



# Identification des titres musicaux

Méthode de fingerprinting

Othman EL HOUFI  
Mohamed DIAWARA

**M1 Systèmes Intelligents et Communicants**

Rapporteur : Dan VODISLAV  
Tuteur technique : Dimitris KOTZINOS  
Encadrant de gestion de projet : Tianxiao LIU

11 juin 2021

# **Remerciement**

C'est avec un grand plaisir que nous réservons cette page en signe de gratitude et de profonde reconnaissance à tous ceux qui ont bien voulu apporter l'assistance nécessaire au bon déroulement de ce travail.

Nos remerciements les plus sincères vont à M. Dimitris Kotzinos, notre tuteur technique et enseignant à l'Université de Cergy Pontoise pour son soutien et ses recommandations judicieuses.

Nous remercions M. LIU Tianxiao, notre encadrant à l'Université de Cergy pour sa confiance, ses conseils, son accompagnement, son encouragement ainsi que pour la qualité et la complémentarité de son encadrement. Il nous a ouvert son savoir-faire et ses expériences pour un insigne accroissement de connaissances acquises.

M. LIU nous nous sommes très honorés de travailler avec vous et nous tenons à vous adresser nos remerciements les plus profondes. Nous adressons nos sincères remerciements à tous les membres du jury qui nous ont fait l'honneur d'accepter de prendre part à ce jury et surtout de lire et d'expertiser notre travail. Et nous souhaitons également remercier tout le corps professoral du département Informatique de l'Université de Cergy pour la qualité de l'enseignement.

En somme, pour tous ceux qui n'ont pas hésité à nous accorder leur soutien, leur amitié ou leur expérience tout au long de nos années d'études et qui ont accompagné nos pas d'une part ou d'une autre, veuillez accepter nos modestes remerciements.

# Table des matières

<b>1</b>	<b>Introduction</b>	<b>5</b>
1.1	Contexte du projet . . . . .	5
1.2	Mise en scénario . . . . .	5
1.3	Objectifs du projet . . . . .	6
1.4	Organisation du rapport . . . . .	7
<b>2</b>	<b>Présentation et spécification du projet</b>	<b>8</b>
2.1	Fonctionnalités attendues . . . . .	8
2.2	Conception globale du projet . . . . .	8
2.3	Problématiques identifiées et solutions envisagées . . . . .	9
2.4	Environnement de travail . . . . .	10
<b>3</b>	<b>Échantillonnage : de l'Analogique au Digital</b>	<b>11</b>
3.1	Fonctionnement de l'oreille humain : transmission de signal . . . . .	11
3.2	Échantillonnage . . . . .	11
<b>4</b>	<b>Traitemennt du signal acoustique</b>	<b>13</b>
4.1	Analyse de la problématique . . . . .	13
4.2	Etat de l'art : études des solutions existantes . . . . .	14
4.2.1	Représentation temporelle . . . . .	14
4.2.2	Représentation fréquentielle . . . . .	15
4.2.3	Représentation temporelle-fréquentielle . . . . .	16
4.3	Solution proposée et sa mise en œuvre . . . . .	18
4.3.1	Extraction des pics spectraux . . . . .	19
4.3.2	Procédure algorithmique complète . . . . .	21
4.3.3	Mise en œuvre . . . . .	22
4.4	Tests et certifications de la solution . . . . .	24
4.4.1	Tests préliminaires . . . . .	25
4.4.2	Tests de robustesse de la constellation . . . . .	26
<b>5</b>	<b>Création d'une empreinte acoustique</b>	<b>31</b>
5.1	Analyse de la problématique . . . . .	31
5.2	État de l'art : études des solutions existantes . . . . .	32
5.2.1	Approche de superposition . . . . .	32
5.2.2	Approche de hachage combinatoire . . . . .	34
5.3	Solution proposée et sa mise en œuvre . . . . .	35
5.3.1	Construction des paires . . . . .	35
5.3.2	Conversion des repères en valeurs de hachage. . . . .	37
<b>6</b>	<b>Base de données : stockage et recherche</b>	<b>39</b>
6.1	Architecture initiale de la base de données . . . . .	40
6.1.1	Mise en œuvre . . . . .	40
6.1.2	Tests et performance . . . . .	41
6.2	Optimisation de la base de données . . . . .	42
6.2.1	Mise en œuvre . . . . .	42
6.2.2	Tests et performance . . . . .	44

<b>7 Rendu final</b>	<b>45</b>
7.1 Interface utilisateur finale . . . . .	45
7.2 Tests utilisateur et certification . . . . .	45
7.3 Autres tests et certifications . . . . .	48
7.3.1 Tests sur une base de données de 50 musiques . . . . .	48
7.3.2 Tests sur une base de données de 8000 musiques . . . . .	49
<b>8 Gestion de projet</b>	<b>50</b>
8.1 Méthode de gestion . . . . .	50
8.1.1 Organisation du travail en équipe : . . . . .	50
8.1.2 Gestion du temps . . . . .	50
8.1.3 Outils de travail : . . . . .	51
8.1.4 Difficultés rencontrés dans la gestion du projet et solution apportés : . . . . .	51
8.2 Répartition de tâches . . . . .	52
<b>9 Conclusion</b>	<b>53</b>
<b>10 Perspectives</b>	<b>54</b>

# 1 Introduction

## 1.1 Contexte du projet

L'identification automatique de titres musicaux fait l'objet de nombreuses recherches, en particulier dans le cadre de l'indexation de larges bases de données et du monitoring de flux de broadcast.

Aujourd'hui les musiques font la base de toute plateforme diffusant un contenu multimédia, comme la radio, les chaînes de télévision, et bien-sûr les géantes plateformes que nous pouvons trouver sur internet telle que YouTube, Spotify, Facebook, Instagram, Netflix et autres. Parmi les problèmes communs que nous rencontrons sur toutes ces plateformes il y a le problème de la diffusion d'un contenu qui ne respecte pas les droits d'auteur, notamment le partage des musiques sans avoir ces droits.

Alors comment pouvons-nous détecter ce comportement illégal, ou plus précisément, existe-t-il un système capable d'identifier une musique par un segment d'audio ?

Cette question nous oblige à poser une série de questions ciblant le côté technique de ce système : comment pouvons-nous comparer un segment d'audio à une musique ? comment pouvons-nous réaliser cette comparaison sur une large base de données de musiques ? combien coûtera cette comparaison/-recherche en termes de temps et de mémoire ?

Il existe plusieurs systèmes intelligents capables de réaliser cette tâche, et le plus connu s'agit de *Shazam*, ce système qui est aussi sous forme d'une application mobile qui peut être utilisée pour identifier une musique qui se joue dans votre entourage en utilisant le micro de votre smartphone en temps réel. Ce logiciel utilise le microphone du téléphone pour capturer un échantillon de musique jouée. Une empreinte acoustique est créée à partir de cet échantillon, elle est comparée à la base de données centrale de la société. Ce qui fait de *Shazam* un système très intelligent et largement utilisé, c'est l'implémentation de la technique du *fingerprinting*. Cette technique est basée sur l'extraction de pics spectraux qui sont associés par paires, ce qui permet de construire une constellation pour chaque signal. Le 11 décembre 2017, Apple rachète *Shazam* sans donner d'indication sur le prix d'acquisition. Selon plusieurs sites spécialisés, le montant se situerait autour de 400 millions de dollars.

Le but de ce projet est de créer une application qui propose les mêmes fonctionnalités que *Shazam* tout en passant par toutes les étapes nécessaires y compris l'échantillonnage du signal, le traitement de ce signal et la création voire aussi le stockage des empreintes acoustiques.

## 1.2 Mise en scénario

Supposons que vous entendiez une chanson dans un restaurant, dans un centre commercial ou dans une voiture, et que vous souhaitez en savoir plus sur cette chanson. Par exemple, vous voulez connaître le titre de la chanson ou le nom de l'interprète ou de l'artiste. Les services de reconnaissance de musiques récents comme *Shazam* aident les utilisateurs dans de telles situations en identifiant l'enregistrement audio et en fournissant des informations appropriées sur le contenu. Un scénario typique est qu'un utilisateur, également appelé client, enregistre un court fragment audio de la chanson inconnue à l'aide d'un smartphone. Le fragment audio est ensuite converti en "empreintes audio", qui sont des caractéristiques audio compactes et descriptives. Ces empreintes sont transmises au service d'identification, également appelé serveur. Le serveur héberge diverses ressources de données, notamment une base de données d'empreintes des morceaux de musiques connus, ainsi qu'une base de métadonnées qui contient des informations de contenu liées à ces enregistrements. Le serveur reçoit les empreintes digitales de requête envoyées par le client et les compare avec les empreintes digitales contenues dans la base de

données. Cette étape est généralement réalisée par une consultation efficace de la base de données, soutenue par des structures d'index appropriées. En cas d'identification réussie, le serveur récupère les informations de contenu liées aux empreintes identifiées et renvoie les métadonnées souhaitées au client. La figure suivante présente un aperçu schématique du modèle client-serveur sous-jacent du service de fourniture de métadonnées décrit.

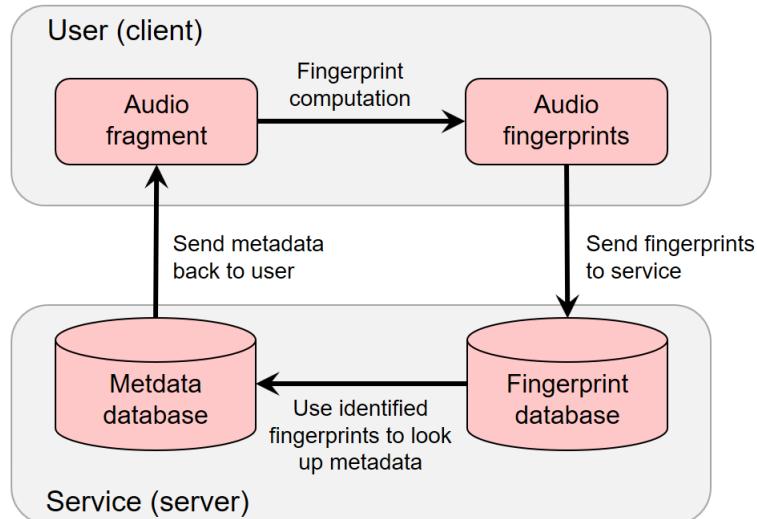


FIGURE 1.1 – Client-Server Processing Pipeline

La tâche doit pouvoir être effectuée dans des milieux très perturbés sur des enregistrements de qualité médiocre, avec des échantillons de quelques secondes, en temps réel, avec peu de ressources computationnelles.

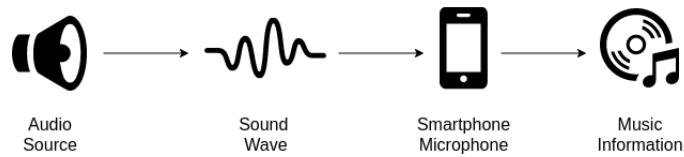
### 1.3 Objectifs du projet

La grande quantité de données disponibles et diffusées de manière continue sur tous les médias (radio, internet, télévision) pose le problème de l'exploitation efficace des contenus et du contrôle de sa diffusion. Dans le cadre du contrôle de flux multimédia, on cherche entre autres à identifier de manière robuste la donnée diffusée. Cette identification peut servir au contrôle des droits d'auteur, à la production de statistiques publicitaires, etc.

L'identification audio a pour but d'assigner son titre à une chanson diffusée. Dans le cadre de ce rapport, on s'intéresse à une identification par fingerprinting et par mise en correspondance d'une requête audio avec un élément d'une large base de données. L'identification par fingerprinting est basée sur l'extraction de caractéristiques qui décrivent de manière concise, unique et robuste les différents titres musicaux. Cette représentation est basée sur des propriétés acoustiques.

Pour cela nous avons besoin d'utiliser les différentes techniques de traitement de signal (échantillonage, transformée de Fourier, spectrogramme, filtres...), ainsi que l'optimisation des requêtes pour une recherche rapide dans une large base de données.

L'application finale donnera une possibilité à l'utilisateur d'identifier une musique diffusée dans son entourage en utilisant le microphone de son appareil (ordinateur ou smartphone), ou bien à travers un segment audio sous forme mp3 qui servira comme un échantillon.



## 1.4 Organisation du rapport

En premier temps nous allons expliquer le processus d'échantillonnage d'un signal acoustique via un microphone dans le chapitre *Échantillonnage : de l'Analogue au Digital*.

Une fois le signal est numérisé nous passerons à l'étape la plus importante qui s'agit de l'extraction des caractéristiques importantes du signal numérique ainsi résoudre les problématiques liées aux distorsions et mémoire. Cette opération est décrite en détails dans le chapitre *Traitements du signal acoustique*.

Ensuite dans le chapitre *Création d'empreinte acoustique* nous allons étudier le résultat retourner par le traitement du signal pour créer une empreinte compressée de ce signal ainsi répondre aux problématiques de mémoire et de vitesse de recherche.

Finalement dans le chapitre *Base de donnée : stockage et recherche*, nous allons présenter en détail l'architecture de notre base de donnée relationnelle voire aussi les optimisations effectuées aux requêtes et le gain en mémoire/recherche résultant de ces optimisations.

L'étude et les tests détaillés de la qualité de notre application dans le cas d'un signal pur et aussi dans la présence des distorsions seront présentés dans le chapitre chapitre *Rendu final*.

## 2 Présentation et spécification du projet

### 2.1 Fonctionnalités attendues

Dans ce projet nous allons réaliser l'échantillonnage d'un signal acoustique à travers un microphone voire aussi à travers un fichier *mp3*, ensuite le traitement de ce signal afin d'extraire ses caractéristiques importantes, puis la création d'une empreinte associée à ce signal et finalement le stockage et la recherche des différentes empreintes dans une large base de données.

Une fois l'application aboutie, l'utilisateur aura la possibilité de :

1. Reconnaître une chanson à partir du microphone.
2. Reconnaître une chanson à partir d'un fichier *mp3*.
3. Traiter, Hacher, Stocker une ou plusieurs musiques dans la base de données.
4. Afficher les détails de la base de données.
5. Réinitialiser la base de données.

Dans notre programme, chaque chanson a une empreinte qui lui est associée. Quand on demande à notre programme de reconnaître un morceau, on décompose le son et le transforme en empreinte, puis on le compare à celles présentes dans sa base de données et on retourne une correspondance si elle existe.

Tout ces opérations (échantillonnage, traitement de signal, hachage, stockage et recherche) seront réalisées d'une manière scientifique voire technique où l'on étudiera des solutions existantes en regardant leurs avantages et leurs inconvénients, puis nous allons présenter une solution détaillée à chaque problème rencontré tout en certifiant sa validité avec des tests assez exhaustifs.

### 2.2 Conception globale du projet

L'identification audio se déroule en plusieurs phases :

1. Nous pré-calculons les empreintes digitales à partir d'une très grande base de données de morceaux de musique. Différents marqueurs peuvent être générés, mais ils sont souvent basés sur une analyse temps-fréquence du signal (*spectrogramme*).
2. Toutes ces empreintes digitales sont placées dans une base de données d'empreintes digitales qui est mise à jour chaque fois qu'une nouvelle chanson est ajoutée dans la base de données de chansons.
3. Lorsqu'un utilisateur utilise l'application, il enregistre d'abord la musique actuelle avec le microphone de l'ordinateur.
4. Pour un signal "requête", l'application applique l'algorithme d'empreinte digitale sur l'enregistrement de la même manière que pour les éléments de la base de données.
5. L'application vérifie si cette empreinte digitale correspond à l'une de ses empreintes digitales déjà présentes dans la base de données de chansons. Les algorithmes de mise en correspondance sont basés sur une recherche soit exacte, soit au plus proche voisin, soit statistique.
  - Si non, il informe l'utilisateur que la musique ne peut pas être identifiée.
  - Si oui, il recherche les métadonnées associées aux empreintes digitales (nom de la chanson, nom de l'artiste) et la restitue à l'utilisateur.

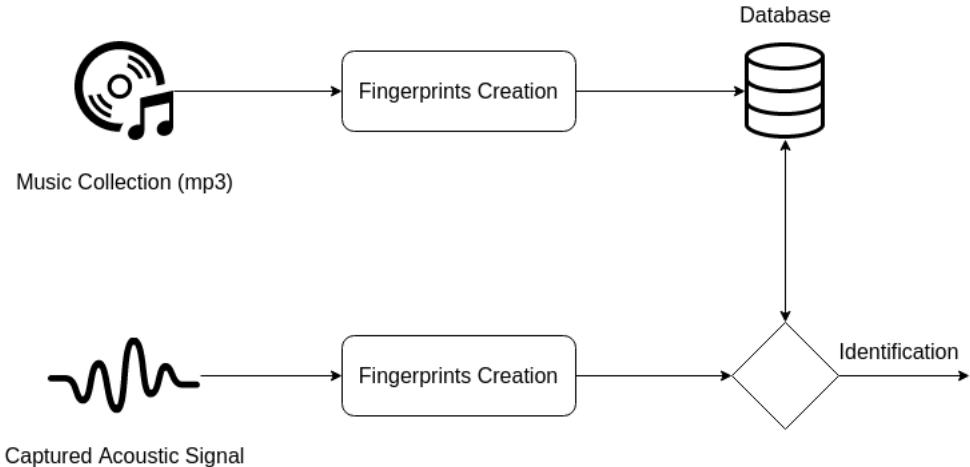


FIGURE 2.1 – Processing Pipeline

## 2.3 Problématiques identifiées et solutions envisagées

Les systèmes de reconnaissance musicale du monde réel doivent être robustes et efficaces sur le plan informatique, ce qui entraîne un certain nombre de défis techniques à résoudre. En particulier, les empreintes audio utilisées dans ces systèmes doivent répondre à certaines exigences, notamment une spécificité, une robustesse, une compacité et une évolutivité élevées.

- **Spécificité** : Les empreintes audio doivent posséder une spécificité élevée, de sorte que même un fragment audio de quelques secondes seulement suffise à identifier de manière fiable l'enregistrement correspondant et à le distinguer de millions d'autres.
- **Robustesse** : Pour une identification fiable, les empreintes digitales doivent être résistantes au bruit de fond et aux distorsions du signal telles que la compression avec perte, le décalage de hauteur, la mise à l'échelle temporelle, l'égalisation ou la compression dynamique.
- **Compacité** : Les empreintes audio doivent être de petite taille afin de pouvoir être transmises sur des canaux à bande passante limitée et être facilement stockées et indexées du côté de la base de données.
- **Évolutivité** : Pour pouvoir s'adapter à des millions d'enregistrements, le calcul des empreintes audio doit être simple et efficace - une exigence qui s'impose également lorsque les empreintes sont calculées sur des appareils mobiles dont la puissance de traitement est limitée.

L'amélioration d'une certaine exigence implique souvent une perte de performance dans une autre, et il faut faire face à un compromis délicat entre des principes contradictoires. Par exemple, l'amélioration de la robustesse conduit généralement à une augmentation des identifications erronées (faux positifs), ce qui détériore la précision du système d'identification. De même, même si elle est bénéfique pour des raisons de compacité et de calcul, une réduction excessive de la taille de l'empreinte digitale affecte négativement la capacité de discrimination. Inversement, les empreintes digitales d'une spécificité et d'une robustesse élevées peuvent ne pas être utilisables dans la pratique si leur calcul nécessite une puissance de traitement importante.

Les solutions que nous avons envisagé pour ces problématiques sont :

- Transformation du signal en *Spectrogramme*.
- Extraction des pics spectraux et construction d'une constellation.
- Formation des paires de pics spectraux et hachage combinatoire.
- Alignement des empreintes en utilisant un offset.
- Amélioration des requêtes et indexation des empreintes.

## 2.4 Environnement de travail

Nous allons utiliser le langage Python pour réaliser ce projet tout en profitant de plusieurs bibliothèques afin d'échantillonner, traiter le signal et hacher le résultat voulu.

Nous utiliserons un répertoire GitHub pour le partage. Concrètement, Git est un système de contrôle de version distribué, ce qui signifie que l'ensemble de la base du code et de l'historique est disponible sur l'ordinateur de chaque développeur, ce qui permet des branchements et une fusion faciles. Le contrôle de version nous aidera (développeurs) à suivre et à gérer les modifications apportées au code du projet. Au fur et à mesure le projet prend de l'ampleur, le contrôle de version devient essentiel.

