Music and code

Andrea Vigani

August 29, 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.

Contents

1 Premises **9**
1.1 Preliminary reflections . . . 9
1.1.1 Concrete things . . . 9
1.1.2 Pre-linguistic . . . 10
Natural language . . . 11
Musical language . . . 11
1.1.3 Compose a phone number . . . 12
Phonetic-phonological level . . . 12
Morphosyntactic level . . . 13
Syntactic level . . . 14
Semantic level . . . 15
1.1.4 Perceive and process . . . 16
1.1.5 Re-compose a puzzle . . . 17
1.2 Music's meanings . . . 17
1.2.1 Signs and sounds . . . 18
1.2.2 Composers, players and listeners . . . 19
1.2.3 Do you know this music? . . . 20
1.2.4 Do you like this music? . . . 21
1.2.5 Texts and narratives . . . 22
1.3 Forms of human expression . . . 23
1.3.1 Poietic dimension . . . 24
1.3.2 Aesthesic dimension . . . 24
1.3.3 The work itself . . . 25
1.3.4 Final reflections . . . 26

2 Genres, instruments and tools **29**
2.1 Genres . . . 29
2.2 Non interactive systems . . . 29
2.2.1 Fixed media music . . . 30
Acousmatic music . . . 30
Wallpaper music, ambient music and soundscapes . . . 32
Radiodrama . . . 33
Computer music - algorithmic composition . . . 34
2.2.2 Live sequencing . . . 35
Sound design for performing arts or video . . . 35
Sound design for gaming . . . 36
2.3 Interactive systems . . . 36
2.3.1 Hyper-instrument . . . 37
2.3.2 Live set . . . 38
2.3.3 Live coding . . . 38

2.4 The virtual instrument paradigm . . . 39
2.4.1 Software installation . . . 39
2.4.2 Instrumental model . . . 39

3 Fixed media **43**
3.1 Historical framework . . . 43
3.2 Elements and strategies . . . 43
3.2.1 Sound objects . . . 44
3.2.2 Objects, structure and form . . . 44
3.2.3 Musical objects . . . 45
3.2.4 Time courses and musical forms . . . 47
3.2.5 The poietic dimension: determinism . . . 49
3.3 Sound processing techniques in SuperCollider . . . 50
3.3.1 Buffers . . . 50
3.3.2 Sampling . . . 51
Sequencing . . . 52
Enveloping . . . 53
3.3.3 Scratching . . . 54
3.3.4 Structures, DAW and scoring . . . 56
3.3.5 Composition sketches proposal . . . 58

4 Computer music **59**
4.1 Historical framework . . . 59
4.2 Randomness . . . 60
4.2.1 Chance . . . 61
4.2.2 Indeterminacy . . . 62
4.2.3 Alea . . . 64
4.3 Sound synthesis . . . 66
4.3.1 Additive . . . 67
Wavetables . . . 67
Vectorial . . . 68
PWM . . . 70
Classic . . . 71
4.3.2 Subtractive . . . 73
4.3.3 FM . . . 76
4.3.4 FFT . . . 78
Spectral Delay . . . 79
Brick Wall filter . . . 79
Rect Comb filter . . . 80
Mag Smear . . . 80
Freeze . . . 81
Cross synthesys. . . . 81
4.4 Algorithmic structures . . . 82
4.5 Composition sketches proposal . . . 84

5 Algorithmic composition **85**
5.1 Historical framework . . . 85
5.2 Software installation . . . 86
5.3 Symbolic musical representation and numbers . . . 87
5.3.1 Pitch . . . 87
5.3.2 Rhythm . . . 88
5.3.3 Dynamic . . . 89

5.3.4 Collections . . . 90
5.3.5 Live . . . 91
5.4 Generative techniques and processes . . . 91
5.4.1 Modes and scales . . . 91
Scale . . . 92
Incises and motives . . . 92
Variations . . . 95
5.4.2 Harmonic fields . . . 99
Chord database . . . 100
Voicing . . . 101
Voice-leading . . . 102
Melodies . . . 104
5.5 Composition sketches proposal . . . 106

6 From live set to laptop ensemble 107
6.1 Historical framework . . . 107
6.1.1 Experimental fields . . . 107
6.1.2 Historical framework . . . 107
6.2 Interaction types . . . 107
6.2.1 Trigger . . . 108
6.2.2 Switch . . . 108
6.2.3 Continuos . . . 109
6.2.4 Conversions . . . 109
Smoothing . . . 109
Sample and hold . . . 110
Thresholds . . . 111
6.3 Control signals . . . 112
6.3.1 Types . . . 112
6.3.2 Scaling . . . 113
6.3.3 Plugging . . . 113
6.4 MIDI . . . 114
6.4.1 Receive . . . 115
6.4.2 Send . . . 116
6.4.3 Voice allocation . . . 116
Monophonic synth . . . 116
Poliphonic synth without sustain envelope . . . 117
Poliphonic synth with sustain envelope . . . 117
6.5 OSC . . . 118
6.5.1 Receive . . . 119
6.5.2 Send . . . 119
6.6 HID . . . 120
6.6.1 Mouse . . . 120
Client side . . . 120
Server side . . . 123
6.6.2 Keyboard . . . 124
Client side . . . 124
Server side . . . 126
6.7 Live playback and synthesis . . . 127
6.7.1 One shot . . . 127
6.7.2 Mapping . . . 128
Key → soundfile . . . 128

Key → playback rate (transposition) . . . 129
Key → start position . . . 129
6.7.3 Cues . . . 130
6.8 Composition sketches proposal . . . 132

7 Hyper-instruments **133**
7.1 Historical framework . . . 133
7.2 Software model . . . 133
7.2.1 Bus and Groups . . . 133
7.3 Feature extraction as control signals . . . 133
7.3.1 Envelope follower . . . 133
7.3.2 Pitch follower . . . 133
7.3.3 Centroid . . . 133
7.4 Classic live electronics sound processing . . . 133
7.4.1 Ring modulation . . . 133
7.4.2 Live recording and playback . . . 133
7.4.3 Granular synthesis . . . 133
7.4.4 Delay lines . . . 133
7.5 Composition sketches proposal . . . 133

8 Live coding **135**
8.1 Historical framework . . . 135
8.2 Composition sketches proposal . . . 135

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.

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.

When it 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.

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.

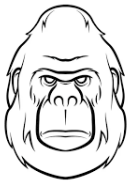
Sound waves have to do with physical world and human perception.

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 because 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 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 because it exists independently of language.

Musical language

About music there exist forms of mental activity independent of any musical language such as children's nursery rhymes or tibetan songs.

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 - individual 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 which are 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).

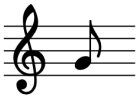
THE INTERNATIONAL PHONETIC ALPHABET (revised to 2018)

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.).



Let us remember that in both languages they are characterised by frequency, duration, amplitude and timbre.

