Projeto No. 3 - Video Encoding

Universidade de Aveiro

Licenciatura em Engenharia Informática

UC 40532 - Complementos Sobre Linguagens de Programação

Aveiro, 6 de janeiro de 2020

Autores

- **89066 - Luís Carlos Duarte Fonseca:** luis99fonseca
- **88886 - Tiago Carvalho Mendes:** tiagocmendes

Docente orientador

Prof. Dr. António José Ribeiro Neves

1. Introdução

O presente documento tem como principal objetivo descrever a solução desenvolvida pelos alunos identificados neste documento no âmbito do terceiro projeto da unidade curricular de Complementos Sobre Linguagens de Programação da Universidade de Aveiro. Os principais objetivos deste terceiro projeto eram os seguintes:

- Implementar a classe **BitStream**, capaz de ler e escrever num ficheiro um determinado número de bits.
- Implementar a classe **Golomb**, capaz de codificar um número inteiro ***n*** utilizando códigos de Golomb.
- Implementar um **leitor de vídeo**.
- Implementar **(des)codificadores de vídeo**, baseados em codificação intra-frame **sem perdas**, usando 7 JPEG preditores lineares e o preditor não linear JPEG-LS.
- Implementar todos os objetivos anteriores nas linguagens de programação **Python** e **C++**.
- Documentar todo o código desenvolvido utilizando o **Doxygen.**

Uma vez que os pontos referidos são bastante genéricos, é importante salientar que foram definidos outros sub-objetivos mais simples, necessários à realização das tarefas principais.

Embora, conceptualmente, a mesma lógica tenha sido pensada e implementada em ambas as linguagens, existem obviamente algumas diferenças importantes entre as mesmas. Desta forma, ao explicar cada direção seguida na realização das tarefas propostas iremos, por questões de simplicidade, dar mais foco à implementação das mesmas em **Python** e, de seguida, de forma mais sucinta em **C++**.

Importante também referir que os videos em questão se encontram no formato YUV, podendo ter diferentes tipos de sub-amostras de cronominancia ([4:4:4], [4:2:2], [4:2:0])

2. Repositório

Todo o código desenvolvido encontra-se disponível no seguinte repositório, na plataforma **GitHub**: https://github.com/luis99fonseca/video-encoding

Algumas considerações:

- Na raiz do repositório, encontram-se duas pastas, cada uma com a solução desenvolvida em **C++** e em **Python**, respetivamente.
- Para cada uma das linguagens, foi gerada documentação automática em **HTML** utilizando o **Doxygen**, dentro da pasta **docs**.
- Para cada uma das linguagens, existe uma pasta **src**, com os ficheiros de código desenvolvidos.
- Dentro de cada pasta **src**, existem duas sub-pastas com os testes unitários das classes **BitStream e Golomb**.
- Para cada uma das linguagens, existe uma pasta **out** que irá ser o destino dos ficheiros de vídeo comprimidos.
- Para a linguagem de **C++**, e na pasta **src**, está presente um **Makefile** para compilar o codificador de vídeo **VideoEncoder.cpp**. Este ficheiro foi compilado utilizando o **cmake.** No entanto, também poderá encontrar o ficheiro binário resultante desta compilação (**VideoEncoder**).
- **IMPORTANTE:** Para testar o vídeo **park_joy_444_720p50.y4m**, tanto para **C++** como para **Python**, deverá colocá-lo na pasta **python/media**.

3. Implementação

3.1. **BitStream**

3.1.1 Python 3

Esta classe é responsável por tratar do *streaming* de dados entre o programa e o sistema de ficheiros de forma binária. Desta forma, os métodos da mesma envolvem não só funcionalidades mais genéricas como **leitura/escrita de N bits** bem como funcionalidades mais específicas como **escrita/leitura** de strings/linhas.

```
class BitStream:
    """
    Class optimised to read/write bits from/to a file
    """
    def __init__(self, fileName, mode):
        # file management
        assert mode in ["wb", "rb", "wbs"]
        self.mode = mode
        self.file = open(fileName, self.mode[:2])

        self.read_byte = None  # buffer
        self.read_byte_idx = -1
        self.read_eof = False

        self.write_byte = 0  # buffer
        self.write_byte_idx = 7

        self.write_array_last = []
```

```
        self.write_array_final = []

        self.closed = False
```

```
# ---------------READING TESTING--------------
if test01:
    bitstream01 = BitStream("./out/test01.txt", "rb")

    assert bitstream01.readBit(8) == [1, 1, 0, 0, 1, 0, 0, 0]
    assert bitstream01.readBit(4) == [0, 1, 1, 1]
    assert bitstream01.readBit(4) == [1, 0, 0, 0]
    assert bitstream01.readByte() == [0, 1, 1, 0, 0, 0, 1, 1]
    assert bitstream01.readBit(8) == []
    assert bitstream01.readBit(4) == []

    bitstream01.closeFile()

# ---------------WRITING TESTING--------------

if test02:
    bitstream02 = BitStream("./out/test02.txt", "wb")

    assert not bitstream02.writeBit(256, 1)
    assert not bitstream02.writeBit(3, 1)
```