# 3 Échantillonnage : de l'Analogique au Digital

Dans ce chapitre nous allons voir dans un premier temps comment l'oreille humaine reçoit un signal analogique et le convertit en signal électrique. Puis dans un deuxième temps comment cette transformation de signal se produit sur une machine, ce que nous appelons l'échantillonnage d'un signal.

## 3.1 Fonctionnement de l'oreille humaine : transmission de signal

Nous savons qu'en réalité, le son est une vibration qui se propage comme une onde mécanique de pression et de déplacement, à travers un milieu tel que l'air ou l'eau. Lorsque cette vibration parvient à nos oreilles, en particulier au tympan, elle déplace de petits os qui transmettent la vibration à de petites cellules ciliées situées dans notre oreille interne. Enfin, les petites cellules ciliées produisent des impulsions électriques, qui sont transmises à notre cerveau par le nerf auditif.

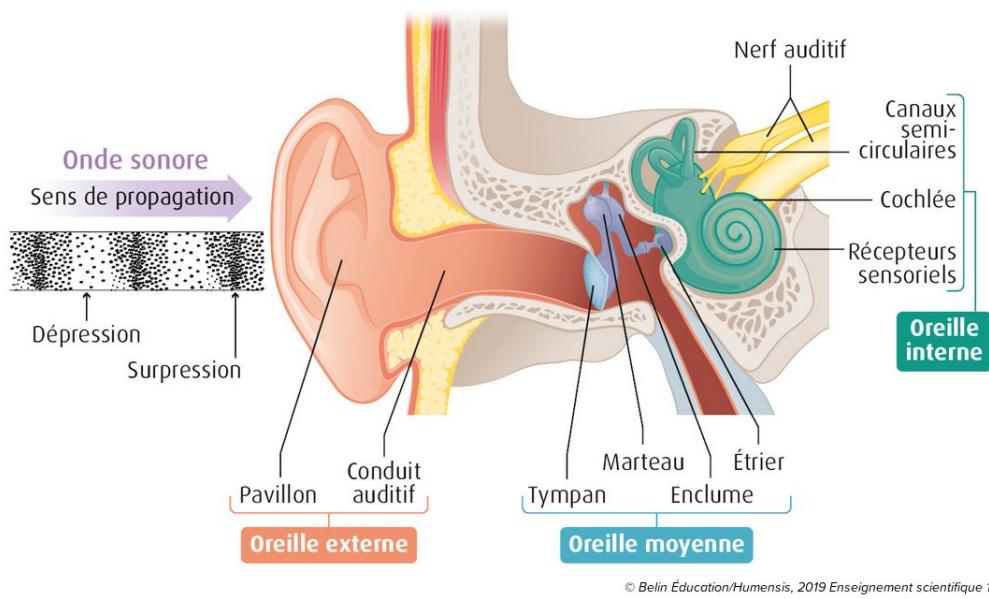


FIGURE 3.1 – Signal Processing Through Human Ear

Les appareils d'enregistrement imitent assez fidèlement ce processus, en utilisant la pression de l'onde sonore pour la convertir en un signal électrique.

## 3.2 Échantillonnage

Une onde sonore réelle dans l'air est un signal de pression continu. Dans un microphone, le premier composant électrique à rencontrer ce signal le traduit en un signal de tension analogique - là encore, continu. Ce signal continu n'est pas très utile dans le monde numérique. Avant de pouvoir être traité, il doit donc être traduit en un signal discret qui peut être stocké numériquement. Cela se fait en capturant une valeur numérique qui représente l'amplitude du signal.

La conversion implique la quantification de l'entrée, et elle introduit nécessairement une petite quantité d'erreur. Par conséquent, au lieu d'une seule conversion, un convertisseur analogique-numérique effectue de nombreuses conversions sur de très petits morceaux du signal - un processus connu sous le nom d'échantillonnage.

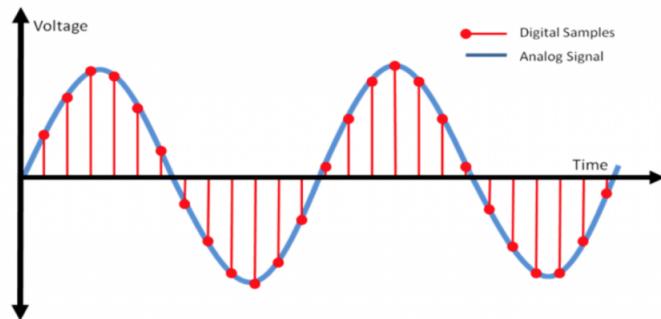


FIGURE 3.2 – Simple Signal Sampling

Le théorème de Nyquist-Shannon nous indique quelle fréquence d'échantillonnage est nécessaire pour capturer une certaine fréquence dans un signal continu. En particulier, pour capturer toutes les fréquences qu'un humain peut entendre dans un signal audio, nous devons échantillonner le signal à une fréquence deux fois supérieure à celle de la gamme d'audition humaine. L'oreille humaine peut détecter des fréquences situées approximativement entre 20 Hz et 20 000 Hz. Par conséquent, les signaux audio sont le plus souvent enregistrés à une fréquence d'échantillonnage de 44 100 Hz.

Il s'agit aussi de la fréquence d'échantillonnage des disques compacts, et c'est aussi la fréquence la plus couramment utilisée avec l'audio MPEG-1 (VCD, SVCD, MP3). Cette fréquence spécifique a été choisie à l'origine par Sony parce qu'elle pouvait être enregistrée sur un équipement vidéo modifié fonctionnant à 25 images par seconde (PAL) ou 30 images par seconde (en utilisant un magnétoscope monochrome NTSC) et couvrir la largeur de bande de 20 000 Hz jugée nécessaire pour correspondre aux équipements d'enregistrement analogiques professionnels de l'époque. Ainsi, pour notre application nous choisissons une fréquence d'échantillonnage égale à 44 100 Hz afin d'enregistrer le son.

Voici un code générique qui nous permet d'enregistrer un son à travers le microphone d'un ordinateur en utilisant la librairie *PyAudio* :

```

1 import pyaudio
2
3 FORMAT = pyaudio.paInt16 # 16 bits samples size
4 CHANNELS = 2 # Stereo
5 RATE = 44100 # Sampling rate
6 CHUNK = 1024
7 RECORD_SECONDS = 5 # Number of seconds to record
8
9 audio = pyaudio.PyAudio()
10
11 # start Recording
12 stream = audio.open(format=FORMAT, channels=CHANNELS,
13                      rate=RATE, input=True,
14                      frames_per_buffer=CHUNK)
15
16 print("recording...")
17 frames = []
18
19 for i in range(0, int(RATE / CHUNK * RECORD_SECONDS)):
20     data = stream.read(CHUNK)
21     frames.append(data)
22
23 print("finished recording")

```

# 4 Traitement du signal acoustique

Le traitement du signal fait référence à l'acquisition, au stockage, à l'affichage et à la génération de signaux, ainsi qu'à l'extraction d'informations des signaux et au ré-encodage des informations. Cela dit, le traitement du signal sous une forme ou une autre est un élément essentiel dans la pratique de tous les aspects de l'acoustique. Les algorithmes de traitement du signal permettent aux acousticiens de séparer les signaux du bruit, d'effectuer une reconnaissance automatique de la parole ou de compresser les informations pour un stockage ou une transmission plus efficaces. Les concepts de traitement du signal sont les éléments de base utilisés pour construire des modèles de la parole et de l'audition.

Dans ce chapitre nous allons étudier la nature du signal acoustique et sous quel forme il peut être représentée afin de déterminer l'approche la plus adéquate pour extraire les données nécessaires à la reconnaissance acoustique voire comparer entre les signaux.

## 4.1 Analyse de la problématique

En ce qui concerne le traitement de signal numérique, nous étudions généralement les signaux numériques dans l'un des domaines suivants : domaine temporel (signaux unidimensionnels), domaine fréquentiel, et domaine temporel-fréquentiel (spectrogramme).

D'une part, la problématique se manifeste dans le choix du domaine dans lequel nous devons étudier le signal, cela veut dire qu'il faut trouver le domaine le plus adéquat afin de traiter notre signal acoustique ainsi avoir la meilleure représentation des caractéristiques essentielles de notre signal. D'autre part, il faut définir ces caractéristiques essentielles relativement à notre étude : le temps ? les amplitudes ? les fréquences ? ... Puis trouver une méthode pour les extraire.

Afin de répondre à ces problématiques il faut prendre en considération les deux objectifs cruciales derrière ce traitement qui sont :

- **robustesse au bruit additif** : reconnaissance en milieu bruité, on considère ici le bruit additif : les autres sources s'ajoutent linéairement au signal cible. Ce bruit peut simplement s'ajouter de manière "transparente", ou faire occlusion au signal. Typiquement, un signal de parole par-dessus la musique sera considéré comme un bruit additif.
- **robustesse aux distorsions non linéaires** : reconnaissance de signaux auxquels on a appliqué des distorsions temporelles (étirement ou compression temporelle) ou fréquentielles (transposition en fréquence), une modulation de volume, un filtrage passe-bande, un ajout d'écho, une compression avec pertes, une modulation de l'égalisation fréquentielle, etc.
- **robustesse à la désynchronisation** : globalement, on se trouve face au problème de la localisation de l'extrait dans le signal complet, et localement, un décalage temporel ne doit pas gêner la reconnaissance.

## 4.2 Etat de l'art : études des solutions existantes

Nous présentons dans cette section les représentations du signal dans trois domaines différents tout en précisant les avantages et les inconvénients de chaque représentations.

### 4.2.1 Représentation temporelle

Cette représentation fournit, explicitement, des informations sur l'amplitude, et le temps de chaque point du signal numérique.

L'échantillonnage d'un signal pris dans le temps retourne une représentation du temps telle que mesurée par un changement périodique du signal ou des données. Par exemple, des données montrant la progression de l'amplitude sur une période de temps spécifique, seraient "amplitude en fonction du temps".

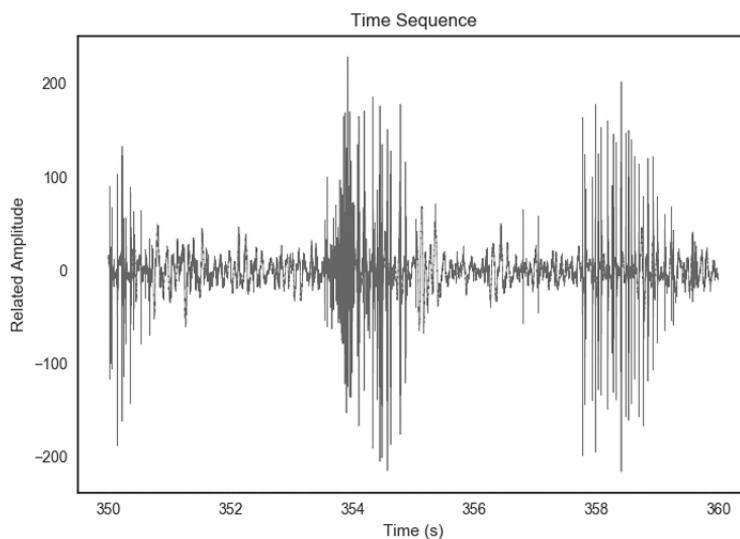


FIGURE 4.1 – Signal Representation In Time Domain

Il s'agit d'une représentation temporelle directe du signal numérique tel que chaque point correspond à un point du signal échantillonné en utilisant le microphone d'un appareil ou une lecture d'un fichier wav ou mp3.

Les points (amplitude, temps) peuvent être facilement récupérés et stocké dans une base de données. Ainsi deux signaux égaux ou correspondance entre deux musiques peut être déterminée en comparant ces points deux-à-deux. Le signal stocké dans la base de données est considéré un "match" est celui qui a le plus de points (amplitude, temps) correspondants à celui qui est échantillonné.

#### Avantages :

Représentation minimal et direct du signal puisqu'il n'y a aucun traitement profond.

#### Inconvénients :

Peu robuste au bruit et aux distorsions ; dans cette représentation de signal (amplitude, temps), nous pouvons facilement perdre de la précision dans la comparaison car le signal échantillonné sera largement différent à celui qui est déjà stocké dans la base de données dans la présence du bruit ou d'une compression avec pertes.

#### 4.2.2 Représentation fréquentielle

Ici, dans le domaine fréquentiel, nous pouvons observer l'amplitude en fonction de la fréquence. L'amplitude d'une onde ou d'une vibration est exprimée en nombres positifs, ainsi l'amplitude maximale étant une mesure de la déviation par rapport à sa valeur centrale. En ce qui concerne la fréquence, elle est exprimée en Hz (ou KHz).

La transformation vers le domaine fréquentiel a pour but de simplifier l'étude et le traitement des signaux, cette méthode est largement utilisé dans le domaine d'informatique quand on parle d'un signal numérique (le son, les images ...) ce qui représente bien notre cas dans ce projet, mais elle est aussi utile dans d'autres domaines, par exemple les systèmes mathématiques gouvernés par des équations différentielles linéaires, une classe très importante de systèmes ayant de nombreuses applications dans le monde réel, la conversion de la description du système du domaine temporel au domaine fréquentiel permet de convertir les équations différentielles en équations algébriques, qui sont beaucoup plus faciles à résoudre.

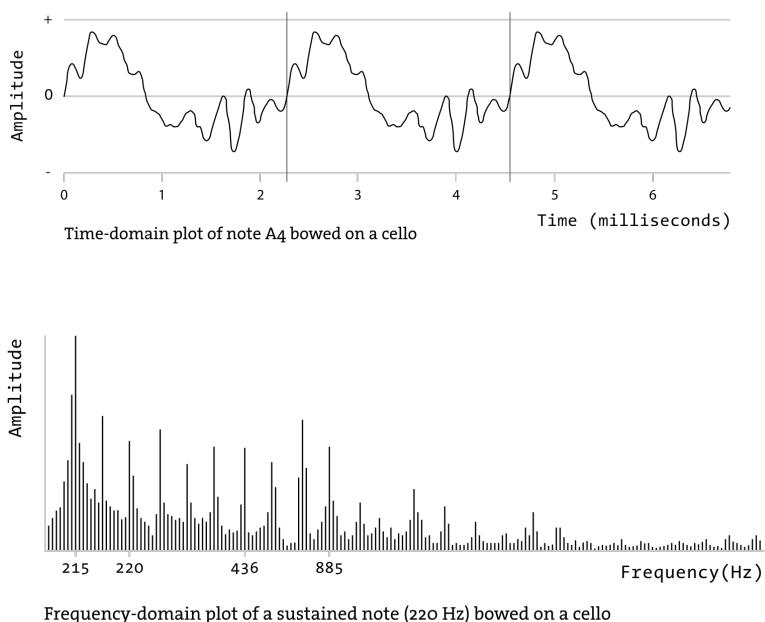


FIGURE 4.2 – Signal Representation In Frequency Domain

Une fonction ou un signal donné peut être converti entre le domaine temporel et le domaine fréquentiel à l'aide d'une paire d'opérateurs mathématiques appelés transformées. Un exemple est la transformée de Fourier, qui convertit une fonction temporelle en une somme ou une intégrale d'ondes sinusoïdales de différentes fréquences, chacune d'entre elles représentant une composante de fréquence. Le "spectre" des composantes de fréquence est la représentation du signal dans le domaine des fréquences. La transformée de Fourier inverse reconvertis la fonction dans le domaine fréquentiel en fonction dans le domaine temporel.

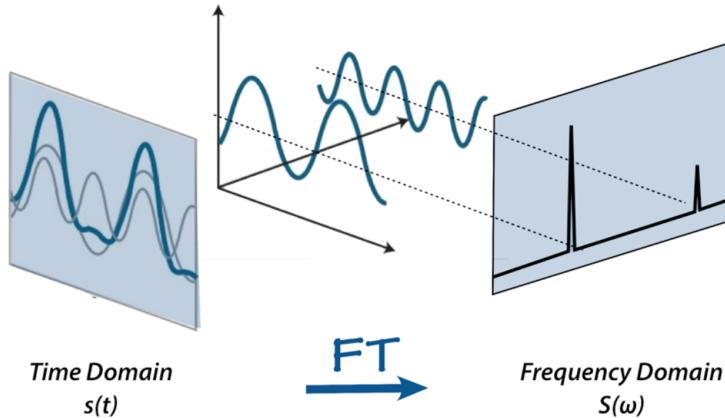


FIGURE 4.3 – Fourier Transform Overview

Dans le cas d'un signal numérique (ou fonction discrète)  $x(t)$  la transformée de Fourier discrète sur N points est définie par :

$$TFD_N(x(k)) = X(n) = \sum_{k=0}^{N-1} x(k) \exp(-j \frac{2\pi n k}{N}) \quad (4.1)$$

Ainsi la transformée de Fourier discrète inverse sur N points est définie par :

$$TFD_N^{-1}(X(n)) = x(k) = \frac{1}{N} \sum_{n=0}^{N-1} X(n) \exp(j \frac{2\pi n k}{N}) \quad (4.2)$$

Les points (amplitude, fréquence) retournés par la transformée de Fourier peuvent être facilement récupérés et stockés dans une base de données. Ainsi deux signaux égaux ou correspondants entre deux musiques peuvent être déterminés en comparant ces points deux-à-deux. Le signal stocké dans la base de données est considéré un "match" est celui qui a le plus de points (amplitude, fréquence) correspondants à celui qui est échantillonné.

#### **Avantages :**

Représentation robuste au bruit car il est devenu possible de choisir un intervalle de fréquence ou même les amplitudes importantes auxquelles on applique une projection sur la base de données afin de trouver une ressemblance entre le signal enregistré via le microphone et les signaux stockés.

#### **Inconvénients :**

Peu robuste à une désynchronisation et à un étirement temporel, car elle ne représente aucune évolution temporelle.

### **4.2.3 Représentation temporelle-fréquentielle**

Un signal dans le domaine temporelle-fréquentiel est appelé *Spectrogramme* et il montre les fréquences du signal et à quels moments (temps) elles sont présentes.

Les spectrogrammes sont largement utilisés dans les domaines de la musique, de la linguistique, des sonars, des radars, du traitement de la parole, de la sismologie, etc. Les spectrogrammes audio peuvent être utilisés pour identifier phonétiquement les mots parlés et pour analyser les différents cris des animaux.

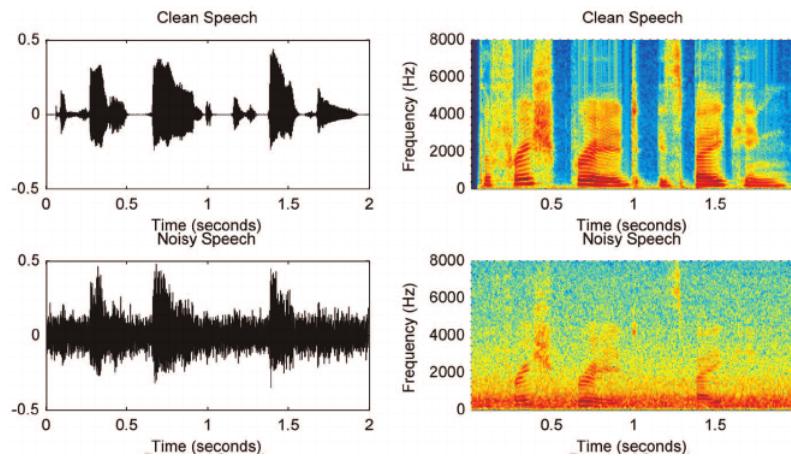


FIGURE 4.4 – Spectrogramme Of Pure and Distorted Signal

Un format courant du Spectrogramme est un graphique à deux dimensions géométriques : un axe représente le temps, et l'autre axe la fréquence ; une troisième dimension indiquant l'amplitude d'une fréquence particulière à un moment donné est représentée par l'intensité ou la couleur de chaque point de l'image.

