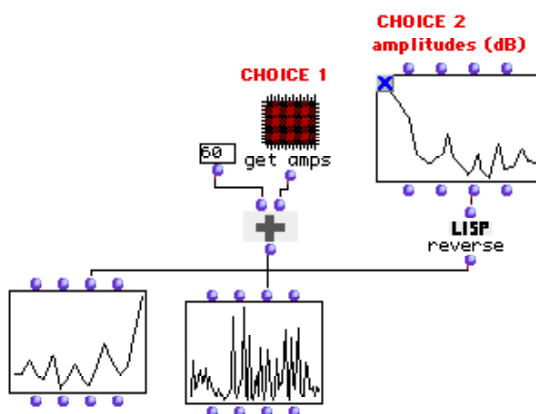


OpenMusic omChroma

Paradigms for the high-level musical control
of sonic processes using omChroma
+
omChroma 1.0 Manual & Tutorial



First Edition, September 2000

© 2000, Ircam. All rights reserved.

This manual may not be copied, in whole or in part, without written consent of Ircam.

This documentation was written by Marco Stroppa, and was produced under the editorial responsibility of Marc Battier, Marketing Office, Ircam.

OpenMusic was conceived and programmed by Gérard Assayag and Carlos Agon.

The omChroma library was originally conceived and programmed by Marco Stroppa.

First edition of the manual, September 2000.

This documentation corresponds to version 1.0 of the omChroma library, and to version 2.0 or higher of OpenMusic.

Apple Macintosh is a trademark of Apple Computer, Inc.

OpenMusic is a trademark of Ircam.

Ircam
1, place Igor-Stravinsky
F-75004 Paris
Tel. 01 44 78 49 62
Fax 01 44 78 15 40
E-mail ircam-doc@ircam.fr

Groupe d'utilisateurs Ircam

L'utilisation de ce programme et de sa documentation est strictement réservée aux membres des groupes d'utilisateurs de logiciels Ircam. Pour tout renseignement supplémentaire, contactez :

Département de la Valorisation
Ircam
1, Place Stravinsky
F-75004 Paris
France

Courrier électronique: bousac@ircam.fr

Veuillez faire parvenir tout commentaire ou suggestion à :

M. Battier
Département de la Valorisation
Ircam
1, Place Stravinsky, F-75004 Paris, France

Courrier électronique : bam@ircam.fr

<http://www.ircam.fr/forumnet>



To see the table of contents of this manual, click on the Bookmark Button located in the Viewing section of the Adobe Acrobat Reader toolbar.

To move between pages, use Acrobat Reader navigations buttons or your keyboard's arrow keys

Content

1 FOREWORD.....	7
1-1) Basics	7
1-2) How to use this text	7

2 THE ISSUE OF THE CONTROL OF DSP UNITS	9
2-1) A simple case	9
2-1-1) Wave-table synthesis	9
2-1-2) Synthesis bank in the orchestra	10
2-1-3) Synthesis bank in the score	11
2-2) Sonic potential	13
2-3) Change of representation.....	14
2-4) Virtual synthesizer	16
2-5) Epistemological significance	16

3 PREREQUISITES	17
------------------------------	-----------

4 INITIALIZATION PHASE	18
4-1) Structure of the directories	18
4-2) Preferences	19
4-3) Important remarks concerning Csound	20

5 SOME SIMPLE EXAMPLES	21
5-0) Before doing anything	21
5-1) Step-by-step example (TUT 01a)	21
5-2) Generalizations of the previous example (TUT 01b-e)	31
5-3) Chant's FOF's (TUT 02).....	35
5-4) Chant's filters (TUT 03).....	37

6 STEP-BY-STEP TUTORIAL	38
6-1) Simple Matrix (main control object) (TUT 04a-b)	38
6-2) Musical controls of a single matrix (TUT 05)	42
6-3) Control slots and parsing fun (elementary usage): single event (TUT 06)	46
6-4) Control slots and parsing fun: many matrices (TUT 07)	51
6-5) Instanciation of a new class (TUT 08).....	54
6-5) Instanciation of a new class (TUT 08)	57
6-6) Tables: all possibilities	60
6-6-1) Local tables (TUT 09a).....	61
6-6-2) Global tables (TUT 09b).....	63
6-7) Percussive FOF in Chant and Csound together (TUT 10)	64
6-7) Control of Chant: all the available patches (TUT 11)	65
6-8) High-level control of the amplitude in Chant (TUT 12).....	66
6-9) Two control layers embedded (TUT 13)	69

7 SPECIAL REMARKS / BUGS.....	71
--------------------------------------	-----------

7-1) Class slots: "source-code" and "globals-list"	71
7-2) Modification of a class	72
7-3) Reserved control keywords: :precision, :parsing-fun	72
7-4) Csound abort and bug	72
7-5) P-fields	73

8 ALPHABETICAL LIST OF THE AVAILABLE OBJECTS AND METHODS 74

CHANT-PATCHES	74
CH-FOB-EVT	74
CH-FILT-EVT	74
CH-NOISE-EVT	75
COMP-FIELD	75
COMP-LIST	75
FILL-COMP	75
GEN05	75
GEN07	76
GEN-CS-TABLE	76
GET-COMP	76
GET-INSTRUMENT	76
NEW-COMP	76
SYNTHESIZE	77

9 PERSPECTIVES..... 78

10 APPENDIX 1 79

11 APPENDIX 2 83

How to make your own version of csound compatible with omChroma	83
---	----

1 FOREWORD

This documentation refers to the version 1.0 of the library *omChroma*.

OmChroma is a generalization of the kernel of *Chroma*, a system for the control of software synthesis I have been developing since 1980 first at the Centro di Sonologia Computazionale (CSC) of the University of Padua, then at IRCAM. *Chroma* was used for all my pieces dealing with electronics (see the Appendix 1 for a concise survey of this system and the collaborators who participated in the project).

Further features are still being regularly developed for my own productions.

By separating the control from the digital-signal-processing (DSP) layer, *omChroma* also aims at providing a common environment which might be used when documenting or analyzing works dealing with composed sonic processes. The huge amount of data, the heterogeneity of the technological tools as well as their becoming rapidly obsolete has made this task particularly excruciating for the time being.

The choice of developing *Chroma* into a library of *OpenMusic* will also provide the user with an integrated environment where processes of computer-aided composition can be easily extended to encompass the layer of micro-structural composition needed by the control of software synthesis¹.

1-1) Basics

OmChroma is a special-purpose library dedicated to the high-level musical control of sonic processes. It implements the concept of a "virtual synthesizer" (see chapter 2) and currently calls the following real synthesizers: **Csound** (the most popular and widely used free synthesis software, available for different platforms²) and the Chant plugins of **Diphone**, developed at IRCAM by the Analysis and Synthesis Group and written by Adrien Lefèvre.

All the patches of the tutorial were written by Carlos Augusto Agon and myself.

1-2) How to use this text

Chapter 2 is a concise analysis of the issue of high-level control of DSP units. If you are already familiar with it, you can skip it and go directly to the chapters 4 to 6.

Chapter 3 presents the prerequisites needed to use *omChroma* efficiently.

-
1. Throughout this text software or DSP synthesis refers to the process of generation of sound that is part of a compositional task and is therefore controlled by a score or even some kind of gesture. Software synthesis includes all sorts of sound processing devices (oscillators, distortion units, samplers, filters, etc.). I will not further discuss the pertinence of this expression. The reader can refer to the abundant literature in this domain.
 2. See www.csounds.com for further information concerning Csound.

Chapters 4 to 6 are the bulk of the documentation. Since the environment is very rich, a good understanding of its features is highly recommended, in order to be able to use all its potentialities. Because of their pedagogical structure, it is best to read the chapters 4 to 6 and try out the tutorials from the beginning to the end.

Chapter 7 contains further, more detailed information about *omChroma* and *OpenMusic*.

Chapter 8 concisely summarizes all the available functions.

Chapter 9 discusses the foreseeable improvements that will be implemented into future versions of the library.

2 THE ISSUE OF THE CONTROL OF DSP UNITS

2-1) A simple case

To state it as plainly as possible, controlling DSP units means devising appropriate abstractions to deal with huge amounts of data sent to banks of sound-generating patches.

Since this statement may "sound" a little obscure for the time being, and before discussing more theoretical issues, I will start with a very simple example: let's suppose that we want to generate a 10-second long harmonic sound containing 10 partials with different amplitudes. If nothing else is specified, there are several ways to write a patch that will produce it.

Taking Csound as a synthesizer, three different implementations are discussed below. When the orchestra and score files are run in Csound, they will give strictly identical sounds¹, although they may differ in computational efficiency.

2-1-1) Wave-table synthesis

The orchestra and score files are **ex1.orc** and **ex1.sco** in "**OM 3.7 / chroma / cs**".

The control paradigm consists in using one single oscillator, whose audio table is made of 10 harmonics with different relative amplitudes. Csound provides easy ways to generate such a table with the GEN number 10.

SCORE:

; Wave table with 10 partials of different amplitude

f1 0 32769 10 1 0.3 0.6 0.2 0.45 0.05 0.1 0.01 0.2 0.22

; Simple amplitude envelope

f11 0 4097 7 0 500 1 1000 0.5 1000 0.5 1597 0

; Notes

i1 0 10 0.5 110.0 11

1. In order to eliminate possible differences in the maximum amplitude, the synthesis should be floating-point and the final sounds rescaled to the same maximum amplitude.

This is surely the most efficient patch, but also probably the most constraining one. For example, all the harmonics must have the same duration and amplitude envelope. If a vibrato needs to be applied¹, they will all vibrate at the same rate and with the same interval.

However, sounds with a different number of harmonics (within the aforementioned limitations) are simple to generate by changing the parameters of the table. If an additional P-field allowed to select the audio table directly from the score, sounds with different spectra might also be superposed. Non-harmonic spectra are even feasible, if the GEN n. 9 or 19 is used².

2-1-2) Synthesis bank in the orchestra

The orchestra and score files are **ex2.orc** and **ex2.sco** in "**OM 3.7 / chroma / cs**".

Here the strategy is different: instead of hard-wiring the components inside a GEN table and using only one oscillator per sound, the main synthesis module is a simple sine-tone generator. The orchestra is a patch of 10 sine-tone oscillators and the control of their input parameters is operated from the score. Such a patch is called a "bank" of oscillators, where "oscillator" is one of many possible elementary DSP modules able to generate sound.

SCORE:

; Sinusoidal wave table

f1 0 32769 10 1

Simple amplitude envelope

f11 0 4097 7 0 500 1 1000 0.5 1000 0.5 1597 0

; Notes

i1 0 10 1 110 0.3 220 0.6 330 0.2 440 0.45 550 0.05 660 0.1 770 0.01 880 0.2 990 0.22 1100 11

The computation is less efficient, since Csound will have to compute ten sine tones within the orchestra. The score is perhaps easier to read, because the frequency and amplitude values of each component are near each other and directly visible as P-fields. The main drawback is the maximum number of synthesis modules, which is fixed once for all³ in the instrument and the amount of control data. If, say, 100 and not 10 partials were to be computed, the patch and the score will become very cumbersome to read.

1. Via a simple modification of the orchestra.

2. This analysis is Csound-specific. Other synthesizers may provide or not similar features.

3. One can always control fewer modules, by writing, for instance, a 0 amplitude when a module is not needed, but this solution would not be very efficient.

On the other hand, this patch can be easily modified to include, for instance, separate amplitude envelopes or vibrato modules for each partial or groups of partials, although all the partials will still have the same duration.

This implementation is a reasonable compromise between efficiency and flexibility and is often the only one possible when using real-time synthesizers. Different versions of this control paradigm were used by Stephen Mc Adams and his collaborators when testing the importance of common patterns of vibrato as a means of fusing or separating simultaneous sound sources¹.

2-1-3) Synthesis bank in the score

The orchestra and score files are **ex3.orc** and **ex3.sco** in "**OM 3.7 / chroma / cs**".

This implementation still deals with a synthesis bank, but the orchestra contains only one single sine-tone oscillator. The bank is entirely controlled from the score. Since each partial corresponds to one score line, there will be as many "i statements" as there are partials.

SCORE:

; Wave table with 10 partials of different amplitude

f1 0 32769 10 1

; Simple amplitude envelope

f11 0 4097 7 0 500 1 1000 0.5 1000 0.5 1597 0

; Notes

i1 0 10 1 110.0 11

i1 0 10 0.3 220.0 11

i1 0 10 0.6 330.0 11

i1 0 10 0.2 440.0 11

i1 0 10 0.45 550.0 11

i1 0 10 0.05 660.0 11

i1 0 10 0.1 770.0 11

i1 0 10 0.01 880.0 11

i1 0 10 0.2 990.0 11

i1 0 10 0.22 1100.0 11

1. I studied these Music-10 patches when I arrived at IRCAM in 1982. The sound examples are published in the IRCAM Research reports.

This is both the least efficient and the most flexible solution: provided that the synthesizer allows for automatic time sorting of the score lines and for dynamic allocation of new instances of the instrument¹, each partial can have its own amplitude envelope, duration and action time. Each score line in fact allocates a copy of the control data of the whole instrument and will use a little more memory and computing time than the other implementations.

1. This is a great and powerful feature of most software synthesizers. Real-time synthesizers, on the other hand, provide little or no means for dynamically allocating new instruments while the patch is running, because of the computational constraints.

2-2) Sonic potential

If the purpose of our discussion were just to synthesize the banal harmonic sound mentioned at the beginning, all the implementations would be equivalent in terms of sonic results.

However, software synthesis usually does not deal with the generation of a unique sound, but with processes of sonic development. From an idea about a given sound or process, the composer generates a multitude of sound processes which both implement and develop the original idea.

In this framework, therefore, a sound-generating patch cannot be considered in terms of a single sound, but in terms of "sonic potential", that is the classes of sonic material bearing certain relationships with the composer's original idea that a patch is able to generate. Since by definition all the possible sounds that a given patch can generate constitute its "potential", this is usually an infinite quantity, although most of the time only a limited amount will satisfy the composer's requirements.

It is only from this perspective that the three implementations above are no longer sonically equivalent. In the very particular case we discussed they will produce the same acoustical sound, but they represent it in three different ways, and therefore belong to three different sonic potentials. If our harmonic sound is to be developed, the three patches cannot be modified in the same way, because not controlled by the same data structures, as is clear when comparing the scores.

This simple example shows that any given solution to a sound-synthesis problem is much more than a simple DSP patch: incorporating a certain control paradigm, it generates a given sonic potential, whose characteristics appear more clearly when trying to generate several sounds.

Since each potential makes some processes of sonic development more or less explicit, it already embeds a particular representation of them. In other terms it is an idea of how sound should be structured, controlled, developed and even "enjoyed"¹. This is at the same time a compositional task, an aesthetical issue and an epistemological question.

1. That is of what a "good" sound is.

2-3) Change of representation

OmChroma is optimized for the control of sonic processes whose implementation corresponds to the third case discussed above, that is where the orchestra contains a single sound-generating patch and the control of each independent instance in the DSP bank is performed within the score.

In this context, as I already observed, the score file will contain as many "i-statements" as there are elements in the bank. From the point of view of the representation, such a score looks like a matrix, whose columns correspond to the P-fields and whose rows to each new instance of an instrument:

Instru- ment num- ber (P1)	Action Time (P2)	Dura- tion (P3)	Ampli- tude (P4)	Fre- quency (P5)	Ampli- tude Envlp (P6)
i1	0	10	1	110	11
i1	0	10	0.3	220	11
i1	0	10	0.6	330	11
i1	0	10	0.2	440	11
i1	0	10	0.45	550	11
i1	0	10	0.05	660	11
i1	0	10	0.1	770	11
i1	0	10	0.01	880	11
i1	0	10	0.2	990	11
i1	0	10	0.22	1100	11

In this matrix, some data will probably vary at every row (for example the frequency), other might, but do not necessarily vary at every row (e.g. the duration), and some data will always be the same, by definition, such as the instrument number, since each matrix is as a collection of instances of the "same" orchestra¹.

The most significant conceptual change put forward by *omChroma* concerns the way this matrix is represented. Instead of having a variable number of rows (one row per instance) with a fixed number of columns (P-fields), *omChroma* deals with matrices with a fixed number of rows (P-fields) and a variable number of columns (instances).

In other words, *omChroma* "turns" the rows into columns and vice versa:

1. It is, of course, possible to generate several matrices, each one dealing with a different instrument, but within one matrix the instrument is the same.

In st nc	1	2	3	4	5	6	7	8	9	1 0
----------------	---	---	---	---	---	---	---	---	---	--------

A T	0 (for all the instances)									
--------	---------------------------	--	--	--	--	--	--	--	--	--

D ur	10 (for all the instances)									
---------	----------------------------	--	--	--	--	--	--	--	--	--

A m p	1	0. 3	0. 6	0. 2	0. 4 5	0. 0 5	0. 1	0. 0 1	0. 2	0. 2 2
-------------	---	---------	---------	---------	--------------	--------------	---------	--------------	---------	--------------

Fr e q	"compute the first 10 partials of a harmonic series with f0 = 110"									
--------------	--	--	--	--	--	--	--	--	--	--

A- e n v	11 (for all the instances)									
-------------------	----------------------------	--	--	--	--	--	--	--	--	--

When the matrix is read vertically, column by column from 1 to 10, each score line will be reconstructed. This is the basic control model used in *omChroma* and will be called an "event"¹. An "event" therefore corresponds to an arbitrarily large bank of a single DSP patch in the orchestra instantiated within the score. Each row specifies the variation of a P-field across the whole bank.

This is however a much more mighty abstraction than a simple change of representation: it is useless to repeat values that do not vary during the matrix; it is also possible, as with the frequencies above, to compute the needed values algorithmically by asking a function to fill the row. Finally, the instrument number need not be specified, since it will always be the same and associated to the whole matrix.

1. In this text both the word "matrix" and "event" will be both used. "Event" will imply an underlying compositional model (the conceptual model), while "matrix" will have a more technical meaning (the control structure) in the context of *omChroma*. In the framework of this text the difference is however not so important.

As we shall see below, *omChroma* implements this model of a control matrix in a very powerful, efficient and ergonomical way.

2-4) Virtual synthesizer

Every sound-generating patch will require specific control data in the order specified in the patch. This order depends on the nature of both the patch and the sound-generating modules. However, there are many controls that are the same independently of the patch, for example, amplitude envelopes, frequencies, maximum amplitudes, vibrato frequencies and depths, and so on and so forth. No matter whether the sound-generating module is a bank of sine tones, of filters or of frequency-modulation units in Csound, Max/msp or Chant or even a bank of resonating plates in Modalys from a conceptual standpoint a vibrato is a vibrato, although it might have to be implemented in a different fashion at the level of the chosen synthesizer!

The abstraction procedures contained in *omChroma* provide very efficient tools for coping with these issues: similar controls will no longer necessitate a change of syntax or structure, depending on the synthesizer being used; *omChroma* will automatically take care of it. The user is then free to concentrate on more interesting musical issues.

This analysis of sound control can be develop even further: *omChroma* is in fact a "virtual synthesizer", that is a "language to represent the specific parameters of sonic processes independently of any given real synthesizer, synthesis engine and computer platform". Some examples of the tutorial will show simple implementations of these important control abstractions.

