

UNIVERSITATEA DIN
BUCUREȘTI

FACULTATEA DE
MATEMATICĂ ȘI
INFORMATICĂ



SPECIALIZAREA INFORMATICĂ

Lucrare de licență

Generare MIDI folosind algoritmi genetici

Absolvent

Gheorghe Andrei

Coordonator științific

Lect. Dr. Sergiu Nisioi

București, Septembrie 2022

Rezumat

Domeniul creativității artificiale a propus sisteme de compoziție algoritmică capabile să producă rezultate impresionante. Totuși, majoritatea prezintă modele autonome, care omit interacțiunea umană din implementare. Ca rezultat, există puține unelte bazate pe compoziție algoritmică accesibile consumatorului obișnuit. Obiectivul acestei lucrări este implementarea unui sistem interactiv de unelte bazate pe algoritmi genetici și operatorii folosiți în cadrul acestora. Produsul final este împachetat sub forma unui audio plugin pentru a putea fi folosit într-un DAW și este astfel mai accesibil și ușor de folosit de către muzicieni.

Abstract

The field of computational creativity has proposed algorithmic composition systems capable of producing impressive results. However, most of them are based on autonomous models, which function without ongoing human intervention. Consequently, there are few examples of algorithmic composition tools accessible to musicians. This paper proposes an interactive system driven by genetic algorithms and the genetic operators they derive. The final product is packaged as an audio plugin and can be opened by most DAWs, resulting in better accessibility and ease of use.

Cuprins

1	Introducere	5
1.1	Motivație	6
1.2	Implementări existente	7
1.3	Contribuția personală	7
2	Preliminarii	9
2.1	Elemente de teorie muzicală	9
2.1.1	Armonie	9
2.1.2	Ritm	11
2.1.3	Compoziție	14
2.2	Metode de măsurare a nivelului de sincopare	14
2.2.1	Gómez "Weighted Note-to-Beat"	15
2.2.2	Toussaint "Off-Beatness"	16
2.3	Formatul MIDI	16
2.4	Audio plugin	18
3	Arhitectura aplicației	19
3.1	Structură	19
3.2	Librării	20
3.3	Build tools	20
3.4	Testare	21
4	Implementarea plugin-ului audio	23
4.1	B2bAIPluginProcessor	23

4.2	Piano roll	24
4.3	Sistemul de fişiere	25
4.4	Parametrii audio	26
5	Modulul midi-generator	28
5.1	Algoritmul genetic	28
5.1.1	Codificare	28
5.1.2	Mutaţie	29
5.1.3	Fitness	29
5.2	API	30
6	Concluzii	32
6.1	Limitări ale aplicaţiei	32
6.2	Posibile dezvoltări ulterioare	32
Anexe		
A	Header files	37
B	Referinţe cod	42

1 Introducere

Compoziția algoritmică este o tehnică folosită frecvent de către muzicieni. Există diverse metode de a folosi compoziția algoritmică în procesul de creație muzicală: procesul poate să fie neasistat (compoziția este în total creată în mod algoritmic fără intervenția unui muzician), sau poate fi centrat în jurul componentei umane; datele generate pot acționa direct asupra undelor sonore, sau pot fi complet abstractizate de modul în care este generat sunetul, precum este generarea algoritmică de partituri muzicale.

În această lucrare voi analiza compoziția algoritmică ca unealtă asistivă în procesul de creație uman și voi încerca să implementez o interfață care să permită generarea parametrizată de secvențe muzicale scurte în format MIDI. În continuarea acestui capitol voi argumenta motivul pentru care consider importantă studierea compoziției algoritmice ca procedeu stilistic și voi prezenta câteva dintre metodele actuale de implementare.

Voi începe cel de-al doilea capitol prin a explica câteva noțiuni fundamentale din teoria muzicală clasică, urmând ca apoi să prezint metode matematice de modelare a noțiunilor prezentate. În continuare voi descrie câteva particularități ale formatului digital MIDI și ale extensiilor folosite în domeniul muzical digital.

Capitolul 3 al lucrării va conține prezentarea structurii implementării pe care o propun, precum și uneltele pe care le-am folosit pentru a configura, testa și construi aplicația.

Al patrulea capitol explică implementarea componentei grafice. Aici voi ilustra modul în care am utilizat componenta teoretică în cadrul implementării, precum și modul de utilizare și funcționare a plugin-ului. Cel de-al 5-lea capitol al lucrării

prezintă implementarea și utilizarea algoritmilor genetici în cadrul compoziției algoritmice.

În final voi analiza limitările implementării prezentate, precum și modul în care aceasta poate fi îmbunătățită. Lucrarea este însoțită de două anexe, prima conține prototipurile claselor din cadrul aplicației, a doua conține fragmente de cod folosite în implementarea componentelor și tehnologiilor diverse utilizate.

1.1 Motivație

Compoziția algoritmică în cadrul creației muzicale este o tehnică utilizată frecvent, care își are originile într-o oarecare formă în antichitate. Conceptul de a utiliza instrucțiuni și procese formale în cadrul compoziției muzicale își are originile în sistemul muzical utilizat în Grecia Antică [19], [13]. Există diferite sisteme muzicale concepute în aceasta perioadă, precum sistemul de acordare Pitagorean, un algoritm care construiește o scară muzicală între două note muzicale cu rația frecvențelor 1:2. Totuși, sistemele de compoziție algoritmică nu au putut să crească în complexitate semnificativ până la apariția sistemelor automate de calcul. Astfel, apariția calculatoarelor și a instrumentelor electronice a facilitat apariția unor noi dimensiuni muzicale, iar procesul de creație muzicală s-a schimbat fundamental. Spre exemplu, introducerea sintetizatoarelor de sunet, instrumente capabile de a genera și reda un număr nelimitat de frecvențe sonore a introdus dimensiunea complet nouă în procesul compozițional. În prezent, procesul de creație muzicală transcede regulilor uzuale definite de teoria muzicală și nu mai este în mod necesar orientat în jurul lor. Totuși, acestea încă reprezintă în general fundația peste care este construită o compoziție. Consider astfel că automatizarea acestor procese ar facilita o libertate de explorare mai mare

în cadrul procesului compozițional, în special în rândul artiștilor neexperimentați.

1.2 Implementări existente

Există numeroase modele folosite în implementarea sistemelor de compoziție algoritmică. Multe dintre acestea tratează problema prezentată ca pe o problemă de optimizare, precum modelele Markov [14], modelele bazate pe învățare ranforsată [7], sau modelele evoluționare [6], [11]. Alte exemple de modele folosite sunt modelele matematice și modelele translaționale. În implementarea propusă am ales folosirea unui model evoluționar, deoarece consider că operatorii genetici pot fi folosiți și în implementarea altor funcționalități utile în creația muzicală. Pe lângă implementările teoretice există și produse comerciale implementate:

- **Magenta Studio** - dezvoltat de Magenta, este o colecție de unelte muzicale implementată folosind modelele open-source de machine learning dezvoltate de companie [17]. Exemple de funcționalități incluse în cadrul colecției sunt *Generate*, *Interpolate*, sau *Drumify*.
- **Bassline Studio** - [20] dezvoltat de Reason Studios, este un sequencer folosit pentru a genera secvențe monofonice de note *reason*. Acesta este un plugin ”proprietary”, însă metoda prin care generează secvențe muzicale pare implementată folosind un model matematic deterministic.

1.3 Contribuția personală

Contribuția personală adusă în cadrul acestei lucrări constă în implementarea unui plugin audio și al unui modul pe python pentru manipularea și generarea automată

de fişiere MIDI. Codul sursă al plugin-ului este disponibil pe [GitHub](#). Modulul de python este publicat prin [PyPi](#), iar codul sursă al acestuia este disponibil într-un alt repository de [GitHub](#).

2 Preliminarii

2.1 Elemente de teorie muzicală

Pentru a prezenta funcționalitățile aplicației este necesară definirea câtorva termeni care descriu aspecte fundamentale ale melodiei.

