Lab Work 2

André Pedrosa [85098], Filipe Pires [85122], João Alegria [85048]

Algorithmic Information Theory
Department of Electronics, Telecommunications and Informatics
University of Aveiro

November 24, 2019

Introduction

This report aims to describe the work developed for the second assignment of the course of 'Algorithmic Information Theory', explaining all programs developed by us, and presenting the results we considered most relevant regarding the quality of the solutions.

The programs implemented in C++ have the purpose of analysing and encoding audio files and ultimately being capable of, from a small audio segment, identifying the music that it most likely belongs to.

Along with the description of the solution, we also discuss the effects of the variation of the programs' parameters and how accurate are the results. All code developed is publicly accessible in our GitHub repository: `https://github.com/joao-alegria/TAI`.

1. Data Visualization

In this chapter we present a description of the dataset used, the small script we developed to convert audio segments from stereo to mono and the histograms we are capable of plotting by adapting one of the scripts given along with the assignment's description (*1*).

1.1. Datasets

We were given the access to a small dataset containing 7 audio files from different musics. It was these music fragments we used to test our code during development. Each audio file is in `.wav` format and has two signal channels (stereo). They vary between 13 and 29 seconds of audio and, when played, none seems to contain significant noise.

For testing the performance of the programs once the development phase was completed, we came up with our own dataset of musics of different genres. These music files vary between 3:07 and 8:09 minutes and have the same format and number of channels as the original dataset. As they were downloaded from the original sources, their quality is close to ideal.

1.2. Mono Conversion

One of the tasks proposed was to create a script that converts stereo audio files into mono. This was fairly straightforward to do, as it only required for us to read the v alues from all signal channels and calculate the average of each. The script is executed in the format presented below, once built:

```
$ ./wavquant [-q quantSize] [-r reductFactor] inputFile outputFile
```

This script, `wavquant.cpp`, is also used for other purposes, in which the `quantSize` and `reductFactor` parameters are useful. For this reason, we made the parameters optional, so that a user can run `wavquant` to simply convert stereo files into mono, with a default number of bits used to encode each value of the segment (each sample) of 16 and no frequency reduction factor. This script is mentioned further ahead in greater detail.

1.3. Histograms

We were also given a script `wavhist.cpp` that outputted to the terminal the histogram of an audio file. We adapted it so that it accepts both stereo and mono audio files and plots in a figure the histogram of one of the audio channels with the help of `gnuplot` (*2*), a portable command-line driven graphing utility. The resulting WAVHist script has the following command interface:

```
$ ./wavhist inputFile channel
```

Figures 1 and 2 contain the histograms plotted from the same music in the original format (stereo) and after its conversion to mono. The x axis represents the frequency of the values and the y axis the number of occurrences in the music of each frequency.

