Departamento de Eletrónica, Telecomunicações e Informática
da Universidade de Aveiro

Information and Coding
Lab work nº. 2

Diogo Silva (108212)
Martim Carvalho (108749)
Miguel Marques (103162)

November 17, 2025

Contents

Chapter 1

Introduction

This project aims to provide a practical understanding of how digital media can be processed and compressed without loss of information. It is organized into four main parts, each introducing different concepts and techniques.

The **first part** focuses on basic image manipulation, where operations such as channel extraction, mirroring, rotation, and intensity adjustment are implemented directly at the pixel level. This helps reinforce how images are structured and how simple transformations can be performed manually.

The **second part** introduces Golomb coding. To support it, a `BitStream` class is created to handle bit-level input and output, followed by the implementation of a complete Golomb encoder and decoder. This section provides hands-on experience with entropy coding and bit-wise operations.

In the **third part**, these ideas are applied to audio. A lossless audio codec is developed using temporal prediction, stereo processing methods, and adaptive Golomb coding. The goal is to compress audio samples while ensuring perfect reconstruction.

The **final part** extends predictive coding to images. Several spatial predictors, including Paeth and Lossless Joint Photographic Experts Group (JPEG-LS) Median Edge Detector (MED), are implemented to generate residuals that are then compressed using Golomb coding. A test suite is used to compare predictor performance and evaluate compression efficiency.

Overall, the project covers the main building blocks of lossless compression-prediction, residual calculation, and entropy coding-showing how these concepts can be applied to both audio and images.

Chapter 2

Exercises

2.1 Part-I

2.1.1 Exercise 1

Digital color images in the RGB color model contain three separate channels (Red, Green and Blue) that combine to form the final image. This exercise develops a practical understanding of image data structures by implementing a program that extracts individual color channels at the pixel level using Open Source Computer Vision Library (OpenCV). The objective is to create a command-line application that reads a color image, extracts a user-specified channel, and saves it as a grayscale image, with the requirement to process pixels individually rather than using higher-level functions.

Code

The program follows a straightforward workflow:

1. It first validates that the channel number is within the valid range (0–2 for RGB channels).
2. It then loads the input color image using OpenCV's image reading functionality.
3. After successful loading, a single-channel output matrix is created with the same dimensions as the input image, using 8-bit unsigned integer format for grayscale values.

Channel Extraction Process

The core extraction process uses two nested loops to iterate through every pixel in the image. For each pixel:

- The program accesses the RGB vector and extracts only the value from the requested channel.
- The extracted value is stored in the corresponding position of the output image.

OpenCV stores colors in RGB order, so index 0 represents Blue, index 1 Green, and index 2 Red.

Diagnostic Output

The program includes diagnostic output showing the extracted channel number, image dimensions, and sample pixel values for verification purposes. This helps ensure the extraction is working correctly during development and testing.

Output Handling

Output handling adapts to the file format:

- For Portable Pixmap (PPM) extensions (color format), the single-channel result is converted to a three-channel grayscale image where all RGB components have identical values.
- For Portable Gray-Map (PGM) or other formats, the single-channel image is saved directly without conversion.

Main Function Behavior

The main function handles command-line arguments, validating that three parameters are provided (input file, output file, and channel number). If arguments are incorrect, it displays usage information. The function then calls the extraction routine and returns its exit code to indicate success or failure.

Usage

The program is executed from the command line as follows:

```
make run-exe1 INPUT=input_file OUTPUT=output_file CHANNEL=X
```

Index	Channel
0	Blue
1	Green
2	Red

Table 2.1: Channel Indexing Scheme

Results

The result in Figure 2.8 the original image alongside three additional images, each displaying only a single RGB channel—red, green, or blue. When an image is separated into its individual RGB channels, each resulting image highlights the intensity levels for that channel alone. Pixels with high intensity in a specific channel appear as lighter shades (closer to white), while pixels with low intensity appear darker (closer to black). Since only one channel is retained in each output, these images typically appear as grayscale representations, where the brightness at each pixel reflects the channel's value and not the full color.

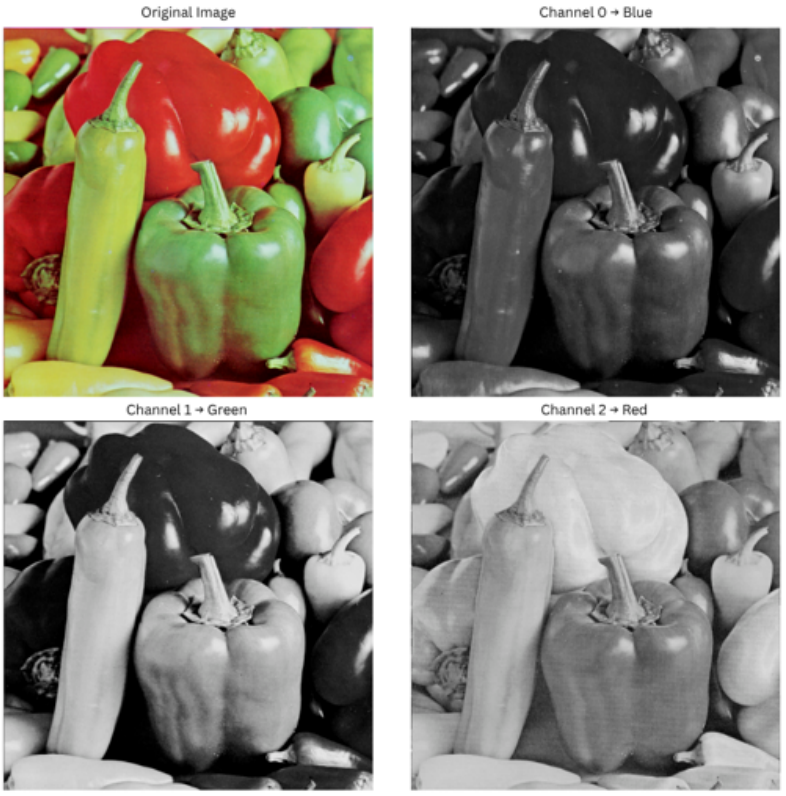

Figure 2.1: Comparasion between original image and single output channel

2.1.2 Exercise 2

This exercise focuses on developing image manipulation functions without relying on built-in OpenCV methods. The program handles standard PPM image files and includes operations for negative, mirroring, rotation, and intensity adjustment. The results below summarize the procedures used in each operation, the effects obtained, and performance measures from standard image tests.