2.1.1 Armonie

Armonia reprezintă procesul prin care sunete individuale sunt aranjate în unități individuale. Fiecare notă muzicală are asociată o anumită frecvență, reprezentând numărul de oscilații produse într-o secundă de semnalul sonor obținut prin redarea sa. Raportul dintre frecvența a două note muzicale corespunde nivelului de *consonanță* dintre cele două. Consonanța este atât un criteriu fizic cât și un fenomen psihologic: două note sunt considerate consonante dacă redate împreună formează un sunet plăcut. Astfel, o secvență muzicală este considerată *armonioasă* dacă notele muzicale din care este compusă sunt consonante.

Gamă

Intervalul cuprins între două note cu rația frecvențelor de $2 : 1$ corespunde unei *octave* și reprezintă baza din care se construiește fiecare *gamă muzicală*. Astfel, problema construcției unei game reprezintă găsirea unei mulțimi de note consonante în acest interval [13]. În general, compozițiile muzicale sunt scrise folosind predominant note aparținând unei singure game pentru a asigura caracterul armonios. Gamele muzicale pot fi clasificate în funcție de numărul de note:

- **Cromatic, sau dodecatonic** - 12 note
- **Nonatonic** - 9 note

- **Octatonic** - 8 note
- **Heptatonic** - 7 note
- **Hexatonic** - 6 note, etc.

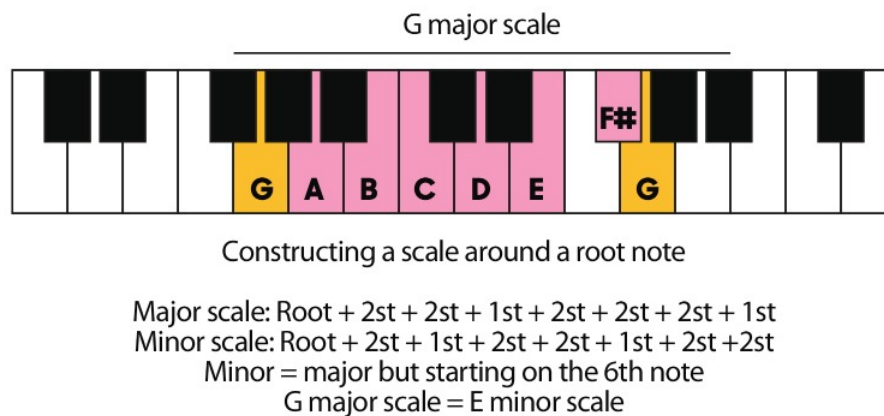


Figura 2.1: Construirea unei game în jurul unei note

Mod

Există mai multe *moduri* în care poate fi construită o gamă, în funcție distanța în frecvență dintre note. Această distanță se calculează în *tonuri* (*t*) și *semitonuri* (*s*). În continuare, voi utiliza doar game heptatonice în următoarele moduri:

- **Aeolian (minor)**: t-s-t-t-s-t-t
- **Locrian**: s-t-t-s-t-t-t
- **Ionian: (major)** t-t-s-t-t-t-s
- **Dorian**: t-s-t-t-t-s-t
- **Phrygian**: s-t-t-t-s-t-t
- **Lydian**: t-t-t-s-t-t-s
- **Mixolydian**: t-t-s-t-t-s-t

Acord

Un acord este o combinație de note consonante redade simultan. Cele mai întâlnite acorduri sunt triadele, compuse din 3 note aparținând aceleiași game muzicale.

Acord	Note	Notăție
1	G B D	I
2	A C E	ii
3	B D F#	iii
4	C E G	IV
5	D F# A	V
6	E G B	vi
7	F# A C	vii

Tabela 1: Triadele diatonice în G major

O înșiruire de acorduri se numește progresie de acorduri. În general, progresiile de acorduri reprezintă fundația armoniei într-o piesă muzicală.

2.1.2 Ritm

Ritmul reprezintă alternarea recurentă și simetrică între elemente distincte. Ticurile unui ceas sunt un exemplu concret de ritm. Ticurile ceasului sunt evenimente despărțite în timpi egali și sunt percepute diferit (tic-tac), deși sunt identice. Gruparea impulsurilor (tic-tac) formează nivele noi de periodicitate, care compun o structură pe mai multe nivele, numită *metru*.

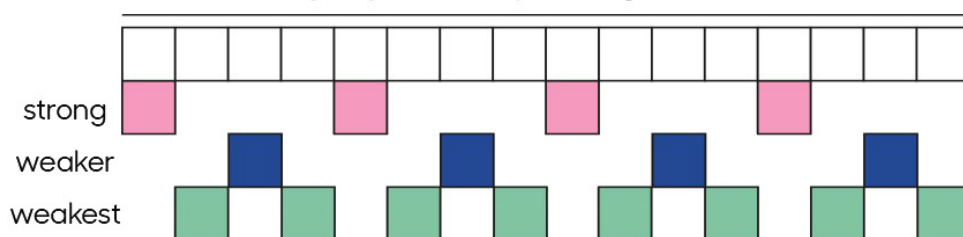


Figura 2.2: Exemplu de metru

Alternarea periodică între *impulsuri (beats) puternice* și *impulsuri slabe* este fundamentală pentru ideea de metru [5]. Impulsurile puternice nu diferă într-un sens fizic (e.g. frecvență, durată, amplitudine) de cele slabe, dar sunt percepute ca fiind fundamentale. Gruparea pulsurilor în grupuri cu număr non-prim de elemente pot fi subdivizate în subgrupuri de pulsuri egale ca dimensiune. Astfel, se crează o structură metrică pe mai multe nivele [16]. În figura 2.2 sunt ilustrate nivelele metrice ale unei grupări cu 4 elemente. Pulsurile marcate cu roz se numesc *downbeats*, cele marcate cu verde *upbeats*, iar cele verzi *backbeats*.

Notă

În teoria muzicală vestică, un eveniment sonor este denumit *notă*, în timp ce un eveniment silențios este numit *pauză*. Fiecărei note sau pauze îi este asociată o *durată (sau valoare)*. Valorile notelor nu reprezintă unități absolute de timp, ci sunt definite relativ la nota întreagă. Notele pot fi acompaniate de diferite *accente*, care modifică durata, frecvența(tonalitatea) sau intensitatea.











Notă	Pauză	Valoarea notei	Denumire
		1	Notă întreagă
		$\frac{1}{2}$	Doime
		$\frac{1}{4}$	Pătrime
		$\frac{1}{8}$	Optime
		$\frac{1}{16}$	Şaisprezecime

Tabela 2: Durata notelor muzicale

O secvență de note poate fi sincronizată cu structura metrică, caz în care aceasta devine o unitate ritmică. Există mai multe moduri prin care se poate realiza sincronizarea: [1]

- **Metric:** Valorile notelor sunt identice cu pulsurile unui nivel metric.
- **Intrametric:** Valorile notelor sunt bazate pe grupări de pulsuri din structura metrică dar nu sunt egale cu pulsurile din structura metrică, însă accentele lor corespund accentuării din structura metrică.
- **Contrametric:** Valorile notelor sunt identice cu pulsurile unui nivel metric sau sunt bazate pe grupări de pulsuri din structura metrică dar accentele lor nu corespund accentuării din structura metrică, ci o perturbă.
- **Extrametric:** Valorile notelor sunt bazate pe grupări de pulsuri din afara structurii metrice.

2.1.3 Compoziție

Structura unei secvențe muzicale diferă în funcție de instrumentul pentru care aceasta este compusă. Instrumentele muzicale pot fi *monofonice* (o singură notă este redată în orice moment) sau *polifonice* (mai multe note pot fi redade simultan). În funcție de rolul lor în compoziție, instrumentele pot fi încadrate în mai multe categorii, printre care:

- **Bass:** monofonic, redă note cu frecvențe reduse (până în 260Hz)
- **Lead:** monofonic, este compus din note cu frecvențe mijlocii sau ridicate și are rolul de a reda melodia
- **Pad:** polifonic, este ambiental și are rolul de a stabili armonia

2.2 Metode de măsurare a nivelului de sincopare

Sincopa este contradicția momentană a structurii ritmice predominante i.e. o secvență de note este sincopată atunci când sincronizarea sa cu structura ritmică se realizează în mod contrametric [1]. Pentru a ilustra modul de calculare al metodelor prezentate voi folosi secvența "Bossa-Nova".

