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 10 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 10 stycznia 2020

.....

(podpis promotora)

Spis treści

Rozdział 1

Wstęp

Wykrywanie tempa i metrum utworu muzycznego jest podstawowym zagadnieniem potrzebnym do cyfrowej obróbki muzyki. Ma to zastosowanie zarówno dla producentów muzycznych, gdzie 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 scharakteryzowania, by następnie stworzyć bazę utworów i móc na niej tworzyć zapytania dla użytkownika na podstawie jego humoru albo przeznaczenia playlisty, 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 jej szukać wśród żywych utworów z prostym, dwudzielnym metrum, na przykład 4/4.

Celem tej pracy inżynierskiej jest napisanie programu, który potrafi wykryć tempo i metrum utworu muzycznego.

W rozdziale 2 zostały opisane takie zagadnienia jak tempo oraz metrum, pozycje w bibliografii i znane rozwiązania komercyjne. Rozdział 3 opisuje wymagania funkcjonalne oraz niefunkcjonalne, użyte narzędzia, środowiska i technologie 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 wraz z fragmentami kodu.

Rozdział 2

Analiza tematu

2.1 Tempo i metrum

Tempo utworu określa, jak szybko dany utwór ma zostać wykonany. W zapisie nutowym istnieją dwa sposoby zapisu tempa. Pierwszy sposób to umowne wyrazy zaczerpnięte z języka włoskiego np.: moderato (pl. *umiarkowanie* ok. 90 BPM), allegro (pl. *żywo* ok. 120 BPM). Drugim sposobem 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.

2.2 Częstotliwość próbkowania

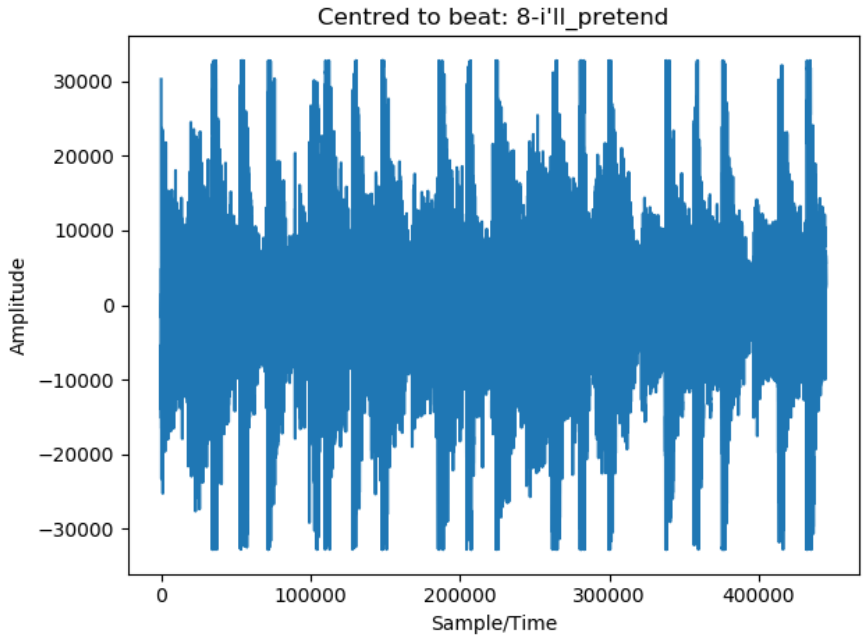
Wykrywanie tempa i metrum utworu zaczęto dynamicznie rozwijać na początku lat 90., kiedy to domowy sprzęt komputerowy pozwolił na zapis nagranej muzyki w pamięci komputera, a następnie jej przetwarzanie w czasie rzeczywistym.

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

$$f_s > 2 \cdot f_p$$

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, standardową częstotliwością próbkowania dla plików audio zostało 44.1 kHz przy rozmiarze 16 bitów na próbkę. Zatem nieskompresowany trzy minutowy 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 2.1 przedstawia reprezentacje graficzną nagranego utworu. Jest to wykres zmian amplitudy natężenia sygnału w czasie. Wyraźnie widać skoki natężenia sygnału. W miejscach występowania tych maksimów lokalnych najprawdopodobniej znajdują się uderzenia pulsu utworu, a częstotliwość ich występowania będzie korelować z tempem oraz metrum utworu.



