

# Informação e Codificação

## Projeto 2

Raquel Pinto (92948), Alexandre Oliveira (93289), Pedro Loureiro (92953)

Link do repositório - [https://github.com/AlexOliZ/IC\\_P2.git](https://github.com/AlexOliZ/IC_P2.git)

Na pasta html do nosso projeto pode-se observar a documentação doxygen

---

### Parte A

#### Problema 1: (BitStream)

Executar o programa: Para testar o programa, é preciso aceder à pasta “bit\_stream” que se encontra no diretório principal. Após abrir um terminal nessa mesma pasta tem que se compilar o programa. Para isso executa-se make bit\_stream.o para compilar e ./bitstream\_output para correr o programa.

Um ponto importante da bit\_stream é que a fstream não apaga o conteúdo do ficheiro quando aberto sendo assim necessário apagar o ficheiro para testar corretamente.

A bit\_stream é a classe usada para ler e escrever ficheiro bit a bit. Para escrever e ler ficheiros usou-se a livraria fstream que permite ler ou escrever ficheiros byte a byte e assim com algumas operações extra conseguiu-se manipular e escrever ficheiros bit a bit.

A bit\_stream tem duas stream's uma para escrever e outra para ler, sendo assim possível criar bit\_stream's para ler, escrever ou ambos, também é possível fechar uma das streams sem terminar a outra.

Para escrever num ficheiro, como só é possível escrever byte a byte e é necessário manipular o ficheiro bit a bit criou-se uma variável wbyte que armazena os 8 bits para escrever, cada vez que é adicionado 1 bit à stream o write\_pointer é decrementado de forma a corresponder à posição do próximo bit que vai ser inserido mantendo assim a ordem em que os bits são inseridos, quando o byte ficar completo escreve-se esse byte no ficheiro.

Para manipular o byte e modificar um bit de cada vez usou-se a seguinte operação:

```
wbyte |= (val & 0x01) << pointer_write
```

Finalmente para terminar a escrita deve-se chamar a função write\_byte() para escrever o último byte caso tenha alguma informação por escrever e fechar a ofstream.

Para ler foi usada a ifstream que lê 1 byte de cada vez, e dá-se return a um bit de cada vez que a função read\_bit é chamada usando um pointer para indicar a posição do próximo bit a ser lido e a variável rbyte para armazenar o último byte que foi lido do ficheiro. Quando o ponteiro tiver percorrido o byte é lido um novo byte e é reposto o ponteiro no bit mais significativo.

Para manipular o byte e modificar um bit de cada vez usou-se a seguinte operação:

```
rbyte |= (val >> pointer_read) & 0x01
```

É importante que a leitura e a escrita tenham a mesma ordem para manter a ordem em que a informação foi introduzida.

Também se disponibilizou diferentes operações de escrita e leitura que permite inserir vários bits de uma vez usando ponteiros de forma a tornar o programa mais eficiente e permitir uma melhor gestão da memória usada na escrita e leitura de um ficheiro.

### Problema 3: (Golomb)

Executar o programa: Para testar o programa, é preciso aceder à pasta “Golomb” que se encontra no diretório principal. Após abrir um terminal nessa mesma pasta tem que se compilar o programa. Para isso executa-se make golomb.o para compilar e ./golomb\_output <M> <max\_n> para correr o programa.

O método de codificação de golomb é uma maneira simples de comprimir números inteiros de forma rápida e eficiente sem perda de informação, isso é possível através da divisão do código em 2 partes, a parte unária e o resto, a parte unária é necessária para a descodificação do código, pois permite determinar o número de bits usado para codificar o resto.

### Codificação:

Na codificação de um valor  $n$  utilizou-se as equações 1, 2, 3 e 4.

$$q = \lfloor \frac{n}{M} \rfloor \quad (1)$$

$$r = n - qM \quad (2)$$

$$b = \lceil \log_2(M) \rceil \quad (3)$$

$$k = 2^{b+1} - M \quad (4)$$

Assim determinou-se o número de bits usados para codificar a parte unária  $q$  e a parte do resto  $r$ .

A parte unária corresponde aos primeiros  $q$  bits do código todos a 0, depois para separar a parte unária do resto adicionou-se um bit a 1 e por fim adicionou-se o resto. O resto é o valor de  $r$  codificado em  $(b - 1)$  bits caso  $r$  seja menor que  $k$ , mas se  $r$  for maior que  $k$  codifica-se o valor de  $(r + k)$  em  $b$  bits.

Este algoritmo é suficiente para codificar inteiros positivos. Para codificar inteiros positivos ou negativos decidiu-se corresponder os valores ímpares aos valores negativos e os valores pares aos valores positivos.

Assim conseguiu-se codificar ambos os casos usando códigos de golomb com uma pequena penalização no número de bits devido ao aumento do valor de  $n$ , como se pode observar na equação 5.

$$\text{encode } n = n \geq 0 ? 2n : -2n - 1 \quad (5)$$

### Descodificação:

Para decodificar o código de golomb primeiramente verificou-se quantos 0's existem a partir do bit mais significativo até se encontrar um bit a 1. Descobrindo assim o valor de  $q$ . Após descobrir o valor de  $q$  conseguiu-se calcular o  $r$ . Como  $b$  e  $k$  são constantes, basta fazer a operação inversa da codificação lendo os próximos  $b - 1$  bits que correspondem a  $r$ . Se  $r$  for menor que  $k$  então  $n = qM + r$ , mas se  $r$  for maior ou igual a  $k$  então temos que adicionar mais 1 bit a  $r$  e  $n = qM + (r - k)$ .

Como na codificação este algoritmo só consegue decodificar inteiros positivos, se se quiser decodificar inteiros negativos e positivos usa-se o mesmo método e após encontrar o valor de  $n$  obtido na descodificação tem que se repor ao valor original usando a equação 6.

$$signed n \% 2 ? - \lceil \frac{res+1}{2} \rceil : \frac{res}{2} \quad (6)$$

Nota: A class golomb tem duas funções para codificar e decodificar, as funções usadas para comprimir o ficheiro são a stream\_decode e stream\_encode que escrevem diretamente no ficheiro, as outras são usadas apenas para testar o algoritmo de codificação.

## Parte B

### Problema 1: (Codec Lossless)

Executar o programa: Neste exercício o ficheiro de áudio que o programa lê deve estar dentro de uma pasta chamada “wavfiles”. Esta pasta deve estar na mesma pasta que o programa. Como se trabalhou com a biblioteca *libsndfile*, com a biblioteca *OpenCV*, com o Golomb e com a bit-stream desenvolvidos anteriormente, ao compilar o código tem que se, para além de abrir um terminal na pasta do programa, correr o comando

```
make lossless.o
```

ou

```
g++ lossless_predictive.cpp Golomb/golomb.cpp predictor.cpp  
bit_stream/bit_stream.cpp -lsndfile -o lossless_output -std=c++17 `pkg-config  
--cflags --libs opencv4`
```

De modo a executar o programa utilizou-se o comando `./losslessaudio <nome do áudio> <calchist>` (por exemplo: `./losslessaudio sample0N.wav calcHist 0`, onde N pertence ao intervalo [1, 6] e calcHist decide se calcula o histogramas [0,1]).

Este exercício consistia em desenvolver um codec áudio sem perda de qualidade. Para isso desenvolveu-se uma classe *lossless\_predictive* que possui 2 métodos principais: o encode e o decode, em que o primeiro lê o ficheiro áudio fornecido e escreve o resultado das operações efetuadas (descritas a seguir) num ficheiro binário. O segundo método transforma o binário em áudio de novo.

O encode é responsável por ler o ficheiro áudio e guardar as samples num array. Em seguida, percorre esse array lido e calcula os valores residuais através do predictor. Para calcular o residual, o predictor usa o sistema de equações 7 para prever o próximo valor e subtrair o valor obtido ao valor original da sample.

$$\left\{ \begin{array}{l} \hat{x}_n^{(0)} = 0 \\ \hat{x}_n^{(1)} = x_{n-1} \\ \hat{x}_n^{(2)} = 2x_{n-1} - x_{n-2} \\ \hat{x}_n^{(3)} = 3x_{n-1} - 3x_{n-2} + x_{n-3} \end{array} \right. \quad (7)$$

Os residuais calculados são de seguida codificados em código de Golomb (Figura 1) e daí são escritos num ficheiro em binário, bit a bit, através do módulo *bitstream* previamente descrito.

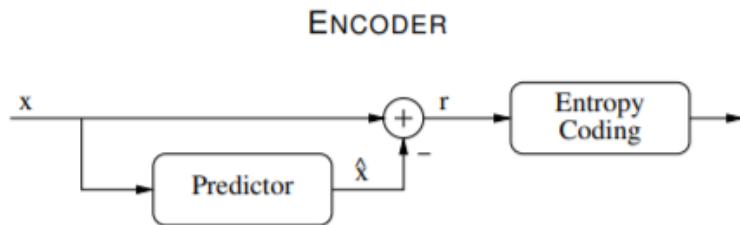


Figura 1 - codificador usado para codificar residuais de áudio.

O decoder (Figura 2) faz o percurso inverso ao encoder de forma a obter os valores originais. Ou seja, inicialmente a bitstream lê os valores, bit a bit, do ficheiro binário criado no encoder, em seguida o golomb descodifica o número previamente codificado e por fim, o predictor “reconstrói” o valor original através da soma do valor residual com uma nova predição do próximo valor, fazendo assim o inverso do encoder que subtrai o original ao valor do predictor.

Os valores obtidos são guardados num array que depois a biblioteca libsnfile escreve num novo ficheiro áudio.

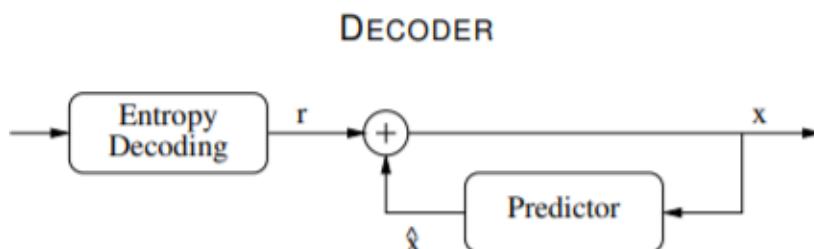


Figura 2 - descodificador usado para ficheiros de áudio.

## Problema 2: (Histogramas e Entropia)

Executar o programa: Para verificar os histogramas e entropias é preciso ativar uma flag nos programas lossless ou lossy para isso é preciso substituir o último argumento de false para true. Ex: ./lossy\_output sample01.wav 8 1.

