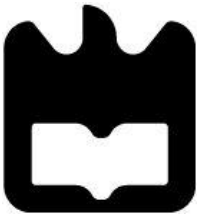
IC - Assignment II

Universidade de Aveiro

Departamento de Eletrónica, Telecomunicações e Informática

Mestrado em Engenharia Informática



Diogo Carvalho nº92969 1/3

Hugo Ferreira nº93093 1/3

Rafael Baptista nº93367 1/3

Repositório: https://github.com/rafaelbaptista13/IC

04 de Dezembro de 2022

Índice

1. Introdução

O presente relatório pretende descrever o trabalho realizado na execução do segundo projeto proposto na unidade curricular de Informação e Codificação, cuja estrutura do enunciado se encontrava dividida em 5 partes distintas.

A primeira fase do projeto era constituída por 2 exercícios (1 deles dividido em 4 alíneas) relacionados com a aplicação de operações de manipulação de imagens, tais como cópia, rotação, variação do brilho, entre outras, pixel por pixel utilizando a livraria de C++ OpenCV.

Na segunda parte, o enunciado é formado por apenas 1 exercício que solicita a implementação de uma classe que representa códigos de Golomb e que, deve possuir métodos para a codificação de inteiros e descodificação de uma String de código.

A terceira parte do projeto, à semelhança da primeira, é também formada por 2 exercícios. Estes exercícios estão relacionados com a utilização da classe construída na fase anterior para a implementação de um codec de áudio sem perda de informação que seja baseado em códigos de Golomb dos residuais de previsão que resultam da aplicação de codificação preditiva às amostras de um sinal sonoro (áudio).

A quarta componente do projeto contém um exercício único que é semelhante ao descrito na terceira parte do projeto, mas desta vez aplicado a imagens invés de áudios.

Por fim, a última parte do projeto consiste na escrita deste mesmo documento cujo objetivo é explicar as decisões efetuadas e resultados obtidos ao longo do projeto.

2. Metodologia

Para a realização deste projeto, utilizamos a plataforma Github como repositório de código compartilhado. A estrutura do nosso repositório é constituída por duas pastas distintas. A estrutura do nosso repositório é constituída por três pastas distintas. A pasta “opencv-src” que contém todo o código desenvolvido ao longo do projeto, uma pasta “audioexamples” que contém os ficheiros de áudio utilizados para testar os diferentes programas que foram desenvolvidos que utilizavam ficheiros de áudio, e uma pasta “pics-examples” que contém os ficheiros de imagens utilizadas para testar os programas que foram desenvolvidos que utilizavam imagens.

No que diz respeito à comunicação entre os elementos do grupo, utilizamos as plataformas Discord e Messenger para efetuar a discussão dos pontos críticos do nosso projeto, facilitando assim a cooperação à distância na realização do mesmo.

Relativamente ao código escrito, o mesmo foi desenvolvido na linguagem de programação C++. A estrutura do código é composta por:

- Programa *cp_image.cpp* que utiliza a livraria OpenCV para aplicar operações de manipulação de imagens, pixel por pixel.
- Programa *audio_encoder.cpp* que faz uso das classes BitStream e GolombCode definidas nos ficheiros *BitStream.h e GolombCode.h, respetivamente,* para criar um codificador de áudio que possua a opção de utilização com e sem perda de informação baseado na aplicação de códigos de Golomb e codificação preditiva.
- Programa *audio_decoder.cpp* que faz uso das classes BitStream e GolombCode definidas nos ficheiros *BitStream.h e GolombCode.h, respetivamente,* para criar um decodificador de áudio que permita obter o áudio original a partir de um áudio codificado pelo programa *audio_codec.cpp*.
- Programa *image_encoder.cpp* que faz uso das classes BitStream e GolombCode definidas nos ficheiros *BitStream.h e GolombCode.h, respetivamente,* para criar um codificador de imagem com funcionamento semelhante ao codificador de áudio desenvolvido, mas neste caso aplicado a imagens.
- Programa *image_decoder.cpp* que faz uso das classes BitStream e GolombCode definidas nos ficheiros *BitStream.h e GolombCode.h, respetivamente,* para criar um decodificador de imagem que permita obter a imagem original a partir de uma imagem codificada pelo programa *image_encoder.cpp*.

3. Implementação e Modo Utilização Programas

Nesta seção do nosso relatório iremos descrever a implementação que efetuamos para os nossos programas bem como analisar e justificar as decisões tomadas durante esta mesma fase de implementação.

3.1. cp_image.cpp

Vamos iniciar por descrever a implementação do programa desenvolvido para dar resposta à primeira fase do projeto (cp_image.cpp). O mesmo tem como objetivo aplicar diversas operações de manipulação de imagens (que iremos descrever mais abaixo), pixel por pixel, através da utilização da livraria OpenCV.

Para executar este programa, é necessário especificar o modo de utilização do programa (operação a efetuar), o caminho para a imagem de input que irá ser manipulada bem como o nome do ficheiro de output que irá representar a manipulação. Dependendo do modo de utilização escolhido, isto é, operação a realizar, poderá ser necessária a passagem de dois parâmetros extra. O parâmetro -r indica os graus de rotação a aplicar à imagem (valor por defeito é de 90) e o parâmetro -i indica o fator de intensidade a aplicar na variação da intensidade de luz da imagem (valor por defeito é de 2).

```
                Usage:
./cp_image [-r degree (def 90)]
           [-i intensity_factor (def 2)]
           <mode> <input_img> <output_img>
```

Relativamente ao mode, os seguintes valores são permitidos:

- *COPY* : Indica a realização da operação de cópia da imagem.
- *ROTATE* : Indica a realização da operação de rotação. O parâmetro -r é obrigatório ser fornecido e deve ser múltiplo de 90. Rotações positivas e negativas são ambas suportadas.
- *NEGATIVE*: Indica a realização da operação de criação da versão negativa da imagem.
- *MIRROR_VERTICAL:* Indica a realização da operação de espelhamento vertical da imagem.
- *MIRROR_HORIZONTAL:* Indica a realização da operação de espelhamento horizontal da imagem.
- *INTENSITY:* Indica a realização da operação de aumento ou diminuição da intensidade da imagem. O parâmetro -i é obrigatório ser fornecido e deve ter um valor superior a 0.

```
                    Usage Examples:
      ./cp_image COPY ../pics-examples/lena.ppm output.ppm
    ./cp_image NEGATIVE ../pics-examples/lena.ppm output.ppm
./cp_image MIRROR_HORIZONTAL ../pics-examples/lena.ppm output.ppm
 ./cp_image MIRROR_VERTICAL ../pics-examples/lena.ppm output.ppm
   ./cp_image -r 90 ROTATE ../pics-examples/lena.ppm output.ppm
  ./cp_image -r -90 ROTATE ../pics-examples/lena.ppm output.ppm
./cp_image -i 1.5 INTENSITY ../pics-examples/lena.ppm output.ppm
```

```
./cp_image -i 0.5 INTENSITY ../pics-examples/lena.ppm output.ppm
```

Ao executar o nosso programa, o primeiro passo a ser efetuado é a leitura e verificação dos parâmetros fornecidos pelo utilizador. Para isso, é verificada a existência e é feita a leitura da imagem de input fornecida pelo utilizador. Posteriormente é verificado o modo de utilização do programa que foi selecionado, e consoante o modo selecionado, se necessário, é verificada a passagem dos parâmetros extra necessários.

Se as verificações forem ultrapassadas, é efetuada uma chamada a uma função que difere consoante o modo utilizado por forma a que possa ser aplicada a operação selecionada. Iremos explicar cada uma das funções de seguida.

