Міністерство освіти і науки України

Національний університет «Києво-Могилянська академія»

Факультет інформатики

Кафедра математики

# Магістерська робота

освітній ступінь – магістр

на тему: «**ЗАСТОСУВАННЯ ПРИХОВАНИХ МАРКОВСЬКИХ МОДЕЛЕЙ ДО РОЗВ'ЯЗАННЯ ЗАДАЧІ РОЗПІЗНАВАННЯ АКОРДІВ**»

Виконав: студент 2-го року навчання
освітньо-наукової програми
«Системний аналіз»,
спеціальності 124 Системний аналіз

Андрущак Григорій Сергійович

Керівник: Чорней Р. К.,
кандидат фіз.-мат. наук, доцент

Рецензент_____
(прізвище та ініціали)

Магістерська робота захищена
з оцінкою _____

Секретар ЕК _____
«____» _____ 20____ р.

Київ – 2021

# Зміст

# Вступ

**Актуальність теми.** Приховані марковські моделі використовуються у алгоритмах розпізнавання мовлення, зокрема цьому присвячені роботи Марка Гейлза та Стіва Янга [6].

Дана робота зосереджена на застосуванні прихованих марковських моделей для розпізнавання музичних акордів. Темі розпізнавання музики також присвятили свої роботи Ятинг Чі, Джон Вільям Пейслі та Лоуренс Карін [7].

Інформація про останні розробки в цій галузі описується в статті "Music Information Retrieval: Recent Developments and Applications" [5].

**Об’єкт дослідження.** Музичні аудіофайли.

**Предмет дослідження.** Музичні акорди в аудіофайлах.

**Мета дослідження.** Розпізнавання музичних акордів в музичних аудіофайлах.

**Методи дослідження**. Швидке перетворення Фур'є, приховані марковські моделі.

**Практичне значення роботи.** Розроблений програмний застосунок дозволяє визначити послідовність акордів в музичних аудіофайлах.

Робота складається зі вступу, шести розділів, висновку, списку літератури та додатків. У першому розділі описується загальна інформація про дослідження та обробку музики. У другому розділі роботи досліджується теорія музики та її представлення. Третій розділ присвячено розпізнаванню нот та акордів. У четвертому розділі описуються методи пошуку початку та кінця акорду в аудіофайлі. У п’ятому розділі описується метод створення спостережуваних станів системи. У шостому розділі описується створення прихованих станів моделі та їх застосування. Загальний обсяг роботи становить 24 сторінки. Робота містить 11 рисунків та 9 додатків. Список використаної літератури налічує 7 найменувань.

# РОЗДІЛ 1. Загальна інформація про дослідження та обробку музики

Музика є всюдисущою і життєво важливою частиною життя мільярдів людей у всьому світі. Музичні твори та виступи є одними з найскладніших наших культурних артефактів, і емоційна сила музики може торкнутися нас дивовижними та глибокими способами. Музика охоплює величезний спектр форм і стилів: від простих, не супроводжуваних народних пісень, до популярної та джазової музики, до симфоній для повних оркестрів. Цифрова революція в розповсюдженні та зберіганні музики одночасно викликала величезний інтерес та увагу до способів застосування інформаційних технологій до такого типу контенту. Від перегляду особистих колекцій до відкриття нових виконавців, до управління та захисту прав творців музики, комп’ютери зараз глибоко залучені майже до кожного аспекту споживання музики, навіть не згадуючи їх життєво важливу роль у більшості сучасного музичного виробництва.

Незважаючи на важливість музики, обробка музики все ще є відносно молодою дисципліною порівняно з обробкою мови, дослідницькою сферою з давньою традицією. Власне, більш широке науково-дослідне співтовариство, представлене Міжнародним Товариством Пошуку Музичної Інформації (ISMIR), яке систематично займається широким спектром комп'ютерних тем музичного аналізу, обробки та пошуку, було сформовано у 2000 році. Традиційно, дослідження, засноване на музиці, здебільшого проводиться на основі символічних уявлень з використанням музичних позначень, або MIDI-репрезентацій. Через збільшення доступності оцифрованих аудіоматеріалів та вибуху обчислювальної потужності, автоматизована обробка звукових сигналів тепер все більше у фокусі дослідників.

Багато з цих досліджень спрямовані на розвиток технологій, які дозволяють користувачам отримувати доступ та досліджувати музику у

різних її аспектах. Наприклад, методи аудіо дактилоскопії сьогодні інтегруються в комерційні продукти, що допомагають користувачам організовувати свої приватні музичні колекції. Методи обробки музики використовуються в аудиоплеерах, які підкреслюють поточні ступені в межах нотного аркуша під час відтворення запису симфонії. На вимогу слухача автоматично подається додаткова інформація про мелодичну та гармонічну прогресії або ритм та темп. Інтерактивні музичні інтерфейси відображають структурні частини поточного музичного твору і дозволяють користувачам безпосередньо переходити до будь-якої ключової частини, наприклад, хорова секція, основна музична тема або сольна секція без нудного швидкого перемотування. Крім того, слухачі оснащені пошуковими системами, схожими на Google, які дозволяють їм досліджувати великі музичні колекції різними способами. Наприклад, користувач може створити запит, вказавши певну групу нот, або якийсь гармонійний, або ритмічний рисунок, насвистуючи мелодію, або настукуючи ритм, або просто обравши короткий уривок із запису. Система надає користувачеві список класифікованих музичних уривків із колекції, які музично пов'язані із запитом. В обробці музики однією з головних цілей є внести концепції, моделі, алгоритми, реалізації та оцінки для вирішення подібних проблем аналізу та пошуку [1].

За останні п’ятнадцять років обробка музики та пошук музичної інформації (MIP) перетворилися на яскраву та багатопрофільну область досліджень. Через різноманітність та багатство музики ця область об’єднує дослідників та студентів з багатьох областей, включаючи інформатику, аудіо інженерію, інформатику та музикознавство.

# РОЗДІЛ 2. Теорія музики та її представлення

Музику можна представити різними способами та формами. Наприклад, композитор може записати композицію у вигляді музичної партитури. У партитурі музичні символи використовуються для візуального кодування нот і того, як музиканти повинні грати на цих нотах. Друкована форма музичної партитури (рис. 1) також називається нотною формою. Оригінальним носієм цього представлення є папір, хоча вона також доступна на екранах комп'ютерів за допомогою цифрових зображень.

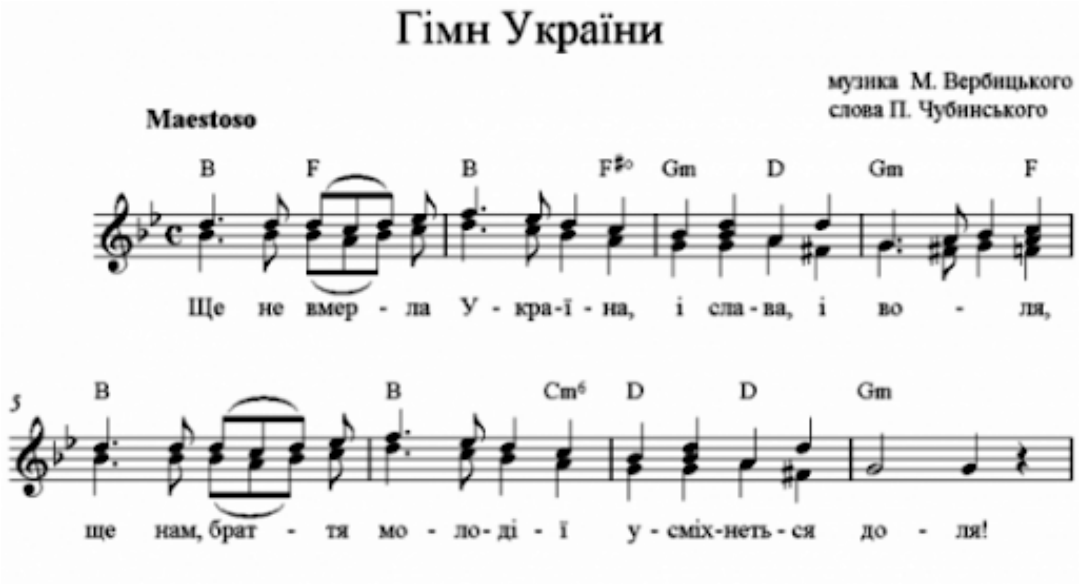



Рис. 1

Для електронних інструментів та комп'ютерів музика може передаватися за допомогою стандартних протоколів, таких як широко використовуваний протокол цифрового інтерфейсу музичного інструменту (MIDI) (рис. 2), де повідомлення про події задають тони, швидкості та інші параметри для генерування передбачених звуків. Термін **символічне представлення музики** використовується для позначення будь-якого машиночитаного формату даних, який явно представляє музичні сутності. Вони можуть варіюватися від нотних подій у певний час, як це стосується

файлів MIDI, до графічних фігур із доданим музичним значенням, як це стосується систем відображення музики. На відміну від символічних представлень, аудіопрезентації (рис. 3), такі як файли WAV або MP3, не вказують явно музичні події. Ці файли кодують акустичні хвилі, які генеруються, коли джерело (наприклад, інструмент) створює звук, який рухається до людського вуха як коливання тиску повітря.

