CHALMERS

Simulating Shazam

Acoustic Fingerprinting for Music Identification

Elias Lennevi, Ludvig Sandh, Noah Sundström, Tom Önnermalm

Department of Electrical Engineering

Chalmers University of Technology
Gothenburg 2024
www.chalmers.se
EENX16-24-44

This page was intentionally left blank

Bachelor Thesis 2024

Simulating Shazam –

Constructing an algorithm that creates, utilizes and analyzes acoustic fingerprints for music identification

Elias Lennevi, Ludvig Sandh,
Noah Sundström, Tom Önnermalm

CHALMERS

Department of Electrical Engineering
Chalmers University of Technology
Gothenburg 2024

Simulating Shazam
Acoustic Fingerprinting for Music Identification
Elias Lennevi, Ludvig Sandh, Noah Sundström, Tom Önnermalm

Supervisor: Javad Aliakbari, Doctoral Student - Communication, Antennas and Optical Networks, Electrical Engineering

Examiner: Giuseppe Durisi, Full Professor - Communication, Antennas and Optical Networks, Electrical Engineering.

Bachelor Thesis 2024
Department of Electrical Engineering
Chalmers University of Technology
SE-412 96 Gothenburg

Written in LaTeX
Gothenburg 2024

Abstract

To identify music in a real-time audio recording, sometimes including noisy elements, is not a particularly trivial audio signal processing task. In this study, the goal is to understand and recreate the Shazam algorithm, a method to accurately detect music from real life recordings. The programming language Python is both used to design and implement the discrete signal processing concepts behind the algorithm, and also to build a functioning application that can be used in a smaller scale in a real environment.

The thesis examines the theoretical concepts from which the Shazam algorithm is built upon. The major components of the algorithm will be highlighted, such as the spectrogram generation, fingerprint identification, and fingerprint matching. By analyzing and implementing these components, we aspire to explore optimization techniques to match the performance of the original Shazam algorithm in terms of speed.

Furthermore, this research aims to analyze the algorithm's accuracy in environments with varying noise levels. By training the algorithm on a library of songs spread throughout history and from diverse genres, we hope to develop an application that respects different cultures and is inclusive, showing it would work well in a real life scenario.

Fundamentally, our thesis helps spread the understanding of the applications of advanced signal processing techniques, especially in the area of music identification. By analyzing and recreating the Shazam algorithm, we aim to highlight its inner workings, limitations, and potential further improvements.

Keywords: Acoustic fingerprint, sound recognition, Python algorithm, digital signal processing

Table of Contents

1 INTRODUCTION

In today's society, music is all around us. Whether you are walking down a street, listening to the radio, or watching a movie, you are prone to hear it. Music is being played either in the background or foreground of almost all of our surroundings. Music can also convey emotions and feelings through rhythm, melody, tone and more [1], therefore, music can be a cultural bridge between people. Regardless of genre or origin, whether it is the catchy beat of a pop song, the soulful charm of country music or the calm chaos in jazz, music unites and inspires people. Thus, when one hears an unknown song that one wants to listen to again, it would be difficult to search through all of the existing songs today to find it. Spotify, the audio streaming service, disclosed that in 2019, on average, 27 000 new tracks were added to their service each day, 55 000 in 2020 and 33 000 in 2021 [2]. In today's society, most people carry their personal mobile phones everywhere they go which could be used to identify unknown songs.

The following sections will introduce the background to music identification using algorithms, followed up with the project's purpose, goals and boundaries.

1.1 Background

Shazam, a music recognition service created by Shazam Entertainment, Ltd., was initially an idea in the year 2000 to give people a tool to recognize and identify songs using their mobile phones [3]. The Shazam algorithm had to identify songs from a short audio sample, for example from the radio, a bar, or a movie, without being affected by background noise. The algorithm could also filter out distortion, reverb as well as background conversations and even identify songs slightly altered in remixes [3]. The same source stated that with almost two million songs in their database, Shazam had to focus on performance as well to be able to quickly give their users accurate answers. When launched in 2003, users dialed in to the service via a phone number and up to 15 seconds of the music was sampled and sent to Shazam's server. Thereafter a text message with the result was sent to the user with information about the artist and song title [3]. Today the technology has moved into an app that can be downloaded on a smartphone, rather than being a dialing service.

1.2 Related technology

With the increasing digital music consumption and technical development, users need quicker music identification to find songs. The Shazam service has made it possible for the public to quickly identify and discover songs as well as receive information about the artists and album. Music identification in complex environments has been streamlined in conjunction with advancements in signal processing and machine learning [4]. Several programming libraries have made it easier to develop audio programs and have simplified access to processing power for developers [5].

Signal processing is being used widely throughout many different industries and applications. Within other fields of research, for instance mental illnesses, signal processing is being used to understand the connection between the brain and the physiological responses, as well as developing diagnostic instruments that rely on the same principles as music identification [6]. Advancements in digital signal processing lead to these technologies becoming useful in more areas than just those related to the music industry.

In summary, for music streaming to continue growing - and to meet users' ongoing expectations for improvement in these areas - music identification and signal processing have become integral parts of the digital landscape. This topic is highly relevant as the foundation of the technology can also be applied in other industries.

1.3 Purpose

The purpose of this project is to create a functioning application for computer units that can identify songs by listening to recorded samples of them and display the name of the song, artist, album and timestamp.

By developing this application, the aim is also to obtain more knowledge regarding implementing the principles of the Shazam algorithm, which would result in a deeper understanding of the functionality and applications of sound recognition. The study also focused on the how the performance of the algorithm is affected by different modifications.

Lastly, the purpose of this project is also to improve the groups' skills in programming and signal processing in an applicable environment.

1.4 Goals and requirement specification

This section addresses the goals of this project. By splitting them into four parts, a more legible understanding regarding the project's challenges is clarified to achieve the aforementioned purpose.

- **Sound recognition via software** – The initial challenge is creating a reliable algorithm that can recognize songs through a short acoustic sample between three to twelve seconds. The total time from starting the algorithm to finding the song should be maximum 30 seconds.
- **Database** – The database should contain at least 100 songs representing various aspects of the music industry. It should be diverse and representative, including music from different genres and artists from around the world.
- **User Interface** – To make the application appealing and easy to use, it needs a simple and straightforward interface, providing information on how the application works and displaying metadata for identified songs.
- **Software Testing** – To ensure the functionality of the algorithm, thorough testing is necessary. Tests are also needed to ensure the fingerprint database works correctly and returns accurate information.

A concrete set of requirements is displayed in this section. The table 1.1 below shows them in detail, their priority (must/want), as well as the method of verification. These requirements are used in section 5.1, where they will be compared to the results of the project.

Table 1.1: *The table containing the requirements specifications for the project. The requirements are grouped into sections, each regarding a separate aspect of the application.*

Criteria	Must / Want	Verification
1. Database		
1.1 Number of songs are enough	Must	Number of songs >= 100
1.2 Songs are diverse in genres	Must	Number of genres >= 5
1.3 Songs are diverse in time	Must	At least three songs from each decade from 1980 and onward
2. Recording		
2.1 The recorded samples are not too long	Must	< 12s
2.2 The recorder is non-blocking		The sample can be read while the recording is still ongoing
3. Performance		
3.1 Recognition cannot take too long	Must	If the song is not found within 30s after the recording started, the song is considered 'not found'
3.2 Recognition is partly accurate without noise	Must	Out of 100 clean 5s random samples from the database, 80 must be identified
3.3 Recognition is very accurate without noise	Want	Out of 100 clean 5s random samples from the database, 95 must be identified
3.4 Recognition is partly accurate with low noise levels	Must	Out of 10 random songs from the database, 5 must be identified with normal talking in the background
3.5 Recognition is very accurate with low noise levels	Want	Out of 10 random songs from the database, 8 must be identified with normal talking in the background
3.6 Recognition is possible with high levels of noise	Must	At least one song should be identified in an environment with strong background noise, eg. a bar
3.7 The algorithm is concurrent	Must	Multiple instances of the matching algorithm is started while the recording gets longer and longer

4. User Interface		
4.1 Similar to real Shazam app	Must	A single button to start recording and identification process.
4.2 Relevant song metadata is displayed	Must	Song name, artist name, album name and release year is displayed
4.3 Clear error messages	Must	When a song is not found, it should be conveyed to the user
4.4 Accessibility: more languages	Want	The application can display both english and swedish text
4.5 Accessibility: blind mode	Want	The application can be started in blind mode, where information is conveyed through audio instead of text
4.6 Reusability	Must	After a song has been found (or not), the user can try again without having to restart the application
4.7 Platform	Must	The application works on both windows and mac operative systems
5. Software testing		
5.1 Unit test for maximum filter function	Must	Verify that the filter works on a small manually crafted input/output sample
5.2 Functional test for maximum filter function	Must	The result is verified against a slower but confident implementation of the same function
5.3 Unit test for peak pairing function	Must	Verify that the pairing works on a small manually crafted input/output sample
5.4 Integration test with database	Must	Verify the connection and that all the queries to the database works without errors
5.5 End to end test	Must	Verify that a song taken from the database, played in front of the computer, can be identified.

1.5 Boundaries and changes from project plan

Throughout the project, the team has encountered various unforeseen difficulties that have resulted in the need to update both the project plan and scope. In the planning report, it was stated that artificial intelligence and machine learning would be explored as an alternative to the Shazam algorithm. This would have been an interesting approach to implement and compare with the initial algorithm. However, as the project progressed and became more complex, the group decided to fully focus on the initial signal processing approach instead of delivering two semi-functioning

applications due to the time constraint.

To keep the project within its time limit, and also make sure there was no issues with data storage, the following boundaries were assigned:

- The project only needs to be stored locally on the team's computers to avoid any copyright inconveniences associated with sharing music information. This means that no web API needs to be developed.
- The real Shazam service is today a mobile application. Since the focus of the project is to recreate the mechanics of the identification capabilities, rather than the user interface, the application's target operative system is PC and Mac. We do not have to create a mobile or web application of any sort.
- Even though the goal is to match Shazam's performance in terms of identifying songs quickly, there is no need to match Shazam in terms of the number of songs that could be identified. To be able to identify slightly above 100 songs is enough for this thesis.
- During the project it was also assumed that all noise affects the performance equally. This is because it makes testing easier and some of the results easier to decipher.

In this section, the project has been presented, as well as goals and boundaries. The next section gives a description of the theory needed to understand the Shazam algorithm itself and the rest of this report.

2

THEORY

The algorithm outlined in this section is based on a scientific paper authored by one of the founders of Shazam, Avery Wang [3]. The core concept of the Shazam algorithm revolves around processing audio tracks, also known as sampling, into what is named as fingerprints. These fingerprints contain a small amount of unique data that effectively represents a part of a song. By utilizing fingerprints, comparisons between songs become more efficient, as opposed to analyzing entire tracks. The resulting fingerprints are assembled in a database, serving as points of reference during the identification process. A notable feature of the Shazam algorithm is its robustness against various sources of noise and filtering during sampling. Even in scenarios where a song is recorded in a noisy environment using a low-quality microphone, the resulting fingerprint is expected to be close enough to resemble the corresponding fingerprint stored in the database. Once recorded, these fingerprints are compared with those in the database, and the song that shows the closest match is then returned by the algorithm. This means that the algorithm can only detect songs that earlier have been processed into fingerprints and then stored in the database. A song that the algorithm has never seen before cannot be detected, in which case the algorithm either outputs the most similar known song, or doesn't find any resemblance at all.

For a more detailed examination of the algorithm, it can be deconstructed into the following key components: **Spectrogram generation**, **Spectrogram maxima**, **Hashing the maxima**, and **Matching**. The former three components regard the construction of the fingerprints. This process is used both when analyzing clean songs, to establish a reference database, and song identification from user recordings. The final component is used to determine how similar fingerprints from two songs are, enabling the algorithm to determine which song best matches the recording. See figure 2.1 for a visual description of the algorithm steps.

