POLITECHNIKA ŚLĄSKA

WYDZIAŁ AUTOMATYKI, ELEKTRONIKI I INFORMATYKI

Praca dyplomowa inżynierska

Wykrywanie tempa i metrum w utworze muzycznym

autor: Marek Żabiałowicz

kierujący pracą: dr inż. Daniel Kostrzewa

Gliwice, styczeń 2020

załącznik nr 2 do zarz. nr 97/08/09

Oświadczenie

Wyrażam zgodę / Nie wyrażam zgody* na udostępnienie mojej pracy dyplomowej / rozprawy doktorskiej*.

Gliwice, dnia 15 stycznia 2020

.....
(podpis)

.....
(poświadczenie wiarygodności
podpisu przez Dziekanat)

* podkreślić właściwe

Oświadczenie promotora

Oświadczam, że praca „Wykrywanie tempa i metrum w utworze muzycznym” spełnia wymagania formalne pracy dyplomowej inżynierskiej.

Gliwice, dnia 15 stycznia 2020

.....

(podpis promotora)

Spis treści

1 Wstęp **1**

1.1 Cel i opis pracy 2

2 Analiza tematu **3**

2.1 Tempo i metrum 3

2.2 Przetwarzanie sygnałów 4

2.2.1 Próbkowanie 4

2.2.2 Transformacja Fouriera 4

2.2.3 Splot i funkcja korelacji 5

2.2.4 Filtr grzebieniowy 5

2.2.5 Energia sygnału 6

2.3 Cyfrowe przetwarzanie dźwięku 6

2.4 Studia literaturowe 7

2.5 Rozwiązania komercyjne 8

3 Wymagania i narzędzia **11**

3.1 Wymagania funkcjonalne 11

3.2 Wymagania niefunkcjonalne 12

3.3 Opis narzędzi 13

3.3.1 Język programowania 13

3.3.2 Środowisko programistyczne 13

3.3.3 Kontrola wersji 14

3.3.4 Wykorzystane moduły Python 14

3.4 Metodyka pracy 14

4 Specyfikacja zewnętrzna **17**
4.1 Wymagania sprzętowe i programowe 17
4.2 Sposób aktywacji 17
4.2.1 Parametry 17
4.3 Pomoc i obsługa błędów 19

5 Specyfikacja wewnętrzna **21**
5.1 Wczytanie pliku .wav 22
5.2 Wycięcie fragmentu ze środka utworu 22
5.3 Wyrównanie fragmentu do pierwszego napotkanego pulsu 23
5.4 Transformacja sygnału do dziedziny częstotliwości 25
5.5 Wygładzenie za pomocą okna czasowego 25
5.6 Obliczenie pochodnej 25
5.7 Wykrycie tempa za pomocą filtra grzebieniowego 27
5.7.1 Alternatywne implementacje wykrywania 30
5.7.2 Normalizacja sygnału transformaty 31
5.7.3 Liczenie splotu sygnałów 31
5.8 Wykrycie metrum 32
5.8.1 Wykrycie metrum za pomocą filtra grzebieniowego 32
5.8.2 Wykrycie metrum za pomocą filtra grzebieniowego ze znormalizowanym sygnałem 32
5.8.3 Wykrycie metrum za pomocą splotu 33
5.8.4 Wykrycie metrum za pomocą splotu ze znormalizowanym sygnałem 33
5.8.5 Wykrycie metrum za pomocą funkcji autokorelacji 33
5.9 Optymalizacja 33
5.9.1 Zmniejszenie częstości próbkowania 33
5.9.2 Rekurencyjne wykrywanie tempa 34
5.9.3 Zastosowanie architektury CUDA 34

6 Weryfikacja i walidacja **35**
6.1 Zbiór testowy 35
6.2 Organizacja testów 35

7 Podsumowanie i wnioski **39**
7.1 Analiza wyników 39
7.1.1 Analiza wyników wykrycia tempa 39
7.1.2 Analiza wyników wykrycia metrum 40
7.2 Dalsze prace 40
7.3 Podsumowanie 41

Rozdział 1

Wstęp

Wykrywanie tempa i metrum utworu muzycznego jest podstawowym zagadnieniem potrzebnym w dziedzinie cyfrowej obróbki muzyki. Ma zastosowanie zarówno dla producentów muzycznych, którym zaznaczanie linii taktów pomaga przy edycji nagrania, na przykład przy operacjach typu kopiuj/wklej, jak i dla dj-ów tworzących muzykę na żywo, którzy potrzebują wyrównać równolegle odtwarzane utwory i sample.

Wiedza na temat tempa i metrum utworu może również pomóc przy próbie jego automatycznego skatalogowania. Mając informacje o wykonawcy, tytule oraz gatunku, a dodatkowo wykrywając tempo, metrum i nastrój możemy stworzyć bazę utworów, na której możemy tworzyć zapytania o zbiór piosenek podstawie humoru słuchacza lub przeznaczenia listy odtwarzania, która ma powstać. Na przykład, biegacz mógłby poprosić o utwory w tempie około 150 BPM (ang. *Beats per minute*), ponieważ do takich mu się najwygodniej biega. Chcąc znaleźć dla niego najlepszą muzykę, będziemy szukać jej wśród żywych utworów z prostym, dwudzielnym metrum, na przykład 4/4.

Stworzenie bazy danych utworów, gdzie każda pozycja będzie miała określone tempo, metrum, gatunek oraz nastrój może pomóc producentom muzycznym oraz filmowym. Istnieje wiele serwisów oferujących bogatą kolekcję utworów na licencji „Creative Commons", w których niestety nie ma podanych informacji innych niż autor, tytuł i gatunek. Automatycznie wykryte informacje o charakterze utworu mogłyby ułatwić pracę i przyspieszyć kompletowanie ścieżki dźwiękowej.

Większa wiedza o utworach mogłaby się również przydać w coraz częściej używanych serwisach streamingowych takich jak Deezer[1] czy Tidal[2]. W Spotify[3] dostępna jest opcja dynamicznego tworzenia list odtwarzania skierowanych do danego użytkownika. Jeżeli ktoś w danej chwili słucha powolnego rocka i skończą mu się piosenki na liście, dobrym pomysłem może być puszczenie kolejnej powolnej rockowej piosenki innego artysty, tak by słuchacz nawet się nie zorientował, że skończyły się utwory i nie nastąpiła cisza.

1.1 Cel i opis pracy

Celem tej pracy inżynierskiej jest napisanie programu, który potrafi wykryć tempo oraz metrum utworu. W literaturze (sekcja 2.4) opisane jest kilka metod wykrywania o podobnej zasadzie działania, dlatego zostanie zaimplementowane parę metod, w celu zbadania, która daje najlepsze wyniki. Napisany program ma umożliwiać wybór metody wykrywania tempa oraz metrum.

W rozdziale 2 zostały wyjaśnione użyte metody przetwarzania sygnałów oraz takie zagadnienia jak tempo i metrum. Następnie opisano pozycje w bibliografii i znane rozwiązania komercyjne. Rozdział 3 zawiera informacje o wymaganiach funkcjonalnych i niefunkcjonalnych, użytych narzędziach i technologiach, środowiskach programistycznych w których pisano kod oraz przyjętą metodykę pracy. W dalszej części przedstawiona została specyfikacja zewnętrzna. Zawiera ona opis wywołania programu, opisane przełączniki oraz przykładowe zrzuty ekranu. W rozdziale 5 została opisana specyfikacja wewnętrzna. Ta część pracy zawiera opis wykorzystanych algorytmów oraz sposób ich działania wraz z fragmentami kodu. W kolejnym rozdziale został opisany sposób testowania programu. Podsumowanie zawiera analizę uzyskanych wyników oraz opis dalszych prac.

[1] `https://www.deezer.com/pl/`
[2] `https://tidal.com/`
[3] `https://www.spotify.com/pl/`

Rozdział 2

Analiza tematu

2.1 Tempo i metrum

Tempo utworu określa, jak szybko dany utwór ma zostać wykonany. Istnieją dwa sposoby zapisu tempa utworu. Pierwszym, częściej spotykanym w zapisie nutowym, sposobem są umowne wyrazy zaczerpnięte z języka włoskiego np.: moderato (pl. *umiarkowanie* ok. 90 BPM), allegro (pl. *żywo* ok. 120 BPM). Drugim sposobem, bardziej współczesnym i rzadziej spotykanym z powodu cen metronomów mechanicznych w przeszłości, jest podanie tempa wprost za pomocą liczby określającej liczbę uderzeń na minutę, czyli BPM. 120 BPM odpowiada 2 Hz.

Metrum określa schemat akcentów w obrębie taktu. Możemy je dzielić według trzech kryteriów:

- Liczbę akcentów w obrębie taktu:
  - metrum proste - jeden akcent w obrębie jednego taktu,
  - metrum złożone -więcej niż jeden akcent w obrębie jednego taktu.
- Liczbę miar akcentowanych i nieakcentowanych:
  - metrum dwudzielne albo parzyste - po mierze akcentowanej przypada jedna miara nieakcentowana,
  - metrum trójdzielne albo nieparzyste - po mierze akcentowanej przypadają dwie miary nieakcentowane.

- Układu metrów prostych w metrum złożonym:
  - metrum złożone regularne - wszystkie metra proste składające się na metrum złożone są identyczne,
  - metrum złożone nieregularne - (przynajmniej jedno metrum proste wchodzące w skład metrum złożonego jest inne niż pozostałe.

Metrum może być powiązane z gatunkiem utworu. Na przykład walce są w 3/4, a polki w 2/4. Współcześnie większość piosenek w muzyce rozrywkowej jest grana w 4/4, 3/4, 6/8.

2.2 Przetwarzanie sygnałów

Przetwarzanie sygnałów jest dziedziną matematyki stosowanej, która zajmuję się wykonywaniem operacji na sygnałach i ich interpretacją. W tej pracy posłużono się metodami cyfrowego przetwarzania sygnałów.

2.2.1 Próbkowanie

Dźwięki instrumentów są analogowym sygnałem ciągłym. Żeby móc przetwarzać sygnał analogowy na komputerze, musi on zostać poddany dyskretyzacji. Próbkowanie to proces tworzenia sygnału dyskretnego za pomocą ciągu wartości zwanych próbkami. Bardzo istotne jest twierdzenie 1 o próbkowaniu, jeżeli chcemy by odtworzony sygnał był bez zniekształceń.