Code

This section details the main image processing operations required by exercise 2. Each algorithm was implemented to work directly on the pixel data, without using high-level OpenCV functions, providing a deeper understanding of the underlying transformations. The following items explain the logic and mathematical calculations for each effect, illustrating how the software manipulates images to achieve the desired results.

1. **Negative**

   The negative operation inverts the color or intensity of each pixel in the image. For each pixel value p, the new value is calculated as

   $$p_{\text{new}} = \text{maxval} - p$$

   where maxval is the maximum possible value for a pixel (usually 255 for 8-bit images). This transformation swaps light and dark areas. The calculation is performed for every pixel, iterating through the image data array and applying the formula directly.

2. **Mirroring**

   Mirroring can be performed horizontally or vertically:

   - **Horizontal Mirroring**: Each pixel in a row is swapped with its counterpart on the opposite side of the vertical center line. For a pixel at position (x, y), the mirrored position is calculated as

     $$p_{\text{new}}(x, y) = p(\text{width} - 1 - x, y)$$

     This is implemented by iterating over half the width of each row and swapping pixel values.

   - **Vertical Mirroring**: Each pixel in a column is swapped with its counterpart on the opposite side of the horizontal center line. For a pixel at (x, y), the mirrored position is calculated as

     $$p_{\text{new}}(x, y) = p(x, \text{height} - 1 - y)$$

     This is implemented by iterating over half the height of each column and swapping pixel values.

3. **Rotation (Multiples of 90°)**

   Rotation is performed by remapping pixel positions according to the desired number of 90° turns. For a 90° rotation, the mapping is

   $$p_{\text{new}}(x, y) = p(y, \text{width} - 1 - x)$$

   The image dimensions are swapped for 90° and 270°, while for 180° the width and height remain the same but pixel positions are reversed. The implementation uses modular arithmetic to handle multiple rotations and ensures that no interpolation is needed, preserving image quality.

4. **Intensity Adjustment**

   To adjust brightness, an integer value v is added to each pixel's intensity. The result is clipped to stay within the valid range $[0, \text{maxval}]$

   $$p_{\text{new}} = \min(\max(p + v, 0), \text{maxval})$$

   This operation is performed for every pixel, ensuring that no value exceeds the allowed limits. Increasing v makes the image brighter, while decreasing v makes it darker.

Usage

The program is executed from the command line as follows:

```
make run-effects <input_file> <output_file> <operation>     [<
    value>]
```

- `input_file`: Path to the original image file.
- `output_file`: Name of the output file to be created.
- `operation`: Operation to be performed (see list below).
- `value`: Extra parameter required for some operations (e.g., rotation, intensity).

Supported Operations and Examples

Operation	Description	Example Usage
negative	Creates the image negative	`make run-effects effects INPUT=inputFile OUTPUT=outputFile EFFECT=negative`
mirror_h	Horizontal mirroring	`make run-effects effects INPUT=inputFile OUTPUT=outputFile EFFECT=mirror_h`
mirror_v	Vertical mirroring	`make run-effects effects INPUT=inputFile OUTPUT=outputFile EFFECT=mirror_v`
rotate	Rotate by multiples of 90°	`make run-effects effects INPUT=inputFile OUTPUT=outputFile EFFECT=rotate ANGLE=2`
intensity	Adjust brightness (+ or - value)	`make run-effects effects INPUT=inputFile OUTPUT=outputFile EFFECT=intensity FACTOR=40`

Table 2.2: Supported operations and example usage for the image processing program.

Results

The visual results of each image processing operation are presented below. For each effect applied, the original and processed images are shown side by side, allowing for a direct and intuitive comparison. This approach highlights the impact of each algorithm and enables a clear evaluation of the effectiveness and accuracy of the implemented methods.

