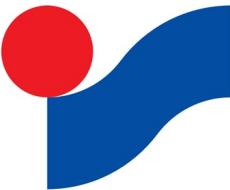
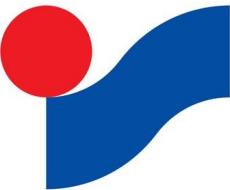
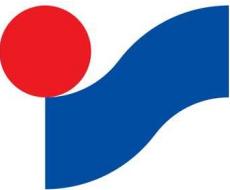
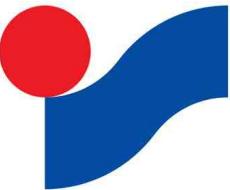


Compressão de Imagem - JPEG

Simão Monteiro 2019215412

1. Compressão de imagens bmp no formato jpeg

1. Comprima as imagens fornecidas segundo o codec JPEG, com qualidade alta.
2. Comprima as imagens fornecidas segundo o codec JPEG, com qualidade média.
3. Comprima as imagens fornecidas segundo o codec JPEG, com qualidade baixa.

Original	Alta(75)	Média(50)	Baixa(25)
			
412Kb	8Kb	7Kb	6Kb
			
580Kb	24Kb	16Kb	12Kb
			
352Kb	28Kb	20Kb	12Kb

4. Compare os resultados e tire conclusões.

Nas imagens acima é possível observar que a imagem *logo* contém transições abruptas entre pixels vizinhos, ou seja, não existem muitas frequências espaciais altas, o que torna a compressão menos eficaz em termos de qualidade. Em relação às imagens mais realistas, como a *peppers* ou a *barn_mountains*, é possível observar que mesmo com compressão alta a distorção do *logo* é muito perceptível. Ainda perceptível, apesar de menos, é a distorção da *peppers* pois já tem uma transição mais suave nos pixels. Por fim, a distorção é muito pouco perceptível na *barn_mountains* devido às suas características fotorealistas. Em termos de compressão e a percepção de distorção, a *barn_mountains* vai ser a melhor, seguida da *peppers* e, por último, do *logo*.

2. Crie duas funções, encoder e decoder, para encapsular as funções a desenvolver nas alíneas 3 a 9

3. Visualização de imagem representada pelo modelo de cor RGB

1. Leia uma imagem .bmp, e.g., a imagem *peppers.bmp*.
2. Crie uma função para implementar um colormap definido pelo utilizador.
3. Crie uma função que permita visualizar a imagem com um dado colormap.
4. Crie uma função para separar a imagem nos seus componentes RGB. Crie também a função inversa.
5. Visualize a imagem e cada um dos canais RGB (com o colormap adequado).

Red	Green	Blue



4. Pré-processamento da imagem: padding

- Crie uma função para fazer padding da imagem. Caso a dimensão da imagem não seja múltipla de 16x16, faça padding da mesma, replicando a última linha e a última coluna em conformidade. Crie também a função inversa. Certifique-se de que recupera a imagem com a dimensão original, visualizando-a.

Unpadded



Shape - (297, 400, 3)

Padded



Shape - (304, 400, 3)

5. Conversão para o modelo cor YCbCr

- Crie uma função para converter a imagem do modelo de cor RGB para o modelo de cor YCbCr. Crie também a função inversa (conversão de YCbCr para RGB). Certifique-se de que consegue obter os valores originais de RGB (teste, por exemplo, com o pixel [0, 0]). Nota: na conversão inversa, garanta que R, G e B sejam número inteiros no intervalo {0, 1, ..., 255}.
- Converta a imagem inicial para o modelo de cor YCbCr.
- Visualize cada um dos canais (com o colormap adequado)

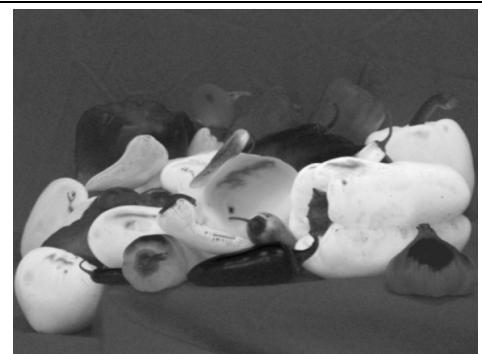
Y



Cb



Cr



R

G

B



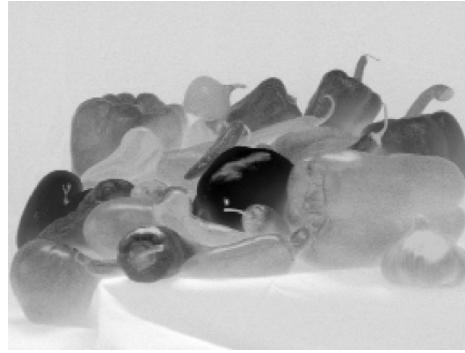
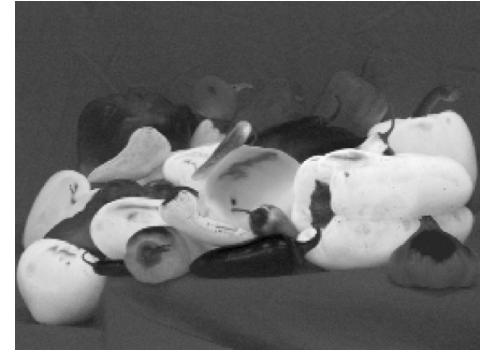
4. Compare a imagem de Y com R, G e B e com Cb e Cr. Tire conclusões.

Na imagem RGB a luminância e o detalhe são distribuídos pelos três canais contendo muita informação redundante. Quando é convertida para YCbCr passam a estar no canal Y. Comparando os dois modelos de cor, podemos observar que o canal Y contém muita informação e detalhe e que os canais da crominância, Cb e Cr, contêm pouca informação, já que não existe tanta sensibilidade por parte do ser humano. Assim, teremos uma possibilidade de compressão desses canais da crominância.

6. Sub-Amostragem

1. Crie uma função para sub-amostrar os canais Y, Cb, e Cr, segundo as possibilidades definidas pelo codec JPEG, a qual deve devolver Y_d, Cb_d e Cr_d. Crie também a função para efectuar a operação inversa, i.e., upsampling. Certifique-se de que consegue reconstruir com exactidão Y, Cb e Cr.
2. Visualize os canais Y_d, Cb_d e Cr_d com downsampling 4:2:0. Apresente as dimensões das matrizes correspondentes.

4:2:0

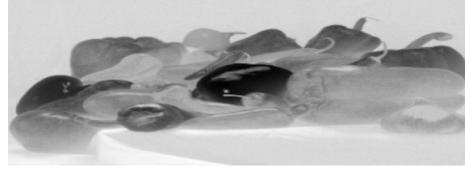
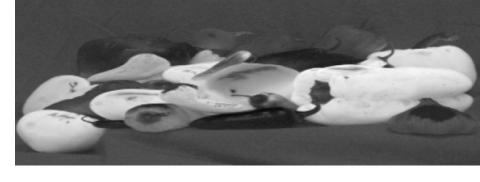
Y	Cb	Cr
		

(384, 512)

(192, 256)

(192, 256)

4:2:2

Y	Cb	Cr
		

(384, 512)

(192, 512)

(192, 512)

3. Apresente e analise a taxa de compressão alcançada para as variantes de downsampling 4:2:2 e 4:2:0 (taxa de compressão, destrutividade, etc.)

No downsampling 4:2:2 os canais da crominância, Cb e Cr, são amostrados a metade, ou seja, a resolução dos canais é reduzida a metade. Isto traz uma taxa de compressão de 1/3 à imagem original. Já no downsampling 4:2:0, tanto a resolução horizontal como vertical são reduzidas a metade, o que traz uma taxa de compressão de 1/2 em relação à imagem original. Neste passo já existiu uma grande compressão da imagem.