Morphosyntactic level

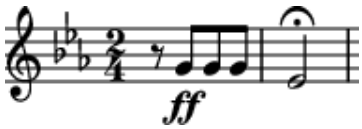
Phonemes can join together to form morphemes.

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

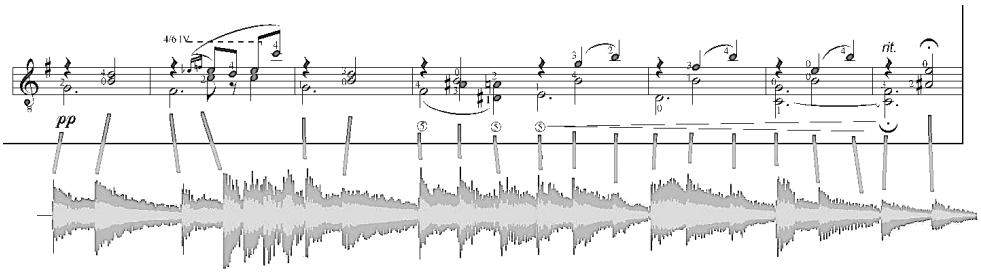
Syntax or formal grammar → a closed system of rules that serves exclusively to generate the set of sequences considered grammatical.

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?

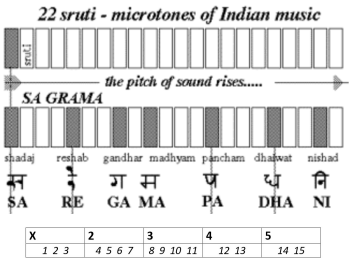
No, again.

In order to be defined music language requires a codification of sounds in vocabularies and systems of rules.

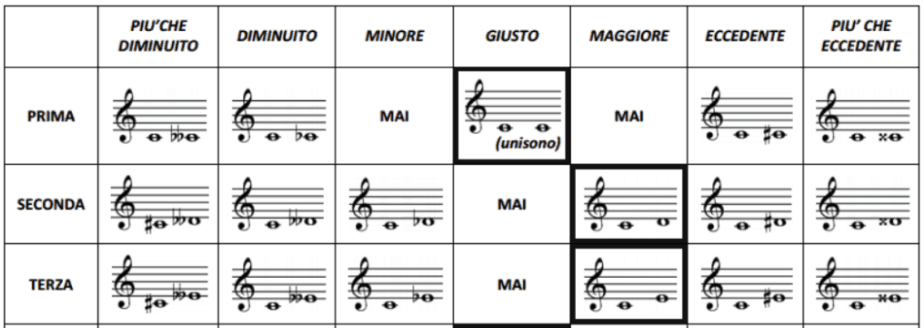


In all musical cultures the physical parameters of sound are organized in symbols.

These symbols can become independent from sound waves.



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.).



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

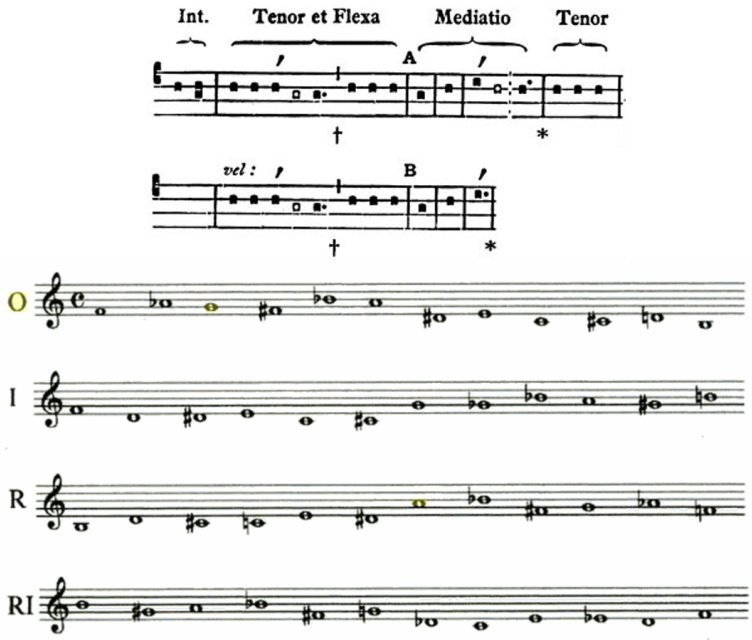


Fig. 1 – Serie della Sinfonia op. 21 di A. Webern

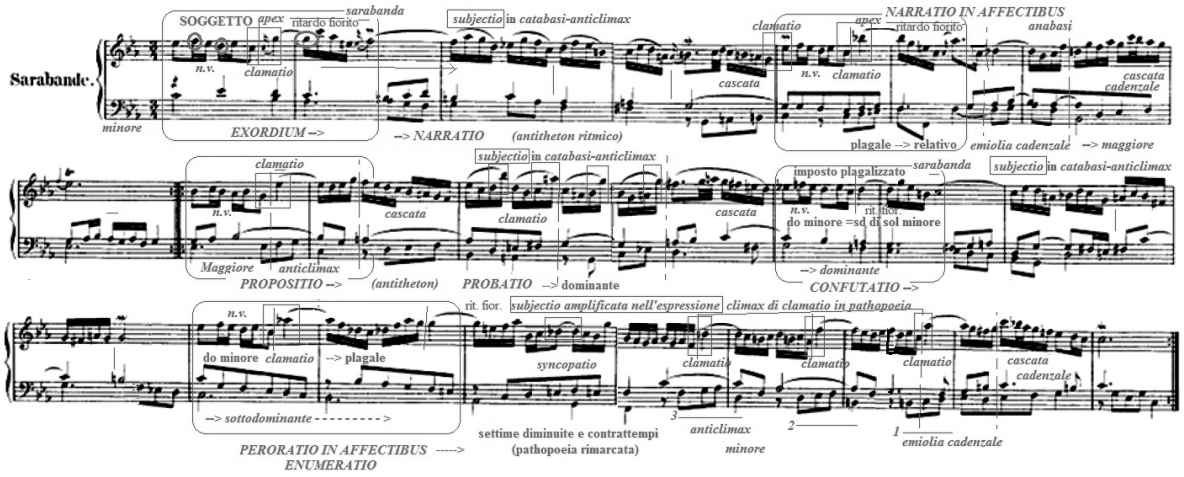
For this reason we cannot define western music as a single language but as a set of languages that use the same vocabulary (classical music, baroque music, pop music, jazz music, etc.).

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.

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 of which 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 → which refers to something extramusical (G.Malher Symphony n 10 - extract).
- 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 Trio - extract).

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.

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 (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 (words, chords).

1.1.4 Perceive and process

The ability to perceive and process musical information (not exclusively acoustic) is also common to all cultures.

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

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

- 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.