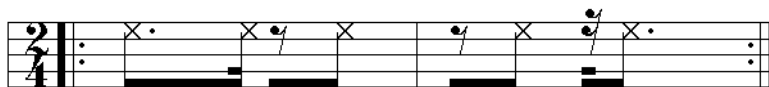


Figura 2.3: Bossa nova

2.2.1 Gómez "Weighted Note-to-Beat"

Modelul "Weighted Note-to-Beat" [8] definește nivelul de sincopare al unei note ca fiind distanța dintre pulsurile metrului. Pentru a calcula distanța $D_{WNBD}(S)$ definim întâi x_n o notă cu x_n^s și x_n^e începutul, respectiv sfârșitul, b_i, b_{i+1} "beat-urile" între care se află, și distanța d .

$$b_i \leq x_n^s \leq b_{i+1} \quad (2.1)$$

$$d(x_n, b_i) = \frac{x_n^s - b_i}{b_{i+1} - b_i} \quad (2.2)$$

În continuare definim $T(x_n)$ ca fiind distanța dintre x_n și cel mai apropiat beat.

$$T(x_n) = \min(d(x_n, b_i), d(x_n, b_{i+1})) \quad (2.3)$$

Valoarea $W(x_n)$ reprezintă nivelul de sincopare al unei note și se calculează astfel:

$$W(x_n) = \begin{cases} 0, & \text{dacă } d(x_n, b_i) = 0 \\ \frac{2}{T(x_n)}, & \text{dacă } b_{i+1} < x_n^e \leq b_{i+2} \\ \frac{1}{T(x_n)}, & \text{altfel} \end{cases} \quad (2.4)$$

În final, nivelul de sincopare al unei structuri ritmice se calculează astfel:

$$D_{WNBD}(S) = \frac{1}{|Y|} \sum_{n=0}^{|Y|-1} W(x_n) \quad (2.5)$$

Spre exemplu, pentru secvența Bossa-Nova ponderile notelor sunt (0, 4, 8, 8, 4), deci WNBD prezice un nivel de sincopare de valoare 3.

2.2.2 Toussaint "Off-Beatness"

Modelul "Off-Beatness" [9] definește o notă ca fiind sincopată atunci când este redată în timpul unui puls "off-beat". Un puls este considerat off-beat dacă începe în același timp cu un puls din mulțimea formată din generatorii grupului ciclic C_n , unde n este numărul de pulsuri al metrului. Pentru a calcula nivelul de sincopare D_{TOB} al unei structuri ritmice definim întâi B ca fiind mulțimea numerelor non-prime față de n .

$$B = \{i, n \bmod i = 0 : 1 < i < n\} \quad (2.6)$$

Valoarea $W(x_n)$ reprezintă nivelul de sincopare al unei note și se calculează astfel:

$$W(x_n) = \begin{cases} 0, & \text{dacă } x \bmod i = 0 \forall i \in B \\ 1, & \text{altfel} \end{cases} \quad (2.7)$$

În final, nivelul de sincopare al unei structuri ritmice se calculează folosind modelul "Off-Beatness" astfel:

$$D_{TOB}(S) = \frac{1}{|Y|} \sum_{n=0}^{|Y|-1} W(x_n) \quad (2.8)$$

Pentru secvența Bossa-Nova conține 16 pulsuri, iar notele sunt distribuite astfel: [x..x..x...x..x..]. Așadar, a 2-a și a 5-a sunt offbeat, deci nivelul de sincopare prezis este 2.

2.3 Formatul MIDI

Formatul MIDI (Musical Instrument Digital Interface) este un standard tehnic care descrie un protocol de comunicații, o interfață digitală și tipuri de conectori electrici

care conectează instrumente muzicale electronice și dispozitive audio. Informația MIDI este transmisă prin *mesaje MIDI*. Acestea sunt formate dintr-un *status byte*, urmat de unul sau mai mulți *data bytes* și pot fi clasificate drept "System Messages, "Channel Voice Messages" și "channel mode messages". [10]

Channel Voice Messages

Channel Voice Messages sunt folosite pentru a transmite informații legate de performanța muzicală și sunt de tipul *Note On*, *Note Off*, *Polyphonic Key Pressure*, *Channel Pressure*, *Pitch Bend Change*, *Program Change*, sau *Control Change messages*. O notă muzicală este transmisă pentru a fi redată folosind un mesaj de tipul *Note On*, indicându-se nota (tonică) și viteza, urmat de un mesaj de tipul *Note Off*. Astfel, durata notei este calculată în funcție de diferența dintre timpii la care s-au primit cele 2 evenimente.

Channel Mode Messages

Channel Mode Messages afectează modul în care sintetizatoarele răspund datelor MIDI. Acestea pot fi folosite pentru a selecta între a reda note monofonic sau polifonic.

System Messages

System Messages diferă de celelalte două tipuri prin faptul că nu sunt specifice unui singur canal, deci nu includ un număr de canal în status byte. Acestea pot fi folosite pentru a sincroniza instrumente, sau pentru a transmite mesaje definite exclusiv pentru anumite echipamente.

2.4 Audio plugin

Un plugin audio este un plugin folosit pentru a adăuga sau îmbunătăți funcționalități audio într-un program, în general un DAW (Digital Audio Workstation). Există mai multe arhitecturi care funcționează diferit în funcție de sistemul de operare sau DAW, cele mai utilizate fiind:

- **VST (Virtual Studio Technology):** cel mai utilizat format, cross-platform
- **AU (AudioUnits):** creat de Apple special pentru MacOS

JUCE

JUCE este un framework de C++ cross-platform folosit pentru a crea audio plugins. Acesta implementează multiple funcționalități audio și oferă posibilitatea exportării proiectelor în majoritatea formatelor.

3 Arhitectura aplicației

Aplicația conține un audio plugin exportat în format VST3, AU și Standalone scrisă în C++ folosind JUCE, un modul scris în python pentru generarea secvențelor muzicale, precum și o librărie comună scrisă în C++ folosind Pybind11.

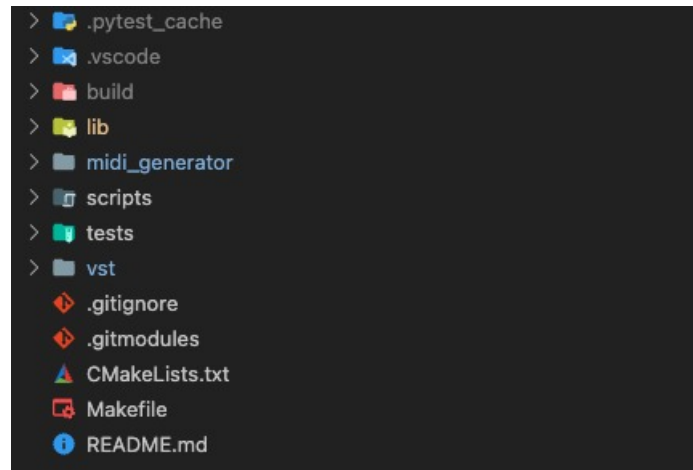


Figura 3.1: Structura proiectului după build

3.1 Structură

Aplicația e construită folosind cmake și make, și a fost testată pe Debian 11 ARM și macOS 12. Pentru a putea utiliza aplicația, următoarele dependențe trebuie instalate:

- Python 3.10 (versiunea dev)
- CMake 3.23
- Make 4.3
- Clang 11

3.2 Librării

Boost

Boost este o colecție de librării scrise în C++ care implementează funcționalități folosite frecvent în dezvoltarea software. Dintre acestea, am folosit librăriile de logging și cel de testare unitară.

Pybind11

Pybind11 este o bibliotecă care permite interoperabilitate între C++ și python. Aceasta este folosită în aplicație pentru a facilita crearea unei interfețe de comunicare între plugin și modulul genetic. Un exemplu conținând implementarea endpoint-ului pentru apelarea funcției de generare din cadrul algoritmului genetic se poate găsi în Anexa [B.3](#).

