FACULDADE DE
CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE D
COIMBRA

MULTIMÉDIA

CODEC JPEG

Daniel Simões

David Silva

Dinu Bosîi

Luiz França

2022/2023

Índice

Introdução

Este projeto tem como objetivo converter uma imagem para JPEG, com vista a comprimir uma imagem de forma a ocupar menos espaço, útil para transmitir de forma mais rápida via internet, como também ocupar o menor espaço possível.

Assim, é possível recuperar a "imagem original" no processo de descompressão, caso o fator de qualidade escolhido na compressão não tenha sido muito reduzido.

Assim, existem X principais passos para comprimir uma imagem no codec JPEG: Conversão dos canais RGB para YCbCr, down-sampling dos canais Cb e Cr, aplicação da DCT nos canais YCbCr, a quantização dos coeficientes da DCT, codificação DPCM e ainda a compressão dos coeficientes AC (não aplicado neste projeto).

A resolução deste projeto foi elaborada em Python com recurso às bibliotecas Matplotlib, Numpy, cv2, Scipy, essencialmente.

Imagens comprimidas (com recurso ao Gimp)

Com recurso à ferramenta GIMP, comprimimos as imagens BMP em formato JPEG com diferentes fatores de qualidade, podendo assim compreender as diferentes imagens obtidas, tanto a nível de parecença com a imagem BMP, bem como a taxa de compressão obtida.

Exemplo de compressão na imagem BarnerMountains.bmp:

Figura 1 - MountainBarners.jpeg com fator de qualidade 75

Figura 2- MountainBarners.jpeg com fator de qualidade 50

Figura 3- MountainBarners.jpeg com fator de qualidade 25

Exemplo de compressão na imagem Logo.bmp:

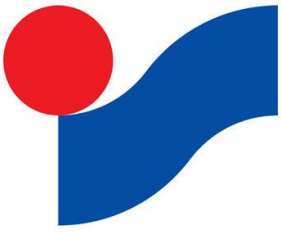


Figura 4 - Logo.jpeg com fator de qualidade 75

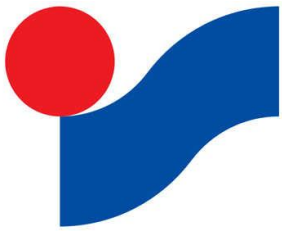


Figura 5- Logo.jpeg com fator de qualidade 50

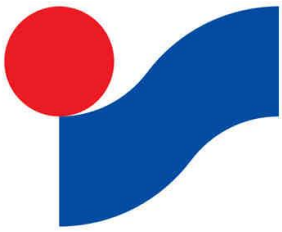


Figura 6- Logo.jpeg com fator de qualidade 25

Exemplo de compressão na imagem Logo.bmp:

Figura 7 - Peppers.jpeg com fator de qualidade 75

Figura 8- Peppers.jpeg com fator de qualidade 50

Figura 9- Peppers.jpeg com fator de qualidade 25

Assim, obtemos os diferentes valores de taxas de compressão para as diferentes imagens enunciadas:

Fator de qualidade	peppers.bmp	barn.bmp	logo.bmp
max (75%)	19:1	10:1	43:1
media (50%)	28:1	16:1	54:1
min (25%)	43:1	28:1	66:1

Assim, concluímos que quanto menor for o fator de qualidade, maior vai ser a taxa de compressão. Além disso, podemos ter em conta que na imagem logo.bmp são obtidas taxas de compressão mais elevadas, pois como existem apenas 3 cores, a codificação entrópica é superior.

Leitura da imagem em Python

Para ler as imagens fornecidas, criamos a seguinte função com recurso à biblioteca matplotlib.pyplot:

```python
def read_image(image):
    img = plt.imread(image)
    R = img[:, :, 0]
    G = img[:, :, 1]
    B = img[:, :, 2]

    return R, G, B, img
```

Nesta função, lemos o caminho da imagem original e retornamos a matriz que contém os 3 canais R,G, B que representam a imagem, como também, os canais R, G, B em separado.

Visualização das imagens em Python

Para observar as imagens do input ou output, é necessário escolher o colormap em que queremos observar a imagem.

Com recurso à biblioteca *matplotlib.colors* construímos esta função que retorna um colormap:

```python
def colormap(name, colors, num):
    return clr.LinearSegmentedColormap.from_list(name, colors, num)
```

Assim, caso queiramos o colormap da região verde teríamos de invocar a função colormap com os parâmetros: 'green', [(0, 0, 0), (0, 1, 0)] e 256.

Além disso, para mostrar a imagem no ecrã recorremos à biblioteca matplotlib.pyplot e construímos a seguinte função:

```python
def show_image(img,  title, cmap=None):

    plt.figure()
    plt.axis('off'), plt.title(title), plt.imshow(img)
    plt.imshow(img, cmap)
```

Assim, caso queiramos ver o canal *G* de uma imagem teríamos de invocar a função show_image com os seguintes parâmetros: img[: , : ,1], "Canal G" e colormap('green', [(0, 0, 0), (0, 1, 0)] e 256).

Padding

O processo de Padding consiste em adicionar dados extra ao início, meio ou fim de uma string, binário ou outras estruturas de dados para alinhar ou ajustar o seu tamanho a algum requisito específico.

No contexto de processamento de imagens, o padding é uma técnica de pré-processamento utilizada para adicionar pixels ou valores em torno da borda de uma imagem. Assim, o objetivo é aumentar a dimensão da imagem para um tamanho especificado.

Na nossa aplicação, foi criada uma função para fazer padding da imagem tornando-a múltipla de 32x32. A última linha e a última coluna são repetidas até que a imagem alcance as dimensões pretendidas.

O objetivo dessa função é preparar a imagem para ser utilizada em algoritmos de compressão de imagem que trabalham com blocos de 8x8 pixels, pois as dimensões múltiplas de 32 garantem que a imagem pode ser dividida em blocos 8x8 sem que sobrem pixels.

Assim, desenvolvemos a função de padding, que iremos descrever em seguida.

```python
def padding(image):
    [l, c, Ch] = image.shape

    nc = nl = 0
    ...
```

Aqui, a função padding recebe uma imagem como input e remodela seu formato. que inclui sua altura “l”, sua largura “c” e seu número de canais “Ch”.