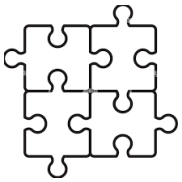
- environmental and cultural interaction → constant, quantitatively preponderant, continuous and more or less conscious perception of sounds organized according to the reference syntax of the cultural environment of reference.

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

To accomplish this 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 (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 a dedicated paragraph *Forms of human expression*.

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 → it 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.

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, etc.

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.

Speaking semiotically 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.

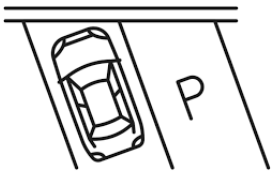
Music can be only comunication.

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 bythe 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 → are based on precise codes.

These codes:

- are formal systems that transcend individual choices, often imposing the use of certain mental categories.
- are not universal.
- are more or less lasting stabilizations of collective ways of thinking, acting, desiring, and preferring.
- 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?).

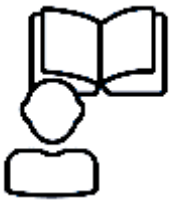
Social codes do not completely transcend individual choices but are linked to the ways in which the individual assumes them by mediating between collective and individual values.

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.

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 (composer) of interpretant signs.

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 he wants to transmit.

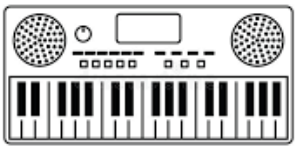
1.3.2 Aesthesic dimension

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.

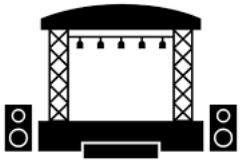
In the case of artistic languages, the interpretative signs attributed to the work by the emitter are not necessarily the same as those projected by the receiver because:

- 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.

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.



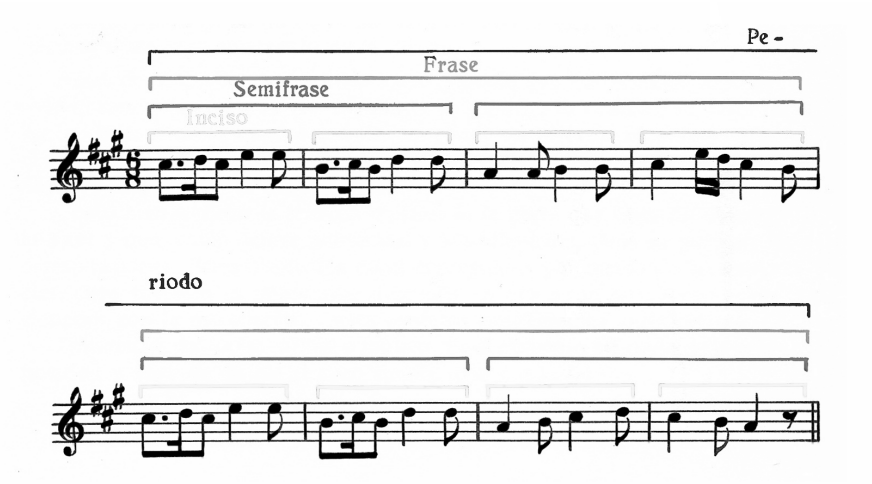
1.3.3 The work itself

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.

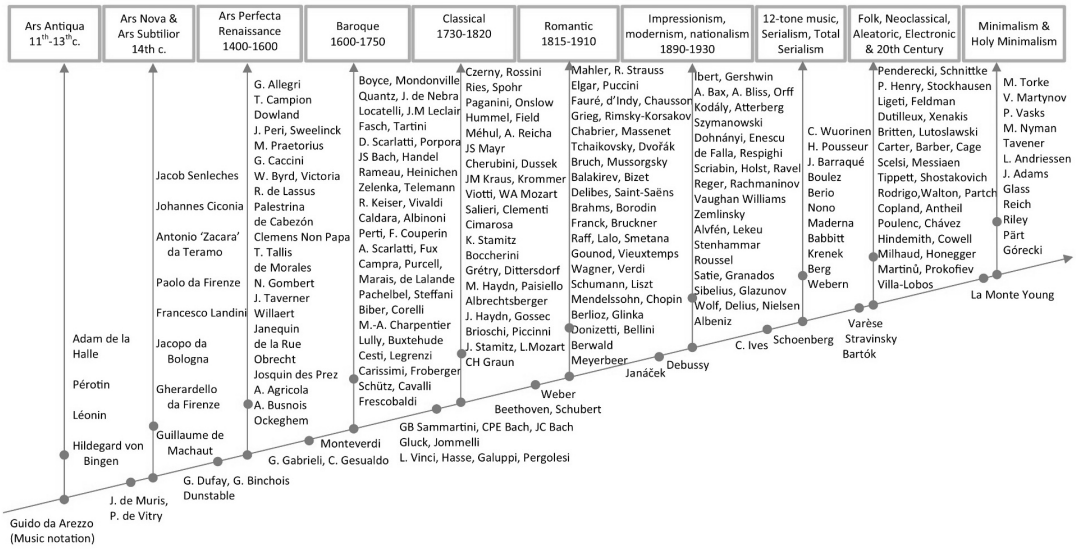


Each musical element (pitch, intensity, duration, timbre) can be separated and taken as a strategic variable in the production of a musical work.

Ending, 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.

For example, 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 (unless we are professional linguists).

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.

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 here 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:

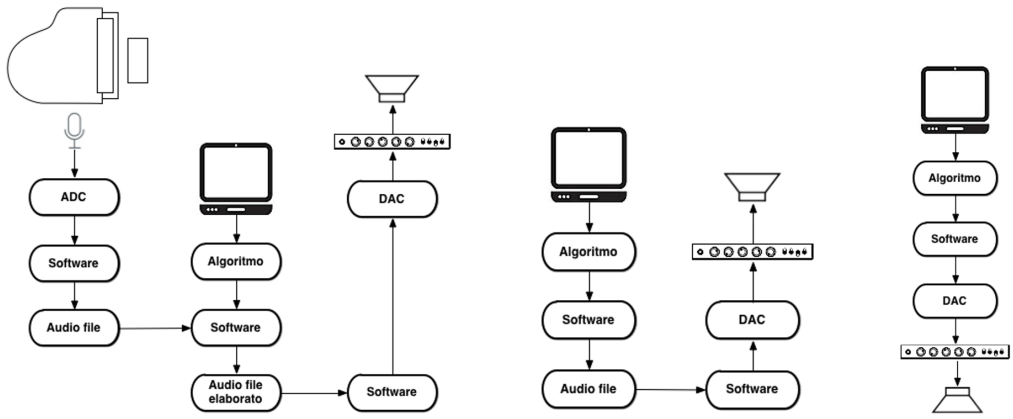
- interactions may influence the listener's perception of the sound text (mainly through the different characteristics of the diffusion systems used).
- interactions don't change the sound text.

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.

