

Relatório do Trabalho Laboratorial nº 2

Informação e Codificação (2025/26)

Pedro Miguel Miranda de Melo (114208)

Rúben Cardeal Costa (114190)

Hugo Marques Dias (114142)

Departamento de Eletrónica, Telecomunicações e Informática (DETI)

Universidade de Aveiro

Novembro de 2025

Conteúdo

1	Introdução	3
2	Parte I: Manipulação Básica de Imagens com OpenCV	3
2.1	Programa <code>extract_channel</code>	3
2.1.1	Funcionalidade e Utilização	3
2.2	Programa <code>image_operations</code>	5
2.2.1	Negativo da Imagem (<code>image_negative</code>)	5
2.2.2	Espelhamento da Imagem (<code>image_mirror</code>)	5
2.2.3	Rotação da Imagem (<code>image_rotate</code>)	6
2.2.4	Ajuste de Intensidade (<code>image_intensity</code>)	6
3	Parte II: Classe de Codificação Golomb	8
3.1	Princípio de Codificação Golomb	8
3.2	Tratamento de Números Negativos	9
3.3	Interface da Classe Implementada	9
4	Parte III: Codec Áudio Lossless	10
4.1	Codificador <code>audio_encoder</code>	10
4.1.1	Algoritmo de Codificação	10
4.1.2	Determinação Adaptativa do Parâmetro m	10
4.1.3	Sintaxe e Exemplos	11
4.2	Descodificador <code>audio_decoder</code>	11
4.2.1	Garantia de Reconstrução Lossless	11
4.3	Análise de Desempenho e Compressão	11
5	Parte IV: Codec Imagem Lossless (Grayscale)	12
5.1	Arquitetura do Codec (<code>image_encoder</code> / <code>image_decoder</code>)	12
5.1.1	Algoritmo de Codificação	12
5.1.2	Algoritmo de Descodificação	13
5.2	Análise de Desempenho e Compressão	13
5.2.1	Análise dos Resultados	13
6	Conclusões	13

1 Introdução

Este relatório documenta o desenvolvimento e os resultados obtidos no âmbito do Trabalho Laboratorial nº 2 da unidade curricular de Informação e Codificação (2025/26), lecionada no Departamento de Eletrónica, Telecomunicações e Informática (DETI) da Universidade de Aveiro.

O projeto foca-se em duas áreas principais: a manipulação básica de imagens digitais utilizando a biblioteca OpenCV e a implementação de um sistema de codificação entrópica (Codificação Golomb) aplicado à compressão sem perdas (*lossless*) de sinais de áudio e imagem. O desenvolvimento foi realizado em C/C++, complementando as ferramentas desenvolvidas no trabalho anterior com novas funcionalidades.

O código-fonte completo do projeto está disponível publicamente no seguinte repositório GitHub: https://github.com/Rubenc1234/IC_miniP1/tree/main/Project2.

2 Parte I: Manipulação Básica de Imagens com OpenCV

A primeira parte do trabalho consistiu na familiarização com a biblioteca OpenCV através da implementação de programas para realizar operações fundamentais em imagens digitais, manipulando diretamente os seus píxeis. Foi instalada a versão 4.x da biblioteca, utilizando os pacotes pré-compilados disponíveis através do gestor de pacotes APT ('sudo apt install libopencv-dev pkg-config').

2.1 Programa extract_channel

Este programa tem como objetivo extrair um canal de cor específico (Azul, Verde ou Vermelho) de uma imagem de entrada, gerando uma imagem de saída em tons de cinza correspondente a esse canal.

2.1.1 Funcionalidade e Utilização

O programa lê uma imagem a cores (representada internamente em formato BGR pelo OpenCV). De seguida, cria uma nova imagem monocromática (CV_8UC1) com as mesmas dimensões. Percorrendo a imagem original pixel a pixel, o valor do canal especificado pelo utilizador (0 para Azul, 1 para Verde, 2 para Vermelho) é copiado para a posição correspondente na imagem de saída. A leitura e escrita dos pixels é feita usando o método `Mat::at<>()`.

A sintaxe de utilização é a seguinte:

```
1 ./bin/extract_channel <imagem_entrada> <imagem_saida> <numero_canal>
```

Listing 1: Sintaxe de Uso do extract_channel

Onde `numero_canal` deve ser 0, 1 ou 2. O formato da imagem de saída deve ser um que suporte imagens monocromáticas, como `.pgm` ou `.png`.

