# PROJET

## CHAINE COMPLÈTE DE COMPRESSION D'IMAGES


Réalisé par

**TRAN Thanh Duy**

Master 1 ICONE

# Part 1 : Overview

The main purpose of this project is to present how to combine the JPEG/JPEG2000 compression/decompression as well as some coding/decoding methods.

Before we start, I would like to present a little bit the main workflow of our project

- It has 2 main phases / process: Compression – Coding , Decompression – Decoding (naturally, these 2 process are opposite)
- For the "Compression – Coding", firstly, the input image (extension .jpg, .jpeg, *.png, *.tif, etc) is "pushed" into "our model" to generate a compressed image. This compressed image will be saved into a file (I think binary file is a good choice – I applied this type in the last TP to store the compressed image)
- The "Decompression – Decoding" process will try to rebuild the original image from the compressed image (binary file). Of course, there are a missing rate between of 2 process (I will try calculate it)

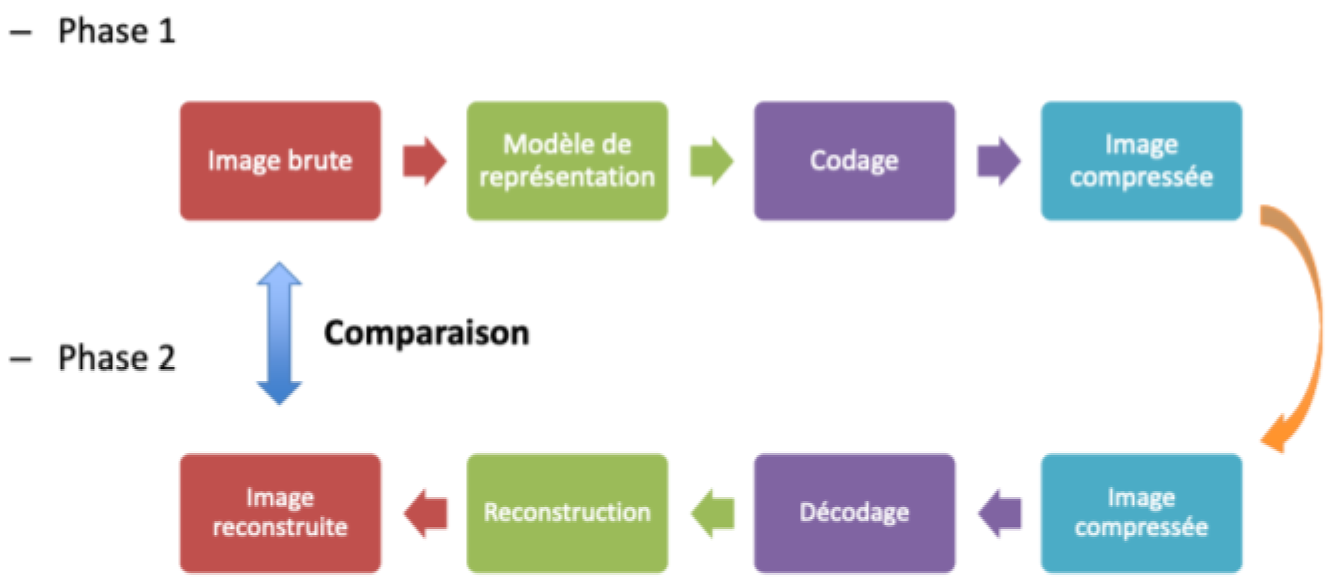



FIGURE 1 – Schéma global du projet

Additionally, I also compress/decompress with many cases of scalability (spatial scalability and quality of scalability) to evaluate our model. For example, for spatial scalability, I will chose different values (QCIF, CIF, 4CIF, etc); for quality of scalability, I also choose different values of noise (to generate an image with different noises, I will use Gaussian noise).

Lastly, I also detect ROI (Region of Interested) by applying the Trial-and-Error method to detect a set of "ideal" values (of course, for each image, a set of "ideal" values is different, and I will detect it manually).

