

# Informação e Codificação

Relatório do Lab work nº 2

Eduardo Alves: nºmec 104179

Bernardo Marujo: nºmec 107322

Simão Almeida: nºmec 113085

---



[eduardoalves@ua.pt](mailto:eduardoalves@ua.pt)  
[bernardomarujo@ua.pt](mailto:bernardomarujo@ua.pt)  
[spsa@ua.pt](mailto:spsa@ua.pt)

[Link Repositório GitHub](#)

Universidade de Aveiro  
Departamento de Electrónica, Telecomunicações e Informática  
2025

# Contents

<b>1</b>	<b>Introdução</b>	<b>3</b>
<b>2</b>	<b>Parte 1</b>	<b>4</b>
2.1	Exercício 1 - Extração de um canal de cor de uma imagem . . . . .	4
2.1.1	Contexto . . . . .	4
2.1.2	Implementação . . . . .	4
2.1.3	Resultados . . . . .	5
2.2	Exercício 2a - Criar a versão negativa de uma imagem . . . . .	8
2.2.1	Contexto . . . . .	8
2.2.2	Implementação . . . . .	8
2.2.3	Resultados . . . . .	11
2.3	Exercício 2b - Criar uma versão espelhada de uma imagem: (a) horizontalmente; (b) verticalmente . . . . .	12
2.3.1	Contexto . . . . .	12
2.3.2	Implementação . . . . .	12
2.3.3	Resultados . . . . .	14
2.4	Exercício 2c - Rodar uma imagem por um múltiplo de 90º . . . . .	14
2.4.1	Contexto . . . . .	14
2.4.2	Implementação . . . . .	15
2.4.3	Resultados . . . . .	17
2.5	Exercício 2d - Aumentar (mais luz) / diminuir (menos luz) a intensidade de uma imagem . . . . .	17
2.5.1	Contexto . . . . .	17
2.5.2	Implementação . . . . .	17
2.5.3	Resultados . . . . .	19
<b>3</b>	<b>Parte 2</b>	<b>19</b>
3.1	Exercício 3 - Golomb coding . . . . .	19
3.1.1	Contexto . . . . .	19
3.1.2	Implementação . . . . .	20
<b>4</b>	<b>Parte 3</b>	<b>22</b>
4.1	Exercício 4 - Lossless audio codec based on Golomb coding . . . . .	22
4.1.1	Contexto . . . . .	22
4.1.2	Implementação do Codificador . . . . .	22
4.1.3	Implementação do descodificador . . . . .	24
4.1.4	Resultados . . . . .	25
<b>5</b>	<b>Parte 4</b>	<b>28</b>
5.1	Exercício 5 - Lossless image codec for grayscale images based on Golomb coding . . . . .	28
5.1.1	Contexto . . . . .	28
5.1.2	Implementação do Codificador . . . . .	28
5.1.3	Implementação do descodificador . . . . .	30
5.1.4	Resultados . . . . .	31

## List of Figures

1	Comparação entre imagens pré e pós extração do canal 0 (Azul). . . . .	6
2	Comparação entre imagens pré e pós extração do canal 1 (Verde). . . . .	6
3	Comparação entre imagens pré e pós extração do canal 2 (Vermelho). . .	7
4	Comparação entre a imagem após extração azul e a extração vermelha. .	8
5	Comparação entre a imagem original e a sua versão negativa. . . . .	11
6	Comparação entre a imagem original e as versões espelhadas. . . . .	14
7	Comparação entre a imagem original e as versões com efeito de rotação. .	17
8	Comparação entre a imagem original e as versões com brilho alterado. . .	19

# 1 Introdução

O objetivo deste projeto é desenvolver, testar e demonstrar uma aplicação prática que combine técnicas de processamento de imagem, codificação de informação e compressão sem perdas, aplicadas a ficheiros de áudio e imagem. Para tal, o projeto será implementado em várias fases progressivas, integrando desde a manipulação básica de imagens com a biblioteca OpenCV até à criação de codecs baseados em codificação de Golomb para compressão eficiente de dados multimédia.

O projeto está dividido em quatro partes principais:

- **Parte I – Processamento de Imagem com OpenCV:** Nesta fase inicial, serão implementados programas utilizando a biblioteca OpenCV para manipular imagens a nível de píxel. O principal objetivo é compreender o funcionamento interno da representação digital de imagens, realizando operações básicas como:
  - Extração de um canal de cor (R, G ou B) de uma imagem, criando uma imagem monocanal resultante;
  - Geração da versão negativa de uma imagem;
  - Criação de versões espelhadas, tanto horizontal como verticalmente;
  - Rotação da imagem por múltiplos de 90°;
  - Aumento e diminuição da luminosidade através da modificação direta da intensidade dos píxeis.
- **Parte II – Codificação Golomb:** Nesta etapa será desenvolvida uma classe em C++ responsável pela implementação da *codificação de Golomb*, um método de compressão entropia-ótima para valores inteiros. Esta classe deverá incluir funções para:
  - Codificar um número inteiro numa sequência de bits;
  - Descodificar uma sequência de bits num número inteiro;
  - Gerir o parâmetro  $m$ , que controla a eficiência da codificação;
  - Tratar números negativos através de dois métodos alternativos: *sign and magnitude* e *interleaving*.
- **Parte III – Codec de Áudio Sem Perdas:** Nesta fase será implementado um codec de áudio sem perdas baseado na codificação de Golomb dos resíduos de predição. O sistema deverá suportar ficheiros de áudio mono e estéreo, explorando tanto predição temporal como predição entre canais no caso estéreo. Além disso, o parâmetro  $m$  poderá ser fixo ou determinado de forma adaptativa durante o processo de codificação/descodificação, de modo a otimizar a taxa de compressão.
- **Parte IV – Codec de Imagem Sem Perdas:** Por fim, será desenvolvido um codec sem perdas para imagens em tons de cinzento, também baseado na codificação de Golomb dos resíduos de predição. O objetivo é obter a melhor compressão possível, através da escolha adequada de preditores e valores de  $m$ , garantindo a reconstrução perfeita da imagem original após a descodificação.

## 2 Parte 1

### 2.1 Exercício 1 - Extração de um canal de cor de uma imagem

#### 2.1.1 Contexto

O objetivo deste exercício é introduzir o uso da biblioteca OpenCV para leitura, manipulação e escrita de imagens a nível de píxel. O programa deve ser capaz de extrair um canal de cor específico (R, G ou B) de uma imagem RGB e gerar uma nova imagem contendo apenas esse canal, representada em tons de cinzento. Esta tarefa permite compreender o acesso direto aos valores individuais dos píxeis numa estrutura matricial e reforça o entendimento sobre a representação digital de imagens.

#### 2.1.2 Implementação

O programa foi implementado em C++ utilizando a biblioteca OpenCV. O ficheiro principal recebe três argumentos na linha de comandos:

```
<nome_programa> <input_image> <output_image> <channel_number>
```

onde `channel_number` indica o canal a extrair (0 = Azul, 1 = Verde, 2 = Vermelho).

**Validação dos argumentos** Logo no início, o programa verifica se o número de argumentos está correto e se o canal indicado é válido (entre 0 e 2). Caso contrário, uma mensagem de erro é apresentada:

```
if (argc != 4) {
    std::cerr << "Usage: " << argv[0]
        << " <input_image> <output_image> <channel_number>" << std::endl;
    return -1;
}
```

**Leitura da imagem** A imagem é carregada em memória através da função `cv::imread()` com o modo `cv::IMREAD_COLOR`, garantindo que é lida como uma imagem de três canais (B, G, R):

```
cv::Mat inputImage = cv::imread(inputPath, cv::IMREAD_COLOR);

if (inputImage.empty()) {
    std::cerr << "Error: Could not read the input image." << std::endl;
    return -1;
}
```

**Criação da imagem de saída** É criada uma nova matriz com o mesmo tamanho da imagem original, mas com apenas um canal (`CV_8UC1`), destinado a armazenar as intensidades do canal escolhido:

```
cv::Mat singleChannel(inputImage.size(), CV_8UC1);
```

**Leitura e escrita píxel a píxel** O processamento principal é feito através de dois ciclos aninhados que percorrem cada píxel da imagem original. Para cada píxel, é obtido um vetor `cv::Vec3b` contendo as três componentes (B, G, R). Em seguida, é extraído o valor correspondente ao canal selecionado e copiado para a imagem de destino:

```
for (int y = 0; y < inputImage.rows; ++y) {
    for (int x = 0; x < inputImage.cols; ++x) {
        cv::Vec3b pixel = inputImage.at<cv::Vec3b>(y, x);
        unsigned char channelValue = pixel[channelToExtract];
        singleChannel.at<unsigned char>(y, x) = channelValue;
    }
}
```

Este método garante a leitura e escrita *píxel a píxel*, conforme o enunciado do exercício, sem recorrer a funções automáticas do OpenCV (como `cv::split()`), permitindo assim o controlo total sobre o processo.

