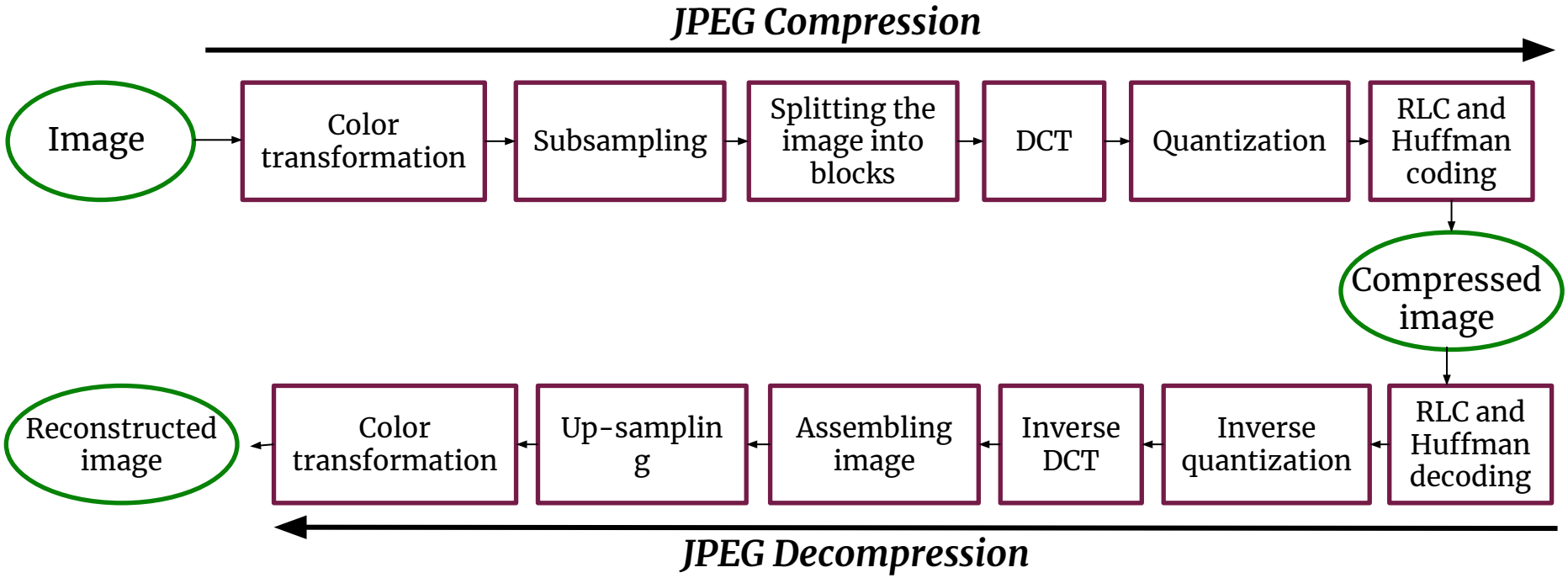
Image compression

Lab 1: JPEG

ARDILA Manuela
LASSO Nicolas

JPEG encoder/decoder

In this project, a simplified JPEG encoder/decoder is programmed in Python. The objective is therefore initially the compression and then the decompression of an image, according to the steps described in the diagram below:



Each pixel is thus coded on one byte, so the values are between 0 (representing the color black) and 255 (representing the color white).

Then we will determine from the base image, the compression result and the decompressed image the compression ratio;

Color transformation

Conversion to luminance / chrominance

The first step in the JPEG process is to convert the image from its original color space to the (YC_bC_r) representation, which separates the image into its luminance and chrominance.

Original image

The starting image is an array of MxN pixels, where each pixel is coded on three bytes that represent the intensities of the three color components: red, green, and blue.