Como recomendado, alguns programas de teste foram criados de forma a garantir o bom funcionamento das classes criadas. Estes mesmos programas tornam-se uteis para entender o funcionamento de cada classe.

Observando excerto de código presente no ficheiro **test_bitstream.txt**, é possível perceber como a classe funciona:

Primeiramente, o construtor da mesma recebe o ***nome do ficheiro*** e o ***modo*** em que a classe será usada. Estes *modos* devem ser um dos seguintes:

- **rb** (read binary) - que indica que se irá proceder à leitura do ficheiro indicado no 1º argumento.
- **wb** (write binary) - que indica que se irá proceder à escrita do ficheiro indicado no 1º argumento.
- **wbs** (write binary special) - semelhante ao modo anterior, mas com otimizações, de forma a aumentar a eficiência da escrita. Estas otimizações passam pelo uso de funções de alto nível do **Python**, em troca do aumento de utilização da memória. Estas otimizações surgem devido ao facto de que, sendo o Python ser uma linguagem de alto nível, é portanto mais lento, o que é indesejável no contexto deste trabalho.

Após a definição de um novo objeto desta classe, vários métodos podem ser chamados. Contudo, existem algumas restrições relacionadas com os *modos* definidos. Por exemplo, não é possível utilizar o método de **ler um bit** de um ficheiro quando se definiu, no momento da criação do objeto desta classe, o **modo de escrita**.

Importante: É de notar que para cada modo de escrita, existem métodos específicos opcionais.

- **wb** - *writeBit()*, *writeByte()*, *writeArray()*

- **wbs** - *addNumber()*

Relativamente aos nomes dos métodos de *leitura*, geralmente estes começam com a palavra **"read"**.

Após a realização de todas as operações, é **obrigatório** chamar o método *closeFile()* de fecho do ficheiro. Isto deve ser feito, não só por ser uma boa prática de programação, mas também para efetivar todas as alterações ao mesmo.

3.1.2 C++

A principal diferença entre a implementação em **Python** para a implementação em **C++** desta classe é nos modos de interação com o ficheiro. Em **C++**, apenas existem 2 (o **rb** e o **wb**), uma vez que sendo um linguagem de mais baixo nível, não só não possuí os métodos equivalentes utilizados em **Python**, como apresenta uma maior eficiência.

```
/**
 * Class optimised to read/write bits from/to a file.
*/
class BitStream {
    public:

        char modeF;
        fstream fileF;

        unsigned char read_byte;
        int read_byte_idx;
        bool read_eof;

        unsigned char write_byte;
        int write_byte_idx;

        bool closedF;

        /**
         * Default constructor.
        */
        BitStream();

        /**
         * Constructor with a given file name and a opening mode.
         *
         * @param fileN: file name.
         * @param mode: opening mode.
         */
        BitStream(string fileN, char mode){
```

As restantes alterações foram feitas sobretudo devido às diferenças entre as sintaxes das duas linguagens.

3.2. Golomb

De um modo geral, o principal objetivo desta classe é o de codificar um determinado valor inteiro ***n***, utilizando códigos de Golomb. No âmbito deste trabalho, esta classe é usada para codificar os valores das matrizes de predição de cada frame, como será explicado mais à frente. Utilizando esta codificação de Golomb, iremos conseguir comprimir um determinado vídeo, reduzindo o seu tamanho substancialmente.

