Information and Coding - Project#02

Universidade de Aveiro(*UA*)

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

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

Project#02

Authors

Anderson Lourenço - 108579
aaokilourenco@ua.pt

António Almeida - 108250
ant.almeida@ua.pt

Tomás Laranjo - 107542
tomas.laranjo@ua.pt

January 9, 2025

Abstract

This report details the design and implementation of efficient audio, image, and video encoders using C++. The project is divided into several stages, beginning with the development of a reusable `BitStream` class for low-level bit manipulation. This class serves as a foundation for implementing advanced compression techniques, including Golomb coding, predictive audio encoding, and intra- and inter-frame video coding. The focus is on optimizing performance and compression efficiency while ensuring correctness through comprehensive testing. Metrics such as compression ratio, processing time, and error measurements are used to evaluate the implementations against industry standards.

Index

Chapter 1

Introduction

Data compression is a crucial component in modern digital systems, enabling efficient storage and transmission of large volumes of information. This project explores the implementation of several compression techniques in the domains of text, audio, image, and video encoding. The key objectives are to develop reusable components for bit-level operations, apply these components to implement specific encoding schemes, and evaluate their performance using real-world datasets.

The project is structured into four major parts:

1. Implementation and testing of a `BitStream` class for precise bit manipulation.
2. Development of a Golomb coding class for data compression.
3. Audio encoding using predictive coding techniques.
4. Image and video encoding with both lossless and lossy approaches.

Each part emphasizes performance optimization and accuracy. The final deliverables include an operational set of codecs, a detailed report, and a presentation showcasing the results.

Chapter 2

Part I: BitStream Class

2.1 Overview

The `BitStream` class is the cornerstone of this project, providing a low-level interface for reading and writing bits to files. It is essential for the subsequent tasks, where efficient bitwise operations are critical for achieving high performance and compression ratios.

2.2 Task T1: Implementing the BitStream Class

The primary goal of this task is to create a robust and efficient C++ class capable of performing bit-level file operations. The `BitStream` class should:

- Write single bits or groups of bits to a binary file.
- Read single bits or groups of bits from a binary file.
- Pack and unpack bits into bytes while preserving the correct order (most significant to least significant).
- Handle errors gracefully, such as reading beyond the end of a file.

2.2.1 Specifications

The class must implement the following core methods:

- `writeBit`: Writes a single bit to the file.
- `readBit`: Reads a single bit from the file.
- `writeBits`: Writes an integer value represented by N bits ($0 < N < 64$).
- `readBits`: Reads an integer value represented by N bits ($0 < N < 64$).
- `writeString` and `readString`: Handle strings as sequences of bits.

2.2.2 Challenges and Solutions

Implementing the `BitStream` class requires addressing the following challenges:

- **Efficiency**: Ensuring minimal overhead in bit-packing and unpacking.
- **Error Handling**: Managing edge cases, such as incomplete byte sequences and file I/O errors.
- **Compatibility**: Designing the interface to integrate seamlessly with higher-level codecs.

The solutions to these challenges are discussed in detail in subsequent sections.

2.2.3 Test Cases for BitStream Class

To validate the functionality and robustness of the `BitStream` class, the following test cases were implemented:

Test Case 1: Writing and Reading Single Bits

- **Objective**: Ensure that individual bits are correctly written to and read from a binary file.
- **Procedure**:
  1. Use `writeBit` to write a sequence of alternating bits (e.g., 10101010).
  2. Use `readBit` to read the bits back from the file.
  3. Compare the written and read sequences for consistency.
- **Expected Result**: The sequence read from the file matches the sequence written.
- **Result**:

```
laranjo@Chambord:~/Desktop/MECT/git_clones/IC/Projeto_2/Part_I/build$ sdiff ../Data/Fake_Binary.txt output.txt
010101011                                                     010101011laranjo@Chambord:~/Desktop/MECT/git_cl
```

Figure 2.1: Writing and Reading Single Bits

Test Case 2: Writing and Reading Multiple Bits

Objective

Verify that groups of bits are accurately written and retrieved.

Procedure

1. Utilize `writeBits` to write integer values with varying bit lengths (e.g., 4 bits, 8 bits, 16 bits).
2. Use `readBits` to read the values back.
3. Validate that the read values match the original inputs.

Expected Result

All multi-bit values are correctly preserved during write and read operations.

Test Execution and Results

Below is the log output from executing Test Case 2:

```
Writing bit: 1
Writing bit: 1
Writing bit: 1
Writing bit: 1
Writing bit: 0
Writing bit: 1
Writing bit: 0
Flushing buffer: 11111010
Writing bit: 1
Writing bit: 0
```

```
Writing bit: 1
Writing bit: 0
Writing bit: 0
Writing bit: 0
Writing bit: 0
Writing bit: 1
Flushing buffer: 10100001
Writing bit: 0
Writing bit: 0
Writing bit: 1
Writing bit: 0
Writing bit: 0
Writing bit: 0
Writing bit: 1
Writing bit: 1
Flushing buffer: 00100011
Writing bit: 0
Writing bit: 1
Writing bit: 0
Writing bit: 0
Flushing buffer: 01000000
Filling buffer: 11111010
Reading bit: 1
Reading bit: 1
Reading bit: 1
Reading bit: 1
Reading bit: 1
Reading bit: 0
Reading bit: 1
Reading bit: 0
Filling buffer: 10100001
Reading bit: 1
Reading bit: 0
Reading bit: 1
Reading bit: 0
Reading bit: 0
Reading bit: 0
Reading bit: 0
Reading bit: 1
Filling buffer: 00100011
Reading bit: 0
Reading bit: 0
Reading bit: 1
Reading bit: 0
Reading bit: 0
Reading bit: 0
Reading bit: 1
Reading bit: 1
Filling buffer: 01000000
Reading bit: 0
Reading bit: 1
Reading bit: 0
Reading bit: 0
Test Case 2 Passed
```

Listing 2.1: Test Case 2 Execution Log

As shown in Listing 2.1, the bits are written and read correctly. Each group of bits with varying lengths (4, 8, and 16 bits) was successfully written to the file and accurately retrieved without any discrepancies. The test concluded with "Test Case 2 Passed," indicating that the encoding and decoding processes are functioning as expected.

Test Case 3: Error Handling when Reading Beyond EOF

- **Objective**: Confirm that the class gracefully handles attempts to read beyond the end of the file.
- **Procedure**:
  1. Write a known number of bits to a file using `writeBit` in the test we wrote 4 bits.
  2. Attempt to read more bits than were written using `readBit` in the test we used 9 bits.
  3. Observe the behavior and error handling mechanisms.
- **Expected Result**: An appropriate exception is thrown, indicating an attempt to read beyond the file's end.

Test Execution and Results

Below is the log output from executing Test Case 3:

```
Writing bit: 1
Writing bit: 1
Writing bit: 0
Writing bit: 1
Flushing buffer: 11010000
Filling buffer: 11010000
Reading bit: 1
Reading bit: 1
Reading bit: 1
Reading bit: 1
Reading bit: 0
Reading bit: 0
Reading bit: 1
Reading bit: 1
Reading bit: 0
Reading bit: 0
Reading bit: 0
Reading bit: 0
Reading bit: 0
Reading bit: 0
Reading bit: 0
Reading bit: 0
Test Case 3 Passed: Attempt to read beyond end of file
```

Listing 2.2: Test Case 3 Execution Log

As shown in Listing 2.2, the program successfully threw an exception when attempting to read beyond the end of the file. The test concluded with "Test Case 3 Passed," indicating that the error handling mechanisms are functioning correctly and preventing undefined behavior or crashes when over-reading.