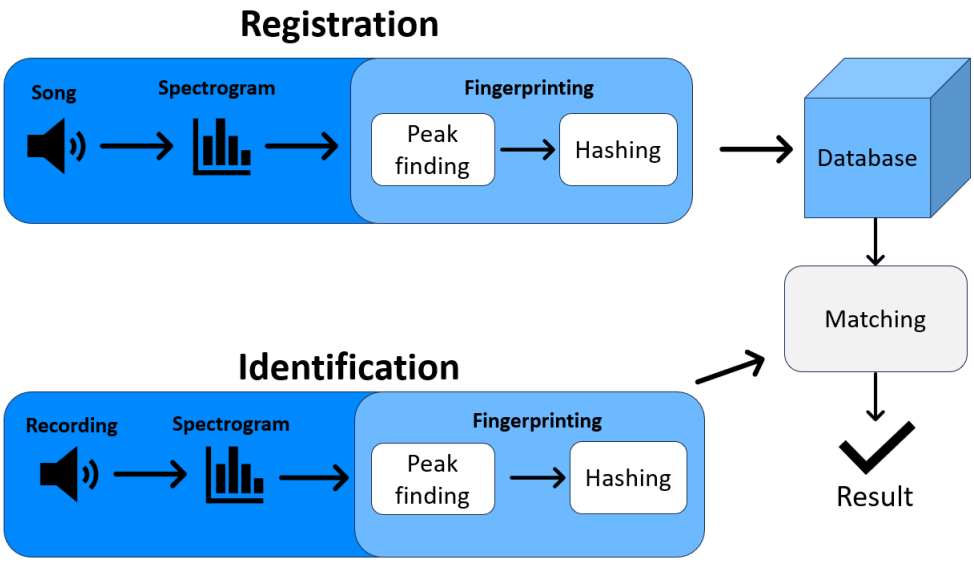


Figure 2.1: *A block diagram for a visual interpretation of how the process of the algorithm works for both registered songs in the database and song identification.*

In the following sections, in-depth explanations of each of the four key algorithm components are presented. Then, section 2.5 introduces theory relevant for comparing loudness between wanted and unwanted elements of sound recordings.

2.1 Spectrogram

A *spectrogram* is a visual depiction illustrating the frequency spectrum of a signal over time. To generate a spectrogram an acoustic signal is processed by transforming it to the frequency domain using *Short-Time Fourier transform* (STFT). Essentially what the Short-Time Fourier Transform does is splitting the time domain signal into smaller sections of equal lengths, then taking the Fourier transform of each of them separately [7]. The resulting Fourier transformed sections are lined up vertically next to each other to create a two-dimensional image. Since the time-domain signal is split into separate sections, frequency components of one section of the song will not spread to all other sections when taking the STFT. Compare this to the regular Fourier transform, where a frequency component will be included in the frequency domain signal regardless of when it occurred in the time domain signal. The following definition shows how the discrete STFT transformation depends on two variables as inputs corresponding with time and frequency, and generates an output that represents how big the amplitude is in the original signal at that time and frequency [7].

$$\mathrm{STFT}\{x[n]\}(m,\omega) \equiv X(m,\omega) = \sum_{n=-\infty}^{\infty} x[n]w[n-m]e^{-i\omega n} \tag{2.1}$$

The discrete function $x[n]$ represents the time domain signal undergoing transformation, while $w[n]$ is a window function centered around zero. The inputs m and ω are the desired time (in number of samples) and frequency, respectively, at which the STFT is applied. The window function is responsible for splitting the time-domain signal into smaller sections. To explain what happens in the rightmost part of equation 2.1, the input signal $x[n]$ is multiplied element-wise by a window function $w[n]$, which is shifted m steps to the right. The result is an isolated section of $x[n]$ at $n = m$, where all other parts of $x[n]$ have become zero. See figure 2.2 for a visual explanation of this process. Now that a short section of $x[n]$ has been isolated, it is Fourier transformed. This corresponds to the $\sum$ and $e^{-i\omega n}$ part of the equation. $X(m, \omega)$ is essentially the discrete Fourier transform of $x[n]w[n-m]$ (**Note:** the frequency ω and window function w are different).

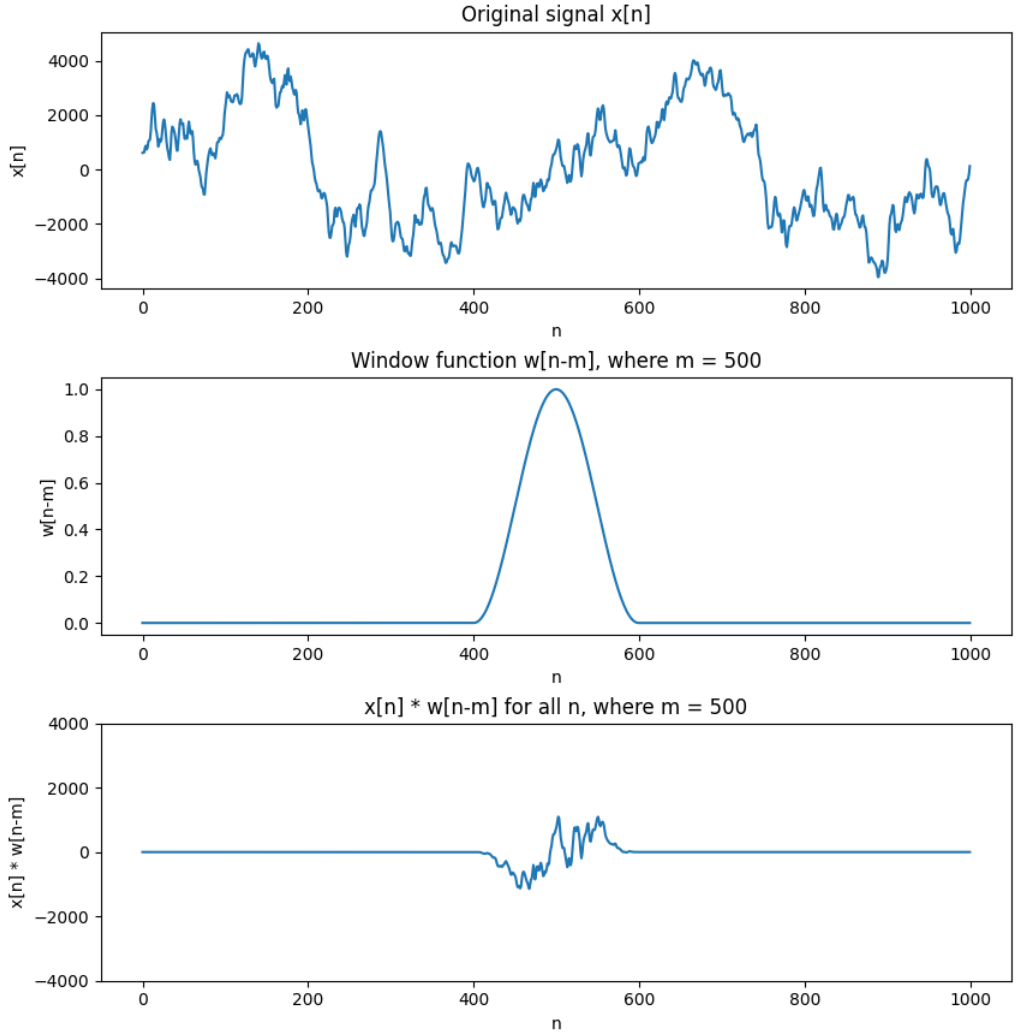


Figure 2.2: *An example audio signal $x[n]$ with 1000 samples is shown in the first plot. The second plot shows a shifted window function centered at $n = m = 500$. The third plot shows how the window function has been applied by multiplying each sample in the first two plots pairwise with each other. The result, $x[n]w[n-m]$, is an isolated part of $x[n]$ that is the input to the Fourier transform step in the STFT.*

The window function directly defines the characteristics of all sections. For example, notice that if the window function is symmetric around $n = 0$, each isolated section will be centered around $n = m$. Also, the width of the window function directly affects the width of each section. Depending on how many sections we decide to use and the width of each section, they may overlap with each other. If the window function is a symmetric gate function of three samples in width, for example, and we take the STFT on each sample in the input signal, then there will be one section centered at each sample. Each section will overlap with neighboring sections, since each section extends one sample forward and backward in time. Overlap isn't necessarily unwanted. No overlap could lead to elements in the input signal that lie just between sections to be filtered out from the STFT result. Too much overlap would cause less precision in the result, since the frequency components would be smeared out in time. A window function that works well on almost all cases is the Hann window, see figure 2.3.

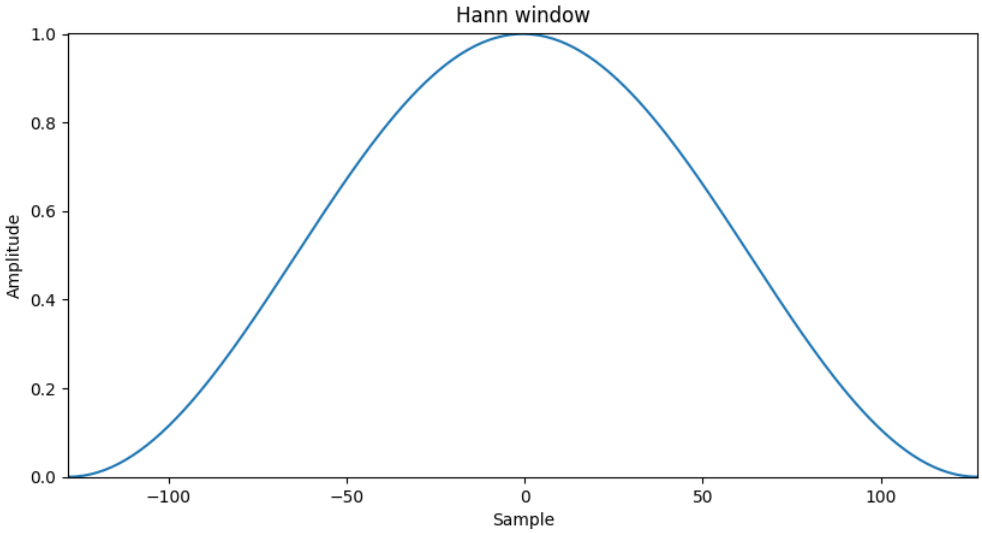


Figure 2.3: *A plot of a Hann window of width 256 samples. This would create a section with the width of 256 samples in the STFT.*

Applying the STFT on a wide range of combinations between m and ω, then plotting each result as a pixel on a two-dimensional plot, where the axes are m and ω, we can visualize the spectrogram of the input signal $x[n]$, see figure 2.4. In short, the STFT-algorithm reconstructs the sound wave, described by the sound intensity over time, into a new signal representing sound intensity over both frequency and time. This is sometimes favorable as key frequency components in the original signal can be clearly seen in the spectrogram, and equally as important, at what time the components occurred. Also, noise usually consists of a broad range of frequencies, which means that the addition of noise in an audio recording is, to put it simply, that the output of the STFT-transformed recording is increased by a constant amount for all inputs m and ω. Looking at the spectrogram, where the brightness of each pixel corresponds to the output of the STFT at specific values of m and ω, the result of added noise is that the entire image becomes brighter. How much brighter

the image becomes depends on the strength of the noise. If the noise isn't large enough, the characteristics of the spectrogram would be roughly the same with and without added noise to the input signal. The difference is that the spectrogram of the input signal with added noise would be slightly brighter than the other one. In the time domain however, the addition of noise would make the signal more or less unrecognizable, especially to a computer. This is why the STFT is used in the Shazam algorithm.

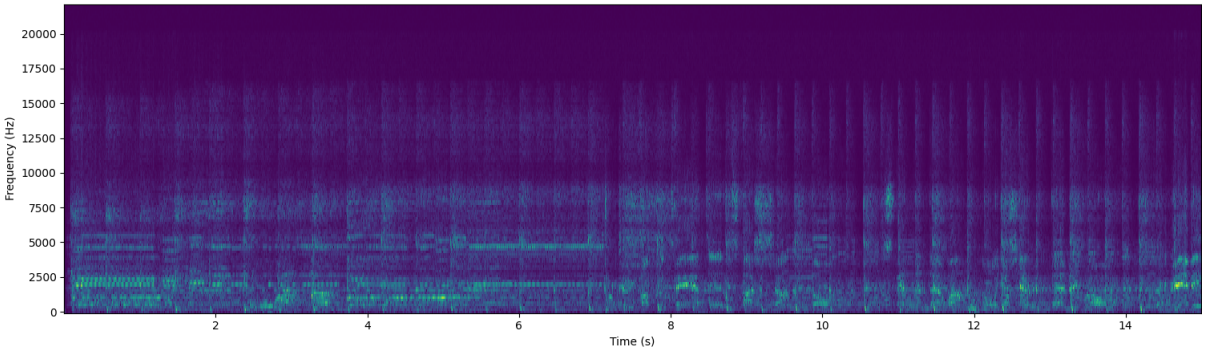


Figure 2.4: *An example spectrogram of a song. In this image, frequency is on the Y-axis (ω), and time is on the X-axis (m). The pixel intensity is on the Z-axis, and directly corresponds with the amplitude of the song at frequency ω and time m, or in other words, the STFT output $X(m, \omega)$. In this spectrogram, brighter colors indicate larger STFT outputs. The range of values used for ω in this spectrogram are 128 values taken linearly between o and 20,000. Each section is 256 samples wide, and the overlap between each pair of neighboring sections is 32 samples.*

What characterizes most musical pieces is their combination of pronounced frequencies, what we humans perceive as distinct tones [8]. The pronounced frequencies are still prominent in the frequency domain even with the addition of noise, so long as the noise doesn't drown out them. With the spectrogram, it is easy to spot these pronounced frequencies, and when they occurred. Extracting these pronounced parts of the spectrogram is a useful step in the process of identifying a song from a noisy recording. In the next section, the pixels in the spectrogram that are brighter than all their neighbors are extracted via a maximum filter algorithm.