3.2.1. Python 3

Começando pela seu construtor (apresentado no seguinte excerto de código do ficheiro **golomb.py**), repare-se que este recebe como argumento um valor inteiro ***m***, que será o divisor de qualquer valor inteiro ***n*** a codificar, atribuindo-o ao atributo **self.m** da classe. Neste construtor, é também criado o atributo **self.base2**, que verifica se o valor de **self.m** é ou não uma potência de base 2. É importante referir isto pois, como se verá de seguida, o algoritmo de codificação de um determinado valor inteiro **n** irá variar consoante **m** seja potência de base 2 ou não (aplicando um código binário truncado em caso negativo).

```
class Golomb:
    """
    The constructor.

    @param m: value of M (default = 2)
    """
    def __init__(self, m=2):
        assert m > 0

        self.m = m

        # Check if M is a power of 2 or not
        self.base2 = True if math.log2(m).is_integer() else False
```

Codificação com *m* sendo potência de base 2

Para implementar o algoritmo de codificação, e caso **self.m** seja uma potência de base 2, foi criado o método que se apresenta no seguinte excerto de código:

```
def base2encoder(self, n):

    # check 'n' sign (0 - positive, 1 - negative)
    sign = [0]
    if n < 0:
        n = abs(n)
        sign = [1]

    # computes values of 'q' and 'r'
    q = math.floor(n / self.m)
    r = n % self.m

    # computes unary and binary codes of 'q' and 'r', respectively.
    unary_code = [1 for i in range(q)] + [0]
    binary_code = [int(i) for i in list(bin(r))[2:]] if r >= 2 else [0] + [int(i)
for i in list(bin(r))[2:]]
```

```
    # returns the golomb code of 'n'
    return  sign + unary_code + binary_code
```

Primeiramente, começa-se por averiguar se o valor **inteiro *n***, passado como argumento do método, é positivo. Em caso afirmativo, coloca-se **o primeiro bit** do seu código de Golomb a **0**. Caso **n** seja negativo, este **primeiro bit** terá o valor **1.** Assim, apenas precisamos de nos preocupar em codificar o valor absoluto de **n**.

Para obter a parte unária e binária do código de Golomb de **n**, é necessário saber os valores de **q** e **r**, respetivamente. No caso de **q**, este obtém-se através do quociente da divisão de **n** por **m** (n // m). No caso de **r**, este é o resto da mesma divisão (n % m).

Tendo o valor **q**, a parte unária é dada por **q** bits a 1, seguidos de um bit a 0. Por exemplo, caso **q = 4**, a parte unária será **11110**.

No caso de **r**, a parte binária é simplesmente o valor de **r** em binário.

Após estes passos, o código de Golomb de **n** é formado pelo sinal, pela parte unária e pela parte binária (nesta ordem), sendo retornado numa lista de **0's** e **1's**.

Para verificar esta implementação, foram realizados os seguintes testes unitários:

```
golomb.m = 4

codes = [[0,0,0,0], [0,0,0,1], [0,0,1,0],\
    [0,0,1,1], [0,1,0,0,0], [0,1,0,0,1],\
    [0,1,0,1,0], [0,1,0,1,1], [0,1,1,0,0,0],\
    [0,1,1,0,0,1], [0,1,1,0,1,0], [0,1,1,0,1,1], \
    [0,1,1,1,0,0,0], [0,1,1,1,0,0,1], \
    [0,1,1,1,0,1,0], [0,1,1,1,0,1,1]]

assert golomb.encode(0) == codes[0]
assert golomb.encode(1) == codes[1]
assert golomb.encode(2) == codes[2]
assert golomb.encode(3) == codes[3]
assert golomb.encode(4) == codes[4]
assert golomb.encode(5) == codes[5]
assert golomb.encode(6) == codes[6]
assert golomb.encode(7) == codes[7]
assert golomb.encode(8) == codes[8]
assert golomb.encode(9) == codes[9]
assert golomb.encode(10) == codes[10]
assert golomb.encode(11) == codes[11]
assert golomb.encode(12) == codes[12]
assert golomb.encode(13) == codes[13]
assert golomb.encode(14) == codes[14]
assert golomb.encode(15) == codes[15]
```

Codificação com *m* NÃO sendo potência de base 2

Para implementar o algoritmo de codificação, e caso **self.m** não seja uma potência de base 2, foi criado o método que se apresenta no seguinte excerto de código:

```
def truncated_encoder(self, n):
    # check 'n' sign (0 - positive, 1 - negative)
    sign = [0]
    if n < 0:
        n = abs(n)
        sign = [1]

    b = math.ceil(math.log2(self.m))
    q = math.floor(n / self.m)
    r = n % self.m

    unary_code = [1 for i in range(q)] + [0]

    first_values = 2**b - self.m

    if r < first_values:
        binary_code = self.decimal_to_binary(r, b - 1)
    else:
        binary_code = self.decimal_to_binary(r + 2**b - self.m, b)

    # return the truncated golomb code of 'n'
    return sign + unary_code + binary_code
```

A diferença deste método para com o método explicado anteriormente está, unicamente, no cálculo da parte binária do código de Golomb do valor **inteiro *n***, passado como argumento. Portanto, para encontrar esta parte binária, primeiramente calcula-se o valor de **b**, que é dado por **b = math.ceil(math.log2(self.m))**.