**Escrita do resultado** Por fim, a imagem resultante é guardada num novo ficheiro através da função `cv::imwrite()`, e o sucesso da operação é verificado:

```
bool success = cv::imwrite(outputPath, singleChannel);

if (!success) {
    std::cerr << "Error: Could not save the output image." << std::endl;
    return -1;
}
```

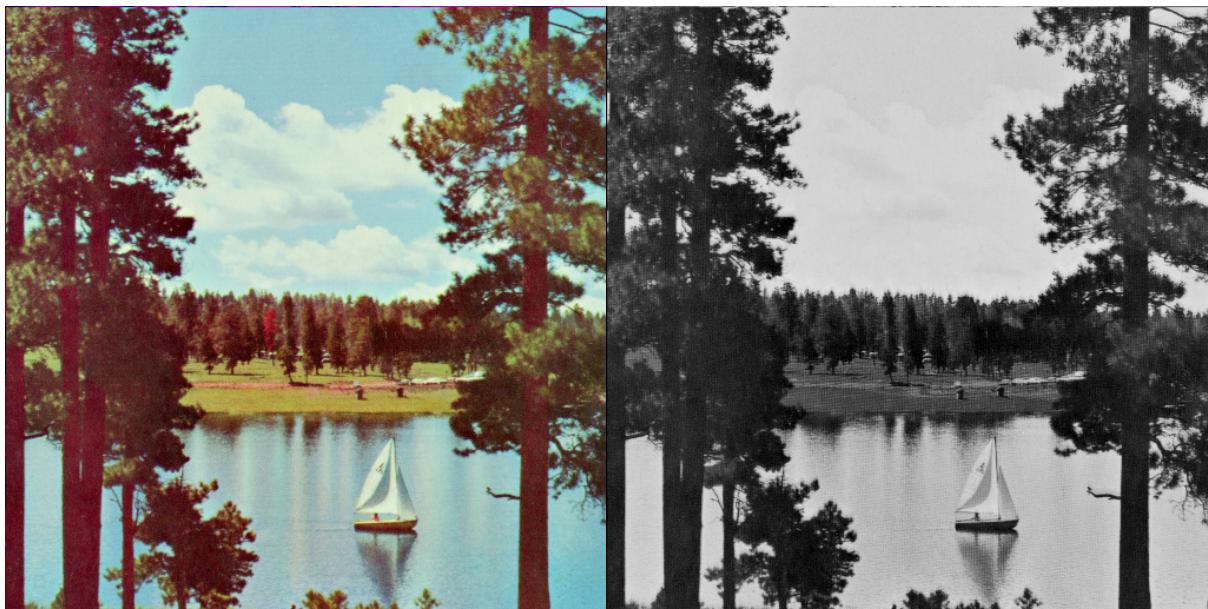
**Execução do programa** O programa pode ser executado, por exemplo, da seguinte forma:

```
./extract_channel input.jpg output.jpg 0
```

Neste caso, será criada uma imagem `output.jpg` que contém apenas as intensidades do canal azul da imagem original.

### 2.1.3 Resultados

O programa foi testado com várias imagens coloridas. A extração do canal azul (`channel = 0`) resultou numa imagem em tons de cinzento, onde as regiões originalmente mais azuladas aparecem mais claras. Da mesma forma, ao selecionar o canal verde (1) ou vermelho (2), observa-se para cada uma das áreas dessas cores, o mesmo efeito. Como podemos analisar nas imagens seguintes.



(a) Imagem original

(b) Imagem após extração do canal 0.

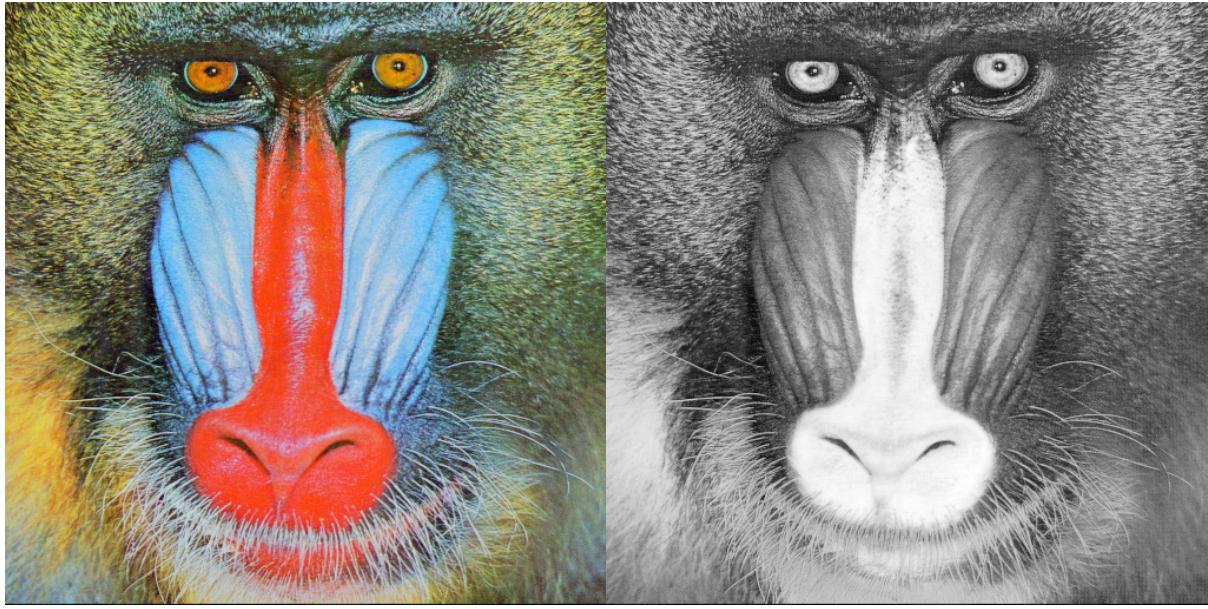
Figure 1: Comparação entre imagens pré e pós extração do canal 0 (Azul).



(a) Imagem original

(b) Imagem após extração do canal 1.

Figure 2: Comparação entre imagens pré e pós extração do canal 1 (Verde).



(a) Imagem original

(b) Imagem após extração do canal 2.

Figure 3: Comparação entre imagens pré e pós extração do canal 2 (Vermelho).

Através das imagens anteriores, é possível observar claramente o efeito da extração de cada canal de cor. Em todas as comparações, verifica-se que a imagem resultante é apresentada em tons de cinzento, representando a intensidade do canal selecionado em cada píxel. As regiões mais claras correspondem a áreas com maior contribuição do respetivo canal na imagem original, enquanto as regiões mais escuras indicam baixa intensidade dessa componente.

Nas imagens 1, a extração do canal 0 (Azul) evidencia zonas como o céu e a água, que apresentam tonalidades claras devido à forte presença de azul. Já nas imagens 2, a extração do canal 1 (Verde) evidencia as áreas com os pimentos verdes tornam-se mais brilhantes, refletindo a predominância dessa componente na imagem original, enquanto os pimentos vermelhos se mostram com tons mais escuros, isto visto que a extração foi do canal 1 correspondente à cor verde. Por fim, nas imagens 3, a extração do canal 2 (Vermelho) destaca as regiões com maior presença dessa cor, que nos caso proposto corresponde ao nariz do babuíno que se tornam significativamente mais claros na imagem resultante.

Conseguimos também reparar melhor nas diferenças entre extrações diferentes comparando a extração da cor azul na imagem do babuíno com a extração da cor vermelha na comparação das imagens abaixo cuja figura é a 4:

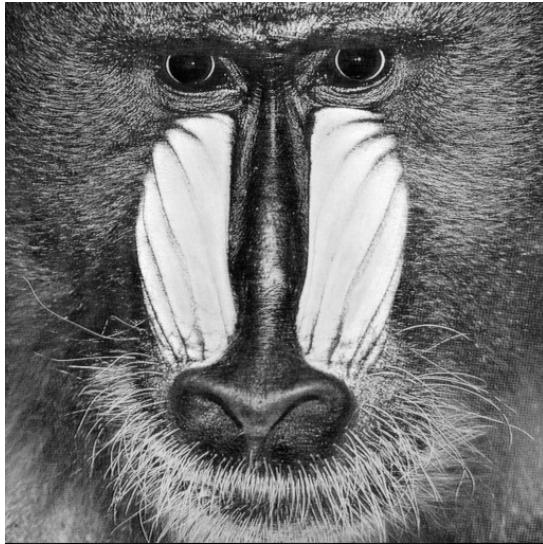


Imagen após extração azul

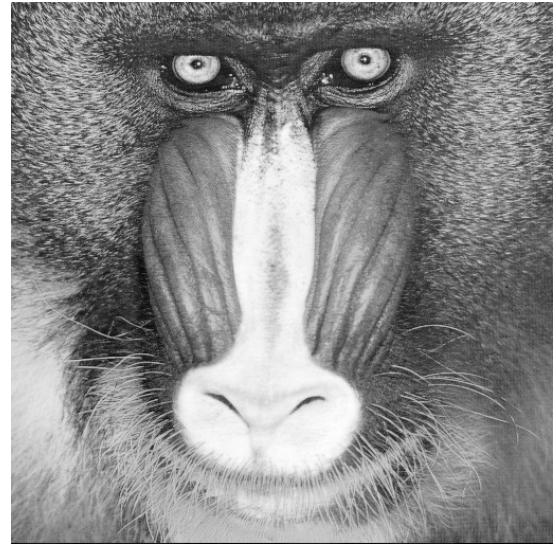


Imagen após extração vermelha

Figure 4: Comparação entre a imagem após extração azul e a extração vermelha.