2.2 Spectrogram maxima

The local maxima in the spectrogram, or *peaks*, as they will be referred to as hereinafter, have the highest chance of staying unaffected by noise and other types of disturbances. A maximum-filter is used to separate these peaks from the rest of the values in the spectrogram. The maximum-filter consists of two steps. To explain it visually, the first step involves scanning each pixel in the spectrogram and copying the value of the strongest nearby pixel in a square of given side length R to it. This will result in a sort of blur effect where bright parts seem to leak onto nearby pixels, see figure 2.5.

The second step is to compare this resulting image with the original spectrogram

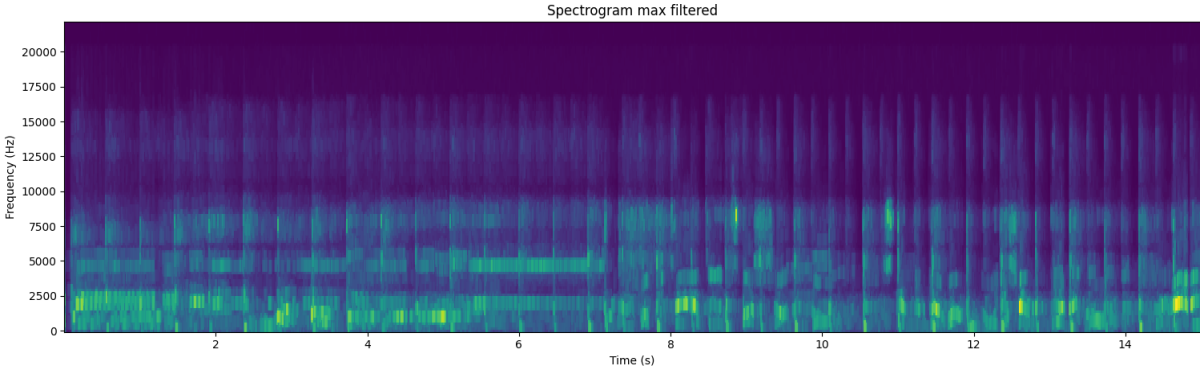


Figure 2.5: *The first step of the maximum filter step applied on the same spectrogram filter as in figure 2.4*

image. Each pixel in one of the images is compared to the corresponding pixel in the other image. When pixels differ in brightness, they become black. When pixels have equal brightness values, they become white. This results in a new image which we call a *constellation map*, where the peaks of the spectrogram are visible (usually not noise), and everything else (usually noise) is left out. See figure 2.6 for an example plot of a constellation map. In theory, if a maximum filter was applied on both the spectrogram of a clean audio signal containing music, and a spectrogram of a the same audio with small amounts of noise added, the resulting constellation map would be the same. The constellation map not only filters out noise, but it also represents key parts of a piece of audio with very little data.

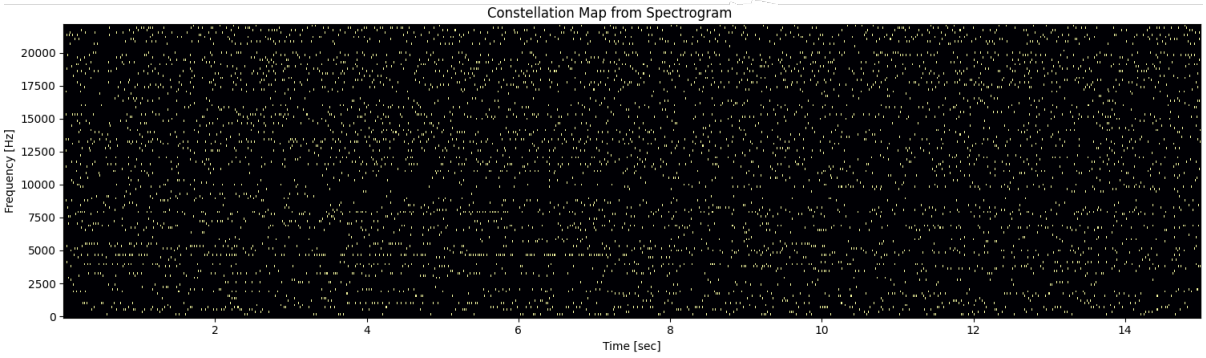


Figure 2.6: *A constellation map of a real song. The name comes from the resemblance of the night sky. Note that each peak (bright pixel) corresponds to a certain time and frequency.*

Note that the value of R, of which the maximum filter is applied using, directly affects the density of peaks in the constellation maps. Assuming two nearby pixels in the spectrogram doesn't have exactly equal values, only one peak will be extracted in each square of side length of R pixels (containing R^2 pixels). This means that the number of expected peaks in a constellation map consisting of S pixels in total is $\frac{S}{R^2}$. The number of peaks that are generated in this step is inversely proportional to R^2. This will be useful to keep in mind in section 3.1.5, where it is discussed that the performance of the algorithm greatly depends on the number of peaks.

Now that the most pronounced parts of the recording are extracted, the next sec-

tion goes through the step of compiling the constellation map into easily comparable fingerprints. The fingerprints will be useful when trying to identify a song, later explained in section 2.4.

2.3 Hashing the maxima

The constellation map generated from the last step is used as the input to this next step, which is hashing the maximum values. In this step the fingerprints are extracted from the constellation map. In theory, to identify a song from a recording, we would now compare the constellation map of the recording with the constellation maps of a set of songs that are candidates. All there is to do is find which of the candidates line up best with the recording when looking at their constellation maps. Algorithmically this process would be really slow, since it would require to try lining up the spectrogram of the recording at all possible discrete time steps in all the recordings. Then the algorithm would have to count the number of peaks lining up.

A way to speed this up would be to create a table that shows, for each frequency, which songs include a peak in that frequency. When trying to match a recording to the entire set of songs, we could look at the peaks in the constellation map. Then all the frequencies of these peaks could be noted, and then the table can be used to quickly find a subset of candidate songs that is known to certainly contain the song that's trying to be identified. This would mean that only a subset of the constellation maps need to be compared more precisely, speeding up the process. The issue here is that since the frequency axis is discrete, there is only a finite number of frequencies in the spectrogram which can hold peaks. Furthermore, songs usually contain many different frequencies, so the subset of candidates would be marginally smaller than the entire set of songs. The speedup would in practice be insignificant.

An insight that builds on the above idea is to put pairs of peaks into the table. It is much more uncommon for songs to contain a specific combination of two peaks, than to contain just one a peak at a specific frequency. The subset of candidate songs that contain a certain pair of peaks is greatly reduced in size, assuming peaks occur rather uniformly across the spectrogram. For a given pair of peaks, we represent it in the table by the two frequencies belonging to respective peak, as well as the difference in time between them. These three numbers can be hashed together to be used in a hash table, see figure 2.7. We call each of these hashes *fingerprints*. Throughout the rest of the study we will use the terms fingerprints and hashes almost interchangeably. The table now maps fingerprints to the set of songs that contain that fingerprint.

The pairing process is carried out like this: for each peak p_i in the constellation map, find F other nearby peaks if possible. For each of those nearby peaks p_j, create a pair consisting of peaks p_i and p_j. To avoid creating duplicate pairs, only look for F

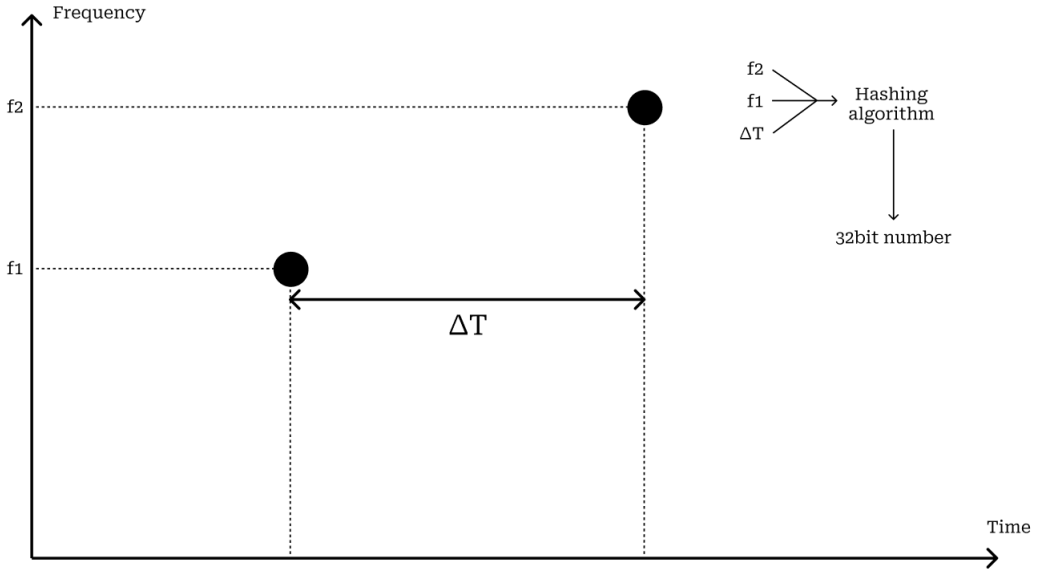


Figure 2.7: *A close-up of a pair of peaks from a constellation map. The hash value is generated using a hashing algorithm that takes the frequencies of the peaks, and the difference in time between them, as input. The result can be a 32-bit number, which would allow a few billion different fingerprints to be added to the hash table.*

nearby peaks to the right of p_i. Note that if the constellation map had P peaks, one would extract at most $F * P$ peak pairs. Also note that each peak can be included in more than one pair. See a visual example of how nearby points are used to create pairs in figure 2.8

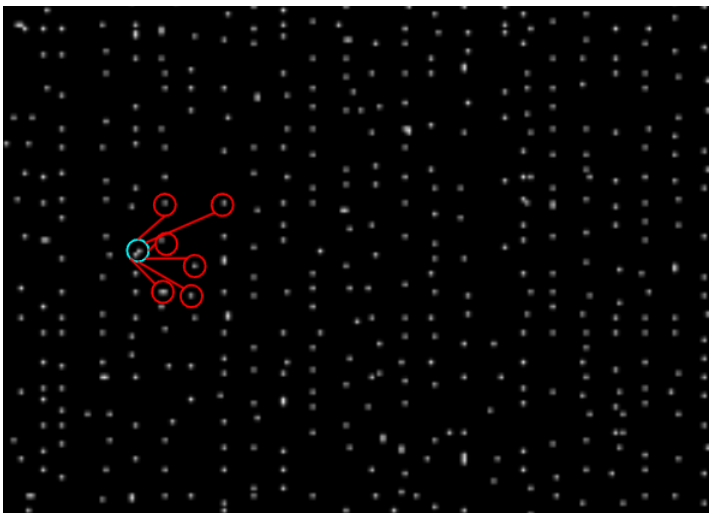


Figure 2.8: *A point P_i circled in blue, and $F = 6$ nearby points circled in red. Each red line between two points represent a pair. Note that this figure is a zoomed in version of a constellation map similar to figure 2.6 to improve visibility*

When trying to match a recording to the entire set of songs, we extract all peak pairs from the recording. The peak pairs are all hashed into single integers. Using the table and indexing it with the extracted fingerprints, a small subset of song

candidates can quickly be found. Combining peaks pairwise is not only useful for finding a small subset of song candidates, but also for the way we will compare the recording to these candidates in the matching process, described in the next section.

