Universidade de Aveiro
Departamento de Electrónica, Telecomunicações e Informática
Mestrado em Engenharia de Computadores e Telemática

Information and Codification

IC - Report Project 1

Fabio Matias - 108011
Gonçalo Cunha - 108352
Matilde Teixeira - 108193

Aveiro
Outubro 2024

Contents

1 Introduction

The purpose of this project was to study, read and analyze data by manipulating through different forms of it (text, audio and image) to obtain information and comprehend useful data that can be obtained by it.

Throughout this project we faced some setbacks, but also learned useful knowledge of c++, like how to deeply analyze signals, images and texts and transform this last one.

In this report we will thoroughly demonstrate the outcomes of the various parts of the project, compare results through graphs and tables, as well as draw conclusions from it.

2 Methodology

In this section we will explain the key transformations and algorithms applied for each data type, whilst explaining methods and tools used in order to obtain the pretended result.

2.1 Text Data Processing

In order to obtain the expected result the flow of action can be separated into different phases.

2.1.1 Reading File

In reading the different files, the file is read using wide-character streams to support Unicode input.

By using this large characters we allow that accents are read and accounted for and transformations (capitals to non-capital) are applied.

After the reading, the data is stored in vectors because it can dynamically change size, and elements can be access through index.

Stores elements like an array but can dynamically change in size. Adding and removing of elements are usually done at the end. Elements can be accessed by index.

In each function a locale (locale loc("");) is specified, which defines the environment for execution. By passing an empty string, we define that the default system locale is to be used.

2.1.2 Data Transformation

The transformations applied were lowercase conversion and punctuation removal.

Lowercase Conversion

For the first step, we used the function `transform` to apply the transformation to all characters of the line. A lambda function `[loc](wchar_t c)` is then applied to define how each character (`wchar_t c`) should be transformed,

by using the ctype<wchar_t> facet of the locale loc. The ctype<wchar_t> facet is responsible for character classification and conversion for wide characters (wchar_t).

The wide characters are converted to their lowercase equivalents according to the rules defined by the locale (loc). If the character c is already in lowercase, no changes are made.

Punctuation Removal

In this step, we iterate through each character of the text's lines and check if it belongs to a set of punctuation characters (such as !, ?, ,, etc.). If a character is identified as punctuation, it is removed from the line.

To achieve this, we use line.erase(...) to remove the identified elements from the line. Nested within this, we utilize remove_if(), which is an algorithm that is part of the C++ Standard Library and is defined in the <algorithm> header.

The purpose of remove_if() is to remove elements from a range that satisfy a certain condition (in this case, the condition defined by the lambda function). It does not actually remove elements but instead rearranges the elements in such a way that the elements that should be removed are moved to the end of the container. It returns an iterator pointing to the new logical end of the range (where the "removed" elements start).

After this call, the elements that are considered "removed" are still present in the container but are now at the end, and their logical position is after the iterator returned by remove_if().

To verify if a character is punctuation, we use iswpunct(), which is a variation of ispunct but for wide characters. This function evaluates each character, even if it has an accent. The C++ Standard Library function iswpunct() returns true or false, indicating whether the character being evaluated is punctuation or not.

Therefore, line.erase(...) erases the elements pointed to by the iterator returned from remove_if(), which analyzes the return values from iswpunct().

2.1.3 Character Frequency Analysis

The next step involved analyzing character frequency. Our approach was to iterate through the text to access each line, and then to iterate through the characters of each line to identify the different characters present.

To check whether a given wide character (wchar_t) is an alphabetic character (i.e., a letter), we utilized the function iswalpha(), which is part of the C++ Standard Library and included in the <cwctype> header.

This function returns true if the character meets the criteria for being alphabetic and false otherwise.

If it met the case, we accessed the current count associated with character in the map. If character does not exist in the map yet, it would be automatically inserted with an initial value of 1, else it would add one.

2.1.4 Word Frequency Analysis

The final part of the obligatory part for Text Processing, was to analyze the word frequency.

For that to be possible we used the same approach of iterating through the text's lines and used a wstringstream object to allow the line to be treated as a stream, enabling easy extraction of words.

In the `while (wss>>word)` the `>>` operator extracts the words from the wstringstream into the word. After that if the word is not anywhere in the map, it is added to the map, else the word is associated to the value 1.

2.1.5 Plotting

To plot graphs to depict the characters and word occurence, we used matplotlibcpp. In order to use it we add to add categories to the make file and clone the repository in order for the file to recognize its library.

For the word frequency, the top 10 most appearing words were selected. Since matplotlib has limited support for wide characters on the axes, we utilized a converter to transform the N-gram (a wide character string) into a standard byte string, ensuring compatibility for proper display on the plot. For the x-axis the index values were pushed, and for the y-axis the values associated with the words.

Then, using the plt namespace the graph is drawn using plot and with its respective labels, title and ticks.

2.1.6 N-Gram Generation

An optional of the presented guide, was to generate Ngrams of the texts and present its graph.

The function starts to choose the appropriate label if the user wants the bi-gram graph or trigram, and then decompose the words using the wstringstream to enable easy word extraction just like in word frequency analysis. If the number of words in line is less than n, it skips to the next line.

For the n-gram construction it results by concatenating n consecutive words, and couting if the these words combined appear more than once. The N-gram is added to the nGramsMap, incrementing its count. If the N-gram already exists in the map, its count is increased; if not, it is initialized to 1.

Following this, the results are shown in a graph using the same function as the one used to plot the frequencies histograms.

2.2 Audio Data Processing

2.2.1 Reading File

To read any given file, our program uses the method `readFile()`, which requires an input of the file path in the form of a string.

Using the `ifstream` object, we can open, and verify the state of the file (whether it was correctly opened or not), then, we load the data on to the buffer, and set the SFML sound to the loaded buffer.

2.2.2 Metadata

There is several metadata to extract from each file, as it was one of the requirements of the project. We extract things like bit-rate, channel numbers, the duration and the sample count.

We obtain that data through the function `getAudioInfo()` which uses some methods from the Sound Buffer object to extract such information.