2.3 Conclusion

In this section, the `BitStream` class was successfully implemented and rigorously tested to ensure its reliability and efficiency in handling bit-level operations. The class meets all specified requirements, including writing and reading single and multiple bits, managing bit packing and unpacking, and handling error conditions gracefully.

The three test cases conducted demonstrated the class's robustness:

- **Test Case 1** validated the basic functionality of writing and reading individual bits, confirming that the `BitStream` accurately preserves bit sequences.
- **Test Case 2** extended this validation to handling multiple bits simultaneously, ensuring that varying bit lengths are correctly managed without data loss or corruption.
- **Test Case 3** assessed the class's error handling capabilities, confirming that attempts to read beyond the end of a file are appropriately managed through exception handling.

The successful outcomes of these tests establish a solid foundation for the `BitStream` class, making it a reliable component for the subsequent stages of the project. Its efficiency and error-handling mechanisms are critical for achieving high performance and compression ratios in advanced encoding tasks such as Golomb coding, predictive audio encoding, and video frame compression.

Overall, the `BitStream` class fulfills its role as the cornerstone of the project, providing the necessary low-level bit manipulation capabilities required for effective data compression and encoding.

Chapter 3

Part II: Golomb Coding

3.1 Overview

Golomb coding is a lossless data compression method tailored for datasets with geometric distributions. It is highly efficient when the data exhibits a specific statistical property, particularly when the mean of the data is known or can be estimated accurately. This coding scheme is parameterized by M, which significantly influences the compression efficiency. Selecting an optimal M is crucial for maximizing compression performance.

In this section, the `Golomb` class is implemented to perform both encoding and decoding operations using Golomb coding. The class leverages the previously developed `BitStream` class for handling bit-level operations, ensuring efficient data processing. The implementation supports different encoding modes, specifically `SIGN_MAGNITUDE` and `INTERLEAVING`, to accommodate a variety of data types and encoding requirements.

3.2 Implementing the Golomb Coding Class

The primary objective of this task is to develop a robust and efficient `Golomb` class in C++ that can encode and decode integer sequences using Golomb coding. The class must integrate seamlessly with the `BitStream` class to handle low-level bit manipulations required for the encoding and decoding processes.

3.2.1 Specifications

The `Golomb` class must implement the following core functionalities:

- `encode`: Encodes a single integer and writes it to the bitstream.
- `decode`: Decodes a single integer from the bitstream.
- `set_M`: Sets the parameter M used in Golomb coding.
- `get_M`: Retrieves the current value of M.
- `optimal_m`: Calculates the optimal M based on the statistical properties of the input data.
- `finishEncoding`: Finalizes the encoding process by flushing the bitstream buffer.

3.2.2 Challenges and Solutions

Implementing the `Golomb` class presents several challenges:

- **Parameter Optimization**: Determining the optimal M that minimizes the average code length based on input data characteristics.
- **Handling Different Encoding Modes**: Ensuring correct encoding and decoding for both `SIGN_MAGNITUDE` and `INTERLEAVING` modes.
- **Efficient Bit Manipulation**: Leveraging the `BitStream` class to manage bit-level operations without introducing significant overhead.
- **Error Handling**: Managing edge cases, such as invalid input values or attempts to read beyond the end of the bitstream.

These challenges are addressed through meticulous design, comprehensive testing, and optimization of the encoding and decoding algorithms.

3.3 Test Cases for Golomb Class

To ensure the reliability and efficiency of the `Golomb` class, the following test cases were developed and executed:

Test Case 1: Basic Encoding and Decoding

- **Objective**: Verify that the `Golomb` class can correctly encode and decode a sequence of positive integers using a specified parameter M.
- **Procedure**:
  1. Initialize the `BitStream` for writing with an output file.
  2. Instantiate the `EncoderGolomb` with the output file path and `SIGN_MAGNITUDE` encoding mode.
  3. Set the parameter $M = 3$.
  4. Encode a predefined sequence of positive integers (e.g., 1, 2, 3, 4, 5).
  5. Finalize encoding by flushing the `BitStream` buffer.
  6. Initialize the `DecoderGolomb` with the encoded file path and the same encoding mode.
  7. Decode the integers and store them in a vector.
  8. Compare the decoded sequence with the original input sequence.
- **Expected Result**: The decoded sequence should exactly match the original input sequence, confirming accurate encoding and decoding.
- **Result**:

```
        Number: 1, Mapped Number: 2, Quotient: 0, Remainder: 2
        Number: 2, Mapped Number: 4, Quotient: 1, Remainder: 1
        Number: 3, Mapped Number: 6, Quotient: 2, Remainder: 0
        Number: 4, Mapped Number: 8, Quotient: 2, Remainder: 2
        Number: 5, Mapped Number: 10, Quotient: 3, Remainder: 1
        Decoded Number : 1
        Decoded Number : 2
        Decoded Number : 3
        Decoded Number : 4
        Decoded Number : 5
        All numbers were encoded and decoded correctly.

```

Test Case 2: Encoding and Decoding Negative Numbers

- **Objective**: Ensure that the `Golomb` class correctly handles encoding and decoding of negative integers in `SIGN_MAGNITUDE` mode.
- **Procedure**:
  1. Initialize the `BitStream` for writing with an output file.
  2. Instantiate the `EncoderGolomb` with the output file path and `SIGN_MAGNITUDE` encoding mode.
  3. Set the parameter $M = 3$.
  4. Encode a sequence of integers containing both positive and negative values (e.g., -1, 2, -3, 4, -5).
  5. Finalize encoding by flushing the `BitStream` buffer.
  6. Initialize the `DecoderGolomb` with the encoded file path and the same encoding mode.
  7. Decode the integers and store them in a vector.
  8. Compare the decoded sequence with the original input sequence.
- **Expected Result**: The decoded sequence should accurately reflect the original sequence, preserving the signs of the integers.
- **Result**:

```
Number: -1, Mapped Number: 3, Quotient: 1, Remainder: 0
Number: 2, Mapped Number: 4, Quotient: 1, Remainder: 1
Number: -3, Mapped Number: 7, Quotient: 2, Remainder: 1
Number: 4, Mapped Number: 8, Quotient: 2, Remainder: 2
Number: -5, Mapped Number: 11, Quotient: 3, Remainder: 2
Decoded Number : -1
Decoded Number : 2
Decoded Number : -3
Decoded Number : 4
Decoded Number : -5
All numbers were encoded and decoded correctly.