Рис. 2

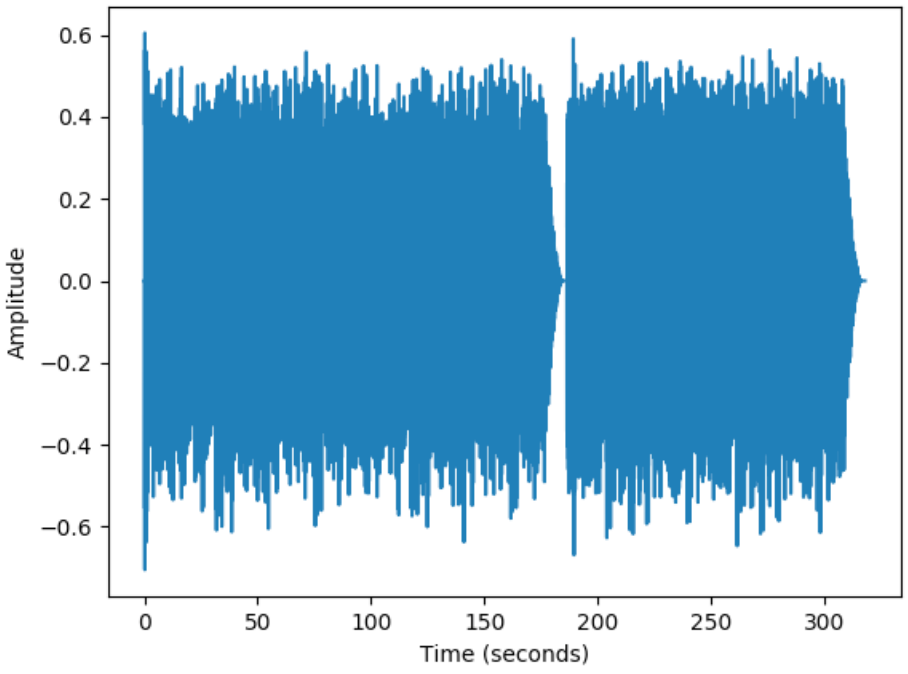



Рис. 3

Дві ноти з частотами у співвідношенні, що дорівнює будь-якій степені двійки, сприймаються як дуже схожі. Через це всі ноти з таким відношенням можуть бути згруповані в один клас. Це спостереження також призводить до фундаментального поняття октави, яке визначається як інтервал між однією музичною нотою та іншою з половиною або подвоєнням її основної частоти. Використовуючи це визначення, клас тонів - це множина усіх тонів або нот, які знаходяться на відстані цілого числа октав одна від одної [1].

Для того щоб описати музику за допомогою кінцевої кількості символів, потрібно дискретизувати простір усіх можливих тонів. Це призводить до поняття музичного строю, яке можна розглядати як скінченний набір репрезентативних тонів. Через близький октавний зв'язок тонів зазвичай вважається, що стрій охоплює одну октаву, а вищі або нижчі октави просто повторюють схему. Також музичний стрій можна задати діленням октавного простору на певну кількість ступенів строю. Елементи шкали часто просто називають нотами сторю і упорядковуються відповідно до відповідних тонів.

На сьогоднішній день найрозповсюдженішим строєм є 12-ступеневий рівномірно темперований стрій (рис. 4 та рис. 5). Він є основним в європейській музиці з XIX століття. У цьому строї октава ділиться на певну кількість однакових ступенів, а саме 12. Вони віддалені один від одного на відстань хроматичного півтону $\left(1:\sqrt[12]{2}\right)$. Нота A4 (*Ля* четвертої октави) відповідає фундаментальній частоті 440Гц, C0 (*До* нульової октави) – 16Гц.

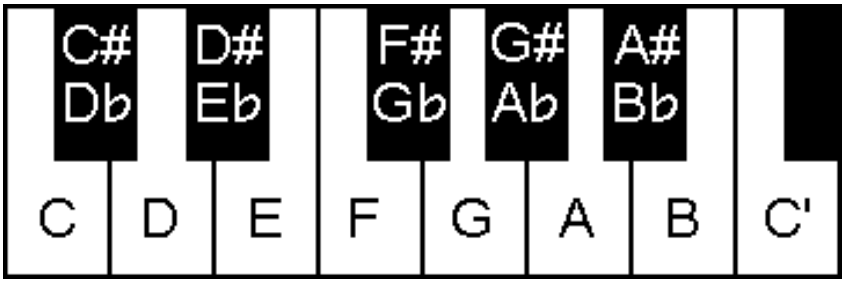



Рис. 4

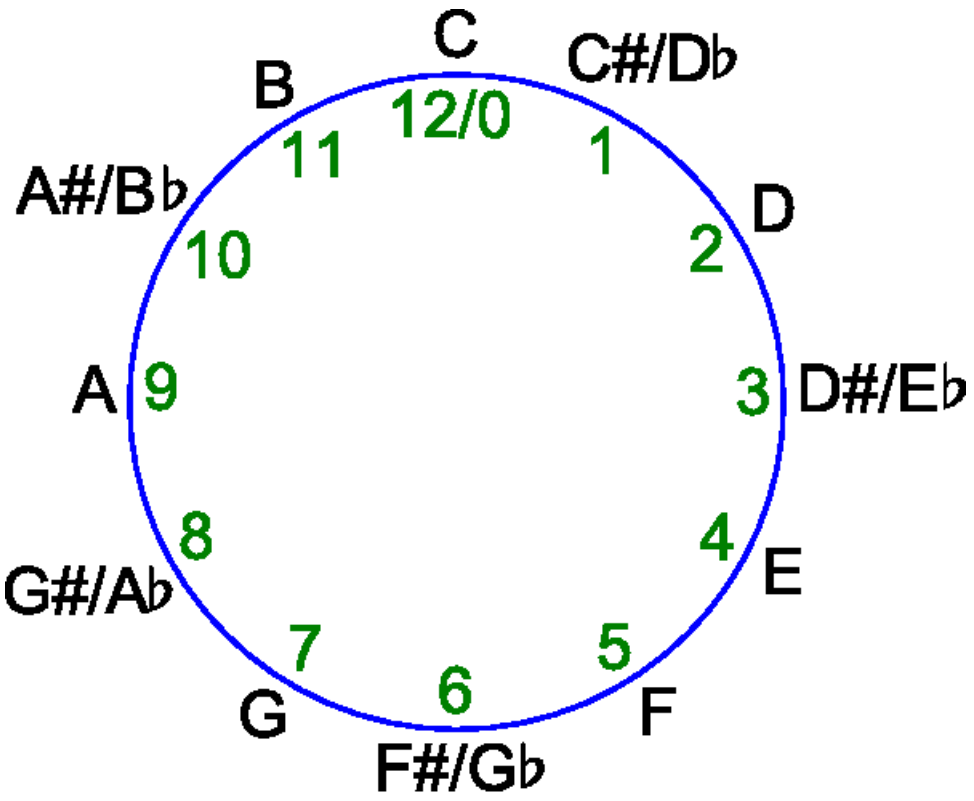
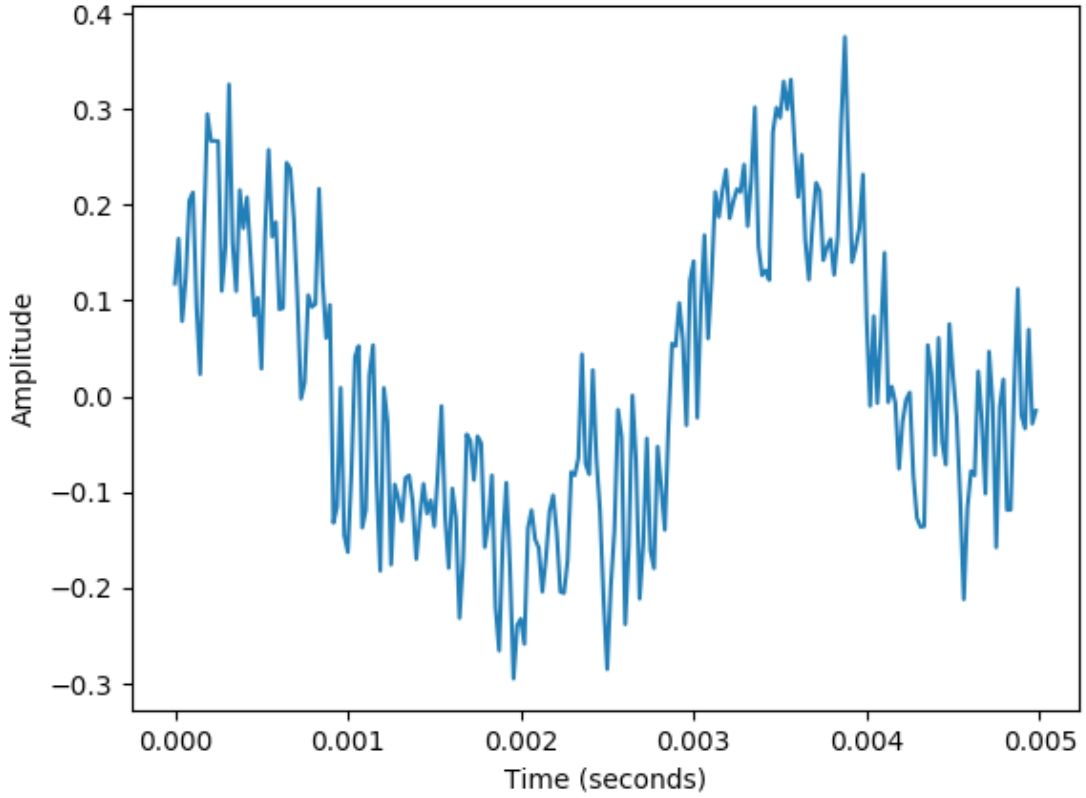



Рис. 5

Стандарт символічного представлення музики MIDI був розроблений у 1981 – 1983 рр. та досі активно використовується. Цей формат дає можливість електронним музичним інструментам взаємодіяти між собою чи комп'ютером та іншим MIDI-сумісним обладнанням, здійснювати з одного інструменту управління іншими. MIDI не передає та не генерує звук — натомість MIDI передає «повідомлення», такі як нота-вкл./нота-викл., висота (тон) та динаміка взятої ноти на інструменті; контрольні сигнали (CC) для таких параметрів як гучність, панорама, сигнали відліку часу для синхронізації темпу, тощо. Музичний інструмент приймає такі повідомлення і генерує звук. Інструментом може бути як реальний пристрій, наприклад, синтезатор, так і віртуальний - програма на комп'ютері.