Lastly, I will evaluate the performance of model through different rate (e.g: entropy, compression ratio, compression curve, histogram, etc).

# Part 2 : Preliminary of version 1

For the TP of third weeks of Codage d'Information, I implemented many steps/methods that I think they are necessary for this version (Huffman and RLE compression/decompression image, Store encoded image into a binary file, etc). Therefore, for this version, I will reuse them to resolve the main goal, as well as provide some new methods (that I think they are important). If I have time, I will present the improvement of my implementation of last TP (my implementation is not really effective when applied to RGB images, I will try to fix it by using OpenCV).

To implement this version, I use the programming language Python version 3 (Python3). In addition, to well manage and install relevant packages/libraries, I used Pip version 3 (pip3), so you can install by using `sudo apt-get install python3-pip`. Some packages/libraries I used in this project are:

- `Pillow`
- `Scipy` (I used the version 1.1.0 to run and test something in this project)
- `Numpy`
- `Matplotlib`
- `Skimage` (you have to install scikit-image instead of skimage)
- `Pandas`
- `Math`
- `Struct`
- `Os`

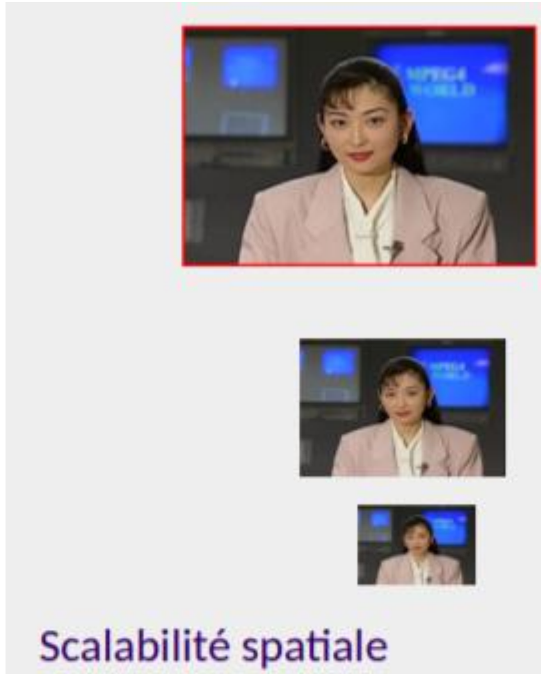
Some library can not be available in Windows, so my advice is to code in Ubuntu (I used the version 18.04).

Moreover, I also reuse some functions defined in JPEG and JPEG2000 packages (provided by professor Renaud Peteri ©).

# Part 3 : Implementation of version 1

Before we start, I would like to present Spatial Scalability and Quality Scalability as well as some results that I obtained.

1. **Spatial Scalability** (Slide 81), can be simply understood as the frame size of an image (in this case, we just consider image) based on its height and width (pixels). Of course, there are many definitions/concepts of spatial scalability, but I use the most simple definition. There are many kinds/types of spatial scalability, e.g: SQCIF, QCIF, SIF(525), CIF/SIF(625), 4SIF(525), 4CIF/4SIF(625), 16CIF.




2. **Quality Scalability** (Slide 81), can be simply understood as the quality of an image based on SNR. That means the quality of an image is good or bad, it depends on the noise level. There are many types/kinds of noise, e.g: Gaussian, Salt-Pepper, Speckle, etc)

So, for spatial scalability, I will build the function `spatialScalable()` with 4 parameters:

- `img` : the input image
- `smallFrame` : the "low" spatial level (I prefer to use "small frame")
- `mediumFrame` : the medium spatial level (I prefer to use "medium frame")
- `largeFrame` : the "high" spatial level (I prefer to use "large frame"