De forma geral, os resultados confirmam o correto funcionamento do programa, produzindo imagens monocanal que representam fielmente a intensidade de cada componente de cor da imagem original. A análise visual das imagens demonstra que a implementação realiza a separação de canais de forma precisa, preservando as dimensões e a correspondência espacial entre o original e o resultado processado.

## 2.2 Exercício 2a - Criar a versão negativa de uma imagem

### 2.2.1 Contexto

O objetivo deste exercício é gerar a versão negativa de uma imagem no formato PPM. O processo consiste em inverter os valores de cada componente de cor (vermelho, verde e azul), de acordo com as expressões:

$$R' = \text{max\_color\_val} - R, \quad G' = \text{max\_color\_val} - G, \quad B' = \text{max\_color\_val} - B$$

Isto faz com que as regiões claras da imagem original se tornem escuras e vice-versa, produzindo o efeito negativo.

### 2.2.2 Implementação

A função principal é definida de modo a aceitar dois argumentos: o ficheiro de entrada e o ficheiro de saída.

```
int main(int argc, char** argv) {  
  
    if (argc != 3) {  
        std::cerr << "Usage: " << argv[0] << " <input_image.ppm> <output_image.ppm>"  
        return 1;  
    }  
}
```

Aqui, o programa verifica se o utilizador forneceu os argumentos corretos. Caso contrário, apresenta uma mensagem de erro e termina a execução.

De seguida, são criadas as variáveis que armazenam os caminhos dos ficheiros e efetuadas as tentativas de abertura:

```
std::string inputPath = argv[1];
std::string outputPath = argv[2];

std::ifstream infile(inputPath);
if (!infile.is_open()) {
    std::cerr << "ERROR: Could not open " << inputPath << std::endl;
    return 1;
}

std::ofstream outfile(outputPath);
if (!outfile.is_open()) {
    std::cerr << "ERROR: Could not create " << outputPath << std::endl;
    return 1;
}
```

O programa valida se os ficheiros foram abertos com sucesso. Caso contrário, emite um erro informativo.

Em seguida, é lido o cabeçalho do ficheiro PPM, que contém informações essenciais: o tipo de formato (P3 ou P6), a largura, a altura e o valor máximo de cor.

```
std::string magic_number;
int width, height, max_color_val;

infile >> magic_number;
if (magic_number != "P3" && magic_number != "P6") {
    std::cerr << "ERROR: Input is not a valid PPM file." << std::endl;
    return 1;
}

bool isBinary = (magic_number == "P6");
```

O programa identifica o formato da imagem e armazena-o na variável `isBinary`, determinando se o ficheiro deve ser tratado como texto ou binário.

```
while (infile.peek() == '\n' || infile.peek() == ' ') infile.ignore();
while (infile.peek() == '#') {
    std::string comment;
    std::getline(infile, comment);
    while (infile.peek() == '\n' || infile.peek() == ' ') infile.ignore();
}

infile >> width >> height >> max_color_val;
```

Esta secção assegura que apenas os dados relevantes (dimensões e valor máximo) sejam processados, evitando erros de leitura.

Depois de processar o cabeçalho, o programa escreve o cabeçalho da nova imagem negativa.

```
outfile << "P6\n";
outfile << "# Created by C++ negative program\n";
outfile << width << " " << height << "\n";
outfile << max_color_val << "\n";
```

Mesmo que a imagem original seja P3, a imagem de saída é escrita em formato binário P6 para maior eficiência.

Segue-se o cálculo do número total de píxeis e o processamento de cada um. Para o formato binário (P6), cada componente de cor é lida como um `unsigned char` e invertida.

```
int pixel_count = width * height;

if (isBinary) {
    for (int i = 0; i < pixel_count; ++i) {
        unsigned char r, g, b;
        infile.read(reinterpret_cast<char*>(&r), 1);
        infile.read(reinterpret_cast<char*>(&g), 1);
        infile.read(reinterpret_cast<char*>(&b), 1);

        unsigned char neg_r = max_color_val - r;
        unsigned char neg_g = max_color_val - g;
        unsigned char neg_b = max_color_val - b;

        outfile.write(reinterpret_cast<char*>(&neg_r), 1);
        outfile.write(reinterpret_cast<char*>(&neg_g), 1);
        outfile.write(reinterpret_cast<char*>(&neg_b), 1);
    }
}
```

Aqui, cada componente é subtraída do valor máximo definido no cabeçalho (`max_color_val`), obtendo-se o negativo correspondente.

Já para o formato P3 (texto), os valores de cada componente são lidos como inteiros e processados da mesma forma:

```
else {
    for (int i = 0; i < pixel_count; ++i) {
        int r, g, b;
        infile >> r >> g >> b;

        int neg_r = max_color_val - r;
        int neg_g = max_color_val - g;
        int neg_b = max_color_val - b;

        unsigned char nr = neg_r, ng = neg_g, nb = neg_b;
```

```

        outfile.write(reinterpret_cast<char*>(&nr), 1);
        outfile.write(reinterpret_cast<char*>(&ng), 1);
        outfile.write(reinterpret_cast<char*>(&nb), 1);
    }
}

```

Apesar de o formato de entrada ser texto, a imagem negativa é igualmente guardada em formato binário, garantindo compatibilidade e menor tamanho do ficheiro.

### 2.2.3 Resultados

Na Figura 5 apresenta-se o resultado da execução do programa sobre a imagem `monarch.ppm`, que retrata uma borboleta amarela pousada em flores de tons rosados.

Após a aplicação do programa, observa-se que as cores foram corretamente invertidas, originando a sua versão negativa. A borboleta, originalmente amarela, passou a apresentar tons azulados, enquanto as flores, que eram rosadas, tornaram-se predominantemente verdes. O mesmo pode ser observado do fundo da imagem que originalmente apresentava cores quentes e na sua versão negativa apresentam tons frios. Esta alteração é consistente com o comportamento que é esperado da inversão de cores.



Imagen original



Imagen negativa

Figure 5: Comparaçāo entre a imagem original e a sua versāo negativa.

Deste modo, o resultado confirma o correto funcionamento do programa, evidenciando que a transformação da foto na sua versão do negativo foi efetuada de forma precisa e que o ficheiro gerado mantém a integridade estrutural e visual da imagem original.

## 2.3 Exercício 2b - Criar uma versão espelhada de uma imagem: (a) horizontalmente; (b) verticalmente

### 2.3.1 Contexto

O objetivo deste exercício é gerar uma versão espelhada de uma imagem. A operação de espelhamento consiste em inverter a disposição dos píxeis da imagem, podendo ser efetuada de duas formas:

- **Espelhamento horizontal (-h)**: a imagem é refletida ao longo do seu eixo vertical, trocando o lado esquerdo com o direito;
- **Espelhamento vertical (-v)**: a imagem é refletida ao longo do seu eixo horizontal, invertendo o topo e a base.

Em ambos os casos, o objetivo é preservar a estrutura visual da imagem, alterando apenas a disposição dos píxeis.

### 2.3.2 Implementação

O programa recebe três argumentos: o modo de espelhamento (**-h** ou **-v**), o ficheiro de entrada e o ficheiro de saída.

```
if (argc != 4) {
    std::cerr << "Usage: " << argv[0]
        << " <-h|-v> <input_image.ppm> <output_image.ppm>" << std::endl;
    return 1;
}

std::string mode = argv[1];
std::string input_filename = argv[2];
std::string output_filename = argv[3];
```

Nesta secção, o programa verifica se os argumentos foram corretamente fornecidos e identifica o tipo de espelhamento a realizar:

```
bool horizontal = false;
bool vertical = false;

if (mode == "-h") {
    horizontal = true;
} else if (mode == "-v") {
    vertical = true;
} else {
    std::cerr << "ERROR: Invalid mode. Use -h or -v." << std::endl;
    return 1;
}
```

Em seguida, o programa abre o ficheiro de entrada e lê o cabeçalho do formato PPM, obtendo as dimensões e o valor máximo de cor.

```

std::ifstream infile(input_filename, std::ios::binary);
std::string magic_number;
int width, height, max_color_val;

infile >> magic_number;
if (magic_number != "P3" && magic_number != "P6") {
    std::cerr << "ERROR: Invalid PPM format." << std::endl;
    return 1;
}

bool isBinary = (magic_number == "P6");
infile >> width >> height >> max_color_val;