2.4 Matching

When a sample has been recorded and the song candidates have been found, the next step is to determine which candidate matches the recording best. For each peak pair in the constellation map from the recording, the table is used to find the set of songs that also contain that peak pair, as well as the location of those peak pairs in those songs. All the peak pairs are grouped by the song they existed in. For each song, a similarity score will be calculated. It is done by comparing the difference in time between each matching peak pair in the recording and reference song. Each matching peak pair is then ordered into buckets based on the time difference. The score for the song candidate is the amount of matches in the largest bucket. The song that is the most similar is the one with the highest similarity score and should be presented as the best match by the algorithm.

When comparing a small sample of a song with a full length clean reference version of the same song, imagine the recording's constellation map sliding across the reference's constellation map. At some point, the peaks - and therefore also the peak pairs - should line up perfectly. Now imagine that we connect the lined up peak pairs from both constellation maps with rubber bands. If we continue sliding the recording's constellation map over the reference's again, the rubber bands will stretch, but they will always have the same length as each other. This is the main idea behind the matching algorithm. When finding a lot of equal distances between the matches, it is certain that there will be many peaks lining up somewhere during a sliding process described earlier. This suggests that the candidate is very similar to the recording. On the other hand, when the distances between the matches differ a lot in length, the peaks will not line up nicely anywhere along the reference constellation map. This suggests that the candidate is dissimilar to the recording. With this algorithm, we take a shortcut and skip the entire "sliding" process, which would be very expensive to simulate. To summarize, it is enough to find many matching peak pairs with the same distance in time in respect to each other in the constellation maps to know that the two corresponding songs are similar.

2.5 Signal to noise ratio

Since the recordings that the algorithm works on contains background noise, it is useful to quantify how strong the noise is compared to the music. The concept

of Signal-to-noise ratio (SNR) can be used for this purpose. The SNR is a ratio between the average power level of the signal and the average power level of noise [9]. Assuming the signal and the noise are separated into two functions of time, the SNR can be calculated as follows:

$$\text{SNR} = \frac{P_{signal}}{P_{noise}} = \left(\frac{A_{signal}}{A_{noise}}\right)^2 \tag{2.2}$$

Where A is the root mean square (RMS), which is calculated using the following formula on a discrete signal a with N samples:

$$\text{A} = \sqrt{\frac{\sum_{n=1}^{N} a[n]^2}{N}} \tag{2.3}$$

The RMS is an indicator for how loud a signal on average is, which is useful for calculating the SNR. The SNR is often described in decibels (dB) and can thus be defined as:

$$\text{SNR}_{dB} = 10 \cdot \log_{10}\left[\left(\frac{A_{signal}}{A_{noise}}\right)^2\right] \tag{2.4}$$

Applying a law of logarithms, the square inside the brackets can be moved out as a factor outside the logarithm:

$$\text{SNR}_{dB} = 20 \cdot \log_{10}\left(\frac{A_{signal}}{A_{noise}}\right) \tag{2.5}$$

This value tells us how many decibels louder the signal is compared to the noise. To prove this, we use equation 2.4 and the first part of 2.2 and apply another law of logarithms.

$$\text{SNR}_{dB} = 10 \cdot \log_{10}\left(\frac{P_{signal}}{P_{noise}}\right) = 10 \cdot \log_{10}(P_{signal}) - 10 \cdot \log_{10}(P_{noise}) \tag{2.6}$$

Then, using the definition for the power of a signal in decibels:

$$P_{dB} = 10 \cdot \log_{10}(P) \tag{2.7}$$

We find an alternate definition for the SNR in decibels:

$$\mathrm{SNR}_{dB} = P_{signal,dB} - P_{noise,dB} \tag{2.8}$$

Thus, for instance, when the signal is 60 dB and the background noise is 40 dB, the SNR is then 20 dB. When the signal and noise have the same loudness the SNR is 0 dB.

3

METHOD

The following sections outline the approach taken on how to use acoustic fingerprinting for music identification. Firstly, the structure of the algorithm is described in section 3.1. Thereafter, in section 3.2, the choice of data management is explained, highlighting the use of a database. Lastly, the methods for testing and verification of the algorithm's success rate is elaborated in section 3.3.

3.1 Implementation and development of the algorithm

The group utilized a modular and object-oriented programming pattern to enhance code readability and facilitate development in later stages of the project. This also made code-reviews easier and ensured a higher standard in the code. The entire algorithm is created using Python because of its easy use and prebuilt libraries for signal analysis.

After researching different methods to recreate the Shazam algorithm, the group decided to use classic signal processing first and foremost. Short time Fourier transform was applied to all the songs in our music library to create spectrograms. The biggest amplitudes were then identified and saved in pairs as fingerprints in our database. Then, for every recording, the same process was repeated to create new fingerprints. These were then to be matched against the fingerprints from known songs in our database.

The implementation of the Shazam recreation could be separated into a few subtasks. To be able to run the algorithm in the first place, a user interface was needed. Another task was to implement the code for the recorder, which would be responsible for capturing audio data and providing it to other parts of the code in a readable format. A further subtask was to set up the database, and the code that is responsible for communicating with it effectively. Then there is the algorithm itself, containing the implementation of the concepts explained under section 2. Last but not least, a main program is needed to integrate the different project components together and make sure that they communicate well with each other.

3.1.1 The audio recorder

The Recorder class utilizes a Python library called **sounddevice**. When a Recorder object is instantiated, a recording channel is opened on the computer and the program begins capturing audio on the default microphone device. The audio is saved into a memory buffer which is regularly read from and copied into a Python list by a separate thread in the program. With this library, it's easy to access microphone metadata such as the recording sample rate, which is important for interpreting the audio data.

Since the task was to recreate the Shazam algorithm, which can find a song in real-time while it's still recording, we needed to implement the Recorder class in such a way that makes two things sure. The first is that we had to be able to read from the buffer at any time, even when the audio capturing is still going on. The second is that the audio capturing can continue even though the audio data has been accessed. These problems were solved simply by using multiple threads that each have their own responsibility. One thread extends the audio data list from the channel buffer. Another reads from it.

The final implementation of the Recorder class has the following API. The method **start()** starts the audio capturing and begins writing audio data in the background. The method is non-blocking. The method **stop()** stops the audio capturing. The class has an attribute called **waveform**, which is a list containing the audio data as an array of integers describing the recorded signal in time-domain. The method **get_waveform()** returns the waveform attribute along with the sample rate of the recording as an integer.

3.1.2 Useful dataclasses

Before the key parts of the algorithm implementations are discussed, it can be useful to understand some other classes that doesn't have much functionality, but is rather convenient still. It is common to utilize the built in Python library **dataclasses** to simplify the writing of these classes. With dataclasses, Python classes can be written like a *struct* would be written in the programming language **C**. By decorating a Python class with "@dataclass", an initializer is automatically created for the class attributes, meaning no methods have to be written (although they can be). In total, three dataclasses were used in the project.

The first one regards peaks. It is useful to represent a peak as a type of its own. The dataclass **Peak** contains two attributes.

- *time* (integer): describes the time at which the peak in a constellation map exists.

- *freq* (integer): describes the frequency of the peak.

The second dataclass regards peak pairs. The **PeakPair** class contains four attributes and one method.

- *timestamp* (integer): the sample index at which the leftmost of the two peaks occur in a constellation map.
- *freq1* (integer): the frequency of the leftmost peak.
- *freq2* (integer): the frequency of the rightmost peak.
- *delta_time* (integer): the difference in time between the two peaks, counted as number of samples between them.
- *__hash__()* (integer): computes the hash/fingerprint for the **PeakPair**, using *freq1*, *freq2* and *delta_time* as inputs.

```
prime1 = 52711
prime2 = 1000000007
hash = int(freq1 + freq2 * prime1 + delta_time
    * prime1 * prime1) % prime2
```

The third and last dataclass regards song metadata, and is called **SongInfo**. It is used to represent songs when inserting and retrieving songs from the database. It contains four pretty self-explanatory attributes:

- *name* (str): the name of the song.
- *artist* (str): the artist's name.
- *album* (str): the song album's name.
- *release_date* (str): the song's release date, in any text format.

3.1.3 Connection to a fingerprint database

The **Connector** class works as an interface between the rest of the code and the database. It plays an important role of the program since it is responsible for configuring and managing the database tables. It handles inserts, retrieval and manipulation of data. The tables are configured to contain relevant primary keys, foreign keys and datatypes for all attributes. More about the details of the tables and their

attributes will be discussed in section 3.2, but a general overview is that tables for fingerprints and song metadata exist in the database.

The class incorporates functions for both retrieving matching fingerprints and adding new fingerprints to the database. This paragraph will explain the API of the **Connector** class and will go over the five public methods. The first method, **insert_song()**, takes a **SongInfo** object, and a list of **PeakPair** objects as arguments. The method is responsible for adding the song to the database, populating it with the song metadata, as well as inserting the hashes of all peak pairs. This is done with two simple *INSERT* SQL queries. The **insert_song()** method also accounts for songs that already exists in the database. This means that if the song to be added and its hashes already exists, they won't be added twice. The second public method **get_matching_hashes()** has an integer parameter *hash_value* and is responsible for retrieving all fingerprints matching *hash_value*. Remember that a hash in this context is synonymous with the term *fingerprint*. The SQL query selects all fingerprints that matches the *hash_value*, after which the function calling the query returns them in a list of tuple objects. Then there are some extra methods. The **delete_all_songs()** method completely wipes the database from all songs and fingerprints. This is useful during development when we occasionally want to reset the working environment. Furthermore, there is a method **song_exists()** that takes a **SongInfo** object as the only parameter, and returns a boolean, indicating whether or not the song described by the **SongInfo** object currently exists in the database or not. Lastly, the public method **get_song_info()** takes two strings as parameters, namely the artist name and song name. The function fetches the rest of the song metadata from the database and returns it as a **SongInfo** object.

3.1.4 Creation of spectrogram and peaks

The Spectrogram class was created using the pre-built libraries *Scipy* and *Numpy*. Both of these libraries are external Python libraries, though very popular. This enabled quick generation of spectrograms for songs and recordings.

```
frequencies, times, spectrogram =
scipy.signal.spectrogram(waveform, sample_rate, window='hann')
```

The **spectrogram()** function takes the waveform, its sample rate and the selected window function as arguments. The Hann window was chosen simply because it was symmetric and some of the group members had used it before. The resulting *frequencies* variable is a list of integers describing which frequencies each row in the spectrogram corresponds to. The *times* variable is a list of floating point numbers describing the timestamp in seconds of each column in the spectrogram. The *spectrogram* variable itself is a two-dimensional Numpy array of floats, which describes the spectrogram itself. The spectrogram is calculated in the initializer method for the **Spectrogram** class. With the Scipy spectrogram function, you can choose the

width of each section (window size) and the overlap between sections. We used the default values, where each section was 256 samples in width, and the overlap between each pair of neighboring sections was 32 samples.

The public method **get_peaks()** takes the already calculated spectrogram, stored in the object itself, and applies a maximum filter to it. This is also done using a function **ndimage.maximum_filter()** from the Scipy library. This function takes the two-dimensional spectrogram Numpy array as an argument, along with an integer argument *radius*. The radius of the maximum filter is explained in section 2.2 (referred to as R), and has a direct impact on the number of peaks generated. Initially, a conservative value was chosen for the radius, with a balance between execution time and successful generated answers. The maximum filter returns a new two-dimensional *Numpy* array of the maximum filtered spectrogram. Peaks are found by comparing the spectrogram object with the maximum-filtered spectrogram and retaining the values where equality exists between them. This can be done quite elegantly using the equality operator between two Numpy multidimensional arrays.

```
constellation_map = self.spectrogram == maximums
```

The resulting *constellation_map* variable is a Numpy array of zeroes and ones. The ones indicate equality. After this, we simply loop through all these ones and zeroes. Whenever a one is found its time and frequency are used to create a **Peak** object, which is then appended to a list. When all peaks have been added to the list, it is returned from the **get_peaks()** method.

3.1.5 Paring and hashing of peaks

To extract the fingerprints from the list of peaks, we need to create pairings between them. We only want to create pairings between peaks that are nearby to the right. A first naive approach was to loop through all P peaks, then for all of them, loop through all other $P-1$ peaks. For each pair of peaks in the nested loop, the distance between them is calculated. The closest F peaks are then used to create pairings with. This approach, looping through on the order of P peaks twice in a nested loop fashion, has an algorithmic time complexity of $O(P^2)$. When trying different maximum filter radii, we could see that P lied between around 10,000 and 500,000 peaks. A quadratic time complexity is not efficient enough to process the list of peaks, so we needed something better.