Rysunek 2.1: Reprezentacja graficzna 10 sekund nagranego utworu rockowego

2.3 Studia literaturowe

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

W [8] tworzony jest spektrogram dla wycinka utworu, z którego za pomocą okna czasowego tworzona jest funkcja przebiegu. W uzyskanym przebiegu za pomocą dyskretnej transformaty Fouriera DFT (ang. *Discrete Fourier Transform*), funkcji autokorelacji ACF (ang. *Autocorellation Function*) oraz algorytmu Viterbiego, szukane jest tempo oraz metrum utworu. Podobne rozwiązanie zostało opisane w [1], ale do wykrycia tempa i metrum wykorzystano kilka różnych znanych algorytmów.

W [10] najpierw za pomocą filtru grzebieniowego szukane jest tempo wśród przebiegu dla różnych częstotliwości, a następnie znając tempo i posługując się maszyną wektorów nośnych SVM (ang. *Support Vector Machine*) szukamy podobieństwa w spektrogramie, by wykryć tempo. Podobne podejście zostało zaimplementowane w [5], gdzie przed podzieleniem spektrogramu na macierze, tempo jest szukane za pomocą filtrów grzebieniowych i ACF.

W [7] opisane szersze zagadnienie jakim jest wykrywanie nastroju utworu. Tempo utworu niewątpliwe wpływa na jego nastrój, dlatego też ten aspekt został poruszony. Podobnie jak w [10, 5] tempo jest wykrywane za pomocą filtra grzebieniowego oraz ACF.

W [4] utwór jest dzielony na 3 podpasma częstotliwości przy wykorzystaniu DFT. W przebiegach dla uzyskanych podpasm jest wykrywane tempo za pomocą ACF i filtra grzebieniowego. Ta metoda różni się od [10, 5] tym, że dla każdej z 3 wykrytych wartości jest przydzielana różna waga, by na końcu wybrać najbardziej prawdopodobną wartość.

Autorzy [6] opisali algorytm wykrywania tempa w 3 przedziałach: wolne, średnie oraz szybkie. Przynosi to dobre rezultaty, jednak celem mojej pracy jest dokładne wykrycie tempa w BPM.

Praca [12] opisuje wykrywanie metrum w utworze z wykorzystaniem ACF oraz zdefiniowanych przebiegów dla różnych metrum.

Autorzy [13] podeszli do tematu w bardziej nowatorski sposób, ponieważ nie wykrywają metrum a rytm(rumba, salsa), wykorzystując do tego sieć neuronową.

W [9] opisany jest algorytm, który jest najbliższy do tego, jak człowiek odbiera rytm w muzyce. Utwór jest najpierw dzielony na przebiegi dla różnych częstotliwości przy pomocy DFT, a następnie przy pomocy filtra grzebieniowego szuka się tempa odpowiadającego dla utworu. Na tej pracy opiera się algorytm opisany w [3].

Posiłkując się powyższymi pracami zdecydowano na użycie algorytmu z [9] oraz [3] do wykrywania tempa. Wykrywanie zostało zaimplementowany w podobny sposób, szukając najlepszego rozwiązania, z wykorzystaniem DFT, ACF oraz filtra grzebieniowego.

2.4 Znane rozwiązania

Na rynku dostępne jest wiele rozwiązań komercyjnych, których algorytm nie został upubliczniony. Programy do edycji muzyki takie jak m.in.: “Logic Pro”, “Ableton Live”, “Virtual DJ”, “FL Studio” posiadają taką funkcjonalność wbudowaną w podstawowy produkt, ponieważ jest ona niezbędna do pracy z muzyką zarówno w studiu nagraniowym, jak i przy występach na żywo. Istnieje również

wiele wtyczek VST (ang. *Virtual Studio Technology*) i programów pomocniczych np.: “BPM Analyzer” albo “HoRNet SongKey MK3”. Istnieje również rozwiązanie na licencji “open-source” jakim jest “Aubio”.

Rozdział 3

Wymagania i narzędzia

3.1 Wymaganie funkcjonalne

1. Program ma wykrywać tempo i zwracać wynik w postaci liczby uderzeń na minutę BPM. Dopuszczalny błąd to ± 1 uderzenie na minutę.

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

3. Plikiem wejściowym jest plik muzyczny w formacie .wav o długości co najmniej 12 sekund.

3.2 Wymaganie niefunkcjonalne

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

3.3 Opis narzędzi

3.3.1 Język programowania

Program został napisany w języku Python 3.7 z wykorzystaniem środowiska Anaconda. Python, jako język słabo typowany i z wysokim poziomem abstrakcji,

jest idealny do przetwarzania sygnałów. Ma 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 w krótkim czasie, jak na przykład przetwarzanie obrazów, muzyki czy implementowanie sztucznej sieci neuronowej. Pomaga konfigurować interpretera Pythona wraz z dodatkowymi modułami poprzez graficzny interfejs użytkownika.

