Trabalho prático 1

- Joana Simões, Nº 2019217013
- Samuel Carinhas, Nº2019217199
- Sofia Alves, Nº 2019227240

No âmbito da cadeira de Multimédia, este trabalho prático visa o estudo de vários mecanismos de compressão de imagem usados pelo codec JPEG. O objetivo será aplicar cada um destes e analisar o seu resultado, observando também se o mesmo se encontra de acordo com o esperado, pondo assim à prova os conhecimentos teóricos sobre compressão de imagem.

In [1]:

```
# imports

from PIL import Image
import os
import matplotlib.pyplot as plt
import matplotlib.image as img
import matplotlib.colors as clr
import numpy as np
import copy
import cv2
import scipy.fftpack as fft
import math as m
```

Compressão de imagens bmp no formato JPEG

In [2]:

```
img1 = Image.open('./imagens/peppers.bmp')
img1 = img1.convert('RGB')
img1.save('./imagens/peppers_compressed_ultra_high.jpg', quality=100)
img1.save('./imagens/peppers_compressed_high.jpg', quality=75)
img1.save('./imagens/peppers_compressed_medium.jpg', quality=50)
img1.save('./imagens/peppers_compressed_low.jpg', quality=25)

img2 = Image.open('./imagens/logo.bmp')
img2 = img2.convert('RGB')
img2.save('./imagens/logo_compressed_ultra_high.jpg', quality=100)
img2.save('./imagens/logo_compressed_high.jpg', quality=75)
img2.save('./imagens/logo_compressed_medium.jpg', quality=50)
img2.save('./imagens/logo_compressed_low.jpg', quality=25)

img3 = Image.open('./imagens/barn_mountains.bmp')
img3 = img3.convert('RGB')
img3.save('./imagens/barn_mountains_compressed_ultra_high.jpg', quality=100)
img3.save('./imagens/barn_mountains_compressed_high.jpg', quality=75)
img3.save('./imagens/barn_mountains_compressed_medium.jpg', quality=50)
img3.save('./imagens/barn_mountains_compressed_low.jpg', quality=25)
```

Resultados e Conclusões

	Original	Ultra Hight	High	Medium	Low	Animation
Images						

	Original	Ultra Hight	High	Medium	Low	Animation
Quality	---	100%	75%	50%	25%	100% - 0%
Size (pixeis)	281 x 500	281 x 500	281 x 500	281 x 500	281 x 500	
Size (KB)	422KB	22KB	64KB	7KB	6KB	---
Compression Rate	---	94.79%	98,10%	98.34%	98,58%	---

	Original	Ultra Hight	High	Medium	Low	Animation
Images						
Quality	---	100%	75%	50%	25%	100% - 0%
Size (pixeis)	384 x 512	384 x 512	384 x 512	384 x 512	384 x 512	
Size (KB)	577KB	142KB	24KB	16KB	11KB	---
Compression Rate	---	75.39%	95,85%	97,23%	98,09%	---

	Original	Ultra Hight	High	Medium	Low	Animation
Images						
Quality	---	100%	75%	50%	25%	100% - 0%
Size (pixeis)	297 x 400	297 x 400	297 x 400	297 x 400	297 x 400	
Size (KB)	349KB	124KB	28KB	19KB	12KB	---
Compression Rate	---	64.47%	91.98%	94.56%	96.56%	---

A imagem logo.bmp, mesmo com uma qualidade alta (75%), já apresenta uma leve distorção no contorno das figuras e, à medida que a qualidade diminui, esta distorção vai ficando cada vez mais evidente. A razão pela qual podemos verificar ruído com uma taxa de compressão alta, deve-se ao facto de existir um elevado contraste entre as tonalidades das formas e o fundo (só existem três cores com apenas uma tonalidade cada uma).

Na imagem peppers.bmp, apenas começam a ser percetíveis perturbações a partir da qualidade média (50%). A razão para tal provém da imagem conter transições mais suaves em comparação com a imagem logo.bmp.

Por fim, na imagem barn_mountains.bmp, não existe uma discrepância da qualidade visual tão saliente entre todos os níveis de compressão visto que esta tem essencialmente transições suaves. No entanto, é possível distinguir-se algumas zonas pixelizadas quando a qualidade de compressão é baixa.

De modo geral, as imagens foto-realistas, como é o caso da peppers.bmp e da barn_mountains.bmp, conseguem alcançar maiores taxas de compressão sem grande perda percetual. Estes resultados encontram-se conforme o esperado, pois sabemos que o JPEG utiliza métodos de compressão pensados para imagens foto-realistas.

Funções auxiliares usadas ao longo do código

In [3]:
```
def read_image(image_name):
    """
    Reads an image from a file and returns a numpy array of its pixels.

    :param image_name: the name of the image to read
    :return: The image as a numpy array
    """
    image = np.array(plt.imread(image_name))
    return image
```

In [4]:
```
def create_colormap(color_list, name='cmap'):
    """
    Create a colormap from a list of colors

    :param color_list: a list of RGB values
    :param name: The name of the colormap, defaults to cmap (optional)
    :return: A colormap object.
    """
    return clr.LinearSegmentedColormap.from_list(name, color_list, N=256)
```

In [5]:
```
graymap = create_colormap(['black', 'white'], 'blackwhite')
```

In [6]:
```
def plot_image(image, colormap=graymap, title=""):
    """
    Plot an image using matplotlib

    :param image: The image to plot
    :param colormap: The colormap to use for the plot
    :param title: The title of the plot
    """
    plt.figure()
    plt.title(title)
    plt.imshow(image, colormap)
    #plt.axis('off')
    plt.show()
```

In [7]:
```
def plot_image_colorbar(image, title=''):
    """
    Plot a color image using the graymap colormap

    :param image: The image to plot
    :param title: The title of the plot
    """
    plt.figure()
    plt.title(title)
    plt.imshow(image, graymap)
    plt.colorbar()
    plt.show()
```

In [8]:
```
def plot_compared_images(image1, image2, title1, title2):
    """
    Plot two images side by side

    :param image1: The first image to compare
```

```
    :param image2: The image to be compared to
    :param title1: The title of the first image
    :param title2: The title of the second image
    """
    fig = plt.figure(figsize=(10, 7))
    ax1 = fig.add_subplot(121)
    ax1.set_title(title1)
    ax1.imshow(image1, graymap)

    ax2 = fig.add_subplot(122)
    ax2.set_title(title2)
    ax2.imshow(image2, graymap)
    plt.show()
```

In [9]:

```
def get_image_rgb(image):
    """
    Given an image, return a tuple of three numpy arrays, one for each of the red, g
    channels

    :param image: the image to be converted to a numpy array
    :return: A tuple of three numpy arrays, each of which is an image channel.
    """
    return np.array((image[:, :, 0], image[:, :, 1], image[:, :, 2]))
```

In [10]:

```
def get_image_from_channels(channels):
    """
    Given a list of channels,
    return a 3-channel image

    :param channels: A list of the channels to use
    :return: a numpy array of shape (lines, columns, 3)
    """
    lines, columns = channels[0].shape
    img = np.zeros((lines, columns, 3), dtype=np.uint8)
    img[:, :, 0] = channels[0]
    img[:, :, 1] = channels[1]
    img[:, :, 2] = channels[2]
    return img
```

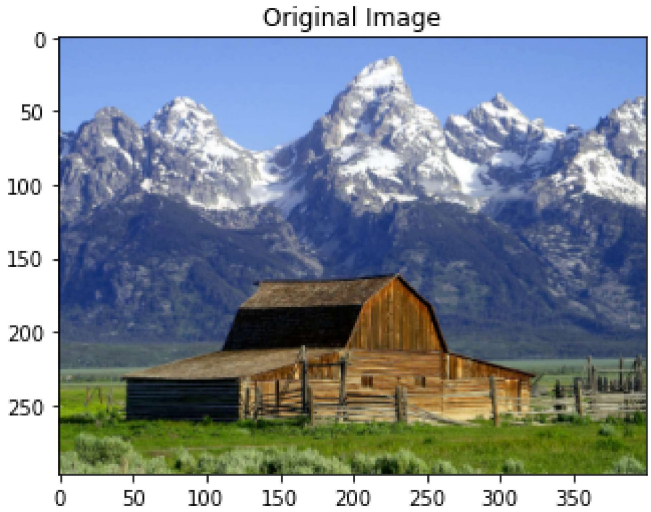
In [11]:

```
red_map = create_colormap(['black', 'red'], 'redmap')
green_map = create_colormap(['black', 'green'], 'greenmap')
blue_map = create_colormap(['black', 'blue'], 'bluemap')
random_map = create_colormap(['blue', 'orange', 'pink'], 'randommap')
```

Imagem a usar ao longo do relatório

In [12]:

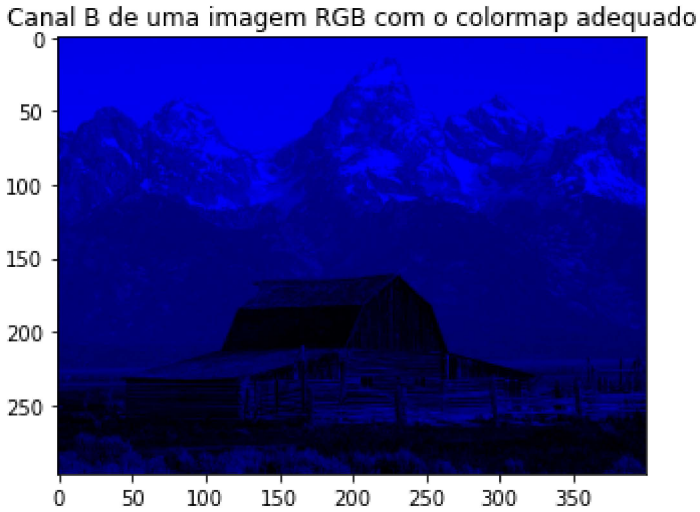
```
image = read_image('./imagens/barn_mountains.bmp')
size = image.shape
plot_image(image, title="Original Image")
```

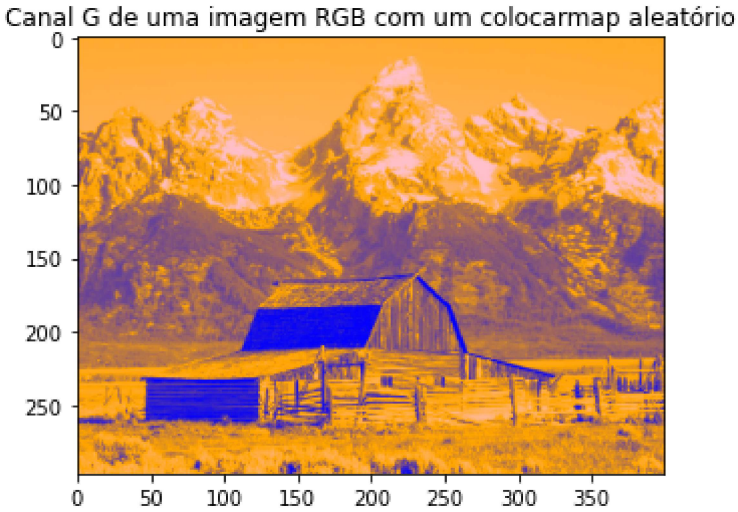


Colormaps

In [13]:

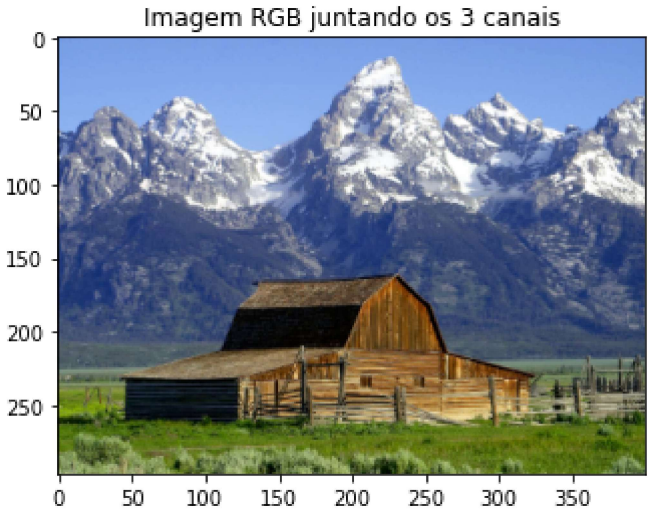
```
img_rgb = get_image_rgb(image)
plot_image(img_rgb[0], red_map, 'Canal R de uma imagem RGB com o colormap adequado')
plot_image(img_rgb[1], green_map, 'Canal G de uma imagem RGB com o colormap adequado
plot_image(img_rgb[2], blue_map, 'Canal B de uma imagem RGB com o colormap adequado'
plot_image(img_rgb[1], random_map, 'Canal G de uma imagem RGB com um colocarmap alea
```





In [14]:

```
plot_image(get_image_from_channels(img_rgb), title='Imagem RGB juntando os 3 canais'
```



Padding

In [15]:

```
def add_padding(image, padding=16):
    """
    Given an image, it adds padding to the image so that the image is a multiple of

    :param image: the image to be processed
    :param padding: , defaults to 16 (optional)
```

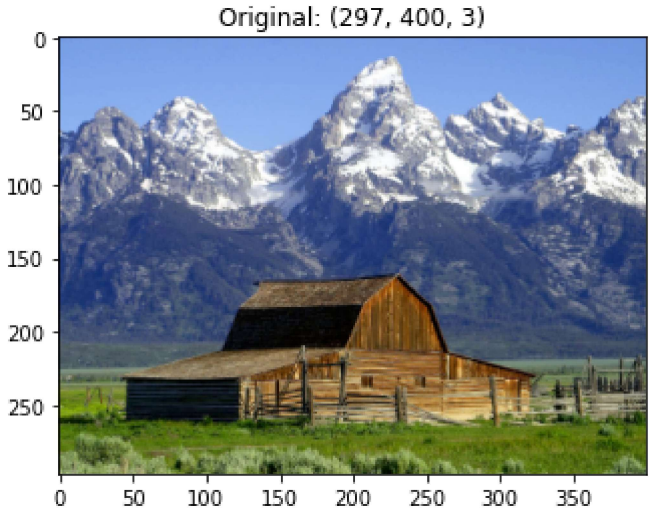
```
    :return: The method returns the image with padding
    """
    rows, columns, _ = image.shape
    red = image[:, :, 0]
    green = image[:, :, 1]
    blue = image[:, :, 2]

    # add rows
    if rows % padding != 0:
        rows_to_add = padding - rows % padding

        aux_red = np.tile(red[-1, :], (rows_to_add, 1))
        aux_green = np.tile(green[-1, :], (rows_to_add, 1))
        aux_blue = np.tile(blue[-1, :], (rows_to_add, 1))

        red = np.vstack([red, aux_red])
        green = np.vstack([green, aux_green])
        blue = np.vstack([blue, aux_blue])

    # add columns
    if columns % padding != 0:
        columns_to_add = padding - columns % padding

        aux_red = np.tile(red[:, -1], (columns_to_add, 1))
        aux_green = np.tile(green[:, -1], (columns_to_add, 1))
        aux_blue = np.tile(blue[:, -1], (columns_to_add, 1))

        red = np.hstack([red, aux_red.T])
        green = np.hstack([green, aux_green.T])
        blue = np.hstack([blue, aux_blue.T])

    return get_image_from_channels((red, green, blue))
```

In [16]:

```
def revert_padding(image, original_rows, original_columns):
    """
    Given an image, revert the padding that was applied to the image

    :param image: The image to be cropped
    :param original_rows: the original number of rows in the image
    :param original_columns: The original width of the image
    :return: The image with the padding removed.
    """
    if(len(image.shape) < 3):
        return image[:original_rows, :original_columns]

    rows, columns, _ = image.shape

    if rows < original_rows or columns < original_columns:
        return image

    return image[:original_rows, :original_columns, :]
```

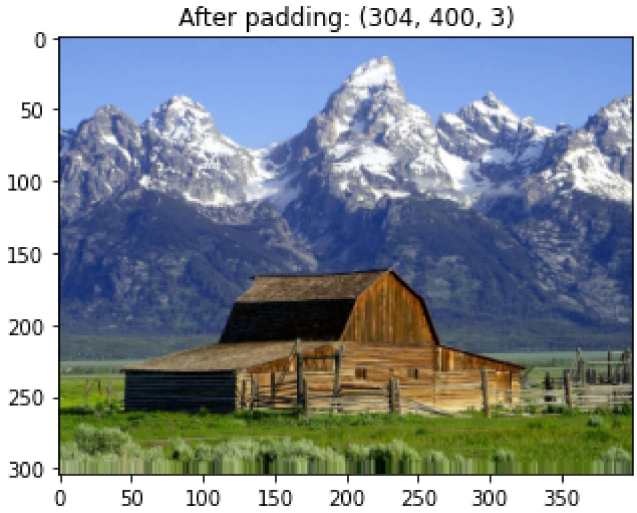
In [17]:

```
plot_image(image, title=f'Original: {size}')
```



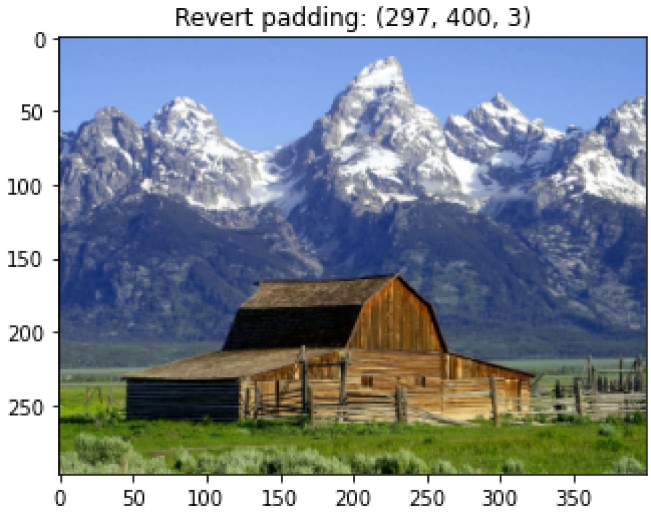
In [18]:

```
image_with_padding = add_padding(image)
plot_image(image_with_padding, title=f'After padding: {image_with_padding.shape}')
```



In [19]:

```
image_reverted_padding = revert_padding(image, size[0], size[1])
plot_image(image_reverted_padding, title=f'Revert padding: {image_reverted_padding.s
```



Modelo YCbCr

In [20]:

```
def convert_rgb_to_ycbcr(image):
```

```
    """
    Convert an RGB image to YCbCr

    :param image: The image to be converted to YCbCr
    :return: a numpy array of the same size as the input image, but with the YCbCr c
    """
    ycbcr_matrix = np.array([
                    [0.299, 0.587, 0.114],
                    [-0.168736, -0.331264, 0.5],
                    [0.5, -0.418688, -0.081312]])

    aux = image.dot(ycbcr_matrix.T)
    aux[:, :, 1:3] += 128
    aux[aux > 255] = 255
    aux[aux < 0] = 0
    aux = aux.round()
    return np.uint8(aux)
```

In [21]:

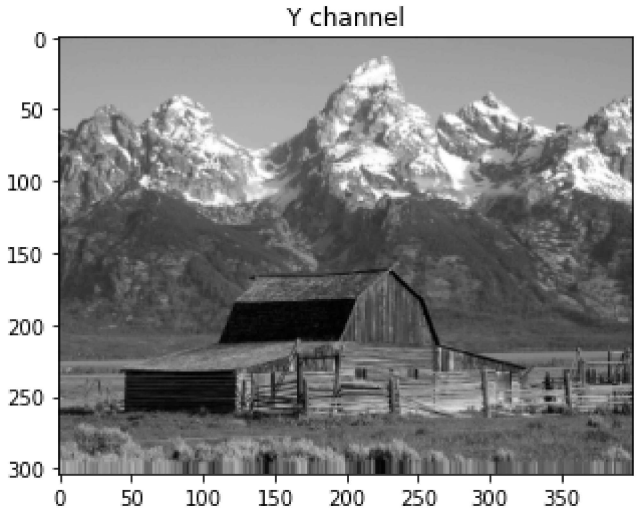
```
def convert_ycbcr_to_rgb(image):
    """
    Convert an image from YCbCr to RGB

    :param image: the image to be converted
    :return: the image converted from YCbCr to RGB.
    """
    image = image.astype(np.float32)
    ycbcr_matrix = np.array([
                    [0.299, 0.587, 0.114],
                    [-0.168736, -0.331264, 0.5],
                    [0.5, -0.418688, -0.081312]])

    inverse = np.linalg.inv(ycbcr_matrix.T)
    aux = np.copy(image)
    aux[:, :, 1:3] -= 128
    aux = aux.dot(inverse)
    aux[aux > 255] = 255
    aux[aux < 0] = 0
    aux = aux.round()
    return np.uint8(aux)
```

