



University of Aveiro

Information and Coding

Lab work n^o 2

Henrique Cruz 103442

Diogo Borges 102954

Piotr Bartczak 130327

November 17, 2025

Contents

1	Introduction	2
2	Exercise 1	4
	2.1 Approach and usage.	4
	2.2 Examples.	4
3	Exercise 2	6
	3.1 Approach and usage.	6
	3.2 Examples.	7
4	Exercise 3	8
	4.1 Objectives	8
	4.2 Implementation details	8
	4.3 Example Usage	12
	4.4 Results and Discussion	12
5	Exercise 4	16
	5.1 Method and Decisions	16
	5.2 Evaluation	16
	5.3 Conclusions	17
6	Exercise 5	18
	6.1 Methodology	18
	6.2 Results	18
	6.3 Comparison with Existing Codecs	19
7	Project Information	21

1 Introduction

This report presents a compact study on lossless coding and basic image processing, implemented in modern C++ with OpenCV for image I/O and manual, pixel-by-pixel operations for the core algorithms. The goals are twofold: (i) to build small utilities that expose fundamental image operations (channel extraction and elementary transforms) without relying on high-level library calls, and (ii) to design a simple, pedagogical lossless codec based on spatial prediction and Golomb coding of residuals, then analyse its behaviour against established codecs.

Concretely, the work comprises:

- **Exercise 1:** a tool to extract a single colour channel (B, G, or R) from a colour image and save a single-channel output, implemented by explicit pixel iteration.
- **Exercise 2:** pixel-wise image transforms (negative, horizontal/vertical mirror, rotations by multiples of 90° , and brightness adjustment), also written as direct per-pixel mappings.
- **Exercise 3:** a parameterised Golomb encoder/decoder for signed integers, supporting distinct negative-value mappings and a small CLI to verify round-trips on concatenated streams.
- **Exercise 4:** a lossless audio coding study using temporal prediction (mono) and inter-channel prediction (stereo) with Golomb coding of residuals and an adaptive choice of m , discussed and compared with standard lossless audio codecs (e.g., FLAC).
- **Exercise 5:** a lossless grayscale image codec that predicts each pixel from neighbours (left or JPEG-LS-style median predictor) and Golomb-codes the residuals. The encoder searches a small set of m values and embeds the chosen parameter in a simple binary container for deterministic decoding.

Evaluation focuses on two kinds of evidence:

- **Runtime:** wall-clock times for the utilities on 512×512 images, measured with the Unix `time` command (single-threaded execution).
- **Compression:** compression ratios for the predictive Golomb codec on several natural and simple images; visual and numerical verification of lossless round-trips; comparison to standard lossless codecs (notably PNG; we comment on JPEG-LS where relevant).

The remainder of this report is organised as follows. Section 1 demonstrates channel extraction and provides visual examples. Section 2 details the pixel-wise transforms

and their timings. Section 3 summarises the Golomb coder and verifies correctness on integer sequences. Section 4 discusses the lossless audio codec (prediction + Golomb), methodology, and comparisons (e.g., FLAC). Section 5 presents the predictive Golomb image codec, reports compression ratios and timings, and contrasts results with standard lossless codecs.

2 Exercise 1

This exercise demonstrates a simple tool developed to extract a single color channel from a color image and save it as a single-channel image. The goal is to provide a small utility that lets the user inspect the contribution of each color plane (B, G, R) to the final picture and to produce examples for further processing.

2.1 Approach and usage.

The implemented program reads a color image (BGR order) and writes a single-channel image where each output pixel holds the value of the selected input channel. The tool performs the extraction by iterating the image pixels and copying the requested channel value into a new single-channel matrix. This avoids using convenience routines that would perform the extraction internally; the implementation is intentionally pixel-by-pixel to illustrate basic image access and manipulation.

Typical usage (from project root):

```
./build/extract_color_channel images-ppm/lena.ppm  
images/figures/lena_red.pgm 2
```

The measured runtime for a single-channel extraction on our test image (Lena, 512×512 PPM) was on the order of tens of milliseconds. Example timings observed on the test machine:

- Red channel extraction: 0.15 s
- Green channel extraction: 0.06 s
- Blue channel extraction: 0.07 s

2.2 Examples.

Figure 1 shows the original image and the three extracted channels. The single-channel results are stored as grayscale images where darker/ lighter tones correspond to smaller/ larger channel values.



