



# Informação e Codificação

## Projeto 1

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

Link do repositório - <https://github.com/AlexOliZ/IC.git>

---

### Parte B

#### Problema 2:

Executar o programa: Para executar o programa, tem de existir um ficheiro de texto com o conteúdo a copiar (ficheiro para leitura) na mesma pasta que o programa. Após abrir um terminal nessa mesma pasta temos de compilar o programa. Para isso executamos `g++ ex2.cpp -o ex2`. De modo a executar o programa escrevemos `./ex2 <nome ficheiro de leitura> <nome do ficheiro de escrita>` (por exemplo: `./ex2 read.txt write.txt`).

Neste exercício temos que ler do ficheiro passado como argumento de linhas de comando (`ifstream ifs(argv[1])`) e escrever noutro ficheiro também este passado como argumento (`ofstream ofs(argv[2])`) para isso lemos carácter a carácter do ficheiro com a função `get(x)`, escrevemos no ficheiro de saída (`ofs<<x`) e no final fechamos o ficheiro criado (`ofs.close()`).

Na Figura 1 podemos observar que os resultados foram os esperados, pois o ficheiro de escrita é igual ao de leitura.

read.txt	write.txt
1 ola	1 ola
2 5	2 5
3 10	3 10
4 20	4 20

Figura 1 - Resultados do problema 2.

### Problema 3:

Executar o programa: Para a execução deste programa, é preciso que o ficheiro áudio a ser lido esteja na pasta “Wav files-20211025”. Para compilar e correr o programa basta escrever o comando `g++ -o ex3 ex3.cpp -lsndfile` e em seguida `./ex3 <ficheiro áudio entrada> <nome ficheiro saída>` num terminal aberto na pasta do programa.

Neste problema usamos a biblioteca *libsndfile* para criar uma cópia do ficheiro original. De modo a obter isso lemos os valores das samples para um buffer através da função `sf_read_int` e em seguida escrevemos com a função `sf_write_int` os valores lidos no novo ficheiro. Estas funções têm como parâmetros os ficheiros para ler ou escrever, o buffer com a informação para ser lida ou escrita e o número de itens. Este número é calculado através da multiplicação do número de frames pelo número de canais, que neste caso é 2.

### Problema 4:

Executar o programa: Neste exercício 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 trabalhamos com a biblioteca *OpenCV*, ao compilar o código temos que, para além de abrir um terminal na pasta do programa, temos de correr o comando `g++ ex4.cpp -o ex4 -std=c++11 `pkg-config --cflags --libs opencv``. De modo a executar o programa utilizou-se o comando `./ex4 <nome da imagem> <nome da imagem cópia>` (por exemplo: `./ex4 lena.ppm lena_copy.ppm`).

Neste exercício, temos que copiar pixel a pixel uma imagem. Percorremos a imagem original pelas suas linhas e para cada linha percorremos todas as colunas (um ciclo *for* dentro de outro ciclo *for*) e copiamos pixel a pixel a imagem original (`output_image.at<Vec3b>(i,j) = input_image.at<Vec3b>(i,j)`). No final é só mostrar e guardar a imagem de saída através das funções do *OpenCV* `imshow()` e `imwrite()`, respetivamente.

Na Figura 2 podemos observar que conseguimos copiar uma imagem pixel a pixel.

Este programa foi testado em todas as imagens de teste, mas por uma questão de simplicidade resolvemos apresentar só o teste feito para a imagem *lena.ppm*.