Finalmente, após a manipulação ser efetuada com sucesso, a matriz com a imagem manipulada é escrita para o ficheiro de output fornecido e o utilizador é notificado do sucesso da operação.

Prosseguindo, irá agora ser efetuada a descrição de cada uma das funções de manipulação, numa secção própria para cada função.

3.1.1. copy_pixels()

A função copy_pixels() é responsável por dar resposta ao exercício 1, onde é solicitada a operação de cópia da imagem, pixel por pixel.

Esta função deve receber dois parâmetros que são duas referências para duas variáveis do tipo Mat definido através da livraria OpenCV, e que representam matrizes dos pixels das imagens original e manipulada.

A função vai iterar a matriz originalImg que contém os pixels da imagem original e copiar os valores para a matriz manipulatedImg que contém os pixels da imagem manipulada.

3.1.2. negative_pixels()

A função negative_pixels() é responsável por dar resposta à alínea a) do exercício 2, onde é solicitada uma operação de criação da versão negativa da imagem original.

Esta função deve receber dois parâmetros que são duas referências para duas variáveis do tipo Mat definido através da livraria OpenCV, e que representam matrizes dos pixels das imagens original e manipulada.

A função vai iterar a matriz originalImg que contém os pixels da imagem original. Para cada posição da matriz, isto é, cada pixel, vai calcular os valores de cor rgb que o pixel da matriz manipulada deve ter e armazenar esses mesmos valores na matriz manipulatedImg. Esses valores, para cada cor rgb, vão ser igual a 255 - o valor da cor na imagem original.

3.1.3. mirror_horizontal_pixels()

A função mirror_horizontal_pixels() é responsável por dar resposta à alínea a) da alínea b) do exercício 2, onde é solicitada uma operação de criação da imagem espelhada horizontalmente da imagem original.

Esta função deve receber dois parâmetros que são duas referências para duas variáveis do tipo Mat definido através da livraria OpenCV, e que representam matrizes dos pixels das imagens original e manipulada.

A função vai iterar a matriz originalImg que contém os pixels da imagem original. Para cada pixel que se encontre na posição dada pelos valores de "i" e "j", sendo "i" o número da linha e "j" o número da coluna, na matriz original, o pixel vai ser colocado na posição ("i", "numero_total_colunas - j") da matriz da imagem manipulada. Ao efetuar a manipulação nas colunas, estamos a espelhar a imagem horizontalmente.

3.1.4. mirror_vertical_pixels()

A função mirror_vertical_pixels() é responsável por dar resposta à alínea b) da alínea b) do exercício 2, onde é solicitada uma operação de criação da imagem espelhada verticalmente da imagem original.

Esta função deve receber dois parâmetros que são duas referências para duas variáveis do tipo Mat definido através da livraria OpenCV, e que representam matrizes dos pixels das imagens original e manipulada.

A função vai iterar a matriz originalImg que contém os pixels da imagem original. Para cada pixel que se encontre na posição dada pelos valores de "i" e "j", sendo "i" o número da linha e "j" o número da coluna, na matriz original, o pixel vai ser colocado na posição ("numero_total_linhas - i", "j") da matriz da imagem manipulada. Ao efetuar a manipulação nas linhas, estamos a espelhar a imagem verticalmente.

3.1.5. rotate_pixels()

A função rotate_pixels() é responsável por dar resposta à alínea c) do exercício 2, onde é solicitada uma operação de rotação da imagem original por um grau que seja múltiplo de 90º.

Esta função deve receber três parâmetros, Os dois primeiros são duas referências para duas variáveis do tipo Mat definido através da livraria OpenCV, e que representam matrizes dos pixels das imagens original e manipulada. O último parâmetro deve ser um inteiro, múltiplo de 90, e representa os graus aplicados na rotação. Rotações com graus positivos e negativos são suportadas.

A função é composta por uma estrutura do tipo switch-case com quatro possibilidades, isto é, uma para rotação de 0º, outra para 90º, outra para 180º e por fim 270º. O valor da variável que representa o número de graus é previamente transformado numa rotação positiva (caso o valor fosse negativo), e posteriormente é efetuada divisão inteira por 90, visto que também são aceites rotações iguais ou superiores a 360º.

3.1.6. change_intensity_pixels()

A função change_intensity_pixels() é responsável por dar resposta à alínea d) do exercício 2, onde é solicitada uma operação de variação da intensidade de luz da imagem.

Esta função deve receber três parâmetros. Os dois primeiros são duas referências para duas variáveis do tipo Mat definido através da livraria OpenCV, e que representam matrizes dos pixels das imagens original e manipulada. O último parâmetro deve ser um

inteiro, superior a 0, e representa o fator de intensidade de luz que deve ser aplicado na alteração da intensidade da imagem.

A função vai iterar a matriz originalImg que contém os pixels da imagem original. Para cada posição da matriz, isto é, cada pixel, vai calcular os valores de cor RGB que o pixel da matriz manipulada deve ter e armazenar esses mesmos valores na matriz manipulatedImg. Esses valores, para cada cor RGB, vão ser igual ao valor na imagem original multiplicado pelo fator de intensidade. Caso este valor seja superior a 255, o valor final será de 255.

3.2. GolombCode.h

Nesta secção do relatório, devido à importância desta classe no presente trabalho, decidimos descrever a forma como a mesma foi implementada. De relembrar que a construção desta classe correspondia à segunda parte do projeto e que esta classe vai ser utilizada nos exercícios da terceira e quarta parte do projeto.

Posto isto, a classe GolombCode tem como objetivo permitir a criação de um determinado tipo de objeto que permite efetuar a codificação de números em códigos de golomb e decodificar códigos de golomb para os respetivos números.

Para criar um objeto desta classe, é necessário fornecer o valor do parâmetro “m” que vai ser utilizado na codificação de golomb.

Ao criar um objeto, dentro do construtor o parâmetro “m” vai ser armazenado numa variável de classe. Posteriormente, ainda dentro do construtor vai ser construído um array que irá conter os códigos binários truncados associados a cada possível valor de resto que pode ser obtido, consoante o parâmetro “m”. Estes valores são apenas calculados uma vez e armazenados no array para que posteriormente o acesso seja direto.

Relativamente aos métodos que a classe oferece, os mesmos encontram-se listados e descritos de seguida:

- *encode(number):* Este método recebe um número inteiro e tem como responsabilidade devolver o código de golomb associado ao número que recebe. Para construir o código, este método vai fazer uso do método encodeQuotient() para obter o código unário do quociente, vai utilizar o array formado dentro do construtor para obter o código binário truncado associado ao resto e por fim vai adicionar um bit “0” caso o número seja positivo, um bit “1” caso o número seja negativo ou então não adiciona nenhum bit extra caso o número a codificar seja igual a 0.
- *encodeQuotient(quotient):* Este método recebe um número inteiro que é o quociente que deve ser codificado. Tem como responsabilidade devolver o código unário associado ao número recebido. Para isso vai construir uma string que contém número de 0’s igual ao número recebido e no fim adiciona um bit “1” à string.
- *decode(number_code):* Este método recebe uma String que corresponde ao código de golomb de um determinado número inteiro e tem como responsabilidade retornar o número original. Este método não está a ser utilizado nos restantes programas do projeto visto que necessita de receber

como argumento a String com o código inteiro. No entanto, o mesmo não foi retirado da classe visto que poderá ser útil para implementações futuras.

