



UNIVERSIDAD  
DE GRANADA



ETS  
INGENIERÍAS INFORMÁTICA  
Y DE TELECOMUNICACIÓN

PROGRAMA DE DOCTORADO EN TECNOLOGÍAS DE LA  
INFORMACIÓN Y LA COMUNICACIÓN

**Soft Computing**

DOCTORAL THESIS

# GenoMus

Towards artificial creativity through  
metaprogramming of musical genotypes

Author

José López-Montes

Thesis Directors

Miguel Molina-Solana

*Department Computer Science and AI – ETSIIT*

Waldo Fajardo Contreras

*Department Computer Science and AI – ETSIIT*

*a Jólogo*

## Acknowledgments

One of the greatest joys  
in my Brownian academic path  
was the reception of my research proposal  
by Waldo Fajardo and Miguel Molina.

Despite my limited knowledge in  
computer science and knowledge engineering,  
both of them supported this proposal  
from the beginning and welcomed it  
into their department at ETSIIT.  
Now I hope to correspond their trust.

Dori's nourishing support  
and the constant emotional care  
from Julia, Carlos and Diego  
have made it possible  
for this thesis to be completed,  
even when it seemed  
unachievable at many moments.

And at the core,  
there are my mother  
and my father (Jólogo),  
who nurtured in us  
a genuine enthusiasm  
for art and knowledge  
as two inseparable sides  
of the same plane.

## Agradecimientos

Una de las mayores alegrías  
de mi browniana trayectoria académica  
fue el recibimiento que  
Waldo Fajardo y Miguel Molina  
hicieron de mi propuesta de investigación.  
A pesar de que mis conocimientos en informática  
e ingeniería eran escasos, los dos apostaron desde  
el principio por esta propuesta y la acogieron  
en su departamento de la ETSIIT.  
Espero poder corresponder ahora a esa confianza.

El nutritivo apoyo de Dori  
y el cuidado emocional constante  
de Julia, Carlos y Diego  
han conseguido que,  
aunque en muchos momentos  
parecía que no lo lograría,  
esta tesis haya llegado a ser completada.

Y en el núcleo,  
están mi madre  
y mi padre (Jólogo),  
quienes incubaron en nosotros  
un entusiasmo genuino por  
el arte y el conocimiento  
como dos caras inseparables  
del mismo plano.

# Abstract

This doctoral thesis introduces GenoMus, a generative and bioinspired computational model for the development of artificial musical creativity based on metaprogramming. It is designed for both autonomous use and human-machine collaboration. The model comprises a representation system, a library of generative and auxiliary functions, and an interactive interface for musical experimentation.

The representation system defines the genotype as the functional tree of underlying procedures in a musical piece, and the phenotype as the musical outcome of these processes. Both elements interconnect to form a bidirectional generative grammar that serves as both a programming language and an abstract numerical representation. Its design is optimized to streamline one-dimensional encoding, metaprogramming, and seamless integration with any machine learning model utilizing numerical vectors.

GenoMus has been developed through a theoretical-practical methodology, undergoing testing in musical creation projects. The model's design has been shaped by these artistic experiences, evolving iteratively through a cycle of conceptual framework review, algorithm refinement, and practical application in musical compositions, presented as part of this research.

## **keywords:**

- **automatic musical composition**
- **metaprogramming**
- **procedural representation of music**
- **artificial creativity**
- **bioinspired composition**
- **GenoMus**

# Resumen

Esta tesis doctoral presenta GenoMus, un modelo computacional generativo y bioinspirado para el desarrollo de la creatividad musical artificial, basado en la metaprogramación, y diseñado tanto para su uso autónomo como para la colaboración humano-máquina. El modelo consta de un sistema de representación, una biblioteca de funciones generativas y auxiliares, y una interfaz interactiva para la experimentación musical.

El sistema de representación define como genotipo al árbol funcional de los procedimientos subyacentes en una pieza musical, y el fenotipo como el resultado musical de esos procesos. Ambos elementos se interrelacionan constituyendo una gramática generativa bidireccional que es simultáneamente un lenguaje de programación y una representación numérica abstracta. Su diseño se ha optimizado para simplificar la codificación unidimensional, la metaprogramación y la integración con cualquier modelo de aprendizaje automático que utilice vectores numéricos.

GenoMus se ha desarrollado desde una metodología teórico-práctica, poniéndose a prueba en proyectos de creación musical. El diseño del modelo ha sido guiado por estas experiencias artísticas, en un ciclo iterativo de revisión del marco conceptual, reescritura de los algoritmos y aplicación en composiciones musicales, presentadas como parte de esta investigación.

## **palabras clave:**

- **composición musical automática**
- **metaprogramación**
- **representación procedimental de la música**
- **creatividad artificial**
- **composición bioinspirada**
- **GenoMus**

# Contents

|   |             |
|---|-------------|
| <b>Abstract</b>   | <b>iv</b>   |
| <b>Resumen</b>  | <b>v</b>    |
| <b>Contents</b>   | <b>vi</b>   |
| <b>Figures</b>  | <b>xi</b>   |
| <b>Tables</b>   | <b>xvi</b>  |
| <b>Code listings</b>  | <b>xvii</b> |
| <b>Introduction</b>   | <b>1</b>    |
| Programming is (meta)composing . . . . .                        | 2           |
| Scope of the study . . . . .                                    | 4           |
| Hypothesis and research objectives . . . . .                    | 8           |
| Interactive experimental setup . . . . .                        | 10          |
| What this research is not about . . . . .                       | 13          |
| Thesis structure and reading recommendations . . . . .          | 14          |
| Source code and software . . . . .                              | 17          |
| <b>1 Background</b>   | <b>18</b>   |
| 1.1 Artificial creativity or creative artifice? . . . . .       | 20          |
| 1.2 The role of the machine in the evolution of style . . . . . | 21          |
| 1.3 Trends in CAC . . . . .                                     | 23          |
| 1.4 Exploration of abstract mathematical processes . . . . .    | 25          |
| 1.5 Grammars for automated composition . . . . .                | 27          |
| 1.6 CAC meets AI . . . . .                                      | 29          |
| 1.7 Bioinspired strategies . . . . .                            | 30          |
| 1.8 Metaprogramming and functional programming . . . . .        | 32          |
| 1.9 The problem of automated aesthetic evaluation . . . . .     | 34          |
| 1.10 Insights into aesthetic pleasure . . . . .                 | 37          |

|          |   |           |
|----------|---|-----------|
| <b>2</b> | <b>Conceptual and formal framework</b>                          | <b>38</b> |
| 2.1      | Composing composers . . . . .                                   | 38        |
| 2.2      | Music as an encoded functional grammar . . . . .                | 40        |
| 2.3      | The biological metaphor . . . . .                               | 41        |
| 2.4      | Formal definitions . . . . .                                    | 44        |
| 2.5      | Retrotranscription of genotypes into germinal vectors . . . . . | 47        |
| <b>3</b> | <b>Main data structures</b>                                     | <b>49</b> |
| 3.1      | Anatomy of a specimen . . . . .                                 | 49        |
| 3.2      | Function types . . . . .  | 54        |
| 3.3      | Anatomy of a genotype function . . . . .                        | 56        |
| 3.4      | Leaf types . . . . .  | 61        |
| 3.5      | Leaf parameters and mapping design . . . . .                    | 62        |
| 3.5.1    | Eligible values and Gaussian conversion . . . . .               | 63        |
| 3.5.2    | Generic parameter leaf . . . . .                                | 65        |
| 3.5.3    | voidLeaf . . . . .  | 66        |
| 3.5.4    | notevalueLeaf . . . . .   | 67        |
| 3.5.5    | midipitchLeaf . . . . .   | 69        |
| 3.5.6    | articulationLeaf . . . . .                                      | 71        |
| 3.5.7    | intensityLeaf . . . . .   | 72        |
| 3.5.8    | quantizedLeaf . . . . .   | 73        |
| 3.5.9    | Golden encoded integers and goldenintegerLeaf . . . . .         | 74        |
| 3.6      | Internal structure of the score . . . . .                       | 77        |
| 3.7      | Representations of generated music . . . . .                    | 81        |
| <b>4</b> | <b>Genotype functions</b>                                       | <b>84</b> |
| 4.1      | Identity functions . . . . .                                    | 84        |
| 4.2      | Lists . . . . .   | 86        |
| 4.3      | Formal structures . . . . .                                     | 87        |
| 4.4      | Deterministic and random processes . . . . .                    | 93        |
| 4.5      | Repetition and iteration . . . . .                              | 95        |
| 4.6      | Harmony . . . . .   | 97        |
| 4.7      | Generative subprocesses . . . . .                               | 105       |
| 4.8      | Recursions with type recursiveF . . . . .                       | 108       |
| 4.9      | Internal autoreferences . . . . .                               | 115       |
| 4.9.1    | The importance of self-reference in music . . . . .             | 115       |
| 4.9.2    | Subgenotype indexing . . . . .                                  | 116       |
| 4.9.3    | Minimal examples of internal autoreference . . . . .            | 117       |
| 4.9.4    | Definition of autoref functions . . . . .                       | 119       |

|          |   |            |
|----------|---|------------|
| 4.9.5    | Indexing tree and subgenotype calls . . . . .                     | 121        |
| 4.10     | Genotype functions libraries . . . . .                            | 124        |
| 4.10.1   | Creation and updating of libraries . . . . .                      | 124        |
| 4.10.2   | Indexing functions with golden encoded integers . . . . .         | 127        |
| 4.10.3   | Influence of the palette of eligible functions on style . . . . . | 128        |
| <b>5</b> | <b>Encoding and decoding</b>                                      | <b>129</b> |
| 5.1      | Genotype encoding . . . . .                                       | 130        |
| 5.1.1    | Leaf type identifiers . . . . .                                   | 130        |
| 5.1.2    | Leaf values . . . . .   | 131        |
| 5.1.3    | Function opening and closing flags . . . . .                      | 131        |
| 5.1.4    | Genotype function indices . . . . .                               | 131        |
| 5.2      | Minimal examples . . . . .  | 132        |
| 5.3      | Visualization of unidimensional vectors . . . . .                 | 133        |
| 5.4      | Germinal vector and genotype decoding . . . . .                   | 138        |
| 5.5      | Initial conditions for specimen rendering . . . . .               | 138        |
| 5.6      | Retrotranscription of genotypes as germinal vectors . . . . .     | 143        |
| 5.7      | Phenotype encoding . . . . .                                      | 144        |
| 5.8      | Phenotype decoding . . . . .                                      | 146        |
| <b>6</b> | <b>Specimens generation</b>                                       | <b>151</b> |
| 6.1      | Metaprogramming of genotypes . . . . .                            | 152        |
| 6.1.1    | Summary of the subprocesses . . . . .                             | 152        |
| 6.1.2    | The core metaprogramming subroutine . . . . .                     | 154        |
| 6.2      | Formatting of specimens . . . . .                                 | 162        |
| 6.3      | Specimen metadata . . . . .                                       | 165        |
| 6.3.1    | specimenID . . . . .  | 165        |
| 6.3.2    | comments . . . . .  | 165        |
| 6.3.3    | rating . . . . .  | 166        |
| 6.3.4    | generativityIndex . . . . .                                       | 166        |
| 6.3.5    | germinalVectorDeviation . . . . .                                 | 166        |
| 6.3.6    | history . . . . .   | 167        |
| 6.4      | Playback options as epigenetic conditions . . . . .               | 168        |
| 6.4.1    | Tempo control with playbackRate . . . . .                         | 168        |
| 6.4.2    | Rhythm quantization with minQuantizedNotevalue . . . . .          | 168        |
| 6.4.3    | Equal temperaments with stepsPerOctave . . . . .                  | 173        |



|                   |  |            |
|-------------------|--|------------|
| <b>7</b>          | <b>Evaluation and evolution</b>                                | <b>176</b> |
| 7.1               | Specimen and variations . . . . .                              | 177        |
| 7.1.1             | Music from pure randomness . . . . .                           | 177        |
| 7.1.2             | Starting with very short germinal vectors . . . . .            | 179        |
| 7.1.3             | Leaves mutation . . . . .                                      | 180        |
| 7.1.4             | Germinal vector mutation . . . . .                             | 183        |
| 7.2               | Coevolutionary techniques . . . . .                            | 185        |
| 7.3               | Sessions and global status . . . . .                           | 187        |
| 7.4               | Temperature and segmentation . . . . .                         | 189        |
| <b>8</b>          | <b>Procedural analysis</b>                                     | <b>191</b> |
| 8.1               | <i>Clapping Music</i> as a procedural genotype . . . . .       | 192        |
| 8.2               | Converting a procedure into a new genotype function . . . . .  | 196        |
| <b>9</b>          | <b>Results</b>   | <b>198</b> |
| 9.1               | A procedural framework optimized for metaprogramming . . . . . | 199        |
| 9.2               | Artistic research shaping software . . . . .                   | 201        |
| 9.3               | An open tool for augmented musical creativity . . . . .        | 204        |
| <b>10</b>         | <b>Conclusions</b>   | <b>205</b> |
| <b>11</b>         | <b>Conclusiones</b>  | <b>209</b> |
| <br>              |  |            |
| <b>Appendices</b> |  |            |
| <b>A</b>          | <b>GenoMus user interface</b>                                  | <b>213</b> |
| A.1               | Main patch . . . . .   | 213        |
| A.2               | Communication with core code . . . . .                         | 219        |
| A.3               | Control of initial conditions . . . . .                        | 220        |
| A.4               | Decoded genotype editor . . . . .                              | 221        |
| A.5               | Specimen monitoring . . . . .                                  | 224        |
| A.6               | Score viewers . . . . .  | 226        |
| A.7               | Selection and evolution of specimens . . . . .                 | 229        |
| A.8               | Outputs . . . . .  | 231        |
| <b>B</b>          | <b>Musical works</b>   | <b>232</b> |
| B.1               | <i>Threnody for Dimitris Christoulas</i> . . . . .             | 233        |
|                   | Artistic concept . . . . .                                     | 233        |
|                   | Reception . . . . .  | 234        |

|  |     |
|--|-----|
| Methods . . . . .                                    | 234 |
| A genetic algorithm for electronics . . . . .        | 235 |
| Genotypes and scores . . . . .                       | 240 |
| B.2 <i>Ada + Babbage – Capricci</i> . . . . .        | 259 |
| Artistic concept . . . . .                           | 259 |
| Genotypes and scores . . . . .                       | 262 |
| B.3 <i>Microcontrapunctus</i> . . . . .              | 287 |
| Artistic concept . . . . .                           | 287 |
| Methods . . . . .                                    | 288 |
| Waveforms and spectrograms . . . . .                 | 305 |
| B.4 <i>Seven Places</i> . . . . .                    | 322 |
| Artistic concept . . . . .                           | 322 |
| Methods . . . . .                                    | 324 |
| B.5 <i>Choral Riffs from Coral Reefs</i> . . . . .   | 326 |
| Artistic concept and methods . . . . .               | 326 |
| B.6 <i>Juno</i> . . . . .                            | 328 |
| Heuristics for harmony and instrumentation . . . . . | 328 |
| B.7 Openings for FACBA Podcasts . . . . .            | 331 |
| Ready-made music . . . . .                           | 332 |
| B.8 <i>Tiento</i> . . . . .                          | 333 |
| Artistic concept . . . . .                           | 333 |
| Methods . . . . .                                    | 334 |
| Waveform and spectrogram . . . . .                   | 336 |
| B.9 <i>Rudepoema na penumbra</i> . . . . .           | 342 |
| Artistic concept . . . . .                           | 342 |
| Methods . . . . .                                    | 342 |
| Waveform and spectrogram . . . . .                   | 343 |

|                     |            |
|---------------------|------------|
| <b>Bibliography</b> | <b>348</b> |
|---------------------|------------|

# Figures

|    |  |     |
|----|--|-----|
| 1  | Automation levels for symbolic music composition . . . . .                             | 4   |
| 2  | Scheme of development and experimental setup . . . . .                                 | 12  |
| 3  | Mappings from germinal conditions to decoded phenotypes . . . . .                      | 48  |
| 4  | Mapping uniform distribution to Gaussian-like . . . . .                                | 65  |
| 5  | Conversion from generic parameter to notevalue . . . . .                               | 67  |
| 6  | Logarithmic plot of conversion from generic parameter to notevalue . . . .             | 68  |
| 7  | Conversion from generic parameter to midipitch . . . . .                               | 70  |
| 8  | Conversion from generic parameter to articulation . . . . .                            | 72  |
| 9  | Conversion from generic parameter to intensity . . . . .                               | 73  |
| 10 | Conversion from generic parameter to quantized value . . . . .                         | 74  |
| 11 | Conversion from integer to golden encoded integer . . . . .                            | 75  |
| 12 | Constituent structures of a score . . . . .  | 78  |
| 13 | Visual depiction of merging scores . . . . .   | 80  |
| 14 | Condensed SVG score . . . . .  | 82  |
| 15 | Multivoice output SVG score . . . . .  | 83  |
| 16 | Simple score created with <b>sConcatS</b> . . . . .                                    | 93  |
| 17 | Random lists produced by <b>lRnd</b> and <b>lGaussianRnd</b> . . . . .                 | 94  |
| 18 | Comparison between repetition and iteration of the same event . . . . .                | 95  |
| 19 | Iteration of a list containing a random item . . . . .                                 | 97  |
| 20 | Steps to assemble a harmonicGrid . . . . .   | 100 |
| 21 | Influence of the degree of chromaticism . . . . .                                      | 101 |
| 22 | Adjustment of an entire score to a pentatonic scale with <b>sHarmonicGrid</b> . .      | 103 |
| 23 | Score with two transpositions of a pitch class set as harmonic grids . . . . .         | 104 |
| 24 | Three versions of a <b>vMotiv</b> with <b>lBrownian</b> generative functions . . . . . | 107 |
| 25 | Sequences generated with <b>lLogisticMap</b> . . . . .                                 | 109 |
| 26 | Score generated using recursiveF functions . . . . .                                   | 115 |
| 27 | Minimal example of autoreference . . . . .   | 119 |
| 28 | Selection of genotype functions with golden encoded integers . . . . .                 | 132 |
| 29 | Decomposition and encoding of minimal genotypes . . . . .                              | 133 |
| 30 | Genotype and its annotated visualization . . . . .                                     | 135 |

|    |   |     |
|----|---|-----|
| 31 | Color assignment for leaf values . . . . .                                  | 136 |
| 32 | Monochromatic visualization of an encoded genotype . . . . .                | 136 |
| 33 | Encoded and decoded genotype, along with its visualization . . . . .        | 137 |
| 34 | Encoding genotypes from germinal vectors . . . . .                          | 139 |
| 35 | Germinal vector in parallel to encoded and decoded genotype . . . . .       | 141 |
| 36 | Visualization of a germinal vector and corresponding encoded genotype . .   | 142 |
| 37 | Equivalent germinal vectors to obtain the same genotype . . . . .           | 143 |
| 38 | Automated rule set for encoding phenotypes into one-dimensional vectors     | 144 |
| 39 | Germinal vector, encoded genotype, and corresponding encoded phenotype      | 147 |
| 40 | Score corresponding to the decodedPhenotype in Listing 49 . . . . .         | 149 |
| 41 | Composition of the specimenID to assign unique specimen names . . . . .     | 165 |
| 42 | Comparison of the same phenotype with different quantization values . . .   | 172 |
| 43 | Visualized encoded phenotypes with variations in minQuantizedNotevalue      | 173 |
| 44 | Comparison of the same excerpt with different temperaments . . . . .        | 174 |
| 45 | Excerpts with non-integer values for stepsPerOctave . . . . .               | 175 |
| 46 | Two examples of specimens generated with a very short germinal vector . .   | 178 |
| 47 | Visualization of a germinal vector and corresponding encoded genotype . .   | 179 |
| 48 | Four versions of a genotype with progressive leaves mutations . . . . .     | 180 |
| 49 | Comparison between a germinal vector and its mutation . . . . .             | 183 |
| 50 | Genotype with progressive germinal vector mutations . . . . .               | 184 |
| 51 | Genotype with progressive germinal vector mutations (continuation) . . . .  | 185 |
| 52 | Scores generated by small mutations on an initial germinal vector . . . . . | 186 |
| 53 | Compressed score of Reich's <i>Clapping Music</i> . . . . .                 | 192 |
| 54 | Functional tree of <i>Clapping Music</i> decoded genotype . . . . .         | 196 |
| 55 | Visualization of <i>Clapping Music</i> encoded genotype . . . . .           | 197 |
| 56 | GenoMus main patch in presentation mode . . . . .                           | 214 |
| 57 | GenoMus main patch in edition mode . . . . .                                | 214 |
| 58 | Communication subpatch with the GenoMus core code . . . . .                 | 219 |
| 59 | Control of the set of eligible functions . . . . .                          | 220 |
| 60 | Graphical display of the germinal vector . . . . .                          | 221 |
| 61 | Decoded genotype text editor with compressed formatting . . . . .           | 221 |
| 62 | Decoded genotype text editor with semiexpanded formatting . . . . .         | 222 |
| 63 | Decoded genotype text editor with expanded formatting . . . . .             | 223 |
| 64 | Viewers of encoded data and bach.roll converted data . . . . .              | 224 |
| 65 | Viewer for the complete data of the specimen as formatted text . . . . .    | 224 |
| 66 | Specimen viewer . . . . .   | 225 |
| 67 | Collapsed score viewer with a microtonal example . . . . .                  | 226 |
| 68 | Collapsed score viewer showing extra parameters, event tags and timegrid.   | 226 |

|     |  |     |
|-----|--|-----|
| 69  | Main score viewer displaying voices separately . . . . .                                       | 227 |
| 70  | Subpatch evolution to handle evolutionary processes . . . . .                                  | 229 |
| 71  | Subpatch sessionInfo to monitor metadata of generative sessions . . . . .                      | 229 |
| 72  | Subpatch outputs . . . . .   | 231 |
| 73  | <i>Threnody for Dimitris Christoulas</i> — all genotypes and graphical phenotypes              | 237 |
| 74  | <i>Recursio I-a</i> — genotype, graphical phenotype and beginning of score . . .               | 240 |
| 75  | <i>Recursio II</i> — genotype, graphical phenotype and beginning of score . . . .              | 241 |
| 76  | <i>Recursio III-a</i> — genotype, graphical phenotype and beginning of score . .               | 242 |
| 77  | <i>Recursio III-b</i> — genotype, graphical phenotype and beginning of score . .               | 243 |
| 78  | <i>Recursio IV-a</i> — genotype, graphical phenotype and beginning of score . .                | 244 |
| 79  | <i>Recursio V</i> — genotype, graphical phenotype and beginning of score . . . .               | 245 |
| 80  | <i>Recursio IV-b</i> — genotype, graphical phenotype and beginning of score . .                | 246 |
| 81  | <i>Recursio VII</i> — genotype, graphical phenotype and beginning of score . . .               | 247 |
| 82  | <i>Recursio VIII</i> — genotype, graphical phenotype and beginning of score . . .              | 248 |
| 83  | <i>Recursio IX</i> — genotype, graphical phenotype and beginning of score . . . .              | 249 |
| 84  | <i>Recursio X</i> — genotype, graphical phenotype and beginning of score . . . .               | 250 |
| 85  | <i>Recursio I-b</i> — genotype, graphical phenotype and beginning of score . . . .             | 251 |
| 86  | <i>Recursio XI</i> — genotype, graphical phenotype and excerpts from score . . .               | 252 |
| 87  | <i>Recursio XI</i> — score continuation . . . . .  | 253 |
| 88  | <i>Recursio XII</i> — genotype, graphical phenotype and score . . . . .                        | 254 |
| 89  | <i>Recursio XIII</i> — genotype and graphical phenotypes, <i>XIII-a</i> & <i>b</i> — score . . | 255 |
| 90  | <i>Recursio XIII-d2</i> & <i>d3</i> — score . . . . .  | 256 |
| 91  | <i>Recursio XIII-d4</i> & <i>d5</i> — score . . . . .  | 257 |
| 92  | <i>Recursio XIII-d6</i> — score . . . . .  | 258 |
| 93  | Max patch for study of multitempi with individual click tracks . . . . .                       | 258 |
| 94  | <i>Ada + Babbage - Capricci</i> — graphical phenotypes . . . . .                               | 261 |
| 95  | <i>Capriccio I</i> — genotype, graphical phenotype and beginning of score . . . .              | 263 |
| 96  | <i>Capriccio II</i> — genotype, graphical phenotype and beginning of score . . . .             | 264 |
| 97  | <i>Capriccio III</i> — genotype, graphical phenotype and beginning of score . . .              | 266 |
| 98  | <i>Capriccio IV</i> — genotype, graphical phenotype and beginning of score . . .               | 267 |
| 99  | <i>Capriccio V</i> — genotype, graphical phenotype and beginning of score . . . .              | 268 |
| 100 | <i>Capriccio VI</i> — genotype, graphical phenotype and beginning of score . . .               | 270 |
| 101 | <i>Capriccio VII</i> — genotype, graphical phenotype and beginning of score . . . .            | 271 |
| 102 | <i>Capriccio VIII</i> — genotype, graphical phenotype and beginning of score . .               | 273 |
| 103 | <i>Capriccio IX</i> — genotype, graphical phenotype and beginning of score . . .               | 275 |
| 104 | <i>Capriccio X</i> — genotype, graphical phenotype and beginning of score . . . .              | 276 |
| 105 | <i>Capriccio XI</i> — genotype, graphical phenotype and beginning of score . . .               | 278 |
| 106 | <i>Capriccio XII</i> — genotype, graphical phenotype and beginning of score . . .              | 280 |

|     |  |     |
|-----|--|-----|
| 107 | <i>Capriccio XIII</i> — genotype, graphical phenotype and beginning of score . . .   | 282 |
| 108 | <i>Capriccio XIV</i> — genotype, graphical phenotype and beginning of score . . .    | 283 |
| 109 | <i>Capriccio XV</i> — genotype, graphical phenotype and beginning of score . . .     | 285 |
| 110 | <i>Capriccio XVI</i> — genotype, graphical phenotype and beginning of score . . .    | 286 |
| 111 | <i>Microcontrapunctus</i> — Speakers setup at Istituto Pietro Mascagni (Livorno)     | 288 |
| 112 | Basic sound created with the Csound instrument for <i>Microcontrapunctus</i> . . .   | 292 |
| 113 | Modifications of the attack . . . . .  | 293 |
| 114 | Amplitude envelopes . . . . .  | 294 |
| 115 | Application of different powers to the amplitude envelope . . . . .                  | 295 |
| 116 | Addition of a modulating noise source . . . . .                                      | 296 |
| 117 | Additional ring modulation . . . . .   | 297 |
| 118 | Single sound grain with different modulations . . . . .                              | 298 |
| 119 | Csound score and sequence of various microsounds . . . . .                           | 299 |
| 120 | Data generated by <i>Microcontrapunctus'</i> genotype functions of type list . . . . | 301 |
| 121 | Data flow from the encoded genotype to the microsound sequence . . . . .             | 303 |
| 122 | Example of actual fragment of the final composition . . . . .                        | 305 |
| 123 | <i>Microcontrapunctus</i> — waveforms and spectrogram 0:24.5 - 0:27.0 . . . . .      | 306 |
| 124 | <i>Microcontrapunctus</i> — waveforms and spectrogram 0:55.3 - 0:58.1 . . . . .      | 307 |
| 125 | <i>Microcontrapunctus</i> — waveforms and spectrogram 1:07.1 - 1:09.9 . . . . .      | 308 |
| 126 | <i>Microcontrapunctus</i> — waveforms and spectrogram 1:26.3 - 1:29.1 . . . . .      | 309 |
| 127 | <i>Microcontrapunctus</i> — waveforms and spectrogram 1:32.6 - 1:35.9 . . . . .      | 310 |
| 128 | <i>Microcontrapunctus</i> — waveforms and spectrogram 1:43.8 - 1:47.0 . . . . .      | 311 |
| 129 | <i>Microcontrapunctus</i> — waveforms and spectrogram 2:12.7 - 2:16.0 . . . . .      | 312 |
| 130 | <i>Microcontrapunctus</i> — waveforms and spectrogram 2:30.0 - 2:32.8 . . . . .      | 313 |
| 131 | <i>Microcontrapunctus</i> — waveforms and spectrogram 3:06.1 - 3:09.3 . . . . .      | 314 |
| 132 | <i>Microcontrapunctus</i> — waveforms and spectrogram 3:26.8 - 3:30.0 . . . . .      | 315 |
| 133 | <i>Microcontrapunctus</i> — waveforms and spectrogram 3:30.4 - 3:33.1 . . . . .      | 316 |
| 134 | <i>Microcontrapunctus</i> — waveforms and spectrogram 4:19.7 - 4:22.4 . . . . .      | 317 |
| 135 | <i>Microcontrapunctus</i> — waveforms and spectrogram 4:29.1 - 4:31.3 . . . . .      | 318 |
| 136 | <i>Microcontrapunctus</i> — waveforms and spectrogram 5:36.0 - 5:38.7 . . . . .      | 319 |
| 137 | <i>Microcontrapunctus</i> — waveforms and spectrogram 6:11.3 - 6:14.0 . . . . .      | 320 |
| 138 | <i>Microcontrapunctus</i> — waveforms and spectrogram 7:13.0 - 7:15.8 . . . . .      | 321 |
| 139 | <i>Seven Places</i> — score's first page . . . . .                                   | 323 |
| 140 | <i>Seven Places</i> — video script, score and tape spectrogram, bars 59 to 69 . . .  | 324 |
| 141 | <i>Seven Places</i> — video script, score and tape spectrogram, bars 70 to 79 . . .  | 325 |
| 142 | Spectrograms of excerpts from <i>Choral Riffs from Coral Reefs</i> . . . . .         | 327 |
| 143 | <i>Juno</i> — bars 27 to 47 . . . . .  | 329 |
| 144 | <i>Juno</i> — bars 78 to 93 . . . . .  | 330 |

|     |   |     |
|-----|---|-----|
| 145 | First conceptualization of germinal vector and retrotranscription . . . . . | 335 |
| 146 | Spectrogram of <i>Tiento</i> . . . . .                                      | 341 |
| 147 | Spectrogram of <i>Rudepoema na penumbra</i> . . . . .                       | 347 |

# Tables

|    |   |     |
|----|---|-----|
| 1  | Definitions of key concepts . . . . .   | 43  |
| 2  | Specimen data structure . . . . .   | 54  |
| 3  | Genotype function types, tags, and identifiers . . . . .  | 55  |
| 4  | Object keys of a subspecimen returned by a genotype function . . . . .                                    | 57  |
| 5  | Leaf types labels . . . . .   | 62  |
| 6  | Arguments to configure a harmonicGrid . . . . .   | 98  |
| 7  | Data structure of the <b>eligibleFunctionsLibrary</b> Object . . . . .                                    | 127 |
| 8  | Numerical encoding of functional expression tokens . . . . .  | 134 |
| 9  | Description and examples of initialConditions for specimen generation . . . . .                           | 140 |
| 10 | Functions related to creation and modification of specimens . . . . .                                     | 154 |
| 11 | Transformations applied to obtain a new generation . . . . .  | 187 |
| 12 | Genotype functions used to model <i>Clapping Music</i> . . . . .  | 193 |
| 13 | Minimal elements required to model <i>Clapping Music</i> procedurally . . . . .                           | 193 |
| 14 | Subgenotypes stored in <i>Clapping Music</i> specimen . . . . .   | 195 |
| 15 | Data structure of new genotype function <b>sClapping</b> . . . . .  | 197 |
| 16 | Iterative stages of prototype creation . . . . .  | 204 |
| 17 | Functionalities of the GenoMus main patch . . . . .   | 219 |
| 18 | Functionalities of the score viewer . . . . .   | 228 |
| 19 | Functionalities of the evolution subpatch . . . . .   | 231 |
| 20 | <i>Threnody for Dimitris Christoulas</i> : formulae for <i>Rekursio I-a</i> to <i>XII</i> . . . . .       | 238 |
| 21 | <i>Threnody for Dimitris Christoulas</i> : formulae for <i>Rekursio XIII-a</i> to <i>XII-d4</i> . . . . . | 239 |
| 22 | <i>Microcontrapunctus</i> — genotype functions . . . . .  | 300 |

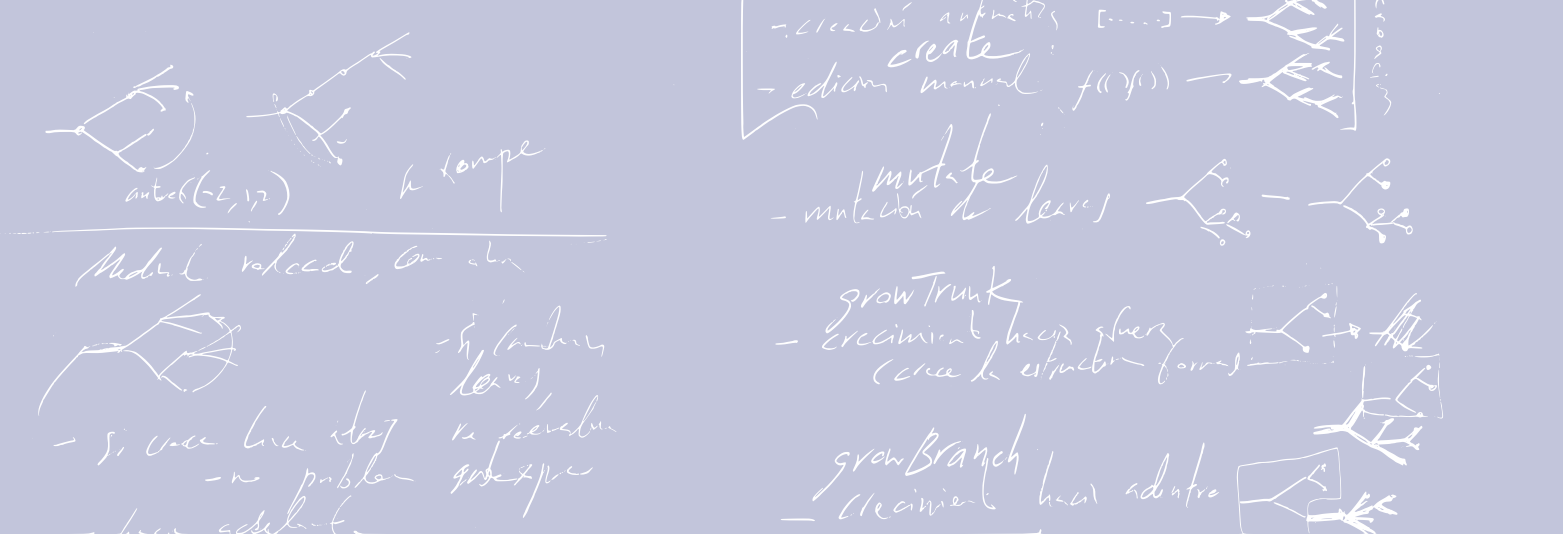


## Code listings

|    |  |     |
|----|--|-----|
| 1  | Simple subspecimen . . . . .   | 50  |
| 2  | Simple rendered specimen . . . . .   | 51  |
| 3  | Minimal genotype function example (midipitchF identity function <b>m</b> ) . . . . | 58  |
| 4  | Subspecimen returned by expression <b>m</b> (60) . . . . .                         | 58  |
| 5  | Declaration of <b>vMotif</b> function . . . . .                                    | 59  |
| 6  | Indexing genotype functions in genotypeFunctionsLibrary . . . . .                  | 60  |
| 7  | Genotype functions indexed in genotypeFunctionsLibrary . . . . .                   | 61  |
| 8  | Subspecimen returned by a generic parameter . . . . .                              | 66  |
| 9  | Subspecimen returned by <b>pRnd</b> () . . . . .                                   | 66  |
| 10 | Implementation of conversion to golden encoded integer . . . . .                   | 76  |
| 11 | Subspecimen returned by eventF identity function . . . . .                         | 85  |
| 12 | Subspecimen returned by eventF identity function with generic parameters           | 85  |
| 13 | Identity functions after reaching depth limit . . . . .                            | 86  |
| 14 | Subspecimen returned by listF identity function . . . . .                          | 87  |
| 15 | Subspecimen returned by lmidipitchF identity function . . . . .                    | 87  |
| 16 | Implementation of <b>vConcatV</b> function . . . . .                               | 88  |
| 17 | Implementation of <b>mergeScores</b> auxiliary function . . . . .                  | 89  |
| 18 | Concatenated scores with <b>sConcatS</b> . . . . .                                 | 92  |
| 19 | Implementation of subgenotype iteration <b>vIterE</b> . . . . .                    | 96  |
| 20 | Iteration of lists with <b>lIterL</b> . . . . .                                    | 97  |
| 21 | Identity function <b>h</b> of harmonyF type . . . . .                              | 99  |
| 22 | Subspecimen of a harmonyF function . . . . .                                       | 99  |
| 23 | Definition of a function to create harmonic grids with octatonic scales . . . .    | 102 |
| 24 | Genotype with two simultaneous harmonic grids based on pitch class sets .          | 104 |
| 25 | Beginning of the definition of the generative function <b>lBrownian</b> . . . . .  | 105 |
| 26 | Framework auxiliary function to create list converters . . . . .                   | 106 |
| 27 | <b>vMotiv</b> with <b>lBrownian</b> generative functions . . . . .                 | 106 |
| 28 | Details of the implementation of a generative functions . . . . .                  | 107 |
| 29 | <b>vMotiv</b> with <b>lLogisticMap</b> generative functions . . . . .              | 108 |
| 30 | Function <b>lRecursioOrder2</b> with recursiveF argument . . . . .                 | 109 |

|    |  |     |
|----|--|-----|
| 31 | Simplest subspecimen of a recursiveF function . . . . .                                | 111 |
| 32 | Genotype with a Fibonacci-like recursion . . . . .                                     | 112 |
| 33 | Genotype with recursions . . . . .   | 112 |
| 34 | Subspecimen returned by a compound recursion . . . . .                                 | 114 |
| 35 | Definition of <b>lConcatL</b> genotype function . . . . .                              | 116 |
| 36 | Indexation of subgenotypes with <b>indexDecGens</b> . . . . .                          | 116 |
| 37 | Internal autoreference to a score . . . . .  | 117 |
| 38 | Example of <b>subGenotypes</b> Object . . . . .  | 117 |
| 39 | Equivalent genotype after evaluating internal autoreference to a score . . .           | 118 |
| 40 | Equivalent genotype with autoreferences . . . . .                                      | 119 |
| 41 | Framework function <b>autoref</b> to create all autoreferences functions . . . . .     | 120 |
| 42 | Definition of some autoreference functions . . . . .                                   | 121 |
| 43 | Visualization of autoreferences and indexing order . . . . .                           | 122 |
| 44 | Object <b>subGenotypes</b> created after evaluating a decoded genotype . . . . .       | 123 |
| 45 | Function <b>indexGenotypeFunction</b> to create an all functions library . . . . .     | 124 |
| 46 | Data structure of <b>genotypeFunctionsLibrary</b> data structure . . . . .             | 125 |
| 47 | Implementation of <b>vConcatV</b> . . . . .  | 145 |
| 48 | Simple decoded genotype using events with one extra parameter . . . . .                | 146 |
| 49 | Data structure of a phenotype within a specimen . . . . .                              | 148 |
| 50 | Implementation of core function <b>createGenotype</b> . . . . .                        | 154 |
| 51 | Function <b>evalExpr</b> as alternative to eval . . . . .                              | 160 |
| 52 | A call to the function <b>createGenotype</b> . . . . .                                 | 160 |
| 53 | Object returned by <b>createGenotype</b> . . . . .                                     | 160 |
| 54 | Decoded genotype generated by a germinal vector of only three values . . .             | 162 |
| 55 | Implementation of specimens creator <b>specimenDataStructure</b> . . . . .             | 163 |
| 56 | Implementation of <b>specimenMinimalData</b> . . . . .                                 | 164 |
| 57 | Example of history in a specimen metadata . . . . .                                    | 167 |
| 58 | Decoded genotype with <b>depthThreshold = 12</b> . . . . .                             | 169 |
| 59 | Comparison of decoded genotypes before and after a leaves mutation . . .               | 181 |
| 60 | Array with positions and values of leaves returned by <b>extractLeaves</b> . . .       | 181 |
| 61 | <b>mutateSpecimenLeaves</b> creates variations of specimens by mutating leaves .       | 182 |
| 62 | Example of a GenoMus session stored in <b>statusGenoMus</b> . . . . .                  | 188 |
| 63 | Function <b>segmentation</b> to control the composition of new generations . . .       | 189 |
| 64 | Temperature-based segmentation of transformations applied to evolution .               | 190 |
| 65 | Decoded genotype of <i>Clapping Music</i> model . . . . .                              | 194 |
| 66 | <i>Threnody for Dimitris Christoulas</i> — Genotype of <i>Rekursio I–a</i> . . . . .   | 240 |
| 67 | <i>Threnody for Dimitris Christoulas</i> — Genotype of <i>Rekursio II</i> . . . . .    | 241 |
| 68 | <i>Threnody for Dimitris Christoulas</i> — Genotype of <i>Rekursio III–a</i> . . . . . | 242 |

|    |  |     |
|----|--|-----|
| 69 | <i>Threnody for Dimitris Christoulas</i> — Genotype of <i>Recursio III-b</i> . . . . .             | 243 |
| 70 | <i>Threnody for Dimitris Christoulas</i> — Genotype of <i>Recursio IV-a</i> . . . . .              | 244 |
| 71 | <i>Threnody for Dimitris Christoulas</i> — Genotype of <i>Recursio V</i> . . . . .                 | 245 |
| 72 | <i>Threnody for Dimitris Christoulas</i> — Genotype of <i>Recursio IV-b</i> . . . . .              | 246 |
| 73 | <i>Threnody for Dimitris Christoulas</i> — Genotype of <i>Recursio VII</i> . . . . .               | 247 |
| 74 | <i>Threnody for Dimitris Christoulas</i> — Genotype of <i>Recursio VIII</i> . . . . .              | 248 |
| 75 | <i>Threnody for Dimitris Christoulas</i> — Genotype of <i>Recursio IX</i> . . . . .                | 249 |
| 76 | <i>Threnody for Dimitris Christoulas</i> — Genotype of <i>Recursio X</i> . . . . .                 | 250 |
| 77 | <i>Threnody for Dimitris Christoulas</i> — Genotype of <i>Recursio I-b</i> . . . . .               | 251 |
| 78 | <i>Threnody for Dimitris Christoulas</i> — Genotype of <i>Recursio XI</i> . . . . .                | 252 |
| 79 | <i>Threnody for Dimitris Christoulas</i> — Genotype of <i>Recursio XII</i> . . . . .               | 254 |
| 80 | <i>Threnody for Dimitris Christoulas</i> — Genotype of <i>Recursio XIII-a to XIII-d6</i> . . . . . | 255 |
| 81 | <i>Ada + Babbage – Capricci</i> — Genotype of <i>Capriccio I</i> . . . . .                         | 262 |
| 82 | <i>Ada + Babbage – Capricci</i> — Genotype of <i>Capriccio II</i> . . . . .                        | 264 |
| 83 | <i>Ada + Babbage – Capricci</i> — Genotype of <i>Capriccio III</i> . . . . .                       | 265 |
| 84 | <i>Ada + Babbage – Capricci</i> — Genotype of <i>Capriccio IV</i> . . . . .                        | 267 |
| 85 | <i>Ada + Babbage – Capricci</i> — Genotype of <i>Capriccio V</i> . . . . .                         | 268 |
| 86 | <i>Ada + Babbage – Capricci</i> — Genotype of <i>Capriccio VI</i> . . . . .                        | 269 |
| 87 | <i>Ada + Babbage – Capricci</i> — Genotype of <i>Capriccio VII</i> . . . . .                       | 271 |
| 88 | <i>Ada + Babbage – Capricci</i> — Genotype of <i>Capriccio VIII</i> . . . . .                      | 272 |
| 89 | <i>Ada + Babbage – Capricci</i> — Genotype of <i>Capriccio IX</i> . . . . .                        | 274 |
| 90 | <i>Ada + Babbage – Capricci</i> — Genotype of <i>Capriccio X</i> . . . . .                         | 276 |
| 91 | <i>Ada + Babbage – Capricci</i> — Genotype of <i>Capriccio XI</i> . . . . .                        | 277 |
| 92 | <i>Ada + Babbage – Capricci</i> — Genotype of <i>Capriccio XII</i> . . . . .                       | 279 |
| 93 | <i>Ada + Babbage – Capricci</i> — Genotype of <i>Capriccio XIII</i> . . . . .                      | 281 |
| 94 | <i>Ada + Babbage – Capricci</i> — Genotype of <i>Capriccio XIV</i> . . . . .                       | 283 |
| 95 | <i>Ada + Babbage – Capricci</i> — Genotype of <i>Capriccio XV</i> . . . . .                        | 284 |
| 96 | <i>Ada + Babbage – Capricci</i> — Genotype of <i>Capriccio XVI</i> . . . . .                       | 286 |
| 97 | <i>Microcontrapunctus</i> — Csound instrument to synthesize microsounds . . . . .                  | 290 |
| 98 | Decoded genotype for synthesis of microsounds . . . . .  | 304 |
| 99 | <i>Rudepoema na penumbra</i> — SuperCollider receiving GenoMus data via OSC . . . . .              | 343 |



## Introduction

“

### Grammars for Music?

Then there is music. This is a domain that you might suppose, on first thought, would lend itself admirably to being codified in. An ATN-grammar,<sup>1</sup> or some such program. [...] There is no reference to things "out there" in the sounds of music; there is just pure syntax. [...] But wait. Something is wrong in this analysis. Why is some music so much deeper and more beautiful than other music? It is because form, in music, is expressive—expressive to some strange subconscious regions of our minds. [...] No, great music will not come out of such an easy formalism as an ATN-grammar. [...] the grammar will be defining not just musical structures, but the entire structures of the mind of a beholder. The "grammar" will be a full grammar of thought—not just a grammar of music.

Douglas R. Hofstadter [66, p. 626]

The individual impulse to create new music as an expressive necessity is difficult to trace. I consider myself omnivorous in terms of genres, styles, or cultures; I equally enjoy mainstream hits and extreme experimental works, provided I perceive authenticity in them (a completely subjective sensation, nonetheless). However, when it comes to composing my own music, I feel that my interests are focused on certain aspects far from the central area of the socially shared concept of *what music is*.

In my case, I can explain what interests me when composing music. While my motivations may differ significantly from the typical ones, they do resonate strongly with those of

<sup>1</sup>Abbreviation for *augmented transition networks*, a concept that expands upon the idea of RTR (recursive transition networks), related to the set of rules studied by generative grammar. ATNs are networks composed of RTRs that refer to each other, which means their level of complexity and self-reference can become enormously intricate.

many other past and present figures, so I perceive myself as part of a well-defined stream, whose musical orientation traces a trajectory that touches upon the following fundamental questions.

Primarily, the exploration begins by attempting to deeply understand the underlying mechanisms of musical composition for each style and how these devices acquire certain almost semantic connotations in the co-evolution of musical creation and reception. How do the bidirectional connections between emotions and musical language materialize? Are there a priori universal elements in this hidden semantics of sound, or is it all an arbitrary cultural product?

Another key aspect is the fascination with stretching these mechanisms to push the style towards unusual sonic territories, with these underlying questions: How far can music be taken without breaking it? What emotional triggers can be reached?

Finally, with the advent of computation, these inquiries escalate to a new level where experimentation can be taken much further. New questions arise: To what extent can the process of musical creation be automated without losing the essence of artistic communication? How does the potential massification of automatic creative artifacts affect the evolution of culture and composition techniques? How will artificial intelligence affect the very concept of music in the future?

## Programming is (meta)composing

Artistic creation, like almost any aspect of human nature, is a phenomenon with strong coupling and feedback. What conditions must something fulfill to fit into the definition of music? This definition is in constant redefinition, changing after each new item enters the set. If we consider the entire spectrum of human musical productions, focusing on the most adventurous and experimental approaches, we might conclude that anything can potentially be considered music, because context is much more important than the sounds themselves. It is inevitable to follow in the footsteps of Formaggio [55], and conclude that *music is whatever someone considers to be music*:

L'arte è tutto ciò che gli uomini chiamano arte. Questa non è, come qualcuno potrebbe credere, una semplice battuta d'entrata, ma, piuttosto, forse, l'unica definizione accettabile e verificabile del concetto di arte.<sup>2</sup>

---

<sup>2</sup>Art is everything that people call art. This is not, as someone might believe, a simple opening remark, but rather, perhaps, the only acceptable and verifiable definition of the concept of art. (Author's translation)

That tautological and recursive definition is not useful in computational terms. However, this thought is crucial to understand why the idea of considering a fitness function<sup>3</sup> of a musical work without incorporating the entire background of its audience into the equation is indeed naive.

Projecting this vision into a distant future, we might have to further expand the framework and accept that music will be anything that *some entity* considers music. And that music may encompass a range of frequencies and temporal scales inaccessible to human perception, or contain information patterns beyond human cognition. And perhaps, by forging a new connection with very old ideas, the truly relevant aspect is the synergy between information and time.

For the reader of this doctoral thesis, these underlying matters can help situate the aesthetic foundations of this research: I am ultimately interested in discovering the very foundations of music and how technology can delve into them, even creating new groundings and paradigm shifts. And, of course, what emotions can be stirred within us by these new developments.

The task of designing algorithms that generate music, or any other type of artistic process, involves a multitude of decisions that carry an aesthetic vision. Every software, no matter how open and flexible it may be, leaves an imprint on the outputs it produces. The design of a framework like this is an actual creative act. This notion has been prevalent from the beginning: programming is not composing, but it is about establishing frameworks and spaces in which potential music exists. In this sense, *programming becomes an act of metacomposition*. The unconscious decisions and assumptions made during prototyping stem from aesthetic preconceptions of the music imagined by the programmer, with far-reaching consequences.

Our perception of music depends on the interplay between temporal proportions, frequencies, timbres, etc., whose patterns can be discerned through listening. These patterns occur at very different temporal orders of magnitude. Figure 1 shows a hierarchy of musical structures that can be analyzed and automated, spanning from the individual components of each *sound atom* to the entire work, and beyond, considering groups of compositions that can constitute a large corpus. Developing composition systems is a complex task, particularly when aiming to encapsulate within their internal representation the multidimensional reality of any musical object. From this starting point, my research aims to delve into this thought of Roads [123, p. 54]:

---

<sup>3</sup>Especially in the context of genetic algorithms, an operation that accepts a potential solution as input and returns a measurement of the solution's quality regarding the addressed problem.

Ultimately, the question is not whether music conforms to the structure of formal grammars, but rather whether particular formal grammars can be designed which be useful representations of certain compositions. Grammars with embedded procedures can be powerful descriptive and expressive tools, but certainly, formal languages will evolve, and in general, knowledge representations will grow more elegant.

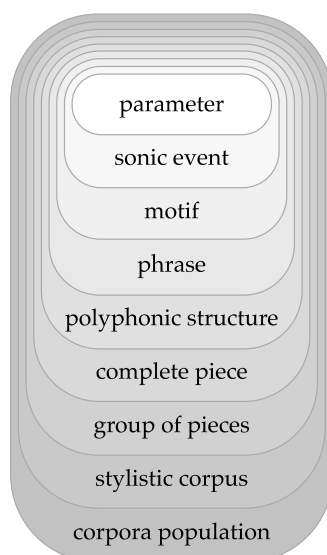


Figure 1: Automation levels for symbolic music composition

## Scope of the study

Among the multitude of studies about automatic composition, the particularity of this research lies in its attempt to combine within a single model two different areas: on the one hand, this investigation continues the old tradition of automatic music composition systems, oriented toward the practical use of the composer, but at the same time seeks to achieve a representation system that can seamlessly integrate with the newest machine learning mechanisms. The challenge of this project is to combine efficiency in the abstract compression of symbolic musical information with an interface expressive enough to be used as a simple programming language. Essentially, this research aims to find the intersection between these two paradigms and explore how they can learn from each other.

One of the primary focuses of the project is to create a practical tool for augmented creativity that enables composers to *bypass their own biases and expand their expressive palette*.

The purpose is to achieve greater autonomy of the machine in making proposals and maximize the openness of the latent music space.<sup>4</sup> The aim is also to facilitate the project's future phases, centered around autonomous and self-evolving aesthetic evaluation, the creation of a database of results that allows the system to learn from itself, as well as integration with more complex deep learning systems. A robust architecture for musical representation is essential to harness computational power for the recursive and open grammars on which this research is based.

Another fundamental characteristic that has shaped the entire project is the dual representation of symbolic music: one from a procedural metalevel and the other as abstract numerical data with the simplest structure. At the heart of this research lies the concept of *modeling the composer's creative process rather than the score*. Prior to embarking on any programming tasks, I explored numerous potential approaches to the challenge of establishing a musical metalanguage that could be as precise as the score it references while also reflecting the processes that underlie it. These efforts leaned toward the paradigm of functional programming.<sup>5</sup>

The subsequent step involved selecting the appropriate computational tools to create the initial prototypes of the ideas that had been outlined on paper. Practical application was another initial requirement. The author of this work is a composer with significant experience in composing using technological means but with far fewer skills in computer science. From this standpoint, it was believed that the most valuable contributions would arise from the experience of using these tools. Indeed, the development methodology has been based on an iterative cycle of programming prototypes and composing pieces with them, revealing the strengths and weaknesses of each system version, ultimately shaping its development.

Many modern approaches to artificial intelligence applied to the automatic composition of music are modeled using scores as their data source. The effectiveness of neural nets favors this kind of experiments. In my case, I generate data from randomness, which

---

<sup>4</sup>In the context of audio signal processing or machine learning applied to music, it refers to an abstract or multidimensional representation where musical features are encoded in a non-explicit but meaningful manner for certain algorithms or models. It represents the set of all possible numerical vectors capturing fundamental musical characteristics such as tonality, rhythm, harmony, or structure in a compressed and abstract way, allowing for efficient manipulation, exploration, or generation of music.

<sup>5</sup>Paradigm based on the definition of functions whose behavior is closed and independent of the program in which they are embedded. The result of their evaluation does not depend on previous states of the main program, and their output does not create side effects in it. A program written using this technique can practically be considered as a set of abstract mathematical functions that operate on a specific number and type of variables and produce a well-defined output. An advantage of this concept is that, once the functions are defined, it is relatively straightforward to port them to any language that supports functional programming.



is selected and refined through human supervision or following a fitness function that assesses specific characteristics. Furthermore, my focus is not solely on the score but primarily on the processes through which the musical text can be deconstructed. This sort of reverse engineering involves applying functional programming and a library of procedures that enable the metaprogramming of these scores. Due to its relative ease of implementation, a custom genetic algorithm<sup>6</sup> is employed for selecting results. However, despite the widespread influence of bioinspiration across various subprocesses of the model I present, genetic algorithms are just one possible approach to handling the data. The aspiration is to establish a representation system basic enough to adapt to almost any other machine learning algorithm, such as neural networks with different architectures or completely newer future paradigms.

I considered the study of McCormack [96] regarding the difficulties of tuning evolutionary algorithms, as well as the comprehensive surveys that study a large number of similar projects. In the usual taxonomy of AI methods applied to music composition, my model can be classified in the domain of *evolutionary algorithms applied to grammars*. Fernández and Vico’s exhaustive survey [53] concludes with some remarks regarding the important topic of encoding data to work with evo-devo<sup>7</sup> systems. After identifying the problem of scalability, due to the enormous amount of information that a musical work can contain, they consider that many systems have addressed this bottleneck using *indirect encodings* that compress musical information by encapsulating the *list of instructions* needed to recreate a piece, but these encoding methods need to be highly improved and optimized. Otherwise, a good toy model can become intractable when scaled to handle and produce real pieces of music.

My research is devoted to this issue and looks for strategies to compress and simplify as much as possible the procedural information while maintaining an interchangeable readable counterpart of this information. I conceive this indirect encoding of music as the core of my framework: *what must be learned by an AI engine<sup>8</sup> is precisely this abstract representation of the composition techniques used in the creative process.*

---

<sup>6</sup>The different types of genetic algorithms share a fundamental concept: formulating a problem in such a way that its possible solutions can be represented with some type of *chromosome* of data, and from there applying a cycle of selection, crossover, and mutation.

<sup>7</sup>In biology, an informal way to refer to *evolutionary development*, an area of study that seeks to infer the developmental stages of an organism. In the context of genetic algorithms, it relates to generative systems that exhibit an evolution from initial conditions to reach a specific state.

<sup>8</sup>Beyond metaprogramming itself, some researchers focus on the even more fundamental plane of artificial *metalearning* [141]. With the achievement of algorithms that learn to learn, multiple alternative systems of data representation could become important in machine-driven scientific discovery.

With all this in mind, I outlined all the desired characteristics that my model must fulfill. All these features can be condensed into these eight facets:

**Modularity:** *based on a very simple syntax.* This should allow the recombination of processes in the most open manner possible, expanding the search space and promoting new and unexpected recombination of subprocesses.

**Compactness:** *maximal compression of procedural information.* In the encoded counterpart of the processes, a representation system must be possible that reduces complex functional structures to simple sequences of values.

**Isomorphism:** *same structure for all encoded entities.* The representation system must allow associating the same type of basic data structure to relate inputs and outputs, that is, programs and results of their evaluation.

**Extensibility:** *subsets and supersets of the grammar easily handled.* The architecture must be so simple that it allows adding (or limiting) functions with new processes smoothly.

**Readability:** *both abstract and human-readable formats interchangeable.* A coding and decoding system is needed to work simultaneously with a programming language and with numerical vectors, and the transformation between both needs to be reliable, secure, and efficient in terms of information compression.

**Repeatability:** *same initial conditions always generate the same output.* This is an evident condition, but it will require delicate design decisions to not limit modularity and extensibility.

**Self-referenceable:** *support for on internal autoreferences.* This is essential for representing many basic formal structures based on repetition and variation, and also for many processes in algorithmic music that require recursion.

**Versatility:** *applicable to other contexts and domains.* The level of abstraction should make it transferable to other fields. As a generative system that handles simultaneous sequences of events in time, with arbitrarily complex information patterns, it can be applied to other fields such as video synthesis, remote lighting control, motion control, etc. Multimedia art is one of the working horizons of this research.

GenoMus is the name of the tool I have developed as a practical realization of this conceptual approach. The name encapsulates the expression *genome of music* and emphasizes the goal of decoding the underlying processes of musical composition in a way that can be sequenced, akin to applying analogs to genetic engineering: enabling us to learn, analyze, and synthesize new music based on previous knowledge.

## Hypothesis and research objectives

In summary, this research aims to gain insight into this hypothesis:

**It is possible to create a procedural representation of music, both as a highly abstract and compressed format optimized for machine learning and as a simple human-readable grammar. This representation can be valuable as the foundation for different systems for music creation and analysis.**

To test this concept, and given the goal of approaching the problem by combining theoretical formalization with real artistic usage, it has been necessary to accomplish a series of objectives.

**O1 • Design a coding system for symbolic musical processes and generated musical pieces of the utmost simplicity.**

The initial goal was reducing both representations of underlying compositional techniques and symbolic music outputs to a one-dimensional vector of normalized values. This simple encoding had to be independent of the complexity of the processes or the generated piece. A simple note or a long work with many voices, notes, and parameters had to be represented as a single decodable stream of floats.

Although this representation system can be used to encode conventional music, its design must emphasize the aesthetic framework of experimental creation. Therefore, this encoding must be capable of representing arbitrarily complex combinations of generative processes, recursive algorithms, etc.

**O2 • Create a decoded counterpart of processes and music that is readable and functions in practice as a functional programming language with equally reduced syntax.**

Equally important to the pure numerical representation, there must be a readable form for both the declaration and transformation of musical events, as well as the resulting symbolic representation of a music score. The readable form should be editable and executable as an extremely simple programming language.

- O3 • Implement a tool for technical testing and real artistic use of the model, allowing visualization and playback of generated music, as well as enabling export to various standard formats for integration into diverse music production environments.**

As an initial prerequisite of the research, I aim to enable close collaboration and intervention of the human user with the generative algorithm. The starting point for the development of a composition can either be a set of ideas proposed by the machine or a manually coded fragment for transformation. In either case, the interface should facilitate this active collaboration throughout the process.

This tool should be manageable for both expert users with specific knowledge of musical composition and those without any prior knowledge. In the latter case, the user interface should be straightforward enough to guide the process of evolving and transforming musical material with basic operations that are capable of capturing personal preferences.

Finally, the produced fragments should be easily exportable to other musical production environments, such as virtual instruments, music notation editors, sound synthesis software, other encoding formats for music scores, etc.

- O4 • Compose and present musical works created at various stages of the development process in a standardized environment with a non-specific audience.**

The use of GenoMus in human-machine collaborative creation should expand the user's expressive palette. This form of computer-assisted or augmented creativity must, in some way, bridge personal aesthetic biases and technical limitations through exposure to a wide array of diverse materials suggested by the system.

- O5 • Publish the necessary documentation for the model to be used, adapted, and extended by new users and incorporated as a resource in machine learning pipelines.**

The true potential of this proof of concept can only be realized through a community of users who interact with the tool from various stylistic, aesthetic, and technical approaches. Gaining access to individuals (or other algorithms) with varying degrees of musical expertise is, therefore, an essential objective for evaluating the validity of my proposal.

The ultimate goal of this research is not to create a standalone and closed tool. Beyond its direct use restricted to the specialized field of music creation, the design of this multiple representation system is aimed at its integration into more sophisticated present and future multimodal systems, capable of processing symbolic information.

In the current context of rapidly evolving deep learning architectures, I believe that it makes the most sense to contribute my musical expertise towards alternative formalizations of musical representation, optimized for these machine learning models, thereby enriching their internal representations and autonomous creative capabilities. Most importantly, these contributions should facilitate interaction with human interlocutors who can further develop their abilities and expand their artistic imagination in new directions.

## Interactive experimental setup

During the development iterations of the core algorithm, numerous technical alternatives have been tested to create an environment that combines simplicity in integrating new functionalities with the ability to see and hear the results of the generative system. The experimental setup that has proven to be the most practical, and has ultimately been used to create a highly responsive and flexible user interface, combines three key elements:

- **Max patch as interface**

Max<sup>9</sup> (formerly MaxMSP) is a widely used software for sound processing and multimedia art. It is a *de facto* standard in work oriented towards experimental music and offers an extensive range of objects and libraries that provide astounding versatility for interconnecting software and hardware elements.

The user interface programmed in Max facilitates the execution of all requests to the core code in a highly simplified manner. It provides the ability to visualize and edit the code generated for each musical piece, as well as to represent the results of internal operations in various ways, translating them into interactive real-time scores, displaying both the data and metadata of musical pieces, graphically visualizing encoded data, and more.

---

<sup>9</sup>Commercial software developed and distributed by Cycling '74. It can be found at <https://cycling74.com/products/max>.

- **Core algorithm contained in a Node.js JavaScript file**

The initial iterations of the core algorithm were programmed in plain JavaScript. Communication with Max could only be achieved in a complicated manner, and due to the characteristics of the Max implementation, the patch would freeze during the execution of a function call. Although this wasn't problematic at this stage of development, it would have been an issue when it came to seamlessly integrating it with other Max objects, especially in the case of live-coding,<sup>10</sup> where audio, MIDI, and code writing need to run flawlessly in parallel without interruptions.

Starting with version 8, released in 2018, Max integrated a new bridge object with Node.js, enabling the execution and data communication between patches and JavaScript using the ECMA 6 standard. This marked a significant boost for the use of JavaScript in the final prototype. Apart from being much faster, benefiting from the functional possibilities and syntactical improvements of the new standard, and being able to call external libraries, since then, internal processes of the core algorithm could run without affecting the host patch's performance.

- **Package bach for real-time interaction with generated music**

The Max extension provided by the `bach`<sup>11</sup> package greatly enhances the monitoring and post-production possibilities for music algorithmically generated. In addition to being able to visualize and execute real-time modifications, hosting the results of operations within the Max environment allows for the use of generated sequences in many different use cases, such as sound synthesis, real-time effects manipulation, and data remapping for multi-channel MIDI output or raw data through OSC,<sup>12</sup> among other possibilities.

The `bach` package is designed to streamline the display of musical notation in Max. It also includes an extensive library of functions for algorithmic composition and analysis. I have used it solely for visualization, playback, and score export. Once the outputs of the prototype are incorporated into a `bach.roll` object, it becomes possible to continue adding new layers of manipulation with it using the numerous features of this powerful tool.

---

<sup>10</sup>Live-coding in music refers to the practice of writing and modifying computer code in real-time to generate music. It's a form of performance where the musician, often referred to as a coder or live coder, uses programming languages to script and manipulate sounds live before an audience. The improvised code typically controls the synthesis, processing, and arrangement of sounds. Live-coding can be visually engaging too, as the audience may see the direct translation of code into music.

<sup>11</sup>Package designed by Agostini and Ghisi [3], available at <https://www.bachproject.net>; `bach` is a recursive acronym for *bach automated composer's helper*.

<sup>12</sup>Open Sound Control. Due to its flexibility and modern architecture, it is a good alternative to MIDI.

Figure 2 illustrates the relationships between the internal processes of the core algorithm, the user interface, and the transmission of the generated music to external applications. While the generative processes may take some time, interactions typically occur within milliseconds, and the most complex processes take at most a few seconds to complete. Consequently, it is an environment that operates almost in real time.

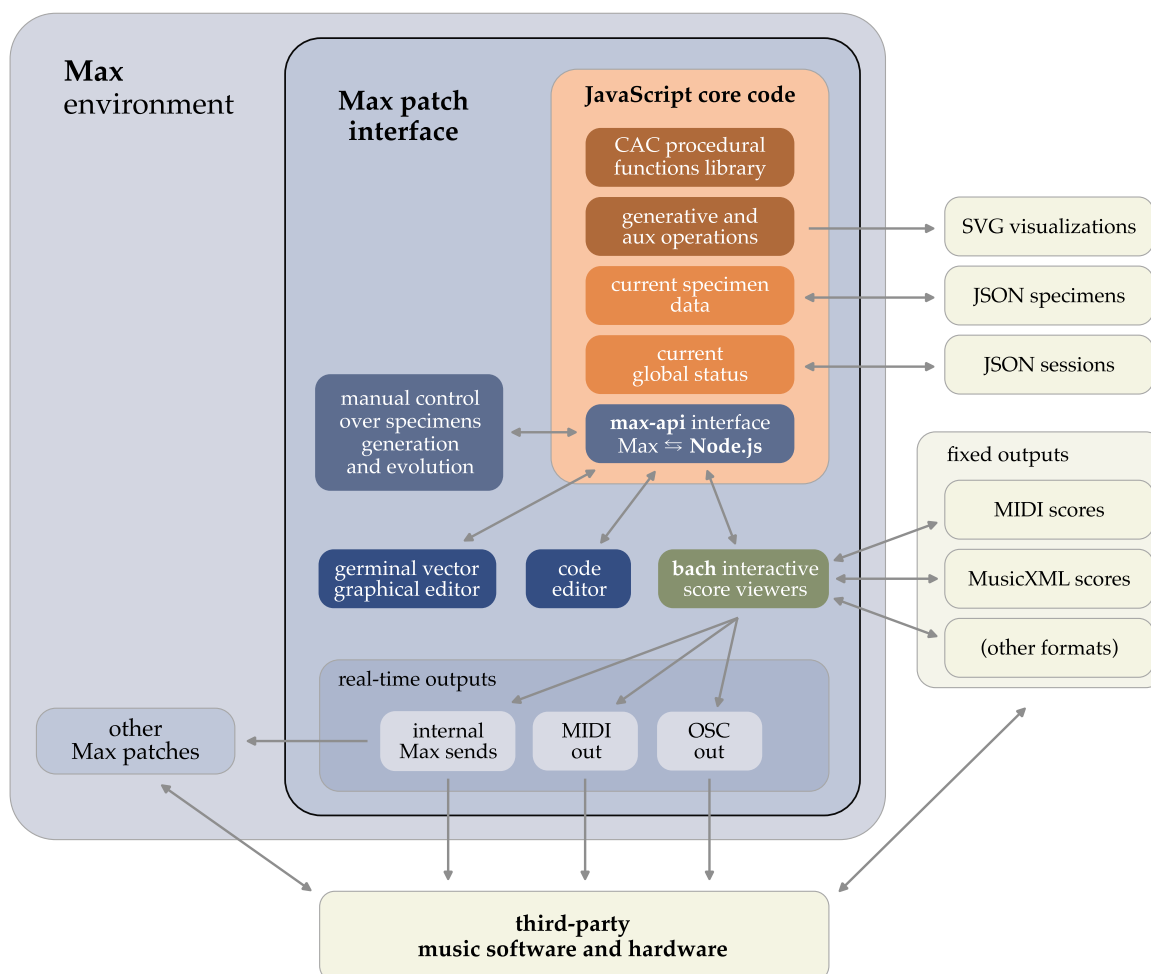


Figure 2: Scheme of development and experimental setup. The core code of the model is represented in light orange, which runs within a GenoMus patch environment in Max, depicted in blue. This environment contains various modules for communication and interaction with the user interface. The entire Max environment is represented in a lighter blue, which can run multiple patches in parallel that can communicate with GenoMus. Outside this environment, third-party software and data output to files of different formats.

## What this research is not about

My work, while situated within the active field of applying artificial intelligence techniques to symbolic music creation, has certain peculiarities that distinguish it from the standard research in this area, and are worth noting.

The proposed model is not aligned with most current efforts in creating systems capable of replicating music with well-defined styles. The enormous commercial potential of generative art tools encourages most current studies to focus on assimilating vast amounts of musical repertoire to achieve original creations that are well-suited to the most successful existing genres and subgenres in the industry.<sup>13</sup>

This is not the target of this research. My proposal lies on the almost opposite side of the spectrum of music creation. I do not aim to replicate music whose structural foundations and rules have already been extensively studied and systematized by modern musicology, although my model can equally represent it. By design, the goal is to facilitate the exploration of new combinations of elementary techniques that can give rise to highly original and innovative music. However, it is also possible to steer the evolution of results towards more common patterns.

This research is rooted in the tradition of composers who, from the early days of computing, began exploring the possibilities of computation to expand and enhance their arsenal of expressive resources. As a new twist in this tradition, my model is oriented towards achieving a seamless integration of diverse traditional methods of automatic composition—such as the creation of grammars, various systems of algorithmic composition, or expert system programming—but optimized to harness the capabilities of modern machine learning methods.

Therefore, this proposal is not simply another specific computer-assisted composition system. It operates on a more fundamental level of abstraction, allowing each user to configure the compositional procedures they wish to employ, whether deliberately limited or highly eclectic, in a collaborative process with the tool. It would be more accurate to define GenoMus as a draft of an automatic composition metasystem.

Despite the code examples, what I present is not a programming language in the strict sense. While in the current implementation of the proposal, musical pieces are programmed using a restricted subset of JavaScript functions, the key aspect is the type of symbolic and encoded data architecture, portable to any other language that may be

---

<sup>13</sup>A good example of the emerging trends in this multimodal generative approach is MusicLM [1], which allows for the generation of audio from natural language prompts.



more appropriate or efficient in the future. The syntax I use is close to the functional paradigm, but it is *impure*, as there are a series of indispensable side effects for certain basic functions of the overall algorithm. Although the user interface allows for programming pieces from scratch, the method of creation and transformation is preferably done through metaprogramming techniques that autonomously generate code.

The simplest symbolic output format to handle in GenoMus is the MIDI file. However, the representation of music is not limited to the score but accommodates the nuances and deviations typical of a performance. This is particularly important when the outcome involves sound synthesis, virtual instruments, or live manipulation of other software. GenoMus does not generate traditional scores; instead, it produces relatively complex structures that, if intended for performance by human musicians, require adaptation to a standard notation, often omitting many small nuances for readability.

Last but not least, it's worth noting that, much like with manually composed experimental music, the evaluation of results is highly subjective and doesn't lend itself well to typical tests seen in many automatic composition studies. As an alternative validation, I offer a collection of pieces that have been composed in part or entirely using this model. These pieces have been successfully premiered and received positively by a general audience in standard contexts of artistic performances.

## Thesis structure and reading recommendations

This doctoral thesis begins with this very **introduction**, which delimits the personal motivations and the scientific scope of the research. Faced with such a wide array of approaches to automatic music composition and representation systems, the technical and conceptual elements in which contributions are sought are explicitly defined. After formulating this approach as a formal **hypothesis**, a series of specific **objectives** are detailed, which must be achieved to provide at least a limited response to the central question. Later, the tools and experimental setup used for prototyping and testing the model are described.

Once the research field has been defined, Chapter 1 briefly analyzes the **history and state of the art in computer-assisted composition and artificial musical creativity**. Starting from some general but important considerations to maintain perspective on the aesthetic and philosophical implications of machine-generated art, a brief historical overview of the relationships between computer science and musical composition is provided. Some of

the key topics addressed include bio-inspired composition strategies, musical grammars, metaprogramming, and current trends in deep learning applied to music generation.

The abstract formalization of the model's **conceptual framework**—an updated version of that presented in a previous article [87]—is presented in Chapter 2. It begins by specifying how the biological metaphor exists behind the model. Then, the necessary elements that make up the system are analyzed. Some of the key concepts addressed include encoding and decoding mechanisms and the *germinal vector* as a determinant of a decision tree designed to enable the retrotranscription mechanism, which is central to the model. In particular, the constructs referred to as *genotypes* and *phenotypes*, ubiquitous in the literature of bioinspired algorithms, are precisely defined.

Putting into practice everything previously analyzed, the following chapters demonstrate the practical **implementation of GenoMus**, showing the mechanisms required to convert the theoretical model into a functional system with a manageable user interface.

Chapter 3 shows the architecture of the necessary **data structures**, such as the *specimen*, the *genotype function*, the mapping strategies for parametric data, etc.

In turn, Chapter 4 reviews all the main **types of genotype functions**, such as identity functions, those that control formal structures, and random processes, among others. Important topics for algorithmic compositions, such as autoreference and generative processes, are also presented. This chapter contains minimal examples that illustrate how musical fragments are constructed at various levels of complexity.

The important topic of **genotype and phenotype encoding and decoding** is addressed in Chapter 5. In particular, the mechanism of *retrotranscription* is unraveled in detail, as a fundamental piece of the model. A method for graphically representing long numerical structures is also proposed to facilitate understanding, debugging, and traceability of the processes occurring at the most abstract level.

Building upon these prior foundations, Chapter 6 analyzes the implementation of the system's core algorithms: those enabling **genotype metaprogramming and phenotype rendering**.

Chapter 7 escalates to a higher level of complexity to introduce the entire mechanism as mentioned earlier within an **evolutionary algorithm that creates, selects, and transforms populations of specimens**, both with human supervision and unsupervised.

Concluding this central block of work, Chapter 8 demonstrates the utility of the model as a **tool for analysis and incorporation of new processes** into the function library.

The most relevant **results** are compiled in Chapter 9. It discusses the iterative development process of the system, explaining how after each real application of the system to an actual artistic production, the needs, weaknesses, and potential of the main code have been identified. The text does not delve deeply into either the software or the musical compositions created. For those interested, technical and artistic details of the composition processes of these works can be found in the Appendices.

In contrast, Chapter 10 adopts a broader perspective, drawing **conclusions** based on the system's capabilities as demonstrated in artistic application experiments. The text encompasses specific aspects, both conceptual and practical, that introduce innovation to the field of computer-assisted composition. Chapter 11 is the Spanish version of the previous one.

Appendix A concisely displays the **user interface** of the GenoMus implementation in Max. It shows the main operating panels of the software and its most important functionalities.

Appendix B is aimed at readers with a more specialized interest in music and provides details of the development of all **artistic projects** that have marked the iterative evolution of GenoMus. It is highly recommended to complement the reading of this visual documentation with the audio and videos from the corresponding links, as this auditory information is truly relevant as the tangible final product of this research.

To enhance the readability of code snippets, several conventions have been added to the usual JavaScript syntax highlighting in the listings and the main text:

- Genotype functions, such as **sConcatS**, which are the building blocks of the generated music, are printed in bold magenta.
- Core algorithm's auxiliary functions, like **indexDecGens**, appear in bold bright orange.
- In Appendix B, there are names of old functions from previous versions of the model that are no longer available. These deprecated functions appear in pale blue, like in **combineArrays**.
- Despite their functional style, genotypes are not pure functional expressions and require global variables, which are printed in red, such as **defaultEvent**.
- For the remaining keywords within the core code, they are written simply as black monospace, such as the label `midipitchF`.
- Finally, executable expressions like **s(v(e(nRnd()), mRnd()), aRnd(), iRnd()))** appear with a light gray background, just like the code listings.

## Source code and software

This document is complemented by numerous musical examples, sheet music, code fragments, graphics, and videos, hosted on the website <https://genomus.dev>, which are not included here to avoid excessive length of the core academic text. It is strongly recommended to use the provided links to access this additional material, as it demonstrates the practical outcome of this model, allowing a complete understanding of the theoretical-practical profile of this project. This website, as the official project platform, will guarantee the availability of these additional materials and the continuity of the corresponding URLs.

The source code for GenoMus, as well as the Max patch that enables user interaction with the generated music, can be found at <https://genomus.dev/thesis>. The program version available at this link corresponds to the one documented in this thesis and will not be altered to maintain consistency with this document.

The latest version of GenoMus can be found at <https://genomus.dev/download>, along with some instructions for its execution within the Max environment. Having previously engaged with the tool and the code from the examples is perhaps the best preparation before reading this thesis. The reproduction of some of the examples from the text and the ability to test some variations will enable the reader to quickly grasp the concepts presented.

The requirements to run GenoMus are as follows:

- Install **Max 8** (the runtime version that remains operational after the 30-day trial period is sufficient to run the program).
- Install the **bach** package from within Max.

The GenoMus folder contains the following items:

- **GenoMus\_1-00.js**, the source code referred to in this text.
- **GenoMus\_1-00.maxpat**, the user interface executable in Max.
- Additionally, the folders **sessions**, **specimens**, **visualizations**, and **midi** are used by the script to store the results that a user may wish to save.

- Analiza los elementos principales  
- Se hace una idea global de la forma.  
- Analiza los ritmos principales.  
Y en el análisis global abunda tener un  
hacer binomio. Muchas estrofas a poder  
la generación más a más se sigue. VIT-  
hacer un gramática de lo global.  
Como a que de, escribir tal gramática?

de modelaje y no demerita en lo  
detalles alternados de programación.  
Debe ser flexible extensible.

Hay que abandonar un enfoque demasiado ingenuo y ver que  
menos de elementos  
no garantiza que un  
pueda escribir procesos de reducción de



# Background

“

Quiero construir la música como está construido un árbol”, era una de las afirmaciones recurrentes de Guerrero. Lo que más le fascinaba era la idea de hacer respirar a la música la rugosidad propia de la materia viva, la complejidad de la pulsación orgánica. Su objetivo era el de crear una música perfectamente estructurada y al mismo tiempo perfectamente natural e inteligible; hacer del sonido un elemento en estrecha simbiosis con la materia, con sus formas y con sus leyes dinámicas. Extender los principios fractales a la música se convertía para él en la manera de bucear en la materia única que une el sonido y la sustancia profunda del mundo. [...] “Si miras al campo estás viendo música, las mismas leyes que lo rigen todo”. En la naturaleza como en la música Guerrero sentía la respiración de un único orden cuya raíz era matemática.<sup>14</sup>

Stefano Russomanno [129, p. 105]

Addressing the state of the art in any computer-assisted composition (CAC) research quickly becomes unwieldy due to the almost daily emergence of new technical approaches, artistic projects, and academic papers. A narrowly focused enumeration of currently dominant lines of work risks becoming obsolete quickly and losing perspective regarding the broader currents in contemporary musical creation. This chapter will attempt to provide

<sup>14</sup> “I want to build music the way a tree is built”, was one of Guerrero’s recurring statements. What fascinated him the most was the idea of infusing music with the roughness inherent in living matter, the complexity of organic pulsation. His goal was to create music that was perfectly structured while also being perfectly natural and intelligible; to make sound an element in close symbiosis with matter, its forms, and its dynamic laws. Extending fractal principles to music became his way of delving into the unique essence that binds sound and the profound substance of the world. [...] “If you look at the countryside, you’re seeing music, the same laws that govern everything.” In nature, as in music, Guerrero felt the breath of a single order rooted in mathematics. (Author’s translation)

some historical insight and clarify the main paradigms of recent decades, with emphasis on those directly related to the subject of this study. My research primarily aims to formalize the symbolic representation of music and engage in open stylistic exploration. Consequently, I will not dwell extensively on works focused on imitating well-defined genres; rather, the focus will be on projects that leverage machine learning to expand existing musical languages and discover new styles and techniques for organizing sound.

First and foremost, it should be noted that CAC can be situated on a spectrum ranging from purely theoretical approaches to practical application without underlying reflection or analysis. In the production of contemporary composers who employ algorithmic processes in their creative work, it is often impossible to separate the musical composition from the research behind it. On many occasions, it seems that a single piece of music must serve as *proof* of a creative hypothesis, making it challenging to assess the potential of that specific CAC technique, especially considering that we don't have more examples. The quality of these compositions is likely to be more influenced by the talent of the composer using them rather than the merits of the specific system being advocated.<sup>15</sup>

Meyer [99, p. 31] clearly distinguishes these two facets of musical creation: the one related to the configuration of the language and the one regarding its effective use:

The distinction between rules and strategies helps, I think, to clarify the concept of originality, as well as its correlative, creativity. For it suggests that two somewhat different sorts of originality need to be recognized. The first involves the invention of new rules. [...] The second sort of originality, on the level of strategy, does not involve changing the rules but discerning new strategies for realizing the rules.

In the practice of many experimental composers, the creation of systems and their application to the score often merge, which complicates the evaluation and critique of these musical products. Each new CAC system typically underpins its propositions with scientific arguments. There exists some confusion between scientific research criteria in any aspect of musical practice<sup>16</sup> and the application of tools from physics or mathematics to creative work. However, it is important not to lose sight of the fundamental difference between the ultimate objective of the scientific and artistic methods: while *science seeks to find the universal* that unifies seemingly different phenomena, *art looks for unique cases* within the various possibilities of each technique or aesthetic. In other words, science deals

<sup>15</sup>The history of the reception and application of Schönberg's serial method illustrates how systems proposed as alternatives to conventional methods become inherently linked to the works and composers. The task of promoting and defending each new musical technique entails the creation of a tradition and, in a way, a biased and directed perception of their true expressive possibilities. This conflicts with the universality aspirations that some of these systems aim to achieve.

<sup>16</sup>An encyclopedic work that employs logical, topological, and algebraic analysis in nearly all aspects of music, including the most intuitive ones like phrasing or rubato, can be found in Mazzola's series. [95, 92]

with the common that unifies the diverse, and art with the special that stands out among the common.<sup>17</sup> This thinking advises considering each new CAC technique as part of the creative world of each composer, rather than as new attempts to reach a definitive system that should be embraced by all. In the realm of art, knowledge of other artists' production primarily serves as a stimulus to personal creativity, regardless of the useful tools and ideas one may adopt.

## 1.1. Artificial creativity or creative artifice?

The arguments frequently revisited in any debate about the true potential of artificial intelligence become more pronounced when focusing on artificial creativity. A thorough examination of artificiality often leads to contradictions and the need to redefine concepts. The semantics of the term itself embody this paradox: on one hand, artificial is defined as *made by human hand or skill*, or *produced by human ingenuity*, implying that the artificial is inherently creative; but at the same time, we describe as artificial what is not natural, false, or even worse when referring to the artistic, artificially contrived.

This contradiction stems from opposing conceptions of the human essence: what is at stake is the very definition of humans, either as something natural —with all its consequences— or as a separate entity in some aspects. In other words, AI, and even more so artificial creativity, supports the argument that human intelligence is not above the physical world. Scientific progress has been eroding everything that humanity has kept as a privilege over the rest of the universe. The evidence that processes of extraordinary complexity can arise from astonishingly simple mechanisms is undermining the last stronghold of human superiority, reducing intelligence and consciousness themselves to a natural phenomenon. Wolfram [156], in a book that became very popular in the circles of algorithmic composition, stated:

Yet in Western thought, there is still a strong belief that there must be something fundamentally special about us. And nowadays the most common assumption is that it must have to do with the level of intelligence or complexity that we exhibit. But [...] the Principle of Computational Equivalence now makes the fairly dramatic statement that even in these ways there is nothing fundamentally special about us.

That's why just the possibility of the existence of artificial creative entities produces both suspicion and fascination. Present approaches to AI tend to be concentrated in specific areas: from modeling trivial behaviors to the automatic formulation of new mathematical

---

<sup>17</sup>This thought is taken from a live video lesson by Miller Puckette.

theorems, but the current trend is that multimodal large language models (LLMs) will rely on specific modules to enhance their capabilities in particular areas such as musical composition. The coalescence and synergy of hyperspecialized layers, coordinated by LLMs capable of having a good overall context in communication with the user, seem to mark the path toward true artificial creativity.

In conjunction with this, the modeling of human reasoning is exerting a significant influence on our understanding of what thought and consciousness truly entail. Some studies on brain functions [5] are more focused on analyzing typical information theory problems than on physiology. Understanding the emergence of neural processes in networks can be equally important as delving into the specific details of their biochemical functioning. This convergence between cybernetics and neurobiology is likely indicative that we are approaching a point where it will become evident that research across all AI fields, including artificial creativity, essentially involves inquiries into what defines us as humans.

## 1.2. The role of the machine in the evolution of style

The idea of finding a recipe that automatically composes unique masterpieces equally contains a contradiction at its core. The aesthetic value attributed to a musical piece is not solely explained by the score but greatly depends on the context of its creation and reception. Furthermore, there are brilliant compositions that formally exhibit identical stylistic characteristics to others that are mediocre, so a perfect model would require something more than an exhaustive analytical description of a specific musical language. Moreover, many key works from each historical period tend to reside on the edges of their systems, often scratching unforeseen relationships and stretching established logic. Frequently, the studies that best understand and explain a particular musical style appear once that system has lost its relevance.<sup>18</sup>

Accepting the idea that the relevance and uniqueness of a great composer's style often reside in details that elude descriptive analysis, how can CAC assume a truly creative role? A broad perspective on the evolution of style over the past centuries in Western music may elucidate this issue. In very general terms, we can distinguish several stages:

---

<sup>18</sup>The classical pedagogical treatises by Schönberg, which demonstrate such profound knowledge of tonality, were written by the composer after he had opened the doors to radical and systematic atonality. His famous *Harmonielehre* [138] dates back to 1922; a decade later, he presented his serial method for the first time in private.



- By the 18th century, the tonal system<sup>19</sup> had completely crystallized in Europe. Apart from certain stylistic peculiarities, it can be affirmed that all composers shared a common idiom, which often remained very stable throughout their entire production.
- During the 19th century, individualities tend to become increasingly marked. Composers aspire for their style to be recognizable, and although there are noticeable transformations in the language of each composer, individual styles remain relatively identifiable and remain well-rooted in the common practice of tonality.
- The crisis of the tonal system in the early 20th century marks a significant turning point. In physical terms, we entered a phase of turbulence. Multiple alternatives to tonality emerge, and various, even opposing, musical techniques rapidly succeed one another. During these decades, we begin to see composers embarking on artistic trajectories that demonstrate substantial transformations in their language. Simultaneously, composers gather into stylistic movements that are in continuous controversy.
- By the mid-20th century, the divergence of styles had become so extreme that, in many cases, each creator could virtually represent a style of their own. The postulation of personal methods started to become widespread.<sup>20</sup> Maconie [90] emphasizes that there is an interest in probing and reinventing all language mechanisms, both for analysis and synthesis. Much of the progressive music of the 20th century aspires to eliminate the human aspect to approach a musical and mental organization situated beyond the scope of ordinary pattern recognition and memory.
- During the 1950s and 60s, electronic music and the beginnings of digital audio had a significant impact on some composers. The brief engagement of Ligeti, for example, with electronic music, led to profound changes in his approach to sound and texture [78]. Analog writing styles like spectralism and stochastic music began to rely on technological innovations as tools.
- In recent years, the rise of interactivity, the enormous impact of digitization, and the ubiquity of computers are redefining many concepts about what musical creation is. Di Scipio [45] has conducted an extensive study on the consequences of information technologies in the new relationships established between creators and audiences. At this moment of saturation in the possibilities of compositional language, extremes

---

<sup>19</sup>Also known as *tonality*, it has been the predominant musical composition technique in Western music for centuries. It is based on a tonal center that dominates the hierarchy of tones, melodies, chords, and formal structures. Everything is structured through tonal functions that create tension and release.

<sup>20</sup>Messiaen [98] in France, Hindemith [62] in Germany, or Barce [31] in Spain, are examples of composers with their own composition method explicitly formulated.

of writing and instrumental resources are being explored. Another approach is the dialogue and re-signification of references to earlier styles, which Wilson [155] interestingly reviews. It is a moment in which musical language has already been taken to its extremes, and experimentation is occurring in other fields such as real-time sound transformation, multimedia interdisciplinarity, and the blending of styles, blurring the boundaries between popular and creative music. Enabling new haptic interfaces [74], based on imagery and movement [14], or even on brain activity [105].

There has been a substantial shift in the awareness of style: if previously musical grammar emerged from a slow collective process, shaped by the interaction between musicians and the audience, from the early 20th century onwards, we find that sometimes new musical systems are formulated *a priori*, and musical compositions are presented as demonstrations of their validity.

Simultaneously, there is a growing interest in the theorization of language, which taken to the extreme tends towards the point where each new work is the enunciation of a unique style; each musical piece becomes a new —and sometimes ephemeral— definition of music. There is a strong interest in understanding the internal mechanisms of each musical grammar. The work of many current composers takes place at a metamusical level: it is no longer so much about placing notes on the staff as it is about specifying the rules (or the absence of them) that will organize those sounds. This focus on processes directly leads to the use of information technologies. Therefore, consciously or unconsciously, CAC is a form of musical creation that directly impacts the mechanisms that shape aesthetics.

### 1.3. Trends in CAC

Many musicologists have claimed that the aesthetics and techniques of works from a historical period reflect the prevailing scientific theories of that time. Longair [80] has pointed out that the convergence in the early decades of the 20th century between the collapse of the tonal system and the scientific revolutions in physics, mathematics, and other fields does not appear coincidental. These reflections began to take place consciously.

Decades later, as automated computing became available in research centers, a certain sector of composers eagerly embraced each new mathematical or physical concept to explore it as a means of producing musical structures or acoustic signals.<sup>21</sup> The availability of personal computers exponentially accelerated this trend.

---

<sup>21</sup>In Spain, it is paradigmatic the interesting space of fruitful exchange between artists and scientists from various disciplines in the 60s and 70s, centered around the Centro de Cálculo de la Universidad de Madrid [81].

In the field of composition, various trends linked to techniques such as stochastic analysis, Markov chains [124], game theory, fractal geometry, grammar creation, different aspects of chaos theory, cellular automata, neural networks, Bayesian networks, etc., can be traced. The contemporary composer has an arsenal of tools available for organizing sound material. Many of these procedures involve significant computation, making computer assistance essential. This marks the beginning of what is referred to as *algorithmic music*.<sup>22</sup>

Koenig was one of the pioneers in exploring these possibilities, who, in addition to the practical use itself, highlighted deeper implications of computing in composition. In the description of his *Project 1*, Koenig discarded the label of CAC, preferring the expression “composition-theoretical investigation”. In the article [73] where he describes these procedures in the 60s, he shows his awareness that the task of programming has first-order stylistic causes and consequences:

The program is seen as a composer’s tool for reflecting the compositional process in music (but not only music) empirically, and for developing a personal theory of the composer’s process.

Various fundamental lines can be appreciated in these compositional applications of computing. Coexisting with analogical composition techniques, which remain fully relevant, a variety of diverse digital tools are now being added. Miranda [104] has compiled many of these approaches. In recent studies, Hernández-Oliván and Beltrán [58, 59] assert that, despite progress, in general, sufficient quality in the generation of musical material has not yet been achieved, and there is a lack of coherence in the macroform. Other methods to CAC have emphasized semantic analysis and emotion techniques [44, 130] as a control element for automatic musical flow in real-time.

Another variant of CAC is its application to human-machine co-improvisation. The most well-known project, hosted by IRCAM, is OMax. Its purpose is to analyze real-time MIDI or audio input to be able to engage in active dialogue improvisation with the performer. Its architecture has been continuously updated [43], and it has recently been reincarnated as the Somax 2 project [23, 54], becoming one of the leading programs in this field.

---

<sup>22</sup>However, even though the term sounds progressive and modern, strictly speaking, the conservative and traditional species counterpoint (a polyphonic composition technique based on rigid rules for voice leading still taught in academia) is indeed algorithmic music. This is evident when reading any manual on the subject.

## 1.4. Exploration of abstract mathematical processes

It is possible to clearly distinguish between techniques that transcribe the output of processes as discrete musical events, typically at the note level or temporal proportions, from those that perform sound synthesis. The application of these techniques can range from mere translation through linear applications and various mappings, to their use as conceptual frameworks supporting a more symbolic manipulation.

Preferred algorithms tend to be those that exhibit complexity and richness of internal subprocesses. Musicalization mostly occurs in decisions regarding how correspondences are established between the algorithm's output and its translation into scores or audio signals. The value of this music does not arise from any supposed intrinsic musicality of the algorithms but rather almost exclusively depends on the composer's talent to transform them. The work of Xenakis, as well as his writings [158], mark one of the paths to follow in the 20th century.

The case of *fractal music* is particularly significant concerning the use and misuse that has been made of mathematical devices in many cases. It is easy to find examples of music that self-justifies through the mythification of its self-similar properties, ignoring that the limitations of auditory perception and memory might render those patterns imperceptible. Mandelbrot, one of the main discoverers of fractal geometry, considers that certain deviations from standard geometric objects are implicit in works of art. It is significant that in his seminal work on the fractality of the natural world, he includes some examples of defective figures [91, p. 347]; due to some glitch in the algorithm programming that generated the graph, these erroneous images exhibit interesting deformations. When placed alongside geometrically perfect figures, they immediately give the impression that there is a creative personality making an altered and unique interpretation of reality.

These techniques have only been truly productive in the hands of composers for whom these procedures have proven inspiring, while at the same time, they have not lost sight of the laws governing human auditory perception. The work of Guerrero is a good example of compositions that, even though their structures are based on fractals, are carefully balanced considering real perceptual faculties without losing their strong, strict formal approach. This ultimately ensures that beyond the sonification of complex internal relationships, they remain essentially music. Besada [17] and Satué [133] are two composers who have also theorized on these topics from their practical expertise. Simultaneously, it's essential not to forget that, from a certain perspective, all traditional tonal music holds evident self-similarities in their formal structures and melodic-harmonic designs [125], not as a result of any intricate contrivance, but rather as the natural consequence of applying the

same principles on various temporal scales. From this point of view, fractal music is not something special, but quite the opposite: it is almost inevitable to find some kind of self-similarity in any musical piece.

The rise of cellular automata in music came later, and its own mystique is also different. Although the study of these dynamic processes began in the 1940s with von Neumann [152], their adaptation to computer simulation —involving manipulations with discrete variables, easily codifiable in very simple programs— led to their popularization starting in the 1970s, with Conway’s famous Game of Life. Its interest was revived in the 21st century with extensive subsequent studies by Wolfram [156], which have given rise to numerous applications in composition and sound synthesis.<sup>23</sup>

The interest of musicians in these processes focuses on the fascinating capacity for self-organization that can be unfolded from very simple principles, which has been condensed under the label of generative music.<sup>24</sup> If we place perceived musical information within a virtuous area situated between overly obvious order and excessive complexity, composers’ interest in those processes exhibiting a balanced mixture of randomness and determination, in which sub-processes can be recognized at various scales, becomes understandable.<sup>25</sup>

It is necessary to distinguish between mere sonifications of these processes and genuinely musical works. The role of the composer seems more akin to that of an arranger, whose task is to extract the maximum expressive potential from previous material. The musical impact of the final product is undoubtedly much more dependent on the intelligence behind these latter choices than on the characteristics of the raw material —anyway, like any other arrangement!—. Admitting that this issue is full of aesthetic nuances, it seems evident that the use of sonification does not exempt the composer from their ultimate

---

<sup>23</sup>It is noteworthy that from simple automata or structures of equivalent simplicity, it is very common to obtain fractal structures. Thus, in a certain sense, the two techniques are linked, as self-similarity can be included as one of the types of behavior generated by cellular automata. Furthermore, one of the most important discoveries in this field is the confirmation that extremely simple cellular automata can be universal Turing machines [32], implying that any level of complexity can potentially be achieved using only these mechanisms as a foundation. In a musical context, this idea is tremendously suggestive.

<sup>24</sup>In a certain sense, the conceptual appeal of these principles connects directly with the German tradition of seeking formal beauty through the derivation of all materials from small thematic cells.

<sup>25</sup>Some connections link fractal geometry, chaotic systems, and cellular automata. Voss and Clarke’s observation [153] that the spectrum of musical signals tends to be self-similar in a  $1/f$  relationship, where  $f$  is the frequency analyzed, holds significant weight. Mandelbrot [91, p. 523] attributes this scaling property of music to the fact that musical compositions are, obviously, *composed*. He states that, once again, fractality and self-reference arise from simultaneous work across various time scales, because every musical piece must be composed down to the smallest meaningful subdivisions. This surprisingly connects with Ninagawa’s analysis [109] of a very simple cellular automaton equivalent to a universal Turing machine: the spectrum of the noise generated by this automaton also corresponded to the  $1/f$  relationship.

responsibility. Bennett [15] warns us of the dangers of getting lost in the fascination of data and losing the perspective of the musician:

Mathematical chaos, when treated with insight, reveals astonishing beauty, a beauty whose regularity derives from the strictly deterministic techniques employed to give birth to it. Music, on the other hand, must deal not with number, but with real, sounding, materials. When treated with insight, these too reveal great beauty, rougher, less regular than fractal beauty, to be sure, but beauty within which the echoes of the real Chaos can clearly be heard.

Miranda, who has experimented with all these aforementioned techniques, has recently pioneered the introduction of new techniques based on quantum computing [106, 107]. This type of application also connects with the field of data sonification, an area that has been vibrant in recent years, where the transformation of raw data into sound coexists for purely scientific purposes along with an aesthetic usage that explores poetic dimensions and musical possibilities of such transductions.<sup>26</sup>

## 1.5. Grammars for automated composition

In this alternative approach, one starts with categories more typical of musical analysis to manipulate them and obtain new results. Following the huge influence of Chomsky's generative grammar theory, attempts to apply these same principles to musical language quickly emerged. The foundational literature [142, 77] sought possible universal laws that condition musical perception. The reverse path to analysis has been the synthesis of music from the formulation of alternative grammars.<sup>27</sup>

Programming grammatical inference rules was already possible with the early computers, and soon these techniques were envisioned in both electroacoustic music and CAC. Roads [123, p. 53] extensively analyzed the various approaches that were emerging for automated composition based on defining different types of grammars, arriving at this conclusion:

Examples of recent composing and control languages [...] demonstrate how the specification of original and innovative composing grammars or parse trees may become increasingly available as a compositional technique. In such a technique, maximum flexibility is achieved by logically separating the syntactic specification (for building

---

<sup>26</sup>Parallel to this study, during these years I have also delved into the field of data sonification, both in its scientific and artistic aspects, in projects such as *Chasmata* [57, 61]. In future versions of the presented model, I will incorporate dedicated functions to enable the integration of datasets into the composition procedures.

<sup>27</sup>Zbikowski [159] has compiled a more updated collection of these already classic approaches.

a grammar for a set of scores or a parse tree for a particular score) from the sonic specification (the lexical mapping or orchestration from the score to sound objects).

Whereas for the techniques in the previous section, the key lies in mapping processes, for musical grammars, the crucial aspect is the potentiality of the domains that each grammar creates, as well as the search and selection strategies used to explore these spaces. Based on the precise definition of musical features, programs such as OpenMusic,<sup>28</sup> and PWGL<sup>29</sup> have emerged. They base part of their power on the ability to perform heuristic searches based on the composer's requirements. These environments particularly rely on user-defined constraints for the filtering of results.<sup>30</sup>

Both stem from the functional programming paradigm of the LISP language, traditionally linked to AI research, which, despite its age, has been revitalized in recent years.

An essential point in the process of synthesizing scores from grammars is the management of information flow. The fact that a musical texture is perfectly grounded on an inferential internal logic does not imply that its auditory reception is effective. Serial music was already a first step that in many cases far exceeded the possibilities of perceiving structures. The excesses of the 20th century in the complication of internal organization seem to be giving way to new generations of composers who consider more carefully the actual cognitive faculties. In their generative theory of tonal music, Lerdahl and Jackendoff [77, p.300] criticized this disconnection between the avant-garde writing of the time and the management of listening information:

The relevance of this distinction to the description of atonal and serial music pertains with equal or greater force to probabilistic methods of composition, to aleatoric methods, to serialism extended to the rhythmic dimension, or to any other procedures that do not directly engage the listener's ability to organize a musical surface. In each of these cases, the gulf between compositional and perceptual principles is wide and deep: insofar as the listener's abilities are not engaged, he cannot infer a rich organization no matter how a piece has been composed or how densely packed its musical surface is. It is in this sense that an apparently simple Mozart sonata is more complex than many twentieth-century pieces that at first seem highly intricate.

However, it can be opposed as an argument that listening to musical languages distant from the tonal system has other focal points of attention. Just as looking at an abstract

---

<sup>28</sup><http://repmus.ircam.fr/openmusic/home>

<sup>29</sup>One of the most advanced CAC systems, launched in 2002 and hosted by the Sibelius Academy in Helsinki, discontinued at the end of 2020.

<sup>30</sup>Sandred [131] has conducted a more updated study on the use of constraints in CAC. Its application for the imitation of traditional styles is also possible [7].

expressionist painting is a very different experience from figurative art, the textures generated by complex grammars can offer an interest in themselves, which in no case requires a conscious grasp of the organization of the underlying patterns.

In musical writing, polyphony<sup>31</sup> adds a significant complication to the design of grammars, as its implications simultaneously pertain to both the verticality and horizontality of listening. Eibensteiner [52] has reviewed approaches to this fundamental aspect of composition. The polyphonic dimension of music is another good example of structures that the common listener *understands* perfectly, without necessarily being aware of the techniques that produce that harmonic balance.

