Music and code

Andrea Vigani

October 14, 2025

Preface

This writing is not:

- intended to be a book or a manual.
- exhaustive.
- written in a descriptive/discursive style.
- aimed to provide absolute truths or certainties.

It aims to:

- provide readers with different aspects of electroacoustic music from different perspectives.
- prompt readers to think individually or collectively about specific topics.
- provide readers with input that will encourage them to delve deeper into the topics they find most interesting among those presented.
- help readers build their own poetic consciousness and learn the most suitable informatic tools to achieve it.
- help readers to find their own compositional models and procedures.

With the same purposes I use it as a framework in my University and Conservatory lessons.

It is always *under construction*.

Check first page to see the last date version.

Contents

1 Premises **11**
1.1 Preliminary reflections . . . 11
1.1.1 Concrete things . . . 11
1.1.2 Pre-linguistic . . . 12
Natural language . . . 13
Musical language . . . 13
1.1.3 Compose a phone number . . . 14
Phonetic-phonological level . . . 14
Morphosyntactic level . . . 15
Syntactic level . . . 15
Semantic level . . . 17
1.1.4 Perceive and process . . . 18
1.1.5 Re-compose a puzzle . . . 19
1.2 Music's meanings . . . 19
1.2.1 Signs and sounds . . . 20
1.2.2 Composers, players and listeners . . . 21
1.2.3 Do you know this music? . . . 22
1.2.4 Do you like this music? . . . 23
1.2.5 Texts and narratives . . . 24
1.3 Forms of human expression . . . 25
1.3.1 Poietic dimension . . . 25
1.3.2 Aesthesic dimension . . . 26
1.3.3 The work itself . . . 27
1.3.4 Final reflections . . . 28

2 Genres, instruments and tools **31**
2.1 Genres . . . 31
2.2 Non interactive systems . . . 31
2.2.1 Fixed media music . . . 32
Acousmatic music . . . 32
Wallpaper music, ambient music and soundscapes . . . 34
Radiodrama . . . 35
Computer music - algorithmic composition . . . 36
2.2.2 Live sequencing . . . 37
Sound design for performing arts or video . . . 37
Sound design for gaming . . . 38
2.3 Interactive systems . . . 38
2.3.1 Live set . . . 39
2.3.2 Hyper-instrument . . . 40
2.3.3 Live coding . . . 40

2.4 The virtual instrument paradigm . . . 41
2.4.1 Software installation . . . 41
2.4.2 Instrumental model . . . 41
2.5 Composition sketches proposal . . . 43

3 Fixed media **45**
3.1 Framework and historical context . . . 45
3.2 Elements and strategies . . . 45
3.2.1 Sound objects . . . 46
3.2.2 Objects, structure and form . . . 46
3.2.3 Musical objects . . . 47
3.2.4 Time courses and musical forms . . . 49
3.2.5 The poietic dimension: determinism . . . 50
3.3 Sound processing techniques in SuperCollider . . . 52
3.3.1 Buffers . . . 52
3.3.2 Sampling . . . 53
Sequencing . . . 54
Enveloping . . . 55
3.3.3 Scratching . . . 56
3.3.4 Structures, DAW and scoring . . . 58
3.3.5 Composition sketches proposal . . . 60

4 Computer music **61**
4.1 Framework and historical context . . . 61
4.2 Randomness . . . 62
4.2.1 Chance . . . 63
4.2.2 Indeterminacy . . . 64
4.2.3 Alea . . . 66
4.3 Sound synthesis . . . 68
4.3.1 Additive . . . 69
Wavetables . . . 69
Vectorial . . . 70
PWM . . . 71
Classic . . . 73
4.3.2 Subtractive . . . 75
4.3.3 FM . . . 78
4.3.4 Granular . . . 80
4.3.5 FFT . . . 84
Spectral Delay . . . 85
Brick Wall filter . . . 85
Rect Comb filter . . . 86
Mag Smear . . . 86
Freeze . . . 87
Cross synthesys. . . . 87
4.4 Algorithmic structures . . . 88
4.5 Composition sketches proposal . . . 90

5 Algorithmic composition **91**
5.1 Framework and historical context . . . 91
5.2 Software installation . . . 92
5.3 Symbolic musical representation and numbers . . . 93
5.3.1 Pitch . . . 93

5.3.2 Rhythm . . . 94
5.3.3 Dynamic . . . 95
5.3.4 Collections . . . 96
5.3.5 Live . . . 97
5.4 Generative techniques and processes . . . 97
5.4.1 Modes and scales . . . 97
Scale . . . 98
Incises and motives . . . 98
Variations . . . 101
5.4.2 Harmonic fields . . . 105
Chord database . . . 106
Voicing . . . 107
Voice-leading . . . 108
Melodies . . . 110
5.5 Composition sketches proposal . . . 112

6 From live set to laptop ensemble 113
6.1 Historical framework . . . 113
6.1.1 Experimental fields . . . 113
6.1.2 Mainstream fields . . . 116
6.2 Interaction types . . . 117
6.2.1 Trigger . . . 117
6.2.2 Switch . . . 118
6.2.3 Continuos . . . 118
6.2.4 Conversions . . . 119
Smoothing . . . 119
Sample and hold . . . 119
Thresholds . . . 120
6.3 Control signals . . . 121
6.3.1 Types . . . 122
6.3.2 Scaling . . . 122
6.3.3 Plugging . . . 122
6.4 MIDI . . . 123
6.4.1 Receive . . . 124
6.4.2 Send . . . 125
6.4.3 Voice allocation . . . 125
Monophonic synth . . . 125
Poliphonic synth without sustain envelope . . . 126
Poliphonic synth with sustain envelope . . . 127
6.5 OSC . . . 127
6.5.1 Receive . . . 128
6.5.2 Send . . . 128
6.6 HID . . . 129
6.6.1 Mouse . . . 129
Client side . . . 129
Server side . . . 133
6.6.2 Keyboard . . . 133
Client side . . . 133
Server side . . . 136
6.7 Live playback and synthesis . . . 136
6.7.1 One shot . . . 136

6.7.2 Mapping . . . 137
Key → soundfile . . . 138
Key → playback rate (transposition) . . . 138
Key → start position . . . 139
6.7.3 Cues . . . 140
6.8 Composition sketches proposal . . . 141

7 Mixed music and hyper-instruments 143
7.1 Framework and historical context . . . 143
7.1.1 Sound and formal augmentation . . . 143
7.1.2 Augmentation of acoustic instruments . . . 144
7.1.3 Common topics to think about . . . 145
7.2 Software model . . . 145
7.2.1 Routing signals . . . 148
In/Out . . . 148
Public vs private buses . . . 149
Creation order rules and Groups . . . 150
7.3 Feature extraction as control signals . . . 150
7.3.1 Envelope follower . . . 151
7.3.2 Pitch follower . . . 153
7.3.3 Onset detection . . . 155
7.3.4 Spectral features . . . 156
7.4 Classic live electronics sound processing . . . 157
7.4.1 Ring modulation and frequency shifting . . . 159
7.4.2 Live recording and playback . . . 161
Asynchronous . . . 162
Loop station . . . 163
7.4.3 Granular synthesis . . . 165
Asynchronous . . . 165
Synchronous (harmonizer) . . . 167
7.4.4 Delay lines . . . 169
Echo . . . 169
Feedback gain . . . 170
Feedback duration . . . 171
Ping pong . . . 171
Multi Tap . . . 172
Variable time . . . 173
Filtred . . . 173
Resonator . . . 174
7.4.5 Multichannel panning . . . 175
Mono, dual-mono and multi-mono panning . . . 175
Stereo . . . 176
Quadriphony . . . 177
Circular systems . . . 179
7.5 Composition sketches proposal . . . 180

Appendices 181

A SuperCollider basics 183
A.1 User interface . . . 183
A.2 Architecture . . . 184
A.3 Main commands . . . 184

A.3.1 New file . . . 184
A.3.2 Open file . . . 184
A.3.3 Lines of code . . . 185
A.3.4 Evaluate the code . . . 185
A.3.5 Execution order . . . 185
A.3.6 Error messages . . . 186
A.3.7 Clear Post window . . . 186
A.3.8 Comments . . . 186
A.3.9 Save file . . . 186
A.3.10 Data types . . . 186
A.4 Audio . . . 187
A.4.1 Server boot . . . 187
A.4.2 Stopping ongoing processes including audio . . . 187
A.4.3 Boot info . . . 188
A.4.4 Global variables, classes and messages . . . 188
Variable . . . 188
Class . . . 189
Message . . . 189
A.4.5 Server settings . . . 189
A.5 UGens . . . 190
A.6 Monitors . . . 193
A.6.1 Computational activity . . . 194
A.6.2 Audio and control signals . . . 194
Server inputs and outputs . . . 194
UGens inputs and outputs . . . 195
A.7 Help files . . . 195
A.8 Web resources . . . 195

B Components, connections and parameters 197
B.1 Source . . . 197
B.1.1 Oscillators . . . 197
Lookup . . . 197
Interpolating wavetable oscillator . . . 198
Sample playback oscillator . . . 200
Buffer reading oscillator . . . 201
B.1.2 Input signals . . . 201
B.2 Ampltude envelope . . . 202
B.2.1 Representations and units of measurement . . . 203
B.2.2 Scaling . . . 205
B.2.3 Interpolations and ramps . . . 205
B.2.4 BPF and control signals . . . 207
Env and EnvGen class . . . 207
B.2.5 Triggers, envelope types and durations . . . 210
Without sustain . . . 210
With sustain . . . 212
B.3 Pan . . . 213
B.4 Output . . . 216

Chapter 1

Premises

1.1 Preliminary reflections

Music composition

What does it mean?

To compose, to assemble, to put together something.

Let's tackle some basic concepts about in no particular order.

1.1.1 Concrete things

Before producing any sound we must choose or assemble a physical object.

When the object (sound source) was choosen or assembled:

- we can interact with it (touch it, move it, etc.).
- we can make a copy (a model).
- we can modify it (improve it).
- we can destroy (or forgot) it.
- it does not require linguistic representation, it simply be.
- it exist in space and not necessary in time.

This has to do with the physical world.

Pyhsical object with elestic characteristics can be exited in some way producing sound waves.

Sound waves are a form of energy that propagates in time and space through an elastic medium.

Sound waves simply exist in our world.

When a sound wave reaches our ears it is perceived as an acoustic information by our brain.

It's a concrete thing.

We live immersed in sound waves.

We listen to music more or less daily but... is music always sound waves?

No.

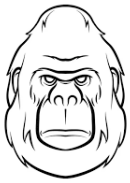
We can think and organize sounds inside our mind without producing them as sound waves.

Sound waves have to do with physical world and human perception.

Music is a complex thing that has to do with both perceptual and cognitive human systems.

1.1.2 Pre-linguistic

We can compose ideas or instinctive sequences of sounds putting thoughts together in a pre-linguistic manner.



An abstract thing.

When we thinked it:

- we can't interact with it (touch it, move it, etc.).
- we can do a copy (replicate pattern structure).
- we can reproduce it but only in its physical forms (kinds of matter):
  - declaiming it (sound waves).
  - writing it on a paper using symbolic representations.
- we can modify it.
- we can't destroy it (we can only destroy its representation in physical world - book, recordings, etc. - not in our memory).
- it does not require linguistic representation.
- it exists in the space of consciousness (our mind) but when we reproduce it in the physical world it exist both in space and time.

This has to do with mind, consciousness and human expression.

Let's explore this concept through a comparison between natural language and musical language.

They:

- are universal (present in all human cultures).
- belong exclusively to our species.
- ensure the cohesion of a social group.
- share formal and functional characteristics.

Natural language

Psycholinguistic studies say that at a deep level all natural languages have the same structure and this can tell us something universal about the human intellect (our brain).

The form of human thought is innate and common to all members of the species.

The function of language is to express it.

The deep structure of an expression is closely related to the thought it represents.

If pre-linguistic thoughts have the same type of form for everyone the linguistic structures that represent them must also have the same type of form.

If we want to transform a deep thought structure into a statement we have to assemble it together into a linear sequence of sounds.

This sequence must convey to the listener the necessary informations for its comprehension.

Thought itself cannot be considered a linguistic sequence: it exists independently of language.

Musical language

About music there exist forms of mental activity independent of any musical language such as children's babbling [1] or tibetan songs [2].

Pre-linguistic musical thought must be an abstract scheme that includes exclusively the characteristics common to all musics.

If both music language and natural language are universal characteristics of the human specie this means that humans have a natural capacity to acquire both linguistic and musical skills.

Natural language and musical language expressions are conveyed in the real world in the form of sound waves.

The sound waves common basic physical parameters are:

- frequency → interval, pitch, intonation.
- amplitude → dynamic.

- time → rhythm.
- timbre → instrumental techniques, vocal expressions, orchestration.

Even though surface forms differ from culture to culture we can find these universal elements not only in physical parameters but also in the subdivision into multiple levels of the languages.

1.1.3 Compose a phone number

Defining a code.

- a sequence of numbers (or letters or sounds) that represents a thought.
- we can't modify it.
- it requires encoding and decoding.
- to understand it we need to know its linguistic structure.
- it follow precise and shared rules.
- if we don't know them, the sequence of numbers or sounds makes no sense to us.
- it require linguistic representation.
- it can be:
  - in space → if you write it in a phone directory.
  - in time → if you digit it on a phone keyboard.

This has to do with language and its representations.

Let's start again with the comparison between natural and musical language.

Leaving aside the first common level represented by the physical parameters of sound both are built on further different levels.

Simplifying, we can identify four levels:

- phonetic-phonological → phonetics and prosody - notes, pitches, scales, tunings, etc.
- morphosyntactic → combination of phonemes into morphemes and morphemes into words - rhythm, chords, etc.
- syntactic → rules defining the relationships between words - counterpoint, harmony, twelve-tone system, asides, phrases, etc.
- semantic → access to meaning.

Phonetic-phonological level

It is present in all languages.

Every linguistic output can be broken down into phonemes (a small set of sound classes).

For example the word *ape* is made up of three phonemes one for each letter.

Phonemes have two distinctive features:

- they are not characterized by absolute values but by a set of sounds within a certain range.
- the continuum of sounds is divided differently in different languages (two sounds that constitute two phonemes in one language form a single phoneme in another).

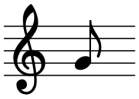
THE INTERNATIONAL PHONETIC ALPHABET (revised to 2018)

CONSONANTS (PULMONIC) © 2018 IPA

	Bilabial	Labiodental	Dental	Alveolar	Postalveolar	Retroflex	Palatal	Velar	Uvular	Pharyngeal	Glottal
Plosive	p b			t d		ʈ ɖ	c ɟ	k g	q ɢ		ʔ
Nasal	m	ɱ		n		ɳ	ɲ	ŋ	ɴ		
Trill	ʙ			r					ʀ		
Tap or Flap		ⱱ		ɾ		ɽ					
Fricative	ɸ β	f v	θ ð	s z	ʃ ʒ	ʂ ʐ	ç ʝ	x ɣ	χ ʁ	ħ ʕ	h ɦ
Lateral fricative				ɬ ɮ							
Approximant		ʋ		ɹ		ɻ	j	ɰ			
Lateral approximant				l		ɭ	ʎ	ʟ			

Symbols to the right in a cell are voiced, to the left are voiceless. Shaded areas denote articulations judged impossible.

In western musical languages the fundamental phoneme could be considered the note with its various types of expression (staccato, tenuto, accentato, legato, etc.).



In both languages they are characterised by frequency, duration, amplitude and timbre.

Morphosyntactic level

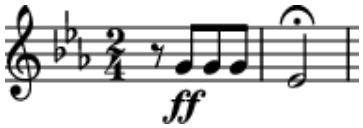
Phonemes can join together to form morphemes.

Morphemes can join together to form words.

Words are basic elements of language that:

- carries meaning.
- can be used on its own.
- are uninterruptible.

In music we could consider them as a short melodic pattern (inciso).



A vocabulary (lexicon) is a set of words in a given language.

In music we could consider it as a set of rhytmic and melodic patterns.

Substantial differences between the two languages:

- in natural language a word has a general function (the word *table* represent a table).
- in music a specific word exist only within a piece (we can consider Beethoven's fifth symphony incise a word only in this symphony).

Syntactic level

Syntax or formal grammar.

a closed system of rules that serves exclusively to generate the set of sequences considered grammatical.

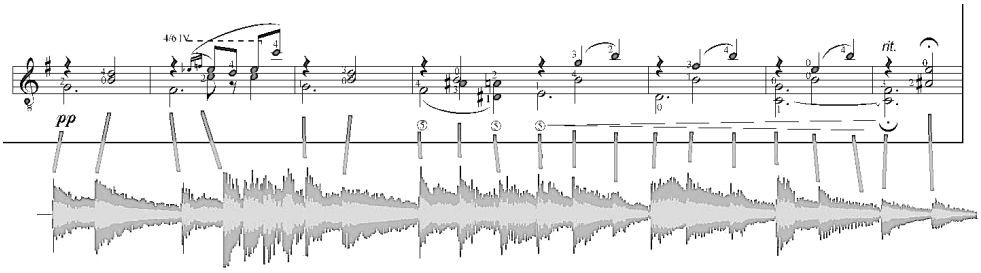
Every natural language like every musical language has its own syntax (italian, french, german, counterpoint, harmony, twelve-tone, etc.).

We have previously defined that music is not necessarily conveyed through sound waves but. . . are sound waves always music language?

No, again.

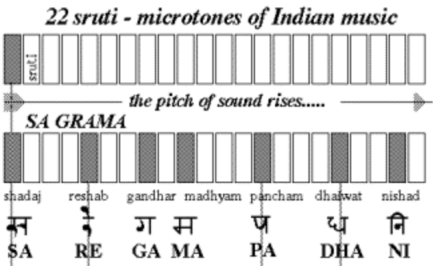
In order to be defined, music language requires:

- a codification of sounds in vocabularies.
- a systems of rules.



In all musical cultures the physical parameters of sound are organized in symbols.

These symbols can become independent from sound waves.



X	2	3	4	5
1 2 3	4 5 6 7	8 9 10 11	12 13	14 15

In Western musical tradition, from psamody until around 1940 these parameters have maintained more or less the same vocabulary over the centuries (diatonic and chromatic intervals, metric and subdivision of beat, musical instruments, etc.).

	PIU'CHE DIMINUITO	DIMINUITO	MINORE	GIUSTO	MAGGIORE	ECCEDENTE	PIU' CHE ECCEDENTE
PRIMA			MAI	(unisono)	MAI		
SECONDA				MAI			
TERZA				MAI			

The systems of rules through which the vocabulary terms were organized (syntax), however, were different (modality, counterpoint, harmonic systems, dodecaphony, alea, etc.).

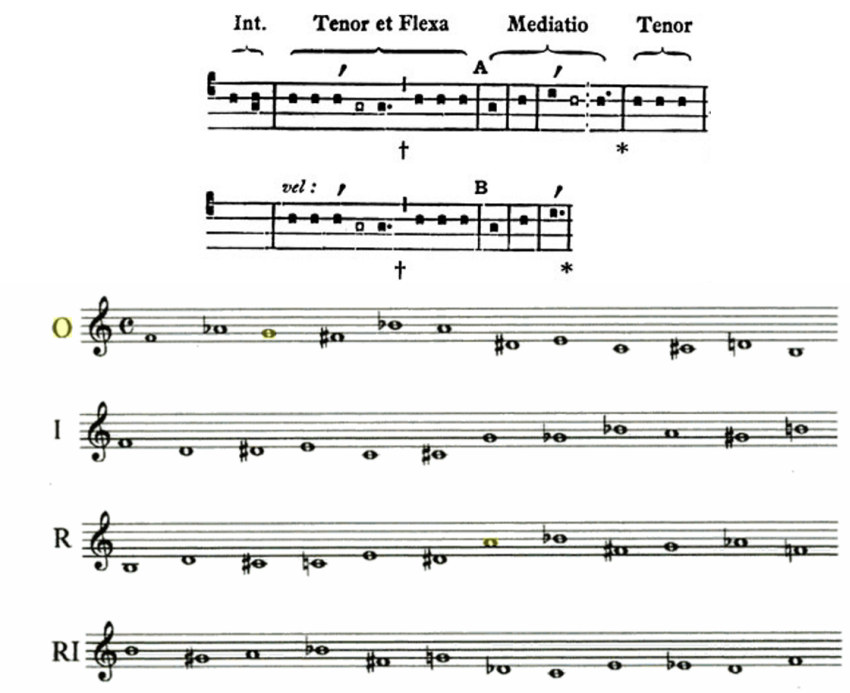


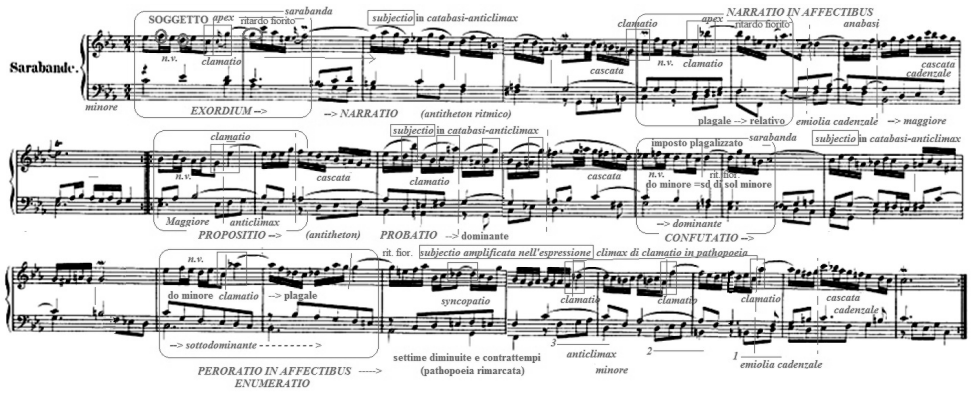
Fig. 1 – Serie della Sinfonia op. 21 di A. Webern

For this reason we cannot define western music as a single language but as a set of languages that use the same vocabulary (classical music, baroque music, pop music, jazz music, etc.).

Even more so if we talk about music from different cultures.

On this level the differences underlined at the end of the previous paragraph regarding musical words disappear because musical grammars are a meta-reflection that deals with symbolic forms.

These can be abstracted to the point of taking on a meaning of their own that goes beyond the perception of the work.



Semantic level

The branch of linguistics and logic concerned with meaning.

In musical language there are several starkly contrasting theories on this subject.

None can be assumed to be universal.

The one that summarizes the two main positions was proposed by the American musicologist L.B.Meyer who distinguishes between two forms of meaning in music:

- designative meaning → refers to something extramusical (G.Malher [3] - X Symphony).
- embodied meaning → the meaning a piece has for the listener in terms of:
  - internal syntactic structure.
  - interactions between this structure and the listener's musical knowledge and expectations. Musical structure can create expectations that can be disappointed or fulfilled generating a dynamic flow of tensions and resolutions that influence the listener's emotional and aesthetic responses. These are aesthetic emotions that have nothing to do with the emotions experienced in real life (F.Schubert [4] - Trio).

Can we express in words what emotions listening to this Schubert trio arouses in us?

We will delve deeper into this last topic in the Music's meaning section (1.2).

These levels enable common phenomena that help us understand the difference between sound waves and language.

- categorical perception → a continuous linguistic or musical sound is segmented into discrete categorized units (phonemes, notes, words). If we repeat the same phrase or melody several times they will always be recognized as the same despite variations in acoustic parameters sometimes even significant ones (exemple 1, exemple 2, exemple 3).
- phonemic restoration → replacing part of the linguistic signal with noise simultaneously with the word or music does not affect the perception of the whole. Semantic-lexical or musical expectations prevail over acoustic analysis [5][6] (musical, natural).
- expectations → the syntactic construction of a sentence leads us to expect one word rather than another within it, just as happens within a sequence of chords in tonal terms [7][8] (words, chords).

1.1.4 Perceive and process

The ability to perceive and process musical information is also common to all cultures.

Listening to music from one's own culture provides the listener with additional information.

The simplest musical competence requires the activation of numerous cognitive abilities:

- memory → short-term recognition of melodic rhythmic patterns.
- maintenance of attention → active listening.
- analysis of the temporal structure → continuous comparison between present and short-term memory.

These abilities are part of a body of knowledge acquired through more or less conscious implicit learning that occurs through two modalities:

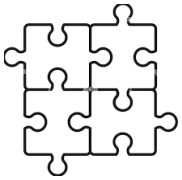
- universal cognitive mechanisms → experimental psychology studies have shown that even prenatally the fetus responds to sounds and noises that activate phonological priming processes. For example, infants have been found to prefer sounds or stories frequently reproduced by their mothers after the third month of pregnancy over those they have never heard. Even in adulthood, some research has highlighted the existence of perceptual mechanisms independent of cultural context.
- environmental and cultural interaction → constant, quantitatively preponderant, continuous and more or less conscious perception of sounds organized according to the reference syntax of the cultural environment of reference.

An example of the dichotomy just described can be found in analyzing the pitches of a melody.

We must activate several cognitive mechanisms which, in their simplest form are:

- processing the melodic profile (the progression of the ups and downs) → occurs more easily in both children and adults, with or without musical experience or literacy.
- processing musical intervals (the distance between pitches) → occurs less easily in general and to a greater extent in individuals with more experience or musical literacy [9] (melody).

1.1.5 Re-compose a puzzle



Re-constructing an image.

An abstract thing reconstructed in the physical world through a sequence of actions.

- we continuously interact between abstract and real world through attempts to correctly reproduce the original image.
- we can reproduce the process many times but we cannot consider them copies.
- we can't modify it.
- it does not require necessary linguistic representation but cognitive skills.
- during the re-composition process it need time and space but when the process is ended it need only space (in the case of a musical reproduction a space in our memory and consciousness).

This has to do with music representations, interpreters, players and performers.

Up to now we have thought about sound, music and their perception but not about the actors involved in the process of making music.

- composer.
- interpreter.
- listener.

We will delve deeper into these concepts in paragraph *Forms of human expression* (1.3).

1.2 Music's meanings

When we talked about the semantic level we mentioned the meaning but... what is the meaning of music and what does music represent for humans?

We start again with sound waves: they represent music in the physical world like clouds or mountains.

This has to do with both perceptual and cognitive processes.

By abstraction we can consider sound waves as signs [10].

Let's think on these topics:

- their production, transmission and interpretation.
- the ways in which something is communicated and signified.
- the production of symbolic objects.

1.2.1 Signs and sounds

Let's go hiking.

We're driving on a mountain road.

We see cars parked on the side of the road.

We realize that the mountain path we're looking for probably starts there.

We can consider the many parked cars a sign that indicated the beginning of the trail.

Our intentions, the context, and the tracks we encountered along the way activated within us an action-reaction mechanism made up of perceptions, expectations, emotions, interpretations.

We can define:

- the sight of the group of cars → the signifier (plane of expression) → what made us understand something.
- the presence of the beginning of the trail → the signified (plane of content) → what we understood.

The union of these two elements gives rise to a sign that generates an increase in knowledge.

Having found the beginning of the trail is what we can call the *paradigmatic effect* of the sign.

We were looking for something and we found it.

Listen (exemple 1, exemple 2, exemple 3) and answer.

1. what kind of information do they provide us?
2. what is the signified content of these acoustic expressions?
3. what kind of increase in knowledge was achieved after listening to them?

These short sound texts could refer to three classic genres of electroacoustic music: soundscape composition, computer music and tape music.

All are carried by sound waves but have profoundly different signified contents that cannot be deduced (or composed) starting from the acoustic parameters of the sound.

1.2.2 Composers, players and listeners

Come back to the mountain path.

- we're walking on the path.
- the clouds are gathering → signifier.
- we understand that it's about to rain → signified.
- we decide to come back to the car → practical effect.

Signs often have a practical effect.

Sounds often have a practical effect (hearing thunder is like seeing clouds).

Have music pratical effect?

Signs connect:

- something perceptual → the sight of cars or clouds with
- something cognitive → the presence of the path or impending rain.

Signs are not signifiers → their perception by someone is.

A sign is not something that represents something else.

It represent the relationship someone establishes between two elements:

- a perceptible dimension (the view of the clouds).
- an intelligible dimension (we connect it to the possibility of rain).

This relationship:

- occurs *a posteriori*.
- is not necessarily intentional on the part of the sender (clouds are not harbingers of rain, just as the person who parked on the side of the road didn't mean to tell us the start of the trail).

We can connect signifier and signified only through an experiential factor (previously, when we saw those clouds, it rained).

Signification → as the examples just given, we are constantly surrounded by a multiplicity of unconsciously produced signs.

All these signs have the potential to become signifier expressions if interpreted by a recipient in the general act we can define as signification.

Communication → the sign is voluntarily produced by the emitter to convey a message, such as the signs in the previous image or the same images within this text.

Listen (exemple 1, exemple 2, exemple 3) and answer.

- what is the practical effect of the three sound texts?
- is sound information contained in the acoustic properties of the sound texts?

Sounds can be both signification and communication.

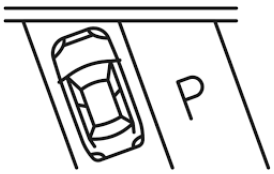
Music can be only comunication.

In the performing arts of non-oral traditions (such as the western musical tradition), things get even more complicated because several actors play in the communication process:

- composer → usually thinks of a composition in his mind (symbolic musical language) and then represent it through some kind of notational code on a score.
- score → the coded instructions needed by an interpreter to reproduce the composer's ideas in the physical world in the form of sound waves.
- interpreter → must know both the musical language used by the composer and the symbolic language through which it was codified (semiography).
- listener → receiver that interprets an interpretation.

In acoustic music it is always like this, while in electroacoustic music things can change.

1.2.3 Do you know this music?



What is the relationship between signifiers and signifieds activated by the receiver based on?

Is this relationship always valid and somehow implicit in the signifier, or does it depend on context and conditions?

Let's consider the example of clouds and rain.

There is no an universal law that says that every time there are clouds in the sky, it will rain.

The relationship is established through *a posteriori* inductive generalization codified in an interpretative rule → it often rained after observing clouds in the sky, and therefore, whenever we encounter clouds, we associate them with the phenomenon.

Does this also apply to the example of cars parked on the side of the road?

In this case:

- there isn't an universal law according to which every time we observe parked cars, a mountain path is nearby.
- there isn't an interpretative rule established by inductive generalization, since parked cars rarely indicate the start of a trail.

The way of reasoning that led to the signification (inference) was induced:

- by the contingent dimension → place and context.

- by the affective dimension → enthusiasm for the hike, desire to find the trail to start walking, etc.
- by the reference culture, or that set of knowledge, value systems, habits, and behaviors within which we live and which have almost naturally led us to that conclusion and not another.

The cognitive inferences used daily in our interpretations are not entirely personal or subjective.

They are based on precise codes.

These codes:

- are formal systems that transcend individual choices.
- are not universal.
- are more or less lasting stabilizations of collective ways of thinking, acting, desiring.
- are dictated, maintained, and modified by social pressure.
- are social and cultural customs, interpretative habits that take on the appearance of a law (without actually being one).

If we hadn't shared the same culture and been part of the same society with the many people who parked their cars on the day of the excursion, that sign simply wouldn't exist.

The sign isn't in things or ideas, but in the forms of their relationship.

Listen (exemple 1, exemple 2, exemple 3) and answer.

Describe the differences between the three sound texts:

- what is the first information that comes to mind when you listen?
- what kind of information is?
- are there differences between the linguistic systems used in the three sound texts?
- if there are any, what information do they influence?

Some of these arguments lead us to understand sound signs as a cultural phenomenon and not merely acoustic since from this point of view the signifier coincides with the signified.

1.2.4 Do you like this music?

The above codes are anthropological conventions.

The relationships between expressions and content are connected to values, preferences, and tastes, all of which are extremely complex and are:

- arbitrary → when viewed from the outside.
- undisputed → when viewed from the inside.

A sign arises through difference when a sensorial discontinuity materializes, such as when, while driving along a mountain road, we encounter parked cars.

The perceptual gap brings to the surface something significant, important, which we value by distinguishing it from other signs encountered along the way.

We value it in two ways:

- we are happy because we have found what we were looking for, namely, the opportunity to embark on a beautiful hike in the mountains.
- we have achieved what we were seeking through a journey of knowledge that has kept our attention high (observing signs, the environment, other signs, etc.) and evaluating by comparing things (would it be better to stop at a restaurant or go for an hike?).

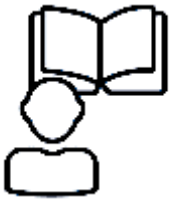
Social codes do not completely transcend individual choices but are linked to the ways in which the individual assumes them by mediating between collective and individual values.

In summary:

- value is what we aim for and what directs our series of actions and passions, giving precise meaning to each of them according to an implicit path.
- value arises from the evaluative comparison of things, objects, and from the recognition of differences; the greater the evaluative comparisons along the path, the greater the value.
- the continuous mediation between collective and individual values generates changes in anthropological codes.

Let's see a video [11].

1.2.5 Texts and narratives



The sign is just an iceberg tip.

It works because:

- it breaks down into many small, interrelated elements.
- it composes larger entities by relating to other similar signs.

The parked cars signify the discovery of the path because:

- their unusual number in that place.
- the way they are parked on the edge.
- they thicken as they approach the path and then thin out.
- they are in that place and not somewhere else.
- they take on value thanks to differential relationships with other signs.
- we are looking for the beginning of the path and not something else.
- all these and other elements are part of a story that unfolds in a precise period of time.
- this narrative is seen through a specific value-based, cultural, and anthropological perspective (a mountain hike).
- we are in a specific emotional state (positive tension generated by the anticipation of being able to begin something we enjoy).
- ...

The relationships established between these elements in a narrative generate the code through which that signifier refers to that probable meaning and not to another, forming what we might call a dynamic text.

Signs function because they are intertwined in texts.

The words of a language are like signs:

- the variable result of constant relationships between smaller elements (morphemes, phonemes, sound features),
- themselves entities that compose larger structures (sentences, texts, speeches).

Every larger element capable of meaning transcends the smaller elements.

The meaning of a sentence transcends the meaning of the individual words that compose it, as well as the individual phonemes.

Texts are not just books, documents, etc., but any portion of the world with:

- determined limits (context).
- precise internal articulation (code).

that carries some configuration of meaning, or better yet, a signification.

Do you remember the puzzle?

In order for meaning to:

- be produced.
- be circulated.
- be received.
- be transformed.

it must refer to texts or units of meaning that various societies, historical periods, and cultures use to define their core values.

Let's see a video.

1.3 Forms of human expression

We can summarize the concepts just presented by following the thoughts of the musicologist J.J.Nattiez set out in his text *Music and discourse*.

Forms of human expression (including music) can be defined as symbolic forms only if we recognize three levels:

- poietic dimension (emitter) → the set of strategies activated by the author that lead to the creation of the work (something that did not exist before).
- aesthesic dimension (receiver) → the set of strategies activated by the listener during the perception of the work.
- the work itself (neutral) → the text with its own internal organization that can be analyzed independently from the poietic and aesthesic dimensions.

1.3.1 Poietic dimension

Active process of construction by the emitter of interpretant signs (composer).

Each sign does not refer directly to an object but makes use of the intermediary action of a second sign which is its interpretant in a process of multiple references.

For example, in verbal language, if we write a word like *table*, we have:

- created a graphic sign (grapheme - formed by a sequence of letters and defined by its own rules) that, if
- correctly interpreted by the reader, will cause him or her to pronounce the corresponding word (morpheme - sequence of phonemes defined by its own rules). If
- correctly interpreted by the hearer, it will refer to a general idea of a table (an abstract concept defined by conventions derived from collective experience). If
- correctly interpreted by someone, it will connect it to a real, meaningful table (object - defined by subjective experience).

This is true for a single word.

The interpretative signs become infinite when individual words are organized into a language.

Ognuno sta solo sul cuor della terra trafitto da un raggio di sole: ed è subito sera

Let's try to imagine the same process in music and the amount of control the composer can have over the sound information [12] he wants to transmit.

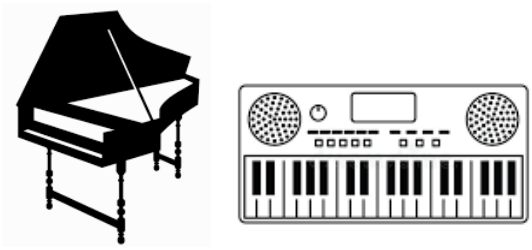
1.3.2 Aesthesic dimension

Active process of construction by the interpretant.

Process of understanding that can be defined as sharing information on cultural operations typical of a human social group.

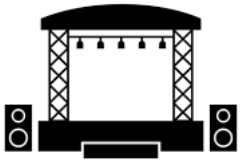
In the case of artistic languages, the interpretative signs attributed to the work by the emitter are not necessarily the same as those projected by the receiver because:

- they are modulated by the context in which the interprete perceives the work.
- they are modulated by any discrepancy in period or cultural context between the emitter and the receiver.
- the perception of a sonata in the Baroque era is presumed to be totally different from the perception of the same work today, as the historical, social, technological, and cultural contexts are significantly different.



The perception of a Beethoven symphony by an Amazonian native is totally different from ours, as is our perception of the microtones present in the muezzin chants [13].

Let's also think about the technological dimension: the perception of the same sound text in a live concert is different from the perception of its reproduction recorded by an electroacoustic system perhaps while we wait for the subway.



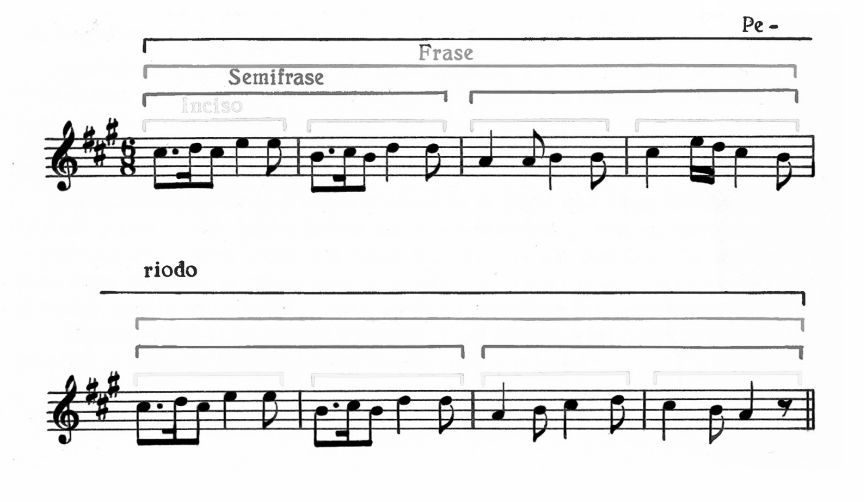
1.3.3 The work itself

Regarding the understanding of the sound text itself Nattiez advocates the need to analyze:

- works and styles as cultural and anthropological objects.
- harmony, rhythm, meter and all aspects that have a similar function in the grammar of natural language (comparison).

When we speak of grammars of musical parameters we are not referring exclusively to the analysis of the formal and structural characteristics of a specific musical text. We must also take into consideration the grammars of musical parameters generalized and historicized in styles and forms.

Nattiez borrows from the study of verbal language the semantic exercise of breaking down meaning according to the use of words, which in music corresponds to the analysis of a melodic profile or a rhythmic system, or the distribution of harmonic fields in a tonal system or twelve-tone series, etc.

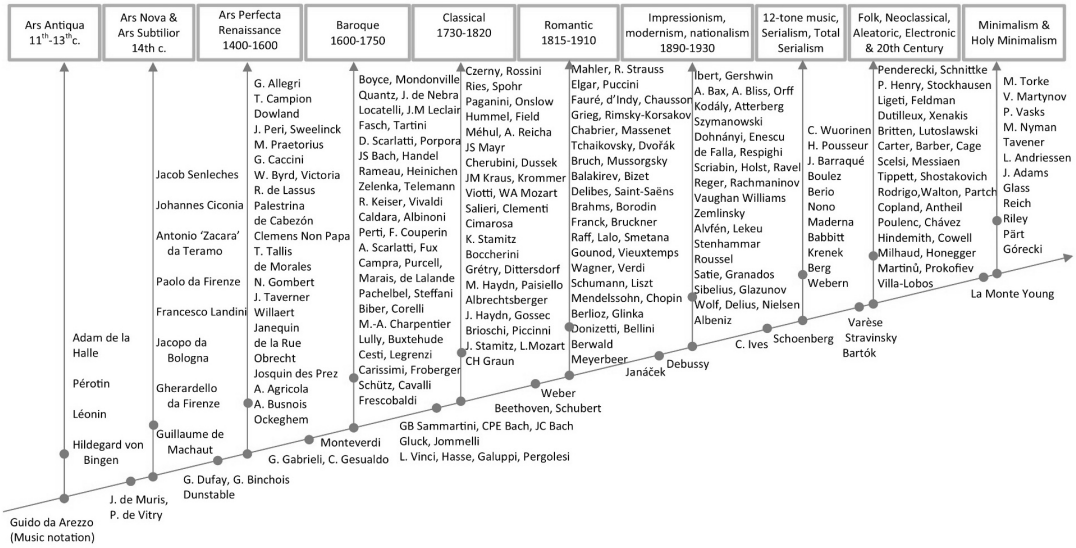


Each musical element (pitch, intensity, duration, timbre) can be separated and taken as a strategic variable in the production of a musical work [14].

For Nattiez constructing a musical theory is the organization of a selection of interpretants adopted within an anthropological context.

Anyone who wants to invent a new musical grammar (a practice that has been widespread among art music composers since the 1950s) must take this into account and not just the definition of rules internal to the text.

Works do not exist in isolation since it is their place in history that provides their meaning and possible interpretations over time.



The sound text can live a life of its own and be isolated from the the emitter but not from the anthropological context in which it was created and/or reproduced.

1.3.4 Final reflections

Everyday life is filled with more or less voluntary signification.

Signification is the amniotic fluid within which human beings live.

We can highlight it and clarify its mechanisms.

This task is often hidden in everyday life.

If we ask someone why they like a piece of music or a dress they will probably answer:

- because it's beautiful.
- because it excites me.
- because it's right.
- because. . .

Meaning often hides behind other motivations that tend to suppress it.

Let's think about written and spoken language.

In using it we put into practice extremely complex knowledge that we tend to be unable to explain.

We would hardly be able to fully explain all the grammatical, phonetic, morphological, and syntactic rules we apply daily.

Language is practical knowledge that we almost never reason about.

Our daily experience works the same way: it appears random, spontaneous, and immediate.

In reality it is based on a dense and constantly changing network of relationships between things, people, places, etc.

Also the act of composing a piece of music should develops in the continuous passage from awareness of linguistic mechanisms (intelligence) to creativity.

Let'see a video [15].

Chapter 2

Genres, instruments and tools

2.1 Genres

Before starting to write an electroacoustic piece we should:

- define the instrumentation to be used which can include acoustic instruments, electroacoustic instruments, virtual instruments or a mix of these types.
- define the sound amplification and diffusion system.

For this reason we categorize the different types and genres of electroacoustic music by dividing them into two macro areas defined by the way of controlling sounds and musical parameters.

- non interactive systems.
- interactive systems.

We will explore the compositional strategies of each in dedicated chapters.

2.2 Non interactive systems

This category includes genres that:

- don't involve any human interaction during the performance.
- involve interactions that don't influence the musical or sound text in any way.

According with the reflections of the previous chapter we can affirm:

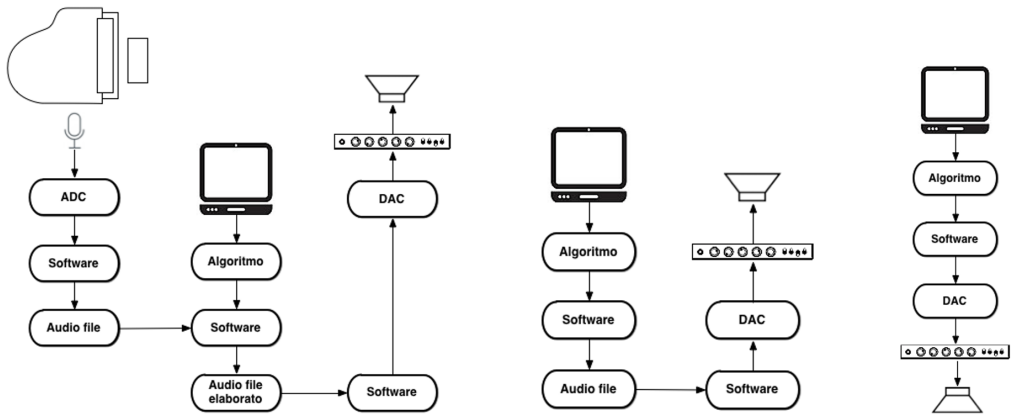
- interactions may influence the listener's perception of the sound text (mainly through the different characteristics of the diffusion systems used).
- interactions don't change the sound text.

This category can be divided into two further subcategories:

- fixed media music.
- live sequencing.

The image shows the possibles audio chains in this category.

- non-real-time sound processing.
- non-real-time synthesis.
- real-time synthesis.



2.2.1 Fixed media music

This subcategory denotes music or video that is played back from a recording.

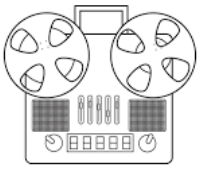
This used to be called *tape music* but tapes are no longer used.

The entire work is stored on a single medium and reproduced from beginning to end.

We can identify three main genres:

- acousmatic music.
- wallpaper music, ambient music and soundscapes.
- radiodrama.

Acousmatic music



We can start from *musique concrète*.

In the 1940s Pierre Schaeffer began composing with the sound (acoustic world).

After applying various sound elaborations (editing, filtering, reversal, etc.) he began to apply musical techniques to their combinations (repetition, transposition, reordering, etc.) inventing new musical structures (symbolic world).

In doing so, he reversed the compositional practice that had been in force for centuries in the western musical tradition.

The listeners could not make the cognitive associations illustrated in the first chapter.

They activating exclusively the basic cognitive mechanisms common to all human beings.

This listening mode has been called *reduced listening*.

All listeners of this new music are comparable to the aborigine who listens to a Beethoven symphony.

P.Shaeffer - Etude aux chemins de fer for tape (1948) [16].

- mono analog tape.

- recorded and processed train sounds.
- sounds from the acoustic environment.
- we can recognize the sound source and its processing.
- perceived as a distortion of reality.

Acousmatic music is the modern evolution of musique concrète.

By the mid-1970s a number of composers felt a need to designate their conception clearly in terms of specific methodology, syntax, and tools.

François Bayle suggested adopting the term *acousmatique*.

This term comes from the greek *akousma*: what is heard.

Students of Pythagoras listened to his teachings from behind a screen unable to see him.

He believed that the lack of visual cues would force his students to focus all their attention on his message.

After his death his followers split into:

- acousmatics (practitioners of the mystic doctrine)
- mathematics (scientists).

The loudspeaker was the modern equivalent of the screen partition.

"From an esthetical point of view acousmatic music concentrates on the poetical and spectral richness of sounds, and plays with this very particular characteristic of sound hearing in which the perception of anacoustic phenomena is associated with its cause; hence the perception of a sound whose cause is unknown or unrecognizable for our perception, induces the listener to imagine non-existing causes and to perceive music as a complex creative phenomena in which musical sense and musical sounds have to be interpreted simultaneously, with generally very little relation with our perceptive reality. The question is not to find out how sounds are made but how their combination will generate imaginary perceptions of imaginary realities in our mind."

Teruggi, D. 1995. What about acousmatics? In Journal of Electroacoustic Music. Vol. 7. London: Sonic Arts Network

More info

B.Parmegiani - De Natura Sonorum for tape (1975) [17].

- stereo digital tape.
- recorded and processed sounds plus synthetic sounds.
- sounds from an imaginary environment.
- we cannot recognize the sound source.
- perceived as an alternative or virtual reality.

Another characteristic of acousmatic music is that it is performed live on loudspeaker orchestras called *acousmonium*.

Indeed Bayle referred to acousmatic musica as:

"art of projected sounds which is shot and developed in the studio, projected in a hall, like cinema".

H.Vaggione - Consort. for convolved violins (2011) [18].

- multitrack digital file.
- recorded and processed instrumental sounds.
- sounds from music environment.

- we can recognize the sound source and its avatars.
- perceived as a piece of music.

More info about acousmatic works interpretation.

Wallpaper music, ambient music and soundscapes

We can start from Eric Satie (non-electroacoustic composer).

He was a precursor of minimalism, muzak, and many other 20th-century musical genres.

Satie conceived this musical genre as a backdrop to everyday activities (passive listening).

Satie was familiar with this idea due to its role as role as a Montmartre café pianist in the late 19th and early 20th century.

E.Satie - Musique d'ameublement (1917) for ensemble [19].

- acoustic instruments.
- pattern iteration.
- music environment.
- time dilation (psychological time).
- passive and non immersive listening.
- perceived as a piece of music.

We know well how nowadays we are bombarded daily by this type of music in a totally involuntary motion.

Let us recall what was stated in the first chapter on the basic cognitive mechanisms.

In the 1950s a visionary composer like J.Cage took this idea to its extreme with the piece 4'33'' whose sonic and musical content consists of the background noise of a finite time.

Later in 1978 another composer Brian Eno coined the term *ambient* as a musical genre.

A Sunday morning in the Cologne airport while waiting for a flight he tought:

The light was beautiful, everything was beautiful, except they were playing awful music. They spend hundreds of millions of pounds... on everything. Except the music.

It was this that compelled him to begin composing music for public environments.

We must consider that the diffusion of sounds and/or music in environments not designed for active listening can change the perception of that environment in a subconscious way.

It is time that through sound redraws the boundaries of a space.

B.Eno - Music for Airports (1978) for tape [20].

- stereo vynil.
- pattern iteration.
- music environment.
- time dilation (psychological time).
- passive and semi immersive listening.
- perceived as a piece of music.

More info

Soundscape composition can represent a meeting point between musique concrete and ambient music because:

- it uses sound materials recorded from the real world or synthesized.
- does not recompose them according to symbolic rules and principles into a musical form.
- its purpose is to construct or reconstruct a real or virtual soundscape.

It's essentially based on the studies and works of canadian musicologist and composer Raymond Murray Schafer.

To put it simply, we can say that different types of sounds coexist in a sound environment and we can identify three types:

- geophony → all the sounds of nature of non-biological origin.
- biophony → all the sounds of nature of biological origin but not caused by humans.
- anthrophony → all sounds generated by humans.
  - language.
  - music.
  - mechanical sound.

Schafer also divides the sounds of a soundscape into two categories:

- hi-fi → all sounds that have an hight signal-to-noise ratio (they are distinguishable from the background noise).
- lo-fi → all sounds that have a low signal-to-noise ratio (they are indistinguishable from the background noise).

Three main elements in a soundscape:

- keynote → sounds that most characterize an environment, analogous to the tonality in tonal music.
- sound signals → all the foreground sounds that do not necessarily characterize an environment but may appear occasionally.
- soundmark → The unique sounds of a given soundscape.

N.Barret - Soundwalk (1998) [21].

- stereo digital tape.
- no pattern.
- acoustic environment.
- time as in real life (clock time).
- passive and immersive listening.
- perceived as an acoustoc environment.

More info

Radiodrama

Radio drama can use only four kinds of signs:

- speech.

- sound effects.
- music.
- silence.

Any one of these by itself can be a very slippery client to deal with.

Silence cane a listener suspect a transmission failure and switch off or tune over to another on.

Sound effects are notoriously misleading - a sneeze can be taken for a bomb explosion!

An unfamiliar music will sound irritatingly like cacophony.

And speech says nothing to someone who doesn't know what you mean by the words.

But if you hit on harmonious combination of the right choies of these four kinds you can move mountains, deploy infantry battalions with Air Force support, immerse a soul in the joys of paradise.

In short do anything you please, all in few minutes one after another.

The main focus is narrativity and a continuous balance in the construction of meaning between explicit sounds (both in natural and auditory language) and perceptual hints and/or allusions.

O.Wells - War of the worlds (1938) [22].

- mono radio signal.
- natural language.
- didactic sound design to simulate the real acoustic environment.
- time as in real life (clock time).
- active listening non immersive.
- perceived as a representation of reality.

More info

Computer music - algorithmic composition

The birth of computer music is naturally linked to the invention of the electronic calculator and the subsequent spread of personal computers as well as the invention of new (formal) languages that allow humans to communicate with these machines.

Starting from the second half of the 1950s, two orientations emerged:

- the first one (developed mainly in Europe) deals with the digital coding of musical parameters (assisted composition, assisted musical analysis, etc.).
- the second one (developed mainly in USA) deals with the digital coding of sound parameters (synthesis and processing).

In 1957 MUSIC I was created by Max Mathews in Bell Laboratories.

It was the first software for playing music from mathematical functions.

After MUSIC I, there were: MUSIC II, III, IV, V, MUSIC 360, and then C-sound and SuperCollider languages still used today.

The main sound synthesis and processing techniques used are:

- additive synthesis.
- subctractive synthesis (filters).
- FM (Frequency Modulation) synthesis.
- granular synthesis.
- analysis and resynthesis (FFT).
- generative algorithms for:
  - musical structures.
  - timbres.
  - pitches.
- remote control of computers (MIDI, OSC, HID).

As we all know, nowadays computer is the main musical instrument, like piano in the 19th century.