- decodeWithBitstream(bitStream): Este método recebe um objeto bitStream da classe BitStream que foi implementada e descrita no projeto anterior desta mesma unidade curricular. Tem como objetivo à medida que vai lendo bit a bit um determinado ficheiro de áudio codificado com códigos de golomb, decodificar e obter o áudio original. Para isso, começa por descodificar a parte do quociente do código de golomb (este processo é concluído assim que lê o primeiro bit a "1"). Após isso, visto que os códigos binários truncados formados no construtor são livres de prefixo, vai ler bit a bit e tentar encontrar no array formado no construtor o valor de resto associado ao código que está a ler. Após decodificar o valor do resto, vai criar o número original e caso o mesmo seja diferente de 0 vai ler mais um bit para determinar o sinal.

3.3. wav_quant.h

Na terceira parte do projeto que iremos falar mais à frente, foi utilizada a classe WAVQuant do trabalho anterior para desenvolver o codificador de áudio com perda de informação.

Para isso, foi necessário efetuar algumas alterações nesta classe. Foi adicionado um dicionário de códigos que codificam cada um dos níveis de quantização e um dicionário inverso que converte os códigos no valor do nível e foram adicionados métodos para usar estes dicionários.

3.4. audio_encoder.cpp

Na terceira parte do projeto, como referido anteriormente, era requisitada a implementação de um codec de ficheiros áudio baseado na utilização dos códigos de golomb desenvolvidos na componente anterior do projeto e no conceito de codificação preditiva.

Para isso, foram implementados dois programas distintos (*audio_encoder.cpp* e *audio_decoder.cpp*). Vamos começar por descrever o programa *audio_encoder.cpp*.

O programa *audio_encoder.cpp* tinha como responsabilidade lidar com a componente de codificação do ficheiro de áudio.

Para executar este programa é necessário especificar o caminho para o ficheiro de áudio a codificar, o nome a atribuir ao ficheiro de output, e ainda é possível customizar o seguinte conjunto de parâmetros:

- *p* : Indica o predictor a ser utilizado na codificação. Caso o parâmetro não seja fornecido, o seu valor por defeito é de 4. Os valores possíveis serão detalhados mais à frente.
- *w* : Indica o tamanho da janela a considerar na otimização do parâmetro m dos códigos de golomb. Caso o parâmetro não seja fornecido, o seu valor por defeito é de 50.
- *q* : Indica o número de bits a utilizar no processo de quantização no caso de utilização do codificador de áudio com perdas. Caso o parâmetro não seja

fornecido ou o seu valor seja de 0, será utilizado o codificador de áudio sem perdas.

No que diz respeito ao parâmetro p, o mesmo pode assumir os seguintes valores:

- *0* : Será utilizado o predictor 0 que assume a seguinte fórmula: $\hat{x}_n{}^{(0)} = 0$
- *1* : Será utilizado o predictor 1 que assume a seguinte fórmula: $\hat{x}_n{}^{(1)} = x_{n-1}$
- 2 : Será utilizado o predictor 2 que assume a seguinte fórmula: $\hat{x}_n{}^{(2)} = 2x_{n-1} - x_{n-2}$
- 3 : Será utilizado o predictor 3 que assume a seguinte fórmula: $\hat{x}_n{}^{(3)} = 3x_{n-1} - 3x_{n-2} + x_{n-3}$
- 4 : Será utilizado em cada bloco de codificação do áudio o melhor predictor para aquele bloco específico.

```
                    Usage:
./audio_encoder [ -p ] [ -w ] [ -q ] <input file> <output file>

                  Usage Example:
./audio_encoder -p 0 -w 100 -q 14 ../audio-examples/sample01.wav
                  compressed.bin
```

Ao executar o nosso programa, o primeiro passo a ser efetuado é a verificação dos parâmetros fornecidos pelo utilizador. É verificada a existência e formato do ficheiro de input fornecido. Para além disso é verificada a existência e valores dos parâmetros *p*, *w* e *q* fornecidos.

Prosseguindo, é escrito no ficheiro de output algumas informações que o decodificador vai necessitar, tais como, o formato do ficheiro de input, número de canais, número de frames, taxa de amostragem, tamanho do bloco de áudio utilizado, tipo de predictor utilizado na codificação, tamanho da janela utilizada na otimização do parâmetro *m* e número de bits utilizados na quantização (no caso do codificador sem perda, este valor é de 0).

Devemos destacar a passagem do tipo de predictor a utilizar. Caso o predictor tenha um valor de 0-3, ou seja, é utilizado o mesmo predictor durante toda a codificação, o mesmo é apenas escrito para o ficheiro de output uma única vez logo no início. Caso o predictor utilizado seja o 4, em que é utilizado o melhor predictor para codificar cada bloco do áudio, aqui nesta escrita inicial é escrito o número 4 para que o decodificador compreenda que foi este o tipo utilizado. Adicionalmente, no início de cada bloco de áudio, o tipo de predictor utilizado para aquele bloco (que terá um valor de 0-3) é escrito no ficheiro de output.

No fim da escrita destes parâmetros iniciais, é chamada a função *encodeMonoAudio()* ou *encodeStereoAudio()* consoante o número de canais do áudio a codificar.

Começando pelo caso em que o áudio a codificar é mono, de uma forma geral, o processo de codificação é efetuado da seguinte forma:

No caso de a codificação ser efetuada com um predictor fixo (0-3), para cada bloco do áudio, as amostras são lidas uma a uma e é calculado o valor do residual consoante as fórmulas descritas no início desta secção. Após isso, o valor do residual é codificado utilizando os códigos de golomb e é escrito para o ficheiro de output. O valor da amostra inicial é armazenado num array com três posições apenas para que possa ser utilizado nos

cálculos dos residuais das amostras seguintes. Adicionalmente, consoante o tamanho da janela definido pelo parâmetro *w,* periodicamente o valor do parâmetro *m* da codificação de golomb é atualizado através das seguintes fórmulas:

$$mean = \frac{sumResiduals}{totalResiduals}$$

$$\alpha = \frac{mean}{mean + 1}$$

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

No caso de a codificação ser efetuado com predictor 4, a codificação é semelhante, mas para cada bloco de áudio é utilizado o melhor predictor. Assim sendo, em cada bloco de áudio são efetuados os cálculos e codificados os residuais para cada predictor possível. As strings de codificação são armazenadas e no fim do bloco é utilizado o predictor cuja String de codificação for mais reduzida, e por isso, é mais eficiente. No que diz respeito à atualização do parâmetro m, visto que a mesma depende do predictor que é utilizado, é efetuado os cálculos de atualização para todos os predictores possíveis. De realçar que, como dito anteriormente, no início de cada bloco é escrito no ficheiro de output o predictor utilizado para codificar aquele bloco, o parâmetro m utilizado no início desse bloco e o valor inicial da soma de residuais no início do bloco, visto que estes dois valores são dependentes do predictor utilizado.

Relativamente à codificação de áudios stereo, existem bastantes semelhanças nos processos de codificação. Nestes áudios, invés de serem utilizados os valores das amostras dos canais 0 e 1 no cálculo dos valores dos residuais, é efetuado o cálculo do mid e side channel do áudio. Os valores das amostras do mid e side channel são posteriormente utilizados para calcular os residuais e é efetuada a codificação destes canais, de forma intercalada. O restante processo de codificação é bastante semelhante.

No que diz respeito ao codificador com perda de informação, o processo de codificação é idêntico. No entanto, existem algumas diferenças que vamos enumerar de seguida.

Após o cálculo do residual, este é quantizado a um dos níveis definidos no WAVQuant e, quando é feita a escrita na BitStream, este é codificado com o código do seu nível antes de ser codificado usando os códigos de Golomb. Outra consideração que tem de ser feita em codificadores com perdas, é que para os predictores tem de ser usados os valores reconstruídos e não o valor real das samples anteriores. Para isso, construímos o valor anterior através da utilização dos cálculos usados na decodificação usando o residual quantizado para que depois este valor reconstruído possa ser usado nos predictores das samples seguintes.

