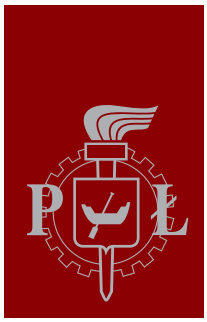


Politechnika Łódzka
Wydział Fizyki Technicznej, Informatyki
i Matematyki Stosowanej

Filip Kobierski
242336

PRACA DYPLOMOWA
inżynierska
na kierunku Informatyka Stosowana

Aplikacja CLI w języku Zig do analizy dźwięku wzorowana na MusicScope

Instytut Informatyki I72

Promotor: dr hab. inż. Bartłomiej Stasiak

ŁÓDŹ 2026

Spis treści

Wstęp **6**

Cel i zakres pracy **7**

1 Analiza albumów muzycznych **8**

2 Cechy dobrego programu do analizy albumów **9**
2.1 Dostępność 9
2.1.1 Dostępność technologiczna 9
2.1.2 Dostępność dla osób z niepełnosprawnościami (*accessibility*) . . 10
2.1.3 Dostępność prawna 10
2.2 Kompleksowość analizy 10
2.2.1 Przykładowe zastosowanie wskaźników 11
2.3 Efektywność 11
2.3.1 Ergonomia interfejsu 11
2.3.2 Efektywność analizy 11
2.3.3 Pomiar efektywności analizy 11
2.3.4 Ergonomia raportów 13

3 MusicScope **14**
3.1 Dostępność technologiczna 14
3.2 *Accessibility* 14
3.3 Dostępność prawna 15
3.3.1 Uzyskiwanie 15
3.3.2 Rozpowszechnianie 15
3.4 Kompleksowość analizy 15
3.5 Ergonomia interfejsu 16
3.5.1 Przedstawienie interfejsu 16
3.5.2 Przedstawienie użytkowania 18
3.5.3 Ocena 21
3.6 Efektywność analizy 21
3.7 Ergonomia raportów 22
3.7.1 Podświetlanie składni 22
3.7.2 Szerokość tabeli 22
3.7.3 Ekscesywna ilość raportowanych danych 23
3.8 Podsumowanie 23

4 Próba pierwsza: SINMS **24**
4.1 Interfejs użytkownika 24
4.1.1 Usunięcie zbędnych elementów GUI 24
4.1.2 Zmiana celu interfejsu analizy 25
4.1.3 Interfejs linii komend (CLI) 25
4.1.4 Deklaratywność 25
4.1.5 Podsumowanie 26
4.2 Język programowania i biblioteki 26
4.3 Raportowanie 27

4.4 Efektywność działania 28
4.5 Status quo SINMS 28
4.6 Haskellowe SINMS 29

5 Próba druga: `sd2` 29
5.1 Język programowania i biblioteki 30
5.1.1 C++ 30
5.1.2 Rust 30
5.1.3 Zig 31
5.2 Zalety Ziga 32
5.2.1 Pełna kompatybilność z C 32
5.2.2 Koncept `comptime` 32
5.2.3 Instrukcje `defer` i `errdefer` 34
5.2.4 Przenośność 34
5.2.5 Lepsze doświadczenie tworzenia oprogramowania 36
5.3 Rozwój `sd2` 36
5.3.1 Porównanie do SINMS 36
5.3.2 Pierwsza biblioteka Ziga 37
5.3.3 Biblioteka `yazap` i model open-source 37
5.3.4 `small_float` 37
5.3.5 Nowe `TrackInfo` 37
5.3.6 Standard REUSE 38
5.3.7 Zmiany biblioteki standardowej Ziga 38
5.3.8 Plik README 39
5.4 Efektywność działania 40
5.5 Status quo `sd2` 42

Podsumowanie 43

Podziękowania 44

Bibliografia 45

Spis rysunków 48

Spis tabel 48

Spis kodów 48

Streszczenie

Ta praca dokumentuje proces powstania projektu `sd2` od konceptów poczętych przed napisaniem pojedynczej linijki kodu aż po upublicznienie gotowego rozwiązania na platformie hostingowej git.

`sd2` to oprogramowanie do analizy albumów muzycznych będące wolnościową reimplementacją od zera (ang. *rewrite from scratch*) funkcjonalności oferowanych przez własnościowy program MusicScope. Jego celem jest naprawienie największych błędów MusicScope i zapewnienie powszechnego dostępu do programu będącego efektem tej pracy.

W tym dokumencie zawiera się zdefiniowanie zagadnienia analizy albumów muzycznych, przedstawienie cech dobrego programu do analizy albumów muzycznych, analiza istniejącego już programu, proces projektowania reimplementacji trzech prototypów w trzech językach programowania, proces rozwoju ostatniego z nich w pełnoprawny produkt gotowy do użycia w prawdziwym życiu oraz analiza tego produktu wedle wcześniej wyznaczonych kryteriów.

Słowa kluczowe: audio, analiza dźwięku, CLI, ebur128, Zig
Keywords: audio, audio analysis, CLI, ebur128, Zig

Wstęp

Reimplementacje od zera są tak stare jak samo oprogramowanie i są szczególnie trudne gdy deweloperzy nie mają dostępu do kodu źródłowego. Są one jednak szczególnie ważne w przypadku gdy reimplementacja rozpowszechniana jest jako Wolne Oprogramowanie – tak właśnie powstał GNU Emacs [34] po raz pierwszy opublikowany w 1985 roku, a używany do dziś na przykład przeze mnie do tworzenia oprogramowania i dokumentów.

Zagadnienie analizy albumów muzycznych, które jest głównym tematem niniejszej pracy, opiszę w detalach w sekcji 1, jednak już teraz napiszę, że jest to bardzo niszowa dziedzina. Posługuję się oprogramowaniem analizującym albumy od niemalże pięciu lat, więc jestem w stanie wskazać pożądane cechy tego oprogramowania, co robię w sekcji 2. Według mnie jedyne używalne rozwiązanie dla użytkownika końcowego to MusicScope, które jest niedostępne i ma mnóstwo wad o co opisuję w sekcji 3. Z tego powodu uważam, że `sd2`, czyli moja finałowa reimplementacja, jest szczególnie przydatna i pożądana.

W sekcjach 4–5 opisuję i uzasadniam najważniejsze decyzje projektowe prób reimplementacji, od języka programowania przez struktury danych aż po paradygmat interfejsu użytkownika. W tych rozdziałach znajdują się również pomiary efektywności funkcjonalnych prototypów.

Na koniec opisuję i komentuję proces tworzenia i utrzymywania `sd2`. Porównuję je również z MusicScope obiektywnie udowadniając, że mój projekt wyzwolenia i ulepszenia tego narzędzia zakończył się sukcesem.

Już we wstępie zachęcę natomiast do analizy kodu źródłowego mojego i samodzielnego przetestowania jak działa. Niezbędne do tego informacje znajdują się na stronie `https://codeberg.org/fkobi/sd2`.

Cel i zakres pracy

Celem niniejszej pracy jest reimplementacja funkcjonalności tworzenia raportów tekstowych programu do analizy dźwięku MusicScope bez korzystania z jego kodu źródłowego (ang. *rewrite from scratch*). Tworzenie programu od zera pozwala zmienić wiele kluczowych jego aspektów. To przedsięwzięcie ma również na celu:

1. uczynienie oprogramowania dostępnym
    - opublikowanie go jako Wolne Oprogramowanie,
    - natywne wsparcie jądra Linux,
2. ulepszenie ergonomii zarówno programu jak i generowanych raportów,
3. znaczące zwiększenie efektywności analizy
    - zmiana paradygmatu interfejsu użytkownika z GUI aktualizowanego w czasie rzeczywistym na CLI,
    - wykorzystanie języka programowania Zig.

Wybrałem ten temat głównie z dwóch powodów. Po pierwsze chciałem aby moja praca inżynierska rozwiązywała realny problem, a po drugie aby była pretekstem do nauczenia się nowego języka programowania systemowego. Wiem, że rozwiązany przeze mnie problem jest realny – przez pięć lat byłem Redaktorem Muzycznym w Studenckim Radiu Żak Politechniki Łódzkiej, z czego prawie dwa lata pełniłem funkcję Szefa Redakcji muzycznej. Ponadto od jeszcze dłuższego czasu jestem audiofilem, więc można powiedzieć, że ów problem dotyka mnie na porządku dziennym. Jeśli chodzi o programowanie systemów, na początku roku 2024 wiedziałem o Zigu tylko to, że, podobnie jak C, jest językiem programowania systemowego z ręcznie zarządzaną pamięcią oraz, że jego interoperacyjność z C jest na bardzo wysokim poziomie. Teraz napisałem kompleksowy program w tym języku wykorzystując biblioteki w obydwu językach. Co chyba ważniejsze, zaproponowane przeze mnie zmiany zostały przyjęte do używanych przeze mnie bibliotek, w tym biblioteki standardowej Ziga!

1 Analiza albumów muzycznych

W kontekście tej pracy album oznacza wydany zbiór utworów muzycznych. Rozpocznę więc od rozpatrzenia go z perspektywy przetwarzania sygnałów.

Album dziełem muzycznym więc, przetwarzany jest dźwięk, czyli słyszalne pasmo fal akustycznych.

Nieodzowną cechą albumu jest to, że jest on sygnałem o ściśle określonej zawartości. Jest tak ponieważ album to dystrybucja efektu procesu produkcji muzycznej, czyli miksu finałowego zwanego też *masterem*. Kopie mastera są rozpowszechniane na różnych mediach, od analogowych płyt winylowych aż po cyfrowe systemy wbudowane [33]. Album jako sygnał ma więc **jedną poprawną formę**. Nawet resampling w wyższej częstotliwości niebędącej wielokrotnością oryginalnej zmienia w małym stopniu zapis i kopia w (niepotrzebnie) wyższej rozdzielczości nie będzie identyczna na poziomie bitowym (bit-perfect).

Wykorzystując bezstratne kodeki można zarejestrować dźwięk na cyfrowym nośniku. W kontekście muzyki, w przypadku modulowania PCM[1], sygnał dźwiękowy, musi być zapisany w *bezstratnym kodeku* przynajmniej z częstotliwością 44100 Hz i szesnastoma bitami głębokości [8]. Ta jakość ma również miano *Redbook audio*, a standard zwie się *Redbookowym* ze względu na to, że dokument który go definiował był czerwoną książką.

Dane cyfrowe mogą być idealnie reprodukowane więc dystrybucja cyfrowa umożliwia słuchaczom doświadczenie dzieła w dokładnie takiej formie jaką przygotował artysta. Jest to możliwe tylko jeśli skorzystają z bezstratnego kodeka dźwięku, co za sprawą ustandaryzowanego Wolnego Bezstratnego Kodeka Dźwięku [11] nie jest trudne. Pliki, z których odtwarzana jest muzyka mogą być jednak wynikiem przetwarzania stratnego. Źródło stratności może wynikać między innymi z:

- fizyczności medium (np. uszkodzona płyta lub czytnik płyt),
- reenkodowania z użyciem stratnego kodeka,
- użycia filtrów wprowadzających przestery, w tym ISP (ang. *Inter-Sample Peaks*).

Założenie, że cyfrowość zapisu zapewnia bezstratność treści jest niepoprawne i może mieć poważne konsekwencje [23].

Instytucje i jednostki prowadzące muzyczne bazy danych którym zależy na jakości dźwięku mogą więc analizować posiadane pliki dźwiękowe celem sprawdzenia ich jakości. W Studenckim Radiu Żak Politechniki Łódzkiej na przykład ten krok jest częścią cotygodniowego procesu wcielania nowych utworów do emisji.

[1] Ze względu na skrajną niszowość dystrybucji muzyki w formacie DSD nie będę jej omawiał w tej pracy.

2 Cechy dobrego programu do analizy albumów

Korzystam z oprogramowania do analizy albumów około pięciu lat. W tym czasie przez niespełna dwa lata pełniłem w Żaku rolę Szefa Redakcji Muzycznej i byłem odpowiedzialny za kontrolę jakości utworów wgrywanych do naszej bazy danych. Uważam więc, że mam wystarczająco dużo doświadczenia abym mógł wskazać najważniejsze cechy dobrego programu do analizy albumów.

2.1 Dostępność

Programu, który nie jest dostępny nie można użyć więc jego inne cechy są pomijalne. Tym samym jest to jego najważniejsza charakterystyka.

Najbardziej podstawowym typem dostępności jest dostęp do danych programu (np. posiadanie płyty lub możliwość pobrania plików z internetu). Oprogramowanie, którego dane są niedostępne (np. port Minecrafta na AppleTV [21] można sklasyfikować jako *lost media*[2]). Ten status czyni oprogramowanie fundamentalnie nieużywalnym całkowicie wykluczając wszelkie dalsze rodzaje dostępności. Z tego powodu w tej pracy będę pisał o oprogramowaniu, którego dane są dostępne.

