions):

xr linrand krange
xr trirand krange
xr exprand krange
xr bexprnd krange
xr cauchy kalpha
xr pcauchy kalpha
xr poisson klambda
xr gauss krange
xr weibull ksigma, ktau
xr betarand krange, kalpha, kbeta
xr unirand krange

SIGNAL MODIFIERS

Gain units:

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

ar dam ain, kthresh, icomp1, icomp2, irtme, iftme

Signal limiters:

ir ilimit isig, ilow, ihigh
nr limit nsig, klow, khigh

xr wrap xsig, xlow, xhigh
xr mirror xsig, xlow, xhigh

Notification:

kout trigger ksig, kthreshold, kmode

Envelope lines:

nr linen kamp, irise, idur, idec
nr linenr kamp, irise, idec, iatdec
nr envlpxkamp, irise, idur, idec, ifn, iatss, iatdec [,ixmod]

kr follow asig, idt

nr adsr iatt, idec, islev, irel[, idelay]
nr madsr iatt, idec, islev, irel[, idelay]

Interpolator:

ar ntrpol asig1, asig2, kpoint [, imin, imax]

Standard filters:

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

ar tonex asig, khp[, inumlayer, istor]
ar atonex asig, khp[, inumalayer, istor]
ar resonx asig, kcf, kbw[, iscl, inumlayer, istor]

ar butterhp asig, kfreq
ar butterlp asig, kfreq
ar butterbp asig, kfreq, kband
ar butterbr asig, kfreq, kband

ar filter2 asig, iM,iN,ib0,ib1,..., ibM,ia1,ia2,...,iaN
kr kfilter2 ksig, iM,iN,ib0,ib1,...,ibM,ia1,ia2,...,iaN
ar zfilter2 asig, kdamp,kfreq,iM,iN,ib0,ib1,...,ibM,ia1,ia2,...,iaN

Specialised filters:

ar nlfilt ain, ka, kb, kd, kL, kC

ar dcblock asig [, igain]

ar lowres asig, kcutoff, kresonance [,istor]
ar lowresx asig, kcutoff, kresonance [, inumlayer, istor]
ar vlowres asig, kcutoff, kresonance, iord, ksep;

ar biquad asig, kb0, kb1, kb2, ka0, ka1, ka2
ar moogvcf asig, kfco, kres
ar rezzy asig, kfco, kres

Sample level operators:

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

Delays:

ar delayr idlt [,istor]
ar delayw asig
ar delay asig, idlt [,istor]
ar delay1 asig [,istor]
ar deltap kdlt
ar deltapi xdlt

ar vdelay asig, adel, imaxdel
ar multitap asig, itime1, igain1, itime2, igain2 . . .

Reverbs:

ar comb asig, krvt, ilpt [,istor]
ar alpass asig, krvt, ilpt [,istor]
ar reverb asig, krvt [,istor]
ar nreverb asig, ktime, khdif
ar reverb2 asig, ktime, khdif

Special effects:

ar harmon asig,kestfrq,kmaxvar, kgenfrq1, kgenfrq2, imode, iminfrq, iprd
ar flanger asig, adel, kfeedback, imaxd

FFT-based morphing synthesis:

asig cross2 ain1, ain2, ilen, iovl, iwin, kbias

OPERATIONS USING SOUNDFILE ANALYSIS DATA

Based on Heterodyne analysis:

HETRO.EXE - Fourier analysis for adsyn generator

ar adsyn kamod, kfmod, ksmod, ifilcod

Based on Phase vocoder analysis:

PVANAL.EXE - Fourier analysis for phase vocoder generators

ar pvoc ktimpnt, kfmod, ifilcod [,ispecwp]
ar pvadd ktimpnt, kfmod, ifile, ifn, ibins [, ibinoffset, ibinincr]
kfrq,kamp pvread ktimpnt, ifile, ibin
pvbufread ktimpnt, ifile
ar pvinterp ktimpnt, kfmod, ifile, kfreqscale1, kfreqscale2, kampscale1, kampscale2, kfreqinterp, kampinterp
ar pvcross ktimpnt, kfmod, ifile, kamp1, kamp2, [ispecwp]

tableseg ifn1, idur1, ifn2[, idur2, ifn3[...]]
tablexseg ifn1, idur1, ifn2[, idur2, ifn3[...]]
ar vpvoc ktimpnt, kfmod, ifile, [ispecwp]

Based on Linear predictive coding (LPC):

LPANAL.EXE - Linear predictive analysis for lpread/lpreson generators

**krmsr, krmso, kerr, kcps **

lpread ktimpnt, ifilcod[, inpoles][,ifrmrate]
ar lpreson asig
ar lpfreson asig, kfrqratio

lpslot islot
lpinterpolislot1,islot2,kmix

Using single Fourier analysis frame:

CVANAL.EXE - Impulse response fourier analysis for convolve operator

ar1[,...[,ar4]]] convolve ain, ifilcod, ichan

OPERATIONS USING SPECTRAL DATA TYPES

Overview

wsig spectrum xsig, iprd, iocts, ifrqs, iq[,ihann, idbout, idsprd, idsinrs]
wsig specaddm wsig1, wsig2[, imul2]
wsig specdiff wsigin
wsig specscal wsigin, ifscale, ifthresh
wsig spechist wsigin
wsig specfilt wsigin, ifhtim

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

SOUND INPUT & OUTPUT

Sound input:

a1 in
a1, a2 ins
a1,...,a4 inq
a1 soundin ifilcod [,iskptim] [,iformat]
a1, a2 soundin ifilcod [,iskptim] [,iformat]
a1,...,a4 soundin ifilcod [,iskptim] [,iformat]
a1[,a2[,a3,a4]] diskin ifilcod, kpitch [,iskiptim][, iwraparound] [,iformat]

Sound output:

outasig
outs asig1, asig2
outs1 asig
outs2 asig
outq asig1, asig2, asig3, asig4
outq1 asig
outq2 asig
outq3 asig
outq4 asig

Panning and 3-D sound:

a1,...,a4 pan asig, kx, ky, ifn [,imode] [,offset]
aL, aR hrtfer asig, kAz, kElev, "HRTFcompact"
a1,..., a4 space asig, ifn, ktime, kreverbsend [,kx, ky]
a1,..., a4 spsend
kr spdist ifn, ktime, [,kx, ky]
a1,... , a4 locsig asig, kdegree, kdistance, kreverbsend
a1,..., a4 locsend