Information and Coding - Project#01

Universidade de Aveiro(UA)

Departamento de Electrónica, Telecomunicações e Informática($DETI$)

Mestrado em Engenharia de Computadores e Telemática ($MECT$)

Project#01

Authors

Anderson Lourenço - 108579
aaokilourenco@ua.pt

António Almeida - 108250
ant.almeida@ua.pt

Tomás Laranjo - 107542
tomas.laranjo@ua.pt

October 25, 2024

Abstract

This project serves as a refresher on data manipulation techniques using C++, divided into three core sections that focus on distinct data types: text, audio, and images.

For text data, the emphasis is on reading, processing, and analyzing content through methods such as frequency analysis and basic transformations like case normalization and punctuation removal. Additionally, character entropy for each Latin alphabet is calculated. The audio section includes loading, visualizing, and manipulating audio files, dividing them into different waveforms targeting specific channels, as well as applying quantization and evaluating metrics like Mean Squared Error (MSE) and Signal-to-Noise Ratio (SNR). Lastly, image processing tasks involve reading and manipulating image files, generating histograms, applying filters, and performing image quantization. The outcomes are represented through various plots, histograms, and image comparisons, offering insights into the effectiveness of these transformations.

Chapter 1

Introduction

This project focuses on manipulating various forms of data—text, audio, and images—using C++ programming. The objective is to develop skills in handling, processing, and analyzing different types of data using algorithms and tools tailored for each data format. The project is divided into three main parts, each addressing a specific data type.

1. **Text Data Manipulation:** This section involves reading and processing text files to extract meaningful statistics like word and character frequencies. Basic transformations such as text normalization (lowercase conversion and punctuation removal) are also performed to facilitate accurate analysis. It is also calculated the entropy of each character.

2. **Audio Data Manipulation:** The objective in this section is to read audio files, extract their samples, and analyze the waveform and amplitude distribution. To gain a clearer understanding of the audio format, it is divided into separate channels. More advanced tasks involve performing audio quantization and comparing the original and modified audio files using metrics such as Mean Squared Error (MSE) and Signal-to-Noise Ratio (SNR).

3. **Image Data Processing:** Image manipulation tasks include loading and visualizing images, calculating pixel histograms, applying filters, and performing image quantization. The effects of these operations are evaluated using visual comparisons and metrics like MSE and Peak Signal-to-Noise Ratio (PSNR).

The project emphasizes understanding how data transformations impact quality and compression, providing insights into the trade-offs between efficiency and data fidelity.

Chapter 2

Data Manipulation

2.1 Part I - Text Data Processing

For this part we learnt how to manipulate text data programmatically doing the following tasks utilizing the *string* library .

2.1.1 Task 1

On the first exercise we learnt how to open a text file reading it line by line by doing the following:

```
    while (std::getline(file, line)) {
        fileContent.push_back(line);
    }
```

Note: $fileContent.push_back(line)$*; adds the line that was just read to the fileContent vector. The* $push_back$ *method of the std::vector class appends the new element to the end of the vector, dynamically resizing the vector as needed. This allows the program to store all lines from the file in the fileContent vector, preserving the order in which they were read.*

and displaying the information on the terminal, to make sure the information is read correctly, handling errors.

Inside the main function we started by checking if the filename is provided by using a condition that checks if, when running the program, the user passes two arguments such as file name.

```
    if (argc < 2) {
        std::cerr << "Error: No file name provided!" << std::endl;
        std::cerr << "Usage: " << argv[0] << " <file_name>" << std::endl;
        return 1;
    }
```

For reading the text files, is first constructed the file path just for project organization, storing the file name in a variable as a string and concatenating several strings.

```
    std::string fileName = argv[1];
    std::string filePath = "./data/" + fileName + ".txt";
```

The program has an instance of std::ifstream (class provided by the C++ Standard Library for reading from files) where its constructor takes a single argument containing the path to the file the user wants to open. When this line is executed, the file specified by filePath is opened in read mode. If the file cannot be opened (for example, if it does not exist or the program lacks the necessary permissions), the std::ifstream object will be in a fail state. You can check this state later in your code to handle any errors that might occur during file opening. This approach is useful for reading data from files in a structured and efficient manner, leveraging the capabilities of the C++ Standard Library.

```
    std::ifstream file(filePath);
```