When new real synthesizers become available or are added to the environment, there will be no need to change the general control structures. Only the minor low-level layer providing the interface with the synthesizer will have to be updated.

2-5) Epistemological significance

Let me finally briefly insist on the fact that an "event" in *omChroma* embeds a certain way of thinking about sound structure and its control. It has therefore an epistemological "flavor" that may not always be suited to certain processes. My personal experience, the experience of those who already worked with *Chroma* in the past and the study of several examples in the literature concerned with the control of DSP modules have however shown that this is a very general and powerful abstraction and that its aesthetical or cognitive biases are not too constraining.

Every user must however be aware of the intrinsic epistemological nature of each paradigm for the control of sound, including those implemented in *omChroma*. The high-level musical control of sonic processes is a very delicate issue, dealing as much with composition as with sound design.

3 PREREQUISITES

Although all the tutorials can be easily run and changed, this documentation presupposes a certain degree of familiarity with Csound, the SDIF protocol and some plugins of Diphone (mainly FOF and Filter banks used in Chant and the Models of Resonance).

It also presumes that the user already has a certain experience in designing DSP patches and their control, including the problems of the right units of measure and of particular control strategies. Although some examples of the tutorials might be taken as paradigms of control, the purpose of this documentation is to show how *omChroma* functions and not to account for all the possible paradigms.

For this reason and for the sake of clarity, the examples will limit themselves to as few Csound and Diphone patches as possible (mainly additive synthesis, FOF and filter banks). The control strategies, however, can be applied to many other patches. I will also not delve into the issue of instrument design, about which a lot of literature is available including several catalogues of instruments.

The current version of *omChroma* is best suited to synthesizers such as Csound or Diphone, with dynamic allocation of instances of instruments and no (reasonable) limits of memory. No mechanisms are provided to allocate and free copies of instruments in fixed-instrument banks, although such mechanisms will not be difficult to implement (see chapter 9).

When using *omChroma*, the DSP patch is always supposed to be grammatically and sonically correct. In other terms, before running it, you should thoroughly verify that the patch is compiled successfully and that it produces the sonic processes being modeled. *OmChroma* is not a GUI¹ for the design of DSP patches and only deals with their control.

To avoid having problems with too small or large amplitudes, all the computation in Csound and Diphone is floating point and then be rescaled to whatever maximum amplitude is specified.

1. Graphical User Interface.

4 INITIALIZATION PHASE

4-1) Structure of the directories

Before starting to use *omChroma* it is worth spending a few minutes to take a look at the structure of the folders in your hard disk and make sure that the preferences are correctly set.

The code of the library is situated in the folder "OM 3.7 / User Library / omChroma 1.0". In the directory "OM 3.7" you will also find a folder called "**chroma**" which contains the results of the computation and data used by the examples of this tutorial.

The required directories in this folder are:

snd: contains the sound files produced by the synthesis process

inter-files: contains various intermediate files

The other two folders are used only by the examples of the tutorial:

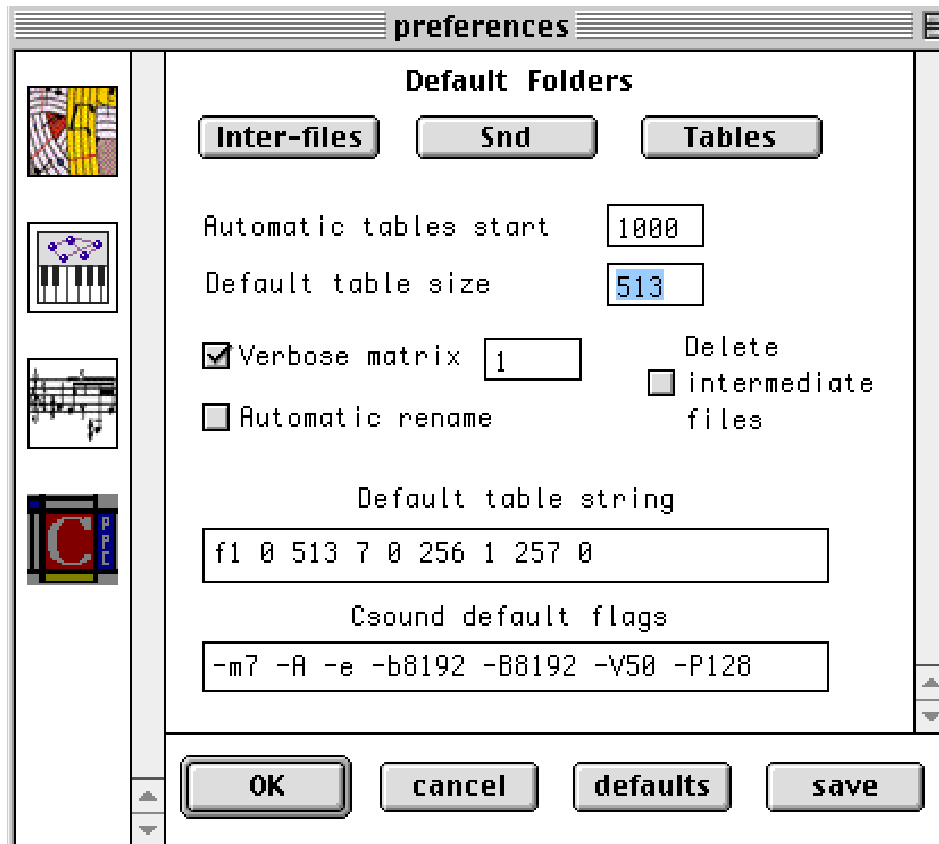
cs: Csound orchestras, tables and test score files

data: mixed data (mainly analysis data)

4-2) Preferences

To set the preferences select the menu "Edit / Preferences" when the Listener is not active and click on the icon of *omChroma*.

Default Settings:



Inter-files: select the directory of used by intermediate files (called by default "my_syntX.orc/sco", where "X" is an integer number).

Snd: select the sound-file directory (by rule sound files have the same name as the intermediate files followed by the extension "aiff").

Tables: select the default directory used to store data bases of Csound GEN instructions.

Automatic tables start (integer number > 0): starting number of Csound GEN instructions when their number is automatically computed.

Default table size (integer number, power of 2 or power of 2 + 1): default size of a Csound table (when none is specified).

Csound default flags: default flags passed to perf's command line when called.

Verbose matrix (integer number): show some data during the computation.

0: show nothing

> 0: show data every "x" matrices

< 0: show data every "x" lines within a matrix (reset at the beginning of each new matrix).

This flag is useful while debugging and to monitor the state of the computation when long processes are run.

Automatic rename: applies only to intermediate files. If checked (ON), the name of each new intermediate file will be followed by an increasing integer number, otherwise only the last files will be kept.

Delete intermediate files: if checked (ON), all the intermediate files will be deleted at the end of the computation.

4-3) Important remarks concerning Csound

For the time being *omChroma* uses a specially adapted version of Csound so that it can be automatically called by *OpenMusic*. This version called "**Csoundom**" and "**PERFom**" is based on Csound 4.05 and found in the folder "**OM 3.7 / User Library / omChroma 1.0 / OMchroma sources / CSOUND / folder**".

To avoid "confusing" the Macintosh operating system about which "perf" to run, no other versions of "perf" ought to be found in the hard disks. Remaining "perf's" should either be trashed (and the trash emptied!), or compressed so that they are invisible to the system¹.

Make also sure that "perf" has enough memory when running long examples or loading large synthesis files. Some tutorials might require more memory for either "perf" or *OpenMusic*.

See the Appendix 2 for further information concerning adapting and compiling Csound so that it can be called by *OpenMusic*.

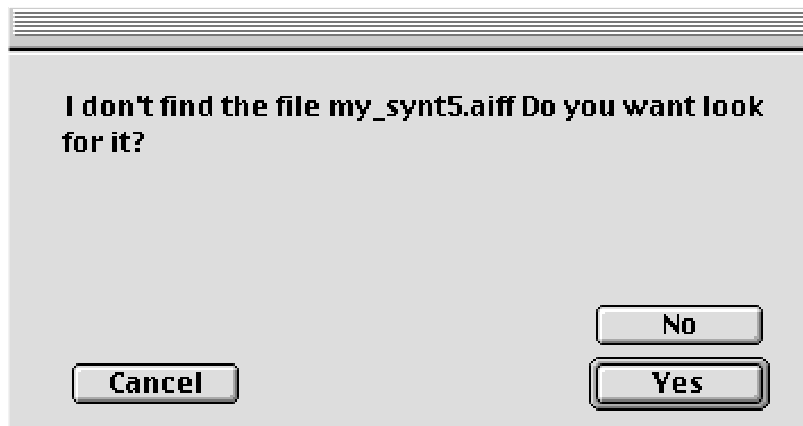
IMPORTANT: If a Csound error appears while "perf" running or if you "kill" the process, *OpenMusic* will not be able to realize that the computation has prematurely stopped, since "perf" replies exclusively to its front end. In this case, you should abort the process in the Listener (in the menu "**Lisp / Abort**" when the Listener's window is the active) before being able to continue to work.

1. We hope that future releases of Csound for the Macintosh will include the small modification that makes them compatible with *OpenMusic*. Please check the mirror sites regularly. Useful links can be found at "<http://seamus.lsu.edu/links/csound-links.html>". If you prefer to use your own version of Csound, you should trash "PERFom". In this case, however, Csound will not be automatically called by *omChroma*.

5 SOME SIMPLE EXAMPLES

5-0) Before doing anything

When loading some patches of the tutorial, *OpenMusic* might warn you that it cannot find a sound file and ask you to find it.



If it is a sound file that will be computed when running the example, you should reply "NO"¹.

Original sound files (such as the ones contained in TUT 05) are in **"OM 3.7 / exemple files / Aiff"**.

All the paths of the tutorials passed as an argument to "tables" in the "synthesize" method are relative to the root of MCL and should load the corresponding files without trouble. However, if you prefer to use absolute paths, they can be typed in using the following syntax: **#P"my-hd:my-folder:om 3.7:chroma:cs:my-funs.lisp"**.

5-1) Step-by-step example (TUT 01a)

We will describe here how to proceed step-by-step to generate the OM patch contained in the file TUT 01a, which reproduces the third example of Risset's catalogue n. 430 (three successive approximations of a bell sound).

Let's suppose that you have already prepared and tested the simple instrument called **"bell.orc"** (found in "chroma / cs"). The score **"bell.sco"** exactly reproduces the score of the third example of Risset's cata-

1. The first time the tutorials are loaded, they contain no sound files at the end of their patch, so this warning should not appear.

logue. Since this example is already prepared in the tutorial, make a copy of "bell.orc" into, for example, "my-bell.orc".

Seen as a matrix (see chapter 2), the score looks like:

Instrument number (P1)	Action Time (P2)	Duration (P3)	Amplitude (P4)	Frequency (P5)	Amplitude Envp (P6)
i1	0	20	0.458	224	11
i1	0	18	0.305	225	11
i1	0	13	0.458	368	11
i1	0	11	0.824	369.7	11
i1	0	6.5	1.221	476	11
i1	0	7	0.763	680	11
i1	0	5	0.671	800	11
i1	0	4	0.061	1096	11
i1	0	3	0.061	1200	11
i1	0	2	0.046	1504	11
i1	0	1.5	0.061	1628	11

We will now generate this score using the model of representation of *omChroma*.

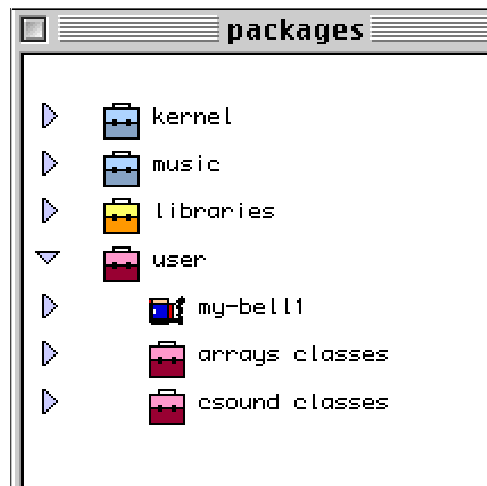
Launch *OpenMusic* and load the Workspace "OMTutorial_WS". Open a new patch, type the function **"get-instrument"** and evaluate it.



Select the orchestra file "my-bell.orc". On the Listener window the following message is displayed:

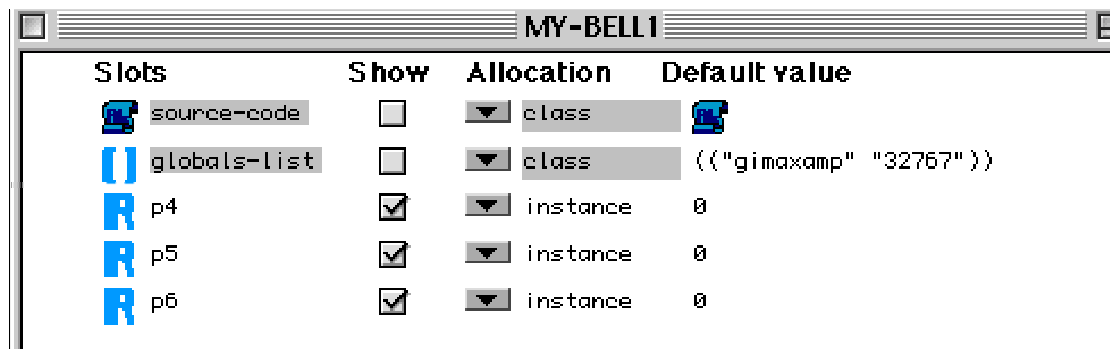
"A class called my-bell1 was defined."

This means that an OpenMusic class corresponding to the instrument was created in:



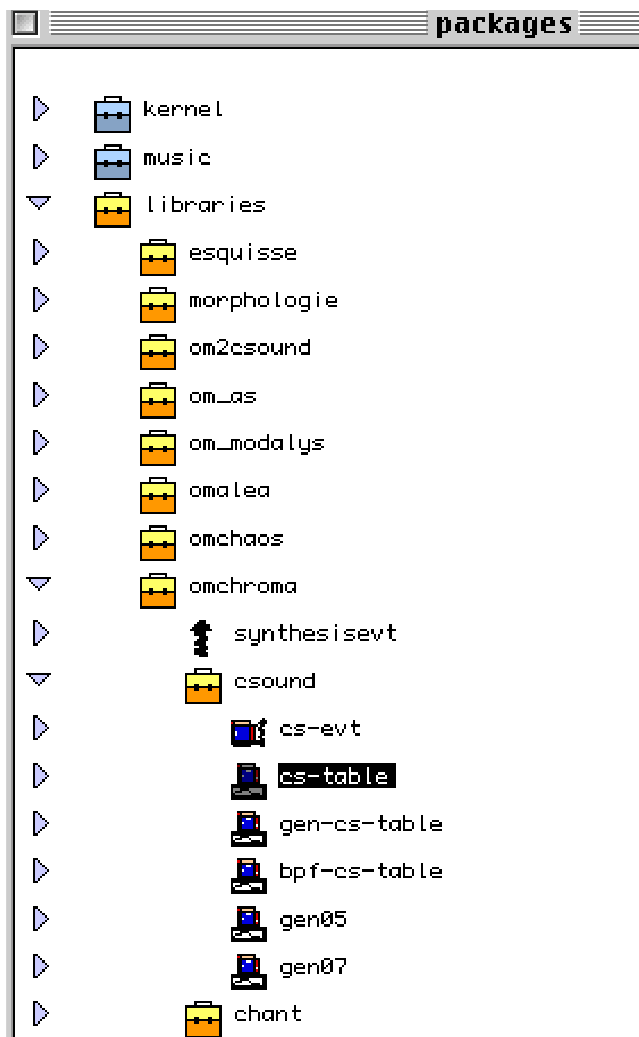
The process of creating a new class is an extremely important feature of *omChroma* and should be done just once at the beginning and only when you are sure that the instrument is correct and suits your needs. *OpenMusic* will make a copy of the code of the instrument within the class. If you wish you can now delete the orchestra file (although it is not very interesting to do so!).

For clarity's sake, it is worth editing the class, so that it is more readable. If you double click on the icon "my-bell1" above the following window will appear:















By clicking on the slots p4, p5 and p6, it is possible to change their default names into something more meaningful. It is these names that will appear as keywords in the control matrix. Assign also a **default value** (which will be used when no input values are available) and a **default type**.

Possible default types are either "real" (that is what you see by default) or a "csound table". For example to change the type of p6 from "real" to "cs-table" drag the type definition onto the type's icon in the MY-BELL1 window.

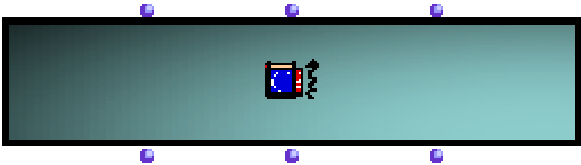


The icon and default values of this slot will change. At the end your window should look like this:

BELL1				
Slots	Show	Allocation	Default value	
 source-code	<input type="checkbox"/>	 class		
 globals-list	<input type="checkbox"/>	 class	(("gimaxamp" "32767"))	
 amp	<input checked="" type="checkbox"/>	 instance	0.5	
 freq	<input checked="" type="checkbox"/>	 instance	440.0	
 amp-fun	<input checked="" type="checkbox"/>	 instance		

This class will remain defined until you purposely delete it¹. Next time you run OM, the class MY-BELL1 will be available (there is no need to load the instrument again).

Close the window and drag the icon of "my-bell1" onto the patch. A matrix-like factory of the class "my-bell1" will be instantiated:



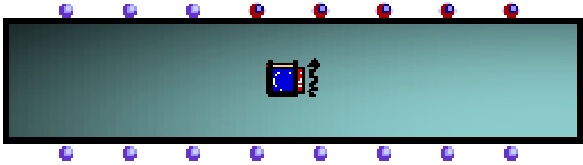
The blue inlets are (from left to right):

self: the object itself (as everywhere in OM)

numrows: the number of rows of the matrix (1)

Action-time: starting time of the whole factory (0.0)

If you select the factory and type "alt -->" (the key "alt" together with the key right arrow), other red inlets will appear until the message "All parameters are already used." is displayed. They correspond to the slots of the class "my-bell1"



1. Classes are found in the folder "user" contained inside the folder "packages" in the Workspace. To delete a class, simply select it and use "Clear" from the Menu "Edit".

The first two inlets from the left are always the same for every class of a Csound orchestra, namely:

:e-dels: entry delays [sec], time interval between the object's "action-time" and the beginning of each line in the score.

:durs: duration [sec], duration of each line in the score.

The others will have the names of the slots of this class preceded by a ":". In our case, they will be called:

:amp: amplitudes of the event [0-1]

:freq: frequencies [Hz]

:amp-fun: amplitude envelopes [gen number or bpf]

Their default value will be the one specified when the class was defined.

To see which keyword correspond to which inlet, simply point the arrow at the inlet and the keyword will appear.