Dostępność oprogramowania definiuję i oceniam więc trzema charakterystykami które opisałem w poniższych sekcjach.

2.1.1 Dostępność technologiczna

Posiadanie kodu źródłowego bądź plików dystrybucji oprogramowania nie jest warunkiem dostępności technologicznej; program musi działać z rozsądną efektywnością.

Jak zazwyczaj w informatyce, dostępność technologiczną można skategoryzować na poziomie oprogramowania (S) i osprzętowania (H). Poniżej wymieniłem przykładowe wymagania:

- H: architektura procesora,
- H: proste wymagania sprzętowe (np. 2GB pamięci operacyjnej),
- H: założenie obecności pewnego sprzętu (np. CUDA),
- S: system operacyjny,
- S: format pliku wykonywalnego,
- S: obecność bibliotek systemowych (np. GNU libc).

[2]:`https://lostmediaarchive.fandom.com/wiki/Minecraft_(lost_Apple_TV_port_of_the_game)`

2.1.2 Dostępność dla osób z niepełnosprawnościami (*accessibility*)

Program może technicznie być w stanie działać na danym sprzęcie, jednak użytkownik dalej może nie być w stanie z niego skorzystać ze względu na swoje niepełnosprawności.

Przykładowo oprogramowanie z GUI na OpenBSD aby być dostępne dla osób niewidomych powinno implementować Assistive Technology Service Provider Interface [28] aby być kompatybilnym z czytnikami tekstu takimi jak Orca [29].

2.1.3 Dostępność prawna

Z punktu widzenia prawa można powiedzieć, że Wolne Oprogramowanie jest definicją dostępności, a rozwiązania własnościowe są jego antytezą.

Wolne oprogramowanie wedle definicji Europejskiej Fundacji Wolnego Oprogramowania gwarantuje swoim użytkownikom cztery wolności:

1. wolność korzystania,
2. wolność analizy,
3. wolność dystrybucji,
4. wolność ulepszeń (zmian).

Na dłuższą metę dostępność prawna jest więc najistotniejsza; dzięki czwartej wolności dwa pozostałe typy dostępności mogą zostać ulepszone.

2.2 Kompleksowość analizy

Analiza ma ocenić jak bardzo zawartość różnych plików jest podobna do mastera celem wybrania tego o najwyższej jakości.

Jeśli finałowy miks jest dostępny, to ocena podobieństwa jest tak prosta jak dodanie do siebie sygnału testowanego i odwrotności mastera a następnie zliczenie na przykład RMS. Jednakże nie ma wtedy potrzeby analizować sygnału pochodnego gdyż można wyeksportować nowy sygnał o odgórnie określonych charakterystykach. Oddolne ustalenie charakterystyk wymaga usługi działającej na zasadzie MusicBrainz czy CDDB i wykracza poza zakres tej pracy.

Wynikiem analizy powinny być więc czytelne dla człowieka charakterystyki pozwalające na ocenę która dystrybucja przejawia najwięcej dobrych praktyk zapisu sygnałów dźwiękowych.

Moim zdaniem najważniejsze wskaźniki to:

1. TPL (ang. *True Peak Level*) – najwyższa zarejestrowana amplituda,

2. I-Loudness (I-LUFS) – głośność mierzona w pełnej skali zgodnie z [2],
3. LRA (ang. *Loudness RAnge*) – zakres głośności,
4. RMS (ang. *Root Mean Square*) – średnia kwadratowa sygnału.

2.2.1 Przykładowe zastosowanie wskaźników

- Europejska Unia Nadawców rekomenduje [18] aby:
  - TPL nie przekraczał $-1(\pm 0.3)$ dB, aby uniknąć clippingu,
  - I-LUFS wynosił $-23(\pm 1)$ LUFS, aby zmniejszyć potrzebę normalizacji.
- Zmniejszony zakres głośności może wskazywać użycie kompresora.
- RMS powinno być jak największe przy TPL nie przekraczającym zera.

2.3 Efektywność

Efektywność jest ważna dopiero w dostępnym programie który przeprowadza kompleksową analizę. Wydzieliłem trzy typy efektywności które opisuję w poniższych podsekcjach.

2.3.1 Ergonomia interfejsu

Interfejs użytkownika, jak przystało na dziedzinę interakcji człowiek-komputer, powinien być ergonomiczny. Wygoda korzystania z interfejsu bezpośrednio przekłada się na efektywność całego procesu. Interfejs powinien więc być czytelny, zrozumiały i zapewniać użyteczne możliwości dostosowania jego działania.

2.3.2 Efektywność analizy

Analiza dźwięku jest złożona obliczeniowo i wymagająca pamięciowo. Ponieważ jest to główna funkcja programu to właśnie na nią powinien on poświęcać prawie całość swoich zasobów.

Wskazane jest aby program optymalnie zarządzał zasobami przydzielonymi mu przez system operacyjny, więc na przykład aby nie miał garbage-collectora i kompilował się do kodu natywnego.

2.3.3 Pomiar efektywności analizy

Aby zmierzyć efektywność analizy przygotowałem serię testów. Są to proste pomiary **czasu analizy** sześciu albumów o różnych częstotliwościach i głębokościach próbkowania, ale też długościach i ilościach utworów. Spis tych albumów i ich charakterystyk znajduje się w tabeli 1 a poniżej znajduje się wyjaśnienie oznaczeń w niej użytych:

- SR (ang. *Sampling Rate*)– częstotliwość próbkowania w Hz,
- BD (ang. *Bit Depth*) – głębokość bitowa[3],
- n – ilość utworów,
- l – długość trwania w sekundach,
- s – całkowity rozmiar plików audio (zmierzony Unixową komendą `du --summarize --human-readable`).

Tabela 1: Informacje o albumach wykorzystanych do testowania efektywności analizy

Oznaczenie	Album	SR [kHz]	BD	n	l [s]	s [B]
HYT	Hybrid Theory	48	24	12	2270	510M
Ż63	Live Żak 63	48	24	25	4700	906M
MID	Made In The Dark	44.1	16	13	3258	352M
PPP	Pink Season: The Prophecy	44.1	16	4	915	111M
RAM	Random Access Memories	88.2	24	13	4475	1.5G
TTS	That's The Spirit	96	24	11	2703	1G

Testy przeprowadziłem w moim środowisku, czyli:

- OS: Gentoo Linux 2.18 (`default/linux/amd64/23.0/desktop`);
  - jądro: `gentoo-kernel-6.12.58`,
  - flagi kompilacji `-O2 -march=native -flto=8`,
- CPU: Intel i5-1135g7,
- RAM: Micron 16 GB LPDDR4,
- SSD: Samsung 860 EVO (M.2).

Aby ujednolicić czynnik cieplny, testy zaczynałem po uprzednim rozgrzaniu sprzętu do temperatury około 80 stopni Celcjusza narzędziem `stress`.

Szybkość v zdefiniowałem jako stosunek długości albumu do średniego czasu jego analizy t:

$$v = \frac{l}{t}$$

Przepływność X jest natomiast stosunkiem rozmiaru albumu do czasu jego analizy t:

$$X = \frac{s}{t}$$

Jest to przydatna metryka ponieważ analizowane przeze mnie albumy są zakodowane w formacie FLAC z różnym stopniem kompresji.

[3]ilość bitów poświęcona na zapis amplitudy sygnału

2.3.4 Ergonomia raportów

Program powinien generować raporty w postaci przejrzystych plików których odczyt nie wymaga specjalistycznego oprogramowania.

3 MusicScope

MusicScope (MS) [37] to graficzny program do analizy dźwięku stworzony przez firmę XiVero. Licencja na jedną instalację kosztowała około 25 Euro.

Oficjalny opis producenta brzmi następująco:

> The MusicScope is a high precision software audio analyzer and measuring tool that works as an Audio-Microscope to **visualize** the different quality aspects of a music collection.

Pozwoliłem sobie pogrubić słowo *visualize* ponieważ moim zdaniem to kluczowa cecha MS: ma to być program do wizualizacji różnych aspektów dźwięku.

Poza analizą plików dźwiękowych program umożliwia ich odtwarzanie oraz analizę sygnałów odbieranych z wejścia audio w czasie rzeczywistym. Są one niezależne i w kontekście analizy albumów duplikują funkcjonalność analizy plików z o wiele gorszą ergonomią, więc więcej o nich nie wspomnę.

Najstarsza informacja o MS jaką mogłem znaleźć to prezentacja programu opublikowana na serwisie YouTube 28 lutego 2015 roku.

W tej sekcji przeanalizuję i ocenię to oprogramowanie jako narzędzie do analizy albumów na podstawie moich doświadczeń z Windowsową wersją 2.1.0. Przez parę lat używałem jej na Windowsie 10, a teraz wyłącznie na potrzeby mojej pracy inżynierskiej, korzystam z niej na Gentoo Linux [12] przy użyciu Wine [38].

3.1 Dostępność technologiczna

MusicScope napisane jest w czystej Javie – powinno być uruchamialne na każdej platformie obsługiwanej przez HotSpot, więc między innymi Solaris, FreeBSD działające na `ppc64` czy `arm`. Publikacja producenta działają jednak tylko na MacOSie na architekturze `amd64` i Windowsie na `amd64` i `x86`.

Instalator zawiera JRE z OpenJDK w wersji ósmej i producent nie zapewnia możliwości korzystania z innego środowiska.

3.2 *Accessibility*

Programy graficzne napisane w Javie celem zwiększenia swojej dostępności dla osób niepełnosprawnych mogą implementować *Accessibility API* przez `javax.accessibility`. MusicScope korzysta z toolkitu Swing który wspiera to API.

Głównym interfejsem MusicScope jest ekran raportu, który jest bitmapą i nie implementuje *Accessibility API*. Swing jest wykorzystywany tylko w zakładce opcji programu i selekcji ścieżek. Tym samym program XiVero nie jest dostępny dla osób z niepełnosprawnościami.

3.3 Dostępność prawna

3.3.1 Uzyskiwanie

Art. 278 Kodeksu Karnego poświęcony jest kradzieży. W przypadku kradzieży nieszczególnie zuchwałej przewiduje karę pozbawienia wolności od 3 miesięcy do 5 lat. Poniżej znajduje się cytat z § 2:

> Tej samej karze podlega, kto bez zgody osoby uprawnionej uzyskuje cudzy program komputerowy w celu osiągnięcia korzyści majątkowej.

Kiedy program był sprzedawany zdobycie go nieoficjalnymi drogami dystrybucji było więc uznawane za przestępstwo.

XiVero zakończyło swoją działalność w 2022 roku i od tego czasu nie ma oficjalnych źródeł dystrybucji. Czy w takim przypadku uzyskanie tego programu może mieć cel osiągnięcia korzyści majątkowej? Odpowiedź na to pytanie nie jest oczywista.

3.3.2 Rozpowszechnianie

Art. 116 ustawy o prawie autorskim dotyczy rozpowszechniania utworu bez pozwolenia. Poniżej znajduje się cytat z § 1:

> Kto bez uprawnienia albo wbrew jego warunkom rozpowszechnia cudzy utwór [...], podlega grzywnie, karze ograniczenia wolności albo pozbawienia wolności do lat 2.

MusicScope nie wyszczególnia warunków rozpowszechniania, więc jest traktowany jako typowe oprogramowanie na własnościowej licencji.

3.4 Kompleksowość analizy

Raport tekstowy MusicScope zawiera informacje o oprogramowaniu generującym raport, czyli jego nazwę i wersję oraz link do strony producenta.

Raportowane informacje o utworach są przechowywane w tabeli i są to:

1. nazwa pliku,
2. format (PCM/DSD),
3. głębokość bitowa (BD),
4. częstotliwość próbkowania (SR),
5. częstotliwość odcięcia, czyli najwyższa zarejestrowana,
6. TPL dla kanałów;

- lewego,
- prawego,
- *Mid*, czyli średnią sumy lewego i prawego,
- *Side*, czyli średnią różnicy sygnału lewego i prawego,

7. RMS: lewy, prawy, *Mid* i *Side*,
8. średni CREST, czyli stosunek TPL do RMS (aCREST),
9. średnie Peak to Loudness Ratio (aPLR),
10. głośność całkowitą (I-LUFS),
11. zakres głośności.

Na samym dole widnieje niejasny[4] *Total Loudness Range*.

