



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

Oświadczenie

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

Gliwice, dnia 17 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 17 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	5
2.2.3	Okno czasowe	5
2.2.4	Splot i funkcja korelacji	6
2.2.5	Filtr grzebieniowy	6
2.2.6	Energia sygnału	6
2.3	Cyfrowe przetwarzanie dźwięku	7
2.4	Studia literaturowe	7
2.5	Rozwiązania komercyjne	9
3	Wymagania i narzędzia	11
3.1	Wymagania funkcjonalne	11
3.2	Wymagania nefunkcjonalne	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	Charakterystyka parametrów	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	26
5.6	Obliczenie pochodnej	26
5.7	Wykrycie tempa za pomocą filtra grzebieniowego	30
5.7.1	Alternatywne implementacje wykrywania tempa	32
5.7.2	Normalizacja sygnału transformaty	34
5.7.3	Liczenie splotu sygnałów	34
5.8	Wykrycie metrum	34
5.8.1	Wykrycie metrum za pomocą filtra grzebieniowego	35
5.8.2	Wykrycie metrum za pomocą filtra grzebieniowego ze znormalizowanym sygnałem	35
5.8.3	Wykrycie metrum za pomocą splotu	36
5.8.4	Wykrycie metrum za pomocą splotu ze znormalizowanym sygnałem	36
5.8.5	Wykrycie metrum za pomocą funkcji autokorelacji	36
5.9	Optymalizacja	37
5.9.1	Zmniejszenie częstości próbkowania	37
5.9.2	Rekurencyjne wykrywanie tempa	37
5.9.3	Zastosowanie architektury CUDA	37
6	Weryfikacja i walidacja	39
6.1	Zbiór testowy	39
6.2	Organizacja testów	39

7	Podsumowanie i wnioski	43
7.1	Analiza wyników	43
7.1.1	Analiza wyników wykrycia tempa	43
7.1.2	Analiza wyników wykrycia metrum	44
7.2	Dalsze prace	44
7.3	Podsumowanie	45

Rozdział 1

Wstęp

Wykrywanie tempa i metrum utworu muzycznego jest podstawowym zagadnieniem poruszonym 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 osób tworzących muzykę na żywo, którzy potrzebują wyrównać równolegle odtwarzane utwory i tzw. sample.

Wiedza na temat tempa i metrum utworu może również pomóc przy próbie jego automatycznego skatalogowania. Posiadając informacje o wykonawcy, tytule oraz gatunku, a dodatkowo wykrywając tempo, metrum i nastrój, można stworzyć bazę utworów. Taka baza potrafiłaby odpowiedzieć na zapytanie, powstałe na podstawie przeznaczenia listy odtwarzania. Na przykład, biegacz mógłby poprosić o utwory w tempie około 150 uderzeń na minutę BPM (ang. *Beats per minute*), ponieważ do takich mu się najwygodniej biega. Chcąc znaleźć dla niego najlepszą muzykę, należy szukać jej wśród żywych utworów z parzystym metrum, na przykład 4/4.

Stworzenie bazy danych utworów, gdzie każda pozycja będzie miała określone tempo, metrum, gatunek oraz nastrój może przyspieszyć pracę producentom filmowym, którzy chcą skompletować ścieżkę dźwiękową do tworzonej produkcji. 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 użytkownikom wyszukiwanie pozycji.

Większa wiedza o utworach mogłaby się również przydać w coraz częściej używanych serwisach oferujących słuchanie muzyki przez internet takich, jak: Deezer¹ czy Tidal². W Spotify³ 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 (podrozdział 2.4) opisanych jest kilka metod wykrywania o podobnej zasadzie działania, dlatego zostanie zaimplementowane parę algorytmów, w celu zbadania, który uzyskuje 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 dokonano przeglądu literatury i dostępnych rozwiązań komercyjnych. Rozdział 3 zawiera informacje o wymaganiach funkcjonalnych i нефункциональных, użytych narzędziach i technologiach, środowiskach programistycznych, w których pisano kod oraz przyjętą metodykę pracy. W dalszej części (rozdział 4) przedstawiona została specyfikacja zewnętrzna. Zawiera ona opis wywołania programu, charakterystykę parametrów 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

W tym rozdziale zostały przedstawione podstawowe zagadnienia z teorii muzyki, jakimi są tempo i metrum, będące tematem tej pracy. Następnie opisano metody cyfrowego przetwarzania sygnałów, których użyto podczas implementacji funkcji wykrywania tempa i metrum. Rozdział kończy się przeglądem literatury i dostępnych rozwiązań komercyjnych.

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 taktu,
 - metrum złożone - więcej niż jeden akcent w obrębie taktu.

- Liczbę miar akcentowanych i nieakcentowanych:
 - metrum parzyste - miary akcentowane i nieakcentowane przypadają na przemienne,
 - metrum nieparzyste - po mierze akcentowanej przypadają dwie miary nieakcentowane.
- Układu metrów prostych w metrum złożonym:
 - metrum złożone regularne - wszystkie metra proste, z których składa się metrum złożone, są identyczne,
 - metrum złożone nieregularne - przynajmniej jedno metrum wchodzące w skład metrum złożonego jest inne niż pozostałe.

Metrum może być powiązane z gatunkiem utworu. Na przykład utwory z gatunku walc są w $3/4$, a utwory z gatunku polka 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 zajmuje 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 w komputerze, musi on zostać poddany dyskretyzacji. Próbkowanie to proces tworzenia sygnału dyskretnego za pomocą ciągu wartości zwanych próbkami. By móc odtworzyć bez zakłóceń sygnał analogowy z cyfrowego, należy odpowiednio dobrać częstotliwość próbkowania. O minimalnej częstotliwości próbkowania mówi twierdzenie 1.

Twierdzenie 1 (Twierdzenie Nyquista-Whittakera-Kotelnikova[7]). *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 częstotliwości ω .

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

Funkcją odwrotną jest odwrotna transformacja Fouriera wyrażona wzorem 2.2 [7],

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

gdzie:

$f(t)$ – funkcja w dziedzinie czasu,

$\hat{f}(\omega)$ - transformata (widmo sygnału) w dziedzinie częstotliwości,

$\omega = \frac{2\pi}{T} = 2\pi\nu$ - częstotliwość.

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

2.2.3 Okno czasowe

Do stworzenia idealnego, niezniekształconego widma sygnału, potrzebna byłaby jego obserwacja w nieskończonym przedziale czasu. Jest to z oczywistych względów

niemożliwe, dlatego też sygnał przemnaża się przez funkcję okna, w celu niwelacji zjawiska zwanego rozmyciem widma.

W tej pracy wykorzystano okno Hanna, które jest zdefiniowane wzorem 2.3 [7],

$$w[n] = \frac{1}{2} - \frac{1}{2} \cos \left(2\pi \frac{n}{N-1} \right) \quad (2.3)$$

gdzie:

N - szerokość okna.