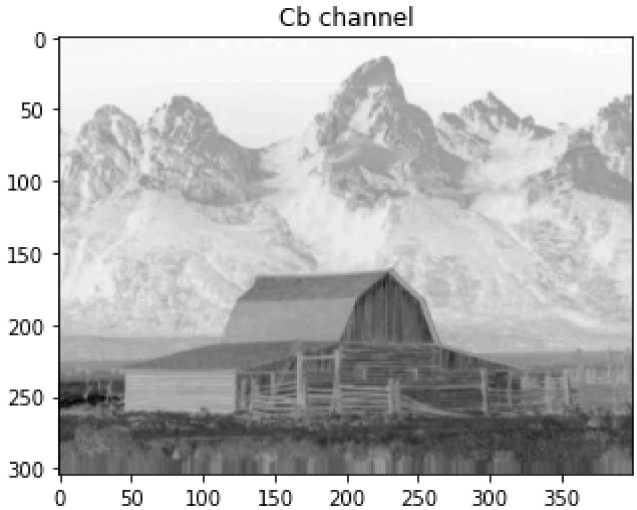
In [22]:

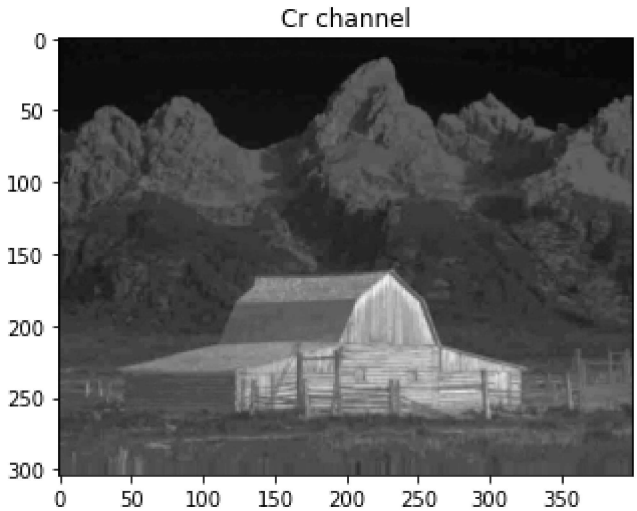
```
image_ycbcr = convert_rgb_to_ycbcr(image_with_padding)
```

In [23]:

```
plot_image(image_ycbcr[:, :, 0], title='Y channel')
plot_image(image_ycbcr[:, :, 1], title='Cb channel')
plot_image(image_ycbcr[:, :, 2], title='Cr channel')
```

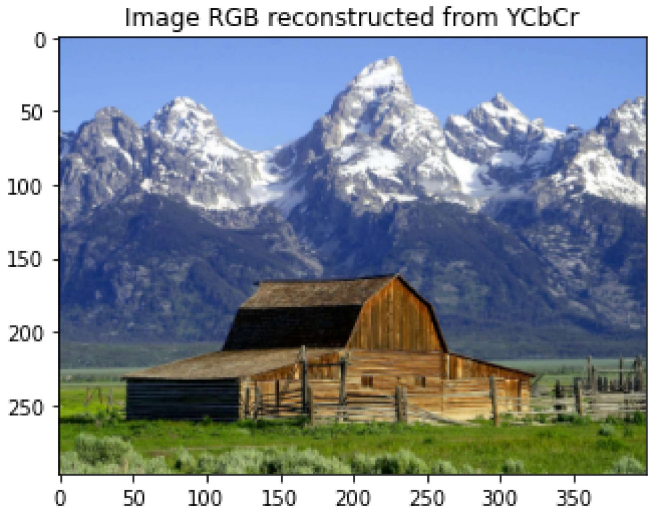






In [24]:

```
image_revert_ycbcr = convert_ycbcr_to_rgb(image_ycbcr)
plot_image(revert_padding(image_revert_ycbcr, size[0], size[1]), title='Image RGB re
```



No modelo RGB, todos os canais contém luminância. Esta redundância é eliminada no modelo YCbCr, guardando a informação sobre a luminância apenas no canal Y. Os restantes canais, Cb e Cr, guardam a informação acerca da variação de azul e vermelho, respetivamente, relativamente à luma (crominância azul e crominância vermelha). Assim sendo, nos canais Cb e Cr é visível um menor detalhe que as torna mais propícias a maiores níveis de compressão.

Sub-amostragem

A função de downsampling foi aplicada apenas aos canais Cb e Cr da imagem, dado que as alterações efetuadas nestes canais não serão tão percetíveis ao olho humano. O canal Y mantém-se inalterado para não perder o detalhe da imagem (luminância).

In [25]:

```
def downsampling(image, ratio, interpolation=False):
    """
    Given an image, it returns the downsampled version of the image

    :param image: The image to be downsampled
    :param ratio: The downsampling ratio
    :param interpolation: If False, use a faster algorithm, otherwise a slower but b
    to False (optional)
    :return: a tuple of three images. The first one is the red channel, the second o
    channel, and the third one is the blue channel.
    """

    ratios = {
        (4, 4, 4): (1, 1),
        (4, 4, 0): (1, 0.5),
        (4, 2, 2): (0.5, 1),
        (4, 2, 0): (0.5, 0.5),
        (4, 1, 1): (0.25, 1),
        (4, 1, 0): (0.25, 0.25)
    }

    scale_x, scale_y = ratios[ratio]

    if scale_x == 1 and scale_y == 1:
        return (image[:, :, 0], image[:, :, 1], image[:, :, 2])

    step_x = int(1//scale_x)
    step_y = int(1//scale_y)

    if interpolation:
        return (image[:, :, 0],
```

```
                cv2.resize(image[:, :, 1], None, fx=scale_x, fy=scale_y, interpolati
                cv2.resize(image[:, :, 2], None, fx=scale_x, fy=scale_y, interpolati
    else:
        return (image[:, :, 0], image[::step_y, ::step_x, 1], image[::step_y, ::step
```

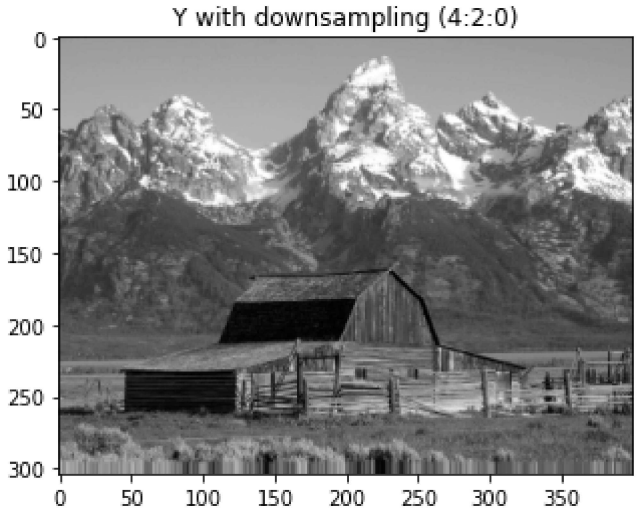
In [26]:

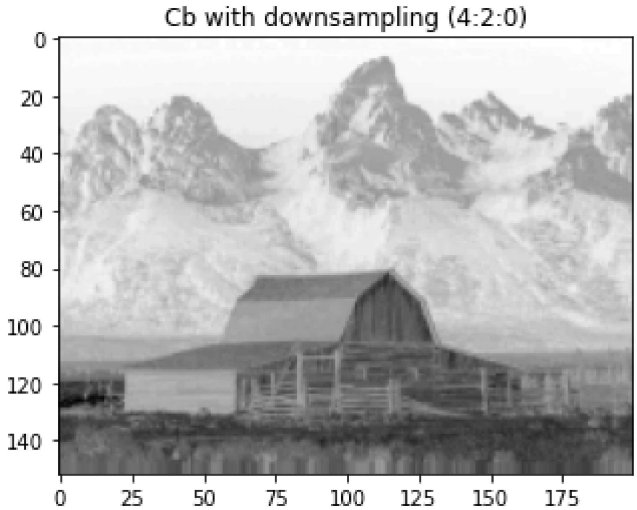
```
def upsampling(y, cb, cr, ratio, interpolation=False):
    """
    Given a image, it will upsample the  channels by the given ratio and return the
    upsampled image

    :param y: The luma component of the image
    :param cb: Chroma Blue (U)
    :param cr: Chrominance component of the image
    :param ratio: The ratio of the input image to the output image
    :param interpolation: If True, uses bilinear interpolation for upsampling. Other
    neighbor, defaults to False (optional)
    :return: the upsampled Y, Cb and Cr channels.
    """
    ratios = {
        (4, 4, 4): (1, 1),
        (4, 4, 0): (1, 0.5),
        (4, 2, 2): (0.5, 1),
        (4, 2, 0): (0.5, 0.5),
        (4, 1, 1): (0.25, 1),
        (4, 1, 0): (0.25, 0.25)
    }

    scale_x, scale_y = ratios[ratio]

    if scale_x == 1 and scale_y == 1:
        return (y, cb, cr)

    step_x = int(1//scale_x)
    step_y = int(1//scale_y)

    if interpolation:
        return (y,
            cv2.resize(cb, None, fx=step_x, fy=step_y, interpolation=cv2.INTER_LINEA
            cv2.resize(cr, None, fx=step_x, fy=step_y, interpolation=cv2.INTER_LINEA
    else:
        upsampled_cb = np.repeat(cb, step_x, axis=1)
        upsampled_cb = np.repeat(upsampled_cb, step_y, axis=0)

        upsampled_cr = np.repeat(cr, step_x, axis=1)
        upsampled_cr = np.repeat(upsampled_cr, step_y, axis=0)

        return (y, upsampled_cb, upsampled_cr)
```

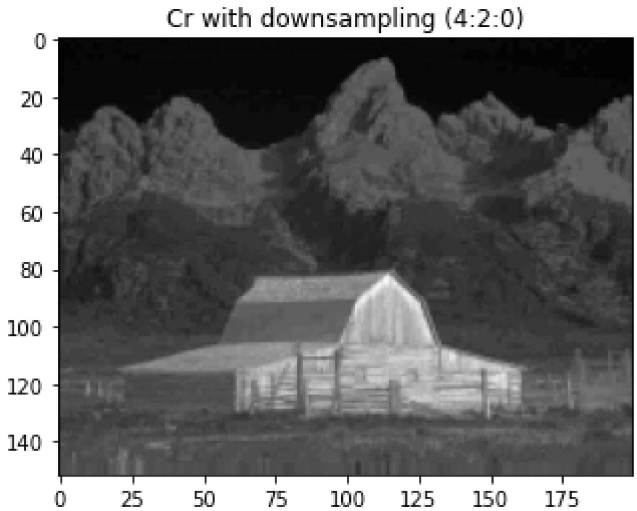
Rácio 4:2:0

In [27]:

```
ratio = (4, 2, 0)
y, cb, cr = downsampling(image_ycbcr, ratio, True)
plot_image(y, title="Y with downsampling (4:2:0)")
plot_image(cb, title="Cb with downsampling (4:2:0)")
plot_image(cr, title="Cr with downsampling (4:2:0)")
```

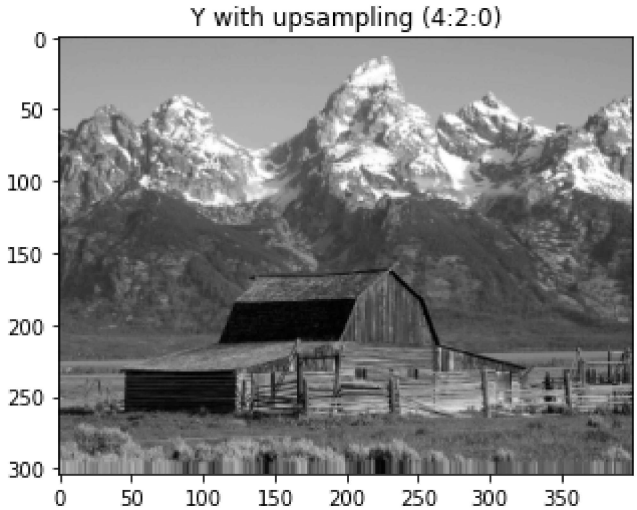


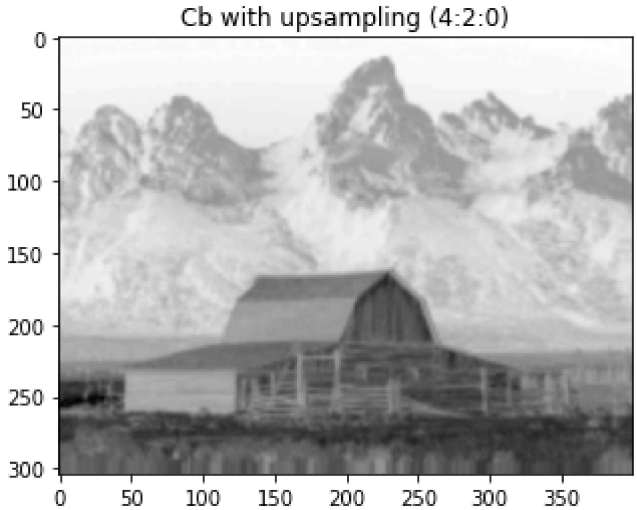


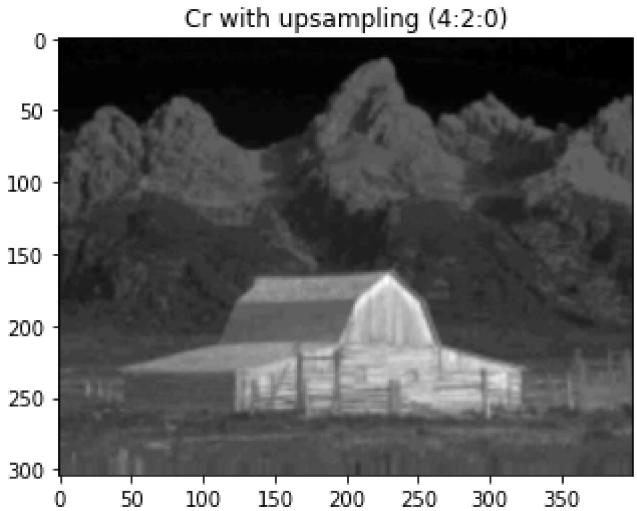


In [28]:

```
y_upsampling, cb_upsampling, cr_upsampling = upsampling(y, cb, cr, ratio, True)
plot_image(y_upsampling, title="Y with upsampling (4:2:0)")
plot_image(cb_upsampling, title="Cb with upsampling (4:2:0)")
plot_image(cr_upsampling, title="Cr with upsampling (4:2:0)")
```





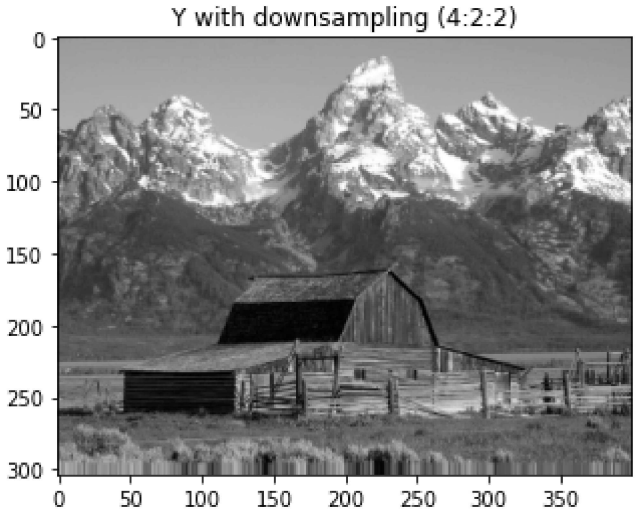


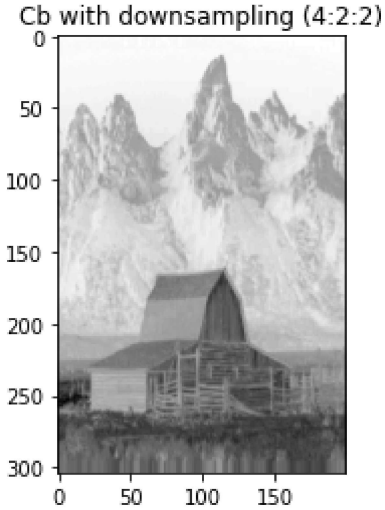
Como se pode observar, ao usar o rácio 4:2:0, ou seja, reduzindo o número de colunas e linhas para metade (passo de 2) dos canais Cb e Cr, conseguimos comprimir estes canais para 25% do seu tamanho original.

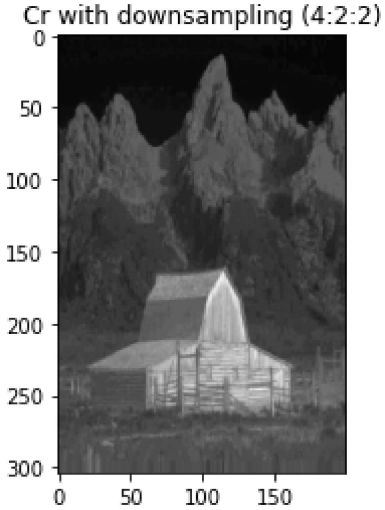
Ao utilizar interpolação quer no downsampling, quer no upsampling, é possível verificar que a imagem final se aproxima mais da original. Enquanto, sem interpolação, a imagem final não fica tão suave, notando-se até alguns "degraus", ou seja, perda de alguma informação, especialmente na imagem logo.bmp dado que apresenta maiores contrastes.

Rácio 4:2:2

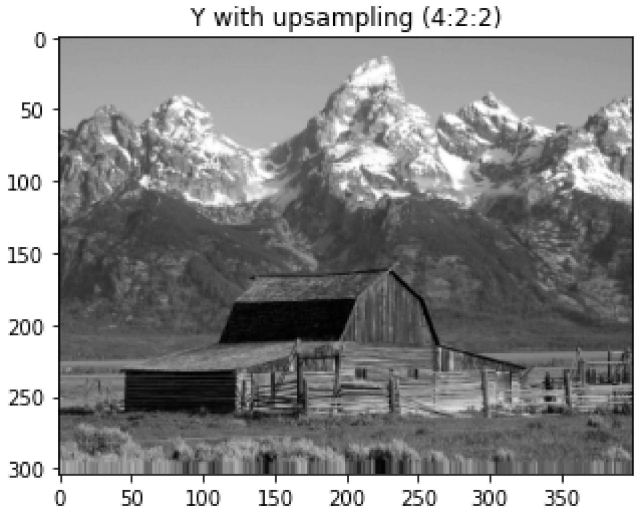
In [29]:
```
ratio = (4,2,2)
y, cb, cr = downsampling(image_ycbcr, ratio, True)
plot_image(y, title="Y with downsampling (4:2:2)")
plot_image(cb, title="Cb with downsampling (4:2:2)")
plot_image(cr, title="Cr with downsampling (4:2:2)")
```

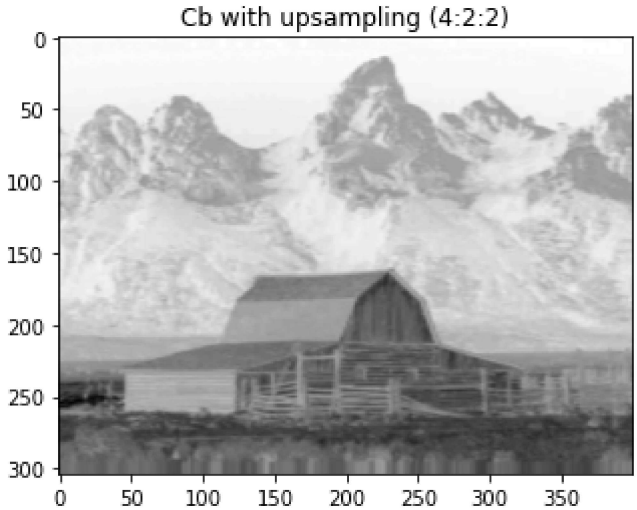


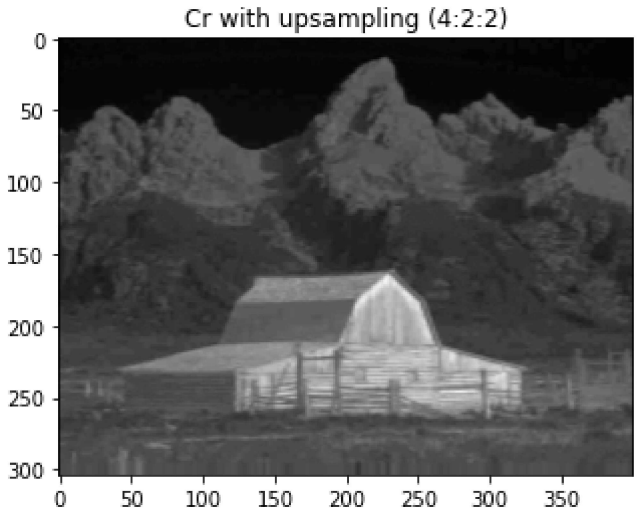




In [30]:
```
y_upsampling, cb_upsampling, cr_upsampling = upsampling(y, cb, cr, ratio, True)
plot_image(y_upsampling, title="Y with upsampling (4:2:2)")
plot_image(cb_upsampling, title="Cb with upsampling (4:2:2)")
plot_image(cr_upsampling, title="Cr with upsampling (4:2:2)")
```







Utilizando um rácio 4:2:2, reduzimos para metade o número de colunas dos canais Cb e Cr. Assim, consegue-se comprimir estes canais para 50% do seu tamanho original.