3.5 Ergonomia interfejsu

3.5.1 Przedstawienie interfejsu

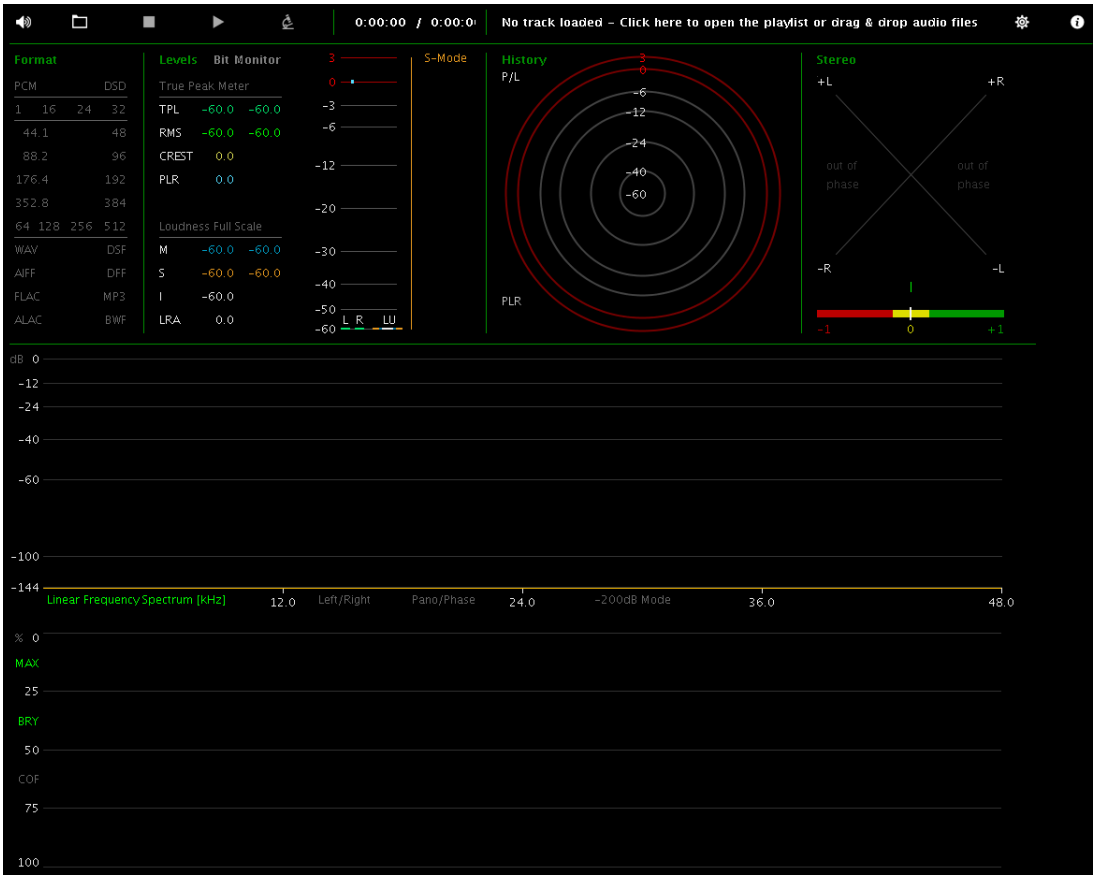
Interfejs MusicScope przedstawia rysunek 1. Jedyną interaktywną jego częścią jest pasek na samej górze zawierający następujące przyciski:

1. głośnik – pozwala wyciszyć program podczas analizy plików w czasie rzeczywistym,
2. folder – pozwala wybrać foldery i pliki do analizy,
3. stop,
4. rozpocznij/wznów analizę w czasie rzeczywistym,
5. mikroskop – rozpocznij/wznów szybką analizę,
6. ustawienia,
7. informacje.

Kliknięcie w pole gdzie później wyświetlana jest nazwa analizowanego pliku otwiera okno zarządzania playlistą[5]. To właśnie tam **przy każdym uruchomieniu** programu należy zaznaczyć *Folder / Album Report* jeśli chcemy otrzymać raport tekstowy.

[4] W przypadku mojej kopii albumu blink182 z 2003 roku MusicScope raportuje 7.8 dB. Zkonkatenowałem i przeanalizowałem te pliki `sd2`. Zaraportował on LRA na poziomie 10.69 dB. Nie wiem jak tę wartość kalkuluje program XiVero.

[5] Tekst następujący po *No track loaded* – jest za długi aby zmieścić się z przeznaczonym na niego miejscu więc wypycha *Ustawienia* i *Informacje* poza normalną ramę okna. Widać to wyraźnie przy porównaniu rysunku 1 do dowolnego zrzutu ekranu oprogramowania z załadowanym utworem.



Rysunek 1: Interfejs MusicScope świeżo po uruchomieniu

Zrzut ekranu okna pojawiającego się po kliknięciu w przycisk *Informacje* przedstawia rysunek 2. Tam kliknięcie w *Software License* otwiera nowe okno z informacjami o licencjach wolnościowych bibliotek z których korzysta projekt, czyli: dekodery ALAC, FLAC i MP3 licencjonowane pod BSD-3 i `jaudiotagger` pod LGPL[6].

Po kliknięciu w *Ustawienia* otwiera się okno z zakładkami. Pod względem generowania raportów tekstowych jedyna użyteczna zakładka to *System*. Można tam suwakami ustawić priorytet wątków i prędkość analizy, kolejno na 11 i 6 poziomów. Domyślnie są one ustawione na maksymalny priorytet i prędkość.

Aby przeanalizować utwór należy oknem wyboru plików wskazać go w systemie plików i rozpocząć analizę *Mikroskopem*. Głównym wynikiem analizy będzie raport graficzny, który jest praktycznie zapisem interfejsu po zakończonej analizie. Taki stan przedstawia rysunek 3.

Po zakończonej analizie albumu program pozostaje w stanie identycznym do tego jak gdyby analizował ostatni utwór; nie komunikuje osiągnięcia celu w żaden sposób.

3.5.2 Przedstawienie użytkowania

MusicScope jest w stanie wygenerować jeden raport na jeden zbiór plików zakolejkowanych do analizy co drastycznie wydłuża czas analizy wielu albumów. Aby lepiej to zilustrować sporządziłem diagram stanów przedstawiający workflow użytkownika. Jest to rysunek 4.

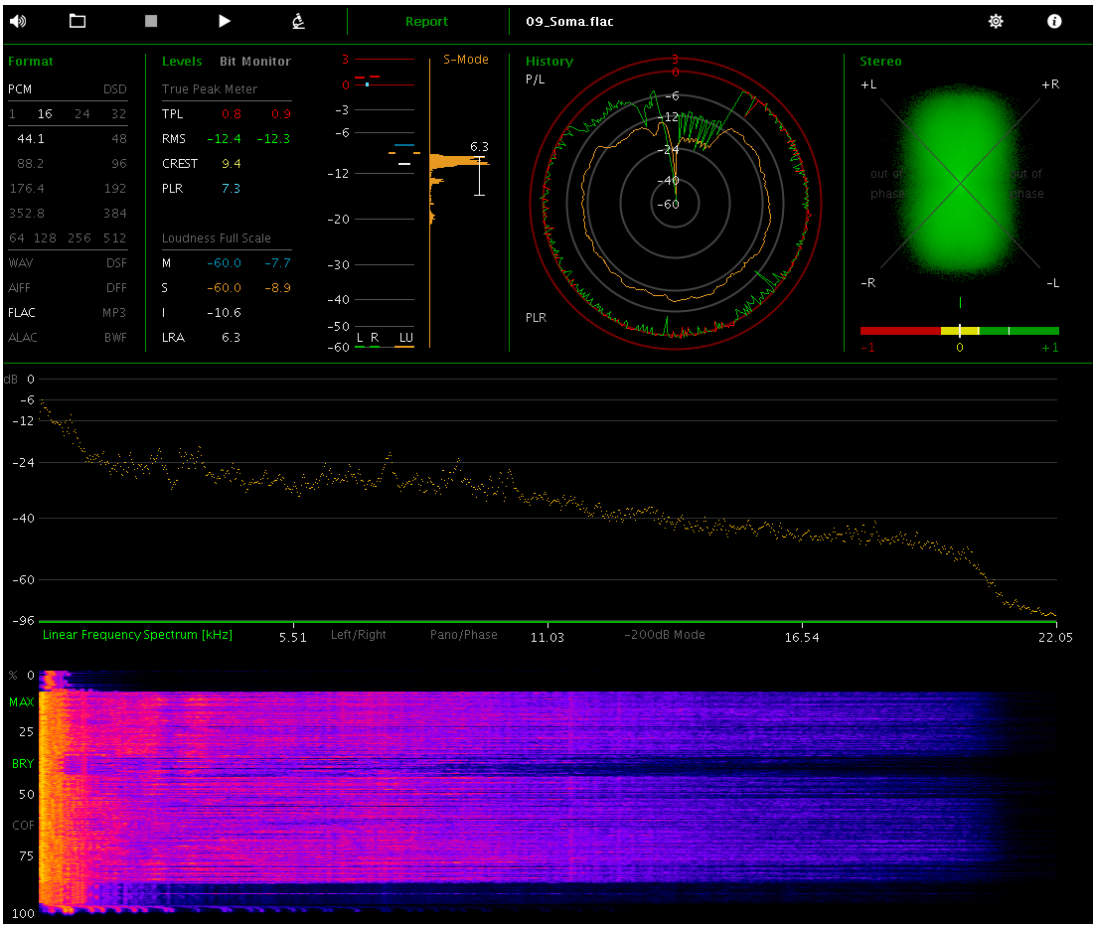
Przeanalizowanie pojedynczego albumu wymaga więc ośmiu akcji:

1. wywołanie programu,
2. kliknięcie folderu,
3. wybór celów analizy,
4. otworzenie okna kolejki,
5. wybranie raportu tekstowego,
6. zamknięcie okna kolejki,
7. rozpoczęcie analizy,
8. zamknięcie programu.

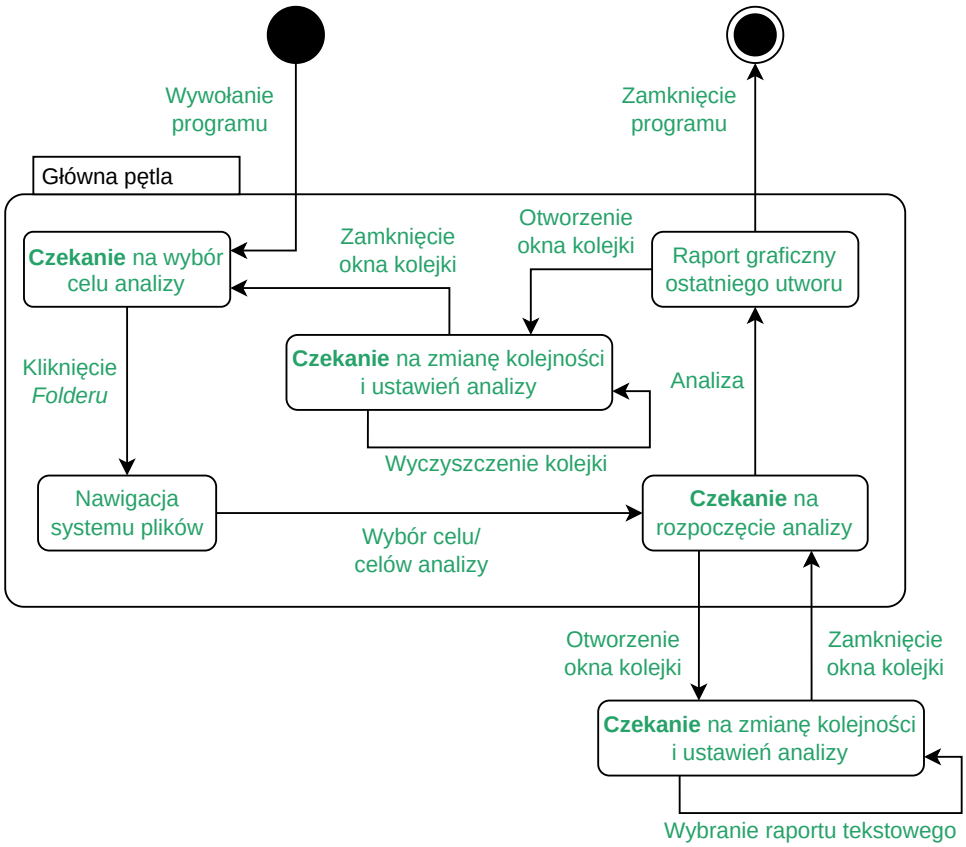
Wybranie raportu tekstowego nie jest zerowane przy wyczyszczeniu kolejki więc może być wybrane raz na uruchomianie programu.