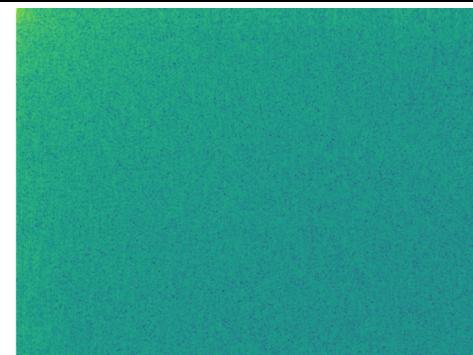
7. Transformada de Coseno Discreta (DCT)

1. DCT nos canais completos

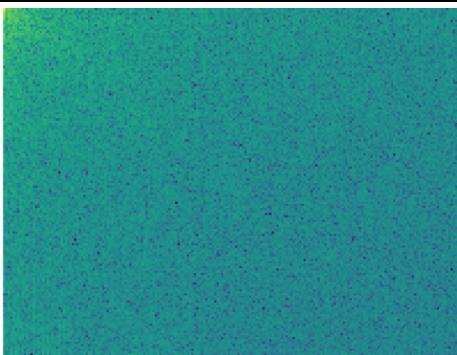
- Crie uma função para calcular a DCT de um canal completo. Utilize a função `scipy.fftpack.dct`. Crie também a função inversa (usando `scipy.fftpack.idct`). Certifique-se de que consegue obter os valores originais de `Y_d`, `Cb_d` e `Cr_d`. Nota: para uma matriz, `X`, com duas dimensões, deverá fazer: `X_dct = dct(dct(X, norm="ortho").T, norm="ortho").T`
- Aplique a função desenvolvida a `Y_d`, `Cb_d`, `Cr_d` e visualize as imagens obtidas (`Y_dct`, `Cb_dct`, `Cr_dct`). Sugestão: atendendo à gama ampla de valores da DCT, visualize as imagens usando uma transformação logarítmica, e.g., de acordo com o seguinte pseudo-código:
`imshow(log(abs(X) + 0.0001))`

barn_mountains.bmp

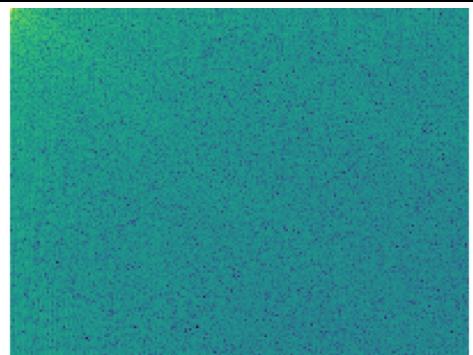
Y - DCT log



Cb - DCT log



Cr - DCT log



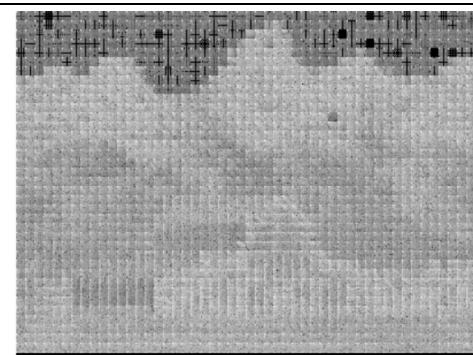
3. Discuta os resultados obtidos em termos de potencial de compressão.

Nas imagens acima, é possível observar que as frequências espaciais mais baixas e mais importantes concentram-se todas no canto superior esquerdo. O resto são frequências espaciais altas e é possível eliminá-las ou reduzir o número de bits para as representar.

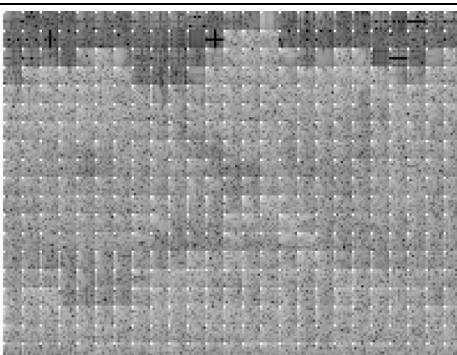
2. DCT em blocos 8x8

- Usando as mesmas funções para cálculo da DCT, crie uma função que calcule a DCT de um canal completo em blocos BSxBS. Crie também a função inversa (IDCT BSxBS). Certifique-se de que consegue obter os valores originais de `Y_d`, `Cb_d` e `Cr_d`.
- Aplice a função desenvolvida (DCT) a `Y_d`, `Cb_d`, `Cr_d` com blocos 8x8 e visualize as imagens obtidas (`Y_DCT8`, `Cb_DCT8`, `Cr_DCT8`).

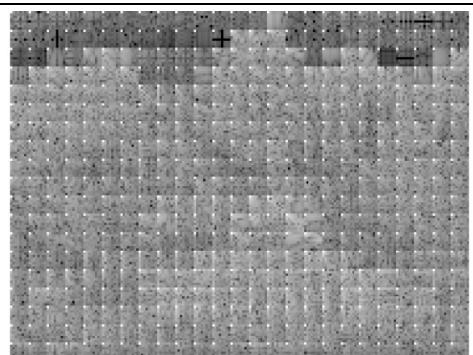
Y_d



Cb_d



Cr_d



Y_d Inv



Cb_d Inv



Cr_d Inv



3. Compare os resultados obtidos com os resultados de 7.1.2 e discuta-os em termos de potencial de compressão.

Como podemos observar, a imagem está dividida igualmente em blocos 8x8. Visualizando os resultados da DCT 8x8, notamos que aqui existe um enorme potencial de compressão pois, como em 7.1.2, as frequências baixas encontram-se concentradas no canto superior esquerdo. Ao representar a grande quantidade de frequências mais altas com uma qualidade mais reduzida (menos bits) será possível diminuir bastante o tamanho da imagem.

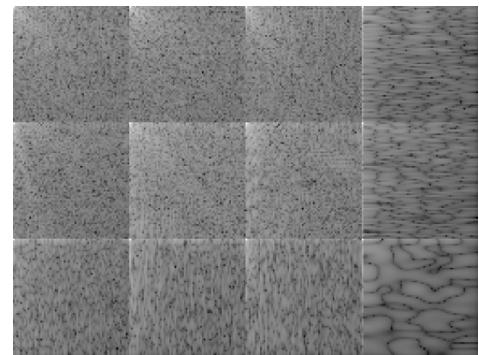
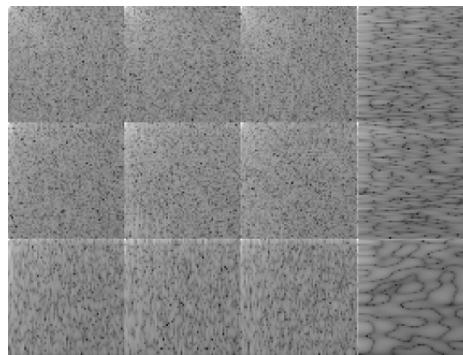
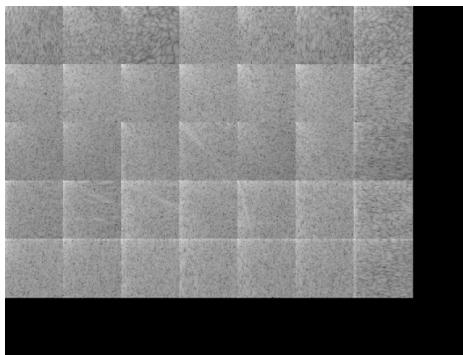
3. DCT em blocos 64x64.

- Repita 7.2.

Y_d

Cb_d

Cr_d



2. Compare com os resultados anteriores e tire conclusões.

Com a DCT em blocos 64x64 é possível visualizar que os blocos são demasiados grandes para apanhar as pequenas diferenças entre pixels, ou seja, a compressão não irá ser de tanta qualidade. Como os blocos são grandes, irão ser mais visíveis ao nível do olho humano.

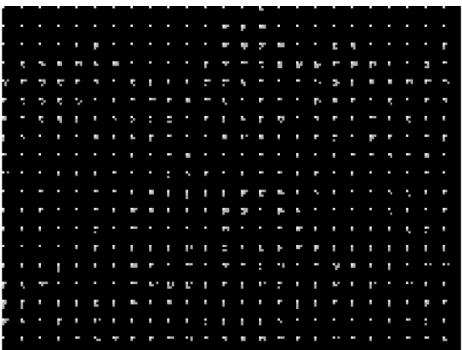
8. Quantização

1. Crie uma função para quantizar os coeficientes da DCT para cada bloco 8x8. Crie também a função inversa.
2. Quantize os coeficientes da DCT, usando os seguintes factores de qualidade: 10, 25, 50, 75 e 100. Visualize as imagens obtidas.