As seen above (chapter 2), the control model of *omChroma* transforms the columns of a matrix into rows and the other way around. Applying this change of representation, the matrix above will become:

Instn	1	2	3	4	5	6	7	8	9	10	11
-------	---	---	---	---	---	---	---	---	---	----	----

e-dels	0										
--------	---	--	--	--	--	--	--	--	--	--	--

Dur	20	18	13	11	6.5	7	5	4	3	2	1
-----	----	----	----	----	-----	---	---	---	---	---	---

Amp	.458	.305	.458	.824	1.22 1	.763	.671	.061	.061	.046	.061
-----	------	------	------	------	-----------	------	------	------	------	------	------

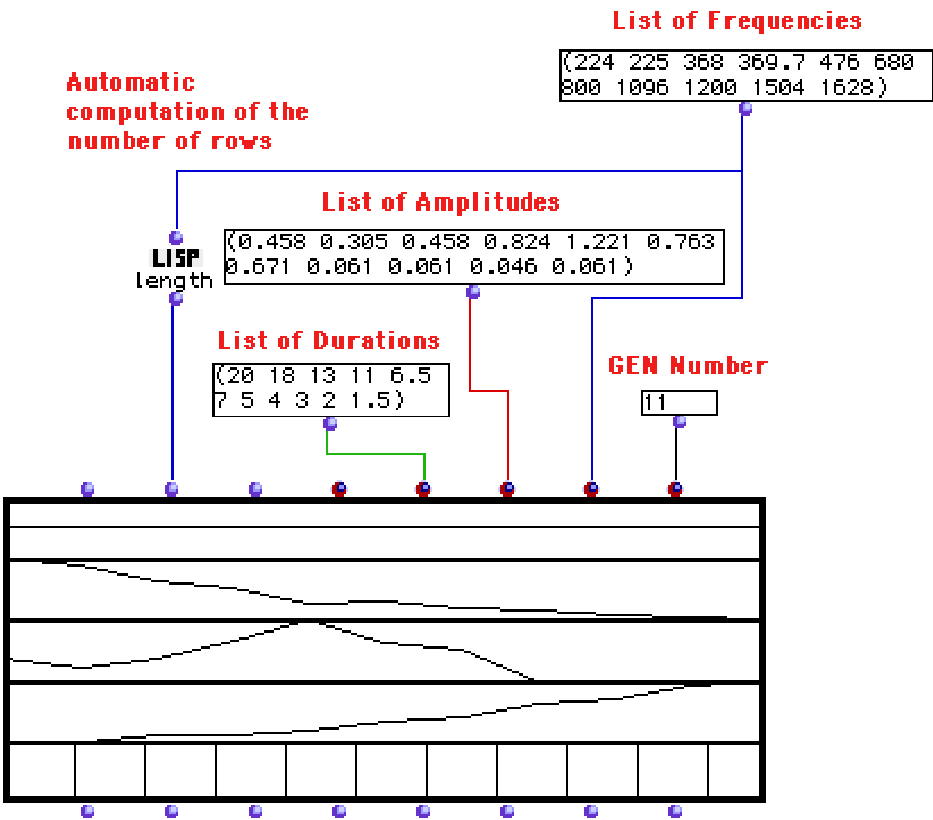
Freq	224	225	368	369. 7	476	680	800	1096	1200	1504	1628
------	-----	-----	-----	-----------	-----	-----	-----	------	------	------	------

Aenv	11										
------	----	--	--	--	--	--	--	--	--	--	--

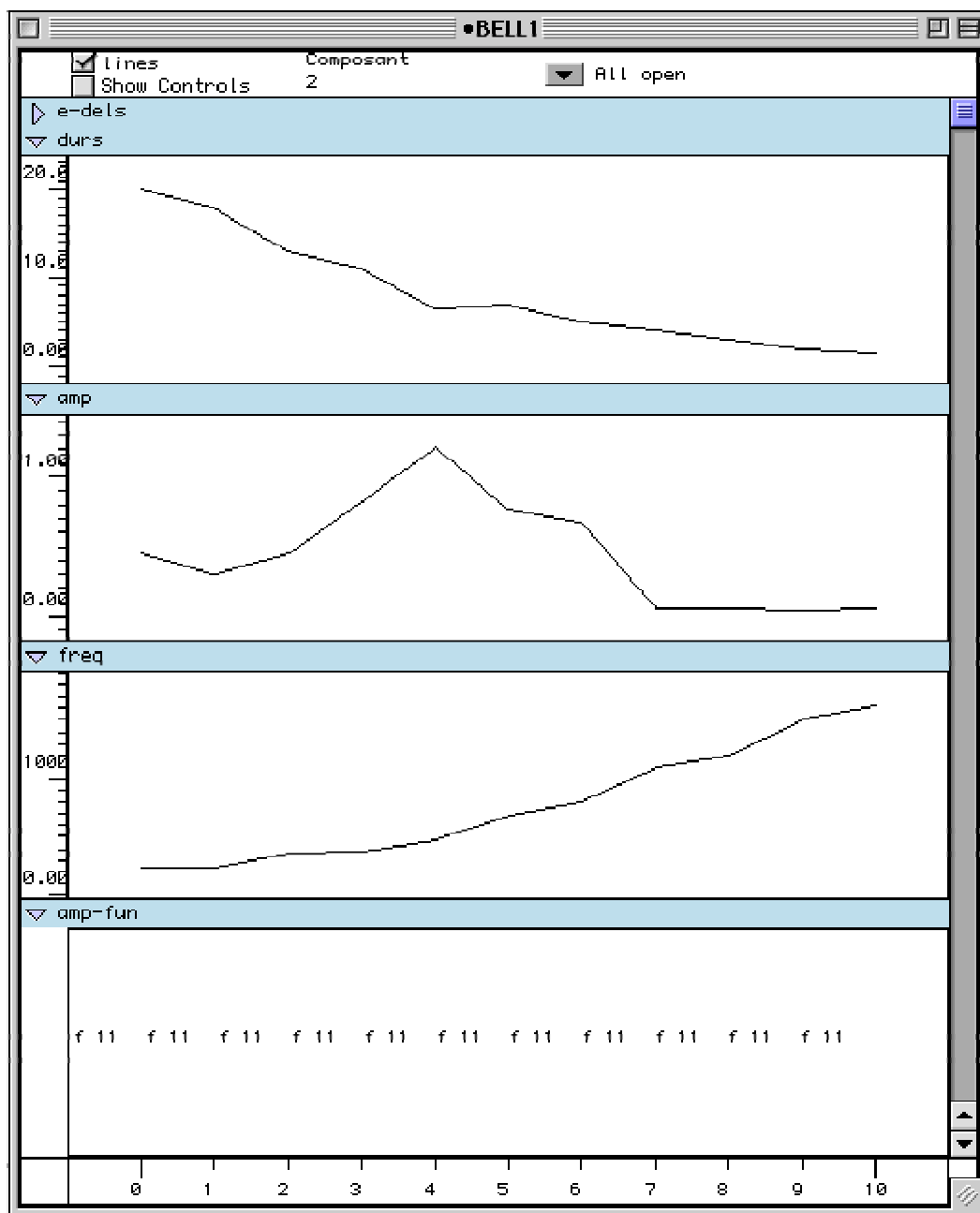
This is very similar to the representation examined in chapter 2, with the difference that the first row now corresponds to entry delays and not to action times. In Risset's example's all the entry delays are 0, the duration decreases irregularly as a function of the frequency, the amplitudes decrease in another

fashion, the frequencies are scalars of a reference frequency (400 Hz) but are expressed in absolute values and the amplitude envelope is an exponential table (here approximated by a linear GEN 7).

After filling all the values using simple OM functions and data, the patch will look like this:



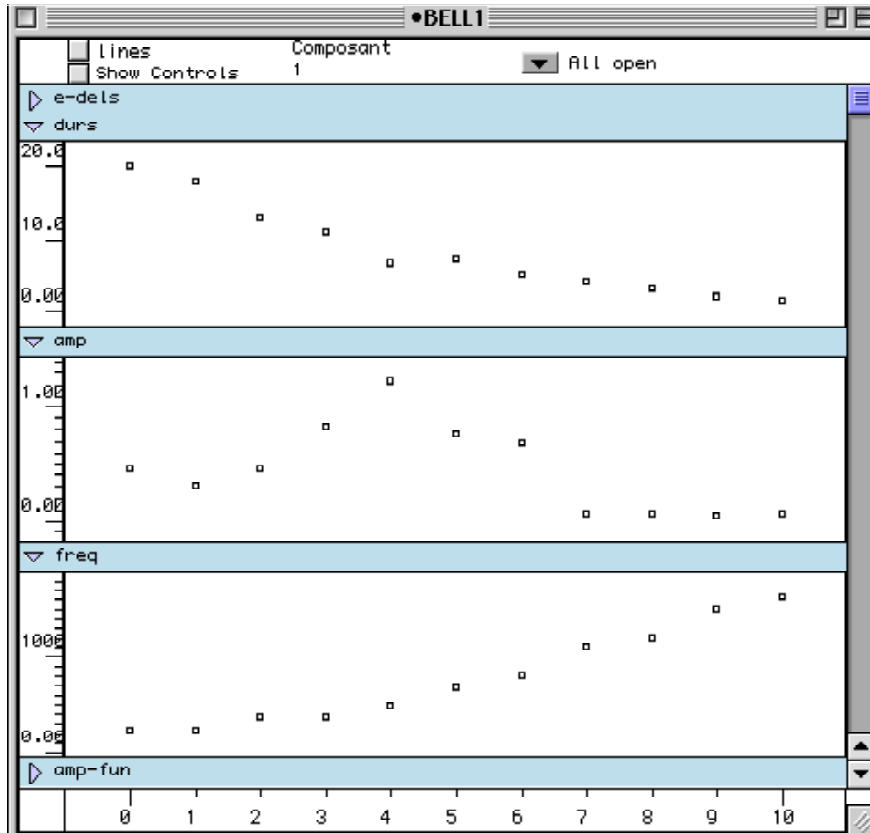
If you select the matrix, evaluate it and type "m" the shape of its values will be graphically displayed. You can also double click on it and look at the values using the editor.



The editor looks like a mixing window. On the Y-axis there are as many rows as slots¹. On the X-axis the "component's number" (Composant) starting from 0.

Clicking on the little triangle on the left of the slot's name will hide its values (here, for example, e-dels). Configurations of visible and hidden groups of parameters can be selected by clicking on the black triangle near "All open" and saving a given configuration.

Clicking on "lines" will uncheck it. The values appear then as single points instead of a function linearly connecting all the points (when the type is not a table).



As in all editors in OM, the values can be edited manually. Do not forget to block the matrix if you want to use an edited version of it, otherwise all the modified values will be lost as soon as the matrix is evaluated again.

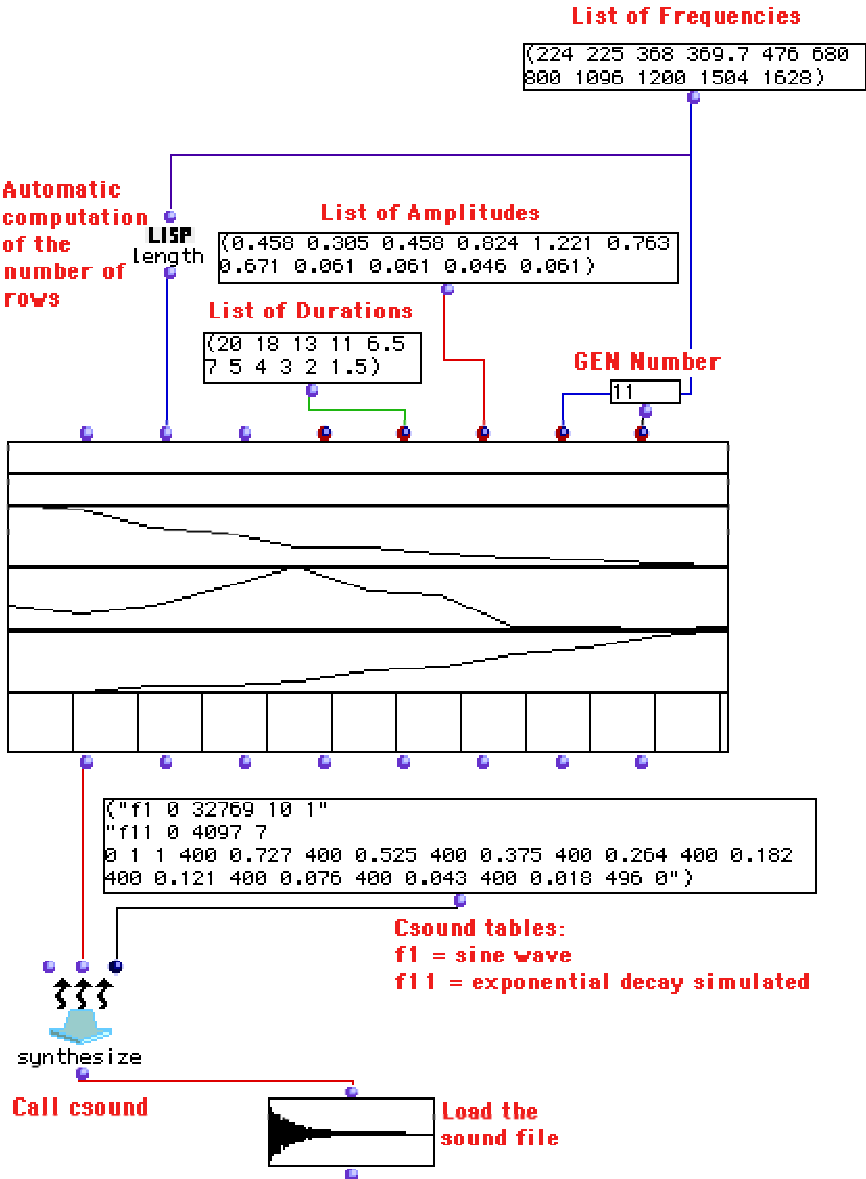
We will now connect this object to the generic function that performs the synthesis. Make an instance of the function "**synthesize**" in the patch and make sure that the value of its first inlet is set to "**csound**". Connect the leftmost outlet of the matrix to the second inlet from the left of "synthesize".

1. See 6.3 for an extension of this structure.

In order to perform the synthesis however, "synthesize" has to be given the GEN functions needed by the score, that is a sinusoidal function number 1 and an exponentially decaying function number 11.

To do this, select "synthesize", type "k" (lower case). A new indigo inlet appears. Each indigo inlet is a control value which requires two values: its "**keyword**" and "**data**". Click once on the inlet and type ":**tables**" (without double quotes), then shift-click again on it and type a list containing the exact definitions of the tables (in Csound syntax and within quotes). Finally connect the output of "synthesize" to a "sound" object.

At the end, the patch should look like this:



The first time the sound object will be empty. Evaluate it and wait until the whole process has ended. *OmChroma* will automatically run Csound, compute the sound and fill the empty box, as above.

A sound file called "**my_syntX.aiff**" (where "X", when present, is an integer number) will be created into the "Snd" directory specified in the preferences (see chapter 4).

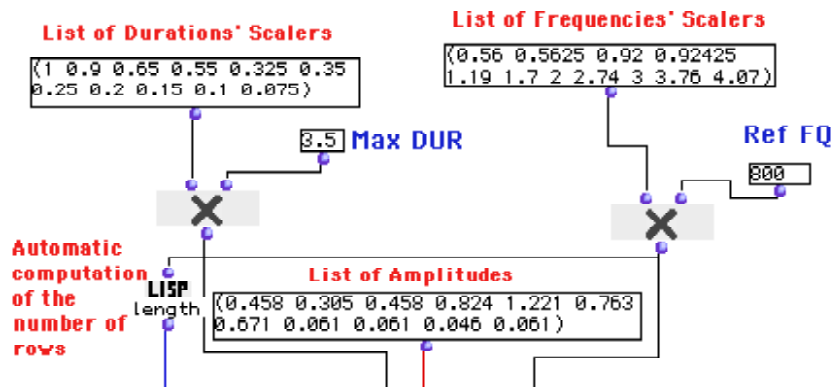
To hear the sound file, type "**p**" on the selected sound-file icon¹, but WAIT until the computation has completely ended (that is until the sound file has been loaded into memory). The computation ends when the word "Idle" (and not "Busy") is displayed near "OM" on the low left side of the Listener's window.



5-2) Generalizations of the previous example (TUT 01b-e)

For the time being the previous example is still very limited. It corresponds to the most straightforward and stiff application of the change of representation described in chapter 2. Aside from the graphical and colored aspects, there are little structural differences between this OM patch and the original Csound score (bell.orc and bell.sco).

We can now however start to generalize this example and transform it into a more abstract model. The first generalization (**TUT 01b**) consists in making the frequencies and the duration depend on a global value.



1. To stop it, type "s".

The list of the scalars of duration and of frequencies becomes a "model" of control that can yield different instances of the same sonic potential sharing similar morphological characteristics. Here the whole sound is shortened and transposed one octave higher.

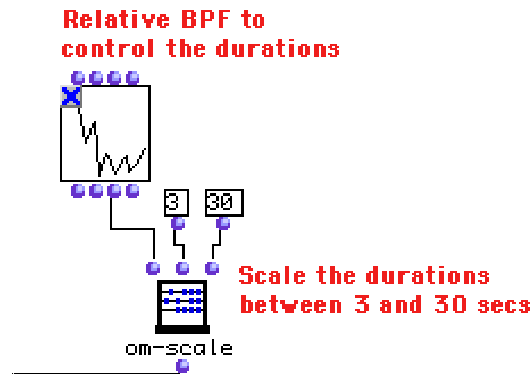
The next generalization (**TUT 01c**) introduces the BPF as a control structure. All the fields are controlled by BPF's rather than by lists of values. Independently on how many rows the matrix is made of, the BPF will be always sampled from the beginning to the end.

This is a crucial feature of *omChroma* and requires further explanations. If the sonic event consists of only 1 element (i.e. if the control matrix has only 1 row), the Y-value corresponding to the BPF's first X-point will be retrieved. If the event has only 2 elements, the first and last Y-values of the BPF will be retrieved. In all the other cases, the BPF will be sampled over the number of points. The more the points, the better the resolution.

On the other hand, when the input arguments are a list of values, as in the previous tutorials, the values will be retrieved one by one, from the beginning to the end. If a list contains more data than rows, the data after the one needed for the last row will be ignored. If it contains less data, the list will be started again from the beginning (see also the tutorial 04a/b for further details).

Here, in order to make the beating less repetitive, the adjacent frequencies of the two low partials were replaced by a small cluster of, respectively, 5 and 6 frequencies (448. 450.0 445.68 455.76 452.24 and 736.0 739.4 741.608 735.144 737.12 744.0).

The duration is controlled by a relative BPF scaled between 3 and 30 seconds.