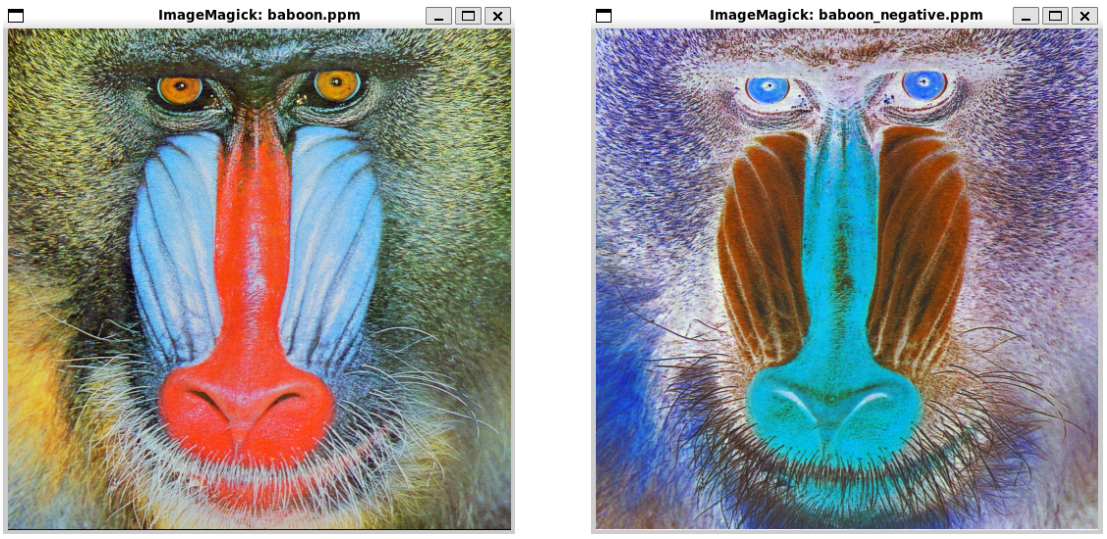

Figure 2.2: Original image vs edited image after applying the negative effect

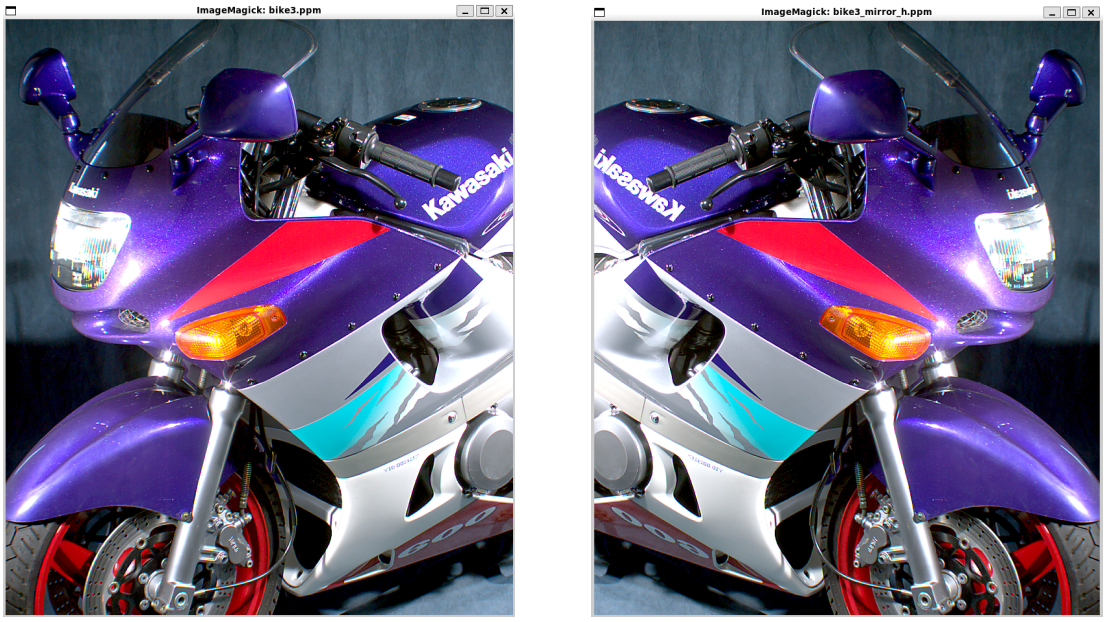

Figure 2.3: Original image vs edited image after applying the horizontal mirror effect

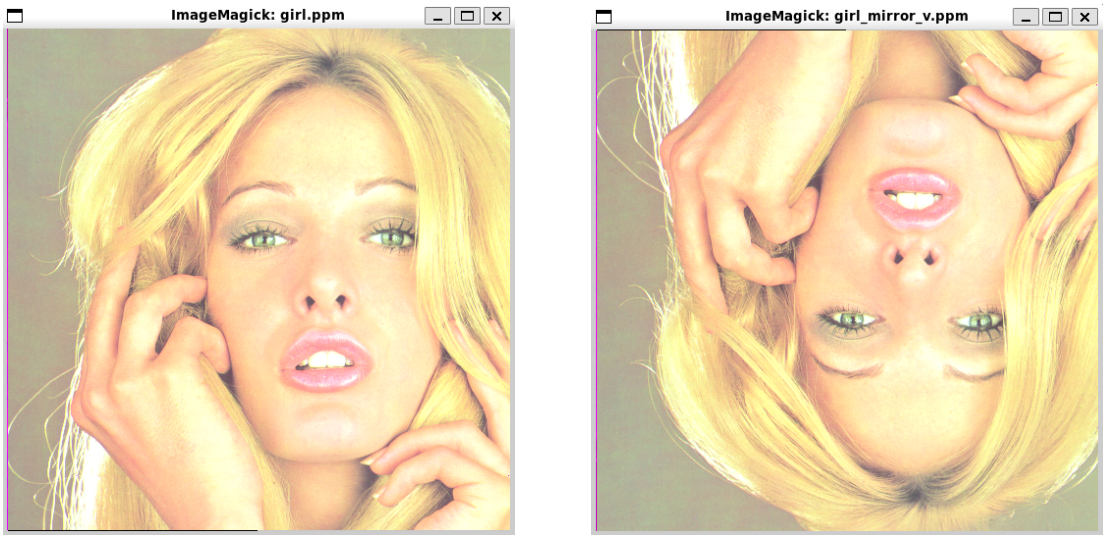

Figure 2.4: Original image vs edited image after applying the vertical mirror effect

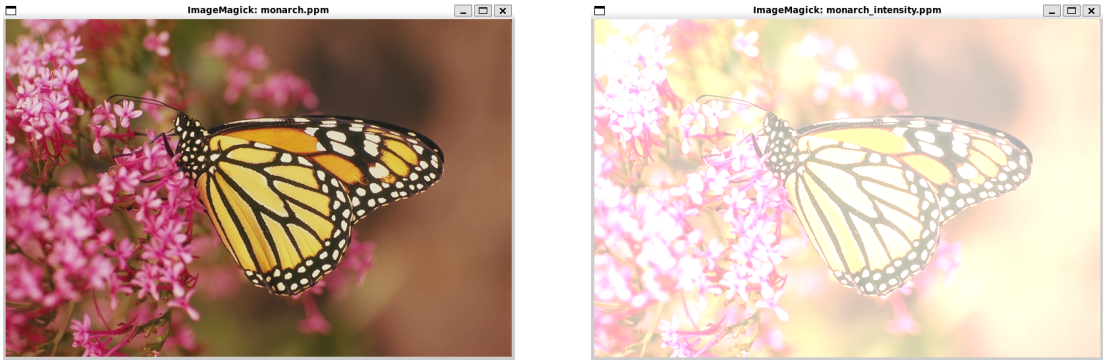

Figure 2.5: Original image vs edited image after applying the intensity adjustment effect (150)

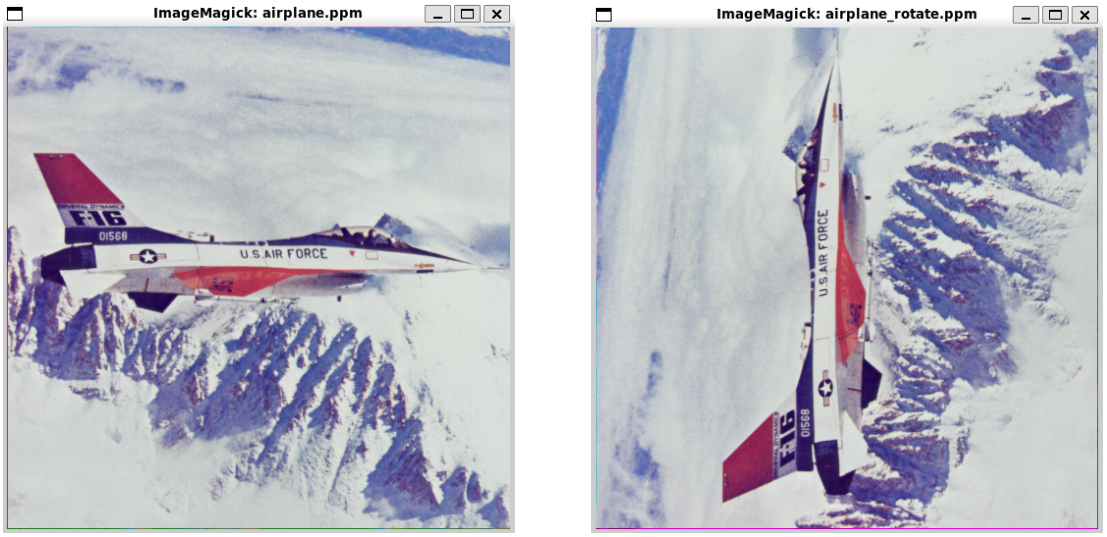

Figure 2.6: Original image vs edited image after applying the rotation effect (-90º)

2.2 Part-II

2.2.1 Exercise 3

Code

The implementation of Golomb coding required the development of two interconnected C++ classes: BitStream and Golomb. These classes work together to provide a complete encoding and decoding solution for integer values using the Golomb algorithm.