É ainda possível verificar que a utilização da interpolação tem o mesmo efeito que no rácio anterior.

Nas imagens foto-realistas, a perda de informação das sub-amostragens não é tão notória quando comparada com o mesmo tipo de sub-amostragem em imagens gráficas vetoriais.

Comparando os rácios 4:2:0 e 4:2:2, concluí-se que, como era expetável, o rácio 4:2:0 apresenta piores resultados visualmente que o rácio 4:2:2, quer com interpolação, quer sem interpolação.

Porém, essas diferenças são mínimas, apenas se distinguindo mais quando se faz zoom das imagens.

Por fim, é também de realçar, que, para o mesmo rácio, a imagem, quer com ou sem interpolação, apresenta a mesma taxa de compressão após o downsampling.

Transformada Discreta do Cosseno (DCT)

In [31]:

```
def get_dct(channel):
    """
    Given a channel, return the DCT of the channel

    :param channel: The channel to be transformed
    :return: The dct of the channel.
    """
    return fft.dct(fft.dct(channel, norm="ortho").T, norm="ortho").T
```

In [32]:

```
def get_inverse_dct(channel):
    """
    Given a channel, return the inverse discrete cosine transform of that channel

    :param channel: The channel to be processed
    :return: The inverse dct of the channel.
    """
    return fft.idct(fft.idct(channel, norm="ortho").T, norm="ortho").T
```

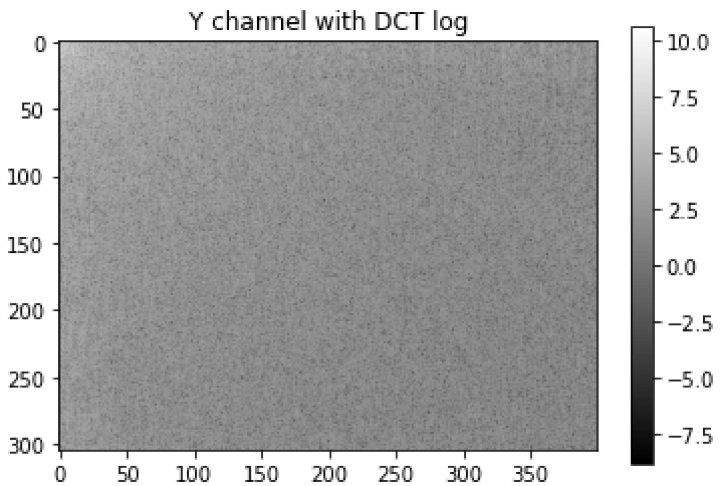
DCT aplicada a toda a imagem

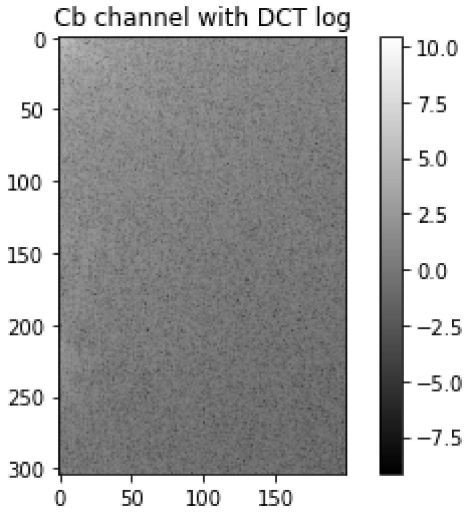
In [33]:

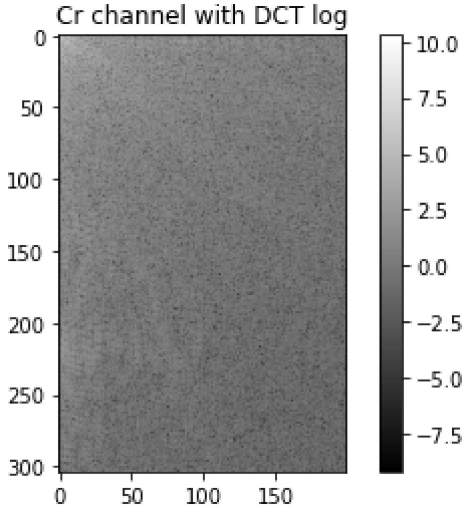
```
y_d = get_dct(y)
cb_d = get_dct(cb)
cr_d = get_dct(cr)
```

In [34]:

```
y_d_log = np.log(np.abs(y_d) + 0.0001)
cb_d_log = np.log(np.abs(cb_d) + 0.0001)
cr_d_log = np.log(np.abs(cr_d) + 0.0001)

plot_image_colorbar(y_d_log, title='Y channel with DCT log')
plot_image_colorbar(cb_d_log, title='Cb channel with DCT log')
plot_image_colorbar(cr_d_log, title='Cr channel with DCT log')
```

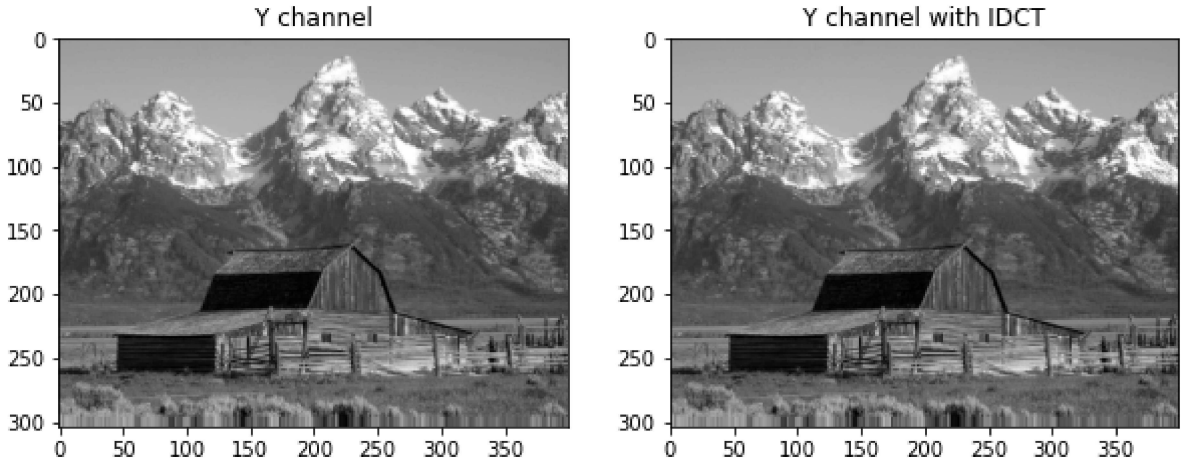


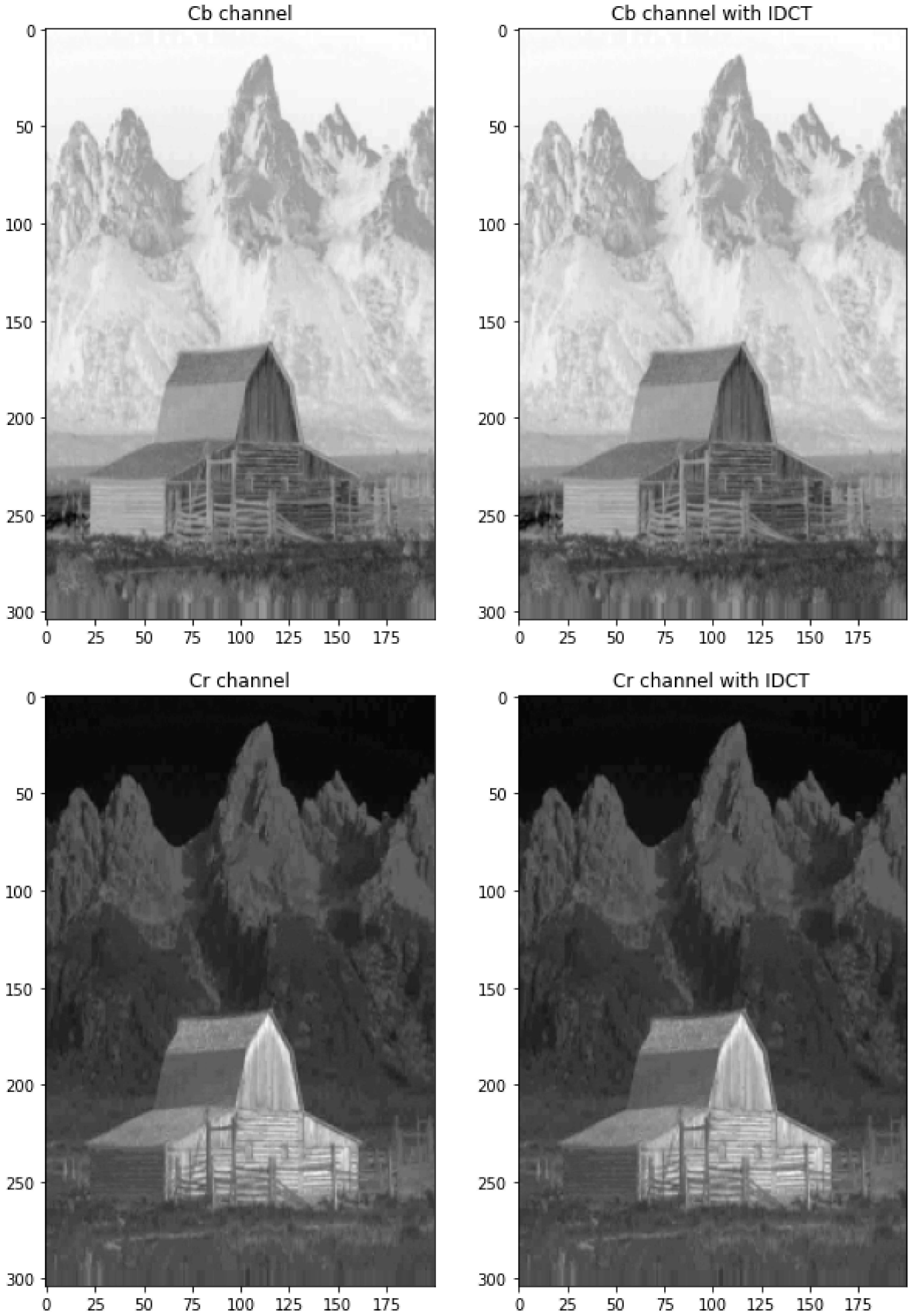




In [35]:

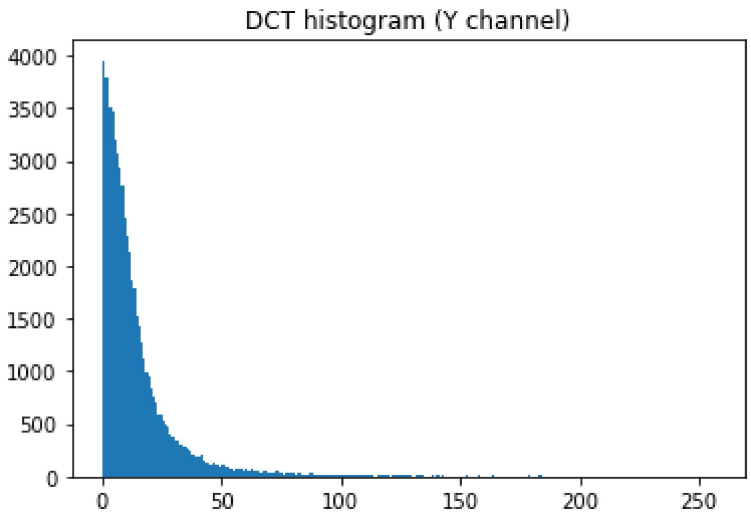
```
y_di = get_inverse_dct(y_d)
cb_di = get_inverse_dct(cb_d)
cr_di = get_inverse_dct(cr_d)

plot_compared_images(y, y_di, 'Y channel', 'Y channel with IDCT')
plot_compared_images(cb, cb_di, 'Cb channel', 'Cb channel with IDCT')
plot_compared_images(cr, cr_di, 'Cr channel', 'Cr channel with IDCT')
```

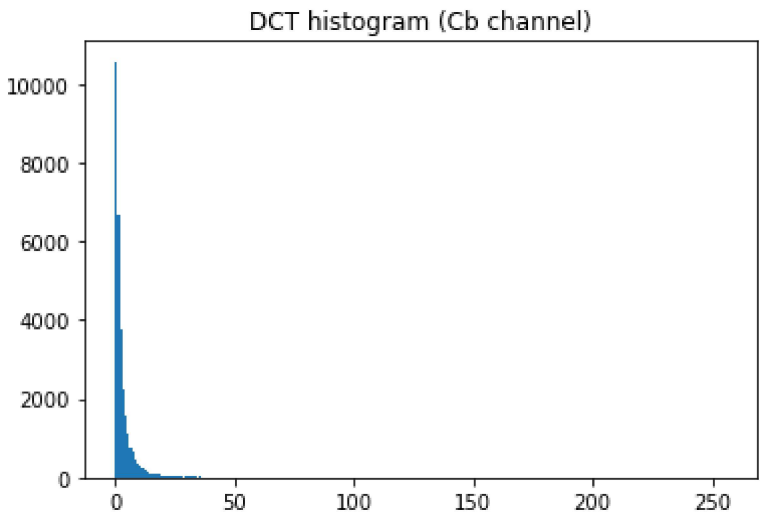


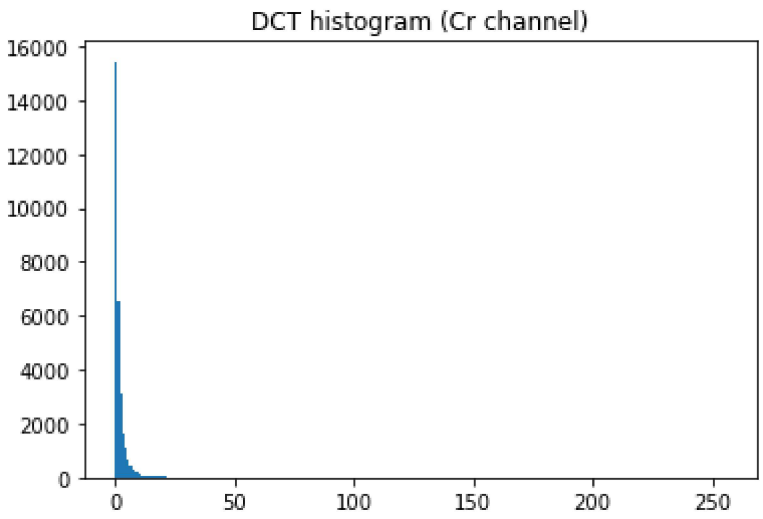


In [36]:

```
plt.hist(y_d.flatten(),256,[0,256])
plt.title("DCT histogram (Y channel)")
plt.show()
plt.hist(cb_d.flatten(),256,[0,256])
plt.title("DCT histogram (Cb channel)")
plt.show()
plt.hist(cr_d.flatten(),256,[0,256])
plt.title("DCT histogram (Cr channel)")
plt.show()
```

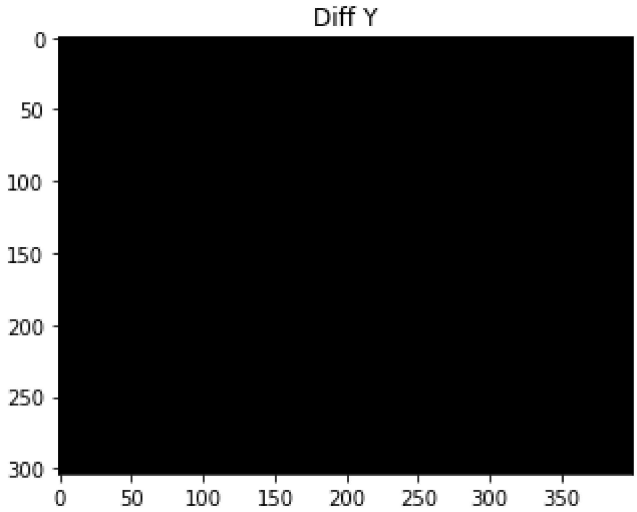


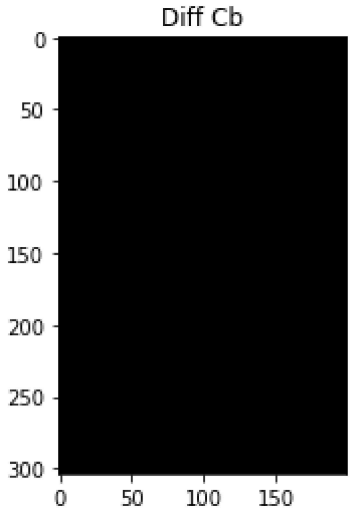


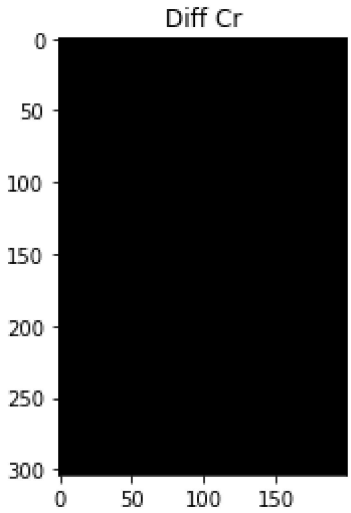


In [37]:

```
diff_image_y = y_di - y
diff_image_y[diff_image_y < 0.000001] = 0
plot_image(diff_image_y, title = 'Diff Y')

diff_image_cb = cb_di - cb
diff_image_cb[diff_image_cb < 0.000001] = 0
plot_image(diff_image_cb, title = 'Diff Cb' )

diff_image_cr = cr_di - cr
diff_image_cr[diff_image_cr < 0.000001] = 0
plot_image(diff_image_cr, title = 'Diff Cr')
```







Ao utilizar a DCT, há uma compactação da energia num número menor de coeficientes. Por este motivo, poderá ser usado um algoritmo de compressão entrópica (RLE, por exemplo) para eliminar a redundância que existe nas altas frequências. Para aumentar, ainda mais, a redundância nestas frequências, poderá ser usado um processo de quantização para aproximar os valores (este processo vai levar à perda de informação).