Twierdzenie 1 (Twierdzenie Nyquista-Whittakera-Kotelnikova[6])**.** *Częstotliwość próbkowania f_s musi być większa niż dwukrotność najwyższej składowej częstotliwości f_p w mierzonym sygnale.*

$$f_s > 2 \cdot f_p$$

2.2.2 Transformacja Fouriera

Sygnał można zapisać jako iloczyn funkcji sinusoidalnych. Transformacja Fouriera (wzór 2.1) jest funkcją przekształcającą sygnał z dziedziny czasu t do dziedziny częstości kołowej ω.

$$\hat{f}(\omega) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{\infty} f(t) e^{-i\omega t} dt \tag{2.1}$$

Funkcją odwrotną jest odwrotna transformacja Fouriera wyrażona wzorem 2.2[6],

$$f(t) = \frac{1}{2\pi} \int_{-\infty}^{\infty} \hat{f}(\omega) e^{i\omega t} d\omega \tag{2.2}$$

gdzie:
$f(t)$ – funkcja w dziedzinie czasu,
$\hat{f}(\omega)$ - transformata (widmo sygnału) w dziedzinie pulsacji,
$\omega = \frac{2\pi}{T} = 2\pi\nu$ - pulsacja proporcjonalna do częstotliwości oscylacji ν.

Dyskretna transformata Fouriera DFT (ang. *Discrete Fourier Transform*) jest transformata Fouriera dla sygnału dyskretnego. Funkcją odwrotną jest odwrotna dyskretna transformata Fouriera IDFT (ang. *Inverse Discrete Fourier Transform*).

2.2.3 Splot i funkcja korelacji

Splot jest funkcją dla dwóch sygnałów, której wynikiem jest jest zmodyfikowana wersja w postaci iloczynu splotowego. Funkcja splotu dyskretnego dla funkcji f i g jest opisana wzorem 2.3[6].

$$(f * g)[n] \ = \ \sum_{m=-\infty}^{\infty} f[m]\, g[n-m] \tag{2.3}$$

Podobną funkcją jest funkcja korelacji wyrażona wzorem 2.4[6], której wynikiem jest miara podobieństwa sygnałów $x[n]$ i $y[n]$.

$$r_{xy}[l] = \sum_{n=-\infty}^{\infty} x[n+l] y*[n] \tag{2.4}$$

2.2.4 Filtr grzebieniowy

Jednym z podstawowych filtrów w przetwarzaniu sygnałów jest filtr grzebieniowy (ang. *Comb filter*), którego zasada działania opiera się na symulacji opóź-

nienia sygnału oryginalnego o jednostkę czasu. Poszczególnego składowe częstotliwości są wzmacnianie lub wygłuszane poprzez zjawisko interferencji.

2.2.5 Energia sygnału

W celu znalezienia filtru, którego sygnał jest najbardziej zbliżony do sygnału oryginalnego obliczana jest energia ze wzoru 2.5[6].

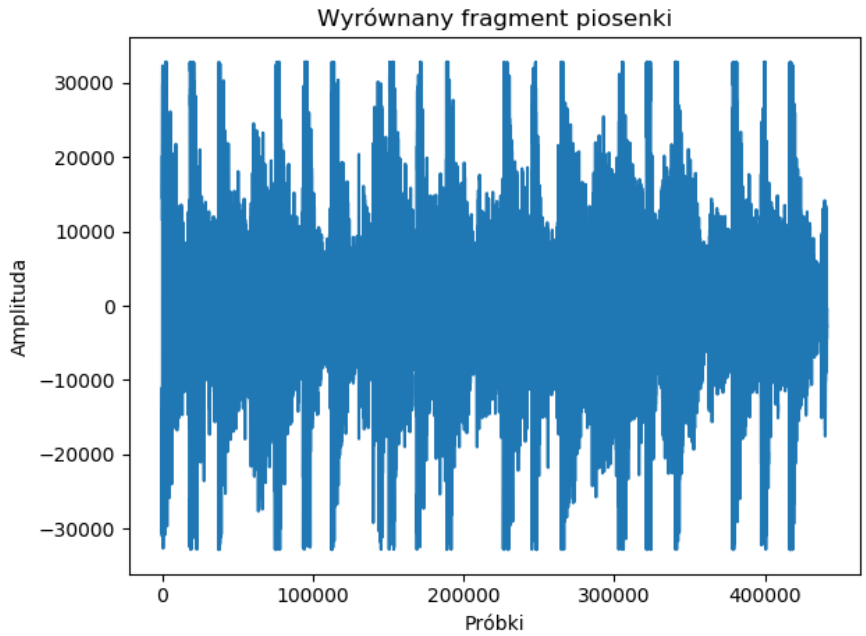
$$E = \sum_{n=-\infty}^{\infty} |x(n)|^2 \tag{2.5}$$

2.3 Cyfrowe przetwarzanie dźwięku

Cyfrowe przetwarzanie nagranego dźwięku, w tym wykrywanie tempa i metrum utworu, zaczęto dynamicznie rozwijać na początku lat 90., kiedy to domowy sprzęt komputerowy pozwolił na zapis nagranej muzyki w pamięci komputera, a następnie jej obróbkę w czasie rzeczywistym.

Wcześniej było to niemożliwe ponieważ ilość danych z nagranego utworu przerastała możliwości ówczesnych komputerów domowych. Zakres częstotliwości słyszalnych przez człowieka to 8 Hz - 20 kHz[11], a znając twierdzenie 1 wiemy, że częstotliwość próbkowania nagranego dźwięku powinna wynosić co najmniej 40 kHz. Z tego powodu, jak i standardu dla płyt CD wprowadzonego przez SONY w 1979 roku, standardową częstotliwością próbkowania dla plików audio zostało 44.1 kHz przy rozmiarze 16 bitów na próbkę. Zatem nieskompresowany trzyminutowy utwór stereo nagrany przy próbkowaniu 44.1 kHz ma rozmiar 31,752 MB. Obecnie taki plik wydaje się być mały i potrafimy go przetwarzać, a także modyfikować w czasie rzeczywistym, ale dla komputerów z lat 80. było to za dużo danych.

Rysunek **??** przedstawia reprezentacje graficzną nagranego utworu. Jest to wykres zmian amplitudy natężenia sygnału. Wyraźnie widać skoki natężenia sygnału. W miejscach występowania tych maksimów lokalnych najprawdopodobniej znajdują się uderzenia pulsu utworu. Celem tej pracy jest znaleźć zależność pomiędzy częstotliwością i metrum a natężeniem i częstotliwością występowania pulsów.



Rysunek 2.1: Reprezentacja graficzna 10 sekund nagranego utworu techno

2.4 Studia literaturowe

Istnieje wiele prac i badań w temacie wykrywania tempa, ale opisane w nich algorytmy są do siebie bardzo zbliżone. Wykrywanie metrum jest zagadnieniem rzadziej opisywanym w literaturze i artykułach.

Najczęściej wykorzystywaną metodą do wykrywania tempa jest stworzenie zespołu filtrów (ang. *filterbank*) z wykorzystaniem DFT. Jedną z możliwości jest utworzenie zespołu filtrów z użyciem filtra prostokątnego.[3, 9]. Inną metodą stworzenia jest wykorzystanie filtru o skończonej odpowiedzi impulsowej FIR (ang. *Finite Impulse Response Filter*)[4, 7] lub filtru o nieskończonej odpowiedzi impulsowej IIR (ang. *Infinite Impulse Response*).[10]

Zamiast zespołu filtrów, można obliczyć obwiednie sygnału[10] lub funkcję przepływu spektrogramu.[8] Obliczona w ten sposób funkcja, zmienia się wolniej niż sygnał oryginalny i lepiej nadaje się do przetwarzania. W porównaniu do zespołu filtrów, te metody dają lepsze rezultaty w muzyce bez sekcji rytmicznej.

Jednym z sposobów wykrycia tempa jest utworzenie serii filtrów grzebieniowych, w których częstotliwość pulsacji opowiada częstotliwości uderzeń w tempie[3,

9],a następnie znalezienie maksymalnej sumy iloczynów widm sygnałów z zespołu filtrów z widmem filtra.[3, 9] Częstotliwość pulsacji filtra, dla którego obliczono maksymalną sumę energii jest najprawdopodobniej częstotliwością uderzeń pulsów w utworze, czyli tempem utworu. Inną możliwością wykrycia tempa w oparciu o filtr grzebieniowy jest wykorzystanie funkcji autokorelacji ACF (ang. *Autocorrelation Function*).[1, 4, 10] ACF można również wykorzystać do wykrycia pulsów w sygnale, by następnie obliczyć prawdopodobieństwo ich pojawienia się w czasie.[7, 8]

Innym sposobem wykrycia tempa jest stworzenie spektrogramu sygnału i szukaniu w nim podobnych wartości i częstotliwości ich występowania.[5]. Szukanie podobieństwa w macierzy powstałej ze spektrogramu przynosi dobre rezultaty do wykrywania metrum.[5, 10] Innym rozwiązaniem wykrywania metrum jest użycie ACF i tworzonych sygnałów z pulsami odpowiadającym akcentom w metrum.[12]

Posiłkując się powyższymi pracami zdecydowano na użycie algorytmu z [3] oraz [9] do wykrywania tempa. Wykrycie metrum zostało zaimplementowane w sposób podobny do [12], z wykorzystaniem DFT, ACF, funkcji splotu oraz filtra grzebieniowego.

2.5 Rozwiązania komercyjne

Na rynku dostępnych jest wiele komercyjnych programów do cyfrowej obróbki muzyki, takich jak m. in.: Logic Pro[1], Ableton Live[2], Virtual DJ[3], w których zostało zaimplementowane wykrywanie tempa utworu muzycznego. Taka funkcjonalność jest niezbędna do pracy z muzyką zarówno w studiu nagraniowym, jak i przy występach na żywo. Niestety autorzy nie upublicznili kodu swojego rozwiązania.

Istnieje również wiele wtyczek VST (ang. *Virtual Studio Technology*) np. Edison[4] i programów pomocniczych np.: BPM Analyzer[5] albo HoRNet SongKey MK3[6].

[1] `https://www.apple.com/logic-pro/`
[2] `https://www.ableton.com/`
[3] `https://www.virtualdj.com/`
[4] `https://www.image-line.com/plugins/Tools/Edison/`
[5] `https://mixmeister-bpm-analyzer.softonic.pl/`
[6] `https://www.hornetplugins.com/plugins/hornet-songkey-mk3/`

W internecie dostępna jest strona Get Song BPM[7], która umożliwia wykrycie tempa. Dostępny jest też program Aubio[8] na licencji "open-source", który między innymi pozwala na wykrycie tempa.

[7] `https://getsongbpm.com/tools/audio`
[8] `https://aubio.org/`

Rozdział 3

Wymagania i narzędzia

Wymagania określają docelową funkcjonalność i zachowanie tworzonego systemu. Są wymagane, żeby oprogramowanie można uznać za gotowe oraz tworzą ograniczenia, które muszą być spełnione przy testach jakościowych. Wyróżnia się dwa typu wymagań:

- funkcjonalne - dotyczą funkcjonalności systemu i wyników jego zachowania,
- niefunkcjonalne - dotyczą cech systemu i jak ma się zachować.

