Informação e Codificação

Projeto 2

Univerisade de Aveiro

Departamento de Electrónica, Telecomunicações e Informática

Ana Rosa (98678), Sara Gonçalves (98376), Julia Abrantes (104170)

https://github.com/CF2001/IC_project2

2022/2023

Conteúdo

Lista de Figuras

Capítulo 1

Introdução

Este relatório visa descrever a resolução do projeto 2, no âmbito da unidade curricular de Informação e Codificação.

O software desenvolvido encontra-se disponível em: https://github.com/CF2001/IC_project2.

No diretório onde se encontram os ficheiros desenvolvidos, encontra-se um ficheiro README a exemplificar como executar todo o código desenvolvido. Além disso, foram adicionados diretórios onde se encontram os resultados dos testes para cada exercício.

Neste projeto foi usado a biblioteca OpenCV para manipulação dos píxeis das imagens.

Capítulo 2

Parte I

2.1 Exercício 1

Este exercício tem como objetivo copiar uma imagem de um ficheiro para outro, píxel a píxel. Nesse sentido foi criado o ficheiro ex1.cpp onde, percorrendo todas as linhas e colunas da imagem original, copiamos píxel a píxel todos os canais **RGB**, para o ficheiro de cópia.

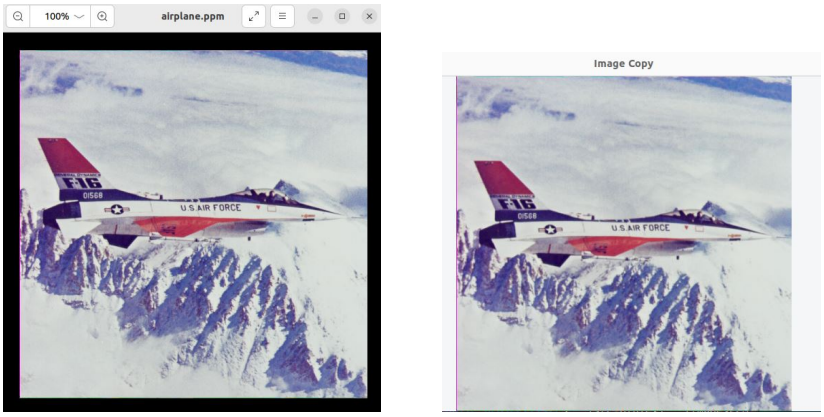

Figura 2.1: Imagem original e imagem copiada, respetivamente

2.2 Exercício 2

No exercício 2 foi criado o ficheiro ex2.cpp que serve para testar as funções desenvolvidas no ficheiro FilterType.h. Este programa foi dividido em casos, conforme as alíneas apresentadas.

- Caso 1 (**Negative Version**):

  O objetivo é criar a versão negativa de uma imagem. A seguinte expressão foi usada:

  $$(255 - R, 255 - G, 255 - B)$$

  A expressão subtrai ao valor 255, que representa o valor máximo de cada canal, o valor do canal respetivo de cada píxel da imagem original.

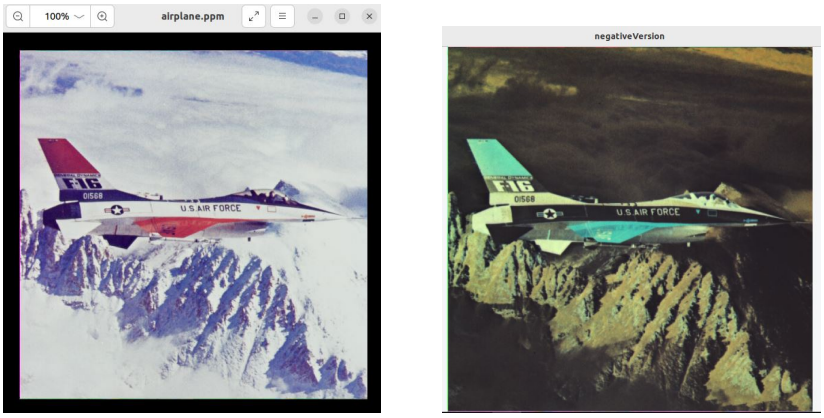

Figura 2.2: Imagem original e imagem negativa, respetivamente

- Caso 2 (**Mirrored Horizontal Version**):

  O objetivo passa por criar uma versão espelhada horizontalmente da imagem original. Para isso, acedemos a cada canal RGB e a partir daí invertemos as colunas.

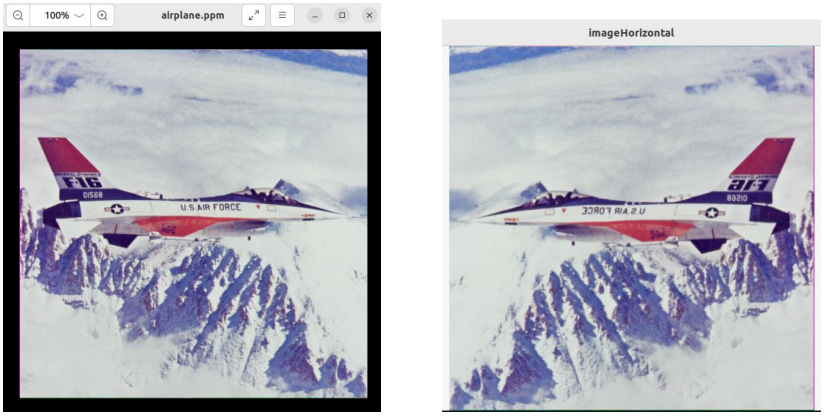

Figura 2.3: Imagem original e imagem espelhada horizontalmente, respetivamente

- Caso 3 (**Mirrored Vertical Version**):

  O objetivo passa por criar uma versão espelhada verticalmente da imagem original. Para isso, acedemos a cada canal RGB e a partir daí invertemos as linhas.