Este processo de quantização está apenas disponível para os predictores de 0 a 3, ou seja, para o modo de codificação com predictor fixo. Para o predictor 4 não foi efetuada a implementação visto que iria seguir exatamente os mesmos passos do predictor 4 efetuado para o codificador sem perdas com as diferenças descritas acima, mas iria adicionar complexidade extra que iria trazer menor desempenho.

3.5. audio_decoder.cpp

Para completar esta terceira parte do projeto, vamos agora descrever a implementação do programa *audio_decoder.cpp,* que tem como responsabilidade lidar com o processo de descodificação dos ficheiros de áudio que foram codificados com o programa anterior.

Para executar este programa é necessário especificar o caminho para o ficheiro de áudio a descodificar e o nome a atribuir ao ficheiro de output.

```
Usage:
./audio_decoder <input file> <output file>

Usage Example:
./audio_decoder compressed.bin output.wav
```

Ao executar o nosso programa, o primeiro passo a ser efetuado é a verificação dos parâmetros fornecidos pelo utilizador. Neste caso, é verificada a existência do ficheiro de input fornecido.

Prosseguindo, é efetuada a leitura de algumas informações necessárias e que foram introduzidas no início do ficheiro de input por parte do codificador, tais como, o formato do ficheiro de input, número de canais, número de frames, taxa de amostragem, tamanho do bloco de áudio utilizado, tipo de predictor utilizado na codificação, tamanho da janela utilizada na otimização do parâmetro m, e o número de bits utilizados na quantização.

No fim da leitura destes parâmetros iniciais, é chamada a função *decodeMonoAudio()* ou *decodeStereoAudio()* consoante o número de canais do áudio a descodificar.

Começando pelo caso em que o áudio a decodificar é mono, de uma forma geral, o processo de descodificação é efetuado da seguinte forma:

No caso da decodificação ser efetuada com um predictor fixo (0-3), as amostras são lidas uma a uma e é descodificado o valor do residual com o qual é efetuada a conversão para o valor original da amostra, consoante o predictor utilizado. O valor da amostra é armazenado num array com três posições apenas para que possa ser utilizado nos cálculos das amostras seguintes. Adicionalmente, consoante o tamanho da janela definido no codificador, periodicamente o valor do parâmetro m da codificação de golomb é atualizado seguindo a mesma fórmula que foi descrita no codificador.

No caso da decodificação ser efetuado com predictor 4, a descodificação é semelhante, mas para cada bloco de áudio é utilizado o melhor predictor. Assim sendo, em cada bloco de áudio é inicialmente lido e decodificado o número de identificação do predictor utilizado naquele bloco, bem como o valor do parâmetro m inicial dos códigos de golomb que é ótimo para aquele predictor naquela secção de descodificação, e também a soma dos residuais no início do bloco para aquele predictor para que o descodificador fique sincronizado com o codificador na otimização do m. De resto, o processo é igual.

Relativamente à decodificação de áudios stereo, existem bastantes semelhanças nos processos de descodificação. Nestes áudios, a principal alteração no processo é o facto

de que as amostras que são descodificadas representam os valores do mid e side channel dos áudios originais. Com estes valores é necessário fazer a conversão para obter os valores de áudio original.

No caso de ter sido utilizado o codificador com perda de informação, o processo de descodificação é idêntico ao descrito anteriormente. No entanto, depois de decodificar os códigos de Golomb obtemos o código do nível de quantização e temos de descodificar este usando WAVQuant para podermos obter assim o residual.

3.6. image_encoder.cpp

Na quarta parte do projeto, como referido anteriormente, era requisitada a implementação de um codec de imagens baseado na utilização dos códigos de golomb desenvolvidos na componente anterior do projeto e no conceito de codificação preditiva.

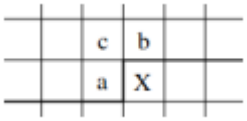
Para isso, foram implementados dois programas distintos (*image_encoder.cpp* e *image_decoder.cpp*). Vamos começar por descrever o programa *image_encoder.cpp*.

O programa *image_encoder.cpp* tinha como objetivo lidar com a componente de codificação do ficheiro da imagem.

Para executar este programa é necessário especificar o caminho para a imagem a codificar, o nome a atribuir ao ficheiro de output, e ainda é possível customizar o seguinte conjunto de parâmetros:

- *p* : Indica o predictor a ser utilizado na codificação. Caso o parâmetro não seja fornecido, o seu valor por defeito é de 0. Os valores possíveis serão detalhados mais à frente.
- *w* : Indica o tamanho da janela a considerar na otimização do parâmetro m dos códigos de golomb. Caso o parâmetro não seja fornecido, o seu valor por defeito é de 50.

Analisando a figura 1, e considerando que X é o pixel em que se pretende calcular o residual, e “a” “b” e “c” são os pixels utilizados para calcular os diferentes predictores, o valor de *p* pode assumir os seguintes valores:



- *0 : Será utilizado em cada bloco de codificação do áudio o melhor predictor para aquele bloco específico.*
- *1 : Será utilizado o predictor 1 que assume a seguinte fórmula:* a
- *2 : Será utilizado o predictor 2 que assume a seguinte fórmula:* b
- *3 : Será utilizado o predictor 3 que assume a seguinte fórmula:* c
- *4 : Será utilizado o predictor 4 que assume a seguinte fórmula:* $a + b - c$
- *5 : Será utilizado o predictor 5 que assume a seguinte fórmula:* $a + (\frac{b-c}{2})$
- *6 : Será utilizado o predictor 6 que assume a seguinte fórmula:* $b + (\frac{a-c}{2})$
- *7 : Será utilizado o predictor 7 que assume a seguinte fórmula:* $\frac{a+b}{2}$

```
Usage:
./image_encoder [ -p ] [ -w ] <input file> <output file>

Usage Example:
./image_encoder -p 0 -w 10 ../pics-examples/airplane.ppm
compressed.bin
```

Ao executar o nosso programa, o primeiro passo a ser efetuado é a verificação dos parâmetros fornecidos pelo utilizador. É verificada a existência e valores dos parâmetros p, w e q fornecidos.

Prosseguindo, é criado um objeto do tipo Mat para carregar a imagem que o utilizador definiu para uma matriz para que mais tarde se consiga realizar operações com os pixels. De seguida, é escrito no ficheiro de output algumas informações que o decodificador vai necessitar, tais como, o tipo de predictor utilizado na codificação, o número de linhas da imagem, o número de colunas da imagem, o tipo da imagem, e o tamanho da janela utilizada na otimização do parâmetro m.

Devemos destacar a passagem do tipo de predictor a utilizar. Caso o predictor tenha um valor de 1-7, ou seja, é utilizado o mesmo predictor durante toda a codificação, o mesmo é apenas escrito para o ficheiro de output uma única vez logo no início. Caso o predictor utilizado seja o 0, em que é utilizado o melhor predictor para codificar cada bloco do áudio, aqui nesta escrita inicial é escrito o número 0 para que o decodificador compreenda que foi este o tipo utilizado. Adicionalmente, no início de cada bloco de áudio, o tipo de predictor utilizado para aquele bloco (que terá um valor de 1-7) é escrito no ficheiro de output.

No fim da escrita destes parâmetros iniciais, é chamada a função *encodeImage()*.