Tendo o valor de **b**, a parte binária de **n** é dada por:

- Caso **r** seja menor que **2^b - m**, a parte binária é dada por **r** em binário com **b - 1** bits.
- Caso contrário, a parte binária é dada pelo resultado de **r + 2^b - m** em binário, com **b** bits.

No final, o código de Golomb de **n** é retornado numa lista de **0's e 1's**, à semelhança do método anterior. Desta forma, a eficiência da codificação para um valor de **self.m** que não seja uma potência de base 2 aumenta.

Para verificar esta implementação, foram realizados os seguintes testes unitários:

```
golomb.set_m(5)

codes = [[0,0,0,0], [0,0,0,1], [0,0,1,0],\
    [0,0,1,1,0], [0,0,1,1,1], [0,1,0,0,0],\
    [0,1,0,0,1], [0,1,0,1,0], [0,1,0,1,1,0],\
    [0,1,0,1,1,1], [0,1,1,0,0,0], [0,1,1,0,0,1],\
    [0,1,1,0,1,0], [0,1,1,0,1,1,0], \
    [0,1,1,0,1,1,1], [0,1,1,1,0,0,0]\
    [1,0,0,1], [1,0,1,0]]
```

```
assert golomb.encode(0) == codes[0]
assert golomb.encode(1) == codes[1]
assert golomb.encode(2) == codes[2]
assert golomb.encode(3) == codes[3]
assert golomb.encode(4) == codes[4]
assert golomb.encode(5) == codes[5]
assert golomb.encode(6) == codes[6]
assert golomb.encode(7) == codes[7]
assert golomb.encode(8) == codes[8]
assert golomb.encode(9) == codes[9]
assert golomb.encode(10) == codes[10]
assert golomb.encode(11) == codes[11]
assert golomb.encode(12) == codes[12]
assert golomb.encode(13) == codes[13]
assert golomb.encode(14) == codes[14]
assert golomb.encode(15) == codes[15]
assert golomb.encode(-1) == codes[16]
assert golomb.encode(-2) == codes[17]
```

Descodificação

Relativamente à descodificação de uma stream de bits, foram implementados dois métodos: um para descodificação sendo **self.m** potência de base 2, e outro em caso negativo, como se ilustra no seguinte excerto de código:

```
 """
This method decodes a given list of bits, encoded with Golomb encoding.

@param bitstream: list of bits to decode
"""
def base2decoder(self, bitstream): ...


"""
This method decodes a given list of bits, encoded with truncated Golomb encoding.

@param bitstream: list of bits to decode
"""
def truncated_decoder(self, bitstream): ...
```

Resumidamente, estes dois métodos recebem uma **lista de bits como argumento** e, implementando o algoritmo inverso da respetiva codificação, retornam o correspondente valor **inteiro n**. De seguida, seguem-se dois exemplos de testes unitários, com **m=4** e **m=5**, respetivamente:

```
print("\n-------------------------------")
print("SECOND TEST - DECODE WITH M = 4")
print("-------------------------------")
```

```
assert golomb.decode(codes[0]) == 0
assert golomb.decode(codes[1]) == 1
assert golomb.decode(codes[2]) == 2
assert golomb.decode(codes[3]) == 3
assert golomb.decode(codes[4]) == 4
assert golomb.decode(codes[5]) == 5
assert golomb.decode(codes[6]) == 6
assert golomb.decode(codes[7]) == 7
assert golomb.decode(codes[8]) == 8
assert golomb.decode(codes[9]) == 9
assert golomb.decode(codes[10]) == 10
assert golomb.decode(codes[11]) == 11
assert golomb.decode(codes[12]) == 12
assert golomb.decode(codes[13]) == 13
assert golomb.decode(codes[14]) == 14
assert golomb.decode(codes[15]) == 15

print("\n------------------------------")
print("FOURTH TEST - DECODE WITH M = 5")
print("------------------------------")

assert golomb.decode(codes[0]) == 0
assert golomb.decode(codes[1]) == 1
assert golomb.decode(codes[2]) == 2
assert golomb.decode(codes[3]) == 3
assert golomb.decode(codes[4]) == 4
assert golomb.decode(codes[5]) == 5
assert golomb.decode(codes[6]) == 6
assert golomb.decode(codes[7]) == 7
assert golomb.decode(codes[8]) == 8
assert golomb.decode(codes[9]) == 9
assert golomb.decode(codes[10]) == 10
assert golomb.decode(codes[11]) == 11
assert golomb.decode(codes[12]) == 12
assert golomb.decode(codes[13]) == 13
assert golomb.decode(codes[14]) == 14
assert golomb.decode(codes[15]) == 15
assert golomb.decode(codes[16]) == -1
assert golomb.decode(codes[17]) == -2
```