3.1 Wymagania funkcjonalne

W wymaganiach funkcjonalnych zostały zdefiniowane wymogi jakości wykrywania tempa oraz metrum.

1. Program ma wykrywać tempo i zwracać wynik w postaci liczby uderzeń na minutę BPM.

2. Dopuszczalny błąd wykrycia tempa to ± 2 uderzenie na minutę. Większe różnice powodują, że sygnał metronomu bardzo szybko desynchronizuje się względem utworu. Wynik można uznać za prawie poprawny, kiedy wykryte tempo jest dwukrotnie większe lub mniejsze niż prawdziwe, ponieważ w dalszym ciągu sygnał metronomu będzie zsynchronizowany z utworem.

3. Program ma wykrywać tempo w przedziale 60-240 BPM, ponieważ w takich tempach jest większość utworów. Utwory szybsze niż 240 BPM mogą zostać rozpoznane z dwa razy wolniejszym tempem.

4. Program ma wykrywać metrum utworu i zwracać wynik w postaci łańcucha znaków na przykład “4/4”.

5. Program ma mieć zaimplementowane wykrywania najpopularniejszych metrum w muzyce rozrywkowej: 4/4, 3/4, 6/8. Może rozpoznawać inne metra.

6. Plikiem wejściowym jest plik muzyczny w formacie .wav.

7. Program ma posiadać funkcję rysowania wykresów.

8. Miarą jakości i celności Acc wykrywania jest procentowy stosunek liczby dobrych wyników P do liczby wszystkich utworów w testowanym zbiorze N.

$$Acc = \frac{P}{N} \cdot 100\% \tag{3.1}$$

9. System uznaje się za dobry, kiedy jakość wykrycia jest równa lub większa 80%.

3.2 Wymagania niefunkcjonalne

System nie jest skierowany do klienta dlatego wymagania niefunkcjonalne nie narzucają żadnych ograniczeń.

1. Nie ma maksymalnego czasu w jakim program powinien zakończyć wykrywanie tempa oraz metrum.

2. Nie ma narzuconego języka programowania, w którym system ma powstać.

3. Program może zostać uruchomiony na dowolnym środowisku, ale zalecane jest, by działał na komputerze domowym.

3.3 Opis narzędzi

3.3.1 Język programowania

Program został napisany w języku Python 3.7 z wykorzystaniem środowiska Anaconda[1]. Python, jako język słabo typowany i z wysokim poziomem abstrakcji, jest idealny do przetwarzania sygnałów. Jest darmowy, posiada prostą składnię i wiele wyspecjalizowanych, darmowych modułów, jak na przykład scipy, tensorFlow czy numpy.

Anaconda to zbiór narzędzi i programów, które bardzo ułatwiają pracę z dużą liczbą danych, jak na przykład przetwarzanie obrazów, muzyki czy implementowanie sztucznej sieci neuronowej. Pomaga konfigurować interpretera Pythona wraz z dodatkowymi modułami poprzez graficzny interfejs użytkownika. Tak jak Python jest darmowa.

3.3.2 Środowisko programistyczne

Program powstawał równolegle w dwóch środowiskach. Pisanie kodu było najwygodniejsze w programie JetBrains PyCharm[2], ale na etapie szukania najlepszego rozwiązania i poprawiania błędów w programie, testowanie kodu było bardziej przejrzyste i szybsze w Jypyter Notebook wchodzącym w skład Anacondy.

PyCharm to zaawansowane środowisko, które natywnie wspiera Anacondę, a interfejs użytkownika jest przystosowany do wyświetlania dużych kolekcji danych oraz wykresów. Posiada funkcję podpowiadania składni i w czasie rzeczywistym sprawdza, czy używamy istniejących funkcji i metod, co jest niezmiernie pomocne w językach słabo typowanych. PyCharm jest środowiskiem bardzo rozbudowanym, które niestety wymaga dużo pamięci operacyjnej, by płynnie działać. Mankamentem pracy z Anacondą przy użyciu środowiska firmy JetBrains jest dłuższy czas wykonywania się napisanego programu, z powodu bycia nakładką na Jyptera.

Jupyter Notebook pozwala tworzyć kod w formie „zeszytów” i uruchamiać kod kawałek po kawałku. Takie rozwiązanie przyspiesza pracę nad dużymi zbiorami danych. W przypadku tej pracy było to duże ułatwienie, ponieważ na program

[1] `https://www.anaconda.com/`
[2] `https://www.jetbrains.com/pycharm/`

składa się kilka funkcji, które przetwarzają sygnał pochodzący z utworu. Wynik pierwszej funkcji jest argumentem drugiej, a wykonanie każdej trwa nawet kilkanaście sekund. Testując czwartą w kolejności funkcję, nie trzeba każdorazowo uruchamiać całego programu, a tylko dany fragment, ponieważ uzyskane wyniki są cały czas zapisane w pamięci komputera.

3.3.3 Kontrola wersji

Do kontroli wersji użyto darmowego oprogramowania Git. Z powodu jednoosobowej pracy, nie było potrzeby konfigurowania potoku ciągłej integracji.

3.3.4 Wykorzystane moduły Python

Do zaimplementowania wymaganej funkcjonalności użyto trzech darmowych bibliotek wchodzących w skład Anacondy:

1. Numpy[3] jest biblioteką do naukowego przetwarzania danych. Pozwala na operowaniu na wielowymiarowych tablicach obiektów oraz udostępnia wiele funkcji matematycznych jak transformata Fouriera.

2. Scipy[4] jest nakładką na Numpy, która udostępnia kilka nowych funkcji, które były niezbędne do realizacji tej pracy, takich jak tworzenie tablicy z pliku .wav czy funkcja splotu albo ACF.

3. Matplotlib[5] pozwala w łatwy sposób tworzyć wykresy i jest kompatybilny z tablicami numpy.

3.4 Metodyka pracy

Z powodu pracy jednoosobowej i szukania najlepszego rozwiązania wykrywania tempa oraz metrum, przyjęto kaskadową metodykę pracy.

[3] `https://numpy.org/`
[4] `https://docs.scipy.org/doc/scipy/reference/index.html`
[5] `https://matplotlib.org/`

1. Studium materiałów literaturowych i istniejących rozwiązań. Ta część pracy została przyspieszona dzięki serwisowi Google Scholar[6], który jest wyszukiwarką do prac naukowych. Po wpisaniu szukanej frazy, użytkownik dostaje odnośniki do pozycji, a najczęściej cytowane prace są oznaczane.
2. Zaimplementowanie wykrywania tempa.
3. Przetestowanie wykrywania tempa dla różnych gatunków muzycznych i ewentualna poprawa błędów oraz próba polepszenia jakości.
4. Zaimplementowanie wykrywania metrum.
5. Przetestowanie wykrywania metrum dla różnych gatunków muzycznych i ewentualna poprawa błędów oraz próba polepszenia jakości.

[6] `https://scholar.google.com/`

Rozdział 4

Specyfikacja zewnętrzna

4.1 Wymagania sprzętowe i programowe

Do uruchomienia programu potrzeba komputera domowego z zainstalowanym oprogramowaniem Anaconda[1]. Wszystkie użyte moduły wchodzą w skład podstawowej wersji Anacondy, dlatego nie potrzeba pobierania dodatkowych zależności.

4.2 Sposób aktywacji

Program uruchamia się za pomocą linii poleceń programu Anaconda. Przykładowe wywołanie programu:

```
python detector.py song.wav
```

4.2.1 Parametry

Program posiada jeden parametr obowiązkowy jakim jest ścieżka do pliku muzycznego w formacie .wav, który ma zostać sprawdzony pod względem tempa i metrum. Oprócz tego dostępne są przełączniki, które służą ustawienia sposobu wykrywania tempa oraz metrum oraz zmiany ustawień domyślnych.

W programie zostały zaimplementowane następujące przełączniki sterujące:

-t Wybór sposobu wykrywania tempa. Dostępne metody to:

[1] `https://www.anaconda.com/distribution/`

- użycie filtra grzebieniowego (5.7)

  `-t combFilterTempoDetector`

- użycie funkcji splotu (5.7.3)

  `-t convolveTempoDetector`

-m Wybór sposobu wykrywania metrum. Dostępne metody to:

- użycie filtra grzebieniowego (5.8.1)

  `-m combFilterMetreDetector`

- użycie filtra grzebieniowego ze znormalizowanym sygnałem (5.8.2)

  `-m combFilterNormalizedMetreDetector`

- użycie funkcji splotu (5.8.3)

  `-m convolveMetreDetector`

- użycie funkcji splotu ze znormalizowanym sygnałem (5.8.4)

  `-m convolveNormalizedMetreDetector`

- użycie ACF (5.8.5)

  `-m correlateNormalizedMetreDetector`

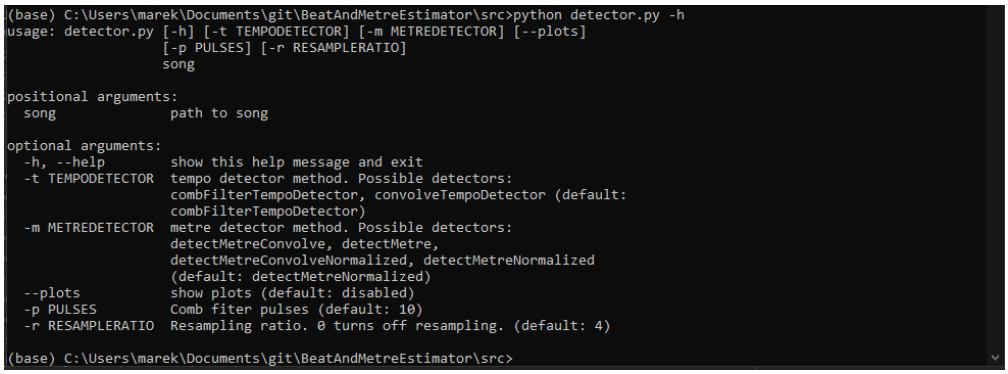
--plots Włączenie wyświetlania wykresów.

-p Liczba pulsów przy filtrze grzebieniowym. Jest to tożsame z długością próbki w sekundach. Domyślna wartość to 10.