Nesta função, no caso de a codificação ser efetuada com um predictor fixo (1-7), para cada bloco de pixels da imagem, os pixels são lidos um a um e é calculado o valor do residual consoante as fórmulas descritas no início desta secção. Após isso, o valor do residual é codificado utilizando os códigos de golomb e é escrito para o ficheiro de output. Os valores dos pixels *a b* e *c* vão sendo atualizados à medida do pixel atual pelo que estes pixels em situações onde não existem na imagem assumem o valor de (0,0,0), mais concretamente no caso do pixel *c* e *b* quando se codifica a primeira linha da imagem, e no caso dos pixels *c* e *a* quando se codifica os pixels da primeira coluna. Adicionalmente, consoante o tamanho da janela definido pelo parâmetro *w*, periodicamente o valor do parâmetro *m* da codificação de golomb é atualizado através das seguintes fórmulas:

$$mean = \frac{sumResiduals}{totalResiduals}$$

$$\alpha = \frac{mean}{mean + 1}$$

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

No caso de a codificação ser efetuado com predictor 0, a codificação é semelhante, mas para cada bloco de pixels (tamanho do bloco igual a 2000 pixels) é utilizado o melhor predictor. Assim sendo, em cada bloco de pixels são efetuados os cálculos e codificados os

residuais para cada predictor possível. As strings de codificação são armazenadas e no fim do bloco é utilizado o predictor cuja string de codificação for mais reduzida, e por isso, é mais eficiente. No que diz respeito à atualização do parâmetro m, visto que a mesma depende do predictor que é utilizado, é efetuado os cálculos de atualização para todos os predictores possíveis. De realçar que, como dito anteriormente, no início de cada bloco é escrito no ficheiro de output o predictor utilizado para codificar aquele bloco, o parâmetro m utilizado inicialmente desse bloco e o valor inicial da soma de residuais no início do bloco, visto que estes dois valores são dependentes do predictor utilizado.

3.7. image_decoder.cpp

Para completar esta quarta parte do projeto, vamos agora descrever a implementação do programa *image_decoder.cpp*, que tem como objetivo lidar com o processo de decodificação das imagens que foram codificadas com o programa anterior. Para executar este programa é necessário especificar o caminho para imagem a descodificar e o nome a atribuir ao ficheiro de output.

```
Usage:
./image_decoder <input file> <output file>

Usage Example:
./image_decoder compressed.bin output.ppm
```

Ao executar o nosso programa, o primeiro passo a ser efetuado é a verificação dos parâmetros fornecidos pelo utilizador. Neste caso, é verificada a existência do ficheiro de input fornecido.

Prosseguindo, é efetuada a leitura de algumas informações necessárias e que foram introduzidas no início do ficheiro de input por parte do codificador, tais como, o tipo de predictor utilizado na codificação, o número de linhas da imagem, o número de colunas da imagem, o tipo da imagem, e o tamanho da janela utilizada na otimização do parâmetro m.

No fim da leitura destes parâmetros iniciais, é chamada a função *decodeImage().*

No caso de a descodificação ser efetuada com um predictor fixo (1-7), os pixels são lidos um a um e é descodificado o valor do residual com o qual é efetuada a conversão para o valor original do pixel, consoante o predictor utilizado. Os valores dos pixels a b e c são guardados conforme na codificação descrita anteriormente. Adicionalmente, consoante o tamanho da janela definido no codificador, periodicamente o valor do parâmetro m da codificação de golomb é atualizado seguindo a mesma fórmula que foi descrita no codificador.

No caso de a descodificação ser efetuado com predictor 0, a descodificação é semelhante, mas para cada bloco de pixels (tamanho do bloco igual a 2000 pixels) é utilizado o melhor predictor. Assim sendo, em cada bloco de pixels é inicialmente lido e decodificado o número de identificação do predictor utilizado naquele bloco, bem como o valor do parâmetro m inicial dos códigos de golomb que é ótimo para aquele predictor naquela secção de descodificação, e também a soma dos residuais no início do bloco para aquele predictor para que o descodificador fique sincronizado com o codificador na otimização do m. De resto, o processo é igual.

4. Análise de resultados

Nesta seção do relatório iremos apresentar algumas experiências efetuadas com os programas desenvolvidos e analisar os resultados obtidos.

4.1. cp_image.cpp

Relativamente ao programa cp_image.cpp, efetuamos experiências com cada um dos modos possíveis de utilizar o programa, ou seja, testamos cada uma das operações possíveis de aplicar.

Todas as operações foram efetuadas utilizando a mesma imagem base, que se encontra no diretório designado “pics-examples” e que possui o nome de “lena.ppm”. A imagem base é a seguinte:

Figura 1: Imagem original

A primeira operação implementada e testada foi a de criação de uma cópia da imagem original. Posto isto, executámos o programa com os seguintes argumentos:

```
./cp_image COPY ../pics-examples/lena.ppm output.ppm
```

Após a execução, o resultado obtido foi o esperado, ou seja, uma cópia da imagem inicial, e pode ser visualizado na seguinte imagem:

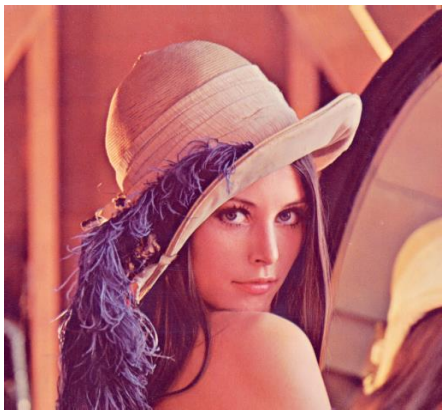


Figura 2: Resultado da operação de COPY

A segunda operação implementada e testada foi a de criação de uma versão negativa da imagem original. Posto isto, executámos o programa com os seguintes argumentos:

```
./cp_image NEGATIVE ../pics-examples/lena.ppm output.ppm
```

Na nossa opinião, através da análise do resultado obtido podemos verificar que a operação foi bem-sucedida, isto é, foi obtida uma imagem com as cores invertidas:

Figura 3: Resultado da operação NEGATIVE

A terceira operação implementada e testada foi a operação de criação de uma imagem espelhada horizontalmente da imagem original. Posto isto, executámos o programa com os seguintes argumentos:

```
./cp_image MIRROR_HORIZONTAL ../pics-examples/lena.ppm output.ppm
```

Ao visualizar a imagem produzida, podemos verificar que a mesma foi obtida através de uma reflexão horizontal, ou seja, utilizando o eixo das ordenadas, e por isso acreditamos que a operação foi bem-sucedida:

Figura 4: Resultado da operação MIRROR_HORIZONTAL

A quarta operação implementada e testada foi a operação de criação de uma imagem espelhada verticalmente da imagem original. Posto isto, executámos o programa com os seguintes argumentos:

```
./cp_image MIRROR_VERTICAL ../pics-examples/lena.ppm output.ppm
```

De forma semelhante à operação anterior, ao visualizar a imagem produzida, podemos verificar que a mesma foi obtida através de uma reflexão. No entanto, neste caso, a reflexão é vertical visto que utiliza o eixo das abscissas como eixo de reflexão, e por isso acreditamos que a operação foi também bem-sucedida:

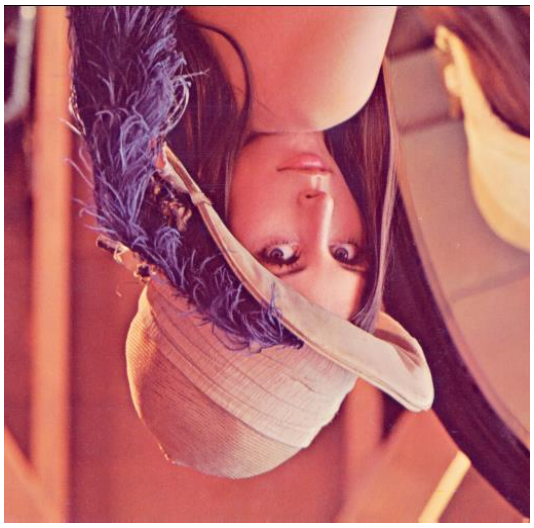


Figura 5: Resultado da operação de MIRROR_VERTICAL

A quinta operação implementada e testada foi a operação de rotação da imagem original. Para esta operação foram efetuados vários testes. Vamos apresentar, para cada teste, com que parâmetros o programa foi executado e o resultado obtido.