Mathématiquement pour obtenir le Spectrogramme d'un signal numérique on utilise la transformée de Fourier à court terme (STFT ou *Short-Time Fourier Transform*), il s'agit d'une séquence de transformées de Fourier du signal fenêtré. La STFT fournit les informations de fréquence localisées dans le temps pour les situations dans lesquelles les composantes de fréquence d'un signal varient dans le temps, alors que la transformée de Fourier standard fournit les informations de fréquence moyennées sur tout l'intervalle de temps du signal.

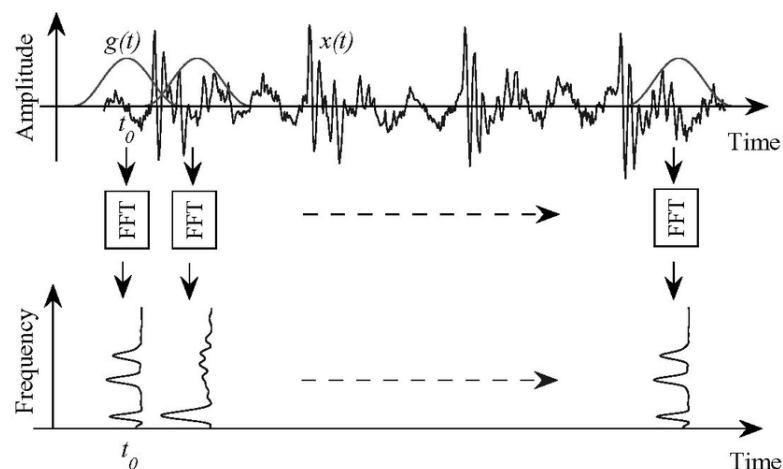


FIGURE 4.5 – STFT Operation

Ainsi la STFT discrète sur N points est définie par :

$$STFT_N(x(k)) = X(m, n) = \sum_{k=0}^{N-1} x(k)w(k-m) \exp(-j\frac{2\pi nk}{N}) \quad (4.3)$$

tel que :

- $x(k)$  est le signal d'entrée ou le signal échantillonné.
- $w(k)$  représente une fonction fenêtre de taille  $k$  (e.g. *Hanning*).

L'amplitude au carré de la STFT donne la représentation spectrogramme :

$$spectrogram(x(k)) = |STFT_N(x(k))|^2 = |X(m, n)|^2 \quad (4.4)$$

Finalement cette représentation, *temporelle-fréquentielle*, est beaucoup plus fiable par rapport aux deux représentations précédentes car elle permet d'avoir une meilleure entropie en ce qui concerne le signal échantillonné. Les points (temps, fréquence) peuvent être facilement récupérés et stockés dans une base de données. Ainsi deux signaux égaux ou correspondants entre deux musiques peuvent être déterminés en comparant ces points deux-à-deux. Le signal stocké dans la base de données est considéré un "match" est celui qui a le plus de points (temps, fréquence) correspondants à celui qui est échantillonné.

**Avantages :**

Représentation robuste au bruit, aux distorsions, et au désynchronisation car on peut filtrer les points à très grande/basse amplitude voire appliquer d'autres filtres afin d'éliminer le bruit, sans oublier que chaque point est temporellement relatif aux autres, ce qui nous aide à déterminer l'emplacement temporelle de notre échantillon à l'intérieur du signal d'origine déjà stocké dans la base de données.

**Inconvénients :**

Bien que cette représentation retourne suffisamment d'informations pour comparer avec précision deux extraits de son (deux signaux acoustiques), cette quantité d'informations pose un problème de mémoire puisqu'on doit stocker chaque point (temps, fréquence) du spectrogramme dans notre base de données, sans oublier la vitesse de comparaison entre deux signaux qui prendra beaucoup de temps si on procède par une comparaison de ces points deux-à-deux.

### 4.3 Solution proposée et sa mise en œuvre

Nous avons établi précédemment que le Spectrogramme est la représentation de signal qui retourne les informations nécessaires afin de résoudre les problèmes éventuels de bruits, distorsions, et désynchronisation. Donc nous avons choisi cette représentation tout en ajoutant un traitement essentiel que nous appelons *Extraction des pics spectraux et constellation* afin de résoudre la problématique liée à la mémoire et à la vitesse de recherche d'un *match*.

Cette méthode qui, au lieu de traiter le spectrogramme de manière linéaire temporellement, elle se base sur une constellation de pics spectraux associés par paires. Ainsi un signal est représenté par un ensemble de paires indépendantes du point de vue temporel.

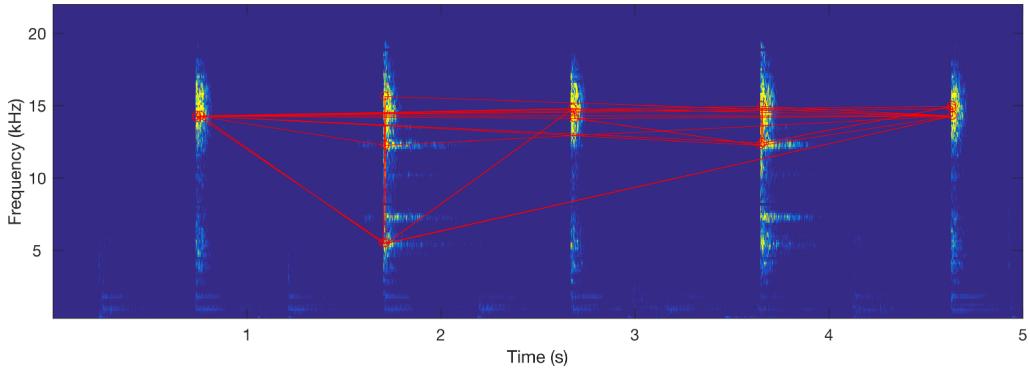
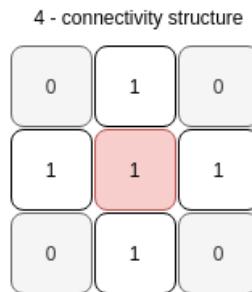


FIGURE 4.6 – Example Of Constellation Map

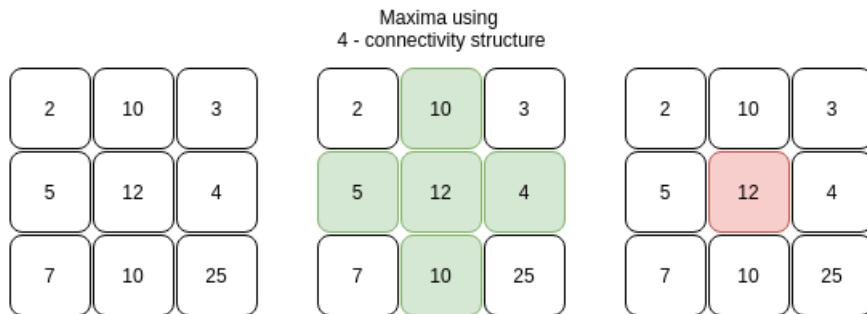
#### 4.3.1 Extraction des pics spectraux

Un pic spectral est une paire de (*temps, fréquence*) qui correspond à une amplitude localement supérieure à ses voisins (de plus forte énergie), ainsi les autres paires (*temps, fréquence*) autour de ce pic spectral ont une amplitude inférieure et donc moins résistible aux bruits.

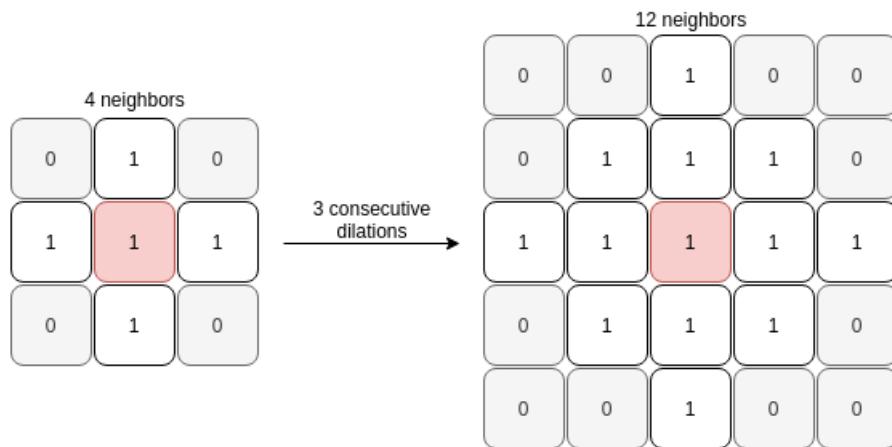
Premièrement on choisit une région de voisins en utilisant une structure binaire ou *kernel*, dans notre application nous avons choisi la structure à 4 voisins :



Ceci implique qu'un pixel doit être connecté à 4 voisins de la même manière que notre structure, et il sera considéré comme un maxima local dans cette région si et seulement si sa valeur est supérieure à ses 4 voisins. Le choix de cette structure est la meilleure stratégiquement car il s'agit d'un nombre de voisins minimal ce qui implique moins de comparaisons pendant ce traitement :



Le nombre de voisins dans une régions est paramétrable tel que si on souhaite augmenter ce nombre on applique une ou plusieurs dilatations morphologique à notre structure par elle-même ce qui étend la région de voisins, or, il faut prendre en considération que si ce nombre augmente cela implique qu'il y aura moins de maxima locaux (ou pics spectraux) à la fin de ce traitement :



Donc après avoir calculé le spectrogramme, nous le traitons comme une image ainsi on applique un filtre pour trouver les maxima locaux puis un deuxième filtre passe-haut pour garder seulement les amplitudes qui dépassent un seuil donné en paramètre, ce qui permet d'éliminer les maxima locaux qui varient lentement. Seule la position *temps-fréquence* du pic est conservée, sans que l'amplitude soit prise en compte, afin d'obtenir directement une représentation binaire.

Voici un exemple de cette opération réalisé sur une tableau à deux dimensions où chaque case représente une amplitude :

4 - connectivity filter for local maxima

2	12	3	5	0
10	15	4	18	13
7	11	25	14	9
4	2	8	19	16
6	0	10	12	8

2	12	3	5	0
10	15	4	18	13
7	11	25	14	9
4	2	8	19	16
6	0	10	12	8

2	12	3	5	0
10	15	4	18	13
7	11	25	14	9
4	2	8	19	16
6	0	10	12	8

2	12	3	5	0
10	15	4	18	13
7	11	25	14	9
4	2	8	19	16
6	0	10	12	8

High Pass filter ( amp > 16 )

2	12	3	5	0
10	15	4	18	13
7	11	25	14	9
4	2	8	19	16
6	0	10	12	8

Cette représentation a l'avantage d'être très robuste au bruit additif comme aux distorsions. Sans oublier une consommation minimale de la mémoire puisqu'on prend en considération que les amplitudes importantes dans notre spectrogramme.

Nous allons voir dans la partie "*Création d'une empreinte acoustique*" comment nous traitons ces pics spectraux pour créer des paires indépendantes d'un point de vue temporelle, ce qui nous aidera à établir une empreinte digitale à notre signal de base qui est reproductible et éventuellement comparer deux empreinte afin de trouver rapidement un *match*.

#### 4.3.2 Procédure algorithmique complète

En guise de conclusion, le traitement du signal numérique capturé par un microphone ou directement par un fichier est une opération primordiale qui nous permettra par la suite de générer une empreinte

digitale robuste au bruits, distorsions, et désynchronisation. Nous résumons ce traitement comme suit :

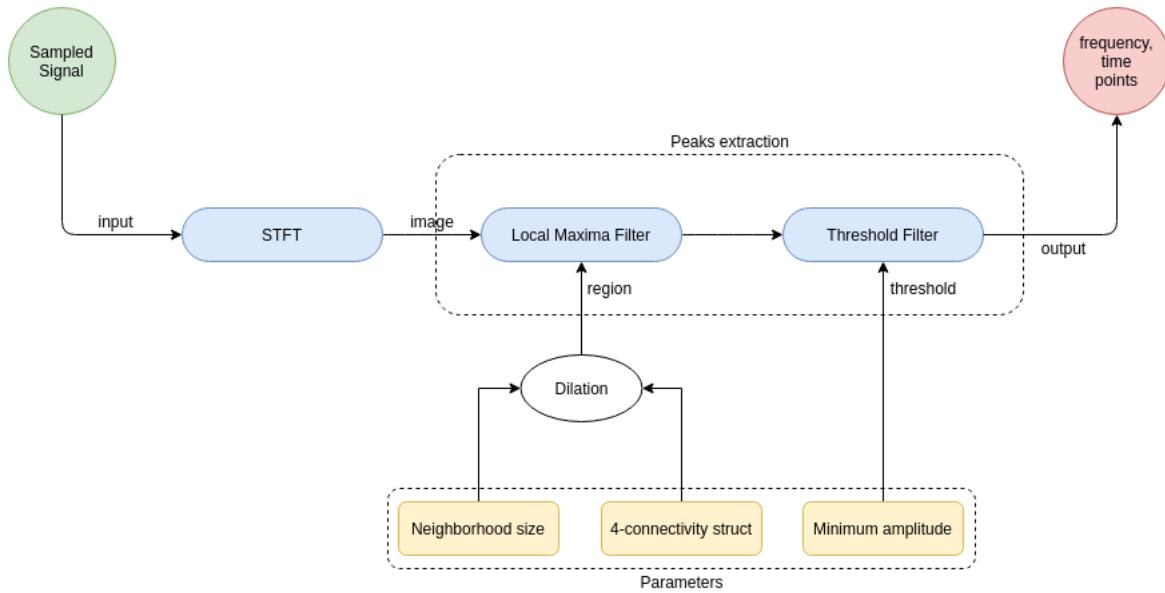


FIGURE 4.7 – Signal Processing Pipeline

#### 4.3.3 Mise en œuvre

Le traitement de notre signal numérique dépend de plusieurs variables/paramètres qui contrôlent non seulement la transformée de Fourier à court terme (STFT) mais aussi la taille de la région de voisins et le seuil minimal d'amplitudes à considérer pendant l'extraction des pics spectraux.

Voici le code responsable à la création d'un *Spectrogramme* à partir du signal échantillonné (*channel samples*) :

```

1 import numpy as np
2 import matplotlib.mlab as mlab
3
4 # Sampling rate, related to the Nyquist conditions, which affects
5 # the range frequencies we can detect.
6 DEFAULT_FS = 44100
7
8 # Size of the FFT window,
9 # The number of data points used in each block for the FFT.
10 # affects frequency granularity
11 DEFAULT_WINDOW_SIZE = 4096
12
13 # Ratio by which each sequential window overlaps the last and the
14 # next window. Higher overlap will allow a higher granularity of offset
15 # matching, but potentially more fingerprints.
16 DEFAULT_OVERLAP_RATIO = 0.5
17
18
19 def get_spectrogram(channel_samples, Fs=DEFAULT_FS, wsize=DEFAULT_WINDOW_SIZE, wratio
=DEFAULT_OVERLAP_RATIO):
20
21     # SFFT the signal and extract frequency components
22     spectrogram = mlab.specgram(
23         channel_samples,
24         NFFT=wsize,
25         Fs=Fs,
26         window=mlab.window_hanning,
27         noverlap=int(wsize * wratio))[0]
28
  
```

```

29     # Apply log transform since specgram function returns linear array. 0s are
30     # excluded to avoid np warning
31     spectrogram = 10 * np.log10(spectrogram, out=np.zeros_like(spectrogram),
32                               where=(spectrogram != 0))
33     return spectrogram

```

Voici le code responsable à l'*Extraction des Pics Spectraux* en utilisant un traitement d'image sur le *Spectrogramme* du signal échantillonné :

```

1 import numpy as np
2 from scipy.ndimage.filters import maximum_filter
3 from scipy.ndimage.morphology import (generate_binary_structure, iterate_structure,
4                                       binary_erosion)
5
6 # Minimum amplitude in spectrogram in order to be considered a peak.
7 # This can be raised to reduce number of fingerprints, but can negatively
8 # affect accuracy.
9 DEFAULT_AMP_MIN = 10
10
11 # Number of cells around an amplitude peak in the spectrogram in order
12 # to consider it a spectral peak. Higher values mean less
13 # fingerprints and faster matching, but can potentially affect accuracy.
14 PEAK_NEIGHBORHOOD_SIZE = 20
15
16 def get_peaks(spectrogram, amp_min=DEFAULT_AMP_MIN):
17
18     # define an 4-connected kernel
19     struct = generate_binary_structure(2, 1)
20
21     # And then we apply dilation using the following function
22     # in order to extend neighborhood size
23     neighborhood = iterate_structure(struct, PEAK_NEIGHBORHOOD_SIZE)
24
25     # find local maxima using our filter mask
26     local_max = maximum_filter(spectrogram, footprint=neighborhood) == spectrogram
27
28     # local_max is a mask that contains the peaks we are
29     # looking for, but also the background.
30     # In order to isolate the peaks we must remove the background from the mask.
31
32     # Applying erosion to create the mask of the background
33     background = (spectrogram == 0)
34
35     # a little technicality: we must erode the background in order to
36     # successfully subtract it form local_max, otherwise a line will
37     # appear along the background border (artifact of the local maximum filter)
38     eroded_background = binary_erosion(background, structure=neighborhood,
39                                         border_value=1)
40
41     # we obtain the final mask, containing only peaks,
42     # by removing the background from the local_max mask (xor operation)
43     # Boolean mask of spectrogram with True at peaks (applying XOR on both matrices).
44     detected_peaks = local_max != eroded_background
45
46     # extract peaks
47     amps = spectrogram[detected_peaks]
48     freqs, times = np.where(detected_peaks)
49
50     # filter peaks
51     amps = amps.flatten()
52
53     # get indices for frequency and time with Threshold
54     filter_idxs = np.where(amps > amp_min)

```

```

54     freqs_filter = freqs[filter_idxs]
55     times_filter = times[filter_idxs]
56
57     peaks = list(zip(freqs_filter, times_filter))
58     return peaks, freqs_filter, times_filter

```

## 4.4 Tests et certifications de la solution

Dans cette partie nous allons effectuer des tests sur un signal échantillonné à travers un fichier mp3 qui s'agit d'une musique intitulée "*Know yourself - Drake*". Nous allons aussi prendre en considération, que pour ce test, qu'un seul canal pour démontrer le fonctionnement du traitement du signal, par contre notre application traite les deux canaux séquentiellement.

Il est nécessaire de prendre en considération tous les paramètres de notre application que nous avons définie précédemment dans le code, et qui influencent directement sur la qualité du résultat final ainsi sur le temps d'exécution.

Pour notre application nous choisissons les valeurs suivantes pour nos paramètres :

- Fréquence d'échantillonnage (FS) = 44100
- Taille de la fenêtre utilisée dans la fonction STFT (WINDOW SIZE) = 4096
- Ratio du chevauchement des fenêtres consécutive (OVERLAP RATIO) = 0.5 (la moitié)
- Seuil d'amplitude minimale pour être considérée comme un pic spectral (AMP MIN) = 10
- Taille de la région des voisins utilisée dans la recherche des maxima locaux (NEIGHBORHOOD SIZE) = 20

La musique dure 277 secondes (4 minutes 37 secondes), ainsi avec une fréquence d'échantillonnage égale à 44100 nous enregistrons 44100 points d'entrée pour chaque secondes, ce qui implique que notre signal échantillonné est de taille :

$$DigitalSignalSize = duration \times FS = 277 \times 44100 = 12215700 \quad (4.5)$$

Comme expliqué dans le code, chacun de ces paramètre influence directement sur le nombres des pics spectraux retourner à la fin du traitement voire aussi sur le temps de ce traitement.

Nous avons expérimenté avec différentes valeurs ainsi nous avons choisi celles-ci par comparaison de résultats.

Il ne s'agit pas des valeurs optimales, il se peut qu'il existe une meilleure approche, or il est difficile de mettre en place un protocole assez profond afin de trouver des valeurs optimales car cela prendra énormément de temps pour le réaliser.

On peut citer comme protocole la mise en place d'un réseaux neurones qui prends en entrée tous les paramètres de notre application du traitement du signal et l'entraîner sur une large base de tests qu'on peut effectuer, ainsi ce réseaux pourra déterminer des valeurs qui se rapprochent à l'optimal pour un résultat souhaité.

#### 4.4.1 Tests préliminaires

Les tests suivants ne comportent aucun bruit ajouté sur le signal d'origine.