2.2.3 Data Transformation

In this part of the work, there are two requirements that relate to data transformation.

One of them is quantization, which, in our program, is done by a function with that same name, an integer needs to be imputed to define the level of quantization applied, the higher the number, the more precise will be the audio output. This function does this by retrieving the audio samples from the buffer as well as the samples count, using methods from the Sound Buffer object, next it verifies if number of audio samples is not 0, otherwise there won't be nothing to process, another condition is to compare the max and minimum values of the samples, because if they have same value, no quantization is needed.

The quantization itself is done by iterating through the `quantizedSamples` array and normalizing, rounding to a quantization level and storing each sample into the array.

Other way in which our program manipulates the audio data is by adding noise, done by the `noiseAdder` method. Like the quantization function it gets the sample count from any given audio, iterates through a copy array of the original audio buffer, and adds random noise to its individual samples. The resulting audio is then outputted into a new file.

2.2.4 Plotting

For plotting, we did some functions, the first of which was, `plotAudioWaveform`

First, it sets up the window dimensions and loads a font for the labels. Then it retrieves the audio samples and calculates the necessary scaling factors to fit the waveform within the window's dimensions. The function determines whether to use the original samples or quantized ones.

Next, it initializes the waveform data for each audio channel, mapping each audio sample to a graphical position based on the scaling factors. The colors of the waveforms are set differently for each channel, and for quantized data, the color is red.

The function also sets up and draws the x-axis and y-axis for each channel, positioning them appropriately. It labels the axes with "Time (s)" for the x-axis and "Amplitude" for the y-axis.

To give a clearer representation, the method creates and positions major and minor tick marks along the x-axis for time intervals and y-axis for amplitude levels. It also labels the major ticks.

Other methods work in similar ways as this one, them being, `plotHistogram` that first checks if the audio data is stereo (contains two channels). Depending on the given title (e.g., "Right Channel", "Left Channel"), it selects the target audio channel from the stereo data. If the target channel is empty, it exits with an error message.Next, it initializes a histogram vector with 256 bins to handle 16-bit audio values. Each bin counts the occurrences of audio sample values within a specific range. It also sets up the graphical window dimensions and properties, including font and colors.

The function then populates the histogram by iterating through the target channel's samples, mapping each sample to a histogram bin. It calculates the maximum bin value to properly scale the histogram bars later.

For graphical representation, it sets up the window and calculates positions for the histogram bars, x-axis, and y-axis. The function then creates and positions the axis labels and tick marks for both axes.

At last, the `plotBothWaveforms()` method defines the window and margin dimensions, then sets up an SFML window titled "Original vs Quantized Audio Waveforms". A font is loaded for labeling.

The function retrieves the audio samples, calculates the necessary scaling factors for both axes, and initializes lambda functions to draw the waveforms, axes, and ticks. These lambda functions ensure that the drawing process is consistent for both waveforms. Then, it draws the original waveform in blue and the quantized waveform in red, positioning them side by side. The drawWaveform lambda quantizes the samples for the quantized waveform and scales both waveforms to fit within their respective graph areas.

Axes are drawn for each waveform, as well as tick marks along both the x-axis (sample numbers) and y-axis (amplitudes). Text labels are created for the axes and waveforms using the createText lambda.

The frequencyAnalyser function processes an audio file to visualize its frequency spectrum using the Fast Fourier Transform (FFT). Initially, it retrieves audio samples from a buffer, calculating the total sample count and channel count to determine the number of samples per channel. It then prepares for the FFT by creating two vectors—one for the real part of the samples and another for the imaginary part, which is initialized to zero. The real samples are normalized by dividing by 32768.0, and memory is allocated for the FFT input and output arrays using the FFTW library.

Once the input is populated with the real samples, the FFT is executed to

transform the time-domain samples into the frequency domain. Afterward, the function calculates the magnitude of the resulting frequency components.

A hidden SFML render window is then created to draw the frequency spectrum. The function sets up a `sf::VertexArray` for representing the spectrum, determining appropriate scaling factors based on the window size and the computed magnitudes. Each frequency bin is drawn as a vertical line corresponding to its magnitude.

To enhance the visualization, the function creates and positions axes along with labels, configuring tick marks and labels for both axes, formatted for clarity. It enters an event loop to keep the window responsive, rendering the axes, spectrum, and tick labels accordingly.

Finally, the content of the window is captured as an image, which is saved to a specified directory, ensuring any necessary folders are created to accommodate the output path. The window is closed after saving the image. In summary, this function effectively analyzes the frequency content of an audio signal, visualizes it, and exports the resulting plot as an image file.

2.3 Image Data Processing

All the methods used for image manipulation were part of the OpenCV library and are integrated in the **imageProcessor** class, with 1-3 functions for each task. We also developed a simple main function that uses every method in this class.

2.3.1 Task 1 - loadImage()

In task 1 our program uses the cv::imread() function to load a image from a file, and storing it in an OpenCV matrix. In the case of color images, the decoded images will have the channels stored in Blue Green Red order. The image is then shown in a window using the cv::imshow() function, and the cv::waitKey(0) function is used to wait for a key press indefinitely to close the image window. This function lays the foundation for any image analysis/manipulation, since the first step in this process is loading an image.

2.3.2 Task 2 - splitImage(), toGrayscale()

In the second task we process a image by using cv::split() to split an image stored in an OpenCV matrix (Mat) in a vector of 3 arrays, which are then displayed individually using cv::imshow(). This allows for the clearer visualization of the different levels of each color on the image.

This task also includes the creation of a function that converts a image to gray scale using cv::cvtColor() with the *COLOR_BGR2GRAY* flag and displays it with cv::imshow(). Converting to gray scale is often used when there is a need for further image analysis for which color information isn't necessary, allowing for better optimization.

2.3.3 Task 3 - calculateHistogram()

Using the already processed gray scale image to calculate the image histogram, we use cv::calcHist(), configuring the histogram to have 256 bins and a value range of 0 to 256 (to cover the full range of pixels for an 8-bit image), no mask, using the channel 0 (gray scale), and 1 dimension. This histogram is then converted to a vector of floats and then is plotted using matplotlib, allowing the visualization of the distribution of pixel intensities in a bar graph.