If the file cannot be opened it outputs an error message to the standard error stream (std::cerr), informing the user that the file could not be opened. This message helps in diagnosing issues such as incorrect file paths, missing files, or insufficient permissions.

```
    if (!file.is_open()) {
        std::cerr << "Error: Could not open the file!" << std::endl;
        return 1;
    }
```

To help completing this task and the following ones the program contains three variables that are essential for handling and processing file content in a C++ program.

```
    std::vector<std::string> fileContent;
    std::vector<std::string> transformedContent;
    std::string line;
```

First, std::vector¡std::string¿ fileContent; declares a std::vector named fileContent that will store strings. In this context, each string in the vector is likely to represent a line from a file. The std::vector class is part of the C++ Standard Library and provides a dynamic array that can grow as needed, making it suitable for storing an unknown number of lines from a file.

Next, std::vector¡std::string¿ transformedContent; declares another std::vector named transformedContent. This vector is intended to store the transformed or processed lines of the file. The transformation could involve various operations such as filtering, modifying, or formatting the lines read from the file. By keeping the original and transformed content in separate vectors, the program can maintain a clear distinction between the raw input and the processed output.

Finally, std::string line; declares a std::string variable named line.

Together, these variables set up the necessary data structures for reading lines from a file, processing them, and storing both the original and transformed content. This approach is common in programs that need to perform text processing tasks, such as data analysis, file conversion, or content filtering.

2.1.2 Task 2

On the second exercise the key features of the code are **Text Transformations**:

- **Lowercasing:** The function ***toLowerCase*** converts all characters in the text to lowercase. This ensures case consistency, so words like "Apple" and "apple" are treated as the same word.
- **Removing Punctuation:** The function ***removePunctuation*** strips out punctuation marks. This normalization ensures that characters such as commas or periods do not interfere with word frequency analysis.

The program effectively normalizes the text by converting everything to lowercase and removing punctuation, which is a necessary preprocessing step for statistical analysis. This ensures consistency when counting word occurrences.

Here is the code sample that's in charge of the key features:

```
for (const auto& line : fileContent) {
    std::string lowerCaseLine = toLowerCase(line);
    std::string transformedLine = removePunctuation(lowerCaseLine);
    transformedContent.push_back(transformedLine);
}
```

2.1.3 Task 3

Jumping to task 3 we have the following features:

1. **Iterating through the text:** The function ***countCharacterFrequencies*** iterates over each line of the transformed text (which has been normalized by converting to lowercase and removing punctuation). It examines each character in the text, excluding whitespace, and counts its occurrence using a std::map¡char, int¿.

2. **Storing Character Frequencies:** The std::map¡char, int¿ data structure stores each unique character as a key and the number of occurrences as its associated value. This ensures that the frequency count of each character is accurately tracked, with efficient lookups and updates.

3. **Printing Character Frequencies:** After counting the characters, the program prints the frequency of each character to the console. This provides an easily interpretable breakdown of how often each character appears in the text. Plotting Character Frequencies:

Although not mandatory, the program utilizes the matplotlibcpp library to generate a plot that visually represents the frequency of characters. The characters are plotted along the x-axis, while their frequencies are plotted along the y-axis. This graphical output (that will be further analyzed) offers a clearer view of the distribution, allowing the user to immediately identify dominant characters and see patterns.

Function for handling this task:

```
std::map<char, int> countCharacterFrequencies(const std::vector<std::string
>& content) {
std::map<char, int> charFrequency;
for (const auto& line : content) {
    for (char c : line) {
        if (!std::isspace(c)) { // ignore whitespace
            charFrequency[c]++;
        }
    }
}

// Prepare data for plotting
std::vector<char> characters;
std::vector<int> frequencies;
for (const auto& pair : charFrequency) {
    characters.push_back(pair.first);
    frequencies.push_back(pair.second);
}

// Plotting
```

```cpp
    plt::figure_size(1200, 800);
    plt::plot(characters, frequencies);
    plt::title("Character Frequency");
    plt::xlabel("Characters");
    plt::ylabel("Frequency");
    plt::grid(true);
    plt::show();

    return charFrequency;
}
```

The software efficiently calculates character frequencies by utilizing a ***std::map*** to count the occurrences of each character in the text. It handles the input by iterating over each line of the transformed text, ensuring that punctuation and whitespace are excluded from the analysis. This approach allows for an accurate breakdown of meaningful characters, such as letters and numbers. The printed frequency results provide users with a clear and detailed view of the character distribution, which can be useful for text analysis or cryptographic purposes. Furthermore, the program enhances the readability of the data by plotting the frequency distribution, allowing for an intuitive visual representation of the most and least common characters. This combination of clear printed output and optional graphical visualization makes the program versatile for users who need both detailed and high-level views of character frequency patterns.