To convert the image to luminance/chrominance representation (YC_bC_r) there are two differen fromats:

For the first one, it just was necessary to aply the converting formulas, to get the three new matrices, which have the same dimesions, equal to the dimensions of the original image (MxN)

```
R=img[:,:,0]
G=img[:,:,1]
B=img[:,:,2]

Y=0.299*R+0.587*G+0.114*B
Cb=-0.1687*R-0.3313*G+0.5*B+128
Cr=0.5*R-0.4187*G-0.0813*B+128
```

For the second one, the python function mean_in_2x2 is defined. This function do a subsampling process, for that block of 2x2 are taken and those are averaged, this value is assigned in a new array. This function is applied for the chrominances matrixes C_b and C_r This second format is more efficient but has losses of information.

```
def mean_in_2x2(image):
    x = int(len(image[:,0])/2) #  x=M/2
    y = int(len(image[0,:])/2) #  y=N/2
    blocks = np.zeros((x,y))   # New array x*y
    for i in range(0,x,1):
        for j in range(0,y,1): # Take blocks of size 2x2
            block = np.array(image[2*i:2*i+2, 2*j:2*j+2])
            block = np.mean(block) # Compute the mean
            blocks[i,j]=block # Assing the mean to the new array
    return np.array(blocks)
```

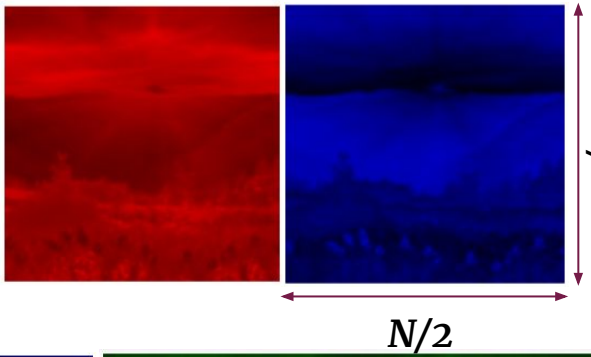
Color transformation

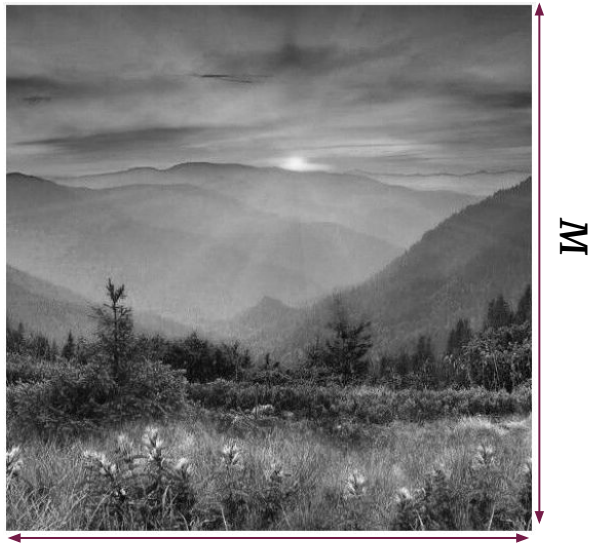
Conversion to luminance / chrominance

The matrices calculated are plotted as images to graphically notices the results:

For the luminance/chrominance representation without subsampling, the results are similar but C_b and C_r are Mx N

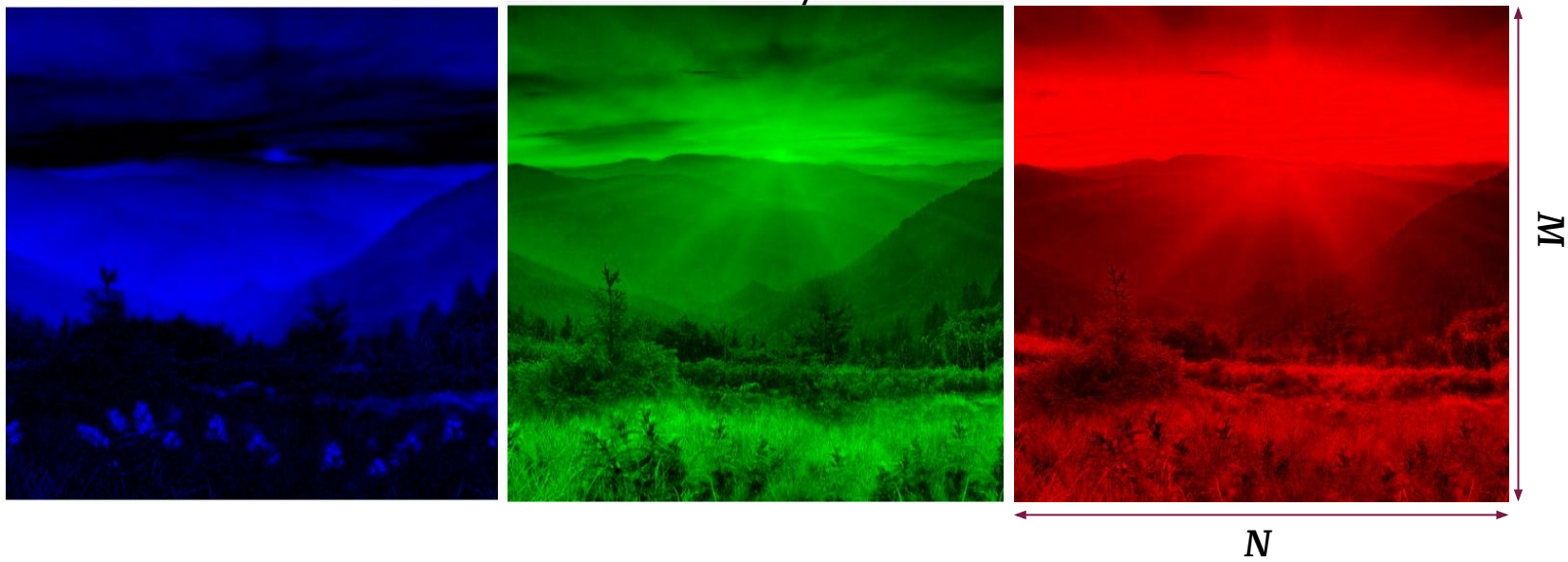
Luminance/chrominance representation with subsampling