Натомість, WAV (waveform audio format) зберігає значення амплітуд коливання з певною дискретизацією, наприклад 44100 значень на секунду, та є поширеним форматом для зберігання та відтворення аудіо файлів без стиснення та втрати якості (lossless). Типовий зміст WAV файлу:


0.4
0.3
0.2
0.1
0.0
−0.1
−0.2
−0.3
Amplitude
0.000
0.001
0.002
0.003
0.004
0.005
Time (seconds)

# РОЗДІЛ 3. Розпізнавання нот та акордів

## 3.1 Розпізнавання нот

У алгоритмах цифрової обробки сигналів широко застосовується перетворення Фур'є, а саме його дискретна версія. Перетворення Фур'є — інтегральне перетворення однієї комплекснозначної функції дійсної змінної на іншу. Перетворення Фур'є функції *f(t)* математично визначається як комплексна функція *F(ω)*, яка задається інтегралом

$$F(\omega) = \int_{-\infty}^{\infty} f(t) e^{-i\omega t} dt$$

Обернене перетворення Фур'є задається виразом

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

Дискретне перетворення Фур'є вимагає в якості входу дискретну функцію. Такі функції часто створюються шляхом дискретизації (вибірки значень з безперервних функцій). Дискретні перетворення Фур'є допомагають вирішувати диференціальні рівняння в частинних похідних і виконувати такі операції, як згортки.

Пряме перетворення:

$$X_k = \sum_{n=0}^{N-1} x_n e^{-\frac{2\pi i}{N} kn} = \sum_{n=0}^{N-1} x_n \left[ cos\left(\frac{2\pi kn}{N}\right) - i sin\left(\frac{2\pi kn}{N}\right) \right], (k = 0, \dots, N-1).$$

Обернене перетворення:

$$x_n = \frac{1}{N} \sum_{n=0}^{N-1} X_k e^{\frac{2\pi i}{N} kn} = \sum_{k=0}^{N-1} X_k \left[ cos\left(\frac{2\pi kn}{N}\right) + i sin\left(\frac{2\pi kn}{N}\right) \right], (n = 0, \dots, N-1).$$

Позначення:

- $N$ - кількість значень сигналу, виміряних за період, а також кількість компонент розкладання;
- $x_n$, $n = 0, \dots, N-1$ - виміряні значення сигналу, які є вхідними даними для прямого перетворення і вихідними для оберненого;

- $X_k$, $k = 0, \dots, N-1$ - $N$ комплексних амплітуд синусоїдальних сигналів, що складають вихідний сигнал; є вихідними даними для прямого перетворення і вхідними для оберненого; оскільки амплітуди комплексні, то по ним можна обчислити одночасно і амплітуду, і фазу;

Знаючи частоту коливання тиску повітря, ми можемо точно сказати яка це нота. Приклад таких пар значень можна побачити у нашій програмі (Додаток Д) [4].

### 3.2 Розпізнавання акордів

У музиці **гармонія** означає одночасне звучання різних нот, які утворюють згуртовану сутність у свідомості слухача. Основними складовими гармонії, принаймні у західній музичній традиції, є **акорди**, які є музичними конструкціями, які, як правило, складаються з трьох і більше нот. Наприклад, акорд C (*До* мажор) складається з нот C (*До*), E (*Мі*) та G (*Соль*). Існують різні методи розпізнавання акордів, наприклад, на базі шаблонів, або на базі HMM (Прихована марковська модель).

#### 3.2.1 Розпізнавання акордів на базі шаблонів

Типова система розпізнавання акордів складається з двох основних етапів. На першому етапі даний аудіозапис розрізається на кадри, і кожен кадр трансформується у нотний вектор. На другому кроці використовуються методи співставлення шаблонів, щоб порівняти кожен вектор з набором заздалегідь заданих моделей акордів. Найкраща міра подібності визначає акорд, що є у цьому кадрі [1].

#### 3.2.2 Розпізнавання акордів на базі прихованої марковської моделі

Основна ідея - запровадити перехідну модель, яка виражає ймовірність переходу від одного аккорда до іншого. Це призводить нас до концепції,

відомої як прихована марковська модель (HMM). Поняття HMM широко застосовується в таких додатках, як розпізнавання мови, а також є фактично стандартним методом у більшості автоматизованих процедур розпізнавання акордів.

На основі ланцюга Маркова ми можемо обчислити ймовірність даного спостереження, що складається з послідовності станів або типів акордів. Однак у нашому сценарії розпізнавання акордів це не те, що нам потрібно. Замість того, щоб спостерігати послідовність типів акордів, ми спостерігаємо послідовність нотних векторів, які так чи інакше пов'язані з типами акордів. Іншими словами, послідовність станів не видно безпосередньо, а лише послідовність нечіткого спостереження, яка генерується на основі послідовностей станів. Крім того, замість обчислення ймовірності послідовності спостереження, мета - виявити взаємозв'язок між спостережуваними ознаками векторів та базовими типами акордів.

Далі ми розширимо поняття ланцюгів Маркова до статистичної моделі (HMM). Ідея полягає в тому, щоб представити зв'язок між спостережуваними характеристиками векторів та типами акордів (станами), використовуючи імовірнісну структуру. Кожен стан оснащений функцією ймовірності, яка виражає ймовірність того, що певний тип акордів виводить певний вектор. В результаті ми отримуємо двошаровий процес, що складається з прихованого шару і шару, що спостерігається. Прихований шар створює послідовність станів, яку не можна спостерігати («прихована»), але генерує послідовність спостережень на основі функцій ймовірностей, залежних від стану [1].

Задачу розпізнавання акордів на базі прихованої марковської моделі можна розділити на 3 складові:

1) розділення аудіофайлу на сегменти, кожен з яких містить один акорд;
2) визначення спостережуваних станів;
3) знаходження ймовірностей переходів між акордами (приховані стани).

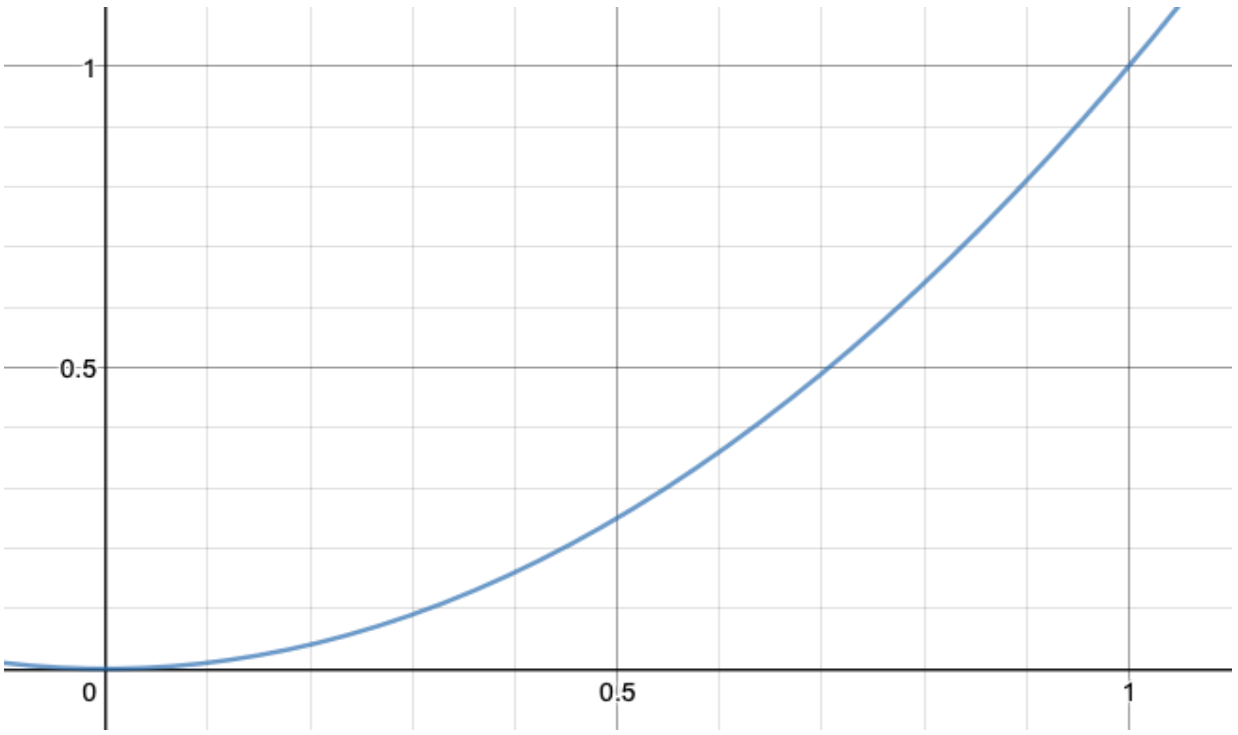
# РОЗДІЛ 4. Пошук початку та кінця акорду у музичному файлі

Пошук моменту початку і кінця акорду є нетривіальною задачею. Пропонуємо два методи її розв'язання. Кожен із них є дуже чутливим до вхідних параметрів. Реалізацію цих методів можна знайти в Додатку Б, функції *extract* та *extract_v2* відповідно.