3.3 Build tools

Make

Pentru construirea, testarea și depanarea aplicației am folosit Make și CMake. În Make sunt definite 7 sarcini executabile:

- **config** - configurează structura proiectelor CMake.
- **build-lib** - construiește bibliotecă comună folosită de plugin și modul.
- **build** - construiește modulul de python și pluginul standalone.
- **test** - execută testele unitare conținute în aplicație.
- **run** - rulează pluginul standalone.

- **clean** - șterge folder-ul în care este construită aplicația (build) și fișierele temporare create de python.
- **all** - execută în ordinea config→build-plugin→build→test→run.

Cmake

CMake este unealta de împachetare principală folosită. Aplicația folosește 4 fișiere de configurare pentru crearea executabilului final. Primul dintre acestea este conținut în directorul principal al aplicației și reprezintă punctul de început al configurației (Anexa B.4). Aici sunt căutate librăriile din python și boost și executabilul de python și sunt incluse în configurație, după care este configurată librăria comună și este construit și instalat modulul genetic. Apoi, sunt adăugate și configurate modulele folosite în cadrul plugin-ului, urmate de plugin-ul audio și configurarea testelor.

Următorul fișier de configurare este localizat în directorul în care este conținută librăria comună și are scopul de a o construi și adăuga în configurație. Directorul în care se crează plugin-ul audio conține de asemenea un fișier de configurare, care crează executabilele în format .VST, .AU și Standalone, și include modulele folosite. Ultimul fișier este localizat în directorul care conține testele unitare și este folosit pentru a crea mai multe executabile, reprezentând suite de teste.

3.4 Testare

Aplicația conține teste unitare scrise folosind pytest (pentru testarea modulului) și librăria de teste unitare conținute în boost (pentru librărie comună și plugin-ul audio). Acestea sunt folosite în special pentru a verifica dacă este configurată corect comunicarea între componentele distincte ale aplicației.

Un exemplu de suită de teste este inclusă în anexa [B.5](#), folosit pentru a verifica dacă librăria poate traduce informație legată de notele muzicale din Python în C++ și invers. Celelalte suite de teste verifică dacă librăria dinamică este încărcată corect la runtime, funcționarea corespunzătoare a API-ului și funcționarea comenzilor conținute în modulul genetic.

4 Implementarea plugin-ului audio

Plugin-ul audio este implementat în C++ folosind framework-ul JUCE, împreună cu modulul PluginGuiMagic [18]. Clasa MagicProcessor din modulul PluginGuiMagic extinde clasa AudioProcessor și reprezintă clasa de bază din care este derivat punctul de început al plugin-ului.

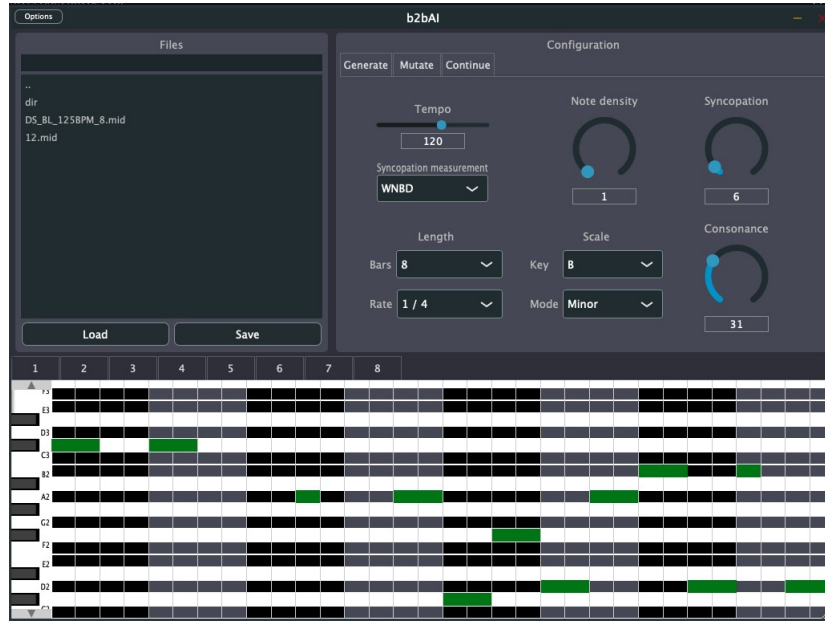


Figura 4.1: Interfața grafică a plugin-ului

4.1 B2bAIPluginProcessor

Interfața grafică a plugin-ului este creată dintr-un fișier XML, care conține o structură arborescentă de componente (Anexa B.1). Acesta este încărcat în constructor-ul clasei B2bAIPluginProcessor (Anexa A.1), care extinde clasa MagicProcessor și reprezintă componenta principală a plugin-ului. Clasa crează de asemenea parametrii audio și celelalte resurse dinamice și le atașează componentelor corespunzătoare. În plus,

aceasta recepționează și procesează mesaje MIDI și evenimente audio.

Metoda `initialiseBuilder`, care suprascrie metoda virtuală a clasei `MagicProcessor`, permite înregistrarea de componente noi refolosibile.

```
void B2bAIAudioProcessor::initialiseBuilder
                                   (foleys::MagicGUIBuilder& builder)
{
    builder.registerJUCEFactories();

    builder.registerFactory ("PianoRoll", &PianoRoll::factory);
    builder.registerFactory ("SearchBar", &SearchBar::factory);
}
```

4.2 Piano roll

Piano roll-ul permite vizualizarea și editarea notelor dintr-o secvență. Opt secvențe pot fi editate simultan; pentru a încarcă o secvență în piano roll se folosesc tab-urile numerotate de la 1 la 8. Notele pot fi adaugate, șterse, mutate sau redimensionate pe grid folosind mouse-ul, valorile notelor (tonice) pot fi derulate folosind mousewheel-ul.

Piano roll-ul este implementat folosind clasa `PianoRollComponent` (Anexa [A.2](#)), care încapsulează 2 componente: un pian (`KeyboardComponent`) și un grid (`GridComponent`). Pianul este un reskin al componentei `JUCE::MidiKeyboardComponent`. Grid-ul este o matrice construită dinamic în funcție de evenimentele generate de mouse, numărul de pulsuri al metrului și o listă de note inițializată la runtime (Anexa [B.2](#)).

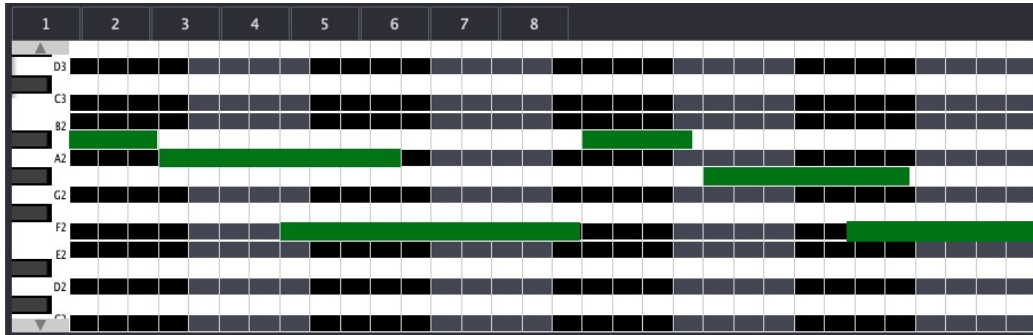


Figura 4.2: Piano roll

Wrapper-ul PianoRoll (Anexa A.7) încapsulează această clasă PianoRollComponent și este folosit pentru a o înregistra drept componentă reutilizabilă în B2bAIAudioProcessor::initialiseBuilder. În plus, wrapperul prezintă și o interfață pentru a inițializa și actualiza referința către lista de note din grid.



Figura 4.3: Selectarea secvenței din piano roll

4.3 Sistemul de fișiere

Sistemul fișier de interne poate fi navigat din aplicației. Secvența MIDI asociată piano roll-ului poate fi salvată ca fișier intern. De asemenea un fișier în format MIDI poate fi descărcat și editat în piano roll.