BitStream Class

The BitStream class (`bitstream.hpp` and `bitstream.cpp`) provides bit-level file I/O operations essential for handling variable-length codes.The class implements a buffering mechanism with the following key components:

- **Private Members**: A file stream object, an 8-bit character buffer, and a bit position counter to track the current position within the buffer;
- **Constructor/Destructor**: Opens files in binary mode with specified access permissions and ensures proper buffer flushing upon destruction;
- **Core Methods**: `writeBit()` and `readBit()` for single-bit operations, `writeBits()` and `readBits()` for multi-bit operations (supporting up to 64 bits), and `writeString()` /`readString()` for null-terminated string handling.
- **Buffer Management**: `flushBuffer()` writes the current buffer to disk when full or upon file closure, while `fillBuffer()` loads the next byte from the file during read operations.

Golomb Class

The Golomb class (`golomb.hpp` and `golomb.cpp`) implements the complete encoding and decoding algorithm based on the mathematical principles of Golomb codes. The implementation includes:

- **Private members:** Parameter m (divisor), calculated values $b = \lfloor \log_2(m) \rfloor$ and $k = 2^{b+1} - m$, for truncated binary encoding, and an enumeration specifying the sign-handling approach.
- **Constructor:** Initializes the encoder with parameter m and validates that $m > 0$, automatically calculating b and k values.
- `encode()` **method:** Applies the selected sign-handling approach (Sign-and-Magnitude or Odd-Even Mapping), computes quotient $q = \left\lfloor \frac{n}{m} \right\rfloor$ and remainder $r = n \bmod m$, writes the unary-encoded quotient followed by the truncated binary-encoded remainder, and returns the total number of bits written.
- `decode()` **method:** Reads the unary quotient by counting consecutive 1-bits until a 0 is encountered, reads the truncated binary remainder (using b or $b + 1$ bits depending on whether $r < k$), reconstructs the original value as $n = q \cdot m + r$, and applies the inverse sign-handling transformation.
- **Sign-handling approaches:** Two methods as specified in the assignment. Sign-and-Magnitude uses an explicit sign bit before encoding the absolute value, while Odd-Even Mapping uses the bijection,

$$f(n) = 2n - 1 \quad \text{for } n > 0 \qquad \text{and} \qquad f(n) = -2n \quad \text{for } n \leq 0$$

  to interleave positive and negative integers without a sign bit.

The `updateParameters()` helper function recalculates bb and kk whenever mm changes, ensuring mathematical correctness.

Usage

The Golomb encoder/decoder is utilized by instantiating both the `BitStream` and `Golomb` classes with appropriate parameters. The typical workflow consists of three phases: initialization, encoding/decoding operations, and file closure.

Encoding workflow:

1. Create a `BitStream` object in write mode:

```
BitStream bs("output.bin", std::ios::out)
```

2. Instantiate a Golomb encoder with desired parameter m and sign-handling approach:

```
Golomb golomb(m, Golomb::HandleSignApproach::ODD_EVEN_MAPPING)
```

3. Encode integer values by calling:

`golomb.encode(value, bs)`,

   for each value to be compressed.

4. The `BitStream` destructor automatically flushes remaining bits and closes the file.

Decoding workflow:

1. Create a `BitStream` object in read mode:

   `BitStream bs("output.bin", std::ios::in)`

2. Instantiate a Golomb decoder with the same parameter m and sign-handling approach used during encoding.

3. Decode values by calling:

   `golomb.decode(bs)`,

   until the end of the file is reached (checked using `bs.eof()`).

4. The `BitStream` destructor handles file closure automatically.

Parameter selection:
The parameter m should be chosen based on the expected probability distribution of the input data, optimal compression occurs when m approximates the inverse of the geometric distribution parameter p, i.e., $m \approx \frac{1}{p}$.
The sign-handling approach should be selected based on the data characteristics: Sign-and-Magnitude is simpler and suitable when positive values dominate, while Odd-Even Mapping provides better compression when both positive and negative values have similar small magnitudes.

2.3 Part-III

2.3.1 Exercise 4

Code

AudioCodec Class Architecture

The AudioCodec class (`audio_codec.hpp` and `audio_codec.cpp`) provides a comprehensive framework for lossless audio compression. The class architecture includes the following fundamental elements.

Configuration enumerations: Two enumerated types define operational modes: `PredictorType` specifies the temporal prediction algorithm (`LINEAR_1`, `LINEAR_2`, `LINEAR_3`, or `ADAPTIVE`), while `StereoMode` determines the stereo encoding strategy (`INDEPENDENT`, `MID_SIDE`, or `ADAPTIVE`).

Private members: The class maintains state variables including the selected predictor type, stereo processing mode, Golomb parameter m, adaptive mode flag, and a `CompressionStats` structure that records metrics such as compression ratio, bits per sample, optimal m, and file size. Static constants define the block size for adaptive parameter estimation (1024 samples) and parameter bounds $m_{\min} = 4$, $m_{\max} = 256$.

File format specification: The encoder generates a custom binary format with a structured header containing a 32-bit magic number (`0x474F4C4D`, "GOLM"), followed by metadata fields for channel count, sample rate, bit depth, predictor type,

stereo mode, and samples per channel. This metadata enables lossless reconstruction of the audio stream by the decoder.

Temporal Prediction

Predictive coding exploits temporal correlation in audio signals by encoding the difference between the actual sample and a predicted value derived from previous samples. The implementation provides four predictor variants with increasing polynomial order:

- **LINEAR_1**: First-order prediction, where $\hat{x}[n]$ represents the predicted value of sample n:

  $$\hat{x}[n] = x[n-1]$$

  This zero-order-hold predictor assumes the signal remains constant between samples, performing optimally for signals with low temporal variation.

- **LINEAR_2**: Second-order prediction, effectively performing linear extrapolation based on the slope between the two previous samples:

  $$\hat{x}[n] = 2x[n-1] - x[n-2]$$

  This predictor models constant-velocity signal evolution and achieves superior compression for signals with smooth trajectories, such as musical tones.