A first step was to realize that we could sort the list of all peaks by their *time*-values. When looping over each peak P_i, nearby peaks P_j to the right of P_i now always had the property that $i < j$. The second loop doesn't need to look at values $j <= i$. This certainly speeds up the process, since approximately half the number

of comparisons vanish. However, we are looking for a speedup factor of about 1000x, not 2x.

A second insight was that once we have found F peaks for peak P_i to pair up with, we don't need to keep looking for more. Since the list of peaks are sorted in time, these F nearby peaks will be found rather quickly. Breaking the inner loop once F peaks have been found meant that we, in the inner loop, only had to iterate through all peaks between P_i to P_{i+K}, where K in fact was a relatively small number compared to the entire number of peaks. This results in a speedup factor of $\frac{P}{K}$ which ended up being a lot closer to 1000.

Even if the subset of nearby peaks had been narrowed down, we hadn't yet decided on the best way to find the F closest peaks. Paradoxically, the subset of nearby peaks wouldn't be known until F nearby peaks had been found. A good enough approach was to define a bounding box just to the right of each peak. We considered peaks in this box being nearby, and peaks outside were considered not being nearby. When looping through peaks P_j starting from $j = i+1$, and we found a peak inside P_i's box, we created a pair using it. This approach made sure we would find F of the leftmost peaks inside each box. The box was made sure to be sufficiently large not to contain less than F peaks regardless of the density of the constellation map. To find a reasonable box size, the density of the constellation map (or in the more data efficient format, the list of peaks) was first calculated. Then a simple calculation was done to find a box size that statistically would fit at least F peaks.

Although it wasn't necessary, the peak matching algorithm could have been sped up more. Note that although the tricks mentioned above sped up the algorithm by constant factors. The time complexity stayed quadratic. There is for example a data structure called a **K-d tree** to query the F nearest neighbours in an n-dimensional space in logarithmic time complexity. In our case, this data structure could be applied with $n = 2$, and would reduce the time complexity to $O(P * log(P))$.

3.1.6 Algorithm class

The spectrogram generation, extraction of peaks, peak pairing, and matching process are all managed by a single class responsible for the signal processing algorithm itself. This class is called **ShazamRecreation** and communicates with other classes like **Spectrogram** and **Connector**. This class has two major public methods. The first one is **add_song()**, which takes a **SongInfo** object, a waveform and its sample rate as arguments. This method calls other pieces of code to find and pair together peaks into fingerprints, then sends these fingerprints together with the **SongInfo** object as inputs to **Connector.insert_song()**, which in turn adds the computed fingerprints and the song to the database. The other major public method, **find_song()**, takes a waveform and its sample rate as inputs, finds the

best matching song, and returns a **SongInfo** object for the match as well as the timestamp at which the match was found. This method also firstly calls other pieces of code to generate fingerprints, but this time for a recording instead of a clean song. Nevertheless, the fingerprint extraction process is the same. Then the method calls **Connector.get_matching_hashes()** to find the candidate songs, and subsequently calculates the score for each of those songs. This is done by putting fingerprint distances into bins and finding the largest bin for each song, more closely described under section 2.4. The song metadata, encapsulated in a **SongInfo** object, the timestamp at which the match was found, and the *certainty score* are together returned from the method as a tuple object. The certainty score is explained thoroughly in the next section.

The timestamp can be known thanks to the fact that each **PeakPair** object stores the timestamp at which it occurs. When matching fingerprints, the time difference between them is the same as the time in the song at which the recording is most likely taken. One minor detail is that the difference in time has to be converted from sample index to seconds. This can be done easily by dividing by the spectrogram sample rate, which is calculated as the sample rate of the initial recording divided by its waveform length (sample count), then multiplied by the number of columns in the spectrogram.

3.1.7 Certainty score

As explained in section 2.4 when matching a recording to candidates, each candidate song gets a score based on the similarity between it and the recorded sample. These scores are the basis on which the song prediction is made. It is relevant to know how confident a prediction is, meaning how big the score for the predicted song is compared to the other song scores. If multiple songs get similar score levels, it indicates that the algorithm isn't very confident about which of them is the best choice. Oppositely, if one song gets a score hundred times larger than all other songs, it is very likely that the song is in fact the correct candidate.

For the purpose of knowing how confident a prediction is, we have invented what we call the *certainty score*. The certainty score is defined as the ratio between the scores of the two highest scoring songs. A certainty score of one means that two songs had equal scores, and both were equally good candidates for being chosen as the answer. This is the worst certainty score possible, since the algorithm still has multiple candidates even after the matching process. A certainty score larger than, say, three, would indicate a quite good candidate that most likely is correct.

3.1.8 User interface

Given that the project's primary focus revolves around recreating the Shazam algorithm, our group opted to craft a user-friendly interface that offers simplicity and functionality, rather than anything fancy or complex. This approach aims to make the application easier to use, test and display results.

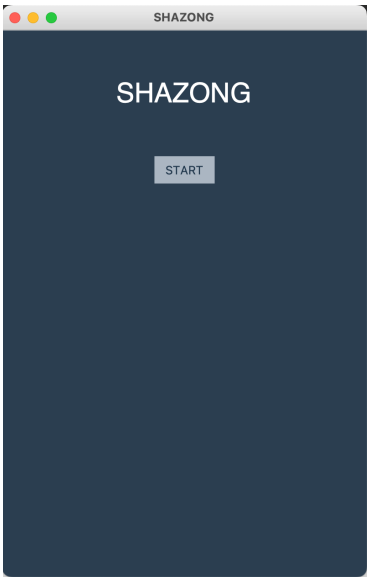
To get started with the interface development process, we utilized the Python library Tkinter. With Tkinter we could easily spawn a window on-screen and draw labels and buttons inside it. The buttons could be hooked up to functions that were called when they were pressed by the user. Although the Tkinter library offers minimal design choices, it was perfect for our needs.

From the beginning, we thought that it would be easiest to have a start and stop button to control the audio capturing process. The initial user interface contained these two buttons, a blue background, and a title text. The issue was that we hadn't yet figured out how to make the recorder non-blocking, meaning we couldn't stop the recording using the stop button. In fact the whole user interface was completely unresponsive as we were capturing audio data. This was because the main thread was busy reading the audio channel buffer. To temporarily fix this, we had to specify in beforehand the duration of a recording. When we figured out how to record audio with a background process, we could start using the stop button. The real Shazam application doesn't have a stop button. It only has a start button, because it listens to the surroundings until a song is found, or a timeout occurs. Eventually, our stop button was removed for good as we focused on replicating the real-time identification algorithm which will be explained in section 3.1.9.

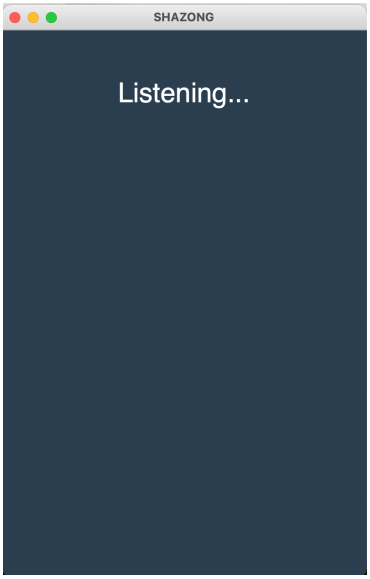
The final interface is still very simple. It contains just the start button and a title (see figure 3.1). When the start button has been pressed by a user, the button disappears and is replaced by a text displaying "listening...". To improve the user interface while the application is listening for sound, we thought about the feedback design principle. By animating the number of dots shown at the end of the "listening..." text, a user won't worry that the program has become unresponsive. Animations usually give a sense that something is still happening, which is a good way to convey feedback to the user. When a song has been found, the "listening..." text is swapped out for the starting title. Song title, artist name, album and year is displayed in a small information box. Also the start button appears so that the user can go again if they want to. In case no song is found, an information message about it is displayed instead of song metadata.

Since we wanted to make the application accessible for more people, we wanted to include at least one more language in the user interface. For every label in the user interface, we made sure to add text for both English and Swedish, depending on the user preference. Also, we implemented a blind mode, which causes each label in the user interface to be read out loud by a computer generated voice. This was done by utilizing the external Python library *gTTS*. Also, the start button would

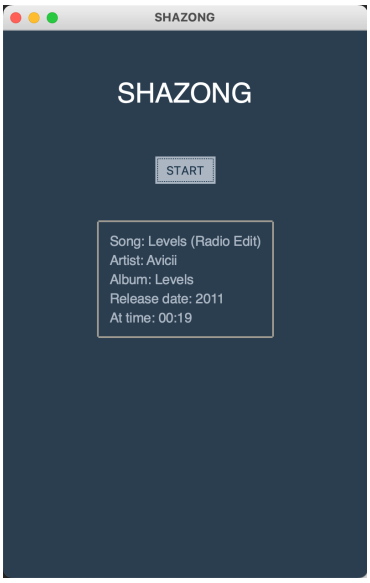
be pressed automatically by the program. This way, a blind user could use the application to identify a song without having to press any button or reading any text.



(a) *When starting the program the following user interface is displayed.*



(b) *During the searching process the following loading screen is displayed.*



(c) *When a match is found the interface shows the matching song with metadata.*

Figure 3.1: *The three different states the user interface can display, namely the starting state, listening state and match found state.*

The final theme used is a built in theme from the Tkinter extension ttkbootstrap. It gives the interface a clean and simplistic look, what is exactly what was sought after. In the next section the driver code will be explained, which integrates the previous parts into the full application.

3.1.9 Main program

The main program, as said before, is responsible for piecing together the different parts of the puzzle. It integrates all the components described in this chapter into a working product. The main program is a Python script, that when run, executes the code that causes a window to appear, showing the graphical application interface to the user. The main program also hooks up the *start* button to the logic that causes the recorder and algorithm to start working.

When the *start* button has been pressed by the user, **Recorder.start()** is first called, starting the microphone recording on the computer. Then, an algorithm is used to find the song in seemingly "real time". The algorithm in itself is just a loop that every three seconds takes the recording and sees if a match is found. More precisely, every third second the so far recorded waveform is copied and sent as an argument into **ShazamRecreation.find_song()**, which tries to find the best match and returns the answer. Running the algorithm takes a few seconds, usually longer than three. This means that when it is once again time to run the algorithm on the now three seconds longer recording, the last algorithm call is probably already running. This causes a problem if **find_song()** is being called on the main thread, because a new algorithm call cannot be made before the old one is complete. The solution is to spawn a new thread to run each algorithm call on. Once a call is complete, the response is saved into a Python object that the main thread can access. The main thread's responsibility is merely to regularly spawn new processes that run the algorithm on the recording, then wait until one of the processes has been terminated. The first resulting match can then be provided to the user via the graphical interface.

Although this approach could quickly and in seemingly real-time identify songs, they were often incorrect. After some investigation, we could see that the first matching attempt (after three seconds of recording) wasn't very accurate. Presumably the recording matched a lot of incorrect songs due to its small length. It was at this moment the need for the certainty score arose. The first matching attempt usually had a particularly low certainty score, meaning the prediction wasn't likely to be correct. As subsequent attempts were made, the certainty score rose. A decision was made to introduce a certainty threshold, to separate song results based on the algorithm's confidence. If a song result had a certainty score below 1.5, it was ignored and the main process kept waiting for other results. As soon as a song had a certainty score equal to or above 1.5, it was displayed to the user and the matching was considered complete.

When a song has been identified and displayed to the user via the user interface, we wanted to continue playing the song at the timestamp at which the song was found. This would showcase the timestamp feature better, which is supposed to accurately detect where in the song the recording is matched. The same way that text was generated into speech and played to the user in blind mode, the songs could be played to the user as well.

It is unwanted to leave the user waiting for too long when trying to find a song. If a song is particularly hard to find, we want the algorithm to stop after a long enough time. The main process makes sure that if 30 seconds have passed since the user first pressed the start button, all following results are invalid. Also, an information message is shown to the user about the song taking too long to find.

3.2 Data collection, data management and the database

The reason a database is used in the first place is to store the fingerprints for each processed reference song in a permanent place. This way, they can be accessed across application runs without having to be recalculated every time. Since the number of hashes are in the millions with approximately a hundred reference songs, a database management system is used to store the fingerprints and song metadata in a robust and streamlined fashion. We opted to utilize the SQL-based database system **PostgreSQL** for our data storage needs. This decision was influenced by its support for handling asynchronous queries, a crucial requirement for our application as it operates multiple threads concurrently, constantly querying the database. Moreover, PostgreSQL offers straightforward setup and local deployment, alleviating concerns about setting up a server.