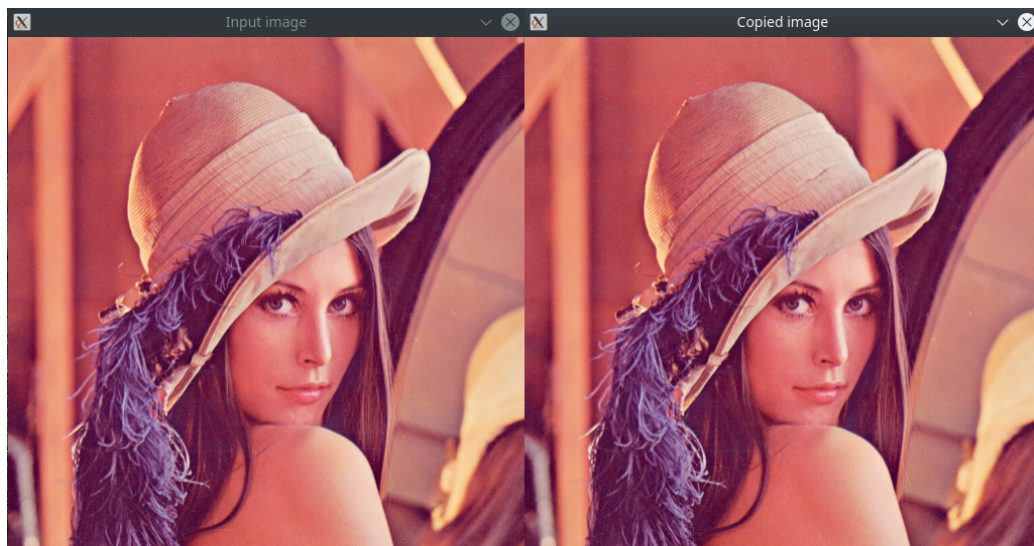


Figura 2 - Imagem de teste e respectivo resultado.

## Parte C

### Problema 5:

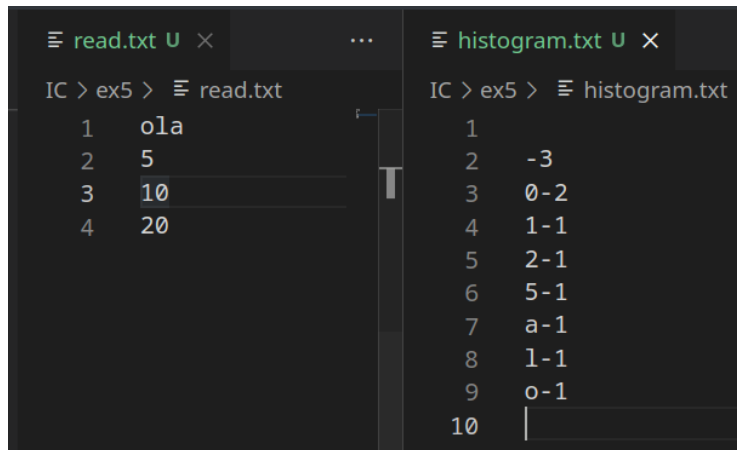
Executar o programa: Para executar o programa, o ficheiro a analisar, no nosso caso os Lusíadas (lusiadas.epub), tem que estar na mesma pasta que o programa. Após abrir um terminal nessa mesma pasta temos de compilar o programa, executando `g++ ex5.cpp -o ex5`. Para executar o programa escrevemos `./ex5 <nome do ficheiro a analisar>` (por exemplo: `./ex5 lusiadas.epub`). O programa cria um ficheiro chamado `histograma.txt` onde pode ser visto o histograma calculado.

Neste programa foi feito um mapa onde vai agrupar a letra e o número de vezes que essa letra aparece nos Lusíadas (`map <char, int> histogram`) para isso fez-se um ciclo `while` para ler caractere a caractere do ficheiro (com a função `get(x)`) e contar o número de vezes que determinado caractere aparece no ficheiro. No final é criado um ficheiro chamado `histograma.txt` com os resultados, onde a primeira coluna são os caracteres do ficheiro e a segunda coluna é o número de vezes que cada caractere aparece no ficheiro, como podemos ver na Figura 3.

0-774	I-710	V-855	i-738	²-778
1-775	J-704	W-743	¢-782	³-771
2-808	K-762	X-741	£-760	Ž-743
3-873	L-788	Y-798	€-780	µ-805
4-783	M-818	Z-789	¥-754	¶-772
5-809	N-784	[ -765	§-794	·-768
6-788	O-777	\-729	§-855	ž-785
7-777	P-722	] -760	§-759	Ž-785
8-735	Q-779	^-781	©-807	¹-818
9-807	R-703	-739	ª-792	º-747
: -744	S-873	`-704	«-826	»-756
; -753	T-834	a-795	¬-802	ƒ-776
<-768	U-779	b-806	-842	€-792
			®-773	Ÿ-732
			-812	ž-766
			°-774	

Figura 3 - Parte do histograma.txt.

Para além do ficheiro dos Lusíadas, este programa também foi testado num ficheiro que nós criamos e como podemos ver (Figura 4) obteve-se os resultados esperados, onde o primeiro -3 é a contagem dos `\n` do ficheiro `read`.



```
IC > ex5 > read.txt
1 ola
2 5
3 10
4 20

IC > ex5 > histogram.txt
1
2 -3
3 0-2
4 1-1
5 2-1
6 5-1
7 a-1
8 1-1
9 o-1
10 |
```

Figura 4 - Teste executado com o nosso ficheiro.