Rotação a 90º:

```
./cp_image -r 90 ROTATE ../pics-examples/lena.ppm output.ppm
```

Figura 6: Resultado da operação de ROTATE (90º)

Rotação a 180º:

```
./cp_image -r 180 ROTATE ../pics-examples/lena.ppm output.ppm
```

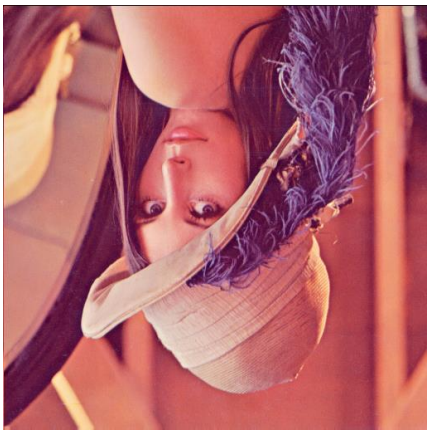


Figura 7: Resultado da operação de ROTATE (180º)

Rotação a 270º:

```
./cp_image -r 270 ROTATE ../pics-examples/lena.ppm output.ppm
```

Figura 8: Resultado da operação de ROTATE (270º)

Rotação a 360º:

```
./cp_image -r 360 ROTATE ../pics-examples/lena.ppm output.ppm
```

Figura 9: Resultado da operação de ROTATE (360º)

Rotação a 450º:

```
./cp_image -r 450 ROTATE ../pics-examples/lena.ppm output.ppm
```

Figura 10: Resultado da operação de ROTATE (450º)

Rotação a -90º:

```
./cp_image -r -90 ROTATE ../pics-examples/lena.ppm output.ppm
```

Figura 11: Resultado da operação de ROTATE (-90º)

Pela análise dos resultados obtidos, consideramos que todas as rotações foram efetuadas corretamente pelo que a operação de rotação foi implementada com sucesso.

A sexta e última operação implementada e testada foi a operação de variação da intensidade da luz da imagem original. De forma semelhante à anterior, para esta operação foram efetuados vários testes. Vamos apresentar, para cada teste, com que parâmetros o programa foi executado e o resultado obtido.

Fator de intensidade igual a 1.5:

```
./cp_image -i 1.5 INTENSITY ../pics-examples/lena.ppm output.ppm
```

Figura 12: Resultado da operação de INTENSITY. Fator = 1.5

Fator de intensidade igual a 2:

```
./cp_image -i 2 INTENSITY ../pics-examples/lena.ppm output.ppm
```

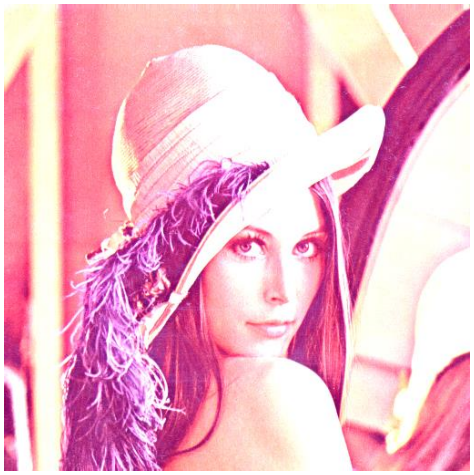


Figura 13: Resultado da operação de INTENSITY. Fator = 2

Fator de intensidade igual a 0.5:

```
./cp_image -i 0.5 INTENSITY ../pics-examples/lena.ppm output.ppm
```

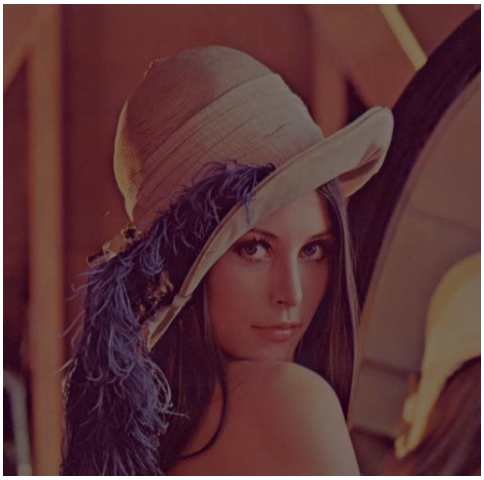


Figura 14: Resultado da operação de INTENSITY. Fator = 0.5

Ao visualizar as imagens obtidas, podemos verificar que a intensidade da luz das mesmas é alterada consoante o fator de intensidade escolhido. Fatores de intensidade superiores a 1 vão aumentar a luz enquanto fatores de intensidade que se situam entre 0 e 1 vão diminuir a luz. Posto isto, acreditamos que a operação foi implementada com sucesso.

4.2. audio_encoder.cpp e audio_decoder.cpp

Os programas *audio_encoder.cpp* e *audio_decoder.cpp*, como já descrito, permitem efetuar a codificação e descodificação com e sem perdas de ficheiros de áudio utilizando codificação preditiva com códigos de golomb. A forma de utilização destes programas é descrita nas secções 3.4 e 3.5 deste relatório.

Diversas experiências foram realizadas para avaliar o impacto dos diferentes parâmetros destes programas, analisar as suas performances, capacidade de solucionar o problema que existia à partida (codificação e decodificação corretas) e ainda taxas de compressão obtidas.

A primeira experiência a relatar consistiu na variação do parâmetro *p,* associado ao predictor utilizado. Nos restantes parâmetros, os valores utilizados foram os valores por defeito. Foi registada a performance (tempo de execução) tanto do codificador como do decodificador e ainda as taxas de compressão obtidas.

Os resultados obtidos para áudios monos estão presentes na seguinte tabela:

Tabela 1: Tabela de resultados de variação do parâmetro p, para áudios monos.

Áudio: sample01Mono.wav. Tamanho: 2 588 420 bytes				
predictor	Tempo Execução Codificador (s)	Tempo de Execução Decodificador (s)	Tamanho Ficheiro Comprimido	Taxa de compressão
0	17.21 s	7.82 s	2 266 733 bytes	- 12.43%
1	5.33 s	1.56 s	1 877 759 bytes	- 27.46%
2	4.57 s	1.13 s	1 781 704 bytes	- 31.17%
3	4.98 s	1.36 s	1 820 667 bytes	- 29.66%
4	20.49 s	1.14 s	1 781 869 bytes	- 31.16%
Áudio: sample02Mono.wav. Tamanho: 1 294 820 bytes				
predictor	Tempo Execução Codificador (s)	Tempo de Execução Decodificador (s)	Tamanho Ficheiro Comprimido	Taxa de compressão
0	5.15 s	1.92 s	1 020 274 bytes	- 21.20%
1	2.63 s	0.75 s	852 195 bytes	- 34.18%

2	2.73 s	0.85 s	838 509 bytes	- 35.24%
3	3.17 s	1.12 s	873 538 bytes	- 32.53%
4	7.16 s	0.78 s	837 598 bytes	- 35.31%

O primeiro e mais importante resultado a sublinhar foi o facto de que para ambos os áudios e para todos os predictors, o áudio final é igual ao original pelo que o mecanismo de codificação e decodificação estará correto.