- **LINEAR_3**: Third-order prediction, modeling constant-acceleration signal dynamics through quadratic extrapolation:

  $$\hat{x}[n] = 3x[n-1] - 3x[n-2] + x[n-3]$$

  Higher-order prediction reduces residual variance for highly correlated audio signals but may increase residual magnitude for signals with rapid transients.

- **ADAPTIVE**: Dynamically selects among the three fixed-order predictors based on local signal characteristics.

Stereo Encoding

Stereo audio signals often exhibit significant inter-channel correlation, particularly in music recordings where the left and right channels capture similar acoustic information with spatial variations. The codec implements two stereo encoding approaches:

- **MID_SIDE mode:** Applies a reversible linear transform to convert the left-right channel pair into mid and side channels. The mid channel $M[n]$ represents the average signal, while the side channel $S[n]$ captures the difference:

  $$M[n] = \frac{L[n] + R[n]}{2}, \qquad S[n] = L[n] - R[n]$$

  For highly correlated channels, the side channel exhibits significantly lower energy than the difference between uncorrelated channels, enabling more efficient Golomb coding. The encoder writes a single-bit flag (1) to indicate mid-side encoding.

- **ADAPTIVE mode:** Encodes left and right channels separately using temporal prediction on each channel independently. This mode incurs no computational overhead for decorrelation but fails to exploit inter-channel redundancy. The encoder writes a single-bit flag (0) to indicate independent encoding, followed by the encoded data for each channel.

Adaptive Parameter Selection

The efficiency of Golomb coding critically depends on matching the parameter mm to the statistical distribution of prediction residuals. The codec implements adaptive parameter estimation through the `estimateOptimalM()` method, which computes the mean absolute value of residuals within a block and selects mm approximately equal to this mean. The optimal mm value is written to the bitstream using 16 bits, allowing the decoder to configure its Golomb decoder identically.

Encoding and Decoding

- **Encoding:** The `encode()` method accepts raw audio samples as a vector of 32-bit integers, along with metadata (channel count, sample rate, bit depth) and an output filename. After writing the file header, the encoder branches based on channel count. For mono audio, samples are encoded directly using the selected predictor. For stereo audio, samples are unraveled into separate left and right channels, then processed according to the selected stereo mode.

  The `encodeChannel()` helper method computes prediction residuals for all samples, estimates the optimal mm parameter, writes it to the bitstream, and encodes each residual using a Golomb encoder configured with the `ODD_EVEN_MAPPING` approach for signed integer handling.

  Upon completion, the method calculates compression statistics by comparing the original uncompressed size with the compressed file size.

- **Decoding:** The `decode()` method reads the file header to extract metadata and verify the magic number. Based on the recorded channel count and stereo mode flag, it invokes `decodeChannel()` for each encoded channel. This helper method reads the mm parameter, instantiates a Golomb decoder, and iteratively decodes residuals.

  For each sample position, the decoder computes the predicted value using the same predictor algorithm as the encoder, adds the decoded residual, and stores the reconstructed sample.

  For stereo mid-side encoded files, the inverse mid-side transform is applied before interleaving the channels into the output vector.

Usage

The AudioCodec is instantiated with four parameters: predictor type, stereo mode, initial mm value (0 for automatic estimation), and adaptive flag (default true).

Encoding Workflow

1. Load audio samples from a Waveform Audio File Format (WAV) file or generate synthetic test signals using helper functions;

2. Instantiate an AudioCodec with desired configuration:

   ```
   AudioCodec codec(AudioCodec::PredictorType::LINEAR_2,
   AudioCodec::StereoMode::MID_SIDE, 0, true);
   ```

3. Call the encode method:

   ```
   codec.encode(audioData, channels, sampleRate,
   bitsPerSample, "output.golomb");
   ```

4. Retrieve compression statistics: `auto stats = codec.getLastStats()`

Decoding Workflow

1. Instantiate an AudioCodec with the same configuration used during encoding
2. Call the decode method:

   ```
   codec.decode("output.golomb", decodedAudio, channels, sampleRate,
   bitsPerSample)
   ```

3. Verify perfect reconstruction by comparing original and decoded samples
4. Write reconstructed samples to a WAV file for auditory verification

The test suite (`test_audio_codec.cpp`) provides comprehensive validation infrastructure, including WAV file I/O utilities, synthetic audio generation functions (sine waves, chords, frequency sweeps, noise, and speech-like signals), audio signal analysis tools, and automated testing frameworks that systematically evaluate all predictor and stereo mode combinations.

Results

Figure 2.7 summarizes the performance analysis for several predictor types applied to audio files using Golomb coding. Six key metrics are compared across multiple predictor configurations: Stereo_Midside_LINEAR2, Stereo_Independent_LINEAR2, Stereo_Midside_LINEAR3, Stereo_Independent_LINEAR3, Stereo_Midside_LINEAR1, and Stereo_Midside_ADAPTIVE.

- **Compression Ratio:** All predictors yield similar compression ratios, ranging from approximately 1.25:1 to 1.33:1, indicating moderate reduction in file size.
- **Bits per Sample:** There is a significant decrease in bits per sample compared to the original, with resulting values between 12.02 and 12.79 bits per sample, demonstrating efficient bit usage.
- **Compression/Decompression Throughput:** Decompression throughput substantially outpaces compression, with values of 35–37 kB/s for decompression compared to 22–24 kB/s for compression, making these codecs suitable for real-time decoding.
- **Optimal Golomb Parameter (m):** The optimal m parameter varies from 1401 to 1793 among predictors, mostly within a medium range, marking a balance between compression efficiency and entropy coding regularity.

- **Space Savings:** Predictors achieve up to 24.6% space savings, reducing the storage requirements for lossless audio data.
- **Processing Time:** Encoding consistently takes longer (up to 92 ms) than decoding (as low as 51 ms), with adaptive methods demanding slightly higher computation.