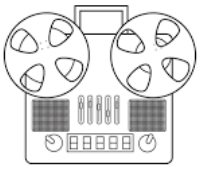
This used to be called 'tape music' but tapes are no longer used.

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) - extract.

- 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 Greekakousma: 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) - extract.

- 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 (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) - extract.

- 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 - extract.

- 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 of human beings.

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 - extract.

- 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) - extract.

- 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) - extract.

- 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) - extract.

- 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 essentially 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).
- differenze musicali rispetto ai brani acusmatici.

Anonymous - Dance! (2010) - extract.

- 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) - extracts.

- 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 significantly influences:

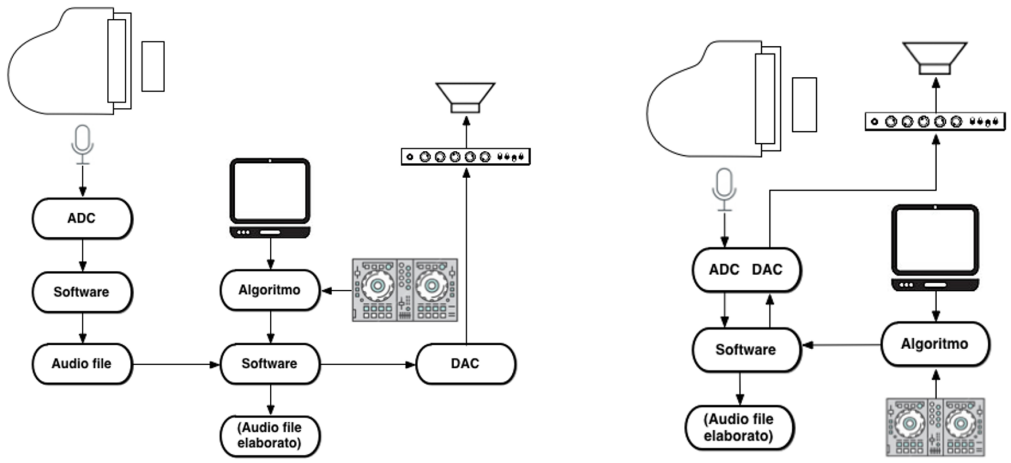
- the anthropological and contextualised perception of the musical action.
- its representative nature.

This category can be divided into three further subcategories:

- hyper-instruments.
- live set.
- 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 (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 Hyper-instrument

P.Boulez - Anthèmes 2 (1994) - extract.

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.

2.3.2 Live set

Slork - Twilight (2013) - extract.

Live performance where songs are performed 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.

2.3.3 Live coding

SuperCollider Live Coding - extract.

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.

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

2.4.1 Software installation

Download and install:

- sc_kernel (if you want to execute supercollider code in this notebook)
- SuperCollider

If you want to play SuperCollider from this notebook you have to select the textitsc_kernel.

Instead, I recommend copying and pasting the code into the SuperCollider IDE Interpreter.

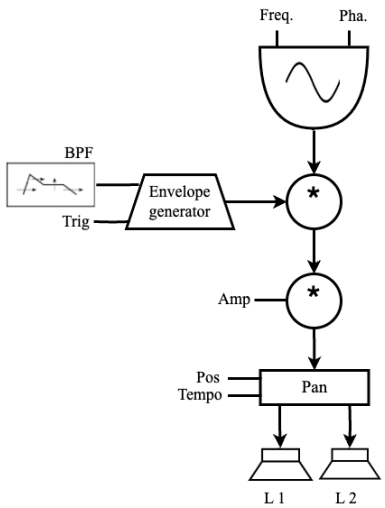
2.4.2 Instrumental model

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

- an oscillator.
- 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).



```
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.new(\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-destructs when the amplitude envelope expires (doneAction:2).

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

```
Synth.new(\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.new(\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.

```
a.free;
```

Or kill all Synths on the Server.

```
s.freeAll;
```

The type of voice allocation is strictly linked to the type of amplitude envelope.

With sustain (as in keyboards)

There isn't a total duration.

In this case we must send a message of *gate 1* (noteon) and then textitgate 0 (noteoff).

```
SynthDef.new(\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.new(\sust, [\done, 0]);
```

Listing 2.4: Instrument model with sustain

Send *gate 1* (noteon)

```
a.set(\amp, 0.5, \gate, 1);
```

Send *gate 0* (noteoff)

```
a.set(\gate, 0);
```

Kill the Synth.

```
a.free;
```

Without sustain (as in percussion or plucked intruments)

There is a total duration.

In this case we must send a gate message preceded by $t_$.

This produce an automatic *gate 0* message after a time calculated on the specified duration

```
SynthDef.new(\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.perc(0.1*dur,0.9*dur);
                            env = EnvGen.kr(env, t_gate, doneAction:done);
                            sig = sig * env * amp;
                            sig = Pan2.ar(sig, 0);
                        Out.ar(0, sig)
                        }).add;
a = Synth.new(\nosust, [\done, 0]);    // Create the Synth
```

Listing 2.5: Instrument model without sustain

Send *t_gate 1* (noteon + noteoff)

```
a.set(\amp, 0.5, \t_gate,1);
```

You can also trigger and kill it automatically.

```
a.set(\amp, 0.5, \done, 2, \t_gate,1);
```

Modifications to this model will be addressed in specific chapters.

Chapter 3

Fixed media

3.1 Historical framework

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:

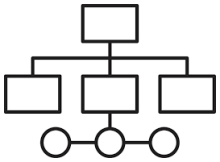
- *it is the point of convergence of all the soundwaves arriving from the sound points in the surrounding space.*
- *all the sound points in the initial space will be condensed into the microphone's membrane.*
- *the initial surrounding space is replaced by a sound point.*
- *if we play back it, it generates 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)

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 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

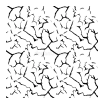
When we start to write a sound or musical work on a fixed media we should define few procedures:

1. choose and/or define the sound material (recorded or synthetic) 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, beting, 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).

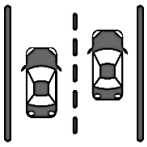
3.2.4 Time courses and musical forms

Some further thoughts regarding time courses and musical forms.

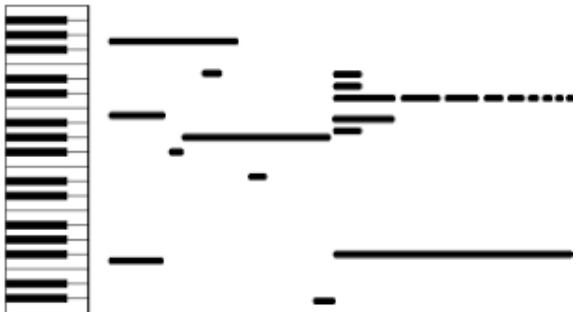
In the figurative arts (architecture, painting, sculpture) forms exist in the dimension of space.

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

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.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,p an=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  = -0.08; // 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; s// 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.