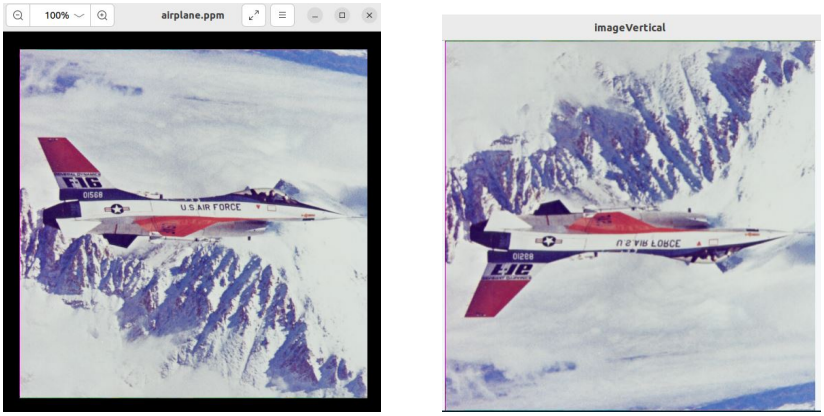

Figura 2.4: Imagem original e imagem espelhada verticalmente, respetivamente

- Caso 4 (**Rotated Version**):

  O objetivo é rodar a imagem original por um múltiplo de 90 graus. Isso é possível criando, primeiramente, uma matriz, através da função fornecida pela biblioteca OpenCV **"getRotationMatrix2D()"**. Esta função tem como argumentos:

  - **center**: centro da imagem original, obtido através da expressão:
    Point2f center((imageIn.cols - 1)/2.0 , (imageIn.rows - 1)/2.0);
  - **angle**: ângulo de rotação, pedido no terminal
  - **scale**: valor de 1.0 por forma ao tamanho da imagem final ser igual ao tamanho da imagem original

  De seguida, outra função fornecida pelo OpenCV usada foi **"warpAffine()"** que aplica uma transformação afim à imagem. Esta função tem como argumentos:

  - a imagem original
  - a imagem final
  - a matriz criada
  - o tamanho da imagem original

  Aqui é apresentado um exemplo de uma rotação para a direita 270º.

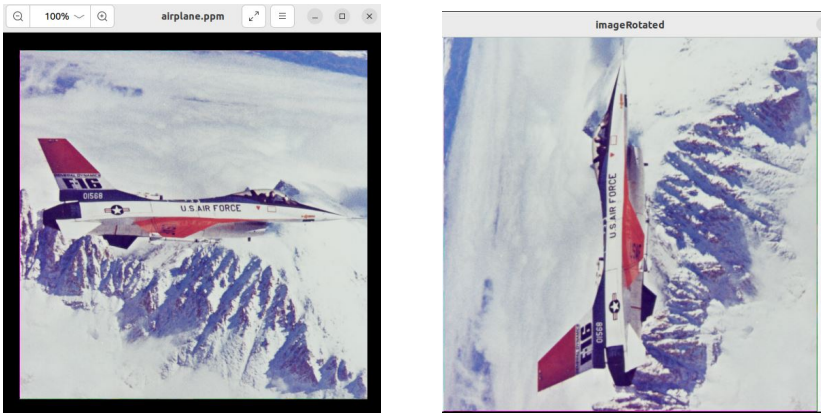

Figura 2.5: Imagem original e imagem rodada, respetivamente

- Caso 5 (**Brighter Version**):

  O objetivo é aumentar o valor da intensidade da imagem original, ou seja, aumentar o brilho. A seguinte expressão foi usada:

  $$imageOut(y, x) = imageIn(y, x) + beta$$

  Onde beta é o valor da intensidade pedido no terminal.

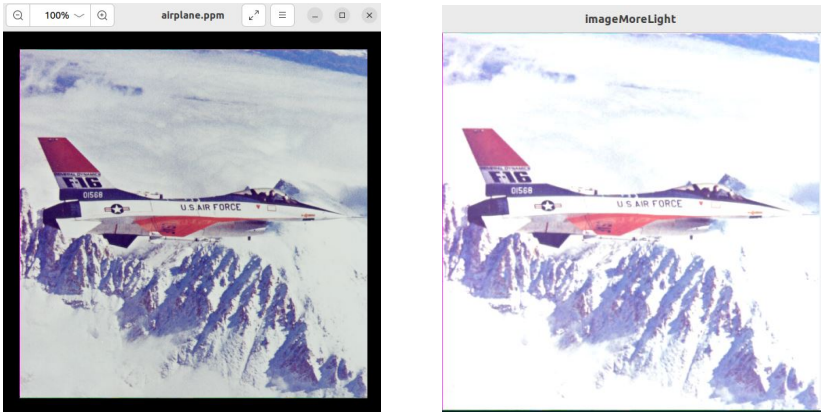

Figura 2.6: Imagem original e imagem com mais brilho, respetivamente

- Caso 6 (**Darker Version**):
  O objetivo é diminuir o valor da intensidade da imagem original, ou seja, diminuir o brilho. A seguinte expressão foi usada:

$$imageOut(y, x) = imageIn(y, x) - beta$$

  Onde beta é o valor da intensidade pedido no terminal.

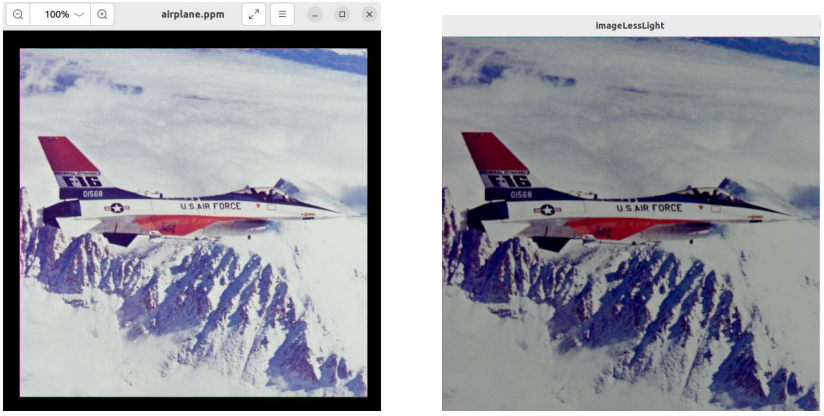

Figura 2.7: Imagem original e imagem com menos brilho, respetivamente

Capítulo 3

Parte II - Golomb

Para a realização da classe de Golomb foi desenvolvido o ficheiro Golomb.h e para o seu teste o ficheiro Golomb.cpp.