This analysis provides a thorough basis for selecting predictor types that best balance compression efficiency and computational speed in Golomb-coded audio codecs

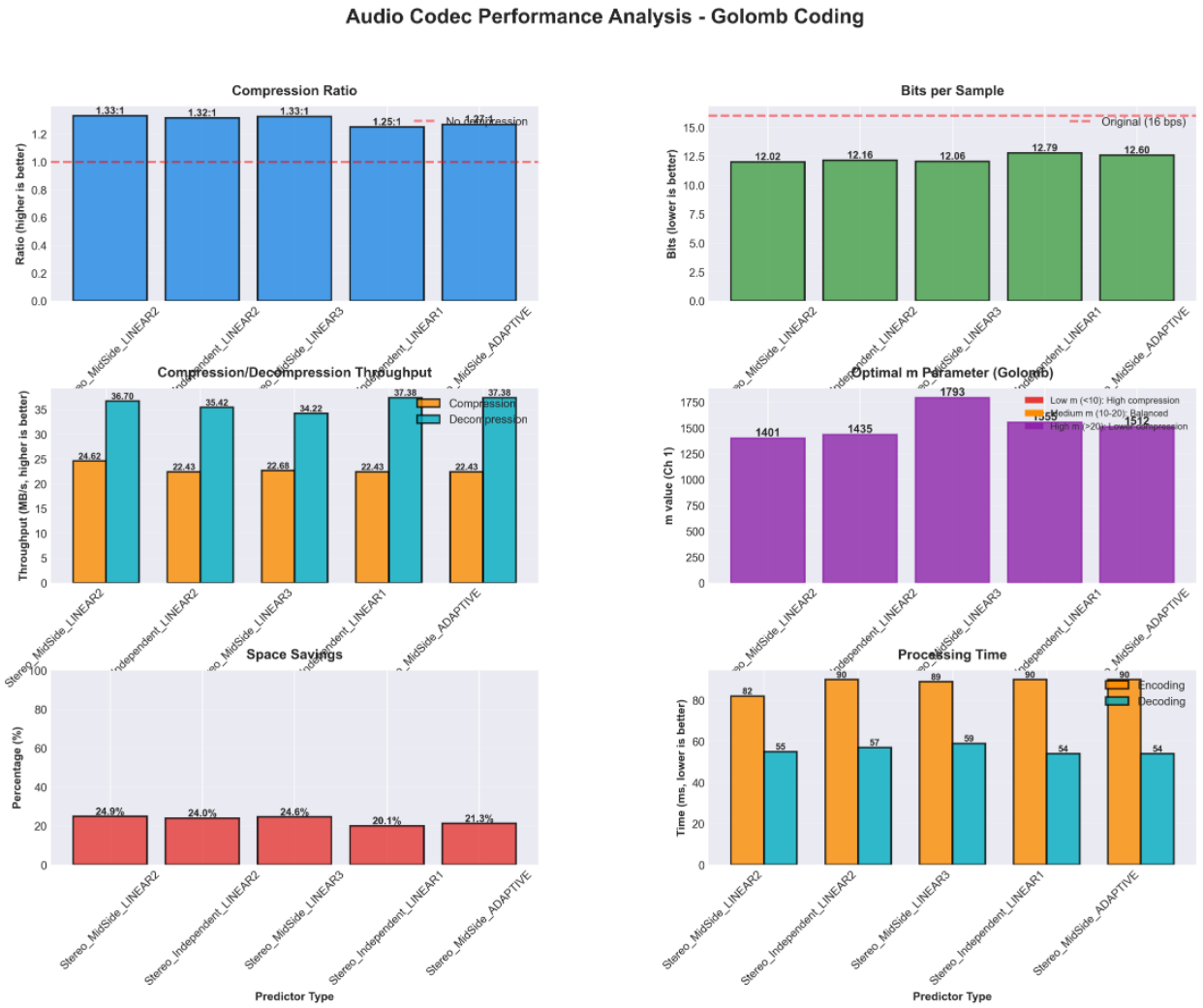

Figure 2.7: Performance Analysis to audio compression and decompression for different predictors

2.4 Part-IV

2.4.1 Exercise 5

Code

The lossless image codec extends the Golomb coding framework to grayscale image compression, exploiting spatial correlation through sophisticated prediction

algorithms. Unlike temporal prediction employed in audio coding, image compression leverages two-dimensional spatial relationships between neighboring pixels.

ImageCodec Class Architecture

The ImageCodec class (`image_codec.hpp and image_codec.cpp`) provides a flexible framework for spatial prediction and entropy coding. The class design incorporates the following key components:

Predictor enumeration: The `PredictorType` enumeration defines six spatial prediction modes, each optimized for different image characteristics:

- **The NONE mode** encodes raw pixel values without prediction, serving as a baseline for comparison;
- **The LEFT and TOP modes** exploit horizontal and vertical correlations respectively;
- **The AVERAGE mode** computes the arithmetic mean of left and top neighbors, balancing directional correlations;
- **The PAETH mode** implements the Portable Network Graphics (PNG) Paeth predictor for adaptive edge-following behavior;
- **The JPEGLS mode** employs the Lossless Joint Photographic Experts Group (JPEG-LS) Median Edge Detector (MED), providing sophisticated edge-aware prediction;
- **The ADAPTIVE mode** dynamically selects optimal predictors on a per-row basis.

Header structure: The `ImageHeader` structure encapsulates image metadata required for decoding. Fields include image dimensions (32-bit width and height), maximum pixel value (8-bit, fixed at 255 for standard grayscale), predictor type (8-bit enumeration value), and the Golomb parameter mm (16-bit integer). This structured header enables the decoder to reconstruct encoding configuration exactly.

Private members: The class maintains state variables including the selected predictor type, initial mm value, adaptive mode flag, and a `CompressionStats` structure recording performance metrics.

Spatial Prediction Implementation

The `predictPixel()` method serves as the central prediction engine, implementing six distinct algorithms with increasing complexity:

- **LEFT predictor:** Returns the immediate left neighbor pixel value, formally $x[i, j] = x[i, j - 1]$, where ii denotes row index and jj denotes column index. This predictor assumes strong horizontal correlation, typical in images with vertical edge structures.
- **TOP predictor:** Returns the immediate top neighbor pixel value, $x[i, j] = x[i - 1, j]$, exploiting vertical correlation present in images with horizontal edge structures.

- **AVERAGE predictor:** Computes the arithmetic mean of the left and top neighbors. The predicted value is $\hat{x}[i,j] = (x[i,j-1] + x[i-1,j])/2$, providing balanced prediction when both horizontal and vertical correlations are significant.

- **PAETH predictor:** Implements the PNG adaptive predictor, which selects among left (a), top (b), and top-left (c) neighbors based on gradient magnitude. The algorithm computes $p = a + b - c$, then evaluates the absolute differences $|p - a|$, $|p - b|$, and $|p - c|$, returning the neighbor that yields the minimum difference. This method follows edge orientations effectively by selecting the neighbor most consistent with local gradient patterns.