2.3.4 Task 4 - gaussianBlur()

In task 4 we apply the cv::GaussianBlur() function to our original image in the Mat format, using a 3x3 core and a 7x7 core, creating two new Mat arrays, which are displayed using cv::imshow, allowing the comparison with the original image, which is noticeably less blurry. The blurred versions are then saved as a *.ppm* file using cv::imwrite().

2.3.5 Task 5 - imageDifference(), calculateMSE(), calculatePSNR()

The first function implemented for task 5, imageDifference(), creates a new Mat array, and uses cv::absdiff() to calculate the difference between the 2 images, then displaying it with cv::imshow().

The other 2 functions are related, since calculateMSE() is called by calculatePSNR(), and the PSNR (Peak Noise to signal Ratio) is impossible to calculate when the MSE (Mean Squared Error) is 0, since there is no noise in the signal. The MSE corresponds to the average squared differences between corresponding pixel values in the two images, and together with the PSNR we can compare a reconstructed image after compression against the original. The higher the PSNR, and the lower the MSE, the better the quality of the reconstructed image.

2.3.6 Task 6 - quantizeImage()

Finnaly, in task 6, we implemented a function that allows for the quantization of the pixel values of an image, acording to the number of levels passed when calling the function. The lesser the number of levels, the worse is the quality of the image, leading to a bigger MSE and smaller PSNR.

This function starts by calculating the scale factor based on the number of levels. This scale factor is used to map the pixel values to the desired quantization levels. After being converted to floating point for better precision, the image's pixel values are scaled down to the range [0, levels-1], after which 0.5 is added to each pixel value to facilitate rounding when converting back to an integer format. The image is converted back to an 8-bit format, rounding the pixel values, and are then scaled back up to the original range, but now they are quantized to the specified number of levels.

3 Results

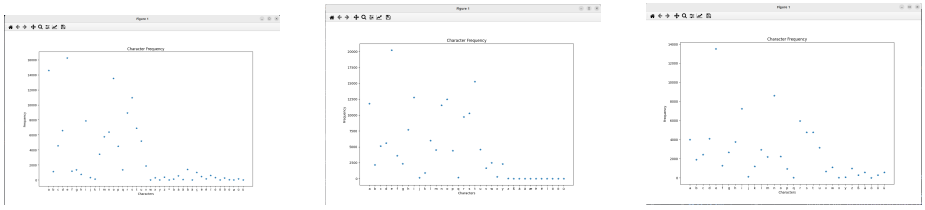
3.1 Text Data

During the text manipulation process, several notable patterns emerged.

Three languages were analyzed: two familiar (Portuguese and English) and one foreign (German). The foreign language shares its linguistic roots with English but differs significantly in structure and writing style from Portuguese.

The analysis utilized three selected texts, chosen randomly from six files available in the repository provided by the instructor. Among these files, we aimed to include one small, one medium, and one large file to compare metrics and analyze processing times across languages. This analysis involved generating graphs, assessing word and character frequency, and other metrics.

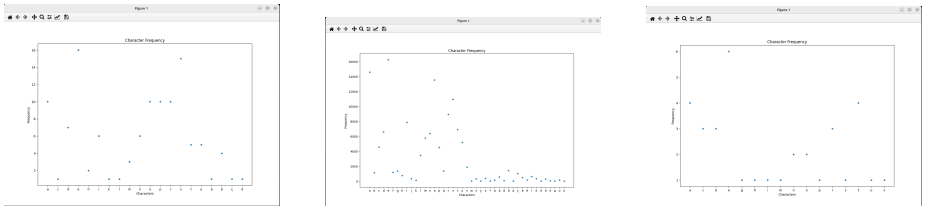
Starting with the letter frequency graphs, a clear pattern emerged across all languages: vowels, particularly "e" and "a," were among the most common letters, a predictable result given their frequent use in words. This result can be confirmed by the graphs below, based on the largest text.



(a) Portuguese (b) English (c) German

Figure 1: Letter frequency comparison across Portuguese, English, and German texts.

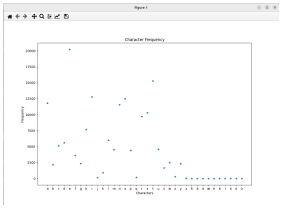
In Portuguese, the letters "a," "e," "i," and "s" were most common. Notably, "o" appeared less frequently despite its prevalence in masculine-gendered words. Additionally, accented characters such as "õ" and "à" were rare, hinting at a formal tone, as accents are often more prominent in informal or colloquial text (e.g., "Ó Maria, vai à padaria buscar pão"). This could imply that the chosen texts lean towards a formal or academic register, where accentuation may be less frequent. This can be seen in the graphs below, showcasing different texts.



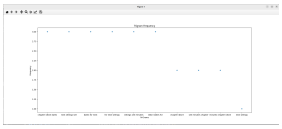
(a) Text 1 (b) Text 2 (c) Text 3

Figure 2: Letter frequency in Portuguese texts.

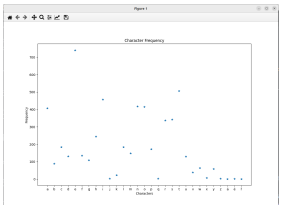
In English, "e," "i," and "t" appeared most often. The frequency of "t" was unexpected, as "h" was anticipated to be more prevalent due to its frequent appearance in common words like "with," "this," and "the." Less common letters included "v" and "z," with accented characters being rare and primarily appearing in borrowed words or as sound indicators (e.g., "ae"). This can be confirmed by the graphs below, showcasing the letter frequency in three different texts.