O ficheiro Golomb.h é composto pelas seguintes funções:

- **Golomb(char *fname, int M)**

  Construtor que permite escolher o parâmetro m de golomb e indicar o nome do ficheiro binário para inserir os valores codificados resultantes da função encoder.

- **void m(int M)**

  Função que permite reatribuir um novo valor ao parâmetro m de Golomb.

- **string mostSigtBit_to_leastSigBit(string binTmp)**

  Função que coloca um conjunto de bits do mais significativo para o menos significativo.

- **string positiveInt_to_binary(int quotient, int remainder)**

  Função que converte um número inteiro positivo para binário, representando os bits do mais significativo para o menos significativo.

- **int binary_to_integer(vector<int> binary, int b)**

  A função converte um valor binário em inteiro.

- **string unaryCode(int quotient)**

  Função que constrói o código unário para a codificação de Golomb.

- **string paddingBinaryCode(int nBits, int binaryCodeBits, string binaryC)**

  Função que adiciona um conjunto de '0' bits caso o código binário da codificação de Golomb não contenham o tamanho de bits necessários para a sua representação.

- **int fold_n(int n)**

  Converte um número positivo num número par ou um número negativo num número impar.

- **int unfold_n(int n)**

  Desconverte o número par num número positivo ou um número impar num número negativo.

- **string encoder(int n)**

  Função que aplica a codificação de Golomb

- **encoder_writeToBinFile (string codeWord)**

  Função que escreve os valores codificados, resultantes da função de Golomb, num ficheiro binário. Além disso, aplica padding dos bits caso o tamanho a ser escrito no ficheiro binário não seja múltiplo de 8.

- **vector<short> decoder(vector<int> &codeWords)**

  Função que descodifica os valores resultantes da codificação de Golomb.

- **int getOptimalm(const vector<short> samples)**

  Função que determina o m ideal para a codificação de Golomb no processo de compressão.

O código de Golomb é uma família de códigos que dependem de um parâmatro inteiro, m > 0. O código de Golomb baseia-se em separar um número inteiro, $n \geq 0$, em duas partes: a parte unária e a parte binária.

A parte **unária** é determinada da seguinte forma:

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

A parte **binária** é determinada da seguinte forma:

$$r = n - qm$$

O quociente, q, pode ter os valores 0,1,2,..., e é representado pela correspondente parte unária. O resto da divisão, r, pode ter os valores 0,1,2,...,m-1, e pode ser representado pela correspondente parte binária.

O codificador de Golomb apenas aceita números positivos. No sentido de se poder codificar números negativos, foi desenvolvida a função **fold_n** já referida. Esta função é sempre usada no início da função **encoder** e só após a sua chamada é que se determinam o quociente e o resto da divisão.
Para o desenvolvimento da função **encoder** teve de se ter em consideração se o m de Golomb é ou não uma potência de 2.

Caso m seja uma potência de 2, o valor codificado tem de ser determinado da seguinte forma:

- Definir $b = \lceil \log_2 m \rceil$
- Determinar o código binário correspondente ao resto da divisão, r, através da função **positiveInt_to_binary**
- Verificar se o tamanho do código binário é igual a b bits. Se sim, a parte binária fica determinada. Se não, é necessário acrescenter os bits que faltam através da função **paddingBinaryCode**.
- No final, concatenar o valor da parte unária, q, com a parte binária, r.

Caso m não seja uma potência de 2, o valor codificado tem de ser determinado da seguinte forma:

- Definir $b = \lceil \log_2 m \rceil$
- Definir $nValuesR = 2^b - m$
- Se $r < nValuesR$, redefine-se o valor de b bits para $b = \lfloor \log_2 m \rfloor$. Desta forma, a parte binária é representada com menos um bit. Após isso, determina-se o valor binário de r e caso a parte unária não contenha b-1 bits é necessário acrescentar os bits finais.
- Se $r > nValuesR$, um novo valor de r tem de ser determinado. O valor de r passa a ser $newR = r + 2^b - m$. Após isso, determina-se a parte binária e caso contenha um tamanho inferior aos b bits adiciona-se os bits necessários.
- No final, concatenar o valor da parte unária, q, com a parte binária, r.

A função de codificação encontra-se disponibilizada na Figura 3.1

Para o processo de descodificação foi necessário também ter em conta se o m de Golomb é ou não uma potência de 2. Além disso, ao determinar o valor descodificado, no final, foi necessário aplicar a função **unfold_n** , podendo o valor final ser positivo ou negativo.

Independentemente se o m é potência de 2 ou não, a parte unária é determinada da seguinte forma:

- Ao percorrer o valor codificado, determina-se a quantidade de vezes que o bit é igual a 1 até aparecer o primeiro bit a 0. A quantidade de bits a 1 será o valor do quociente. Este método pode ser verificado na Figura 3.2, da linha 19 à 24.

Caso o m seja potência de 2:

- Definir $b = \lceil \log_2 m \rceil$
- Após a leitura dos bits da parte unária, ler os b bits seguintes.
- Determinar o resto da divisão a partir dos bits lidos através da função **binary_to_integer**.
- Determinar o número inteiro positivo através da expressão $n = q * m + r$.
- Aplicar a função **unfold_n** ao valor de n.

Caso o m não seja potência de 2:

- Definir $b = \lceil \log_2 m \rceil$
- Após a leitura dos bits da parte unária, ler os b-1 bits seguintes.
- Determinar o resto da divisão , r, a partir dos bits lidos através da função **binary_to_integer**.
- Se $r < (2^b - m)$, então a representação da quantidade de bits da parte binária é dada pelo r. De seguida, determina-se o valor de n através da expressão $n = q * m + r$ e aplica-se o unfold a n.
- Se $r > (2^b - m)$, então a quantidade de bits da parte binária tem de ser b bits. Logo, é necessário ler o bit seguinte aos b-1 bits lidos inicialmente. Após a leitura do bit determina-se um novo r e obtem-se o valor inteiro do código binário. Por fim, determina-se n através da expressão $n = q * m + r$ e aplica-se o unfold a $n - (2^b - m)$

