

Kierunek: **Informatyka Stosowana (IST)**
Specjalność: **Zastosowania Specjalistycznych Technologii Informatycznych**

**PRACA DYPLOMOWA
MAGISTERSKA**

**Opracowanie algorytmu generacji
grafu DSP do rozwiązania problemu
syntezy dźwięku**

**Automated generation of signal
processing graphs for sound synthesis**

Mateusz Bączek

Opiekun pracy
dr inż. Maciej Hojda

Słowa kluczowe: synteza, dźwięk, graf, optymalizacja, generatywna sztuczna inteligencja



Tekst zawarty w niniejszym szablonie jest udostępniany na licencji Creative Commons: *Uznanie autorstwa – Użycie niekomercyjne – Na tych samych warunkach, 3.0 Polska*, Wrocław 2023.

Oznacza to, że wszystkie przekazane treści można kopiować i wykorzystywać do celów niekomercyjnych, a także tworzyć na ich podstawie utwory zależne pod warunkiem podania autora i nazwy licencjodawcy oraz udzielania na utwory zależne takiej samej licencji. Tekst licencji jest dostępny pod adresem: <http://creativecommons.org/licenses/by-nc-sa/3.0/pl/>.

Licencja nie dotyczy latexowego kodu szablonu. Sam szablon (tj. zbiór przygotowanych komend formatujących dokument) można wykorzystywać bez wzmiankowania o jego autorze. Dlatego podczas redakcji pracy dyplomowej niniejszą stronę można usunąć.

Streszczenie

Praca prezentuje metodę automatycznej konstrukcji grafu przetwarzania sygnałów dźwiękowych, które wykonują syntezę zadanego przez użytkownika dźwięku. Wytworzony w ramach pracy algorytm może zostać wykorzystany jako narzędzie w pracy inżynierów dźwięku, podczas tworzenia nowych instrumentów elektronicznych lub efektów specjalnych. Wynikiem działania algorytmu wytworzzonego w ramach pracy jest zrozumiały dla człowieka graf przetwarzania sygnałów, przypominający konwencjonalne konfiguracje syntezatorów dźwięku wykorzystywane w programach do pracy nad dźwiękiem.

Słowa kluczowe: synteza, dźwięk, graf, optymalizacja, generatywna sztuczna inteligencja

Abstract

Thesis explores a technique for automated design of sound synthesizers, which can be conceptualized as signal processing graphs. Thesis introduces an algorithm for dynamically generating signal processing graphs that are both comprehensive and modifiable by end-users. By enabling users to easily understand and modify the generated graphs, the algorithm offers a versatile and user-friendly solution for composers and sound engineers, allowing them to create novel musical instruments or audio effects by cooperating with a synthesizer-designing algorithm.

Keywords: synthesis, sound, audio, dsp, graph, optimisation, generative

Spis treści

1. Wstęp	11
1.1. Cel pracy	13
1.1.1. Podstawy syntezy dźwięku	14
1.1.2. Generowanie grafu przetwarzania sygnałów	14
1.1.3. Funkcja celu oceniająca podobieństwo barwy dźwięku	15
1.1.4. Problem optymalizacyjny	15
1.2. Zakres pracy, plan badań	15
1.2.1. Metody generowania grafu przetwarzania sygnałów oraz późniejsza modyfikacja grafu	15
1.2.2. Dobór funkcji błędu: różnica między wygenerowanym a docelowym sygnałem dźwiękowym	16
1.3. Struktura i zawartość pracy	16
2. Definicja problemu	17
2.1. Budowa grafu	18
2.1.1. Struktura grafu	18
2.1.2. Przypisanie parametrów do „wolnych wejść”	18
2.2. Funkcja celu	19
2.2.1. Wyliczanie współczynników MFCC	19
2.2.2. Porównywanie wektorów MFCC z wykorzystaniem DTW	20
2.3. Ograniczenia	20
2.4. Definicja problemu optymalizacyjnego	20
3. Funkcja celu – porównanie barwy dźwięku	21
3.1. Porównanie barwy dźwięku w literaturze	21
3.1.1. Systematyzacja metod z literatury	21
3.1.2. Wybór funkcji celu do przetestowania	22
3.2. Proces testowania funkcji celu	22
3.3. Prezentacja wyników	22
3.4. Przekrój wartości funkcji celu dla prostego problemu syntezy typu FM	22
3.4.1. Analiza wyników	24
3.5. Optymalizacja parametrów dla predefiniowanych grafów syntezy FM oraz <i>analog modeling</i>	24
3.5.1. Synteza FM	25
3.5.2. Synteza <i>analog modeling</i>	25
3.5.3. Plan testów	27
3.5.4. Wyniki testów	27
3.5.5. Wybór funkcji celu na podstawie wyników	28
4. Algorytm rozwiązania	29
4.1. Wybór źródeł sygnału	30
4.1.1. Synteza FM	30

4.1.2. Synteza <i>analog modeling</i>	32
4.1.3. Synteza <i>physical modeling</i>	33
4.2. Wybór filtrów	33
4.3. Wybór efektów	34
5. Implementacja grafu przetwarzania sygnałów	36
5.1. Podstawy syntezy dźwięku w syntezatorach modułowych	36
5.2. Wymagania	37
5.2.1. Węzły DSP	37
5.2.2. Połączenia między węzłami – modulacja parametrów węzłów	40
5.2.3. Graf przetwarzania sygnałów	41
5.2.4. Automatyzacja pracy ze środowiskiem eksperymentowym za pośrednictwem języka Python	41
5.3. Opis zaimplementowanego środowiska eksperymentowego	41
5.3.1. Przykłady użycia	41
5.3.2. Detaile techniczne	42
6. Badania symulacyjne	44
6.1. Dźwięk fletu	45
6.2. Sampel z syntezatora <i>OP-1</i>	46
6.3. Transjent	48
6.4. Dźwięki inne niż przykłady z literatury	51
7. Analiza wyników, możliwe drogi dalszych badań	55
7.1. Zbiór danych testowych	55
7.2. Optymalizacja parametrów grafu dla problemów o małej złożoności	55
7.3. Potencjał wykorzystania w przemyśle muzycznym	56
7.4. Potencjalne usprawnienia wydajności algorytmu	56
7.5. Potencjalne drogi dalszego rozwoju algorytmu	57
7.5.1. Rozmiar okna w algorytmie DTW	57
7.5.2. Lepsza reprezentacja struktury grafu DSP w genotypie	57
7.5.3. Dalsze poszukiwania funkcji celu porównującej barwę sygnałów dźwiękowych	58
7.5.4. Trenowanie na coraz dłuższych fragmentach dźwięku	58
7.5.5. Rozszerzenie liczby dostępnych w grafie DSP węzłów	58
7.5.6. Rozszerzenie genotypu grafu DSP o dodatkowe źródła modulacji	58
7.6. Subiektywna natura porównania	58
Literatura	60
A. Instrukcja wdrożeniowa	63
A.1. Wymagane oprogramowanie	63
A.2. Instalacja	63
A.3. Weryfikacja poprawności działania projektu	63
A.4. Praca ze środowiskiem eksperymentowym	64
A.4.1. Funkcja celu i detaile algorytmu genetycznego	64
A.4.2. Implementacja algorytmów syntezy i grafu DSP	64
B. Opis załączonej płyty CD/DVD	65

Spis rysunków

1.1.	Zapis nutowy utworu <i>Opus One</i> , wygenerowany przez komputer <i>Lamus</i> [27]. Możliwe jest również generowanie utworów w formacie MIDI.	11
1.2.	Przykładowy spektrogram wygenerowany przez algorytm <i>Stable Riffusion</i> dla danych wejściowych funk <i>bassline with a jazzy saxophone solo</i>	12
1.3.	Syntezator <i>Mother 32</i> firmy <i>Moog</i> , po prawej stronie widoczny jest <i>patch bay</i> z podłączonymi przewodami, które nadpisują konfigurację połączeń między układami generującymi i przetwarzającymi sygnał dźwiękowy.	13
1.4.	Zbiór parametrów konfigurujących syntezator dźwięku <i>Wavetable</i> w programie <i>Ableton</i>	14
1.5.	Diagram blokowy pojedynczego głosu w syntezatorze <i>Minilogue xd</i> firmy <i>Korg</i> [31].	15
2.1.	Przykładowy węzeł w grafie, generujący sygnał sinusoidalny z możliwością modulacji fazy.	17
2.2.	Przykładowy graf DSP. Wolne wejścia, które nie są modulowane przez źródła sygnału w grafie są optymalizowanymi parametrami.	18
2.3.	Schemat algorytmu obliczania współczynników MFCC, zaczerpnięty z [26]. Praca wykorzystuje gotową implementację algorytmu obliczającego współczynniki MFCC z pakietu <i>librosa</i> [36].	19
3.1.	Przykład trzech próbek dźwięku, które dla słuchacza brzmią identycznie, mimo znacznych różnic w kształcie fali. Źródło obrazka: [17].	21
3.2.	Prosty graf syntezy FM, zawierający jeden oscylator służący za sygnał nośny i jeden oscylator służący za sygnał modulujący.	23
3.3.	Zmiany w wartościach testowanych funkcji celu podczas przesuwania różnych parametrów syntezy dźwięku. Kształt pierwszego wykresu wynika z zastosowania kwantyzacji dostępnych częstotliwości modulacji, aby wykluczyć nieharmoniczne stosunki częstotliwości modulacji i nośnej. Tego rodzaju praktyka jest wykorzystywana w syntezatorach FM [16], ponieważ ułatwia dostosowywanie parametrów syntezy.	24
3.4.	Spektrogram, kształt fali oraz wizualizacja MFCC dla próbki dźwięku, którą ma imitować graf syntezy FM podczas testów różnych funkcji celu.	25
3.6.	Spektrogram, kształt fali oraz wizualizacja MFCC dla próbki dźwięku, którą ma imitować graf syntezy <i>analog_modeling</i> podczas testów różnych funkcji celu.	25
3.5.	Graf wykonujący syntezę typu <i>analog modeling</i> , wykorzystany do testów funkcji celu.	26
3.7.	Wykresy zmian funkcji celu podczas optymalizacji dla grafu syntezy FM.	27
3.8.	Spektrogram dźwięku docelowego oraz dźwięków uzyskanych w procesie optymalizacji parametrów grafu FM. Czerwoną strzałką oznaczono składową harmoniczną (słabo widoczną na spektrogramie), która została poprawnie odtworzona przez algorytm optymalizacji.	27

3.9. Wykresy zmian funkcji celu podczas optymalizacji dla grafu syntezy <i>analog modeling</i>	28
3.10. Spektrogram dźwięku docelowego oraz dźwięków uzyskanych w procesie optymalizacji parametrów grafu <i>analog modeling</i>	28
4.1. Diagram algorytmu rozwiązania zaimplementowanego w ramach pracy. Algorytm oceny może wykorzystywać różne funkcje celu, finalnie zastosowano MFCC oraz <i>dynamic time wrapping</i> , proces wyboru funkcji celu opisuje rozdział 3.	29
4.2. Sekcje przetwarzania sygnałów oraz przykładowe węzły przetwarzania sygnałów, które są w nich powszechnie wykorzystywane.	30
4.4. Graf wykorzystujący gen FM1.	30
4.3. Przykład wygenerowanej struktury grafu, oznaczono segmenty z diagramu 4.2.	31
4.5. Graf wykorzystujący gen FM2.	32
4.6. Graf wykorzystujący gen AN1.	32
4.7. Graf wykorzystujący gen AN2.	33
4.8. Graf wykorzystujący gen AN3.	33
4.9. Graf wykorzystujący gen PM1.	33
4.10. Przykładowy łańcuch efektów w grafie.	34
5.1. Przykładowy układ modułów w standardzie <i>Eurorack</i> [1]. W prawym dolnym rogu widoczne połączenia modulujące między modułami.	36
5.2. Przykładowy układ węzłów DSP w zaimplementowanym środowisku eksperymentalnym. Układ wykonuje syntezę subtraktywną z modulowaną wartością częstotliwości granicznej filtra niskoprzepustowego oraz dodaje efekt pogłosu (<i>reverb</i>) [15].	38
5.3. Węzeł DSP w zaimplementowanym środowisku eksperymentalnym, generujący falę sinusoidalną z możliwością modulacji fazy.	39
5.4. Moduł syntezy <i>Mutable Instruments Elements</i> , umożliwiający ręczne ustawianie parametrów oraz ich modulację za pomocą sygnału CV. Moduł wykonuje syntezę typu <i>physical modeling</i> [23].	39
5.5. Przykładowa modulacja parametru <code>input_modulation</code> za pomocą sygnału sinusoidalnego, charakterystyczna dla syntezy typu FM [32].	40
5.6. Spektrogram oraz wykres sygnału wygenerowanego za pomocą układ z rysunku 5.2.2. Widoczne dodatkowe składowe harmoniczne wpływające na barwę dźwięku.	40
5.7. Spektrogram oraz wykres sygnału wygenerowanego przez układ z rysunku 5.2.2 po usunięciu połączenia modulującego fazę oscylatora #2. Widoczna tylko jedna składowa harmoniczna: częstotliwość podstawowa.	40
5.8. Wynik wykonania kodu przedstawionego w listingu 5.2 w środowisku <i>Jupyter Notebook</i> , wizualizacja utworzonego grafu.	42
6.1. Spektrogram i wykres fali dla dźwięku <code>flute.wav</code> wykorzystywanego do eksperymentów w [33].	45
6.2. Spektrogram i wykres fali dla dźwięku <code>flute.wav</code> wytworzonyego przez zaimplementowany algorytm optymalizacji dla dźwięku <code>flute.wav</code>	45
6.3. Spektrogram i wykres fali dla dźwięku wygenerowanego przez algorytm z literatury [34] na wzór <code>flute.wav</code>	46
6.4. Zmiany wartości funkcji celu podczas optymalizacji.	46
6.5. Spektrogram i wykres fali dla dźwięku <code>op1_1.wav</code> wykorzystywanego do eksperymentów w [33].	47
6.6. Spektrogram i wykres fali dla dźwięku wygenerowanego przez algorytm optymalizacji na wzór <code>op1_1.wav</code>	47

6.7. Spektrogram i wykres fali dla dźwięku wygenerowanego przez algorytm z literatury [34] na wzór op1_1.wav.	48
6.9. Zmiany wartości funkcji celu podczas optymalizacji.	48
6.8. Graf DSP wygenerowany przez zaimplementowany algorytm dla dźwięku docelowego op1_1.wav.	49
6.10. Spektrogram i wykres fali dla dźwięku transient.wav wykorzystywanego do eksperymentów w [33].	50
6.11. Spektrogram i wykres fali dla dźwięku wygenerowanego na wzór transient.wav	50
6.12. Spektrogram i wykres fali dla dźwięku wygenerowanego przez algorytm z literatury [34] na wzór transient.wav	51
6.13. Zmiany wartości funkcji celu podczas optymalizacji.	51
6.14. Graf DSP wygenerowany przez zaimplementowany algorytm dla dźwięku docelowego transient.wav	52
6.15. Spektrogram i wykres fali dla dźwięku wygenerowanego na syntezatorze <i>Korg Minilogue xd</i>	53
6.16. Spektrogram i wykres fali dla dźwięku wygenerowanego przez zaimplementowany w ramach pracy algorytm na wzór 6.15.	53
6.17. Graf DSP wygenerowany przez zaimplementowany algorytm dla dźwięku docelowego 6.15. Poprawnie został odtworzony filtr niskoprzepustowy oraz sterujący nim sygnał ADSR.	54
7.1. Wizualizacja danych wygenerowanych przez profiler języka Python dla przykładowego problemu optymalizacji. Widoczne składowe wpływające na sumaryczny czas ewaluacji funkcji celu. Czerwoną strzałką oznaczono czas poświęcony na syntezę dźwięku w grafie DSP.	56
7.2. Porównanie czasu obliczania współczynników MFCC (<code>dsp.py:115</code>) oraz czasu działania algorytmu DTW (<code>dsp.py:171</code>).	57

Spis listingów

5.1. Implementacja węzła SineOscillator.	39
5.2. Utworzenie prostego grafu generującego sygnał sinusoidalny.	41
5.3. Typ danych zwracanych przez środowisko eksperymentalne.	42

Skróty

DAW (ang. *Digital Audio Workstation*) – typ oprogramowania dostępnego na komputery osobiste, służące do komponowania utworów muzycznych. Popularne programy typu DAW to między innymi *Ableton*, *Apple Logic Pro* i *FL Studio*.

STFT (ang. *Short-time Fourier Transform*) – wariant transformaty Fouriera, wykonujący transformatę na ruchomym okno przesuwające się wzdłuż analizowanego sygnału. **STFT** pozwala na zwiększenie dokładności transformaty dla sygnałów o dużej zmienności w czasie. W kontekście syntezy audio, **SFTP** zwiększa dokładność z jaką rejestrowane są transjenty, czyli dynamiczne zmiany charakterystyki barwy dźwięku w czasie.

CV (ang. *Control Voltage*) – Sygnał sterujący parametrami syntezy dźwięku, standardowo wykorzystywanych w syntezatorach modułowych (przykładowo w standardzie *EuroRack*). Sygnał **CV** wykorzystuje się do przekazywania sygnałów kontrolnych między modułami.

VCO (ang. *Voltage Controlled Oscillator*) – komponent elektroniczny generujący sygnał dźwiękowy. Parametry generowanego sygnału sterowane są za pomocą napięcia kontrolnego (**CV**).

VCF (ang. *Voltage Controlled Filter*) – komponent elektroniczny wykonujący filtrację dźwięku w domenie częstotliwości. Parametry filtra sterowane są za pomocą napięcia kontrolnego (**CV**).

Rozdział 1

Wstęp

Rozpowszechnione algorytmy sztucznej inteligencji wspomagające pracę inżynierów dźwięku i kompozytorów można podzielić na trzy główne kategorie [25] [14] :

1. algorytmy generujące symboliczny zapis muzyki (rysunek 1.1), [43],
2. algorytmy generujące gotowy plik audio na podstawie opisu użytkownika (rysunek 1.2).
3. algorytmy symulujące brzmienie instrumentów muzycznych [18].

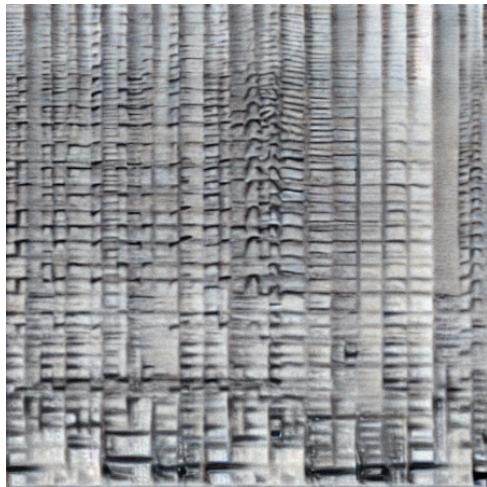
Pierwsza grupa algorytmów znana jest już od lat 80, gdyż zagadnienie generowania zapisu symbolicznego wymaga mniej mocy obliczeniowej niż wytworzenie pełnego pliku audio. Powszechnie wykorzystywana jest w nich teoria muzyki, pozwalająca określić matematyczne relacje występujące w rytmach, melodiach i progresjach akordów. Wiedza dotycząca teorii muzyki pozwala na wyznaczenie możliwej przestrzeni stanów, w której generowana jest kompozycja, natomiast modele matematyczne takie jak łańcuchy Markowa służą za mechanizmy decyzyjne.



Rys. 1.1: Zapis nutowy utworu *Opus One*, wygenerowany przez komputer *Lamus* [27]. Możliwe jest również generowanie utworów w formacie MIDI.

Druga grupa algorytmów, generująca pliki audio, bazuje na klasie algorytmów wywodzących się ze *Stable Diffusion* [37]. Modele generujące pliki audio zgodne z opisem tekstowym

(przykładowo „smutny jazz” bądź „muzyka taneczna w stylu Depeche Mode”) szkolene są w taki sam sposób jak algorytmy *stable diffusion*, dane treningowe składają się z obrazów spektrogramów [20]. Po trenowaniu, model jest w stanie wygenerować spektrogram zawierający utwór muzyczny zgodny z poleceniem użytkownika (1.2). Wygenerowany przez model spektrogram jest konwertowany do sygnału dźwiękowego za pomocą odwrotnej transformaty Fouriera.



Rys. 1.2: Przykładowy spektrogram wygenerowany przez algorytm *Stable Riffusion* dla danych wejściowych funk bassline with a jazzy saxophone solo.

Trzecia grupa algorytmów, symulowanie brzmienia instrumentów muzycznych, najczęściej wykorzystuje sieci neuronowe trenowane na brzmieniu prawdziwych instrumentów [18] [17]. Tego typu algorytmy pozwalają symulowanie instrumentów o złożonych barwach, takich jak instrumenty smyczkowe lub dęte oraz na płynne przechodzenie między brzmieniami różnych instrumentów muzycznych.