3.3.2 Środowisko programistyczne

Program powstawał równolegle w dwóch środowiskach. Pisanie kodu było najwygodniejsze w programie “JetBrains PyCharm” ale jego testowanie było bardziej przejrzyste w “Jypyter Notebook” wchodzącym w skład Anacondy.

PyCharm to potężne 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.

Jupyter Notebbok pozwala tworzyć “zeszyty” 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 składa się kilka funkcji, gdzie wynik pierwszej jest wejściem do drugiej, a każde wykonanie trwa nawet parę sekund. Testując czwartą w kolejności funkcję, nie trzeba każdorazowo uruchamiać całego programu, a tylko dany fragment, ponieważ wynik funkcji trzeciej jest cały czas zapisany w pamięci komputera.

3.3.3 Kontrola wersji

Do kontroli wersji użyto darmowego oprogramowania Git. Z powodu jednoosobowego zespołu nie było potrzeby konfigurowania potoku ciągłej integracji.

3.3.4 Wykorzystane moduły Python

Do zaimplementowania programu użyto trzech darmowych bibliotek.

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

2. Scipy swoją funkcjonalnością jest zbliżony do Numpy, z którego sam z resztą korzysta, ale ma parę funkcji, których brakowało, jak tworzenie tablicy z pliku .wav czy liczenie splotu sygnału.

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

3.4 Metodyka pracy

Praca na programem została podzielona na następujące etapy:

1. Studium materiałów literaturowych i istniejących rozwiązań.

2. Zaimplementowanie wykrywania tempa.

3. Przetestowanie wykrywania tempa dla różnych gatunków i refaktoryzacja.

4. Zaimplementowanie wykrywania metrum.

5. Przetestowanie wykrywania metrum dla różnych gatunków i refaktoryzacja.

Taka prosta metodyka kaskadowa jest wystarczające dla małego projektu i jednoosobowego zespołu.

Rozdział 4

Specyfikacja zewnętrzna

4.1 Wymagania sprzętowe i programowe

Do uruchomienia programu potrzeba komputera z zainstalowanym oprogramowaniem Anaconda. Wszystkie użyte moduły wchodzą w skład podstawowej biblioteki, więc 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:

- użycie filtra grzebieniowego (5.7)
- użycie splotu sygnałów (5.7.3)

-m Wybór sposobu wykrywania metrum. Dostępne metody to:

- użycie filtra grzebieniowego (5.8.1)
- użycie filtra grzebieniowego ze znormalizowanym sygnałem (5.8.2)
- użycie splotu sygnałów (5.8.3)
- użycie splotu sygnałów ze znormalizowanym sygnałem (5.8.4)
- użycie ACF (5.8.5)

--plots Włączenie wyświetlania wykresów.