As variáveis “nc” e “nl” são inicializadas a zero. Estas serão usadas posteriormente para guardar quantas colunas e linhas serão adicionadas à imagem para tornar suas dimensões múltiplas de 32.

```python
if l % 32 != 0:
        #padding horizontal
        nl = 32 - l % 32 # número de linhas adicionar
        #li = x[nl-1, :]

        ll = image[l-1, :][np.newaxis, :]

        repl = ll.repeat(nl, axis=0)

        # para adicionar repl a x, vertically
        image = np.vstack([image, repl])
```

Se a altura da imagem não for múltipla de 32, a variável “nl” receberá o valor que representa o número de linhas que serão necessárias ser adicionadas para tornar a altura múltipla de 32. Assim, o código guarda a última linha da imagem original “ll” e repete-a “nl” vezes para criar a imagem “repl”. Finalmente, a nova imagem é empilhada verticalmente no topo da imagem antiga para criar uma imagem múltipla de 32 em altura.

```python
if c % 32 != 0:
        nc = 32 - c % 32 # número de colunas a adicionar

        lc = image[:, c-1][:, np.newaxis] #last column

        repc = lc.repeat(nc, axis=1)

        image = np.hstack([image, repc]) #repetição horizontal
```

Aqui, caso a largura da imagem não seja múltipla de 32, a variável “nc” recebe o número de colunas que são necessárias para que a largura seja múltipla de 32. O código repete a última coluna da imagem original “lc” repete-a em nc vezes horizontalmente para criar a imagem “repc”. Finalmente, a nova imagem é posta ao lado da imagem original, para criar o padding vertical.

Por fim, retornamos a imagem com largura e altura múltipla de 32

Figura 10- MountainBarners.bmp original

Figura 11- MountainBarns.bmp com padding aplicado

Importa ainda frisar que desenvolvemos também a função de padding inverso, isto é, dado uma imagem com padding conseguimos obter a imagem original, retirando as linhas e colunas que adicionamos.

```python
def padding_inv(l, c, imagem_pad):
    return imagem_pad[0:l, 0:c, :]
```

Assim, a função padding_inv recebe como argumento o l e o c, que representam as dimensões originais da imagem, e também a própria imagem com o padding.

Após isso, é retornada um subarray de imagem_pad, cortando a imagem para que fique com suas dimensões originais.

Conversão de RGB para YCbCr

O **YCbCr** é um modelo de cor que separa as informações de cor e brilho de uma imagem. O componente Y (luma) representa o brilho da imagem, enquanto ambos os componentes Cb e Cr representam o chroma, isto é, as informações relacionadas às cores.

O componente Y é calculado usando-se a equação *Y = 0.299R + 0.587G + 0.114B*, onde R, G e B são os componentes vermelho, verde e azul de um dado pixel no espaço de cor RGB.

Os componentes Cb e Cr são então calculados usando-se as equações *Cb = -0.168736R - 0.331264G + 0.5B + 128* e Cr = *0.5R - 0.418688G - 0.081312B + 128*.

Comparando o componente Y com os componentes R, G e B, podemos observar que o componente Y contém apenas a informação referente ao brilho da imagem, enquanto R, G e B também possuem informação sobre as cores. Isso acontece devido ao fato do componente Y ser calculado como o resultado de uma média ponderada de R, G e B, onde os pesos são escolhidos para refletir a sensibilidade do olho humano.

Por outro lado, comparando o componente Y com os componentes Cb e Cr, podemos ver que Cb e CR contém as cores da imagem, mas de uma forma diferente de R, G e B.

O Cb e Cr são obtidos subtraindo-se o Y dos canais azul e vermelho, respetivamente, e então subamostrando-os para reduzir sua quantidade de dados.

Isso significa que os componentes Cb e CR representam a informação da cor de uma maneira menos sensível ao brilho da imagem.

Figura 12- MountainBarers, paddin

Figura 13- MountainBarners, padding, canal Y

Figura 14- MountainBarners, padding, canal Cb

Figura 15- MountainBarners, padding, canal Cr

Assim, convertemos a imagem para YCbCr, de forma a puder desprezar a quantidade de informação armazenada nos canais Cb e Cr, dado que somos menos sensíveis a estes 2 canais.

Concluindo, o componente Y do YcBCr representa o brilho da imagem, enquanto Cb e Cr representam a cor de uma forma diferente do R, G e B.

Subamostragem

Fazer a subamostragem de uma imagem é redimensionar uma imagem de forma a torná-la mais pequena consoante um rácio de compressão.

De forma a fazer a subamostragem das imagens necessitamos das duas componentes *chroma* da imagem isto é, Cb e Cr, e também os parâmetros de dimensão da imagem, FCb e FCr.

No caso de FCr ser igual 0 então passa os fatores Cr_fx, Cb_fx, C_fy para 0,5 (Figura 16). Se não for então calcula as dimensões a utilizar dividindo Cr_fx e Cb_fx por 4 (Figura 17).

```
1  if FCr == 0:
2          Cr_fx = Cb_fx = C_fy = FCb / 4
```

Figura 16

```
1  else:
2          Cb_fx = FCb / 4
3          Cr_fx = FCr / 4
4          C_fy = 1.0
```

Figura 17

Usamos de seguida a função *resize* do módulo *cv2* passando como argumentos a imagem a usar (Cb e Cr), a dimensão da imagem, neste caso um tuple (0,0), os fatores de redimensionamento fx e fy para os seus valores (Cb_fx/Cr_fx e Cb_fy/Cr_fy) e o modo de interpolação previamente definido por cv2.INTER_LINEAR (Figura 18).

```
1  Cb_down = cv2.resize(Cb, (0, 0), fx = Cb_fx, fy = C_fy, interpolation=interpolacao)
2  Cr_down = cv2.resize(Cr, (0, 0), fx = Cr_fx, fy = C_fy, interpolation=interpolacao)
```