Metody opisane w rozdziale 1 można porównać pod względem ich przydatności dla użytkownika końcowego, czyli osoby zajmującej się produkcją nagrań muzycznych. Metoda pierwsza, generowanie zapisu symbolicznego, może wydawać się mniej zaawansowana niż generowanie całych plików dźwiękowych. Jednakże, z perspektywy użytkownika, zapis symboliczny jest bardziej praktyczny, ponieważ możliwe jest zaimportowanie go do programu DAW i późniejsza modyfikacja zapisu nutowego. Obecnie dostępne modele generujące pełne nagrania z muzyką nie umożliwiają szczegółowego edytowania parametrów wygenerowanego dźwięku, ponieważ operują bardzo wysokopoziomowo – syntezują muzykę na podstawie opisu słownego. Podobne problemy występują również podczas wykorzystywania algorytmów z grupy trzeciej, głębokie sieci neuronowe nie są przystosowane do ręcznej modyfikacji przez użytkownika.

Podsumowując, wykorzystanie wygenerowanego przez komputer zapisu nutowego jest proste, ze względu na symboliczną naturę zapisu. Wykorzystanie wygenerowanego przez komputer dźwięku jest ograniczone ze względu na fakt, że do generowania złożonych sygnałów dźwiękowych wykorzystywane są techniki takie jak głębokie sieci neuronowe, w których utrudniona jest dokładna kontrola nad konkretnymi parametrami funkcjonowania sieci.

Niniejsza praca sugeruje nową metodę podejścia do problemu generowania sygnałów dźwiękowych, którego nie da się zaklasyfikować do żadnej z wyżej wymienionych (1) dziedzin komputerowej kompozycji muzycznej. Graf przetwarzania sygnałów wytworzony przez algorytm implementowany w ramach pracy magisterskiej jest przepisem na gotowy elektroniczny instrument muzyczny, który może być wykorzystany w programie do komponowania muzyki. Algorytm nie generuje bezpośrednio sygnału dźwiękowego, lecz tworzy graf przetwarzania sygnałów, który jest zrozumiały dla użytkownika i pozwala na precyzyjne dostosowanie parametrów

syntezy. Tego typu proces generowania grafów przetwarzania sygnałów dźwiękowych może być porównany z procesem projektowania instrumentu muzycznego.

Modyfikowanie ścieżki przetwarzania sygnału jest techniką często wykorzystywaną w muzyce elektronicznej, do tworzenia dźwięków o interesującej barwie bądź dynamice. Syntezatory dźwięku dostępne na rynku często wyposażone są w tzw. *patch bay* (1.3), pozwalający na modyfikowanie grafu przepływu sygnałów wewnątrz syntezatora, bądź połączenie go z zewnętrznym sprzętem muzycznym bądź elektronicznym.



Rys. 1.3: Syntezator *Mother 32* firmy *Moog*, po prawej stronie widoczny jest *patch bay* z podłączonymi przewodami, które nadpisują konfigurację połączeń między układami generującymi i przetwarzającymi sygnał dźwiękowy.

1.1. Cel pracy

Celem pracy jest **opracowanie algorytmu generacji grafów przetwarzania sygnałów, który wykona syntezę próbki dźwięku zadanej przez użytkownika**. Problem poruszany w pracy można zakwalifikować do grupy zagadnień związanych z pojęciami *computer-aided design* oraz *generative artificial intelligence*, zastosowanymi w dziedzinie inżynierii dźwięku. Docelowo zaimplementowany algorytm będzie automatyzował pracę inżyniera dźwięku, tworząc i konfiguruje grafy przetwarzania sygnałów dźwiękowych, dostępne w programach typu *digital audio workstation* (1.4). Badania obejmują dwa zagadnienia:

1. **opracowanie metody generowania grafu przetwarzania sygnałów oraz późniejszej modyfikacji grafu – jego struktury i parametrów,**
2. **dobór funkcji celu, na podstawie której algorytm optymalizujący będzie modyfikował graf przetwarzania sygnałów.**
3. **przeprowadzenie badań symulacyjnych, które zweryfikują skuteczność opracowanego algorytmu.**
4. **porównanie wyników badań symulacyjnych z pracami naukowymi o podobnej tematyce.**



Rys. 1.4: Zbiór parametrów konfigurujących syntezator dźwięku *Wavetable* w programie Ableton

1.1.1. Podstawy syntezy dźwięku

W syntezie dźwięku wykorzystuje się algorytmy generujące i przetwarzające sygnały w zakresie częstotliwości słyszalnych. Praca wykorzystuje powszechnie używane typy algorytmów syntezy, opisane w [32].

Syntez FM

Syntez FM wykorzystuje sygnał sinusoidalny jako źródło sygnału podstawowego (*carrier*). Zróżnicowane barwy dźwięku uzyskiwane są przez modulowanie częstotliwości sinusoidy podstawowej przez inne sygnały sinusoidalne (*modulators*), które zazwyczaj generują sygnał o częstotliwości będącej wielokrotnością częstotliwości podstawowej [40]. Siła modulacji może być zmieniana w trakcie syntezy, co powoduje zmiany w odczuwanej barwie dźwięku.

Syntez subtraktynwa

W syntezie subtraktywnej wykorzystuje się sygnały dźwiękowe o dużej liczbie składowych harmonicznych, takie jak sygnał piłokształtny lub kwadratowy. Podstawową barwę dźwięku uzyskuje się przez dobór kilku sygnałów o różnych kształtach, następnie sygnał jest filtrowany za pomocą filtrów dolno-, górnego- i pasmoprzepusowych [38], aby usunąć wybrane przez użytkownika składowe częstotliwościowe. Dynamiczne zmiany w barwie dźwięku uzyskuje się poprzez modulowanie częstotliwości odcięcia filtra.

Algorytmy *physical modeling*

Algorytmy typu *physical modeling* wykorzystują uproszczone modele fizyczne rzeczywistych obiektów wytwarzających sygnały dźwiękowe, takie jak struny czy membrany [23].

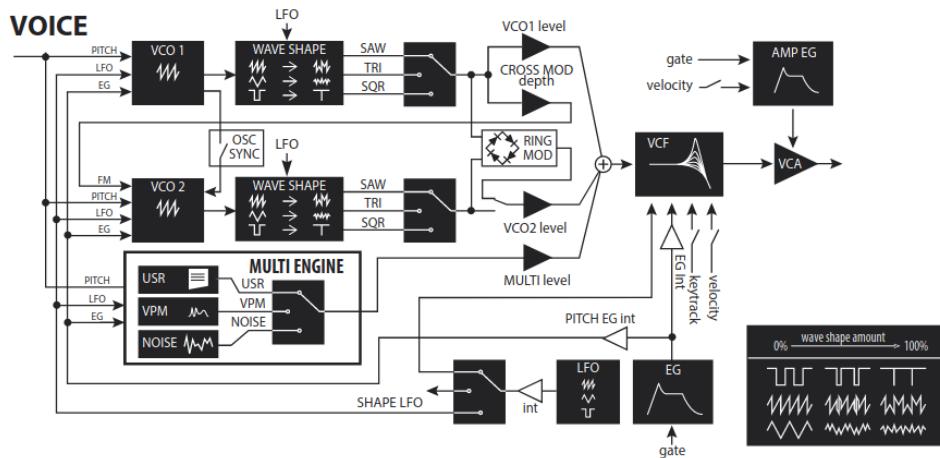
symulacja efektu pogłosu/echo [15] [39]

Sygnał wygenerowany przez dany algorytm syntezy (lub połączenie wielu algorytmów) może być przetworzony przez algorytmy symulujące echo lub pogłos (*reverb, delay*). Tego typu algorytmy naśladują roznoszenie się dźwięku w pudle rezonansowym instrumentu bądź w dużej przestrzeni (sala koncertowa, jaskinia).

1.1.2. Generowanie grafu przetwarzania sygnałów

Proces syntezy dźwięku często jest przedstawiany jako graf przetwarzania sygnałów, w którym każdy węzeł wykonuje na sygnale określoną operację. Przykładowy graf przetwarzania sygnału dla syntezatora analogowego subtraktynwego przedstawiony jest na schemacie 1.5. Pierwsze zagadnienie poruszane w pracy sprowadza się do opracowania algorytmu pozwalającego na wygenerowanie grafu przetwarzania sygnałów DSP oraz jego późniejszą modyfikację. Przykładem

modyfikacji grafu może być wprowadzanie do niego nowych źródeł modulacji bądź zmiana algorytmu generującego sygnał.



Rys. 1.5: Diagram blokowy pojedynczego głosu w syntezatorze *Minilogue xd* firmy *Korg* [31].

1.1.3. Funkcja celu oceniająca podobieństwo barwy dźwięku

Drugie zagadnienie obejmuje przetestowanie szeregu algorytmów, które można wykorzystać do zbudowania funkcji celu, która będzie optymalizowana poprzez „dostrajanie” parametrów i struktury grafu przetwarzania sygnałów dźwiękowych. Problem porównywania barwy dwóch sygnałów dźwiękowych nie jest problemem trywialnym, ponieważ wymaga zamodelowania wrażeń psychoakustycznych [17] odczuwanych podczas odsłuchu próbki dźwięku. Skuteczność danej funkcji celu finalnie musi być poddana subiektywnej ocenie, ponieważ nie istnieją obiektywne metryki mierzące z natury subiektywne odczucia słuchacza. Praca proponuje wykorzystanie współczynników cepstralnych sygnału (MFCC) w połączeniu z dynamicznym skalowaniem czasu (*dynamic time wrapping*, DTW) jako funkcji celu. Proces wyboru funkcji celu został opisany w rozdziale 3.

1.1.4. Problem optymalizacyjny

W pracy rozwiązywany jest problem optymalizacyjny, w którym struktura oraz parametry grafu przetwarzania sygnałów dostosowywane są tak, aby wygenerować zadaną próbkę dźwięku. Tak wytworzony graf może być wykorzystany jako elektroniczny instrument muzyczny.

1.2. Zakres pracy, plan badań

1.2.1. Metody generowania grafu przetwarzania sygnałów oraz późniejsza modyfikacja grafu

Głównym problemem przy generowaniu grafu przetwarzania sygnałów są ograniczenia nałożone na strukturę grafu, które należy spełnić, by graf był logicznie interpretowalny jako łańcuch przetwarzania sygnałów. Graf musi być grafem skierowanym, który nie zawiera pętli o dodatnim sprzężeniu zwrotnym. Automatyczna ewolucja może dążyć w kierunku wykorzystania nadmiarowej liczby bloków przetwarzania sygnału, jeśli funkcja celu nie będzie zawierała kary za zbyt złożone grafy. Podobne prace [34] wykorzystują podejście oparte o *mixed-typed carthe-*

sian genetic programming, które będzie punktem startowym dla pracy. Finalnie, badania dążą do wyznaczenia algorytmu o następujący właściwościach:

1. algorytm generuje grafy będące logicznie spójnymi łańcuchami przetwarzania dźwięku (skierowany, bez pętli o dodatnim sprzężeniu zwrotnym w natężeniu sygnału),
2. algorytm maksymalizuje wykorzystanie poszczególnych bloków przetwarzania w grafie, co minimalizuje finalny rozmiar grafu, czyniąc go bardziej czytelnym,
3. generowany graf posiada reprezentację umożliwiającą wykonanie krzyżowania dwóch grafów przetwarzania sygnału. Graf będący wynikiem krzyżowania nadal musi być poprawnym grafem przetwarzania sygnału.

Elementami grafu przetwarzania sygnałów są używane powszechnie w syntezie dźwięku algorytmy, opisane w sekcji 1.1.1.

1.2.2. Dobór funkcji błędu: różnica między wygenerowanym a docelowym sygnałem dźwiękowym

Funkcja celu poszukiwana w ramach projektu musi określać, jak dobrze sygnał wygenerowany przez graf przetwarzania sygnałów pokrywa się z sygnałem docelowym. Porównanie sygnałów musi skupiać się na cechach sygnału, które są najbardziej słyszalne dla ludzkiego ucha. Jednocześnie funkcja nie powinna „karać” sygnałów, które są względem siebie przesunięte w fazie. Wśród algorytmów, które zostały wybrane do przetestowania w ramach projektów zawarte są:

1. algorytmy porównywania sygnałów oparte o transformatę Fouriera [24] [44],
2. techniki wykorzystywane do generowania „cyfrowych podpisów” sygnałów dźwiękowych (*sound fingerprinting*) [28],
3. algorytmy wykrywające spadek jakości dźwięku z perspektywy psychoakustycznej [29] [30].

1.3. Struktura i zawartość pracy

Rozdział **Definicja problemu** formalizuje i opisuje problem optymalizacyjny rozwiązywany w pracy. W rozdziale **Analiza i wybór funkcji celu** opisano proces porównywania funkcji z dziedziny przetwarzania sygnałów, które pozwalają określić jak podobna jest barwa dźwięku dwóch sygnałów dźwiękowych i uzasadniono wybór funkcji celu, która została zastosowana w pracy. Rozdział **Algorytm rozwiązań** opisuje algorytm wykorzystany do rozwiązania problemu zdefiniowanego w rozdziale 2. W rozdziale **Implementacja grafu przetwarzania sygnałów** opisane zostało zaimplementowane w ramach pracy środowisko eksperymentalne, pozwalające na wytwarzanie grafów przetwarzania sygnałów o dowolnej strukturze. Przedstawione są w nim również zaimplementowane algorytmy syntezy i przetwarzania sygnałów dźwiękowych. Rozdział **Badania symulacyjne** opisuje proces badawczy, w którym narzędzią wytworzone w rozdziałach 5 oraz 3 zostały wykorzystane do automatycznegotworzenia grafu DSP, który naśladuje barwę zadanej próbki dźwięku. Porównuje uzyskane wyniki z podobną pracą badawczą [34]. Rozdział **Analiza wyników, możliwe drogi dalszego rozwoju** podsumowuje uzyskane wyniki badań, podejmuje dyskusję nad ogólną skutecznością i przydatnością zaimplementowanego rozwiązania oraz kreśli potencjalne drogi dalszego rozwoju prac badawczych w podobnej tematyce. W czasie, gdy niniejsza praca była tworzona, zostały opublikowane badania dotyczące podobnego problemu [19], rozdział podejmuje dyskusję o różnicach w podejściu do problemu oraz potencjalnych zalet i wad każdego z podejść. **Dodatek A** zawiera instrukcję wdrożeniową dla zaimplementowanego w ramach pracy środowiska eksperymentalnego, w **Dodatku B** opisana jest zawartość załączonej do pracy płyty DVD.

Rozdział 2

Definicja problemu

Graf przetwarzania sygnałów można opisać jako zbiór połączonych węzłów generujących i przetwarzających sygnał dźwiękowy. Każdy węzeł opisany jest poprzez:

1. zbiór wejść,
2. zbiór wyjść,
3. procedurę obliczeniową wykonywaną na sygnale.

SineOscillator #2
o input_frequency: 0.000
o input_modulation: 0.000
o input_modulation_index: 0.100
output_output •

Rys. 2.1: Przykładowy węzeł w grafie, generujący sygnał sinusoidalny z możliwością modulacji fazy.

Przykładowo, dla syntezy subtraktywnej powszechnie wykorzystywane są następujące typy węzłów:

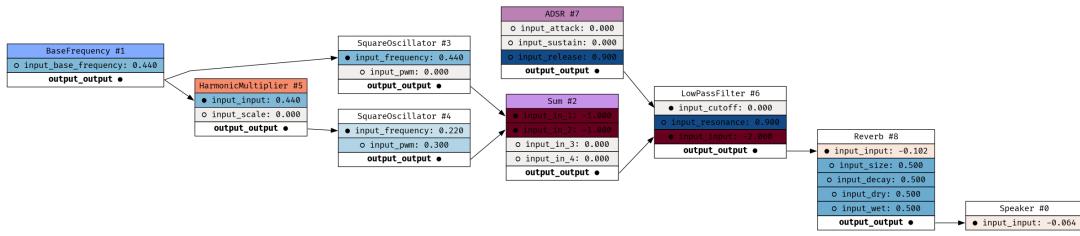
Oscylator (VCO):

1. Wejścia:
 - częstotliwość,
 - kształt fali.
2. Wyjścia:
 - wygenerowany sygnał.

Filtr (VCF):

1. Wejścia:
 - sygnał wejściowy
 - częstotliwość odcięcia,
 - rezonans.
2. Wyjścia:
 - przefiltrowany sygnał.

2.1. Budowa grafu



Rys. 2.2: Przykładowy graf DSP. Wolne wejścia, które nie są modulowane przez źródła sygnału w grafie są optymalizowanymi parametrami.

Pełny graf przetwarzania można opisać za pomocą zbioru węzłów oraz listy połączeń między węzłami:

$$N = [n_1, n_2, \dots, n_n] - \text{liczba węzłów},$$

$i_j = [p_1, p_2, \dots, p_m]$ – Zbiór wejść (*inputs*) j-go węzła. $i_{l,m}$ oznacza m-te wejście l-go węzła.

$o_j = [p_1, p_2, \dots, p_m]$ – Zbiór wyjść (*outputs*) j-go węzła,

$f_i(x)$ – operacja wykonywana na sygnale przez i-ty węzeł.

$C = [\{(j, k), (l, m)\}, \dots]$: lista połączeń między węzłami, opisujący, które k-te wyjście j-go węzła podłączone jest do którego m-go wejścia l-go węzła. Przykładowo, dla diagramu 2.1, jednym z połączeń będzie $\{(1, 0), (5, 0)\}$, ponieważ zerowe wyjście węzła BaseFrequency #1 podłączone jest do zerowego wejścia węzła HarmonicMultiplier #5.

Nie wszystkie wejścia w grafie muszą być podłączone do któregoś z wyjść, co jest widoczne na diagramie 2.1. Wejście, które nie zostało nigdzie podłączone przyjmuje jako wartość wejściową parametr liczbowy, optymalizowany na podstawie wartości funkcji celu (2.13). W przypadku schematu 1.5 takimi „wolnymi” wejściami są przykładowo sygnał określający częstotliwość odcięcia filtru sygnału lub parametry określające parametry generatora obwiedni (*EG*). Graf przetwarzania sygnałów wykorzystywany w pracy wytwarzany jest na podstawie dwóch list parametrów sterujących:

1. parametry definujące strukturę grafu, $S = [s_1, s_2, \dots]$,
2. lista wartości parametrów w wolnych wejściach, $P = [p_2, p_2, \dots]$.

2.1.1. Struktura grafu

Różne rodzaje syntezy dźwięku wykorzystują różnorodne struktury grafu przetwarzania sygnałów [31] [16]. Aby umożliwić dostosowanie grafu przetwarzania sygnałów do wykonywania różnych rodzajów syntezy, struktura grafu przetwarzania sygnałów jest dynamicznie modyfikowana przez algorytm optymalizacji. Algorytm generujący określoną strukturę grafu na podstawie genotypu opisany jest w rozdziale 4. Genotyp odpowiadający za strukturę grafu ma formę krotki liczb rzeczywistych S . Praca definiuje funkcję generującą strukturę grafu G_s z genotypu:

$$G_s(S) = N, C \quad (2.1)$$

2.1.2. Przypisanie parametrów do „wolnych wejść”

Po stworzeniu grafu o danej strukturze G_s , druga część genotypu wykorzystywana jest jako wartości poszczególnych parametrów P dla wolnych wejść w grafie przetwarzania sygnału.

$$\forall_{(l,m)} (l, m) \notin C, i_{l,m} = P_j \quad (2.2)$$

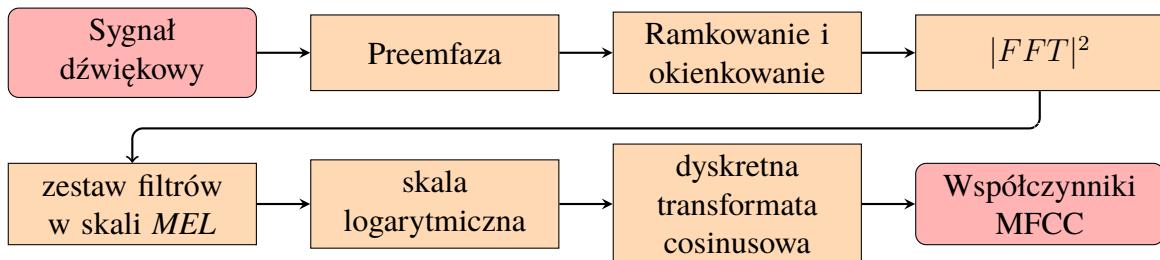
2.2. Funkcja celu

Wykorzystana w pracy funkcja celu F , oceniająca, jak sygnał wygenerowany (\bar{x}) przez algorytm jest bliski sygnałowi docelowemu (x) przedstawiona jest w następujący sposób:

$$F(x, \bar{x}) = DTW(MFCC(x), MFCC(\bar{x})) \quad (2.3)$$

Gdzie $MFCC$ oznacza *mel-frequency cepstrum coefficients* (2.2.1), natomiast DTW oznacza algorytm *dynamic time warping* [41]. Uzasadnienie wybranej funkcji celu opisane jest w rozdziale 3.