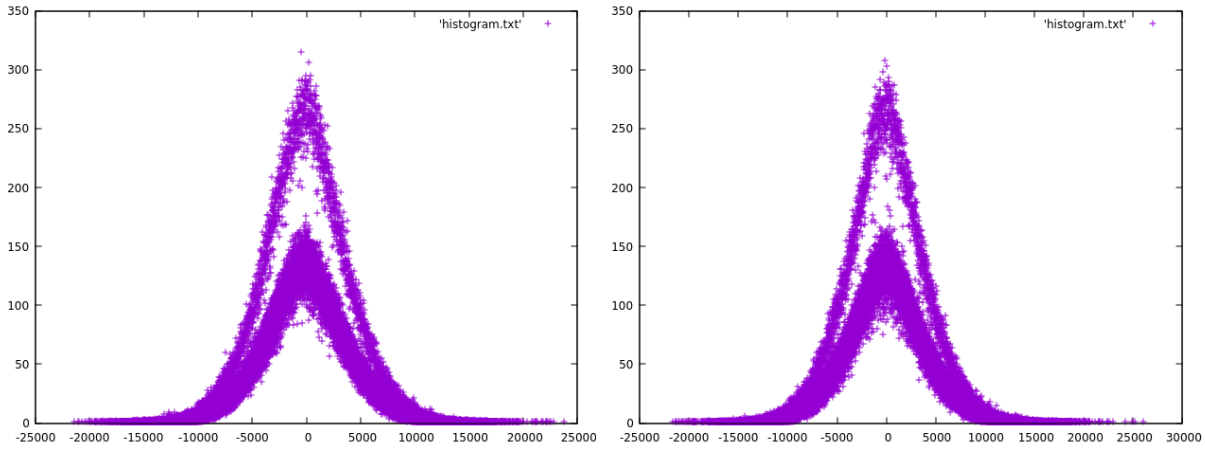

Figure 1: Histogram of sample01.wav in the original format - channels 0 and 1.

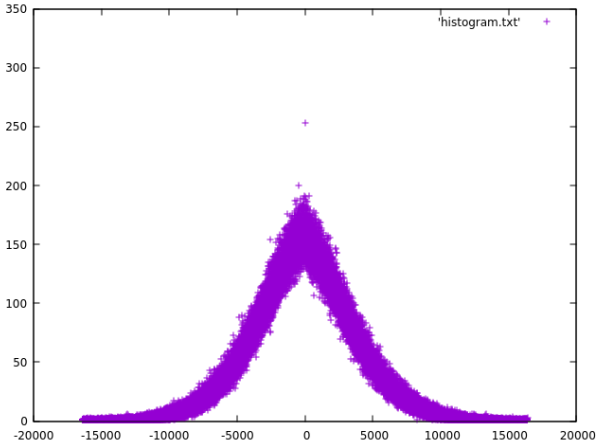

Figure 2: Histogram of sample01.wav after its conversion to mono (1 channel only).

It is clear that the mono conversion affected the way the music file is represented, although it maintains a high fidelity to the original music. When analyzing Figures 1 and 2, the first thing we notice is the disappearance of the gap present in both channels of the stereo signal. This gap exists because the music has that frequency registry - note that each value in the x axis has one and only one y value -, so for this gap to occur close frequencies must have very different counts. When converting to mono, it is natural that this gap disappear, since it is necessary to take both sample channels and obtain their average. If for one channel the value is in the top curve and for the other it's on the bottom curve, their average will be placed in the middle. If this happens enough times, the gap is closed with the average values.

Another interesting phenomenon observed from these figures is that, in the mono signal, the number of occurrences decreases a bit. This once again can be explained by interpreting the mono conversion process: as each pair of samples will be aggregated into one, all the frequencies that occur very frequently have a high probability of being paired with each other, spreading the frequency counts in the way we see in Figure 2.