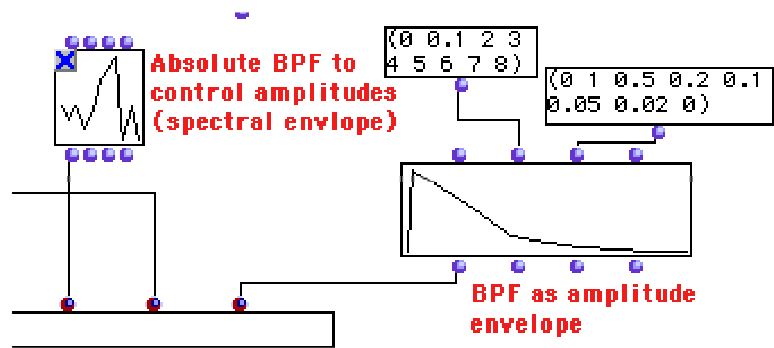


The X-axis of the BPF corresponds to the component's number in the order of definition of the frequencies, from the first to the last one in the list¹. The Y-axis displays the relative value of the duration of each component. This function is then scaled so that its lowest Y-value corresponds to a duration of 3 seconds and the highest to a duration of 30 seconds. Note that the highest and lowest Y-values can be anywhere in the BPF.

1. The frequencies are often, but not always, increasing, as in the tutorial 6.

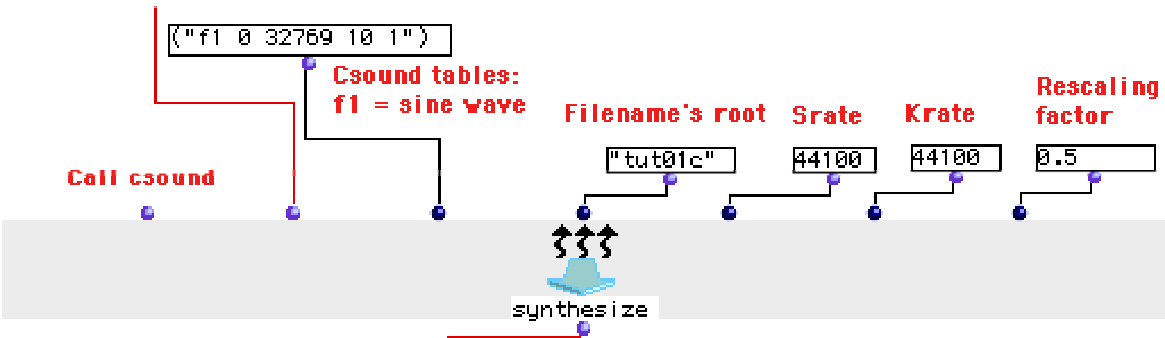
The amplitudes are controlled by an absolute BPF. Seen from this perspective the BPF corresponds to a sort of spectral envelope of the sound, although, more correctly, it describes the maximum amplitudes of the components from the first to the last one in the list of frequencies.

The amplitude envelope becomes a BPF as well. This is very convenient, since it frees the user from the obligation of passing the definition of the GEN as an argument to "synthesize", as shown above. *OmChroma* will automatically generate a GEN 7 (break-point function) starting at the GEN ID specified in the preferences (by default 1000). In this example the envelope is less percussive than the previous amplitude envelope, hence resulting in a slightly softer attack.



The function "synthesize" also reveals some newer controls that "personalize" its behavior.

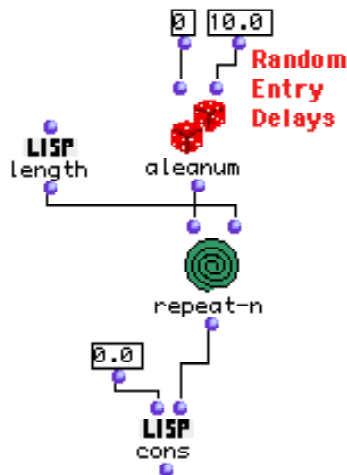
First of all, because of the automatic definition of the amplitude envelope, only the audio sine is passed and an argument to "tables". Selecting "synthesize" and typing "k" (lower case) will make new slots appear. As before, click on each slot, type in the following keywords (in any sequence, "tables", "name", "sr", "kr" and "rescale"), then shift-click on each of them again and type in the values. A sound file called "tut01cX.aiff"¹ will be generated at the place of the default name. Csound will run with sampling and control rates of 44100 Hz and the final sound file will be rescaled to 0.5², that is half the maximum amplitude. All the computation is done floating point.



1. As usual, X is an integer number that will not be displayed the first time the patch is evaluated/
2. The rescaling phase is performed by calling a plugin of Diphone.

The final generalization demonstrates the power of abstraction contained in the model of representation chosen in *omChroma* (TUT 01d). The patch is the same as the previous ones, but we will now generate aleatoric entry delays for each component by repeating the function "aleanum" as many times as there are elements in the list of frequencies¹.

However, since "aleanum" will probably not generate a value of 0 and since we want the sound to start at 0, "0.0" is added at the beginning of the list. The first (and lowest) component will always start at 0. Each time this patch is evaluated a completely different set of entry delays is generated. In this example the values have here been purposely exaggerated. By setting aleatoric values on the order of the onset time of transients (roughly 10 to 500 msec, depending on the shape of the amplitude envelope), one can simulate very living attacks changing each time the patch is evaluated.



1. It is the possibility of independently controlling the onset time and the duration of each partial that was not possible in the first two implementations examined in the chapter 2.

5-3) Chant's FOF's (TUT 02)

This tutorial shows how to use *omChroma* to control Formantic Waveforms (FOF's). The patch will generate an "SDIF" file and call the Chant plugin from Diphone.

Similar control structures (lists of values) as the ones used for the first tutorial are employed. This demonstrates the essentially abstract nature and the portability of the controls paradigms of *omChroma*.

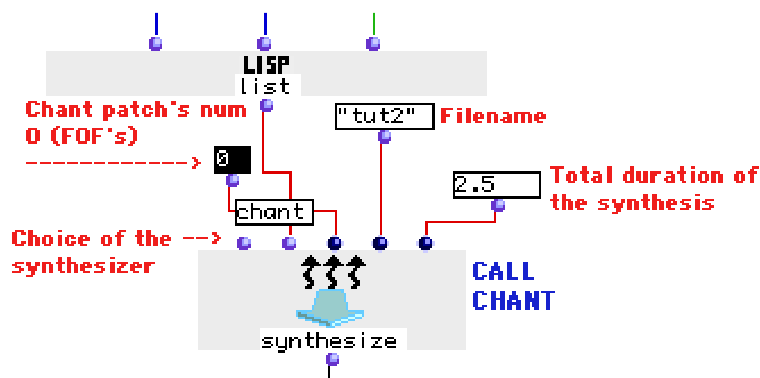
Open the patch TUT 02.

It contains three factories of the class CH-FOB-EVT (CHant-FOF-Bank-EVENt). The first factory starts at time 0.0 and generates the vowel "O"¹. The third factory starts at time 1.0 and generates the vowel "I". The middle factory has the same values as the first one and starts at time 0.8.

This means that the Chant synthesizer will hold the values of the first factory between 0.0 and 0.8, then make a linear interpolation between them and the values corresponding to the vowel "I" for a duration of 0.2 seconds. Had the second factory not been there, the synthesizer would have started the interpolation from the beginning of the sound.

Each factory has a maximum of 8 formantic triplets (frequency / amplitude / bandwidth). This example uses only the first 4 triplets.

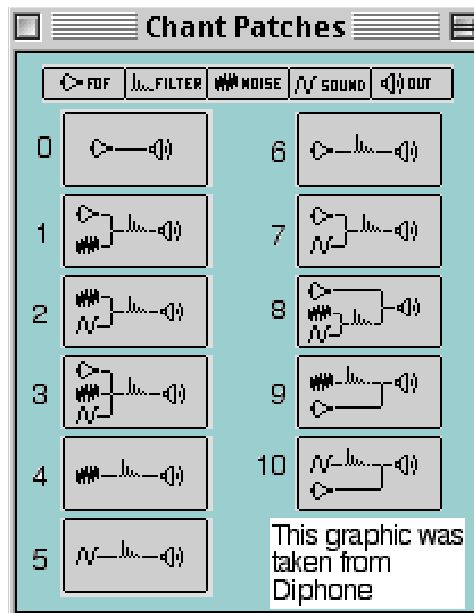
All the events are collected into a list and given to the generic function "synthesize" that will perform the synthesis. If the arrow of the mouse is approached to the first "inlet" the name "chant" will appear, thus showing that this is now the current synthesizer.



When run with Chant "synthesize" needs two special arguments (whose keywords are **":patch"** and **":duration"**), which specify which patch is being used and how long the synthesis should be.

For information's purposes, if the box named "chant-patches" is evaluated, all the currently available patches will be displayed (see the documentation of Diphone for further details about them).

1. The formantic data were taken from a data base of vowels found in an old manual of Chant.



A file called "**tut2.aiff**" will be computed in the "Snd" directory. If the flag is enabled, an intermediate file called "**tut2.sdif**" will be placed in the "Inter-files" directory. For debugging purposes, the file can be opened and read using the SDIF library.

This tutorial will produce a simple change of vowel and a portamento of fundamental frequency over a descending minor third from E3 to C#3. The quality of the synthesis is still relatively rustic: since there is no amplitude envelope, a click starts and ends the sound. There is also no vibrato on the fundamental frequency, nor rules that relate the formant's frequencies with the fundamental frequency. If the purpose is to imitate a natural vocal sound, much additional information is mandatory.

The Chant SDIF synthesizer does not provide any high-level controls for these parameters. In order to generate more convincing sounds, it is necessary to find better control strategies, have more elaborated instruments and more sophisticated paradigms. These strategies therefore either are realized within Diphone (when Chant is called by it) or must be implemented in *omChroma*. Some examples of the kind of higher-level controls that can be envisaged will be described in the tutorials 12 and 13.

5-4) Chant's filters (TUT 03)

This tutorial uses the same control structures as the previous one, but calls the Chant's patch number 4 (noise into a bank of filters).

Open the patch TUT 03.

It looks the same as TUT 02, but the factories are of the class CH-FILT-EVT (CHant-FILTer-bank-EVEnT) and the chant's patch is number 4. The result is an unvoiced vocal change from "O" to "I" without fundamental frequency.

A small modification was nonetheless added: the bandwidths of the first vowel have been multiplied by 0.01, hence resulting in very small values and yielding resonating formants. During the transition (lasting 1.5 seconds) the bandwidths increase until 0.3 times the original ones and the sound gets noisier. The glissando of the first formant frequency (one descending octave) is very clearly perceptible.

Tutorials 2 and 3 show how easy it is to keep similar control structures in *omChroma* and to steer different synthesizers.

6 STEP-BY-STEP TUTORIAL

This chapter will progressively introduce all the main control paradigms in a detailed and exhaustive way.

6-1) Simple Matrix (main control object) (TUT 04a-b)

The main control structure of a sonic "event" is the "matrix", a special class in the kernel of *OpenMusic*. This tutorial shows all the possible controls that can be used within a general matrix, before it is specialized as a synthesis class (for Csound, Chant or other synthesizers). A synthesis class inherits from this original matrix.













The tutorial 4a shows two matrices: the upper one is of the class ONE-ARRAY. When such a matrix is instantiated only the light-blue inlets will appear. To make the red inlets appear until all the slots of the class are shown type "**alt -->**". To make them disappear, type "**alt <--**"¹.

The class is found in the menu "**Classes / user / omChroma / arrays classes**" or through the folder "packages" in the Workspace:



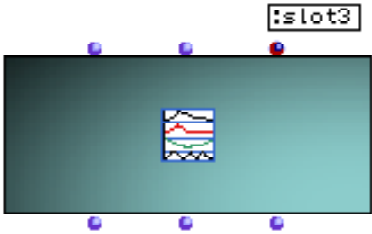
1. This the standard OM procedure for functions with a variable number of arguments.

Each slot has the keyword which was specified when the class was created.

ONE-ARRAY				
Slots	Show	Allocation	Default value	
 slot	<input checked="" type="checkbox"/>	 instance	0	
 slot1	<input checked="" type="checkbox"/>	 instance	0	
 slot2	<input checked="" type="checkbox"/>	 instance	0	
 slot3	<input checked="" type="checkbox"/>	 instance	0	
 slot4	<input checked="" type="checkbox"/>	 instance	0	
 slot5	<input checked="" type="checkbox"/>	 instance	0	

By default, the red inlets will appear in the same order of definition as the slots, but they can in fact be placed in any order and any amount. If some slots are not shown, their default value will be used.

For instance, if you want to control only the slot with keyword `":slot3"`, just instanciate a factory of the class ONE-ARRAY, have one single red inlet appear, click once on the inlet and type in the keyword `":slot3"`.



To assign a value to it "shift click" on the inlet, as usual.

The arguments of a matrix can be:

BLUE INLETS

Self: the object itself.

numrows: the number of rows of the matrix.

RED INLETS

Their meaning depends on the type of the slots in the class. In the class ONE-ARRAY they are all of type "real number".

Let's call "N" the number of rows and "item" a single element of the right type. The possible control structures and their effect are:

Case 1: single item. Action: the item will be repeated N times.

Case 2: list of items. Action: each item will be assigned to a different row, from the first to the last one. If there are more than N items, only the first N's will be used. If there are less than N items, the missing values will be retrieved starting again from the beginning of the list as many times as needed.

Case 3: patch or algorithm generating a list of items. Action: the patch will be evaluated once when the matrix or the corresponding outlet is evaluated, then as in case 2.

Case 4: BPF. Action: the BPF will be sampled N times, starting from the first and ending with the last X-point. The corresponding Y-values will be retrieved.

Case 5: function or lambda-patch with one argument. Action: the function is called when the value of each row is computed (hence N times) with an integer number between 1 and N as an argument.

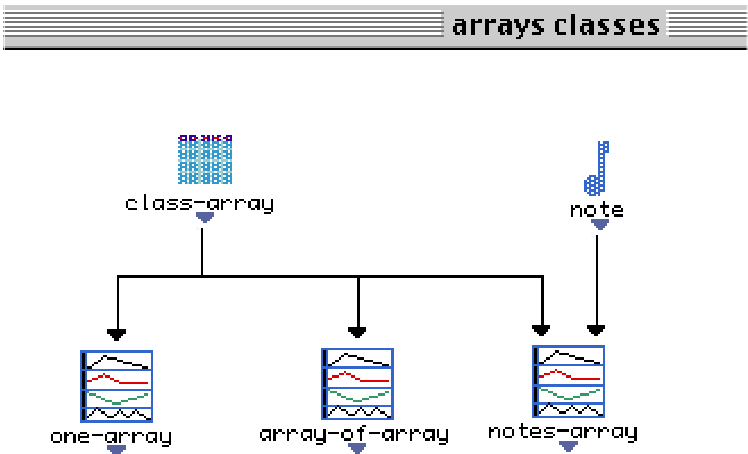
Case 6: function or patch with more than one argument. Action: only one argument must be kept free, the others have to be coerced to a fixed value before the function is called. Then as in case 5.

The lower matrix in the tutorial is of type ARRAY-OF-ARRAY and has two slots, but the first one is of type ONE-ARRAY.

ARRAY-OF-ARRAY			
Slots	Show	Allocation	Default value
 slot	<input checked="" type="checkbox"/>	 instance	
 slot1	<input checked="" type="checkbox"/>	 instance	0

The matrix will therefore contain instances of single matrices.

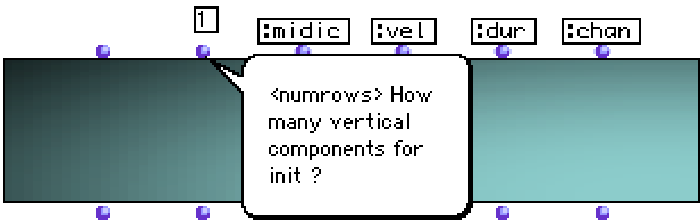
The tutorial 04b shows the power of inheritance in object-oriented programming. The class NOTES-ARRAY inherits from both the general class "CLASS-ARRAY" (the general class for matrices, which has only the blue slots) and the class "NOTE".



To display the structure of classes just click on the upper half of the folder "arrays classes" in the "packages" folder.



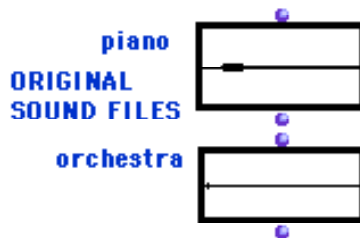
The slots of the class "notes-array" will automatically inherit the keywords from both classes.



6-2) Musical controls of a single matrix (TUT 05)

This tutorial shows musically pertinent ways of using the control structures described above and introduces further control paradigms. It uses a matrix of class SIN-EXP-DB1 corresponding to the simple additive-synthesis instrument **"sin-exp-db.orc"** found in the folder **"chroma / cs"**.

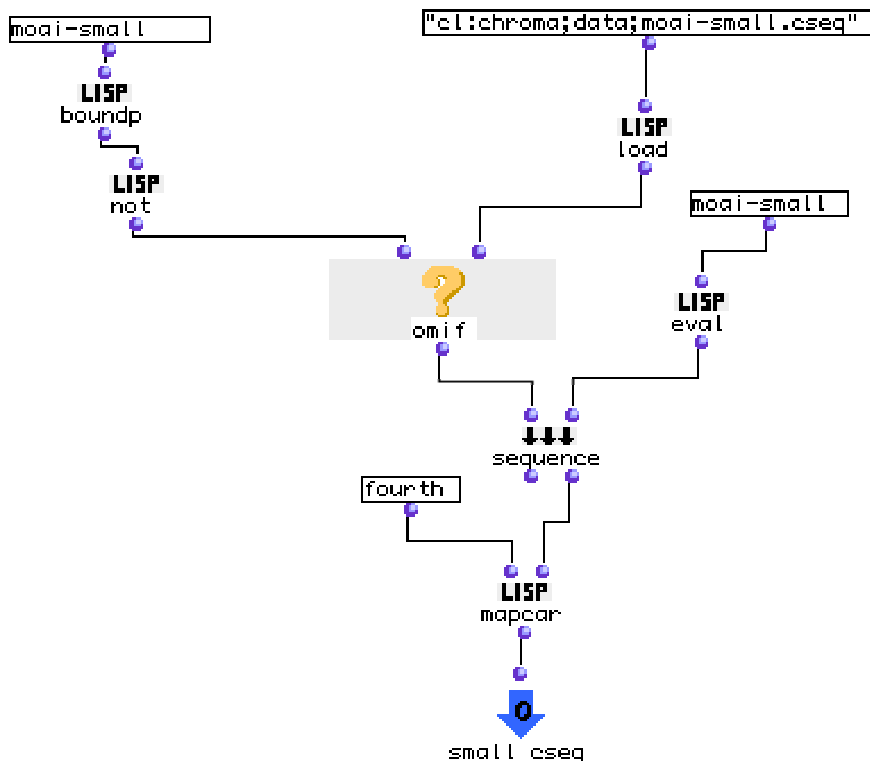
The starting point is the chord at the beginning of my piece **"Moai"** (one of the Miniature Estrose, 1991-95 for piano solo). The first page of this piece was orchestrated at the beginning of **"Hiranyaloka"**, a piece for orchestra composed in 1993-94. The piano chord's low register (left hand) is to be played as soft as possible, while the right hand plays as loud as possible. The sonogram of the piano recording, however, does not show this difference of amplitudes as clearly as the sonogram of the orchestra (probably due to the poor resolution of the picture). To hear the original sounds, you might have to load the files **"moai.aiff"** and **"hiranyaloka.aiff"** from **"OM 3.7 / example files / Aiff"**.