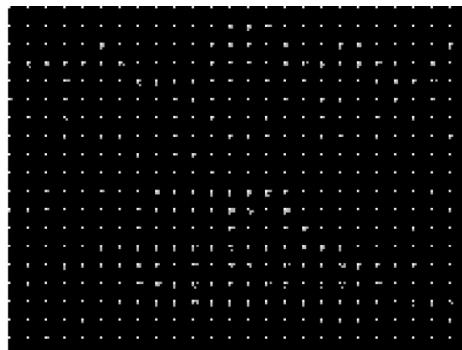
Y Channel	Cb Channel	Cr Channel
Muito Alta(100)	Muito Alta(100)	Muito Alta(100)
Alta(75)	Alta(75)	Alta(75)
Média(50)	Média(50)	Média(50)
Baixa(25)	Baixa(25)	Baixa(25)



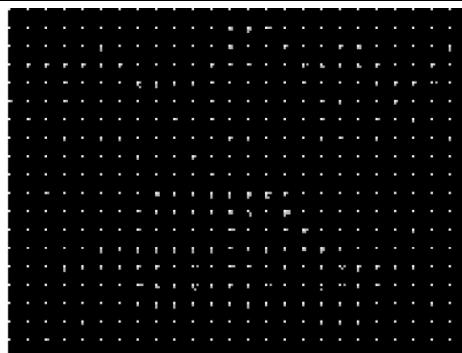
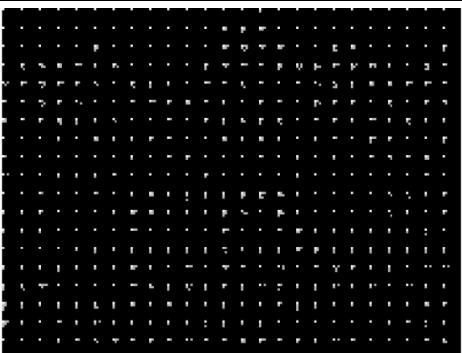
Muito Baixa(10)



Muito Baixa(10)



Muito Baixa(10)

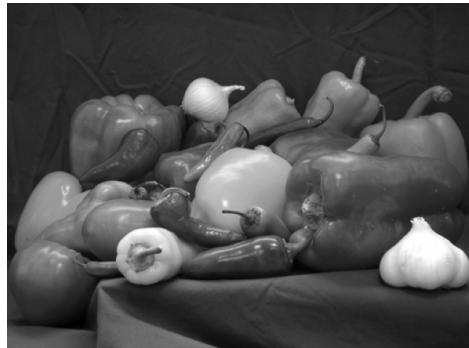


3. Compare os resultados obtidos com os vários factores de qualidade e discuta-os em termos de potencial de compressão

Com a quantização é possível observar a representação dos diferentes coeficientes representados, com mais ou menos bits, dependendo da sua frequência espacial. Nas imagens acima, consegue-se observar bem os efeitos que tem a quantização. Isto porque em cada bloco 8x8 são representados os coeficientes com mais informação no canto superior esquerdo, enquanto que no resto do bloco encontram-se a 0 ou perto disso.

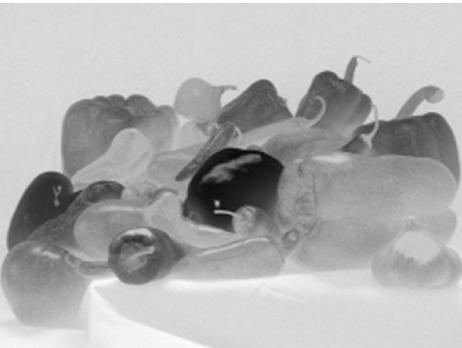
Y Channel

Muito Alta(100)



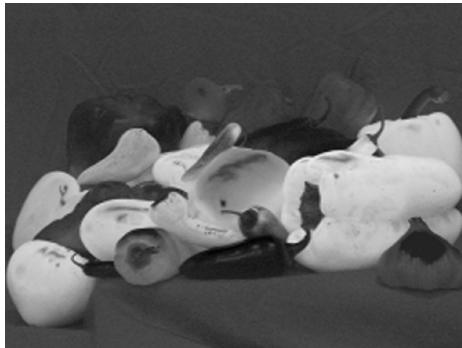
Cb Channel

Muito Alta(100)

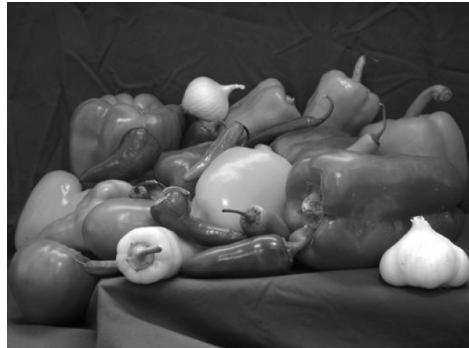


Cr Channel

Muito Alta(100)



Alta(75)



Alta(75)



Alta(75)



Média(50)



Média(50)



Média(50)





Baixa(25)



Baixa(25)



Baixa(25)



Muito Baixa(10)



Muito Baixa(10)



Muito Baixa(10)



4. Compare os resultados obtidos com os resultados da alínea 5 e tire conclusões.

Comparando as imagens obtidas com os resultados da alínea 5, observa-se que os canais Cb e Cr ganham bastante distorção e perdem bastante informação à medida que o fator de qualidade diminui. Isto acontece porque são os canais que nos dão a maior possibilidade de compressão. No canal Y é quase imperceptível a perda de informação à medida que os fatores de qualidade diminuem. Como podemos observar, o fator de qualidade 75 tem a melhor qualidade/compressão, pois é aquele em que a distorção não é enorme e oferece uma compressão boa.

9. Codificação DPCM dos coeficientes DC

1. Crie uma função para realizar a codificação dos coeficientes DC de cada bloco. Em cada bloco, substitua o valor DC pelo valor da diferença. Crie também a função inversa.
2. Aplique a sua função aos valores da DCT quantizada.

Y Channel

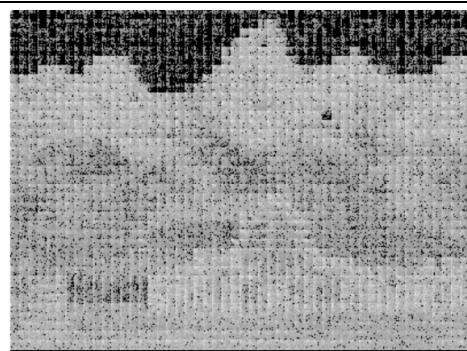
Muito Alta(100)

Cb Channel

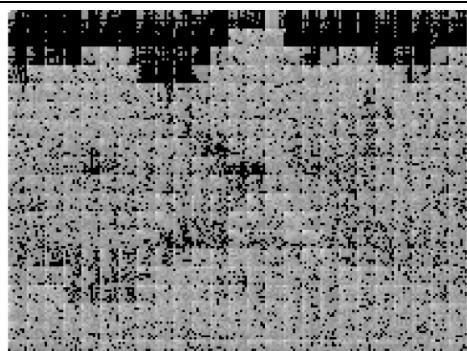
Muito Alta(100)

Cr Channel

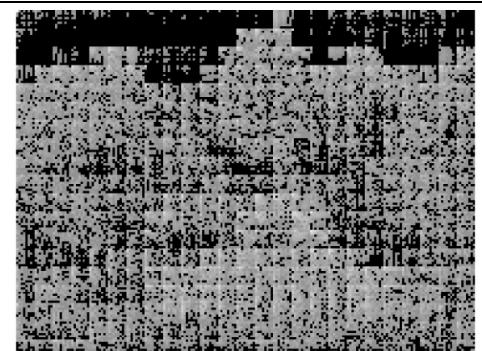
Muito Alta(100)