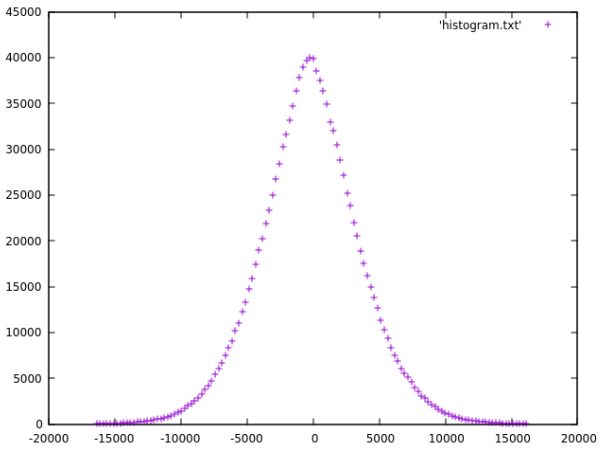

Figure 3: 8 bits, no reduction

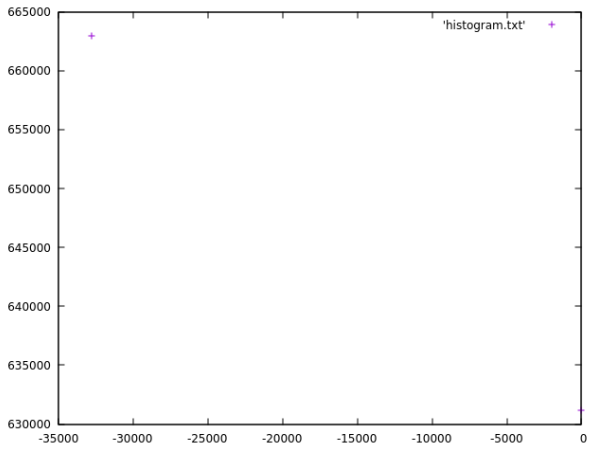

Figure 4: 1 bit, no reduction

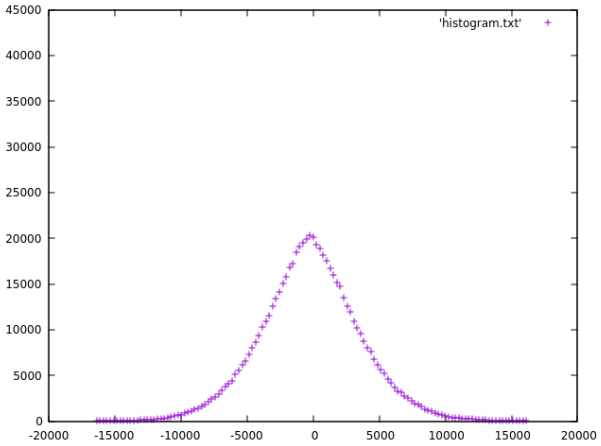

Figure 5: 8 bits, 2 reduction

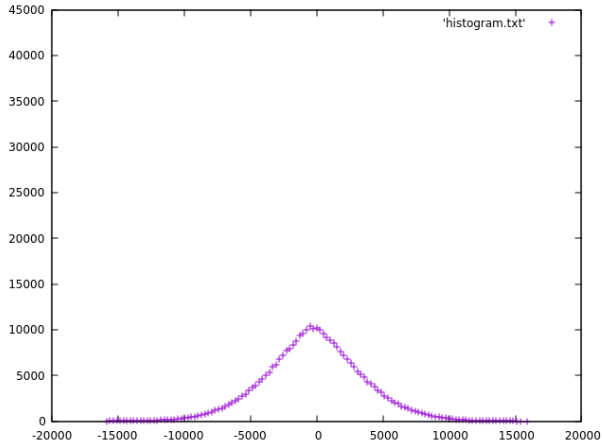

Figure 6: 8 bits, 4 reduction

Figures 3, 4, 5 and 6 depict two different studies: the first two dedicated to varying the quantization bit size; the last two dedicated to varying the sampling rate reduction amount.

The first two figures represent the histograms of the music segment `sample01.wav` quantized with 8 and 1 bits, respectively, and without a reduction of the sampling rate. A clear takeaway from these figures is that Figure 3 has a lot more frequencies than the other, which contains only two values. Also, the frequency count values in Figure 4 are considerably higher. This behavior was to be expected, since when quantizing a signal we intend to decrease the number of bits required to represent each value, discarding the less significant values; for that reason, when using 1 bit to quantize the signal, there should be 2 resulting values, one in case

the most significant bit is 1 and another if the bit is 0, and between them there should be all the samples present in the original signal.

Theoretically speaking, it is possible to encounter 2^b values/levels, being b the number of bits used to quantize the signal. However, in reality this number is not always observed since the original signal can have values that generate every level.

In relation to the Figures 5 and 6, which represent the reduction sampling rate factor behavior over a mono signal, quantized with 8 bits of the `sample01.wav` music, it is observable that the higher the reduction the lower the count values become. This is directly related to the sampling reduction algorithm implementation, which aggregates as many samples the user indicated, i.e. if the user gave 4 as the reduction factor the algorithm will then aggregate every 4 samples, creating a 4:1 ratio and decreasing the sampling rate to 25% of the original. This in turn will imply that the number of frequency count values will decrease, since there are less values but also due to the aggregation: if there are very common frequencies, those occurrences end up distributed between other frequencies.