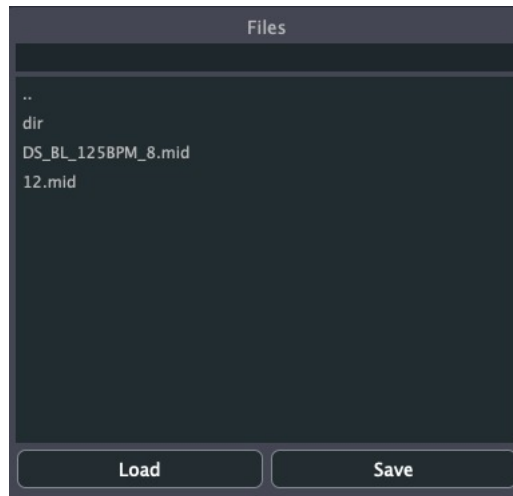


Figura 4.4: File tree

Implementarea este realizată folosind clasa `MidiFileListBox` (Anexa A.3), care extinde clasele `ListBoxModel`, `ChangeBroadcaster`, `ChangeListener` și `Label::Listener`. Astfel, aceasta definește 3 funcții care sunt implementate în procesorul audio, care modifică setările comune ale aplicației, și actualizează starea componentelor interioare de fiecare dată când sunt apelate. În plus, acest lucru se întâmplă și pentru chenarul de deasupra, actualizarea având loc în momentul în care este schimbată valoarea din chenar. Chenarul este definit în clasa `SearchBar` (Anexa A.4) și este înregistrat drept componentă reutilizabilă.

4.4 Parametrii audio

Parametrii audio ai aplicației sunt folosiți pentru a configura algoritmul genetic. Fiecare dintre cele 4 funcționalități ale aplicației are parametrii de configurație proprii.

Generate

- Syncopation - nivelul de sincopare

- Note density - densitatea notelor
- Consonance - proporția notelor consonante
- Key - gama în care este generată secvența
- Mode - modul gamei
- Tempo - tempo-ul secvenței; este folosit în scrierea fișierelor midi
- Bars - numărul de măsuri al secvenței
- Rate - durata unui puls
- Syncopation measurement - algoritmul folosit pentru măsurarea sincopării

Mutate

- Pitch - rata de mutație a frecvenței
- Velocity - rata de mutație a vitezei
- Duration - rata de mutație a lungimilor notelor
- Consonance - proporția notelor consonante

Continue

- Compression method: algoritmul folosit pentru compresia secvențelor

Combine

- Compression method: algoritmul folosit pentru compresia secvențelor
- Sequences: secvențele folosite în combinaire

5 Modul midi-generator

Modulul *midi-generator* este un modul scris în Python care folosește librăria *deap* [15] pentru a implementa diverse funcționalități folosind algoritmi genetici. Acesta conține atât o interfață de linie de comandă pentru a putea fi folosit de sine stătător, cât și un API care prezintă funcționalitățile implementate. Modulul este construit, testat și distribuit automat prin github actions la fiecare acțiune de push sau merge pe branch-ul principal de GitHub.

5.1 Algoritmul genetic

Algoritmul genetic este implementat folosind *deap*, un framework care oferă o colecție de unelte compusă din modulul *creator* și modulul *toolbox*. Primul modul permite creerea de clase noi care pot fi folosite în cadrul genotipului, iar al doilea reprezintă un container pentru operatorii genetici.

5.1.1 Codificare

Pentru a rezolva problema codificării unei secvențe de note muzicale am definit întâi clasa *Gene*:

```
@dataclass
class Gene:
    pitch: int
    velocity: int
    remaining_ticks: int
```

Astfel, genele sunt reprezentate de pulsurile metrului folosit și conțin frecvența, viteza și numărul de pulsuri rămase din nota redată la momentul respectiv. Pentru a putea înregistra genotipul în *toolbox* am creat un generator (Anexa B.7) de note

continue în timp, de lungime variată. Acesta este implementat folosind un closure care reține valoarea notei anterioare. Funcția de generare din cadrul closure-ului verifică dacă pulsul curent are loc în timpul redării notei anterioare, caz în care returnează nota precedentă, din care este scăzut un tick. În caz contrar, generatorul returnează o notă aleatoare. Așadar, un individ reprezintă o listă formată din rezultatele returnate de generator. Înregistrarea notei în toolbox se realizează în felul următor:

```
creator.create("Individual", list, fitness=creator.Fitness)
toolbox.register("individual", tools.initRepeat,
    -> creator.Individual, toolbox.generator, n=NO_TICKS)
toolbox.register("population", tools.initRepeat, list,
    -> toolbox.generator)
```

5.1.2 Mutație

Mutația (Anexa B.6) poate afecta frecvența, viteza sau durata unei note. În cazul în care mutația modifică durata unei note, aceasta poate afecta mai multe gene simultan (valabil și în cadrul crossover-ului). Pentru a rezolva această problemă am definit un decorator (Anexa B.8) care parcurge pulsurile cromozomului rezultat în urma mutației (sau a crossover-ului) și setează valoarea numărului de pulsuri rămase din notă ca fiind egală cu numărul de gene imediat consecutive care au valoare frecvenței egală cu cea a pulsului curent. Modul prin care sunt decorate cele 2 funcții este următorul:

```
decorator = check_remaining_ticks()
toolbox.decorate("mate", decorator)
toolbox.decorate("mutate", decorator)
```

5.1.3 Fitness

Funcția de fitness este calculată în funcție de nivelul de sincopare, densitatea notelor și rația notelor consonante. Pentru calcularea nivelului de sincopare, se poate alege între

măsura WNBD (Capitolul 2.2.1, implementare Anexa B.10) și off-beatness (Capitolul 2.2.2, implementare Anexa B.9). Densitatea și rația notelor consonante sunt calculate prin raportul dintre numărul de pulsuri în care sunt redată note, respectiv numărul de pulsuri în care sunt redată note din gama selectată și numărul total de pulsuri.

Pentru funcțiile continue și combine funcția de fitness este calculată folosind distanța NCD (Normalized Compression Distance) care se calculează prin formula:

$$NCD(x, y) = \frac{\max(C(xy) - C(x), C(yx) - C(y))}{\max(C(x), C(y))} \quad (5.1)$$

Unde $C(x)$ reprezintă lungimea rezultatului compresiei lui x folosind orice algoritm aproximativ pentru complexitatea Kolmogorov [12]. Astfel, pentru un set de secvențe S și un individ x funcția de fitness este dată de formula:

$$f(x) = \frac{1}{\sum_{s \in S} NCD(x, s)} \quad (5.2)$$

În aplicație sunt implementați 3 algoritmi de compresie, anume LZ77 [2], LZ78 [3] și LZW [4]. Implementarea algoritmului LZ77 se poate găsi în Anexa B.11.

5.2 API

Modulul expune funcționalitățile implementate sub forma unui API format din următoarele endpoints:

- **mutate** - primește ca argument o configurație; returnează o secvență de note generate folosind algoritmul genetic, în funcție de configurația primită
- **continue** - primește ca argumente o secvență de note și o configurație; retur-

nează rezultatul aplicării operatorului de mutație pe secvența de note, în funcție de configurația primită

- **continue** - primește ca argumente o secvență de note și o configurație; returnează o secvență de note generate folosind algoritmul genetic cu funcția de fitness kolmogorov, în funcție de configurația și secvențele primite
- **combine** - primește ca argumente o listă de secvențe de note și o configurație; returnează o secvență de note generate folosind algoritmul genetic cu funcția de fitness kolmogorov, în funcție de configurația și secvențele primite
- **write_file** - primește ca argumente o secvență de note și calea unui fișier; scrie în calea primită un fișier MIDI format din secvența de note transmisă ca argument

6 Concluzii

În această lucrare am implementat o colecție de unelte care poate asista muzicienii în procesul de creație, incluzând unelte pentru generarea, mutația, continuarea și combinarea de secvențe muzicale. Acestea sunt configurabile, păstrând astfel expresia creativă a utilizatorului.

6.1 Limitări ale aplicației

Deși aplicația reprezintă un plugin audio, aceasta nu interacționează deloc cu DAW-ul din care este deschisă; pentru a putea folosi aplicația în cadrul unui DAW este nevoie ca secvențele generate să fie salvate în format MIDI, după care să fie deschise în DAW.