2.1.4 Task 4

The software effectively implements the task by breaking down the text into words, counting their occurrences, and visualizing the results. Here's how the program handles the task:

1. **Tokenizing the Text:** The software uses the ***countWordFrequencies*** function to tokenize the transformed text. It reads each line and splits it into individual words using whitespace as the delimiter. This is achieved with an istringstream object, which separates words based on spaces. By processing the text line by line and word by word, the program ensures that each word is identified and isolated for further analysis.

2. **Storing Word Frequencies:** The program uses a std::map¡std::string, int¿ to store the frequency of each word. As it processes the text, each word is either added to the map (if it's the first occurrence) or its count is incremented (if the word has been encountered before). This data structure efficiently keeps track of unique words and their frequencies, allowing for fast lookups and updates during the word counting process.

3. **Printing Word Frequencies:** After counting all words in the text, the software prints the frequency of each word to the console. This output provides users with a detailed view of how often each word appears in the text, helping to identify common or repeated terms.

4. **Visualizing Word Frequencies:** The program also optionally uses the *matplotlibcpp* library to plot the word frequencies. Instead of plotting words directly (which could make the graph cluttered), it indexes each word and plots its frequency based on this index. This gives users a clear visual representation of the word frequency distribution, making it easier to spot patterns, such as high-frequency or rare words.

Function for handling this task:

```cpp
std::map<std::string, int> countWordFrequencies(const std::vector<std::string>&
    content) {
    std::map<std::string, int> wordFrequency;
    for (const auto& line : content) {
        std::istringstream stream(line);
        std::string word;
        while (stream >> word) {
            wordFrequency[word]++;
        }
    }

    // Prepare data for plotting
    std::vector<int> indices;
    std::vector<int> frequencies;
    int index = 0;
    for (const auto& pair : wordFrequency) {
        indices.push_back(index++);
        frequencies.push_back(pair.second);
    }

    // Plotting
    plt::figure_size(1200, 800);
    plt::plot(indices, frequencies);
    plt::title("Word Frequency");
    plt::xlabel("Word Index");
    plt::ylabel("Frequency");
    plt::grid(true);
    plt::show();

    return wordFrequency;
}
```

The frequency counts are accurately tracked and displayed to the user, offering detailed insights into word usage within the text. Additionally, the software's use of a plotting library to visualize these frequencies provides a clearer understanding of word distribution patterns. The use of an efficient data structure ensures that word counting can scale even with larger texts, while the transformation steps (such as converting to lowercase and removing punctuation) ensure that the results are meaningful and not distorted by case differences or punctuation marks.

2.1.5 Calculate Entropy

The implementation of character entropy calculates the entropy of characters within the text, which measures the randomness or unpredictability of the character distribution. Entropy is a key concept in information theory, and in this case, it helps quantify how much information is carried by the text's character distribution.

The formula used for Entropy is based on shannon entropy:

$$E = -\sum p(x)\log_2 p(x)$$

Steps in the Code:

- Character Frequency Calculation: First, the program uses the ***countCharacterFrequencies*** function to determine how many times each character appears in the transformed text. This is stored in a ***std::map¡char, int¿***.

- Total Character Count: The program sums up the total number of characters (excluding spaces) in the text. Probability Calculation: For each character, the probability

  $$p(x)$$

  is calculated as the frequency of that character divided by the total number of characters.

- Entropy Formula: The entropy is then computed by summing up the product of each character's probability and the logarithm (base 2) of that probability, using the formula above. The logarithmic function is implemented as ***std::log2()***.

What the Software Achieves:

1. **Quantifying Text Predictability:** The entropy value gives a quantitative measure of how predictable or random the character distribution in the text is. Higher entropy values suggest that the text has a diverse character distribution with less repetition, indicating a high level of randomness or unpredictability. Conversely, lower entropy suggests more repetition and predictability. For example, a text with many repeated characters (such as "aaaaaa") will have low entropy, while a text with a mix of diverse characters (such as a sentence or paragraph from a book) will have higher entropy.

2. **Efficient Calculation:** By using a std::map to store character frequencies and iterating over the characters only once to calculate probabilities, the code ensures that the entropy calculation is both time- and space-efficient. This makes it scalable for large texts.