Infelizmente, como podemos observar no ficheiro histogram.txt alguns dos caracteres especiais não são visíveis no ficheiro.

#### Problema 6:

Executar o programa: Neste exercício o ficheiro de áudio que o programa lê deve estar dentro de uma pasta chamada “Wav files-20211025”. Esta pasta deve estar na mesma pasta que o programa. Como trabalhamos com a biblioteca *libsndfile*, ao compilar o código temos que, para além de abrir um terminal na pasta do programa, temos de correr o comando `g++ ex6.cpp -lsndfile`. De modo a executar o programa utilizou-se o comando `./a.out <nome do áudio>` (por exemplo: `./a.out sample0N.wav`, onde N pertence ao intervalo [1, 6] ).

Neste problema foi pedido para calcularmos o histograma de um ficheiro de áudio e a sua entropia para isso usamos a livreria *libsndfile*. Deste modo, obtemos os valores do ficheiro .wav e calculamos o histograma (cada um dos histogramas é escrito num ficheiro .txt. Esta livreria permite escolher entre várias funções para ler o ficheiro, podendo escolher entre `int`, `double` ou `float`. Também existe a opção de escolher entre `readf` (read frames) ou `read` (read items). A diferença entre estas funções é nos argumentos necessário para a função, na função `read` é preciso especificar o número de objetos e na função `readf` precisa do número de frames (número de objetos \* número de canais).

Para calcular o histograma usamos um mapa para armazenar o número de vezes que cada valor aparece no ficheiro. Para calcular a entropia usamos um histograma diferente para cada uma das entropias e aplicamos a fórmula.

$$H(X) = - \sum_{i=1}^n P(x_i) \log_b P(x_i) \quad (1)$$

onde  $P(x)$  é a probabilidade de ocorrência desse símbolo.

Nota: Como os valores do ficheiro são de 16 bits mudamos a base do logaritmo para 16.

No final o cada um dos histogramas são escritos no seu ficheiro chamado histograma.txt e a entropia é impressa no terminal.

Este programa foi testado em todos os áudios de teste, mas por uma questão de simplicidade resolvemos apresentar só o teste feito para o áudio sample01.wav.

```
frames=1294188
samplerate=44100
channels=2
Read 2588376 items
entropy c1 -> 3.472612
entropy c2 -> 3.476768
entropy avg -> 3.724116
entropy -> 3.477144
```

Figura 5 - Entropia calculada para um áudio.

## Problema 7:

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. Tal como no problema 4 trabalhamos com a biblioteca *OpenCV*. Ao compilar o código temos que abrir um terminal na pasta do programa, correr o comando `g++ ex7.cpp -o ex7 -std=c++11 `pkg-config --cflags --libs opencv`` para compilar e para executar o comando `./ex7 <nome da imagem>` (por exemplo: `./ex7 lena.ppm`).

Neste exercício tem que ser fornecido uma imagem ao programa para calcular os histogramas das suas componentes B, G, R, a escala de preto e branco (grayscale) e ainda a entropia para cada uma das componentes.

Para isto, na imagem colorida separamos as suas componentes através da função `split()` do *OpenCV*. Assim, podemos calcular o histograma (função `calcHist()`) para cada uma das suas componentes. Antes de desenharmos o histograma, temos que usar a função `normalize()` para que os seus valores estejam dentro de uma

gama indicada pelos parâmetros introduzidos. De seguida, através da função `line()`, já podemos criar a linha do histograma para o seu tamanho (256).

Para o cálculo do histograma da imagem a preto e branco, temos que converter a imagem colorida a preto e branco usando a função `cvtColor()` (sendo o seu terceiro parâmetro `COLOR_BGR2GRAY`). Feito isto, já podemos calcular o histograma (função `calcHist()`) e encontrar o mínimo e o máximo da matriz do histograma da imagem a preto e branco (função `minmaxLoc()`). Finalmente podemos criar a linha do histograma para o seu tamanho (256) através da função `line()`.