Voici le résultat retourner par la fonction `get_spectrogram` :

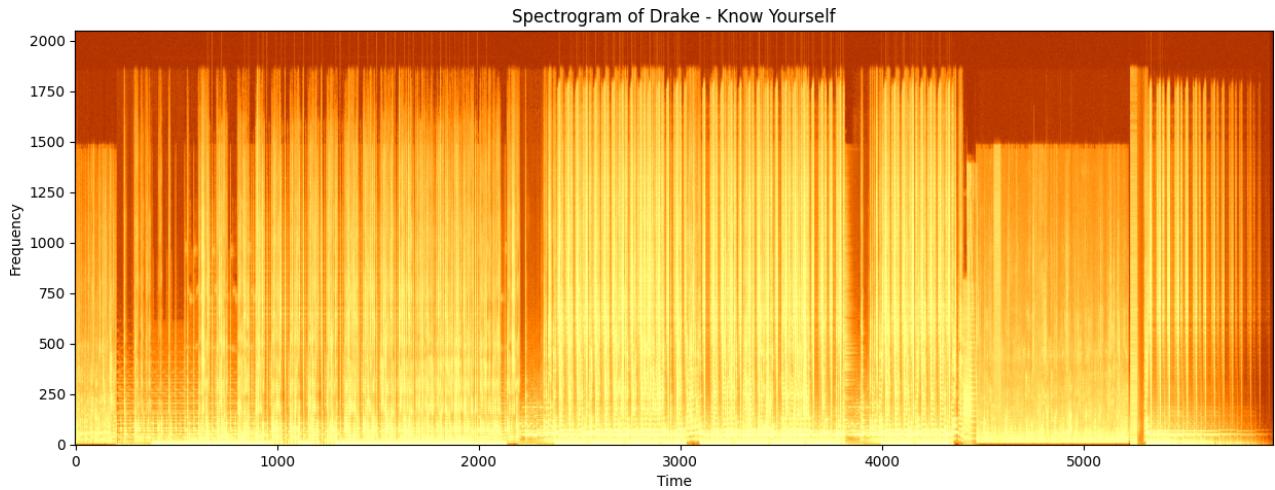


FIGURE 4.8 – Spectrogramme Of Drake - Know Yourself With No Noise

Maintenant nous allons extraire les pics spectraux ainsi construire une carte de constellation en utilisant la fonction `get_peaks` :

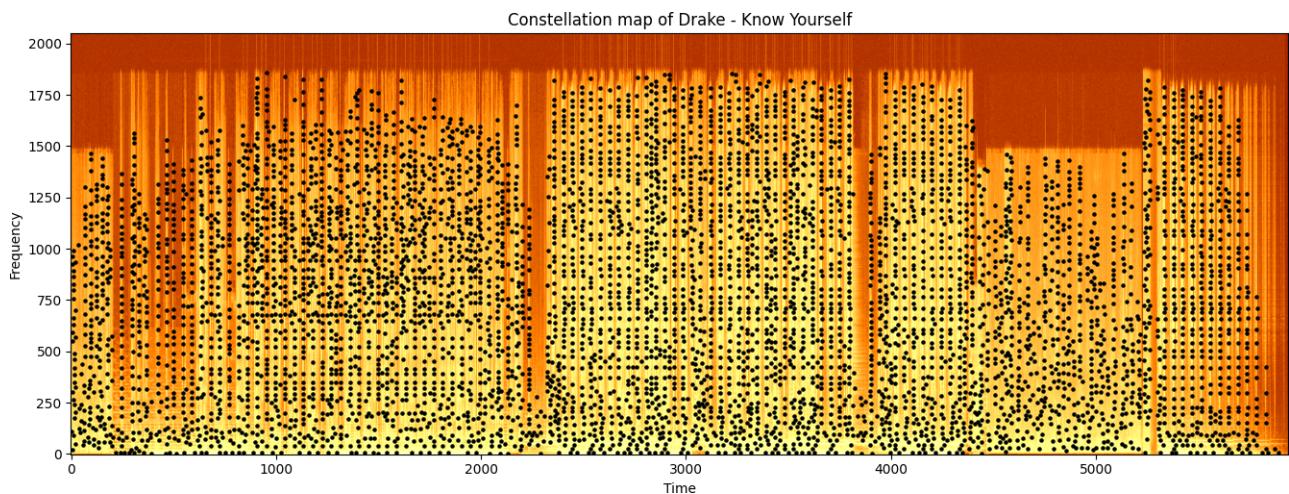


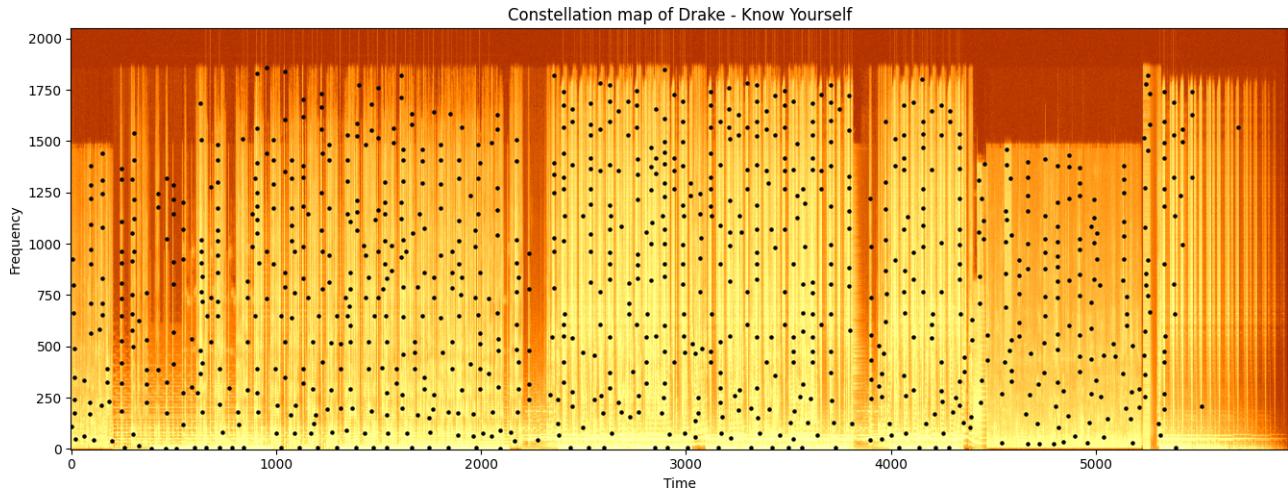
FIGURE 4.9 – Constellation Map Of Drake - Know Yourself With No Noise

Pour les deux opérations nous avons :

operation	execution time (sec)	output size
spectrogram	0.91859579	12169011
peaks extraction	11.18466210	6128

On remarque que la création d'un *Spectrogramme* prend un temps négligeable par rapport à l'extraction des pics spectraux (avec la taille de région de voisin égale à 20). Or, la réduction de la taille finale est considérablement bénéfique car nous avons économisé 99.95% de mémoire en choisissant les amplitudes importantes dans notre signal numérique.

Si on augmente le nombre de voisins à prendre en considération lors de l'identification des maxima locaux alors le nombre des pics spectraux diminuera considérablement, voici un exemple pour une valeur *NEIGHBORHOOD\_SIZE = 50* :



Pour les deux opérations nous avons :

operation	execution time (sec)	output size
spectrogram	0.90590810	12169011
peaks extraction	69.41175055	987

Nous économisons encore plus de mémoire, or, le temps de traitement augmente drastiquement. Sans oublier que le nombre final des pics spectraux influence sur la précision de l'identification d'un *match*, ainsi elle sera moins précis si on réduit beaucoup le nombre des pics spectraux.

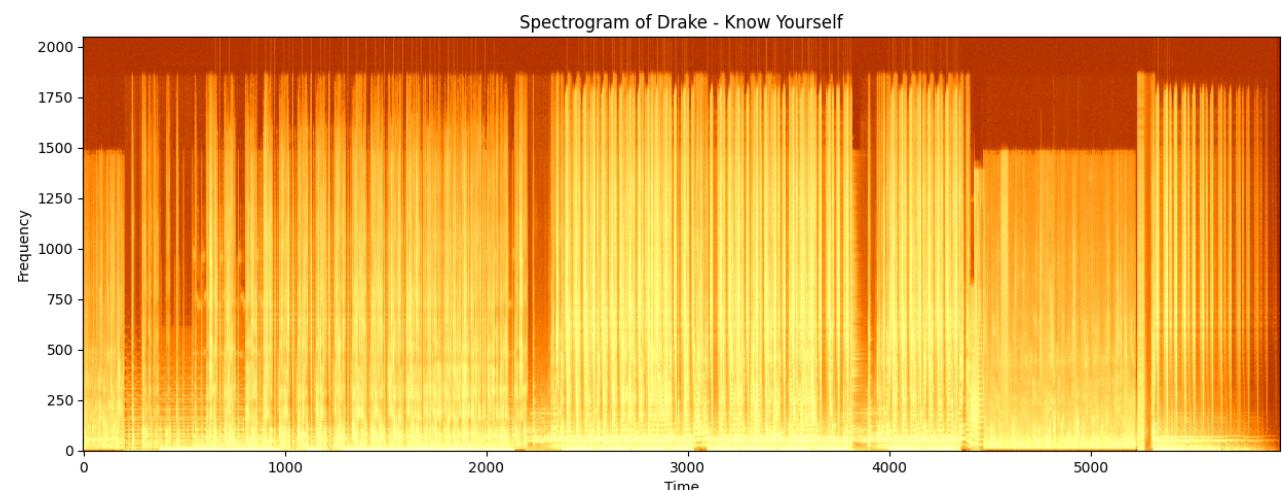
#### 4.4.2 Tests de robustesse de la constellation

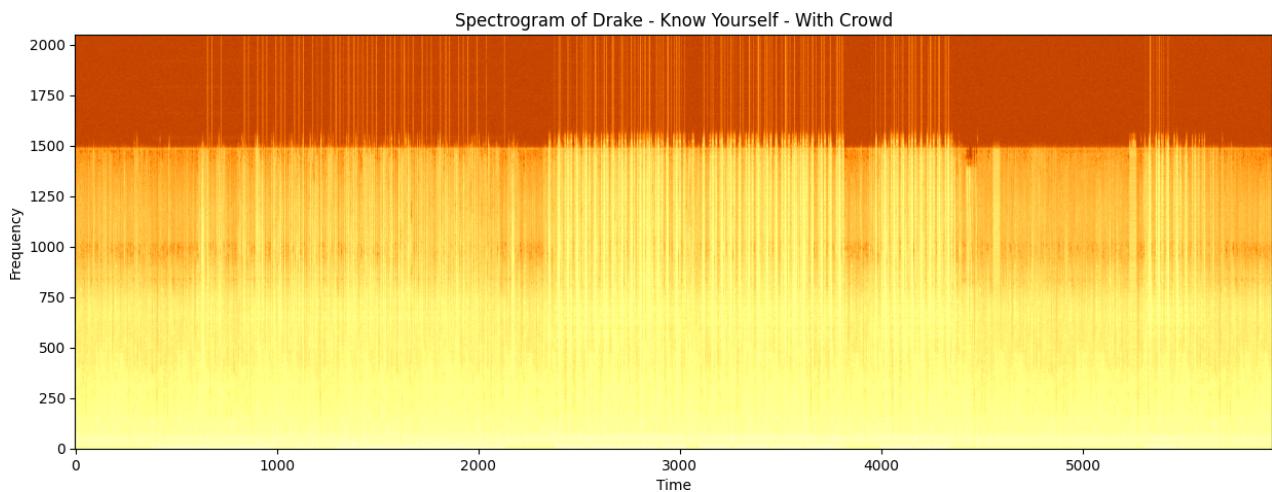
Dans cette partie nous allons effectuer une série de tests sur la même musique tout en rajoutant différents types de distorsions. Nous utilisons les mêmes paramètres de l'application cités précédemment.

##### Bruit réel : groupe de gens dans un bar

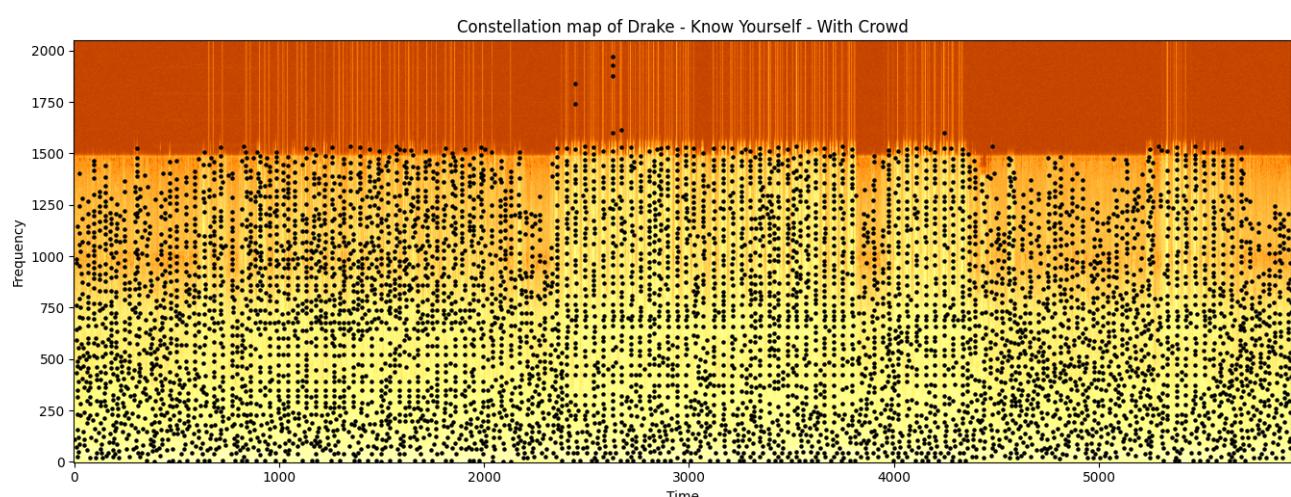
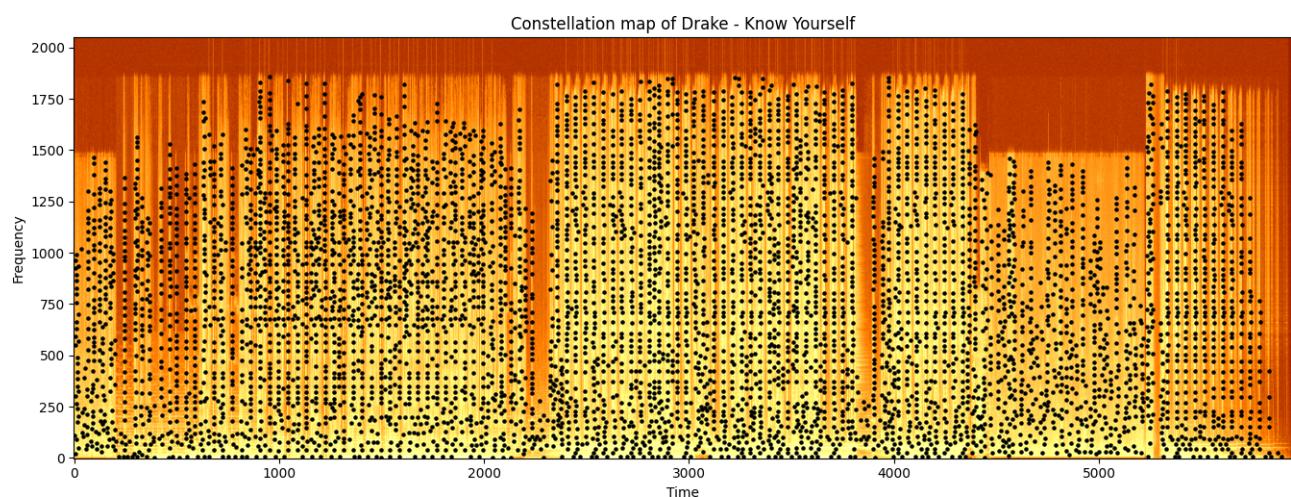
Nous avons rajouté un bruit au-dessus notre musique originale qui s'agit d'un groupe de gens qui parlent dans un bar tout en contrôlant l'intensité (loudness) de ce bruit. Cette intensité est exprimé en *dBFS* (Decibels relative to Full Scale), il s'agit d'une unité de niveau de signal audio. Elle indique le rapport entre le niveau de ce signal et le niveau maximal qui est 0 dBFS.

Comme un premier test nous allons rajouter le bruit en question avec une intensité de -17.23 dBFS, ainsi peut directement visualiser les changements du signal qui résulte de l'ajout de notre bruit réel sur le *Spectrogramme* :

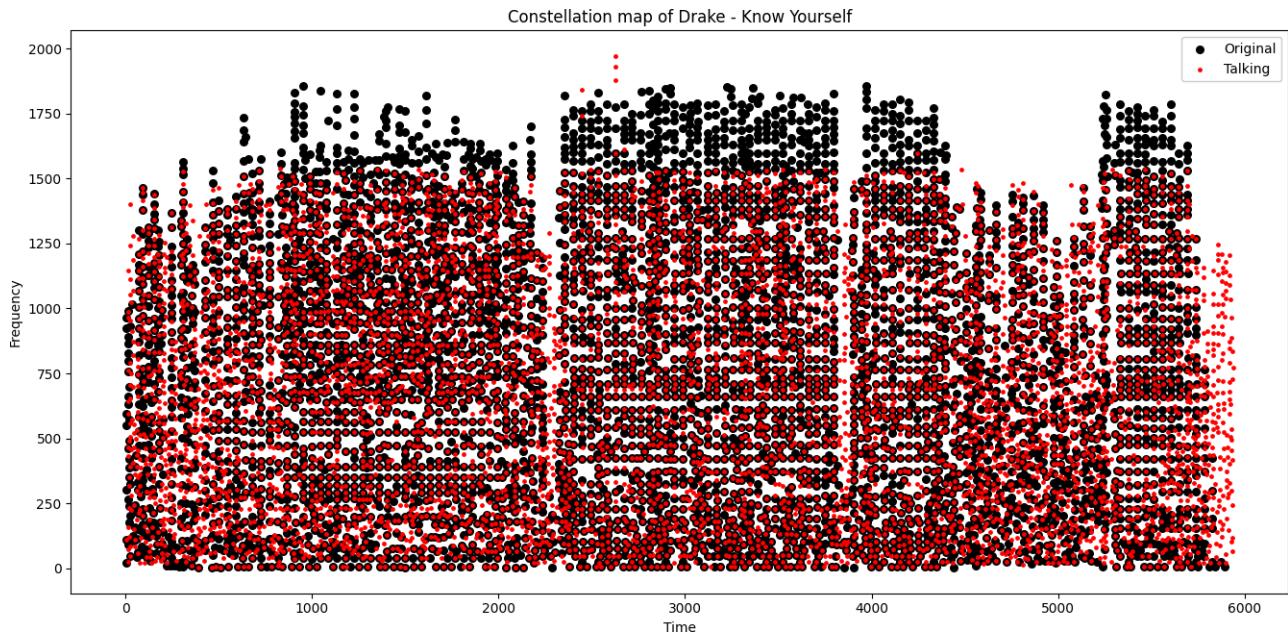




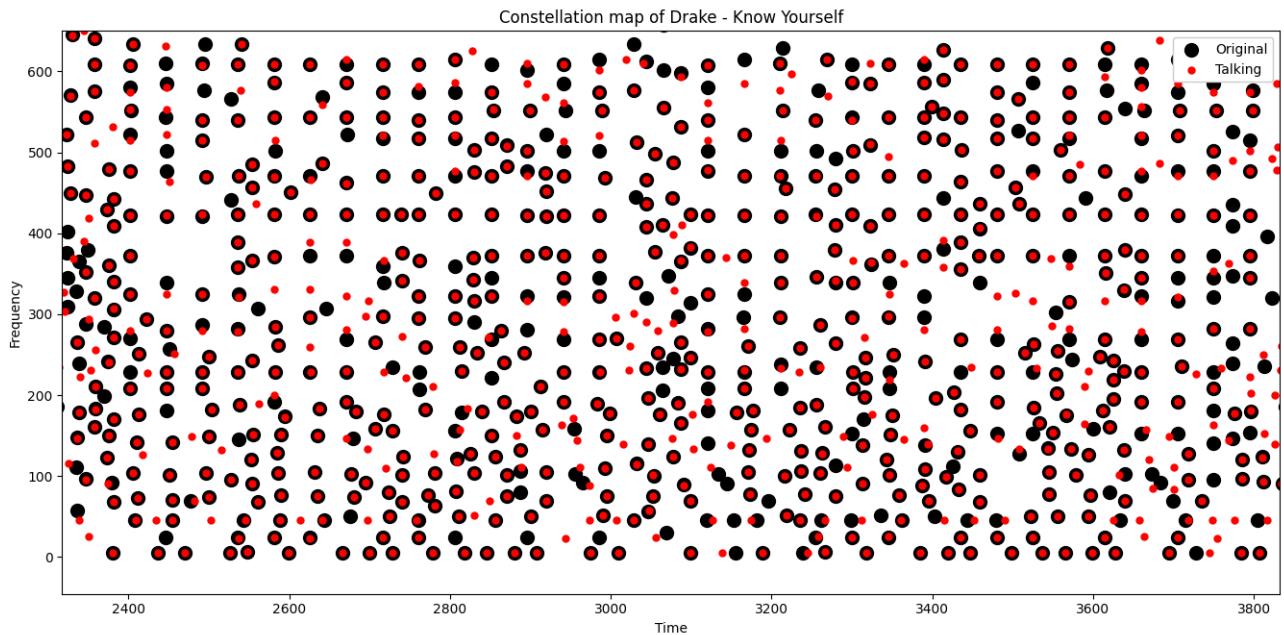
Comme on peut aussi voir les changements après l'extraction des pics spectraux, nous allons voir que le nombre de pics augmente tout en rajoutant plus d'intensité à notre bruit :



Si on superpose les deux constellations précédente nous allons voir les points différents et aussi les points similaires entre le signal original et le signal bruité :



Focalisons nous sur une petite région afin de mieux voir la différence et la similitude :



On remarque bien que l'union des deux constellation n'est pas disjointe et qu'il existe des points en commun entre les deux signaux. Ceci implique que notre approche au traitement du signal numérique est bien robuste au bruit additif.