(a) Text 1



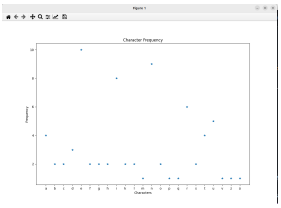
(b) Text 2



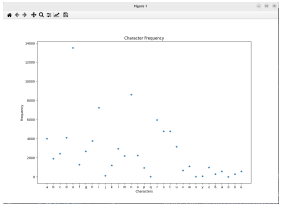
(c) Text 3

Figure 3: Letter frequency across three different English texts.

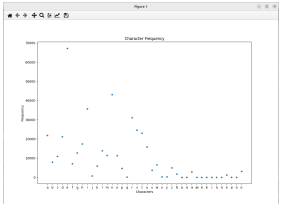
In German, "e," "a," and "n" were the most frequent letters, aligning with external sources even for non-German speakers. However, "v," "z," and "u" were the least used, despite "zu" being a frequently used preposition in German. This can also be seen in the following graphs.



(a) Text 1



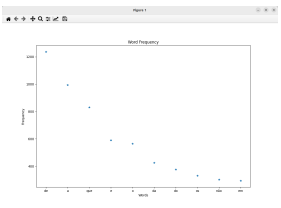
(b) Text 2



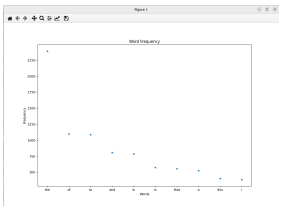
(c) Text 3

Figure 4: Letter frequency across three different German texts.

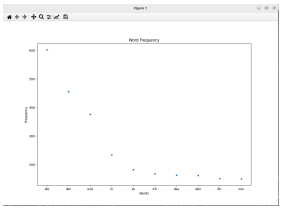
When analyzing words, the most frequent words in all three languages were typically determiners ("o," "a," "the," "die") and prepositions ("de," "in," "of," "von"). Less common words were usually proper nouns. Only graphs representing significant frequency variations were considered; those with uniform frequency distributions were disregarded. This can be observed in the following graphs for each language.



(a) Portuguese



(b) English



(c) German

Figure 5: Most common words across Portuguese, English, and German texts.

For bigrams, which are pairs of consecutive elements, patterns provided insight into text structures and usage. In Portuguese, bigrams frequently began

with prepositions, such as "de." In English, the definite article "the" frequently appeared at the beginning of bigrams. In German, prepositions like "die" or "der" typically followed nouns. This can be analyzed in the graphs below.

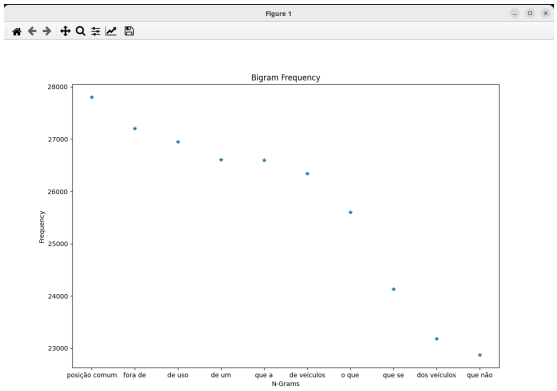


Figure 6: Bigram analysis for Portuguese texts.

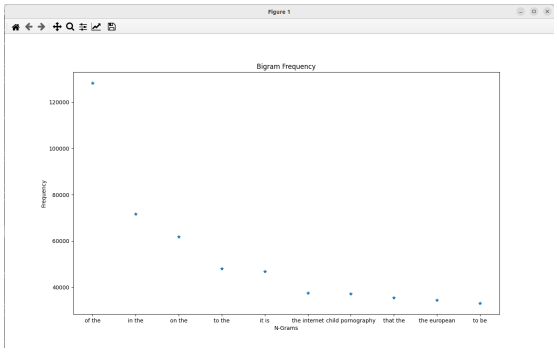


Figure 7: Bigram analysis for English texts.

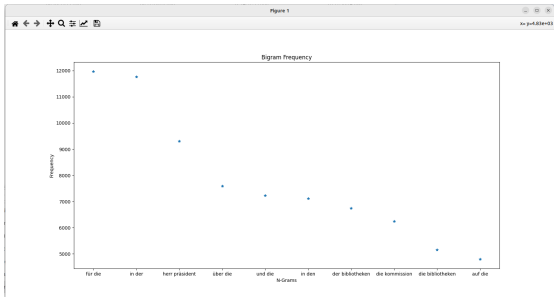


Figure 8: Bigram analysis for German texts.

For trigrams (sequences of three elements), patterns followed similar trends. Portuguese trigrams often combined a preposition with two nouns, while English trigrams included common expressions, such as "in order to" or "the European Union." German trigrams frequently included prepositions like "von" or "die." This can also be shown by the graphs below.

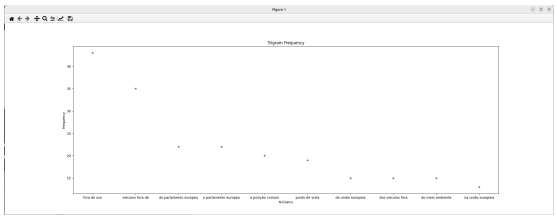


Figure 9: Trigram analysis for Portuguese texts.

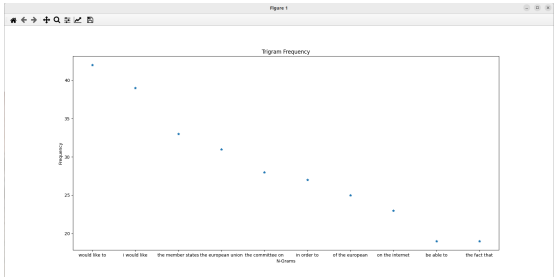


Figure 10: Trigram analysis for English texts.

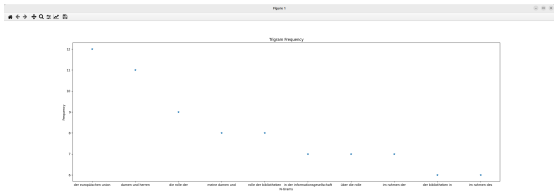


Figure 11: Trigram analysis for German texts.