DCT aplicada em blocos

In [38]:

```
def dct_block(channel, bs):
    """
    This function takes a channel and a block size and returns a dct of the channel
```

```
    :param channel: the channel of the image we want to compress
    :param bs: block size
    :return: The DCT coefficients of the image.
    """
    size = channel.shape
    dct = np.zeros(size)
    for i in np.r_[:size[0]:bs]:
        for j in np.r_[:size[1]:bs]:
            dct[i:(i+bs),j:(j+bs)] = get_dct(channel[i:(i+bs),j:(j+bs)])
    return dct
```

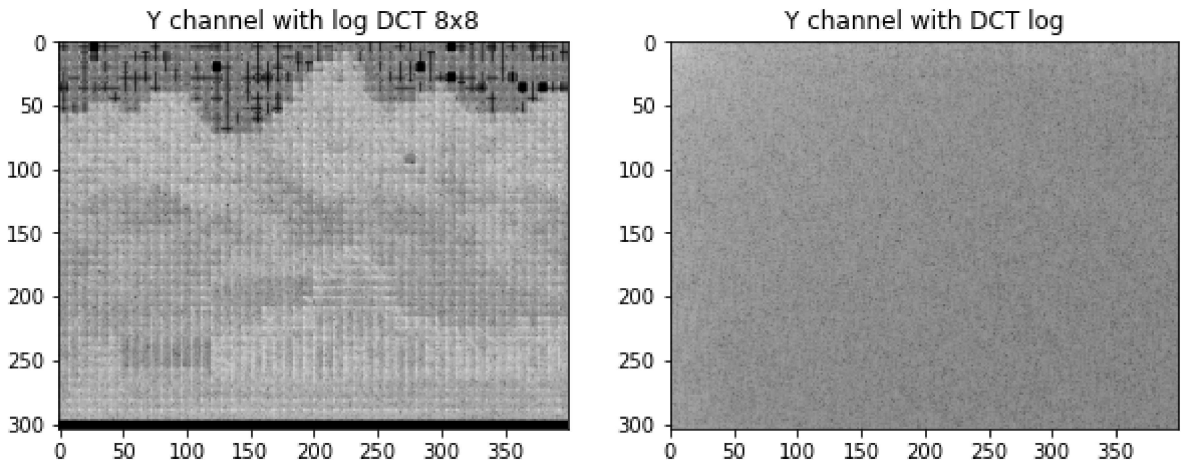
In [39]:

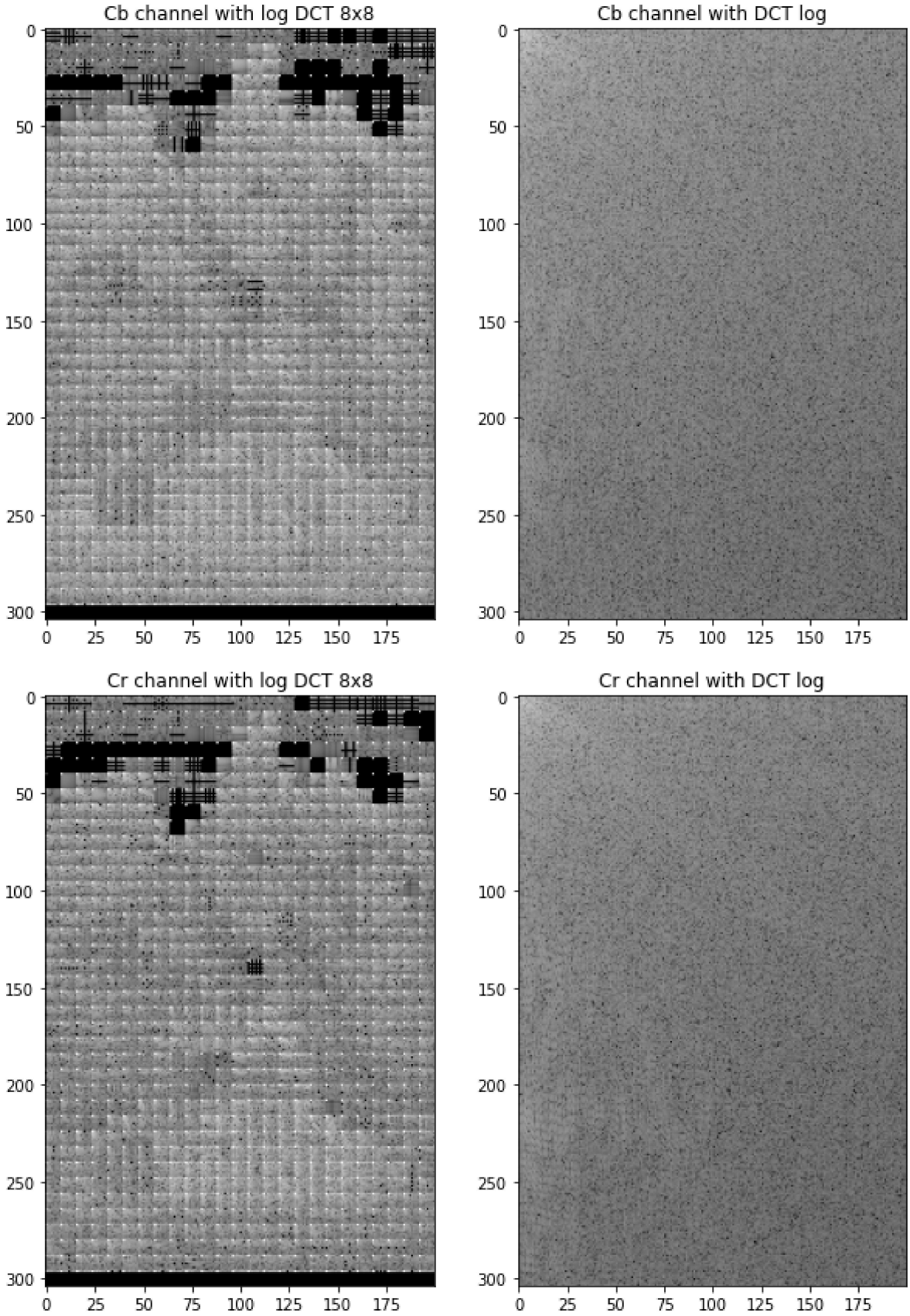
```
def idct_block(channel, bs):
    """
    This function performs the inverse discrete cosine transform on a block of the i

    :param channel: the channel of the image
    :param bs: block size
    :return: The inverse discrete cosine transform of the block.
    """
    size = channel.shape
    idct = np.zeros(size)
    for i in np.r_[:size[0]:bs]:
        for j in np.r_[:size[1]:bs]:
            idct[i:(i+bs),j:(j+bs)] = get_inverse_dct(channel[i:(i+bs),j:(j+bs)])
    idct[idct < 0] = 0
    idct[idct > 255] = 255
    return idct
```

Blocos 8x8

In [40]:

```
y_d_block_8 = dct_block(y, 8)
cb_d_block_8 = dct_block(cb, 8)
cr_d_block_8 = dct_block(cr, 8)

y_d_block_8_log = np.log(np.abs(y_d_block_8) + 0.0001)
cb_d_block_8_log = np.log(np.abs(cb_d_block_8) + 0.0001)
cr_d_block_8_log = np.log(np.abs(cr_d_block_8) + 0.0001)
plot_compared_images(y_d_block_8_log, y_d_log, "Y channel with log DCT 8x8", "Y chan
plot_compared_images(cb_d_block_8_log, cb_d_log, "Cb channel with log DCT 8x8", "Cb
plot_compared_images(cr_d_block_8_log, cr_d_log, "Cr channel with log DCT 8x8", "Cr
```





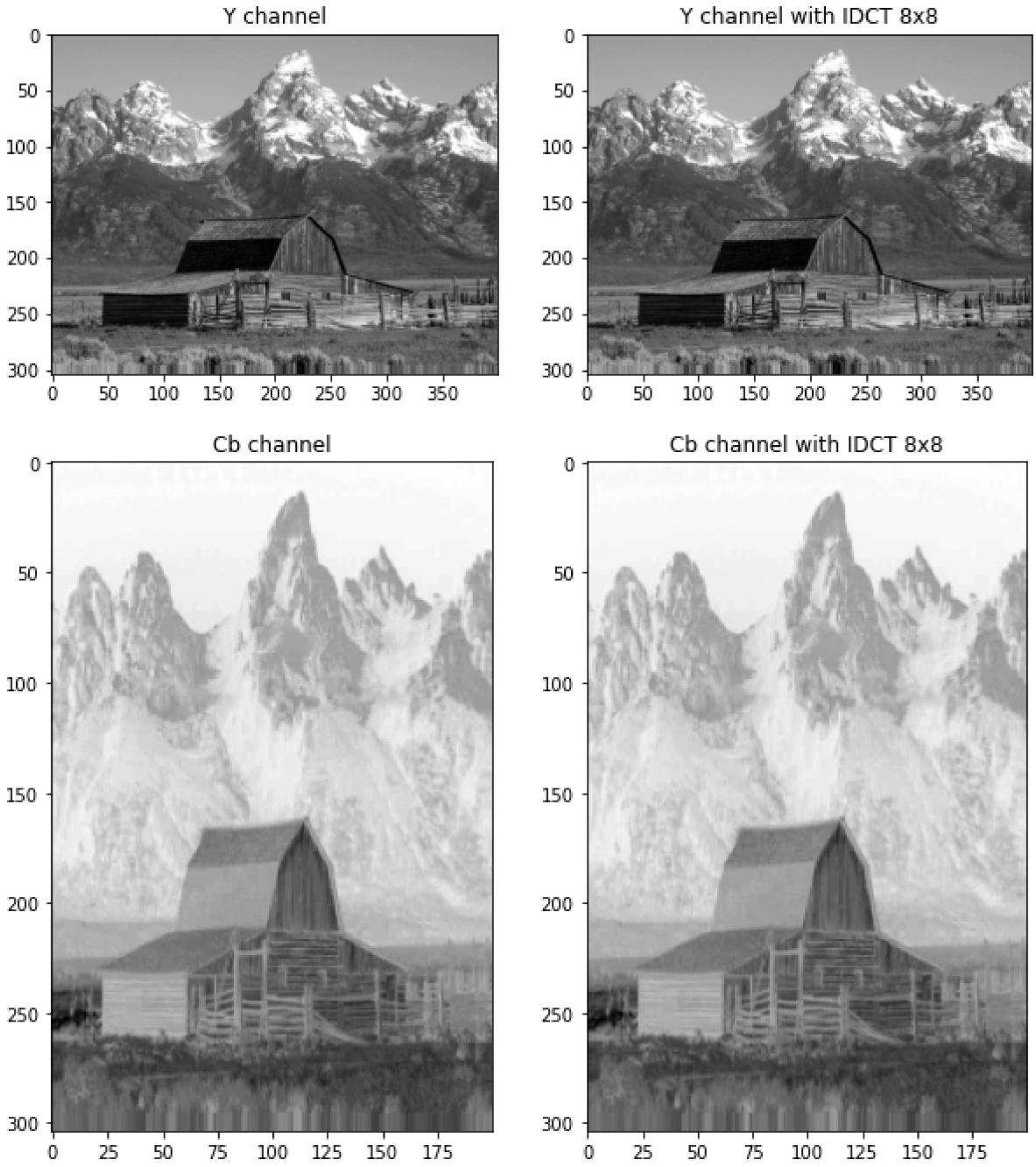
In [41]:

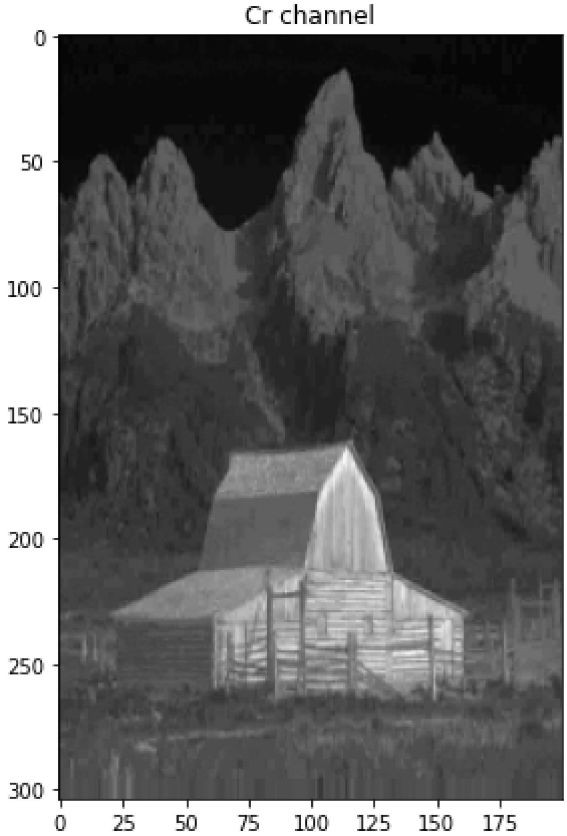
```
y_block_8_di = idct_block(y_d_block_8, 8)
cb_block_8_di = idct_block(cb_d_block_8, 8)
cr_block_8_di = idct_block(cr_d_block_8, 8)
```

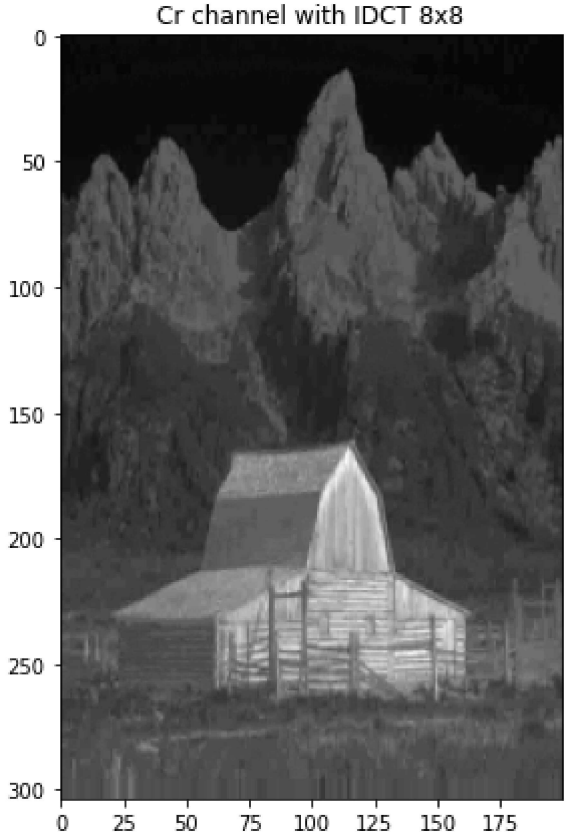
In [42]:

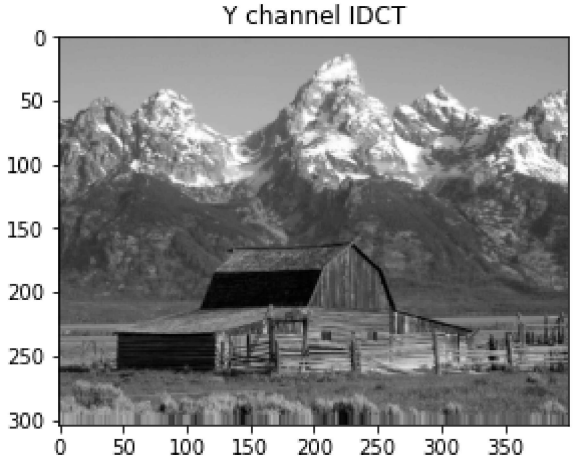
```
plot_compared_images(y, y_block_8_di, 'Y channel', 'Y channel with IDCT 8x8')
plot_compared_images(cb, cb_block_8_di, 'Cb channel', 'Cb channel with IDCT 8x8')
plot_compared_images(cr, cr_block_8_di,'Cr channel', 'Cr channel with IDCT 8x8')
```

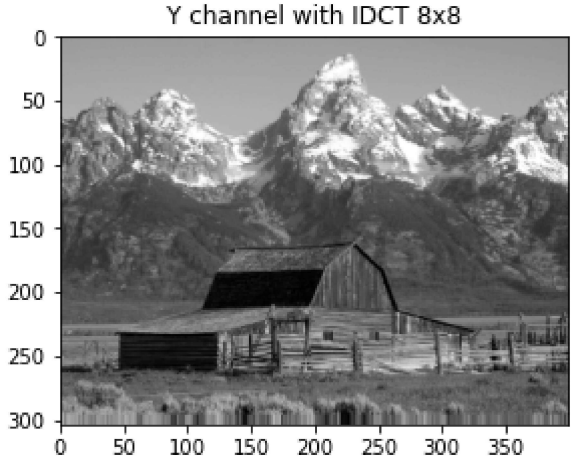
```
plot_compared_images(y_di, y_block_8_di, 'Y channel IDCT', 'Y channel with IDCT 8x8'
plot_compared_images(cb_di, cb_block_8_di, 'Cb channel IDCT', 'Cb channel with IDCT
plot_compared_images(cr_di, cr_block_8_di, 'Cr channel IDCT', 'Cr channel with IDCT
```