The subsampled is applied in the chrominance components and not in the luminance because the human eye is less sensitive to color changes than changes in brightness.

Original representation RGB

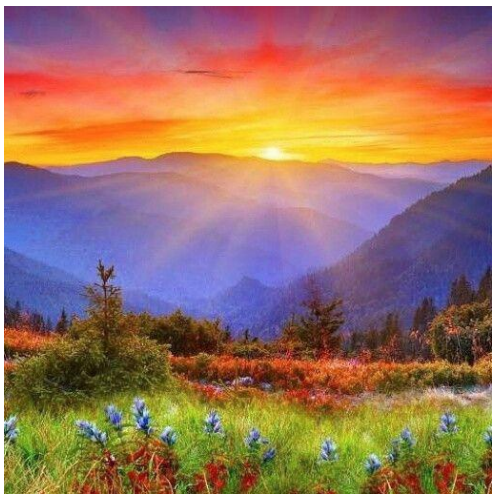


Color transformation

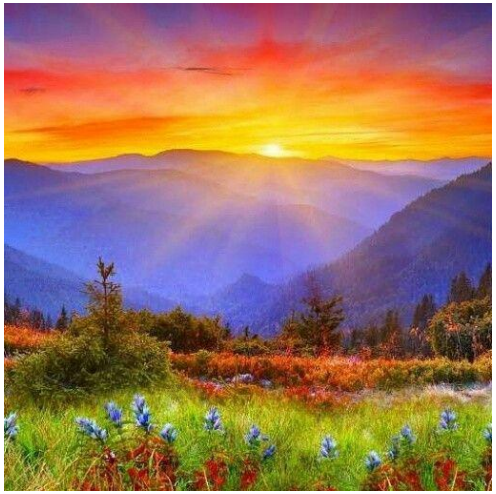
Conversion to luminance / chrominance

Now, using both representations the image is construed using the code:

```
def Add_twice(image):
    blocks = np.zeros((2*len(image[:,0]),2*len(image[0,:])))
    for i in range(0,len(image),1):
        for j in range(0,len(image),1):
            blocks[2*i:2*i+2, 2*j:2*j+2]=image[i,j]
    return np.array(blocks)

Cr_n=Add_twice(Cr)
Cb_n=Add_twice(Cb)

R=Y+1.14020*(Cr_n-128);
G=Y-0.34414*(Cb_n-128)-0.71414*(Cr_n-128)
B=Y+1.77200*(Cb_n-128)

rgb= np.dstack((R,G,B))
imshow(rgb)
```