Figura 18

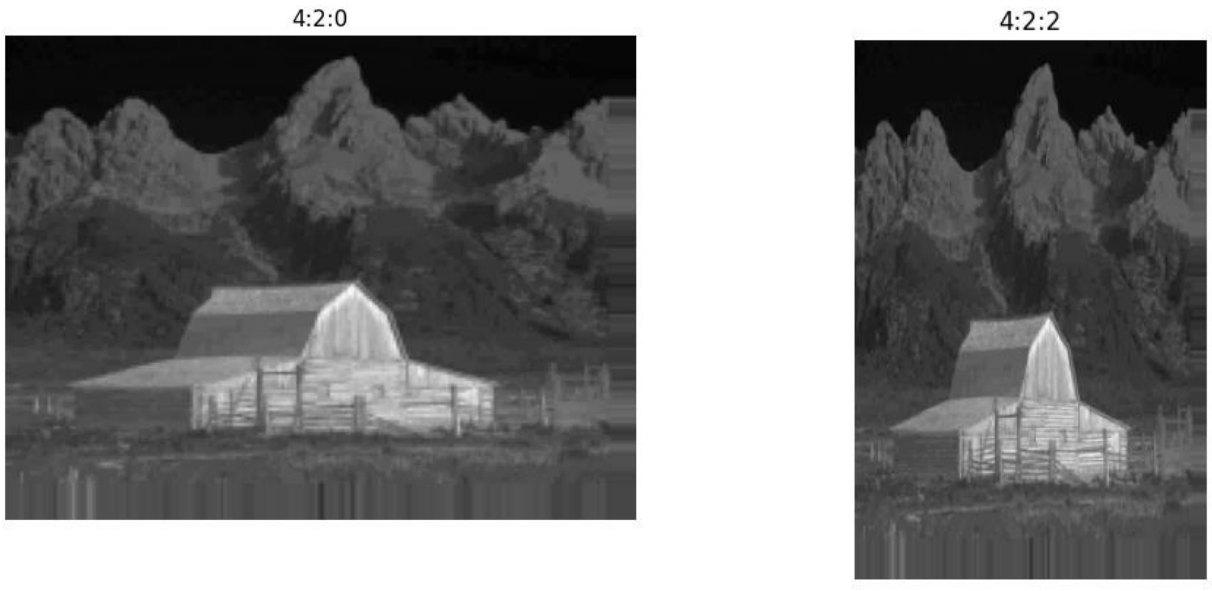


Figura 19

Está então demonstrado as diferentes formas de redimensionamento, 4:2:0 na figura 4 e 4:2:2 na figura 19.

Reverter a Subamostragem

Para reverter a Subamostragem realizada anteriormente pela função downsampling usamos uma função semelhante a esta supramencionada, mas com a pequena alteração no método de calcular os fatores da dimensão. Em vez de se multiplicar o fator de redimensionamento, em upsampling divide-se (Figura 20), de forma a obter a imagem com o seu tamanho original.

```
if FCr == 0:
    Cr_fx = Cb_fx = C_fy = 4 / FCb
else:
    Cb_fx = 4 / FCb
    Cr_fx = 4 / FCr
    C_fy = 1.0
```

Figura 20

Transformada de Coseno Discreta (DCT)

Na compressão de imagens, a DCT é usada para transformar a imagem do domínio espacial para o domínio de frequência, onde a imagem pode ser representada por um conjunto de coeficientes que correspondem a diferentes frequências. Assim sendo, é uma maneira eficaz de representar e comprimir dados de uma imagem.

Usamos então a função shape para definir o tamanho da *img_dct* para o qual vamos introduzir o valor retornado do cálculo da DCT.

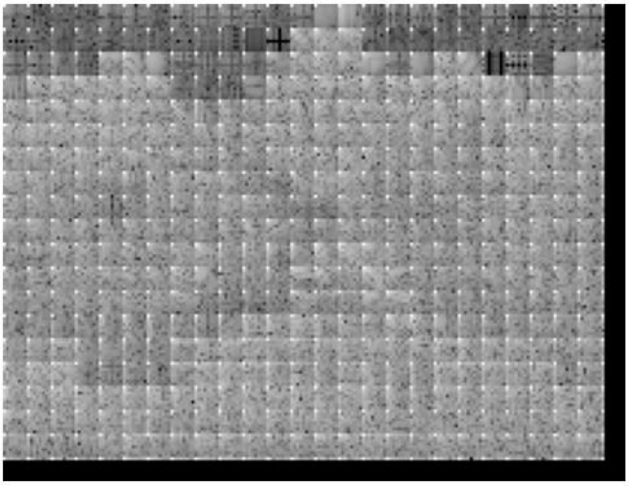
Depois implementamos dois ciclos para percorrer a imagem pelos dois eixos com um step do tamanho do bloco que queremos implementar (bsize) para obtermos um bloco desse tamanho.

```
for i in range(0, im_size[0], bsize):
    for j in range(0, im_size[1], bsize):
        img_dct[i:(i+bsize), j:(j+bsize)] = DCT(image[i:(i+bsize),j:(j+bsize)])
```

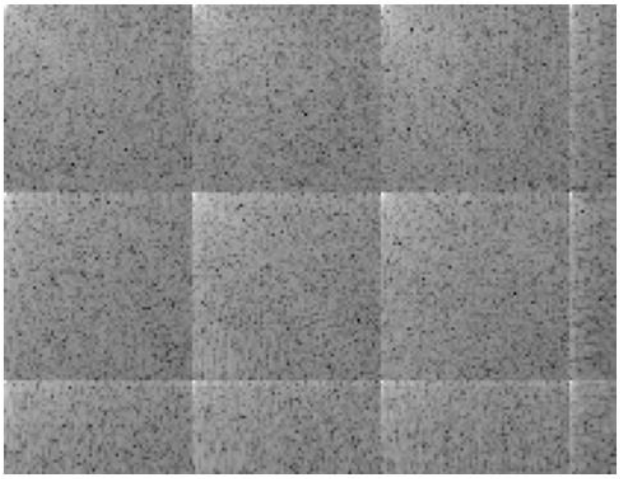
Para calcular a DCT usamos a função auxiliar *DCT* que vai ser aplicada em cada bloco da imagem.

```
def DCT(image):
    return dct(dct(image, norm='ortho').T, norm='ortho').T
```