[6] Jest to niezgodne z prawdą. Jaudiotagger w wersji 2.2.6, czyli tej w której jest wykorzystywany w MusicScope, jest licencjonowany pod warunkami LGPL-2.1-or-later, a nie LGPL-2.0*. XiVero złamało warunki licencji tej biblioteki więc ich program nie powstał legalnie.

Rysunek 2: Okno *About* MusicScope



Rysunek 3: Raport graficzny MusicScope utworu *Soma*



Rysunek 4: Diagram stanów MusicScope z perspektywy użytkownika

Przy analizie większej ilości albumów użytkownik pozostaje w ramach stanów, które określiłem *pętlą główną* do momentu zamknięcia programu.

3.5.3 Ocena

Interfejs MusicScope być może wygląda imponująco na pierwszy rzut oka, jednak już po krótkim obcowaniu z programem łatwo zauważalne są jego znaczące wady.

Po pierwsze, brak responsywności. Użytkownicy są przyzwyczajeni do tego, że jeśli ich wskaźnik myszy najedzie na element, który nie wiedzą czy można kliknąć to ten element zmieni swój stan pokazując tym samym możliwość interakcji. W MusicScope żaden interaktywny element interfejsu nie jest responsywny.

Po drugie, aby program tworzył raport tekstowy przy każdym jego wywołaniu należy zaznaczyć tę opcję.

Po trzecie, co najważniejsze, brak obsługi kolejkowania wielu folderów. Oznacza to, że użytkownik chcący przeanalizować n albumów musi n razy przejść przez żmudną *pętlę główną* interakcji.

Po czwarte, brak obsługi folderów jako cele. Chcąc przeanalizować wszystkie pliki w folderze, należy przy użyciu okna dialogowego Swing wybrać wszystkie pliki z albumu. Jest to tym bardziej uciążliwe im więcej utworów ma wydawnictwo i im więcej plików niedźwiękowych (np. okładka, log ripowania) znajduje się w folderze.

3.6 Efektywność analizy

MusicScope jest programem napisanym w czystej Javie z interfejsem będącym bitmapą. Interfejs ten jest aktualizowany wiele razy na sekundę.

Wykonałem na MusicScope testy opisane w sekcji 2.3.3. Czas mierzyłem ręcznie stoperem cyfrowym i wyniki zaokrąglałem do jedności. Wyniki pomiarów przedstawia tabela 2.

Obserwacje dotyczące procesu analizy:

1. co prawda jest on wielowątkowy jednak najwyższe zużycie procesora jakie zaobserwowałem nie przekraczało 65%,
2. w wyniku niezrozumiałego dla mnie błędu przy analizowaniu tych samych albumów ponownie szybkość analizy spadała drastycznie więc musiałem zamykać program aby pozbyć się tego efektu,
3. błąd opisany w poprzednim punkcie występował również przy analizie innych albumów, więc każda analiza oznaczała nowe wywołanie programu.

Zdziwił mnie fakt, że MusicScope analizuje sygnały o różnych przepływnościach z bardzo zbliżoną szybkością. Aby potwierdzić czy na pewno tak jest przeanalizowałem następujący plik:

Tabela 2: Zestawienie szybkości analizy MusicScope wybranych albumów

Album	t_1 [s]	t_2 [s]	t_3 [s]	$t_{śr}$ [s]	v	X [MBps]
HYT	226	225	225	225.(3)	10.1	2.3
Ż63	454	454	453	453.(6)	10.4	2.0
MID	313	316	314	315.(3)	10.3	1.1
PPP	86	90	86	87.(3)	10.5	1.3
RAM	419	420	418	419	10.7	3.6
TTS	258	257	258	257.(3)	10.6	3.5

- SR: 410 Hz,
- BD: 8,
- długość: 90s,
- zawartość: sinusoida o częstotliwości 100 Hz.

Wtedy okazało się, że MusicScope nie obsługuje plików o niestandardowych charakterystykach; program nie zakolejkowywuje pliku i nie komunikuje dlaczego, ani nawet, że tego nie zrobił. Po zmianie SR i BD na RedBookowe dowiedziałem się, że plik w formatach zarówno RIFF WAV jak i FLAC jest analizowany w około 9 sekund.

Wykonane przeze mnie pomiary udowadniają, że szybkość analizy[7] MusicScope nie zależy od przepływności pliku a od czasu jego trwania i jest ona równa około 10.

3.7 Ergonomia raportów

3.7.1 Podświetlanie składni

Raporty tekstowe MusicScope to pliki o nazwie `MusicScope-Report.txt`. To najprostsze pliki tekstowe których przytłaczająca większość oprogramowania nie będzie nawet próbowała interpretować, a więc również podświetlać składni.

Ponadto składnia zapisu tabel jest niestandardowa, więc nawet ręczne włączenie podświetlenia nie polepszy czytelności raportu.

3.7.2 Szerokość tabeli

Tabela w pliku tekstowym będzie czytelna tylko jeśli nie będzie przekraczała szerokości wyświetlanych linii lub jeśli tekst może wykraczać poza wyświetlany obszar, jak komórki w arkuszu kalkulacyjnym. Nie jest to jednak typowe zachowanie i nie powinno się na nim polegać.

Tabela z charakterystykami dźwięku ma szerokość **223 znaków**.

[7] zdefiniowana w sekcji 2.3.3

Jest to niespotykanie szeroka tabela która bardzo rzadko będzie mogła być odczytana bez zmniejszenia rozmiaru wyświetlanego fontu. Dla porównania poniżej wymieniłem najdłuższe akceptowalne długości linijek[8] plików tekstowych w różnych konwencjach:

- 80: klasyczny dalekopis (TTY), Linux C, Haskell, Perl, Go, Ruby, Java…
- 100: Zig i Rust
- 120: C#
- 140: Gentoo `ebuild` (zawiera URLe)

Ponadto kolumna *Track* przechowująca nazwę pliku ma stałą szerokość 33 znaków, niezależnie od długości nazw plików.

3.7.3 Ekscesywna ilość raportowanych danych

Uważam, że wypisywanie charakterystyk *Mid* jest niepotrzebne: użytkownik samodzielnie jest w stanie obliczyć tą wartość. Jeśli zależy nam na podaniu tej wartości to można pominąć oddzielne informacje na temat kanałów.

Poddaję w wątpliwość również zasadność zapisywania charakterystyk *Side*; jest to ciekawa acz mało przydatna charakterystyka.

3.8 Podsumowanie

MusicScope jest programem niedostępnym, ponieważ:

- dostępność technologiczna jest ograniczona,
- dostępność dla osób z niepełnosprawnościami jest bliska zeru,
- dostępność prawna jest bardzo niska:
  - rozpowszechnianie jest nielegalne,
  - legalność uzyskania nie jest pewna.

MusicScope analizuje o wiele więcej charakterystyk dźwięku niż jest zapisywanych w raporcie tekstowym. Tam dużo uwagi poświęcone jest na TPL i RMS: oddzielnie raportowane są informacje na kanał oraz ich sumę i różnicę.

Interfejs MusicScope jest nietypowy i niewspółczesny, a korzystanie z niego niewygodne i niesatysfakcjonujące. Został on zaprojektowany pod analizę pojedynczych albumów więc przy większych ilościach użytkownik zmuszony jest wielokrotne powtarzać te same czynności co jest nieefektywne i niepotrzebnie męczące.

MusicScope analizuje pliki o różnej przepływności w czasie niewiele mniejszym niż jedna dziesiąta ich trwania.

[8] Maksymalna szerokość tabeli jest równoważna z maksymalną szerokością linii w danej konwencji.

Raporty MusicScope są okropne do czytania; tabela jest **zdecydowanie** za szeroka a separatorami nienagłówkowych kolumn są po prostu spacje. Pliki to zwykły tekst bez prób zastosowania jakiegokolwiek markupu.

4 Próba pierwsza: SINMS

W 2024 roku postanowiłem zrobić tak zwany *rewrite from scratch* funkcjonalności MusicScope służącej do generowania raportów. Pierwszą próbę nazwałem SINMS, czyli *SINMS Is Not MusicScope* nawiązując do starej hakerskiej tradycji rekursywnych akronimów.

W tym rozdziale opiszę najważniejsze decyzje projektowe podjęte przeze mnie podczas pracy nad SINMS oraz zaprezentuję jego wydajność i stan obecny.

4.1 Interfejs użytkownika

4.1.1 Usunięcie zbędnych elementów GUI

Jeśli użytkownikowi zależy tylko na raportach tekstowych, więc jeśli analizuje on albumy, to interfejs MusicScope ogranicza się do:

- folderu,
- nazwy pliku / kolejki,
- mikroskopu.

Reszta przycisków jest zbędna.

Te elementy interfejsu przekładają się na raport tekstowy:

1. Format:
    - głębokość bitowa,
    - częstotliwość próbkowania.
2. Poziomy:
    - TPL,
    - RMS,
    - CREST,
    - PLR,
    - I-LUFS,
    - LRA.
3. Wykres amplitud częstotliwości.

Wykres jest jednak nieefektywny: z perspektywy raportu potrzebuje on przechowywać tylko jedną najwyższą częstotliwość podczas gdy przechowuje on wartość amplitudy dla wszystkich możliwych.

Oznacza to, że zupełnie zbędne są:

1. spektrogram,
2. wizualizacja korelacji,
3. wykres kołowy z poziomami,
4. wykres pionowy z poziomem *S-Mode*,
5. część wykresu poziomego opisująca *S-Mode* i *M-Mode*.

Usunięcie ich zmniejszy ilość wymaganych obliczeń i przyspieszy działanie programu.

4.1.2 Zmiana celu interfejsu analizy

W MusicScope celem interfejsu jest wyświetlanie analizowanych charakterystyk w czasie rzeczywistym. W przypadku analizy albumów jest to bezużyteczne ponieważ raportowany jest stan wskaźników na koniec analizy.

Dokładna interpretacja interfejsu podczas analizy wymaga jej zatrzymania, co jest nieakceptowalne jeśli użytkownikowi zależy na jej efektywności.

Próby pobieżnej interpretacji charakterystyk w trakcie analizy będą mało miarodajne ze względu na ich ciągłą zmienność. Nawet jeśli użytkownik skupi się na parametrach które na pewno się nie zmienią, na przykład na wykresie kołowym poziomów początku utworu, nie będzie on miał czasu na interpretację ponieważ analiza utworu wkrótce dobiegnie końca.

Interfejs analizy moim zdaniem powinien co najwyżej powiadamiać użytkownika o jej postępach. Uważam też, że zatrzymanie interfejsu na czas analizy jest akceptowalne a nawet wskazane jeśli użytkownikowi zależy na maksymalizacji efektywności.

4.1.3 Interfejs linii komend (CLI)

Zmiana interfejsu graficznego na ten linii komend znacząco zmniejsza skomplikowanie programu, co w konsekwencji przyspiesza jego działanie.

Korzystanie z programu mogłoby więc wyglądać tak jak na listingu 1.

4.1.4 Deklaratywność

Imperatywne programy CLI, zwane też dialogowymi, są przydatne gdy program ma dużo funkcjonalności. Dobrym przykładem będzie `fdisk` z `util-linux`, którego funkcjonalna część strony `man` ma około trzysta linijek. W przypadku prostszych narzędzi

```
$ sinms
Press h to see the list of available commands
> h
cd PATH -- change directory to PATH
a PATH -- start analysis. If no PATH provided uses current directory
q -- quit
> cd Music/Angels_and_Airwaves
> a Chasing_Shadows
Analysed Overload.flac
Analysed Artillery.flac
Analysed Voyager.flac
Analysed Chasing_Shadows.flac
Text report saved!
> q
$ file Music/Angels_and_Airwaves/Chasing_Shadows/report.txt
report.txt: ASCII text
```

Listing 1: Mockup wykorzystania imperatywnego interfeju linii komend sinms

lepiej sprawdza się interfejs deklaratywny który zbiera wszystkie informacje od użytkownika przed wywołaniem programu. Przykładem takiego narzędzie będzie na przykład `touch` z GNU Coreutils, którego funkcjonalna część strony `man` ma zaledwie 70 linijek.

Zastosowanie deklaratywnego interfejsu oznacza też, że z perspektywy użytkownika program staje się bezstanowy. Użytkownik wywołuje oprogramowanie zapewniając mu wszystkie informacje wymagane do działania, a program wykonuje swoje akcje i wyłącza się. Znacząco zwiększa to ergonomię pracy z narzędziem zwiększając wygodę i efektywność.

Dodatkową zaletą jest łatwa skryptowalność zadania: wywołanie programu z poziomu skryptu powłoki niczym nie różni się wtedy od wywołania go przez użytkownika.

4.1.5 Podsumowanie

Interfejs użytkownika SINMS implementuje cztery zmiany, które wymieniłem w poprzednich sekcjach. Listing 2 przedstawia przykładowe wykorzystanie tego interfejsu w jego najnowszej wersji[9].