A função de descodificação encontra-se disponibilizada na Figura 3.2

Na Figura 3.3, é disponibilizado os resultados obtidos ao executar o código do ficheiro Golomb.cpp para m = 10.

```
string encoder(int n)
	{
		n = fold_n(n);
		int q = floor(n/mGolomb);	// quotient -- unary Code
		int r = n - q*mGolomb;		// remainder -- binary Code

		string unaryC = unaryCode(q);
		string binaryC {};
		string codeWord { }; // unaryCode + binaryCode

		int nBits = ceil(log2(mGolomb));	// // the number of bits that represent the binary code
		int nValuesR = pow(2,ceil(log2(mGolomb))) - mGolomb; // values of r (r -> 0,1,m-1)

		if ((mGolomb & (mGolomb-1)) == 0)	// mGolomb is a power of 2
		{
			binaryC = positiveInt_to_binary(r,r%2);
			// Garantir que binaryC para a potencia de 2 tenha exatamente 'nBits' bits
			if ((int)binaryC.size() < nBits)
			{
				binaryC = paddingBinaryCode(nBits, binaryC.size(), binaryC);
			}
		}else{			// mGolomb is not a power of 2

			// 1º - Determine the first values of r (0,1,m-1)
			if (r < nValuesR)
			{
				// The binary code must have 'nBits' of representation
				nBits = floor(log2(mGolomb));  // Para potencias != 2 tem-se b-1 bits
				binaryC = positiveInt_to_binary(r, r%2);

				// The first nValuesR values of r are represented with nBits.
				// If binaryCode does not have the sufficient number
				//   of bits it is necessary to add zeros behind.
				if ((int)binaryC.size() < nBits)
				{
					binaryC = paddingBinaryCode(nBits, binaryC.size(), binaryC);
				}
			}else{
				int newR = r + pow(2, nBits) - mGolomb;
				binaryC = positiveInt_to_binary(newR, newR%2);

				// If binaryCode does not have the sufficient number
				//   of bits it is necessary to add zeros behind.
				if ((int)binaryC.size() < nBits)
				{
					binaryC = paddingBinaryCode(nBits, binaryC.size(), binaryC);
				}
			}
		}
		codeWord = unaryC + binaryC;

		return codeWord;
	}
```

Figura 3.1: Função desenvolvida para a codificação de Golomb

```
vector<short> decoder(vector<int> &codeWords)
	{
		int q { 0 };
		int r { 0 };
		int n { 0 };
		vector<short> samples;

		int b = ceil(log2(mGolomb));
		vector<int> binary;
		bool unaryC = true;
		int readBits_powerTwo = b;
		int readBits_notPowerTwo = b-1;

		int bit0_unaryC = 0;
		int auxCount = 0;

		for (size_t bit = 0; bit < codeWords.size(); bit++)
		{
			if (codeWords[bit] == 1 && unaryC == true)
			{
				q++;
				bit0_unaryC = q;	// fixo
				auxCount = q;
			}else{
				unaryC = false;
				auxCount++;	// Se auxCout++

				if (auxCount != (bit0_unaryC+1))
				{
					// mGolomb is a power of 2
					if ((mGolomb & (mGolomb-1)) == 0)
					{
						binary.push_back(codeWords[bit]);	// ler b bits !!!
						readBits_powerTwo--;

						if (readBits_powerTwo == 0)
						{
							r = binary_to_integer(binary, b);

							n = q * mGolomb + r;
							n = unfold_n(n);
							samples.push_back(n);

							// Initialize all values
							unaryC = true;
							readBits_powerTwo = b;
							bit0_unaryC = 0;
							auxCount = 0;
							binary.clear();
							q = 0;
							r = 0;
							n = 0;
						}
					}else{
						// ler b-1 bits !!!
						binary.push_back(codeWords[bit]);
						readBits_notPowerTwo--;

						if (readBits_notPowerTwo == 0)
						{
							r = binary_to_integer(binary, b-1);

							/**
							* pow(2,b)-mGolomb: representa os primeiros valores de r que
							*		contem b bits de representacao
							*/
							if (r < (pow(2,b)-mGolomb))
							{
								n = q * mGolomb + r;
								n = unfold_n(n);
								samples.push_back(n);

								unaryC = true;
								readBits_notPowerTwo = b-1;
								bit0_unaryC = 0;
								auxCount = 0;
								binary.clear();
								q = 0;
								r = 0;
								n = 0;
							}
						}else if (readBits_notPowerTwo == -1)  // If we have to read +1 bit
						{
							r = 0; // r passa a ser diferente
							r = binary_to_integer(binary, b);

							n = mGolomb*q + r;
							n = unfold_n(n - (pow(2, b) - mGolomb));
							samples.push_back(n);

							unaryC = true;
							readBits_notPowerTwo = b-1;
							bit0_unaryC = 0;
							auxCount = 0;
							binary.clear();
							q = 0;
							r = 0;
							n = 0;
						}
					}
				}
			}
		}
		return samples;
	}
```


Figura 3.2: Função desenvolvida para a descodificação de Golomb

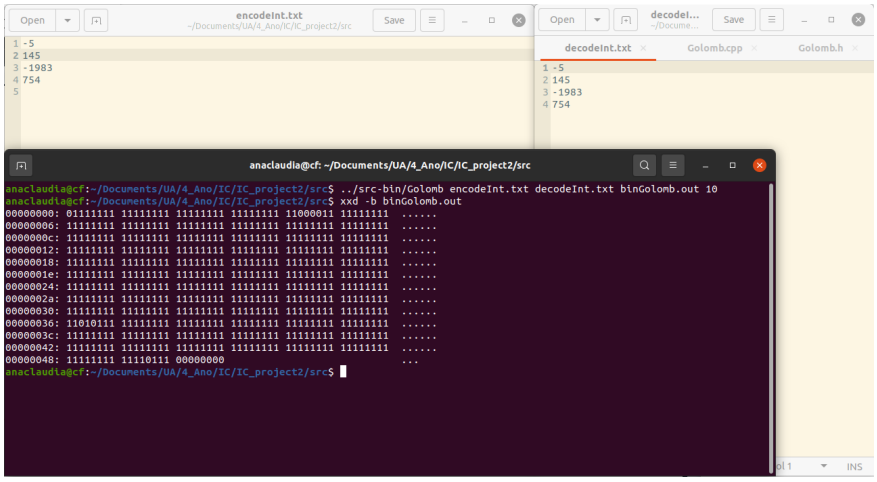

Figura 3.3: Resultados obtidos da classe de Golomb para m = 10

Capítulo 4

Parte III - Audio Codec

Para esta parte foram desenvolvidos os ficheiros Predictor.h, AudioCodec.h e AuxiliaryFunc.h. Para o teste dos áudios codec foi desenvolvido o ficheiro audioCodec.cpp.

