UNIVERSIDADE DE AVEIRO

DEPARTAMENTO DE ELECTRÓNICA, TELECOMUNICAÇÕES E INFORMÁTICA

INFORMAÇÃO E CODIFICAÇÃO

PROJETO 2

Autores:
António Ferreira (89082)
Guilherme Claro (98432)
Luis Couto (89078)

https://github.com/Toja-Ferreira/IC_G07_Proj2

2022

Contents

Introdução

O presente relatório visa descrever a resolução do Projeto 2 desenvolvido no âmbito da unidade curricular de Informação e Codificação.
Para cada problema é apresentada uma explicação teórica que fundamenta a solução implementada, seguida de uma explicação prática e finalmente apresentação de resultados.

No repositório do projeto, para além de todo o material desenvolvido, existe um ficheiro README.md, com instruções de como compilar e testar todos os programas.

Parte I

Nesta primeira parte é utilizada a livraria OpenCV para manipulação variada de imagens.

Para compilar todos os programas da Parte I:

```
# Estando no diretório base do projeto
> cd Part_I/
> cmake .
> make
```

Exercício 1

Neste exercício foi criado um programa que copia uma imagem, pixel por pixel, de um ficheiro para o outro.
Para implementar esta funcionalidade, cada pixel da imagem original é copiado para a posição correspondente na nova imagem.

```
for(int y = 0; y < toCopy.rows; y++)
{
    for(int x = 0; x < toCopy.cols; x++)
    {
        for(int c = 0; c < toCopy.channels(); c++)
        {
            newImage.at<Vec3b>(y,x)[c] = toCopy.at<Vec3b>(y,x)[c];
        }
    }
}
```

Para testar o programa desenvolvido (no terminal):

```
# Estando no diretório IC_G07_Proj2/Part_I/test_programs
> ./copyImage lena.ppm lenacopy.ppm

# Se a imagem for copiada com sucesso receberá a mensagem:
A copy named lenacopy.ppm has been created!
```

Figure 1: Imagem Original

Figure 2: Imagem Copiada

Exercício 2

Para as alíneas deste exercício, a livraria OpenCV dispõe de métodos que tornariam as suas implementações muito mais simples e diretas, contudo foi tomada uma abordagem de aprendizagem e desenvolvidas soluções sem a utilização desses métodos.

Alínea a) - Versão Negativa de uma Imagem

Para implementar esta funcionalidade, cada canal de cada pixel da nova imagem toma o valor de [255 - valor do canal do pixel correspondente da imagem original].

```
for(int y = 0; y < toCopy.rows; y++)
{
    for(int x = 0; x < toCopy.cols; x++)
    {
        for(int c = 0; c < toCopy.channels(); c++)
        {
            newImage.at<Vec3b>(y,x)[c] = 255 - toCopy.at<Vec3b>(y,x)[c];
        }
    }
}
```

Para testar o programa desenvolvido (no terminal):

```
# Estando no diretório IC_G07_Proj2/Part_I/test_programs
> ./negativeImage lena.ppm lenanegative.ppm

# Se a imagem negativa for criada com sucesso receberá a mensagem:
A negative image named lenanegative.ppm has been created!
```

Figure 3: Imagem Original

Figure 4: Imagem Negativa

Alínea b) - Versão Espelhada de uma Imagem

Para implementar a funcionalidade de espelhar uma imagem verticalmente, é necessário inverter as linhas da matriz da imagem original, criando assim a matriz da nova imagem.

```
int k = toCopy.rows;
for(int y = 0; y < toCopy.rows; y++)
{
    for(int x = 0; x < toCopy.cols; x++)
    {
        newImage.at<Vec3b>(y,x) = toCopy.at<Vec3b>(k,x);
    }
    k--;
}
```

Para implementar a funcionalidade de espelhar uma imagem horizontalmente, é necessário inverter os pixeis de cada linha da matriz da imagem original, criando assim a matriz da nova imagem.

```
int k = toCopy.cols;
for(int y = 0; y < toCopy.rows; y++)
{
    for(int x = 0; x < toCopy.cols; x++)
    {
        newImage.at<Vec3b>(y,x) = toCopy.at<Vec3b>(y,k-x);
    }
}
```