Quanto ao cálculo da entropia, criamos uma função que, com a variável do histograma, a variável do tamanho da imagem e a variável do tamanho do histograma como argumentos, percorre as linhas do histograma e para cada linha lê a coluna com os resultados do histograma. Aplicando a fórmula seguinte

$$H(X) = - \sum_{i=1}^n P(xi) \log_b P(xi) \quad (2)$$

conseguimos obter a entropia de cada componente das imagens, onde  $n$  é o número de níveis de determinada cor e  $P(x)$  é a probabilidade de um pixel ter nível de determinada cor.

No final a imagem a cores e a imagem preto e branco tal como os seus histogramas são mostrados através da função `imshow()` e a entropia é impressa no terminal.

Este programa foi testado em todas as imagens de teste, mas por uma questão de simplicidade resolvemos apresentar só o teste feito para a imagem `lena.ppm`.

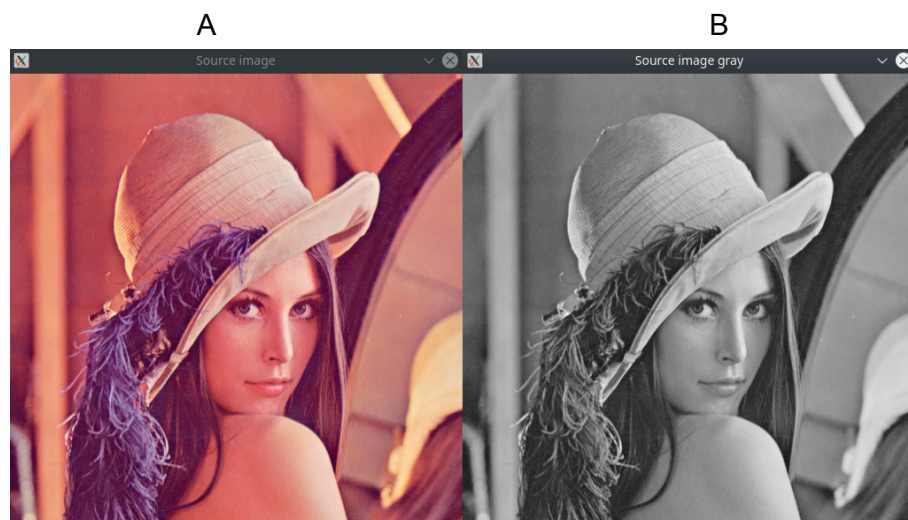


Figura 5 - Imagem `lena.ppm` exibida a cores e a preto e branco.

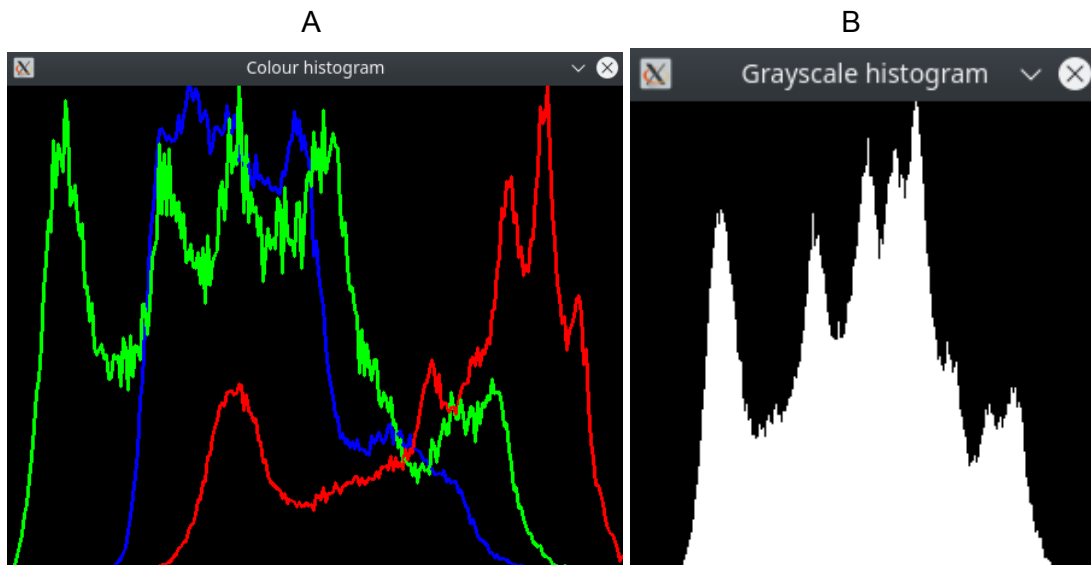


Figura 6 - Histograma de cores e a preto e branco da imagem lena.ppm.

```
entropy B: 4.83015
entropy G: 5.26378
entropy R: 5.02747
entropy grayscale: 5.16052
```

Figura 7 - Entropia para cada componente da imagem lena.ppm.