```

Após a leitura do cabeçalho, os valores dos píxeis são carregados para uma estrutura bidimensional de `Pixel`, que contém as componentes RGB de cada píxel:

```

struct Pixel { int r, g, b; };
std::vector<std::vector<Pixel>> original_image(height, std::vector<Pixel>(width));

for (int y = 0; y < height; ++y) {
    for (int x = 0; x < width; ++x) {
        infile >> original_image[y][x].r
            >> original_image[y][x].g
            >> original_image[y][x].b;
    }
}

```

Depois de carregada a imagem, o espelhamento é realizado dependendo do modo selecionado. Para o espelhamento **horizontal**, os píxeis de cada linha são invertidos:

```

if (horizontal) {
    for (int y = 0; y < height; ++y) {
        for (int x = 0; x < width; ++x) {
            mirrored_image[y][x] = original_image[y][width - 1 - x];
        }
    }
}

```

Para o espelhamento **vertical**, a inversão é feita nas linhas da imagem:

```

else if (vertical) {
    for (int y = 0; y < height; ++y) {
        for (int x = 0; x < width; ++x) {
            mirrored_image[y][x] = original_image[height - 1 - y][x];
        }
    }
}

```

Finalmente, a função `write_ppm()` escreve o resultado num novo ficheiro PPM, preservando o cabeçalho e os valores de cor.

```

bool write_ppm(const std::string& filename,
               const std::vector<std::vector<Pixel>>& image,
               int max_val) {
    std::ofstream outfile(filename);
    outfile << "P3\n";
    outfile << "# Created by C++ mirror program\n";
    outfile << image[0].size() << " " << image.size() << "\n";
    outfile << max_val << "\n";
    ...
}

```

### 2.3.3 Resultados

Na Figura 6, é apresentada a comparação entre a imagem original, a sua versão espelhada horizontalmente e a versão espelhada verticalmente.

O espelhamento horizontal inverte os elementos da esquerda para a direita, mantendo o topo e a base inalterados, enquanto o espelhamento vertical inverte a imagem de cima para baixo. Os resultados confirmam o correto funcionamento de ambas as operações, preservando a estrutura e as cores originais da imagem.



Figure 6: Comparação entre a imagem original e as versões espelhadas.

Deste modo, verifica-se que o programa executa corretamente ambos os modos de espelhamento, mantendo a qualidade e o formato da imagem de saída.

## 2.4 Exercício 2c - Rodar uma imagem por um múltiplo de 90º

### 2.4.1 Contexto

O objetivo deste exercício é desenvolver um programa capaz de rodar uma imagem por um ângulo múltiplo de 90º, mais precisamente, 90º, 180º ou 270º. A rotação é implementada através da reorganização dos píxeis da imagem original, sem realizar qualquer interpolação, dado que os ângulos são múltiplos exatos de 90º. Com isto, a imagem é reconstruída numa nova matriz de píxeis reorganizados de acordo com o ângulo especificado, preservando as cores originais e a integridade visual.

## 2.4.2 Implementação

O programa é executado através da linha de comandos e recebe três argumentos: o ficheiro de entrada, o ficheiro de saída e o ângulo de rotação.

```
if (argc != 4) {
    std::cerr << "Usage: " << argv[0]
        << " <input_image.ppm> <output_image.ppm> <angle>" << std::endl;
    std::cerr << "Angle must be: 90, 180, or 270" << std::endl;
    return 1;
}

std::string input_filename = argv[1];
std::string output_filename = argv[2];
int angle = std::stoi(argv[3]);
```

O programa verifica se o ângulo fornecido é válido (90, 180 ou 270 graus), rejeitando outros valores.

```
if (angle != 90 && angle != 180 && angle != 270) {
    std::cerr << "ERROR: Invalid angle. Please enter 90, 180, or 270." << std::endl;
    return 1;
}
```

De seguida, o programa lê o cabeçalho da imagem PPM, identificando o formato (P3 ou P6), as dimensões e o valor máximo de cor.

```
std::string magic_number;
int orig_width, orig_height, max_color_val;

infile >> magic_number;
if (magic_number != "P3" && magic_number != "P6") {
    std::cerr << "ERROR: Input is not a valid PPM file." << std::endl;
    return 1;
}

bool isBinary = (magic_number == "P6");
infile >> orig_width >> orig_height >> max_color_val;
```

A imagem é então carregada para uma estrutura bidimensional de píxeis:

```
struct Pixel { int r, g, b; };
std::vector<std::vector<Pixel>> original_image(orig_height, std::vector<Pixel>(orig_w
```

```
for (int y = 0; y < orig_height; ++y) {
    for (int x = 0; x < orig_width; ++x) {
        infile >> original_image[y][x].r
            >> original_image[y][x].g
            >> original_image[y][x].b;
    }
}
```

Após o carregamento, é criada uma nova matriz para armazenar a imagem rotacionada. As dimensões da nova imagem variam consoante o ângulo: - para 90º ou 270º, a largura e a altura trocam de valor; - para 180º, permanecem iguais.

```
int new_width  = (angle == 180) ? orig_width : orig_height;
int new_height = (angle == 180) ? orig_height : orig_width;
```

```
std::vector<std::vector<Pixel>> rotated_image(new_height, std::vector<Pixel>(new_width));
```

A rotação é então aplicada conforme o ângulo indicado:

```
for (int y_new = 0; y_new < new_height; ++y_new) {
    for (int x_new = 0; x_new < new_width; ++x_new) {
        switch (angle) {
            case 90:
                rotated_image[y_new][x_new] =
                    original_image[orig_height - 1 - x_new][y_new];
                break;
            case 180:
                rotated_image[y_new][x_new] =
                    original_image[orig_height - 1 - y_new][orig_width - 1 - x_new];
                break;
            case 270:
                rotated_image[y_new][x_new] =
                    original_image[x_new][orig_width - 1 - y_new];
                break;
        }
    }
}
```

Por fim, a imagem resultante é escrita num novo ficheiro PPM através da função auxiliar `write_ppm()`, que preserva o formato de saída e o valor máximo de cor.

```
bool write_ppm(const std::string& filename,
               const std::vector<std::vector<Pixel>>& image,
               int max_val) {
    std::ofstream outfile(filename);
    outfile << "P3\n";
    outfile << "# Created by C++ rotate program\n";
    outfile << image[0].size() << " " << image.size() << "\n";
    outfile << max_val << "\n";
    ...
}
```

### 2.4.3 Resultados

Na Figura 7 apresenta-se a imagem original e as suas versões rotacionadas por  $90^\circ$ ,  $180^\circ$  e  $270^\circ$ .

A rotação de  $90^\circ$  faz com que os elementos da parte superior sejam deslocados para a direita, enquanto a de  $270^\circ$  os move para a esquerda. A rotação de  $180^\circ$  resulta numa inversão completa, mantendo a imagem na mesma orientação geral, mas invertendo todas as posições de píxeis.

Em todas as transformações, a integridade das cores e a estrutura visual são preservadas, confirmando o correto funcionamento do algoritmo de rotação.

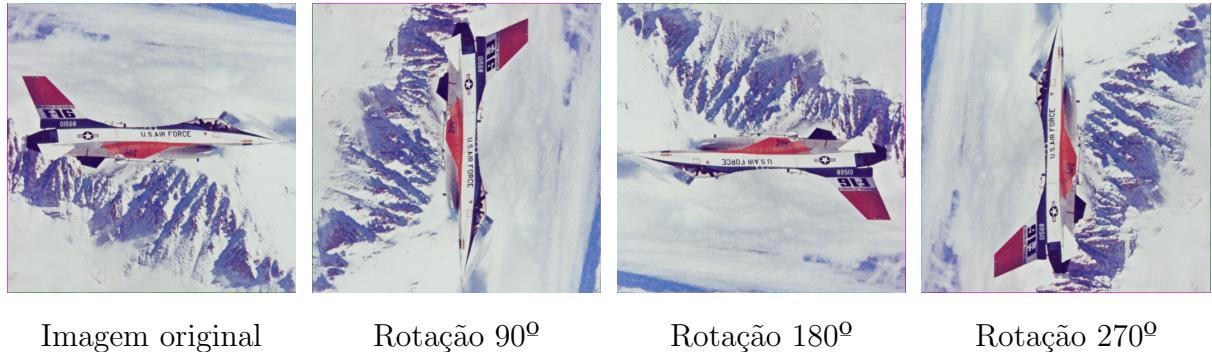


Imagem original                    Rotação  $90^\circ$                     Rotação  $180^\circ$                     Rotação  $270^\circ$

Figure 7: Comparação entre a imagem original e as versões com efeito de rotação.

Os resultados demonstram que o programa executa corretamente a rotação em qualquer dos três ângulos suportados, mantendo a fidelidade e estrutura da imagem original.

## 2.5 Exercício 2d - Aumentar (mais luz) / diminuir (menos luz) a intensidade de uma imagem

### 2.5.1 Contexto

Neste exercício, o objetivo é criar um programa capaz de alterar o nível de luminosidade de uma imagem, tornando-a mais clara ou mais escura conforme um valor de ajuste definido pelo utilizador.