Exemplo de Teste: Para extraer os canais Azul (0), Verde (1) e Vermelho (2) da imagem `airplane.ppm`:

```
1 ./bin/extract_channel img/airplane.ppm imagens/airplane_extract_0.png 0
2 ./bin/extract_channel img/airplane.ppm imagens/airplane_extract_1.png 1
3 ./bin/extract_channel img/airplane.ppm imagens/airplane_extract_2.png 2
```

A Figura 5 mostra o resultado da extração dos três canais de cor.



Figura 1: *
Imagen Original (airplane)



Figura 2: *
Canal Azul (0)



Figura 3: *
Canal Verde (1)



Figura 4: *
Canal Vermelho (2)

Figura 5: Extração dos canais B, G, R da imagem original.

2.2 Programa image_operations

Este programa engloba um conjunto de operações geométricas e de intensidade sobre imagens, implementadas através da manipulação direta dos píxeis, sem recurso a funções específicas do OpenCV para essas transformações. Foram criados executáveis separados para cada operação para maior clareza.

2.2.1 Negativo da Imagem (image_negative)

Esta operação inverte os valores de intensidade de cada canal de cor. Para uma imagem de 8 bits por canal, o valor do píxel negativo P'_{cor} é calculado a partir do original P_{cor} como:

$$P'_{\text{cor}} = 255 - P_{\text{cor}}$$

A operação é aplicada independentemente a cada um dos canais B, G, R.

Utilização e Exemplo:

```
1 ./bin/image_negative <imagem_entrada> <imagem_saida> [view]
```

Listing 2: Sintaxe de Uso do image_negative

```
1 ./bin/image_negative img/airplane.ppm imagens/airplane_neg.png
```

O resultado é apresentado na Figura 8.



Figura 6: *
Imagen Original



Figura 7: *
Imagen Negativa

Figura 8: Resultado da operação de negativo.

2.2.2 Espelhamento da Imagem (image_mirror)

Esta funcionalidade permite espelhar a imagem horizontal ou verticalmente.

- **Espelhamento Horizontal (h):** O píxel (r, c) recebe o valor do píxel $(r, \text{largura} - 1 - c)$.
- **Espelhamento Vertical (v):** O píxel (r, c) recebe o valor do píxel $(\text{altura} - 1 - r, c)$.

Utilização e Exemplo:

```
1 ./bin/image_mirror <imagem_entrada> <imagem_saida> <h | v> [view]
```

Listing 3: Sintaxe de Uso do image_mirror

```
1 ./bin/image_mirror img/airplane.ppm imagens/airplane_mirror_h.png h
2 ./bin/image_mirror img/airplane.ppm imagens/airplane_mirror_v.png v
```

Os resultados são apresentados na Figura 12.



Figura 9: *
Original

Figura 10: *
Espelhada Horizontalmente

Figura 11: *
Espelhada Verticalmente

Figura 12: Resultados da operação de espelhamento.

2.2.3 Rotação da Imagem (image_rotate)

Implementa a rotação da imagem por qualquer ângulo múltiplo de 90 graus (positivo, negativo ou zero). O programa normaliza o ângulo fornecido para um equivalente em $\{0, 90, 180, 270\}$ graus no sentido horário e calcula a posição do píxel de origem. Para rotações de 90 ou 270 graus, as dimensões da imagem são trocadas.

Utilização e Exemplo:

```
1 ./bin/image_rotate <imagem_entrada> <imagem_saida> <angulo>
```

Listing 4: Sintaxe de Uso do image_rotate

```
1 ./bin/image_rotate img/airplane.ppm imagens/airplane_rotated90.png 90
2 ./bin/image_rotate img/airplane.ppm imagens/airplane_rotated180.png 180 %
3 ./bin/image_rotate img/airplane.ppm imagens/airplane_rotated270.png 270 %
```

A Figura 17 ilustra a rotação de 90 graus.

2.2.4 Ajuste de Intensidade (image_intensity)

Permite aumentar ou diminuir o brilho geral da imagem. O programa aceita um valor percentual no intervalo $[-100, 100]$. Este valor é mapeado para um ajuste aditivo A no intervalo $[-255, 255]$:

$$A = \text{round}(\text{percentagem} \times 2.55)$$

Este valor A é somado a cada canal de cor (B, G, R) de cada píxel. A função `saturate_cast<uchar>` garante que o resultado final permaneça no intervalo válido $[0, 255]$.