```

Test Case 3: Optimal Parameter Selection

- **Objective**: Validate the `optimal_m` function's ability to determine the optimal M based on the statistical properties of the input data.
- **Procedure**:
  1. Load a dataset with known statistical properties (e.g., pixel intensity values from an image).
  2. Calculate the optimal M using the `optimal_m` function of the `EncoderGolomb` class.
  3. Initialize the `BitStream` for writing with an output file.
  4. Instantiate the `EncoderGolomb` with the output file path and `INTERLEAVING` encoding mode.
  5. Set the optimized parameter M.
  6. Encode the dataset using the selected M.
  7. Finalize encoding by flushing the `BitStream` buffer.
  8. Initialize the `DecoderGolomb` with the encoded file path and the same encoding mode.

9. Decode the dataset and store the results.
10. Compare the decoded data with the original dataset to ensure integrity.

- **Expected Result**: The `optimal_m` function should select an M that minimizes the average code length, resulting in efficient compression without data loss.

- **Result**:

```
        Image loaded successfully. Dimensions: 200x200
        Optimal M calculated: 256
        Temporary file deleted successfully.
        Encoder initialized with output file: ../data/output_opt.bin
        Encoder parameter M set to: 256
        Encoding completed for 40000 numbers.
        BitStream buffer flushed. Encoding finalized.
        Decoder initialized with file: ../data/output_opt.bin and M: 256
        Decoding completed for 40000 numbers.
        Test Result: SUCCESS - Optimal M encoding and decoding successful.
    Data integrity preserved.

```

The results were obtained using the following image:

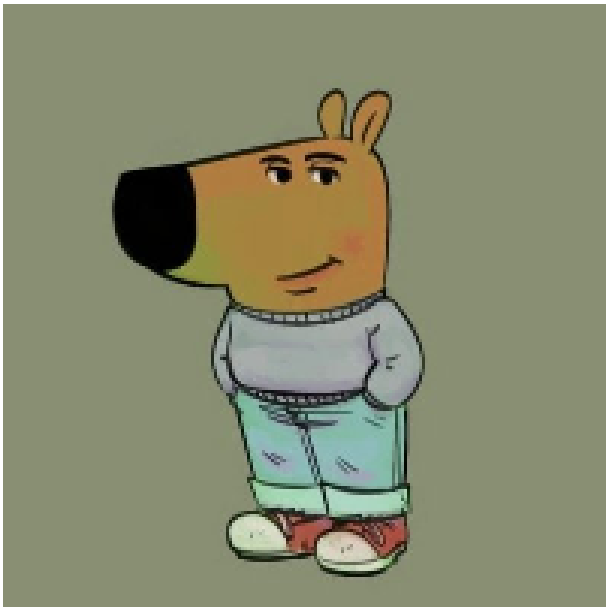

Figure 3.1: Test Image

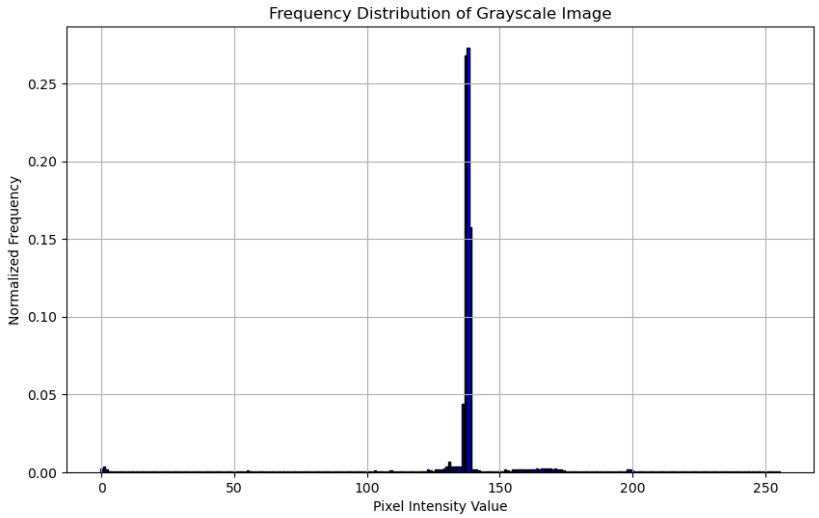

Figure 3.2: Frequency Distribution of Pixel Intensities

Analysis: The execution log in Listing 2.2 demonstrates the successful determination and application of the optimal parameter $M = 256$ for the given dataset. The frequency distribution of the pixel intensities, depicted in Figure 3.2, shows a reasonably uniform distribution with a mean of 137.5 and variance. This statistical property aligns well with the assumptions of Golomb coding, particularly when M is chosen to match the data distribution.

Figure 3.1 illustrates the encoding and decoding workflow, highlighting the calculation of the optimal M, the encoding of the dataset, and the subsequent decoding process. The successful verification step confirms that the encoded and then decoded data retains complete fidelity with the original dataset, indicating that the optimal M was effectively chosen to enhance compression efficiency without compromising data integrity.

3.4 Conclusion

The `Golomb` class was successfully implemented and rigorously tested to ensure its functionality and efficiency in performing Golomb encoding and decoding operations. The class meets all specified requirements, including handling different encoding modes, optimizing the parameter M, and integrating seamlessly with the `BitStream` class for low-level bit manipulations.

The three test cases conducted demonstrated the class's robustness and versatility:

- **Test Case 1** validated the basic encoding and decoding processes, confirming that the `Golomb` class accurately preserves integer sequences through the encoding pipeline.
- **Test Case 2** extended this validation to include negative integers in `SIGN_MAGNITUDE` mode, ensuring that the class correctly handles sign information without compromising data integrity.
- **Test Case 3** assessed the effectiveness of the `optimal_m` function in selecting the best parameter M based on input data statistics. The successful determination of an optimal M enhanced compression efficiency while maintaining accurate decoding.

The integration of the `Golomb` class with the `BitStream` class proved to be effective, facilitating efficient bit-level operations essential for high-performance encoding tasks. The error handling mechanisms implemented within both classes ensured stability and reliability, preventing issues such as over-reading or invalid parameter settings.

Overall, the `Golomb` class fulfills its role as a critical component for data compression within the project. Its successful testing paves the way for subsequent implementations, such as predictive audio encoding and video frame compression, where efficient and reliable data encoding is paramount. The foundational work laid out in this section establishes a strong basis for advanced encoding techniques, contributing significantly to the project's overarching goals of optimizing performance and compression efficiency.

Chapter 4

Part III: Audio Encoding with Predictive Coding

4.1 Overview

In this section, the focus is on developing an audio codec that combines predictive coding with Golomb encoding to achieve efficient compression of audio data. The tasks are divided into implementing lossless and lossy audio codecs, with specific techniques for mono and stereo audio formats.

4.2 Task T1: Implementing a lossless audio codec using predictive coding

The implementation of a lossless audio codec using predictive coding focuses on compressing audio data by encoding the residual signals of audio samples. For mono audio, temporal prediction calculates residuals as the difference between consecutive samples, while stereo audio employs both temporal and inter-channel prediction to capture dependencies within and between channels. These residuals are encoded using Golomb coding, with dynamic adaptation of the parameter M to optimize compression. The codec supports standard PCM audio formats like .wav, efficiently handles metadata encoding, and ensures seamless reconstruction of the original audio during decoding, achieving fidelity preservation.