3. **Practical Use Cases:**

   - **Text Compression:** Entropy is a critical concept in compression algorithms. Higher entropy indicates that the text carries more information per character, meaning it would be harder to compress. Conversely, low-entropy texts, which are more predictable, are easier to compress since they contain redundant information.

   - **Cryptographic Analysis:** In cryptography, entropy can indicate how secure or complex a cipher is. Higher entropy in ciphertext suggests more secure encryption, as it is less predictable and harder to crack.

   - **Linguistic and Data Analysis:** Entropy can also be used in linguistic analysis to understand the structure of a language or the complexity of a piece of text. Texts with higher entropy might indicate richer or more varied use of language, while lower entropy might suggest repetitive or simpler language.

2.2 Part II- Audio Data Manipulation

This section centers on the analysis and manipulation of audio data using the SFML library for processing. The primary goal is to understand the structure of audio data and how to manage it effectively. The tasks range from reading and loading audio files, visualizing the waveform, separating different channels, and applying quantization techniques. The processed signals are then compared to the original using metrics like Mean Squared Error (MSE) and Signal-to-Noise Ratio (SNR). These exercises provide fundamental skills necessary for more advanced work in audio compression, enhancement, and analysis.

2.2.1 Task 1

By utilizing the SFML library to process audio files, a .wav file can be read by creating a *sf::SoundBuffer* variable. This variable calls the *loadFromFile()* function, which transfers the file's data into the buffer.

Furthermore, information such as raw audio samples, sample count, sample rate, and channel count is extracted from the buffer. This data is then used to calculate the audio's duration using the formula: $\text{duration} = \frac{\text{sampleCount}}{\text{sampleRate}\times\text{channelCount}}$.

2.2.2 Task 2

At this stage, the buffer remains in memory, and only the necessary information is extracted for each specific task. In this case, the samples, sample count, and sample rate are retrieved as needed.

Using this data, we generate a waveform plot with the help of the *matplotlibcpp* library, where the x-axis represents time and the y-axis indicates the amplitude at each point.

Additionaly, to validate the results, we compare the plotted graph with open-source software, specifically Audacity.

2.2.3 Task 3

In this section, we obtained the samples, sample count, channel count, and sample rate from the buffer to generate histograms for each channel: left, right, middle, and side.

Considering the distribution on a stereo audio, the samples from the left channel are all on the $2 * i$ index of the buffer, while the samples from the right channel are on the $2 * i + 1$. To calculate the middle channel we used the formula (leftChannel + RightChannel)/2, the side channel used the formula (leftChannel-rightChannel)/2

We defined the bin size using a simple variable set to 2^{numBits}, which can be adjusted as needed to accommodate the programmer's requirements.

2.2.4 Task 4

For this, we got the Samples, Sample Count and Sample rate. Since the audio is only 16 bits, the quantization threshold indicates that the maximum value that can be assigned to the numBits representing each audio sample is 15, in other words, 2^{numBits}-1 .

The *quantizedSamples* vector, as its name implies, holds the values for each sample after quantization. It is created using the formula:

$$\text{round}\left(\frac{\text{originalSamples}[i]}{\text{maxVal}} \times \text{numLevels}\right) \times \left(\frac{\text{maxVal}}{\text{numLevels}}\right)$$

After the quantization, we plot both the original Samples as the quantized Samples to see the diference, if numBits is 15, the plots should be the same

2.2.5 Task 5

To compare two audio signals, we employed the Mean Squared Error (MSE), which measures the average squared difference between the corresponding sample points of the two signals. Additionally, we used the Signal-to-Noise Ratio (SNR), which assesses the level of noise introduced in the processed signal relative to the original.

To calculate MSE, subtract each sample of the second audio from the corresponding sample of the first audio, square the differences, sum them up, and then divide by the total number of samples

$$\text{mse} = \frac{\sum_{i=0}^{\text{count}-1} (\text{original}[i] - \text{processed}[i])^2}{\text{count}}$$

To calculate SNR, find the ratio of the power of the original signal to the power of the noise (the difference between the original and processed signals). This is often expressed in decibels (dB) using the formula:

$$\text{SNR (dB)} = 10 \cdot \log_{10} \left(\frac{\text{Power of Signal}}{\text{Power of Noise}} \right)$$

To calculate the Power of Signal and Power of Noise use the formulas:

$$\text{Power of Signal} = \sum_{i=0}^{\text{count}-1} (\text{original}[i])^2$$

$$\text{Power of Noise} = \sum_{i=0}^{\text{count}-1} (\text{original}[i] - \text{processed}[i])^2$$

2.3 Part III - Image Data Processing

This section of the project involves the manipulation and analysis of image data, using a standard library in the performance of computer vision tasks known as OpenCV. The different tasks that are involved in this proposal deal with reading images, visualizing image channels, calculating histograms to display results, filtering, comparison between images, and quantization. Each of these tasks gives a different perspective on how digital images are represented and preprocessed for many applications.