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

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

metrum

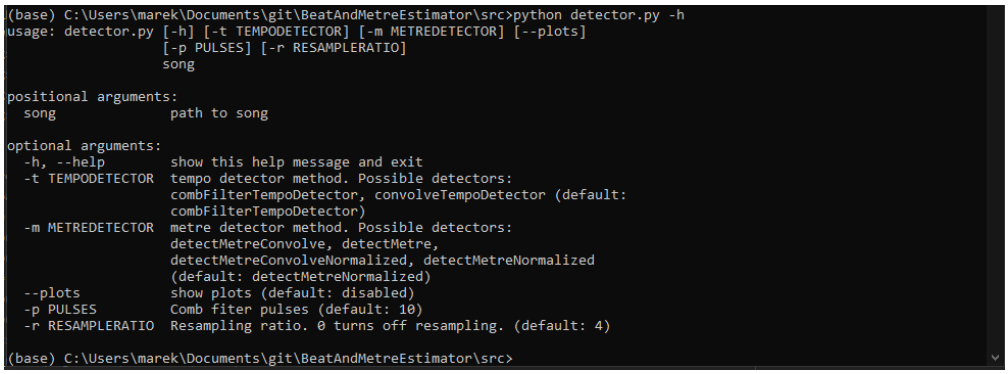
4.3 Pomoc i obsługa błędów

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

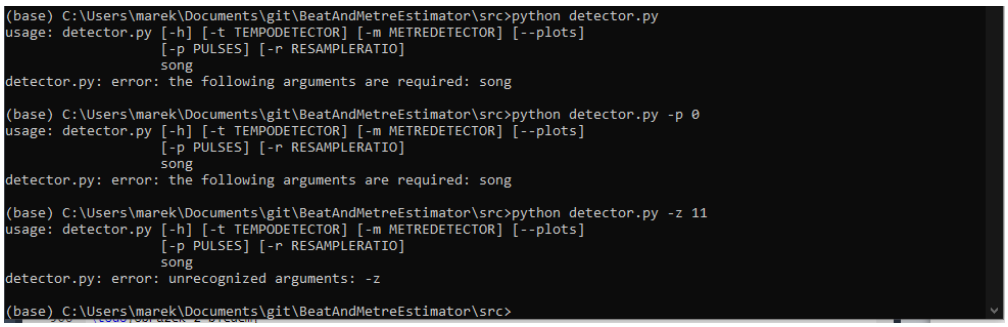
```
python detector.py -h
```

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

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



Rysunek 4.1: Wywołanie programu z zamiarem uzyskania pomocy.



Rysunek 4.2: Przykłady błędnego wywołania programu.

Rozdział 5

Specyfikacja wewnętrzna

Jak wspomniano w 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. Transformata sygnału do dziedziny częstotliwości i podzielenie na zespół filtrów.
5. Odwrotna transformata filtrów do dziedziny czasu i wygładzenie za pomocą okna czasowego.
6. Obliczenie sygnału różnicowego.
7. Wykrycie tempa.
8. Wykrycie metrum.
9. Zwrócenie wyniku.

```
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.

5.1 Wczytanie pliku .wav

Format .wav (ang. *wave form audio format* jest standardowym, niekompresowanym i bezstratnym formatem plików dźwiękowych. 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.

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 brany 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, jednak ten aspekt został pominięty w tej pracy.

W tym miejscu wycinany jest fragment czterokrotnie większy niż wymagany, ponieważ w następnym kroku opisanym w 5.3 zostanie on wyrównany do pierwszego napotkanego uderzenia pulsu i przycięty do porządnej długości. Zostało tutaj

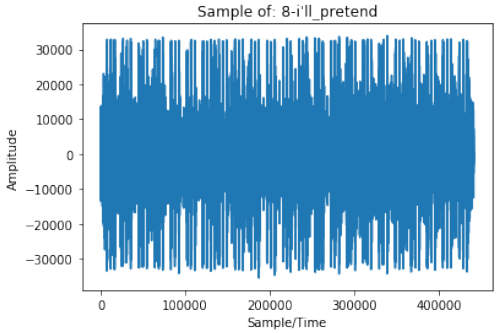
```
signal, samplingFrequency = read_song(song.filepath)

if(resampleSignal):
    signal = scipy.signal.resample(signal, int(len(signal)/
        resampleRatio))
    samplingFrequency /= resampleRatio

sample_length = 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 fragmnetu ze środka utworu.

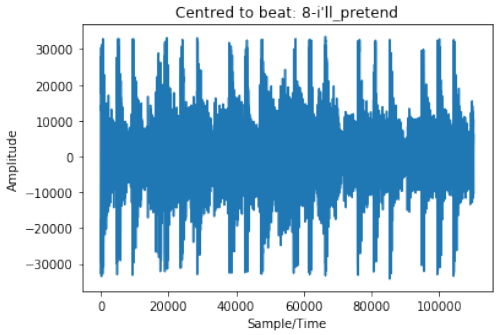
także zaimplementowane przepróbkowanie sygnału w celu optymalizacji czasu wykonania.

5.3 Wyrównanie fragmentu do 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 tablicy, co może spowodować przekłamany wynik wykrycia tempa i metrum. Z tego powodu szukana jest pierwsza próbka, której wartość byłaby równa co najmniej 90% maksimum globalnego. Kod funkcji odpowiedzialnej za wyrównanie został przedstawiony na rysunku 5.5.



Rysunek 5.3: Wycięty fragment utworu.



Rysunek 5.4: Fragment wyrównany do pulsu.

```
1  def center_sample_to_beat(signal, seconds):
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 = seconds
13     lastindex += index
14     if lastindex > n:
15         lastindex = n
16     return signal[index:int(lastindex)]
```

Rysunek 5.5: Funkcja center_sample_to_beat odpowiedzialna za wyrównanie fragmentu do pierwszego pulsu.

5.4 Transformata sygnału do dziedziny częstotliwości

Przygotowana próbka, którą zwraca funkcja opisana w 5.3, została przedstawiona na rysunku 5.4. Widać na nim wyraźne regularne maksima, jednak szukanie pulsu na takiej próbce może być niedokładne, z powodu różnych szumów, które mogą się pojawić w wyższych częstotliwościach.