(a) Original



(b) Red channel (index 2)



(c) Green channel (index 1)



(d) Blue channel (index 0)

Figure 1: Original image and extracted single-channel images produced by the channel-extraction utility.

3 Exercise 2

This exercise presents a collection of basic pixel-wise image transforms implemented as a second utility. The goal is to provide elementary image processing operations implemented by direct pixel manipulation (no use of high-level OpenCV transformations). The operations implemented are:

1. Negative (invert intensities)
2. Horizontal mirror (left-right flip)
3. Vertical mirror (top-bottom flip)
4. Rotation by multiples of 90° (implemented for $k \cdot 90$ where k is integer)
5. Brightness adjustment (additive delta applied to all channels, clamped to $[0, 255]$)

3.1 Approach and usage.

Each operation iterates through the input pixels and writes the transformed value into an output image buffer. For rotations the destination image size is adjusted (width/height swapped for $90/270$ degree rotations) and indices are mapped explicitly. The brightness adjustment clamps values to the valid 8-bit range.

Typical usage (from project root):

```
./build/image_transform images-ppm/lena.ppm images/figures/lena_neg.png  
neg  
./build/image_transform images-ppm/lena.ppm images/figures/lena_rot90.png  
rotate 1  
./build/image_transform images-ppm/lena.ppm  
images/figures/lena_bright_plus50.png bright 50
```

Measured execution times for these transforms on the same test image were consistently in the tens of milliseconds (single-threaded):

- Negative: 0.07 s
- Horizontal mirror: 0.05 s
- Vertical mirror: 0.05 s
- Rotate 90° : 0.05 s
- Brightness +50: 0.05 s

3.2 Examples.

Figure 2 shows a selection of results produced by the transform utility: negative, horizontal and vertical mirrors, a 90° rotation and a brightness increase.



(a) Negative



(b) Mirror (horizontal)



(c) Mirror (vertical)



(d) Rotate 90°



(e) Brightness +50

Figure 2: Examples of pixel-by-pixel transforms produced by the utility. These illustrate common basic image operations that can be composed for simple image editing or used as preprocessing before analysis.

4 Exercise 3

4.1 Objectives

The goal of this exercise was to implement a C++ class capable of performing Golomb coding on integer values. The implementation needed to provide functions for converting an integer into its corresponding Golomb bit sequence and for decoding a bit sequence back into the original integer.

A key requirement was that the coder should be fully parameterized by the value of m , allowing the user to adapt the code length and distribution to different data characteristics. Since the exercise also concerns signed values, the implementation had to support two strategies for handling negative integers:

- **Sign-and-magnitude:** encode the sign using a single bit and apply Golomb coding to the absolute value.
- **Positive/negative interleaving:** map signed integers to non-negative integers using the sequence $0, -1, +1, -2, +2, \dots$, and then apply Golomb coding to the mapped value.

Another objective was to build a small command-line interface (CLI) that allows users to test the coder directly. The CLI should accept parameters for m , the chosen negative-handling mode, and whether to perform encoding or decoding. It should also display the resulting bit sequences and support decoding of concatenated streams to verify correctness.

4.2 Implementation details

The implementation is split into three source files:

- `golomb.hpp` – public declarations (classes, enums and methods).
- `golomb.cpp` – implementation of the coder, bit I/O helpers and decoding logic.
- `golomb_main.cpp` – a small CLI that parses `-m` and `-mode` and performs `encode` / `decode` operations.

Below are the main components and the important implementation parts.