2.2.1. Wyliczanie współczynników MFCC



Rys. 2.3: Schemat algorytmu obliczania współczynników MFCC, zaczerpnięty z [26]. Praca wykorzystuje gotową implementację algorytmu obliczającego współczynniki MFCC z pakietu `librosa` [36].

Algorytm obliczania współczynników MFCC przedstawiony jest na rysunku 2.2.1. Pierwszym krokiem w algorytmie jest zastosowanie preemfazy, która wzmacnia składowe wysokoczęstotliwościowe i osłabia składowe niskoczęstotliwościowe:

$$x'_n = x_n - a x_{n-1} \quad (2.4)$$

W następnym kroku sygnał jest ramkowany, na każdą ramkę nakładane jest okno Hamminga:

$$Ham(N) = 0.54 - 0.46 \cos\left(2\pi \frac{n-1}{N-1}\right) \quad (2.5)$$

Dla każdej ramki wyliczana jest transformata Fouriera, aby obliczyć widmo mocy sygnału $|FFT|^2$. Widmo przetwarzane jest przez zbiór filtrów H_m , których środki są rozmiieszczone w równomiernych odstępach w skali mel. Typ i liczba filtrów zależy od implementacji algorytmu (w pracy wykorzystano [36]). Wyjście każdego z filtrów wykorzystane jest do obliczenia energii przefiltrowanego pasma:

$$S_m = \sum_{k=1}^N |X_r(k)|^2 H_m(k), \quad (2.6)$$

gdzie X_r oznacza widmo danej ramki, m jest numerem filtra.

W ostatnim kroku do obliczenia wartości współczynników MFCC wykorzystuje się dyskretną transformatę kosinusową. Aby lepiej przybliżyć wrażliwość ludzkiego ucha na głośność dźwięku, wykorzystuje się logarytm energii pasma:

$$c_i = \sqrt{\frac{2}{M}} \sum_{m_1}^M \log(S_m) \cos\left(\frac{\pi i}{M}(M-0.5)\right), \quad (2.7)$$

Gdzie M to liczba użytych filtrów w zbiorze 2.6, a i jest numerem współczynnika.

2.2.2. Porównywanie wektorów MFCC z wykorzystaniem DTW

Algorytm DTW służy do porównywania sekwencji punktów lub wektorów, które różnią się między sobą pod względem prędkości zmian w czasie [22] [41]. Dla danych sekwencji $A = (p_1, \dots, p_n)$, $B = (q_1, \dots, q_m)$ oraz funkcji $dist(p_i, q_j)$ obliczającej różnicę między dwoma dowolnymi elementami w sekwencjach, zdefiniowane są możliwe połączenia (*coupling*) $C = (c_1, \dots, c_k)$:

$$c_r(p_i, q_j) = c_{r+1} \in \{(p_{i+1}, q_j), (p_i, q_{j+1}), (p_{i+1}, q_{j+1})\}, \quad (2.8)$$

Dystans między sekwencjami A i B to takie połączenie c_r , dla którego suma dystansów $dist(p_i, q_j)$ między elementami sekwencji w przypisaniu jest najmniejsza:

$$dtw(A, B) = \min_{C:coupling} \left\{ \sum_{(p_i, q_j) \in C} dist(p_i, q_j) \right\} \quad (2.9)$$

Funkcja $dist(p_i, q_j)$ wykorzystana w pracy to długość wektora będącego różnicą między p_i i q_j :

$$dist(p_i, q_j) = \|p_i - q_j\| \quad (2.10)$$

2.3. Ograniczenia

W algorytmach DSP powszechnie wykorzystuje się ograniczenie wartości sygnału do przedziału $(-1.0, 1.0)$, gdy implementacja danego algorytmu wykorzystuje typ `float` do przechowywania wartości sygnałów. Środowisko zaimplementowane w ramach pracy oczekuje, że wartości sygnałów i parametrów sterujących będą znajdowały się w tym przedziale:

$$\forall p_j \in P, -1.0 \leq p_j \leq 1.0, \quad \forall s_i \in S, -1.0 \leq s_i \leq 1.0 \quad (2.11)$$

2.4. Definicja problemu optymalizacyjnego

Sygnal wygenerowany przez graf przetwarzania sygnałów zależy od następujących parametrów:

$$\bar{x} = G(G_s(S), P) \quad (2.12)$$

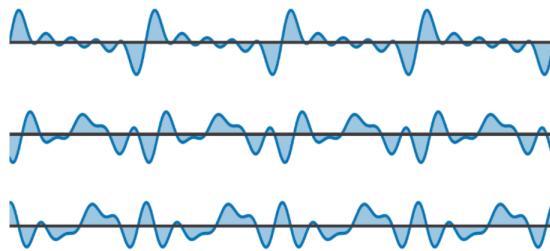
Gdzie G jest funkcją generującą sygnał dźwiękowy dla konkretnej struktury grafu G_s (2.1.1, 2.1), dla wartości parametrów przypisanych z P (2.1.2, 2.2). Dla parametrów S oraz P , ograniczonych przez 2.11, rozwiązywany jest problem optymalizacyjny:

$$\text{Minimize } F(x, G(G_s(S), P)) \quad (2.13)$$

Rozdział 3

Funkcja celu – porównanie barwy dźwięku

Aby stopniowo dostosować graf przetwarzania sygnałów tak, aby imitował zadaną próbkę dźwięku, należy wykorzystać funkcję celu, która maleje wraz ze wzrostem podobieństwa barwy dźwięku między próbki zadaną i sygnałem generowanym przez graf.



Rys. 3.1: Przykład trzech próbek dźwięku, które dla słuchacza brzmią identycznie, mimo znacznych różnic w kształcie fali. Źródło obrazka: [17].

3.1. Porównanie barwy dźwięku w literaturze

Żadna z prac przeanalizowanych podczas przeglądu literatury ([17], [19], [12], [20], [34], [11], [41]) nie wykorzystuje metod porównywania sygnału osadzonych jedynie w dziedzinie czasu, ponieważ nie są one skuteczne do porównywania dźwięków pod względem odczuć psychoakustycznych. Przykład różnych kształtów fali, które z perspektywy słuchacza brzmią jak taki sam dźwięk zademonstrowano na rysunku 3.

Ponieważ porównywanie barwy dźwięku instrumentów muzycznych nie należy do popularnych tematów badań, podczas przeglądu literatury wykorzystano również badania dotyczące rozpoznawania głosu, wykorzystujące współczynniki MFCC oraz *dynamic time warping* (DTW) [41].

3.1.1. Systematyzacja metod z literatury

Metody zaczerpnięte z literatury wykorzystują różne podejścia do porównywania barwy dźwięku pomiędzy sygnałami. Podejścia te można usystematyzować za pomocą dwóch cech.

1. Rodzaj wykonanej transformacji z dziedziny czasu do dziedziny częstotliwości:

- transformata Fouriera (w różnych konfiguracjach) [20] [12],
 - MFCC [19] [34] [41].
2. Dalsze przetwarzanie reprezentacji sygnału w domenie częstotliwości, w celu ułatwienia optymalizacji:
- dostosowywanie wagi konkretnych próbek na podstawie metryki określającej siłę sygnału (na przykład *root-mean-square*, RMS) [11], aby wzmacnić istotność głośniej-szych fragmentów dźwięku,
 - wykorzystanie *dynamic time warping*, aby funkcja celu przyzwała na niedokładności w odwzorowaniu dokładnej dynamiki zmian w charakterystyce spektralnej [41].

3.1.2. Wybór funkcji celu do przetestowania

Na podstawie analizy metod z literatury opisanej w rozdziale 3.1.1 zostały wybrane wszystkie warianty funkcji celu wykorzystywane w przeanalizowanej literaturze:

1. Różnica w spektrum Fouriera,
2. Różnica w spektrum Fouriera liczona za pomocą DTW,
3. Różnica w MFCC,
4. Różnica w MFCC liczona za pomocą DTW,
5. Różnica w MFCC ważonym za pomocą RMS.

3.2. Proces testowania funkcji celu

Metoda testowania została zaczerpnięta z [34]. Funkcje celu zostały wpierw przetestowane poprzez wykonanie zbioru przekrojów przez uproszczony problem syntezy typu FM. Następnie przeprowadzono próby automatycznego dostosowania parametrów dwóch grafów o predefiniowanej strukturze, dla syntezy FM oraz *analog modeling*. Ponieważ barwa dźwięku jest odczuciem z natury subiektywnym [25], ostateczna decyzja dotycząca wyboru funkcji celu została dokonana przez autora pracy, proces decyzyjny został opisany w sekcjach 3.4.1 oraz 3.5.4.

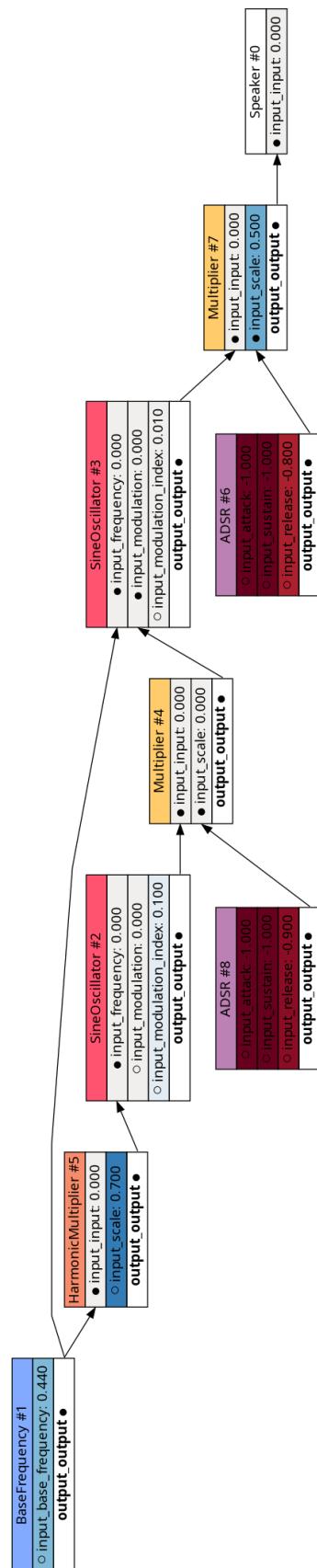
3.3. Prezentacja wyników

Próbki dźwięku (docelowe oraz wygenerowane) zaprezentowane są w pracy w formie spektrogramów, ponieważ taki format jest najbardziej czytelny dla człowieka [32] i pozwala ocenić skuteczność algorytmu optymalizacji. Inne prace w podobnych dziedzinach również wykorzystują spektrogramy [34] [17].

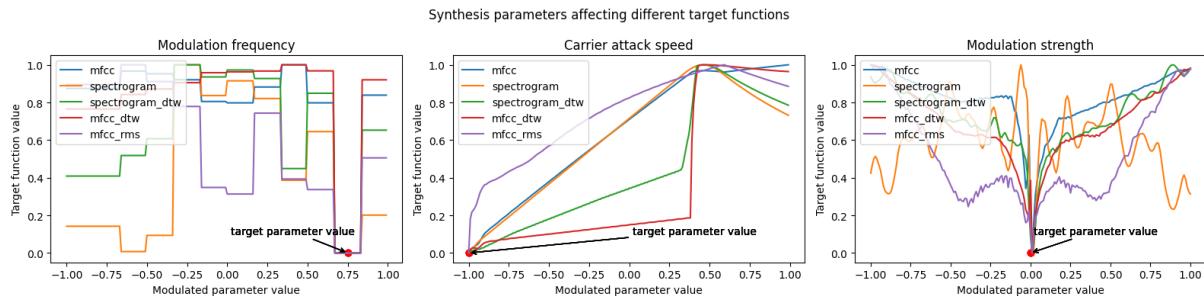
3.4. Przekrój wartości funkcji celu dla prostego problemu syntezy typu FM

Testy obejmowały wygenerowanie wartości funkcji celu podczas modyfikowania pojedynczego parametru w grafie przetwarzania sygnałów przedstawionym na rysunku 3.2. Modyfikowane parametry odpowiadają za różne cechy barwy uzyskanego dźwięku:

- *HarmonicMultiplier/input_scale*: częstotliwość modulacji FM,
- *SineOscillator/input_modulation_index*: siła składowych harmonicznych,
- *ADSR/input_attack*: dynamika dźwięku.



Rys. 3.2: Prosty graf syntezy FM, zawierający jeden oscylator służący za sygnał nośny i jeden oscylator służący za sygnał modulujący.



Rys. 3.3: Zmiany w wartościach testowanych funkcji celu podczas przesuwania różnych parametrów syntezy dźwięku. Kształt pierwszego wykresu wynika z zastosowania kwantyzacji dostępnych częstotliwości modulacji, aby wykluczyć nieharmoniczne stosunki częstotliwości modulacji i nośnej. Tego rodzaju praktyka jest wykorzystywana w syntezatorach FM [16], ponieważ ułatwia dostosowywanie parametrów syntezy.

3.4.1. Analiza wyników

Wyniki testów zaprezentowane na wykresach 3.4 pozwalają na wyeliminowanie różnic między spektrogramami jako funkcji celu, ponieważ w przypadku zmian częstotliwości modulacji posiada ona minimum globalne w niewłaściwej pozycji parametru. Późniejsze testy wykorzystują tylko funkcje celu wykorzystujące MFCC.

Częstotliwość modulacji FM

Wszystkie funkcje z wyjątkiem różnic między spektrogramami pokazują poprawną, najniższą wartość dla właściwej wartości parametru.

Dynamika dźwięku

W przypadku wpływu zmian w dynamice dźwięku na wartości funkcji celu, zastosowanie DTW znacząco zmienia kształt funkcji celu, zależnie od wybranego rozmiaru okna DTW. Duży rozmiar okna powoduje zmniejszenie kary za niedokładne odwzorowanie dynamiki dźwięku.

Sila modulacji

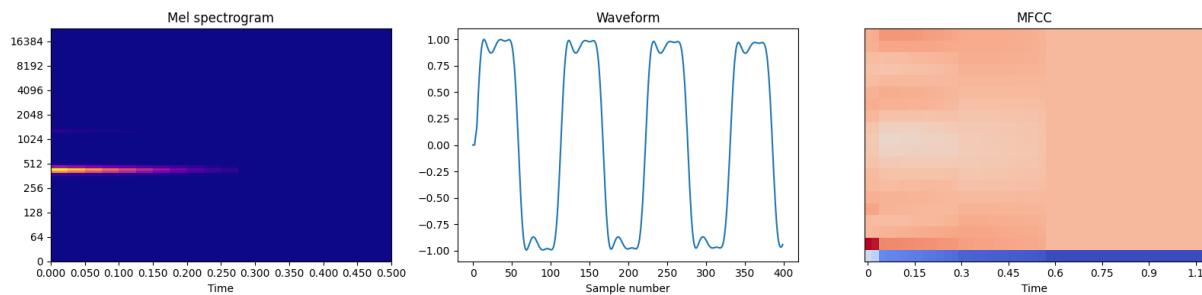
Różnica między spektrogramami jest najbardziej chaotyczna, nie maleje wraz ze zbliżaniem się do poprawnej wartości parametru. Pozostałe funkcje celu wykorzystujące MFCC mają lepszą charakterystykę – maleją wraz ze zbliżaniem się do poprawnej wartości parametru.

3.5. Optymalizacja parametrów dla predefiniowanych grafów syntezy FM oraz analog modeling

Drugą częścią procesu testowania funkcji celu z literatury było zweryfikowanie skuteczności każdej z funkcji w uproszczonym problemie optymalizacyjnym, polegającym jedynie na odnalezieniu właściwych parametrów dla predefiniowanej struktury grafu DSP. Wykorzystano dwa grafy DSP (rysunki 3.5 i 3.2), wykonujące różne rodzaje syntezy.

3.5.1. Synteza FM

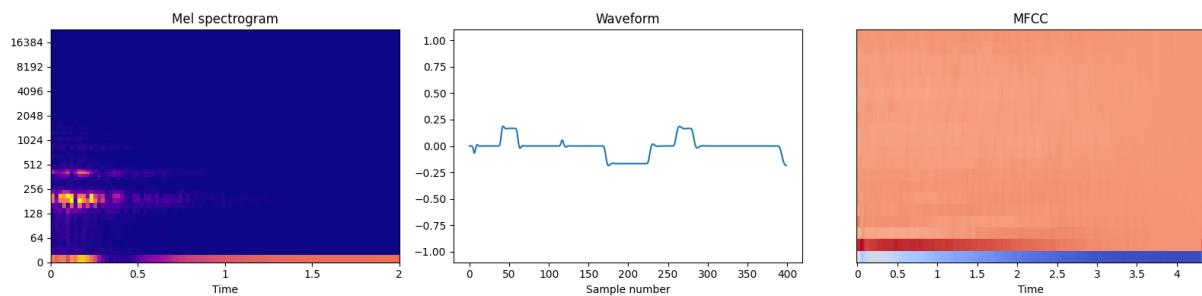
Testowana struktura grafu wykonującego syntezę typu FM 3.2 wykorzystuje 2 operatory: jedną nośną i jeden modulator. Parametry grafu zostały ręcznie dostrojone aby wygenerować krótki dźwięk typu *pluck*, w którym modulator przekształca nośną sinusoidę w sygnał zbliżony do sygnału prostokątnego. Z perspektywy wynikowego spektrum sygnału, przedstawionego na rysunku 3.8 sygnał składa się z częstotliwości podstawowej i jednej składowej harmonicznej. Wizualizacja sygnału została przedstawiona na wykresie 3.4.



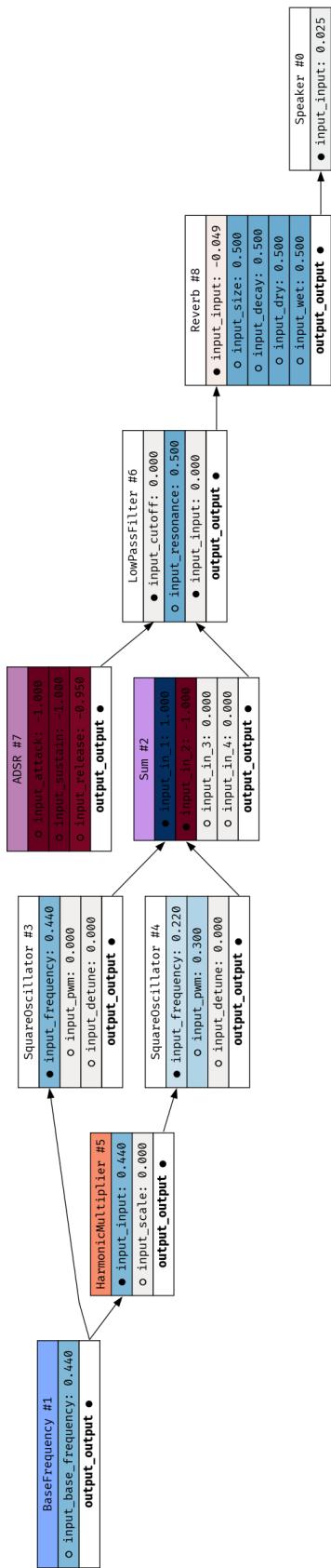
Rys. 3.4: Spektrogram, kształt fali oraz wizualizacja MFCC dla próbki dźwięku, którą ma imitować graf syntezы FM podczas testów różnych funkcji celu.

3.5.2. Synteza *analog modeling*

Synteza *analog modeling* zazwyczaj wykorzystuje mniej parametrów niż syntezą FM (3.2), aby zwiększyć trudność problemu optymalizacyjnego graf został rozszerzony o węzeł dodający efekt pogłosu [15]. Struktura grafu (przedstawiona na rysunku 3.5) składa się z dwóch oscylatorów generujących sygnał prostokątny. Oscylator Square0scillator #4 generuje sygnał przesunięty o oktawę w dół w stosunku do częstotliwości podstawowej, jednocześnie jego parametr `input_pwm` skraca szerokość generowanego impulsu, aby wzbogacić barwę dźwięku o dodatkowe składowe harmoniczne. Barwa dźwięku zmienia się dynamicznie w czasie dzięki zastosowaniu filtra niskoprzepustowego (LowPassFilter #6), którego częstotliwość odcięcia jest modulowana przez sygnał sterujący ADSR #7. Długość dźwięku generowanego przez oscylatory jest podobna jak w przypadku grafu FM (3.2), zastosowanie węzła Reverb #8 przedłuża czas trwania dźwięku i dodatkowo „rozmywa go” w czasie, co pokazuje spektrum sygnału na wykresie 3.6.



Rys. 3.6: Spektrogram, kształt fali oraz wizualizacja MFCC dla próbki dźwięku, którą ma imitować graf syntezы *analog_modeling* podczas testów różnych funkcji celu.