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 za pomocą FFT, a następne 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 5.7 oraz 5.8.

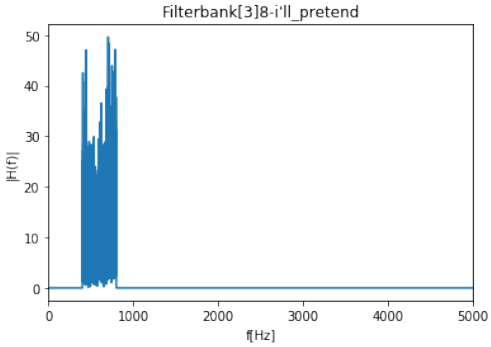
Zastosowany tutaj filtr prostokątny ma ostre granice i może zniekształcać sygnał. Lepszym rozwiązaniem może okazać się użycie filtru o skończonej odpowiedzi impulsowej FIR (ang. *finite impulse response filter*), który często jest wykorzystany we współczesnych rozwiązaniach audio i z jego wykorzystaniem stworzenie zespołu filtrów (ang. *filterbank*) jak w [4], jednak w tej pracy nie zostało to zaimplementowane.

5.5 Wygładzenie za pomocą okna czasowego

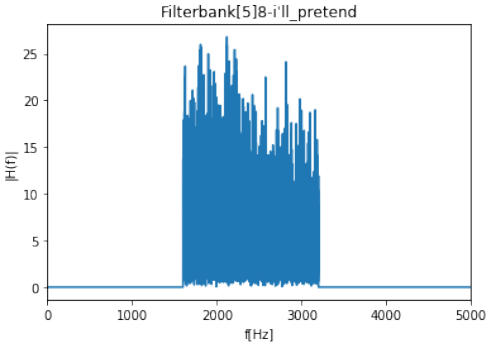
Sygnał podzielony na podpasma częstotliwości zostaje następnie z powrotem przeniesiony do dziedziny czasu i wygładzony za pomocą okna czasowego. Funkcja odpowiedzialna za to została przedstawiona na rysunku 5.9, a przykładowy rezultat na rysunku 5.10.

```
1  def filterbank(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      for band in range(0, nbands - 1):
9          bl[band] = np.floor(bandlimits[band] /
               samplingFrequency * n / 2) + 1
10         br[band] = np.floor(bandlimits[band + 1] /
               samplingFrequency * n / 2)
11
12     bl[0] = 0
13     bl[nbands - 1] = np.floor(bandlimits[nbands - 1] /
           samplingFrequency * n / 2) + 1
14     br[nbands - 1] = np.floor(n / 2)
15
16     output = np.zeros([nbands, n], dtype=complex)
17
18     for band in range(0, nbands):
19         for hz in range(bl[band], br[band]):
20             output[band, hz] = dft[hz]
21         for hz in range(n - br[band], n - bl[band]):
22             output[band, hz] = dft[hz]
23
24     output[1, 1] = 0
25     return output
```

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



Rysunek 5.7: Natężenie w zakresie <400;800> Hz.



Rysunek 5.8: Natężenie w zakresie <1600;3200> Hz.

5.6 Obliczenie sygnału różnicowego

Na rysunku 5.10 został przedstawiony przebieg sygnału w konkretnym podpaśmie, jednak poszukiwane szczyty sygnału są niewyraźne. Żeby uwydatnić maksima, zostaje stworzona tablica, w której zapisywane są różnice wartości próbki z poprzednią. Na rysunku 5.11 został przedstawiony sygnał różnicowy. Wyraźnie na niej widać regularne uderzenia pulsu.

5.7 Wykrycie tempa za pomocą filtra grzebieniowego

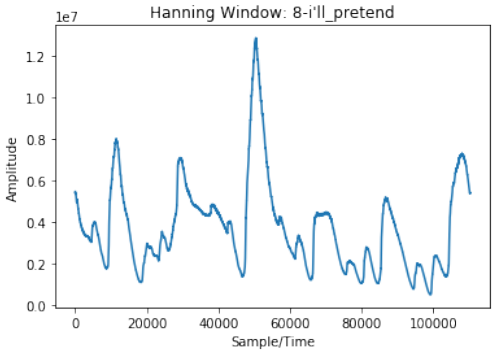
Funkcja bpm_comb_filter odpowiedzialna za wykrycie tempa z wykorzystaniem filtra grzebieniowego została umieszczona na końcu pracy w 7.