În cadrul evaluării fitness-ului unei secvențe viteza notelor nu este deloc utilizată, astfel că o dimensiune muzicală care ar putea fi fost folosită în cadrul generării nu este deloc valorificată. În plus, momentan aplicația nu poate genera secvențe polifonice, fiind limitată astfel mulțimea de tipuri de instrumente pentru care pot fi generate secvențe.

6.2 Posibile dezvoltări ulterioare

Plugin-ul ar putea interacționa cu DAW-ul în care este deschis folosind mesaje trimise pe MIDI Channels. Acestea ar putea fi folosite pentru a reda sau a înregistra secvența într-un track. De asemenea, designul interfeței grafice ar putea fi îmbunătățit.

Viteza ar putea fi folosită pentru a accentua sau diminua momentele de tensiune create de notele consonante sau pentru a schimba nivelul de sincopare al unei

secvențe. Secvențele polifonice ar putea fi generate folosind acorduri și progresii de acorduri. Pentru a identifica și valorifica expresiile muzicale dintr-o secvență, generarea ar putea fi implementată folosind optimizare swarm în loc de algoritmi genetici, utilitățile necesare fiind prezente în modulul deap.

Referințe

- [1] R. DeLone et al., *Aspects of Twentieth-century Music*, Prentice-Hall, 1975, ISBN: 9780130493460, URL: <https://books.google.ro/books?id=ZGQXAQAAIAAJ>.
- [2] J. Ziv și A. Lempel, “A universal algorithm for sequential data compression”, în *IEEE Transactions on Information Theory* 23.3 (1977), pp. 337–343, DOI: [10.1109/TIT.1977.1055714](https://doi.org/10.1109/TIT.1977.1055714).
- [3] J. Ziv și A. Lempel, “Compression of individual sequences via variable-rate coding”, în *IEEE Transactions on Information Theory* 24.5 (1978), pp. 530–536, DOI: [10.1109/TIT.1978.1055934](https://doi.org/10.1109/TIT.1978.1055934).
- [4] Welch, “A Technique for High-Performance Data Compression”, în *Computer* 17.6 (1984), pp. 8–19, DOI: [10.1109/MC.1984.1659158](https://doi.org/10.1109/MC.1984.1659158).
- [5] F. Lerdahl și R.S. Jackendoff, *A Generative Theory of Tonal Music, reissue, with a new preface*, The MIT Press, MIT Press, 1996, ISBN: 9780262260916, URL: <https://books.google.ro/books?id=6HGiEW33lucC>.
- [6] Christopher Ariza, “Prokaryotic Groove: Rhythmic Cycles as Real-Value Encoded Genetic Algorithms”, în (Ian. 2002).
- [7] Judy Franklin, “Multi-phase learning for jazz improvisation and interaction”, în (Apr. 2002).
- [8] Francisco Gómez et al., “Mathematical measures of syncopation”, în *Proceedings of the BRIDGES: Mathematical Connections in Art, Music and Science* (Ian. 2005).

- [9] Godfried Toussaint, “Mathematical Features for Recognizing Preference in Sub-Saharan African Traditional Rhythm Timelines”, în vol. 3686, Aug. 2005, pp. 18–27, ISBN: 978-3-540-28757-5, DOI: [10.1007/11551188_2](https://doi.org/10.1007/11551188_2).
- [10] MIDI Manufacturers Association, *The Complete MIDI 1.0 Detailed Specification: Incorporating All Recommended Practices ; Document Version 96.1*, MIDI Manufacturers Association Incorporated, 2006, ISBN: 9780972883108, URL: <https://www.midi.org/specifications-old/item/the-midi-1-0-specification>.
- [11] Manuel Alfonseca, Manuel Cebrián și Alfonso De la Puente, “A simple genetic algorithm for music generation by means of algorithmic information theory”, în Sept. 2007, pp. 3035–3042, DOI: [10.1109/CEC.2007.4424858](https://doi.org/10.1109/CEC.2007.4424858).
- [12] Manuel Alfonseca, Manuel Cebrián și Alfonso De la Puente, “A simple genetic algorithm for music generation by means of algorithmic information theory”, în (Sept. 2007), pp. 3035–3042, DOI: [10.1109/CEC.2007.4424858](https://doi.org/10.1109/CEC.2007.4424858).
- [13] J.P. Burkholder, D.J. Grout și C.V. Palisca, *A History of Western Music*, W. W. Norton, 2010, ISBN: 9780393931259, URL: <https://books.google.ro/books?id=INQIAQAAMAAJ>.
- [14] Andrew Hawryshkewich, Philippe Pasquier și Arne Eigenfeldt, “Beatback : A Real-time Interactive Percussion System for Rhythmic Practise and Exploration”, în *Proceedings of the International Conference on New Interfaces for Musical Expression* (Sydney, Australia), Zenodo, Iun. 2010, pp. 100–105, DOI: [10.5281/zenodo.1177797](https://doi.org/10.5281/zenodo.1177797), URL: <https://doi.org/10.5281/zenodo.1177797>.

- [15] Félix-Antoine Fortin et al., “DEAP: Evolutionary algorithms made easy”, în *Journal of Machine Learning Research, Machine Learning Open Source Software* 13 (Iul. 2012), pp. 2171–2175.
- [16] Chunyang Song, “Syncopation: Unifying Music Theory and Perception”, în 2014.
- [17] Adam Roberts et al., “Magenta Studio: Augmenting Creativity with Deep Learning in Ableton Live”, în *Proceedings of the International Workshop on Musical Metacreation (MUME)*, 2019, URL: http://musicalmetacreation.org/buddydrive/file/mume_2019_paper_2/.
- [18] Foleys Finest Audio, *Plugin Gui Magic*, <https://foleysfinest.com/PluginGuiMagic>, Accesat la data de 20-08-2022.
- [19] John A. Marauver, *"A Brief History of Algorithmic Composition"*, ["https://ccrma.stanford.edu/~blackrse/algorithm.html"](https://ccrma.stanford.edu/~blackrse/algorithm.html), Accesat la data de 25-08-2022.
- [20] Reason Studio, *"Bassline generator"*, <https://www.reasonstudios.com/shop/rack-extension/bassline-generator>, Accesat la data de 28-08-2022.

A Header files

```
PluginProcessor.h
1
2 {
3 public:
4     B2bAIAudioProcessor();
5     ~B2bAIAudioProcessor() override;
6
7     void prepareToPlay (double sampleRate, int samplesPerBlock)
8         ↪ override;
9     void releaseResources() override;
10
11     #ifndef JucePlugin_PreferredChannelConfigurations
12         bool isBusesLayoutSupported (const
13             AudioProcessor::BusesLayout& layouts)
14             const override;
15     #endif
16     void processBlock (AudioBuffer<float>&, MidiBuffer&) override;
17     void saveMidiFile();
18     void loadMidiFile(const File& file);
19     void loadDirectory(const File& file);
20     void updateListBox(const String& text);
21     double getTailLengthSeconds() const override;
22
23 private:
24     MidiFileListBox *midiFileListBox;
25     MidiSequence *midiSequence;
26     File midiFilesDir;
27     AudioProcessorValueTreeState treeState;
28     JUCE_DECLARE_NON_COPYABLE_WITH_LEAK_DETECTOR (B2bAIAudioProcessor)
29     File getFile(int index);
30     void initialiseBuilder(foleys::MagicGUIBuilder &builder) override;
31 };
```

```
PianoRollComponent.h
1
2 class PianoRollComponent: public Component {
3 private:
4     std::unique_ptr<KeyboardComponent> keyboardComponent;
5     std::unique_ptr<GridComponent> gridComponent;
6     void timerCallback() override;
7 public:
8     enum ColourIds {
9         noteColour = 0x00FF00
```

```

10     };
11
12     PianoRollComponent(MidiKeyboardState& state,
13         KeyboardComponent::Orientation orientation);
14     void paint (juce::Graphics&) override;
15     void resized() override;
16     void mouseWheelMove(const MouseEvent &event,
17         const MouseWheelDetails &wheel) override;
18     void setMidiSequence(MidiSequence *sequence);
19 };