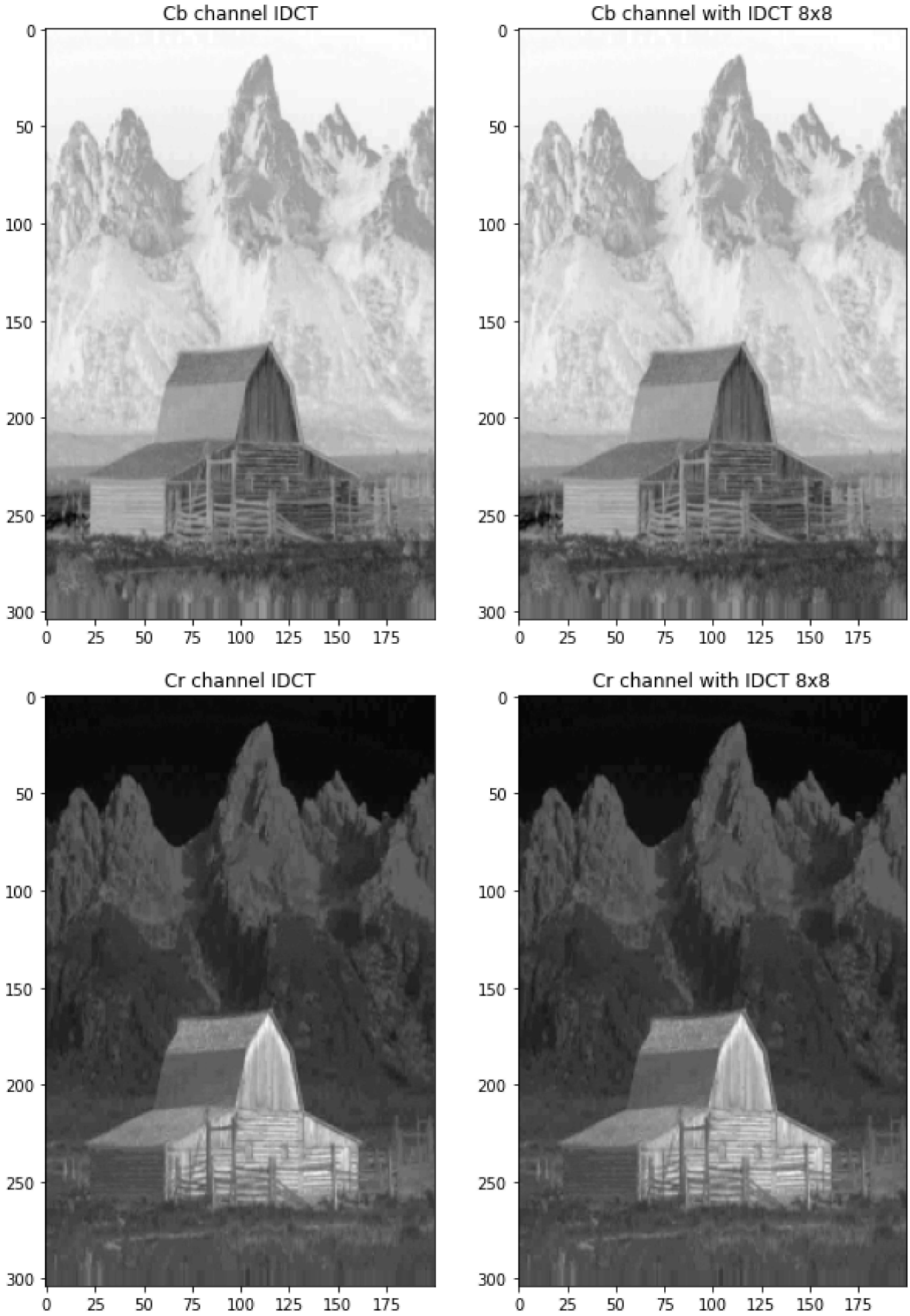



Cr channel
0
50
100
150
200
250
300
0 25 50 75 100 125 150 175


Cr channel with IDCT 8x8
0
50
100
150
200
250
300
0 25 50 75 100 125 150 175


Y channel IDCT
0
50
100
150
200
250
300
0 50 100 150 200 250 300 350


Y channel with IDCT 8x8
0
50
100
150
200
250
300
0 50 100 150 200 250 300 350

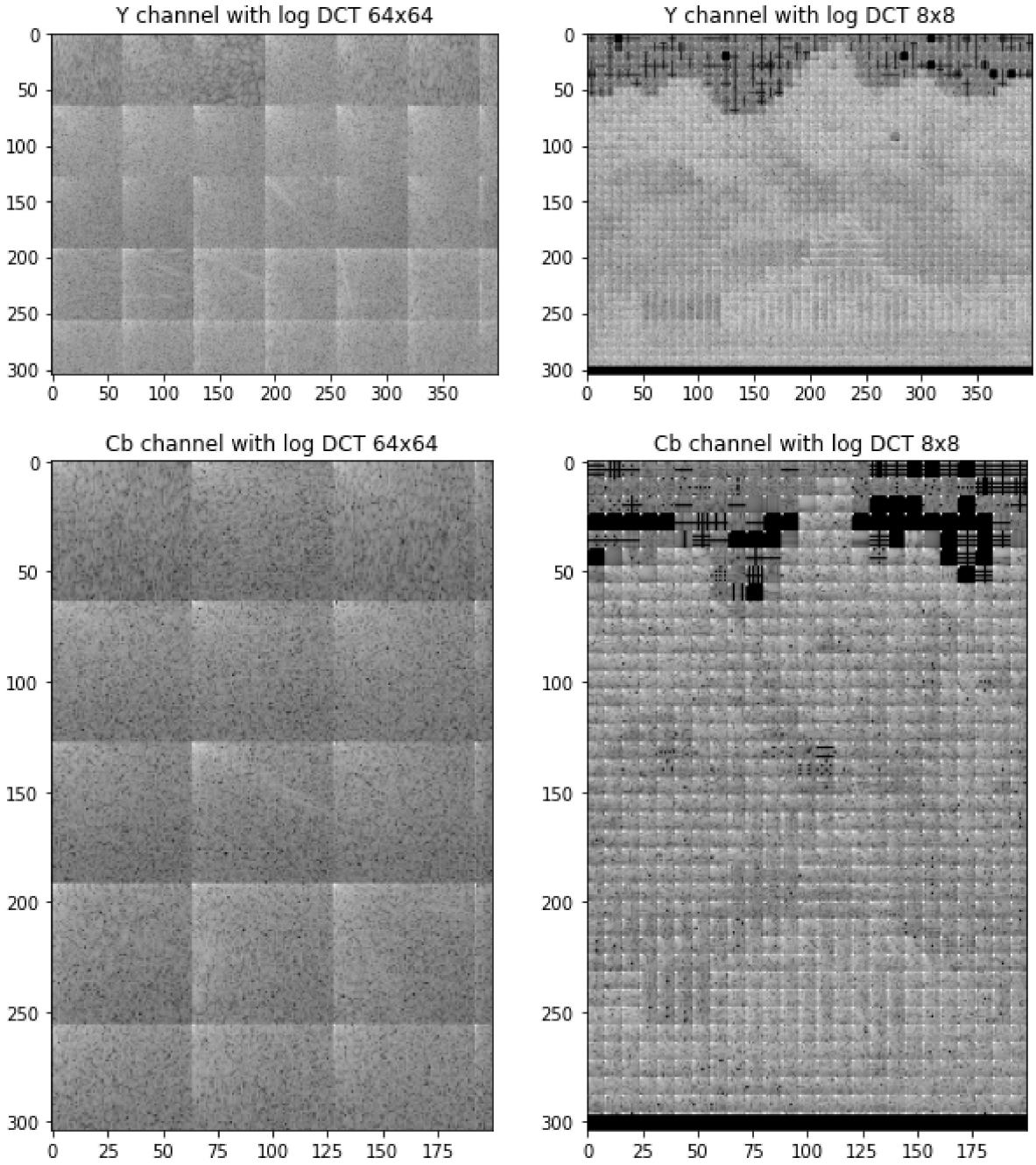


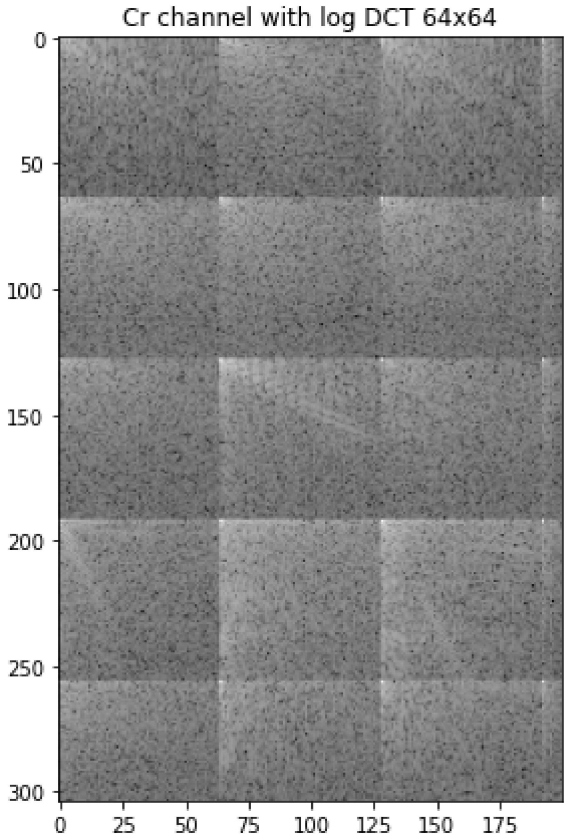
Em blocos 8x8, existe uma menor probabilidade de encontrar transições abruptas. Deste modo, ao aplicar a DCT a estes blocos, os valores AC tendem a aproximar-se de zero enquanto o valor DC se distingue mais desses devido à ausência de frequências altas. Como os valores AC são semelhantes e próximos de zero, consegue-se comprimir mais a imagem através de métodos entrópicos que tiram partido deste aspeto.

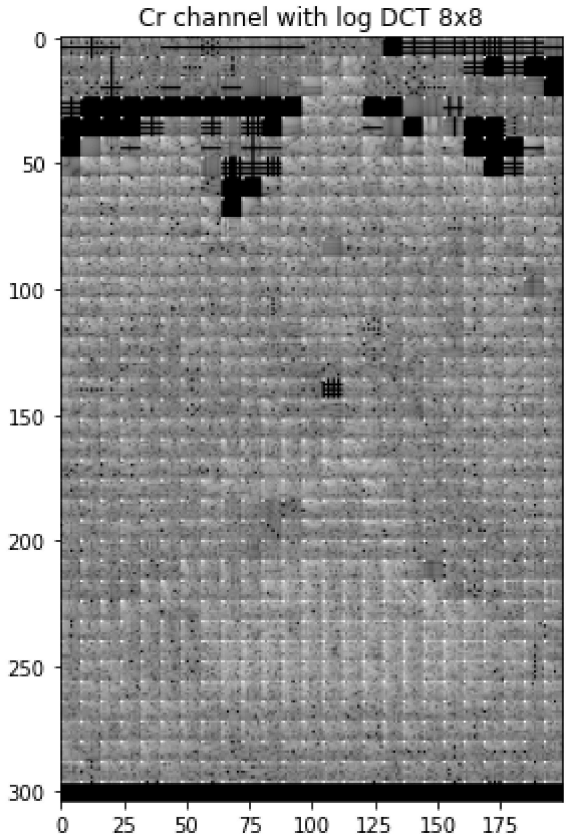
Blocos 64x64

In [43]:

```
y_d_block_64 = dct_block(y, 64)
cb_d_block_64 = dct_block(cb, 64)
cr_d_block_64 = dct_block(cr, 64)

y_d_block_64_log = np.log(np.abs(y_d_block_64) + 0.0001)
cb_d_block_64_log = np.log(np.abs(cb_d_block_64) + 0.0001)
cr_d_block_64_log = np.log(np.abs(cr_d_block_64) + 0.0001)
plot_compared_images(y_d_block_64_log, y_d_block_8_log, "Y channel with log DCT 64x6
plot_compared_images(cb_d_block_64_log, cb_d_block_8_log, "Cb channel with log DCT 6
plot_compared_images(cr_d_block_64_log, cr_d_block_8_log, "Cr channel with log DCT 6
```





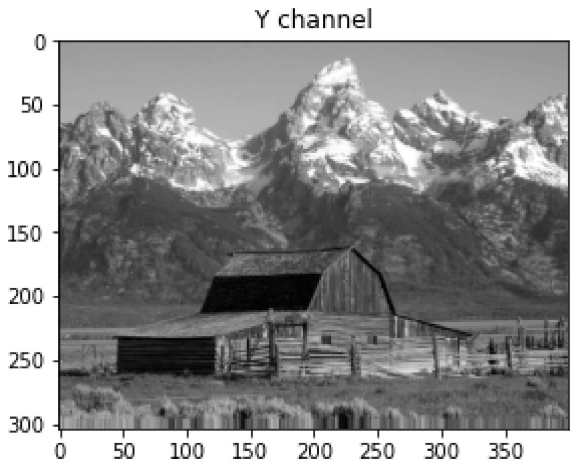


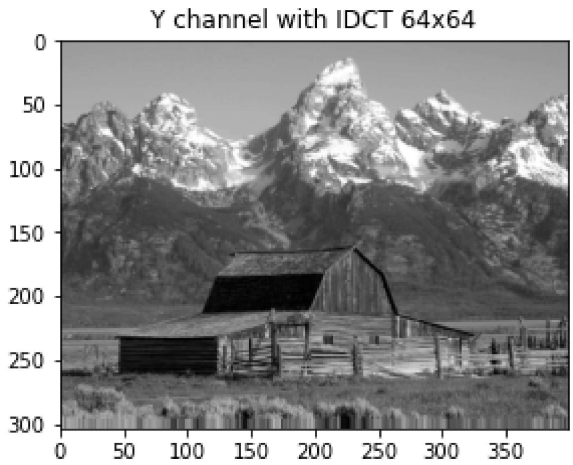
In [44]:

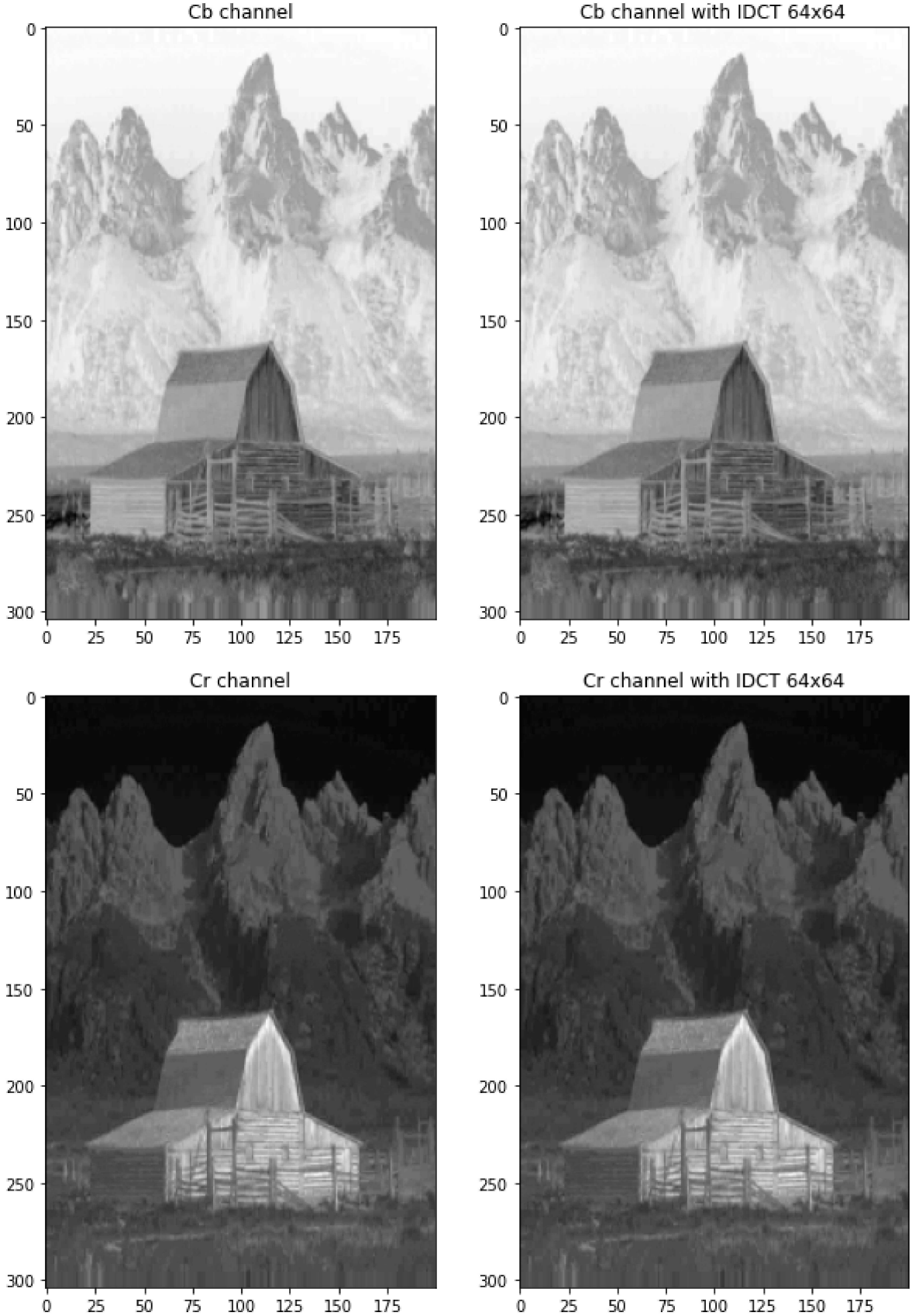
```
y_block_64_di = idct_block(y_d_block_64, 64)
cb_block_64_di = idct_block(cb_d_block_64, 64)
cr_block_64_di = idct_block(cr_d_block_64, 64)
```

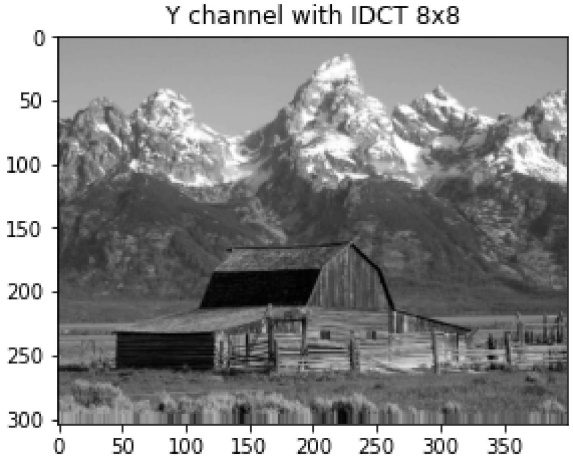
In [45]:

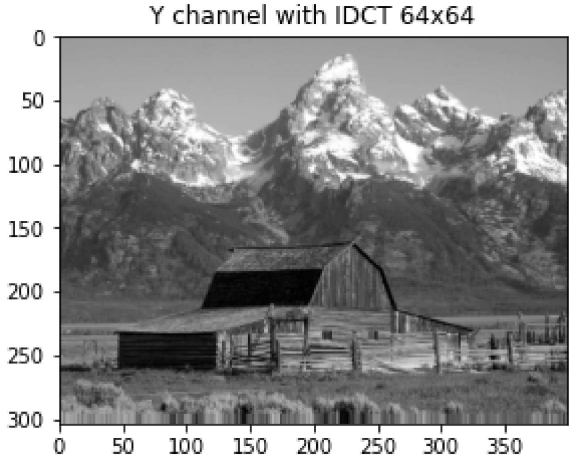
```
plot_compared_images(y, y_block_64_di, 'Y channel', 'Y channel with IDCT 64x64')
plot_compared_images(cb, cb_block_64_di, 'Cb channel', 'Cb channel with IDCT 64x64')
plot_compared_images(cr, cr_block_64_di,'Cr channel', 'Cr channel with IDCT 64x64')

plot_compared_images(y_block_8_di, y_block_64_di, 'Y channel with IDCT 8x8', 'Y chan
plot_compared_images(cb_block_8_di, cb_block_64_di, 'Cb channel with IDCT 8x8', 'Cb
plot_compared_images(cr_block_8_di, cr_block_64_di,'Cr channel with IDCT 8x8', 'Cr c
```

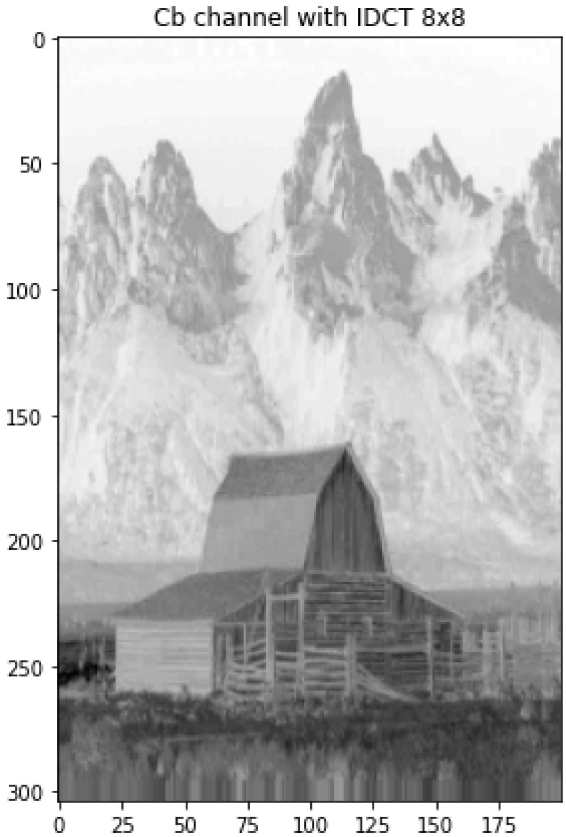


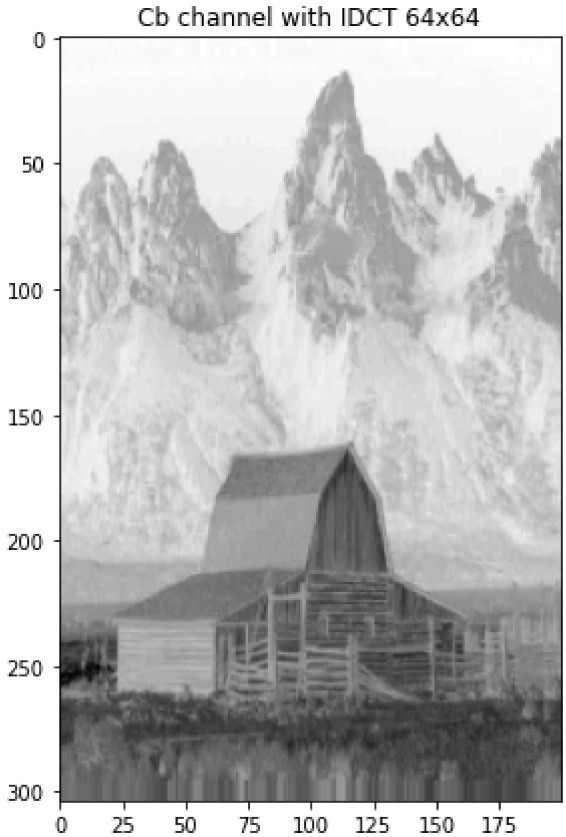


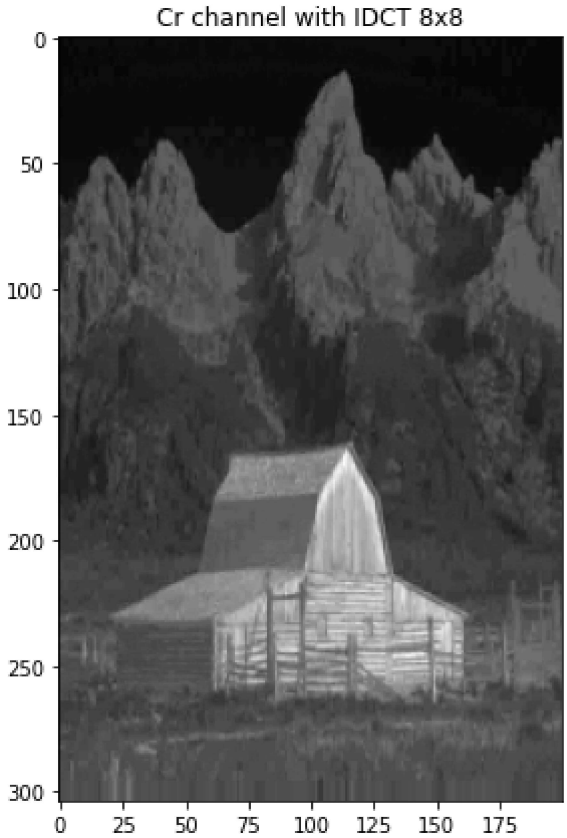

Cb channel
Cb channel with IDCT 64x64
Cr channel
Cr channel with IDCT 64x64
0
50
100
150
200
250
300
0
25
50
75
100
125
150
175

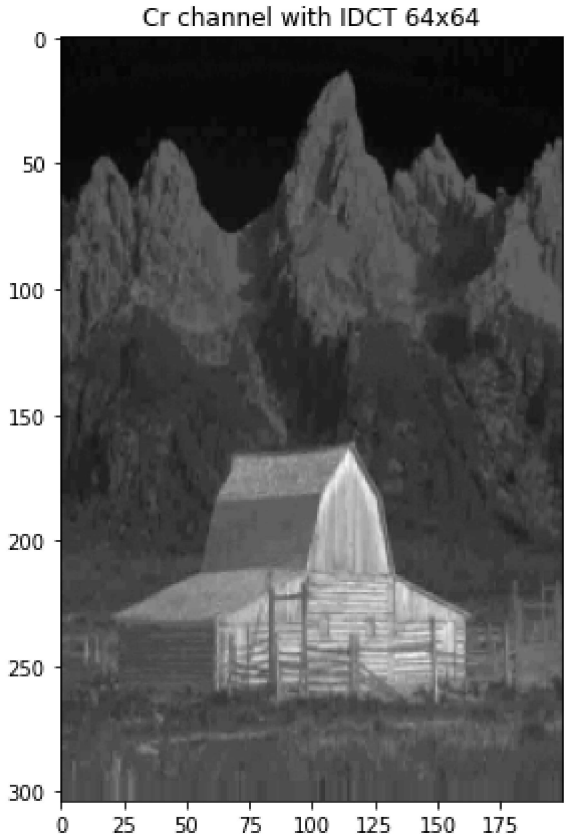

Y channel with IDCT 8x8
0
50
100
150
200
250
300
0
50
100
150
200
250
300
350


Y channel with IDCT 64x64
0
50
100
150
200
250
300
0
50
100
150
200
250
300
350


Cb channel with IDCT 8x8
0
50
100
150
200
250
300
0
25
50
75
100
125
150
175


Cb channel with IDCT 64x64
0
50
100
150
200
250
300
0
25
50
75
100
125
150
175





Aumentando o tamanho dos blocos para 64x64, verifica-se que há mais probabilidade de apanhar transições abruptas na imagem do que em blocos 8x8. Assim, haverá mais discrepâncias entre os coeficientes AC dos blocos e, consequentemente, serão mais difíceis de comprimir com modelos entrópicos.