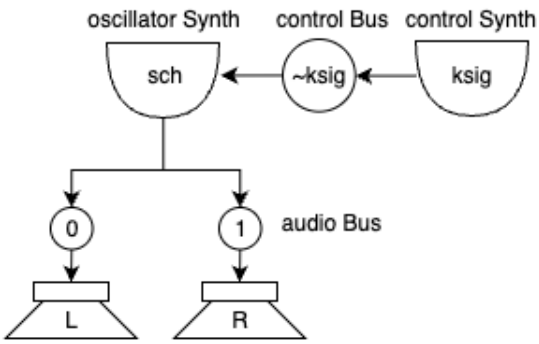
N.B. In this case the interaction is to be considered as preparation of the material to be fixed on the support not in the performance.

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, 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(0, 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.

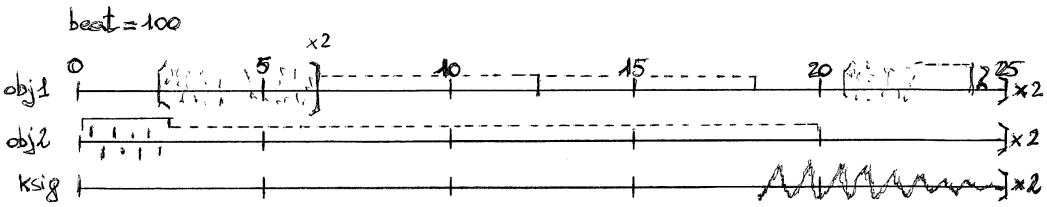
```
~bpm = 100;
t    = TempoClock.new(~bpm/60); // Set bpm tempo

~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 Historical framework

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.

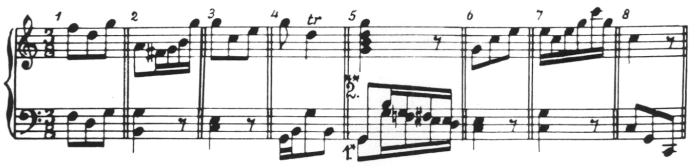
Part One

	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

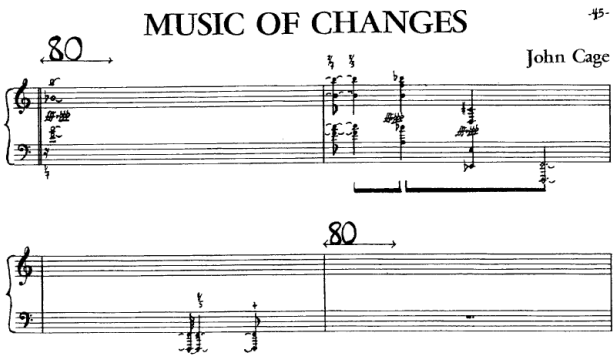
	I	II	III	IV	V	VI	VII	VIII
2	70	121	26	9	112	49	109	14
3	117	39	126	56	174	18	116	83
4	66	139	15	132	73	58	145	79
5	90	176	7	34	67	160	52	170
6	25	143	64	125	76	136	1	93
7	138	71	150	29	101	162	23	151
8	16	155	57	175	43	168	89	172
9	120	88	48	166	51	115	72	111
10	65	77	19	82	137	38	149	8
11	102	4	31	164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

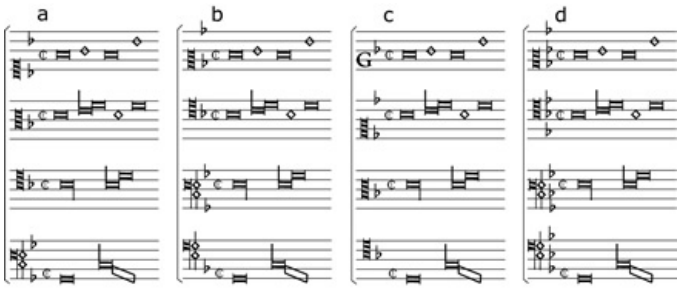
Indeterminacy relies on the performer's sensitivity.

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

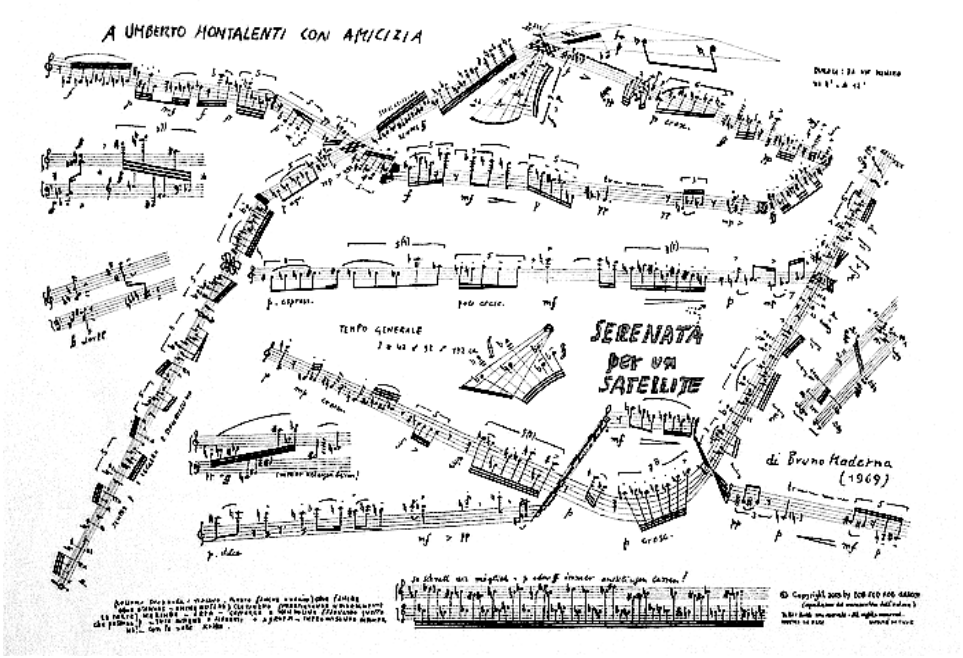
When taken to its extreme limits it ideally coincides with the opposite deterministic extreme: total improvisation or *libero arbitrio*.

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

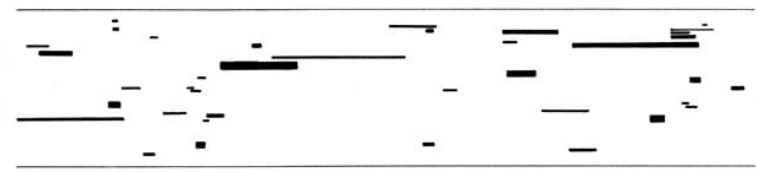
1. free choice of performance mode or form
   - Missa Cuiusvis Toni (1539) by Johannes Ockeghem.
     - the composer did not specify the clefs at the beginning of the staff.
     - it can be performed in all four modes (Dorian, Phrygian, Lydian, Mixolydian).
     - the choice of which clef to use for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



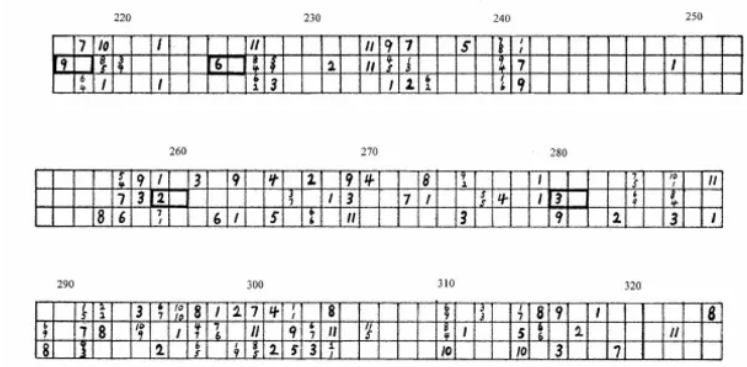
   - Serenata per un satellite (1969) by B. Maderna
     - It can be played by violin, flute (including piccolo), oboe (including oboe d'amore, even musette), clarinet (transposing the part, of course), marimba, harp, guitar, and mandolin (playing what they can), all together, individually, or in groups, 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}.defer(5);
```