Core classes The design uses three small classes:

- **BitWriter** — collects output bits in an ASCII string of '0'/'1' for debugging:

```

class BitWriter {
public:
    std::string bits; // '0' / '1'
    void writeBit(bool b);
    void writeBits(uint64_t value, int count);
};

```

- **BitReader** — reads bits from the ASCII bit string and tracks the current position:

```

class BitReader {
public:
    const std::string &bits;
    size_t pos;
    BitReader(const std::string &bstr);
    bool readBit();
    uint64_t readBits(int count);
};

```

- **Golomb** — the encoder/decoder with parameter `m` and a negative-number mode:

```

class Golomb {
public:
    explicit Golomb(uint64_t m_, NegativeMode negMode_ =
        NegativeMode::INTERLEAVED);
    std::string encode(int64_t value) const;
    std::pair<int64_t, size_t> decode(const std::string &bits) const;
private:
    uint64_t m, b, cutoff;
    NegativeMode negMode;
    void encodeUnsigned(uint64_t n, BitWriter &w) const;
    uint64_t decodeUnsigned(BitReader &r) const;
};

```

Initialization: computing `b` and `cutoff` On construction the code computes:

$$b = \lceil \log_2 m \rceil, \quad \text{cutoff} = 2^b - m$$

These values are used to implement truncated-binary remainder coding. In the code this is done with a loop that increments `b` until `(1ULL << b) >= m` and then sets `cutoff = (1ULL << b) - m`.

Unsigned encoding / decoding (core algorithm) The unsigned Golomb algorithm is implemented in two functions: `encodeUnsigned` and `decodeUnsigned`. Key steps:

- Compute quotient and remainder:

```
uint64_t q = n / m;
uint64_t r = n % m;
```

- **Quotient** is written using unary coding: `q` zeros followed by a 1.
- **Remainder** uses truncated binary with the precomputed `b` and `cutoff`:

```
if (r < cutoff) {
    // write r in (b-1) bits
    w.writeBits(r, b-1);
} else {
    // write r+cutoff in b bits
    w.writeBits(r + cutoff, b);
}
```

- Special case `m == 1`: the coder uses unary coding (`n` zeros then a one).

Decoding mirrors these steps:

- Read unary quotient by counting zeros until the terminating 1.
- Read either `b-1` or `b` bits to reconstruct the remainder according to whether the first read value is below `cutoff`.
- Combine quotient and remainder: `value = q * m + r`.

Signed values: two handling modes Signed support is implemented exactly as required, with a user-selectable `NegativeMode`:

- **Sign-and-magnitude**: write a single sign bit (0 = non-negative, 1 = negative) and then Golomb-code the absolute value.

```
// sign bit first, then absolute value as unsigned Golomb
w.writeBit(sign ? true : false);
encodeUnsigned(abs(value), w);
```

- **Positive/negative interleaving**: map signed integers to non-negative integers using the sequence

$$0, -1, +1, -2, +2, \dots$$

implemented by:

```
static uint64_t toInterleaved(int64_t x) {
    if (x >= 0) return uint64_t(x) << 1;
    return uint64_t((-x << 1) - 1);
}
```

Then the mapped non-negative value is encoded with the unsigned routine. Decoding applies the inverse map.

Decoding follows the inverse logic of the encoding pipeline. The decoder first counts the number of leading 1-bits until it encounters the terminating 0; this count yields the quotient. After consuming the stop bit, it reads the next b or $b-1$ bits, depending on the truncated-binary boundary, and reconstructs the remainder. The original non-negative value is then recovered as $q \cdot m + r$.

If sign handling is set to **sign-magnitude**, the decoder simply reads one additional bit after the Golomb code and applies the sign to the magnitude. For the interleaving strategy, the decoder inverts the mapping by checking whether the recovered integer is even or odd and applying the inverse rule:

$$x = \begin{cases} \frac{k}{2}, & \text{if } k \text{ is even} \\ -\frac{k+1}{2}, & \text{if } k \text{ is odd} \end{cases}$$

In both cases, the bitstream is fully reversible because every step of the mapping and coding process is injective.

CLI and verification The CLI (`golomb_main.cpp`) handles:

- Parsing `-m <value>` and `-mode <sign|interleave>`.
- Two operations: `encode <ints...>` and `decode <bitstring>`.
- For `encode`, each integer is encoded separately, the program prints each encoding, concatenates them and then decodes the concatenated stream to verify the round-trip.
- For `decode`, the program accepts a raw ASCII bitstring (spaces tolerated) and repeatedly decodes values until all bits are consumed or an error occurs.