A ideia principal consiste em adicionar (ou subtrair) um valor inteiro a cada componente de cor (R, G, B) de todos os píxeis da imagem. Um valor positivo aumenta a luminosidade (imagem mais clara), enquanto um valor negativo reduz a luminosidade (imagem mais escura).

### 2.5.2 Implementação

O programa recebe três argumentos:

- `input_image.ppm` — imagem de entrada;
- `output_image.ppm` — imagem de saída;
- `adjustment` — valor de correção de luminosidade (positivo ou negativo).

```

if (argc != 4) {
    std::cerr << "Usage: " << argv[0]
        << " <input_image.ppm> <output_image.ppm> <adjustment>" << std::endl;
    std::cerr << "Adjustment: positive for brighter, negative for darker" << std::endl;
    return 1;
}

int adjustment = std::stoi(argv[3]);

```

De seguida, o programa abre o ficheiro PPM, lê o cabeçalho (tipo de ficheiro, dimensões e valor máximo de cor), e verifica se é do tipo P3 (texto) ou P6 (binário):

```

std::string magic_number;
int width, height, max_color_val;

infile >> magic_number;
if (magic_number != "P3" && magic_number != "P6") {
    std::cerr << "ERROR: Input is not a valid PPM file." << std::endl;
    return 1;
}
bool isBinary = (magic_number == "P6");
infile >> width >> height >> max_color_val;

```

Após a leitura dos dados da imagem, cada píxel é ajustado individualmente. Para evitar saturação (valores fora do intervalo válido), é usada a função auxiliar `clamp()`:

```

int clamp(int value, int max_val) {
    return std::max(0, std::min(value, max_val));
}

```

A função `clamp()` assegura que nenhum canal de cor excede o valor máximo permitido nem fica abaixo de 0. A seguir, o programa aplica o ajuste a cada componente RGB:

```

Pixel& p_out = adjusted_image[y][x];
p_out.r = clamp(p_in.r + adjustment, max_color_val);
p_out.g = clamp(p_in.g + adjustment, max_color_val);
p_out.b = clamp(p_in.b + adjustment, max_color_val);

```

Depois de processar todos os píxeis, a imagem resultante é escrita num novo ficheiro PPM com a função auxiliar `write_ppm()`:

```

bool write_ppm(const std::string& filename,
               const std::vector<std::vector<Pixel>>& image,
               int max_val) {
    std::ofstream outfile(filename);
    outfile << "P3\n";
    outfile << "# Created by C++ brightness program\n";
    outfile << image[0].size() << " " << image.size() << "\n";
    outfile << max_val << "\n";
    ...
}

```

### 2.5.3 Resultados

Na Figura 8 apresentam-se os resultados do programa. A imagem à esquerda mostra a versão original, enquanto as duas seguintes representam as versões com brilho aumentado e reduzido, respectivamente.

Observa-se que o aumento de intensidade (+50) torna a imagem mais clara e viva, enquanto a diminuição de intensidade (-50) escurece a imagem, reduzindo a visibilidade dos detalhes. Em ambos os casos, os valores de cor foram corretamente limitados, evitando saturação excessiva ou subexposição total.



Figure 8: Comparação entre a imagem original e as versões com brilho alterado.

Os resultados comprovam que o algoritmo de ajuste de luminosidade funciona corretamente, permitindo clarear ou escurecer uma imagem de forma controlada, preservando a integridade visual e os limites de cor da mesma.

## 3 Parte 2

### 3.1 Exercício 3 - Golomb coding

#### 3.1.1 Contexto

O objetivo deste exercício é implementar uma classe capaz de realizar a codificação e descodificação de inteiros utilizando o método de *Golomb coding*. O algoritmo de Golomb é uma técnica de compressão sem perdas eficiente para valores inteiros, particularmente adequada para distribuições geométricas, onde números menores são mais prováveis que números grandes.

A codificação baseia-se num parâmetro  $m$ , que controla o equilíbrio entre a parte unária e binária do código. Cada número  $n$  é dividido em:

$$n = q \cdot m + r$$

Além disso, a classe deve lidar com números negativos segundo dois modos de operação:

- **Sign and Magnitude:** o primeiro bit indica o sinal (0 para positivo, 1 para negativo), e os restantes representam o valor absoluto;

- **Interleaved:** os valores positivos e negativos são intercalados ( $0, -1, 1, -2, 2, \dots$ ), garantindo uma sequência contínua de inteiros não negativos.

### 3.1.2 Implementação

A classe `GolombCoding` foi implementada com quatro componentes principais:

1. Cálculo de parâmetros dependentes de  $m$ ;
2. Mapeamento entre inteiros signed e unsigned;
3. Codificação (`encode`);
4. Descodificação (`decode`).

**Inicialização e parâmetros** O construtor recebe o parâmetro  $m$  e o modo de tratamento de números negativos. São calculados automaticamente os valores auxiliares  $b = \lfloor \log_2(m) \rfloor$  e o  $cutoff = 2^{b+1} - m$ :

```
void calculateParameters() {
    if (m == 0) throw std::invalid_argument("Golomb parameter m must be positive");
    b = static_cast<unsigned int>(std::floor(std::log2(m)));
    cutoff = (1 << (b + 1)) - m;
}
```

**Mapeamento de inteiros negativos** Para permitir a codificação de valores negativos, é necessário convertê-los para inteiros unsigned. O código suporta dois modos distintos:

```
unsigned int mapToUnsigned(int value) const {
    if (mode == SIGN_MAGNITUDE) return abs(value);
    return (value >= 0) ? 2 * value : 2 * (-value) - 1;
}
```

O modo `SIGN_MAGNITUDE` adiciona um bit de sinal, enquanto o modo `INTERLEAVED` mistura valores positivos e negativos numa única sequência crescente de inteiros.

De forma inversa, o método `mapToSigned()` reconverte o valor unsigned para o seu equivalente com sinal:

```
int mapToSigned(unsigned int value, bool isNegative = false) const {
    if (mode == SIGN_MAGNITUDE)
        return isNegative ? -static_cast<int>(value) : static_cast<int>(value);
    else
        return (value % 2 == 0) ? value / 2 : -(static_cast<int>(value + 1) / 2);
}
```

**Codificação** A função `encodeUnsigned()` é responsável por converter um valor unsigned num vetor de bits, através da combinação de codificação unária e binária truncada:

```
std::vector<bool> encodeUnsigned(unsigned int n) const {
    std::vector<bool> bits;
    unsigned int q = n / m;
    unsigned int r = n % m;

    for (unsigned int i = 0; i < q; i++) bits.push_back(false);
    bits.push_back(true);

    if (r < cutoff) {
        for (int i = b - 1; i >= 0; i--)
            bits.push_back((r >> i) & 1);
    } else {
        unsigned int adjusted = r + cutoff;
        for (int i = b; i >= 0; i--)
            bits.push_back((adjusted >> i) & 1);
    }

    return bits;
}
```

O método principal `encode()` combina esse processo com o mapeamento de sinal:

```
std::vector<bool> encode(int value) const {
    if (mode == SIGN_MAGNITUDE) {
        std::vector<bool> bits;
        bits.push_back(value < 0);
        unsigned int absValue = mapToUnsigned(value);
        auto encoded = encodeUnsigned(absValue);
        bits.insert(bits.end(), encoded.begin(), encoded.end());
        return bits;
    }
    return encodeUnsigned(mapToUnsigned(value));
}
```

**Descodificação** A operação inversa é implementada em `decodeUnsigned()` e `decode()`, que interpretam a sequência de bits e reconstruem o valor original:

```
std::pair<unsigned int, size_t> decodeUnsigned(const std::vector<bool>& bits, size_t start)
{
    unsigned int q = 0; size_t pos = start;
    while (pos < bits.size() && !bits[pos]) { q++; pos++; }
    pos++;
    unsigned int r = 0;
    for (unsigned int i = 0; i < b; i++) r = (r << 1) | bits[pos++];
    if (r >= cutoff) { r = (r << 1) | bits[pos++]; r -= cutoff; }
    unsigned int n = q * m + r;
    return {n, pos - start};
}
```

```

std::pair<int, size_t> decode(const std::vector<bool>& bits, size_t start=0) const{
    size_t bitsUsed = 0;
    int result;

    if (mode == SIGN_MAGNITUDE) {
        bool isNegative = bits[start];
        bitsUsed++;
        auto [magnitude, magBits] = decodeUnsigned(bits, start + 1);
        bitsUsed += magBits;
        result = mapToSigned(magnitude, isNegative);
    } else {
        auto [mapped, usedBits] = decodeUnsigned(bits, start);
        bitsUsed = usedBits;
        result = mapToSigned(mapped);
    }

    return {result, bitsUsed};
}

```

O método retorna o valor decodificado e o número de bits consumidos no processo.

## 4 Parte 3

### 4.1 Exercício 4 - Lossless audio codec based on Golomb coding

#### 4.1.1 Contexto

O objetivo deste exercício é implementar um codec de áudio *lossless* para ficheiros de áudio, cuja compressão é baseada na codificação de Golomb.