The term *computer music* as an aesthetic category identifies the musical genre adopted by the pioneers from the mid-50s to the late 90s

J.Chowning - Stria. (1977) [23].

- stereo digital tape.
- sonic environment.
- sounds from imaginary environment (only synthetic sounds - FM).
- time dilation (psychological time).
- active listening.
- perceived as a piece of computer music.

More info

2.2.2 Live sequencing

This subcategory is almost identical to the previous one except for two aspects:

- works can be generated in real time through algorithms (computer music).
- tracks are divided into different sound files triggered in real time (sound design for theatre and gaming).

Sound design for performing arts or video

Works on fixed media which differ from the works illustrated in the previous section for:

- sound does not have an absolute value but must be functional to dramaturgy.
- divised into more or less short fragments.
- open form → fragments can be triggered in non-linear sequences (slightly different than what was said about random generation).

Anonymous - Dance! (2010) [24].

- stereo digital tape.
- music functional to movement or dramatic timing.

- secondary sensitive plane.
- passive listening.
- perceived as a complement to visual information.

Sound design for gaming

In this case too the sound must be functional to the game

It should:

- suggest a mood, evoke a feeling.
- indicate a geographical locale.
- define a character.
- mirror or exaggerate how things sound in real life.
- clarify the narrative

To simplify there are five sounds categories:

- dialogue → any verbal speech in the game (player talk).
- music → any non-diegetic music (orchestral music).
- sound Effects (Hard Effects) → any sound from an real-life object (sounds of rocks falling).
- foley → any sound effect that the player makes (footsteps, etc.).
- backgrounds (ambience) → noise from the environment (wind, rain, etc.).

Procedural audio:

- sounfiles triggered by user actions.
- real time shynthesis by internal oscillators.

Artistically we could merge game music, soundscape composition, acousmatic music and radio-drama.

Various - 8 bit music(1990) [25].

- stereo digital samples or real time oscillators.
- music functional to gamer emotions.
- also called procedural music.
- pattern iteration.
- music environment.
- time dilation (psychological time).
- non-immersive audio diffusion but immersive psychological function.
- passive listening.

More info

2.3 Interactive systems

This category includes genres that:

- involve human interaction during the performance as happens with acoustic instruments.

- these interactions influence the musical or sound text at different levels.

Compared to the previous typology the presence of one or more performers influences:

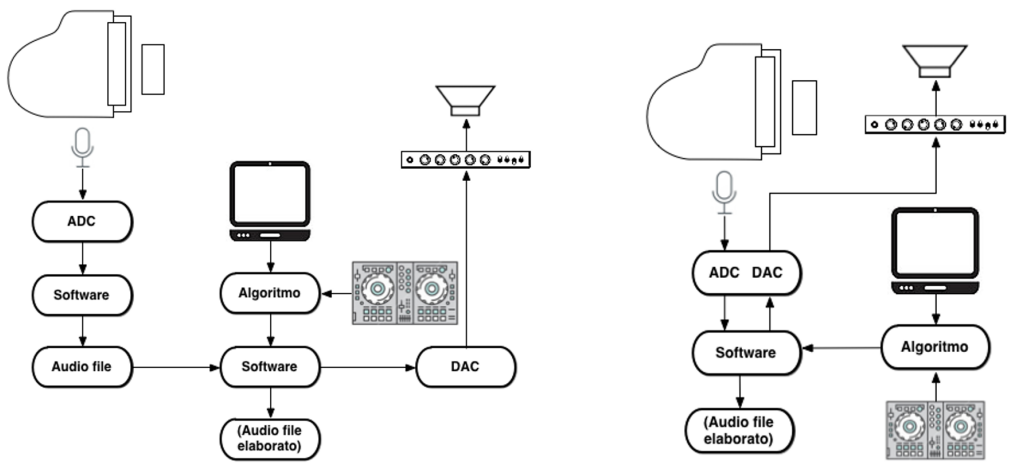
- the anthropological and contextualised perception of the musical action.
- its representative nature.

This category can be divided into three further subcategories:

- live set.
- hyper-instruments.
- live coding.

The image shows the possibles audio chains in this category.

- real-time processing of pre-recorded or pre-synthesized sounds (parameters can be controlled through various types of devices - MIDI, OSC, HID, sensors).
- real-time processing of signals captured through microphones or sensors (parameters can be controlled in the previous modes but also through control signals derived in some way from the incoming audio signal).



2.3.1 Live set

Slork - Twilight (2013) [26].

Live performance using analog or digital electronic musical instruments.

Often musicians improvise in a live set, making each performance unique.

Can be a solo performance or a laptop ensemble.

One of the most important aspects of this genre is to think about and assemble original performance environments which leads to the search for new human/machine interfaces dedicated to the generation of new sounds in the musical field.

We should design integrated organisms understood as networks of humans, musical instruments, and machines where the former communicate with the latter through *creative embodied interfaces* (sensors, microphones, cameras, smartphones, etc.) in the continuous search for new real and

virtual luthiers (and related gestural techniques of instrumental performance) oriented to the changing expressive needs of our time.

We must find the right balance between what humans are good at doing and what computers are good at doing, trying to overcome their respective limitations in an attempt to reach and explore new artistic goals.

For every project we must ask ourselves:

- does the aesthetic-artistic aim of the project justify the technology used?
- is there at least one expressive element in the project that can't be represented exclusively through traditional musical instruments?
- is human-machine interaction intrinsic to the generative idea of the project, so much so as to involve the design of specific executive techniques linked to both real and virtual instrumental gestures?

The same questions apply to projects similar to the next paragraph.

2.3.2 Hyper-instrument

P.Boulez - Anthèmes 2 (1994) [27].

Augmented musical instrument that uses technology to extend the capabilities of a traditional instrument and enhance the performer's expressiveness.

Two types:

- real-time augmented instrumental techniques → sound processes are controlled by:
  - an electronic performer who interacts with different types of controllers.
  - by information extracted from the audio signal produced by the instrument itself (feature extraction as control signals).
- cyborg luthiery → sound processing is controlled by movements and/or gestures of the performer captured by different types of sensors.

Three main goals:

- respond to the performer's input in a dynamic and expressive way allowing for a deeper connection between the musician and the music.
- produce sounds and musical effects that are impossible to achieve with traditional instruments alone.
- expanding musical forms.

2.3.3 Live coding

SuperCollider Live Coding [28].

Also called:

- on-the-fly programming.
- just in time programming.
- conversational programming.

Search for new musical forms, new places for performance and new audiences.

It is part of a larger artistic movement called *creative coding*.

If channeled into the western musical tradition, we can think of it as a new version of organ improvisations in the baroque era.

From a semiographic point of view the most interesting aspect is that the score is the code and is created and manipulated in real time.

The compositional process which has always belonged to the author's private sphere becomes public and an integral part of the performance itself (do you remember the first paragraphs of the previous chapter?).

2.4 The virtual instrument paradigm

In this section we define a virtual instrument model that will be useful in next chapters.

If anyone is unfamiliar with SuperCollider, they can download an introductory text at this link or read the *Appendix A*.

In the chapter on live electronics we'll make some small modifications to allow signal input.

2.4.1 Software installation

Download and install SuperCollider.

We can copy and paste the code into the SuperCollider IDE Interpreter from this file or download and launch the code from link at the beginning of each section.

2.4.2 Instrumental model

Download chapter's SuperCollider code.

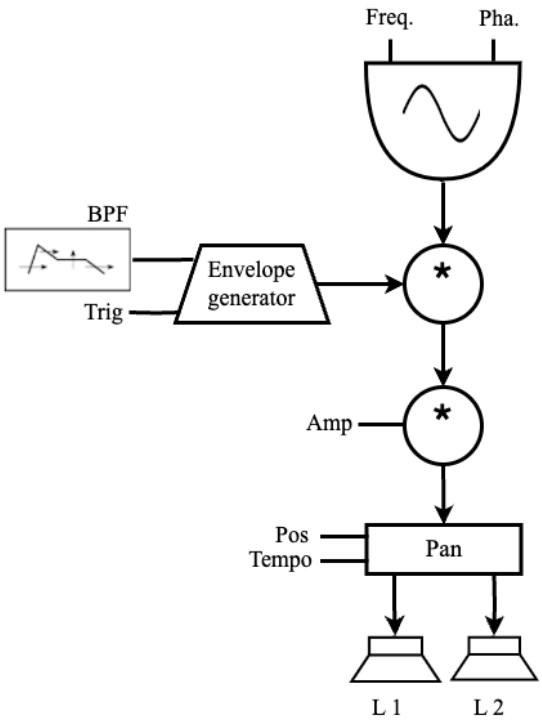
Let's create a computer model that contains functions common to most virtual instruments (sound generators).

- a source signal (oscillator or mic/line input).
- an amplitude envelope (with or without sustain).
- a stereo panner.
- a signal output.

We also establish the basic parameters we want to control dynamically (they will increase depending on the synthesis algorithm chosen).

- frequency (Hz).
- general amplitude (0.0 to 1.0).
- duration (seconds).
- pan position (-1.0 to +1-0).
- pan interpolation time (movements)
- output bus.
- trigger.

- kind of voice allocation (doneAction:0 or 2).



We find a more detailed description of the different components, connections and parameters of this virtual instrument model in Appendix [B].

```
s.boot;
s.meter;
s.scope;
s.plotTree;
```

In SuperCollider we can define it with the *SynthDef* (synthesis definition) class.

This class contains the instructions (algorithms) needed to build multiple copies (instances) derived from the model.

```
SynthDef(\model, {arg freq=500, amp=0, dur=1, pan=0, pantime=0.0,
                      out=0, t_gate=0, done=2;
                  var sig, env;
                      sig = SinOsc.ar(freq);
                      env = Env.new([0.0,1.0,0.5,0.7,0],
                                    [0.1,0.1,0.3,1.0].normalizeSum * dur,
                                    -4
                                    );
                      env = EnvGen.kr(env, t_gate, doneAction:done);
                      sig = sig * env * amp;
                      sig = Pan2.ar(sig, pan.varlag(pantime));
                  Out.ar(out, sig)
                  }).add;
```

Listing 2.1: Instrument model

Now we can generate as many instances as we want.

We have two ways to control the instance parameters.

They vary depending on the voice allocation we want.

Dynamic voice allocation (poliphonic)

Instance self-kill when the amplitude envelope expires (*doneAction:2*).

In this case we pass parameters at instantiation time (array of *\key, value*).

```
Synth(\model, [\freq, rrand(890,1234),
               \amp, 0.5,
               \dur, rrand(0.1,2),
               \pan, rand2(1.0),
               \pantime, 0.1,
               \done, 2,
               \t_gate,1 ])
```

Listing 2.2: Dynamic voice allocation

Static voice allocation (monophnic)

Two passages:

- create instance and assign it to a variable.
- send one or multiple parameter changes by *set* method.

```
a = Synth(\model, [\done, 0]);

a.set(\freq, rrand(890,1234),
      \amp, 0.5,
      \dur, rrand(0.1,2),
      \pan, rand2(1.0),
      \pantime, 0.1,
      \t_gate,1)
```

Listing 2.3: Static voice allocation

Then we can kill it or kill all Synths (Nodes) on the Server.

```
a.free;
s.freeAll;
```

The type of voice allocation is strictly linked to the type of amplitude envelope [B.2.5].

2.5 Composition sketches proposal

Create at least one of the following instrumental models by searching in the UGen browser (Help files) for anything not known.

- – source → white noise signal.
  – envelope → without sustain, percussive.
  – voice allocation → dynamic poliphony.
- – source → *Blip* UGen.
  – number of harmonics → controlled by a LFO.
  – envelope → trapezoidal with sustain.
  – voice allocation → static.
  – pan → controlled by smoothing values from Interpreter.
- poliphony → two homorhythmic indipendent voices.
  – source voice 1 → Pulse wave.
  – source voice 2 → Sawtooth wave.
  – envelope → both adsr.
  – pan voice 1 → stereo left.
  – pan voice 2 → stereo right.
  – voice allocation → static.

Chapter 3

Fixed media

3.1 Framework and historical context

The first musical works recorded on media were created in the 1950s in the Radio France studios in Paris by composers belonging to the Groupe de recherches musicales (GRM) by Pierre Schaeffer.

Composers such as Pierre Henry, François Bayle, Bernard Parmegiani, Christian Zanési and Daniel Teruggi have collaborated with this institute.

These composers, more or less, referred to Schaeffer's sound theories about *musique concrete* later outlined in his book Traité des Objects Musicaux.

The design of the studio followed strict Schaefferian theory and was completely centered around tape manipulation, recording and editing.

Two main electroacoustic instruments:

- phonogène → a multi-headed tape instrument in three versions:
  - Chromatic phonogène. Tape loop at varied speeds. It produced short bursts of tape sounds at varying pitches defined by a small one-octave keyboard.
  - Sliding phonogène. It produced continuous tone by varying the tape speed via a control rod.
  - Phonogène Universal. It allowed time stretching without transposing the pitches and vice versa obtained through a rotating magnetic head.
- morphophone → a type of tape loop-delay mechanism. The sound was picked up at varying points on the tape by ten magnetic heads (one recording, one erasing and ten playback heads) then passed through ten bandpass fileters (one for each head).

Theories of musique concrète are based on the concept of a *sound object*.

3.2 Elements and strategies

In this section we outline some concepts underlying any compositional process.

They are valid in subsequent chapters where we will explore the specifics.

They are some of the main tools of composing electronic music.

3.2.1 Sound objects

Let's explore the concepts of sound object.

Pierre Schaeffer in his *Traité des objets musicaux* (1966) wrote:

Let's assume a single microphone:

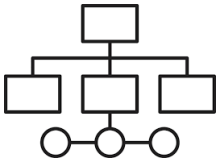
- *it is the point of convergence of all the soundwaves arriving from the sound points in the surrounding space.*
- *all the sound points in the initial space will be condensed into the microphone's membrane.*
- *the initial surrounding space is replaced by a sound point.*
- *if we play back it, it generates a new sound distribution in a new surrounding space.*
- *the diffusion medium (loudspeaker) is neutral favoring an acousmatic sound perception.*
- *we can consider this sound point as a word, a representation, not a recording of reality.*
- *we can call this representation a sound object.*

A sound object is any sound phenomenon perceived:

- as a coherent whole.
- heard in an acousmatic situation, regardless of its origin and meaning.

P.Henry - Futuriste for tape (1975) [29]

3.2.2 Objects, structure and form



Schaeffer argues also that the object/structure pair is at the basis of our perceptual activity.

What is a structure?

A structure is a model the skeleton of a form.

What is a form?

General concept exposed by A.Lalande (Vocabulaire technique et critique de la philosophie):

Forms are sets, which constitute autonomous units, manifest an internal solidarity, and have their own laws.

The nature of each element depends on the structure of the whole and the laws that govern it.

Neither psychologically nor physiologically does the element (object) pre-exist the whole.

The concept of form is therefore more complex than that of structure

Schaeffer illustrates this concept with two examples from western musical tradition:

- a melody [30] cannot be reduced to the sequence of notes that compose it.
  - if when we listen to a melody we focus our attention on a single note, the perception of the melody vanishes.
  - if we focus our attention on the entire melody (melodic profile) the perception of the individual notes vanishes.

- if the melody is in G major and we play an F natural we perceive a discordant note.
  - the musical scale is a structure that influences the encoding and decoding of the notes.

Let us remember the continuous reference of the signs discussed in the paragraph about musical meaning.

Structure provides the relevant features that enable members of a musical tradition to encode and decode sounds.

In music, these structures are usually organized according to coherent variations of characterizing models (major, minor, etc.).

Schaeffer then aims to define a new sound vocabulary based on the intrinsic sound characteristics (morphosyntax) of sound objects.

Sound objects become musical objects.

3.2.3 Musical objects

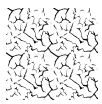
When we start to write a sound or musical work on a fixed media we should define few procedures:

1. choose and/or define the sound material to be used in the piece (sound objects).
   This choice should be made by:
   - recorded sound objects → analysis of their morphological characteristics (perception).
   - synthesized sound objects → definition of their morphological characteristics (idea of a sound).

   In both cases (analysis or definition) we can employ some simple strategies derived from the much more complex theories of composers and musicologists such as P.Schaeffer, M.Chion, D.Smalley and others.
   We should think about:
   - mass → (spectral) quality that describes its perceived consistency, density, or *weight*.
     - pure sound (sinusoid).
     - tonic sound - defined pitch harmonic spectra (instrument).
     - tonic group (chord or melodic pattern whitin a range).
     - mixed sound (ambiguos).
     - node (tremolo or ribattuto).
     - nodal group (texture of inharmonic sounds).
     - noise (white, pink, etc.).
   - matter → grain - texture - object surface.
     - smooth (static).

     - harmonica (trembling).

     - compact harmonica (fluctuating).



     - compact (rustling).

- compact discontinuous (rhombus).

- discontinuous (crackling).

- discontinuous harmonic (squeak).

- rhytmic (measured pattern).

- dynamic.
  - global (dynamic profile) - if a sound object is fortissimo or piano or...
  - internal (grain envelopes) - dynamic evolution or contrast in internal parts.

Each of these parameters may or may not evolve over the course of the sound object in different ways:

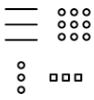
- static (-) → no evolution or small small fluctuations (tremolo, vibrato, beating, etc.).

- opening (<) → up glissando, crescendo, increase in density, etc.

- closing (>) → down glissando, diminuendo, decrease in density, etc.

- alternated (<>) → open and close like a pendoluum.

- variated → variation in pattern (rhythmic or melodic or spactral, etc.).



- random → non evolutive.

All these evolution can be fast or slow.

2. build a sound vocabulary (a sound palette) that includes one or more sets of sounds elaborated from those chosen in the previous point.
   The criteria adopted in generating this sound palette can be varied, including:
   - a set of coherent sound objects with morphological similarity to the original.
   - a set of sound objects with gradual differentiation (direction, interpolation, etc.).

- a set of incoherent sound objects where the sound processing distort the original morphology.

The sound objects included in these sets become *musical objects* because they have acquired a semantic function within a system (such as the division of the octave into intervals of a scale in Western music theory).

3. order (compose) these sounds over time into different types of sound textures and paths, designing a musical form.
   The criteria we can adopt in doing so can include among others:
   - staticity - evolution - variation.
   - continuity - discontinuity.
   - contrast - uniformity.
   - etc.

In this way we can construct and formalize an abstract thought.

Let's try to recognize musical objects, their characteristics and how they evolve over time in these extracts from B.Parmegiani works (extract1, extract2, extract3).

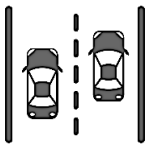
3.2.4 Time courses and musical forms

Some further thoughts regarding time courses and musical forms.

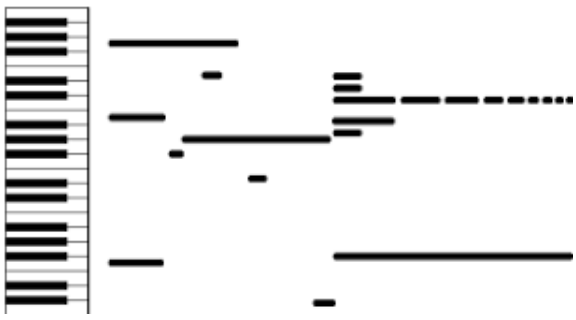
In the figurative arts (architecture, painting, sculpture) forms exist in the dimension of space.

Musical forms are created over time.

We cannot think of time in forms but only as events with different temporal courses.



A musical form can be defined as a thought that thinks about the course of events.



In the temporal domain (and therefore in the musical domain) there are two main formal typologies:

- teleo-logical.
- circular.

Teleo-logical time

This time-course is goal-oriented and plays a dominant role in Western music.

The mode of a medieval melody is determined by the finalis which is the last event of the course.

In tonal harmony a dissonance tends toward resolution just as a cadence moves toward the closing tonic.

Over a longer time an opera or the movements of a symphony gravitate towards the Finale (micro and macro structure).

Tonality based on harmonic functions has a teleo-logical nature as the sound aggregates are constantly moving towards reference points.

A point-to-point movement.

Only at the end when we reach the final cadence the process of the whole being formed into a figure temporally accomplished.

Circular time

Movement returns incessantly to itself; it has no goal, no beginning, and no end.

In Western music, it appears:

- in the dissolution of functional harmony (early 1900) - exemple.
- in phenomena of the serial or stochastic pointillistic form (mid 1900) - exemple.
- in phenomena of the anvantgarde and experimental music (after 1950) - exemple.

Music is de-teleologized and de-temporalized, eliminating any defined profile.

Thought that considers the process frees itself from the chains that bind it to hierarchies and the need to strive toward a goal.

A typical position of Eastern philosophies, thought seeks to free itself from all limitations to achieve the unity of being.

The concept of time in music has always historically stemmed from a general reflection.

For this reason, today we can choose which temporal typology to adopt to best construct the desired musical form.

3.2.5 The poietic dimension: determinism

In the previous chapter we saw that the musicologist J.J.Nattiez defines the poietic dimension as the set of strategies activated by the author that lead to the creation of the work (something that did not exist before).

As for the simple organization of sounds over time, stripped of all other components that contribute to their creation, the processes we can employ in design and implementation are essentially two types:

- deterministic procedures.
- stochastic procedures.

We will deal with the first type in this paragraph while the second in the next chapter on computer music.

In philosophical and scientific language, determinism is a conception according to which the events of metaphysical, physical, or moral reality are necessarily and invariably connected to each other.

If we make this choice when composing a piece, we face two options:

- use our *libero arbitrio* and choose all the parameters at our own discretion (not randomly, but according to a specific intention).
- follow or create systems of rules to generate or transform some or all of the parameters.

Libero arbitrio

This procedure may seem the simplest.

In reality it is the most complex because it cannot be formalized.

Its success depends exclusively on the composer's determination and experience.

It requires a high level of consciousness.

It presupposes a full awareness of oneself and the external world with which we interact, our identity and the complexities of our internal activities.

This is because the conveyance of the musical idea cannot by definition be governed by rules shared by:

- composer.
- performer.
- listener.

If we choose this procedure we must be careful to avoid any mnemonic reference to any musical systems.

It is a-thematic and a-motivic in the form of an infinite ever-changing continuous line.

Without developments, perceptible variations, or repetitions.

It is perhaps the procedure closest to pure unmediated expression.

We can consider it the ultimate point of arrival in the historical development of a musical civilization, absolute freedom understood not as a void but as a container of all possible rules.

Rules and theories

The second option is to:

- choose a system of rules from a past or present musical tradition.
- define our own system of rules that can be valid only for a single piece or a distinctive feature of our poetics.

The main difference is that in the first case the codes of meaning are shared within a reference culture while in the second they are not, or not yet.

Unlike in the past, today we can choose the system that best conveys our musical thinking because never before have our ears and senses been bombarded as continuously by every type of sound and language from the most disparate media as we are in our daily lives.

This idea is masterfully expressed in the appendix to some editions of I. Calvino's American Lessons, entitled *Beginning and Ending*.

[...] Start writing a novel.

And this is the moment of choice: we are offered the possibility of saying everything, in every possible way; and we must arrive at saying one thing, in a particular way.

The starting point [...] will therefore be this decisive moment for the writer: the detachment from the unlimited and multifaceted potentiality to encounter something that does not yet exist but that can exist only by accepting limits and rules.

Up until the moment before we begin to write, we have the world at our disposal, a sum of information, experiences, values – the world given as a whole, without a before or an after, the world as individual memory and as implicit potentiality; and we want to extract from this world a discourse, a story, a feeling: or perhaps more precisely we want to perform an operation that allows us to situate ourselves in this world.

We have all the languages at our disposal: those developed by literature, the styles in which civilizations and individuals have expressed themselves over the centuries and countries, and also the languages developed by the most varied disciplines, aimed at achieving the most varied forms of knowledge: and we want extract from it the language suitable for saying what we want to say, the language that is what we want to say. [...]

3.3 Sound processing techniques in SuperCollider

Download chapter's SuperCollider code.

In this section we implement in SuperCollider some historical sound processing techniques controlled through deterministic procedures (scoring and sequencing).

Through these sound processing techniques we can define the sound palette (vocabulary) illustrated in point 2 of the previous paragraph.

We can divide them into two categories based on the characteristics of the oscillator type used:

- sampling.
- scratching.

In both cases we need to load the soundfiles into one or more buffers.

3.3.1 Buffers

When we want to process a sounfile I suggest to use mono files.

Boot the audio system with few GUI.

```
s.boot;
s.meter;
s.scope;
s.freqscope;
s.plotTree;
```

Load a single soundile in one buffer.

```
b = Buffer.read(s, "/absolute/path/to/file.wav");
```

Listing 3.1: Load one sound file in a Buffer

A best practice it's to include all the soundfiles in one folder providing the relative path.

It work only in SuperCollider IDE, not in Jupyter sc_kernel.

```
"sounds/bach.wav".resolveRelative;
```

Some utility.

```
b.normalize;
b.plot;
b.play;
```

Delete Buffer(s) from Server's memory.

```
b.free;      // one specific Buffer
s.freeAll;   // all Buffers
```

Load many sounfiles from a folder.

One soundfile → one Buffer.

```
~paths = ("/absolute/path/to/folder" ++ "/*.wav").pathMatch;
~bufs  = ~paths.collect{arg i; Buffer.read(s, i)};
```

Listing 3.2: Load many sounfiles from a folder

We can then recall them by index (uditive monitor).

```
~bufs[1].play;
```

Stopping audio:

- SuperCollider → cmd + .
- Jupyter sc_kernel → execute the cell below.

3.3.2 Sampling

Let's define an instrument and the parameters we want to control:

- buffer to play.
- start position.
  - absolute (0 to Buffer duration in seconds).
  - relative (0 to 1).
- duration (seconds).
- amplitude (0 to 1).
- trasposition in semitones (float → cents).
- direction (1 → recto -1 → verso).
- pan (-1 to 1).
- trigger (without sustain).
- doneAction (voice allocation type).

```
SynthDef(\smp, {arg buf=0, pos=0, dur=0.2, amp=0, trsp=0, dir=1, pan=0,
                    t_gate=0,done=2;
                var sig, env;
                    sig = PlayBuf.ar(1, buf,
                                     BufRateScale.kr(buf)*trsp.midiratio*dir,
                                     t_gate,
              BufSampleRate.kr(buf)* pos); // Absolute start position (frames)
          // BufFrames.kr(buf) * pos);    // Relative start position (0 to 1)
                    env = Env.linen(0.01,dur-0.02,0.01);
                    env = EnvGen.kr(env, t_gate, doneAction:done);
                    sig = Pan2.ar(sig * env * amp, pan);
                Out.ar(0,sig)
```

```
                }).add;
```

Listing 3.3: Sampler model

Test it.

```
Buffer.freeAll;
b = Buffer.read(s, "/absolute/path/to/file.wav");

Synth(\smp,[\buf, b,
            \pos, 2.3,
            \dur, 0.2,
            \amp, 1,
            \trsp, 0,
            \dir, 1,
            \pan, 0,
            \done, 2,
            \t_gate, 1])
```

Sequencing

We can define a sequence of parameter changes in the form of an Array and then dynamically recall it over time.

Next exemple has to do with *libero arbitrio*: we can set all the parameters as we want.

```
~tstr = 4.01; // Time stratching factor

~pos = [5.82, 9.11, 9.76, 0.39, 9.21, 2.85, 2.31, 0.05];
~amp = [0.40, 0.65, 0.85, 1.00, 0.70, 0.50, 0.30, 0.10];
~dur = [1.40, 0.46, 1.18, 2.39, 0.69, 1.15, 3.41, 1.55]* ~tstr;
~trs = [1.40, 0.46, 2.18,-2.39, 3.69, 2.15,-3.41, 1.55];
~del = [1.48, 0.54, 2.41, 1.26, 3.07, 0.89, 1.16, 0.89]* ~tstr;
~dir = [1,      -1,   -1,    1,    1,   -1,    1,   1 ];
~pan = [-1,   -0.8, -0.6, -0.4,    0,  0.4,  0.6,   1 ];

~evt = ~del.size.postln; // Number of sonic events (notes)

r = Routine({ ~evt.do({arg id;
                        Synth(\smp,[\buf,b,
                                    \pos, ~pos[id], // index call
                                    \amp, ~amp[id],
                                    \dur, ~dur[id],
                                    \trsp,~trs[id],
                                    \dir,~dir[id],
                                    \pan,~pan[id],
                                    \done,2,
                                    \t_gate,1]);
                    ~del[id].wait })
             }).play
```

Listing 3.4: Deterministic sequencing

Next example has to do with setting up deterministic rules.

The rule is cutting the soundfile in a defined number of cues and play it back with a defined time stratching.

```
~nfrag = 80;                // choose a number of cues
~dur   = b.duration/~nfrag; // compute fragment duration
~pos   = 0;                 // start position
~stch  = 0; // if > 0 = rest elif < 0 = superimposion (poliphony)
```

```
r = Routine({
            ~nfrag.do({
                    Synth(\smp,[\buf,b,
                                \pos, ~pos.postln,
                                \dur, ~dur,
                                \amp, 1, \trsp, 0, \dir, 1, \pan, 0,
                                \done, 2, \t_gate, 1 ]);
                    ~pos = ~pos+~dur; // position increases at each step
                    (~dur+~stch).wait
                  })
            }).play
```

Enveloping

This technique consists in modifying the type of amplitude envelope of the original fragment.

We can do it in two ways:

- uniform → one envelope type for all sonic events (grain).
  Here a percussive envelope.

```
SynthDef(\grain, {arg buf=0, pos=0, dur=0.2, amp=0, trsp=0, dir=1, pan=0,
                     t_gate=0, atk=0.1, done=2;
                   var sig,env;
                       sig = PlayBuf.ar(1, buf,
                                BufRateScale.kr(buf) * trsp.midiratio * dir,
                                t_gate,
                                BufSampleRate.kr(buf)* pos );
                      env = Env.perc(atk, dur-atk);
                      env = EnvGen.kr(env,t_gate,doneAction:done);
                      sig = Pan2.ar(sig * env * amp, pan);
                 Out.ar(0,sig)
                 }).add;
```

Listing 3.5: Percussive grain model

We can change dynamically attack time (0.01 to 0.99).

```
~durs = [0.79,0.68,0.57,0.46,0.25,0.14,0.23,0.32,0.41];
~pos  = [1.1,2.3,0.5,3.7,1.9,2.1,0.3,3.5,2.7];
~atk  = [0.01,0.02,0.43,0.04,0.35,0.06,0.57,0.08,0.89];

r = Routine({
            inf.do({arg item,id;
                        Synth(\grain, [\buf, b, \amp, 1,
                                       \pos, ~pos.foldAt(id), // foldAt()
                                       \done,2,
                                       \atk, ~atk.foldAt(id),
                                       \t_gate, 1]);
                    ~durs.foldAt(id).wait})
            }).reset.play
```

- dynamic → a choice of different envelopes from a set at each event.
  New SynthDef with envelope set dinamically by argument.

```
SynthDef(\envi,{arg buf=0,pos=0,amp=0,trsp=0,dir=1,pan=0,t_gate=0,done=2;
                 var sig,env;
                     sig  = PlayBuf.ar(1,buf,
                              BufRateScale.kr(buf) * trsp.midiratio * dir,
                              t_gate,
                              BufSampleRate.kr(buf)* pos );
                     env = Env.newClear(4); // New Envelope with 4 nodes
```

```
                    env = \env.kr(env.asArray); // read from argument
                    env = EnvGen.kr(env, t_gate, doneAction:done);
                    sig = Pan2.ar(sig * env * amp, pan);
                Out.ar(0,sig)
                }).add;
```

Listing 3.6: Dynamic envelope model

Here a sequence with different envelope at each event.

```
d = 0.2;                            // Duration
f = [Env.perc(0.1,1).duration_(d), // Different types of envelopes
     Env.triangle(d),               // (all same duration)
     Env.sine(d),
     Env.linen(0.1,1,0.7).duration_(d)
     ];

~env = [f[0],f[1],f[2],f[3],f[2],f[4],f[1],f[2],f[0],f[2],f[3],f[1]];

r = Routine({
          inf.do({arg id;
                      Synth(\envi,[\buf,b, \amp,1, \pos,1, \done,2,
                                   \env, f.wrapAt(id), // wrapAt()
                                   \t_gate, 1]);
                d.wait;
                 })
          }).reset.play;
```

3.3.3 Scratching

This second category allows us to illusinvestigate two other ways of controlling parameters:

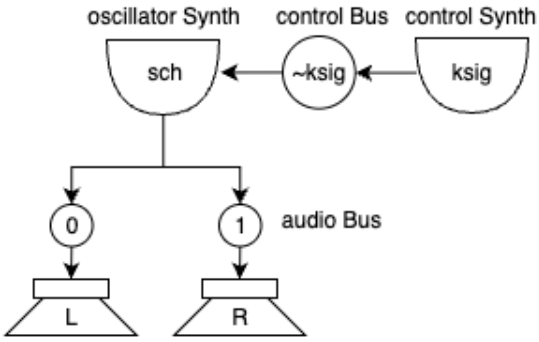
- interaction.
- control signals.

N.B. In this case the interaction is to be considered as preparation of the material to be fixed on the support not in the performance.

We need:

- one Synth that generate different types of control signals.
  - mouse x position scaled from -1 to +1.
  - mouse y position scaled from -1 to +1.
  - random signal at sub audio frequency.
- one control bus → SuperCollider have a lot of control and audio buses and we can:
  - write signal on it.
  - read signal from it.
- one Synth that play the buffer. Here the parameters we want to control:
  - buffer to play
  - pointer signal (dynamic movement of position (phase) in the Buffer)
  - smoothing time factor (seconds) - interpolation time
  - amplitude (0 to 1)
  - pan (-1 to 1)

Here a simple flow diagram:



Let's define a SynthDef that generate different types of control signals and a control Bus:

- 0 → MouseX position.
- 1 → MouseY position.
- 2 → random values at a determined frequency.

For each we can set the range (min, max).

```
SynthDef(\ksig,{arg type=0, range=#[0,1], freq=1, busOut=0;
                var sig;
                    sig = Select.kr(type,
                                [MouseX.kr(range[0], range[1]),
                                 MouseY.kr(range[0], range[1]),
                                 LFNoise1.kr(freq).range(range[0], range[1])
                                 ]);
                    Out.kr(busOut,sig)
                    }).add;

x = Bus.control(s, 1);   // a control bus (server, n_canali)
```

Listing 3.7: Control signals model

The SynthDef

```
SynthDef(\sch, {arg buf=0, busOut=0, smooth=0.02, amp=0, pos=0, pan=0;
                var pnt, sig;
                    pnt = Lag.kr(pos.linlin(0,1,0,BufFrames.kr(buf)),smooth);
                    sig = BufRd.ar(1, buf, K2A.ar(pnt));
                    sig = LeakDC.ar(sig);
                    sig = Pan2.ar(sig * amp, pan);
                Out.ar(busOut, sig)
                }).add;
```

Listing 3.8: Scratcher model

The Synths.

```
Buffer.freeAll;
b = Buffer.read(s,"/absolute/path/to/file.wav");

k = Synth(\ksig, [\type,0, \range, [0, 1], \busOut, x]);   // Synth per ksig
a = Synth(\sch, [\buf,b, \amp,1, \pos, x.asMap, \smooth,0.02, \pan,0]);
```

Change the smoothing time.

```
a.set(\smooth,0.2);
```

Apply a random control signal and change its sub audio freuency.

```
p.set(\type, 2, \freq, 2);
```

3.3.4 Structures, DAW and scoring

In the previous paragraph we saw some tools for generating the sound palette with which to create the composition.

Now we need to string them together into a formal structure.

Two options:

- record the musical objects into soundfiles and edit them in a DAW.
  1. create the soundfile.

```
s.prepareForRecord("/absolute/path/to/prova.wav", 2); // path,
    channels
```

  2. start recording and sound.

```
s.record;
a = Synth(\ksig, [\buf,b, \amp,1, \pan,0, \freq, 3, \gate, 1])
```

  3. fade out and stop recording.

```
r = Routine({
            a.set(\fOut, 2, \gate,0); // Trig fade out
            3.wait;                   // Wait for fadeout end
            s.stopRecording;          // Stop recording
            }).play
```

Listing 3.9: Recording sound

  You can now open it in a DAW and mix with others in a structure.
- compose the entire structure directly in SuperCollider (scoring).
  1. define all the SynthDefs.

```
SynthDef(\grain, {arg buf=0, pos=0, dur=0.2, amp=0, trsp=0, dir=1,
                     pan=0, t_gate=0, atk=0.1, done=2;
                 var sig,env;
                     sig = PlayBuf.ar(1, buf,
                           BufRateScale.kr(buf) * trsp.midiratio
                                                * dir,
                           t_gate,
                           BufSampleRate.kr(buf)* pos );
                     env = Env.perc(atk, dur-atk);
                     env = EnvGen.kr(env,t_gate,doneAction:done);
                     sig = Pan2.ar(sig * env * amp, pan);
                 Out.ar(0,sig)
                 }).add;

SynthDef(\ksig,{arg buf=0, amp=0,frq=0.1,fIn=0.2,fOut=0.2,pan=0,
                    gate=0,done=2;
                var punta,sig,fade;
                    punta = LFCub.ar(frq).range(0,BufFrames.kr(buf));
                    sig   = BufRd.ar(1, buf, K2A.ar(punta));
                    fade  = Linen.kr(gate,fIn,1,fOut,done);
                    sig   = Pan2.ar(sig*amp.lag(0.2)*fade,pan);
                Out.ar(0, sig);
        }).add;
```

  2. define the different sound objects within Routines or Synths or functions.

```
Buffer.freeAll;
b = Buffer.read(s, "/absolute/path/to/file.wav");

~durs = [0.9,0.8,0.7,0.6,0.5,0.4,0.3,0.2,0.1];
~pos  = [0.1,0.3,0.5,0.7,0.9,1.1,1.3,1.5,1.7];
~atk  = [0.01,0.02,0.43,0.04,0.35,0.06,0.57,0.08,0.89];
~pan  = 1;
~obj1 = Routine({
                inf.do({arg id;
                        Synth(\grain, [\buf, b,
                                       \amp, 0.4,
                                       \pos, ~pos.foldAt(id),
                                       \done, 2,
                                       \atk, ~atk.foldAt(id),
                                       \t_gate, 1]);
                        ~durs.foldAt(id).wait;
                        });
                });

~amps = [0.01, 0.4, 0.1, 0.6];
~pos  = [0.5, 1.4, 3.1, 2.6, 0.5, 2.8];
~obj2 = Routine({
                inf.do({arg id;
                        ~pan = ~pan * -1;        // flip flop
                        Synth(\grain, [\buf, b,
                                       \amp, ~amps.foldAt(id),
                                       \dur, 0.2,
                                       \pos, ~amps.wrapAt(id),
                                       \done, 2,
                                       \atk, 0.01,
                                       \pan, ~pan,
                                       \t_gate, 1]);
                    0.3.wait;
                        });
                });
```

3. arrange it in a score (we can specify the temporal events or bars as comments).

```
~score = Routine({
                  2.do({                    // Ritornello (loop)
                        ~obj2.reset.play(t);         // 0 Start obj2
                        3.wait;
                        3.do({                       // 3 Ritornello
                              ~obj1.reset.play(t);   // Start obj1
                              4.wait;
                              ~obj1.stop;            // Stop obj1
                              1.wait});
                        ~obj3 = Synth(\ksig,[\buf,b, // 18 Start obj3
                                             \amp,0.1,
                                             \fIn,3,
                                             \frq,2,
                                             \gate,1]);
                        2.wait;
                        ~obj2.stop;                // 20 Stop obj1
                        1.wait;
                        ~obj1.reset.play(t);       // 21 Start obj2
                        ~obj3.set(\fOut, 4, \gate, 0); // Fout obj3
                        3.wait;
                        ~obj1.stop;                // Stop obj1
                        1.wait;                    // 25 End obj2
                        })
                })
```

4. play the score.

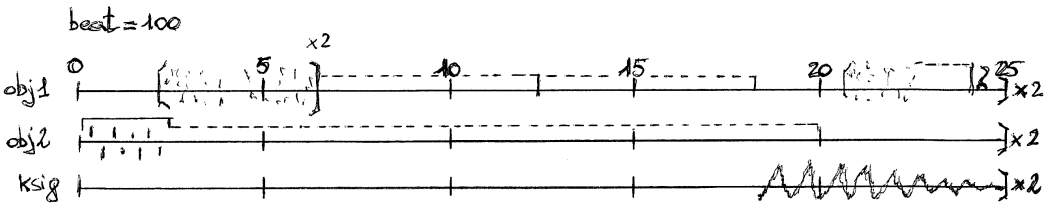
```
~bpm = 100;
t    = TempoClock.new(~bpm/60); // Set bpm tempo

~score.reset.play(t);
```

Listing 3.10: Scoring

Through this modular procedure, we can define the structure of:

1. the entire piece.
2. individual tracks.
3. sections and fragments.

We can represent it in a graphic score.



Link to an example of a graphic score of a historical piece by J.C.Risset - Sud (1984-85).

3.3.5 Composition sketches proposal

Sound palette

1. design and create a sound palette starting from a short soundfile (2-10 seconds).
2. group the different sets of musical objects made by morphological characteristics.

Fixed media work or sound design

Two different strategies:

- bottom to top.

  1. choose one to three short sound texts (soundfiles).
  2. analyze the morphological structure of the sound files (microform)
  3. starting from a musical object begin to compose the general structure (macroform) step by step in a DAW or through scoring techniques. At each step creating new musical objects and their development into microstructures. Idea of the macroform will not be clear until the end of the process.

  Macroform develops from microstructure.

- Top to bottom.

  1. define a global formal structure in a graphical sketch.
  2. design and create a sound palette based on the types of sounds described in the sketch according to the principles set out in the chapter (rules or *libero arbitrio*).
  3. realize the sketch in a DAW or through scoring techniques.

  Microstructures fill the macroform.

Chapter 4

Computer music

4.1 Framework and historical context

First experiments in sound synthesis belong to the analogue era.

In the 1950s the study of the Cologne Radio (WDR) was in contrast to the GRM both in:

- terms of sound generation and manipulation techniques.
- terms of aesthetic positions.

The reference figure in this study was the composer Karheinz Stockhausen.

At that time his compositional ideas brought the strict rules of serialism into the sound synthesis.

Every parameter of sound (pitch, spectrum, intensity, envelope, duration, etc.) was governed by rigid numerical rules.

The electroacoustic instruments available in the Cologne studio (oscillators, filters and ring modulators) and the possibility of recording the resulting sound on magnetic tape were the ideal tools for developing these ideas.

Karlheinz Stockhausen - Elektronische Studie II for tape (1954) - extract.

Still in the electroacoustic field a mix of the technical equipment of the two studios was found in the RAI Musical Phonology Studio in Milan where the composers of reference were Luciano Berio, Bruno Maderna and later Luigi Nono.

Their aesthetic positions were also less rigid than those of their French and German colleagues, mixing together different compositional techniques.

Bruno Maderna - Musica su due dimensioni for flute and tape (1958) - extract.

Computer music was born later for reasons obviously linked to the birth and development of digital technologies but it takes its cue from these (and other) electroacoustic experiences.

The first computer used for playing music was the CSIR Mk1 developed in Sydney in the late 1940s and presented at a public concert in 1951 where it performed Colonel Bogey March.

Alan Turing was also working on primeval computer music in the summer of 1951 at the Computing Machine Laboratory in Manchester.

He and his collaborators produced three short melodic sequences played by the computer.

In the late 1950s Max Mathews was the most important computer scientist working at Bell Telephone Laboratories in New Jersey.

He worked primarily with mainframe computers made by IBM on several series of their MUSIC X program, a musical programming language that was upgraded over the next twenty years.

An early piece created in these studios was Beat Canon (1960) by J.R.Pierce (director of studio).

In 1963 a human voice was synthesized for the first time (speech synthesis).

Later musicians and engineers alike began to collaborate more on the making of music with computers creating more sophisticated work.

Aa example an exrept from Lyric Variations for Violin and Computer (1965-1968) by J.K.Randall - extract.

In USA starting from the 60s artistic currents began to develop linked to the Studios and cultural environment where they arose.

- west coast → *Experimantal Music Studio MIT Media Laboratory* - Boston.
  Founded by Barry Vercoe in 1971, it develops software derived from the Music IV series on IBM 360 computers.
  In this studio Curtis Road created Computer Music Journal the world's leading computer music magazine.
  Composers such as James Dashow, Miller Puckette, Charles Dodge worked there, whose approach to composition was characterised by a strong structural rigor derived from the ideas developed by Stockhousen in Cologne.
  Three exemples:
  - James Dashow - Conditional Assemblies for computer generated sounds (1980)
  - Charles Dodge - The Waves for voice and electronics (1984)
  - Curtis Road - Granule for computer generated sounds (1999)
- california two main poles:
  - Stanford University CCRMA (Karma) → music production.
  - San Diego University (CARU) and Lucas film studios → software and hardware research.

  The CCRMA was founded and directed by John Chowning, the inventor of FM synthesis and a leading figure in computer music.
  A figure of synthesis between technology and music, he addressed in each of his compositions a specific problem linked both to the synthesis of sound and to musical language.
  Through his work, which also took place in important European production centers such as IRCAM in Paris, he created a bridge between the two continents.
  An audio extract from his work Turenas for computer generated sounds (1972) and a paper about it.

Here we can find a comprehensive list of computer music research and production centers.

4.2 Randomness

Following or pursuing an random procedure means generating or transforming musical or acoustic material by integrating random choices into the decision-making process.

The concept of indeterminism affirms the denial of any determination of the will by external motives.

In our case a sequence of sonic events must be governed in whole or in part by chance or by a set of probabilities, regardless of the author's experience, taste, knowledge, and personality.

This makes it inevitable to rethink the composer's role in the creative act.

A reflection of this kind plays an important role in the poetics of American composer J. Cage who has often stated that his compositional principle is a departure from unity and a movement toward multiplicity:

[...] to abdicate one's subjectivity, to abandon one's own judgment to move toward the multiplicity that is life itself, of which man is but a small component, in the sea of constant variation [...]

In Western musical tradition three different indeterministic procedures have been theorized and used.

4.2.1 Chance

The musical or sonic materials of the piece are:

- generated through random processes.
- fixed in symbolic form to allow a performance where the random elements fall within the limits of deterministic musical interpretation.

Two examples:

- the musical games in Europe between the second half of the 18th century and the beginning of the 19th century (C.Ph.E. Bach, F.J. Haydn, W.A. Mozart, etc.).
  - a set of musical bars that follow a harmonic scheme indexed and arranged by the composer in any order.
  - use of a pair of dice to determine the order of performance (recomposition).
  - the game allowed anyone even those with no musical knowledge to create harmonious and original pieces.

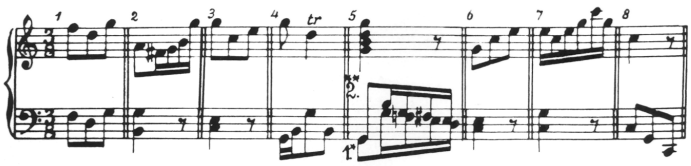
Part One

	I	II	III	IV	V	VI	VII	VIII
2	96	22	141	41	105	122	11	30
3	32	6	128	63	146	46	134	81
4	69	95	158	13	153	55	110	24
5	40	17	113	85	161	2	159	100
6	148	74	163	45	80	97	36	107
7	104	157	27	167	154	68	118	91
8	152	60	171	53	99	133	21	127
9	119	84	114	50	140	86	169	94
10	98	142	42	156	75	129	62	123
11	3	87	165	61	135	47	147	33
12	54	130	10	103	28	37	106	5

Part Two

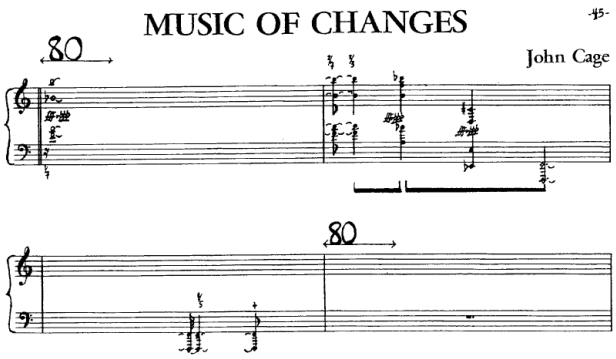
	I	II	III	IV	V	VI	VII	VIII
2	70	121	26	9	112	49	109	14
3	117	39	126	56	174	18	116	83
4	66	139	15	132	73	58	145	79
5	90	176	7	34	67	160	52	170
6	25	143	64	125	76	136	1	93
7	138	71	150	29	101	162	23	151
8	16	155	57	175	43	168	89	172
9	120	88	48	166	51	115	72	111
10	65	77	19	82	137	38	149	8
11	102	4	31	164	144	59	173	78
12	35	20	108	92	12	124	44	131

Table of Measures



- music of Changes by J.Cage for piano (1951) in which the formal structure, compositional method and choice of musical materials (pitches, dynamics and tempos) are determined by the roll of the dice of the I-Ching.
  - the collection consists of eight *books* of music.
  - to compile the score Cage used 8x8 grids (two-dimensional matrices) to facilitate correspondence with the 64 hexagrams of the I Ching.
  - a first roll of the dice determines which sound event to write among those included in the sound grid, after which the next two rolls are dedicated to durations and

dynamics.



4.2.2 Indeterminacy

It seeks the variable multiplicity of possible performances not chance.

Indeterminacy is the opposite of chance which adopts a method.

Chance relies on the logic of the method.

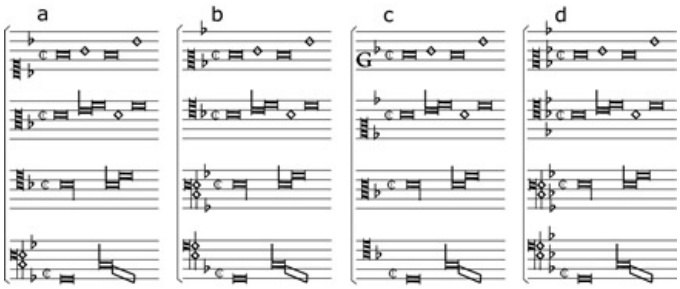
Indeterminacy relies on the performer's sensitivity.

It excludes the precise determination of certain parameters (pitch, duration, intensity, timbre) to give the performer a wide margin of freedom.

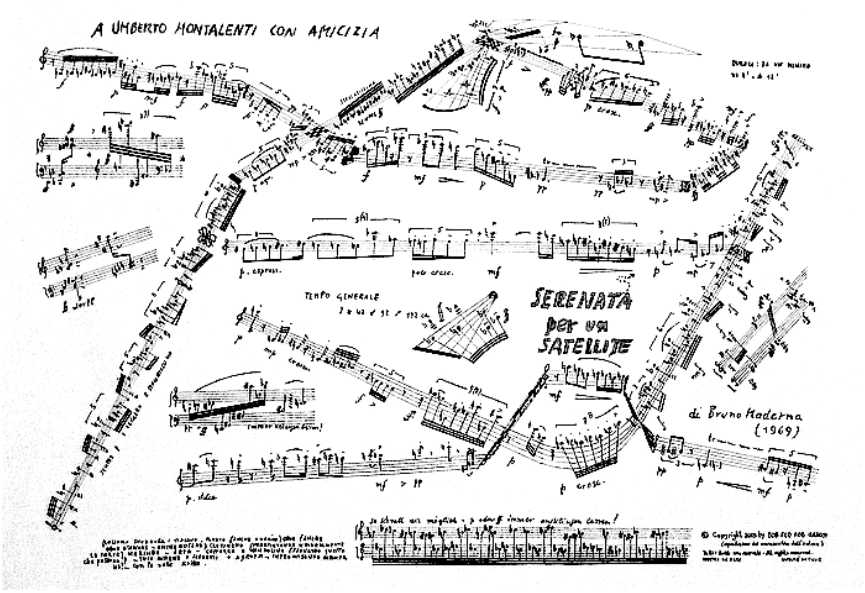
When taken to its extreme limits it ideally coincides with the opposite deterministic extreme: total improvisation or *libero arbitrio*.

The concept of indeterminacy in music can be developed in two different ways:

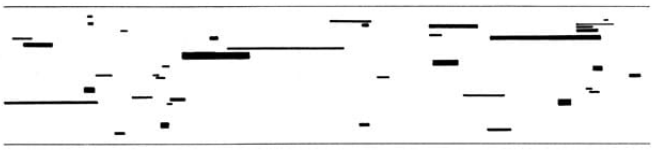
1. free choice of performance mode or form
   - Missa Cuiusvis Toni (1539) by Johannes Ockeghem.
     - the composer did not specify the clefs at the beginning of the staff.
     - it can be performed in all four modes (Dorian, Phrygian, Lydian, Mixolydian).
     - the choice of which clef to use for reading (and therefore the mode) is left to the performers.