Entropia Lossless Original (sample01.wav): 3.47714

Entropia Lossless Residuals (sample01.wav): 3.26908

Entropia Lossless Original (sample07.wav): 3.79624

Entropia Lossless Residuals (sample07.wav): 3.65077

Entropia Lossy Residuals 8bits (sample01.wav): 1.25892

Entropia Lossy Residuals 8bits (sample07.wav): 1.65386

Entropia Lossy Residuals 4bits (sample01.wav): 0.712157

Entropia Lossy Residuals 4bits (sample07.wav): 0.807469

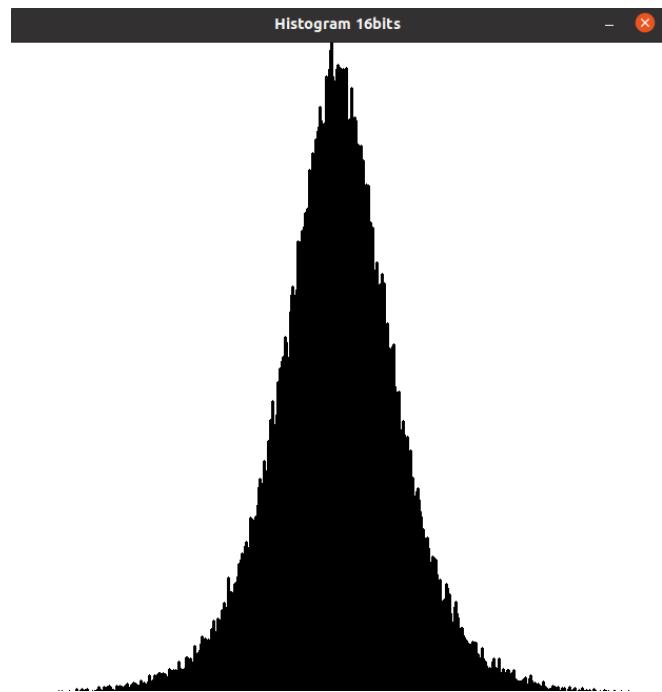


Figura 3 - Histograma dos resíduais da sample01 quantizados a 16 bits.

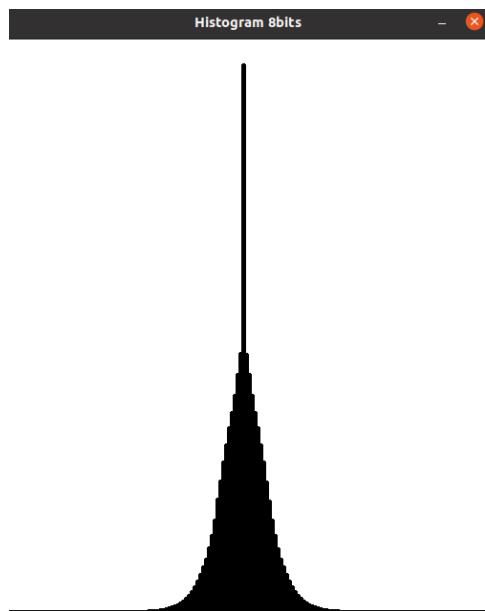


Figura 4 - Histograma dos resíduais da sample01 quantizados a 8 bits.

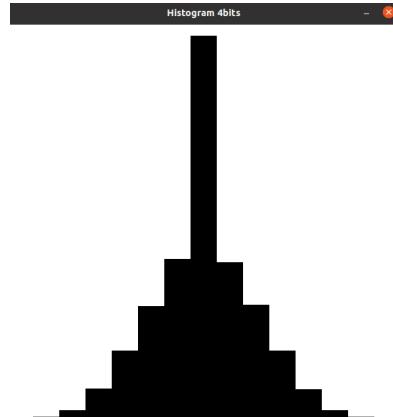


Figura 5 - Histograma dos residuais da sample01 quantizados a 4 bits.

Como é possível observar, a entropia diminui com uma quantização mais baixa, já que são cada vez mais limitados os valores disponíveis para codificar cada sample de áudio. Além disso, é possível observar os histogramas a ficarem com o valor do zero relativamente maior do que os outros já que é a zona que engloba o maior número de samples. Os histogramas não estão feitos à mesma escala, já que o máximo valor do histograma de 16 bits é 654, de 8 bits é 287214 e de 4 bits é 875345.

### Problema 3: (Codec Lossy)

Executar o programa: Neste exercício o ficheiro de áudio que o programa lê deve estar dentro de uma pasta chamada “wavfiles”. Esta pasta deve estar na mesma pasta que o programa. Como se trabalhou com a biblioteca *libsndfile*, com a biblioteca *OpenCV*, com o Golomb e com a bit-stream desenvolvidos anteriormente, ao compilar o código tem que se, para além de abrir um terminal na pasta do programa, correr o comando:

```
make lossy.o
ou
g++ lossy_predictive.cpp Golomb/golomb.cpp predictor.cpp
bit_stream/bit_stream.cpp -lsndfile -o lossy_output -std=c++17 `pkg-config --cflags
--libs opencv4`
```

De modo a executar o programa utilizou-se o comando `./lossyaudio <nome do áudio> <quantizationbits> <calcHist>` (por exemplo: `./lossyaudio sample0N.wav, onde N pertence ao intervalo [1, 6], quantizationbits [4,8,16] e calcHist decide se calcula o histogramas [0,1]).).`

Neste exercício foi pedido uma alteração ao codec lossless de modo a suportar maior compressão à custa de qualidade de som. Este possui uma função

extra que quantiza o valor consoante o número de bits que se quer alcançar. No codec lossy os valores a quantizar são os residuais que resultam do predictor. Assim, a diferença para o lossless é que os valores antes de serem codificados pelo Golomb são quantizador para n bits, o que resulta em perda de informação já que só existem certos valores que a sample pode ter.

O módulo de quantização de bits usa a fórmula 8.

$$quantsample = delta * \text{floor}(sample/delta + 0.5) \quad (8)$$

onde o delta é dado pela fórmula 9.

$$\text{delta} = (\text{MAXINT} - (-\text{MAXINT})) / 2^{nbits} - 1 \quad (9)$$

### Cálculo do M ideal:

Tanto no lossy como no lossless, inicialmente o M é calculado usando a fórmula 10.

$$M = \log_2\left(\frac{-1}{(\text{average} \div (\text{average}+1))}\right) \quad (10)$$

A média não é uma boa maneira de representar a distribuição dos valores, já que com o uso do predictor haverá um número elevado de valores próximos de zero com alguns valores muito superiores. Por causa disto, se ocorrerem poucos valores muito elevados junto com um número elevado de valores próximos de zero, vai dar um valor baixo de M que irá dar problemas com os valores mais elevados levando a uma má taxa de compressão.

### Lossless

ficheiro	tamanho original	tamanho comprimido	taxa de compressão
sample01	5.2 Mb	4.4 Mb	0.85
sample02	2.6 Mb	4.2 Mb	1.62
sample03	3.5 Mb	2.7 Mb	0.77
sample04	2.4 Mb	1.9 Mb	0.82
sample05	3.6 Mb	2.6 Mb	0.74
sample06	4.2 Mb	2.8 Mb	0.65
sample07	3.8 Mb	10.5 Mb	2.78

### Lossy com 4 bits

ficheiro	tamanho original	tamanho comprimido	taxa de compressão
sample01	5.2 Mb	7.0 Mb	1.35
sample02	2.6 Mb	3.4 Mb	1.30
sample03	3.5 Mb	4.8 Mb	1.36
sample04	2.4 Mb	2.0 Mb	0.84
sample05	3.6 Mb	2.6 Mb	0.71
sample06	4.2 Mb	2.5 Mb	0.59
sample07	3.8 Mb	6.0 Mb	1.58

Como podemos ver os valores variam muito e em grande parte dos casos o ficheiro comprimido é maior que o original.

Para corrigir este problema optamos por calcular a média e contar o número de valores abaixo da média através das seguintes fórmulas:

$$val = \frac{average}{count \ below \ average} \quad (11)$$

$$M = \log_2\left(\frac{-1}{(val \div (val+1))}\right) \quad (12)$$

Assim conseguimos mitigar o problema que ocorria com ficheiros com uma distribuição pouco uniforme, levando a um M que representa melhor os valores codificados.

### Lossless sem histograma:

ficheiro	tamanho original	tamanho comprimido	taxa de compressão	tempo de compressão(ms)	tempo de descompressão(ms)
sample01	5.2 Mb	4.3 Mb	0.83	0.43	0.55
sample02	2.6 Mb	2.2 Mb	0.85	0.23	0.27
sample03	3.5 Mb	2.7 Mb	0.78	0.30	0.35
sample04	2.4 Mb	2.0 Mb	0.83	0.20	0.24
sample05	3.6 Mb	2.7 Mb	0.75	0.30	0.37

sample06	4.2 Mb	2.8 Mb	0.66	0.31	0.38
sample07	3.8 Mb	3.5 Mb	0.91	0.42	0.42

Lossy com 4 bits sem histograma:

ficheiro	tamanho original	tamanho comprimido	taxa de compressão	tempo de compressão(ms)	tempo de descompressão(ms)
sample01	5.2 Mb	5.1 Mb	0.99	0.59	0.60
sample02	2.6 Mb	2.6 Mb	0.98	0.30	0.29
sample03	3.5 Mb	3.5 Mb	0.99	0.42	0.40
sample04	2.4 Mb	1.9 Mb	0.82	0.25	0.23
sample05	3.6 Mb	2.6 Mb	0.71	0.34	0.35
sample06	4.2 Mb	2.5 Mb	0.59	0.37	0.31
sample07	3.8 Mb	3.9 Mb	1.03	0.42	0.43

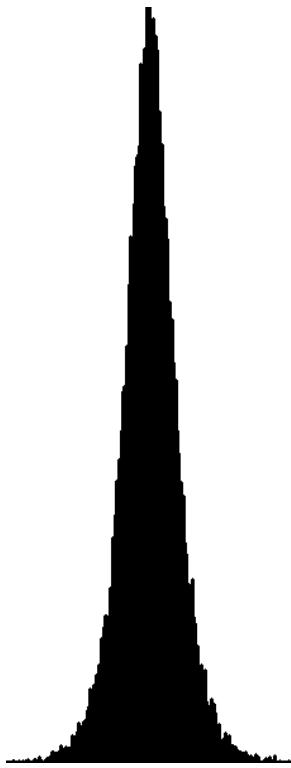


Figura 6 - Histograma sample01.wav.

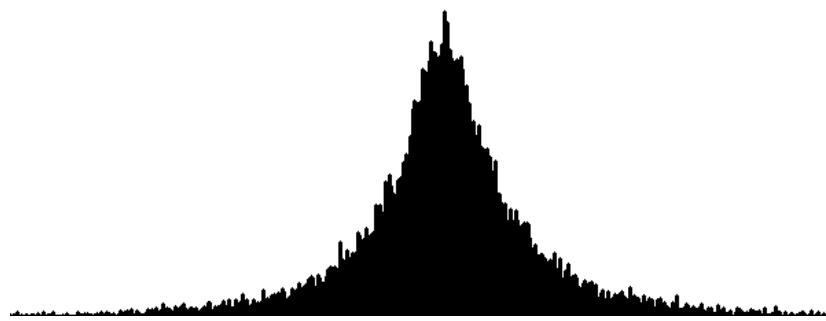


Figura 7 - Histograma sample07.wav.

Estes foram os 2 casos que se mostraram mais problemáticos, pois são casos opostos sendo que o sample01 tem muitos valores próximo de 0 com alguns valores superiores tendo uma melhor taxa de compressão com um M ligeiramente superior àquele que se obteve usando a média, enquanto que o sample07 obtém melhores resultados com um M bastante mais elevado, sendo assim é difícil encontrar um método que se adaptasse aos 2 casos.

Como só vamos contar os valores abaixo da média vai levar a que um M superior em ambos os casos, mas como no sample01 a maior parte dos valores são inferiores à média esse aumento vai ser menor, contrariamente a sample07 vai haver um aumento mais significativo devido ao maior número de valores acima da média.