2.2.4 Splot i funkcja korelacji

Splot jest funkcją dla dwóch sygnałów, której wynikiem jest zmodyfikowana wersja w postaci iloczynu splotowego. Funkcja splotu dyskretnego dla funkcji f i g jest opisana wzorem 2.4 [7].

$$(f * g)[n] = \sum_{m=-\infty}^{\infty} f[m] g[n-m] \quad (2.4)$$

Podobną funkcją jest funkcja korelacji wyrażona wzorem 2.5 [7], której wynikiem jest miara podobieństwa sygnałów $x[n]$ i $y[n]$.

$$r_{xy}[l] = \sum_{n=-\infty}^{\infty} x[n+l] y^*[n] \quad (2.5)$$

2.2.5 Filtr grzebieniowy

Jednym z podstawowych filtrów w przetwarzaniu sygnałów jest filtr grzebieniowy (ang. *Comb filter*), którego zasada działania opiera się na symulacji opóźnienia sygnału oryginalnego o jednostkę czasu. Poszczególne składowe częstotliwości są wzmacniane lub wygłuszane poprzez zjawisko interferencji [6].

2.2.6 Energia sygnału

W celu znalezienia filtru, którego sygnał jest najbardziej zbliżony do sygnału oryginalnego, obliczana jest energia ze wzoru 2.6 [7].

$$E = \sum_{n=-\infty}^{\infty} |x(n)|^2 \quad (2.6)$$

2.3 Cyfrowe przetwarzanie dźwięku

Cyfrowe przetwarzanie nagranych dźwięków, 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.

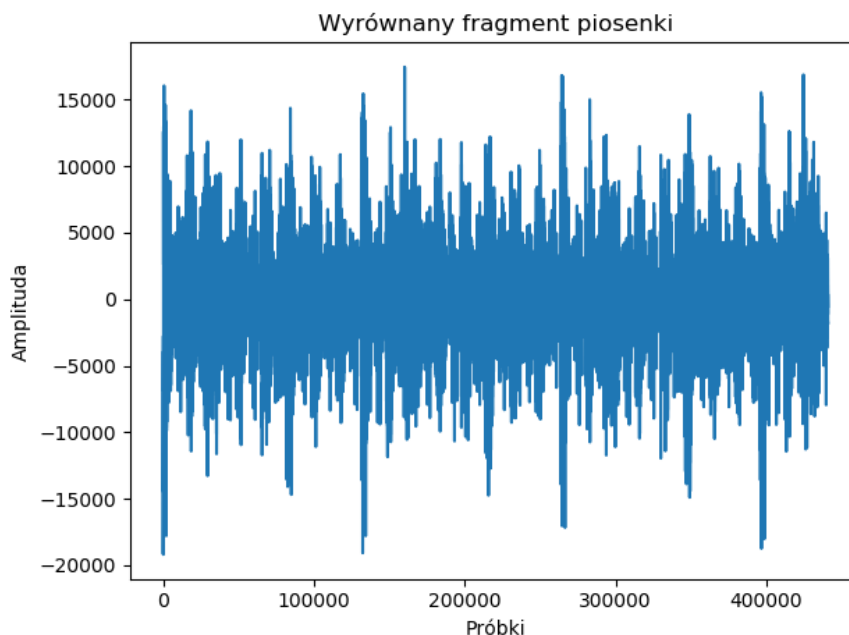
Wcześniej było to niemożliwe, ponieważ ilość danych z nagranych utworów przekraczała możliwości ówczesnych komputerów domowych. Zakres częstotliwości słyszalnych przez człowieka to 8 Hz - 20 kHz [12], a znając twierdzenie 1 wiadomo, że częstotliwość próbkowania nagranych dźwięków powinna wynosić co najmniej 40 kHz. Z tego powodu, jak i standardu dla płyt CD wprowadzonego przez SONY w 1979 roku, za typową częstotliwość próbkowania plików audio przyjęto 44,1 kHz. Pojedyncza próbka ma rozmiar 16 bitów, zatem nieskompresowany trzuminutowy utwór stereo nagrany przy próbkowaniu 44,1 kHz ma rozmiar 31,752 MB. Obecnie taki plik wydaje się być mały i współczesne komputery potrafią go przetwarzać, a także modyfikować w czasie rzeczywistym.

Rysunek 2.1 przedstawia reprezentację graficzną nagranych utworów. Jest to wykres zmian amplitudy natężenia sygnału. Można zaobserwować wyraźne skoki natężenia. W miejscach występowania tych maksimów lokalnych najprawdopodobniej znajdują się uderzenia pulsu utworu. Celem tej pracy jest znalezienie zależności między częstotliwością i metrum a natężeniem i częstotliwością występowania pulsów.

2.4 Studia literaturowe

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

Najczęściej wykorzystywaną metodą do wykrywania tempa jest stworzenie ze-



Rysunek 2.1: Reprezentacja graficzna 10 sekund nagranych utworu gatunku techno.

społu filtrów (ang. *filterbank*) z wykorzystaniem DFT. Zespół filtrów można utworzyć z wykorzystaniem m. in.: filtru prostokątnego [3, 10], filtru o skończonej odpowiedzi impulsowej FIR (ang. *Finite Impulse Response Filter*) [4, 8] lub filtru o nieskończonej odpowiedzi impulsowej IIR (ang. *Infinite Impulse Response*) [11]. Zamiast zespołu filtrów, można obliczyć obwiednie sygnału [11] lub funkcję przepływu spektrogramu [9]. Obliczona w ten sposób funkcja zmienia się wolniej niż sygnał oryginalny i lepiej nadaje się do przetwarzania. W porównaniu do zespołu filtrów, te metody pozwalają na osiągnięcie lepszych rezultatów w muzyce bez sekcji rytmicznej.

Jednym ze sposobów wykrycia tempa jest utworzenie serii filtrów grzebieniowych, w których częstotliwość pulsacji opowiada częstotliwości uderzeń w tempie [3, 10], a następnie znalezienie maksymalnej sumy iloczynów widm sygnałów z zespołu filtrów i filtra [3, 10]. 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. *Autocorrel-*

ation Function) [1, 4, 11]. ACF można również wykorzystać do wykrycia pulsów w sygnale, by następnie obliczyć prawdopodobieństwo ich pojawienia się w czasie [8, 9].

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, 11]. Innym rozwiązaniem zadania wykrywania metrum jest użycie ACF i tworzonych sygnałów z pulsami odpowiadającym akcentom w metrum [13].

Posiłkując się wspomnianymi pracami zdecydowano się na użycie algorytmu z [3] oraz [10] do wykrywania tempa. Wykrycie metrum zostało zaimplementowane w sposób podobny do [13], 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.: Ableton Live¹, Virtual DJ², w których zostało zaimplementowane wykrywanie tempa utworu muzycznego. Taka funkcja 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. Edison³ i programów pomocniczych np.: BPM Analyzer⁴ albo HoRNet SongKey MK3⁵. W internecie dostępna jest strona Get Song BPM⁶, która umożliwia wykrycie tempa. Dostępny jest też program Aubio⁷ na licencji „open-source”, który między innymi pozwala na wykrycie tempa. Niestety podczas przeglądu dostępnych programów do obróbki dźwięku, nie znaleziono komercyjnego rozwiązania oferującego

¹<https://www.ableton.com/>

²<https://www.virtualdj.com/>

³<https://www.image-line.com/plugins/Tools/Edison/>

⁴<https://mixmeister-bpm-analyzer.softonic.pl/>

⁵<https://www.hornetplugins.com/plugins/hornet-songkey-mk3/>

⁶<https://getsongbpm.com/tools/audio>

⁷<https://aubio.org/>

wykrywanie metrum.