Ao nível da performance do codificador, podemos verificar que a codificação com o predictor 4 é mais lenta em ambos os ficheiros, visto que, requer que sejam efetuados os cálculos e codificação para cada um dos predictors por forma a avaliar qual deles é o melhor.

Ao nível da performance do decodificador, a mesma é bastante aceitável à exceção do predictor 0 para o primeiro áudio que tem um tempo de execução mais longo. O mesmo poderá dever-se ao facto de que os valores que foram codificados, e que por consequente, necessitam de ser decodificados são mais elevados.

Por fim, podemos verificar que as melhores taxas de compressão para o primeiro áudio superam os 30% de compressão do ficheiro. A razão que leva a que a compressão com o predictor 4 seja ligeiramente menor do que com o predictor 2 deve-se ao facto de que no predictor 4, no início de cada bloco é necessário escrever alguns valores tais como qual foi o predictor utilizado naquele bloco. Nos predictor 0-3, como é sempre utilizado o mesmo, estes valores não são escritos. Assim sendo, caso todos os blocos do ficheiro de áudio sejam melhor comprimidos com o predictor 2, utilizando o predictor 2 invés do 4 vamos ter uma ligeira melhoria.

No que diz respeito ao segundo ficheiro, novamente os melhores predictors foram o 2 e o 4 obtendo taxas de compressão superiores a 35% do tamanho original.

O mesmo teste foi efetuado para ficheiros stereo. Os resultados obtidos para áudios stereos estão presentes na seguinte tabela:

Tabela 2: Tabela de resultados de variação do parâmetro p, para áudios stereos.

Áudio: sample01.wav. Tamanho: 5 176 796 bytes				
p	Tempo Execução Codificador (s)	Tempo de Execução Decodificador (s)	Tamanho Ficheiro Comprimido	Taxa de compressão
0	27.79 s	10.88 s	4 339 629 bytes	- 16.17%
1	10.90 s	2.90 s	3 717 242 bytes	- 28.19%
2	10.02 s	2.58 s	3 626 166 bytes	- 29.95%
3	12.60 s	3.42 s	3 747 225 bytes	- 27.61%
4	36.78 s	2.56 s	3 626 491 bytes	- 29.95%

Áudio: sample02.wav. Tamanho: 2 589 596 bytes				
p	Tempo Execução Codificador (s)	Tempo de Execução Decodificador (s)	Tamanho Ficheiro Comprimido	Taxa de compressão
0	12.94 s	5.37 s	2 109 509 bytes	- 18.54%
1	6.19 s	2.02 s	1 777 702 bytes	- 31.35%
2	6.51 s	2.26 s	1 751 277 bytes	- 32.37%
3	8.00 s	3.15 s	1 821 864 bytes	- 29.65%
4	20.95 s	2.12 s	1 749 320 bytes	- 32.45%

Como podemos verificar ao observar os resultados, os mesmos são semelhantes aos resultados obtidos para os áudios mono. De realçar que, os tempos de execução são mais elevados, visto que neste caso os áudios são formados por dois canais, pelo que são ficheiros de tamanho mais elevado.

A segunda experiência a relatar consistiu na variação do parâmetro *w,* associado ao tamanho da janela a considerar na otimização do parâmetro *m* dos códigos de golomb. Nos restantes parâmetros, os valores utilizados foram os valores por defeito. Foi registada a performance (tempo de execução) tanto do codificador como do decodificador e ainda as taxas de compressão obtidas.

Esta experiência foi efetuada para um áudio mono e outro stereo. Os resultados obtidos estão presentes na seguinte tabela:

Tabela 3: Tabela de resultados de variação do parâmetro w, para áudios monos e stereos.

Áudio: sample01Mono.wav. Tamanho: 2 588 420 bytes				
w	Tempo Execução Codificador (s)	Tempo de Execução Decodificador (s)	Tamanho Ficheiro Comprimido	Taxa de compressão
20	23.13 s	1.26 s	1 783 838 bytes	- 31.08%
50	21.74 s	1.19 s	1 781 869 bytes	- 31.16%
100	20.19 s	1.11 s	1 781 036 bytes	- 31.19%
1000	18.80 s	1.01 s	1 788 330 bytes	- 30.91%
10000	18.34 s	1.08 s	1 804 039 bytes	- 30.30%
Áudio: sample01.wav. Tamanho: 5 176 796 bytes				

w	Tempo Execução Codificador (s)	Tempo de Execução Decodificador (s)	Tamanho Ficheiro Comprimido	Taxa de compressão
20	40.58 s	2.84 s	3 631 472 bytes	- 29.85%
50	36.69 s	2.60 s	3 626 491 bytes	- 29.94%
100	35.19 s	2.53 s	3 625 183 bytes	- 29.97%
1000	32.52 s	2.40 s	3 642 918 bytes	- 29.63%
10000	31.50 s	2.35 s	3 678 712 bytes	- 29.94%

Ao analisar os resultados obtidos podemos identificar que o tempo de execução do codificador aumenta com a diminuição do parâmetro *w*, algo que seria de esperar visto que a diminuição deste parâmetro implica um maior número de otimizações do parâmetro *m* dos códigos de golomb. De forma semelhante, também no decodificador podemos observar esta correlação, no entanto com um significado muito menos expressivo visto que as diferenças de tempo são reduzidas.

No que diz respeito à taxa de compressão do ficheiro, não foi detetada variação significativa da mesma. No entanto, nos valores obtidos no primeiro ficheiro podemos verificar que à medida que o parâmetro *w* é mais reduzido, maiores são as taxas obtidas, visto que existe uma otimização do parâmetro *m* dos códigos de golomb mais frequente.

A terceira e última experiência a relatar consistiu na variação do parâmetro *q,* associado ao número de bits a utilizar no processo de quantização do codificador com perdas. Nos restantes parâmetros, o predictor utilizado foi o 2 e o tamanho da janela assumiu o valor por defeito. Foi registada a performance (tempo de execução) tanto do codificador como do decodificador, as taxas de compressão obtidas e ainda o valor SNR que reflete a relação sinal-ruído do áudio final por forma a identificar se a perda de informação foi elevada ou não.

Esta experiência foi efetuada para um áudio mono e outro stereo. Os resultados obtidos estão presentes na seguinte tabela:

Tabela 4: Tabela de resultados de variação do parâmetro q, para áudios monos e stereos.