Finally, a clear distinction must be made between the use of the computer as a mere assistant for calculating structures designed and modeled by an external composer, and the attempts to automate aesthetic decisions —and eventually, to achieve machines capable of creating quality music autonomously- leading us to the domain of AI.

## 1.6. CAC meets AI

The history of AI is irregular. Spectacular advancements have been followed by periods of certain stagnation, and controversies about its true meaning and scope reappear periodically. Virtually all possible techniques of automated reasoning have been applied to music composition. We can distinguish three degrees in this relationship:

- Use of automated reasoning to perform specialized tasks, for which specific algorithms are programmed.
- Utilization of the machine as a tool for assisted creativity, where algorithms are given broad freedom to explore musical spaces defined according to certain rules.
- Complete creative autonomy, through algorithms that autonomously generate musical pieces.

However, one cannot properly speak of AI until there exists a self-observation of the processes, that is, until a mechanism of self-evaluation of the system is established allowing autonomous evolution of the products of this automated reasoning. It is evident that, beyond certain superficial parameters of a composition, real aesthetic evaluation —that is, experiencing pleasure firsthand through an individual sense of beauty— implies social,

---

<sup>31</sup>A musical texture consisting of multiple simultaneous melodic lines that, while preserving their temporal integrity, also sustain a certain level of vertical harmonic coherence.



cultural, and even ideological issues that are still beyond the reach of current possibilities of machines, and perhaps may never be attainable, at least in a manner comparable to humans.

In AI, there are two opposing philosophical positions: for those who advocate *weak AI*, such as Searle [139], intelligence can be mimicked, but the machine will never experience real self-awareness and, consequently, will never feel true aesthetic pleasure. Conversely, proponents of *strong AI*, led by Turing, argue that our own consciousness is nothing but the product of computation and that, in the long term, artificial computation will achieve the same degree of self-awareness as a human being. At this point, it would be the machines themselves that autonomously desire to create their own artistic productions...

How will that path be traversed? The paradigms of knowledge engineering have shifted in recent decades. The expert systems of the last decades have given way to the remarkably successful resurgence of neural networks [110] in their multiple variations. From relatively simple principles, a diversity of possible architectures has emerged: adversarial networks, the integration of feedback, convolution, and in recent years, the overwhelming success of large language models based on transformers [68],<sup>32</sup> whose possibilities have also begun to be explored for composition.

## 1.7. Bioinspired strategies

The programming mechanisms based on evolution and selection, primarily represented by genetic algorithms, have been experimented with for several decades, sometimes yielding surprising results, especially when applied to the search for solutions to well-defined problems. In the field of CAC, numerous experiments combine musical representation systems with algorithms that seek the best solutions through the crossing and mutation of successive generations of generated musical specimens. Beyls [20, 19] has explored interesting combinations of cellular automata and evolutionary algorithms. In all artistic disciplines, creative possibilities are being explored within societies of virtual individuals [79, 127]. A decade ago, the Iamus project [13, 47] reached a level of sophistication in its scores that raised questions about whether a hypothetical Turing test in the field of contemporary music had already been surpassed. Iamus was based on Melomics [130],

---

<sup>32</sup>It remains striking that the foundation of the attention mechanisms in transformers lies in applying algorithms derived from the Fourier transform to long sequences [76]. The spectral analysis of sound, ingeniously applied to language, enables the establishment of connections between distant elements and represents statistical relationships with surprising inference power.

a completely autonomous bioinspired system supported by evo-devo techniques that also rely on the concept of genotype as a precursor that will germinate into a musical score.

Expanding the historical perspective, it can be considered that already in the early serialist postulates of Schönberg and Webern, there was a hint of the idea of surpassing the traditional theme<sup>33</sup> and creating a seed. Later, Stockhausen [146] would coin the term *formula* as a generative principle preceding the form.

Formula is like the seeds sown by a man, a seed that fertilizes the woman. It amazes me to ponder the fact that a microscopic element, an incorporeal sperm, is able to fertilize another human being, to generate a new complex being who contains a plentiful genetic inheritance. This mystery is also valid for the musical genetics. [...]

A formula is a very small musical structure, a kind of musical seed, which like a DNA is programmed with different and varied musical elements, from the raga and the tala to the themes of the fugues and the sonatas, from the musical cells of the impressionism to the serial series, from micro-tonality to electroacoustics. The formula, therefore, is an integration of all these elements, constituting a cultural inheritance, prepared and produced by cultures, divergent between themselves, since the dawn of time.

Schaathun [134] clarified this concept in a less allegorical way:

What makes these immanent structures more audible, is that this approach to composition also takes care of the «higher levels», in the musical hierarchy: in addition to forming the basis of the microstructures, the individual note, the various rhythmic figures etc., it also takes into account the proportions of sections and, finally, the overall form of a composition.

The imitation of different aspects of biological mechanisms has led to experimentation with all kinds of models. Some of them use abstractions of cellular communication systems [30] to search for self-organizing structures. Other projects, like Jive [140], emphasize interactivity with the processes.

Numerous approaches have been made to put into practice grammars that work on these models of musical seed. In many of these works, the term *musical genotypes* is used to designate that formula that, once processed or decoded, generates a *musical phenotype*, usually a score or an audio signal. The combination of grammar design and evolutionary

---

<sup>33</sup>Each of the identifiable melodic motifs that traditionally appear in a composition, which are repeated and transformed throughout it.

algorithms [41, 117] arises quite naturally. And bioinspiration directly connects with certain fractal models. Lindenmayer systems (or L-systems)<sup>34</sup> has been extensively used by composers as Kyburz [111]. An example of excellent results is the application that Posadas makes of these arborescent structures combined with spectralist techniques for instrumental composition, studied among others by Besada [18] and Díaz de la Fuente [46].

It is common to find works like those of Albarracín-Molina et al. [4], which make a conceptual and practical connection between grammars and the concept of genotype. Once a grammar for the encoding of these musical genotypes is defined, a vast space of possibilities for exploration opens up. To carry out this prospecting, it becomes necessary to write programs that enable the automatic writing of genotypes, which are nothing but another type of programs, and this is achieved through *metaprogramming*.

## 1.8. Metaprogramming and functional programming

Metaprogramming, defined as the writing of programs by other programs,<sup>35</sup> is a rapidly evolving field. Experts such as Rideau [119] defend it as the keystone of programming, and Wolfram [156] stated that the true essence of computing lies in the self-organizing capacity of simple programs. If the future of AI involves creating algorithms that can generate other algorithms, it is essential to investigate how a piece of code can write more sophisticated code. The analogies with biological evolution are obvious. There are also resemblances between the mechanisms of metaprogramming and the functioning of enzymes: the genetic code of an enzyme, when activated, carries out the synthesis of other genetic codes that in turn can form other enzymes, and so on until it performs extremely complex operations within the cell.

For modern metaprogramming, based on probabilistic learning, numerous systems for representing code are being studied to optimize automatic inference. Allamanis et al. [6] have analyzed the diversity of current models and their relationship with the understanding of natural language, which is key in the development of large language models. The key moment in metaprogramming would be reached when algorithms capable of self-improvement are developed. Still far from that moment, in recent years the metaprogramming capabilities of the successful *copilots*, which have become an essential

---

<sup>34</sup>Manousakis carried out a practical implementation for composition with L-systems. His master's thesis [93] presents practical examples that illustrate the type of results obtained, both in symbolic information and in sound synthesis.

<sup>35</sup>Beyond this brief definition, the conceptual frameworks of metaprogramming are complex and multifaceted. Damaševičius and Štuikys [37] have conducted an extensive study on the related taxonomy.

tool for the human programmer in a very short time, bring us a little closer to Gödel's machines [137], hypothetical computational systems capable of tackling any type of problem through autonomous reprogramming and self-reference. Functional programming is one of the programming paradigms well suited for this multilevel coding.

Functional programming is based on the use of functions with independent internal behavior, meaning they do not affect external variables nor are affected by them. The main idea is that each procedure works in a way analogous to a standard mathematical function, and is therefore perfectly defined and its behavior identical and predictable in any data flow context. Functional programming is a direct heir to the notation used in  $\lambda$ -calculus (lambda-calculus),<sup>36</sup> a formalism conceived in the 1930s for researching the concept of function, its applications, and recursion. Lisp or Haskell can be seen as applications of *pure* lambda calculus; many other general-purpose languages, such as JavaScript [12] and Python [97], can be adapted to closely resemble a functional programming language, exhibiting various degrees of *purity*.<sup>37</sup> While functional programming was once thought to be more applicable to academia than to industry, and distinct from object-oriented programming commonly used in commercial application development, this perception has changed over the last two decades.

The convenience and almost necessity of functional programming were already analyzed and advocated by Hughes [70] in the 1980s, who pointed out the importance of compartmentalizing and modularizing software design in a scenario of increasing complexity. As the technology industry has evolved and the benefits of functional programming have become more evident, there has been a renewed interest in this methodology. These are some of the reasons why functional programming has experienced a resurgence:

**Performance and scalability:** For multiprocessor and multithreaded systems, it facilitates concurrency and parallelization due to its features of immutability and functions without side effects. It also aids in error prevention and detection.

**Modernization:** Modern functional languages such as Haskell, Scala, Erlang, Clojure, Elixir, and Elm have become popular in real-world applications.

**Suitability for web and distributed systems:** It aligns well with design patterns for web applications, distributed systems, and multi-core setups, areas of significant current growth. In web development, React and Redux have greatly promoted their adoption in creating and managing interactive user interfaces.

---

<sup>36</sup>Introductory texts to this formalism include the books by Michaelson [101] and Revesz [118].

<sup>37</sup>In functional programming, a style is considered pure when there are no side effects following the evaluation of a function. In other words, each function takes input data and produces output data, without affecting variables or states of the program outside its internal scope.

**Convenience for big data and data science:** Pure functions and immutability are highly recommended in the manipulation of large data sets.

**Incorporation into machine learning:** Many AI libraries and tools use functional techniques, stemming from a decades-long tradition that favored functional programming in pioneering knowledge engineering research.

**Presence in academia:** Functional programming remains an active research topic, with a growing community.

In the musical field, different approaches have arisen: from direct applications of Haskell [69] for musical output, to specialized languages like Common Music [149], which is built on Lisp implementations. Many more applications have been developed based on the concept of conceiving *scores as programs*, including the Canon language, about which Dannenberg [38, p. 47], its author, has commented:

In contrast to note lists, sequencers, and other approaches in which music is represented as data to be operated upon by an editor or other program, Canon scores are themselves programs. This allows scores to be parametrized and to incorporate arbitrary calculations including compositional algorithms.

Currently seeing more extensive use, SuperCollider<sup>38</sup> is a programming language and environment that combines the functional paradigm with object-oriented architecture. Its syntax offers expressiveness, suitability for live coding, and an effective combination of algorithmic composition functions and sound synthesis.

## 1.9. The problem of automated aesthetic evaluation

An essential characteristic of intelligence is its ability to assess its own reasoning and actions. The self-evaluation of processes with well-defined objectives —such as obtaining the structure with the greatest resistance, the most efficient locomotion system, or the best chess move— relies on measuring objective variables. In the case of CAC, we are once again faced with fundamental and challenging questions: How can the quality of a composition be measured? Is it even possible to quantify aesthetic pleasure? What is the ultimate goal of music?

After considering the necessary tools for automated music writing, it becomes essential to establish a mechanism for evaluating and selecting the results. The idea of applying a

---

<sup>38</sup><https://supercollider.github.io/>

Turing test to artistic creations is tempting, but it is not without problems, as Ariza [10] pointed out. Other authors, such as Loughran and O'Neill, ask "why do we limit to human?" [89]. They consider that a Turing-style approach undervalues the *superhuman* results that algorithmic composition and artificial creativity can achieve. In other words, it is possible to conceive music with high aesthetic value that is, at the same time, evidently an artificial production.

The individual assessment of the aesthetic qualities of a piece is the result of all the circumstances and personal backgrounds of each listener. Knowledge of the historical context of the creation process also adds factors to the equation. Meyer [99, p. 32] states about the perception of the value of a musical work:

We understand and appreciate a work not only in terms of the possibilities and probabilities actually realized, but in terms of our sense of what might have occurred in a specific compositional context: that is, in terms of the work's implied structure. This is perhaps especially clear in music. [...]

Not only is understanding dependent upon stylistic knowledge but so is evaluation. The patterns that result from a composer's actual choices are judged, as well as understood, in terms of the options known to have been available given the constraints of the style he employed.

This implies that a clear distinction must be made between the evaluation of a work of *style imitation* and another in which the *development of a personal style* is sought:

- In the first case, it is possible to equip the machine with extensive prior knowledge. With a sufficient amount of well-curated data and a variety of heuristic functions, it becomes feasible to develop a certain level of automatic critical sense. The already classic works of Cope [33, 34] represented a great achievement at the time, just before the new rise of neural networks, and continue to be excellent results in the field of recreating styles with thematic, harmonic, and formal coherence.
- In the second case, which is applicable when seeking artificial creativity and the emergence of distinguishable individual styles, it becomes enormously elusive to establish valid evaluation criteria.<sup>39</sup> Galanter [56] has conducted a panoramic study of past and future perspectives on criteria for implementing automated evaluation. Recently, attempts have begun to develop an automated *objective evaluation* [154], which allows for establishing some kind of benchmarks for comparing artificial creativity systems.

---

<sup>39</sup>This does not differ much from the uncertainty faced by human musical criticism and self-criticism in the face of new creations that propose original ideas.

The use of new compositional techniques, such as those of CAC, is by definition situated in a musical *terra incognita* where it is very difficult to exercise solid criticism, but at the same time, they are endowed with enormous computational power, allowing for the exploration of a large number of possibilities in the search for interesting musical structures. For now, at the end of any automated process, there is a human programmer-composer who, more or less elusively, defines selection criteria. The delimitation of these criteria constitutes the compositional process itself, and they would be the ones that ultimately define the style and bear the ultimate responsibility for the aesthetic result. Meyer later states [99, p. 34-35]:

Human choices are involved in the making of aleatory music. [...] Aleatory composers make choices, then, not on the level of successive sound relationships within works, but on the level of precompositional constraints [...]. As a result, it seems reasonable to argue that though style plays no role in the listener's understanding and experience of such pieces, the composer's behavior has style and can for this reason be evaluated.

The experience of every musician corroborates that, in music, the categories of coherence or correctness or are often irrelevant regarding the aesthetic valuation of a musical fragment. There are plenty of correct and coherent, *well-formed* pieces in the grammatical sense, that are prosaic, and vice versa. The history of the codification and implementation of scholastic fugue in the academic field is a good example of the sterility and absurdity of formalist approaches taken to the extreme.<sup>40</sup> This dichotomy in pedagogical paradigms somewhat reproduces two of the basic lines along which AI has developed:

- The deconstructionist path, focused on the pure transmission of prior knowledge, which would be equivalent to the implementation of expert systems capable of performing a task very well, but quite limited in transcending their results through extrapolation and analogy,
- The constructionist approach, characteristic of more progressive educators, based on stimulating of cognitive abilities. In AI, this is equivalent to approaches that emphasize the modeling of learning and discernment processes. The core idea here is that, in the long term, *what* is done is not as important as *how* it is done.

---

<sup>40</sup>The didactics of counterpoint have been the subject of controversies between mechanistic approaches, which are reduced to the assimilation of rules for correct writing that can well be implemented in a computer, and others that try to take a broader and less dogmatic perspective to capture the essence of polyphony. De la Motte [40] offers a sharp critique of the traditional teaching of species counterpoint and advocates that musicality should be sought from the beginning because the mere exercise of a technical rule is not meaningful if it is disconnected from a guiding musical idea.

## 1.10. Insights into aesthetic pleasure

Some analysts have tried to find common patterns in the aesthetic experience. And, especially in conceptual art from the 20th century onwards, simplicity and conciseness in conveying an idea seem to be valued. The scientist's pleasure in finding the simplest formula that explains a phenomenon thus has a parallel in artistic creations that are capable of condensing a thought, an emotion, into the least amount of information. Schmidhuber [136] has formulated a theory of aesthetic pleasure centered on information compression. According to it, we feel pleasure in finding the most concentrated and effective way to express something. Thus, the best version of an aphorism, a proverb, or a joke is the one that uses the least amount of information to convey the message:

The principle of Occam's razor is not only relevant to science and mathematics, but to fine arts as well. Some artists consciously prefer "simple" art by claiming: "art is the art of omission". Furthermore, many famous works of art were either consciously or unconsciously designed to exhibit regularities that intuitively simplify them. For instance, every stylistic repetition and every symmetry in a painting allows one part of the painting to be described in terms of its other parts. Intuitively, redundancy of this kind helps to shorten the length of the description of the whole painting, thus making it simple in a certain sense.

We can thus understand that musical analysis seeks to extract the truly significant elements from a structure and reduce a complex surface to a few constructive principles. Conversely, the possibility of composing dense networks of musical relationships starting from very simple principles should please us.

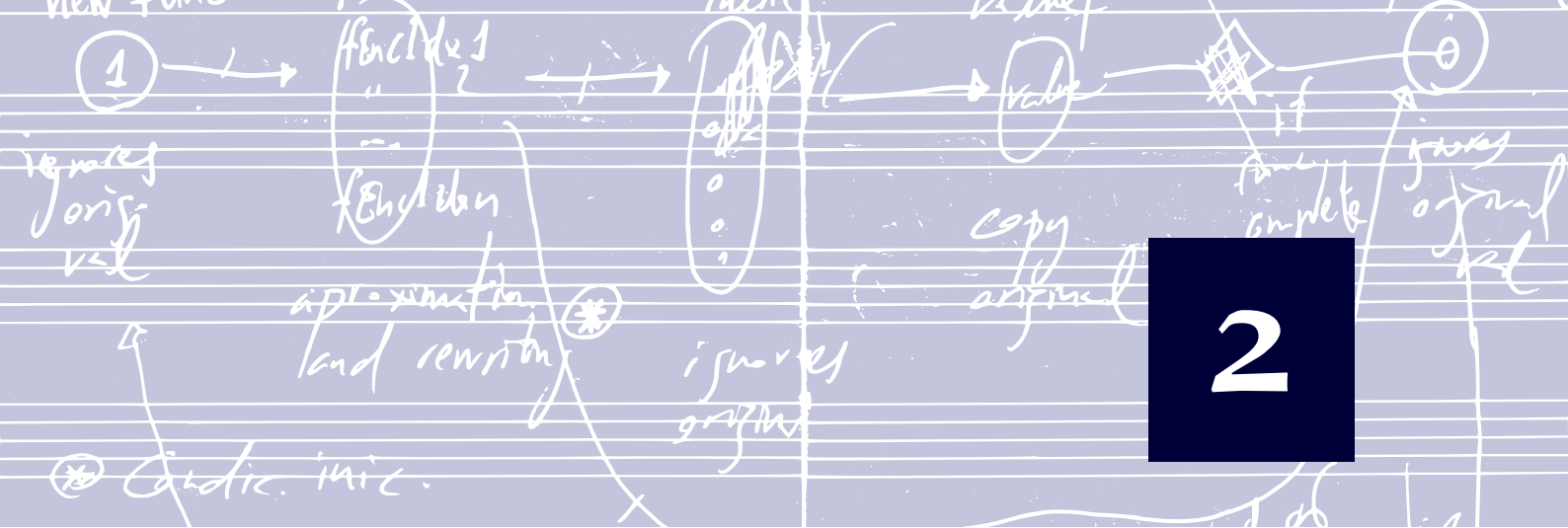
\* \* \*

In summary, in the history of automated music composition, there has been an ascending interrelation woven between computational frameworks such as the creation of grammars, functional programming, bioinspired models, and different paradigms of machine learning.<sup>41</sup> The model I present below also draws from many of these previous experiences to try to unite and readapt them in a paradigm that is oriented towards the most current techniques of deep learning, not just as another mechanism that blindly writes scores, but primarily as the next logical step in the evolution of the composer's craft, from the conviction that *composing is (meta)programming*. Entangled in all the recursive phenomena that occur around art, I wonder if there is beauty in the very development of code capable of creating beauty.

---

<sup>41</sup>A good example of these fertile interactions is an experiment from DeepMind [126] that combined genetic metaprogramming with LLMs to achieve some mathematical discoveries.





## Conceptual and formal framework

“

Lerner appears to believe that transformations that could be carried out by a computer program [...] could not possibly generate anything sensible—and that no program could tell sense from nonsense anyway. The implication [...] is that no computational theory could describe the generation of valuable new ideas, and that only an unanalyzable faculty of “intuition” or “insight” could recognize their value. None of these beliefs is justified.

Margaret Boden [22, p. 105]

### 2.1. Composing composers

Music creation and perception are extremely complex phenomena, simultaneously involving many time scales, cognitive layers, and social spheres. As an eminently creative activity, music is an excellent field of research to model mechanisms that lead to artistic production and to any human activity that requires creativity. This introduction summarizes the conceptual issues addressed by the GenoMus framework, outlined after considering many reviews of current research on automatic composition and artificial artistic creativity.

Despite the wide variety of existing systems, documented in metastudies and surveys [53, 88], and the development of new techniques in artificial intelligence in many domains, the autonomous creation of music by computers is still clearly behind human skills. These limited results may be due to overly specific approaches. Pearce et al. [115] strongly criticized many investigations for narrowing their focus to processes too restricted to the idiosyncratic style of a particular composer, or systems too oriented to the modeling of well-known historical styles. In particular, recent advances based on neural nets

struggle to capture big structures, as Nierhaus [108] has noted after comparing the current paradigms in automated music generation.

Rowe and Partridge [128], and Boden [22], propose the necessary conditions for the emergence of artificial creativity, such as a flexible knowledge representation with a high degree of ambiguity, capable of exploring, transforming, and expanding the search space. Multiple possible representations of the same idea are also recommended. Papadopoulos and Wiggins [114], Crawford [35], and López de Mántaras [82] claim that more imaginative behavioral models will arise from hybrid multiagent systems embedding a variety of functionalities. These algorithms must devise musical patterns along with their expressive potentialities, just as humans do.

Composers' interest in rethinking and reinventing musical language has pervaded aesthetics and techniques since the 20th century. Transformation and the overcoming of well-established methods inherited from Romanticism led to post-tonal music. Linguistic structuralism applied to musical syntax stimulated a relativization and awareness of compositional procedures. Reversing the logic of this analytic knowledge, some methods laid their foundation stone for an inverse creative strategy: synthesize new styles from the predefinition of new rules.

The availability of computers led to the thinking of music composition from a higher level. Composers such as Boulez [25] and Xenakis, who designed composition tools to work *towards a metamusic* [158], began to exploit the new ideas of generative grammars in the sound domain. Computer-assisted composition enabled far more complex procedures that were too tedious or unfeasible to explore by hand. Eventually, composers began to use computers not only for analysis and the calculation of complex structures but also for the automation of the creative processes themselves. That fact opened the door to a new approach to composition: a metamusical level characterized by modeling the processes within the minds of composers.

Research in artificial musical intelligence demands formalized grammars of musical structures. Furthermore, a model of the creative mind is required to operate these abstractions. Aesthetic criteria are extremely subjective. Furthermore, the implementation of every model of automatic composition imposes, consciously or not, a limited search space. Delimiting these boundaries and *setting evaluation principles can be seen as metacomposition*, namely, modeling composers' reasoning, often very obscure to themselves. As Jacob [72] analyzed, automatic composition not only creates new styles but also new ways of perceiving and feeling the music. Beyond this, the very concept of authorship becomes paradoxical. In general, *programming creativity* is an oxymoron that inevitably leads to metaprogramming [26].

## 2.2. Music as an encoded functional grammar

After studying many different approaches, and considering previous experience with CAC tools, I decided to create GenoMus as a new model of procedural representation of music optimized to be handled with different machine-learning techniques. Its key feature is an identical encoded representation of both compositional procedures and musical results as one-dimensional arrays. Functional expressions and their output are both encoded as sequences of normalized floats. Some systems, such as Jive [140], have explored this conversion of pure numeric sequences into some kind of intermediate computable expression. The system I present opens this formalism to be arbitrarily extensible.

The GenoMus grammar is designed to favor the broadest diversity of combinations and transformations. Genetic algorithms are suitable for the automation of incremental exploration and selection in multiple ways. Burton and Vladimirova [28] have studied several genetic methods applied to the generation of musical sequences; Dostál [49] also published a survey of techniques of evolutionary composition. GenoMus design favors genetic search processes in a very flexible manner since data structures have no determined length and are one-dimensional. The evolution of musical ideas without constraints and based on serendipity<sup>42</sup> is easily implemented and complemented.

On the other hand, approaches based on neural networks need a very controlled format of data and big training datasets. GenoMus' format represents any piece of music as a simple one-dimensional sequence of normalized floats, which can be profitable for techniques such as recurrent neural networks, which are capable of learning patterns from sequential streams of data. Both procedural and declarative information (composition techniques and music scores) share the same data format. This correspondence inevitably resonates with other Gödelian approaches to bioinspired generative projects based on genetic algorithms [42].

From a conceptual point of view, some of the projects most similar to GenoMus are those by Miranda [103] and Vico and Díaz-Jerez [151]. In its more practical aspect, as a CAC tool, the closest proposal is the Music Processing Suite<sup>43</sup> language and environment designed by Hofmann [63, 64, 65], also sharing commonalities with the projects by de la Puente et al. [41] and athenaCL by Ariza [9]. However, *GenoMus is not only oriented towards manual editing but primarily towards the encoding and compression of procedural information.*

---

<sup>42</sup>Discovery made by chance while investigating another matter. Applied to musical composition and within the context of this research, it involves finding interesting and unexpected musical outcomes by combining procedures designed for a different purpose.

<sup>43</sup><https://www.musicprocessing.net/>

Unlike many current models of automatic music creation, which primarily learn from audio signals or sequences of score events, my effort emphasizes a layer preceding these sources: the combination of primary abstract processes that operate before reaching the final output of musical compositions. What I aim to capture is the underlying structure of relationships established among musical elements. GenoMus operates on a new procedural metalanguage that functions as a precursor to declarative musical language.

For the technical specifications of this metalanguage to be suitable for both manual manipulation by a human user and external automated agents, the core of my proposal is *the design of an interchangeable dual grammar*: an extremely simple syntax for the compositional procedures that generate a music score, and a numerical encoding of that language compressed into a single-dimensional numeric vector. In the paradigm I present, this bidirectional relationship between the two representations is constructed in such a way that any list of numbers becomes a valid program and vice versa.

## 2.3. The biological metaphor

The artistic results of every algorithm designed for automated composition are strongly constrained by their representation of musical data. GenoMus is a framework for the exploration of artificial musical creativity based on a generative grammar focused on the abstraction of creative processes as a metalevel of compositional tasks.

Musical genotypes are conceived as functional nested expressions, and phenotypes as the pieces created by evaluating these computable expressions. GenoMus' grammar is designed to ease the combination of fundamental procedures behind very different styles, ranging from basic to complex contemporary techniques, particularly those that can produce rich outputs from very simple recursive algorithms. At the same time, maximal modularity is provided to simplify metaprogramming routines to generate, assess, transform, and categorize the selected musical excerpts. The system is conceived to maintain a long-term interrelation with different users, developing their individual musical styles. This proposed grammar can also be an analytic tool, from the point of view of composition as computation, considering that the best analysis of a piece is the shortest accurate description of the methods behind it.

Although its target is not only genetic algorithms but a variety of machine learning techniques, my framework uses the evolutionary analogy, similar to many other automatic composition systems [130, 148]. The Darwinian metaphor can be confusing, since each

system has its particular application of the same terms, sometimes denoting even opposite concepts.

In this functional approach, the key idea is considering any piece of *music as the product of a program that encloses compositional procedures*. Thereupon, the precise meaning of bioinspired terms in the context of the GenoMus framework is defined in Table 1:

| Key concept                | Definition   |
|----------------------------|--|
| <b>germinal vector</b>     | An array of floats $\in [0, 1]$ used as an initial decision tree to build a genotype.  |
| <b>genotype</b>            | Computable tree of genotype functions representing compositional procedures.   |
| <b>phenotype</b>           | Music score generated by a genotype.   |
| <b>encoded genotype</b>    | Genotype coded as an array of floats $\in [0, 1]$ .  |
| <b>decoded genotype</b>    | Genotype coded as a human-readable string, evaluable as a JavaScript functional expression.  |
| <b>subgenotype</b>         | Branch of a decoded genotype, which in turn is executable as a valid decoded genotype.   |
| <b>encoded phenotype</b>   | Music score coded as an array of floats $\in [0, 1]$ .   |
| <b>decoded phenotype</b>   | Music score coded as a human-readable nested data structure.   |
| <b>converted phenotype</b> | Phenotype converted to a format suitable for third-party music software.   |
| <b>initial conditions</b>  | Minimal data required to deterministically construct a genotype, consisting of a germinal vector and several constraints to handle the generative process. |
| <b>playback options</b>    | Conditions imposed on the generated score to globally influence some of its characteristics.   |

|                              |   |
|------------------------------|---|
| <b>genotype function</b>     | Minimal computable unit representing a musical procedure, designed in a modular way to enable taking other genotype functions as arguments.         |
| <b>leaf</b>                  | Single numeric value or list of values at the end of a genotype branch.   |
| <b>specimen</b>              | Output of the algorithm, representing a symbolic musical score in an abstract form.   |
| <b>rendered specimen</b>     | Initial conditions and playback options, along with the genotype/phenotype pair generated, metadata, and many other useful analytical informations. |
| <b>minimal data specimen</b> | Specimen containing metadata, initial conditions, and playback options. Must be rendered to obtain a musical piece.                                 |
| <b>subspecimen</b>           | JavaScript Object returned upon evaluation of a subgenotype, passed as an argument of the subgenotype containing it.                                |
| <b>species</b>               | Group of specimens that share the same parameter structure of their musical events.   |

---

*Table 1: Definitions of key concepts*

The genotype, in its decoded format, is a computable expression.<sup>44</sup> The phenotype, as the product of the evaluation of the functional expression content in a genotype, is a sequence of music events declared in a format designed to encode music according to a hierarchical structure of events, voices, and scores.

Extending the biological analogy, a species can be alternatively defined as a group of specimens that share the same parameter structure of their musical events.

Events built with many parameters can be set. For instance, a species for a very specific electroacoustic setup could need events defined by dozens of features. Events specification can also be extended to other domains beyond music, such as visuals, lighting, etc., along with musical events, or standalone. Ultimately, this framework can be applied to generate any output describable as sequences of actions.

---

<sup>44</sup>Consequently, the automatic writing and manipulation of genotypes can be seen as a metaprogramming process. This central idea of parsing languages (including musical language) to extract an essential abstract functional tree has been suggested by Bod [21].

## 2.4. Formal definitions

The Equation 1 defines a *GenoMus framework* as the 5-tuple:

$$G = \langle Types, \vec{t}, Maps, Funcs, Coders \rangle, \quad (1)$$

where:

*Types* is the set of parameter types integrating a genotype functional tree,

$\vec{t} = (t_1, t_2, \dots, t_n)$  is the vector of parameter types  $\in Types$  that constitutes an event data structure,

*Maps* is the set of conversion functions mapping human-readable specific formats of each parameter type  $\in Types$  into numbers  $\in [0, 1]$ , and their inverse functions,

*Funcs* is the set of genotype functions defined and indexed in a specific library, which take and return data structures  $\in Types$ , and

*Coders* =  $\{tran, dec, enc, eval, conv\}$  is the set of functions covering all required transformations to compute phenotypes from germinal conditions.

The 6-tuple of auxiliary parameters shown in Equation 2 defines the *restrictions* needed for the construction of a genotype from a germinal vector:

$$R = \langle extrap, type, Elig_{Funcs}, depth, maxl, seed \rangle, \quad (2)$$

where:

*extrap*  $\in \mathbb{N}$  is the number of extra parameters in each event, determining the *species*,

*type*  $\in Types$  is the output type of the genotype function tree,

$Elig_{Funcs} \subseteq Funcs$  is the subset of eligible functions encoded indices,

*depth*  $\in \mathbb{N}_{>0}$  is a depth limit to the branching of the genotype function tree,

*maxl*  $\in \mathbb{N}_{>0}$  is the maximal length for lists of parameters, and

*seed*  $\in \mathbb{N}$  is a seed state used to produce deterministic results with random processes.

Finally, the *initial conditions* needed to generate deterministically generate a specimen are contained in the ordered pair  $\langle \vec{x}, R \rangle$ , where  $\vec{x}$  is a *germinal vector*.

The three main abstract data structures of the GenoMus framework —germinal vectors, encoded genotypes, and encoded phenotypes— contain a vector of the same form: a simple one-dimensional array of  $n$  numbers  $\in [0, 1]$ . Due to limitations on the representation of reals, but mainly to enable some crucial numeric transformations as pointers to functions (explained in Section 5.1.4), these arrays only include values with a 6-digit mantissa. More formally, Equation 3 establishes that the only elements these vectors include are those members of the set:

$$V = \{x \in \mathbb{Q} \mid 0 \leq x \leq 1, x \cdot 10^6 \in \mathbb{N}\}. \quad (3)$$

Data structures handled below are encoded as vectors  $\vec{x} = (x_1, x_2, \dots, x_n)$  of any length  $n$ , belonging to a finite  $n$ -dimensional vector space defined in Equation 4:

$$V^n = \{\vec{x} \mid x_n \in V, n \in \mathbb{N}_{>0}\}. \quad (4)$$

The complete search space needed to contain all possible encoded representations is defined in Equation 5 as the vector space

$$S = \bigcup_{i=1}^n V^i, \quad (5)$$

a superset that contains all vectors of any length  $\geq 1$  up to  $n$ , where the upper limit of  $n$  depends on the practical limitations of computation and memory.

For a given set of restrictions  $R$ , any arbitrarily long vector  $\in S$  is a germinal vector  $\vec{x}$ . So, let  $I_{cond}$  be the set of all possible initial conditions, as shown in Equation 6:

$$I_{cond} = \{(\vec{x}, R) \mid \vec{x} \in S\}. \quad (6)$$

Let  $E_{gen}$  be the set of all possible encoded genotypes  $\vec{y} = (y_1, y_2, \dots, y_m)$  with the same restrictions  $R$ . Every germinal condition  $\in I_{cond}$  corresponds to a valid encoded genotype  $\in E_{gen}$  capable of generating a valid functional expression representing a music score. Unlike  $I_{cond}$ , which accepts as germinal vector any member  $\in S$ ,  $E_{gen}$  includes only vectors decodable as computable functional expressions.

The Equation 7 defines a surjective map  $\text{tran} : I_{cond} \rightarrow E_{gen}$ , which transcribes any germinal conditions into an encoded genotype. Hence, the set  $E_{gen}$  can be defined as:

$$E_{gen} = \{(\vec{y}, R) \mid \vec{y} \in S, \exists (\vec{y}, R) = \text{tran}(\vec{x}, R)\}. \quad (7)$$



The application *tran* involves several subprocesses covered next in detail, in such a way that the germinal conditions  $((x_1, x_2, \dots, x_n), R)$  generate a deterministic decision tree leading to the construction of a unique genotype  $((y_1, y_2, \dots, y_m), R)$ , mapping the values  $x_n$  to  $y_m$  one-by-one according to restrictions  $R$ . Since the number of choices  $m$  needed to build a valid encoded genotype rarely matches the length  $n$  of a germinal vector  $\vec{x}$ , truncation and loops are employed:

- If  $\vec{x}$  has more items than needed, they are ignored, effectively acting as a truncation of the remaining unused part of  $\vec{x}$ .
- If *tran* needs more elements than those supplied by  $\vec{x}$ , *tran* reads  $\vec{x}$  elements repeatedly from the beginning as a circular array, until reaching a closure, to complete a valid vector  $\vec{y}$ . That implies that germinal vectors with even a single value are valid inputs to be transformed into functional expressions.
- To avoid infinite recursion when loops occur while applying  $\text{tran}(\vec{x}, R)$ , the restrictions  $R$  introduce some limits to the depth of functional trees and the length of parameter lists.

Let  $D_{gen}$  be the set of all possible decoded genotypes. Members in this set are all possible text strings representing well-formed function trees, plus a *seed* value, necessary when random processes are involved. Another surjective map,  $\text{dec} : E_{gen} \rightarrow D_{gen}$ , takes an encoded genotype and produces its corresponding decoded genotype, presented as a human-readable evaluable expression. This is also a surjection since different but equivalent encoded genotypes can generate the same executable function tree as a text output.

Similarly, the map  $\text{eval} : D_{gen} \rightarrow E_{phen}$  evaluates deterministically decoded genotypes to produce encoded phenotypes; this is a surjective map too, because the same output can be obtained as the result of different music composition processes. Again, this encoded form is a sequence of values  $\in S$  representing a musical score. Now, we can define with Equation 8 the set of all encoded phenotypes —or latent music space— that can be produced from decoded genotypes  $\in D_{gen}$  as:

$$E_{phen} = \{\vec{z} \in S \mid \exists \vec{x} = \text{eval} \circ \text{dec} \circ \text{tran}(\vec{x}, R)\}. \quad (8)$$

It is noteworthy to point out that the length of most vectors  $\vec{z}$  does not correlate with the lengths of their corresponding encoded genotypes and germinal vectors in initial conditions. Indeed, simple procedures can generate long music scores, while a single chord or melodic motif may be the result of complex manipulations.

Finally, let  $D_{phen}$  be the set of decoded phenotypes generated at the end of the process. Creating decoded phenotypes implies a further map  $conv : E_{phen} \rightarrow D_{phen}$  to convert this data to another standard or custom musical format, either to generate symbolic information such as sheet music or to directly synthesize sound. This is a last trivial transformation, once it is known how a score is encoded.

## 2.5. Retrotranscription of genotypes into germinal vectors

Occasionally, when beginning with a manually programmed decoded genotype to construct a precise musical composition procedure, there emerges a need to identify corresponding germinal conditions capable of generating it. This step is crucial in obtaining abstract vectorial representations of manually edited expressions, enabling their integration into a hypothetical training dataset.

Reciprocally to  $dec$ , the function  $enc : D_{gen} \rightarrow E_{gen}$  is a map that takes as input a couple consisting of a valid expression  $\alpha$  and an auxiliary value *seed* (for repeatability dealing with random processes), and returns a corresponding encoded genotype. This transformation of  $\alpha$  is a simple conversion from text tokens to numbers  $\in V$ ;  $enc$  is an injective map since it returns only one encoded genotype from a text expression. Restrictions  $R$  for the obtained encoded genotype are derived directly from the features of the original functional expression.

The key point is that starting from a well-formed functional expression  $\alpha$ , an encoded genotype  $(\vec{y}, R) = enc(\alpha, seed)$  is at the same time one of its many possible germinal vectors. The Equation 9 shows this identity:

$$enc(\alpha, seed) = (\vec{y}, R) = tran(\vec{y}, R). \quad (9)$$

The  $tran$  function is built in such a way that a decoded genotype  $(\vec{y}, R) \in E_{gen}$  generated from any germinal vector is one of its reciprocal germinal vectors itself. In other words,  $\vec{y}$  simultaneously represents a symbolic functional expression and the decision chain that leads to the algorithmic writing of the very same functional expression.

Hence,  $E_{gen} \subset I_{cond}$ , and the map  $tran$  acts as an identity function when applied to decoded genotypes. This feature enables the repeatability and consistency of encoded genotypes regardless of changes in the function library  $Funcs$ , since the numeric pointers to eligible functions  $Elig_{Funcs}$  are preserved, while at the same time, the couple  $(\vec{y}, R)$  can be directly introduced back in the pool of germinal conditions without transformations.

Figure 3 illustrates this chain of surjective mappings and the retrotranscription to germinal vectors. Remarkably, a conversion from phenotype to genotype is far from trivial, as it is a reverse engineering process: the construction of a procedural generator of music can be seen as an analytical problem with many alternative solutions since the same musical pattern can be obtained by applying a combination of very different logical relations.

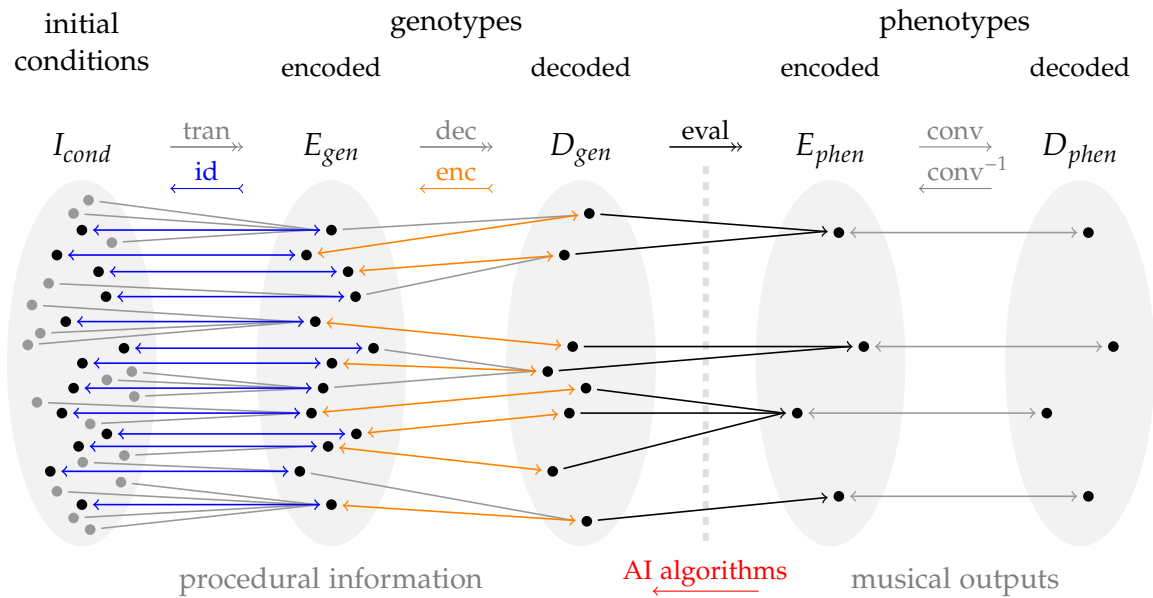


Figure 3: Mappings from germinal conditions to decoded phenotypes. Double-ended blue arrows illustrate the retrotranscription feature of the tran conversion: any encoded genotype vector  $\vec{y} \in E_{gen}$  belongs to the germinal vectors set  $I_{cond}$  as well, as it autogenerates itself. Orange arrows show how the encoding of a decoded genotype  $\in D_{gen}$  generates a unique numerical representation, although different encoded genotypes may correspond to the same decoded expression, due to the readjustment of parameters needed to fit into valid ranges.

\* \* \*

This last step can be provided by a variety of machine learning algorithms. Facilitating this type of abstract analysis, based on the learning of relationships between pure one-dimensional arrays, is the essence that determines the entire design of the GenoMus framework, and the field of research where the potential of this paradigm lies.

The ensuing pages endeavor to breathe life into this concept through a tangible implementation, exploring its technical and artistic capabilities.



## Main data structures

“

Debussy had been very helpful to Varèse. [...] Varèse often quoted his motto, “Works of art make rules, rules do not make works of art.” [...] According to Descartes, the senses are a source of errors, which the reason should correct: hence science should not have to do with the senses. Varèse was of the contrary opinion. As computer synthesis experiments progressed, he realized more and more the importance of perception as an unavoidable and often unintuitive interface between the physical world and its inner representation.

Jean-Claude Risset [120, p. 19]

This chapter presents the representation system with which the model operates. This system consists of various data structures designed to facilitate the bidirectional encoding processes discussed in Chapter 5. The primary abstractions represented are the specimen, the genotype functions, and the data structures they return, as well as the method of structuring the generated music. It explains how the musical output is structured in a modular manner, enabling flexible articulation both polyphonically and formally. Furthermore, it delves into the multidimensional structure of the event, placing a particular emphasis on the mapping solutions for its constituent parameters.

### 3.1. Anatomy of a specimen

The specimen is the output unit of GenoMus. It is stored as a JavaScript Object, maintaining compatibility for export and saving as a JSON file. It exists in two versions:

- **Rendered specimen**

The complete specimen generated each time initial conditions are evaluated along with playback options. It contains the generated genotype and the resulting phenotype, both in encoded and decoded formats, in addition to metadata. A specimen must be rendered to visualize and play the musical outcome in the Max interface.

- **Minimal data specimen**

It contains only the necessary data for regeneration: metadata, initial conditions, and playback options. Due to the deterministic implementation of the system, evaluating these minimal data always yield the same result. Since the file size is usually much smaller than the rendered version, this is the format used to save genotypes during candidate selection and evolution phases.

- **Subspecimen**

Any JavaScript Object returned by any genotype upon evaluation. It contains its own evaluable expression, as both encoded and decoded genotype, the phenotype, also encoded and decoded, and additional characteristics of the phenotype that will be used by the parent functions.

All formats can be exported as JSON files. In the case of rendered specimens, the musical result will be loaded into the user interface without the need to reevaluate the genotype. As a minimal example, Listing 1 displays the subspecimen returned by the genotype `s(v(e(n(2),m(60),a(100),i(100))))`, which is the simplest expression to generate a complete score with a *fortissimo* middle C lasting 2 seconds.

```
1 {  
2   funcType: 'scoreF',  
3   encGen: [ 1, 0.472136, 1, 0.854102, 1, 0.236068, 1, 0.09017, 0.51, 0.920833, 0,  
4           1,0.326238, 0.53, 0.430607, 0, 1, 0.562306, 0.55, 0.613655, 0, 1, 0.18034, 0.56,  
5           1, 0, 0, 0, 0 ],  
6   decGen: 's(v(e(n(2),m(60),a(100),i(100))))',  
7   encPhen: [ 0.618034, 0.618034, 0.920833, 0.618034,0.430607, 0.613655, 1 ],  
8   phenLength: 1,  
9   phenVoices: 1,  
10  harmony: { root: 0.430607 },  
    timegrid: { tempo: 1 },  
}
```

Listing 1: Simple subspecimen

The subgenotype is merely an intermediate data structure passed to the parent functions of a functional tree. It remains invisible at any moment. If this subgenotype corresponds to the complete final expression, it will be used to construct the complete specimen shown in Listing 2. The minimal data version would only contain the framed elements, as the rest is all a result of the evaluation of these preceding initial conditions. Even in this scenario where the score is the minimum possible for a single event, the amount of information required by the minimal data specimen is significantly less.

```
1 {
2   "metadata" : {
3     "specimenID" : "jlm-20231030_022052003-12",
4     "GenoMusVersion" : "1.0.0",
5     "rating" : 0,
6     "duration" : 2,
7     "voices" : 1,
8     "events" : 1,
9     "depth" : 4,
10    "encGenotypeLength" : 29,
11    "decGenotypeLength" : 33,
12    "generativityIndex" : 0.97,
13    "germinalVectorLength" : 29,
14    "germinalVectorDeviation" : 0,
15    "iterations" : 1,
16    "millisecondsElapsed" : 0,
17    "renderTime" : null,
18    "history" : {
19      "1" : "Genotype manually edited, novelty 0.7093 - 1v 1e 2s"
20    },
21    "storeIndex" : 214
22  },
23  "initialConditions" : {
24    "eventExtraParameters" : 0,
25    "specimenType" : "scoreF",
26    "localEligibleFunctions" : [ 0, 1, 2, 3, 4, 5, 7, 9, 10, 11, 12, 14, 15, 17, 19, 20, 21, 22, 25, 26,
27    27, 28, 29 ],
28    "depthThreshold" : 8,
29    "maxListCardinality" : 10,
30    "seed" : 426212866477420,
31    "germinalVector" : [ 1, 0.472136, 1, 0.854102, 1, 0.236068, 1, 0.09017, 0.51, 0.920833, 0, 1,
32    0.326238, 0.53, 0.430607, 0, 1, 0.562306, 0.55, 0.613655, 0, 1, 0.18034, 0.56, 1, 0, 0, 0, 0 ]
33  },
34  "playbackOptions" : {
35    "playbackRate" : 1,
36    "minQuantizedNotevalue" : 0,
37    "stepsPerOctave" : 12
38  },
39  "encodedGenotype" : [ 1, 0.472136, 1, 0.854102, 1, 0.236068, 1, 0.09017, 0.51, 0.920833, 0, 1, 0.326238,
40  0.53, 0.430607, 0, 1, 0.562306, 0.55, 0.613655, 0, 1, 0.18034, 0.56, 1, 0, 0, 0, 0 ],
41  "decodedGenotype" : "s(v(e(n(2),m(60),a(100),i(100))))",
42  "encodedPhenotype" : [ 0.618034, 0.618034, 0.920833, 0.618034, 0.430607, 0.613655, 1 ],
```

```

40 "subgenotypes" : {
41   "scoreF" : [ "s(v(e(n(2),m(60),a(100),i(100))))" ],
42   "voiceF" : [ "v(e(n(2),m(60),a(100),i(100)))" ],
43   "eventF" : [ "e(n(2),m(60),a(100),i(100))" ],
44   "paramF" : [ ],
45   "listF" : [ ],
46   "notevalueF" : [ "n(2)" ],
47   "lnotevalueF" : [ ],
48   "midipitchF" : [ "m(60)" ],
49   "lmidipitchF" : [ ],
50   "articulationF" : [ "a(100)" ],
51   "larticulationF" : [ ],
52   "intensityF" : [ "i(100)" ],
53   "lintensityF" : [ ],
54   "goldenintegerF" : [ ],
55   "lgoldenintegerF" : [ ],
56   "quantizedF" : [ ],
57   "lquantizedF" : [ ],
58   "harmonyF" : [ ],
59   "operationF" : [ ],
60   "recursiveF" : [ ]
61 },
62 "leaves" : [ [ 9, 0.920833, 2 ], [ 14, 0.430607, 60 ], [ 19, 0.613655, 100 ], [ 24, 1, 100 ] ],
63 "decodedPhenotype" : {
64   "metadata" : {
65     "totalVoices" : 1,
66     "effectiveVoices" : 1,
67     "totalEvents" : 1,
68     "effectiveEvents" : 1,
69     "eventsPerVoice" : [ 1 ],
70     "effectiveEventsPerVoice" : [ 1 ],
71     "durationsPerVoice" : [ 2000 ],
72     "rhythmicDurationsPerVoice" : [ 2000 ],
73     "scoreDuration" : 2000,
74     "rhythmicScoreDuration" : 2000,
75     "generalOnsetTime" : 0
76   },
77   "score" : {
78     "voice-1" : {
79       "event-1-1" : {
80         "onset" : 0,
81         "pitches" : [ 60 ],
82         "duration" : 2000,
83         "intensity" : 100
84       }
85     }
86   }
87 },
88 "roll" : [ "[", "markers", "[", 2000, "soundEnd", "]", "[", 2000, "rhythmEnd", "]", "]", "(", "(", 0, "
90   ("", 6000, 2000, 127, "(", "slots", "(", 7, "1-1", ")")", ")", ")", ")", ")", "]" ]
91 }

```

Listing 2: Simple rendered specimen

The Table 2 explains the specimen data structure. It can serve as a good map to visualize how they relate and their purposes within the unit, providing an overview of the rest of this chapter.

| Documentation       | Key                      | Description   |
|---------------------|--------------------------|---|
| <b>minimal data</b> |                          |   |
| 6.3                 | <b>metadata</b>          | Identification of the specimen, along with formal characteristics and data regarding its generation process |
| 6.3.1               | specimenID               | Unique identifier name of the specimen  |
| 6.3.2               | comments                 | User optional comments  |
|                     | GenoMusVersion           | Core code version   |
| 6.3.3               | rating                   | Last subjective rating assigned by the user   |
|                     | duration                 | Total effective duration in seconds   |
|                     | voices                   | Number of effective voices  |
|                     | events                   | Number of effective events  |
|                     | depth                    | Maximum depth reached in the decoded genotype functional tree   |
|                     | encGenotypeLength        | Length of the encoded genotype numeric array  |
|                     | decGenotypeLength        | Length of the decoded genotype string   |
| 6.3.4               | generativityIndex        | Ratio between the length of score and the length of encoded genotype  |
|                     | germinalVectorLength     | Length of germinal vector   |
| 6.3.5               | germinalVectorDeviation  | Deviation between the germinal vector and the encoded genotype  |
|                     | iterations               | Number of attempts needed to find the genotype  |
|                     | millisecondsElapsed      | Milliseconds elapsed during genotype generation   |
|                     | renderTime               | Milliseconds elapsed during phenotype rendering   |
| 6.3.6               | history                  | Log of the processes of specimen generation and transformation  |
|                     | storeIndex               | Candidate number of the specimen in the current generation  |
| 5.5                 | <b>initialConditions</b> | Key parameters that deterministically generate the specimen   |
|                     | eventExtraParameters     | Number of additional parameters for events, determining the species   |
| 3.2                 | specimenType             | Main function type of genotype, determining output type   |
| 4.10                | localEligibleFunctions   | Pool of available genotype functions  |
|                     | depthThreshold           | Limit of functional tree depth for the genotype; once reached, only identity functions can be selected      |
|                     | maxListCardinality       | Maximum number of items in parameter lists  |
| 4.4                 | seed                     | Global seed value for the repeatability of random functions   |
| 5.4                 | germinal vector          | Primary vector that initiates the decision tree determining the genotype                                    |



|       |                              |   |
|-------|------------------------------|---|
| 6.4   | <b>playbackOptions</b>       | Modifiers that globally alter the final result                          |
| 6.4.1 | <b>playbackRate</b>          | Global tempo rate applied to playback.                                  |
| 6.4.2 | <b>minQuantizedNotevalue</b> | Minimum notevalue to establish quantization of event durations          |
| 6.4.3 | <b>stepsPerOctave</b>        | Equal divisions of the octave to determine the available set of pitches |

---

|                      |                         |   |
|----------------------|-------------------------|---|
| <b>rendered data</b> |                         |   |
| 5.1                  | <b>encodedGenotype</b>  | Genotype encoded as an array of floats $\in [0, 1]$   |
| 5.4                  | <b>decodedGenotype</b>  | Genotype decoded as a string containing a functional expression                                       |
| 5.7                  | <b>encodedPhenotype</b> | Phenotype encoded as an array of floats $\in [0, 1]$  |
| 4.9                  | <b>subgenotypes</b>     | Dictionary containing all substrings of the decoded genotype that can be internally referenced        |
| 7.1.3                | <b>leaves</b>           | Positional index of all terminal numeric values at the end of each branch in the decoded genotype     |
| 5.8                  | <b>decodedPhenotype</b> | Decoded phenotype stored as a human-readable Object   |
|                      | <b>metadata</b>         | Formal characteristics of the phenotype: number of voices, events, partial and global durations, etc. |
|                      | <b>score</b>            | Tree-list of voices, events, and parameters   |
| A.5                  | <b>roll</b>             | Decoded phenotype converted to the bach.roll input format   |

---

Table 2: Specimen data structure. The horizontal line indicates where a minimal data specimen would end. In the Documentation column, there is a reference to the section that details the implementation of each data type.

## 3.2. Function types

The construction of specimens relies on a library of genotype functions. These functions are grouped according to the type of output, marked with a label. For each function type, there is an identifier that is one or two fixed characters at the beginning of the name of each genotype function. This is a convention that facilitates the analysis of functional trees and simplifies some parsing tasks. The required function types, their labels, and identifiers are listed in Table 3.

| Category               | Function type label | Identifier | Output                            |
|------------------------|---------------------|------------|-----------------------------------|
| <b>Main types</b>      | paramF              | p          | generic parameter                 |
|                        | listF               | l          | list of generic parameters        |
|                        | eventF              | e          | event                             |
|                        | voiceF              | v          | voice                             |
|                        | scoreF              | s          | score                             |
| <b>Human-readable</b>  | notevalueF          | n          | encoded duration                  |
|                        | lnotevalueF         | ln         | list of encoded durations         |
|                        | midipitchF          | m          | encoded MIDI pitch                |
|                        | lmidipitchF         | lm         | list of encoded MIDI pitches      |
|                        | articulationF       | a          | encoded articulation              |
|                        | larticulationF      | la         | list of encoded articulations     |
|                        | intensityF          | i          | encoded intensity                 |
|                        | lintensityF         | li         | list of encoded intensities       |
|                        | quantizedF          | q          | encoded quantized value           |
|                        | lquantizedF         | lq         | list of encoded quantized values  |
|                        | goldenintegerF      | g          | encoded golden integer            |
|                        | lgoldenintegerF     | lg         | list of encoded golden integers   |
| <b>Domain-specific</b> | harmonyF            | h          | harmonic grid                     |
|                        | timegridF           | t          | rhythmic pattern grid             |
|                        | recursiveF          | r          | operation for recursive equations |

Table 3: Genotype function types, tags, and identifiers

The function types can be categorized in this manner (though this classification holds no relevance in the implementation and serves solely to facilitate a better understanding of the needs they address):

- **Main function types**

Necessary basic types to articulate complete scores. They are the basic building blocks that enable the construction of the structures explained in Section 3.6.

- **Human-readable function types**

Variants of the `paramF` and `listF` types adapted for their decoded version to facilitate the reading and writing of genotypes. They cover the mandatory parameters of an event (`notevalue`, `midipitch`, `articulation`, and `intensity`) as well as other common parameters, such as integers referring to the number of iterations, repetitions, intervals. Nothing prevents adding other types for future needs that require different numerical mappings. On the other hand, in a genotype, all these functions can always be replaced by others from the generic `paramF` types for individual parameters and `listF` for parameter lists.

- **Domain-specific function types**

Types used for specific procedural tasks, such as harmonic and rhythmic structuring, or for creating recursive iterations as subroutines within a genotype. There is no issue in adding new types for emerging needs. It is advisable to design these types to be human-readable as well.

### 3.3. Anatomy of a genotype function

The algorithmic writing of functional trees is similar to the one proposed by Laine and Kuskankaare [75], also focused on bioinspired algorithmic composition. A useful and more formal study of the algorithmic generation of trees was published by Drewes et al. [50]. A simpler but comparable project, also based on the automatic writing of executable programs, was presented by Spector and Alpern [143].

A decoded genotype is a procedural representation of a music score written as a nested functional expression under the common syntax:

$$\text{funcName}(\text{argument1}, \text{argument2}, \dots, \text{argumentN}),$$

where each genotype function can take other functions as arguments until the limit imposed by the germinal condition *depth* is reached.

All genotype function calls share the same modular structure, to ease the algorithmic metaprogramming of genotypes.<sup>45</sup> The output of every genotype function is a JavaScript Object that includes the properties listed in Table 4.

| Key        | Value example                                      | Description   |
|------------|--|---|
| funcType   | 'lmidipitchF'                                      | output type of the function, a property used to create a pool of subgenotypes that can be referenced as an argument for other functions |
| encGen     | [ 1, 0.506578, 0.53, 0.430607, 0.53, 0.470107, 0 ] | encoded genotype  |
| decGen     | 'lm(60,62)'  | decoded genotype; that is, the expression of the evaluated function that returns itself   |
| encPhen    | [ 0.430607, 0.470107 ]                             | encoded phenotype   |
| phenLength | 1  | Total events in the phenotype (= 1 if not a voice or a score)   |
| phenVoices | 1  | Total voices in the phenotype (= 1 if not a score)  |
| (others)   | (not returned by function lm)                      | Specific properties generated after evaluation, containing useful information for the parent function.                                  |

Table 4: Object keys of a subspecimen returned by a genotype function after evaluation, using as example the expression `lm(60,62)`, which represents a list of two MIDI pitches.

This data structure is what each genotype function expects for every argument, and what is passed to the next one. A crucial item is the property `decGen`, where a function returns its own code as a string. It allows reevaluations of its code in execution time, which is essential to enable generative procedures involving iteration, recursion, or stochastic processes.

<sup>45</sup>Although, due to the experimental setup, the syntactic structure in this implementation is effectively an executable JavaScript expression. Many programming languages share this basic syntax for making function calls. This would allow the system to be portable to other languages while maintaining the textual integrity of genotypes without the need for adaptation. In this regard, Pedregosa [116] already conducted an initial exploratory study on the possibilities of implementing `GenoMus` in C++.

As an example of a minimal genotype function, in Listing 3 it is commented the code for the identity function `m` of the `midipitchF` type. This function serves as a mere container for a MIDI pitch, returning it with some adjustment.

```
1 // midipitch identity function
2 indexGenotypeFunction("m", "midipitchF", 7, ["midipitchLeaf"]);
3 m = midipitch => {
4   // stores encoded function index number and leaf index number
5   var encodedFuncID = g2p(7);
6   var leafID = leavesInfo["midipitchLeaf"].ID;
7   // calculations and transformations
8   midipitch = p2m(m2p(midipitch)); // adjusts the pitch to tuning system
9   // Object returned after indexing all subgenotypes
10  return indexDecGens({
11    funcType: "midipitchF",
12    encGen: [1, encodedFuncID, leafID, m2p(midipitch), 0],
13    decGen: "m(" + midipitch + ")",
14    encPhen: [m2p(midipitch)],
15    phenLength: 1,
16    phenVoices: 1,
17  });
18 };
```

Listing 3: Minimal genotype function example (`midipitchF` identity function `m`)

The use of the `indexDecGens` function in line 8 is common to almost all functions. Its purpose is to index all new genotype subgenotypes contained in the result of the function evaluation. The evaluation of the expression `m(60)` to refer to middle C returns this Object:

```
{
  funcType: 'midipitchF',
  encGen: [ 1, 0.326238, 0.53, 0.430607, 0 ],
  decGen: 'm(60)',
  encPhen: [ 0.430607 ],
  phenLength: 1,
  phenVoices: 1
}
```

Listing 4: Subspecimen returned by expression `m(60)`

The same basic structure can be found in function `vMotif`, in this case of the type `voiceF`, which constructs motifs from parameter lists. In its declaration, as shown in Listing 5, these elements are evident:

```
1 // creates a voice based on lists without no loops (shortest list determines number
  of events)
2 indexGenotypeFunction("vMotif", "voiceF", 199, ["lnotevalueF", "lmidipitchF", "
  larticulationF", "lintensityF"], "listF"); // indexes the function in the
  library
3 vMotif = (listNotevalues, listPitches, listArticulations, listIntensities, ...
  extraParams) => {
4   var encodedFuncID = g2p(199); // calculates encoded function index number
5   var comma = extraParams.length > 0 ? "," : "";
6   // this block performs previous calculations from given arguments
7   var seqLength = Infinity;
8   for (var idx = 0; idx < eventExtraParameters; idx++) {
9     seqLength = Math.min(seqLength, (extraParams[idx].encPhen.length));
10  }
11  seqLength = Math.min(
12    seqLength,
13    listNotevalues.encPhen.length,
14    listPitches.encPhen.length,
15    listArticulations.encPhen.length,
16    listIntensities.encPhen.length);
17  // checks that the result does not exceed certain established limits
18  if (seqLength > phenMaxLength) {
19    validGenotype = false;
20    if (verbosity) printLog("aborted genotype due to exceeding the max length");
21    return indexDecGens(evalExpr("v(" + defaultEvent + ")"));
22  }
23  // now the procedure of the function is executed
24  var eventsSeq = [g2p(seqLength)];
25  for (var event = 0; event < seqLength; event++) {
26    eventsSeq.push(listNotevalues.encPhen[event]);
27    eventsSeq.push(0.618034);
28    eventsSeq.push(listPitches.encPhen[event]);
29    eventsSeq.push(listArticulations.encPhen[event]);
30    eventsSeq.push(listIntensities.encPhen[event]);
31    for (var idx = 0; idx < eventExtraParameters; idx++) {
32      eventsSeq.push(extraParams[idx].encPhen[event]);
33    }
34  }
```

```
35 // Object returned after indexing all subgenotypes
36 return indexDecGens({
37   funcType: "voiceF",
38   encGen: flattenDeep([1, encodedFuncID,
39     listNotevalues.encGen,
40     listPitches.encGen,
41     listArticulations.encGen,
42     listIntensities.encGen,
43     extraParams.map(extraPar => extraPar.encGen),
44     0]),
45   decGen: "vMotif(" +
46     listNotevalues.decGen + "," +
47     listPitches.decGen + "," +
48     listArticulations.decGen + "," +
49     listIntensities.decGen + comma +
50     extraParams.map(extraPar => extraPar.decGen) + ")",
51   encPhen: eventsSeq,
52   phenLength: seqLength,
53   phenVoices: 1,
54 });
55 };
```

Listing 5: Declaration of **vMotif** function

As seen in Listings 3 and 5, each declaration of a genotype function is preceded by a call to **indexGenotypeFunction**, which indexes them in the `genotypeFunctionsLibrary` according to this format:

```
indexGenotypeFunction(<functionName>, <functionType>, <functionIndexNumber>,
  [<parameter1_funcType>, <parameter2_funcType>, ..., <parameterN_funcType>],
  <extraParameters_funcType>);
```

Listing 6: Indexing genotype functions in `genotypeFunctionsLibrary`

The function type designated last for `<extraParameters_funcType>` is only mandatory for functions that need to make use of extra parameters for events. For instance, the declaration of the **vMotif** function, which creates a voice containing a motif, is indexed as follows. The global genotype functions library is created during the system initialization. If the number of extra parameters is 3, the first lines of functions **m** and **vMotif** will index them as shown in Listing 7. All the details about the `genotypeFunctionsLibrary` can be found in Section 4.10.

```
m: {  
  encIndex: 0.326238,  
  intIndex: 7,  
  functionType: 'midipitchF',  
  arguments: [ 'midipitchLeaf' ]  
},  
vMotif: {  
  encIndex: 0.988764,  
  intIndex: 199,  
  functionType: 'voiceF',  
  arguments: [  
    'lnotevalueF',  
    'lmidipitchF',  
    'larticulationF',  
    'lintensityF',  
    'listF',  
    'listF',  
    'listF'  
  ]  
}
```

Listing 7: Genotype functions indexed in genotypeFunctionsLibrary

### 3.4. Leaf types

In the terminals of each branch of a functional tree, you find the leaves, concrete numerical values passed as arguments. When constructing genotypes, each function uses type labels to specify the type of function it requires as arguments. Certain functions don't require another function but a terminal parameter, that is a leaf. Each leaf type is similarly identified with a label. Table 5 gathers the leaf labels and shows their correspondences with function types.

Both function type labels and leaf type labels are keywords used by the genotype construction algorithm, as explained in Section 6.1.2.



| Category                | Leaf type label    | Homologous function type |
|-------------------------|--------------------|--------------------------|
| <b>Parameter leaves</b> | leaf               | paramF                   |
|                         | voidLeaf           | —                        |
|                         | notevalueLeaf      | notevalueF               |
|                         | midipitchLeaf      | midipitchF               |
|                         | articulationLeaf   | articulationF            |
|                         | intensityLeaf      | intensityF               |
|                         | quantizedLeaf      | quantizeF                |
|                         | goldenintegerLeaf  | goldenintegerF           |
| <b>List leaves</b>      | listLeaf           | listF                    |
|                         | lnotevalueLeaf     | lnotevalueF              |
|                         | lmidipitchLeaf     | lmidipitchF              |
|                         | larticulationLeaf  | larticulationF           |
|                         | lintensityLeaf     | lintensityF              |
|                         | lquantizedLeaf     | lquantizedF              |
|                         | lgoldenintegerLeaf | lgoldenintegerF          |

*Table 5: Leaf types labels*

### 3.5. Leaf parameters and mapping design

Considerable attention has been devoted to the design of musical parameters mapping. Although it may seem like a minor issue, its design has far-reaching consequences in the construction of the latent musical space that a generative algorithm explores. This has already been noted by researchers such as Doornbusch [48]. In his thesis, he exclusively investigates the influence and importance of mapping in algorithmic design and asserts:

[...] the pieces of music are inseparable from the mapping used in their creation, from the micro to the macro level, so much so that for at least some parts of the music, the music and structure is the mapping rather than the music being a product of the underlying data.

To obtain different functions that transform generic parameters  $\in [0, 1]$  into a readable representation of musical event properties, many variants have been tested to achieve a good balance in satisfying the following criteria:

- Capability to represent a **wide range of values** for each parameter. The ability to encompass extreme values mapped within the normalized range implies that the search space becomes much broader. This is positive in terms of significantly expanding the expressiveness of the system, but it comes with the trade-off that it will statistically be more challenging to find optimal solutions.
- **Probabilistic adjustment** to create a search space biased towards the most common values for each parameter, better representing typical usage in real music.
- **Readability** in the decoded version. Use of easily manageable ranges, close to common standards.

### 3.5.1. Eligible values and Gaussian conversion

Before analyzing the mapping of each parameter type, it is necessary to specify that eligible values, in their normalized and encoded version, are values  $\in [0, 1]$  with a maximum of 6 decimal places. This limitation has several reasons:

- It ensures conformity with the Max interface, which does not support greater precision in floats for communication with Node.js. This is a purely circumstantial matter but has certain other interesting consequences:
- Reduces the size of the generated data. There are  $10^6$  possible values for each parameter, which is more than sufficient.
- Also facilitates the use of the special type of parameter I have denominated *golden encoded integer*, which is employed in various contexts of the core code and benefits from this limitation of available values.

To maintain the readability of decoded genotype expressions, each parameter type uses common values (for example, pitch is represented with standard MIDI numbers). These convenient numeric ranges are converted to normalized values following these criteria:

- A wide range of values is covered (for instance, for parameters regarding rhythm and articulation, very short and long durations are available).
- For each parameter type, a central value of 0.5 is assigned to a midpoint among the typical values, interval  $[0.2, 0.8]$  covers the most usual range, and values  $< 0.2$  and  $> 0.8$  are reserved for extreme values rarely used in common music scores.
- To favor the predominance of ordinary values, a previous custom conversion, similar to the lognormal function, is applied to each encoded parameter value  $x$  supplied by a germinal vector.

So, by randomly generating germinal vectors, uniformly distributed values  $\epsilon \in [0, 1]$  are remapped to a Gaussian-like distribution, introducing a desired bias to produce musical scores with common characteristics, without limiting the possibilities to the generation of specimens exhibiting more extreme features. Let gaussian be the function according to Equation 10, that enables this conversion remapping a uniform distribution to a normal one:

$$\text{gaussian}(x) = \frac{1}{2} + \frac{1}{10} \ln \left( \frac{\epsilon + x - 1}{\epsilon - x} \right), \quad (10)$$

where  $\epsilon = 1.00678367$  is a value to adjust the boundaries of the function to the values 0 and 1.

The inverse conversion is obtained using Equation 11:

$$\text{gaussian}^{-1}(x) = \frac{-\epsilon + \epsilon \cdot e^{10x}}{-148.413 - e^{10x}} \quad (11)$$

Figure 4 shows the result of this mapping of values from the linear distribution. The design of this conversion ensures that the central range of numbers  $\epsilon \in [0.2, 0.8]$ , reserved for the most common values of each specific parameter type, remains nearly as a straight line.

For the next equations, let  $\text{round}_0$  be the standard function that rounds a real number to its nearest integer, and let  $\text{round}_n$  be the auxiliary function that rounds a float to have only  $n$  decimals. The encoded values, denoted as  $x_e$ , fall within the range  $[0, 1]$ , while the decoded values, represented by  $x_d$ , are displayed in the human-readable code of the decoded genotypes.

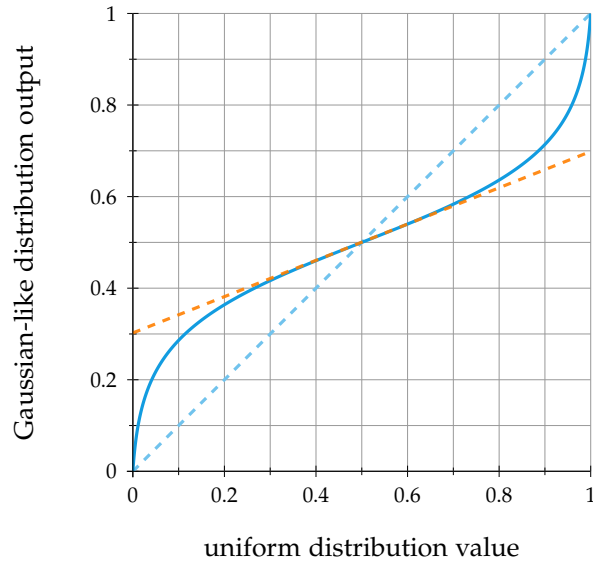


Figure 4: Mapping uniform distribution to Gaussian-like. The solid blue line shows the result of the conversion using the Gaussian function. The dashed orange line displays the best adjustment of a straight line for that conversion, to illustrate the fit for central values. The dashed blue line indicates the interval without any conversion, to assess the extent of deviation introduced by the distribution change.

### 3.5.2. Generic parameter leaf

The terminal value type leaf is employed by the generic parameter type identified with the label `paramF`, which can be used for any user-defined purpose as well as for mandatory specific parameters. It is solely filtered through the `formatParam` function shown in Equation 12, which automatically limits the normalized range between 0 and 1, applied to prevent manually entered genotype values from falling outside the valid interval.

$$\text{formatParam}(x_e) = \begin{cases} 0, & \text{if } x_e < 0 \\ \text{round}_6(x_e), & \text{if } x_e \in [0, 1] \\ 1, & \text{if } x_e > 1 \end{cases} \quad (12)$$

According to this conversion, the evaluation of the minimal genotype `p(0.12345678)` returns the Object shown in Listing 8:

```
{
  funcType: 'paramF',
  encGen: [ 1, 0.708204, 0.5, 0.123457, 0 ],
  decGen: 'p(0.123457)',
  encPhen: [ 0.123457 ],
  phenLength: 1,
  phenVoices: 1
}
```

*Listing 8: Subspecimen returned by a generic parameter*

A generic parameter, once filtered to be within the normalized range and rounded to display only six values in the mantissa, is effectively identical to the corresponding values within the encoded genotypes.

### 3.5.3. voidLeaf

Some functions do not require any leaf parameter. For this situation, the tag `voidLeaf` is used. The function `pRnd`, indexed with the code `indexGenotypeFunction("pRnd", "paramF", 131, ["voidLeaf"])`, for instance, generates a random generic parameter, returning an Object like this when evaluated `pRnd()`:

```
{
  funcType: 'paramF',
  encGen: [ 1, 0.962453, 0 ],
  decGen: 'pRnd()',
  encPhen: [ 0.179478 ],
  phenLength: 1,
  phenVoices: 1
}
```

*Listing 9: Subspecimen returned by `pRnd()`*

### 3.5.4. notevalueLeaf

This type of leaf parameter represents the duration of an event in seconds. As it will be seen in Section 6.4.2, there is a global playback option that determines the *tempo* for an entire phenotype, called `playbackRate`. If this parameter is equal to 1, the duration will correspond to seconds. For a `playbackRate = 2`, the actual values will be half, and the musical segment will be played at double speed.

For the design of mapping durations to the normalized interval  $[0,1]$ , the aim has been to combine the possibility of representing a very wide range of rhythmic values with the need to avoid excessive dispersion of proportions among the most probable values, resulting in overly eccentric outcomes too frequently. This has been implemented using Equation 13.

$$\text{notevalue}(x_e) = \begin{cases} \text{round}_6\left(\frac{1}{4} \tan\left(\frac{\pi x_e}{2}\right)\right), & \text{if } x_e \leq 0.08 \\ \text{round}_5\left(\frac{1}{4} \tan\left(\frac{\pi x_e}{2}\right)\right), & \text{if } x_e \in [0.08, 0.990053) \\ 16, & \text{if } x_e > 0.990053 \end{cases} \quad (13)$$

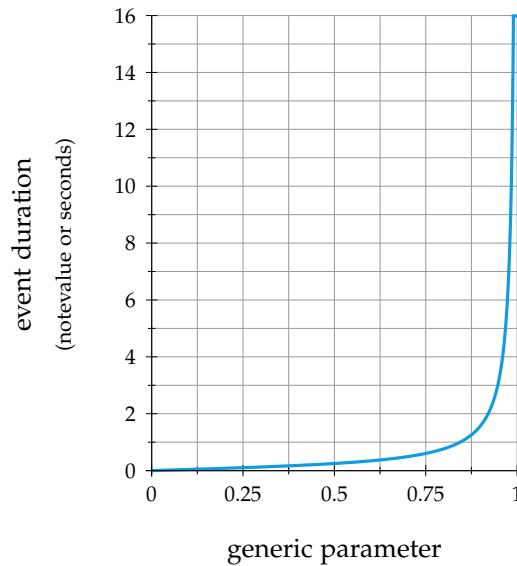


Figure 5: Conversion from generic parameter to notevalue

The differentiation in rounding type based on the input range has been introduced so that simple rhythmic values, which are more commonly used, have a simpler numerical form. The loss of precision is minimal, but readability and consistency between the input code and the recoded values after conversions are significantly improved for those commonly used numbers.

With this conversion, as can be observed in Figure 5, the defined range allows for representing extremely short durations. This spans from extremely short events suitable for enabling granular textures, to durations lasting many seconds, suitable for sequences of widely spaced events.

To verify the achievement of the second requirement, thereby avoiding excessive value differences in the central area of the conversion, it is preferable to represent the previous graph on a logarithmic scale. Figure 6 allows visualizing that the central half covers a ratio of 1 : 16. This corresponds, for example, to the five rhythmic values between the whole note and the sixteenth note. In a typical musical piece, the maximum and minimum durations tend to not exceed that quantity of binary orders of magnitude.

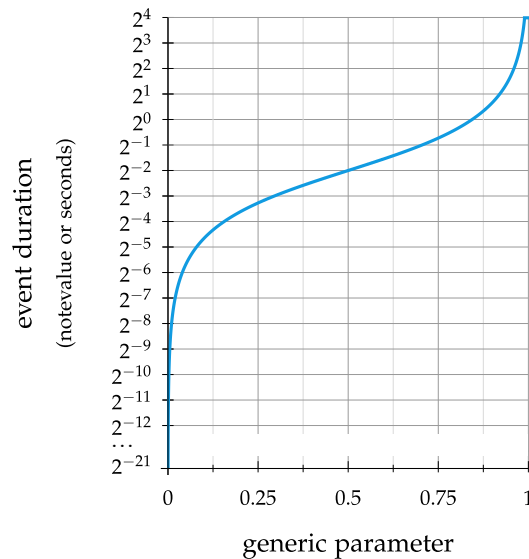


Figure 6: Logarithmic plot of the conversion from generic parameter to notevalue. The logarithm base 2 is used to represent steps to rhythmic values that double the duration, similar to how the palette of rhythmic figures in the musical notation system operates.

To revert a notevalue to the generic parameter used by encoded genotypes, Equation 14 is employed.

$$\text{notevalue}^{-1}(x_d) = \begin{cases} \text{round}_6\left(\frac{2 \arctan(4x_d)}{\pi}\right), & \text{if } x_d \in [0, 16) \\ 1, & \text{if } x_d \geq 16 \end{cases} \quad (14)$$

Among the playback options is the quantization of rhythmic values to fit a minimum value and its multiples. For the modification of the previous conversion, we first need Equation 15, which will be used to quantize a rhythmic value  $x$  to the nearest value that is a multiple of  $q \in [0, 16]$ , the minimum rhythmic value from which the quantization grid is derived.

$$\text{quant}(x) = q \cdot \text{round}_0\left(\frac{x}{q}\right) \quad (15)$$

Now, for a note value  $q$  that determines the minimum duration, Equation 16, a variation of the previous equation 13, is applied.

$$\text{quantizedNotevalue}^{-1}(x_d) = \begin{cases} q, & \text{if } x_d \leq \text{notevalue}(q) \\ \text{round}_6\left(\text{quant}\left(\frac{1}{4} \tan\left(\frac{\pi x_d}{2}\right)\right)\right), & \text{if } x_d \in [0.08, 0.990053] \\ 16, & \text{if } x_d > 0.990053 \end{cases} \quad (16)$$

### 3.5.5. midipitchLeaf

The numerical representation of pitch in the MIDI standard is widely adopted, so it is used as a human-readable representation. Depending on the tuning system employed, we can confine ourselves to the conventional 12-semitone chromatic scale, establish other microtonal divisions, or use the entire chromatic range with a maximal resolution of about 0.005 cents.

The configuration of the set of eligible pitches can be done at various moments in the process. The global variable `stepsPerOctave` affects the entire generative process and conditions the adjustment of a `midipitchLeaf` to the available values. When this variable is changed from the interface, the auxiliary converter functions are updated, and the current specimen is recalculated accordingly.



This is independent of the harmonic grids that can affect all or part of a genotype. In any case, the hierarchy of the global variable is always superior. So, pitches in quarter tones will result in a degraded resolution if the variable  $\text{stepsPerOctave} = 12$ , that is a conventional chromatic scale.

Defining  $s$  as the number of equal divisions of the octave, the Equation 17 converts a generic parameter to midipitch:

$$\text{midipitch}(x_e) = \text{round}_6 \left( \frac{\text{round}_0(127t \cdot \text{gaussian}(x_e) + s)}{s} - 12 \right) \quad (17)$$

The inverse conversion, in equation 18, does not require the constant  $s$ :

$$\text{midipitch}^{-1}(x_d) = \text{round}_6 \left( \text{gaussian}^{-1} \left( \frac{x_d}{127} \right) \right) \quad (18)$$

Figure 7 shows how the generic parameters are mapped to midipitch values. The graph shows how for values of  $x \in [0.2, 0.8]$ , the range covers the three central octaves, maintaining an almost constant slope within the resulting intervals.

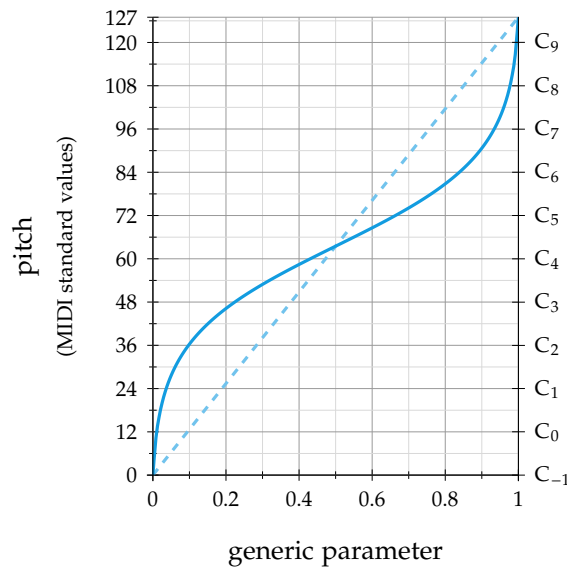


Figure 7: Conversion from generic parameter to midipitch. The blue dashed line shows how the mapping would look without the Gaussian-like adjustment of the input value.

### 3.5.6. articulationLeaf

The articulation allows differentiation between the duration of a concatenated event within a voice and the effective duration of sound. The duration, established by `notevalue`, conditions the time that will elapse until the insertion of the next event in the same voice. The articulation sets the percentage of that duration in which the event will effectively sound. In Figures 12 and 13, I already visually depicted this distinction between the nominal duration of the event and the effective sound duration.

Unlike the parameter for pitch, which already embeds a logarithmic relationship among frequency and musical intervals, for articulation, I introduced a double stretching of the available range: at the lower end to achieve extremely short effective durations, and at the higher end to create dense textures of overlay. Both cases are especially useful in sound synthesis and remote handling of virtual instruments. The implemented conversions from a generic parameter to articulation and its inverse are achieved using Equations 19 and 20.

$$\text{articulation}(x_e) = \begin{cases} \text{round}_0\left(75 \tan^2\left(\frac{\pi}{2} \cdot \text{gaussian}(x_e)\right)\right), & \text{if } x_e \leq 0.988456 \\ 10^4, & \text{otherwise} \end{cases} \quad (19)$$

$$\text{articulation}^{-1}(x_d) = \begin{cases} \text{gaussian}^{-1}\left(\text{round}_6\left(\frac{2}{\pi} \cdot \arctan\left(\frac{\sqrt{x_d}}{5\sqrt{3}}\right)\right)\right), & \text{if } x_d < 10^4 \\ 10^4, & \text{otherwise} \end{cases} \quad (20)$$

The rounding of the value to an integer in Equation 19 simplifies readability. Greater precision is not necessary to have an ample enough palette of effective durations. The graph in Figure 8 depicts the range of available articulations. The central value represents a percentage of 75% of the event's duration, resulting in a *non-legato*, while the central zone of values corresponds to the most common articulations, ranging from *staccato* to *molto legato*.

With this approach, an `articulation = 100` guarantees a *legato* between all events within a voice, regardless of the duration of each one. On the contrary, when the same effective duration is required regardless of the nominal duration, the `aFix` function should be used. This function, in turn, employs a parameter `notevalue` as effective duration by applying the necessary calculations.

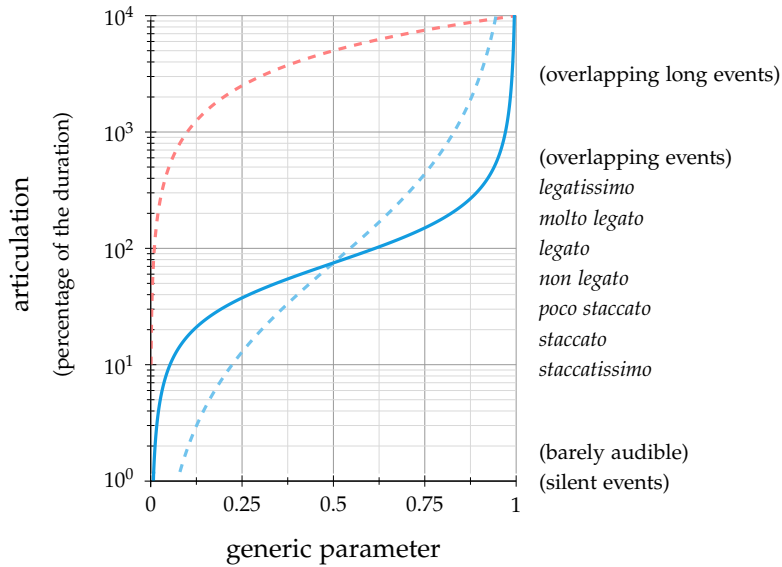


Figure 8: Conversion from generic parameter to articulation. The blue dashed line shows how the mapping would look without the Gaussian-like adjustment of the input value. Note that the solid line. As a reference, the dashed red line shows a completely linear transition between 0 and  $10^4$ .

### 3.5.7. intensityLeaf

The intensity parameter relies on the velocity<sup>46</sup> setting in the MIDI standard, which already incorporates a logarithmic relationship that ensures increments in its value are psychoacoustically consistent. However, instead of using its integer scale up to 127, I also rescale it as a percentage of the maximum possible intensity, aiming for a more intuitive understanding. However, as highlighted by Palamara and Deal [113] in their project focused on this subject, there are many technical and perceptual reasons why it is difficult to model objective scales for dynamics. Register, timbre, and many other issues affect our final sense of intensity.

In the conversion, performed with Equation 21 and plotted in Figure 9, the mantissa is limited to two decimal places for easier readability. This yields sufficient resolution for very fine nuances in dynamics. Equation 22 reverses this conversion.

$$\text{intensity}(x_e) = \text{round}_2(100 \cdot \text{gaussian}(x_e)) \quad (21)$$

$$\text{intensity}^{-1}(x_d) = \text{round}_6\left(\text{gaussian}^{-1}\left(\frac{x_d}{100}\right)\right) \quad (22)$$

<sup>46</sup>In the MIDI protocol, *velocity* is associated with dynamics and not with *tempo* as it might seem.

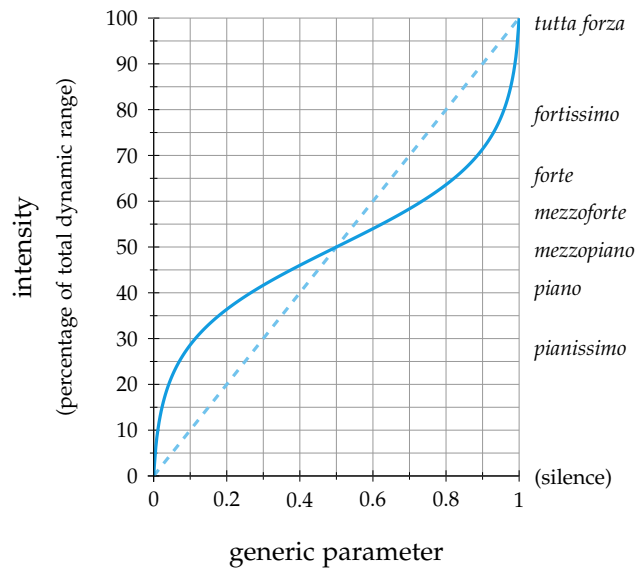


Figure 9: Conversion from generic parameter to intensity. The blue dashed line shows how the mapping would look without the Gaussian-like adjustment of the input value. There is no unified correspondence between numeric values and traditional indications of dynamics in sheet music; each notation software uses a slightly different mapping. The one used here is merely indicative.

### 3.5.8. quantizedLeaf

This type covers an auxiliary need and does not refer to any particular specific musical parameter. It serves multiple purposes where an integer, typically small, is involved. Determining the number of iterations, a transposition interval, a self-reference to another function, etc., requires the use of `quantizedLeaf` to make these values readable.

The conversion of a generic parameter to a `quantizedLeaf`, implemented with Equation 23 and reversed with Equation 24, allows for a range of integer values within the interval  $[-1000, 1000]$ . Again, the mapping introduces a stretching so that extreme values become progressively much less probable, as shown in Figure 10.

$$\text{quantize}(x_e) = \begin{cases} -1000, & \text{if } x_e < 0.003822 \\ \text{round}_0\left(12 \tan\left(\pi x_e + \frac{\pi}{2}\right)\right), & \text{if } x_e \in [0.003822, 0.996178] \\ 1000, & \text{if } x_e > 0.996178 \end{cases} \quad (23)$$

$$\text{quantize}^{-1}(x_d) = \begin{cases} 0, & \text{if } x_d \leq -1000 \\ \frac{1}{2} + \frac{\arctan\left(\frac{x_d}{12}\right)}{\pi}, & \text{if } x_d \in [-1000, 1000] \\ 1, & \text{if } x_d \geq 1000 \end{cases} \quad (24)$$

To better visualize how the mapping distributes integer values, progressively becoming less probable, Figure 10 displays both linear and logarithmic representations of the conversions. The logarithmic plot displays only the positive part of the conversion.

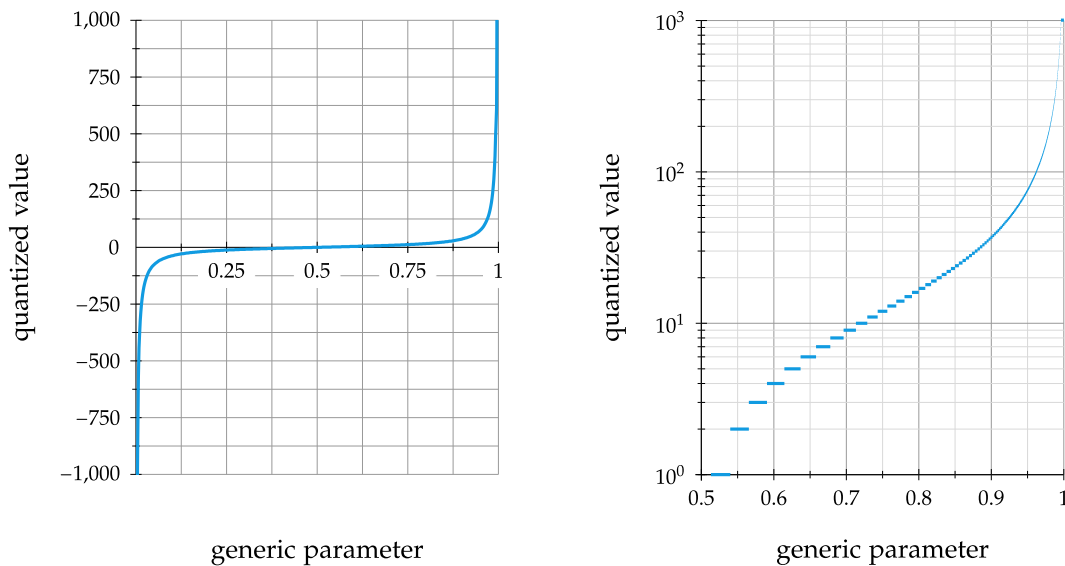


Figure 10: Linear and logarithmic plot of the conversion from generic parameter to quantized value. It can be observed how the encoded values in the range  $[0.5, 0.75]$  are mapped to integer values within the range  $[0, 12]$ .

### 3.5.9. Golden encoded integers and goldenintegerLeaf

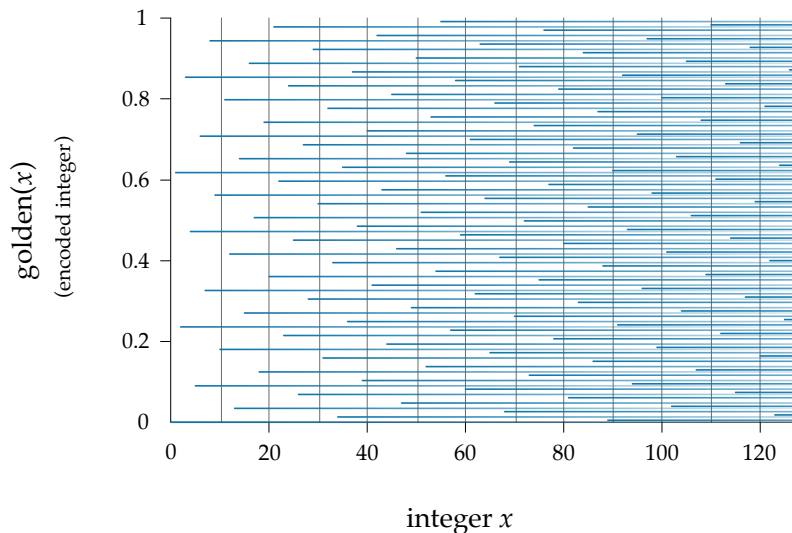
A special type of conversion has been added for various purposes. Although in specific cases it is also used as leaves in a genotype, its development stemmed from another motivation: it was essential to find a simple way to assign numerical identifiers to genotype functions that were within the normalized range  $[0, 1]$ , like the rest of the encoded elements, but distributed uniformly, regardless of the number of items involved. This issue will be

discussed in Section 5.1.4. To deal with all these requirements, this special conversion has been implemented. I refer to a *golden encoded integer* (or simply golden integer) as a float  $\in [0, 1]$  obtained after applying a bijective map based on the well-known properties of the golden angle (an angular version of the golden section), combined with modular arithmetics.

Taking  $\varphi = \frac{1 + \sqrt{5}}{2}$  as the golden ratio, this map is achieved by the Equation 25:

$$\text{golden}(n) = \text{round}_6(n\varphi \bmod 1) \mid n \in \mathbb{N}_{>0}, n < \max_n, \quad (25)$$

where mod is a modulus operator that works with floating-point numbers, and the first integer producing a repeated value is  $\max_n = 514262$ , returns the golden value corresponding to the integer  $n$ . The limited quantity of  $\max_n$  integers able to generate unique values  $\text{golden}(n)$  until reaching a repeated value, is big enough to attend its diverse applications.<sup>47</sup>



*Figure 11: Conversion from integer to golden encoded integer. First 125 integers mapped across interval  $[0, 1]$ . Every value is projected horizontally to visualize the balanced accumulative distribution. The lines gradually fade in intensity to better perceive the entry of each new value in the sequence.*

This conversion has a very convenient feature: despite ignoring how many values will be needed to be stored, the distribution across the interval  $[0, 1]$  is stochastically well-

<sup>47</sup>The calculation of  $\max_n = 514262$  has been purely empirical, generating values with  $\text{golden}(n)$  until finding the first value  $\in [0, 1]$  that repeats.

balanced. Figure 11 shows how the distribution of indices uniformly covers the range of normalized values.<sup>48</sup>

Golden integers are also useful for encoding discrete values in other contexts, both in genotypes and phenotypes, as shown in Figures 34 and 38. Some special genotype functions, as the autoreference class discussed in Section 4.9, need integers without a defined range as arguments, which involve this map again. For phenotype encoding, golden values are also indispensable for specifying discrete features such as the number of voices per score, events per voice, and items inside a multiparameter.

Although all the previous conversions are carried out using standard arithmetic operations, when it comes to golden integers, there is a peculiarity to consider. Calculating with the auxiliary function `goldenEncodedInteger`, for instance, that `golden(100) = 0.803399`, is a straightforward process. However, the inverse operation `golden-1(0.803399) = 100` would require a hundred operations and comparisons until a match is found. If the value  $\in [0, 1]$  has no possible conversion, hundreds of thousands of operations would be needed each time. That's why a lookup table is created at initialization time to streamline the `golden-1` function to this simple check. Listing 10 demonstrates how this conversion table is created and used.

```

1 // golden ratio constant
2 const PHI = (1 + Math.sqrt(5)) / 2;
3 // calculates golden encoded integers
4 var goldenEncodedInteger = integ => r6d(integ * PHI % 1);
5 // creates lookup tables for golden encoded integers at initialization time
6 var g2pTable = {};
7 (function(){ for (var integ = 0; integ < 514262; integ++) g2pTable[
   goldenEncodedInteger(integ)] = integ }());
8 var p2gTable = Object.fromEntries(Object.entries(g2pTable).map(([key, value]) =>
   [value, key]));
9 // generic parameter to golden encoded integer conversion
10 var p2g = param => g2pTable[param] == undefined ? -1 : g2pTable[param];
11 var g2p = integ => p2gTable[integ] == undefined ? -1 : parseFloat(p2gTable[integ]);

```

*Listing 10: Implementation of conversion to golden encoded integer*

---

<sup>48</sup>As far as I know, this kind of mapping based on the modulation of golden-angle properties to obtain the balanced distributions of an unknown quantity of indices is not common, although some methods have been recently proposed [100] to obtain simple algorithms to deal with similar optimization problems in other domains.

### 3.6. Internal structure of the score

The GenoMus syntax reconciles extreme simplicity with the expressiveness and flexibility of representation. The essential criterion in the design of the procedural representation system for musical composition can be summarized as follows: finding the ideal compromise between creating a modular framework reduced to the minimum variety of substructures and maintaining the ability to represent music with significant stylistic differences and internal complexity.

The score is the data structure generated as an output of the generative process. Its design maintains a highly simplified symbolic representation of the musical text while retaining flexibility to capture everything from simple monophonic fragments to intricate micropolyphonic swarms of overlapping sounds.

On the other hand, the event, as the fundamental sonic structure in the score, can accommodate an arbitrary number of additional parameters, in addition to the mandatory ones, to possess as many dimensions as necessary to characterize it.<sup>49</sup>

Between these two structures lies the voice, which serves as the container for sequences of events, often an abstraction that encompasses cells, thematic motifs, melodies, and sequential material in general.

Some features similar to those of the musical representation format in `bach.roll` for Max have also been adopted (which moreover facilitates their conversion). In `bach`, events are chords, with pitch as a multi-parameter that may have one or more pitches. This ability to treat a sequence of notes as a sequence of chords has been incorporated into the format, owing to its convenience in various composition contexts, where a single voice can contain a harmonically complex flow.

The following list outlines the essential rules for forming and combining GenoMus scores, as schematically illustrated in Figure 12.

---

<sup>49</sup>The events in the sequential scores of `Csound`, the veteran programming language for sound synthesis, also have mandatory initial parameters and an indefinite number of user-defined extra parameters for any purpose.



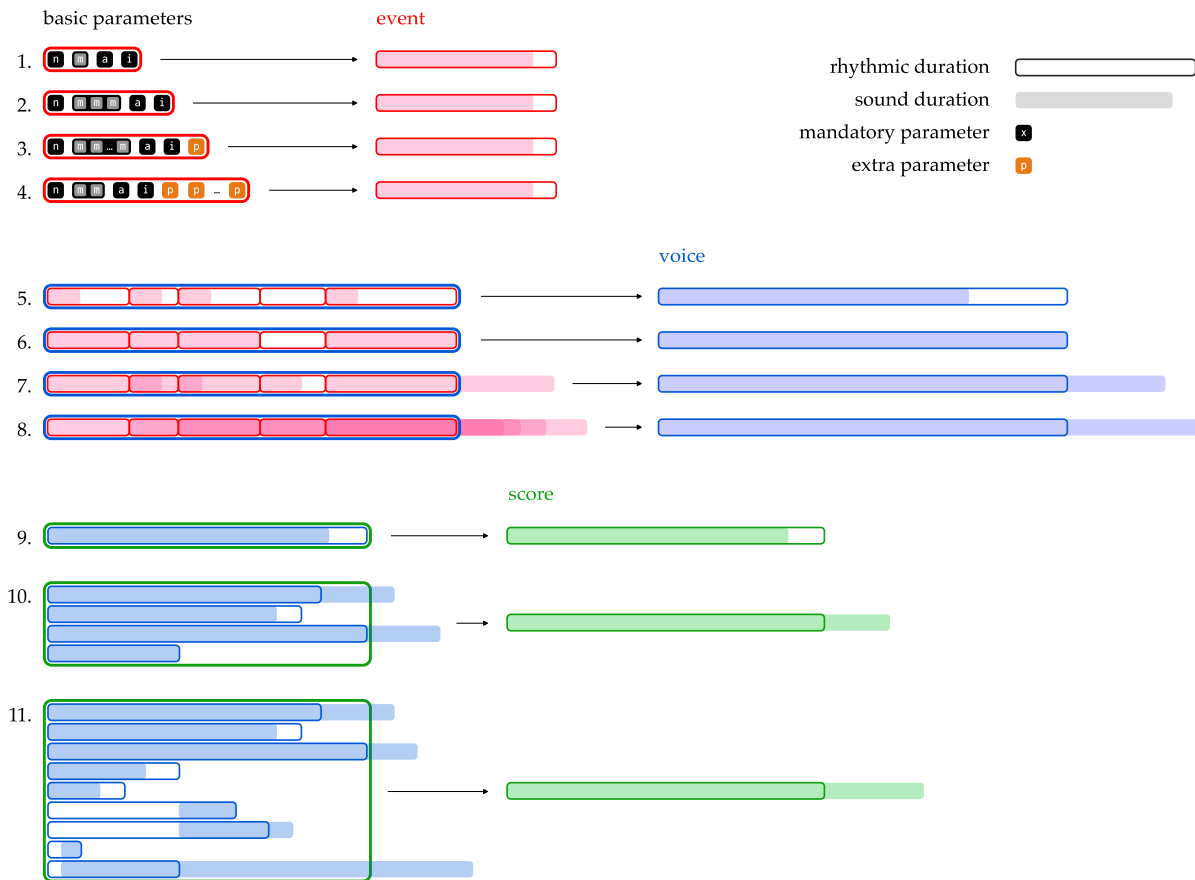


Figure 12: Constituent structures of a score. The three basic components, event, voice, and score, are represented in red, blue, and green, respectively. The shaded areas indicate the duration of sound events, while the edges with more intense color show the duration of the underlying rhythmic pattern. Often, both durations differ.