Descodificação de uma enorme cadeia de bits

Como já foi dito, esta classe **Golomb** é extremamente importante para comprimir um vídeo. No entanto, para que seja possível descomprimí-lo, tivemos a necessidade de adaptar os métodos de descodificação anteriores para conseguirem identificar diferentes **números inteiros** numa enorme cadeia de bits. Por exemplo, lendo de um ficheiro comprimido a seguinte cadeia de bits:

- 1,0,1,1,0,0,0,0,0,0,0,1,0,1,1,1,0,1,1

Deveríamos ser capazes de identificar os números inteiros **-3, 0, 1 e 15**, por esta ordem. Para tal, desenvolvemos o seguinte método na **classe Golomb**:

```
"""
This method decodes a bitstream with multiple integers.
Example: stream_decoder([0,0,0,1,0,0,1,0,0,0,0,0]) -> [1,2,0]

@param bitstream: a list of bits
"""
def stream_decoder(self, bitstream, total=1): ...
```

Este método, portanto, recebe uma cadeia de bits e um parâmetro **total** como argumento. Esta variável **total** irá ser o número de elementos a identificar e descodificar na cadeia de bits.

Elaborámos o seguinte teste unitário para este método:

```
print("\n-------------------------------")
print("FIFTH TEST - DECODE WITH M = 4")
print("-------------------------------")

golomb = Golomb(4)

l = [-3, 0, 1, 15, 7, -6, 5, 10]
assert l ==
golomb.stream_decoder([1,0,1,1,0,0,0,0,0,0,0,1,0,1,1,1,0,1,1,0,1,0,1,1,1,1,0,1,0,0
,1,0,0,1,0,1,1,0,1,0], len(l))[0]

print("Fifth test finished with success!")
```

3.2.2. C++

Relativamente à implementação desta classe em **C++**, apenas há que referir que foram feitos os devidos ajustes tendo em conta as características da linguagem, mas que a lógica explicada até aqui não se alterou. Também os testes unitários utilizados foram os mesmos.

3.3. Video Player

3.3.1. Python 3

Classe responsável por **reproduzir o video**, dada a localização do mesmo.

```
class VideoPlayer:
    """
    Class that implements a video player.
    It reads a file in 'y4m' format and display it on a window, frame by frame.
    """
    def __init__(self, filename):
        self.fileName = filename
        # self.filePointer = 0
```

```
        self.mediaFormat = None
        self.height = 0
        self.width = 0
        self.frame = None
```

De um modo geral, esta classe carrega um ficheiro de vídeo, abre uma nova janela no ambiente gráfico do sistema operativo e exibe o vídeo. A forma de o reproduzir encontra-se no seguinte excerto de código:

```
if __name__ == "__main__":
    videoPlayer01 = VideoPlayer("../media/park_joy_444_720p50.y4m")
    videoPlayer01.openYUV()

    print("for more, see: https://wiki.multimedia.cx/index.php?title=YUV4MPEG2")
    while (videoPlayer01.visualizeFrame()):
        pass
```

Ao instanciar um novo objeto da classe **VideoPlayer**, é necessário especificar o caminho até ao ficheiro de vídeo a carregar. Após este passo, é necessário invocar o método **openYUV()**, de forma a ler o cabeçalho do ficheiro consequentemente interpretar o seu conteúdo. Por fim, basta invocar o método **visualizeFrame()** num ciclo, visualizando o vídeo.*visualizeFrame()* em loop, e visualizar o video.