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 energie ilorazu 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.

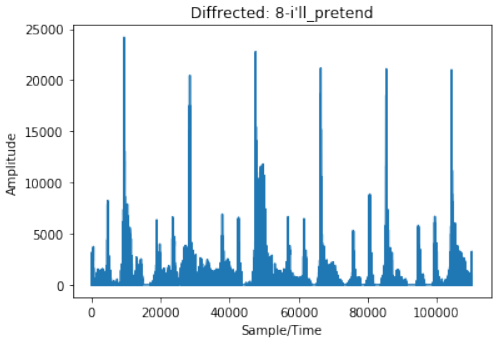
Na początku sygnał różnicowy jest przenoszony do dziedziny częstotliwości.

```
1  def hann(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
```

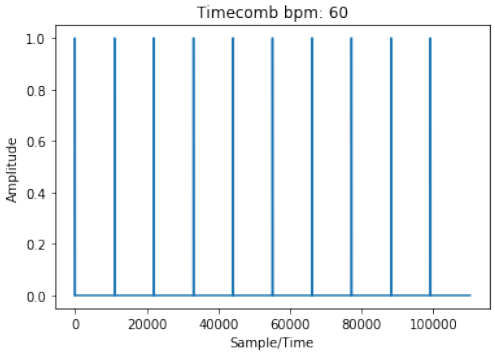
Rysunek 5.9: Funkcja hann odpowiedzialna za transformatę do dziedziny czasu i wyrównanie.



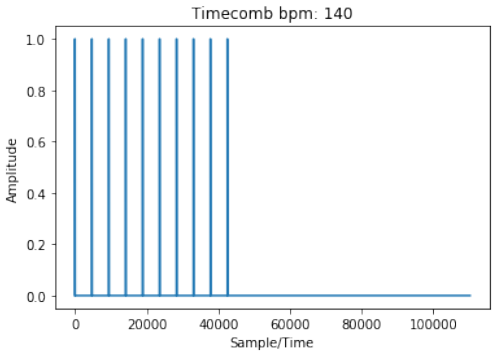
Rysunek 5.10: Wygładzony sygnał w zakresie <200;400> Hz w dziedzinie czasu.



Rysunek 5.11: Różnice próbek sygnału w zakresie <200;400> Hz.



Rysunek 5.12: Sygnał filtra dla BPM=60.



Rysunek 5.13: Sygnał filtra dla BPM=140.

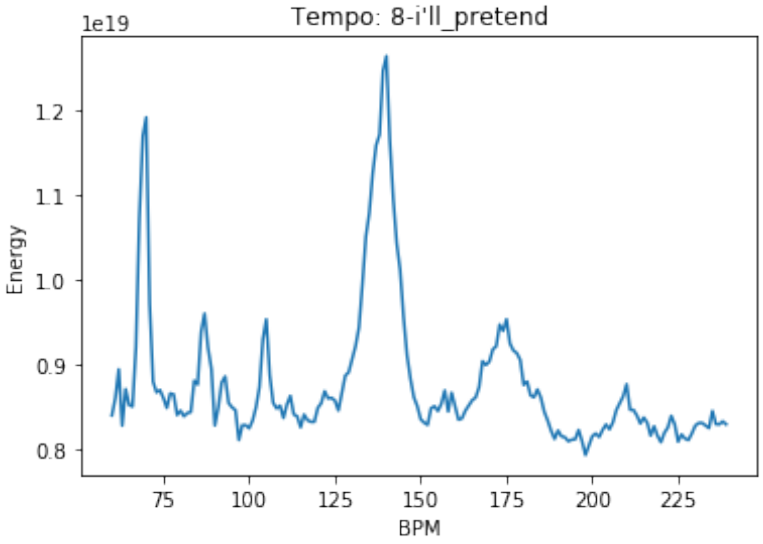
Transformaty sygnału wejściowego będą wielokrotnie wykorzystywane podczas działania funkcji, dlatego są obliczane na początku. Następne obliczenia mają miejsce w pętli, gdzie iteratorem jest tempo w BPM. Zakres tej pętli od minimalnego do maksymalnego tempa z krokiem podanym w argumencie accuracy.

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 5.12 oraz 5.13.

Następnie sygnał filtra jest przenoszony do dziedziny częstotliwości za pomocą



Rysunek 5.14: Rozkład energii dla tempa.