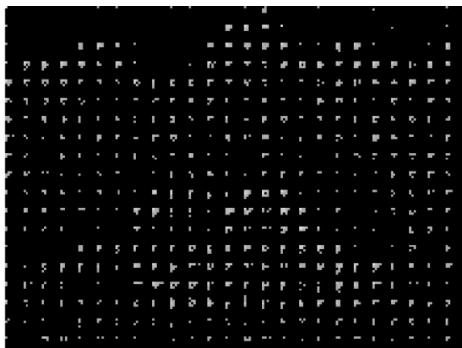
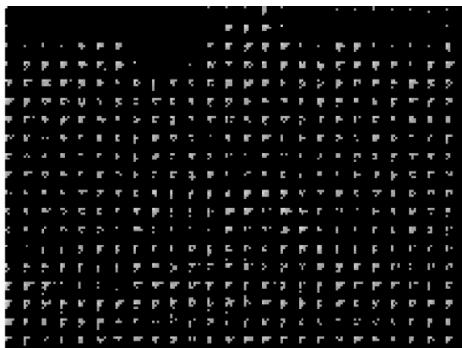
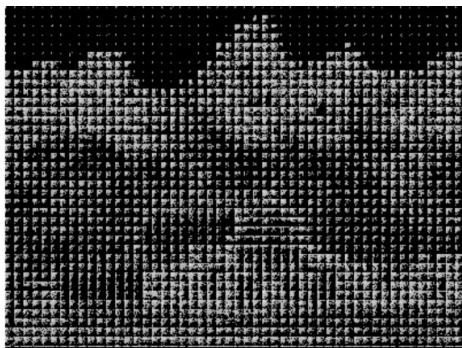
Alta(75)



Alta(75)



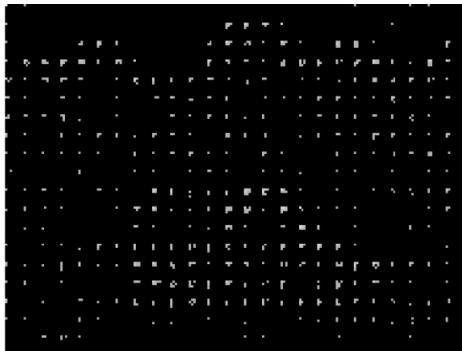
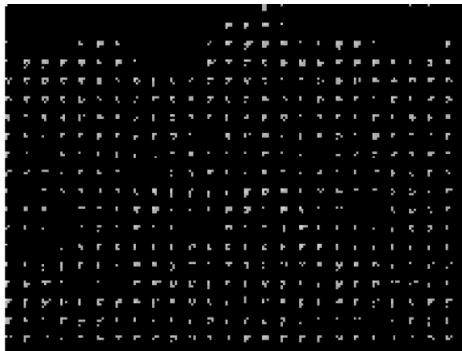
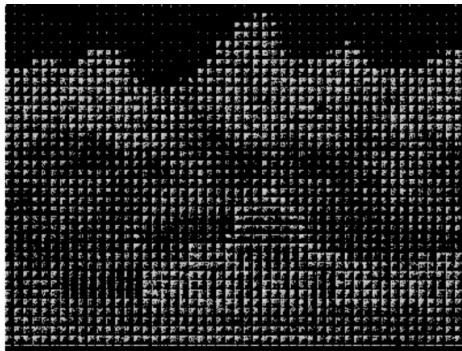
Alta(75)



Média(50)

Média(50)

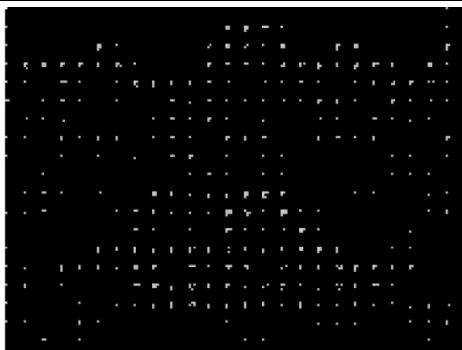
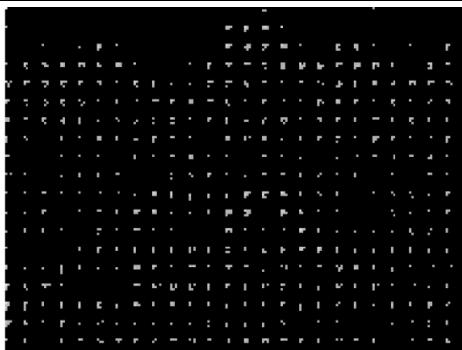
Média(50)



Baixa(25)

Baixa(25)

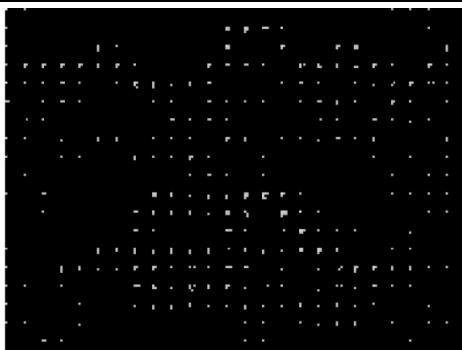
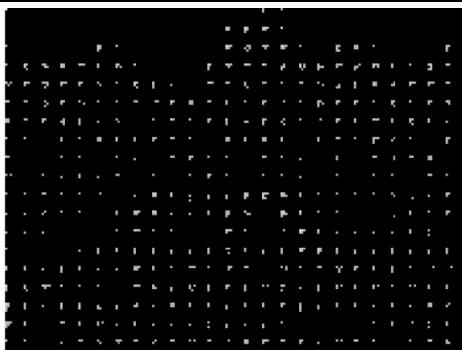
Baixa(25)



Muito Baixa(10)

Muito Baixa(10)

Muito Baixa(10)



3. Analise os resultados e tire conclusões.

Visualizando as imagens da DPCM, é possível observar a remoção das redundâncias em comparação às imagens da quantização. Assim é possível ainda uma maior compressão que não é destrutiva, ou seja, que poderá ser recuperada na descompressão.

Os fatores de qualidade vão ter influência na produção dessa redundância, como mostram as imagens acima, já que aquelas com menor fator de qualidade terão uma possibilidade maior de remover redundâncias.

10. Codificação e decodificação end-to-end

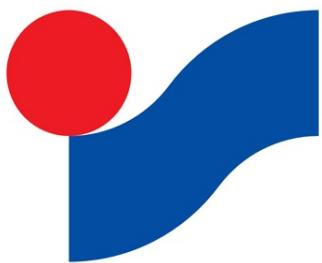
1. Codifique as imagens fornecidas com os seguintes parâmetros de qualidade: 10, 25, 50, 75 e 100
2. **Visualize as imagens descodificadas. Visualize também a imagem das diferenças entre o canal Y de cada uma das imagens originais e da imagem descodificada respectiva para cada um dos factores de qualidade testados. Calcule as várias métricas de distorção (MSE, RMSE, SNR e PSNR) para cada uma das imagens e factores de qualidade. Tire conclusões.**

Imagens Descodificadas

Original

Original

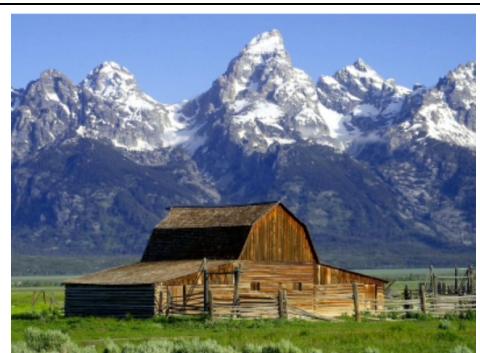
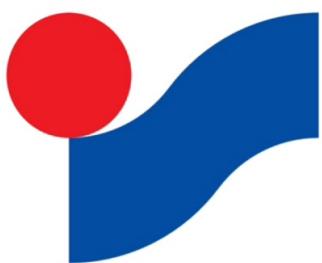
Original



Muito Alta(100)

Muito Alta(100)

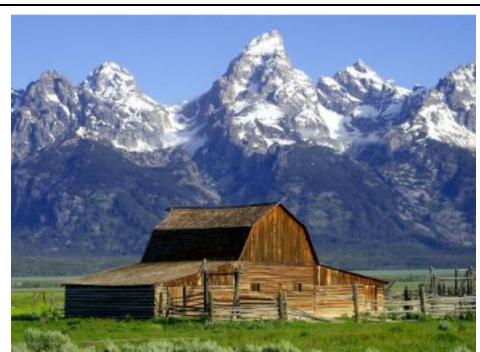
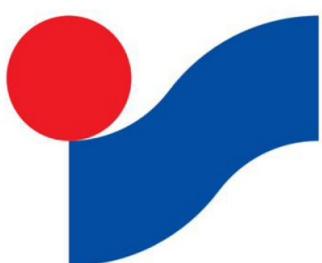
Muito Alta(100)