Voici un tableau qui montre plus de détails sur cette similitude tout en augmentant l'intensité du bruit :

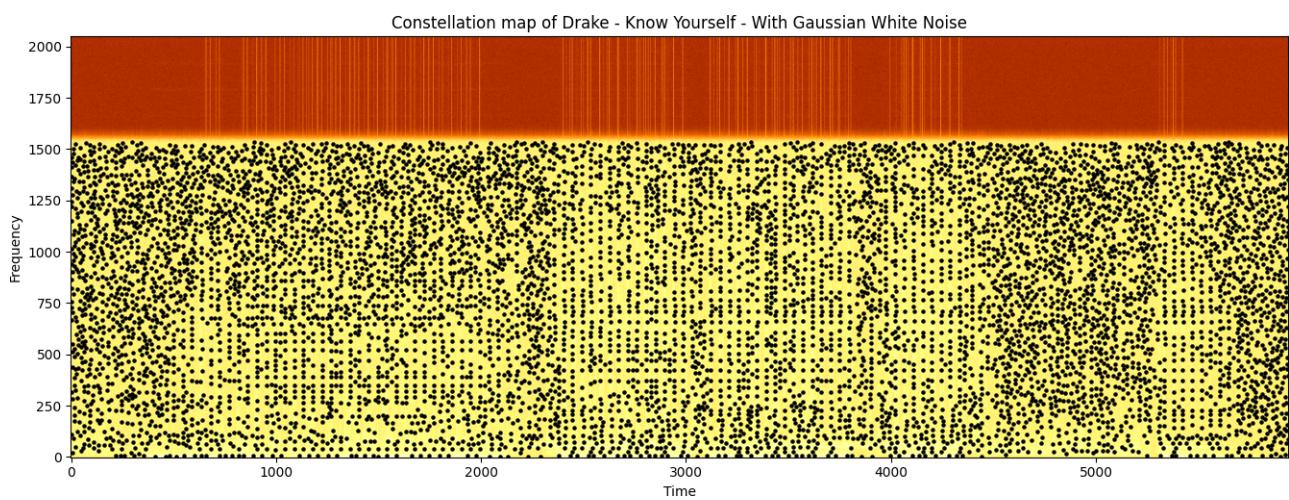
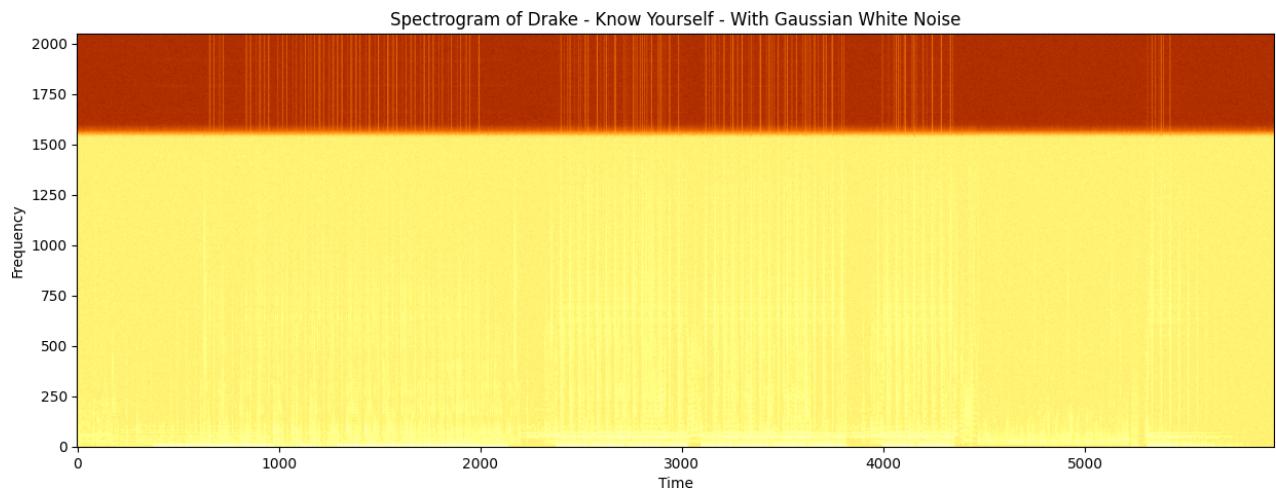
source (mp3)	loudness (dBFS)	output size	similitude with original (%)
original	-12.99	6128	100
crowd noise	-37.23	5840	58.44
crowd noise	-27.27	6088	51.23
crowd noise	-17.23	6593	38.12
crowd noise	-7.50	7806	16.44

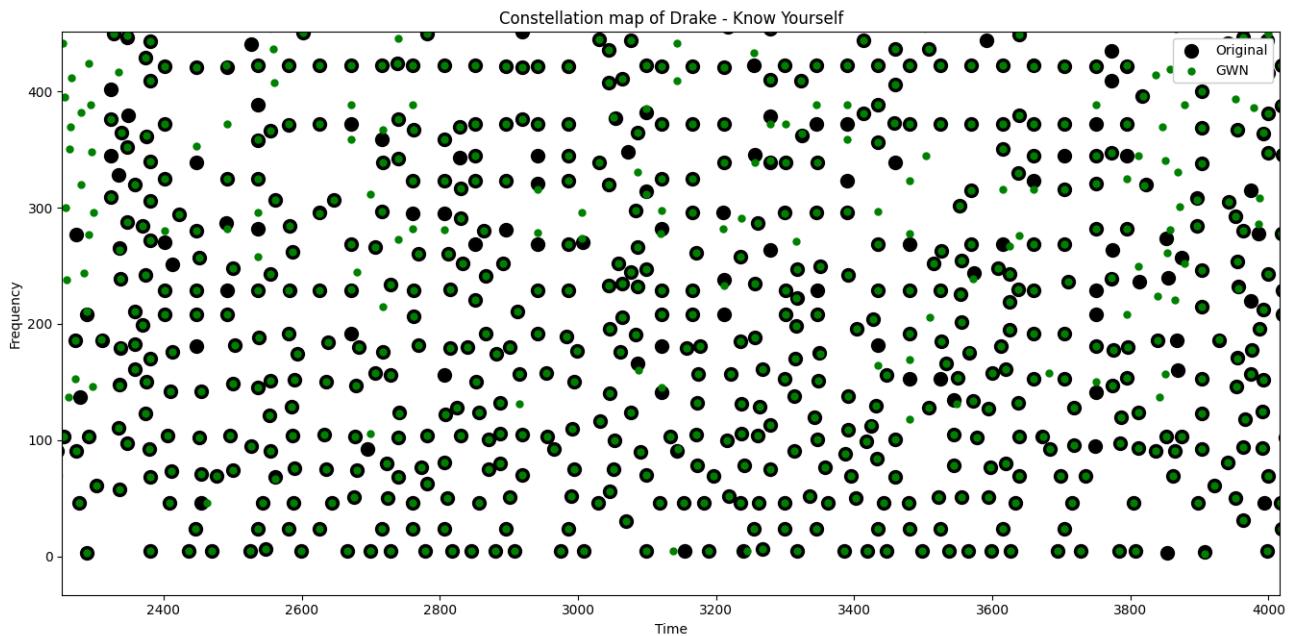
## Bruit additif blanc Gaussien

Dans ce test nous allons rajouter du bruit blanc Gaussien à notre signal original. Le bruit additif blanc Gaussien est un modèle élémentaire de bruit utilisé en théorie de l'information pour imiter de nombreux processus aléatoires qui se produisent dans la nature. Les adjectifs indiquent qu'il est :

- **additif** : il s'ajoute au bruit intrinsèque du système d'information ;
- **blanc** : sa puissance est uniforme sur toute la largeur de bande de fréquences du système, par opposition avec un bruit coloré qui privilégie une bande de fréquences par analogie avec une lumière colorée dans le spectre visible ;
- **gaussien** : il a une distribution normale dans le domaine temporel avec une moyenne nulle.

Comme un premier test nous allons rajouté le bruit en question avec une intensité de -24.76 dBFS, ainsi peut directement visualiser les changements du signal qui résulte de l'ajout de notre bruit gaussien sur le *Spectrogramme* et sur la *Constellation*, sans oublier les points communs entre les deux signaux :





Voici un tableau qui montre plus de détails sur cette similitude tout en augmentant l'intensité du bruit gaussien :

source (mp3)	loudness (dBFS)	output size	similitude with original (%)
original	-12.99	6128	100
gaussian white noise	-34.77	7162	47.88
gaussian white noise	-24.76	8200	32.31
gaussian white noise	-14.77	9681	15.93

On peut observer ce qui suit :

- Les cartes de constellations des versions déformées partagent plusieurs pics avec la version originale.
- Les pics des versions déformées sont parfois légèrement décalés en temps et/ou en fréquence par rapport à ceux de la version originale.

Le nombre de points de correspondance sera significatif en présence de pics parasites injectés en raison du bruit, car les positions des pics sont relativement indépendantes ; en outre, le nombre de correspondances peut également être significatif même si de nombreux points corrects ont été supprimés. L'enregistrement des cartes de constellation est donc un moyen puissant d'établir des correspondances en présence de bruit et/ou de suppression de caractéristiques. Cette procédure réduit le problème de recherche à une sorte d'*astronavigation*, dans laquelle une petite parcelle de points de constellation de fréquence-temps doit être rapidement localisée dans un grand univers de constellations.

Nous allons voir par la suite comment ces pics spectraux peuvent être utilisés afin de créer une empreinte digitale au signal d'entrée et à quel point cela influence sur la vitesse/robustesse de la recherche d'une correspondance.

# 5 Création d'une empreinte acoustique

Après avoir converti le signal numérique à un *Spectrogramme* puis le traiter pour extraire les pics spectraux, nous allons voir comment utiliser ces pics afin de produire une empreinte digitale de ce signal.

Une empreinte audio est un résumé numérique qui peut être utilisé pour identifier un échantillon audio. Cette empreinte permet de localiser rapidement des éléments similaires dans une base de données audio.

Dans ce chapitre nous allons présenter les problématiques liées à l'utilisation directe des pics spectraux et pourquoi nous avons besoin de les représenter différemment, ainsi le processus de création d'une empreinte qui est basé principalement sur la construction des paires de pics.

## 5.1 Analyse de la problématique

A ce niveau nous avons les pics de notre signal audio, ces pics peuvent être des milliers pour une seule musique. Ils caractérisent très bien notre signal, or comment ils peuvent être utilisés pour créer une empreinte digitale à notre audio tout en respectant la définition (conditions) d'une empreinte ?

Il reste à retrouver dans cette masse d'empreintes, la centaine d'empreintes obtenues à partir de la musique enregistrée depuis notre microphone.

Puisque les empreintes tirées de l'échantillon sont bruitées, il faut comparer une centaine d'empreinte qui vont chacune correspondre à plusieurs morceaux, donc on se retrouvera avec beaucoup de morceaux potentiellement candidats.

Une empreinte acoustique doit satisfaire plusieurs conditions :

- **Spécificité** : L'empreinte doit être assez précise pour que deux sons très différents n'aient pas la même empreinte.
- **Robustesse aux les distorsions** : Un algorithme d'empreinte acoustique doit tenir compte des caractéristiques audibles du son.

Si deux échantillons sont perçus comme identiques par l'oreille humaine, leur empreinte doit être égale ou très proche, même si leur représentation numérique est différente. À ce titre les empreintes acoustiques ne sont pas des empreintes au sens habituel du terme.

Un bon algorithme d'empreinte acoustique va permettre d'identifier un enregistrement même après qu'il a subi une compression, une légère variation de vitesse ou un bruit léger doit aussi ne pas trop changer l'empreinte.

- **Efficacité** : Pour être utilisée, l'empreinte doit être facile à calculer, et facile à stocker, c'est-à-dire compacte, courte.

Sans oublier la problématique de l'alignement des empreintes. Si l'enregistrement à partir du microphone de l'ordinateur n'est pas commencé depuis le début de la musique alors les empreintes enregistrées à partir du microphone ne vont pas s'aligner correctement avec ceux obtenus à partir de la base de données.

## 5.2 État de l'art : études des solutions existantes

Dans cette partie, nous allons présenter deux approches qui amènent à résoudre ces problématiques tout en précisant les avantages et les inconvénients de chaque représentations.

### 5.2.1 Approche de superposition

Les pics spectraux caractérisent très bien notre signal, ils sont assez spécifiques, robuste au bruit et aux distorsions et plus ou moins efficace, ainsi ils peuvent être explicitement considérés comme une empreinte de notre signal.

Ce que nous voulons dire par cela c'est que nous allons stocker directement les pics spectraux ou les points (*temps, fréquence*) dans notre base de données, ainsi lors d'une recherche de correspondance, nous allons comparer ces pics deux-à-deux.

Il est plus simple d'imaginer ce processus comme une superposition de la carte de constellation du signal enregistré à travers le microphone et celle du signal enregistré dans la base de données.

Le pattern des points doit être le même pour les segments audio correspondants. Si vous mettez la carte des constellations d'une chanson de la base de données sur un ruban, et la carte de constellation de d'un court échantillon audio de quelques secondes sur un morceau de ruban transparent, puis vous glissez le second sur le premier, à un moment donné, un nombre significatif de points coïncideront lorsque le décalage temporel adéquat sera trouvé et que les deux cartes de constellation seront alignées.

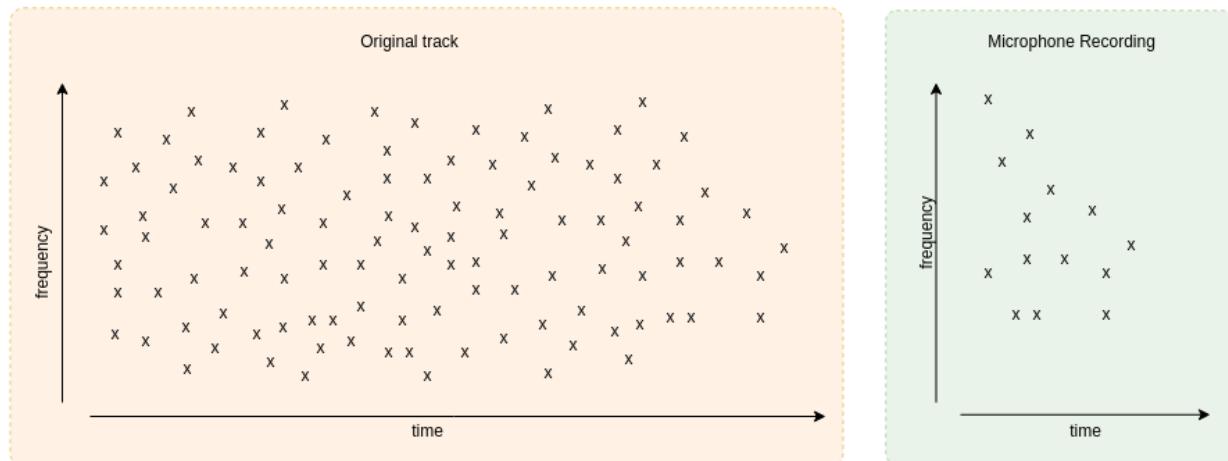


FIGURE 5.1 – Original Track and Recording Constellations

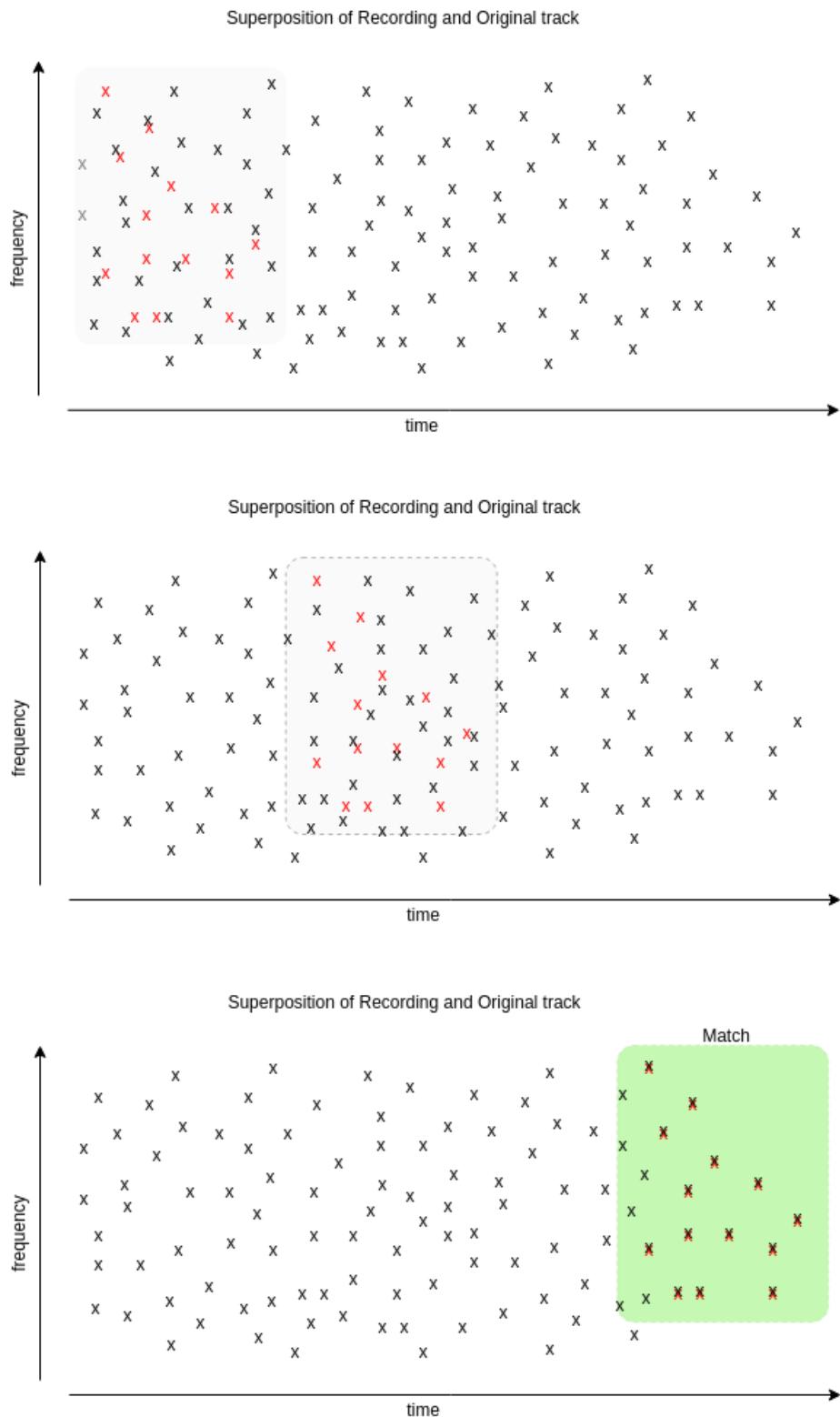


FIGURE 5.2 – Superposition Process

Dans cet exemple, il y a une correspondance parfaite entre l'enregistrement et la fin de la chanson. Si ce n'est pas le cas, nous devons comparer l'échantillon avec une d'autre chansons dans la base de données et ainsi de suite jusqu'à ce qu'on trouve une correspondance parfaite. Et si on ne trouve pas de correspondance parfaite, nous pouvons choisir la correspondance la plus proche que nous avons trouvée (dans toutes les chansons) si le taux de correspondance est supérieur à un seuil qui pourra être fixé en paramètre.