The C++ scripts mentioned in this chapter all use `libsndfile`, a C library used for reading and writing files containing sampled sound (*3*). This was proposed on the assignment and allowed us to read and manipulate the audio files for more complex tasks.

2. Data Processing

Once we were capable of visualizing the data, we proceeded to actually doing something useful with it. In this chapter we explain the implementation of the program `wavquant.cpp`, responsible for reducing the number of bits used to represent each audio sample. The implementation of the formulas presented on the assignment's description for the signal-to-noise ratio and the energies of the signals and noises is described as well, along with the computation of vector quantization codebooks of audio files.

2.1. Uniform Scalar Quantization

The idea behind a uniform scalar quantization (USQ) is the reduction of bits used to represent a signal. Its usage has an instrinsic tradeoff between signal quality and memory space required to store the information. We do not get into much detail regarding the mathematics behind this process, but we make available a figure taken from a presentation from the Stanford University (*4*) that helps visualizing the outcome of applying the USQ to a signal. Figure 7 contains a signal presented in blue and the outcome of the signal after it is quantized is presented in red. The figure also contains the quantization error variation on the second plot.

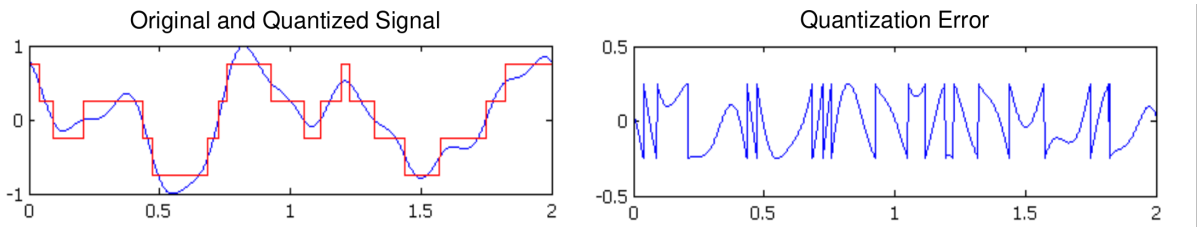

Figure 7: Example of a quantized waveform.

It is `wavquant.cpp` that is responsible for this process. As we have seen in Section 1.2, the script accepts two optional parameters: `quantSize` defines the number of bits to be used to represent the audio fragment given as input (ideally, this value should be less than of the original source); `reductFactor` defines the number of times the user would like to reduce the total number of values of the audio signal. This reduction factor works by calculating the average between `n` values, where `n = reductFactor`, and doing this for all values from the segment. The result is a signal with `n` times less values (samples).

2.2. Error Calculation

A signal-to-noise (SNR) ratio compares a level of signal power to a level of noise power. Higher numbers generally mean a better specification, since there is more useful information (the signal) than there is unwanted data (the noise).

In `wavcb.cpp` we calculate this ratio, along with the maximum absolute error per sample. The SNR is defined as stated in equation 1, taken from the assignment's description.

$$SNR = 10log_{10}\frac{E_s}{E_n}(dB) \tag{1}$$

Here, E_s is the energy of the signal, given by $E_s = \sum_k x_k^2$, and E_n is the energy of the noise, given by $E_n = \sum_k (x_k - \bar{x}_k)^2$, where x is the values from the audio segment. The maximum error is defined in equation 2, derived from the noise energy equation.

$$error = |x_k - \bar{x}_k| \Leftrightarrow max(error) = max(|x_k - \bar{x}_k|) \tag{2}$$

In practice, the error of a quantization tells us how distant from the original signal is the quantized one. Figure 7 shows an example of how this error looks like.