- **JPEGLS predictor:** Implements the JPEG-LS median edge detector, which adaptively adjusts the prediction based on edge configurations. The algorithm evaluates the top-left pixel $c = x[i-1, j-1]$ relative to the left and top neighbors. If $c \geq \max(a, b)$, a vertical edge is detected and the predictor returns $\min(a, b)$. If $c \leq \min(a, b)$, a horizontal edge is detected and the predictor returns $\max(a, b)$. Otherwise, the predictor computes the planar prediction $a + b - c$. This context-adaptive approach minimizes prediction errors across diverse image content.

- **ADAPTIVE predictor:** Employs a two-stage selection process to determine the optimal predictor for each image row. The `evaluatePredictor()` helper method computes the mean absolute prediction error for a candidate predictor over a reference row. The `selectBestPredictor()` method evaluates all candidate predictors and selects the one yielding minimum error. This selected predictor is then applied to the current row, and its 8-bit identifier is written to the bitstream to inform the decoder. This approach provides robustness across varying image characteristics at the cost of increased computational complexity and bitstream overhead

Optimal Golomb Parameter Selection

The `calculateOptimalM()` method estimates the optimal Golomb parameter from residual statistics. The method accepts a vector of prediction residuals and computes their mean absolute value. For image prediction residuals, which typically follow a Laplacian distribution. This scaling factor accounts for the relationship between the Laplacian distribution parameter and the geometric distribution parameter underlying Golomb codes.

Encoding Workflow

The `encode()` method orchestrates the complete compression pipeline:

1. **Header Initialization:** The method constructs an `ImageHeader` structure and populates it with image dimensions, maximum value, and predictor type;

2. **Residual computation:** All prediction residuals are pre-computed by iterating through the image in raster scan order, applying the selected predictor, and computing differences between actual and predicted values;

3. **Parameter optimization:** If adaptive mode is enabled, `calculateOptimalM()` estimates the globally optimal mm from the residual distribution;

4. **Header serialization:** The header is written to the output bitstream, with each field encoded according to its specified bit width;

5. **Residual encoding:** For each pixel in raster scan order, the corresponding residual is encoded using a Golomb encoder configured with the `ODD_EVEN_MAPPING` approach for signed integer handling. For `ADAPTIVE` predictor mode, an 8-bit predictor identifier is written at the start of each row;

6. **Optional per-row adaptation:** If configured, the encoder may update the mm parameter periodically by writing a 1-bit update flag followed by a new 16-bit mm value;

7. **Statistics calculation:** Upon completion, compression metrics are computed by comparing original and compressed file sizes.

Decoding Workflow

The `decode()` method performs the inverse transformation:

1. **Header deserialization:** The method reads and parses the file header to extract image dimensions, predictor type, and initial mm parameter;

2. **Decoder initialization:** A Golomb decoder is instantiated with the extracted mm value and `ODD_EVEN_MAPPING` configuration;

3. **Residual decoding:** For each pixel position in raster scan order, the decoder reads the encoded residual from the bitstream. For `ADAPTIVE` predictor mode, the 8-bit predictor identifier is read at the start of each row;

4. **Prediction and reconstruction:** Using the same predictor algorithm employed during encoding, the decoder computes the predicted pixel value based on previously reconstructed neighbors. The decoded residual is added to the prediction, and the result is clamped to the range to ensure valid pixel values;

5. **Parameter updates:** If per-row adaptation was used during encoding, the decoder reads update flags and synchronously updates mm at the same positions;

6. **Image assembly:** Reconstructed pixels are stored in the output vector in row-major order.

The critical property of the decoding process is that predictions operate exclusively on previously reconstructed pixels, ensuring that the decoder replicates the exact prediction values computed by the encoder, thereby guaranteeing lossless reconstruction.

Usage

The ImageCodec interface provides a straightforward Application Programming Interface (API) for image compression and decompression.

Instantiation

The codec is instantiated via its constructor, which accepts three parameters:

```
ImageCodec codec(PredictorType predictor, int m, bool adaptiveM);
```

The `predictor` parameter specifies the spatial prediction algorithm from the `PredictorType` enumeration. The `m` parameter sets the initial Golomb parameter, with a value of 0 indicating automatic estimation. The `adaptiveM` flag enables adaptive parameter optimization when set to true.

Encoding Process

The encoding workflow involves the following steps:

1. **Image Loading:** Load an 8-bit grayscale image into a `std::vector<uint8_t>` container using appropriate I/O utilities. The test suite provides `readPGM()` for reading P2 (ASCII) and P5 (binary) PGM files;
2. **Codec Instatiation:** Create an ImageCodec instance with desired configuration parameters;
3. **Compression:** Invoke the `encode()` method with image data and output filenmame:

```
codec.encode(imageData, width, height, "compressed.golomb");
```

4. **Statistics retrieval:** Access compression metrics via `getLastStats()`:

```
auto stats = codec.getLastStats();
std::cout << "Compression ratio: " << stats.compressionRatio
<< std::endl;
std::cout << "Bits per pixel: " << stats.bitsPerPixel
<< std::endl;
```

Decoding Process

The decoding workflow mirrors the encoding process:

1. **Codec instantiation:** Create an ImageCodec instance (predictor type extracted from file header, so instantiation parameters are not critical);
2. **Decompression:** Invoke the `decode()` method with input filename and output containers:

```
std::vector<uint8_t> decodedImage;
int width, height;
codec.decode("compressed.golomb", decodedImage, width, height);
```

3. **Verification:** Compare original and decoded images to verify lossless reconstruction;
4. **Image Export:** Write reconstructed image to disk using `writePGM()` utility.

Test Suite infrastructure

The test program (`test_image_codec.cpp`) provides a complete environment to validate and compare the image codec:

File I/O Utilities

- Handles both ASCII (P2) and binary (P5) PGM formats;
- Validates file structure (magic number, dimensions, bit depth);
- Loads pixel data into vectors for processing (`readPGM()`);
- Can also save processed images back to disk (`writePGM()`).

This ensures that the test suite can reliably work with external PGM files and produce reproducible outputs.

Synthetic Image Generation

- When no input file is provided, the test suite generates a 256×256 artificial test image.
- Uses multiple structured patterns: horizontal/vertical gradients, diagonal patterns, checkerboards, Gaussian Noise.

These controlled patterns expose strengths and weaknesses of each predictor when encoding different types of image content.

Command-Line Interface
Program supports two modes:

- **Automatic mode:** no parameters → generates synthetic test image.
- **Manual mode:** user provides a PGM file path → processes that image.

This flexibility allows fast automated testing and manual investigation using any real image.

Predictor Comparison

- The test suite evaluates all six predictor algorithms;
- For each predictor it: Encodes the image, Decodes it back, Measures compression performance (statistics printed in the console).