The database contains two tables. The first table is for storing the metadata for each song inserted by the algorithm. This table is called *songInfo*, and contains the same pieces of information as the dataclass **SongInfo**. Since we assumed that a song could be uniquely identified using its name and artist, we selected the combination of those two attributes as the primary key for the songInfo table. The other table, called *fingerprints*, contains all hashes generated by the maxima hashing step in the algorithm. Not only are the hashes stored, but also the timestamp at which their corresponding peak pair occurred in the song, and also a reference to songInfo, indicating which song the fingerprint belongs to. Since the song name and artist are the primary keys for the songInfo table, the reference contained both the song name and artist in the fingerprints table as a foreign key.

Initially the only primary key in the fingerprints table were set to be the hashes, since we thought that the hashes would be unique, but this caused problems. A primary key comes with a unique constraint, meaning that there can only exist one row for every value of the primary key. Only allowing unique hashes would defeat the purpose of matching fingerprints, since the identification process depends on finding identical fingerprints. With the hash as the only primary key we weren't able to add identifcal fingerprints, since the primary key has that unique constraint. Consequently it was decided to make all attributes: songName, songArtist, hash and timestamp the set of primary keys. This allows for identical fingerprints from

different songs but at the same timestamps, as well as from the same song but at different timestamps. An advantage of PostGreSQL is that indices are automatically created for the primary keys. An index is a data structure that greatly reduces the search time in database queries. The database ended up having the schema displayed in figure 3.2.

Database schema for fingerprints and songInfo tables.

fingerprints(<u>hash</u>, <u>timestamp</u>, <u>songName</u>, <u>songArtist</u>) (songName, songArtist) → songInfo.(name, artist)	songInfo(<u>name</u>, <u>artist</u>, album, albumcover, releaseDate)

Figure 3.2: *The schema describes the tables in the database, what attributes they have, as well as how the primary and foreign keys are set up. In other words, the schema shows the relations and the relationships between them.*

After adding more and more songs to the database, it housed metadata and fingerprints from 101 songs in the end. The songs stored in the database where taken from already downloaded music files stored on our own computers.

We deliberately refrained from storing the actual songs in the database, firstly due to uncertainty surrounding the legal implications of handling downloaded music files, and secondly due to the fact that the songs are huge in file size.

Utilizing the index for the fingerprints table in the database, lookup times are greatly reduced. The index is essentially a hash table. A hash table allows inserting and retrieving key-value pairs in amortized constant time complexity. The keys inserted into the hash table are the hash values. With most hash tables, their size is dynamically updated not to be too large. When inserting or retrieving values for a key, the modulo operator is applied on the key using the table size. When roughly half the table is filled with key-value pairs, the table is swapped out for a new table of double the size, and all old key-value pairs are reinserted into the new table at new locations. This is also the way Python hash tables (or *dictionaries*) work. This means that the hash table will never be over double the size it needs to be, but most importantly, all of this is handled by the database management system. The only thing one has to care about when using the index, is inserting and retrieving key-value pairs. The hash table is dynamically updated to have the smallest address space possible, while still having amortized constant time complexity $O(1)$. Furthermore, since the database management system constrains the key into the space of the table itself, it doesn't really matter how big our fingerprint hashes are. However, when calculating the fingerprint hash, we still take it modulo a billion to reduce the fingerprint into a number that fits in four bytes. There are multiple ways to create a good hash from the three attributes of a **PeakPair** object. We could concatenate the binary representation of the three values, and the result will always be a fixed number of bits, unique for each unique combination of the three attributes. Or we could hash the three values together using the so called *Polynomial Rolling Hash Function*, which doesn't guarantee uniqueness for each unique attribute combina-

tion, but in practice works just as well. This is because collisions are rare. We implemented the latter approach for the simple reason that we though about that approach first. Which approach we take does not affect neither the time complexity of inserting or finding hashes in the database, nor does it affect the address space in the database hash table. The hash function implementation is presented under section 3.1.2, where each of the three attributes is multiplied with a prime number raised to the power zero, one, and two, respectively. The terms are summed and then taken modulo of. If we were to change any of the three values slightly, the hash value would be different. Similarly, if we were to swap the order of the three input values, the hash value would also be different.

3.3 Testing

To ensure that the program worked as intended, tests were performed. First, tests were conducted to ensure that the algorithm could match clean recordings of songs. Then, noise was added and new tests were conducted. Since almost all real-life applications of the Shazam algorithm involves noisy recordings, we tested the algorithm against a wide variety of noise levels. We tested the application in real environments, but also in a simulated environment where noise was artificially added to clean music. Also, some of the algorithm's hyperparameters were optimized.

3.3.1 Testing without noise

In the requirement specifications for this project (see table 1.1), there are two requirements regarding recognition of songs without noise. To verify these requirements, a Python script was written to generate 101 recordings. Since the requirement was to test without noise, these recordings were taken directly from our music library. For each of the 101 songs, a five seconds long recording from a random part of the song was taken and then inputted to the **ShazamRecreation.find_song()** method automatically. The script then displayed the number of correct identifications out of the 101 trials. The test was performed 20 times on all the 101 songs added to the database. Each time the test was performed, a new five second sample from each song was used. The maximum number of hashes per peak (F) and the radius in the maximum filter (R) were both set to ten in this test.

3.3.2 Real-life noise testing

The real noise tests where conducted simply by testing the algorithm by pressing start in the application, playing music from an external audio source with talking in the background. To vary the noise for the different tests, the talking was done closer

to add more noise and further to add less noise. The distance to the microphone from the music source was the same.

The test was conducted on ten different songs and the result of each test was either a success or a fail (shown in section 4.2). As with the last test, F and R were both set to ten in this test, and no timeout was set during this test.

3.3.3 Computer automated noise testing

A disadvantage of testing the application against real life recordings was firstly that they were slow to do, and secondly, it was hard to maintain constant noise levels across tests. With computer automated tests, hundreds of recordings could be generated in under a second. Also, we could add equal noise levels to all the generated recordings, making sure the tests had similar conditions.

A computer automated test that was performed was to test the algorithm across a wide range of noise levels. For this we utilized the concept of SNR explained in section 2.5. For SNR values from 60dB all the way down to -20dB, the algorithm was stress tested on generated recordings of these SNR values.

To artificially add noise to a clean recording that results in a recording of desired SNR, we rearranged equation 3.1 in the following way. This equation solves for the desired RMS value A_{noise} for the noise signal, given the desired SNR_{dB} value, and given that A_{signal} is computed.

$$\text{SNR}_{dB} = 20 \cdot \log_{10}\left(\frac{A_{signal}}{A_{noise}}\right) \Leftrightarrow A_{noise} = \frac{A_{signal}}{10^{\frac{SNR_{dB}}{20}}} \tag{3.1}$$

To generate noise with a desired RMS value $RMS_{desired}$ (which in our case is equal to A_{noise}), a signal containing random amplitudes was first created, because such a signal is considered noisy. This signal's length exactly matched the clean music signal's length, since they were going to be added together in a later step. The RMS value for this noise signal, $RMS_{initial}$, is calculated using equation 2.3. To make this noise signal have the desired RMS value, we can scale each sample in the entire signal by the factor $\frac{RMS_{desired}}{RMS_{initial}}$. The noise was then added to the clean music signal.

In the test, 200 random five seconds long recordings were generated for each of the different SNR values to experiment with, using a Python script. The percentage of correctly matched songs was automatically calculated and stored. Some multiprocessing techniques were utilized to run the test in parallel on a computer with 24 cores. A process pool was created so that 24 different instances of the test could be run at the same time, each with their own set of SNR values. This resulted in a speedup that made the test run in approximately five hours instead of the estimated

50. The resulting percentages of correctly matched songs for each SNR value was compiled into a plot, which is later shown in section 4.3. Again, the parameters F and R were both set to ten in this test.

3.3.4 Hyperparameter testing

To refine the parameters and enhance the performance of our algorithm, we conducted some computer automated tests with different hyperparameter values. The optimization aimed at refining the hyperparameters F and R, regarding the maximum number of hashes per peak, and the radius parameter in the maximum filter to be applied to the spectrogram, respectively. Both these parameters has a direct effect on the number of fingerprints extracted from a piece of audio, and thus, the execution time of the algorithm. However, more fingerprints could lead to a more effective identification. These hyperparameters needed to be selected to result in a good trade off between performance and accuracy.

During the optimization testing, we systematically varied the values of F and R within predefined ranges. In the test section, F is referred to as `Pairvalue` and R is referred to as `Filtervalue`. To compute the best values, we created a test bench that evaluated each parameter combination. For each parameter combination, the algorithm was tested with those parameter values on clean music recordings and the resulting performance was automatically stored to then later be analyzed. The goal of the optimization was to find the combination of hyperparameters that yielded the highest certainty in the testbench. The thought process was that a higher certainty would indicate a more confident answer. Our primary focus was on identifying parameters that maximized confident results rather than algorithm speed. Later on, we could pick from these parameters in a way that also maximized the algorithm efficiency. The testing bench's time complexity is primarily determined by the number of songs in the database, and the time complexity of **ShazamRecreation.find_song()**, which partly depends on the hyperparameter values. The decision was made to optimize only the two parameters F and R, which we found most important since they have a great effect on the number of fingerprints generated. Increasing the number of parameters would result in much longer testing times. This is because we need to test all combinations of the parameters. To further increase the test bench execution speed, we also decided to limit the number of songs in the database to only eight.

To represent the result of each test bench, the average value of all certainties for all combinations of parameters was calculated. This was then displayed in a heatmap with `Pairvalue` and `Filtervalue` as the two axes. The brightest pixels in the heatmap reflected the best parameters in terms of algorithm accuracy. The result of the optimization and the heatmaps themselves are later presented in section 4.4.

A major oversight that was done in this test was that when trying to match songs

using varying values for the hyperparameters, the fingerprints that were already in the database had been generated on specific values of those hyperparameters. In practice, the fingerprint extraction process will use the same values for the hyperparameters when inserting songs and when matching songs. An ideal scenario would be to use the same hyperparameters for when loading in songs as when identifying songs. However, loading in all songs with all combinations of hyperparameters would take way too long. Thus, we decided to load songs into the database using three different combinations of the hyperparameters, and generate heatmaps just for those three cases.

3.3.5 Code tests

The previous sections mostly discuss testing the whole algorithm against recordings. In this section, on the other hand, some more specific tests are brought up. These are tests that verify the functionality of smaller code blocks. In particular, the maximum filter and the peak pairing functions are tested.

The first type of test made on both functions is a *unit* test. A unit test verifies the functionality of a piece of code by comparing its output to some expected value. In the unit test for the maximum filter, an eight by eight matrix of integers between zero and ten is created. The values inside the matrix has been hand-picked by us. Another eight by eight matrix is created, containing the expected result of running the maximum filter on the first matrix. The values in this matrix has been calculated by hand. In the test, the radius parameter $R = 3$ for the maximum filter. Finally, the maximum filter is applied on the first matrix. Then, that result is compared with the second matrix. Using the *assert* keyword in Python, we can make sure that the program terminates with an error message if the test fails. The same thing was done for the peak pairing function. Some peaks where manually created, and the resulting expected peak pairs where calculated by hand. Then, the result of running the function on the peaks is compared to the expected values, and the program terminates with an error if they are not equal. In this test, the parameter F was selected to be equal to one for simplicity purposes.

The second type of test made on both functions is a *functional* test. In such a test, a more precise and often slower function is written, to compare the result with. For the maximum filter, such a function was written that loops through each element in the matrix, and compares it to all values in the surrounding box of width R. This function has four nested loops, since we are looping through each row and column of the matrix, and for each combination of those, we are looping through each row and column in the surrounding box. Although a lot slower, this function quite clearly does exactly what we want it to. An input matrix was generated and sent as input to both the fast and the slow versions of the maximum filter, and their outputs were then compared to each other to make sure they were equal. The same was done once again with the peak pairing function. A slower version was written

that didn't utilize sorting tricks for skipping peaks. The resulting peak pairs from both functions were compared with each other.

Unit tests and functional tests were not the only tests made on the code. The interface against the database was tested using an integration test, which essentially consisted of running the main program and verifying that it could call the methods in the **Connector** class to store and retrieve data from the database.