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 typy 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 powinien wykrywać najpopularniejsze metra 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 reprezentujących oryginalny i przetworzony sygnał utworu oraz rozkładu dopasowania tempa.
8. Miarą jakości Acc wykrywania jest procentowy stosunek liczby dobrych wyników P do liczby wszystkich utworów w testowanym zbiorze N .

$$Acc = \frac{P}{N} \cdot 100\% \quad (3.1)$$

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

3.2 Wymagania нефunkcjonalne

System nie jest skierowany do klienta dlatego wymagania нефunkcjonalne 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 Anaconda¹. 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`, `tensor-Flow` 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. Pozwala na konfigurację interpretera Pythona wraz z dodatkowymi modułami przez graficzny interfejs użytkownika. Oprogramowanie Anaconda, tak jak Python, jest darmowe.

3.3.2 Środowisko programistyczne

Program powstawał równolegle w dwóch środowiskach. Pisanie kodu było najwygodniejsze w programie JetBrains PyCharm², ale na etapie szukania najlepszego rozwiązania i poprawiania błędów w programie, testowanie kodu było bardziej przejrzyste i szybsze w Jupyter 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 programista korzysta z 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 Juptera.

Jupyter Notebook pozwala tworzyć kod w formie „zeszytów” i uruchamiać kod kawałek po kawałku. Takie rozwiązanie przyspiesza pracę nad dużymi zbiorami

¹<https://www.anaconda.com/>

²<https://www.jetbrains.com/pycharm/>

danych. W przypadku tej pracy było to duże ułatwienie, ponieważ na program 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. Numpy³ jest biblioteką do naukowego przetwarzania danych. Pozwala na operowanie na wielowymiarowych tablicach obiektów oraz udostępnia wiele funkcji matematycznych jak transformata Fouriera.
2. Scipy⁴ jest nakładką na Numpy, która udostępnia kilka dodatkowych funkcji, które były niezbędne do realizacji tej pracy, takich jak tworzenie tablicy z pliku .wav czy funkcja splotu albo ACF.
3. Matplotlib⁵ 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.
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.

Rozdział 4

Specyfikacja zewnętrzna

4.1 Wymagania sprzętowe i programowe

Do uruchomienia programu wymagany jest komputer domowy z zainstalowanym oprogramowaniem Anaconda¹. Wszystkie użyte moduły wchodzą w skład podstawowej wersji Anacondy, dlatego nie ma potrzeby 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 Charakterystyka parametrów

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:

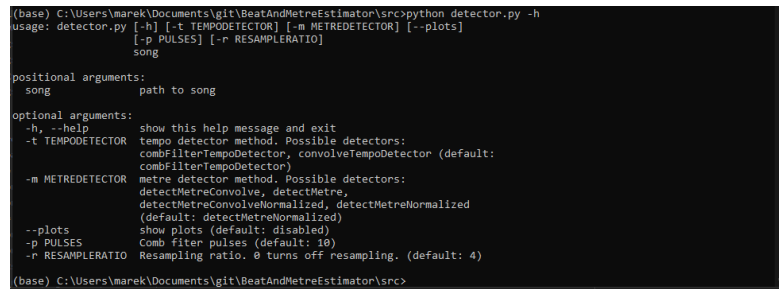
¹<https://www.anaconda.com/distribution/>

- t Wybór sposobu wykrywania tempa. Dostępne metody to:
 - 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

4.3 Pomoc i obsługa błędów

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

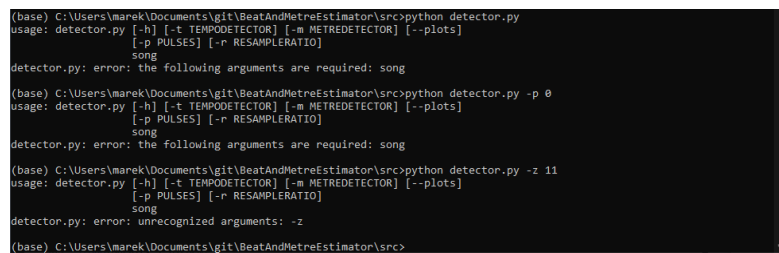
```
python detector.py -h
```



```
(base) C:\Users\marek\Documents\git\BeatAndMetreEstimator\src>python detector.py -h
usage: detector.py [-h] [-t TEMPODETECTOR] [-m METREDETECTOR] [--plots]
                  [-p PULSES] [-r RESAMPLERATIO]
                  song
positional arguments:
  song                  path to song
optional arguments:
  -h, --help            show this help message and exit
  -t TEMPODETECTOR      tempo detector method. Possible detectors:
                        combFilterTempoDetector, convolveTempoDetector (default:
                        combFilterTempoDetector)
  -m METREDETECTOR      metre detector method. Possible detectors:
                        detectMetreConvolve, detectMetre,
                        detectMetreConvolveNormalized, detectMetreNormalized
                        (default: detectMetreNormalized)
  --plots              show plots (default: disabled)
  -p PULSES             comb filter pulses (default: 10)
  -r RESAMPLERATIO      Resampling ratio. 0 turns off resampling. (default: 4)
(base) C:\Users\marek\Documents\git\BeatAndMetreEstimator\src>
```

Rysunek 4.1: Wywołanie programu z zamiarem uzyskania pomocy.

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



```
(base) C:\Users\marek\Documents\git\BeatAndMetreEstimator\src>python detector.py
usage: detector.py [-h] [-t TEMPODETECTOR] [-m METREDETECTOR] [--plots]
                  [-p PULSES] [-r RESAMPLERATIO]
                  song
detector.py: error: the following arguments are required: song

(base) C:\Users\marek\Documents\git\BeatAndMetreEstimator\src>python detector.py -p 0
usage: detector.py [-h] [-t TEMPODETECTOR] [-m METREDETECTOR] [--plots]
                  [-p PULSES] [-r RESAMPLERATIO]
                  song
detector.py: error: the following arguments are required: song

(base) C:\Users\marek\Documents\git\BeatAndMetreEstimator\src>python detector.py -z 11
usage: detector.py [-h] [-t TEMPODETECTOR] [-m METREDETECTOR] [--plots]
                  [-p PULSES] [-r RESAMPLERATIO]
                  song
detector.py: error: unrecognized arguments: -z
(base) C:\Users\marek\Documents\git\BeatAndMetreEstimator\src>
```

Rysunek 4.2: Przykłady błędnego wywołania programu.