### Basic parameters wrapped as events

- An event contains four mandatory parameters: notevalue *n*, midipitch *m*, articulation *a* and intensity *i* (ex. 1).
- The midipitch parameter is actually a multi-parameter and can contain one or more values. This is very convenient when it's more appropriate to use the chord as an individual entity and not as separate voice events (ex. 2).
- Depending on the species, an event can contain one or more extra parameters. All extra parameters are generic parameters, and their meaning and mapping are user-defined (ex. 3 & 4).

### Rhythmic duration and sound duration

- As the compact representation of the events (in red) shows, there are two different durations for an event: the *rhythmic duration* is a nominal duration that determines when the next concatenated event starts, and the *sound duration*, which is the actual duration of the musical event, determined as a percentage of the rhythmic duration.
- Rhythmic duration and sound duration are properties that also apply to the voice and the score, determined by the maximum durations of each type of the elements that comprise them.
- The sound duration, determined by articulation, can be very different from an element's rhythmic duration. In Figures 12 and 13, the contours of an element define its rhythmic duration, while the color-filled content illustrates its sound duration.
- A blank section at the beginning of an element indicates the presence of one or more silent events intended to delay the onset of the first sonic event.

### Events wrapped as a voice

- A voice wraps one or more events sequentially concatenated in time, without gaps (ex. 5, 6, 7 & 8).
- An articulation = 0 will result in time gaps within a sequence, functioning as a rest (fourth event in ex. 5, 6, 7 & 8).
- Voice events with articulation < 100 allow sound durations to be less than rhythmic durations, functioning, for example, as *staccato* (ex. 5).
- In events with articulation = 100, rhythmic durations and sound durations are equal, producing a perfect *legato* effect (ex. 6).
- Voice events with articulation > 100 create an overlapping effect (ex. 7). Longer articulation values can generate dense clusters of sounds using only one voice (ex. 8).

### Voices wrapped as a score

- A score wraps one or more voices. All voices in a score start at time 0, but their rhythmic and sound durations can be different (ex. 9, 10 & 11).
- A voice in a score can contain silent events at the beginning, delaying its effective entrance (ex. 11).

## Merging scores

The horizontal juxtaposition of events to form a voice, as well as the vertical superposition of voices to create a score, are relatively straightforward processes. However, the formation of scores by combining other scores is a slightly more complex process, as it involves several cases, as represented in Figure 13.

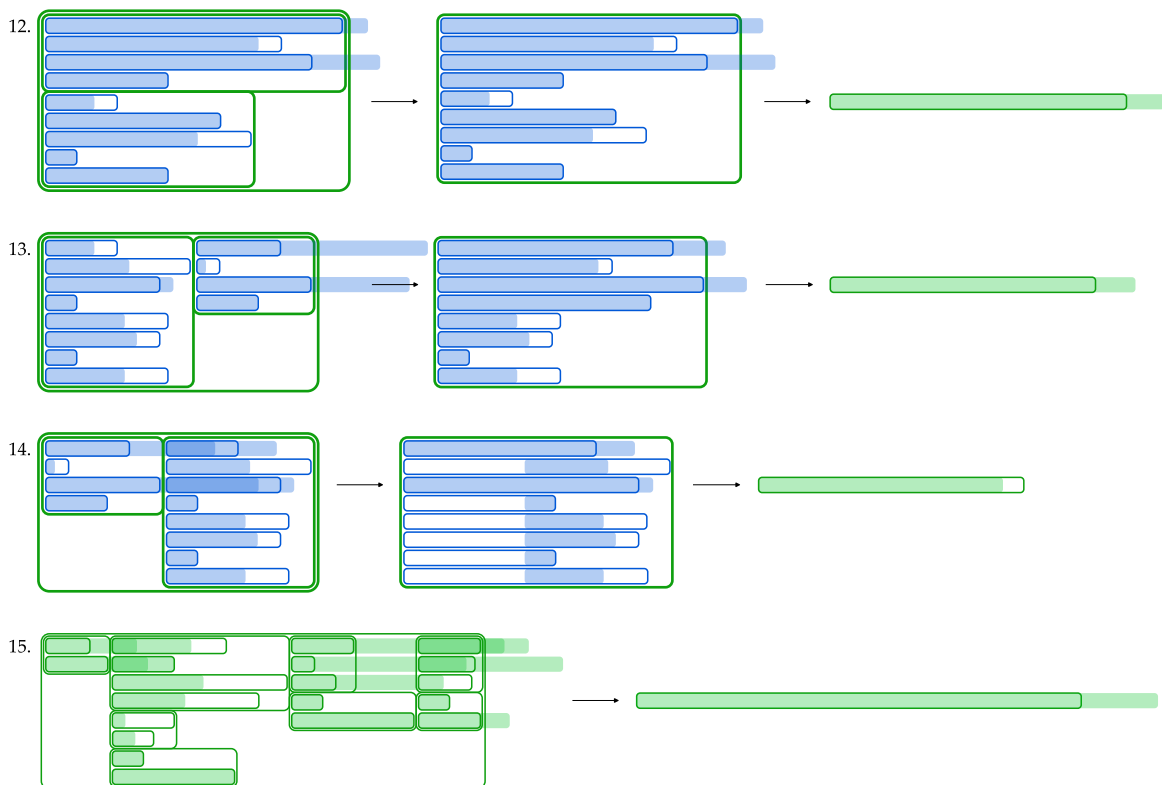


Figure 13: Visual depiction of merging scores

When a score is constructed as a result of merging two scores,  $A$  and  $B$ , each with  $m$  and  $n$  voices respectively, various scenarios can arise:

- $A$  and  $B$  merged vertically, producing one score with  $m + n$  voices, all starting at the beginning (ex. 12).
- $A$  and  $B$  merged sequentially, producing a new score with  $\max(m, n)$  voices, where voices from  $B$  start at the rhythmic end of  $A$ . This case can occur in two variants:

- $m \leq n$ : The resulting voices inside the new score result from joining each pair of voices from  $A$  and  $B$ . Voices from  $A$  without pairs remain untouched.
  - $m < n$ : The voices inside the new score result from joining each pair of voices from  $A$  and  $B$ . In voices from  $B$  without pairs some silent events must be added to fill the gaps, producing a coherent sequential result according to the rhythmic duration of  $A$ .
- The end of a score's sound duration is typically different from its rhythmic end. Merged scores may result in either silent gaps or sound overlapping, depending on the articulation of events within.

In summary, there are no limits to score merging. Since scores can be merged both horizontally and vertically, arbitrarily complex structures are possible (ex. 15). However, the final output will be a single score with several voices, each containing a number of events.

### 3.7. Representations of generated music

The primary output of GenoMus is its abstract format, purely numerical, as explained in Chapter 5. This format consists of a one-dimensional numerical sequence, which compactly encodes the musical voices and events of each generated score. This numerical representation is complemented by a human-readable JavaScript Object representation, containing explicit information about the parameters of each event, along with global metadata for the musical piece. Various other representations suitable for visualization, playback, and evaluation of the generated musical segments are derived from this internal format.

It is important to note that the musical output I aim to obtain is significantly different from a conventional score. The standard notation intended for *performance* is schematic: rhythmic values are identical, and articulation and dynamics are established flatly, often with only very general indications. A truly expressive interpretation builds upon this simplified representation—which is very convenient for reading and understanding the formal structure—to introduce all the subtleties, nuances, and *inaccuracies* that constitute music.

With this model, I try to capture not only the basic musical structure but also the interpretation, ensuring that it includes all the *desired inaccuracies* that make the result engaging and appealing. This poses a methodological challenge: the conventional notation of a score

becomes inadequate when trying to capture all those variations in dynamics, articulation, or duration present in the rendered output. An alternative form of representation must be sought.

As a compromise solution, the output is represented as a score, building upon conventional musical symbols but arranging them in a manner that can capture the subtle variations in each event. I use the paradigm that the bach library employs in the bach.roll viewer, while incorporating some new features. The computer-aided composition package bach for Max, by Agostini, and Ghisi [3], allows a simple and responsive visualization of experimental results as interactive music scores, which can also send data to synthesizers or music editors using MIDI, OSC, or any custom format. The data structure of the GenoMus phenotypes is close to the hierarchical structure of music employed by bach package [2], based on scores, voices, events, event multiparameters, and slots (applied to contain extra parameters). Genotypes' functional expressions are displayed in an embedded text editor, similar to the system created by Burton [27], also based on bio-inspired music patterns evolution.

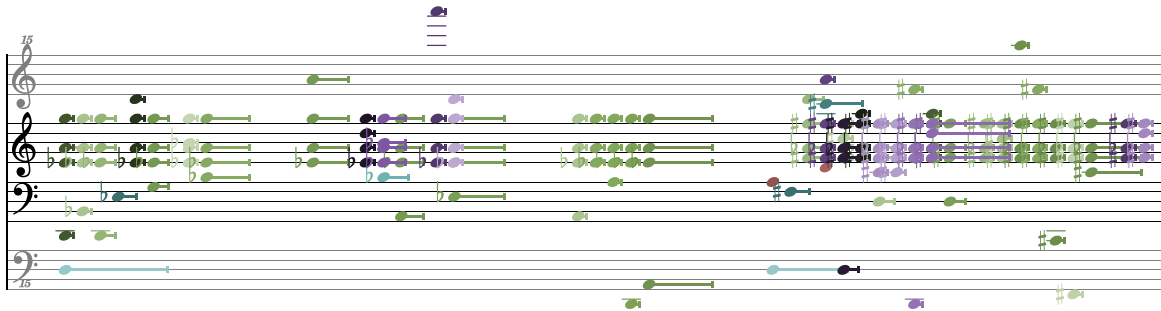


Figure 14: Example of a condensed SVG score. The duration of each event is represented with a line. Darker colors indicate louder dynamics. As an optional feature, different color hues are used to mark events belonging to different voices.

The bach.roll integrated into the patch allows for all kinds of subsequent modifications within the Max environment. However, for the final export, I have created a direct transformation from the textual representation to generate an SVG file with certain optional display features. Figure 14 shows the SVG export in condensed score mode, with all voices collapsed into a single system. On the other hand, in Figure 15, we can see the same music with the voices separated, which can be convenient in many cases.

The direct translation to conventional notation can be done from the bach environment itself. Depending on each situation, this may involve challenging adaptation scenarios that

require manual adjustments and *ad hoc* decisions on what information should be preserved in the score or how certain characteristics, such as degrees of randomness in dynamics and rhythmic precision, can be expressed with simple but clear verbal indications.

Note that the use of accidentals is unconventional. Since there are no bar lines or key signature indications, I follow the criteria found in many of Lutosławski's works: accidentals only affect the note they are placed on. For a sequence of several consecutive F#, all notes will carry the sharp symbol. However, unless altered for particular purposes, the automatic usage of sharps or flats is arbitrary and does not adhere to any specific key signature, prioritizing readability over diatonicity.

The image displays a musical score for three systems, each with a treble and bass staff. A horizontal timegrid is positioned at the top, with measures numbered from 0 to 18. The notes are represented by colored dots (red, green, blue) and horizontal lines indicating their duration. Accidentals (sharps and flats) are placed above or below notes. The notation is minimalist, focusing on pitch and rhythm without traditional bar lines or key signatures.

Figure 15: Example of multivoice output SVG score. This example shows the optional timegrid.

The examples of traditional notation in this text have been transcribed in this manner. A future addition could be to incorporate this translation to MusicXML notation directly from the core algorithm, without using bach as an intermediary.

Alongside the primary use of `bach.roll` to monitor the results of the experiments, a MIDI file is simultaneously created, enabling quick manipulation of these fragments with other programs such as VST instruments, score editors, DAWs, etc. This file can also serve as the main monitoring source, replacing real-time playback of `bach.roll`. This alternative output allows for improved rhythmic precision in cases of high polyphony or event density.



## Genotype functions

“

There is a lot of focus both among machine learning researchers and the general public about using machine learning and AI to replace people and duplicate human creative processes and I find this such a limiting viewpoint. From an artistic and humanist perspective, people derive a lot of value from making creative work, so we need to make tools that people actually use, where we are adding value to people’s lives, rather than replacing people.

Rebecca Fiebrink [67]

This chapter introduces the types of genotype functions in the current implementation of GenoMus. It then presents the internal structure of a genotype function’s declaration, along with the data structures it outputs. The text focuses on a select few functions, illustrating the main types and those with special characteristics requiring further clarification. The complete and updated documentation of all available genotype functions is available at <https://genomus.dev/genotype-functions>.

### 4.1. Identity functions

For each function type, there exists an identity function that simply passes its arguments without any musical transformation, apart from formatting the output data accordingly when necessary. For the sake of simplicity, identity functions are named using only their corresponding function type identifiers. They primarily serve as wrappers and testing functions.

For instance, for the eventF type, with the identifier e, the function e returns an event without making any modifications, simply serving as a wrapper for the data of its required parameters, which in this case are also identity functions for each specific type. The simple genotype e(n(1),m(69),a(100),i(100)) returns this subspecimen as a JavaScript Object:

```
{
  funcType: 'eventF',
  encGen: [ 1, 0.236068, 1, 0.09017, 0.51, 0.844042, 0, 1, 0.326238, 0.53, 0.608053,
    0, 1, 0.562306, 0.55, 0.613655, 0, 1, 0.18034, 0.56, 1, 0, 0 ],
  decGen: 'e(n(1),m(69),a(100),i(100))',
  encPhen: [ 0.844042, 0.618034, 0.608053, 0.613655, 1 ],
  phenLength: 1,
  phenVoices: 1,
  harmony: { root: 0.608053 }
}
```

Listing 11: Subspecimen returned by eventF identity function

Identity functions of different types, as those from previous example, can be replaced with generic parameters: the evaluation of e(p(0.844042),p(0.608053),p(0.613655),p(1)) yields the same phenotype encoded in the key encPhen as that in Listing 11.

```
{
  funcType: 'eventF',
  encGen: [ 1, 0.236068, 1, 0.123457, 0.5, 0.844042, 0, 1, 0.123457, 0.5, 0.608053,
    0, 1, 0.123457, 0.5, 0.613655, 0, 1, 0.123457, 0.5, 1, 0, 0 ],
  decGen: 'e(p(0.844042),p(0.608053),p(0.613655),p(1))',
  encPhen: [ 0.844042, 0.618034, 0.608053, 0.613655, 1 ],
  phenLength: 1,
  phenVoices: 1,
  harmony: { root: 0.608053 }
}
```

Listing 12: Subspecimen returned by eventF identity function with generic parameters

Identity functions are the first of their kind in the genotype function library listing. They are the only ones used when an automatically constructed function tree exceeds the maximum depth threshold. The genotype presented in Listing 13 has been constructed with a depth limit of 3. Consequently, starting from line 4, only identity functions are used for the required types until reaching the leaf values of the functional tree.



```

1 sHarmonicGrid(
2   sConcatS(
3     sConcatS(
4       s(
5         v(
6           e(
7             n(2.577),
8             m(28),
9             a(177),
10            i(100))))),
11      s(
12        v(
13          e(
14            n(0.9222),
15            m(101),
16            a(11),
17            i(100))))),
18    s(
19      v(
20        e(
21          n(0.24554),
22          m(82),
23          a(82),
24          i(100))))),
25    hJapanesePentatonicScale(
26      m(98)))

```

Listing 13: Indentity functions after reaching depth limit

## 4.2. Lists

Another fundamental element consists of lists of a specific parameter type. Similarly, the identifier of the function type is used to name the identity function within that category. Listing 14 shows the Object returned when evaluating the genotype `l(-0.3, 0.17, 0.42, 1.4)`, using the identity function `l` belonging to `listF` type. Notice how the parameters `-0.3` and `1.4`, outside the range `[0,1]` are rewritten. This could occur when manually entering code, as in the automatic generation of genotypes, this would never happen. It can also be observed that the values `0.17` and `0.42` in the input list remain the same in both `encGen` and `encPhen` keys since they are generic parameters to which no mapping is applied.

```
{  
  funcType: 'listF',  
  encGen: [ 1, 0.618034, 0.5, 0, 0.5, 0.17, 0.5, 0.42, 0.5, 1, 0 ],  
  decGen: 'l(0,0.17,0.42,1)',  
  encPhen: [ 0, 0.17, 0.42, 1 ],  
  phenLength: 1,  
  phenVoices: 1  
}
```

*Listing 14: Subspecimen returned by listF identity function*

The evaluation of the identity function `lm(60,62,64,65,66,68,70,71,-30,200)` clearly shows the performed conversions. Here, I have also introduced two values outside their range, the last two in the list, which are rewritten to stay within the allowed boundaries of a midipitch parameter.

```
{  
  funcType: 'lmidipitchF',  
  encGen: [ 1, 0.506578, 0.53, 0.430607, 0.53, 0.470107, 0.53, 0.509975, 0.53,  
    0.529894, 0.53, 0.54972, 0.53, 0.588857, 0.53, 0.62693, 0.53, 0.645439, 0.53, 0,  
    0.53, 1, 0 ],  
  decGen: 'lm(60,62,64,65,66,68,70,71,0,127)',  
  encPhen: [ 0.430607, 0.470107, 0.509975, 0.529894, 0.54972, 0.588857, 0.62693,  
    0.645439, 0, 1 ],  
  phenLength: 1,  
  phenVoices: 1  
}
```

*Listing 15: Subspecimen returned by lmidipitchF identity function*

### 4.3. Formal structures

Another type of elementary function is that which combines elements vertically or in juxtaposition to create formal structures. Listing 16 shows the implementation of the function `vConcatV`, which takes two voices as arguments and returns the resulting voice obtained by juxtaposing both.

```

1  indexGenotypeFunction("vConcatV", "voiceF", 43, ["voiceF", "voiceF"]);
2  vConcatV = (v1, v2) => {
3    var encodedFuncID = g2p(43);
4    var totalEvents = v1.phenLength + v2.phenLength;
5    if (totalEvents > phenMaxLength) {
6      validGenotype = false;
7      return indexDecGens(evalExpr("v(" + defaultEvent + ")"));
8    }
9    return indexDecGens({
10     funcType: "voiceF",
11     encGen: flattenDeep([1, encodedFuncID, v1.encGen, v2.encGen, 0]),
12     decGen: "vConcatV(" + v1.decGen + "," + v2.decGen + ")",
13     encPhen: [g2p(totalEvents)]
14               .concat((v1.encPhen).slice(1))
15               .concat((v2.encPhen).slice(1)),
16     phenLength: totalEvents,
17     phenVoices: 1,
18     harmony: v1.harmony,
19     timegrid: v1.timegrid,
20   })
21 };

```

Listing 16: Implementation of **vConcatV** function

Several observations on common characteristics in the implementation of many genotype functions:

- The variable `totalEvents` calculates the total number of events resulting from the concatenation of voices. Then, it is checked not to exceed the predefined maximum value. Otherwise, the genotype is declared unviable and returns a default expression.
- In line 12, the element `decGen` is formed from the same element of voices `v1` and `v2` taken as arguments. The auxiliary function `flattenDeep` flattens the array to make it one-dimensional and eliminate intermediate brackets.
- Each function returns a string with its own decoded genotype; in line 13, this self-expression is formed with the function name and by joining the inherited `decGen` elements from the arguments.

- The encoded genotype of a voice type starts with the golden encoded integer representing the number of contained events. This quantity is taken from `totalEvents` and converted using `g2p`. Then, it is concatenated with the `encPhen` of each voice, excluding the unnecessary first element.
- The self-analysis properties `harmony` and `rhythm` inherit characteristics from the first voice. For this function, these features are taken as primary.

The creation of a score can result from multiple operations of horizontal and vertical juxtaposition. Among the elementary formal functions, the most basic is `sConcatS`, whose purpose is the simple concatenation of two scores. Although its implementation is completely analogous to that of `vConcatV`, the situation is much more complex due to the circumstances shown in Figure 13. For this union, the auxiliary function `mergeScores` is called. It is worth reproducing its code in Listing 17 to visualize the details involved in each of the cases that may arise depending on the number of voices integrated by each score element. The comments are quite enlightening to understand the details of the concatenation procedure.

```

1 // aux functions to concatenate encoded phenotypes from two scores
2 var mergeScores = (scoEncPhen1, scoEncPhen2) => {
3   var sco1data = decodePhenotype(scoEncPhen1);
4   var numVoicesSco1 = p2g(scoEncPhen1[0]);
5   var numVoicesSco2 = p2g(scoEncPhen2[0]);
6   var maxVoices = Math.max(numVoicesSco1, numVoicesSco2);
7   var minVoices = Math.min(numVoicesSco1, numVoicesSco2);
8   var mergedPhenotype = [g2p(maxVoices)];
9   var largestVoiceDur = sco1data.metadata.rhythmicScoreDuration * 0.001;
10  var currentVoiceDur = 0;
11  var readCurrentVoiceDur;
12  var posSco1 = 1;
13  var posSco2 = 1;
14  var numEventsVoiceSco1 = 0;
15  var numEventsVoiceSco2 = 0;
16  var numPitchesEventVoiceSco1, numPitchesEventVoiceSco2;
17  var timeGap, wholeSecondEvents;
18  // joins common voices of two scores
19  for (var voice = 0; voice < minVoices; voice++) {
20    numEventsVoiceSco1 = p2g(scoEncPhen1[posSco1]);
21    numEventsVoiceSco2 = p2g(scoEncPhen2[posSco2]);
22    readCurrentVoiceDur = sco1data.metadata.rhythmicDurationsPerVoice[voice] * 0.001;

```

```

23     timeGap = largestVoiceDur - readCurrentVoiceDur;
24     wholeSecondEvents = Math.trunc(timeGap);
25     // updates total events in merged voice
26     if (timeGap > 0) {
27         mergedPhenotype.push(g2p(
28             numEventsVoiceSco1 + numEventsVoiceSco2 + wholeSecondEvents + 1));
29     }
30     else {
31         mergedPhenotype.push(g2p(numEventsVoiceSco1 + numEventsVoiceSco2));
32     }
33     // measures total voice dur to add a silent element to fill the voice gaps
34     currentVoiceDur = 0;
35     // copies first score voice
36     for (var event = 0; event < numEventsVoiceSco1; event++) {
37         posSco1++; mergedPhenotype.push(scoEncPhen1[posSco1]);
38         currentVoiceDur += p2n(scoEncPhen1[posSco1]);
39         posSco1++; mergedPhenotype.push(scoEncPhen1[posSco1]);
40         numPitchesEventVoiceSco1 = p2g(scoEncPhen1[posSco1]);
41         for (var pitch = 0; pitch < numPitchesEventVoiceSco1; pitch++) {
42             posSco1++; mergedPhenotype.push(scoEncPhen1[posSco1]);
43         }
44         posSco1++; mergedPhenotype.push(scoEncPhen1[posSco1]);
45         posSco1++; mergedPhenotype.push(scoEncPhen1[posSco1]);
46         for (var idx = 0; idx < eventExtraParameters; idx++) {
47             posSco1++; mergedPhenotype.push(scoEncPhen1[posSco1]);
48         }
49     }
50     // adds needed events to fill the gap until the end of first score block
51     // adds 1-second silent events (to enable big gaps no matter how long it is)
52     if (timeGap > 0) {
53         for (var silentEvent = 0; silentEvent < wholeSecondEvents; silentEvent++) {
54             mergedPhenotype = mergedPhenotype.concat([n2p(1 * playbackRate),
55                 0.618034, 0.95, 0.612134, 0]).concat(silentEventsExtraParams);
56         }
57         // adds a last silent element adding the remaining time left
58         mergedPhenotype = mergedPhenotype
59             .concat([n2p((timeGap - wholeSecondEvents) * playbackRate), 0.618034, 0.95,
0.612134, 0]).concat(silentEventsExtraParams);
60     }
61     // copies second score voice
62     for (var event = 0; event < numEventsVoiceSco2; event++) {
63         posSco2++; mergedPhenotype.push(scoEncPhen2[posSco2]);
64         posSco2++; mergedPhenotype.push(scoEncPhen2[posSco2]);
65         numPitchesEventVoiceSco2 = p2g(scoEncPhen2[posSco2]);
66         for (var pitch = 0; pitch < numPitchesEventVoiceSco2; pitch++) {
67             posSco2++; mergedPhenotype.push(scoEncPhen2[posSco2]);
68         }

```

```

69     posSco2++; mergedPhenotype.push(scoEncPhen2[posSco2]);
70     posSco2++; mergedPhenotype.push(scoEncPhen2[posSco2]);
71     for (var idx = 0; idx < eventExtraParameters; idx++) {
72         posSco2++; mergedPhenotype.push(scoEncPhen2[posSco2]);
73     }
74 }
75 posSco1++;
76 posSco2++;
77 }
78 // adds rest of voices if needed, distinguishing two cases: first score has more voices
    than second one and vice versa
79 if (numVoicesSco1 >= numVoicesSco2) {
80     // copies the remaining voices without changes
81     while (posSco1 < scoEncPhen1.length) {
82         mergedPhenotype.push(scoEncPhen1[posSco1]); posSco1++;
83     }
84 }
85 else {
86     wholeSecondEvents = Math.trunc(largestVoiceDur); // * playbackRate);
87     var numRemainingVoices = numVoicesSco2 - numVoicesSco1;
88     for (var voice = 0; voice < numRemainingVoices; voice++) {
89         // adds all needed silent events to start voices' events just after first score
90         numEventsVoiceSco2 = p2g(scoEncPhen2[posSco2]);
91         // increments voice length to include silent events at the beginning
92         mergedPhenotype.push(g2p(numEventsVoiceSco2 + wholeSecondEvents + 1)); posSco2++;
93         // adds one second silent events (to enable big gaps no matter how long it is)
94         for (var silentEvent = 0; silentEvent < wholeSecondEvents; silentEvent++) {
95             mergedPhenotype = mergedPhenotype
96                 .concat([n2p(1 * playbackRate), 0.618034, 0.95, 0.612134, 0])
97                 .concat(silentEventsExtraParams);
98         }
99         // adds a last silent element adding the remaining time left
100        mergedPhenotype = mergedPhenotype
101            .concat([n2p((largestVoiceDur - wholeSecondEvents) * playbackRate),
102                0.618034, 0.95, 0.612134, 0]).concat(silentEventsExtraParams);
103        for (var event = 0; event < numEventsVoiceSco2; event++) {
104            // adds duration
105            mergedPhenotype.push(scoEncPhen2[posSco2]); posSco2++;
106            numPitchesEventVoiceSco2 = p2g(scoEncPhen2[posSco2]);
107            // adds number of pitches
108            mergedPhenotype.push(scoEncPhen2[posSco2]); posSco2++;
109            // adds pitches
110            for (var pitch = 0; pitch < numPitchesEventVoiceSco2; pitch++) {
111                mergedPhenotype.push(scoEncPhen2[posSco2]); posSco2++;
112            }
113            // adds articulation
114            mergedPhenotype.push(scoEncPhen2[posSco2]); posSco2++;

```

```

115         // adds intensity
116         mergedPhenotype.push(scoEncPhen2[posSco2]); posSco2++;
117         for (var idx = 0; idx < eventExtraParameters; idx++) {
118             posSco2++; mergedPhenotype.push(scoEncPhen2[posSco2]);
119         }
120     }
121 }
122 }
123 return mergedPhenotype;
124 };

```

Listing 17: Implementation of `mergeScores` auxiliary function

The following genotype, in Listing 18, uses `sConcatS` to join two scores that, in turn, vertically overlay two voices each. Figure 16 displays the generated score, marking its breakdown into its elements.

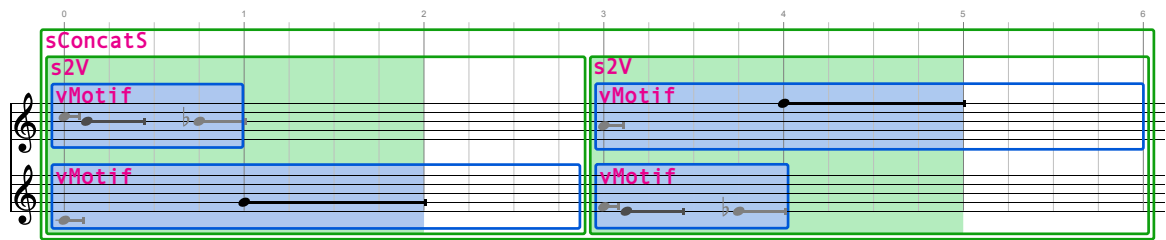
```

1 sConcatS(
2   s2V(
3     vMotif(
4       ln(0.125, 0.625, 0.25),
5       lm(72, 71, 70),
6       la(60, 50, 100),
7       li(50, 70, 50)),
8     vMotif(
9       ln(1, 2),
10      lm(60, 67),
11      la(10, 50),
12      li(50, 100))),
13   s2V(
14     vMotif(
15       ln(1, 2),
16       lm(69, 77),
17       la(10, 50),
18       li(50, 100)),
19     vMotif(
20       ln(0.125, 0.625, 0.25),
21       lm(65, 64, 63),
22       la(60, 50, 100),
23       li(50, 70, 50)))

```

Listing 18: Concatenated scores with `sConcatS`

genotype formal structure



phenotype flattened structure

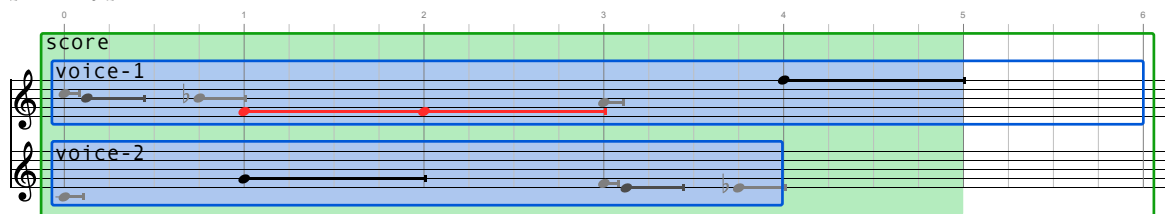


Figure 16: Simple score created with **sConcatS**. As in the preceding figures, blue indicates a voice structure, and green represents a score. The borderline indicates rhythmic duration, while the fill color indicates the time of effective sound. Notice how the concatenation of the second score created with **s2V** occurs at the limit of the rhythmic duration of the first, even if there are no sounding notes. As a voice is always a concatenation of events without gaps, in the joining process, silent events are created to fill the discontinuities. The red highlights indicate these newly introduced silent events (whose intensity = 0). Given that there is a maximum duration for events, as many one-second duration items (plus a fraction of a second if required) are added as needed to avoid that limitation.

## 4.4. Deterministic and random processes

To ensure precise tuning of the results, the generation of a phenotype must be a repeatable and deterministic process. Due to its dependency on each browser's implementation, JavaScript currently does not include its seeded random generator. To achieve repeatable randomness, several auxiliary functions have been incorporated. The deterministic generation of uniformly distributed values is accomplished using **mulberry32** and **mulberry32Local**,<sup>50</sup> For generating values with a Gaussian distribution, the auxiliary func-

<sup>50</sup>Both auxiliary functions are based on an adaptation of the Mulberry32 algorithm, adapting an implementation taken from <https://stackoverflow.com/questions/521295/seeded-the-random-number-generator-in-javascript>. It is not an exceedingly good generator for millions of values, but it is highly efficient in generation time, which is why it has been chosen. For the generation of music scores, it's rarely necessary to have such a high number of random values where the repetition cycle of this method can be perceived.



tion `gaussianRandLocal` adapts an implementation of the Box-Muller transform.<sup>51</sup> Some of the functions that introduce randomness are conditioned by a global seed value, which is part of the initial conditions documented in Section 5.5.

Genotype functions that generate more than one random value have their independent seed. For instance, `lRnd` and `lGaussianRnd`, which create random lists of generic parameters, take two arguments: a local seed value and the number of items in the list. To approximately visualize the distribution generated by these random generators, different outcomes of the expressions `lRnd(p(0.740253), q(31))` and `lGaussianRnd(p(0.740253), q(31))` are juxtaposed in Figure 17.

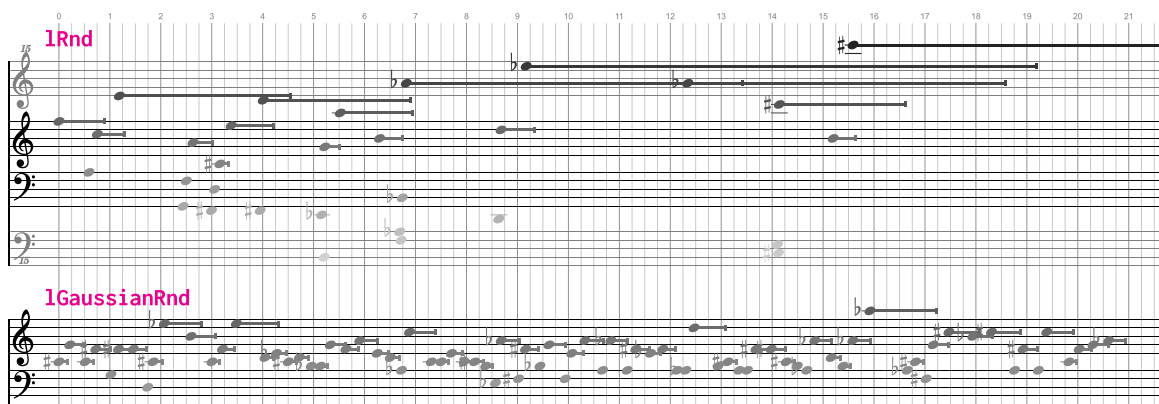


Figure 17: Random lists produced by `lRnd` and `lGaussianRnd`. When displaying the phenotype of generic parameters, the function `wrapEncodedPhenotype` employs them for all specific parameters, enabling comparison across all main dimensions of the event. Consequently, visualizing a larger generic parameter will result in longer duration, higher pitch, longer articulation, and increased dynamics. The Gaussian distribution combines its effect with the inherent Gaussian-like mapping of the specific parameters (discussed in Section 3.5), resulting in a significantly reduced dispersion around the central values.

Each local seed affects only the function to which it is an argument and is not influenced by the global seed of the specimen.

<sup>51</sup>Adapted from <https://stackoverflow.com/questions/25582882/javascript-math-random-normal-distribution-gaussian-bell-curve>.

## 4.5. Repetition and iteration

Numerous musical processes can be conceived as repetitions, variations, or transformations of any kind. Here, I will only discuss the fundamental details.

Undoubtedly, the simplest of these procedures is mere repetition. The key distinction to be made here is between the repetition of an element and the repetition of a process, referred to as iteration. Let's compare two very similar genotypes in Figure 18, where two parallel processes are depicted: in red, the result of executing `vRepeatE`, which *repeats an event* several times; in blue, the application of `vIterE`, which *iterates the subgenotype of an event*. In both cases, the event contains random elements. While in repetition, the random characteristics are repeated, in iteration, they are recalculated each time, resulting in different events.

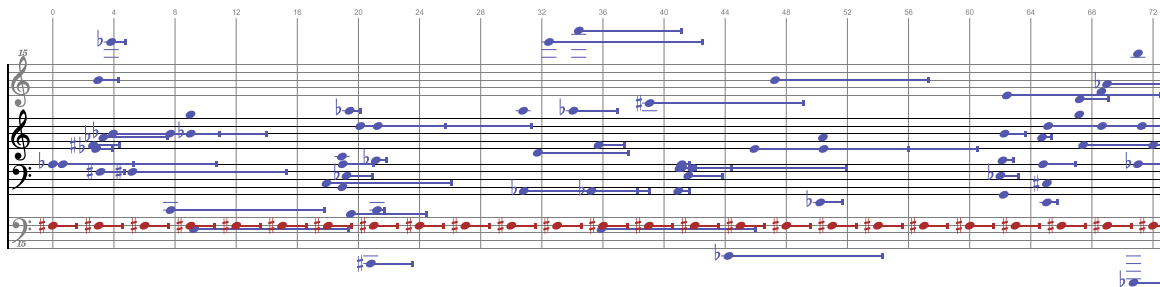


Figure 18: Comparison between repetition and iteration of the same event. The red voice has been generated using the genotype `vRepeatE(e(nRnd(), mRnd(), a(50), i(60)), q(70))`. Once an event with random duration and pitch has been generated, it is repeated identically. In blue, the result of executing `vIterE(e(nRnd(), mRnd(), a(800), i(60)), q(70), p(0.26))`, which iterates the reevaluation of an event, also with random duration and pitch. It can be observed that the intensity, represented by the color brightness, remains fixed. The articulation (represented by horizontal lines) is also constant, although since it depends on the duration, the lines have different lengths; `vIterE` requires an additional parameter which will be the seed value for the random regeneration of the event.

The repetition is a straightforward operation within the genotype function. However, iteration involves the juxtaposition of a series of reevaluations of the subgenotype taken as an argument. In Listing 19, with the implementation of the function `vIterE`, it can be observed how this subprocess is carried out. In line 13, we encounter the expression `evalExpr(event.decGen).encPhen`: first, it takes the key `decGen`, which holds the subgenotype's own expression as a string, evaluates it using `evalExpr`, and from the generated subspecimen, it retrieves the `encPhen` key, where the phenotype generated by the reevaluation is found.

```

1 indexGenotypeFunction("vIterE", "voiceF", 37, ["eventF", "quantizedF", "paramF"]);
2 vIterE = (event, iterations, seedValue) => {
3   var encodedFuncID = g2p(37);
4   var numIterations = adjustRange(Math.abs(p2q(iterations.encPhen[0])), 2, 1000);
5   // number of times rescaled to range [2, 1000], mapped according to the
6   // deviation from the center value 0.5 using the quantizedF map
7   if (numIterations > phenMaxLength) {
8     validGenotype = false;
9     if (verbosity) printLog("aborted genotype due to exceeding the max length");
10    return indexDecGens(evalExpr("v(" + defaultEvent + ")"));
11  }
12  reinitSeed(seedValue.encPhen[0]);
13  var iteratedEvent = [g2p(numIterations)].concat(flattenDeep(Array(numIterations).
14  fill().map(() => evalExpr(event.decGen).encPhen)));
15  return indexDecGens({
16    funcType: "voiceF",
17    encGen: flattenDeep([1, encodedFuncID, event.encGen, iterations.encGen,
18    seedValue.encGen, 0]),
19    decGen: "vIterE(" + event.decGen + "," + iterations.decGen + "," +
20    seedValue.decGen + ")",
21    encPhen: iteratedEvent,
22    phenLength: numIterations,
23    phenVoices: 1,
24    rhythm: event.rhythm,
25    timegrid: event.timegrid,
26    analysis: event.analysis
27  });
28 };

```

Listing 19: Implementation of subgenotype iteration **vIterE**

Let's consider one more example to clarify how the reevaluation of an expression caused by an iterative function works. In Listing 20, the function **lIterL** reevaluates 16 times the subgenotype **l3P(p(0.37), p(0.41), pGaussianRnd())**, of type listF. In this subgenotype, there are two fixed values. However, the function **pGaussianRnd()** will generate different random parameters in each iteration. The additional parameter in line 7 is the seed value that allows repeatability of the entire process.

```

1 lIterL(
2   l3P(
3     p(0.37),
4     p(0.41),
5     pGaussianRnd()),
6   q(16),
7   p(0.21827))

```

Listing 20: Iteration of lists with **lIterL**

In the musical score of Figure 19, the effect of these reevaluations can be easily observed. The melodic line shows the cycle of two fixed pitches plus a third that varies in each iteration.

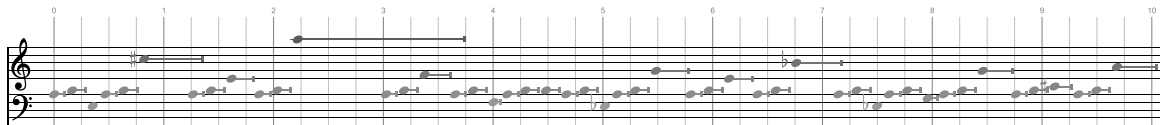


Figure 19: Iteration of a list containing a random item. Similar to previous examples, the conversion to sheet music of unassigned generic parameters is done by applying them to all dimensions of the event. Therefore, even though the three-element cycle in varied repetition can be mainly observable in pitch changes, it is also reflected in duration, dynamics, and articulation.

## 4.6. Harmony

The function type `harmonyF` belongs to the category of specific functions dedicated to covering a particular aspect of constructing musical structures. In programming languages for music, there are numerous paradigms, nomenclatures, and different approaches for dealing with harmony. As in other aspects of GenoMus design, the architecture allows for a wide variety of treatments while also facilitating the most common uses. Nevertheless, this section presents just one way to organize harmonic structure. Nothing prevents the incorporation of other `harmonyF` functions to provide alternative treatments.

The central concept here is the *harmonic grid*, defined as the set of pitches that the pitch parameter of events will adhere to. A harmonic grid can affect a portion or an entire musical piece. To compute this set of eligible pitches, various hierarchies of harmonic organization are taken into consideration, interacting with each other to determine this

pitch set as an array under the variable `harmonicGrid`. Table 6 displays and explains the arguments required to create a harmonic grid.

| Key          | Description   | Function type |
|--------------|---|---------------|
| tuning       | <b>Tuning employed.</b> Set of all pitches available within the octave. This set precisely specifies the temperament using floating-point numbers. The remaining keys use integers to ultimately refer to the position occupied by each pitch in this set, so that the <code>harmonicGrid</code> ultimately adjusts them to this tuning.                                      | lmidipitchF   |
| scale        | Set of <b>all eligible pitches</b> to form modes.   | lmidipitchF   |
| mode         | Subset of <b>eligible pitches from the scale</b> , typically used to build diatonic sequences.  | lmidipitchF   |
| chord        | Subset of <b>eligible pitches from the mode</b> , usually used to construct chords or aggregates of a few different pitches.  | lmidipitchF   |
| root         | <b>Fundamental pitch</b> to which the <code>harmonicGrid</code> is transposed.  | midipitchF    |
| chromaticism | Level of <b>permissiveness in using pitches outside the chord</b> . If <code>chromaticism = 0.5</code> , strictly only the chord values are used; if <code>&lt; 0.5</code> , notes are progressively reduced until unison (0); if <code>&gt; 0.5</code> , more notes are introduced, first from the mode and then from the scale, until all eligible pitches (1) are reached. | paramF        |
| octavation   | <b>Range extension</b> in which pitches can be chosen. If <code>octavation = 0.5</code> , only one octave is used. Higher or lower values increase the range towards higher or lower octaves, respectively.   | paramF        |

Table 6: Arguments to configure a `harmonicGrid`. For more detailed information on how it is precisely calculated, please check the implementation of the auxiliary function `calculateHarmonicGrid` in the source code.

The identity function `h` generates a harmonic grid and returns it within a subspecimen containing this information under the key `harmony`. Consider, for example, the expression in Listing 21:

```

1 h(
2   lm(0,1,2,3,4,5,6,7,8,9,10,11),
3   lm(0,1,2,3,4,5,6,7,8,9,10,11),
4   lm(0,2,3,5,7,9,11),
5   lm(0,3,7,9,11),
6   m(40),
7   p(0.5),
8   p(0.7))

```

Listing 21: Identity function **h** of harmonyF type

This will return the Object shown in Listing 22, where, in addition to the harmonicGrid itself, the arguments that originated it are stored, so they can be used by the parent functions if necessary.

```

1 {
2   funcType: 'harmonyF',
3   encGen: [ 1, 0.652476, 1, 0.506578, 0.53, 0, 0.53, 0.000552, 0.53, 0.001148, 0.53,
4     ... 92 more items ],
5   decGen: 'h(lm(0,1,2,3,4,5,6,7,8,9,10,11),lm(0,1,2,3,4,5,6,7,8,9,10,11),lm
6     (0,2,3,5,7,9,11),lm(0,3,7,9,11),m(40),p(0.5),p(0.7))',
7   encPhen: [ 0.130886, 0.161479, 0.210422, 0.238502, 0.268956, 0.285042, 0.336467,
8     0.411143, 0.45028, 0.490026, 0.509975, 0.569394, 0.645439, 0.681175, 0.714958,
9     0.731044, 0.775839, 0.82718, 0.849282, 0.869114, 0.878216 ],
10  phenLength: 1,
11  phenVoices: 1,
12  harmony: {
13    tuning: [ 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11 ],
14    scale: [ 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11 ],
15    mode: [ 0, 2, 3, 5, 7, 9, 11 ],
16    chord: [ 0, 3, 7, 9, 11 ],
17    root: 40,
18    chromaticism: 0.5,
19    octavation: 0.7,
20    harmonicGrid: [ 40, 43, 47, 49, 51, 52, 55, 59, 61, 63, 64, 67, 71, 73, 75, 76,
21      79, 83, 85, 87, 88 ]
22  }
23 }

```

Listing 22: Subspecimen of a harmonyF function

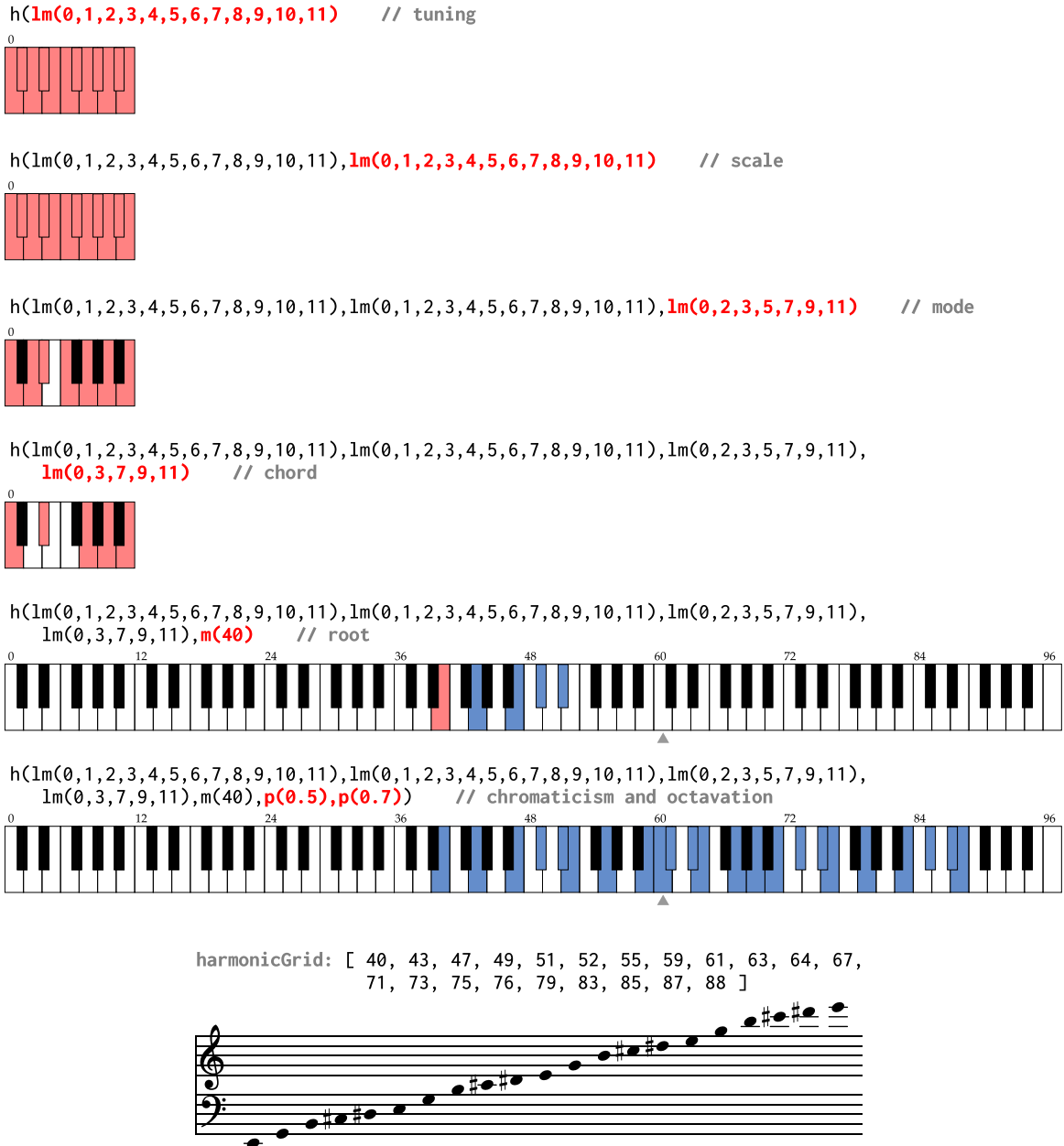


Figure 20: Steps to assemble a harmonicGrid. In red, pitches specifically indicated in the arguments of the function **h**. In blue, pitches derived from the different transformation steps up to the final set of eligible pitches. In this example, after defining the standard equal-tempered tuning and the complete chromatic scale as available pitches, the mode configures a harmonic minor scale, and the chord restricts it to only specific notes. Everything is transposed to the MIDI note 40 (E-2), and the values of chromaticism and octavation determine the use of all notes from the subset and an extension of three octaves upward.

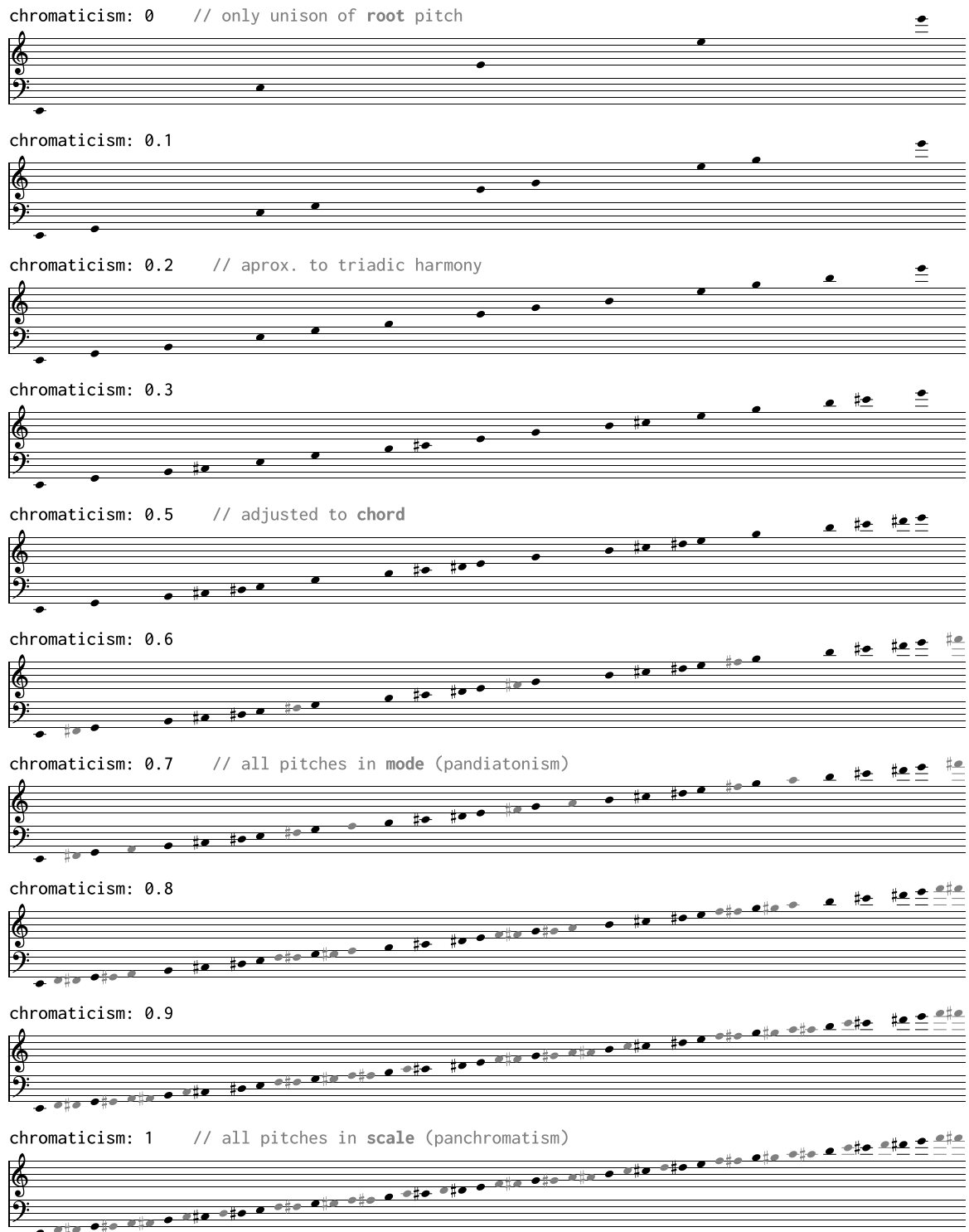


Figure 21: Influence of the degree of chromaticism in the harmonicGrid of Figure 20. In black, pitches specific to the chord; in gray, pitches that fill the spaces towards maximal chromatic density.



Figure 20 details the step-by-step process until determining the harmonicGrid. Meanwhile, Figure 21 illustrates the influence that the parameter chromaticism exerts on the selection of the set of pitches that will ultimately be eligible. Interestingly, this parameter is fluid since in much music, it plays with the concept of real versus strange notes of a harmony, distinguishing between the pitches that make up a recognizable chord or scale and the sounds that deviate from these structures, either for melodic reasons or to introduce tension through dissonance at various levels of disruption.

Leveraging the auxiliary function `harmonicGridFramework`, it is easy to obtain the harmonic grid of the desired chord, mode, or scale. A family of `harmonyF` functions has been incorporated, which already returns harmonic grids for the main types of diatonism and chords. Listing 23 shows the definition of `hOctatonicScale`, which creates the grid for the semitone-tone scale across the entire range.<sup>52</sup> The only parameter of this function is `root`, which determines the initial pitch class.

```
1 indexGenotypeFunction("hOctatonicScale", "harmonyF", 179, ["midipitchF"]);
2 hOctatonicScale = (root) => harmonicGridFramework(
3   "hOctatonicScale",
4   g2p(179),
5   lm(0,1,3,4,6,7,9,10,11),
6   root);
```

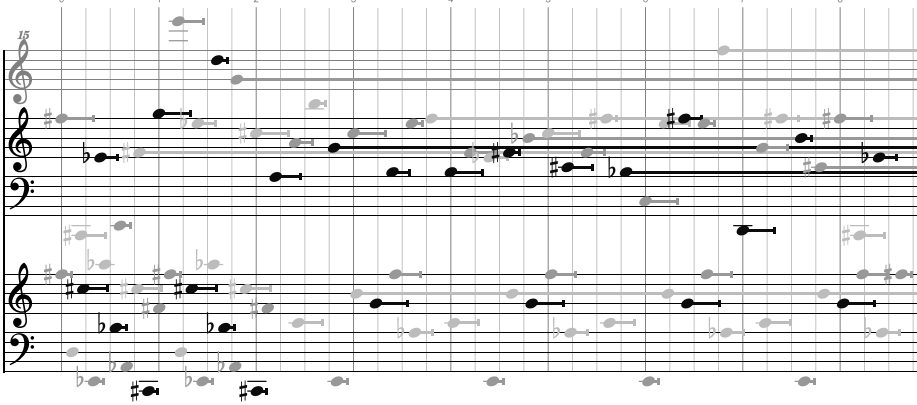
*Listing 23: Definition of a function to create harmonic grids with octatonic scales*

In Figure 22, the result of readjusting all the notes of a score to a harmonic grid can be seen. In this case, `sHarmonicGrid` takes a score and a `harmonicGrid` as arguments, and returns the same music adapted to the new harmony, which in the example is a simple pentatonic scale generated with `hPentatonicScale`.

---

<sup>52</sup>Also known as the second of Messiaen's modes of limited transposition.

<scoreF\_genotype>



sHarmonicGrid(<scoreF\_genotype>, hPentatonicScale(m(10)))

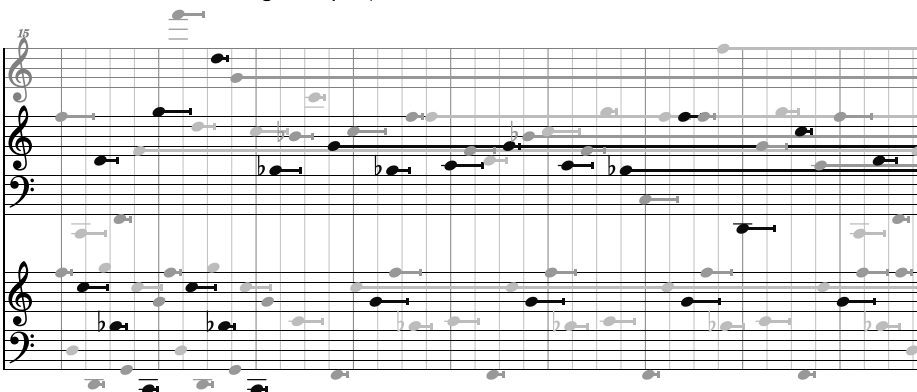


Figure 22: Adjustment of an entire score to a pentatonic scale with **sHarmonicGrid**. Two identical excerpts, except that in the second one, all pitches have been remapped to B $\flat$  major pentatonic.

Other ways of easily creating harmonic structures are based on pitch class set theory.<sup>53</sup> The function **hPCSet** takes a pitch class set (PCS) and a pitch as the root to generate a harmonic grid across the entire range. Listing 24 combines two voices with nearly identical processes. Both involve the iteration of random four-note chords, and the resulting output is passed as an argument to **vHarmonicGrid**, which then adjusts it to a PCS. Both voices use the same PCS,  $\{0, 6, 11\}$ , but in the second voice it is transposed up by 3 semitones, resulting in the new set  $\{3, 9, 14\} \bmod 12 \equiv \{3, 6, 9\}$ .

<sup>53</sup>Pitch class set theory is a method of musical analysis used to analyze and categorize compositions based on their pitch content. This theory considers the pitch classes of notes, often ignoring aspects like octave placement, rhythm, and timbre. It's particularly useful in analyzing atonal music, where traditional harmonic analysis might not be applicable. The theory groups pitches into sets and examines the relationships and structures within these sets.

```

1 s2V(
2   vHarmonicGrid(
3     vIterE(
4       e4Chord(n(0.2), mRnd(), mRnd(), mRnd(), mRnd(), aRnd(), iRnd()),
5       q(24),
6       pRnd()),
7     hPCSet(
8       lm(0, 6, 11),
9       m(0))),
10  vHarmonicGrid(
11    vIterE(
12      e4Chord(n(0.2), mRnd(), mRnd(), mRnd(), mRnd(), aRnd(), iRnd()),
13      q(20),
14      pRnd()),
15    hPCSet(
16      lm(0, 6, 11),
17      m(3))))

```

Listing 24: Genotype with two simultaneous harmonic grids based on pitch class sets

The image displays two systems of musical notation, each representing a different transposition of a pitch class set. The top system is titled 'hPCSet(0,6,11) // C, F#, B' and the bottom system is titled 'hPCSet(2,3,9) // D, Eb, A'. Each system consists of multiple staves (treble and bass clefs) with horizontal lines indicating the duration of events and vertical lines indicating the pitch classes. The notation is complex, showing a dense arrangement of notes and rests across the staves.

Figure 23: Score with two transpositions of a pitch class set as harmonic grids. Two similar voices, in the second one, all pitches have been remapped to B $\flat$  major pentatonic. There is also a 4 : 5 relationship in the duration of the events in each voice, resulting in observable polyrhythm. Let's remember that the internal rhythmic duration is independent of the effective duration resulting from applying the articulation parameter: in each voice, all chords appear in a perfectly regular rhythm, although their audible durations, indicated by the horizontal lines, vary randomly.

Finally, it should be noted that there exists a global playback option that affects the pitches of an entire phenotype. The playbackOption `stepsPerOctave` determines the division of the octave in equal intervals. The default value is `stepsPerOctave = 12`, representing the usual chromatic scale. By combining this value and the pitch precision of the `bach.roll` where the score is displayed, interesting effects can be achieved, as it will be explored more thoroughly in Section 6.4.3.

## 4.7. Generative subprocesses

Besides a declarative style more typical of traditional writing techniques, the implementation of generative processes, very common since the mid-20th century, is straightforward. The combination of traditional and algorithmic procedures is among the initial objectives of this implementation. As a proof of concept, I tested the integration of some of the already classical algorithms for generative music here.

Brownian motion creates random paths from an initial point based on displacements of a defined maximum amplitude. The first two lines in the definition of the function `lBrownian` in Listing 25 are quite self-explanatory in describing the parameters it uses and the required types:

```
1 indexGenotypeFunction("lBrownian", "listF", 266, ["paramF", "paramF", "quantizedF",  
   "paramF"]);  
2 lBrownian = (start, maxStep, numSteps, seedValue) => {
```

Listing 25: Beginning of the definition of the generative function `lBrownian`

Given that the function is of type `paramF` and there are no conversions to specific types, by accessing the key `encPhen` of the subspecimen, we can print the generated sequence in the console by evaluating `lBrownian(p(0.5), p(0.1), q(100), p(0.1234)).encPhen`, which returns `[0.5, 0.514419, 0.421108, 0.422797, 0.509397, ... 95 more items ]`.

To avoid creating different versions of these generative functions for each specific parameter, I added converters that take a list of generic parameters and encapsulate them into lists of the specific type needed. Listing 26 contains the function `lConverterFramework` that allows creating all list conversions. In lines 10 and 11, the function `lnWrap` is declared; it wraps a generic list as a list with values of type `notevalueLeaf`.

```

1 var lConverterFramework = (functionName, functionTyp, functionIndex, paramListFunc)
  => indexDecGens({
2   funcType: functionTyp,
3   encGen: flattenDeep([1, g2p(functionIndex), paramListFunc.encGen, 0]),
4   decGen: functionName + "(" + paramListFunc.decGen + ")",
5   encPhen: paramListFunc.encPhen,
6   phenLength: 1,
7   phenVoices: 1
8 });
9
10 indexGenotypeFunction("lnWrap", "lnotevalueF", 319, ["listF"]);
11 lnWrap = (paramList) => lConverterFramework("lnWrap", "lnotevalueF", 319, paramList)

```

Listing 26: Framework auxiliary function to create list converters

Let's see an example of usage, creating a motif of 100 events in which all dimensions are generated by independent Brownian sequences. In Listing 27, it can be observed that the seed value is generated by **pRnd**, therefore, it will be different for each call to **lBrownian**. The value of these seeds is determined by the global seed value of the initial conditions. Figure 24 compares three fragments generated with the same genotype but with different seed values.

```

1 vMotif(
2   lnWrap(lBrownian(p(0.5), p(0.2), q(100), pRnd())),
3   lmWrap(lBrownian(p(0.5), p(0.2), q(100), pRnd())),
4   laWrap(lBrownian(p(0.5), p(0.2), q(100), pRnd())),
5   liWrap(lBrownian(p(0.5), p(0.2), q(100), pRnd())))

```

Listing 27: **vMotif** with **lBrownian** generative functions

The Equation 26 is the logistic map, a classic recursive equation in the study of chaotic deterministic systems:<sup>54</sup>

$$x_{n+1} = rx_n(1 - x_n) \quad (26)$$

where  $r \in (0, 4)$  is a constant that determines the degree of apparent randomness.

<sup>54</sup>Unlike Brownian motion, this is an intrinsically repeatable process, even though it may seem random in its evolution, and in fact, it can be used as a pseudo-random process generator that allows a gradual transition from monotonic repeatability to chaos.

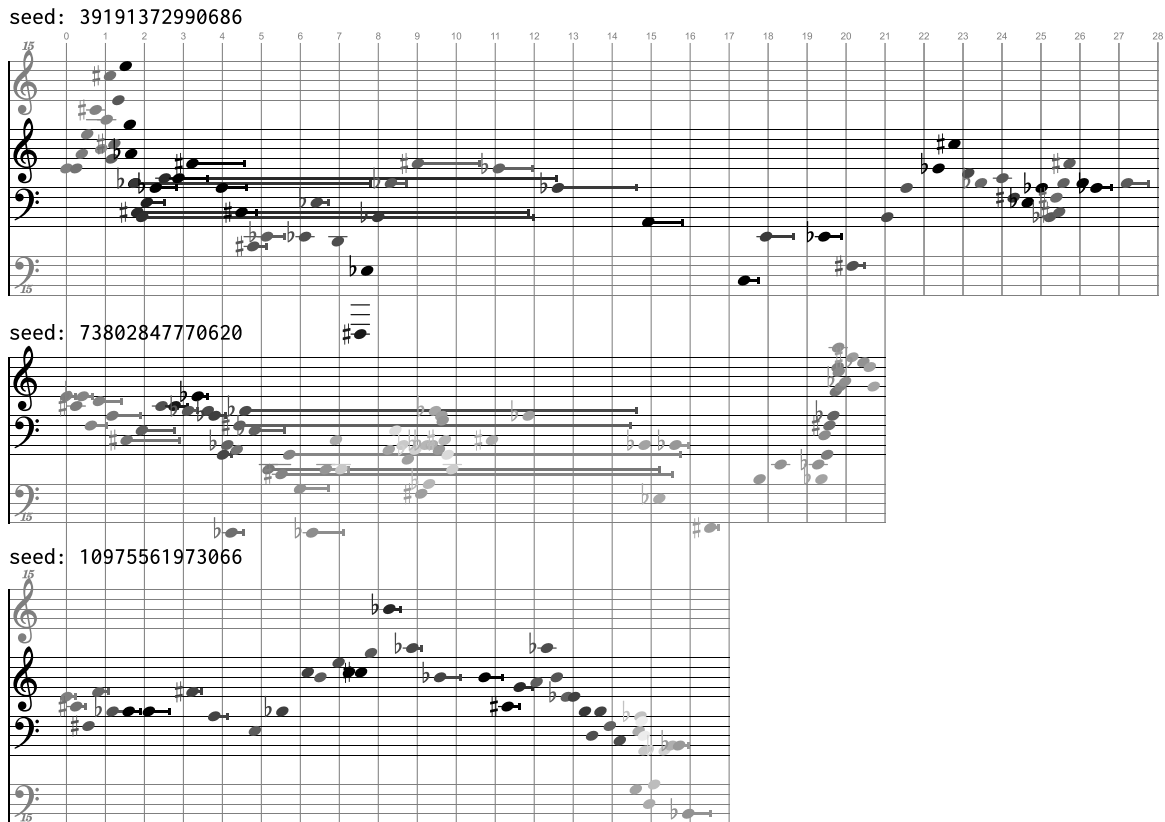


Figure 24: Three versions of a **vMotiv** with **lBrownian** generative functions using different seed values. It can be observed how all fragments start with the same initial event, and from there, each of their parameters varies according to different Brownian movements.

Listing 28 displays part of the implementation of **lLogisticMap**. The last argument `paramF` represents this constant  $r$  that determines how chaotic the generated sequence will be. Since the interesting behavior of the system occurs when  $r \in (3.5, 4)$ , in line 4 the normalized range of the generic parameter is rescaled. These internal remappings of input and output of generic parameters will be necessary for many generative processes.

```

1  indexGenotypeFunction("lLogisticMap", "listF", 274, ["paramF", "paramF", "paramF",
   "quantizedF", "paramF"]);
2  lLogisticMap = (start, rangeMin, rangeMax, numSteps, growConstant) => {
3    ...
4    var rRemapped = rescale(growConstant.encPhen[0], 0, 1, 3.5, 4); // only uses
   chaotic output of equation
5    ...

```

```

6   for (var lmstep = 0; lmstep < totalSteps; lmstep++) {
7       chaosLine.push(rRemapped * chaosLine[lmstep] * (1 - chaosLine[lmstep]));
8       chaosLine[lmstep] = r6d(rescale(chaosLine[lmstep], 0, 1, rangeMin.encPhen[0],
9           rangeMax.encPhen[0]));
9   }
10  ...
11 }

```

*Listing 28: Details of the implementation of a generative functions*

Listing 29 shows a variation of the genotype in Listing 27. The  $r$  value of each evaluation of the **lLogisticMap** is determined by the global seed. Figure 25 compares six different phenotypes produced by this same genotype by varying that initial condition. The sequences demonstrate the tendency of this recursive process to create cycles with different degrees of irregularity and moments of disruption.

```

1  vMotif(
2      lnWrap(lLogisticMap(p(0.5), p(0.1), p(0.9), q(100), pRnd())),
3      lmWrap(lLogisticMap(p(0.5), p(0.1), p(0.9), q(100), pRnd())),
4      laWrap(lLogisticMap(p(0.5), p(0.1), p(0.9), q(100), pRnd())),
5      liWrap(lLogisticMap(p(0.5), p(0.1), p(0.9), q(100), pRnd()))

```

*Listing 29: vMotiv with lLogisticMap generative functions*

## 4.8. Recursions with type recursiveF

Both examples from the previous section are recursive processes. However, they are predefined algorithms, merely operating based on their arguments. The specific type of genotype function, `recursiveF`, enables the metaprogramming mechanism itself to compose new recursive equations by combining basic mathematical operations, thus opening up a wide field of experimentation.<sup>55</sup> To make this possible, it is necessary to create this new family of functions with peculiar characteristics in handling their subspecimens.

---

<sup>55</sup>The exploration of recursion through metaprogramming of abstract mathematical expressions lies at the core of the GenoMus project. The initial proof of concept for the general idea consisted of a prototype that explored melodic and harmonic material from generatively created recursive equations. The quintet with electronics, *Threnody for Dimitris Christoulas*, documented in Appendix B.1, is the practical realization of this preliminary investigation.

Similarly to what was seen earlier, recursive functions are created within `listF` functions that contain them, and they output a sequence of generic parameters that will then be applied to specific dimensions of events through the proper converters.



Figure 25: Sequences generated with `lLogisticMap`. Once again, the six fragments are phenotypes generated from the same genotype by varying the global seed value, which in turn determines the constant of the logistic map.

Listing 30 shows the implementation of `lRecursioOrder2`, which takes as the first argument a recursive equation, the next two arguments are the initial items of the sequence, which will be referenced within the equation itself, and finally, the last argument, quantifying how many iterations will be calculated.

```

1 indexGenotypeFunction("lRecursioOrder2", "listF", 1004, ["recursiveF", "paramF",
2   "paramF", "quantizedF"]);
3 lRecursio2ndOrder = (recursiveExpression, firstTerm, secondTerm, totIterations) => {
4   var encodedFuncID = g2p(1004);

```



```

4   initTermsLength = 2;
5   var totalNewTerms = p2q(totIterations.encPhen) + initTermsLength;
6   terms = [
7     p2pi(firstTerm.encPhen[0]),
8     p2pi(secondTerm.encPhen[0])
9   ];
10  var stopRecursion = false;
11  var idx = initTermsLength;
12  do {
13    newTerm = modPi(evalExpr(recursiveExpression.recursiveExpr));
14    if (isNaN(newTerm)) { // in case a NaN is produced, recursion ends
15      stopRecursion = true;
16    }
17    else {
18      terms.push(newTerm);
19    }
20    idx++;
21  } while (
22    stopRecursion == false
23    && idx < totalNewTerms
24    && arrayEquals(
25      [terms[idx - 1]].map(x => r2d(x)),
26      [terms[idx - 2]].map(x => r2d(x))
27    ) == false
28  ); // avoids long stacked sequences
29  return indexDecGens({
30    funcType: "listF",
31    encGen: flattenDeep([1, encodedFuncID, recursiveExpression.encGen,
32      firstTerm.encGen,
33      secondTerm.encGen,
34      totIterations.encGen, 0]),
35    decGen: "lRecursio2ndOrder(" + recursiveExpression.decGen + "," +
36      firstTerm.decGen + "," +
37      secondTerm.decGen + "," +
38      totIterations.decGen + ")",
39    encPhen: terms.map(x => r6d(pi2p(x))),
40    phenLength: 1,
41    phenVoices: 1,
42  });
43 };

```

Listing 30: Function `lRecursioOrder2` with `recursiveF` argument

Let's point out these key features:

- For this entire family of functions, by convention, the series of calculated values is stored in an array within the global variable `terms`, external to the function itself. Thus, to refer to the current penultimate value of the sequence, for example, the expression `terms[terms.length - 2 % initTermsLength - 1]` should be used. The global variable `initTermsLength` is used to adjust any value referring to a previous term of the sequence applying the modulo operator.
- The function `r` functions as the leaf value of these recursive expressions, but actually takes an integer argument through a `quantizedF` function, indicating which previous element of the stored sequence is taken for the reevaluation of the equation (at line 13).
- The initial values of each sequence, provided here by the arguments `firstTerm` and `secondTerm`, will be the first elements of the array `terms`, but first, a value conversion occurs using `p2pi`. This conversion is a remapping from the normalized range  $[0, 1]$  to the interval  $[-\pi, \pi]$ . This is done so that trigonometric functions, relevant in obtaining sequences with interesting patterns, can operate in their full range. At line 39, the inverse conversion of the entire sequence is performed using `pi2p` to return to the normalized range specific to the generic parameters.

The subspecimens of the `recursiveF` functions are coordinated with this internal architecture. For instance, the function `r(q(2))` indicates the penultimate item of the recursive sequence and, unlike the rest of the types, returns in the key `recursiveExpression` a string containing the evaluable expression expected by its parent function, as shown in Listing 31:

```
1 {  
2   funcType: 'recursiveF',  
3   encGen: [ 1, 0.033989, 1, 0.416408, 0.58, 0.552568, 0, 0 ],  
4   decGen: 'r(q(2))',  
5   phenLength: 1,  
6   phenVoices: 1,  
7   recursiveExpr: 'terms[terms.length - 2 % initTermsLength - 1]'  
8 }
```

Listing 31: Simplest subspecimen of a `recursiveF` function

Internally, the remaining recursiveF functions progressively edit this expression to construct a string according to the variables expected by `lRecursioOrder2`. Therefore, to generate eighty elements of a Fibonacci-like sequence, summing the two previous terms starting from two initial terms, it is required the subgenotype `rAdd(r(q(1)), r(q(2)))`, which returns a string: `'terms[terms.length - 1 % initTermsLength - 1] + terms[terms.length - 2 % initTermsLength - 1]'`. The only remaining task is to determine the initial terms and the number of iterations, forming the following genotype:

```

1 lRecursioOrder2(
2   rAdd(           // Fibonacci-like recursive equation
3     r(q(1)),
4     r(q(2))),
5   p(0.515915),   // sequence 1st term
6   p(0.515915),   // sequence 2nd term
7   q(80))         // number of iterations

```

Listing 32: Genotype with a Fibonacci-like recursion

Now, let's consider a randomly written genotype, depicted in Listing 33. There are two recursive equations, mapped to values `midipitch` and `intensity` using `lmWrap` and `liWrap` respectively. Each recursion is framed within the code.

```

1 s(
2   vPerpetuumMobileLoop(
3     n(0.05),
4     lmWrap(
5       lRecursioOrder4(
6         rTanh(
7           rCos(
8             rSubtract(
9               rSubtract(
10                r(q(9)),
11                r(q(2))),
12              rSubtract(
13                r(q(2)),
14                r(q(-4)))))),
15         p(0.495282),
16         p(0.605767),
17         p(0.450958),
18         pGaussianRnd(),
19         q(46))),

```

```
20  la(5),  
21  liWrap(  
22    lRecurasioOrder3(  
23      rHypot(  
24        rDivide(  
25          rCosh(  
26            rHypot(  
27              r(q(2)),  
28              r(q(1)))),  
29          rAdd(  
30            rSin(  
31              r(q(-12))),  
32              r(q(-7))),  
33          rTan(  
34            rTan(  
35              r(q(0))))),  
36      p(0.743),  
37      p(0.32),  
38      p(0.63),  
39      q(700))))))
```

Listing 33: Genotype with recursions

If we extract and evaluate separately the first recursion taken by **lRecurasioOrder4** as its first argument (lines 6-14), which is the subgenotype

```
rTanh(rCos(rSubtract(rSubtract(r(q(9)), r(q(2))), rSubtract(r(q(2)), r(q(-4)))))),
```

the returned Object, shown in Listing 34, contains the expression that will be evaluated within the recursiveExpr key.

```

1 {
2   funcType: 'recursiveF',
3   encGen: [ 1, 0.304499, 1, 0.450397, 1, 0.978261, 1, 0.978261, 1, 0.033989, 1,
4     0.416408, 0.58, 0.704833, 0, 0, 1, 0.033989, 1, 0.416408, 0.58, 0.552568, 0, 0,
5     0, 1, 0.978261, 1, 0.033989, 1, 0.416408, 0.58, 0.552568, 0, 0, 1, 0.033989, 1,
6     0.416408, 0.58, 0.397584, 0, 0, 0, 0, 0, 0 ],
7   decGen: 'rTanh(rCos(rSubtract(rSubtract(r(q(9)),r(q(2))),rSubtract(r(q(2)),r(q(-4)
8     )))))',
9   phenLength: 1,
10  phenVoices: 1,
11  recursiveExpr: 'Math.tanh(Math.cos(terms[terms.length - 9 % initTermsLength - 1] -
12    terms[terms.length - 2 % initTermsLength - 1] - terms[terms.length - 2 %
13    initTermsLength - 1] - terms[terms.length - 4 % initTermsLength - 1]))'
14 }

```

Listing 34: Subspecimen returned by a compound recursion

Translated into standard notation, this recursive process represents the Equation 27:

$$x_n = \tanh(\cos((x_{n-1} - x_{n-3}) - (x_{n-3} - x_{n-2}))) \quad (27)$$

From the console, the recursiveF key can be called to directly retrieve the resulting compound expression. For the second recursive equation enclosed in lines 23-35 of Listing 33, this property is obtained by evaluating

```
rHypot(rDivide(rCosh(rHypot(r(q(2)),r(q(1)))),rAdd(rSin(r(q(-12))),r(q(-7))),rTan(rTan(r(q(0)))))).recursiveExpr
```

which returns the string

```
'Math.hypot(Math.cosh(Math.hypot(terms[terms.length - 2 % initTermsLength - 1],
terms[terms.length - 1 % initTermsLength - 1])) / Math.sin(terms[terms.length -
12 % initTermsLength - 1] + terms[terms.length - 7 % initTermsLength - 1],
Math.tan(Math.tan(terms[terms.length - 0 % initTermsLength - 1])))'
```

that will be passed to **lRecursioOrder3** as its first argument. Equation 28 represents its translation into standard notation. The beginning of the resulting phenotype is found in Figure 26.

$$x_n = \sqrt{\frac{\cosh^2 \sqrt{x_{n-3}^2 + x_{n-2}^2}}{\sin^2(x_{n-1}) + x_{n-2}} + \tan^2(\tan(x_{n-1}))} \quad (28)$$



Figure 26: Score generated using recursiveF functions. To better visualize the fluctuations in the pattern generated by the recursive expression of Equation 27 applied to the pitches, a quarter-tone scale has been chosen by setting `stepsPerOctave = 24` as `playbackOption`.

## 4.9. Internal autoreferences

### 4.9.1. The importance of self-reference in music

In the construction of a musical composition, self-reference is an essential procedure. Fundamental mechanisms such as repetition, variation, and transposition, whether of minimal elements or large sections, can be considered as internal self-references within the structure. The sense of coherence in the discourse is partly achieved by playing with the listener's memory and their ability to recognize previous elements, whether identical or transformed. Musical memory and pattern recognition operate at all temporal levels, from milliseconds to hours.

Since the inception of the project, the ability to represent multiple self-references at all structural levels has been considered an essential feature. In designing the syntax of genotypes, various practical approaches have been tested, each with its pros and cons. While meta-programming of functional trees may lead to unforeseen outcomes, the incorporation of self-references that have an effect throughout the evaluation process poses several additional problems that must be well resolved. The system described below has proven to be the most reliable, especially due to one characteristic: naturally, as the nodes of the functional tree are evaluated, all available subgenotypes are stored. This ensures that only previous elements in the constructive process can be referenced. This sets a good analogy with what also happens in listening because memory can only establish links between patterns to the past, and to what has already been perceived, to construct the mental image of the musical discourse.

## 4.9.2. Subgenotype indexing

In the context of this model, *autoreference* denotes the reuse of parts of a functional tree through a reference from another part of it. As the decoded genotype is evaluated, an Object stored in the variable `subGenotypes` is written that stores the decoded subgenotypes it contains. Consider, for example, the definitions of the `lConcatL` genotype function in Listing 35, which takes two `listF` items and concatenates them:

```

1 indexGenotypeFunction("lConcatL", "listF", 41, ["listF", "listF"]);
2 lConcatL = (l1, l2) => indexDecGens({
3   funcType: "listF",
4   encGen: flattenDeep([1, g2p(41), l1.encGen, l2.encGen, 0]),
5   decGen: "lConcatL(" + l1.decGen + "," + l2.decGen + ")",
6   encPhen: l1.encPhen.concat(l2.encPhen)
7 });

```

Listing 35: Definition of `lConcatL` genotype function

In the architecture of a genotype function, before returning the constructed subspecimen, it is sent to the auxiliary function `indexDecGens`. Its implementation, in Listing 36, consists of checking if the decoded genotype contained in the property `decGen` already exists in the Object held in the global variable `subGenotypes`; if it is a new expression, it is included following those already indexed within the listing of the corresponding function type.

```

1 var indexDecGens = subspecimen => {
2   // if subgenotype is found, returns subspecimen
3   var subgenotypeRepeated;
4   for (var idx = 0; idx < subGenotypes[subspecimen.funcType].length; idx++) {
5     subgenotypeRepeated = subspecimen.decGen.localeCompare(
6       subGenotypes[subspecimen.funcType][idx]);
7     if (subgenotypeRepeated == 0) return subspecimen;
8   }
9   // if subgenotype is new, indexes it and returns subspecimen back
10  subGenotypes[subspecimen.funcType].push(subspecimen.decGen);
11  return subspecimen;
12 };

```

Listing 36: Indexation of subgenotypes with `indexDecGens`

### 4.9.3. Minimal examples of internal autoreference

Let's consider a minimal genotype in Listing 39, containing an autoreference to a subgenotype of type score. In line 9, the expression `sAutoref(1)` will be replaced at runtime by the first score subgenotype currently present in the `subGenotypes` library.

```

1 sConcatS(
2   s(
3     vPerpetuumMobileLoop(
4       n(0.2),
5       lm(60, 61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 71, 72),
6       la(50, 100),
7       li(90, 80, 50))),
8   sHarmonicGrid(
9     sAutoref(1),
10    hPentatonicScale(
11      m(65)))

```

Listing 37: Internal autoreference to a score

Listing 38 displays the Object contained in the variable `subGenotypes` at the precise moment when `sAutoref(1)` is evaluated. Note that there are subgenotypes, such as `m(65)`, that do not yet appear in the listing because they have not been evaluated.

```

1 {
2   scoreF: [
3     's(vPerpetuumMobileLoop(n(0.2),lm(60,61,62,63,64,65,66,67,68,69,70,71,72),la
4       (50,100),li(90,80,50)))',
5   ],
6   voiceF: [
7     'vPerpetuumMobileLoop(n(0.2),lm(60,61,62,63,64,65,66,67,68,69,70,71,72),la
8       (50,100),li(90,80,50))'
9   ],
10  eventF: [],
11  paramF: [],
12  listF: [],
13  notevalueF: [ 'n(0.2)' ],
14  lnotevalueF: [],
15  midipitchF: [],
16  lmidipitchF: [ 'lm(60,61,62,63,64,65,66,67,68,69,70,71,72)' ],
17  articulationF: [],

```



```

16  larticulationF: [ 'la(50,100)' ],
17  intensityF: [],
18  lintensityF: [ 'li(90,80,50)' ],
19  quantizedF: [],
20  lquantizedF: [],
21  goldenintegerF : [],
22  lgoldenintegerF : [],
23  harmonyF: [],
24  timegridF: [],
25  recursiveF: []
26  }

```

Listing 38: Example of `subGenotypes` Object

The output of this genotype is equivalent to the evaluation of what is shown in Listing 39. Within the code, the replication of the branch that occurs internally is framed.

```

1  sConcatS(
2    s(
3      vPerpetuumMobileLoop(
4        n(0.2),
5        lm(60, 61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 71, 72),
6        la(50, 100),
7        li(90, 80, 50))),
8    sHarmonicGrid(
9      s( // effective substitution caused by 'sAutoref(1)'
10       vPerpetuumMobileLoop(
11         n(0.2),
12         lm(60, 61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 71, 72),
13         la(50, 100),
14         li(90, 80, 50))),
15     hPentatonicScale(
16       m(65)))

```

Listing 39: Equivalent genotype after evaluating internal autoreference to a score

The musical output of this example, shown in Figure 27, involves the concatenation of two elements: a chromatic scale and the autoreference to that same scale, which serves as the argument of `sHarmonicGrid`. This function readjusts the pitches to fit the F pentatonic mode. Perhaps the most convenient aspect of iterative indexing of subgenotypes is the simple facilitation, and even encouragement, of reusing previous elements within a musical score. This is a ubiquitous circumstance in nearly any genre of music, hence this

architecture introduces a bias in the musical latent space toward results that encompass this internal cohesion and linkage of materials. In any case, if one explicitly wishes to avoid the use of self-references, it is merely a matter of deactivating their inclusion in the list of eligible functions within the general library via the user interface.



Figure 27: Minimal example of autoreference

Autoreferences also allow for more concise writing. For instance, the genotype in Listing 29 is equivalent to the following Listing 40:

```
1 vMotif(  
2   lNWrap(lLogisticMap(p(0.5),p(0.1),p(0.9),q(100),pRnd())),  
3   lMWrap(lAutoref(1)),lAWrap(lAutoref(1)),liWrap(lAutoref(1)))
```

Listing 40: Equivalent genotype with autoreferences

#### 4.9.4. Definition of autoref functions

For the implementation of autoreferences, the framework function deployed in Listing 41 is used, which operates as follows:

- At line 2, subGenLength stores the number of available subgenotypes for its funcType.
- If there are no subgenotypes to reference, the autoreference is replaced by the alternative genotype alternDecGen, which is a random function of the corresponding type. In this case, making a self-reference is impossible, but the automatic generation of the genotype continues its course smoothly.
- If creating the autoreference is possible, the argument subGenIndex is used to determine which subgenotype is chosen. If the given value exceeds the available expressions, a modulo operator is applied at line 8 to scale it to a valid range.
- Line 9 evaluates the chosen subgenotype, and the resulting subspecimen is stored in evaluatedSubGen.
- The index of the referenced subgenotype is converted in line 13 into a golden encoded integer, with **g2p**, which enables about half a million possible references, much more than ever possible needed.

- Finally, the own subgenotype of the autoreference function is returned, where the decGen key returns the function name in the form <funcID> + "autoref", provided by the funcName argument. All subsequent keys related to the phenotype are extracted from the subspecimen stored in evaluatedSubGen.
- Note that, unlike the rest of the genotype functions, the returned subspecimen is not passed to the function `indexDecGens`. The reason for this is to avoid autoreference from being indexed as a subgenotype, which leads to an excessive iteration effect due to the chain of autoreferences, often resulting in an overrepresentation of the initially referenced branches. However, it has been the practical experience that has led to the clear conclusion that the results are more balanced without such inclusion. Nevertheless, the effects of allowing chained autoreferences can be easily verified by introducing the Object from lines 12 to 20 into the function `indexDecGens`.

```
1 var autoref = (funcName, funcType, encodedFuncIndex, subGenIndex, alternDecGen) => {
2   var subGenLength = subGenotypes[funcType].length;
3   if (subGenLength == 0) {
4     var evaluatedSubGen = evalExpr(alternDecGen);
5     return evaluatedSubGen;
6   }
7   else {
8     subGenIndex = Math.abs(subGenIndex - 1) % subGenLength;
9     var evaluatedSubGen = evalExpr(subGenotypes[funcType][subGenIndex]);
10  }
11  return {
12    funcType: funcType,
13    encGen: flattenDeep([1, encodedFuncIndex, 0.57, g2p(subGenIndex + 1), 0]),
14    decGen: funcName + "(" + (subGenIndex + 1) + ")",
15    encPhen: evaluatedSubGen.encPhen,
16    phenLength: evaluatedSubGen.phenLength,
17    phenVoices: evaluatedSubGen.phenVoices,
18    harmony: evaluatedSubGen.harmony,
19    timegrid: evaluatedSubGen.timegrid,
20    analysis: evaluatedSubGen.analysis
21  };
22  };
```

Listing 41: Framework function `autoref` to create all autoreferences functions

The **autoref** function thus facilitates the creation of specific autoreference functions for each type. Listing 42 shows the definition of some of these specific functions. Some of them use the `defaultEvent` to build the alternative subgenotypes they will return in case an autoreference is not possible. This default event is constructed based on the number of extra parameters of the active species. If, for example, `eventExtraParameters = 2`, the resulting `defaultEvent` will be the string `"e(nRnd(),mRnd(),aRnd(),iRnd(),pRnd(),pRnd())"`.

```

1 // autoreferences functions for each output type
2 indexGenotypeFunction("pAutoref", "paramF", 25, ["goldenintegerLeaf"]);
3 pAutoref = subgenotypeIndex => autoref("pAutoref", "paramF", g2p(25),
    subgenotypeIndex, "pRnd()");
4 indexGenotypeFunction("lAutoref", "listF", 26, ["goldenintegerLeaf"]);
5 lAutoref = subgenotypeIndex => autoref("lAutoref", "listF", g2p(26),
    subgenotypeIndex, "lRnd(pRnd(),qRnd())");
6 indexGenotypeFunction("eAutoref", "eventF", 27, ["goldenintegerLeaf"]);
7 eAutoref = subgenotypeIndex => autoref("eAutoref", "eventF", g2p(27),
    subgenotypeIndex, defaultEvent);
8 indexGenotypeFunction("vAutoref", "voiceF", 28, ["goldenintegerLeaf"]);
9 vAutoref = subgenotypeIndex => autoref("vAutoref", "voiceF", g2p(28),
    subgenotypeIndex, ("v(" + defaultEvent + ")"));
10 indexGenotypeFunction("sAutoref", "scoreF", 29, ["goldenintegerLeaf"]);
11 sAutoref = subgenotypeIndex => autoref("sAutoref", "scoreF", g2p(29),
    subgenotypeIndex, "s(v(" + defaultEvent + ")"));

```

Listing 42: Definition of some autoreference functions

#### 4.9.5. Indexing tree and subgenotype calls

Let's analyze a more complex genotype, with multiple autoreferences. Listing 43 depicts multiple autoreferences using arrows. Additionally, within comments on each line, it is displayed the subgenotype number assigned to each branch in the global variable `subGenotypes`. The label displacement indicates the indexing order: the more to the left, the earlier it has been included in the array of subgenotypes of its type.

```

1  s2V( // s-1
2  vABCAB( // v-7
3  vRepeatV( // v-2
4  vConcatE( // v-1
5  e3Chord( // e-1
6  n(0.21853), // n-1
7  mRnd(), // m-1
8  m(49), // m-2
9  mAutoref(2),
10 aRnd(), // a-1
11 iRnd(), // i-1
12 e4Chord( // e-2
13 nRnd(), // n-2
14 m(69), // m-3
15 mAutoref(3),
16 m(90), // m-4
17 mAutoref(4),
18 aRnd(), // already indexed
19 iAutoref(1))),
20 q(12), // q-1
21 vConcatV( // v-6
22 vAutoref(2),
23 vRepeatV( // v-5
24 vRepeatV( // v-4
25 vPerpetuumMobile( // v-3
26 nAutoref(2),
27 lm(54,75,27,53,78,67), // lm-1
28 laWrap( // la-1
29 lRemap( // l-2
30 l5P( // l-1
31 pRnd(), // p-1
32 pRnd(), // already indexed
33 pAutoref(1),
34 p(0.238), // p-2
35 pAutoref(2),
36 pRnd(), // already indexed
37 pAutoref(1))),
38 liWrap( // li-1
39 lRnd( // l-3
40 pAutoref(2),
41 qAutoref(1))),
42 q(6), // q-2
43 qAutoref(2)),
44 vAutoref(3),
45 vAutoref(5))

```

Listing 43: Visualization of autoreferences and indexing order

Listing 44 provides an abbreviated display of the `subGenotypes` content generated at the end of the complete genotype evaluation, where you can verify the entry order of the previous functional tree. Several observations are relevant to this process:

- The indexing order goes from inside out and top to bottom, directly caused by the sequential evaluation of nodes in the functional tree. This is particularly evident when comparing the label numbering of the indexing of voice nodes. This order explains why the arrows revealing autoreferences always point upwards and, consequently, always reuse previous musical material.
- Multiple autoreferences may point to the same branch. Both lines 33 and 37 contain the same self-reference as line 31.
- The expressions in lines 18, 32, and 36 do not generate a new entry in the library since they already exist. This filtering is also aimed at avoiding the overrepresentation of subgenotypes.

```

1 {
2   scoreF : [ 's2V(vABCAB(vRepeatV(...' ], // s-1
3   voiceF : [ 'vConcatE(e3Chord(n(...' , // v-1
4             'vRepeatV(vConcatE(e3Chord(...' , // v-2
5             'vPerpetuumMobile(nAutoref(2)...' , // v-3
6             'vRepeatV(vPerpetuumMobile(...' , // v-4
7             'vRepeatV(vRepeatV(vPerpetuumMobile(...' , // v-5
8             'vConcatV(vAutoref(2),vRepeatV(...' , // v-6
9             'vABCAB(vRepeatV(vConcatE(e3Chord(...' ], // v-7
10  eventF : [ 'e3Chord(n(0.21853),mRnd(),m(49),mAutoref(2)...' , // e-1
11            'e4Chord(nRnd(),m(69),mAutoref(3)...' ], // e-2
12  paramF : [ 'pRnd()' , // p-1
13            'p(0.238)' ], // p-2
14  listF : [ 'l5P(pRnd(),pRnd(),pAutoref(1),p(0.238),...' , // l-1
15           'lRemap(l5P(pRnd(),pRnd(),pAutoref(1)...' , // l-2
16           'lRnd(pAutoref(2),qAutoref(1))' ], // l-3
17  notevalueF : [ 'n(0.21853)' , // n-1
18               'nRnd()' ], // n-2
19  lnotevalueF : [],
20  midipitchF : [ 'mRnd()' , // m-1
21               'm(49)' , // m-2
22               'm(69)' , // m-3
23               'm(90)' ], // m-4
24  lmidipitchF : [ 'lm(54,75,27,53,78,67)' ], // lm-1
25  articulationF : [ 'aRnd()' ], // a-1
26  larticulationF : [ 'laWrap(lRemap(l5P(...' ], // la-1
27  intensityF : [ 'iRnd()' ], // i-1
28  lintensityF : [ 'liWrap(lRnd(pAutoref(2),...' ], // li-1

```

```
29   quantizedF : [ 'q(12)',           // q-1
30                 'q(6)' ],         // q-2
31   lquantizedF : [],
32   goldenintegerF : [],
33   lgoldenintegerF : [],
34   harmonyF : [],
35   timegridF : [],
36   recursiveF : []
37 }
```

Listing 44: Object `subGenotypes` created after evaluating a decoded genotype

## 4.10. Genotype functions libraries

### 4.10.1. Creation and updating of libraries

The genotype functions library is another main component of this model. It contains a catalog with the characteristics of all available functions that can come into play for the creation of specimens. The procedures for automatic genotype writing will access the necessary data through this library.

During the initialization time of the core code, when each function is loaded, a call to the function `indexGenotypeFunction` is made. This function, shown in Listing 45, progressively includes in the Object `allGenotypeFunctionsData` the identifiers and the list of arguments required by each function.

```
1  var indexGenotypeFunction = (funcName, funcType, funcIndex, parameters,
2    extraParametersType) => {
3    if (allGenotypeFunctionsData[funcType] == undefined)
4      allGenotypeFunctionsData[funcType] = {};
5    allGenotypeFunctionsData[funcType][funcName] = {
6      "functionIndex": funcIndex,
7      "functionType": funcType,
8      "arguments": parameters,
9      "extraArguments": extraParametersType
10   };
11 }
```

Listing 45: Function `indexGenotypeFunction` to create an all functions library

Once all genotype functions data has been gathered, the `genotypeFunctionsLibrary` is constructed as another Object within a global variable. Listing 46 shows how this data structure is organized. All items are listed several times according to different sorting criteria. Each classification will be necessary at some point in the generative processes of functional tree composition.

From this complete library, the `eligibleFunctionsLibrary` is extracted. It is another Object with a structure identical to `genotypeFunctionsLibrary`, but gathering only the functions that can be chosen in the automatic genotype writing process. Additionally, it adds the key `eligibleFunctions` with the list of integer indices of the current eligible functions.

The eligible functions library is regenerated by `createEligibleFunctionLibrary` each time the list of eligible functions changes from the user interface, or when rendering a specimen containing this information as one of its `initialConditions`. If there is a change in the species, altering the total of event extra parameters, both libraries must be regenerated since the parameter lists of all those genotype functions that need to incorporate additional arguments are also updated.

```
1 {
2   decodedIndices: {           // functions sorted by integer decoded index
3     '0': 'p',
4     '1': 'l',
5     '2': 'e',
6     '3': 'v',
7     ...
8   },
9   encodedIndices: {         // functions sorted by encoded index
10    '0': 'p',
11    '0.01005': 'hHexatonicScale',
12    '0.012703': 'rConstant',
13    '0.026311': 'l5P',
14    ...
15  },
16  'functionNames': {       // functions sorted by name, with main data
17    a: {
18      encIndex: 0.562306,
19      intIndex: 9,
20      functionType: 'articulationF',
21      arguments: [ 'articulationLeaf' ]
22    },
```



```

23   aAutoref: {
24     encIndex: 0.667551,
25     intIndex: 281,
26     functionType: 'articulationF',
27     arguments: [ 'goldenintegerLeaf' ]
28   },
29   ...
30 },
31 'functionLibrary': {           // functions grouped by output types, with main data
32   paramF: {
33     p: { functionIndex: 6, functionType: 'paramF', arguments: [Array] },
34     pRnd: { functionIndex: 131, functionType: 'paramF', arguments: [Array] },
35     pGaussianRnd: { functionIndex: 132, functionType: 'paramF', arguments: [Array] },
36     pAutoref: { functionIndex: 25, functionType: 'paramF', arguments: [Array] },
37     ...
38   },
39   notevalueF: {
40     n: { functionIndex: 5, functionType: 'notevalueF', arguments: [Array] },
41     nRnd: { functionIndex: 310, functionType: 'notevalueF', arguments: [Array] },
42     nAutoref: { functionIndex: 277, functionType: 'notevalueF', arguments: [Array] },
43     ...
44   }
45   ...
46 }
47 }

```

Listing 46: Data structure of `genotypeFunctionsLibrary` data structure

The Table 7 provides further details about the constituents of the library of eligible genotype functions, which is the data structure that will be extensively manipulated during the rendering of specimens.

| Key                            | Description  |
|--------------------------------|--|
| <code>eligibleFunctions</code> | Ordered list of function indices, assigned as integers at the moment of their inclusion in the core code as unique identifiers for each function. It's an array of integers that exactly matches the <code>eligibleFunctions</code> element, which is part of the initial conditions that characterize a specimen. |
| <code>decodedIndices</code>    | Expands the previous numeric array by pairing each integer with the name of the genotype function it points to.  |

|                 |  |
|-----------------|--|
| encodedIndices  | Ordered list of function indices converted to golden encoded integers. This reordering is used when developing the decision tree determined by a germinal vector. The use of golden encoded integers is essential to achieve a balanced distribution of these pointers in the range [0, 1], avoiding overrepresentation of certain procedures over others. |
| functionNames   | Alphabetically ordered list of function names paired with their encoded and decoded identifiers, along with the output function type, and an array containing the function types for each required argument.   |
| functionLibrary | List of functions similar to the previous one, but grouped by output function type.  |

---

Table 7: Data structure of the `eligibleFunctionsLibrary` Object

#### 4.10.2. Indexing functions with golden encoded integers

The two libraries must function coordinately. The handling of these structures has been designed to meet these requirements:

- The structure of the libraries should be consistent over time, while also being configurable by the user.
- The addition of new functions should not compromise repeatability in specimen rendering. In other words, a genotype should always produce the same phenotype upon rendering, regardless of whether the library has been expanded. This motivates certain decisions in the design of the genotype indexing system, explained below.
- To ensure that the process of automatic genotype writing is optimal in terms of the variety of its outcomes, the encoded indices identifying each genotype function should have a balanced distribution that avoids biases and underrepresentations of certain functions in the latent musical space. This should be the case whether there are just a few dozen functions or thousands of them, all within the range of [0,1].

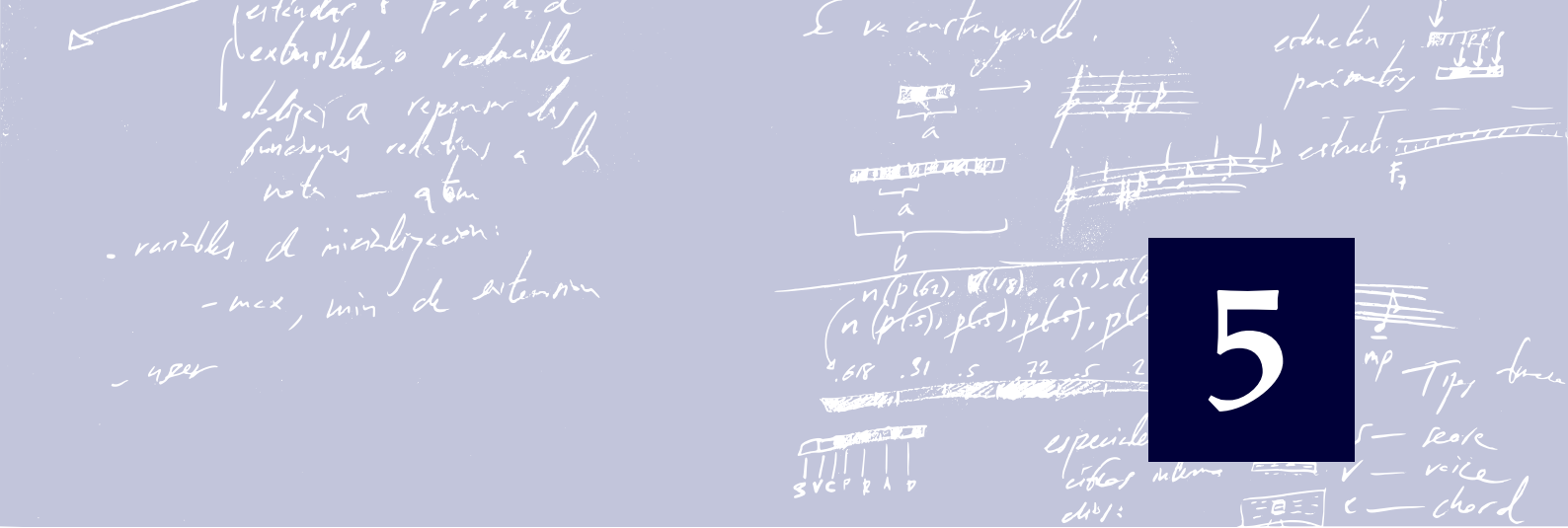
To reconcile these three elements, golden encoded integers are employed. The details of this conversion are left for the following Section 5.1.4, where the encoding and decoding of specimens are globally addressed.