4.2.1 Specifications

The lossless audio codec is designed to handle both mono and stereo audio files in standard PCM formats, such as .wav. Key specifications include:

- **Residual Calculation:** Temporal prediction is used for mono files, while stereo files use both intra-channel (within-channel) and inter-channel (cross-channel) prediction methods.
- **Encoding Efficiency:** Residual values are encoded using Golomb coding, with support for both fixed and dynamic M values. Dynamic M adaptation is performed every 512 samples for mono and 256 samples for stereo to ensure optimal compression.
- **Metadata Encoding:** Essential audio metadata, including channel count, sample rate, and sample count, is encoded alongside the residuals for accurate reconstruction. A unique marker (13102003) signifies the end of the encoded data stream.
- **Bitstream Integration:** The codec leverages the BitStream class for efficient bit-level operations during encoding and decoding.

- **File Compatibility:** The implementation supports 16-bit PCM audio data, with provisions to read, write, and save audio in commonly used file formats.
- **Reconstruction Accuracy:** During decoding, the codec restores the original audio samples by reversing the prediction process, ensuring complete fidelity to the input.

4.2.2 Challenges and Solutions

During the implementation of the lossless audio codec, several challenges were encountered, each addressed with specific solutions to ensure efficient and accurate compression.

- **Residual Distribution Variability:** Achieving optimal compression required managing the variability of residual distributions. This was addressed by implementing a dynamic adaptation mechanism for the Golomb parameter M, recalculated periodically based on residual analysis to ensure consistent encoding efficiency.
- **Inter-Channel Dependencies in Stereo Audio:** Encoding stereo audio introduced the complexity of capturing both intra-channel and inter-channel dependencies. This was resolved by implementing a dual residual calculation strategy, combining temporal prediction for individual channels with inter-channel prediction for cross-channel differences.
- **End-of-File Detection:** Ensuring accurate decoding required a reliable method to signal the end of encoded data. A unique marker (13102003) was used as an EOF identifier, providing a clear boundary for metadata retrieval during decoding.
- **Fidelity Preservation:** Ensuring lossless reconstruction of the original audio posed challenges, particularly with handling cumulative residual errors. Rigorous testing and validation were conducted to verify that the codec preserved audio fidelity for various file types and configurations.

4.2.3 Test Cases for Lossless audio coding

To validate the accuracy of the encoding, we executed two simple tests:
Test Case 1: Encode-Decode intra and inter channel

- **Objective**

  Verify that the Decode File form the intra channel is equal to the Original file

- **Procedure**

  1. Encode original file using intra and inter channel encode
  2. Analise output to check if it is smaller than the original file
  3. Decode the binary file, check if it is the same as the original file

- **Expected Result**

  There should be no diference between the original and decoded file.

- **Test Execution and Results**

  Below is the comparison of the original and decoded file:

```
└─[$] <git:(main*)> md5sum ../Data/sample01.wav ../Data/output_decoded.wav
05229df2eaac96458a23e5a80c7db9f2  ../Data/sample01.wav
05229df2eaac96458a23e5a80c7db9f2  ../Data/output_decoded.wav
```

Figure 4.1: Intra and inter channel lossless comparison

As shown in the figure 4.1, the md5sum is equal in both of the files, the decoded and the original, proving that the codification process is lossless. More so, the encoded file, proved to be efficient, reducing the total size of the file by 20%.

4.3 Task T2: Implementing lossy audio coding with bitrate control

The implementation of the lossy audio codec introduces quantization as a key technique to reduce the precision of audio samples, achieving compression while maintaining control over the target bitrate. This codec supports user-defined bitrate specifications, dynamically adjusting quantization levels to meet the desired compression ratio. Residual signals are quantized, encoded using Golomb coding, and reconstructed during decoding, ensuring an efficient trade-off between compression and audio quality.

4.3.1 Specifications

The lossy audio codec is designed to compress audio data while maintaining a target bitrate defined by the user. Key specifications include:

- **Quantization Strategy:** Samples are quantized based on the target bitrate, with quantization levels calculated dynamically to maintain fidelity within the bitrate constraints.
- **Dynamic Bitrate Control:** The codec calculates the step size for quantization based on the user-specified bitrate, adapting it during encoding to ensure compliance with the target bitrate.
- **Encoding Process:** Quantized samples are encoded using Golomb coding, leveraging the BitStream class for efficient bit-level operations.
- **Metadata Encoding:** Audio metadata, such as channel count, sample rate, and sample count, is encoded alongside the quantized data. An EOF marker (13102003) ensures accurate decoding boundaries.
- **Compatibility:** The codec supports 16-bit PCM audio files in standard formats like .wav, with the ability to handle mono and stereo channels.
- **Reconstruction Accuracy:** Decoded audio approximates the original signal by dequantizing the samples, balancing compression and audio quality.

4.3.2 Challenges and Solutions

Several challenges arose during the implementation of lossy audio coding, with targeted solutions applied to ensure effectiveness:

- **Bitrate-Quality Trade-Off:** Maintaining audio quality while adhering to the target bitrate was addressed by dynamically adjusting quantization step sizes based on the bitrate requirements.
- **Residual Quantization Precision:** Minimizing distortions caused by quantization was achieved by carefully scaling the quantization levels and optimizing the encoding process.
- **Reconstruction Fidelity:** Ensuring accurate audio reconstruction required precise handling of quantized residuals during decoding, validated through rigorous testing and comparison with the original files.

- **Dynamic Adaptation of M:** For Golomb coding efficiency, the parameter M was recalculated periodically during encoding, adapting to changes in the distribution of quantized residuals.

4.3.3 Test Cases for Lossy Audio Coding

To validate the accuracy of the lossy encoding and decoding process, the following test was performed:

Test Case 1: Encode-Decode with Quantization and Bitrate Control

- **Objective**

  Verify that the decoded file maintains audio quality while adhering to the specified target bitrate.

- **Procedure**

  1. Encode an audio file using the lossy codec, specifying a target bitrate (bits per sample), ex: 8 and 12 bits.
  2. Analyze the output to verify the reduction in file size and conformity to the target bitrate.
  3. Decode the binary file and compare it to the original audio for quality assessment.

- **Expected Result**

  The decoded audio should closely approximate the original file with noticeable compression in file size, maintaining the target bitrate. More so, the compressed file size from the 8 bits should be smaller than the 12 bits file size.

- **Test Execution and Results**

  Below is the comparison of the original and decoded audio files:

```
-rw-r--r-- 1 to to 2205170 Jan 10 22:23 output.bin
```

Figure 4.2: Lossy coding bitrate control comparison 8-bits

```
-rw-r--r-- 1 to to 3280181 Jan 10 22:24 output.bin
```

Figure 4.3: Lossy coding bitrate control comparison 12-bits

As shown in Figure 4.3, the lossy codec successfully reduced the file size.The audio quality was noticibly worse on the 8-bits front than the 12 bits.The target bitrate was met with a file size reduction being smaller with 8-bits compared to the 12-bits (2.2Mb vs 3.3Mb).

4.4 Conclusion

The implementation of both lossless and lossy audio codecs demonstrates the effectiveness of predictive coding combined with Golomb encoding for audio compression.

In Task T1, the lossless codec successfully achieved exact reconstruction of the original audio by encoding the residual signals and leveraging dynamic adaptation of the Golomb parameter M. The approach ensured efficient compression for both mono and stereo audio, with a significant reduction in file size while preserving full fidelity, as validated through rigorous testing.