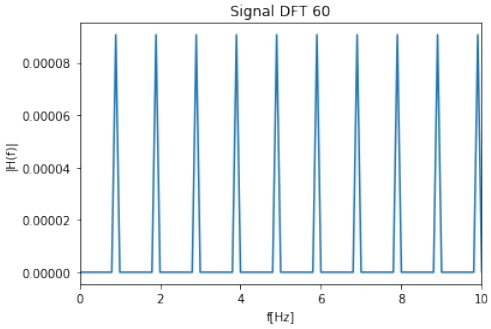
szybkiej transformaty Fouriera FFT (ang. *Fast Fourier Transform*). Na rysunku 5.16 widać, że z powodu stałej długości filtra oraz stałej ilości impulsów, w sygnale transformaty pojawiło się wiele szumów w porównaniu do sygnału widocznym na rysunku 5.15. Pomimo tych szumów ta metoda okazała się być najszybsza i najdokładniejsza. Bardziej poprawne, z perspektywy przetwarzania sygnałów, metody zostały zaprezentowane w 5.7.1.

Obliczona transformata jest kolejno przemnażana przez transformaty sygnału różnicowego i obliczana jest suma energii ilorazów. Podczas przebiegu pętli szukane jest maksimum sum energii. Odpowiadające tempo filtra, dla którego została obliczona maksymalna suma, będzie najprawdopodobniej tempem utworu.

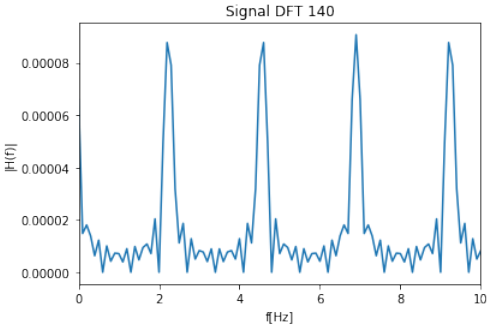
Na rysunku 5.14 zaprezentowano rozkład sum energii dla tempa. Widać, że tempo równe 140 ma największą sumę i takie jest też tempo podanego utworu. Druga największa wartość energii odpowiada tempu 70, ponieważ sygnał filtra pasuje w co drugie uderzenie pulsu utworu.

5.7.1 Alternatywne implementacje wykrywania

Wygląd transformaty filtra dla tempa różnego niż minimalne tempo (rysunek 5.16 może budzić obawy, że wynik funkcji może być niedokładny. Z tego powodu,



Rysunek 5.15: Transformata filtra dla BPM=60.



Rysunek 5.16: Transformata filtra dla BPM=140.

zostały sprawdzone dwa alternatywne sposoby, które mimo poprawności działania (z perspektywy przetwarzania sygnałów) okazały się gorsze pod względem czasu 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 sygnaóow 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, a następnie obliczenie splotu sygnału i filtra i zsumowanie energii obliczonych splotów. Ten sposób trwał zdecydowanie dłużej i koniecznie było przepróbkowanie sygnału, żeby skrócić czas wykonania. Niestety wyniki uzyskane tą metodą nie zgadzały się z faktycznymi tempami w piosenkach i metoda ta została

odrzucona. Kod funkcji wykorzystującej splot został umieszczony w załącznikach w rozdziale 7.

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 rozdziale **??**.

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. Przykładowe przebiegi filtrów zostały zaprezentowane na rysunku 5.17 oraz 5.18. Analogicznie do funkcji wykrywającej tempo, ta funkcja tworzy filtry dla zaimplementowanych metrum i szuka maksimum sum energii ilorazów transformat filtra i sygnału.

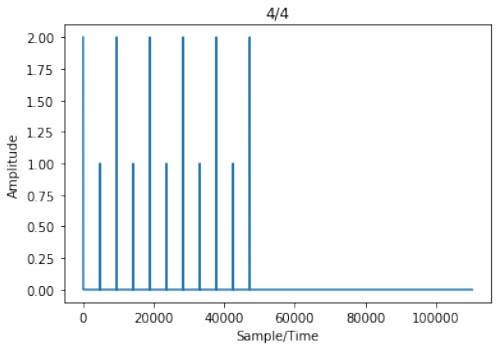
nowe screeny

5.8.2 Wykrycie metrum za pomocą filtra grzebieniowego ze znormalizowanym sygnałem

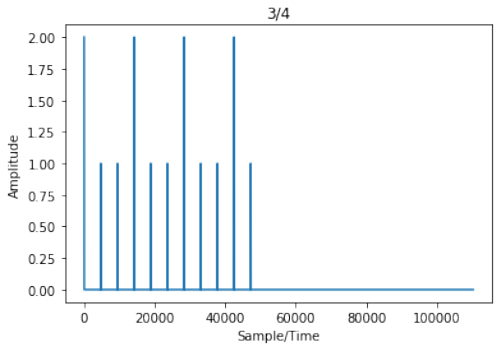
Metoda opisana w 5.8.1 jest niedokładna, ponieważ uzyskane transformaty filtrów, mają różną energię. Do normalizacji sygnału z zastosowano obliczenie wartości pulsów, tak by ich suma była równa 1 oraz podzielenie sygnału transformaty filtra przez jego energię.