Internamente, para além de se utilizar as matrizes características do módulo **numpy** e do visualizador de imagens do **opencv**, estão são em grande parte encapsuladas na classe **Frame**, como irá ser explicado no próximo ponto.

De salientar ainda que este ***player*** de vídeo funciona para ficheiros com informação não comprimida (***raw***), para qualquer um dos três formatos.

3.3.2 C++

Infelizmente, não conseguimos implementar esta classe em **C++**.

3.4. Frame

Aplicando as boas práticas programação orientada a objetos, esta classe surge no sentido de extrair e manipular da melhor forma os dados lidos do ficheiro de vídeo **y4m.** Em cada momento do vídeo, esta classe contém todas as informações do respetivo ***frame*** em reprodução (com as componentes **YUV**).

3.4.1. Python 3

```
class Frame(ABC):
    """
    Abstract class of a Frame.
    """
    def __init__(self, height, width, fileName):
        self.height = height
```

```
        self.width = width

        self.YComponent = None
        self.UComponent = None
        self.VComponent = None

        # so that we advance past the header
        self.file = open(fileName, "rb")
        self.file.readline()

    @abstractmethod
    def advance(self):
        """
        Advances and consumes the current bunch of data, so that it can refresh
the current set of matrices with new
        ones.
        :return whether or not the reading (and setting operation was successful)
        """
        pass

    @abstractmethod
    def getY(self):
        return self.YComponent

    @abstractmethod
    def getU(self):
        return self.UComponent

    @abstractmethod
    def getV(self):
        return self.VComponent
```

De forma a abstrair a implementação de cada tipo de formato de cada frame, decidiu-se que esta classe **Frame** seria abstrata. Portanto, cada frame possui atributos relativos à sua **altura**, à sua **largura** e componentes **Y, U e V.**

Relativamente ao método **advance()**, este carrega do ficheiro o ***frame* seguinte**.Além disso, possuí três métodos *getters* que retornam uma matriz homóloga da respetiva componente (Y, U ou V).

```
@abstractmethod
def advance(self):...

@abstractmethod
def getY(self):...

@abstractmethod
def getU(self):...

@abstractmethod
def getV(self):...
```

Assim sendo, foram implementadas outras três classes, derivadas da classe **Frame**, representando os diferentes tipos de sub-amostras de crominância (respetivamente, formatos **[4:4:4], [4:2:2] e [4:2:0]**):

```
class Frame444(Frame):...

class Frame422(Frame):...

class Frame420(Frame):...
```

O funcionamento de qualquer uma destas classes é muito simples. Após um objeto ser instanciado, apenas será necessário invocar o método **advance()** sempre que se quiser ler um *frame* do vídeo a partir do ficheiro. Esta classe é fundamental no **Video Player** e no **codificador do vídeo**, que será explicado no próximo ponto.

3.4.2. C++

Relativamente à implementação destas quatro classes em **C++** (*Frame, Frame444, Frame422 e Frame 420*), deve-se dizer que é bastante semalhante à implementação em **Python**. A única diferença está na utilização de **vetores de inteiros** para representar cada componente **Y, U, V**, em vez das matrizes do módulo **numpy**.

3.5. Preditores

Outra grande tarefa proposta no enunciado do trabalho era a implementação de preditores lineares **7JPEG** e do preditor não-linear **JPEG-LS**. Para tal, decidiu-se implementar diversas classes, representando cada um dos diferentes preditores. Resumidamente, a principal função destes preditores é de calcular a diferença de um valor de uma determinada matriz das componentes **Y, U ou V** com um ou vários valores anteriores, de forma a reduzir a quantidade de informação a codificar.

3.5.1. Python 3

Tendo em conta o que foi dito no último parágrafo, de seguida segue-se um excerto de código com os diferentes preditores implementados.