Figura 13: *
Imagen Original



Figura 14: *
Rotação 90° Horário

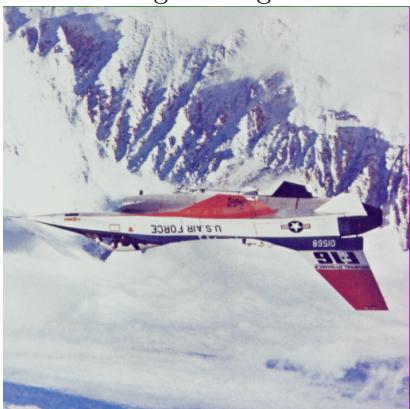


Figura 15: *
Rotação 180° Horário



Figura 16: *
Rotação 270° Horário

Figura 17: Resultado da operação de rotação.

Utilização e Exemplo:

```
1 ./bin/image_intensity <imagem_entrada> <imagem_saida> <percentagem_ajuste>
```

Listing 5: Sintaxe de Uso do image_intensity

```
1 ./bin/image_intensity img/airplane.ppm imagens/airplane_brighter_50.png 50
2 ./bin/image_intensity img/airplane.ppm imagens/airplane_darker_-50.png -50
```

A Figura 21 mostra os resultados para aumento e diminuição de 50%.



Figura 18: *
Original (0%)

Figura 19: *
Brilho +50%

Figura 20: *
Brilho -50%

Figura 21: Resultados da operação de ajuste de intensidade.

3 Parte II: Classe de Codificação Golomb

Esta parte focou-se na implementação de uma classe C++ para a codificação Golomb, conforme solicitado no enunciado. Esta é uma técnica de codificação entrópica eficiente para fontes com distribuições geométricas. A classe foi desenhada para ser a base dos codecs *lossless* das partes seguintes.

3.1 Princípio de Codificação Golomb

A codificação Golomb é uma família de códigos que depende de um parâmetro inteiro $m > 0$. A implementação segue a teoria descrita nos documentos de apoio:

1. Um inteiro não-negativo i é dividido em duas partes: um quociente q e um resto r .
2. As fórmulas para q e r são:

$$q = \left\lfloor \frac{i}{m} \right\rfloor \quad \text{e} \quad r = i - qm$$

3. O quociente q é codificado em código unário (uma sequência de q bits '1' seguida de um bit '0').
4. O resto r é codificado usando um código binário.

A forma como o resto r é codificado depende do valor de m :

Caso 1: m é uma potência de 2 (Golomb-Rice) Se $m = 2^b$, o resto r (que estará no intervalo $[0, m - 1]$) é simplesmente codificado usando a sua representação binária com $b = \log_2(m)$ bits.

Caso 2: m não é uma potência de 2 (Código Binário Truncado) Se m não é uma potência de 2, a codificação do resto é otimizada usando a regra do "truncated binary code":

1. Calcula-se $b = \lceil \log_2 m \rceil$.
2. Calcula-se o valor de "corte" $t = 2^b - m$.
3. Se $r < t$, o resto é codificado usando $b - 1$ bits.
4. Se $r \geq t$, o resto é codificado usando b bits, representando o valor $r + t$.

A nossa classe deteta automaticamente se m é uma potência de 2 e aplica a regra de codificação do resto apropriada (Golomb-Rice ou genérica).

3.2 Tratamento de Números Negativos

O enunciado pedia o suporte para números negativos usando duas estratégias distintas. Estas são selecionadas no construtor da classe.

- **1. Sinal e Magnitude (SIGN_MAGNITUDE):** Esta abordagem utiliza um bit de sinal extra. No nosso caso, '0' para positivo/zero e '1' para negativo. Este bit é escrito no início do *codeword*. O codificador Golomb é depois aplicado ao valor absoluto (magnitude) do número.
- **2. Intercalamento (INTERLEAVING):** Esta abordagem mapeia valores positivos e negativos para inteiros não-negativos antes da codificação. A regra de mapeamento utilizada foi:

$$i = \begin{cases} 2n & \text{se } n \geq 0 \\ 2|n| - 1 & \text{se } n < 0 \end{cases}$$

Isto gera a sequência $0 \rightarrow 0, -1 \rightarrow 1, 1 \rightarrow 2, -2 \rightarrow 3, \dots$. O descodificador simplesmente inverte este mapeamento após a descodificação Golomb.