However, for the following assignment tasks, we determined that the energy of the noise would be more useful to us. One way to compare audio fragments would be through the signal-to-noise ratios; But we found that the only calculations actually required are of the noise energies as they work as the distances between two samples. By focusing on these values, not only do we save processing time, but we are also able to compare input audio segments to quantization codebooks already present in a program to determine the similarity between samples (and consequently between segments). These codebooks are explained in this next section.

2.3. Vector Quantization Codebook

Each music from the datasets is represented as a sequence of samples, where each sample is a singular frequency (if the file is in mono) or the pair of frequencies (one per channel, if the file is in stereo) that was registered in that exact second. What a codebook is is basically an abstract representation of the overall sequence of samples.

In the context of our project, we integrated the concept of clusters to represent a given number of samples. The base idea of this clustering is that similar data entities should have similar properties and can therefore be aggregated in a group. On the other hand, data entries that differ significantly from each other have properties considerably different as well, implying that they should belong to different groups (*6*).

Dividing the totality of samples present in each music into vectors with 'x' samples and representing them as points in a multidimensional vectorial space - as seen in Figure 8 - enables us to better comprehend the structure of the clusters. Geometrically speaking, clusters can be directly identified by the close grouping of the multidimensional points, since each axis represents a data property and similar points with similar properties stay closer to each other.

By taking advantage of this natural grouping, clustering algorithms try to find points that represent the several data groups in a useful way, usually corresponding to the middle point of each group, known as centroid.

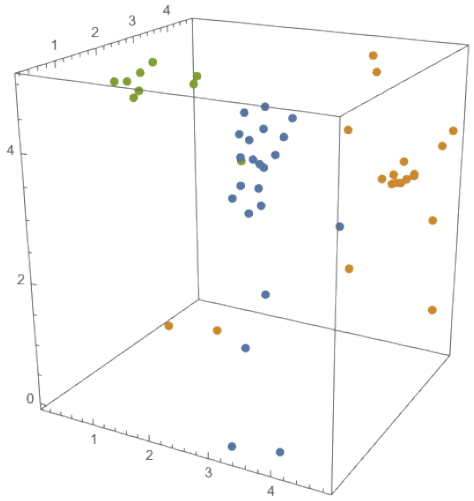

Figure 8: Example of a 3-D visualization of point clusters.

So, in short, our interest in using such algorithms lies in the applicability of converting audio files into multidimensional clusters so that the most likely cluster a new audio segment belongs to can be more easily identified. A codebook is defined by the clusters that resulted from the process of mapping the input audio file to its multidimensions.

In our implementation we used the K-Means Clustering algorithm, one of the most well-known clustering algorithm (*6*). K-Means was considered a good approach because it is relatively simple to implement and fast to execute, as it only requires the calculation of the distance between a given point and all existing centroids.

The way it works is as follows. The code starts by doing the already mentioned music partition in different blocks/vectors of a given size provided by the user; Then the algorithm chooses, in a random fashion, an number of centroids given *a priori*; Once the pre-processing is done, the main algorithm begins calculating the closest centroid for each vector, and then updates each centroid by calculating the middle point of the points assigned to that specific centroid - this is repeated as many times as necessary for the error delta, i.e. the error between each iteration, to be lower than a given threshold specified by the user.

The script `wavcb.cpp` is the implementation of this algorithm. It enables the creation of a codebook for a given music by following the format below:

```
$ ./wavcb inputFile blockSize overlapFactor errorThreshold
  numRuns outputFile
```

InputFile is the path to the .wav music whose codebook is to be constructed. *BlockSize* is the value used as the size (number of samples) of the blocks in which the audio file is split. *OverlapFactor* corresponds to the factor (a value between 0 and 1) that each block will be overlapped with the previous block; the bigger the overlapping percentage is, the better the music is represented in the multidimensional space. *CodebookDir* is the path to the directory containing the preprocessed codebooks of the music dataset. *ErrorThreshold* is the maximum error that is allowed to exist in the last iteration; below this threshold, the centroids are no longer being adjusted in a worthwhile manner. *NumRuns* is the number of times the K-Means algorithm will run to find the best local minimum; each run is initialized with the centroids in different positions. At last, *outputFile* is the path to the file were the codebook will be stored.