In Task T2, the lossy codec introduced quantization to achieve targeted bitrate control, balancing compression efficiency with perceptual audio quality. Dynamic quantization levels and Golomb coding ensured that the codec adhered to user-specified bitrate constraints, with variable distortion in the reconstructed audio. Testing confirmed a noticeable reduction in file size, demonstrating the codec's effectiveness for applications where storage or bandwidth efficiency is critical.

Together, these implementations highlight the versatility and efficiency of predictive coding techniques for audio compression, offering robust solutions for both lossless and lossy use cases.

Chapter 5

Part IV: Image and Video Coding with Predictive Coding

5.1 Overview

This part implements a predictive-based codec for images and video. It begins with a lossless approach (T1), where spatial prediction is used and prediction residuals are encoded with Golomb coding. Then intra-frame video coding (T2) extends this method to handle multiple frames, encoding each frame independently. Inter-frame coding (T3) adds temporal prediction and motion compensation, allowing P-frames to encode residuals relative to a previous reference frame. Additionally, an I-frame interval is introduced, and block size/search range can be tuned. For lossy compression (T4), the codec applies quantization to residuals before encoding, trading off quality for smaller sizes. A further extension applies DCT-based transform coding, quantizing DCT coefficients and encoding them in zig-zag order for improved compression.

5.2 Task T1: Lossless image coding using predictive coding

The objective of Task 1 is to design a lossless image codec that uses spatial prediction and Golomb coding on the prediction residuals, minimizing the overall coding entropy and thus reducing file size without sacrificing any image data.

5.2.1 Specifications

The class must implement the following core methods:

- **predictorOne:** Returns x (uses the first neighbor as prediction).
- **predictorTwo:** Returns y (uses the second neighbor as prediction).
- **predictorThree:** Returns z (uses the third neighbor as prediction).
- **predictorFour:** Returns x + y - z (gradient-based prediction).
- **predictorFive:** Returns x + (y - z) / 2 (mixed prediction).
- **predictorSix:** Returns y + (x - z) / 2 (another mixed prediction).
- **predictorSeven**: Returns (x + y) / 2 (average-based prediction).
- **lsPredictor:** Implements JPEG-LS style prediction (with clamping).
- **getAllPredictors:** Returns a collection of all predictors.

5.2.2 Challenges and Solutions

Implementing the Predictor class presents several challenges:

- **Selecting Appropriate Predictors:** Different types of image regions may benefit more from specific prediction formulas, requiring careful design of multiple predictor methods.
- **Balancing Efficiency and Accuracy:** More complex prediction equations can improve compression but may increase computation.
- **Handling Edge Cases:** Pixels near boundaries require careful handling to avoid invalid references.
- **Ensuring Flexible APIs:** Providing multiple predictor methods demands a straightforward interface for easy extension and testing.

These challenges are addressed through rigorous experimentation to find the best predictors, efficient data structures for boundary handling, and by exposing a clean interface that combines multiple predictor functions in one class.

5.2.3 Test Cases for Predictor Class

Test Case 1: Basic Predictor Validation

- **Objective:** Verify that each predictor function correctly computes the expected predicted value.
- **Procedure:**
  1. Define inputs (x, y, z) = (10, 20, 15).
  2. Call Predictor_One, Predictor_Two, ..., LS_Predictor on these inputs, storing results.
  3. Compare each function's result to the expected outcome.
- **Expected Result:** Each predictor function yields the correct prediction value (e.g., Predictor_One returns x, others match their formulas).

Test Case 2: Edge Case Behavior

- **Objective:** Confirm that boundary conditions (e.g., minimal pixel values) do not cause errors.
- Procedure:
  1. Define inputs (x, y, z) = (0, 0, 0).
  2. Repeat same calls for all predictor functions.
  3. Verify no exceptions or negative indices occur.
- **Expected Results:** No runtime errors; each function still returns valid predictions for boundary values.

Test Case 3: LS Predictor with Swapped Inputs

- Objective: Evaluate LS_Predictor's clamping logic when z is outside the range of x and y.
- Procedure:
  1. Define inputs such that $z > x$ and $z > y$ (e.g., x = 10, y = 12, z = 30).
  2. Call LS_Predictor with these values.
  3. Check if the returned value equals the minimum (x or y) as per logic.
- **Expected Result:** LS_Predictor clamps prediction to the correct boundary value.

5.3 Task T2

The objective of Task 2 is to adapt the lossless image codec from Task 1 to support intra-frame video coding. This involves encoding videos as sequences of images, where each frame is encoded independently using spatial prediction within each frame. The encoded video file should include all necessary parameters, such as image dimensions and Golomb parameter m, to ensure proper decoding.

5.3.1 Specifications

The IntraEncoder class must implement the following core methods:

- encode: Encodes a single frame using a specified predictor function.
- encodeVideo: Encodes a video sequence, handling multiple frames and storing necessary parameters.
- getGolomb: Returns the EncoderGolomb instance used for encoding.

The IntraDecoder class must implement the following core methods:

- decode: Decodes a single frame using a specified predictor function.
- decodeVideo: Decodes a video sequence, reconstructing multiple frames from encoded data.

5.3.2 Challenges and Solutions

Implementing the IntraEncoder and IntraDecoder classes presents several challenges:

- **Efficient Prediction:** Accurately predicting pixel values to minimize residuals and improve compression efficiency.

  - **Solution:** Implement multiple predictor functions and select the best one based on the image characteristics.

- **Handling Different Color Formats:** Supporting various color formats (e.g., YUV420, YUV422, YUV444) for video encoding and decoding.

  - **Solution:** Convert RGB frames to the required YUV format before encoding and handle each format appropriately during processing.

- **Managing Large Data:** Processing large video files efficiently without excessive memory usage or slow performance.

  - **Solution:** Use optimized data structures and algorithms to handle frame-by-frame processing and minimize memory overhead.

- **Error Handling:** Ensuring robust error handling to manage edge cases, such as invalid input values or corrupted data.

  - **Solution:** Implement comprehensive error checking and validation throughout the encoding and decoding processes.

- **Bitstream Management:** Efficiently managing bit-level operations for encoding and decoding residuals using Golomb coding.

  - **Solution:** Leverage the BitStream and Golomb classes to handle bit-level operations and ensure accurate encoding and decoding.

- **Frame Boundary Handling:** Correctly handling frame boundaries to avoid referencing invalid pixels during prediction.

  - **Solution:** Pad frames with zeros and carefully manage boundary conditions during prediction and residual calculation.

  These challenges are addressed through meticulous design, comprehensive testing, and optimization of the encoding and decoding algorithms, ensuring efficient and accurate intra-frame video coding.

5.3.3 Test Cases for Intra-Frame Classes

Test Case 1: Basic Frame Encoding and Decoding

- **Objective:** Verify that the `IntraEncoder` and `IntraDecoder` classes can correctly encode and decode a single frame using a specified predictor function.