Para testar o programa desenvolvido (no terminal):

```
# Estando no diretório IC_G07_Proj2/Part_I/test_programs
> ./mirrorImage lena.ppm lenavertical.ppm V        // espelhada verticalmente

# Se a imagem for espelhada com sucesso receberá a mensagem:
A mirrored image named lenavertical.ppm has been created!

# Estando no diretório IC_G07_Proj2/Part_I/test_programs
> ./mirrorImage lena.ppm lenahorizontal.ppm H      // espelhada horizontalmente

# Se a imagem for espelhada com sucesso receberá a mensagem:
A mirrored image named lenavertical.ppm has been created!
```

Figure 5: Imagem Original

Figure 6: Espelhada Verticalmente

Figure 7: Espelhada Horizontalmente

Alínea c) - Rodar uma Imagem em Múltiplos de 90º

Para implementar esta funcionalidade, após confirmar que o ângulo de rotação introduzido pelo utilizador é múltiplo de 90º, existem 3 passos fundamentais:

1. Calcular o ponto de rotação, que neste caso é dado pelo ponto central da matriz da imagem original

   ```
   Point2f rotationPoint(toCopy.cols/2., toCopy.rows/2.);
   ```

2. Criar a matriz de rotação, usando o ponto de rotação, o ângulo de rotação em graus (introduzido pelo utilizador) e o fator de escala (considerado como 1.0)

   ```
   Mat rotationMatrix = getRotationMatrix2D(rotationPoint, stoi(argv[3]), 1.0);
   ```

3. Aplicar uma transformação na matriz original usando a matriz de rotação, criando assim a matriz da nova imagem rodada

   ```
   warpAffine(toCopy, newImage, rotationMatrix, Size(toCopy.cols, toCopy.rows));
   ```

Desta maneira é então possível rodar uma imagem, sendo que um ângulo de rotação com valor positivo corresponde a uma rotação no sentido anti-relógio, e um valor negativo a uma rotação no sentido relógio.

Para testar o programa desenvolvido (no terminal):

```
# Estando no diretório IC_G07_Proj2/Part_I/test_programs
> ./rotateImage lena.ppm lenarotate90.ppm 90        // rotação de 90º

# Se a imagem for rodada com sucesso receberá a mensagem:
A rotated image named lenarotate90.ppm has been created!
```

Figure 8: Imagem Original

Figure 9: Rotação de -90º

Figure 10: Rotação de 90º

Figure 11: Rotação de -180º e 180º

Figure 12: Rotação de -270º

Figure 13: Rotação de 270º

Alínea d) - Manipular a Intensidade de uma Imagem

Para implementar esta funcionalidade, é necessário adicionar ao valor de cada canal de cada pixel da imagem original um valor inteiro constante, no intervalo [-255, 255], criando assim a matriz da nova imagem.

```
for(int y = 0; y < toCopy.rows; y++)
{
    for(int x = 0; x < toCopy.cols; x++)
    {
        for(int c = 0; c < toCopy.channels(); c++)
        {
            newImage.at<Vec3b>(y,x)[c] = saturate_cast<uchar>(toCopy.at<Vec3b>(y,x)[c]
            + constantVal);
        }
    }
}
```

Desta maneira é então possível manipular a luminosidade de uma imagem, sendo que uma constante inteira de valor positivo corresponde a aumentar a luminosidade, e uma de valor negativo a diminuir a luminosidade.

Para testar o programa desenvolvido (no terminal):

```
# Estando no diretório IC_G07_Proj2/Part_I/test_programs
> ./intensityImage lena.ppm lenaplus100.ppm 100 // intensidade da luz +100

# Se o valor da intesidade da imagem for alterado com sucesso receberá a mensagem:
An image with altered intensity named lenaplus100.ppm has been created!

# Estando no diretório IC_G07_Proj2/Part_I/test_programs
> ./intensityImage lena.ppm lenaminus100.ppm -100 // intensidade da luz -100

# Se o valor da intesidade da imagem for alterado com sucesso receberá a mensagem:
An image with altered intensity named lenaminus100.ppm has been created!
```