2.3.1 Task 1

This task involves loading an image from a file and displaying it using OpenCV. The image is read in color mode using $\boxed{\text{cv::imread()}}$ and is displayed in a window with $\boxed{\text{cv::imshow()}}$. This method introduces basic image manipulation by enabling the visualization of images in different formats. In real-world applications, this forms the foundation for further image analysis, as image acquisition is the first step in any vision-related project. Successfully loading and displaying the image ensures that the system can process image data.

2.3.2 Task 2

Task 2, delves deeper into image processing by splitting an image into its three color channels: red, green, and blue. The $\boxed{\text{cv::split()}}$ function separates the image into these channels, which are then individually displayed. The method of visualizing the color channels helps us understand how each color contributes to the final image. Additionally, the image is converted to grayscale using $\boxed{\text{cv::cvtColor()}}$ to demonstrate a simpler representation of the image. Grayscale conversion is commonly used in image processing tasks where color information is not critical, simplifying the data and speeding up processing times.

2.3.3 Task 3

Task 3, introduces the calculation and visualization of image histograms. A histogram provides a graphical representation of the pixel intensity distribution in an image. Using $\boxed{\text{cv::calcHist()}}$, the pixel intensities of the grayscale image are calculated and then plotted using matplotlibcpp to visualize the distribution. The histogram helps identify key features of the image, such as contrast, brightness, and dynamic range. This information is crucial for applications such as image enhancement, thresholding, and segmentation.

2.3.4 Task 4

Task 4, applies Gaussian blur filters with varying kernel sizes are applied to the image. The function $\boxed{\text{cv::GaussianBlur()}}$ smooths the image, reducing noise and detail by averaging pixel values. This technique is commonly used in preprocessing to reduce noise in images before further analysis or to highlight more significant structures. By applying different kernel sizes, the effect of blurring on the image is clearly visible, with larger kernels leading to stronger smoothing. This experiment illustrates the trade-off between noise reduction and loss of detail in the image.

2.3.5 Task 5

Task 5, focuses on comparing two images using the absolute difference method. The $\boxed{\text{cv::absdiff()}}$ function calculates the pixel-by-pixel differences between the two input images, providing a new image where the differences are highlighted. Additionally, Mean Squared Error

(MSE, this metric measures the average squared differences between corresponding pixel values of the two images) and Peak Signal-to-Noise Ratio (PSNR, this metric is derived from MSE and provides a logarithmic measure of the image quality) are computed to quantify the similarity between the images. These metrics are valuable in tasks such as image compression, where the quality of the compressed image is compared with the original. A low MSE and a high PSNR indicate minimal differences and good quality retention.

2.3.6 Task 6

The final task involves quantizing the grayscale image to reduce the number of bits required for storage and display. Using uniform scalar quantization, pixel values are mapped to a reduced set of intensity levels. The effect of reducing the number of quantization levels is observed, and the quality of the image is evaluated using MSE and PSNR. This task demonstrates the trade-off between image quality and storage efficiency, as reducing the number of bits decreases the file size but increases the quantization error, which can degrade the image quality.

2.3.7 Usage

The compilation and execution of Part 3 is straightforward. After running the following commands:

```
cmake ..
make
```

inside the ./build directory, you can execute the program by using the following command:

```
./P3 <file_name> <file_name>
```

Note that for tasks requiring only one image, the first file provided in the command will be used. The second file will be ignored for such tasks.

2.3.8 Discussion

The implemented methods in Part 3 worked effectively for image processing tasks, including image visualization, filtering, and error measurement. Key strengths include ease of implementation and successful outcomes for tasks like channel separation and filtering.

However, limitations include handling edge cases in MSE and PSNR calculations, and the current quantization process could offer more flexibility. Potential improvements involve parallelizing operations for faster processing, adding advanced error metrics like SSIM, and refining the input system for better usability.

These optimizations could enhance both performance and user experience in future iterations.