Quantização

In [46]:

```
def apply_quantization_block(channel, factor):
    """
    Given a channel, apply quantization by dividing each 8x8 block by a factor and r

    :param channel: the channel to be quantized
    :param factor: the quantization factor
    :return: The quantized image.
    """
    size = channel.shape
    quant = np.zeros(size, dtype=np.float32)
    for i in np.r_[:size[0]:8]:
        for j in np.r_[:size[1]:8]:
            quant[i:(i+8),j:(j+8)] = np.round(channel[i:(i+8),j:(j+8)] / factor)
    return quant
```

In [47]:

```
def apply_quantization_block_inverse(channel, factor):
    """
    Given a channel of the image, apply the inverse quantization block

    :param channel: the channel to be quantized
    :param factor: The quantization factor
    :return: The inverse quantization of the channel
    """
    size = channel.shape
    inverse_quant = np.zeros(size, dtype=np.float32)
    for i in np.r_[:size[0]:8]:
```

```
        for j in np.r_[:size[1]:8]:
            inverse_quant[i:(i+8),j:(j+8)] = channel[i:(i+8),j:(j+8)] * factor
    return inverse_quant
```

In [48]:

```
def calculate_quantization_factor(quality):
    """
    Given a quality factor, the function returns the quantization matrices for Y and

    :param quality: The image quality, on a scale from 1 (worst) to 95 (best)
    :return: a tuple of two matrices.
    """
    if quality > 100:
        quality = 100
    if quality < 0:
        quality = 1
    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]])
    q_ones = np.ones((8, 8))
    scaling_factor = 0
    if quality >= 50:
        scaling_factor = (100 - quality) / 50
    else:
        scaling_factor = 50 / quality

    qy_factor = q_ones
    qc_factor = q_ones

    if scaling_factor != 0:
        qy_factor = np.round(qy * scaling_factor)
        qc_factor = np.round(qc * scaling_factor)

    qy_factor[qy_factor > 255] = 255
    qc_factor[qc_factor > 255] = 255
    qy_factor[qy_factor < 1] = 1
    qc_factor[qc_factor < 1] = 1

    return (qy_factor, qc_factor)
```

In [49]:

```
def quantization(y, cb, cr, quality=75):
    """
    Given the quantization factor, apply the quantization to the given channels

    :param y: The y channel of the image
    :param cb: The cb channel of the image
    :param cr: The cr channel of the image
    :param quality: a value between 1 and 100, defaults to 75 (optional)
    :return: The quantized y, cb, and cr values.
```

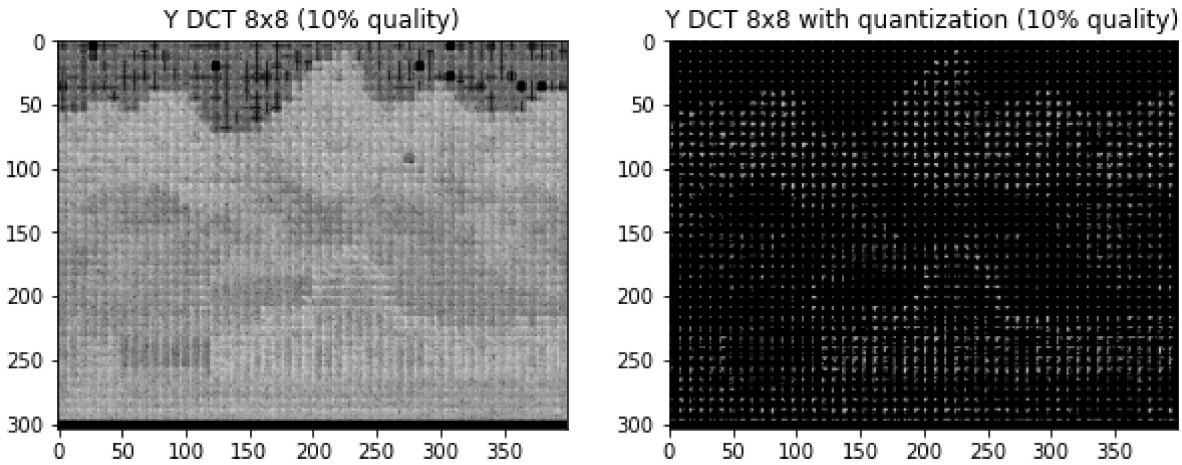
```
    """
    qy_factor, qc_factor = calculate_quantization_factor(quality)
    return (apply_quantization_block(y, qy_factor), apply_quantization_block(cb, qc_
```

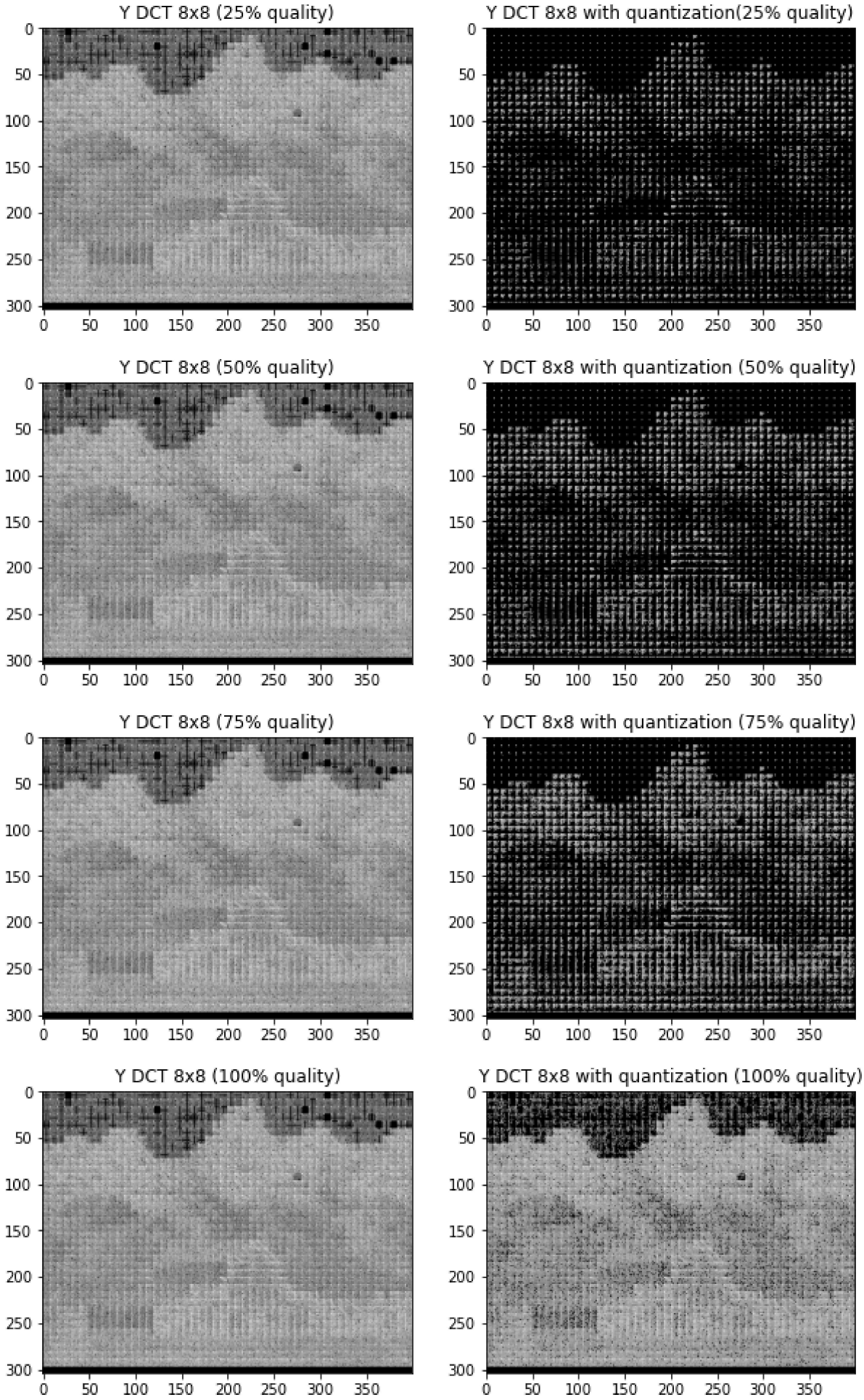
In [50]:

```
def inverse_quantization(y, cb, cr, quality=75):
    """
    Given the quantization factor, apply the inverse quantization to the given chann

    :param y: The y channel of the image
    :param cb: The cb channel of the image
    :param cr: The cr channel of the image
    :param quality: a value between 1 and 100, defaults to 75 (optional)
    :return: the inverse quantization of y, cb and cr channels
    """
    qy_factor, qc_factor = calculate_quantization_factor(quality)
    return (apply_quantization_block_inverse(y, qy_factor), apply_quantization_block
```

In [51]:

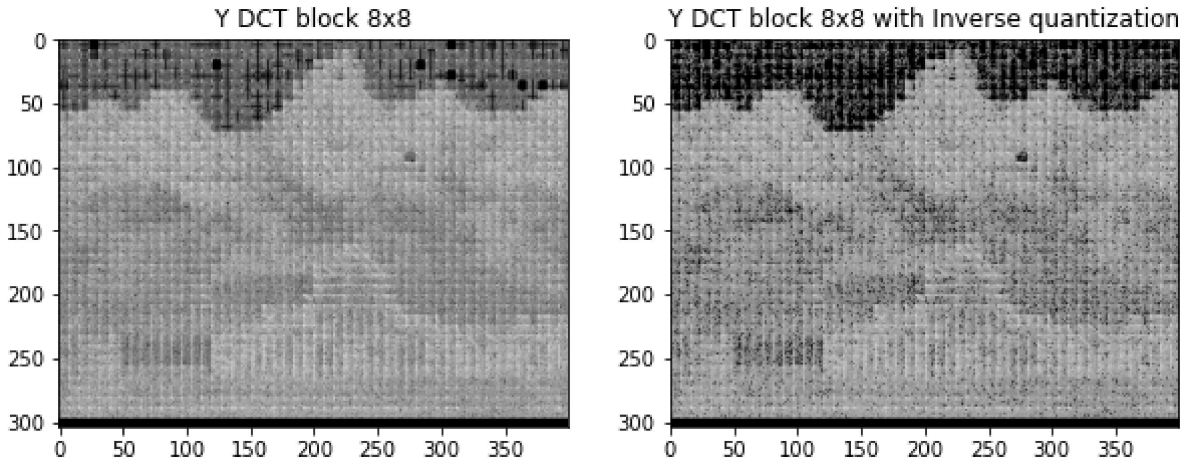
```
quality = 10
y_quant, cb_quant, cr_quant = quantization(y_d_block_8, cb_d_block_8, cr_d_block_8,
plot_compared_images(np.log(np.abs(y_d_block_8) + 0.001),np.log(np.abs(y_quant) + 0.

quality = 25
y_quant, cb_quant, cr_quant = quantization(y_d_block_8, cb_d_block_8, cr_d_block_8,
plot_compared_images(np.log(np.abs(y_d_block_8) + 0.001),np.log(np.abs(y_quant) + 0.

quality = 50
y_quant, cb_quant, cr_quant = quantization(y_d_block_8, cb_d_block_8, cr_d_block_8,
plot_compared_images(np.log(np.abs(y_d_block_8) + 0.001),np.log(np.abs(y_quant) + 0.

quality = 75
y_quant, cb_quant, cr_quant = quantization(y_d_block_8, cb_d_block_8, cr_d_block_8,
plot_compared_images(np.log(np.abs(y_d_block_8) + 0.001),np.log(np.abs(y_quant) + 0.

quality = 100
y_quant, cb_quant, cr_quant = quantization(y_d_block_8, cb_d_block_8, cr_d_block_8,
plot_compared_images(np.log(np.abs(y_d_block_8) + 0.001),np.log(np.abs(y_quant) + 0.
```





In [52]:

```
y_quant_inv, cb_quant_inv, cr_quant_inv = inverse_quantization(y_quant, cb_quant, cr
plot_compared_images(np.log(np.abs(y_d_block_8) + 0.001), np.log(np.abs(y_quant_inv)
```



Pela análise dos gráficos anteriores, verifica-se que, à medida que a qualidade diminui, os valores resultantes da aplicação da quantização aproximam-se cada vez mais de preto (valor 0). É ainda de notar que, com a diminuição da qualidade, a perda de informação aumenta, dado que o resultado da quantização torna a maioria dos coeficientes AC igual a zero, e como tal, impossíveis de recuperar. Como a redundância dos coeficientes AC aumenta com a diminuição da qualidade, a aplicação de modelos de compressão entrópicos torna-se mais eficaz e, assim, o potencial de compressão aumenta também.

Comparando com os resultados obtidos da aplicação da DCT em blocos 8x8 à imagem, é possível observar que as frequências altas desaparecem após a quantização. O facto de na DCT em blocos 8x8 existirem poucas frequências altas, torna a eliminação das mesmas pouco percetível para o olho humano.

Codificação DPCM dos coeficientes DC

In [53]:

```
def dpcm(channel):
    """
    Given a channel, the function will return a channel with the same size, but with
    encoded.

    :param channel: The channel to be processed
    :return: the encoded channel.
    """
    size = channel.shape
    dpcm_image = copy.deepcopy(channel.astype(np.float32))
    prev = channel[0, 0]
    for i in np.r_[:size[0]:8]:
        for j in np.r_[:size[1]:8]:
            if i == 0 and j == 0:
                continue
            dc = channel[i, j]
            dpcm_image[i, j] = dc - prev
            prev = dc

    return dpcm_image
```

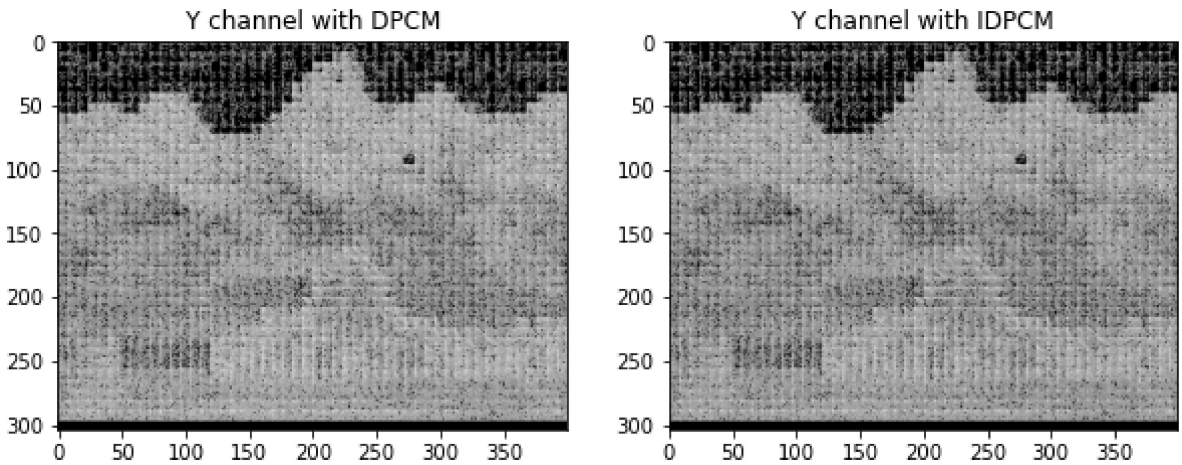
In [54]:

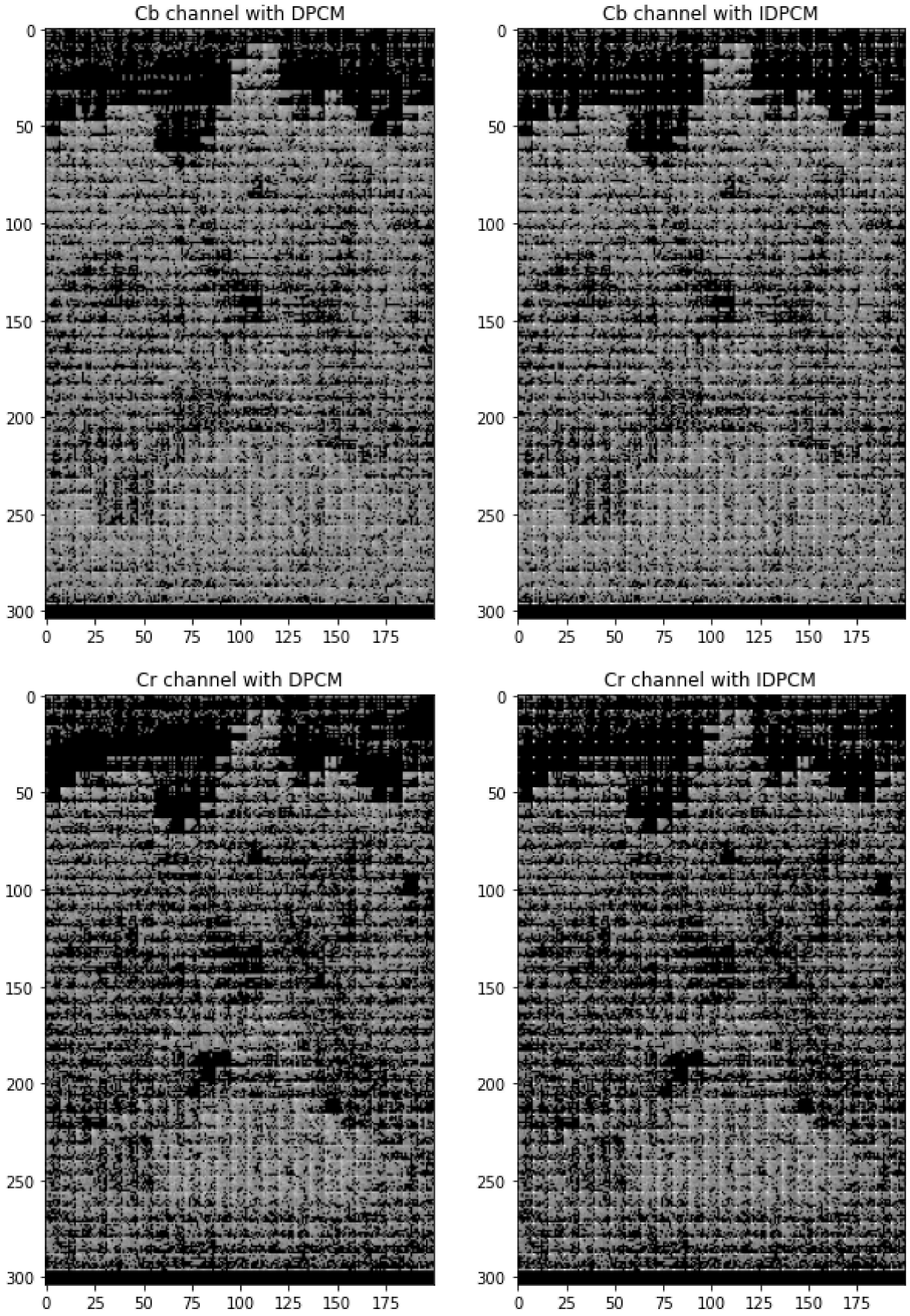
```
def idpcm(channel):
    """
    Given a channel, the function returns the channel with the decoding DPCM values

    :param channel: The channel to be processed
    :return: the decoded channel.
```

```
    """
    size = channel.shape
    image = copy.deepcopy(channel.astype(dtype=np.float32))
    prev = channel[0, 0]
    for i in np.r_[:size[0]:8]:
        for j in np.r_[:size[1]:8]:
            if i == 0 and j == 0:
                continue
            image[i, j] = channel[i, j] + prev
            prev = image[i, j]

    return image
```