Nota: uma vez apenas o método **predict()** é útil, estas classes não necessitam de ser instânciadas, pelo que foram declaradas como estáticas.

```
class JPEG1:...

class JPEG2:...

class JPEG3:...

class JPEG4:...

class JPEG5:...

class JPEG6:...
```

```
class JPEG7:...

class JPEGLS:...
```

3.5.2. C++

Uma vez que não são permitidas classes estáticas em **C++**, decidimos implementar a mesma lógica com uma classe abstrata e classes derivadas desta, como se apresenta no seguinte excerto:

```
/**
 * Abstract class predictor;
 */
class Predictor {
    public:
        virtual int predict(unsigned char a, unsigned char b, unsigned char c) =
0;
};

/**
 * JPEG1 linear predictor.
 */
class JPEG1 : public Predictor {
    public:
        int predict(unsigned char a, unsigned char b, unsigned char c){
            return a;
        };
};

/**
 * JPEG2 linear predictor.
 */
class JPEG2 : public Predictor {
    public:
        int predict(unsigned char a, unsigned char b, unsigned char c){
            return b;
        };
```

3.6. Intra Frame Encoder/Decoder

No seguimento do ponto anterior, surgem duas classes, que implementam um **codificador intra-frame sem perdas de informação** e o respetivo **descodificador**. Para tal, faz-se uso das classes **Frame** e **Predictor**, já explicadas anteriormente.

3.6.1 Python 3

```
class IntraFrameEncoder():
    """
    This class implements a lossless intra-frame encoder, using 7PEG linear
```

```
predictors.
    """
    def __init__(self, predictor):...

    def write_code(self, code):...

    def setMatrix(self, new_matrix):...

    def encode(self):...


class IntraFrameDecoder():
    def __init__(self, matrix, predictor):...

    def setMatrix(self, new_matrix):...

    def decode(self):...
```

Ao instanciar um novo objeto de cada uma destas classes, estas devem receber uma outra classe do tipo **Preditor**. Assim, e dando uso às boas práticas de programação orientada a objetos, apenas será necessário espeficiar como argumento do construtor destas duas classes, o preditor linear ou não linear a utilizar, não afetando as respetivas implementações internas. Em cada momento, este **codificador** e **descodificador** necessitam de saber qual a matriz da componente **Y, U ou V** de um determinado *frame* é que devem codificar/descodificar, sendo portando invocado o método **setMatrix()**. Com a utilização deste método, é evitada a constante instanciação de novos objetos destas classes.

Nota: De seguida, reforça-se a intenção de procurar escrever código o mais eficiente possível, através da não utilização de **condições *ifs*** desnecessárias.

```
def encode(self):
        """
        This method encodes the original matrix in a new one, based on the current
predictor.
        It also uses golomb codification for the entropy encoding.
        """
        if self.original_matrix is None:
            logger.error("No matrix to encode was given!")
            return False

        # TODO: ver o que é aquele K do stor
        # write header with bitstream
        #self.bitstream.writeString("
{}\t{}".format(self.original_matrix.shape[0],self.original_matrix.shape[1]))

        # matrix size/shape is the same no mather which one
        self.encoded_matrix[0, 0] = int(self.original_matrix[0,0] -
self.predictor.predict(0,0,0))

        for col in range(1, self.original_matrix.shape[1]):
            self.encoded_matrix[0, col] = int(self.original_matrix[0, col]) -
self.predictor.predict(self.original_matrix[0, col -1], 0, 0)
```

```
        for line in range(1, self.original_matrix.shape[0]):
            self.encoded_matrix[line, 0] = int(self.original_matrix[line, 0]) -
self.predictor.predict(0, self.original_matrix[line - 1, 0], 0)

        for line in range(1, self.original_matrix.shape[0]):
            for col in range(1, self.original_matrix.shape[1]):
               self.encoded_matrix[line, col] = int(self.original_matrix[line,
col]) - self.predictor.predict(
                    self.original_matrix[line, col - 1], self.original_matrix[line
- 1, col], self.original_matrix[line - 1, col -1])
```

3.6.2 C++

Relativamente à implementação da classe **IntraFrameEncoder** em **C++**, de salientar que esta é muito semelhante à implementada em **Python**.

4. Execução dos programas

Este ponto serve para exemplificar a execução dos nossos programas de teste.

4.1. BitStream

Python 3

```
$ cd python/src/bitstream
$ python3 test_bitstream.py
```

C++

```
$ cd cpp/src/bitstream
$ g++ testing_BitStream.cpp -o testing_BitStream
$ ./testing_BitStream
```

4.2. Golomb

Python 3

```
$ cd python/src/golomb
$ python3 test_golomb.py
```

C++

```
$ cd cpp/src/golomb
$ g++ test.cpp -o test
$ ./test
```

4.3. Video Player

Python 3

```
$ cd python/src/
$ python3 VideoPlayer.py
```

Nota importate: Para alterar o vídeo a ser reproduzido, deve primeiro colocá-lo na pasta **python/media**, e de seguida especificar o caminho até ele no ficheiro **VideoPlayer.py.**

```
if __name__ == "__main__":
    videoPlayer01 = VideoPlayer("../media/park_joy_444_720p50.y4m")
```

4.4. Codificador Intra-Frame

Python 3

```
$ cd python/src
$ python3 video-encoder.py
```

C++

```
$ cd cpp/src
$ cmake .
$ make
$ ./VideoEncoder
```

Estes dois programas codificam o vídeo ***park_joy_444_720p50.y4m***, presente na pasta **python/media/** (PARA AMBAS AS LINGUAGENS!), e guardam o resultado na pasta **python/out/** e **cpp/out/**.