Através do histograma da imagem de cores da Figura 6-A concluímos que a imagem lena.ppm tem bastante cor verde e pouca cor vermelha tendo em conta o que o olho humano observa na imagem (o olho humano vê a imagem mais parecida com a cor vermelha do que com a verde). Quanto à cor azul o olho humano vê mais azul na parte da pena do chapéu e confirmamos isso com o histograma.

No histograma da imagem a preto e branco da Figura 6-B podemos ver que quando os tons de cinza ficam mais escuros, existe menos branco no histograma tal como o esperado.

Quanto à entropia, esta é utilizada na análise e avaliação da imagem em dados quantitativos, ou seja, mede o nível de intensidade de cada pixel (Figura 7).



## Parte D

### Problema 8:

Executar o programa: Para a execução deste programa, é preciso que o ficheiro áudio a ser lido esteja na pasta “Wav files-20211025”. Para compilar e correr o programa basta escrever o comando `g++ -o ex8 ex8.cpp -lsndfile` e em seguida `./ex3 <ficheiro áudio entrada>`.

Neste programa era pretendido reduzir o número de bits para codificar cada sample de áudio e para alcançar isso foi aplicada uma fórmula de quantização a cada sample. A fórmula é dada por  $Q(x) = \Delta * \text{floor}(x/\Delta + 1/2)$ , em que o  $\Delta$  (step de quantização) é calculado em função do número de bits que se pretende transformar o sinal:  $\Delta = (t-s)/(\lambda-1)$ . Sendo  $t=1$  e  $s=-1$  o valor máximo e mínimo do sinal, respetivamente, e  $\lambda = 2^4$  (4 bits finais), o step é igual a 0,1333 aproximadamente.

O resto do programa é igual ao problema 3 já que se resume a copiar as samples de um ficheiro áudio e guardar as samples numa cópia. A única diferença é que neste problema o conteúdo do buffer lido é alterado pelos cálculos previamente descritos.

### Problema 9:

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. Tal como no problema 4 e 7 trabalhamos com a biblioteca `OpenCV`. Ao compilar o código temos que abrir um terminal na pasta do programa, correr o comando `g++ ex9.cpp -o ex9 -std=c++11 `pkg-config --cflags --libs opencv`` para compilar e para executar o comando `./ex9 <input filename> <output filename>` (por exemplo: `./ex9 lena.ppm lena7.ppm`).

Este exercício é semelhante ao anterior, mas aplicado à imagem. Neste caso, a imagem tem um tamanho de 256 o que significa que existem 8 bits para representar cada pixel. Foi-nos pedido para reduzir o número de bits utilizados em cada pixel de uma imagem. Deste modo, percorremos a imagem pixel a pixel tal como no problema 4 e 7 e alteramos a informação de cada pixel. Para isso percorremos a imagem original pelas suas linhas e para cada linha percorremos todas as colunas (um ciclo *for* dentro de outro ciclo *for*) e alteramos pixel a pixel a imagem original. Ou seja, reduzimos a intensidade da cor (fazendo  $n$  shifts à direita) e descartamos a informação desses  $n$  bits reduzindo os níveis de cada pixel

(fazendo  $n$  shifts à esquerda), onde  $n$  pertence ao intervalo  $[1, 8]$ . Reduzindo assim o tamanho de cada pixel na imagem.

Feito isto guardamos a imagem criada através da função `imwrite()`.

Este programa foi testado em todas as imagens de teste, e para cada uma delas para todos os valores possíveis de  $n$ , mas por uma questão de simplicidade resolvemos apresentar só o teste feito para a imagem `lena.ppm` com  $n$  igual a 2, 5 e 7.



Figura 8 - Imagem original (A), imagem modificada 2 bits (B), imagem modificada 5 bits (C) e imagem modificada 7 bits (D).

Sabe-se que se  $n$  for igual a 1 descarta-se só um bit de informação, a imagem final parece igual a imagem inicial e se for igual a 8 descartamos todos os bits de informação (imagem fica preta). Ou seja, quanto maior o número de bits descartados mais escura ficará a imagem.