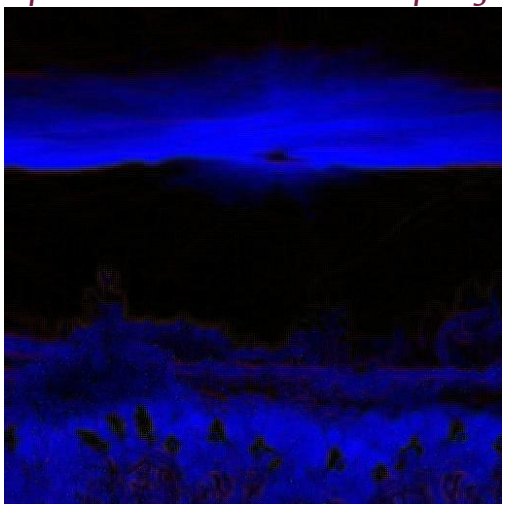
Original representation RGB



Luminance/chrominance representation with subsampling

As it was excepted the result are visually identical.

To notice the error caused by the subsampling, the difference between both images is plotted. In a perfect match, each pix will be zero and that's why most of the image is back, but especially in the divisor lines of the images, there are pixels different from zero.



As the error is also visually unperceivable, the difference is multiplied by a factor of x10

Splitting the image into blocks

Conversion to luminance / chrominance

makenxnblocks(img, n) is a function that receives an image and returns a list of nxn blocks

The image is divided into 8x8 pixel blocks, and each block (data unit) will be processed independently.

```
def makenxnblocks(img, n=8):
    blocks = []
    for i in range(0, img.shape[0], n):
        for j in range(0, img.shape[1], n):
            blocks.append(img[i:i+n, j:j+n])
    return blocks
```

If the dimensions of the component are not multiples of 8, the image is completed by duplication of the last row (or column) until obtaining the multiple of 8 immediately above.

Discrete Cosine Transform (DCT)

Centering

*Before applying the Discrete Cosine Transform (DCT), it is important to center or shifts the image data, **for removing any DC offset or bias from the image data**, which can affect the efficiency of the compression process. Due to if the image data contains a DC offset, it can create a high-frequency component in the DCT that is difficult to compress*

For that, from each value of each block, which contains numbers between 0 and 255, are subtracted 128, in order to obtain values between -128 and 127 (by subtracting 128 from each value).

Centering() receives a list of blocks and returns a list of centered block by subtracting 128 to each pixel

```
def centering(blocks):
    centered = []
    for block in blocks:
        centered.append(block - 128)
    return np.array(centered)
```

The results image is center, it has a zero mean, and it's ready for the DCT transformation and subsequent quantization.

Discrete Cosine Transform (DCT)

The DCT transforms the pixel values of each block into a set of coefficients that represent the block's spatial frequency content. By breaking down the image into frequency components, the DCT enables efficient compression of the image data while theoretically preserving all information.

DTC() receives a list of blocks and returns a list of DCT coefficients of each block

```
def DCT(blocks):
    dcts= []
    for block in blocks:
        dcts.append(cv2.dct(block))
    return np.array(dcts)
```

- The DCT is theoretically an operation ***without loss of information*** *because the initial coefficients can be found by applying the inverse DCT.*

- The DCT ***can be separated*** *into one-dimensional transformations that are applied successively to the rows and columns of the image.*

The Discrete Cosine Transform *is a mathematical technique used to convert an image from the spatial domain to the frequency domain*

Discrete Cosine Transform (DCT)

display the DCTs of different blocks and interpret the results

Block:

```
[[142 142 143 143 144 143 141 138]
 [142 142 142 143 143 141 138 137]
 [141 141 141 140 140 139 137 134]
 [139 139 139 139 139 138 135 134]
 [136 136 136 136 137 137 136 134]
 [135 135 135 135 136 135 135 134]
 [135 134 134 134 134 134 134 133]
 [135 135 134 134 134 134 133 132]]
```

DCT:

```
[[74.8  7.4 -5.7  3.5 -0.8 -0.1 -0.1  0.5]
 [23.2  3.2 -3.8  0.7 -0.1 -0.2  0.3  0.2]
 [ 3.2 -0.3  0.1  0.   0.2  0.2 -0.1 -0.1]
 [ 0.5 -4.   0.2 -0.   0.1  0.2 -0.2  0.6]
 [ 0.5 -0.6 -0.5  0.5 -0.5 -0.2  0.2 -0.2]
 [-0.1  0.4  0.2 -0.  -0.3  0.4  0.3  0.2]
 [ 0.   0.2  0.1  0.1 -0.9  0.1 -0.3  0.5]
 [-0.8 -0.6  0.6  0.5 -1.   0.7 -0.1  0.4]]
```

The values in the DCT result represent the amplitudes of the different frequency components that make up the input signal or image.