Para tal, foi desenvolvida uma solução de linha de comandos capaz de operar em dois modos distintos: codificação (para comprimir o áudio) e descodificação (para o reconstruir). A solução explora a elevada correlação temporal entre amostras de áudio, aplicando predição linear e codificando apenas o sinal residual. O codec suporta áudio mono e stereo, com otimizações para este último (Mid-Side), e permite a seleção de diferentes ordens de preditores.

#### 4.1.2 Implementação do Codificador

O objetivo do codificador não é codificar os valores das amostras diretamente, mas sim codificar o residual da predição. O fluxo da compressão é o seguinte:

**Inicialização e Leitura** O programa começa pela análise dos argumentos da linha de comandos. Em modo de codificação, são processados os parâmetros:

- **-p**: O tipo de preditor (Ordem 1, 2 ou 3).
- **-s**: O modo de stereo (Independent ou Mid-Side).
- **-n**: O modo de gestão de negativos (Interleaved ou Sign-Magnitude).

- **-m**: Opcionalmente, um valor fixo para o parâmetro Golomb.

Se o parâmetro **-m** for omitido, o codec opera em modo adaptativo. O ficheiro de áudio .wav de entrada é lido usando a biblioteca `libsndfile`, que extrai os metadados (sample rate, canais) e as amostras PCM.

**Escrever o Header do ficheiro de saída** Para permitir a descodificação, é escrito um cabeçalho personalizado no ficheiro de saída (.agol). Este cabeçalho armazena os metadados essenciais para a reconstrução do áudio, contendo, pela ordem:

1. **Magic Number**: A assinatura "AGOL" (4 bytes).
2. **Canais**: O número de canais (ex: 1 para mono, 2 para stereo).
3. **Sample Rate**: A taxa de amostragem (ex: 44100).
4. **Frames**: O número total de *frames* (amostras por canal).
5. **Predictor**: O ID do preditor usado.
6. **Modo Stereo**: O ID do modo stereo (0=Independent, 1=Mid-Side).
7. **Flag Adaptativo**: Um valor (1 ou 0) que indica ao descodificador se o parâmetro  $m$  é adaptativo ou fixo.
8.  **$m$  Fixo**: O valor de  $m$  (só é relevante se o flag anterior for 0).
9. **Modo de negativos**: O ID do modo de gestão de negativos utilizado.

**Gestão de Canais (Stereo)** Se o áudio for stereo (2 canais), o utilizador pode escolher o modo MID-SIDE (-s 1). Este modo explora a correlação entre os canais L (left) and R (right), transformando-os em Mid (M) e Side (S):

$$M = (L + R)/2$$

$$S = L - R$$

Esta transformação é *lossless* e, tipicamente, o canal 'Side' tem uma entropia muito mais baixa que os canais L/R originais, resultando em melhor compressão. O codificador passa então a codificar os canais M e S em vez de L e R. Se o modo for INDEPENDENT, L e R são codificados separadamente.

**Predição** O valor de cada amostra ( $x_i$ ) é estimado com base nos valores das amostras anteriores já processadas. O preditor específico é selecionado pelo utilizador:

- **ORDER\_1**:  $P = x_{i-1}$
- **ORDER\_2**:  $P = 2 \cdot x_{i-1} - x_{i-2}$
- **ORDER\_3**:  $P = 3 \cdot x_{i-1} - 3 \cdot x_{i-2} + x_{i-3}$

O preditor de ordem 2 é o *default*, pois oferece um bom equilíbrio entre complexidade e eficácia na predição de sinais de áudio.

**Calculo do residual** Após a predição, é calculada a diferença (residual) entre o valor real da amostra e o valor previsto. Este residual é o dado que será efetivamente codificado.

```
// Obtém o valor previsto
int16_t prediction = predict(samples, i, predictor);

// Calcula o residual
int residual = static_cast<int>(samples[i]) - static_cast<int>(prediction);
```

**Codificação da entropia** Os valores residuais são comprimidos usando a codificação de Golomb. Para se adaptar às características locais do sinal (ex: silêncio vs. ruído), o processo é feito em blocos de 1024 amostras.

1. **Parametro  $m$ :** No início de cada bloco, se o modo for adaptativo, o parâmetro  $m$  ideal é estimado para os resíduos desse bloco (usando `estimateGolombParameter`). O valor de  $m$  (16 bits) é então escrito no fluxo de bits, para que o descodificador se possa sincronizar.
2. **Gestão de Negativos:** Os resíduos negativos são mapeados para inteiros não-negativos usando os modos **INTERLEAVED** ou **SIGN-MAGNITUDE**.
3. **Escrita no fluxo de bits:** O objeto `GolombCoding` é instanciado com o  $m$  e o modo de negativos corretos, e cada residual do bloco é codificado e escrito bit-a-bit no ficheiro de saída.

Este processo repete-se para cada bloco de cada canal (ou dos canais M/S).

#### 4.1.3 Implementação do descodificador

O descodificador executa o processo inverso ao do codificador para reconstruir o áudio original sem qualquer perda.

**Leitura e validação do cabeçalho** O processo inicia-se com a abertura do ficheiro `.agol` e a leitura do seu cabeçalho.

- **Validação:** O programa lê os primeiros 4 bytes e verifica se correspondem à assinatura "AGOL".
- **Extração de Metadados:** São lidos todos os parâmetros guardados pelo codificador: sample rate, canais, frames, tipo de preditor, modo stereo, flag adaptativo e modo de negativos.

**Descodificação de Entropia** Com os metadados lidos, o `BitStream` é aberto em modo de leitura e posicionado imediatamente após o fim do cabeçalho. O descodificador itera então para ler o número exato de *frames* esperado para cada canal.

- **Sincronização de Bloco:** No início de cada bloco (a cada 1024 amostras), o descodificador lê o valor de  $m$  (16 bits) do fluxo. Isto assegura a sincronia com o codificador, usando o mesmo  $m$  para o mesmo bloco.
- **Descodificação Golomb:** Com o  $m$  correto e o modo de negativos lido do cabeçalho, o programa descodifica o residual, lendo os bits necessários do fluxo e revertendo o mapeamento de negativos.

## Predição e Reconstrução da Amostra

- **Predição:** O descodificador chama a \*mesma\* função `predict` (usando o tipo de preditor lido do cabeçalho). O preditor é aplicado sobre o vetor de amostras que está atualmente a ser reconstruído. Como a ordem de iteração é idêntica à do codificador, as amostras vizinhas necessárias ( $x_{i-1}$ ,  $x_{i-2}$ , etc.) já foram descodificadas nos passos anteriores deste loop. Isto garante que a predição é bit-a-bit idêntica à do codificador.
- **Reconstrução:** O valor final da amostra é calculado somando o residual ao valor previsto.

```
// Descodifica o residual do fluxo de bits
auto [residual, bitsUsed] = golomb.decode(bits, 0);

// Faz a mesma predição, mas usando o vetor de amostras em reconstrução
int16_t prediction = predict(samples, samples.size(), predictor);

// Reconstrói o valor original da amostra
int32_t sample = std::clamp(static_cast<int32_t>(prediction) + residual,
                           -32768, 32767);
samples.push_back(static_cast<int16_t>(sample));
```

**Reconstrução de Canais (Stereo)** Se o áudio for stereo e o modo MID-SIDE tiver sido usado, os dois canais descodificados são M e S. O descodificador aplica a transformação inversa *lossless* para recuperar L e R:

$$L = M + (S >> 1) + (S \& 1)$$

$$R = M - (S >> 1)$$

**Finalização e escrita** Após o loop preencher todos as amostras (e opcionalmente convertê-las de M/S para L/R), os canais são intercalados (ex: [L, R, L, R, ...]) no vetor de amostras final. Por fim, é usada a biblioteca `libsndfile` para salvar este vetor num ficheiro .wav de saída, com o sample rate e número de canais corretos lidos do cabeçalho.

### 4.1.4 Resultados

Para avaliar a eficácia do codec implementado, foram realizados testes de compressão sistemáticos no ficheiro `sample.wav`, um ficheiro de áudio stereo com as seguintes características:

- **Formato:** WAV PCM 16-bit
- **Canais:** 2 (stereo)
- **Sample Rate:** 44100 Hz
- **Frames:** 529200
- **Tamanho original:** 2116800 bytes (2.02 MB)

Foram testadas todas as combinações dos três preditores implementados (Order-1, Order-2, Order-3), dois modos de processamento stereo (Independent e Mid-Side) e dois modos de gestão de negativos (Interleaved e Sign-Magnitude), todos com parâmetro  $m$  adaptativo.