Par exemple, si la meilleure correspondance que nous avons trouvée nous donne une similitude de 90% entre le segment échantillon et une partie d'une chanson, nous pouvons supposer que c'est un *match* car les 10% de non-similitude sont certainement dus aux distorsions et au bruit.

### Avantages :

Il s'agit d'une solution directe, aucun traitement supplémentaire sur les cartes de constellation. On les enregistre directement dans la base de données et on les compare tout en respectant la séquentialité temporelle pour un bon alignement.

### Inconvénients :

Bien que les pics spectraux sont résistants au bruit et aux distorsions, **individuellement**<sup>1</sup> ils ont une faible spécificité, cela veut dire qu'ils ne sont pas très caractéristiques lorsqu'ils sont considérés de manière isolée. Par conséquent, lorsqu'on utilise des pics spectraux individuels et leurs timbres de fréquence comme valeurs de hachage, alors les listes de hachage résultantes (la liste des numéros de page) sont longues et l'indexation devient lente.

### 5.2.2 Approche de hachage combinatoire

L'idée principale est de former des hachages d'empreintes digitales en considérant des paires de pics plutôt que des pics individuels. Pour cela, on fixe un point  $(n_0, k_0)$  qui sert de point d'ancrage ainsi qu'une zone cible  $Z \times$  qui lui est associée. La zone cible doit être considérée comme une petite région rectangulaire dans le plan temps-fréquence proche du point d'ancrage. On considère alors des paires de points  $((n_0, k_0), (n_1, k_1))$  constitués du point d'ancrage  $(n_0, k_0)$  et d'un certain point cible  $(n_1, k_1) \in Z$ . Chaque paire donne lieu à un triplet  $(k_0, k_1, n_1 - n_0)$  constitué de deux estampilles de fréquence et d'une différence de deux estampilles de temps. L'idée est d'utiliser ces triplets comme des hachages au lieu d'estampilles de fréquence simples.

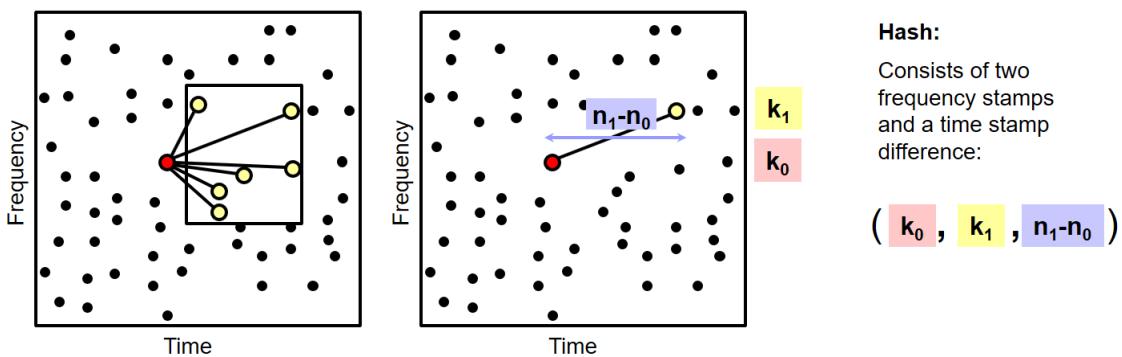


FIGURE 5.3 – Combinatorial Hash Generation

Au stade de la mise en correspondance, on compte maintenant les triples correspondances entre la requête décalée et le document de la base de données au lieu de considérer uniquement les timbres de fréquence correspondants. On peut montrer que, même si le nombre d'éléments de données à indexer augmente (en considérant les triples au lieu des timbres de fréquence simples), on obtient une accélération considérable du processus de recherche grâce à la grande spécificité beaucoup des triples.

1. pour éviter une contradiction avec ce qu'on a dit au début de cette section.

## Avantages :

Cette solution permet de gagner une accélération dans la recherche d'une correspondance, sans oublier que les paires spectraux sont indépendants du points de vue temporelle c'est à dire un paire  $(k_0, k_1, \Delta t_0)$  est indépendant temporellement du paire  $(k_2, k_3, \Delta t_2)$  et ainsi de suite, ce qui rend la comparaison entre les paires plus simple voire plus robuste.

## Inconvénients :

Ce processus consiste à itérer sur les pics spectraux, chaque pics sera considéré comme un point d'ancrage et associé à une zone cible puis itérer sur cette zone pour construire les paires à trois valeurs  $(k_0, k_1, \Delta t_0)$  : 2 valeurs de fréquence absolues et 1 valeur de différence temporelle. Donc on peut facilement conclure que ce processus peut générer une quantité immense de données en fonction de la taille de la zone cible, ce qui pourra poser des problématique du point de vue de mémoire.

## 5.3 Solution proposée et sa mise en œuvre

D'après l'étude des solutions existantes réalisée précédemment, nous avons choisi d'implémenter la solution du *Hachage Combinatoire* afin de construire les empreintes digitale de n'importe quel signal d'entrée. Bien que celle-ci posera un défi du point de vue de mémoire, nous allons voir par la suite comment nous allons optimiser l'espace qui sera occupé par le nombre immense d'empreintes.

La procédure du *Hachage Combinatoire* ce réalise en passant par deux étapes cruciales :

- Construction des paires en utilisant les zones cibles.
- Conversion des repères en valeurs de hachage.

### 5.3.1 Construction des paires

Nous choisissons une zone cible en fonction des voisins, c'est à dire pour un point d'ancrage  $k$ , nous allons choisir un nombre fini de points à son voisinage, ce nombre est défini en paramètre de notre application.

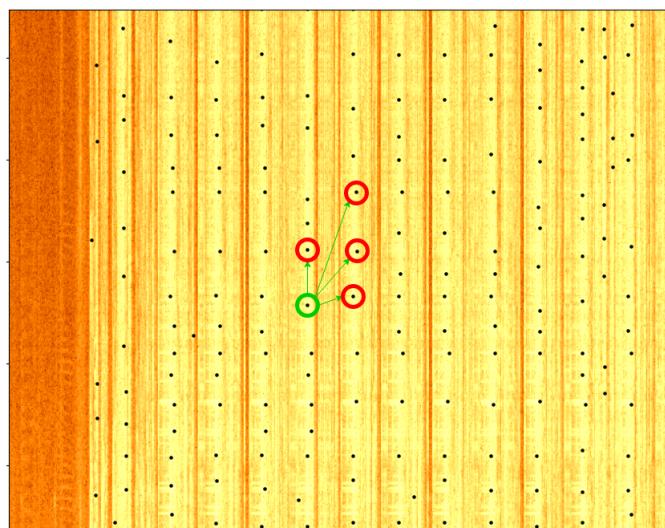


FIGURE 5.4 – Pairs Connections

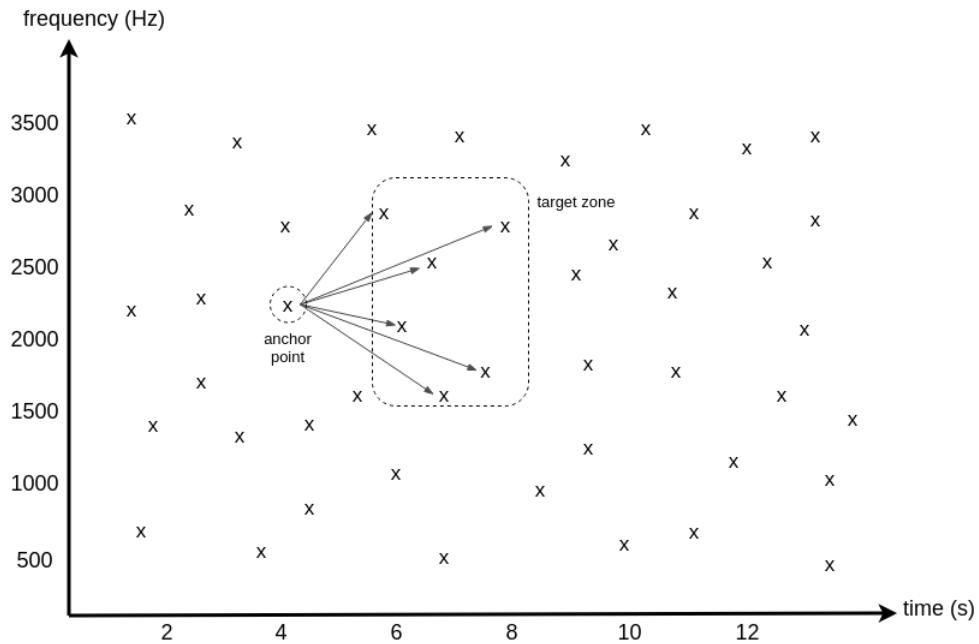


FIGURE 5.5 – Anchor Point and Target Zone

A partir d'un point donné, ses voisins sont choisis en fonction de leur distance, c'est à dire les points les plus proches temporellement.

Par souci de compréhension, nous fixerons la taille de la zone cible à 5 points temps-fréquence. Afin d'être sûr que l'enregistrement et le morceau complet généreront les mêmes zones cibles, nous avons besoin d'une relation d'ordre entre les points (temps-fréquence) dans un spectrogramme filtré :

- Si deux points (temps,fréquence) ont le même temps, le point avec la fréquence la plus basse est avant l'autre.
- Si un point (temps,fréquence) a un temps inférieur à un autre point, alors il est avant.

Voici ce que nous obtenons si nous appliquons cet ordre sur le *Spectrogramme* simplifié que nous avons vu précédemment :

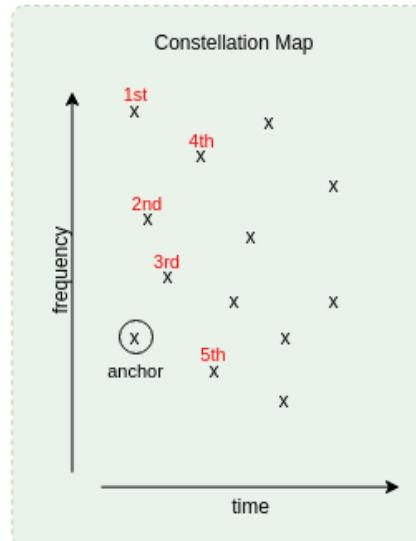


FIGURE 5.6 – Pairs Order

On associe à chacun des pics extraits (les points ancrés, anchor points) une zone cible (target zone). La taille de la zone cible influe directement sur le nombre de paires résultant.

À chaque *anchor point* est associée sa position temporelle par rapport au début du signal, et chaque paire  $\{anchor\ point, target\ point\}$  est codée par 3 valeurs  $(f_1, f_2, \Delta t)$  : 2 valeurs de fréquence absolues et 1 valeur de différence temporelle. Ces couples sont formés de trois éléments :

- la fréquence du point d'ancrage ( $f_1$ )
- la fréquence du point de la zone cible ( $f_2$ )
- la différence de temps entre les deux ( $\Delta t$ ).

Nous allons voir par la suite comment nous allons convertir cette représentation de triplet à un hachage reproductible et simple à stocker.

### 5.3.2 Conversion des repères en valeurs de hachage.

Le hachage est Opération qui consiste à appliquer une fonction mathématique permettant de créer l'empreinte numérique d'un message, en transformant un message de taille variable en un code de taille fixe, en vue de son authentification ou de son stockage.

Une fonction de hachage est typiquement une fonction qui, pour un ensemble de très grande taille (théoriquement infini) et de nature très diversifiée, va renvoyer des résultats aux spécifications précises (en général des chaînes de caractère de taille limitée ou fixe) optimisées pour des applications particulières. Les chaînes permettent d'établir des relations (égalité, égalité probable, non-égalité, ordre...) entre les objets de départ sans accéder directement à ces derniers, en général soit pour des questions d'optimisation (la taille des objets de départ nuit aux performances), soit pour des questions de confidentialité.

En termes très concrets, on peut voir une fonction de hachage comme un moyen de replier l'espace de données que l'on suppose potentiellement très grand et très peu rempli pour le faire entrer dans la mémoire de l'ordinateur.

Après avoir formé des paires de points  $\{anchor\ point, target\ point\}$  comme décrit avant, et que nous les avons encodé par 3 valeurs  $(f_1, f_2, \Delta t)$  : 2 valeurs de fréquence absolues et 1 valeur de différence temporelle, tel que :

- la fréquence du point d'ancrage ( $f_1$ )
- la fréquence du point de la zone cible ( $f_2$ )
- la différence de temps entre les deux ( $\Delta t$ ).

Ensuite nous avons choisi la fonction SHA-1 afin de hacher ces 3 valeurs. Ces haches sont tout à fait reproductibles, même en présence de bruit et de la compression du codec vocal.

```
1 hash(frequencies of peaks, time difference between peaks) = fingerprint hash value
```

Chaque hachage est également associé au décalage temporel entre le début du fichier respectif et son point d'ancrage, ce que nous appelons dans notre application un *offset*. Bien que le temps absolu ne fasse pas partie du hachage lui-même.

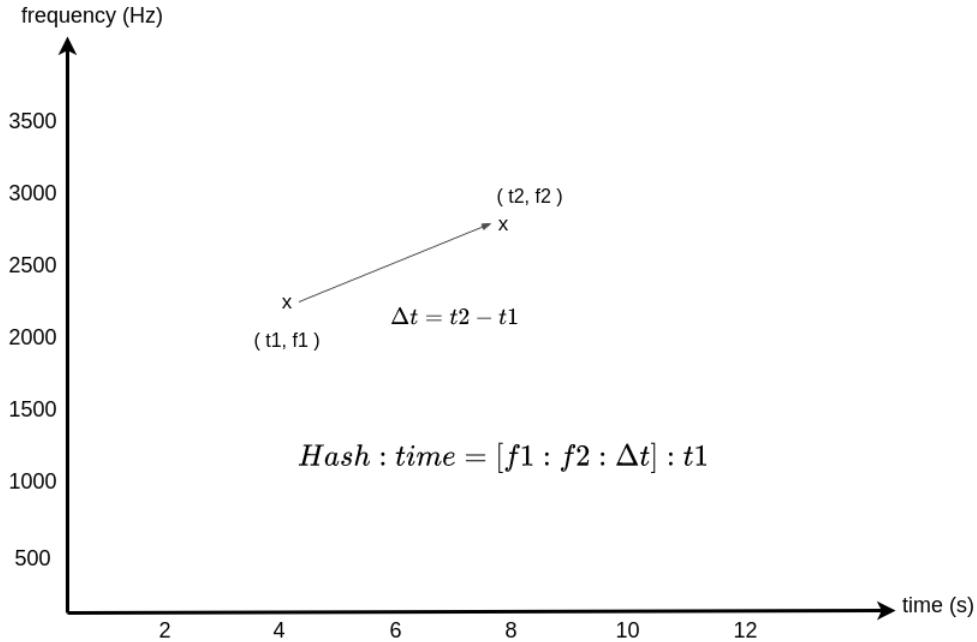


FIGURE 5.7 – Combinatorial Hash Function

Ce couple est ce que l'on appelle un *marqueur*. On assigne ensuite à ce *marqueur* l'instant  $t$  du point d'ancrage pour obtenir un *marqueur temporel*. Le regroupement de tous les marqueurs temporels forme l'empreinte.

A cette empreinte, on associera par la suite les informations du morceau (artiste, nom de la chanson, etc). En moyenne, une empreinte se retrouve dans seulement trois morceaux.

Pour effectuer une recherche, l'étape d'empreinte digitale ci-dessus est exécutée sur un échantillon de fichier sonore capturé pour générer un ensemble de *[hache : offset]*. Chaque hachage de l'échantillon est utilisé pour rechercher dans la base de données les haches correspondants. Pour chaque hache correspondant trouvé dans la base de données, les temps de décalage ou *offset* correspondants depuis le début de l'échantillon et les fichiers de la base de données sont associés en paires de temps. Les paires de temps sont distribuées dans des bacs en fonction de l'ID de la musique associée au hache correspondant dans la base de données.

Nous allons voir dans le chapitre *Base de données : stockage et recherche*, en détails, la procédure de recherche et le nombre d'empreintes généré pour une collections de 50 musiques ainsi étudier la problématique de mémoire et voir comment on pourra optimiser l'espace occupé par les haches.

## 6 Base de données : stockage et recherche

Récapitulons ce que nous avons réalisé jusqu'à maintenant :

1. Nous avons d'abord capturer le signal en utilisant le microphone et avec une fréquence d'échantillonnage égale à 44 100 Hz.
2. Une fois le signal est numérisé, nous appliquons la fonction *Short-Time Fourier Transform (STFT)* afin de passer d'une représentation temporelle simple à une représentation fréquentiellement-temporelle et générer un *Spectrogramme* du signal.
3. Ensuite nous avons appliqué un filtre pour extraire les maxima locaux de ce *Spectrogramme* et un deuxième filtre pour garder seulement les amplitudes qui dépassent un certain seuil. Ceci nous donne les pics spectraux du signal, ou en autres mots les amplitudes importantes du signal qui résistent aux bruits et autres distorsions. Bien-sûr nous gardons seulement le couple (*temps, fréquence*) associé à chaque pic spectral comme ça on aura une représentation binaire.
4. Après nous avons vu qu'au lieu de stocker directement les couples (*temps, fréquence*) des pics spectraux directement dans une base de données puis comparer les points deux-à-deux afin de trouver une correspondance entre deux signaux, nous formons des paires de points en choisissons un point d'ancrage et une zone cible, puis on garde les fréquences du point d'ancrage et du point cible voir aussi la différence temporelle entre ces deux points, ce qui nous donne une représentation plus compacte et suffisamment informative qui pourra être facilement stocker.
5. Finalement nous utilisons SHA-1 pour hacher cette représentation ainsi créer une empreinte digitale du signal d'entrée ce qui rend la recherche/comparaison plus simple voire plus rapide.

Tous ces éléments réunis nous donne la possibilité de mettre en œuvre une reconnaissance de musique car un tel système doit réaliser deux tâches principales :

1. Apprendre de nouvelles chansons en enregistrons leurs empreintes digitales.
2. Reconnaître des chansons inconnues en les recherchant dans la base de données des chansons déjà apprises.

Alors dans ce chapitre nous allons voir comment nous avons traité le côté base de données (tables, contraintes sur des champs, indexation...) afin de stocker une large quantité de données tout en permettant une recherche rapide qui ne dépasse pas 1 seconde pour trouver un *match*.

Certes qu'après l'extraction des points importants du *Spectrogramme* du signal puis le hachage combinatoire de ces points, nous diminuons largement la quantité de données à stocker. Néanmoins, cette quantité reste quand même très grande car nous obtenons des milliers de haches par signal, donc si nous appliquons une extrapolation sur une collection de 8411 musiques mp3 à hacher et à stocker, alors il est clair que notre base de données contiendra des millions voire des milliards d'articles (lignes). Ceci nous oblige à étudier la façon dans laquelle nous allons créer nos tables ainsi l'architecture globale de notre base de données afin de permettre une fouille très rapide pour trouver un *match*.