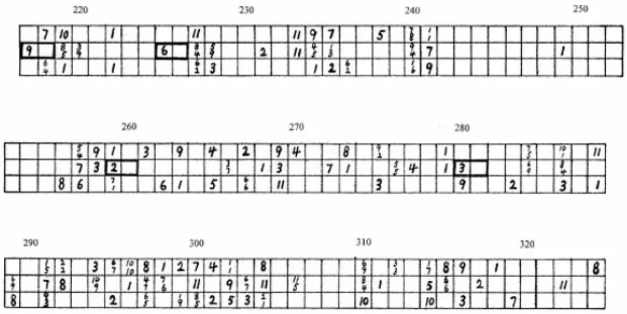
   - Serenata per un satellite (1969) by B. Maderna
     - It can be played by violin, flute (including piccolo), oboe (including oboe d'amore, even musette), clarinet (transposing the part, of course), marimba, harp, guitar, and mandolin (playing what they can), all together, individually, or in groups, improvising, in short, with the written notes.
     - the whole piece should last between 4 and 12 minutes.



2. graphic scores or verbal indications that suggest to the performer how the piece can be performed.
   - 4 Systems by Earle Brown.



   - Intersection 3 by Morton Feldman (1964).



   - Intuitive Musik (1968) by K.Stockhausen

DOWNWARDS

play a vibration in the rhythm of your limbs
play a vibration in the rhythm of your cells
play a vibration in the rhythm of your molecules
play a vibration in the rhythm of your atoms
play a vibration in the rhythm of your smallest particles
which your inner ear can reach

change slowly from one rhythm to another
until you become freer
and can interchange them at will

   - expressive need - Bruno Maderna at Venice Biennale

4.2.3 Alea

Aleatoric strategies became common among composers of the European avant-garde.

Acousticist Werner Meyer-Eppler during the Darmstadt summer courses defined it this way:

A process can be defined as "aleatory" [...] if its development is established in general terms but depends on chance in the details.

Let's explore random procedures in SuperCollider using a SynthDef from previous chapter.

```
s.boot;
s.meter;
s.plotTree;
```

A SynthDef to test processes.

```
SynthDef(\smp, {arg buf=0,pos=0,dur=0.2,amp=0,trsp=0,dir=1,pan=0,
                    t_gate=0,done=2;
                var sig,env;
                    sig = PlayBuf.ar(1, buf,
                                     BufRateScale.kr(buf) * trsp.midiratio
                                                          * dir, t_gate,
                                     BufSampleRate.kr(buf)* pos);
                    env = Env.linen(0.01,dur-0.02,0.01);
                    env = EnvGen.kr(env, t_gate, doneAction:done);
                    sig = Pan2.ar(sig * env * amp, pan);
                Out.ar(0,sig)
        }).add;
```

We can obtain random numbers in several ways, here the main objects.

- boolean → 0.5.coin (0.0 = false, 1.0 = true, 0.x = probability).
  For example we can use it in a control structure to define the percentage of rests in a sequence.

```
Buffer.freeAll;
b = Buffer.read(s, "/absolute/path/to/file.wav");

~rest = 0.8;       // probability of rests in sequence

r = Routine({
            20.do({
                  if(0.8.coin==true)
                        {Synth(\smp,[\buf,b, \pos,0.23, \amp,1, \dur,0.08,
                                     \pan,0, \t_gate, 1])};
                  0.1.wait;
                  })
                 }).reset.play;
```

Listing 4.1: coin(0.5)

- within a range (linear distribution) → rrand(min, max).
  For example we can use it to define random positions in a Buffer.

```
Buffer.freeAll;
b = Buffer.read(s, "/absolute/path/to/file.wav");
{b.duration.postln}.defer(1);  // range max

~rest = 0.8;         // probability of rests in sequence
~posi = [0.5,3.4];   // start-end position points

r = Routine({
```

```
            20.do({
                if(0.8.coin==true)
                  {Synth(\smp,[\buf,b,
                              \pos, rrand(~posi[0], ~posi[1]),
                              \amp, 1, \pan,0, \dur,0.08, \t_gate,1])};
                  0.1.wait})
            }).reset.play;
```

Listing 4.2: rrand(min max)

- between 0 and max (linear distribution) → rand(max).
  For example we can use it to define random amplitude.

```
Buffer.freeAll;
b = Buffer.read(s, "/absolute/path/to/file.wav");

~rest = 0.8;        // probability of rests in sequence
~posi = [0.5,3.4]; // start-end position points

r = Routine({
            20.do({
                  if(0.8.coin==true)
                   {Synth(\smp,[\buf,b,
                                \pos, rrand(~posi[0], ~posi[1]),
                                \amp, rand(1.0),
                                \pan, 0, \dur,0.08, \t_gate,1])};
                  0.1.wait})
            }).reset.play;
```

Listing 4.3: rand(max)

- bipolar (linear distribution) → rand2(+/- max)).
  For example we can use it to define random pan.

```
Buffer.freeAll;
b = Buffer.read(s, "/absolute/path/to/file.wav");

~rest = 0.8;        // probability of rests in sequence
~posi = [0.5,3.4]; // start and end position points

r = Routine({
            20.do({
                  if(0.8.coin==true)
                   {Synth(\smp,[\buf,b,
                                \pos, rrand(~posi[0], ~posi[1]),
                                \amp, rand(1.0),
                                \pan, rand2(1.0),
                                \dur, 0.08, \t_gate,1])};
                  0.1.wait})
            }).reset.play;
```

Listing 4.4: rand2(+/-max)

- choose in a finite set with a linear distribution → [3,5,67,89].choose.
  For example we can use it to define random choice among a set of durations.

```
Buffer.freeAll;
b = Buffer.read(s, "/absolute/path/to/file.wav");

~rest = 0.8;                  // probability of rests in sequence
~posi = [0.5,3.4];            // start and end position points
~durs = [0.08,0.04,0.2,0.5]; // duration set

r = Routine({
```

```
            20.do({
                    if(0.8.coin==true)
                     {Synth(\smp,[\buf,b,
                                  \pos, rrand(~posi[0], ~posi[1]),
                                  \amp, rand(1.0),
                                  \dur, ~durs.choose,
                                  \pan, rand2(1.0),
                                  \t_gate, 1]
                    )};
                    0.1.wait;
                    })
            }).reset.play;
```

Listing 4.5: choose(Array)

There are other objects dedicated to nonlinear distributions but the principles on which they are based are the same as those just illustrated.

4.3 Sound synthesis

In this paragraph we will not delve into all the aspects of the individual synthesis techniques.

Only their main sonic and expressive possibilities related to composing music are focused.

Before starting let's define some utility:

- one SynthDef → set of source signals.
  - 0 → Live input.
  - 1 → WhiteNoise.
  - 2 → Impulse.
  - 3 → Dust.
  - 4 → Sawtooth wave.
  - 5 → Soundfiles player.

```
SynthDef(\srcs,{arg type=0, freq=400, amp=0, buf=b, busOut=2, gate=0;
                var sig,env;
                    sig =  Select.ar(type,
                                    [SoundIn.ar(0),              // 0
                                     WhiteNoise.ar,              // 1
                                     Impulse.ar(freq),           // 2
                                     Dust.ar(freq),              // 3
                                     Saw.ar(freq),               // 4
                                     PlayBuf.ar(1, buf, loop:1) // 5
                                     ]);
                  env = Linen.kr(gate,doneAction:2);
                  sig = sig * env * amp;
              Out.ar(busOut, sig)
        }).add;
```

Listing 4.6: Source signals model

- one SynthDef → set of control signals (mouse, continuos, impulsive).

```
SynthDef(\ksig,{arg type=0, range=#[0,1], freq=1, busOut=0;
              var sig;
                  sig = Select.kr(type,
                                [MouseX.kr(range[0], range[1]), // 0
                                 MouseY.kr(range[0], range[1]), // 1
                                 LFNoise1.kr(freq).range(range[0],
                                                         range[1]), // 2
                                 LFSaw.kr(freq).range(range[0],
```

```
                                                        range[1]),// 3
                                  Impulse.kr(freq),                // 4
                                  Dust.kr(freq),                   // 5
                                 ]);
                Out.kr(busOut,sig)
          }).add;
```

- one spectroscope.

```
s.freqscope;        // a spectroscope
```

4.3.1 Additive

With additive synthesis techniques we can realize:

- harmonic spectra (pitched sounds).
- inharmonic spectra (non-pitched sounds).

Both categories can have:

- fixed spectrum.
- variable spectrum.
- spectral envelope.

There are different ways to do that.

Wavetables

Harmonic spectra.

Periodic noises.

Fixed spectrum.

Procedure:

1. draw a single waveform period.
2. store it in a Buffer.
3. play it back at a defined frequency by an oscillator.

```
SynthDef(\wts, {arg buf=b, freq=500, amp=0, dur=1, t_gate=0, pan=0, done=2;
                 var sig,bpf,env;
                     sig = Osc.ar(buf,freq);
                     env = Env.triangle(dur); // change envelope and list
                     env = EnvGen.ar(env,t_gate,doneAction:done);
                     sig = Pan2.ar(sig*amp*env,pan);
               Out.ar(0,sig)
          }).add;
```

Listing 4.7: Wawtable lookup synthesis model

We can play it as in previous exemples.

- deterministic.

```
Buffer.freeAll;
b = Buffer.alloc(s, 2048);
// Draw Partial_number, magnitudes
b.sine2([1,3,4,6,8,9],[20,30,10,6,7,3].normalizeSum);
b.plot;
~pitch = [60,64,65,68,72,73,68,67,60].midicps;
~amps  = [0.3,0.5,0.7,0.3,0.9,0.1]*0.2;
```

```
~pans  = [-1,0,1];
~durs  = [0.1,0.5,0.2,0.8,0.4];
~delta = [0.1,0.3,0.1,0.2];

r = Routine({
              20.do({arg i;
                     Synth(\wts,[\buf,b,
                           \freq, ~pitch.wrapAt(i),
                           \amp,  ~amps.foldAt(i),
                           \dur,  ~durs.wrapAt(i),
                           \pan, ~pans.foldAt(i),
                           \done,2,
                           \t_gate,1]);
                     ~delta.foldAt(i).wait;
                     })
              }).reset.play;
```

- random.

```
v = Routine({
            inf.do({Synth(\wts,[\buf,b,
                        //  \freq,rrand(50,100).midicps,          // Atonal
                            \freq, [68,72,75,80].midicps.choose, // Triade
                            \amp,rand(0.2),
                            \dur,rrand(4,8),
                            \fade,rrand(0.2,8),
                            \pan, rand2(1.0),
                            \done,2,
                            \t_gate,1]);
                rrand(0.1,2).wait;
                })
             }).reset.play
```

Vectorial

This technique is a variant of wavetable lookup synthesis.

Harmonic spectra.

Periodic noises.

Variable spectrum.

1. define an array of different waveforms.

```
Buffer.freeAll;
b = Buffer.allocConsecutive(4,            // Number of Buffer
                            s, 1024);
b[0].sine2([1,3,5,7], [1,0.3,0.5,0.7]); // Fill the buffers
b[1].sine2([2,4,6,8], [0.2,0.4,1.3,0.6]);
b[2].sine2([2,3,10,12,13,14], [1,0.2,0.5,0.6,1.3,0.5]);
b[3].sine2([1,3,4,7,8,9,13,18], [1.3,0.3,0.5,0.7,0.8,0.9,0.2,0.3]);
```

2. during playback interpolate the reading dynamically with a control parameter.

```
SynthDef(\vec, {arg bufn=4,freq=400,amp=0,pos=0,pan=0,gate=0,done=2;
                 var sig,env;
                     sig  = VOsc.ar(pos.linlin(-1,1,0,bufn), freq);
                     env  = Linen.kr(gate,doneAction:done);
                     sig  = Pan2.ar(sig*amp*env,pan);
                 Out.ar(0, sig)
                 }).add;
```

Listing 4.8: Vetorial synthesis model

As usual we can change timbre parameter in four different ways.

1. by sending single values (o to number of buses - 1).

```
a = Synth(\vec, [\bufn,b.size-1, \freq,400, \amp,0.5, \pan,0, \gate,1]);
r = Routine({
              5.do({
                    rrand(2,20).do({
                                    a.set(\pos,rand2(1.0));
                                    0.1.wait});
                    rrand(0.2,1).wait});
                    a.free;
              }).reset.play;
```

2. by mapping a control signal on a control bus as argument.

```
k = Synth(\ksig, [\type,0, \range,[-1, 1], \busOut, x]); // Mouse x
a = Synth(\vec,  [\bufn,b.size-1, \freq,400, \pos, x.asMap,
                  \amp,0.5, \pan,0, \gate,1]);
```

Change the control signal type and sub-audio frequency.

```
k.set(\type,2, \freq,1);
```

Another classic technique in digital synth design is to map the amplitude envelope with spectral modification.

A simulation of acoustic natural sounds.

```
SynthDef(\vmap, {arg buf=0,bufn=4,freq=400,dur=1,amp=0,pan=0,t_gate=0,done=2;
                  var sig,bpf,env;
                      bpf   = Env.sine(dur);
                      env   = EnvGen.ar(bpf,t_gate,doneAction:done);
                      sig   = VOsc.ar(env.linlin(0,1,buf,bufn-1), freq);
                      sig   = Pan2.ar(sig*amp*env,pan);
                  Out.ar(0, sig)
         }).add;
```

Listing 4.9: Envelope mapping parameter

Play it.

```
r = Routine({
              10.do({
                     Synth(\vmap, [\buf,b, \bufn,3,
                                   \freq,rrand(40, 76).midicps,
                                   \dur,rrand(4,12),
                                   \amp,rand(1.0),
                                   \pan,rand2(1.0),
                                   \t_gate,1]);
                     rrand(3,5).wait;
                     })
              }).reset.play;
```

PWM

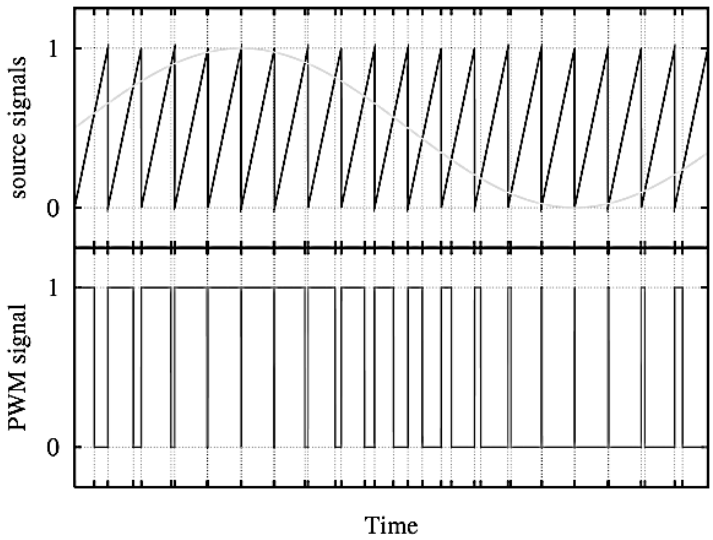
Pulse Width Modulation.

This technique is a variant of wavetable lookup synthesis and allows the generation of variable spectra.

Harmonic spectra.

Variable spectrum.

It is based on the concept of duty cycle.



By changing this parameter we modify the timbre.

```
SynthDef(\pwm, {arg freq=300,duty=0.5,amp=0,gate=0,pan=0,done=2;
                var sig, env;
                    sig  = Pulse.ar(freq, duty);
                    env  = Linen.kr(gate,doneAction:done);
                    sig  = Pan2.ar(sig * amp * env,pan);
                Out.ar(0,sig)
        }).add;
```

Listing 4.10: Pulse Width Modulation model

The Synth.

```
a = Synth(\pwm, [\amp,0.5,\duty,1/2,\gate,1]);
```

We can change the duty cycle from 0.0 to 0.5.

Let's observe how the spectrum changes by set 1/3, 1/4, 1/5 etc.

```
a.set(\duty, 1/4);
```

Duty cycle and other parameter control methods are the same as those illustrated previously.

Some variations of this technique.

Blip → classic pulse train derived from MusicN languages (buzz) implemented in commercial synthesizers of the 90s.

We can define the number of harmonics as an argument.

```
SynthDef(\blip,{arg freq=300,nharm=20,amp=0,gate=0,pan=0,done=2;
                var sig,env;
                    sig   = Blip.ar(freq, nharm);
                    env   = Linen.kr(gate,doneAction:done);
                    sig   = Pan2.ar(sig*amp*env,pan);
                Out.ar(0,sig)
        }).add;
```

Listing 4.11: Blip model

The sequence.

```
a = Synth(\blip, [\amp,0.5,\gate,1]);
r = Routine({
              10.do({
                     rrand(2,10).do({
                                    a.set(\freq, [50,57,62].midicps
                                                          .choose
                                          \nharm,rrand(1,30));
                                    0.1.wait;
                                    });
                     0.2.wait;
                     });
              a.set(\gate,0);
              }).reset.play;
```

Control signal.

```
k = Synth(\ksig, [\type,2, \range, [1, 20], \freq,1, \busOut, x]);
a = Synth(\blip, [\amp,0.5, \nharm, x.asMap, \gate,1])
```

We can also realize PWM with different waveforms stored in a Buffer (timbre stretching).

```
SynthDef(\wsth,{arg buf=b, freq=500, sfact=1, amp=0,gate=0,pan=0,done=0;
                var p,f,c,sig,env;
                    p   = BufFrames.kr(buf);
                    f   = sfact;
                    c   = LFSaw.ar(freq) *  0.5 * p * f + (p * 0.5);
                    sig = BufRd.ar(1,buf,c,1,4);
                    env = Linen.kr(gate,doneAction:done);
                    sig = Pan2.ar(sig*amp*env,pan);
                Out.ar(0,sig)
         }).add;
```

Listing 4.12: Timbre stretching model

The Synths.

```
Buffer.freeAll;
b = Buffer.alloc(s, 2048);
b.sine2([1,3,4,6,8,9],                  // Draw Partial_number
        [20,30,10,6,7,3].normalizeSum, // magnitudes
        asWavetable: false);

k = Synth(\ksig, [\type,0, \range, [1,10], \freq,1, \busOut, x]);
a = Synth(\wsth, [\buf,b,\freq,100,\amp,0.12,\sfact,x.asMap,\done,2,\gate,1]);
```

Classic

Harmonic spectra.

Inharmonic spectra.

Spectral envelope.

The techniques just described belong to the world of digital audio.

In the early electroacoustic music studios (Cologne and Milan), additive synthesis was achieved by recording each individual partial and then mixing it with the others.

Let's simulate this process defining a SynthDef that play only a single partial.

```
SynthDef(\par, {arg freq=500,amp=0,pha=0,dur=1,pan=0,t_gate=0;
                var sig, env;
                    sig = SinOsc.ar(freq, pha, amp);
                    env = Env.perc(0.1*dur,0.9*dur);
                    env = EnvGen.ar(env, t_gate, doneAction:2);
                    sig = sig * env;
                    sig = Pan2.ar(sig, pan);
         Out.ar(0, sig)
         }).add;
```

Listing 4.13: Partial model

Then we can define spectra as Arrays and play it through a loop structure.

```
//~freqs = [1,   3,   4,   5,    8]* 52.midicps;   // Harmonic
~freqs = [1169,1733,854, 1596, 1480];              // Inharmonic
~amps  = [10,  100,  80,   30,  171].normalizeSum; // Amplitudes (sum=1)
~dur   = 3;                                        // Overall duration
~durs  = [0.1, 0.8,  0.5, 0.3,  1.0] * ~dur;       // Partials durations

~freqs.size.do({arg i;      // loop without .wait
                Synth(\par, [\freq, ~freqs[i],
                      \amp,  ~amps.at(i),
                      \dur,  ~durs[i],
                      \t_gate,1])
                })
```

Deterministic sequence.

```
~seq = [60,  64, 67, 68, 71, 72].scramble.midicps;
~amp = [0.9,0.7,0.5,0.3,0.2,0.1].scramble;
~dur = [0.1,0.4,0.2,0.8,0.5,  1].scramble;
g    = ~seq.size;

u = Routine({var parz, pamp, pdur;
                 parz = [ 1,   3,  4,  5,  8];  // Spectrum
                 pamp = [10, 100, 80, 30,171].normalizeSum;
                 pdur = [0.1,0.8,0.5,0.3,1.0];
                 h    = parz.size;
             g.do({arg i;
                   var freq, amps, durs;
                       freq = parz * ~seq[i];
                       amps = pamp * ~amp[i];
                       durs = pdur * ~dur[i];
                       h.do({arg i;
                             Synth(\par,[\freq, freq[i],
                                         \amp,  amps[i],
                                         \dur,  durs[i],
                                         \t_gate, 1])
                             });
                       0.1.wait;
                   })
             }).reset.play;
```

Random sequence - difference between chords and spectra.

```
~bpm = 72;
t = TempoClock.new(~bpm/60);
r = Routine({
             inf.do({var nharm;
                     if(0.2.coin==false) // Random rest percentage
                     {nharm = rrand(2,9);
```

```
                        nharm.do({arg i;
                                  Synth(\par,[\freq, [70,74,75,78,82,86,90].
    midicps.choose,
                                        \amp,  rand(1/nharm),
                                        \dur,  [0.05,0.2,0.4].choose,
                                        \pan, [-1,1].choose,
                                        \t_gate,1])
                                  });
                        };
                        [0.125,0.25].choose.wait;
                       })
              }).reset.play(t);
```

4.3.2 Subtractive

Sound source ranging from white noise to rich spectra filtered by one or more filter(s).

Main types:

1. Low Pass Filter (LPF)
2. High Pass Filter (HPF)
3. Band Pass Filter (BPF)
4. Band Reject Filter (BRF)
5. Resonant Filter (Resonz)

In SuperCollider are all 2nd order Butterworth filters (-12 dB/oct).

We can control cut/central frequency (Hz).

In BPF, BRF and Resonz we can also control the bandwidth (reciprocal of Q).

- low values → narrow band.
- high values → wide band.

Sources can be chosen with *Source signals model* (4.6) already defined previously.

Filters can be defined with dedicated SynthDefs (one for each filter type).

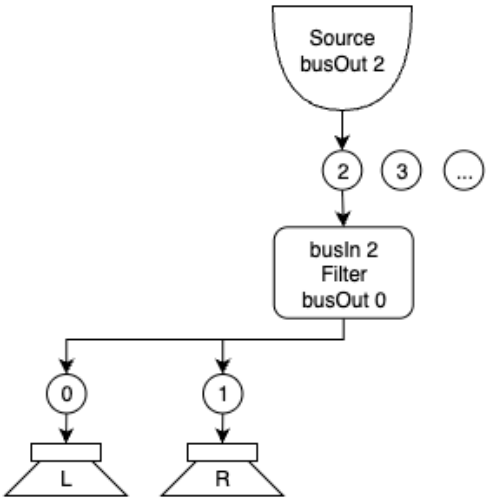
- the first we can set audio Bus out.
- the second audio Bus in.

In SuperCollider there are many audio buses.

In a stereo system:

- public buses → 0 and 1 (L and R)
- private buses → 2 and up.

A simple signal flow.




```
SynthDef(\lpf, {arg freq=500, amp=0, busIn=2, busOut=0, pan= -1;
                var sig;
                    sig = In.ar(busIn);
                    sig = LPF.ar(sig, freq) * amp;
                    sig = Pan2.ar(sig, pan);
                Out.ar(busOut, sig)
         }).add;
SynthDef(\hpf, {arg freq=500, amp=0, busIn=2, busOut=0, pan= -1;
                var sig;
                    sig = In.ar(busIn);
                    sig = HPF.ar(sig, freq) * amp;
                    sig = Pan2.ar(sig, pan);
                Out.ar(busOut, sig)
         }).add;
SynthDef(\bpf, {arg freq=500, amp=0, bw=1, busIn=2, busOut=0, pan= -1;
                var sig;
                    sig = In.ar(busIn);
                    sig = BPF.ar(sig, freq) * amp;
                    sig = Pan2.ar(sig, pan);
                Out.ar(busOut, sig)
         }).add;
SynthDef(\brf, {arg freq=500, amp=0,  bw=1, busIn=2, busOut=0, pan= -1;
                var sig;
                    sig = In.ar(busIn);
                    sig = BRF.ar(sig, freq) * amp;
                    sig = Pan2.ar(sig, pan);
                Out.ar(busOut, sig)
         }).add;
SynthDef(\res, {arg freq=500, amp=0,  bw=1, busIn=2, busOut=0, pan= -1;
                var sig;
                    sig = In.ar(busIn);
                    sig = Resonz.ar(sig, freq) * amp;
                    sig = Pan2.ar(sig, pan);
                Out.ar(busOut, sig)
         }).add;
```

Listing 4.14: Filters models

The Synths (in right order).

1. signal.
2. filter.

Here we set creation order with *addToTail* command.

```
Buffer.freeAll;
b = Buffer.read(s, "/absolute/path/to/file.wav"); // for source Synth

a = Synth(\srcs, [\type,1, \buf,b, \amp,1, \busOut,12, \gate,1]);
f = Synth(\lpf,  [\freq,1000, \amp,1, \busIn,12, \busOut,0, \pan, -1],
          s, \addToTail);
```

Change the filter cutting frequency parameter in the usual ways.

```
f.set(\freq, rrand(200,2000).postln);
```

We can sculpt the spectrum using filter banks in two combinations:

1. series → output of one filter is the input of the next until the public output.

```
Buffer.freeAll;
b = Buffer.read(s, "/absolute/path/to/file.wav"); // for source Synth

a = Synth(\srcs, [\type,5,\buf,b,\amp,1,\busOut,12,\gate,1]);
b = Synth(\lpf, [\freq,100,\amp,0.6,\busIn,12,\busOut,13,\pan,-1],
          s,\addToTail);
c = Synth(\brf, [\freq,800,\bw,2,\amp,0.6,\busIn,13,\busOut,14,\pan,-1],
          s,\addToTail);
d = Synth(\brf, [\freq,800,\bw,1,\amp,2.5,\busIn,14,\busOut,0,\pan,0],
          s,\addToTail);

r = Routine({
            inf.do({
                    b.set(\freq, rrand(8000,12000));
                    c.set(\freq, rrand(200,1000));
                    d.set(\freq, rrand(800,3000));
                    0.25.wait;
                    })
            }).reset.play
```

2. parallel → output of source is the input of all filters, ouputs of filters mix together.

```
Buffer.freeAll;
b = Buffer.read(s, "/absolute/path/to/file.wav"); // for source Synth

~bw     = 0.1;    // Try to change
~amp    = 1;
~source = 5;
~rib    = 0.1;

a = Synth(\srcs, [\type,~source,\buf,b,\amp,1,\busOut,12,\gate, 1]);
b = Synth(\bpf, [\freq,100,\bw,~bw,\amp,~amp,\busIn,12,\busOut,0,\pan,-1],
          s,\addToTail);
c = Synth(\bpf, [\freq,800,\bw,~bw,\amp,~amp,\busIn,12,\busOut,0,\pan,-1],
          s,\addToTail);
d = Synth(\bpf, [\freq,800,\bw,~bw,\amp,~amp,\busIn,12,\busOut,0,\pan,0],
          s,\addToTail);

r = Routine({
            inf.do({
                    b.set(\freq, rrand(100,700));
                    c.set(\freq, rrand(1700,2100));
                    d.set(\freq, rrand(3000,3100));
                    ~rib.wait;
                    })
            }).reset.play
```

In SuperCollider there are also pre builded filterbanks which can be used to simulate an object's resonant modes.

We can set for each resonance:

- frequencies.
- amplitudes.
- duration.

```
SynthDef(\reson, {arg freqs=#[800, 1071, 1153, 1723],
                      amps=#[0.1,0.1,0.1,0.1],
                      ringtimes=#[1, 1, 1, 1],
                      busIn=2,
                      busOut=0,
                      pan=0;
                  var in, sig;
                      in  = In.ar(busIn);
                      sig = DynKlank.ar(`[freqs, amps, ringtimes ], in);
                      sig = Pan2.ar(sig, pan);
                  Out.ar(busOut, sig);
         }).add;
```

Listing 4.15: Resonators model

The Synths.

```
Buffer.freeAll;
b = Buffer.read(s, "/absolute/path/to/file.wav"); // for source Synth

e = Synth(\srcs,  [\type, 2,\freq,3,\amp,0.5,\busOut,12,\gate,1]);
a = Synth(\reson, [\busIn,12, \busOut,0,\pan,0], s, \addToTail);
```

Change resonator parameter.

```
a.setn(\freqs, Array.rand(4, 500, 9000));
a.setn(\amps, Array.rand(4, 0.1, 1));
a.setn(\ringtimes, Array.rand(4, 0.2, 2.4) );
```

Change source.

1. → White Noise.
2. → Impulse
3. → Dust.

```
e.set(\type,3,\amp,0.1);
```

Be careful with amplitude.

4.3.3 FM

In additive synthesis and in some cases in subtractive synthesis the signals are added together to obtain complex spectra.

The computational cost is high since for each partial or filter we have to generate a UGen.

Through the different basic modulation techniques (RM, AM, FM and PM) we can obtain equally rich spectra with only two signals:

- carrier.
- modular.

In this paragraph we explore characteristics of frequency modulation in its most basic form.

We use a rescaled sinusoidal control signal (modular) to vary the frequency of a sinusoid (carrier).

Three parameters:

- carrier frequency.
- modular frequency.
- deviation from carrier frequency (ampitude of modular signal).

If the frequency of modular signal is sub audio and deviation is under ~5 Hz → *vibrato effect*.

```
SynthDef(\fm_bas, {arg carf=440;
                    var modf, dev, mod, car;
                        modf = MouseX.kr(1,500);
                        dev  = MouseY.kr(1,500);
                        mod = carf + (SinOsc.ar(modf) * dev);
                        car = SinOsc.ar(mod);
                    Out.ar(0,car*0.3)
                  }).add;
```

Listing 4.16: Basic FM model

The Synth.

```
a = Synth(\fm_bas);
```

We can already recognize the characteristic sound of this technique.

In classic FM algorithm we can control:

- carrier frequency.
- modulation index → quantity of modulation.
- harmonicity index → quality of modulation.

We can realize a lot of different sounds changing only these two parameters.

```
SynthDef(\fm, {arg freq=440, amp=0, hid=0, mid=0, smth=0.001, pan=0, gate=0;
                var fmod, mod, por, env, sig;
                    fmod = freq * hid.round(smth);
                    mod  = freq + (SinOsc.ar(fmod) * fmod * mid);
                    por  = SinOsc.ar(mod);
                    env  = Linen.kr(gate, doneAction:2);
                    sig  = por * amp * env;
                    sig  = Pan2.ar(sig, pan);
                Out.ar(0, sig)
          }).add;
```

Listing 4.17: Classic FM model

Let's explore it with mouse position using the SynthDef already defined (3.7).

```
~kbusX = Bus.control(s,1);
~kbusY = Bus.control(s,1);

k = Synth(\ksig, [\type,0, \range, [0, 15], \busOut, ~kbusX]); // Harmon id
j = Synth(\ksig, [\type,1, \range, [0, 25], \busOut, ~kbusY]); // Modula iid
a = Synth(\fm,   [\freq,200, \amp,0.4, \hid,~kbusX.asMap, \mid,~kbusY.asMap, \
    gate,1]);
```

I suggest exploring the sonic potential of this technique empirically through mouse controls.

1. move the mouse. X and y positions can be dynamically reported in the Post window with *.poll* method invoked on MouseX.kr and MouseY.kr UGens in *ksig* SynthDef.
2. hear an intresting sound and write the values.
3. recall them in a deterministic way eventually interpolating them with other sounds through envelopes or other kind of controls.

In this way we can realize different musical objects fixing deterministic values with an empirical interactive strategy.

An exemple.

Let's define a SynthDef for envelope contol.

```
SynthDef(\env, {arg levels=#[0,1,0.5,0.7,0], times=#[1,1,1,1], dur=1,
                    busOut=0, t_gate=0;
                var env;
                    env = Env.new(levels,times.normalizeSum * dur);
                    env = EnvGen.kr(env, t_gate);
                Out.kr(busOut,env)
        }).add;
```

Listing 4.18: Envelope control model

The Synths.

```
~kbusX = Bus.control(s,1);
~kbusY = Bus.control(s,1);

h = Synth(\env,[\levels, [3.5,3.4,3.6,3.7,3.35],  // Harmonicity index
                \times,  [3,4,2,5],
                \dur, 40,
                \busOut, ~kbusX]); // write on control bus
i = Synth(\env,[\levels, [0,1,0.5,1.2,0.7],       // Modulation index
                \times,   [1,4,9,5],
                \dur, 42,
                \busOut, ~kbusY]); // write on control bus

a = Synth(\fm, [\freq,100, \amp,0.4, \hid,~kbusX.asMap, \mid,~kbusY.asMap]);

r = Routine.new({
                 a.set(\gate,1);
                 h.set(\t_gate,1);
                 i.set(\t_gate,1);
                 50.wait;
                 a.release(8);h.free;i.free;k.free;j.free;
                 }).reset.play
```

4.3.4 Granular

A technique that chops audio signals (pre-recorded or live) into tiny sound fragments (grains) and then manipulates and rearranges these grains to create entirely new sounds, textures, and soundscapes.

Technically it is identical to the enveloping process but it operates on a much finer scale than traditional sampling or wavetable synthesis.

Idea was introduced by Dennis Gabor in 1947: sounds can be represented by a series of elementary grains (short pulse).

The Greek composer Iannis Xenakis was the first to explicate a compositional theory for grains of sound in his works Analogique A and B (1958-59) for strings and tape - extract.

Here granular synthesis was on synthesized sounds.

Composer an researcher Curtis Roads was the first to implement granular synthesis on a computer in 1974 and adopt in many works as Half-life (1999) - extract.

He wrote also a book about composing with grains (Microsound).

In 1986 the Canadian composer Barry Truax implemented a real-time versions using the DMX-1000 Signal Processing Computer.

An example is his work Riverrun (1986) - extract.

Usually we can control:

- buffer to read.
- grain duration.
- grain position in buffer (normalized 0-1).
- transposition.

In this model there is a fixed sine envelope.

```
SynthDef(\grain_1,{arg busOut=0, buf=0, pos=0, trig=0,
                      gdur=0.1, amp=0, trsp=0, pan=0,
                      gate=0, done=2;
                  var gfreq, sig, env;
                      gfreq = 2.0/gdur;        // 2 overlap...
                      pos   = pos * BufDur.kr(buf);
                      sig   = TGrains.ar(2,    // channels
                                       trig,
                                       buf,
                                       trsp.midiratio,
                                       pos, // in seconds
                                       gdur,
                                       pan, // depends o chls number
                                       amp,
                                       4    // interpolation 1,2 4
                                         );
                    env   = Linen.kr(gate,doneAction:done);
                      sig   = sig * amp * env;
                  Out.ar(busOut, sig * env);
                    }).add;
```

Listing 4.19: Granular Synth sine envelope model

A basic exemple.

```
Buffer.freeAll;
b = Buffer.read(s, ExampleFiles.child);

~kpos  = Bus.control(s, 1);
~ktrig = Bus.control(s, 1);

~pointer = Synth(\ksig,  [\type, 0, \freq, 1, \busOut, ~kpos]);
~trigger = Synth(\ksig,  [\type, 4, \freq, 2, \busOut, ~ktrig]);
~grain   = Synth(\grain_1, [\busOut, 0,
                            \buf, b,
                            \pos, 0.1,
                            \trig, 0,
                            \gdur, 0.1,
                            \trsp, 0,
                            \amp, 1,
                            \pan, 0,
                            \gate, 1]);
```

Connect the trigger to a control signal and try to change parameters.

```
~grain.set(\trig, ~ktrig.asMap);
~trigger.set(\type,5,\freq,20);
~grain.set(\gdur, 0.1);
~grain.set(\pos, rand(1.0));
```

Connect a control signal to position parameter.

```
~grain.set(\pos, ~kpos.asMap);
```

Classical controls of position:

- random positions within the buffer.

```
~pointer.set(\type, 2);
```

- from beginning to end at a fixed rate (time stretching without pitch shifting).

```
~pointer.set(\type, 3, \freq, 0.1);
```

- from external devices (here mouse x).

```
~pointer.set(\type, 0);
```

- sequencing and scoring (Client side).

```
r = Routine({
              inf.do({
                        ~grain.set(\pos, [0.1,0.5,0.7].choose);
                        [0.2,0.4,0.6].choose.wait
                      })
             }).reset.play;

r.stop; ~grain.set(\gate, 0); ~pointer.free;  ~trigger.free;  ~kpos.free; ~
    ktrig.free;
```

We can find further exemples on *TGrains* Help file.

In textitTGrains UGen we can't change the grain envelope type.

textitGrainBuf do it.

```
SynthDef(\grain_2,{arg busOut=0, buf=0, pos=0, trig=1,
                      gdur=0.1, amp=0, trsp=0, pan=0, genv= -1,
                      gate=0, done=2;
                  var gfreq, sig, env;
                      gfreq = 2.0/gdur;        // 2 overlap...
                      sig   = GrainBuf.ar(2,   // channels
                                          trig,
                                          gdur,
                                          buf,
                                          trsp.midiratio,
                                          pos,  // 0-1
                                          4,    // interpolation 1,2 4
                                          pan,  // depends o chls number
                                          genv, // grain envelope
                                          512,  // max grain overlap
                                           );
                    env   = Linen.kr(gate,doneAction:done);
                      sig   = sig * amp * env;
                  Out.ar(busOut, sig * env);
                    }).add;
```

Listing 4.20: Granular Synth custom envelope model

Define an envelope.

Load it in a Buffer.

```
Buffer.freeAll;
e = Signal.hanningWindow(2048); // default
e = Signal.hammingWindow(2048);
e = Signal.welchWindow(2048);
e = Env.sine.asSignal(2048);
e = Env.triangle.asSignal(2048);
e = Env.linen(0.1,0.8,0.1).asSignal(2048);
e = Env.perc.asSignal(2048);

e = Buffer.loadCollection(s, e);
```

Set it in the Synth.

```
b = Buffer.read(s, ExampleFiles.child);

~kpos  = Bus.control(s, 1);
~ktrig = Bus.control(s, 1);

~pointer = Synth(\ksig,  [\type, 2, \freq, 5, \busOut, ~kpos]);
~trigger = Synth(\ksig,  [\type, 4, \freq, 15, \busOut, ~ktrig]);
~grain   = Synth(\grain_2, [\busOut, 0,
                            \buf, b,
                            \pos, ~kpos.asMap,
                            \trig, ~ktrig.asMap,
                            \gdur, 0.1,
                            \genv, -1,
                            \trsp, 0,
                            \amp, 1,
                            \pan, 0,
                            \gate, 1]);
e.free;
e = Env.perc.asSignal(2048);
e = Buffer.loadCollection(s, e);

~grain.set(\genv, e);

~grain.set(\gate, 0); ~pointer.free;  ~trigger.free;  ~kpos.free; ~ktrig.free;
```

Through granular synthesis, we can separate time and frequency:

- pitch shifting without changing durations.
- time stretching without changing pitch.

For these techniques three additional parameters are typically added:

- overlap factor.
- short random walk delay to avoid comb filter effect.
- small frequency deviation (chorus effect).

```
SynthDef(\harm_1,{arg busOut=0, buf=0, pos=0, trig=1,
                     gdur=0.1, amp=0, overlaps=8, rand=0, trsp=0, pan=0, genv=
    -1,
                     gate=0, done=2;
                 var sig, env;
                     sig = Warp1.ar(1,    // channels
                                  buf,
                              pos,  // 0-1
                              trsp.midiratio,
                                  gdur,
```

```
                                    genv, // grain envelope
                                    overlaps,
                                    rand, // 0-1 no comb filtering effevts
                                    4,    // interpolation 1,2 4
                                      );
                        env   = Linen.kr(gate,doneAction:done);
                      sig   = Pan2.ar(sig, pan);
                      sig   = sig * amp * env;
                  Out.ar(busOut, sig * env);
                    }).add;
```

Listing 4.21: Granular time stretching model

Pointer is from 0 to 1 in original Buffer duration (seconds).

There is no trigger (internal).

Overall amplitude not single grain.

```
Buffer.freeAll;
b = Buffer.read(s, ExampleFiles.child);

~kpos    = Bus.control(s, 1);
~pointer = Synth(\ksig,  [\type, 3, \freq, 1/b.duration, \busOut, ~kpos]);

~grain   = Synth(\harm_1, [\busOut, 0,
                            \buf, b,
                            \pos, ~kpos.asMap,
                            \gdur, 0.1,
                            \genv, -1,
                            \trsp, 0,
                            \amp, 1,
                            \pan, 0,
                            \gate, 1]);

~pointer.set(\freq, 0.4); // change time rate
~grain.set(\trsp, 2);     // change pitch in semitones
~grain.set(\rand, 0.98);  // change to limit comb filter effect
~grain.set(\gate, 0); ~pointer.free;  ~kpos.free;
```

4.3.5 FFT

Some classical techniques related to the analysis and resynthesis via the Fast Fourier Transform.

We analyze a signal by transforming it from time to frequency domain obtaining values of:

- magnitude.
- phase.

of a set of equidistant frequencies (bins) within the spectrum (as teeth of a comb).

We can manipulate these values in various ways and then bring them back into the time domain.

Two main procedures:

- analyze, process, and resynthesize a single signal.
- analyze two signals, combine the data in various ways and resynthesize the result.

In SuperCollider there ia a set of classes dedicated to Phase Vocoder which is a historical FFT analysis and resynthesis technique.

To use classes in these examples download and install sc3-plugins.

Be careful because FFT processes are CPU expensive.

In all the exemples we use the previous defined *source signal model* (4.6) and *control signal model* (3.7) SynthDefs.

Spectral Delay

Change delay time without pitch artefacts.

```
SynthDef(\sdel,{arg busIn=0,busOut=0,amp=0.5,fadeT=0.5,gate=0,pan=0,
                    maxdel=1,del= 0;
                var in,size,chain,env,sig;
                    in    = In.ar(busIn,1);
                    size  = 2048;
                    chain = FFT(LocalBuf(size), in); // analysis
                    chain = chain.pvcollect(size, {arg mag,fasi,bin;
   [mag + DelayN.kr(mag, maxdel, del.linlin(0,1,0,maxdel)), // process
    fasi] },0,256,-1);
                    env = Linen.kr(gate,fadeT,1,fadeT,doneAction:2);
                    sig = IFFT(chain) * env * amp;   // resyntesis
                    sig = Pan2.ar(sig, pan);
                Out.ar(busOut, sig);
         }).add;
```

Listing 4.22: Spectral Delay model

The Synths.

```
Buffer.freeAll;
b = Buffer.read(s, "/absolute/path/to/file.wav");

k = Synth(\ksig, [\type, 0, \range, [0, 1], \busOut,x]); // Normalized
e = Synth(\srcs, [\type, 5, \amp,1, \buf,b, \busOut,12, \gate,1 ],
          s,\addToTail);
a = Synth(\sdel, [\busIn,12, \amp,1, \maxdel,5, \del,x.asMap, \gate,1],
          s,\addToTail);
```

Brick Wall filter

Only bins above or below a threshold between +/-1 pass.

We can define ideal low and high pass filters.

```
SynthDef(\bwall, {arg busIn=0,busOut=0,amp=0.5,fadeT=0.5,gate=0,pan=0,
                      lpf=20000,hpf=20;
                      var in,size,chain,env,sig;
                      in    = In.ar(busIn,1);
                      size  = 2048;
                      chain = FFT(LocalBuf(size), in);
          chain = PV_BrickWall(chain, lpf.linlin(20,20000,-1,0)); // LPF
          chain = PV_BrickWall(chain, hpf.linlin(20,20000,0,1));  // HPF
                      env   = Linen.kr(gate,fadeT,1,fadeT,doneAction:2);
                      sig   = IFFT(chain) * env * amp;
                      sig   = Pan2.ar(sig, pan);
                  Out.ar(busOut, sig);
         }).add;
```

Listing 4.23: Brick Wall spectral filter model

The Synths.

```
e = Synth(\srcs,  [\type, 5,  \amp,1, \buf,b, \busOut,12, \gate,1 ]);
a = Synth(\bwall, [\busIn,12, \amp,1, \lpf,6000, \hpf, 1000, \gate,1],
          s,'addToTail');
```

Change filter freqs.

```
a.set(\lpf, 500, \hpf,200);
```

Dynamically (mouse x and y).

```
k = Synth(\ksig, [\type,0, \range, [500,10000], \busOut,x]);
j = Synth(\ksig, [\type,0, \range, [200, 1000], \busOut,y]);
a.set(\lpf, x.asMap, \hpf, y.asMap);
```

Rect Comb filter

Make gaps in spectrum.

- teeth → number of teeth in the comb.
- fase → starting phase of comb pulse.
- width → pulse width of the comb.

```
SynthDef(\rcomb, {arg busIn=0,busOut=0,amp=0.5,fadeT=0.5,gate=0,pan=0,
                      teeth=10,fase=0,width=0.5;
                  var in,size,chain,env,sig;
                      in    = In.ar(busIn,1);
                      size  = 2048;
                      chain = FFT(LocalBuf(size), in);
                      chain = PV_RectComb(chain, teeth, fase, width);
                      env   = Linen.kr(gate,fadeT,1,fadeT,doneAction:2);
                      sig   = IFFT(chain) * env * amp;
                      sig   = Pan2.ar(sig, pan);
                  Out.ar(busOut, sig);
         }).add;
```

Listing 4.24: Rect Comb filter spectral model

The Synths.

```
e = Synth(\srcs,  [\type, 5, \amp,1, \buf,b, \busOut,12, \gate,1 ]);
a = Synth(\rcomb, [\busIn,12, \amp,1, \gate,1],s,'addToTail');
```

Change phases.

```
a.set(\fase,rand(1.0).postln);
```

Change width and teeth dynamically.

```
k = Synth(\ksig, [\type,0, \range, [0.01, 0.5], \busOut,x]);
j = Synth(\ksig, [\type,0, \range, [1, 30], \busOut,y]);
a.set(\width, x.asMap, \teeth, y.asMap);
```

Mag Smear

Average magnitudes across bins, we can set the number of bins to average on each side of bin.

```
SynthDef(\smear, {arg busIn=0,busOut=0,amp=0.5,fadeT=0.5,gate=0,pan=0,
                      bins=1;
                  var in,size,chain,env,sig;
```

```
                        in    = In.ar(busIn,1);
                        size  = 2048;
                        chain = FFT(LocalBuf(size), in);
                        chain = PV_MagSmear(chain, bins);
                        env   = Linen.kr(gate,fadeT,1,fadeT,doneAction:2);
                        sig   = IFFT(chain) * env * amp;
                        sig   = Pan2.ar(sig, pan);
                    Out.ar(busOut, sig);
          }).add;
```

Listing 4.25: Mag Smear spectral model

The Synths.

```
k = Synth(\ksig,  [\type,0, \range, [0, 10], \busOut,x]);
e = Synth(\srcs,  [\type, 5, \amp,1, \buf,b, \busOut,12, \gate,1 ]);
a = Synth(\smear, [\busIn,12, \amp,1, \bins, x.asMap, \gate,1],s,'addToTail');
```

Freeze

Freeze magnitudes when freeze argument > 0.

```
SynthDef(\frz, {arg busIn=0,busOut=0,amp=0.5,fadeT=0.5,gate=0,pan=0,
                    frz=0;
                var in,size,chain,env,sig;
                    in    = In.ar(busIn,1);
                    size  = 2048;
                    chain = FFT(LocalBuf(size), in);
                    chain = PV_MagFreeze(chain, frz);
                    env   = Linen.kr(gate,fadeT,1,fadeT,doneAction:2);
                    sig   = IFFT(chain) * env * amp;
                    sig   = Pan2.ar(sig, pan);
                Out.ar(busOut, sig);
          }).add;
```

Listing 4.26: Freeze spectral model

The Synths and the sequence.

```
e = Synth(\srcs, [\type, 5, \amp,1, \buf,b, \busOut,12, \gate,1 ]);
a = Synth(\frz,  [\busIn,12, \amp,1, \gate,1],s,'addToTail');

r = Routine({
             10.do({a.set(\frz,0);
                    rrand(0.2,2).wait;
                    a.set(\frz,1);
                    rrand(0.6,2).wait;
                    });
             e.set(\gate,0); a.set(\gate,0);
             }).reset.play;
```

Cross synthesys.

There are several cross-synthesis techniques.

In this example the bins of signal A are replaced by those of signal B.

```
SynthDef(\cross, {arg busIn=#[0,1],busOut=0,amp=0.5,fadeT=0.5,gate=0,pan=0,
                    morph=0;
                  var inA,inB,size,chainA,chainB,chain,env,sig;
                    inA    = In.ar(busIn[0],1);
```

```
                        inB    = In.ar(busIn[1],1);
                        size   = 2048;
                        chainA = FFT(LocalBuf(size), inA);
                        chainB = FFT(LocalBuf(size), inB);
                        chain  = PV_Morph(chainA, chainB, morph);
                        env    = Linen.kr(gate,fadeT,1,fadeT,doneAction:2);
                        sig    = IFFT(chain) * env * amp;
                        sig    = Pan2.ar(sig, pan);
                    Out.ar(busOut, sig);
          }).add;
```

Listing 4.27: Morphing spectral model

The Synths (two different sources).

```
k = Synth(\ksig,  [\type,0, \range, [0, 1.0],    \busOut,x]);
e = Synth(\srcs,  [\type, 5, \amp,1, \buf,b,     \busOut,12, \gate,1 ]);
g = Synth(\srcs,  [\type, 2, \amp,0.8, \freq, 8, \busOut,13, \gate,1 ]);
a = Synth(\cross, [\busIn,[12, 13], \amp,1, \morph, x.asMap, \gate,1],
           s,'addToTail');
```

4.4 Algorithmic structures

In this chapter we expanded our sonic vocabulary.

We can now design formal structures by adopting the same strategies illustrated in the previous chapter or through non-deterministic algorithms.

Let's give an example of this second case.

Suppose we want to create a piece with dreamlike feeling based on three musical objects:

1. continuos changing crystal sounds (filter bank).
2. short sequences of percussive brilliant sounds (vector synth).
3. a strange piano (FFT filter).

For each we define one SynthDef, a function and a score sequence.

1. continuos changing crystal sounds.

```
SynthDef(\res, {arg freqs=#[800, 1071, 1153, 1723],
                    amps=#[0.1,0.1,0.1,0.1],
                    ringtimes=#[1, 1, 1, 1],
                    busOut=0, gate=0;
                var sig, amp, env, pan;
                    sig = WhiteNoise.ar;
                    amp = LFNoise1.kr(0.2).range(0,0.07);
                    sig = sig * amp;
                    sig = DynKlank.ar(`[freqs, amps, ringtimes ], sig);
                    env = Linen.kr(gate, 3, doneAction:2);
                    pan = LFNoise1.kr(0.3);
                    sig = Pan2.ar(sig * amp * env, pan);
                Out.ar(busOut, sig);
}).add;
// Define the musical object
a = {Routine({var fqs, aps, rng, res;
              fqs = Array.rand(4, 500, 9000);
              aps = Array.rand(4, 0.1, 1);
              rng = Array.rand(4, 0.2, 0.4);
              res = Synth(\res, [\freqs,fqs,
                                 \amps,aps,
                                 \ringtimes,rng,
```

```
                                    \gate,1]);
                rrand(0.5,4).wait;
                res.release(rrand(6,8))
                }).play;
    }
// Turns the musical object on and off randomly.
~glass = Routine({
                    inf.do({
                            if(0.8.coin==true){a.value};
                            [0.2,3,8].choose.wait})
                    }).reset.play;
```

2. short sequences of percussive brilliant sounds.

```
Buffer.freeAll;
b = Buffer.allocConsecutive(4, s, 1024);
b[0].sine2([1,2,5,7], [1,0.3,0.5,0.7]);   // Fill the buffers
b[1].sine2([2,3,8,9], [0.2,0.4,1.3,0.6]);
b[2].sine2([3,4,8,9,13,14], [1,0.2,0.5,0.6,1.3,0.5]);
b[3].sine2([1,3,4,7,8,9,12,15], [1.3,0.3,0.5,0.7,0.8,0.9,0.2,0.3]);

SynthDef(\pik, {arg bufn=4,freq=400,dur= 1,amp=0,pan=0,t_gate=0;
                 var sig,env;
                     env = Env.perc(0.01,dur-0.01);
                     env = EnvGen.kr(env,t_gate,doneAction:2);
                     sig = VOsc.ar(env.linlin(0,1,0,bufn), freq);
                     sig = Pan2.ar(sig * amp * env,pan);
                 Out.ar(0, sig)
                 }).add;
// Define the musical object
b = {Routine({
                rrand(2,10).do({ Synth(\pik,[\freq,
                                [90,94,95,97,100,104,105].choose.midicps,
                                            \dur, exprand(0.3,4.3),
                                            \amp, rrand(0.01,0.1),
                                            \pan, rand2(1),
                                            \t_gate,1,
                                            [0.15,0.1,0.2,0.25].choose.wait})
                }).play;
     }
// Turns the musical object on and off randomly.
~plink = Routine({
                    inf.do({b.value; [3,6,8].choose.wait})
                    }).reset.play;
```

3. a strange Piano.

```
v = Buffer.read(s, "/absolute/path/to/file.wav");
SynthDef(\spy,{arg buf=v,busOut=0,amp=0.5,fadeT=4,gate=0,pan=0,
                    teeth=10,fase=0,width=0.5;
                var sig,size,chain,env;
                    sig   = PlayBuf.ar(1, buf, loop:1);
                    size  = 2048;
                    chain = FFT(LocalBuf(size), sig);
                    chain = PV_RectComb(chain, teeth, fase, width);
                    env   = Linen.kr(gate,fadeT,1,fadeT,doneAction:2);
                    sig   = IFFT(chain) * env * amp;
                    sig   = Pan2.ar(sig, pan);
                Out.ar(busOut, sig);
         }).add;
// Define the musical object
c = {Routine({var piano = Synth(\spy, [\buf, v,
                                        \amp,rrand(0.5,1),
                                        \fadeT,rrand(1,5),
```

```
                                        \pan,rand2(1),
                                        \teeht,rrand(4,20),
                                        \width, rrand(0.1,0.4),
                                        \fase,rrand(0.4,0.7),
                                        \gate,1]);
                    rrand(0.5,4).wait;
                    piano.release(rrand(5,10)) // note off
                }).play
      };
// Turns the musical object on and off randomly.
~piano = Routine({
                    inf.do({c.value; [3,6,8].choose.wait})
                    }).reset.play;
```

We can now put it all together in a score that turns different combinations on and off randomly.

```
~sec = 10; // Number of sections
~score = Routine({
                ~sec.do({
                    if(0.6.coin==true){~plink.reset.play}{~plink.stop};
                    rrand(0.1,3).wait;
                    if(0.8.coin==true){~piano.reset.play}{~piano.stop};
                    rrand(0.1,3).wait;
                    if(0.4.coin==true){~glass.reset.play}{~glass.stop};
                    rrand(0.1,3).wait;
                });
                ~plink.stop;
                3.wait;
                ~piano.stop;
                5.wait;
                ~glass.stop
                }).reset.play
```

At each performance the piece will be a little different in its form but will always maintain the characteristics that we had set ourselves.