One Buffer and a SynthDef.

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

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.

```
~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.

```
~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.

```
~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.

```
~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.

```
~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).

```
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;

```

- two audio bus to connect the source Synths with filters and FFT Synth.
- two control buses to control dynamically parameters (x, y and random signal values).
- one spectroscope.

```
x     = Bus.control(s, 1);
y     = Bus.control(s, 1);

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

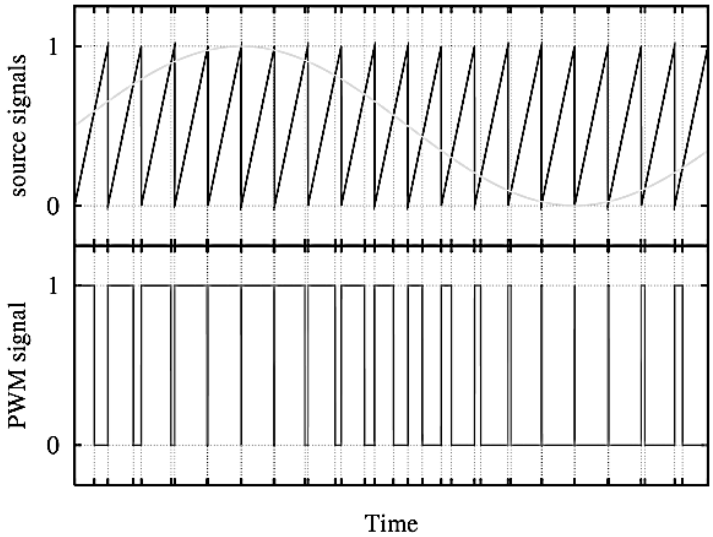
Pulse Width Modulation.

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).

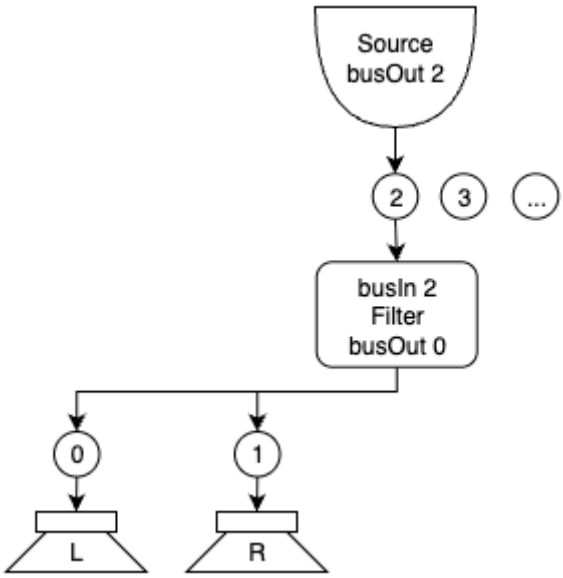
- the first we can set audio Bus out.
- the second audio Bus in.

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.

```
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.

```
~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.

```
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).