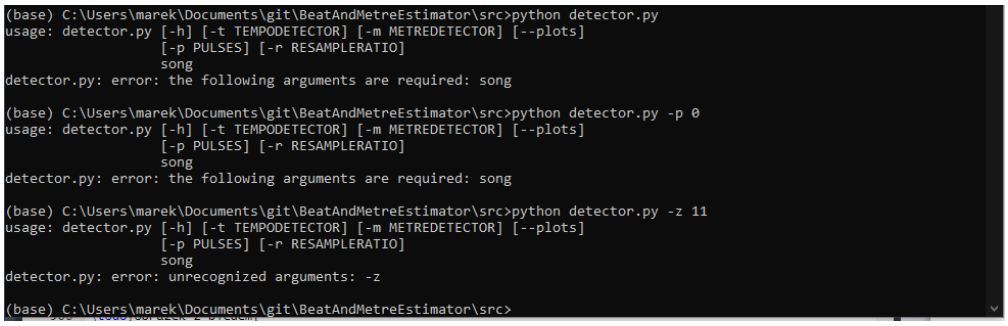
`-p 12`

-r Współczynnik zmiany częstotliwości próbkowania. Domyślna wartość to 4.

`-r 2`



Rysunek 4.1: Wywołanie programu z zamiarem uzyskania pomocy.



Rysunek 4.2: Przykłady błędnego wywołania programu.

4.3 Pomoc i obsługa błędów

Po uruchomieniu programu z przełącznikiem −h lub −−**help** wyświetlana jest pomoc i objaśnienie każdego parametru (Rys. 4.1).

```
python detector.py -h
```

W razie podania błędnego parametru lub nie podania ścieżki do piosenki wyświetlany odpowiedni komunikat z błędem i sposób aktywacji. (Rys. 4.2)

- przykład działania
- scenariusze korzystania z systemu (ilustrowane zrzutami z ekranu lub generowanymi dokumentami)

Rozdział 5

Specyfikacja wewnętrzna

Jak wspomniano w rozdziale 3 na algorytm działania programu składa się kilka uruchamianych po sobie i zależnych od siebie funkcji. W tym rozdziale zostanie wytłumaczony sposób ich działania, a także zaproponowane zostaną inne rozwiązania.

Algorytm wykrywania tempa i metrum można zapisać w następujących punktach:

1. Wczytanie pliku .wav do tablicy.
2. Wycięcie fragmentu ze środka utworu.
3. Wyrównanie fragmentu do pierwszego znalezionego uderzenia pulsu.
4. Transformacja sygnału do dziedziny częstotliwości i podzielenie na podpasma przy użyciu zespołu filtrów.
5. Odwrotna transformacja sygnału z podpasm do dziedziny czasu i wygładzenie za pomocą okna czasowego.
6. Obliczenie pochodnej sygnału.
7. Wykrycie tempa.
8. Wykrycie metrum.
9. Zwrócenie wyniku.

5.1 Wczytanie pliku .wav

Format .wav (ang. *wave form audio format*) jest standardowym, niekompresowanym i bezstratnym formatem plików dźwiękowych bazującym na formacie RIFF (ang. *Resource Interchange File Format*). Oprócz informacji o strumieniu audio, pliki .wav zawierają również informacje o użytym kodeku, częstotliwości próbkowania oraz liczbie kanałów. Z perspektywy opisanego w tej pracy programu, istotne są tylko informacje o strumieniu i próbkowaniu.

Na rysunku 5.1 zaprezentowano funkcję do wczytywania plików .wav. Biblioteka scipy.io.wavfile ma zaimplementowaną funkcję do wczytywania plików, która oprócz tablicy ze strumieniem audio zwraca również częstotliwość próbkowania.

```
1 def read_song(filename):
2     sample_freq, data = scipy.io.wavfile.read(filename)
3     signal = np.frombuffer(data, np.int16)
4     return signal, sample_freq
```

Rysunek 5.1: Funkcja read_song wczytująca pliki .wav.

5.2 Wycięcie fragmentu ze środka utworu

Nierzadko utwory muzyczne trwają po kilka, a nawet kilkanaście minut. Przetwarzanie takiej ilości danych byłoby bardzo czasochłonne i niepotrzebne, dlatego też wycinany jest krótki fragment. Jego długość zależy od liczby pulsów filtra grzebieniowego oraz minimalnego tempa jakie chcemy wykryć. Kod odpowiedzialny za wycinanie został zaprezentowany na rysunku 5.2.

Fragment jest wycinany ze środka, ponieważ na początku utworu często możemy usłyszeć różnego rodzaju wprowadzenia w postaci ciszy, szumów czy narastającej dynamiki, które mogą sfałszować wynik. Oczywiście w środku, również mogą pojawić się różnego rodzaju przejścia, które nie będą w tempie albo zmiany tempa oraz metrum, jednak ten aspekt został pominięty w tej pracy.

Czas trwania fragmentu utworu będącego wynikiem działania opisanej funkcji jest czterokrotnie dłuższy od wymaganego, ze względu na to, że w kolejnych kro-

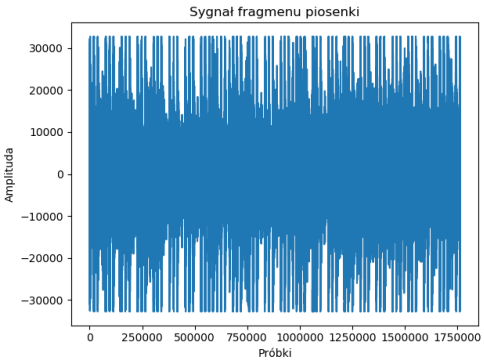
kach przetwarzania zostanie on zredukowany po odpowiednim wyrównaniu. Taki nadmiar jest spowodowany, gdyby w środku utworu był cichszy fragment, w którym wykrycie tempa i metrum byłoby trudniejsze.

```
signal, samplingFrequency = songReader.read_song(song.
    filepath)

sample_length = settings.combFilterPulses *
    samplingFrequency
seconds = sample_length * 4
song_length = signal.size

start = int(np.floor(song_length / 2 - seconds / 2))
stop = int(np.floor(song_length / 2 + seconds / 2))
if start < 0:
    start = 0
if stop > song_length:
    stop = song_length

sample = signal[start:stop]
```

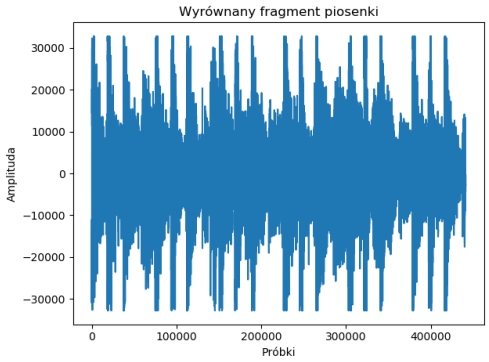
Rysunek 5.2: Wycinanie fragmentu ze środka utworu.

5.3 Wyrównanie fragmentu do pierwszego napotkanego pulsu

Efekt funkcji opisanej w sekcji 5.2 został przedstawiony na rysunku 5.3. Można zauważyć, że pierwsze uderzenie pulsu nie znajduje się na początku sygnału, co może spowodować przekłamany wynik wykrycia tempa i metrum. Z tego powodu oraz chęci przedstawienia bardziej przejrzystych wykresów, szukana jest pierwsza próbka, której wartość byłaby równa co najmniej 90% maksimum globalnego w wyciętym fragmencie. Kod funkcji odpowiedzialnej za wyrównanie został przedstawiony na rysunku 5.5.



Rysunek 5.3: Wycięty fragment utworu.



Rysunek 5.4: Fragment wyrównany do pierwszego napotkanego pulsu i zredukowany do pożądanej długości.

```
1  def __center_sample_to_beat(self, signal, required_length):
2      n = len(signal)
3      index = 0
4
5      max = np.max(abs(signal))
6
7      for i in range(0, n):
8          if abs(signal[i]) > max * 0.9:
9              index = i
10             break
11
12     lastindex = required_length
13     lastindex += index
14     if lastindex > n:
15         lastindex = n
16     if lastindex - index < required_length:
17         index = index - (required_length - (lastindex -
               index))
18
19     return signal[index:int(lastindex)]
```

Rysunek 5.5: Funkcja `__center_sample_to_beat` odpowiedzialna za wyrównanie fragmentu do pierwszego napotkanego uderzenia pulsu.

5.4 Transformacja sygnału do dziedziny częstotliwości

Przygotowany fragment sygnału, który zwraca funkcja opisana w sekcji 5.3, został przedstawiony na rysunku 5.4. Można na nic zaobserwować wyraźne regularne maksima, jednak szukanie tempa i metrum na takiej próbce może być niedokładne, z powodu różnych szumów wysokoczęstotliwościowych oraz faktu, że instrumenty melodyczne, nie zawsze grają akcenty, które wynikają z metrum.

Często w skład zespołu muzycznego, a zwłaszcza w muzyce rozrywkowej, wchodzi sekcja rytmiczna, której głównym zadaniem jest nadanie i trzymanie rytmu. Instrumentami rytmicznymi są między innymi perkusja i gitara basowa (ew. podobny instrument, na przykład kontrabas), a grane na nich dźwięki mają niskie częstotliwości. Z tego powodu, chcąc znaleźć rytm, próbka jest przenoszona do dziedziny częstotliwości z wykorzystaniem szybkiej transformaty Fouriera FFT (ang. *Fast Fourier Transform*), a następnie dzielona na następujące podpasma w Hz: <0;200>, <200;400>, <400;800>, <800;1600>, <1600;3200>, <3200;$\frac{f_s}{2}$). Kod odpowiedzialny za transformatę został przedstawiony na rysunku 5.6, a efekt podzielenia na zespół filtrów na rysunku 5.7a oraz 5.7b.

5.5 Wygładzenie za pomocą okna czasowego

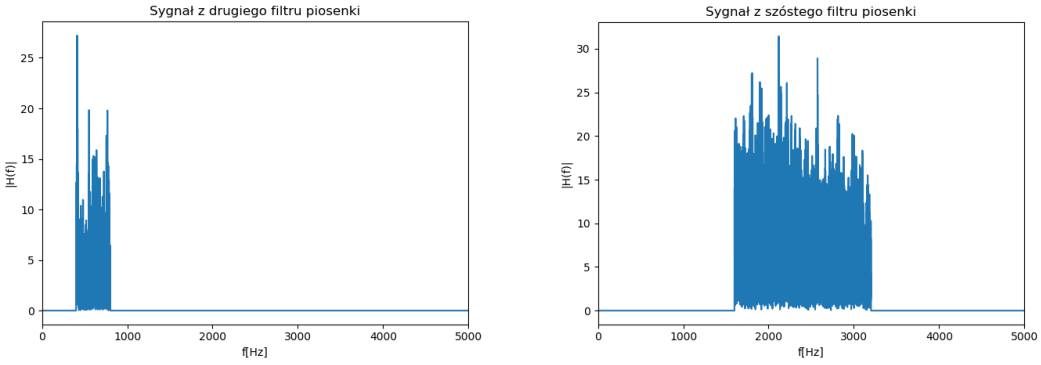
Sygnał podzielony na podpasma częstotliwości zostaje następnie z powrotem przeniesiony do dziedziny czasu i wygładzony za pomocą okna czasowego Hanna. Sygnał jest wygładzany, żeby móc wykryć nagłe zmiany. Funkcja odpowiedzialna za to została przedstawiona na rysunku 5.8, a przykładowy rezultat na rysunku 5.9.