- **Procedure:**

  1. Initialize the `BitStream` for writing with an output file.
  2. Instantiate the `EncoderGolomb` with the output file path and `SIGN MAGNITUDE` encoding mode.
  3. Instantiate the `IntraEncoder` with the `EncoderGolomb` instance and a shift value.
  4. Load a sample frame (e.g., a small image).
  5. Encode the frame using the `IntraEncoder` and a predictor function.
  6. Finalize encoding by flushing the `BitStream` buffer.
  7. Initialize the `DecoderGolomb` with the encoded file path and the same encoding mode.
  8. Instantiate the `IntraDecoder` with the `DecoderGolomb` instance and the same shift value.
  9. Decode the frame using the `IntraDecoder` and the same predictor function.
  10. Compare the decoded frame with the original input frame.

- **Expected Result:** The decoded frame should exactly match the original input frame, confirming accurate encoding and decoding.

Test Case 2: Multi-Frame Video Encoding and Decoding

- **Objective:** Ensure that the `IntraEncoder` and `IntraDecoder` classes can correctly handle encoding and decoding of a video sequence with multiple frames.
- **Procedure:**
  1. Initialize the `BitStream` for writing with an output file.
  2. Instantiate the `EncoderGolomb` with the output file path and `SIGN MAGNITUDE` encoding mode.
  3. Instantiate the `IntraEncoder` with the `EncoderGolomb` instance and a shift value.
  4. Load a sequence of frames (e.g., a short video).
  5. Encode each frame using the `IntraEncoder` and a predictor function.
  6. Finalize encoding by flushing the `BitStream` buffer.
  7. Initialize the `DecoderGolomb` with the encoded file path and the same encoding mode.
  8. Instantiate the `IntraDecoder` with the `DecoderGolomb` instance and the same shift value.
  9. Decode each frame using the `IntraDecoder` and the same predictor function.
  10. Compare each decoded frame with the corresponding original input frame.
- **Expected Result:** Each decoded frame should exactly match the corresponding original input frame, confirming accurate encoding and decoding for the entire video sequence.

Test Case 3: Handling Different Color Formats

- **Objective:** Verify that the `IntraEncoder` and `IntraDecoder` classes can correctly handle different color formats (e.g., `YUV420`, `YUV422`, `YUV444`).
- **Procedure:**
  1. Initialize the `BitStream` for writing with an output file.
  2. Instantiate the `EncoderGolomb` with the output file path and `SIGN MAGNITUDE` encoding mode.
  3. Instantiate the `IntraEncoder` with the `EncoderGolomb` instance and a shift value.
  4. Load a sample frame in different color formats (e.g., `YUV420`, `YUV422`, `YUV444`).
  5. Encode each frame using the `IntraEncoder` and a predictor function.
  6. Finalize encoding by flushing the `BitStream` buffer.
  7. Initialize the `DecoderGolomb` with the encoded file path and the same encoding mode.
  8. Instantiate the `IntraDecoder` with the `DecoderGolomb` instance and the same shift value.
  9. Decode each frame using the `IntraDecoder` and the same predictor function.
  10. Compare each decoded frame with the corresponding original input frame.
- **Expected Result:** Each decoded frame should exactly match the corresponding original input frame, confirming accurate encoding and decoding for different color formats.

5.4 Task T3: Inter-Frame Video Coding

The objective of Task 3 is to expand the intra-frame video codec from Task 2 to support inter-frame (temporal) prediction with motion compensation. This involves using intra-frames (I-frames) at regular intervals and inter-frames (P-frames) that encode only changes relative to the previous frame. The codec should allow the user to specify the interval for I-frames, the block size, and the search range for motion estimation. Additionally, the codec should decide whether to use intra or inter mode for each block in P-frames based on the resulting bitrate, encoding blocks in intra mode if they result in a lower bitrate.

5.4.1 Specifications

The InterEncoder class must implement the following core methods:

- encode: Encodes a single frame using a specified predictor function, handling both I-frames and P-frames.
- encodeVideo: Encodes a video sequence, managing multiple frames and storing necessary parameters.
- motionEstimation: Estimates motion vectors between the current frame and a reference frame.
- motionCompensation: Reconstructs the predicted frame using motion vectors and the reference frame.
- shouldUseIntraMode: Decides whether to use intra-frame coding for a block based on the resulting bitrate.

The InterDecoder class must implement the following core methods:

- decode: Decodes a single frame using a specified predictor function, handling both I-frames and P-frames.
- decodeVideo: Decodes a video sequence, reconstructing multiple frames from encoded data.
- motionCompensation: Reconstructs the predicted frame using motion vectors and the reference frame.

5.4.2 Challenges and Solutions

Implementing the `InterEncoder` and `InterDecoder` classes presents several challenges:

- **Motion Estimation Accuracy:** Accurately estimating motion vectors to minimize prediction error and improve compression efficiency.
  - **Solution:** Implement a robust block-matching algorithm with a configurable search range to find the best matching blocks.
- **Motion Compensation:** Efficiently reconstructing frames using motion vectors to ensure high-quality video playback.
  - **Solution:** Develop an optimized motion compensation method that applies motion vectors to reference frames accurately.
- **Balancing Intra and Inter Modes:** Deciding when to use intra-frame coding versus inter-frame coding based on bitrate and quality.

  - **Solution:** Implement a decision mechanism that evaluates the cost of intra and inter modes for each block and selects the optimal mode.

- **Handling Large Video Files:** Processing large video files efficiently without excessive memory usage or slow performance.
  - **Solution:** Use optimized data structures and algorithms to handle frame-by-frame processing and minimize memory overhead.

- **Bitstream Management:** Efficiently managing bit-level operations for encoding and decoding motion vectors and residuals.
  - **Solution:** Leverage the `BitStream` and `Golomb` classes to handle bit-level operations and ensure accurate encoding and decoding.

- **Error Handling:** Ensuring robust error handling to manage edge cases, such as invalid input values or corrupted data.
  - **Solution:** Implement comprehensive error checking and validation throughout the encoding and decoding processes.

- **Frame Boundary Handling:** Correctly handling frame boundaries to avoid referencing invalid pixels during motion estimation and compensation.
  - **Solution:** Pad frames with zeros and carefully manage boundary conditions during prediction and residual calculation.

These challenges are addressed through meticulous design, comprehensive testing, and optimization of the encoding and decoding algorithms, ensuring efficient and accurate inter-frame video coding.

5.4.3 Test Cases for Inter-Frame Classes

Test Case 1: Basic Frame Encoding and Decoding

- **Objective:** Verify that the `InterEncoder` and `InterDecoder` classes can correctly encode and decode a single frame using a specified predictor function.

- **Procedure:**
  1. Initialize the `BitStream` for writing with an output file.
  2. Instantiate the `EncoderGolomb` with the output file path and `SIGN MAGNITUDE` encoding mode.
  3. Instantiate the `InterEncoder` with the `EncoderGolomb` instance, block size, search range, and I-frame interval.
  4. Load a sample frame (e.g., a small image).
  5. Encode the frame using the `InterEncoder` and a predictor function.
  6. Finalize encoding by flushing the `BitStream` buffer.
  7. Initialize the `DecoderGolomb` with the encoded file path and the same encoding mode.
  8. Instantiate the `InterDecoder` with the `DecoderGolomb` instance, block size, and I-frame interval.
  9. Decode the frame using the `InterDecoder` and the same predictor function.
  10. Compare the decoded frame with the original input frame.