Entropy

In addition to analyzing letter, word, and phrase frequencies, we looked at entropy to understand how predictable or complex each language is. Entropy measures the amount of information in language data and tells us how efficiently each language communicates.

By comparing entropy in Portuguese, English, and German, we gain insight into how predictable each language is in terms of structure and word usage.

Portuguese generally has higher entropy because it uses a wide range of word endings to show gender, number, and verb forms. This variation makes Portuguese words less predictable, adding complexity and increasing entropy.

English, with simpler verb forms and a fixed word order, tends to have lower entropy. There is less variation in word forms, which makes English slightly more predictable. However, English still has a large vocabulary and flexible phrasing, which can raise entropy in more complex texts.

German, with its case system and tendency to combine words, has a unique mix of structure and unpredictability. German uses compound words that convey specific meanings, which adds unpredictability. German's grammatical cases also require word forms to change based on their role in a sentence, increasing entropy compared to English.

In summary, Portuguese and German have higher entropy due to their richer grammatical structures, while English, with simpler word forms, generally has lower entropy.

Conclusion

In sum, the analysis done of letter and word frequency, along with entropy, revealed some key differences between Portuguese, English, and German. While all three languages show similar patterns in letter frequency—like the common use of vowels "e" and "a"—entropy tells us more about the structural differences.

Higher entropy in Portuguese and German reflects their greater complexity and variation in word forms, while English's simpler structure gives it lower entropy, making it slightly easier to process.

These findings suggest that languages with more complex grammar, like Portuguese and German, require more resources for text processing. Higher entropy makes compression and natural language processing tasks more challenging for these languages compared to English. This helps to understand how different language structures affect computational tasks, and it offers useful insights for fields like machine translation and language modeling.

3.2 Audio Data

3.2.1 Waveform

The file which was the main subject for our testing was `sample04.wav`, since it was the shortest of the provided samples, which accelerated the testing and developing of the program, given that, the first thing we did was extracting its waveform, something that gave us a base for the work presented below.

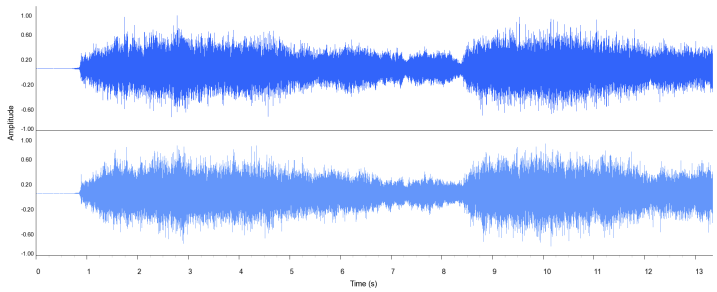


Figure 12: Waveform of the 2 channels present in `sample04.wav`.

3.2.2 Frequency Spectrum

As the next step of our development, we worked on the frequency-spectrum graph, as it helped the analysis of dominant frequencies as-well as the interpretation of the harmonic content (since the file is a music).

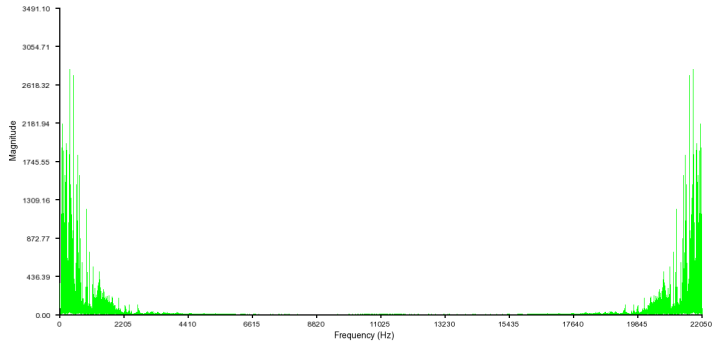


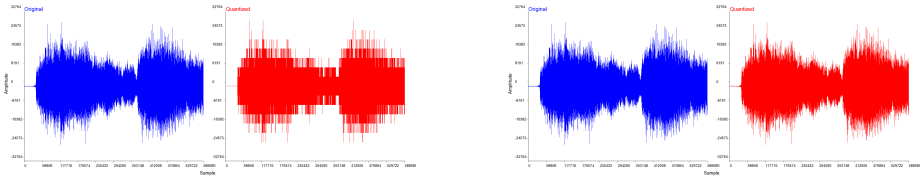
Figure 13: Frequency-Spectrum graph of `sample04.wav`.

3.2.3 Quantization

Quantization is a very well known way of compressing audio information with losses. In the case of our program, it was clear that a quantized audio had

some differences, especially when compressing it more. In a 4 bit quantization the `Mean Squared Error was 1.29277e+06`, which means that there was quite some difference between the audios, which was clearly audible aswell, the `Signal to Noise Ratio was 2.36107e+07`, expressed as a ratio, this indicates that the original signal still prevails, in spite of the losses.

However, when we use 16 bits for the signal quantization, the differences are clear, the `Mean Squared Error is 0.499129`, this value is so close to 0, that compressing the file like this does not yeld any perceivable loss in audio quality, this can also be expressed by the much higher `Signal to Noise Ratio of 6.1153e+13`, that indicates that the noise is much less noticeable in relation to the sound.

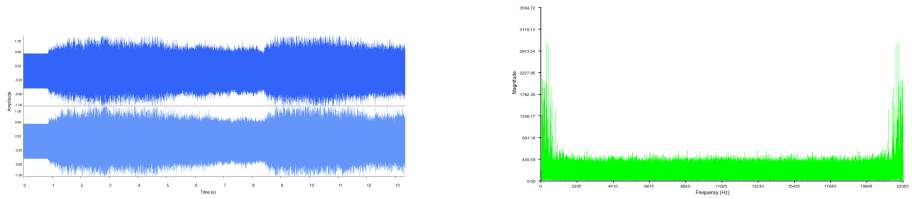


(a) 4 bits quantization. (b) 16 bits quantization.