4.2 Język programowania i biblioteki

Zależało mi na maksymalnej efektywności mojego programu więc wybrałem najbardziej efektywny znany mi język: C, a konkretnie standard 2x, który okazał się być standardem C23 [14].

[9] Więcej detali zawiera sekcja 4.5.

```
$ sinms --help
Usage: sinms [OPTION]... [TARGET]...
Analyzes audio in TARGET(s)

  -e, --explain-headers include explanation of header-meanings in reports
  -p, --print           instead of reporting to file, print output
  -r, --recursive       analyze all folders within a folder
  -s, --silent          suppress all output
  -v, --verbose         explain what is being done
  -l, --lazy            do not analyze a folder if has a report present
      --help            display this help and exit
      --version         output version information and exit
TARGET(s) can be either files or directories
$ sinms -v Music/Angels_and_Airwaves/Chasing_Shadows/
INFO: File "01_Overload.flac" analyzed
INFO: File "02_Artillery.flac" analyzed
INFO: File "03_Voyager.flac" analyzed
INFO: File "04_Chasing_Shadows.flac" analyzed
INFO: A report was generated to "Music/Angels_and_Airwaves/Chasing_Shadows/"
$ file Music/Angels_and_Airwaves/Chasing_Shadows/report.org
report.org: ASCII text
```

Listing 2: Przykładowe wykorzystanie SINMS

Wiedziałem, że szybkie programy których używam na co dzień takie jak PipeWire [26] czy MPD [22] wykorzystują bibliotekę sndfile [6] do odczytywania plików audio więc postanowiłem z niej skorzystać.

Podczas testowania prototypów SINMS zauważyłem, że amplituda nigdy nie przekraczała wartości 1 (0 dB), nawet jeśli MusicScope mówiło inaczej. Dowiedziałem się wtedy o Inter-Sample Peaks i zacząłem korzystać z biblioteki ebur128 [17], która implementuje Rekomendację 128 Europejskiej Unii Nadawców [18] wykorzystujący wspomniany wcześniej standard ITU [2]. Tym samym moje oprogramowanie jest również zgodne z dokumentami EBU Tech 3341 i EBU Tech 3342 ponieważ to właśnie ta biblioteka liczy większość charakterystyk dźwięku.

Początkowo planowałem aby SINMS mógł działać również na Windowsie jednak oznaczałoby to na przykład rezygnację z korzystania z POSIXowego `getopt.h`. Stwierdziłem, że nie zależy mi na tym na tyle, aby trudzić się szukaniem rozwiązania tego problemu. Później okazało się, że na Windowsie nie ma również `dirent.h`. Utwierdziło mnie to w przekonaniu, że SINMS nie będzie wspierało systemów niePOSIXowych.

4.3 Raportowanie

W SINMS postanowiłem również naprawić problemy z ergonomią raportów które opisałem w sekcji 3.7.

Przede wszystkim postanowiłem wykorzystać język markupu tekstu. Wybierałem między CommonMark [20] a Emacsowym `org-mode`; obydwa to lekkie języki które są używane od ponad 20 lat. Zdecydowałem się wybrać ten drugi ze względu na wbudowaną obsługę metadanych która polepszy czytelność pliku. Zmniejszyłem również szerokość tabeli z 223 do 107 znaków tym samym zapewniając jego zgodność z wytycznymi dla języka C#. Inną ważną zmianą było usunięcie charakterystyk *Mid* i *Side*, w powodów opisanych we wcześniej wspomnianym rozdziale.

4.4 Efektywność działania

SINMS to program napisany w czystym C. Wykorzystywane przez niego biblioteki zostały skompilowane zgodnie z systemowymi `CFLAGS`, a sam plik wykonywalny z poziomem `-O3`.

Mierzyłem efektywność programu przypisanego do konkretnej jednostki procesującej przy użyciu `taskset` i `time`. Wyniki pomiarów przedstawia tabela 3

Ponieważ SINMS jest jednowątkowe, zajmowało tylko jeden rdzeń logiczny mojego procesora; maksymalne obciążenie systemu wynosi więc 12,5%.

Tabela 3: Zestawienie szybkości analizy SINMS wybranych albumów

Album	t_1 [s]	t_2 [s]	t_3 [s]	$t_{śr}$ [s]	v	X [MBps]
HYT	27.48	28.12	28.72	28.11	81	18.14
Ż63	58.73	58.69	60.80	59.41	79	15.25
MID	40.67	40.69	40.11	40.49	80	8.69
PPP	9.25	8.91	8.71	8.96	102	12.39
RAM	99.95	97.67	100.13	99.25	45	15.11
TTS	49.85	48.94	48.41	49.07	55	20.38

Zdziwiła mnie rozbieżność v więc powtórzyłem analizę trzech ostatnich albumów ponownie otrzymując podobne czasy.

Różne czasy analizy plików są dowodem tego, że wąskim gardłem MusicScope było aktualizowanie interfejsu; dopiero po usunięciu go zauważalne były różnice w plikach.

SINMS wykorzystując ćwierć czasu obliczeniowego CPU zużywanego przez MusicScope jest od niego od czterech do dziesięciu razy szybsze.

4.5 Status quo SINMS

SINMS osiągnęło pełną funkcjonalność pod koniec lipca 2024 roku. Od tego czasu korzystałem z niego zamiast z MusicScope ponieważ już wtedy uznałem moje starania za udane. Mój program był o wiele wygodniejszy i o wiele szybszy oraz generował lepsze raporty.

Ostatnią poprawkę wcieliłem do repozytorium szóstego sierpnia 2024 roku i od tego czasu kod programu pozostał bez zmian.

Wiedziałem, że następnym krokiem rozwoju aplikacji będzie wielowątkowość. Dowiedziałem się, że najlepiej będzie abym zaimplementował ją przy użyciu POSIXowej biblioteki `pthread.h` która nie doda zależności mojemu programowi. Działa ona jednak na stosunkowo niskim poziomie co zniechęciło mnie, szczególnie ponieważ na potrzeby SINMS napisałem już tysiące linijek kodu C.

Rozważałem jeszcze zastosowanie OpenMP [24] jednak poza wprowadzaniem dodatkowej zależności opiera się ono na preprocesorze. Preprocesor został dodany do języka C rok po jego pierwszym wydaniu [32] jako próba obejścia istotnych ograniczeń samego języka. Jedną z funkcjonalności, którą umożliwił były makra. Pozwalają one na wiele, ale:

1. ciężko się je debuguje,
2. nie są typowane,
3. często mają (nieoczekiwane) efekty uboczne.

Kiedy programuję w C staram się więc unikać makr jeśli jest to możliwe.

4.6 Haskellowe SINMS

Haskell to czysto funkcjonalny (*purely functional*) język programowania, więc nie ma w nim efektów ubocznych.

O SIMNS można myśleć jak o funkcji: przyjmuje ona na wejściu cele i flagi (opcje) a zwraca przeanalizowane dane. Ponieważ chciałem nauczyć się funkcjonalnego języka programowania pomyślałem, że przepisanie mojego programu do tego języka będzie ciekawym i rozwojowym wyzwaniem.

Udało mi się zintegrować `libsndfile` za sprawą biblioteki zapewniającej Haskellowe powiązania. Niestety dla `libebur128` nikt nie napisał jeszcze tych powiązań. Pisanie takich powiązań bez znajomości języka to bardzo czasochłonne i wymagające stwierdziłem, że Haskell nie jest odpowiednim językiem programowania dla mojego projektu.

5 Próba druga: `sd2`

Największym problemem SIMNS był dla mnie brak wielowątkowości i fakt, że nie chciałem jej implementować w C. Głównym celem `sd2` było więc przepisanie SINMS w języku programowania systemowego, który wspiera wielowątkowość na poziomie swojej biblioteki standardowej. Nie potrafię programować w żadnym języku który spełnia te wymagania, więc personalnie oznaczało to nauczenie się zupełnie mi obcego języka programowania.

5.1 Język programowania i biblioteki

Zależało mi na tym aby skorzystać z tych samych bibliotek, które wykorzystałem w SINMS. Pozostawiło mi to wybór trzech języków programowania: C++, Rusta i Ziga. C++ w oczywisty sposób wspiera biblioteki C, Zig robi to na niemalże równie wysokim poziomie a Rust ma powiązania do `libsndfile` i swój odpowiednik `libebur128`.

5.1.1 C++

Od publikacji standardu C++11 język ten obsługuje współbieżność na poziomie biblioteki standardowej i z oczywistych powodów jest w dużym stopniu kompatybilny z C.

Prace nad językiem znanym dzisiaj jako C++ rozpoczęły się w 1979. Wtedy nazywany był jeszcze "C z klasami", co pokazuje jakimi wartościami kierował się Bjarne Stroustrup przy jego tworzeniu. Obiektowość znacząco komplikuje język programowania i nie jest potrzebna w moim programie więc wolałem jej uniknąć. Ponadto nie chciałem korzystać z języka programowania który ma ponad czterdzieści lat i dalej ma problemy C. Pomijając wspomniany wcześniej procesor nadal budowa nietrywialnych projektów wymaga dodatkowego oprogramowania z własną składnią itd. (GNU autotools, Meson, CMake, Bazel, Waf...).

Bardzo zaintrygował mnie cytat który znalazłem w internecie szukając informacji na ten temat [15]:

> C++: jest pełen błędów, jest nieefektywny, jest niekompletny, jest *bloated*, kompiluje się o rząd wielkości dłużej niż C, *offloaduje* połowę implementacji języka do konsolidatorów czego konsekwencją są enigmatyczne i niemożliwe do rozwiązania problemy pojawiające się tak późno w procesie tworzenia oprogramowania, że chce się implementować lovecraftowskie obejścia problemu niszcząc swoją poczytalność do momentu w którym korzystanie z pomocy Sanapii zdaje się być normalne... Ponadto fiasko kolejności statycznej inicjalizacji wydaje się być bardziej funkcjonalnością niż błędem. Nie ma stabilnego ABI lecz jego użytkownicy błędnie wierzą, że jest inaczej. C++ wstydzi się danych zamiast wstydzić się własnej implementacji wyjątków.

Zgłębiłem temat i przekonałem się, że te zarzuty zdecydowanie nie są bezpodstawne. Wtedy podjąłem decyzję, że nie skorzystam z tego języka programowania.

5.1.2 Rust

Rust powstał w 2006 roku a wydanie 1.0.0 miało miejsce w 2015 roku. Nie jest on obiektowy i nie wykorzystuje garbage-collectora co również mi się podoba. Jedną z najważniejszych cech Rusta jest bogaty system typów który ma zapewnić bezpieczeństwo na poziomie wątków i pamięci (*thead-safety* i *memory-safety*).

Ciężko mi było nie słyszeć o Ruście ze względu na rewrite'y klasycznych narzędzi jak `cat` [25] `sudo` [35] czy nawet całego `coreutils` [13]. Poza tym byłem świadomy efektywności narzędzi takich jak `ruff` [3], `paru` [19] i `alacritty` [1] a nawet zawarcie go w jądrze Linux[10]!

Kanał No Boilerplate Trisa Oatena na YouTube był kolejnym i chyba najbardziej wpływowym źródłem informacji na ten temat[11]. Po moich niedawnych doświadczeniach z Haskellem funkcjonalne elementy Rusta zachęcały mnie jeszcze bardziej. Menedżer paczek i system budowy `cargo` również mi się podobał: nie musiałem uczyć się kolejnego języka, aby móc skompilować swoje oprogramowanie.

Z drugiej strony Rust korzysta jednak z makr, tak samo jak pięćdziesięcioletnie C... Są one oczywiście o wiele lepsze, ale dalej są to makra.

Nie byłem również przekonany, czy bezpieczeństwo pamięciowe jest dla mnie tak ważne. Oczywiście pisząc SINMS tworzyłem tego typu błędy; przy pierwszym `segmentation fault` nawet celebrowałem to doświadczenie, tak integralne dla programowania w C. Błędy te nie są jednak krytyczne dla mojego programu więc może lepiej byłoby gdybym wybrał język programowania, którego główny cel jest dla mnie bardziej znaczący?

Rust zdecydowanie jest dobrym wyborem do napisania aplikacji do analizy albumów jednak nie byłem pewien czy jest on doskonałym wyborem dla mnie.

5.1.3 Zig

Zig po raz pierwszy pojawił się w 2016 roku i jeszcze nie miał swojej wersji 1.0.0. Podobnie jak Rust nie jest obiektowy i nie wykorzystuje garbage-collectora, jednak w przeciwieństwie do Rusta pamięcią trzeba zarządzać ręcznie.