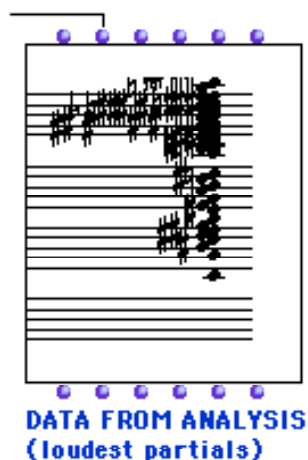
We will now make a "synthetic" orchestration. The original piano chord of 8 notes was "enriched" for the purpose of synthesis by transposing the notes of the lower part one and two octaves above.



The orchestral sound file was analyzed using Audiosculpt. From this analysis the loudest partials were extracted and written onto a file. The patch **"get freq"** tests whether the analysis data have already been loaded into memory. If not, it will load the analysis file **"moai-small.cseq"** - coming from Audiosculpt after a little manual editing (cseq = Chord SEquence) - and extract the frequencies from it.



The Chord object containing the analysis data patently shows the lack of the lower partials.



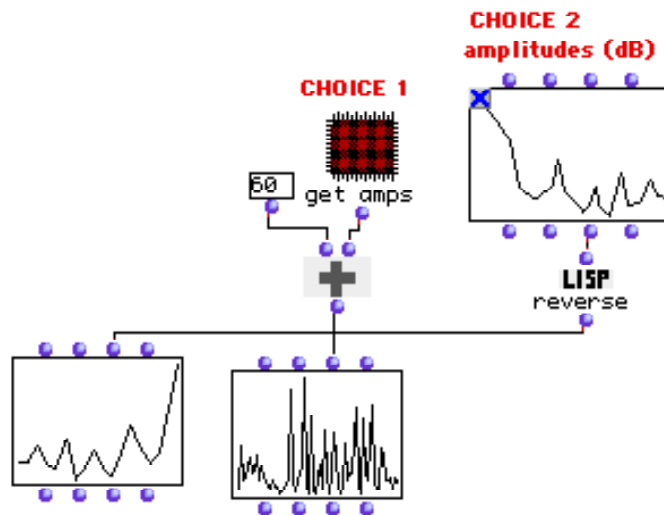
If you connect the outlet of either "get freq" or "mc->f" to "length" and to the slot of keyword "':freq'" of the event, you can choose which frequency material to use. The original tutorial uses the analysis data.

The other slots receive the following data:

e-dels: lambda-patch (ran ed) containing the same algorithm as in tutorial 01d with random entry delays between 0 and 10 msecs.

durs: the basic control BPF has been drawn and blocked. Its absolute range is between 3 and 10 seconds. A global scaler allows for a global modification of this range. A new BPF is then constructed, so as to be sampled over the whole range of components¹.

amp: two choices, depending on which object is connected. **Choice 1** (default): the patch "get amps" will load the amplitudes directly from the analysis model. Since the retrieved amplitudes are negative dB values and that the instrument requires positive values (we could of course write an orchestra accepting negative values), the number 60 is added to each amplitude so as to make the amplitudes positive. A BPF connected to the output graphically shows the curve of the amplitudes. Either the output of "+" or of the BPF could be connected to the corresponding slot, provided that the total number of components is the same². **Choice 2:** an arbitrary, formant-like BPF is drawn and blocked, then reversed in order to interpret the dynamics of the piano chord (lows are soft, highs are loud). Connect this box to the corresponding inlet if you want to use this choice. We could have also connected the output of "reverse" to the corresponding inlet, but we built another BPF in order to verify the effect of "reverse" graphically. In addition, if the number of components is to change, the behavior of a list or of a BPF as input data will be different (see tutorial 04 and the footnote 25).



aenv: the choice of the amplitude envelopes is here more sophisticated. **Choice 1:** a BPF-LIB was manually created and blocked. This is equivalent to a list of BPF's (either outlet of BPF-LIB could be con-

1. If the output of the multiplication (a list of Y-values) were directly connected, the matrix would consider it as a list and not as a BPF.
2. As seen above, this is not semantically equivalent, since the values of a list will be used sequentially and started again if they are not enough, while a BPF will be always sampled from the beginning to the end. The two control paradigms yield the result only when the number of points in the BPF is the same as the number of elements in the list, as is the case here.

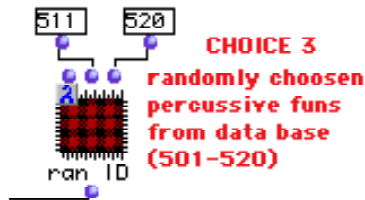
nected to the inlet of the matrix): as with all lists, each table is used sequentially and when the list is completed *omChroma* will read it again from the beginning. **Choice 2:** the patch "bpf-lib" constructs a BPF-LIB by assembling a list of BPF's, while the patch "ran bpf lib" will aleatorically choose one of them for each component. **Choice 3** (default): the patch "ran ID" generates an aleatoric number between 501 and 510. The matrix understands that the number corresponds to an absolute GEN ID and stores it as such. The definition of the GEN's with the same ID's will have to be passed as an argument to "synthesize".

"Synthesize" presents only one new feature: the arguments of "tables" contain a list with two sorts of data, an absolute GEN ("f1 0 4097 10 1"), and a relative path ("cl:chroma;cs;cs-funs.lisp"). This means that the file "cs-funs.lisp" will be loaded together with the GEN. In our case it is the data base of percussive GEN's having ID's between 501 and 520.

The format of the file is a sequence of definitions with the syntax (om::ScSt "<GEN>"), where "<GEN>" is a specification of a GEN in the syntax of Csound.

Ex: (om::ScSt "f 501 0 513 5 1 513 0.00001")

The first 10 functions in the data base are more or less exact approximations of descending exponential amplitude envelopes. The functions between 511 and 520, while still having a percussive attack and an approximated exponential shape, contain a sort of "bump" in the amplitude towards the end. To hear the difference, simply restrict the range of "ran ID" to 511-520:



You can of course add more GEN's in the file or modify the existing ones.

6-3) Control slots and parsing fun (elementary usage): single event (TUT 06)

This tutorial introduces an extremely powerful control paradigm: the "parsing-fun".

In the tutorial 01d I remarked that to make the beatings of the bell-like sound less periodic, further components around the first two frequencies are to be added. This concept can be further generalized, by positing that "every" component of an "event" can be considered as a "**micro-cluster**" of "**npart**" components spread around the frequency of the main component by a "**ston**" percentage.

For instance, instead of being made of a set of N sine tones an additive-synthesis bank will be conceived as a set of N clusters. When the density of the cluster is 1 (npart = 1), it is like a simple additive-synthesis bank. I have been using this model of control since the beginning of my work in Padua and am convinced it is a very flexible paradigm.

To implement it, however, new components will have to be added to the score around each main component. These new components will be called "**sub-components**". *OmChroma* provides a built-in feature that allows the user to express this sort of control paradigms in an elegant and efficient manner: the "**parsing-fun**".

Open the patch TUT 06.

The basic material is a factory of the class CHORD-SEQ containing a sequence of chords computed by Joshua Fineberg. The patch "**cs-events**" builds a single very large matrix passed as an argument to "synthesize". Open this patch. The matrix has inlets of three colors: blue, red and indigo. Indigo inlets are control inlets that belong to the matrix's instance and not to the class. To make them appear type "**k**" (lower case) after selecting the matrix and "**K**" (upper case) to make them disappear. Control inlets are of type real and are processed in exactly the same way as red inlets¹. Their default keyword is "**ki**" where "i" is an integer number starting from 0.

Two keywords have a special significance as control inlets:

:parsing-fun: function or a patch of two arguments (**matrix comp-num**), where "**matrix**" is the whole matrix itself and "**comp-num**" is the number of the current component (integer from 0 to "numrows-1").

:precision: integer number specifying the precision (maximum number of digits) used in the Csound's score (default = 6, for example: i1 0.002 7.781 34.367 1014.830 512.000). See the Lisp "format" syntax for further details (not used in this tutorial).

The control of the red inlets in the patch is quite straightforward and will not be further explained. I will just observe that all the temporal information in the sequence of chords (onset and offset) is translated into entry delays (see the loop "make-ED").

The three indigo inlets have the following keywords:

npart: number of sub-components around each component (0 means no sub-components).

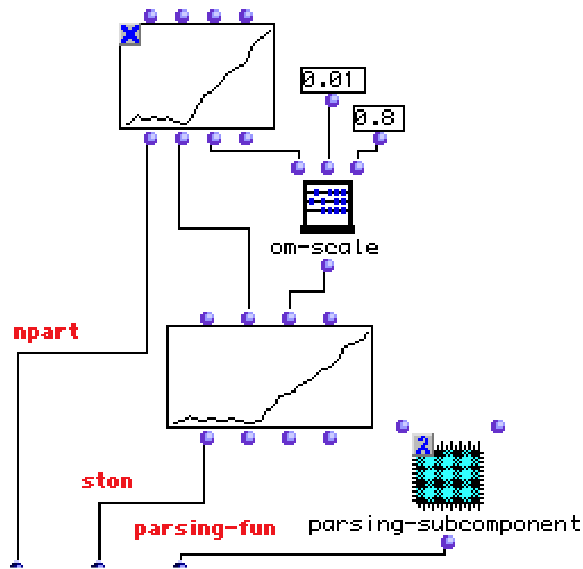
1. Except, as I mentioned, that they belong to a specific instance of the class.

ston: "mistuning" (from the Italian "stonatura") of each sub-component (0-1, linear, 0.5 is 50%, that is one octave down and a fifth up maximum).

parsing-fun: special keyword, see below.

The patch "**parsing-subcomponent**" will generate "npart" sub-components aleatorically spread "ston" apart around each component. For instance, if the frequency of the component "i" is 1000.0 and "ston" is 0.1, the aleatoric range will be between 900 and 1100. A semitone corresponds thus to a little less than 0.06¹.

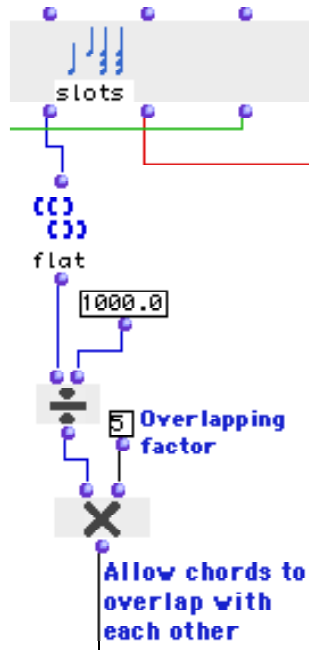
In our example "npart" increases between 2 and 37 following the curve of the BPF shown below.



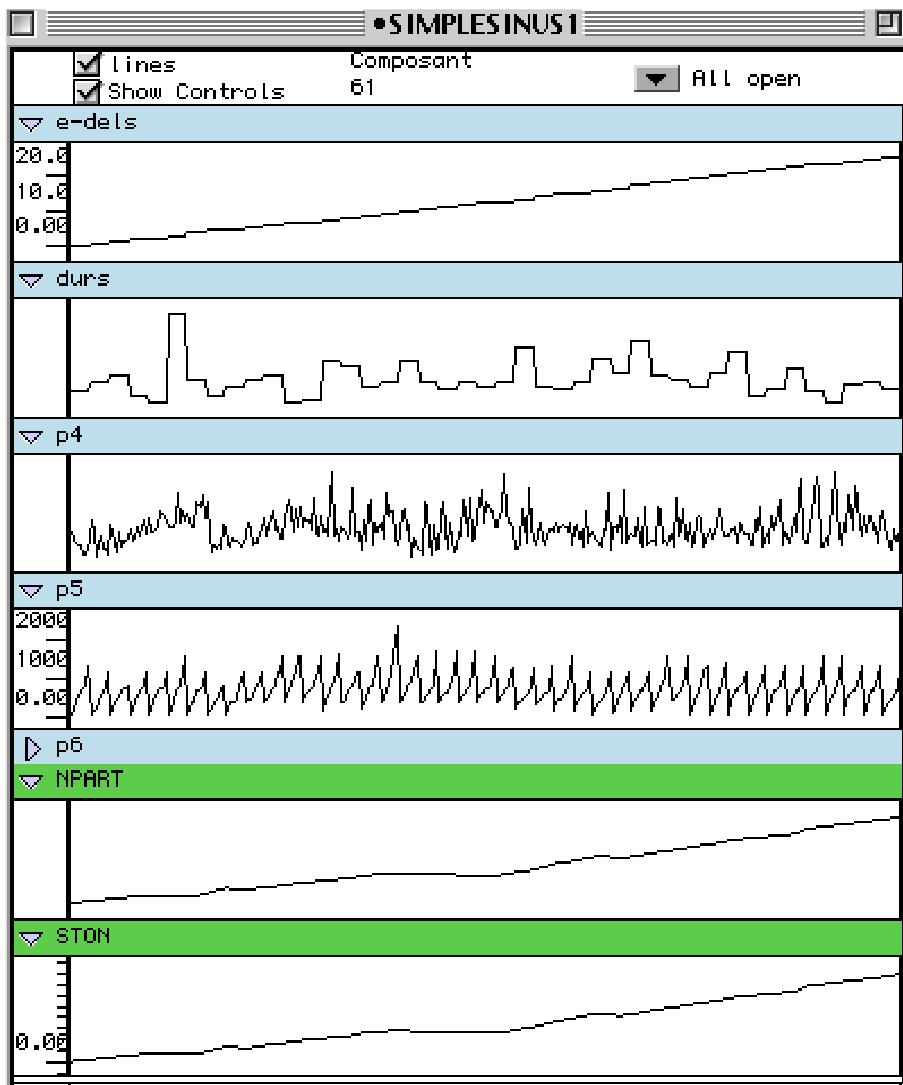
The same curve (scaled between 0.01 and 0.8, which is a tremendous amount, used here for the purpose of the demonstration!) also controls the distribution factor of the sub-components (ston). The resulting file will sound at first like a sort of "choir" of sinusoidal instruments. Toward the middle of its length it will rapidly turn into a large additive cluster, like a singing crowd of drunken people!

1. A logarithmic instead of a linear range should be implemented if the interval must be the same in both directions.

Try changing the values and the BPF's controlling "npart" and "ston" to generate other effects. All the chords overlap a great deal because of the overlapping factor added when computing the duration of each chord.



Finally, since the original material is a CHORD-SEQ, it can also be played as such by any MIDI synthesizer. Control fields are not shown in the icon of the factory, but can be edited by clicking on "Show Controls" in the editor.



Note that this factory is an instance of the class "SIMPLESINUS" and that no tables are specified. The default triangular GEN specified in the preferences is used.

Open now the patch "**parsing-subcomponent**". This patch, which must be a "lambda patch" with two inputs (matrix, comp-num), is evaluated once for each component after the computation of its red inlets. This is a crucial point, since it means that the "parsing fun" will access the final value of the fields after all the rules and other processes have been called. The output of a parsing fun must be a list containing all the components that ought to be passed to "synthesize". The list may include comments (strings of characters that will be automatically preceded by a ";" in the score).

"Parsing-subcomponent" shows a typical structure of a "parsing fun". All the values of the current component's slots are at first accessed through the function "**get-comp**" (let's remind that "comp-num" refers to the current component and is automatically updated). The component is added onto a list containing a comment¹ (giving the number of the current component) and the component itself. Then the same component is passed to the patch "**subcomponents**" which will compute the extra sub-components and return them in a list starting with another comment and appended to the previous list. The core of the patch is the loop "**freq-dev** which receives the frequency (p5) and the control fields (npart, ston) together with the component itself, computes all the sub-components and returns them in a list. The patches are thoroughly documented and it should not be too hard to figure out what they are doing.

Study this example carefully, so as to achieve a perfect mastery of its different phases and of its control and data-structuring functions (cons, append, list, where to put the "new-line", etc.). It is important that the returned value be a straight list of either components or comments (strings of characters) and not a list of lists or other data structures, otherwise an error will be produced.

In the score file the structure of the event appears very clearly:

```
;COMPONENT No. 10
i1 0.430 2.800 0.299 232.544 1010.000
;
;2 sub-components
i1 0.430 2.800 0.299 232.661 1010.000
i1 0.430 2.800 0.299 234.700 1010.000
;

;COMPONENT No. 11
i1 0.430 2.800 0.425 388.389 1011.000
;
;3 sub-components
i1 0.430 2.800 0.425 394.672 1011.000
i1 0.430 2.800 0.425 397.810 1011.000
i1 0.430 2.800 0.425 389.026 1011.000
```

Generating sub-components does not modify the original matrix, but adds new score lines to the score. Because of the large amount of sub-components at the end, this example is relatively long to compute.

1. Adding comments is very convenient to make the Csound score more readable.

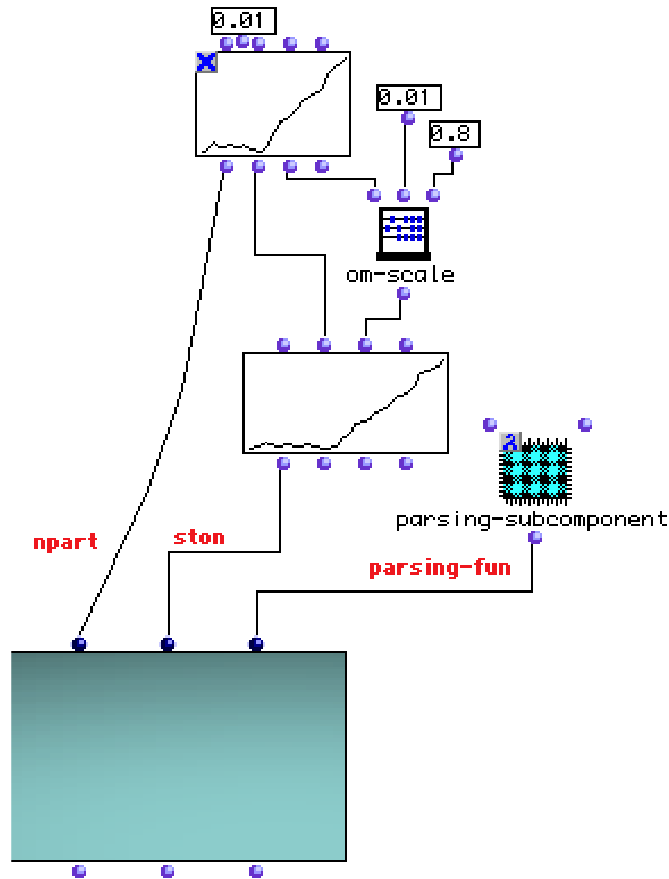
6-4) Control slots and parsing fun: many matrices (TUT 07)

This tutorial implements another way of realizing the same control paradigm as the previous one. Instead of generating one single very large matrix, we will produce the same sound with a different matrix for every chord in the sequence. However, as we examined in chapter 2, this paradigm embodies a slightly different "concept" about sound control, which will highlight some processes, but will also make others more obscure or difficult to implement.

The patch **"cs-events-list"** is extremely simplified, since all the computation is performed within the loop **"make-event"**.