Chapter 3

Analyses and Results

3.1 Part I - Text Data Processing

Processing times varied across tasks, with **T3** (character frequency counting) taking the longest due to detailed iteration through each character, particularly when combined with graphical visualizations. In contrast, **T4** (word frequency counting) was quicker, as it required only tokenizing by whitespace, leading to faster computations.

In **T3**, transforming text to lowercase and removing punctuation ensured accurate character frequency calculations. The visualization of these frequencies revealed clear patterns, though it added to processing time. Despite this, T3 provided deep insights into character distributions.

T4 was faster because of its focus on word-level analysis. The word frequency graph showed common words and phrases, though stop words were not excluded. Its faster processing time made it more efficient for larger texts, despite the slightly reduced granularity compared to character-level analysis.

The **entropy calculation** revealed that texts with diverse character distributions had higher entropy, indicating greater unpredictability. In contrast, repetitive texts had lower entropy, reflecting higher predictability. While less computationally demanding than frequency counting, entropy added valuable insight into text complexity.

Overall, there is a balance between processing time, granularity, and information richness. **T3** provides deeper insights into character distributions but requires more time due to the per-character analysis and visualization. In contrast, **T4** offers quicker results for word-level patterns but may overlook finer details like common word filtering. The entropy analysis, while efficient, adds valuable insights into the text's complexity, showing a trade-off between data uniformity and information content. Future improvements could focus on optimizing the visualization steps in both T3 and T4 to reduce processing time, or exploring methods to filter out non-essential elements like stop words in word frequency analysis. Parallel processing or more efficient data structures might further enhance the performance of these text processing tasks.

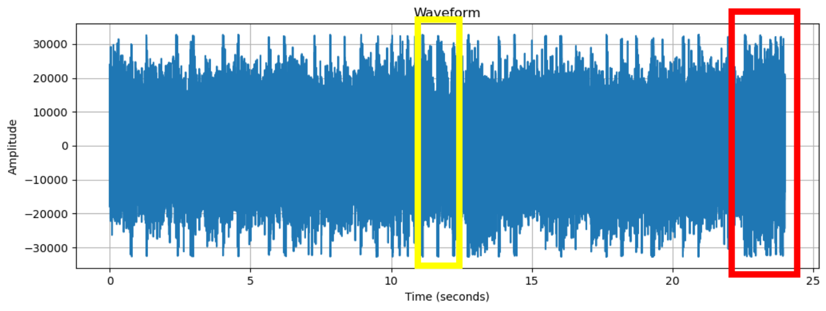
The tasks executed in Part 1 yielded various outcomes in analyzing text data, including character and word frequency analysis, as well as entropy calculation. These operations play a crucial role in understanding the underlying structure of text and assessing its complexity. Below is a summary of the key results:

- T2: Basic text normalization (lowercase conversion, punctuation removal) ensured cleaner data for further analysis.
- T3: Character frequency counting provided detailed distributions, with certain letters showing clear patterns reflective of language tendencies. The visualization helped interpret these patterns, though it increased processing time.
- T4: Word frequency analysis highlighted common words, showing repetition patterns efficiently. The graph provided insight into the text's structure, though stop words may have skewed the results.
- Entropy Calculation: Texts with more diverse characters had higher entropy, indicating complexity. Uniform texts had lower entropy, reflecting predictability.

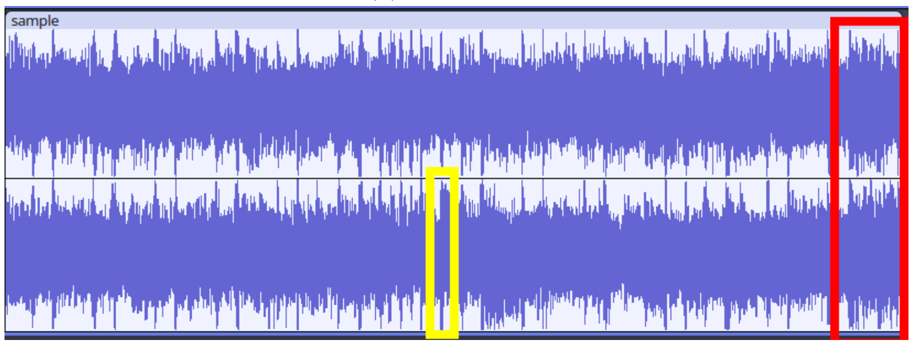
Visualizations, including character and word frequency graphs, were generated to provide a clearer understanding of these analyses. These visual tools enhance the interpretation of the results by displaying frequency trends and patterns that might be harder to discern from raw data alone. Overall, the results demonstrate how different forms of text analysis can uncover both surface-level and deeper structural characteristics of the text, with entropy offering an additional layer of insight into the complexity and information content of the data.