Figure 14: Imagem Original

Figure 15: Intensidade +100

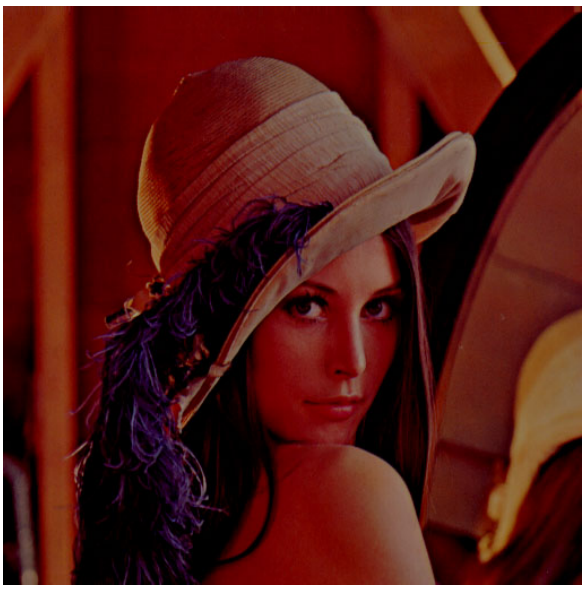

Figure 16: Intensidade -100

Parte II

Código Golomb - Codificador

O código Golomb transforma números inteiros, separando-os em duas partes, uma parte representada em código unário e outra parte representada em código binário.
Ou seja, um inteiro $i \geq 0$ é representado por dois números, q e r onde:

$$q = \left\lfloor \frac{i}{m} \right\rfloor$$

$$r = i - qm$$

O quociente, q, pode ter os valores 0, 1, 2, ..., e é representado pelo código unário correspondente.
O resto da divisão, r, pode ter os valores 0, 1, 2, ..., m-1, e é representado pelo código binário correspondente.
O parâmetro m, do qual o código depende, é sempre um inteiro positivo > 0 e pode ser obtido a partir de:

$$m = \left\lceil -\frac{1}{log(\alpha)} \right\rceil$$

Se m não for potência de 2, então o código binário (que representa o resto da divisão), não é eficiente. Neste caso, o número de bits usados pode ser reduzido usando um código binário truncado.
Toda esta explicação é traduzida no seguinte pseudocódigo:

Algorithm 1 Codificador de Golomb

```
1: if m é potência de 2 then
2:     bin ← r codificado em binário com b bits
3:     un ← q codificado em unário
4:     res ← un + bin
5:     return res
6: else
7:     un ← q codificado em unário
8:     if r < (2^b − m) then
9:         bin ← r codificado em binário com b − 1 bits
10:    else
11:        bin ← r + (2^b − m) codificado em binário com b bits
12:    end if
13:    res ← un + bin
14:    return res
15: end if
```

Apesar de ter sido desenhado para números inteiros positivos, é possível também estender este código para gerir números inteiros com sinal. Para tal, é necessário, antes de começar a codificação, transformar o inteiro com sinal num inteiro positivo único, através de uma função que seja posteriormente reversível.

Para este efeito, é utilizado um algoritmo, descrito seguidamente. Este destaca-se pelo facto de que:

* Inteiros positivos são sempre convertidos em inteiros positivos pares únicos
* Inteiros negativos são sempre convertidos em inteiros positivos ímpares únicos

Algorithm 2 Conversor para inteiros positivos únicos

if $n \geq 0$ **then**
    return $2n$
else
    return $-2n - 1$
end if

Sabendo os detalhes do algoritmo anterior, é bastante fácil implementar as operações matemáticas opostas ás operações realizadas. Para isto, é utilizado um algoritmo, descrito seguidamente, para reverter a transformação do inteiro positivo único e obter o inteiro com sinal original:

Algorithm 3 Conversor para inteiros com sinal

if n é par **then**
    return $n/2$
else
    return $(-1)\lceil n/2 \rceil - 1$
end if

Código Golomb - Descodificador

Para obter os valores originais, é necessário implementar um descodificador. Sabendo os detalhes de como o codificador do Código Golomb funciona, é possível implementar um algoritmo que inverta as operações realizadas. Importante também relembrar que é necessário obter o valor original do inteiro com sinal, pelo que o valor descodificado tem depois ser revertido usando o algoritmo 3 explicado anteriormente.
O descodificador é então descrito pelo seguinte pseudocódigo:

Algorithm 4 Descodificador de Golomb

$s \leftarrow$ número de 1s consecutivos do input (parar ao encontrar o primeiro 0)
$x \leftarrow$ próximos b -1 bits do input

if $x < (2^b - m)$ **then**
    $s \leftarrow s \times m + x$
    return reverter(s)
else
    $x \leftarrow x \times x \times 2$+próximo bit do input
    $s \leftarrow s \times m - x - (2^b - m)$
    return reverter(s)
end if

Exercício 3 - Golomb Codec

Para compilar todos os programas da Parte II:

```
# Estando no diretório base do projeto
> cd Part_II/
> cmake .
> make
```

Neste exercício foi então implementada uma classe para codificação e descodificação Golomb, partindo da teoria e algoritmos explicados acima. Foram criados 2 ficheiros, um **Golomb.hpp** com os cabeçalhos e descrições detalhadas e um **Golomb.cpp** com a sua implementação.
De seguida são apresentadas algumas das funções implementadas mais importantes:

```
// Codifica um inteiro com sinal n com codigo de Golomb,
// implementando o algoritmo 1
string encode(int n);

// Descodifica uma string de bits com codigo de Golomb,
// implementando o algoritmo 4
int decode(string stringOfBits);

// Converte um inteiro com sinal num inteiro positivo único, implementando
// o algoritmo 2
int transformInteger(int n)

// Converte um inteiro positivo único no inteiro com sinal original,
// implementando o algoritmo 3
int revertInteger(int n)
```

Pelo ficheiro de teste, com os resultados agora apresentados, é possível verificar o correto funcionamento do codificador e descodificador, sendo que os resultados destes confirmam-se um ao outro, usando vários valores de m e inteiros com sinal.

Para testar o programa desenvolvido (no terminal):

```
# Estando no diretório IC_G07_Proj2/Part_II/test_programs
> ./test

# Resultados
--------------------------------Testing encoding-----------------------------
Original value: 8
m: 4
Encoded Value: 1111000
-------------------------
Original value: 8
m: 5
Encoded Value: 111001
-------------------------
Original value: -12
m: 9
Encoded Value: 110101
```

```
------------------------------Testing decoding----------------------------
Encoded String: 1111000
m: 4
Decoded Value: 8
--------------------------
Encoded String: 111001
m: 5
Decoded Value: 8
--------------------------
Encoded String: 110101
m: 9
Decoded Value: -12
```

Parte III

Para compilar todos os programas da Parte III:

```
# Estando no diretório base do projeto
> cd Part_III/
> cmake .
> make
```

Nesta parte do projeto, as soluções desenvolvidas baseiam-se em programação preditiva, pelo que primeiramente foi feita uma análise teórica dos detalhes desta, de modo a salientar os aspetos mais importantes a ter em conta na implementação.

Princípios de Programação Preditiva

- Sejam $x^n = x_1 x_2 ... x_n$ a sequência de valores (escalares ou vetores) produzidos por uma fonte de informação até tempo n.
- A programação preditiva é baseada na codificação de uma sequência $r^n = r_1 r_2 ... r_n$, em vez da sequência original x^n, onde

$$r^n = x^n - \hat{x}^n$$

  e

$$\hat{x}^n = p(x^{n-1}) = p(x_1 x_2 ... x_{n-1})$$

- Os $\hat{x}^n$ são as estimativas e os valores da sequência r^n são os valores residuais.

O previsor usado para o cálculo dos residuais é o seguinte:

$$\begin{cases} \hat{x}_n^{(0)} &= 0 \\ \hat{x}_n^{(1)} &= x_{n-1} \\ \hat{x}_n^{(2)} &= 2x_{n-1} - x_{n-2} \\ \hat{x}_n^{(3)} &= 3x_{n-1} - 3x_{n-2} + x_{n-3} \end{cases}$$

Os residuais correspondentes:

$$\begin{cases} \hat{r}_n^{(0)} &= x_n \\ \hat{r}_n^{(1)} &= r_n^{(0)} - r_{n-1}^{(0)} \\ \hat{r}_n^{(2)} &= r_n^{(1)} - r_{n-1}^{(1)} \\ \hat{r}_n^{(3)} &= r_n^{(2)} - r_{n-1}^{(2)} \end{cases}$$