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 znalezionej 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` zawiera 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 można usłyszeć różnego rodzaju wprowadzenia w postaci ciszy, szumów czy narastającej dynamiki, które mogą sfalszować wynik. Oczywiście w środku również mogą pojawić się różnego rodzaju przejścia, które nie będą w tempie albo zmiany tempa lub 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-

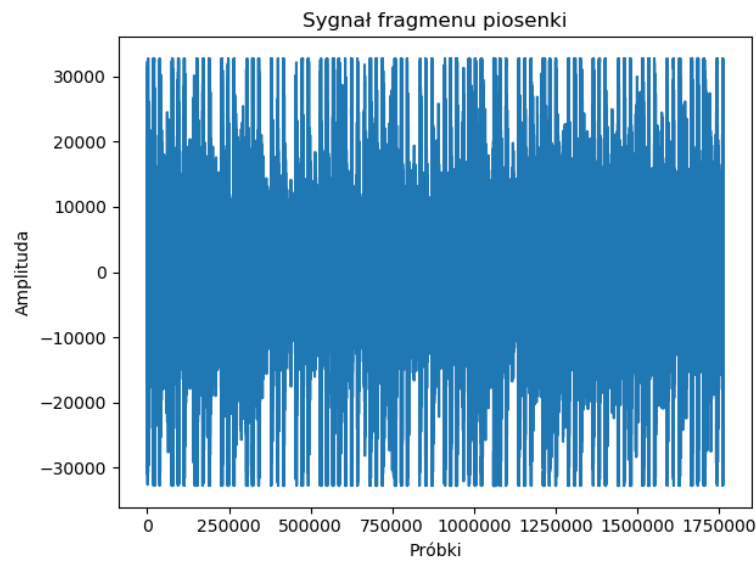
```
1 signal, samplingFrequency = songReader.read_song(song.filepath)
2
3 sample_length = settings.combFilterPulses * samplingFrequency
4 seconds = sample_length * 4
5 song_length = signal.size
6
7 start = int(np.floor(song_length / 2 - seconds / 2))
8 stop = int(np.floor(song_length / 2 + seconds / 2))
9 if start < 0:
10     start = 0
11 if stop > song_length:
12     stop = song_length
13
14 sample = signal[start:stop]
```

Rysunek 5.2: Wycinanie fragmentu ze środka utworu.

kach przetwarzania zostanie on zredukowany po odpowiednim wyrównaniu. Taki nadmiar jest spowodowany faktem, że w środku utworu może wystąpić cichszy fragment, w którym poprawne wykrycie tempa i metrum byłoby trudniejsze.

5.3 Wyrównanie fragmentu do pierwszego napotkanego pulsu

Efekt funkcji opisanej w podrozdziale 5.2 został przedstawiony na rysunku 5.3. Można zauważyć, że pierwsze uderzenie pulsu nie znajduje się na początku sygnału, co może spowodować przekłamaną 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 5.4.



Rysunek 5.3: Wycięty fragment utworu.

```

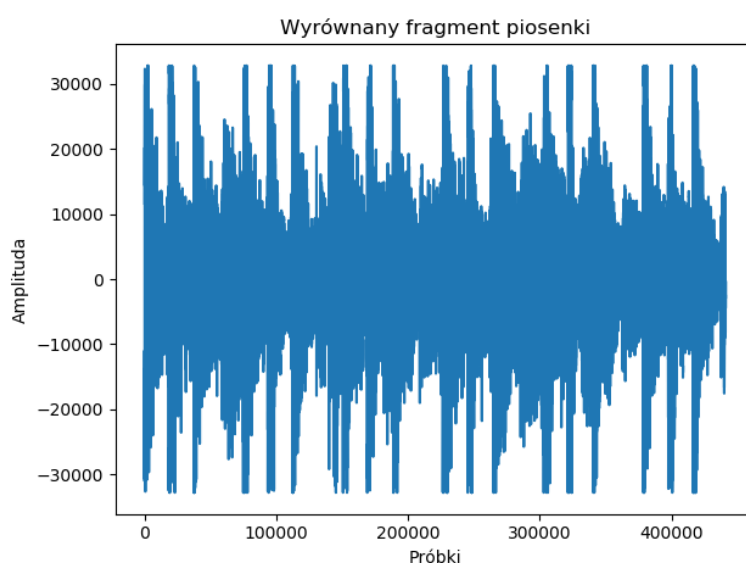
1 def __center_sample_to_beat(self, signal, required_length):
2     n = len(signal)
3     index = 0
4
5     max = np.max(abs(signal))
6
7     for i in range(0, n):
8         if abs(signal[i]) > max * 0.9:
9             index = i
10            break
11
12     lastindex = required_length
13     lastindex += 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 5.4: Funkcja `__center_sample_to_beat` odpowiedzialna za wyrównanie fragmentu do pierwszego napotkanego uderzenia pulsu.

5.4 Transformacja sygnału do dziedziny częstotliwości

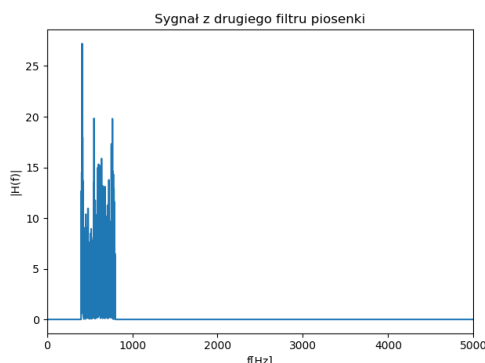
Przygotowany fragment sygnału, który zwraca funkcja opisana w podrozdziale 5.3, został przedstawiony na rysunku 5.5. Można na nim zaobserwować wyraźne regularne maksima, jednak szukanie tempa i metrum na takiej próbce może być niedokładne, z powodu różnych szumów wysokoczęstotliwościowych oraz faktu, że instrumenty melodyczne, nie zawsze grają akcenty, które wynikają z metrum.



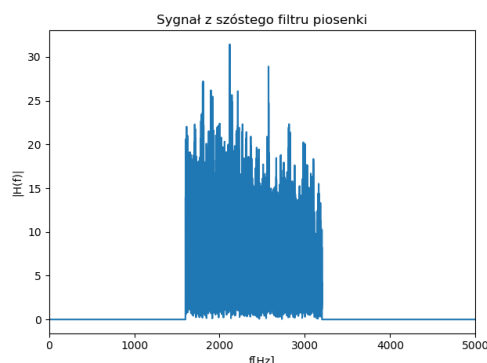
Rysunek 5.5: Fragment wyrównany do pierwszego napotkanego pulsu i zredukowany do pożądanej długości.

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: $\langle 0;200 \rangle$, $\langle 200;400 \rangle$, $\langle 400;800 \rangle$, $\langle 800;1600 \rangle$, $\langle 1600;3200 \rangle$, $\langle 3200; \frac{f_s}{2} \rangle$, gdzie f_s jest częstotliwością próbkowania.. Kod odpowiedzialny za podzielenie sygnału

na zespół filtrów został przedstawiony na rysunku 5.6, a jego efekt na rysunku 5.7 oraz 5.8.



Rysunek 5.7: Widmo sygnału w zakresie $<400;800>$ Hz.



Rysunek 5.8: Widmo sygnału w zakresie $<1600;3200>$ Hz.

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 (podrozdział 2.2.3). Sygnał jest wygładzany, ponieważ w dalszej części przetwarzania zostanie obliczone jego widmo. Kod Funkcji odpowiedzialnej za to został przedstawiony na rysunku 5.9, a przykładowy rezultat na rysunku 5.10.

5.6 Obliczenie pochodnej

Na rysunku 5.10 został przedstawiony przebieg sygnału w konkretnym podpasmie, jednak poszukiwane szczyty sygnału są niewyraźne. Żeby uwydatnić mak-sima, zostaje stworzona tablica, w której zapisywane są różnice między wartościami kolejnych próbek. Na rysunku 5.11 została przedstawiona pochodna sygnału. Wyraźnie na niej widać regularne uderzenia pulsu.