The basic structure of **"make-event"** is a set of **"listloops"** reading the values of each chord and filling the slots of the event after having converted the data in a fashion similar to the one implemented in the tutorial 06. Most of the controls remain the same. The only substantial change is that the onset time of each chord is assigned to the matrix's **"start-time"** while the offset of each chord's note is simply fed into **"e-dels"** (in this example they are all 0).

The major modification concerns however the control inlets. If we had connected the BPF's directly to the matrix, as below:

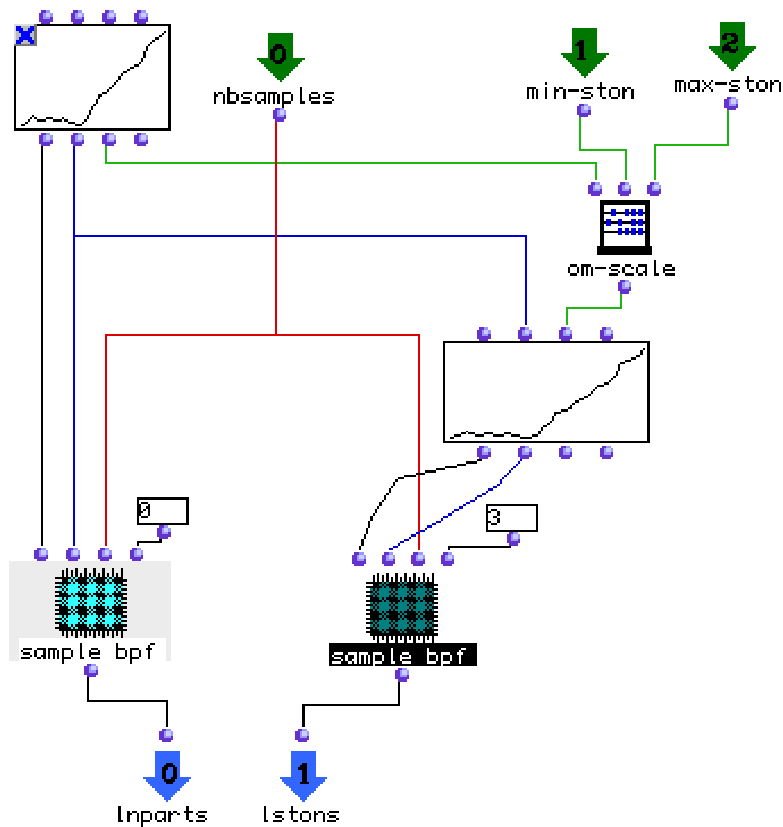


We would have had "npart" and "ston" increasing between respectively 2 and 37 and 0.01 and 0.8 within each single matrix. Each chord would have received the same pattern: the low frequencies with relatively few sub-components and the high frequencies with very many. In addition, since each chord has only few notes, the BPF would be very poorly sampled. This may be interesting, but it is not what was implemented in the previous tutorial, where the values of "npart" and "ston" progressively increase across the whole process. A control of a higher level needs to be devised, a control applying a BPF to the whole list of events.

There are several ways to do it more or less exactly. For simplicity's sake, let's suppose that within one event the value of "npart" and "ston" does not change and that the difference between having values continuously evolve inside an event (as in the tutorial 6) or having them continuously evolve across events (but constant within) is negligible.

The patch "**kontrols**" implements this task: it receives the number of chords in the sequence (nbsamples) and the range of "ston". Since the BPF for "npart" already gives the absolute values, it need not be further rescaled.

The control function is sampled as many times as there are chords in the CHORD-SEQ and will return a list with the right amount of "nparts" and "stons".



The parsing-fun "parsing-subcomponent" is the same as in TUT 06. Sampling the BPF that controls "npart" and "ston" has to be made within the main patch "cs-events-list" so as to give "make-events" a list of values, i.e. a data structure compatible with the structures of a chord-seq. The patch "sample bpf" is just a simplification of the OM function "bpf-sample"¹.

The tutorials 6 and 7 show how important it is to imagine and implement control paradigms matching the composer's needs as closely as possible.

1. The OM function "sample-bpf" is a little tricky to use. The two instances of "sample bpf" simplify this task. Note that the left patch has 0 "decimals" (integer number), while the right patch has 3.

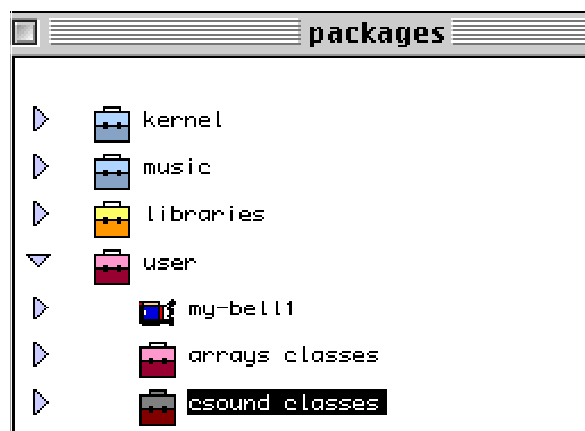
6-5) Instanciation of a new class (TUT 08)

Let's now imagine that a "parsing-fun" generating sub-components for every component is regularly employed. Since control inlets belong only to an instance, each time a factory is created three control inlets will have to be created and their keywords edited.

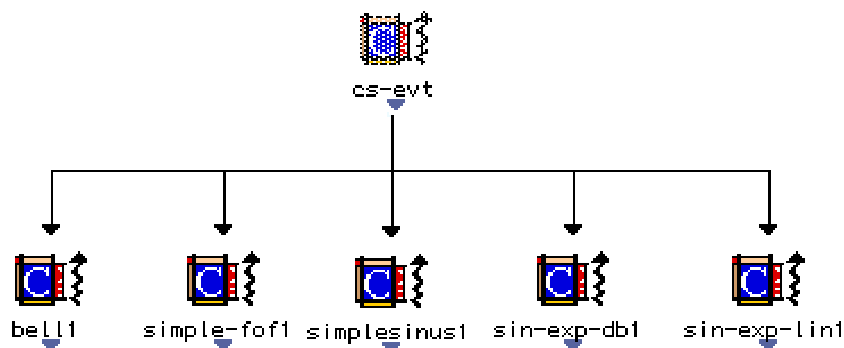
It may be more efficient to create a new class which will include the needed control parameters in its slots, so that each time a new instance is created these slots will appear as red inlets. Let's call them in the same way as above, "npart" and "ston"¹.

This tutorial has the same structure as the previous one, but uses the new class BELL+ which inherits from "bell1" with two more red control inlets.

Click on the upper half of the package " classes".



After some editing the class structure will look like:



1. Since "parsing-fun" and "precision" are reserved keywords, they cannot be added to the slots of a class.

Create a new class (in the menu "File / New Class"), call it, for instance, "bell+" document it...

Class Name

bell+

Documentation

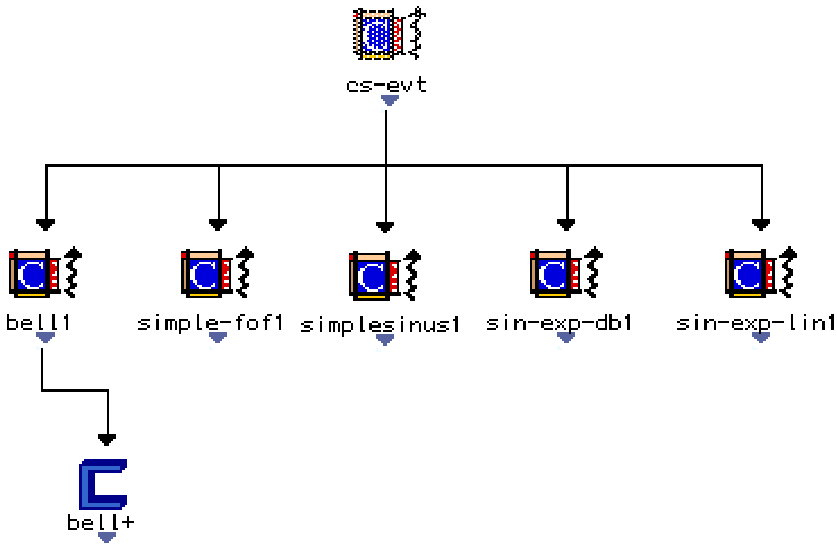
Same as bell1 with 2 added slots.

Icon

cancel

OK

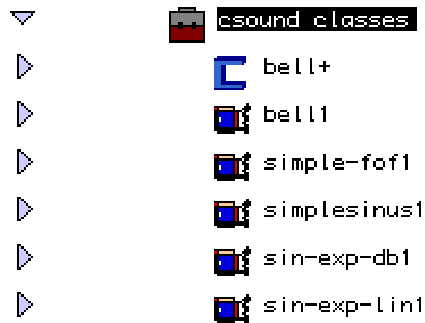
... and have it inherit from "bell1" by dragging an arrow from "bell1" to the new class.



Double click then on the new class, add "New Slots" (from the File menu) and assign a name and a default value to both of them. By definition the type of all the new slots is a real number.

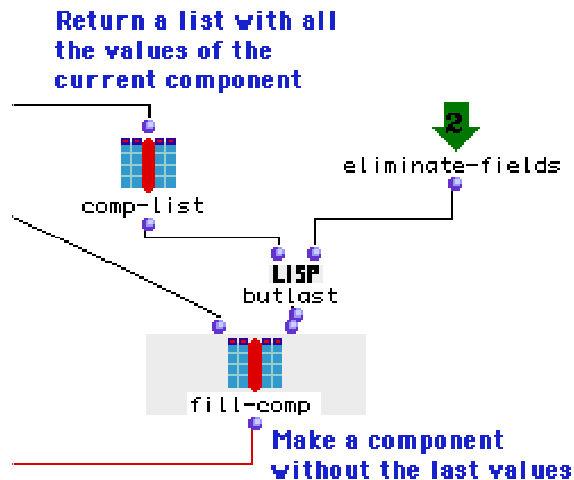
BELL+			
Slots	Show	Allocation	Default value
npart	<input checked="" type="checkbox"/>	instance	1
ston	<input checked="" type="checkbox"/>	instance	0.02

It is as simple as that! A class called "bell+" will appear in:



When an object of this class is instantiated (drag the icon of "bell+" onto a patch), the new slots will appear as red inlets after the list of P-fields.

There is however a problem: red inlets are treated by *omChroma* as P-fields and will therefore be passed as values to the score if nothing is done. To avoid it, the parsing function must be told not to return the last 2 fields. "**parse-subcomp-elim**" performs this task: the patch on the left of the window and the patch "subcomponents" are absolutely the same as in the previous tutorial. The patch on the right on the contrary adds to the output list a component without the last two fields.



Since "butlast" needs a list, "comp-list" is used to extract a list of values from a component, eliminate the last 2 fields and build again a component which is to be passed to the output list.

6-5) Instanciation of a new class (TUT 08)

Let's now imagine that a "parsing-fun" generating sub-components for every component is regularly employed. Since control inlets belong only to a factory, each time it is instantiated three control inlets will have to be created and their keywords edited.

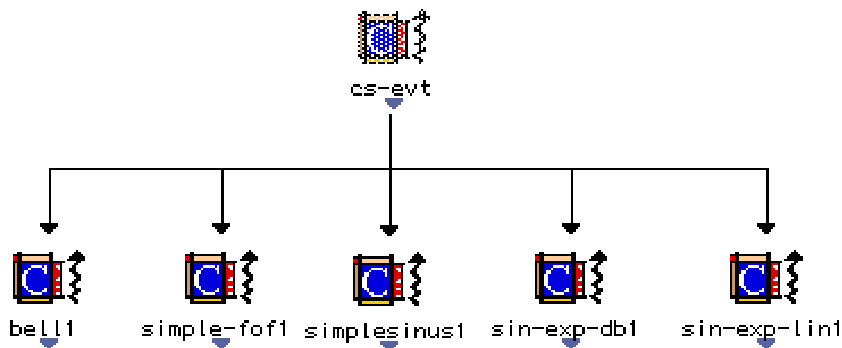
It may be more efficient to prepare a new class that will include the needed control parameters in its slots, so that each new instance of this class will show them as red inlets. Let's call them again "npart" and "ston"¹.

This tutorial has the same structure as the previous one, but uses the new class BELL+ which inherits from "bell1" with two more red control inlets.

Click on the upper half of the package "classes".

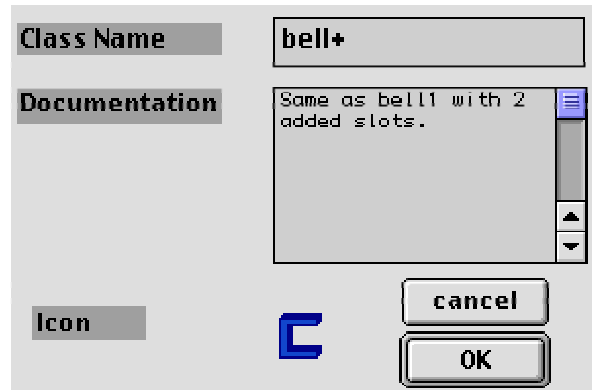


After some editing the class structure will look like:

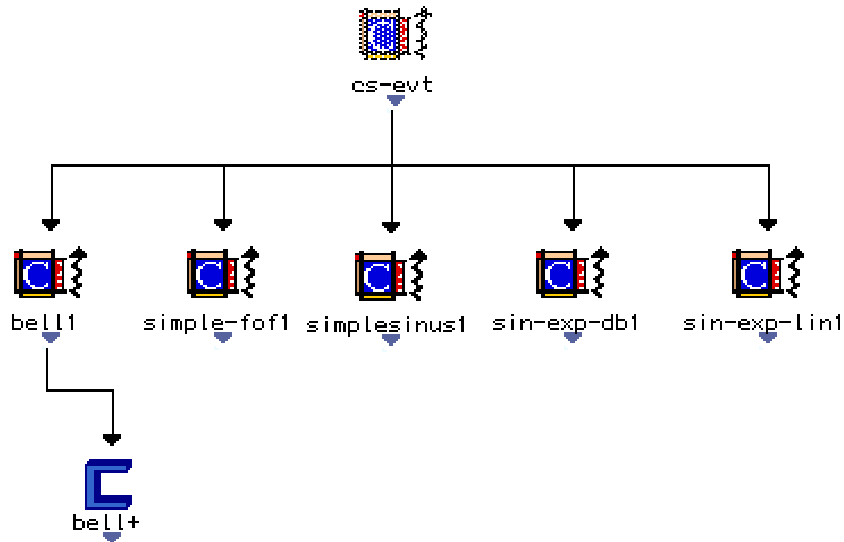


1. "parsing-fun" and "precision" are reserved keywords and cannot be used as slots of a class.

Create a new class (in the menu "File / New Class"), call it, for instance, "**my-bell+**"¹ document it...



... and have it inherit from "bell1" by dragging an arrow from "bell1" to the new class.



Double click on the new class, add "**New Slots**" (from the File menu) and assign a name and a default value to both of them. By definition the type of all the new slots is a real number.

1. A class called "bell+" already exists and is used by the tutorial 08. If a class is redefined, all the connections to the objects using this class will be deleted. The figures of this tutorial correspond to the definition of the class "bell+" when the patch was written, but if you wish to reproduce the process you should use another name.

BELL+			
Slots	Show	Allocation	Default value
npart	<input checked="" type="checkbox"/>	instance	1
ston	<input checked="" type="checkbox"/>	instance	0.02

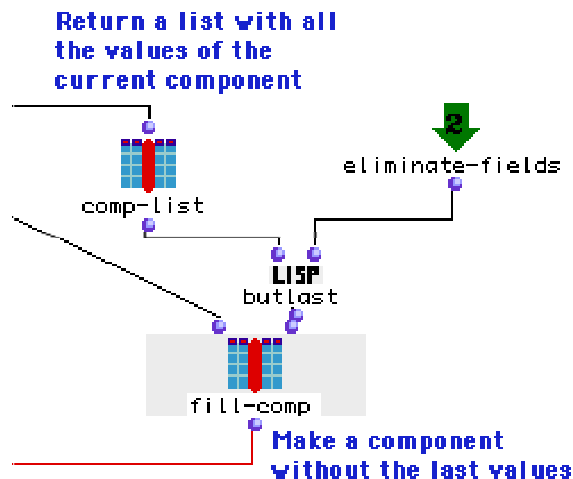
It is as simple as that!

A class called "my-bell+" will appear in:



When an object of this class is instantiated (drag the icon of "my-bell+" onto a patch), the new slots will appear as red inlets after the list of P-fields.

There is however a problem: red inlets are treated by *omChroma* as P-fields and will therefore be passed as values to the score if nothing is done. To avoid it, the parsing function must be told not to return the last 2 fields. "**parse-subcomp-elim**" performs this task: the patch on the left of the window and the patch "subcomponents" are the same as in the previous tutorial. The patch on the right on the contrary adds to the output list a component without the last two fields.



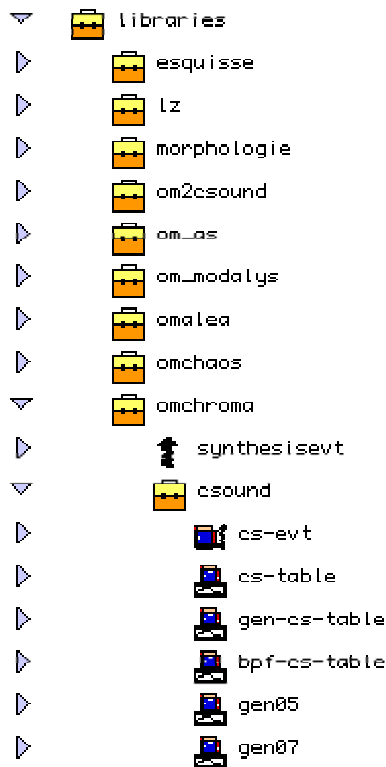
Since "butlast" needs a list, "comp-list" is used to extract a list of values from a component, eliminate the last 2 fields and build again a component which is passed to the output list.

6-6) Tables: all possibilities

There are many ways to specify tables that will be translated into Csound's "f" statements calling GEN functions. Tables are processed only when the matrix's slot is of type "cs-table".

Tables can be **local** (attached to a matrix) or **global** (passed as an argument to "synthesize"). This tutorial does not have any special musical purpose, but summarizes all the possible ways to use tables. It generates a simple non-harmonic additive sound, whose frequencies are computed by the function "arithm-ser".

Connect the different cases shown in the tutorials to the inlet ":aenv" to see how they are processed. The classes for all the tables are in the menu "Classes / Libraries / omChroma / CSound" or in:



6-6-1) Local tables (TUT 09a)

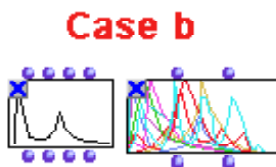
Local tables are passed to a matrix's slot of type "cs-table". When computing the score omChroma will automatically convert the value into a correct "f" statement. If no ID is specified, an ID will be automatically assigned starting at the value specified in the preferences. Tables without ID's¹ are quite practical to use, but not very efficient, since omChroma considers that in the most general case all the tables may be different and will generate a new "f" statement for every component (as was the case in the previous tutorials). To avoid it, the table's ID should be specified, either in the table object or by passing it directly (cases c and e).