Figure 14: Histograms of the different channels in `sample04.wav`

3.2.4 Noise Adding

When random noise to the audio, the results are as can be expected, not only did the audio change to the naked ear, but the `Signal to Noise Ratio plummeted to 338590`, this meant that the noise was noticeable and not only that, if we interpret the noise as errors, something we do, the `Mean Squared Error is now 9.01481e+07`, indicating some grotesque loss of data.

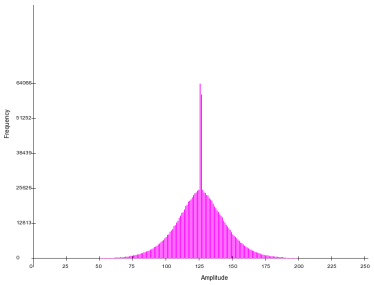


(a) Noisy Waveform Graph. (b) Noisy Frequecy-Spectrum Graph

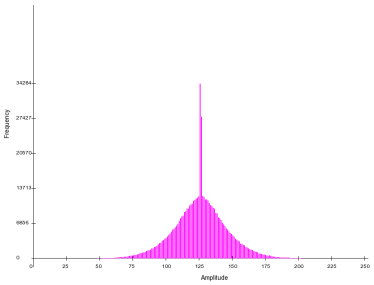
Figure 15: Histograms of the different channels in `sample04.wav`

3.2.5 Audio Channels

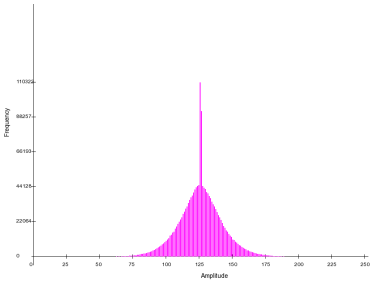
In respect to the audio channels the data we were able to extract can be resumed into the histograms below, the channels present some similarities, despite not being completely equal, which in the end, can help convey different information through its raw data.



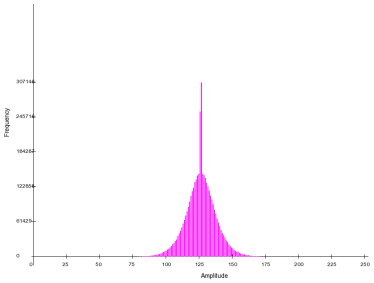
(a) Left Channel



(b) Right Channel



(a) Middle Channel



(b) Side Channel

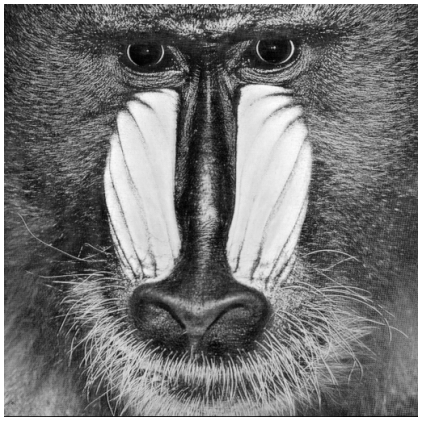
Figure 17: Histograms of the different channels in `sample04.wav`

3.3 Image Data

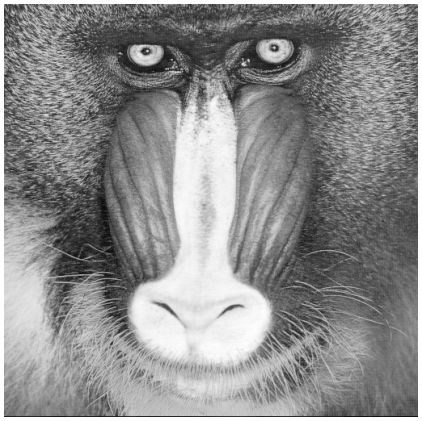
3.3.1 Color Manipulation

In the following figure we can observe the original image we loaded and the images prodduced from each color channel separately.

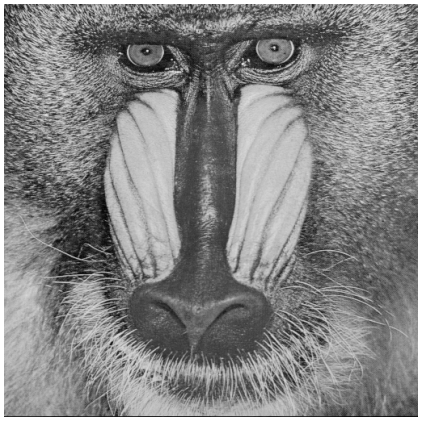
(a) Original Image



(b) Blue Channel



(a) Green Channel



(b) Red Channel

Figure 19: Loaded image and respective color channels

3.3.2 Gaussian Blur

With **task 4** we were able to conclude blurring a image does not affect its pixel density and/or size, occupying the same size as the original, with a bigger kernel resulting in a more noticeable and spread blur.