```

1 def __prepare_filterbanks(self, signal, bandlimits, samplingFrequency
2 ):
3     dft = np.fft.fft(signal)
4     n = len(dft)
5     nbands = len(bandlimits)
6     bl = np.zeros(nbands, int)
7     br = np.zeros(nbands, int)
8
9     # % Bring band scale from Hz to the points in our vectors
10    for band in range(0, nbands - 1):
11        bl[band] = np.floor(bandlimits[band] / samplingFrequency * n
12                             / 2) + 1
13        br[band] = np.floor(bandlimits[band + 1] / samplingFrequency
14                             * n / 2)
15
16    bl[0] = 0
17    bl[nbands - 1] = np.floor(bandlimits[nbands - 1] /
18                               samplingFrequency * n / 2) + 1
19    br[nbands - 1] = np.floor(n / 2)
20
21    output = np.zeros([nbands, n], dtype=complex)
22
23    for band in range(0, nbands):
24        for hz in range(bl[band], br[band]):
25            output[band, hz] = dft[hz]
26        for hz in range(n - br[band], n - bl[band]):
27            output[band, hz] = dft[hz]
28
29    output[1, 1] = 0
30    return output

```

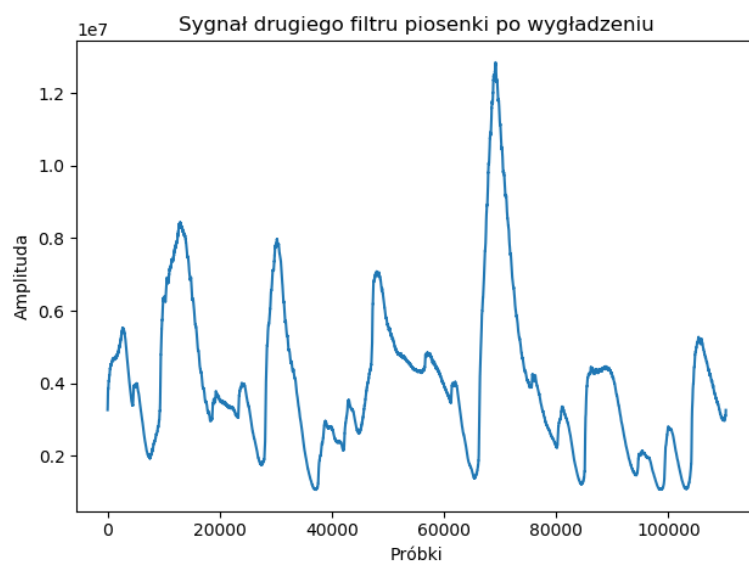
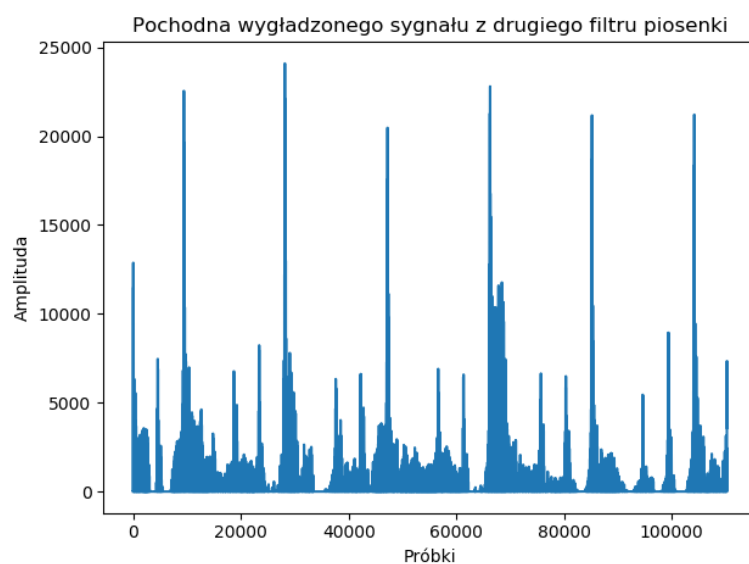
Rysunek 5.6: Funkcja `__prepare__filterbanks` odpowiedzialna za transformatę do dziedziny częstotliwości i podzielenie na zespół filtrów.

```

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 5.9: Funkcja hann odpowiedzialna za transformację do dziedzin czasu i wyrównanie.

Rysunek 5.10: Wyglądzonego sygnału w zakresie $\langle 200; 400 \rangle$ Hz w dziedzinie czasu.Rysunek 5.11: Pochodna sygnału w zakresie $\langle 200; 400 \rangle$ Hz.

5.7 Wykrycie tempa za pomocą filtra grzebieniowego

Metoda `detect_tempo` odpowiedzialna za wykrycie tempa z wykorzystaniem filtra grzebieniowego została umieszczona jako listing 1 w źródłach dołączonych do pracy.

Metoda posiada 7 argumentów. Pierwszym jest dwuwymiarowa tablica pochodnych sygnałów obliczonych w poprzedniej funkcji opisanej w podrozdziale 5.6. Kolejnymi argumentami są precyzja, minimalne i maksymalne tempo, które ma zostać wykryte. Precyzja definiuje krok w pętli, która oblicza maksymalną energię iloczynu widm 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 pochodne sygnału są przenoszone 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. Zakresem pętli jest przedział od minimalnego do maksymalnego tempa, z krokiem podanym w argumentcie `accuracy`.

Dla danego tempa jest tworzony sygnał filtra grzebieniowego. Odległość impulsów `step` jest obliczana ze wzoru 5.1,

$$step = \frac{60}{bpm} * f_s \quad (5.1)$$

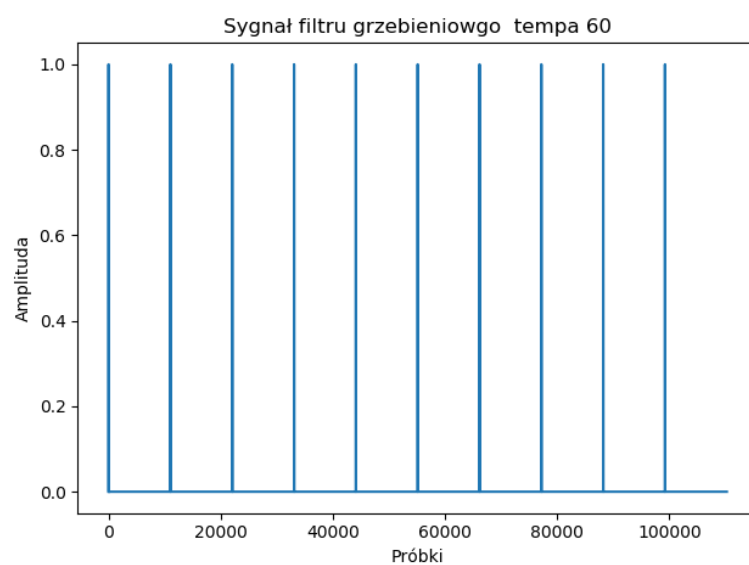
gdzie:

bpm - częstotliwość pulsów w uderzeniach na minutę,

f_s - częstotliwość próbkowania sygnału.