3.3 Interface da Classe Implementada

Foi implementada uma classe Golomb com a seguinte interface pública:

```

1 // Enum para selecionar o modo de tratamento de negativos
2 enum class SignHandling {
3     SIGN_MAGNITUDE,
4     INTERLEAVING
5 };
6
7 class Golomb {
8 public:
9     // Construtor que recebe o parâmetro 'm' e o modo
10    Golomb(int m_param, SignHandling mode_param);
11
12    // Codifica um inteiro numa string de '0's e '1's
13    std::string encode(int n);
14
15    // Descodifica bits de uma string, atualizando o índice
16    int decode(const std::string& bits, size_t& index);
17
18 private:
19     // Métodos privados para o mapeamento de negativos e
20     // codificação/descodificação de inteiros não-negativos.
21     // ...
22 };

```

Listing 6: Interface C++ da classe Golomb (Golomb.h)

A classe foi testada com os valores de exemplo fornecidos nos documentos de apoio (como o Exemplo 5.7, para $m = 5$) e com os testes de exemplo desenvolvidos (incluindo $m = 4$ para Golomb-Rice), verificando-se a correção de ambas as estratégias de tratamento de negativos e da lógica de codificação genérica e Golomb-Rice.

4 Parte III: Codec Áudio Lossless

Nesta secção, foi desenvolvido um codec de áudio sem perdas (*lossless*), conforme solicitado na Parte III do enunciado. O objetivo é comprimir ficheiros de áudio (mono e estéreo) aplicando a codificação Golomb (implementada na Parte II) sobre os resíduos de um preditor.

Para maximizar a compressão, o codec implementa uma arquitetura de m adaptativo por bloco, similar à do codec de imagem.

4.1 Codificador audio_encoder

4.1.1 Algoritmo de Codificação

O codificador (`audio_encoder.cpp`) processa o áudio em blocos, calculando o m ótimo para cada um e escrevendo-o no *bitstream*.

1. **Leitura e Cabeçalho:** O ficheiro WAV é lido para a memória. O codificador escreve um cabeçalho no ficheiro de saída contendo os metadados (`samplerate`, `numChannels`, `numFrames`).
2. **Processamento por Blocos:** O áudio é processado em blocos (`blockSize = 4096` tramas). Para cada bloco:
 - **Passagem 1 (do Bloco):** É feita uma passagem interna no bloco para calcular todos os resíduos.
 - **Estéreo (MID/SIDE):** Se `numChannels == 2`, é aplicada a transformação *lossless* ($MID = \lfloor (L + R)/2 \rfloor$, $SIDE = L - R$).
 - **Predição (Ordem 1):** É aplicado um preditor temporal de 1^a ordem ($P[n] = x[n - 1]$) a cada canal (Mono, ou MID e SIDE). O estado do preditor (e.g., `mono_pred`) é mantido entre os blocos.
 - Os resíduos ($e = x - P$) são guardados em vetores temporários.
 - **Cálculo de m Ótimo:** A função `calculate_optimal_m` é chamada para os vetores de resíduos.
 - **Escrita de m :** Os m 's ótimos (16 bits cada) são escritos no *bitstream*.
 - **Passagem 2 (do Bloco):** Os resíduos são codificados com os m 's ótimos e escritos no *bitstream*.
3. **Empacotamento de Bits:** A *string* de bits total é empacotada em bytes e escrita no ficheiro de saída.

4.1.2 Determinação Adaptativa do Parâmetro m

A chave para a compressão é "sintonizar" o codificador Golomb com os dados. A função `calculate_optimal_m` é usada para encontrar o m ideal para cada bloco de resíduos.

Teoria vs. Prática A teoria define m em função de parâmetros estatísticos teóricos (α ou $p = P(x = 0)$). Na prática, não conhecemos estes parâmetros, mas podemos medi-los através da média dos nossos dados (os resíduos).

A nossa classe Golomb usa o mapeamento INTERLEAVING para converter os resíduos com sinal em inteiros não-negativos i . A teoria de compressão (usada em *standards* como o JPEG-LS) demonstra que o m ótimo pode ser estimado diretamente da média (μ) destes valores i mapeados, usando a aproximação:

$$m \approx \mu \times \ln(2)$$

Implementação A nossa função `calculate_optimal_m` implementa esta lógica:

1. Calcula a média ('mean') dos resíduos após o mapeamento INTERLEAVING.
2. Estima m usando a fórmula: ' $m = \text{round}(\text{mean} * \log(2))$ '.

Isto garante que cada bloco usa o codificador Golomb mais eficiente para os seus resíduos específicos.