Error handling and edge cases

- The code throws `std::runtime_error` when the bit reader runs out of bits unexpectedly.
- The constructor checks `m >= 1` and throws `std::invalid_argument` otherwise.

- The CLI validates user input (missing `-m`, invalid integer strings, empty bitstrings, etc.) and returns non-zero exit codes on errors.

4.3 Example Usage

General command format The Golomb coder can encode multiple integers in a single invocation. The general usage pattern is:

```
./build/golomb -m <m_value> -mode <mode> encode <list_of_integers>
```

Examples Interleave mode:

```
./build/golomb -m 3 -mode interleave encode 0 -1 5 10
```

Sign-magnitude mode:

```
./build/golomb -m 4 -mode sign encode 0 -7 12
```

4.4 Results and Discussion

In this section we present the results of running the Golomb encoder and decoder with different parameters and input sequences. Each example shows how the choice of m and the negative-value strategy affects the produced bitstream. After each run, the decoder was used to verify that the original values were recovered correctly.

Below are the four tested commands, their outputs, and a discussion of what they show.

Run 1: Interleave mode with $m = 3$

Command

```
./build/golomb -m 3 -mode interleave encode 0 -1 5 10
```

Output

Parameters: m=3 mode=INTERLEAVED

Value[0] = 0 -> bits: 10 (len=2)

Value[1] = -1 -> bits: 110 (len=3)

Value[2] = 5 -> bits: 000110 (len=6)

Value[3] = 10 -> bits: 000000111 (len=9)

Concatenated bitstream (20 bits):

```
10110000110000000111
```

Decoding:

Decoded[0] = 0

Decoded[1] = -1

Decoded[2] = 5

Decoded[3] = 10

Round-trip OK.

Discussion: This example uses the positive/negative interleaving mapping, which converts signed integers to non-negative ones before applying Golomb coding. This mapping places 0, -1, +1, -2, +2, ... in increasing order, so small positive and negative values remain close together. As a result, both 0 and 1 produce short bit sequences. Larger values, like 5 and 10, need more bits because their quotient $q = \lfloor n/m \rfloor$ grows. The decoder consumes the exact number of bits for each value, confirming that the stream is uniquely decodable.

Run 2: Sign-and-magnitude with $m = 4$

Command

```
./build/golomb -m 4 -mode sign encode 0 -7 12
```

Output

Parameters: m=4 mode=SIGN_MAGNITUDE

Value[0] = 0 -> bits: 0100 (len=4)

Value[1] = -7 -> bits: 10111 (len=5)

Value[2] = 12 -> bits: 0000100 (len=7)

Concatenated bitstream (16 bits):

0100101110000100

Decoding:

Decoded[0] = 0

Decoded[1] = -7

Decoded[2] = 12

Round-trip OK.

Discussion: Here each value begins with a sign bit: 0 for non-negative and 1 for negative. This makes the output straightforward to interpret but can increase the number of bits for small values compared with interleave mode. For example, encoding 0 already

requires 4 bits because of the sign bit plus the unary prefix and remainder. The values -7 and 12 produce different lengths depending on their magnitude after the sign bit is removed. Again, the decoder reconstructs all values correctly, showing that the sign bit does not interfere with Golomb decoding.

Run 3: Sign-and-magnitude with $m = 2$ (repeated zeros and small values)

Command

```
./build/golomb -m 2 -mode sign encode 0 0 0 -1 1 -2 2
```

Output

Parameters: m=2 mode=SIGN_MAGNITUDE

```
Value[0] = 0 -> bits: 010 (len=3)
Value[1] = 0 -> bits: 010 (len=3)
Value[2] = 0 -> bits: 010 (len=3)
Value[3] = -1 -> bits: 111 (len=3)
Value[4] = 1 -> bits: 011 (len=3)
Value[5] = -2 -> bits: 1010 (len=4)
Value[6] = 2 -> bits: 0010 (len=4)
```

Concatenated bitstream (23 bits):

```
01001001011101110100010
```

Decoding:

All values decoded correctly.

Round-trip OK.