In [55]:

```
y_dpcm = dpcm(y_quant)
cb_dpcm = dpcm(cb_quant)
cr_dpcm = dpcm(cr_quant)

y_idpcm = idpcm(y_dpcm)
cb_idpcm = idpcm(cb_dpcm)
cr_idpcm = idpcm(cr_dpcm)

plot_compared_images(np.log(np.abs(y_dpcm) + 0.001), np.log(np.abs(y_idpcm) + 0.001)
plot_compared_images(np.log(np.abs(cb_dpcm) + 0.001), np.log(np.abs(cb_idpcm) + 0.00
plot_compared_images(np.log(np.abs(cr_dpcm) + 0.001), np.log(np.abs(cr_idpcm) + 0.00
```



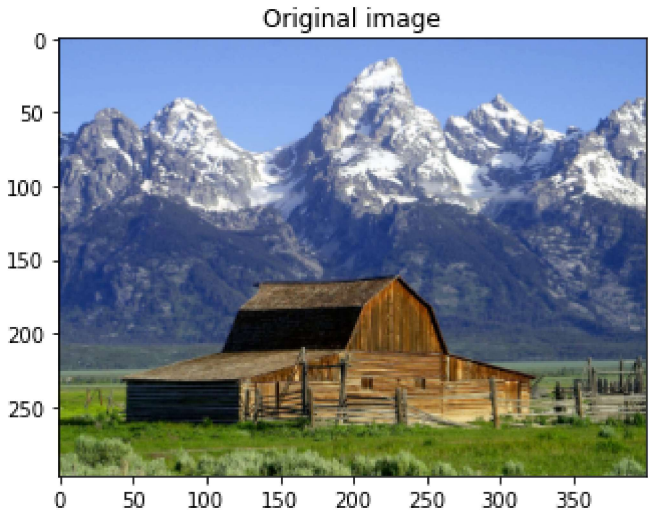


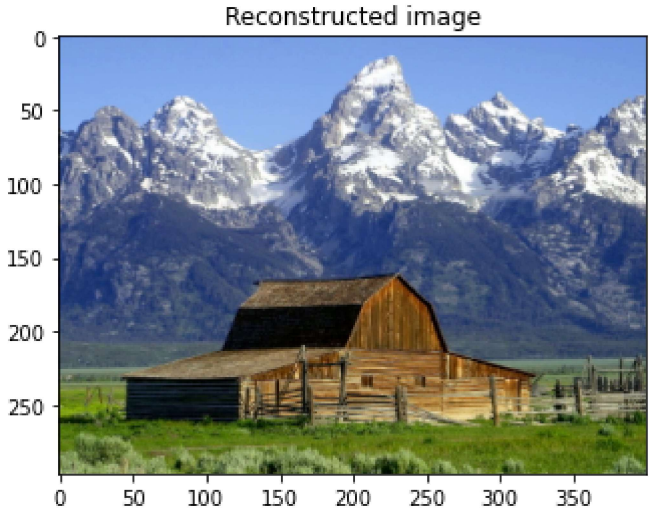
Aos aplicar a DPCM por blocos 8x8, geralmente a diferença entre os coeficientes DC codificados de blocos adjacentes irá ser próximo de zero pois blocos adjacentes tendem a ser semelhantes, principalmente em imagens foto-realistas. Estes resultados irão possibilitar uma maior compressão, sobretudo ao aplicar métodos de compressão entrópica como RLE.

Reconstrução da imagem original

In [56]:

```
y_quant_inv, cb_quant_inv, cr_quant_inv = inverse_quantization(y_idpcm, cb_idpcm, cr
y_quant_idct_inv = idct_block(y_quant_inv, 8)
cb_quant_idct_inv = idct_block(cb_quant_inv, 8)
cr_quant_idct_inv = idct_block(cr_quant_inv, 8)

y_upsampling, cb_upsampling, cr_upsampling = upsampling(y_quant_idct_inv, cb_quant_i
reconstructed_image = get_image_from_channels((y_upsampling, cb_upsampling, cr_upsam
reconstructed_image = revert_padding(reconstructed_image, size[0], size[1])

reconstructed_image = convert_ycbcr_to_rgb(reconstructed_image)
plot_image(image, title='Original image')
plot_image(reconstructed_image, title='Reconstructed image')
```





Encoder e decoder

In [57]:

```
def encoder(original, ratio, interpolation, quality=75):
    """
    Given an image, the function will first add padding to the image, then convert t
    downsample the image, apply DCT to each block, quantize the DCT coefficients, ap
    pulse-code modulation, and return the three DPCM coefficients.

    :param original: the original image
    :param ratio: The downsampling ratio
    :param interpolation: the interpolation method used for downsampling
    :param quality: the quality of the image, defaults to 75 (optional)
    :return: a tuple containing the dpcm coefficients of the Y, Cb and Cr channels a
    """
    #plot_image(original, title="Original image")
```

```
    shape = original[:, :, 0].shape
    image = add_padding(original)
    image = convert_rgb_to_ycbcr(image)
    y, cb, cr = downsampling(image, ratio, interpolation)
    y_d = dct_block(y, 8)
    cb_d = dct_block(cb, 8)
    cr_d = dct_block(cr, 8)

    y_quant, cb_quant, cr_quant = quantization(y_d, cb_d, cr_d, quality)
    y_dpcm = dpcm(y_quant)
    cb_dpcm = dpcm(cb_quant)
    cr_dpcm = dpcm(cr_quant)

    return (y_dpcm, cb_dpcm, cr_dpcm), shape
```

In [58]:

```
def decoder(channels, size, ratio, interpolation, quality=75):
    """
    Given the DPCM compressed channels, the quantization quality, and the interpolat
    we can reconstruct the image

    :param channels: The three channels of the image (y, cb, cr)
    :param size: The size of the original image
    :param ratio: The ratio used in to encode the channels
    :param interpolation: The interpolation value
    :param quality: The quality of the image,defaults to 75 (optional)
    :return: the reconstructed image.
    """
    y_idpcm = idpcm(channels[0])
    cb_idpcm = idpcm(channels[1])
    cr_idpcm = idpcm(channels[2])

    y_iquant, cb_iquant, cr_iquant = inverse_quantization(y_idpcm, cb_idpcm, cr_idpc

    y_di = idct_block(y_iquant, 8)
    cb_di = idct_block(cb_iquant, 8)
    cr_di = idct_block(cr_iquant, 8)
    y, cb, cr = upsampling(y_di, cb_di, cr_di, ratio, interpolation)
    image = get_image_from_channels((y, cb, cr))
    image = convert_ycbcr_to_rgb(image)
    image = revert_padding(image, size[0], size[1])
    #plot_image(image, title="Reconstructed Image")

    return image
```

Métricas de distorção das imagens

In [59]:

```
def mse_error(original, reconstructed):
    """
    Compute the mean squared error between the original and reconstructed images

    :param original: the original image
    :param reconstructed: the reconstructed image
    :return: The MSE error
    """
    size = original.shape
    mse = (1 / (size[0] * size[1])) * np.sum(np.power((original - reconstructed), 2)
    return mse
```

In [60]:

```
def rmse_error(mse):
```

```
    """
    Return the square root of the mean squared error

    :param mse: Mean squared error
    :return: rmse_error(mse)
    """
    return m.sqrt(mse)
```

In [61]:

```
def snr_error(original, mse):
    """
    Given an original image and an MSE value, calculate the SNR of the image

    :param original: the original image
    :param mse: the mean squared error between the original and the reconstructed im
    :return: The SNR value
    """
    size = original.shape
    p = (1/(size[0] * size[1])) * np.sum(np.power(original, 2))
    return 10 * m.log10(p / mse)
```

In [62]:

```
def psnr_error(original, mse):
    """
    Given an original image and an image with a certain error,
    return the PSNR of the error

    :param original: the original image
    :param mse: The mean squared error between the two images
    :return: The PSNR value
    """
    return 10 * m.log10((np.max(original)**2) / mse)
```

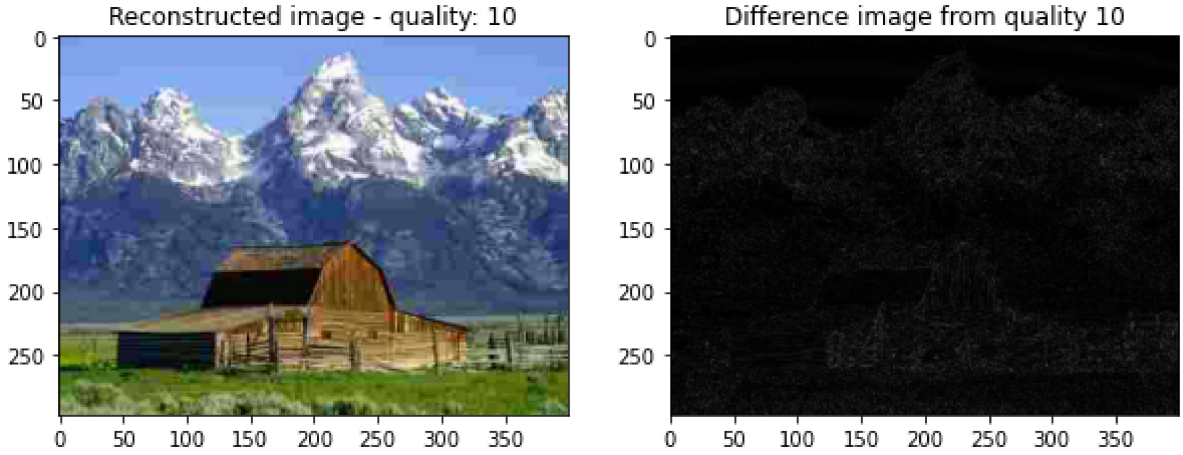
In [63]:

```
def compare_results(image_name, quality, ratio=(4, 2, 0), interpolation = False):
    """
    Given an image, a quality, and a ratio, it encodes the image, decodes it, and co
    image with the decoded image

    :param image_name: the name of the image to be used for the test
    :param quality: The quality of the jpeg image. This is a value between 1 and 100
    """
    original = read_image(f"./imagens/{image_name}.bmp")
    channels, shape = encoder(original, ratio, interpolation, quality)
    original_ycbcr = convert_rgb_to_ycbcr(original)
    image_r = decoder(channels, shape, ratio, interpolation, quality)
    img.imsave(f"{image_name}_{quality}.png", image_r)
    image_r_ycbcr = convert_rgb_to_ycbcr(image_r)
    diff_image = np.abs(original_ycbcr[:, : , 0].astype(np.int16) - image_r_ycbcr[:,
    diff_image[0, 0] = 255

    plot_compared_images(image_r, diff_image, f"Reconstructed image - quality: {qual
    mse = mse_error(original.astype(np.float32), image_r.astype(np.float32))
    print("Diff Image: " + image_name + " Quality: " + str(quality))
    print("MSE: " + str(mse))
    print("RMSE: " + str(rmse_error(mse)))
    print("SNR: " + str(snr_error(original.astype(np.float32), mse)))
    print("PSNR: " + str(psnr_error(original.astype(np.float32), mse)))
```

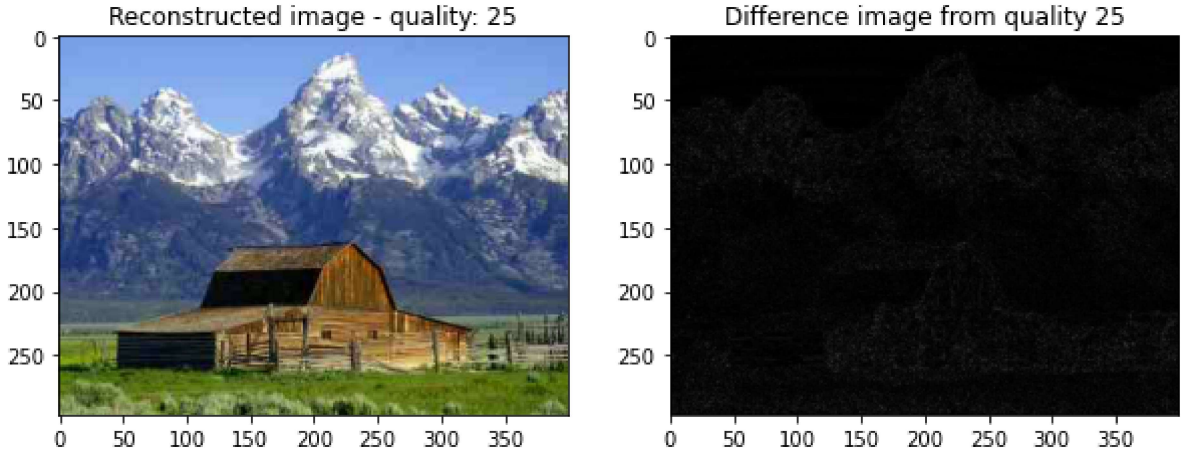
In [64]:

```
compare_results("barn_mountains", 10)
compare_results("barn_mountains", 25)
compare_results("barn_mountains", 50)
```

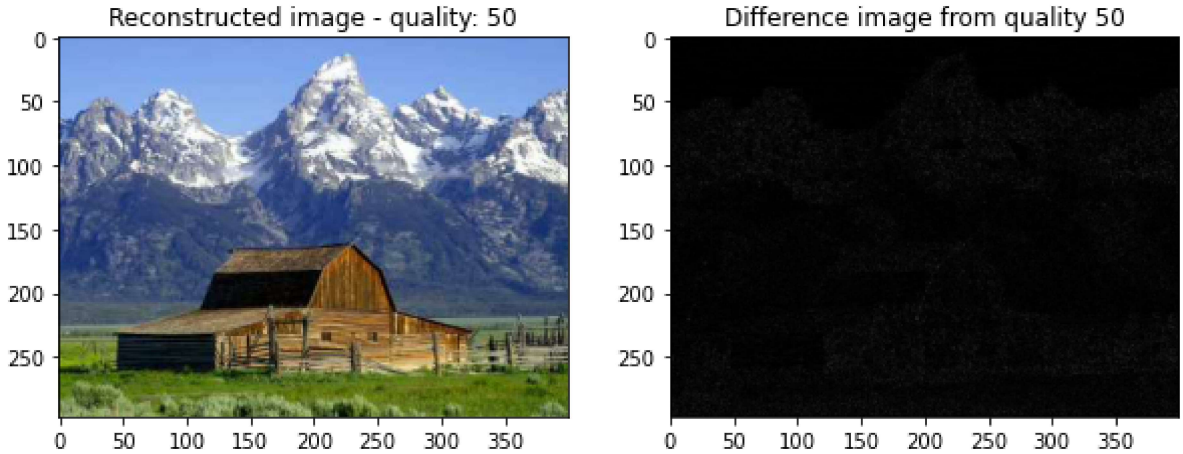
```
compare_results("barn_mountains", 75)
compare_results("barn_mountains", 100)
```



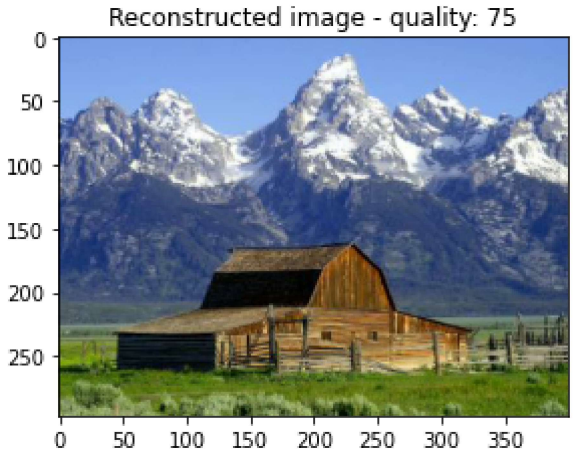

```
Diff Image: barn_mountains Quality: 10
MSE: 746.9398653198652
RMSE: 27.33020060884781
SNR: 18.45342027332363
PSNR: 19.397947218517164
```

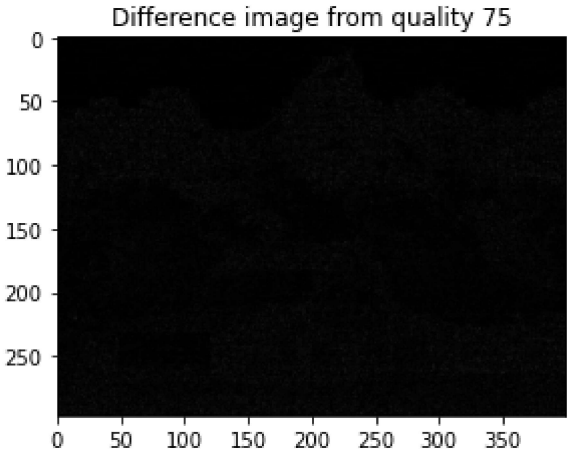



```
Diff Image: barn_mountains Quality: 25
MSE: 433.41010101010096
RMSE: 20.818503812956898
SNR: 20.817286376254206
PSNR: 21.76181332144774
```

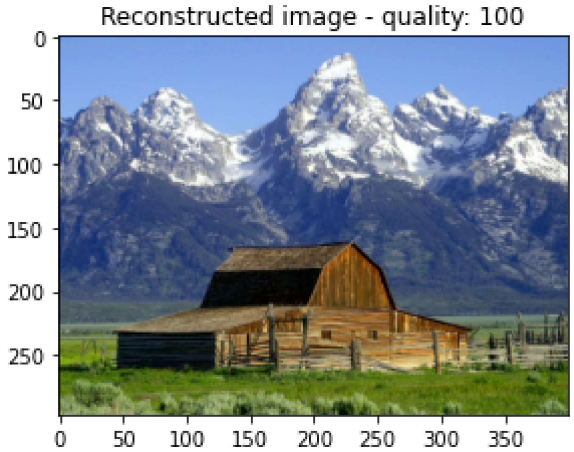


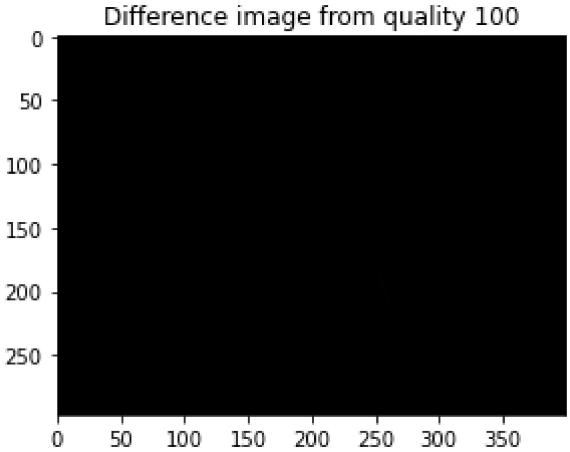

```
Diff Image: barn_mountains Quality: 50
MSE: 296.19717171717167
RMSE: 17.21037976679108
SNR: 22.4704675906914
PSNR: 23.414994535884937
```






```
Diff Image: barn_mountains Quality: 75
MSE: 190.62340067340065
RMSE: 13.806643352871857
SNR: 24.384514533646104
PSNR: 25.329041478839642
```