### 4.10.3. Influence of the palette of eligible functions on style

The set of procedures, beyond being a mere requirement for the construction of genotypes, strongly conditions the latent musical space and can therefore be considered a primary identifying characteristic of a particular style. Especially from the 20th century onwards, certain aesthetic movements, composers, or even specific works can be linked to one or several characteristic procedures. So much so that certain pieces in the repertoire are primarily recognized for their introduction of specific techniques.

GenoMus allows for the precise configuration of which functions to activate from the user interface. It is possible, to drive experimentation, and to narrow down this variety, which can be useful in obtaining stylistically consistent results. In his well-known text *Poetics in Music*, Stravinsky [147] considered self-limitation to certain restraints as an essential element to stimulate creativity, akin to a game that requires clear rules to create the necessary tension among its elements:

My freedom will be so much the greater and more meaningful the more narrowly I limit my field of action and the more I surround myself with obstacles. Whatever diminishes constraint diminishes strength. The more constraints one imposes, the more one frees one's self of the chains that shackle the spirit.



# 5

## Encoding and decoding

“

I believe that music today could surpass itself by research into the outside-time category, which has been atrophied and dominated by the temporal category. Moreover, this method can unify the expression of fundamental structures of all Asian, African, and European music. It has a considerable advantage: its mechanization—hence tests and models of all sorts can be fed into computers, which will effect great progress in the musical sciences.

Iannis Xenakis [158, p. 200]

We have reached the point where the core of the conceptual framework discussed in Chapter 2, the encoding of these processes as one-dimensional vectors, will be put into practice. The design of the grammar outlined up to this point aims to create an extremely simple syntax in which all processes are expressed in a formally identical manner. The only concession made has been the inclusion of certain specific types of parameters to make them readable and manageable for humans.

This chapter delves into the details of encoding the elements that constitute a specimen. Let’s remember that the fundamental data contained in a rendered specimen are the initial conditions for its creation, the genotype (which is the program resulting from metaprogramming from simple numeric sequences), and the phenotype (the musical structure generated by evaluating that program). The detailed structure of a specimen was previously presented in Table 2.

## 5.1. Genotype encoding

The encryption of the genotypes decoded as a sequence of numbers is achieved by decomposing the functional expressions into combinations of these three components, which in total involve five different kinds of tokens:

**Function opening** — (2 tokens) — It consists of the function name and the opening parenthesis. A *flag* value 1, marking the beginning of this opening, and the *function index*, identifying which function is being used.

**Leaf** — (2 tokens) — Numerical values at the end of each branch. Before the *leaf value*, a *leaf type identifier* is required to employ the correct conversions in the decoding process.

**Function closing** — (1 token) — The closing parenthesis. This element is necessary because the leaves can be a series of values (or no value at all), hence a *flag* 0 indicates the closure of the function.

To represent different components of the functional tree as normalized values in the range  $[0, 1] \in V$ , several strategies and conversions are employed, depending on the token type and the handled numeric ranges. All these mappings have been designed to cover a very wide spectrum of possible results, while also creating encodings with clear numerical contrasts that allow any analysis algorithm to easily capture the differences between vectors.

For this encoding to also work in reverse, from the numerical vector to the functional expression, it is necessary to consider in detail how this translation works for each of these tokens. Let's remember that the encoded genotype has a dual function: it serves as the abstract representation created from an executable alphanumeric expression, but simultaneously, it must act as an initial element capable of regenerating the exact expression from which it has been derived. In other words, every encoded genotype must be a valid germinal vector that produces the same decoded genotype as the original. This fact enables the retrotranscription discussed in Section 5.6.

### 5.1.1. Leaf type identifiers

The identification of the next leaf type is needed to handle the right conversions since a function is often fed with arguments of a different type. All leaf types are tagged with numbers  $\geq 0.5$  for a reason: this number, besides being an identifier, is also used after the

retrotranscription as a threshold value to decide whether a new value should be added to the list of parameters. Since the threshold imposed by germinal condition *maxl* is always  $\leq 0.5$ , this ensures that all values in the list are correctly encoded until a flag value  $\emptyset$  closes the list and ends the function.

### 5.1.2. Leaf values

In the encoding of the leaf values of a functional tree, all the conversions explained in Section 3.5 come into play. In the case of generic parameters, no conversion is necessary (beyond correcting values outside the range or rounding floats with more than six digits of mantissa).

### 5.1.3. Function opening and closing flags

The encoded genotypes simply use 1 and  $\emptyset$  to indicate the opening and closing of a subgenotype. When writing an expression from any unidimensional vector, the original values from the germinal vector are simply ignored and overwritten. The choice of extreme values within the normalized range is aimed at creating well-differentiated numerical structures that can be recognized by any statistical machine-learning process. In the case of closing a substructure (which represents a closing parenthesis), it is required that this number be  $< 0.5$ , as we have just seen above, to indicate that no further argument is added to the open subexpression at that moment.

### 5.1.4. Genotype function indices

Encoded genotypes must refer unequivocally to the available functions  $\in E_{func}$  contained in any functional tree. So, as a first requirement, any genotype function must be pointed by a unique index  $\in V$ . Once indexed, a function keeps its index unchanged, to ensure that an encoded genotype will always be decoded as the same functional expression, regardless of changes in set *Funcs* after adding new functions to the library. At the same time, these indices must be as separated and uniformly distributed as possible in the interval  $[0,1]$  to obtain distinct vectors that are easily distinguishable for machine learning algorithms. Finally, integer indices must be assigned to each new genotype function without knowing how many new ones will be added in the future.

The functional structure of a genotype is primarily defined by the processes it encompasses, hence the numerical identification plays an extremely important role in this

abstract representation. The statistical measurability of the diversity among these structures significantly determines the expressive power of the entire encoding system.

To meet these conditions, these identifying indices are golden encoded integers, converted from their integer identifier using the special type of conversion explained in Section 3.5.9.

Once this bijection is defined, each new genotype function is assigned an integer index converted to its reciprocal encoded golden value. The transformation *tran* then takes each number in a germinal vector that represents a call for a genotype function and substitutes it with the closest available index of the output type required by its parent function, as illustrated in Figure 28.

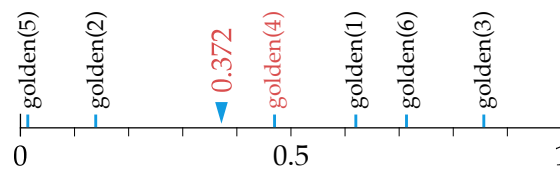


Figure 28: Selection of genotype functions with golden encoded integers. For this example, let's assume that only genotype functions with the decoded indices  $n = 1, 2, 3, \dots, 6$  are eligible for a given output type. Consequently, the value provided by the germinal vector is substituted by its closest number among the available golden values  $\text{golden}(n)$  and 0.372 is replaced by  $\text{golden}(4) = 0.472136$ , a permanent pointer to a function. This transformation ensures the rendering of the very same specimen, regardless of working with an extended function library in the future, since this identifier will remain always the same. To check these values from the JavaScript console, it can be employed the converters **g2p** and its inverse **p2g**. Thus, the expression **g2p(4)** will return the corresponding encoded index.

An important detail is that integer index assignments to functions start from 1. Function index zero is avoided because  $\text{golden}(0) = 0$ , which would make its encoded value coincide with the function closing flag value. Although this situation is distinguishable within the context, I have preferred to avoid it to facilitate statistical analysis of the encoded patterns.

## 5.2. Minimal examples

The Figure 29 displays several examples of deconstruction and encoding of minimal genotypes, given a valid functional expression as input. In turn, Table 8 compiles the conversions of different tokens to normalized values.

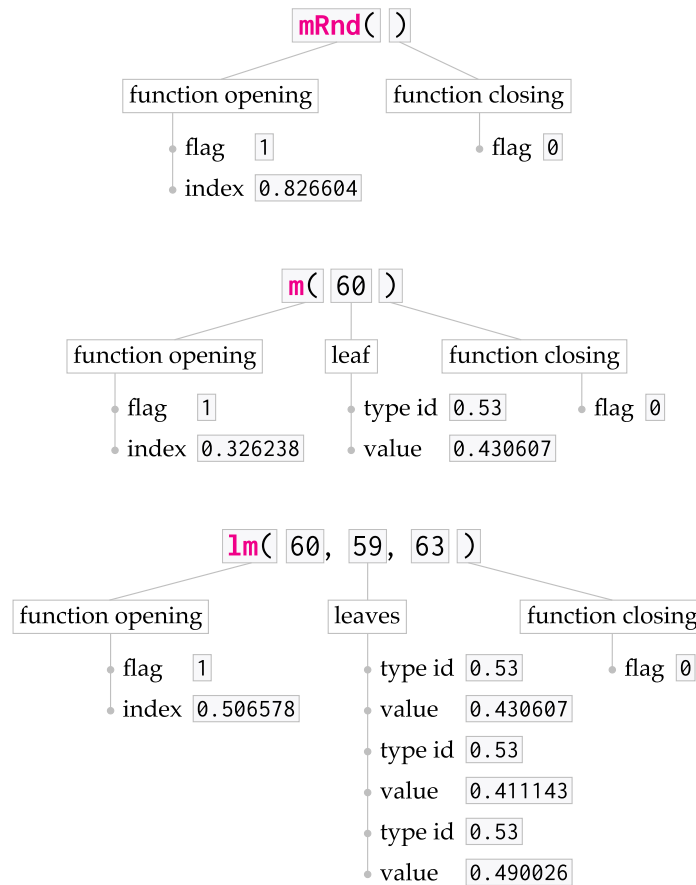


Figure 29: Decomposition and encoding of minimal genotypes. These three expressions demonstrate how different cases of leaves are handled: functions without arguments (voidLeaf), only one value, or a list of values.

### 5.3. Visualization of unidimensional vectors

To work with the lengthy numerical sequences that can be generated, especially during the addition of new functions and debugging, we can visualize these vectors in a sort of barcode. The table also shows the colors used for visualization. Table 8 also demonstrates how different color tones are assigned to various values to ease the readability of these numerical structures.<sup>56</sup> Figure 33 shows an example of an encoded and visualized genotype.

<sup>56</sup>Visualizing genetic code sequences is a discipline in itself with multiple approaches [112]. I draw inspiration, in a very loose manner, from some of these ways of visualizing intricate structures to gain insight into how functional trees are being translated and flattened into one-dimensional sequences.








| Genotype element        | Token                | Encoding        | Color code  |
|-------------------------|----------------------|-----------------|---|
| <b>function opening</b> | new function flag    | 1               |  |
|                         | function index       | golden integer  |  |
| <b>leaf</b>             | leaf type identifier | values > 0.5    |  |
|                         | leaf value           | converted value |  |
| <b>function closing</b> | closing parenthesis  | 0               |  |

Table 8: Numerical encoding of functional expression tokens. For visualizing the resulting numerical vectors, each type of token is represented by a color scheme: black to denote the beginning and end of subgenotypes, warm colors for references to function indices, cool colors for leaf values, and gray for leaf type identifiers.

The conventions defined for this visualization are as follows:

- The length of each bar indicates the value within the range  $[0, 1]$ .
- Black indicates the opening or closing of a subgenotype.
- Reddish tones are reserved for values corresponding to a golden encoded integer used in the function library.
- Various shades of gray indicate a leaf type identifier, aiding in identifying how many leaves there are and where they are located.
- The range of cool colors from green to violet is used for the rest of the values (always leaf values).
- A leaf value may be one of the key values mapped to black, reddish, or gray tones. For simplicity, no distinction is made in this case. These cases are easily detectable from context.
- To visualize very similar values, the colors in each range vary widely even with small increments. Figure 31 shows the treatment for the closest values within a range.

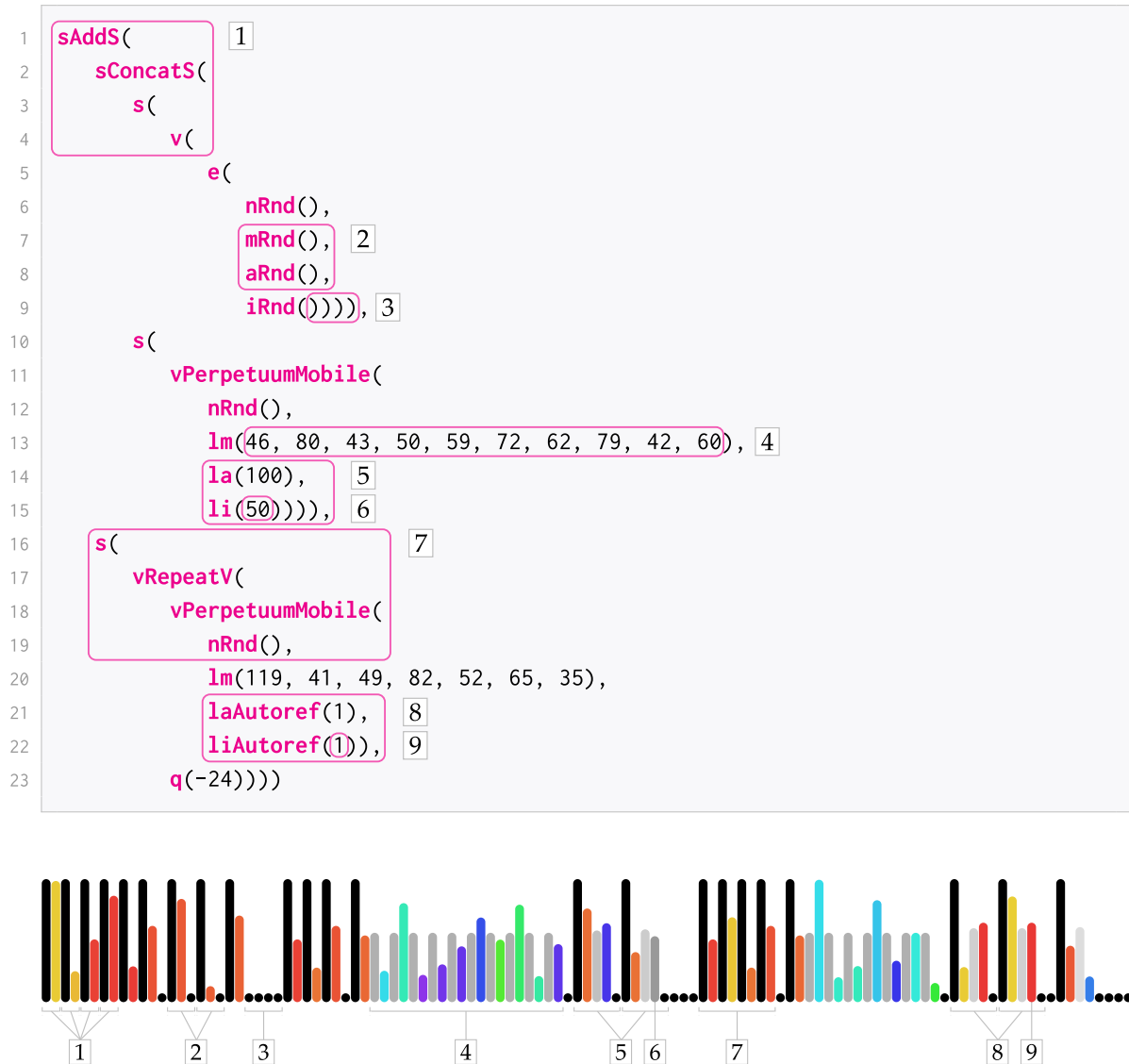


Figure 30: Genotype and its annotated visualization. Recognizable patterns can be observed in the graphical representations of the encoded genotypes, also marked on the decoded genotype above:

1. Function openings.
2. Subgenotypes of functions without leaf.
3. Several consecutive function closings.
4. List leaf type.
5. Subgenotypes with a single leaf.
6. Leaf value appearing in gray due to having an encoded value matching the leaf type identifier.
7. Similar to the beginning, a sequence of functions indicates a nested structure.
8. Self-references, identifiable by the red-colored leaf (such as the one marked with 9), indicating the use of a golden encoded integer to reference another subgenotype.

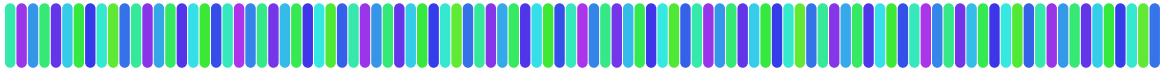


Figure 31: Color assignment for leaf values between 0.4 and 0.4001 are visualized in minimal increments of 0.000001. This aims to ensure that even the smallest possible difference between values receives highly contrasting colors. Although it may seem that there is a repeated pattern, the hue values of the color change slightly in each cycle to cover the entire possible color palette.

To better apprehend the numerical structures resulting from the encoding, Figure 30 displays a decoded genotype alongside its encoding visualization. Various recurring features are highlighted in the functional expression and the barcode representation.

The colorization aids in recognizing these patterns for the human eye, but it's important not to forget that for statistical analysis in any machine learning system, an appropriate graphical representation would eliminate color and would resemble that of Figure 32.

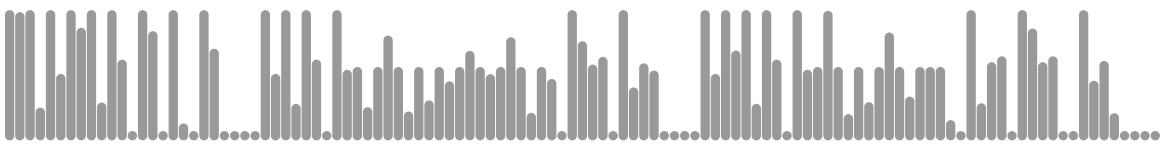
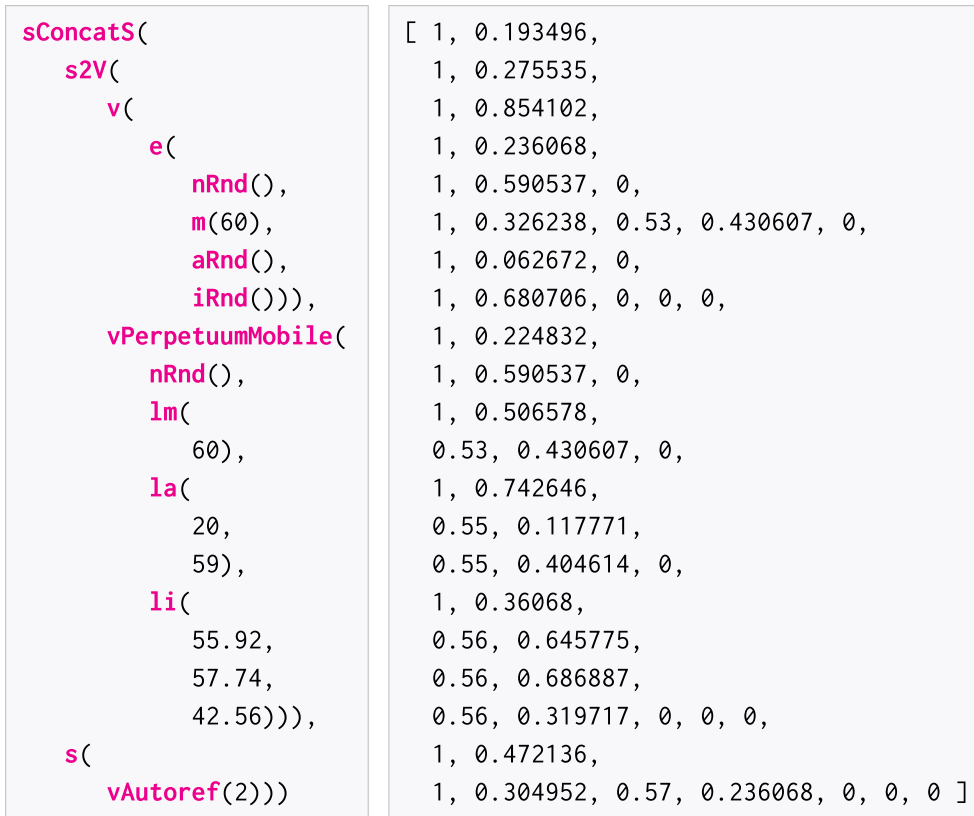


Figure 32: Monochromatic visualization of an encoded genotype

The Figure 33 is an additional example of encoding and visualizing a genotype. In this case, each line of code is matched with its numerical representation to provide more clarity in the translation of these substructures.

decoded genotype  
decGen:

encoded genotype  
encGen:



visualization

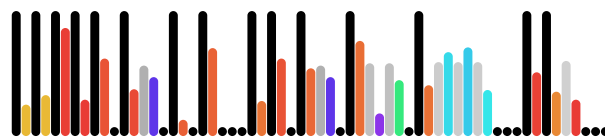


Figure 33: Compared listings of encoded and decoded versions of the same genotype. Paired by rows, the elements composing a functional tree and their purely numerical encoding. To enhance readability, the maximal decomposition into individual elements has been avoided, instead, certain blocks have been kept grouped with their parentheses, resembling the presentation in the user interface.

## 5.4. Germinal vector and genotype decoding

Now let's consider the reverse process: the conversion of a numerical sequence into a valid functional expression. It is important to recall that I refer to any sequence of numbers  $\in [0, 1]$ , of any length, as a germinal vector<sup>57</sup>. This sequence serves as the main initial condition and is used as a decision tree for generating a decoded genotype. This transformation must fulfill several requirements:

- Initially, a germinal vector is *universally computable*: any germinal vector of any length is valid as input and will ultimately result in a valid expression. However, a conversion process may be interrupted if the output exceeds certain length constraints imposed.
- To prevent infinite loops, additional constraints are imposed as complementary initial conditions: the maximum depth level and the maximum length of lists. A detailed analysis of the initial conditions set is provided in Section 5.6.
- The conversion process is deterministic: the same germinal vector, *accompanied by the same complementary initial conditions*, always produces the same specimen as a result. All random subprocesses, therefore, have a seed, either global or local.

Given a germinal vector, during the specimen rendering process, this sequence will result in an encoded genotype and its decoded counterpart. Figure 34 illustrates the translation from an encoded genotype to the functional expression and demonstrates the step-by-step operation on the numerical sequence of the germinal vector to transform it into a decoded genotype following the conventions specified in Table 8. This is the key process that enables the postulated retrotranscription discussed in Section 2.5 of the conceptual framework.

## 5.5. Initial conditions for specimen rendering

To reach the functional expression from the germinal vector, it's essential to determine certain complementary conditions, which I previously discussed theoretically in Section 2.4 and are now translating into practical implementation. These conditions are summarized in Table 9, where it can be verified that the first six keys determine the handling of the germinal vector according to three purposes:

---

<sup>57</sup>I adopt this term as yet another nod to the biological metaphor. In genetics, germinal line, most often *germline*, refers to the population of organisms that produces a new generation, such as the ovum or the sperm.

- `eventExtraParameters` and `specimenType` respectively define the species and the output type, and are the most defining conditioners since, in general, specimens of different species and output types cannot interact in evolutionary processes.
- `depthThreshold` and `maxListCardinality` limit excessive growth and the appearance of infinite loops.
- `localEligibleFunctions` and `seed` ensure the repeatability of the generative process, regardless of whether more functions are added to the available procedures library.

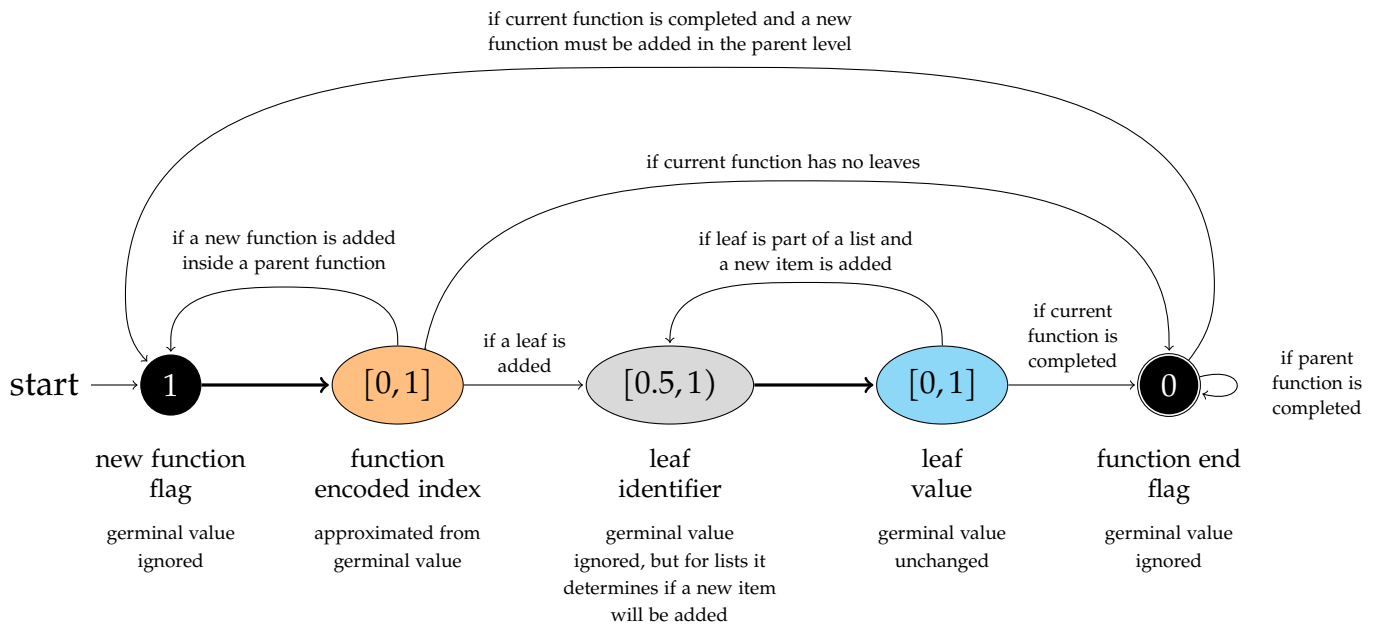


Figure 34: Numerical transformations applied by the `map trans` to create encoded genotypes from germinal vectors. Black and gray values work as flags and identifiers that replace the original numbers of a germinal vector. Orange indicates that a golden value conversion is needed to create an exact reference to a genotype function. Values in cyan nodes are not changed, since they correspond to leaf parameters fed as numeric arguments to terminal functions.

| Key                    | Description  | Example   |
|------------------------|--|---|
| eventExtraParameters   | Number of additional generic parameters, besides the mandatory ones (notevalue, midipitch, articulation and intensity). This feature defines the genotype species. | 3   |
| specimenType           | Output type of the core genotype function at the base of the functional tree.  | "scoreF"  |
| localEligibleFunctions | Set of indices of genotype functions eligible for building the functional expression.  | [ 0, 1, 2, 3, 4, 5, 7, 9, 10, 11, 12, 14, 15, 17, 19, 20, 21, 22, 25, 26, 27, 28 ]  |
| depthThreshold         | The depth limit from which, in constructing the functional tree, only identity functions will be used, thereby limiting potential excessive genotype growth.       | 9   |
| maxListCardinality     | Maximum number of items that a list-type leaf can contain.   | 15  |
| seed                   | Global seed value that will make repeatable the outcome of genotype processes involving pseudorandom numbers.  | 712114190846465   |
| germinalVector         | The sequence that will be used as a decision tree for the construction of a genotype, based on all previous initial conditions.                                    | [ 0.473092, 0.82544, 0.00228, 0.146925, 0.39915, 0.426203, 0.028851, 0.596555, 0.905445, 0.690755, 0.267705, 0.752557, 0.209328, 0.9131, 0.660553 ] |

*Table 9: Description and examples of initialConditions for specimen generation*

Encoding and decoding  
5.5. Initial conditions for specimen rendering

```

[ 0.647584, 0.367261,
0.228059, 0.766723,
0.992706, 0.661887,
0.290716, 0.841774,
0.738358,
0.024492, 0.37101 ]
[ 0.647584, 0.367261,
0.228059, 0.766723,
0.992706, 0.661887,
0.290716
0.841774, 0.738358,
0.024492, 0.371538 ]
[ 0.647584,
0.367261, 0.228059,
0.766723, 0.992706,
0.661887,
0.290716, 0.841774,
0.738358, 0.024492,
0.371538 ]
[ 0.647584, 0.367261,
0.228059, 0.766723,
0.992706,
0.661887,
0.290716,
0.841774, 0.738358,
0.024492, 0.371538 ]
[ 0.647584, 0.367261,
0.228059
0.766723, 0.992706,
0.661887, 0.290716,
0.841774, 0.738358,
0.024492,
0.371538 ] [ 0.647584,
0.367261, 0.228059,
0.766723, 0.992706,
0.661887, 0.290716,
0.841774,
0.738358, 0.024492,
0.371538 ] [ 0.647584,
0.367261, 0.228059,
0.992706, 0.661887,
0.290716, 0.841774,
0.738358
0.024492, 0.371538 ]
[ 0.647584, 0.367261,
0.228059
0.766723, 0.992706,
0.661887, 0.290716,
0.841774,
0.738358, 0.024492,
0.371538 ] [ 0.647584,
0.367261
0.228059
0.766723
0.992706
0.647584, 0.367261,
0.228059,
0.766723
0.992706

```

```

[ 1, 0.472136,
1, 0.842866,
1, 0.09017,
0.51, 0.841778,
0,
1, 0.38891,
1, 0.341313,
1, 0.708204,
0.5, 0.661887,
0,
1, 0.708204,
0.5, 0.371538,
0,
1, 0.708204,
0.5, 0.992706,
0,
1, 0.416408,
0.58, 0.024594,
0,
1, 0.708204,
0.5, 0.766723,
0,
0,
1, 0.742646,
0.55, 0.37101,
0.55, 0.364086,
0,
1, 0.36068,
0.56, 0.290792,
0.56, 0.738358,
0,
1, 0.618034,
0.5, 0.228059,
0.5, 0.992706,
0.5, 0.290716,
0,
1, 0.105245,
1, 0.708204,
0.5, 0.228059,
0,
1, 0.708204,
0.5, 0.841774,
0,
1, 0.708204,
0.5, 0.367261,
0,
1, 0.708204,
0.5, 0.290716,
0,
1, 0.416408,
0.58, 0.647584,
0,
0,
0,
0,
0,

```

```

s(
  vPerpetuumMobileLoop(
    n(
      0.9851
    ),
    lmWrap(
      lLogisticMap( 1
        p(
          0.661887
        ),
        p(
          0.371538
        ),
        p(
          0.992706
        ),
        q(
          -155
        ),
        p(
          0.766723
        )
      )
    ),
    la(
      54,
      53
    ),
    li(
      41.22,
      60.21
    ),
    2
  ),
  1(
    3
    0.228059,
    0.992706,
    0.290716
  ),
  lFibonacci(
    p(
      0.228059
    ),
    p(
      0.841774
    ),
    p(
      0.367261
    ),
    p(
      0.290716
    ),
    q(
      6
    )
  )
)
)

```

Figure 35: Germinal vector and conversion to encoded and decoded genotype in parallel. The influence of initialConditions has been highlighted in the code: 1. The function **lLogisticMap** has reached the predefined depthThreshold = 4, hence in the subsequent level, there are only identity functions. 2. Under the initial conditions eventExtraParameters = 2, the functions **l** and **lFibonacci** cover that requirement. 3. Another constraint is the maximum length of the lists. For this example, maxListCardinality = 3.



In the preceding Figure 35, the values of the germinal vector and their transformation into code following the prior automaton are depicted line by line. Additionally, the implications of the initial conditions on the metaprogramming of the functional expression are highlighted. The germinal vector in this example is extremely short, consisting of merely eleven values. During the construction of the encoded genotype, the values of the germinal vector are looped repetitively until the expression successfully closes in accordance with the constraints imposed by the initial conditions.

The process of numerical transformation becomes significantly clearer through the graphical representation of both vectors. Figure 36 illustrates the same process as the previous figure but also demonstrates the correspondences between values, elucidating which values are overwritten, which remain unchanged, and which undergo some adjustment.

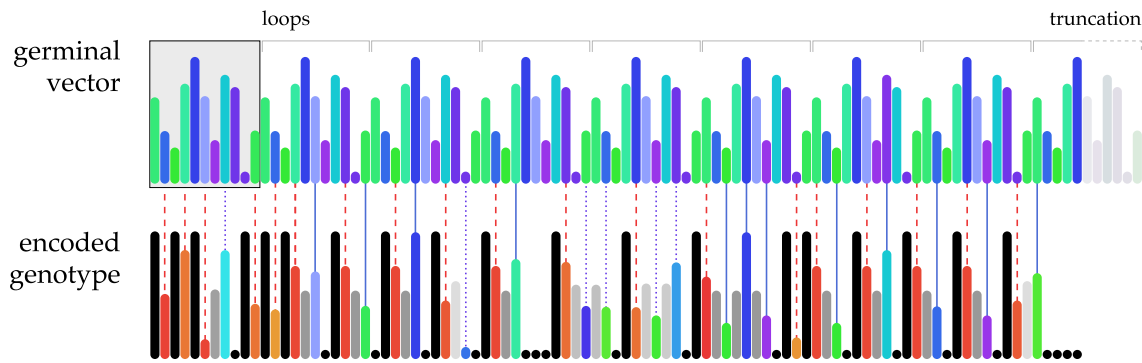


Figure 36: Comparative visualization of a germinal vector and its corresponding encoded genotype. The germinal vector (framed in black) contains only eleven values. Until the genotype is completed, this sequence repeats itself. Once the function tree is completed, no more values from the germinal vector are taken, and it is truncated at that point. The lines connecting both vectors illustrate these situations:

1. Solid blue lines mark leaf values that are copied without changes, displaying identical values (maintaining their color).
2. Dashed blue lines represent leaf values with adjustments (due to the quantization of certain specific parameters), hence the lengths of the lines remain the same, but there is a color change indicating slight modifications in the genotype values.
3. Red dashed lines indicate values used for the selection of a genotype function (hence their reddish tone) and have been adjusted following the process described in Section 5.1.4. It can be observed that although the tendency is for the final value to be similar to that of the germinal vector (similar length), in this case, the eligible functions in each class have been greatly reduced. Consequently, sometimes the length of the bar is quite different, even though it represents the best possible approximation to the available values.
4. Values not connected indicate their lack of relevance, as they are replaced by flags or corresponding identifiers.

It is easy to distinguish a germinal vector from its transformation into an encoded genotype due to the notable absence of colors associated with key values such as identifiers or golden encoded indices. Considering that roughly half of the values in the germinal vector are overwritten, the search space when applying machine learning algorithms becomes much smaller than it initially appears.

As we will observe through several musical examples in Section 7.1.2, experimenting with very short germinal vectors and mutations upon them enables the attainment of unexpected results.

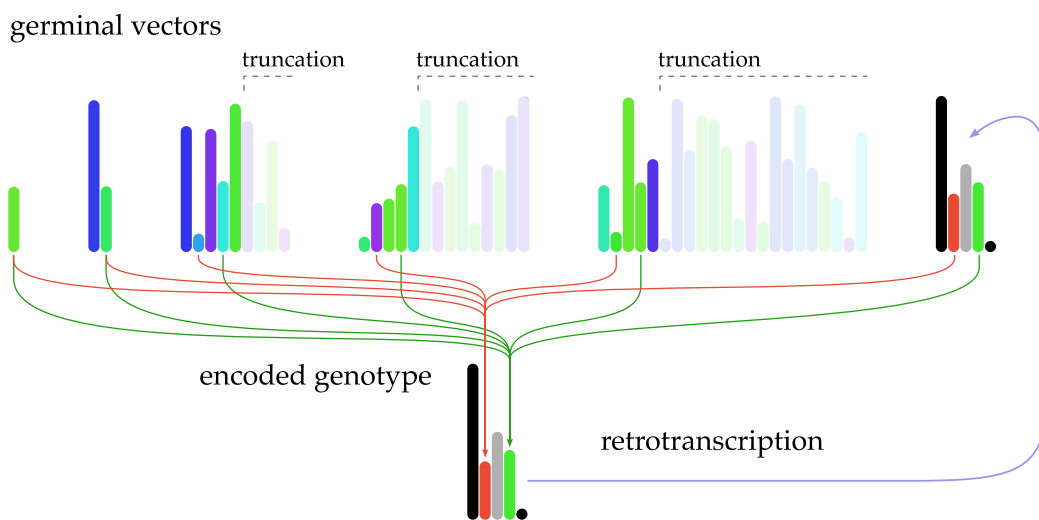


Figure 37: Equivalent germinal vectors to obtain the same genotype. As the red and blue arrows show, in this example, only two important values are determining the chosen function  $m$  and the parameter value 60. The retrotranscription of the encoded genotype vector is indicated by the blue arrow: the generated vector can be returned to the pool of its corresponding germinal vectors, as long as this vector generates itself. Colors help to easily identify retrotranscribed germinal vectors.

## 5.6. Retrotranscription of genotypes as germinal vectors

Figure 37 compares several equivalent germinal vectors corresponding to the encoded genotype of the minimal specimen generated with  $m(60)$ , seen in Figure 4, whose encGen key is  $[ 1, 0.326238, 0.53, 0.430607, 0 ]$ . This vector can be included back in the pool of germinal vectors because it is self-generating. That is exactly the desired feature called retrotranscription working as described in Section 2.5.

In GenoMus, the retrotranscription is an efficient method to find a germinal vector from a functional expression created or edited by hand. In other words, it enables a crucial process for the entire model to make sense: it allows for a kind of reverse engineering where, from any program, it obtains the decision tree that the metaprogramming process must follow to arrive at that specific program. The key point that should be emphasized is that *every encoded genotype is simultaneously an encoded representation of the executable program and the germinal vector that generates that same program.*<sup>58</sup>

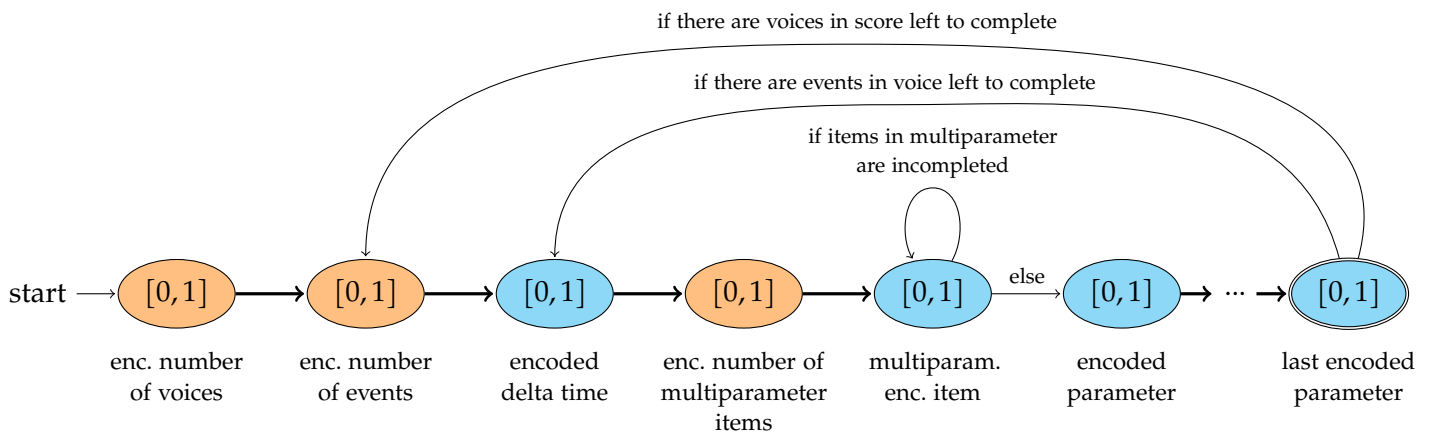


Figure 38: Automated rule set for encoding phenotypes into one-dimensional vectors. As in Figure 34, orange indicates golden encoded integers, used to indicate the total number of voices, total events within a voice, and total items within an event multiparameter.

## 5.7. Phenotype encoding

The evaluation of a decoded genotype results in a coded phenotype that, as seen in Listing 8, is included in each generated subspecimen under the key `encPhen`. The phenotype includes the musical score generated following a format similar to that used for the genotype encoding: both are one-dimensional vectors with values  $\in [0, 1]$ . This isomorphism between genotypes and phenotypes is a key characteristic of the model, as it enables experimentation with various machine learning paradigms that manipulate input and output data structured as sequences.

<sup>58</sup>It is noteworthy that this transformation also exists in genomics through the reverse transcriptase enzyme, which transcribes DNA from RNA, reversing the usual flow of genetic information.

Figure 38 illustrates, in the form of an automaton, the steps involved in this encoding, both to write these vectors and to decode them, converting them into the final output that can be translated into readable musical notation. It is important to note that the structure of the scores is simplified and reduced to specifying the number of voices, the number of events in each voice, and listing the parameters of each event. In Figures 12 and 13, it has been already shown this internal articulation of the scores.

In the encoding of phenotypes, golden encoded integers also play a significant role as they encode the number of voices, the number of events, and the number of items in multi-parameters such as `midipitch`.<sup>59</sup>

Given that the evaluation of each branch of a genotype yields a subspecies with its phenotype encoded in the key `encPhen`, the parent functions operate on these phenotypes to produce their own. In Listing 47, we observe part of the implementation of the function `vConcatV`, which concatenates two voices sequentially. It can be seen in lines 9-11 how it creates its `encPhen` property based on those inherited from its arguments. The `g2p` function encodes the total number of resulting events from the merge of voices as a golden encoded integer.

```

1  indexGenotypeFunction("vConcatV", "voiceF", 43, ["voiceF", "voiceF"]);
2  vConcatV = (v1, v2) => {
3    var encodedFuncID = g2p(43);
4    var totalEvents = v1.phenLength + v2.phenLength;
5    return indexDecGens({
6      funcType: "voiceF",
7      encGen: flattenDeep([1, encodedFuncID, v1.encGen, v2.encGen, 0]),
8      decGen: "vConcatV(" + v1.decGen + ", " + v2.decGen + ")",
9      encPhen: [g2p(totalEvents)]
10     .concat((v1.encPhen).slice(1))
11     .concat((v2.encPhen).slice(1)),
12     phenLength: totalEvents,
13     phenVoices: 1,
14     // more metadata
15   })
16 };

```

Listing 47: Implementation of `vConcatV`

<sup>59</sup>In this GenoMus version, I only utilize one type of multiparameter, applied to the pitch to ease chord handling. There is no restriction preventing the addition of further types of multiparameters if necessary, thereby creating new specific parameter types.

The following demonstrates the process starting from a decoded genotype. From the functional expression in Listing 48, a score with three voices is rendered. Note that the events in this example have an extra parameter, utilizing the function type *p* for generic parameters.

```

1 sAddV(
2   sConcatS(
3     s(
4       v(
5         e(
6           nRnd(), mRnd(), aRnd(), iRnd(), pRnd()),
7         s2V(
8           vAutoref(1),
9           vAutoref(1))),
10    vConcatE(
11      e5Chord(
12        nRnd(), m(54), mRnd(), mAutoref(2), mRnd(), mAutoref(1), aRnd(), iRnd(), p(0.6907)),
13      e3Chord(
14        nAutoref(1), m(63), mRnd(), mAutoref(3), aAutoref(1), iRnd(), p(0.515688)))

```

Listing 48: Simple decoded genotype using events with one extra parameter

The phenotype resulting from the evaluation of this expression will thus be the key `encPhen` which outputs the function of the main stem `sAddV`.

## 5.8. Phenotype decoding

Unlike genotypes, which can stem from their encoded version (exploring any numerical expression) and from the decoded version (writing or manually editing as in any other programming language), the decoding of the phenotype only occurs at the end of the evaluation process. This is a subprocess within the rendering of specimens.

The output subspecimen of any functional tree contains only the encoded phenotype in the numerical format just seen above. First, translation is performed into a readable structure—the *decoded phenotype*—, and from this data, conversions are applied to some form of graphical notation, audio synthesis, MIDI usage, or transformation into any other sequential format or real-time interaction.

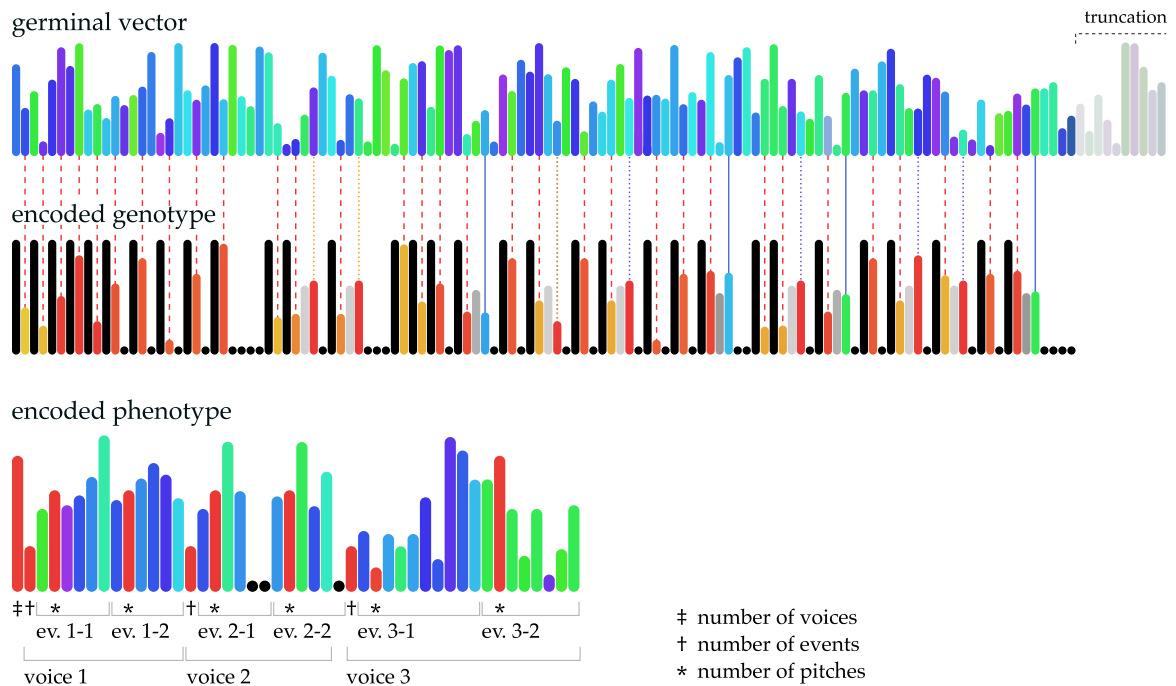


Figure 39: *Germinal vector, encoded genotype, and corresponding encoded phenotype. Visualization of the vectors related to the previous Listing 48. In this case, the germinal vector is longer than the encoded genotype, so it is truncated without the need for loops. The correspondences between values are marked similarly to the previous figure, although there is a new element: the orange dotted lines indicate the leaf values of autoreference functions, which also turn red when using golden encoded integers. The encoded phenotypes are also structured by golden integers, as can be observed in the visualization.*

Let's continue with the same example as in the previous figures to illustrate how this decoding is internally organized. In the rendering of the specimen, both versions of the phenotype are included. Listing 49 displays an excerpt of the complete specimen where the two corresponding keys, `encodedPhenotype` and `decodedPhenotype`, are gathered.

```
"encodedPhenotype" : [ 0.854102, 0.236068, 0.490318, 0.618034, 0.515814, 0.583115, 0.708952,
  0.993757, 0.551135, 0.618034, 0.698402, 0.80451, 0.724587, 0.564028, 0.236068, 0.490319,
  0.618034, 0.95, 0.612134, 0, 0, 0.576042, 0.618034, 0.948476, 0.508954, 0.744982, 0,
  0.236068, 0.339352, 0.09017, 0.318825, 0.234693, 0.318825, 0.570247, 0.146883, 0.982682,
  0.89011, 0.6907, 0.692587, 0.854102, 0.490026, 0.169299, 0.490026, 0.040777, 0.215134,
  0.515688 ],
"decodedPhenotype" : {
  "metadata" : {
    "totalVoices" : 3,
    "effectiveVoices" : 3,
    "totalEvents" : 6,
    "effectiveEvents" : 5,
    "eventsPerVoice" : [ 2, 2, 2 ],
    "effectiveEventsPerVoice" : [ 2, 1, 2 ],
    "durationsPerVoice" : [ 785.985, 487.516, 2641.904 ],
    "rhythmicDurationsPerVoice" : [ 536.278, 560.7, 624.35 ],
    "scoreDuration" : 2641.904,
    "rhythmicScoreDuration" : 624.35,
    "generalOnsetTime" : 0
  },
  "score" : {
    "voice-1" : {
      "event-1-1" : {
        "onset" : 0,
        "pitches" : [ 64 ],
        "duration" : 223.1092,
        "intensity" : 58.77,
        "param-5" : 0.993757
      },
      "event-1-2" : {
        "onset" : 242.51,
        "pitches" : [ 74 ],
        "duration" : 543.4745,
        "intensity" : 59.52,
        "param-5" : 0.564028
      }
    },
    "voice-2" : {
      "event-2-2" : {
        "onset" : 242.51,
        "pitches" : [ 99 ],
        "duration" : 245.0063,
        "intensity" : 60.55
      }
    },
    "voice-3" : {
```

```

"event-3-1" : {
  "onset" : 0,
  "pitches" : [ 42, 49, 54, 67 ],
  "duration" : 2641.9041,
  "intensity" : 70.4,
  "param-5" : 0.690759
},
"event-3-2" : {
  "onset" : 147.51,
  "pitches" : [ 44, 63 ],
  "duration" : 38.1472,
  "intensity" : 37.28,
  "param-5" : 0.515688
}
}
}
}

```

Listing 49: Data structure of a phenotype within a specimen

To cross-reference this data with the resulting music, the following Figure 40 illustrates the conversion into musical notation.

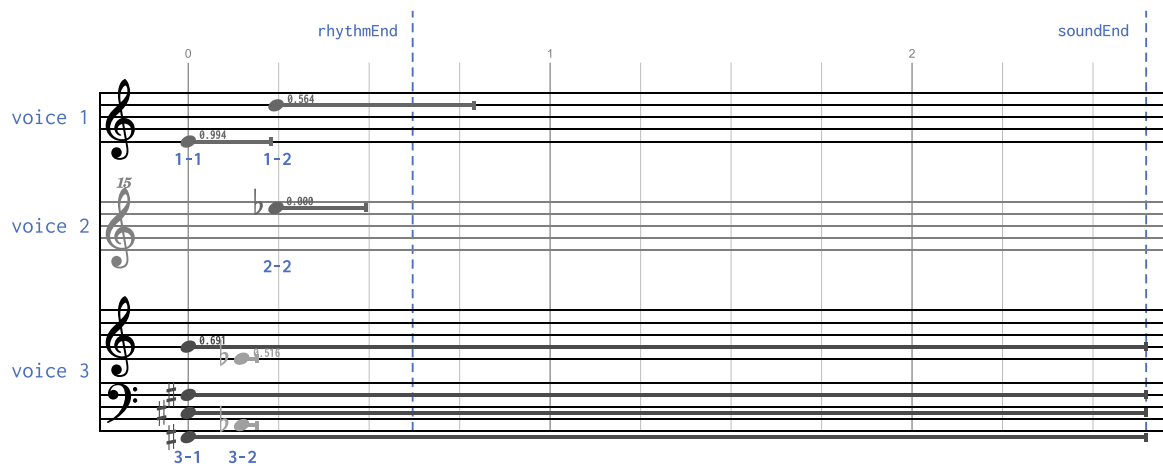


Figure 40: Score corresponding to the decodedPhenotype in Listing 49. The specimen's species includes one extra parameter, which is reflected as a numerical value alongside each note. In this example, the rhythmEnd and soundEnd points are also indicated, corresponding to the internal durations within the musical metric structure and the effective sound duration associated with articulation.



The `decodedPhenotype` consists of an initial block of metadata, followed by the actual score. Several important points should be noted during this conversion, as seen in the example:

- The metadata is quite self-explanatory, but it's essential to remember the distinction between rhythmic duration and sound duration of an event, which translates into the voice and complete score levels. These durations, often different, are decisive in the processes of joining voices and the score, as previously seen in Listing 17.
- While processing the decoded phenotype with the `decodePhenotype` function, events that do not produce sound are deleted. In the score data and subsequent musical notation, it can be observed that the second voice starts with event 2-2, as for the first one, the intensity parameter is zero, as seen in the visualization in Figure 39.
- Similarly, in cases where an event has multiple pitches, note repetitions are removed and reordered from low to high. For example, even though the `e5Chord` and `e3Chord` function generate chords with five and three pitches respectively, the resulting events 3-1 and 3-2 have one pitch less due to some repetition that occurred (as can be easily verified in the vector visualization).

## Specimens generation

“

Computer programming is tremendous fun. Like music, it is a skill that derives from an unknown blend of innate talent and constant practice. Like drawing, it can be shaped to a variety of ends —commercial, artistic, and pure entertainment. Programmers have a well-deserved reputation for working long hours but are rarely credited with being driven by creative fevers. Programmers talk about software development on weekends, vacations, and over meals not because they lack imagination, but because their imagination reveals worlds that others cannot see.

*Larry O'Brien and Bruce Eckel [51]*

After explaining the representation system employed by the model, I proceed to detail how operations are conducted with these abstractions. Several methods have been provided in parallel for the creation and manipulation of specimens. I will explain the essential design details to ensure the repeatability of results, the practical effectiveness of a generative process within a potentially vast search space with many unpredictable interactions, as well as the protections implemented to avoid security issues in the evaluation of programs written or edited by hand.

The second part of the chapter focuses on the next level of complexity: that which pertains to the production of populations of specimens, available processes for supervised and unsupervised candidate evaluation and selection, and the procedures used for generating new populations through the evolution of previous generations. This system draws inspiration from classical evolutionary algorithms but includes several peculiarities in the segmentation of the subprocesses that constitute each new series of modifications from a given population.

## 6.1. Metaprogramming of genotypes

### 6.1.1. Summary of the subprocesses

From the core process of the entire model, involving metaprogramming of computable expressions from initial conditions, to their transformation into an interactive score within the Max user interface, a significant number of intermediate subroutines are required. There are various methods for creating and transforming a specimen. Before delving into details, Table 10 provides an overview of the subprocesses that take place, aiding in better orientation across the following pages.

| Aux function name                  | Description   |
|------------------------------------|---|
| <b>Core functions</b>              |   |
| <b>createGenotype</b>              | Creates a deterministic genotype from initial conditions.   |
| <b>createSpecimen</b>              | Creates a brand new specimen.   |
| <b>renderSpecimen</b>              | Renders a specimen from minimal initial conditions and playback options stored as an Object.      |
| <b>renderInitialConditions</b>     | Creates specimen from current initial conditions.   |
| <b>specimenFromGerminalVector</b>  | Renders a specimen from a germinal vector.  |
| <b>regenerateEligibleFunctions</b> | Rewrites the complete eligible genotype functions library after a change in eventExtraParameters. |
| <b>specimenDataStructure</b>       | Formats specimen data structure as a JSON file exportable Object.                                 |
| <b>specimenMinimalData</b>         | Reduces a specimen data structure to the minimum needed to reconstruct the specimen.              |
| <b>Genotype auxiliary handling</b> |   |
| <b>encodeGenotype</b>              | Genotypes encoder.  |
| <b>decodeGenotype</b>              | Genotypes decoder.  |
| <b>formatDecGen</b>                | Expands and indents a compressed expression in a human-readable format.                           |
| <b>evalDecGen</b>                  | Encodes and decodes a genotype to filter invalid or dangerous expressions before being evaluated. |

- extractLeaves** Extracts leaves from an encoded genotype as an array.
- mutateSpecimenLeaves** Mutates only leaves of a specimen according to certain probabilities.
- replaceSpecimenBranch** Replaces a branch of a given output type in a specimen, with a brand new generated branch, and returns only the new decodedGenotype.

---

## UI Communication

- saveSpecimenAndUpdateUI** Synchronizes the information of the latest elements generated by the core code with the user interface.
- wrapEncodedPhenotype** Wraps encoded phenotypes into a scoreF function before processing phenotypes to enable working with simple genotypes of any output type.
- formatDecGen** Expands and indents a compressed expression in a human-readable format.
- setPlaybackRate** Playback rate adjustment, which rewrites the score according to this tempo factor.
- setEqualStepsPerOctave** Rewrites pitches according to a division of the octave in equal tempered parts.
- setQuantization** Rewrites notevalues of events according to a quantized time grid defined by a minimal duration.
- newRandomSpecimen** Creates a new random specimen from scratch, trying to satisfy a set of given constraints.
- mutateLeaves** Mutates only leaves values, according to specified probability and maximal amount of change.
- text** Creates specimen by direct text input in Max patch.
- encGenAsGerminal** Rewrites current specimen to get a retrotranscribed germinal vector. Germinal vector and decoded genotype are identical after this operation.
- replaceBranch** Removes a branch from the decoded genotype and replaces it with a brand new branch of the same function type.
- growHorizontally** Concatenates new randomly generated music to the current specimen.
- growVertically** Adds new randomly generated voices to the current specimen.

|                                  |  |
|----------------------------------|--|
| <b>setRating</b>                 | Stores last rating given by the user.  |
| <b>symbol</b>                    | Adds user comments incrementally.  |
| <b>deleteComments</b>            | Deletes all user comments.   |
| <b>decodedPhenotype2bachRoll</b> | Converts decoded phenotype into a bach roll compatible format to print and play the generated music. |

---

*Table 10: Functions related to creation and modification of specimens. The functions are grouped into categories: **core functions** perform fundamental operations for genotype synthesis, creation, and manipulation of the main data structures of the specimens. Under **genotype auxiliary handling**, some functions carry out very specific processes on genotypes. **UI Communication** gathers functions that are called from the Max interface or are necessary for proper interaction with it.*

### 6.1.2. The core metaprogramming subroutine

The core function of GenoMus is **createGenotype**, which generates a deterministic genotype from initial conditions. This function represents the actual implementation of the mapping tran as described in the conceptual framework in Section 2.5. Due to the importance of details for result repeatability, I reproduce the complete function in Listing 50. Besides the comments, the code is fairly self-explanatory.

```

1  var createGenotype = (
2      extraParameters,
3      specimenType,
4      localEligibleFunctions,
5      genotypeDepthThreshold,
6      listsMaxNumItems,
7      seedForAlea,
8      germinalVector
9  ) => {
10     var newGenotypeStartTime = new Date();
11     validGenotype = true;
12     initSubgenotypes();
13     // main variable
14     var newGenotype;
15     // regenerates functions library only if set of eligible functions changes
16     if (arrayEquals(localEligibleFunctions, eligibleFunctions) == false) {
17         eligibleFunctionsLibrary = createEligibleFunctionLibrary(
18             genotypeFunctionsLibrary, localEligibleFunctions);
19         createGenFuncOrderedEncIndices();
20     };
21     eligibleFunctions = [...localEligibleFunctions];

```

```
22 // if eventExtraParameters has changed eligible functions are regenerated
23 if (eventExtraParameters != extraParameters) {
24     eventExtraParameters = extraParameters;
25     regenerateEligibleFunctions();
26 };
27 // update global initial conditions
28 mainFunctionType = specimenType;
29 defaultDepthThreshold = genotypeDepthThreshold;
30 defaultListsMaxCardinality = listsMaxNumItems;
31 globalSeed = seedForAlea;
32 currentGerminalVector = [...germinalVector];
33 // aux variables
34 var germinalVectorLength = germinalVector.length;
35 var germinalVectorReadingPos = 0;
36 var preEncGen = []; // rewriting of germinal vector as a retrotranscribable enc. genotype
37 var newDecodedGenotype = "";
38 var genotypeDepth = 0;
39 var notFilledParameters = []; // number of parameters to fill per depth level
40 // functions types in process of fulfilling their arguments
41 var expectedFunctions = [specimenType]; // starting with the output type function
42 var chosenFunction, chosenEncIndex;
43 var openFunctionTypes = [];
44 var nextFunctionType = specimenType;
45 var valueForChoosingNewFunction, chosenFunctionInfo;
46 var newLeaf, preItemValue, listItem, newItemThreshold, conversionFunc, typeIdentifier;
47 // the writing of the encoded and encoded genotype starts here
48 do {
49     // adds a function
50     if (leafTypes.includes(nextFunctionType) == false) {
51         germinalVectorReadingPos++;
52         preEncGen.push(1); // replace germinal value with new function identifier
53         // chooses a new function
54         // if depth threshold is reached, the current funcType identity func. is chosen
55         if (notFilledParameters.length >= genotypeDepthThreshold) {
56             chosenFunction = Object.keys(
57                 eligibleFunctionsLibrary.functionLibrary[nextFunctionType])[0];
58             chosenEncIndex = eligibleFunctionsLibrary.
59                 functionNames[chosenFunction].encIndex;
60             germinalVectorReadingPos++;
61         }
62         else {
63             valueForChoosingNewFunction = germinalVector[
64                 germinalVectorReadingPos % germinalVectorLength];
65             germinalVectorReadingPos++;
66             chosenEncIndex = findEligibleFunctionEncIndex(
67                 orderedElegibleEncIndices[nextFunctionType], valueForChoosingNewFunction);
```

```

68     chosenFunction = eligibleFunctionsLibrary.encodedIndices[chosenEncIndex];
69 }
70 preEncGen.push(chosenEncIndex);
71 newDecodedGenotype += chosenFunction + "(";
72 // reads the expected parameters of the chosen function
73 openFunctionTypes[openFunctionTypes.length] = nextFunctionType;
74 notFilledParameters[notFilledParameters.length] = Object.keys(
75     eligibleFunctionsLibrary.
76     functionLibrary[nextFunctionType][chosenFunction].arguments).length;
77 expectedFunctions[notFilledParameters.length - 1] = chosenFunction;
78 if (notFilledParameters.length > genotypeDepth)
79     genotypeDepth = notFilledParameters.length;
80 }
81 // adds a leaf
82 else {
83     if (nextFunctionType != "voidLeaf") {
84         germinalVectorReadingPos++;
85         // reads leaf value
86         newLeaf = germinalVector[germinalVectorReadingPos % germinalVectorLength];
87         conversionFunc = leavesInfo[nextFunctionType].converter;
88         typeIdentifier = leavesInfo[nextFunctionType].ID;
89         newDecodedGenotype += conversionFunc(newLeaf);
90         // replaces germinal value with leaf type identifier
91         preEncGen.push(typeIdentifier, newLeaf);
92         germinalVectorReadingPos++;
93         preItemValue = germinalVector[
94             germinalVectorReadingPos % germinalVectorLength];
95         // important: as leaf types codes are numbers bigger or equal than 0.5,
96         // and newItemThreshold is always <= 0.5, leaf type id values
97         // always will pass when processing a retrotranscribed germinal vector.
98         listItem = 0;
99         newItemThreshold = 1 / ( listsMaxNumItems - listItem);
100        // if the leaf is actually a list
101        if (listLeafTypes.includes(nextFunctionType)) {
102            while (preItemValue >= newItemThreshold
103                && listItem < listsMaxNumItems) {
104                germinalVectorReadingPos++;
105                newLeaf = germinalVector[
106                    germinalVectorReadingPos % germinalVectorLength];
107                germinalVectorReadingPos++;
108                preItemValue = germinalVector[
109                    germinalVectorReadingPos % germinalVectorLength];
110                newDecodedGenotype += "," + conversionFunc(newLeaf);
111                preEncGen.push(typeIdentifier, newLeaf);
112                listItem++;
113                newItemThreshold = 1 / (listsMaxNumItems - listItem);
114            }

```

```

115     }
116   }
117   notFilledParameters[notFilledParameters.length - 1]--;
118   // if all parameters of this depth level are written,
119   // deletes this count level and adds ")"
120   if (notFilledParameters[notFilledParameters.length - 1] == 0) {
121     do {
122       if (notFilledParameters.length > 1) {
123         notFilledParameters.pop();
124         expectedFunctions.pop();
125         openFunctionTypes.pop();
126       }
127       germinalVectorReadingPos++;
128       newDecodedGenotype += ")";
129       preEncGen.push(0); // replaces germinal value with closing function flag
130       notFilledParameters[notFilledParameters.length - 1]--;
131     } while (notFilledParameters[notFilledParameters.length - 1] == 0
132             && validGenotype == true)
133   }
134   if (notFilledParameters[0] > 0) newDecodedGenotype += ",";
135 }
136 chosenFunctionInfo = eligibleFunctionsLibrary.functionLibrary
137   [openFunctionTypes[openFunctionTypes.length - 1]]
138   [expectedFunctions[expectedFunctions.length - 1]];
139 nextFunctionType = chosenFunctionInfo.arguments[
140   chosenFunctionInfo.arguments.length - notFilledParameters[
141     notFilledParameters.length - 1]];
142 } while (notFilledParameters[0] > 0
143         && new Date() - newGenotypeStartTime < maxIntervalPerNewGenotype);
144 newDecodedGenotype.substring(0, newDecodedGenotype.length - 1); // omits trailing commas
145 reinitSeed(seedForAlea);
146 if (checkParenthesesBalance(newDecodedGenotype) == false) validGenotype = false;
147 if (validGenotype == false) {
148   newGenotype = evalExpr("s(v(" + defaultEvent + "))");
149   subGenotypes = {};
150   newGenotype.data = {
151     specimenID: getSpecimenDateName("not_viable_genotype"),
152     eventExtraParameters: extraParameters,
153     specimenType: "scoreF",
154     germinalVector: germinalVector,
155     localEligibleFunctions: localEligibleFunctions,
156     depthThreshold: genotypeDepthThreshold,
157     maxListCardinality: listsMaxNumItems,
158     seed: seedForAlea,
159     depth: 4,
160     leaves: extractLeaves(newGenotype.encGen),
161     playbackRate: playbackRate,

```



```
162     minQuantizedNotevalue: minQuantizedNotevalue,  
163     stepsPerOctave: stepsPerOctave,  
164     rating: 0,  
165   };  
166 }  
167 else {  
168   newGenotype = evalExpr(newDecodedGenotype);  
169   newGenotype.data = {  
170     specimenID: getSpecimenDateName(currentUser),  
171     eventExtraParameters: eventExtraParameters,  
172     specimenType: specimenType,  
173     localEligibleFunctions: eligibleFunctions,  
174     depthThreshold: genotypeDepthThreshold,  
175     maxListCardinality: listsMaxNumItems,  
176     seed: seedForAlea,  
177     germinalVector: germinalVector,  
178     renderTime: r3d((new Date() - newGenotypeStartTime) * 0.001),  
179     encGenotypeLength: newGenotype.encGen.length,  
180     decGenotypeLength: newDecodedGenotype.length,  
181     depth: genotypeDepth,  
182     leaves: extractLeaves(newGenotype.encGen),  
183     germinalVectorDeviation: arraysDistance(newGenotype.encGen, germinalVector),  
184     playbackRate: playbackRate,  
185     minQuantizedNotevalue: minQuantizedNotevalue,  
186     stepsPerOctave: stepsPerOctave,  
187     rating: 0  
188   };  
189 }  
190 return newGenotype;  
191 }
```

Listing 50: Implementation of core function `createGenotype`

Let's highlight some important details of this process which lies at the core of the transformations:

- The seven required arguments are the initial conditions described in Table 9.
- The variables `preEncGen` and `newDecodedGenotype` store the genotype in its encoded and decoded versions respectively. In the case of `preEncGen`, the received values in `germinalVector` are converted or replaced according to the rules shown in Figure 34.
- As previously discussed, if the nesting level of functions reaches the threshold determined by the argument `genotypeDepthThreshold`, from that point onward, only identity functions of the required type are used, eventually closing that branch of

the functional tree without new ramifications. This process takes place in lines 55-61. In the construction of the Genotype functions library, the first function of each type must be that identity function, accessed in lines 56-57.

- The function `findEligibleFunctionEncIndex`, at line 66, approximates the nearest encoded index among the functions included in the argument `localEligibleFunctions`, following the procedure explained in Section 5.1.4.
- The maximum number of elements appearing in a list is limited by the initial condition `maxListCardinality`, and depends on how many elements from the germinal vector exceed the threshold set by `newItemThreshold`. If it surpasses this threshold, a new number is added to the current list; otherwise, the list terminates at that point, and the flag value of a closing function is inserted. To achieve equiprobability among lists of any length within the permitted maximum, this threshold is modulated on line 113 according to the Equation 29:

$$t_n = \frac{1}{max_{card} - n} \quad (29)$$

where  $n$  is the position of an item in a list (starting with position 0),  $t_n$  is the value used by `newItemThreshold` to decide if a new value will be included in the list, and  $max_{card}$  is the maximal list length allowed by initial condition `maxListCardinality`.

- If, at the end of the metaprogramming process, the genotype is declared invalid, in line 146, to prevent a fatal error that halts other higher-level processes, an output genotype using the `defaultEvent` is provided. This default expression is used for various purposes at different points throughout the code.
- Should the genotype be valid, the functional expression concluded at line 168 is evaluated with `evalExpr`, generating the subspecimen based on the data structure I previously discussed in the section. Along with this information, the data block is added, which will later be used to construct the complete specimen.
- The direct evaluation of expressions contained in strings is a practice that can entail significant security risks. To circumvent these dangers, the algorithm uses the safer auxiliary function `evalExpr` (Listing 51) instead of the `eval` function. More importantly, the main filter is the function `evalDecGen`, which is used with direct text inputs and analyzes and reconstructs the expression. If it does not identify all functions as part of the available library, it returns an error and does not evaluate the expression.

```
var evalExpr = str => Function("return " + str)();
```

*Listing 51: Function `evalExpr` as alternative to `eval`*

Listing 52 provides an example of calling the `createGenotype` function, as it occurs internally in the higher-level processes discussed next.

```
1 createGenotype(  
2   3, // extra parameters  
3   'scoreF', // main function type  
4   [ 1,2,3,4,5,6,7,9,10,11,12,26,27,28,29,41,42,43,44,48,58,98,99 ], // eligible functions  
5   6, // depth threshold  
6   10, // lists max. length  
7   606599857109, // global seed value  
8   [ 0.94, 0.26, 0.178 ] // germinal vector  
9 )
```

*Listing 52: A call to the function `createGenotype`*

The Object returned, in an intermediate stage between the subspecimen and the specimen, includes several new keys necessary for subsequent processes, which I will detail later on. The decoded genotype contained in `decGen` is a compact string. Listing 53 presents it in a formatted version, allowing for a better study of several of its features.

```
1 {  
2   funcType: 'scoreF',  
3   encGen: [ 1, 0.193496, 1, 0.472136, 1, 0.854102, 1, ... 243 more items ],  
4   decGen: 'sConcatS(s(v(e(nRnd(),mRnd(),aRnd(),iRnd(),pRnd(),pRnd()))),sConcatS(  
5     sConcatS(sAutoref(1),sConcatS(sConcatS(sAutoref(1),sConcatS(s(v(e(n(2.645),m(44),a(39),i  
6       (76.52),p(0.178),p(0.26),p(0.94))))),s(v(e(n(0.07178),m(50),a(502),i(34.99),p(0.26),p  
7       (0.94),p(0.178))))),sConcatS(sAutoref(3),sConcatS(s(v(e(n(2.645),m(44),a(39),i(76.52),p  
8       (0.178),p(0.26),p(0.94))))),s(v(e(n(0.07178),m(50),a(502),i(34.99),p(0.26),p(0.94),p  
9       (0.178)))))))))',  
5   encPhen: [ 0.618034, 0.562306, 0.426898, 0.618034, 0.301631, 0.062601, ... 68 more items ],  
6   phenLength: 9,  
7   phenVoices: 1,  
8   harmony: { root: 0.301631 },  
9   data: {  
10    specimenID: 'jlm-20231220_153902054-46',  
11    eventExtraParameters: 3,  
12    specimenType: 'scoreF',  
13    localEligibleFunctions: [ 1,2,3,4,5,6,7,9,10,11,12,26,27,28,29,41,42,43,44,48,58,98,99 ],
```

```
14   depthThreshold: 6,  
15   maxListCardinality: 10,  
16   seed: 606599857109,  
17   germinalVector: [ 0.94, 0.26, 0.178 ],  
18   renderTime: 0.004,  
19   encGenotypeLength: 250,  
20   decGenotypeLength: 410,  
21   depth: 10,  
22   leaves: [  
23     [ 61, 0.940006, 2.64499 ],  
24     [ 66, 0.172821, 44 ],  
25     [ 71, 0.261308, 39 ],  
26     [ 76, 0.940024, 76.52 ],  
27     [ 81, 0.178, 0.178 ],  
28     [ 86, 0.26, 0.26 ],  
29     ... 22 more items  
30   ],  
31   germinalVectorDeviation: 247.948504,  
32   playbackRate: 1,  
33   minQuantizedNotevalue: 0,  
34   stepsPerOctave: 12,  
35   rating: 0  
36 }
```

Listing 53: Object returned by `createGenotype`

Among other constraints, the arguments in this example define a species of events with three extra parameters, and use an exceptionally short germinal vector, consisting of only three values, to easily illustrate how the loop process in reading and transforming those three numbers is reflected in the writing of the genotype, as shown in the following Listing 53. Structures that repeat due to the cyclic reading of this triplet can be observed. Specifically, from line 3 onwards, the loop creates a nested iteration of the expression `sConcatS(sConcatS(sAutoref(1))` which would otherwise repeat, creating a branch of unlimited depth. However, due to the limitation imposed by `depthThreshold = 6`, only identity functions are used from line 10 onwards, as their configuration solely calls leaf values. Once that branch is closed and the current depth level is  $< 6$ , it is possible again to select among all eligible functions to continue the metaprogramming process.

```

1 sConcatS(
2   s(v(e(nRnd(), mRnd(), aRnd(), iRnd(), pRnd(), pRnd(), pRnd()))),
3   sConcatS(
4     sConcatS(
5       sAutoref(1),
6       sConcatS(
7         sConcatS(
8           sAutoref(1),
9           sConcatS(
10            s(v(e(n(2.645), m(44), a(39), i(76.52), p(0.178), p(0.26), p(0.94))))),
11            s(v(e(n(0.07178), m(50), a(502), i(34.99), p(0.26), p(0.94), p(0.178)))))),
12          sConcatS(
13            sAutoref(3),
14            sConcatS(
15              s(v(e(n(2.645), m(44), a(39), i(76.52), p(0.178), p(0.26), p(0.94))))),
16              s(v(e(n(0.07178), m(50), a(502), i(34.99), p(0.26), p(0.94), p(0.178))))))),
17          sAutoref(3)))

```

Listing 54: Decoded genotype generated by a germinal vector of only three values

## 6.2. Formatting of specimens

With the Object supplied by `createGenotype`, we are now in a position to complete a full specimen, using the data architecture outlined in Table 2. This action is carried out by the function `specimenDataStructure`, as shown in Listing 55. In addition to formatting the received data, this function performs some additional actions in its initial lines before returning the specimen:

- At line 2, it formats the decodedGenotype so that it can be displayed in the Max interface in a readable manner, introducing indentation, line breaks, and other elements. It performs this formatting in various compactness options to accommodate each user's preferences. These formatted strings are passed as a separate JSON dictionary to avoid extending the specimen with redundant information.
- At line 3, it calculates the length of the encoded genotype.
- At line 4, if the output is not of type `scoreF`, it wraps the decoded genotype in the necessary layers to make it so, facilitating the direct editing and testing of any type of function in the interface.
- At line 5, it decodes the phenotype, which until now existed only in its encoded version. This results in a structure as studied in Listing 49.

- At line 6, the decoded phenotype is converted into the appropriate format for the `bach.roll` module of the interface to display the interactive score in Max.
- At line 7, it calculates the so-called `generativityIndex`, which is simply the ratio between the length of the `bach.roll` array and that of the encoded genotype. Section 6.3.4 discusses how is this calculated.
- From line 8 onward, the complete structure constituting the specimen is constructed and returned.

```
1 var specimenDataStructure = specimenData => {
2   formatDecGen(specimenData.decGen);
3   var encGenotypeLength = specimenData.data.encGenotypeLength;
4   var wrappedEncPhen = wrapEncodedPhenotype(specimenData);
5   var decPhen = decodePhenotype(wrappedEncPhen);
6   var bachRoll = decodedPhenotype2bachRoll(decPhen);
7   var generativityIndex = r2d(bachRoll.length / encGenotypeLength);
8   return ({
9     metadata: {
10      specimenID: specimenData.data.specimenID,
11      comments: specimenData.data.comments,
12      GenoMusVersion: GenoMusVersion,
13      rating: specimenData.data.rating,
14      duration: r3d(bachRoll[3] * 0.001),
15      voices: decPhen.metadata.effectiveVoices,
16      events: decPhen.metadata.effectiveEvents,
17      depth: specimenData.data.depth,
18      encGenotypeLength: encGenotypeLength,
19      decGenotypeLength: specimenData.data.decGenotypeLength,
20      generativityIndex: generativityIndex,
21      germinalVectorLength: specimenData.data.germinalVector.length,
22      germinalVectorDeviation: specimenData.data.germinalVectorDeviation,
23      iterations: specimenData.data.iterations,
24      millisecondsElapsed: specimenData.data.millisecondsElapsed,
25      renderTime: r3d(specimenData.data.renderTime * 0.001),
26      history: specimenData.data.history
27    },
28    initialConditions: {
29      eventExtraParameters: specimenData.data.eventExtraParameters,
30      specimenType: specimenData.data.specimenType,
31      localEligibleFunctions: specimenData.data.localEligibleFunctions,
32      depthThreshold: specimenData.data.depthThreshold,
33      maxListCardinality: specimenData.data.maxListCardinality,
34      seed: specimenData.data.seed,
```

```
35     germinalVector: specimenData.data.germinalVector
36   },
37   playbackOptions: {
38     playbackRate: specimenData.data.playbackRate,
39     minQuantizedNotevalue: specimenData.data.minQuantizedNotevalue,
40     stepsPerOctave: specimenData.data.stepsPerOctave
41   },
42   encodedGenotype: specimenData.encGen,
43   decodedGenotype: specimenData.decGen,
44   encodedPhenotype: specimenData.encPhen,
45   subgenotypes: subGenotypes,
46   leaves: specimenData.data.leaves,
47   decodedPhenotype: decPhen,
48   roll: bachRoll,
49   });
50  };
```

Listing 55: Implementation of specimens creator **specimenDataStructure**

The Object returned by **specimenDataStructure** is what I defined as *rendered specimen*, which contains all the genotype evaluation results necessary for its representation as music. Storing specimens can be done in two ways: by saving all this information or by limiting it to the necessary information to recreate it again. This is what was previously defined as *minimal data specimen*. The brief function **specimenMinimalData**, in Listing 56, is limited to creating an Object that only replicates those minimal elements.

The minimal data specimens, due to their reduced size, will be particularly useful in the next stage of processing to expedite the generation of populations comprising numerous specimens for tasks involving evaluation, selection, and evolution.

```
1  var specimenMinimalData = renderedSpecimen => {
2    return ({
3      metadata: {...renderedSpecimen.metadata},
4      initialConditions: {...renderedSpecimen.initialConditions},
5      playbackOptions: {...renderedSpecimen.playbackOptions}
6    });
7  };
```

Listing 56: Implementation of **specimenMinimalData**

However, the UI allows saving a specimen in its rendered version, including complete data from its evaluation and conversion to a score, enabling third-party applications to access this information. Specifically, the `decodedPhenotype` has a straightforward structure that can be easily converted into other standard formats such as MusicXML, MIDI, etc. Indeed, the conversion to a Bach roll and the rendering of scores into SVG are performed by accessing this data block.

## 6.3. Specimen metadata

The data contained in the metadata block has already been briefly explained in Table 2. Most of them are self-explanatory, hence, I will only elaborate on the aspects that require clarification.

### 6.3.1. specimenID

To set up a unique identifying label for each specimen, a string is composed using the elements detailed in Figure 41. In addition to the username and the date and time of generation, a number is added corresponding to the order of valid specimens generated in the current session. This last number is necessary to avoid identical identifiers if more than one specimen is generated within the same millisecond.

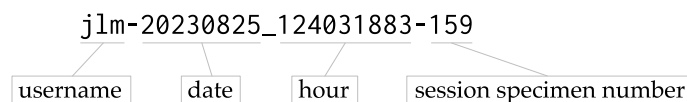


Figure 41: Composition of the specimenID to assign unique specimen names. This order ensures a chronological arrangement when displaying specimens in a list.

### 6.3.2. comments

User comments can be added at any time. A specimen can result from numerous successive transformations, both manual and automatic. Comments are incremental and can be seen as a log file for the user, aiding in recalling and identifying the selected material.



### 6.3.3. rating

The rating assigned to a specimen is also a normalized value between 0 and 1. Only the last recorded value is collected, so it might have been modified several times throughout its transformations. The rating can be set manually by a user or automatically in processes designed to apply a function that evaluates its fitness value.

### 6.3.4. generativityIndex

In the context of specimen metaprogramming, I called *index of generativity* to the ratio between the length of the bach roll array and that of the encoded genotype. This measure gives us information about whether it exhibits a more declarative or generative-style process: a higher `generativityIndex` indicates that more musical text has been generated in proportion to the length of the code that generated it.

The bach roll array is used instead of the encoded phenotype because the former solely encompasses effective events and voices, whereas the latter comprises all that is generated before filtering out elements that are silent or redundant. Hence, the measurement of generativity is more accurate by excluding those fragments of the phenotype that are going to be discarded. This does not imply any kind of aesthetic evaluation or fitness; it is merely a quantitative estimation about the nature of the compositional procedures used.

### 6.3.5. germinalVectorDeviation

As we have seen in Section 5.6 on retrotranscription, if a genotype is generated from any numerical sequence, transformations are applied to that vector to ensure it can be autotranscribed. The property `germinalVectorDeviation` compares the encoded genotype with the germinal vector and measures the discrepancy between both vectors using the function `arraysDistance` that employs Formula 30 to sum up the accumulated deviations of each element in a vector with its counterpart:

$$\text{deviation}(\text{germ}V, \text{enc}G) = |\text{germ}V_l - \text{enc}G_l| + \sum_{n=1}^m |\text{germ}V_n - \text{enc}G_n| \quad (30)$$

where `germV` is the germinal vector, `encG` is the encoded genotype generated from `germV`,  $n$  is the  $n$ -th element of a vector,  $l$  is the length of a vector, and  $m = \min(\text{germ}V_l, \text{enc}G_l)$  is the length of the shortest vector.

As can be seen, in the highly probable case where the germinal vector and encoded genotype do not have the same length, each surplus-value adds 1 to the overall sum, increasing that deviation as a penalty due to their length difference. When the germinal vector has been computed through retrotranscription, for example when the genotype has been manually edited, the deviation will be 0, as it is a backward-generated germinal vector. In such cases, a value of  $\text{deviation}(\text{germV}, \text{encG}) < 0$  implies that there is some error in the retrotranscription process. Debugging was the main reason for adding this feature in the metadata, but aside from this, it may have other uses in comparing transformations of the same specimen.

### 6.3.6. history

Under this key, a log is being written, documenting all the actions that have been performed since the creation of the specimen. Listing 57 shows an example of specimen history that has undergone several stages of manual and automatic transformations. In addition to noting the type of transformation, some log entries indicate the number of voices and events and the duration of the generated score.

```

1 "history" : {
2   "1" : "Random new specimen - 4v 34e 10.168s",
3   "2" : "Mutated leaves, probability 0.5, range 0.3 - 4v 34e 10.116s",
4   "3" : "Selected as elite spec. #5 with rating 0.551 for generation 2 from generation 1",
5   "4" : "Seed 205862798854746 to 67538728584317 - 4v 34e 24.929s",
6   "5" : "Grown vertically - 5v 35e 24.929s",
7   "6" : "Grown horizontally - 6v 80e 44.6s",
8   "7" : "Typed genotype, novelty 0.0065 - 6v 80e 37.731s",
9   "8" : "Selected as elite spec. #2 with rating 0.856 for generation 3 from generation 2"
10 }

```

Listing 57: Example of history in a specimen metadata

Entry 7 of this history points that the genotype was manually edited, indicating a *novelty* value. This is an index measured by the function `stringsDistance`, which gauges how much the edited code has changed within a range from 0 to 1. This is accomplished by using Levenshtein distance as the basis.<sup>60</sup> If the resulting value is greater than 0.5, it is considered so different from the original that it is treated as a new specimen (based on fragments of previous code but deemed too distinct to be considered a simple variation).

<sup>60</sup>The Levenshtein distance is an algorithm used to determine the distance between two text strings. Put simply, it considers how many editing operations are needed to transform one string into another. Among its various applications, it is also used in the computational study of genetic material sequencing. [16]

## 6.4. Playback options as epigenetic conditions

Alongside the `initialConditions` and metadata, we have seen how minimal data specimens include `playbackOptions` as an additional block that completes the necessary information to produce a rendered specimen. Returning to the bioinspired metaphor, these determinants would be equivalent to "epigenetic mechanisms," given that epigenesis refers to external circumstances that induce changes in the final development of a phenotype. Thus, two specimens with the same germinal vector and identical initial conditions may exhibit phenotypes with very different characteristics based on changes in the `playbackOptions`.

We can then view these determinants as adjustments and transformations made in the final phase of the process. If the `initialConditions` are constraints before the metaprogramming process, the `playbackOptions` allow users to make global changes in the last phase of the generative process. This enables, above all, simple control from the interface that conveniently modifies properties that can be very important in practical composition work. Hence, it is straightforward to ensure that several specimens have the same rhythmic grid, share attributes concerning their tempo, harmony, etc.

At the moment, only three of these final options have been implemented as a proof of concept, which I detail below. However, there is a considerable number of other similar options that affect dimensions such as articulation, dynamics, interval transposition, etc., and these will be added in upcoming versions.

### 6.4.1. Tempo control with `playbackRate`

This modification is technically trivial but crucial for the user to fine-tune the result to the ideal tempo. When the value of `playbackRate` is changed, it simply alters the corresponding key of the specimen. Since this modification is only superficial and does not imply alterations beyond tempo, changes in the playback rate do not generate a new specimen but rather overwrite the data of the current one.

### 6.4.2. Rhythm quantization with `minQuantizedNotevalue`

The minimum rhythmic quantization notevalue establishes a minimum duration in seconds for any event and generates a series of multiples to which all durations are adjusted in seeking the closest value to the original. A `minQuantizedNotevalue = 0` means that there is no quantization at all. One of the most obvious uses of this option is to

standardize and combine many specimens on the same rhythmic grid, achieving rhythmic structures without excessive complexities.

To provide an example illustrating the influence of this condition on the same genotype, let's start from the expression in Listing 58. This functional tree already has a certain depth and a good number of branches.

```

1  sConcatS(
2    sAddS(
3      sHarmonicGrid(
4        sConcatS(
5          sConcatS(
6            sAddV(
7              sConcatS(
8                s(
9                  vPerpetuumMobile(
10                     nRnd(),
11                     lmLine(
12                       m(77),
13                       m(64),
14                       q(-9)),
15                     la(138, 86, 132, 3, 49, 37, 261),
16                     liWrap(
17                       lFibonacci(
18                         p(0.691632),
19                         p(0.994283),
20                         p(0.183172),
21                         p(0.464113),
22                         q(-35))))),
23             sAddS(
24               sConcatS(
25                 sAddV(
26                   sConcatS(
27                     s(v(e(n(0.28303),m(106),a(7),i(34.83))))),
28                     s(v(e(n(0.30092),m(74),a(124),i(38.79))))),
29                   vMotifLoop(
30                     ln(0.14889, 0.40164),
31                     lm(82, 75),
32                     la(60, 22, 32, 246),
33                     li(30.04, 54.43, 30.77, 30.97, 52.91, 76.19, 42.96, 55.81, 62.52))),
34                 sAutoref(5)),
35             s(
36               vPerpetuumMobile(
37                 nRnd(),
38                 lm(54, 36, 60, 36, 31),
39                 laWrap(
40                   l(0.298616, 0.974181, 0.181658, 0.9688, 0.724786)),
41                 liAutoref(2))))),
42     vABCAB(

```

```
43     vMotifLoop(  
44         ln(0.54272, 1.36524, 0.0341, 0.12948, 0.35474, 0.08418, 0.66646),  
45         lmWrap(  
46             l5P(  
47                 pAutoref(3),  
48                 pAutoref(3),  
49                 pAutoref(1),  
50                 pRnd(),  
51                 pAutoref(3))),  
52         laAutoref(1),  
53         li(81.5, 18.37, 73.3, 20.61, 60.71, 71.63, 81.61, 49.91, 41.57, 41.37)),  
54     vMotif(  
55         lnLine(  
56             nRnd(),  
57             nRnd(),  
58             q(-12)),  
59         lm(59, 71, 85, 75, 53, 119, 97, 70),  
60         laWrap(  
61             l2P(  
62                 pAutoref(2),  
63                 p(0.176112))),  
64         liWrap(  
65             lBrownian(  
66                 p(0.965725),  
67                 pAutoref(3),  
68                 q(-10),  
69                 p(0.089379))),  
70     vPerpetuumMobileLoop(  
71         n(0.26927),  
72         lmWrap(  
73             lTribonacci(  
74                 pAutoref(8),  
75                 pAutoref(3),  
76                 pGaussianRnd(),  
77                 pRnd(),  
78                 p(0.803329),  
79                 q(-7))),  
80         laWrap(  
81             lGaussianRnd(  
82                 pAutoref(3),  
83                 q(18))),  
84         liWrap(  
85             l4P(  
86                 pAutoref(1),  
87                 pAutoref(1),  
88                 pGaussianRnd(),  
89                 pAutoref(3))))),  
90     sHarmonicGrid(  
91         sHarmonicGrid(  
92             s2V(  
93                 vAutoref(3),  
94                 vMotifLoop(  
95                     ln(0.10998, 1.31791, 0.37039, 0.04489, 0.78768),  
96                     lmWrap(  

```

```

97         lFibonacci(
98             p(0.126965),
99             p(0.298616),
100            p(0.159767),
101            p(0.027738),
102            q(26)),
103     laRemap(
104         la(70, 17, 117, 36),
105         aRnd(),
106         aAutoref(2)),
107     li(15.84, 34.97, 46.39, 30.79, 58.09, 50.62, 42.88, 58.67)),
108     hBluesScale(
109         m(28)),
110     hBluesScale(
111         mAutoref(5))),
112     sAutoref(3)),
113     hNaturalScale(
114         mRnd()),
115     sAutoref(3)),
116     sAutoref(3))"

```

Listing 58: Decoded genotype with `depthThreshold = 12`

Figure 42 displays in parallel five versions of the phenotype corresponding to different quantization values, ranging from no quantization to an adjustment with minimum values of one second. Quantization is independent of the `playbackRate`, allowing both configurations to be combined to achieve different rhythmic textures, regularities, and densities.

As a complement, Figure 43 shows the beginning of the encoded phenotypes of these five variations. It can be globally appreciated how the rendered sequence was altered. The modifications caused in the specimen are relatively significant, so each new value of `minQuantizedNotevalue` generates a new specimen with a different ID label, allowing for a quick comparison between variants.

minimal quantized notevalue

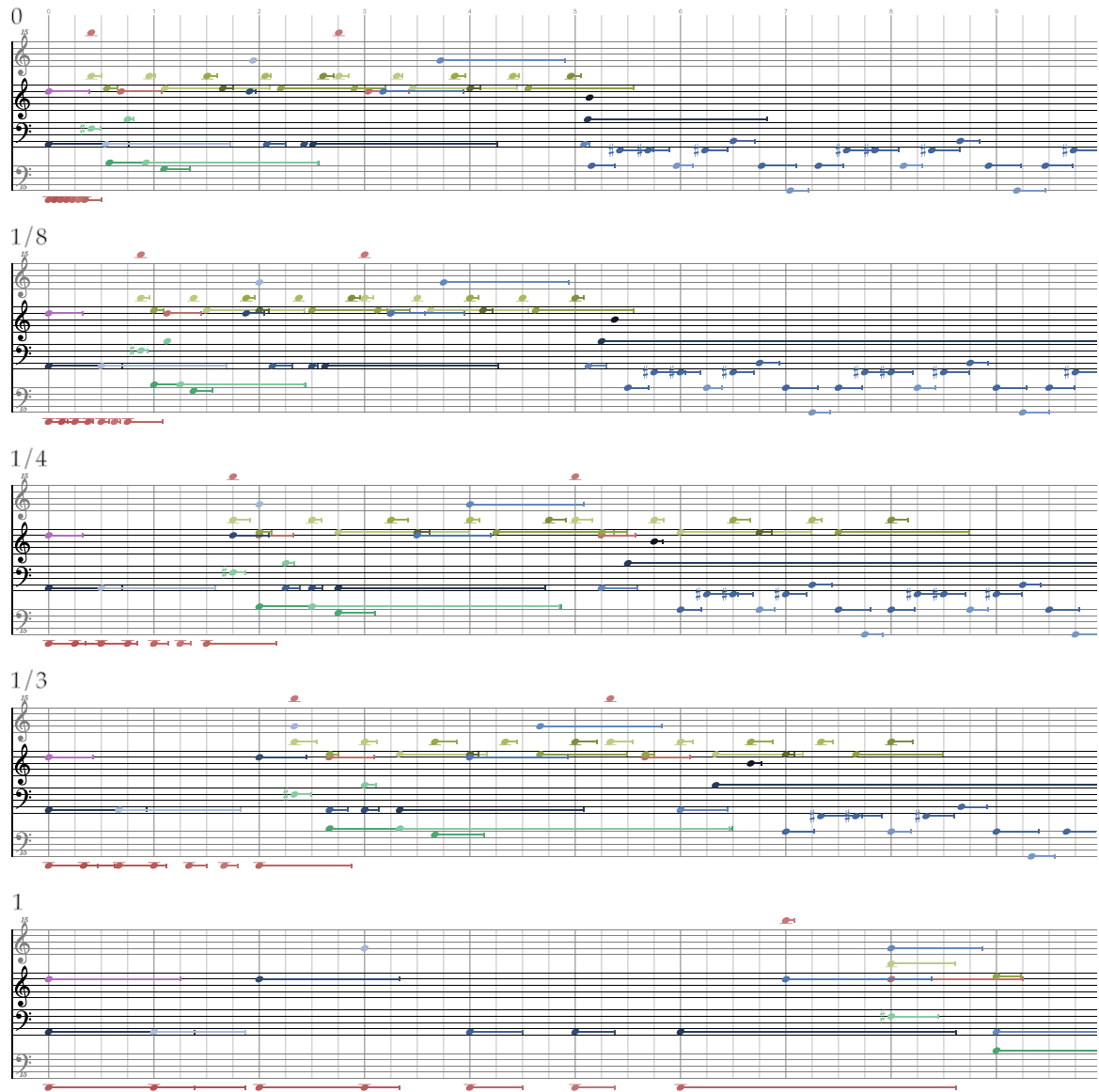
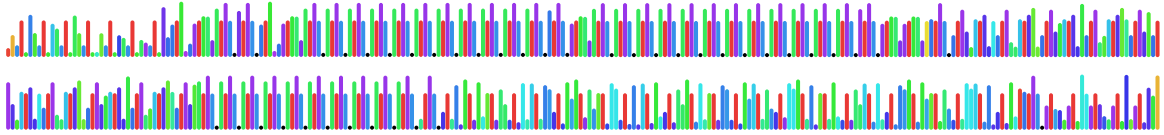


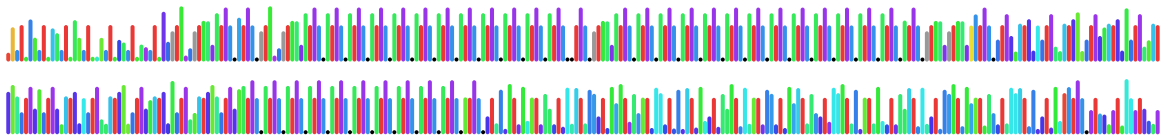
Figure 42: Comparison of the same phenotype with different quantization values. The first one does not quantize durations at all. The following ones establish increasingly longer values for minimum durations. These examples use divisors of 1 for `minQuantizedNotevalue`, making it easy to see the alignment of the events to the timegrid. The color hue indicates different voices. Note that quantization is global and affects all voices.

minimalQuantizedNotevalue =

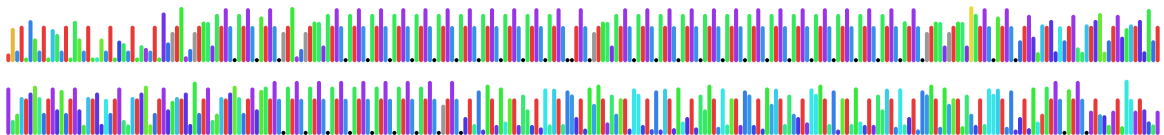
0



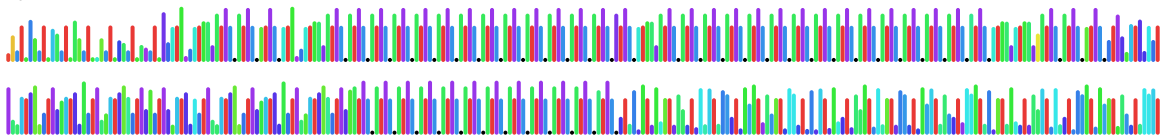
1/8



1/4



1/3



1

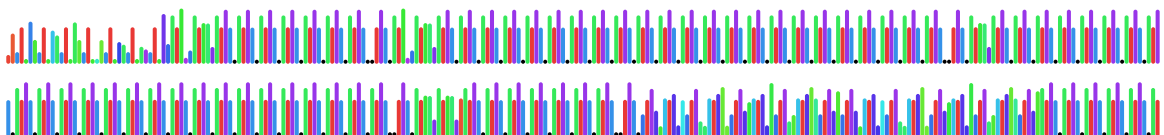


Figure 43: Visualized encoded phenotypes with variations in `minQuantizedNotevalue`. Only the beginning of each vector is visualized. Note that several blocks can be distinguished, and their progressive alteration as the established minimum duration becomes larger.

### 6.4.3. Equal temperaments with `stepsPerOctave`

The possibility to modify the global tuning has been implemented in a rather straightforward manner, as a way to test possibilities. However, this does not prevent future alternative options to globally adjust the pitches of a specimen to any harmonic grid from



the interface. In the current version of the model, the only way to alter the harmony globally is controlled with `stepsPerOctave`, determining how many equal parts the octave is divided into. This division must be combined with the representation made by the bach roll, which allows for adjusting the precision of representing accidental alterations, consequently altering the frequencies of the final pitches of each event. Figure 44 demonstrates how this tuning control affects the global harmony of a specimen.

stepsPerOctave = 12      12 equal semitones (default)

stepsPerOctave = 8      octatonic (Messiaen's 2nd mode)

stepsPerOctave = 7      diatonic natural scale

stepsPerOctave = 1      unison across octaves

Figure 44: Comparison of the same excerpt with different temperaments. In the second and third examples, divisions of the octave into equal parts produce intervals with a non-integer number of semitones. However, in this notation, rounding to semitones from the bach roll setup is employed. This allows, for example, the easy production of global harmonic grids such as the octatonic scale (alternating semitone and whole tone), the natural scale, and many more. Red notes indicate that they have been modified from the original first excerpt that uses standard tuning.

It is also possible to set fractional divisions of the octave. This is an uncommon practice but allows for the creation of global harmonic grids that may be of some interest. Figure 45 shows a couple of examples generated from the same previous fenotype.

stepsPerOctave = 0.91 augmented octaves

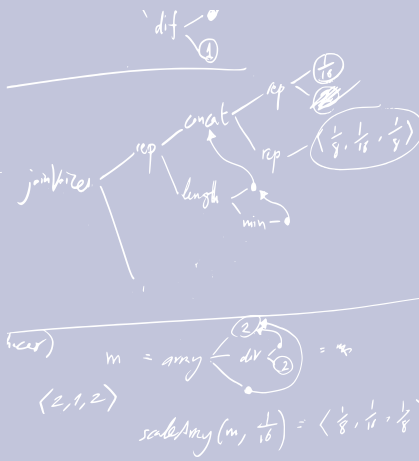
stepsPerOctave = 2.35 quartal harmony

Figure 45: Excerpts with non-integer values for stepsPerOctave. The first example applies a value close to 1, creating stretched octaves and a dissonant harmony typical of dodecaphonic writing. The second produces a concatenation of fourths. In these examples, all pitches are again rounded to the nearest standard tempered semitone.

### Evaluation and evolution

When modeling artistic creativity with computers, probably the most evasive issue to address is defining a fitness function. The assessment of an artistic product is by definition a very subjective one, and can be made only from a subjective point of view. The goal of art is to provoke an inner reaction.