## Parte C

Nesta parte do nosso trabalho foi pedido para implementar um codec de imagem utilizando o algoritmo de codificação Golomb. Este codec deve contar com a codificação preditiva. Considerou-se uma fase de pré-processamento a fim de transformar a imagem para o formato YUV 4:2:0 antes do processo de codificação.

### Problema 1:

Executar o programa: Para executar o programa, a imagem que o programa lê deve estar dentro de uma pasta chamada imagensPPM. Esta pasta deve estar na mesma pasta que o programa. Como se trabalhou com a biblioteca *OpenCV*, com o Golomb e com a bit-stream desenvolvidos anteriormente, ao compilar o código tem que se, para além de abrir um terminal na pasta do programa, correr o comando `g++ ex1_parteC.cpp ./bit_stream/bit_stream.cpp ./Golomb/golomb.cpp -o ex1 `pkg-config --cflags --libs opencv``.

De modo a executar o programa utilizou-se o comando `./ex1_parteC <input image> <output filename>` (por exemplo: `ex1_parteC lena.ppm lena.txt`). É importante referir que o ficheiro de texto onde será escrito os códigos de Golomb não deverá ser o mesmo de imagem para imagem.

Pode-se observar os detalhes da classe `lossless_codec` na documentação acedida através da pasta html no nosso projeto.

Neste exercício, foi pedido para desenvolver um codificador sem perdas onde as frames devem ser codificadas utilizando uma codificação espacial preditiva baseada no preditor não linear de JPEG-LS, a codificação da entropia deve ser realizada utilizando códigos Golomb e toda a informação requerida pelo descodificador deve ser incluída na bit-stream.

Para isto, primeiramente percorreu-se a imagem original pelas suas linhas e para cada linha percorremos todas as colunas (um ciclo *for* dentro de outro ciclo *for*), separando as componentes RGB. A seguir aplicou-se as fórmulas 13, 14 e 15 para converter uma imagem RGB numa imagem YUV.

$$Y = 16 + 65.481 * R + 128.553 * G + 24.966 * B \quad (13)$$

$$U = 128 - 37.797 * R - 74.203 * G + 112.0 * B \quad (14)$$

$$V = 128 + 112.0 * R - 93.786 * G - 18.214 * B \quad (15)$$

Obteve-se assim três matrizes uma para cada componente ( $Y$ ,  $U$  e  $V$ ), onde  $Y$  pertence ao intervalo  $\{16, \dots, 235\}$  e  $U$  e  $V$  pertencem ao intervalo  $\{16, \dots, 240\}$ .

Também se considerou que para reduzir a quantidade de informação de uma imagem, diminuiu-se a informação das componentes U e V. Para isso percorreu-se a matriz U e V e retirou-se as colunas e as linhas ímpares de cada uma dessas matrizes (descartou-se metade da informação nas colunas e nas linhas), assim se a matriz for de 100:100 no final fica-se com uma matriz 50:50.

Assim, ficou-se com três matrizes (Y, U e V) sendo que as matrizes U e V têm metade das colunas e das linhas que a matriz Y.

Como se pode ver nas Figuras 8, 9, 10, 11, 12 e 13 as componentes U e V são metade da componente Y. É de notar que as imagens são a preto e branco, pois estas componentes da imagem só têm um canal. Este exercício foi testado para todas as imagens dadas, mas por uma questão de simplicidade apresentou-se apenas 6 destas imagens.

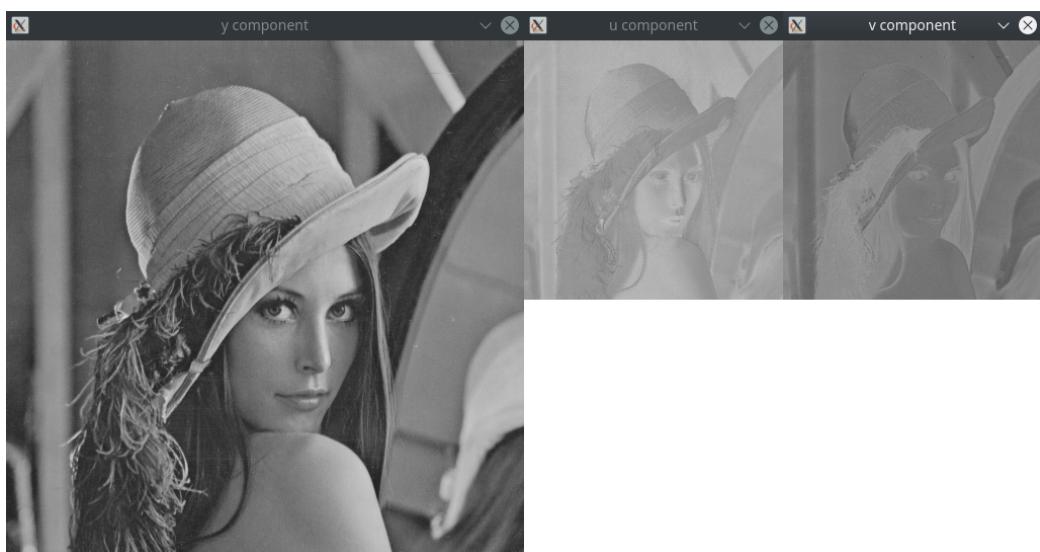


Figura 8 - Componentes y, u e v da imagem lena.ppm.

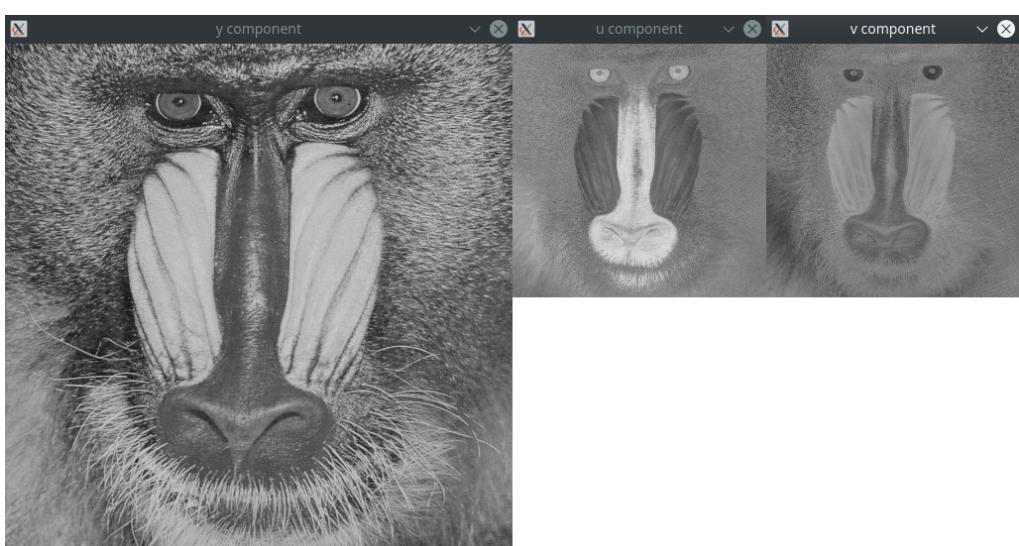


Figura 9 - Componentes y, u e v da imagem baboon.ppm.

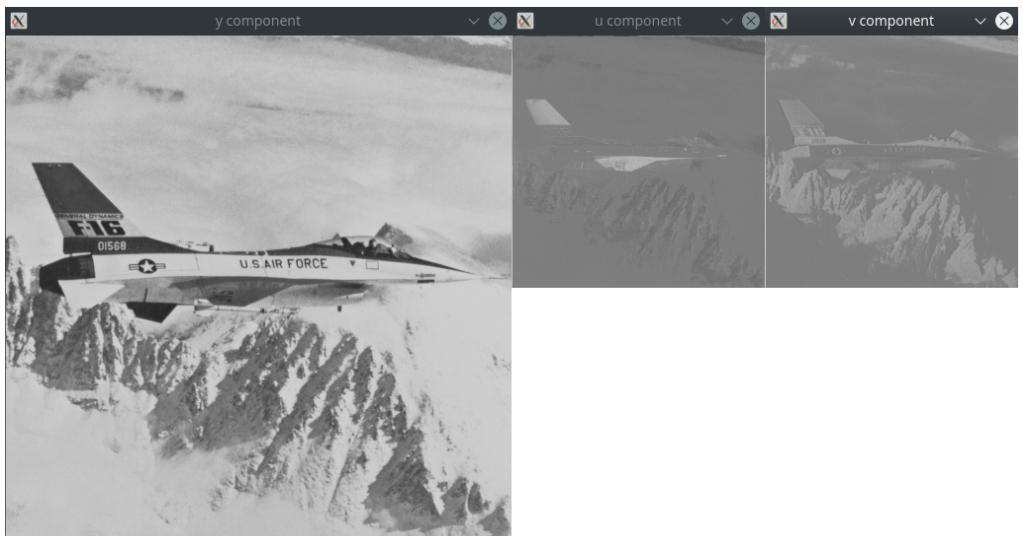


Figura 10 - Componentes y, u e v da imagem airplane.ppm.

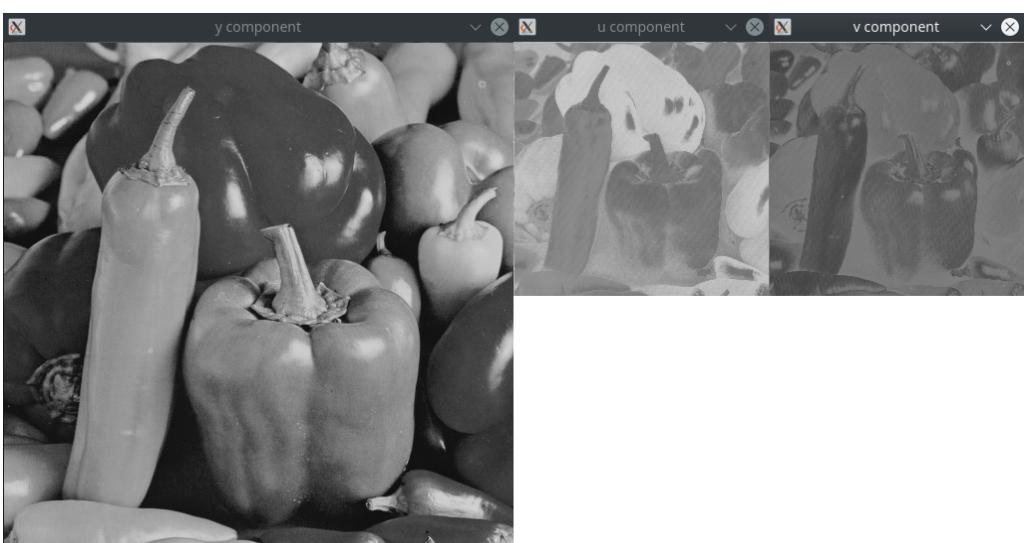


Figura 11 - Componentes y, u e v da imagem peppers.ppm.