O Zigu po raz pierwszy usłyszałem w ramach prezentacji *Zig w 100 sekund* od Fireship jeszcze w 2023 roku. Wtedy postrzegałem Ziga jako C bez preprocesora, bez ukrytego przepływu sterowania ale z `defer`.

Przypomniałem sobie o Zigu w marcu 2025 roku niedługo po wydaniu wersji 0.14. W międzyczasie widziałem w internecie następujące zdanie:

> Jeśli Rust jest ulepszeniem dla użytkowników C++, Zig jest ulepszeniem dla użytkowników C.

Po sprawdzeniu, że biblioteka standardowa obsługuje wielowątkowość postanowiłem, że spróbuję zrobić w nim minimalny prototyp (*Proof of Concept*) celem stwierdzenia, czy jest sens zmiany języka programowania.

Zig mi się spodobał więc postanowiłem zreimplementować cały swój program w tym właśnie języku programowania.

Podczas pracy poznałem więcej zalet Ziga które wymienię w następnym rozdziale

[10] `https://lore.kernel.org/lkml/202210010816.1317F2C@keescook/`

[11] Największe wrażenie wywarł na mnie film Rust: Your code can be PERFECT.

5.2 Zalety Ziga

Zig został stworzony jako próba naprawienia największych problemów języka C bez dodawania wielu funkcjonalności [16].

5.2.1 Pełna kompatybilność z C

Używanie bibliotek C w Zigu jest niemalże tak proste jak używanie ich w samym C – wystarczy zaimportować je z użyciem `@cImport` i `@cInclude`.

Największą wadą korzystania z bibliotek C jest fakt, że zazwyczaj prefiksują swoje publiczne funkcje i zmienne, co odrobinę zmniejsza czytelność kodu Ziga.

Na listingu 3 zamieściłem fragment kodu mojego programu odpowiadający za obliczenie TPL. Importuję tam `ebur128.h` i umożliwiam bezpośrednie odwoływanie się do jego symboli przez prefix `r128`. Aby poprawić czytelność kodu zdefiniowałem również stałą `r_ok`.

```
const r128 = @cImport(@cInclude("ebur128.h"));
const r_ok = r128.EBUR128_SUCCESS;
...
var state: ?*r128.ebur128_state = undefined;
var tmp_double: f64 = undefined;

state = r128.ebur128_init(
    bi.getChannels(),
    bi.samplerate,
    r128.EBUR128_MODE_TRUE_PEAK
);
defer r128.ebur128_destroy(&state);

if (r128.ebur128_add_frames_float(state, bi.samples.ptr, bi.frames) != r_ok)
    log.err("Allocating frames went wrong", .{});

const calc_rval: i32 = r128.ebur128_true_peak(state, i, &tmp_double);
if (calc_rval != r_ok)
    log.err("TPL Calculations went wrong (r128 code {d})", .{calc_rval});
```

Listing 3: Fragment `track_info.zig` ukazujący użycie biblioteki C `ebur128`

5.2.2 Koncept `comptime`

Jedną z najpotężniejszych cech Ziga jest możliwość wykonywania kodu w czasie kompilacji. Jest to możliwe ze względu na to, że kompilator traktuje wszystko co interpretuje jako wyrażenie. Pozwala to między innymi na pozbycie się makr z pełnym zachowaniem jego funkcjonalności. Dalej w tym rozdziale zaprezentuje te przypadki użycia na

przykładach z prawdziwego życia.

1. **Operowanie na stałych**

   Chciałem aby w moim oprogramowaniu można było sprawdzić wersje zarówno przez stringa jak i za pomocą liczb całkowitych, podobnie jak można to zrobić w Pythonie przy pomocy `sys.version` i `sys.version_info`.

   W Zigu po prostu definiuję stałe liczby całkowite i korzystam z funkcji biblioteki standardowej `std.fmt.comptimePrint` aby w czasie kompilacji stworzyć stringa, co pokazuje listing 4.

```
pub const major = 1;
pub const minor = 2;
pub const patch = 3;

pub const full comptimePrint("{}.{}.{}", .{ major, minor, patch });
```

Listing 4: Deklarowane wersji w Zigu

   W czystym C można tylko zdefiniować oddzielnie liczby całkowite i string lecz jest to niepotrzebnie uciążliwe do zmiany.

   Aby aktualizacja wersji nie była uciążliwa możemy więc skorzystać z preprocesora, na przykład jak pokazuję na listingu 5. Kosztem tej wygody jest wprowadzenie praktycznie drugiego języka programowania i przyrost objętości kodu o 150%!

```
#define MAJOR 1
#define MINOR 2
#define PATCH 3
const int VERSION_MAJOR = MAJOR;
const int VERSION_MINOR = MINOR;
const int VERSION_PATCH = PATCH;

#define STR_HELPER(x) #x
#define STR(x) STR_HELPER(x)
#define VERSION_STRING STR(MAJOR) "." STR(MINOR) "." STR(PATCH)
const char* VERSION_STR = VERSION_STRING;
```

Listing 5: Deklarowanie wersji w C

   Ponieważ działanie z preprocesorem jest nieintuicyjne można zaprząc do tego system budowy. Tak proste zadanie moim zdaniem nie powinno wymagać wykorzystania tak złożonego oprogramowania.

2. **Programowanie generyczne**

W `sd2` korzystam z biblioteki C `ebur128` i tam funkcja `add_frames` jest tworzona w czasie kompilacji dzięki programowaniu generycznemu. Osiąga to nieczytelne i nieprzyjazne dla IDE makro, które przedstawia listing 6 podczas gdy Zig robi to idiomatycznie, estetycznie i zrozumiale, co widać na listingu 7.

```
#define EBUR128_ADD_FRAMES(type)          \
  int ebur128_add_frames_##type(          \
    ebur128_state* st,                    \
    const type* src,                      \
    size_t frames                         \
  ) {                                     \
    size_t src_index = 0;                 \
    unsigned int c = 0;                   \
    for (c = 0; c < st->channels; c++) {  \
      st->d->prev_sample_peak[c] = 0.0;   \
...
```

Listing 6: Generyczna funkcja `add_frames` z `ebur128.c`

```
fn add_frames(
    comptime T: type,
    st: *state,
    src: const T*,
    frames: usize
)
```

Listing 7: Generyczna deklaracja funkcji `add_frames` w Zigu

5.2.3 Instrukcje `defer` i `errdefer`

Te dwie funkcjonalności są inspirowane funkcjonalnością języka Go [16]. Pozwalają one na wykonanie pewnej procedury z końcem zasięgu widoczności (ang. *scope*) bloku w które została zadeklarowana. `errdefer` wywołuje swoją procedurę gdy poprzedzające go `try` nie powiedzie się.

Moim zdaniem znacząco ułatwiają one realizowanie procedur, które mają określony początek i koniec, jak na przykład zarządzanie pamięcią. Jest to szczególnie ważne co pokazuje pseudokod C i Ziga na listingach 8 i 9 które mają obliczyć i wypisać RMS danego pliku. Im bardziej skomplikowane są owe procedury tym większy jest zysk.

5.2.4 Przenośność

W sekcji 5.1 wspomniałem, że pierwotnie miałem zamiar aby moje oprogramowanie działało również na Windowsie. Zależało mi również na tym, aby wspierał platformy RISC-V i ARM64 ze względu na ich rosnącą popularność w urządzeniach konsumenckich.

```
infile = sf_open(file_path);
buffer = malloc();
if (buffer == NULL) {
  // obsługa błędu
  sf_close(file_path);
}
sf_readf(infile, buffer);
if (infile == NULL) {
  // obsługa błędu
  free(buffer);
  sf_close(file_path);
}
// wyliczenie i wyświetlenie RMS
free(buffer);
sf_close(file_path);
```

Listing 8: Pseudokod C: kalkulacja i wyświetlenie RMS

```
infile = sf_open(file_path);
defer sf_close(file_path);

const buffer = try alloc(<rozmiar>);
defer free(buffer);

try sf_readf(infile, buffer);
errdefer free(buffer);
// wyliczenie i wyświetlenie RMS
```

Listing 9: Pseudokod Ziga: kalkulacja i wyświetlenie RMS

Kod Ziga jest w pełni przenośny między systemami operacyjnymi i architekturami. Poziom przenośności jest naprawdę imponujący: Zig wspiera takie architektury jak `m68k`, `loong`, `mips` czy `thumb` i środowiska jak Haiku, UEFI, Serenity czy VisionOS. Oznacza to, że mój projekt skonsolidowany dynamicznie jest ograniczany przez biblioteki C których używa, czyli przez ANSI `sndfile` a bardziej **C99** `sndfile`.

5.2.5 Lepsze doświadczenie tworzenia oprogramowania

W języku Zig 0.14 i 0.15 pisało mi się o wiele lepiej niż w języku C w standardzie C2x.

Przede wszystkim podobało mi się uproszczenie ręcznego zarządzania pamięcią przez wykorzystanie alokatorów. Początkowo korzystałem ze standardowego alokatora C, czyli `std.heap.c_allocator` i zarządzałem pamięcią dokładnie w ten sam sposób jak w SIMNS. Kiedy zacząłem korzystać z `std.heap.GeneralPurposeAllocator`, który przy podstawowym trybie kompilacji jest w istocie `std.heap.debug_allocator`, okazało się, że moja aplikacja ma całkiem dużo wycieków pamięci! Później pomógł mi też w niekorzystaniu z pamięci po jej zdealokowaniu. Bardzo cieszę się, że Zig zapewnia również te funkcje, które dla C zapewnia kolejne oprogramowanie (np. Valgrind [36]).

Bardzo podobały mi się również:

- czyste wyróżnienie operacji kompilatora składnią `@foo()`,
- wbudowane logowanie, zarówno w czasie kompilacji jak i wykonania,
- domyślna zawartość śladu stosu i *error return trace*[12],
- w porównaniu do GCC o wiele bardziej czytelne błędy kompilacji.

5.3 Rozwój `sd2`

Zanim usiadłem do programowania w Zigu postanowiłem stworzyć okrojoną wersję SINMS która będzie analizowała plik i wyświetlała wyniki analizy w konsoli. Zmniejszyło to ilość linijek kodu (ang. *Lines Of Code*, LOC) o 80%.

5.3.1 Porównanie do SINMS

Wersja 0.0.2 miała w sobie wszystkie funkcjonalności *dema* SINMS zawierając 30% więcej kodu. Wersja 0.1.1 działała na systemach nie tylko POSIXowych, miała wszystkie funkcjonalności SINMS i ich testy jednostkowe **mając nieznacznie mniejszą ilość LOC**.

[12] Oryginalny i efektywny sposób na pokazanie tego skąd pochodzą błędy w kodzie i jak rozchodzą się w programie.

5.3.2 Pierwsza biblioteka Ziga

W wersji 0.1.0 wprowadziłem możliwość zapisu czasu w raporcie timestampem nie unixowym (`1000000000`) a HTTP (`2001.09.09 13:46:40`). Tę integrację z biblioteką `zig-time` [9] można wyłączyć kompilując program z flagą `-Dwith_time=false`.

5.3.3 Biblioteka `yazap` i model open-source

W tej samej wersji zacząłem też wykorzystywać `yazap` [7] do tworzenia kompleksowego interfejsu linii komend. Spełniał on wszystkie moje wymagania poza jednym: nie wspierał określania wielu argumentów pozycyjnych (*positional arguments*) w deklaratywny sposób.

Dodałem tą funkcję lokalnie i w duchu oprogramowania open-source zaproponowałem swoje zmiany twórcom biblioteki. Zostały one przyjęte i `sd2-0.3.1` wykorzystuje funkcje `Arg.multiValuesPositional` z globalnie dostępnej kopii biblioteki.

Końcowo mój interfejs spełnia wytyczne standardu POSIX.1-2024 [4].

5.3.4 `small_float`

W wersji 0.1.2 zmieniłem dokładność zapisu efektów analizy (`trackFloat`) z `f32` do `f16`, czyli z około 7 do 3 cyfr po przecinku [10]. Tę opcję można zmienić ustawiając opcję kompilacji `-Dsmall_float` na `false`. Domyślnie program używa szesnastu bitów ponieważ w tabeli nie ma miejsca na więcej niż dwa miejsca po przecinku.

5.3.5 Nowe `TrackInfo`

W wersji 0.2.0 zoptymalizowałem najważniejszą strukturę: `TrackInfo`. Jest ona tak ważna ponieważ przechowuje rezultaty analizy poszczególnych utworów.

Jej starą wersję przedstawia listing 10.