Características necessárias do previsor:

- Os valores estimados devem ser o mais perto possível dos valores reais, isto é, os valores de r_k devem ser pequenos.
- O descodificador deve ser capaz de gerar a mesma sequência, $\hat{x}^n$, de valores estimados, isto é, não pode introduzir nenhum erro durante a codificação/descodificação.
- O previsor deverá ser casual, e, em programação lossy, o previsor do encoder deverá utilizar os valores reconstruidos, $\tilde{x}^{n-1}$, em vez dos valores originais, x^{n-1}.

Exercício 4 - Lossless Audio Codec

Neste exercício foi implementado um lossless codec, partindo da teoria explicada acima. Para esta implementação, para além das novas classes e métodos criados, é usada também a classe Golomb implementada no exercício anterior, e a classe BitStream, implementada pelo grupo no projeto 1, que servirá para ler e escrever bits num ficheiro binário. De seguida são apresentadas algumas das funções implementadas mais importantes:

```
// Comprime a informação do ficheiro de áudio num ficheiro binário
void compress(const char *inFile, const char *outFile, int predOrder, char isLossy=0,
int cutBits=0);

// Descomprime o ficheiro binário no respetivo ficheiro de áudio
void decompress(const char *inFile, const char *outFile);

// Previsor do codec
vector<short> predictor(vector<short> samples, int order, char isLossy, int cutBits);

// Reconstruir os valores obtidos com o previsor
vector<short> revertPredictor(vector<short> samples, int order, char isLossy,
int cutBits);

// Calcular o valor da entropia de Shannon
float calculateEntropy(vector<short> values);

// Calcular o ratio de compressão entre dois ficheiros
float calculateCompression(const char *uncompressedFile, const char *compressedFile);
```

A codificação de um ficheiro de áudio é obtida pela chamada da função **compress**, que realiza todos os passos necessários. De uma maneira simplificada, a implementação do codificador segue os seguintes passos:

- Leitura do ficheiro de áudio para obter todos os dados necessários
- Previsão das samples de áudio lidas, através da chamada à função **predictor**. Nesta função é utilizado o previsor explicado acima para calcular os residuais
- Transformação de todos os novos valores de samples em inteiros positivos únicos, através da chamada à função **transformInteger**
- Cálculo do parâmetro **m** a utilizador no código Golomb. De maneira a ser dinâmico, este é calculado anteriormente á codificação, aplicando nos novos valores das samples (residuais já calculados pelo previsor) a seguinte fórmula :

$$m = \left\lceil \frac{-1}{log_2(\frac{media}{media+1.0})} \right\rceil$$

com média calculada como:

$$media = \frac{\sum samples}{N}$$

- Escrita do cabeçalho do ficheiro binário, que irá conter informação necessário para posterior descodificação. Este cabeçalho irá conter os seguintes campos:
  - Número de Frames do ficheiro áudio → 32 bits
  - Sample Rate do ficheiro áudio → 32 bits
  - Formato do ficheiro áudio → 32 bits
  - Valor do campo m usado na codificação de Golomb → 32 bits
  - Número de canais do ficheiro áudio → 8 bits
  - Bits quantizados pelo codec (0-15) → 8 bits
  - Ordem do previsor (1, 2 ou 3) → 2 bits
  - Tipo de compressão (lossless-0 ou lossy-1) → 1 bit
- Codificação de Golomb dos valores de samples (já residuais), através de chamadas sucessivas à função **encode**. Á medida que os valores são codificados, são também escritos no ficheiro binário usando a classe BitStream.
- Cálculo da entropia de Shannon para o original e o residual, através da chamada à função **calculateEntropy**
- Cálculo da taxa de compressão, através da chamada à função **calculateCompression**
- Retornar mensagem com informação referente à codificação