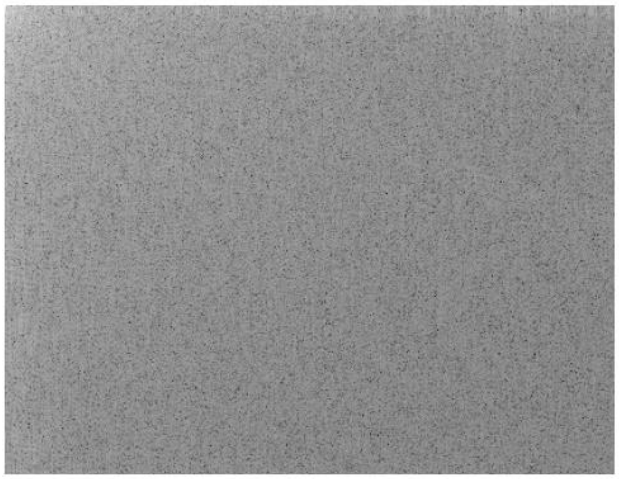
Canal Cb DCT 8x8



Canal Cb DCT 64x64



Canal Cb DCT



Reduzir o tamanho dos blocos em que é aplicada a DCT, aumenta a probabilidade de obter valores menores nas altas frequências (por ser mais provável conter apenas transições suaves num bloco de menor tamanho) sendo possível alcançar uma maior taxa de compressão. Como o objetivo é remover a maior quantidade de informação possível de forma a que não tenha uma impacto muito visível na imagem, os blocos de 8x8 são a melhor opção. Usar a DCT sem blocos obtém-se valores mais elevados nas frequências altas que depois de serem transformados, terão um maior impacto na qualidade da imagem reconstruída.

Inversão da Transformada de Cosseno Discreta (IDCT)

De igual forma como se reverteu a Subamostragem, neste ponto usamos exatamente o mesmo código usado na alínea anterior mas com a função auxiliar *IDCT* que vai calcular a inversa da DCT.

```
1  def IDCT(image):
2      return idct(idct(image, norm='ortho').T, norm='ortho').T
```

Depois de calculada a inversa obtemos então a figura 21.

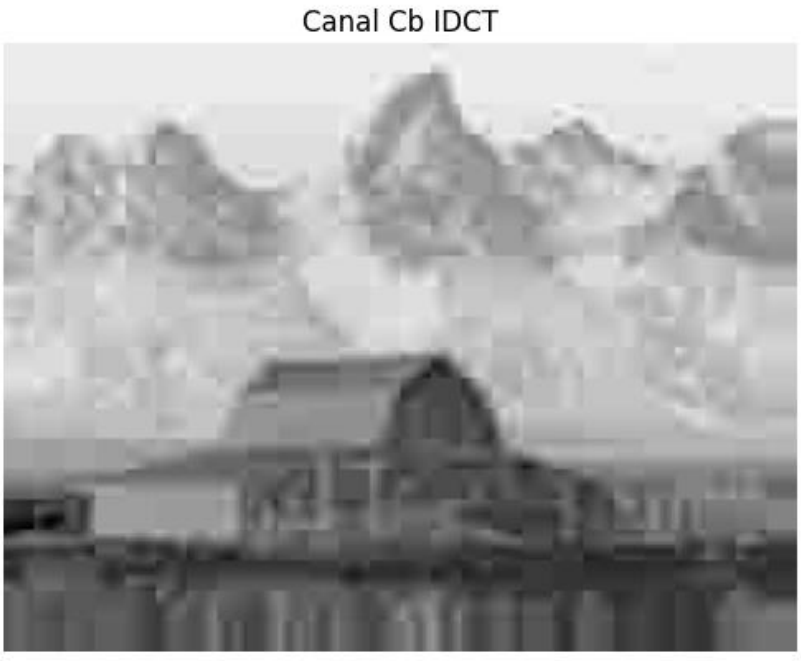


Figura 21

Quantização

Após a aplicação da DCT, obtém-se uma matriz com valores em que se observam baixos valores para as altas frequências (canto inferior direito).

1359.1	2.9	2.0	1.2	0.2	-0.8	-0.9	-0.4
3.0	2.8	2.2	0.7	-0.1	-0.4	-0.8	-0.7
0.1	0	0	0	0	-0.1	-0.1	0
-0.9	-1.1	-0.9	-0	-0.1	0.1	0.4	0.1
-0.0	-0	-0	0	0	0	0	0
0.6	0.7	0.6	0.0	1	0	-0.3	0
.0	0	0	0	0	0	0	0
-0.6	-0.5	-0.4	-0.2	-0.0	0	0.1	0.1

Tabela 1 - valores do 1º bloco DCT 8x8 do canal Cb da imagem barn mountains

Como o olho humano não tem tanta sensibilidade para distinguir a intensidade exata da variação em componentes de alta frequência espacial, estas vão ser representadas com menos detalhe.Para isso, os coeficientes da DCT serão representados com menos informação dividindo-os por uma matriz de quantização.

Para obter a matriz de quantização correta, a matriz pré-definida Q passa pela função *Quantization_quality*, e, de acordo com o fator de qualidade *qf* é aplicada uma fórmula sobre essa matriz, obtendo-se a matriz para a qualidade desejada.

```
def Quantization_quality(Q, qf):
    if qf >= 50:
        sf = (100 - qf)/ 50
    else: sf = 50/qf
    if sf == 0:
        return np.ones((8, 8), dtype=np.uint8)


    Qs = np.round(np.multiply(Q,sf))
    Qs[Qs > 255] = 255
    Qs[Qs < 1] = 1
    Qs = Qs.astype(np.uint8)

    return Qs
```

Figura 22- Função que retorna a matriz de quantização com a qualidade fornecida

A quantização é feita com auxílio da função *Quantization_aux* onde uma matriz inicializada a zero é preenchida com blocos de tamanho *bsize* do resultado do arredondamento da imagem DCT a dividir pela matriz de quantização:

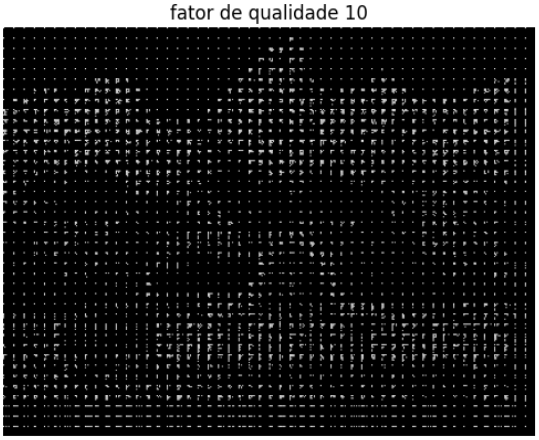
```python
def Quantization_aux(arr, bsize, Q):
    arr_Q = np.zeros(arr.shape)
    for i in range(0, arr.shape[0], bsize):
        for j in range(0, arr.shape[1], bsize):
            arr_Q[i:(i+bsize), j:(j+bsize)] = np.round(np.divide(arr[i:(i+bsize), j:(j+bsize)], Q))

    return arr_Q.astype(np.int32)
```