```

MidiFileListBox.h

```

1
2 class MidiFileListBox: public ListBoxModel,
3                       public Label::Listener,
4                       public ChangeBroadcaster,
5                       public ChangeListener {
6 private:
7     File midiFilesDir;
8     Array<File> midiFiles;
9     foleys::SharedApplicationSettings settings;
10    String searchText;
11
12    JUCE_DECLARE_NON_COPYABLE_WITH_LEAK_DETECTOR (MidiFileListBox)
13 public:
14     MidiFileListBox();
15     ~MidiFileListBox() override;
16
17     void listBoxItemClicked (int rowNumber, const juce::MouseEvent&
18         ↪ event) override;
19     void listBoxItemDoubleClicked(int row, const juce::MouseEvent &)
20         ↪ override;
21     void paintListBoxItem (int rowNumber, juce::Graphics &g, int width,
22         ↪ int height, bool rowIsSelected) override;
23
24     void labelTextChanged(juce::Label *labelThatHasChanged) override;
25     int getNumRows() override;
26     void changeListenerCallback (juce::ChangeBroadcaster*) override;
27     std::function<void(File file)> onSelectionChanged;
28     std::function<void(File file)> onDoubleClick;
29     std::function<void(String text)> update;
30 };

```

SearchBar.h

```

1
2 class SearchBar : public foleys::GuiItem
3 {
4 private:
5     juce::Label label;
6     Label::Listener *listener = nullptr;
7
8     JUCE_DECLARE_NON_COPYABLE_WITH_LEAK_DETECTOR (SearchBar)
9 public:
10    FOLEYS_DECLARE_GUI_FACTORY (SearchBar)
11
12    static const juce::Identifier pText;
13    static const juce::Identifier pJustification;
14    static const juce::Identifier pFontSize;
15    static const juce::Identifier pDestination;
16
17    SearchBar (foleys::MagicGUIBuilder& builder, const juce::ValueTree&
18        ↪ node);
19    void update() override;
20    std::vector<foleys::SettableProperty> getSettableProperties() const
21        ↪ override;
22    juce::Component* getWrappedComponent() override;
23 };

```

NoteRectangle.h

```

1
2 class NoteRectangle: public Rectangle<int> {
3 public:
4     NoteRectangle(int x=0, int y=0, int width=0, int height=0, int
5         ↪ p=0);
6     NoteRectangle(int p, int v, double s, double e);
7     [[nodiscard]] int getPitch() const;
8     void setPitch(int pitch);
9     friend std::ostream &operator<<(std::ostream &os, const
10        ↪ NoteRectangle &note);
11     [[nodiscard]] int getVelocity() const;
12     void setVelocity(int velocity);
13     bool operator==(const NoteRectangle &rhs) const;
14     bool operator!=(const NoteRectangle &rhs) const;
15     [[nodiscard]] double getStart() const;
16     void setStart(double start);
17     [[nodiscard]] double getEnd() const;
18     void setEnd(double end);
19 private:
20     Note note;
21 };

```

```

1                                     GridComponent.h
2
3 class GridComponent: public Component {
4 private:
5     OwnedArray<Range<int>> noteLineRanges;
6     OwnedArray<Range<int>> noteRowRanges;
7     MidiSequence *notes = nullptr;
8     NoteRectangle pressed, new_position;
9     NoteRectangle find_note_rect(Point<int> position);
10    int normalise(double w, double wMax);
11 public:
12     GridComponent();
13
14     void updateNoteLineRanges(int firstKeyStartPosition);
15
16     void paint(Graphics &g) override;
17     void mouseMove(const MouseEvent &event) override;
18     void mouseDown(const MouseEvent &event) override;
19     void mouseDrag(const MouseEvent &event) override;
20     void mouseDoubleClick(const MouseEvent &event) override;
21     void mouseUp(const MouseEvent &event) override;
22     void setMidiSequence(MidiSequence *sequence);
23 };

```

```

1                                     PianoRoll.h
2
3 class PianoRoll: public foleys::GuiItem {
4 private:
5     PianoRollComponent pianoRoll;
6
7     JUCE_DECLARE_NON_COPYABLE_WITH_LEAK_DETECTOR(PianoRoll)
8 public:
9     FOLEYS_DECLARE_GUI_FACTORY(PianoRoll)
10
11     static const juce::Identifier pSource;
12     static const juce::Identifier pNoteColor;
13
14     PianoRoll(foleys::MagicGUIBuilder& builder, const juce::ValueTree&
15         ↪ node);
16
17     void update() override;
18
19     [[nodiscard]] std::vector<foleys::SettableProperty>
20         ↪ getSettableProperties() const override;

```



```
18  
19     juce::Component* getWrappedComponent() override;  
20 };  


---


```

B Referințe cod

Arbore XML

```
1
2 <View background-color="FF4E505F" height="10%" display="flexbox"
  ↳ margin="0"
3     padding="0" flex-align-self="stretch" flex-direction="column"
4     radius="0" border="0" caption="Search" caption-size="0">
5   <SearchBar font-size="16.0" justification="top-left"
6     ↳ label-background="212c31"
7       text="" max-height="25" height=""
8       ↳ flex-align-self="stretch" margin="0"
9       padding="1" border="0" label-outline="FF4E505F"
10      ↳ background-color="FF4E505F"
11      radius="0" caption="SearchBox" caption-size="0" pos-x="0%"
12      ↳ pos-y="0%"
13      pos-width="100%" pos-height="8.25083%"
14      ↳ destination="filetree"/>
15   <ListBox list-box-model="filetree" pos-x="0"
16     ↳ background-color="FF4E505F"
17     pos-y="0%" pos-width="100%" pos-height="10.7527%"
18     ↳ padding="2"
19     caption="FileTree" caption-size="0"/>
20 </View>
```

Paint grid

```
1
2 void GridComponent::paint(Graphics& g)
3 {
4     int height = getHeight();
5     int width = getWidth();
6     // Draw the background
7     for (int i = 0; i < 8; i += 2) {
8         auto shadow = DropShadow(Colours::black, 1, Point<int>(0, 0));
9         auto rect = Rectangle<int>(noteRowRanges[i * 4]->getStart(),
10                                     0,
11                                     noteRowRanges[(i + 1) *
12                                                         ↳ 4]->getStart() - noteRowRanges[i
13                                                         ↳ * 4]->getStart(),
14                                     height);
15         shadow.drawForRectangle(g, rect);
16     }
17     g.setColour(Colours::white);
18     // Draw note lines
```

```

18     for (int i = 0; i < noteLineRanges.size(); i++) {
19         auto rect = Rectangle<int>(0, noteLineRanges[i]->getStart(),
20             ↪ width, noteLineRanges[i]->getLength());
21         if (rect.getBottom() < 0 || rect.getY() >= height)
22             continue;
23         g.drawRect(rect, 1);
24         if (!MidiMessage::isMidiNoteBlack(i))
25             continue;
26         g.fillRect(rect);
27     }
28
29     g.setColour(Colours::lightgrey);
30     // Draw beats lines
31     for (auto noteRowRange : noteRowRanges) {
32         g.fillRect(static_cast<int>(noteRowRange->getStart() - 1), 0,
33             ↪ 1, height);
34     }
35
36     // draw notes
37     g.setColour(Colours::green);
38     if (notes)
39         for (const auto& note : *notes) {
40             if (pressed == note) {
41                 g.fillRect(new_position.expanded(1));
42                 continue;
43             }
44             g.fillRect(note);
45         }
46     }
47 }

```

```

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

```

```

Pybind11
std::list<mg::Note> mg::generate() {
    try {
        pybind11::module_ commands =
            ↪ pybind11::module_::import("midi_generator.commands");
        py::object result = commands.attr("generate")();
        std::list<mg::Note> notes;
        for (const auto& obj: result)
            notes.emplace_back(
                obj.attr("pitch").cast<int>(),
                obj.attr("velocity").cast<int>(),
                obj.attr("start").cast<double>(),
                obj.attr("end").cast<double>()
            );
    }
}

```

```

14         return notes;
15     } catch (py::error_already_set &error) {
16         error.discard_as_unraisable(__func__ );
17         return {};
18     }
19 }
20

