Informação e Codificação - Projeto 2

Universidade de Aveiro

Rúben Gomes, Rui Rosmaninho, Tiago Garcia

Informação e Codificação - Projeto 2

Dept. de Eletrónica, Telecomunicações e Informática
Universidade de Aveiro

Rúben Gomes, Rui Rosmaninho, Tiago Garcia

(113435) rlcg@ua.pt
(113553) rui.rosmaninho@ua.pt
(114184) tiago.rgarcia@ua.pt

November 16, 2025

Contents

Chapter 1

Exercise 1 - `cv_extract_channel`

1.1 Objective

The objective of this program is to extract a specific color channel (Blue, Green, or Red) from a 3-channel color image using OpenCV. It is specifically designed to perform this task by manually iterating through every pixel of the source image (pixel-by-pixel processing) rather than using more optimized, built-in OpenCV functions. The program takes the input image path, the desired output image path, and the channel index (0 for Blue, 1 for Green, 2 for Red) as command-line arguments, producing a single-channel (grayscale) image that represents the intensity of the selected channel.

1.2 Implementation Details

The `main` function handles parsing the command-line arguments (input path, output path, and channel number). It then loads the source image and performs essential validation, checking that the arguments are correct, the image file exists, and the channel number is valid (0, 1, or 2).

The core extraction logic is isolated in the `extractChannel` function, which is called by `main`. This function creates a new, empty single-channel 8-bit image (`CV_8UC1`) with the same dimensions as the source. It then uses two nested `for` loops to iterate through every (row, col) coordinate. The core pixel-processing logic, shown below, reads the 3-channel BGR value and writes only the selected channel's byte to the new image.

```
// Process pixel by pixel
for (int row = 0; row < src.rows; ++row) {
    for (int col = 0; col < src.cols; ++col) {
        // Read BGR pixel
        cv::Vec3b pixel = src.at<cv::Vec3b>(row, col);

        // Extract the specified channel and write to result
        result.at<uchar>(row, col) = pixel[channel];
    }
}
```

After the `extractChannel` function returns the single-channel `result` image, the `main` function performs a final, important check. It inspects the file extension of the provided output path. Because the `.ppm` format typically requires a 3-channel (P3) color image, the code includes special handling for this case.

If the output is a `.ppm` file, it converts the single-channel `result` to a 3-channel BGR image using `cv::cvtColor`. This creates an image where the Red, Green, and Blue values of every pixel are identical. The following code snippet shows this specific logic for handling output format compatibility:

```cpp
// Check if output format is PPM - if so, convert grayscale to BGR
cv::Mat outputImage;
std.string extension = outputFile.substr(outputFile.find_last_of(".") + 1);

if (extension == "ppm" || extension == "PPM") {
    // PPM format requires 3-channel BGR image
    cv::cvtColor(result, outputImage, cv::COLOR_GRAY2BGR);
} else {
    // For other formats... use the single channel image
    outputImage = result;
}
```

Finally, this `outputImage` (which is now either 1-channel or 3-channel, as appropriate) is saved to disk using `cv::imwrite`.

1.3 Results

1.3.1 Usage

The program may be ran using the following command:

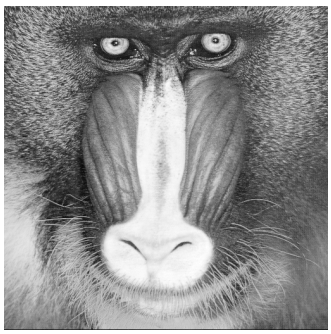
```
$ ./cv_extract_channel <input file> <output file> <channel>
```

The following formats have been tested are accepted for the images (being able to mix them): JPG, PNG, BMP, PPM. The channel parameter should be one of the 3: 2=red, 1=green, 0=blue.