## 4.1 Метод квадратів амплітуд

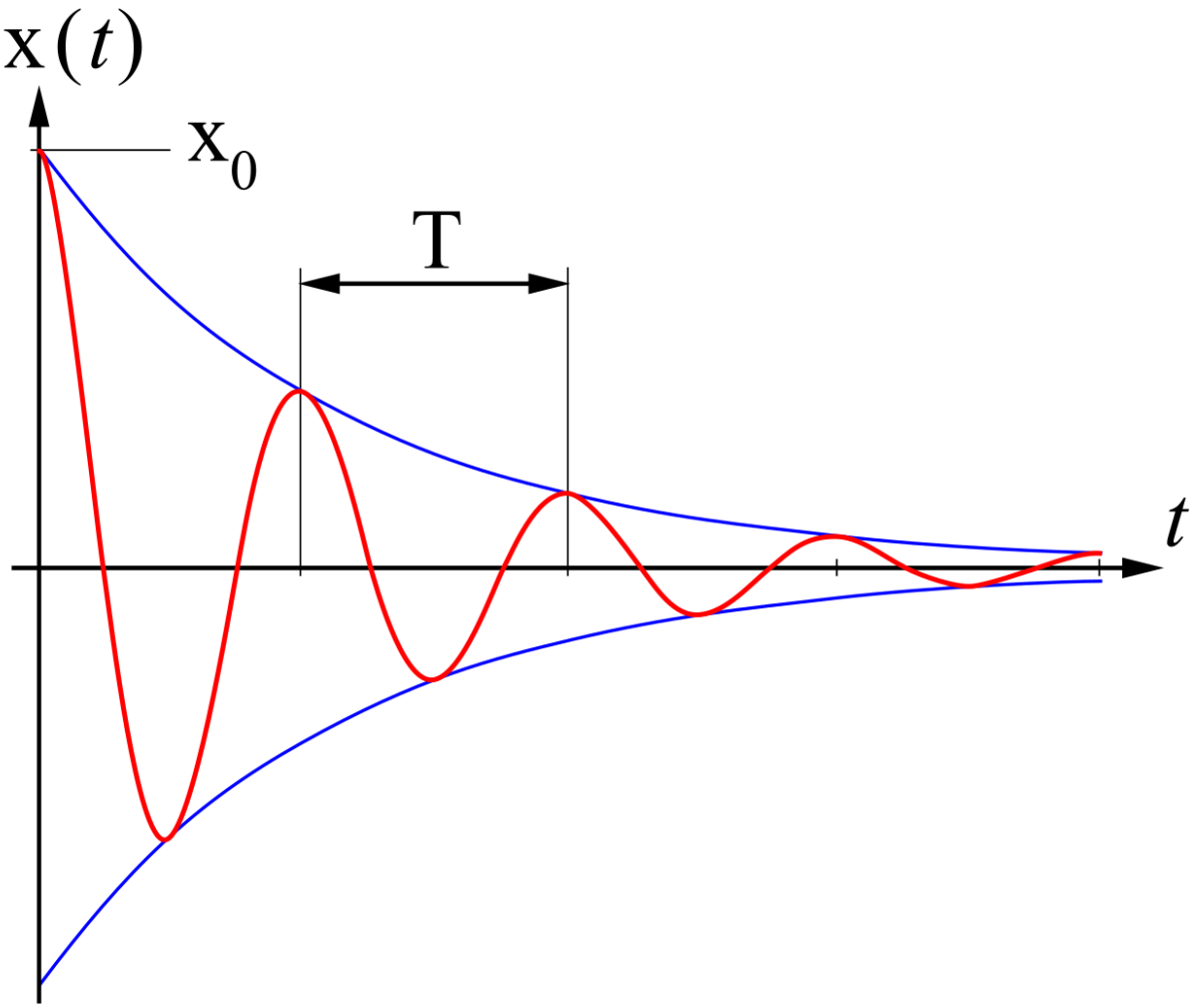
Значення амплітуд в аудіофайлах, переважно, знаходиться на проміжку [-1; 1], де 1 - це 100% гучності, отже у разі, коли існує значення амплітуди, модуль якої більше за 1, усі значення варто нормалізувати відносно модуля найбільшої.

Метод квадратів амплітуд базується на тому, що значення функції гіперболи тим швидше спадає, чим ближче до 0 значення аргумента.




$$f(x) = x^2$$

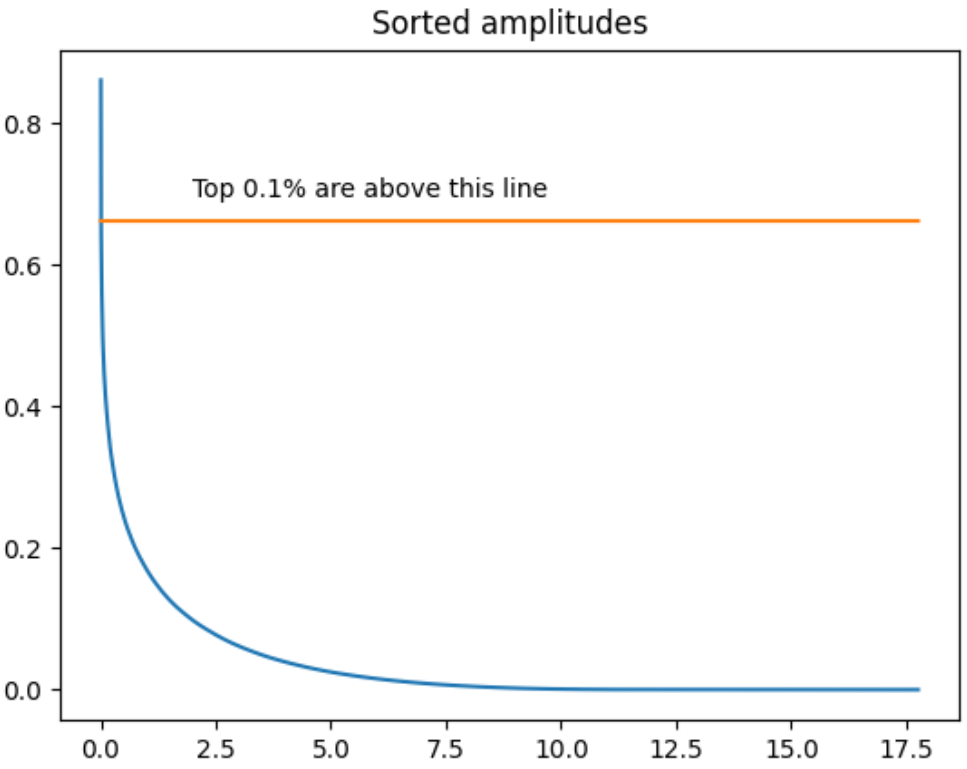
Головна ідея цього методу полягає у тому, що з часом, через розсіювання енергії, амплітуда коливань зменшується, тобто існує декремент затухання, наприклад, логарифмічний.



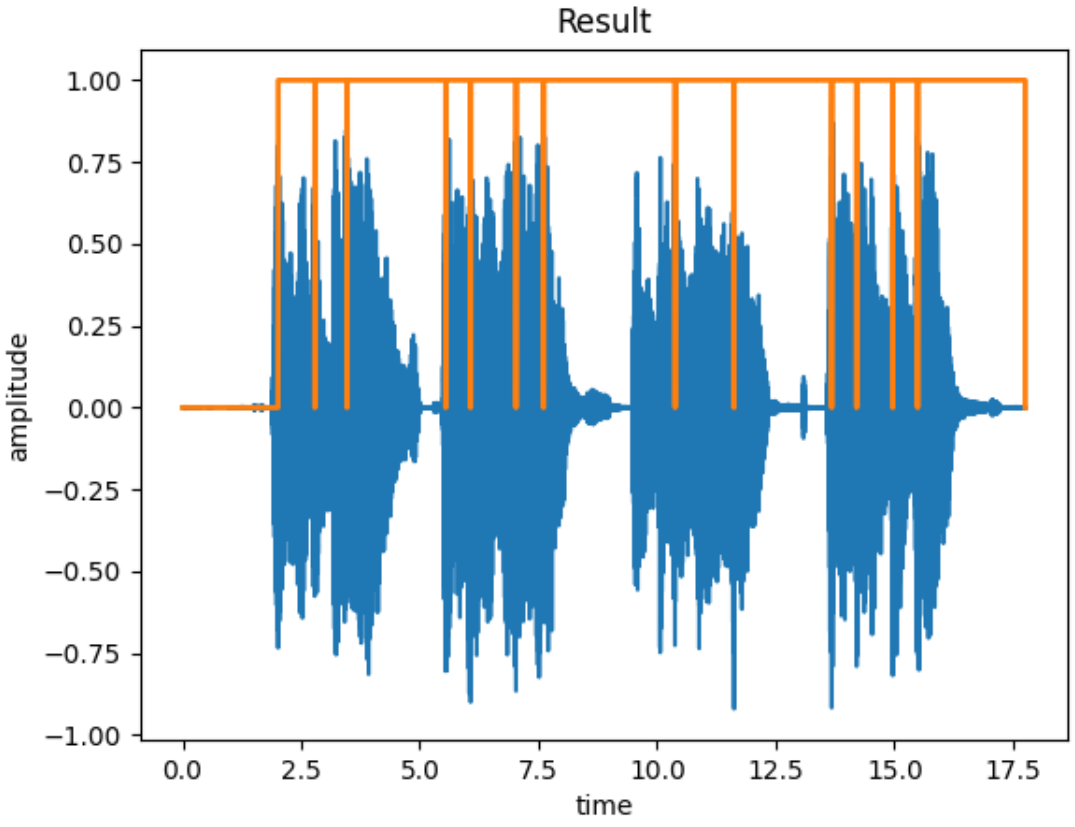

Через це можна вважати, що в момент початку відтворення акорду, його амплітуда максимальна, і з часом зменшується до моменту початку відтворення наступного акорду.

Метод, який ми реалізували на мові програмування Python 3, залежить від двох вхідних параметрів:

- **`top_perc`** - частка найбільших амплітуд, серед яких ми шукатимемо початок акорду;
- **`min_dur`** - мінімальна тривалість акорду.




Серед точок, вищих за оранжеву лінію, ми обираємо ті, різниця в часі між якими, наприклад, 0.5 секунди.