Local tables can have the following structure:

Case a: nothing

The slot does not appear or is not connected in the matrix. The default definition of the table specified in the preferences will be used (see the tutorial 06).

Case b: a BPF or a BPF-LIB

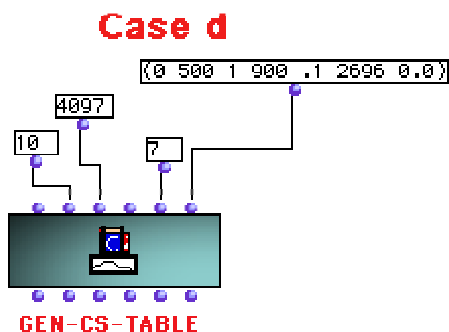


A BPF will be translated onto a GEN 7 (break-point function). All the components will be given the same table. A BPF-LIB will be treated as any list of values: all the tables in the library are retrieved sequentially and they are not enough, the retrieval will start again from the beginning. Notice that in the case of a BPF-LIB either outlet can be connected to the matrix' inlet.

Case c: an integer number

It will be considered as a GEN's absolute ID. The definition of this GEN should be passed as a global table, otherwise an error when running Csound will be produced.

Case d: an instance of the subclass GEN-CS-TABLE



1. Case a, b, d and e below.

This class allows the user to specify in a OM style any Csound's "f" statements.

Arguments:

self: the object itself

id: the table's ID (integer, if it is the string "?" the ID will be automatically computed and incremented for each component)

size: GEN's size (power of 2 or power of 2 + 1)

stime: action time of the GEN

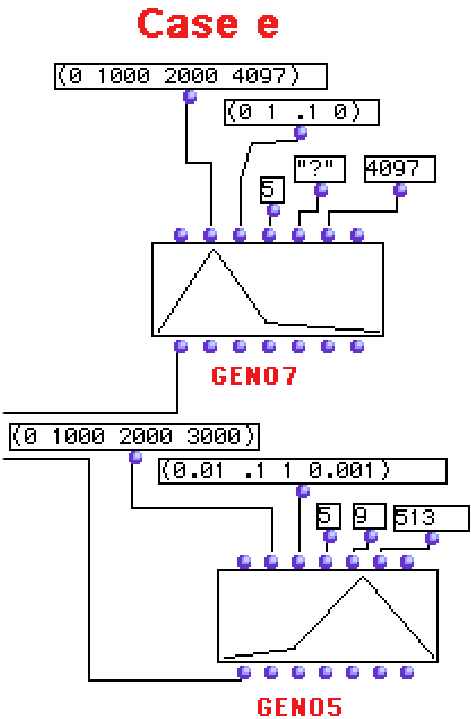
gen-num: GEN number

param-list: list of parameters as needed by the syntax of each GEN

For example, the table shown above will be translated into the following statement:

f 10 0 4097 7 0 500 1 900 0.1 2696 0.0

Case e: an instance of the subclasses GEN05, GEN07.



These classes inherit from the class BPF. They allow the user to specify break-point GEN's using a syntax similar to the BPF and to see the table graphically within the factory.

Arguments:

self: the object itself

x-points: X-points, automatically scaled between 0 and "size" and turned into incremental points as needed by the Csound's syntax

y-points: Y-points, not rescaled

decimals: number of decimals of the points (default = 0)

id: the table's ID (integer, if it is the string "?" the ID will be automatically computed and incremented for each partial)

size: GEN's size (a power of 2 or a power of 2 + 1)

stime: action time of the GEN

For example, the instance of the class GEN07 and GEN05 shown above will result in the following Csound GEN's:

```
f 1001 0 4097 7 0.0 125 1.0 125 0.1 263 0.0
```

```
f 9 0 513 5 0.01 171 0.1 171 1.0 171 0.001
```

Case e: a string or a list of strings.

Strings should contain a GEN definition in the right Csound syntax and will be passed as such to the score.

All these different specifications can be assembled in a list in whichever order.

6-6-2) Global tables (TUT 09b)

Global tables are passed as an argument to "synthesize". Since at this stage of the process no dynamic computation if the GEN's ID is allowed, all the tables should have their own ID.

Global tables can have the following structure:

Case a: list of strings in Csound syntax

It is the structure most frequently used in all the previous tutorials.

Case b: filename (with path)

It can be either **absolute** (`#P"PB marco: CAO:OM 3.6:chroma:cs:audio-funs.lisp"`) or relative (`"cl:chroma:cs:cs-funs.lisp"`), where "cl" corresponds to the root position of MCL. The file should contain a data base of definitions of Csound GEN's in the following format: (`om::ScSt "<csound-GEN>"`).

For example, (`om::ScSt "f2 0 32769 10 1 0.3 0.6 0.2 0.45 0.05 0.1 0.01 0.2 0.22 "`). See the file "cs-funs.lisp" in the folder "chroma" for an example.

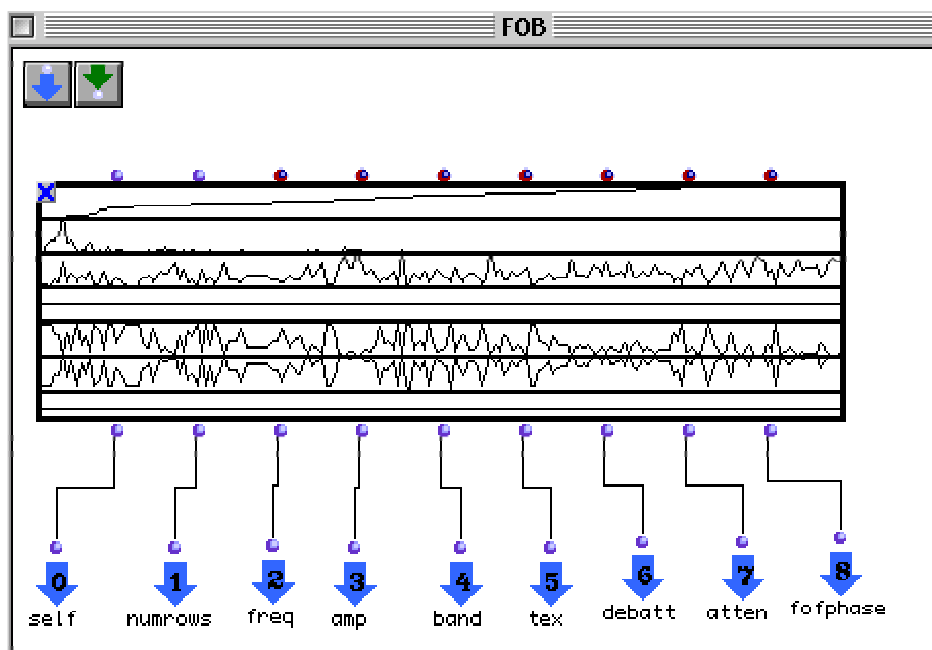
Case c: type GEN-CS-TABLE, GEN05, GEN07

Same structure as above, but the GEN's ID must be specified.

All these various definitions must be assembled into a unique list passed as an argument to "synthesize".

6-7) Percussive FOF in Chant and Csound together (TUT 10)

This tutorial shows how to use the same control data to generate a percussive FOF calling either Chant or Csound. The basic material comes from a simple analysis of the sound of a cymbal using Diphone's ModRes. ModRes produced an SDIF file loaded into the patch "fob" (FOF Bank) by the object 1FOF (see the documentation of the SDIF library).



The left side of the tutorial instantiates a factory of the class CH-FOB-EVT and passes it to "synthesize" calling the chant's patch number 0 (FOF-bank).

The right side of the tutorial instantiates a factory of the Csound class SIMPLE-FOF1 and feeds it with the same data coming from the analysis. The two sounds are very similar.

This tutorial elucidates the features of a virtual synthesizer: the same data are used to run different real synthesis engines or algorithms. In this case the same DSP unit is run using two synthesizers. It would be very easy to use the same analysis data to control other synthesis algorithms (additive synthesis, formantic frequency modulation, filters, etc.) within the same synthesizer.

Some restrictions, however apply, since any given synthesis engine will have some peculiarities that are not be found in others, let alone changes in the implementation or in the efficiency.

For instance, the Csound FOF contains a built-in "octaviation" control that is not directly accessible when using Chant and ought to be implemented in the control layer. Being a matrix, it also directly allows for different entry delays of each component (play with them by typing a value different from 0.0 in the patch "ran ed"). This would be relatively cumbersome to implement in Chant. On the other hand, Chant allows for embedded layers of control (see tutorials 12 and 13) that are quite laborious to realize in Csound.

6-7) Control of Chant: all the available patches (TUT 11)

This tutorial shows all the currently possible ways of using Chant. The musical material is coming from the same analysis data as in the previous tutorial (patch "fob") and from the analysis of a low piano sound using ModRes for the object (patch "reb").

All the possible Chant objects are used, namely a bank of FOF's and filters, a source of noise and an input sound. "Synthesize" is given a list with all the objects (except the sound file). Depending on the chosen patch, only the usable objects will be taken into account, while the others are ignored. For instance, if the patch 4 is selected, only the "reb" and "noise" objects will be used.

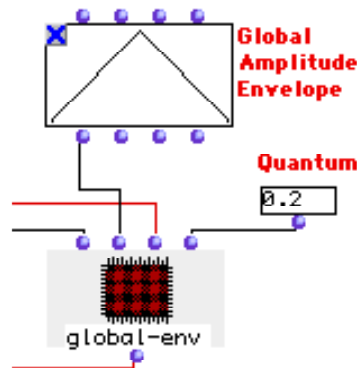
Sound files must be an object of type "sound" directly passed to a control inlet of "synthesize" of keyword **":sound"**.

6-8) High-level control of the amplitude in Chant (TUT 12)

The SDIF Chant synthesizer provides very limited control structures, since it only interpolates values between frames. Higher control paradigms must therefore be implemented in *omChroma*.

This and the next tutorial will show how to implement higher-level controls in Chant. We will try to reproduce some paradigms that were previously available in Formes, a Lisp-based high-level control environment for Chant developed at IRCAM by the Analysis and Synthesis Team in the 80's and no longer available.

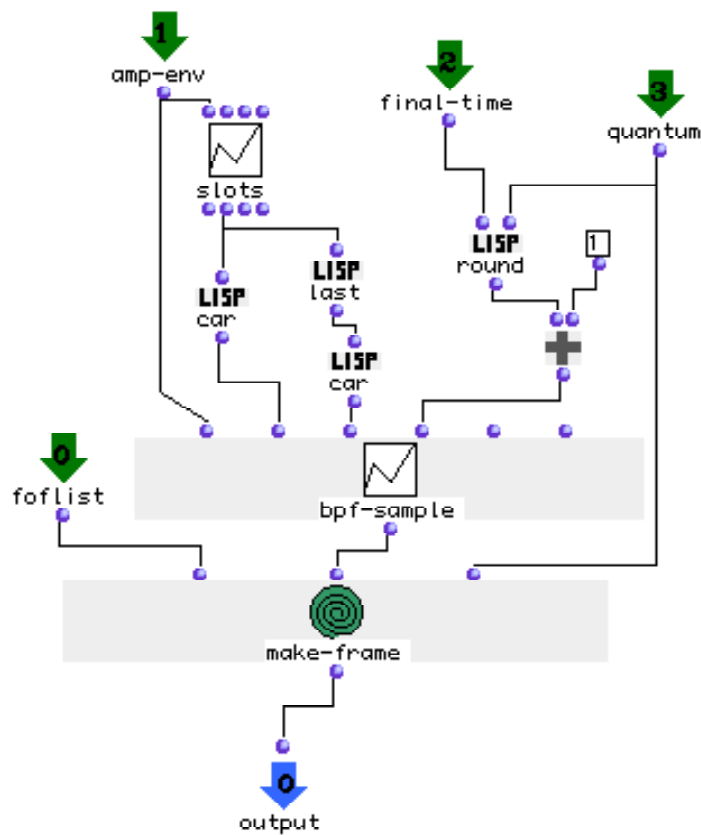
The tutorial 12 is the same as the tutorial 2, but implements a global control of the amplitude envelope, so as to avoid the click at the beginning and at the end of the sound. The patch "global-env" requires as arguments a list of fofs, a BPF, a duration and a quantum.



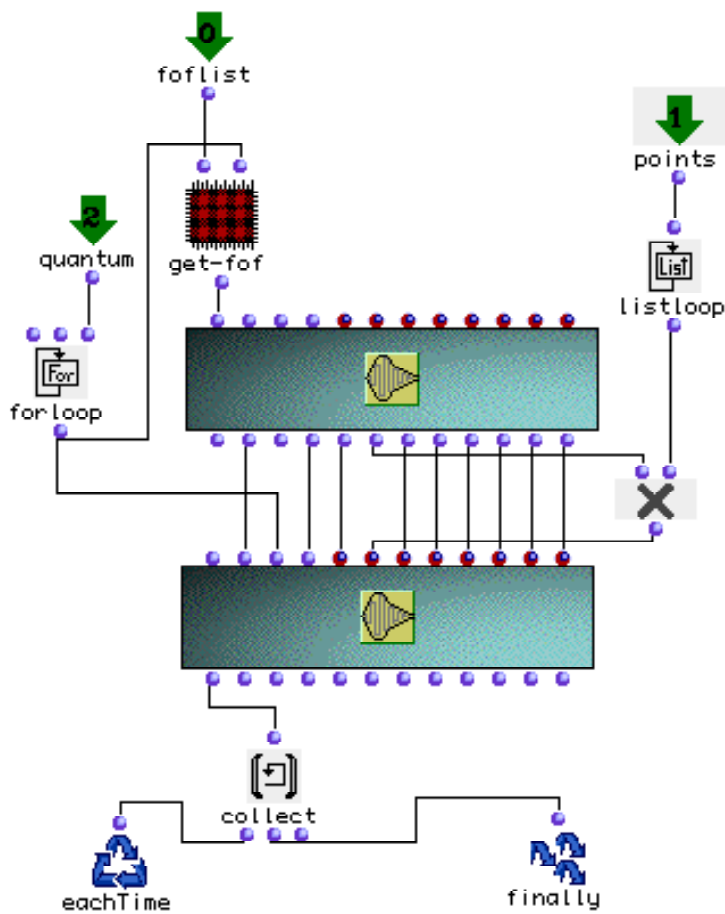
The BPF is the global amplitude envelope and will be sampled at a period of "quantum" (in seconds¹). At each quantum a new SDIF frame of type CH-FOB-EVT will be generated whose amplitudes are scaled by the current value of the BPF.

1. The concept of "quantum" is taken from Formes.

The patch "global-env" samples the amplitude envelope and prepares the data for the main loop, "make-frame".

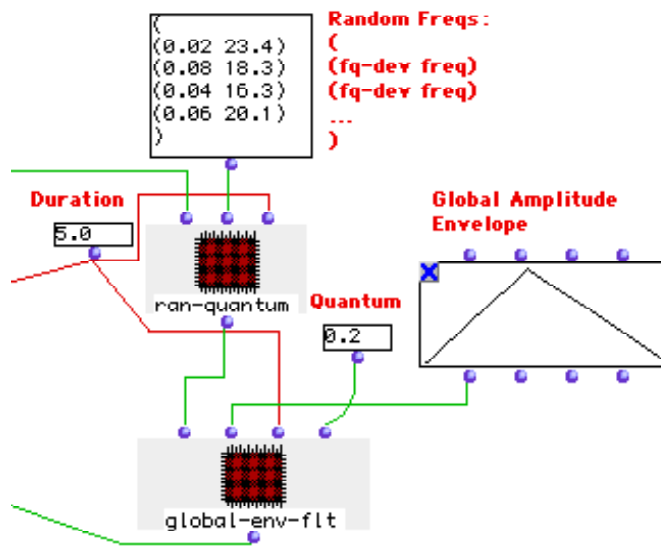


The loop "make-frame" computes a new frame at every quantum modifying its amplitude. The patch "get-fof" returns the currently active FOF in the list of FOF's.



6-9) Two control layers embedded (TUT 13)

This tutorial uses the same filter bank as the tutorial 3 and implements two levels of control: first a random function is applied to the filters' frequencies, then the same global amplitude envelope is applied to the result of this computation.



The main patch is "ran-quantum", whose arguments are:

fitlist: a list of filters (free order).

changeslist: a list of changes in the format ((fq-dev1 jit-fq1) ... (fq-devN jit-fqN)), where "fq-dev" is the frequency deviation (0.1 = 10%) and "jit-fq" is the frequency of the random changes of filter frequency. This list MUST contain as many values as there are frequencies in the filter (this example uses 4 frequencies), otherwise an error will be produced.

finaltime: the final time when the computation should stop (seconds).

The patch returns a (probably) very long list of filters passed as an argument to "global-env-flt" which will apply a global amplitude envelope as described above. The patch "global-env-flt" is the same as "global-env" of the previous tutorial, but it instantiates objects of type CH-FILT-EVT.

"Change-rep" (directly written in Lisp) changes the internal representation of "changeslist". Each pair "(fq-dev-i jit-fq-i)" generates a list with the following structure:

```
( (ti ( (fq-1 fact-1) )  
  (ti+ $\pi$  ( (fq-1 fact-1) )  
  ...  
  (ti+ $\pi$ -end ( (fq-1 fact-1) ) ) ) )
```

where:

"**ti**" = time "i", at the beginning the start-time of the event

"**ti+ π** " = next time (π is the period corresponding the filter's jitter frequency)

"**ti+ π -end**" = last time (π -end approximately corresponds to start-time + duration)

Since some filters might require a change in their frequency at the same absolute time, the most general structure of the list is in fact:

```
( (t1 ( (fq-1 fact-1) ... (fq-i fact-i))  
  (t2 ...)))
```

This structure is passed to the loop "**make-filter**" which will look for the currently active filter (get-filter) and will change its frequency aleatorically.

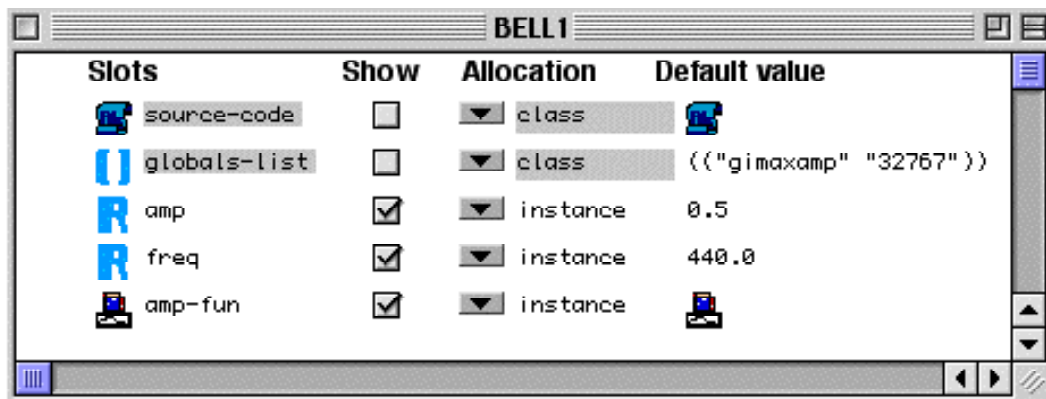
The tutorials 12 and 13 are very useful to have an idea of what embedded control layers are and how they might be implemented. These solutions are however relatively "verbose" and look quite complicated. Future releases of *omChroma* will implement a more general control paradigm directly within a Lisp function.