Tal como podemos ver na Figura 8 a imagem com  $n$  igual (B) a 2 tem menos 2 bits de informação do que a original (A), por isso não se consegue ver muitas diferenças entre as duas. Já a imagem com  $n$  igual a 5 (C) tem menos 5 bits de informação, notando-se que está mais escura que a original. Por último a imagem com menos 7 bits que a imagem original (D) tem bastante menos informação logo, como podemos ver esta é quase totalmente preta.

## Problema 10:

Executar o programa: Neste exercício o ficheiro de áudio que o programa lê deve estar dentro de uma pasta chamada Wav files-20211025. Esta pasta deve estar na mesma pasta que o programa. Como trabalhamos com a biblioteca *libsndfile*, ao compilar o código temos que, para além de abrir um terminal na pasta do programa, temos de correr o comando `g++ ex10.cpp -lsndfile`. De modo a executar o programa utilizou-se o comando `./a.out <nome do áudio> (por exemplo: ./a.out sample0N.wav <n_bits>, onde N pertence ao intervalo [1, 6] )`.

Após reduzir o número de bits do ficheiro de som calculamos a soma dos valores obtidos no ficheiro original (equação 3) e a soma do quadrado da diferença entre o ficheiro original e o ficheiro comprimido (equação 4), a maior diferença vai ser armazenada numa variável para determinar o maior erro absoluto, após determinar esses valores podemos calcular o SNR com logaritmo de base 10 (equação 5).

$$Ex = \sum_N x(n)^2 \quad (3)$$

$$Er = \sum_N (x(n) - xr(n))^2 \quad (4)$$

$$SNR = 10 \times \log_{10}\left(\frac{Ex}{Er}\right) \quad (5)$$

Como podemos verificar nos testes efetuados no ficheiro `sample01.wav`, quanto maior a quantização dos bits maior a precisão da cópia do ficheiro de áudio sendo 0 se for uma cópia bit a bit do ficheiro (Figura 9).

A	B
<pre>sample01.wav SUM: 36806.118695 SUM_R: 30875.306043 SNR: 1.757074 MAX ABS ERROR: 0.062500</pre>	<pre>sample01.wav SUM: 36806.118695 SUM_R: 3347.533254 SNR: 23.974404 MAX ABS ERROR: 0.003906</pre>
C	D
<pre>sample01.wav SUM: 36806.118695 SUM_R: 13.162761 SNR: 79.360277 MAX ABS ERROR: 0.000015</pre>	<pre>sample01.wav SUM: 36806.118695 SUM_R: 0.000000 SNR: inf MAX ABS ERROR: 0.000000</pre>

Figura 9 - Cálculo de SNR e erro máximo absoluto para áudios de 2 bits (A) para 4 bits (B), para 8 bits (C) e para 16 bits (D)

## Problema 11:

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. Ao compilar o código temos que abrir um terminal na pasta do programa, correr o comando `g++ ex11.cpp -o ex11 -std=c++11 `pkg-config --cflags --libs opencv`` para compilar e para executar o comando `./ex11 <input filename> <output filename>` (por exemplo: `./ex11 lena.ppm lena7.ppm`).

Para o cálculo do SNR e do máximo erro absoluto foi usado como base o código do problema 9. A partir deste foi calculado o SNR através das fórmulas descritas no problema anterior em que é subtraído o valor original ao transformado a cada iteração. Neste problema como cada pixel tem 3 canais, primeiramente foi preciso somar o conteúdo de cada canal para podermos obter um número relativo a um pixel de forma a poder calcular os somatórios e posteriormente o SNR. O valor máximo de erro foi verificado a cada iteração subtraindo o valor do pixel original ao modificado e caso este valor fosse o maior ao anterior máximo, este era substituído.

Como é possível observar (Figura 10), quanto maior for a modificação menor é o SNR, enquanto que o máximo erro absoluto é maior se mudarmos mais bits.

A	B
SNR: 10 MAX ABSOLUTE ERROR: 252	SNR: 41 MAX ABSOLUTE ERROR: 93
C	D
SNR: 108 MAX ABSOLUTE ERROR: 3	SNR: 71 MAX ABSOLUTE ERROR: 21

Figura 10 - Cálculo de SNR e erro máximo absoluto para imagens com redução de 7 bits (A), 5 bits (B), 3 bits (C) e 1 bit (D).

## Contribuição dos autores:

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