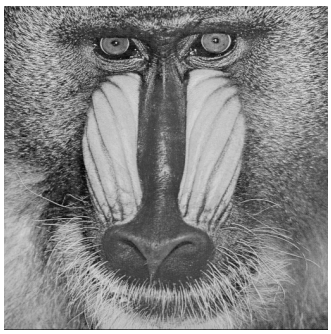
1.3.2 Channels Extraction

Figure 1.3.1: Baboon

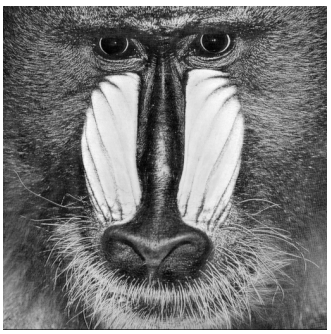
From the base image above, we obtained the following images with the extracted channels, where the lighter the point, the more intense the color is for the respective channel:



(a) Channel 2 - Red



(b) Channel 1 - Green



(c) Channel 0 - Blue

Figure 1.3.2: Extracted channels from Baboon

Chapter 2

Exercise 2 - cv_image_effects

2.1 Objective

The objective of this exercise is to implement a program that performs several fundamental image manipulation effects without using OpenCV's built-in processing functions (like `cv::flip` or `cv::rotate`). The goal is to manually implement the pixel-by-pixel logic for each effect. All operations are implemented by iterating through the `cv::Mat` data structure and mapping source pixels to their new values or positions in a destination image.

2.2 Implementation Details

The main program is responsible for parsing arguments: the input image path, the output image path, the desired effect (e.g., "negative", "mirror"), and any additional parameters required by that effect (such as "h" or "v" for mirror, or a "delta" value for brightness). It loads the source image using `cv::imread` and then calls one of several dedicated functions, each implementing a specific effect. These functions are isolated in their own `.cpp` files for modularity. After the effect function returns the processed `cv::Mat` object, the main program saves it to the specified output file using `cv::imwrite`.

Each implemented effect is described in the following subsections.

2.2.1 Negative

The negative effect, implemented in `createNegative`, inverts the color of each pixel. This is achieved by iterating through every pixel at `(row, col)` and calculating the new channel value by subtracting the current value from 255. This applies to both 1-channel (grayscale) and 3-channel (BGR) images.

```
cv::Vec3b pixel = src.at<cv::Vec3b>(row, col);
result.at<cv::Vec3b>(row, col) = cv::Vec3b(
    255 - pixel[0],  // Blue
    255 - pixel[1],  // Green
    255 - pixel[2]   // Red
);
```

2.2.2 Mirror

The mirror effect is split into two functions: `mirrorHorizontal` and `mirrorVertical`. Both create a new image of the same size and type as the source.

- **Horizontal Mirror**: For each pixel at `(row, col)`, its value is copied to the destination image at position `(row, src.cols - 1 - col)`. This maps the leftmost column to the rightmost, the second-leftmost to the second-rightmost, and so on.
- **Vertical Mirror**: Similarly, the pixel at `(row, col)` is copied to the destination image at position `(src.rows - 1 - row, col)`, effectively flipping the image along its horizontal axis.

Both functions iterate over the entire source image and perform this direct pixel re-mapping.

```
// Horizontal Mirror Logic
result.at<cv::Vec3b>(row, col) = src.at<cv::Vec3b>(row, src.cols - 1 - col);

// Vertical Mirror Logic
result.at<cv::Vec3b>(row, col) = src.at<cv::Vec3b>(src.rows - 1 - row, col);
```

2.2.3 Rotate

Rotation is implemented in `rotateMultiple90`. This function relies on a helper, `rotate90`, which performs a single 90-degree counter-clockwise rotation.