This makes it easy to identify which predictor performs best for different image types.

Results

This image presents a comparative analysis of various predictor types applied to an image codec using Golomb coding, visually summarizing their impact through six performance metrics. The "Compression Ratio" bar chart shows that JPEG-LS and Adaptive predictors achieve the highest compression ratios, both outperforming other approaches, while the Left predictor performs the least effectively. The "Bits per Pixel" plot reveals that all predictors result in substantially lower bpp values than the original 8 bpp, with JPEG-LS and Adaptive again achieving the lowest bpp, and the Left predictor having the highest.

The "Compression/Decompression Throughput" chart indicates that JPEG-LS and Average predictors offer the highest throughput during both compression and

decompression, with Adaptive slightly trailing in decompression speed. In the "Optimal m Parameter" section, all predictors converge on an m value of 2, associated with strong compression for Golomb coding in this dataset.

The "Space Savings" bar chart further supports these findings, highlighting maximum space savings for JPEG-LS (32.8

Overall, JPEG-LS and Adaptive predictors consistently deliver the best compression efficiency and significant space savings, while maintaining competitive processing times and throughput performance.

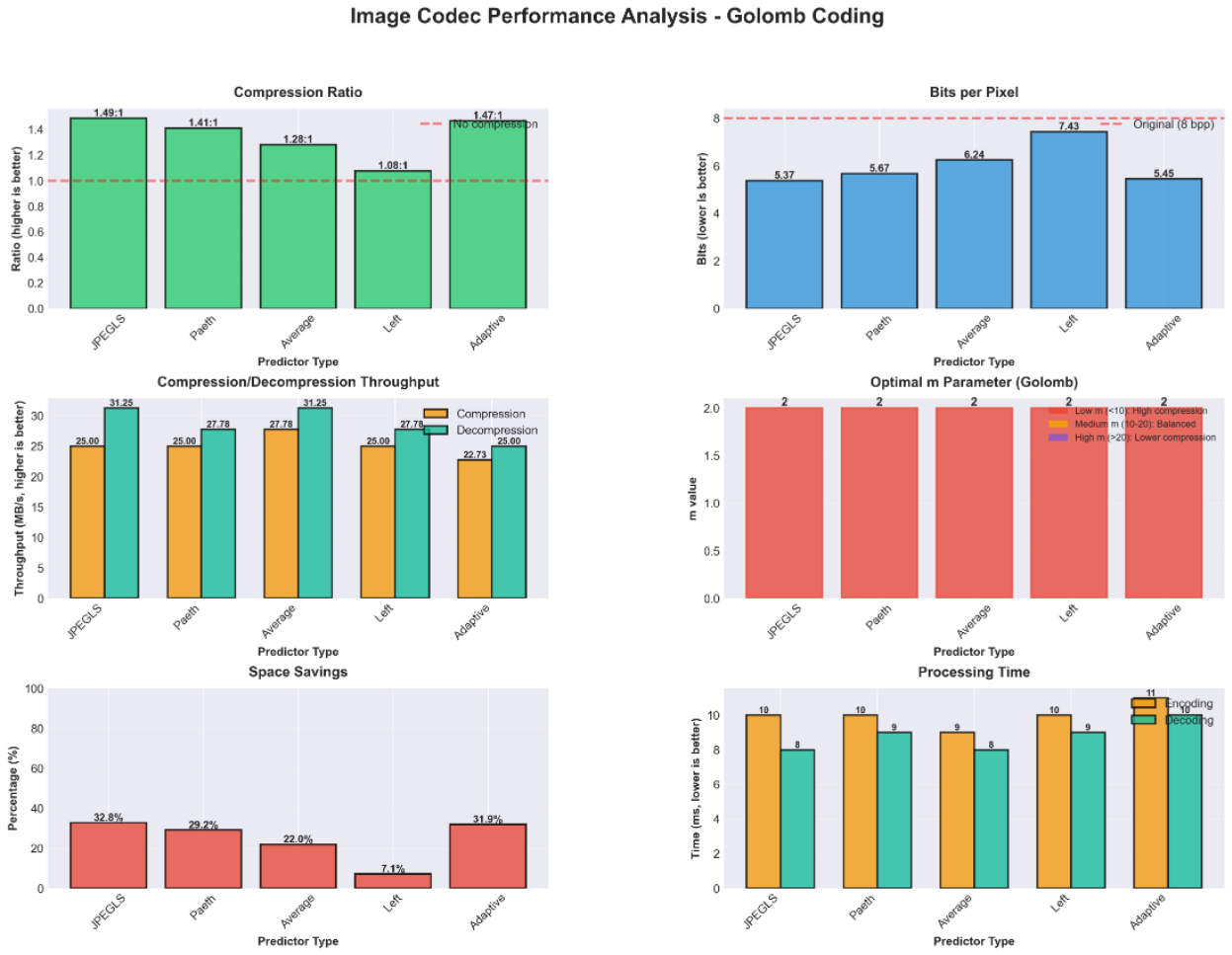

Figure 2.8: Performance Analysis to image compression and decompression for different predictors

Chapter 3

Conclusion

This laboratory work provided comprehensive exploration of lossless compression principles through progressive implementation of image manipulation tools, entropy coding frameworks, and predictive compression systems for audio and visual media. The project successfully integrated theoretical concepts from information theory with practical software engineering, demonstrating how mathematical formulations translate into functional compression algorithms.

All specified objectives were accomplished across the four project phases:

- **Part I** - established foundational competencies in pixel-level image operations, implementing channel extraction and geometric transformations without relying on high-level library functions.
- **Part II** - introduced Golomb entropy coding with custom BitStream infrastructure for bit-level I/O operations.
- **Part III** - developed a lossless audio codec incorporating multiple prediction orders, stereo decorrelation via mid-side transform, and adaptive parameter selection.
- **Part IV** - implemented sophisticated spatial prediction for image compression, including the Paeth predictor and JPEG-LS Median Edge Detector with per-row adaptive predictor selection.

The project successfully demonstrated fundamental principles underlying modern compression systems, where predictive coding exploits correlation structure by encoding residuals rather than raw values, and Golomb entropy coding provides efficient representation for geometric residual distributions.

The modular architecture facilitated code reuse across different media types, while systematic testing validated the lossless nature of all codecs through perfect reconstruction verification. By implementing algorithms from first principles, this work established foundational understanding of compression mechanisms applicable to modern multimedia codecs and information theory research, with potential extensions including arithmetic coding, multi-resolution decomposition, and benchmarking against standardized test datasets