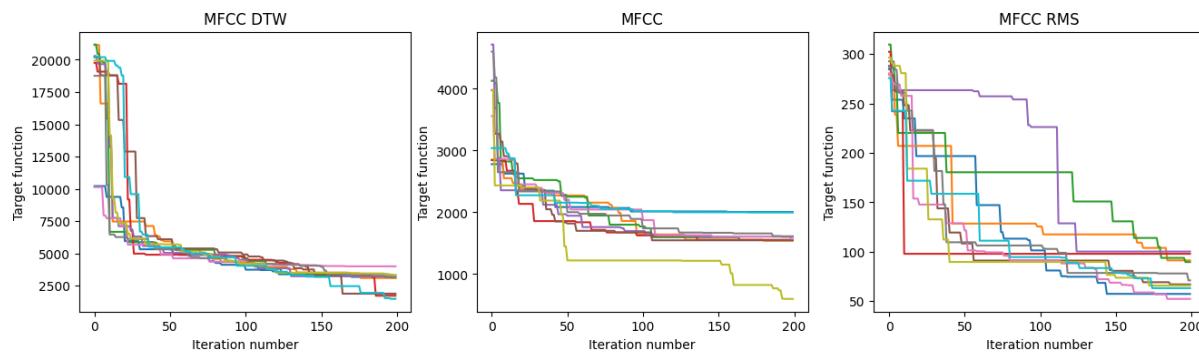
Rys. 3.5: Graf wykonujący syntezę typu *analog modeling*, wykorzystany do testów funkcji celu.

3.5.3. Plan testów

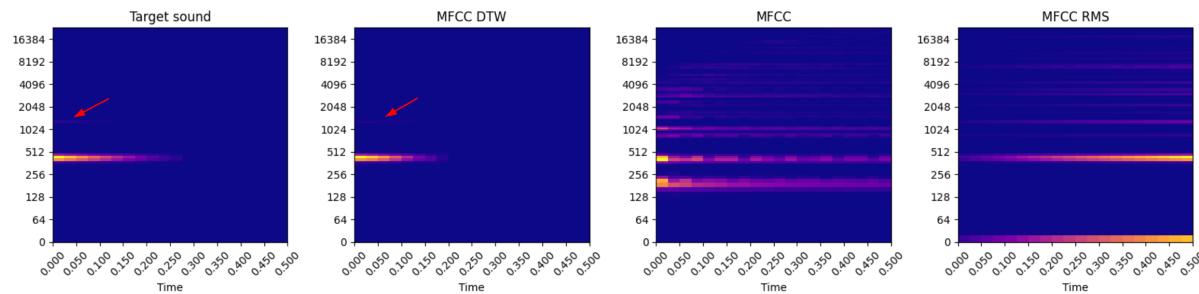
Dla obu grafów wykonano optymalizację parametrów wejściowych w celu imitacji danej próbki dźwięku. Optymalizację wykonano 10 razy dla każdej rozważanej (3.1.2) funkcji celu. Do optymalizacji parametrów wykorzystano algorytm evolucyjny *differential evolution* [42]. Zastosowano następujące parametry algorytmu genetycznego:

1. liczba iteracji: 200,
2. rozmiar populacji: 40,
3. strategia mutacji: best1bin,
4. rekombinacja: 0.7.

3.5.4. Wyniki testów



Rys. 3.7: Wykresy zmian funkcji celu podczas optymalizacji dla grafu syntezy FM.

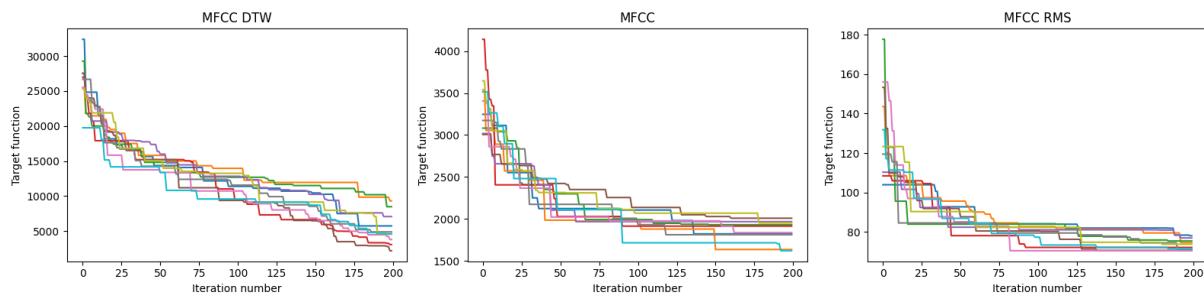


Rys. 3.8: Spektrogram dźwięku docelowego oraz dźwięków uzyskanych w procesie optymalizacji parametrów grafu FM. Czerwoną strzałką oznaczono składową harmoniczną (słabo widoczną na spektrogramie), która została poprawnie odtworzona przez algorytm optymalizacji.

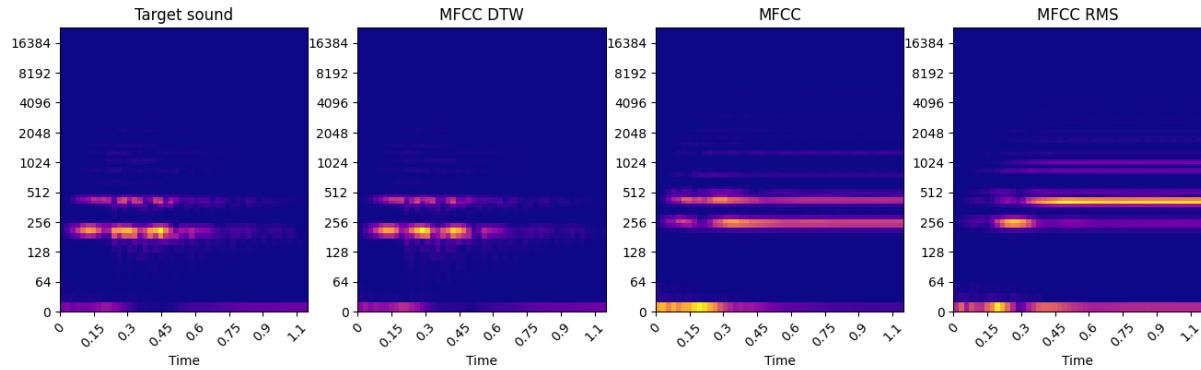
Syntez FM

Algorytm optymalizacji jest w stanie poprawnie dostosować wartości parametrów grafu w przypadku zastosowania MFCC+DTW jako funkcji celu. Jak pokazuje spektrogram (3.8), poprawnie odtworzona jest zarówno dynamika dźwięku i składowa harmoniczna. Pozostałe funkcje celu nie pozwalają na odtworzenie sygnału, który w bliski sposób przypomina dźwięk docelowy, pomimo uzyskiwania wyników zbliżonych do MFCC+DTW we wcześniejszych testach (3.4).

Syntez analog modeling



Rys. 3.9: Wykresy zmian funkcji celu podczas optymalizacji dla grafu syntezy *analog modeling*.



Rys. 3.10: Spektrogram dźwięku docelowego oraz dźwięków uzyskanych w procesie optymalizacji parametrów grafu *analog modeling*.

Podobnie jak podczas testów syntezy FM, funkcja celu MFCC+DTW pozwala na dokładne odwzorowanie barwy dźwięku. Na spektrogramie 3.10 widoczne są poprawnie odtworzone składowe harmoniczne, przebieg dynamiczny dźwięku oraz parametry efektu pogłosu. Funkcja celu wykorzystująca tylko współczynniki MFCC nie jest w stanie odtworzyć przebiegu dynamicznego dźwięku i jedynie częściowo odtwarza poprawne składowe harmoniczne. MFCC+RMS nie jest w stanie odtworzyć żadnej z cech docelowego dźwięku.

3.5.5. Wybór funkcji celu na podstawie wyników

Wyniki testów pozwalają jednoznacznie wybrać funkcję celu, która porównuje wartości MFCC sygnałów za pomocą algorytmu *dynamic time warping*. W zakresie pracy nie leży szczegółowe wyjaśnienie, czemu zastosowanie DFT usprawnia proces optymalizacji. Możliwym intuicyjnym wyjaśnieniem tego fenomenu jest fakt, że DFT pozwala na rozpoznanie poszczególnych fonemów w nagraniach mowy [41], niezależnie od prędkości wypowiadania słów. Analogicznie, w przypadku porównywania sygnałów dźwiękowych generowanych przez grafy przetwarzania sygnałów, wykorzystanie DFT może powodować „wygładzenie” niedokładności w zmianach tembru (transientach [4]) i dynamiki dźwięku, które występują pomiędzy sygnałem docelowym i wygenerowanym.

Rozdział 4

Algorytm rozwiązańia

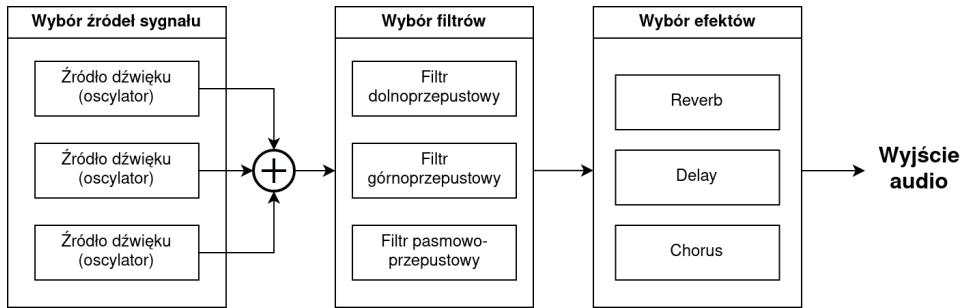
Jak opisano w rozdziale poświęconym definicji problemu (2), praca rozwiązuje problem budowy grafu DSP z wykorzystaniem dwóch algorytmów, których współpracę przedstawiono na rysunku 4.1.

1. Algorytm generujący graf DSP, opisany w rozdziale 5,
2. Algorytm oceniający jak bardzo wygenerowany dźwięk jest bliski dźwiękowi docelowemu pod względem barwy, opisany w rozdziale 3.



Rys. 4.1: Diagram algorytmu rozwiązańia zaimplementowanego w ramach pracy. Algorytm oceny może wykorzystywać różne funkcje celu, finalnie zastosowano MFCC oraz *dynamic time wrapping*, proces wyboru funkcji celu opisuje rozdział 3.

Praca wykorzystuje algorytm genetyczny [42], w którym genotyp odpowiada za strukturę grafu 4.3 oraz wartości przypisane do parametrów grafu (2.1, 2.2). Algorytm generuje różne warianty grafów przetwarzania sygnałów powszechnie wykorzystywane w syntezatorach dźwięku [31] [16] [6]. Takie podejście pozwala na dostosowanie grafu do danego rodzaju syntez, lub połączenie wielu typów syntez w celu uzyskania bardziej złożonego brzmienia. Algorytm wybiera węzły dla każdej sekcji zilustrowanej na rysunku 4.2.



Rys. 4.2: Sekcje przetwarzania sygnałów oraz przykładowe węzły przetwarzania sygnałów, które są w nich powszechnie wykorzystywane.

Przykładowo, w syntezatorach wykorzystujących syntezę typu FM [6], źródłem sygnału będą operatory wykorzystujące proste sygnały sinusoidalne, poddane modulacji częstotliwości. Dla syntezatorów analogowych (subtraktywnych), zamiast prostych sygnałów wykorzystywane są oscylatory generujące fale kwadratowe i piłokształtne, o dużej liczbie składowych harmonicznych. Uzasadnia to wykorzystanie filtrów dolnoprzepustowych, które z kolei nie występują w tradycyjnych syntezatorach FM (pojawiają się dopiero we współczesnych modelach [16]). Instrumenty eksperymentalne, wykorzystujące mniej popularne rodzaje syntezy, takie jak *physical modeling* [7], często opierają się jedynie na rozbudowanej sekcji oscylatorów, które posiadają wystarczająco dużo parametrów by wynagrodzić tym brak sekcji subtraktywnej. Po drugiej stronie spektrum znajdują się instrumenty wykorzystujące głównie sekcję efektów i syntezę granularną [3], aby w nieoczekiwany sposób modyfikować proste próbki dźwięku.

4.1. Wybór źródeł sygnału

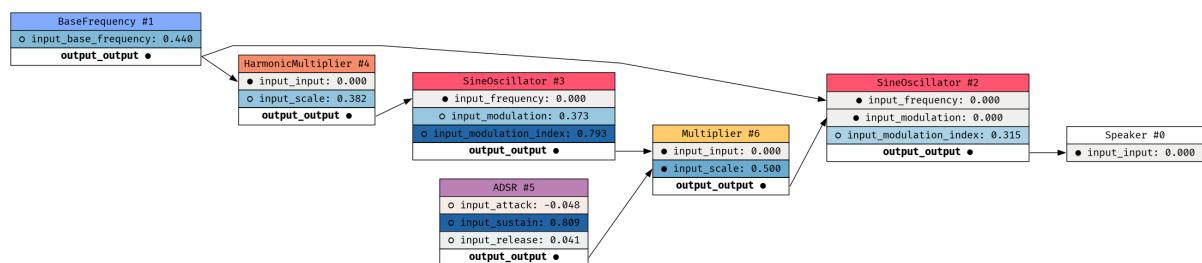
Graf posiada 3 sloty dla węzłów generujących sygnał. W pracy zaimplementowano różne typy syntezy, które mogą być wykorzystane przez algorytm do syntezy dźwięków o różnorodnych barwach.

4.1.1. Syntezator FM

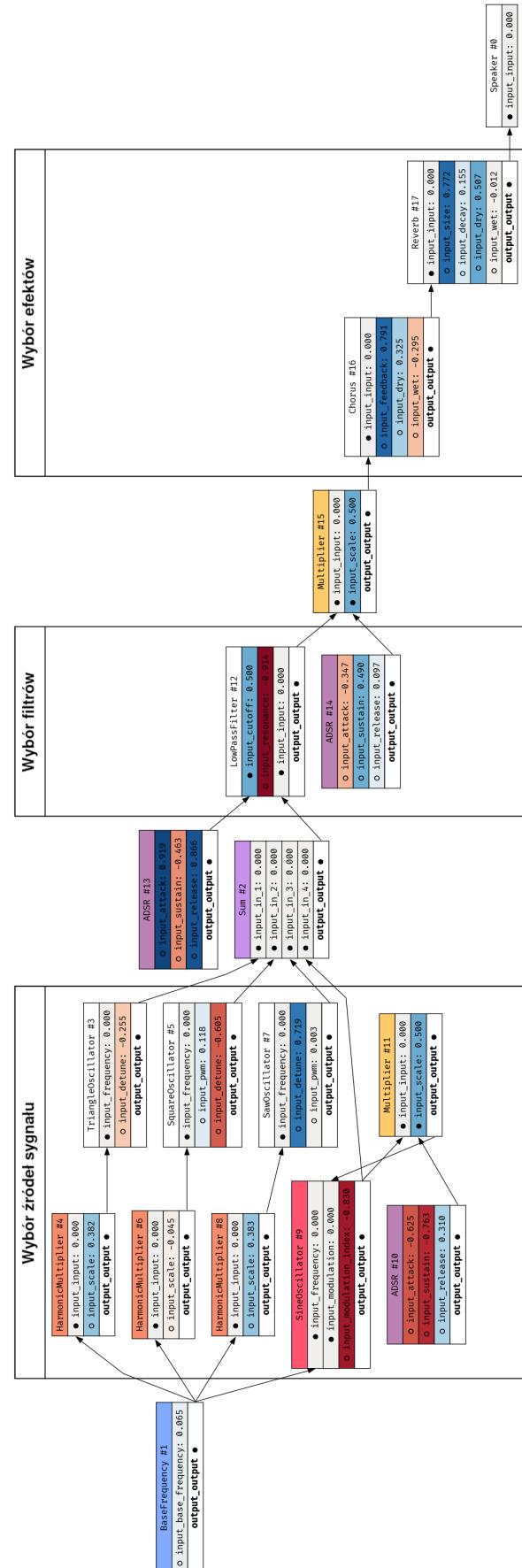
Zaimplementowane w pracy geny odpowiedzialne za syntezy FM odwzorowują uproszczone algorytmy syntezy wykorzystane w syntezatorze Yamaha DX7 [6].

Gen FM1

Gen FM1



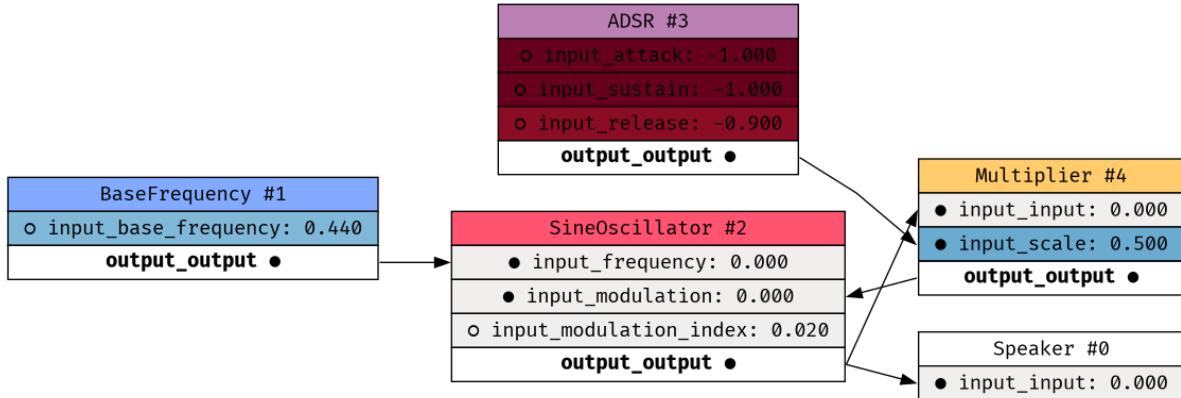
Rys. 4.4: Graf wykorzystujący gen FM1.



Rys. 4.3: Przykład wygenerowanej struktury grafu, oznaczono segmenty z diagramu 4.2.

Gen FM1, przedstawiony na diagramie 4.5, typową dla syntezy FM modulację fali sinusoidalnej za pomocą innej fali sinusoidalnej. Taki układ umożliwia uzyskanie dźwięków przypominających dźwięki fletu, trąbki lub dzwonków.

Gen FM2



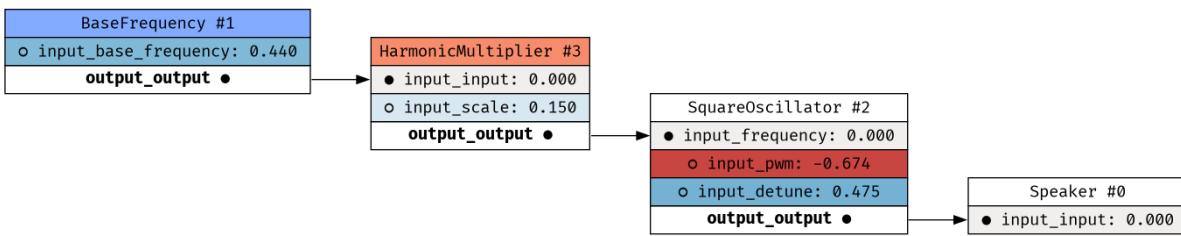
Rys. 4.5: Graf wykorzystujący gen FM2.

Gen FM2, zilustrowany na diagramie 4.5, wykorzystuje pojedynczy operator ze sprzężeniem zwrotnym. W zależności od ustawionych parametrów, taka struktura pozwala na uzyskanie dźwięków przypominających uderzenie w strunę lub służyć jako źródło szumu.

4.1.2. Synteza *analog modeling*

Zaimplementowane w pracy geny odpowiedzialne za syntezy FM odwzorowują uproszczone algorytmy syntezy wykorzystane w syntezatorze *Korg Minilogue xd* [6].

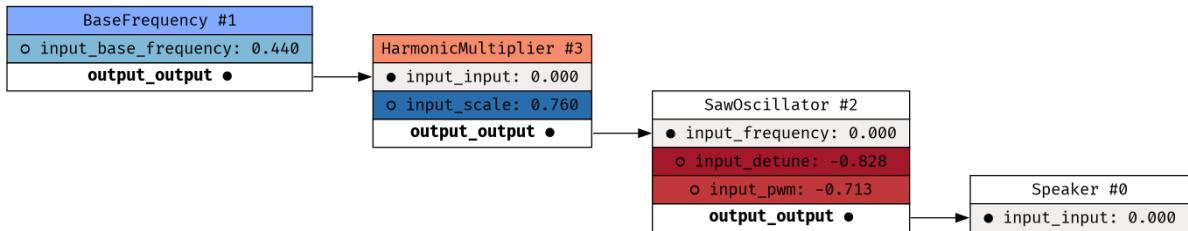
Gen AN1



Rys. 4.6: Graf wykorzystujący gen AN1.

Gen AN1 (diagram 4.6) pozwala na uzyskanie fal prostokątnych o różnej szerokości impulsów.