5.6 Obliczenie pochodnej

Na rysunku 5.9 został przedstawiony przebieg sygnału w konkretnym podpaśmie, jednak poszukiwane szczyty sygnału są niewyraźne. Żeby uwydatnić maksima, zostaje stworzona tablica, w której zapisywane są różnice między wartościami

```
1  def __prepare_filterbanks(self, signal, bandlimits,
       samplingFrequency):
2      dft = np.fft.fft(signal)
3      n = len(dft)
4      nbands = len(bandlimits)
5      bl = np.zeros(nbands, int)
6      br = np.zeros(nbands, int)
7
8      #   % Bring band scale from Hz to the points in our
           vectors
9      for band in range(0, nbands - 1):
10         bl[band] = np.floor(bandlimits[band] /
               samplingFrequency * n / 2) + 1
11         br[band] = np.floor(bandlimits[band + 1] /
               samplingFrequency * n / 2)
12
13     bl[0] = 0
14     bl[nbands - 1] = np.floor(bandlimits[nbands - 1] /
           samplingFrequency * n / 2) + 1
15     br[nbands - 1] = np.floor(n / 2)
16
17     output = np.zeros([nbands, n], dtype=complex)
18
19     for band in range(0, nbands):
20         for hz in range(bl[band], br[band]):
21             output[band, hz] = dft[hz]
22         for hz in range(n - br[band], n - bl[band]):
23             output[band, hz] = dft[hz]
24
25     output[1, 1] = 0
26     return output
```

Rysunek 5.6: Funkcja __prepare__filterbanks odpowiedzialna za transformatę do dziedziny częstotliwości i podzielenie na zespół filtrów.



(a) Widmo sygnału w zakresie <400;800> Hz.

(b) Widmo sygnału w zakresie <1600;3200> Hz.

kolejnych próbek. Na rysunku 5.10 została przedstawiona pochodna sygnału. Wyraźnie na niej widać regularne uderzenia pulsu.

5.7 Wykrycie tempa za pomocą filtra grzebieniowego

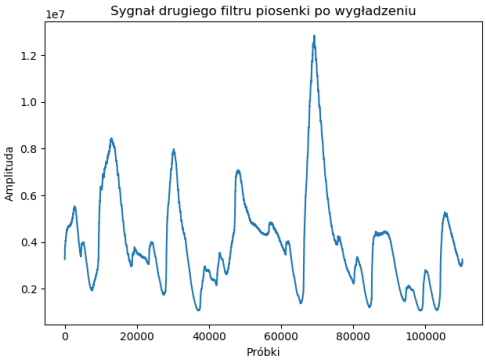
Funkcja detect_tempo odpowiedzialna za wykrycie tempa z wykorzystaniem filtra grzebieniowego została umieszczona w źródłach dołączonych do pracy.

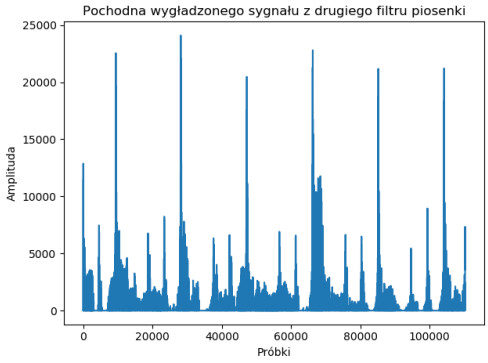
Funkcja posiada 7 argumentów. Pierwszym jest dwuwymiarowa tablica różnic obliczona w poprzedniej funkcji opisanej w sekcji 5.6. Kolejnymi argumentami są precyzja, minimalne i maksymalne tempo, które ma zostać wykryte. Precyzja definiuje krok w pętli, która oblicza maksymalną energię iloczynu sygnału i filtra. Ważnym argumentem, mającym wpływ na działanie funkcji, jest combFilterPulses określającym liczbę pulsów w filtrze grzebieniowym. Liczba pulsów wpływa na długość wycinanego fragmentu utworu, ponieważ jest dobierana tak, by zmieściły się wszystkie pulsy dla minimalnego tempa.

Na początku sygnał różnicowy jest przenoszony do dziedziny częstotliwości. Transformaty sygnału wejściowego będą wielokrotnie wykorzystywane podczas działania funkcji, dlatego są obliczane na początku. Następne obliczenia mają miejsce w pętli, gdzie iteratorem jest tempo w BPM. Zakres tej pętli od minimalnego do maksymalnego tempa z krokiem podanym w argumencie accuracy.

```
1  def __hann(self, signal, winLength, bandslimits,
       samplingFrequency):
2      n = len(signal[0])
3      nbands = len(bandslimits)
4      hannlen = winLength * 2 * samplingFrequency
5      hann = np.zeros(n)
6      wave = np.zeros([nbands, n], dtype=complex)
7      output = np.zeros([nbands, n], dtype=complex)
8      freq = np.zeros([nbands, n], dtype=complex)
9      filtered = np.zeros([nbands, n], dtype=complex)
10
11     for a in range(1, int(hannlen)):
12         hann[a] = (np.cos(a * np.pi / hannlen / 2)) ** 2
13
14     for band in range(0, nbands):
15         wave[band] = np.real(np.fft.ifft(signal[band]))
16
17     for band in range(0, nbands):
18         for j in range(0, n):
19             if wave[band, j] < 0:
20                 wave[band, j] = -wave[band, j]
21         freq[band] = np.fft.fft(wave[band])
22
23     for band in range(0, nbands):
24         filtered[band] = freq[band] * np.fft.fft(hann)
25         output[band] = np.real(np.fft.ifft(filtered[band]))
26
27     return output
```

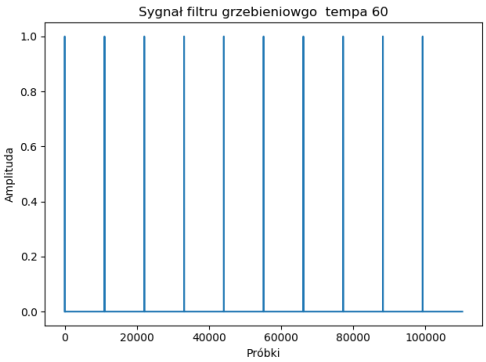
Rysunek 5.8: Funkcja hann odpowiedzialna za transformatę do dziedziny czasu i wyrównanie.



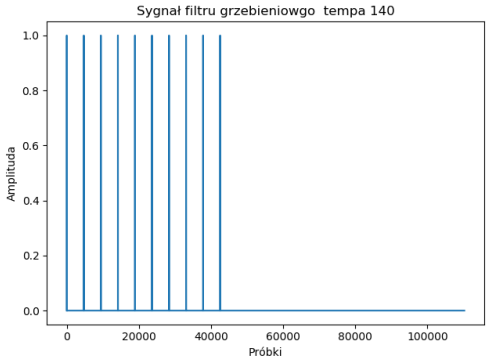


Rysunek 5.9: Wygładzony sygnał w za-kresie <200;400> Hz w dziedzinie czasu.

Rysunek 5.10: Pochodna sygnału w za-kresie <200;400> Hz.



Rysunek 5.11: Sygnał filtra dla BPM=60.



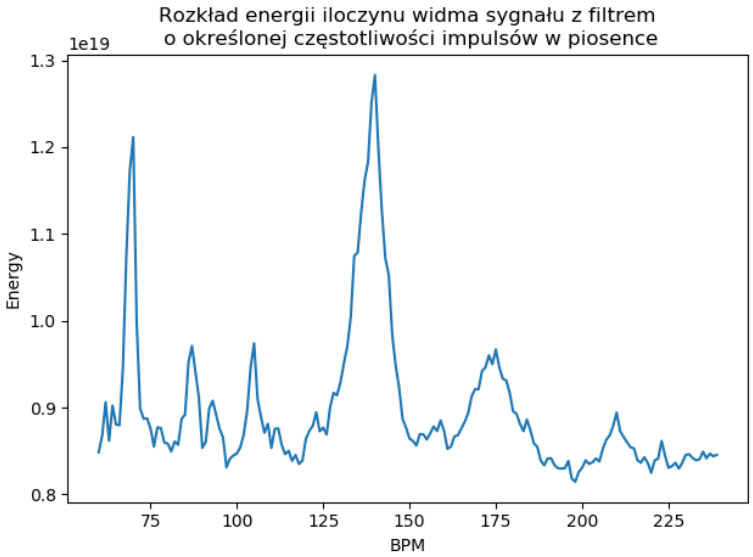
Rysunek 5.12: Sygnał filtra dla BPM=140.

Dla danego tempa jest tworzony sygnał filtru grzebieniowego. Odległość impulsów jest obliczana ze wzoru:

$$step = \frac{60}{bpm} * f_s \tag{5.1}$$

Długość filtra jest stała dla wszystkich wartości BPM i jest zależna od minimalnej wartości tempa, które ma zostać wykryte oraz ilości impulsów filtra. Przykładowe sygnały zostały zaprezentowane na rysunku 5.11 oraz 5.12.

Następnie sygnał filtra jest przenoszony do dziedziny częstotliwości za pomocą szybkiej transformaty Fouriera (ang. *Fast Fourier Transform*). Na rysunku 5.15 widać, że z powodu stałej długości filtra oraz stałej ilości impulsów, w sygnale transformaty pojawiło się wiele szumów w porównaniu do sygnału widocznym



Rysunek 5.13: Rozkład energii iloczynu widma sygnału z filtrem o określonej częstotliwości impulsów.

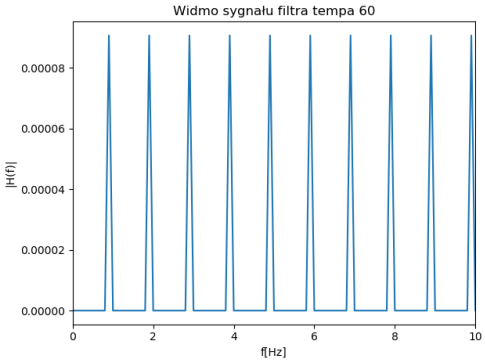
na rysunku 5.14. Pomimo tych szumów ta metoda okazała się być najszybsza i najdokładniejsza. Bardziej poprawne, z perspektywy przetwarzania sygnałów, metody zostały zaprezentowane w 5.7.1.