This is a simple example for explanatory reasons but it is clear how through this process we can arrive at designing extremely complex structures and forms.

4.5 Composition sketches proposal

Create a musical work using the instruments and the procedure just described.

Suggestions:

- research other synthesis techniques or delve deeper into some of them.
- modify the contents of the proposed SynthDefs based on your musical parameter control needs.
- explore algorithms that dynamically modify the form through random control structures such as if, case, and switch (search for help files).
- verify how and how much random parameters can influence the realization of the initial musical idea.

Chapter 5

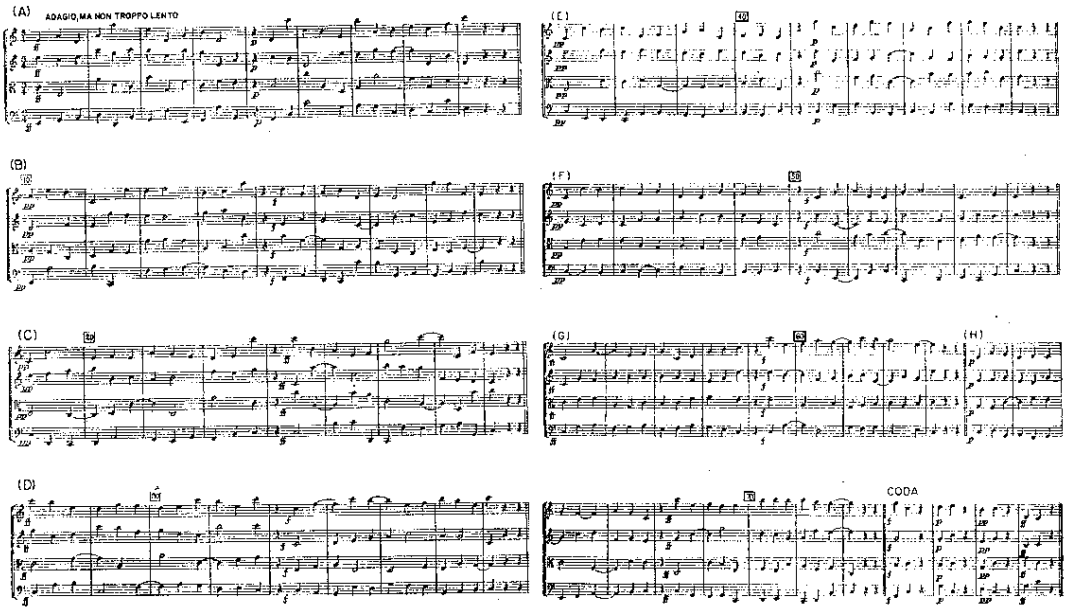
Algorithmic composition

5.1 Framework and historical context

The first piece of music written by a computer is the *Illiac Suite* for string quartet.

It was realized in 1956 at Illinois University.

Differently from music illustred in previous chapter computer did not generate the waveform but a symbolic representation of musical parameters to be transcribed into a musical score.



This work was an experiment to test various algorithms for composition.

1. random music → no rules.
2. skip-stepwise rule → no more than one repeated note.
3. cantus firmus starts on C with C chord for opening → cadence on C with leading tone in one of the four voices → resolution f tritone in VII-6, e.g. F/B must resolve to E/C.
4. octave-range rule.
5. only consonant chords permitted except for 6-4 chords → harmonic subroutine added.
6. parallel unisons, octaves, fifths and fourths still permitted → melodic subroutine added.
7. parallel fourths, 6-4 chords containing tenth still permitted.
8. best counterpoint.

Another important composer in the development of algorithmic music was the greek composer (naturalized french) Iannis Xenakis.

He used formulas developed by scientists to describe the behavior of particles in gases.

He saw his stochastic compositions as clouds of sound with individual notes as the analogue of gas particles organizing in this way clouds of notes.

The choice and distribution of notes was determined by procedures involving random choice, probability tables weighing the occurrence of specific events against those of others.

Iannis Xenakis - Pithoprakta for Orchestra (1955-56) - extract.

Another composer involved in this fields was Gottfried Michael Koenig that worked at the Utrecht University Institute of Sonology and wrote about his Project 1 software:

Project 1 is based on the findings of serial composition technique, and aims at the experimental experience of chance-governed constellations of parameter values. Chance, in this instance, replaces the permutations to which the serial composer was accustomed to subjecting rows, the consequences of which action for the resulting constellations of material was scarcely more predictable than in the case of aleatoric manipulations. Just as serial technique soon abandoned the "pointillist" style in order to apply the organizational principle to pre-formed units, "groups", the idea behind Project 1 likewise assumes that the unpredictability of chance will be neutralized by given lists of material and the appropriate selection of items from the lists, in order to facilitate the formation of medium-sized and larger form categories.

In more recent times and outside the academic environment a composer like Brian Eno writes:

Since I have always preferred making plans to executing them, I have gravitated towards situations and systems that, once set into operation, could create music with little or no intervention on my part. That is to say, I tend towards the roles of planner and programmer, and then become an audience to the results.

Nowadays it is one of the most explored artistic fields in generative artificial intelligence.

5.2 Software installation

In addition to SuperCollider and sc_kernel (optional) install:

- Lilypond
- Fosc (a Supercollider API for generating musical notation in lilypond)

Configure Fosc:

1. download the zip file
2. rename the *fosc-master* folder to *fosc*.
3. select SuperCollider (base) Kernel in this notebook (optional).
4. find your SuperCollider user extension hidden folder path:

```
Platform.userExtensionDir
```

5. move the *fosc* folder to your SuperCollider User Extensions folder.
6. search in SuperCollider Qarks *wslib* and installi it.
7. recompile Class library in SuperCollider or Refresh SuperCollider Kernel in Notebook.
8. add code to allow Fosc to communicate with LilyPond to your sclang startup file.

Mac users:

```
this.executeFile("/absolute/path/to/init_mac.scd");
```

Windows users:

```
this.executeFile("/absolute/path/to/init_win.scd");
```

9. Test it - generates three files:
   - test.ly
   - test.pdf
   - test.mid

   We should also hear an audio preview with default intrumental timbre.

```
~bpm   = 60;
~pitch = [60,64,67,nil,67,64,72,[67,64],nil,64,60,[64,67,72]];
~dur   = [8];
~exp   = [nil];

~print.(~pitch, dur:~dur, vel:~exp, bpm:~bpm);
```

We can load the midifile in an external midi sequencer or notation software as Musescore, Garageband, Sibelius, etc.

5.3 Symbolic musical representation and numbers

The purpose of this computer music practice is to help the composer generate and manipulate musical elements according to the syntactic rules of their chosen musical language.

One of the goals of algorithmic composition or computer-aided composition is to translate numerical values into symbols specific to musical notation and vice versa.

The final result consists of a musical score that can be interpreted by human performers.

In computer music as well as in western musical tradition the following sound parameters are typically generated and/or manipulated:

- pitch (frequency).
- rhythm (time).
- dynamic (amplitude).
- expression (timbre).

5.3.1 Pitch

Some symbolic representations of frequency sorted by level of abstraction:

- Diatonic tones.
  - diatonic representation by letters (notenames) or degrees.
  - fixed octave.
  - relative to syntactically defined rules (western music tones, semitones).
  - typically strings or chars data types.

```
a = [   0,    1,    2,    3,     4,    5,    6,    7]; // Scale degrees
a = ['do', 're', 'mi', 'fa', 'sol', 'la', 'si', 'do']; // Note name
a = ['C3', 'D3', 'E3', 'F3',  'G3', 'A3', 'B3', 'C4']; // Octave
```

- intervals.
  - determine the pitch relative to a root note (0) in semitones.
  - no fixed octave.

  - they may or may not be related to defined syntactic rules (scale degrees or other).
  - typically int (positive and negative).

```
a = [ 0, 2, 4, 5, 7, 9, 11, 12]; // Major scale model
a = [-3, 5, 6, -4, 0, 2];        // A sequence
```

- MIDI note.
  - determines pitch as a fractional MIDI note.
  - they are not related to syntactically defined rules.
  - root note → 60 = C4 = middle C.
  - divided in semitones (int).

```
a = [ 60, 62, 64, 65, 67, 69, 71, 72];
```

- frequency.
  - determines the pitch as a frequency in Hertz (or cps).
  - absolute values not relate in any way to syntactically defined rules.
  - int or float

```
a = [ 262, 294, 330, 349, 392, 440, 494, 523];
```

In computer music there are further forms of pitch representation but they can easily be traced back to those just mentioned.

In SuperCollider we can convert from a form to another:

```
~degree    = [0, 2, 4, 5, 7];   // Degree
~root      = 60;
~midinote  = ~root + a;         // Midi note
~freqency  = ~midinote.midicps; // Frequency
~midinote1 = ~freqency.cpsmidi;
~degree1   = ~midinote1 - 60;

~midinote.postln;               // change
```

Chords → 2D arrays.

```
~pitch = [[60,64,67,72],62,[60,65,69,72],64,[62,65,67,71],62,[60,64,67,72]];
~dur   = 4!6 ++ [2];

~print.(~pitch, ~dur);
```

Rest → nil instead note midi number.

```
~pitch = [60,62,64,nil,65,67,nil,74,73,74,75,nil,76,72,nil,64];
~dur = [16];

~print.(~pitch, ~dur);
```

5.3.2 Rhythm

We can define any kind of musical event in time choosing at least two of these parameters:

- onsets → absolute time of a sound event relative to the beginning (as in DAW software).
- delta times → time between two consecutive sound events.
- durations → duration of a sound event.

All these parameters can be specified as:

- absolute measurement units (seconds or millisecond).

- relative to a bpm.
  - 1 → beat, reference unit (could be quarter, eight, whole, etc).
  - 1/4 → fractional notation.
  - 0.5 → multiply factor (result from fractional notation).
  - 3:2 → Rhythmic proportions (ratios of one single beat or its subdivisions).

We can convert the values by simple mathematical operations.

```
~bpm = 82;
~sec = 60/~bpm;  // bpm     --> seconds
~bpm = 60/~sec;  // seconds --> bpm
```

In this chapter we adopt a notation similar to lilypond notation:

- 1 → whole.
- 2 → half.
- 4 → quarter.
- 8 → eight.
- 16 → sixteen.
- 32 → thirty-second.
- 64 → sixty-fourth.

```
a = [72,76];
b = [16,16,8,32,32,32,32,8,4];

~print.(a, b);
```

If we want to define dotted or irregular rhythms we must use this syntax:

```
//    [unity, [subdivisions] ]
a = [ [4,     [3, 1         ] ] ];
```

We can mix regular and irregular/dotted rhythms.

```
a = [60,     nil,  62,       63,64,nil,[62,66],67,  68, [64,72]];
b = [4,  [4,[3,   1]], [4, [1,  1,  1,     1,  1]], 8, 8];

~print.(a, b);
```

5.3.3 Dynamic

Three different measurement units:

- 1.0 → linear amplitude (floating point form 0.0 to 1.0).
- 127 → midi velocity (integer from 0 to 127).
- -6 → decibels (from -inf to 0.0).

Conversions:

```
~lin  = 64  / 127;  // velocity to linear
~vel  = 0.5 * 127;  // linear to velocity
~lin1 = -6.dbamp;   // dB to linear
~db   = 0.5.ampdb;  // linear to dB
~cub  = 0.5**4;     // linear to quartic (best for human perception)
```

In this chapter we adopt a mix between amplitude and expression signs because I think that if we want define overall amplitude, crescendo and diminuendos algorithms are not necessary.

Only three velocity levels:

- nil → mezzoforte.
- 1-42 → piano.
- 43-80 → marcato.
- 81-127 → accentato.

5.3.4 Collections

As we saw we can describe sets of values as collections.

In SuperCollider Array and List.

Collections are a musically neutral data type as they can contain any data type.

In algorithmic composition we can think it musically in two ways:

- out of time (pre compositional materials).
  - rules paradigm.
  - item positions on x axis are indexes without time references (scale degree or other).
  - scale

```
//          0   1   2   3   4   5   6   7          // Scale degrees
~dorico = [60, 62, 64, 65, 67, 69, 71, 72];        // Absolute pitches
~chrom  = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11]; // Intervals
~dur    = [8];

~print.(~dorico, ~dur);
```

  - series

```
~serie  = [1, 10, 6, 4, 8, 2, 9, 7, 0, 5, 11, 3]; // Dodecaphony
~reich  = [64, 66, 71, 72, 73, 66];               // Minimalism
~dur    = [8];

~print.(~reich, ~dur);
```

  - harmonic fields (Array or Array 2D)

```
~mayor = [0, 4, 7, 12];
~harm  = [[60, 64, 67, 72],
          [62, 65, 67, 71],
          [60, 65, 69,72],
          [60, 64, 69,72]];
~harmd = [[0, 4, 7, 12],  [2, 5, 7, 11],  [0, 5, 9,12], [0, 4, 9,12]];
~dur   = [8];

~print.(~harm, ~dur);
```

- timeline.
  - sequence paradigm.
  - items are placed on a timeline (x axis).

```
//   - Sequence of index from defined scale
//          0   1   2   3   4   5   6   7
~scale  = [60, 62, 64, 65, 67, 69, 71, 72];
~melody = 10.collect({rrand(0,7)});
~seq    = ~melody.collect({arg i; ~scale[i]});
~dur    = ([16,8,16]!3++[4]).flat;
~exp    = ~melody.collect({ [nil,100,40].wchoose([0.6,0.2,0.2]) });

~print.(~seq,~dur, ~exp);
```

5.3.5 Live

Send numerical data via MIDI to external sound generators (Reaktor, GarageBand, Logic) without print it in a musical score.

Open a midi sound generator software and set it.

```
MIDIClient.init; // Checks available MIDI devices

c = MIDIOut.new(0); // Choose the destination - 0 = "Driver IAC"
c.latency_(0);      // Set latency to 0 sec

~evt   = 100;
~bpm   = 60;
~pitch = ~evt.collect({rrand(60,75)});
~durs  = ~evt.collect({[32, 32, 32, 32, 32, 8, 16].choose});
~expr  = ~evt.collect({[100, 40, 60].choose});
p = Pbind(
          \type,     \midi,
          \midicmd,  \noteOn,
          \midiout,  c,
          \chan,     0,
          \midinote, Pseq(~pitch.replace([nil],\rest)),
          \sustain,  Pseq(1/~durs * 4, inf),   // Duration
          \dur,      Pseq(1/~durs * 4,inf),    // Delta time
          \amp,      Pseq(~expr/127,inf),      // 0.0 to 1.0
          ).play;
```

Listing 5.1: Live midi out sender model

5.4 Generative techniques and processes

In western musical tradition superposition of two or more voices have only two approaches:

- counterpoint (horizontal dimension - static voice allocation).
- harmony (vertical dimension - dynamic voice allocation).

Approaches to which different musical languages refer.

In general if we want formalize our musical ideas we should act as in previous chapters.

Recap.

1. choose a set of parametric structures that we want to use as basic material according to musical rules (scale, modes, series, chords, rhythmic or dynamic patterns, etc.).
2. define a set of possible modifications always starting from musical needs and rules (variations, filters, interpolations, augmentations, diminutions, etc.).
3. organize them over time by defining a formal structure choosing from two strategies:
   - top to bottom (from a fixed general structure to particular).
   - bottom to top (from particular to a general formal structure).

In carrying out these three steps, we have only two possible approaches often mixed toghether:

- nondeterministic.
- deterministic.

5.4.1 Modes and scales

Simplifying:

- *mode* → gregorian to baroque music.

- *scale* → baroque to jazz/pop music.

The way through which composer organize the pitches of a piece.

They are usually some kind of subdivision of the octave interval.

We can represent it in two ways:

- intervals → chromatic distance from 0 (root) in semitones.
- degrees → diatonic distance from 0 (root) in note names.

For exemple major scale:

```
~degrees   = [0, 1, 2, 3, 4, 5,  6,  7];
~intervals = [0, 2, 4, 5, 7, 9, 11, 12];
```

Then we have to choose a root note in midi values which will assume the function of the first degree (0)

- in modal rules is called *finalis*.
- in tonal (harmonic) rules is the *key* (D major, E-flat minor, etc.).

```
~major = [0,2,4,5,7,9,11,12];
~root  = 60;
~pitch = ~root + ~major;
~dur   = [1/8];
~print.value(~pitch, ~dur, filepath:~filepath);
```

We can use historical models or invent our own.

Scale

In SuperCollider there is a repository of different scales from different coltures.

```
Scale.directory;
```

Choose a scale from the availables and recall the interval model.

```
~model     = Scale.lydian;
~intervals = ~model.degrees;
~root      = 60;
~pitch     = ~root + ~intervals;
~dur       = [8];
~intervals,postln;
~pitch.postln;
~print.(~pitch, ~dur);
```

Once the interval model has been established, we can create a melody by recalling the different degrees of the model according to deterministic or indeterministic principles.

Incises and motives

We can write an incise in a deterministic way.

```
~mod  = Scale.lydian.degrees;
~degr = [0,3,nil,5,2,3,6,5]; // in scale degrees
~root = 60;
~ptch = ~degr.collect({arg i; if(i.notNil) {~mod.at(i) + ~root}});
~durs = [8,8,16,16,[8,[1,1,1]],4];
~expr = [nil, 100, nil, nil, nil, nil, nil, 40];
~print.(~ptch, ~durs, ~expr);
```

We can also program a *random incise generator* algorithm.

It's not necessary but can be a good pratice to formalize algorithm in functions.

This algotithm generate inceses according to simple rules:

1. set the number of notes.
2. set an interval model.
3. set a root note.
4. set a percentage of random rests (0.0 - 1.0).

```
~inc  = {arg evt=1, imodel, root=60, rest=0.0;
         var pitch, rests;
             rests = [imodel,[nil]];
             pitch = evt.collect{var i;
                                 i = rests.wchoose([1-rest,rest]).choose;
                                 if(i.notNil)
                                 {i = i + root} };
        };
```

Test it.

```
~evt  = rrand(2,5);
~mod  = Scale.lydian.degrees;
~root = 60;
~rest = 0.2;
~durs = [8];
~seq  = ~inc.(~evt, ~mod, ~root, ~rest);

~print.(~seq, ~durs)
```

5. set a database of rhythmic patterns from which to extract one according to the number of pitches. Each pattern should be of the same lenght (a beat or its multiple).

```
~inc  = {arg evt=1, imodel, root=60, rest=0.0;
         var pitch, rests, durs;
            rests = [imodel,[nil]];
            pitch = evt.collect{var i;
                                i = rests.wchoose([1-rest,rest]).choose;
                                if(i.notNil)
                                {i = i + root} };
            durs = switch(evt,
                        2, {[[8,8],
                             [[4,[3,1]]],
                             [[4,[1,3]]],
                             [[4,[2,1]]],
                             [[4,[1,2]]] ].choose},
                        3, {[[8,16,16],
                             [16,8,16],
                             [16,16,8],
                             [[4,[1,1,1]]] ].choose},
                        4, {[[16,16,16,16],
                             [[4,[2,1,1,2]]],
                             [[4,[2,2,1,1]]]].choose},
                        5, {[[16,16,16,16],
                             [[4,[1,1,1,1,1]]],
                             [[4,[2,2,1,1,1]]],
                             [[4,[1,2,1,2,1]]]].choose}
                             );
            [pitch, durs]
        };
```

Test it.

```
~evt  = rrand(2,5);
~mod  = Scale.lydian.degrees;
~root = 60;
~rest = 0.2;
~seq  = ~inc.(~evt, ~mod, ~root, ~rest );

~print.(~seq[0], ~seq[1]);
```

6. set a percentage of random velocities/expressions to apply on notes.
   2D array:
   0 → dataset from wich values are chosen.
   1 → percentage of random velocities (0.0-1.0).

```
~inc  = {arg evt=1, imodel, root=60, rest=0.0, exp=[[nil],[0.0]];
         var pitch, rests, durs,  vel;
             rests = [imodel,[nil]];

             pitch = evt.collect{var i;
                                 i = rests.wchoose([1-rest,rest]).choose;
                                 if(i.notNil)
                                 {i = i + root}
                                 };

            durs = switch(evt,
                        2, {[[8,8],
                             [[4,[3,1]]],
                             [[4,[1,3]]],
                             [[4,[2,1]]],
                             [[4,[1,2]]] ].choose},
                        3, {[[8,16,16],
                             [16,8,16],
                             [16,16,8],
                             [[4,[1,1,1]]] ].choose},
                        4, {[[16,16,16,16],
                             [[4,[2,1,1,2]]],
                             [[4,[2,2,1,1]]]].choose},
                        5, {[[16,16,16,16],
                             [[4,[1,1,1,1,1]]],
                             [[4,[2,2,1,1,1]]],
                             [[4,[1,2,1,2,1]]]].choose}
                            );
            vel = pitch.collect{arg i;
                                 if(i.notNil)
                                 {[exp[0],[nil]].wchoose([exp[1],1-exp[1]])
                                                 .choose} }.flat;
            [pitch, durs, vel]
        };
```

Listing 5.2: random incise function

Test it.

```
~evt  = rrand(2,5);
~mod  = Scale.lydian.degrees;
~root = 60;
~rest = 0.2;
~exp  = [[40,100],0.3];
~seq  = ~inc.(~evt, ~mod, ~root, ~rest, ~exp );

~print.(~seq[0], ~seq[1] , ~seq[2]);
```

In this way we can also construct motives with several incises.

```
~mot = {arg evt, imodel, root=60, rest=0.0, exp=[[nil],[0.0]];
        var p,r,v;
            p = [];
            r = [];
            v = [];
            evt.do({arg i;
                    i = ~inc.(rrand(2,5), imodel, root, rest, exp);
                    p = p ++ i[0];
                    r = r ++ i[1];
                    v = v ++ i[2];
                    });
            [p.flat,r,v.flat]};
```

Listing 5.3: random motive function

Test it.

```
~evt  = rrand(3,10);
~mod  = Scale.lydian.degrees;
~root = 60;
~rest = 0.3;
~exp  = [[40,100],0.3];
~seq  = ~mot.(~evt, ~mod, ~root, ~rest, ~exp );
~print.(~seq[0], ~seq[1] , ~seq[2],
        filepath:'/absolute/path/to/test');
```

I'd like to underline:

- the *inc* function is based on musical rules we established previously (choice of scale, number of notes, set of rhythmic cells, etc.).
- in implementing it we mixed deterministic and indeterministic procedures.

Now we can save short sequences as single midi files (different names) and edit it in software like Musescore, Sibelius or generic midi sequencer.

Variations

In the previous paragraph we generated incises and motives.

Let's see now few possible variations.

Pitches, rhythms and velocities sequencies are represented as Array or List.

In SuperCollider there are numerous methods that we can call on them to modify the contents.

In next exemples we apply methods on specific parameters but in principle almost all methods are valid for all parameters.

- chromatic trasposition → sum an offset in semitones (absolute).
  - pitch.

```
~ctsp = {arg seq=[60], trsp=0;
             seq.collect({arg i;
                          if(i.notNil)
                          {trsp + i}
                          });
         };
```

Listing 5.4: chromatic transposition function

An exemple.
Rules:

1. define an incise.
2. traspose pitch random +/- 5 semitones n times.
3. duplicate rhythms and velocities n times.
4. put it toghether.

```
~ptch = [60, 64, 67,  69];
~dur  = [ 8, 16, 16,   8];
~evt  = 3;
~ptch = ~ptch ++ ~evt.collect({
                                ~ctsp.(~ptch, rand2(5))
                                });
~ptch = ~ptch.flat;
~print.(~ptch, ~dur);
```

- diatonic trasposition → sum an offset in degrees (relative to a scale or interval model).
  - pitch.

```
~dtsp = {arg seq=[60],trsp=0,scale=[0,2,4,7,9,11];
         var dg;
         seq.collect({arg i, id;
                      if(i.notNil)
                      {dg = scale.performKeyToDegree(i, 12);
                      dg = dg + trsp;
                      scale.performDegreeToKey(dg,12,0)}
                   })
         };
```

Listing 5.5: diatonic transposition function

An exemple.

```
~mod  = Scale.major.degrees;
~ptch = [60, 64, 67,  69];
~dur  = [ 8, 16, 16,   8];
~evt  = 3;
~ptch = ~ptch ++ ~evt.collect({
                                ~dtsp.(~ptch, rand2(5), ~mod)
                                });
~ptch = ~ptch.flat;
~print.value(~ptch, ~dur);
```

- retrograde → reverse array items.
  - pitch.
  - rhythm.
  - expression.

```
~retr = {arg seq; seq.reverse};
```

Listing 5.6: retrograde function

An exemple.
Different combinations give different musical results.

```
~ptch = [60, 67, 76,  nil,72,74,  77];
~dur  = [8,  16, 16,[4,[1, 1, 1]],8];
~exp  = [100, nil, 40, nil, 100, 40,nil];
~ptch  = ~ptch ++ ~retr.(~ptch);
~dur   = ~dur ++ ~retr.(~dur);
~print.(~ptch , ~dur, ~exp);
```

- inversion → hoose a root note for mirroring.
  - pitch.

```
~inv  = {arg seq, root=60;
         seq.collect({arg i;
                      if(i.notNil)
                      {i-root * -1 + root}
                      })
         };
```

Listing 5.7: inversion function

A deterministic exemple.
Different combinations give different musical results.

```
~ptch = [67, 69, 71,  nil, 65,64];
~dur  = [16];
~exp  = [100, nil, 40, nil, 100, 40,nil];
~root = 73;

~ptch  = ~ptch ++ ~inv.(~ptch, ~root);

~print.(~ptch , ~dur, ~exp);
```

- retrograde of inversion → hoose a root note for mirroring.
  - pitch.

```
~rinv = {arg seq, root=60;
             seq = seq.collect({arg i;
                                if(i.notNil)
                                {i-root * -1 + root}
                                });
             seq.reverse
       };
```

Listing 5.8: retrograde of inversion function

An example that combines different random variations on different parameters.

```
~evt  = 4;
~ptch = [60,64,67,nil,69,71];
~ptch = [~ptch,
         ~retr.(~ptch),
         ~inv.(s, rrand(68,72)),
         ~rinv.(s, rrand(68,72))];

~dur = [8,16,16,8,8,16,16,8];
~dur = [~dur, ~retr.(~dur)];

~exp = [100, nil, 40, nil, 100, 40];

~ptch = (~evt.collect({~ptch.choose})).flat;
~dur  = (~evt.collect({~dur.choose})).flatten(1);

~print.(~ptch, ~dur, ~exp);
```

- palindrome without last element → there are different versions for different musical situations, see Array help file.
  - pitch.
  - rhythm.
  - expression.

```
~pal = {arg seq; seq.mirror1};
```

Listing 5.9: palindrome function

As exemple a radom palindrome plus random diatonic trasposition.

```
~evt  = 10;
~mod  = Scale.major.degrees;
~ptch = [60,64,67,69];
~ptch = [~pal.(~ptch), ~ptch];
~ptch = ~evt.collect({ ~dtsp.(~ptch.choose, rand2(3), ~mod) });
~ptch = ~ptch.flat;
~dur  = [32];
~exp  = [100, nil, nil,nil,nil,nil];

~print.(~ptch, ~dur,~exp);
```

- stretching → re-scale rhythms by a scaling factor (multiples of 2).
  - rhythm.

```
~evt  = 3;
~ptch  = [60,64,67,nil,69,71,74,67];
~dur   = [8,16,16,16,16,8];
~scale = [0.5,1,2];
~dur   = (~evt.collect({
                        (~dur * ~scale.choose).asInteger
                        })
          ).flatten(1);

~print.value(~ptch, ~dur);
```

- permutation → all possible (array size max 170 items).
  - pitch.
  - rhythm.
  - expression.

```
~perm = {arg seq;
       var evt;
           evt = seq.size.factorial.postln;
           evt.collect({arg i;
                      seq.permute(i)
                      })
         }
```

Listing 5.10: all permutations function

As exemple we can apply it on irregular rhytmic subdivisions.

```
~evt  = 5;
~ptch = [60,67,nil,76,72,83, 53,nil,55];
~ptch = ~perm.(~ptch).postln;
~ptch = ~evt.collect({~ptch.choose}).flat;

~subd = [1,2,1,1];
~prms = ~perm.(~subd).postln;
~dur  = ~evt.collect({[[4,~prms.choose],4]}).flatten(1);

~exp = [100, nil, 30,nil,nil,nil];

~print.(~ptch, ~dur, ~exp);
```

- random order. → change order in random way.
  - pitch.
  - rhythm.
  - expression.

```
~rordr = {arg seq; seq.scramble};
```

Listing 5.11: random order function

A randomic variation of previous exemple.

```
~evt  = 5;                  // repetitions
~subd = [1,2,1,1];          // subdivision
~ptch = [60,67,nil,76,72,83, 53,nil,55];
~ptch = ~evt.collect({~rordr.(~ptch)}).flat;
~dur  = ~evt.collect({[4,[4,~rordr.(~subd)]]}).flatten(1);
~exp  = [100, nil, 30,nil,nil,nil];
~print.(~ptch, ~dur, ~expr);
```

This is an example of when deterministic and indeterministic procedures achieve the same perceptual result.
This happens when deterministic procedures provide too many similar variations making them indistinguishable from each other.
As can be intuited from some of the examples the most musically interesting results are obtained by combining different variation techniques.
It's true for a single parameter and in the combination of different parameters.

- phasing. → use arrays of different lengths for the parameters.
  - pitch.
  - rhythm.
  - expression.

  Replace the .at() method with .wrapAt() or foldAt().
  Look at Array help files what they do.

```
~evt  = 40;
~ptch = [60, 75, nil, 76, 79, 92, nil, 70];
~rhyt = [ 8, 16,  16, 16,  8, 4];
~exp  = [80, nil, nil,nil, nil,  30, nil, nil, nil];
~ptch = ~evt.collect({arg i; ~ptch.wrapAt(i)}).flat;
~rseq = ~evt.collect({arg i; ~rhyt.foldAt(i)}).flatten(1);
~vseq = ~evt.collect({arg i; ~acc.wrapAt(i)}).flat;

~print.(~ptch, ~rseq, ~exp);
```

In the Array help file we can find further examples of operations on sequences of symbols that can easily be applied to parameters taken into consideration.

5.4.2 Harmonic fields

In harmonic systems the interval patterns that define the type of chord are built on the degrees of a scale:

- tonica → I degree.
- mediante → III degree.
- dominante → V degree.
- etc.

```
// degree   I        II       III       IV        V          VI         VII
~triadi = [[0,4,7],[2,5,9],[4,7,11],[5,9,12],[7,11,14],[9,12,16],[11,14,17]];
~root   = 60;
~print.(~triadi + ~root,[2])
```

The basic chords are made up of three notes → triads.

In many harmonic systems one of them is doubled → four notes.

There are also chords made up of four or more notes without doubling (seventh, ninth, eleventh and thirteenth chords).

```
~settime = [[0,4,7,11],[2,5,9,12],[4,7,11,14],[5,9,12,16],
            [7,11,14,17],[9,12,16,19],[11,14,17,21]];
~root    = 60;
~print.(~settime + ~root,[2])
```

Chord database

We define the function *chord* which contains a database of interval patterns of the language we want to use.

To allow for the use of seventh chords and to even out the voicing we use four-note chords.

Arguments:

- root note.
- chord sigle.

Output → array of midi notes obtained by mapping the acronym to the corresponding interval pattern.

```
~chord = {arg root, type;
          var intv;
              intv = (maj:    [0, 4, 7, 12],
                      maj7:   [0, 4, 7, 11],
                      maj6:   [0, 4, 7,  9],
                      maj9:   [0, 4, 7, 14],
                      min:    [0, 3, 7, 12],
                      min7:   [0, 3, 7, 10],
                      min6:   [0, 4, 7, 9],
                      min9:   [0, 3, 7, 14],
                      dom7:   [0, 4, 7, 10],
                      dom9:   [0, 4, 7, 14],
                      ecc:    [0, 4, 8, 12],
                      dim:    [0, 3, 6, 12],
                      dim7:   [0, 3, 6, 9]
                      )[type];
          intv.collect({arg i; root + i});
          };
```

Listing 5.12: chord model function

Choose a chord sigle and a root note.

```
~ptch = ~chord.(60, \min9);
~print.([~ptch], [2]);
```

We can apply a chord pattern to all degrees of a scale, creating an harmonic progression.

```
~progClassic = [~chord.(60, \maj),
                ~chord.(64, \min),
                ~chord.(67, \dom7),
                ~chord.(69, \min7),
                ~chord.(62, \dom7),
                ~chord.(67, \dom7),
                ~chord.(60, \min),
                ~chord.(65, \dom7),
                ~chord.(58, \maj),
                ~chord.(71, \dim),
                ~chord.(72, \maj),
                ~chord.(60, \maj),
                ];
~print.(~progClassic, [4]);
```

A little more Jazzy.

```
~progJazzy=[~chord.(60, \maj7),
            ~chord.(65, \maj6),
            ~chord.(68, \min7),
            ~chord.(67, \dom7),
            ~chord.(73, \min6),
            ~chord.(68, \maj7),
            ~chord.(60, \min9),
            ~chord.(63, \maj6),
            ~chord.(57, \dim),
            ~chord.(54, \maj9),
            ~chord.(63, \min),
            ~chord.(60, \maj7),
            ];
~print.(~progJazzy, [4]);
```

Voicing

In the previous sequences all chords are presented in close voicing form.

Intervals between the highest and lowest notes is relatively small, usually within an octave.

All chords can also assume the open voicing configuration.

Intervals are spanned over multiple octaves.

We change the shape of a chord from close to open voicing form.

```
~voicing = {arg note=[60,64,67,72], octave = [-1,0,-1,0];
                note.collect({arg note, i; octave[i]*12 + note}) };
```

Listing 5.13: voicing model function

Test it.

```
~closed = ~chord.(60, \maj);
~open   = ~voicing.(~closed, [-1,0,-1,0]);
~print.([~closed, ~open],[2])
```

In both forms chords are in the root state and proceed in parallel motion.

In classical harmony, a chord can be represented in an inversion.

Inversions are obtained by transposing one of the chord's notes up an octave.

In a three-note chord there are three possible inversions, in a four-note chord four, and so on.

We can use the same function.

```
~fond   = ~chord.(60, \maj);
~first  = ~voicing.(~fond, [1,0,0,0]);
~second = ~voicing.(~fond, [1,1,0,0]);
~print.([~fond, ~first, ~second],[2])
```

In pop and jazz, there are various techniques for creating inversions.

The most commonly used are:

- drop 2 → transpose one octave below the second note from the top.
- drop 3 → transpose one octave below the third note from the top.

Chords in these forms appear fuller and more rounded than their root form.

These techniques combine the concepts of inversion and close-open voicing.

Let's define two functions that realize them.

```
~drop2={arg chord;
        var sorted, dropNote,newVoicing;
            sorted        = chord.sort;
            dropNote      = sorted[2];    // 2nd highest
            newVoicing    = sorted.copy;
            newVoicing[2] = dropNote - 12;
            newVoicing.sort
        };
~drop3={arg chord;
        var sorted, dropNote,newVoicing;
            sorted        = chord.sort;
            dropNote      = sorted[1]; // 3rd highest
            newVoicing    = sorted.copy;
            newVoicing[1] = dropNote - 12;
            newVoicing.sort
        };
```

Listing 5.14: drop model functions

Test it.

```
~fond = ~chord.(72, \maj6);
~d2   = ~drop2.(~fond);
~d3   = ~drop3.(~fond);
~print.([~fond,~d2,~d3], [2])
```

These two techniques work well with typical jazz harmonies, but they don't work with classical harmonic progressions because they keep the voices moving in parallel.

Let's transform the previous jazzy progression into late parts (drop 3).

```
~prog = ~progJazzy.collect({arg i; ~drop3.(i)});
~print.(~prog, [4]);
```

Make a progression more dynamic by randomly choosing the shape of each chord.

```
~prog = ~progJazzy.collect({arg i;
                                d = rand(3);
                                switch(d,
                                       0, {i},
                                       1, {~drop2.(i)},
                                       2, {~drop3.(i)}) });
~print.(~prog, [4]);
```

Voice-leading

Voice leading seeks to:

- minimize pitch distance between successive notes in each voice.
- preserve common tones (same note between chords).
- avoid big leaps (> octave) unless musically justified.
- maintain a consistent order: low → high (no voice crossing).

Let's define a function that realizes it between two chords.

```
~vLead = {arg ch1, ch2;
          var result, octVar;
              result = List.new;
              octVar = {arg note; [note - 12, note, note + 12]};
              ch2.do({arg note, i;
                      var cand, best;
                          cand = octVar.(note);
                          best = cand.sortMap({arg n; (n - ch1[i]).abs}).first;
                      result.add(best) });
              result };
```

Listing 5.15: voice-leading model function

Test it.

```
a = [~chord.(60, \maj7), ~chord.(71, \maj6)];
b = ~vLead.(a[0],a[1]);
~print.(a++[a[0]]++[b],[2]);
```

Let's apply the function to a chords sequence.

```
~prevSeq   = ~progJazzy;            // Starting sequence
~prevChord = ~prevSeq[0];           // First chord (midi notes)
~voicedSeq = [~prevChord];          // Put it in final sequence
~prevSeq.drop(1).do({arg nextChord; // Start from second chord
                     var voiced;
                         voiced     = ~vLead.(~prevChord, nextChord);
                         ~voicedSeq = ~voicedSeq.add(voiced);
                         ~prevChord = voiced });
~print.(~prevSeq ++ ~voicedSeq,[4]);
```

The algorithm just described works well with jazz and pop harmonies but is less effective when applying traditional harmony rules. We add a function that searches for common notes.

```
~vLeadC= {arg ch1, ch2;
          var result, remain, octVar;
              result = List.new;
              remain = ch2.copy;
              octVar = {arg note; [note-12, note, note+12]};
          ch1.do({arg note1;
                  var intv, common;
                      intv   = note1 % 12;
                      common = remain.detect({arg n; (n % 12) == intv});
                      if(common.notNil)
                      {result.add(common);
                       remain = remain.reject({arg n; n == common})} });
          ch1.do({arg note1;
                  var best, cand, baseNote, octaveShift;
                  if(result.collect({arg n; n%12}).includes(note1%12)==false)
                      {cand = remain.collect({arg n; octVar.(n)}).flat;
                       cand = cand.sortMap({arg cand; (cand - note1).abs});
                       best = cand.detect({arg cand;
                       remain.includes(cand%12 + (cand-(cand%12))) || true});
                       best = cand.first;
                       baseNote = remain.detect({arg n; (n%12)==(best%12) });
                       octaveShift = best - baseNote;
                       result.add(baseNote + octaveShift);
                       remain = remain.reject({arg n; n == baseNote });
                       } });
          result };
```

Listing 5.16: voice-leading common notes model function

Test it.

```
a = [~chord.(60, \maj7), ~chord.(71, \maj6)];
b = ~vLeadC.(a[0],a[1]);

~print.(a++[a[0]]++[b],[2]);
```

Test it with a chords sequence.

```
~prevSeq   = ~progClassic.collect({arg i;  ~voicing.(i, [-1,0,-1,0])});
~prevChord = ~prevSeq[0];        // First chord (midi notes)
~voicedSeq = [~prevChord];       // Put it in final sequence

~prevSeq.drop(1).do({arg nextChord; // Start from second chord
                     var voiced     = ~vLeadC.(~prevChord, nextChord);
                         ~voicedSeq = ~voicedSeq.add(voiced);
                         ~prevChord = voiced });
~print.(~prevSeq ++ ~voicedSeq,[4]);
```

By applying voicing, dropping, and voice-leading, we can generate variations in chord sequences while maintaining musical coherence and unity.

We can apply rhythms and all the variations illustrated above.

```
~prevSeq   = ~progJazzy.collect({arg i;  ~drop3.(i)});
~prevChord = ~prevSeq[0];
~voicedSeq = [~prevChord];

~prevSeq.drop(1).do({arg nextChord;
                      var voiced     = ~vLeadC.(~prevChord, nextChord);
                          ~voicedSeq = ~voicedSeq.add(voiced);
                          ~prevChord = voiced });
~rhythm = [4,16,8,16,8,8,4,4];
~print.(~prevSeq ++ ~voicedSeq,~rhythm);<<<
```

An example of further variations in the construction of a chord sequence can be represented by the use of a random-walk.

```
~chWalk = {arg chords, n=10;
           var idx = 0;
               n.collect({var ch, walk;
                              ch   = chords[idx];
                              walk = [ -1, 0, 1 ].wchoose([0.4, 0.2, 0.4])
                                                 .clip(0, chords.size-1);
                              idx  = idx + walk;
                              ch }) };
~voicedSeq = ~chWalk.(~progJazzy,25);
~rhy       = 10.collect({[16,8,16].scramble}).flat;
~print.(~voicedSeq,~rhy);
```

Listing 5.17: chord random walk model function

Melodies

In harmonic systems, melodic lines are based on chord progressions.

We must somehow "extract" a possible melody from a defined chord sequence.

To do this automatically, we can use various algorithms.

Once we have obtained the monodic arrays we can apply all the techniques illustrated previously for any kind of variations.

Top note The simplest.

```
~meloTop = ~progJazzy.collect({arg chord; chord.maxItem});
~exp     = [120, nil, nil,100, nil,  30, nil, nil, nil].scramble;
~print.(~meloTop++~progJazzy,[16],~exp);
```

Listing 5.18: top note melody model function

Random note The classic.

```
~meloRand = ~progJazzy.collect({arg chord; chord.choose});
~exp      = [120, nil, nil,100, nil,  30, nil, nil, nil];
~print.(~meloRand++~progJazzy,[16],~exp);
```

Listing 5.19: random note melody model function

Arpeggiating note The complete.

```
~meloArp = ~progJazzy.collect({arg chord; chord.scramble}).flat;
~exp     = [120, nil, nil,100, nil,  30, nil, nil, nil];
~print.(~meloArp++~progJazzy,[16],~exp, 96);
```

Listing 5.20: arpeggiating note melody model function

Voice-leading style A version that tries to move by small intervals into the next chord.

```
~vLeadMel = {arg chord, n = 10;
             var lastNote = 60, nextNote;
                 nextNote = chord.sort
                                 .minItem({arg n; (n-lastNote).abs});
                 lastNote = nextNote };
~melo = ~progJazzy.collect({arg ch; ~vLeadMel.(ch, ~noteForChord)}).flat;
~rhy  = 3.collect({[16,8,16].scramble}).flat;
~exp  = [120, nil, nil,100, nil,  30, nil, nil, nil].scramble;
~print.(~melo++~progJazzy,~rhy,~exp);
```

Listing 5.21: voice-leading style melody model function

Adding probabilistic movement around the “closest” tone.

It doesn’t always take the strict nearest note creating more variations.

```
~vLeadMel2 = {arg chord, n = 10;
              var lastNote = 60, nextNote;
                  if(0.8.coin)
                  {nextNote = chord.sort
                                  .minItem({arg n; (n-lastNote).abs})}
                  {nextNote = chord.choose};
                  lastNote = nextNote };
~melo = ~progJazzy.collect({arg ch; ~vLeadMel2.(ch, ~noteForChord) }).flat;
~rhy  = 3.collect({[16,8,16].scramble}).flat;
~exp  = [120, nil, nil,100, nil,  30, nil, nil, nil].scramble;
~print.(~melo++~progJazzy,~rhy,~exp, 80);
```

Listing 5.22: voice-leading 2 style melody model function

Passing notes between chords

If you want it to “walk” through stepwise notes between chord changes instead of jumping only to chord tones you can interpolate.

```
~melodyWalk = {arg chords, n = 10;
                var lastNote, idx;
                    lastNote = 60;
                    idx = 0;

    n.collect({var chord, nextChordNote,step,nextNote;
                chord = chords[idx];
                nextChordNote = chord.sort.minItem({arg n; (n-lastNote).abs});
                step = (nextChordNote - lastNote).sign;
                nextNote = if((lastNote != nextChordNote) and: {0.7.coin})
                            {lastNote + step }  // take step
                            {nextChordNote};
                lastNote = nextNote;
                idx = (idx + 1) % chords.size;
                nextNote })
               };
~prevSeq = ~progClassic.collect({arg i;  ~drop3.(i)}); // Starting sequence
~melo    = ~melodyWalk.(~prevSeq);
~rhy     = 3.collect({[16,8,16,4].scramble}).flat;
~exp     = [120, nil, nil,100, nil,  30, nil, nil, nil].scramble;
~print.(~melo++~prevSeq,~rhy,~exp, 52);
```

Listing 5.23: melodic-walk style model function

5.5 Composition sketches proposal

1. sketch a formal scheme (A B A or other) divided into sections and measures.
2. define a basic chord sequence (theme).
3. create a database of variations of the basic sequence, saving them as MIDI files.
   - diversify the vocal lines.
   - apply variations.
   - transpose it.
4. assemble the individual files in scoring software (MuseScore, Sibelius, or other) or in a MIDI sequencer according to the formal scheme in point 1.
5. extract melodies from the harmonic scheme and, if necessary, create melodic variations.
6. assemble everything in scoring software or sequencers to create the complete score.

Remember that chords can be tonal, a-tonal, jazzy, spectral, randomic, dodecaphonic, etc.

Chapter 6

From live set to laptop ensemble

6.1 Historical framework

The history of live sets closely follows technological evolution over the past few decades.

The same instruments, however, are used differently in different musical fields.

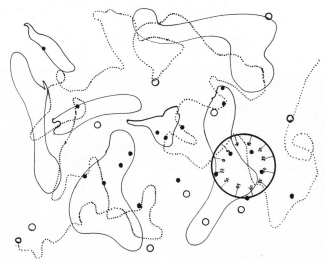
As has always been over the centuries we can consider two distinct aesthetic fields:

- experimental music.
- mainstream (popular) music.

Sometimes they intersect and contaminate each other but more often continue along their own aesthetic lines.

6.1.1 Experimental fields

One of the first works for live electronic performers without acoustic instruments can be considered Cartridge music (1960) by J.Cage - extract.



It is a composition for small amplified sounds, turntables and speakers.

Performers insert some small objects such as matches, wires, feathers and other things inside the metal case of the microphones (cartridge) contained in the old gramophones.

In the early 1960s Harold Bode followed by Bob Moog and Don Buchla developed the first commercial modular synthesizers based on the voltage-controlled modular synthesis.

Artists like John Mills-Cockel Marigolds (1973) - extract and Richard Teitelbaum - Antony Braxton Crossing (1977) - extract explored potentialities of these instruments through real-time manipulation.

Teitelbaum like many other musicians of the time mixed the sounds of these new instruments with acoustic instruments in the context of both classical and jazz improvisation.

The main experiences of this type are represented by instrumental collectives.

Let's see the most knowed ones.

Sonic Art Union (1966-76)

Gordon Mumma, David Behrman, Robert Ashley and Alvin Lucier.

Pioneers of the "circuit bending" concept they designed and built the instruments for their songs themselves using various sensors, microphones and circuitry (cybersonic systems).

In this way composers are the performers of their own works.

See Alvin Lucier Music for solo performer (1965) - extract.

AMM in London (1965)

Cornelius Cardew, Keith Rowe, Lou Gare e Eddie Prevost.

They explored a form of radical improvisation that was often highly convulsive, almost wild in character.

An AMM impro

Musica Elettronica Viva in Rome (1966)

Alvin Curran, Frederic Rzewski, Richard Teitelbaum and Allan Bryant.

The concept of collective improvisation is complemented by the audience's actions, creating a strongly social and political synthesis.

The concert becomes *a gathering place, a performance group, a resting place, and a school where old and young learn from each other and play together on the same stage.*

MEV Cosmic Communion (1970) - extract.

Karlheinz Stockhousen - Intuitive musik A form of musical improvisation based on instant creation in which fixed principles or rules may or may not have been given.

It is a type of process music where instead of a traditional music score, verbal or graphic instructions and ideas are provided to the performers.

Für kommende Zeiten - Watch (1970) - extract.

The score:

Star constellations

with common points

and falling stars

with secret wishes

and nocturnal forest

with dialogues

Abrupt end.

Following the experiences of improvised music between the late 1970s and early 1990s, new electronic instruments were frequently used by minimalist composers, both in ensembles and solo performances.

A prominent early example is the piece A Rainbow in Curved Air (1969) - extract by composer Terry Riley.

The late 1990s saw the development of smaller more affordable and compatible modular systems most notably the Eurorack standard and electronic keyboards.

An example of their use in mixed ensembles is given by the Philip Glass Ensemble Music in 12 parts - extract for soprano, electronic keyboards and winds.

This new generation of modular synths reignited interest and fostered intense experimentation and innovation transforming the modular format into a viable and popular choice for live performance.

Performers can simplify setups or build complex systems incorporating performance-oriented modules and adapting to different scenarios like solo gigs or collaborations.

Eurorack systems are designed for hands-on interaction allowing musicians to improvise and create unique sounds and textures with each live performance.

This new way of making electronic music encompasses much of the previous electroacoustic experimental musical experience:

- compositional prototyping of musical instruments.
- improvisation.
- iterative and drone music.
- freedom in the use of different sonic and musical languages.

A live solo example: Alberto Novello - Solo in a Silos (2007) - extract.

Since the 2000s, with the development of the potential offered by both hardware and software of personal computers, live sets have become almost exclusively digital, also including visual elements.

Ryoji Ikeda - Data Matrix (2010)

In recent years, several Laptop orchestras and ensembles have been formed in the music departments of major American universities, developing hardware and software research according to new performance needs.

SLOrk - Twilight (2013)

A further development in the use of the computer as a performative musical instrument is given by the practice of live coding, to which a chapter of this paper is dedicated.

Live coding impro (2020) - extract.

6.1.2 Mainstream fields

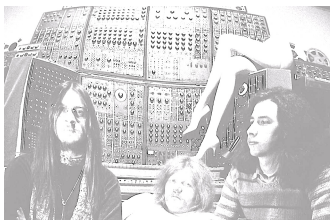
In the 1960s the duo Silver Apples from New York (synth and drums) took their homemade synthesizer (Simeon) on the road jamming on primitive oscillators.

They were one of the first groups to employ electronic music techniques extensively within a rock idiom.

As example: You and I (1969) - extract.

In the mid-70s the group Tangerine Dream had a more complex approach by integrating the first mammoth modular synth into their live sets.

As exemple Ricochet (1975) - extract.



Always in 1970 the album Kraftwerk - extract was released by the omonymous group founded in Düsseldorf by four students at the local conservatory.

They became the first laptop orchestra and would significantly influence the electronic music scene of the following years.

Live electronic performance also moved from theaters into clubs (DJing).

The disco era of the late 1970s

Patrick Cowley - Megatron Man radio edit (1981) - extract.

House music in the 1980s

Juan Atkins - I'm Your Audio Tech (1987) - extract.

The Techno era in 1990s

Aphex Twin - Cornish Acid (1996) - extract.

The Rave era (2000s)

Electronic music quit the clubs and populate big places and concerts.

Reinier Zonneveld - Live set (2023) - extract.

6.2 Interaction types

There are three main ways to control sound parameters:

- trigger.
- switch.
- continuos.

These ways are representative of different types of possible interaction.

Iterations with any type of device and with any communication protocol can be reduced to these modes.

6.2.1 Trigger

It can be described as pressing a button to ring the doorbell.

One action → do it!

In sound synthesis it is closely connected with envelopes without sustain.

Let's think of it as the action of plucking a string.

Look at it.

```
s.boot;
s.scope;
s.plotTree;

{Impulse.kr(2,0.5)}.plot(0.5);
```

Audio example.

```
SynthDef.new(\trig,{var sig, env;
                        env = Env.perc;
                        env = EnvGen.kr(env,doneAction:2);
                        sig = Pulse.ar([600,602],env);
                        sig = sig * env * 0.5;
                    Out.ar(0,sig)
            }).add;

Synth.new(\trig);
```

6.2.2 Switch

It can be described as pressing a button to switch on/off a light.

Two actions → two staes (on and off).