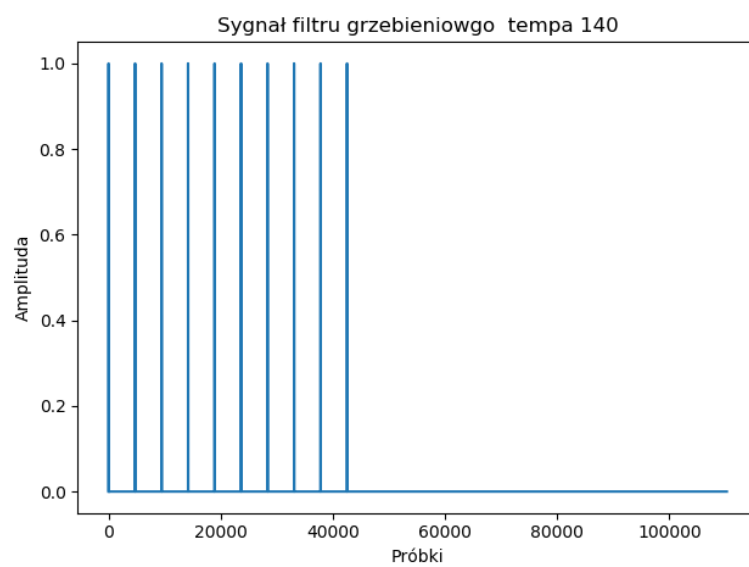
Długość filtra jest stała dla wszystkich wartości BPM i jest zależna od minimalnej wartości tempa, które ma zostać wykryte oraz ilości impulsów filtra. Przykładowe sygnały zostały zaprezentowane na rysunku 5.12 oraz 5.13.

Następnie sygnał filtra jest przenoszony do dziedziny częstotliwości za pomocą szybkiej transformaty Fouriera FFT (ang. *Fast Fourier Transform*). Na rysunku 5.16 widać, że z powodu stałej długości filtra oraz stałej ilości impulsów, w wid-



.5

Rysunek 5.12: Sygnał filtra dla BPM=60.



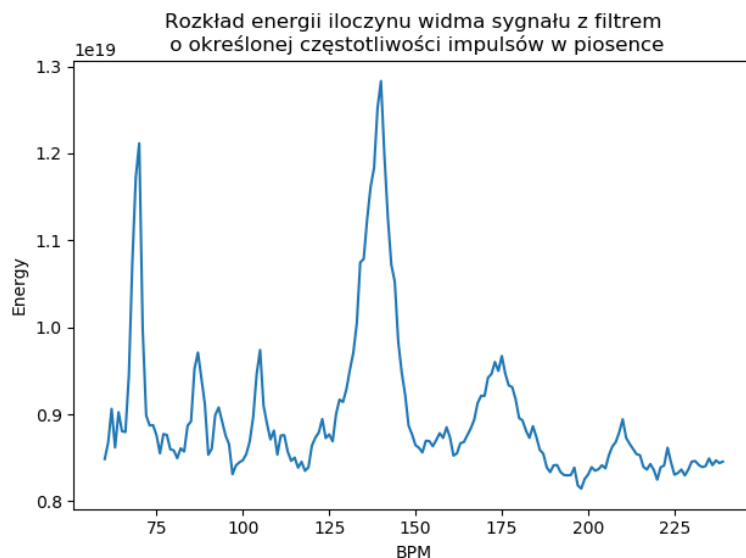
.5

Rysunek 5.13: Sygnał filtra dla BPM=140.

mie pojawiło się wiele szumów w porównaniu do widma na rysunku 5.15. Mimo tych szumów ta metoda okazała się być najszybsza i najdokładniejsza (podrozdział 7.1.1). Bardziej poprawne, z perspektywy przetwarzania sygnałów, metody zostały zaprezentowane w 5.7.1.

Obliczona transformata jest kolejno przemnażana przez transformaty pochodnych. W ten sposób obliczana jest suma energii iloczynów widm pochodnej sygnału i filtra. Podczas przebiegu pętli szukane jest maksimum sum energii. Tempo odpowiadające maksymalnej sumie energii uznawane jest za tempo 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.

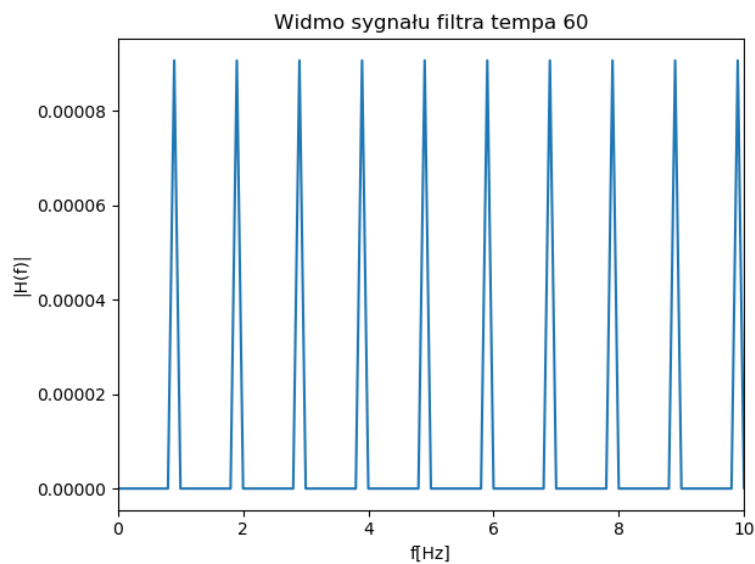


Rysunek 5.14: Rozkład energii iloczynu widma sygnału z filtrem o określonej częstotliwości impulsów.

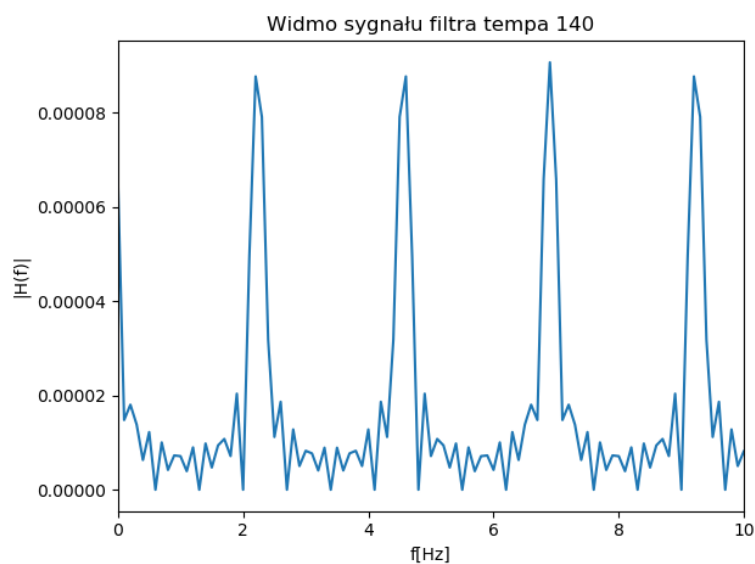
5.7.1 Alternatywne implementacje wykrywania tempa

Wygląd transformaty filtra dla tempa różnego niż minimalne tempo (rysunek 5.16) 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 wykonania, jak i samego rezultatu (podrozdział 7.1.1).



Rysunek 5.15: Widmo filtra dla tempa równego 60.



Rysunek 5.16: Widmo filtra dla tempa równego 140.

5.7.2 Normalizacja sygnału transformaty

Pierwszym sposobem, który miał wyczyścić transformatę filtra, było stworzenie sygnału filtra grzebieniowego o wymaganej długości L obliczonej ze wzoru 5.2,

$$L = step * N \quad (5.2)$$

gdzie:

$step$ - odległość między pulsami obliczania ze wzoru 5.1,

N - liczba pulsów.