Obliczona transformata jest kolejno przemnażana przez transformaty pochodnych. W ten obliczana jest suma poszczególnych energii ilorazów. Podczas przebiegu pętli szukane jest maksimum sum energii. Tempo odpowiadające maksymalnej sumie energii uznawane jest za tempo utworu.

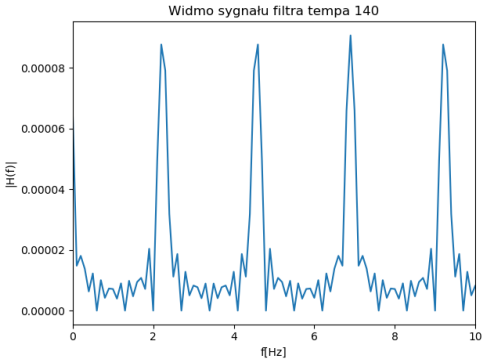
Na rysunku 5.13 zaprezentowano rozkład sum energii dla tempa. Widać, że tempo równe 140 ma największą sumę i takie jest też tempo podanego utworu. Druga największa wartość energii odpowiada tempu 70, ponieważ sygnał filtra pasuje w co drugie uderzenie pulsu utworu.

5.7.1 Alternatywne implementacje wykrywania

Wygląd transformaty filtra dla tempa różnego niż minimalne tempo (rysunek 5.15) może budzić obawy, że wynik funkcji może być niedokładny. Z tego powodu, zostały sprawdzone dwa alternatywne sposoby, które mimo poprawności działania (z perspektywy przetwarzania sygnałów) okazały się gorsze pod względem czasu



Rysunek 5.14: Widmo filtra dla tempa równeg 60.



Rysunek 5.15: Widmo filtra dla tempa równego 140.

wykonania, jak i samego rezultatu.

5.7.2 Normalizacja sygnału transformaty

Pierwszym sposobem, który miał wyczyścić transformatę filtra, było stworzenie sygnału filtra grzebieniowego o wymaganej długości L. Odległość między pulsami filtra znamy ze wzoru 5.1.

$$L = step * pulses \tag{5.2}$$

Następnie przeróbkowanie sygnału z zespołu filtrów do długości filtra grzebieniowego L, przemnożenie transformat sygnałów i obliczenie energii. Obliczona w ten sposób energia została normalizowana poprzez podzielenie przez długość filtra L. Niestety mimo próby normalizacji, maksymalna energia odpowiadała najdłuższemu filtrowi, czyli najmniejszemu tempu, więc ten sposób został odrzucony.

5.7.3 Liczenie splotu sygnałów

Drugim sposobem było stworzenie filtra grzebieniowego z użyciem dwóch impulsów. Następnie obliczany jest splot sygnału różnicowego i filtra. W kolejnym kroku wyliczana jest transformata sygnału uzyskanego przez splot i obliczana jej energia. Energie z wszystkich podpasm są sumowane i szukana jest maksymalna suma i odpowiadające jej tempo.

Ten sposób trwał zdecydowanie dłużej i koniecznie było przepróbkowanie sy-

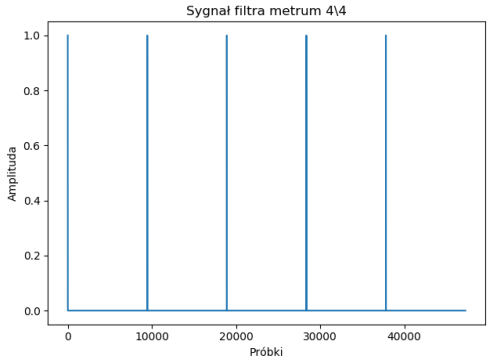
gnału z utowru, żeby skrócić czas wykonania. Kod funkcji wykorzystującej splot został umieszczony w załącznikach w rozdziale 7.3.

5.8 Wykrycie metrum

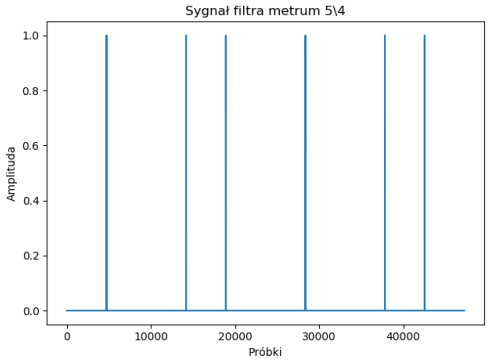
Wykrywanie metrum okazało się być pojęciem dużo trudniejszym w implementacji. Z tego powodu powstało parę metod, a uzyskane wyniki zostały opisane w sekcji 7.1.2.

5.8.1 Wykrycie metrum za pomocą filtra grzebieniowego

Wykrywanie metrum odbywa się w podobny sposób co wykrywanie tempa (5.7), z tą różnicą, że tempo utworu jest już znane i na jego podstawie, można stworzyć filtry odpowiadające różnym metrum (rysunki 5.16 i 5.17). Analogicznie do funkcji wykrywającej tempo, ta funkcja tworzy filtry dla założonych metrum i szuka maksimum sum energii iloczynów transformat filtra i sygnału.



Rysunek 5.16: Sygnał filtra metrum 4/4.



Rysunek 5.17: Sygnał filtra metrum 5/4.

5.8.2 Wykrycie metrum za pomocą filtra grzebieniowego ze znormalizowanym sygnałem

Metoda opisana w sekcji 5.8.1 jest niedokładna, ponieważ uzyskane transformaty filtrów, mają różną energię. Do normalizacji sygnału z zastosowano obliczenie

wartości pulsów, tak by ich suma była równa 1 oraz podzielenie sygnału transformaty filtra przez jego energię.

5.8.3 Wykrycie metrum za pomocą splotu

Innym sposobem na wykrycie metrum jest metoda podobna do tej opisanej w sekcji 5.7.3. Tworzony jest jeden takt, w którym zaznaczone są tylko akcenty. Następnie obliczany jest splot sygnału różnicowego z wcześniej stworzonym sygnałem dla każdego metrum i obliczana energia splotu. Energie są sumowane, a metrum dla którego suma jest największa jest metrum utworu.

5.8.4 Wykrycie metrum za pomocą splotu ze znormalizowanym sygnałem

Ta metoda jest połączeniem 5.8.2 oraz 5.8.3. Sygnał dla taktu jest tworzony w ten sposób, by jego suma była równa 1.

5.8.5 Wykrycie metrum za pomocą funkcji autokorelacji

W tej metodzie sygnał metrum jest taki sam jak w 5.8.4, ale zamiast splotu, wykorzystywana jest ACF.

5.9 Optymalizacja

Mimo iż w wymaganiach niefunkcjonalnych nie było narzuconego maksymalnego czasu wykonania programu (sekcja 3.2), poczyniono kroki, by przyspieszyć działanie programu.

5.9.1 Zmniejszenie częstości próbkowania

Tak jak zostało to wspomniane w sekcji 5.7.3, sygnał z utworu muzycznego musiał zostać przepróbkowany. Z związku z zastosowaniem filtrów w zakresie 0-3200 Hz i zakładając, że częstotliwość próbkowania oryginalnego pliku wynosi 44,1

kHz, możemy próbkowanie nawet 4 krotnie zmniejszyć, a sygnał w zakładanych częstotliwościach nie ulegnie zdeformowaniu.

Zmniejszenie częstości próbkowania było wymagane, żeby liczenie splotu mogło się odbyć w skończonym czasie, ale również przyspieszyło działanie programu, nie wpływając negatywnie na uzyskiwane rezultaty.

5.9.2 Rekurencyjne wykrywanie tempa

Funkcja bpm_comb_filter opisana w 5.7 jest wywoływana dwa razy dla różnych wartości argumentów accuracy,minBpm,maxBpm. Za pierwszym razem przedział jest szeroki 60-240 BPM, ale z dużym krokiem równym 5. Pozwala to mniej więcej wykryć prawdziwe tempo utworu, ponieważ większość utworów ma tempo podzielne przez 10. Natomiast, gdyby tempo utworu nie było liczbą podzielną przez 5, funkcja jest wywoływana drugi raz, tym razem z parametrem accuracy równym 1, a przedział to tempo wykryte przy pierwszym wywołaniu ± 5 BPM. Pozwala to na dokładne wykrycie tempa w krótszym czasie, ponieważ stworzenie filtra, obliczenie jego transformaty i przemnożenie trwa nawet parę sekund.

5.9.3 Zastosowanie architektury CUDA

Użyte typy danych(wielowymiarowe tablice liczb całkowitych i zmiennoprzecinkowych) oraz algorytmy wydają się być idealne do wykonania równoległego z wykorzystaniem kart graficznych i architektury CUDA. Biblioteka Cupy jest nakładką na bibliotekę “Numpy”, która udostępnia część funkcjonalności Numpy, ale z wykorzystaniem GPU (ang. *graphics processing unit*). Przeniesienie obliczeń z procesora na kartę graficzną prawdopodobnie znacząco przyspieszyłoby obliczenia, niestety nie wszystkie funkcje zostały przeniesione do Cupy, a kodu opisanego w tym rozdziale ne udało się uruchomić z pomocą tej biblioteki.

Rozdział 6

Weryfikacja i walidacja

6.1 Zbiór testowy

Utwory muzyczne, użyte do testowania, pochodzą z konferencji ISMIR z roku 2004[2]. Oryginalna baza posiada 2187 utworów w formacie .mp3, a informacje o tempie oraz metrum nie są podane. W celu skompletowania zbioru testowego wybrano 34 utwory (tabela 6.1), przekonwertowano do formatu .wav, a następnie empirycznie oszacowano tempo oraz metrum, posiłkując się metronomem i wiedzą z teorii muzyki. Dodatkowo dodano dwa pliki dźwiękowe jako gatunek "test". Są to sygnały metronomu w podanym tempie oraz metrum.

6.2 Organizacja testów

Stworzono 21 przypadków testów z różnymi metodami wykrycia tempa i metrum oraz zmienną liczbą pulsów. Test polegał na wykryciu tempa, a następnie wykryciu metrum, wykorzystując wykryte tempo do stworzenia przebiegu filtrów metrum. Przypadki oraz uzyskane wyniki zostały przedstawione w tabeli 6.2. Tabela 6.3 zawiera wyniki wykrywania tempa, uznając wynik za poprawny, gdy wykryte tempo jest dwa raz mniejsze niż rzeczywiste. Celność jest obliczona ze wzoru 3.1.

Wszystkie testy zostały przeprowadzone z pięciokrotnie zmniejszoną częstotliwością próbkowania w celu przyspieszenia czasu wykonania.