Приклад результату роботи алгоритму (**`top_perc`** = 0.001, **`min_dur`** = 0.5)

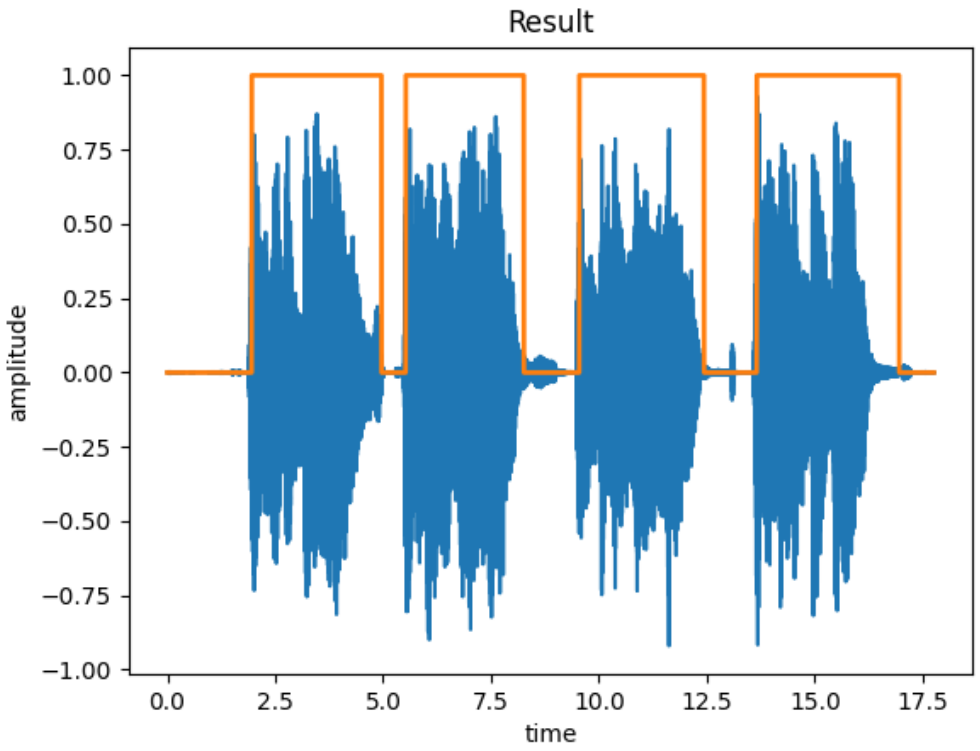
### 4.2 Метод ітерацій локальних екстремумів

Ідея цього методу схожа з попереднім, але відрізняється спосіб знаходження шуканих амплітуд. В даному випадку ми шукатимемо локальні екстремуми функції амплітуди від часу. Але, оскільки коливання є гармонійною функцією, похідна цієї функції також буде гармонійною, що зводить задачу до попередньої.

Альтернативою може бути ручний пошук значень амплітуд, які більші за попереднє і наступне значення, у декілька ітерацій.

Метод, який ми реалізували на мові програмування Python 3, залежить від трьох вхідних параметрів:

- **`min_x`** - кратність зменшення кількості точок до припинення ітерацій (на кожній ітерації алгоритму кількість точок зменшується);
- **`f_amp`** - частка від максимального значення амплітуди, після якої ми розглядатимемо точку як можливий початок акорду;
- **`min_dur`** - мінімальна тривалість акорду.




Приклад результату роботи алгоритму (**`min_x`** = 1500, **`f_amp`** = 0.15, **`min_dur`** = 0.5)

# РОЗДІЛ 5. Спостережувані ознаки нотних векторів

## 5.1 Створення нотних векторів

Кожній ноті відповідає певна частота коливання тиску в атмосфері.

| Нота \ Октава | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| C | 16.35 | 32.7 | 65.41 | 130.81 | 261.63 | 523.25 | 1046.5 | 2093 |
| C# | 17.32 | 34.65 | 69.3 | 138.59 | 277.18 | 554.37 | 1108.73 | 2217.46 |
| D | 18.35 | 36.71 | 73.42 | 146.83 | 293.66 | 587.33 | 1174.66 | 2349.32 |
| D# | 19.45 | 38.89 | 77.78 | 155.56 | 311.13 | 622.25 | 1244.51 | 2489.02 |
| E | 20.6 | 41.2 | 82.41 | 164.81 | 329.63 | 659.26 | 1318.51 | 2637.02 |
| F | 21.83 | 43.65 | 87.31 | 174.61 | 349.23 | 698.46 | 1396.91 | 2793.83 |
| F# | 23.12 | 46.25 | 92.5 | 185 | 369.99 | 739.99 | 1479.98 | 2959.96 |
| G | 24.5 | 49 | 98 | 196 | 392 | 783.99 | 1567.98 | 3135.96 |
| G# | 25.96 | 51.91 | 103.83 | 207.65 | 415.3 | 830.61 | 1661.22 | 3322.44 |
| A | 27.5 | 55 | 110 | 220 | 440 | 880 | 1760 | 3520 |
| A# | 29.14 | 58.27 | 116.54 | 233.08 | 466.16 | 932.33 | 1864.66 | 3729.31 |
| B | 30.87 | 61.74 | 123.47 | 246.94 | 493.88 | 987.77 | 1975.53 | 3951.07 |

Використовуючи алгоритм швидкого перетворення Фур'є [2], ми створюємо нотний вектор, за допомогою функції **note_vector**, яку ми реалізували на мові програмування Python 3.

```
def note_vector(fft_res: np.ndarray, note_width: float = 0.01) -> np.ndarray:
    res_vector = np.zeros(12)
    for i, hzs in enumerate(NOTE_TO_FREQ.values()):
        res_vector[i] += sum(
            [
                sum([fft_res[int(i)]])
                for hz in hzs
                for i in range(int(hz * (1 - note_width)),
                               int(hz * (1 + note_width)))
            ]
        )
    return res_vector / max(res_vector)
```

На вхід ця функція приймає два параметри:

- **`fft_res`** - результат роботи алгоритму швидкого перетворення Фур'є;
- **`note_width`** - ширина діапазону частот, в якому ми визначаємо ноту.

### 5.2 Функція ймовірності стану

Функція ймовірності стану може повертати певну міру відповідності нотного вектору до акорду (стану прихованого ланцюга). Цією мірою може виступати, наприклад, косинус кута між нотним вектором і вектором, який відповідає акорду з музичної теорії. Ми створили матрицю, у якій кожному акорду відповідає 12-вимірний вектор-стовпчик, де **1**, якщо нота є частиною акорду, та **0**, якщо ні.

| | C | C# | D | D# | E | F | F# | G | G# | A | A# | B | C m | C# m | D m | D# m | E m | F m | F# m | G m | G# m | A m | A# m | B m |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| B | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 1 |
| A# | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 0 |
| A | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 |
| G# | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 0 |
| G | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 0 |
| F# | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 1 |
| F | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 0 |
| E | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 0 | 0 |
| D# | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 |
| D | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 1 |
| C# | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 0 |
| C | 1 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 0 |

Реалізацією такої функції ймовірності в нашому коді програми виступає функція **`notes_to_chords`**, яка повертає список об'єктів **`Chord`**, кожен з яких зберігає значення косинусу кута (**`cos_sim`**) між нотним вектором (**`n_v`**), і вектором відповідного акорду (**`chord`**) з матриці.

```
def notes_to_chords(n_v: np.ndarray) -> List[Chord]:
    result = []
    for chord in CHORDS:
        chord_matrix_vector = CHORD_TO_VECTOR[chord]
        cos_sim = np.dot(n_v, chord_matrix_vector) / (
            np.linalg.norm(n_v) * np.linalg.norm(chord_matrix_vector)
        )
        result.append(Chord(chord, n_v, cos_sim))
    return result
```

Косинус кута ми шукаємо як

$$cos(\theta) = \frac{A \bullet B}{\|A\|\|B\|}$$

де:

- $A \bullet B$ – скалярний добуток векторів,
- $\|A\|\|B\|$ – добуток норм векторів.

# РОЗДІЛ 6. Приховані стани моделі

Особливістю прихованих станів моделі є те, що вони моделюються як марковський процес, а отже конкретні значення будь-якого заданого часового параметру *t + 1* залежать від значення у момент *t*, але не залежать від його значень у момент часу *t - 1, t - 2* і т. д. Іншими словами «майбутнє» процесу залежить лише від «поточного» стану, але не залежить від «минулого».

Така особливість одночасно є і недоліком, і перевагою. В музичній теорії існує поняття **акордової послідовності**, або **акордової прогресії**. Акордова послідовність - це основа гармонії західної музичної традиції, починаючи від епохи загальної практики класичної музики до 21-го століття. Акордові послідовності є основою західних жанрів популярної (поп-музика, рок-музика) і народної музики (блюз, джаз). У цих жанрах мелодія і ритм будуються на акордах, а не навпаки. Більшість прогресій складаються з трьох, або чотирьох акордів. Певні послідовності більш вживані за інші, отже, маючи інформацію про два/три акорди з послідовності, апостеріорний ймовірнісний розподіл третього/четвертого відрізнятиметься від апріорного. Через це, використання марковського процесу є недоліком даної системи. З іншого боку, відсутність «пам’яті» в системі зменшує шкоду від можливих збоїв та помилок у алгоритмі розпізнавання акордів.

Визначення ймовірностей переходу з одного акорду на інший є нетривіальною задачею. Існує декілька підходів до її розв’язання:

1) опитування великої кількості музичних експертів із подальшим статистичним дослідженням їх відповідей;
2) методи машинного навчання із використанням великих об’ємів навчальних даних.

Кожен з вищезазначених підходів вимагає великої кількості ресурсів та часу, тому ми пропонуємо швидкий, проте не дуже точний метод побудови

матриці ймовірностей переходу. Головна ідея цього методу полягає у обчисленні частоти вживання певних акордів після інших базуючись на вживаних акордових послідовностях. Реалізацію цього методу можна знайти у додатку Г.