In sound synthesis it is closely connected with envelopes with sustain.

Let's think of it as the action of press and release an Organ key.

Look at it.

```
{Trig1.kr(Impulse.kr(2,0.7),0.2)}.plot(0.5);
```

Audio example.

```
SynthDef.new(\switch, {arg gate=0;
                        var sig, env;
                            env = Env.adsr;
                            env = EnvGen.kr(env,gate,doneAction:2);
                            sig = VarSaw.ar([1300,1310],env);
                            sig = sig * env * 0.5;
                        Out.ar(0,sig)
            }).add;

a = Synth.new(\switch, [\gate,1]);
a.set(\gate,0);
```

6.2.3 Continuos

It can be described as gradually increasing or decreasing the intensity of a light through a knob.

One actions → two staes (movement and stop).

In sound synthesis it is closely connected with interpolation and ramps.

Let's think of it as the action of crescendo, diminuendo or glissato.

Look at it.

```
{LFNoise2.kr(5)}.plot(0.5);
```

Audio example.

```
SynthDef.new(\cont,{arg gate=0;
                    var sig, frq, env;
                        env = Env.adsr;
                        env = EnvGen.kr(env,gate,doneAction:2);
                        frq = LFNoise1.kr(3).range(990,1100);
                        sig = Saw.ar([frq,frq+5]);
                        sig = sig * env * 0.5;
```

```
                        Out.ar(0,sig)
              }).add;

a = Synth.new(\cont, [\gate,1]);
a.set(\gate,0);
```

6.2.4 Conversions

We can move from one type to another through three operations:

- smoothing.
- sample and hold.
- thresholds.

Smoothing

From discrete values (trigger or switch) to continuos (ramp).

In SuperCollider we have two basic methods.

1. *.lag(0.2* → logarithmic ramp.

```
{[Impulse.kr(5),
  Impulse.kr(5).lag(0.2),
  LFNoise0.kr(5).lag(0.2)
]}.plot(1);
```

   Audio example

```
SynthDef.new(\lag, {arg amp=0,lagt=0.2, pan=0;
                      var sig;
                          sig = WhiteNoise.ar * amp.lag(lagt); // lag
                          sig = Pan2.ar(sig, pan.varlag(lagt));// varlag
                      Out.ar(0,sig)
              }).add;
a = Synth.new(\lag,[\lagt,3,\amp,0.5]);
a.set(\lagt,3,\amp,rand(0.8));          // Change lag time
a.free;
```

2. *.varlag(0.2* → linear ramp.

```
{[Impulse.kr(5),
  (WhiteNoise.kr * Impulse.kr(5)).varlag(0.2),
  LFNoise0.kr(5).varlag(0.2)
]}.plot(1);
```

   Varlag is a complex object see help file for more.
   Test it on Pan argument with previuos Synth.

```
a = Synth.new(\lag,[\lagt,3,\amp,0.3]);
a.set(\lagt,0.5,\pan,rand2(1.0));
a.free;
```

All continuos input values from any type of port and protocol (MIDI, OSC, HID, Serial, etc.) should be smoothed to avoid clicks.

Sample and hold

From continuos signals to trigger or switch.

In SuperCollider we can use UGens 'Trig1' or 'Trig'.

Trig when a bipolar signal goes from negative to positive.

Second argument is dead zone (duty cycle duration) in seconds.

After this time goes to zero.

See Help files for differences.

```
{[SinOsc.kr(3),
  Trig1.kr(SinOsc.kr(3), 0.25)
]}.plot(1);
```

Test it with mouse x position between -1 and 1.

```
SynthDef.new(\trig,{arg amp=0, sustime=0.5;
                     var ksig, env, sig;
                         ksig = MouseX.kr(-1,1);
                         ksig = Trig1.kr(ksig, sustime);
                         env  = Env.adsr;
                         env  = EnvGen.kr(env,ksig);
                         sig  = Blip.ar(600,20) * env * amp;
                         sig  = Pan2.ar(sig, 0);
                     Out.ar(0,sig)
            }).add;

a = Synth.new(\trig, [\amp,0.5]);
a.set(\sustime,0.1);
a.free;
```

If we set duty cycle at 1/samplerate → trig.

```
{[SinOsc.kr(3),
  Trig1.kr(SinOsc.kr(3), 1/44100)
]}.plot(1);
```

Thresholds

We can also set thresholds and generate:

- signal values > threshold → 1.0 (and vceversa).
- signal values < threshold → 1.0 (and viceversa).
- signal values pass threshold up and down.
- signal values are within a range.

We test it with mouse x position as control signal but is valid for any kind of signal.

Four audio exemples.

1. Trigger envelope without sustain when pass up.

```
SynthDef.new(\thresh1, {arg amp=0;
                        var ksig, env, sig;
                            ksig = MouseX.kr(0,10) > 5;
                            env  = Env.perc;              // no sustain
                            env  = EnvGen.kr(env,ksig);
                            sig  = Formant.ar * env * amp;
                            sig  = Pan2.ar(sig, 0);
                        Out.ar(0,sig)
            }).add;
a = Synth.new(\thresh1, [\amp,0.5]);
a.free;
```

2. Trigger envelope without sustain when pass threshold up or down.

```
SynthDef.new(\thresh2, {arg amp=0;
                        var ksig, env, sig;
                            ksig = Changed.kr(MouseX.kr(0,10) > 5);
                            env  = Env.perc;              // no sustain
                            env  = EnvGen.kr(env,ksig);
                            sig  = PinkNoise.ar * env * amp;
                            sig  = Pan2.ar(sig, 0);
                        Out.ar(0,sig)
            }).add;
a = Synth.new(\thresh2, [\amp,0.5]);
a.free;
```

3. Trigger envelope with sustain when pass up (noteON) and down (noteOff).

```
SynthDef.new(\thresh3, {arg amp=0;
                        var ksig, env, sig;
                            ksig = MouseX.kr(0,10) > 5;
                            env  = Env.asr;               // with sustain
                            env  = EnvGen.kr(env,ksig);
                            sig  = GrayNoise.ar * env * amp;
                            sig  = Pan2.ar(sig, 0);
                        Out.ar(0,sig)
            }).add;
a = Synth.new(\thresh3, [\amp,0.5]);
a.free;
```

4. Trigger envelope with sustain when the signal values are within a range.

```
SynthDef.new(\thresh4, {arg amp=0;
                        var ksig, env, sig;
                            ksig = InRange.kr(MouseX.kr(0,1),0.2,0.8);
                            env  = Env.asr;               // with sustain
                            env  = EnvGen.kr(env,ksig);
                            sig  = Pluck.ar(WhiteNoise.ar(0.1),
                                            Impulse.ar(2),
                                            1/440,1/440,10
                                            ) * env * amp;
                            sig  = Pan2.ar(sig, 0);
                        Out.ar(0,sig)
            }).add;

a = Synth.new(\thresh4, [\amp,0.5]);
a.free;
```

More UGens about triggers.

We will apply one or more of these types of interaction depending on the characteristics of the external devices and the communication protocols we will use.

6.3 Control signals

Exemples of the previous paragraph are developed with control signals.

A control signal is a signal that shape an audio signal at a low frequency rate (LF $<$ 20Hz)).

In SuperCollider they can be UGens:

- at audio rate (.ar).
- at control rate (.kr) $\rightarrow$ generate one sample value for every sixty-four audio sample values.

We can convert the signal rate.

```
{[WhiteNoise.kr,  A2K.kr(WhiteNoise.ar)]}.plot;  // Audio --> control
{[WhiteNoise.ar,  K2A.ar(WhiteNoise.kr)]}.plot;  // Control --> audio
```

6.3.1 Types

There are three type of signals that correspond to the types of interaction:

- impulsive (trigger).
- discrete (switch).
- continuos (continuos).

```
{[ Impulse.kr(10), LFNoise0.kr(10), LFNoise2.kr(10) ]}.plot(0.5)
```

Each of them can be:

- periodic.
- a-periodic.

```
{[ Impulse.kr(100), Dust.kr(100), LFPulse.kr(10),
   LFNoise0.kr(10), LFPar.kr(10), LFNoise2.kr(10) ]}.plot(1)
```

6.3.2 Scaling

An important aspect when we use control signals is change its original range according values of controlled parameter.

In SuperCollider three main strategies:

1. if the original range is +/- 1 as for audio signals
   - *.range(min,max).*
   - *.unipolar.*

   ```
   {[SinOsc.ar(10),SinOsc.ar(10).range(20,30),SinOsc.ar(10).unipolar]}.plot(1)
   ```

2. if the original range is custom:
   - *.linlin(minIn.maxIn,minOut,maxOut).*
   - *.linexp(minIn.maxIn,minOut,maxOut).*

   ...and their vairants (see help files).

   ```
   {[EnvGen.kr(Env([20,30,25,27],[0.2,0.2,0.2])),
     EnvGen.kr(Env([20,30,25,27],[0.2,0.2,0.2])).linlin(20,30,50,80)
   ]}.plot(0.6)
   ```

3. using last two UGens aguments (*mul* and *add*).

   ```
   {LFSaw.kr(5,mul:20,add:50)}.plot(1)   // from 30 to 70
   ```

6.3.3 Plugging

Two ways to plug control signals to parameters.

1. inside SynthDef

   ```
   SynthDef(\ksig1 , {arg freq=900,dur=0.2,atk=0.01,pan=0,gate=0;
                      var sig, bps, dec, trig, env, fade;
                          sig  = Saw.ar(freq);
                          bps  = dur.reciprocal;
                          dec  = 1-atk * dur;
                          trig = Impulse.ar(bps); // control signal...ar
   ```

```
                        env  = Decay2.ar(trig,atk,dec);
                        fade = Linen.kr(gate,doneAction:2);
                        sig  = sig * env * fade;
                        sig  = Pan2.ar(sig,pan);
                  Out.ar(0, sig)
       }).add;
a = Synth(\ksig1, [\gate,1]);
a.set(\dur,rrand(0.05,0.8));
a.set(\gate,0);
```

2. Through control bus.

```
SynthDef(\ksig, {arg type=0,freq=3,bus=0;
                  var ksig;
                  ksig = Select.kr(type,
                                   [
                                    Dust2.kr(freq),     // 0
                                    LFNoise0.kr(freq), // 1
                                    LFNoise2.kr(freq)  // 2
                                    ]);
                  Out.kr(bus, ksig)
         }).add;

SynthDef(\asig, {arg freq=0, amp=0, pan=0, gate=0;
                  var sig, fade;
                      sig  = SinOsc.ar(freq.range(800,1200)); // scaled
                      fade = Linen.kr(gate,doneAction:2);
                      sig  = sig * fade;
                      sig  = Pan2.ar(sig, pan);               // not scaled
                  Out.ar(0, sig)
}).add;

~kbus = Bus.control(s, 1);

a = Synth(\ksig, [\bus, ~kbus]);
b = Synth(\asig, [\freq,~kbus.asMap, \pan,~kbus.asMap, \amp,0.5, \gate,1]);
a.set(\type,2);
b.set(\gate,0); a.free; ~kbus.free;
```

6.4 MIDI

Musical Instrument Digital Interface.

Born on 1982.

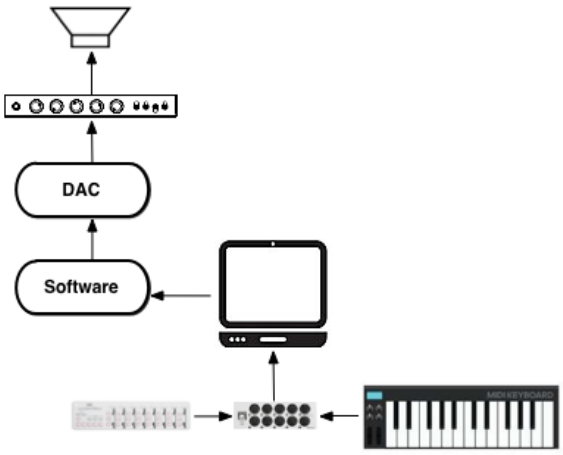
It includes:

- a communication procotol.
- digital interfaces.
- electrical connectors (cables and ports).

We can connect electronic musical instruments, audio devices and computers with midi cables.

Serial transmission.

Midi message values → int from 0 to 127.



Initialize MIDI communication in SuperCollider.

```
MIDIClient.init;
```

Searching for physical and virtual midi devices connected to the computer.

```
MIDIClient.sources;      // from
MIDIClient.destinations; // to
```

- ports → virtual and physical connectors.
- channels
  - 16 for each port (as empty staff of a score).
  - one channel can be mono or poli.
- voice → instrument (timbre) that we assign to a channel.
- message → can be channel message or system message.

More on MIDI protocol.

6.4.1 Receive

Connect a MIDI device to the computer USB port (or a software to a virtual port).

1. disconnect all MIDI message connections from/to SuperCollider (clean up).
2. connect all MIDI message connections from/to SuperCollider.

```
MIDIIn.disconnectAll;
MIDIIn.connectAll;
```

3. check communication and the incoming message type.

```
MIDIFunc.trace(true);  // check message (Post window)
MIDIFunc.trace(false); // Stop checking
```

4. choose the MIDI message type we want to receive.

```
MIDIdef.noteOn( \noteOn, {arg ...args; args.postln});
                         // velocity, note,      channel, uid
MIDIdef.noteOff(\noteOff,{arg ...args; args.postln});
                         // velocity, note,      channel, uid
MIDIdef.cc(     \control,{arg ...args; args.postln});
                         // value,   cc number,  channel, uid
```

MIDIdef second argument ia a function that is evaluated each time a midi message is incoming.

Its argument is an array containing values from incoming message.

The values type depends on MIDI message type.

Main MIDI message types → *.noteOn, .noteOff, .cc, .polytouch, .touch, .bend, .program, .sysex, .smpte, .sysrt.*

If you need more open *MIDIdef* help file.

Kill one *MIDIdef* or all.

```
MIDIdef(\noteOn).free; // Cancella una specifica MIDIdef
MIDIdef.freeAll;       // Cancella tutte le MIDIdef
```

If we want receive messages from a specific channel or control number.

```
MIDIdef.cc(\control_0,{arg val; val.postln});    // from all
MIDIdef.cc(\control_1,{arg val; val.postln},1);  // only from cc 1 chan 0
MIDIdef.cc(\control_2,{arg val; val.postln},2);  // only from cc 2 chan 0
MIDIdef.cc(\control_3,{arg val; val.postln},3);  // only from cc 3 chan 0
MIDIdef.cc(\control_1,{arg val; val.postln},1,0);  // only from cc 1 chan 0
MIDIdef.cc(\control_2,{arg val; val.postln},1,1);  // only from cc 1 chan 1
MIDIdef.cc(\control_3,{arg val; val.postln},1,2);  // only from cc 1 chan 2
```

6.4.2 Send

1. search for available MIDI device destinations (Array of devices).
2. choose and set destination by index.
3. send MIDI messages.

```
MIDIClient.destinations;
c = MIDIOut.new(0);
//       channel,    pitch, velocity
c.noteOn(0,      rand(127), rand(127));
c.noteOff(0,     rand(127), rand(127));
c.allNotesOff(0);
//       channel, ccn,  val
c.control(0,        1,  rand(127));
```

6.4.3 Voice allocation

There are three different Synth models designed for incoming midi message:

- monophonic.
- poliphonic without sustain envelope (one message trigger + duration).
- poliphonic with sustain envelope (two message switch - noteOn and noteOff).

They are usually triggered by keys or buttons but we can also control parameters with continuos signals from knobs, fader, etc.

Monophonic synth

```
s.boot;
MIDIIn.disconnectAll;
MIDIIn.connectAll;
```

The model

```
SynthDef(\midi, {arg freq=440, amp=0, gate=0, gain=0, pan=0;
                 var sig, env;
                     sig = SinOsc.ar(freq);
                     env = Env.adsr;
```

```
                        env = EnvGen.kr(env,gate);
                        sig = sig * env * gain.lag(0.2); // lag
                        sig = Pan2.ar(sig,pan);
                    Out.ar(0, sig)
            }).add;

MIDIdef.cc(\masterOut, {arg val; // only first argument (cc value)
                          ~synth.set(\gain,val/127) }); // master out volume
MIDIdef.noteOn(\noteOn,{arg vel,note;
                          ~synth.set(\freq,note.midicps, // MIDI to Hz
                                     \amp,vel/127,       // scaled amplitude
                                     \gate, 1)} );       // noteOn

MIDIdef.noteOff(\noteOff,{~synth.set(\gate,0)});       // note off

~synth = Synth(\midi);
~port = MIDIOut.new(0);
~port.noteOn(0,  72, rand(127));
~port.control(0,  1,  64);
~port.control(0,  1,  rand(127));
~port.noteOff(0,     72, 0);
~synth.free; ~port.free; MIDIdef.freeAll;
```

Listing 6.1: MIDI monophonic instrument model

Poliphonic synth without sustain envelope

Dynamic voice allocation.

Every time a MIDI message is received a new Synth instance is created.

After a defined time (duration) the instance is killed automatically (*doneAction:2*).

Note off messages are not necessary (*t_gate*).

The model.

```
SynthDef(\midi_poli1, {arg freq=440, amp=0, dur=0.5, pan=0, t_gate=0;
                        var sig,env;
                            sig = Blip.ar(freq,40);
                            env = Env.perc(dur * 0.1,dur * 0.9);
                            env = EnvGen.kr(env,t_gate, doneAction:2);
                            sig = sig * env * amp;
                            sig = Pan2.ar(sig,pan);
                        Out.ar(0, sig)
          }).add;

MIDIdef.noteOn(\noteOn1,{arg vel, note;
                          Synth(\midi_poli1,[\freq,note.midicps,
                                             \amp,vel/127,
                                             \dur,1.5,
                                             \pan, rand2(1.0),
                                             \t_gate,1])
                }))

~synth = Synth(\midi);

~port = MIDIOut.new(0);
~port.noteOn(0,  rrand(85,90), rand(127));
~port.free; MIDIdef.freeAll;
```

Listing 6.2: MIDI poliphonic no-sustain instrument model

Poliphonic synth with sustain envelope

Static voice allocation.

We define an empty array with 128 items.

MIDI note → array index (from 0 to 127).

Note On message → put a Synth instance at index (pitch).

Note Off message → send *gate 0* to instance at index (pitch) and substitute the Synth with *nil*.

```
SynthDef(\midi_poli2, {arg freq=789, amp=0, pan=0, gate=0;
                        var sig, env;
                            sig = SinOsc.ar(freq);
                            env = Env.adsr(0.01,0.3,0.5,1);
                            env = EnvGen.kr(env, gate, doneAction:2);
                            sig = sig * env * amp;
                            sig = Pan2.ar(sig, pan);
                        Out.ar(0, sig)
          }).add;

~note = Array.newClear(128); // Empty array with 128 items (all MIDI notes)
                             // All elements are 'nil'

MIDIdef.noteOn(\noteOn1,{arg vel, pitch;
                        ~note.put(pitch,
                                    Synth(\midi_poli2,[\freq,pitch.midicps,
                                                       \amp,vel/127,
                                                       \gate,1]))
               });

MIDIdef.noteOff(\noteOff1,{arg vel, pitch;
                          ~note.at(pitch).set(\gate,0);
                          ~note.put(pitch, nil)
               });

~synth = Synth(\midi);

~port = MIDIOut.new(0);
~port.noteOn(0, rrand(80,90).postln, rand(40));
~port.noteOff(0, rrand(80,90), 0);
~port.allNotesOff(0); ~port.free; ~note.free; MIDIdef.freeAll;
```

Listing 6.3: MIDI poliphonic w-sustain instrument model

6.5 OSC

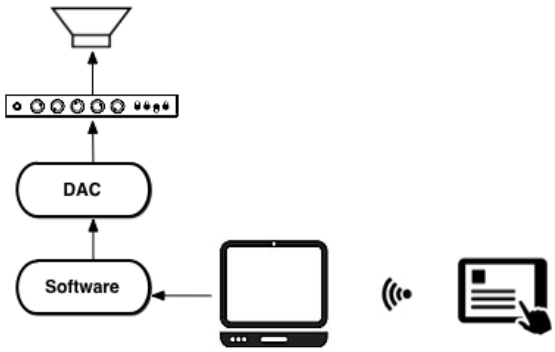
Open Sound Control.

Born on 1997 CNMAT - Berkeley Univerity.

- communication procotol.
- wireless.

We can connect electronic musical instruments, audio devices and computers without cables.

OSC message data types → int32, float32 and strings.



It is the protocol used in SuperCollider for communication between Interpreter and Server.

There are many smartphone and tablet app that support it.

You can use these devices as musical instrument controlling sound synthesi and processing in SuperCollider.

TouchOsc is a cheaper and powerful app.

FaceOsc can be useful to check communication between software in local machine.

There are not physical connections then we have to open one or more communication channels among devices and/or software setting an:

- IP address like "127.0.0.1" → it uniquely identify a device.
- OSC port → an integer value that identify a monodirectional channel.

Each IP address can have many ports, usually for send and receive messages.

6.5.1 Receive

Check communication and the incoming message type.

Server should be quit because we can monitor messages from Interpreter.

```
OSCFunc.trace(true);  // Check OSC messages
OSCFunc.trace(false); // Stop checking
```

Messages are identified by a label starting with slash symbol.

Syntax is similar to the *MIDIdef* class.

```
//              name,    function,             label        port
OSCdef.new(\criccio, {arg msg; msg.postln}, '/x',        recvPort:8001);
OSCdef.new(\ciaccio, {arg msg; msg.postln}, '/y',        recvPort:8001);
OSCdef.new(\ciuccio, {arg msg; msg.postln}, '/z',        recvPort:8001);
OSCdef.new(\cieccio, {arg msg; msg.postln}, '/1/fader1',recvPort:8001);

OSCdef(\criccio).free;
OSCdef.freeAll;
```

6.5.2 Send

Set a device Net address.

Send messages to it.

```
c = NetAddr("127.0.0.0",  9000); // local address

c.sendMsg("/fade1", rand(1.0));
c.sendMsg("/fade1", rand(1.0), rand(1.0), rand(1.0)); // send more than one
    value (array)
```

Check communication in local.

A send/receive model.

```
SynthDef(\osc, {arg freq=400,amp=0,pan=0,gate=0;
                var sig, env;
                    env = Linen.kr(gate,doneAction:2);
                    sig = SinOsc.ar(freq);
                    sig = sig * env * amp;
                    sig = Pan2.ar(sig, pan);
                Out.ar(0,sig)
                }).add;

OSCdef.new(\gate, {arg msg;
                   a.set(\gate,msg[1]) },
                   '/gate', recvPort:8000);

OSCdef.new(\osck, {arg msg;
                   msg.postln;
                   a.set(\freq,msg[1],\amp,msg[2],\pan,msg[3]) },
                   '/osck', recvPort:8000);

c = NetAddr("127.0.0.1",  8000);
a = Synth(\osc,[\amp,0.5]);

c.sendMsg("/gate", 1);
c.sendMsg("/gate", 0);
c.free;
```

6.6 HID

Human Interface Device.

Devices designed for human-machine interaction → mouse, keyboard, joystick, graphic tablet, etc.

In SuperCollider two ways to receive values:

- Client side → interaction in Interpreter (as MIDI and OSC).
- Server side → interaztion in Server (signals at control rate).

6.6.1 Mouse

All the three control typologies:

- trigger.
- switch.
- continuos.

Client side

We must create a GUI window and interact with it.

Two reasons:

- avoid unwanted interactions.
- delimit a space (range) for values.

Different methods for different actions.

.mouseDownAction → as for *MIDIdef* and *OSCdef* the function is evaluated each time a mouse button is clicked.

...args:

- 0 wiew → the window clicked by mouse.
- 1 x → x position on widow in pixels (0 = leftmost).
- 2 y → y position on window in pixels (0 = higher).
- 3 modifiers → if a modifier key is pressed.
- 4 botton n → botton number clicked (0=left, 1=right, 2=central).
- 5 counter → number of clicks (1=single click, 2 double click).

```
w = Window.new("mouse");
w.view.mouseDownAction_({arg ...args;
                                 args.postln;     // all info
                                 args[1].postln; // only x position
                        });
w.front;
w.alwaysOnTop_(true);
w.onClose_({w.free});
```

Listing 6.4: mouseDownAction model

Audio example.

- left click → trigger sine.
- right click → trigger noise.

```
SynthDef.new(\sine, {arg freq=0, amp=0, t_gate=0;
                      var sig,env;
                          sig = SinOsc.ar(freq);
                          env = EnvGen.kr(Env.perc,t_gate,doneAction:2);
                      Out.ar(0,sig*env*amp);
              }).add;
SynthDef.new(\noise, {arg amp=0;
                       var sig;
                           sig = PinkNoise.ar;
                       Out.ar(0,sig*amp.lag(3));
              }).add;

w = Window.new("mouse");
w.view.mouseDownAction_({arg ...args;
                          if(args[4] == 0,     // if...
                          {Synth.new(\sine,
                                    [\freq, args[1].linlin(0,400,300,2000),
                                     \amp,1-args[2].linlin(0,400,0,1.0),
                                     \t_gate, 1])},
                      {a.set(\amp, 1-args[2].linlin(0,400,0,1.0))})
                          });
w.front;
w.alwaysOnTop_(true);
w.onClose_({w.free;s.freeAll});

a = Synth.new(\noise);
```

.mouseUpAction → as previous but function is evaluated when button is released.

```
w = Window.new("mouse");
w.view.mouseUpAction_({arg ...args;
                                 args.postln;     // all info
                                 args[1].postln; // only x position
                        });
w.front;
w.alwaysOnTop_(true);
w.onClose_({w.free});
```

Listing 6.5: mouseUpAction model

Audio example.

- left click → trigger sine (trigger).
- right click → fade in noise (switch).
- right release→ fade out noise.

```
SynthDef.new(\sine, {arg freq=0, amp=0, t_gate=0;
                      var sig,env;
                          sig = SinOsc.ar(freq);
                          env = EnvGen.kr(Env.perc,t_gate,doneAction:2);
                      Out.ar(0,sig*env*amp);
              }).add;
SynthDef.new(\noise, {arg amp=0;
                       var sig;
                           sig = PinkNoise.ar;
                       Out.ar(0,sig*amp.lag(0.2)); // fade di 0.2 secondi
              }).add;

w = Window.new("mouse");
w.view.mouseDownAction_({arg ...args;
                          if(args[4] == 0,
                          {Synth.new(\sine,
                                    [\freq, args[1].linlin(0,400,300,2000),
                                     \amp,1-args[2].linlin(0,400,0,1.0),
                                     \t_gate, 1])},
                      {a.set(\amp, args[2].linlin(0,400,0,1.0))}) // note On
                          });
w.view.mouseUpAction_({arg ...args;
                                 args.postln;
                                 a.set(\amp, 0) // note Off
                        });
w.front;
w.alwaysOnTop_(true);
w.onClose_({w.free;s.freeAll});

a = Synth.new(\noise)
```

.mouseMoveAction → as previous but if we click and draw values change dynamically in a continuos way.

Top-left corner → (0, 0)

Out of window positions → negative or positive values.

```
w = Window.new("mouse");
w.view.mouseMoveAction_({arg ...args;
                                   args.postln;     // all info
                                   args[1].postln; // only x position
                        });
```

```
w.front;
w.alwaysOnTop_(true);
w.onClose_({w.free});
```

Listing 6.6: mouseMoveAction model

Audio example.

- click and release → switch (noteOn and noteOff).
- draw → continuos values.

```
SynthDef.new(\spectra, {arg freq=0, amp=0, harm=1, gate=0;
                        var sig,env;
                           sig = Blip.ar(freq,harm);
                           env = EnvGen.kr(Env.asr(0.3,1,5),gate,doneAction:2);
                        Out.ar(0,sig*env*amp);
             }).add;

w = Window.new("mouse");
w.view.mouseDownAction_({arg ...args;                    // note On
                         a = Synth.new(\spectra,
                                      [\freq, args[1].linlin(0,400,300,2000),
                                       \gate, 1]);
                         });
w.view.mouseMoveAction_({arg ...args;                   // continuos signal
                       a.set(\amp,1-args[2].linlin(0,400,0,1.0),
                           \harm, args[1].linlin(0,400,1,20))
                         });
w.view.mouseUpAction_({arg ...args;
                       a.set(\gate,0)                    // note Off
                      });
w.front;
w.alwaysOnTop_(true);
w.onClose_({w.free;s.freeAll});
```

.mouseOverAction → as previous but without click (passing mouse on the window).

Top-left corner → (0, 0)

Out of window positions → negative or positive values.

```
w = Window.new("mouse");
w.view.mouseOverAction_({arg ...args;
                                 args.postln;     // all info
                                 args[1].postln; // only x position
                       });
w.front;
w.alwaysOnTop_(true);
w.onClose_({w.free});
```

Listing 6.7: mouseOverAction model

Audio example.

```
b = Buffer.read(s, "/absolute/path/to/file.wav");
SynthDef.new(\scratch, {arg pos=0, amp=0, buf=0, smooth=0.2;
                        var sig, punt;
                            punt = BufFrames.ir(buf)*K2A.ar(pos).varlag(smooth)
    ;
                            sig  = BufRd.ar(1,b,punt);
                        Out.ar(0,sig*amp.lag(0.2))
              }).add;
```

```

w = Window.new("mouse").acceptsMouseOver_(true);
w.view.mouseOverAction_({arg ...args;
                          a.set(\pos, args[1].linlin(0,400,0,1),     // x pos
                                \amp,1-args[2].linlin(0,400,0,1.0)) // y pos
                        });
w.front;
w.alwaysOnTop_(true);
w.onClose_({w.free;a.free;b.free});

a = Synth.new(\scratch, [\buf,b]);
```

Server side

It's not necessary a GUI window.

Screen boundaries.

Three UGens:

- *MouseX.kr* → min, max, curve, lag time.
- *MouseY.kr* → min, max, curve, lag time.
- *MouseButton* → min, max, lag time.

```
{[MouseX.kr(0,1,0,0.2), MouseX.kr(0.0001,1,1,0.2)]}.scope; // lin vs exp
{[MouseY.kr(0,1,0,5),   MouseY.kr(0.0001,1,1,5)]}.scope;   // 5 sec lag time
{[MouseButton.kr(0,1,0.1),   MouseButton.kr(0,1,)]}.scope; // 3 sec lag time
```

Listing 6.8: mouse controls Server model

Audio example.

```
b = Buffer.read(s, "/absolute/path/to/file.wav");
SynthDef.new(\scratch, {arg gate=1, buf;
                        var sig, speed, env;
                            env   = Env.new([0,1,0],
                                            [0.1, 0.1],
                                            \sin, 1).kr(20,gate);
                            speed = MouseX.kr(-10, 10);
                            speed = speed - DelayN.kr(speed, 0.1, 0.1);
                            speed = MouseButton.kr(1, 0, 0.3) + speed ;
                            sig   = PlayBuf.ar(1, buf,
                                               speed * BufRateScale.kr(buf),
                                               loop: 1);
                        Out.ar(0, sig * env );
              }).add;

a = Synth.new(\scratch, [\buf, b]);
a.set(\gate, 0); a.free; b.free;
```

6.6.2 Keyboard

Only two control typologies:

- trigger → press and release the key quickly. .
- switch → press and release.

Client side

We must create a GUI window and interact with it.

The reason is avoid unwanted code writing.

Different methods for different actions.

.keyDownAction → as for *MIDIdef* and *OSCdef* the function is evaluated each time a key is clicked.

...args:

- 0 wiew → the window clicked by mouse.
- 1 char → pressed character.
- 2 modifiers → if a modifier key is pressed.
- 3 unicode → unicode value (ASCII).
- 4 keycode → keycode value (JavaScript).
- 5 key → deprecated.

```
w = Window.new("key");
w.view.keyDownAction_({arg ...args;
                              args.postln;
                              args[1].postln
                       });
w.front;
w.alwaysOnTop_(true);
w.onClose_({w.free})
```

Listing 6.9: keyDownAction model

Audio example (trigger).

```
SynthDef.new(\if_1, {arg t_gate=0;
                     var sig, env;
                         sig = Saw.ar([600,602]);
                         env = EnvGen.ar(Env.perc,t_gate,doneAction:2);
                     Out.ar(0,sig*env);
         }).add;

w = Window.new("key");
w.view.keyDownAction_({arg ...args;
                           if(args[1]==$q, {Synth(\if_1, [\t_gate,1])})
                       });
w.front;
w.alwaysOnTop_(true);
w.onClose_({w.free});
```

.keyUpAction → as previous but function is evaluated when key is released.

```
w = Window.new("key");
w.view.keyUpAction_({arg ...args;
                            args.postln;
                            args[1].postln
                       });
w.front;
w.alwaysOnTop_(true);
w.onClose_({w.free})
```

Listing 6.10: keyUpAction model

If we hold down a key we obtain redundant data.

```
SynthDef.new(\if_1, {arg t_gate=0;
                     var sig, env;
```

```
                        sig = Saw.ar;
                        env = EnvGen.ar(Env.perc,t_gate);
                    Out.ar(0,sig*env);
            }).add;

w = Window.new("key");
w.view.keyDownAction_({arg ...args;
                    if(args[1]==$q, {a.set(\t_gate,1)}); // 'q' = Note On
                      });
w.view.keyUpAction_({arg ...args;
                  if(args[1]==$q,{a.set(\t_gate,0)});    // Note Off
                    });
w.front;
w.alwaysOnTop_(true);
w.onClose_({w.free;a.free});
a = Synth.new(\if_1);
```

To avoid → filter incoming data.

```
var ora, prec = 0;                          // variable init
w = Window.new("key");
w.view.keyDownAction_({arg ...args;
                        ora = args[3];
                        if((ora != prec), // if different from previuos
                        {"noteOn".postln; // play and
                         prec = ora });   // assign 'prec' to last received value
                        });

w.view.keyUpAction_({
                    prec = 0;          // reset Array
                    "noteOff".postln; // Action
                    });
w.front;
w.alwaysOnTop_(true);
w.onClose_({w.free})
```

Audio example (switch).

```
SynthDef.new(\if_1, {arg gate=0;
                    var sig, env;
                        sig = Saw.ar;
                        env = EnvGen.ar(Env.adsr,gate,doneAction:2);
                    Out.ar(0,sig*env);
            }).add;

~ora;
~prec = 0;
w     = Window.new("key");
w.view.keyDownAction_({arg ...args;   // 'q' = Note On
                            ~ora = args[1];
                            if(        // if = 'q' AND != from previous
                            (~ora == $q).and(~ora != ~prec),
                            {a = Synth.new(\if_1, [\gate,1]);
                            ~prec = ~ora})
                        });
w.view.keyUpAction_({                             // Note Off
                    if(~ora==$q, {a.set(\gate,0);
                                  ~prec = 0});  // Reset
                    });
w.front;
w.alwaysOnTop_(true);
w.onClose_({w.free;a.free});
```

Server side

It's not necessary a GUI window (but suggested).

One UGen:

- *KeyStart.kr* → min, max, lag time.

We must know keycode in advance.

```
w = Window.new;
w.front;
SynthDef.new(\key, {var amp,sig;
                        amp = KeyState.kr(38,0,1.0);   // 'j' key
                        sig = SinOsc.ar * amp;
                    Out.ar(0,sig)
            }).add;

Synth.new(\key);
```

Listing 6.11: key control Server model

6.7 Live playback and synthesis

If we want playback sound files or trigger synthesis sounds or sequences in a perforance we can:

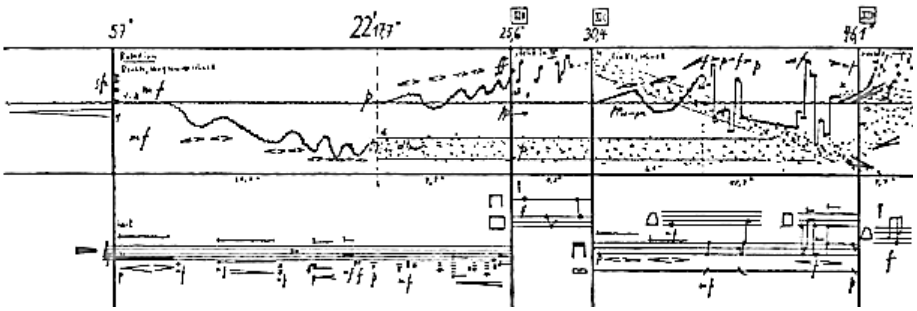
- one-shot → play it from beginning to end (old school).
- map key (HID, MIDI or OSC) → event (soundfile or other).
  - key → trig soundfile or Synth.
  - key → set playback rate (sampler-like transposition) or other parameters.
  - key → set start position (and duration) in a soundfile.
- play it in a sequential way (cues + counter).
  - cue → trig soundfile or Synth.
  - cue → set start position (and duration) in a soundfile or other parameters.

6.7.1 One shot

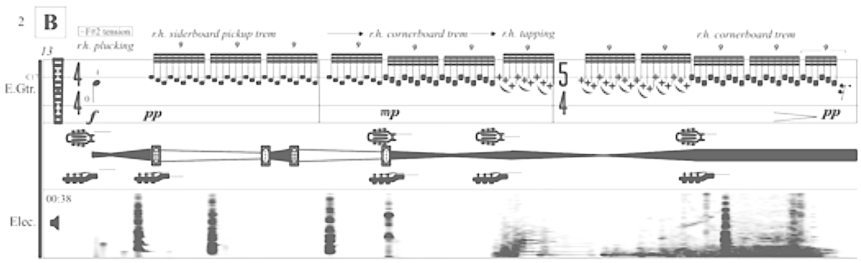
The old tape playback.

If we want to sync it with live player parts we need to write it on the score in two ways:

1. a seconds grid (chronometric way).



2. a graphic representation staff (waveform or sonogram).



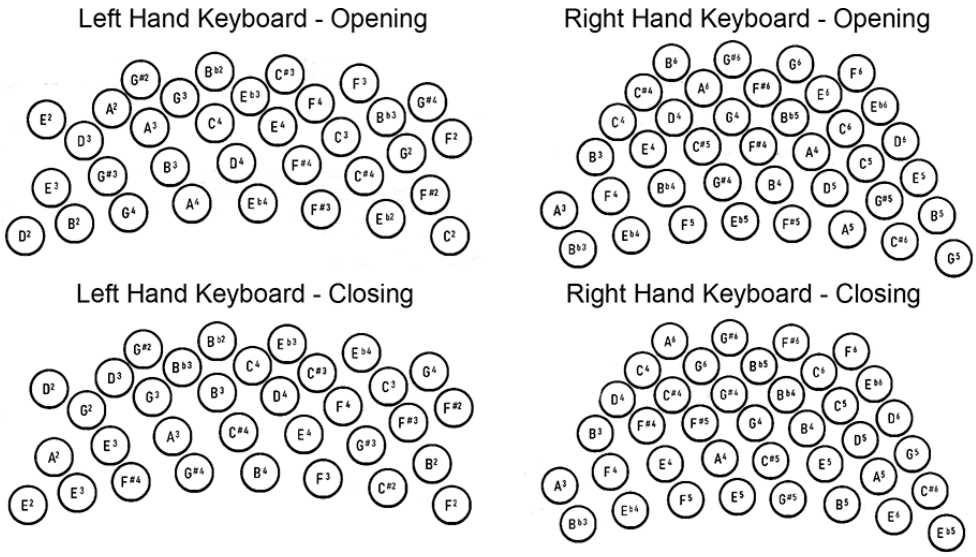
We can create a click-track for live performers.
Both are rigid and difficult systems in syncro for live player.
Good for metrical grid rhythms (pop and rock music).
Bad for rapsodic music.
Complicated for rehearsals.
Only start and stop actions.
We can control it by a triggered switch on asingle key.

```
b = Buffer.read(s, "/absolute/path/to/voce.wav");
SynthDef.new(\simple_player, {arg buf=0, amp=0, pan=0, gate=0;
                                var env, sig;
                                    env = Linen.kr(gate, doneAction:2);
                                    sig = PlayBuf.ar(1, buf,
                                                     BufRateScale.kr(buf),
                                                     doneAction:2);
                                    sig = sig * env * amp.lag(0.2);
                                    sig = Pan2.ar(sig, pan);
                            Out.ar(0, sig);
               }).add;

w = Window.new("key");
~switch = [1,0]; // Flip flop
~synth;
w.view.keyDownAction_({arg ...args;
                        if(args[3]==32){
                                        if(~switch.first==1)
                                        {~synth = Synth.new(\simple_player,
                                                            [\buf,b,
                                                             \amp,1,
                                                             \gate,1])}
                                        {~synth.set(\gate,0)};
                                        ~switch = ~switch.reverse;
                                        }
                        });
w.front;
w.alwaysOnTop_(true);
w.onClose_({w.free;a.free});
```

Listing 6.12: playback one-shot model

6.7.2 Mapping

We can map an event (soundfile or other) to a key as a piano keyboard or bandoneon button.



Key → soundfile

Load all soundfiles in a folder in one array of Buffers.

Soundfile durations should be short (no stop command).

```
~paths = ("/absolute/path/to/folder" ++ "/*.wav").pathMatch;
~bufs  = ~paths.collect{arg i; Buffer.read(s, i)};

SynthDef.new(\map_player, {arg buf=0, amp=0, pan=0;
                            var sig;
                                sig = PlayBuf.ar(1, buf,
                                                 BufRateScale.kr(buf),
                                                 doneAction:2);
                                sig = sig * amp;
                                sig = Pan2.ar(sig, pan);
                            Out.ar(0, sig);
              }).add;

w = Window.new("key");
w.view.keyDownAction_({arg ...args;
                       switch(args[1],
                       $q, {Synth.new(\map_player,[\amp,0.5, \buf,~bufs[0]])},
                       $w, {Synth.new(\map_player,[\amp,0.5, \buf,~bufs[1]])},
                       $e, {Synth.new(\map_player,[\amp,0.5, \buf,~bufs[2]])},
                       $r, {Synth.new(\map_player,[\amp,0.5, \buf,~bufs[3]])},
                       $t, {Synth.new(\map_player,[\amp,0.5, \buf,~bufs[4]])} );
                        });
w.front;
w.alwaysOnTop_(true);
w.onClose_({w.free;a.free});
```

Listing 6.13: playback key-map-soundfile model

Key → playback rate (transposition)

One soundfile, transposition in semitones.

```
b = Buffer.read(s, "/absolute/path/to/voce.wav");
SynthDef.new(\smp_player, {arg buf=0, amp=0, trsp=0, pan=0;
                            var sig;
                                sig = PlayBuf.ar(1, buf,
```

```
                                                    trsp.midiratio*BufRateScale.kr(buf),
                                                    doneAction:2);
                                sig = sig * amp;
                                sig = Pan2.ar(sig, pan);
                            Out.ar(0, sig);
                }).add;

w = Window.new("key");
w.view.keyDownAction_({arg ...args;
                        switch(args[1],
                        $q, {Synth.new(\smp_player,[\amp,0.5, \buf,b,\trsp,-4])},
                        $w, {Synth.new(\smp_player,[\amp,0.5, \buf,b,\trsp, 0])},
                        $e, {Synth.new(\smp_player,[\amp,0.5, \buf,b,\trsp, 2])},
                        $r, {Synth.new(\smp_player,[\amp,0.5, \buf,b,\trsp, 3])},
                        $t, {Synth.new(\smp_player,[\amp,0.5, \buf,b,\trsp, 5])},
                        $y, {Synth.new(\smp_player,[\amp,0.5, \buf,b,\trsp, 7])},
                        );
                        });
w.front;
w.alwaysOnTop_(true);
w.onClose_({w.free;a.free});
```

Listing 6.14: playback key-playback-rate model

Key → start position

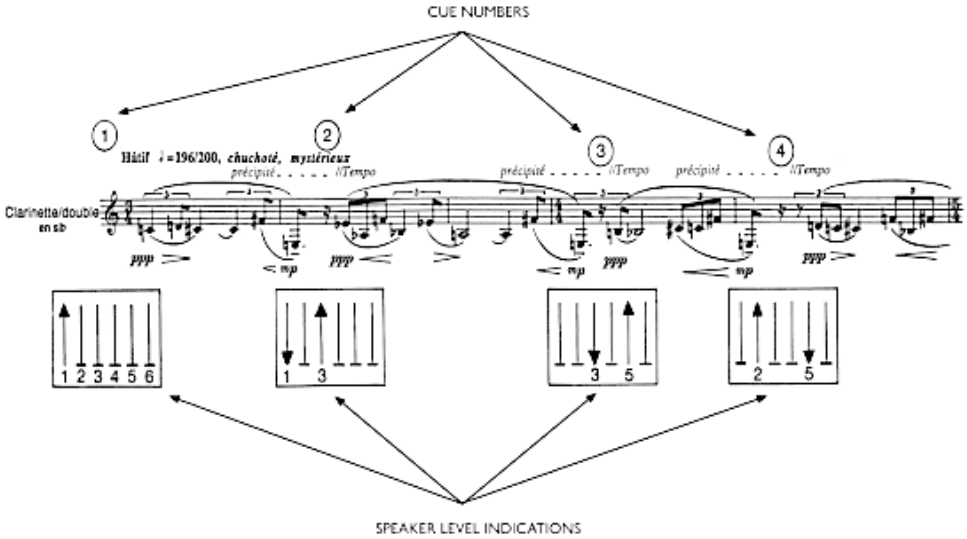
One soundfile, positions and durations in seconds.

```
b = Buffer.read(s, "/absolute/path/to/voce.wav");
SynthDef.new(\pos_player, {arg buf=0, amp=0, pos=0, dur=0.5, pan=0;
                            var env, sig;
                                env = Env.linen(0.01,dur-0.02,0.01);
                                env = EnvGen.kr(env,doneAction:2);
                                sig = PlayBuf.ar(1, buf,
                                                    BufRateScale.kr(buf),
                                            startPos:pos*BufSampleRate.kr(buf));
                                sig = sig * env * amp;
                                sig = Pan2.ar(sig, pan);
                            Out.ar(0, sig);
                }).add;

w = Window.new("key");
w.view.keyDownAction_({arg ...args;
                        switch(args[1],
                        $q, {Synth.new(\pos_player,[\amp,0.5, \buf,b,
                                                    \pos,0,   \dur,1])},
                        $w, {Synth.new(\pos_player,[\amp,0.5, \buf,b,
                                                    \pos,1,   \dur,0.5])},
                        $e, {Synth.new(\pos_player,[\amp,0.5, \buf,b,
                                                    \pos,2,   \dur,0.2])},
                        $r, {Synth.new(\pos_player,[\amp,0.5, \buf,b,
                                                    \pos,3,   \dur,0.3])},
                        $t, {Synth.new(\pos_player,[\amp,0.5, \buf,b,
                                                    \pos,4,   \dur,0.4])}
                        );
                        });
w.front;
w.alwaysOnTop_(true);
w.onClose_({w.free;a.free});
```

Listing 6.15: playback key-start-position model

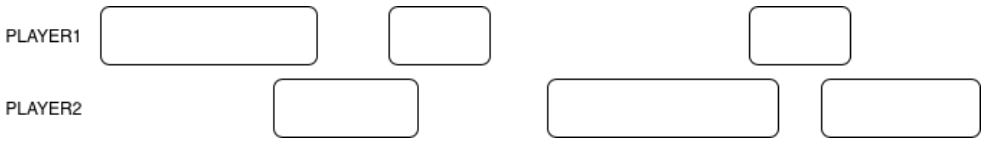
6.7.3 Cues

We can set events in a sequential order and recall it during performance by number.



We need to program a counter.

A usual strategy is program also a cross playback with two alternate players.



This need a crossfade at trigger point.

More dynamic than one-shot playback and better for rehearsals.

Cue durations should be between 1 and 10-15 seconds.

```
~paths = ("/absolute/path/to/folder" ++ "/*.wav").pathMatch;
~bufs  = ~paths.collect{arg i; Buffer.read(s, i)};

SynthDef(\cross, {arg buf=0, amp=0, fade=0.1, pan=0, gate=0;
                  var sig,dur,fad,env;
                      sig = PlayBuf.ar(2, buf);
                      dur = BufDur.kr(buf);
                      fad = Linen.kr(gate,fade,amp.lag(0.2),fade,doneAction:2);
                      env = EnvGen.kr(Env.linen(0.2,dur-0.4,0.2),
                                      gate,doneAction:2);
                      sig = sig * env * fad;
                      sig = Pan2.ar(sig,pan);
                  Out.ar(0,sig)
         }).add;

~cue = 0; // Init counter

w = Window.new("Num", Rect.new(0,0,300,300));
h = NumberBox.new(w, Rect(5,5,290,290))
             .align_(\center)
             .font_(Font("Helvetica", 200););

w.front;
w.alwaysOnTop_(true);
w.onClose_({w.free;h.free; ~cue.free});
```

```
w.view.keyDownAction_({arg ...args;
                        switch(args[3],
                        32, {          // space bar --> increment
                             h.value = ~cue;                          //
    visualization
                             if(s.numSynths > 0, {a.set(\gate,0)});  // Fade out
                             a = Synth(\cross, [\fade, 2,
                                                \amp, 1,
                                                \buf, ~bufs[~cue],
                                                \gate,1]);
                             ~cue  = ~cue + 1 % ~bufs.size;          // modulo
    increment
                             },
                        105, {~cue = 0;                              // 'i' -->
    reset to 0
                              h.value = 0;
                              if(s.numSynths > 0, {a.set(\gate,0)}); })
                        });

h.action_({arg val; ~cue = val.value}); // we can set cue on GUI
```

Listing 6.16: playback cues model

All the examples are with soundfiles players but we can subtitute it with any type of live synthesis generator.

All the examples are triggered by computer key but we can use any kind of controller (MIDI, OSC, mouse, etc.).

6.8 Composition sketches proposal

- design and build a live set with available tools (MIDI, OSC, HID; etc.).
  Keep in mind:
  - sound palette we want to have (sound synthesis and processing techniques).
  - which and how many parameters we want to control.
  - how do we want to control them (knob, fader, botton, accelerometer, gyroscope, etc.).
  - type of interaction (physical and musical gestures) we want to use.
- define a performance that can be:
  - written in a gestural score (we write in a time grid the gestures to be made live on the devices).
  - improvised by interpreting a graphic score.
  - totally improvised in search of the possibilities offered by the performance environment according to the principe of free will.
- performance can be in solo or in laptop ensemble.
  If you play it in ensemble ypu should think about:
  - differentiate or sonically and musically unify the individual sets.
  - what kind of strategies to adopt for synchronisation.
  - have a conductor or not.

You can find more SuperCollider software for live performance here.

Chapter 7

Mixed music and hyper-instruments

7.1 Framework and historical context

Mixed music combines acoustic instruments and electronic elements.

Two different musical contexts:

1. sound and formal augmentation (live composing).
2. acoustic instruments augmentation (hyper-instruments).

For both of them we can do it in two ways:

- superimposing pre-composed electronic sounds and acoustic instruments.
- acoustic instruments real-time sound processing.

For both we have usually:

- one electronic performer.
- one sound director.
- one or more instrumental performers.

7.1.1 Sound and formal augmentation

In this context we can:

1. design a chamber music work for one or more acoustic instruments whose sounds develop immersed in a sound background created by electronics (virtual orchestration and scoring) as in Dai Fujikura - Prism spectra (2007) - extract.
   If we want to do it:
   - with pre-composed electronic sounds (soundfiles) there are different configurations:
     - fixed electronic sounds - fixed intrumental part.
     - fixed electronic sounds - improvised instrumental part.
     - improvised electronics - fixed intrumental part.
     - improvised electronics - improvised instrumental part.

     Technical implementation for this mode is explained in the *Live playback and synthesis* (6.7) paragraph of the previous chapter.
   - in real-time the main sound processing techniques are:
     - accumulation of musical materials → delay lines with or without feedback and/or transpositions.
     - iteration of rhythmic-melodic patterns → loop machines and live recording-playback systems.

  - freeze sounds → create fixed or continuously evolving harmonic fields-sound backgrounds.

  Technical implementations are presented in *Classic live-electronics sound processing* (7.4) paragraph.

2. change the perception of a place by modifying its acoustic characteristics:
   - within a composition as in Luigi Nono - A Pierre. Dell'Azzurro Silenzio, Inquietum (1985) for doublebass flute, doublebass clarinet and live electronics - extract.
   - in the creation of a sound art installation as in David Tudor - Rainforest (1973) - extract.

   Here the focus is on type of sound diffusion systems.
   We talk about this topic in the *Common topics to think about* (7.1.3) section.

7.1.2 Augmentation of acoustic instruments

Hyper-instruments are based on the goal of designing expanded musical instruments using technology to give extra power to intrumental performers.

The first works of this type were created in the 1960s in the same studios that saw the birth of early electronic music on tape (WDR - Cologne and the RAI Phonology Studio in Milan).

Mainly they were pieces for solo instrument or small ensemble and magnetic tape.

The magnetic tape part usually contained processed or simply assembled acoustic instrumental sounds and/or synthesis sounds as in Bruno Maderna - Musica su due dimensioni (1958) for flute, cymbal and tape - extract or Karlheinz Stockhausen - Kontakte (1958-60) for Piano, percussion and 4 channels tape - extract.

In 1976 Luigi Nono wrote with the pianist Maurizio Pollini ...sofferte onde serene for piano and magnetic tape where the magnetic tape creates a virtual piano alter ego.

Beginning in 1975 Giuseppe Di Giugno at IRCAM in Paris began development of digital processors: 4A, 4B, 4C, and in 1981 the very powerful 4X system.