- **Expected Result:** The decoded frame should exactly match the original input frame, confirming accurate encoding and decoding.

Test Case 2: Multi-Frame Video Encoding and Decoding

- **Objective:** Ensure that the `InterEncoder` and `InterDecoder` classes can correctly handle encoding and decoding of a video sequence with multiple frames, including both I-frames and P-frames.
- **Procedure:**
  1. Initialize the `BitStream` for writing with an output file.
  2. Instantiate the `EncoderGolomb` with the output file path and `SIGN MAGNITUDE` encoding mode.
  3. Instantiate the `InterEncoder` with the `EncoderGolomb` instance, block size, search range, and I-frame interval.
  4. Load a sequence of frames (e.g., a short video).
  5. Encode each frame using the `InterEncoder` and a predictor function.
  6. Finalize encoding by flushing the `BitStream` buffer.
  7. Initialize the `DecoderGolomb` with the encoded file path and the same encoding mode.
  8. Instantiate the `InterDecoder` with the `DecoderGolomb` instance, block size, and I-frame interval.
  9. Decode each frame using the `InterDecoder` and the same predictor function.
  10. Compare each decoded frame with the corresponding original input frame.
- **Expected Result:** Each decoded frame should exactly match the corresponding original input frame, confirming accurate encoding and decoding for the entire video sequence.

Test Case 3: Motion Estimation and Compensation

- **Objective:** Verify that the motion estimation and compensation methods in the `InterEncoder` and `InterDecoder` classes correctly handle motion vectors and reconstruct frames.
- **Procedure:**
  1. Initialize the `BitStream` for writing with an output file.
  2. Instantiate the `EncoderGolomb` with the output file path and `SIGN MAGNITUDE` encoding mode.
  3. Instantiate the `InterEncoder` with the `EncoderGolomb` instance, block size, search range, and I-frame interval.
  4. Load a pair of consecutive frames (e.g., from a video).
  5. Perform motion estimation using the `InterEncoder` and store the motion vectors.
  6. Perform motion compensation using the `InterEncoder` and the stored motion vectors to predict the second frame.
  7. Encode the residual between the predicted frame and the actual second frame.
  8. Finalize encoding by flushing the `BitStream` buffer.
  9. Initialize the `DecoderGolomb` with the encoded file path and the same encoding mode.
  10. Instantiate the `InterDecoder` with the `DecoderGolomb` instance, block size, and I-frame interval.
  11. Decode the motion vectors and the residual using the `InterDecoder`.
  12. Perform motion compensation using the `InterDecoder` and the decoded motion vectors to reconstruct the second frame.

13. Compare the reconstructed frame with the actual second frame.

- **Expected Result:** The reconstructed frame should closely match the actual second frame, confirming accurate motion estimation and compensation.

5.5 Task T4

The objective of Task 4 is to extend the video codec to support lossy video coding by introducing quantization on the prediction residuals before applying Golomb coding. This involves allowing the user to specify the quantization level, which controls the trade-off between compression and quality. By quantizing the prediction residuals, the codec achieves better compression at the cost of some data loss, resulting in smaller file sizes while maintaining acceptable video quality.

5.5.1 Specifications

The `LossyVideoCodec` class implements the following core methods:

- `calculateResiduals`: Calculates the residuals between the current frame and the predicted frame.
- `quantizeResiduals`: Quantizes the residuals to reduce precision and achieve compression.
- `inverseQuantizeResiduals`: Reconstructs the residuals from the quantized values.
- `selectPredictor`: Selects the appropriate predictor function based on the frame number or other criteria.

5.5.2 Challenges and Solutions

Implementing the `LossyVideoCodec` class presents several challenges:

- **Quantization Level Selection:** Determining the appropriate quantization level to balance compression and video quality.
  - **Solution:** Allow the user to specify the quantization level and provide guidelines on selecting appropriate values based on desired quality.
- **Residual Calculation:** Accurately calculating the residuals between the predicted and actual frames to ensure efficient compression.
  - **Solution:** Implement robust methods for calculating residuals and ensure they are correctly computed for each frame.
- **Efficient Quantization:** Applying quantization to residuals without introducing significant artifacts or loss of important details.
  - **Solution:** Use optimized quantization algorithms that minimize the impact on video quality while achieving desired compression.
- **Bitstream Management:** Efficiently managing bit-level operations for encoding and decoding quantized residuals.
  - **Solution:** Leverage the `BitStream` and `Golomb` classes to handle bit-level operations and ensure accurate encoding and decoding.

- **Error Handling:** Ensuring robust error handling to manage edge cases, such as invalid input values or corrupted data.
  - **Solution:** Implement comprehensive error checking and validation throughout the encoding and decoding processes.
- **Frame Prediction:** Accurately predicting frames using previous frames to minimize residuals and improve compression efficiency.
  - **Solution:** Implement multiple predictor functions and select the best one based on the frame characteristics.
- **Handling Large Video Files:** Processing large video files efficiently without excessive memory usage or slow performance.
  - **Solution:** Use optimized data structures and algorithms to handle frame-by-frame processing and minimize memory overhead.

These challenges are addressed through meticulous design, comprehensive testing, and optimization of the encoding and decoding algorithms, ensuring efficient and accurate lossy video coding.

5.5.3 Test Cases for Inter-Frame Classes

To ensure the reliability and efficiency of the `LossyVideoCodec` class, the following test cases were developed and executed:

Test Case 1: Basic Frame Encoding and Decoding

- **Objective:** Verify that the `LossyVideoCodec` class can correctly encode and decode a single frame using a specified predictor function and quantization level.
- **Procedure:**
  1. Initialize the `BitStream` for writing with an output file.
  2. Instantiate the `EncoderGolomb` with the output file path and `INTERLEAVING` encoding mode.
  3. Load a sample frame (e.g., a small image).
  4. Encode the frame using the `LossyVideoCodec` with a specified predictor function (e.g., `Predictor_One`) and quantization level.
  5. Finalize encoding by flushing the `BitStream` buffer.
  6. Initialize the `DecoderGolomb` with the encoded file path and the same encoding mode.
  7. Decode the frame using the `LossyVideoCodec` with the same predictor function and quantization level.
  8. Compare the decoded frame with the original input frame.
- **Expected Result:** The decoded frame should closely match the original input frame, confirming accurate encoding and decoding with acceptable quality loss.

Test Case 2: Multi-Frame Video Encoding and Decoding

- **Objective:** Ensure that the `LossyVideoCodec` class can correctly handle encoding and decoding of a video sequence with multiple frames, including both I-frames and P-frames.
- **Procedure:**
  1. Initialize the `BitStream` for writing with an output file.
  2. Instantiate the `EncoderGolomb` with the output file path and `INTERLEAVING` encoding mode.
  3. Load a sequence of frames (e.g., a short video).
  4. Encode each frame using the `LossyVideoCodec` with a specified predictor function (e.g., `Predictor_One`) and quantization level.
  5. Finalize encoding by flushing the `BitStream` buffer.
  6. Initialize the `DecoderGolomb` with the encoded file path and the same encoding mode.
  7. Decode each frame using the `LossyVideoCodec` with the same predictor function and quantization level.
  8. Compare each decoded frame with the corresponding original input frame.
- **Expected Result:** Each decoded frame should closely match the corresponding original input frame, confirming accurate encoding and decoding for the entire video sequence with acceptable quality loss.