No ficheiro Predictor.h, as seguintes funções foram desenvolvidas:

- **Predictor(int order)**

  Construtor que tem como parâmetro a ordem do preditor a ser usada.

- **vector<short> predLossless_separateChannels(const vector<short> &origSamples)**

  Função que a aplica o preditor para o lossless audio através dos canais separados.

- **vector<short> predLossy_separateChannels(const vector<short> &origSamples, int quantization_factor)**

  Função que a aplica o preditor para o lossy audio através dos canais separados.

- **vector<short> reversePred_separateChannels(vector<short> &residualSamples)**

  Função que reverte as amostras residuais para as amostra originais, através dos canais separados.

- **vector<short> predLossless_joinChannels(const vector<short> &origSamples)**

  Função que a aplica o preditor para o lossless audio através dos canais juntos.

- **vector<short> predLossy_joinChannels(const vector<short> &origSamples, int quantization_factor**

  Função que a aplica o preditor para o lossy audio através dos canais juntos.

- **vector<short> reversePred_joinChannels(vector<short> &residualSamples)**

  Função que reverte as amostras residuais para as amostra originais, através dos canais juntos.

- **vector<short> predLossless_MidSideChannels(const vector<short> &origSamples)**

  Função que a aplica o preditor para o lossless audio através da transformação MID + SIDE dos canais.

- **vector<short> predLossy_MidSideChannels(const vector<short> &origSamples, int quantization_factor)**

  Função que a aplica o preditor para o losst audio através da transformação MID + SIDE dos canais.

- **vector<short> reversePred_MidSideChannels(vector<short> &residualSamples)**

  Função que reverte as amostras residuais para as amostra originais, através da transformação MID + SIDE dos canais.

No ficheiro AudioCodec.h, as seguintes funções foram desenvolvidas:

- **AudioCodec(const SndfileHandle& sfh)**

  Construtor que tem como parâmetro a referência ao ficheiro de áudio original que se quer aplicar a compressão.

- **string encode_HeaderOfAudioFile(int typeAudioCodec)**

  Função que codifica em binário os valores do ficheiro de áudio que são colocados no cabeçalho do ficheiro binário.

- **void compress(char *fname, const vector<short> &origSamples, int typeAudioCodec, int predType, int quantization_factor)**

  Função que codifica, através da codificação de golomb, todas as amostras residuais resultantes do preditor. Além disso, adiciona os valores codificados do ficheiro de áudio e das amostras residuais ao ficheiro binário. Sendo que o ficheiro binário contém todos os valores comprimidos.

- **vector<int> decode_HeaderOfAudioFile(vector<int> audioHeader)**

  Função que descodifica o cabeçalho do ficheiro binário.

- **void decompress(char *compressFile, char *wavOutputFile, int predType)**

  Função que descodifica todos os dados após o cabeçalho do ficheiro binário. Após a descodificação das amostras residuais aplica o reverso do preditor para obter as amostras originais. No final, reconstrói o ficheiro de áudio através de todos os valores obtidos da descompressão.

No ficheiro AuxiliaryFunc.h, as seguintes funções foram desenvolvidas:

- **string positiveInt_to_binary(int quotient, int remainder)**

  Função que converte um número inteiro positivo em binário. O valor binário é representado do bit menos significativo para o mais significativo.

- **int binary_to_integer(vector<int> binary, int b)**

  Determina um número inteiro a partir de um valor binário começando no peso do bit menos significativo.

- **string paddingBits(string binary, int padding)**

  Função que adiciona bits '0' num valor binário, em função do parâmetro padding.

4.1 Preditor

O preditor polinomial usado é o seguinte:

$$\begin{cases} \hat{x}_n^{(0)} = 0 \\ \hat{x}_n^{(1)} = x_{n-1} \\ \hat{x}_n^{(2)} = 2x_{n-1} - x_{n-2} \\ \hat{x}_n^{(3)} = 3x_{n-1} - 3x_{n-2} + x_{n-3} \end{cases} \tag{4.1}$$

No processo de codificação, após a escolha da ordem do preditor determina-se $\hat{x}_n$ e após esse cálculo determina-se as amostras residuais:

$$r_n = x_n - \hat{x}_n \tag{4.2}$$

É de ter em conta que, no lossless e lossy audio codec, a predição das amostras é feita em canais separados. Isto é, a predição é realizada no canal 0 separadamente do canal 1.
No caso do lossless audio, após a determinação de $\hat{x}_n$ para cada canal, determina-se r_n separadamente para cada canal e adiciona-se cada resultado ao vetor de amostras residuais.
No caso do lossy audio, o mesmo método é usado. No entanto, antes de adicionar o valor da amostra residual ao vetor é necessário aplicar a redução dos bits de resolução da amostra em função do pârametro quantization_factor que é disponibilizado na função predLossy_separateChannels. De seguida, é necessário atualizar o valor da amostra original para efetuar o sincronismo do erro.

No processo de descodificação, aplica-se a seguinte fórmula:

$$x_n = r_n + \hat{x}_n \tag{4.3}$$

, sendo $\hat{x}_n$ o valor dado pelo preditor e que tem de ser igual ao estimado no codificador.

O reverso do preditor é aplicado tanto no lossless audio como no lossy de forma igual.
No caso do lossy, ao efetuar a descodificação das amostras residuais, de seguida é efetuado um deslocamento para a esquerda em função do fator de quantização, no sentido de a amostra residual ser atualizada. E só após a amostra residual ser descodificada e atualizada é que se efetua o reverso do preditor.