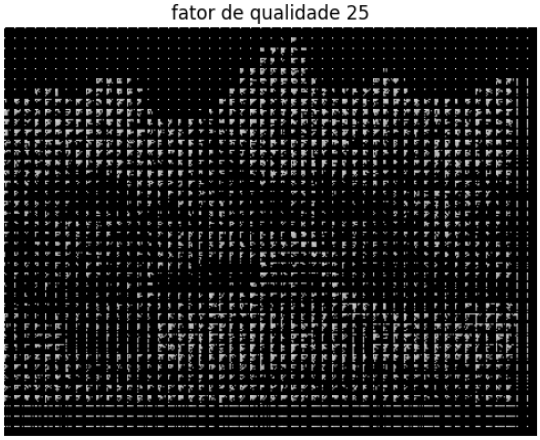
Para a quantização inversa, o processo é semelhante, diferindo apenas na operação usada (multiplicar em vez de dividir a DCT pela matriz de quantização), retornando também uma matriz do tipo float em vez de int.

Apresenta-se a imagem da DCT quantizada com diferentes fatores de qualidade para o canal Y da imagem *barn mountains*:

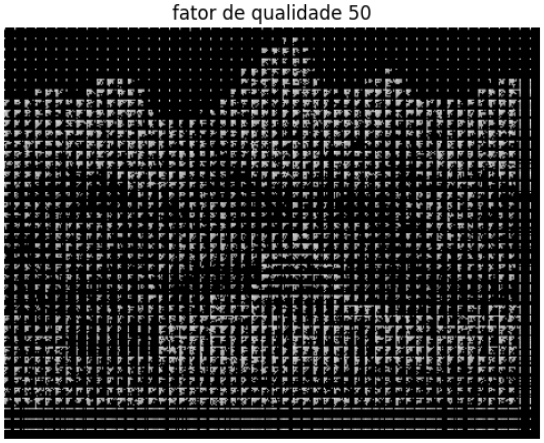
fator de qualidade 10



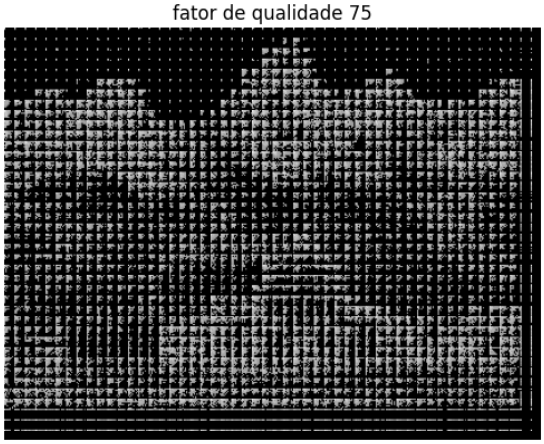
fator de qualidade 25



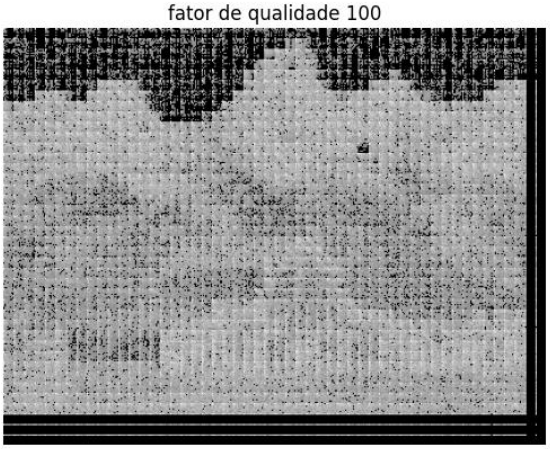
fator de qualidade 50



fator de qualidade 75



fator de qualidade 100



Observa-se que quanto menor for o fator de qualidade, menos informação contém a imagem da DCT quantizada, o que se traduz num maior potencial de compressão dado que contêm um elevado número de valores próximos de 0(resultado da perda de informação nas altas frequências) e uma gama de valores menor de forma geral.

Em comparação com os resultados obtidos antes da etapa de quantização, torna-se evidente o impacto que a quantização terá na taxa de compressão da imagem, dado que a DCT apresenta uma gama de valores superior e valores mais altos nas altas frequências, sendo estes valores eliminados na quantização reduzindo desta forma a quantidade de informação menos significante a armazenar.

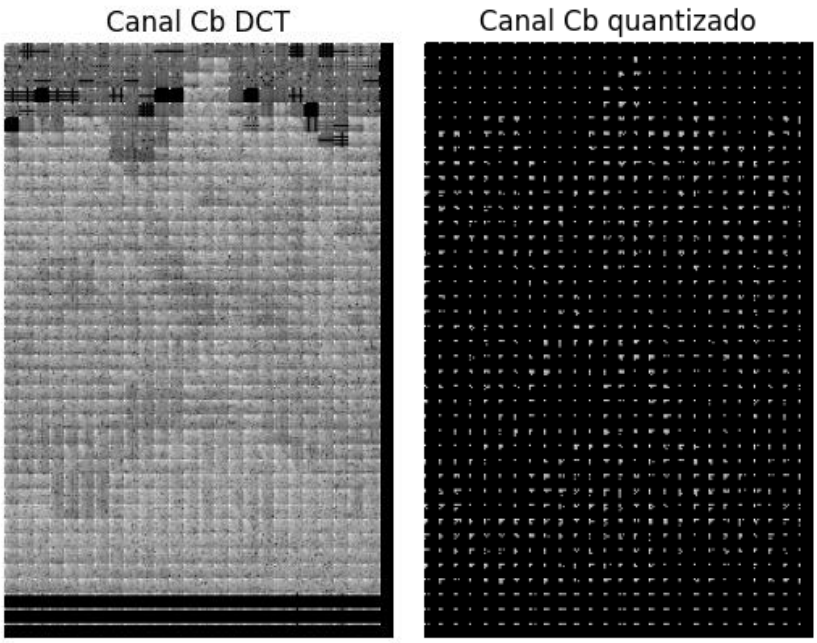


Figura 23 e 24 – DCT do Canal Cb e quantização com fator de qualidade 50 desse canal

Codificação DPCM

Feita a quantização da DCT, observa-se que o coeficiente DC de cada bloco contêm os valores mais elevados da imagem. Há uma alta correlação para os coeficientes de blocos adjacentes (por ser mais provável ter transições suaves) sendo possível reduzir a redundância desses valores através de uma codificação diferencial.