Приклад матриці, отриманої вищезазначеним методом:

| | C | C# | D | D# | E | F | F# | G | G# | A | A# | B | Cm | C#m | Dm | D#m | Em | Fm | F#m | Gm | G#m | Am | A#m | Bm |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **C** | .04 | .01 | .13 | .01 | .01 | .26 | .01 | .13 | .01 | .01 | .02 | .01 | .01 | .01 | .15 | .01 | .04 | .02 | .01 | .01 | .01 | .06 | .01 | .01 |
| **C#** | .01 | .04 | .01 | .13 | .01 | .01 | .26 | .01 | .13 | .01 | .01 | .02 | .01 | .01 | .01 | .15 | .01 | .04 | .02 | .01 | .01 | .01 | .06 | .01 |
| **D** | .02 | .01 | .04 | .01 | .13 | .01 | .01 | .26 | .01 | .13 | .01 | .01 | .01 | .01 | .01 | .01 | .15 | .01 | .04 | .02 | .01 | .01 | .01 | .06 |
| **D#** | .01 | .02 | .01 | .04 | .01 | .13 | .01 | .01 | .26 | .01 | .13 | .01 | .06 | .01 | .01 | .01 | .01 | .15 | .01 | .04 | .02 | .01 | .01 | .01 |
| **E** | .01 | .01 | .02 | .01 | .04 | .01 | .13 | .01 | .01 | .26 | .01 | .13 | .01 | .06 | .01 | .01 | .01 | .01 | .15 | .01 | .04 | .02 | .01 | .01 |
| **F** | .13 | .01 | .01 | .02 | .01 | .04 | .01 | .13 | .01 | .01 | .26 | .01 | .01 | .01 | .06 | .01 | .01 | .01 | .01 | .15 | .01 | .04 | .02 | .01 |
| **F#** | .01 | .13 | .01 | .01 | .02 | .01 | .04 | .01 | .13 | .01 | .01 | .26 | .01 | .01 | .01 | .06 | .01 | .01 | .01 | .01 | .15 | .01 | .04 | .02 |
| **G** | .26 | .01 | .13 | .01 | .01 | .02 | .01 | .04 | .01 | .13 | .01 | .01 | .02 | .01 | .01 | .01 | .06 | .01 | .01 | .01 | .01 | .15 | .01 | .04 |
| **G#** | .01 | .26 | .01 | .13 | .01 | .01 | .02 | .01 | .04 | .01 | .13 | .01 | .04 | .02 | .01 | .01 | .01 | .06 | .01 | .01 | .01 | .01 | .15 | .01 |
| **A** | .01 | .01 | .26 | .01 | .13 | .01 | .01 | .02 | .01 | .04 | .01 | .13 | .01 | .04 | .02 | .01 | .01 | .01 | .06 | .01 | .01 | .01 | .01 | .15 |
| **A#** | .13 | .01 | .01 | .26 | .01 | .13 | .01 | .01 | .02 | .01 | .04 | .01 | .15 | .01 | .04 | .02 | .01 | .01 | .01 | .06 | .01 | .01 | .01 | .01 |
| **B** | .01 | .13 | .01 | .01 | .26 | .01 | .13 | .01 | .01 | .02 | .01 | .04 | .01 | .15 | .01 | .04 | .02 | .01 | .01 | .01 | .06 | .01 | .01 | .01 |
| **Cm** | .01 | .03 | .03 | .01 | .01 | .11 | .01 | .01 | .13 | .01 | .05 | .01 | .21 | .01 | .05 | .01 | .01 | .21 | .01 | .03 | .01 | .01 | .01 | .01 |
| **C#m** | .01 | .01 | .03 | .03 | .01 | .01 | .11 | .01 | .01 | .13 | .01 | .05 | .01 | .21 | .01 | .05 | .01 | .01 | .21 | .01 | .03 | .01 | .01 | .01 |
| **Dm** | .05 | .01 | .01 | .03 | .03 | .01 | .01 | .11 | .01 | .01 | .13 | .01 | .01 | .01 | .21 | .01 | .05 | .01 | .01 | .21 | .01 | .03 | .01 | .01 |
| **D#m** | .01 | .05 | .01 | .01 | .03 | .03 | .01 | .01 | .11 | .01 | .01 | .13 | .01 | .01 | .01 | .21 | .01 | .05 | .01 | .01 | .21 | .01 | .03 | .01 |
| **Em** | .13 | .01 | .05 | .01 | .01 | .03 | .03 | .01 | .01 | .11 | .01 | .01 | .01 | .01 | .01 | .01 | .21 | .01 | .05 | .01 | .01 | .21 | .01 | .03 |
| **Fm** | .01 | .13 | .01 | .05 | .01 | .01 | .03 | .03 | .01 | .01 | .11 | .01 | .03 | .01 | .01 | .01 | .01 | .21 | .01 | .05 | .01 | .01 | .21 | .01 |
| **F#m** | .01 | .01 | .13 | .01 | .05 | .01 | .01 | .03 | .03 | .01 | .01 | .11 | .01 | .03 | .01 | .01 | .01 | .01 | .21 | .01 | .05 | .01 | .01 | .21 |
| **Gm** | .11 | .01 | .01 | .13 | .01 | .05 | .01 | .01 | .03 | .03 | .01 | .01 | .21 | .01 | .03 | .01 | .01 | .01 | .01 | .21 | .01 | .05 | .01 | .01 |
| **G#m** | .01 | .11 | .01 | .01 | .13 | .01 | .05 | .01 | .01 | .03 | .03 | .01 | .01 | .21 | .01 | .03 | .01 | .01 | .01 | .01 | .21 | .01 | .05 | .01 |
| **Am** | .01 | .01 | .11 | .01 | .01 | .13 | .01 | .05 | .01 | .01 | .03 | .03 | .01 | .01 | .21 | .01 | .03 | .01 | .01 | .01 | .01 | .21 | .01 | .05 |
| **A#m** | .03 | .01 | .01 | .11 | .01 | .01 | .13 | .01 | .05 | .01 | .01 | .03 | .05 | .01 | .01 | .21 | .01 | .03 | .01 | .01 | .01 | .01 | .21 | .01 |
| **Bm** | .03 | .03 | .01 | .01 | .11 | .01 | .01 | .13 | .01 | .05 | .01 | .01 | .01 | .05 | .01 | .01 | .21 | .01 | .03 | .01 | .01 | .01 | .01 | .21 |

Отже, для кожного сегменту аудіофайлу, який містить один акорд, за допомогою швидкого перетворення Фур’є, ми створюємо нотний вектор, який являтиме собою спостережувану ознаку моделі. За допомогою косинусу подібності, ми отримуємо розподіл ймовірностей відносно кожного

музичного акорду (прихованого стану системи). Маючи інформацію про попередній акорд послідовності, ми отримуємо ймовірності переходу на інші акорди з матриці ймовірностей переходу. Ймовірності отримання даного нотного вектора з прихованих станів ми додаємо до ймовірностей переходу. Найбільша ймовірність визначатиме який акорд ми почули. Реалізація цього алгоритму знаходиться у додатку А.

# Висновок

В роботі розроблено програмне забезпечення, яке дозволяє користувачеві розпізнати з яких акордів складається музичний аудіофайл. Були досліджені методи пошуку початку та кінця акорду, побудови нотних векторів, та методи розпізнавання акордів за допомогою прихованих марковських моделей.

# Список літератури


1. Meinard Müller, Fundamentals of music processing, Springer International Publishing Switzerland 2015, ISBN 978-3-319-21945-5
2. Андрущак Григорій, Застосування перетворення Фур'є на прикладі музики, Національний університет «Києво-Могилянська академія», Україна, Київ, 2019
3. Андрущак Григорій, Розпізнавання акордів, Національний університет «Києво-Могилянська академія», Україна, Київ, 2020
4. Відповідність нот частотам [Електронний документ] – Режим доступу: URL: https://gist.github.com/i-Robi/8684800
5. Markus Schedl, Emilia Gómez and Julián Urbano (2014), "Music Information Retrieval: Recent Developments and Applications", Foundations and Trends® in Information Retrieval: Vol. 8: No. 2-3, pp 127-261. http://dx.doi.org/10.1561/1500000042
6. Mark Gales and Steve Young (2007), "The Application of Hidden Markov Models in Speech Recognition", Foundations and Trends® in Signal Processing Vol. 1, No. 3 195–304 2008 DOI: 10.1561/2000000004
7. Yuting Qi, John William Paisley and Lawrence Carin (2007), "Music Analysis Using Hidden Markov Mixture Models", IEEE Transactions on Signal Processing, Vol. 55, No. 11

# Додаток А

## Файл main.py

```
from dataclasses import dataclass
from pprint import pprint

import soundfile as sf

from chord_extractor import ChordExtractor
from hmm import HMM
from note_vector import process_sound
from utils import CHORDS

FILENAME = "samples/1.wav"
COS_WEIGHT = 0.7
HMM_WEIGHT = 0.3


@dataclass
class ResultChord:
    name: str
    start: float  # seconds
    end: float  # seconds
    confidence: float


def main():
    data, samplerate = sf.read(FILENAME)
    chord_timings = ChordExtractor(FILENAME).extract_v2()
    hmm = HMM()
    chords_found = list()
    for i, chord_t in enumerate(chord_timings):
        print(f"\n==========    {i + 1}    ==========")
        start, end = chord_t
        chords = process_sound(data[start:end])
        if len(chords_found) == 0:
            chord = sorted(chords, reverse=True, key=lambda x: x.confidence)[0]
            chords_found.append(
                ResultChord(
                    chord.name,
                    round(start / samplerate, 2),
                    round(end / samplerate, 2),
                    chord.confidence,
                )
            )
```

```
            print("chord\tconfidence")
            print(f"{chord.name}\t{chord.confidence}")
        else:
            prev_chord = chords_found[-1]
            next_chord_prob = hmm.next_chord_prob_normalized(prev_chord.name)
            print("chord\tcos\tHMM\tres")
            for c, n in zip(chords, next_chord_prob):
                print(
                    f"{c.name}\t"
                    + f"{round(c.confidence, 2)}\t"
                    + f"{round(n, 2)}\t"
                    + f"{round(c.confidence * COS_WEIGHT + n * HMM_WEIGHT, 2)}"
                )
            print("- - - - - - - - - - - - - -")
            chord = sorted(
                [
                    (c.name, c.confidence * COS_WEIGHT + n * HMM_WEIGHT)
                    for c, n in zip(chords, next_chord_prob)
                ],
                reverse=True,
                key=lambda x: x[1],
            )[0]
            chords_found.append(
                ResultChord(
                    chord[0],
                    round(start / samplerate, 2),
                    round(end / samplerate, 2),
                    chord[1],
                )
            )
            print("chord\tconfidence")
            print(f"{chord[0]}\t{chord[1]}")
            print(
                f"\nBest by cos:\t"
                + f"{sorted(chords, reverse=True, key=lambda x: x.confidence)[0].name}"
            )
            print(
                f"Best by HMM:\t"
                + f"{CHORDS[next_chord_prob.index(max(next_chord_prob))]}"
            )

    print("\n\nRESULT:\n")
    pprint(chords_found)


if __name__ == "__main__":
    main()
```