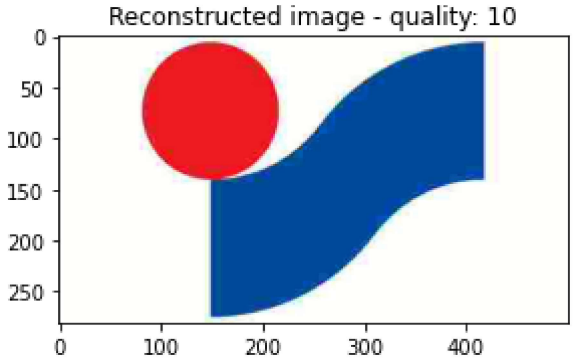


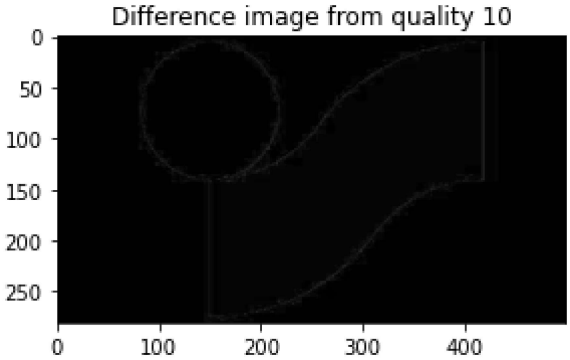



```
Diff Image: barn_mountains Quality: 100
MSE: 59.72268518518518
RMSE: 7.7280453664031485
SNR: 29.424883405893972
PSNR: 30.369410351087502
```

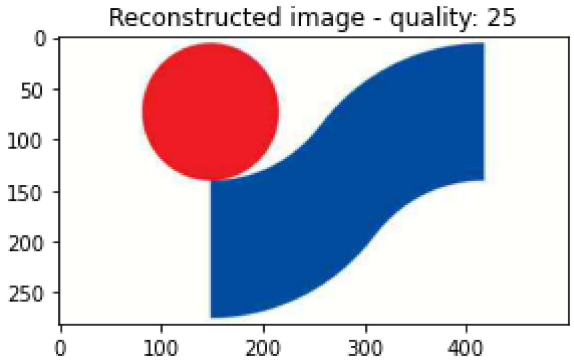
In [65]:

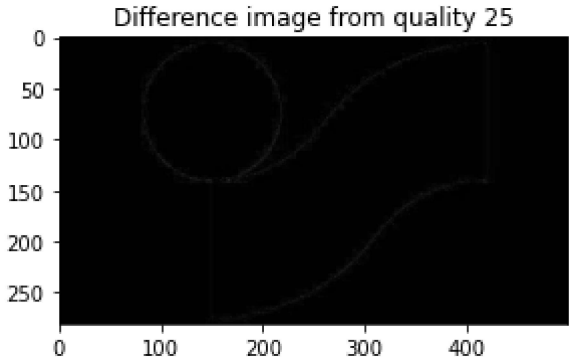
```
compare_results("logo", 10)
compare_results("logo", 25)
compare_results("logo", 50)
compare_results("logo", 75)
compare_results("logo", 100)
```



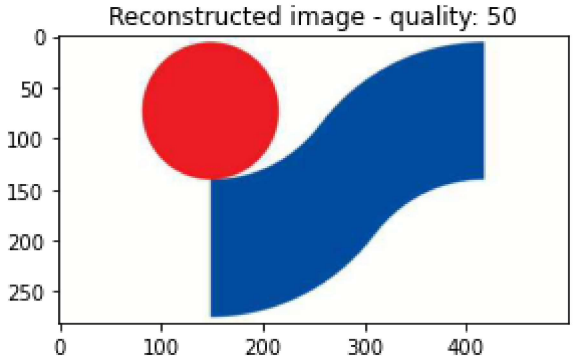


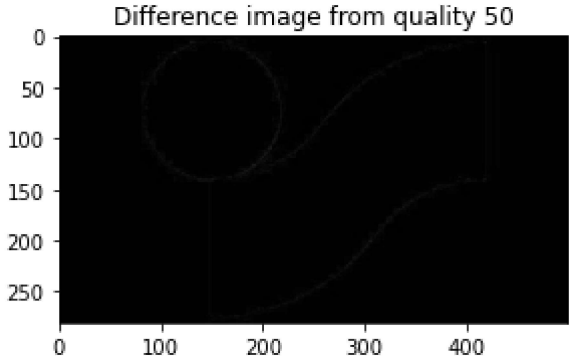

```
Diff Image: logo Quality: 10
MSE: 226.3775658362989
RMSE: 15.045848790822633
SNR: 27.783785351470943
PSNR: 24.58246975084256
```



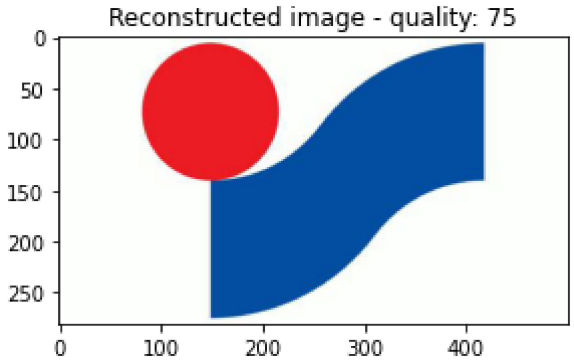


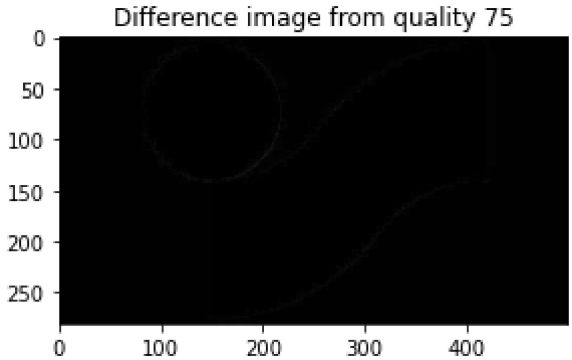

```
Diff Image: logo Quality: 25
MSE: 112.15037722419929
RMSE: 10.590107517121783
SNR: 30.834111822350106
PSNR: 27.632796221721723
```



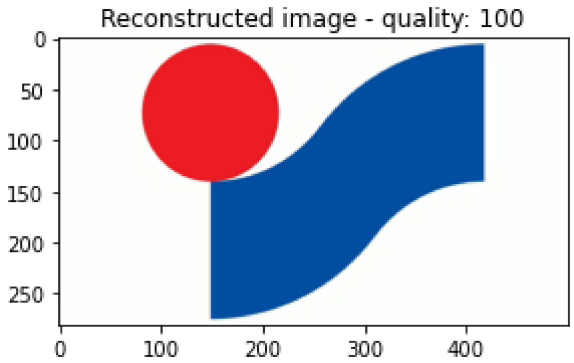


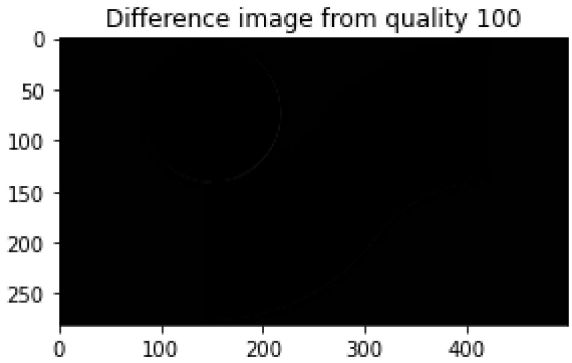

```
Diff Image: logo Quality: 50
MSE: 91.02922419928825
RMSE: 9.540923655458535
SNR: 31.740310794782992
PSPNR: 28.53899519415461
```






```
Diff Image: logo Quality: 75
MSE: 72.52864056939501
RMSE: 8.516374849041991
SNR: 32.727023835316416
PSNR: 29.525708234688032
```



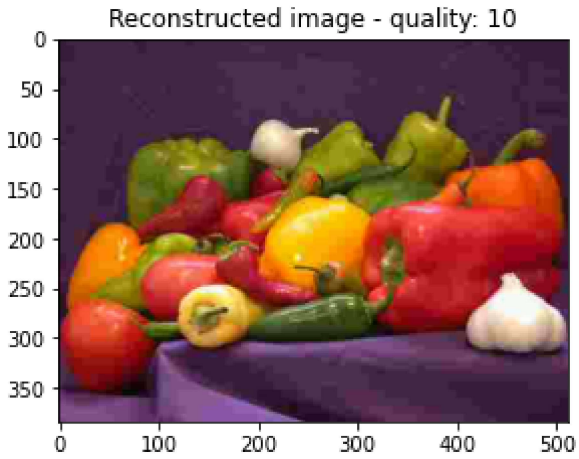


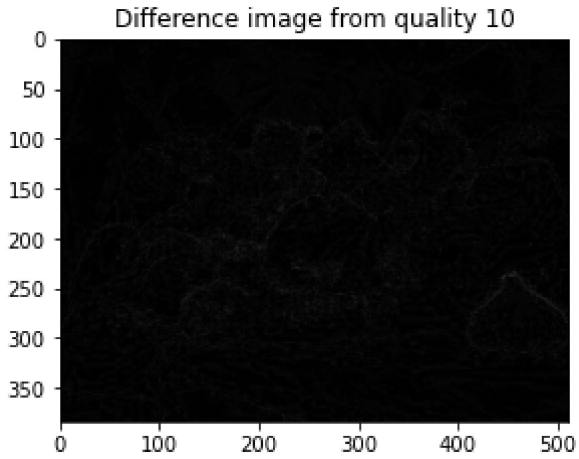

```
Diff Image: logo Quality: 100
```

```
MSE: 52.07978647686833
RMSE: 7.216632627262408
SNR: 34.16542725941042
PSNR: 30.964111658782038
```

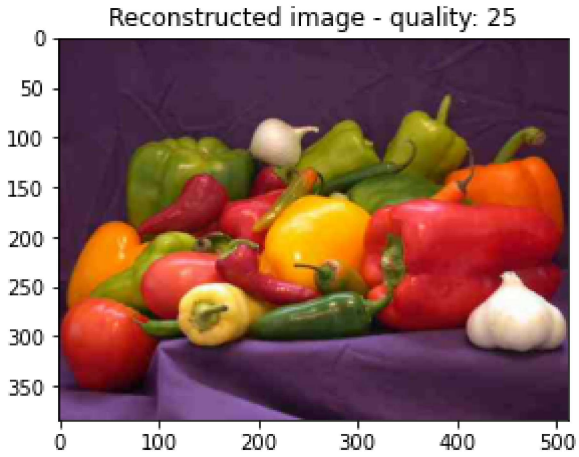
In [66]:

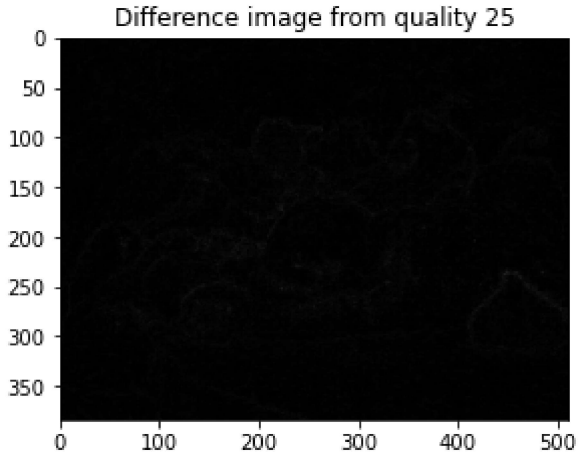
```
compare_results("peppers", 10)
compare_results("peppers", 25)
compare_results("peppers", 50)
compare_results("peppers", 75)
compare_results("peppers", 100)
```



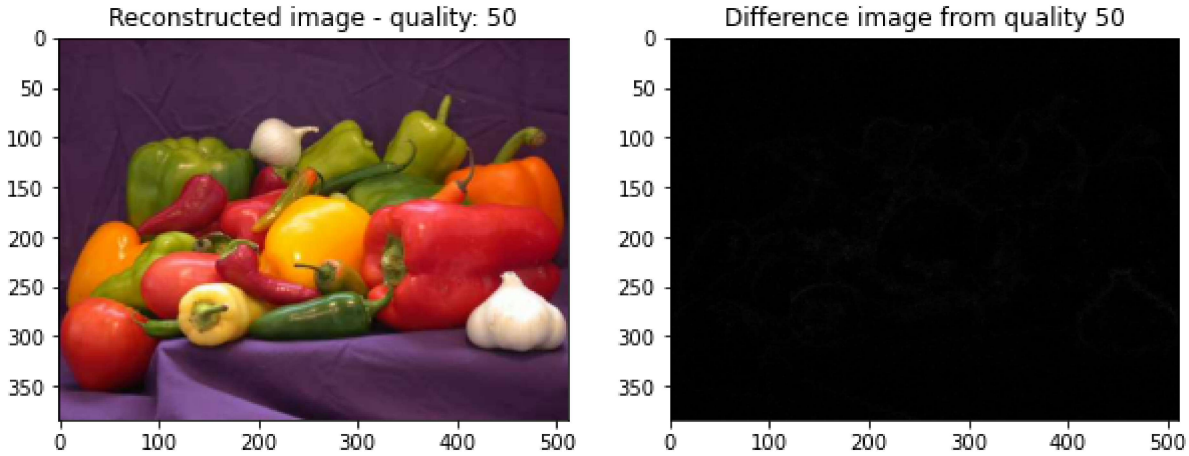



```
Diff Image: peppers Quality: 10
MSE: 364.9576009114583
RMSE: 19.103863507454673
SNR: 19.303528440151663
PSNR: 22.50837947808035
```

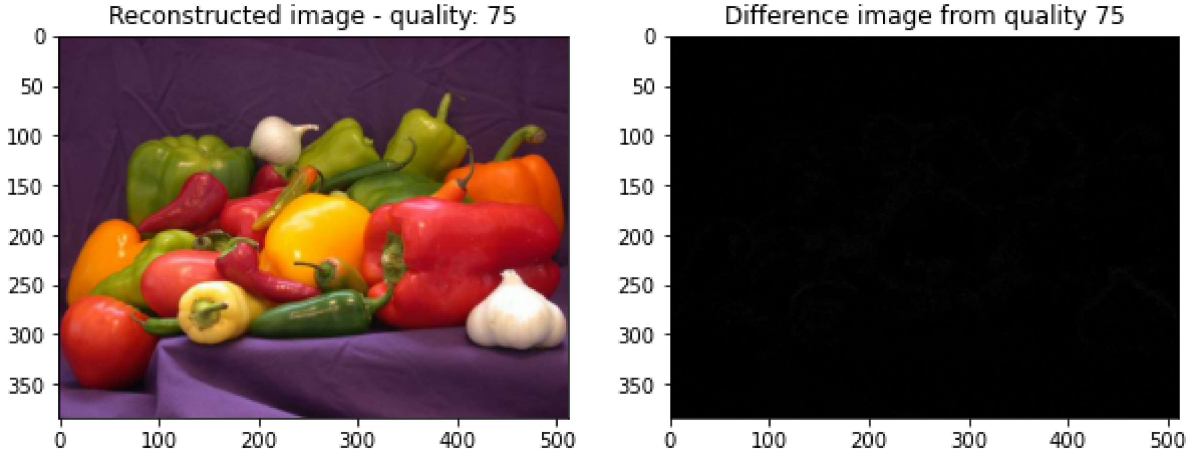




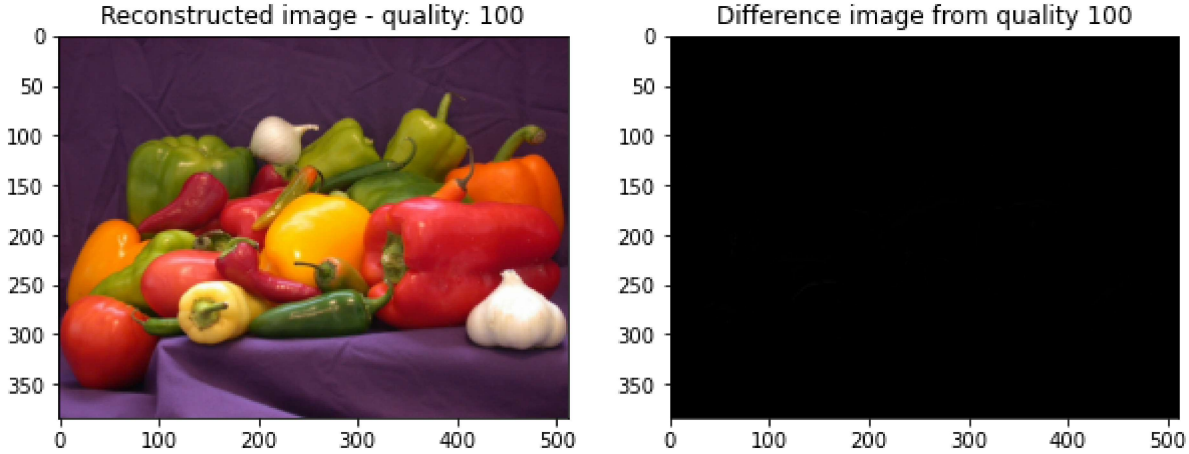

```
Diff Image: peppers Quality: 25
MSE: 190.23295084635416
RMSE: 13.792496178950138
SNR: 22.133095124478736
PSNR: 25.337946162407423
```




```
Diff Image: peppers Quality: 50
MSE: 136.0071818033854
RMSE: 11.662211702905475
SNR: 23.59033415357547
PSNR: 26.795185191504153
```




```
Diff Image: peppers Quality: 75
MSE: 107.15592447916666
RMSE: 10.351614583202306
SNR: 24.625790695881765
PSNR: 27.830641733810452
```




```
Diff Image: peppers Quality: 100
MSE: 67.46712239583333
RMSE: 8.213837251603742
SNR: 26.635030700301957
PSNR: 29.839881738230645
```

Resultados do encoder com interpolação e racio 4:2:2

Original	100%	75%	50%	25%	10%

	Original	100%	75%	50%	25%	10%
Images						
MSE	--	14.49	152.84	261.87	398.95	706.87
RMSE	--	3.81	12.36	16.18	19.97	26.59
SNR	--	35.57	25.34	23.01	21.18	18.69
PSNR	--	36.52	26.29	23.95	22.12	19.63

	Original	100%	75%	50%	25%	10%
Images						
MSE	--	10.90	51.08	77.76	126.31	278.71
RMSE	--	3.20	7.15	8.82	11.23	16.69
SNR	--	34.55	27.84	26.02	23.91	20.47
PSNR	--	37.76	31.05	29.22	27.12	23.68

	Original	100%	75%	50%	25%	10%
Images						
MSE	--	6.70	24.28	48.41	70.86	170.45
RMSE	--	2.59	4.93	6.96	8.41	13.06
SNR	--	43.07	37.48	34.48	32.83	29.02
PSNR	--	39.87	34.28	31.28	29.63	25.81

Numa análise quantitativa, pode-se observar que pela métrica MSE, que acentua mais as diferenças entre as imagens quando comparado com outras métricas, o erro cresce de forma quase linear até aos 25% e depois para 10% existe perda substancial de informação. Isto confirma a animação inicialmente criada para visualizar as diferentes imagens comprimidas com qualidade de 100 até 0. Nesta animação é possível verificar que a imagem vai perdendo pouca informação à medida que a qualidade diminiu, porém tem uma perda muito mais acentuada a partir da qualidade 25%/20%. As animações podem ser encontradas na pasta "imagens" com os seguintes nomes: barn-mountains.bmp.gif, peppers.bmp.gif e logo.bmp.gif

Pela visualização das imagens das diferenças para as várias qualidades, como era expectável, existe uma maior perda de informação para qualidades inferiores. Esta perda diminui à medida que aumentamos a qualidade de compressão.

Verifica-se em todas as imagens que, a partir da qualidade 75 para cima, quase não são visíveis perdas percetuais, podendo-se confirmar pela imagem das diferenças que se encontra quase toda preta. Para qualidades inferiores a 75, a perda é mais notória, sendo possível observar a formação de ruído principalmente próximo dos contornos. Em contraste, se fossem

desenvolvidos os restantes passos do JPEG, seria possível atingir taxas de compressão bastante elevadas, semelhantes às taxas obtidas pelo JPEG.

Em suma, os resultados obtidos confirmam a primeira análise feita sobre o codec JPEG, pois as imagens obtidas são bastante semelhantes. É possível observar, mais uma vez, que a parte destrutiva do codec funciona melhor em imagens foto-realistas. Isto deve-se ao facto das perdas nestas imagens serem quase impercétiveis ao olho humano.

In [67]:

```
! jupyter nbconvert --to html TP1.ipynb
```

```
[NbConvertApp] Converting notebook TP1.ipynb to html
[NbConvertApp] Writing 11799115 bytes to TP1.html
```