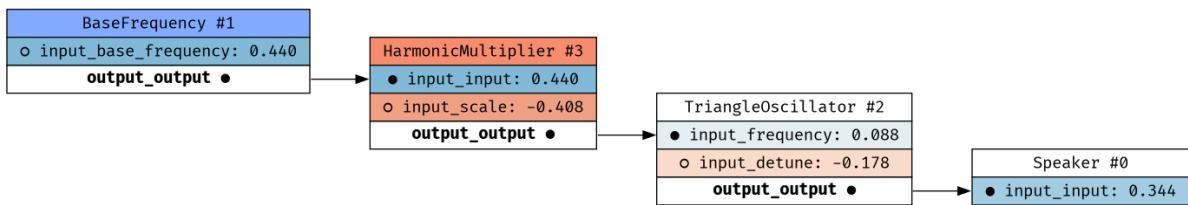
Gen AN2



Rys. 4.7: Graf wykorzystujący gen AN2.

Gen AN2, przedstawiony na diagramie 4.7, pozwala na uzyskanie sygnałów piłokształtnych.

Gen AN3



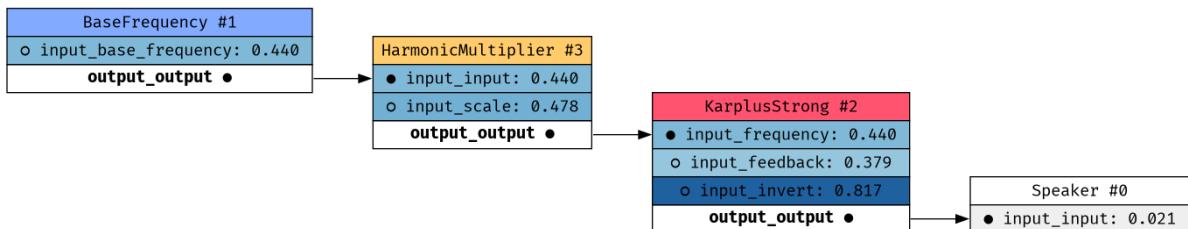
Rys. 4.8: Graf wykorzystujący gen AN3.

Gen AN3, (diagram 4.9), generuje sygnały trójkątne.

4.1.3. Synteza *physical modeling*

Algorytmy wykonujące syntezę typu *physical modeling* pochodzą z [32]. W pracy zaimplementowano podstawowy wariant algorytmu *Karplus-Strong*.

Gen PM1



Rys. 4.9: Graf wykorzystujący gen PM1.

Gen PM1 generuje sygnał przypominający strunę gitarową za pomocą algorytmu *Karplus-Strong*.

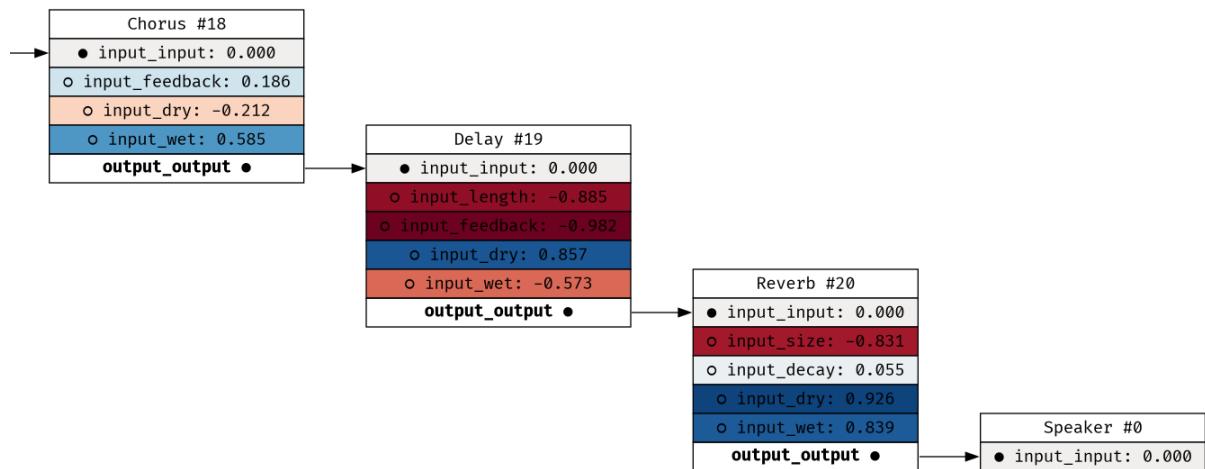
4.2. Wybór filtrów

W pracy wykorzystano gotową implementację filtra cyfrowego, emulującego rezonansowy filtr drabinkowy [2]. Implementacja została wykorzystana do tworzenia filtrów dolno oraz górnoprzepustowego. Węzeł filtru posiada 3 wejścia:

1. częstotliwość odcięcia – kontroluje zakres częstotliwości, które nie są tłumione przez filtr,
2. intensywność rezonansu – kontroluje intensywność wzmocnienia częstotliwości bliskich częstotliwości odcięcia filtru,
3. sygnał poddawany filtracji.

4.3. Wybór efektów

Efekty połączone sa w łańcuchach *chorus* → *delay* → *reverb*. Algorytm generacji wybiera, które efekty będą obecne w grafie. Diagram 4.10 ilustruje fragment grafu, w którego strukturze znajdują się wszystkie 3 efekty.



Rys. 4.10: Przykładowy łańcuch efektów w grafie.

Pogłos

W pracy wykorzystano gotowy algorytm pogłosu (*reverb*) oparty na [15]. Węzeł **reverb** posiada następujące parametry:

1. **input_size** – kontroluje rozmiar symulowanej przestrzeni, małe wartości generują pogłos typowy dla małych pokojów lub kameralnych sal koncertowych, duże wartości symulują pogłos dużych hal,
2. **input_decay** – kontroluje prędkość spadku poziomu głośności wygenerowanego pogłosu,
3. **input_dry**, **input_wet** – określa balans głośności między surowym sygnałem wejściowym a wygenerowanym pogłosem.

Echo

Algorytm generujący echo (*delay*) został zaimplementowany jako bufor kołowy. Występują w nim parametry:

1. **input_length** – długość bufora kołowego, duże wartości spowodują że echo będzie słyszalne po dłuższym czasie,
2. **input_feedback** – kontroluje prędkość spadku głośności echo.
3. **input_dry**, **input_wet** – tak samo jak w węźle **reverb**.

Chorus

Algorytm generujący efekt *chorus* został zaimplementowany jako wariant algorytmu *delay*, w którym długość bufora jest modulowana przez falę sinusoidalną. Węzeł *chorus* posiada następujące parametry:

1. **input_feedback** – tak samo jak w węźle **delay**.
2. **input_dry**, **input_wet** – tak samo jak w węźle **reverb**.

Rozdział 5

Implementacja grafu przetwarzania sygnałów

Na potrzeby badań zostało zaimplementowane środowisko, pozwalające na dynamiczne tworzenie grafów przetwarzania sygnałów (5.1). W projekcie nie zostało zastosowane gotowe rozwiązanie symulujące syntezator modułowy, takie jak Bespoke Synth [13], VCVRack [5] lub Pure Data [9], ponieważ nie udostępniały one gotowego interfejsu pozwalającego na łatwą integrację z językiem Python. Istniejące w internecie gotowe przykłady algorytmów syntezy audio pozwoliły na szybkie zaimplementowanie środowiska eksperymentowego, które posiada szeroki zbiór dostępnych algorytmów DSP oraz w przystępny sposób interfejsuje się z językiem Python, co umożliwia wykorzystanie gotowych pakietów obliczeniowych z dziedziny przetwarzania sygnałów.



Rys. 5.1: Przykładowy układ modułów w standardzie *Eurorack* [1]. W prawym dolnym rogu widoczne połączenia modulujące między modułami.

5.1. Podstawy syntezy dźwięku w syntezatorach modułowych

Proces syntezy dźwięku może zostać przedstawiony jako zbiór węzłów wykonujących syntezę lub przetwarzanie sygnału audio oraz połączeń między węzłami. Przykładowe elementy grafu:

1. Węzły:

1. generujące sygnał:
 - synteza sygnałów (sinusoida, trójkąt, sygnał prostokątny),
 - sygnał modulujący (LFO, ADSR).
2. przetwarzające sygnał:
 - filtry (górnoprzepustowy, dolnoprzepustowy, pasmowo-przepustowy),
 - efekty (pogłos, echo).
2. Połączenia między węzłami:
 - modulowanie parametrów syntezy i przetwarzania sygnału.

Odpowiednikiem implementowanego środowiska w świecie rzeczywistym są syntezatory modułowe (przykładowo 5), które pozwalają na dowolne łączenie modułów wykonujących operacje DSP. Barwę dźwięku w syntezatorze modyfikuje się na dwa sposoby:

1. Ustawienie stałej wartości danego parametru w węźle DSP,
2. Modulacja wartości danego parametru w węźle DSP za pomocą wartości wyjściowej innego węzła.

Dla przykładowego układu DSP, przedstawionego na rysunku 5.1, skonfigurowane są między innymi parametry:

1. Częstotliwość podstawowa (węzeł `BaseFrequency #1`),
2. wartość, przez którą mnożona jest częstotliwość podstawowa w węźle `HarmonicMultiplier #5`,
3. Wartości `input_pwm` w węzłach `SquareOscillator #3` oraz `#4`,
4. Parametry algorytmu pogłosu w węźle `Reverb #8`.

Z kolei wartość parametru `input_cutoff` w węźle `LowPassFilter #6` jest modulowana przez sygnał wychodzący w węźle `ADSR #7`, co pozwala na dynamiczne zmiany częstotliwości odcięcia filtru w czasie, wzbogacając barwę generowanego dźwięku.

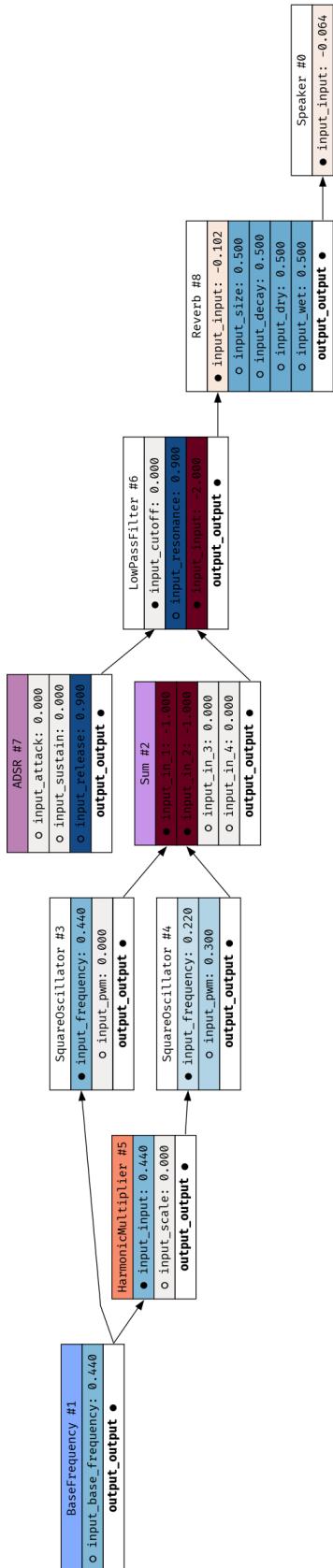
5.2. Wymagania

W ramach pracy zostały zdefiniowane wymagania dotyczące implementowanego później środowiska eksperymentowego, opisane w niniejszym rozdziale.

5.2.1. Węzły DSP

Pojedynczy węzeł DSP może zostać opisany za pomocą trzech cech:

1. Zbiór sygnałów wejściowych,
2. zbiór sygnałów wyjściowych,
3. wykonywana przez węzeł operacja.



Rys. 5.2: Przykładowy układ węzłów DSP w zaimplementowanym środowisku eksperymentalnym. Układ wykonuje syntezę subtraktywną z modulowaną wartością częstotliwości granicznej filtra nisko-przepustowego oraz dodaje efekt pogłosu (*reverb*) [15].

SineOscillator #2
o input_frequency: 0.000
o input_modulation: 0.000
o input_modulation_index: 0.100
output_output •

Rys. 5.3: Węzeł DSP w zaimplementowanym środowisku eksperymentowym, generujący falę sinusoidalną z możliwością modulacji fazy.



Rys. 5.4: Moduł syntezy *Mutable Instruments Elements*, umożliwiający ręczne ustawianie parametrów oraz ich modulację za pomocą sygnału CV. Moduł wykonuje syntezę typu *physical modeling* [23].

Przykładowo, przedstawiony na rysunku 5.2.1 węzeł posiada:

1. sygnały wejściowe:
 - `input_frequency` - częstotliwość generowanej sinusoidy,
 - `input_modulation` - wartość modulacji fazy, według równania 5.1,
 - `input_modulation_index`.
2. Sygnały wyjściowe:
 - `output_output` - wartość generowanego sygnału sinusoidalnego.

Węzeł generuje sygnał sinusoidalny o fazie modulowanej poprzez parametr `input_modulation` z siłą modulacji ustawianą przez parametr `input_modulation_index`, opisane za pomocą równania 5.1 (jest to uproszczenie równania syntezy FM przedstawionego w [40]) oraz listingu 5.1:

$$f(t) = \sin(t * f + m * m_i) \quad (5.1)$$

Listing 5.1: Implementacja węzła SineOscillator.

```
impl DspNode for SineOscillator {
    fn tick(&mut self) {
        let frequency = (self.input_frequency * 1000.0).abs();
        let phase_diff = (2.0 * std::f64::consts::PI * frequency) /
            ↪ SAMPLE_RATE;
        self.output_output =
            (self.phase + self.input_modulation * self.
            ↪ input_modulation_index * 10.0).sin();
        self.phase += phase_diff;
    }
}
```

```

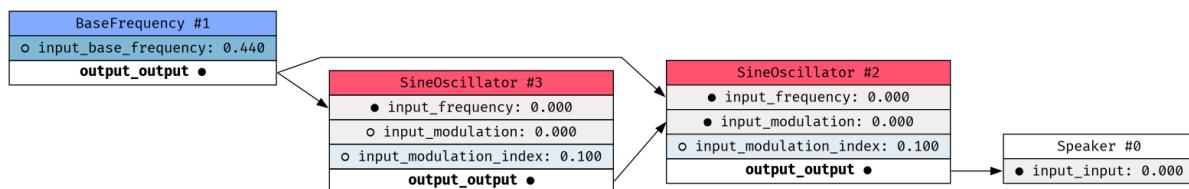
        while self.phase > std::f64::consts::PI * 2.0 {
            self.phase -= std::f64::consts::PI * 2.0
        }
    }
}

```

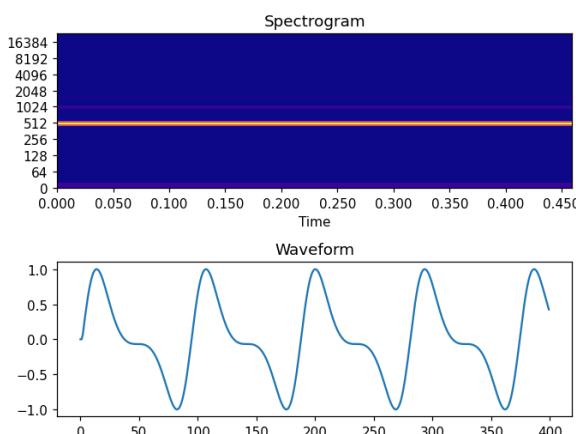
Wymaganie: zaimplementowane w ramach pracy środowisko eksperymentalne musi pozwalać na zdefiniowanie węzłów DSP, które generują lub przetwarzają sygnał. Węzły posiadają sloty wejściowe, z których czytają wartości parametrów sterujących wykonywanymi przez węzły operacjami.

5.2.2. Połączenia między węzłami – modulacja parametrów węzłów

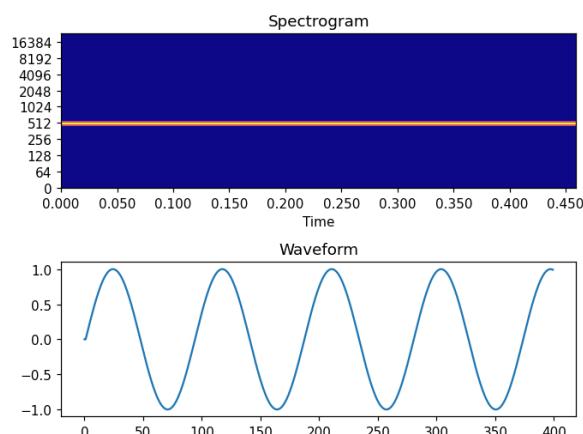
Każdy węzeł DSP w środowisku eksperymentalnym posiada zbiór parametrów wejściowych. Poza możliwością ustawienia danego parametru wejściowego na konkretną wartość, możliwa jest też modulacja parametru wejściowego. Na rysunku 5.2.2 przedstawiony jest przykładowy układ węzłów i modulacji, które pozwalają na uzyskanie syntezy FM.



Rys. 5.5: Przykładowa modulacja parametru `input_modulation` za pomocą sygnału sinusoidalnego, charakterystyczna dla syntezy typu FM [32].



Rys. 5.6: Spektrogram oraz wykres sygnału wygenerowanego za pomocą układ z rysunku 5.2.2. Widoczne dodatkowe składowe harmoniczne wpływające na barwę dźwięku.



Rys. 5.7: Spektrogram oraz wykres sygnału wygenerowanego przez układ z rysunku 5.2.2 po usunięciu połączenia modulującego fazę oscylatora #2. Widoczna tylko jedna składowa harmoniczna: częstotliwość podstawowa.

Wymaganie: zaimplementowane środowisko pozwala na modulowanie dowolnego parametru wejściowego w węźle za pomocą wartości wyjściowej dowolnego węzła, **w tym modulowanie wejścia węzła wyjściem tego samego węzła** (tzw. *circular patching*, popularny zarówno w syntezie FM jak i w układach analogowych).

5.2.3. Graf przetwarzania sygnałów

Węzły DSP oraz połączenia między nimi istnieją w ramach danego grafu przetwarzania sygnałów, który agreguje wiele węzłów i wiele połączeń. Instancja grafu DSP musi umożliwiać dynamiczną modyfikację grafu, na którą składają się następujące operacje:

1. Dodanie nowego węzła,
2. Dodanie nowego połączenia między węzłami,
3. Usunięcie węzła,
4. Usunięcie połączenia między węzłami,
5. Ustawienie i -tego parametru wejściowego danego węzła na określoną przez użytkownika wartość.

Po utworzeniu grafu, użytkownik musi mieć możliwość „uruchomienia” na grafie procesu syntezы dźwięku, który zwróci użytkownikowi strukturę danych zawierającą wygenerowany sygnał.

Wymaganie: zaimplementowane środowisko pozwala na dynamiczną modyfikację grafu przetwarzania sygnałów oraz na wygenerowanie sygnału z wytworzzonego w środowisku grafu.

5.2.4. Automatyzacja pracy ze środowiskiem eksperymentowym za pośrednictwem języka Python

Wymaganie: ze względu na dużą dostępność gotowych algorytmów optymalizacyjnych oraz DSP w języku Python ([36], [42]), zaimplementowane środowisko musi udostępniać interfejs pozwalający na wykonywanie operacji zdefiniowanych w wymaganiach za pośrednictwem języka Python.

5.3. Opis zaimplementowanego środowiska eksperymentowego

W ramach pracy zaimplementowane zostało środowisko pozwalające na dynamiczne budowanie grafów DSP oraz na generowanie sygnałów dźwiękowych za pomocą wytworzonych grafów, według wymagań opisanych w sekcji 5.2. Środowisko zaimplementowano w języku Rust, dzięki czemu proces syntezы sygnałów jest szybszy niż w przypadku implementacji w języku interpretowanym. Zaimplementowana biblioteka udostępnia interfejs zgodny ze standardem *Python Extension Module* [21].

5.3.1. Przykłady użycia

Utworzenie grafu

Zaimplementowane środowisko pozwala na tworzenie grafów przetwarzania sygnałów za pomocą poleceń w języku Python. Listing 5.2 przedstawia proces tworzenia prostego grafu generującego sygnał sinusoidalny.

Listing 5.2: Utworzenie prostego grafu generującego sygnał sinusoidalny.

```
g = DspGraph()

carrier = g.add_sine(SineOscillator())
g.patch(
    g.base_frequency_node_id, "output_output",
```

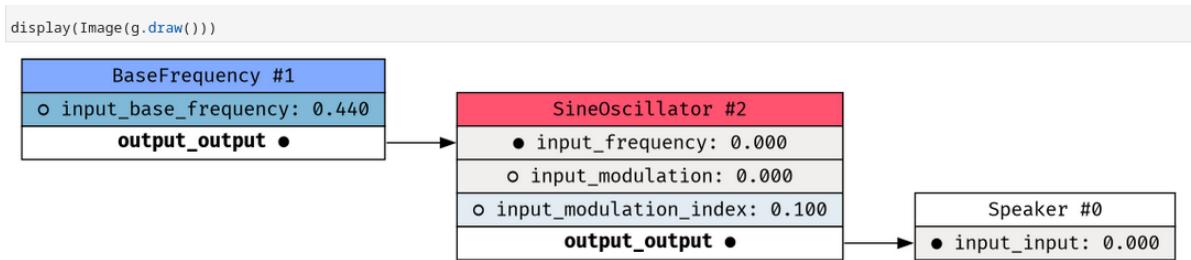
```

        carrier, "input_frequency"
    )

g.patch(
    carrier, "output_output",
    g.speaker_node_id, "input_input"
)

display(Image(g.draw()))

```



Rys. 5.8: Wynik wykonania kodu przedstawionego w listingu 5.2 w środowisku *Jupyter Notebook*, wizualizacja utworzonego grafu.