4.2 Lossless e Lossy Codec

No processo de codificação, tanto do lossless e lossy audio, o ficheiro binário com os dados comprimidos é criado com o seguinte formato:

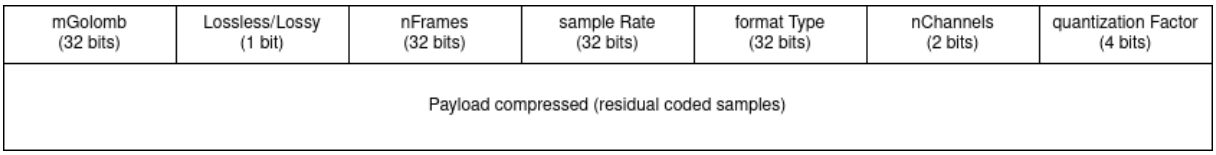

Figura 4.1: Formato dos dados codificados no ficheiro binário, resultantes da compressão.

O processo de compressão, implementado na função **compress** é o seguinte:

- Se o tipo de áudio codec for lossless, as amostras residuais são determinadas através do preditor implementado pela função **predLossless_separateChannels**. Caso o áudio seja lossy, a função que implementa o preditor lossy é a **predLossy_separateChannels**.
- Após a determinação das amostras residuais, determina-se o valor de m de Golomb ideal para codificar estas amostras através da codificação de Golomb. O valor de m é obtido através da função **getOptimalm** referida na secção anterior de Golomb.
  A função **getOptimalm** aplica o fold das amostras residuais e em função disso determina a média aritmética desses valores:

$$mean = \frac{\sum fold\,(r_n)}{N} \tag{4.4}$$

De seguida, determina o valor de m ideal:

$$m = \left\lceil \frac{-1}{log_2(\frac{mean}{mean+1})} \right\rceil \tag{4.5}$$

Ao aplicar a média aritmética através das amostras residuais, garante-se que o valor de m seja mais fiável do que ao ser calculado pelas amostras originais. No final, o m é codificado em binário e caso não tenha 32 bits é lhe acrescentado os bits em falta.

- Após a codificação do m, são codificados os valores provenientes da informação do ficheiro de áudio wav. Assim, a função **encode_HeaderOfAudioFile** codifica em binário o seguinte: bit de indicação se o áudio codec é losseless ou lossy, o número de frames, a frequência de amostragem, o formato do ficheiro de áudio, o número de canais e o fator de quantização. É de realçar que, se o tipo de codec for lossless por defeito o campo do fator de quantização é 0 (0000 em binário), caso contrário é diferente de 0.

- Após a codificação do cabeçalho do ficheiro binário, os valores das amostras residuais são codificadas através da função **encoder** de Golomb. Por fim, juntam-se todos os valores codificados e insere-se esses valores no ficheiro binário.

O processo de descompressão, implementado na função **decompress** é o seguinte:

- Lê-se todos os valores do ficheiro binário que contêm o cabeçalho e as amostras residuais codificadas.

- Descodifica-se os valores do cabeçalho de binário para um valor inteiro positivo.

- Se o valor que indica o tipo de áudio codec for 0 (lossless), então as amostras residuais codificadas são descodificadas diretamente através da função **decoder** de Golomb. Caso o tipo de áudio codec for 1 (lossy), então as amostras residuais codificadas são também descodificadas através do decoder de Golomb, mas de seguida é necessário aplicar um deslocamento para a esquerda através do fator de quantização também obtido na descodificação para binário.

- Após as amostra residuais serem obtidas, determina-se as amostras originais através da função **reversePred_separateChannels**.

- Por fim, reconstrói-se o ficheiro de áudio através do número de frames, frequência de amostragem, formato, número de canais e as amostras originais.

4.3 Resultados Obtidos

Nesta secção são apresentados os resultados obtidos tanto para o lossless como para o lossy áudio codec. Os resultados são determinados através de três ficheiros de áudio: sample01.wav, sample02.wav e sample03.wav.
Para ambos os áudios serão apresentadas as durações das compressões para os três preditores, a taxa de compressão (Equação 4.4), a entropia original e residual das amostras e ainda os histogramas resultantes das amostras residuais determinadas.

$$TaxaDeCompressao = \frac{FicheiroWavOriginal}{FicheiroWavComprimido} \tag{4.6}$$

4.3.1 Testes realizados com o preditor 1, 2 e 3 com os canais separados

Os resultados para o **lossless audio codec** são os seguintes:

Na Tabela 4.1 são disponibilizados os resultados para a duração e taxa de compressão.
Relativamente à duração, verifica-se que para a sample01 a compressão leva mais tempo com o preditor 3, no entanto, para a sample02 e sample03 a compressão com o preditor 1 dura mais. Em relação à taxa de compressão, no caso da sample01 e sample03 a utilização do preditor 2 faz com que a taxa seja supeior, sendo que, no caso da sample02, o preditor 1 faz com que a taxa de compressão seja maior. Neste sentido, conclui-se que o preditor 2 permite ter uma melhor taxa de compressão, no geral.

Ficheiro de áudio	Preditores	Duração	Taxa de Compressão
sample01.wav	1	0.8883	1.3578
	2	1.0734	1.4090
	3	1.1444	1.3673
sample02.wav	1	0.447813	1.3848
	2	0.435708	1.3770
	3	0.442638	1.3171
sample03.wav	1	0.571085	1.4662
	2	0.555764	1.5259
	3	0.565663	1.5004

Tabela 4.1: Resultados da duração e taxa de compressão do lossless audio codec

Na Tabela 4.2 é possível observar os resultados da entropia para as amostras originais e após estas passarem pelo processo do preditor (resultando nas amostras residuais).
A entropia original, independentemente do preditor usado, é sempre a mesma. Em relação à entropia residual, quanto menor o valor dessa entropia melhor é o resultado, isto é, menor será o número de símbolos binários necessários para codificar os símbolos do ficheiro de áudio (o que implica uma maior taxa de compressão).

Ficheiro de áudio	Preditores	Entropia Original	Entropia Residual
sample01.wav	1	13.9086	11.7262
	2	13.9086	11.3135
	3	13.9086	11.6489
sample02.wav	1	13.035	11.2859
	2	13.035	11.2006
	3	13.035	11.5980
sample03.wav	1	13.2678	10.786
	2	13.2678	10.3173
	3	13.2678	10.4410

Tabela 4.2: Resultados da entropia original e residual do lossless audio codec

Assim, para todos os ficheiro de áudio verifica-se uma menor entropia ao utilizar o preditor 2. Na Figura 4.2 é disponibilizado os histogramas das amostras origianais das sample01, sample02 e sample03. Na figura 4.3 pode-se observar os histogramas das amostras residuais dos ficheiros de áudio para o preditor 2, onde se obtiveram os melhores resultados na entropia.