5.8.3 Wykrycie metrum za pomocą splotu

Innym sposobem na wykrycie metrum jest metoda podobna do tej opisanej w 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.



Rysunek 5.17: Filtr dla metrum 4/4.



Rysunek 5.18: Filtr dla metrum 3/4.

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 (3.2), poczyniono kroki, by przyspieszyć działanie programu.

5.9.1 Zmniejszenie częstości próbkowania

Tak jak zostało to wspomniane w 5.2, sygnał z utworu muzycznego może zostać przebróbkowany. 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 duży 60-230 BPM, ale z dużym krokiem równym 5. Pozwala to mniej więcej wykryć prawdziwe tempo utworu, ponieważ większość utworów ma tempo podzielne przez 5. Gdyby jednak kompozytor napisał utwór w tempie 118 funkcja jest wywoływana drugi raz. Tym razem accuracy równe jest 1, a przedział to tempo wykryte za pierwszym razem ± 5. 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

- sposób testowania w ramach pracy (np. odniesienie do modelu V) [2]
- organizacja eksperymentów
- przypadki testowe zakres testowania (pełny/niepełny)
- wykryte i usunięte błędy
- opcjonalnie wyniki badań eksperymentalnych

Rozdział 7

Podsumowanie i wnioski

- uzyskane wyniki w świetle postawionych celów i zdefiniowanych wyżej wymagań
- kierunki ewentualnych danych prac (rozbudowa funkcjonalna ...)
- problemy napotkane w trakcie pracy

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] A. Lazaro, R. Sarno, R. J. Andre, M. N. Mahardika. Music tempo classification using audio spectrum centroid, audio spectrum flatness, and audio spectrum spread based on mpeg-7 audio features. *2017 3rd International Conference on Science in Information Technology (ICSITech)*, strony 41–46, 2017.

[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.

[13] Marc Velasco, Edward Large. Pulse detection in syncopated rhythms using neural oscillators. strony 185–190, 2011.

Dodatki

Spis skrótów i symboli

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

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

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

ACF funkcja autokorelacji (ang. *Autocorellation Function*)

SVM Maszyna wektorów nośnych (ang. *Support Vector Machine*)

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

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

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

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

f_s częstotliwość próbkowania

MVC model – widok – kontroler (ang. *model–view–controller*)

N liczebność zbioru danych

μ stopnień przyleżności do zbioru

$\mathbb{E}$ zbiór krawędzi grafu

$\mathcal{L}$ transformata Laplace'a

Źródła

```
1  def bpm_comb_filter(signal, accuracy:int, minBpm:int,
       maxBpm:int, bandsLimits, samplingFrequency,
       combFilterPulses, plotDictionary):
2      n = len(signal[0])
3      bands_amount = len(bandsLimits)
4      dft = np.zeros([bands_amount, 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, bands_amount):
13         dft[band] = np.fft.fft(signal[band])
14         draw_fft_plot(True, dft[band], f"Band[{band}] DFT",
               samplingFrequency)
15
16     maxEnergy = 0
17     for bpm in range(minBpm, maxBpm, accuracy):
18         # % Initialize energy and filter to zero(s)
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(percent_done)
25
26
27          for a in range(0, combFilterPulses):
28              fil[a * int(filter_step) + 1] = 1
29
30          draw_plot(True, fil, f"Timecomb bpm: {bpm}", "
                Sample/Time", "Amplitude")
31          # Get the filter in the frequency domain
32          dftfil = np.fft.fft(fil)
33          draw_comb_filter_fft_plot(True, dftfil, f"Signal
                DFT {bpm}", samplingFrequency)
34
35          for band in range(0, bands_amount):
36              x = (abs(dftfil * dft[band])) ** 2
37              this_bpm_energy = this_bpm_energy + sum(x)
38
39          plotDictionary[bpm] = this_bpm_energy
40          if this_bpm_energy > maxEnergy:
41              songBpm = bpm
42              maxEnergy = this_bpm_energy
43
44      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.

Spis rysunków

Spis tablic