# Додаток Б

## Файл chord_extractor.py

```
import os
import shutil
from collections import namedtuple
from typing import Tuple

import numpy as np
import soundfile as sf
from matplotlib import pyplot as plt


class ChordExtractor:
    def __init__(self, filename: str) -> None:
        self.filename = filename
        self.data, self.samplerate = sf.read(self.filename)
        if hasattr(self.data[0], "__len__"):
            self.data = [sum(x) / len(x) for x in self.data]
        self.abs_data = [abs(amp) for amp in self.data]
        self.T = 1 / self.samplerate
        self.t = len(self.data) / self.samplerate
        self.N = len(self.data)
        self.t_vec = np.arange(self.N) * self.T

    def extract(
        self,
        top_perc: float = 0.001,
        min_dur: float = 0.5,
        show_plot: bool = False,
        return_data: bool = False,
    ) -> Tuple[Tuple[int]]:
        class Point:

            instances = []
            top_perc = []
            top_perc_value = 0

            def __init__(self, x, y, i):
                self.x = x
                self.y = y
                self.i = i
                self.y_cmp = abs(y) ** 2
```

```
    def __repr__(self):
        return str(self.y_cmp)

    @classmethod
    def sort_process_normalize(cls):
        cls.instances.sort(key=lambda p: p.y_cmp, reverse=True)
        temp = 0
        max_y_cmp = cls.instances[0].y_cmp
        for point in cls.instances:
            point.y_norm = point.y_cmp / max_y_cmp
            temp += point.y_norm
        inst_len = len(cls.instances)
        cls.top_perc = cls.instances[: int(inst_len * top_perc)]
        cls.top_perc_value = cls.top_perc[-1].y_norm

Point.instances = [
    Point(t, a, i) for a, t, i in zip(self.data, self.t_vec, range(self.N))
]
Point.sort_process_normalize()

if show_plot:
    plt.plot(self.t_vec, [p.y_cmp for p in Point.instances])
    plt.plot(self.t_vec, [Point.top_perc_value] * len(self.t_vec))
    plt.title("Sorted amplitudes")
    plt.text(
        2,
        Point.top_perc_value + 0.05 * Point.top_perc_value,
        f"Top {top_perc * 100}% are above this line",
    )
    plt.show()

result = [Point.top_perc[0]]
for point in Point.top_perc[1:]:
    is_in_scope = True
    for r in result:
        if abs(point.x - r.x) <= min_dur:
            is_in_scope = False
            break
    if is_in_scope:
        result.append(point)
temp = sorted([p.i for p in result])
temp1 = temp[1:] + [self.N]
result = tuple((start, end - 1) for start, end in zip(temp, temp1))
if return_data:
    return tuple([self.data[start:end] for start, end in result])
else:
    return tuple(result)
```

```
def extract_v2(
    self,
    min_x: int = 1500,
    f_amp: float = 0.15,
    min_dur: float = 0.5,
    return_data: bool = False,
) -> Tuple[Tuple[int]]:
    Point = namedtuple("Point", ["time", "amplitude"])
    max_amp = max(self.data)
    local_extrema = [Point(t, abs(a)) for t, a in enumerate(self.data)]
    threshold = self.N // min_x
    while len(local_extrema) > threshold:
        local_extrema_temp = []
        prev_point = Point(0, 0.0)
        for i, p in enumerate(local_extrema):
            if i == len(local_extrema) - 1:
                local_extrema_temp.append(p)
            elif (
                p.amplitude > prev_point.amplitude
                and p.amplitude > local_extrema[i + 1].amplitude
            ):
                local_extrema_temp.append(p)
            prev_point = p
        local_extrema = local_extrema_temp
    result = list()
    temp = list()
    for this_point in local_extrema:
        if this_point.amplitude > f_amp * max_amp:
            if len(temp) == 0:
                temp.append(this_point.time)
        else:
            if len(temp) == 2 and temp[1] - temp[0] > min_dur * self.samplerate:
                result.append(tuple(temp))
                temp = list()
            elif len(temp) == 1:
                temp.append(this_point.time - 1)
    if return_data:
        return tuple([self.data[start:end] for start, end in result])
    else:
        return tuple(result)

def show_chord_plot(self, parts: Tuple[Tuple[int]]) -> None:
    local_extrema_res = [0] * len(self.data)
    for start, end in parts:
        for i in range(start, end):
            local_extrema_res[i] = 1
```

```
        plt.plot(self.t_vec, self.data)
        plt.plot(self.t_vec, local_extrema_res, linewidth=2)
        plt.xlabel('time')
        plt.ylabel('amplitude')
        plt.title('Result')
        plt.show()

    def save_parts(self, parts: Tuple[Tuple[int]]) -> None:
        if not os.path.exists("results"):
            os.mkdir("results")
        path = "results/{}".format("_".join(self.filename[-5:].split(".")[-2:]))
        if os.path.exists(path):
            shutil.rmtree(path)
        os.mkdir(path)

        for i, part in enumerate(parts):
            start, end = part
            sf.write(
                "{}/{}.wav".format(path, i + 1),
                self.data[start : end + 1],
                self.samplerate,
            )


if __name__ == "__main__":
    filename = "./samples/1.wav"
    chord_extractor = ChordExtractor(filename)
    # parts = chord_extractor.extract()
    parts = chord_extractor.extract_v2()
    print(parts)
    chord_extractor.show_chord_plot(parts)
    chord_extractor.save_parts(parts)
```

# Додаток В

## Файл note_vector.py

```
from dataclasses import dataclass
from pprint import pprint
from typing import List

import numpy as np
import soundfile as sf

from utils import CHORD_TO_VECTOR, CHORDS, NOTE_TO_FREQ


@dataclass
class Chord:
    name: str
    note_vector: np.ndarray
    confidence: float


def note_vector(fft_res: np.ndarray, note_width: float = 0.01) -> np.ndarray:
    res_vector = np.zeros(12)
    for i, hzs in enumerate(NOTE_TO_FREQ.values()):
        res_vector[i] += sum(
            [
                sum([fft_res[int(i)]])
                for hz in hzs
                for i in range(int(hz * (1 - note_width)),
                               int(hz * (1 + note_width)))
            ]
        )
    return res_vector / max(res_vector)


def notes_to_chords(n_v: np.ndarray) -> List[Chord]:
    result = []
    for chord in CHORDS:
        chord_matrix_vector = CHORD_TO_VECTOR[chord]
        cos_sim = np.dot(n_v, chord_matrix_vector) / (
            np.linalg.norm(n_v) * np.linalg.norm(chord_matrix_vector)
        )
        result.append(Chord(chord, n_v, cos_sim))
    return result
```

```python
def gauss_w(n: int, frame: int) -> float:
    Q = 0.5
    a = (frame - 1) / 2
    t = (n - a) / (Q * a)
    t = t * t
    return np.exp(-t / 2)


def process_sound(data: np.ndarray, gauss_window: bool = True) -> Chord:
    if hasattr(data[0], "__len__"):
        data = [sum(x) / len(x) for x in data]

    fft_res = np.abs(np.fft.fft(data)[0 : int(len(data) / 2)] / len(data) * 2)
    if gauss_window:
        fft_res = np.array(
            [value * gauss_w(i, len(fft_res)) for i, value in enumerate(fft_res)]
        )
    return notes_to_chords(note_vector(fft_res))


if __name__ == "__main__":
    data, _ = sf.read("samples/test.wav")
    res = tuple(process_sound(data))
    print("Chord\tConfidence")
    for res_str in map(lambda c: f"{c.name}\t{c.confidence}", res):
        print(res_str)
```

# Додаток Г

## Файл hmm.py

```
import csv
from dataclasses import dataclass
from functools import lru_cache
from typing import Tuple

import numpy as np

from utils import CHORDS, NOTES


@dataclass(frozen=True)
class Progression:
    name: str
    scale: str
    degrees: Tuple[int]


@dataclass(frozen=True)
class ChordProgression:
    sequence: Tuple[str]
    progression: Progression


with open("common_chord_progressions.txt") as f:
    f = f.read().split("\n\n")

split_to_str = lambda text: tuple(text.split("\n"))
split_to_ints = lambda text: tuple(
    tuple(int(i) - 1 for i in row.split()) for row in text.split("\n")
)
progressions_raw = {
    "major": (split_to_str(f[0]), split_to_ints(f[2])),
    "minor": (split_to_str(f[1]), split_to_ints(f[3])),
}
COMMON_PROGRESSIONS = tuple(
    Progression(name, scale, degrees)
    for scale, prog in progressions_raw.items()
    for name, degrees in zip(prog[0], prog[1])
)
```

```
@lru_cache(maxsize=24)
def key_to_degrees(key: str, scale: str = "major") -> Tuple[str]:
    a = NOTES.index(key)
    if scale == "major":
        # degrees = ("I", "ii", "iii", "IV", "V", "vi", "VII")
        minors = ("", "m", "m", "", "", "m", "")
        indexes = (a, a + 2, a + 4, a + 5, a + 7, a + 9, a + 11)
    elif scale == "minor":
        # degrees = ("i", "II", "III", "iv", "v", "VI", "VII")
        minors = ("m", "", "", "m", "m", "", "")
        indexes = (a, a + 2, a + 3, a + 5, a + 7, a + 8, a + 10)
    else:
        raise ValueError(f'scale "{scale}" does not exist')
    return tuple(f"{NOTES[i % 12]}{m}" for i, m in zip(indexes, minors))


def all_possible_chord_progressions(
    progressions: Tuple[Progression],
) -> Tuple[ChordProgression]:
    sequence = lambda prog, key: tuple(
        key_to_degrees(key, prog.scale)[chord] for chord in prog.degrees
    )
    return tuple(
        ChordProgression(sequence(prog, key), prog)
        for prog in progressions
        for key in NOTES
    )


class HMM:
    def __init__(self, csv_file=None) -> None:
        if csv_file:
            self._import_from_csv(csv_file)
        else:
            chord_progressions = all_possible_chord_progressions(COMMON_PROGRESSIONS)
            transition_matrix = np.zeros((len(CHORDS),) * 2)
            for i in range(4):
                for y, chord in enumerate(CHORDS):
                    target_seq = filter(
                        lambda x: x.sequence[i] == chord, chord_progressions
                    )
                    for next_chord in target_seq:
                        transition_matrix[y][
                            CHORDS.index(next_chord.sequence[(i + 1) % 4])
                        ] += 1
            found_partition = 0.85
            for y, row in enumerate(transition_matrix):
```

```
                norm = found_partition / row.sum()
                zero_value = (1 - found_partition) / (row == 0).sum()
                for x, v in enumerate(row):
                    if v == 0:
                        transition_matrix[y][x] = zero_value
                    else:
                        transition_matrix[y][x] *= norm
            self._transition_matrix = transition_matrix

    @property
    def transition_matrix(self) -> np.ndarray:
        return self._transition_matrix

    @lru_cache(maxsize=len(CHORDS))
    def next_chord_prob(self, chord: str) -> Tuple[np.ndarray]:
        return tuple(self.transition_matrix[CHORDS.index(chord)])

    @lru_cache(maxsize=len(CHORDS))
    def next_chord_prob_normalized(self, chord: str) -> Tuple[np.ndarray]:
        row = self.transition_matrix[CHORDS.index(chord)]
        return tuple(row / max(row))

    def export_to_csv(self, round_values=True) -> None:
        with open("transition_matrix.csv", "w") as csvfile:
            csvwriter = csv.writer(csvfile, delimiter=",")
            csvwriter.writerow([""] + list(CHORDS))
            for chord, row in zip(CHORDS, self.transition_matrix):
                csvwriter.writerow(
                    [chord]
                    + list([str(round(v, 2))[1:] if round_values else v for v in row])
                )

    def _import_from_csv(self, filename: str) -> None:
        with open(filename) as csvfile:
            csvreader = csv.reader(csvfile, delimiter=",")
            transition_matrix = list()
            for row in list(csvreader)[1:]:
                transition_matrix.append(np.array([float(v) for v in row[1:]]))
            self._transition_matrix = np.array(transition_matrix)


if __name__ == "__main__":
    print("TRANSITION MATRIX:")
    for row in HMM(csv_file="transition_matrix.csv").transition_matrix:
        print([round(v, 2) for v in row])
    HMM().export_to_csv()
```