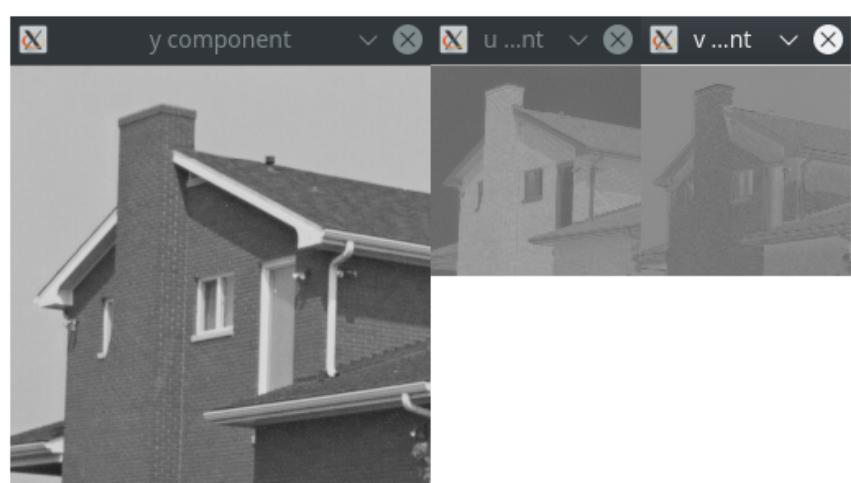


Figura 12 - Componentes y, u e v da imagem house.ppm.

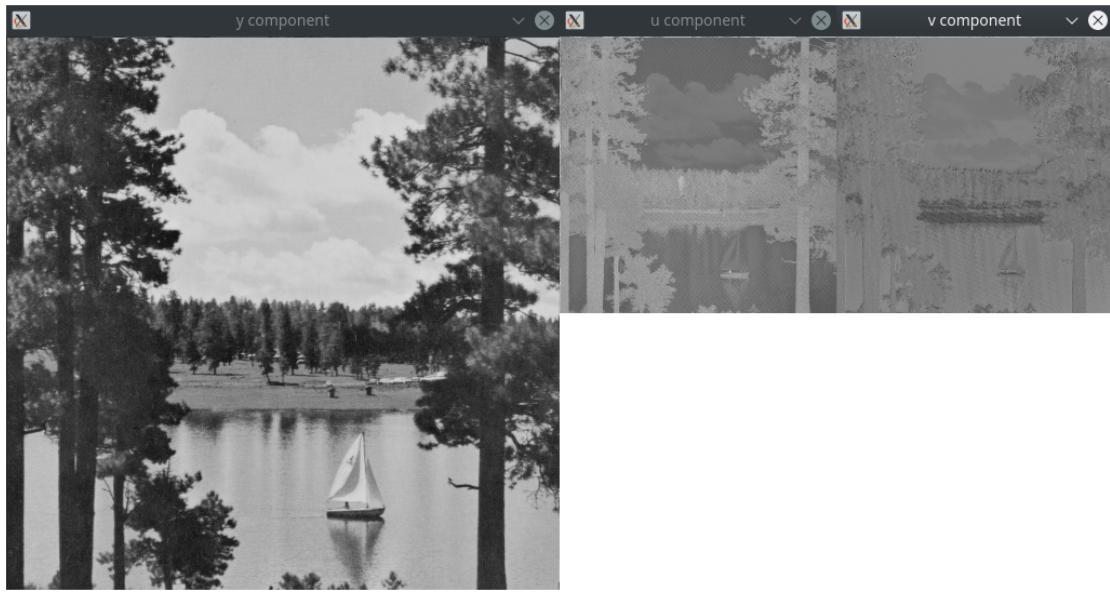


Figura 13 - Componentes y, u e v da imagem boat.ppm.

De seguida calculou-se as estimativas dos valores dos pixels para cada matriz (Y, U e V) através das fórmulas 16, 17 e 18.

$$predictor = \min(a, b) , se c \geq \max(a, b) \quad (16)$$

$$predictor = \max(a, b) , se c \leq \min(a, b) \quad (17)$$

$$predictor = a + b - c , caso contrário \quad (18)$$

onde a, b e c são os valores dos pixels próximos do pixel estimado como se pode ver na Figura 14.

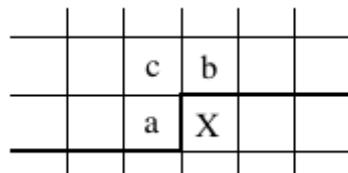


Figura 14 - Posições dos pixels a, b e c.

Foi preciso ter cuidado quando se calcula a estimativa para valores da primeira coluna ou da primeira linha, pois não existe a, b ou c. Assim, demos-lhe valor 0.

A partir do cálculo do *predictor* calculou-se o residual ( $r_n$ ) para cada valor dos pixels das três matrizes (Y, U e V) através da fórmula 20.

$$r_n = x_n - predictor \quad (20)$$

onde  $x_n$  é o valor do pixel em questão.

Codificou-se estes residuais usando o encoder do Golomb que retorna códigos para cada valor, como foi explicado na Parte A.

Depois disto, escreveu-se todos os códigos de Golomb num ficheiro através da bit-stream, como referido na Parte A.

Para o descodificador, leu-se através da bit-stream este ficheiro com os códigos de Golomb. Usou-se o descodificador de Golomb explicado na Parte A, obtendo-se assim os residuais da imagem.

Perante estes valores, fez-se o inverso do processo descrito anteriormente, ou seja, calculou-se o *predictor* através das fórmulas 16, 17 e 18.

De seguida calculou-se o valor de cada pixels através da fórmula 21.

$$x_n = r_n + \text{predictor} \quad (21)$$

onde  $x_n$  é o valor do pixel em questão e  $r_n$  é o residual obtido através do descodificador do Golomb .

Assim tendo os valores de cada pixel, converteu-se as matrizes Y, U e V descodificadas numa imagem RGB usando as fórmulas 22, 23 e 24.

$$R = 1.164 * (y - 16) + 1.596 * (v - 128) \quad (22)$$

$$B = 1.164 * (y - 16) + 2.018 * (u - 128) \quad (23)$$

$$G = 1.164 * (y - 16) - 0.813 * (u - 128) - 0.391 * (v - 128) \quad (24)$$

onde R, G e B pertencem ao intervalo {0,...,255}.

Com isto, como podemos observar nas Figuras 14, 15, 16, 17, 18 e 19 que o resultado obtido foi o esperado, pois as imagens obtidas são iguais às imagens originais.

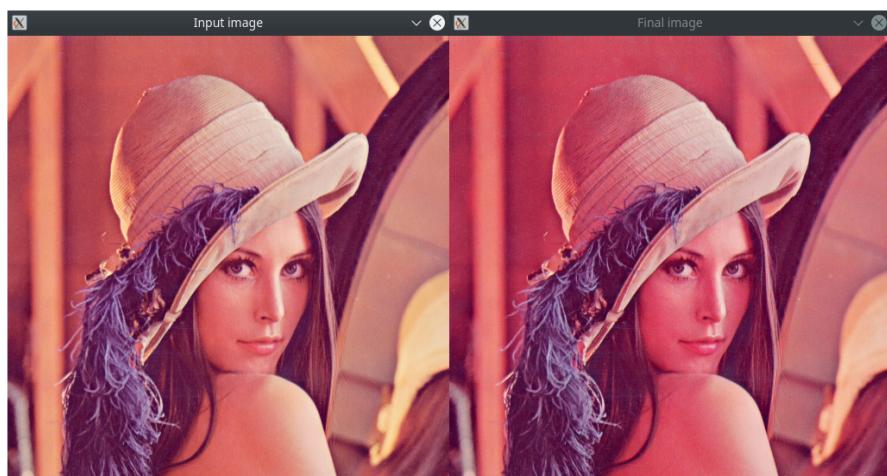


Figura 14 - Imagem inicial e final da lena.ppm.

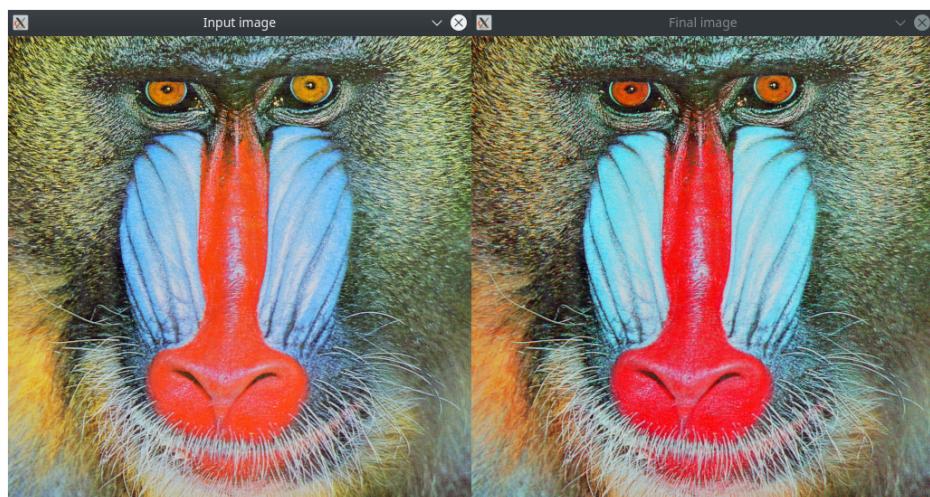


Figura 15 - Imagem inicial e final da baboon.ppm.

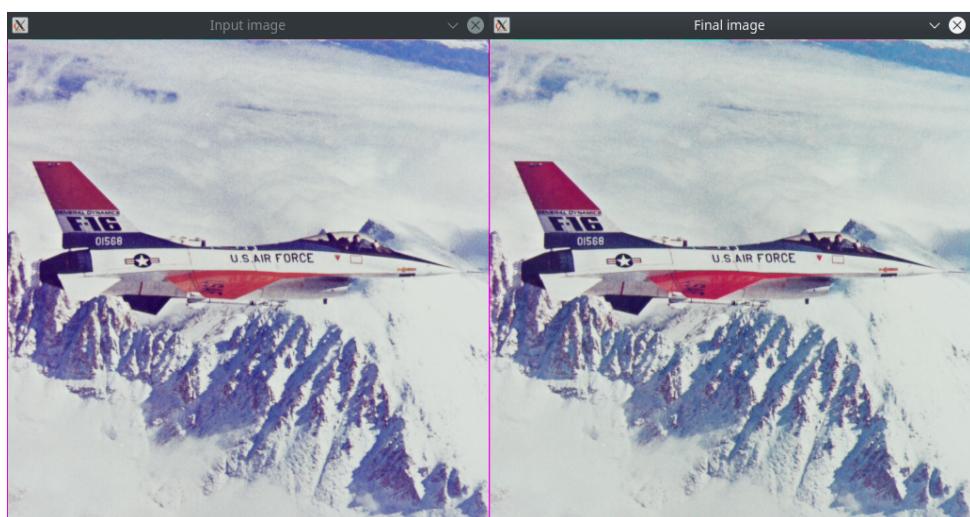


Figura 16 - Imagem inicial e final da airplane.ppm.