Nous avons donc étudié deux approches de mise en place de la base de données, dans ces deux approches nous aurons que deux tables, or, il y aura une différence très importante quand il s'agit des champs et des contraintes.

## 6.1 Architecture initiale de la base de données

Il s'agit d'une approche simple qui ne prend aucune considération de la complexité en mémoire et en temps de recherche. Nous avons choisi de parler de cette architecture afin d'accentuer le niveau d'optimisation atteint dans notre application et aussi pour montrer que des petites modification dans les contraintes des champs peut avoir un grand impacte sur les performances de la base de données.

### 6.1.1 Mise en œuvre

Commençant d'abord par la table **SONGS**, nous l'utiliserons essentiellement pour contenir des informations sur les chansons. Nous en aurons besoin pour associer un `song_id` à la chaîne de caractères du nom de la chanson :

```
1 CREATE TABLE IF NOT EXISTS SONGS (
2     song_id MEDIUMINT UNSIGNED NOT NULL PRIMARY KEY AUTO_INCREMENT,
3     song_name VARCHAR(250) NOT NULL,
4     filehash VARCHAR(60)
5 );
```

Cette table ne change pas dans les deux architectures que nous étudions. Or, la vrai différence est sera dans la deuxième table intitulée **FINGERPRINTS**, dans cette première approche la table est créée comme suit :

```
1 CREATE TABLE IF NOT EXISTS FINGERPRINTS (
2     id INTEGER PRIMARY KEY AUTO_INCREMENT,
3     song_fk INTEGER NOT NULL,
4     hash VARCHAR(40) NOT NULL,
5     offset INTEGER NOT NULL,
6     FOREIGN KEY (song_fk) REFERENCES SONGS (song_id)
7 );
```

Tout d'abord, remarquez que nous avons non seulement champ `hash` et un identifiant `song_id`, mais aussi un `offset`. Celui-ci correspond à la fenêtre temporelle du *Spectrogramme* d'où provient le hachage. Cela entrera en jeu lorsque nous devrons filtrer les haches correspondants pendant la recherche. Seuls les haches qui "s'alignent" proviendront du véritable signal que nous voulons identifier. En supposant que nous avons déjà crée des empreintes digitales des pistes connues, c'est-à-dire que nous avons déjà inséré nos empreintes digitales dans la base de données étiquetée avec les ID des chansons, nous pouvons simplement chercher un *match*.

Notre pseudo-code pour une recherche d'un match sera écrit comme suit :

```
1 channels = capture_audio()
2
3 fingerprints_matching = []
4 for channel_samples in channels
5     hashes = process_audio(channel_samples)
6     fingerprints_matching += find_database_matches(hashes)
7
8 predicted_song = align_matches(fingerprints_matching)
```

Qu'est-ce que cela signifie pour les hachages d'être alignés ? Considérons l'échantillon que nous écoutons comme un sous-segment de la piste audio originale. Une fois que nous avons fait cela, les hachages que nous extrayons de l'échantillon auront un décalage relatif au début de l'échantillon.

Le problème, bien sûr, est que lorsque nous avons crée les empreintes digitales des pistes originales *mp3*, nous avons enregistré le décalage absolu du hachage. Les hachages relatifs de l'échantillon et

les hachages absolus de la base de données ne correspondront jamais, sauf si nous avons commencé à enregistrer un échantillon exactement au début de la chanson. Assez peu probable.

Mais même si elles ne sont pas identiques, nous avons une idée sur les correspondances à partir du signal réel derrière le bruit. Nous savons que tous les décalages relatifs seront à la même distance les uns des autres. Bien-sûr il faut que la chanson est lue et échantillonnée à la même vitesse qu'elle a été enregistrée pendant la création des empreintes (44 100 Hz).

Dans cette hypothèse, pour chaque correspondance, nous calculons une différence entre les décalages :

```
1 difference = database offset from original track - sample offset from recording
```

Ce qui donnera toujours un entier positif puisque la chanson de la base de données sera toujours au moins de la même longueur de l'échantillon. Toutes les vraies correspondances auront cette même différence. Ainsi, nos correspondances de la base de données sont modifiées pour ressembler à ceci :

```
1 (song_id, difference)
```

Maintenant, il suffit de regarder toutes les correspondances et de prédire l'identifiant de la chanson pour laquelle le nombre de différences est le plus élevé. C'est facile à imaginer si vous le visualisez sous forme d'histogramme.

### 6.1.2 Tests et performance

Maintenant que notre base de données est prête à être utilisée, nous allons choisir les mêmes paramètres qu'avant quand il s'agit du traitement de signal et qui sont :

- Fréquence d'échantillonnage (FS) = 44100
- Taille de la fenêtre utilisée dans la fonction STFT (WINDOW SIZE) = 4096
- Ratio du chevauchement des fenêtres consécutive (OVERLAP RATIO) = 0.5 (la moitié)
- Seuil d'amplitude minimale pour être considérée comme un pic spectral (AMP MIN) = 10
- Taille de la région des voisins utilisée dans la recherche des maxima locaux (NEIGHBORHOOD SIZE) = 20
- Taille de la zone cible pour chaque point d'ancre (FAN VALUE) = 15

Ensuite nous allons utiliser 50 chansons mp3 afin de peupler notre base de données avec leurs empreintes digitales ainsi étudier la taille de mémoire occupée et le temps de recherche d'un *match*.

Voici un tableau qui montre le nombre de lignes dans chaque tables ainsi que sa taille totale :

table	number of lines	data size (Mo)	index size (Mo)	total size (Mo)
SONGS	50	0.016	0.016	0.016
FINGERPRINTS	9 681 644	702.0	116.7	818.7

On remarque bien que même après extraction de pics le nombre d'empreintes est immense, 9 681 644 empreintes pour seulement 50 musiques. On peut imaginer ce nombre atteindre des milliards voire même plus si on rajoute d'autres musiques à notre base de données.

Voici un graphique qui montre le nombre d'empreintes (ou hache) pour chaque musique dans la base de données :

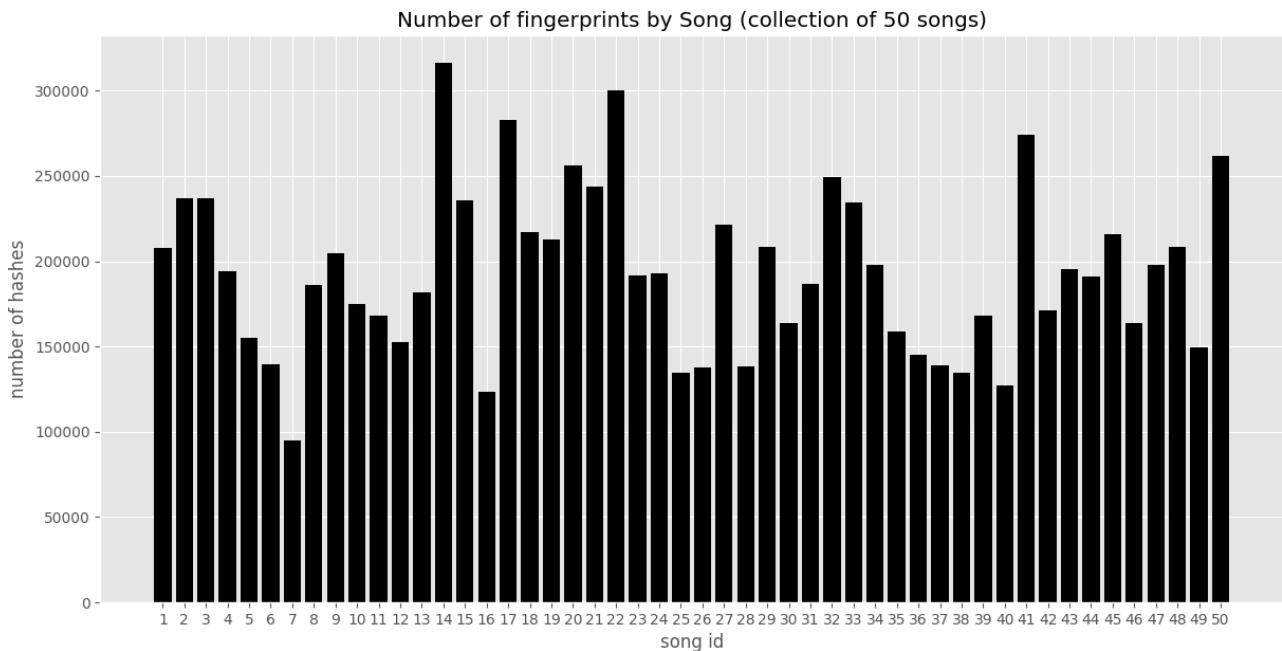


FIGURE 6.1 – Bar chart representing the number of fingerprints by song

Ce graphique montre la grandeur que notre application peut atteindre en terme de mémoire, puisqu'on peut bien avoir des musiques avec un nombre d'empreintes qui dépasse 300 000, et que la moyenne dépasse 100 000.

Il faut aussi retenir que ce nombre est un résultat direct des pics du signal, et que le nombre de pics peut diminuer si on change les paramètres cités au début. Ceci résultera une décroissance dans le nombre d'empreintes ainsi moins d'espace occupé. Or, moins d'empreintes signifie moins de précision. Donc il faut absolument réaliser une étude approfondie pour trouver les paramètres optimales en terme de mémoire et de précision.

En même temps il faut se demander si il existe un moyen pour optimiser la mémoire, même avec une petite marge, sans changer les paramètres.

## 6.2 Optimisation de la base de données

Comme décrit avant, la table **SONGS** ne changera pas. Néanmoins nous allons voir des améliorations non négligeable en changeant la table **FINGERPRINTS**.

### 6.2.1 Mise en œuvre

Dans cette deuxième approche la table **FINGERPRINTS** est créée comme suit :

```

1 CREATE TABLE IF NOT EXISTS FINGERPRINTS (
2   song_fk MEDIUMINT UNSIGNED NOT NULL ,
3   hash BINARY(10) NOT NULL ,
4   offset INT UNSIGNED NOT NULL ,
5   INDEX(hash) ,
6   UNIQUE(song_fk, offset, hash) ,
7   FOREIGN KEY (song_fk) REFERENCES SONGS (song_id)
8 );

```

Nous avons créé un INDEX sur notre champs *hash* - avec une bonne raison. Toutes les requêtes devront correspondre à ce hache, nous avons donc besoin d'une récupération très rapide.

Ensuite, l'index UNIQUE permet de s'assurer qu'il n'y a pas de doublons. Puisque nous traitons les deux canaux stéréo, il peut y avoir donc plusieurs informations qui se répètent, une similarité qui peut dépasser 90% d'après notre étude du *spectrogramme* des deux canaux, donc plusieurs (song\_fk, hache, offset) qui se répètent. Il n'est pas nécessaire de gaspiller de l'espace ou d'alourdir la recherche audio en ayant des doublons qui traînent.

Finalement pour optimiser la mémoire occupée par les empreintes nous avons décidé de réduire le stockage disque inutile au niveau de la valeur de hachage. Pour le hachage de nos empreintes digitales, nous commencerons par utiliser un hachage SHA-1, puis nous le réduirons en prenant seulement les 20 premiers caractères, donc la moitié.

```
1 char(40) => char(20) goes from 40 bytes to 20 bytes
```

Ensuite, nous allons prendre cet encodage hexagonal et le convertir en binaire, en réduisant encore une fois considérablement l'espace :

```
1 char(20) => binary(10) goes from 20 bytes to 10 bytes
```

Donc on est parti d'une taille de 320 bits (taille d'un hache SHA-1) pour se retrouver avec 80 bits, une optimisation de 75%.

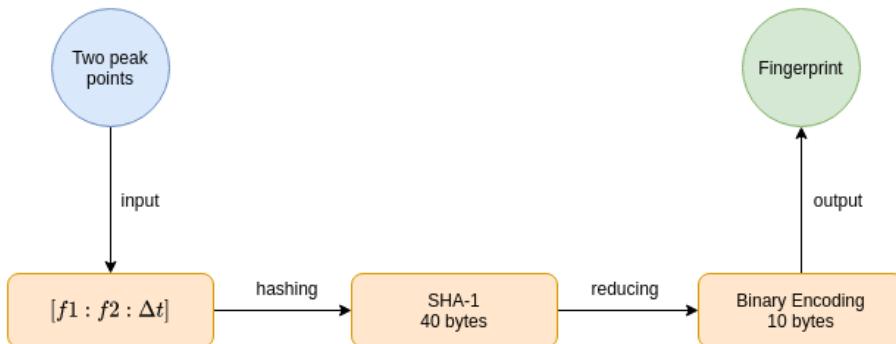


FIGURE 6.2 – Fingerprint size optimisation

Certes nous perdons une partie de l'information - nos hachages seront, statistiquement parlant, beaucoup plus souvent en conflit. Nous avons considérablement réduit l'entropie du hachage. Cependant, il est important de se rappeler que notre entropie (ou information) comprend également le champ *offset*, qui est de 4 octets.

$$10 \text{ bytes (hash)} + 4 \text{ bytes (offset)} = 14 \text{ bytes} = 112 \text{ bits} = 2^{112} \approx 5.2 \times 10^{33} \text{ possible fingerprints}$$

(6.1)

Ce qui nous permet toujours d'avoir une entropie assez forte tout en optimisant la mémoire.

### 6.2.2 Tests et performance

Après avoir appliqué la réduction des haches et une contrainte UNIQUE sur le tuple (*song\_fk*, *hash*, *offset*) nous avons constaté une décroissance dans la taille de mémoire occupée par les 50 chansons, voici un tableau descriptif de cette situation :

table	number of lines	data size (Mo)	index size (Mo)	total size (Mo)
SONGS	50	0.016	0.016	0.016
FINGERPRINTS	7 761 966	459.0	289.0	748.0

Comme avant, voici un graphique qui montre le nombre d'empreintes (ou hache) pour chaque musique dans la base de données :

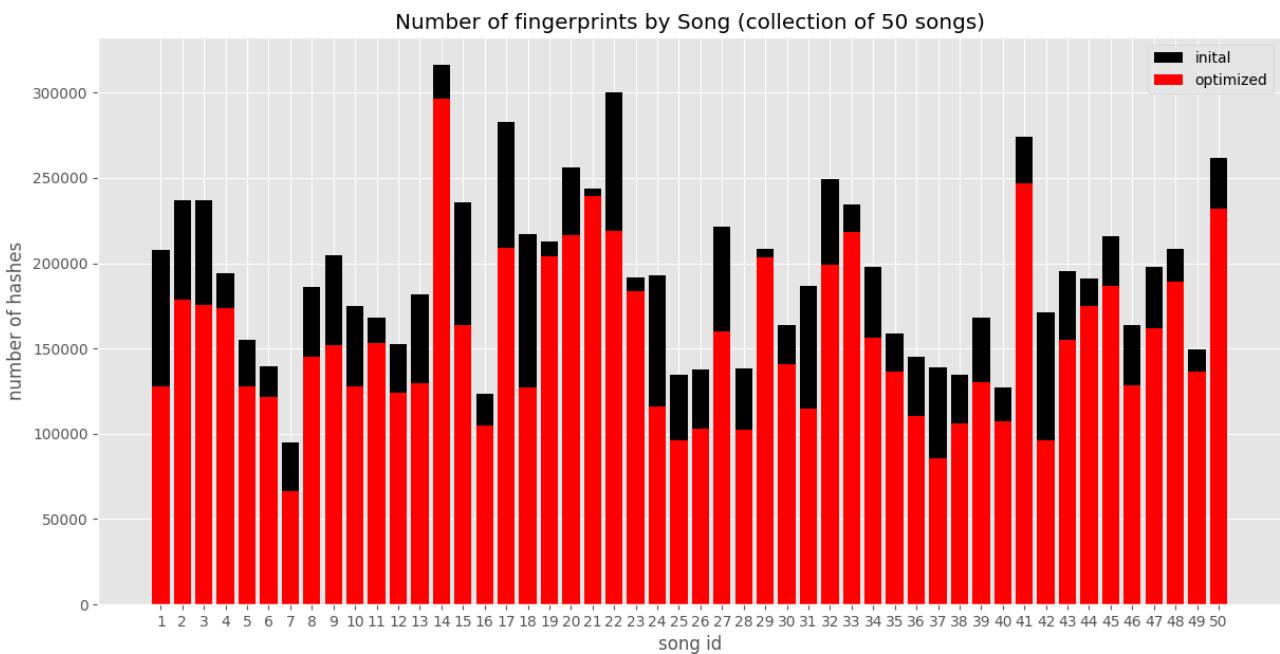


FIGURE 6.3 – Bar chart representing the number of fingerprints by song (initial vs optimized)

Certes la taille finale n'est pas considérablement plus petite que la première approche, or, ceci est dû à l'index que nous avons rajouté sur le tuple (*song\_fk*, *hash*, *offset*), cet index nous permet de réaliser une recherche rapide dans la base de données. Donc si on compare les deux solutions sans prendre en compte la taille des indexes et voir seulement la taille des données (data size) nous trouvons que la deuxième solution utilise beaucoup moins d'espace.

Ceci nous a permis d'économiser de l'espace dans un premier temps, puis de rajouter un index et augmenter la vitesse de recherche dans un deuxième temps.

Il existe un compromis assez direct entre le temps d'enregistrement nécessaire et la quantité de stockage requise. En ajustant le seuil d'amplitude pour les pics et la taille de la région à prendre en considération lors de l'extraction des maxima locaux, vous ajouterez plus d'empreintes digitales et améliorerez la précision au prix d'un espace plus important.

C'est vrai, les empreintes digitales prennent une quantité surprenante d'espace (un peu plus que les fichiers MP3 bruts). Cela semble alarmant jusqu'à ce que vous considériez qu'il y a des dizaines et parfois des centaines de milliers de hachages par chanson. Nous avons également permis de faire correspondre les chansons de manière très fiable en cinq secondes, de sorte que notre compromis espace / vitesse semble avoir porté ses fruits.

Plus tard nous allons voir, en détails, à quel point ces compromis dans l'espace nous ont permis d'avoir une recherche très rapide et robuste aux distorsions.

# 7 Rendu final

A travers ce projet nous avons réaliser une application qui permet de retrouver le titre d'une chanson et son auteur après seulement quelques secondes d'écoute par l'intermédiaire d'un microphone et aussi à travers un fichier mp3. Quand on demande à l'application de reconnaître un morceau, elle décompose le son et le transforme en empreintes. Puis le compare à ceux présentes dans sa base de données et trouve le résultat correspondant.

Pour cela nous avons procédé par le traitement de signal pour extraire des caractéristiques importantes, et nous créons ensuite une empreinte associée à ce signal afin de la stocker et finalement faire la recherche et la comparaison des différentes empreintes dans une large base de données.

## 7.1 Interface utilisateur finale

Par souci de temps, nous nous sommes limité à une version terminale, une interface interactive permet à l'utilisateur d'interagir avec un programme informatique, grâce à l'exécution du programme. Néanmoins, notre application ne demande pas de grands besoins du côté UI, une interface web ou mobile peut être réalisée dans un petit délai de temps en utilisant des frameworks moderne comme *ReactJS* ou *ReactNative*.

Au lancement du programme, six (6) options sont proposées à l'utilisateur qui sont les suivantes :

1. Reconnaître une chanson à partir du microphone.
2. Reconnaître une chanson à partir d'un fichier mp3.
3. Traiter, Hacher, Stocker une ou plusieurs musiques dans la base de données.
4. Afficher les détails de la base de données.
5. Réinitialiser la base de données.
6. Quitter le programme.

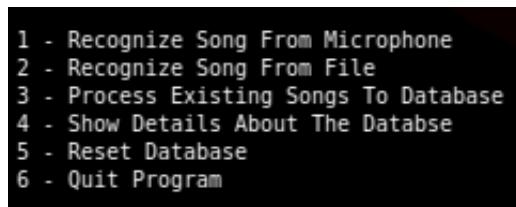


FIGURE 7.1 – Terminal Interactive Interface

## 7.2 Tests utilisateur et certification

**Reconnaître une chanson à partir du microphone :**

L'option une (1) demande à l'utilisateur le temps qu'il souhaite pour enregistrer la musique diffusée dans son environnement puis un tableau contenant les top 4 correspondances est affiché en montrant surtout les noms des chansons trouver dans la base de données (et bien-sûr d'autres informations importantes surtout aux développeurs). Sans oublier l'affichage d'un deuxième tableau qui montre le temps d'exécution de chaque opération de la reconnaissance notamment le temps du traitement du

signal enregistré à travers le microphone, et le temps de recherche dans la base de données.