### Resultados com Modo Interleaved

Preditor	Modo Stereo	Tamanho (KB)	Bits/amostra	Compressão (%)
Order-1	Independent	1694.6	13.12	18.03
Order-1	Mid-Side	1669.1	12.92	19.26
Order-2	Independent	1612.3	12.48	22.01
Order-2	Mid-Side	<b>1593.9</b>	<b>12.34</b>	<b>22.89</b>
Order-3	Independent	1609.0	12.45	22.16
Order-3	Mid-Side	1599.5	12.38	22.63

Table 1: Resultados de compressão com modo de negativos Interleaved

### Resultados com Modo Sign-Magnitude

Preditor	Modo Stereo	Tamanho (KB)	Bits/amostra	Compressão (%)
Order-1	Mid-Side	1612.3	12.48	22.00
Order-2	Mid-Side	<b>1537.3</b>	<b>11.90</b>	<b>25.63</b>
Order-3	Mid-Side	1542.9	11.94	25.36

Table 2: Resultados de compressão com modo de negativos Sign-Magnitude (apenas Mid-Side testado)

**Análise dos Resultados** Os resultados obtidos permitem extrair várias conclusões importantes sobre a eficácia das diferentes estratégias de compressão:

#### 1. Eficácia dos Preditores:

- O preditor **Order-2** apresenta consistentemente os melhores resultados em todos os modos testados, alcançando a melhor compressão global de **25.63%** (1.345:1) com Mid-Side e Sign-Magnitude.
- O preditor Order-1, sendo o mais simples, apresenta a pior performance (18.03%–22.00%), demonstrando que a correlação temporal de ordem superior é significativa em sinais de áudio.
- O preditor Order-3, apesar de mais complexo, não oferece ganhos significativos sobre Order-2 (diferença de apenas 0.27% no melhor caso). Isto sugere que a complexidade adicional não compensa, possivelmente devido a overfitting em transientes abruptos do sinal.

#### 2. Impacto do Modo Stereo:

- O modo **Mid-Side** oferece consistentemente melhor compressão que o modo Independent em todos os preditores testados com Interleaved.
- No caso do Order-2 com Interleaved, o ganho do Mid-Side sobre Independent é de **0.88 pontos percentuais** (22.89% vs 22.01%).

- Este ganho deve-se à elevada correlação entre os canais esquerdo e direito em gravações stereo típicas. O canal "side" ( $L - R$ ) tem valores muito menores que os canais originais, resultando em resíduos mais pequenos e mais fáceis de comprimir.

### 3. Comparação dos Modos de Negativos:

- O modo **Sign-Magnitude** demonstra superioridade clara sobre Interleaved para sinais de áudio.
- Com Order-2 Mid-Side, Sign-Magnitude alcança 25.63% de compressão, enquanto Interleaved atinge apenas 22.89% — um ganho de **2.74 pontos percentuais**.
- Esta diferença é significativamente maior do que a observada em imagens (tipicamente < 1%). A razão prende-se com a distribuição dos resíduos: em áudio, os resíduos podem ter maior magnitude e distribuição mais ampla que em imagens. O modo Interleaved mapeia um residual negativo  $-k$  para  $2k - 1$ , duplicando efetivamente o valor. Para resíduos grandes, este "custo" torna-se proibitivo. O modo Sign-Magnitude, por outro lado, codifica apenas a magnitude mais 1 bit de sinal, sendo muito mais eficiente para valores maiores.

### 4. Parâmetro $m$ Adaptativo:

- A utilização de blocos de 1024 amostras com cálculo dinâmico de  $m$  permite ao codec ajustar-se às variações locais do sinal (ex: passagens de silêncio vs. passagens com maior energia).
- A fórmula de estimação  $m = \lceil -1 / \log_2(p) \rceil$  com  $p = \mu / (\mu + 1)$  assegura que o parâmetro Golomb é próximo do ótimo para a distribuição geométrica dos resíduos de cada bloco.

### 5. Verificação Lossless:

- Para validar a natureza lossless do codec, o ficheiro comprimido com a melhor configuração (Order-2, Mid-Side, Interleaved) foi descodificado e comparado bit-a-bit com o original.
- Os checksums MD5 são idênticos:

```
2c263e0baf279f6dec088f66a97e8080 sample.wav
2c263e0baf279f6dec088f66a97e8080 decoded_sample.wav
```

- Isto confirma que o processo de codificação e descodificação é perfeitamente reversível, sem qualquer perda de informação.

O codec de áudio implementado demonstra que a combinação de predição linear, transformação Mid-Side para stereo e codificação de Golomb adaptativa permite compressão lossless eficiente de sinais de áudio PCM.

Os resultados mostram taxas de compressão entre 18% e 25.6%, dependendo das características do sinal e dos parâmetros escolhidos, mantendo sempre a reconstrução perfeita do sinal original (verificado por checksum MD5 idêntico).

A configuração ótima identificada foi: **Order-2, Mid-Side, Sign-Magnitude,  $m$  adaptativo**, alcançando 25.63% de compressão (1.345:1) e 11.90 bits por amostra (redução de 25.6% face aos 16 bits originais).

## 5 Parte 4

### 5.1 Exercício 5 - Lossless image codec for grayscale images based on Golomb coding

#### 5.1.1 Contexto

O objetivo deste exercício é implementar um codec lossless para imagens em tons de cinza, cuja compressão é baseada na codificação de Golomb.

Para tal, foi desenvolvida uma solução de linha de comandos capaz de operar em dois modos distintos: codificação (para comprimir a imagem) e descodificação (para a reconstruir).

#### 5.1.2 Implementação do Codificador

O objetivo do codificador não é codificar os valores dos píxeis diretamente, mas sim codificar o residual da predição. O fluxo da compressão é o seguinte:

**Inicialização e Leitura** O programa começa pela análise dos argumentos da linha de comandos. Em modo de codificação, são processados os parâmetros de compressão: o tipo de preditor (-p), o modo de gestão de negativos (-s) e, opcionalmente, um valor fixo para o parâmetro Golomb (-m). Se o parâmetro -m for omitido, o codec calcula o melhor valor para o mesmo.

**Escrever o Header do ficheiro de saída** Para permitir a descodificação, é escrito um cabeçalho personalizado no ficheiro de saída (.gimg). Este cabeçalho armazena os metadados essenciais para a reconstrução da imagem, contendo, pela ordem:

1. **Magic Number:** A assinatura "GIMG", para identificação do ficheiro
2. **Largura:** A largura da imagem
3. **Altura:** A altura da imagem
4. **Preditor:** O ID do preditor usado
5. **Flag Adaptativo** Um valor (1 ou 0) que indica ao descodificador se o parâmetro m é adaptativo ou fixo
6. **m Fixo:** O valor de m (só é relevante se o flag anterior for 0)
7. **Modo de negativos:** O ID do modo de gestão de negativos utilizado

**Predição** O valor de cada pixel é estimado com base nos valores dos seus vizinhos já processados. O preditor específico é selecionado pelo utilizador. Exemplos de preditores implementados incluem:

1. **LEFT:** Utiliza o valor do pixel imediatamente à esquerda (a)
2. **TOP:** Utiliza o valor do pixel imediatamente a cima (b)
3. **TOP\_LEFT:** Utiliza o valor do pixel no canto superior esquerdo (c)

4. **AVG**: Calcula a média dos píxeis à esquerda e acima:  $(a+b)/2$ .
5. **PAETH**: Utiliza a formula  $a + b - c$
6. **A\_PLUS\_HALF\_B\_MINUS\_C**: Utiliza a formula  $a + (b - c) / 2$
7. **B\_PLUS\_HALF\_A\_MINUS\_C**: Utiliza a formula  $b + (a - c) / 2$

**Calculo do residual** Após a predição, é calculada a diferença entre o valor real do pixel e o valor previsto. Este residual é o dado que será efetivamente codificado, pois a sua distribuição de valores é muito mais compacta do que a dos píxeis originais.

```
// Extrai o valor real do pixel
uint8_t pixel = img.at<uint8_t>(row, col);

// Obtém o valor previsto
int prediction = predict(img, row, col, predictor);

// Calcula o residual
int residual = static_cast<int>(pixel) - prediction;
```

**Codificação da entropia** Por fim os valores residuais são comprimidos usando a codificação de Golomb. Para se adaptar às características locais da imagem, este processo é feito em blocos.

1. **Parametro m**: No início de cada bloco, se o modo for adaptativo, o parâmetro m ideal é calculado para esse bloco específico e o seu valor é escrito no fluxo de bits. Se o modo for fixo, o m definido no inicio é usado.
2. **Gestão de Negativos**: Como a Codificação Golomb só aceita inteiros não-negativos, os resíduos negativos (quando prediction < pixel) têm de ser mapeados usando os modos **INTERLEAVED** ou **SIGN-MAGNITUDE**.
3. **Escrita no fluxo de bits**: O objeto GolombCoding é instanciado com o m e o modo de negativos corretos, e cada residual é codificado e escrito bit-a-bit no ficheiro de saída.

```
GolombCoding golomb(m, negativeMode);

// Itera sobre os resíduos calculados para este bloco
for (int res : residuals) {
    // Converte 'res' num vetor de bits
    std::vector<bool> encoded = golomb.encode(res);
    // Escreve bit a bit no ficheiro de saída
    for (bool bit : encoded) {
        bs.write_bit(bit ? 1 : 0);
    }
}
```

### 5.1.3 Implementação do descodificador

O descodificador executa o processo inverso ao do codificador, agindo como um "espelho" para reconstruir a imagem original sem qualquer perda de dados.