4.1.3 Sintaxe e Exemplos

O codificador calcula m automaticamente, não sendo um argumento.

```
1 ./bin/audio_encoder <ficheiro_entrada.wav> <ficheiro_saida.bin>
```

Listing 7: Sintaxe de Uso do `audio_encoder`

```
1 ./bin/audio_encoder wav/sample.wav wav_out/sample_encoded.glob
```

Listing 8: Exemplo de Codificação de Áudio

4.2 Descodificador `audio_decoder`

O descodificador (`audio_decoder.cpp`) inverte o processo, lendo o m de cada bloco.

```
1 ./bin/audio_decoder <ficheiro_entrada.bin> <ficheiro_saida.wav>
```

Listing 9: Sintaxe de Uso do `audio_decoder`

```
1 ./bin/audio_decoder wav_out/sample_encoded.glob wav_out/sample_decoded.wav
```

Listing 10: Exemplo de Descodificação de Áudio

4.2.1 Garantia de Reconstrução Lossless

Duas decisões de implementação são cruciais para garantir a reconstrução 100% *lossless* do sinal:

- **Cálculos em 32-bit:** Todos os cálculos de predição e resíduos (incluindo MID/SIDE) são feitos com `int32_t`. Isto previne qualquer *integer overflow* que poderia ocorrer ao subtrair amostras de 16 bits (e.g., $30000 - (-30000)$), garantindo a integridade dos resíduos.
- **Transformação MID/SIDE Reversível:** A transformação estéreo é implementada de forma a ser 100% reversível com aritmética inteira. O codificador usa `side = L - R` e `mid = R + (side >> 1)`. O descodificador inverte-a com `R = mid - (side >> 1)` e `L = R + side`, garantindo que não há erros de arredondamento.

4.3 Análise de Desempenho e Compressão

O codec foi testado com dois ficheiros de áudio estéreo.

Tabela 1: Análise de Compressão do Codec Áudio (Preditor Ordem 1, M Adaptativo)

Ficheiro	Original	Comprimido	Taxa de Compressão
sample.wav	2.1 MB	1.6 MB	1.31 : 1
sample2.wav	4.8 MB	3.1 MB	1.55 : 1

Tabela 2: Tempos de Execução do Codec Áudio (ms)

Ficheiro	Codificação (ms)	Descodificação (ms)
sample.wav	105.80 ms	57.91 ms
sample2.wav	216.66 ms	113.14 ms

Análise dos Resultados Os testes de verificação com `wav_cmp` (demonstrados nos testes executados) confirmaram que o codec é 100% lossless, com ‘MSE = 0.0’ e ‘SNR = inf’. A compressão alcançada (1.3:1 a 1.5:1) é modesta. Isto deve-se inteiramente à simplicidade do preditor de 1ª ordem ($P[n] = x[n - 1]$). Este preditor é fraco para sinais de áudio complexos, gerando resíduos médios elevados. Consequentemente, o m ótimo calculado por bloco é alto, e a codificação Golomb não consegue ser muito eficiente.

5 Parte IV: Codec Imagem Lossless (Grayscale)

Aplicando a mesma arquitetura de m adaptativo por bloco, foi desenvolvido um codec *lossless* para imagens em escala de cinza, conforme a Parte IV do enunciado.

5.1 Arquitetura do Codec (`image_encoder` / `image_decoder`)

5.1.1 Algoritmo de Codificação

O codificador (`image_encoder.cpp`) implementa uma lógica adaptativa por blocos de 16x16.

1. **Leitura e Conversão:** A imagem de entrada (PPM) é lida e convertida para escala de cinza usando a função `readPPMtoGray`.
2. **Escrita do Cabeçalho:** Um "magic number" ("GOL1") é escrito, seguido das dimensões da imagem e `maxval`.
3. **Processamento por Blocos (16x16):** Para cada bloco:

- **Passagem 1 (Análise):** O codificador calcula os resíduos de predição para o bloco. É usado o preditor **MED (Median Edge-Detection)**, que estima o píxel atual $P(x, y)$ com base nos vizinhos Oeste (a), Norte (b) e Noroeste (c):

$$P(x, y) = \begin{cases} \min(a, b) & \text{se } c \geq \max(a, b) \\ \max(a, b) & \text{se } c \leq \min(a, b) \\ a + b - c & \text{caso contrário} \end{cases}$$

- **Cálculo de m :** O m ótimo para os resíduos do bloco é calculado com `calculate_optimal_m`.
- **Escrita de m :** O m ótimo (16 bits) é escrito no *bitstream* do bloco.
- **Passagem 2 (Codificação):** Os resíduos do bloco são codificados com `Golomb(m_otimo, INTERLEAVING)` e escritos no *bitstream*.