The musical workstation 4X is one of the first computer tools that enables a musician to digitally manipulate sounds in real-time.

Pierre Boulez is the first to use the 4X in his work Répons (1981) for six solistes, chamber ensemble and live-electronics - extracts.

Since the early 1990s with the advent of personal computers and new software dedicated to real-time sound synthesis and processing (Max/MSP, Pure Data, SuperCollider, Chuck, etc.) up to the present day, this musical field has been explored on a vast scale both musically and technologically.

The focus is on designing computer systems (sensors, signal processing, and software) that measure and interpret human expression and feeling as well as on exploring the appropriate modalities and innovative content of interactive art.

Few examples.

Yan Maresz - Metallics (1995) for trumpet and live electronics - extract.

Fausto Romitelli - Trash TV Trance (2002) for electric guitar and live electronics - extract.

Georg Hajdu - Just Her - Jester - Gesture (2010) for kalimba and live electronics - extract.

In last years also the composer/performer body became an instrument for live processing sounds.

Marco Donnarumma - Corpus Nil (2016)

7.1.3 Common topics to think about

- music and sound relationship between acoustic and electronic parts.
  - musical and timbre coherence.
  - musical and timbre contrast.
- musical and physical gestures.
  - strictly connected with previous point.
  - acousmatic like approach → no relation to the historicized instrumental timbre and to traditional instrumental techniques and sound morphologies.
  - evident augmented musical gestures → clear cause-effect relationship or mickey mouse effect (usually short time).
  - masked augmented musical gestures → no preceptive relationship between cause and effect (usually long time).
- direction of sounds and type of sound diffusion systems.
  - strictly connected with previous points.
  - front diffusion systems → mono or stereo pop concert like.
  - immersive systems → acoustic fields (quadriphonic, octophonic, ambisonic, wavefield synthesis, etc.).
  - discrete diffusion systems → speaker as musical instrument (acousmonium, laptop orchestra, etc.).
- timbre and visual recognition of the sound source.
  - strictly connected with previous points.
  - the player is on a stage and audience see it and the way he produce sounds.
- choice between pre composed electronic parts (sound files) or live processing.
  - first choice → pre-composed sounds (soundfiles triggering).
  - we should choose live processing only when we don't want precision in the execution and the aesthetic of the work need uncontrollable margins of variation between different executions.
  - motivations must be musical and not tied to technological fetishes.

7.2 Software model

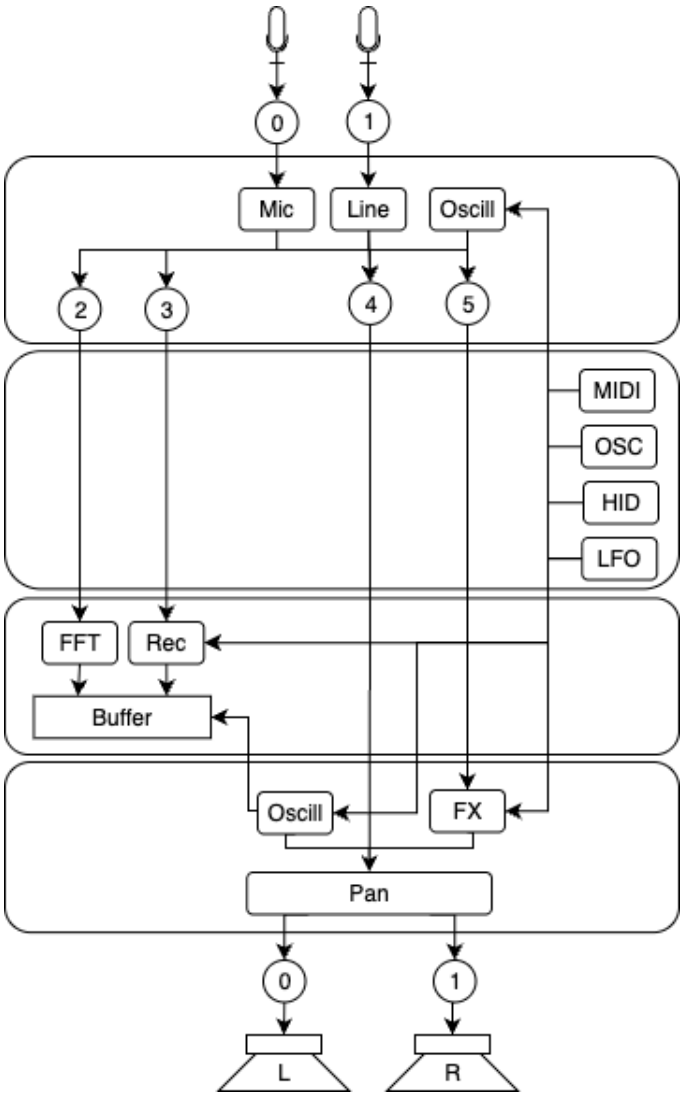
In a live-electronic patch we have four types of intruments:

- sound (audio) sources.
  - external → mics, line inputs.
  - internal → oscillators.
- control sources.
  - external → midi, osc, hid, sensors, etc.
  - internal → source signals, oscillators, sequencer.
- recorders or analyzers.
- sound processors.
  - FX
  - panners and spats.

Set input and ouput audio connections (before Server boot).

```
ServerOptions.devices; // Array with all connected audio devices
s.options.inDevice_("Microfono␣MacBook␣Air"); // set input device
s.options.outDevice_("Auricolari␣esterni");   // set output device
s.options.numInputBusChannels_(1);            // set number of input channels
s.options.numOutputBusChannels_(8);           // set number of ouput channels
```

Listing 7.1: Server settings model



As example:

- one ring modulator.
- one recorder (click any computer key to start recording).
- one automatic buffer player (scratcher).

```
s.waitForBoot{
// ---------------------------------- BUFFERS
Buffer.freeAll;
//~bfile = Buffer.read(s, "folder/file.wav".resolveRelative);
~blive = Buffer.alloc(s, s.sampleRate * 6, 1); // Live source Buffer
s.sync;       // wait until commands have been completed.
// --------------------------------- SYNTHDEFS
SynthDef(\mic, {arg busIn=0,busOut=0,amp=0;   // sig In --> Bus
                    var sig;
                    sig = SoundIn.ar(busIn) * amp.lag;
                    sig = LeakDC.ar(sig);
                Out.ar(busOut,sig)
                }).add;
SynthDef(\sch, {arg buf=0, amp=0, pos=0, pan=0, busOut=0;
                var pnt, sig;
                    pnt = Lag.kr(pos.linlin(-1,1,0,BufFrames.kr(buf)),0.2);
                    sig = BufRd.ar(1, buf, K2A.ar(pnt));
                    sig = Pan2.ar(sig * amp, pan);
                Out.ar(busOut, sig)
```

```
                 }).add;
SynthDef(\recOne,{arg buf=0, busIn=0;        // Bus --> Buffer
                   var in;
                       in = In.ar(busIn, 1);
                       RecordBuf.ar(in, buf, loop:0, doneAction:2);
                   }).add;
SynthDef(\ksig,{arg freq=1, busOut=0;        // control sig --> Bus
                 var sig;
                     sig = LFNoise1.ar(1);
                     Out.ar(busOut, sig)
                 }).add;
SynthDef(\ring, {arg busIn=0, busOut=0,      // Bus In --> Bus Out (FX)
                      idx=0, fmod=400, amp=0, pan=0;
                  var port, mod;
                      port  = In.ar(busIn, 1) * amp.lag;
                      mod   = SinOsc.ar(fmod.linlin(-1,1,400,1000)) * idx.
    unipolar;
                      port  = Pan2.ar(port * mod, pan);
                  Out.ar(busOut, port)
                  }).add;
s.sync;
// ----------------------------- BUSES AND GROUPS
~mic1.free;
~mic2.free;
~mic1 = Bus.audio(s,1);
~mic2 = Bus.audio(s,1);
~ksig = Bus.audio(s,1);

~micGrp = Group.new;             // sources
~ksiGrp = Group.after(~micGrp); // control signals
~recGrp = Group.after(~ksiGrp); // recorder and/or analyzers
~outGrp = Group.after(~recGrp); // fx and/or panning
s.sync;
// ---------------------------------- SYNTHS
Synth(\mic,   [\busIn, 0, \amp,1, \busOut, ~mic1], ~micGrp);
Synth(\sch,   [\buf, ~blive, \amp,1, \pos, ~ksig.asMap, \busOut, 0], ~micGrp);
Synth(\ksig,  [\busOut, ~ksig, \freq, 5], ~ksiGrp);
Synth(\ring,  [\busIn, ~mic1, \busOut, 0,
               \idx, ~ksig.asMap, \fmod, ~ksig.asMap,
               \amp, 1, \busOut, 0], ~outGrp);
// ------------------------------------ GUI
w = Window.new(\key);
w.alwaysOnTop_(true);
w.front;
// ------------------------------- MIDI OSC HID
MIDIClient.init;
MIDIIn.disconnectAll;
MIDIIn.connectAll;

// MIDIFunc.trace(true);
// MIDIFunc.trace(false);

MIDIdef.cc(\rec, {arg val, ccn;
                      if(val==127)
                 {Synth(\recOne, [\busIn, ~mic1, \buf, ~blive], ~recGrp);
                       "Recording␣Buffer".postln}
                   }, 0);
w.view.keyDownAction_({arg ...args;
                       Synth(\recOne, [\busIn, ~mic1, \buf, ~blive], ~recGrp);
                       "Recording␣Buffer".postln;
                       });
s.sync;
s.meter;
```

```
s.plotTree;
}
```

Listing 7.2: Live electronics model

```
~blive.plot;
s.freeAll;
```

7.2.1 Routing signals

One of the main aspects of a live electronics patch is the the signals routing.

In SuperCollider we can organize it through In, Out, Bus and Group objects.

In/Out

Let's define two Synths to test connections:

- source.
- sound processor (fx reverb).

```
SynthDef(\source_1, {arg busOut=0, den=2, dur=0.8; // Out mono
                      var trig,sig,env;
                          trig = Dust.ar(den);
                          sig  = Saw.ar(TRand.ar(70,82,trig).midicps);
                          env  = Decay2.ar(trig,0.02,dur-0.02);
                          sig  = sig * env;
                      Out.ar(busOut, sig); // Out.ar(bus, sig)
                      }).add;

SynthDef(\rev, {arg busIn=0, busOut=0; // In + Out mono
                 var sig, rev;
                     sig = In.ar(busIn, 1);  // In.ar(bus, n_channels)
                     rev = FreeVerb.ar(sig,0.5,0.8,0.7);
                 Out.ar(busOut, rev)
                 }).add;
```

Dynamic bus changes → clicks!

```
a = Synth(\rev, [\busIn, 20, \busOut, 0]);
b = Synth(\source_1, [\busOut,  20]);
b.set(\busOut, 0);
a.free; b.free;
```

To acquire input signals from public buses → *SoundIn.ar()*.

```
SynthDef(\mic, {arg busIn=0, busOut=0, amp=0;
                 var sig;
                     sig  = SoundIn.ar(busIn, amp.lag);
                     sig  = LeakDC.ar(sig);   // remove DC offset
                 Out.ar(busOut, sig)
                 }).add
```

As exemple plug it in a reverber.

```
a = Synth(\rev, [\busIn, 20, \busOut, 0]);
b = Synth(\mic, [\busOut,  20, \amp, 1]);
a.free; b.free;
```

Public vs private buses

Two Bus types:

- public → connected to the outside world (soundcard).
- private → only for internal signals routing.

Do not use public buses for internal routing.

Few useful informations:

```
s.options.numAudioBusChannels;  // default --> 1024
s.options.numInputBusChannels;  // public
s.options.numOutputBusChannels; // public
s.options.firstPrivateBus;      // first private bus idx
```

The *Bus* Class only assigns *private* buses.

It's better to specify the modifier name (FX send) as the label rather than the output signal name.

```
~revBus = Bus.audio(s, 1); // Server, n_channels
~revBus.index;

a = Synth(\rev, [\inBus, ~revBus, \outBus, 0]);
b = Synth(\source_1, [\outBus,  ~revBus]);

a.free;b.free;
```

Useful Bus commands.

```
~revBus.index;
~revBus.numChannels;
~revBus.rate;
~revBus.free;
```

If source is stereo or multichannel we must to specify the number of channels.

```
SynthDef(\source_2, {arg busOut=0, den=2, dur=0.8;
                      var trig,sig,env;
                          trig = Dust.ar(den);
                          sig  = Saw.ar(TRand.ar(70,82,trig).midicps);
                          env  = Decay2.ar(trig,0.02,dur-0.02);
                          sig  = sig * env;
                          sig  = Pan2.ar(sig, LFNoise1.ar(10)); // Pan random
                      Out.ar(busOut, sig)
                      }).add;

SynthDef(\rev2, {arg busIn=0, busOut=0;
                  var sig, rev;
                      sig = In.ar(busIn, 2);  // two channels
                      rev = FreeVerb.ar(sig,0.5,0.8,0.7);
                  Out.ar(busOut, rev)
                  }).add;

~revBus1 = Bus.audio(s, 2); // two channels bus
~revBus1.index;

a = Synth(\rev2,      [\busIn,  ~revBus1, \busOut, 0]);
b = Synth(\source_2, [\busOut, ~revBus1]);

b.set(\busOut,0);  // direct Out (without reverber)
b.set(\busOut, ~revBus1);
```

```
s.freeAll;
```

Creation order rules and Groups

Synths that receive an input signal must be created before those that send it.

Look at the plot tree.

It work.

```
s.plotTree;

a = Synth(\rev2,      [\inBus,  ~revBus1, \outBus, 0]); // IN + OUT
b = Synth(\source_2, [\outBus, ~revBus1]);              // only OUT

a.free;b.free;
```

It does not work.

```
b = Synth(\source_2, [\outBus, ~revBus1]);              // only OUT
a = Synth(\rev2,      [\inBus,  ~revBus1, \outBus, 0]); // IN + OUT

a.free;b.free;
```

This happens because the server by default creates a head-to-tail order.

The last Synth created is placed at the top.

We can force creation to always be at the bottom with arguments:

```
b = Synth(\source_2, [\outBus, ~revBus1]);
a = Synth(\rev2,      [\inBus,  ~revBus1, \outBus, 0],  s, 'addToTail');

a.free;b.free;
```

Further options in the Server help file.

We can simplify the structure by using Groups.

Groups are containers for Nodes and Synths.

When we launch SuperCollider the default Group is create (look at the node tree).

As we saw in the software model proposed at the beginning of the section we can create further Groups within the main one and place the Synths within one or the other.

7.3 Feature extraction as control signals

In a paragraph of previous chapter (6.3) we saw that control signals can:

- come from external sources (MIDI, OSC, HID, etc.).
- be generated by low-frequency oscillators (LFO).

We also saw the main possible operations:

- scale them to a new range.
- smooth them.
- convert them between different typologies.
- map them to any parameter of any sound synthesis or processing technique.

We can apply the same techniques to transform an input audio signal into a control signal.

The techniques commonly used are:

- envelope following.
- pitch following.
- onset detection.
- spectral analyses (FFT).

Let's define a SynthDef for source, Buses and Groups.

```
s.waitForBoot{
SynthDef(\mic, {arg busIn=0, busOut=0, amp=0;
                var sig;
                    sig = SoundIn.ar(busIn) * amp.lag;
                    sig = LeakDC.ar(sig);
                Out.ar(busOut,sig)
                }).add;
s.sync;
~micGrp   = Group.new;              // Gruppo sources
~featGrp  = Group.after(~micGrp);   // Gruppo analisi
~playGrp  = Group.after(~featGrp); // Gruppo synth
s.sync;
s.meter;
s.plotTree;
};
```

7.3.1 Envelope follower

Analysis of amplitude envelope (unipolar) from an incoming audio signal (bipolar).

In SuperCollider we can use *Amplitude.ar()* UGens.

```
// continuos
SynthDef(\efol1, {arg busIn=0, busOut=0, atk=0.001, dec=0.01;
                  var sig, amp;
                      sig = In.ar(busIn,1);
                      amp = Amplitude.ar(sig,atk,dec);
                  Out.ar(busOut,amp)
                  }).add;
// trigger
SynthDef(\efol2, {arg busIn=0, busOut=0, atk=0.001, dec=0.01,
                      thresh=0.3, dur=0.1;
                  var sig, amp, trig;
                      sig  = In.ar(busIn,1);
                      amp  = Amplitude.ar(sig,atk,dec);
                      trig = Trig1.ar(amp > thresh, dur); // Trigger
                  Out.ar(busOut, [amp, trig]);            // two channels

      // Server --> Interpreter

                SendTrig.ar(trig,
                            100,  // trigger id (if more than 1)
                            amp); // signal to send
                }).add;
// with noise gate
SynthDef(\efol3, {arg busIn=0, busOut=0, atk=0.01, dec=0.01, thresh=0.3;
                  var sig, env;
                      sig = In.ar(busIn, 1);
                      sig = Compander.ar(sig, sig, thresh, 10, 1, 0.01, 0.2);
                      env = Amplitude.ar(sig, atk,dec);
                  Out.ar(busOut, [env, env.pow(0.5)])     // power...
                  }).add;
```

Listing 7.3: Envelope follower models

Continuos mapping example.

```
SynthDef(\noise, {arg busIn=0, busOut=0, pan=0;
                   var sig, env;
                       sig = PinkNoise.ar;
                       env = In.ar(busIn, 1); // from envelope follower
                       sig = sig * env;
                       sig = Pan2.ar(sig, pan);
                   Out.ar(busOut, sig)
                 }).add;

~featureBusIn  = Bus.audio(s, 1);  // Bus signal In
~featureBusOut = Bus.audio(s, 1);  // Bus featured values

~envf   = Synth(\efol1, [\busIn,~featureBusIn,
                         \busOut,~featureBusOut,
                         \dec,0.6], ~featGrp);
~player = Synth(\noise, [\busIn, ~featureBusOut,
                         \busOut,0], ~playGrp);
~monitor  = ~featureBusOut.scope;

c.set(\atk,0.1,\dec,0.2);  // try to change values
s.defaultGroup.deepFree;   // Clear Synths and Bus but not Groups
```

Trigger mapping example.

```
SynthDef(\sine, {arg busIn=0, busOut=0;
                  var sig, gate, env;
                      gate = In.ar(busIn, 2);                          // 2
    channels
                      env  = EnvGen.kr(Env.perc, gate[1]);             // switch
                      sig  = SinOsc.ar(TRand.ar(500,1000,gate[1]));    // trig
                      sig  = sig * env;
                      sig  = Pan2.ar(sig, gate[0].linlin(0,1,-1,1)); //
    continuos
                  Out.ar(busOut, sig)
                  }).add;

~featureBusIn  = Bus.audio(s, 1);  // Bus signal In
~featureBusOut = Bus.audio(s, 1);  // Bus featured values

~source  = Synth(\mic, [\busIn,0, \busOut, ~featureBusIn, \amp,1], ~micGrp);
~envf    = Synth(\efol2, [\busIn,~featureBusIn,
                          \busOut,~featureBusOut,
                          \dec,0.6,\thresh, 0.1,\dur,0.2], ~featGrp);
~player  = Synth(\sine,  [\busIn,~featureBusOut,
                          \busOut,0], ~playGrp);
~monitor = OSCFunc.new({arg msg; msg.postln},'/tr', s.addr); // from Server...

~envf.set(\thresh,0.1,\dur,0.1); // try to change threshold
s.defaultGroup.deepFree;
```

In some situations it may be useful to insert a noise gate before the envelope following.

Noise gate → if signal in under a threshold is cutted (UGen Compander).

```
SynthDef(\saw, {arg busIn=0, busOut=0;
                 var sig, env;
                     env  = In.ar(busIn, 1);  // only first bus channel
```

```
                    sig  = Saw.ar([560,563]);
                    sig = sig * env * 0.5;
                Out.ar(busOut, sig)
                }).add;

~featureBusIn  = Bus.audio(s, 1);  // Bus signal In
~featureBusOut = Bus.audio(s, 1);  // Bus featured values

~source = Synth(\mic, [\busIn,0, \busOut, ~featureBusIn, \amp,1], ~micGrp);
~envf   = Synth(\efol3, [\busIn,~featureBusIn,
                         \busOut,~featureBusOut,
                         \dec,0.6,\thresh, 0.001,\dur,0.2], ~featGrp);
~player = Synth(\saw,   [\busIn,~featureBusOut,
                         \busOut,0], ~playGrp);

s.defaultGroup.deepFree;
```

More musically interesting examples are illustrated in the paragraph on real-time synthesis and processing techniques.

Few ideas:

- mix synthesized sounds that react to the envelope follower with the original signal.
- define virtual dampers with filter banks whose cutoff frequencies and/or bandwidths are dynamically controlled through envelope following.
- using the signal of an acoustic instrument exclusively as a controller of synthesis sounds in a live set (no original signal in out).

7.3.2 Pitch follower

Pitch following is generally less stable and precise than envelope following.

Usually it returns two parameters:

- estimated pitch.
- founded or not founded pitch (few times accuracy percentage).

There are two main techniques:

- autocorrelation → *Pitch* UGens.
- FFT → *Tartini* UGens (SuperCollider extension).

First is cpu cheaper.

To improve accuracy we add an envelope follower: if the amplitude of the analyzed signal is less than a threshold it is not resynthesized.

```
// continuos autocorrelation
SynthDef(\pfol1, {arg busIn=0,busOut=0,high=1000,low=60,median=1,
                      atk=0.01, dec=0.01, filt=1000;
                  var sig, amp, freq, hasFreq;
                      sig = In.ar(busIn,1);
                      amp = Amplitude.kr(sig, atk, 0.01);
                      sig = LPF.ar(sig, filt);
                      // two variables (Python like syntax)
                      #freq, hasFreq = Pitch.kr(sig,
                                                high,    // reference pitch
                                                low,     // lowest pitch
                                                high,    // hightest pitch
                                                median:median, // Median filter
                                                );
                      freq.poll;
```

```
                        hasFreq.poll;
                    Out.kr(busOut,[freq, hasFreq, amp])  // three channels bus
                    }).add;
// continuos FFT
SynthDef(\pfol2, {arg busIn=0,busOut=0,atk=0.01,dec=0.01,filt=2000;
                    var sig, amp, freq, hasFreq;
                        sig = In.ar(busIn,1);
                        amp = Amplitude.kr(sig, atk, 0.01); // Amplitude follower
                        sig = LPF.ar(sig, filt);            // Low pass filter
                        #freq, hasFreq = Tartini.kr(sig);   // Pitch follower
                        freq.poll;
                        hasFreq.poll;
                    Out.kr(busOut,[freq, hasFreq, amp])      // 3 canali...
                    }).add;
// trigger
SynthDef(\pfol3, {arg busIn=0, busOut=0, low=200, high=300, filt=2000;
                    var sig, freq, hasFreq,trig1,trig2,trig;
                        sig = In.ar(busIn, 1);
                        sig = LPF.ar(sig, filt);
                        #freq, hasFreq = Tartini.kr(sig);
                        freq.poll;
                        trig1 = freq > low;
                        trig2 = freq < high;
                        // limits redundancies
                        trig  = Trig1.ar((trig1 + trig2 -1).poll, 0.5);
                    Out.kr(busOut,[freq, hasFreq]);

    // Server --> Interpreter

                SendTrig.ar(trig,
                            113,
                            freq)
                }).add;
```

Listing 7.4: Pitch follower models

Continuos mapping example.

Try pfol1 and pfol2 to compare results.

```
SynthDef(\sine1, {arg busIn=0, busOut=0, thresh=0.05, pan=0;
                    var sig, ksig, env;
                        ksig = In.kr(busIn, 3);
                        sig  = Mix(SinOsc.ar(ksig[0] * [0.5,1,3])); // three oct
                        // if amp > thresh = 1
                        env  = (ksig[2] > thresh).lag(0.4) * 0.2;
                        sig  = sig * env * ksig[1];
                        sig = Pan2.ar(sig, pan);
                    Out.ar(busOut, sig)
                  }).add;

~featureBusIn  = Bus.audio(s, 1);  // Bus signal In
~featureBusOut = Bus.control(s, 3); // Bus featured control values 3 channels

~source = Synth(\mic, [\busIn,0, \busOut, ~featureBusIn, \amp,1], ~micGrp);
~pchf   = Synth(\pfol1, [\busIn,~featureBusIn, // Try to change with pfol1
                         \busOut,~featureBusOut,
                         \median,7], ~featGrp);
~player = Synth(\sine1, [\busIn, ~featureBusOut,
                         \busOut,0], ~playGrp);

s.defaultGroup.deepFree;
```

Trigger mapping example.

If an estimated pitch is in a range → trig.

```
SynthDef(\sine2, {arg freq;
                  var sig, env;
                      sig  = Mix(SinOsc.ar(freq * [0.5,2,3])); // three oct
                      env  = EnvGen.kr(Env.perc, doneAction:2);
                  Out.ar(0, sig * env * 0.2!2)
                  }).add;

~featureBusIn  = Bus.audio(s, 1);   // Bus signal In
~featureBusOut = Bus.control(s, 3); // Bus featured control values 3 channels

~source = Synth(\mic, [\busIn,0, \busOut, ~featureBusIn, \amp,1], ~micGrp);
~pchf   = Synth(\pfol3, [\busIn,~featureBusIn,     // Pitch UGen
                         \busOut,~featureBusOut,
                         \median,7], ~featGrp);
~player = OSCFunc.new({arg msg; msg.postln;
                       Synth(\sine2, [\freq, msg[3]], ~playGrp)},
                       '/tr', s.addr);

s.defaultGroup.deepFree;
```

Few ideas:

- trig selected pitched soundfiles (pitch choose which file to play).
- control pitch of a frequency shifter
- control a modular oscillator frequency (Ring, AM, FM).

7.3.3 Onset detection

The main technique consists in comparing a signal with a copy of itself.

In SuperCollider we can use the Slope UGen.

```
SynthDef(\onsDec, {arg busIn=0, busOut=0,
                       atk=0.01, dec=0.01, dur=0.2, thresh=12;
                   var sig, amp, slope, trig;
                       sig   = In.ar(busIn,1);
                       amp   = Amplitude.ar(sig,atk,dec); // Env follower
                       slope = Slope.ar(amp);
                       trig  = Trig1.ar(slope > thresh, dur);
                   Out.ar(busOut, trig);

    // Server --> Interpretere
               SendTrig.ar(trig,153, 1)
               }).add;
```

Listing 7.5: Onset detection model

An example with threshold.

```
SynthDef(\saw, {arg busIn=0,busOut=0,pan=0;
                var sig, gate, env;
                    gate = In.ar(busIn,1);
                    env  = Env.adsr;
                    env  = EnvGen.kr(env, gate); // receive from detect
                    sig  = Saw.ar(LFNoise0.kr([10,12]).range(300,1000));
                    sig  = sig * env * 0.5;
                    sig  = Pan2.ar(sig, pan);
                Out.ar(busOut,sig * env * 0.5)
                }).add;
```

```
SynthDef(\noise, {arg busOut=0,pan=0;
                   var sig, env;
                       sig  = PinkNoise.ar;
                       env  = Env.perc(0.01,TRand.kr(0.02,0.1,1));
                       env  = EnvGen.kr(env, 1, doneAction:2);
                       sig  = sig * env;
                       sig  = Pan2.ar(sig, pan);
                   Out.ar(busOut, sig)
                   }).add;

~featureBusIn  = Bus.audio(s, 1);  // Bus signal In
~featureBusOut = Bus.audio(s, 1);  // Bus featured values

~source = Synth(\mic, [\busIn,0, \busOut, ~featureBusIn, \amp,1], ~micGrp);
~detect = Synth(\onsDec, [\busIn,~featureBusIn,
                          \busOut,~featureBusOut,
                          \thresh, 12], ~featGrp);
~player = Synth(\saw, [\busIn, ~featureBusOut,
                       \busOut,0], ~playGrp);
~noise  = OSCFunc.new({arg msg;
                       Synth(\noise,target:~playGrp)},
                       '/tr', s.addr);

s.defaultGroup.deepFree;
```

More about onset detection and beat tracking see these UGens help files:

- Onsets (FFT).
- BeatTrack (FFT).
- Coyote (Extension non FFT).

7.3.4 Spectral features

There are many FFT analyses that can be used for feature extraction.

As example here:

- spectral centroid → useful indicator of the perceptual brightness of a signal.
- spectral flatness → from approx 0 for a pure sinusoid, to approx 1 for white noise.

The syntax is similar for all.

```
s.freqscope;

SynthDef(\scentr, {arg busIn=0, busOut=0;
                    var sig, chain;
                        sig   = In.ar(busIn, 1);
                        chain = FFT(LocalBuf(2048), sig);
                        sig   = SpecCentroid.kr(chain);
                        sig.poll(10);
                    Out.kr(busOut, sig); // control rate
                    }).add;

SynthDef(\sflat, {arg busIn=0, busOut=0;
                   var sig, chain;
                       sig   = In.ar(busIn, 1);
                       chain = FFT(LocalBuf(2048), sig)
                       sig   = SpecFlatness.kr(chain);
                       sig.poll(10);
                   Out.kr(busOut, sig); // control rate
                   }).add;
```

Listing 7.6: Spectral features model

Spectral centroid exemple.

```
SynthDef(\fnoise, {arg busIn=0, busOut=0, freq=0, pan=0, gate=0, bw=0.2;
                  var sig, amp, env;
                      sig = Saw.ar(100);
                      amp = In.ar(busIn,1);
               amp = Amplitude.ar(amp);       // Env follower
                      sig = Resonz.ar(sig, freq.lag(0.5), bw.lag(0.5));
                      env = Linen.kr(gate, doneAction:2);
                      sig = sig * env * amp * 4;
                      sig = Pan2.ar(sig, pan);
                  Out.ar(busOut, sig);
         }).add;

~featureBusIn  = Bus.audio(s, 1);   // Bus signal In
~featureBusOut = Bus.control(s, 1); // Bus featured control values

~source = Synth(\mic, [\busIn,0, \busOut, ~featureBusIn, \amp,1], ~micGrp);
~ksig   = Synth(\scentr, [\busIn,~featureBusIn,
                           \busOut,~featureBusOut], ~featGrp);
~player = Synth(\fnoise, [\busIn, ~featureBusIn,
                         \busOut,0,
                       \freq, ~featureBusOut.asMap,
                       \gate, 1], ~playGrp);
s.defaultGroup.deepFree;
```

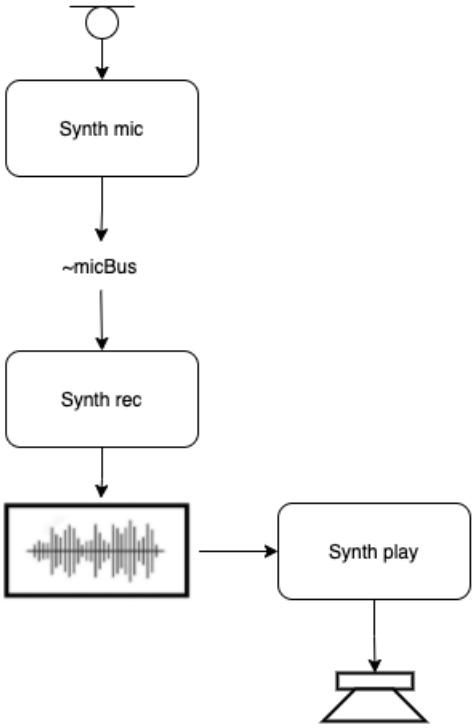
Spectral flattness example.

```
~featureBusIn  = Bus.audio(s, 1);   // Bus signal In
~featureBusOut = Bus.control(s, 1); // Bus featured control values

~source = Synth(\mic, [\busIn,0, \busOut, ~featureBusIn, \amp,1], ~micGrp);
~ksig   = Synth(\sflat, [\busIn,~featureBusIn,
                          \busOut,~featureBusOut], ~featGrp);
~player = Synth(\fnoise, [\busIn, ~featureBusIn,
                         \busOut,0,
              \freq, 2800,
                       \bw, ~featureBusOut.asMap,
                       \gate, 1], ~playGrp);
s.freeAll;
```

7.4 Classic live electronics sound processing

In this section we find the main techniques historically used in pieces for instruments and live processing.

They are all almost all based on the real-time recording of one or more buffers whose contents are playbacked according to different techniques.



Let's build the environment.

```
s.waitForBoot{
~micGrp = Group.new;             // sources Group
~recGrp = Group.after(~micGrp); // recorder and ksig Group
~fxGrp  = Group.after(~recGrp); // fx Group
s.sync;
SynthDef(\mic, {arg busIn=0, busOut=0, amp=0;
                    var sig;
                    sig = SoundIn.ar(busIn) * amp.lag;
                    sig = LeakDC.ar(sig);
                Out.ar(busOut,sig)
                }).add;
SynthDef(\ksig,{arg type=0, range=#[0,1], freq=1, busOut=0;
              var sig;
                  sig = Select.kr(type,
                                 [MouseX.kr(range[0], range[1]), // 0
                                  MouseY.kr(range[0], range[1]), // 1
                                  LFNoise1.kr(freq).range(range[0], range[1]),
    // 2
                                  LFSaw.kr(freq).range(range[0], range[1]),
    // 3
                                  Impulse.kr(freq),              // 4
                                  Dust.kr(freq),                 // 5
                                 ]);
              Out.kr(busOut,sig)
                }).add;
s.sync;
s.meter;
s.plotTree;
};
```

Define a Synth for recording.

```
SynthDef(\rec,{arg buf=0, busIn=0, loop=0, mix=0, done=2;
                var in;
                    in = In.ar(busIn, 1);
```

```
                    RecordBuf.ar(in,
                                 buf,
                                 preLevel:mix,  // 1 = add audio
                                                // 0 = clear before
                                 loop:loop,     // 1 = loop
                                 doneAction:done)
               }).add;
```

Listing 7.7: Live recording model

Create an empty Buffer mono (set duration in frames)

In live electronics there are two ways to record it:

- one shot → fill the Buffer until the end (or a stop message) then stop the recording.
- loop → when the buffer is full it starts over from the beginning in a loop.

In both cases we can choose whether to overwrite the contents of the Buffer or delete it dynamically.

One shot example.

```
Buffer.freeAll;
~bufDur = 3;     // buffer size in seconds
~bufLive = Buffer.alloc(s, s.sampleRate * ~bufDur, 1 );

~source   = Synth(\mic, [\busIn,0, \busOut, ~bufBusIn, \amp,1], ~micGrp);
~recorder = Synth(\rec, [\buf, ~bufLive,
                         \busIn, ~bufBusIn,
                         \mix, 0,    // no overwriting
                         \loop, 0,   // no loop
                         \done, 2], ~recGrp);

~bufLive.play;           // monitoring not playback
s.defaultGroup.deepFree; // clear Synths, not Groups
```

Loop exemple.

```
Buffer.freeAll;
~bufDur = 3;     // buffer size in seconds
~bufLive = Buffer.alloc(s, s.sampleRate * ~bufDur, 1 );

~source   = Synth(\mic, [\busIn,0, \busOut, ~bufBusIn, \amp,1], ~micGrp);
~recorder = Synth(\rec, [\buf,~bufLive,
                         \busIn,~bufBusIn,
                         \mix,0,    // clear before
                         \loop,1,   // no loop
                         \done,2], ~recGrp);

~bufLive.play;
s.defaultGroup.deepFree;
```

7.4.1 Ring modulation and frequency shifting

An input signal is multiplied by a sinusoid (both signals are bipolar).

Output signal consisting of the sum and difference of their original partial (known as sidebands) while eliminating the original input signals.

This process usually generates inharmonic, metallic, bell-like sounds or robotic voices.

One of the earliest musical instruments utilizing a ring modulator was the Melochord (1947) built by Harald Bode.

Karlheinz Stockhausen used for the first time ring modulation in 1956 for some sounds in Gesang der Jünglinge - extract.

The distinctive voice of the Daleks in the television series Doctor Who starting in 1963 is produced by ring modulation.

John McLaughlin employs the ring modulator heavily in the 1974 Mahavishnu Orchestra album Visions of the Emerald Beyond, especially on the track On the Way Home to Earth - extract.

The Synth model.

```
SynthDef(\ring, {arg busIn=0, busOut=0, fmod=400, idx=0, amp=0, gate=0, pan=0;
                 var car, mod, env, sig;
                     car = In.ar(busIn, 1);
                     mod = SinOsc.ar(fmod) * idx;
                     env = Linen.kr(gate, doneAction:2);
                     sig = car * mod * amp * env;
                     sig = Pan2.ar(sig, pan);
                 Out.ar(busOut, sig)
                 }).add;
```

Listing 7.8: Live ring modulation model

In this example we dynamically control the modulation index and the frequency of the modulating oscillator with the mouse.

```
~xbus = Bus.control(s,1);
~ybus = Bus.control(s,1);

~fxBusIn  = Bus.audio(s, 1);  // Bus fx in

~source = Synth(\mic,   [\busIn,0, \busOut, ~fxBusIn, \amp,1], ~micGrp);
~mousex = Synth(\ksig,  [\type, 0, \range, [300, 1300], \busOut, ~xbus], ~
    recGrp);
~mousey = Synth(\ksig,  [\type, 0, \range, [0, 1],      \busOut, ~ybus], ~
    recGrp);

~ring   = Synth(\ring, [\busIn, ~fxBusIn,
                        \busOut, 0,
                        \idx,  ~ybus.asMap, // mouse y
                        \fmod, ~xbus.asMap, // mouse x
                        \amp, 1,
                        \gate, 1], ~fxGrp);
s.defaultGroup.deepFree;
```

As an example of augmented instrumental techniques we can:

- map envelope follower → modulation index (louder more modulated or vice versa).
- map spectral centroid → modulation frequency.

Added also a chorusing to modulation frequency.

```
SynthDef(\ring1, {arg busIn=0, busOut=0, amp=0, gate=0, pan=0;
                  var car, mod, chain, fmod, idx, env, sig;
                      car = In.ar(busIn, 1);
                      idx = Amplitude.ar(car).lag(0.5); // amp follower
                      chain = FFT(LocalBuf(2048), car);
                      fmod  = SpecCentroid.kr(chain).lag(0.5);   // centroid
                      mod = Mix(SinOsc.ar([fmod,fmod+0.5])) * idx; // chorus
                      env = Linen.kr(gate, doneAction:2);
```

```
                        sig = car * mod * amp * env;
                        sig = Pan2.ar(sig, pan);
                    Out.ar(busOut, sig)
                    }).add;

~fxBusIn  = Bus.audio(s, 1);  // Bus fx in

~source = Synth(\mic,   [\busIn,0, \busOut, ~fxBusIn, \amp,1], ~micGrp);
~ring   = Synth(\ring1, [\busIn, ~fxBusIn,
                         \busOut, 0,
                         \amp, 1,
                         \gate, 1], ~fxGrp);
s.defaultGroup.deepFree;
```

Ring modulation is a special case of AM and frequency shifting is a special case of AM.

Single-sideband AM shifts all partials by a constant amount but does not preserve the original frequency relationships.

In SuperCollider we can use th FreqShift UGen.

```
SynthDef(\fshift, {arg busIn=0, busOut=0, fshift=0,
                       amp=0, pan=0, mix=0, gate=0;
                    var sIn, sig, env;
                       sIn = In.ar(busIn, 1);
                       sig = FreqShift.ar(sIn, fshift);
                       sig = XFade2.ar(sIn, sig, mix);
                       env = Linen.kr(gate, doneAction:2);
                       sig = sig * env * amp;
                       sig  = Pan2.ar(sig, pan);
                    Out.ar(busOut, sig)
                    }).add;
```

Listing 7.9: Live frequency shifter model

Mouse x → mix between original and shofted sound.

Mouse y → freq shift range (here 0-500 Hz)

```
~xbus = Bus.control(s,1);
~ybus = Bus.control(s,1);

~fxBusIn  = Bus.audio(s, 1);  // Bus fx in

~source = Synth(\mic,   [\busIn,0, \busOut, ~fxBusIn, \amp,1], ~micGrp);
~mousex = Synth(\ksig,  [\type, 0, \range, [0, 1], \busOut, ~xbus], ~recGrp);
~mousey = Synth(\ksig,  [\type, 0, \range, [0, 500],      \busOut, ~ybus], ~
    recGrp);

~ring   = Synth(\fshift, [\busIn, ~fxBusIn,
                          \busOut, 0,
                          \fshift,  ~ybus.asMap, // mouse y
                          \mix,     ~xbus.asMap, // mouse x
                          \amp, 1,
                          \gate, 1], ~fxGrp);
s.defaultGroup.deepFree;
```

7.4.2 Live recording and playback

During a performance we can record some parts played live in one or more buffers and then process them with the techniques illustrated in the dedicated paragraph of chapter 3 (3.3).

Remember that there are two oscillator types:

- sample playback oscillator (*PlayBuf*).
- buffer reading oscillator (*bufRd*).

```
SynthDef(\plbk, {arg buf=0, busOut=0, pos=0, dur=0.2, amp=0,
                     trsp=0, dir=1, pan=0, t_gate=0, done=2;
                  var sig, env;
                     sig = PlayBuf.ar(1, buf,
                                      BufRateScale.kr(buf) * trsp.midiratio
                                                           * dir,
                                      t_gate,
                                      BufSampleRate.kr(buf)* pos
                                   );
                     env = Env.linen(0.01,dur-0.02,0.01);
                     env = EnvGen.kr(env, t_gate, doneAction:done);
                     sig = sig * env * amp;
                     sig = Pan2.ar(sig);
                  Out.ar(busOut, sig)
                  }).add;

SynthDef(\sch, {arg buf=0, busOut=0, smooth=0.02, amp=0, pos=0, pan=0;
                 var pnt, sig;
                     pnt = Lag.kr(pos.linlin(0,1,0,BufFrames.kr(buf)),smooth);
                     sig = BufRd.ar(1, buf, K2A.ar(pnt));
                     sig = Pan2.ar(sig * amp, pan);
                 Out.ar(busOut, sig)
                 }).add;
```

Asynchronous

We can:

- record live sources in one or more Buffers.
- play it back with:
  - simple playback (transposition, direction, cutting, etc.)
  - windowing, enveloping, looping or slicing sequences (both deterministic and non-deterministic),
  - scratching machines.

A simple slicing example controlled by computer keys.

- r trig the buffer recording.
- p trig the processing.
- s stop the porcessing.

```
Buffer.freeAll;
~bufDur  = 3;     // buffer size in seconds
~bufLive = Buffer.alloc(s, s.sampleRate * ~bufDur, 1 );

~bufBusIn = Bus.audio(s, 1);  // Bus to buffer

~source   = Synth(\mic, [\busIn,0, \busOut, ~bufBusIn, \amp,1], ~micGrp);

// triggers from keyboard
w = Window.new("key");
w.front;
w.view.keyDownAction_({arg ...args;
                           switch(args[1],
                                 $r, {Synth(\rec, [\buf, ~bufLive,
                                                                 \busIn, ~bufBusIn
    ,
```

```
                                                                \mix, 0,
                                                                \loop, 0], ~recGrp)},
                                        $p, {r.reset.play},
                                        $s, {r.stop})
                        });
r = Routine.new({
                  inf.do({
                          Synth(\plbk, [\buf, ~bufLive,
                                        \out, rand(2),
                                        \pos, rand(~bufLive.duration)-0.3,
                                        \dur, rrand(0.05,0.3),
                                        \amp, rand(1.0),
                                        \trsp, rand2(5),
                                        \dir, [1,-1].choose,
                                        \pan, rand2(1.0),
                                        \t_gate, 1
                                        ], ~fxGrp);
                          rrand(0.05,0.3).wait
                        })
                 });
s.defaultGroup.deepFree;   w.close.free;
```

A scratching machine controlled by mouse x position.

```
Buffer.freeAll;
~bufDur    = 3;      // buffer size in seconds
~bufLive   = Buffer.alloc(s, s.sampleRate * ~bufDur, 1 );
~bufBusIn  = Bus.audio(s, 1);  // Bus to buffer

~xbus      = Bus.control(s,1); // scratch values

~source = Synth(\mic,    [\busIn,0, \busOut, ~bufBusIn, \amp,1], ~micGrp);
~mousex = Synth(\ksig,   [\type, 0, \range, [0, 1], \busOut, ~xbus], ~recGrp);

// triggers from keyboard
w = Window.new("key");
w.front;
w.view.keyDownAction_({arg ...args;
                           switch(args[1],
                             $r, {Synth(\rec, [\buf, ~bufLive,
                                                   \busIn, ~bufBusIn,
                                                   \mix, 0,
                                                   \loop, 0], ~recGrp)},
                                 $p, {~str = Synth(\sch, [\buf, ~bufLive,
                                                          \busOut, 0,
                                                          \amp, 1,
                                                          \pos, ~xbus.asMap,
                                                          \smooth, 0.2], ~fxGrp)},
                                 $s, {~str.free})
                        });
s.defaultGroup.deepFree;  w.close.free;
```

Let's remember that as position (normalized between 0 and 1) we can use any type of control signal whether coming from external devices (MIDI, OSC, etc.) or from internal oscillators or from feature extractions.

Loop station

The processes illustrated in the previous paragraph are asynchronous.

Here we design a loop station that playback to a regular pulse and its subdivisions.

Let's define a *metronome* synth that:

- generates beats in bpm
- calculates the delta time between them.

It writes the values on two buses.

We then define a player that accepts the synchronized trigger and the delta time as input, with the ability to define beat subdivisions.

This way all players will be synchronized to the same external metronome.

```
SynthDef(\metro, {arg bpm=60, busTrig=99, busDur=100;
                   var bps, sec, sig;
                       bps = bpm/60*100;  // bpm to Hz / 100 parts
                       sig = Impulse.ar(bps);
                       sec = 60/bpm;      // Delta in seconds (duration)
                   Out.ar(busTrig,sig);   // Trigger on audio Bus
                   Out.kr(busDur,sec);    // Duration on contro Bus
                   }).add;
SynthDef(\loop, {arg buf=0, busTrig=99, busDur=100, busOut=0,
                       pos=0, sudd=1, legato=0, amp=0,
                       dir=1, gate=0, pan=0, done=2;
                  var dur, trig, sig, env, fade;
                       dur  = In.kr(busDur) * (1/sudd);       // scaled
                       trig = PulseDivider.ar(In.ar(busTrig), 100/sudd);
                       sig  = PlayBuf.ar(1, buf,
                                         BufRateScale.kr(buf) * dir,
                                         trig,      // external trigger
                                         BufFrames.kr(buf) * pos);
                       env  = Env.linen(0.01,(dur * legato)-0.02,0.01);
                       env  = EnvGen.ar(env, trig); // external trigger
                       fade = Linen.kr(gate, doneAction:done);
                       sig  = sig * fade * env * amp;
                       sig  = Pan2.ar(sig, pan);
                  Out.ar(busOut, sig)
                  }).add;
```

Listing 7.10: Live looping machine model

Let's assemble everything and record the Buffer.

```
~bpm     = 60;
~dur     = 3;

Buffer.freeAll;
~bufLive  = Buffer.alloc(s, s.sampleRate * ~dur);
~bufBusIn = Bus.audio(s, 1);  // Bus to buffer

~trigBus = Bus.audio(s);      // Bus Trigger (loopers)
~durBus  = Bus.control(s);    // Bus delta (loopers)

~source = Synth(\mic,   [\busIn,0, \busOut, ~bufBusIn, \amp,1], ~micGrp);
~pulse  = Synth(\metro, [\bpm,  ~bpm,  \busTrig, ~trigBus, \busDur, ~durBus], ~
    recGrp);

// triggers from keyboard
w = Window.new("key");
w.front;
w.view.keyDownAction_({arg ...args;
                          if(args[1]==$r)
                        {Synth(\rec, [\buf, ~bufLive,
```

```
                                                \busIn, ~bufBusIn,
                                                \mix, 0,
                                                \loop, 0], ~recGrp)}
                        });
```

Play it.

```
~loop1  = Synth(\loop,  [\buf,  ~bufLive,
                         \busTrig, ~trigBus,
                         \busDur,  ~durBus,
                         \busOut, 0,
                         \pos, 0,   // 0-1
                         \sudd, 1,
                         \legato,1, // 0-1
                         \amp, 1,
                         \dir, 1,
                         \gate, 1], ~fxGrp);

~loop1.set(\pos, 0.1); // change parameters
~loop1.set(\legato, 0.7);
~loop1.set(\sudd, 2);
~loop1.set(\amp, 0.7);
```

Let's make a second looper.

```
~loop2  = Synth(\loop,  [\buf,  ~bufLive,
                         \busTrig, ~trigBus,
                         \busDur,  ~durBus,
                         \busOut, 0,
                         \pos, 0.8,
                         \sudd, 2,
                         \legato,0.3,
                         \amp, 1,
                         \dir, 1,
                         \gate, 1], ~fxGrp);

~loop2.set(\pos, 0.7); // change parameters
~loop2.set(\sudd, 5);
~loop1.set(\gate, 0); ~loop2.set(\gate, 0);
s.defaultGroup.deepFree;
```

We can work with many buffers and many loopers recording asynchronously thus creating complex rhythmic and harmonic grids.

7.4.3 Granular synthesis

We can perform live granular synthesis in two ways:

- asynchronous.
- synchronous.

Asynchronous

In this mode, we record the buffer in one of two possible ways (one shot or loop) and play it back with one of the synths presented in the computer music chapter (4.3.4).

An example.

The SynthDefs.

```
SynthDef(\grain_1,{arg busOut=0, buf=0, pos=0, trig=0,
                      gdur=0.1, amp=0, trsp=0, pan=0,
                      gate=0, done=2;
                  var gfreq, sig, env;
                      gfreq = 2.0/gdur;        // 2 overlap...
                      pos   = pos * BufDur.kr(buf);
                      sig   = TGrains.ar(2,    // channels
                                         trig,
                                         buf,
                                         trsp.midiratio,
                                         pos, // in seconds
                                         gdur,
                                         pan, // depends o chls number
                                         amp,
                                         4    // interpolation 1,2 4
                                           );
                    env   = Linen.kr(gate,doneAction:done);
                      sig   = sig * amp * env;
                  Out.ar(busOut, sig * env);
                    }).add;
```

Indipendent recorder.

Indipendent player.

- r trig the buffer recording.
- p trig the processing.
- s stop the processing.

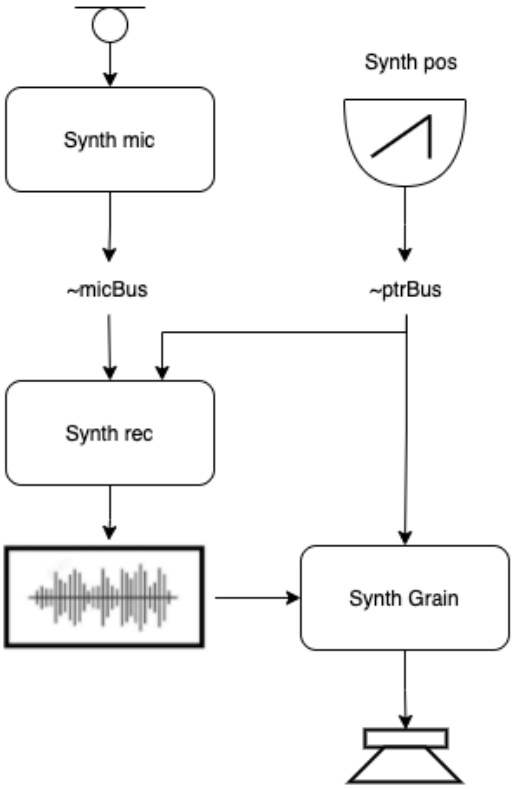
```
Buffer.freeAll;
~bufDur   = 3;     // buffer size in seconds
~bufLive  = Buffer.alloc(s, s.sampleRate * ~bufDur, 1 );
~bufBusIn = Bus.audio(s, 1);  // Bus to buffer

~kpos  = Bus.control(s, 1);
~ktrig = Bus.control(s, 1);

~source  = Synth(\mic,   [\busIn,0, \busOut, ~bufBusIn, \amp,1], ~micGrp);

~pointer = Synth(\ksig,  [\type, 2, \freq, 2, \busOut, ~kpos], ~recGrp);
~trigger = Synth(\ksig,  [\type, 4, \freq, 15, \busOut, ~ktrig], ~recGrp);

// triggers from keyboard
w = Window.new("key");
w.front;
w.view.keyDownAction_({arg ...args;
                          switch(args[1],
                                $r, {Synth(\rec, [\buf, ~bufLive,
                                                  \busIn, ~bufBusIn,
                                                  \mix, 0,
                                                  \loop, 0], ~recGrp)},
                                    $p, {~gr = Synth(\grain_1, [\busOut, 0,
                                                                \buf, ~bufLive,
                                                                \pos, ~kpos.
    asMap,
                                                                \trig, ~ktrig.
    asMap,
                                                                \gdur, 0.1,
                                                                \trsp, 0,
                                                                \amp, 1,
                                                                \pan, 0,
```

```
                                                                    \gate, 1], ~fxGrp);
                                        },
                                        $s, {~gr.set(\gate, 0)})
                            });
s.defaultGroup.deepFree; w.close.free;
```

Synchronous (harmonizer)

If we want a real live granular synthesis we must sync:

- position in buffer recording Synth.
- position in buffer playback Synth



The sync phasor and the recorder.

```
SynthDef(\pos, {arg busOut=0,buf=0,rate=1;
               var sig;
                   sig = Phasor.ar(0,
                           BufRateScale.kr(buf) * rate, // incremental factor
                   0, BufFrames.kr(buf));       // start - end
               Out.ar(busOut, sig);
                 }).add;

SynthDef(\syncrec, {arg busIn=0, busPos=0, buf=0;
                   var ptr, sig;
                       ptr = In.ar(busPos, 1);
                       sig = In.ar(busIn, 1);
                   BufWr.ar(sig, buf, ptr);
                     }).add;

// N.B. rate=0 stop at last sample position
```

Listing 7.11: Live sync phasor and buffer recorder model

Then we can use any granular synth from chapter 3 with few changes:

- audio bus for pointer (not normailzed) instead a control bus.
- position signal is not normalized.

```
SynthDef(\gharm,{arg busOut=0,  busPos=0,  buf=0, trig=0,
                    gdur=0.1, amp=0, trsp=0, pan=0,
                    gate=0, done=2;
               var pos, sig, env;
                   pos    = In.ar(busPos,1) / SampleRate.ir(buf);
                   sig    = TGrains.ar(2, trig, buf, trsp.midiratio, pos, gdur,
    pan, amp, 4);
                  env    = Linen.kr(gate,doneAction:done);
                      sig    = sig * env;
               Out.ar(busOut, sig);
}).add;
```

Test it.

```
Buffer.freeAll;
~bufDur    = 3;      // buffer size in seconds
~bufLive   = Buffer.alloc(s, s.sampleRate * ~bufDur, 1 );
~bufBusIn = Bus.audio(s, 1);  // Bus to buffer

~kpos   = Bus.audio(s, 1);      // audio bus...
~ktrig = Bus.control(s, 1);

~source   = Synth(\mic,      [\busIn,0, \busOut, ~bufBusIn, \amp,1], ~micGrp);
~pointer = Synth(\pos,      [\buf, ~bufLive, \busOut, ~kpos], ~micGrp);
~record   = Synth(\syncrec, [\busIn,~bufBusIn, \busPos, ~kpos, \buf,~bufLive], ~
    recGrp);
~trigger = Synth(\ksig,     [\type, 4, \freq, 15, \busOut, ~ktrig], ~recGrp);

~grain    = Synth(\gharm, [\busOut, 0,
                           \buf,  ~bufLive,
                           \busPos, ~kpos,
                           \trig, ~ktrig.asMap,
                           \gdur, 0.1,
                           \trsp, 4,
                           \amp,  1,
                           \pan,  0,
                           \gate, 1], ~fxGrp);

~trigger.set(\freq,20);  // grain freq
~grain.set(\trsp,4);     // trasposition
(
// granulator
~trigger.set(\type,5, \freq,25); // trigger type
~grain.set(\trsp,4, \gdur, 0.1);
)

s.defaultGroup.deepFree;
```

If we change the frequency of the control signal:

- freq $<$ 20Hz $\rightarrow$ audible grains.
- freq $>$ 20Hz $\rightarrow$ harmonizer.