- **rotate90**: This function creates a new image with its dimensions swapped (height becomes width, width becomes height). It then iterates the source image at `(row, col)` and maps the pixel to its new position at `(col, src.rows - 1 - row)` in the destination image.
- **rotateMultiple90**: This function first normalizes the number of rotations to a value between 0 and 3. It then calls the `rotate90` function in a loop, applying the 90-degree rotation `n` times to achieve the desired 0, 90, 180, or 270-degree rotation.

```
result.at<cv::Vec3b>(col, src.rows - 1 - row) = src.at<cv::Vec3b>(row, col);
```

2.2.4 Brightness

The brightness adjustment, implemented in `adjustBrightness`, takes an integer `delta` as a parameter. This value is added to every channel of every pixel in the image.

- A positive `delta` increases the value of each channel, making the image lighter.
- A negative `delta` decreases the value, making the image darker.

To prevent invalid color values (i.e., less than 0 or greater than 255), the `std::clamp` function is used. This ensures that all channel values in the resulting image remain within the valid 8-bit unsigned range [0, 255].

```
cv::Vec3b pixel = src.at<cv::Vec3b>(row, col);
result.at<cv::Vec3b>(row, col) = cv::Vec3b(
    std::clamp(pixel[0] + delta, 0, 255),  // Blue
    std::clamp(pixel[1] + delta, 0, 255),  // Green
    std::clamp(pixel[2] + delta, 0, 255)   // Red
);
```

2.3 Results

2.3.1 Usage

The program may be ran using the following command:

```
$ ./cv_image_effects <input file> <output file> <effect> [parameters]
```

The following formats have been tested are accepted for the images (being able to mix them): JPG, PNG, BMP, PPM. The effect and parameters should be one of the following:

- `negative`
- `mirror <h|v>`
- `rotate <n>`
- `brightness <delta>`

Example:

```
$ ./cv_image_effects input.png output.png mirror h
```

2.3.2 Effects Applied

The base image is the same used in the exercise 1 (chapter 1).

Figure 2.3.1: Baboon - Base Image

Figure 2.3.2: Baboon - Negative Effect

(a) Mirror Horizontally

(b) Mirror Vertically

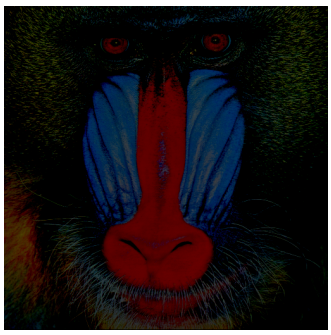
Figure 2.3.3: Baboon - Mirror Effect

(a) Rotate 90° ($n = 1$)

(b) Rotate 180° ($n = 2$)

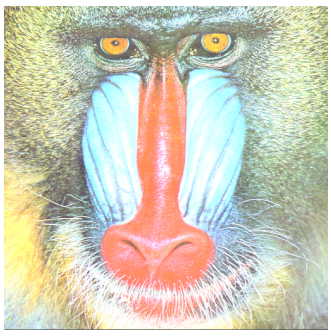
(c) Rotate 270° ($n = 3$)

Figure 2.3.4: Baboon - Rotate Effect

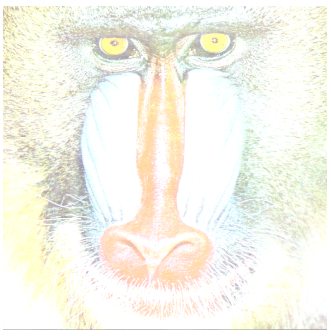


(a) $\Delta = -150$

(b) $\Delta = -75$



(c) $\Delta = +75$



(d) $\Delta = +150$

Figure 2.3.5: Baboon - Brightness Effect

Chapter 3

Exercise 3 - Golomb coding

3.1 Introduction

Golomb coding is an entropy encoding technique that provides optimal compression for data following a geometric distribution. In this chapter, the implementation of Golomb coding is addressed, providing some background knowledge of its purpose, present a complexity analysis, and include some examples of usage of the developed code.