Although quite flexible, K-Means has some disadvantages. The most significant to our needs are the fact that it is necessary to insert the number of centroids to take in consideration which, in many situations, is not the best option, since the main purpose of using clustering is to get some insights about the data, preferring that the algorithm finds the number of centroids by itself; also, as K-Means' centroids start from a random starting position that is different each time it is executed, this makes it non-deterministic and therefore not optimal as it can converge into a local minimum. This last issue is dealt with in our solution by running K-Means several times to gain access to several local minimums and then choose the best one. As it is stated above, the number of times it is executed is defined by *numRuns*.

2.4. Codebook Parallel Processing

As audio files get longer, the number of frames per file increases and consequently so does the number of blocks that will be extracted from them. Furthermore there is the overlapping factor detail, which increases even more the total number of blocks. This high number of blocks brings computational time implications since, on the blocks classification step of the K-Means algorithm, each centroid has to be compared with each block and the closest centroid for each block must be determined.

Even though K-Means is considered a fast clustering algorithm, if given a considerable number of data entries, it is understandable that it starts to become cumbersome. To deal with this effect, we introduced parallelism into the process. Practically speaking this means that we divide the blocks into groups and create threads responsible for them and the calculation of the closest centroid. To prevent having to create complex synchronization mechanisms, each thread stores the association between blocks and the closest centroids on different data structures and then the main thread is in charge of reading from those to recalculate the centroids.

As mentioned in the previous section, we deal with one of the K-Means algorithm's disadvantages by testing it several times with different centroid initializations. It is needless to say that this also slows down the algorithm. To ease this, each K-Means run can be assigned to a thread, allowing to execute several runs at the same time. In terms of implementation it implies some synchronization mechanisms, so the main thread knows when a run was completed and to manipulate the structure to store the centroids plus the associated error calculated on each run.

Figure 9 is a visual representation of the execution flow of generating a codebook. The two independent time axis correspond to executing without additional threads and with 2 threads in each parallelization point. Note that this is not an exact time representation and is ment only to illustrate the advantage of multithreading.

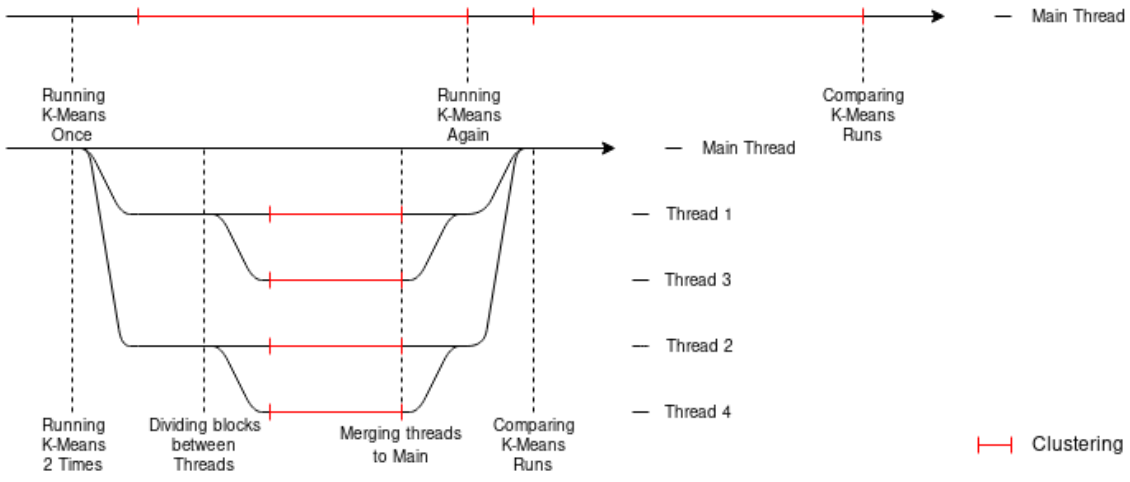

Figure 9: Visual representation of the execution flows with and without threading.