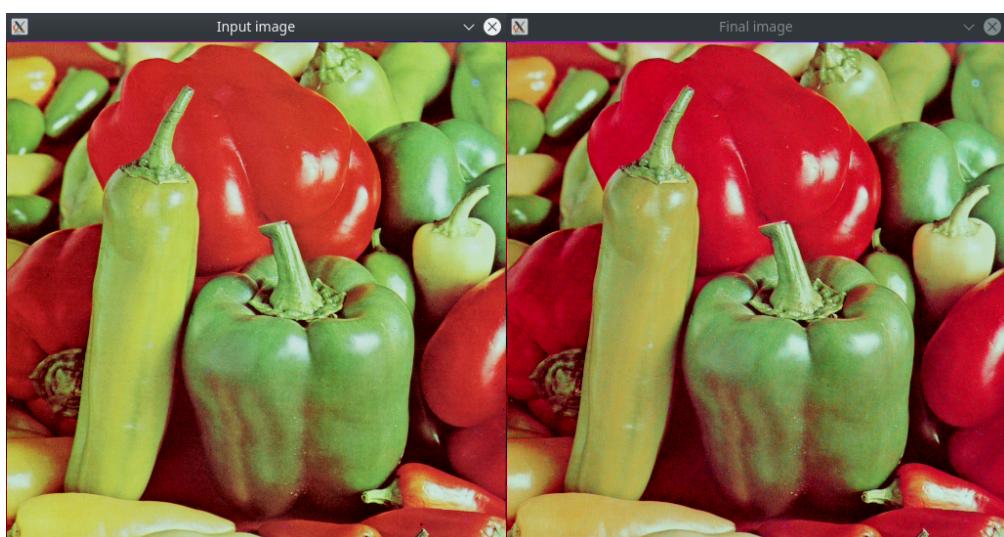


Figura 17 - Imagem inicial e final da peppers.ppm.

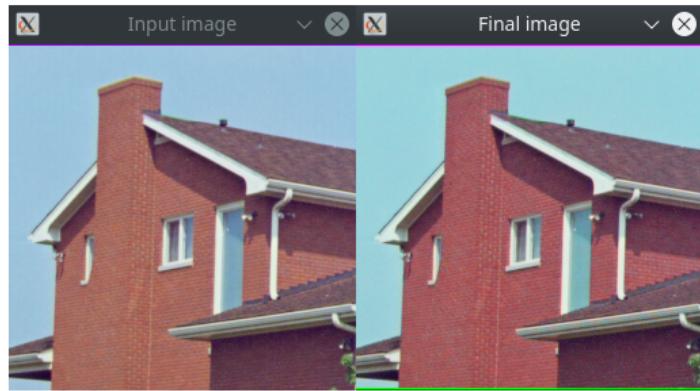


Figura 18 - Imagem inicial e final da house.ppm.

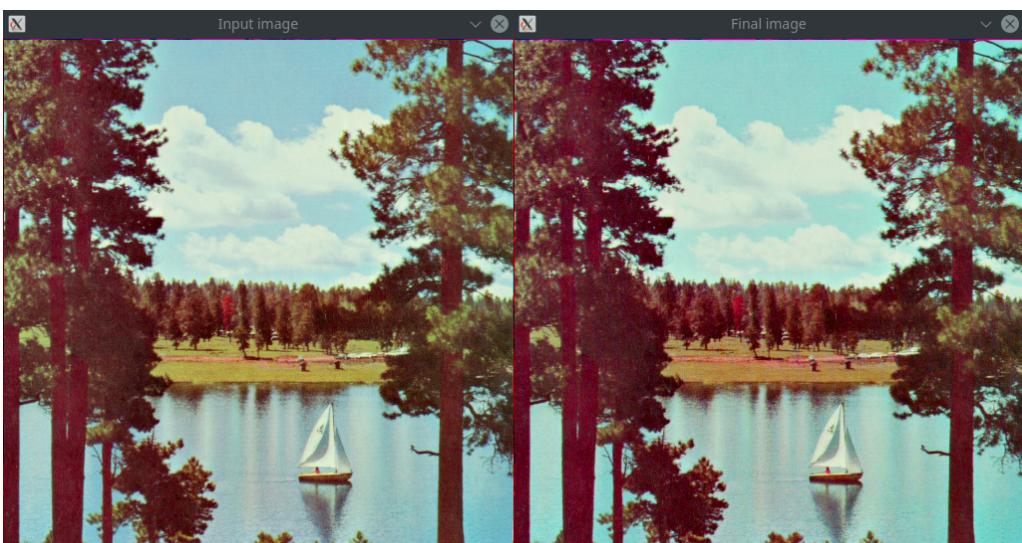


Figura 19 - Imagem inicial e final da boat.ppm.

Este programa foi testado para todas as imagens dadas, mas por uma questão de simplicidade apresentou-se apenas 6 destas imagens.

É de notar que em algumas destas imagens existe uma mudança leve na cor. Isto acontece pois ao transformar as imagens de RGB para YUV e de YUV para RGB, nas equações 16, 17, 18, 22, 23 e 24 acabou por haver arredondamentos perdendo-se assim algumas casas decimais que influenciam a cor no final.

## Problema 2:

Executar o programa: Para executar o programa, a imagem que o programa lê deve estar dentro de uma pasta chamada `imagensPPM`. Esta pasta deve estar na mesma pasta que o programa. Como se trabalhou com a biblioteca `OpenCV`, com o Golomb e com a bit-stream desenvolvidos anteriormente, ao compilar o código tem que se, para além de abrir um terminal na pasta do programa, correr o comando

```
g++ ex2_parteC.cpp ./bit_stream/bit_stream.cpp ./Golomb/golomb.cpp -o ex2  
`pkg-config --cflags --libs opencv`.
```

De modo a executar o programa utilizou-se o comando `./ex2_parteC <input image> <output filename>` (por exemplo: `ex2_parteC lena.ppm lena.txt`). É importante referir que o ficheiro de texto onde será escrito os códigos de Golomb, não deverá ser o mesmo de imagem para imagem.

Pode-se observar os detalhes da classe `lossy_coding` na documentação acedida através da pasta html no nosso projeto.

Neste exercício, foi pedido para alargar o problema anterior de modo a permitir uma codificação com perdas, ou seja, introduzir quantização. Os valores serão codificados por entropia utilizando códigos Golomb.

Para acrescentar quantização ao exercício anterior, alterou-se a equação 20, ou seja, onde se calcula o residual ( $r_n$ ) para cada valor dos pixels das três matrizes (Y, U e V). Assim calculou-se o  $r_n$  a partir das equações 25 e 26.

$$Auxr_n = x_n - predictor \quad (25)$$

$$r_n = Auxr_n + ((Auxr_n >> quantization) << quantization) \quad (26)$$

onde  $x_n$  é o valor do pixel em questão.

Com isto, teve-se que alterar também a equação 21 onde se calcula o valor de cada pixels, passando a ser a fórmula 27.

$$x_n = (r_n + predictor) << quantization \quad (27)$$

onde  $x_n$  é o valor do pixel em questão e  $r_n$  é o residual obtido através do descodificador do Golomb.

Isto vai fazer com que existam perdas na imagem, ou seja, a imagem vai aparecer mais desfocada quanto maior for a quantização como podemos ver nas Figuras 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30 e 31.

Este programa foi testado para todas as imagens dadas, mas por uma questão de simplicidade apresentou-se apenas 6 destas imagens.

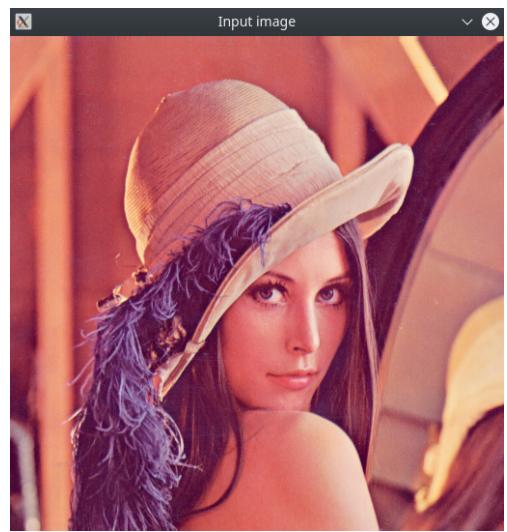
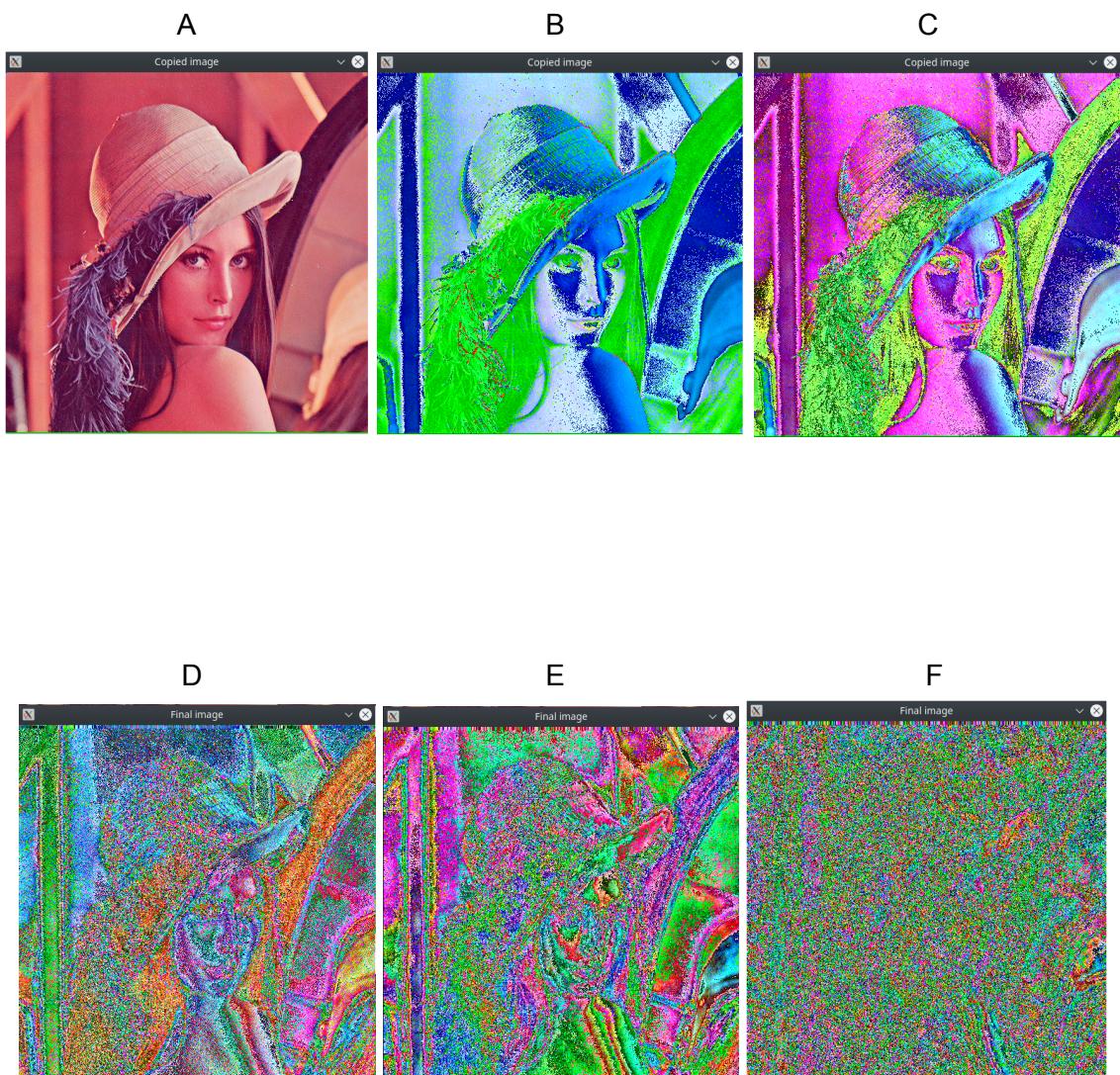


Figura 20 - Imagem original lena.ppm.