Tablica 6.1: Baza utworów testowych.

Utwór	gatunek	Tempo	Metrum
7-don't_look_back	rock	89	4/4
8-i'll_pretend	rock	140	4/4
10-at_least_you've_been_told	rock	130	4/4
1-don't_rain_on_my_parade	rock	160	4/4
4-goodbye_on_a_beautiful_day	rock	60	4/4
5-apartment_a	rock	115	4/4
12-rotgut	rock	147	4/4
4-buffalo_nights	rock	147	8/8
6-didn't_i	rock	85	4/4
4-patterns	rock	106	6/8
1-prospects	jazz	116	4/4
18-slide_boogie	jazz	120	4/4
6-invisible_pants	jazz	118	4/4
7-heavenly_rain	metal	126	4/4
1-by_mourning	metal	202	4/4
5-kc_rip_off	metal	140	4/4
5-code_red_love_is_dead	metal	120	4/4
7-serenade_for_samantha	metal	120	4/4
1-sweet_dissonance	world	120	4/4
6-life_glides	world	120	4/4
1-for_a_good_day_o_host_to_you	world	77	4/4
5-bliss	world	68	4/4
10-the_last_faery	world	90	4/4
1-odysseia	world	93	4/4
28-la_victoire_suite_in_d_from_l	classical	134	4/4
18-handel_oxford_water_music_suit	classical	167	3/4
22-handel_oxford_water_music_suit	classical	150	4/4
10-sonata_iii_moderato	classical	152	4/4
19-sonata_v_allegro	classical	140	4/4
6-click_full_pussy	electronic	160	4/4
1-blue-tinted_sunglasses	electronic	123	4/4
1-sunset_(endless_night_journey	electronic	140	4/4
7-northern_lights	electronic	140	4/4
9-nibtal_7	electronic	140	4/4
120	test	120	4/4
100	test	100	3/4

Tablica 6.2: Uzyskane rezultaty w testach.

Liczba pulsów	Metoda wykrycia tempa	Celność	Metoda Wykrycia metrum	Celność
16	Filtr grzebieniowy	30.56%	ACF	30.56%
12[1]	Filtr grzebieniowy	19.44%	ACF	25%
12	Filtr grzebieniowy	19.44%	ACF	25%
12	Filtr grzebieniowy	19.44%	Splot (znormalizowany)	19.44%
12	Filtr grzebieniowy	19.44%	Filtr grzebieniowy	8.33%
12	Filtr grzebieniowy	19.44%	Filtr grzebieniowy (znormalizowany)	5.56%
12	Filtr grzebieniowy	19.44%	Splot	0%
4	Filtr grzebieniowy	16.67%	Filtr grzebieniowy	38.89%
2	Filtr grzebieniowy	16.67%	Filtr grzebieniowy	25%
8	Filtr grzebieniowy	16.67%	Splot (znormalizowany)	16.67%
8	Filtr grzebieniowy	16.67%	Filtr grzebieniowy	13.89%
8	Filtr grzebieniowy	16.67%	Filtr grzebieniowy (znormalizowany)	8.33%
8	Filtr grzebieniowy	16.67%	ACF	8.33%
4	Filtr grzebieniowy	16.67%	Splot (znormalizowany)	5.56%
4	Filtr grzebieniowy	16.67%	ACF	2.78%
4	Filtr grzebieniowy	16.67%	Filtr grzebieniowy (znormalizowany)	2.78%
4	Filtr grzebieniowy	16.67%	Splot	0%
8	Filtr grzebieniowy	16.67%	Splot	0%
4	Splot	5.56%	Filtr grzebieniowy	19.44%
8	Splot	5.56%	Filtr grzebieniowy	11.11%
1	Filtr grzebieniowy	2.78%	Filtr grzebieniowy	88.89%

Tablica 6.3: Uzyskane rezultaty, jeżeli dwukrotnie mniejsze tempo jest poprawne.

Liczba pulsów	Metoda wykrycia tempa	Celność
12	Filtr grzebieniowy	63.89%
16	Filtr grzebieniowy	50.56%
4	Filtr grzebieniowy	50%
8	Filtr grzebieniowy	50%
4	Splot	50%
8	Splot	44.44%
2	Filtr grzebieniowy	30.56%
1	Filtr grzebieniowy	25%

Dodatkowo przeprowadzono jeszcze 15 testów wykrywania metrum dla różnych metod wykrycia i parametrów. W ramach testów wykrycia metrum, do stworzenia przebiegu filtrów zostało wykorzystane rzeczywiste tempo utworu. Przypadki i uzyskane wyniki zostały zaprezentowane w tabeli 6.4.

Tablica 6.4: Uzyskane rezultaty wykrycia metrum, przy dobrze rozpoznanym tempie.

Liczba pulsów	Metoda wykrycia metrum	Celność
12	Splot (znormalizowany)	36.11%
8	Filtr grzebieniowy	36.11%
8	Splot (znormalizowany)	30.56%
12	ACF	27.78%
4	Splot (znormalizowany)	22.22%
8	ACF	19.44%
8	Filtr grzebieniowy (znormalizowany)	16.67%
12	Filtr grzebieniowy	13.89%
4	Filtr grzebieniowy	13.89%
12	Filtr grzebieniowy (znormalizowany)	11.11%
4	ACF	11.11%
4	Filtr grzebieniowy (znormalizowany)	2.78%
12	Splot	0%
8	Splot	0%
4	Splot	0%

Rozdział 7

Podsumowanie i wnioski

7.1 Analiza wyników

Uzyskane wyniki są gorsze niż się spodziewano. Niewątpliwie zbiór testowy jest jednym z problemów, ponieważ jest stosunkowo mało liczny, a oszacowane tempo i metrum, może nie zgadzać się ze stanem faktycznym. Dodatkowo utwory z metrum 4/4 stanowią znaczącą większość.

7.1.1 Analiza wyników wykrycia tempa

Wyniki jednoznacznie pokazują, że użycie funkcji splotu do wykrycia tempa nie jest najlepszym rozwiązaniem. Jeśli patrzeć tylko na dokładnie wykryte tempo to ta metoda jest najgorsza, bez względu na długość fragmentu utworu. Zaskakująca jest celność, jeśli za poprawne uznamy dwukrotnie mniejsze tempo. Wówczas dokładność wykrycia jest zbliżona to metody wykorzystującej filtr grzebieniowy.

Widać wzrost celności wykrycia tempa w metodzie wykorzystującej filtr grzebieniowy wraz ze wzrostem liczby pulsów filtra. Jednak jakość rośnie asymptotycznie i przy 16 impulsach dokładny wynik jest stały, ale gorzej wykrywa tempa dwa razy mniejsze.

Uzyskane wyniki wykrycia tempa na poziomie 30% i 63% są niezadowalające, ale pokazują że metoda ma potencjał, tylko potrzebuje rozwinięcia. Dobrym pomysłem może się okazać połączenie filtra grzebieniowego z funkcją z ACF. Innym rozwiązaniem może być podzielenie sygnału na zespół filtrów przy użyciu FIR i

nadanie wag podpasmom albo stworzenie sygnału do przetwarzania z obwiedni sygnału utworu.

7.1.2 Analiza wyników wykrycia metrum

Uzyskane wyniki wykrywania metrum są niezadowalające i dużo gorsze niż się spodziewano. 88.89% przy metodzie wykorzystującej filtr grzebieniowy jest uznana za błąd. Funkcja dla każdego utworu wskazała metrum 4/4. Wysoki wynik wynika ze znaczącej przewagi utworów w metrum 4/4 w zbiorze testowym.

W wykrywaniu metrum, najlepszą metodą okazało się być wykorzystanie splotu z sygnałem znormalizowanym oraz filtra grzebieniowego z celnością na poziomie 36.11%.

Ciekawym wynikiem jest zerowa celność metody wykorzystującej splot z sygnałem nieznormalizowanym. W tym przypadku również długość przetwarzanego fragmentu utworu nie wpłynęła na liczbę poprawnych wykryć. Ta metoda prawie wszystkim utworom przypisywała metrum 6/8.

Z uzyskanych rezultatów wynika, że celność wykrycia, zarówno tempa jak i metrum, rośnie wraz z wydłużeniem przetwarzania fragmentu utworu muzycznego. Jednak jest to problem, ponieważ wraz z wydłużeniem sygnału, wydłuża się czas obliczania splotu i ACF. Dla metody wykorzystującej filtr grzebieniowy narzut czasu nie jest taki długi.

7.2 Dalsze prace

Do celów rozwojowych niezbędne będzie rozwinięcie bazy utworów testowych. Potrzeba więcej utworów, gdzie informacje o tempie i metrum będą pewne i dokładne. Szczególnie ma to znaczenie, jeżeli w przyszłości zaimplementowana zostanie metoda, wykorzystująca nauczanie maszynowe.

Przydatną funkcjonalnością będzie możliwość obsługi skompresowanych plików muzycznych. Skompresowane pliki zajmują mniej miejsca na dysku twardym komputera. Użyta baza utworów[2] posiada utwory wyłącznie w formacie .mp3, które trzeba było uprzednio przekonwertować. Wszystkie 2187 skompresowanych utworów waży około 2,5 GB. Gdyby wszystkie przekonwertowano i przycięto do około

minuty, jak zostało to zrobione z utworami ze zbioru testowego, to waga zwiększyłaby się ok. 9 krotnie.

Zastąpienie prostokątnego filtra, filtrem o skończonej odpowiedzi impulsowej FIR, może przynieść lepsze rezultaty. Szczególnie może to pomóc przy wykrywaniu metrum, ponieważ dla każdego z podpasm zostanie stworzony inny przebieg filtru, który bardziej będzie odpowiadał sygnałowi dla danego przedziału częstotliwości. Na przykład w metrum 4/4 częstotliwości dużego bębna w zestawie perkusyjnym granym na 1 i 3 są duże niższe niż werbla granego na 2 i 4. Ta różnica mogłaby pomóc określić początek taktu, szczególnie w metrum nieregularnym.

Dodatkowo można uzyskać funkcję z obwiedni sygnału oraz z jego spektrogramu. Ta metoda może pomóc przy przetwarzaniu muzyki bez sekcji rytmicznej.

Wykrycie tempa wymaga polepszenia dokładności, szczególnie że jest to ważne do poprawnego wykrycia metrum. Połączenie kilku rozwiązań może dać lepszy efekt.

Wykrycie metrum musi zostać zaimplementowane całkowicie od nowa. Detekcja za pomocą ACF, wydaje się być chybionym pomysłem, ale przed całkowitym odrzuceniem, należy sprawdzić jakie rezultaty przyniesie zmiana dzielenia sygnału na podpasma oraz inne przebiegi sygnałów dla danego metrum. Dużo lepszym rozwiązaniem może okazać się utworzenie macierzy ze spektrogramu i szukanie w niej podobieństwa taktów.