**Leitura e validação do cabeçalho** O processo inicia-se com a abertura do ficheiro .gimg e a leitura do seu cabeçalho.

- **Validação:** O programa lê os primeiros 4 bytes e verifica se correspondem à assinatura "GIMG". Se não corresponderem, o ficheiro é considerado inválido.
- **Extração de Metadados:** São lidos todos os parâmetros de compressão que o codificador guardou: largura, altura, o tipo de preditor, o flag adaptativo e o modo de negativos.

**Descodificação de Entropia** Com os metadados do cabeçalho lidos, o descodificador prepara-se para a reconstrução. É alocada em memória uma matriz vazia, com as dimensões exatas da imagem.

O BitStream é aberto em modo de leitura e posicionado imediatamente após o fim do cabeçalho, apontando para o início dos dados comprimidos.

O descodificador itera então pela matriz (pixel por pixel, na mesma ordem do codificador), lendo os resíduos do fluxo de bits da seguinte forma:

- **Sincronização de Bloco:** No início de cada bloco, o descodificador lê o valor de  $m$  do fluxo de bits. Isto assegura a sincronia com o codificador, usando o mesmo  $m$  para aquele bloco.
- **Descodificação Golomb:** Com o  $m$  correto e o modo de negativos lido do cabeçalho, o programa chama `golomb.decode()`. Esta função lê os bits necessários do fluxo e reverte o mapeamento, devolvendo o residual original.

### Predição e Reconstrução do Pixel

- **Predição:** O descodificador chama a mesma função `predict` (usando o tipo de preditor lido do cabeçalho). O preditor é aplicado sobre a matriz que está atualmente a ser reconstruída. Como a ordem de iteração é idêntica à do codificador, os píxeis vizinhos necessários (à esquerda, acima, etc.) já foram descodificados e escritos na matriz nos passos anteriores deste loop. Isto garante que a predição do descodificador é bit-a-bit idêntica à predição feita pelo codificador.
- **Reconstrução:** O valor final do pixel é calculado somando o residual ao valor previsto. O resultado é limitado (clamp) ao intervalo [0, 255] para garantir que é um valor de pixel válido.

```
// Descodifica o residual do fluxo de bits
auto [residual, bitsUsed] = golomb.decode(bits, 0);

// Faz a mesma predição, mas usando a imagem em reconstrução
int prediction = predict(img, row, col, predictor);
```

```

// Reconstrói o valor original do pixel
int pixel = std::clamp(prediction + residual, 0, 255);

// Escreve o pixel final na matriz de saída
img.at<uint8_t>(row, col) = static_cast<uint8_t>(pixel);

```

**Finalização e escrita** Após o loop preencher todos os pixels, a matriz em memória é uma réplica exata da imagem original. Por fim é usada a função `cv::imwrite` para salvar esta matriz no ficheiro PGM de saída.

#### 5.1.4 Resultados

Para avaliar a eficácia do codec implementado, foram realizados testes de compressão em três imagens de referência com características distintas: House, Lena e Bike3.

Como o codec opera sobre imagens em tons de cinza, as imagens de origem foram primeiramente convertidas para este formato. Os tamanhos-base resultantes, sobre os quais os testes incidiram, foram, respetivamente: 65.54 KB (House), 262.14 KB (Lena) e 717.74 KB (Bike3).

Para cada uma destas imagens, foram testados todos os sete preditores implementados, em combinação com os dois modos de gestão de negativos (Interleaved e Sign-Magnitude) e  $m$  adaptativo.

house.ppm Interleaved		
ID do preditor	Tamanho do ficheiro de saída (KB)	Compressão (%)
1	39.64	39.51
2	42.68	34.87
3	45.77	30.15
4	39.27	40.08
5	36.15	44.84
6	36.85	43.77
7	38.36	41.46

Table 3: Resultados obtidos na compressão da imagem house.ppm usando o modo de gestão de negativos Interleaved

lena.ppm Interleaved		
ID do preditor	Tamanho do ficheiro de saída (KB)	Compressão (%)
1	176.36	32.72
2	161.09	38.55
3	184.15	29.75
4	160.49	38.78
5	159.00	39.34
6	163.93	37.47
7	156.46	40.32

Table 4: Resultados obtidos na compressão da imagem lena.ppm usando o modo de gestão de negativos Interleaved

bike3.ppm Interleaved		
ID do preditor	Tamanho do ficheiro de saída (KB)	Compressão (%)
1	504.71	29.68
2	503.43	29.86
3	545.65	23.98
4	483.65	32.61
5	482.59	32.76
6	489.58	31.79
7	488.12	31.99

Table 5: Resultados obtidos na compressão da imagem bike3.ppm usando o modo de gestão de negativos Interleaved

house.ppm Sign-Magnitude		
ID do preditor	Tamanho do ficheiro de saída (KB)	Compressão (%)
1	39.49	39.75
2	42.08	35.80
3	44.80	31.63
4	38.97	40.54
5	36.82	43.82
6	37.36	42.99
7	38.50	41.25

Table 6: Resultados obtidos na compressão da imagem house.ppm usando o modo de gestão de negativos Sign-Magnitude

lena.ppm Sign-Magnitude		
ID do preditor	Tamanho do ficheiro de saída (KB)	Compressão (%)
1	172.71	34.12
2	160.60	38.74
3	179.36	31.58
4	159.65	39.10
5	158.77	39.43
6	162.42	38.04
7	156.94	40.13

Table 7: Resultados obtidos na compressão da imagem lena.ppm usando o modo de gestão de negativos Sign-Magnitude

bike3.ppm Sign-Magnitude		
ID do preditor	Tamanho do ficheiro de saída (KB)	Compressão (%)
1	491.05	31.58
2	490.67	31.64
3	526.82	26.60
4	473.70	34.00
5	473.11	34.08
6	478.79	33.29
7	477.67	33.45

Table 8: Resultados obtidos na compressão da imagem bike3.ppm usando o modo de gestão de negativos Sign-Magnitude

Através da análise das tabelas, é possível observar que as imagens com maior tamanho original apresentam taxas de compressão inferiores.

Esta correlação não se deve diretamente ao tamanho do ficheiro em si, mas sim à complexidade da imagem.

- Imagens de Baixa Entropia: A imagem house.ppm é um bom exemplo. Possui grandes áreas de baixa complexidade, como o céu e as paredes lisas. Nestas regiões, os pixéis vizinhos são muito semelhantes, permitindo ao preditor fazer estimativas extremamente precisas. Isto resulta em resíduos muito pequenos, que são massivamente comprimidos pela codificação Golomb.
- Imagens de Alta Entropia: Em contrapartida, a imagem bike3.ppm é dominada por texturas complexas e com bastantes detalhes. A textura rugosa da parede e as variações abruptas causadas pela iluminação (luzes e sombras) representam uma baixa correlação espacial. Os preditores falham com mais frequência, gerando resíduos maiores, o que resulta numa sequência de dados com maior entropia, sendo mais difícil de comprimir.

Média dos preditores			
ID	Preditor	Média (Interleaved) (%)	Média (Sign-Magnitude) (%)
5	PAETH	38.98	39.11
7	B+(A-C)/2	37.92	38.28
6	A+(B-C)/2	37.68	38.11
4	AVG	37.16	37.88
2	TOP	34.43	35.39
1	LEFT	33.97	35.15
3	TOPLEFT	27.96	29.94

Table 9: Resultados obtidos na compressão da imagem bike3.ppm usando o modo de gestão de negativos Sign-Magnitude

Para determinar a eficácia global de cada preditor, foi calculada a média das taxas de compressão obtidas nas três imagens, para cada modo de negativo. Os resultados estão na tabela abaixo, ordenados do preditor mais eficaz (maior compressão média) para o menos eficaz.

Além disso, a análise dos dados mostra que a diferença de desempenho entre os modos Interleaved e Sign-Magnitude aumenta significativamente à medida que a eficácia do preditor diminui. Isto deve-se ao modo de como cada preditor funciona:

- **Modo Interleaved:** É otimizado para resíduos pequenos e centrados em zero (ex: -1, 0, 1). No entanto, quando um preditor fraco gera um resíduo grande (ex: 30 ou -30), este modo mapeia-o para um valor ainda maior (60 ou 59), o que é extremamente ineficiente para a codificação Golomb.
- **Modo Sign-Magnitude:** Este modo lida com resíduos grandes de forma mais robusta. Para codificar -30, ele simplesmente codifica a magnitude (30) e gasta apenas 1 bit extra para o sinal

Quando os resíduos são grandes, o ”imposto” de 1 bit do Sign-Magnitude é muito mais barato do que duplicar o valor do Interleaved. É por isso que a diferença entre ambos aumenta conforme pior for o preditor.