If we want only an harmonizer we can use a dedicate UGens: *PitchShift* that have an internal buffer.

```
SynthDef(\harm,{arg busIn=0, busOut=0, trsp=0,
                    gdur=0.1, prand=0, trand=0, amp=0, pan=0,
```

```
                    gate=0, done=2;
              var sig, env;
                    sig = In.ar(busIn);
                  sig = PitchShift.ar(sig,
                                        gdur,            // grin dur
                                        trsp.midiratio, // rate
                                        prand,  // pitch chorus
                                        trand); // time chorus < gdur
                env = Linen.kr(gate,doneAction:done);
                    sig = sig * env * amp;
                    sig = Pan2.ar(sig, pan);
              Out.ar(busOut, sig);
}).add;
```

Listing 7.12: Live granular harmonizer model

A chord exemple.

```
~fxBusIn = Bus.audio(s, 1);  // Bus to buffer

~chord  = [ -5,   0,   4,   7];   // major chord
~amps   = [0.1, 0.4, 0.3, 0.2];   // amps
~source = Synth(\mic,  [\busIn,0, \busOut, ~fxBusIn, \amp,1], ~micGrp);

~harm = ~chord.collect({arg i, id;
                        Synth(\harm, [\busIn, ~fxBusIn,
                                      \busOut, 0,
                                      \gdur, 0.2,
                                      \trsp, ~chord[id],
                                      \amp,  ~amps[id],
                                      \pan,  0,
                                      \gate, 1], ~fxGrp)
                        });

~harm.do({arg i; i.set(\gate, 0)}); ~source.free;
s.defaultGroup.deepFree;
```

7.4.4 Delay lines

Delay lines can be used for:

- the creation of augmented instrumental gestures.
- the generation of sonic/musical backgrounds that evolve over time.

We take care to mask or musically underline the mechanical characteristics intrinsic to this process.

There are different types of delays:

- echo.
- feedback gain.
- feedback duration.
- ping pong.
- multi tap.
- variable time delays.
- filtred.
- resonator.

Echo

A simple delay without feedback.

- max delay → internal buffer size.
- delay < max delay.
- mix → with or without original sound.

The model.

```
SynthDef(\echo, {arg busIn=0, busOut=0, maxdel=10, del=0.5, mix=0,
                     amp=0, gate=0, pan=0, done=2;
                 var sig, dly, env;
                     sig = In.ar(busIn);
                     dly = DelayL.ar(sig, maxdel, del);
                     dly = LeakDC.ar(dly);
                     env = Linen.kr(gate,doneAction:done);
                     sig = XFade2.ar(dly, sig, mix); // mix -1 = only dly,
                                                     // 0 = both,
                                                     // 1 = only direct sound
                                                     // float = ratio
                     sig = sig * env * amp;
                     sig = Pan2.ar(sig, pan);
                 Out.ar(busOut, sig)
                 }).add;
```

Listing 7.13: Live echo model

Test it.

```
~fxBusIn = Bus.audio(s, 1);

~source = Synth(\mic,  [\busIn,0, \busOut, ~fxBusIn, \amp,1], ~micGrp);
~delay  = Synth(\echo, [\busIn,~fxBusIn,
                        \busOut, 0,
                        \maxdel, 3,
                        \del, 1,
                        \mix, 0,
                        \amp,1,
                        \gate, 1], ~fxGrp);

s.defaultGroup.deepFree; ~fxBusIn.free; ~source.free; ~delay.free;
```

Feedback gain

We can add a feedback.

Duration of feedback is not directly controlled.

It depend from feedback gain (0.0 - 1.0).

The model.

```
SynthDef(\del_1, {arg busIn=0, busOut=0, maxdel=10, del=0.5, fbk=0.0, mix=0,
                     amp=0, gate=0, pan=0, done=2;
                 var in, local, sig, dly, env;
                     in    = In.ar(busIn);
                     local = LocalIn.ar(1);   // feedback internal bus
                     sig   = in + local;
                     dly   = DelayL.ar(sig, maxdel, del);
                     dly   = LeakDC.ar(dly);  // long feedback
                 LocalOut.ar(dly * fbk);      // feedback internal bus
                     env   = Linen.kr(gate,doneAction:done);
                     sig   = XFade2.ar(dly, sig, mix) * env * amp;
                     sig  = Pan2.ar(sig, pan);
                 Out.ar(busOut, sig)
                 }).add;
```

Listing 7.14: Live feedback gain delay model

Test it.

```
~fxBusIn = Bus.audio(s, 1);

~source = Synth(\mic,  [\busIn,0, \busOut, ~fxBusIn, \amp,1], ~micGrp);
~delay  = Synth(\del_1, [\busIn,~fxBusIn,
                         \busOut, 0,
                         \maxdel, 3,
                         \del, 1,
                         \fbk, 0.5,
                         \mix, 0,
                         \amp,1,
                         \gate, 1], ~fxGrp);

s.defaultGroup.deepFree; ~fxBusIn.free; ~source.free; ~delay.free;
```

Feedback duration

If we want to control the duration of the feedback in seconds (decayT) we can use the *CombL* UGen.

The model.

```
SynthDef(\del_2, {arg busIn=0, busOut=0, maxdel=10, del=0.5, decayT=1, mix=0,
                      amp=0, gate=0, pan=0, done=2;
                  var in, sig, dly, env;
                      in  = In.ar(busIn);
                      dly = CombL.ar(in, maxdel, del, decayT);
                      env = Linen.kr(gate,doneAction:done);
                      sig = LeakDC.ar(dly);    // long feedbacks
                      sig = XFade2.ar(dly, in, mix);
                      sig = sig * env * amp;
                      sig = Pan2.ar(sig, pan);
                  Out.ar(busOut, sig)
                  }).add;
```

Listing 7.15: Live feedback duration delay model

Test it.

```
~fxBusIn = Bus.audio(s, 1);

~source = Synth(\mic,  [\busIn,0, \busOut, ~fxBusIn, \amp,1], ~micGrp);
~delay  = Synth(\del_2, [\busIn,~fxBusIn,
                         \busOut, 0,
                         \maxdel, 3,
                         \del, 0.25,
                         \decayT, 3,
                         \mix, 0,
                         \amp,1,
                         \gate, 1], ~fxGrp);

s.defaultGroup.deepFree; ~fxBusIn.free; ~source.free; ~delay.free;
```

Ping pong

A delay line with alternate panning.

The model.

```
SynthDef(\pingpong, {arg busIn=0, busOut=0, maxdel=2, del=0.5, fbk=0.0,
                        amp=0, gate=0, done=2;
                    var in, local, sig, dly, env;
                        in    = In.ar(busIn);
                        local = LocalIn.ar(2) + [in, 0];
                        dly   = DelayL.ar(local, maxdel, del, 1);
                        dly   = LeakDC.ar(dly);
                        env   = Linen.kr(gate,doneAction:done);
                    LocalOut.ar(dly.reverse * fbk); // reverse = ping pong
    effect
                    Out.ar(busOut, local * env * amp)
                    }).add;
```

Listing 7.16: Live ping-pong delay model

Test it.

```
~fxBusIn = Bus.audio(s, 1);

~source = Synth(\mic,  [\busIn,0, \busOut, ~fxBusIn, \amp,1], ~micGrp);
~delay  = Synth(\pingpong, [\busIn,~fxBusIn,
                            \busOut, 0,
                            \maxdel, 3,
                            \del, 0.25,
                            \fbk, 0.5,
                            \mix, 0,
                            \amp,1,
                            \gate, 1], ~fxGrp);

s.defaultGroup.deepFree; ~fxBusIn.free; ~source.free; ~delay.free;
```

Multi Tap

Delay lines with different times and amplitudes.

The model.

```
SynthDef(\multitap, {arg busIn=0,busOut=0,buf=0,
                        times=#[0.1,0.5,1.2,0.8], // delay times
                        amps =#[1,  0.5,0.8,0.3], // delay amps
                        mix= 0,
                        amp=0,pan=0,gate=0,done=2;
                    var in, env, del, sig;
                        in  = In.ar(busIn);
                        env = Linen.kr(gate,doneAction:done);
                        del = MultiTap.ar(`times, `amps, in,bufnum:buf);
                        sig = XFade2.ar(del, in, mix);
                        sig = sig * amp * env;
                        sig = Pan2.ar(sig, pan);
                    Out.ar(busOut, sig)
                    }).add;
```

Listing 7.17: Live multi tap delay model

Test it.

```
~fxBusIn = Bus.audio(s, 1);

b = Buffer.alloc(s, s.sampleRate * 4, 1); // a Buffer

~source = Synth(\mic, [\busIn,0, \busOut, ~fxBusIn, \amp,1], ~micGrp);
```

```
~delay  = Synth(\multitap, [\busIn,~fxBusIn,
                            \busOut, 0,
                            \buf, b,
                            \times, [0.1,0.3,0.6,1.2],
                            \amps,   [1,  0.5,0.8,0.3],
                            \amp,1,
                            \gate, 1], ~fxGrp);

s.defaultGroup.deepFree; ~fxBusIn.free; ~source.free; ~delay.free; b.free;
```

Variable time

If we use a continuous control signal that modulates the delay time we obtain transpositions and strange glitches.

The model.

```
SynthDef(\vardel, {arg busIn=0, busOut=0,
                       maxdel=10, rate=3, decayT=1,
                       mix=0, delMin=0.1,delMax=0.4,
                       amp=0, gate=0, pan=0, done=2;
                   var in, sig, ksig, dly, env;
                       in  = In.ar(busIn);
                       ksig = LFNoise2.kr(rate)
                                      .exprange(delMin,delMax); // ksig
                       dly = CombL.ar(in, maxdel, ksig, decayT);
                       env = Linen.kr(gate,doneAction:done);
                       sig = LeakDC.ar(dly);    // Per fbk lunghi
                       sig = XFade2.ar(dly, in, mix) * env * amp;
                       sig = Pan2.ar(sig, pan);
                   Out.ar(busOut, sig)
                   }).add;
```

Listing 7.18: Live variable time delay model

Test it.

```
~fxBusIn = Bus.audio(s, 1);

~source = Synth(\mic,  [\busIn,0, \busOut, ~fxBusIn, \amp,1], ~micGrp);
~delay  = Synth(\vardel, [\busIn,~fxBusIn,
                          \busOut, 0,
                          \maxdel, 4,
                          \rate, 5, // try change it
                          \decayT, 4,
                          \delMin,  0.1,
                          \delMax, 0.3,
                          \amp,1,
                          \gate, 1], ~fxGrp);

s.defaultGroup.deepFree; ~fxBusIn.free; ~source.free; ~delay.free;
```

Filtred

We can filter the delayed sound to partially mask the mechanical nature of the feedback making the sound gesture more natural.

OnePole filter:

- coeff -0.95 → High pass.
- coeff +0.95 → Low pass.

The model.

```
SynthDef(\delfilt, {arg busIn=0, busOut=0,
                        maxdel=10, del=0.5, fbk=0.0,
                        mix=0, coef=0,
                        amp=0, gate=0, pan=0, done=2;
                    var in, local, sig, dly, env;
                        in    = In.ar(busIn);
                        local = LocalIn.ar(1);
                        sig   = in + local;
                        dly   = DelayL.ar(sig, maxdel, del);
                        dly   = OnePole.ar(dly,coef); // Filter
                        dly   = LeakDC.ar(dly);
                    LocalOut.ar(dly * fbk);      // Feedback sig
                        env = Linen.kr(gate,doneAction:done);
                        sig = XFade2.ar(dly, sig, mix);
                        sig = sig * env * amp;
                        sig = Pan2.ar(sig, pan);
                    Out.ar(busOut, sig)
                    }).add;
```

Listing 7.19: Live filtred delay model

Test it.

```
~fxBusIn = Bus.audio(s, 1);

~source = Synth(\mic,  [\busIn,0, \busOut, ~fxBusIn, \amp,1], ~micGrp);
~delay  = Synth(\delfilt, [\busIn,~fxBusIn,
                           \busOut, 0,
                           \maxdel, 4,
                           \del, 0.5,
                           \fbk, 0.4,
                           \coef,  0.95,
                           \amp,1,
                           \gate, 1], ~fxGrp);

~delay.set(\coeff, rand2(0.95).postln); // try to change

s.defaultGroup.deepFree; ~fxBusIn.free; ~source.free; ~delay.free;
```

Resonator

Similar to the Karplus-Strong algorithm.

The model.

```
SynthDef(\resondel, {arg busIn=0, busOut=0, decayT=0.2, reson=0.01,
                        amp=0, gate=0, pan=0, done=2;
                    var in, dly, env, sig;
                        in    = In.ar(busIn);
                        reson = MouseY.kr(0.0001,0.01);
                        reson.reciprocal.poll(10);      // frequency
                        dly   = CombL.ar(in, 0.01, reson, decayT);
                        env   = Linen.kr(gate,doneAction:done);
                        sig   = dly * env * amp;
                        sig   = Pan2.ar(sig, pan);
                    Out.ar(busOut, sig)
                    }).add;
```

Listing 7.20: Live resonator delay model

Test it.

```
~fxBusIn = Bus.audio(s, 1);

~source = Synth(\mic,  [\busIn,0, \busOut, ~fxBusIn, \amp,1], ~micGrp);
~delay  = Synth(\resondel, [\busIn,~fxBusIn,
                            \busOut, 0,
                            \decayT, 0.2,
                            \reson, 0.01,
                            \amp,1,
                            \gate, 1], ~fxGrp);

~delay.set(\reson, rrand(0.0001,0.01).postln);

s.freeAll;
```

7.4.5 Multichannel panning

How to position a monophonic signal in a two or three-dimensional space delimited in various ways by two or more loudspeakers.

Three possible ways of thinking about space in the use of stereophonic and multichannel diffusion systems:

- acoustic space within which to position the different monophonic sources.
  - listeners should not perceive the presence of the loudspeakers as if the sounds were coming from the walls of the concert hall.
  - ambisonic, wavefield synthesis, bineural systems.
- loudspeakers are a sound diffusion tool with musical and historical value.
  - acousmonium, stereophonic, quadriphonic, and octophonic systems.
- loudspeakers take on the same value as individual acoustic instruments, characters or sound sculptures.
  - louspeakers as musical instruments.

```
s.options.numOutputBusChannels_(8);
s.reboot;
s.plotTree;
s.scope;
s.meter;
SynthDef(\ksig,{arg type=0, range=#[0,1], freq=1, busOut=0;
                var sig;
                    sig = Select.kr(type,
                                    [MouseX.kr(range[0], range[1]), // 0
                                     MouseY.kr(range[0], range[1]), // 1
                                     LFNoise1.kr(freq).range(range[0], range[1]), // 2
                                     LFSaw.kr(freq).range(range[0], range[1]), // 3
                                     Impulse.kr(freq),               // 4
                                     Dust.kr(freq),                  // 5
                                     ]);
                    Out.kr(busOut,sig)
                    }).add;
```

If we want multichannels systems we must set it before booting Server.

Mono, dual-mono and multi-mono panning

Related to third typology → loudspeaker as music instument.

In SuperCollider are represented with *public out buses*.

Bus 0 → leftmost loudspeakeer.

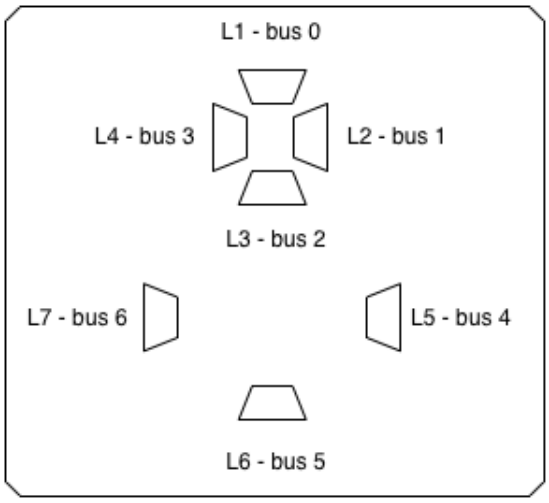
```
SynthDef(\out,{arg busOut = 0;
                var sig, env;
                    sig = SinOsc.ar;
                    env = Env.perc;
                    env = EnvGen.kr(env, 1, doneAction:2);
                    sig = sig * env;
                Out.ar(busOut, sig)
                }).add;

Synth(\out, [\busOut, rand(8)]);
```

If we duplicate a monophonic signal by writing it on two adjacent channels → dual mono.

In SuperCollider we can duplicate objects by *!* shortcut.

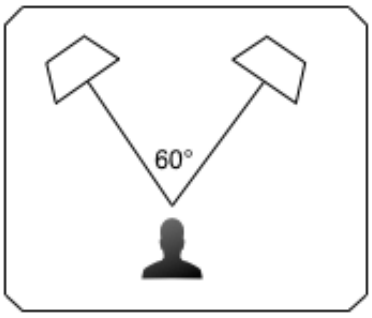
```
SynthDef(\out2,{arg busOut = 0;
                 var sig, env;
                     sig = SinOsc.ar;
                     env = Env.perc;
                     env = EnvGen.kr(env, 1, doneAction:2);
                     sig = sig * env;
                 Out.ar(busOut, sig!2)
                 }).add;

Synth(\out2, [\busOut, rand(8)]);
```

Multi-monophonic diffusion systems lend themselves to interesting artistic expressive solutions of mirrored sounds and spatial sound references between amplified acoustic instruments and electroacoustic sounds.



Stereo

Two front speakers ideally positioned at a 60° angle to the optimal listening point.



It is a one-dimensional frontal system mainly attributable to the first type of musical space types.

To obtain a stereo signal we must take a monophonic signal and distribute its amplitude between two adjacent channels.

To correct problems related to human perception the distribution is not linear.

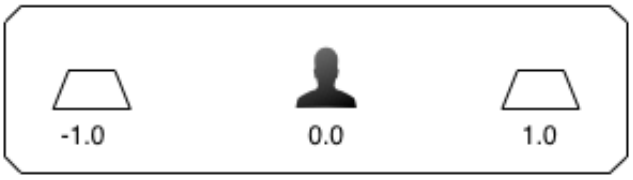
In SuperCollider we can use the *Pan2* UGen.

We can also simulate distance by acting on the amplitude of the signal.

Parameters are:

- position → -1 to 1.
- distance → 0.001 to 1.
- movement duration.

We can control them in the usual ways (MIDI, OSC, Envelopes, LFO, etc.).




```
SynthDef(\stereo, {arg pos=0,ptime=0,dis=1,gate=0;
                    var sig, env;
                        sig = SinOsc.ar(Rand(400,2000));
                        env = Linen.kr(gate,doneAction:2);
                        sig = sig * env;
                        sig = Pan2.ar(sig, pos.lag(ptime), dis.lag(ptime));
                    Out.ar(0, sig)
                    }).add;
```

Listing 7.21: Stereo pan model

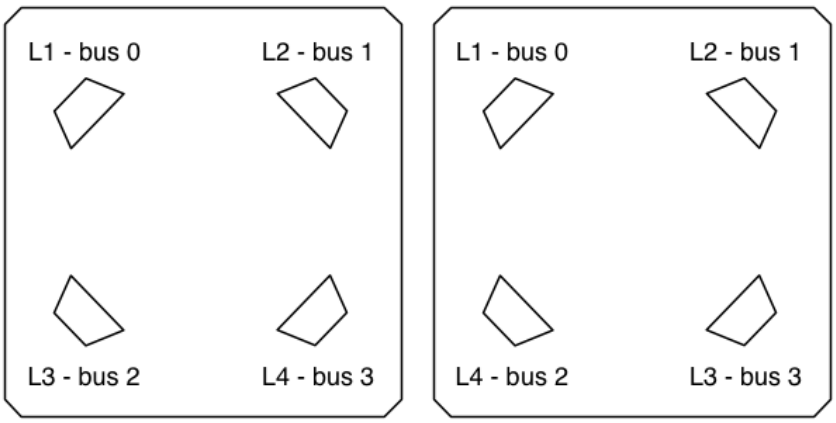
Test it

```
~synth = Synth(\stereo, [\gate,1]);

~synth.set(\pos,rand2(1.0),\ptime,1);
~synth.set(\gate,0);
```

Listing 7.22: Stereo pan model

Quadriphony

Four speakers ideally placed at the corners of an imaginary square.

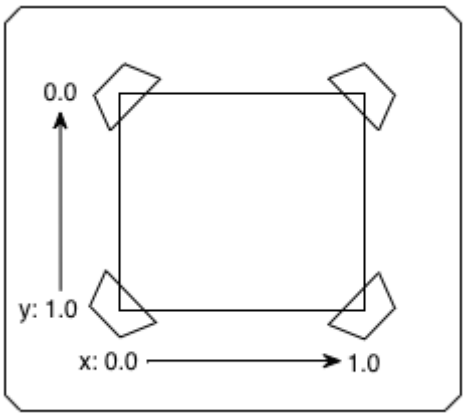


Speaker numbering (L1 - L4) can be done in two ways:

- double stereo (left).
- circular (right).

As in stereo systems we have to divide the amplitude of each source (monophonic signal) not over two but over four loudspeakers.

We define the square as a Cartesian plane on which we specify each point as x/y coordinates between 0.0 and 1.0.



In SuperCollider we can use *Pan4* UGen similar to *Pan2*.

```
SynthDef(\quad, {arg xpos=0, ypos=0,ptime=0,gate=0;
                  var sig, env;
                      sig = SinOsc.ar(Rand(400,2000));
                      env = Linen.kr(gate,doneAction:2);
                      sig = sig * env;
                      sig = Pan4.ar(sig, xpos.lag(ptime), ypos.lag(ptime));
                  Out.ar(0, sig)
                  }).add;
```

Listing 7.23: Quadriphonic pan model

Test it.

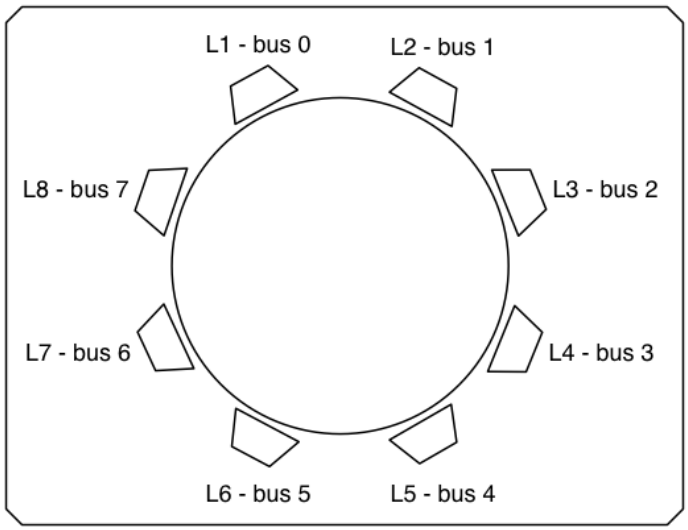
```
~synth = Synth(\quad, [\gate,1]);

~synth.set(\xpos,rand2(1.0),\ypos,rand2(1.0),\ptime,1)
~synth.set(\gate,0);
```

Listing 7.24: Stereo pan model

Circular systems

Can be from 3 to 128 channels.

Five, six or eight are multichannel standards.



In SuperCollider we can use the *PanAz* UGen.

Parameters are:

- n_chans → number of channels.
- sig → source (mono channel).
- pos → position - from 0.0 to 2.0
  - delta between two adiacent channels → 2/n_channels.
- level → amplitude.
- width →.
  - 2 → spread between two adiacent channels.
  - 4 → spread between four adiacent channels.
  - 6 → spread between six adiacent channels.
  - etc.
- orientation → set leftmost bus channel.
  - 0.0 → L1 - bus 0.
  - 0.5 → between L1 - bus 0.

```
SynthDef(\circular, {arg pos=0, width=2, orient=0.5, gate=0;
                      var sig, env;
                          sig = SinOsc.ar(800);
                          env = Linen.kr(gate,doneAction:2);
                          sig = sig * env;
                          sig = PanAz.ar(8,  // numChans
                                         sig,
                                         pos, // 0 to 2
                                         1,   // level
                                         width,
                                         orient);
                      Out.ar(0, sig)
                      }).add;
```

Listing 7.25: Circular pan model

Test it.

```
~synth = Synth(\circular, [\gate,1]);

~pos = [0.0, 0.25, 0.50, 0.75, 1.00,  1.25,  1.50,  1.75]; // 0 to 2
~pos = [0.0, 0.25, 0.50, 0.75, 1.00, -0.75, -0.50, -0.25]; // -1 to 1 is the
    same

r = Routine({
          inf.do({arg i;
                ~synth.set(\pos, ~pos.wrapAt(i));
              1.wait
                  })
            }).reset.play;
```

Change orientation, see and listen.

```
~synth.set(\orient, 0);  // or 0.5
```

Change width, see and listen.

```
~synth.set(\orient, 0.5);
~synth.set(\width, 2); // try 2, 4, 6 or 8

~synth.set(\gate,0); r.stop.free;
```

If we map a control signal (-1, 1) to the position we get circular or random movements depending on the type of signal.

```
~kbus = Bus.control(s,1);

~ksig  = Synth(\ksig,  [\type, 3, \range, [-1, 1], \freq, 0.2, \busOut, ~kbus])
    ;
~synth = Synth(\circular, [\pos, ~kbus.asMap, \gate,1]);

~ksig.set(\type, 0); // mouse x
~ksig.set(\type, 2); // random zig zag
~synth.set(\gate,0); ~ksig.free; ~kbus.free;
```

7.5 Composition sketches proposal

Design and realize a work for a live source and electronic sounds.

Combinations can be:

- instrument and pre-recorded sounds.
- instrument and live computer-generated sounds.
- instrument and live processing.

By instrument we mean:

- musical instrument from any culture.
- voice (sung or recited).
- any type of acoustic source.

The work can be in the form of:

- art music composition.
- improvisation.
- installation.
- sound art project (sound-sculpture or site specific soundscape).

Appendices

Appendix A

SuperCollider basics

Open Source Software

It uses a programming language (Smalltalk) that follows the Object Oriented Programming (OOP) paradigm.

We write the commands as lines of code saved in text files called *scripts*.

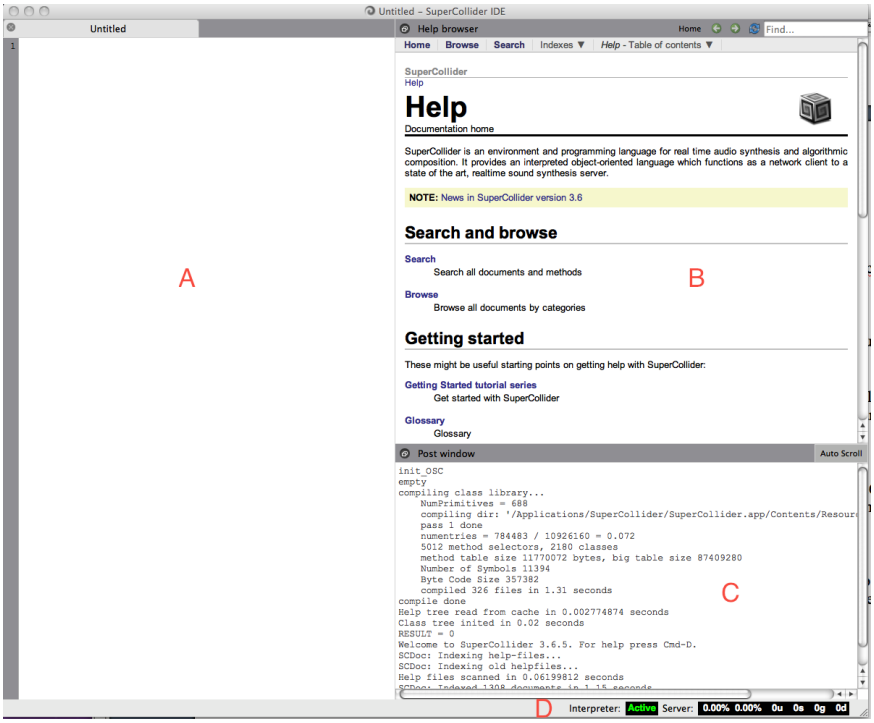
A.1 User interface

Let's launch the application.

The figure below shows the home screen of the most recent versions (IDE).

Identical on all operating systems.

Divided into four parts:



- A - Interpreter → the space where the code is written.
- B - Help browser → the space where we can consult help files and documentation.

- C - Post window → a space where you can read and post various useful information about what's happening in SuperCollider.
- D → the space where information about the Interpreter and the Server is displayed.

A.2 Architecture

Two separate applications that communicate via the OSC protocol.

- Interpreter (or client or sclang).
  - where we write the commands to be executed by the computer in the form of code.
  - it work at schedule time.
  - if we do not perform any action (such as executing the code) or generate some kind of automation (Routine or interaction) it does not return any value.
- Server (or scsynth).
  - it works behind the scenes, we don't see it.
  - it is the SuperCollider sound synthesis engine.
  - it receives instructions from the Interpreter and it builds the virtual instruments necessary to generate or process an audio signal.
  - it work at a sample rate.

All audio signals, internal, incoming and outgoing, exist only on the Server.

Interpreter → sheet containing:

- instructions needed to build a virtual musical instrument.
- instrumental techniques with which it will be played.
- a score indicating the actions to be performed over time to play it.

This sheet is sent via OSC to the Server, which like a virtual luthier/instrumentalist:

- interprets the assembly instructions.
- based on these he builds the virtual instrument and then.
- play it by performing actions specified in a score or those sent in real time by a real or virtual performer external to the software.

When we launch the application, two default Servers are automatically created:

- internal: a server internal to the client application, to be considered a *service*.
- localhost: A server independent of the client application, on the same or a different computer. This is the one normally used.

A.3 Main commands

A.3.1 New file

MAC → cmd + n
WIN → ctrl + n

A.3.2 Open file

MAC → cmd + o
WIN → ctrl + o

After creating or opening a file, if we want to make SuperCollider perform an action (such as performing a mathematical operation, writing a score, defining a synthesis algorithm, changing the parameters of a Synth, making it play, stopping it, etc.) we must:

- write one or more lines of code in the interpreter containing a sequence of instructions.
- select one or more lines of code with the mouse.
- run (evaluate) the selected code by pressing cmd + enter or shift + enter.

A.3.3 Lines of code

Interpreter is divided into lines numbered 1 to n.

Each line should end with a *;* (semicolon).

If we type on multiple lines but don't put a semicolon before the line break SuperCollider interprets them as a single line.

If we write code on the same line after a semicolon SuperCollider interprets it as if it were written one line below.

```
1+2;           // this is a line
(              // but also from here...
1+
2/
3;             // ...to here
1 + 2; 3 * 7; // these are two lines
)
```

We can group multiple lines into code blocks by enclosing them in parentheses.

```
(

"ciao".postln;
a = 12 +24;       // this is a code block
a.postln;

)
```

A.3.4 Evaluate the code

Three options:

1. one line at a time, placing the cursor anywhere on the line we want to execute → cmd + enter.

```
2 + 2;  // see result in the Post window
```

2. an entire code block by selecting it with the mouse (click and drag) or by double-clicking on the first or last parenthesis.

```
(
13.postln;
[12,"ciao", \miao, 45.567].postln;
(12 + 56).postln
)
```

3. the entire code by selecting *Evaluate File* from the *Language* menu.

A.3.5 Execution order

In these last two modes, the operations are performed consecutively:

- from left to right.
- line after line.
- from top to bottom (in almost all cases).

at the max speed provided by the power of the computer.

A.3.6 Error messages

Let's evaluate the following code which intentionally contains an error (the comma before 67 is missing).

```
a = [12,34,56 67]
```

Let's read what SuperCollider tells us in the post window: the arrows or the dot indicate where the compilation stopped, so the error will necessarily be earlier and it's up to us to find it.

A.3.7 Clear Post window

MAC → cmd + shift + p
WIN → ctrl + shift + p

A.3.8 Comments

- to facilitate memory in case we want to modify a patch after some time.
- give it to someone who would be helped in understanding the code.

Comments are colored red and there are two possible syntaxes:

1. on one line.

```
2 + 2; // sum
```

2. on multiple lines.
3. on one line.

```
/*

...comment...

*/
```

A.3.9 Save file

MAC → cmd + s
WIN → alt + s

A pop-up menu opens where we can specify four different formats:

- all Files (filename.txt). Generic format, can be read by any text editor such as TextEdit, Word, LibreOffice, etc.
- SuperCollider Document (filename.scd). SuperCollider document format. If SuperCollider isn't already open, double-clicking an .scd file icon will open the software automatically.
- SuperCollider Class File (filename.sc). If we want to save a class we wrote.
- SuperCollider Help Source (filename.schelp). If you want to save a Help file you've written.

N.B. Please note that in these formats the code written in the interpreter is saved in the form of a text file, not the audio eventually generated by it.

A.3.10 Data types

As in all formal languages there are several data types that define:

- amount of memory they occupy.
- operations we can perform on them.

The main ones are.

```
12              // Int
12.23           // Float
\ciao           // Symbol
'ciao␣ciao'     // Symbol
$c              // Char
"vado␣al␣mare"  // String

3   +  5;       // # 8
"3" + "5";      // # 35
```

A.4 Audio

A.4.1 Server boot

If we want to generate or process audio signals, we must first power on the server (boot) and then evaluate the code.

We can compare this action to turning on or off the power supply to an electroacoustic system (player, mixer, amplifier, and speakers).

There are three ways to do this:

1. from the user interface:
   - Boot (power on) → click *0.00%* to the right of *Server* → select *Boot Server*. The numbers turn green, and some begin to change.
   - Quit (power off) → select *Quit Server*. The numbers turn blank and reset.
2. by writing and evaluating commands with lines of code:

```
s.boot; // Power on
s.quit; // Power off
```

3. using keyboard shortcuts:
   - Boot: MAC → cmd + b, WIN → ctrl + b.
   - Quit: Only from the user interface or code.

When we *Boot* the Server we turn on the audio computation in SuperCollider but if we don't send any signal to the output we won't hear any sound.

```
(
b = {PMOsc.ar(440, MouseY.kr(1, 550), MouseX.kr(1, 15))*Line.kr(0,0.3,1)};
b.play;
b.plot(0.5);
b.scope;
b.freqscope;
)
```

A.4.2 Stopping ongoing processes including audio

MAC → cmd + .
WIN → alt + .

When we run the code and stop it as we saw it is as if we pressed the *Play* and *Stop* buttons on a audio player or other sound generator connected to the electroacoustic system mentioned above.

A.4.3 Boot info

When we boot, the Server connects to the audio drivers currently being used by the computer's operating system.

If we want to use an external sound card → select it from the system preferences as with all other software.

When we boot the Server some useful information appears in the Post window:

- the name and IP address that identifies the server we're running. Using this address, we can send messages to that specific server and not to any others.
- the number and names of the devices (both real and virtual) that we have available to make the Server communicate with the outside world.
- the device (among those in the previous list) connected to the Server and how many audio channels it supports for both input and output signals.
- the sample rate and block size that the server in question is working at.
- some control messages indicating the status of communications between Server and Client.

A.4.4 Global variables, classes and messages

There are two default Servers → *local* and *internal*.

The first is the default Server.

The second is mainly for debugging (obsolete).

We can choose which one to use.

```
s = Server.local;
```

- *s* → variable.
- *Server* → Class.
- *.local* → message.

Variable

A portion of computer memory designated to hold data (numbers, characters, audio files, synthesizers, tables, arrays, etc.), which can be modified during the execution of a program.

To distinguish a single portion of memory from many others we must mark it with a *name* (or label, or address).

The line of code in the previous names a portion of memory *s* and assigns it to the local Server.

In SuperCollider three types of variables:

- global → only lowercase letters.
  - generally declared at the beginning of the file.
  - we can refer to them in any part of code.
- environment → a letter or word preceded by the tilde symbol (~).
  - the entire script is considered a programming environment.
  - they are valid within this entire environment (currentEnvironment).
  - they are we can use them as global variables with word names.
- local → preceded by the keyword *var* and included in bracket structures (round or curly).
  - its syntactic structure end with *;*
  - the assigned value is valid only within the parentheses (scope of the variable).

Class

A model through which we can construct many copies of an object (*instances*).

Classes begin with a capital letter.

Message

Messages can be sent to objects (classes, instances, numbers or any data type).

Messages have two functions:

- define the characteristics of an object.
- ask the object to perform an action.

We can send messages to objects separating them by a period.

In the previous code we:

- define the *Server* as local by sending the *.local* message
- assign a copy (instance) of it to the variable *s*.

This is one of the SuperCollider most common syntactic structures.

```
// variable_name   assignment    Object.messagge
   s                   =         Server.local;

   s.boot;     // send a new message to Server.local
   s.quit;
   s.reboot;
```

Through these syntactic structures the code is more readable and less redundant.

A.4.5 Server settings

Main messages.

```
s = Server.local;     // or .internal
s.boot;
s.quit;
s.reboot;             // quit and reboot
s.waitForBoot({...}) // boot the Server and value the function after booting
s.freeAll;            // clear the Server
s.volume = 0;         // set il volume in uscita (in dB)
s.mute;               // mute audio output
s.unmute;             // unmute
s.record;             // record outputs.
```

By default SuperCollider receives and sends all audio signals to the audio drivers selected in the operating system.

We can override this setting from code using *ServerOptions* class.

Get a list of devices connected to the computer.

```
ServerOptions.devices;
ServerOptions.inDevices;
ServerOptions.outDevices;
```

Select the one to use.

```
s = Server.local;

s.options.device;                        // which audio dirver is selected
s.options.device_("Soundflower␣(2ch)"); // set audio driver (in and out)
s.reboot;                                // if booted we must quit and reboot

s.options.device = nil;                  // 'nil' = system audio driver
s.reboot;
```

Different devices for input and output signals.

```
s.options.inDevice  = "Built-in␣Microph";
s.options.outDevice = "Soundflower␣(2ch)";
s.reboot;
```

Change the number of channels.

```
s.options.numOutputBusChannels.postln; // get
s.options.numOutputBusChannels_(8);    // set

s.options.numInputBusChannels.postln;  // get
s.options.numInputBusChannels_(8);     // set

s.reboot;
```

Change sampling rate and block size.

Change the number of channels.

```
s.options.sampleRate;
s.options.sampleRate_(44100);

s.options.blockSize;
s.options.blockSize_(64);

s.reboot;
```

A.5 UGens

UGens (Unit Generators) are special classes optimized for generating or modifying audio signals.

UGens *live* in the Server.

They can only receive three messsages that define their rate (how many values they must generate or modify over time).

- UGen.ar (audio rate) → generates or modifies as many values as the sample rate.
- UGen.kr (control rate) → generates or modifies one sample value for every sixty-four sample values made by an audio rate UGen.
- UGen.ir (initialization rate) → generates or modifies only one value at the time it is created.

See the difference.

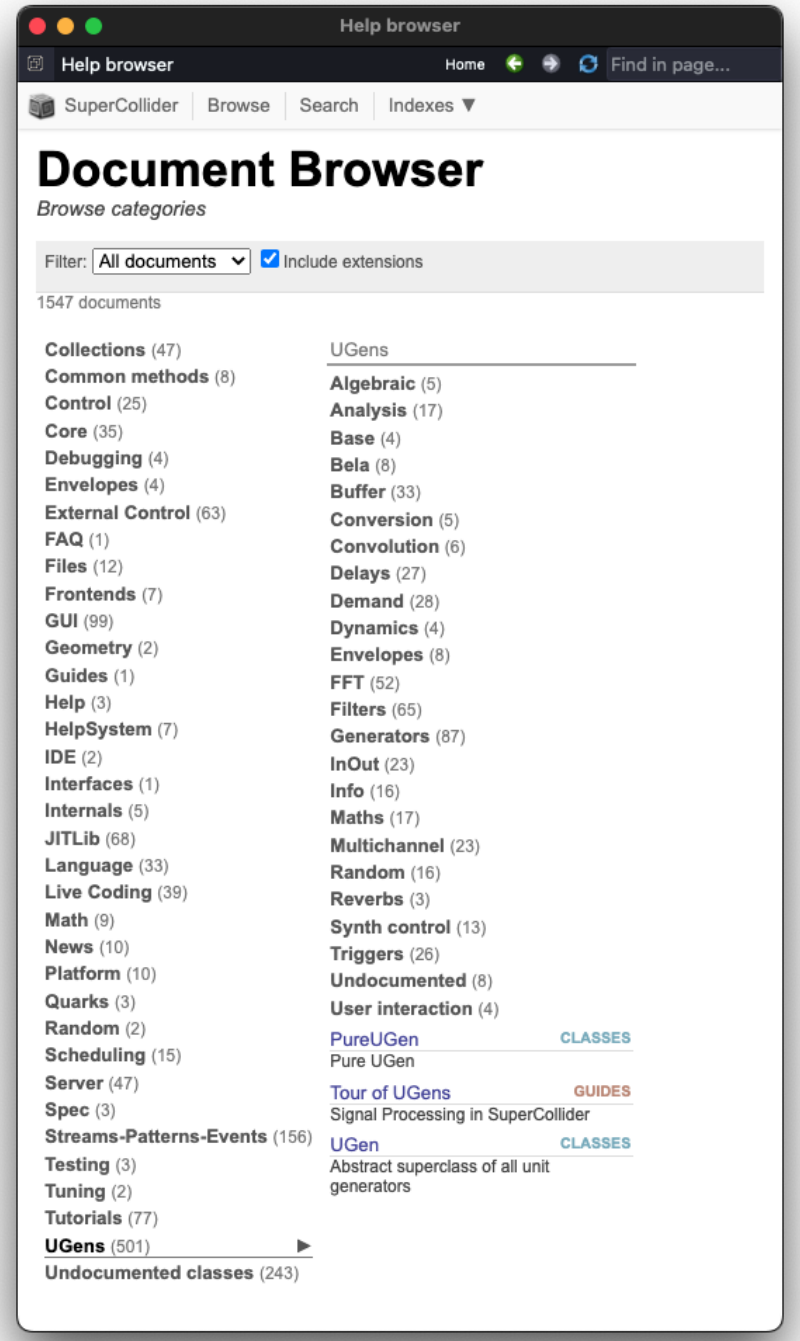
```
{[SinOsc.ar,SinOsc.kr]}.plot(1/50, bounds:1000@500);
```

Audio rate signals → can be connected to both public inputs (ADC) or public outputs (DAC) and used as parameter control signals of other UGens

Control rate signals → can be connected only to private inputs and outputs for internal routing.

While the previous messages define how many values are generated over time the UGen specification name indicate what kind of relationship exist between the generated values (*waveform*).

We can get a list of all UGens by going to the Help files window and clicking first on Browse and then on UGens.



All messages that can be invoked on UGens have *arguments* that specify the values of certain parameters of the signal to be generated or modified such as frequency, amplitude or other.

Arguments are specified within round brackets and are different for each message.

To find out the arguments of a message to a specific UGen we need to read its Help file.

For example, let's look at *SinOsc* class which is a sinusoidal oscillator.

Class Methods

`SinOsc.ar(freq: 440.0, phase: 0.0, mul: 1.0, add: 0.0)`

`SinOsc.kr(freq: 440.0, phase: 0.0, mul: 1.0, add: 0.0)`

Arguments:

freq Frequency in Hertz. Sampled at audio-rate.

phase Phase in radians. Sampled at audio-rate.

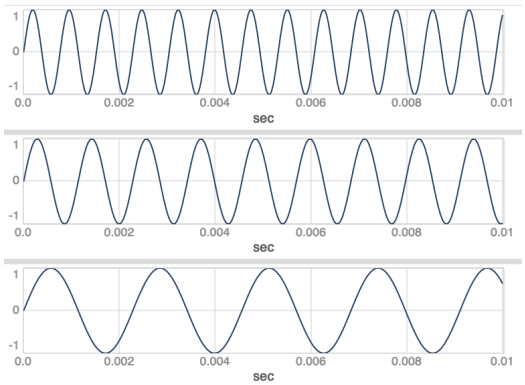
NOTE: phase values should be within the range +-8pi. If your phase values are larger then simply use `.mod(2pi)` to wrap them.

mul Output will be multiplied by this value.

add This value will be added to the output.

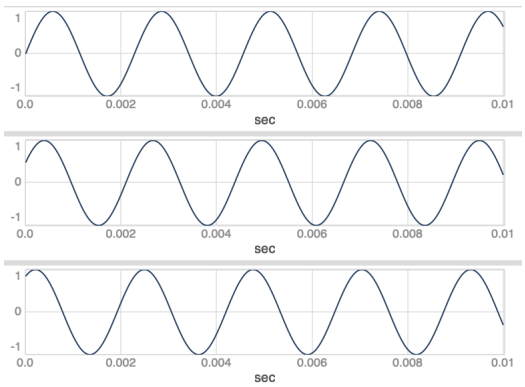
The first specifies the frequency in Hertz.

```
{
[SinOsc.ar(1320),
 SinOsc.ar(880),
 SinOsc.ar(440)]
}.plot(minval:-1,maxval:1)
```



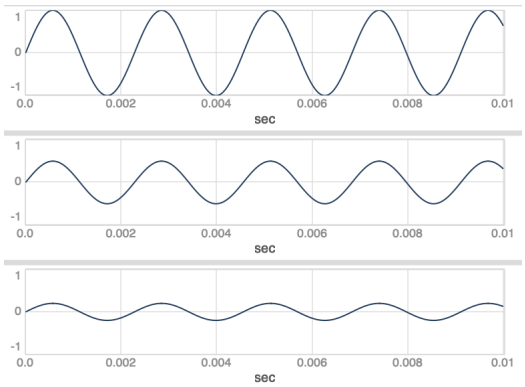
The second specifies the initial phase in radians between -1 and 1.

```
{
[SinOsc.ar(440, 0.0),
 SinOsc.ar(440, 0.5),
 SinOsc.ar(440, 1.0)]
}.plot(minval:-1,maxval:1)
```



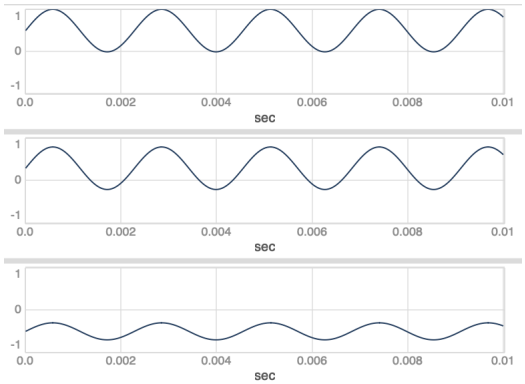
The third specifies a multiplication factor for the output values.

```
{
[SinOsc.ar(440, 0.0, 1.0),
 SinOsc.ar(440, 0.0, 0.5),
 SinOsc.ar(440, 0.0, 0.2)]
}.plot(minval:-1,maxval:1)
```



The fourth specifies a value to add to the output values.

```
{
[SinOsc.ar(440, 0.0, 0.5, 0.5),
 SinOsc.ar(440, 0.0, 0.5, 0.3),
 SinOsc.ar(440, 0.0, 0.2,-0.5)]
}.plot(minval:-1,maxval:1)
```



These last two arguments (*mul* and *add*) are included in almost all UGens and are commonly used to rescale the signal output range.

It is not necessary to specify all arguments.

Any arguments not specified will take the default value found in the help file.

Two different syntaxes.

```
SinOsc.ar(440,0,0.3);          // we must specify args from left to right
SinOsc.ar(mul:0.7, freq:500); // keyword
```

A.6 Monitors

Three types:

- auditory → we get information about the code by listening to the sound output.

- graphic → we display the waveform (oscillogram), spectrum (spectrogram) or intensity (meters) od the signal.
- numeric → we display the values of individual samples or rather samples extracted from the numerical stream at defined times (subsampling).

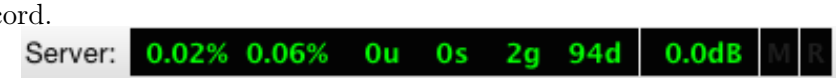
In SuperCollider we can monitor:

- computational activity.
- audio and control signals.
  - Server input and output public buses.
  - UGens input and output buses.

A.6.1 Computational activity

Two numeric monitor types:

1. read the information on the IDE (bottom right).
   - 0.02%: Average CPU usage.
   - 0.06%: Peak CPU usage.
   - 0u: number of active UGens in the Server.
   - 0s: Number of active Synths in the Server.
   - 2g: number of active Groups in the server.
   - 94d: number of active SynthDefs in the Server.
   - 0.0dB: Server Master Gain.
   - M: mute.
   - R: record.

   By clicking on the numbers a window appears and we can activate or deactivate various commands.
2. get values sending messages to the Server.

```
s.status;        // actual Server
s.avgCPU;        // CPU average
s.peakCPU;       // CPU peak
s.numUGens;      // number of UGens
s.numSynths;     // number of Synths
s.numGroups;     // number of Groups
s.numSynthDefs;  // number of SynthDefs
s.volume;        // output gain (dB)
s.mute;
s.unmute;
s.record;

```

   By getting the values from the code we can assign them to variables and use them within the script.

A.6.2 Audio and control signals

Server inputs and outputs

Three graphical interfaces:

```
s.meter;     // (PPM) monitoring signals amplitude
s.scope;     // (oscilloscope) monitoring waveforms
s.freqscope; // (spectroscope) monitoring spectra
```

UGens inputs and outputs

- auditory → we can quickly hear the output signals from a UGen with the following syntax.

```
{SinOsc.ar}.play;
{SinOsc.ar + WhiteNoise.ar}.play;
```

- graphic → we can change the *.play* message with *.scope* message.

```
{Saw.ar}.scope;
{SinOsc.ar(LFNoise0.kr(15,1000,2000))}.scope;
```

  We can change the *.play* message with *.plot* message.
  A lot of arguments, read the Help file.

```
{Pulse.ar}.plot;
{Pulse.ar + Saw.ar}.plot(duration: 0.01 ); // window duration in seconds
{Pulse.ar + WhiteNoise.kr}.plot(minval: -2, maxval: 2); // y axis
```

- numeric → we can use the *.poll* message directly on UGen.
  Argument is how many values show in one second (Hz).

```
SinOsc.ar.poll(1)}.play;
{SinOsc.ar(MouseX.kr(40,10000,1).poll(10), 0, 0.1) }.play;
```

A.7 Help files

Mainly divided into three sections:

1. contains general information about the object and suggestions on similar objects.
2. contains information about the messages (methods) we can invoke on the object and their arguments. They are divided into class methods and instance methods.
3. contains examples of using the UGen that can be evaluated directly on site.