A codificação DPCM é feita através da função Codificacao_DPCM, em que para o coeficiente DC de cada bloco 8x8, é armazenada a diferença do valor desse coeficiente com o do coeficiente anterior.

```python
def Codificao_DPCM(Y_qt):
    last = 0
    aux = 0

    for i in range(0, Y_qt.shape[0], 8):
        for j in range(0, Y_qt.shape[1], 8):
            aux = Y_qt[i][j]
            Y_qt[i][j] = Y_qt[i][j] - last
            last = aux

    return Y_qt
```

No processo inverso, é feita a soma de cada coeficiente DC com o coeficiente do bloco anterior.

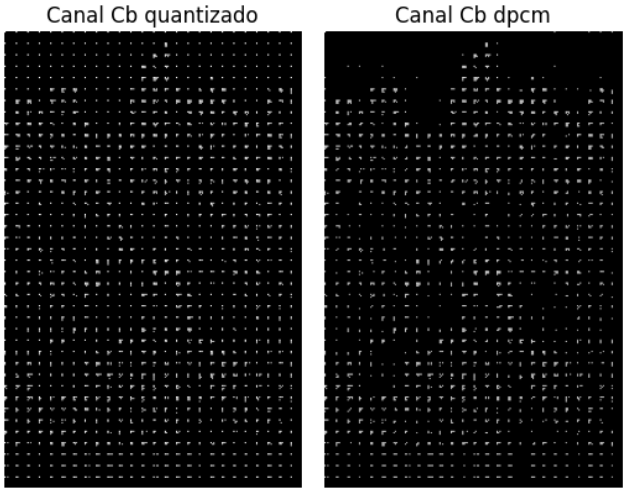


Figura 25 e 26 – Quantização (qf = 50) do Canal Cb e codificação DPCM desse mesmo canal da imagem barn mountains

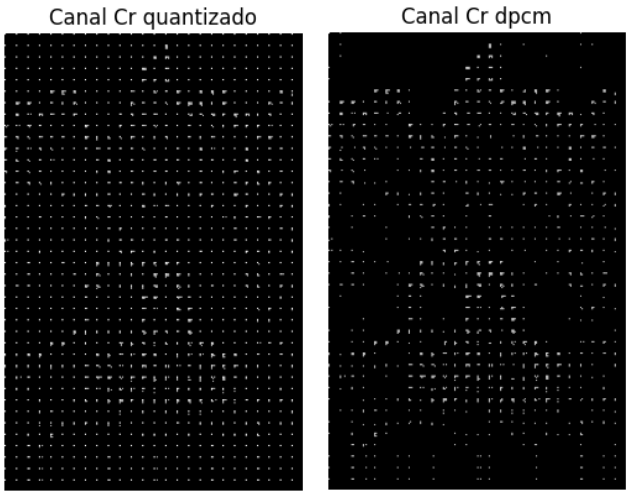


Figura 27 e 28 – Quantização (qf = 50) do Canal Cr e codificação DPCM desse mesmo canal da imagem barn mountains

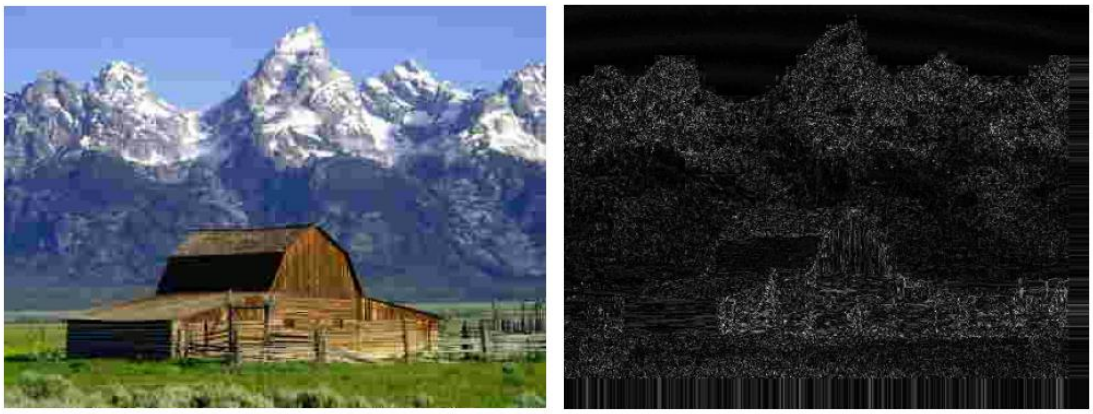
Após a codificação DPCM, observa-se uma maior quantidade de valores próximos de 0, que posteriormente se iria refletir na compressão entrópica, dado que uma gama de valores mais estreita e com menor variância leva a uma maior taxa de compressão.

Métricas e resultados

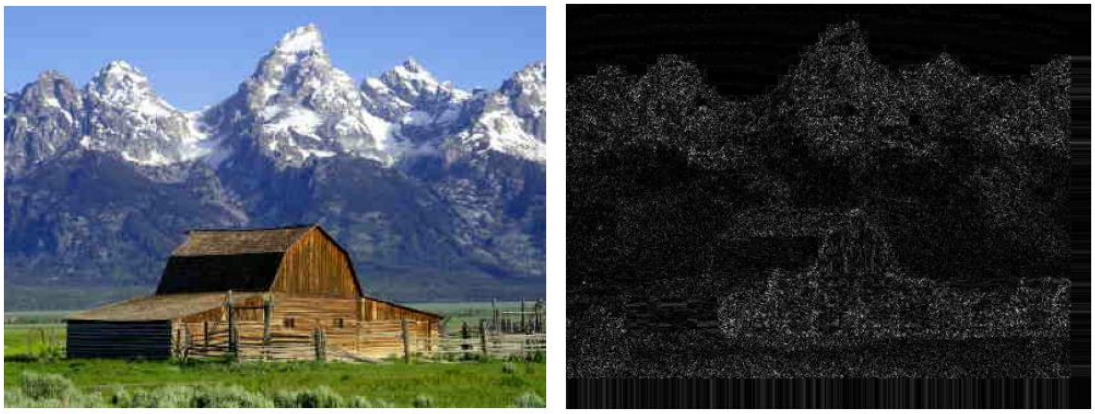
De forma, a analisar os resultados calculamos os diferentes erros relativos à diferença entre a imagem original e a imagem reconstruída através da compressão JPEG.