Áudio: sample01Mono.wav. Tamanho: 2 588 420 bytes					
q	Tempo Execução Codificador (s)	Tempo de Execução Decodificador (s)	Tamanho Ficheiro Comprimido	Taxa de compressão	SNR Áudio Final (dB)
1	74.33 s	1.95 s	2 426 591 bytes	- 6.25%	- 8.40 dB
2	44.67 s	1.17 s	2 265 011 bytes	- 12.49%	- 1.97 dB
4	15.52 s	0.43 s	1 951 709 bytes	- 24.59%	10.17 dB
6	7.09 s	0.28 s	1 710 785 bytes	- 33.90%	22.20 dB

8	6.46 s	0.24 s	1 561 383 bytes	- 39.67%	34.25 dB
10	5.12 s	0.37 s	1 540 479 bytes	- 40.48%	46.28 dB
12	5.14 s	0.56 s	1 554 275 bytes	- 39.95%	58.29 dB
14	5.25 s	1.02 s	1 603 363 bytes	- 38.05%	69.86 dB
Áudio: sample01.wav. Tamanho: 5 176 796 bytes					
q	Tempo Execução Codificador (s)	Tempo de Execução Decodificador (s)	Tamanho Ficheiro Comprimido	Taxa de compressão	SNR Áudio Final
1	154.00 s	3.86 s	4 853 150 bytes	- 6.25%	- 14.64 dB
2	88.86 s	2.28 s	4 530 484 bytes	- 12.48%	- 2.83 dB
4	31.52 s	0.90 s	3 902 642 bytes	- 24.61%	9.41 dB
6	13.96 s	0.50 s	3 449 312 bytes	- 33.37%	21.46 dB
8	11.34 s	0.53 s	3 178 364 bytes	- 38.60%	33.50 dB
10	10.89 s	0.76 s	3 146 090 bytes	- 39.22%	45.53 dB
12	10.98 s	1.24 s	3 171 915 bytes	- 38.73%	57.52 dB
14	11.51 s	2.25 s	3 269 264 bytes	- 36.85%	68.81 dB

A partir dos resultados obtidos podemos verificar que, como esperado, a utilização de um compressor com perdas permite obter melhores taxas de compressão em comparação com compressor sem perdas. No entanto, como podemos observar pelos valores de SNR, esta melhor compressão implica a perda de qualidade no áudio e consequente introdução de ruído no áudio final.

4.3. image_encoder.cpp e image_decoder.cpp

Os programas *image_encoder.cpp* e *image_decoder.cpp*, como já descrito, permitem efetuar a codificação e descodificação sem perdas de ficheiros de imagem utilizando codificação preditiva com códigos de golomb. A forma de utilização destes programas é descrita nas secções 3.6 e 3.7 deste relatório.

Diversas experiências foram realizadas para avaliar o impacto dos diferentes parâmetros destes programas, analisar as suas performances, capacidade de solucionar o

problema que existia à partida (codificação e decodificação corretas) e ainda taxas de compressão obtidas.

A primeira experiência a relatar consistiu na variação do parâmetro *p,* associado ao predictor utilizado. Nos restantes parâmetros, os valores utilizados foram os valores por defeito. Foi registada a performance (tempo de execução) tanto do codificador como do decodificador e ainda as taxas de compressão obtidas.

Os resultados obtidos estão presentes na seguinte tabela:

Tabela 5: Tabela de resultados de variação do parâmetro p, para imagens.

Imagem: airplane.ppm. Tamanho: 786 447 bytes				
predictor	Tempo Execução Codificador (s)	Tempo de Execução Decodificador (s)	Tamanho Ficheiro Comprimido	Taxa de compressão
0	0.43 s	0.26 s	482 438 bytes	- 38.65%
1	1.29 s	0.27 s	530 923 bytes	- 32.49%
2	1.27 s	0.24 s	516 185 bytes	- 34.36%
3	1.34 s	0.25 s	563 390 bytes	- 28.36%
4	1.27 s	0.24 s	502 846 bytes	- 36.06%
5	1.20 s	0.26 s	497 836 bytes	- 36.69%
6	1.20 s	0.26 s	491 401 bytes	- 37.51%
7	1.21 s	0.26 s	496 129 bytes	- 36.91%
Imagem: lena.ppm. Tamanho: 786 447 bytes				
predictor	Tempo Execução Codificador (s)	Tempo de Execução Decodificador (s)	Tamanho Ficheiro Comprimido	Taxa de compressão
0	0.42 s	0.27 s	534 563 bytes	- 32.02%
1	1.38 s	0.29 s	581 085 bytes	- 26.11%
2	1.45 s	0.27 s	541 479 bytes	- 31.14%
3	1.44 s	0.25 s	595 781 bytes	- 24.24%
4	1.38 s	0.25 s	564 202 bytes	- 28.25%
5	1.36 s	0.27 s	555 485 bytes	- 29.36%

6	1.33 s	0.27 s	538 179 bytes	- 31.56%
7	1.33 s	0.24 s	540 839 bytes	- 31.23%

Relativamente aos tempos de execução tanto do codificador como decodificador, eles são bastante ótimos e semelhantes entre as imagens visto que elas possuem exatamente o mesmo tamanho.

No que diz respeito às taxas de compressão, as mesmas são superiores na primeira imagem onde vários predictores distintos chegam a ultrapassar o valor de 35%. Na segunda imagem são ligeiramente inferiores, no entanto, bastante aceitáveis.

A segunda experiência a relatar consistiu na variação do parâmetro *w,* associado ao tamanho da janela a considerar na otimização do parâmetro *m* dos códigos de golomb. Nos restantes parâmetros, os valores utilizados foram os valores por defeito. Foi registada a performance (tempo de execução) tanto do codificador como do decodificador e ainda as taxas de compressão obtidas.

Os resultados obtidos estão presentes na seguinte tabela:

Tabela 6: Tabela de resultados de variação do parâmetro w, para imagens.

Imagem: airplane.ppm. Tamanho: 786 447 bytes				
w	Tempo Execução Codificador (s)	Tempo de Execução Decodificador (s)	Tamanho Ficheiro Comprimido	Taxa de compressão
20	0.45 s	0.22 s	475 056 bytes	- 39.59%
50	0.39 s	0.26 s	482 438 bytes	- 38.65%
100	0.38 s	0.22 s	488 962 bytes	- 37.82%
1000	0.41 s	0.26 s	492 154 bytes	- 37.42%
10000	0.41 s	0.26 s	498 228 bytes	- 36.64%
Imagem: lena.ppm. Tamanho: 786 447 bytes				
w	Tempo Execução Codificador (s)	Tempo de Execução Decodificador (s)	Tamanho Ficheiro Comprimido	Taxa de compressão
20	0.45 s	0.24 s	532 000 bytes	- 32.35%
50	0.42 s	0.26 s	534 563 bytes	- 32.02%
100	0.40 s	0.23 s	537 544 bytes	- 31.64%
1000	0.45 s	0.27 s	538 479 bytes	- 31.53%

10000	0.42 s	0.27 s	543 841 bytes	- 30.84%

Nesta experiência, ao nível dos tempos de execução podemos verificar que a alteração do parâmetro *w* não provoca qualquer tipo de melhoria ou perda de performance.

No entanto, na componente das taxas de compressão podemos verificar que existe uma melhoria das taxas obtidas quando o parâmetro *w* é mais reduzido. Isto deve-se ao facto de que, com valores de *w* mais reduzidos, o parâmetro *m* dos códigos de golomb é otimizado mais vezes pelo que os códigos utilizados na compressão são mais adequados ao conjunto de samples recebido.

5. Conclusão

A realização deste projeto permitiu-nos, numa primeira fase, obter um maior contacto com manipulação de ficheiros de imagem através da manipulação dos seus pixels por forma a conseguir efetuar operações de rotação, reflexão, variação da intensidade dos pixels, entre outras.

Este projeto permitiu também colocar em práticas os conhecimentos obtidos nas aulas teóricas relativos à construção de códigos de golomb e aplicação dos mesmos em codificação preditiva. Deste modo fomos capazes de desenvolver codificadores e decodificadores tanto para ficheiros de áudio como de imagem. Foi ainda possível, no caso do áudio, implementar e verificar as diferenças nos resultados obtidos entre um codificador com e sem perdas de informação.

Por fim, consideramos que os resultados obtidos vão de acordo com os esperados, e que, conseguimos de uma forma geral desempenhar a tarefa que nos foi proposta com êxito. Acreditamos ainda que este projeto foi útil no desenvolvimento das nossas capacidades de programação, especialmente na linguagem C++.