Aby uprościć obliczenie jej rozmiaru, wprowadzę następujące oznaczenia:

- S – rozmiar `usize` (`size_t` z C),
- t_f – rozmiar `trackFloat`
- r_s – `@sizeOf(TrackInfo)` dla `<sd2-0.2.0`,
- r_s – `@sizeOf(TrackInfo)` dla `>=sd2-0.2.0`.

Przy domyślnych ustawieniach kompilacji architektury 64bitowej, rozmiary te przyjmują następujące rozmiary (podane w bajtach):

- $S = 4$
- $t_f = 2$

```
pub const TrackInfo = struct {
    file_path: []const u8,

    channels: u4,
    bit_depth: u7,
    samplerate: u21,

    iLUFS: trackFloat,
    LRA: trackFloat,
    TPLs: []trackFloat,
    RMSs: []trackFloat,
...
```

Listing 10: Struktura `TrackInfo` w wersjach <0.2.0

Oznacza to, że r_s będzie równy 32B:

$$r_s = 2S + 4 + 2f_t + 4S = 32$$

Zainspirowany *A Practical Guide to Applying Data Oriented Design* postanowiłem zmienić swoją strukturę tak, aby obsługiwała maksymalnie dźwięk stereo zastępując dwa grube wskaźniki na cztery pola `trackFloat`. To zmniejszyło rozmiar `TrackInfo` o niemalże czterdzieści procent:

$$r_n = 2S + 4 + 2f_t + 4f_t = 20$$

5.3.6 Standard REUSE

W wersji 0.4.0 zmieniłem styl licencjonowania swojego oprogramowania z klasycznego pliku `LICENSE` na standard REUSE 3.3 [31]. Jest to inicjatywa Europejskiej Fundacji Wolnego Oprogramowania (FSFE) mająca uprościć licencjonowanie oprogramowania.

Jest to niezmiernie ważny temat, który dla małych projektów może wydawać się wręcz pomijalny. Ja doświadczyłem tego jak uciążliwe mogą być kwestie licencyjne kiedy chciałem zaktualizować paczkę Signala w Gentoo i napotkałem ponad dwudziestojednotysięcznolinijkowy plik ACKNOWLEDGMENTS.md.

Przyznam również, że chciałem skorzystać z usługi API REUSE, ponieważ w tym czasie pracowałem w FSFE i *de facto* byłem menedżerem zespołu który miał ją zmodernizować.

5.3.7 Zmiany biblioteki standardowej Ziga

Zig 0.15 wprowadził najwięcej zmian łamiących kompatybilność od wersji 0.9.0 z 2021 roku. Jedną z nich jest *Writergate* [39] które wycofuje (*deprecates*) wszystkie implemen-

tacje operacji I/O z `std.Io` na rzecz niegenerycznych `std.io.Reader` i `std.Io.Writer`.

Sama lista zmian opisuje te zmiany jako *extremely breaking* i w sekcji *Motywacja* odsyła do seminarium *Don't Forget to Flush* z *Systems Distributed 2025*. Po obejrzeniu tego wystąpienia byłem przekonany, że jest to dobry eksperyment i cieszyłem się, że deweloperzy podjęli taką decyzję – w najgorszym wypadku będzie to dowód, że jest to zła ścieżka. Wspomniany wcześniej ekstremalny stopień zepsucia funkcjonalnego kodu odczułem znacząco ponieważ zapis i odczyt plików jest krytyczny dla mojego programu.

Uważam, że jest to jeden z "uroków"korzystania z pół-ezoterycznego przez swoją innowacyjność języka programowania i z tego powodu postanowiłem wspomnieć o tym w tej pracy.

5.3.8 Plik README

Pisanie pliku README odłożyłem na sam koniec procesu tworzenia oprogramowania. Jako haker i miłośnik wolnego oprogramowania widziałem wiele takich plików i nie napotkałem jeszcze takiego, który w pełni mi się podobał.

Zanim zacząłem korzystać z qBitTorrenta [30], który *nota bene* teraz utrzymuję w Gentoo, korzystałem i wspierałem BiglyBT [5]. Tam właśnie po raz pierwszy miałem styczność z naukowym podejściem do pisania tych plików gdy pewien doktorant Uniwersytetu w Melbourne zaproponował ulepszenie README projektu. Nie zgadzałem się z zaproponowanym przez niego wzorem ale po raz pierwszy mogłem wskazać i uzasadnić dlaczego.

Przeczytałem wiele nieautorytatywnych wpisów na blogach na ten temat i parę artykułów naukowych. Jeden z nich [27] zdefiniował siedem przykładowych kategorii sekcji:

1. Co – kontekst i wstęp,
2. Dlaczego – zalety projektu,
3. Jak – instalacja i użytkowanie,
4. Kiedy – status, wersje i plan na przyszłość,
5. Kto – drużyna, społeczność, kontakt itd.,
6. Odwołania – dokumentacja, link do wsparcia, tłumaczenia
7. Wkład – sposób na wsparcie projektu.

Tak oceniam ich przydatność w moim projekcie:

1. Co – chciałem opisać czym jest moje oprogramowanie i co robi,
2. Dlaczego – jako ciekawostkę spisałem genezę projektu,

3. Jak – instalacja i kompilacja są oczywiste lecz udokumentowałem również opcje kompilacji,
4. Kiedy – mam oddzielny plik do spisywania zdań a do innych funkcji wykorzystuję funkcjonalności platformy hostingowej,
5. Kto – fakt, że to solowy projekt jest oczywisty i nie zależy mi na rozpoznawalności,
6. Odwołania – nie mam do czego się odwołać,
7. Wkład – nie zależy mi na wkładzie innych ludzi, ale mogę skorzystać z funkcjonalności hostingu.

Jestem raczej zadowolony ze swojego pliku README i doświadczyłem tego jak wymagające jest dobre jego zaprojektowanie i implementacja.

5.4 Efektywność działania

`sd2` jednak korzysta z bibliotek C które zostały skompilowane zgodne z systemowymi `CFLAGS`, podobnie jak SINMS. Sam program został skompilowany z flagą `--release=fast` z kompilatorem `zig-0.15.2`.

Procedura pomiaru czasu wywołania pojedynczego procesu jest identyczna jak w przypadku SINMS a jej wyniki przedstawia tabela 4. Procedura pomiaru czasu wywołania wieloprocesowego pomija użycie `taskset` a jej wyniki znajdują się w tabeli 5. Ponieważ jest to ostatnie pomiar podsumowałem zmierzone przeze mnie czasy na wykresie na rysunku 5.

W idealnych warunkach program ośmioprocesowy na ośmiordzeniowym procesorze powinien być osiem razy szybszy niż program jednoprocesowy. Moje pomiary są kolejnym dowodem, że idealne warunki są daleko od rzeczywistości ponieważ mój program nie był szybszy o 700% a zaledwie o 136%.

Jeśli wyłączyłbym hyperthreading idealny przyrost efektywności zmalałby do 300% i zapewne ten realny również by się zwiększył.

Kolejnym kluczowym elementem jest ziarnistość zadań: im liczba plików n jest bliżej wielokrotności liczby rdzeni procesora n_{CPU} tym przyrost będzie większy. W albumach które wybrałem średnia odległość $n \bmod n_{\text{CPU}}$ jest równa $3.(6)$.

Ponadto program będzie najbardziej efektywny jeśli wszystkie procesy skończą swoje zadania jednocześnie. Na przykład analizując 6 identycznych utworów czterema procesami na czterodzieniowym procesorze przy aktualnym poziomie ziarnistości obciążenie procesora byłoby równe około 75%.

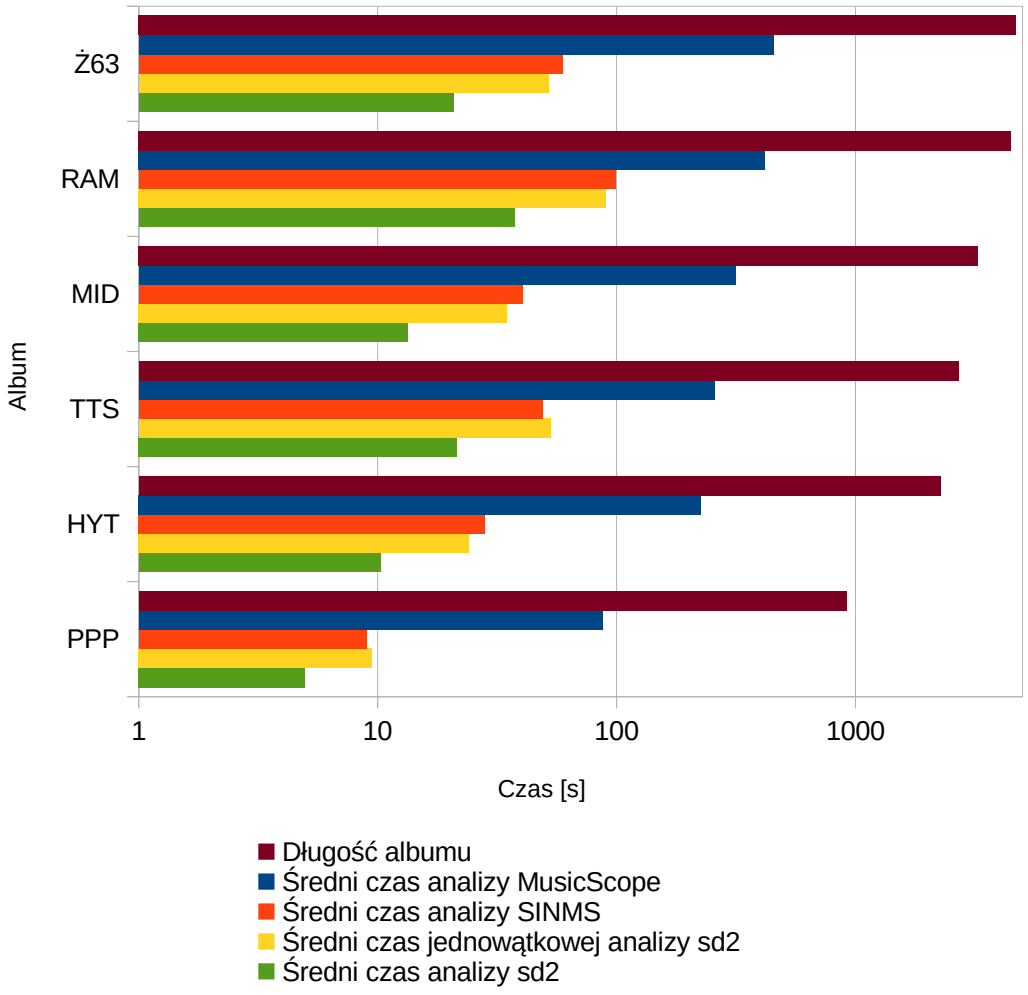
Końcowo jestem zadowolony z tego w jakim stopniu dodanie wielowątkowości zwiększyło efektywność mojego programu i uważam, że implementacja tej funkcjonalności zdecydowanie była tego warta.

Tabela 4: Zestawienie jednowątkowej szybkości analizy `sd2` wybranych albumów

Album	t_1 [s]	t_2 [s]	t_3 [s]	$t_{śr}$ [s]	v	X [MBps]
HYT	22.60	23.97	25.94	24.17	94	21.1
Ż63	52.59	51.39	51.75	51.91	90	17.4
MID	36.24	35.45	32.95	34.88	93	10.1
PPP	9.41	9.39	9.62	9.47	97	11.7
RAM	90.05	90.52	90.43	90.(3)	50	16.6
TTS	52.39	53.61	53.08	53.03	51	18.9

Tabela 5: Zestawienie wielowątkowej szybkości analizy `sd2` wybranych albumów

Album	t_1 [s]	t_2 [s]	t_3 [s]	$t_{śr}$ [s]	v	X [MBps]
HYT	10.45	10.25	10.17	10.29	220	49.6
Ż63	21.00	20.75	20.68	20.81	226	43.5
MID	13.40	13.35	13.38	13.38	244	26.3
PPP	4.98	4.97	4.92	4.96	185	22.4
RAM	38.45	36.66	37.29	37.47	119	40.0
TTS	19.32	21.61	23.53	21.49	126	46.5



Rysunek 5: Podsumowanie czasów

5.5 Status quo `sd2`

Korzystałem z `sd2` od kiedy osiągnęło funkcjonalność SINMS, czyli od kwietnia 2025 roku. W drugiej połowie tego roku program osiągnął swoją pełną funkcjonalność. W styczniu 2026 roku, po ekstensywnej dokumentacji kodu i dodaniu README, został on opublikowany. Od tego momentu można znaleźć go pod adresem `https://codeberg.org/fkobi/sd2`.