```
k = Synth(\ksig, [\type,0, \range, [0, 15], \busOut, x]); // Harmonicity index
j = Synth(\ksig, [\type,1, \range, [0, 25], \busOut, y]); // Modulation index
a = Synth(\fm,   [\freq,200, \amp,0.4, \hid,x.asMap, \mid,y.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.

```
h = Synth(\env,[\levels, [3.5,3.4,3.6,3.7,3.35],  // Harmonicity index
                \times,  [3,4,2,5],
                \dur, 40,
                \busOut, x]); // 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, y]); // write on control bus

a = Synth(\fm, [\freq,100, \amp,0.4, \hid,x.asMap, \mid,y.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 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 4.19: 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 4.20: 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 4.21: 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 4.22: 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 4.23: 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 4.24: 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

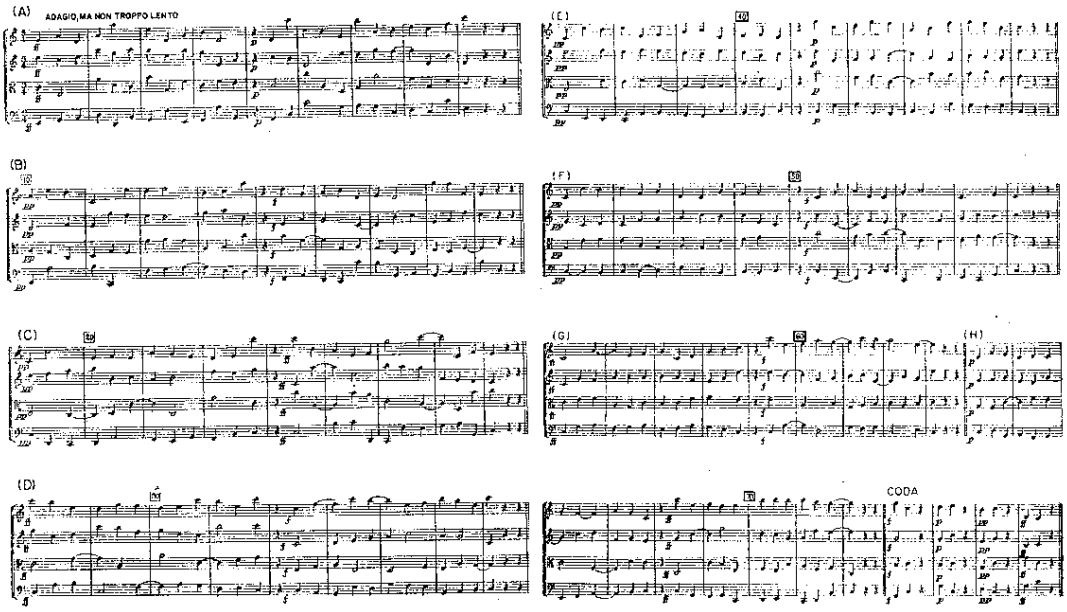
Algorithmic composition

5.1 Historical framework

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.

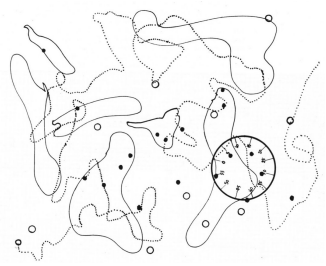
As has always been over the centuries we can consider two distinct aesthetic fields:

- 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.



It is a composition for small amplified sounds and an indeterminate number of 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.

6.1.2 Historical framework

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) → 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.

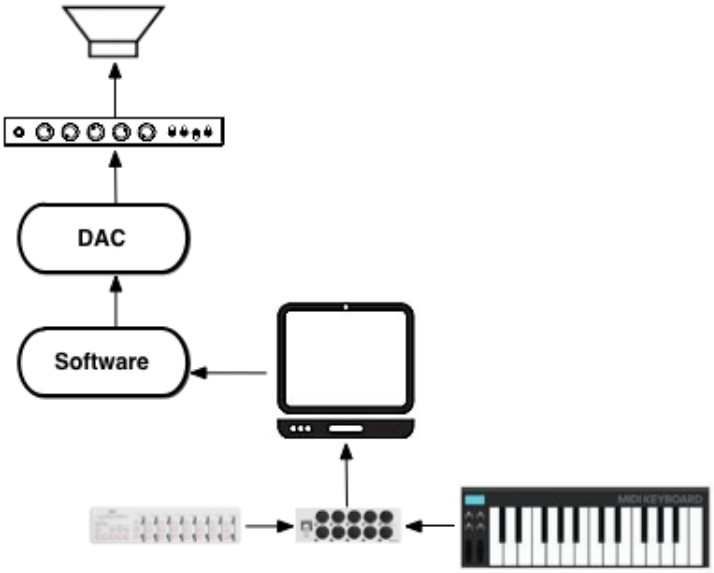
It includes:

- 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. METTI LINK DI GITHUB

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

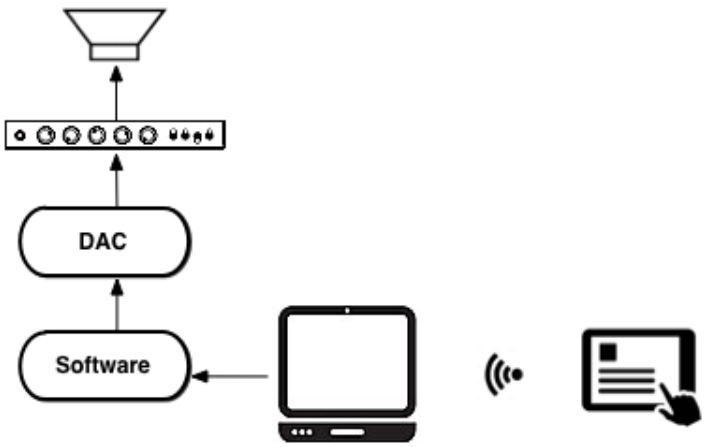
Open Sound Control.

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:

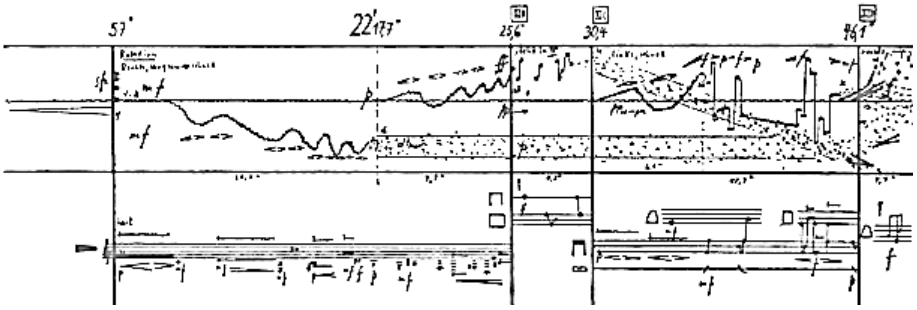
- one-shot → play it from beginning to end (old school).
- map key (HID, MIDI or OSC) → event (soundfile or other).
  - key → trig soundfile or Synth.
  - key → set playback rate (sampler-like transposition) or other parameters.
  - key → set start position (and duration) in a soundfile.
- play it in 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

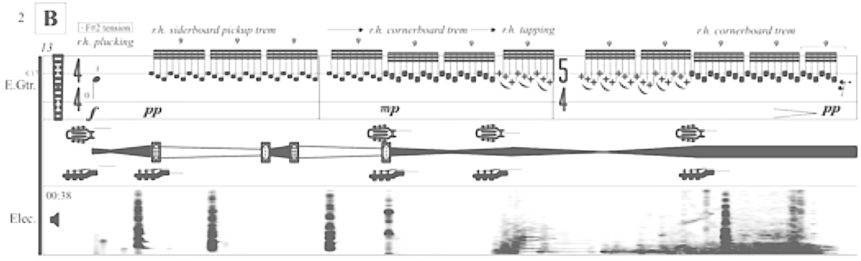
The old tape playback.

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.

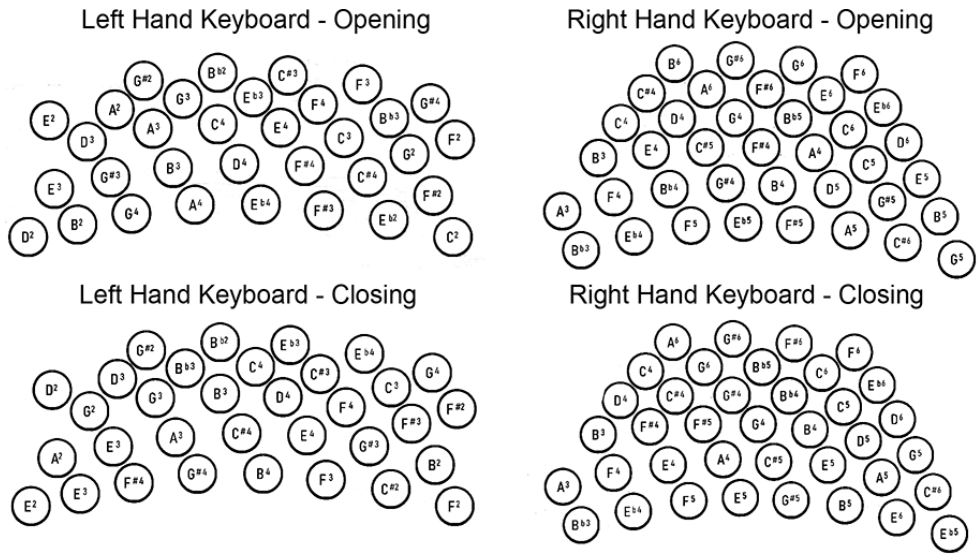
```
b = Buffer.read(s, "/absolute/path/to/voce.wav");
SynthDef.new(\simple_player, {arg buf=0, amp=0, pan=0, gate=0;
                                var env, sig;
```

```
                                    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.