Imagem	Qualidade	Mean	MSE	RMSE	SNR	PNSR
BarnMountains.bmp	10	9.45	704.44	26.54	18.71	19.65
	25	6.78	395.66	19.89	21.21	22.16
	50	5.25	258.80	16.09	23.06	24.00
	75	3.83	150.17	12.25	25.42	26.36
	100	0.21	11.26	3.35	36.67	37.62
Peppers.bmp	10	4.49	280.17	16.74	20.45	23.66
	25	2.68	125.86	11.22	23.93	27.13
	50	1.96	76.39	8.74	26.09	29.30
	75	1.48	49.50	7.04	27.98	31.18
	100	0.23	7.71	36.05	39.26	39.25
Logo.bmp	10	5.21	154.54	12.43	29.44	26.24
	25	1.52	63.98	8.00	33.27	30.07
	50	1.27	41.72	6.46	35.13	31.92
	75	0.44	22.37	4.73	37.83	34.63
	100	0.04	5.88	2.42	43.64	40.44

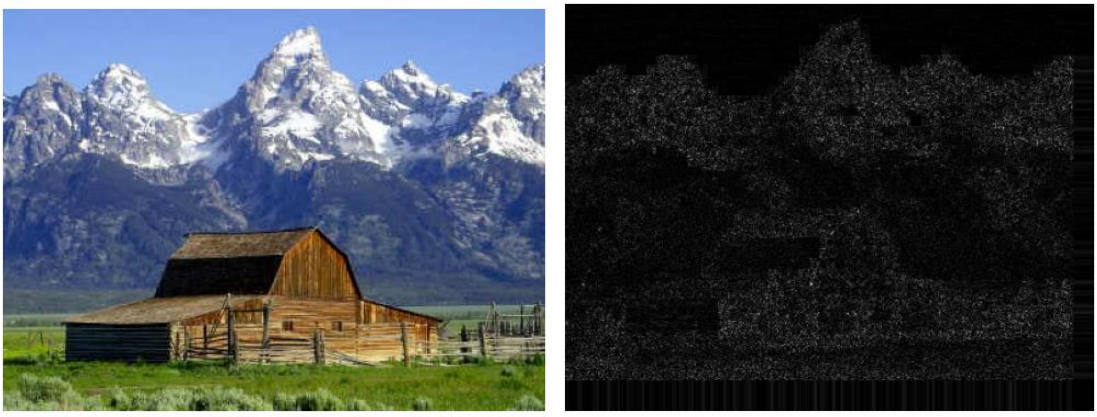
De seguida, mostramos as imagens obtidas após o processo de decoder (à esquerda) e a imagem das diferenças (à direita)



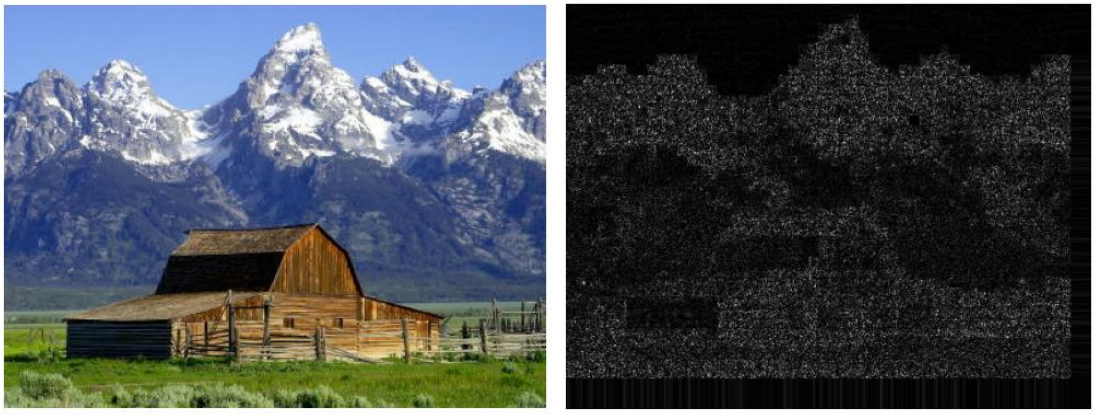
Qualidade 10 (BarnMountains)



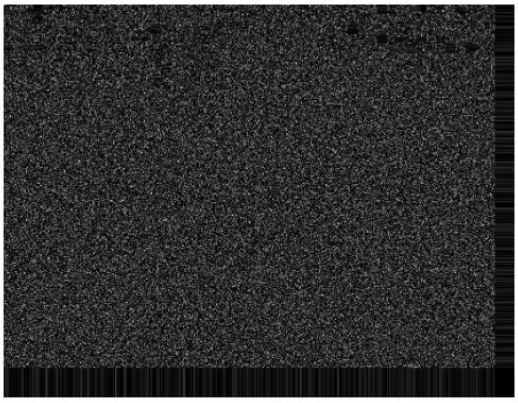
Qualidade 25(BarnMountains)



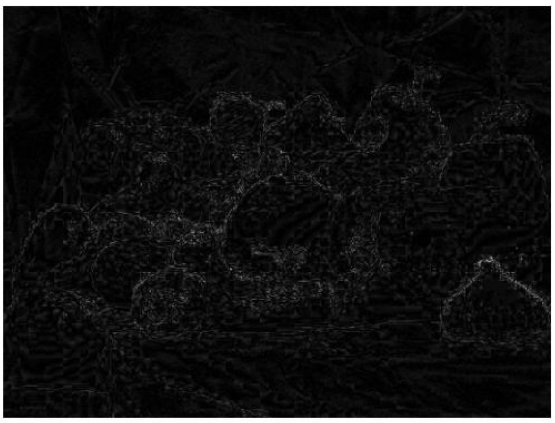
Qualidade 50 (BarnMountains)