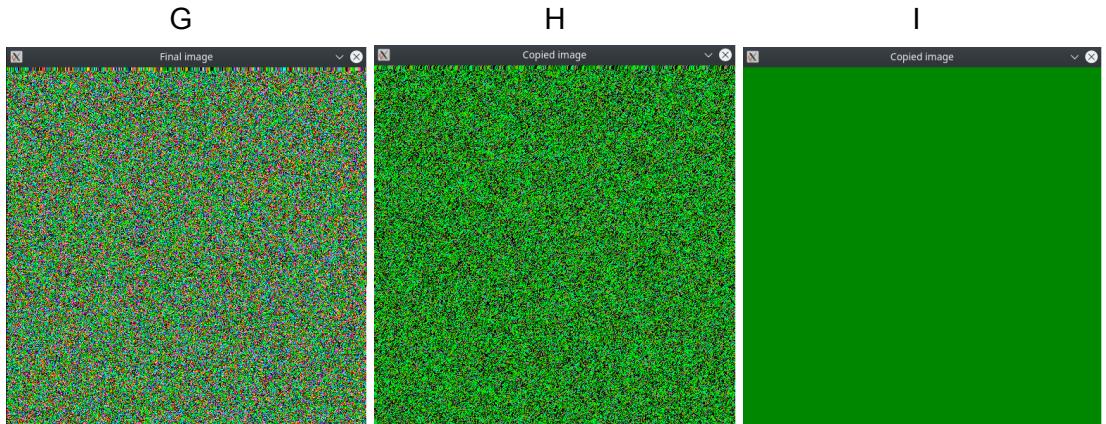


Figura 21 - Imagem modificada 0 bits,  $quantization = 0$  (A), imagem modificada 1,  $quantization = 1$  bit (B), imagem modificada 2 bits,  $quantization = 2$  (C), imagem modificada 3 bits,  $quantization = 3$  (D), imagem modificada 4 bits,  $quantization = 4$  (E), imagem modificada 5 bits,  $quantization = 5$  (F), imagem modificada 6 bits,  $quantization = 6$  (G), imagem modificada 7 bits,  $quantization = 7$  (H), imagem modificada 8 bits,  $quantization = 8$  (I) para a imagem lena.ppm.

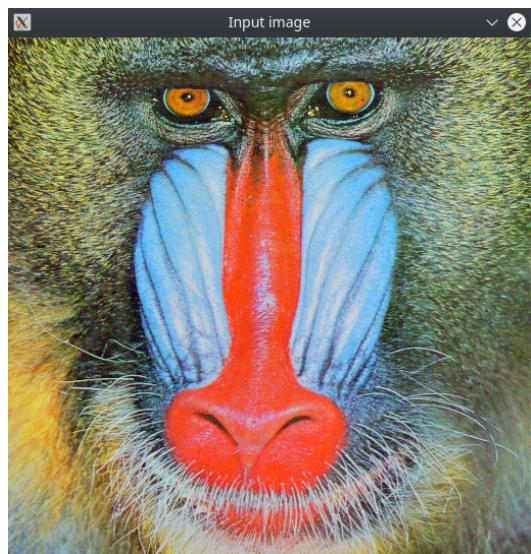
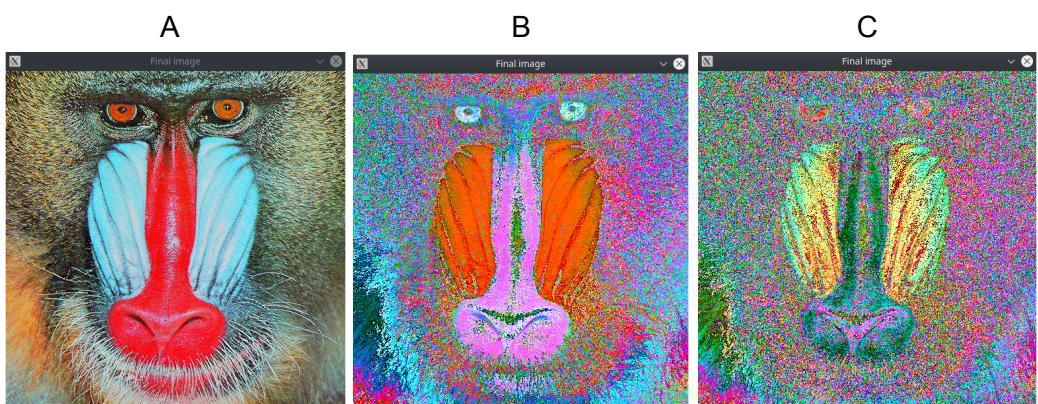


Figura 22- Imagem original baboon.ppm.



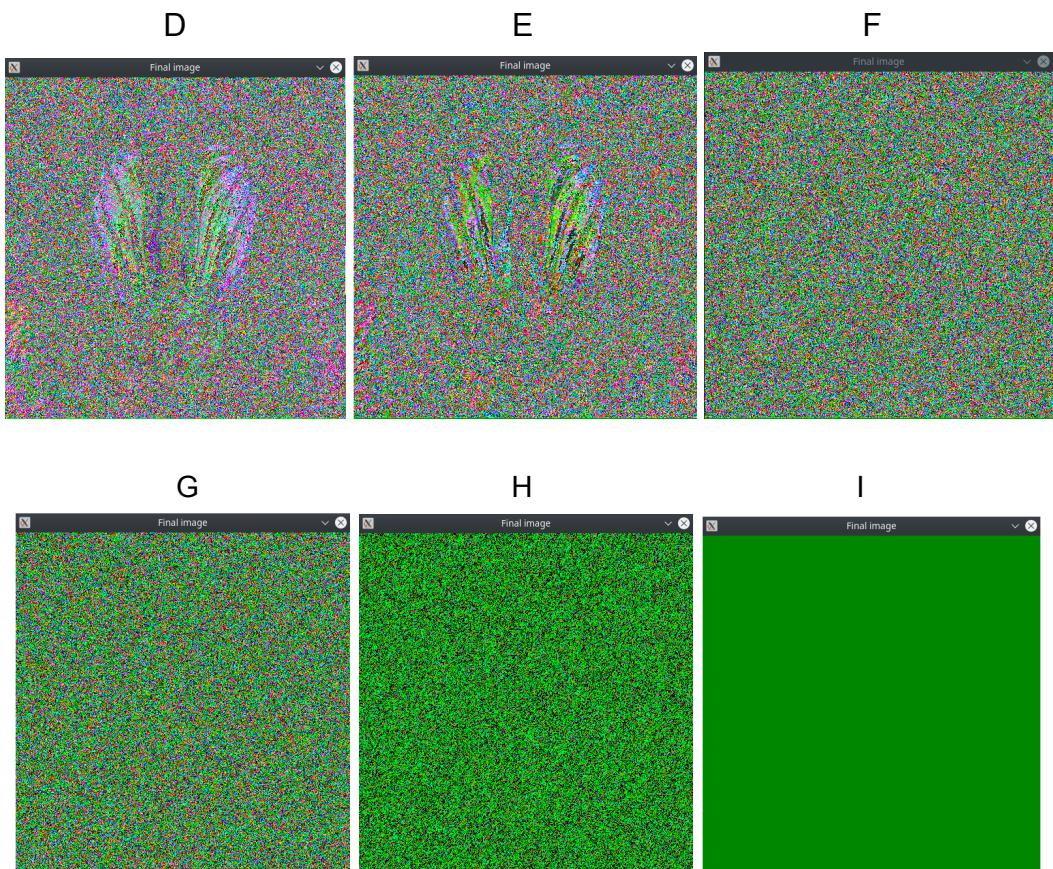


Figura 23 - Imagem modificada 0 bits,  $quantization = 0$  (A), imagem modificada 1,  $quantization = 1$  bit (B), imagem modificada 2 bits,  $quantization = 2$  (C), imagem modificada 3 bits,  $quantization = 3$  (D), imagem modificada 4 bits,  $quantization = 4$  (E), imagem modificada 5 bits,  $quantization = 5$  (F), imagem modificada 6 bits,  $quantization = 6$  (G), imagem modificada 7 bits,  $quantization = 7$  (H), imagem modificada 8 bits,  $quantization = 8$  (I) para a imagem baboon.ppm.



Figura 24 - Imagem original airplane.ppm.

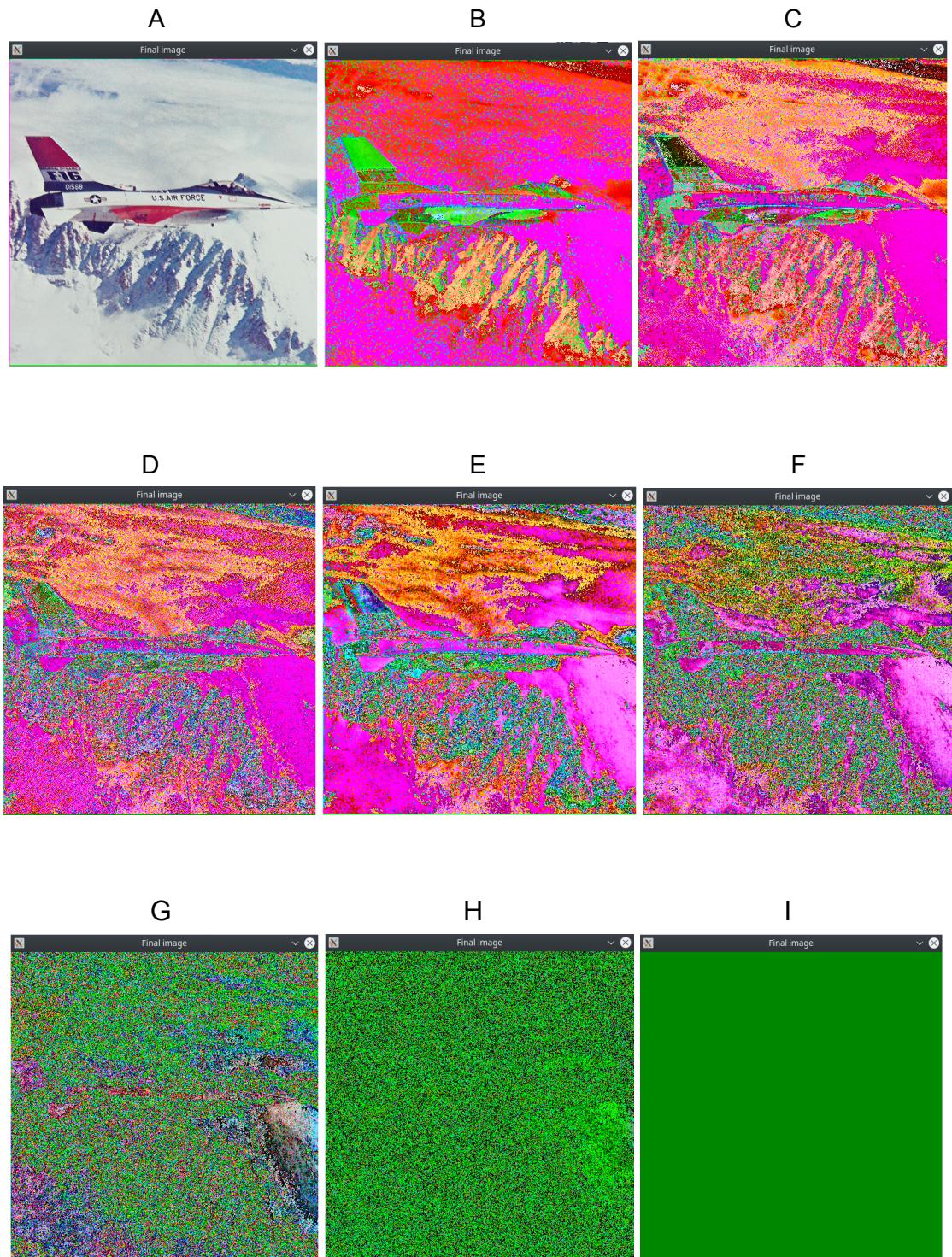


Figura 25 - Imagem modificada 0 bits,  $quantization = 0$  (A), imagem modificada 1,  $quantization = 1$  bit (B), imagem modificada 2 bits,  $quantization = 2$  (C), imagem modificada 3 bits,  $quantization = 3$  (D), imagem modificada 4 bits,  $quantization = 4$  (E), imagem modificada 5 bits,  $quantization = 5$  (F), imagem modificada 6 bits,  $quantization = 6$  (G), imagem modificada 7 bits,  $quantization = 7$  (H), imagem modificada 8 bits,  $quantization = 8$  (I) para a imagem airplane.ppm.