3.2 Part II - Audio Data Manipulation

With nothing significant to report regarding **T1 (Reading and loading audio files)**, **T2 (Visualizing the audio waveform)** is more engaging, as it displays both the plotted waveform and the waveform as shown by Audacity.



(a) Code image

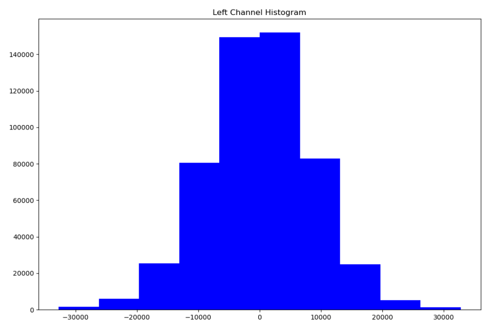


(b) Audacity image

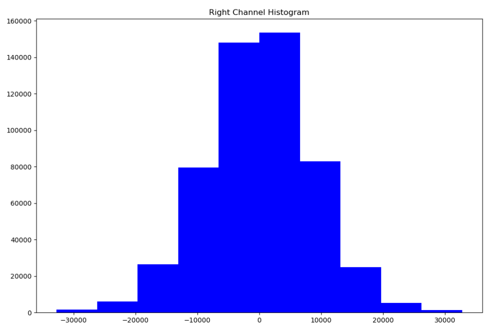
Figure 3.1: T2 Waveforms

Although it may be difficult to spot, the resemblance between them is noticeable, with a spike in amplitude appearing at the end (red bar) and in the middle (yellow bar).

In **T3 (Histogram of amplitude values)**, it is evident that mid-range amplitudes are more frequently used compared to extreme values.



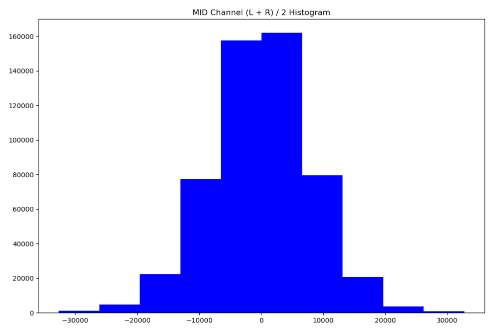
(a) Left Channel



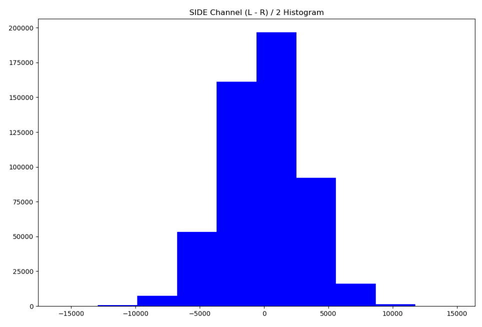
(b) Right Channel

Figure 3.2: T3 Histograms

Moreover, the mid and side channels exhibit the same pattern, with mid-range amplitudes being more prevalent than extreme ones.



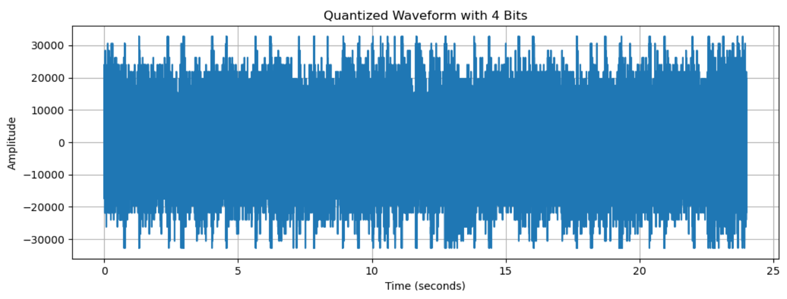
(a) Middle Channel



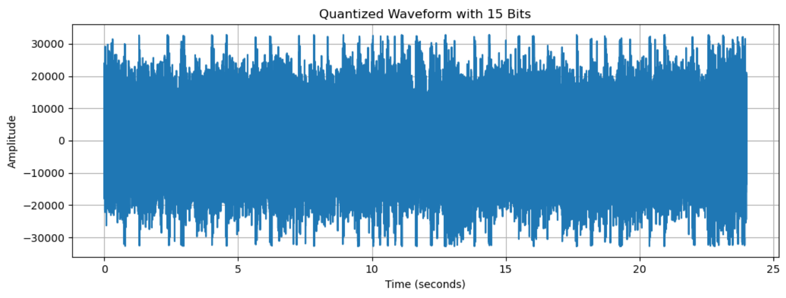
(b) Side Channel

Figure 3.3: T3 Histograms

With the **T4 (Quantization of audio)**, we can see the reduced number of distinct amplitude levels