Od tego momentu projekt jest publicznie dostępny jako wolne oprogramowanie i rozwijany w modelu otwartoźródłowym. Każdy internauta może wchodzić w różnorakie interakcje z projektem: od zostawienia gwiazdki czy obserwowania repozytorium, przez zgłaszanie błędów aż po sugerowanie konkretnych zmian z pomocą funkcjonalności *Pull requests*.

Posumowanie

Znalazłem pewien aspekt życia w którym nieoptymalność MusicScope była dla mnie problemem. Dokonałem analizy tego programu i wyróżniłem najważniejsze cechy tego typu oprogramowania celem znalezienia konkretnych aspektów, które mogę obiektywnie ulepszyć.

Następnie zgodnie z iteracyjnym modelem kaskadowym stworzyłem prototypy w różnych językach programowania do momentu wybrania Ziga, czyli nowej dla mnie technologii.

Korzystając z metodologii CI/CD zreimplementowałem od zera istniejące funkcjonalności MusicScope ulepszając je i czyniąc je bardziej zgodnymi ze standardami, zaleceniami i dobrymi praktykami.

Swoją pracę upubliczniłem pod Europejską Licencją Publiczną 1.2 na platformie Codeberg zgodnie z duchem Wolnego i Otwartźródłowego Oprogramowania.

Moim celem było stworzenie lepszego programu. Twierdzę, że mi się to udało ponieważ `sd2` w porównaniu do MusicScope:

1. **Można pobrać** prosto od dewelopera z renomowanej platformy,
2. Działa na **znacznie** większej ilości platform,
3. Jest on używalny dla użytkowników z niepełnosprawnościami,
4. Jest **wolnym oprogramowaniem** typu *copyleft*, co zapobiega zabsorbowaniu przez korporacje[13],
5. Również analizuje pliki **zgodnie z międzynarodowymi standardami**,
6. Ma standardowy, deklaratywny, efektywny i intuicyjny interfejs,
7. W realnych warunkach działa **ponad dwadzieścia** razy szybciej,
8. Produkuje o wiele bardziej ergonomiczne raporty.

[13] Przykładem tego zjawiska niech będzie freenginx, czyli fork projektu nginx rozwijanego permisywnym BSD-2 który został kupiony.

Podziękowania

Zacznę od podziękowań dla mojego promotora, czyli Bartłomieja Stasiaka – dziękuję mu za zadeklarowanie poparcia dla tematu tej pracy jeszcze przed moim czwartym semestrem Informatyki Stosowanej.

W drugiej kolejności podziękuję Europejskiej Fundacji Wolnego Oprogramowania za przyjęcie mnie na staż na stanowisko administratora systemów podczas mojego ostatniego semestru pierwszego stopnia studiów. W tym kontekście dziękuję również administracji Wydziału Fizyki Technicznej, Matematyki i Informatyki Stosowanej za umożliwienie mi *de facto* mieszkania i pracowania w Berlinie podczas studiowania w Łodzi.

Chcę również podziękować każdemu kto zapewnił mi warunki do pracy nad tym projektem, służył radą i pomocą w testowaniu a nawet po prostu słuchał moich pomysłów i prezentacji. Tym samym dziękuję rodzicom, kolegom z kierunku i pracy oraz teraźniejszym i przeszłym przyjaciołom. Z szczególnością dziękuję *Università degli Studi di Cagliari*, ponieważ to podczas mojego Sardyńskiego Erasmusa podjąłem najwięcej decyzji projektowych i to właśnie w Cagliari napisałem najwięcej kodu `sd2`. Pomimo mojej później rejestracji ta instytucja zapewniła mi fizyczną i mentalną przestrzeń w której moja kreatywność prosperowała.

Jako, że jest to moja praca dyplomowa na kierunku Informatyka Stosowana chciałbym również podziękować wszystkim którzy zainspirowali mnie, umożliwili i byli wsparciem w mojej zmianie kierunku z Fizyki Technicznej, a w szczególności: Michałowi Wasiakowi, Michałowi Dobrskiemu, Idzie Haider, Mariowi Linsowi, Tobiasowi Höllerowi, Gabrielowi Molnarowi, Agnieszce Wosiak, Jakubowi Samkowi, Łukaszowi Moskwie, Romanowi Krasiukianisowi, Michałowi Karbowańczykowi i Mateuszowi Smolińskiemu.

Na koniec podziękuję twórcom języka programowania Zig i wykorzystanych przeze mnie bibliotek – jeśli miałbym stworzyć chociaż jedną z nich przed terminem skończenia studiów `sd2` było by z pewnością gorszym oprogramowaniem.

Poznajcie inżyniera

W ostatnich godzinach pisania tego dokumentu myślałem coraz więcej o cytacie z filmu *Meet the Engineer* który jest w zasadzie monologiem tytułowej postaci. Zaczyna się on następująco:

> Jestem inżynierem więc rozwiązuję problemy. Nie takie w stylu *Czym jest piękno?* ponieważ one przypadają do obszaru twoich dylematów filozoficznych. Ja rozwiązuję **praktyczne problemy.**

Bibliografia

[1] *Alacritty, a cross-platform, OpenGL terminal emulator*. 3 paź. 2018. URL: https://alacritty.org.

[2] *Algorithms to measure audio programme loudness and true-peak audio level*. Rekomendacja BS.1770. International Telecommunication Union, 2007.

[3] Astral. *Ruff, an extremely fast Python linter and code formatter*. 29 sierp. 2022. URL: https://docs.astral.sh/ruff/.

[4] Austin Common Standards Revision Group. *Portable Operating System Interface*. Standard IEEE 1003.1. Institute of Electrical i Electronics Engineers, 2024.

[5] *BiglyBT, a feature filled, open source, ad-free, bittorrent client*. 16 maj. 2006. URL: https://www.biglybt.com.

[6] Erik de Castro Lopo. *libsndfile*. 15 lut. 1999. URL: https://libsndfile.github.io/libsndfile/.

[7] Prajwal Chapagain. *yazap, The ultimate Zig library for seamless command line argument parsing*. 28 sierp. 2022. URL: https://github.com/PrajwalCH/yazap.

[8] *Compact disc digital audio system*. Standard IEC 60908:1987. Genewa, Szwajcaria: International Electrotechnical Commission, 1987.

[9] Meghan Denny. *zig-time, a date and time parsing and formatting library*. 17 paź. 2021. URL: https://github.com/nektro/zig-time.

[10] *Floating-Point Arithmetic*. Standard IEEE 754. Institute of Electrical i Electronics Engineers, 1985.

[11] *Free Lossless Audio Codec (FLAC)*. Standard RFC 9639. Internet Engineering Task Force, 2024.

[12] Fundacja Gentoo. *Gentoo Linux*. 31 mar. 2002. URL: https://www.gentoo.org.

[13] Roy Ivy III i Terts Diepraam. *uutils, a cross-platform Rust rewrite of the GNU coreutils*. 20 kw. 2020. URL: https://github.com/trifectatechfoundation/sudo-rs.

[14] *Information technology — Programming languages — C*. Standard ISO/IEC 9899:2024. Genewa, Szwajcaria: International Organization for Standardization, 2024.

[15] Andrew Kelly. „A Systems-Minded Approach to Creating a Music Player Application”. Seminarium na konferencji. Systems Distributed ’24. 2024. URL: https://www.youtube.com/watch?v=SCLrNqc9jdE&t=330 (term. wiz. 03. 12. 2025).

[16] Andrew Kelly. „ZIG, a Programming Language for Maintaining Robust, Reusable software”. Seminarium na konferencji. Emerging Technologies for The Enterprise Conference. 2019. URL: https://www.youtube.com/watch?v=Gv2I7qTux7g (term. wiz. 03. 01. 2026).

[17] Jan Kokemüller. *libebur128*. 13 list. 2013. URL: https://github.com/jiixyj/libebur128.

[18] *Loudness normalisation and permitted maximum level of audio signals*. Rekomendacja R 128. European Broadcasting Union, 2014.

[19] Lulu "Morganamilo". *Ruff, an extremely fast Python linter and code formatter*. 28 paź. 2020. URL: https://github.com/Morganamilo/paru.

[20] John MacFarlane, Martin Woodward i Jeff Atwood. *CommonMark Spec*. Standard 0.31.2. 2024. URL: https://commonmark.org.

[21] Mojang Studios. *Minectaft na AppleTV*. 19 grud. 2016. URL: https://minecraft.fandom.com/wiki/Apple_TV_Edition.

[22] *Music Player Daemon*. Wer. 0.24.5. 31 lip. 2025. URL: https://www.musicpd.org.

[23] NME. *Bloc Party remove ‘A Weekend In The City: B-Sides’ from streaming*. Data publikacji: 2024-11-22. 2024. URL: https://www.nme.com/news/music/bloc-party-remove-a-weekend-in-the-city-b-sides-from-streaming-the-audio-quality-was-well-below-what-we-expect-3815170 (term. wiz. 04. 12. 2025).

[24] OpenMP Architecture Review Board. *OpenMP*. 1 paź. 1997. URL: http://openmp.org.

[25] David Peter. *bat, a cat(1) clone with wings*. 22 kw. 2018. URL: https://github.com/sharkdp/bat.

[26] *PipeWire*. Wer. 1.4.9. 9 paź. 2025. URL: https://pipewire.org.

[27] G.A.A. Prana, C. Treude i F. Thung. „Categorizing the Content of GitHub README Files”. W: *Empirical Software Engineering* 24 (2019), s. 1296–1327.

[28] Projekt GNOME. *Assistive Technology Service Provider Interface*. 2001. URL: https://gitlab.gnome.org/GNOME/at-spi2-core.

[29] Projekt GNOME. *Orca*. 3 wrz. 2006. URL: https://gitlab.gnome.org/GNOME/orca.

[30] *qBittorrent BitTorrent client*. 16 maj. 2006. URL: https://ziglang.org/download/0.15.1/release-notes.html.

[31] *REUSE Specification*. Standard 3.3. Berlin, Niemcy: Free Software Foundation Europe, 14 list. 2024.

[32] Dennis M. Ritchie. *The Development of the C Language*. Nowy Jork, USA: ACM SIGPLAN Notices, 1993, s. 201–208.

[33] Sky News. *Kanye West's new album Donda 2 will be only be available exclusively on his own platform the Stem Player*. Data publikacji: 2022-02-18. 2022. URL: https://news.sky.com/story/kanye-wests-new-album-donda-2-will-be-only-be-available-exclusively-on-his-own-platform-the-stem-player-12545167 (term. wiz. 03. 12. 2025).

[34] Richard Stallman. *GNU Emacs*. 20 mar. 1985. URL: https://www.gnu.org/software/emacs/.

[35] Trifecta Tech Foundation. *sudo-rs, a memory safe implementation of sudo and su*. 29 sierp. 2023. URL: https://github.com/trifectatechfoundation/sudo-rs.

[36] *Valgrind,an instrumentation framework for building dynamic analysis tools*. 27 lip. 2002. URL: https://www.valgrind.org.

[37] Xi-Vero. *MusicScope*. WebArchive. 2015. URL: https://www.xivero.com/musicscope/ (term. wiz. 20. 05. 2018).

[38] *Wine*. Wer. 10.0. 21 sty. 2025. URL: https://www.winehq.org.

[39] Zig Software Foundation. *Zig 0.15.1 Release Notes*. Wer. 0.15. 2025. URL: https://ziglang.org/download/0.15.1/release-notes.html.

Spis rysunków

1 Interfejs MusicScope świeżo po uruchomieniu 17
2 Okno *About* MusicScope 19
3 Raport graficzny MusicScope utworu *Soma* 19
4 Diagram stanów MusicScope z perspektywy użytkownika 20
5 Podsumowanie czasów 41

Spis tabel

1 Informacje o albumach wykorzystanych do testowania efektywności analizy 12
2 Zestawienie szybkości analizy MusicScope wybranych albumów 22
3 Zestawienie szybkości analizy SINMS wybranych albumów 28
4 Zestawienie jednowątkowej szybkości analizy `sd2` wybranych albumów . 41
5 Zestawienie wielowątkowej szybkości analizy `sd2` wybranych albumów . 41

Spis kodów

1 Mockup wykorzystania imperatywnego interfeju linii komend sinms . . . 26
2 Przykładowe wykorzystanie SINMS 27
3 Fragment `track_info.zig` ukazujący użycie biblioteki C `ebur128` 32
4 Deklarowane wersji w Zigu 33
5 Deklarowanie wersji w C 33
6 Generyczna funkcja `add_frames` z `ebur128.c` 34
7 Generyczna deklaracja funkcji `add_frames` w Zigu 34
8 Pseudokod C: kalkulacja i wyświetlenie RMS 35
9 Pseudokod Ziga: kalkulacja i wyświetlenie RMS 35
10 Struktura `TrackInfo` w wersjach <0.2.0 38