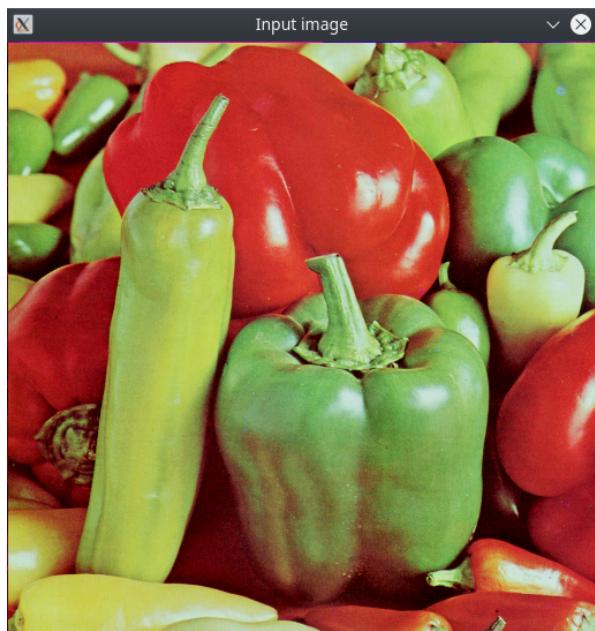
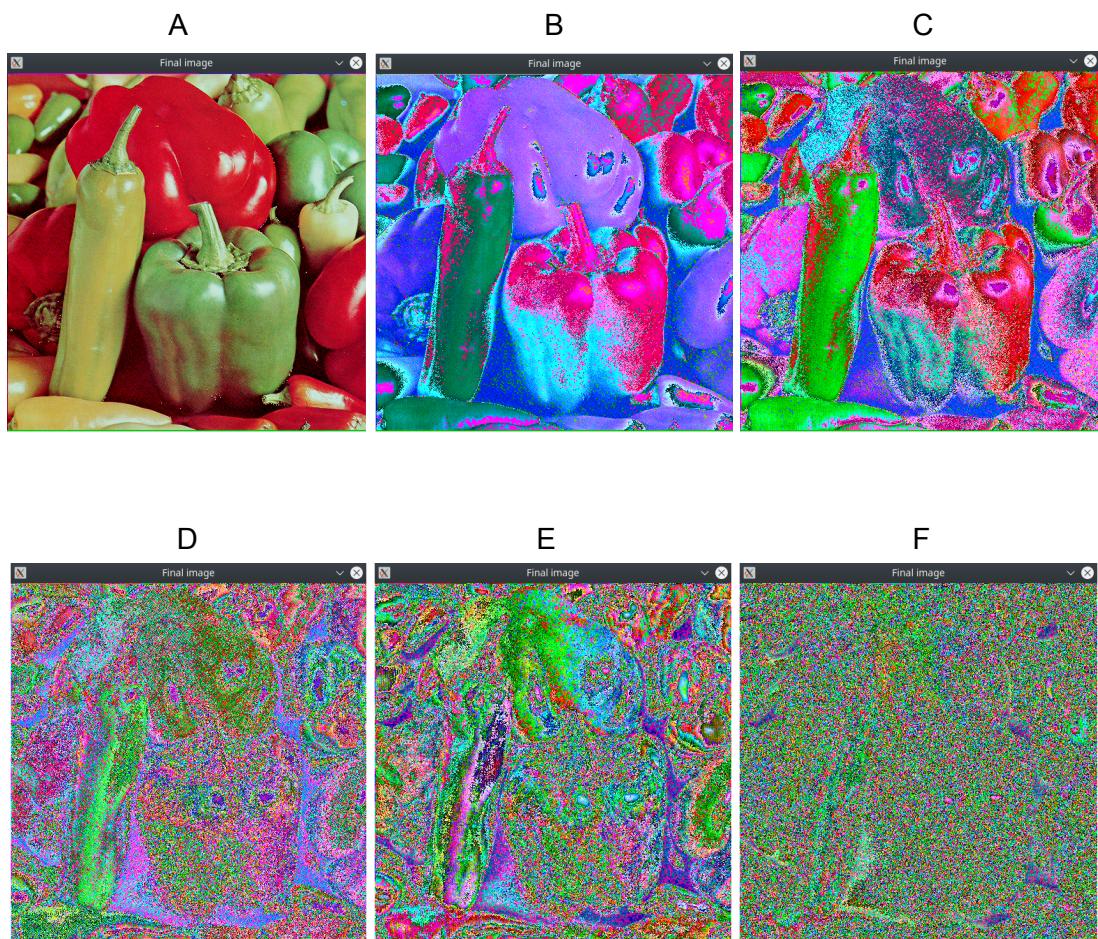


Figura 26 - Imagem original peppers.ppm.



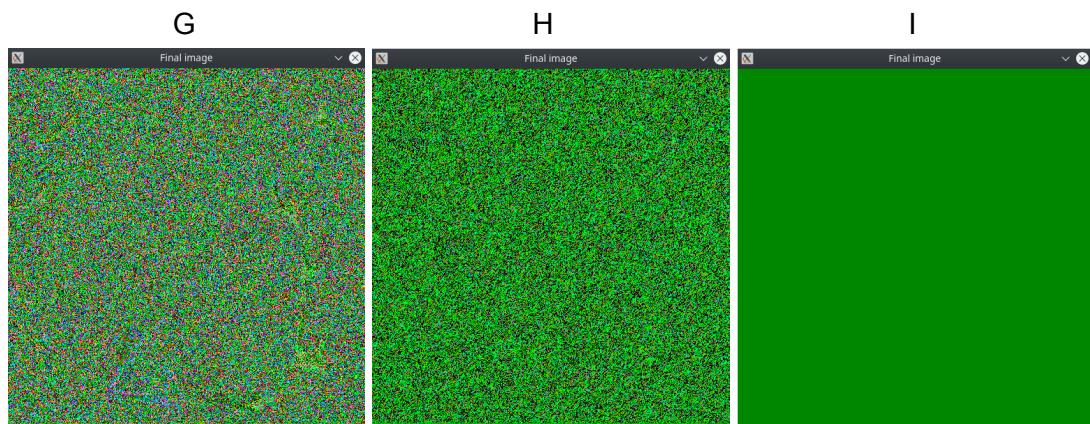


Figura 27 - Imagem modificada 0 bits,  $quantization = 0$  (A), imagem modificada 1,  $quantization = 1$  bit (B), imagem modificada 2 bits,  $quantization = 2$  (C), imagem modificada 3 bits,  $quantization = 3$  (D), imagem modificada 4 bits,  $quantization = 4$  (E), imagem modificada 5 bits,  $quantization = 5$  (F), imagem modificada 6 bits,  $quantization = 6$  (G), imagem modificada 7 bits,  $quantization = 7$  (H), imagem modificada 8 bits,  $quantization = 8$  (I) para a imagem peppers.ppm.

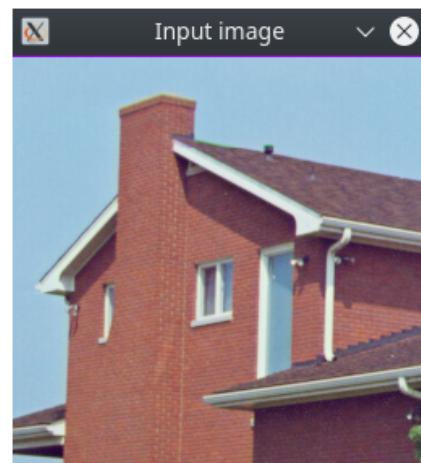
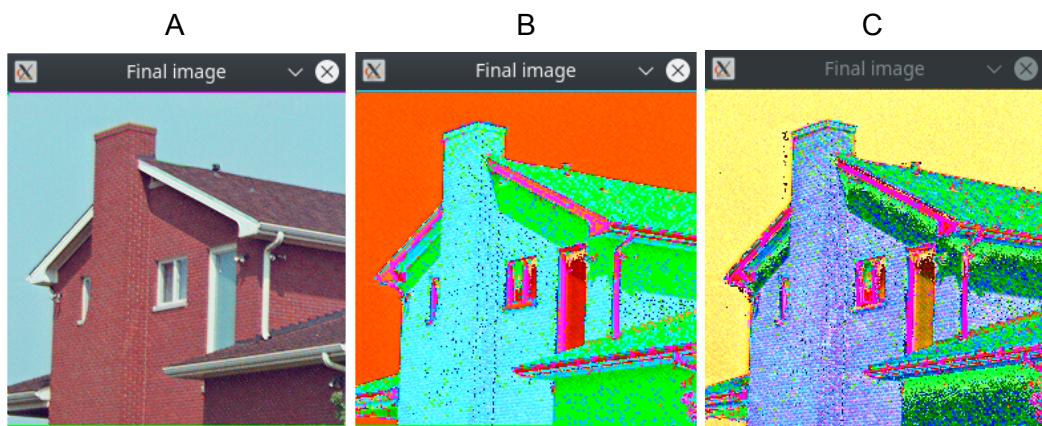


Figura 28 - Imagem original house.ppm.