Discussion: This run tests repeated values and small signed integers. Since $m = 2$, the quotient changes quickly, meaning many values fall into small q regions and produce short codes. The three zeros produce identical codes, but because Golomb codes are prefix-free under this construction, the decoder can separate them without ambiguity. The sign-and-magnitude method makes 1 and -1 differ only by the first bit, and similar behavior appears for ± 2 . This example confirms that the implementation handles repeated symbols and short codes without alignment issues.

Run 4: Interleave with $m = 5$ (boundary values)

Command

```
./build/golomb -m 5 -mode interleave encode -3 0 9 10 11
```

Output

Parameters: m=5 mode=INTERLEAVED

```
Value[0] = -3 -> bits: 0100 (len=4)
Value[1] = 0 -> bits: 100 (len=3)
Value[2] = 9 -> bits: 0001110 (len=7)
Value[3] = 10 -> bits: 0000100 (len=7)
Value[4] = 11 -> bits: 0000110 (len=7)
```

Concatenated bitstream (28 bits):

```
0100100000111000001000000110
```

Decoding:

All values decoded correctly.

Round-trip OK.

Discussion: This input includes values close to a boundary where the quotient changes. With $m = 5$, the values 9, 10, and 11 all lie between two multiples of m : they are near $m \cdot 1 = 5$ and $m \cdot 2 = 10$. Because of this, their quotients and remainders behave differently, but they still produce codes of equal length. The negative number -3 is handled through interleaving, giving it a non-negative mapped value before coding. The decoder confirms that even with values crossing the boundary between different (q, r) regions, the stream can be decoded without errors.

Summary

Across all tests, the encoder and decoder produced consistent and correct results. The examples show how different choices of m affect the bit lengths, especially for values near multiples of m . They also highlight the practical differences between sign-and-magnitude and interleave mode: the former adds an explicit sign bit, while the latter tries to keep small signed values close together. The experiments confirm that the implementation works as intended for a range of normal and edge-case inputs.

5 Exercise 4

The purpose of this exercise is to create a lossless audio compression module using Golomb coding of the prediction residuals from the previous exercise. Lossless characteristic is accomplished by first applying a simple predictor to each audio sample to remove redundancy, and then encoding the resulting residuals with a Golomb entropy coder. The Golomb parameter m is chosen adaptively based on the residual statistics to make the coding more efficient. The design considers both temporal prediction and inter-channel prediction for stereo audio.

The codec operates on 16-bit PCM WAV files and provides two commands:

```
golomb_codec encode input.wav output.glb
golomb_codec decode input.glb output.wav
```

5.1 Method and Decisions

Procedure

For each test audio file:

1. File was compressed using the developed codec.
2. It was decompressed back to WAV.
3. File sizes and processing times were recorded.

The codec uses:

- a simple linear predictor (temporal prediction for mono, inter-channel for stereo),
- adaptive Golomb coding of residuals (EMA-based parameter selection).

5.2 Evaluation

Three WAV files were chosen for the testing, one mono music, one stereo music and one speech mono. Compression rate was compared with FLAC and ZIP codecs.

Audio File	Original	Own Codec	Ratio (%)	FLAC	ZIP
sample.wav	2.11	1.81	86%	1.49	2.07
sample_mono.wav	1.05	0.83	79%	0.74	1.03
speech.wav	2.11	1.04	49%	0.87	1.54

Table 1: Compression results for several test audio files (sizes in MB).

Speech sample compresses much better than the rest due to strong sample predictability, while music has moderate compression. FLAC performs better than the codec created

here, although still doesn't reduce the size by much. ZIP performs worse than all audio-specific codecs, however in speech case, it does bring some gains to the table, unlike with music, where we can observe basically no gains at all.

Audio File	Encode Time	Decode Time	FLAC Encode	FLAC Decode
sample.wav	0.13	724.45	0.02	0.01
sample_mono.wav	0.06	140.88	0.01	0.00
speech.wav	0.08	368.83	0.02	0.01

Table 2: Processing times (in seconds) for encoding and decoding audio files using the custom codec and FLAC.