Następnie skrócenie sygnału z zespołu filtrów do długości filtra grzebieniowego L , przemnożenie widm sygnałów i obliczenie energii. Obliczona w ten sposób energia została normalizowana przez 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 sygnału z utworu, żeby skrócić czas wykonania. Kod metody `detect_tempo` wykorzystującej splot został umieszczony jako listing 2 w źródłach dołączonych do pracy.

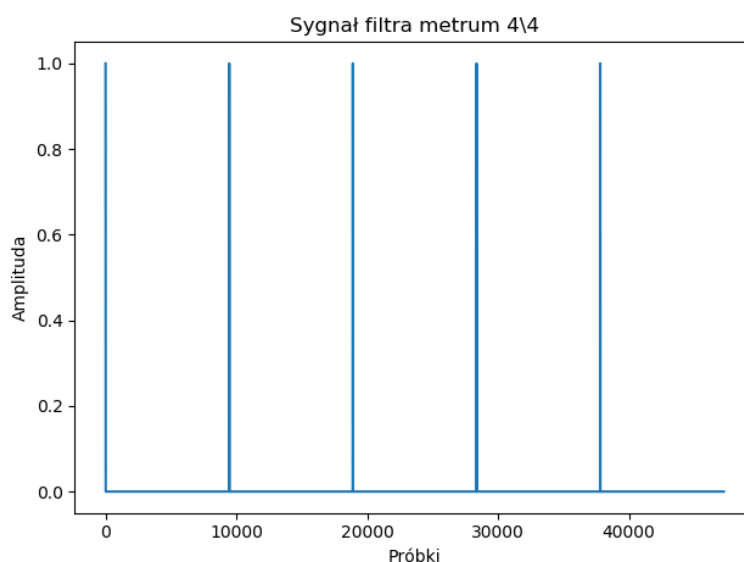
5.8 Wykrycie metrum

Wykrywanie metrum okazało się być zadaniem dużo trudniejszym w implementacji. Z tego powodu powstało parę metod, a uzyskane wyniki zostały opisane w podrozdziale 7.1.2. Metody wykrywania metrum, podobnie jak w przypadku tempa, zostały zaimplementowane z wykorzystaniem wzorca projektowego strategii. Dzięki temu, użytkownik sam może wybrać metodę wykrywania poprzez

wskazanie metody przy wywołaniu programu (podrozdział 4.2.1).

5.8.1 Wykrycie metrum za pomocą filtra grzebieniowego

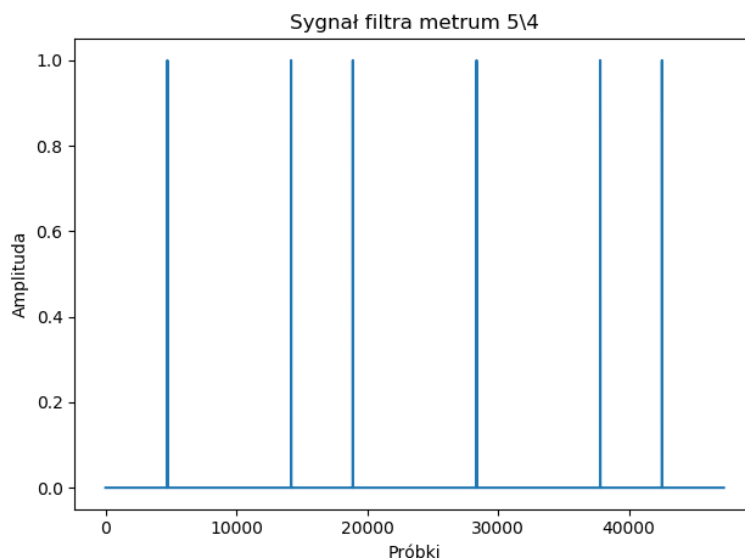
Wykrywanie metrum odbywa się w podobny sposób co wykrywanie tempa (podrozdział 5.7), z tą różnicą, że tempo utworu jest już znane i na jego podstawie, można stworzyć filtry odpowiadające różnym metrum (Rys. 5.17 i 5.18). Analogicznie do funkcji wykrywającej tempo, ta funkcja tworzy filtry dla założonych metrum i szuka maksimum sum energii iloczynów widm filtra i sygnału.



Rysunek 5.17: Sygnał filtra metrum 4/4.

5.8.2 Wykrycie metrum za pomocą filtra grzebieniowego ze znormalizowanym sygnałem

Metoda opisana w podrozdziale 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.



Rysunek 5.18: Sygnał filtra metrum 5/4.

5.8.3 Wykrycie metrum za pomocą splotu

Innym sposobem na wykrycie metrum jest metoda podobna do tej opisanej w podrozdziale 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 jest 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 нефunkcjonalnych nie było narzuconego maksymalnego czasu wykonania programu (podrozdział 3.2), poczyniono kroki, by przyspieszyć jego działanie.

5.9.1 Zmniejszenie częstości próbkowania

Tak jak zostało to wspomniane w podrozdziale 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 zniekształceniu.

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 `detect_tempo` opisana w podrozdziale 5.7 jest wywoływana dwa razy dla różnych wartości argumentów. 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 krokiem 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 widma 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 na-

kładką na bibliotekę Numpy, która udostępnia część funkcji Numpy, ale z wykorzystaniem GPU (ang. *graphics processing unit*). Przeniesienie obliczeń z procesora głównego na kartę graficzną prawdopodobnie znacząco przyspieszyłoby obliczenia, niestety nie wszystkie funkcje zostały przeniesione do Cupy, a kodu opisanego w tym rozdziale nie udało się uruchomić z pomocą wspomnianej 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, posilując 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 razy mniejsze niż rzeczywiste. Jakość 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	Jakość	Metoda Wykrycia metrum	Jakość
16	Filtr grzebieniowy	30,56%	ACF	30,56%
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	Jakość
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	Jakość
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 jakość, jeśli za poprawne uzna się dwukrotnie mniejsze tempo. Wówczas dokładność wykrycia jest zbliżona to metody wykorzystującej filtr grzebieniowy.

Widać wzrost jakości 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.

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 jakością na poziomie 36.11%.

Ciekawym wynikiem jest zerowa jakość 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 jakość 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 duży.

7.2 Dalsze prace

Do celów rozwojowych niezbędne będzie rozwinięcie bazy utworów testowych. Potrzebna jest większa liczba utworów, dla których informacje o tempie i metrum będą pewne i dokładne. Szczególnie ma to znaczenie, jeżeli w przyszłości zaimplementowana zostanie metoda, wykorzystująca uczenie maszynowe.

Przydatną funkcją 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 zajmuje około 2,5 GB. Gdyby wszystkie przekonwertowano i przycięto do około minuty, jak zostało to zrobione z utworami ze zbioru testowego, to zajmowane miejsce

na dysku twardym 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 filtra, 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 w określeniu początku 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 będzie wykonywana większa liczba 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 funkcji 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 tą samą funkcję. Po drugie nie posiada mechanizmu enkapsulacji pól i metod obiektów.