The value of a piece of music  
We divide musical ideas into two categories: objective analysis of musical features and subjective valuation reduce to a very simple task. The analysis and self-analysis of



$z = \text{rep}(d, 4)$   
 $e = \text{truncate}(z, 1) = \langle \frac{1}{8}, \frac{1}{10}, \frac{1}{8} \rangle$   
 $h = \text{rep}(e, 1)$   
 $\text{concat}(d, \text{rep}(e, 1))$   
 $\text{diff}(f, 1)$

$d = \langle \frac{1}{8}, \frac{1}{10}, \frac{1}{8} \rangle$   
 $e = \langle \frac{1}{8}, \frac{1}{10}, \frac{1}{8} \rangle$   
 $h = \langle \frac{1}{8}, \frac{1}{10}, \frac{1}{8} \rangle$   
 $f = \langle \frac{1}{8}, \frac{1}{10}, \frac{1}{8} \rangle$

$m = \text{army} \left( \begin{matrix} \text{diff} \\ \text{diff} \end{matrix} \right) = \langle 2, 1, 2 \rangle$   
 $\text{scaling}(m, \frac{1}{10}) = \langle \frac{1}{8}, \frac{1}{10}, \frac{1}{8} \rangle$

$\text{concat}(d, \text{rep}(e, 1))$   
 $\text{diff}(f, 1)$

**7**

## Evaluation and evolution

“

But there's a big difference between "impossible" and "hard to imagine". The first is about *it*; the second is about *you*!

Marvin Minsky [102, p. 4]

When modeling artistic creativity with algorithms, probably the most evasive issue to address is programming fitness functions. A thorough analysis of how to evaluate and select products of automatic composition falls outside the scope of this research. However, some insights can be provided to lay the groundwork for further research in this area, based on preliminary experiments conducted during code testing.

By definition, the assessment of a piece of art can only make sense from a subjective point of view, since the goal of art is to provoke inner and personal reactions. These individual responses are very dependent on cultural, social, and individual contexts. Furthermore, the rating of musical ideas can be identified with the very act of composition, as long as composing music is ultimately making choices. However, provided with enough data, some predictions can be made regarding the expected reception for a new piece.

GenoMus is primarily conceived as a tool for discovering new music, both for users with no technical skills in music composition and for expert composers who can implement their functions and musical data to feed the system and create creative feedback. With this in mind, some guidelines for the evolutionary strategies of the model have been established:

- Multiple methods of evolution in parallel: Starting from a given genotype, a wide range of manipulations can be combined, mutating and crossing leaves and branches of the functional tree, and also introducing previously learned patterns at any time scale. The architecture of germinal vectors as universally computable inputs has been designed to enable high flexibility for any manipulation of preexisting material as simple numeric manipulations.
- Specimen autoanalysis: Some genotype functions can return an objective analysis of a set of musical characteristics, such as variability, rhythmic complexity, tonal stability, global dissonance index, level of inner autoreference, etc. These genotype metadata are very helpful for reducing the search space when looking for some specific styles, and allow any AI system to measure the relative distance and similarities to other specimens, classify results, and drive evolution processes.
- Human supervised evaluations: subjective ratings made by human users, attending to aesthetic value, originality, mood, and emotional intensity, can be stored and classified to build a database of interesting germinal conditions to be taken as starting points for new interactions with each user profile.
- Analysis of existing music: selected excerpts recreated as a decoded genotype (as the example shown in next Chapter 8), both manual or automated, can enrich the corpus of the learned specimens of a general database.

## 7.1. Specimen and variations

Next, we will examine some of the transformation procedures that will later be used in the creation of new generations of specimens. These methods can be activated manually from the interface or programmed as part of more complex evolutionary routines.

### 7.1.1. Music from pure randomness

With barely the integration of very simple genotype functions in a basic library, some appealing results have already emerged. Just to illustrate the expressiveness and stylistic variability of the outputs, even without any application of machine-learning techniques, at <https://genomus.dev/thesis/early-experiments> it provides a collection of sequences generated during early tests with GenoMus, rendered without any manipulation.<sup>61</sup>

---

<sup>61</sup>The author of this thesis gave a TEDx talk on musical composition and AI, where some of these fragments were played. The presentation is available at <https://vimeo.com/lopezmontes/tedx-music-ia>.

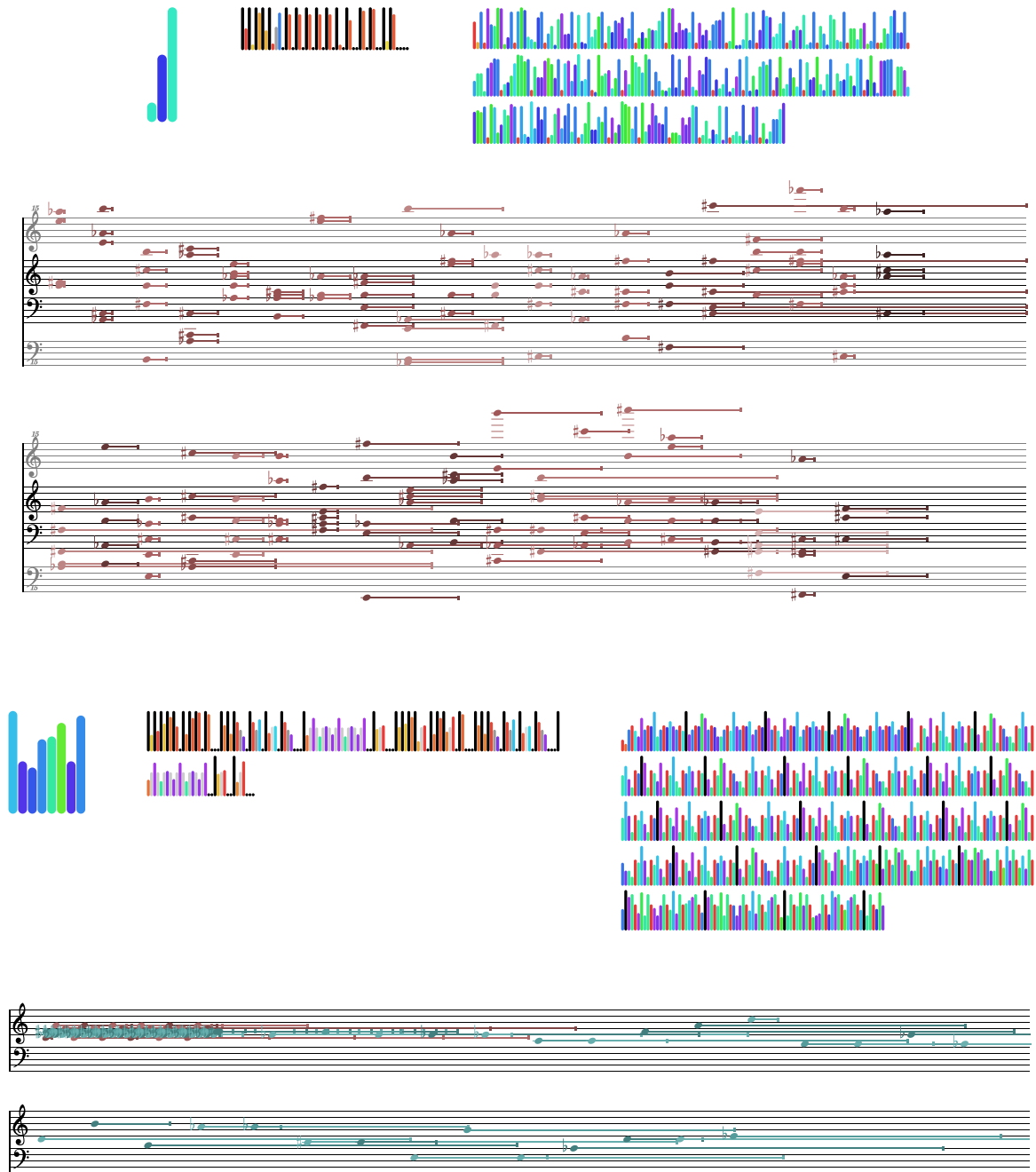


Figure 46: Two examples of specimens generated with a very short germinal vector. In each of the two examples, three vectors are visualized: the germinal vector (of 3 and 8 values respectively), the encoded genotype, and the encoded phenotype, which is transformed below into a score. Although the germinal vector is read in a loop, the requirements of the chosen functions can take complex paths. In the second example, it is possible to see how a loop has originated in the encoded genotype.

### 7.1.2. Starting with very short germinal vectors

The previous Figure 46 shows a pair of specimens generated with very short germinal vectors. The vectors have been generated manually. With just a few values, a complete program is written, and the generated music can be extensive, depending on the functions that have been called when decoding the germinal vector as a decision tree that writes the genotype. In Figure 47, the germinal vector generates a very long program.

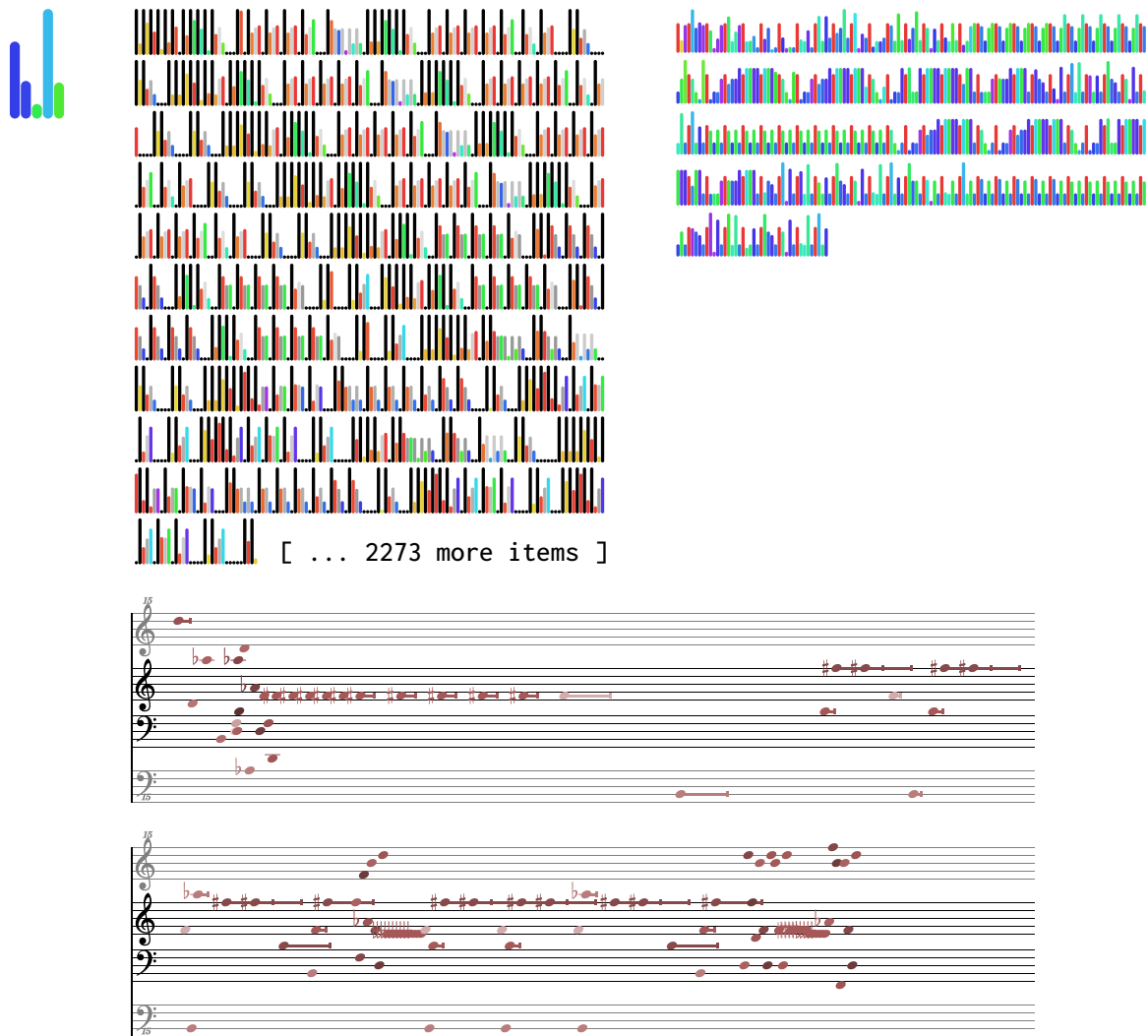


Figure 47: A specimen generated with a very short germinal vector. The germinal vector, with only 5 values, triggers a metaprogramming process that ends in a genotype with more than 2000 tokens. The phenotype is shorter in this case.

### 7.1.3. Leaves mutation

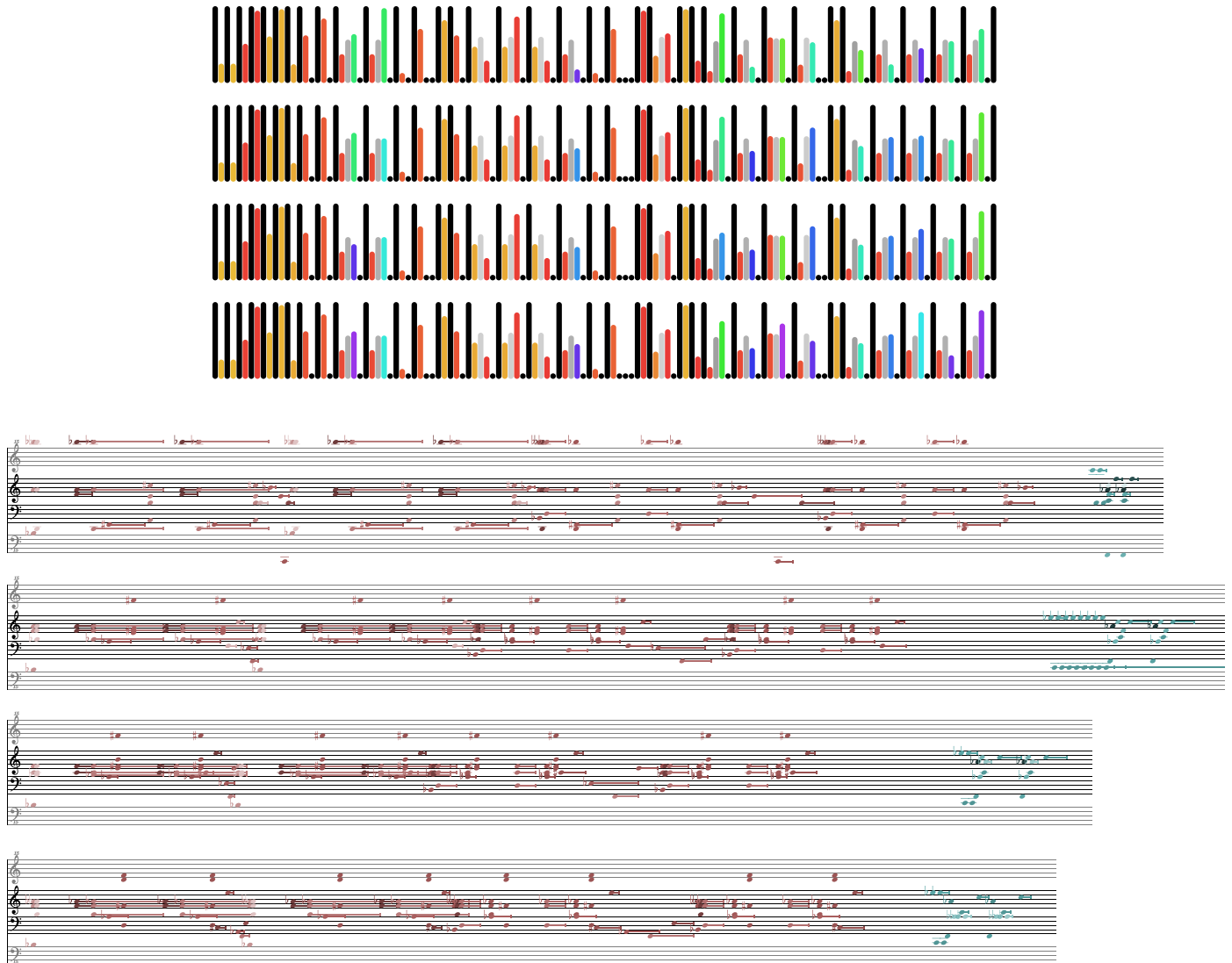


Figure 48: Four versions of a genotype with progressive leaves mutations are displayed. The beginnings of four encoded genotypes are shown in parallel. At the top is the original, and successive mutations appear towards the bottom. Note that the black and reddish bars (the function calls) do not change. In the four resulting musical fragments, the progressive transformation can be observed.

The mutation of leaves completely preserves the structure of the functional tree of the decoded genotype because only modifies the values of the leaves, which are the final numerical parameters. Listing 59 shows an example of mutation. On the left, is the original

genotype; on the right, is the mutated one. It can be seen that only numerical values have been modified, not the functions. It should also be noted that the numerical parameters of the autoreference functions, such as `pAutoref`, are never altered, as this would imply a change in the resulting internal functional structure.

```

sHarmonicGrid(
  s(
    vMotifLoop(
      lnWrap(
        l2P(
          p(0.880346),
          p(0.504815))),
      lm(54, 27, 56, 41, 58, 38, 57, 74, 56, 67),
      laWrap(
        lRecursioOrder4(
          rTanh(
            rRnd()),
          pRnd(),
          pAutoref(3),
          pAutoref(1),
          pAutoref(1),
          qRnd()),
          li(67.24, 49.9, 41.71, 54.77))),
      hOctatonicScale(
        m(26)))
    sHarmonicGrid(
      s(
        vMotifLoop(
          lnWrap(
            l2P(
              p(0.778022),
              p(0.504815))),
          lm(54, 46, 64, 53, 58, 16, 45, 76, 56, 67),
          laWrap(
            lRecursioOrder4(
              rTanh(
                rRnd()),
              pRnd(),
              pAutoref(3),
              pAutoref(1),
              pAutoref(1),
              qRnd()),
              li(60.27, 49.9, 34.36, 49.52))),
            hOctatonicScale(
              m(47)))

```

Listing 59: Comparison of decoded genotypes before and after a leaves mutation

To achieve this, we need the auxiliary function `extractLeaves`, which takes a genotype and returns an array like the one in Listing 60. This array gathers all the necessary information to identify the position and values of all the leaves in the functional expression.

```

"leaves" : [ [ 23, 0.608053, 69 ], [ 28, 0.972311, 106 ], [ 60, 0.113201, 38 ], [ 84, 0.89971, 1.5738 ],
  [ 89, 0.150718, 42 ], [ 94, 0.550278, 85 ], [ 99, 0.494173, 49.770000000000003 ], [ 107, 0.386315,
  0.17355 ] [ ... 97 more items ] ]

```

Listing 60: Array with positions and values of leaves returned by `extractLeaves`

From there, it is trivial to take a specimen and create a variation with those modifications. Listing 61 shows the implementation of `mutateSpecimenLeaves`, whose mission is to extract the leaves from a genotype (line 8), modify their values avoiding (lines 9 to 24), and create a new specimen with that structure (lines 25 to 49). The mutation is done considering two variables set by the user from the interface: `mutProbability` determines how likely it is for a leaf to be mutated, and `mutAmount` limits the largest possible deviation that can be applied in each mutation.



```
1 // mutates only leaves of a specimen according to certain probabilities
2 // mutProbability is mutations probability (0 -> no mutations, 1 -> everything mutated)
3 // mutAmount is range of a mutation, no trespassing interval [0, 1]
4 var mutateSpecimenLeaves = (originalSpecimen, mutProbability, mutAmount) => {
5   var startDate = new Date();
6   initSubgenotypes();
7   var mutatedSpecimen = { ...originalSpecimen };
8   var extractedLeaves = extractLeaves(mutatedSpecimen.encodedGenotype);
9   var numLeaves = extractedLeaves.length;
10  var mutationValue, tempLeafvalue;
11  for (var currentLeaf = 0; currentLeaf < numLeaves; currentLeaf++) {
12    if (Math.random() < mutProbability) {
13      mutationValue = mutAmount * (Math.random() * 2 - 1);
14      tempLeafvalue = mutatedSpecimen.encodedGenotype[extractedLeaves[currentLeaf][0]];
15      if (tempLeafvalue + mutationValue < 1 && tempLeafvalue + mutationValue > 0) {
16        mutatedSpecimen.encodedGenotype[extractedLeaves[currentLeaf][0]] =
17          r6d(tempLeafvalue + mutationValue);
18      }
19      else {
20        mutatedSpecimen.encodedGenotype[extractedLeaves[currentLeaf][0]] =
21          formatParam(tempLeafvalue - mutationValue);
22      }
23    }
24  }
25  reinitSeed(originalSpecimen.initialConditions.seed);
26  mutatedSpecimen = evalExpr(decodeGenotype(mutatedSpecimen.encodedGenotype));
27  mutatedSpecimen.data = {
28    specimenID: getSpecimenDateName(currentUser),
29    eventExtraParameters: originalSpecimen.initialConditions.eventExtraParameters,
30    specimenType: originalSpecimen.initialConditions.specimenType,
31    localEligibleFunctions: originalSpecimen.initialConditions.localEligibleFunctions,
32    depthThreshold: originalSpecimen.initialConditions.depthThreshold,
33    maxListCardinality: originalSpecimen.initialConditions.maxListCardinality,
34    seed: originalSpecimen.initialConditions.seed,
35    germinalVector: mutatedSpecimen.encGen,
36    renderTime: r3d((new Date() - startDate) * 0.001),
37    encGenotypeLength: mutatedSpecimen.encGen.length,
38    decGenotypeLength: mutatedSpecimen.decGen.length,
39    germinalVectorDeviation: 0,
40    depth: originalSpecimen.metadata.depth,
41    leaves: extractLeaves(mutatedSpecimen.encGen),
42    playbackRate: originalSpecimen.playbackOptions.playbackRate,
43    minQuantizedNotevalue: originalSpecimen.playbackOptions.minQuantizedNotevalue,
44    stepsPerOctave: originalSpecimen.playbackOptions.stepsPerOctave,
45    rating: 0,
46    iterations: 1,
47    millisecondsElapsed: 0
48  };
49  return specimenDataStructure(mutatedSpecimen);
50 };
```

Listing 61: `mutateSpecimenLeaves` creates variations of specimens by mutating leaves

#### 7.1.4. Germinal vector mutation

A similar mutation process can be carried out on the germinal vector. In this case, no distinction is made between the types of tokens, and any value in the sequence can be altered, also using the variables `mutProbability` and `mutAmount` to adjust the depth and extent of the mutation. Figure 49 compares side by side two germinal vectors with slight differences and the decoded genotype resulting from each of them.

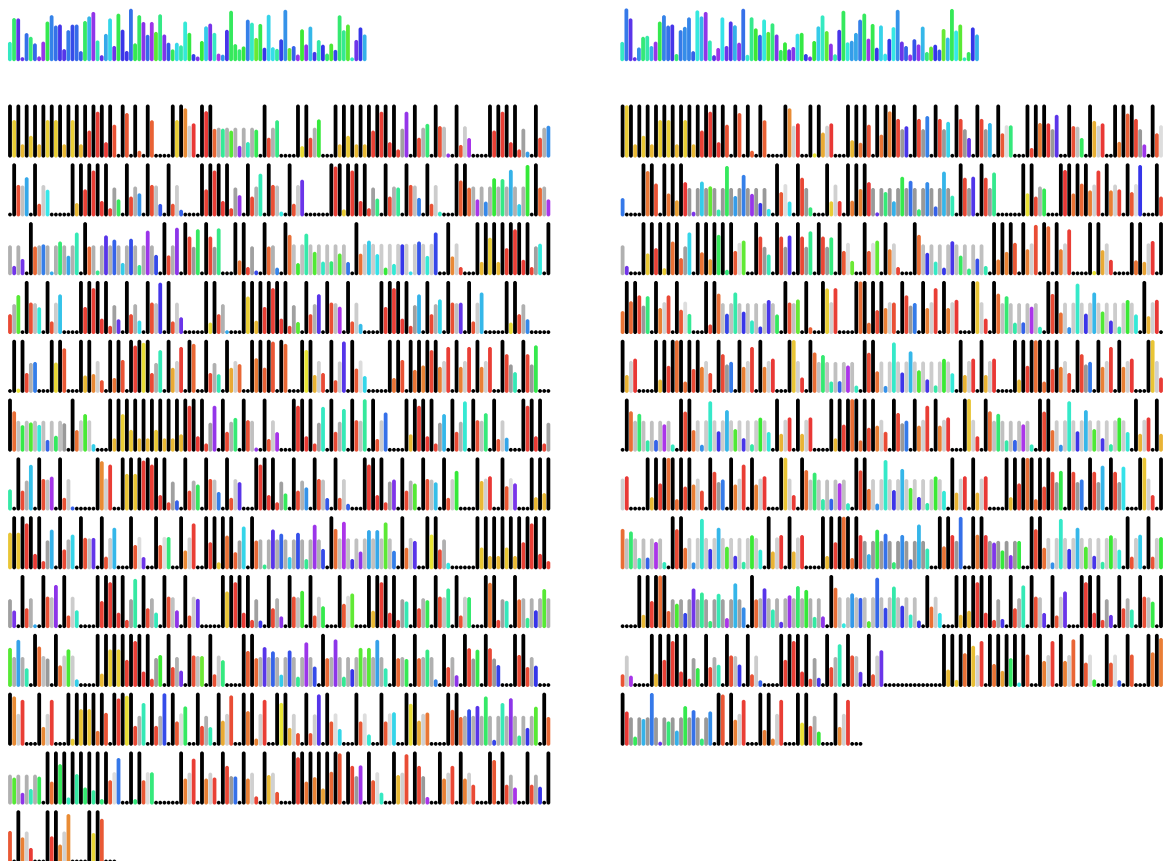


Figure 49: Comparison between a germinal vector and its mutation. This mutation affects the functional structure, as can be seen in the genotypes under their respective germinal vector. The length of both differs, and the functional structure, marked by the black bars, has visibly changed.

Comparing the impact of these changes on the code, in Figures 50 and 51, is clear how the mutations have consequences of different magnitudes on the overall framework.<sup>62</sup>

<sup>62</sup>Just like in biology, some mutations are indifferent and do not show changes in the phenotype, and others with drastic consequences, even by changing just one letter of the code.

```

sHarmonicGrid(
  sConcatS(
    sAddV(
      sConcatS(
        sHarmonicGrid(
          sHarmonicGrid(
            sConcatS(
              sHarmonicGrid(
                sConcatS(
                  s(
                    v(
                      e(
                        nRnd(),
                        mRnd(),
                        aRnd(),
                        iRnd()))),
                      sHarmonicGrid(
                        hJapaneseAutoref(1)),
                        mRnd()),
                    sV(
                      l(64, 62, 43, 49, 63),
                      vPerpetuum(78)),
                      hPentatonicScale(
                        i(43)),
                      sConcatS(
                        lReebosaoOrder4(
                          sAddV(
                            p(0.559776),
                            sConcatS(
                              q(65799),
                              sConcatS(
                                p(0.382089),
                                p(0.22802),
                                p(0.25673),
                                laRemag(2)),
                                n(1.0713),
                                laWafcap(
                                  m(69),
                                  l3PI(0.79121a(1),
                                    a(49)(0.418109(42.58))),
                                    aUp(0.856935),
                                    liWrap(0.64472))),
                                  li(503)33janid(
                                    hPentatonicScale(0.19139n(0.01772),
                                      m(75)),
                                      q(-9))),
                                      m(67),
                                      hJapanesePentatonicScale(
                                        l(135),
                                        m(49)),
                                        nRnd(),
                                        i(48.61))))),
                      sAddV(
                        lmsConcatS(
                          sHarmonicGrid(
                            lIts(
                              l(
                                0.19047, 0.629864, 0.533928, 0.06391, 0.945831, 0.33728, 0.765765, 0.380897, 0.197037, 0.075365),
                                q(-14))e(
                                  vAutoref(10917n(0.11269),
                                    sHarmonicGrid(
                                      m(58),
                                      sAutoref(
                                        a(28),
                                        hPentatonicScale(
                                          i(23.28))),
                                          m(47)),
                                          s(
                                            0.005334, 0.418109, 0.230681, 0.727062, 0.645859, 0.068854, 0.620039, 0.127021, 0.831874, 0.344264),
                                            hPentatonicScale(
                                              0.29138,
                                              mRnd()),
                                              p(0.81e(35))))),
                                  vMohiipopsePentatonicScale(
                                    0.15558,
                                    ln(0.7375),
                                    m(80),
                                    hPCmWfap(
                                      a(4),
                                      lmsWfap(
                                        i(56.87))))),
                                  lli(
                                    vABCAB(
                                      pAutoref(
                                        sHarmonicGrid(
                                          p(0.44264),
                                          pRnd()),
                                          e(
                                            m(38))Autoref(1)),
                                          n(0.12025),
                                          sAddV(
                                            la(266, 38, 37, 163, m(64),

```

Figure 50: Genotype with progressive germinal vector mutations. The expressions are directly superimposed in various colors to visualize the deviation from the original (in black). For all these mutations, mutProbability = 0.1 and mutAmount = 0.1.

```

s( liRemap( a(44),
vHarmonicGrid(7.49, 2i(13.56)))5, 46.53, 36.21, 36.8),
iRnd(7)Loop( h(lm(51, 48, 74, 73, 87, 44, 104),
i(37n(0))3375),lm(51, 38, 50),
vPerpetuumMobile(lWrap( lm(66, 1, 69, 54, 80),
nRnd(), lIterL( lm(18, 76, 71, 44, 72, 40, 83, 55),
lmWrap( lRecurm(89)ler4(
lJitter( r( p(0.856089),
l3P( p(0.874203))),
pAutoref(vPerpetuumMobile(
pAutoref(8),n(0.00748)),
pAutoref(3))m(38)4235),
p(0.344264), la(2791159),58, 31, 32, 53, 30),
pRnd()), li(55)04, 43.99, 26.07, 33.24, 52.03, 0.75, 54.34, 42.35, 63.09)),
la(238, 68, 78, 69wAutoref(5))),
li(57.69, 25sConcatS(toref(5))),
sConcatS( la(sHarmonicGrid(0, 47, 11, 39, 19),
sHarmonicGrid( liWrapSConcatS(
sAddV( lBrownian(
sConcatS( pGaussv(nRnd(),
sConcatS( pAutoref(e),
sAddV( qRnd(), n(0.30496),
sConcatS(toref(3))m(75),
hHexatonicScale( a(69),
mAutoref(60000)) i(59.68))))),
vPerpetuumMobile( s(
nAutoref(1), v(
lmWrap( e(
lLine( n(0.09131),
p(0.691556), m(46),
pAutoref(6), a(84),
q(-7))), i(39.9))))),
laRemap( hPentatonicScale(
la(49, 159, m(1)), 14, 100, 47),
a(103)sHarmonicGrid(
aRnd()), sConcatS( n(1.2626),
liAutoref(3)))s( m(113),
vMotif( v( a(135),
lnWrap( e( i(34.01))))),
l4P( sAddV( n(0.0675)),
pAutoref(9), s( m(76),

```

Figure 51: Genotype with progressive germinal vector mutations (continuation)

Finally, we can see the musical realization of these same mutations in Figure 52. These are just some examples of the many possibilities for automated manipulation of specimens.

## 7.2. Coevolutionary techniques

In the current version of GenoMus, a very simple genetic algorithm has been introduced to coordinate these types of manipulation in an overall workflow. The manipulations implemented so far are listed in Table 11.

Each new generation integrates specimens obtained through different methods, derived from the elite specimens selected from the previous generation by assigning them

a manual or autonomous rating. Once a group of selected specimens is available, a new generation can be constructed by combining such methods. These transformations have been ordered according to the degree of deviation from the selected specimens. The types of transformations that are more likely to deviate from the original specimen appear lower down in Table 11.

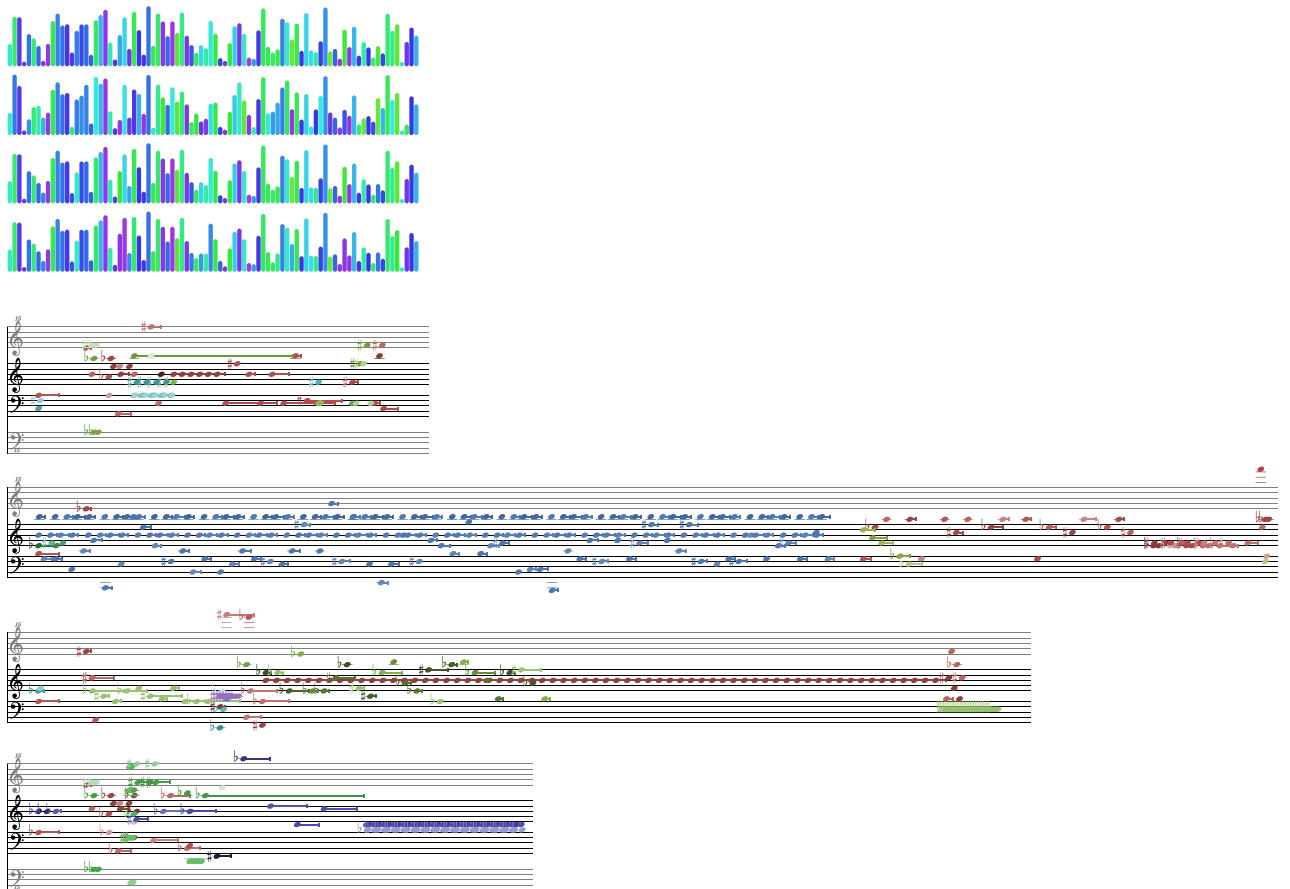


Figure 52: Scores generated by small mutations on an initial germinal vector. The four germinal vectors at the top correspond to the four scores according to their position. It can be seen that the small numerical changes have had a significant impact on the modification of the music.

Let  $A$  and  $B$  be two selected specimens from a given generation. To obtain a new generation, these transformations will be applied:

| Transformation type     | Description   |
|-------------------------|---|
| elite                   | Selected specimens from previous generation, preserved unchanged and chosen in decreasing order of rating.                |
| changedSeed             | New seed value in the initialConditions of <i>A</i> , which will only affect functions with intrinsic randomness.         |
| merged                  | Juxtaposition of the genotypes of <i>A</i> and <i>B</i> .   |
| crossedBranches         | A branch of the decoded genotype of <i>A</i> is selected and replaced by one from <i>B</i> .                              |
| replacedBranch          | A branch of the decoded genotype of <i>A</i> is selected and replaced by another one generated randomly.                  |
| grewedHorizontally      | New music is juxtaposed with <i>A</i> , wrapping its decoded genotype in an sConcatS function along with new random code. |
| grewedVertically        | New voices are added to <i>A</i> , wrapping its decoded genotype in an sAddS function along with new random code.         |
| mutatedGerminalVector   | Values in the germinal vector of <i>A</i> are modified with mutations of increasing probability and range.                |
| crossedGerminalVectors  | Part of the germinal vector of <i>A</i> is replaced by a fragment of the germinal vector of <i>B</i> .                    |
| replacedGerminalSegment | Part of the germinal vector of <i>A</i> is replaced randomly.   |
| brandNew                | Completely new specimens added to the pool of candidates.   |

Table 11: Transformations applied to obtain a new generation. If there is a conflict between the characteristics of specimens *A* and *B* involved in a cross, the properties of *A* are taken preferentially.

### 7.3. Sessions and global status

We have finally reached the largest data structure: the *session*. The global variable `statusGenoMus` stores the complete global state of the program, allowing it to be saved and reloaded at any moment. A session stores metadata and all descendant generations from the start of that session. Within each generation, there are two groups of specimens: candidates to be chosen for use in the next generation, and those already selected. Listing 62 displays the beginning of an actual session data.<sup>63</sup>

<sup>63</sup>The specimens are saved in the minimal version that we studied in Section 3.1, to prevent these files from becoming gigantic.

```
1 {
2   "sessionMetadata": {
3     "GenoMus_version": "1.00",
4     "user": "jlm",
5     "session": "sessions/GenoMus_status_jlm.json",
6     "totalGenerations": 5,
7     "currentGeneration": 1,
8     "currentGenerationCandidates": 256,
9     "currentGenerationSelected": 6,
10    "lastEvolutionTemperature": 84,
11    "lastSpecimensPerGeneration": 70,
12    "creationDate": "2024/01/10 22:54:19",
13    "lastUpdate": "2024/01/28 05:13:03",
14    "renderTime": 0
15  },
16  "generations": {
17    "1": {
18      "generationMetadata": {
19        "renderTime": 0,
20        "totalCandidates": 256,
21        "totalSelected": 6,
22        "selectedSpecimensRanking": [
23          0.888,
24          0.811,
25          0.786,
26          0.466,
27          0.254,
28          0.144,
29        ],
30        "selectedSpecimensID": [],
31        "originalCandidates": 371
32      },
33      "candidateSpecimens": {
34        "1": {
35          "metadata": {
36            "specimenID": "jlm-20240128_022506882-4",
37            "comments": {},
38            "GenoMusVersion": "0.10.03",
39            "rating": 0,
40            "duration": 51.027,
41            "voices": 1,
42            "events": 112,
43            "depth": 19,
44            "encGenotypeLength": 4112,
45            "decGenotypeLength": 10002,
46            "generativityIndex": 0.41,
47            "germinalVectorLength": 5,
48            "germinalVectorDeviation": 4109.341077729064,
49            "iterations": 1,
50            "millisecondsElapsed": 158,
51            "renderTime": null,
52            "history": {
53              "1": "Random new specimen - 6v 115e 67.294s",
54              "2": "Germinal vector edited, novelty 0.2 - 1v 112e 51.027s"
55            },
56            "storeIndex": 1
57          },
58          "initialConditions": {
59            "eventExtraParameters": 0,
60            "specimenType": "scoreF",
61            "localEligibleFunctions": [
62 [ ... 66349 more lines ]
63            "selectedSpecimens": {
64              "0.144": {
65                "metadata": {
66                  "specimenID": "jlm-20240110_225406653-27",
67 [ ... 79948 more lines ]
```

Listing 62: Example of a *GenoMus* session stored in `statusGenoMus`

## 7.4. Temperature and segmentation

```
1 var segmentation = (totalItems, temperature) => {
2   temperature = adjustRange(temperature, 0, 100);
3   var modificationTypes = Object.keys(modificationTypesAbundance);
4   var typeItems, adjustment = 0, itemsCounter = 0;
5   modificationTypes.map(tag => {
6     typeItems = Math.round(totalItems * 0.01 * rescale(
7       temperature, 0, 100,
8       modificationTypesAbundance[tag].low,
9       modificationTypesAbundance[tag].high));
10    modificationTypesAbundance[tag].items = typeItems;
11    itemsCounter += typeItems;
12  });
13  adjustment = totalItems - itemsCounter;
14  if (modificationTypesAbundance.eliteSpecimens.items + adjustment >= 0) {
15    modificationTypesAbundance.eliteSpecimens.items =
16      modificationTypesAbundance.eliteSpecimens.items + adjustment;
17  }
18  };
```

Listing 63: Function `segmentation` to control the composition of new generations

Like many analogous systems, *GenoMus* uses a *temperature* parameter to probabilistically control the range of deviation in transformations that will form the pool of candidates for the next generation. The temperature, within a range of 0 to 100, determines the percentage of specimens constructed according to different types of transformations. Applying this percentage to the total desired candidates determines the final content that will form the next generation. This calculation is carried out with the function `segmentation`. If the corresponding number of elite specimens for the new generation exceeds the specimens selected from the previous generation, these same specimens are included with mutated leaves with an increasing but very slight degree of mutation.

To distribute the proportion of methods used, the variable `modificationTypesAbundance` is employed, which establishes default boundaries for each type of transformation in case of minimum and maximum temperature.



```
1 var modificationTypesAbundance = {  
2   eliteSpecimens: {low: 40, high: 6},  
3   mutatedLeaves: {low: 18, high: 1},  
4   changedSeed: {low: 10, high: 2},  
5   merged: {low: 7, high: 3},  
6   crossedBranches: {low: 7, high: 4},  
7   replacedBranch: {low: 6, high: 5},  
8   grewedHorizontally: {low: 5, high: 6},  
9   grewedVertically: {low: 3, high: 7},  
10  mutatedGerminalVector: {low: 2, high: 10},  
11  crossedGerminalVectors: {low: 1, high: 15},  
12  replacedGerminalSegment: {low: 1, high: 20},  
13  brandNew: {low: 0, high: 20}  
14 }
```

*Listing 64: Temperature-based segmentation of transformations applied to evolution*

- procedimientos  
- motivos musicales

Los dos han de estar ligados (lo importante de un proced. es la parametrización)

GenTipo

param. a b c ... d

librería de vectores acústicos

primer hit que diseñe las funciones de evolución para componer sistemas de búsqueda

$a_2 = \text{not}(2,3,2,4,4) \Rightarrow \text{trans}(2, (0,1,0,2,2)) \Rightarrow$

Algoritmo de análisis:

- Se analiza set
- la escala de la
- el análisis de a
- después de la ventana mínima se analizan

8

## Procedural analysis

“

Grammars may lead beyond unified composing and analysis models and toward intelligent musical devices. An intelligent musical device will be able to convert the iconic musical signal into symbolic form and be able to recognize for example, not only frequency, amplitude, and duration [...] but also larger syntactic forms such as phrases and other macrostructures as well as extra-syntactical aspects of the music. Acting from a base of programmed or even acquired grammatical knowledge, such a device will be able to listen and respond intelligently not just to sound, but to music.

Curtis Roads [123, p. 54]

Music analysis is focused on capturing and compressing the information contained in real music. As López de Mántaras [82] states, “in general, the rules don’t make the music, it is the music that makes the rules”. In this section, this advice will be taken in a pretty literal way.

To illustrate how GenoMus represents and encodes a piece of music using basic procedures in an abstract format, let’s model *Clapping Music*, a famous piece composed by Steve Reich in 1972. Like many of his works, it begins with a minimal motif that undergoes simple transformations with big consequences in the overall form. Two performers start the piece by repeating a clapped pattern. One of the performers removes the first note of the pattern after each cycle of eight repetitions, creating a phase shift among both rhythmic lines. The work is finished when both players are in phase again. Figure 53 shows a compressed version of the score and one of the possible analytical deconstructions in patterns.

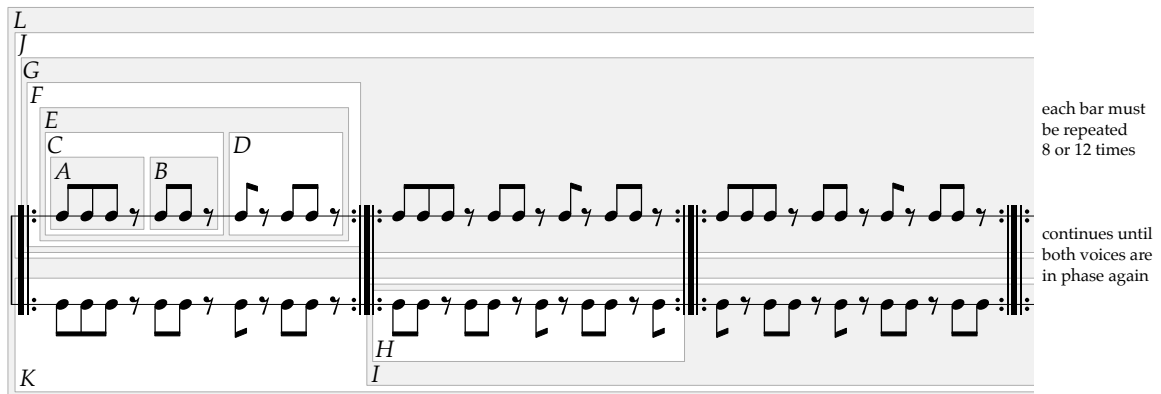


Figure 53: Compressed score of S. Reich's *Clapping Music*. Each box contains nested musical patterns derived from the initial motif A. Derived pattern L embraces the whole piece. This is only one of many possible procedural analyses of this famous work.

## 8.1. *Clapping Music* as a procedural genotype

There exist several ways to model the piece since its patterns can be obtained using different methods.<sup>64</sup> To create this genotype, I looked for the most concise code assembling simple and generic operations. I have recreated the whole composition from transformations of the first four notes (motif A), applying six generic operations using the genotype functions explained in Table 12.

The piece was modeled using the default simplest species which, according to the formalism stated in Section 2.4, is defined as the set  $G = \langle \text{Types}, \vec{t}, \text{Maps}, \text{Funcs}, \text{Coders} \rangle$ . The required elements for completing this definition are enumerated in Table 13.

Beyond the simple identity functions  $q$ ,  $ln$ ,  $lm$ ,  $la$ , and  $li$ , which simply take and return unchanged numeric values or lists that serve as leaf values of the functional tree, only a few functions are needed to model *Clapping Music*. The processes carried out by these functions are described in Table 12. The complete work is recreated by the decoded genotype displayed in Listing 65 (the uppercase letters in the comments refer to the patterns analyzed in Figure 53).

<sup>64</sup>Comparing different strategies of generating this work is an interesting question beyond this paper's scope. Each model can correspond to alternative manners of perceiving structural aspects. A different analysis but also a procedural model of *Clapping Music* can be found in [65, p. 113].

| Function name     | Arguments type   | Output type | Description  |
|-------------------|--|-------------|--|
| <b>vMotifLoop</b> | (lnotevalue,<br>lmidipitch,<br>larticulation,<br>lintensity) | voice       | Creates a sequence of events based on repeating lists. The number of events is determined by the longest list. Shorter lists are treated as loops. |
| <b>vConcatV</b>   | (voice, voice)   | voice       | Concatenates two voices sequentially.  |
| <b>vRepeatV</b>   | (voice, quantized)   | voice       | Repeats a voice a number of times.   |
| <b>vSlice</b>     | (voice, quantized)   | voice       | Removes a number of events at the beginning or the end of a voice.   |
| <b>vAutoref</b>   | (quantized)  | voice       | Returns a copy of a previous voice branch of the functional tree, referenced by an index.  |
| <b>s2V</b>        | (voice, voice)   | score       | Joins two voices simultaneously.   |

Table 12: Genotype functions used to model Clapping Music. The table shows only functions that are not mere identity functions that act as simple data containers for each parameter type.

| Types   | $\tilde{t}$  | Maps  | Funcs  | Coders                                   |
|---|--|---|--|--|
| {notevalue,<br>midipitch,<br>articulation,<br>intensity,<br>quantized,<br>lnotevalue <sup>1</sup> ,<br>lmidipitch,<br>larticulation,<br>lintensity,<br>event,<br>voice,<br>score} | (notevalue,<br>midipitch,<br>articulation,<br>intensity) | {p2n,<br>n2p,<br>p2m,<br>m2p,<br>p2a,<br>a2p,<br>p2i,<br>i2p,<br>p2g,<br>g2p,<br>q2p,<br>g2p} | {q,<br>ln,<br>lm,<br>la,<br>li,<br>vMotifLoop,<br>vConcatV,<br>vRepeatV,<br>vSlice,<br>vAutoref,<br>s2V} | {tran,<br>dec,<br>enc,<br>eval,<br>conv} |

Table 13: Minimal elements in default species required to model Clapping Music procedurally

```

1  s2V( // score L: joins the 2 voices vertically
2  vSlice( // voice J: slices last cycle due to phase shift
3  vRepeatV( // phase G: F 13 times
4  vRepeatV( // cycle F: E 8 times
5  vConcatV( // pattern E: C + D
6  vConcatV( // motif C: A + B
7  vMotifLoop( // core motif A: 3 8th-notes and a silence
8  ln(1/8), // note values
9  lm(65), // pitch (irrelevant for this piece)
10 la(50), // articulation
11 li(60,60,90,0)), // intensities (last note louder for clarity)
12 vSlice( // motif B: A with 1st note sliced
13 vAutoref(0),
14 q(1))),
15 vSlice( // motif D: C with first two notes sliced
16 vAutoref(3),
17 q(2))),
18 q(8)),
19 q(13)),
20 q(-12)),
21 vConcatV( // voice K: F + H
22 vAutoref(7),
23 vRepeatV( // phase I: H 12 times
24 vSlice( // cycle H: F without 1st note, for phase shift
25 vAutoref(10),
26 q(1)),
27 q(12))))

```

Listing 65: Decoded genotype of *Clapping Music* model

Another key feature of the GenoMus grammar is the ability to set internal references to branches inside a functional tree to reuse music materials throughout the development of a piece. The argument supplied to a `vAutoref` function refers to a list of subexpressions stored and updated after each genotype function is evaluated. An autoreference can refer only to the available subexpressions indexed at the time of its evaluation, which implies that during the generative metaprogramming process, a function will only reuse preexisting musical material. As discussed before, this reflects how human perception and memory work, establishing interrelations only with preceding elements.

To clarify the autoreferences inside a genotype, Table 14 lists all subexpressions stored at the end of the genotype evaluation. This network of internal pointers allows efficiency and data reduction, and more importantly, it reflects the deep structure of music.

| Index | Subgenotypes  |
|-------|---|
| 1     | "vMotifLoop(ln(1/8), lm(65), la(50), li(60, 60, 90, 0))"  |
| 2     | "vAutoref(0)"   |
| 3     | "vSlice(vAutoref(0), q(15))"  |
| 4     | "vConcatV(vMotifLoop(ln(1/8), lm(65), la(50), li(60, 60, 90, 0)), vSlice(vAutoref(0), q(1)))"   |
| 5     | "vAutoref(3)"   |
| 6     | "vSlice(vAutoref(3), q(2))"   |
| 7     | "vConcatV(vConcatV(vMotifLoop(ln(1/8), lm(65), la(50), li(60, 60, 90, 0)), vSlice(vAutoref(0), q(1))), vSlice(vAutoref(3), q(2)))"  |
| 8     | "vRepeatV(vConcatV(vConcatV(vMotifLoop(ln(1/8), lm(65), la(50), li(60, 60, 90, 0)), vSlice(vAutoref(0), q(1))), vSlice(vAutoref(3), q(2))), q(8))"                                  |
| 9     | "vRepeatV(vRepeatV(vConcatV(vConcatV(vMotifLoop(ln(1/8), lm(65), la(50), li(60, 60, 90, 0)), vSlice(vAutoref(0), q(1))), vSlice(vAutoref(3), q(2))), q(8)), q(13))"                 |
| 10    | "vSlice(vRepeatV(vRepeatV(vConcatV(vConcatV(vMotifLoop(ln(1/8), lm(65), la(50), li(60, 60, 90, 0)), vSlice(vAutoref(0), q(1))), vSlice(vAutoref(3), q(2))), q(8)), q(13)), q(-12))" |
| 11    | "vAutoref(7)"   |
| 12    | "vAutoref(10)"  |
| 13    | "vSlice(vAutoref(10), q(1))"  |
| 14    | "vRepeatV(vSlice(vAutoref(10), q(1)), q(12))"   |
| 15    | "vConcatV(vAutoref(7), vRepeatV(vSlice(vAutoref(10), q(1)), q(12)))"  |

Table 14: Subgenotypes of voice function type stored during the evaluation of Clapping Music genotype, available to be pointed by internal autoreferences with function vAutoref. Only voice type subexpressions are shown, although they exist for other categories too.

Figure 54 displays the functional tree of this decoded genotype, along with its internal autoreferences. The tree is represented in inverse order, from left to right, to reflect how substructures are feeding to subsequent functions that construct larger musical patterns, as well as the subexpressions indexing order.

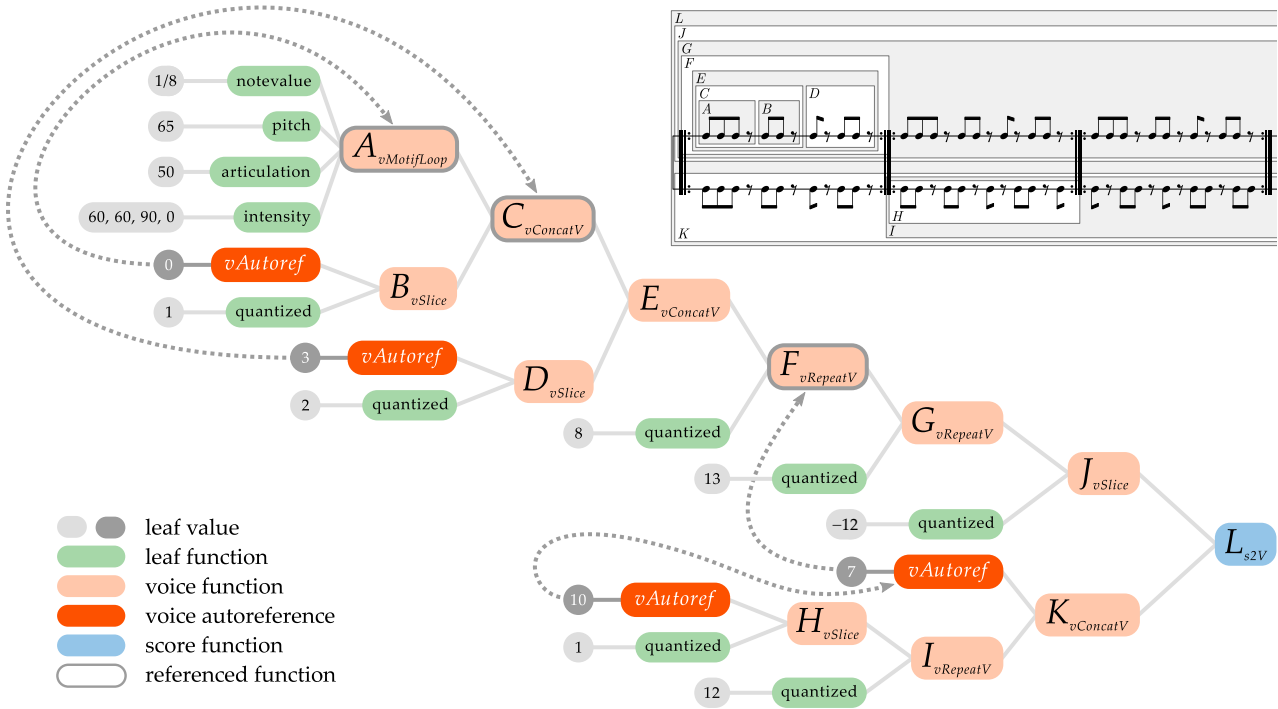


Figure 54: Functional tree of Clapping Music decoded genotype, along with the patterns in the score.

## 8.2. Converting a procedure into a new genotype function

The encoded genotype of this *Clapping Music* model consists of 117 values, which is remarkable for a piece lasting several minutes. This fact reflects how this work is a really good example of the minimalist principle of reducing the development of a composition to essential elements and to a process capable of exhibiting its capacity for autonomous growth. Figure 55 shows the visualization of the abstract pure numerical representation of the composition.

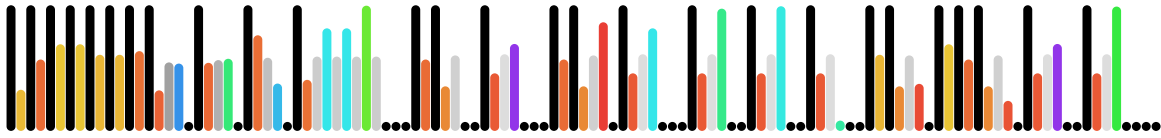


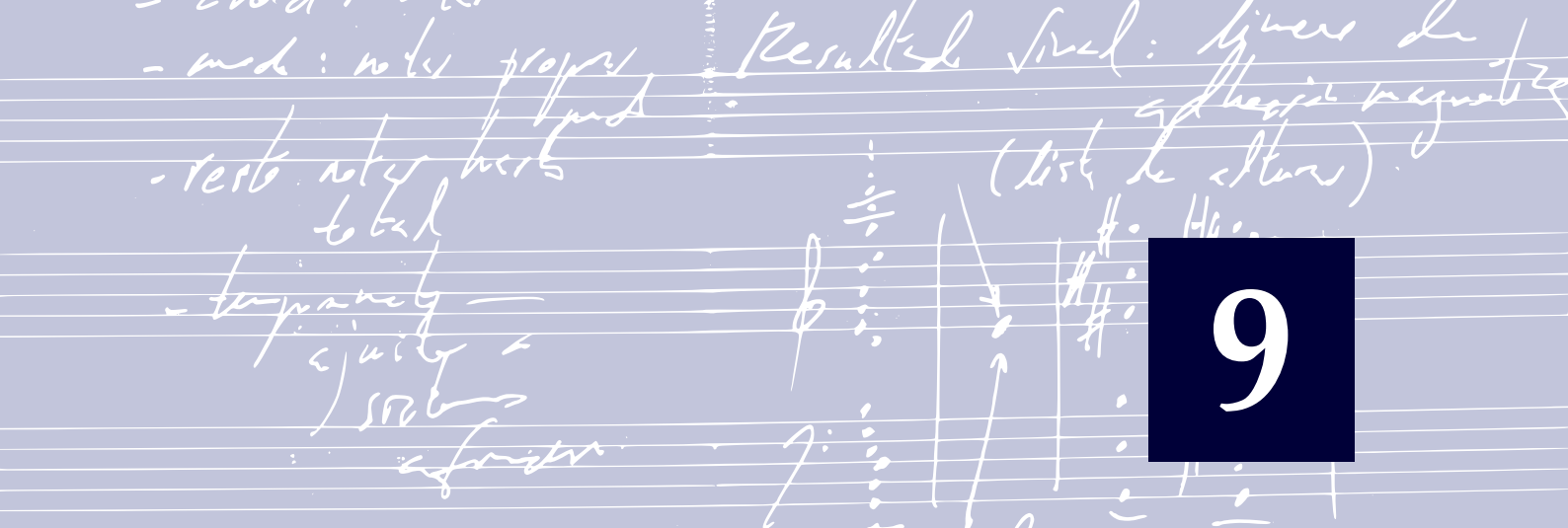
Figure 55: Visualization of Clapping Music encoded genotype. The visualization of this numeric sequence, using the color code detailed in Table 8, shows that most of the code consists of genotype functions (introduced and closed by black long and short bars), while there are only a few leaf values (numeric values given as final parameters, preceded by gray bars). As discussed previously, this vector is also one of its related germinal vectors.

The genotype of a whole composition like this (or only part of it) can be automatically flattened to create a new genotype function where all leaves are assembled as a single array of arguments, although the numeric values that feed arguments for the autoreferences are excluded from the arguments array of the new derivate function because of their structural character. These numbers must be immutable to preserve the internal consistency of the procedure. For instance, a new genotype function called **sClapping** could be created to be handled as a new procedure, abstracted from the original piece and compacted following the structure shown in Table 15.

| Function Name    | Arguments Type  | Output Type | Description   |
|------------------|---|-------------|---|
| <b>sClapping</b> | (lnotevalueF,<br>lmidipitchF,<br>larticulationF,<br>lintensityF,<br>quantizedF,<br>quantizedF,<br>quantizedF,<br>quantizedF,<br>quantizedF,<br>quantizedF,<br>quantizedF) | scoreF      | Creates two voices with a repeated pattern with a progressive phase shift of the second voice created by slicing notes at the beginning of the pattern. |

Table 15: Data structure of the genotype function **sClapping**, a new function generated after flattening the functional tree of Clapping Music shown in Listing 65.





## Results

“

I have come to the conclusion that much can be learned about music by devoting oneself to the mushroom. For this purpose I have recently moved to the country. Much of my time is spent poring over “field companions” of fungi. These I obtain at half price in second-hand bookshops, which latter are in some rare next door to shops selling dog-eared sheets of music, such an occurrence being greeted by me as irrefutable evidence that I am on the right track.

John Cage [29, p. 274]

The outcome of this research is twofold: *an open operational system that has already proven its utility as a tool for augmented creativity, and a substantial number of artistic works of various genres and characters*. The system is still quite limited and can be seen as a mere demonstrator, but now the path is clear to approach its expansion and enhancement, continuing this process of, in the words of Roads [121], *composing grammars*.

A combination of software development and artistic research<sup>65</sup> has been utilized. Thereby, the material results I present here are of two types: technical (Appendix A) and musical (Appendix B). Recapping the objectives described in the Introduction, O<sub>1</sub>, O<sub>2</sub> and O<sub>3</sub>, concerning the conception of a framework and its practical implementation, have been completed with the design and creation of GenoMus; O<sub>4</sub>, focused on artistic application, has been extensively carried out with a variety of projects of different kinds; finally, O<sub>5</sub>, related to the release of the tool, has been basically fulfilled and has much potential development ahead. These three aspects are analyzed in detail below.

<sup>65</sup>The consideration of *artistic research* as legitimate science *per se* [24, 36, 135] has been a topic of intense debate in academia for two decades. This thesis presents a mixed case aiming for synergy between research in art and computer science.

## 9.1. A procedural framework optimized for metaprogramming

The practical experiments with actual artistic applications have shown the strengths and weaknesses of each version, and have been fundamental in laying the foundations for the current prototype. Rather than introducing many different procedures, it has been established that the priority was to lay solid foundations for building a flexible and easily expandable platform. After these cycles of experimentation, creation, and development, I have achieved these desired features:

- *Grammar based on a symbolic and generative approach to music composition and analysis.* GenoMus is focused on the correspondence between compositional procedures and musical results. The analytical representation of music represented as trees has been also used by Ando et al. [8], representing classical pieces. It adopts the genotype/phenotype metaphor, as many other similar approaches, but in a very specific way.
- *Style-independent grammar, able to integrate and combine traditional and contemporary techniques.* In any approach to artificial creativity, a representation system is a precondition that restricts the search space and imposes aesthetic biases a priori, either consciously or unconsciously. The design of algorithms to generate music can be ultimately seen as an act of composition itself. With this in mind, this proposal seeks to be as open and generic as possible, to represent virtually any style, and to enclose any procedure. The purpose of the project is not to imitate styles, but to create results of certain originality, worthy of being qualified as *creative*. A smooth integration of modern and traditional techniques has been achieved. GenoMus allows for the inclusion and interaction of any compositional procedure, even those from generative techniques that imply iterative subprocesses, such as recursive formulas, automata, chaos, constraint-based and heuristic searches, L-systems, etc. This simple architecture integrates the three different paradigms described by Wooller et al. [157]: analytic, transformational, and generative. So, a genotype can be viewed as a multi-agent tree.
- *Optimized modularity for metaprogramming.* Each musical excerpt is generated by a function tree made with a palette of procedures attending all dimensions: events, motifs, rhythmic and harmonic structures, polyphony, global form, etc. All function categories share the same input/output data structures, which eases the implementation of the metaprogramming routines encompassing all time scales and polyphonic layers of a composition, from expressive details to the overall form. This follows the

advice of Jacob [71] and Herremans et al. [60], who suggest working with larger building blocks to capture longer music structures.

- *Support for internal autoreferences.* In almost any composition, some essential procedures require the reuse of previously heard patterns. As many pieces consist of transformations and derivations of motifs presented at the very beginning, this framework enables pointing to preceding patterns. At execution time, each subexpression is stored and indexed, being available to be referenced by the subsequent functions of the evaluation chain. Beyond the benefit of avoiding internal redundancy when there are repeated patterns, the possibility of creating internal autoreferences of nodes inside a function tree is an indispensable precondition for the inclusion of procedures that demand the recursion and reevaluation of subexpressions. This kind of reuse of genotype excerpts is also observed in genomics [144].
- *Consistency of the correspondences among procedures and musical outputs.* To obtain an increasing knowledge base, correspondences between expressions, encoded representations, and the resulting music remain always the same, regardless of the forthcoming evolution of the grammar and the progressive addition of new procedures by different users. To encode musical procedures, each function name is assigned a number, but to keep the encoded vectors as different as possible, function name indices are evenly scattered across a normalized interval and are registered in a library containing all available functions.
- *Possibility of generating music using subsets of the complete library of compositional procedures.* Before the automatic composition process begins, users can select which specific procedures should be included or excluded from it. It can also be used to set the mandatory functions to be used in all the results proposed by the algorithm.
- *Applicability to other creative disciplines beyond music.* Although this framework is presented for the automatic composition of music, the model is easily adaptable to other areas where creative solutions are sought. Whenever it is possible to decompose a result into nested procedures, a library of such compound procedures can be created, taking advantage of their encoding as numerical vectors that serve as input data for machine learning algorithms.

## 9.2. Artistic research shaping software

Perhaps the key point of the methodology has been a years-long alternation between periods dedicated to software development and others to testing it in the creation of musical projects intended for real public events. The role that each version of the software has played in these musical pieces has been diverse: in some cases, it was limited to obtaining materials that were combined with more traditional writing techniques, while in others, almost the entire piece was generated by the system, with the composer only selecting from the multiple available outcomes.

Since the first sketches presented in 2015 [84], each iteration of the project has been tested with the composition of real instrumental and electroacoustic pieces [83, 85, 86]. Following the completion of each artistic project, potentialities, weaknesses, or needs were identified to configure new versions. Throughout these iterations, it became evident which data and interrelation architecture between functions could be improved. On five occasions, a total code rewrite was carried out based on the knowledge extracted from practical applications. Retrospectively, it can be stated that the model would be much less flexible and expressive without these intensive testing periods.

Table 16 provides a very concise summary of the iteration stages in the model's development, associating each musical project with the main conclusions derived from it. The title of each project is a link to the section in Appendix B, which provides many more details about the model's application in its composition.

| Musical project  | Primary conclusions   |
|--|---|
| <p data-bbox="236 1402 735 1438"><i><b>Threnody for Dimitris Christoulas</b></i></p> <p data-bbox="236 1447 735 1509"><i>for flute (alt. piccolo), clarinet (alt. bass clarinet), violoncello, piano and tape — 14 min</i></p> <p data-bbox="236 1536 735 1760">Initial prototype test for exploring the generative possibilities of the functional paradigm. Application for the automatic composition of melodic patterns from recursive equations generated from a library of basic mathematical functions. Used exclusively for the pitch dimension.</p> | <ul style="list-style-type: none"> <li data-bbox="767 1447 1359 1541">▪ Interesting generative paradigm that is easy to program and produces results of arbitrary complexity that need to be constrained.</li> <li data-bbox="767 1559 1359 1684">▪ The musical results are diverse and inspiring, although since they are only one-dimensional patterns, their incorporation into real music requires a lot of manual arrangement.</li> <li data-bbox="767 1702 1359 1792">▪ Practical work reveals limitations that demand real-time visualization and auditory tools for the generator algorithm's outputs.</li> </ul> |

### *Ada + Babbage – Capricci*

*for violoncello and piano — 20 min*

Experimentation with a limited library of procedural functions typical of musical composition (creation of motifs, repetition, transposition, inversion, reversion, expansion, etc.). Implementation of a self-reference system as an essential feature.

- The use of motifs in function trees results in a highly effective source of musical variety with recognizable thematic unity.
- The importance of direct self-references to branches of the functional tree for their reuse and modification is demonstrated.
- The initial self-reference system necessitates inefficient transformations and recoding of the code. The numerical encryption solution for code fragments is intricate, but at the same time, it will open the door to ideas about encoding that will later be fundamental.

### *Microcontrapunctus*

*for 24 channels tape — 7 min*

Integrated use of multiparameters to generate an entire electroacoustic composition. Export as Csound scores for multichannel granular synthesis. Addition of new procedural functions more typical of sound synthesis.

- The tool exhibits great potential in the field of electroacoustics, especially for handling high density, both in terms of rhythm and the complexity of involved parameters.
- Need for a roll score representation for monitoring results quickly.
- Creative efficiency: once the generative algorithm was ready, the entire composition was composed in a few hours.

### *Seven Places*

*for violin, tape and optional video — 7 min*

Creation of scores for sound synthesis. First model for the generation of multiparameter events (with microtonal pitch, complex rhythm, and dynamics). First prototype using bach package for the visualization of generated scores.

- Need to find a method of using multiparameters in a flexible and unified manner without adding too much complexity to the functional tree.
- Conventional notation is inappropriate for working with the full range of possible rhythmic complexity without quantization.
- It is much more convenient to visualize the scores with bach.roll than with bach.score, for performance efficiency and simplicity in internal conversions.

### *Choral Riffs from Coral Reefs*

*for tape — 8 min*

Use of real-time MIDI output playback for highly dense rhythmic scores, applied to external physical modeling virtual instruments outside of Max. Experimentation with manual changes for mutation, seed change, and other initial conditions.

- Efficiency in the speed of execution. Once again, a piece created almost entirely algorithmically, with minimal tasks of assembly and final mixing.
- Need to optimize performance to achieve smooth real-time usage.
- Highly unexpected musical outcomes, resulting in a very different style from previous electroacoustic pieces.

### *Juno*

*for brass septet and optional tape — 4 min*

Experiment for generating harmonic progressions based on constraints for the desired chord characteristics (both internally and in transitions between them), as well as for the template of instruments involved and their respective registers.

- Development of ideas for the implementation of specific functions for harmonic grids.
- Potential of this constraint-based formalization: the composition is created very quickly, with many fragments that required minimal adaptation.

### *Openings for FACBA Podcasts*

*for tape — 3 min*

Direct creation of intro themes, without additional edition. Testing the integration of Node.js within the Max interface and a more refined system of sheet music representation. Addition of new procedural functions.

- The use of Node.js represents a significant leap in efficiency, computational speed, and overall interface fluidity. Being able to use a more up-to-date version of JavaScript allows for many improvements in the core code.
- The most effective way to store the generated scores and visualize them in Max is by using JSON format syntax embedded in dictionaries.
- The selection and rejection of ideas is much faster with better visual feedback from the score.
- Gradual mutation functions are effective for generating variations. With the new functions, a wider range of styles is achieved, ranging from the most abstract contemporary to much simpler and effective patterns for more commercial uses.

### Tiento

for binaural tape — 35 min

Experimentation with a new use case as a real-time controller for an independent audio sample manipulation patch. Creation of Csound scores for real-time synthesis within Max. Preliminary user experience, including collaboration with composer Pilar Miralles.

- The effectiveness and value of being able to work with many parameters simultaneously, now controlling an interface designed for human manipulation, are evident.
- The results are musically interesting due to the simultaneous control of many elements, which would be impossible to achieve manually.
- The algorithm's output functions as a *movement score*, enabling the repetition of sequences, their subtle modification, and the creation of real-time modification patterns that would otherwise be unattainable.
- Sound synthesis in Csound creates a highly varied sound palette, with often unexpected results in timbres and motivic sequences.
- The interface allows other users to use the application directly and intuitively, with minimal specific instruction.

### Rudepoema na penumbra

for quadraphonic tape — 23 min

Experimentation with the final system for real-time multiparameter control via OSC for sound synthesis with SuperCollider.

- Good rhythmic fluency and precision of the performance controlled by OSC. It is more convenient for electroacoustics because it is not as limited in the range of parameter values.
- It is interesting to create a final layer of parameter remappings by the user before sending the events to the final destination. This greatly increases productivity and the recombination of fragments and allows for the modulation of harmonic, rhythmic, textural, etc. to be more coherent.

---

Table 16: Iterative stages of prototype creation. The title of each project is a link to the section in Appendix B, which provides many more details about the model's application in its composition.

## 9.3. An open tool for augmented musical creativity

The website <https://genomus.dev> now serves as the primary resource for the dissemination, documentation, and development of GenoMus. This project is open-source and represents the most tangible result of this research cycle.

$$\text{perm}(M, e_1, e_2) = \langle m_i \rangle_{i=1}^{\min(e_1, e_2)} \parallel \langle m_{\max(e_1, e_2)} \rangle$$

etc

$$\langle m_1, m_{e_1}, \dots, m_{e_2}, m_n \rangle$$

$$a = \text{int}(\min(e_1, e_2)) / M_{\text{length}}$$

$$b = \text{int}(\max(e_1, e_2)) / M_{\text{length}}$$

$$\text{perm}(M, e_1, e_2) = \langle m_i \rangle$$

$$\begin{matrix} a-1 & & b-1 & & n \\ \parallel & m_i & \parallel & m_i & \parallel & m_i \\ i=0 & & i=a+1 & & i=a+1 & \end{matrix}$$

motif (chord)

escalas: n → natural  
m → melodic

m4 → meyo / meyo

m4, 7 → meyo / meyo desde sal

Le impo de necessiter de utiliser el tipo de notas, se desquivalen en otros

## Conclusions

“

Guerrero passed away in October 1997. Months before, Guerrero mentioned to me how he wanted his *opus ultimum* to be: not a composition, but a simple algorithmic formula. His desire was that anyone, using this algorithm, could compose a piece of music, each one different from the other yet connected by a common mathematical root. His intention was to leave behind an ultimate legacy, an idea so pure that anyone who wished could extract their own personal music from it without altering its essence. The algorithm was one of many topics that came up in that conversation. We never spoke of it again. However, as I reflect now, could Guerrero’s body of work be seen as a gradual attempt to approach that universal algorithm capable of encompassing everything?<sup>66</sup>

Stefano Russomanno [129, p. 107]

From the research and development cycle of tools documented in this thesis, the following conclusions can be drawn, which pertain to both the issues of knowledge engineering and those specific to artistic application:

### C1 • Importance of iterative development with alternation between technical prototyping and artistic application.

The process has shown that direct experimentation with real creative projects is essential to identify shortcomings and potentialities in the development of successive prototypes. Many initial ideas for implementing a feature have changed when it came to recreating musical passages or introducing certain procedures. Often, the modeling of such fragments has indicated paths for prototyping specific tasks.

<sup>66</sup> Author’s translation.



**C2 • Efficiency of encoding based on open and close indicators.**

Various ways of encoding processes and musical output as one-dimensional vectors have been experimented with. The proposed system has proven to be the most expressive and flexible. This primarily stems from how the metaprogramming sub-routines operate, manipulating one-dimensional numerical vectors through stacking, substitution, truncation, etc. The use of *flags* with extreme numerical values helps to easily discriminate structures and substructures.

**C3 • Suitability of an open structure of the *event* element.**

The variable architecture of the *event*, defined by the addition of extra parameters, enables system adaptation to various output format requirements, catering to specific needs. With the proposed encoding framework, the higher-level structures *voice* and *score* operate without changes.

**C4 • Retrotranscription as a bidirectional dual representation of metaprogramming processes.**

The mechanism known as retrotranscription has become one of the pillars of the model. Its necessity was not anticipated at the beginning of the research, but it ultimately is the most notable technical feature of the system. Retrotranscription allows a numerical vector to be simultaneously the abstract representation of a program and a decision tree that writes that same code with metaprogramming. The primary application of this bidirectionality is that, from code created or manually edited, a purely numerical counterpart can be directly introduced into any machine learning mechanism.

Having access to the dual numeric and textual representation at any point in the workflow has enabled a new way of approaching assisted composition. Pure generative processes can alternate with manual editing. This integration of mixed processes quickly leads to results that combine user control and the surprising capacity of metaprogramming.

**C5 • Interest of the *germinal vector* as a generating element.**

Although initially the germinal vector was conceived as the representation of a simple initial decision tree, the retrotranscription mechanism made it take on unexpected interest. Particularly in the case of very short vectors that require cyclical reading, highly complex music is generated from a minimal generating element.

Moreover, this implies a close link between the different sonic dimensions of musical sequences.<sup>67</sup>

**C6 • Influence of parameter mapping on generative processes of metaprogramming and evolution.**

The tasks of searching, selecting, evolving, and modeling outcomes from pure randomness in metaprogramming are heavily impacted by the design of the mapping for fundamental parameters like duration, pitch, articulation, and intensity. Since encoding is done with a limited range, a compromise between the size of the latent musical space and the probability of finding balanced results is necessary.

Expanding the search space increases the possibility of finding results with unbalanced characteristics. After experimenting with many possible conversions, the best model combines a very wide spectrum of values for each parameter with the use of a Gaussian distribution, so that extreme results are possible but unlikely.

**C7 • Various applications of the *golden encoded integers*.**

A remapping function as unusual as that of the *golden encoded integers* emerged from a very specific need: to optimize the uniform distribution of pointers within a numerical range without knowing their current or future quantity. However, this conversion has turned out to be useful for many more encoding purposes than initially expected.

**C8 • Multiplicity of workflows with the interface.**

The user interface integrates generative processes and evolutionary algorithms along with manual editing and selection. This diversity of manipulations has led to a wide variety of ways of working with the tool, sometimes unanticipated. In particular, the graphical manipulation of the germinal vector and its real-time impact on the metaprogramming of genotypes represents a new way of programming from a visual input, thereby swiftly exploring a multitude of possibilities. Sending sequences as files or in real-time with different formats can be applied to musical notation, sound synthesis, the use of virtual instruments, or any other type of software (not necessarily musical) that operates sequentially.

---

<sup>67</sup>This interrelation between melodic motifs, rhythmic patterns, and sequences of articulation and dynamics is characteristic of techniques such as integral serialism; subsequent procedures and styles, such as certain branches of spectralism, also establish these very unusual interrelations until then.

**C9 • Increase in stylistic flexibility and productivity of the tool with an optional final layer of manual processing.**

Having a palette of options for the final transformation of the musical result is very convenient for giving harmonic, rhythmic, and textural consistency to diverse materials. This facilitates the quick combination of fragments for subsequent composition processes.

**C10 • Evolutionary metaprogramming as reverse engineering for musical analysis and style imitation.**

The selection and transformation processes designed for automatic composition can be reversed to approach a phenotype serving as *fitness function*. Although this mode of work has been tested only with simple initial situations, it has been observed that evolutionary metaprogramming routines can perform tasks of analysis and generation of compositions with characteristics close to the given model.

**C11 • Feedback between open serendipity, the perception of underlying structures, and the evolution of musical listening.**

The relationship between the generation of materials and their listening mutually affects and co-evolves. Production from pure randomness exposes the ear to intricate internal structures. After many hours of listening to musical fragments produced during the tool's prototyping, it has been observed that the musical ear is capable of perceiving the existence of many layers of internal organization not evident in the score. This is the case even with very complicated genotypes, which are difficult to discern by reading the self-generated code.

\* \* \*

Returning to the hypothesis formulated at the beginning of this work, it can be concluded as a global conclusion that a meaningful procedural representation of music with a dual character is indeed possible, both as compressed data and in a form that is readable and manageable as a programming language. Likewise, it has been verified that the interrelation between both representation systems opens the field to new methods of artistic creation that enhance the synergy between current and future machine-learning techniques and augmented human creativity.

Let this research serve as one more stepping stone towards the democratization of new means of expression that expand the diversity of artistic sensibilities.



## Conclusiones

“

Guerrero falleció en octubre de 1997. [...] Meses antes, [...] Guerrero me comentó cómo quería que fuese su *opus ultimum*: no una composición, sino una simple fórmula algorítmica. Su deseo era que cualquiera, a partir de este algoritmo, pudiese componer una pieza musical: cada una diferente de la otra y al mismo tiempo ligadas entre ellas por una común raíz matemática. Su voluntad era dejar, como legado extremo, una idea tan pura que todos los que quisieran pudiesen extraer de ella una música personal sin modificar su esencia. El algoritmo fue uno de los tantos temas que se deslizaron en aquella otra conversación. [...] Nunca volvimos a hablar de ello. Sin embargo, ahora que lo pienso, ¿no será acaso la obra de Guerrero un intento gradual de acercarse a ese algoritmo universal capaz de abarcarlo todo?<sup>68</sup>

Stefano Russomanno [129, p. 107]

Del ciclo de investigación y desarrollo de herramientas que ha sido documentada en esta tesis pueden extraerse las siguientes conclusiones, que atañen tanto a las cuestiones de ingeniería del conocimiento como a las propias de la aplicación artística:

### **C1 • Importancia del desarrollo iterativo con alternancia entre prototipado técnico y aplicación artística.**

El proceso de trabajo ha mostrado que la experimentación directa con proyectos reales de creación es imprescindible para identificar carencias y potencialidades en el desarrollo de prototipos sucesivos. Muchas ideas iniciales para la implementación de alguna característica se han visto modificadas cuando se trataba de recrear pasajes musicales concretos o introducir determinados procedimientos compositivos.

En muchas ocasiones el modelado de estos fragmentos ha señalado el camino para el prototipado de funciones específicas.

**C2 • Eficiencia de la codificación basada en indicadores de apertura y cierre.**

Se han experimentado diversas maneras de codificar procesos y salida musical como vectores unidimensionales. El sistema propuesto ha resultado ser el más expresivo y flexible. Esto se deriva principalmente de cómo funcionan las subrutinas de metaprogramación, que manipulan los vectores numéricos unidimensionales mediante apilado, sustitución, truncamiento, etc. El uso de *flags* con valores numéricos extremos ayuda a discriminar fácilmente estructuras y subestructuras.

**C3 • Conveniencia de una estructura abierta del elemento *event*.**

La arquitectura variable del *event*, determinada por el número de parámetros extra, permite adecuar el sistema a cualquier requisito del formato de salida para adaptarse así a diferentes requerimientos concretos. Con el marco de codificación propuesto las estructuras de nivel superior *voice* y *score* operan sin cambios.

**C4 • Retrotranscripción como representación dual bidireccional de los procesos de metaprogramación.**

El mecanismo denominado retrotranscripción se ha convertido en uno de los pilares del modelo. Su necesidad no fue anticipada al comienzo de la investigación, pero finalmente es la característica técnica más destacada del sistema. La retrotranscripción posibilita que un vector numérico sea al mismo tiempo la representación abstracta de un programa y un árbol de decisión que escribe ese mismo código con metaprogramación. La principal aplicación de esta bidireccionalidad es que, a partir de código creado o editado manualmente, se puede introducir de manera directa una contrapartida puramente numérica del mismo en cualquier mecanismo de aprendizaje automático.

Contar en todo momento con la representación dual numérica y textual ha posibilitado una nueva manera de trabajar la composición asistida. En el flujo de trabajo se pueden alternar los procesos generativos puros con la edición manual. Esta integración de procesos mixtos lleva rápidamente a resultados que combinan control del usuario y capacidad de sorpresa por parte de la metaprogramación.

**C5 • Interés del *germinal vector* como elemento generador.**

Aunque inicialmente el vector germinal se concibió como la representación de un simple árbol de decisión inicial, el mecanismo de la retrotranscripción hizo que tomara un interés inesperado. Especialmente en el caso de vectores muy cortos que

requieren leerse cíclicamente, se genera música de gran complejidad a partir de un elemento generador mínimo. Además, esto implica una vinculación estrecha entre las diferentes dimensiones sonoras de las secuencias musicales.<sup>69</sup>

**C6 • Influencia del mapeo de parámetros en los procesos generativos de metaprogramación y evolución.**

Las tareas de búsqueda, selección, evolución y modelado de resultados a partir de la aleatoriedad pura en la metaprogramación se ven muy afectados por el diseño del mapeo de los parámetros fundamentales de duración, altura, articulación e intensidad. Dado que la codificación se hace con un rango limitado, es necesario encontrar un compromiso entre el tamaño del espacio musical latente y la probabilidad de encontrar resultados equilibrados.

La ampliación del espacio de búsqueda incrementa la posibilidad de encontrar resultados con características desequilibradas. Tras experimentar con muchas posibles conversiones, el mejor modelo conjuga un espectro muy amplio de valores para cada parámetro con el empleo de una distribución gaussiana, de modo que los resultados extremos sean posibles pero poco probables.

**C7 • Aplicaciones diversas de los *golden encoded integers*.**

Una función de remapeo tan poco habitual como la de los *golden encoded integers* surgió de una necesidad muy particular: optimizar la distribución uniforme de punteros en un rango numérico sin conocer su cantidad actual o futura. Sin embargo, esta conversión ha resultado útil para bastantes más propósitos de codificación de los previstos.

**C8 • Multiplicidad de flujos de trabajo con la interfaz.**

La interfaz de usuario integra procesos generativos y algoritmos evolutivos junto a la edición y selección manual. Esta diversidad de manipulaciones ha llevado a una amplia variedad de modos de trabajar con la herramienta, a veces no anticipados. En particular, la manipulación gráfica del vector germinal y su impacto en tiempo real en la metaprogramación de genotipos es una nueva manera de programar a partir de una entrada visual, y explorar así muchas posibilidades con agilidad. El envío de secuencias como archivos o en tiempo real con diferentes formatos puede aplicarse a la notación musical, a la síntesis de sonido, al empleo de instrumentos virtuales, o a cualquier otro tipo de software secuencial, no necesariamente musical.

<sup>69</sup>Esta interrelación entre motivos melódicos, patrones rítmicos y secuencias de articulación y dinámicas es característico de técnicas como el serialismo integral; procedimientos y estilos posteriores, como ciertas ramas del espectralismo, también establecen estas interrelaciones muy poco usuales hasta entonces.

**C9 • Incremento de la flexibilidad estilística y la productividad de la herramienta con una última capa opcional de procesado manual.**

Contar con una paleta de opciones de transformación final del resultado musical es muy conveniente para dar consistencia armónica, rítmica y textural a materiales diversos. Esto facilita la rápida combinación de fragmentos para procesos posteriores de composición.

**C10 • Metaprogramación evolutiva como ingeniería inversa para el análisis musical y la imitación de estilo.**

Los procesos de selección y transformación diseñados para la composición automática pueden invertirse para acercarse a un fenotipo en el rol de *fitness function*. Aunque este modo de trabajo se ha probado solo con situaciones sencillas de partida, se ha observado que se consiguen rutinas de metaprogramación evolutiva que funcionan como análisis y generación de composiciones con características próximas al modelo dado.

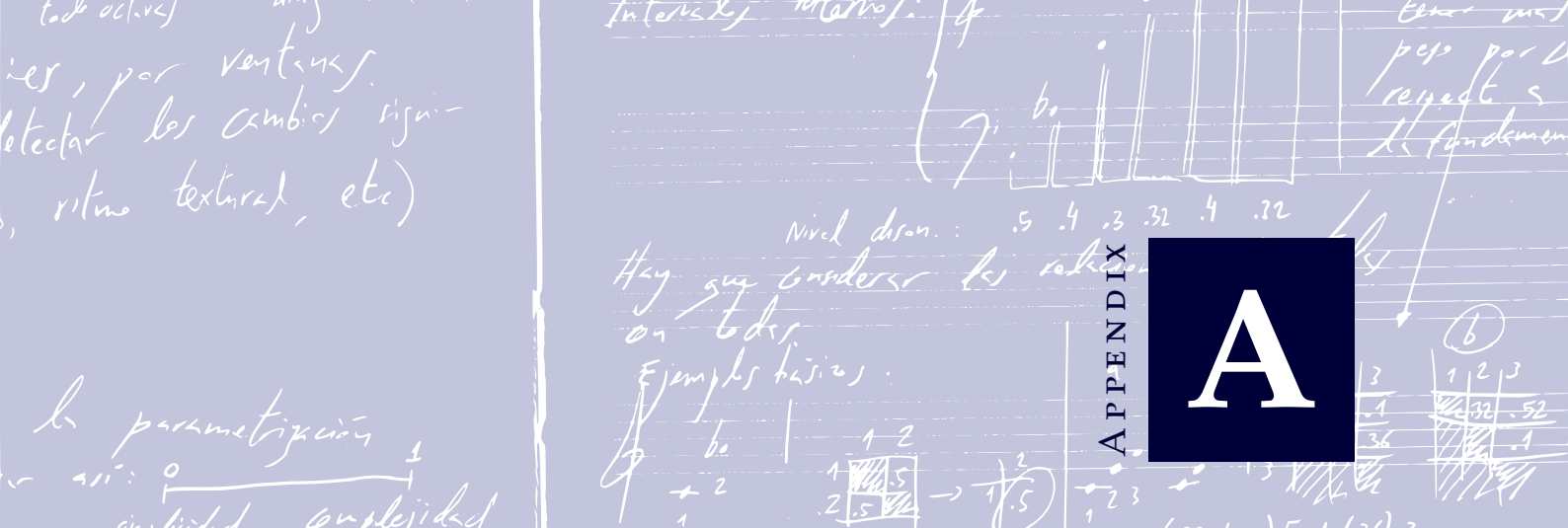
**C11 • Retroalimentación entre la serendipia abierta, la percepción de estructuras subyacentes y la evolución de la escucha musical.**

La relación entre la generación de materiales y su escucha se afectan mutuamente y coevolucionan. La producción a partir de la aleatoriedad pura expone al oído a estructuras internas intrincadas. Tras muchas horas de escucha de fragmentos musicales producidos durante el prototipado de la herramienta, se ha constatado que el oído musical es capaz de percibir la existencia de muchas capas de organización interna no evidentes en la partitura. Esto es así incluso en genotipos muy complicados, difíciles de discernir leyendo el código autogenerado.

\* \* \*

Volviendo a la hipótesis formulada al inicio de este trabajo, puede afirmarse como conclusión global que sí es posible una representación procedimental significativa de la música con un carácter dual, como datos comprimidos y en una forma legible y manejable como un lenguaje de programación. Asimismo, se ha comprobado que la interrelación entre ambos sistemas de representación abre el campo a nuevos métodos de creación artística que potencian la sinergia entre la creatividad humana aumentada y las técnicas actuales y futuras de aprendizaje automático.

Sirva esta investigación como una piedra más en el camino hacia la democratización de nuevos medios de expresión que expandan la diversidad de sensibilidades artísticas.



## GenoMus user interface

The main elements of the user interface of the current version at the time of writing this thesis are shown below. Some figures are accompanied by a table that explains details about the available actions. To operate this version of the core code, it is necessary to have Max 8 and the bach package installed. The most current version of the Max patch can be found at the URL <https://genomus.dev/download>.

### A.1. Main patch

The main patch, shown in Figure 56, is divided into several panels. On the right are all the subpatches from which to access windows dedicated to specialized functions, which will be discussed further below. The six panels cover these functionalities:

**Engine control:** Starts the core code for music generation. It allows to setting certain time limits in the search processes, to avoid bottlenecks that excessively slow down the interaction.

**Initial conditions:** Allow changing the generation conditions that were studied in Section 5.5. Changes are reflected in real-time.

**Manipulation:** Introduce conditions that affect the mutation of results and allow operating on the current specimen by adding new voices, replacing branches, etc.

**Playback:** Control the final transformation options studied in Section 6.4. Also responsible for playback, format, and MIDI output ports.

**Constraints:** Introduce restrictions in the random search for specimens.

**Load and save:** Load and save individual specimens or complete generations.



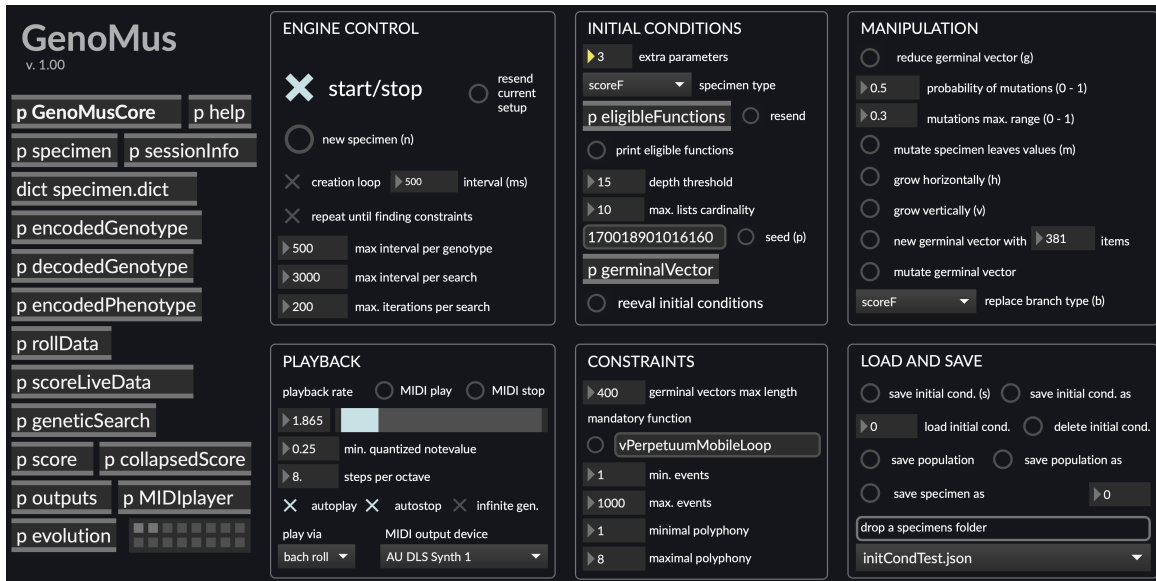


Figure 56: Main patch in presentation mode. On the left side are the subpatches containing modules for various operations and data monitoring. Double-clicking accesses the secondary windows.

All the patches shown in the subsequent figures are in *presentation mode*, which is the interface presented to the user. Max is a visual programming environment. To better understand the type of work that has been done, Figure 57 shows the same main patch in *edition mode*.

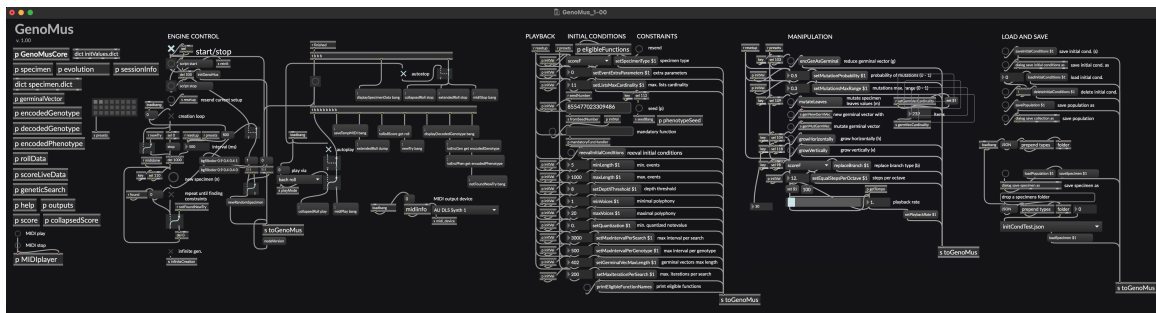


Figure 57: Main patch in edition mode. In editing mode, objects are arranged differently than in presentation mode, which allows selecting which elements will be visible and their position.

| Interface item                   | Description   |
|----------------------------------|---|
| <b>Engine control</b>            |   |
| start/stop                       | Starts the execution of the Node.js core code.  |
| new specimen                     | Creates a brand new specimen.   |
| resend current setup             | Resends the last options setup to the core code.  |
| creation loop                    | Creates brand new specimens cyclically according to a time interval.  |
| repeat until funding constraints | Stops the creation loop if a specimen that satisfies all the constraints is found.  |
| max. interval per genotype       | Some complex genotypes may take a long time to process. This value sets a limit to that time.   |
| max. interval per search         | Sets a limit to the genotype creation routine in the search for specimens that meet the given constraints.  |
| max. iterations per search       | Imposes a maximum on the genotypes generated in a search cycle.   |
| <b>Initial conditions</b>        |   |
| extra parameters                 | Determines the quantity of extra parameters in an event.  |
| specimen type                    | Changes the output type of the genotype's trunk function. This allows for easier exploration of certain specific processes.   |
| eligibleFunctions                | Subpatch for controlling the eligible functions (see Section A.3).  |
| resend                           | Resends the list of eligible functions to the core code.  |
| print eligible functions         | Prints the list of all eligible functions to the Max Console.   |
| depth threshold                  | Determines the depth level of the decoded genotype beyond which only identity functions are used, thus limiting their growth and avoiding infinite loops. In case of manual editing of genotype, this value is updated accordingly. |

|                           |  |
|---------------------------|--|
| max. list cardinality     | Maximum number of elements that can be part of a list-type function. In case of manual editing of a genotype that exceeds this threshold, this value is updated accordingly. |
| seed                      | Changes the seed that controls the global random processes to make them repeatable. The button produces a new random seed value.   |
| germinal vector           | Subpatch for visual control of the germinal vector (see Section A.3).  |
| reeval initial conditions | Resends all the initial conditions from the interface to the core code.  |

---

### Manipulation

|                               |  |
|-------------------------------|--|
| reduce germinal vector        | Creates a new specimen by retrotranscribing the current encoded genotype as a germinal vector. This allows checking the deviation between germinal vector and encoded genotype; if the retrotranscription is correct, the distance (displayed in the specimen's metadata) must be 0. |
| probability of mutations      | Controls how many of the leaves will be changed when making a mutation. For a value of 0.2, an average of 20% of the leaves will be mutated.   |
| mutations max. range          | Determines the maximum value change range in a mutation on the normalized encoded values. Thus, a value of 0.1 indicates that each mutated leaf can change its value by $\pm 0.1$ (without going out of the range $[0, 1]$ ).  |
| mutate specimen leaves values | Produces a new version of the current specimen by changing the leaf values according to the two previous parameters.   |
| grow horizontally             | Adds new music by juxtaposing it to the end of the current specimen, increasing its overall duration.  |
| grow vertically               | Adds new voices to the current specimen, increasing its polyphonic density.  |

|                        |   |
|------------------------|---|
| new germinal vector    | Generates a new germinal vector with a determined length.   |
| mutate germinal vector | Produces a new specimen by changing some values of the current germinal vector according to the specified probability and mutation range. |
| replace branch type    | Randomly changes a branch of the specified type in the functional tree of the genotype.   |

---

### Playback

|                          |  |
|--------------------------|--|
| playback rate            | Adjustment factor for the tempo of the current specimen.   |
| MIDI play/stop           | Manual playback control of the current phenotype.  |
| min. quantized notevalue | Minimum rhythmic value that determines the timegrid to which all events will be adjusted.  |
| steps per octave         | Tempered divisions of the octave. Non-integer values produce non-standard divisions along all octaves, which can be useful as specific harmonizations.   |
| autoplay                 | Plays the music as soon as it has been generated.  |
| autostop                 | Stops active events when a new sequence starts playing. This is useful when working in live coding mode, generating many fragments quickly, both deactivated, to be able to create dense textures, and activated, to avoid unwanted overlapping. |
| infinite generation      | Asks the system to generate a new specimen as soon as the current sequence has finished playing. This works as an <i>installation mode</i> operation, where there is music being generated continuously.   |

play via Allows two modes of playback: the preferred one is through the bach.roll viewer, which gives visual feedback and allows taking advantage of all its interactive editing options, and through a standard MIDI file that is saved and played immediately. This last possibility can be rhythmically more precise, especially with complex scores on not-very-powerful machines.

MIDI output device Chooses the output port for MIDI data. This allows to send the sequence in real-time to any other musical software, as well as to control external instruments, such as synthesizers, Disklavier pianos, etc.

---

### Constraints

germinal vectors max. length Limit to the number of values of the germinal vector. Interesting for working with short vectors that generate unpredictable internal interactions.

mandatory function Includes that function compulsorily in the functional tree of the genotype. Useful for debugging and for creating specimens that share the use of a specific procedure.

min. / max. events Minimum and maximum number of effective events that the generated music must contain.

min. / max. polyphony Minimum and maximum number of voices that the generated music must contain.

---

### Load and Save

save initial conditions as Saves a specimen in its reduced version with a name defined by the user.

load initial conditions Reads a specimen from the current population, according to its position number in the collection.

delete initial conditions Deletes the current specimen from the current population.

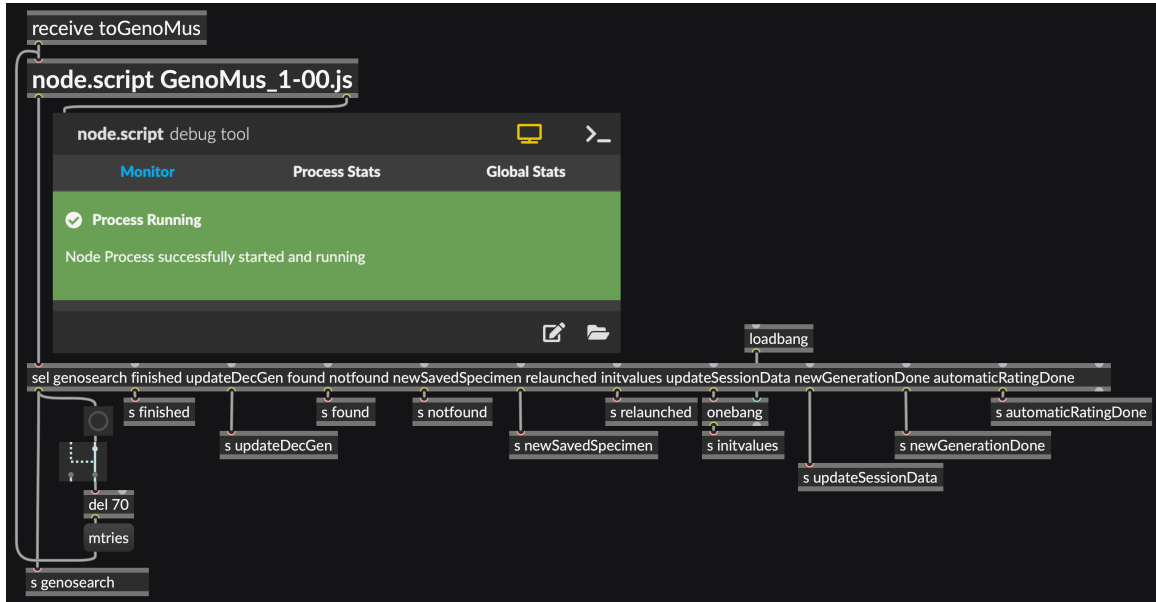
save population Saves the current population.

|                        |   |
|------------------------|---|
| save population as     | Saves the current population with a name defined by the user.                             |
| save specimen as       | Saves the current specimen in its fully rendered version with a name defined by the user. |
| drop a specimen folder | Allows dragging a folder to load all the specimens it contains as a population.           |

*Table 17: Description of the functionalities of the GenoMus main patch*

## A.2. Communication with core code

The subpatch in Figure 58 houses the core code of GenoMus inside a Node.js object. It routes some messages received from the algorithm that synchronize and control various actions of the UI. The code runs in the background without interfering with the real-time operability of the Max patches that are running at that moment.