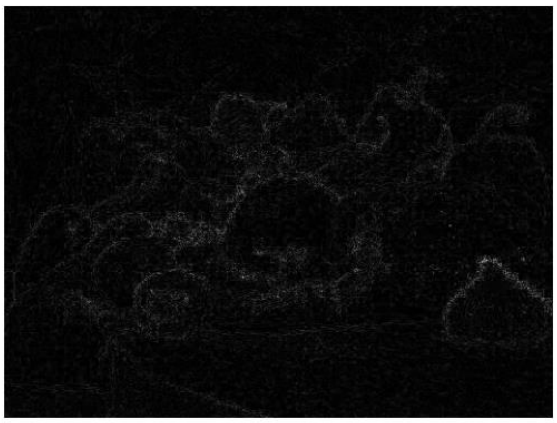
Qualidade 75 (BarnMountains)



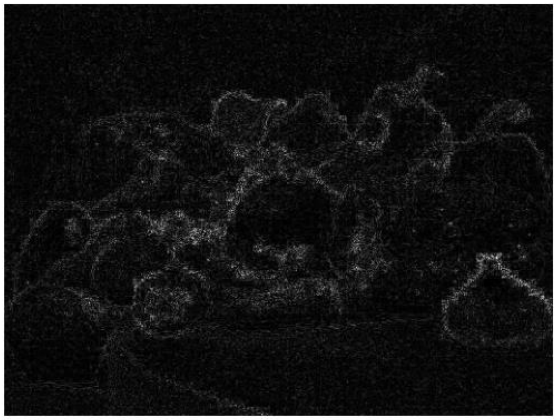
Qualidade 100 (BarnMountains)



Qualidade 10 (Peppers)



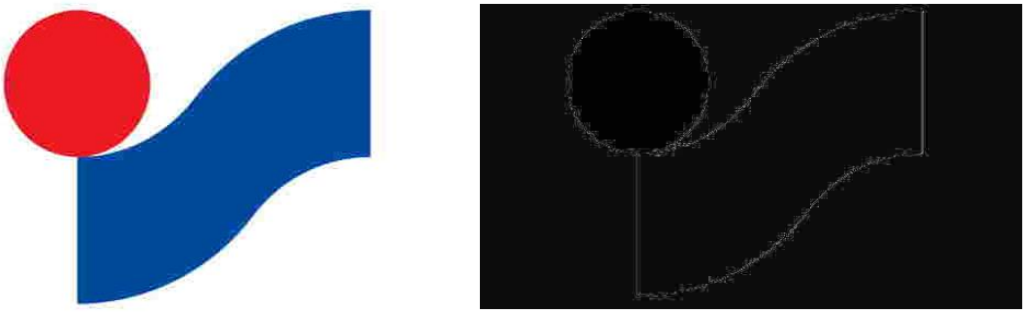
Qualidade 25 (Peppers)



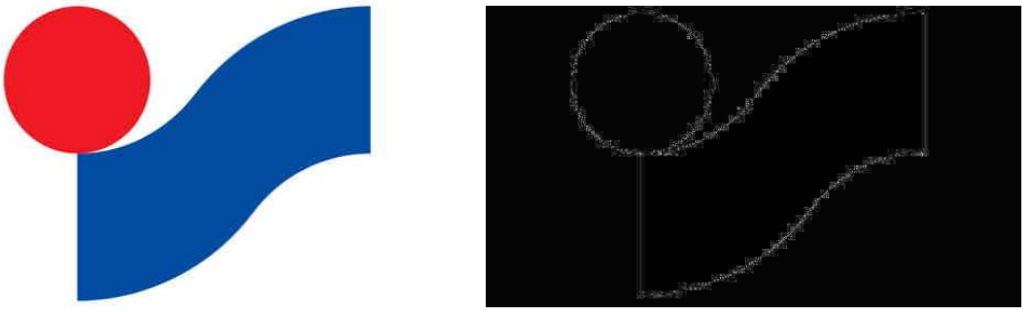
Qualidade 50 (Peppers)

Qualidade 75 (Peppers)

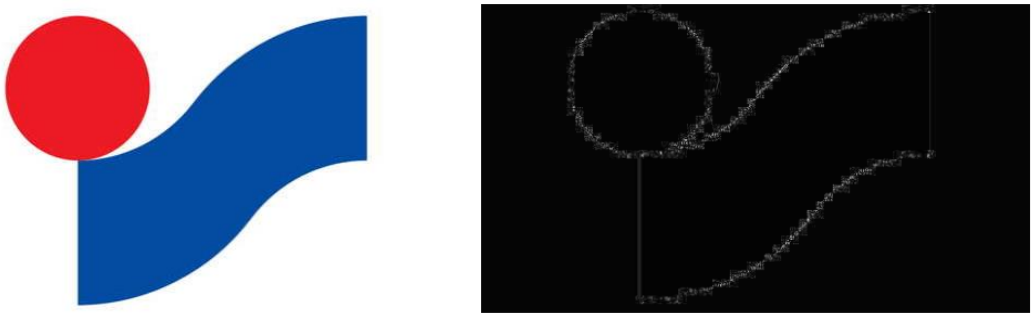
Qualidade 100 (Peppers)



Qualidade 10 (Logo.bmp)



Qualidade 25 (Logo.bmp)



Qualidade 50 (Logo.bmp)

Qualidade 75 (Logo.bmp)

Qualidade 100 (Logo.bmp)

Observando os resultados, a imagem 'Barn Mountains' apresenta valores superiores para a média do erro, para o MSE e RMSE, sendo por isso a imagem que mais informação perdeu. Obteve também os piores valores para Signal-to-Noise Ratio e Peak Signal-to-Noise Ratio. Este resultado pode ser devido à elevada quantidade de detalhes que a imagem contém e a elevada game de cores.

Na imagem logo é observável o erro produzido derivado das transições de cor abruptas, mas como estas constituem uma área reduzida da imagem, os erros obtidos são inferiores.

Conclusão

Ao longo da resolução deste projeto, conseguimos compreender de forma prática o codec JPEG, e como implementá-lo em Python.

Contudo, o codec JPEG não está completo, dado faltar a codificação entrópica (código Huffman e codificação aritmética), após a quantização dos coeficientes DCT, algo que não era objeto de estudo deste projeto.