Alta(75)

Alta(75)

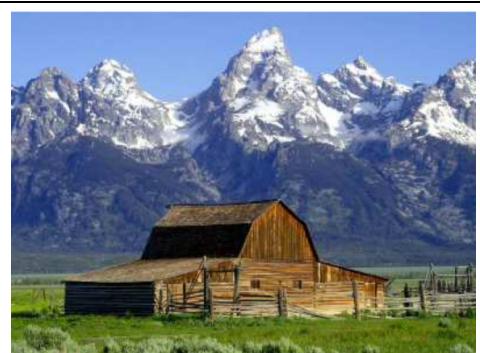
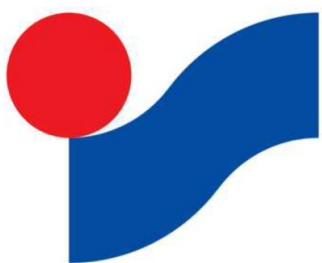
Alta(75)



Média(50)

Média(50)

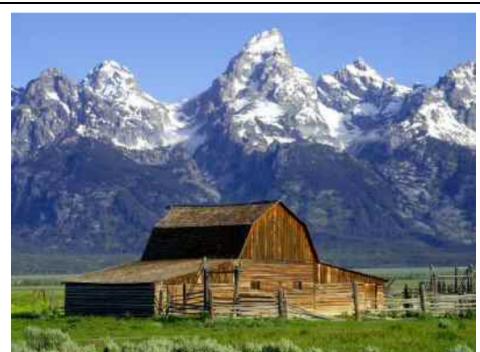
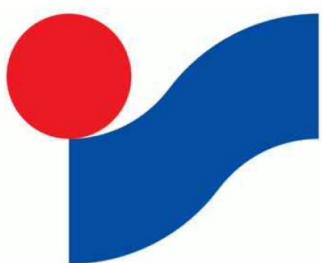
Média(50)



Baixa(25)

Baixa(25)

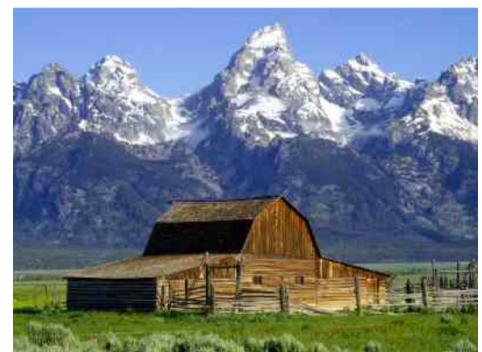
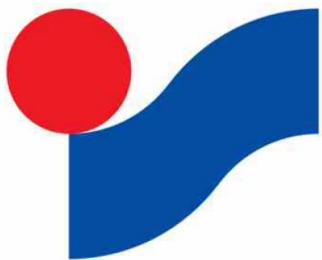
Baixa(25)



Muito Baixa(10)

Muito Baixa(10)

Muito Baixa(10)



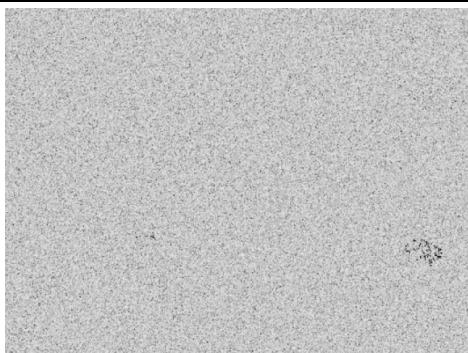
Nas imagens acima é possível observar que a imagem *logo* ganha distorção mesmo com qualidades altas, o que acontece menos na *peppers* e ainda menos na *barn mountains*. Penso que a qualidade ideal para usar o JPEG seria a 75, dependendo da imagem, visto que para qualidades inferiores a imagem começa a ganhar uma distorção já muito visível ao olho humano. A interpolação cúbica ajudaria bastante a eliminar algum desse ruído, tornando a imagem mais nítida.

Diferenças do Canal Y

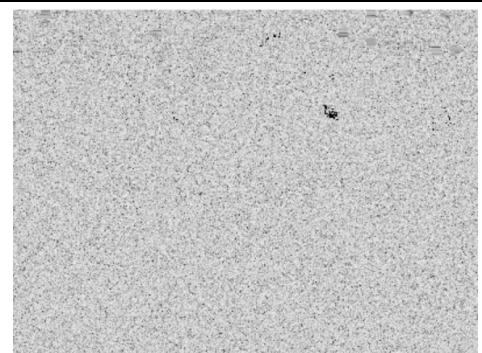
Diff(100)



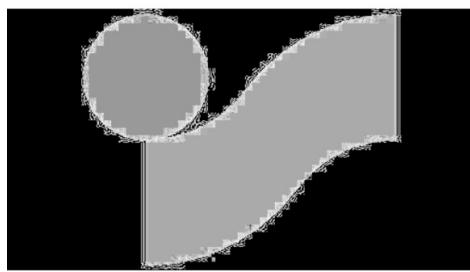
Diff(100)



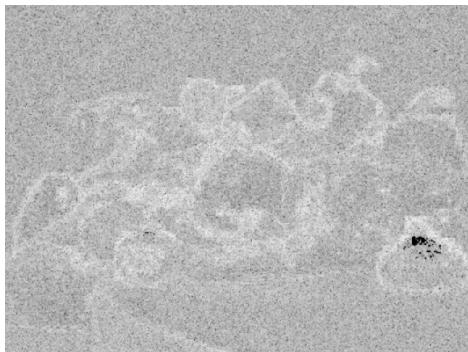
Diff(100)



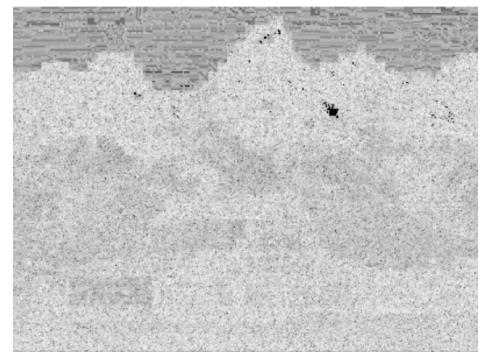
Diff(75)



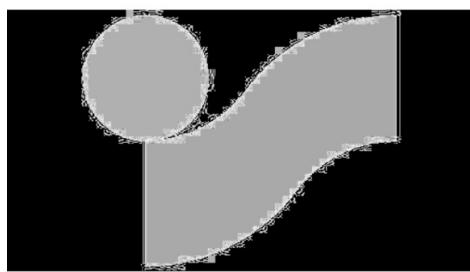
Diff(75)



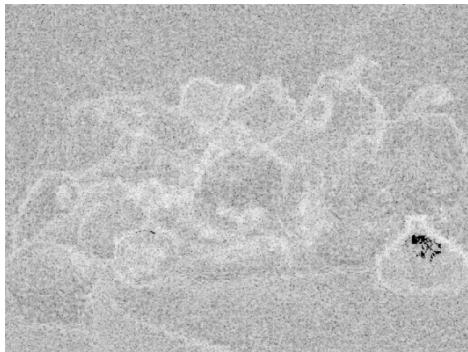
Diff(75)



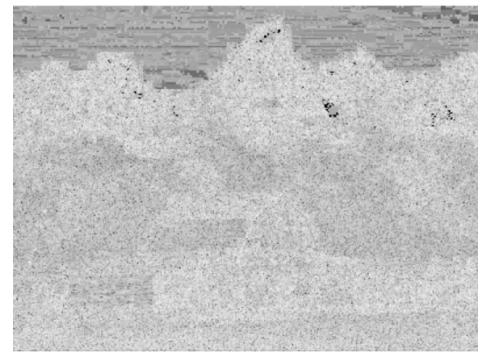
Diff(50)