Żeby przyspieszyć wykonanie programu, zakładając, że w przyszłości wykonywanie będzie więcej obliczeń, niezbędne będzie stworzenie programu działającego współbieżnie.

7.3 Podsumowanie

Zrealizowany temat pracy w znaczący sposób poszerzył wiedzę autora w dziedzinie przetwarzania sygnałów.

Pisanie pracy w słabo typowanym Pythonie przysporzyło parę problemów. Po pierwsze nie jest to język pisany z myślą o paradygmacie obiektowym. Nie udostępnia funkcjonalności tworzenia i implementacji interfejsów, znanej z takich języków jak Java czy C. Było to niewygodne przy zmianie nazwy metody albo jednego z jej parametrów. Należało pamiętać, by to samo zrobić w klasach realizujących tę

samą funkcjonalność. Po drugie nie posiada mechanizmu enkapsulacji pół i metod obiektów.

Uzyskane wyniki niestety nie są zadowalające. Jednak rozbudowanie programu o założenia z sekcji 7.2 nie powinno być czasochłonne. Zaimplementowanie metod wykrywania przy użyciu wstrzykiwania zależności i wzorca projektowego w postaci strategii, ułatwi zaprogramowanie nowej funkcjonalności. Dużo trudniejszym zadaniem, wydaje się być skompletowanie zbioru testowego.

Bibliografia

[1] Miguel Alonso, Bertrand David, Gaël Richard. Tempo and beat estimation of musical signals. *ENST-GET, Département TSI*, 2004.

[2] P. Cano, E. Gómez, F. Gouyon, P. Herrera, M. Koppenberger, B. Ong, X. Serra, S. Streich, N. Wack. Ismir 2004 audio description contest, 2004.

[3] Kileen Cheng, Bobak Nazer, Jyoti Uppuluri, Ryan Verret. Beat this - a beat synchronization project. `https://www.clear.rice.edu/elec301/Projects01/beat_sync/beatalgo.html`. [data dostępu: 2018-09-30].

[4] M. Gainza, E. Coyle. Tempo detection using a hybrid multiband approach. *IEEE Transactions on Audio, Speech, and Language Processing*, 19(1):57–68, 2011.

[5] Mikel Gainza. Automatic musical meter detection. *2009 IEEE International Conference on Acoustics, Speech and Signal Processing*, strony 329–332, 2009.

[6] Andrzej Leśnicki. *Technika Cyfrowego Przetwarzania Sygnałów*. Wydawnictwo Politechniki Gdańskiej, 2014.

[7] Lie Lu, D. Liu, Hong-Jiang Zhang. Automatic mood detection and tracking of music audio signals. *IEEE Transactions on Audio, Speech, and Language Processing*, 14(1):5–18, 2006.

[8] Geoffroy Peeters. Time variable tempo detection and beat marking. *IRCAM - Analysis/Synthesis Team*, 2005.

[9] Eric D. Scheirer. Tempo and beat analysis of acoustic musical signals. *The Journal of the Acoustical Society of America*, 103(1):588–601, 1998.

[10] Björn Schuller, Florian Eyben, Gerhard Rigoll. Tango or waltz?: Putting ballroom dance style into tempo detection. *EURASIP Journal on Audio, Speech, and Music Processing*, 8(1), 2008.

[11] Bożena Smagowska, Małgorzata Pawlaczyk-Łuszczyńska. Effects of ultrasonic noise on the human body—a bibliographic review. *International journal of occupational safety and ergonomics : JOSE*, 19:195–202, 2013.

[12] Petri Toiviainen, Tuomas Eerola. Classification of musical metre with autocorrelation and discriminant functions. strony 351–357, 2005.

Dodatki

Spis skrótów i symboli

ACF funkcja autokorelacji (ang. *Autocorellation Function*)

BPM Uderzenia na minutę (ang. *beats per minute*)

DFT Dyskretna transformata Fouriera (ang. *Discrete Fourier Transform*)

f_s częstotliwość próbkowania

FFT Szybka transformata Fouriera (ang. *Fast Fourier Transform*)

FIR filtr o skończonej odpowiedzi impulsowej (ang. *finite impulse response*)

GPU karta graficzna (ang. *graphics processing unit*)

IIR filtr o nieskończonej odpowiedzi impulsowej (ang. *infinite impulse response*)

RIFF format plików przeznaczony do przechowywania danych multimedialnych (ang. *Resource Interchange File Format*)

VST standard wtyczek efektowych oraz wirtualnych instrumentów (ang. *Virtual Studio Technology*)

WAV format plików dźwiękowych (ang. *wave form audio format*

Źródła

```
1  def detect_tempo(self, signal, accuracy: int, minBpm: int,
       maxBpm: int, bandsLimits, samplingFrequency,
2                   combFilterPulses, plotDictionary):
3      n = len(signal[0])
4      bands_amount = len(bandsLimits)
5      dft = np.zeros([bands_amount, n], dtype=complex)
6
7      if minBpm < 60:
8          minBpm = 60
9
10     if maxBpm > 240:
11         maxBpm = 240
12
13     for band in range(0, bands_amount):
14         dft[band] = np.fft.fft(signal[band])
15         plots.draw_fft_plot(settings.drawFftPlots, dft[band
               ], f"Band[{band}] DFT", samplingFrequency)
16
17     maxEnergy = 0
18     for bpm in range(minBpm, maxBpm, accuracy):
19         this_bpm_energy = 0
20         fil = np.zeros(n)
21
```

```
22          filter_step = np.floor(60 / bpm * samplingFrequency
                )
23          percent_done = 100 * (bpm - minBpm) / (maxBpm -
                minBpm)
24          print("%.2f" % percent_done, "%")
25
26          for a in range(0, combFilterPulses):
27              fil[a * int(filter_step) + 1] = 1
28
29          plots.draw_plot(settings.drawCombFilterPlots, fil,
                f"Timecomb bpm: {bpm}", "Sample/Time", "
                Amplitude")
30          dftfil = np.fft.fft(fil)
31          plots.draw_comb_filter_fft_plot(settings.
                drawFftPlots, dftfil, f"Filter's signal DFT {bpm
                }",
32                                          samplingFrequency)
33
34          for band in range(0, bands_amount):
35              x = (abs(dftfil * dft[band])) ** 2
36              this_bpm_energy = this_bpm_energy + sum(x)
37
38          plotDictionary[bpm] = this_bpm_energy
39          if this_bpm_energy > maxEnergy:
40              songBpm = bpm
41              maxEnergy = this_bpm_energy
42
43      return songBpm
```

```
1 def bpm_comb_filter_convolve(signal, accuracy, minBpm,
      maxBpm, bandlimits, maxFreq, plot_dictionary):
2       n = len(signal[0])
```

```
3       nbands = len(bandlimits)
4       dft = np.zeros([nbands, n], dtype=complex)
5
6       if minBpm < 60:
7           minBpm = 60
8
9       if maxBpm > 240:
10          maxBpm = 240
11
12      for band in range(0, nbands):
13          dft[band] = np.fft.fft(signal[band])
14          draw_fft_plot(drawFftPlots, dft[band], f"Band[{band
                }] DFT", maxFreq)
15
16      maxe = 0
17      for bpm in range(minBpm, maxBpm, accuracy):
18          e = 0
19
20          filterLength = 2
21          nstep = np.floor(60 / bpm * maxFreq)
22          percent_done = 100 * (bpm - minBpm) / (maxBpm -
                minBpm)
23          fil = np.zeros(int(filterLength * nstep))
24
25          print(percent_done)
26
27          for a in range(0, filterLength):
28              fil[a * int(nstep)] = 1
29
30          draw_plot(drawCombFilterPlots, fil, f"Timecomb bpm:
                {bpm}", "Sample/Time", "Amplitude")
31
32          dftfil = np.fft.fft(fil)
```

```
33
34          draw_comb_filter_fft_plot(drawCombFilterPlots,
                dftfil, f"Signal DFT {bpm}", maxFreq)
35          for band in range(0, nbands-1):
36              filt = scipy.convolve(signal[band], fil)
37              f_filt = abs(np.fft.fft(filt))
38              draw_fft_plot(drawFftPlots, f_filt, f"Signal
                    DFT {bpm}", maxFreq)
39
40              x = abs(f_filt)**2
41              e = e + sum(x)
42
43
44          plot_dictionary[bpm] = e
45          if e > maxe:
46              sbpm = bpm
47              maxe = e
48
49      return sbpm
```

Zawartość dołączonej płyty

Do pracy dołączona jest płyta CD z następującą zawartością:

- praca (źródła LaTeXowe i końcowa wersja w `pdf`),
- źródła programu,
- dane testowe.
- uzyskane wyniki

Spis rysunków

2.1 Reprezentacja graficzna 10 sekund nagranego utworu techno 7

4.1 Wywołanie programu z zamiarem uzyskania pomocy. 19
4.2 Przykłady błędnego wywołania programu. 19

5.1 Funkcja read_song wczytująca pliki .wav. 22
5.2 Wycinanie fragmentu ze środka utworu. 23
5.3 Wycięty fragment utworu. 24
5.4 Fragment wyrównany do pierwszego napotkanego pulsu i zredukowany do pożądanej długości. 24
5.5 Funkcja ___center_sample_to_beat odpowiedzialna za wyrównanie fragmentu do pierwszego napotkanego uderzenia pulsu. 24
5.6 Funkcja ___prepare___filterbanks odpowiedzialna za transformatę do dziedziny częstotliwości i podzielenie na zespół filtrów. 26
5.8 Funkcja hann odpowiedzialna za transformatę do dziedziny czasu i wyrównanie. 28
5.9 Wygładzony sygnał w zakresie <200;400> Hz w dziedzinie czasu. . 29
5.10 Pochodna sygnału w zakresie <200;400> Hz. 29
5.11 Sygnał filtra dla BPM=60. 29
5.12 Sygnał filtra dla BPM=140. 29
5.13 Rozkład energii iloczynu widma sygnału z filtrem o określonej częstotliwości impulsów. 30
5.14 Widmo filtra dla tempa równeg 60. 31
5.15 Widmo filtra dla tempa równego 140. 31
5.16 Sygnał filtra metrum 4/4. 32

5.17 Sygnał filtra metrum 5/4. 32

Spis tablic

6.1 Baza utworów testowych. 36
6.2 Uzyskane rezultaty w testach. 37
6.3 Uzyskane rezultaty, jeżeli dwukrotnie mniejsze tempo jest poprawne. 37
6.4 Uzyskane rezultaty wykrycia metrum, przy dobrze rozpoznanym tempie. 38