3. Automatic Music Identification

The program `wavfind.cpp` is the application of the previous scripts on a program with a specific purpose. WAVFind is meant to interpret a mono audio sample and attempt to identify which music from a database it belongs to. In this chapter we discuss our solution, the consequences of varying the parameters passed to it and the quality of the results.

3.1. Most Probable Music

The idea of WAVFind is very similar to the well known mobile app Shazam (*5*) - a user plays a sound, the app listens to it, processes it and determines the most probable music that sound belongs to. Shazam has a large infrastructure, counting on online real-time music detection, a nice user interface with access to the smartphone's microphone, and a variety of other features. We, on the other hand, focus on the functional end of this idea, implementing it as a command line program with a limited dataset of known musics and with the capacity to receive audio segments as input through their respective paths on the computer running the command.

To execute this command, one needs to use the following format:

```
$ ./wavfind inputFile blockSize overlapFactor codebookDir
```

All arguments are mandatory and most of them we have seen on previous commands. *InputFile* is the path to the .wav audio whose origin music is to be identified. *BlockSize* is the value used as the size of the blocks in which the audio segment is split; this value must be the same size used when the codebooks of the music dataset were built. *OverlapFactor* corresponds to the factor (a value between 0 and 1) that each block will be overlapped with the previous block. At last, *codebookDir* is the path to the directory containing the preprocessed codebooks of the music dataset.

The dataflow of the program is as follows: first, the *inputFile* is validated and its information is extracted; then, it is split into blocks with size according to the parameter *blockSize*; now, for each codebook present in the *codebookDir*, each segment block is compared to each codebook block by calculating the E_n (noise energy) between the two; here, the minimum E_n of each segment block is summed to a cumulative total that represents the error between the segment and the codebook; as this cumulative error is calculated for all codebooks, we then check which codebook returned the smallest error and assign it as the most similar to the given audio segment. This smallest error is determined iteratively - as each codebook is processed, we compare its cumulative error to the previous one and keep the smallest. The identified codebook's is finally printed onto the console (as its name is supposed to identify the music it belongs to).

Now, for the program to work properly, there are a few prerequisites. First of all, the program must have access to the codebook dataset. The process of creating codebooks is neither trivial nor short-lasting, so to have access to the audio dataset is not enough. Next, the audio segments passed as input files must contain only one signal channel (mono). This restriction reduces the complexity of the comparison process, with the obvious tradeoff of reducing the quality of the audio. On the other hand, in a real case scenario (like with Shazam), the user would usually record audio fragments from external sources such as speakers; so, to consider characteristics related to stereo files in the music identification process could lead to deceiving the program as the recording could fail in distributing the collected information to the proper channels.

3.2. Parameters Variation

In this section we present some results of executing WAVFind for a few samples from the datasets, explaining the experiences done with variations to the parameters passed to the command and their consequences.

The first experience conducted regarded the effect of the overlapping factor on the accuracy of the results. For this particular study, the use of the larger dataset did not bring any advantages so we used only the samples from the initial dataset. The applied method focused on fixing all the parameters at meaningful values and varying the overlapping factor so that conclusions could be retrieved from the results of testing the identification capabilities of WAVFind.

The way we tested the samples was through a shell script `evaluateCodebooks.sh` that randomly chooses one of the samples and also randomly chooses a small portion of that file and then executes `wavfind.cpp` for that portion and returns the correct answer along side with WAVFind's guess (it repeats this process as many times the user wishes). It is up for the user to manually check whether the guesses were correct or not.

The audio files were first preprocessed and converted to mono files with a quantization of 8 bits and no reduction factor. The codebooks were built with blocks of size 800, an error threshold of 0.01 and with K-Means being run twice. Five codebook folders were created, each containing the codebooks of all audio segments of the initial dataset, but created with different overlapping factors: 0.00, 0.10, 0.25, 0.50 and 0.75. `EvaluateCodebooks.sh` was executed 10 times for each codebook folder, and each execution ran 10 tests. The average values of the 10 executions were then calculated. The results are presented in Table 1.