Para testar o codificador desenvolvido (no terminal):

```
# Estando no diretório IC_G07_Proj2/Part_III/test_programs
> ./testEncoding sample01.wav sample01Encoded.wav 1 0 0

# Se o ficheiro for codificado com sucesso receberá a mensagem
# (valores podem variar consoante a máquina):
Started encoding, please wait....
The audio file: sample01.wav has been sucessfully compressed
into the binary file sample01Encoded.wav
-Type of Compression: Lossless
-Quantized Bits: 0
-Predictor Order: 1
-Original Entropy: 13.2673
-Residual Entropy: 10.7858
-Compression Ratio: 1.46623
-Time Taken: 1.10227s

# USAGE:
> ./testEncoding <input audioFile.type> <output filename> seguido de
<predictor order (1, 2 or 3)> <0 (lossless) | 1 (lossy)> <bits to cut (0-15)>
```

Na seguinte tabela, são apresentados resultados referentes ao lossless codec desenvolvido. Pode ser observado que a entropia residual é sempre menor que a entropia original, e que a taxa de compressão é sempre superior a 1.3. Conclui-se então que o codificador lossless funciona como esperado.

Nome do Ficheiro	Ordem do Previsor	Entropia Original	Entropia Residual	Taxa de Compressão	Duração (s)
sample01.wav	1	13.9082	11.7262	1.35769	1.60532
	2	13.9082	11.3134	1.40897	1.70041
	3	13.9082	11.6487	1.36735	1.76223
sample02.wav	1	13.0336	11.284	1.38477	0.833044
	2	13.0336	11.1981	1.37702	0.838442
	3	13.0336	11.5948	1.31706	0.872042
sample03-wav	1	13.2673	10.7858	1.46623	1.05912
	2	13.2673	10.317	1.52593	1.12477
	3	13.2673	10.4407	1.50038	1.09567

Table 1: Resultados obtidos com o lossless codec

Por sua vez, a descodificação de um ficheiro binário é obtida pela chamada da função **decompress**, que realiza todos os passos necessários. De uma maneira simplificada, a implementação do descodificador segue os seguintes passos:

- Leitura do cabeçalho do ficheiro binário para obter todos os dados necessários
- Leitura da restante informação e respetiva descodificação dos valores usando a descodificação de Golomb
- Reversão dos valores das samples aos valores originais usando a chamada à função **revertPredictor**
- Escrita dos novos valores no novo ficheiro de áudio criado

Para testar o descodificador desenvolvido (no terminal):

```
# Estando no diretório IC_G07_Proj2/Part_III/test_programs
> ./testDecoding sample01Encoded.wav sample01Decoded.wav

# Se o ficheiro for descodificado com sucesso receberá a mensagem:
Started decoding, please wait....
The binary file sample01Encoded.wav has been sucessfully decompressed
into the audio file sample01Decoded.wav

# USAGE:
./testDecoding <input filename> <output audioFile.type>)>
```

Exercício 5 - Lossy Audio Codec

Para implementar um lossy codec, foi adicionada a possibilidade de quantizar bits. Para isto, ao codificar, é verificado se a compressão escolhida é lossy, e caso seja, são cortados os bits indicados pelo utilizador e a previsão feita de acordo. Ao descodificar, a mesma condição é verificada, sendo depois realizado um shift na direção oposta, sobre os valores, com o mesmo número de bits cortados na codificação.

Na seguinte tabela, são apresentados resultados referentes ao lossy codec desenvolvido. Neste caso podem ser observado os mesmos factos que no lossless codec, com a entropia residual e taxa de compressão mais acentuadas devido ao facto de bits terem sido quantizados. Conclui-se então que o codificador lossy funciona como esperado.

Nome do Ficheiro	Ordem do Preditor	Entropia Original	Entropia Residual	Taxa de Compressão	Duração (s)
sample01.wav	1	13.9082	8.72964	1.82135	1.62617
	2	13.9082	8.31631	1.91482	1.55688
	3	13.9082	8.65279	1.8388	1.58666
sample02.wav	1	13.0336	8.30958	1.87044	0.81418
	2	13.0336	8.21851	1.85613	0.746477
	3	13.0336	8.62098	1.74903	0.832096
sample03-wav	1	13.2673	7.81993	2.02133	0.923415
	2	13.2673	7.3462	2.13772	0.97022
	3	13.2673	7.47127	2.08753	1.11341

Table 2: Resultados obtidos com o lossy codec (quantizando 3 bits)