Lastly, an end to end test was performed. An end to end test in our case means running the entire application, and using it to identify a song correctly. Since some of the tests explained in previous sections (for example section 3.3.2) have been end to end tests, no extra test had to be made.

These tests are of course not enough evidence that everything works correctly, since there may be edge cases not existing in the input data. However, these tests shows that if they fail, it is definitely something wrong with the implementations. The further we develop our testbench, the higher confidence we have in the robustness of the program.

4

RESULT

The following sections show figures presenting the result of all tests described in the last chapter. The tests are partly a verification of the program's functionality, and they also showcase its performance.

4.1 Results for tests without noise

The result of the test described in section 3.3.1, where clean recordings were used directly on the algorithm, is presented here.

Test	Result
1	101/101
2	101/101
3	101/101
4	101/101
5	101/101
6	101/101
7	101/101
8	101/101
9	100/101
10	101/101
11	101/101
12	101/101
13	97/101
14	101/101
15	101/101
16	101/101
17	101/101
18	101/101
19	101/101
20	100/101

Table 4.1: *Number of songs correctly identified from different five second samples*

Out of all 20 iterations, the algorithm only failed to recognize 6 recordings. That is an accuracy of 99.8%.

4.2 Real-life noise test results

The result from testing with low levels of noise gave:

```
Fail:  5
Success:  5
```

When testing with significantly larger amount of noise the following result was obtained:

```
Fail:  7
Success:  3
```

Success means that the correct song was identified within time, and fail means either that the song identified was incorrect. The hyperparameters used was `Pairvalue` equals four and `Filtervalue` equals ten. This due to pass the time criteria stated in table 1.1.

4.3 Computer automated noise test results

The result for the automated test described under 3.3.3 is neatly shown in the following figure. Note that the SNR-axis goes from positive to negative which is intentional.

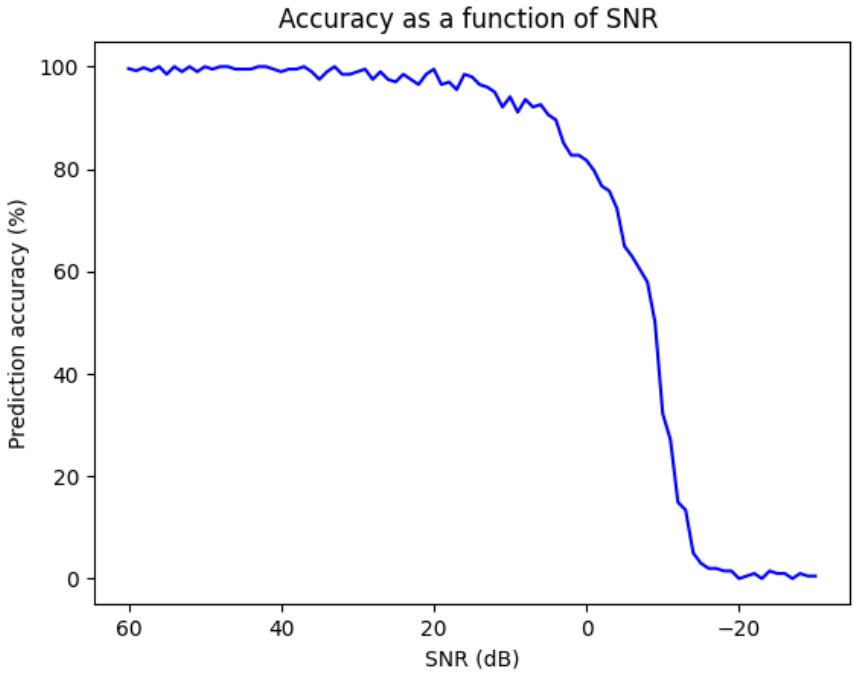


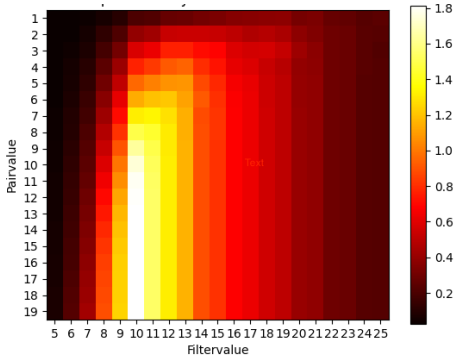
Figure 4.1: *A plot showcasing the algorithm's accuracy as it was tested on recordings of different signal-to-noise ratios.*

The algorithm maintains a high certainty even when the noise is almost as loud as the input signal. There is a steady decline in accuracy when the SNR $\approx 5dB$ which means the input is < 1.78 times larger than the noise level in terms of RMS.

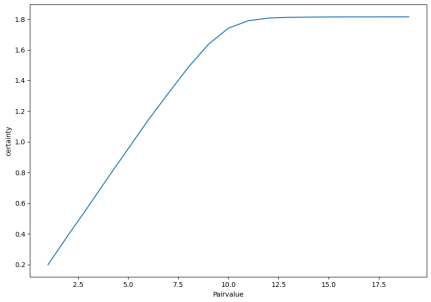
4.4 Results of hyperparameter testing

The heatmaps are the result from the parameter optimization test in section 3.3.4, and describe how the mean certainty of the matched songs depends on the hyperparameters `Filtervalue` and `Pairvalue`. The figures 4.2, 4.4 and 4.6 illustrate heatmaps, which represent how the certainty (z-axis) depends on `Pairvalue` (y-axis) and `Filtervalue` (x-axis) when matching tests are performed. Due to the mistake we made, where we inserted songs using different hyperparameters compared to later when testing, three plots have been generated. In each of the three plots, a certain `Filtervalue` and `Pairvalue` was used to add the songs, while the plot axes tell us the `Filtervalue` and `Pairvalue` which was used to identify songs.

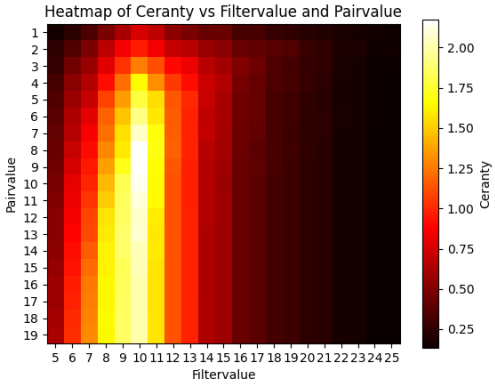
The related graph to each heatmap is the column containing the highest certainty value. The y-axis represents certainty and the x-axis represents the `Pairvalue`.



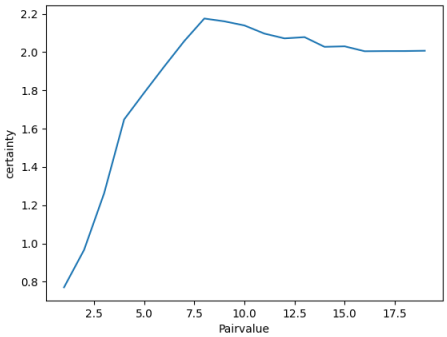
(a) *Heatmap when Pairvalue equals ten, Filtervalue equals ten.*



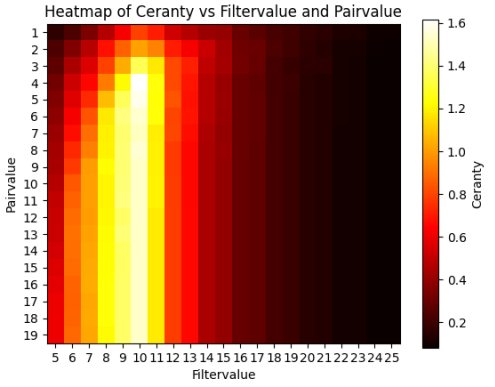
(b) *Graph over column ten in Figure 4.2.a.*



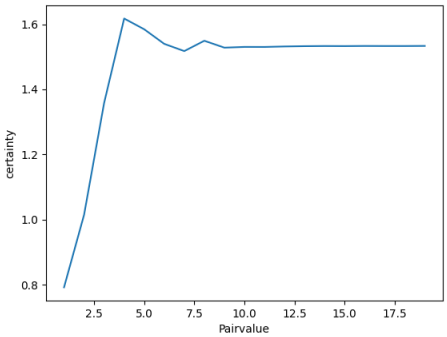
(c) *Heatmap when Pairvalue equals eight, Filtervalue equals ten.*



(d) *Graph over column ten in Figure 4.2.c.*



(e) *Heatmap when Pairvalue equals four, Filtervalue equals ten.*



(f) *Graph over column ten in Figure 4.2.e.*

Figure 4.2: *Heatmaps and the resulting graphs from the highest certainty column for three different combinations of hyperparameter values when preprocessing songs.*

Higher certainty gives brighter color in the heat maps and lower certainty gives darker color.

4.5 Results for code tests

The unit tests and functional tests explained in section 3.3.5 were executed. If any of the four tests failed, the program would terminate with an error message. If all tests succeeded, the program would terminate with a message indicating success. After executing the tests, the message indicating success appeared, meaning all tests had passed. Also the integration test and the end to end tests were verified, since multiple songs were successfully found using the complete application, as stated in section 4.2.

5

DISCUSSION

In this section, the results are evaluated in order to create a discussion about the requirements specification (see table 1.1) and the overall project objectives. Then, some improvements and ideas for future development are raised.

5.1 Verification of requirements

The first group of requirements regards the database containing fingerprints and information about songs. The database is filled with fingerprints from 101 songs. This means that requirement 1.1, which is about having enough songs, is verified. Requirement 1.2 says that the songs should be diverse genre-wise, by having at least five different genres. Many would say that genres are hard count since they aren't always very well defined. However, we can safely assume that we have at least five genres in our music database. To name a few, we have pop, electronic music, disco, hip hop, and Swedish classics. Another requirement for the music database is that the music is spread out in time. The verification for this was that at least three songs from the 1980s, 1990s, 2000s, 2010s, and 2020s should exist in the music database. Since the information we store for each song in the database contains not only the song and artist names, but also the release date, we could easily look in the database to see that it contained at least three songs from these decades. To conclude the requirements regarding the database, all three of them have been successfully verified.

The second group of requirements is about the recording part of the project. Requirement 2.1 states that the recorded samples cannot be too long. There are two reasons for including this criteria. Firstly it is unwanted that the application keeps listening forever, and secondly, the algorithm should be able to identify songs from short recordings. The main script for the application sends the recording as it is being recorded when it is three, six, nine, and twelve seconds long. This way, the longest recording that the algorithm receives as input is twelve seconds, which means that this criteria is fulfilled. The next requirements is about the recorder code being non-blocking. As explained in section 3.1.1, separate threads were used to copy the microphone audio buffer into main memory and for the rest of the code. This was done exactly for the purpose of concurrency, because this allows for the algorithm

to try to find the song at the same time as it is listening to it. With criteria 2.1 and 2.2 fulfilled, the second group of criteria in the requirements specification is complete.

The third group of requirements regards the performance of the algorithm itself. Requirement 3.1 declares that the identification process cannot be too slow. When running the application and pressing the start button, the identification result has to be shown to the user within 30 seconds of the button press. Whether a song is found or not, some result has to be ready before 30 seconds have passed. One way to achieve this time limit would have been to optimize the algorithm to always finish matching in a short time. Since the matching time depends on a large number of factors, for example the computer it runs on and the characteristics of the recording, there is no clear goal in terms of speed. Rather, we we consider a slow identification as being failed, and stop the algorithm prematurely. This is, as explained in section 3.1.9, done by after 30 seconds displaying to the user that the identification failed because it took too long. By doing it this way, we fulfill the requirement. The second and third subrequirements for performance is about identifying songs that does not contain noise. As displayed in section 4.1, most of the time 101 out of 101 songs were correctly identified when no noise was added. It was a must that at least 80 songs could be identified, and we hoped for 95. With the worst result having close to 97% accuracy, and in the average case 100% accuracy, it is clear that both these criteria have been met. The following three requirements, 3.4, 3.5 and 3.6 is about testing with real noise. The tests were conducted by simply recording a song from a speaker and talking normally in the background. When conducting tests for 3.6 specifically, which states that at least one song should be recognized with strong background noise, additional talking was done and closer to the music source. As shown in section 4.2, the result of low levels of noise resulted in a success rate of 50% and with higher levels of noise a success rate of 30%. Thereby, we deemed that the requirements was passed. Due to the nature of the test it was hard to determine the level of noise in each of the trials, but by analyzing the graph in 4.3 we can assume that the overall noise for the low noise level test was about 5dB higher than the music, and for the high noise level test, it was about 15 dB higher than the music. Lastly, requirement 3.7 is about the algorithm being concurrent, in the sense that two different recordings can be identified simultaneously by one computer. As described in section 3.1.9, new threads are spawned as soon as it is time to run **ShazamRecreation.find_song()** with a longer recording. With multiple threads active, the desired concurrency is achieved, and thus the criteria is fulfilled as well.