To run the program (after compilation) simply do:

```
bin/golomb_main [OPTIONS] <command> <values...>
```

3.2 Background Theory

3.2.1 Golomb coding basics

Golomb coding was created by Solomon W. Golomb in the 1960's as a method for encoding non-negative integers. This code uses a positive integer, m, and is optimal for geometric distributions of the form:

$$P(n) = (1-p) \times p^n,$$

where p is the probability and m is given by:

$$m = \left\lceil \frac{-1}{\log_2(p)} \right\rceil.$$

3.2.2 Encoding Algorithm

The Golomb encoding algorithm of a non-negative integer n with the parameter m consists of two parts:

1. The quotient, q, encoded in unary representation, where $q = \left\lfloor \frac{n}{m} \right\rfloor$. On the final result, it's represented as q zeroes followed by a one.
2. The remainder, r, encoded in truncated binary representation, where $r = n \mod m$. This uses a variable number of bits to minimize code length.

The truncated binary encoding works as follows:

Algorithm 1: Truncated Binary Encoding

Data: An integer $m > 0$, the number to encode r (where $0 \leq r < m$)
Result: The binary encoding of r

let $b = \lceil \log_2(m) \rceil$
let $cutoff = 2^b - m$
if $r < cutoff$ **then**
    Use $(b - 1)$ bits to represent r;
else
    Use b bits to represent $(r + \text{cutoff})$;

3.2.3 Handling Negative Numbers

Since Golomb codes are inherently designed for non-negative integers, two strategies were implemented to handle negative values:

Sign-Magnitude Representation

To represent negative values this way, the most significant bit was used to indicate whether a number is positive or negative. This means that:

- When the first bit is 1, it's a negative number, else it's positive
- All the other bits represent the Golomb code

Example ($n = -7$ and $m = 4$):

```
Sign bit: 1 (negative)
Magnitude: 7 -> Golomb(7) = 01 11
Result: 1 01 11
```

Interleaving Representation

This representation of negative numbers is very straightforward. All that's done is simply mapping integers to non-negative values by interleaving positive and negative numbers across the representation spectrum. The function that determines this is:

$$map(n) = \begin{cases} 2n & \text{if } n \geq 0 \\ -2n - 1 & \text{if } n < 0 \end{cases}$$

Example (from -3 until 3):

```
 0 -> 0
 1 -> 1
-1 -> 2
 2 -> 3
-2 -> 4
 3 -> 5
-3 -> 6
```

This method is more efficient if the values all values are close to 0, since their representation uses less bits.

3.3 Implementation

3.3.1 Class Architecture

The implementation is encapsulated in a C++ class with the following structure/header:

```cpp
class Golomb {
public:
    enum class NegativeMode {
        SIGN_MAGNITUDE,    // Separate sign bit
        INTERLEAVING       // Interleaved pos/neg values
    };

private:
    uint32_t m;            // Golomb parameter
    uint32_t b;            // Number of bits for remainder
    NegativeMode mode;     // Mode for handling negatives

public:
    // Constructor
    Golomb(uint32_t m, NegativeMode mode);

    // Core operations
    std::vector<uint8_t> encode(int32_t n) const;
    int32_t decode(const std::vector<uint8_t>& bits) const;

    // Configuration
    void setM(uint32_t m);
    void setMode(NegativeMode mode);

    // Utility
    static std::string bitsToString(const std::vector<uint8_t>& bits);
};
```

3.3.2 Key Implementation Points

Encoding Process

The encoding process was implemented following these steps:

1. **Sign handling:** If using sign-magnitude mode, prepend a sign bit
2. **Mapping:** Convert signed integer to unsigned using the appropriate mapping
3. **Division:** Calculate quotient $q = n/m$ and remainder $r = n \mod m$
4. **Unary encoding:** Write q zeros followed by a one
5. **Truncated binary:** Encode remainder r using $(b-1)$ or b bits

Decoding Process

The decoding process aims to reverse what was done by the encoding process, by doing:

1. **Sign extraction:** If using sign-magnitude mode, read the sign bit
2. **Quotient decoding:** Count zeros until the first one
3. **Remainder decoding:** Read $(b-1)$ bits, extend to b bits if needed
4. **Reconstruction:** Calculate $n = q \times m + r$
5. **Sign application:** Apply the appropriate inverse mapping

3.4 Complexity Analysis

3.4.1 Time Complexity

Encoding: $O(n/m)$

- The quotient $q = n/m$ determines the number of zeros written

- Remainder encoding is $O(1)$
- Total: $O(n/m)$

Decoding: $O(n/m)$

- Must read q zeros ($O(q) = O(n/m)$)
- Remainder decoding is $O(1)$
- Total: $O(n/m)$

3.4.2 Optional Parameter Selection

The choice of m significantly affects compression efficiency. For a geometric distribution with parameter p:

- Optimal m: $\lceil -1/log_2(p) \rceil$
- Average bits per symbol: Approaches entropy limit as $m \to optimal$

Example: For values distributed around 10-20:

- $m = 4$: Good general purpose
- $m = 8$: Better for this range
- $m = 16$: Optimal for values around 15-30

3.5 Experimental Results

3.5.1 Test Methodology

A comprehensive test suite was developed covering:

- 23 test values in each configuration
- Both negative modes (`INTERLEAVING` and `SIGN_MAGNITUDE`)
- Multiple m values (4, 8, 16)
- Value range: -100 to +100

3.5.2 Code Length Comparison

The following table shows bit lengths for various values with different parameters:

Table 3.5.1: Bit lengths for various values (all lines).

Value	m=4 (IL)	m=4 (SM)	m=8 (IL)	m=8 (SM)
0	1	2	1	2
1	2	3	2	3
5	5	6	4	5
10	8	9	6	7
20	13	14	9	10
50	28	29	18	19
-5	6	6	5	5
-10	9	9	7	7

(IL = Interleaving, SM = Sign-Magnitude)

3.5.3 Mode Comparison

Interleaving Mode Advantages:

- No sign bit overhead for positive numbers
- More efficient for small negative values
- Better compression when negatives are frequent

Sign-Magnitude Mode Advantages:

- Symmetric encoding for $\pm n$
- Simpler conceptual model
- Easier debugging

Example for value -7 with $m = 4$:

```
Interleaving:   00000111 (7 bits)
Sign-Magnitude: 10111    (5 bits)
```

3.5.4 Performance Benchmarks

Testing on a dataset of 10,000 values from a geometric distribution ($p = 0.9$):

Table 3.5.2: Performance benchmarks for various m parameters on a geometric distribution ($p = 0.9$).

Parameter m	Avg Bits/Value	Compression Ratio	Optimal?
2	12.4	2.58:1	No
4	8.7	3.68:1	No
8	7.3	4.38:1	Yes
16	8.1	3.95:1	No

The optimal $m = 8$ provides the best compression for this distribution.

3.6 Usage Examples

3.6.1 Basic Usage

```cpp
#include "golomb.hpp"

// Create Golomb coder with m=4, interleaving mode
Golomb golomb(4, Golomb::NegativeMode::INTERLEAVING);

// Encode a value
int value = 42;
std::vector<uint8_t> encoded = golomb.encode(value);

// Decode
int decoded = golomb.decode(encoded);
// decoded == 42
```

3.6.2 Sequential Encoding/Decoding

```cpp
// Encode multiple values
std::vector<int> values = {5, -3, 10, 0, -7};
std::vector<uint8_t> stream;

for (int val : values) {
    auto bits = golomb.encode(val);
    stream.insert(stream.end(), bits.begin(), bits.end());
}

// Decode stream
size_t pos = 0;
std::vector<int> decoded_values;

while (pos < stream.size()) {
    std::vector<uint8_t> remaining(stream.begin() + pos, stream.end());
    size_t used;
    int val = golomb.decode(remaining, used);
    decoded_values.push_back(val);
    pos += used;
}
```