# Додаток Ґ

## Файл utils.py

```
import csv
import json
from pprint import pprint
from typing import Tuple

import numpy as np
import soundfile as sf
from matplotlib import pyplot as plt

################################################################################
###############                                                  ###############
###############                    CONSTANTS                     ###############
###############                                                  ###############
################################################################################

with open("chord_matrix.csv", "r") as f:
    _chord_matrix_csv_list = list(csv.reader(f, delimiter=","))
with open("notes.json", "r") as notes_json:
    _notes_json_dict = json.loads(notes_json.read())

NOTE_TO_FREQ = {k: tuple(v) for k, v in _notes_json_dict.items()}
CHORDS = tuple(_chord_matrix_csv_list[0][1:])
NOTES = tuple(NOTE_TO_FREQ.keys())
CHORD_TO_VECTOR = {
    chord: tuple(
        [
            v[i]
            for v in np.array([i[1:] for i in
_chord_matrix_csv_list[1:]]).astype("int")
        ][::-1]
    )
    for i, chord in enumerate(CHORDS)
}

################################################################################
###############                                                  ###############
###############                    FUNCTIONS                     ###############
###############                                                  ###############
################################################################################
```

```
def make_wav(
    seq: Tuple[str, int],
    s_r: int = 44100,
    octave: int = 5,
    dcoef: float = 1,
    show_plot: bool = True,
) -> None:
    T = 1 / s_r  # sampling period
    N = s_r * sum([x[1] for x in seq])  # total points in signal
    t_vec = np.arange(N) * T  # time vector for plotting
    result = []
    for chord, t in seq:
        freqs = [
            NOTE_TO_FREQ[NOTES[i]][octave]
            for i, n in enumerate(CHORD_TO_VECTOR.get(chord))
            if n == 1
        ]
        omegas = [2 * np.pi * freq for freq in freqs]
        t_vec_local = np.arange(s_r * t) * T
        y_list = [
            np.sin(omega * t_vec_local) * np.exp(-dcoef * t_vec_local)
            for omega in omegas
        ]
        y = y_list[0]
        for y_l_e in y_list[1:]:
            y = np.add(y, y_l_e)
        y_max_index = np.where(y == np.amax(y))[0][0]
        y = [value / y[y_max_index] for value in y]
        result.extend(y)
    sf.write("samples/test.wav", result, s_r)
    if show_plot:
        try:
            plt.plot(t_vec, result)
        except ValueError as e:
            err_vals = [
                int(e.args[0].split()[-3][1:-2]),
                int(e.args[0].split()[-1][1:-2]),
            ]
            print(err_vals)
            if err_vals[0] > err_vals[1]:
                plt.plot(t_vec[: (err_vals[1] - err_vals[0])], result)
            else:
                plt.plot(t_vec, result[: (err_vals[0] - err_vals[1])])
        plt.ylabel("Amplitude")
        plt.xlabel("Time (seconds)")
        plt.show()
```

```
if __name__ == "__main__":
    print("NOTE_TO_FREQ:")
    pprint(NOTE_TO_FREQ)
    print("CHORDS:")
    print(CHORDS)
    print("NOTES:")
    print(NOTES)
    print("CHORD_TO_VECTOR:")
    pprint(CHORD_TO_VECTOR)
    seq = [("Am", 1), ("F", 1), ("C", 1), ("E", 1)]
    make_wav(seq)
```

# Додаток Д

## Файл notes.json


```
{
   "C": [16.35, 32.70, 65.41, 130.81, 261.63, 523.25, 1046.50, 2093.00, 4186.01],
  "C#": [17.32, 34.65, 69.30, 138.59, 277.18, 554.37, 1108.73, 2217.46, 4434.92],
   "D": [18.35, 36.71, 73.42, 146.83, 293.66, 587.33, 1174.66, 2349.32, 4698.64],
  "D#": [19.45, 38.89, 77.78, 155.56, 311.13, 622.25, 1244.51, 2489.02, 4978.03],
   "E": [20.60, 41.20, 82.41, 164.81, 329.63, 659.26, 1318.51, 2637.02],
   "F": [21.83, 43.65, 87.31, 174.61, 349.23, 698.46, 1396.91, 2793.83],
  "F#": [23.12, 46.25, 92.50, 185.00, 369.99, 739.99, 1479.98, 2959.96],
   "G": [24.50, 49.00, 98.00, 196.00, 392.00, 783.99, 1567.98, 3135.96],
  "G#": [25.96, 51.91, 103.83, 207.65, 415.30, 830.61, 1661.22, 3322.44],
   "A": [27.50, 55.00, 110.00, 220.00, 440.00, 880.00, 1760.00, 3520.00],
  "A#": [29.14, 58.27, 116.54, 233.08, 466.16, 932.33, 1864.66, 3729.31],
   "B": [30.87, 61.74, 123.47, 246.94, 493.88, 987.77, 1975.53, 3951.07]
}
```

# Додаток Е

## Файл chord_matrix.csv

,C,C#,D,D#,E,F,F#,G,G#,A,A#,B,Cm,C#m,Dm,D#m,Em,Fm,F#m,Gm,G#m,Am,A#m,Bm
B,0,0,0,0,1,0,0,1,0,0,0,1,0,0,0,0,1,0,0,0,1,0,0,1
A#,0,0,0,1,0,0,1,0,0,0,1,0,0,0,0,1,0,0,0,1,0,0,1,0
A,0,0,1,0,0,1,0,0,0,1,0,0,0,0,1,0,0,0,1,0,0,1,0,0
G#,0,1,0,0,1,0,0,0,1,0,0,0,0,1,0,0,0,1,0,0,1,0,0,0
G,1,0,0,1,0,0,0,1,0,0,0,0,1,0,0,0,1,0,0,1,0,0,0,0
F#,0,0,1,0,0,0,1,0,0,0,0,1,0,0,0,1,0,0,1,0,0,0,0,1
F,0,1,0,0,0,1,0,0,0,0,1,0,0,0,1,0,0,1,0,0,0,0,1,0
E,1,0,0,0,1,0,0,0,0,1,0,0,0,1,0,0,1,0,0,0,0,1,0,0
D#,0,0,0,1,0,0,0,0,1,0,0,1,1,0,0,1,0,0,0,0,1,0,0,0
D,0,0,1,0,0,0,0,1,0,0,1,0,0,0,1,0,0,0,0,1,0,0,0,1
C#,0,1,0,0,0,0,1,0,0,1,0,0,0,1,0,0,0,0,1,0,0,0,1,0
C,1,0,0,0,0,1,0,0,1,0,0,0,1,0,0,0,0,1,0,0,0,1,0,0

# Додаток Є

## Файл common_chord_progressions.txt

**I IV V I**
**I vi IV V**
**ii V I ii**
**I vi ii V**
**I V vi IV**
**I IV vi V**
**I iii IV V**
**I IV I V**
**I IV ii V**

**i VI VII i**
**i iv VII i**
**i iv v i**
**i VI III VII**
**ii V i ii**
**i iv v i**
**VI VII i i**
**i VII VI VII**
**i iv i i**

**1 4 5 1**
**1 6 4 5**
**2 5 1 2**
**1 6 2 5**
**1 5 6 4**
**1 4 6 5**
**1 3 4 5**
**1 4 1 5**
**1 4 2 5**

**1 6 7 1**
**1 4 7 1**
**1 4 5 1**
**1 6 3 7**
**2 5 1 2**
**1 4 5 1**
**6 7 1 1**
**1 7 6 7**
**1 4 1 1**

# Додаток Ж

## Файл requirements.txt

```
appdirs==1.4.4
black==21.5b1
cffi==1.14.5
click==8.0.1
cycler==0.10.0
kiwisolver==1.3.1
matplotlib==3.4.2
mypy-extensions==0.4.3
numpy==1.20.3
pathspec==0.8.1
Pillow==8.2.0
pycparser==2.20
pyparsing==2.4.7
python-dateutil==2.8.1
regex==2021.4.4
six==1.16.0
SoundFile==0.10.3.post1
toml==0.10.2
```