The first element (o,o) in the DCT result represents the DC component, which is the average value of the input signal. The other elements in the matrix represent the different frequency components, with higher values indicating higher frequencies.

These DCT coefficients are used to identify the parts of the image that can be discarded or quantized with less precision, in order to achieve a smaller file size without significant loss of image quality.

Quantization

Introducing loss in compression

```python
def quantization(block, flag="RGB"):
    """
    Applies quantization to a block by dividing the
    DCT coefficients by the quantization table.

    Args:
    - block: A NumPy array that represents a block.
    - flag: A string that specifies the quantization table to use.
    Default is "RGB".

    Returns:
    - A NumPy array that represents the quantized block.
    """
    if flag == "RGB":
        return np.round(block/qY)
    else:
        return np.round(block/qC)
```

Quantization(DCTs, flag) takes the floor round of the DCT coefficients divided by the quantization table. This function takes a flag that is "RGB" by default, but can be "chrominance". Depending on the flag the quantization table is chosen between RGB or YC_bC_r *respectively*

Quantization is a step in image compression where each block is divided by a pre-defined 8x8 quantization matrix. This reduces the number of bits needed to represent the coefficients, thereby reducing the size of the compressed image. However, the quantization step introduces the most errors due to floor rounding, and its goal is to attenuate high frequencies as the human eye is less sensitive to them.

Quantization matrices regulate the losses and, therefore, the compression ratio, but in practice, pre-calculated matrices are used. The error encountered may be due to the use of a pre-calculated quantization matrix, and strictly speaking, these matrices should be calculated for each image, considering the desired compression rate and properties of the image and the eye.

Quantization

Introducing loss in compression

JPEG compression mainly introduces loss through quantization. Each block is divided by a pre-defined quantization matrix to reduce the size of the compressed image, ***but this step introduces the most errors.*** *The goal is to attenuate high frequencies as the human eye is less sensitive to them. The pre-calculated quantization matrices regulate the losses and compression ratio, but strictly speaking, custom matrices should be used for each image.*

ZigZag and Run-Length

vector representation

```python
def rlencode(data:list, symbol:int=0, escape=257) -> np.ndarray:
    '''
    Encode a list of values using run length encoding
    when `symbol` is encountered,
    the next value is `escape` followed by the number of `symbol`.
    '''
    out = []
    symbols = 0
    for i in range(len(data)):
        if data[i] == symbol:
            symbols += 1
        else:
            if symbols > 0:
                out.append(escape)
                out.append(symbols)
                out.append(data[i])
                symbols = 0
            else:
                out.append(data[i])
    if symbols > 0:
        out.append(escape)
        out.append(symbols)
    return np.array(out)
```

Using the previously defined function to traverse a ZigZag array and the Run-Length encoding, we can reduce a block to its representation depending on how many o's we got when performing the quantization.

This is that when a range of o is detected, the length of this range must be specified, preceded by a number 257. In this way, by keeping our frequencies in the upper corner of the matrix, thanks to the DCT, and the ZigZag , we can store an entire block in long arrays of 7 elements, on average.

Differents Pipeline Results

Original Image

lena.jpg

Differents Pipeline Results

Grayscale Pipeline

Original Image

Compressed Image

"Error Image"

Compression ratio: 1.0
MSE: 31536.65141433868
RMSE: 177.58561713815305
PSNR: 3.1426478314579445

Differents Pipeline Results

RGB Pipeline

Original Image

Compressed Image

"Error Image"

Compression ratio: 1.0
MSE: 36186.9229571594
RMSE: 190.22860709462023
PSNR: 2.5452870507500593

Differents Pipeline Results

Chrominance without downsampling Pipeline

Original Image

Compressed Image

"Error Image"

Compression ratio: 1.0
MSE: 34.9731038402405
RMSE: 5.913806205840745
PSNR: 32.693461834951485

Differents Pipeline Results

Chrominance with downsampling Pipeline

Original Image

Compressed Image

"Error Image"

Compression ratio: 1.0
MSE: 34.217145502777996
RMSE: 5.849542332762281
PSNR: 32.78836584303978