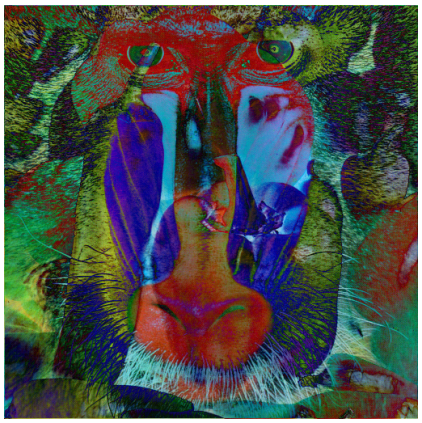
(a) Gaussian blur with 7x7 kernel

(b) Gaussian blur with 13x13 kernel

Figure 20: Different sized kernel blurs

3.3.3 Image Difference

(a) Second image

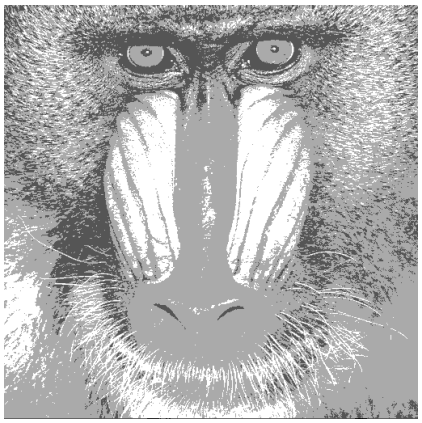


(b) Difference image

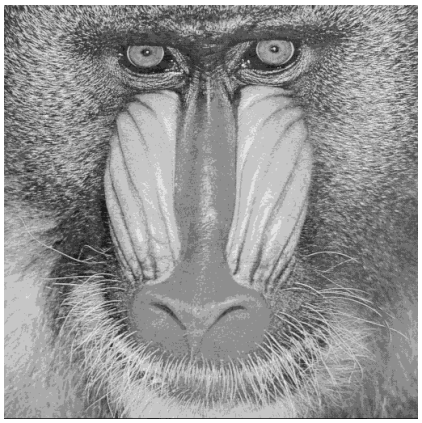
Figure 21: Difference between 2 images

3.3.4 Quantization

While quantization is an effective method of compression, excessive scaling will result in a visible loss in image quality, as can be concluded from our obtained images.



(a) Quantized image with 4 levels



(b) Quantized image with 10 levels

Figure 22: Different levels of quantization

Regarding MSE and PSNR, using quantization with 4 levels we obtained a MSE of 2585.04 and a PSNR of 14.0061, and with 10 levels we obtained a MSE of 267.402 and a PSNR of 23.8592. With this we can confirm our previous knowledge that lower image quality translates to a higher MSE and loer PSNR.

4 Performance Analysis

Through this analysis, we assess the performance of various functions applied to text, audio, and image data, comparing their execution times and exploring the computational demands of different tasks. The data spans across multiple datasets, including both small and large inputs, allowing us to observe how processing time scales with data size and complexity.

Text Processing

Text processing involved functions such as counting characters, words, and generating n-grams (2-grams and 3-grams). The key observations from the performance analysis are:

- n-gram generation consistently requires the most processing time across all datasets, especially as the text size increases. This is due to the complexity of iterating through the text to form multiple overlapping word pairs or triplets.
- Functions such as `applyTransformation`, `countCharacterOccurrence`, and `countWordOccurrence` are relatively fast, even with larger datasets, showing that these operations scale well with text size.

- `readFile` remains efficient, indicating that text loading does not pose a significant bottleneck in comparison to the analysis tasks.

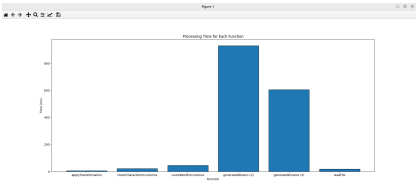


Figure 23: Performance for Large Text in Portuguese

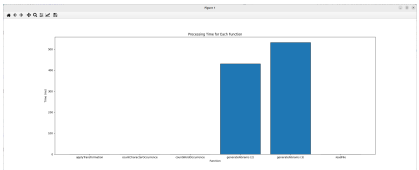


Figure 24: Performance for Small Text in Portuguese

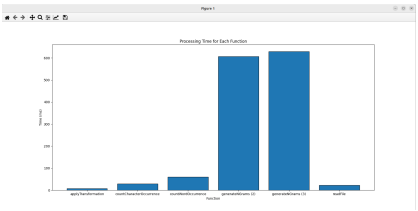


Figure 25: Performance for Large Text in English

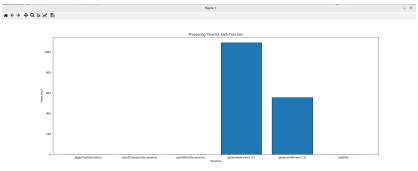


Figure 26: Performance for Small Text in English

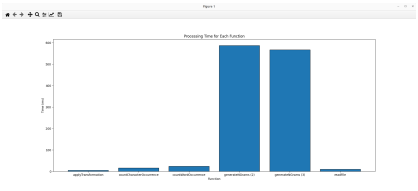


Figure 27: Performance for Large Text in German

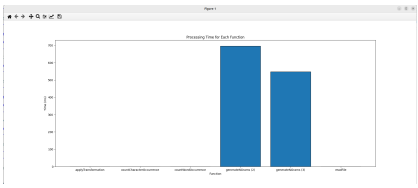


Figure 28: Performance for Small Text in German

The data demonstrates that computational costs increase significantly for n-gram generation as dataset size grows, which could pose challenges for real-time applications or larger datasets.

Audio Processing

For audio data, we measured the time taken by several key functions, including reading the file, playing the audio, extracting audio information, and performing waveform and histogram plotting. For this experience we observed a 13 second audio vs a 10 minute song. The results show that:

- `playAudio` and `plotAudioWaveform` are the most time-consuming functions, which can be attributed to the complexity of rendering and playing back the audio in real-time.
- Simpler tasks, such as extracting audio information (`getAudioInfo`) or reading the file (`readFile`), are relatively quick, even with larger datasets.
- `getLeftRightMidSideChannels` functions exhibit moderate processing time as they require decomposing stereo audio into separate channels.
- Quantization takes a consistent amount of time regardless of the number of levels used, suggesting that this operation scales linearly with the data size.

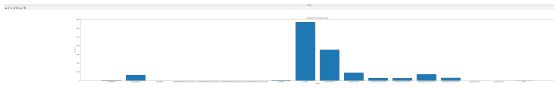


Figure 29: Performance for Small Audio (13 seconds)

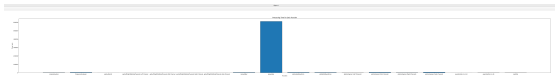


Figure 30: Performance for Large Audio (10 minutes)

These results indicate that while audio file handling and basic operations are computationally light, real-time playback and visualization (like waveform and histogram plotting) significantly increase the processing time, particularly with larger datasets. In the larger audio we can verify that given its size, none of the operations took as much time as the audio being played.