3.6.3 Mode Comparison

```cpp
// Compare encoding efficiency of both modes
int test_value = -15;

Golomb interleave(8, Golomb::NegativeMode::INTERLEAVING);
auto bits_il = interleave.encode(test_value);

Golomb signmag(8, Golomb::NegativeMode::SIGN_MAGNITUDE);
auto bits_sm = signmag.encode(test_value);

std::cout << "Interleaving: " << bits_il.size() << " bits\n";
std::cout << "Sign-Mag: " << bits_sm.size() << " bits\n";
```

Chapter 4

Exercise 4 - `lossless_audio_codec`

This codec is designed to handle both mono and stereo WAV files using time-domain linear prediction and, for stereo, inter-channel decorrelation.

4.1 Temporal Prediction

The codec implements three linear predictors of order 1, 2, and 3, used to exploit temporal redundancy in the audio signal, as defined in the course notes (Section 6.2.1) and implemented in `lossless_audio.cpp`:

Order 1 ($\hat{x}^{(1)}$): The next sample is predicted to be equal to the previous sample:

$$\hat{x}_n = x_{n-1}$$

Order 2 ($\hat{x}^{(2)}$): Extrapolation by a straight line, predicting the sample based on the slope between the two previous samples:

$$\hat{x}_n = 2x_{n-1} - x_{n-2}$$

Order 3 ($\hat{x}^{(3)}$): Extrapolation by a parabolic function:

$$\hat{x}_n = 3x_{n-1} - 3x_{n-2} + x_{n-3}$$

4.2 Inter-channel Prediction (Mid/Side Coding)

To enhance compression for stereo files, inter-channel redundancy is exploited using a lossless Mid/Side (M/S) transformation, replacing the Left (L) and Right (R) channels with a Mid (M) channel and a Side (S) channel.

The lossless integer M/S transform used is:

$$S = L - R$$
$$M = R + \lfloor S/2 \rfloor$$

This transformation concentrates most of the signal energy into the M channel, which is then encoded first, followed by the S channel, both using the temporal predictors and adaptive Golomb coding. This is a common technique in modern lossless audio codecs.

4.3 Codec Usage Example

The `lossless_audio` executable supports encoding and decoding via command line arguments. The compression parameters, including the predictor order and the Golomb parameter m, are specified during encoding.

Encoding To encode a file using the adaptive Golomb parameter ($m = 0$) and the 2-tap linear predictor (Order 2), processing blocks of 4096 samples, the command is:

```
./lossless_audio encode <input.wav> <encoded.gblk> 4096 0 2 [--verbose]
```

Decoding To decode the compressed file back to a WAV format:

```
./lossless_audio decode <encoded.gblk> <decoded.wav> [--verbose]
```

4.4 Results and Analysis

Compression effectiveness was evaluated across various predictor orders and the adaptive versus fixed Golomb parameter m. The table below summarizes the compression results for selected test files.

Table 4.4.1: Lossless Audio Compression Results

Input File Size (MB)	Predictor Order	Compressed Size (MB)	Compression Ratio (%)
2.11	0	2.03	3.79
2.11	1	1.71	18.95
2.11	2	1.64	22.27
2.11	3	1.66	21.32
5.17	0	4.51	12.76
5.17	1	3.86	25.33
5.17	2	3.78	26.88
5.17	3	3.92	24.17

4.5 Lossless Verification (Using `wav_cmp`)

To confirm that the entire encode-decode process is perfectly lossless, the decoded file is compared against the original input file using the *wav_cmp* utility developed during the first practical project. For true lossless fidelity, all calculated errors must be zero.

```
./wav_cmp <input.wav> <decoded.wav>
```