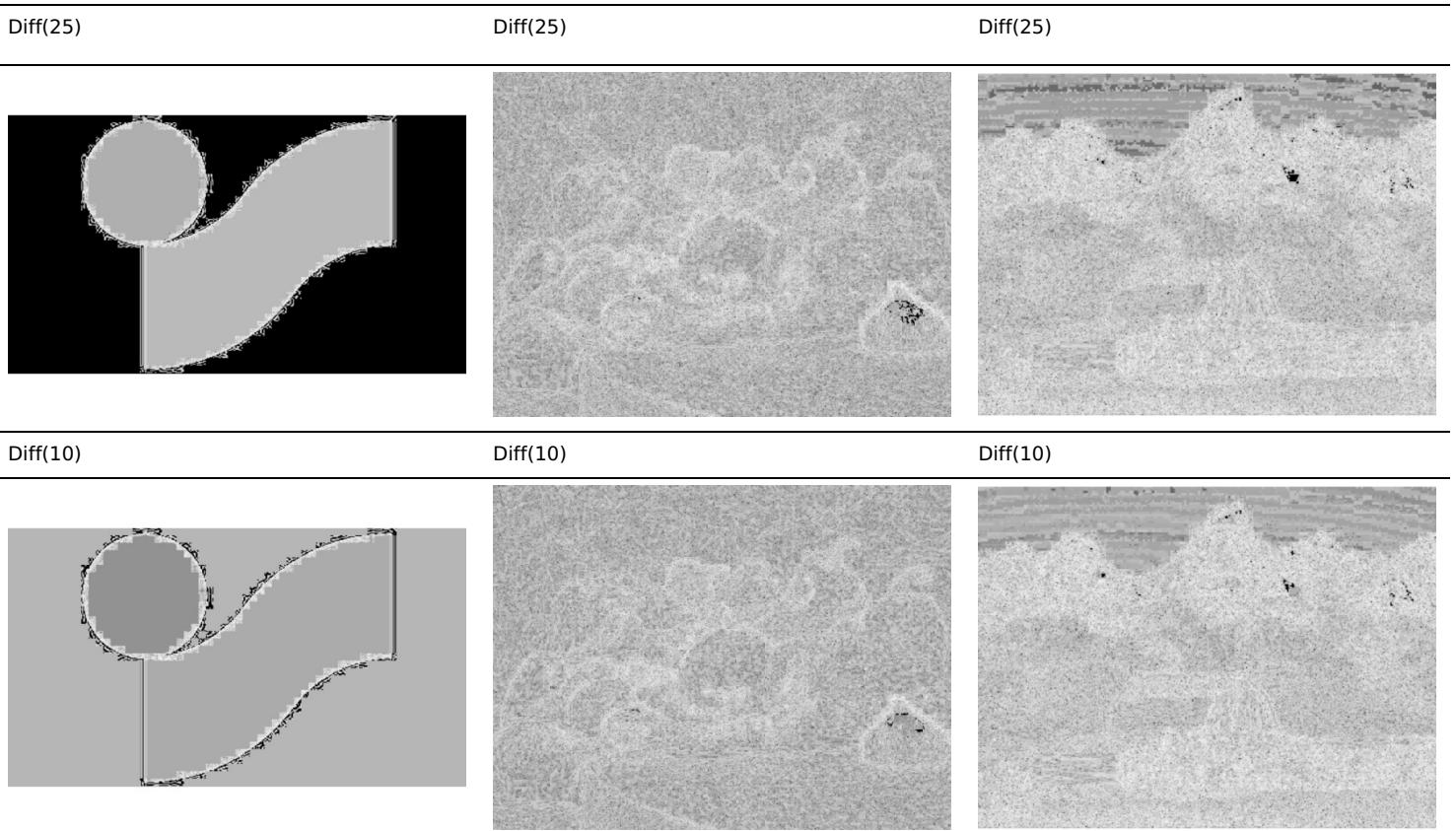


Diff(50)



Diff(50)





Na diferença entre canal Y das imagens originais e descodificadas, conseguimos observar que o *logo* perde bastante informação visível ao olho humano. Na *peppers* e na *barn mountais*, com um fator de qualidade 100, a diferença é apenas ruído, que não é visível pelo ser humano. A partir da qualidade 75 para baixo começa a perder-se informação mais sensível ao olho humano, sendo que a diferença já mostra elementos das diferentes imagens.

Logo Distortion

Distortion Calculation	CompressionRate(10)	CompressionRate(25)	CompressionRate(50)	CompressionRate(75)	CompressionRate(100)
MSE	100.3356512455516	96.39738078291815	84.93334519572954	62.13719572953737	43.08691103202847
RMSE	10.016768503142698	9.818216782232819	9.215928884042539	7.882714997355757	6.564062083194252
SNR	31.317565894823957	31.491466295313742	32.04133633761491	33.398602138420934	34.98866503407382
PSNR	28.116250869884986	28.290151270374764	28.840021312675933	30.197287113481963	31.78735000913484

Peppers Distortion

Distortion Calculation	CompressionRate(10)	CompressionRate(25)	CompressionRate(50)	CompressionRate(75)	CompressionRate(100)
MSE	177.6107635498047	161.6694132486979	133.53300984700522	104.56081644694011	62.27788289388021
RMSE	13.32706882813339	12.714928755156196	11.55564839578486	10.225498347119329	7.891633727808217
SNR	22.431259359027575	22.839673550923116	23.670065790615137	24.732262515068832	26.982613766949445
PSNR	25.636110795568612	26.04452498746415	26.87491722715617	27.937113951609874	30.18746520349049

Barn Mountains Distortion

Distortion Calculation	CompressionRate(10)	CompressionRate(25)	CompressionRate(50)	CompressionRate(75)	CompressionRate(100)
MSE	406.7184848484848	369.03262626262625	293.16542087542086	187.50007575757576	55.53707912457912
RMSE	20.167262700934028	19.21022192122273	17.122074082173015	13.693066703904416	7.4523203852611655
SNR	21.0933366982159	21.515628168825938	22.515148372520578	24.45626133281708	29.740445457608157

Poderíamos pensar que no cálculo da distorção o *logo* tivesse o maior valor, seguido da *peppers* e, por último, da *barn mountains*. Isto não se observa, já que estamos a calcular a distorção matemática e não perceptual. Apesar de a distorção ser mais visível nessa ordem decrescente, o *logo* tem cores muito frequentes e transições abruptas. Assim, o cálculo da distorção é menor, pois apenas nesses limites é que a imagem vai variar mais comparativamente à original, originando num erro menor. Nas fotos mais fotorealistas existem mais transições suaves que causam um erro maior.

3. Volte a analisar o ponto 1, de forma a validar/complementar as conclusões tiradas nesse ponto.

Em forma de conclusão, depreendo que este trabalho prático validou as afirmações feitas inicialmente, sendo que existe uma maior possibilidade de compressão de imagens realistas, já que contêm uma maior quantidade de frequências espaciais altas. Tal como dito no ponto 1, o fator qualidade/compressão iria ser melhor na *barn mountains*, seguido da *peppers* e, por fim, do *logo*. Na *peppers* e *barn mountain*, uma qualidade de 75 na compressão é aceitável e perto de invisível ao olho humano. No *logo*, isto já não é tão eficaz, uma vez que se observa uma distorção bastante notável. Este trabalho prático ajudou-me a compreender como funciona a compressão JPEG e os diferentes conceitos como modelos de cor, padding, subsampling, para além da importância da DCT, quantização e DPCM neste tipo de compressão.