*Figure 58: Communication subpatch with the GenoMus core code. A small viewer is included for JavaScript code debugging.*

### A.3. Control of initial conditions

Some initial conditions require separate control. The subpatch in Figure 59 allows for the selection of eligible functions for the automatic construction of genotypes.

Figure 60 shows the subpatch that displays a graphical representation of the germinal vector. This graph is interactive, and sliding the bars alters the vector, producing real-time changes in the generated specimen. It is thus easy to identify which values correspond to the evaluable expression of the decoded genotype, making this window an unusual tool for indirect programming.

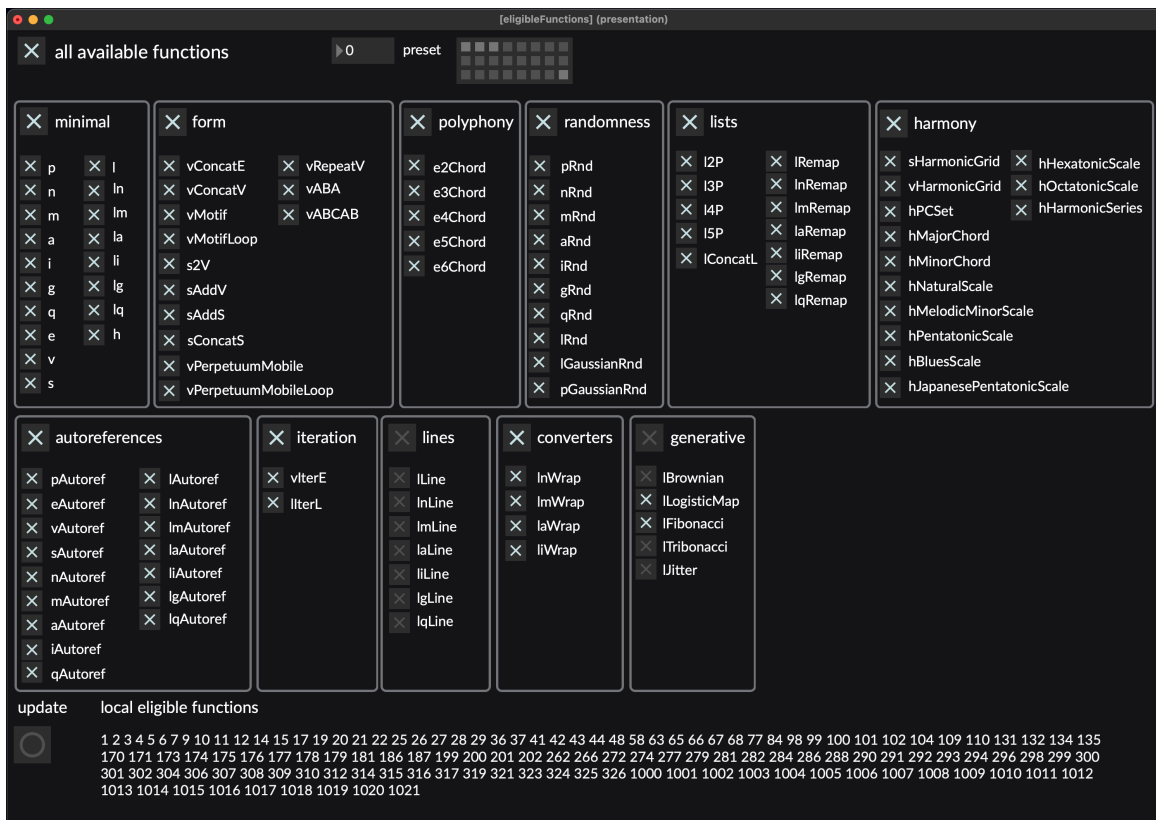


Figure 59: Control of the set of eligible functions for genotype construction. Functions can be selected by group or individually. Presets with specific configurations can be saved.

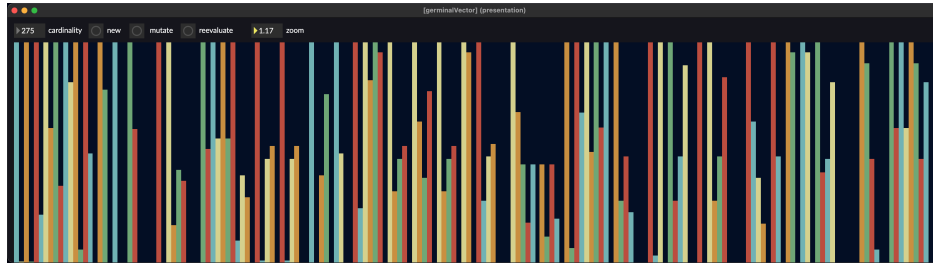


Figure 60: Graphical display of the germinal vector. This plot is editable, and it is interesting to see the impact that a small change has on the evaluable expression of the decoded genotype. Especially with short vectors, the consequences of small changes have macro-scale effects. From the same window, new vectors can be created, existing ones mutated, and the length of the vector determined, among other functionalities.

## A.4. Decoded genotype editor

A subpatch displays the functional expression of the decoded genotype. These programs can be enormously complex, with many levels of branching. The window allows for text editing and formats it to make it more readable. The functional trees can be displayed in a fully compact manner (Figure 61), in a semi-expanded form (Figure 62), keeping the lists on a single line, or completely expanded (Figure 63), with one line per element. It is also possible to expand or compact the closing parentheses. Expanding parentheses requires more space but allows for better identification of the blocks of the subspecimens, which is convenient for copying and pasting into new specimens.

Figure 61: Text editor of the decoded genotype functional expression with compressed formatting. This is how the generated genotypes are stored. The auxiliary function `formatDecGen` rewrites it in the various modes displayed by this viewer.



The screenshot shows a window titled "[decodedGenotype] (presentation)". The interface includes a toolbar with buttons for "semiexpanded" (a dropdown menu), "expand parentheses" (a button with a red 'X'), a page indicator "8", "fontsize" (a dropdown menu), "Font Name" (a dropdown menu), and "Menlo" (a dropdown menu). The main area is a text editor displaying a complex nested functional expression in a dark theme. The expression is formatted with semiexpanded parentheses, where each opening parenthesis is followed by a space and the closing parenthesis is indented. The code is as follows:

```
"sConcatS(
  sConcatS(
    sConcatS(
      sHarmonicGrid(
        s2V(
          vRepeatV(
            vABCAB(
              vMotif(
                ln(0.47882, 0.29681, 0.40898, 3.95885, 0.70653, 0.60552, 0.78023),
                lm(72, 94, 61, 23, 74, 58, 91),
                la(50),
                li(39.2, 49.57, 66.03, 56.28, 40.11, 58.27, 73.34)
              ),
              vPerpetuumMobile(
                n(0.28101),
                lm(67, 94, 63, 53, 79),
                la(4, 87, 5, 73),
                li(46.72)
              ),
              vAutoref(1)
            ),
            qRnd()
          ),
          vMotif(
            lnRemap(
              lnAutoref(1),
              nRnd(),
              nRnd()
            ),
            lmRemap(
              lmWrap(
                l(0.155117, 0.768208, 0.881024)
              ),
              mRnd(),
              mAutoref(1)
            ),
            laWrap(
              lIterL(
                l(0.992643, 0.483754, 0.8414),
                q(3),
                p(0.869124)
              )
            ),
            li(44.82, 49.4, 40.45)
          ),
          hPCSet(
            lm(46, 94, 92, 81, 58, 66),
            mRnd()
          )
        ),
        sHarmonicGrid(
          sConcatS(
            sAutoref(1),
            s(
              vConcatV(
                vPerpetuumMobileLoop(
                  n(0.19934),
                  lm(52, 110, 93, 59, 75, 0, 45),
                  la(59),
                  li(29.32, 37.68, 46.54, 29.76)
                ),
                vConcatV(
                  v(
                    e(
                      n(0.12631),
                      m(53),
                      a(18),
                      i(58.05)
                    )
                  ),
                  v(
                    e(
                      n(0.26708),
                      m(60),
                      a(65),
                      i(46.47)
                    )
                  )
                )
              )
            )
          ),
          hHexatonicScale(
            mAutoref(3)
          )
        )
      ),
      sHarmonicGrid(
        s2V(
          vRepeatV(
            vABCAB(
              vPerpetuumMobileLoop(
                nAutoref(3),
                lmRemap(
```

Figure 62: Text editor of the decoded genotype functional expression with semiexpanded formatting



```
[decodedGenotype] (presentation)
expanded expand parentheses 8 fontsize Font Name Menlo
"sConcatS(
  sConcatS(
    sConcatS(
      sHarmonicGrid(
        s2V(
          vRepeatV(
            vABCAB(
              vMotif(
                ln(
                  0.47882,
                  0.29681,
                  0.40898,
                  3.95885,
                  0.70653,
                  0.60552,
                  0.78023),
                lm(
                  72,
                  94,
                  61,
                  23,
                  74,
                  58,
                  91),
                la(
                  50),
                li(
                  39, 2,
                  49, 57,
                  66, 03,
                  56, 28,
                  40, 11,
                  58, 27,
                  73, 34)),
              vPerpetuumMobile(
                n(0.28101),
                lm(
                  67,
                  94,
                  63,
                  53,
                  79),
                la(
                  4,
                  87,
                  5,
                  73),
                li(
                  46, 72)),
                vAutoref(1)),
              qRnd()),
              vMotif(
                lnRemap(
                  lnAutoref(1),
                  nRnd(),
                  nRnd()),
                lmRemap(
                  lmWrap(
                    l(
                      0.155117,
                      0.768208,
                      0.881024)),
                    mRnd(),
                    mAutoref(1)),
                  laWrap(
                    lIterL(
                      l(
                        0.992643,
                        0.483754,
                        0.8414),
                        q(3),
                        p(0.869124))),
                    li(
                      44.82,
                      49.4,
                      40.45))),
                hPCSet(
                  lm(
                    46,
                    94,
                    92,
                    81,
                    58,
                    66),
                  mRnd()),
                sHarmonicGrid(
                  sConcatS(
                    sAutoref(1),
                    s(
                      vConcatV(
                        vPerpetuumMobileLoop(
                          n(0.19934),
                          lm(
                            52,
                            110,
```

Figure 63: Text editor of the decoded genotype functional expression with expanded formatting

## A.5. Specimen monitoring

Several auxiliary windows monitor the encoded data of the current specimen. Figure 71 shows the encoded genotype and encoded phenotype alongside the conversion of the phenotype to the format of the `bach.roll` object. The raw data of the rendered specimen can be accessed in two ways: through a text viewer (Figure 65), which is more convenient if an element needs to be copied, and a Max-specific viewer for dictionaries (Figure 66), which is more comfortable for navigating through the different data blocks.

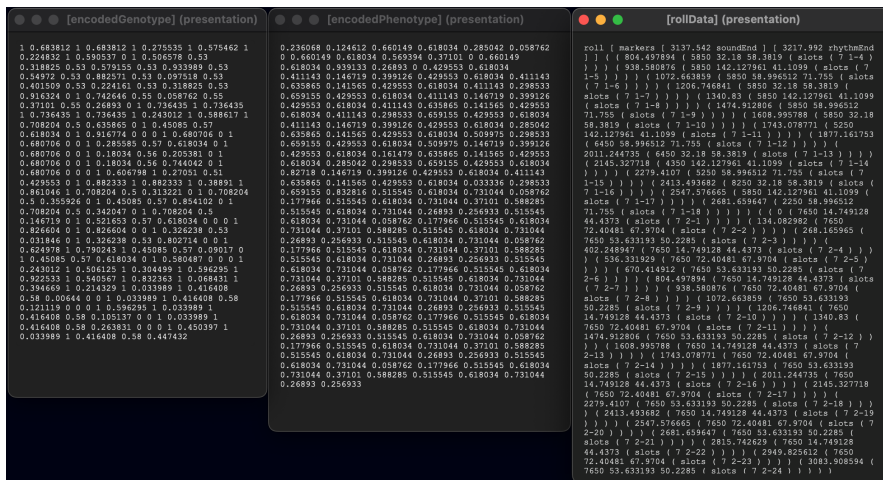


Figure 64: Viewers of encoded data and `bach.roll` converted data

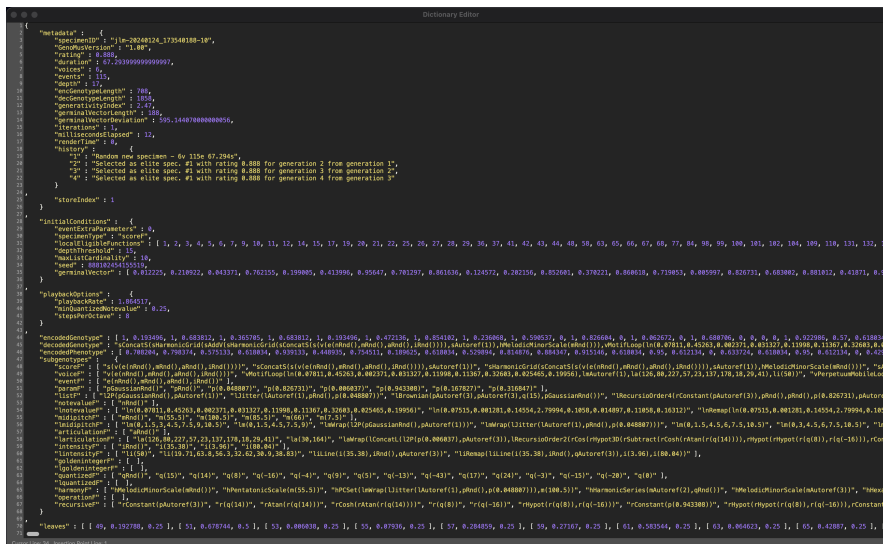


Figure 65: Viewer for the complete data of the specimen as formatted text

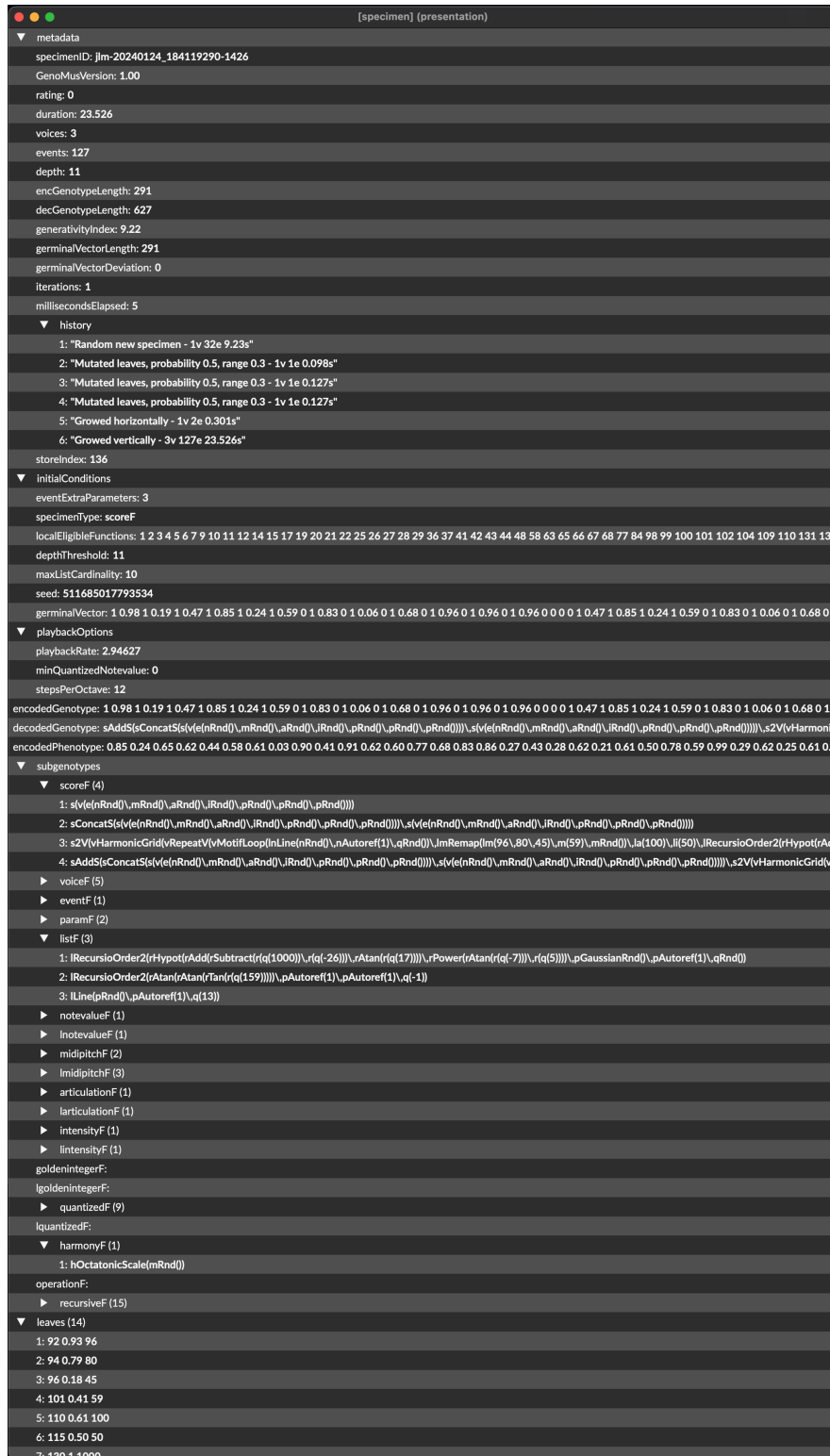


Figure 66: Specimen viewer. This display can collapse and expand data blocks.

## A.6. Score viewers

The score viewer is another fundamental piece of the interface. The visual response of the score is much faster than the auditory one, and in the selection process, it helps to quickly discard results. There are two ways to access the generated scores. The subpatch collapsedScore, in Figures 67 and 68, unifies all voices into a single system. The score subpatch, in Figure 69, displays each voice separately and adds some additional functionalities, detailed in Table 18.

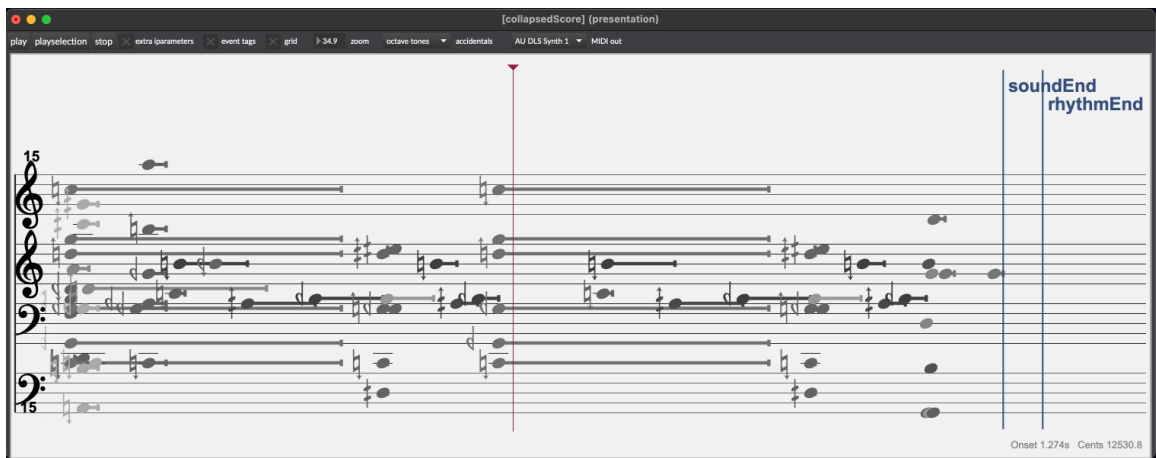


Figure 67: Collapsed score viewer. This example demonstrates the display and playback capabilities for microtonal scores. The score also shows where the end of the sound events is located, marked with soundEnd, and where the internal rhythmic structure ends, at rhythmEnd.

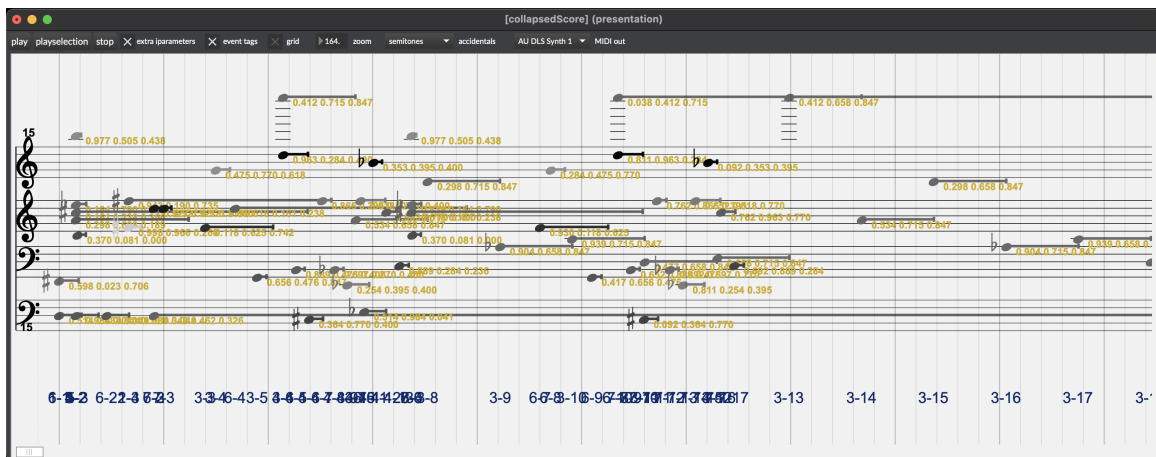


Figure 68: Collapsed score viewer showing event extra parameters, event tags and a timegrid.

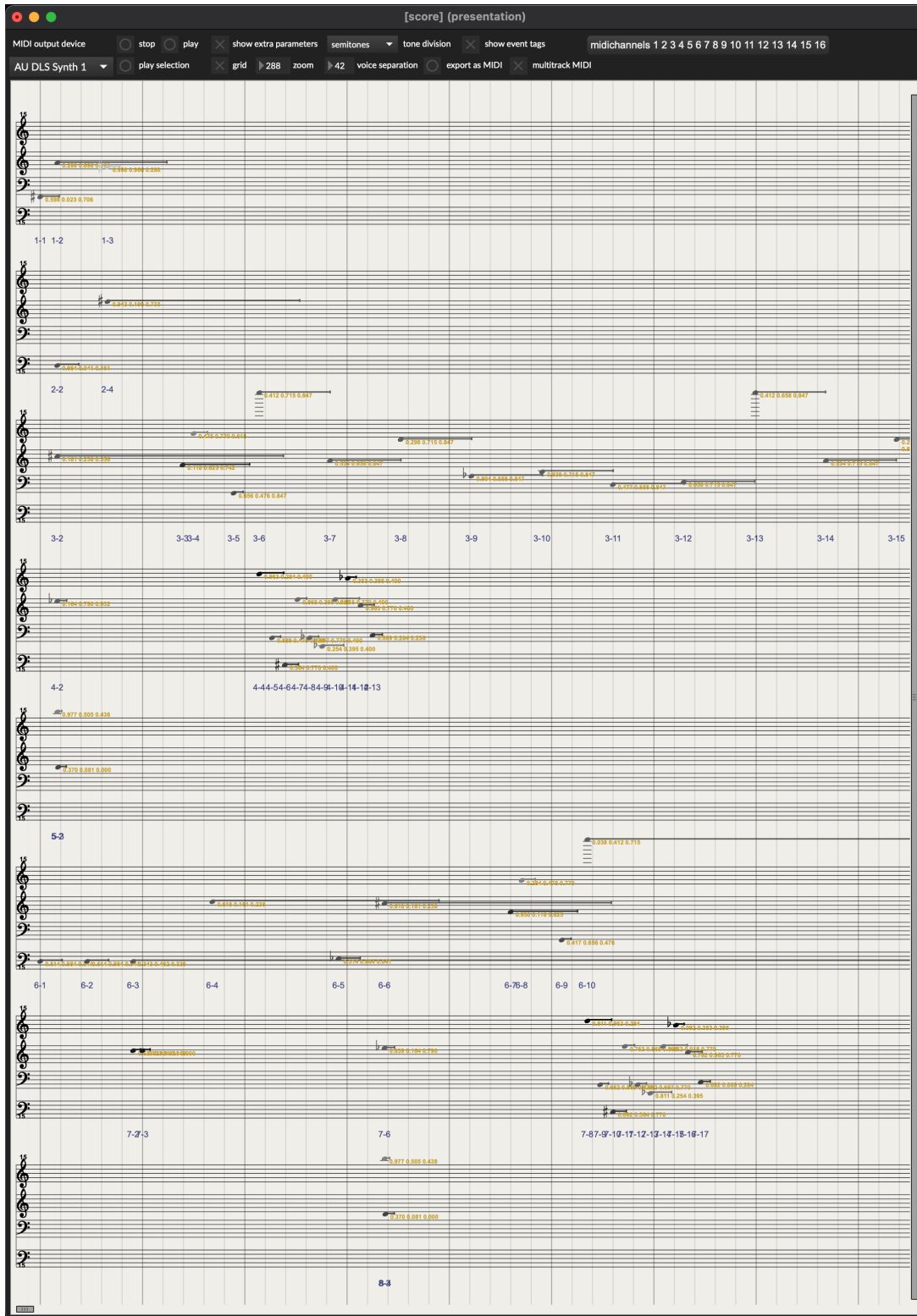


Figure 69: Main score viewer, displaying each voice separately. Functionalities explained in Table 18.

| Interface item               | Description   |
|------------------------------|---|
| MIDI output device           | Chooses the output port for MIDI data. This allows to send the sequence in real time to any other musical software, as well as to control external instruments, such as synthesizers, Disklavier pianos, etc.   |
| stop / play / play selection | Activates and deactivates the playback of the entire score or a part of it.   |
| show extra parameters        | Displays the numerical values of the extra parameters of the events, if they exist. These parameters are editable from the score, by clicking on the note and pressing 4.   |
| grid                         | Shows or hides the timegrid.  |
| zoom                         | Horizontal zoom. The score is displayed in a single line, without system breaks, so the zoom and the scrollbar are what allow navigating through the musical text.  |
| tone division                | Allows choosing among three types of pitch notation: semitone, quarter-tone, and eighth-tone resolution.  |
| voice separation             | Sets a vertical separation in the graphical display of the voices.  |
| show event tags              | Shows the tag of each event, which corresponds to the one listed in the decoded phenotype data of the specimen. Events that do not produce sound and those that exactly replicate the parameters of another have been eliminated in the phenotype itself, but these discarded events retain their label, to allow for tracking. |
| export as MIDI               | Generates a MIDI file with the content of the score.  |
| multitrack MIDI              | Allows choosing between exporting to a multi-channel MIDI file, or with everything collapsed into a single channel.   |
| <message> midichannels       | Assigns channels for each voice in the score, which applies both to real-time playback and to MIDI file export, enabling the use of many synths or virtual instruments simultaneously.  |

Table 18: Description of the functionalities of the score viewer. In addition to the functionalities described, the *bach.roll* object integrates countless options for graphical manipulation and editing of the events. This possible manual editing of these events is not retroactive to the corresponding specimen.

## A.7. Selection and evolution of specimens

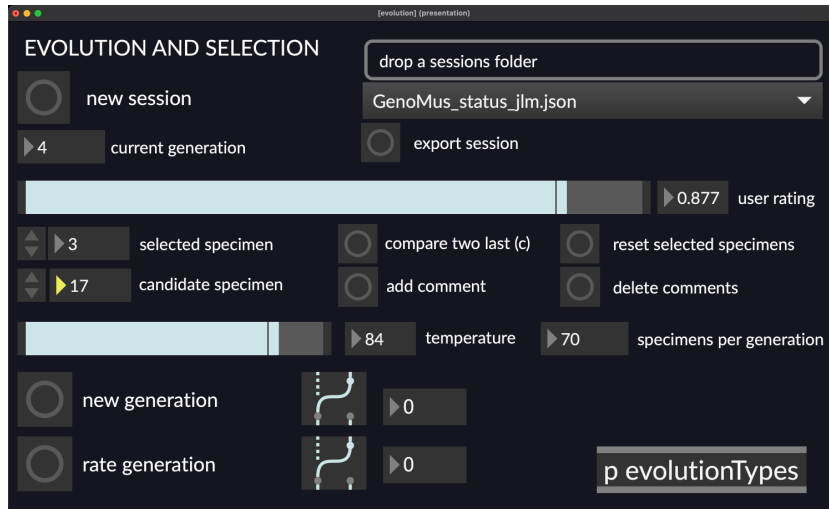


Figure 70: Subpatch evolution to handle evolutionary processes

The subpatch that drives the process of selection, evaluation, and evolution of specimens (Figure 70) is still a very simple prototype, but capable of handling the large number of musical fragments that can be generated in a very short time. Figure 71 illustrates the metadata viewer of the current state of the evolutionary processes, while Table 19 describes the functionalities of this interface.

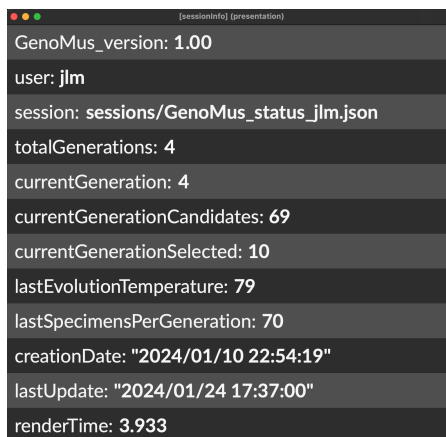


Figure 71: Subpatch sessionInfo to monitor metadata of generative sessions



| Interface item           | Description   |
|--------------------------|---|
| new session              | Begins a new <i>session</i> of music generation, selection, and evolution.  |
| current generation       | Displays the number of the current generation. It is possible to return to previous generations to create new branches of evolution or recover specific specimens.  |
| drop a session folder    | Loads a recorded session from a file. This includes all the selected specimens from all the generations produced up to that moment. The process of selection and evolution can continue from any of the generations in that session.  |
| export session           | Saves the current session in JSON format, with all the selected specimens from all the generations produced.  |
| user rating              | Whether entered numerically or using the slider, this is the most important action for human-supervised selection. Each time a specimen receives a rating $> 0$ , it is selected and has the chance to move to the next generation. A specimen rating can be changed at any time. If its rating is reduced to 0, it is discarded and will not be used in the next generation. |
| selected specimen        | Displays the number of the current selected specimen. The number is assigned descendingly based on its rating. For a quick comparison, keyboard shortcuts are useful: the Q and W keys move backward and forward in the selected specimens, and the numeric keys 1 to 9 directly call the specimen that occupies that ranking.  |
| candidate specimen       | Displays the number of the current candidate specimen, according to the order of production in the current generation. For a quick comparison, keyboard shortcuts are useful: the A and S keys move backward and forward in the candidate specimens.  |
| compare two last         | To facilitate comparison between candidates, the button (or the shortcut C) alternately calls the last two specimens displayed in the score viewer.   |
| reset selected specimens | Sets the rating of all specimens in the current generation to 0.  |
| add comment              | Adds user comments to the metadata of the current specimen. As many as desired can be added at any time.  |
| delete comments          | Deletes all comments from the current specimen.   |

|                           |  |
|---------------------------|--|
| temperature               | <i>Temperature</i> has become a standard property to refer to how much a generative AI is allowed to deviate from the models. Here, this analogy is also used. The higher the temperature, the more possible deviation from the selected specimens, and the more new specimens are added to the pool of candidates for the new generation. |
| specimens per generation  | Determines the number of specimens that will make up the new generation.   |
| new generation            | Produces a new generation according to the current parameters.   |
| rate generation           | When there is a predefined fitness function, such as a phenotype to approach with metagramming, an unsupervised assessment of all candidates takes place. The bypasses to the right allow the process of creating new generations and their automatic assessment to occur in a loop, allowing the system to work autonomously.             |
| <subpatch> evolutionTypes | Accesses the configuration of multiple processes used to generate each generation, and allows altering their weight in the mix made according to the temperature.  |

Table 19: Functionalities of the evolution subpatch

## A.8. Outputs

The outputs subpatch (Figure 72) exports scores as MIDI files and SVG files (producing vector graphics like those used in many of the figures in this text). It also exports visualizations of the encoded data according to the criteria outlined in Section 5.3.

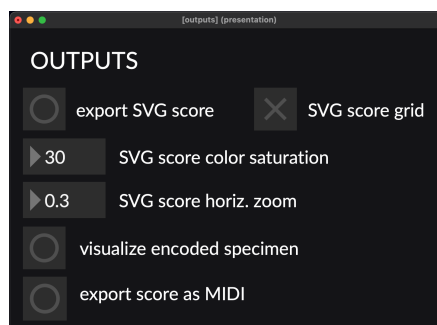


Figure 72: Subpatch outputs



## Musical works

“

Tout le monde vous dira que je ne suis pas un musicien. C'est juste. Dès le début de ma carrière, je me suis, de suite, classé parmi les phonométrographes. Mes travaux sont de la pure phonométrie. Que l'on prenne *Le Fils des étoiles* ou les *Trois morceaux en forme de poire*, *En habit de Cheval* ou les *Sarabandes*, on perçoit qu'aucune idée musicale n'a présidé à la création de ces oeuvres. C'est la pensée scientifique qui domine. Du reste, j'ai plus de plaisir à mesurer un son que je n'en ai à l'entendre. Le phonomètre à la main, je travaille joyeusement et sûrement.<sup>70</sup>

Erik Satie [132]

This appendix delves into the composition methods applied in the musical pieces created at different stages of GenoMus development. They are presented here in chronological order.

Although the initial works presented here were composed with simplified prototypes of the current model, quite different in their implementation, I consider it very interesting to gather them here to observe the initial motivations and how the different technical solutions have evolved based on the artistic needs identified in each case.

The latest electroacoustic pieces in this section have indeed been created using versions very similar to the final one presented in the main text of this thesis.

---

<sup>70</sup>Everyone will tell you that I am not a musician. That's true. From the very beginning of my career, I immediately classified myself among the phonometers. My work is purely phonometric. Whether one takes *The Son of the Stars* or the *Three Pieces in the Shape of a Pear*, *In Horse's Attire* or the *Sarabands*, it is evident that no musical idea presided over the creation of these works. It is scientific thought that prevails. Furthermore, I take more pleasure in measuring a sound than I do in hearing it. With the phonometer in hand, I work joyfully and confidently. (Author's translation)

## B.1. *Threnody for Dimitris Christoulas*

|                         |   |
|-------------------------|---|
| <b>Instrumentation:</b> | flute (alt. piccolo), B-flat clarinet (alt. bass clarinet), violoncello, piano and electronics (stereo)<br>The performers wear headphones and synchronize with the tape using 4 different click tracks  |
| <b>Duration:</b>        | 14 min  |
| <b>Premieres:</b>       | November 4th, 2012 at Teatros del Canal, Madrid<br>November 15th, 2012 at Auditorio de Zaragoza<br>December 3rd, 2012 at Auditori de Barcelona<br>December 11th, 2012 at Sala Joaquín Turina de Sevilla |
| <b>Performers:</b>      | Taller Sonoro<br>Jesús Sánchez, flute<br>Camilo Irizo, clarinet<br>Mery Coronado, violoncello<br>Ignacio Torner, piano<br>Javier Campaña, electronics   |

### Artistic concept

Dimitris Christoulas is the name of a retired Greek pharmacist who shot himself in the head at Syntagma Square in Athens, in front of the Greek Parliament, on April 4, 2012. He left a suicide note explaining that, due to the global economic crisis, the reduction of his pension forced him to search for food in the trash, so he preferred to end his life with dignity, calling for the rebellion of the youth. The conceptual framework of the piece has this theme as an extramusical backdrop, although without programmatic or illustrative intent. The reference to Dimitris Christoulas establishes a connection with the emotional world and the historical context in which the composition task took place. At some point, this event became linked to the work. Its influence on the piece and the audience's perception lies in a subjective realm and falls outside the scope of this work. *Threnody for Dimitris Christoulas* was the first work exploring the techniques of GenoMus's metaprogramming. Its melodic materials are derived almost entirely from the genotypes generated with the first operational prototype of the program.

## Reception

After the premiere, the critic Pablo J. Payón [150] published a review:

[...] Taller Sonoro cerró su propio festival con la propuesta más arriesgada de cuantas se han visto en él. El conjunto sevillano ha cultivado siempre repertorios muy cercanos a la experimentación, y ese terreno en el que el arte convencional se aproxima al llamado arte sonoro, cuando no se cruza con él, es el que pisaron ayer.

[...] Para el final quedó la, en mi opinión, más ambiciosa obra del programa. Originales del granadino José López-Montes (Guadix, 1977), esos *Threnody* están concebidos como un virtuoso diálogo instrumental, con un trabajo refinadísimo sobre las texturas y un empleo muy sutil de la electrónica.<sup>71</sup>

## Methods

For this initial exploration of metaprogramming possibilities, a very simple first prototype was devised that created recursive mathematical formulas. Recursion allows for a wide variety of behaviors to be achieved from extremely simple formulas. A review of the genotypes and phenotypes detailed in Figure 73 showcases the richness of possible textures.

The notion that each element of a composition is deduced from the previous one is very old, but in the case of recursion, its meaning is literal. The sequences thus generated embody a certain sense of *truth*, akin to Cage, opposing manufactured beauty: they are objects with an independent and unalterable existence, not bending to the composer's tastes.

Each *Recurso* starts from a motif of up to four notes, which constitute the initial conditions for each recursion. The remaining notes are deduced from these initial pitches. These recursive equations combine basic arithmetic and trigonometric operators, numerical constants, and variables referencing the initial terms of each succession. The names of the genotype functions are self-explanatory. In Tables 20 and 21, the recursions created through metaprogramming have been translated into standard mathematical notation.

---

<sup>71</sup>[...] Taller Sonoro closed its own festival with the most daring proposal of all that have been seen in it. The Sevillian ensemble has always cultivated repertoires very close to experimentation, and it was that ground where conventional art approaches what are known as sound art, if it doesn't outright intersect with it, that they tread upon yesterday. [...] Saving the most ambitious work of the program for last, in my opinion, was the work by José López-Montes (Guadix, 1977). These *Threnody* are conceived as a virtuoso instrumental dialogue, with an extremely refined work on textures and a very subtle use of electronics. (Author's translation)

The rules established for this composition process allowed total freedom for the manual production and shaping of genotype-phenotype pairs. However, once these pairs were selected, they required strict adherence, despite difficulties in tessitura and, in some cases, rhythm.

Translating these abstract sequences of pitches into rhythmic, contrapuntal, or timbral textures was guided primarily by allowing oneself to be carried toward the kind of motion that, in many cases, the sequence itself clearly suggested. Always aiming for the instrumental arrangement to reveal its formal details in the most perceptible manner.

In some cases, a stretched rhythmic approach was chosen to emphasize the harmonic aspects of the pitches; in others, a rapid and constant *perpetuum mobile* rhythm was employed to highlight much more prolonged structural details.

## A genetic algorithm for electronics

For the production of the electroacoustic stereo tape, the following premises were employed:

- Sound synthesis was used as a testing ground for a prototype genetic algorithm in MaxMSP, which was applied in manipulating a synthesizer programmed for this purpose.
- Electronic sounds served two functions: to enhance the sonority of the instruments by coloring or expanding their timbre and to act as an additional virtual instrument, engaging in counterpoint as another part. In many cases, the line between these two functions is blurred.
- All electronic sounds originate from a single virtual instrument acting as a filter bank that operates on an audio source (usually noise or real sounds with numerous impurities).

In the composition of *Threnody for Dimitris Christoulas*, a genetic algorithm programmed in JavaScript was utilized and integrated as part of the MaxMSP synthesizer described above. Given that the behavior of this synthesizer depends on numerous variables, whose interaction is also hard to predict, the decision was made to use a genetic algorithm due to two compelling consequences it had for the composer:

- It facilitated the rapid creation of a wide range of timbres. This search system is immune to the composer's biases, who tends to manipulate the synthesizer controls based on a preconceived idea of the type of sounds being sought.
- It expedited the timbre modeling process, allowing for a global perspective without concerning oneself with the detailed parameterization of each timbre.

The genetic algorithm generated a fertile interaction between the richness of initial proposals (independent of the composer's hand) and its ability to restrict and mutate the selected results based on artistic decisions, which could be made without attending to the technical details of the implementation.

Although this algorithm is not documented in this work, it served as a precursor to the programming of *GenoMus*, as it reproduced, in a way, the same approach to assisted creativity in the realm of timbre and sound synthesis.

As preparation for the electronic section, an electronic miniature titled *eHayku (Study for Threnody)* was composed. This piece delved into exploring the timbral palette of the instrument created. The timbre of the electroacoustic elements *Threnody for Dimitris Christoulas* closely resembles this piece, available at <https://vimeo.com/lopezmontes/ehayku>.



Figure 73: Threnody for Dimitris Christoulas — all genotypes and graphical phenotypes



| Recursio | Recursive expression  | Initial values  | MIDI mapping                |
|----------|---|---|-----------------------------|
| I-a      | $x_n = x_{n-1} + \sin \frac{2x_{n-1} + x_{n-3}}{x_{n-2} + 0.482}$   | $\begin{cases} x_1 = -0.7349 \\ x_2 = 0.663 \\ x_3 = 0.4338 \end{cases}$                    | $f(x) = 30x + 66$           |
| I-b      |   | $\begin{cases} x_1 = -0.3749 \\ x_2 = 0.0063 \\ x_3 = 0.2416 \end{cases}$                   |                             |
| II       | $x_n = \sin \left( \frac{0.422}{x_{n-1} + x_{n-3}} + \frac{x_{n-2}(x_{n-1} + 1)}{x_{n-1} \cdot x_{n-3}} \right)$                      | $\begin{cases} x_1 = -0.4249 \\ x_2 = -0.7214 \\ x_3 = -0.3279 \end{cases}$                 | $f(x) = 30x + 66$           |
| III-a    | $x_n = -0.0336 - x_{n-1} \left( x_{n-2} + \frac{x_{n-4}}{x_{n-3}} \right)$  | $\begin{cases} x_1 = 0.3205 \\ x_2 = 0.3341 \\ x_3 = 0.349 \\ x_4 = -0.1753 \end{cases}$    | $f(x) = 30x + 66$           |
| III-b    |   | $\begin{cases} x_1 = 0.0958 \\ x_2 = -0.4391 \\ x_3 = -0.3248 \\ x_4 = -0.7869 \end{cases}$ |                             |
| IV-a     | $x_n = \tan \left( \frac{x_{n-4}}{x_{n-2}} + \frac{x_{n-2} + x_{n-3}}{x_{n-2} + x_{n-4}} \right)$                                     | $\begin{cases} x_1 = -0.7776 \\ x_2 = -0.6955 \\ x_3 = -0.6598 \\ x_4 = -0.102 \end{cases}$ | $f(x) = 30x + 66$           |
| IV-b     |   | $\begin{cases} x_1 = -0.3769 \\ x_2 = -0.1265 \\ x_3 = 0.0131 \\ x_4 = 0.0387 \end{cases}$  |                             |
| V        | $x_n = \sin \frac{x_{n-3}}{x_{n-1}}$  | $\begin{cases} x_1 = 0.1859 \\ x_2 = -0.1703 \\ x_3 = -0.2052 \end{cases}$                  | $f(x) = 30x + 66$           |
| VII      | $x_n = \tan \left( \left( x_{n-2} \left( (x_{n-4} \cdot x_{n-3}) + 0.219 \right) \right) - x_{n-1} \right)$                           | $\begin{cases} x_1 = 0.4338 \\ x_2 = 0.663 \\ x_3 = 2 \end{cases}$                          | $f(x) = 30x + 66$           |
| VIII     | $x_n = \cos \frac{x_{n-1}}{x_{n-4}}$  | $\begin{cases} x_1 = -0.4 \\ x_2 = -0.8 \\ x_3 = 0.657 \\ x_4 = 0.526 \end{cases}$          | $f(x) = 30x + 66$           |
| IX       | $x_n = \frac{x_{n-1} + 6}{x_{n-1}}$   | $x_1 = 0.4678$  | $f(x) = \frac{3x + 119}{2}$ |
| X        | $x_n = \tan \frac{x_{n-2} + x_{n-3}}{\frac{\sin x_{n-1}}{x_{n-3}} + \sin \frac{x_{n-2}}{x_{n-3}} \cdot \sin \frac{x_{n-4}}{x_{n-2}}}$ | $\begin{cases} x_1 = 0.9293 \\ x_2 = -0.0314 \\ x_3 = -0.4355 \\ x_4 = -0.2143 \end{cases}$ | $f(x) = 30x + 66$           |
| XI       | $x_n = \tan \frac{0.145}{\frac{x_{n-2}}{x_{n-1}}}$  | $\begin{cases} x_1 = 0.1962 \\ x_2 = 0.141 \end{cases}$                                     | $f(x) = 30x + 66$           |
| XII      | $x_n = \frac{x_{n-3}}{\frac{\sin x_{n-4}}{x_{n-4}}}$  | $\begin{cases} x_1 = 0.297 \\ x_2 = 0.8 \\ x_3 = 0.4281 \\ x_4 = 0.54 \end{cases}$          | $f(x) = 30x + 66$           |

Table 20: Threnody for Dimitris Christoulas: *formulae for Recursio I-a to XII*

| Recursio            | Recursive expression                   | Initial values   | MIDI mapping      |
|---------------------|--|--|-------------------|
| XIII-a              | $x_n = x_{n-2} - x_{n-4} - \sin 0.033$ | $\begin{cases} x_1 = -0.4024 \\ x_2 = -0.6307 \\ x_3 = -0.6255 \\ x_4 = -0.3047 \end{cases}$ | $f(x) = 30x + 66$ |
| XIII-b              |  | $\begin{cases} x_1 = -0.4419 \\ x_2 = 0.3535 \\ x_3 = -0.3537 \\ x_4 = -0.6362 \end{cases}$  |                   |
| XIII-c              |  | $\begin{cases} x_1 = 0.1802 \\ x_2 = -0.0433 \\ x_3 = -0.531 \\ x_4 = -0.2095 \end{cases}$   |                   |
| XIII-d <sub>1</sub> |  | $\begin{cases} x_1 = -0.198 \\ x_2 = 0.7603 \\ x_3 = -0.3265 \\ x_4 = -0.1742 \end{cases}$   |                   |
| XIII-d <sub>2</sub> |  | $\begin{cases} x_1 = -0.19 \\ x_2 = -0.76 \\ x_3 = -0.28 \\ x_4 = -0.08 \end{cases}$         |                   |
| XIII-d <sub>3</sub> |  | $\begin{cases} x_1 = -0.22 \\ x_2 = -0.73 \\ x_3 = -0.248 \\ x_4 = -0.08 \end{cases}$        |                   |
| XIII-d <sub>4</sub> |  | $\begin{cases} x_1 = -0.26 \\ x_2 = -0.73 \\ x_3 = -0.248 \\ x_4 = -0.08 \end{cases}$        |                   |

Table 21: Threnody for Dimitris Christoulas: *formulae for Recursio XIII-a to XII-d<sub>4</sub>*

Below is the beginning of all the sections of the piece, relating genotypes, phenotypes, and musical notation. The order of the sections in the piece slightly alters the numerical order of the listed recursive processes. Additionally, the *Recursio VI* is not included because it is a purely electronic interlude not generated using the general method.

Recursio I-a

$(vAdd(x1, (vSin((vDiv((vAdd(x1, (vAdd(x3, x1))))), (vDiv((vAdd(x2, 0.48196260851064854)), x3))))), (vSin(x1, x2))))$



Θρηνωδία για τον Δημήτρης Χριστούλας  
Threnody for Dimitris Christoulas

for flute (alt. piccolo), clarinet in B $\flat$  (alt. bass clarinet), cello, piano and tape

José López-Montes 1977

transposing score  
accidentals only affect the notes they immediately precede

**Recursio I-a**

$\text{♩} = 70$   
ascoltando

flute

clarinet in B $\flat$

cello

piano

2da

1

3/4, 5/4, 4/4, 5/8, 3/4, 4/4, 5/8

pp, mp, p, PPP, poco vibr., mf, pp, senza vibr.

con sord., sul II-III, sul pont., sul tasto, senza vibr. col piano, III, IV, sfz

fingertips, sul pont. (playing on the keyboard but pressing strings very close to the bridge with the other hand), fingertips, muted with palm of hand, ond., f, p

Figure 74: Recursio I-a — genotype, graphical phenotype and beginning of score

## Recursio II

```
(vSin((vAdd((vDiv(0.4224016310636478, (vAdd(x1, x3))))), (vDiv((vAdd(x2, (vMult(x2, x1))))), (vDiv(x1, (vMult(x3, x2))))))))), (vAdd((vDif((vDiv((vDiv(x2, x2)), (vSin(x1, x4))))), 0.600562209309903)), (vDiv(x3, (vDif(x4, (vAdd(x2, x4))))))))))
```



*Recursio II*  
 con moto  
 quasi cadenza ad libitum

picc. *presto*  
 ↓ (from cello) ↓ (from cello)  
 stringendo.....  $\frac{4}{4} = 86$   $\frac{2}{4}$  marcato

cel. arco sul pontic.— sul tasto sul III. ord. poco sul pontic. s. pontic. ord. sul IV. *pp subito molto cresc.*

*fff* *p* *ff feroce* *mp espress.* *pp cresc.* *p* senza vibr. *mf* *ppp cresc.* *mf* *sf f* *ppstático*

*fff* senza vibr. *mf* dinamica e tempo molto irregolare ma sempre cresc. *pp subito molto cresc.*

*pp subito molto cresc.*

Figure 75: Recursio II — genotype, graphical phenotype and beginning of score

**Recursio III-a**

```
(vDif(-0.0335660162618765, (vMult(x1, (vAdd(x2, (vDiv(x4, x3))))))))
```



*Recursio III-a*

cl. *f* molto rubato *ord.* *mp* *sf* *ppp* *molto cresc.* *f* *pp* *sfz* *sf* *non dim.* *mp* *ppp* *vibr.* *senza vibr.*

cel. *sf* *pp statico*

$\text{♩} = 92$   
ancora più

Figure 76: Recursio III-a — genotype, graphical phenotype and beginning of score

### Recursio III-b

```
(vDif(-0.0335660162618765, (vMult(x1, (vAdd(x2, (vDiv(x4, x3))))))))
```



Recursio III-b



Figure 77: Recursio III-b — genotype, graphical phenotype and beginning of score

### Recursio IV-a

$(\text{vTan}((\text{vAdd}((\text{vDiv}(x4, x2))), (\text{vDiv}((\text{vAdd}(x2, x2))), (\text{vAdd}(x4, x2))))), (\text{vMult}(x2, (\text{vAdd}(x4, x1))))))$



*Recursio IV-a*

fl. *piccolo*

bass cl. *bass clarinet in B $\flat$*

cel. *arco*

pno.

5/4 2/4 3/4 2/4 3/4

*ff sf sf secco mp f pp mf pp f p pp possibile p*

*ff sf mp sf p pp mf pp mf ff p pp p pp sf sf*

Figure 78: Recursio IV-a — genotype, graphical phenotype and beginning of score

### Recursio V

$(v\text{Sin}((v\text{Div}(x3, x1)), x4))$



*Recursio V*

♩ = 94.5  
2/4 incisivo

picc.

♩ = 84  
6/8 incisivo

bass cl.

arco sempre estr. sul pont.  
6/8 incisivo

cel.

with clarinet  
♩ = 94.5  
2/4 incisivo

pno.

sul pont.  
f sempre molto secco

Figure 79: Recursio V — genotype, graphical phenotype and beginning of score



**Recursio IV-b**

`(vTan((vAdd((vDiv(x4, x2)), (vDiv((vAdd(x2, x3)), (vAdd(x4, x2)))))), (vMult(x2, (vAdd(x4, x1))))))`



*Recursio IV-b*

6/16      9/16      6/16      2/4      3/4      2/4      7/8

picc. *pp* *sf* *f* *sf* *mp* *sfpp* *espress.* *poco cresc.* *ff* *p* *mf* *f* *p possibile*

bass cl. *sfp* *pp* *sf* *mf* *f* *pp* *p* *sf* *wide vibr.* *ff* *sf* *senza vibr.* *mf* *p* *mf* *ff* *p dolce*

col. *ord.* *sfp* *sf* *f* *f* *wide vibr. molto cresc.* *sf* *sfp senza vibr. e non dim.* *sf* *p* *sf* *f* *f* *mp* *f* *mf*

pno. *pp* *p* *sf* *sf* *mf* *f* *pp* *mf* *mf* *mf* *f* *pp* *mf* *sf* *pp* *ppp*

Figure 80: Recursio IV-b — genotype, graphical phenotype and beginning of score

### Recursio VII

(vTan((vDif((vMult((vAdd((vMult(x4, x3)), -0.21903280447337825)), x2)), x1)), x4))



**Recursio VII**

$\frac{3}{4}$   $\downarrow = 82$   $\frac{2}{4}$   $\frac{3}{4}$   $\frac{2}{4}$   $\frac{3}{8}$   $\frac{3}{4}$   $\frac{5}{4}$

feroce

picc. senza vibr. *sfpp* *8va*

bass cl. senza vibr. *fff* *sf secco* *Lu clarinet in Bb*

cel. *ord.* *sul pont.* *tutta forza* *mf* *sf* *pp* *fff* *sf* *secco* *molto vibr.* *pizz.* *senza vibr.* *arco sul tasto* *fff* *senza vibr.*

pno. *f* *mf* *sf* *pp* *fff* *f espress.* *mf* *p* *pp* *ppp* *pp* *f* *ppp* *fff* *ppp* *pp* *f* *ppp* *fff* *ppp* *pp* *f* *ppp* *fff*

Figure 81: Recursio VII — genotype, graphical phenotype and beginning of score

### Recursio VIII

$(\text{vCos}((\text{vDiv}(x1, x4)), (\text{vLog}(x3, x4))))$



*Recursio VIII*



Figure 82: Recursio VIII — genotype, graphical phenotype and beginning of score

## Recursio IX

$(vDiv((vAdd(x1, -6)), x1))$



*Recursio IX*

The musical score is divided into four staves: Piccolo (pic.), Clarinet (cl.), Cello (cel.), and Piano (pno.). Above the Piccolo staff, the following time signatures are indicated:  $\frac{4}{8}$  (with  $\text{♩} = 116$  and *incisivo*),  $\frac{5}{16}$  (with  $3+2$ ),  $\frac{7}{16}$  (with  $2+2+3$ ),  $\frac{2}{8}$ ,  $\frac{5}{16}$  (with  $2+3$ ),  $\frac{7}{16}$  (with  $2+2+3$ ),  $\frac{2}{8}$ ,  $\frac{5}{16}$  (with  $2+3$ ), and  $\frac{7}{16}$  (with  $2+2+3$ ). Performance instructions for the Piccolo part include *sfpp*, *molto cresc.*, *flatterzunge*, *stap longue*, *quasi scratch tone*, *ff*, and *sempre simile*. The Piano part is marked *ff sempre martellato* and *sempre simile*. The Cello part includes the instruction *ff sempre molto secco*.

Figure 83: Recursio IX — genotype, graphical phenotype and beginning of score

### Recursio X

```
(vAtan2((vAdd(x2, x3)), vAdd((vDiv((vSin(x1, x2))), x3)), vMult((vSin((vDiv(x2, x3))), (vLog(x2, x1))))), (vSin((vDiv(x4, x2)), x4))))))
```



*Recursio X* 14

Figure 84: Recursio X — genotype, graphical phenotype and beginning of score

### Recursio I-b

$(\mathbf{vAdd}(x1, (\mathbf{vSin}((\mathbf{vDiv}((\mathbf{vAdd}(x1, (\mathbf{vAdd}(x3, x1))))), (\mathbf{vDiv}((\mathbf{vAdd}(x2, 0.48196260851064854)), x3))))), (\mathbf{vSin}(x1, x2))))))$



*Recursio I-b*

♩ = 66  
statico

5/4 4/4 5/4 3/8 2/4 3/4 5/4

fl. *p* sempre senza vibr. *p* *sfp*

cl. *mp* sempre senza vibr. *p* *mp* *p dolce*

cel. con sord. *mp* *ppp* *mf* *mp* *mf*

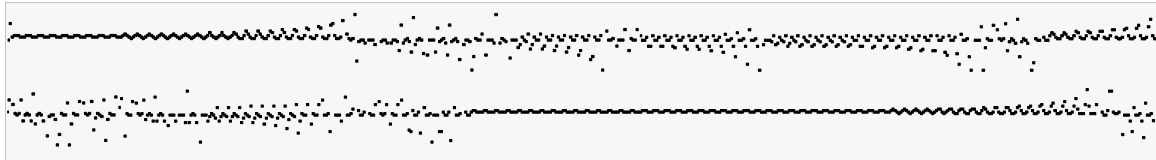
18

pno. *mf* *f* *mp* *f* *p* *mp*

Figure 85: Recursio I-b — genotype, graphical phenotype and beginning of score

**Recursio XI**

```
(vTan((vDif((vAdd((vDiv((vDiv(0.12,0.83)),(vDiv(x2,x1))))),x1)),x1)),(vAdd(-0.02,(vDiv((vPow(x1,(vLog(x1,x2))))),x2))))))
```



**Recursio XI**

$\text{♩} = 72$   
 $\frac{6}{16}$  quasi toccata

21  $\frac{3}{16}$

picc. *[flute]*  
*pppp* key clicks (accents possibly with tongue pizz)  
*pppp* key clicks

d. *stacc.* *mf* *ppp*

cel. *pizz.* *stacc.* *arco* *legno battuto* *ord.* *sul pont.*

pno.  $\text{♩} = 72$   $\frac{6}{16}$  quasi toccata  
*ff* *mp* *f* *ppppp* *2* *2* *2*

*plectrum or fingernail* *sul pont.* *muted* *ord.*

*sempre senza*  $\text{♩} = 72$

Figure 86: Recursio XI — genotype, graphical phenotype and excerpts from score

Musical works  
B.1. Threnody for Dimitris Christoulas

The score continuation for Figure 87 consists of two systems of music, measures 31 and 32, for four instruments: Flute (fl.), Clarinet (cl.), Cello (cel.), and Piano (pno.).

**Measure 31:**

- Flute:** Starts with a box containing the number 31. The tempo is  $\text{♩} = 193.656$  and the mood is "giusto". The time signature is  $\frac{2}{32}$ , which changes to  $\frac{7}{32}$ , then  $\frac{5}{16}$ ,  $\frac{2}{32}$ ,  $\frac{5}{16}$ , and finally  $\frac{2}{32}$ . Dynamics include *sf* and *fff*. Performance instruction: "molto percussivo e marcato".
- Clarinet:** Starts with a box containing the number 31. The tempo is  $\text{♩} = 186.207$  and the mood is "giusto". The time signature is  $\frac{2}{32}$ , which changes to  $\frac{5}{32}$ ,  $\frac{4}{32}$ ,  $\frac{3}{32}$ , and finally  $\frac{2}{32}$ . Dynamics include *f stacc.*, *pp*, *f*, *ff stacc.*, and *pp*.
- Cello:** Starts with a box containing the number 31. The tempo is  $\text{♩} = 216$  and the mood is "giusto". The time signature is  $\frac{2}{32}$ , which changes to  $\frac{7}{32}$ ,  $\frac{3}{16}$ ,  $\frac{5}{32}$ , and finally  $\frac{2}{32}$ . Performance instruction: "estrem. sul pont.". Dynamics include *f*. A "ord." (ordine) instruction is present.
- Piano:** Starts with a box containing the number 31. The tempo is  $\text{♩} = 201.103$  and the mood is "giusto". The time signature is  $\frac{2}{32}$ , which changes to  $\frac{1}{32}$ ,  $\frac{2}{32}$ ,  $\frac{1}{32}$ ,  $\frac{2}{32}$ , and finally  $\frac{2}{32}$ . Dynamics include *ppp*, *ff*, *ppp*, *ff*, *pp*, and *pp*. Performance instruction: "senza ped." (senza pedale).

**Measure 32:**

- Flute:** Starts with a box containing the number 32. The tempo is  $\text{♩} = 216$  and the mood is "giusto". The time signature is  $\frac{2}{32}$ , which changes to  $\frac{1}{32}$ ,  $\frac{4}{8}$ , and finally  $\frac{4}{8}$ . The mood changes to "lirico". Dynamics include *sf*, *sf*, *p*, *ff*, *p*, *f*, *fff*, and *f rubato ad lib.*. Performance instruction: "slap tongue".
- Clarinet:** Starts with a box containing the number 32. The tempo is  $\text{♩} = 216$  and the mood is "giusto". The time signature is  $\frac{2}{32}$ , which changes to  $\frac{1}{32}$ ,  $\frac{4}{8}$ , and finally  $\frac{4}{8}$ . The mood changes to "lirico". Dynamics include *sf*, *f espress.*, *f*, *ff*, and *pp*.
- Cello:** Starts with a box containing the number 32. The tempo is  $\text{♩} = 216$  and the mood is "giusto". The time signature is  $\frac{2}{32}$ , which changes to  $\frac{1}{32}$ ,  $\frac{4}{8}$ , and finally  $\frac{4}{8}$ . The mood changes to "lirico". Dynamics include *ff*, *sfpppp*, *sfpppp*, *f espress.*, *p*, *sf*, *pp*, *sf*, and *p*. Performance instruction: "sul pont.". A "rubato ad lib." instruction is present.
- Piano:** Starts with a box containing the number 32. The tempo is  $\text{♩} = 216$  and the mood is "giusto". The time signature is  $\frac{2}{32}$ , which changes to  $\frac{1}{32}$ ,  $\frac{4}{8}$ , and finally  $\frac{4}{8}$ . The mood changes to "lirico". Dynamics include *ff secco*, *ff*, *tutta forza*, *ff*, *p*, *f*, and *sf*.

Figure 87: Recursio XI — score continuation



### Recursio XII

$(\text{vDiv}(x3, (\text{vDiv}((\text{vSin}(x4, (\text{vAdd}(x2, x3))))), x4))))$



### Recursio XII

The musical score for *Recursio XII* is presented in four staves. The flute (fl.) and clarinet (cl.) parts feature a series of triplets, with dynamics ranging from *mp* to *tutta forza (più acuto possibile)*. The cello (cel.) part begins with a *ff* dynamic and includes the instruction *estrem. sul tasto*. The piano (pno.) part includes the instruction *stacc. possibile* and features sixteenth-note patterns. A *cresc.* marking is placed above the piano staff. The score concludes with a *tutta forza* instruction.

Figure 88: Recursio XII — genotype, graphical phenotype and score

**Recursio XIII**

$(\text{vDif}(\text{vDif}(x2, x4)), (\text{vSin}(-0.033, x1)))$



**Recursio XIII-a**

fl.  $\text{♩} = 69.821$   
ipnotico  
*mp* espress. senza vibr. *sf mp* espress. senza vibr. *sf mp* espress. senza vibr. sempre simile

cl.  $\text{♩} = 69.821$   
ipnotico  
*pp* staccatissimo

cel.  $\text{♩} = 90.904$   
ipnotico  
con sord. ott.  
*mp* sempre senza vibr. *ppp* 3+3+3+2 estrem. sul tasto

pno.  $\text{♩} = 46$   
ipnotico  
*sf p* *sf p* *sf p* *sf p* *sf p* *sf p* *sf p* *sf p*

**Recursio XIII-b**

fl.  $\text{♩} = 66.7$   
*mf* espress. *sf* *sf* *sf* *sf* *sf* *sf* *sf* *sf* *sf* *sf*

cl.  $\text{♩} = 45.233$   
*sfmp* *sfmp* *sfmp* *sfmp* *sfmp* *sfmp* *sfmp* *sfmp* *sfmp* *sfmp*

cel.  $\text{♩} = 95.067$   
*ppp* *mp*

pno.  $\text{♩} = 46$   
*sf p* *sf p* *sf p* *sf p* *sf p* *sf p* *sf p* *sf p* *sf p* *sf p*

Figure 89: Recursio XIII — genotype and graphical phenotypes, XIII-a & b — score

Musical works  
B.1. Threnody for Dimitris Christoulas

**Rekursio XIII-d2**

Flute (fl.):  $\frac{3}{4}$   $\text{♩} = 71.464$

Clarinet (cl.):  $\frac{6}{8}$   $\text{♩} = 47.095$

Cello (cel.):  $\frac{12}{16}$   $\text{♩} = 95.833$

Piano (pno.):  $\frac{6}{8}$   $\text{♩} = 46$

Dynamic markings: *sfmp*, *sempre mp*, *fmp*, *sempre sim. ma un poco diminuendo*, *sf p*, *estrem. sul tasto*.

**Rekursio XIII-d3**

Flute (fl.):  $\frac{3}{4}$   $\text{♩} = 69.575$

Clarinet (cl.):  $\frac{6}{8}$   $\text{♩} = 47.15$

Cello (cel.):  $\frac{12}{16}$   $\text{♩} = 95.833$

Piano (pno.):  $\frac{6}{8}$   $\text{♩} = 46$

Dynamic markings: *sfmp*, *sfmp*, *mf*, *sempre sim. ma un poco diminuendo*, *p*, *sf*, *sempre sim. ma un poco diminuendo*, *fmp*, *sempre sim. ma un poco diminuendo*, *sf p*, *estrem. sul tasto*.

Figure 90: Recursio XIII-d2 & d3 — score

Musical works  
B.1. Threnody for Dimitris Christoulas

**Rekursio XIII-d4**

**Rekursio XIII-d5**

Figure 91: Rekursio XIII-d4 & d5 — score

Musical works  
B.1. Threnody for Dimitris Christoulas

*Rekursio XIII-d6*

Figure 92: *Rekursio XIII-d6* — score

Figure 93: Max patch for study of multitempi with individual click tracks

## B.2. *Ada + Babbage – Capricci*

|                         |  |
|-------------------------|--|
| <b>Instrumentation:</b> | violoncello and piano  |
| <b>Duration:</b>        | 18 min   |
| <b>Premieres:</b>       | March 15th, 2013 at Conservatorio Superior de Música<br>Rafael Orozco de Córdoba |
| <b>Performers:</b>      | Trino Zurita, violoncello<br>Óscar Martín, piano                                 |

### Artistic concept

*Ada + Babbage — Capricci* is a duo for cello and piano consisting of 16 caprices inspired by the names of Babbage and Ada<sup>72</sup>, pioneers of modern automated computing in the mid-19th century. Charles Babbage was famous for his mechanical contrivances capable of executing complex calculations, while Ada Lovelace conceived what would later become programming languages and was often described as a virtuoso interpreter of Babbage's machines. The piece is not primarily a tribute but rather a reference to these scientists through an analogy with the procedures of *GenoMus*: while Babbage's role involved conceiving systems of great potential, Ada was the one who understood and formalized the procedures to extract brilliant results from these machines.<sup>73</sup> The similarity between the designer of a musical instrument and the virtuoso performer is direct. In the case of *GenoMus*, the machine is the metalanguage and its possible manipulations, while the composer's decisions and skill are the decisive factors in achieving valuable artistic results.

---

<sup>72</sup>The musical potential of these names was previously explored by Hofstadter [66], who plays with the initials of Bach in dialogue with those of CHARLES Babbage.

<sup>73</sup>It's no coincidence that she is considered the pioneer of programming, and there is a programming language named after her.

The names of Babbage and Ada are musicalized and used as constructive material for the production of the employed genotypes. A series of specific functions were created for the genotypes of this piece. Starting from the special functions `scoNameADA` and `scoNameBABBAGE`, which respectively yield the motives A–D–A and B $\flat$ –A–B $\flat$ –B $\flat$ –A–G–E in Germanic notation, a group of functions (identifiable by using their proper names as well) were derived and integrated into the standard function library of `GenoMus`. The genotypes of each caprice perform various transformations with these motivic cells, which in many cases are easily identifiable in the musical transcription.

The criteria for musicalizing pitch sequences have been closely related to those employed previously in *Threnody* for Dimitris Christoulas. *Ada + Babbage — Capricci* is composed with the first version of `GenoMus` that begins to integrate genotype functions with manipulations and methods specific to musical composition, but still lacks the ability to handle more than one parameter. Therefore, in musical notation, dynamics, and rhythmic notation have been done manually, while the harmonic and melodic material has been generated almost entirely by the metaprogramming mechanisms described.

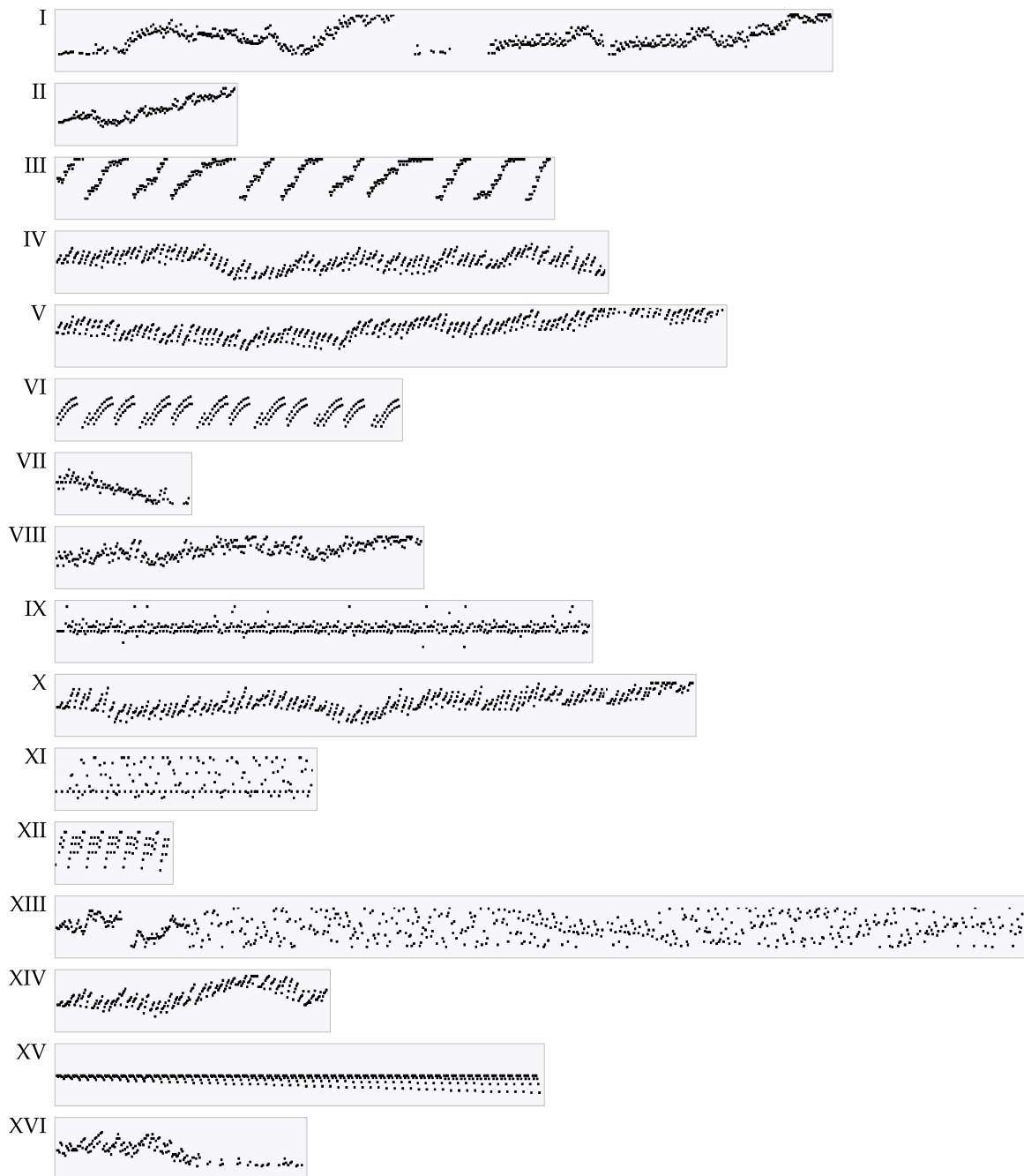


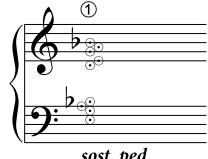
Figure 94: Ada + Babbage – Capricci — graphical phenotypes



## Capriccio I

```
// seed: 0.12267276066200561  
scoTransp(  
  scoADA2voices(  
    scoPCGenPrim(),  
    300),  
  35)
```

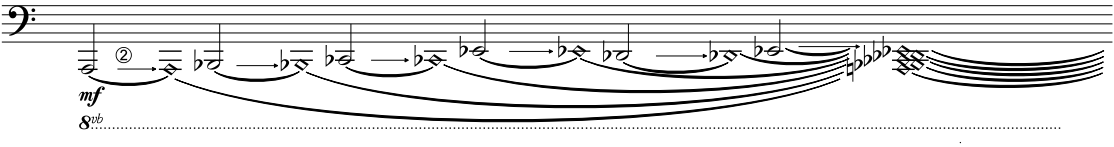


piano  *sost. ped.*

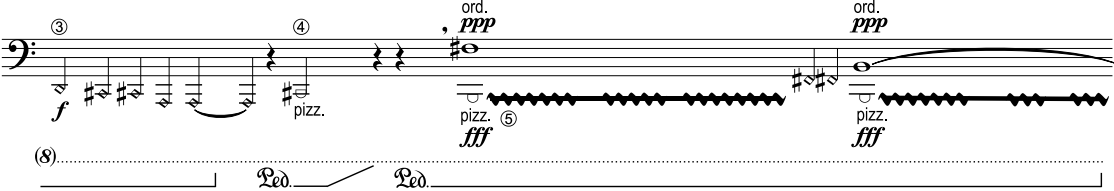
① press down the key silently

*Capriccio I*

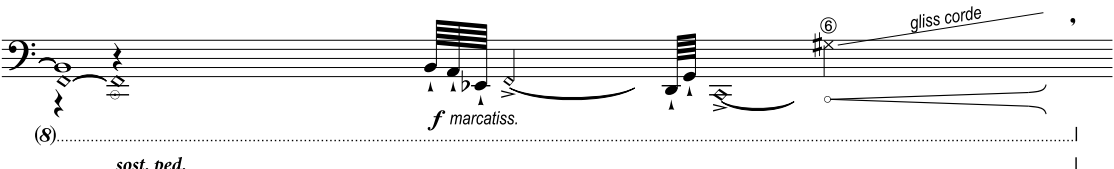
0 *ad libitum ma lentissimo*

pn  *mf*  
*8<sup>va</sup>*

② play a normal note and then put a finger on the string to obtain gradually a random natural harmonic, keeping the keys pressed on the keyboard *ped.*

pn  *f* *ord. ppp* *pizz.* *ord. ppp* *pizz. fff* *fff*

③ random natural harmonic (put the fingertip or a nail somewhere on the string, and press the note on the keyboard)  
④ pizz. the string with the finger  
⑤ push a string to collide with a contiguous vibrating string, sounding as a percussive thrill

pn  *f marc. ass.* *gliss corde* *sost. ped.*

⑥ silently press the note on the keyboard and quickly gliss with the nail longitudinally on the string


pn  *tranquillo* *pp* *poco cresc.* *mp* *sf* *6*

Figure 95: Capriccio I — genotype, graphical phenotype and beginning of score

## Capriccio II

```
// seed: 0.14049629543007258  
scoTransp(  
  scoADA2voices(  
    scoPCGenPrim(),  
    300),  
  35)
```



Capriccio II

*feroce e presto possibile*

vc *ff* *détaché* *p*

1  $\frac{4}{8}$  *feroce e presto possibile*

pn *f* *p*

*marcato*  
sub. senza *Red.* *8<sup>va</sup>*

vc *sf* *pp* *8<sup>va</sup>*

pn *pp* *Red.*

Figure 96: Capriccio II — genotype, graphical phenotype and beginning of score

## Capriccio III

```
// seed: 0.2366373293347337
scoTranspMatrix(
  scoRender(
    scoNameADA(),
    scoExcerptMulti(
      scoInvert(
        scoExcerpt(
          scoRandFlo(
            94,
            25.99744415283203,
            pyth(
              55.94232177734375,
              101)),
          root(
            randInt(
              75,
              54.761192321777344),
            root(
              94,
              15)),
          sin(
            50.85978317260742))),
      scoInter(
        [0,0,0]))))
```



*Capriccio III*

*senza misura ma presto*

*sempre energico e con dinamica molto irregolare ad libitum*

vc

7

*senza misura ma presto*

*sempre energico e con dinamica molto irregolare ad libitum*

8<sup>va</sup>

8<sup>va</sup>

8<sup>vb</sup>

Ped.

9

8<sup>va</sup>

15<sup>ma</sup>

8<sup>vb</sup>

Ped.

The image displays a musical score for 'Capriccio III' by B.2. Ada + Babbage. It is divided into two systems. The first system includes a vocal line (vc) and a piano line (pn). The vocal line starts with the instruction 'senza misura ma presto' and the piano line with 'sempre energico e con dinamica molto irregolare ad libitum'. The piano part features a complex rhythmic pattern with dynamic markings like accents and slurs. The second system continues the vocal and piano parts, with the piano part reaching a '15<sup>ma</sup>' (15th measure) and including further dynamic markings. Pedal points (Ped.) are indicated at the end of both systems.

Figure 97: Capriccio III — genotype, graphical phenotype and beginning of score

## Capriccio IV

```
// seed: 0.06906914511016293  
scoBABBAGE5voicesHarmonicGlobal(  
  scoNameBABBAGE(),  
  200)
```



*Capriccio IV*

$\text{♩} = 112$   
*moderato e molto delicato*  
pizz.  
ben sonoro ma dolce vibrato espress. vibrato espress.

vc

14  $\frac{4}{8}$  *moderato e molto delicato*  
*p* *dolcissimo*  
Ped. Ped. Ped. Ped. Ped. sempre simile

18  $\frac{7}{8}$  senza arpegg. sempre simile

pn

Figure 98: Capriccio IV — genotype, graphical phenotype and beginning of score

## Capriccio V

```
// seed: 0.06906914511016293  
scoBABBAGE5voicesHarmonicGlobal(  
  scoNameBABBAGE(),  
  200)
```



*Capriccio V*

$\text{♩} = 92$   
*tumultuoso*

arco  
*f*

vc

33  $\frac{4}{4}$   $\text{♩} = 92$  *tumultuoso*  $\frac{3}{4}$   $\frac{4}{4}$

*f*

pn

vc

*p* *mf*

35  $\frac{4}{4}$  *8va*

*p* *mf*

Red. Red. Red.

Figure 99: Capriccio V — genotype, graphical phenotype and beginning of score

## Capriccio VI

```
// seed: 0.5024332028236203
scoTranspMatrix(
  scoExcerptMulti2(
    scoTranspMatrix(
      scoInvert(
        scoPermut(
          scoNameADAintervals()),
        scoEnumInt(
          20,
          2)),
      scoNameADA(),
      scoNameADA()),
    scoRender(
      scoNameBABBAGEintervals()))
```





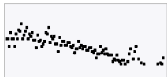
**Capriccio VI**  
♩ = ca. 72  
*come fanfarria*

The image displays the musical score for 'Capriccio VI' in 4/4 time, marked '♩ = ca. 72' and 'come fanfarria'. It is divided into two systems. The first system covers measures 43-44, and the second system covers measures 45-46. The Violoncello (vc) part is written in the bass clef and features a melodic line with dynamic markings of *sf* and *sempre f*, and glissando markings. The Piano (pn) part is written in grand staff (treble and bass clefs) and includes a complex accompaniment with dynamic markings of *sempre f* and *sf*, as well as glissando markings. Pedal markings (*Ped.*) and octave markings (*8vb*) are present in the piano part. The score is annotated with various performance instructions and graphical elements like slurs and accents.

Figure 100: Capriccio VI — genotype, graphical phenotype and beginning of score

## Capriccio VII

```
// seed: 0.6449608976441318  
scoTransp(  
  scoBABBAGE4voicesHarmonic(  
    scoNameBABBAGE(),  
    40),  
  10)
```



*Capriccio VII*

*senza misura ma molto tranquillo*

vc *p sempre quasi senza vibr.* *pp*

*senza misura ma molto tranquillo*

55 pn *p* *pp*

vc *poco più f* *ff* *ff* *ff* *ff*

pn *poco più f* *f* *sf* *sf* *sf* *pizz.* *fff*

*overpressure ord.* *overpressure*

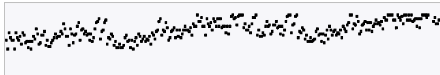
*8vb*

*Leg.* *Leg.* *Leg.* *Leg.* *Leg.*

Figure 101: Capriccio VII — genotype, graphical phenotype and beginning of score

## Capriccio VIII

```
// seed: 0.07204728925454051
scoBABBAGE5voicesHarmonic(
  scoExcerpt(
    scoExpand(
      scoNameADA(),
      floPrim(),
      intPrim(),
      ratio(
        sin(
          log(
            floPrim(),
            floPrim()),
          floPrim()),
        floPrim())
    )
  )
```



**Capriccio VIII**

senza misura e più tranquillo  
ord.  
*pp* spettrale e sempre senza vibrato

56 *pp* spettrale *p* senza misura e più tranquillo

*sost. ped.*

*mp* *f*

*mf* *f* 8<sup>va</sup>

*ppp* molto irregolare ma presto

(8)

*ppp* *mp* *l. vibr.*

The image displays a musical score for 'Capriccio VIII' for voice (vc) and piano (pn). The score is divided into three systems. The first system shows the vocal line with notes and rests, and the piano accompaniment with chords and some melodic lines. Dynamics include *pp* and *p*. Performance instructions include 'senza misura e più tranquillo', 'ord.', 'pp spettrale e sempre senza vibrato', and 'sost. ped.'. The second system continues the vocal and piano parts, with dynamics *mp* and *f* for the voice, and *mf* and *f* for the piano. An '8<sup>va</sup>' marking indicates an octave shift. The third system shows the vocal line with a long note and the piano part with a complex, irregular rhythmic pattern. Dynamics include *ppp* and *mp*. Performance instructions include 'molto irregolare ma presto', '(8)', and 'l. vibr.'. A 2/4 time signature is shown at the end of the third system.

Figure 102: Capriccio VIII — genotype, graphical phenotype and beginning of score

## Capriccio IX

```
// seed: 0.07755023860351185
scoMutate(
  scoADA3voices(
    scoRep(
      scoPitchClass(
        scoNameBABBAGE(),
        scoPCGen(
          intPrim()),
        intPrim()),
      cos(
        intPrim()),
    root(
      absDif(
        floPrim(),
        floPrim()),
      floPrim()),
    randInt(
      intPrim(),
      floPrim()))
```



**Capriccio IX**

$\text{♩} = 78$   
*colico*  
con sordina, flautando e vibr. ord.  
*mp* mezza voce

vc

57  $\frac{2}{4}$  *colico* 5/16 2/4 5/16 2/4

pn  
*mp* sempre mezza voce e legato  
col *Ped.* ad libitum

11

*p*

63  $\frac{2}{4}$  5/16 2/4

pn  
*p*

Figure 103: Capriccio IX — genotype, graphical phenotype and beginning of score



## Cruciverba and Capriccio XI

```
// seed: 0.6912066419198322
scoExpandIter(
  scoRandPrim(),
  dif(
    atan(
      floPrim()),
    floPrim()),
  sqr(
    intPrim()),
  rFlo(
    floPrim(),
    sum(
      intPrim(),
      root(
        intPrim(),
        floPrim()))))
```





*Cruciverba*

*durata ad libitum, ma sempre espressivo*

vc *f* espress. *gliss.* *dim.* *pp* senza vibrato

*Capriccio XI*

$\text{♩} = 102$   
sempre estr. sul pont.  
sempre *f*

143  $\text{♩} = 102$   
5/16  
sempre *f* staccatiss.  
sempre senza Ped.

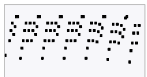
8<sup>va</sup> 8<sup>va</sup> 8<sup>va</sup> 8<sup>va</sup> 8<sup>va</sup>

8<sup>vb</sup> 8<sup>vb</sup> 8<sup>vb</sup> 8<sup>vb</sup>

Figure 105: Capriccio XI — genotype, graphical phenotype and beginning of score

## Capriccio XII

```
// seed: 0.30938694044888126
scoTranspMatrix(
  scoExcerptMulti2(
    scoTranspMatrix(
      scoInvert(
        scoPermut(
          scoNameADAintervals()),
        scoEnumInt(
          46,
          32)),
      scoNameADA(),
      scoNameADA()),
    scoRender(
      scoNameBABBAGEintervals()))
```



Capriccio XII

ancora più  
arco ord.  
ff

188

ancora più

8<sup>va</sup>

f

8<sup>vb</sup>

Red.

vc

pn

191

8

8

Red.

The image displays a musical score for 'Capriccio XII' by B.2. Ada + Babbage. It is divided into two systems. The first system starts at measure 188 and features a violin part (vc) and a piano part (pn). The violin part is marked 'ancora più', 'arco ord.', and 'ff', with 'pochiss. port.' written above the notes. The piano part is marked 'ancora più' and 'f', with '8<sup>va</sup>' above the treble clef and '8<sup>vb</sup>' below the bass clef. The piano part includes a 'Red.' (Reduction) line with four vertical bars. The second system starts at measure 191 and continues the violin and piano parts. The piano part includes a circled '8' above the treble clef and another circled '8' below the bass clef, with 'Red.' lines below. The key signature is one sharp (F#) and the time signature is 4/4.

Figure 106: Capriccio XII — genotype, graphical phenotype and beginning of score

## Capriccio XIII

```
// seed: 0.8853635125980808
scoMutate(
  scoConcat(
    scoConcat(
      scoADA(
        scoBABBAGE(
          scoRand(
            7,
            60,
            72),
          50)),
      scoRetrog(
        scoInter(
          [0,0,0,0,0]))),
    scoTranspMatrix(
      scoRandPrim(),
      scoInter(
        [0,0,0,1]))))
```



Capriccio XIII

*l'istesso tempo ma come toccata*

vc *p* *détaché* *sf* *p*

194 *l'istesso tempo ma come toccata* *p non legato* *8<sup>va</sup>* *Ped.*

vc *sf* *mf*

199 *mf* *8<sup>va</sup>* *Ped.* *8<sup>vb</sup>* *Ped.*

vc *sf* *pp*

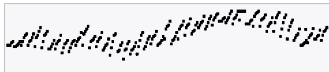
205 *pp* *8<sup>vb</sup>* *8<sup>vb</sup>*

The image displays a musical score for 'Capriccio XIII' in G major, 3/4 time. It is divided into three systems. The first system (measures 188-193) features a violin part with dynamics *p* *détaché*, *sf*, and *p*, and a piano part starting at measure 194 with dynamics *p non legato* and an *8<sup>va</sup>* octave marking. The second system (measures 194-200) shows the violin part with *sf* and *mf* dynamics, and the piano part with *mf* dynamics and *8<sup>va</sup>* and *8<sup>vb</sup>* markings. The third system (measures 201-206) features the violin part with *sf* and *pp* dynamics, and the piano part with *pp* dynamics and *8<sup>vb</sup>* markings. Performance instructions include *Ped.* (pedal) and *(8)* (octave) markings.

Figure 107: Capriccio XIII — genotype, graphical phenotype and beginning of score

## Capriccio XIV

```
// seed: 0.6449608976441318  
scoBABBAGE5voicesHarmonicGlobal(  
  scoTransp(  
    scoNameBABBAGE(),  
    -12),  
  40)
```



*Capriccio XIV*

*molto rubato ad libitum*

arco  
*f* sempre détaché

*ff*

*p* 7

non arpeg.  
tutta forza

pizz non arpegg. sempre

pizz alla chitarra  
*mf* espress.

più *p*

Figure 108: Capriccio XIV — genotype, graphical phenotype and beginning of score

## Capriccio XV

```
// seed: 0.7370028217611009  
scoRender(  
  scoExpandIter(  
    scoNameBABBAGE(),  
    1,  
    6,  
    60))
```



Capriccio XV

$\text{♩} = 98$   
come Babbage's Difference Engine No. 2

vc

232  $\frac{7}{16}$   $\text{♩} = 98$   
come Babbage's Difference Engine No. 2  
*p non stacc.*

pn

236  $\frac{9}{16}$  *mp*

240  $\frac{21}{16} (\frac{3}{16} \times 7)$  *mf*  
Ped.

244

pn

The image displays a musical score for 'Capriccio XV'. It features a vocal line (vc) and a piano accompaniment (pn). The score is divided into four systems. The first system shows the vocal line with a tempo marking of quarter note = 98 and a reference to 'come Babbage's Difference Engine No. 2'. The piano part begins at measure 232 with a 7/16 time signature and a dynamic of piano (p) non staccato. The second system starts at measure 236 with a 9/16 time signature and a dynamic of mezzo-piano (mp). The third system starts at measure 240 with a 21/16 time signature (notated as 3/16 x 7) and a dynamic of mezzo-forte (mf). The fourth system starts at measure 244. The piano part includes various rhythmic patterns, including sixteenth and thirty-second notes, and rests. A pedal point (Ped.) is indicated at the end of the third system.

Figure 109: Capriccio XV — genotype, graphical phenotype and beginning of score



## Capriccio XVI

```
// seed: 0.7370028217611009  
scoBABBAGE5voicesHarmonic(  
  scoNameADA(),  
  40)
```



Capriccio XVI

vc *giocoso*  
arco  
*f*

369 *5* *16* *giocoso*  
*f*

col cello

Red.

vc

377 *ff*

Red.

Figure 110: Capriccio XVI — genotype, graphical phenotype and beginning of score

### B.3. *Microcontrapunctus*

**Instrumentation:** 24-channels tape  
**Duration:** 7 min  
**Premiere:** May 6th, 2016 at Istituto Superiore de Studi Musicali  
Pietro Mascagni in Livorno (Italy)  
**URL:** Vimeo:  
<https://vimeo.com/lopezmontes/microcontrapunctus>

#### Artistic concept

*Microcontrapunctus* was composed to be premiered at the Istituto Superiore de Studi Musicali Pietro Mascagni in Livorno (Italy), within a large space featuring a fixed installation of 24 independent speakers placed on various floors and heights, distributed throughout the volume of the auditorium, as shown in Figure 111. A setup like this suggested the possibility of composing a piece that aimed to highlight the acoustic characteristics specific to each region of the interior space. Thus, the piece was initially conceived as a bombardment of brief purely synthetic sound impulses with a sharp attack and rich harmonic content, emitted following various spatial patterns and trajectories. One of the objectives was for the piece to yield diverse results depending on each acoustic space and technical configuration. This approach is, moreover, quite common in multichannel acousmatic work.<sup>74</sup>

---

<sup>74</sup>De Benedictis [39] has edited several compilations of articles documenting different approaches to acousmatic composition for this unique space.

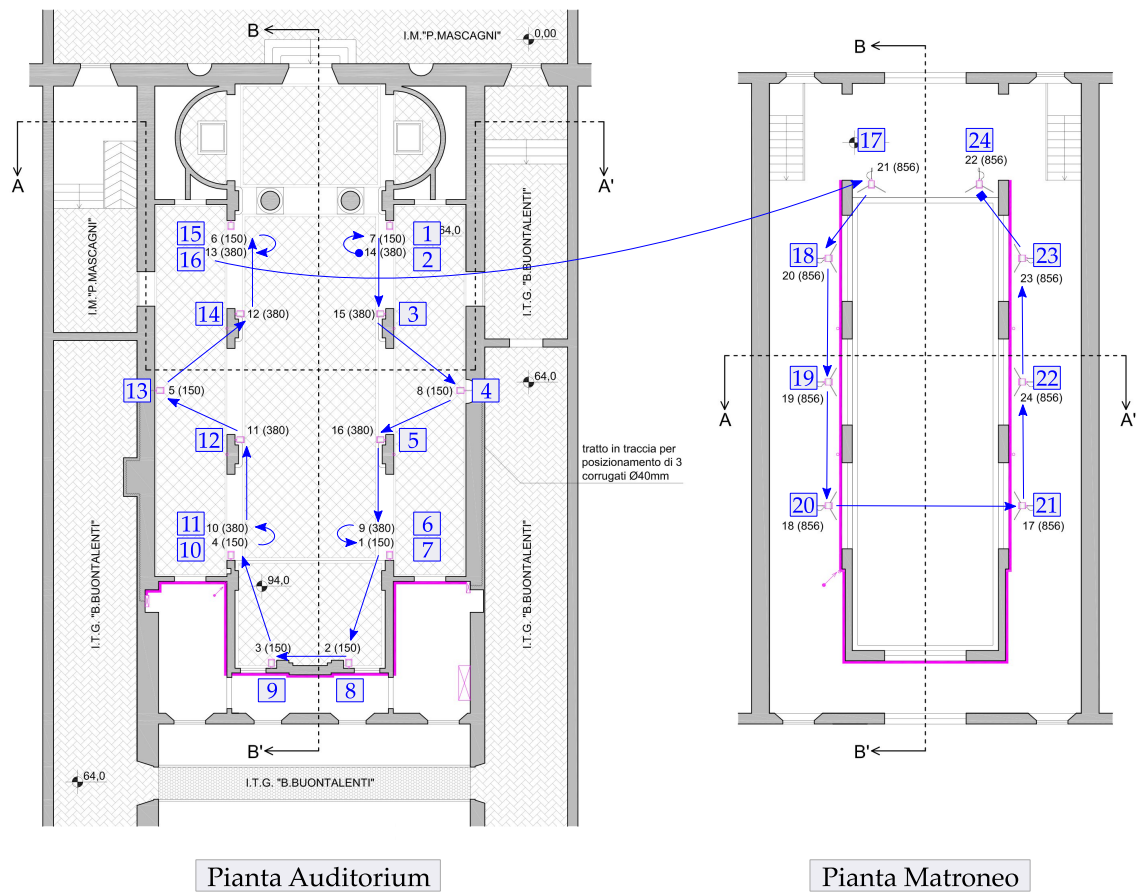


Figure 111: Speakers setup for *Microcontrapunctus* at Istituto Superiore de Studi Musicali Pietro Mascagni in Livorno (Italia). The numbers in blue represent the channel assignment in the multitrack audio of the piece. The arrows indicate the path of the sound as it moves from audio channels 1 to 24, starting from speaker 14 and ending at 22, on the second floor of the hall, according to the hall internal channels assignment. The height in centimeters at which each speaker is located is indicated in parentheses.

## Methods

The piece was produced to be projected using an ideal quantity of 24 speakers used as individual instruments. Consequently, each sound event is assigned a single channel, with each speaker treated as an instrument in itself, possessing its own acoustic qualities and its particular score. Among the many possible approaches to spatialization, I opted for

a treatment of sound diffusion sometimes described as *orchestra of speakers*,<sup>75</sup> particularly suitable when there are so many points of emission.

All other technical and artistic decisions derive from this initial concept focused on echolocation and the resonances of the acoustic space as primary parameters of interest. The first step was to build a sound synthesis engine suitable for harnessing the possibilities of metaprogramming that were to be explored.

I first present the Csound instrument employed as the sole synthesizer throughout the piece, illustrating how each of its components integrates. Later, I will detail how the scores were generated, starting from a small function library that constructed the genotypes.

Given that the purpose of the piece was to test the metaprogramming capability in achieving a varied and vibrant palette of timbral colors through the combination of swarms of these small impulses, the search space was largely limited in two fundamental aspects:

- **Timbre:** after several previous experiments programming Csound instruments aimed at producing small sound grains, the sound synthesis source was reduced to a single, straightforward virtual instrument capable of considerable timbral flexibility.
- **Compositional procedures:** the function library used to construct the scores was assembled with a minimal number of generic methods, emphasizing the possibilities of recombination and recursion of these simple methods rather than relying on complex procedures.

The type of synthesis used in *Microcontrapunctus* is closely related to granular synthesis. The brief sound impulses used as typical events have durations ranging from less than a millisecond to about half a second.<sup>76</sup> To focus the work on the variation capacity of the metaprogramming algorithm, all sounds in the piece originate, without exception, from a single virtual instrument.

Csound allows for precise control of sound synthesis. However, a drawback often arises: to achieve synthetically interesting sounds, a large volume of data must be managed within the scores executed by these instruments. This obstacle is what I aim to overcome with the computational power of metaprogramming.

---

<sup>75</sup>Arranz [11] has conducted an extensive study on these paradigms of real and virtual sound spaces.

<sup>76</sup>A seminal text to learn more about the microtemporal scale of sound in electroacoustic composition is *Microsound* by Roads. [122]

Listing 97 displays the only Csound instrument for *Microcontrapunctus*, which requires thirteen additional parameters beyond those indicating the instrument number.<sup>77</sup>

```
1  instr 1
2
3  ifreq = p4           ; fundamental frequency for the buzz generator (Hz)
4  imodfreq = p5       ; modulation frequency for ring modulation (Hz)
5  iringpresence = p6  ; amount of ring modulation in the final mix (from 0.0 to 1.0)
6  iphasestart = p7    ; starting point of phase for amplitude envelope (1 = 2pi radians)
7  iphaseend = p8      ; ending point of phase for amplitude envelope
8  ipow = p9 * 2 + 1   ; power for envelope modulation (converted to an odd number)
9  inoisepow = p10 * 2 + 1 ; power for noise envelope modulation for attack
10 iharmonicsstart = p11 ; total harmonics for the buzz opcode at the beginning of the event
11 iharmonicsend = p12 + 1 ; total harmonics at the end of the event
12 iseed = p13         ; initialization for random value generation
13 ichannel = floor(p14) ; audio channel assigned to the sound event
14
15 ; buzz synth
16 kharmonics line iharmonicsstart, p3, iharmonicsend
17 ifn = 1
18 aenv line $M_PI * 2 * iphasestart, p3, $M_PI * 2 * iphaseend
19 asin sin aenv
20 asin pow asin, ipow
21 abuzz buzz asin, ifreq, kharmonics, ifn
22 ; attack noise
23 arand rand abuzz, iseed
24 arandenv line 1, p3, 0.0001
25 arandenv pow arandenv, inoisepow
26 aoscilenv = 1 - arandenv
27 ; attack and amplitude envelopes
28 amainsignal = (abuzz * aoscilenv + arand * arandenv)
29 ; additional ring modulation
30 amod poscil 1, imodfreq
31 aring = (abuzz * aoscilenv*amod + arand * arandenv * amod)
32 ; final mix and output channel assignment
33 amix = amainsignal * (1 - iringpresence) + aring * iringpresence
34 outrg ichannel, amix
35
36 endin
```

Listing 97: *Microcontrapunctus* — Csound instrument to synthesize microsounds

<sup>77</sup>In the syntax of Csound scores, the first three parameters are fixed: p1 is the instrument index (here always i1), p2 is the start time in seconds, and p3 is the duration of each event. The parameters required from p4 onwards are defined by the user in the instrument's code.

The instrument combines classic additive synthesis with ring modulation and the use of white noise. At the core of the instrument, in line 20, we find the additive synthesis generator `buzz`<sup>78</sup>. It groups a set of harmonic partials by summing sinusoidal waves, based on four parameters:

`asin` — amplitude envelope.

`ifreqenvelope` — fundamental frequency of the partials set.

`kharmonics` — number of harmonics added from the fundamental frequency.

`ifn` — index of the function used for synthesis (in this case, a normal sine function).

---

<sup>78</sup>Documentation on `buzz` can be found at <http://www.csounds.com/manual/html/buzz.html>. For a deeper understanding of the implementation of this type of synthesis, see the article by Stilson and Smith. [145]

```

; p4 -> fundamental frequency for the buzz generator (Hz)
; p5 -> modulation frequency for ring modulation (Hz)
; p6 -> amount of ring modulation in the final mix (from 0.0 to 1.0)
; p7 -> starting point of phase for amplitude envelope (1 = 2pi radians)
; p8 -> ending point of phase for amplitude envelope
; p9 -> power for envelope modulation (converted to an odd number)
; p10 -> power for noise envelope modulation for attack
; p11 -> total harmonics for the buzz opcode at the beginning of the event
; p12 -> total harmonics at the end of the event
; p13 -> initialization for random value generation
; p14 -> audio channel assigned to the sound event

```

```

;p1 p2 p3 p4 p5 p6 p7 p8 p9 p10 p11 p12 p13 p14
i1 0 .1 500 1 0 0 .5 1 5000 1 10 0 1

```

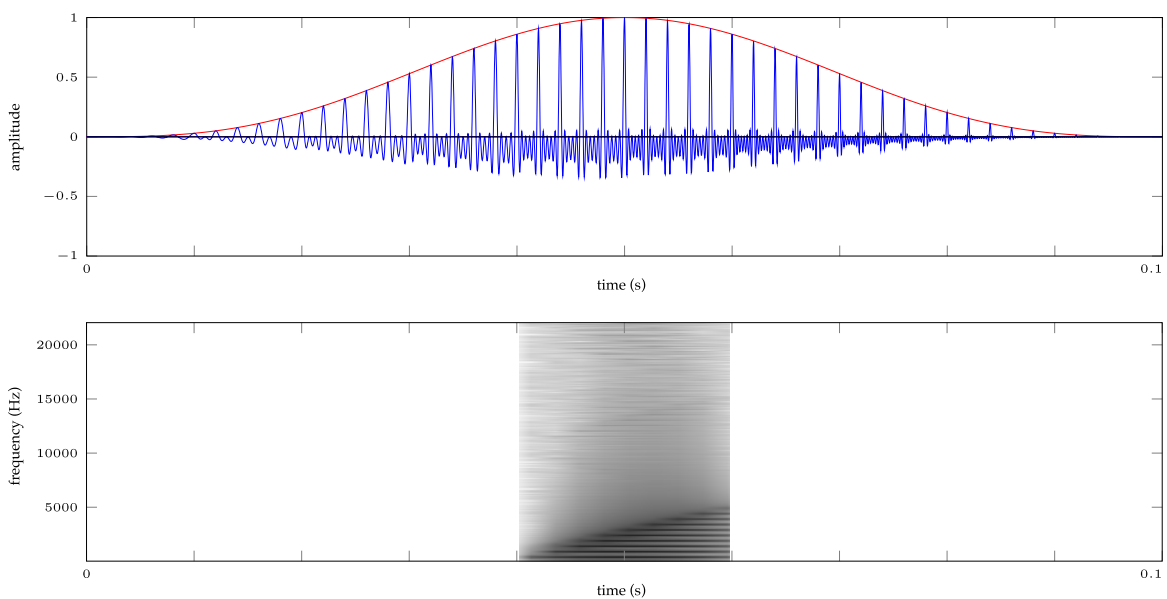


Figure 112: Basic sound created with the Csound instrument for Microcontrapunctus. The **buzz** opcode acts as a dynamic battery of simple oscillators. Therefore, if the *kharmonics* parameter ranges from 1 to 10, the first ten partials of a harmonic series are successively added. In the upper part of Figure 112, the audio signal generated in blue is modulated with the *asin* amplitude envelope, also created using the sine function and marked with the red line. The spectral analysis in the lower part shows the successive entry of the ten partials starting from the 500 Hz fundamental. The top of the figures contains the Csound score that has synthesized it.