Uruchomienie procesu syntezy dźwięku

Jak pokazano na listingu 5.3, środowisko eksperymentalne zaimplementowane w ramach pracy w języku Rust zwraca obiekty typu `ndarray`, wykorzystywane w większości pakietów obliczeniowych wykorzystywanych w języku Python. Umożliwia to wykorzystanie gotowych bibliotek dostępnych w języku Python, aby przeanalizować sygnał lub zoptymalizować parametry syntezy [42] [36].

Listing 5.3: Typ danych zwracanych przez środowisko eksperymentalne.

```

>>> generated_signal = g.play(num_samples=100)
>>> type(generated_signal)
<class 'numpy.ndarray'>

```

5.3.2. Detale techniczne

Połączenia między węzłami w grafie

Ponieważ wymagania zdefiniowane w rozdziale 5.2 zawierają dynamiczne modyfikowanie grafu przetwarzania sygnałów, nie jest możliwe sprefiniowanie mechanizmu wymiany danych między połączonymi węzłami. Podczas implementacji rozważane były następujące architektury:

1. Przejście przez graf przed uruchomieniem syntezy i określenie kolejności wywołania węzłów,
2. Wykorzystanie struktury danych kolejki w każdym połączeniu,
3. Bezpośrednie przepisywanie wartości wyjść z węzłów do modulowanych przez nie wejść po każdej iteracji przetwarzania.

Ostatecznie wybrane zostało podejście 3, ze względu na konieczność spełnienia wymagań 5.2.2, konkretnie możliwości modulowania wejścia danego węzła przez wyjście tego samego węzła, co uniemożliwia wykorzystanie podejścia 1. Jednocześnie podejście 3 jest łatwiejsze do

implementacji niż podejście 2. Implementację mechanizmu przesyłu danych między węzłami ułatwiało wykorzystanie makr proceduralnych, opisane w sekcji 5.3.2.

Zastosowanie procedural macros do automatycznej generacji akcesorów struktur węzłów

Podczas implementacji grafu DSP zostały wykorzystane makra proceduralne [10], które umożliwiają automatyczne zaimplementowanie metod odczytujących i -te wejście lub wyjście danego węzła. Alternatywnym podejściem byłoby wykorzystanie struktur takich jak słowniki lub mapy, które umożliwiają na inspekcję kluczy w strukturze danych podczas działania programu i dostęp do nich za pomocą indeksu, jednakże takie podejście zmniejsza wydajność programu i nie wykorzystuje wykorzystywania systemu typów wbudowanego w język, co potencjalnie może być źródłem błędów podczas utrzymywania dużego zbioru węzłów i algorytmów, które wykonują. Wykorzystanie makr proceduralnych pozwala na ograniczenie powtarzalnych implementacji podobnych akcesorów i zachowanie zalet silnego systemu typów języka Rust.

Implementacja natywnego modułu dla języka Python

Aby umożliwić wykorzystanie grafu przetwarzania sygnałów z poziomu języka Python, wykorzystano narzędzie *Maturin* [8], służące do implementowania rozszerzeń zgodnych ze standardem *Python Extension Module* [21] w języku Rust.

Rozdział 6

Badania symulacyjne

Praca wykorzystuje próbki dźwięku z literatury [33] aby przetestować zaimplementowany algorytm i jednocześnie porównać jego wyniki z podobną pracą. Wykorzystano te próbki dźwięku z literatury, które **nie zostały wygenerowane** za pomocą oprogramowania do syntezy *Pure Data* [9], ponieważ oryginalna praca wykorzystywała to oprogramowanie jako środowisko do syntezy dźwięku – takie porównanie nie byłoby miarodajne, ponieważ algorytm z literatury miałby do dyspozycji dokładnie takie same algorytmy, które zostały wykorzystane do wygenerowania dźwięków docelowych. Przed przeprowadzeniem optymalizacji ustalono wartość częstotliwości podstawowej f_0 dla każdego z testowanych dźwięków. Zastosowano implementację algorytmu estymującego częstotliwość podstawową *YIN* [35], dostępną w pakiecie obliczeniowym *librosa* [36]. Przebieg procesu optymalizacji dla każdego z testowanych dźwięków umieszczony jest na płycie CD, co opisuje dodatek B.

Parametry algorytmu genetycznego

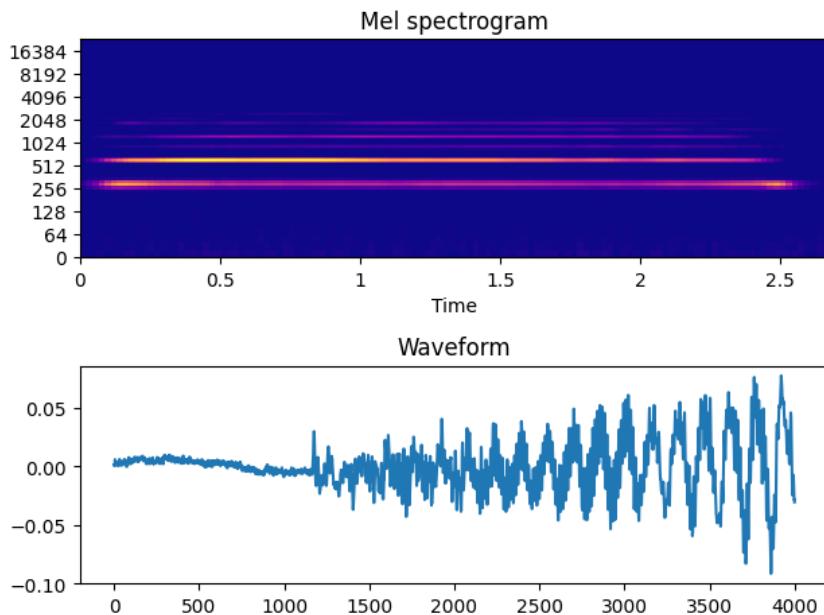
W badaniach zastosowano następujące parametry algorytmu genetycznego:

1. rozmiar populacji: 50,
2. liczba iteracji: 200,
3. strategia mutacji: `best1bin`,
4. rekombinacja: 0.7.

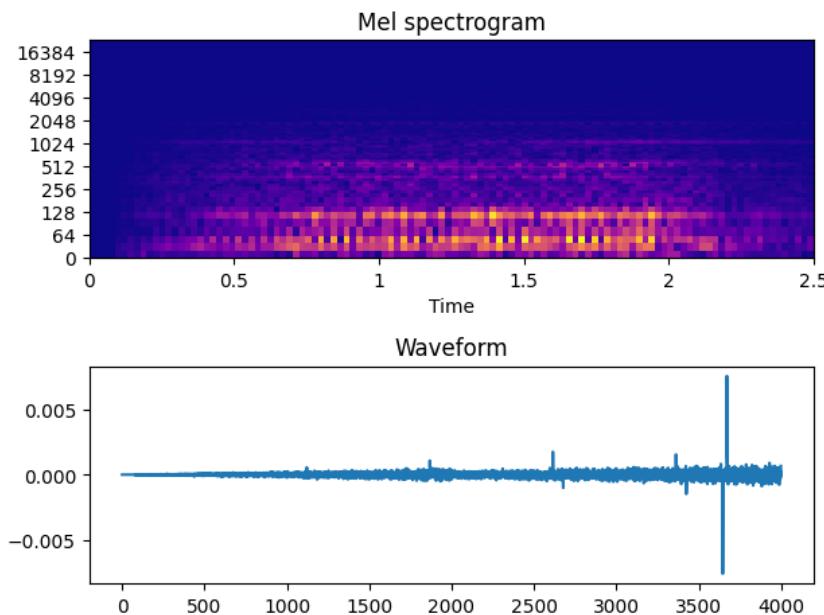
Ze względu na długi czas działania zaimplementowanego algorytmu optymalizacji (średnio 10 minut na pojedynczą iterację), w pracy nie udało się porównać jak różne parametry algorytmu genetycznego wpływają na proces optymalizacji. Finalnie wykonano tylko jedną pełną optymalizację dla każdego z testowanych dźwięków.

6.1. Dźwięk fletu

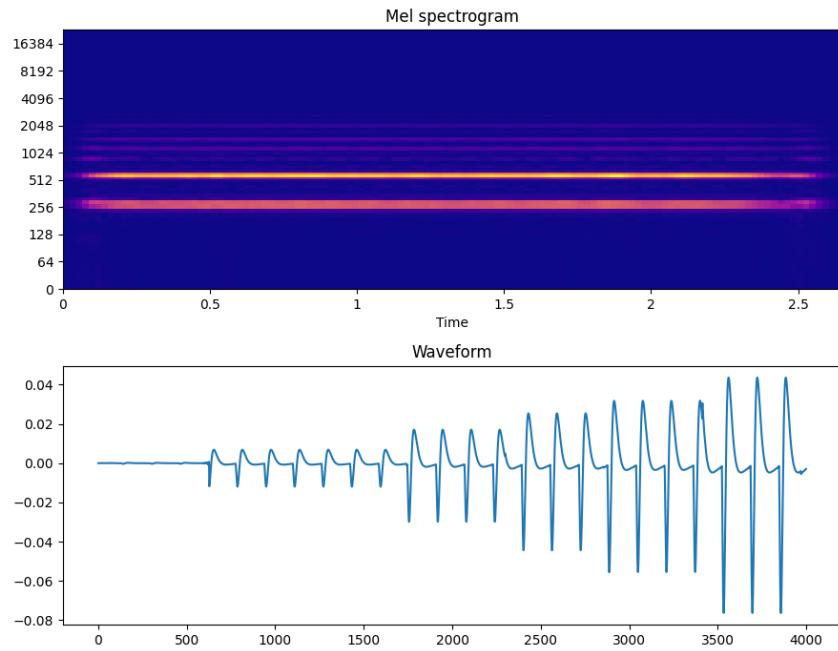
Dźwięk `flute.wav` (rysunek 6.1) jest nagraniem prawdziwego instrumentu dętego. Kształt fali jest nieregularny, charakterystyka spektralna zawiera dynamicznie pojawiające się i słabnące składowe częstotliwościowe.



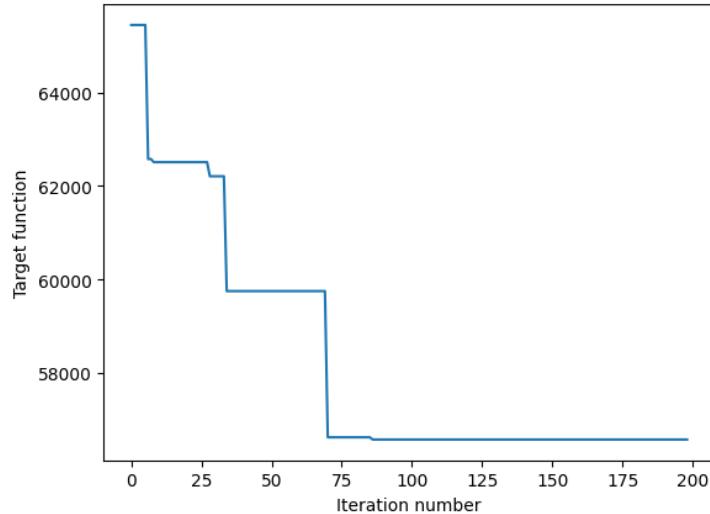
Rys. 6.1: Spektrogram i wykres fali dla dźwięku `flute.wav` wykorzystywanego do eksperymentów w [33].



Rys. 6.2: Spektrogram i wykres fali dla dźwięku `flute.wav` wytworzonyego przez zaimplementowany algorytm optymalizacji dla dźwięku `flute.wav`.



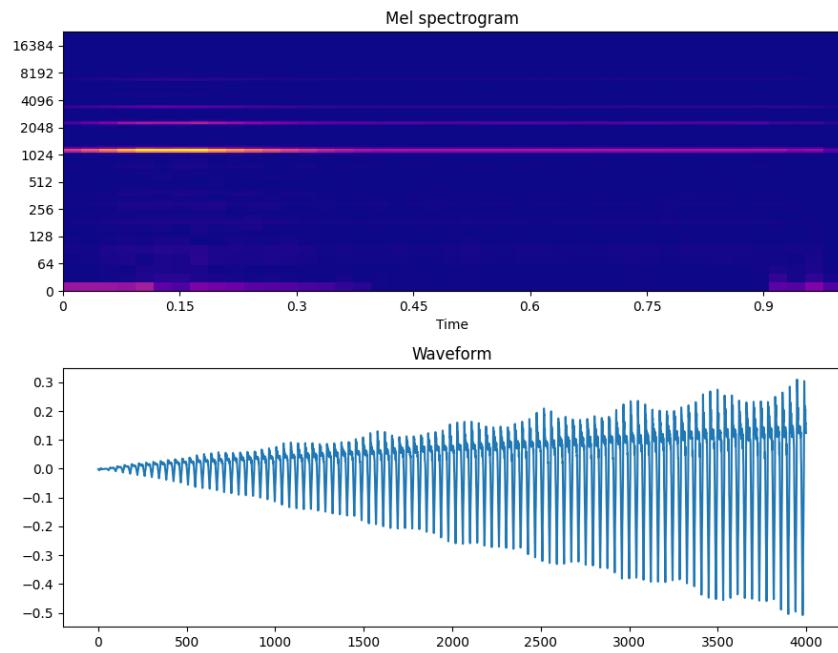
Rys. 6.3: Spektrogram i wykres fali dla dźwięku wygenerowanego przez algorytm z literatury [34] na wzór flute.wav.



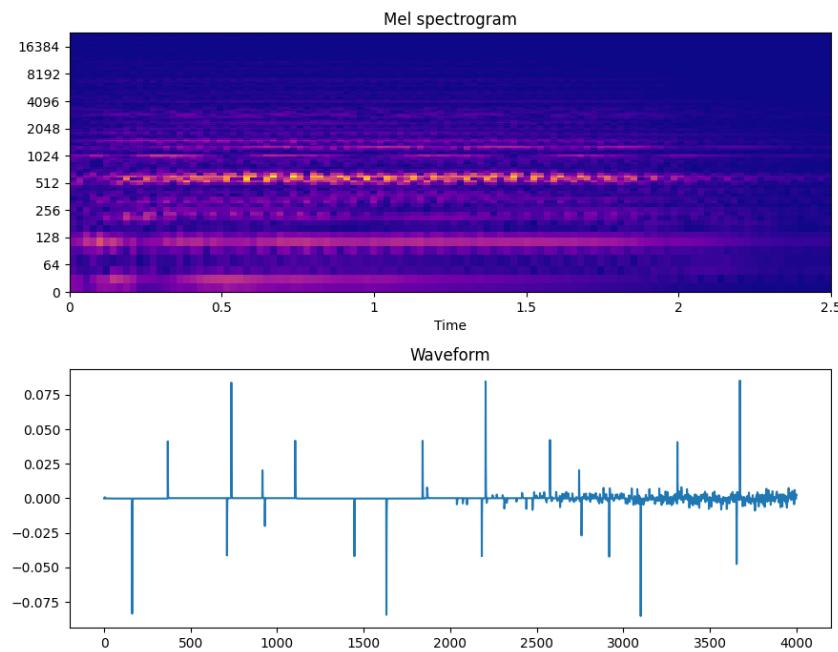
Rys. 6.4: Zmiany wartości funkcji celu podczas optymalizacji.

6.2. Sampel z syntezatora *OP-1*

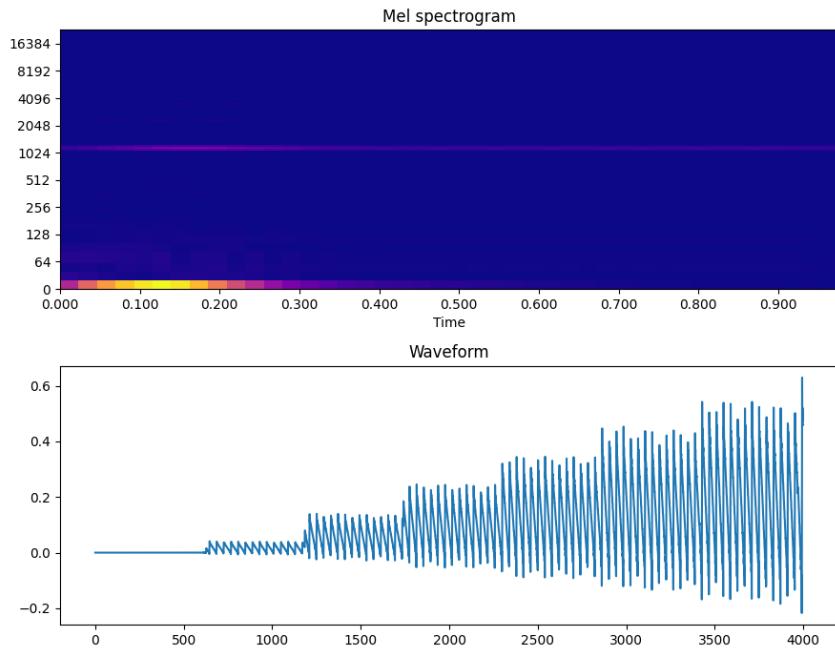
Dźwięk *op1_1.wav*, wygenerowany przy pomocy syntezatora *OP-1* jest próbą zasymulowania dźwięku instrumentu dętego przez syntezator. Podobnie jak w przypadku prawdziwego dźwięku fletu, widoczne są zmienne w czasie składowe harmoniczne, jednak sygnał wygenerowany na syntezatorze jest bardziej stabilny.



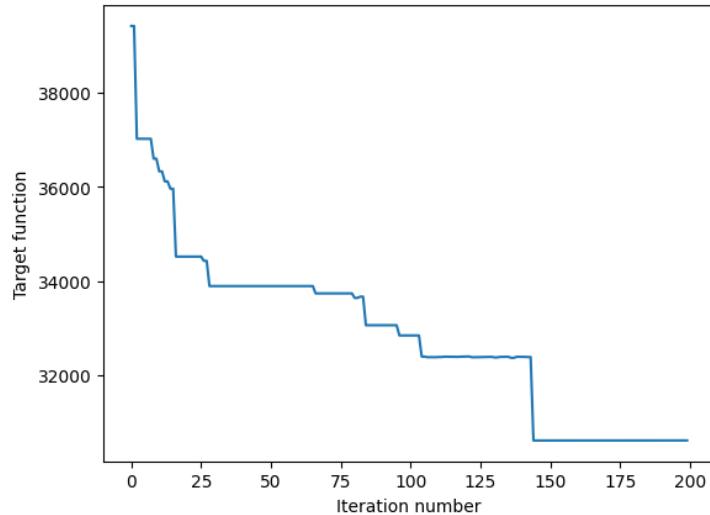
Rys. 6.5: Spektrogram i wykres fali dla dźwięku op1_1.wav wykorzystywanego do eksperymentów w [33].



Rys. 6.6: Spektrogram i wykres fali dla dźwięku wygenerowanego przez algorytm optymalizacji na wzór op1_1.wav.



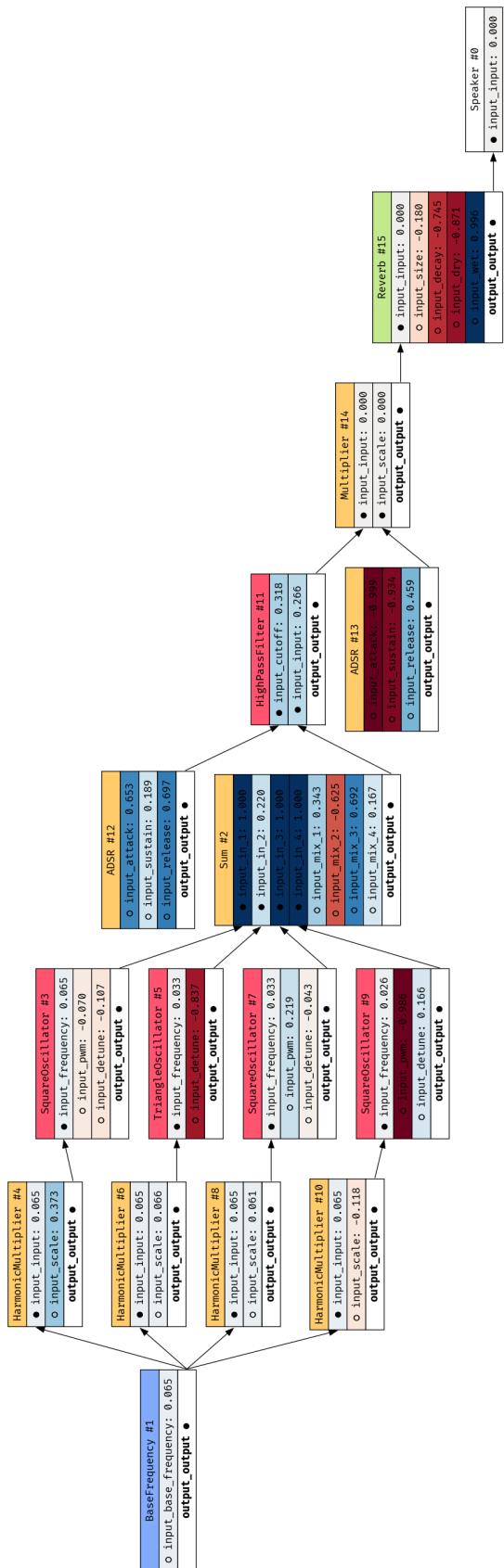
Rys. 6.7: Spektrogram i wykres fali dla dźwięku wygenerowanego przez algorytm z literatury [34] na wzór op1_1.wav.