Code

```

import os.path
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.colors as clr
import scipy.fftpack as fft
import cv2
import math as m

# Global Variables
Ct = np.array([[0.299, 0.587, 0.114],
               [-0.168736, -0.331264, 0.5],
               [0.5, -0.418688, -0.081312]]))

QY = np.array([[16, 11, 10, 16, 24, 40, 51, 61],
               [12, 12, 14, 19, 26, 58, 60, 55],
               [14, 13, 16, 24, 40, 57, 69, 56],
               [14, 17, 22, 29, 51, 87, 80, 62],
               [18, 22, 37, 56, 68, 109, 103, 77],
               [24, 35, 55, 64, 81, 104, 113, 92],
               [49, 64, 78, 87, 103, 121, 120, 101],
               [72, 92, 95, 98, 112, 100, 103, 99]]))

QC = np.array([[17, 18, 24, 47, 99, 99, 99, 99],
               [18, 21, 26, 66, 99, 99, 99, 99],
               [24, 26, 56, 99, 99, 99, 99, 99],
               [47, 66, 99, 99, 99, 99, 99, 99],
               [99, 99, 99, 99, 99, 99, 99, 99],
               [99, 99, 99, 99, 99, 99, 99, 99],
               [99, 99, 99, 99, 99, 99, 99, 99],
               [99, 99, 99, 99, 99, 99, 99, 99]]))

# JPEG Compression
def main():
    print("\n#### - JPEG Encoder/Decoder - ####")
    while True:
        file = input("Enter file name: ")
        if not os.path.isfile(file):
            print("File not found!")
        else:
            break

    greyCm = colormap([(0, 0, 0), (1, 1, 1)])
    global size
    img = readImg(file)
    size = img.shape

    while True:
        subsampling = input("\nSubsampling:\n1 - 4:2:0\n2 - 4:2:2\nChoose: ")
        try:
            subsampling = int(subsampling)
            if subsampling == 1 or subsampling == 2:
                break
        except:
            print("Choice not valid!\n")

    while True:
        cubic = input(
            "\nDo you want cubic interpolation?\n1 - Yes\n2 - No\nChoose: ")
        try:
            cubic = int(cubic)
            if cubic == 1 or cubic == 2:
                break
        except:
            print("Choice not valid!\n")

```

```

        break
    except:
        print("Choice not valid!\n")

if (cubic == 1):
    interpolation = True
elif (cubic == 2):
    interpolation = False

while True:
    compressionRate = input(
        "\nPick a compression quality between 1 and 100\nCompression rate: ")
    try:
        compressionRate = int(compressionRate)
        if compressionRate >= 1 and compressionRate <= 100:
            break
    except:
        print("Choice not valid!")

# Encode Image to JPEG
jpegEncoded = encoder(img, compressionRate, interpolation, subsampling)
jpegDecoded = decoder(jpegEncoded, compressionRate,
                      interpolation, subsampling)

# Visualize Original Image
visualize(img)

# Visualize Decoded Image
visualize(jpegDecoded)

# Get Distortion
getDistortion(img, jpegDecoded)

def encoder(img, compressionRate, interpolation, subsampling):
    if (subsampling == 1):
        subX = 2
        subY = 2
    elif (subsampling == 2):
        subX = 2
        subY = 1
    paddedImg = padding(img, 16)
    ycbcrImg = rgb_to_ycbcr(paddedImg)
    Y_d, Cb_d, Cr_d = subSampling(ycbcrImg, subX, subY, interpolation)
    Y_dct, Cb_dct, Cr_dct = dct(Y_d, Cb_d, Cr_d, 8)
    Y_quant, Cb_quant, Cr_quant = quantization(
        Y_dct, Cb_dct, Cr_dct, compressionRate, 8)
    Y_quant, Cb_quant, Cr_quant = dpcm(Y_quant, Cb_quant, Cr_quant)

    return Y_quant, Cb_quant, Cr_quant

# Decode JPEG Image
def decoder(img, conversionRate, interpolation, subsampling):
    if (subsampling == 1):
        subX = 2
        subY = 2
    elif (subsampling == 2):
        subX = 2
        subY = 1
    Y_invDpcm, Cb_invDpcm, Cr_invDpcm = invDpcm(img[0], img[1], img[2])
    Y_invQuant, Cb_invQuant, Cr_invQuant = invQuantization(
        Y_invDpcm, Cb_invDpcm, Cr_invDpcm, conversionRate, 8)
    Y_invDct, Cb_invDct, Cr_invDct = invDct(
        Y_invQuant, Cb_invQuant, Cr_invQuant, 8)
    imgUp = upSampling(Y_invDct, Cb_invDct, Cr_invDct,
                        subX, subY, interpolation)
    rgbImg = ycbcr_to_rgb(imgUp)
    decodedImg = unpading(rgbImg, size)

    return decodedImg

# Reads Image Data
def readImg(file):
    image = plt.imread(file)
    return image

# Visualize Image
def visualize(img, cm=None, log=False):
    if log:

```

```

    img = np.log(np.abs(img) + 0.0001)
    plt.figure()
    plt.imshow(img, cm)
    plt.axis('off')
    plt.show()

# Define Colormap
def colormap(value):
    nl = 5
    nc = 50
    linGray = np.linspace(0., 1., nc).reshape(1, nc)
    linGray = np.repeat(linGray, nl, axis=1).reshape(nc, nl).T
    linGrayImg = np.zeros((nl, nc, 3))
    linGrayImg[:, :, 0] = linGray
    linGrayImg[:, :, 1] = linGray
    linGrayImg[:, :, 2] = linGray
    cm = clr.LinearSegmentedColormap.from_list('cmap', value, N=256)
    return cm

# Separates RGB Channels
def channelSep(imgData):
    red = imgData[:, :, 0]
    green = imgData[:, :, 1]
    blue = imgData[:, :, 2]

    return red, green, blue

# Unifie RGB Channels
def channelUni(red, green, blue):
    auxImage = np.array(red)
    auxImage[:, :, 1] = green[:, :, 1]
    auxImage[:, :, 2] = blue[:, :, 2]

    return auxImage

# Visualize RGB Channels
def visualizeRGB(red, green, blue):
    redCm = colormap([(0, 0, 0), (1, 0, 0)])
    greenCm = colormap([(0, 0, 0), (0, 1, 0)])
    blueCm = colormap([(0, 0, 0), (0, 0, 1)])

    visualize(red, redCm)
    visualize(green, greenCm)
    visualize(blue, blueCm)

# Pad Image
def padding(img, n):
    imgShape = img.shape
    red = img[:, :, 0]
    green = img[:, :, 1]
    blue = img[:, :, 2]

    # Número de linhas a adicionar
    if imgShape[0] % n != 0:
        r = n - (imgShape[0] % n)
        red = np.vstack([red, np.tile(red[-1, :], (r, 1))])
        green = np.vstack([green, np.tile(green[-1, :], (r, 1))])
        blue = np.vstack([blue, np.tile(blue[-1, :], (r, 1))])

    # Número de colunas a adicionar
    if imgShape[1] % n != 0:
        c = n - (imgShape[1] % n)
        red = np.column_stack([red, np.tile(red[:, -1], (c, 1)).T])
        green = np.column_stack([green, np.tile(green[:, -1], (c, 1)).T])
        blue = np.column_stack([blue, np.tile(blue[:, -1], (c, 1)).T])

    paddedImg = np.zeros((red.shape[0], red.shape[1], 3), np.uint8)
    paddedImg[:, :, 0] = red
    paddedImg[:, :, 1] = green
    paddedImg[:, :, 2] = blue

    return paddedImg

# Unpad Image
def unpadding(paddedImg, shapeImg):
    unpaddingImg = paddedImg[:shapeImg[0], :shapeImg[1], :]

```

```

return unpaddedImg

# Convert RGB to YCbCr
def rgb_to_ycbcr(img):
    convertedImg = img.dot(Ct.T)
    convertedImg[:, :, [1, 2]] += 128

    return convertedImg

# Convert YCbCr to RGB
def ycbcr_to_rgb(img):
    invertedCt = np.linalg.inv(Ct.T)
    img[:, :, [1, 2]] -= 128
    convertedImg = img.dot(invertedCt)
    convertedImg = convertedImg.round()
    convertedImg[convertedImg > 255] = 255
    convertedImg[convertedImg < 0] = 0
    convertedImg = convertedImg.astype(np.uint8)

    return convertedImg

# Visualize YCbCr Channels
def visualizeYCbCr(ycbcrImg):
    greyCm = colormap([(0, 0, 0), (1, 1, 1)])

    visualize(ycbcrImg[:, :, 0], greyCm)
    visualize(ycbcrImg[:, :, 1], greyCm)
    visualize(ycbcrImg[:, :, 2], greyCm)

# SubSampling Image
def subSampling(img, factorX, factorY, interpolation):
    Y_d = img[:, :, 0]
    if(interpolation):
        Cb_d = cv2.resize(img[:, :, 1], None, fx=1/factorX,
                          fy=1/factorY, interpolation=cv2.INTER_CUBIC)
        Cr_d = cv2.resize(img[:, :, 2], None, fx=1/factorX,
                          fy=1/factorY, interpolation=cv2.INTER_CUBIC)
    else:
        Cb_d = img[:, :, 1][::factorX, ::factorY]
        Cr_d = img[:, :, 2][::factorX, ::factorY]

    return Y_d, Cb_d, Cr_d

# UpSampling Image
def upSampling(Y_d, Cb_d, Cr_d, factorX, factorY, interpolation):
    Y_u = Y_d
    if(interpolation):
        Cb_u = cv2.resize(Cb_d, None, fx=factorX, fy=factorY,
                           interpolation=cv2.INTER_CUBIC)
        Cr_u = cv2.resize(Cr_d, None, fx=factorX, fy=factorY,
                           interpolation=cv2.INTER_CUBIC)
    else:
        Cb_u = np.repeat(Cb_d, factorX, axis=0)
        Cr_u = np.repeat(Cr_d, factorX, axis=0)
        Cb_u = np.repeat(Cb_u, factorY, axis=1)
        Cr_u = np.repeat(Cr_u, factorY, axis=1)

    imgUp = np.dstack([Y_u, Cb_u, Cr_u])

    return imgUp

# DCT Image
def dct(Y_dct, Cb_dct, Cr_dct, n):
    sizeY = Y_dct.shape
    sizeC = Cb_dct.shape
    for i in range(0, sizeY[0], n):
        for j in range(0, sizeY[1], n):
            Y_dct[i:i + n, j:j + n] = fft.dct(fft.dct(Y_dct[i:i + n, j:j + n], norm="ortho").T, norm="ortho").T
            if i < sizeC[0] and j < sizeC[1]:
                Cb_dct[i:i + n, j:j + n] = fft.dct(
                    fft.dct(Cb_dct[i:i + n, j:j + n], norm="ortho").T, norm="ortho").T
                Cr_dct[i:i + n, j:j + n] = fft.dct(
                    fft.dct(Cr_dct[i:i + n, j:j + n], norm="ortho").T, norm="ortho").T