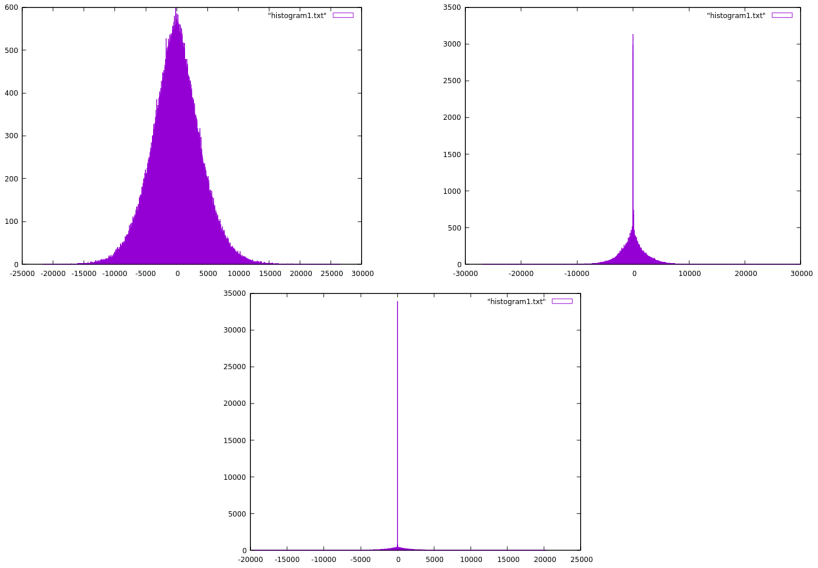

Figura 4.2: Histogramas das amostras originais dos ficheiros de áudio sample01, sample02 e sample03, respetivamente.

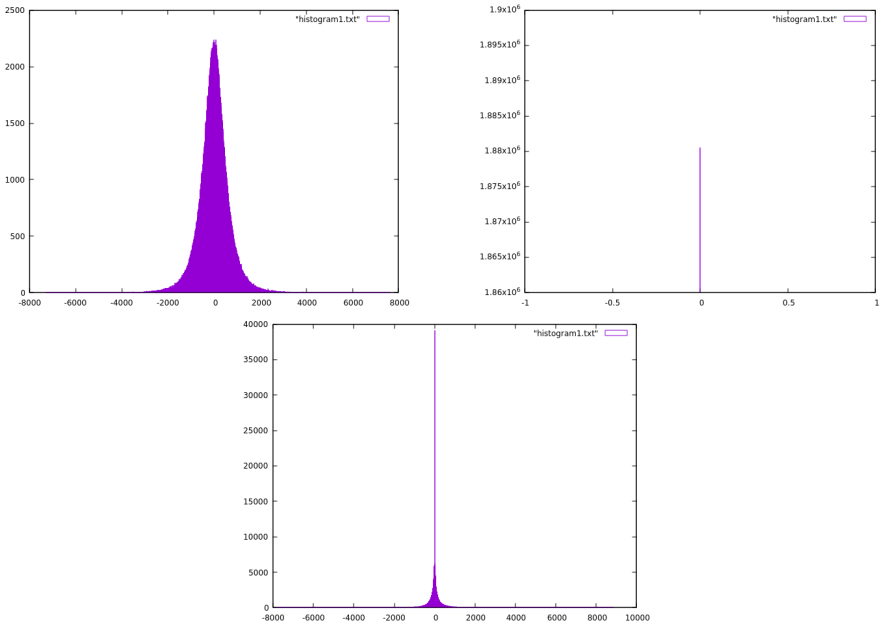

Figura 4.3: Histogramas das amostras residuais resultantes do preditor 2 e do lossless audio codec, dos ficheiros de áudio sample01, sample02 e sample03, respetivamente.

Os resultados para o **lossy audio codec**, com um fator de quantização igual a 3, são os seguintes:

Observando a Tabela 4.3, relativamente à duração da compressão, o caso da sample01 e sample02 demoraram mais tempo com o preditor 3, no entanto a sample03 demorou mais com o preditor 3.
Em relação à taxa de compressão, a sample01 e sample03 obtiveram uma maior compressão com o preditor 2, e a sample02 obteve com o preditor 1 uma maior compressão.

Ficheiro de áudio	Preditores	Duração	Taxa de Compressão
sample01.wav	1	0.7323	1.8213
	2	0.7273	1.9148
	3	0.7518	1.8388
sample02.wav	1	0.3740	1.8704
	2	0.3617	1.8551
	3	0.4134	1.7472
sample03.wav	1	0.4556	2.0214
	2	0.4594	2.1299
	3	0.4496	2.0865

Tabela 4.3: Resultados da duração e taxa de compressão do lossy audio codec

Observando a Tabela 4.4, relativamente à entropia, todos os ficheiros obtiveram uma menor entropia residual com o preditor 2.
Neste sentido, na Figura 4.4 são ilustrados os histogramas obtidos para as amostras residuais resultantes do preditor 2 com um fator de quantização igual a 3. Nessa Figura é possível observar as difenças obtidas após as amostras originais passarem pelo preditor e com uma redução de 3 bits na representação das amostras.

Ficheiro de áudio	Preditores	Entropia Original	Entropia Residual
sample01.wav	1	13.9086	8.7296
	2	13.9086	8.3164
	3	13.9086	8.6530
sample02.wav	1	13.0350	8.3104
	2	13.0350	8.2197
	3	13.0350	8.6226
sample03.wav	1	13.2678	7.8197
	2	13.2678	7.3476
	3	13.2678	7.4755