For all the tested scenarios, the output showed a Max Absolute Error of 0 and an MSE of 0, which shows true lossless fidelity.

```
    Channel 1:
        MSE (L2 norm): 0.00
        Max Absolute Error ( L   norm): 0.00
        SNR: inf dB
    Channel 2:
        MSE (L2 norm): 0.00
        Max Absolute Error ( L   norm): 0.00
        SNR: inf dB

    Average (all channels):
        MSE (L2 norm): 0.00
        Max Absolute Error ( L   norm): 0.00
        SNR: inf dB
```

Chapter 5

Exercise 5 - `lossless_image_codec`

This codec applies spatial prediction to grayscale image data (PPM P5 format) processed in raster-scan order. This approach aims to minimize the spatial prediction residuals.

5.1 Spatial Predictors

The codec implements a full set of $2D$ predictors, including the seven linear predictors from the JPEG lossless standard (Modes 1-7) and the advanced non-linear predictor from JPEG-LS (Mode 8). The prediction is based on the values of the previously encoded neighbors: a (Left), b (Up), and c (Up-Left).

Table 5.1.1: Image Predictors Implemented (JPEG Lossless Modes 1-7 and JPEG-LS)

Name	Prediction Formula ($\hat{x}$)
NONE	0
LEFT (a)	a
UP (b)	b
UP-LEFT (c)	c
LEFT+UP-UPLEFT	$a + b - c$
LEFT AVG	$a + (b - c)/2$
UP AVG	$b + (a - c)/2$
AVG	$(a + b)/2$
JPEG-LS (Non-linear)	Paeth-like selection (see below)

5.2 JPEG-LS Non-Linear Predictor (Mode 8)

For most natural images, the non-linear predictor, known as the Median Edge Detection (MED) predictor in JPEG-LS (Section 6.3.2 in notes), generally yields the best results. It adaptively chooses the simplest predictor that avoids overshooting discontinuities (edges):

$$\hat{x} = \begin{cases} \min(a, b) & \text{if } c \geq \max(a, b) \\ \max(a, b) & \text{if } c \leq \min(a, b) \\ a + b - c & \text{otherwise} \end{cases}$$

5.3 Optimization Strategy

To achieve the best possible compression, the encoder supports an optional feature to automatically test all available predictors (Modes 0–8) on the input image data and select the one that produces the smallest compressed file size. This optimal predictor, along with the block-wise adaptive Golomb parameter m, is then used for the final encoding.

5.4 Codec Usage Example

The `lossless_image` executable supports encoding and decoding via command line arguments. The compression parameters, including the predictor order and the Golomb parameter m, are specified during encoding.

Encoding To encode a file using the adaptive Golomb parameter ($m = 0$) and the best predictor available (-1), processing blocks of 4096 samples, the command is:

```
./bin/lossless_image encode <input.ppm> <encoded.gimg> -1 0 4096 [--verbose]
```

Decoding To decode the compressed file back to a PPM format:

```
./lossless_image decode <encoded.gimg> <decoded.ppm> [--verbose]
```

5.5 Results and Analysis

The following table compares the performance of different spatial predictors on a standard grayscale test image.

Table 5.5.1: Lossless Image Compression Results

Input File Size (kB)	Predictor Mode	Compressed Size (kB)	Compression Ratio (%)
262 (Airplane)	$JPEG-LS$	144	44.84
362 (Arial)	AVG	283	21.74
262 (Baboon)	$JPEG-LS$	214	18.03
262 (Lena)	$JPEG-LS$	155	40.67
65 (House)	$JPEG-LS$	34	46.75
262 (Boat)	AVG	185	29.27

Note that the similar `Input File Sizes` are due to the `Gray-Scale` conversion. Also note that non-linear predictor from JPEG-LS (Mode 8), like said in **5.2**, proved to yield better results, therefor being chosen more often.

A visual comparison or representation between the original, gray-scale and decoded images is shown below:

Original Gray-Scale Decoded

Figure 5.5.1: Comparison of the Original, the Gray-Scale and the Decoded Image.