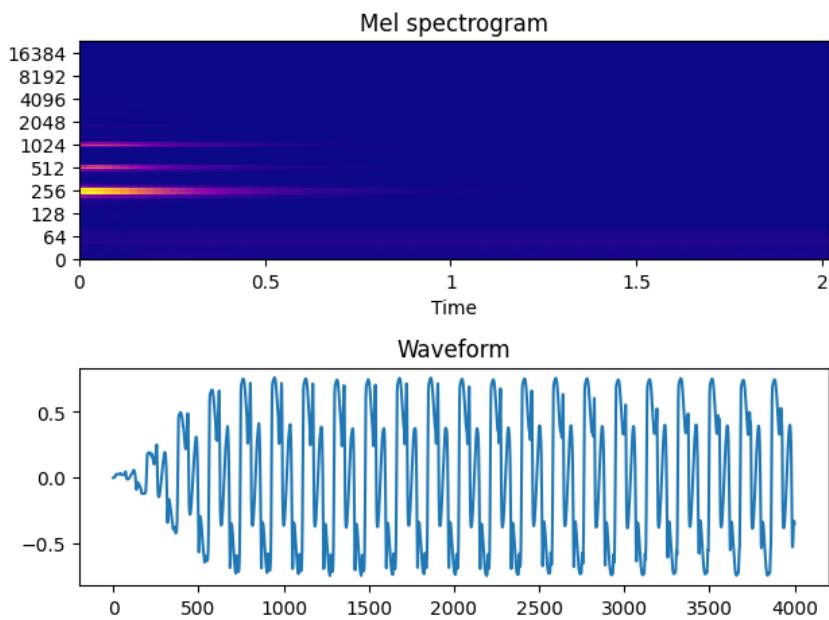
Rys. 6.9: Zmiany wartości funkcji celu podczas optymalizacji.

6.3. Transjent

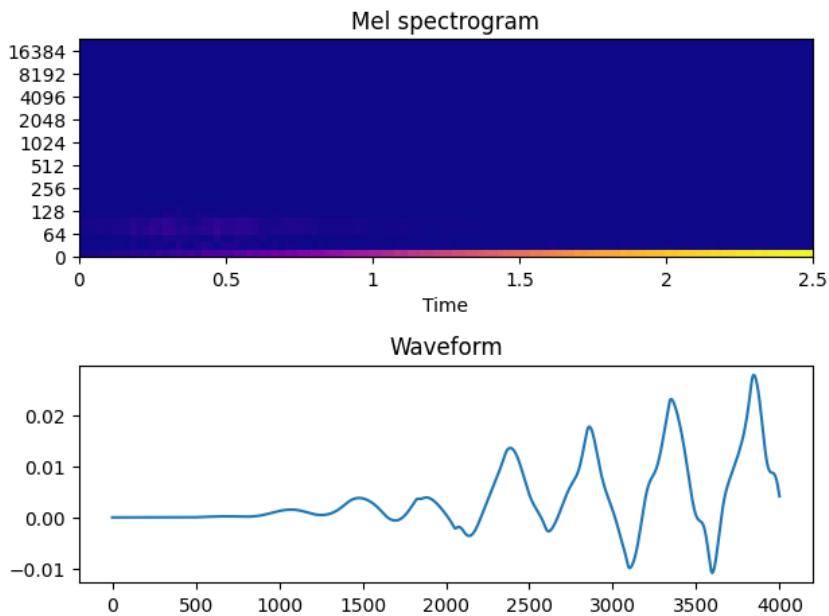
Dźwięk `transient.wav`, przedstawiony na rysunku 6.10, został wykorzystany w literaturze [34] do sprawdzenia jak dobrze algorytm generujący dźwięk potrafi przybliżyć dźwięki o dynamicznych zmianach w charakterystyce spektralnej. Tego typu dźwięki przypominają brzmienie instrumentów klawiszowych: fortepianu lub klawesynu.



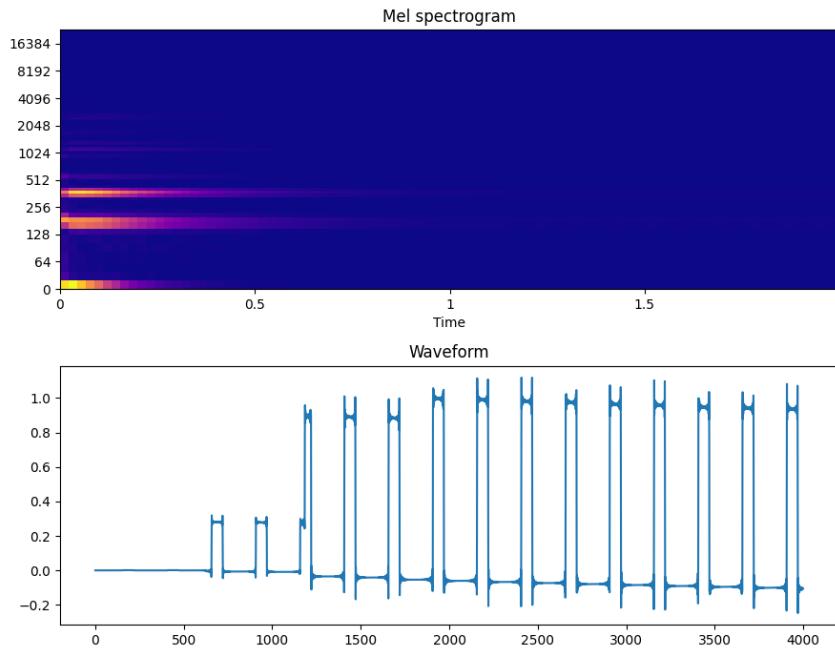
Rys. 6.8: Graf DSP wygenerowany przez zaimplementowany algorytm dla dźwięku docelowego op1_1.wav.



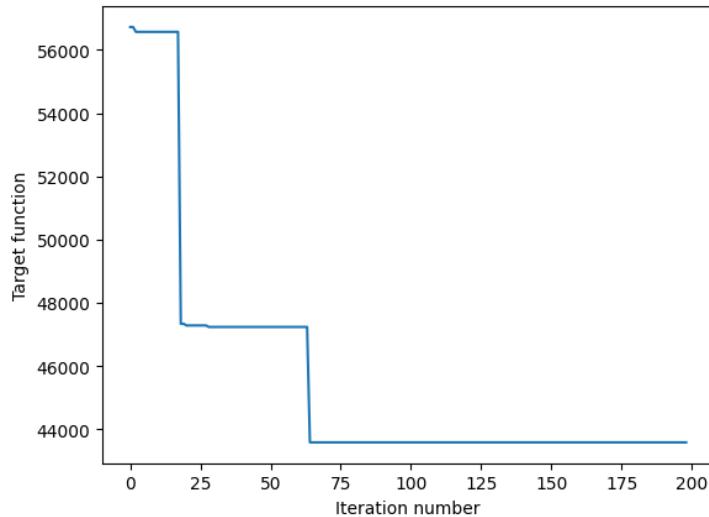
Rys. 6.10: Spektrogram i wykres fali dla dźwięku `transient.wav` wykorzystywanego do eksperymentów w [33].



Rys. 6.11: Spektrogram i wykres fali dla dźwięku wygenerowanego na wzór `transient.wav`.



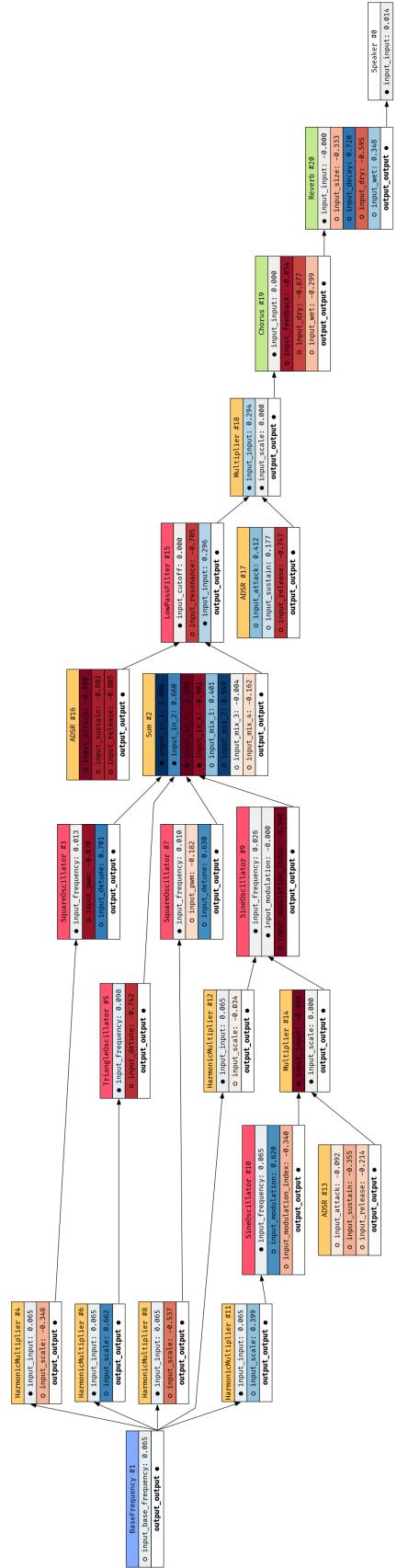
Rys. 6.12: Spektrogram i wykres fali dla dźwięku wygenerowanego przez algorytm z literatury [34] na wzór `transient.wav`.



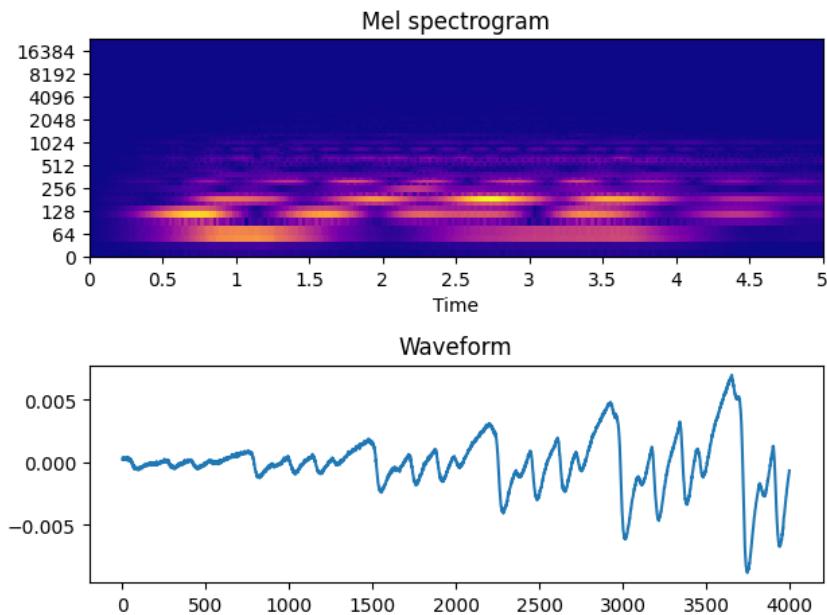
Rys. 6.13: Zmiany wartości funkcji celu podczas optymalizacji.

6.4. Dźwięki inne niż przykłady z literatury

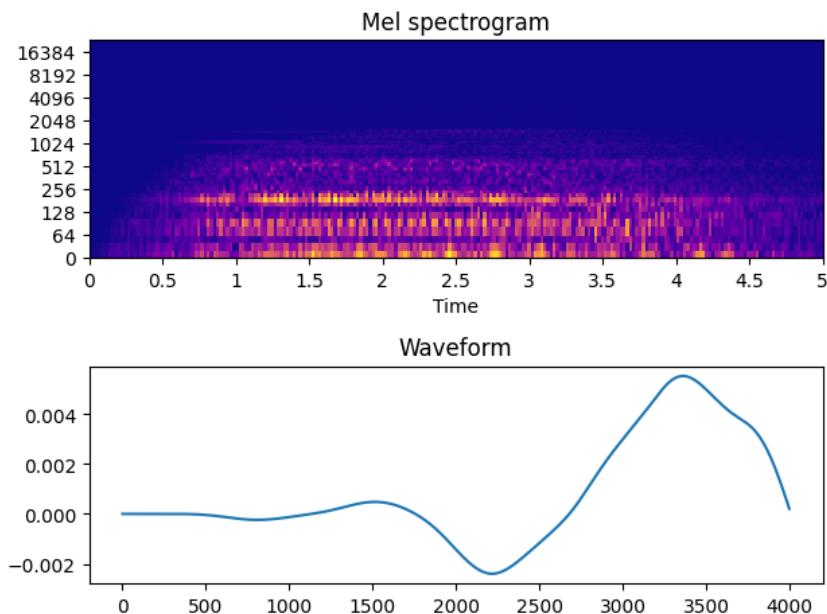
Badania przeprowadzone w ramach [34] wykorzystują ograniczony zbiór dźwięków, koncentrując się na instrumentach dętych i syntezie FM. Algorytm przetestowano również na samodzielnie wytworzonych dźwiękach, wykorzystując syntezę subtraktywną 6.15.



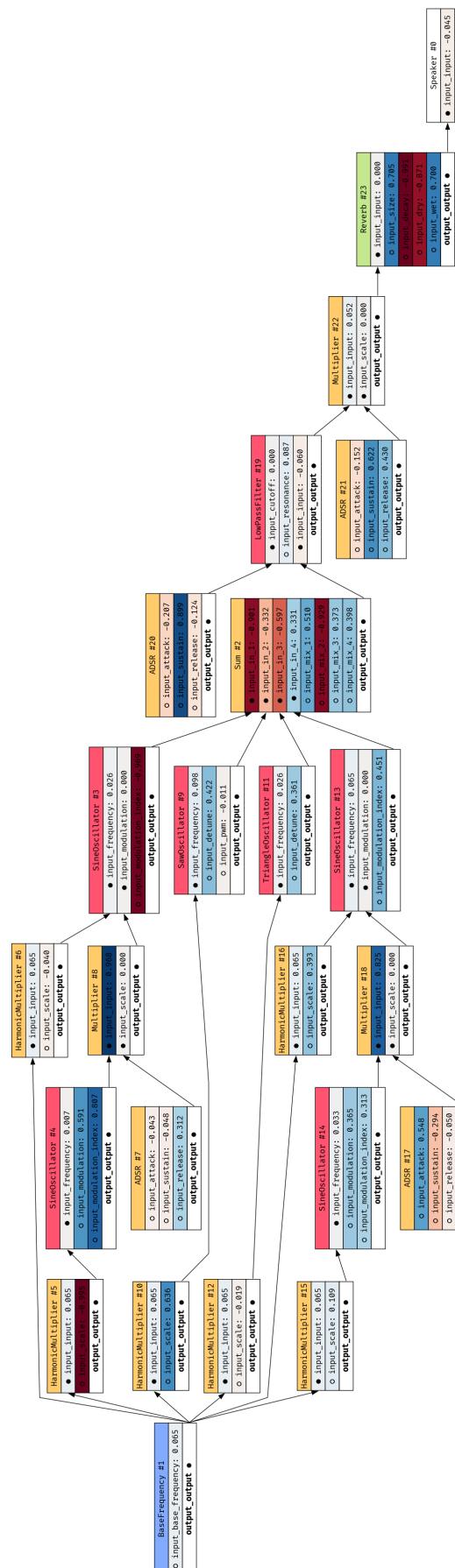
Rys. 6.14: Graf DSP wygenerowany przez zaimplementowany algorytm dla dźwięku docelowego transient.wav.



Rys. 6.15: Spektrogram i wykres fali dla dźwięku wygenerowanego na syntezatorze *Korg Minilogue xd*.



Rys. 6.16: Spektrogram i wykres fali dla dźwięku wygenerowanego przez zaimplementowany w ramach pracy algorytm na wzór 6.15.



Rys. 6.17: Graf DSP wygenerowany przez zaimplementowany algorytm dla dźwięku docelowego 6.15. Poprawnie została odtworzony filtr niskoprzepustowy oraz sterujący nim sygnał ADSR.

Rozdział 7

Analiza wyników, możliwe drogi dalszych badań

Wyniki przedstawione w rozdziałach 6 oraz 3 wskazują, że zaimplementowany algorytm optymalizacji wytwarza poprawne rozwiązania dla części problemów. Nawet gdy finalnie wygenerowane brzmienie różni się od dźwięku docelowego, algorytm wytwarza interesujące barwy dźwięku, które można zakwalifikować jako „wariacje” na temat oryginalnego sygnału. Tego typu wyniki mogą być wartościowe dla potencjalnych użytkowników algorytmu. W niniejszym rozdziale przeprowadzono szczegółową analizę obserwacji, uzyskanych podczas procesu badawczego.

7.1. Zbiór danych testowych

Automatyczna konstrukcja elektronicznych instrumentów muzycznych jest stosunkowo nowym obszarem badań, w związku z czym brakuje odpowiednich zbiorów danych umożliwiających porównywanie sprawności różnych algorytmów. Porównanie wykonane w niniejszej pracy nie stanowi przekroju przez wiele możliwych typów syntezy, ponieważ w procesie przeglądu literatury nie udało się znaleźć stosownego zbioru danych.

Jak pokazały wyniki opisane w sekcji 6.4, zaimplementowany algorytm generuje lepsze rozwiązania w przypadku syntezy *analog modeling*, podczas gdy praca służąca za porównanie [34] zawiera głównie dźwięki wytworzone za pomocą syntezy FM oraz nagrania instrumentów dętych. Sugestie dotyczące stworzenia bardziej różnorodnego zbioru dźwięków o różnych barwach i dynamice zawarto w sekcji 7.5.

7.2. Optymalizacja parametrów grafu dla problemów o małej złożoności

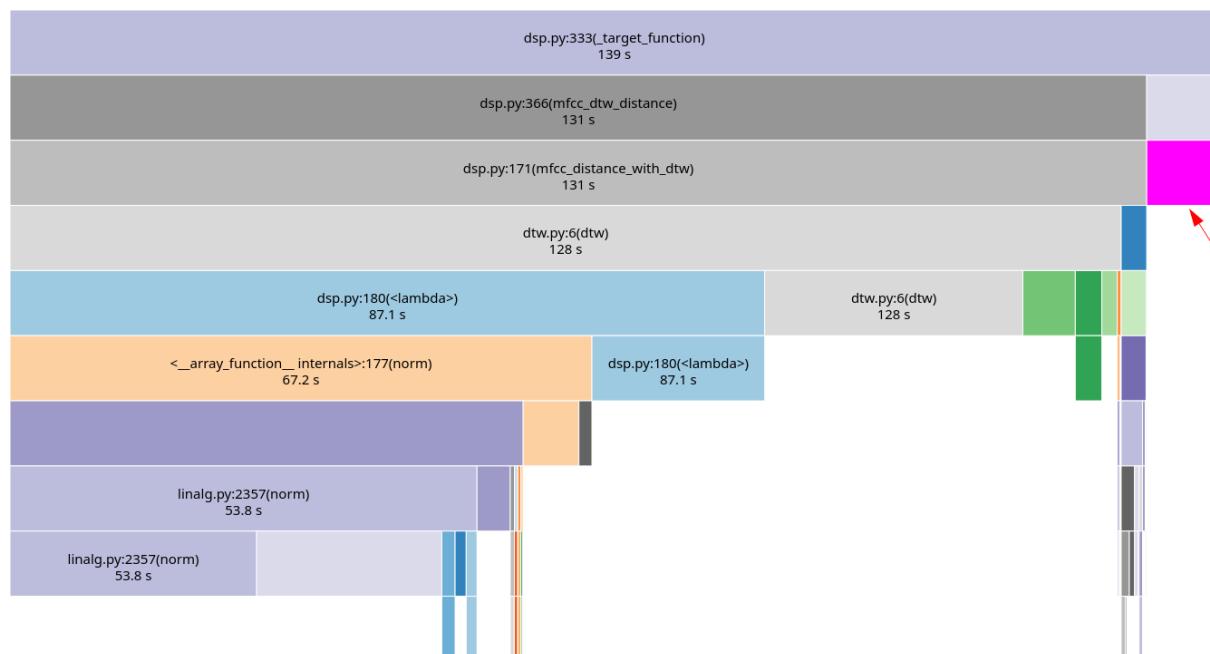
Jak wykazały testy przeprowadzone w rozdziale 3, zaimplementowany algorytm jest w stanie dokładnie odtworzyć wartości parametrów grafu dla prostych problemów syntezy FM (3.8) i umiarkowanie złożonych problemów syntezy *analog modeling* (3.10). Testy na małej próbie słuchaczy pozwalają stwierdzić, że **nie są oni w stanie odróżnić** sygnałów wygenerowanych dla problemów z rozdziału 3 od dźwięków docelowych.