The signal produced by **buzz** is modulated by several additional signals that control the attack timbre, the overall envelope, and a ring modulation with an additional sinusoidal wave that dynamically enriches the timbre with series of inharmonics. By modifying this initial example, I will separately demonstrate the effect that each of these modulations has on the signal.

The importance of the attack in the identification and character of a sound event is well known. In notes as brief as those in this work, the attack time can be extremely short. The asin signal is controlled by two parameters that operate on the phase of the sine function that generates it, normalized with values from 0 to 1. With a phase start of 0, as in the initial example, the signal gradually grows. A value of 0.25 creates a sharp attack. Figure 113 shows three variations of the previous note.

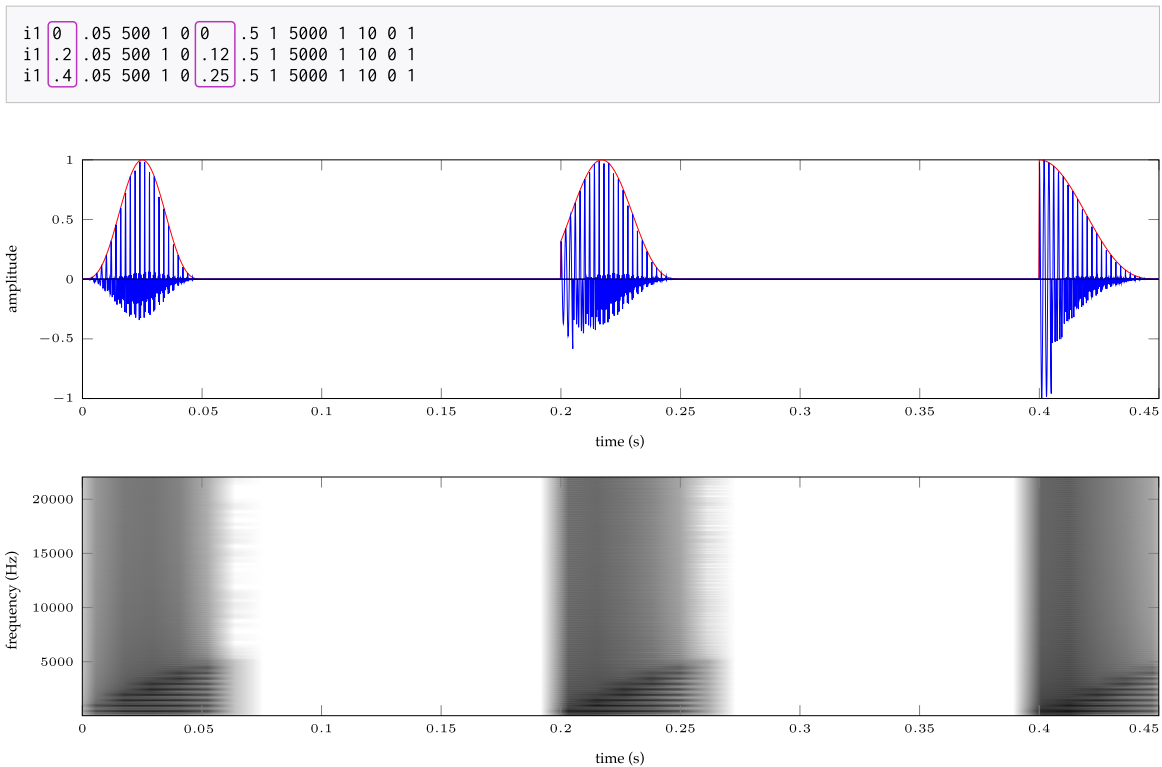


Figure 113: Modifications of the attack. The sound of the previous figure with half the duration (0.05 s) and with increasingly rough attack envelopes. The spectral analysis shows how, in addition to multiples of 500 Hz, at the beginning of each note, there is progressively more energy across all frequencies.

The instrument design allows the asin envelope to complete several cycles of the sine function. Figure 114 shows that, in practice, this can function as a tremolo effect with a low-frequency oscillator (LFO), or it can even produce a classic ring modulation, whose effects are well-known as a resource to enrich the spectral content.



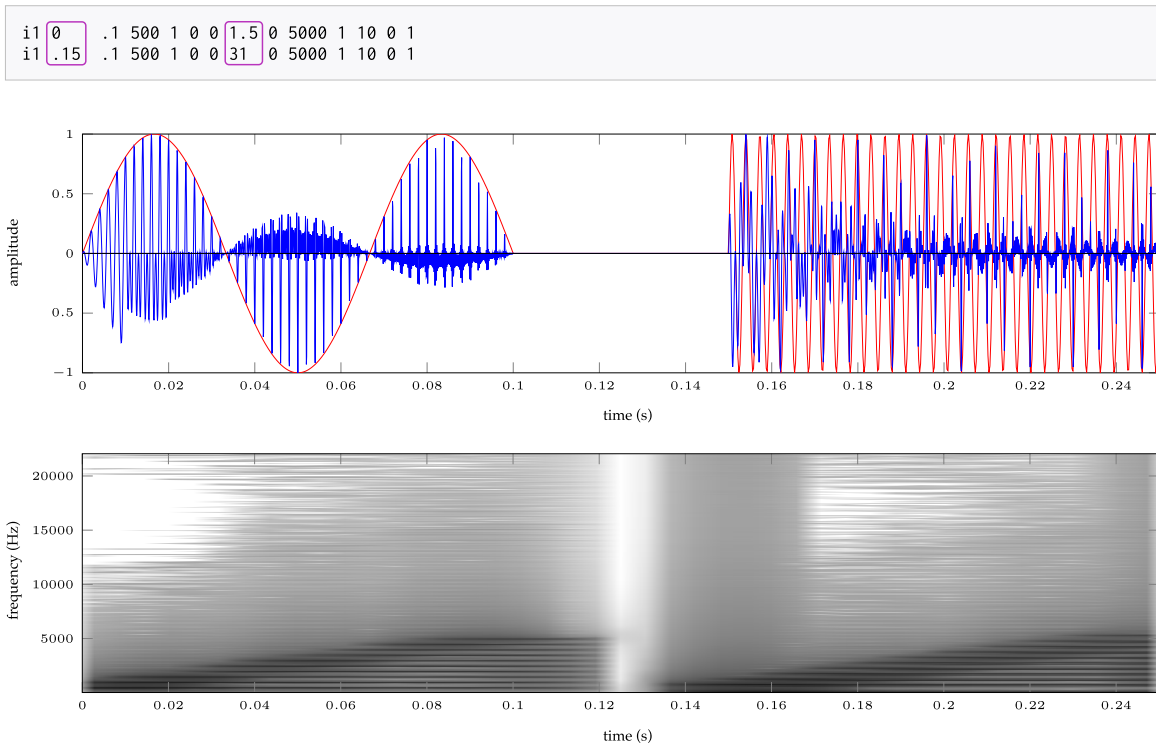


Figure 114: Amplitude envelopes creating two variations of the initial sound: the first with an amplitude envelope (in red) acting as an LFO, and the second as ring modulation, adding inharmonic partials to the harmonic series of 500 Hz.

A third element that defines this envelope is the constant `ipow`, which is the exponent of a power calculated on the values of `asin`. The higher this value, the more angular the envelope becomes, and the more abrupt the attack and overall amplitude evolution, as can be seen in Figure 115.

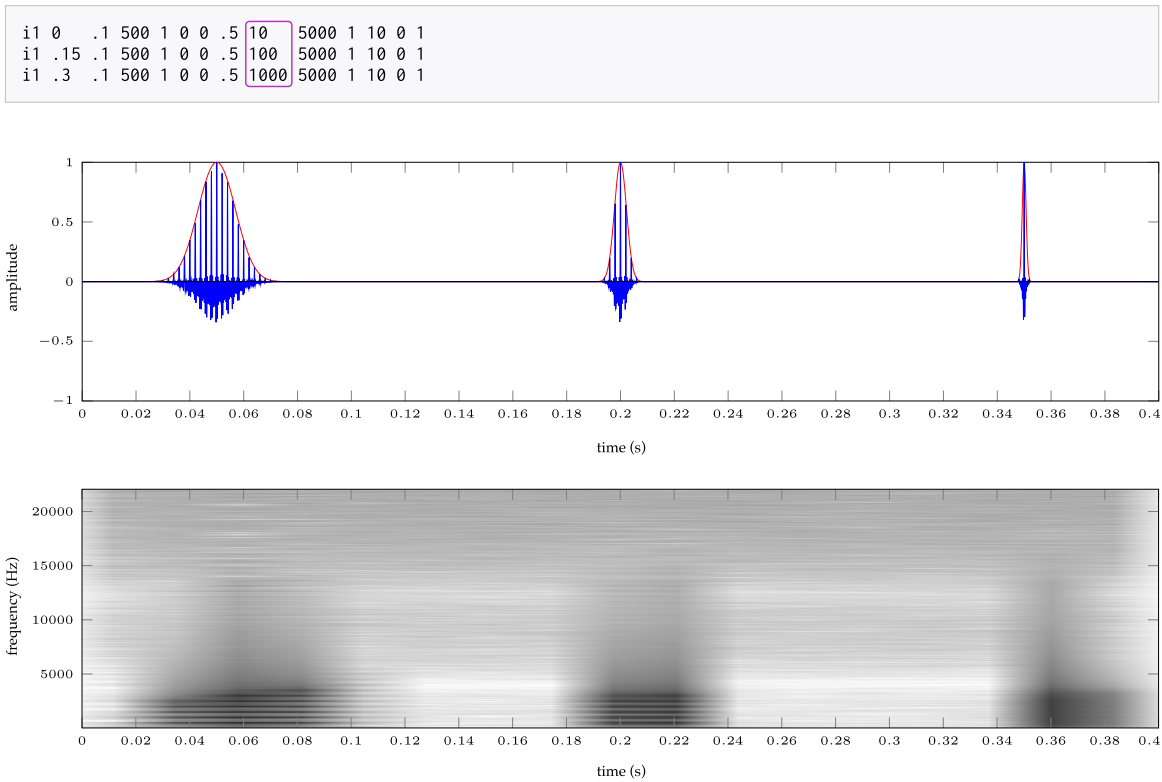


Figure 115: Application of different exponents to the envelope curve of the note in Figure 112. Increasing exponents gradually transform the note into an impulse; the harmonic partials become less perceptible, revealing the spectrogram clearly.

As a complementary component to the attack, a new layer of noise is added, multiplied by the main signal abuzz, illustrated in Figure 116. This initial instability of the signal is a common characteristic in many physical sounds; at times, it lends a synthetic sound a veneer of credibility and analogy to recognizable real sounds.

```
i1 0 .1 500 1 0 .25 .5 1 0 1 10 0 1
i1 .15 .1 500 1 0 .25 .5 1 10 1 10 0 1
```

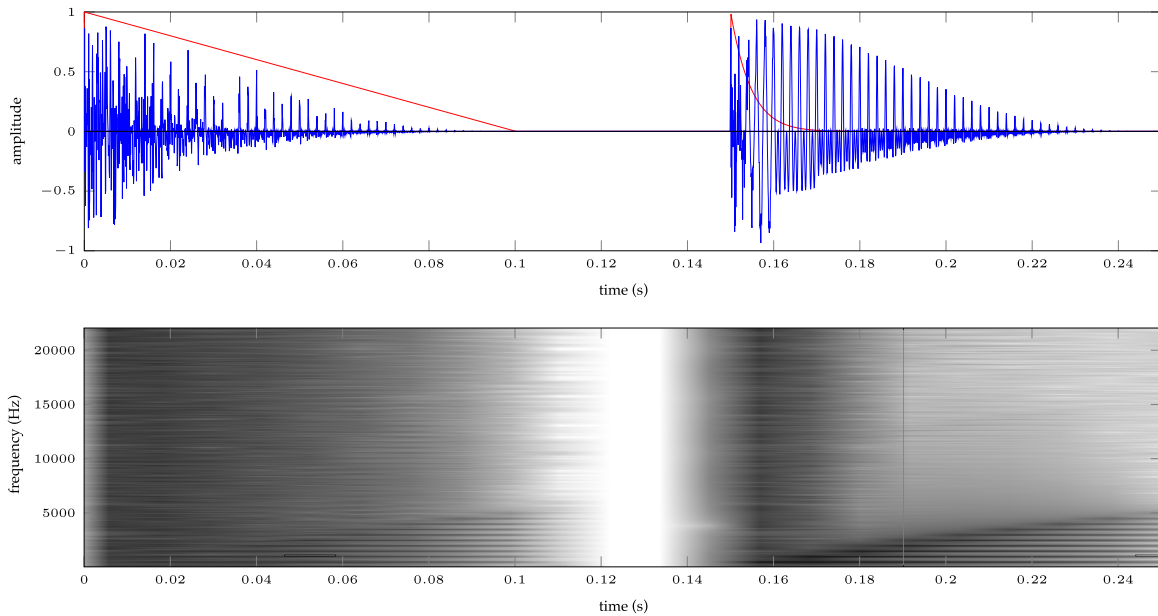


Figure 116: Addition of a modulating noise source. The degree of affectation to the initial layer is modulated by the signal *arandenv*, delineated in red, which in turn depends on another exponent in the variable *inoisepow* controlled by the parameter *p10*.

Figure 117 represents the final layer, which is a new ring modulation added from the simple oscillation of a single harmonic. The constant *iringpresence* determines how much weight this modulation has on the final signal.

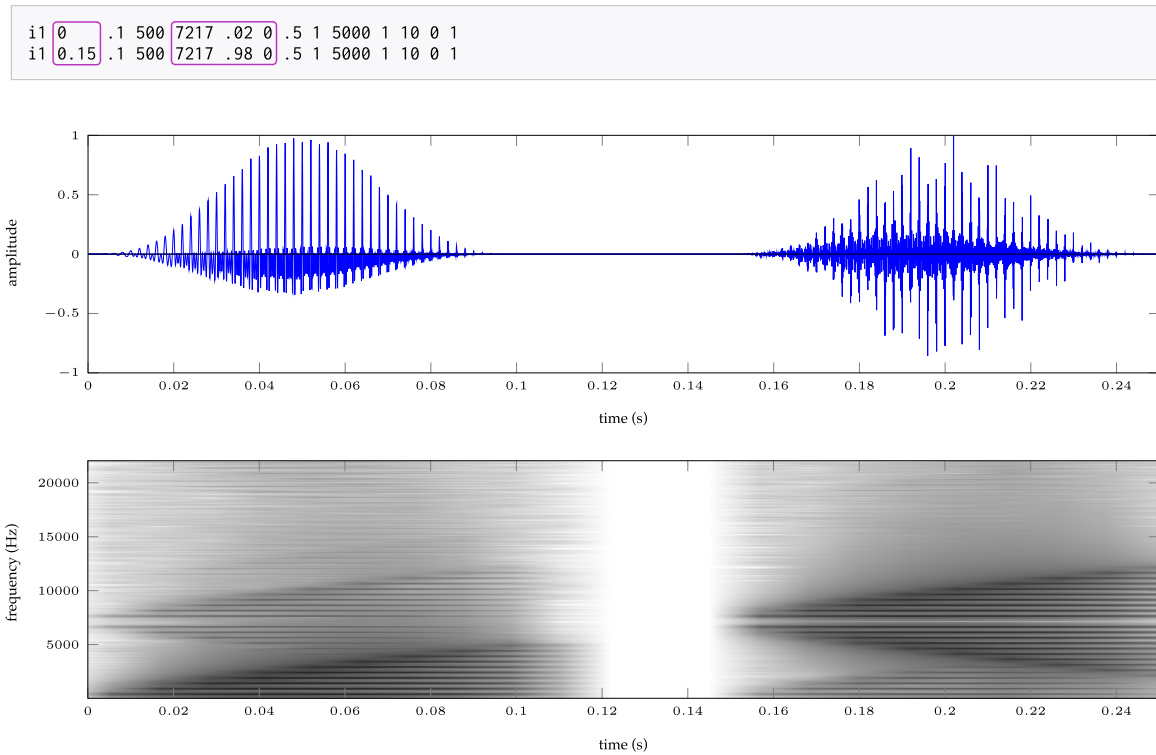


Figure 117: Additional ring modulation. Application of a modulating signal of 7217 Hz to the sound of the initial example. In these two variations, the presence of the ring modulation is 2% and 98%, respectively.

For clarity, the previous examples display each component separately in notes of a relatively long duration compared to the events used in *Microcontrapunctus*. Figure 118 shows a note that integrates all the previous modulations with a duration of 0.02 seconds. The spectral analysis still displays traces of harmonic content that are no longer perceptible as pitches but rather as the tonal quality of a sound particle barely longer than an impulse.

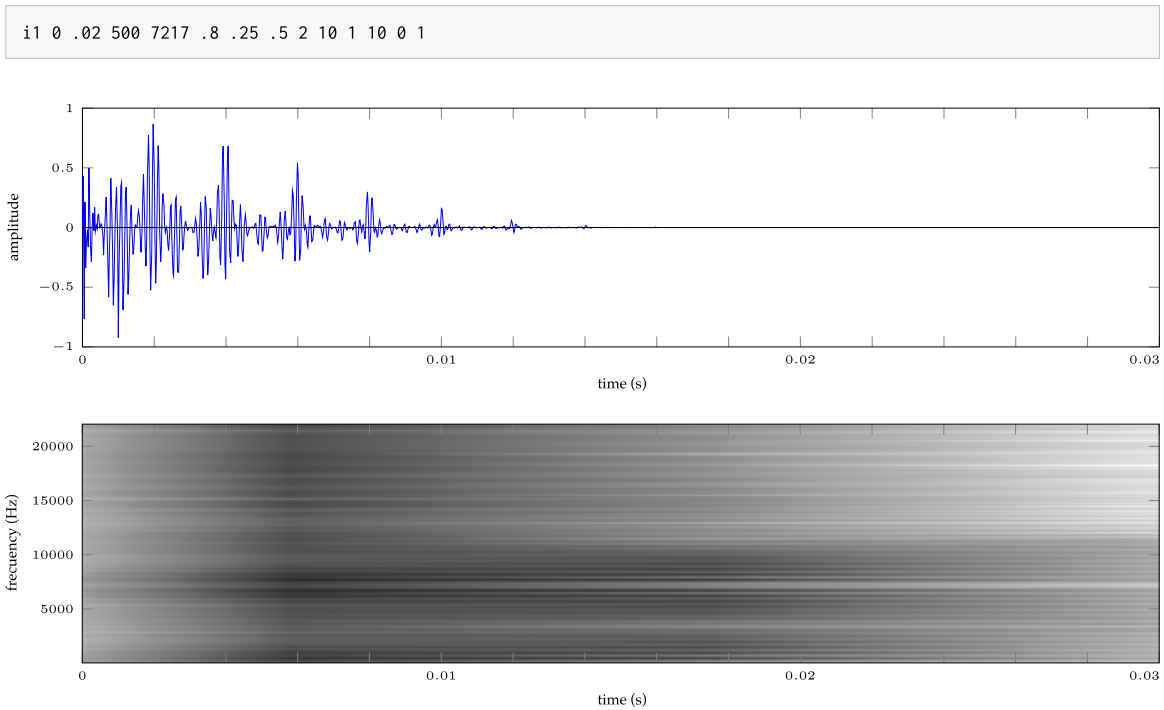


Figure 118: Single sound grain with different modulations. The spectral analysis cannot be very precise due to the brevity of the sound particle. This type of very brief event is predominant in the overall texture of the composition.

Finally, to get even closer to the type of sound character on which the piece is based, Figure 119 displays a group of notes that, starting from the previous event, gradually modifies all parameters, including the last two: the penultimate one, `iseed`, which is an initializer for the random values that prevents the repetition of pseudo-random series generating white noise, and `ichannel`, which indicates the channel to which each event is assigned.

```

;p1 p2 p3 p4 p5 p6 p7 p8 p9 p10 p11 p12 p13 p14
il 0 .05 .05 20 7217 .5 .25 .5 2 10 1 10 0 1
il .05 .04 < < < < < < < < < < < 1
il .1 .036 < < < < < < < < < < < 2
il .16 .023 < < < < < < < < < < < 2
il .23 .014 < < < < < < < < < < < 3
il .29 .012 < < < < < < < < < < < 4
il .36 .009 < < < < < < < < < < < 3
il .45 .008 < < < < < < < < < < < 2
il .52 .007 < < < < < < < < < < < 2
il .69 .006 < < < < < < < < < < < 1
il .9 .004 13000 18000 1 0 .75 100 0 40 15 1 2
e

```

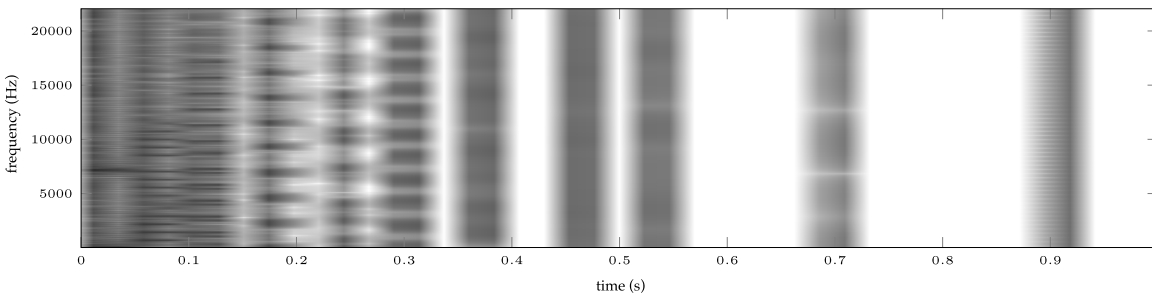
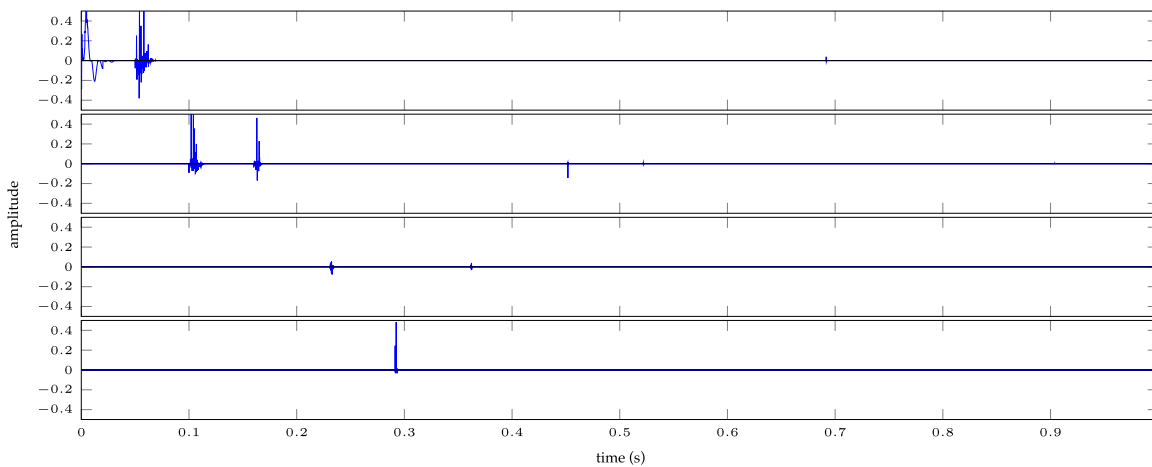


Figure 119: Csound score and sequence of various microsounds. This synthesis example uses only four channels in p14 to determine the output bus. For the final composition, it takes values from 1 to 24. The spectral analysis combines all channels into the same spectrogram. The last sounds are so brief that they are only visible in the spectrogram. In the Csound score, the symbol < creates a linear transition between the upper and lower values in the column.

| Function name              | Output type     | Description  |
|----------------------------|-----------------|--|
| <code>combineArrays</code> | score           | Returns a score that combines 13 independent arrays (one for each parameter of the Csound instrument). As the arrays it takes can have different sizes, the number of events in the score adjusts to the shortest array. |
| <code>concatScores</code>  | score           | Creates a group of events by concatenating two scores given as parameters.   |
| <code>gestureCurve</code>  | parameters list | Constructs a curve with four intermediate inflection points, and according to an exponential value for the development of each of the five segments of the curve.  |
| <code>jitter</code>        | parameters list | Applies stochastic deformation to each of the values in an input sequence, according to a parameter indicating the range of that deformation.  |
| <code>jitterScore</code>   | score           | Applies stochastic deformation to a score in the same way that <code>jitter</code> did to an array.  |
| <code>loopArray</code>     | parameters list | Returns a sequence formed by repeating the input sequence a number of times.   |
| <code>permutArray</code>   | parameters list | Takes a sequence and performs permutations on the order of its values.   |
| <code>permutScore</code>   | score           | Analogous to the <code>permutArray</code> function, this function does the same with an event sequence, applying stochastic permutations to each of the arrays contained in a score.                                     |
| <code>remapArray</code>    | parameters list | Takes a series of values and remaps them according to a new minimum and maximum, and an exponent that applies an exponential curve.  |
| <code>steps</code>         | parameters list | Builds a linear progression in several steps from an initial value to a final one.   |
| <code>stepsScore</code>    | score           | Constructs an event sequence by generating linear arrays for each parameter.   |

Table 22: Microcontrapunctus' genotype functions. Since this project was a proof of concept, a very limited number of functions were implemented for genotype construction. However, the musical results were rich enough to construct the composition.

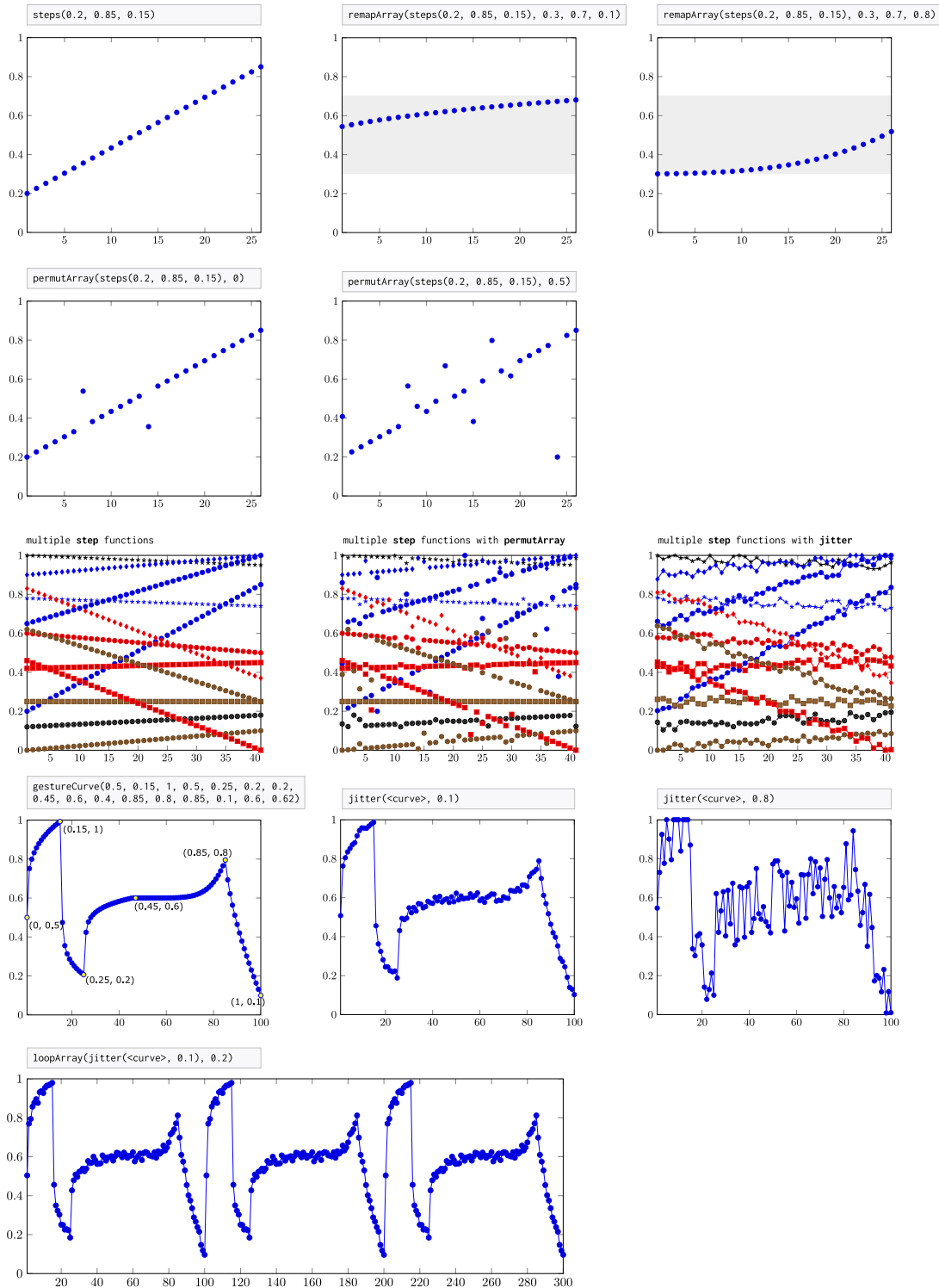


Figure 120: Data generated by Microcontrapunctus' genotype functions of type list



The amount of text required to code just one second of sound is considerable. This is where metaprogramming with GenoMus comes into play as a tool to create symbolic metalevels capable of generating thousands of these small events from a comprehensive and descriptive high-level grammar. In Table 22 the genotype functions that this version of GenoMus handled for building the phenotypes from which the entirety of the sound synthesis of the piece was produced.

In this experiment, there are only functions of two types based on their output: those that generate or transform lists of parameters, and those that form scores that will eventually be converted into sequences of events in Csound. Figure 120 illustrates the operations performed by the functions in the first group.

In this project, a primitive version of the encoded genotype is introduced, albeit lacking many features of the current model. Below is an example that generates a small sequence of sounds. Figure 121 already displays a reduced example of the complete flow from the generation of a randomly encoded genotype to the final synthesized sound.

encoded genotype

```
[ 13, 17, 13, 17, 21, 12, 2, 0.066, 2, 0.009, 2, 0.398, 2, 0.931, 2, 0.507, 2, 0.507, 2, 0.038, ... 168 more items ]
```

decoded genotype

```
permutScore(concatScores(
  permutScore(
    concatScores(
      jitterScore(
        stepsScore(
          0.06626327212062832, 0.009959305358693715, 0.3975889620947618, ... 26 more items),
          0.6262773750252124),
        stepsScore(
          0.34781127570154424, 0.7461084774898173, 0.6410042580732761, ... 26 more items)),
        0.029288705969629514),
      stepsScore(
        0.6199538246165506, 0.5087386197977247, 0.8265339087512206, ... 26 more items)),
      0.9569573079326134)
```

encoded phenotype

```
[ 0.041323310761538, 0.2890827973892515, 0.3791474629545431, 0.043880651938419335, ... 3506 more items ]
```

decoded phenotype

```
;p1 p2 p3 p4 p5 p6 p7 p8 p9 p10 p11 p12 p13 p14
i1 0.0000 0.0976 10122.8460 2286.6087 0.0011 0.2021 5.0000 0.0000 2.8106 1.1616 1.0000 0.3846 2.5301
i1 0.0001 0.1542 7500.8279 13413.7447 0.3453 0.1833 2.5223 0.0073 2.4873 3.1179 1.1081 0.0010 2.1233
i1 0.0002 0.1775 9566.1978 1432.8500 0.0010 0.2073 0.5425 0.0000 0.0066 28.079 1.0000 0.8174 1.7647
[ ... 266 more events ]
i1 3.0311 0.0208 19298.5315 6927.3148 0.2718 0.2550 0.8007 0.3901 0.8647 1.0345 6.0762 0.6245 1.1204
```

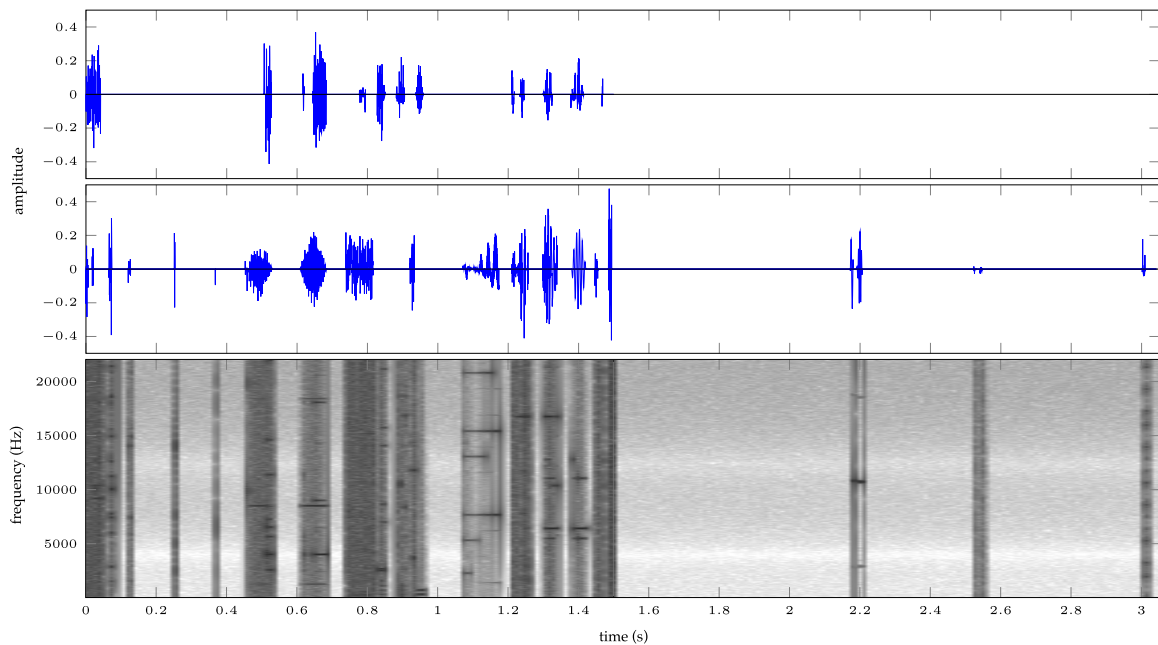


Figure 121: Data flow from the encoded genotype to the microsound sequence

Finally, I present a genotype of the type that generated the sounds integrated into the composition. In Listing 98, the limit of branching depth of the functional tree is seven. The corresponding audio sequence is displayed in Figure 122.

```
concatScores(  
  concatScores(  
    stepsScore(  
      0.6540395879606171, 0.9728245717009265, [ ... 27 more items ]),  
    stepsScore(  
      0.22239203910797967, 0.14671288015170747, [ ... 27 more items ])),  
  concatScores(  
    permutScore(  
      jitterScore(  
        permutScore(  
          stepsScore(  
            0.09403539164890773, 0.9615761914053887, [ ... 27 more items ]),  
            0.9228131948278162),  
            0.05852691014936773),  
            0.9792464395755205),  
        permutScore(  
          combineArrays(  
            remapArray(  
              remapArray(  
                steps(  
                  0.6898430470723919, 0.3178026718069705, 0.7250774804970264),  
                  0.2856020633365356, 0.18892184442333881, 0.8067271131987117),  
                  0.45361192313536014, 0.2745975219583152, 0.0086719702346012),  
                steps(  
                  0.6999633672393484, 0.7529553293430471, 0.12130300364743696),  
                  gestureCurve(  
                    0.30333519779989215, 0.8737315017891157, [ ... 14 more items ]),  
                    permutArray(  
                      permutArray(  
                        gestureCurve(  
                          0.40850730149379844, 0.8736568430313747, [ ... 14 more items ]),  
                          0.7519616413317407),  
                          0.849375264041261),  
                        remapArray(  
                          permutArray(  
                            remapArray(  
                              [0.5094212213500061, 0.3926701689407963, 0.2701658138653338, 0.6013940194469194],  
                              0.828738054337547, 0.9916017320681684, 0.7547844127637742),  
                              0.08418369951459648),  
                              0.6534955465804004, 0.01579824265424623, 0.28027864315430007),  
                            permutArray(  
                              permutArray(  
                                loopArray(  
                                  [0.45008436614797165, 0.464658651175221, 0.43841162663198097, 0.9379684800699739],  
                                  0.8667414484888062),  
                                  0.9513909506410053),  
                                  0.6695437749750031),  
                                gestureCurve(  
                                  0.9485584438515572, 0.8477380481162651, [ ... 14 more items ]),  
                                  steps(  
                                    0.48068848956642485, 0.4065502846653838, 0.21034266920832467),  
                                    steps(  
                                      0.7684912533395216, 0.3961615424042998, 0.006148885875535415),  
                                      remapArray(  
                                        steps(  
                                          0.6136431169684109, 0.21682431647081424, 0.5478260434514265),  
                                          0.09961457771972615, 0.15841868629550226, 0.04075662172665495),  
                                          steps(  
                                            0.4871680119123927, 0.02402244477127835, 0.8246469425787919),  
                                            remapArray(  
                                              jitter(  
                                                remapArray(  
                                                  [0.6522292937294387, 0.05297404161504882, 0.7504096204406197, 0.8058722851477045],  
                                                  0.6647645735811639, 0.9620394819433622, 0.16542848329930093),  
                                                  0.0414964833636281),  
                                                  0.14268542896184022, 0.0724526494387625, 0.1411451247506038),
```

```
remapArray(  
  loopArray(  
    remapArray(  
      [0.9805650058308085, 0.6974184170011729, 0.32846390366276146, 0.8657352742243655],  
      0.3963293030160825, 0.13961838950996475, 0.8694078855826461),  
      0.2938724629924293),  
    0.257491421879254, 0.2356517065542676, 0.48210572692632436)),  
  0.6333066620908956)))
```

Listing 98: Decoded genotype for synthesis of microsounds

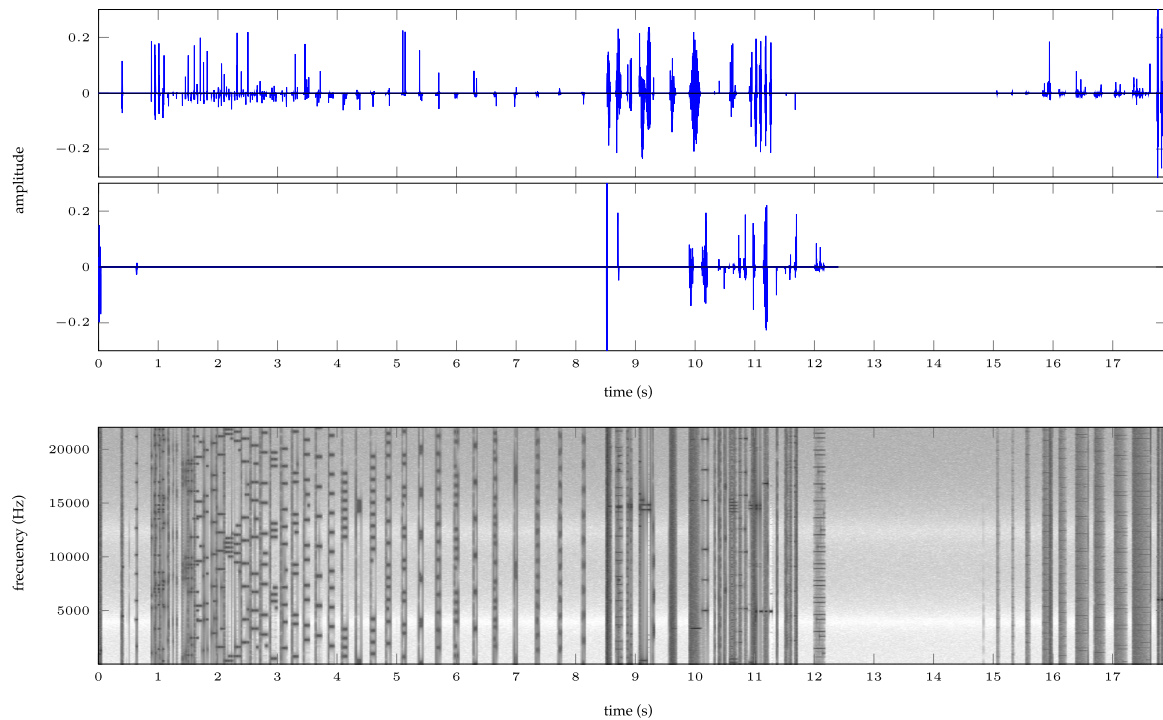


Figure 122: Example of actual fragment of the final composition

Next, I display the graphical representation of various passages from the composition, which was assembled from many fragments of diverse durations generated with GenoMus with minimal modification, aside from material selection and arrangement. In Figures 123 to 138, the waveforms of the 24 audio channels appear first, allowing a clear visualization of shifts in multichannel spatialization. Below are the combined spectra of all signals, with a vertical range capped at 20 kHz. The entire piece can be listened to (and visualized) at <https://vimeo.com/lopezmontes/microcontrapunctus>.

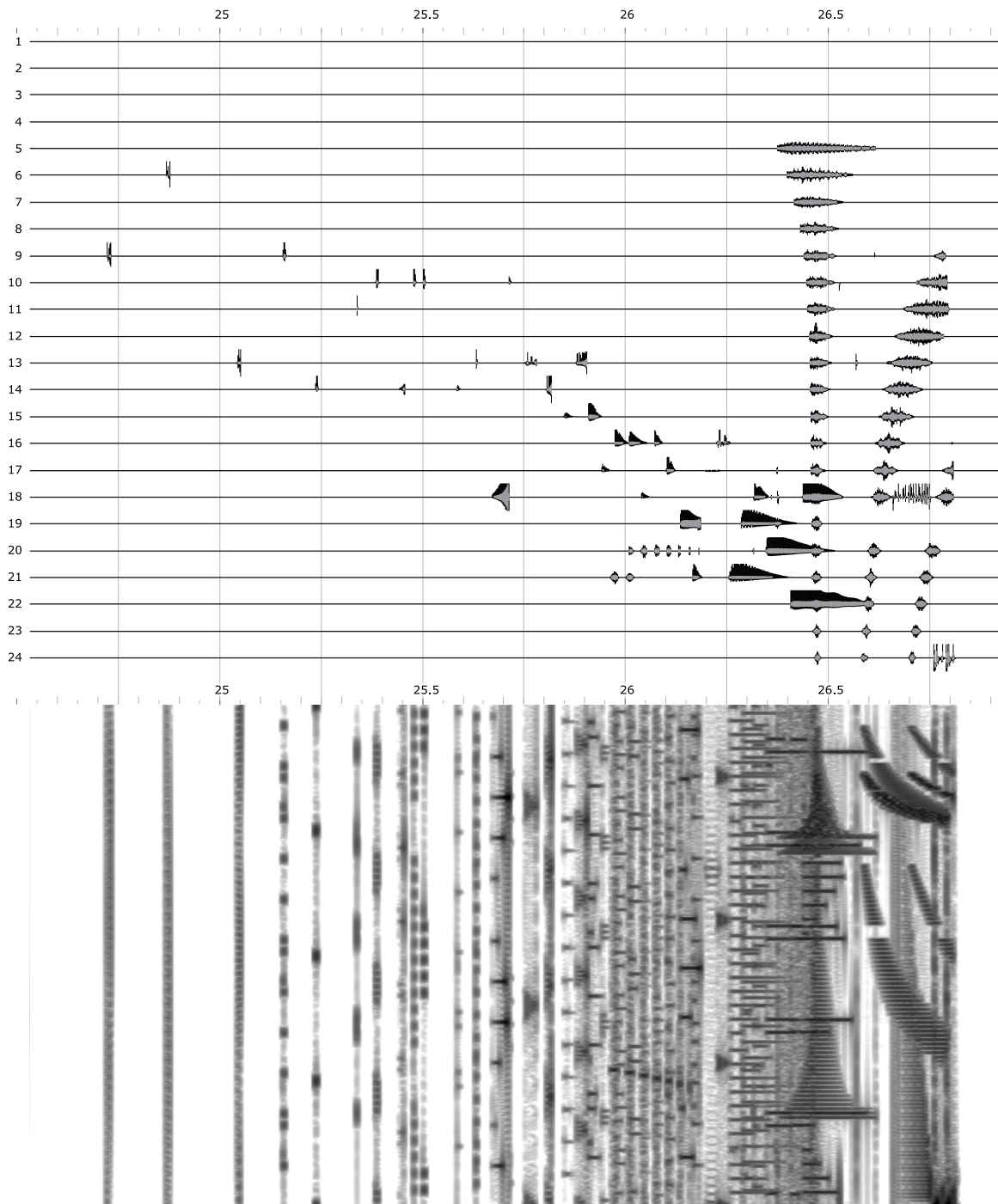


Figure 123: Microcontrapunctus — waveforms and spectrogram 0:24.5 - 0:27.0

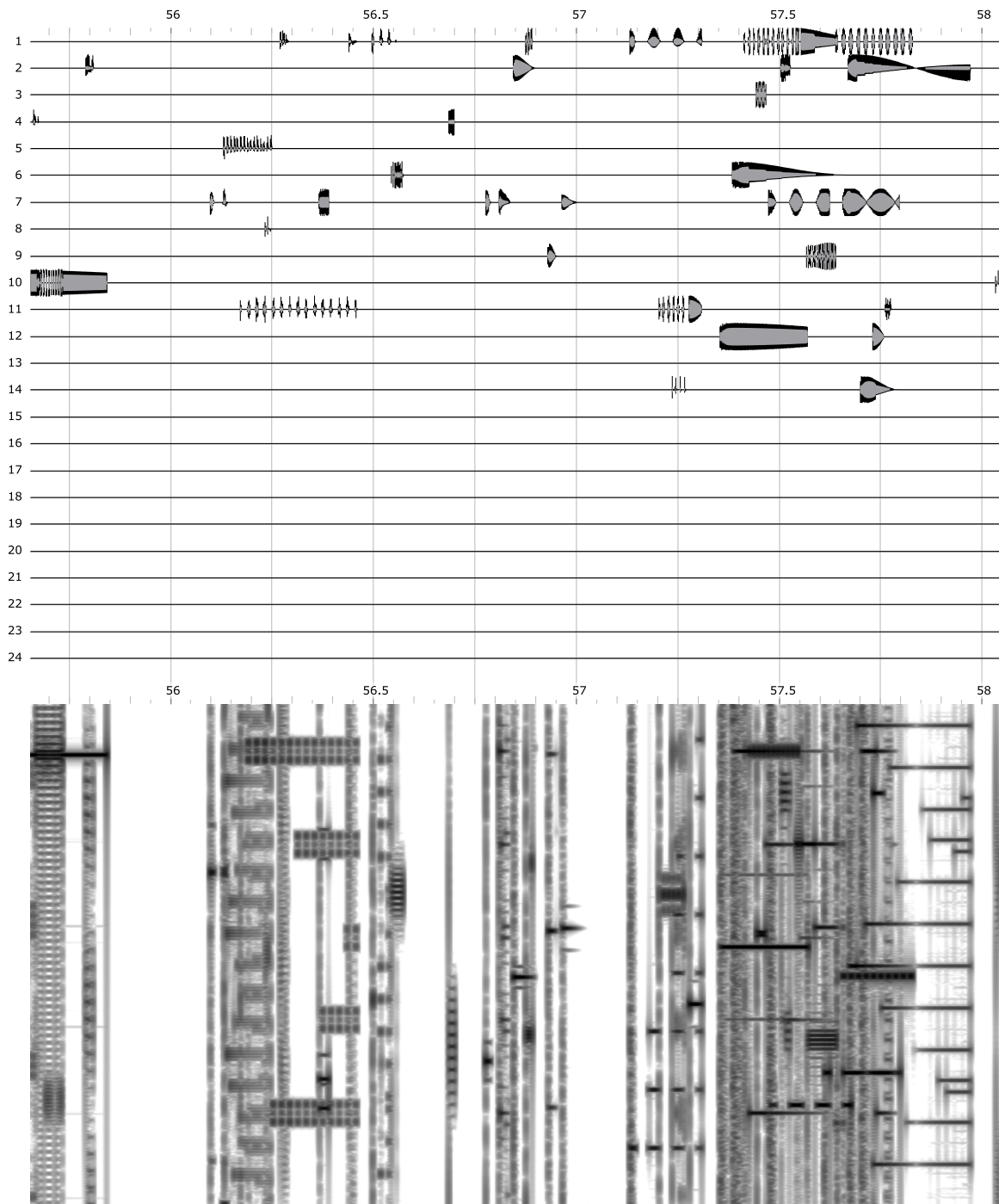


Figure 124: Microcontrapunctus — waveforms and spectrogram 0:55.3 - 0:58.1

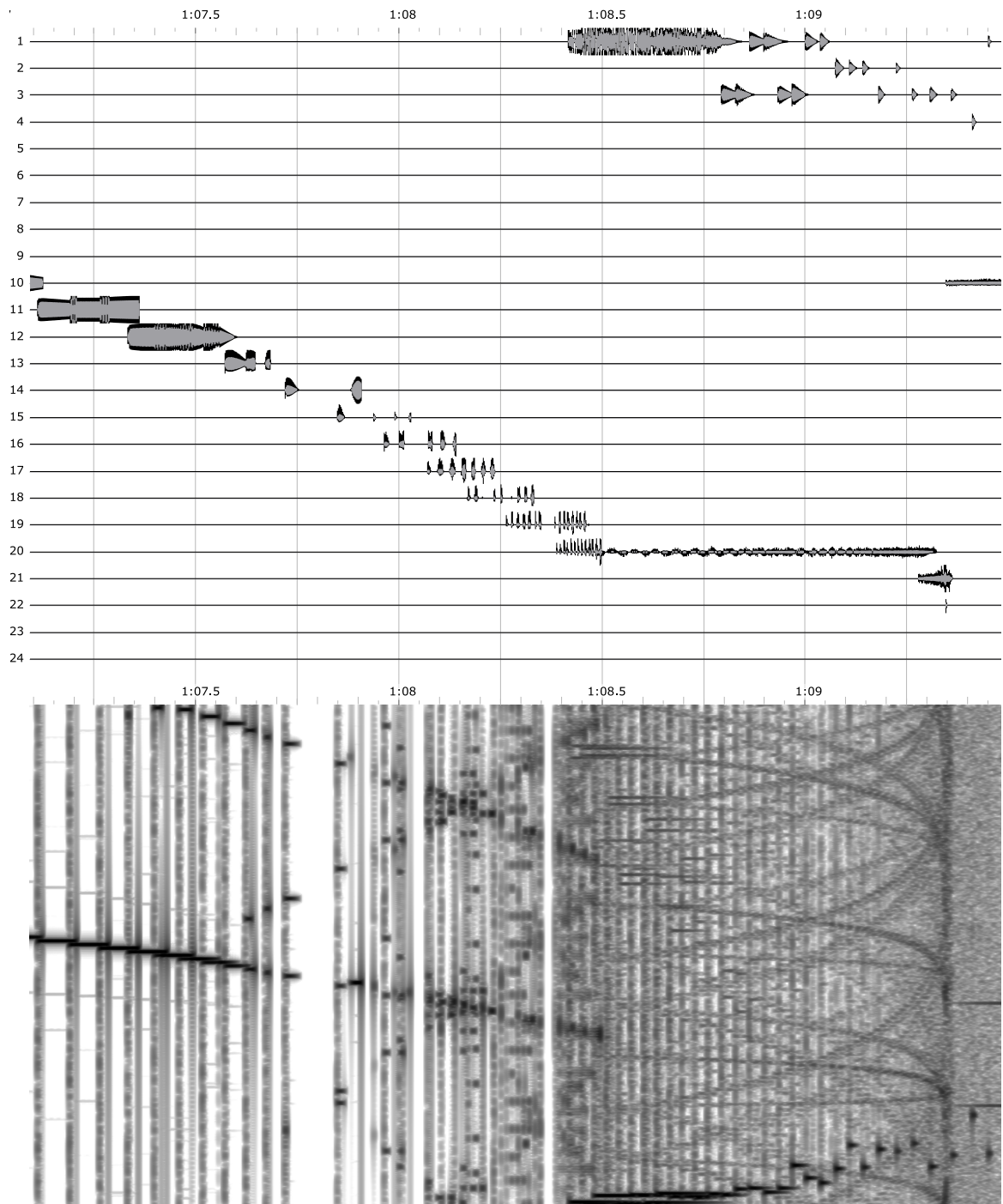


Figure 125: *Microcontrapunctus* — waveforms and spectrogram 1:07.1 - 1:09.9

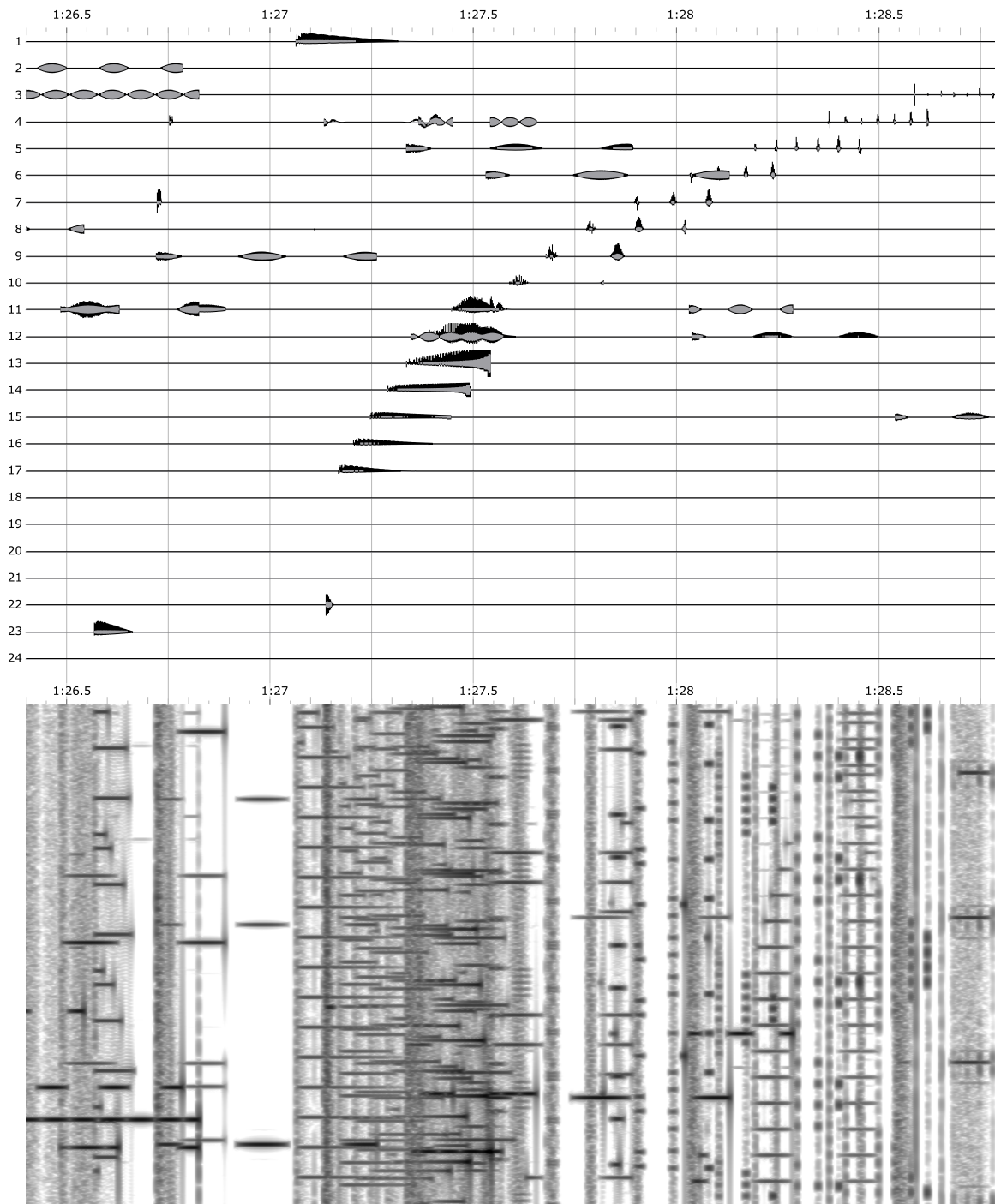


Figure 126: Microcontrapunctus — *waveforms and spectrogram 1:26.3 - 1:29.1*



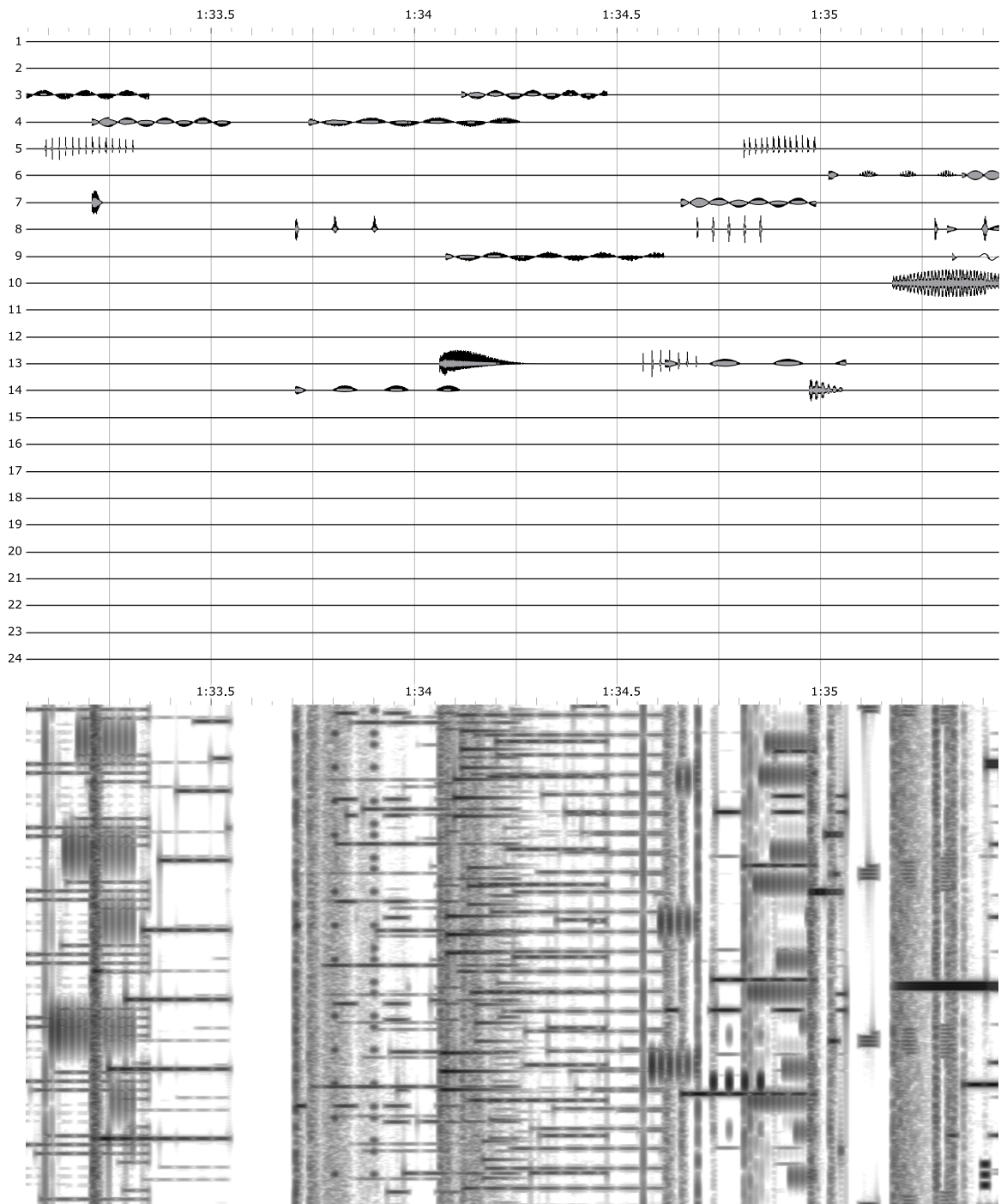


Figure 127: *Microcontrapunctus* — waveforms and spectrogram 1:32.6 - 1:35.9

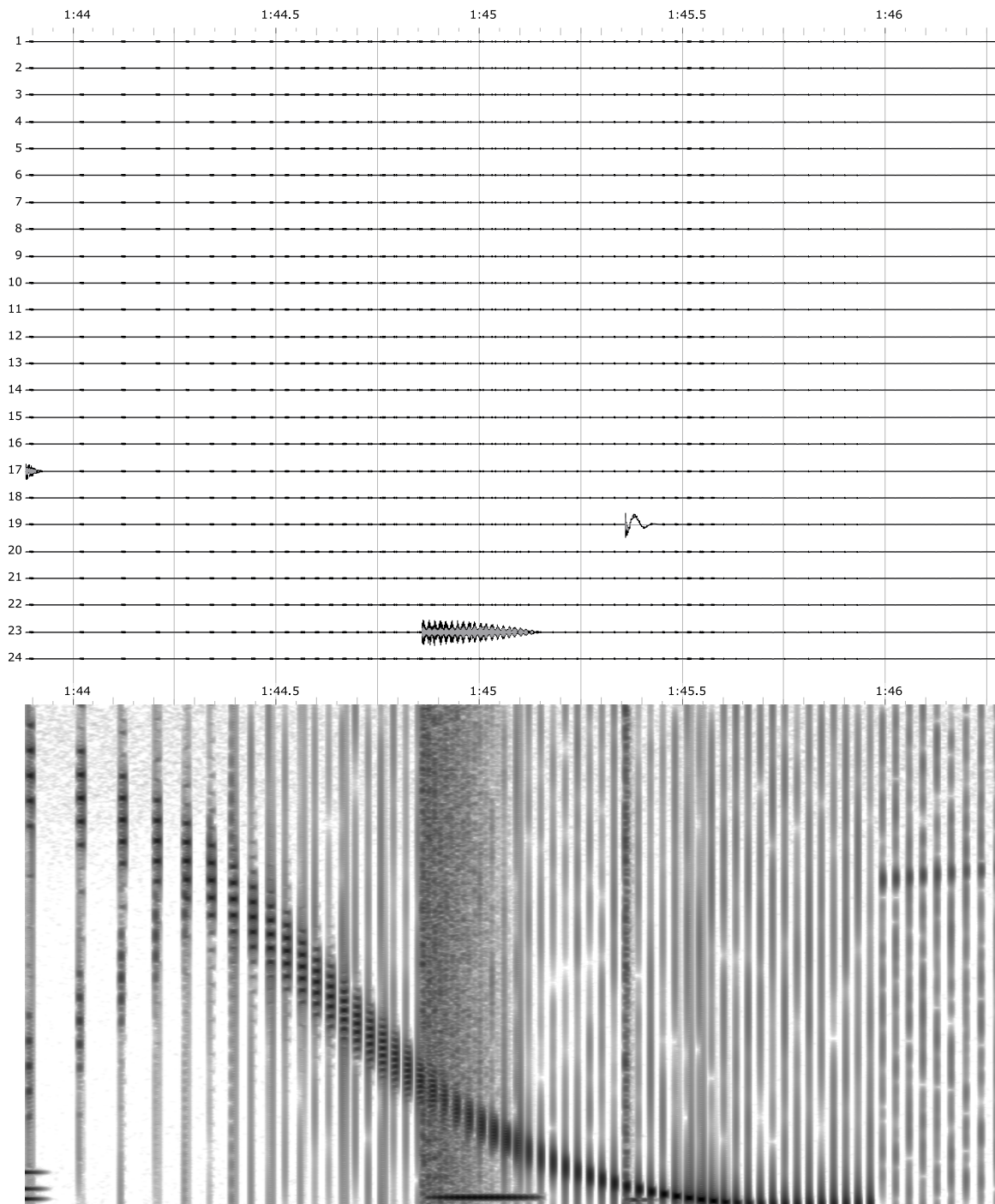


Figure 128: Microcontrapunctus — *waveforms and spectrogram* 1:43.8 - 1:47.0

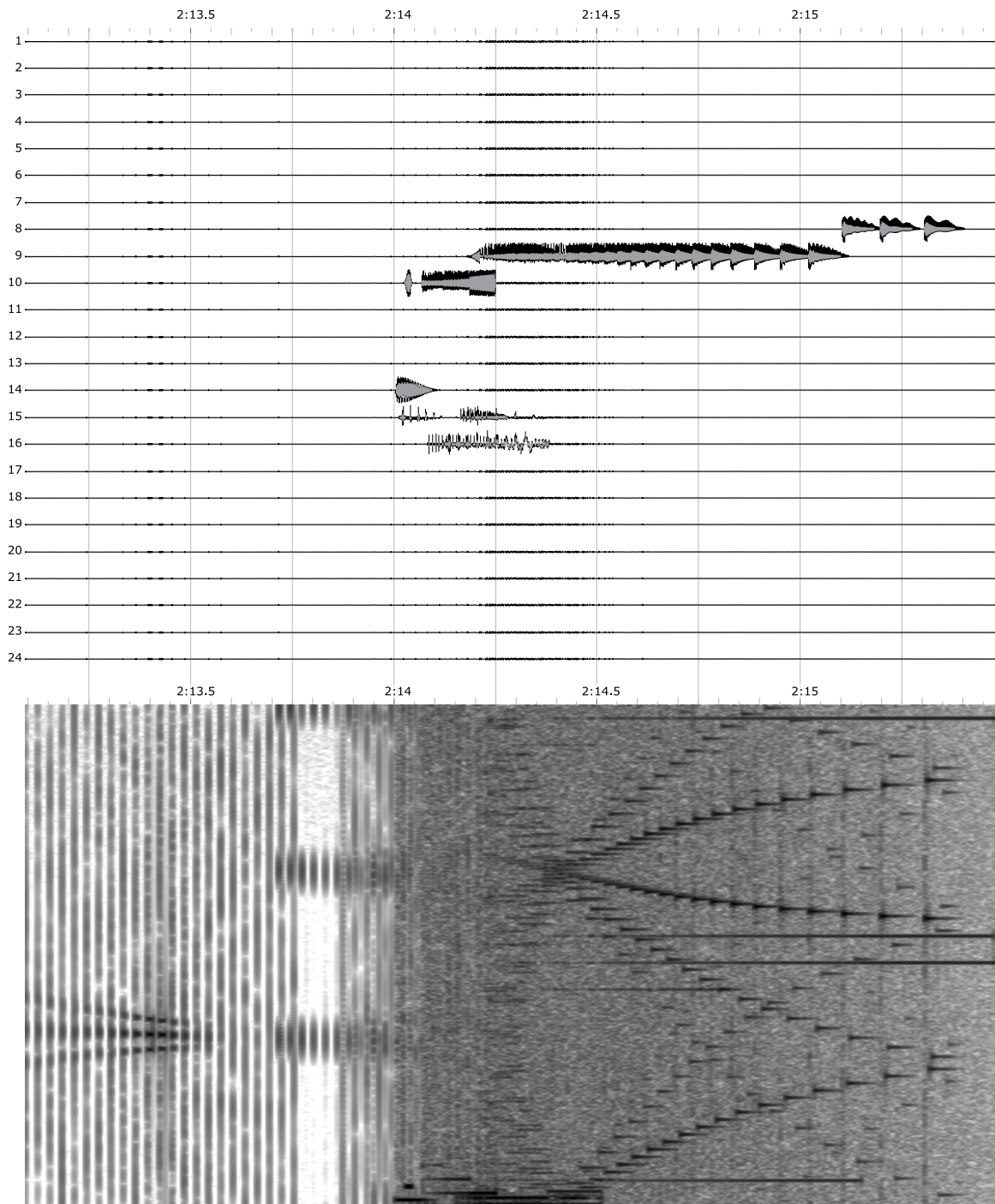


Figure 129: *Microcontrapunctus* — waveforms and spectrogram 2:12.7 - 2:16.0

Musical works  
B.3. *Microcontrapunctus*

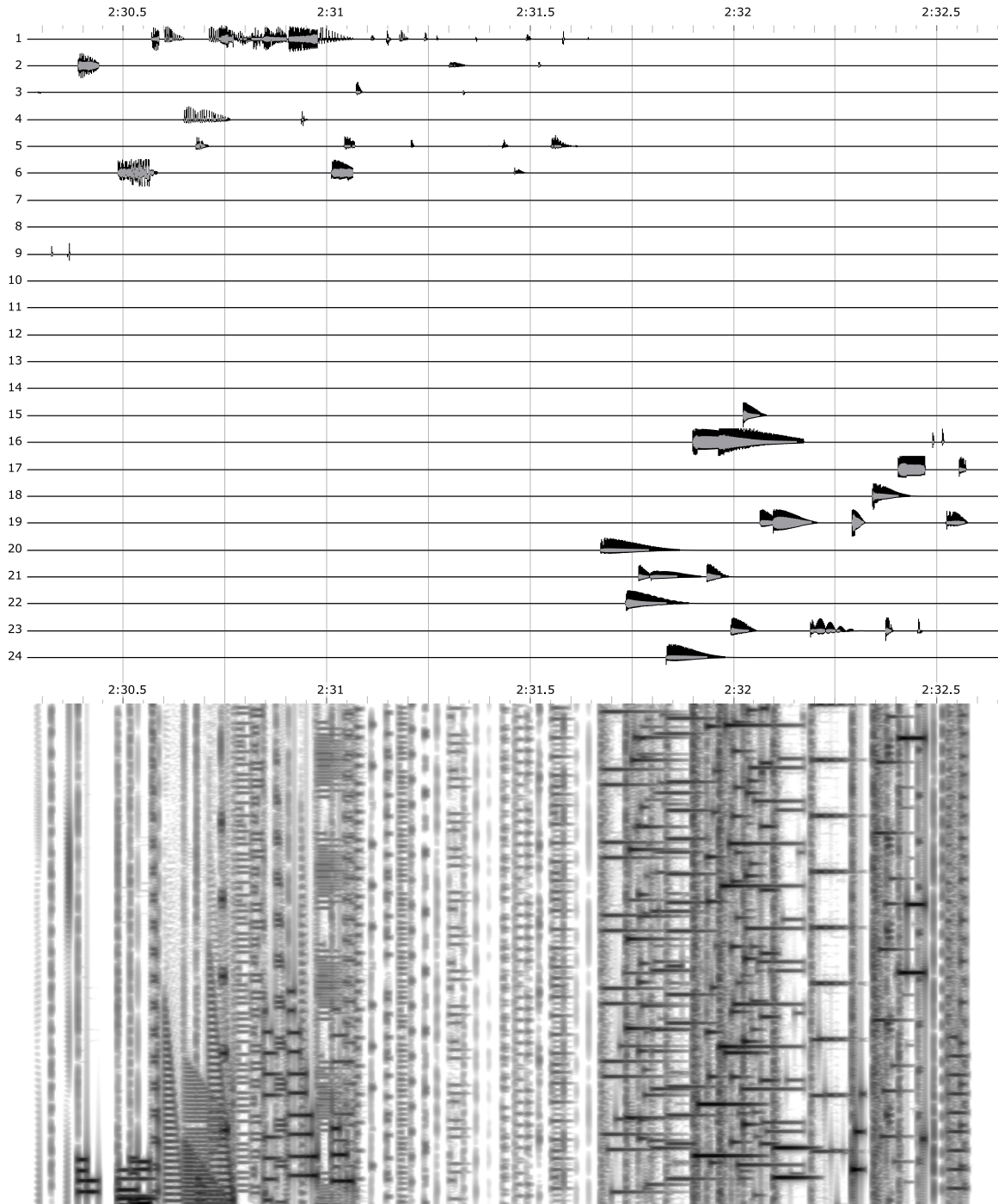


Figure 130: *Microcontrapunctus* — waveforms and spectrogram 2:30.0 - 2:32.8

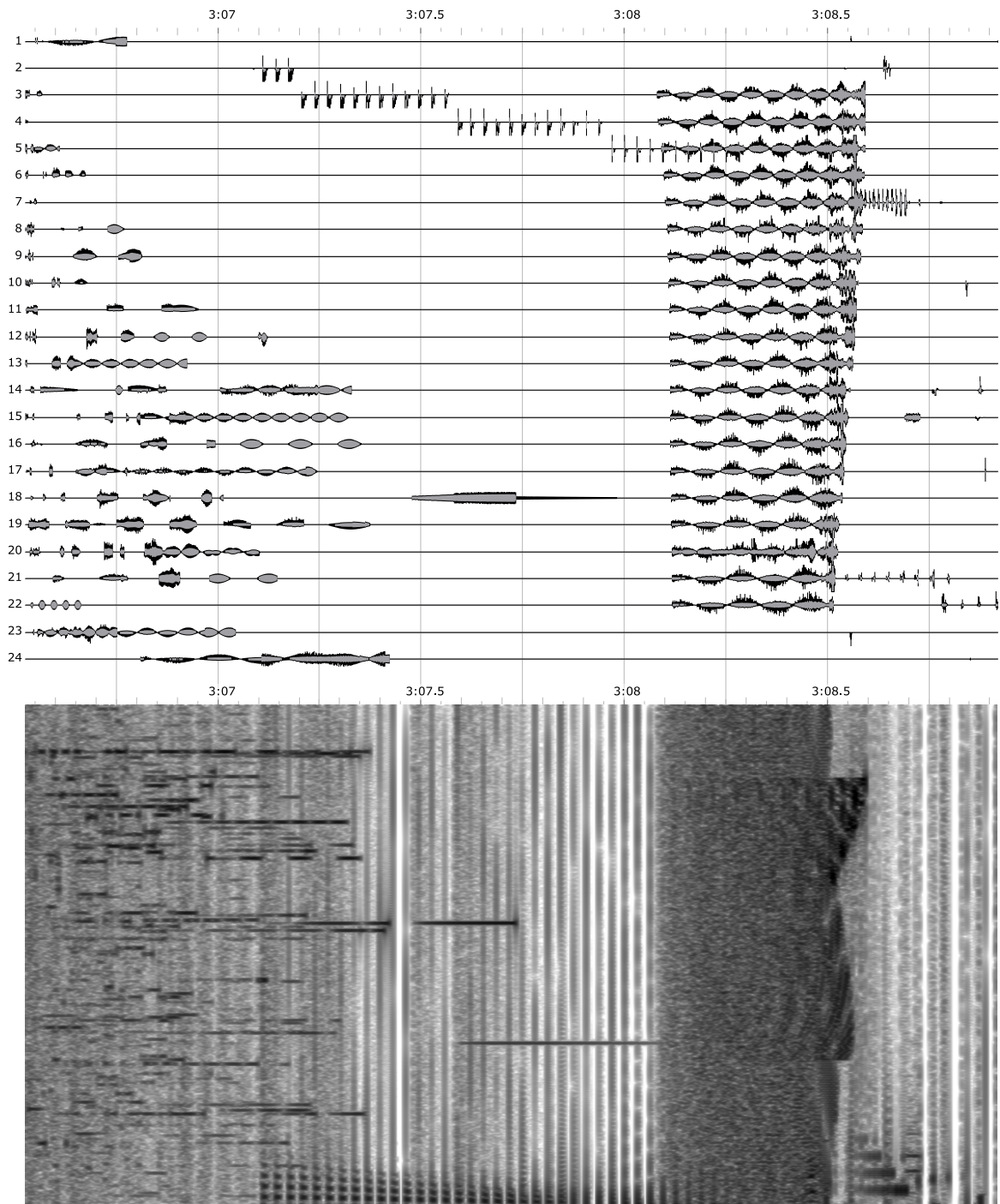


Figure 131: Microcontrapunctus — waveforms and spectrogram 3:06.1 - 3:09.3

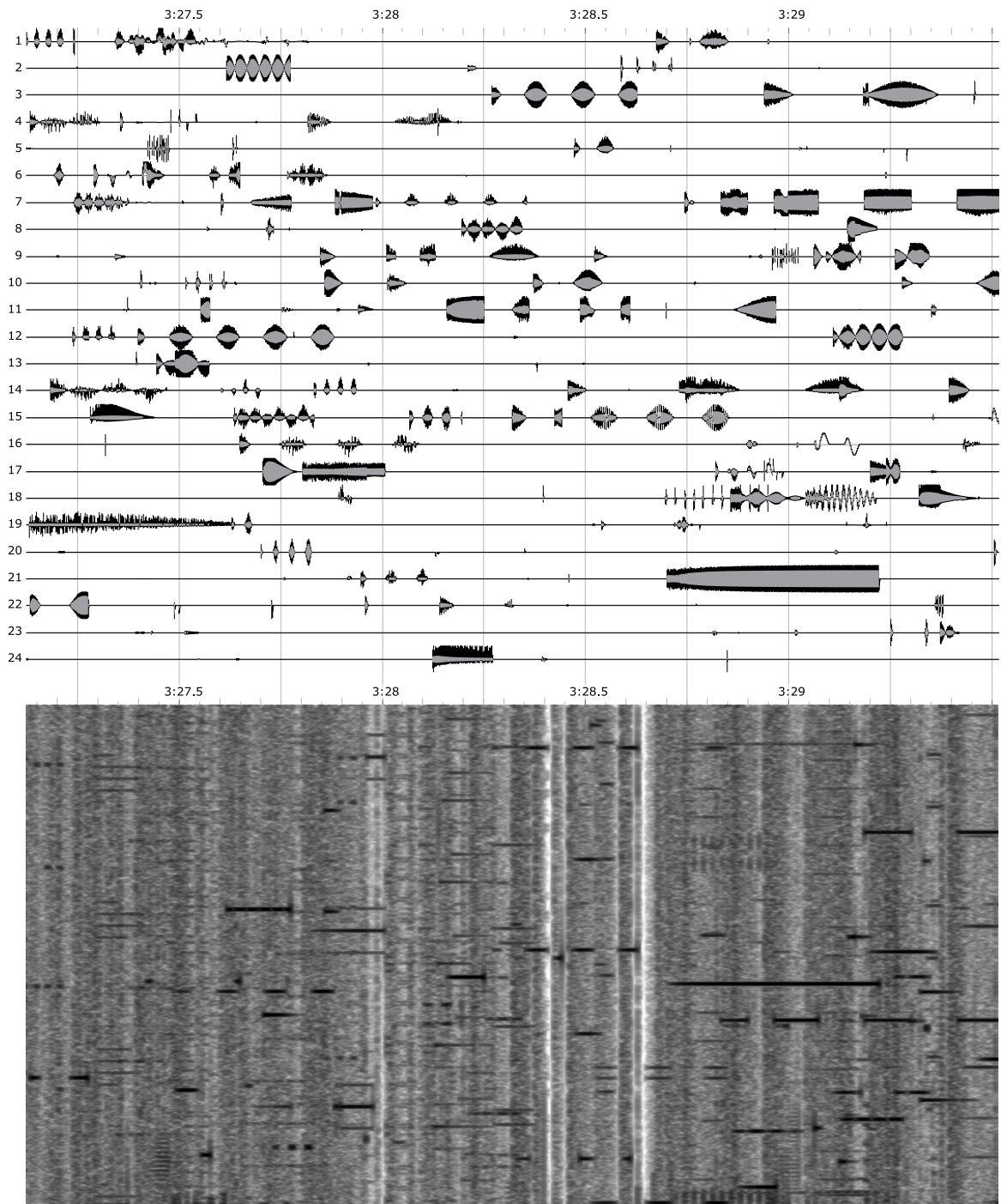


Figure 132: *Microcontrapunctus* — waveforms and spectrogram 3:26.8 - 3:30.0

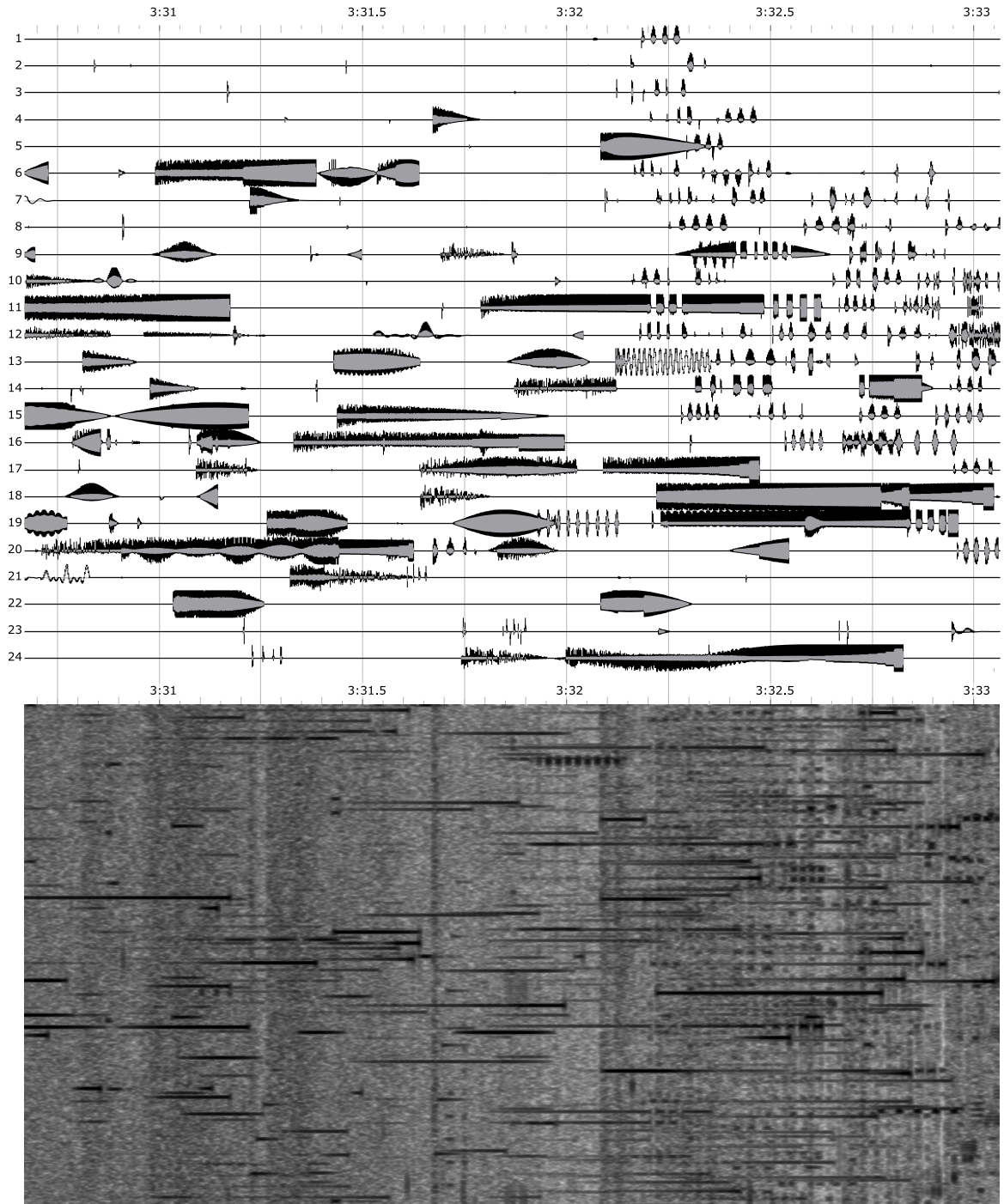


Figure 133: Microcontrapunctus — waveforms and spectrogram 3:30.4 - 3:33.1

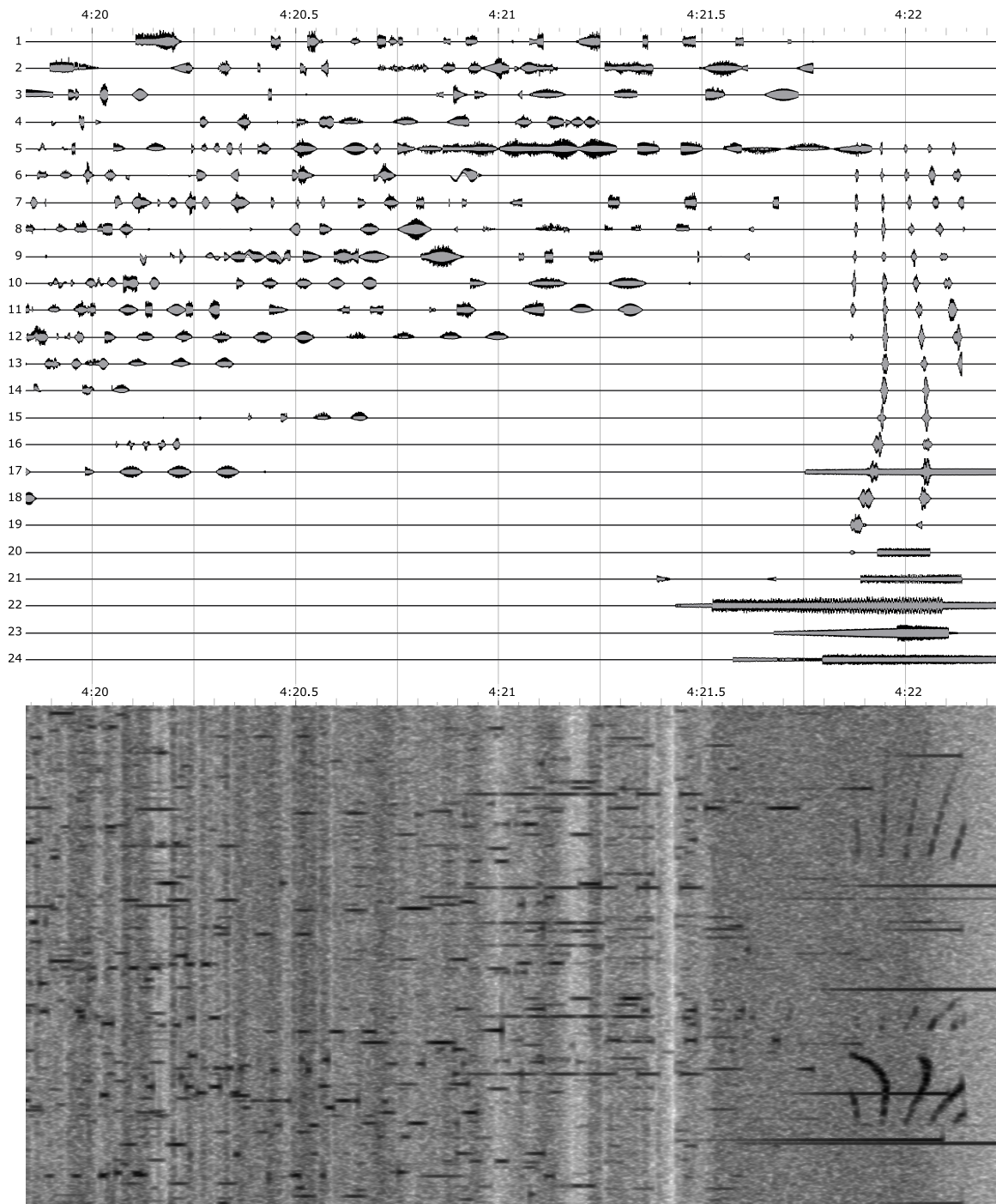


Figure 134: Microcontrapunctus — waveforms and spectrogram 4:19.7 - 4:22.4



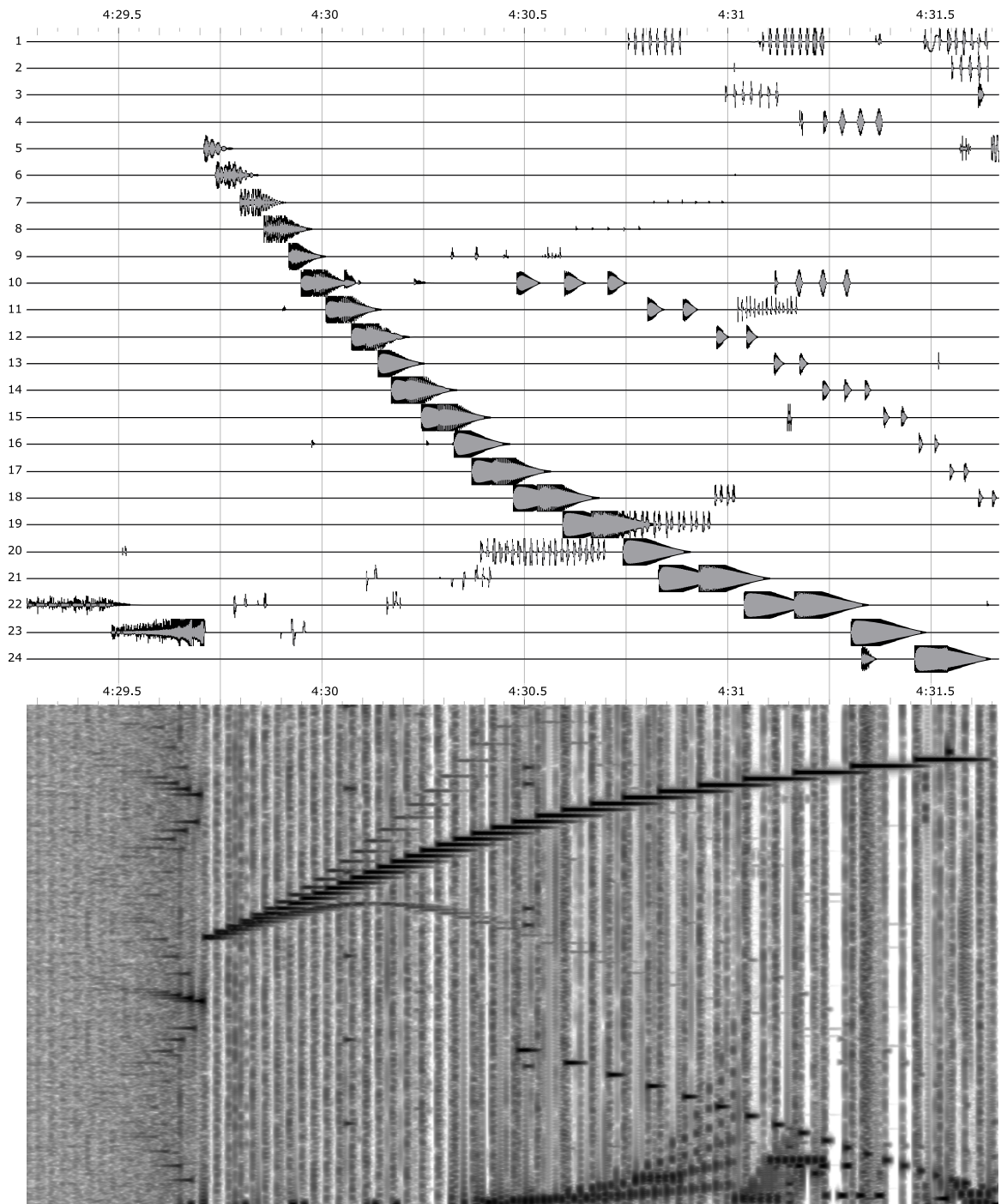


Figure 135: Microcontrapunctus — waveforms and spectrogram 4:29.1 - 4:31.3

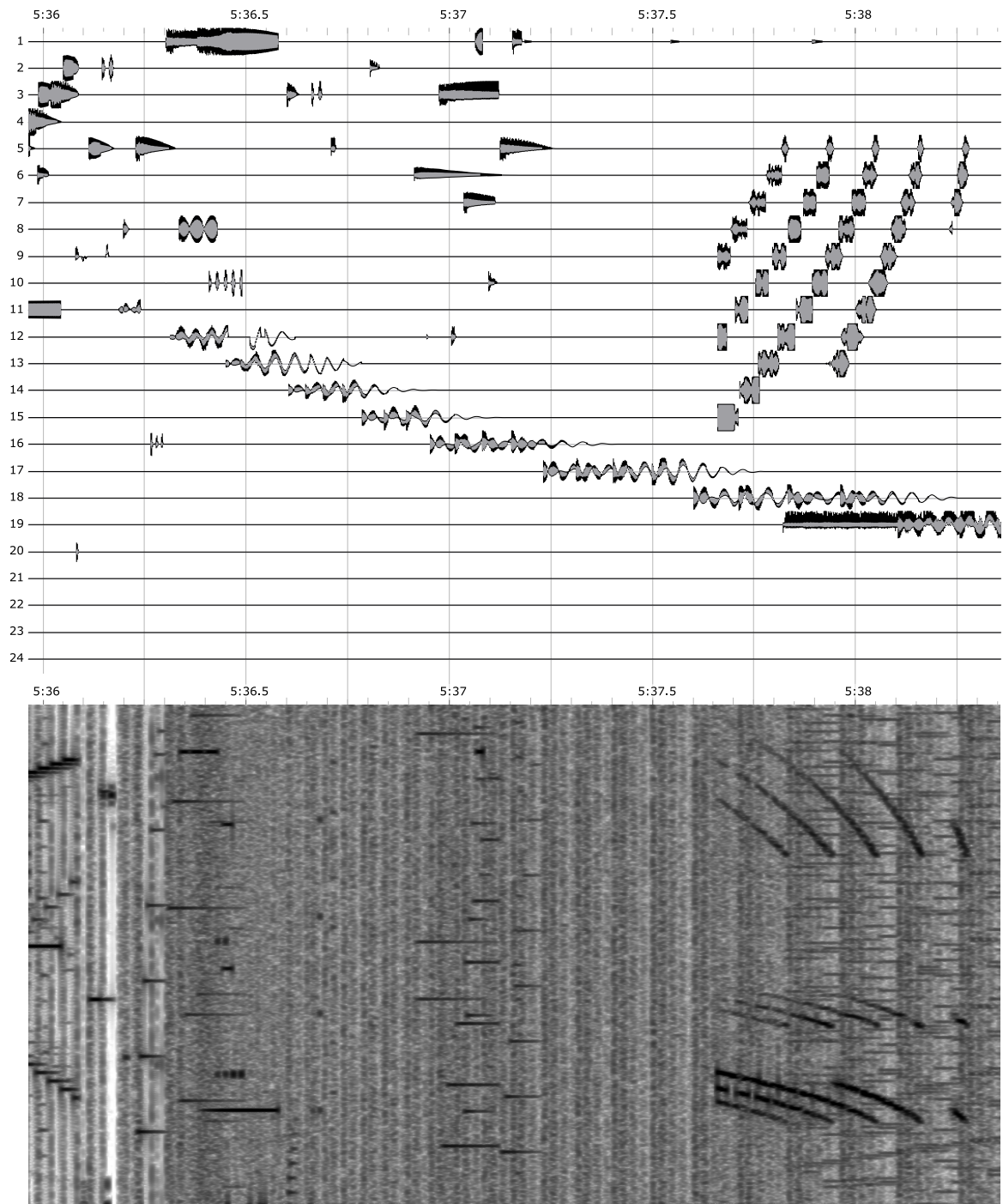


Figure 136: Microcontrapunctus — waveforms and spectrogram 5:36.0 - 5:38.7

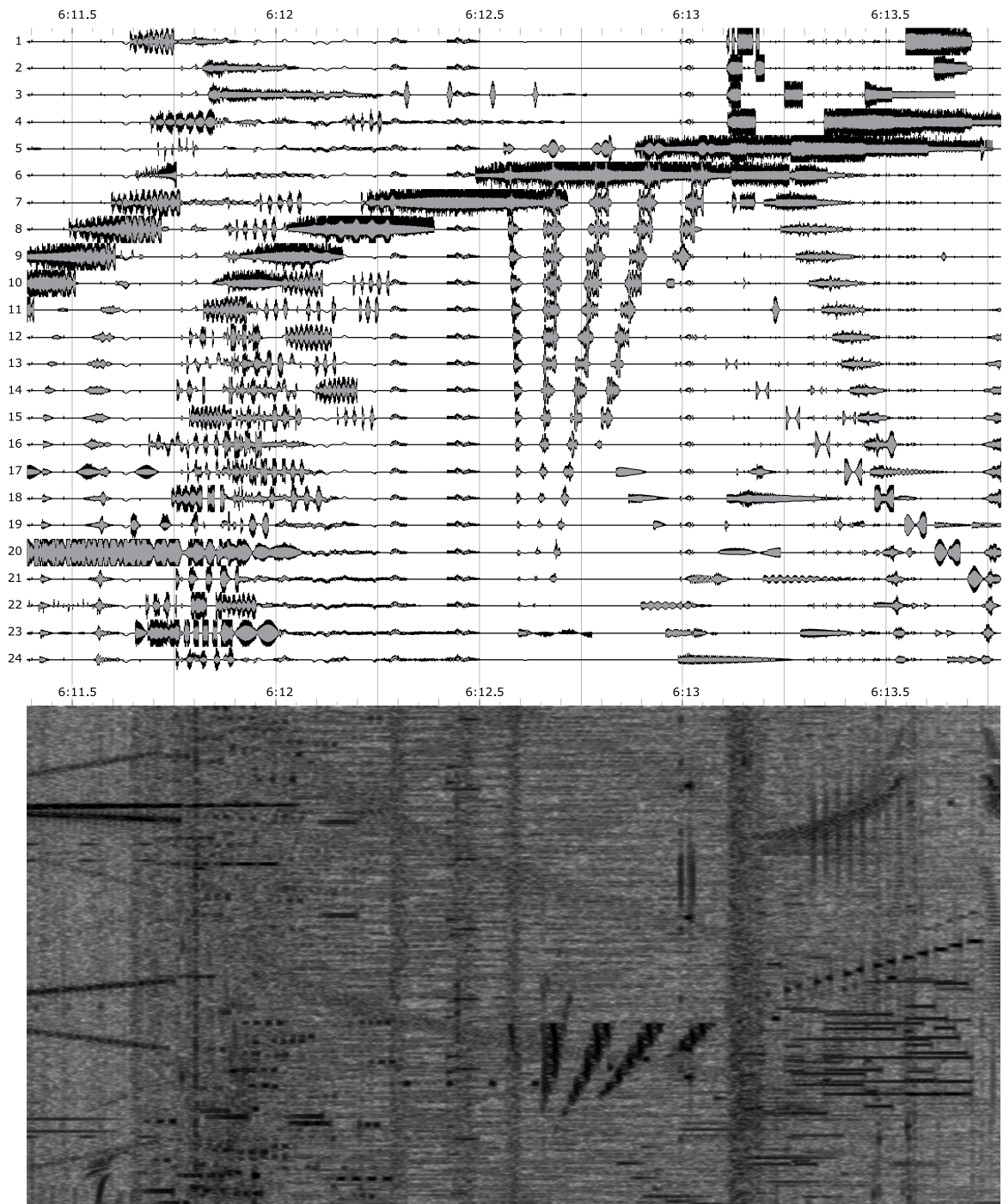


Figure 137: *Microcontrapunctus* — waveforms and spectrogram 6:11.3 - 6:14.0

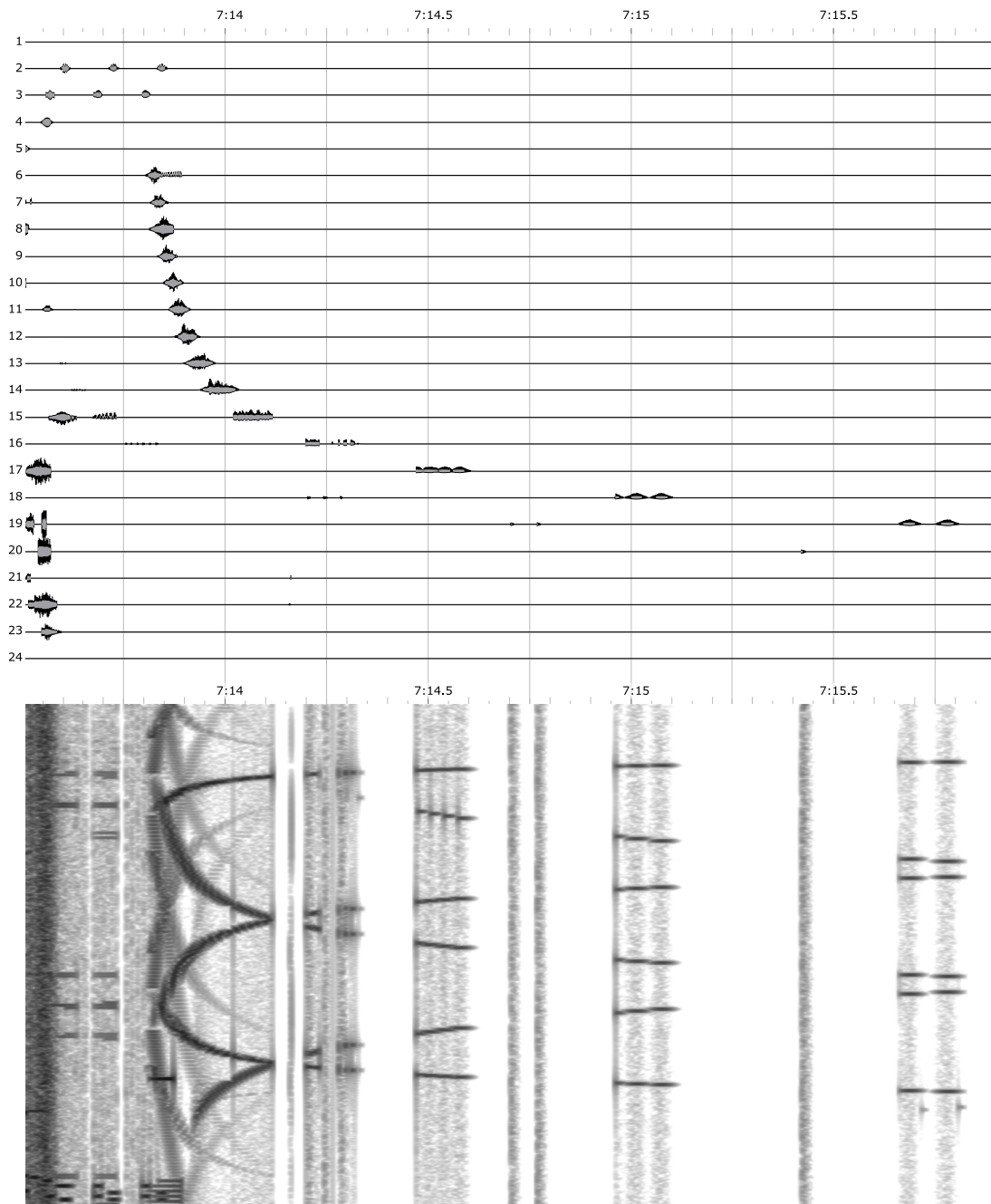


Figure 138: *Microcontrapunctus* — waveforms and spectrogram 7:13.0 - 7:15.8

## B.4. *Seven Places*

**Instrumentation:** violin, tape (stereo) and optional video  
**Duration:** 7 min  
**Premiere:** May 8th, 2016 at XII Mostra Sonora de Sueca, Valencia  
**Performers:** Carmen Antequera, violin  
Gregorio Jiménez, video and sound  
**URL:** Vimeo:  
<https://vimeo.com/lopezmontes/sevenplaces>

### Artistic concept

*Seven Places* closes a cycle of pieces derived from a collaboration with the violist, performer, and visual artist Charlotte Hug. During the preliminary work for the composition of *Badlands to the Skies* (2009), numerous multimedia materials were produced from Hug's analog visual and sonic raw materials, subsequently explored using various digital techniques.