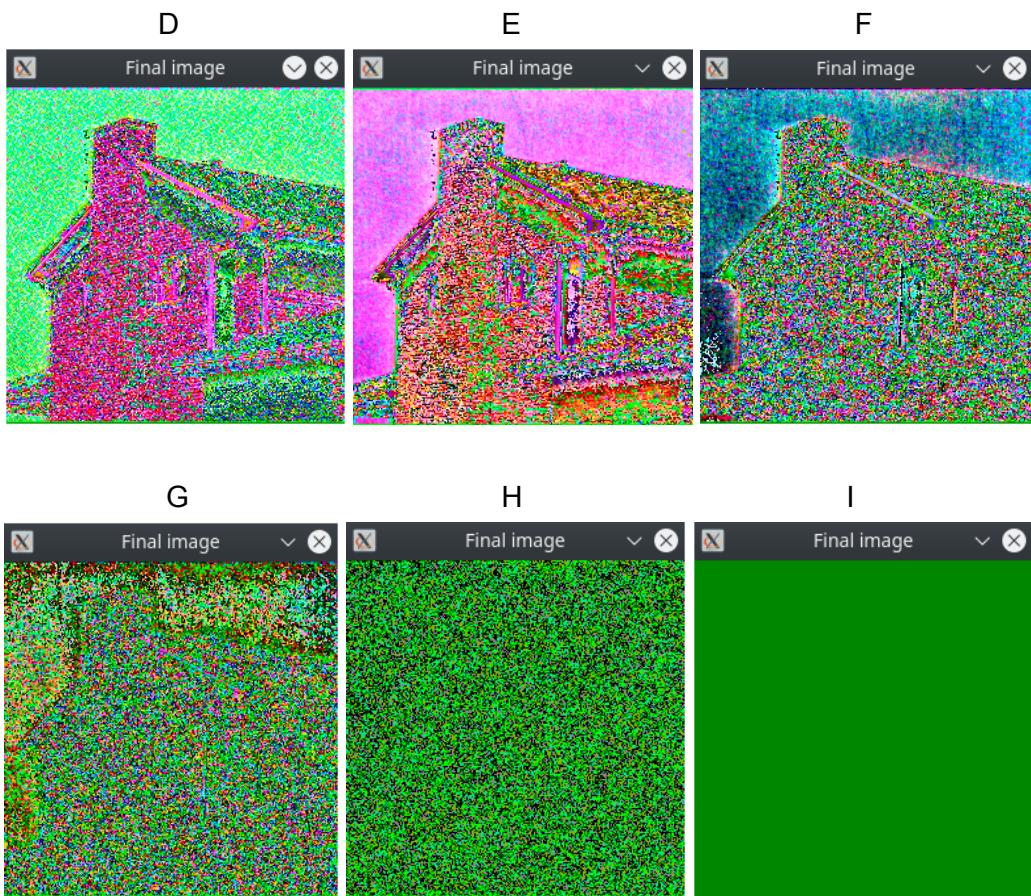


Figura 29 - Imagem modificada 0 bits,  $quantization = 0$  (A), imagem modificada 1,  $quantization = 1$  bit (B), imagem modificada 2 bits,  $quantization = 2$  (C), imagem modificada 3 bits,  $quantization = 3$  (D), imagem modificada 4 bits,  $quantization = 4$  (E), imagem modificada 5 bits,  $quantization = 5$  (F), imagem modificada 6 bits,  $quantization = 6$  (G), imagem modificada 7 bits,  $quantization = 7$  (H), imagem modificada 8 bits,  $quantization = 8$  (I) para a imagem house.ppm.

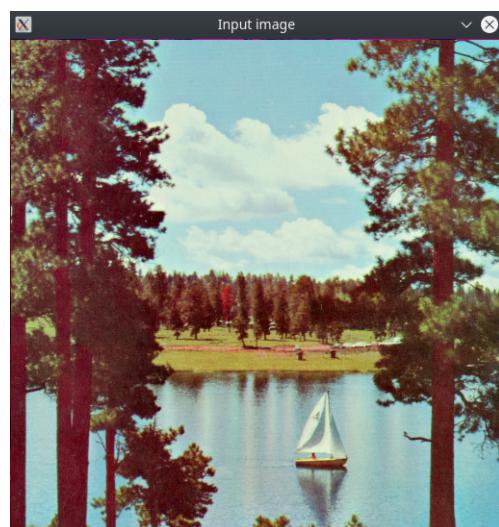


Figura 30 - Imagem original boat.ppm.

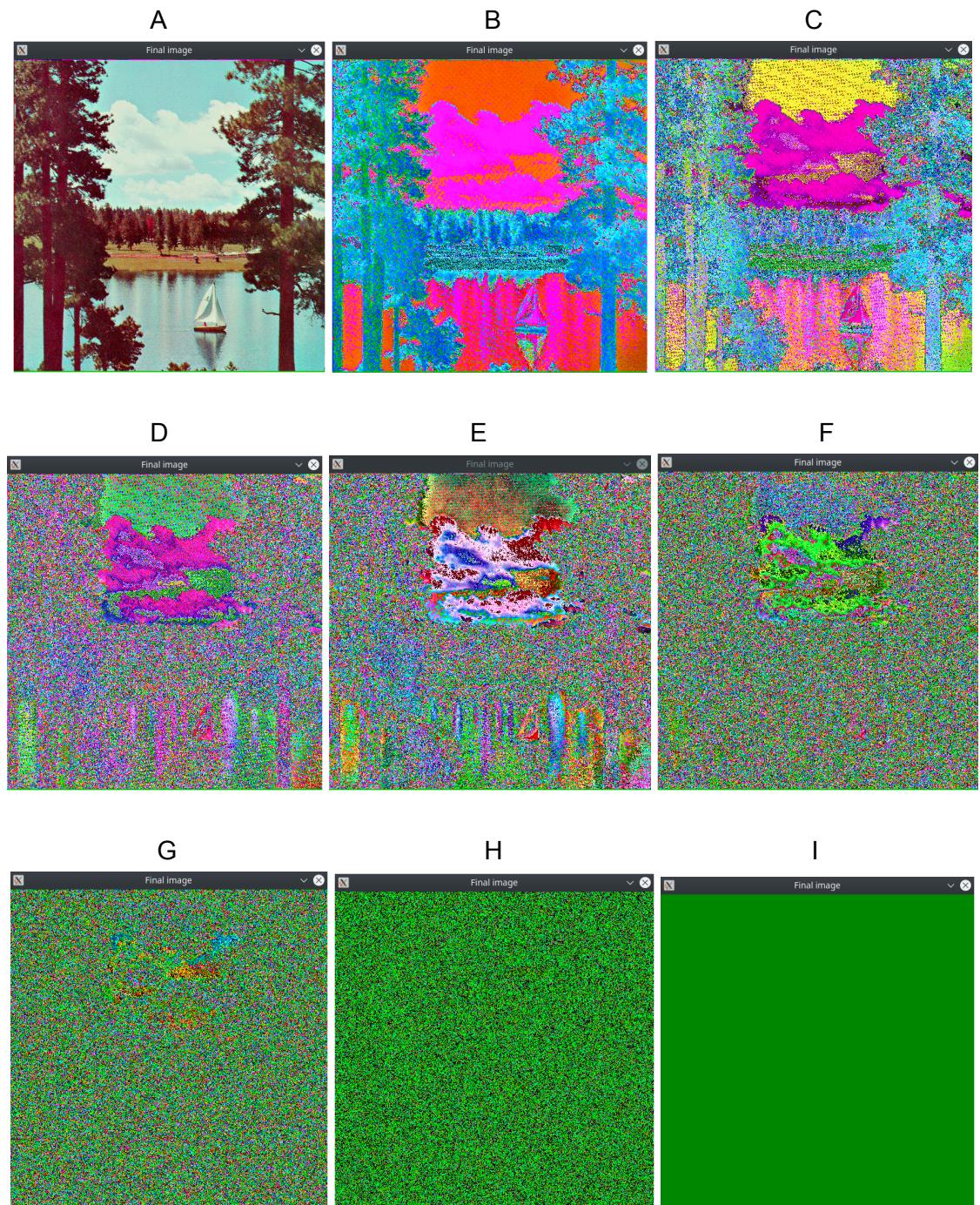


Figura 31 - Imagem modificada 0 bits,  $quantization = 0$  (A), imagem modificada 1,  $quantization = 1$  bit (B), imagem modificada 2 bits,  $quantization = 2$  (C), imagem modificada 3 bits,  $quantization = 3$  (D), imagem modificada 4 bits,  $quantization = 4$  (E), imagem modificada 5 bits,  $quantization = 5$  (F), imagem modificada 6 bits,  $quantization = 6$  (G), imagem modificada 7 bits,  $quantization = 7$  (H), imagem modificada 8 bits,  $quantization = 8$  (I) para a imagem boat.ppm.

Tal como no exercício anterior, é de notar que entre as imagens originais e as imagens com *quantization* = 0 (Figuras com A) existe uma mudança leve na cor. Isto acontece pois ao transformar as imagens de RGB para YUV e de YUV para RGB nas equações (.equacoes em cima..) acabou por haver arredondamentos perdendo-se assim algumas casas decimais que influenciam a cor no final.

Quanto ao bônus deste problema, foi pedido para implementar outra versão do codec com perdas, baseada na transformação e codificação dos resíduos de previsão, utilizando o DCT (Discrete Cosine Transform) como no JPEG, e quantização dos coeficientes. Os valores quantizados são codificados por entropia usando códigos Golomb.

Para introduzir DCT, foi usada a fórmula 28 para cada uma das componentes Y, U e V antes do cálculo do residual.

$$G_{ij} = \frac{1}{\sqrt{2N}} C_i C_j \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} p_{xy} \cos\left(\frac{(2y+1)j\pi}{2c}\right) \cos\left(\frac{(2x+1)i\pi}{2L}\right), \quad 0 \leq i, j \leq n - 1 \quad (28)$$

onde, c é o número de colunas que existem na matriz, L é o número de linhas existem na matriz, N é o número de amostras numa matriz (c\*L),  $p_{xy}$  é o valor do pixel na posição x:y da matriz  $p$  e  $C_i, C_j = \frac{1}{\sqrt{n}}$  quando  $i$  ou  $j = 0$  ou  $C_i, C_j = \sqrt{2/n}$  quando  $i$  ou  $j > 0$ .

Para a recuperação dos dados transformados usou-se a transformação inversa (IDCT) após o cálculo do valor do pixel depois da descodificação através da fórmula 29.

$$p_{xy} = \frac{1}{4} \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} C_i C_j G_{ij} \cos\left(\frac{(2x+1)i\pi}{2L}\right) \cos\left(\frac{(2y+1)j\pi}{2c}\right) \quad (29)$$

onde, mais uma vez, c é o número de colunas que existem na matriz, L é o número de linhas que existem na matriz, N é o número de amostras numa matriz (c\*L),  $G_{ij}$  é o valor do DCT na posição i:j da matriz  $G$  e  $C_i, C_j = \frac{1}{\sqrt{n}}$  quando  $i$  ou  $j = 0$  ou  $C_i, C_j = \sqrt{2/n}$  quando  $i$  ou  $j > 0$ .

Feito isto, ao correr o programa, é possível observar que demora muito tempo a correr por causa de muitos ciclos for dentro de ciclos for, o que faz com que o programa fique com uma ordem de complexidade elevada e por isso lento. Assim, não se sabe se este programa está certo ou não pois o programa fica muito tempo a correr.

Para executar o programa, a imagem que o programa lê deve estar dentro de uma pasta chamada imagensPPM. Esta pasta deve estar na mesma pasta que o programa. Como se trabalhou com a biblioteca *OpenCV*, com o Golomb e com a bit-stream desenvolvidos anteriormente, ao compilar o código tem que se, para além de abrir um terminal na pasta do programa, correr o comando `g++ ex2b_parteC.cpp ./bit_stream/bit_stream.cpp ./Golomb/golomb.cpp -o ex2b `pkg-config --cflags --libs opencv``.

De modo a executar o programa utilizou-se o comando `./ex2b <input image> <output filename> <quantization>` (por ex `./parteC lena.ppm copy.ppm 2`).

É importante referir que o ficheiro de texto onde será escrito os códigos de Golomb não deverá ser o mesmo de imagem para imagem.

Contribuição dos autores:

Como todos trabalhamos igualmente decidimos atribuir 33% a cada elemento do grupo.