Image Processing

Image processing tasks include operations like loading an image, converting it to grayscale, applying Gaussian blur, and quantizing the image at different levels. For this experience we used a 2*2 pixel image and a 1000*1000 image to showcase the difference. The performance analysis shows:

- `calculateHistogram` and `gaussianBlur` consume the most time, particularly with larger images, due to the complexity of computing histograms and applying convolution filters.
- Functions like `toGrayscale` and `splitImage` have relatively low processing times, even for larger images, indicating that these operations are efficient and scale well.

- Quantization of images, particularly with a higher number of levels (e.g., 10 levels), increases the processing time compared to lower levels (e.g., 4 levels). However, the time increase is proportional to the complexity of the task.

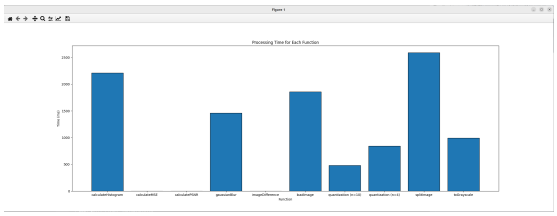


Figure 31: Performance for Small Image (2*2 pixels)

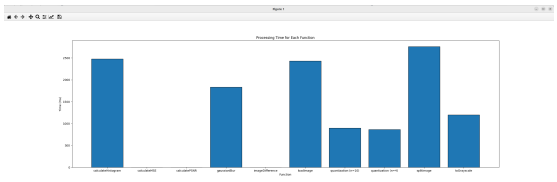


Figure 32: Performance for Large Image (1000*1000 pixels)

These findings suggest that while basic transformations such as gray scale conversion or channel splitting are computationally light, more complex operations like blurring and histogram calculation demand more processing power, especially with high-resolution images.

Conclusion

Across all three types of data (audio, image, and text), the processing demands vary significantly depending on the complexity of the operation and the size of the input data.

In text processing, tasks such as n-gram generation are particularly impacted by the size of the dataset, with performance deteriorating as the text size increases. This highlights a clear sensitivity to scaling, especially in tasks involving multiple iterations over large text bodies.

For image processing, more complex operations like histogram calculation and Gaussian blur also exhibit increased processing times as the dataset size grows. However, simpler transformations, such as converting to grayscale or splitting image channels, remain computationally efficient, even when dealing with larger images.

Audio processing, on the other hand, is generally less sensitive to dataset size. However, functions like real-time playback and waveform visualization (e.g.,

`playAudio` and `plotWaveform`) are more demanding and take up considerably more processing time compared to simpler tasks like file reading or channel extraction.

This analysis highlights the trade-offs between computational complexity and data size across different types of media, underlining the importance of optimizing more demanding functions, particularly in scenarios where real-time or large-scale processing is critical.

5 Discussion

During this project, we identified both strengths and limitations in our implemented methods. A major strength was the broader understanding we gained of processing different types of media—text, images, and audio—and the computational demands unique to each. Although we previously worked primarily with C, adapting to C++ was manageable, especially with the helpful documentation provided by the professor. This project allowed us to observe performance differences across these media types and the trade-offs involved in processing time and efficiency. Another strength was the precise time measurements for each function, which enabled detailed performance comparisons and highlighted areas for future improvement. By visualizing these time measurements with `matplotlib` graphs, we could easily spot outliers and resource-intensive tasks, greatly aiding in streamlining performance analysis.

However, some limitations became apparent. Scalability issues were significant, as certain methods struggled to handle larger datasets efficiently, particularly in text and image processing. Additionally, the lack of real-time optimization in audio processing created a bottleneck, as we had to wait for the entire audio file to play out, which slowed down testing and analysis. The single-threaded nature of the program further contributed to slower performance, as the absence of multithreading or parallelism limited the program from utilizing modern CPU capabilities to run tasks concurrently.

To address these limitations, one improvement would be to implement multithreading or parallelism, which could significantly reduce processing times by allowing multiple functions to run simultaneously. Enhanced memory management and real-time audio optimization would also be beneficial, particularly for functions like `playAudio` and `plotWaveform`. Using buffering techniques or chunk-based processing could help manage large audio files more effectively.

The learning experience from this project was invaluable. We learned the importance of optimizing computational time when working with larger datasets, especially in image and text processing, where performance can degrade quickly. Integrating `matplotlib` into a C++ project was challenging, as it required working with both C++ and Python code—an unfamiliar combination for us. Additionally, using the SFML interface for audio processing posed its own difficulties, particularly in managing axis, tick marks, and labels, which was more complex than `matplotlib`'s straightforward API.

Despite these challenges, we successfully implemented media processing and

performance comparisons, using graphical insights to understand how different media types are handled. This experience not only deepened our understanding of media processing but also underscored the potential for future optimization, especially in real-time and large-scale applications.

6 Conclusion

In summary, this project provided valuable insights into data manipulation, compression techniques, and quality analysis across different media types in C++. We found that certain tasks, such as n-gram generation in text processing and Gaussian blur in image processing, are inherently resource-intensive, particularly with larger datasets. The integration of `matplotlib` for data visualization enabled us to observe the computational impact of each task, helping us identify optimization needs, especially for real-time or large-scale applications.

Working with larger datasets underscored the importance of computational efficiency. Integrating `matplotlib` and handling Python within a C++ project broadened our understanding of cross-language library usage, presenting both challenges and learning opportunities. Despite these challenges, the project succeeded in our exploration of media processing and performance analysis in C++.

This experience has shown us not only how different types of media—text, images, and audio—are processed but also the transformations that can be applied to each, highlighting the trade-offs in quality, compression, and processing time. In addition, working extensively with C++ allowed us to recognize its unique strengths and challenges compared to other programming languages, enriching our programming knowledge and skill set.

7 Appendices

- C++ Reference
- C++ Tutorial - Tutorialspoint
- C++ Vectors - W3Schools
- C++ tolower Function - Naukri
- Lambda Expression in C++ - GeeksforGeeks
- Matplotlib C++ Documentation
- Matplotlib Accented Text - Matplotlib
- SFML - Simple and Fast Multimedia Library
- OpenCV - Open Source Computer Vision Library