Para testar o codificador desenvolvido (no terminal):

```
# Estando no diretório IC_G07_Proj2/Part_III/test_programs
> ./testEncoding sample01.wav sample01Encoded.wav 1 1 3

# Se o ficheiro for codificado com sucesso receberá a mensagem
# (valores podem variar consoante a máquina):
Started encoding, please wait....
The audio file: sample01.wav has been sucessfully compressed
into the binary file sample01Encoded.wav
-Type of Compression: Lossy
-Quantized Bits: 3
-Predictor Order: 1
-Original Entropy: 13.9082
-Residual Entropy: 8.72964
-Compression Ratio: 1.82135
-Time Taken: 1.62617s

# USAGE:
> ./testEncoding <input audioFile.type> <output filename> seguido de
<predictor order (1, 2 or 3)> <0 (lossless) | 1 (lossy)> <bits to cut (0-15)>
```

Parte IV

Exercício 6 - Lossless Image Codec

Neste exercício foi implementado um lossless image codec. Mais uma vez, para fazer este exercício, utilizamos a class BitStream do primeiro projeto e a classe Golomb da parte II. Foi usado o previsor não linear do **JPEG-LS**, que implementa um previsor baseado na mesma configuração espacial do **JPEG**

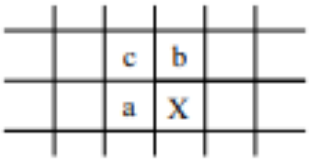

Figure 17: JPEG

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

Figure 18: Equation

Este previsor foi implementado da seguinte maneira:

```
int predictorJPEG(int a, int b, int c) {
    if (c >= max(a, b))
        return min(a, b);
    else if (c <= min(a, b))
        return max(a, b);
    else
        return a + b - c;
}
```

Para testar este codec é necessário compilar e correr o código com dois argumentos de entrada, a imagem pretendida para codificar para binário e o ficheiro de output

```
#Compilar o codigo
#IC/IC_G07_Proj2/Parte_IV usar este diretorio para a continuação desta Parte
>make


//Funcao do encoder
void encodeImage(Mat image, string output_file)

#Comando para comprimir a imagem
>../bin/test_image_codec -e lena.ppm output
```

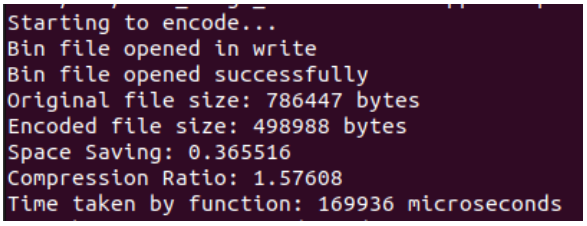

Figure 19: output do terminal do encoder

Ao fim disto o ficheiro output vai estar em binário, portanto a imagem esta comprimida

Agora para descomprimir o ficheiro teremos que compilar e correr o código de teste com dois argumentos de entrada, o ficheiro comprimido(output) do ponto atrás e a imagem nova que é descodificada.

```
// Função do decoder
void decodeImage(string filename, string output_file)

# Comando para descomprimir a imagem
>../bin/test_image_codec -d output new_image.ppm
```

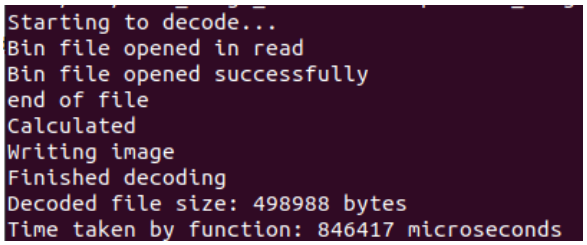

Figure 20: output do terminal do decoder

Figure 21: Imagem original

Figure 22: Imagem descomprimida

Podemos observar que a imagem descomprimida é igual a imagem original, logo o resultado que pretendia-mos obter equivale ao resultado obtido.

Contribuição dos autores

Todos os autores participaram no desenvolvimento deste projeto de forma igual, na qual a percentagem de cada membro fica:

- Antonio Ferreira - 33.33%
- Guilherme Claro - 33.33%
- Luis Couto - 33.33%