Tabela 4.4: Resultados da entropia original e residual do lossy audio codec

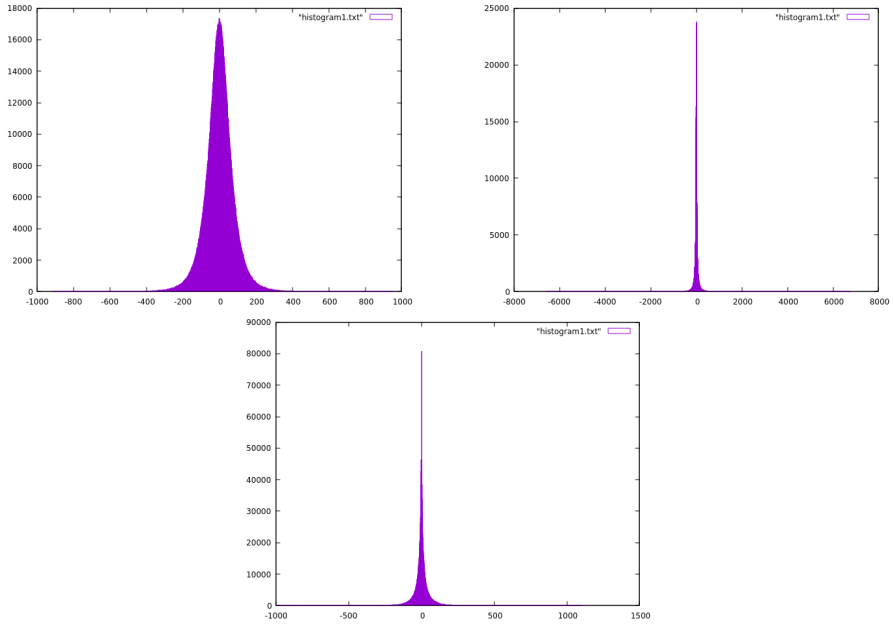

Figura 4.4: Histogramas das amostras residuais resultantes do preditor 2 e do lossy audio codec, dos ficheiros de áudio sample01, sample02 e sample03, respetivamente.

Comparando os resultados obtidos entre o lossless e lossy audio codec, verifica-se que a duração da compressão para o lossless é superior do que no lossy. Relativamente à taxa de compressão, verifica-se que é superior no lossy pelo que a entropia residual torna-se mais baixa no lossy do que no lossless, sendo que o fator de quantização usado no lossy para estes testes foi igual a 3.
Conclui-se, em função dos testes realizados e dos ficheiros de áudio usados, que o processo de compressão é melhor ao utilizar o lossy audio codec.

4.3.2 Testes realizados com o preditor 1, 2 e 3 com os canais juntos

Os resultados para o **lossless audio codec** são os seguintes:

Ficheiro de áudio	Preditores	Duração	Taxa de Compressão
sample01.wav	1	0.9766	1.2376
	2	1.0749	1.1513
	3	1.0927	1.0747
sample02.wav	1	0.5150	1.1617
	2	0.5468	1.0842
	3	0.5775	1.0208
sample03.wav	1	0.6826	1.2535
	2	0.6854	1.1639
	3	0.7291	1.0854

Tabela 4.5: Resultados da duração e taxa de compressão do lossless audio codec

Ficheiro de áudio	Preditores	Entropia Original	Entropia Residual
sample01.wav	1	13.9086	12.8763
	2	13.9086	13.8355
	3	13.9086	14.8162
sample02.wav	1	13.0350	13.6776
	2	13.0350	14.6380
	3	13.0350	15.4874
sample03.wav	1	13.2678	12.6443
	2	13.2678	13.6019
	3	13.2678	14.5652

Tabela 4.6: Resultados da entropia original e residual do lossless audio codec

Em função dos dados observados na Tabela 4.5 e 4.6, ao comparar com os dados da Tabela 4.1 e 4.2, verifica-se que a duração do processo de compressão é superior, a taxa de compressão é mais baixa e os valores das entropias residuais consequentemente são mais altas. A entropias das amostras residuais ao serem mais altas e superiores aos valores das entropias originais faz com que não seja realizada uma boa compressão.

Os resultados para o **lossy audio codec**, com um fator de quantização igual a 3, são os seguintes:

Ficheiro de áudio	Preditores	Duração	Taxa de Compressão
sample01.wav	1	0.8445	1.6116
	2	0.8607	1.4683
	3	0.9357	1.3456
sample02.wav	1	0.4436	1.4852
	2	0.5036	1.3608
	3	0.4990	1.2613
sample03.wav	1	0.5605	1.6386
	2	0.5781	1.4888
	3	0.6370	1.3627

Tabela 4.7: Resultados da duração e taxa de compressão do lossy audio codec

Ficheiro de áudio	Preditores	Entropia Original	Entropia Residual
sample01.wav	1	13.9086	9.8821
	2	13.9086	10.8454
	3	13.9086	11.8346
sample02.wav	1	13.0350	10.6977
	2	13.0350	11.6726
	3	13.0350	12.5653
sample03.wav	1	13.2678	9.69565
	2	13.2678	10.6635
	3	13.2678	11.6406

Tabela 4.8: Resultados da entropia original e residual do lossy audio codec

Em função dos dados das Tabelas 4.7 e 4.8, ao comparar com os dados das Tabelas 4.3 e 4.4, verifica-se que a duração continua a ser mais elevada e a taxa de compressão também mais baixa o que implica maiores valores obtidos nas entropias residuais determinadas. A pesar de o valor das entropias residuais serem mais altas, estão, ao contrário dos valores do lossless audio referido em cima, a baixo dos valores das entropias originais. Conclui-se, assim, que o lossy é a melhor escolha ao utilizar o preditor 1, 2 e 3 com os canais juntos.

4.3.3 Conclusão final dos testes

Em função do estudo dos preditores 1, 2 e 3 para os canais separados e juntos, conlcui-se que a implementação para os canais separados faz com que a predição dos valores, e por sua vez o processo de compressão, seja melhor. Além disso, verificou-se que independentemente dos preditores usados o lossy audio codec é mais eficinete para a compressão de ficheiro de áudio.

4.3.4 Notas da implementação dos preditores

Para a realização dos testes desta secção foram realizados os preditores 1,2 e 3 dos canais separados, juntos e ainda da transformação MID + SIDE. No entanto, apenas os canais separados e juntos estavam a funcionar corretamente. A pesar disso, a implementação da transformação dos canais MID + SIDE está disponível no ficheiro Predictor.h tanto para o lossless como para o lossy audio codec.

Capítulo 5

Contribuição dos Autores

Neste projeto, os participantes foram apenas a Sara Gonçalves e a Ana Rosa. Sendo 50% de participação para ambas e 0% para Júlia Abrantes.