The fourth group of requirements is about the graphical user interface to the application. Criteria 4.1 says that the application should be similar to the real one. The verification for this is to have a single button to start the recording and identification process. In section 3.1.8, it is explained that the final interface contains just a start button and a title text. With a simple interface like this, the user experience will be very close to the one when using the real Shazam application. After finding a song, the song and artist names should be displayed, as well as the release year and album name. This is what requirement 4.2 is about. As described in section 3.1.8, all four of these variables are shown to the user if a match is found. In case no match is

found, instead of the song metadata, a notification message about no match being found is displayed. Since this is the criteria for requirement 4.3, it is also fulfilled. Requirement 4.4 and 4.5 regards accessibility. According to verification criteria for requirement 4.4, the user has to be able to select to use either English or Swedish in the graphical user interface. The criteria for requirement 4.5 is that the program must be able to be started in blind mode, where text that is displayed is also played as audible speech to the user. In the same section about the user interface, it is stated that both English and Swedish versions of the application exists, and that blind mode has been implemented. With blind mode, not only is all text presented as audible speech, but the start button is pressed automatically, so that the user doesn't have to do it. Comparing these facts with the verification criteria, it can be concluded that both the requirements 4.4 and 4.5 were successfully met, even though they weren't mandatory. Criteria 4.6 states that after a song has been found, a user can simply continue using the application and find another song if they would like to. In the same section it is stated that the start button reappears once the result has been shown, so that a user can try again. The final criteria regarding the user interface is about it working on both windows and mac operating systems. As we have someone in the group that uses a mac computer, this has been verified while developing the application. To conclude the part in the requirements specification that is about the user interface, all of the seven sub-criteria have been met.

The final part in the requirements specification regards software testing. The tests for all five requirements were explained in section 3.3.5. The results for the tests were presented in section 4.5. The tests were constructed according to the verification description in the requirements table. Since all tests were successful, the criteria for requirements 5.1 through 5.5 have been met, which means that the fifth group of requirements were all verified.

5.2 Analyzing heatmaps and hyperparameters

The testing of the program consisted of multiple tests, mainly to make sure that our program and algorithm worked smoothly but also to potentially refine some of the parameters.

The generated heatmaps (see section 4.4) displays how the certainty changed with different values on the parameters. By analyzing the maps we can see that for both low and high values of `Filtervalue` the certainty is low and less effected by the `Pairvalue`.

The reason behind this is because the lower the `Filtervalue`, the higher the number of peaks in the constellation map (see section 2.4). With a large amount of peaks present in the search area (see section 2.5), fingerprints get extracted from not only

the music, but the noise as well. This might result in a lot of incorrectly matched hashes. For higher values of `Filtervalue`, a lower number of peaks are created. The reason that no matches are found here is due to the lack of peaks in the search area. There isn't enough peaks to pair with and therefore not enough data to search after. This will generate wrong answers.

The column corresponding to a `Filtervalue` of ten contains the highest certainty values. By referring to the graphs, which represents the same certainty and `Pairvalues` plotted against column ten, we observe that the graph begins to flatten out at a certain `Pairvalue`. As `Pairvalue` increases, the certainty won't go higher and in some cases it slightly decreases. The reason behind this is because when the songs in their entirety was calculated by the algorithm and their hashes were added to the database, the `Pairvalue` and `Filtervalue` had values of almost the same parameters as where the maximum certainty was found. If we look at all the heatmaps we can see a similar pattern. We didn't get the chance to test different values of `Filtervalue` when preprocessing songs due to the lack of time.

In conclusion, the heatmaps show that matching works best when using the same hyperparameters during testing as when preprocessing songs, which is expected. When `Pairvalue` gets bigger we can see an exception where the certainty gets somewhat better with a slightly higher value of the variable `Pairvalue` in comparison to the value used when adding the music files. The reason for this is that it probably doesn't hurt to generate a few more fingerprints from a recording to match with.

5.3 Future improvements

For future projects, it would be interesting to explore the alternative approach of using artificial intelligence (AI) and machine learning (ML) for music identification. It could be possible that such a path could result in a faster identification process and recognize songs with more added noise. This was something we had in mind to explore and to compare our algorithm with, as stated in section 1.5. The alternate approach of using AI and ML for music recognition could perform better when it is trained to adapt to the different surroundings and environments. However, it is important to recognize that implementing an AI-based solution could cause challenges, such as data acquisition, model training, and computational requirements, which would have to be directed for future projects.

Many of the tests that were documented in section 4 took substantial time to execute. One solution - for further studies - to be more time efficient, could use the graphical processing unit (GPU) for faster calculations. The central processing unit (CPU) of the computer handles typically all calculations and does it very well. Although CPUs can efficiently handle sequential operations with great speed, processing large

volumes of data is less efficient. This is because the CPU doesn't have very many cores, which means that it's not very good at multitasking. However, a GPU has far more cores, although running at lower clock speeds. The GPU uses these cores for parallel computations, as well as pulls and pushes data from several different in-/outlets at the same time. Thus, for a GPU to perform better than a CPU, the calculations have to be simple and there should be many of them. For instance, a GPU could probably be used both to pair together peaks and to calculate scores for candidate songs faster. Some parallelization could even have been done on the CPU. For example, instead of spawning a new thread that tries to run the algorithm on a recording, we could spawn a new process instead. Threads in Python offers concurrency, but not parallelization, which means that although multithreading allows a CPU core to context switch between multiple pieces of code, it cannot truly run them simultaneously. Using multiprocessing instead allows multiple cores to run code at the same time, which can lead to a performance boost.

The spectrogram is generated following the procedure outlined in section 3.1.4, utilizing the theory described in section 2.1. However, some parameters used to generate the spectrogram were left unchanged. It would be interesting to experiment with different section widths as well as different levels of overlapping. Investigating these as additional hyperparameters in future testing could be worthwhile.

The windowing function was applied in the spectrogram generation (see section 3.1.4) and dampens the leakage in the frequency domain. Since the spectrogram class was one of the first classes created in the project, the Hann window function was selected without much thought. It would be interesting to explore how the algorithm behaves when other window functions are applied.

As covered in section 3.1.9, the use of multiple threads was described for running the algorithm on different recording lengths concurrently. To examine and optimize the effects of the threads in regards to the matching accuracy we would like to test the recorder using more threads, with a tighter sleeping interval between them. This would probably increase the accuracy for the simple reason that we try to match more recordings from the same song than before, but would also introduce more computations. Future testing would include finding the best number of threads to start to maximize accuracy, without slowing down the search time too much. Also the best time interval between starting threads could be explored.

In the tests displayed in figure 4.2, we saw that the lower `Pairvalue` the lower the average certainty gets, but what isn't shown is that lower `Pairvalue` also results in fewer fingerprints created witch then decreases search time. One interesting aspect that we would have liked to further look into is the possibilities of finding the best trade-off between speed and certainty of the algorithm by making a more complete search through the values of the hyperparameters. Considering that the optimal hyperparameters, as discussed in section 5.2, vary depending on the hyperparameters used during the database creation process, the database should therefore also contain fingerprints created with said hyperparameters.

It is important to note that the heatmaps were created with noise-free samples. This implies that for recordings with background noise, the ideal parameters might differ from those illustrated in the current heatmaps.

The major problems with testing the program is that it takes a lot of time. Therefore, we weren't able to conduct all the tests and optimizations that we wanted to.

5.4 Scalability

As stated in section 3.2, the database contained 101 songs during most of the tests. If the database's size was to be scaled up by adding more songs, it is interesting to discuss how the algorithm performance would scale in proportion to the number of songs. Let us assume there are S songs in the database, and that a constant N fingerprints are extracted from a recording. There are two ways the performance can scale as S increases. The first case is where we assume that only fingerprints from the correct song are matched from the database. If so, the post processing (matching) will be done on the same number of matched fingerprints, which is on the order of N, which is independent of S. The other case is where we assume that not only fingerprints from the correct song are matched from the database, but also incorrect ones. Assuming that a fingerprint from an incorrect song is being matched with probability p, and that each song has a constant M fingerprints in the database, the number of matched fingerprints will be on the order of $N + SMp$. To clarify, we match with around N correct fingerprints and around SMp incorrect ones. As S grows larger, the term SMp grows linearly in proportion to S. How the total sum $N + SMp$ grows in relation to S depends on p, since N and M are considered constants. The post processing time depends on the number of matched fingerprints $N + SMp$. For large enough values of p, the post processing execution time will scale linearly with the number of songs, S.

In practice, the second case is probably the better model, with quite a low value of p. From our experience testing the algorithm on values of S between a dozen and a hundred, the matching time seems to increase as we increase S. However, the matching time doesn't seem to tenfold as the number of songs tenfold. A conclusion can be drawn that some of the processing time comes from searching through the correctly matched fingerprints, which will stay the same. Likewise, some of the processing time comes from iterating through the matched fingerprints from incorrect songs, which grows with S.

Assuming that fingerprints rarely match with incorrect songs the time complexity of finding a recording is $O(\frac{TF}{R^2})$ because that's the number of fingerprints extracted from the recording. Each fingerprint is processed in amortized constant time complexity since we are using a hashmap in the database.

6

CONCLUSION

This thesis aimed to create a functioning application that can use acoustic fingerprinting for music identification. Both clean songs as well as songs with noise added to them, were able to be identified by the algorithm. The successful implementation of the project underscores its feasibility and that our approach was suitable for the set goals.

The Shazam algorithm can be divided into five parts, recording of a sound sample, creation of the spectrogram, finding maximum peaks, hashing peaks and finding peaks in database.

Testing of the program through both noiseless and noisy recordings showed that it was capable of reaching expected criteria by passing all of the stated requirements. For future scaling, we expect the matching time of the algorithm to slowly increase as we add more songs, until new techniques have to be implemented for it to be efficient again. Future possibilities to develop this program can be done by implementing AI solutions and further test the program to optimize the algorithm.

The project has not only given us a deep understanding of how the Shazam algorithm works. It has also taught us how advanced signal processing can be applied in practice.

7

BIBLIOGRAPHY

[1] R. Panda, R. Malheiro, and R. P. Paiva, "Audio features for music emotion recognition: A survey," *IEEE Transactions on Affective Computing*, vol. 14, no. 1, pp. 68–88, 2023.

[2] G. Peoples, "The ledger: Are there really 100,000 new songs uploaded a day? maybe more." Feb 2023, accessed on 02.04.2024. [Online]. Available: https://www.billboard.com/pro/how-much-music-added-spotify-streaming-services-daily/

[3] A. Wang, "An industrial strength audio search algorithm." Jan 2003.

[4] H. Purwins, B. Li, T. Virtanen, J. Schlüter, S.-Y. Chang, and T. Sainath, "Deep learning for audio signal processing," *IEEE Journal of Selected Topics in Signal Processing*, vol. 13, no. 2, pp. 206–219, 2019.

[5] Y. Ke, D. Hoiem, and R. Sukthankar, "Computer vision for music identification," in *2005 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'05)*, vol. 1, 2005, pp. 597–604 vol. 1.

[6] D. Bone, C.-C. Lee, T. Chaspari, J. Gibson, and S. Narayanan, "Signal processing and machine learning for mental health research and clinical applications [perspectives]," *IEEE Signal Processing Magazine*, vol. 34, no. 5, pp. 196–195, 2017.

[7] N. Kehtarnavaz, "Chapter 7 - frequency domain processing," in *Digital Signal Processing System Design (Second Edition)*, second edition ed., N. Kehtarnavaz, Ed. Burlington: Academic Press, 2008, pp. 175–196. [Online]. Available: https://www.sciencedirect.com/science/article/pii/B9780123744906000076

[8] C. MacLeod, "abracadabra: How does shazam work?" Feb 2022, accessed on 04.04.2024. [Online]. Available: https://www.cameronmacleod.com/blog/how-does-shazam-work

[9] D. H. Johnson, "Signal-to-noise ratio," *Scholarpedia*, vol. 1, no. 12, p. 2088, 2006.