Overlapping Factor	Score
0.00	6/10
0.10	8/10
0.25	9/10
0.50	9/10
0.75	9/10

Table 1: Test results on varying the overlapping factor between blocks.

Our second study was dedicated to find ..... //

The last study was related to multithreading and the benefits of parallelism. The goal was to build codebooks for musics using different strategies and determining what is the fastest alternative. It is important to state that, due to the shortage of time for conducting this test for all available musics, we focused on one music from the larger dataset. The chosen music was *Don't Worry Be Happy*, by Bob Marley, with a disk space usage of 51.1MB, converted to mono, quantized with 8 bits with a reduction factor of 5. The remaining parameters were once again kept at fixed values, with the blocks' size equal to 800, the overlapping factor at 0.1, an error threshold of 0.01 and the K-Means algorithm executed twice. The computer used to perform this study was an SSD, i7 with 4 cores and 16GB of RAM. The times taken for building the codebooks are presented in Table 2. Note: most of the elements of the first column contain two values, these correspond to the number of threads used for all K-Means runs and to the number of threads distributed with blocks for clustering.

Number of Additional Threads	Execution Time
Main Thread Only	40 min. 06 sec.
1, 4	29 min. 54 sec.
2, 2	27 min. 36 sec.
2, 1	31 min. 25 sec.
1, 2	33 min. 03 sec.

Table 2: Test results on varying the number of threads during codebook construction.

The results are not so easy to interpret, as we must take in consideration that multithreading also depends on the number of available processing cores in the hardware executing our code. However, it is possible to see that, on average, multithreading will be faster than using one thread only. This is not linear, i.e. more threads does not necessarily mean more speed, but there is a clear decrease in execution time from the first alternative to the remaining. .....

3.3. Results Discussion

In this section we analyse the results of the previous studies done on the developed programs and reach some conclusions regarding the optimal parameters.

From the results of the first study, we were able to conclude that introducing an overlapping between blocks does indeed benefit the program's accuracy. On the other hand, a high value does not always mean a better result. Taking in consideration the increase in execution time with higher overlapping factors, we concluded that 0.25 was a good balance between effectiveness and speed.

Conclusions

After completing the assignment, we drew a few conclusions regarding our solutions and the applicability of algorithms such as the K-Means to solving problems such as music identification.

First of all, ...

Regarding our satisfaction with the delivered code,...

In terms of code organization and readability, we made sure our repository was as well structured as possible and our code properly commented and documented. The base folder contains a *README* file for basic instructions and a *Makefile* to make the compilation process easier. All code is in the *src* folder and its documentation is accessible, with the help of the *Makefile* and the command "make docs", through the automatically generated *index.html* file in the *docs* directory.

References

1. Armando J. Pinho, *AIT: Lab Work no.2*, University of Aveiro, 2019/20.

2. H.B. Broeker, G. Clark, L. Hecking and E. Merritt, *Gnuplot: graphing utility*, `http://www.gnuplot.info/` , May 2019, [accessed in: November 2019].

3. Free Software Foundation, *GLibsndfile API*, `http://www.mega-nerd.com/libsndfile/api.html` , April 2013, [accessed in: November 2019].

4. Bernd Girod, *Image and Video Compression: Quantization*, `https://web.stanford.edu/class/ee398a/handouts/lectures/05-Quantization.pdf` , [accessed in: November 2019].

5. Apple Inc., *Shazam for iOS & Android*, `https://www.shazam.com/apps` , [accessed in: November 2019].

6. George Seif, *The 5 Clustering Algorithms Data Scientists Need to Know*, `https://towardsdatascience.com/the-5-clustering-algorithms-data-scientists-need-to-know-a36d136ef68` , [accessed in: November 2019].