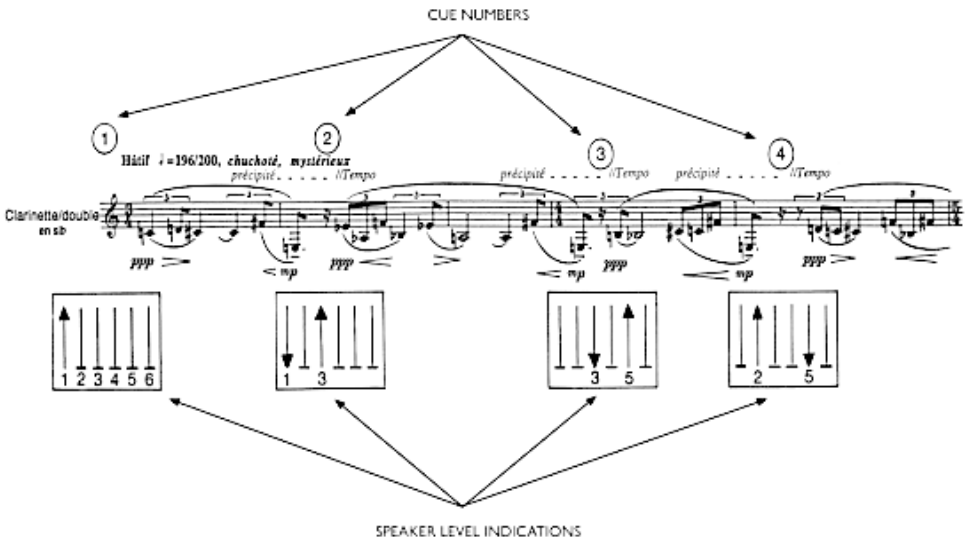
Positions and durations in seconds.

```
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

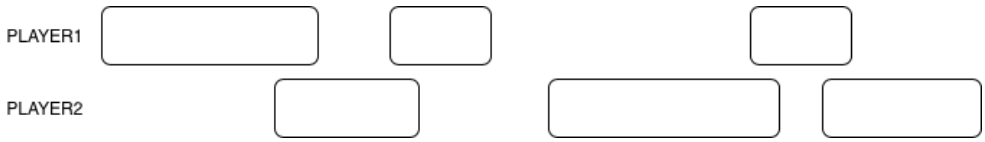
6.7.3 Cues

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.

Chapter 7

Hyper-instruments

7.1 Historical framework

7.2 Software model

7.2.1 Bus and Groups

7.3 Feature extraction as control signals

7.3.1 Envelope follower

7.3.2 Pitch follower

7.3.3 Centroid

7.4 Classic live electronics sound processing

7.4.1 Ring modulation

7.4.2 Live recording and playback

7.4.3 Granular synthesis

7.4.4 Delay lines

7.5 Composition sketches proposal

Chapter 8

Live coding

8.1 Historical framework

8.2 Composition sketches proposal

Listings

2.1 Instrument model . . . 40
2.2 Dynamic voice allocation . . . 40
2.3 Static voice allocation . . . 40
2.4 Instrument model with sustain . . . 41
2.5 Instrument model without sustain . . . 41
3.1 Load one sound file in a Buffer . . . 51
3.2 Load many sounfiles from a folder . . . 51
3.3 Sampler model . . . 52
3.4 Deterministic sequencing . . . 52
3.5 Percussive grain model . . . 53
3.6 Dynamic envelope model . . . 54
3.7 Control signals model . . . 55
3.8 Scratcher model . . . 55
3.9 Recording sound . . . 56
3.10 Scoring . . . 58
4.1 coin(0.5) . . . 65
4.2 rrand(min max) . . . 65
4.3 rand(max) . . . 65
4.4 rand2(+/-max) . . . 65
4.5 choose(Array) . . . 66
4.6 Source signals model . . . 66
4.7 Wawtable lookup synthesis model . . . 67
4.8 Vetorial synthesis model . . . 69
4.9 Envelope mapping parameter . . . 69
4.10 Pulse Width Modulation model . . . 70
4.11 Blip model . . . 71
4.12 Timbre stretching model . . . 71
4.13 Partial model . . . 72
4.14 Filters models . . . 74
4.15 Resonators model . . . 76
4.16 Basic FM model . . . 77
4.17 Classic FM model . . . 77
4.18 Envelope control model . . . 78
4.19 Spectral Delay model . . . 79
4.20 Brick Wall spectral filter model . . . 79
4.21 Rect Comb filter spectral model . . . 80
4.22 Mag Smear spectral model . . . 80
4.23 Freeze spectral model . . . 81
4.24 Morphing spectral model . . . 81
5.1 Live midi out sender model . . . 91
5.2 random incise function . . . 94

5.3 random motive function . . . 95
5.4 chromatic transposition function . . . 95
5.5 diatonic transposition function . . . 96
5.6 retrograde function . . . 96
5.7 inversion function . . . 97
5.8 retrograde of inversion function . . . 97
5.9 palindrome function . . . 97
5.10 all permutations function . . . 98
5.11 random order function . . . 98
5.12 chord model function . . . 100
5.13 voicing model function . . . 101
5.14 drop model functions . . . 102
5.15 voice-leading model function . . . 103
5.16 voice-leading common notes model function . . . 103
5.17 chord random walk model function . . . 104
5.18 top note melody model function . . . 105
5.19 random note melody model function . . . 105
5.20 arpeggiating note melody model function . . . 105
5.21 voice-leading style melody model function . . . 105
5.22 voice-leading 2 style melody model function . . . 105
5.23 melodic-walk style model function . . . 106
6.1 MIDI monophonic instrument model . . . 116
6.2 MIDI poliphonic no-sustain instrument model . . . 117
6.3 MIDI poliphonic w-sustain instrument model . . . 117
6.4 mouseDownAction model . . . 121
6.5 mouseUpAction model . . . 121
6.6 mouseMoveAction model . . . 122
6.7 mouseOverAction model . . . 123
6.8 mouse controls Server model . . . 124
6.9 keyDownAction model . . . 125
6.10 keyUpAction model . . . 125
6.11 key control Server model . . . 126
6.12 playback one-shot model . . . 127
6.13 playback key-map-soundfile model . . . 128
6.14 playback key-playback-rate model . . . 129
6.15 playback key-start-position model . . . 130
6.16 playback cues model . . . 131