5.1.2 Algoritmo de Descodificação

O descodificador (`image_decoder.cpp`) inverte o processo.

1. **Processamento por Blocos:** Itera pela imagem na mesma ordem de blocos.
2. **Leitura de m :** No início de cada bloco, lê 16 bits do *bitstream* para obter o m específico desse bloco.
3. **Reconstrução:** Itera pelos píxeis do bloco. Calcula o valor de predição $P(x, y)$ usando o mesmo preditor MED sobre os píxeis **já reconstruídos**.
4. Descodifica o resíduo e e reconstrói o píxel $p = P(x, y) + e$.

5.2 Análise de Desempenho e Compressão

A eficácia do codec foi testada com três imagens de teste.

Tabela 3: Análise de Compressão do Codec de Imagem

Imagen	Original (PPM)	Comprimido (GOL)	Taxa de Compressão
airplane.ppm	769 kB	130 kB	5.92 : 1
arial.ppm	1.1 MB	262 kB	4.20 : 1
bike3.ppm	2.1 MB	423 kB	4.96 : 1

Tabela 4: Tempos de Execução do Codec de Imagem

Imagen	Codificação (s)	Descodificação (ms)
airplane.ppm	1.05 s	24.12 ms
arial.ppm	4.73 s	34.86 ms
bike3.ppm	11.42 s	60.16 ms

5.2.1 Análise dos Resultados

Os resultados demonstram uma eficácia excepcional do codec de imagem, com taxas de compressão a rondar 5:1 ou 6:1.

- **Eficácia do Predictor MED:** O sucesso deve-se principalmente ao preditor não-linear MED. Ao contrário do preditor linear simples do áudio, o MED adapta-se às arestas da imagem, gerando resíduos extremamente pequenos (muitas vezes zero) em áreas uniformes.
- **Assimetria de Tempo:** Observa-se uma grande assimetria nos tempos de execução (Tabela 4). A descodificação é extremamente rápida (milissegundos) porque é uma operação direta. A codificação é mais lenta (segundos) devido à necessidade de processar cada bloco duas vezes: uma para calcular estatísticas e o m ótimo, e outra para codificar.

6 Conclusões

Este trabalho laboratorial permitiu uma exploração prática e aprofundada dos conceitos de processamento de imagem e de codificação entrópica, culminando no desenvolvimento de codecs *lossless* funcionais.

Na **Parte I**, adquirimos competências fundamentais na manipulação de imagens com a biblioteca OpenCV. A implementação de operações como rotação, espelhamento e ajuste de intensidade, manipulando diretamente os píxeis, serviu de base para a compreensão da estrutura de dados de uma imagem, que se revelou crucial para a Parte IV.

Na **Parte II**, foi implementada a classe **Golomb**, o motor de compressão central do projeto. A implementação de um m parametrizável e de estratégias de tratamento de negativos (*Interleaving*) foi um exercício fundamental de codificação entrópica.

Nas **Partes III e IV**, foram implementados codecs de áudio e imagem usando a mesma arquitetura de m adaptativo por bloco. A comparação dos seus resultados é a conclusão mais importante deste trabalho:

- O **Codec de Áudio** (Parte III) alcançou uma compressão modesta (Taxa 1.24:1).
- O **Codec de Imagem** (Parte IV) alcançou uma compressão excelente (Taxa 5.93:1).

Esta discrepância demonstra de forma clara que a eficácia de um codec *lossless* baseado em predição não depende tanto do codificador de entropia (Golomb), mas sim da qualidade do seu modelo de predição.

O nosso codec de imagem foi altamente eficaz porque usou um preditor 2D sofisticado (MED) que gerou resíduos minúsculos. O nosso codec de áudio teve um desempenho fraco porque usou um preditor 1D muito simples (Ordem 1) que foi incapaz de modelar a natureza complexa do sinal de áudio, gerando resíduos grandes e difíceis de comprimir.

Em suma, este projeto ilustrou de forma prática a diferença fundamental between a codificação de áudio e imagem, e provou que a compressão *lossless* é um exercício de modelação estatística: quanto melhor o modelo de predição, mais "pequenos" e "previsíveis" se tornam os resíduos, e mais eficaz é a codificação entrópica final.