7.3. Potencjał wykorzystania w przemyśle muzycznym

Wytwarzane przez algorytm barwy są zróżnicowane i brzmią w sposób „muzyczny” – nawet gdy wygenerowany dźwięk znacząco różni się od docelowej barwy, algorytm nie generuje szumu ani „kakofonii”. Zaimplementowany algorytm może być uruchomiony na standardowym komputerze, dostępnym dla przeciętnego użytkownika, co otwiera możliwość jego integracji z oprogramowaniem typu *digital audio workstation*. Integracja z oprogramowaniem do produkcji muzyki otwiera drogę do wykorzystania wtyczek VST, znacznie zwiększając liczbę dostępnych dla algorytmu węzłów przetwarzania sygnału, potencjalnie usprawniając jego działanie. Proponowanym środowiskiem, które można wykorzystać w celu zintegrowania zaimplementowanego algorytmu z oprogramowaniem *digital audio workstation* i wtyczkami w technologii VST jest program DawDreamer (<https://github.com/DBraun/DawDreamer>).

7.4. Potencjalne usprawnienia wydajności algorytmu

Rysunki 7.1 oraz 7.2 przedstawiają wyniki profilowania zaimplementowanego algorytmu, wykonane za pomocą narzędzia *snakeviz*. W pracy została wykorzystana gotowa implementacja algorytmu DTW (*dynamic time warping*) w języku Python, której wykonanie zajmuje największą część czasu obliczania wartości funkcji celu. Wykorzystanie pakietu numerycznego *numpy* bądź implementacja DTW w komplikowanym języku programowania może znacząco poprawić szybkość działania algorytmu.



Rys. 7.1: Wizualizacja danych wygenerowanych przez profiler języka Python dla przykładowego problemu optymalizacji. Widoczne składowe wpływające na sumaryczny czas ewaluacji funkcji celu. Czerwoną strzałką oznaczono czas poświęcony na syntezę dźwięku w grafie DSP.

ncalls	totime	percall	cumtime	percall	filename:lineno(function)
303	0.00333	1.099e-05	2.942	0.009709	dsp.py:115(mfcc)
302	0.00407	1.348e-05	131.2	0.4344	dsp.py:171(mfcc_distance_with_dtw)
302	0.000755	2.5e-06	131.2	0.4344	dsp.py:366(mfcc_dtw_distance)
303	0.00511	1.686e-05	2.892	0.009543	spectral.py:1844(mfcc)

Rys. 7.2: Porównanie czasu obliczania współczynników MFCC (`dsp.py:115`) oraz czasu działania algorytmu DTW (`dsp.py:171`).

7.5. Potencjalne drogi dalszego rozwoju algorytmu

Pierwszym krokiem, który należy wykonać w celu usprawnienia algorytmu optymalizacji jest przygotowanie zbioru dźwięków, które będą służyły za **zbiór weryfikujący poprawność działania algorytmu**. Przykłady zaczerpnięte z literatury [33] nie zawierają wystarczająco bogatej gamy możliwych do wygenerowania barw dźwięku, które wymusiłyby na algorytmie optymalizacji wykorzystanie różnorodnych algorytmów syntezy i struktur grafu DSP. Wyniki eksperymentów wykonanych w rozdziale 6 pozwalają na zasugerowanie zbioru cech dźwięków, których warianty powinny być zawarte w zbiorze weryfikującym:

1. Typ syntezy:
 - synteza *analog modeling*,
 - synteza FM,
 - synteza *physical modeling*.
2. Dynamika dźwięku:
 - Głośny od samego początku, dynamicznie słabnący,
 - Równomiernie głośny przez całą długość nagrania,
 - Powoli narastający i powoli słabnący.

7.5.1. Rozmiar okna w algorytmie DTW

W domyślnym wariancie, algorytm DTW przeszukuje cały sygnał przy poszukiwaniu pasujących do siebie segmentów. Takie zachowanie zwiększa złożoność czasową algorytmu, jednocześnie zmniejszając karę za niedokładne odwzorowanie zmian w dynamice dźwięku. Przeprowadzone w ramach pracy testy pozwalają wnioskować, że zmniejszenie rozmiaru okna w algorytmie DTW pozwoli na skrócenie czasu wyliczania funkcji celu i jednocześnie usprawni wyniki optymalizacji w zakresie dokładności odwzorowania dynamiki dźwięku.

7.5.2. Lepsza reprezentacja struktury grafu DSP w genotypie

Jak opisano w sekcji 2.1.1, genotyp grafu DSP podzielony jest na dwa fragmenty:

- S – fragment odpowiadający za strukturę grafu,
- P – fragment odpowiadający za wartości parametrów w grafie.

Po wygenerowaniu danej struktury grafu G_s (2.1), parametry grafu przypisywane są poprzez równoczesne iterowanie przez wektor P oraz przez listę parametrów w wygenerowanym grafie G_s . Zastosowanie takiego algorytmu powoduje, że zmiana struktury grafu może spowodować przypisanie parametrów w inne miejsca na nowym grafie, gdzie ich wartości będą miały zupełnie inny wpływ na sygnał generowany przez graf. Problem można rozwiązać poprzez trwałe przypisanie danych parametrów p_i do konkretnych parametrów konkretnej struktury w grafie. Jeśli gen tworzący daną strukturę nie będzie aktywny, geny określające parametry tej struktury

nie będą miały wpływu na generowany sygnał i nie zaburzą sposobu przypisania pozostałych genów.

7.5.3. Dalsze poszukiwania funkcji celu porównującej barwę sygnałów dźwiękowych

W pracach przeanalizowanych podczas przeglądu literatury wykorzystano 2 podejścia do porównania barwy sygnałów dźwiękowych: różnica między spektrogramami oraz różnica między wartościami współczynników MFCC (szczegóły opisano w rozdziale 3). Wyniki przeglądu literatury pozwalają podejrzewać, że porównywanie sygnałów dźwiękowych pod względem ich barwy w kontekście brzmienia muzycznego nie jest szeroko zbadanym problemem i istnieje potencjał na opracowanie nowych rozwiązań.

7.5.4. Trenowanie na coraz dłuższych fragmentach dźwięku

Ponieważ złożoność czasowa algorytmu DTW wynosi $O(N^2 / \log \log N)$ [22], im dłuższy jest zadany sygnał dźwiękowy, tym bardziej czas wyliczenia wartości funkcji celu dominuje nad czasem syntezy sygnału dźwiękowego. Potencjalnym rozwiązaniem tego problemu jest podzielenie optymalizacji na etapy, w których do optymalizacji grafu DSP wykorzystywane są coraz dłuższe fragmenty dźwięku docelowego.

7.5.5. Rozszerzenie liczby dostępnych w grafie DSP węzłów

Poza zasugerowaną w sekcji 7.3 integracją zaimplementowanego algorytmu z oprogramowaniem typu *digital audio workstation*, możliwe jest również dalsze implementowanie kolejnych węzłów generujących i przetwarzających sygnał. Ze względu na ograniczenia czasowe, w pracy nie zaimplementowano wielu rodzajów filtrów (przykładowo o różnych szerokościach pasma przepustowego) ani wielu rodzajów efektów *delay* oraz *reverb*. Zastosowanie ręcznie zaimplementowanych węzłów przetwarzania sygnału może pozwolić na przeprowadzenie dokładniejszych testów algorytmu optymalizacji (jak dostępne węzły wpływają na sprawność optymalizacji), niż w przypadku wykorzystania gotowych wtyczek VST, które spowodują znaczne zwiększenie liczby modyfikowalnych parametrów w grafie.

7.5.6. Rozszerzenie genotypu grafu DSP o dodatkowe źródła modulacji

W syntezatorach dźwięku często wykorzystuje się sygnały modulujące (przedstawione na rysunkach 1.3 oraz 1.5), które dynamicznie modyfikują parametry generowanego dźwięku za pomocą sygnałów kontrolnych (CV). W pracy wykorzystano jedynie sygnał obwiedni (*ADSR envelope*), modulujący intensywność modulacji w przypadku syntezy FM (4.4) lub częstotliwość odcięcia w przypadku syntezy *analog modeling* (4.2). Rozszerzenie zbioru dostępnych sygnałów kontrolnych o oscylator niskoczestotliwościowy (*LFO - low frequency oscillator*), który w zależności od genotypu moduluje inne parametry w grafie DSP może znaczco zwiększyć różnorodność generowanych barw dźwięku.

7.6. Subiektywna natura porównania

Generowanie dźwięku stanowi zagadnienie, które w dużej mierze zależy od subiektywnego odbioru słuchaczy. Ocena zgodności barwy dźwięku dźwięku według określonej funkcji celu niekoniecznie przekłada się na subiektywną preferencję użytkownika. Istnieje możliwość, że niedoskonałości funkcji celu mogą wpływać korzystnie na odbiór przez słuchaczy, ponieważ dostarczą

nieoczekiwanych efektów, które mogą być wykorzystane jako źródło inspiracji w procesie kreatywnym.

Literatura

- [1] Eurorack standard - wikipedia article. <https://en.wikipedia.org/wiki/Eurorack>, dostęp z dnia 20.05.23.
- [2] Ladder filter implementation. https://github.com/RustAudio/vst-rs/blob/master/examples/ladder_filter.rs, dostęp z dnia 20.05.23.
- [3] Microcosm hologram user manual. https://www.hologramelectronics.com/_files/ugd/74428b_c4e6e20555914198bdb59c12f9a9e4d4.pdf, dostęp z dnia 29.05.23.
- [4] Transient (acoustics). [https://en.wikipedia.org/wiki/Transient_\(acoustics\)](https://en.wikipedia.org/wiki/Transient_(acoustics)), dostęp z dnia 17.05.23.
- [5] Vcv rack - virtual eurorack studio. <https://vcvrack.com/>, dostęp z dnia 24.05.23.
- [6] Yamaha dx7 user manual. https://usa.yamaha.com/files/download/other_assets/9/333979/DX7E1.pdf, dostęp z dnia 03.06.23.
- [7] Yamaha vl1 user manual. https://europe.yamaha.com/files/download/other_assets/9/321049/VL1E1.pdf, dostęp z dnia 01.06.23.
- [8] Maturin - build and publish crates with pyo3, rust-cpython, cffi and uniffi bindings as well as rust binaries as python packages., 2023. <https://github.com/PyO3/maturin>, dostęp z dnia 04.05.23.
- [9] Pure data - an open source visual programming language for multimedia, 2023. <https://puredata.info/>, dostęp z dnia 25.05.23.
- [10] The rust reference - procedural macros, 2023. <https://doc.rust-lang.org/reference/procedural-macros.html>, dostęp z dnia 29.05.23.
- [11] B. Bozkurt, K. A. Yüksel. Parallel evolutionary optimization of digital sound synthesis parameters. e. Di Chio, redaktor, *Applications of Evolutionary Computation*, strony 194–203, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.
- [12] F. Caspe, A. McPherson, M. Sandler. Ddx7: Differentiable fm synthesis of musical instrument sounds, 2022.
- [13] R. Challinor. Bespoke synth - a modular daw for mac, windows, and linux., 2023. <https://www.bespokesynth.com/>, dostęp z dnia 20.05.23.
- [14] N. Collins. The analysis of generative music programs. *Organised Sound*, 13(3):237–248, 2008.
- [15] J. Dattorro. Effect design 1: Reverberator and other filters. 1997.
- [16] Elektron. Elektron digitone user manual, 2022. https://cdn.www.elektron.se/media/downloads/digitone/Digitone_User_Manual_ENG_OS1.40A_221123.pdf, dostęp z dnia 21.05.23.

- [17] J. Engel, L. H. Hantrakul, C. Gu, A. Roberts. Ddsp: Differentiable digital signal processing. *International Conference on Learning Representations*, 2020. <https://openreview.net/forum?id=B1x1ma4tDr>, dostęp z dnia 02.06.23.
- [18] J. Engel, C. Resnick, A. Roberts, S. Dieleman, D. Eck, K. Simonyan, M. Norouzi. Neural audio synthesis of musical notes with wavenet autoencoders, 2017.
- [19] D. Faronbi, I. Roman, J. P. Bello. Exploring approaches to multi-task automatic synthesizer programming. *ICASSP 2023 - 2023 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, strony 1–5, 2023.
- [20] S. Forsgren, H. Martiros. Riffusion - Stable diffusion for real-time music generation. 2022.
- [21] P. S. Foundation. Python documentation - extending python with c or c++, 2019. <https://docs.python.org/3/extending/extending.html>, dostęp z dnia 20.05.23.
- [22] O. Gold, M. Sharir. Dynamic time warping and geometric edit distance: Breaking the quadratic barrier. *ACM Trans. Algorithms*, 14(4), aug 2018.
- [23] N. Hind. *Common Lisp Music (CLM) Tutorials*. wydanie first, 2021. Available for free at <https://ccrma.stanford.edu/software/clm/compmus/clm-tutorials/toc.html>.
- [24] E. Jacobsen, R. Lyons. The sliding dft. *IEEE Signal Processing Magazine*, 20(2):74–80, 2003.
- [25] S. Ji, J. Luo, X. Yang. A comprehensive survey on deep music generation: Multi-level representations, algorithms, evaluations, and future directions, 2020.
- [26] S. Kacprzak. Inteligentne metody rozpoznawania dźwięku.
- [27] T. Kathiresan. *Automatic Melody Generation*. Praca doktorska, 06 2015.
- [28] Y. Ke, D. Hoiem, R. Sukthankar. Computer vision for music identification. *2005 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'05)*, wolumen 1, strony 597–604 vol. 1, 2005.
- [29] A. F. Khalifeh, A.-K. Al-Tamimi, K. A. Darabkh. Perceptual evaluation of audio quality under lossy networks. *2017 International Conference on Wireless Communications, Signal Processing and Networking (WiSPNET)*, strony 939–943, 2017.
- [30] K. Kilgour, M. Zuluaga, D. Roblek, M. Sharifi. Fréchet audio distance: A metric for evaluating music enhancement algorithms, 2018. <https://arxiv.org/abs/1812.08466>.
- [31] KORG. Korg minilogue xd user manual, 2017. <https://www.korg.com/us/support/download/product/0/811/>, dostęp z dnia 21.05.23.
- [32] S. Luke. *Computational Music Synthesis*. wydanie first, 2021. Available for free at <http://cs.gmu.edu/~sean/book/synthesis/>.
- [33] M. Macret, P. Pasquier. Automatic design of sound synthesizers as pure data patches using coevolutionary mixed-typed cartesian genetic programming – trained sounds. <https://metacreation.net/mmacret/GECCO2014>, dostęp z dnia 22.05.23.
- [34] M. Macret, P. Pasquier. Automatic design of sound synthesizers as pure data patches using coevolutionary mixed-typed cartesian genetic programming. *Proceedings of the 2014 Annual Conference on Genetic and Evolutionary Computation*, GECCO '14, strona 309–316, New York, NY, USA, 2014. Association for Computing Machinery.

-
- [35] M. Mauch, S. Dixon. Pyin: A fundamental frequency estimator using probabilistic threshold distributions. *2014 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, strony 659–663, 2014.
 - [36] B. McFee, C. Raffel, D. Liang, D. Ellis, M. McVicar, E. Battenberg, O. Nieto. librosa: Audio and music signal analysis in python. strony 18–24, 01 2015.
 - [37] R. Rombach, A. Blattmann, D. Lorenz, P. Esser, B. Ommer. High-resolution image synthesis with latent diffusion models, 2021.
 - [38] J. O. Smith. *Introduction to Digital Filters with Audio Applications*. W3K Publishing, <http://www.w3k.org/books/>, dostęp z dnia 20.04.23, 2007.
 - [39] J. O. Smith. Physical signal audio processing. 2010.
 - [40] J. O. Smith. *Spectral Audio Signal Processing*. <https://ccrma.stanford.edu/~jos/sasp/>, accessed 20.04.2023. online book, 2011 edition.
 - [41] M. Sood, S. Jain. Speech recognition employing mfcc and dynamic time warping algorithm. P. K. Singh, Z. Polkowski, S. Tanwar, S. K. Pandey, G. Matei, D. Pirvu, redaktorzy, *Innovations in Information and Communication Technologies (IICT-2020)*, strony 235–242, Cham, 2021. Springer International Publishing.
 - [42] P. Virtanen, R. Gommers, T. E. Oliphant, H. et al. SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python. *Nature Methods*, 17:261–272, 2020.
 - [43] L. Zhang, C. Callison-Burch. Language models are drummers: Drum composition with natural language pre-training, 2023.
 - [44] F. Zheng, G. Zhang, Z. Song. Comparison of different implementations of mfcc. *Journal of Computer Science and Technology*, 16(6):582–589, Nov 2001. <https://doi.org/10.1007/BF02943243>.

Dodatek A

Instrukcja wdrożeniowa

Algorytm optymalizacyjny i środowisko eksperymentowe zostały zaimplementowane w systemie Linux, powinny działać również na innych systemach z rodziną UNIX oraz na platformie Windows.

A.1. Wymagane oprogramowanie

- Instalacja języka Python w wersji 3.10 (w momencie pisania pracy biblioteka `llvmlite`, wymagana przez `librosa`, wspierała najwyższą wersję 3.10),
- instalacja języka Rust w wersji 1.71 lub wyższej. Rekomendowana instalacja z wykorzystaniem <https://rustup.rs/>, toolchain w wersji stable,
- pakiet `poetry` (<https://python-poetry.org/>) do zarządzania zależnościami w języku Python.
- pakiet `graphviz` dostępny za pośrednictwem zmiennej środowiskowej `$PATH`.
- biblioteka `libasound2-dev` lub jej odpowiednik dla danego systemu operacyjnego (nie wymagana w przypadku systemu Windows).

A.2. Instalacja

Środowisko eksperymentowe wykorzystuje bibliotekę *extension module* [21], zaimplementowaną w języku Rust, proces instalacji wymaga zainstalowania bibliotek języka Python **oraz** skompilowania *extension module*. W powłoce systemu operacyjnego należy uruchomić następujące polecenia:

```
poetry install # instalacja pakietów dla Pythona  
poetry run maturin develop # skompilowanie extension module w Ruście
```

A.3. Weryfikacja poprawności działania projektu

Projekt wyposażony jest w testy jednostkowe, które należy uruchomić w celu zweryfikowania poprawności instalacji:

```
cargo test # Uruchomienie testów jednostkowych dla języka Rust  
poetry run pytest . # Uruchomienie testów dla języka Python
```

A.4. Praca ze środowiskiem eksperymentowym

Repozytorium projektu zawiera pliki `.ipynb` dla środowiska *Jupyter Lab*, które jest instalowane jako zależność projektu. Repozytorium zawiera następujące „notebooki”:

1. `demo.ipynb` – prezentuje podstawowe funkcjonalności projektu: budowę grafów, analizę barwy dźwięku oraz generowanie wykresów.
2. `target_function.ipynb` – wykonuje porównanie między różnymi funkcjami celu, opisane w rozdziale 3.
3. `prototyping.ipynb` – zawiera kod dynamicznie budujący strukturę grafu DSP na podstawie genotypu oraz wykonuje optymalizację struktury i parametrów grafu dla porównania opisanego w rozdziale 6.

A.4.1. Funkcja celu i detale algorytmu genetycznego

Dostępne funkcje celu oraz algorytm genetyczny optymalizujący strukturę i parametry grafu zaimplementowane są w pliku `luthier/dsp.py`.

A.4.2. Implementacja algorytmów syntezy i grafu DSP

Algorytmy związane z syntezą dźwięku i grafem DSP podzielone są na 3 moduły:

1. `node_traits/lib.rs` – definicja interfejsów (*traits* w języku Rust) dla węzłów w grafie DSP,
2. `node_macro/lib.rs` – makro proceduralne ([10]) automatycznie implementujące powtarzalne interfejsy zdefiniowane w `node_traits`,
3. `src/lib.rs` - implementacja węzłów DSP oraz grafu DSP, wykorzystująca wyżej wymienione moduły.

Dodatek B

Opis załączonej płyty CD/DVD

Załączona płyta CD zawiera następujące pliki:

1. **luthier.zip** – archiwum zawierające repozytorium ze środowiskiem eksperymentowym zaimplementowanym w ramach pracy. Stan z dnia przekazania finalnej wersji pracy na uczelnię,
2. **Dyplom.pdf** – praca dyplomowa w formacie PDF, stan z dnia przekazania finalnej wersji pracy na uczelnię.
3. **snapshots/** – katalog zawierający logi z przebiegu optymalizacji dla dźwięków wytwarzanych w rozdziale 6.