Test Case 3: Quantization Level Impact

- **Objective:** Evaluate the impact of different quantization levels on the compression efficiency and video quality.
- **Procedure:**
  1. Initialize the `BitStream` for writing with an output file.
  2. Instantiate the `EncoderGolomb` with the output file path and `INTERLEAVING` encoding mode.
  3. Load a sample frame (e.g., a small image).
  4. Encode the frame using the `LossyVideoCodec` with a specified predictor function (e.g., `Predictor_One`) and varying quantization levels (e.g., 1, 5, 10).
  5. Finalize encoding by flushing the `BitStream` buffer.
  6. Initialize the `DecoderGolomb` with the encoded file path and the same encoding mode.
  7. Decode the frame using the `LossyVideoCodec` with the same predictor function and quantization levels.
  8. Compare the decoded frames with the original input frame and evaluate the compression ratio and quality loss.
- **Expected Result:** Higher quantization levels should result in better compression but more noticeable quality loss, while lower quantization levels should maintain higher quality with less compression.

The results for Part IV were obtained using the following image:

And for the video data we used a .y4m named bus_cif.y4m downloaded from **Website**

5.6 Results & Analysis

Table 5.1: Decoding Results

Technology	Processing Time (ms)	Error Metrics(Lossy)	Compression Ratio	Size of File (Bytes)
BitStream	123 ms	N/A	7.99	30969
Golomb - Positive Numbers	15 ms	N/A	1.67	12
Golomb - Negative Numbers	18 ms	N/A	1.67	12
Golomb - Mixed Numbers	20 ms	N/A	1.67	12
Audio Lossless	33ms (Encode) + 40ms (Decode)	N/A	1.44	3997796
Audio Lossy (8-bits)	18ms (Encode) + 26ms (Decode)	5418.4	2.33	5176796
Video IntraEncoder	2689 ms	N/A	1.34	22810538
Video IntraDecoder	2278 ms	N/A	1.00	22810538
Video InterEncoder	35390 ms	N/A	0.68	22810538
Video InterDecoder	34217 ms	N/A	1.00	22810538
LossyVideoCodec	1241 ms	6273.9	1.68	22810538

Predictor	Processing Time (ms)	Size of File (Bytes)
Predictor_1	38.7703 ms	786447
Predictor_2	38.0391 ms	786447
Predictor_3	38.9147 ms	786447
Predictor_4	38.9285 ms	786447
Predictor_5	38.9021 ms	786447
Predictor_6	36.8539 ms	786447
Predictor_7	37.5877 ms	786447
LS_Predictor	44.4008 ms	786447

Table 5.2: Performance of Predictors

5.7 Comparative Analysis

Comparing with existing industry-standard codecs (e.g., JPEG for images, MP3 for audio, H.264 for video):

- **Jpeg:** size 786447, ratio 8:1, 5ms
- **MP3:** size 5,176,796 Bytes, ratio 11:1, 10–50 ms
- **H264:** size 22,810,538 Bytes, 30:1, 50–200 ms

Industry-Standard Codecs:

- JPEG: 8:1 compression ratio
- MP3: 11:1 compression ratio
- H.264: 30:1 compression ratio

Our Codec Implementation (Table 5.1):

- Best compression: BitStream (7.99), Golomb variants (1.67)
- Lossy Audio: 1.43
- Lossy Video: 1.68

Observations:

- H.264 outperforms all codecs in Table 5.1 in terms of compression efficiency.
- BitStream from our implementation achieves the highest compression ratio among our codecs.
- Golomb variants offer decent compression but are significantly lower than H.264 and BitStream.
- Lossy Audio and Video codecs in our implementation have lower compression ratios compared to MP3 and H.264.

5.7.1 Quality

Industry-Standard Codecs:

- Lossy compression, but generally considered to provide high-quality output for their respective media types (images, audio, video).

Our Codec Implementation:

- Lossy codecs have error metrics (indicating quality loss), but the exact impact on perceived quality is unclear without subjective evaluation.

5.7.2 Processing Time

Industry-Standard Codecs:

- JPEG: 5ms
- MP3: 50ms
- H.264: 200ms

Our Codec Implementation:

- BitStream: 123ms
- Golomb variants: 15–20ms
- Lossy Audio: 83406ms
- Lossy Video: 1241ms

Observations:

- JPEG is the fastest industry-standard codec.
- H.264 has the longest encoding time among industry-standard codecs.
- Golomb variants are the fastest in our implementation.
- Lossy Audio and Video have significantly longer processing times compared to industry-standard codecs.

5.7.3 Possible Reasons for Differences

Algorithm Complexity:

- H.264 employs sophisticated algorithms with high computational complexity, leading to longer encoding times.
- Our Lossy Audio and Video codecs might involve complex algorithms, resulting in longer processing times.

Compression Trade-offs:

- Higher compression often comes at the cost of increased processing time.

Implementation Efficiency:

- The efficiency of our codec implementations can significantly impact processing time. Optimizations like parallelization and vectorization can improve performance.

Data Characteristics:

- The specific characteristics of the data being compressed can influence the performance of different codecs.

5.7.4 Recommendations

- **Optimization:** Investigate ways to optimize our codec implementations for faster processing times while maintaining reasonable compression ratios.
- **Algorithm Comparison:** Compare our algorithms with those used in industry-standard codecs to identify potential improvements.
- **Quality Assessment:** Conduct subjective and objective quality assessments to evaluate the perceptual quality of our lossy codecs.

5.7.5 Note

The analysis is based on the limited data provided. A more comprehensive comparison would require additional information such as:

- Specific data types and sizes used for testing.
- Subjective quality assessments of the decoded output.
- Hardware and software specifications used for testing.

Chapter 6

Conclusion

Based on the content provided, here is a potential conclusion for your project:

The project explored the implementation and evaluation of multiple data compression techniques across various domains, including text, audio, image, and video. By leveraging foundational tools such as the BitStream class for bit-level operations and Golomb coding for efficient data representation, the study achieved promising results in both lossless and lossy compression scenarios. Comprehensive testing across different modules demonstrated the effectiveness of the implemented codecs, highlighting their reliability and performance in real-world datasets.

Key achievements include the development of lossless and lossy audio codecs using predictive coding and Golomb encoding, enabling precise control over compression ratios and maintaining high fidelity in reconstructed data. For image and video compression, advanced techniques like spatial and temporal prediction were successfully integrated, supporting intra- and inter-frame video encoding. The introduction of quantization for lossy video coding allowed a balance between compression efficiency and quality, further extending the codec's applicability.

While the project outcomes show significant progress in optimizing compression techniques, further enhancements are necessary to bridge the gap with industry-standard codecs in terms of speed and compression ratios. Future work could focus on algorithm optimization, parallel processing, and advanced predictive models to improve the efficiency and scalability of the codecs. Overall, the findings contribute to the ongoing pursuit of efficient data storage and transmission in modern digital systems.

Chapter 7

Appendix

Link to our code repository: Github