*Seven Places* is characterized by both condensation and density of textures. It seeks the integration of the violin and electronics in a balance where each part maintains a clear idiomatic identity. The violin writing combines strong microtonality with a rhetoric that does not relinquish the instrument's inherent expressiveness.

Similar to *Badlands to the Skies*, the inspiration continues to stem from the fascination and abstraction of the desert landscape of gullies in the southeastern region of Andalusia where the composer resides.

Each microsection refers to a specific geographical point, identified by its coordinates and locatable using a QR code reader incorporated in the score, as seen in Figure 139, which displays the first page of the piece.

to Carmen Antequera

# Seven Places

for violin, electronics and optional video

José López-Montes

1  37.346617,-3.147881

Accelerando lievissimamente, ♩ = 29

violin



1 *ppp* senza vibrato *p* *pp*

4 *poco vibrato* *sfp* *p* senza vibrato

6 *mp* *ppp* *pp* *mf* vibrato

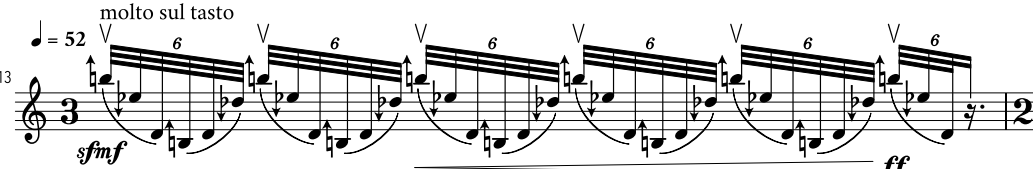
9 *f* *ff* molto espressivo

11 *molto détaché* *fff*

2  37.343318,-3.144546

molto sul tasto

♩ = 52



13 *sfmf* *ff*

\* This piece is based on non equal tempered sonorities. Microtonal accidentals are not intended to be played as an exact 24-quarter-tone equal temperament scale. The player must adjust these tiny intervals within a certain degree of freedom.

\*\* All glissandi must be played very gradually, unlike the usual expressive portamento.

\* Esta pieza está basada en sonoridades no temperadas. Las alteraciones accidentales microtonales no deben tocarse como intervalos de una escala temperada de cuartos de tono. El intérprete debe ajustar estos pequeños intervalos dentro de un cierto margen de libertad.

\*\* Todos los glissandi deben ser tocados muy gradualmente, evitando el usual portamento expresivo.

Figure 139: Seven Places — score's first page

## Methods

The use of GenoMus in this piece is limited to the generation of electronic sound in some parts, and becomes particularly evident, especially in section 6, partially reproduced in Figures 140 and 141. The textures of the electronics, characterized by abrasive sounds reminiscent of stony and rigid materials, have been synthesized in real-time from scores designed for electronic manipulation.

59

6

37.46365, -3.128509

♩ = 98

*sf* *sf tutta forza* *sf* *sf* *sf* *sf*

rallentando lievissimamente

64

*sf tutta forza* *sf* *sf* *sf* *sf* *sf* *f possibile* *sf*

sempre senza vibrato

Figure 140: Seven Places — video script, score and tape spectrogram, bars 59 to 69.

The figures display in parallel parts of the video accompanying the tape, the score, and a spectrogram, where these changes in rhythmic and timbral patterns can be observed.<sup>79</sup>

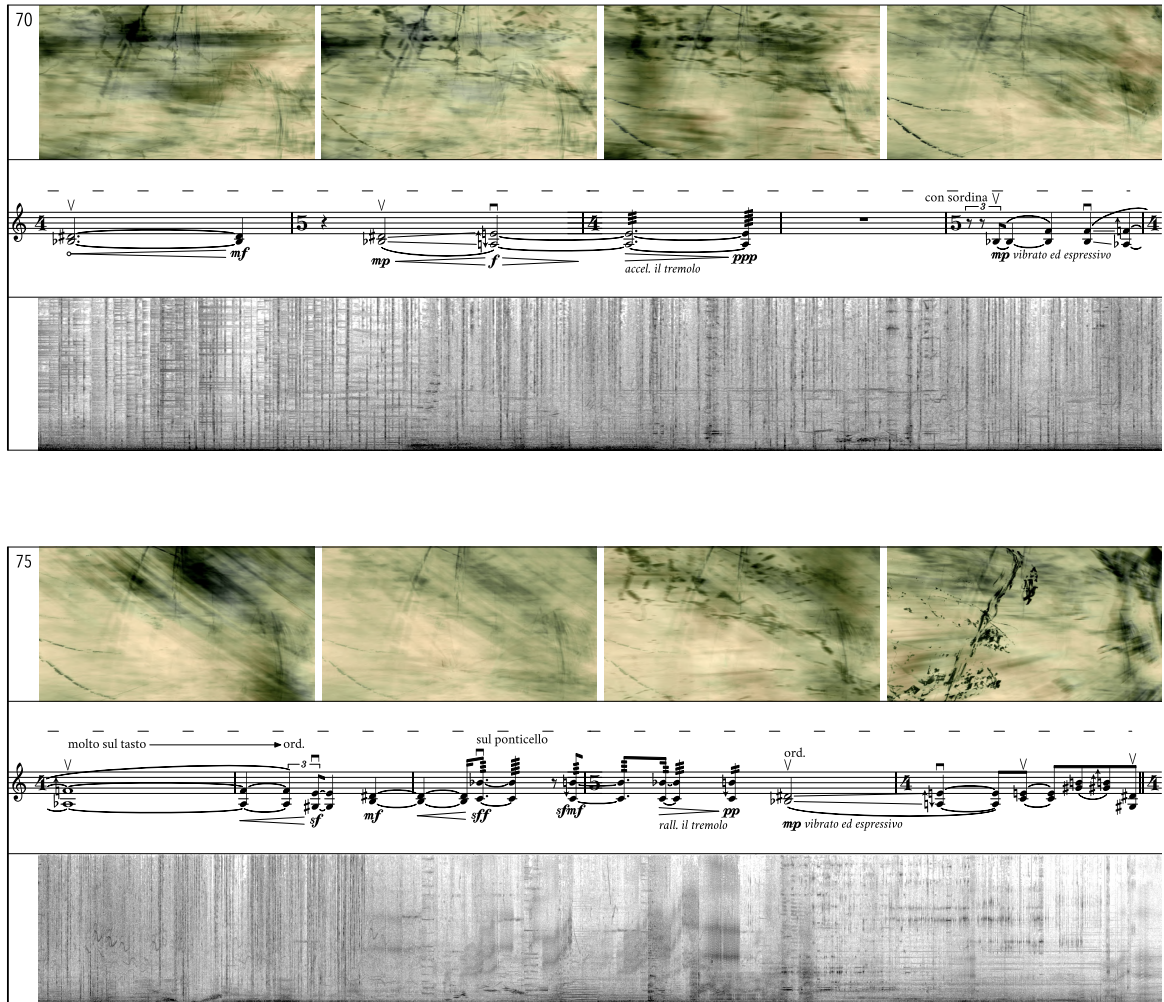


Figure 141: Seven Places — video script, score and tape spectrogram, bars 59 to 69. The spectrogram represents only the stereo tape accompanying the violin, displaying frequencies up to 20 kHz.

<sup>79</sup>It exists a Master's Thesis by Martín Gutiérrez [94] that includes an analysis conducting a detailed study of the multimedia aspects of the work.



## B.5. Choral Riffs from Coral Reefs

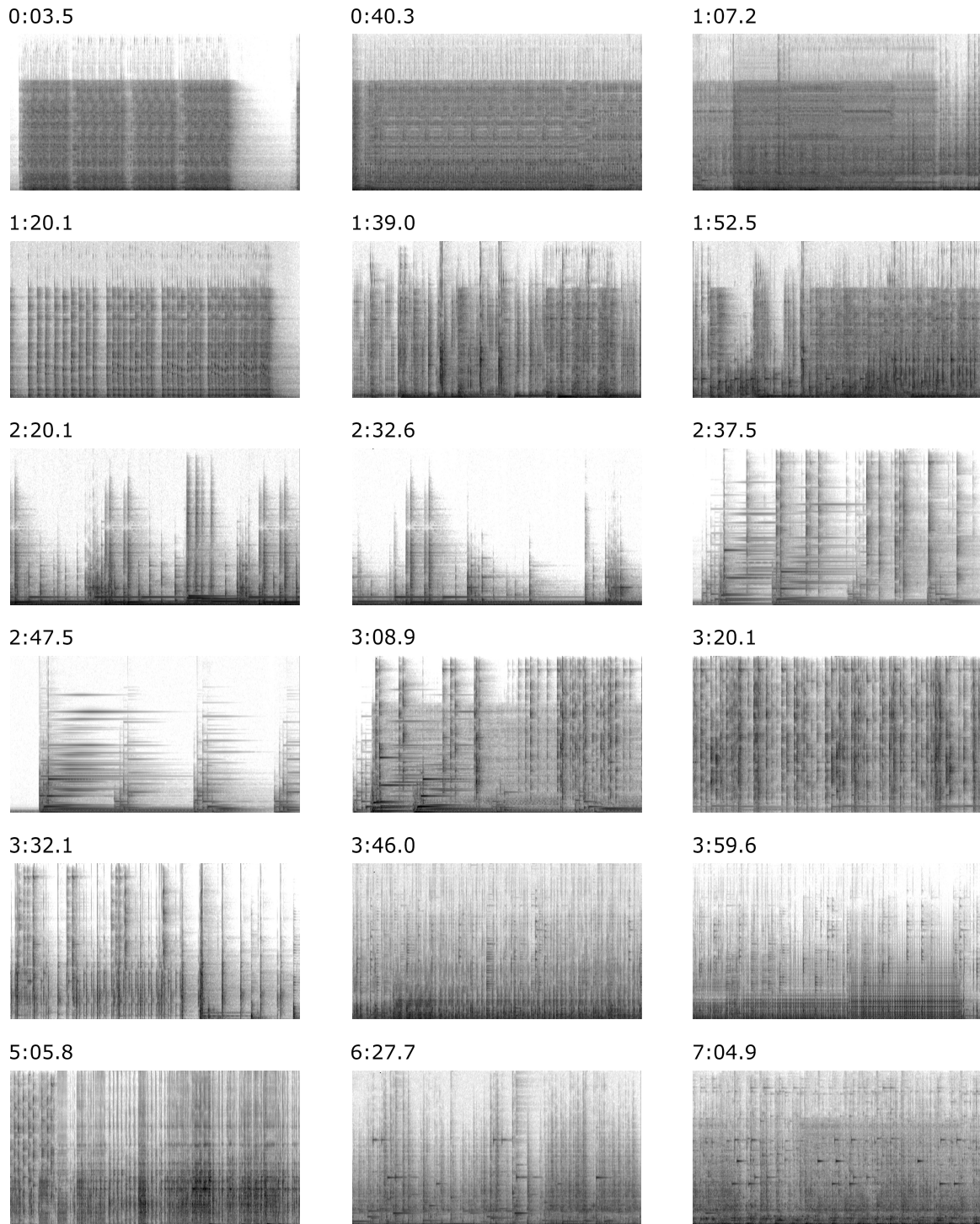
**Instrumentation:** stereo tape  
**Duration:** 8 min  
**Premiere:** December 5th, 2017 — La Madraza (Granada)  
XXIV Punto de encuentro de la Asociación de Música Electroacústica de España (AMEE)  
**URL:** Vimeo:  
<https://vimeo.com/lopezmontes/choralriffs>

### Artistic concept and methods

For a concert themed around coral reefs, the piece explored the possibilities of generating real-time audio with sequences of extremely high event density. These intricate scores were sent to virtual percussion and string instruments based on physical modeling. By pushing the parameters of these instruments' virtual physics to extremes, electronic-like timbres and textures were achieved, yet they maintained a high degree of believability as imaginary analog acoustic objects.

The musical interest lies in applying compositional processes to very small time scales to bring the use of the VST closer to granular synthesis. Figure 142 illustrates the diversity of spectral and temporal textures achieved with this method.

**Musical works**  
B.5. Choral Riffs from Coral Reefs



*Figure 142: Spectrograms of excerpts from Choral Riffs from Coral Reefs. Each image represents a slice of 3.5 seconds, and frequencies up to 20 kHz. At this time scale, the rhythmic textures generated by GenoMus can be well discerned.*

## B.6. *Juno*

**Subtitle:** Fanfare to Celebrate the Arrival of the Juno Spacecraft at Jupiter  
**Instrumentation:** brass septet and optional tape  
**Duration:** 5 min  
**Comission:** Bóreas Ventus  
**Release:** October, 2019  
**Performers:** Bóreas Ventus  
Alberto Castillo, piccolo trumpet  
Enrique Morillas, trumpet  
Antonio M. García, trumpet  
Manuel Herrera, horn  
Álvaro del Pino, trombone  
Antonio J. Delgado, euphonium  
Alberto Vallejo, tuba  
Pablo Rojas, conductor  
**URLs:** Spotify:  
<https://open.spotify.com/intl-es/track/0j00et9YcFq8G0v13LF3cE>

### Heuristics for harmony and instrumentation

GenoMus's role in this composition was limited to a very specific function: searching for harmonies that meet certain requirements:

- Suitability to the registers of the seven instruments
- Overall range
- Largest and smallest allowed internal intervals
- Global lower and upper limits
- Degree of diatonicity
- Presence of repeated notes
- Relationships with the preceding chord
- Link with pitches of preceding chords

The primary goal was therefore to test the incorporation of constraint-based search algorithms into the general architecture of GenoMus, similar to those used in OpenMusic and PWGL. This procedure quickly produced many harmonic progressions that could be easily adapted to the score. Figures 143 and 144 display fragments where these structures can be clearly seen, especially the use of common notes between chords.

The image displays two systems of a musical score for the piece 'Juno'. The first system covers bars 27 to 36, and the second system covers bars 37 to 47. The score is for a transposing ensemble and includes parts for Piccolo Trumpet in B-flat, Trumpet 1 in B-flat, Trumpet 2 in B-flat, Horn in F, Tenor Trombone, Euphonium in B-flat, and Trombone in C. The music is in 3/4 time. The first system features a melodic line with dynamic markings of *f*, *mp*, *f*, and *p*. The second system, starting at bar 37, is marked with a 'C' in a box above the first staff and features a dynamic crescendo from *ff* to *pp* across the measures. A *pppp* marking is also present at the end of the second system.

Figure 143: Juno — bars 27 to 47 (transposing score)

The image displays a musical score for the piece 'Juno', specifically bars 78 to 93, presented as a transposing score. The score is organized into two systems, labeled 'F' and 'G' at the top of each system. The first system (F) covers bars 78 to 86, and the second system (G) covers bars 87 to 93. The instrumentation includes Piccolo Trumpet in B-flat, Trumpet 1 in B-flat, Trumpet 2 in B-flat, Horn in F, Tenor Trombone, Euphonium in B-flat, and Trombone in C. The score is written in a common time signature (C) and features a variety of dynamic markings such as *p*, *mf*, *f*, *ff*, *fz*, *sfmf*, and *mp*. The notation includes slurs, accents, and dynamic hairpins to indicate the intended performance dynamics. The key signature is one flat (B-flat major or F minor).

Figure 144: Juno — bars 78 to 93 (transposing score)

On the other hand, the GenoMus prototype for this composition is the first to use, in a simplified version, the subspecimen model described in Section 3.3 as a key element of the functional metaprogramming.

## B.7. Openings for FACBA Podcasts

- Instrumentation:** synthesizers and virtual instruments
- Total duration:** 3 min
- Comission:** University of Granada — Faculty of Fine Arts — Podcast FACBA
- Website:** <https://facba.info/>
- Season 2020:** **FACBA'20 - Seminario El saber oscuro: ritos, senderos y tránsitos del conocimiento artístico** — Ep. 0 to 13
- Release:** May, 2020
- URLs:** Soundcloud:  
<https://soundcloud.com/webmaster-bbaa/episodio-creditos-seminario-el-saber-oscuro>  
Spotify:  
<https://open.spotify.com/episode/2GAGDAdB1frEAuEAoPECoh>  
Apple Podcast:  
<https://podcasts.apple.com/es/podcast/episodio-cr%C3%A9ditos-seminario-el-saber-oscuro/id1513608842?i=1000475196881>
- Season 2021:** **FACBA'21 - Seminario La variación infinita** — Ep. 14 to 28
- Release:** Jun, 2021
- URLs:** Soundcloud:  
<https://soundcloud.com/webmaster-bbaa/episodio-14-equipo-comisarial-facba-21-marisa-mancilla-regina-perez-y-rosario-velasco>  
Spotify:  
<https://open.spotify.com/episode/3eTR7fHFmVYPc4vqlcipQV>  
Apple Podcast:  
<https://podcasts.apple.com/es/podcast/radio-bellas-artes-granada/id1513608842>
- Season 2022:** **FACBA'22 - Seminario Contrarritmo, reescalas y distorsiones de nuestro tiempo acelerado** — Ep. 29 to 37
- Release:** July, 2022
- URLs:** Soundcloud:  
<https://soundcloud.com/webmaster-bbaa/sets/seminario-facba-22>  
Spotify:  
<https://open.spotify.com/episode/4EnTFFAEmGrVkATz91tYkb>  
Apple Podcast:  
<https://podcasts.apple.com/es/podcast/episodio-29-equipo-comisarial-marisa-mancilla-y-rosario/id1513608842?i=1000569252673>

## Ready-made music

Following the lockdown caused by the COVID-19 pandemic in 2020, the Faculty of Fine Arts at the University of Granada, traditionally hosting an annual festival, opted to replace their usual exhibition with a podcast featuring artist interviews. I was tasked with creating the musical introduction that opens and closes each episode, a role that continued in subsequent seasons as the podcast remained active.

My working method involved preparing a series of musical proposals under one premise: to use the music exactly as it was produced by the GenoMus in just a few milliseconds. Human intervention was limited to selecting a virtual instrument and, in some cases, adding complementary sound effects. For the 2021 edition, the musical output was a sound synthesis score for Csound, unaltered except for mastering.

It should be noted that these have been the first proper commercial use cases of the tool and that the ratio between work time and economic benefit was very favorable, if this can be considered a possible objective criterion for validating the tool.

## B.8. *Tiento*

|                         |  |
|-------------------------|--|
| <b>Composers:</b>       | Pilar Miralles & José López-Montes   |
| <b>Instrumentation:</b> | binaural tape  |
| <b>Duration:</b>        | 35 min   |
| <b>Comission:</b>       | University of Granada — Faculty of Fine Arts<br>Podcast <i>La variación infinita</i> in Festival FACBA   |
| <b>Premieres:</b>       | July 21st, 2021, released at several digital platforms   |
| <b>URLs:</b>            | Soundcloud:<br><a href="https://soundcloud.com/webmaster-bbaa/episodio-especial-m-del-pilar-miralles-y-jose-lopez-montes-tiento">https://soundcloud.com/webmaster-bbaa/episodio-especial-m-del-pilar-miralles-y-jose-lopez-montes-tiento</a><br>Spotify:<br><a href="https://open.spotify.com/episode/2jGEQovaGj1280KeV5epeb">https://open.spotify.com/episode/2jGEQovaGj1280KeV5epeb</a><br>Apple Podcast:<br><a href="https://podcasts.apple.com/es/podcast/episodio-especial-m-del-pilar-miralles-y-jos%C3%A9-l%C3%B3pez/id1513608842">https://podcasts.apple.com/es/podcast/episodio-especial-m-del-pilar-miralles-y-jos%C3%A9-l%C3%B3pez/id1513608842</a> |

### Artistic concept

*Tiento* is a lengthy piece of pure electronic composition designed to be premiered at the FACBA festival within its podcast *La variación infinita*, centered around the thematic thread of mutation as a central element of artistic creation. After successive stages of generation, crossbreeding, mutation, and evolution applied to sound synthesis in its purest form, the authors presented this kind of acousmatic symphony as a demonstration of artificial creativity in its most open and unbiased mode, allowing the machine to chart its own paths with the greatest freedom and capacity for surprise. The preferred way to listen to the piece is with headphones, as it extensively employs 3D binaural spatialization techniques.

The title refers to the term used in the Spanish Renaissance for instrumental pieces with a character of contrapuntal study and sound exploration. It is the Spanish equivalent of the Italian *ricercare*.



## Methods

This piece marks the first instance in which another composer utilizes the model to experiment with it. In this case, Pilar Miralles created all of her musical material using GenoMus, employing it in real-time to modify audio samples also produced with this tool. The collaborative process unfolded in several stages:

- Creation of raw synthetic sound material, starting from open exploration of the algorithm in connection with Csound and utilizing SuperCollider routines as audio synthesis engines.
- Exchange of selected materials among the authors, subsequently mutated using GenoMus as a real-time controller for a sound manipulation tool created in Max.
- Composition of medium-length sections by organizing and overlaying the obtained materials, avoiding modification of the products of autonomous algorithmic manipulation.
- Proposals for the sequential arrangement of the selected sections.
- Mixing, binaural spatialization, and final post-production processes.

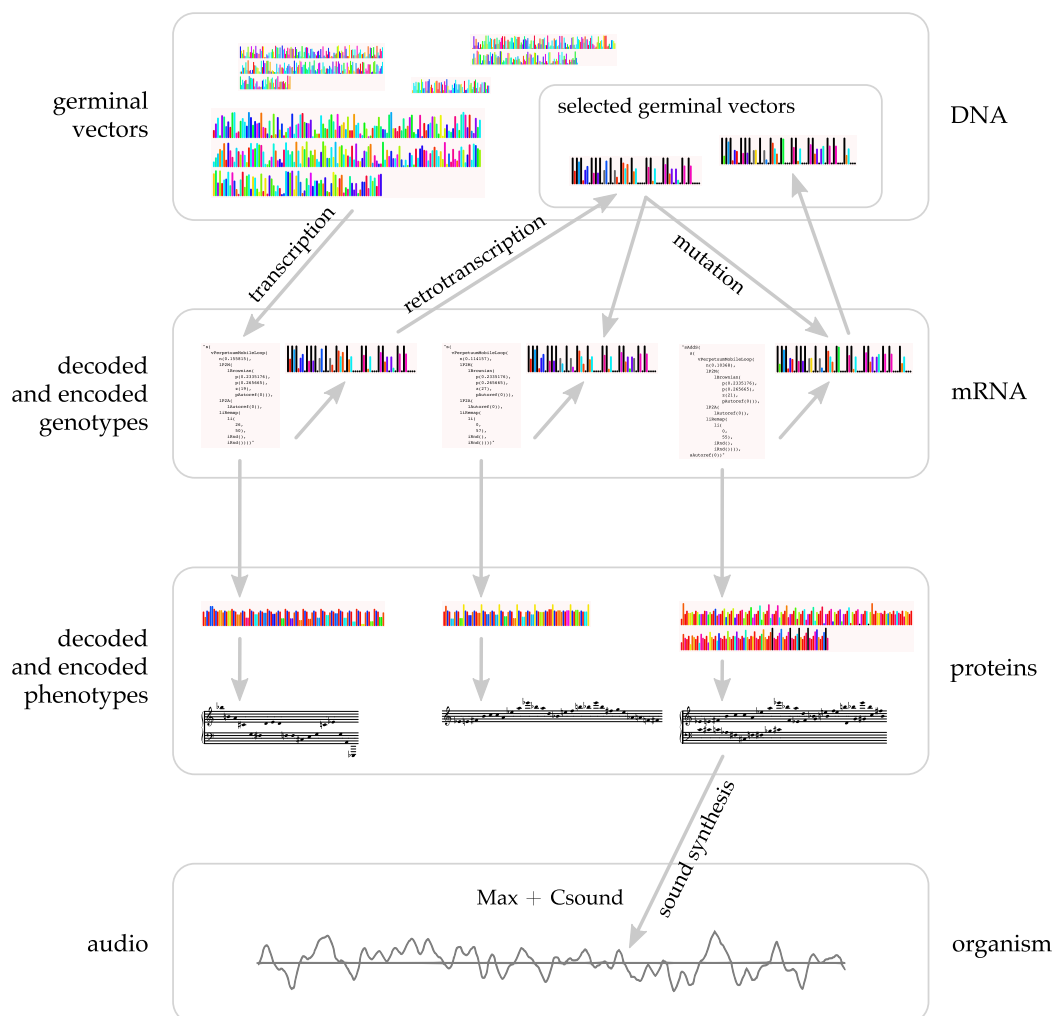


Figure 145: First conceptualization of the germinal vector and retrotranscription in the model. Adapted from Lopez-Montes and Miralles [86].

Besides using the tool as a remote controller for another program, the main novelty introduced by this composition in the development of GenoMus was the first implementation of the germinal vector and retrotranscription, as explained in Sections 5.4 and 5.6. In an article by the authors dedicated to this piece in the publication associated with the FACBA podcast [86], Figure 145 appeared to explain the analogies between genomics processes and the data abstractions handled for composition. This was described in that article as follows:

**Germinal vector:** The primordial material from which the process begins is a random sequence of numbers between 0 and 1, a pure mathematical abstraction that determines the complex decision tree that forms each musical fragment. I call this numerical sequence the germinal vector, which would correspond to DNA as the repository of genetic information from which the rest of the generative processes are initiated. Any numerical sequence is valid as a germinal vector.

**Decoded genotype:** The germinal vector, along with other initial conditions such as constraints on the maximum allowed dimensions of branching and extension of the parameter tree, is translated into the decoded genotype, which is a textual expression comprised of musical functions taken from a library of basic compositional procedures (repetition, remapping, generative and stochastic processes, addition of more polyphonic lines, etc.). The encoded genotype corresponds in this analogy to mRNA, the product of the transcription of DNA strands as an intermediate step in protein production.

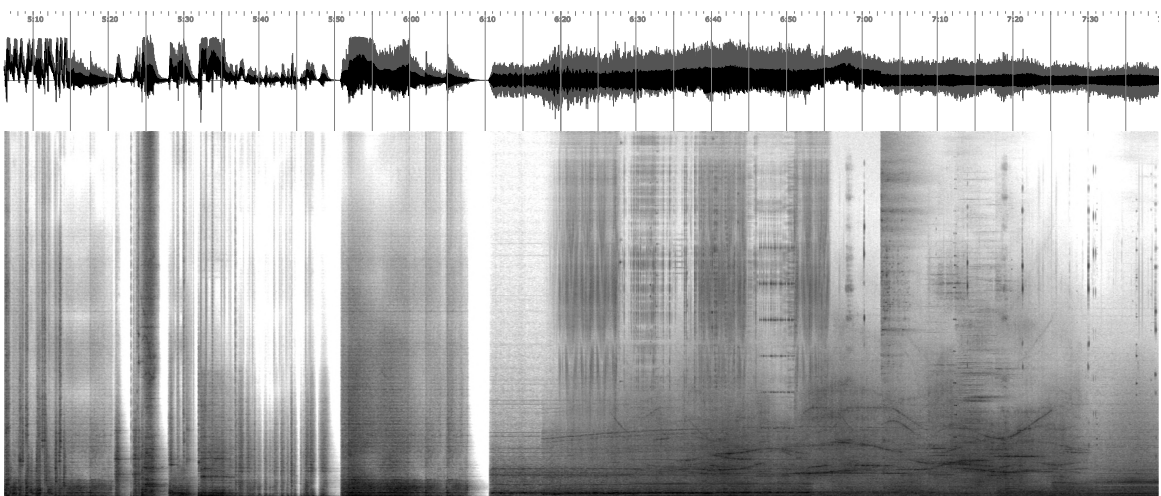
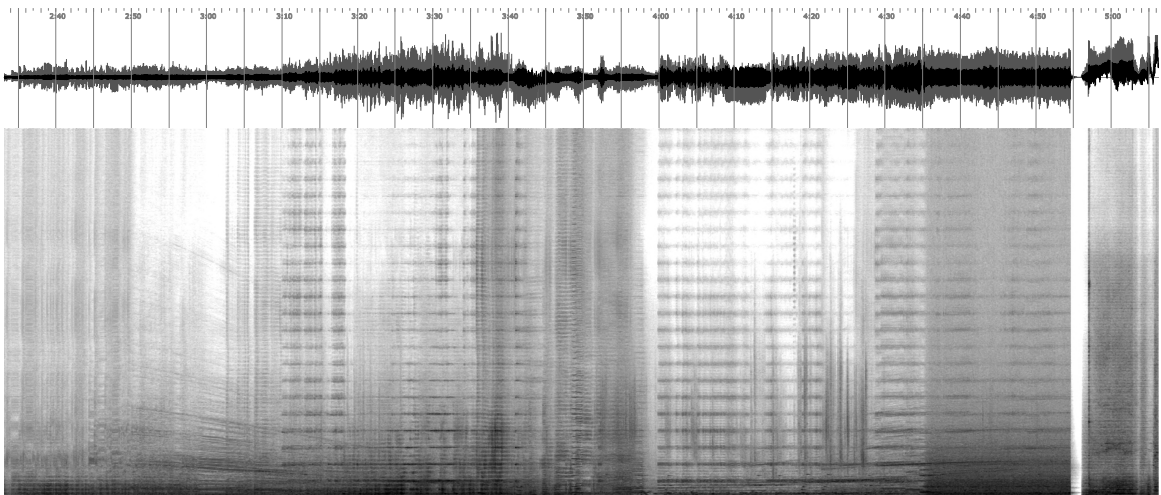
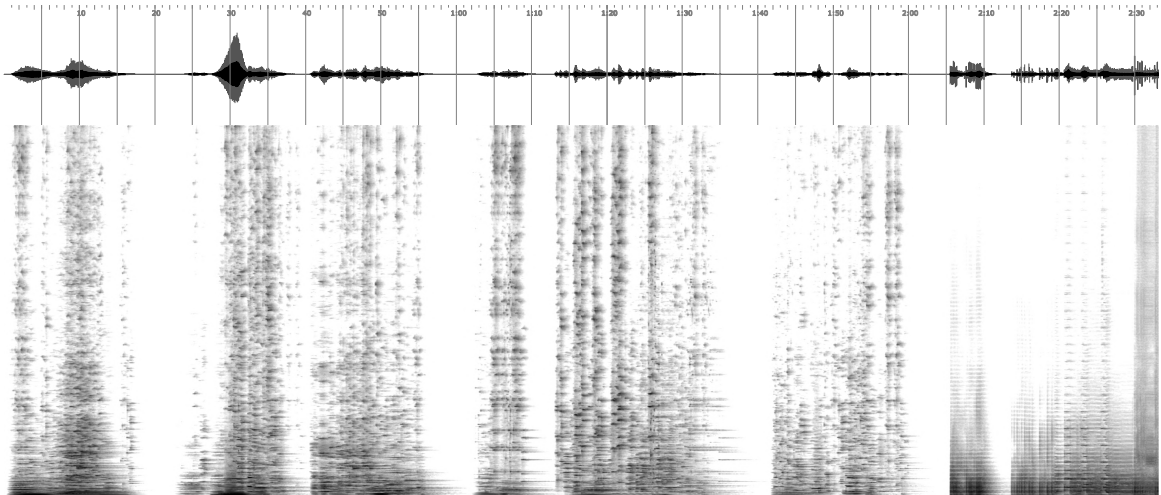
**Phenotype:** The final step is the translation of these processes invoked by the decoded genotype, achieved by finally evaluating the functional expression that constitutes the genotype itself. The phenotype would be analogous to the final organism, comprised of different musical elements, akin to proteins materialized by the execution of the information transcribed in mRNA.

**Encoded genotype:** Once a decoded genotype, considered useful for its musical result, has been selected, a reverse transcription is performed, recoding it again as an optimized germinal vector (purely numerical). This operation allows selected characteristics to be separated from unnecessary parts of the original germinal vector, thereby enhancing the quality and expressive precision of the material with which mutation and recombination will continue. Reverse coding also exists in biology: the enzyme reverse transcriptase copies mRNA into DNA, as seen in retroviruses, for example. In our case, this operation is essential to obtain high-quality germinal vectors from the initial random genetic material, free from redundant or unnecessary information.

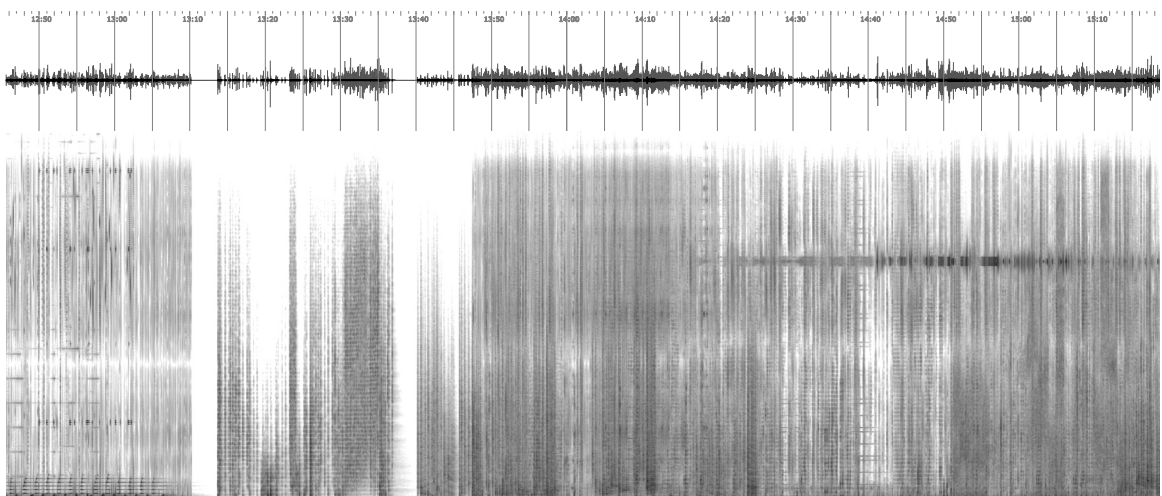
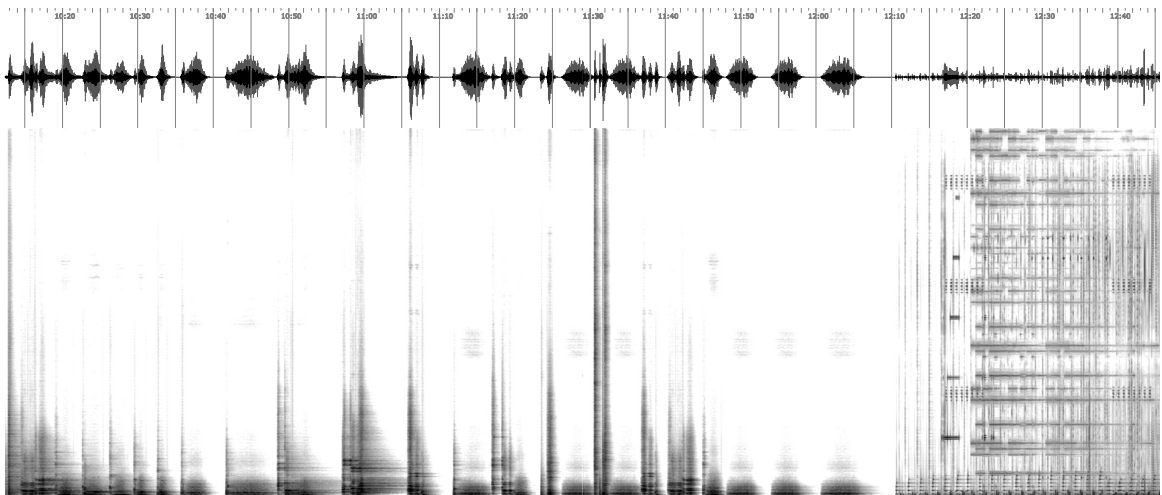
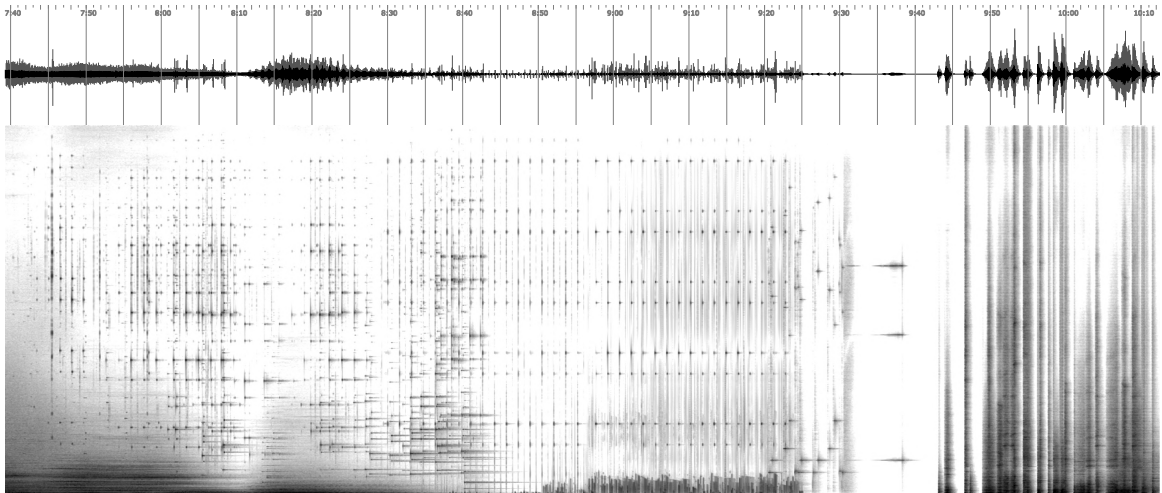
## Waveform and spectrogram

Attached below is the graphical representation of the entire piece, displaying the waveform (monophonic version) alongside the spectrogram, with a vertical range of up to 20 kHz. Although this doesn't replace listening, it can aid in visualizing the diversity of textures in event generation and timbre modulation achieved with the tool for a large-scale composition.

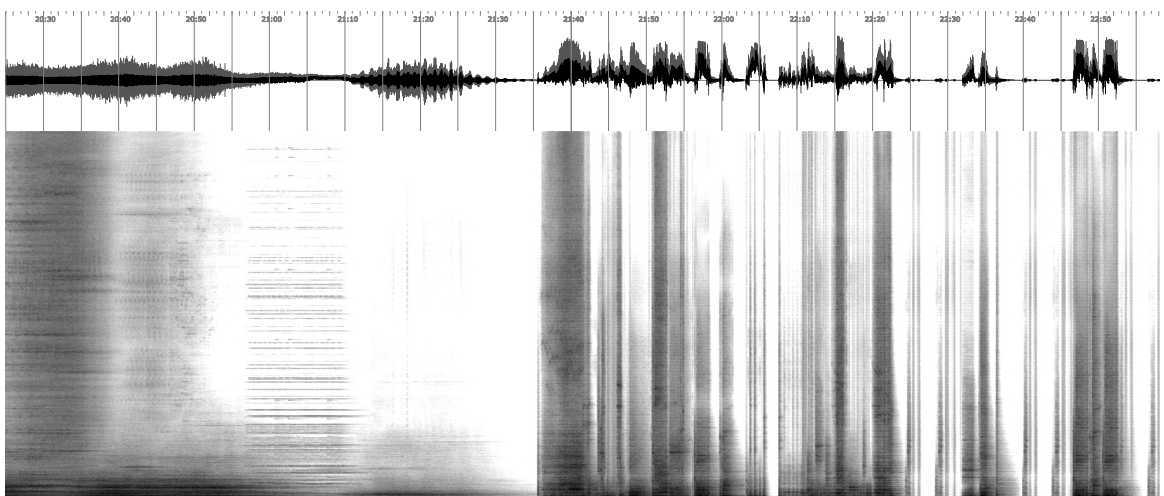
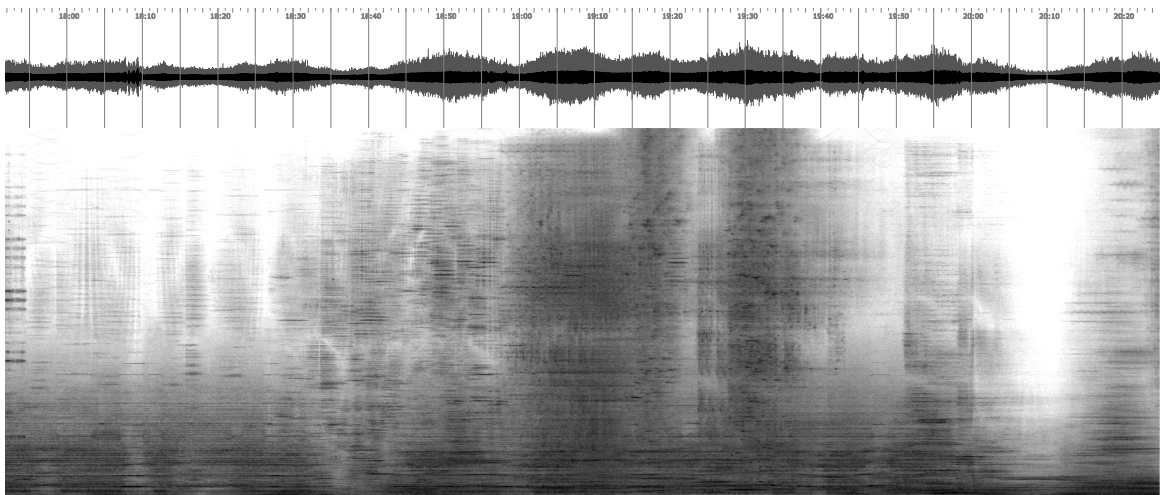
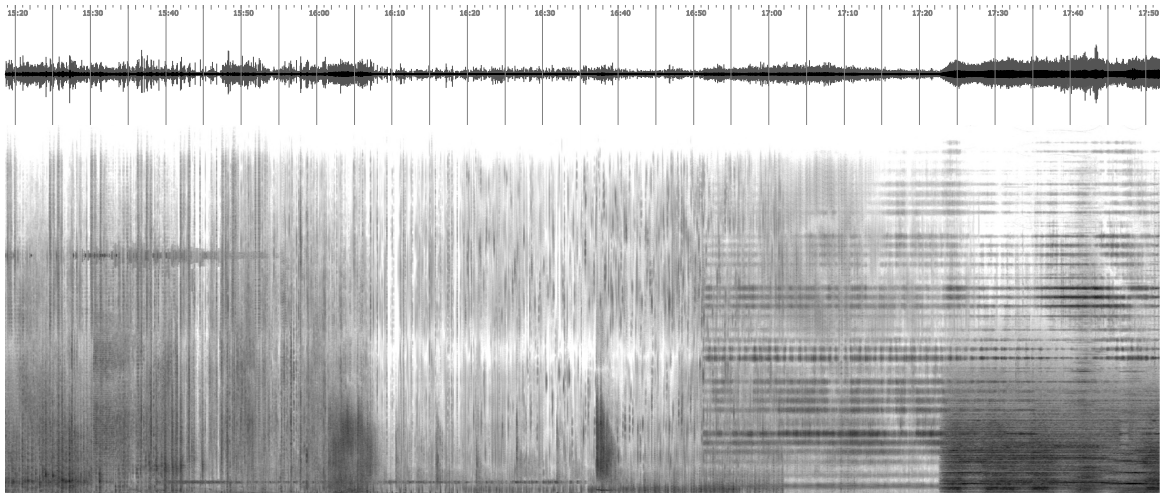
Musical works  
B.8. Tiento



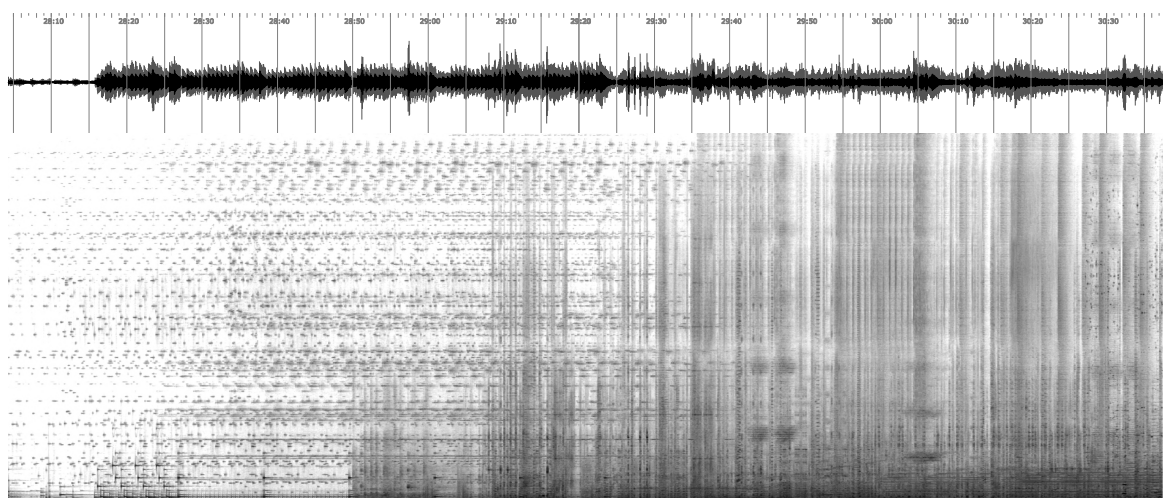
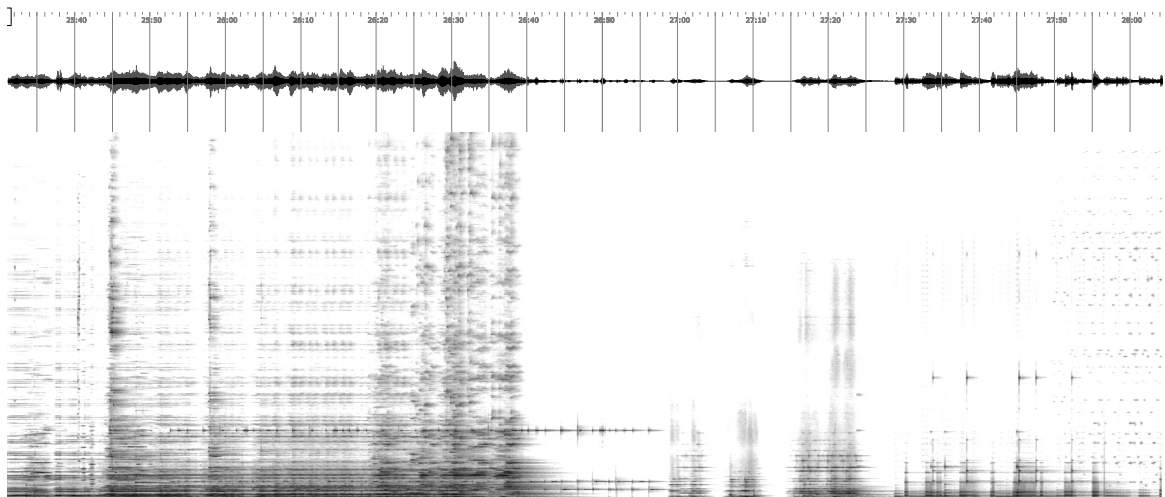
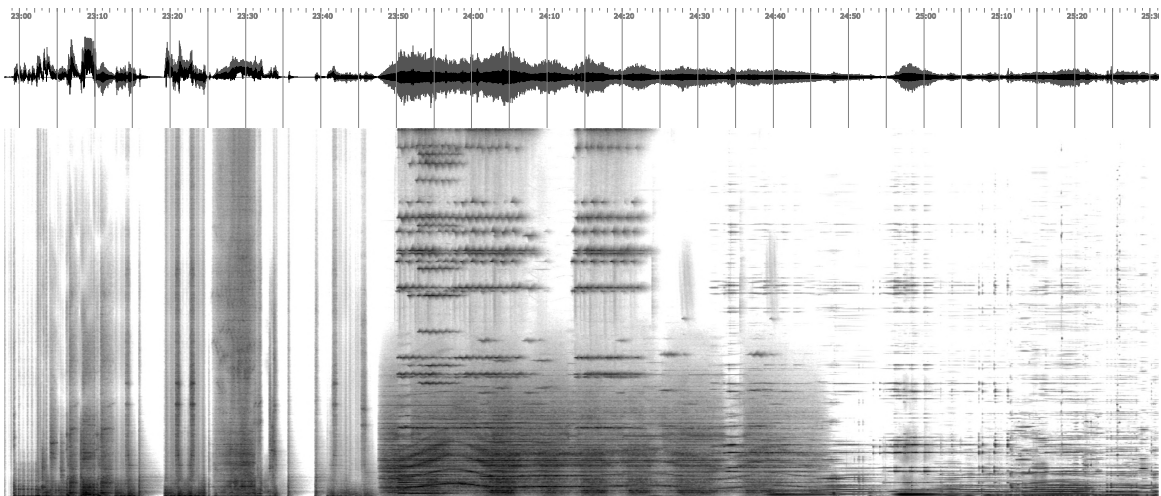
Musical works  
B.8. *Tiento*

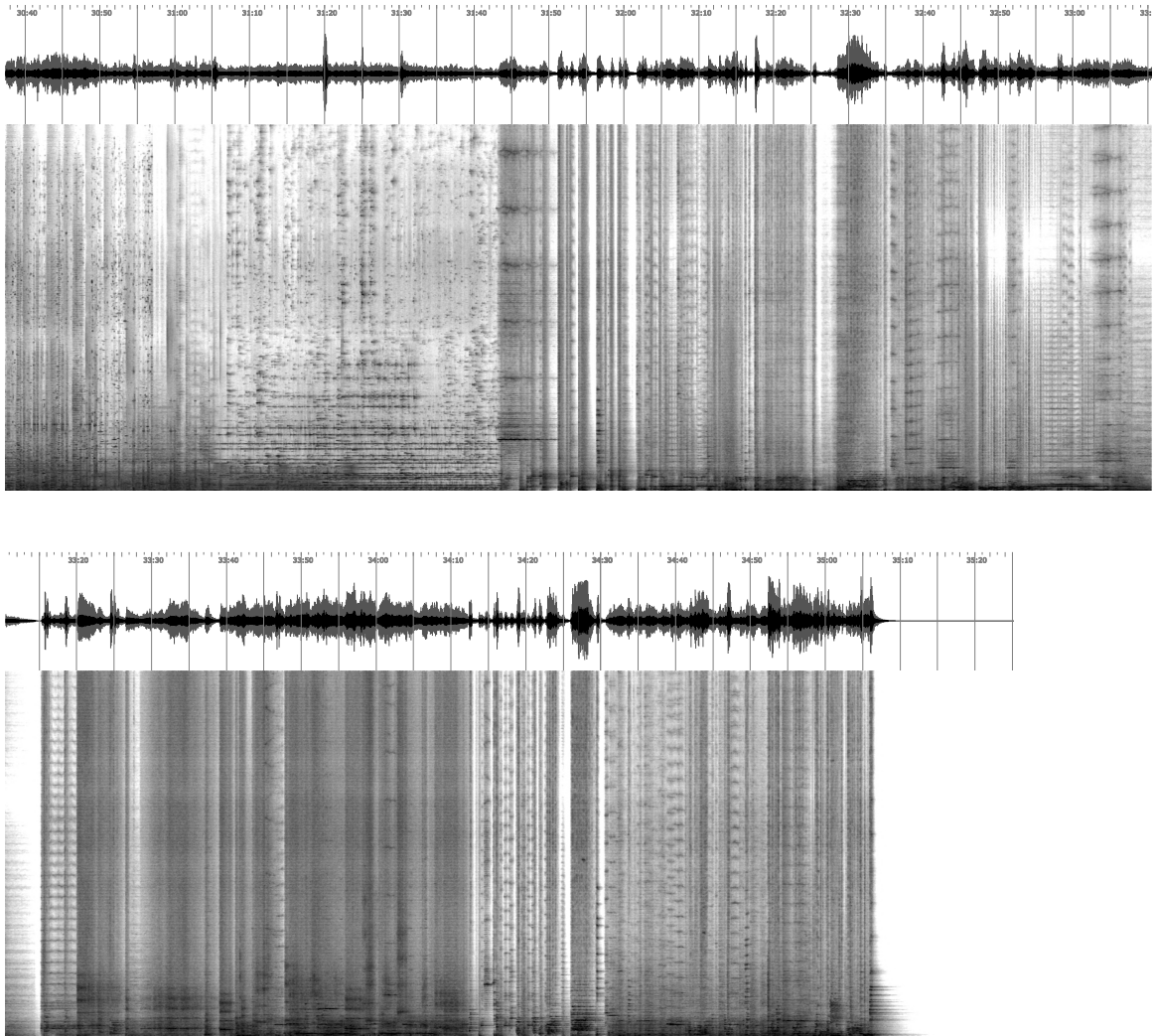


Musical works  
B.8. *Tiento*



Musical works  
B.8. Tiento





*Figure 146: Spectrogram of Tiento*



## B.9. *Rudepoema na penumbra*

**Instrumentation:** quadraphonic tape  
**Duration:** 23 min  
**Comission:** University of Granada — Cátedra Manuel de Falla  
**Premieres:** March 24th, 2022 — Espacio V Centenario (Granada)  
**URLs:** Stereo version:  
[https://www.lopezmontes.es/archivos/audio/Lopez-Montes-Rudeapoema\\_na\\_penumbra-stereo.mp3](https://www.lopezmontes.es/archivos/audio/Lopez-Montes-Rudeapoema_na_penumbra-stereo.mp3)

### Artistic concept

*Rudepoema* is a magnificent piano composition by the Brazilian composer Heitor Villa-Lobos. Its style embodies a wild virtuosity, and its language might evoke this question: What if Stravinsky had envisioned *The Rite of Spring* in the Amazonian jungle? This electroacoustic piece looks forward to the abundance of sounds and harmonic colors found in Villa-Lobos' work. The goal is to recreate an atmosphere of perceptual saturation, focusing the listener on the macro-scale musical form. The quadraphonic spatialization equally relies on very rapid algorithmic trajectories that shape the space itself, adding textures to the sonic layers. This aesthetic approach is tailored to leverage of the possibilities offered by the latest iteration of GenoMus presented in this thesis.

### Methods

Every audio signal was generated in real-time using SuperCollider, employing various relatively simple synthesis and sampling modules. Event sequences were created using GenoMus, utilizing the current interface, and immediately sent to SuperCollider using OSC. Listing 99 displays a Max-OSC bridge in SuperCollider. Different versions of each sequence were generated using transformation and evolution utilities until achieving the best results. Various *species* with several event extra parameters were used, corresponding to the employed synthesis module. Furthermore, an intermediate layer of fine mapping was added, allowing quick adjustments with a MIDI interface, further streamlining the

workflow. With this working method, the time needed to produce all musical materials was just a few hours.

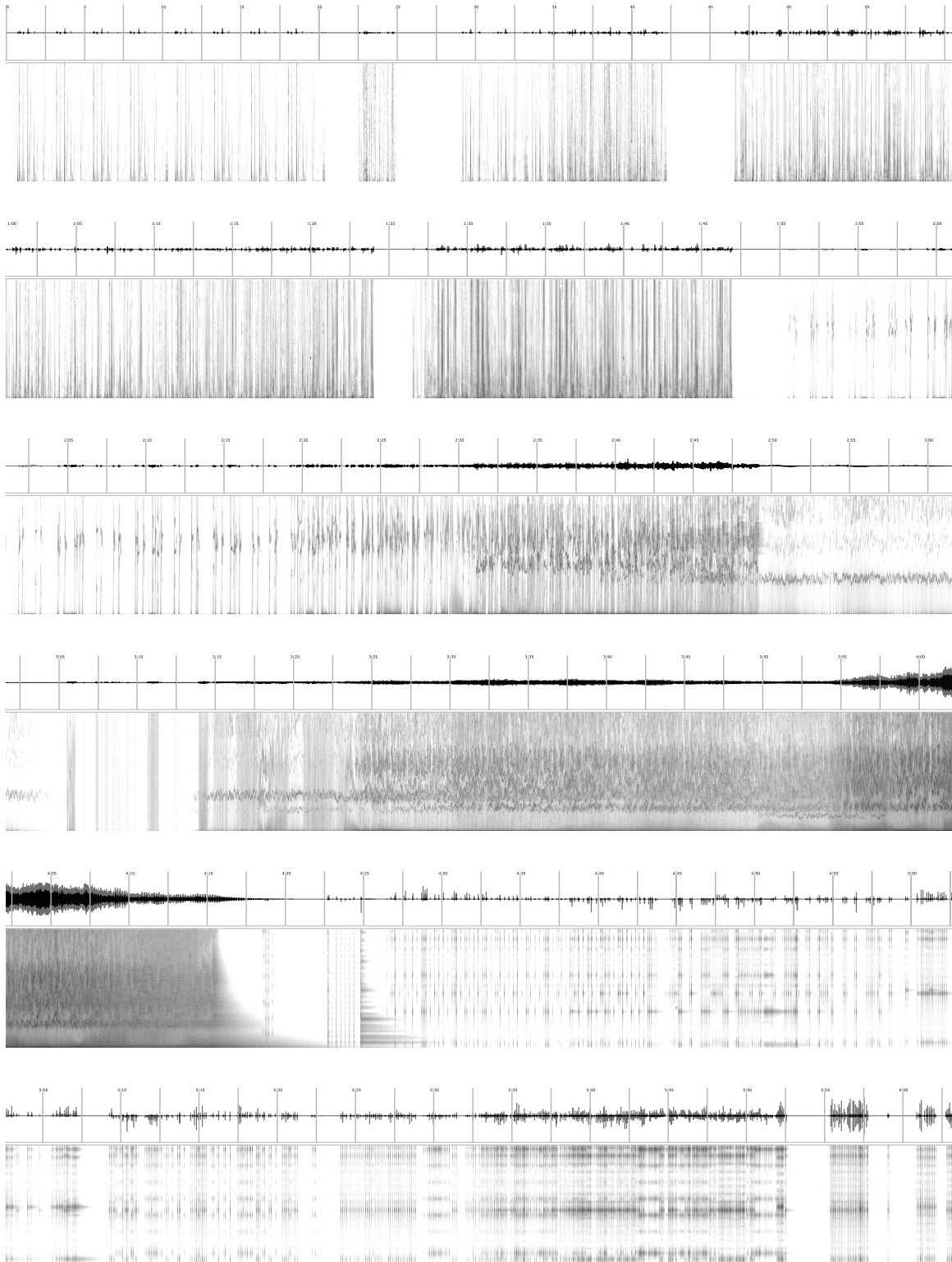
```
1 ( // creates a synth for dense textures based on additive synthesis
2 SynthDef("klankOverlapTexture", {
3     // arguments to be controlled with GenoMus
4     |out = 0,
5     freqs = #[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
6     rings = #[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
7     attack = 0.9, sustain = 8, release = 5, panning = 0|
8     // internal variables
9     var envelope = EnvGen.kr(
10        Env.perc(attackTime: 0.1, releaseTime: sustain, level: 1, curve: -4), doneAction: 2);
11    var inputSignal = Decay.ar(Impulse.ar(0.00001, mul: attack), 0.03, ClipNoise.ar(0.02));
12    var coreSynth = Klank.ar(`[freqs, nil, rings], inputSignal);
13    Out.ar(out, PanAz.ar(4, coreSynth * envelope, panning));
14 }).add;
15 )
16 ( // routes GenoMus events from Max in real-time to play the Synth
17 ~partiels = 20;
18 ~playGenoMusScoreFromMax = { |msg| // this argument receives each event parameters as an
19     array
20     if(msg[0] == 'list') {
21         msg.postln; // posts OSC messages for monitoring
22         Synth("klankOverlapTexture", [ // creates calls for each new received event
23             \attack, msg[5],
24             \sustain, msg[2],
25             \release, msg[4],
26             \panning, msg[6],
27             \freqs, (msg[1] * 0.01).midicps * ((1..~partiels) + {(msg[7].rand)}!~partiels)),
28             \rings, Array.rand(~partiels, 0.1, 1)
29         ]);
30     }
31 };
32 thisProcess.addOSCRecvFunc(~playGenoMusScoreFromMax);
33 );
```

Listing 99: Rudepoema na penumbra — SuperCollider receiving GenoMus data via OSC

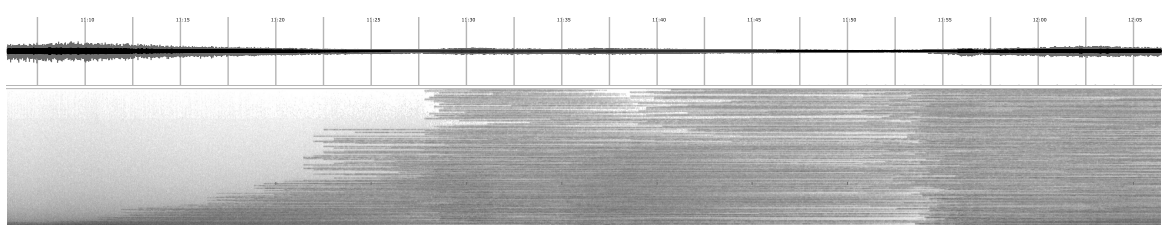
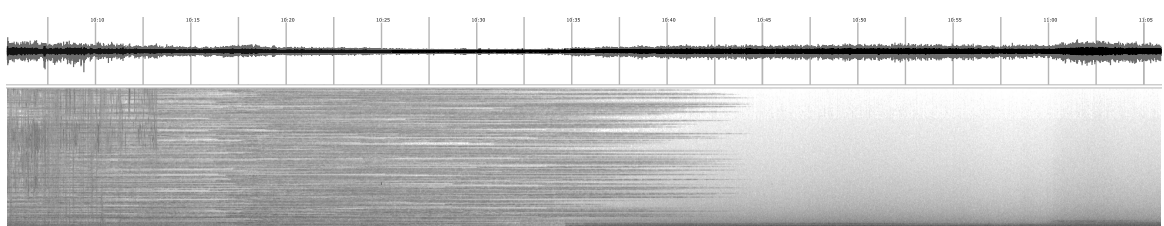
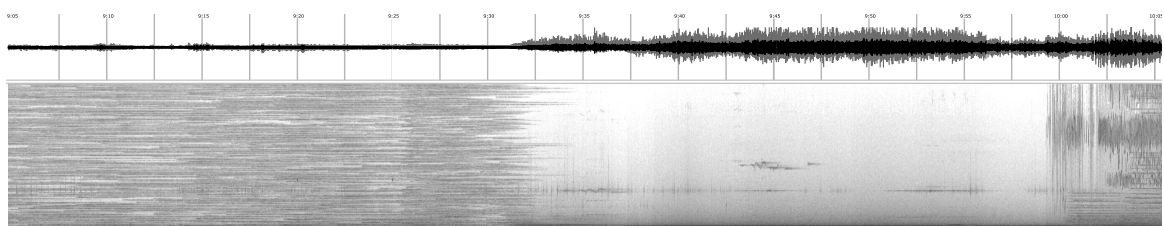
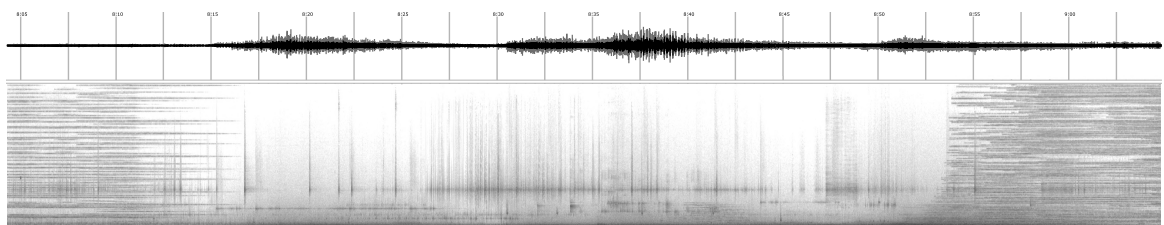
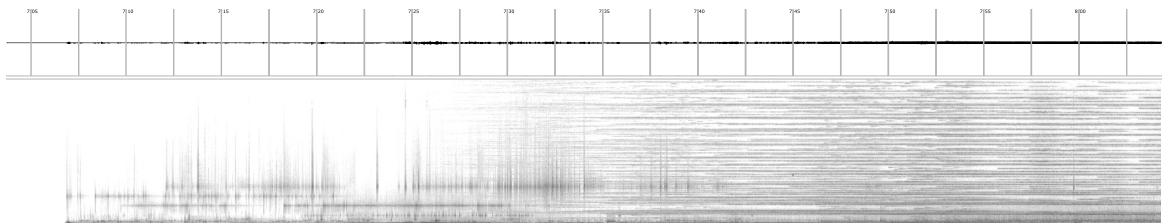
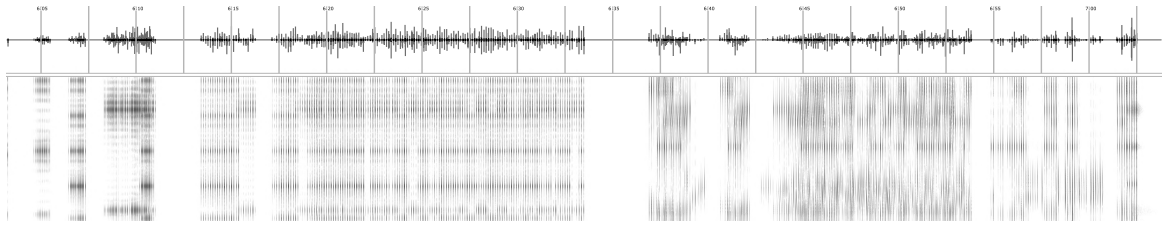
## Waveform and spectrogram

Below is the graphical representation of the entire piece, displaying the waveform (monophonic version) alongside the spectrogram, with a vertical range of up to 20 kHz.

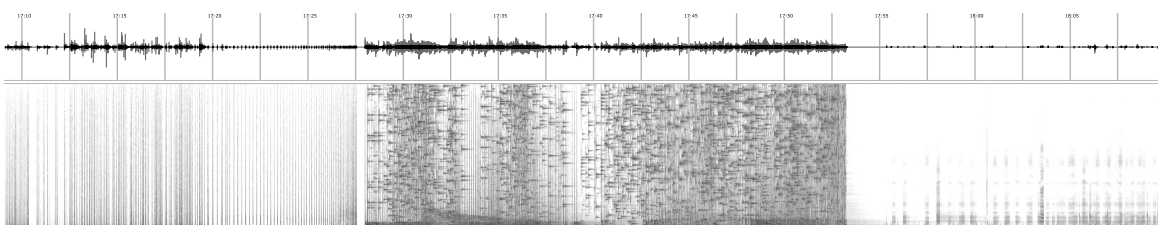
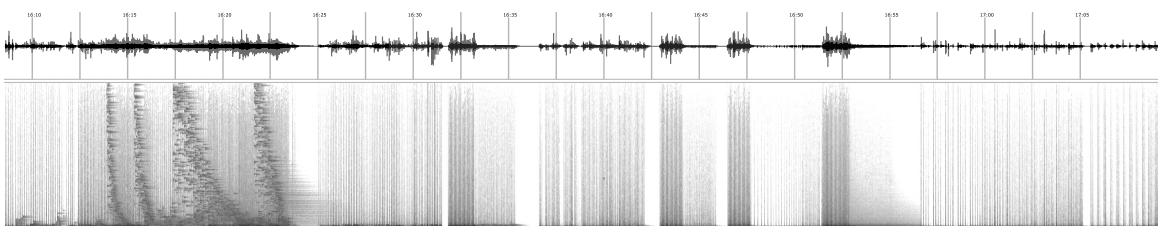
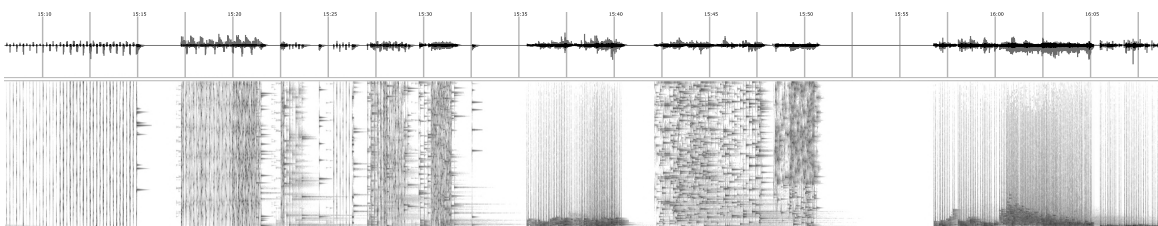
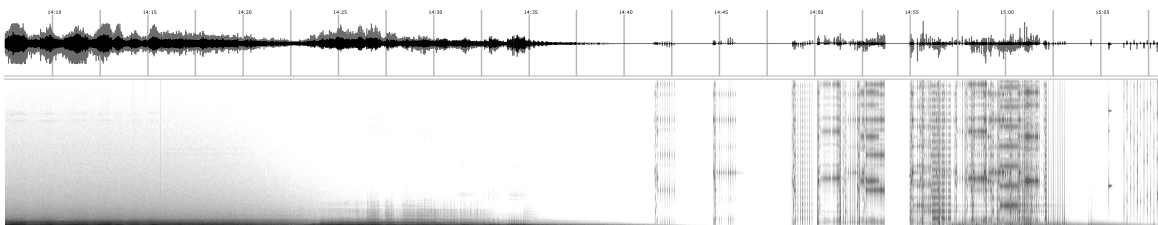
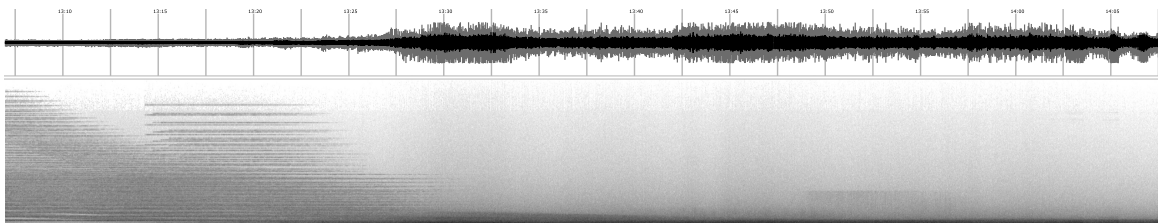
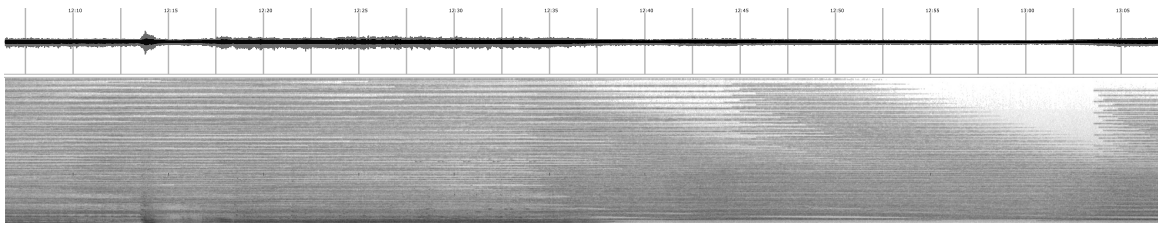
Musical works  
B.9. *Rudepoema na penumbra*



Musical works  
B.9. *Rudepoema na penumbra*



**Musical works**  
*B.9. Rudopoema na penumbra*



Musical works  
11.9. *Rudepoema na penumbra*

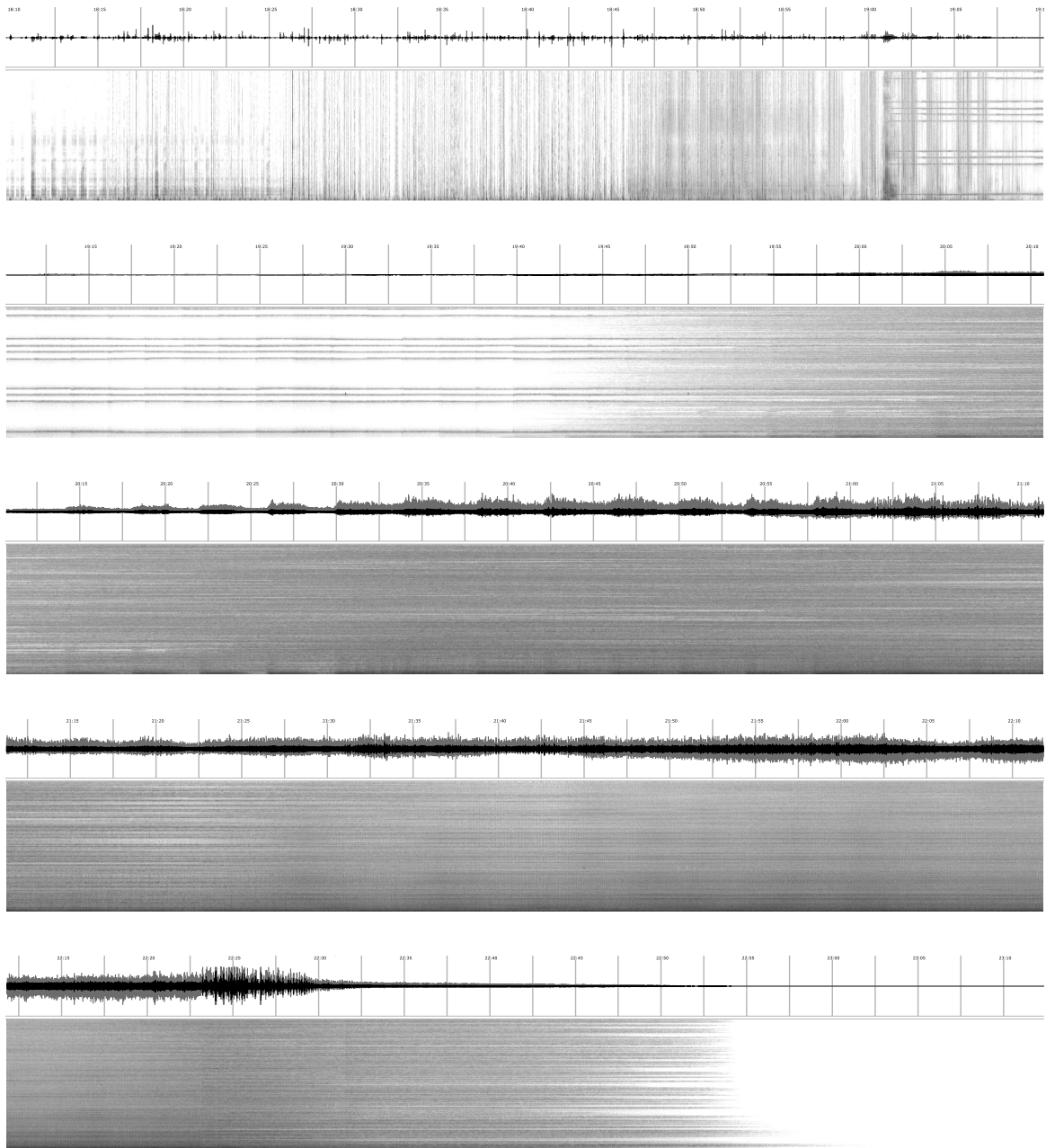


Figure 147: Spectrogram of *Rudepoema na penumbra*

# Bibliography

- [1] A. Agostinelli, T. I. Denk, Z. Borsos, J. Engel, M. Verzetti, A. Caillon, Q. Huang, A. Jansen, A. Roberts, M. Tagliasacchi, et al. MusicLM: Generating music from text. *arXiv preprint arXiv:2301.11325*, 2023.
- [2] A. Agostini and D. Ghisi. Gestures, events and symbols in the bach environment. In *Journées d'Informatique Musicale (JIM 2012)*, pp. 247–255, 2012.
- [3] A. Agostini and D. Ghisi. A Max library for musical notation and computer-aided composition. *Computer Music Journal*, 39(2):11–27, 2015.
- [4] D. D. Albarracín-Molina, A. Raglio, F. Rivas-Ruiz, and F. J. Vico. Using formal grammars as musical genome. *Applied Sciences*, 11(9), 2021.
- [5] A. P. Alivisatos, M. Chun, G. M. Church, R. J. Greenspan, M. L. Roukes, and R. Yuste. The brain activity map project and the challenge of functional connectomics. *Neuron*, 74(6):970–974, 2012.
- [6] M. Allamanis, E. T. Barr, P. Devanbu, and C. Sutton. A survey of machine learning for big code and naturalness. *ACM Computing Surveys*, 51(4), 2018.
- [7] T. Anders. On modelling harmony with constraint programming for algorithmic composition including a model of Schoenberg's theory of harmony. In *Handbook of artificial intelligence for music: foundations, advanced approaches, and developments for creativity*, pp. 283–326. Springer, 2021.
- [8] D. Ando, P. Dahlsted, M. G. Nordahl, and H. Iba. Interactive GP with tree representation of classical music pieces. In *Lecture notes in computer science*, pp. 577–584. Springer Berlin Heidelberg, 2017.
- [9] C. Ariza. *An open design for computer-aided algorithmic music composition: athenaCL*. PhD thesis, New York University, 2005.
- [10] C. Ariza. The interrogator as critic: The turing test and the evaluation of generative music systems. *Computer Music Journal*, 33(2):48–70, 2009.
- [11] Á. Arranz. *La poética del espacio en la creación musical electrónica contemporánea: Orígenes, técnicas y extensiones desde la perspectiva del Instituto de Sonología en La Haya*. PhD thesis, Universidad de Salamanca, 2017.

- [12] L. Atencio. Functional programming in JavaScript. *Manning*, 2019.
- [13] P. Ball. Computer science: Algorithmic rapture, 2012.
- [14] I. Barbancho, A. Rosa-Pujazón, L. J. Tardón, and A. M. Barbancho. Human-computer interaction and music. In *Sound - Perception - Performance*, pp. 367–389. Springer, 2013.
- [15] G. Bennett. Chaos, self-similarity, musical phrase and form. (*Unpublished paper*), 1996.
- [16] B. Berger, M. S. Waterman, and Y. W. Yu. Levenshtein distance, sequence comparison and biological database search. *IEEE Transactions on Information Theory*, 67(6):3287–3294, 2021.
- [17] J. L. Besada. Computer, formalisms, intuition and metaphors. A Xenakian and post-Xenakian approach. In *Proceedings International Computer Music Conference 2014*, pp. 136–141, 2014.
- [18] J. L. Besada. Math and music, models and metaphors: Alberto Posadas’ “tree-like structures”. *Contemporary Music Review*, 38(1-2):107–131, 2019.
- [19] P. Beyls. Aesthetic navigation: Musical complexity engineering using genetic algorithms. In *Journées d’Informatique Musicale*, 1997.
- [20] P. Beyls. Selectionist musical automata: Integrating explicit instruction and evolutionary algorithms. In *IX Brazilian Symposium on Computer Music*. Brazilian Computing Society, 2003.
- [21] R. Bod. The data-oriented parsing approach: Theory and application. In *Computational intelligence: A compendium*, pp. 330–342. Springer Berlin Heidelberg, 2008.
- [22] M. A. Boden. What is creativity? In *Dimensions of creativity*, pp. 75–118. The MIT Press, 1996.
- [23] J. Borg. The Somax 2 software architecture. Technical report, 2021.
- [24] H. Borgdorff. *The conflict of the faculties: Perspectives on artistic research and academia*. Leiden University Press, 2012.
- [25] P. Boulez. Technology and the composer. *Leonardo*, 11(1):59–62, 1978.
- [26] B. G. Buchanan. Creativity at the metalevel: AAAI-2000 Presidential address. *AI Magazine*, 22(3):13, 2001.
- [27] A. R. Burton. *A hybrid neuro-genetic pattern evolution system applied to musical composition*. PhD thesis, University of Surrey, 1998.
- [28] A. R. Burton and T. Vladimirova. Generation of musical sequences with genetic techniques. *Computer Music Journal*, 23(4):59–73, 1999.
- [29] J. Cage. *Silence: Lectures and writings*. Wesleyan University Press, 1961.
- [30] A. Carretero. *El proceso de composición musical a través las técnicas bio-inspiradas de inteligencia artificial. Investigación desde la creación musical*. PhD thesis, Universidad Rey Juan Carlos, 2013.



- [31] T. Catalán. *El compositor Ramón Barce en la música española del siglo XX. Análisis y edición comentada de sus escritos técnicos, estéticos y sociológicos esenciales*. PhD thesis, University of Valencia, 2005.
- [32] M. Cook. Universality in elementary cellular automata. *Complex Systems*, 15(1):1–40, 2004.
- [33] D. Cope. *Virtual music: Computer synthesis of musical style*. The MIT Press, 2004.
- [34] D. Cope. *Computer models of musical creativity*. The MIT Press, 2005.
- [35] R. Crawford. *Algorithmic music composition: A hybrid approach*. Northern Kentucky University, 2015.
- [36] D. Crispin. Artistic research as a process of unfolding. In *Research catalogue: An international database for artistic research*, 2019.
- [37] R. Damaševičius and V. Štuikys. Taxonomy of the fundamental concepts of metaprogramming. *Information Technology and Control*, 37(2), 2008.
- [38] R. B. Dannenberg. The Canon score language. *Computer Music Journal*, 13(1):47–56, 1989.
- [39] F. D. S. De Benedictis. *L’Auditorium che canta 1.0*. Istituto Superiore de Studi Musicali Pietro Mascagni, Livorno, 2015.
- [40] D. de la Motte. *Kontrapunkt. Ein Lese- und Arbeitsbuch*. Bärenreiter, 1981.
- [41] A. O. de la Puente, R. S. Alfonso, and M. A. Moreno. Automatic composition of music by means of grammatical evolution. In *2002 Conference on Array Processing Languages - APL '02*. ACM Press, 2002.
- [42] C. de Lemos Almada. Gödel-vector and Gödel-address as tools for genealogical determination of genetically-produced musical variants. In *Computational Music Science*, pp. 9–16. Springer, 2017.
- [43] K. Déguernel, E. Vincent, and G. Assayag. Using multidimensional sequences for improvisation in the OMax paradigm. In *13th Sound and Music Computing Conference*, 2016.
- [44] M. Delgado, W. Fajardo, and M. Molina-Solana. Inmamusys: Intelligent multiagent music system. *Expert Systems with Applications*, 36(3, Part 1):4574–4580, 2009.
- [45] A. Di Scipio. *Pensare le tecnologie del suono e della musica*. Editoriale scientifica, 2013.
- [46] A. Díaz de la Fuente. Una mirada a Cripsis de Alberto Posadas. In *Des\_Ar. Investigar desde el Arte*, pp. 173–185, 2011.
- [47] G. Díaz-Jerez. Composing with Melomics: Delving into the computational world for musical inspiration. *Leonardo*, 21:13–14, 2011.
- [48] P. Doornbusch. *Mapping in algorithmic composition and related practices*. RMIT University, Victoria, 2010.

- [49] M. Dostál. Evolutionary music composition. In *Handbook of optimization*, pp. 935–964. Springer Berlin Heidelberg, 2013.
- [50] F. Drewes and J. Högborg. An algebra for tree-based music generation. In *2nd International Conference on Algebraic Informatics, Lecture notes in computer science*, volume 4728, pp. 172–188, 2007.
- [51] B. Eckel and L. O’Brien. *Thinking in C#*. Prentice Hall, 2002.
- [52] L. Eibensteiner. *Polyphonic music composition with grammars*. PhD thesis, Technischen Universität Wien, 2021.
- [53] J. D. Fernández and F. Vico. AI methods in algorithmic composition: A comprehensive survey. *Journal of Artificial Intelligence Research*, 48:513–582, 2013.
- [54] M. Fiorini and M. Malt. Somax2—A distributed co-creative system for human-machine co-improvisation. In *HHAI 2023: Augmenting Human Intellect*, pp. 389–391. IOS Press, 2023.
- [55] D. Formaggio. *L’arte: Come idea e come esperienza*. Mondadori, 1990.
- [56] P. Galanter. Computational aesthetic evaluation: Past and future. In *Computers and creativity*, pp. 255–293. Springer Berlin Heidelberg, 2012.
- [57] R. García-Benito and E. Pérez-Montero. Painting graphs with sounds: Cosmonic sonification project. *Revista Mexicana de Astronomía y Astrofísica*, (54):28–33, 2022.
- [58] C. Hernández-Oliván and J. R. Beltrán. Music composition with deep learning: A review. In *Advances in speech and music technology: Computational aspects and applications*, pp. 25–50. Springer, 2023.
- [59] C. Hernández-Oliván, J. Hernández-Oliván, and J. R. Beltrán. A survey on artificial intelligence for music generation: Agents, domains and perspectives. *arXiv preprint arXiv:2210.13944*, 2022.
- [60] D. Herremans, C.-H. Chuan, and E. Chew. A functional taxonomy of music generation systems. *ACM Computing Surveys*, 50(5):1–30, 2017.
- [61] M. Hervás. Música en los museos: el caso del Guggenheim de Bilbao. *Espacio Tiempo y Forma. Serie VII, Historia del Arte*, (10):255–276, 2022.
- [62] P. Hindemith. *The craft of musical composition*. Schott, 1942.
- [63] D. M. Hofmann. A genetic programming approach to generating musical compositions. In *Evolutionary and biologically inspired music, sound, art and design*, pp. 89–100. Springer, 2015.
- [64] D. M. Hofmann. Introducing a context-based model and language for representation, transformation, visualization, analysis and generation of music. In *Proceedings of the International Computer Music Conference*, p. 381, 2016.

- [65] D. M. Hofmann. *Music processing suite: A software system for context-based symbolic music representation, visualization, transformation, analysis and generation*. PhD thesis, University of Music Karlsruhe, 2018.
- [66] D. R. Hofstadter. *Gödel, Escher, Bach: an eternal golden braid*. Basic Books Inc., 1979.
- [67] S. Holland and R. Fiebrink. Machine learning, music and creativity: An interview with Rebecca Fiebrink. In *New directions in music and human-computer interaction*, pp. 259–267. Springer, 2019.
- [68] Z. Hu, X. Ma, Y. Liu, G. Chen, Y. Liu, and R. B. Dannenberg. The beauty of repetition: an algorithmic composition model with motif-level repetition generator and outline-to-music generator in symbolic music generation. *IEEE Transactions on Multimedia*, pp. 1–14, 2023.
- [69] P. Hudak and D. Quick. *The Haskell school of music: From signals to symphonies*. Cambridge University Press, 2018.
- [70] J. Hughes. *Why functional programming matters*. Programming Methodology Group, Chalmers University of Technology / Göteborg University, 1984.
- [71] B. L. Jacob. Composing with genetic algorithms. In *International Computer Music Conference*, 1995.
- [72] B. L. Jacob. Algorithmic composition as a model of creativity. *Organised Sound*, 1(3):157–165, 1996.
- [73] G. M. Koenig. Working with “Project 1” my experiences with computer composition. *Interface*, 20(3-4):175–180, 1991.
- [74] A. Kontogeorgakopoulos. Music, art installations and haptic technology. *Arts*, 12(4), 2023.
- [75] P. Laine and M. Kuuskankare. Genetic algorithms in musical style oriented generation. In *First IEEE Conference on Evolutionary Computation*. IEEE World Congress on Computational Intelligence, 1994.
- [76] J. Lee-Thorp, J. Ainslie, I. Eckstein, and S. Ontanon. FNet: Mixing tokens with Fourier transforms. In *Proceedings of the 2022 Conference of the North American Chapter of the Association for Computational Linguistics: Human language technologies*, pp. 4296–4313, 2022.
- [77] F. Lerdahl and R. Jackendoff. *A generative theory of tonal music*. The MIT Press, 1983.
- [78] B. R. Levy. *The electronic works of György Ligeti and their influence on his later style*. University of Maryland, College Park, 2006.
- [79] D. Lobo. *Evolutionary development based on genetic regulatory models for behavior-finding*. PhD thesis, Universidad de Málaga, 2014.
- [80] M. Longair. Revolutions in music and physics, 1900–30. *Interdisciplinary Science Reviews*, 31(3):275–288, 2006.

- [81] A. López and J. Munarriz. *El Centro de Cálculo de la Universidad de Madrid (1968-1973): ciencia, arte y creación computacional*. Ediciones Complutense, 2021.
- [82] R. López de Mántaras. Making music with AI: some examples. In *2006 Conference on Rob Milne: A tribute to a pioneering AI scientist, entrepreneur and mountaineer*, pp. 90–100. IOS Press, 2006.
- [83] J. López-Montes. Ada + Babbage – Capricci, for cello and piano. *Espacio Sonoro*, (35), 2015.
- [84] J. López-Montes. Genomus como aproximación a la creatividad asistida por computadora. *Espacio Sonoro*, (35), 2015.
- [85] J. López-Montes. Microcontrapunctus: metaprogramación con GenoMus aplicada a la síntesis de sonido. *Espacio Sonoro*, (48), 2016.
- [86] J. López-Montes and P. Miralles. Tiento: creatividad artificial con GenoMus para la composición colaborativa de música electrónica. In *FACBA'21: Seminario "La variación infinita"*, pp. 62–69. Ed. Universidad de Granada, 2021.
- [87] J. López-Montes, M. Molina-Solana, and W. Fajardo. Genomus: Representing procedural musical structures with an encoded functional grammar optimized for metaprogramming and machine learning. *Applied Sciences*, 12(16), 2022.
- [88] O. López-Rincón, O. Starostenko, and G. A.-S. Martín. Algorithmic music composition based on artificial intelligence: A survey. In *2018 International Conference on Electronics, Communications and Computers*. IEEE, 2018.
- [89] R. Loughran and M. O'Neill. Generative music evaluation: Why do we limit to 'human'? In *Proceedings of the first Conference on Computer Simulation of Musical Creativity*, 2016.
- [90] R. Maconie. *The concept of music*. Clarendon Press, 1990.
- [91] B. Mandelbrot. *The fractal geometry of nature*. W. H. Freeman and Co., 1982.
- [92] M. Mannone, F. Thalmann, M. Rahaim, J. Ho, M. Mannone, A. Lubet, R. Guitart, and G. Mazzola. *The topos of music III: Gestures – Musical multiverse ontologies*. Springer, 2017.
- [93] S. Manousakis. *Musical L-systems*. Master's thesis, Koninklijk Conservatorium, The Hague, 2006.
- [94] C. Martín. *La música visual del compositor José López-Montes*. Master's thesis, Universidad de Oviedo, Universidad de Granada y Universidad Internacional de Andalucía, 2020.
- [95] G. Mazzola. *The topos of music I: Theory – Geometric logic, classification, harmony, counterpoint, motives, rhythm*. Springer, 2002.
- [96] J. McCormack. Open problems in evolutionary music and art. In *Applications of evolutionary computing, EvoWorkshops 2005: EvoBIO, EvoCOMNET, EvoHOT, EvoIASP, EvoMUSART, and EvoSTOC*, pp. 428–436, 2005.

- [97] D. Mertz. *Functional programming in Python*. O'Reilly Media, 2015.
- [98] O. Messiaen. *Technique de mon langage musical*. Alphonse Leduc, 1944.
- [99] L. B. Meyer. *Style and music: theory, history, and ideology*. University of Chicago Press, 1996.
- [100] Z. Mheich, L. Wen, P. Xiao, and A. Maaref. Design of SCMA codebooks based on golden angle modulation. *IEEE Transactions on Vehicular Technology*, 68(2):1501–1509, 2019.
- [101] G. Michaelson. *An introduction to functional programming through Lambda calculus*. Courier Corporation, 2011.
- [102] M. Minsky. Why people think computers can't. *AI Magazine*, 3(4):3–15, 1982.
- [103] E. R. Miranda. Genetic music system with synthetic biology. *Artificial Life*, 26(3):366–390, 2020.
- [104] E. R. Miranda. *Handbook of artificial intelligence for music: foundations, advanced approaches, and developments for creativity*. Springer, 2021.
- [105] E. R. Miranda and J. Castet. *Guide to brain-computer music interfacing*. Springer, 2014.
- [106] E. R. Miranda and H. Miller-Bakewell. Cellular automata music composition: From classical to quantum. In *Quantum computer music: Foundations, methods and advanced concepts*, pp. 105–130. Springer, 2022.
- [107] E. R. Miranda and H. Shaji. Generative music with partitioned quantum cellular automata. *Applied Sciences*, 13(4), 2023.
- [108] G. Nierhaus. *Algorithmic composition: Paradigms of automated music generation*. Springer, 2008.
- [109] S. Ninagawa.  $1/f$  Noise in elementary cellular automaton rule 110. In *Unconventional computation*, pp. 207–216. Springer Berlin Heidelberg, 2006.
- [110] T. Oliwa and M. Wagner. Composing music with neural networks and probabilistic finite-state machines. In *Applications of evolutionary computing: EvoWorkshops 2008: EvoCOMNET, EvoFIN, EvoHOT, EvoIASP, EvoMUSART, EvoNUM, EvoSTOC, and EvoTransLog*, pp. 503–508. Springer, 2008.
- [111] D. C. Ong. Quasiperiodic music. *Journal of Mathematics and the Arts*, 14(4):285–296, 2020.
- [112] B. Ostash and M. Anisimova. Visualizing codon usage within and across genomes: Concepts and tools. In *Statistical modelling and machine learning principles for bioinformatics techniques, tools, and applications*, pp. 213–288. Springer Singapore, 2020.
- [113] J. Palamara and W. S. Deal. A dynamic representation solution for machine learning-aided performance technology. *Frontiers in Artificial Intelligence*, 3, 2020.

- [114] G. Papadopoulos and G. Wiggins. AI methods for algorithmic composition: A survey, a critical view and future prospects. In *AISB Symposium on Musical Creativity*, pp. 110–117, 1999.
- [115] M. Pearce, D. Meredith, and G. Wiggins. Motivations and methodologies for automation of the compositional process. *Musicae Scientiae*, 6(2):119–147, 2002.
- [116] M. Pedregosa. *GenoMus: Rediseño y desarrollo. Implementación de un motor de cómputo funcional basado en prototipos sobre estructuras musicales*. Bachelor’s thesis, Universidad de Granada, 2022.
- [117] J. Reddin, J. McDermott, and M. O’Neill. Elevated pitch: Automated grammatical evolution of short compositions. In *Applications of evolutionary computing*, pp. 579–584. Springer Berlin Heidelberg, 2009.
- [118] G. Revesz. *Lambda-calculus combinators and functional programming*. Oxford University Press, 1988.
- [119] F.-R. Rideau. Métaprogrammation et libre disponibilité des sources. In *Actes de la conférence «Autour du Libre 1999»*, 1999.
- [120] J.-C. Risset. The liberation of sound, art-science and the digital domain: Contacts with Edgard Varèse. *Contemporary Music Review*, 23(2):27–54, 2004.
- [121] C. Roads. Composing grammars. In *International Conference on Mathematics and Computing*, 1977.
- [122] C. Roads. *Microsound*. The MIT Press, 2004.
- [123] C. Roads and P. R. Wieneke. Grammars as representations for music. *Computer Music Journal*, 3:48, 1979.
- [124] C. Roig, L. J. Tardón, I. Barbancho, and A. M. Barbancho. A non-homogeneous beat-based harmony markov model. *Knowledge-Based Systems*, 142:85–94, 2018.
- [125] V. Rolla, P. Riera, P. Souza, J. Zubelli, and L. Velho. Self-similarity of classical music networks. *Fractals*, 29(02):2150041, 2021.
- [126] B. Romera-Paredes, M. Barekatin, A. Novikov, M. Balog, M. P. Kumar, E. Dupont, F. J. Ruiz, J. S. Ellenberg, P. Wang, O. Fawzi, et al. Mathematical discoveries from program search with large language models. *Nature*, pp. 1–3, 2023.
- [127] J. Romero. *Metodología evolutiva para la construcción de modelos cognitivos complejos. Exploración de la creatividad artificial en composición musical*. PhD thesis, Universidade da Coruña, 2002.
- [128] J. Rowe and D. Partridge. Creativity: a survey of AI approaches. *Artificial Intelligence Review*, 7(1):43–70, 1993.
- [129] S. Russomanno. *La música invisible. En busca de la armonía de las esferas*. Fórcola Ediciones, 2017.

- [130] C. Sánchez-Quintana, F. Moreno-Arcas, D. Albarracín-Molina, J. D. Fernández, and F. J. Vico. Melomics: A case-study of AI in Spain. *AI Magazine*, 34(3):99, 2013.
- [131] Ö. Sandred. Constraint-solving systems in music creation. In *Handbook of artificial intelligence for music: Foundations, advanced approaches, and developments for creativity*, pp. 327–344. Springer, 2021.
- [132] E. Satie. *Mémoires d'un amnésique*. Ombres, 2010 (Original work written in 1912).
- [133] C. Satué. Arquitecturas musicales desarrolladas con técnicas fractales. *Matematicalia*, 2(5), 2007.
- [134] A. Schaathun. Formula-composition modernism in music made audible. In *Inspirator – Tradisjonsbærer – Rabulist*, pp. 132–147. Norsk Musikforlag, 1996.
- [135] J. Schacher, C. Eck, K. Reese, and T. Lossius. sonozones. Sound art investigations in public places. *Journal for Artistic Research*, 2014, 09 2014.
- [136] J. Schmidhuber. Low-complexity art. *Leonardo*, 30(2):97–103, 1997.
- [137] J. Schmidhuber. Gödel machines: Fully self-referential optimal universal self-improvers. In *Artificial general intelligence*, pp. 199–226. Springer, 2007.
- [138] A. Schönberg. *Harmonielehre*. Universal-Edition, 1911.
- [139] J. Searle. *Freedom and neurobiology: Reflections on free will, language, and political power*. Columbia University Press, 2006.
- [140] J. Shao, J. McDermott, M. O'Neill, and A. Brabazon. Jive: A generative, interactive, virtual, evolutionary music system. In *Applications of evolutionary computation*, pp. 341–350. Springer Berlin Heidelberg, 2010.
- [141] K. Singh and D. Malhotra. Meta-health: Learning-to-learn (meta-learning) as a next generation of deep learning exploring healthcare challenges and solutions for rare disorders: A systematic analysis. *Archives of Computational Methods in Engineering*, pp. 1–32, 2023.
- [142] J. Sloboda. *Exploring the musical mind: Cognition, emotion, ability, function*. PhD thesis, 2005.
- [143] L. Spector and A. Alpern. Induction and recapitulation of deep musical structure. In *IJCAI-95 Workshop on artificial intelligence and music*, pp. 41–48, 2019.
- [144] K. O. Stanley and R. Miikkulainen. A taxonomy for artificial embryogeny. *Artificial Life*, 9(2):93–130, 2003.
- [145] T. Stilson and J. O. Smith III. Alias-free digital synthesis of classic analog waveforms. In *ICMC*, 1996.
- [146] K. Stockhausen and M. Tannenbaum. *Conversations with Stockhausen*. Clarendon Press, 1987.
- [147] I. Stravinsky. *Poetics of music in the form of six lessons*. Harvard University Press, 1970.

- [148] C. Sulyok, C. Harte, and Z. Bodó. On the impact of domain-specific knowledge in evolutionary music composition. In *Genetic and Evolutionary Computation Conference*. ACM Press, 2019.
- [149] H. Taube. Common Music: A music composition language in Common Lisp and CLOS. *Computer Music Journal*, 15(2):21–32, 1991.
- [150] P. J. Vayón. Entre la armonía y el ruido. *Diario de Sevilla*, December 12th, 2012.
- [151] F. Vico, D. Albarracín-Molina, G. Díaz-Jerez, and L. Manzoni. Automatic music composition with evolutionary algorithms: Digging into the roots of biological creativity. In *Handbook of artificial intelligence for music: Foundations, advanced approaches, and developments for creativity*, pp. 455–483. Springer, 2021.
- [152] J. von Neumann. *The collected works of John von Neumann*. Reader’s Digest Young Families, 1963.
- [153] R. F. Voss and J. Clarke. “1/fnoise” in music and speech. *Nature*, (258):317–318, 1975.
- [154] S. Wang, Z. Bao, and J. E. Armor: A benchmark for meta-evaluation of artificial music. In *Proceedings of the 29th ACM International Conference on Multimedia*, MM ’21, page 5583–5590. Association for Computing Machinery, 2021.
- [155] S. Wilson. *An aesthetics of past-present relations in the experience of late 20th –and early 21st–century art music*. PhD thesis, University of London, 2013.
- [156] S. Wolfram. *A new kind of science*. Wolfram Media, 2002.
- [157] R. Wooller, A. R. Brown, E. R. Miranda, J. Diederich, and R. Berry. A framework for comparison of process in algorithmic music systems. In *Generative Arts Practice*, pp. 109–124. Creativity and Cognition Studios, 2005.
- [158] I. Xenakis. *Formalized music: Thought and mathematics in composition*. Indiana University Press, 1971.
- [159] L. M. Zbikowski. *Foundations of musical grammar*. Oxford University Press, 2017.