Kompletowanie bibliografii potrzebnej do stworzenia tej pracy zostało przyspieszone przez wykorzystanie serwisu Google Scholar¹, który jest wyszukiwarką do prac naukowych. Po wpisaniu szukanej frazy, użytkownik w rezultacie otrzymuje odnośniki do pozycji, a najczęściej cytowane prace są oznaczane.

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

¹<https://scholar.google.com/>

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: 2019-11-20].
- [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] Julius O. Smith III. Physical audio signal processing. <https://ccrma.stanford.edu/~jos/pasp/>. [data dostępu: 2020-01-14].
- [7] Andrzej Leśnicki. *Technika Cyfrowego Przetwarzania Sygnałów*. Wydawnictwo Politechniki Gdańskiej, Gdańsk, 2014.
- [8] 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.
- [9] Geoffroy Peeters. Time variable tempo detection and beat marking. *IRCAM - Analysis/Synthesis Team*, 2005.

-
- [10] Eric D. Scheirer. Tempo and beat analysis of acoustic musical signals. *The Journal of the Acoustical Society of America*, 103(1):588–601, 1998.
 - [11] 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.
 - [12] Bożena Smagowska, Małgorzata Pawlaczyk-Łuszczynska. Effects of ultrasonic noise on the human body—a bibliographic review. *International journal of occupational safety and ergonomics : JOSE*, 19:195–202, 2013.
 - [13] Petri Toiviainen, Tuomas Eerola. Classification of musical metre with auto-correlation 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

Na dołączonym listingu 1 widoczna jest metoda `detect_tempo`, która wykrywa tempo utworu, wykorzystując do tego filtr grzebieniowy. Tworzenie sygnału filtrów o różnej częstotliwości pulsacji odbywa się w pętli, w której szukana jest maksymalna suma energii iloczynów widm sygnału i filtra. Wynikiem metody jest tempo, przy którym obliczona suma iloczynów widm była największa.

Listing 1: Metoda `detect_tempo` klasy `CombFilterTempoDetector` wykrywającej tempo z wykorzystaniem filtra grzebieniowego.

```
1 def detect_tempo(self, signal, accuracy: int, minBpm: int,
2     maxBpm: int, bandsLimits, samplingFrequency,
3     combFilterPulses, plotDictionary):
4     n = len(signal[0])
5     bands_amount = len(bandsLimits)
6     dft = np.zeros([bands_amount, n], dtype=complex)
7
8     if minBpm < 60:
9         minBpm = 60
10
11    if maxBpm > 240:
12        maxBpm = 240
13
14    for band in range(0, bands_amount):
15        dft[band] = np.fft.fft(signal[band])
16
17    maxEnergy = 0
```

```
17     for bpm in range(minBpm, maxBpm, accuracy):
18         this_bpm_energy = 0
19         fil = np.zeros(n)
20
21         filter_step = np.floor(60 / bpm * samplingFrequency
22                                )
23         percent_done = 100 * (bpm - minBpm) / (maxBpm -
24                                                minBpm)
25         print("%.2f" % percent_done, "%")
26
27         for a in range(0, combFilterPulses):
28             fil[a * int(filter_step) + 1] = 1
29
30         dftfil = np.fft.fft(fil)
31
32         for band in range(0, bands_amount):
33             x = (abs(dftfil * dft[band])) ** 2
34             this_bpm_energy = this_bpm_energy + sum(x)
35
36         plotDictionary[bpm] = this_bpm_energy
37         if this_bpm_energy > maxEnergy:
38             songBpm = bpm
39             maxEnergy = this_bpm_energy
40
41     return songBpm
```

Na listingu 2 widać metodę wykrywającą tempo z wykorzystaniem spłotu sygnałów. Jest struktura jest podobna do metody przedstawionej na listingu 1. W tym przypadku szukana jest maksymalna suma energii widm spłotu sygnałów i filtra.

Listing 2: Metoda `detect_tempo` klasy `ConvolveTempoDetector` wykrywającej tempo z wykorzystaniem spłotu sygnałów.

```
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}]\nDFT", 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 -
23            minBpm)
24        fil = np.zeros(int(filterLength * nstep))
25
26        print(percent_done)
27
28        for a in range(0, filterLength):
29            fil[a * int(nstep)] = 1
30
31        draw_plot(drawCombFilterPlots, fil, f"Timecomb_{bpm}:\n{bpm}", "Sample/Time", "Amplitude")

```

```
31
32     dftfil = np.fft.fft(fil)
33
34     draw_comb_filter_fft_plot(drawCombFilterPlots,
35                               dftfil, f"Signal_DFT_{bpm}", maxFreq)
36     for band in range(0, nbands-1):
37         filt = scipy.convolve(signal[band], fil)
38         f_filt = abs(np.fft.fft(filt))
39         draw_fft_plot(drawFftPlots, f_filt, f"Signal_
40                       DFT_{bpm}", maxFreq)
41
42         x = abs(f_filt)**2
43         e = e + sum(x)
44
45     plot_dictionary[bpm] = e
46     if e > maxe:
47         sbpm = bpm
48         maxe = e
49
49     return sbpm
```

Zawartość dołączonej płyty

Do pracy dołączona jest płyta CD z następującą zawartością:

- praca (źródła \LaTeX owe i końcowa wersja w pdf),
- źródła programu,
- dane testowe,
- uzyskane wyniki.

Spis rysunków

2.1	Reprezentacja graficzna 10 sekund nagranych utworu gatunku techno.	8
4.1	Wywołanie programu z zamiarem uzyskania pomocy.	19
4.2	Przykłady błędnego wywołania programu.	19
5.1	Funkcja <code>read_song</code> wczytująca pliki .wav.	22
5.2	Wycinanie fragmentu ze środka utworu.	23
5.3	Wycięty fragment utworu.	24
5.4	Funkcja <code>__center_sample_to_beat</code> odpowiedzialna za wyrównanie fragmentu do pierwszego napotkanego uderzenia pulsu.	24
5.5	Fragment wyrównany do pierwszego napotkanego pulsu i zredukowany do pożądanej długości.	25
5.7	Widmo sygnału w zakresie <400;800> Hz.	26
5.8	Widmo sygnału w zakresie <1600;3200> Hz.	26
5.6	Funkcja <code>__prepare__filterbanks</code> odpowiedzialna za transformację do dziedziny częstotliwości i podzielenie na zespół filtrów.	27
5.9	Funkcja <code>hann</code> odpowiedzialna za transformację do dziedziny czasu i wyrównanie.	28
5.10	Wygładzony sygnał w zakresie <200;400> Hz w dziedzinie czasu.	29
5.11	Pochodna sygnału w zakresie <200;400> Hz.	29
5.12	Sygnał filtra dla BPM=60.	31
5.13	Sygnał filtra dla BPM=140.	31
5.14	Rozkład energii iloczynu widma sygnału z filtrem o określonej częstotliwości impulsów.	32
5.15	Widmo filtra dla tempa równego 60.	33

5.16	Widmo filtra dla tempa równego 140.	33
5.17	Sygnał filtra metrum 4/4.	35
5.18	Sygnał filtra metrum 5/4.	36

Spis tabel

6.1	Baza utworów testowych.	40
6.2	Uzyskane rezultaty w testach.	41
6.3	Uzyskane rezultaty, jeżeli dwukrotnie mniejsze tempo jest poprawne.	41
6.4	Uzyskane rezultaty wykrycia metrum, przy dobrze rozpoznanym tempie.	42