A.8 Web resources

- B.Ruviaro - A Gentle Introduction to SuperCollider.
- E.Fieldsteel - You Tube Tutorials.
- VV.AA. - sc-code.
- VV.AA. - sc-tweets.
- T.Magnusson - Scoring aound

Appendix B

Components, connections and parameters

In this appendix we will delve deeper into the components, connections and parameters used in section *The virtual instrument paradigm* [2.4] in designing a generic virtual instrument model.

B.1 Source

Audio rate signal in range +/- 1.

Two types:

- oscillators.
- input signals.

B.1.1 Oscillators

Generate audio signals by reading sample values (lookup) from wavetables or buffers.

Lookup

Buffers and Arrays are indexed collections and do not have a time dimension.

Sound waves are small changes in atmospheric pressure over time.

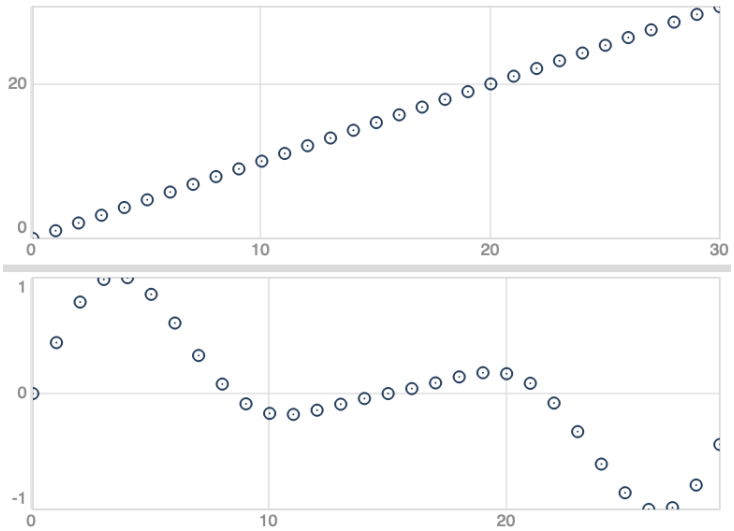
To dynamically reproduce these variations we can obtain the individual y values stored in a Buffer (instantaneous amplitudes that correspond to the measurement of the atmospheric pressure at that moment) by recalling the underlying indices (x values).

To create a playback we can use a phasor (or counter) that generates values at a sampling rate frequency ranging from 0 to the size - 1 of the Buffer to be read.

These values are the indexes used to recall the values of the stored samples.

Graphic representation.

```
a = (0..30);
b = Signal.sineFill(30,[0.4,0.6,0.2]);
[a,b].plot(discrete:true);
```



Audio representation.

```
s.boot;
s.scope;

b.free;
b = Buffer.read(s, ExampleFiles.apollo11);

SynthDef(\lookup, {arg buf=0,amp=0;
                    var pha, sig;
                        pha = Phasor.ar(0,MouseX.kr(0,10),0,BufFrames.kr(buf));
                        pha.linlin(0,BufFrames.kr(buf),0,1).scope;
                        sig = BufRd.ar(1,buf,pha);
                        sig = sig * amp;
                    Out.ar(0, sig)
                    }).add;

a = Synth(\lookup, [\buf,b, \amp, 0.2]);
a.free;
```

Table lookup is a fundamental process in music computing.

It underpins numerous procedures ranging from sequencing (client side) to various sound synthesis and processing techniques.

This process is used by all three main types of oscillators:

- interpolating wavetable oscillator.
- sample playback oscillator.
- buffer reading oscillator.

Interpolating wavetable oscillator

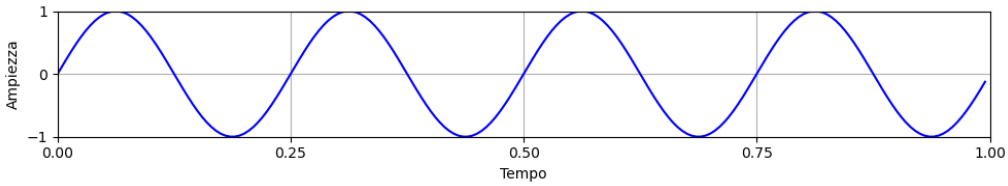
Read wavetables that contain a single cycle of a waveform in a cyclic way.

```
b.free;
b = Buffer.alloc(s, 512, 1);
b.sine1(1.0/[1,2,3,4,5,6], true, true, true);
b.plot;

SynthDef(\woscil, {arg buf=0, amp=0;
                    var sig;
                        sig = Osc.ar(buf, MouseX.kr(200,800));
                        sig = sig * amp;
```

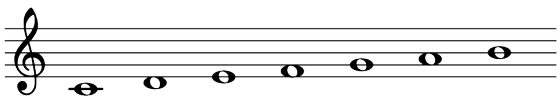
```
                    Out.ar(0, sig)
                    }).add;

a = Synth(\woscil, [\buf,b, \amp, 0.2]);
a.free;
```

Main parameters are:

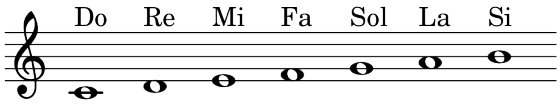
- buffer to read.
- frequency (or pitch).
  Frequency can only be recognized for periodic waves.
  A periodic wave is a vibrational phenomenon that repeats itself identically at equal intervals of time.
  The single interval measured in time is called the period (or cycle) of the wave and represented by the letter T.
  We can graphically represent a wave through a cartesian reference system:
  - x-axis (x) → time.
  - y-axis (y) → atmospheric pressure variations.



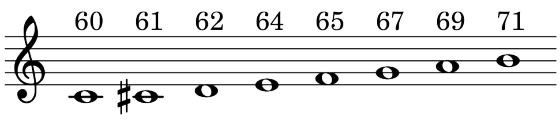
  In science the frequency of a periodic sound is measured in Hertz or cps (cycles per second) which is how many times the period repeats itself in one second.
  The human ear in its optimal condition perceives sounds between 20 and 20,000 Hz.
  We can specify this parameter in three different types of symbolic notation:
  1. Musical symbols → determine pitch based on graphic position in a coded language context (musical notation).



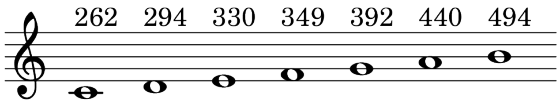
  2. Alphabetic symbols → determine pitch as the names of a scale or mode.



  3. Numeric symbols → in this case, as far as the contents of this paper are concerned, we can make a further subdivision into two different units:
     - MIDI notes → determine the pitch as numerical values (60 = Middle C).



     - frequencies (Hertz) → determine pitch as numerical values representing the number of cycles per second of a periodic wave (A4 = 440 Hz - diapason).

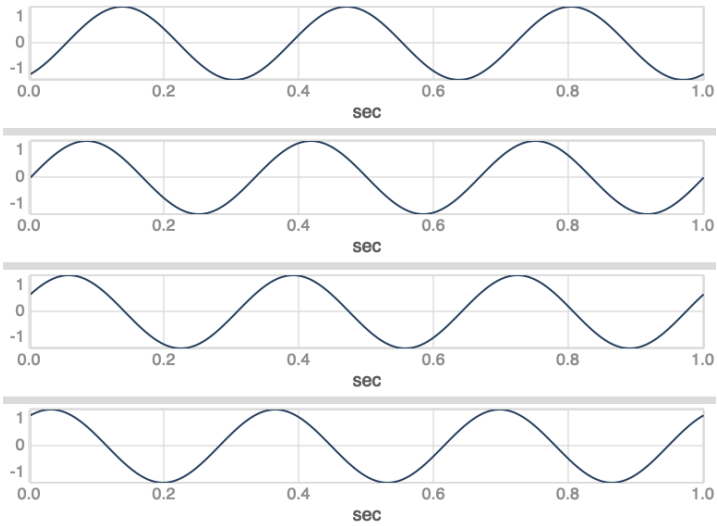


Useful conversions

```
440.cpsmidi; // Hz --> midi
440.cpsname;
60.midicps;  // midi --> Hz
60.midiname;
```

- phase → point in the period where the oscillation begins.

```
{[SinOsc.ar(3,-1),
  SinOsc.ar(3, 0),
  SinOsc.ar(3, 0.5),
  SinOsc.ar(3, 1),
]}.plot(1);
```



Sample playback oscillator

Read buffers that:

- contain a loaded soundfile.
- are continuously written by a recorder UGen.

```
b.free;
b = Buffer.read(s, ExampleFiles.apollo11);
b.plot;

SynthDef(\plbk, {arg buf=0, start=0, rate=1, amp=0;
                  var sig;
                      sig = PlayBuf.ar(1, buf,
                                       rate,
                                       startPos: start, // samples
                                       doneAction:2);
                      sig = sig * amp;
                  Out.ar(0, sig)
                  }).add;

a = Synth(\plbk, [\buf,b, \rate, 1, \start, 50000, \amp, 0.2]);
```

Main parameters are:

- buffer to read.
- rate (phasor incremental step value).
  Affects pitch and duration.
  - 1.0 → normal speed, 1/1 duration, actual pitch,
  - 2.0 → double speed, 1/2 duration, 1 octave up,
  - 0.5 → half speed, 2/1 duration, 1 octave low,
  - ...

  Negative values → retrograde direction.
- start position → (sample index).

Buffer reading oscillator

Read buffers that:

- contain a loaded soundfile.
- are continuously written by a recorder UGen.

It differs from the previous one in that we use a continuous audio rate control signal (pointer) of any type instead of the phasor.

```
b.free;
b = Buffer.read(s, ExampleFiles.apollo11);
b.plot;

SynthDef(\bread, {arg buf=0, amp=0, smth=0.2;
                    var sig;
                        sig = BufRd.ar(1, buf,
                                        K2A.ar(MouseX.kr(0, BufFrames.kr(buf),
                                                         0, smth)));
                        sig = sig * amp;
                    Out.ar(0, sig)
                    }).add;

a = Synth(\bread, [\buf,b, \amp, 0.2]);
a.set(\smth, 1);
a.free;
```

Main parameters are:

- buffer to read.
- control signal (pointer) parameters.
- smoothing time.

B.1.2 Input signals

Typically, a computer has:

- one input channel (microphone).
- two output channels (left and right speakers).

Softwares have audio buses that we can connect to these channels as we wish through audio buses.

One audio bus → one input or output mono channel.

In SuperColliders we have 128 audio channels (from 0 to 127) called buses.

By default the first three → *public buses* used for input and output signals from the outside.

The others → *private buses* availables for any internal software routing.

Two UGens for incoming channels:

- signals from public audio buses such as microphones or line inputs (ADC).

```
SynthDef(\mic, {arg busIn=0, amp=0;
                var sig;
                    sig = SoundIn.ar(busIn) * SinOsc.ar(400);
                    sig = FreeVerb.ar(sig);
                    sig = sig * amp;
                Out.ar(0, sig)
                }).add;

a = Synth(\mic, [\busIn,0, \amp, 0.5]);
a.free;
```

- signals from private audio buses (internal routing).

```
~bus = Bus.audio;

SynthDef(\source, {arg busOut=0;
                   var sig;
                       sig = Dust.ar(10);
                       sig = Decay.ar(sig);
                       sig = Resonz.ar(sig, 2000,1);
                   Out.ar(busOut, sig)
                   }).add;
SynthDef(\rev, {arg busIn=0, amp=0;
                var sig;
                    sig = In.ar(busIn);
                    sig = FreeVerb.ar(sig, 0.7, 0.8, 0.6);
                    sig = sig * amp;
                Out.ar(0, sig)
                }).add;

a = Synth(\rev,    [\busIn, ~bus, \amp, 1]);
b = Synth(\source, [\busOut, ~bus]);
b.set(\busOut, 0);
b.set(\busOut, ~bus);

a.free; b.free;
```

For both main parameters are:

- bus number.
- gain (amplitude)

B.2 Ampltude envelope

Float number or control/audio rate signal in range 0-1.

Two ways to measure the amplitude of a signal.

- peak amplitude → the highest absolute energy value among the instantaneous amplitudes (sample values from -1 to 1) within a finite time.
  Values between 0.0 and 1.0.
  If the highest value is negative, we remove the minus sign.
- root mean square (RMS) → a particular average of the energy values of the instantaneous amplitudes within a finite time.
  Values between 0.0 and 1.0.
  - finite set of instantaneous amplitudes.

```
a = [0.0, 1.5, 1.0, 0.4, 0.6, 0.0, -0.4, -0.2, -1.0, -1.5, 0.0];
```

– squared.

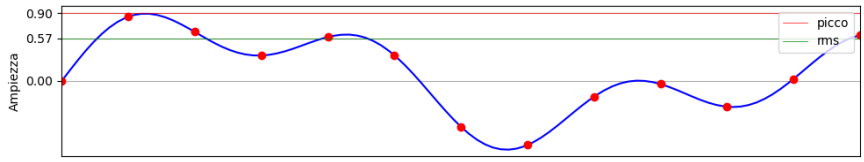
```
a = a.pow(2);
```

– mathematical mean.

```
a = a.mean;
```

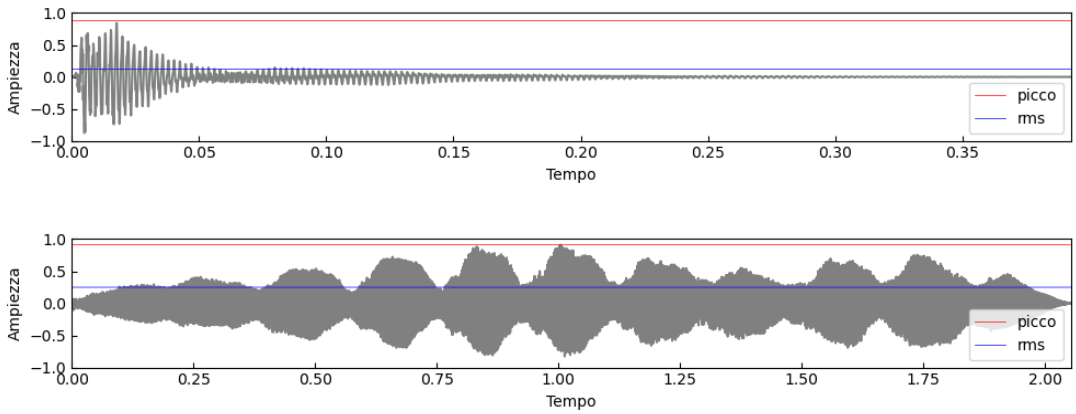
– square root.

```
a = a.sqrt;
```



- red points → instantaneous amplitudes (sample values).
- red line → peak amplitude.
- green line → rms amplitude.

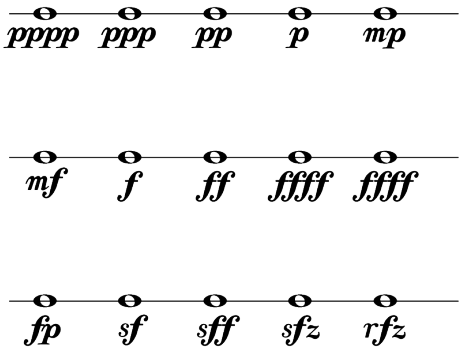
The main difference between peak and RMS amplitude is that the former is independent of the type of amplitude envelope of the signal while the latter, being an average, is strictly linked to it.



B.2.1 Representations and units of measurement

This parameter is also commonly called volume or amplification factor and in music we can represent it in two ways.

1. musical symbols → values are relative to the musical context.
   For example, a *forte* in a baroque violin sonata does not have the same intensity as a *forte* given to the brass instruments in a symphonic poem by R. Strauss.
   Unlike pitches, where there is a precise correspondence between the symbolic representation in musical notation and the physical measurement, musical dynamics are characterized by a significant subjectivity that goes far beyond the simple physical measurement of the amplitudes of sounds.



2. numerical values relating to different units of measurement → all refer to the peak amplitude.
   - linear → an absolute unit of measurement, the furthest from human perception.
     A sound with an amplitude of 0.5 will always be half the *loudest possible* sound and will always have the same intensity regardless of the musical context.
   - key velocity (MIDI) → an absolute unit of measurement expressed as integer values between 0 and 127.
     The least accurate due to the limited number of levels available.
     A sound with a key velocity of 127 will always be *as loud as possible* and will always have the same intensity regardless of the musical context.
   - quartic → absolute unit of measurement, closest to human perception.
     To calculate the correct values we must raise the linear amplitude to the fourth power.

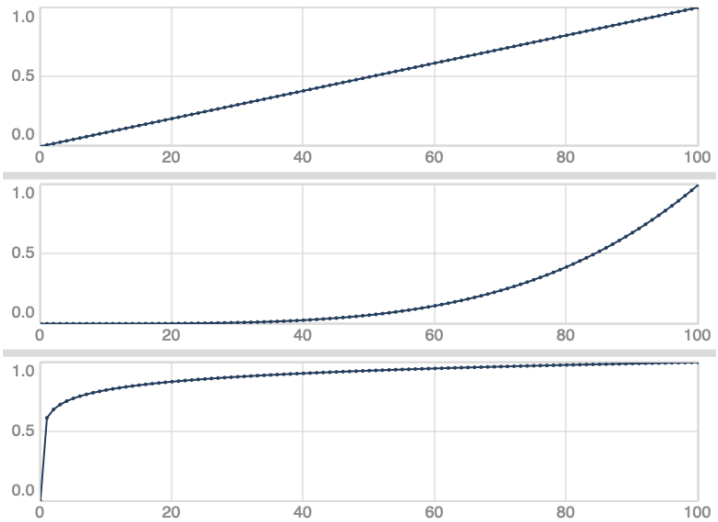
     $$A^4$$

     Since they range from 0.0 to 1.0, the range remains the same.
   - decibels (dB) → a relative unit of measurement expressed in decimal values between 0.0 and *-inf*.
     It measures the difference in intensity between the amplitude of a sound compared to a reference amplitude.
     A sound with an amplitude of -6.02 dB will always be half as loud as a sound whose amplitude was used as the reference for measurement.
     To calculate the correct values, we must use the following formula where:
     - A → linear amplitude value of the sound we want to measure.
     - A_0 → amplitude of the reference sound (1.0).

     $$A_{\text{dB}} = 20 \log_{10} \left(\frac{A}{A_0} \right)$$

Conversions.

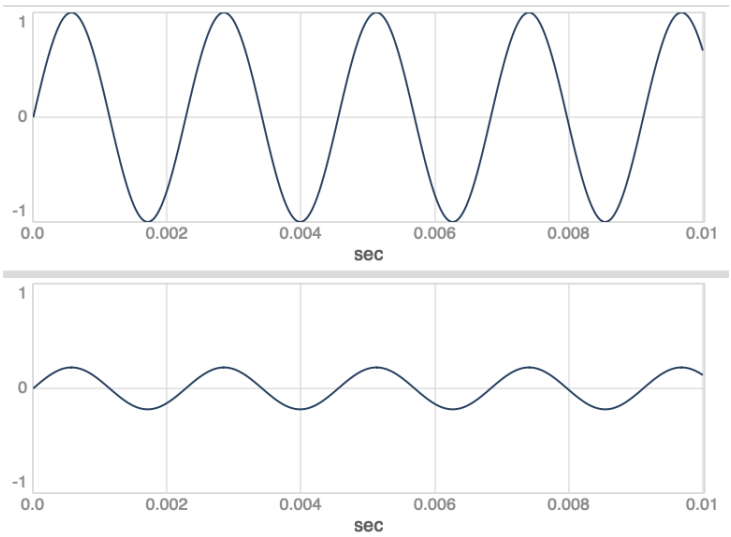
```
a = 0.5;
v = a * 127;      // lin --> velocity
a = v  / 127;     // velocity --> lin
q = a**4;         // lin --> quartic
a = pow(q, 1/4); // quartic --> lin
d = a.ampdb;      // lin --> db
a = d.dbamp;      // db --> lin
```



B.2.2 Scaling

We can rescale output signal from UGens by multiplying the sample values (+/-1.0) by a constant value between 0.0 and 1.0.

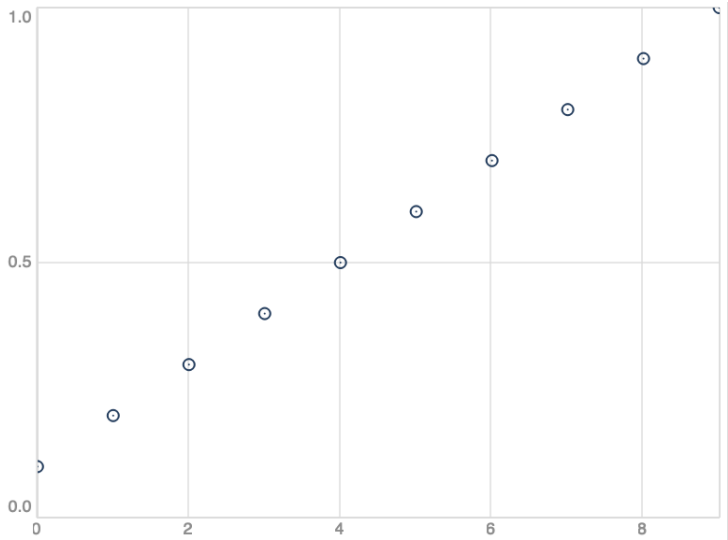
```
{[SinOsc.ar, SinOsc.ar * 0.2]}.plot
```



B.2.3 Interpolations and ramps

Interpolation → go from a to b in n *steps*.

```
a = 0.1; // start
b = 1.0; // end
n = 10;  // steps

i = Array.interpolation(n, a, b);
i.postln;
i.plot.plotMode_(\points);
```



In the previous sentence we can replace *a* with the last *value reached* (state) → from the *value reached* go to *b* in *n steps*.

```
a = 0.1;        // last value init
(
b = rand(50);   // end
n = 6;          // steps

i = Array.interpolation(n, a, b);
i.postln;
i.plot(minval:0,maxval:50).plotMode_(\points);
a = b;          // replace last value
)
```

In both the increase in value at each step is constant → linear interpolations.

Ramp → go from *a* to *b* in this *time* (seconds or milliseconds).

Number of *x* steps → number of samples for the specified time at the sampling rate used.

Example: go from 0.0 to 1.0 in 1 second.

If sampling rate is 44.100 Hz the steps are 44.100 and the incremental time 1/44.100.

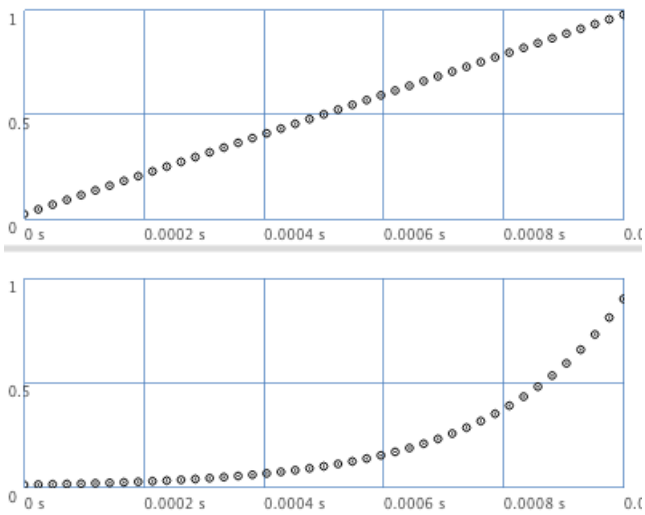
If it is 48.000 Hz the steps are 48.000 and the incremental value lower (1/48.000).

The ramp still go from 0.0 to 1.0 in one second.

It simply have more or less definition in amplitude changes.

Linear → constant incremental value.

Curved → incremental value change over time (steps).



In curved interpolations (or ramps):

- values between 0.0 and 1.0 → the curve changes but not the start and end values.
- other values → start and/or end are rescaled according to the exp or log base.
- exponential curve → positive values.
- logarithmic curve → negative values.

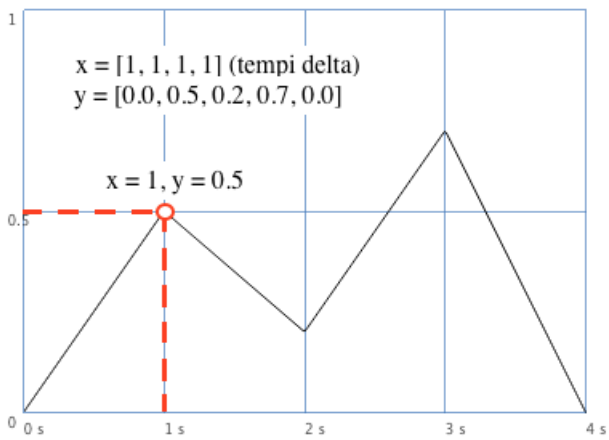
B.2.4 BPF and control signals

In the previous paragraph, we looked at single-segment ramps.

Amplitude envelopes are multi-segment ramps.

We can define the different points on a Cartesian plane by specifying their coordinates (xy) where the x correspond to time and the y correspond to levels.

This type of description and visualization is called BPM (Break Point Function) and almost all music software has a graphical interface that allows us to set and visualize all the multi-ramp segments we want.



In SuperCollider the *Env* class.

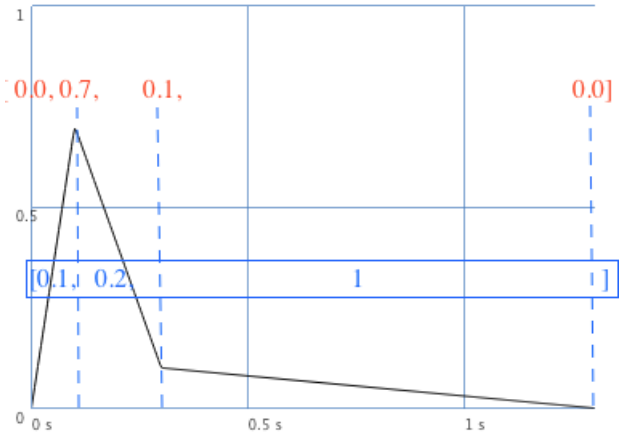
Env and EnvGen class

Env class allows us to define any type of envelope we want in the form of a BPF.

The *.new* method (which remember can be omitted) creates a new envelope (instance).

Its first two arguments are:

- levels → y values (red in the figure) from 0.0 to 1.0 in the form of an array.
- delta times → x values (blue in the figure) in seconds in the form of an array with one fewer element than the levels.



The *.test* method → auditory monitor.

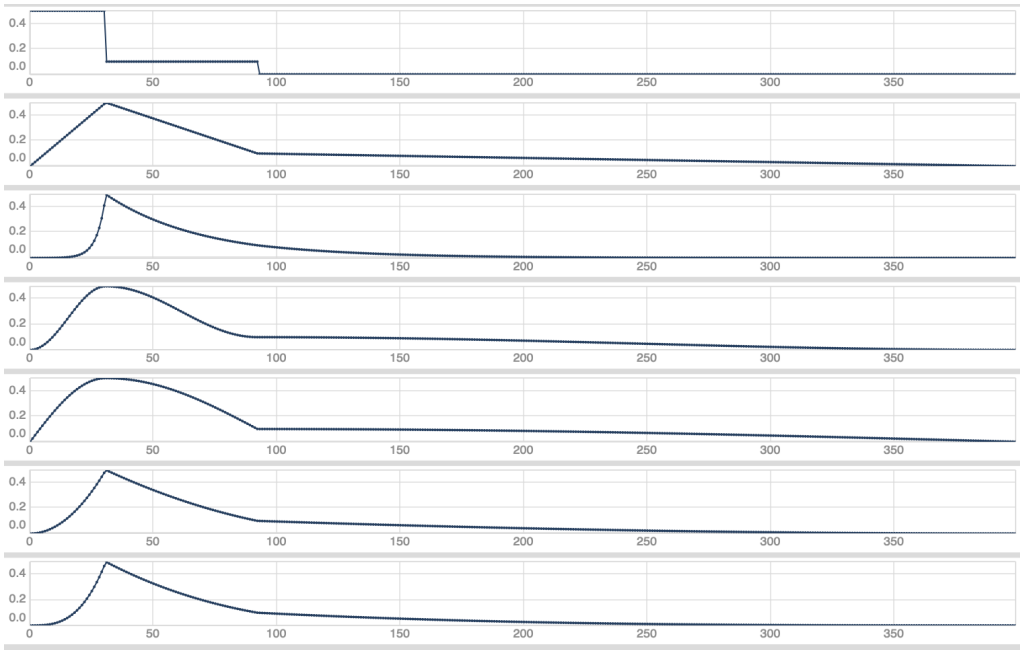
The *.plot* method → visual monitor.

```
s.boot;

Env.new([0, 0.7, 0.1, 0], // levels
        [ 0.1, 0.2,  1 ]  // delta times
       ).test.plot(minval: 0, maxval: 1);
```

We can also specify a third argument that specifies the curve in two ways.

- using a symbol.

```
[Env([0,0.5,0.1,0],           [0.1,0.2,1], \step),
 Env([0,0.5,0.1,0],           [0.1,0.2,1], \lin),
 Env([0.0001,0.5,0.1,0.0001],[0.1,0.2,1], \exp),
 Env([0,0.5,0.1,0],           [0.1,0.2,1], \sin),
 Env([0,0.5,0.1,0],           [0.1,0.2,1], \wel),
 Env([0,0.5,0.1,0],           [0.1,0.2,1], \sqr),
 Env([0,0.5,0.1,0],           [0.1,0.2,1], \cub)].plot;
```



- through the curvature value.

```
[Env([0,0.5,0.1,0],[0.1,0.2,1], -0.8),       // logarithmic_n
 Env([0,0.5,0.1,0],[0.1,0.2,1],  0.0),       // linear
 Env([0,0.5,0.1,0],[0.1,0.2,1],  0.8)].plot // exponential_n
 Env([0,0.5,0.1,0], [0.1,0.2,1], [   2,   0,  -8]), // different for each
    ramp
 Env([0,0.5,0.1,0], [0.1,0.2,1], [\lin,\exp,\sqr])
```

A Break Point Function describes points on a Cartesian plane not ramps.

We must generate the intermediate values between the points through interpolations that transform the BPF into *control signals*.

We can do it with *EnvGen* UGen.

```
{EnvGen.kr(                                          // also .ar
           Env.new([0,0.5,0.1,0],[0.01,0.2,1],\cub), // BPF
           1,                // gate (1 = noteon, 0 = noteoff)
           doneAction:2)     // doneAction
}.scope;
```

One of the arguments common to some UGens is *doneAction:n* which is extremely useful for dynamic voice allocation [2.4.2] and for saving CPU resources.

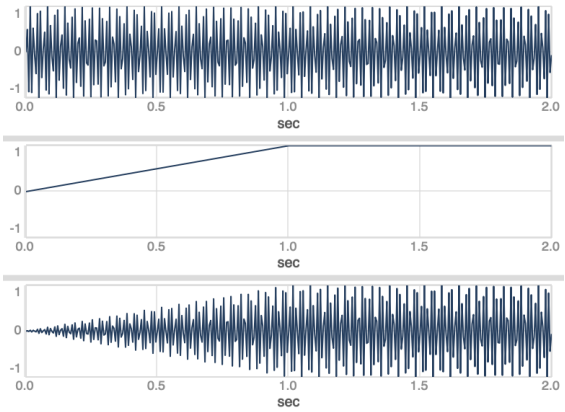
We can assign this argument a value between 0 and 14 (read *Done* class help file for more).

Each of these represents an automatic action that affects the *Synth* instance containing the UGen in which this argument is specified when it has completed its task (in the case of envelopes at the end of the last ramp).

The most used are:

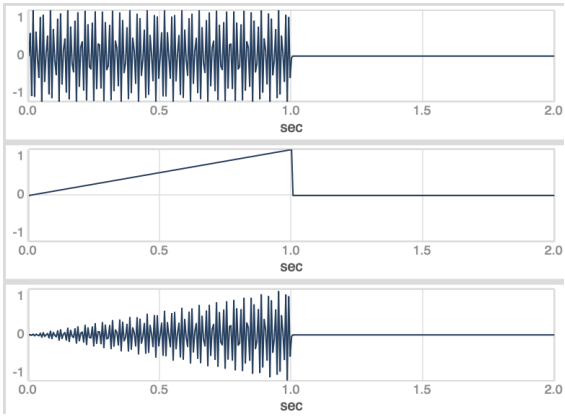
- *doneAction: 0.*

```
{SinOsc.ar * Line.ar(0, 1, 1, doneAction:0).scope}.scope
```



- *doneAction: 2.*

```
{SinOsc.ar * Line.ar(0, 1, 1, doneAction:2).scope}.scope
```



B.2.5 Triggers, envelope types and durations

A *trigger* is the command that starts any action or event such as sending the new frequency of a note, starting a sequence of notes or a sound, sending control parameters to a synth or starting an envelope.

Two trigger types:

- single trig (bang) → do it!
- switch → two states: 1/on, 0/off.

Let's think about the different interactions with doorbells at home

Two envelope types:

- without sustain as in percussions or plucked intruments.
- with sustain as in organ or electronic keyboards.

Both types require two triggers (gate 1 - noteon, gate 0 - noteoff).

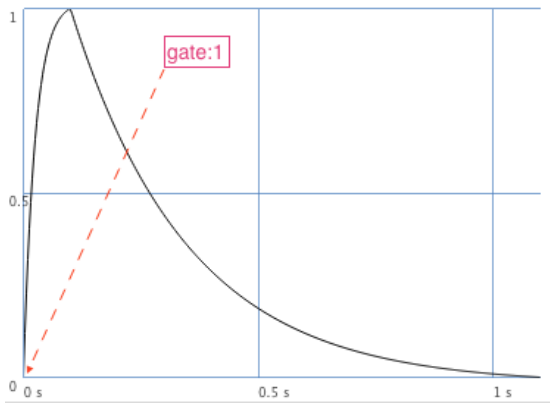
Trigger types and envelope types are closely connected to each other and to the different types of interaction with the musical instrument we want to design.

Without sustain

Strike the instrument or pluck the string → the sound starts and after a duration end.

In this type we can send a *gate* message preceded by *t_* .

This produce an automatic *gate 0* message after a time calculated on the specified duration.




```
SynthDef(\nosust,  {arg freq=500, amp=0, pan=0, pantime(0.0),
                        out=0, t_gate=0, done=2;
                    var sig, env;
                        sig = SinOsc.ar(freq);
                        env = Env.new([0, 1, 0.6, 0.8, 0],
                                      [0.1, 0.2, 0.4, 1],
                                      -4);
                        env = EnvGen.kr(env, t_gate, doneAction:done);
                        sig = sig * env * amp;
                        sig = Pan2.ar(sig, 0);
                    Out.ar(0, sig)
                    }).add;
a = Synth(\nosust, [\done, 0]);         // create the Synth
a.set(\amp, 0.5, \t_gate,1);            // trig
a.set(\amp, 0.5, \done, 2, \t_gate,1); // trig and kill
```

Listing B.1: Instrument model without sustain

In the previous example, the total duration of the sound is specified in absolute values (seconds).

We can calculate it by adding together the delta values of the individual ramps.

```
[0.1, 0.2, 0.4, 1].sum;
```

Musically it is complicated to calculate the duration of a sound through this sum.

To work around this, we can:

- define the ratio between the delta times in a relative/proportional manner.
- rescale all values so that the sum is 1.0 by invoking the *.normalizeSum* method.
- multiply the desired total duration by the resulting proportional values.

This way we can dynamically stretch the envelope model over time.

```
SynthDef(\nosust,  {arg freq=500, amp=0, dur=1, pan=0, pantime(0.0),
                        out=0, t_gate=0, done=2;
                    var sig, env;
                        sig = SinOsc.ar(freq);
                        env = Env.new([0, 1, 0.6, 0.8, 0],
                                      [10, 20, 40, 100].normalizeSum * dur,
                                      -4);
                        env = EnvGen.kr(env, t_gate, doneAction:done);
                        sig = sig * env * amp;
                        sig = Pan2.ar(sig, 0);
                    Out.ar(0, sig)
                    }).add;

a = Synth(\nosust, [\done, 0]);     // Create the Synth
a.set(\amp, 0.5, \t_gate,1);
```

```
a.free;
```

Listing B.2: Instrument model without sustain

The *Env* class also accepts some methods dedicated to common non sustained envelope shapes.

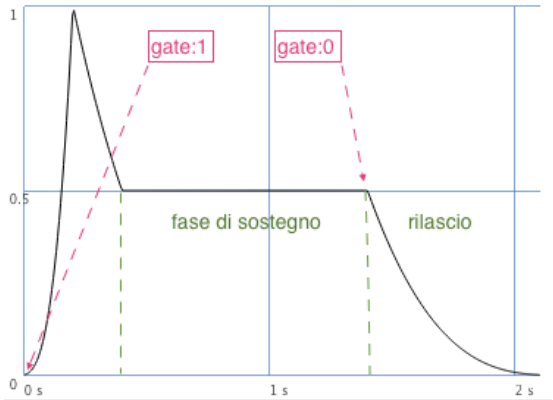
- *.linen* → trapezoidal envelope.
- *.perc* → percussive envelope.
- *.sine* → sinusoidal envelope.
- *.triangle* → triangular envelope.

Read the *Env* help file for more.

With sustain

Press a key → the sound starts and continues until we release the key.

In this type we must send a message of *gate:1* (noteon) and then textitgate:0 (noteoff).




```
SynthDef(\sust, {arg freq=500, amp=0, pan=0, pantime(0.0),
                     out=0, gate=0, done=2;
                 var sig, env;
                     sig = SinOsc.ar(freq);
                     env = Env.adsr;
                     env = EnvGen.kr(env, gate, doneAction:done);
                     sig = sig * env * amp;
                     sig = Pan2.ar(sig, 0);
                 Out.ar(0, sig)
                 }).add;

a = Synth(\sust, [\done, 0]);
a.set(\amp, 0.5, \gate, 1);
a.set(\gate, 0);
a.free;
```

Listing B.3: Instrument model with sustain

There isn't a global duration.

When the sound is triggered and reaches the sustain level it can theoretically go on forever until it receives the *gate 0* message.

We can only define the duration of the attack and release ramps in absolute values.

Some *Env* methods dedicated to common sustained envelope shapes.

- *.asr* → trapezoidal envelope.

- *.adsr* → classic envelope.
- *.circle* → loop envelope.

There is also a useful UGen that generates a fadein and fadeout: *Linen.kr*.

```
SynthDef(\linen, {arg freq=890, gate=0, a=0.1, r=0.6, amp=0, done=2;
                    var sig,env;
                        sig = SinOsc.ar(freq);
                        env = Linen.kr(gate, a, amp, r, done);
                        sig = sig * env;
                    Out.ar(0, sig)
                    }).add;

a = Synth(\linen, [\amp,0.5]);
a.set(\gate,1);
a.set(\gate,0);
```

Listing B.4: Instrument model with Linen.kr

B.3 Pan

Float number or control/audio rate signal in range +/- 1.

Almost all UGens output a monophonic signal (one channel).

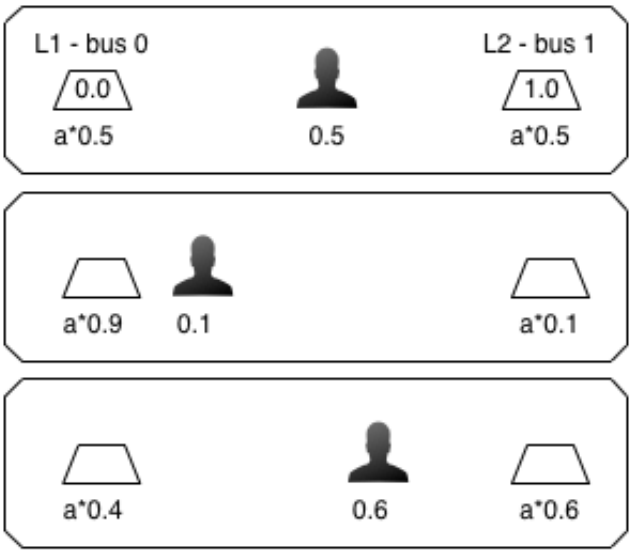
```
s.boot;
s.meter;

SynthDef(\mono, {arg freq=987, gate=0, amp=0, done=2;
                    var sig, env;
                        sig = SinOsc.ar(freq);
                        env = Linen.kr(gate, doneAction:done);
                        sig = sig * env * amp;
                    Out.ar(0, sig)
                    }).add;

a = Synth(\mono, [\amp,0.5]);
a.set(\gate,1);
a.set(\gate,0);
```

If we duplicate a monophonic signal by writing it on two adjacent channels and broadcast it through two loudspeakers, we obtain a dual-mono signal, not stereo.

```
SynthDef(\dual_mono, {arg freq=987, gate=0, amp=0, done=2;
                        var sig, env;
                            sig = SinOsc.ar(freq);
                            env = Linen.kr(gate, doneAction:done);
                            sig = sig * env * amp;
                        Out.ar(0, [sig, sig])
                        }).add;

a = Synth(\dual_mono, [\amp,0.5]);
a.set(\gate,1);
a.set(\gate,0);
```

To obtain a stereo signal we must take a monophonic signal and distribute its amplitude between two adjacent channels so that the sum of the amplitudes output from the two speakers is the same as the original monophonic signal and never exceeds 1.0.

The ratio between the amplitudes output from the speakers defines the position of the source in the stereo front.




```
SynthDef(\stereo, {arg pan=0.5;          // 0 = left, 0.5 = center, 1 = right
                    var sig, sx, dx;
                        sig = SinOsc.ar;
                        sx  = 1.0 - pan; // left loudspeaker amplitude
                        dx  = pan;       // right loudspeaker amplitude
                        Out.ar(0, [sx, dx] * sig)
                        }).add;

a = Synth(\stereo);
a.set(\pan,0);   // left
a.set(\pan,0.5); // center
a.set(\pan,1);   // right
```

We define a single value to control panning by automatically calculating the output signal amplitudes.

The perceived intensity of the sound at different positions is not the same.

At the center it is softer than at the sides.

Possible solution → calculate the square root of the amplitude multiplication factor.

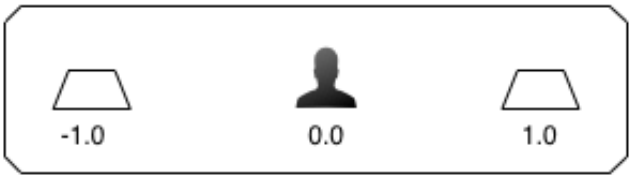
The sum of the signals when positioned at the center is greater than 1.0 ($\sqrt{0.5} = 0.707*2 = 1.141$) but the perceptual result is correct and there is not harmonic distorsion.

```
SynthDef(\stereo, {arg pan=0.5;          // 0 = left, 0.5 = center, 1 = right
                    var sig, sx, dx;
                        sig = SinOsc.ar;
                        sx  = (1.0 - pan).sqrt; // left loudspeaker amplitude
                        dx  = pan.sqrt;         // right loudspeaker amplitude
                        Out.ar(0, [sx, dx] * sig)
                        }).add;

a = Synth(\stereo);
a.set(\pan,0);   // left
a.set(\pan,0.5); // center
a.set(\pan,1);   // right
```

In SuperCollider two dedicated UGens.

- *LinPan2.ar* → linear panning with the softer sound at the center.
- *Pan2.ar* → with perceptual correction (equal power).

Both accept the following arguments: mono_signal, position (-1.0 = left, 0 = center, 1.0 = right), level.



In the previous examples we changed the position instantly.

If we want to create movements within the stereo field we can:

- convert values (float) into rounded control signals (lag time).

```
SynthDef(\pansmooth, {arg pos=0, ptime=0.5; // target position, lag time
                        var sig, pan;
                            sig = SinOsc.ar;
                            pan = pos.lag(ptime); // logarithmic
                            sig = Pan2.ar(sig, pan);
                        Out.ar(0, sig)
                        }).add;

a = Synth(\pan);
a.set(\pos, rand2(1.0).postln, \ptime, 2);
```

- use envelopes.

```
SynthDef(\muovi, {arg t_gate=0, dur=1, amp=0;
                    var sig, pos;
                        sig = SinOsc.ar;
                        pos = Env.new([-1, 1, 0, 0.5, -0.7, 0], // positions
                                      [  1, 0.2, 2,   0.3, 1 ].normalizeSum
                                                                  * dur,
                                      0);
                        pos = EnvGen.kr(pos, t_gate);
                        sig = Pan2.ar(sig, pos);
                        sig = sig * amp;
                    Out.ar(0, sig)
                    }).add;

a = Synth(\muovi);

a.set(\dur, 10, \amp, 0.5, \t_gate, 1);
```

- use LFO signals.

```
SynthDef(\lfo, {arg rate=0;
                 var sig, lfo;
                     sig = SinOsc.ar;
                     lfo = SinOsc.ar(rate/2);   // periodic bipolar signals
                      //lfo = LFNoise1.ar(rate); // Random
                      // other...
                     sig = Pan2.ar(sig, lfo);
                 Out.ar(0, sig)
                 }).add;

a = Synth(\lfo);

a.set(\rate, 2);
```

B.4 Output

Public or private buses.

UGen for output audio signals: *Out.ar*.

Arguments:

- bus number (0 → leftmost loudspeaker).
- signal or array of signals to write on bus.

If the signal in output is more than one channel → the first channel is written to the bus specified as the first argument while the others are sent to nearby buses in ascending order.

```
s.scope(3);

SynthDef(\out2,{arg t_gate=0, outBus=0;
                var siga, sigb, env;
                    siga = WhiteNoise.ar;
                    sigb = Saw.ar;
                    env  = Env.perc;
                    env  = EnvGen.kr(env,t_gate);
                    siga = siga * env * 0.2;
                    sigb = sigb * env * 0.2;
                Out.ar(outBus, [siga, sigb]) // Array
                }).add;

a = Synth(\out2);

a.set(\outBus,0, \t_gate,1);
a.set(\outBus,1, \t_gate,1);
```

If we change it dynamically → clicks!

We can have more than one *Out* UGen in a *SynthDef* for multiple signal routing.

Listings

2.1 Instrument model . . . 42
2.2 Dynamic voice allocation . . . 43
2.3 Static voice allocation . . . 43
3.1 Load one sound file in a Buffer . . . 52
3.2 Load many sounfiles from a folder . . . 53
3.3 Sampler model . . . 53
3.4 Deterministic sequencing . . . 54
3.5 Percussive grain model . . . 55
3.6 Dynamic envelope model . . . 55
3.7 Control signals model . . . 57
3.8 Scratcher model . . . 57
3.9 Recording sound . . . 58
3.10 Scoring . . . 60
4.1 coin(0.5) . . . 66
4.2 rrand(min max) . . . 66
4.3 rand(max) . . . 67
4.4 rand2(+/-max) . . . 67
4.5 choose(Array) . . . 67
4.6 Source signals model . . . 68
4.7 Wawtable lookup synthesis model . . . 69
4.8 Vetorial synthesis model . . . 70
4.9 Envelope mapping parameter . . . 71
4.10 Pulse Width Modulation model . . . 72
4.11 Blip model . . . 72
4.12 Timbre stretching model . . . 73
4.13 Partial model . . . 74
4.14 Filters models . . . 76
4.15 Resonators model . . . 78
4.16 Basic FM model . . . 79
4.17 Classic FM model . . . 79
4.18 Envelope control model . . . 80
4.19 Granular Synth sine envelope model . . . 81
4.20 Granular Synth custom envelope model . . . 82
4.21 Granular time stretching model . . . 83
4.22 Spectral Delay model . . . 85
4.23 Brick Wall spectral filter model . . . 85
4.24 Rect Comb filter spectral model . . . 86
4.25 Mag Smear spectral model . . . 86
4.26 Freeze spectral model . . . 87
4.27 Morphing spectral model . . . 87
5.1 Live midi out sender model . . . 97

5.2 random incise function . . . 100
5.3 random motive function . . . 101
5.4 chromatic transposition function . . . 101
5.5 diatonic transposition function . . . 102
5.6 retrograde function . . . 102
5.7 inversion function . . . 103
5.8 retrograde of inversion function . . . 103
5.9 palindrome function . . . 103
5.10 all permutations function . . . 104
5.11 random order function . . . 104
5.12 chord model function . . . 106
5.13 voicing model function . . . 107
5.14 drop model functions . . . 108
5.15 voice-leading model function . . . 109
5.16 voice-leading common notes model function . . . 109
5.17 chord random walk model function . . . 110
5.18 top note melody model function . . . 111
5.19 random note melody model function . . . 111
5.20 arpeggiating note melody model function . . . 111
5.21 voice-leading style melody model function . . . 111
5.22 voice-leading 2 style melody model function . . . 111
5.23 melodic-walk style model function . . . 112
6.1 MIDI monophonic instrument model . . . 125
6.2 MIDI poliphonic no-sustain instrument model . . . 126
6.3 MIDI poliphonic w-sustain instrument model . . . 127
6.4 mouseDownAction model . . . 130
6.5 mouseUpAction model . . . 131
6.6 mouseMoveAction model . . . 131
6.7 mouseOverAction model . . . 132
6.8 mouse controls Server model . . . 133
6.9 keyDownAction model . . . 134
6.10 keyUpAction model . . . 134
6.11 key control Server model . . . 136
6.12 playback one-shot model . . . 137
6.13 playback key-map-soundfile model . . . 138
6.14 playback key-playback-rate model . . . 138
6.15 playback key-start-position model . . . 139
6.16 playback cues model . . . 140
7.1 Server settings model . . . 145
7.2 Live electronics model . . . 146
7.3 Envelope follower models . . . 151
7.4 Pitch follower models . . . 153
7.5 Onset detection model . . . 155
7.6 Spectral features model . . . 156
7.7 Live recording model . . . 158
7.8 Live ring modulation model . . . 160
7.9 Live frequency shifter model . . . 161
7.10 Live looping machine model . . . 164
7.11 Live sync phasor and buffer recorder model . . . 167
7.12 Live granular harmonizer model . . . 168
7.13 Live echo model . . . 170

7.14 Live feedback gain delay model . . . 170
7.15 Live feedback duration delay model . . . 171
7.16 Live ping-pong delay model . . . 172
7.17 Live multi tap delay model . . . 172
7.18 Live variable time delay model . . . 173
7.19 Live filtred delay model . . . 174
7.20 Live resonator delay model . . . 174
7.21 Stereo pan model . . . 177
7.22 Stereo pan model . . . 177
7.23 Quadriphonic pan model . . . 178
7.24 Stereo pan model . . . 178
7.25 Circular pan model . . . 179
B.1 Instrument model without sustain . . . 211
B.2 Instrument model without sustain . . . 211
B.3 Instrument model with sustain . . . 212
B.4 Instrument model with Linen.kr . . . 213

Bibliography

[1] https://www.youtube.com/shorts/jq7CD9XM-ow

[2] https://www.youtube.com/watch?v=8-wLLhyVlLk&list=PLZkB3Z_XAGdpYsuhpNKfekeVIGlzoGEIO

[3] https://www.youtube.com/watch?v=QVoviGYThxw

[4] https://www.youtube.com/watch?v=e52IMaE-3As

[5] https://www.youtube.com/watch?v=QpEit_Lll0k

[6] https://www.youtube.com/watch?v=lwFGVv9h80o

[7] https://www.youtube.com/watch?v=qENwEuD__2E

[8] https://www.youtube.com/watch?v=a_B02BZp-5Y&t=923s

[9] https://www.youtube.com/watch?v=wmTHiCq8T3I&list=PLrt7VbxNS8reSl1mCcPnHIPB_e01so0of

[10] https://youtu.be/PZpH9Khn0E0?si=2RppXwy3-4cGLpqz&t=77

[11] https://www.youtube.com/watch?v=i1X3mz-k0oE

[12] https://www.youtube.com/watch?v=MHFuaaGpGKQ&t

[13] https://www.youtube.com/watch?v=G96FEkkFCzg

[14] https://www.youtube.com/watch?v=7eOaIiHB58U

[15] https://www.youtube.com/watch?v=asOFi1W228M

[16] https://www.youtube.com/watch?v=aL77mHnCrNs

[17] https://www.youtube.com/watch?v=c_JHjUFfOs8&t=12s

[18] https://www.youtube.com/watch?v=F20kvxuXuE0

[19] https://www.youtube.com/watch?v=QwvOxJ-4Saw

[20] https://www.youtube.com/watch?v=vNwYtllyt3Q

[21] https://www.youtube.com/watch?v=xf1JjUtnHs8

[22] https://www.youtube.com/watch?v=Xs0K4ApWl4g

[23] https://www.youtube.com/watch?v=988jPjs1gao

[24] https://www.youtube.com/watch?v=VstAnd5GA6w&t=2276s

[25] https://www.youtube.com/watch?v=oMgQJEcVToY&list=PLzjkiYUjXuevVG0fTOX4GCTzbU0ooHQ-0

[26] https://www.youtube.com/watch?v=chA-4GRCb-I&list=PLYlChgfa88gm_2m5EWUVulfytL7re0zpE

[27] https://www.youtube.com/watch?v=TMYDgwNALY8

[28] https://www.youtube.com/watch?v=Ix2b_qFYfAA&t=163s

[29] https://youtu.be/CIPTgie3Wjc?si=T6FxI8Lj6ynsKx7I

[30] https://www.youtube.com/watch?v=8ftsFsScFbM