(a) 4 Bits waveform



(b) 15 Bits waveform

Figure 3.4: T4 Histograms

In **T5 (Comparing two audio samples)**, the comparison between the original audio and the 4-bit quantized audio yields a Mean Squared Error of 397605 and a Signal-to-Noise Ratio of 22.7002 dB. This indicates good signal quality, with the signal being significantly stronger than the noise. However, the MSE reflects some error between the original and processed signals, though it remains within a tolerable range. Overall, this suggests that the signal processing or reconstruction maintains acceptable quality, with adequate accuracy and clarity.

3.3 Part III - Image Data Processing

Processing times varied across tasks, with T3 (resizing) taking the longest at 7.16 seconds, while T2 (histogram calculation) was the quickest at 0.84 seconds. T6 (quantization) completed in 1.35 seconds, showing that more complex transformations like quantization and filtering tend to increase time complexity. This highlights the computational demands of more advanced image manipulations compared to basic operations.

In T5, image quality, measured by PSNR, decreased slightly after transformations, with values of 24.6 dB for Image 1 and 24.07 dB for Image 2. Despite this minor quality reduction, T5 maintained an efficient processing time of 1.71 seconds, indicating a good balance between processing speed and acceptable quality degradation. The difference image, with an MSE of 251.19, reflected the subtle changes introduced by these transformations.

In T6, quantization had minimal impact on perceived image quality, with a PSNR of 57.8 dB and an MSE of 50,384. While the MSE is relatively high, the high PSNR suggests that the image quality loss is not perceptible to the human eye, demonstrating the effectiveness of the quantization method used.

- T1 through T4 showed processing times ranging from 0.84 to 7.16 seconds, with resizing and filtering demonstrating consistent performance.
- T5 highlighted a slight image quality degradation after transformations, with MSE values around 223 for Image 1 and 254.86 for Image 2. The difference image had an MSE of 251.19 and a PSNR of 24.13 dB.
- T6 revealed that quantization resulted in minimal quality loss, as indicated by an MSE of 50,384 and a PSNR of 57.8 dB.

Overall, these results demonstrate a trade-off between processing time, compression efficiency, and image quality. Higher compression ratios, as seen in T6, resulted in a small drop in image quality, though this loss is offset by computational savings. On the other hand, more intensive transformations, like resizing in T3, took significantly more time, emphasizing the need to optimize for efficiency when processing larger or more complex images.

Future optimizations should focus on reducing processing times while maintaining quality, particularly for tasks that involve larger images or complex transformations. Investigating parallel processing or more efficient algorithms could help maintain the quality seen in tasks like T6 while achieving faster performance.

Chapter 4

Conclusion

To wrap up this lab project, it is essential to emphasize the key outcomes and lessons gained from implementing and analyzing the manipulation of text, audio, and image data.

Throughout this project, various essential techniques were applied to different types of data. For text, fundamental transformations were conducted, including the analysis of character and word frequencies along with their entropy, as well as text normalization through case conversion and punctuation removal. These steps underscored the significance of pre-processing in making the text more uniform and easier to analyze.

In the audio section, tasks like waveform visualization, histogram analysis, and quantization provided insights into how changes in sample precision influence audio quality. Metrics like Mean Squared Error (MSE) and Signal-to-Noise Ratio (SNR) were used to measure these effects, offering a deeper understanding of how lossy compression affects signal clarity.

Similarly, for image data, transformations such as histogram calculations, image filtering, and quantization offered hands-on experience with compression techniques and their visual effects. Like with audio, metrics such as Mean Squared Error (MSE) and Peak Signal-to-Noise Ratio (PSNR) were used to better understand the trade-offs between compression levels and image quality.

Overall, this lab offered a valuable hands-on exploration of data manipulation across various formats, with a particular focus on compression, quality assessment, and the role of visualization in tracking data transformations.