In terms of encoding speed, the custom codec is really fast, however decode times are very very high, possibly due to implementation specifics and lack of general optimization of both golomb coding and the codec itself. In FLAC's case, the decoder is performing even better than the encoder.

5.3 Conclusions

Codec successfully performs lossless compression and decompression of PCM audio. Compression strength varies with signal predictability, it's high on speech and a little worse on music. FLAC performs and compresses more efficiently, as expected from renowned and optimized software, although the developed codec still achieves meaningful compression using just simple golomb coding.

6 Exercise 5

This exercise focuses on a lossless grayscale image codec based on predictive coding and Golomb coding of prediction residuals. The main goal is to study what can be achieved with a simple, custom scheme in terms of compression ratio and speed, and to compare it with existing lossless codecs.

6.1 Methodology

The codec operates on 8-bit grayscale images. For each pixel, a predictor estimates its value using already-reconstructed neighbours, and the difference (prediction residual) is entropy-coded using Golomb codes. Two spatial predictors were considered:

- **Left predictor** — uses the pixel to the immediate left.
- **Median (JPEG-LS style) predictor** — takes the median of {left, top, left+top-top-left}, which adapts better to edges and smooth areas.

For each image, the encoder computes residuals and searches a small set of candidate Golomb parameters m ; it selects the one that minimizes the total bit count. The final bitstream stores image dimensions, chosen predictor and m , followed by the packed code bits, ensuring fully deterministic decoding.

We evaluated the codec on several 512×512 grayscale images, including natural scenes (Lena, Peppers, Baboon, Monarch, Kodak samples) and simpler graphics with flat regions. For each case we recorded:

- **Compression ratio:** $\frac{\text{original bytes}}{\text{compressed bytes}}$.
- **Encode/decode times:** measured with Unix `time`.

Typical usage (from the project root):

```
# encode (predictor: 0=left, 1=median; default=1)
./build/image_codec encode images/figures/lena_red.pgm
images/lena_codec.gimg 1

# decode
./build/image_codec decode images/lena_codec.gimg
images/figures/lena_red_decoded.pgm
```

6.2 Results

On natural images like Lena, the predictive Golomb codec achieved compression ratios in the range of roughly 1.8:1 to 2.5:1 with the median predictor. The left-only predictor

generally performed slightly worse, particularly on images with strong diagonal structures or complex textures where the median predictor better follows local gradients.

For smoother images containing large uniform regions and simple edges, residuals were highly concentrated around zero; the automatically selected m tended to be small, producing shorter codes and higher compression. For highly textured images such as Baboon or Monarch, residuals were more spread out, which limited the compression gain; nevertheless, reconstruction remained strictly lossless.

In terms of speed, encoding and decoding completed in tens of milliseconds per 512×512 image on the test machine, even with the small search over m . Decoding was consistently faster than encoding because it does not perform parameter search.

To verify lossless reconstruction, Figure 3 compares the original grayscale channel with the decoded image; they are visually indistinguishable and match pixel-by-pixel.



Figure 3: Round-trip comparison for the predictive Golomb codec. The decoded image matches the original visually and numerically (lossless).

6.3 Comparison with Existing Codecs

- **PNG (lossless)** — Combines simple predictors with LZ77+Huffman coding. In our trials, PNG typically matched or slightly exceeded our compression on complex natural images, benefiting from mature modeling and backend entropy coding. On simpler images, our codec was competitive and sometimes marginally better.
- **JPEG-LS family** — The median predictor mirrors JPEG-LS design, but our codec omits context modeling and other refinements. Standard JPEG-LS implementations generally achieve better ratios across diverse content at comparable or higher speeds.

Overall, a lightweight combination of spatial prediction and Golomb coding already delivers meaningful lossless compression with low complexity and fast execution. The comparison with PNG and JPEG-LS highlights the gains from stronger modeling and mature entropy coding, positioning our codec as an educational and practical baseline.

7 Project Information

Division of Work

All group members contributed equally to the development of the project.

Repository

The complete source code, including all modules, examples, and test files, is available at the following repository:

`https://github.com/borges7555/ic-proj2`

How to Build and Test

In order to build and test the project there is a README file in the repository.