Voici un cas pratique : Ici l'utilisateur choisit l'option une(1), puis 5 secondes d'enregistrement pour une musique déjà dans la base de données.

1							
How many seconds you want to record ?							
5							
RECORDING...							
FINISHED RECORDING							
SONG_ID	SONG_NAME	INPUT_HASHES	HASHES_MATCHED	INPUT_CONFIDENCE	OFFSET	OFFSET_SECS	SONG_FILE_HASH
42	21. Drake - Know Yourself	819	19	0.020	109	5.062	CE4864B6B81B82744 34F25CFD7485AE0CC 280C4F
45	5. Drake - Best I Ever Had	819	16	0.020	1092	50.712	837AA046E733F30FD 3B67DDAA8066515F8 18D096
11	7. Drake - Headlines	819	7	0.010	294	13.653	C521E940B2494BA5A E553AF8AB83C37A89 DEC24B
41	809. Ultramagnetic Mcs - Watch Me Now	819	11	0.010	5669	263.268	E7C83A0A8D471345B B5FD4615C55871D68 8B150D
PROCESSING TIME	QUERY TIME	ALIGN TIME	TOTAL TIME				
0.517	0.132	0.002	0.651				

FIGURE 7.2 – Identification using 5 seconds microphone recording

#### Reconnaître une chanson à partir d'un fichier mp3 :

L'option deux (2) demande à l'utilisateur le chemin du fichier mp3 qu'il souhaite identifier puis se charge de le traiter et finalement chercher une correspondance dans la base de données et afficher le titre et le nom de l'artiste du morceau (si la musique existe dans la base de données).

```
2
What is the file path ?
/home/thorium90/test_samples/drake_know_yourself_5sec.mp3
```

SONG_ID	SONG_NAME	INPUT_HASHES	HASHES_MATCHED	INPUT_CONFIDENCE	OFFSET	OFFSET_SECS	SONG_FILE_HASH
42	21. Drake - Know Yourself	2448	1621	0.660	1927	89.490	CE4864B6B81B82744 34F25CFD7485AE0CC 280C4F
8	23. Drake - HYFR (Hell Ya Fucking Right)	2448	96	0.040	2352	109.227	3A3AAB78C03A3E00B 67601D89B19C8FFF8 58C02E
11	7. Drake - Headlines	2448	97	0.040	2307	107.137	C521E940B2494BA5A E553AF8AB83C37A89 DEC24B
2	004. Pete Rock & C.L. Smooth - They Reminisce Over You (T.R.O.Y.)	2448	30	0.010	843	39.149	066BF252B7E026B8F 3B46208480BB6FA76 6ADF1A
+-----+-----+-----+-----+							
PROCESSING TIME	QUERY TIME	ALIGN TIME	TOTAL TIME				
0.479	0.130	0.008	0.616				

FIGURE 7.3 – Identification using 5 seconds mp3 file

#### Traiter, Hacher, Stocker une ou plusieurs musiques dans la base de données :

L'option trois (3) demande à l'utilisateur le chemin de la collection mp3 qu'il souhaite puis l'application se charge de Traiter, Hacher, Stocker dans la base de données les musiques existantes dans le répertoire indiqué dans le chemin.

```
3
What is the file path ?
./songs/
100% |██████████| 50/50 [00:37<00:00,  1.34it/s]
```

FIGURE 7.4 – Choice 3

#### Afficher les détails de la base de données :

L'option quatre (4) se charge d'afficher les details de la base de données c'est à dire le nombre totales de musiques existantes dans la base de données, et le nombre totales d'empreintes associées, ainsi que la tailles des tables en Mo.

#### Réinitialiser la base de données :

L'option cinq (5) vide la base de données c'est à dire supprimer tout les éléments dans les deux tables de la base de données (FINGERPRINTS et SONGS).

#### Quitter le programme :

L'option six(6) ferme le programme avec un message de Bienveillance "Good bye"

## 7.3 Autres tests et certifications

Dans cette partie nous allons détailler une séries de tests importants afin de prouver la robustesse de notre application ainsi que la rapidité de l'identification.

Ces tests seront divisés en deux grandes parties :

1. Série de tests sur une petite collection de musique (50 musiques en totale). Dans cette série nous allons voir :
  - Des tests en utilisant des fichiers mp3 purs puis bruités.
  - Des tests en utilisant le microphone de l'ordinateur.
2. Série de tests sur une large collection de musique, ainsi une immense base de données. Dans cette série surtout lancer des tests en utilisant le microphone de l'ordinateur.

### 7.3.1 Tests sur une base de données de 50 musiques

Avant de commencer les tests voici la taille de la base de données ainsi que le nombre d'empreintes totales qui existent :

table	number of lines	data size (Mo)	index size (Mo)	total size (Mo)
SONGS	50	0.016	0.016	0.016
FINGERPRINTS	7 761 966	459.0	289.0	748.0

#### A travers le microphone

Nous avons établi un le protocole suivant afin d'accomplir cette série de tests dans un contexte scientifique :

1. Une enceinte Google Home Mini est placée à 3 mètres du microphone.
2. L'intensité sonore moyenne de cette enceinte est de 60 dB.
3. Une musique est lancer sur l'enceinte et un enregistrement est effectué à travers le microphone de l'ordinateur.

Voici donc les résultats obtenus en variant le nombre de secondes enregistré à travers le microphone :

Number of Seconds	Number Correct	Percentage Accuracy	Average Execution time (sec)
1	37/50	74%	0.152
2	48/50	96%	0.268
3	49/50	98%	0.386
4	50/50	100%	0.477
5	50/50	100%	0.578
6	50/50	100%	0.684

Même avec une seule seconde, choisie au hasard dans la chanson, notre application obtient 74% ! Une seconde ou deux secondes de plus nous permettent d'atteindre environ 98%, alors qu'il fallait 4 secondes ou plus pour atteindre la perfection.

## A travers des fichier mp3

Dans cette partie nous aurons la possibilité de rajouter du bruit à notre signal et tester la précision de notre application en fonction sur des segments de 5 secondes des 50 fichiers mp3.

Voici donc les résultats obtenus en rajoutant un bruit réel qui s'agit d'une foule de personne qui parlent dans bar tout en variant l'intensité de ce bruit :

Crowd Noise Loudness (dBFS)	Number Correct	Percentage Accuracy	Average Execution time (sec)
-17.22	49/50	98%	0.610
-12.22	48/50	96%	0.589
-7.49	46/50	92%	0.571
-4.03	37/50	74%	0.582
-2.10	25/50	50%	0.599
-1.11	2/50	4%	0.599

Voici donc les résultats obtenus en rajoutant un bruit blanc gaussien tout en variant l'intensité de ce bruit :

GW Noise Loudness (dBFS)	Number Correct	Percentage Accuracy	Average Execution time (sec)
-17.78	49/50	98%	0.572
-12.77	46/50	92%	0.566
-7.77	39/50	78%	0.565
-3.28	14/50	28%	0.569
-1.53	11/50	22%	0.564
-0.72	7/50	14%	0.564

On remarque bien que notre application est assez robuste même à l'existence des bruits.

Autres tests seront réalisés avant le jour de la soutenance.

### 7.3.2 Tests sur une base de données de 8000 musiques

Cette partie sera complète avant le jour de la soutenance, elle nécessite un temps de traitement qui dépasse 58 heures.

Le but derrière cette série de tests est d'étudier le changement du temps de recherche de correspondance en fonction de la taille de la base de données. Est ce que nous allons voir un temps de recherche qui augmente si notre base de données contient des milliards d'empreintes digitales ?

# 8 Gestion de projet

## 8.1 Méthode de gestion

### 8.1.1 Organisation du travail en équipe :

La réalisation de ce projet a été confié à deux (2) personnes :

- EL HOUFI Othman chargé de Traitement du signal acoustique.
- DIAWARA Mohamed Lamine chargé de Création d'une empreinte acoustique.

Le contexte de la réalisation du projet à deux (2) personnes nous a poussé à mettre en place un référent (chef de groupe) ayant pour rôle de représenter le groupe.

Ce dernier a été choisi naturellement au bout de quelques semaines en fonction de son aptitudes à la communication et de son implication dans la collaboration interne au projet.

Nous avons choisi pour la réalisation de ce projet de mettre en avant les qualités propre à chacun et donc de ne pas imposer, pour la partie gestion de projet, un cahier des charges spécifique à chaque partie.

Chaque personne a donc pu proposer des idées autant dans la conception que dans la réalisation de sa partie. Le référent étant chargé de d'organiser des réunion avec le tuteur de projet, les choix de conception du groupe.

### 8.1.2 Gestion du temps

#### Organisation de la gestion du temps :

Ce projet s'est déroulé durant le semestre 2 de la première année de master Systèmes Intelligents et Communicants à l'Université de Cergy Pontoise.

L'échéance a été fixée à la semaine du 01 decembre de l'année 2020. La première réunion collective a eu lieu lors de la semaine du 18 janvier de la même année. La date de rendu des rapports de projet étant fixée au 11 juin, nous avions donc un délai de 28 semaines pour mener à bien ce projet.

Comme convenu au départ du projet nous avions décidé de laisser chaque personne s'auto gérer au niveau du temps.

Pour que le travail avance de manière homogène nous avons mis en place un système de planning, chaque personne devait tout les mois transmettre un planning expliquant le travail qu'il a réalisé pour les mois à venir écoupé en 4 semaines.

Ces plannings ont servi à produire un diagramme de Gantt qui a permis à chaque personne dans son ensemble de connaître l'avancée de chaque partie. Nous avons donc pu conserver une certaine homogénéité dans le développement de ce projet.

#### Présentation rapide des réunions :

Chaque mois nous avions mis en place une réunion générale et le tuteur technique du projet M. Dimitris KOTZIMOS.

Ces réunions nous permettaient de partager les avancées produites durant les semaines passée et de s'accorder avec le tuteur technique sur le travail à produire durant les semaines à venir.

Durant la partie conception nous avons pu partager des documents servant à expliquer au tuteur technique nos choix, dans le but de mettre en avant les contraintes posées par ces choix et de pouvoir homogénéiser notre travail.

Une fois ce délai passé nous avons pu passer à la partie développement, les réunions ont donc eu pour but principal de partager du code et de faire remonter les problèmes de conception de la base de données à la personne chargée.

### **8.1.3 Outils de travail :**

Nous avons, dès le début du projet, mis en place deux types d'outils de gestion pour la réalisation de ce projet.

Outils de communication :

- Teams pour la communication avec l'encadrant de gestion de projet.
- Zoom pour des réunions avec le tuteur technique.
- Discord pour la communication de base.
- Liste de diffusion de mail.

Outils de gestion de version et de partage de données :

- Système de gestion de version GIT.
- Système de la base de données : PhpMyAdmin.

#### **Outils de communication**

Nous avons choisi d'utiliser pour la communication un groupe privé sur Discord.

Ce choix a été justifié par la simplicité de l'outil Discord et par le fait que tout les membres du projet possédaient un compte et une bonne maîtrise de l'application.

Ce groupe a servi principalement à la diffusion des informations de suivi telles que le choix des horaires de rendez vous collectif ainsi qu'à la mise à disposition de liens vers les outils de gestion de code. La plupart des informations postées sur le groupe ont été doublé d'un mail collectif.

#### **Outils de gestion de version et partage de la base de données :**

Pour ce projet nous avons pu choisir tout les langages de programmation par nos propres moyens et rien ne nous ait imposé par les enseignants.

Langages :

- SQL pour la base de données :

L'outil SQL a été choisi car nous n'avons pas besoin d'une base de données avec des graphes et aussi parce que c'est plus simple d'être en terrain connu. Parce que la majorité des bases de données d'hier et d'aujourd'hui fonctionnent avec le SQL. Ça simplifie les interfaces et les connexions avec d'autres systèmes.

- Python pour le développement du code :

Nous avons choisi Python car il permet de créer des fonctions avec moins de lignes de code, ce qui ne serait pas le cas avec d'autres langages de programmation. C'est un langage facile à apprendre et avec la pratique, il devient possible de créer rapidement un jeu rudimentaire en quelques jours.

### **8.1.4 Difficultés rencontrés dans la gestion du projet et solution apportés :**

Durant la mise en place initiale du projet nous nous sommes heurtés à quelques soucis d'organisation au niveau des choix techniques.

Dans un premier temps le fait que le groupe soit composé que de deux personnes cela n'a pas été simple à gérer aussi bien au niveau des disponibilités horaires. Nous avons eu du mal à trouver une base commune pour la communication avec le tuteur technique.

Néanmoins ce problème a été rapidement réglé par l'institution d'un rendez-vous où la présence était fortement conseillée.

A partir de ce point nous avons pu mettre en place des solutions pour améliorer le dialogue et dans le groupe et avec le tuteur technique.

La page Teams a été créée, elle a permis à toute les parties du projet de pouvoir être tenues au courant rapidement des avancées clés du projet ainsi que des modifications de rendez vous ou du travail demandé par le tuteur responsables.

Une fois la communication de base établie nous avons pu mettre à profit les chacun pour améliorer la cohésion dans le projet.

Chaque étudiant a pu mettre à profit ses connaissances dans les domaines où sa formation (ou auto formation) lui permettait de donner une plus value au projet.

## 8.2 Répartition de tâches

Tâche réalisée	Non	Date
Enregistrement d'un extrait acoustique à travers un microphone	EL HOUFI	23/12/2020
Fonctions d'insertions et de modification de la BDD	DIAWARA	25/12/2020
Création d'un Spectrogramme	EL HOUFI	15/01/2021
Extraction des pics spectraux	EL HOUFI	25/03/2021
Création d'empreinte acoustique (hachage)	DIAWARA	28/03/2021
Mise en place d'une base de donnée relationnelle	DIAWARA	01/04/2021
Fonctions de recherche dans la BDD	DIAWARA	12/04/2021
Comparaison de deux extraits acoustiques.	DIAWARA	19/04/2021
Un « main » complet et Interface interactive sur terminal	EL HOUFI	02/05/2021
Remplissage de la BDD avec une petites collection de musiques (50 musiques)	EL HOUFI	05/05/2021
Calcul de la précision de l'identification en fonction du temps d'enregistrement	EL HOUFI	05/06/2021
Calcul de la précision de l'identification en fonction du bruit ajouté	EL HOUFI	05/06/2021
Création des empreintes pour les 8411 musiques	EL HOUFI	13/06/2021
Remplissage de la BDD avec les empreintes précédemment créées	EL HOUFI et DIAWARA	13/06/2021
Calcul du temps de recherche en fonction de la taille de la BDD	EL HOUFI	16/06/2021

## 9 Conclusion

Ces dernières années, de nombreuses techniques différentes de création d'empreintes digitales et d'indexation ont été proposées et sont maintenant utilisées dans des produits commerciaux. Dans ce projet, nous avons examiné de plus près l'une de ces techniques, qui a été développée à l'origine pour le système d'identification audio *Shazam*. Nous avons discuté des idées principales qui sous-tendent ce système, mais il y a de nombreux paramètres qui doivent être ajustés afin de trouver un bon compromis entre les différentes exigences, notamment la robustesse, la spécificité, l'évolutivité et la compacité. Les aspects importants sont les suivants :

- les paramètres de la STFT (longueur de la fenêtre, taille du saut) qui déterminent les résolutions temporelle et spectrale,
- la stratégie de sélection et d'extraction des pics spectraux (avec ses paramètres de voisinage),
- la taille des zones cibles (utilisées pour définir les triplets), et
- des structures de données appropriées pour le hachage.

Bien que ce système est robuste à de nombreux types de distorsions du signal, l'approche de création d'empreinte discutée n'est pas conçue pour gérer les déformations temporelles. La correspondance des cartes de constellation ainsi que les différences d'horodatage (*timestamp*) dans les paires de pics sont toutes deux sensibles aux différences de tempo relatif entre la requête et le document de base de données. Par conséquent, il est nécessaire d'utiliser d'autres techniques pour être invariant aux modifications de l'échelle de temps.

Les empreintes digitales utilisant les pics spectraux sont conçues pour être très sensibles à une version particulière d'un morceau de musique. Par exemple, face à une multitude d'interprétations différentes d'une chanson par le même artiste, le système d'empreintes digitales est susceptible de choisir la bonne, même si elles sont pratiquement indiscernables par l'oreille humaine. En général, les systèmes d'identification audio sont conçus pour cibler l'identification d'enregistrements qui sont déjà présents dans la base de données. Par conséquent, ces techniques ne sont généralement pas généralisables aux enregistrements en direct ou aux performances qui ne font pas partie de la base de données.

## 10 Perspectives

Comme décrit avant, il y a de nombreux paramètres qui doivent être ajustés afin de trouver un bon compromis entre les différentes exigences, notamment la robustesse, la spécificité, l'évolutivité et la compacité. Trouver des valeurs optimales à ces paramètres pourra augmenter largement les performances de notre application du point de vue de la robustesse aux distorsions, voire aussi du point de vue la mémoire utilisée et la vitesse de recherche d'une correspondance. Or, ce n'est pas une simple tâche, de plus les paramètres de notre application augmentent, le processus de trouver des valeurs optimales à ces paramètres devient très compliqué.

Parmi les solutions que nous envisageons comme extension à notre application est l'utilisation d'un modèle de réseau neurones artificielles qui prendra en entrée les paramètres de notre application, et la sortie sera divisée sur les différentes exigences voulues tel que la robustesse, la mémoire, et le temps de recherche.

Ce réseau sera entraîné sur une large base d'apprentissage qui provienne de plusieurs tests déjà effectués d'une manière dynamique, par exemple nous allons exécuter la reconnaissance des morceaux sur une large collections de musiques tout en ajoutant du bruit et d'autres distorsions et aussi en variant le temps d'enregistrement du microphone, les résultats obtenus feront une très bonne base d'apprentissage pour notre réseau de neurones artificielles. Peut-être même on pourra ajuster les paramètres de notre application dynamiquement par rapport à chaque situation.

# Bibliographie

- [1] Avery L. Wang. An industrial-strength audio search algorithm. In *Proceedings of the 4th Symposium Conference on Music Information Retrieval*, 2003.
- [2] Peter Grosche, Meinard Müller, and Joan Serrà : Audio Content-Based Music Retrieval. In *Meinard Müller and Masataka Goto and Markus Schedl (ed.) : Multimodal Music Processing, Schloss Dagstuhl—Leibniz-Zentrum für Informatik*, 2012.
- [3] Audio Identification : [https://www.audiolabs-erlangen.de/resources/MIR/FMP/C7/C7S1\\_AudioIdentification.html](https://www.audiolabs-erlangen.de/resources/MIR/FMP/C7/C7S1_AudioIdentification.html).
- [4] J. Haitsma, T. Kalker, and J. Oostveen, "Robust Audio Hashing for Content Identification". In *n International Workshop on Content-Based Multimedia Indexing*, 2001.
- [5] C.J. Burges, J. C. Platt, and S. Jana, "Distortion discriminant analysis for audio fingerprinting". In *IEEE Transaction on Speech and Audio Proc*, 2003.
- [6] 6.050J/2.110J – Information, Entropy and Computation – Spring 2008  
6.05https://mtlsites.mit.edu/Courses/6.050/2008/notes/mp3.html.
- [7] Seeing circles, sines, and signals <https://jackschaedler.github.io/circles-sines-signals/sound.html>.
- [8] Piotr Indyk, Rajeev Motwani. Approximate nearest neighbors : towards removing the curse of dimensionality.
- [9] Jerome Schalkwijk, A Fingerprint for Audio <https://medium.com/intrasonics/a-fingerprint-for-audio-3b337551a671>.
- [10] Jang et al. Pairwise Boosted Audio Fingerprint, 2009.
- [11] Short-Time Fourier Transform. In *Sensor Technologies for Civil Infrastructure*, 2014.
- [12] Nasser Kehtarnavaz. In *Digital Signal Processing System Design (Second Edition)*, 2008.