4.5. Descodificador Intra-Frame

Python 3

```
$ cd python/src
$ python3 video-decoder.py
```

Este programa descodifica o vídeo codificado com apenas 3 frames, ***3_frames_encoded_park_joy_444_720p_50.bin***, presente na pasta **python/out/**, exibindo-o numa janela do ambiente gráfico do sistema operativo com o auxílio do **VideoPlayer.**

5. Considerações e Notas

5.1. Compressão 4:4:4 em Python 3

Com o intuito de testar o nosso compressor de vídeo, decidimos fazê-lo codificando o vídeo ***park_joy_444_720p50.y4m***, no formato **4:4:4**. No entanto, apenas codificamos metade dos frames presentes no mesmo, para pouparmos tempo.

Na seguinte imagem mostra-se o *output* no terminal dessa compressão.

```
Frame compressed in 16 s. Total bits: 3511846655. Frames no. 238
Frame compressed in 14 s. Total bits: 3526484951. Frames no. 239
Frame compressed in 15 s. Total bits: 3541238255. Frames no. 240
Frame compressed in 15 s. Total bits: 3556047682. Frames no. 241
Frame compressed in 15 s. Total bits: 3570883894. Frames no. 242
Frame compressed in 14 s. Total bits: 3585790590. Frames no. 243
Frame compressed in 16 s. Total bits: 3600801435. Frames no. 244
Frame compressed in 16 s. Total bits: 3615888446. Frames no. 245
Frame compressed in 14 s. Total bits: 3630992010. Frames no. 246
Frame compressed in 16 s. Total bits: 3646164560. Frames no. 247
Frame compressed in 15 s. Total bits: 3661274648. Frames no. 248
Frame compressed in 14 s. Total bits: 3676420489. Frames no. 249
Frame compressed in 14 s. Total bits: 3691615039. Frames no. 250
Compressed frames in 3680 s.
```

Como se pode constantar, o tempo de compressão de um frame está dentro do intervalo de 14 a 16 segundos, sendo que são necessários 3680 segundos (aproximadamente **1 hora**). Assumimos que este valor temporal não é ótimo mas que foi, genuinamente, o melhor que conseguimos, face a diversos fatores envolvidos como os conhecimentos de ambos os elementos do grupo, ferramentas utilizadas e tempo disponível.

Salienta-se ainda que, para testar os restantes formatos, optou-se apenas por comprimir um número reduzido de frames, para comprovar a sua funcionalidade.

Relativamente ao preditor utilizado, optámos por manter o **JPEG1**, por não se justificar testar a compressão de vídeo exaustivamente com todos os preditores implementados.

5.2. C++

Devido a limitações temporais e às dificuldades inerentas a esta linguagem de baixo nível, não foi possível implementar todas as classes que desejaríamos (em especial o **VideoPlayer** nesta linguagem). No entanto, pensamos que as restantes classes foram implementadas com sucesso, ainda que talvez não na sua máxima eficiência.

6. Conclusão

Após a realização deste projeto, destaca-se sobretudo a obtenção de novas competências por parte de ambos os elementos do grupo. Destaca-se o aumento de conhecimentos sobre os seguintes tópidos:

- Paradigma de programação orientada a objetos
- Linguagens de programação **Python** e **C++**
- Módulos **Numpy** e **OpenCV**
- Processamento de informação de um vídeo

Ainda que não tivéssemos conseguido implementar todas as tarefas pedidas no enunciado do projeto, salientamos que os objetivos alcançados foram muito importantes no melhoramento das competências enumeradas em cima.

De um modo geral, este trabalho, apesar de desafiador, foi muito interessante.

7. Bibliografia

- [Página da UC](https://elearning.ua.pt/course/view.php?id=7133
- https://numpy.org/
- https://opencv.org/