7 SPECIAL REMARKS / BUGS

This chapter deals with some extended features of *omChroma* that are useful only when the basic functions of the system are well understood.

7-1) Class slots: "source-code" and "globals-list"

The first two slots of a class corresponding to a Csound instrument are grayed and have a special meaning. They are unique for the whole class and are not shown instances.



"source-code" contains a copy of the code of the instrument, i.e. what is contained between "instr" and "endin". It is this code that will be used when generating the orchestra file for Csound. Double-clicking on the icon under "Default value" will show the Csound code of the instrument. This code can be edited, provided that the number of P-fields remains the same. Be aware that opening the window containing the source code is considered as a modification of the class (see below), even if the code is not changed.

"globals-list" contains the list of the global variables used within the instrumen¹. It can be edited.

It is possible to use classes created from instruments scattered in different files. If an orchestra contains several instruments, as many new classes as instruments will be created. Each class is called "<file-name>i" (the name of the file, followed by an increasing integer number). All the instruments are renumbered, so as not to raise conflicts should instruments coming from different files have the same number.

Although the tutorials did not use it, "synthesize" can be given a list of events using different matrices. All the instruments used by the events will be written into a single "orc" file.

1. The header (sr, kr, etc.) will always be ignored, since it is contained in the keywords of "synthesize"

If different instruments use same global variables, but the value of the variables does not change, *omChroma* will keep only one of them. But if a global variable with the same name has a different value, *omChroma* will ask which value is to be used when "synthesize" is evaluate¹d.

7-2) Modification of a class

Any action modifying a class will provoke the destruction of all the connections to instances of the class in all the patches which are using it. A modification is any action operated on the class window, such as changing the name of a slot or a default value, opening the source code or the list of globals, redefining the class, even if the new class is exactly the same as the old one².

For this reason it is recommended to come up with as stable a definition of a class as possible before starting to use it in patches. In case a major change is needed, it is better to create a new class by giving the Csound orchestra file a new name and manually substitute the old instances with the new ones in the patches. In this way the connections will not be automatically destroyed.

7-3) Reserved control keywords: :precision, :parsing-fun

Any name can be used as a control keyword of a matrix except these two reserved words. The meaning of "parsing-fun" has already been amply discussed. "Precision" (integer number, default 6) applies to the format of all the P-fields within a single matrix. It is useful to change it when a precision of 6 digits is not enough.

7-4) Csound abort and bug

If Csound is killed or exits with an error while it is being run by *omChroma*, the computation should be manually aborted in the Listener's window.

Sometimes, the Csound listing gives the warning below. Simply ignore it. The computation will not be affected.

-
1. Since events belonging to different classes can be freely appended in the list of events passed to "synthesize", it is only the latter that can check for the consistency of the global variables. Unless the code of the instrument is changed, "synthesize" will have to interrogate the user each time a conflict is found.
 2. As is the case if the same instrument is loaded again. When trying to redefine a class, *omChroma* will always warn the user, but in all the other cases the modification and consequent destruction of all the connections will be made silently.

tuut09b.sco listing

```

B 4.101 .. 4.130 T 4.130 TT 4.130 M: 1067.8
B 4.130 .. 4.219 T 4.219 TT 4.219 M: 1123.9
WARNING: instr 1 pmax = 6, note pent = 37
new alloc for instr 1:
WARNING: instr 1 pmax = 6, note pent = 37
B 4.219 .. 4.259 T 4.259 TT 4.259 M: 1193.1
B 4.259 .. 4.421 T 4.421 TT 4.421 M: 1520.9
WARNING: instr 1 pmax = 6, note pent = 37
B 4.421 .. 4.451 T 4.451 TT 4.451 M: 1378.2
new alloc for instr 1:
WARNING: instr 1 pmax = 6, note pent = 37
B 4.451 .. 4.512 T 4.512 TT 4.512 M: 1658.7
new alloc for instr 1:
WARNING: instr 1 pmax = 6, note pent = 37
B 4.512 .. 4.874 T 4.874 TT 4.874 M: 2115.6
new alloc for instr 1:
WARNING: instr 1 pmax = 6, note pent = 37
B 4.874 .. 4.885 T 4.885 TT 4.885 M: 1884.9
new alloc for instr 1:
WARNING: instr 1 pmax = 6, note pent = 37
B 4.885 .. 5.452 T 5.452 TT 5.452 M: 2132.8
B 5.452 .. 5.842 T 5.842 TT 5.842 M: 1805.9
B 5.842 .. 7.186 T 7.186 TT 7.186 M: 1701.6
B 7.186 .. 7.650 T 7.650 TT 7.650 M: 1128.7
B 7.650 .. 7.821 T 7.821 TT 7.821 M: 1029.7
B 7.821 .. 8.219 T 8.219 TT 8.219 M: 1000.4
B 8.219 .. 8.381 T 8.381 TT 8.381 M: 909.5

```

7-5) P-fields

If a P-field is missing in the instrument (e.g. there is no p7, but p8 exists), the missing field will nevertheless appear in the slots of the class. "get-instrument" in facts looks for the highest P-field in a given instrument and assumes that all the previous ones will also exist.

8 ALPHABETICAL LIST OF THE AVAILABLE OBJECTS AND METHODS

CHANT-PATCHES

Evaluate it to see a picture with all the available Chant patches

CH-FOB-EVT

Bank of FOF's (see the Chant documentation for the meaning of each argument).

Blue Inlets

self

numrows

start time

fq0

Red Inlets

:frequency

:amplitude

:bandwidth

:saliance

:correction

:**channel1**

CH-FILT-EVT

Bank of filters (see the Chant documentation for the meaning of each argument).

Blue Inlets

self

numrows

start time

Red Inlets

:frequency

:amplitude

:bandwidth

:tex

:debatt

:atten

:fofphase

:channel1

CH-NOISE-EVT

Noise generator (see the Chant documentation for the meaning of each argument).

Blue Inlets

self

numrows

start time

Red Inlets

:distribution

:amplitude

COMP-FIELD

Return or set the value of the slot of a component.

self

lineid

Name of the slot in the class (within quotes)

val (optional)

Set this value in the slot.

COMP-LIST

Return the values of a component as a list.

self

FILL-COMP

Fills a component with the list of "vals". Used often in association with "comp-list".

self

val

GEN05

Instantiate a Csound "f" statement using the GEN 5 (exponential interpolation). Inherits from a BPF (see gen-cs-table for the meaning of the Csound-dependent arguments).

self

x-points

y-points

decimals

size

stime

GEN07

Instantiate a Csound "f" statement using the GEN 7 (linear interpolation). Inherits from a BPF(see gen-cs-table for the meaning of the Csound-dependent arguments).

self
x-points
y-points
decimals
size
stime

GEN-CS-TABLE

Instantiate any Csound "f" statement in the score file.

self
The object itself
id
GEN ID (integer number). If "?", it will be automatically assigned.
size
Table size (power of 2 or power of 2 + 1 if using interpolation)
stime
Action time.
gen-num
GEN number.
param-list
List of parameters needed for the specific type of GEN being used.

GET-COMP

Return the component "compid" in the matrix "matrix".

matrix
compid

GET-INSTRUMENT

Evaluate it to create a new class of a Csound instrument.

NEW-COMP

Creates a new component and fill it with the list of "vals".

self
vals

SYNTHESIZE

Main generic function performing the synthesis and calling a real synthesizer.

Inlets

synthesizer

Choice of the real synthesizer (csound or chant, for the time being) [csound]

elements

List of synthesis events. Events not compatible with the selected synthesizer will be ignored.

Control keywords:

General

name: sound filename [my-synt]

rescale: rescaling factor [1.0], the message "Sound Converted to AIFF" appears on the Listener.

sr: sampling rate [44100]

run: if nil do not run the synthesizer, but generate the score and orchestra files (very useful for debugging a OM patch) [t]

Csound only

tables: list of strings (filename with an absolute or relative pathname or a Csound "f" statement) [nil]

kr: control rate [4410]

nchnls: number of channels [1]

Chant only

duration: duration of the synthesis [nil -> until the end computed in the object itself]

sound: input sound file for chant [nil]

patch: chant patch number [0]

The keywords reserved to a given synthesizer will be ignored if the synthesizer is not active.

9 PERSPECTIVES

There are numerous ways to improve and enrich the examples of the tutorials. Among the¹m: the generalization of some control paradigms into a Lisp function, the utilization of a "maquette" to schedule a sequence of events and the implementation of more sophisticated algorithms.

Although *omChroma* was originally designed for the control of software synthesis through a data file, the interpretation of the matrix data is actually concentrated into the method "synthesize". As a consequence, instead of writing a data file to be passed to a synthesis engine, "synthesize" might be asked to automatically dispatch to a function that will talk in real time through a communication channel with such engines as MAX/MSP, jMax or SuperCollider. The interpretation of the list of matrices therefore always depends on the target synthesizer.

Note that the choice of this interpretation of the matrix structure is arbitrary, and is discussed here because it is the most practical when thinking about a Csound-like style of control. When applied another engines, other interpretations will eventually be necessary. More generally, such a matrix can control any engine whose parameters evolve in a sequence, no matter if the sequence is time or anything else².

1. Some improvements will be available in future releases of *omChroma*.

2. For example, one can imagine an object of type matrix controlling different points in space of a graphic animation.

10 APPENDIX 1

Survey of the development of Chroma

The development of Chroma started in 1980 at the CSC of the University of Padua. Its development went through four different phases marked by the production of a piece or a cycle of pieces.

As of today, Chroma consists of three interconnected systems (see the picture below):

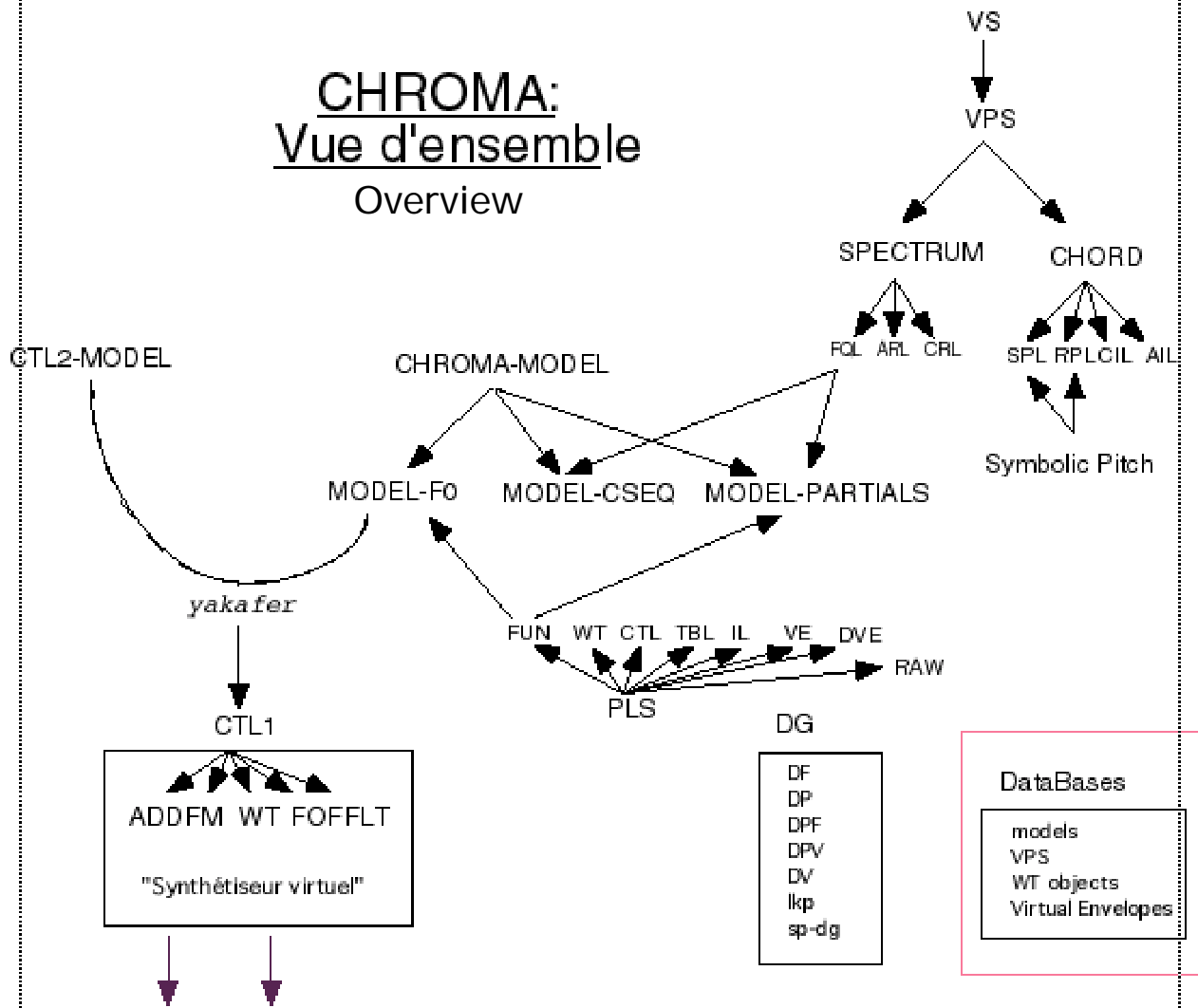
CTL1, PLS, DG: first level of control using abstract typing and implementing the main concepts discussed in this text (virtual synthesizer, control matrix, data bases, etc.). It is this level that was generalized and ported onto OM.

CTL2-MODEL, CHROMA-MODEL: modeling and processing of analysis data (coming from Audiosculpt and Diphone).

VS: system for Computer-Aided Composition, dealing with a unified representation of chords and spectra and used for both instrumental and electronic pieces.

CHROMA: Vue d'ensemble

Overview



The table below shows the different phases of the development of Chroma, as well as the works that were composed and the assistants who collaborated in its development.

YEAR	PLACE	SYSTEM	FEATURES	LANGUAGE
1980-82	CSC, University of Padua	<ul style="list-style-type: none"> • control of additive and formantic FM synthesis • rule-based through table-lookup 	<ul style="list-style-type: none"> • conceptual model of sound event • simple orchestra (ADD, FM) • automatic sort • unlimited dynamic allocation of instruments 	<ul style="list-style-type: none"> • MUSIC V's PLFs and conversion routines • Fortran • JCL scripts for an IBM computer
PIECE: Traiettoria (1982-84) for piano and computer-generated tape				
1984-85	MIT, USA	<ul style="list-style-type: none"> • PLS: musically pertinent abstract types • theoretical basis of a virtual synthesizer 	<ul style="list-style-type: none"> • orchestra rewritten in Csound • l-time perceptual compensation of some parameters • richer base of instruments (added a sampler) • sketches of a virtual synthesizer 	<ul style="list-style-type: none"> • Csound • LeLisp • Unix scripts • various tests in c (score language)
1989-90	IRCAM	<ul style="list-style-type: none"> • Chroma: full virtual synthesizer (for Csound, the Moon Array Processor and a Yamaha TX816) • added CTL1 and Data Generation • added a simple constraint-based system (ELET) • sketched the representation of polymorphic VPS's (unified representation for spectra and chords) • "sound concepts" implemented • data bases of amplitude envelopes and VPS's 	<ul style="list-style-type: none"> • full-grown environment from CAO to synthesis control • simultaneous processing (multiple Csounds on different Unix machines) • parallel constraint-based system in Prolog for dealing with VPS (developed with Francis Courtot) and communication with synthesis control through exchange of data files 	<ul style="list-style-type: none"> • CTL1 developed by Ramon Gonzalez-Arroyo in LeLisp • Chroma used for the piece Feuillages by Philippe Schoeller
PIECES: Leggere il Decamerone (1989-90) computer music for the integral reading of Boccaccio's Decameron Proemio (1990) , radio opera for 3 actors and computer élet...fogytiglan (1989) for ensemble -> usage of Prolog & ELET				

1992	IRCAM	<ul style="list-style-type: none"> • specification of the features of a musical freeze algorithm • sketch of CTL2 (high-level synthesis control) 		<ul style="list-style-type: none"> • musical freeze algorithm developed by Jan Vandenheede in LeLisp and inserted into CTL1 code
PIECES: in cielo in terra in mare (1992) radio opera for 8 actors, vocal ensemble and computer				
11-12 1995	IRCAM		<ul style="list-style-type: none"> • translation of the whole system into MCL running on a Macintosh • compatibility MAC/Unix • complete tests and local improvements • interface MCL--> Csound through Apple events 	<ul style="list-style-type: none"> • translation and development by Serge Lemouton
4-6 1996	IRCAM	<ul style="list-style-type: none"> • 1st phase of Chroma on the Macintosh completed • interface with analysis data: CTL2, Analysis-models • VPS system redesigned and extended in CLOS • large data bases of models 	<ul style="list-style-type: none"> • fully operational • produces scores for Csound running locally (Mac) or remotely (Unix) 	<ul style="list-style-type: none"> • Musical assistant: Serge Lemouton
PIECE: Come Natura di Foglia, Canti lontani per voci ed elettronica Canto primo for vocal quartet and electronics.				
3-4 1999	IRCAM	<ul style="list-style-type: none"> • extension of the CTL2 level to analysis data coming from Diphone • added algorithms for set theory to VPS 	<ul style="list-style-type: none"> • concentration on the Macintosh. Unix side disconnected. 	<ul style="list-style-type: none"> • Musical assistant: Serge Lemouton
PIECE: From Needle's Eye for trombone and ensemble (CAO and set theory)				
1999- 2000	IRCAM	<ul style="list-style-type: none"> • development of omChroma in OM 		

How to make your own version of csound compatible with omChroma

For the time being, *omChroma* works only with the special version of Csound found in the folder of the library (csound 4.05). We hope that newer versions of Csound for the Macintosh will include the changes needed to make it possible to be called by OM. However, if another version of Csound is needed, you have to compile your own project using Code Warrior.

Here is what is required:

"perf" has to be sent a command line through an Apple Event from OpenMusic and not from the front end "csound".

When "perf" ends its computation, it systematically returns a message to "csound" using its signature. This mechanism has to be modified so that it returns a message to OM (signature "CCL2"). The argument "@" in the command line has to be added, so that when it is present "perf" returns a message to OM and not to "csound".

This modification will not change the normal behavior of Csound.

If this modification is not made and another version of Csound is used, "perf" will be correctly started by OM, but OM will not receive the "end-of-perf" message. The Lisp computation will then have to be manually aborted (the same as if an error had been produced).

One way this modification can be hacked up might be:

Define a variable "OSType frontendsign = 'VRmi';" in "**perf_menu.c**" and replace all the occurrences of "VRmi" (in "**perf_AEHandlers.c**" and "**perf.c**") with this variable.

Finally introduce the code below in the function "argdecode" (in "**argdecode.c**").

```
case '@':
```

```
frontendsign = 'CCL2';      /* no func displays */
```

```
break;
```

Compile Csound and call it **Csoundom** and **PERFom**.