```

CMake

```

1
2 cmake_minimum_required(VERSION 3.23)
3 project(b2bAI VERSION 1.0)
4
5 # Find python
6 set(Python3_ADDITIONAL_VERSIONS 3.10.6)
7 find_package (Python3 COMPONENTS Interpreter Development)
8 # Find boost
9 find_package(Boost 1.79.0 REQUIRED COMPONENTS log python310
   ↳ unit_test_framework filesystem)
10 add_definitions(${Boost_DEFINITIONS})
11 # Build shared library
12 option(BUILD_SHARED_LIBS "Build using shared libraries" ON)
13 add_subdirectory(lib/midi_generator)
14
15 # Build python package
16 if(NOT IS_DIRECTORY ${CMAKE_BINARY_DIR}/midi_generator)
17     file(COPY midi_generator DESTINATION ${CMAKE_BINARY_DIR} PATTERN
   ↳ ".github" EXCLUDE)
18     file(COPY scripts/build.sh DESTINATION ${CMAKE_BINARY_DIR}/scripts)
19     file(COPY scripts/create_note.py DESTINATION
   ↳ ${CMAKE_BINARY_DIR}/scripts)
20     find_program(BASH bash)
21     exec_program(${BASH} ${CMAKE_BINARY_DIR}/scripts ARGS "build.sh"
   ↳ RETURN_VALUE PACKAGE_NOT_BUILT)
22     if(PACKAGE_NOT_BUILT)
23         message(FATAL_ERROR "Couldn't build package")
24     endif()
25 endif()
26
27 #JUCE
28 add_subdirectory(lib/JUCE EXCLUDE_FROM_ALL)
29 juce_add_module(lib/foleys_gui_magic)
30 set_property(GLOBAL PROPERTY JUCE_COPY_PLUGIN_AFTER_BUILD TRUE)
31 option(JUCE_ENABLE_MODULE_SOURCE_GROUPS "Enable Module Source Groups"
   ↳ ON)
32 set_property(GLOBAL PROPERTY USE_FOLDERS YES)

```

```

33 # Company settings
34 set_directory_properties(PROPERTIES JUCE_COMPANY_COPYRIGHT "GNU GENERAL
    ↳ PUBLIC LICENSE Version 3")
35 set_directory_properties(PROPERTIES JUCE_COMPANY_NAME "brahman")
36 set_directory_properties(PROPERTIES JUCE_COMPANY_WEBSITE
    ↳ "https://github.com/speedyleath/b2bAI-VST")
37 set_directory_properties(PROPERTIES JUCE_COMPANY_EMAIL
    ↳ "gheorgheandrei13@gmail.com")
38
39 # Plugin
40 add_subdirectory(vst)
41 # Tests
42 enable_testing()
43 add_subdirectory(tests)

```

```

1
2 void test_extract_note() {
3     pybind11::scoped_interpreter guard{};
4     auto locals = py::dict();
5     py::exec(R"(
6         from note import Note
7         test_note = Note(60, 100, 0, 2)
8     )", py::globals(), locals);
9     try {
10         auto src = locals["test_note"];
11         midi_generator::Note note;
12         note.pitch = src.attr("pitch").cast<int>();
13         note.velocity = src.attr("velocity").cast<int>();
14         note.start = src.attr("start").cast<double>();
15         note.end = src.attr("end").cast<double>();
16         BOOST_TEST(note.pitch == 60);
17         BOOST_TEST(note.velocity == 100);
18         BOOST_TEST(note.start == 0.0f);
19         BOOST_TEST(note.end == 2.0f);
20     } catch (py::error_already_set &error) {
21         BOOST_TEST(false);
22     }
23 }
24
25 void text_embed_note() {
26     pybind11::scoped_interpreter guard{};
27
28     py::module_ module = py::module_::import("note");
29
30     auto constructor = module.attr("Note")(60, 100, 0.0f, 2.0f);

```

```

31     auto locals = py::dict();
32
33     py::exec(R"(
34         from note import Note
35
36         def check_note(note: Note):
37             if note.pitch == 60 and note.velocity == 100:
38                 return True
39             return False
40     )", py::globals(), locals);
41     auto check = locals["check_note"];
42     auto src = check(constructor);
43     BOOST_TEST(src.cast<bool>());
44 }
45
46 test_suite* init_unit_test_suite( int /*argc*/, char* /*argv*/[] )
47 {
48     framework::master_test_suite().
49         add(BOOST_TEST_CASE_NAME(&test_extract_note, "extract note"));
50     framework::master_test_suite().
51         add(BOOST_TEST_CASE_NAME(&text_embed_note, "embed note"));
52     return nullptr;
53 }

```

Mutație

```

1
2 def mutation(config: Configuration, genes):
3     for gene in genes:
4         change = random.random()
5         if change < config.pitch_change_rate:
6             change_2 = random.random()
7             if change_2 > config.consonance_rate:
8                 gene.pitch = random.choice(config.scale.notes[30:40])
9             else:
10                gene.pitch = random.choice(list(set(POSSIBLE_NOTES) -
11                ↪ set(config.scale.notes))[30:40])
12
13        change = random.random()
14        if change < config.length_change_rate:
15            gene.remaining_ticks = 0
16    return genes,

```

Generator

```
1
2 def generator(config: Configuration=Configuration()):
3     prev_gene: Gene | None = None
4
5     def create_random_gene() -> Gene:
6         nonlocal prev_gene
7         if prev_gene is not None and prev_gene.remaining_ticks > 1:
8             prev_gene = Gene(prev_gene.pitch, prev_gene.velocity,
9                               ↪ prev_gene.remaining_ticks - 1)
10            return prev_gene
11
12        tick = min(config.rate)
13        duration = int(random.choice(config.rate) / tick)
14        key = random.choice(list(config.scale.notes)[30:40])
15        velocity = random.randint(80, 100)
16        prev_gene = Gene(key, velocity, duration)
17        return prev_gene
18
19    return create_random_gene
```

Wrapper

```
1
2 def check_remaining_ticks():
3     def decorator(func):
4         def wrapper(*args, **kwargs):
5             offspring = func(*args, **kwargs)
6             for genes in offspring:
7                 for i, gene in enumerate(genes):
8                     if gene.remaining_ticks == 0:
9                         continue
10                    gene.remaining_ticks = len(list(
11                        takewhile(lambda x: x.pitch == gene.pitch,
12                                ↪ genes[i:]))))
13            return offspring
14        return wrapper
15    return decorator
```

Off-Beatness

```
1
2 def off_beatness(notes: list[Note]) -> float:
3     b = [x for x in filter(lambda i: len(notes) % i == 0, range(1,
4                               ↪ len(notes) + 1))]
5     weights = [0 if len(list(filter(lambda i: x % i == 0, b))) else 1
6               ↪ for x in notes]
7     return sum(weights) / len(notes)
```

WNBD

```

1
2 def weighted_note_to_beat(notes: list[Note]) -> float:
3     for note in notes:
4         left = floor(note.start)
5         right = left + 1
6         distance = min(note.start - left, abs(note.start - right))
7
8         if distance == 0:
9             continue
10        if right < note.end < right + 1:
11            total += 2 / distance
12        else:
13            total += 1 / distance
14
15    return total / len(notes)

```

LZ77

```

1
2 def encode_lz77(string: list, window_size=100):
3     encoded = string[: window_size + 1]
4     i = window_size
5     while i < len(string) - window_size:
6         input_buffer = string[i: i + window_size + 1]
7         window = string[i - window_size: i + window_size + 1]
8
9         substring = max([reduce(lambda x, y: x + y,
10                                map(lambda x: x[0],
11                                   takewhile(lambda x: x[0] == x[1],
12                                             zip_longest(string[i + j:
13                                                         ↪ i + window_size],
14                                                         ↪ input_buffer)
15                                             )
16                                , ''') for j in range(-window_size, 0)]
17                                , key=len)
18
19         if substring == '':
20             i += 1
21             encoded += f'0,0${string[i]}'
22         else:
23             i += len(substring)
24             offset = window.find(substring)
25             encoded +=
26                 ↪ f'{str(offset)},{str(len(substring))}${string[i]}'

```


25

26

```
return encoded
```