```

```

return Y_dct, Cb_dct, Cr_dct

# Inverse DCT Image
def invDct(Y_invDct, Cb_invDct, Cr_invDct, n):
    sizeY = Y_invDct.shape
    sizeC = Cb_invDct.shape
    for i in range(0, sizeY[0], n):
        for j in range(0, sizeY[1], n):
            Y_invDct[i:i + n, j:j + n] = fft.idct(fft.idct(Y_invDct[i:i + n, j:j + n], norm="ortho").T, norm="ortho").T
            if i < sizeC[0] and j < sizeC[1]:
                Cb_invDct[i:i + n, j:j + n] = fft.idct(
                    fft.idct(Cb_invDct[i:i + n, j:j + n], norm="ortho").T, norm="ortho").T
                Cr_invDct[i:i + n, j:j + n] = fft.idct(
                    fft.idct(Cr_invDct[i:i + n, j:j + n], norm="ortho").T, norm="ortho").T

    Y_invDct[Y_invDct < 0] = 0
    Y_invDct[Y_invDct > 255] = 255

    Cb_invDct[Cb_invDct < 0] = 0
    Cb_invDct[Cb_invDct > 255] = 255

    Cr_invDct[Cr_invDct < 0] = 0
    Cr_invDct[Cr_invDct > 255] = 255

# Verificar se a DCT Inversa está correta
# diffY = Y_d - Y_invDct
# diffY[diffY < 0.000001] = 0.

# diffCb = Cb_d - Cb_invDct
# diffCb[diffCb < 0.000001] = 0.

# diffCr = Cr_d - Cr_invDct
# diffCr[diffCr < 0.000001] = 0.

return Y_invDct, Cb_invDct, Cr_invDct

# Quantitize Image
def quantization(Y_dct, Cb_dct, Cr_dct, qf, n):
    if (qf == 100):
        qy = np.ones((8, 8))
        qc = np.ones((8, 8))
    elif (qf == 50):
        qy = QY
        qc = QC
    else:
        qy = np.round(QY * ((100 - qf) / 50))
        qy[qy > 255] = 255
        qc = np.round(QC * ((100 - qf) / 50))
        qc[qc > 255] = 255

    sizeY = Y_dct.shape
    sizeC = Cb_dct.shape
    for i in range(0, sizeY[0], n):
        for j in range(0, sizeY[1], n):
            Y_dct[i:i + n, j:j + n] = np.round(Y_dct[i:i + n, j:j + n] / qy)
            if i < sizeC[0] and j < sizeC[1]:
                Cb_dct[i:i + n, j:j + n] = np.round(Cb_dct[i:i + n, j:j + n] / qc)
                Cr_dct[i:i + n, j:j + n] = np.round(Cr_dct[i:i + n, j:j + n] / qc)

    return Y_dct, Cb_dct, Cr_dct

# Inverse Quantitize Image
def invQuantization(Y_dct, Cb_dct, Cr_dct, qf, n):
    if (qf == 100):
        qy = np.ones((8, 8))
        qc = np.ones((8, 8))
    elif (qf == 50):
        qy = QY
        qc = QC
    else:
        qy = np.round(QY * ((100 - qf) / 50))
        qy[qy > 255] = 255
        qc = np.round(QC * ((100 - qf) / 50))
        qc[qc > 255] = 255

    sizeY = Y_dct.shape

```

```

sizeC = Cb_dct.shape
for i in range(0, sizeY[0], n):
    for j in range(0, sizeY[1], n):
        Y_dct[i:i + n, j:j + n] = np.round(Y_dct[i:i + n, j: j + n] * qy)
        if i < sizeC[0] and j < sizeC[1]:
            Cb_dct[i:i + n, j:j +
                    n] = np.round(Cb_dct[i:i + n, j:j + n] * qc)
            Cr_dct[i:i + n, j:j +
                    n] = np.round(Cr_dct[i:i + n, j:j + n] * qc)

return Y_dct, Cb_dct, Cr_dct

# DPCM Image
def dpcm(Y_dct, Cb_dct, Cr_dct):
    sizeY = Y_dct.shape
    sizeC = Cb_dct.shape

    for i in range(0, sizeY[0], 8):
        for j in range(0, sizeY[1], 8):
            if (i == 0 and j == 0):
                dcY0 = Y_dct[0, 0]
                dcCb0 = Cb_dct[0, 0]
                dcCr0 = Cr_dct[0, 0]
                continue
            dcY = Y_dct[i, j]
            diffY = dcY - dcY0
            Y_dct[i, j] = diffY
            dcY0 = dcY
            dcY0 = dcY
            if i < sizeC[0] and j < sizeC[1]:
                dcCb = Cb_dct[i, j]
                dcCr = Cr_dct[i, j]

                diffCb = dcCb - dcCb0
                diffCr = dcCr - dcCr0

                Cb_dct[i, j] = diffCb
                Cr_dct[i, j] = diffCr

                dcCb0 = dcCb
                dcCr0 = dcCr

    return Y_dct, Cb_dct, Cr_dct

# Inverse DPCM Image
def invDpcm(Y_dct, Cb_dct, Cr_dct):
    sizeY = Y_dct.shape
    sizeC = Cb_dct.shape
    for i in range(0, sizeY[0], 8):
        for j in range(0, sizeY[1], 8):
            if (i == 0 and j == 0):
                dcY0 = Y_dct[0, 0]
                dcCb0 = Cb_dct[0, 0]
                dcCr0 = Cr_dct[0, 0]
                continue
            dcY = Y_dct[i, j]
            sumY = dcY0 + dcY
            Y_dct[i, j] = sumY
            dcY0 = sumY
            if i < sizeC[0] and j < sizeC[1]:
                dcCb = Cb_dct[i, j]
                dcCr = Cr_dct[i, j]

                sumCb = dcCb0 + dcCb
                sumCr = dcCr0 + dcCr

                Cb_dct[i, j] = sumCb
                Cr_dct[i, j] = sumCr

                dcCb0 = sumCb
                dcCr0 = sumCr

    return [Y_dct, Cb_dct, Cr_dct]

# Calculate Distortions
def getDistortion(imgOr, imgDec):
    imgOr = imgOr.astype(float)
    imgDec = imgDec.astype(float)
    p = np.sum(imgOr**2)/(size[0]*size[1])
    spike = np.max(imgOr)

```

```
mse = np.sum((imgOr - imgDec)**2)/(size[0]*size[1])
rmse = m.sqrt(mse)
snr = 10 * m.log(p/mse, 10)
psnr = 10 * m.log((spike**2)/mse, 10)

print(
    f"\nDistortion Calculations:\nMSE -> {mse}\nRMSE -> {rmse}\nSNR -> {snr}\nPSNR -> {psnr}")

if __name__ == "__main__":
    main()
```