Although the frame size of each level can be customized, I prefer to use QCIF (176 x 144)) for smallFrame, CIF (352 x 288) for mediumFrame, and 4CIF (704 x 576) for largeFrame.

```
def spatialScalable(img, smallFrame, mediumFrame, largeFrame):
    image = Image.open(img) #open the input image

    # scaling
    smallImage  = image.resize(smallFrame)      # generate small image
    mediumImage = image.resize(mediumFrame)     # generate medium image
    largeImage  = image.resize(largeFrame)      # generate large image

    # store in numpy array
    np.save("smallImage.npy", np.array(smallImage, dtype = np.float32))
    np.save("mediumImage.npy", np.array(mediumImage, dtype = np.float32))
    np.save("largeImage.npy", np.array(largeImage, dtype = np.float32))

    # return a dict
    return {"small":np.array(smallImage, dtype = np.float32),
                "medium": np.array(mediumImage, dtype = np.float32),
                    "high": np.array(largeImage, dtype = np.float32)}
```

The body of this method have 4 steps:

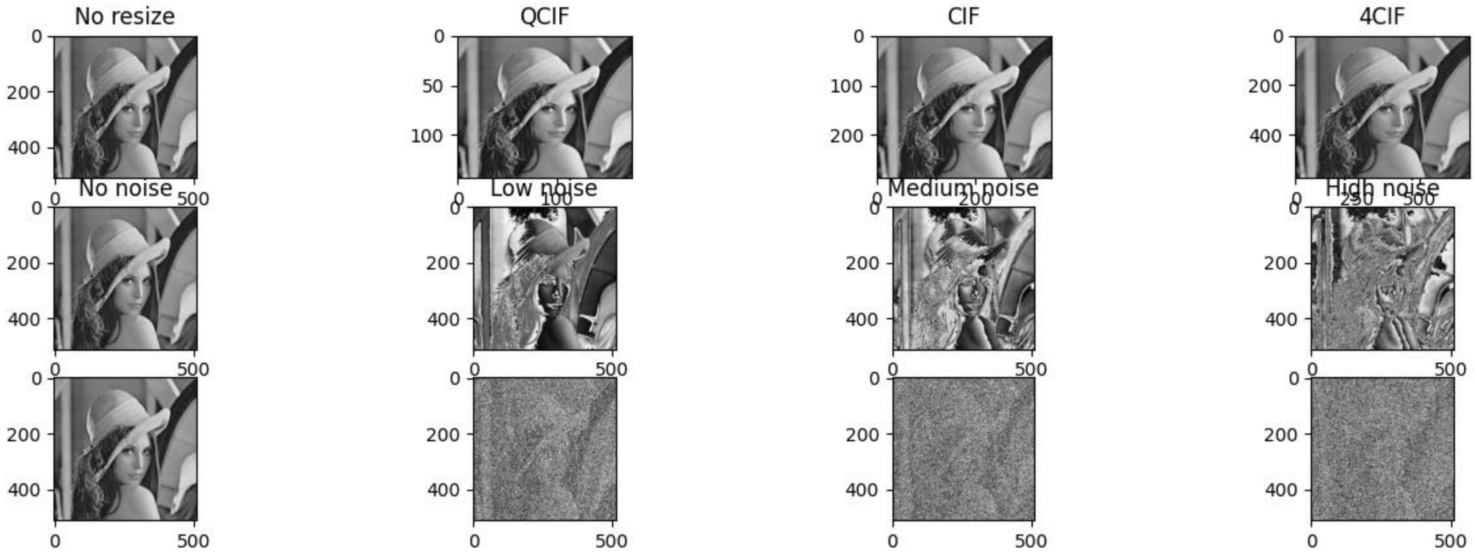
1. Load an input image by using `Image.open()`
2. Resize the image by use `resize()`
3. Store the output into a numpy array (npy file). Instead of store output image into JPG/JPEG/PNG, I decide to store into numpy array to easy reload later to calculate some relevant informations
4. Lastly, return a dict of results (I prefer to return a dict instead of simple value)

Next, for the quality scalability, I will add the noise into the input image. I will show how I did with 2 approaches (I called "the simple way" and "the complex way"). These two methods are `.simpleQualityScalable()` and `.qualityScalable()`, each functions has 4 parameters: img, lowNoise, mediumNoise, highNoise

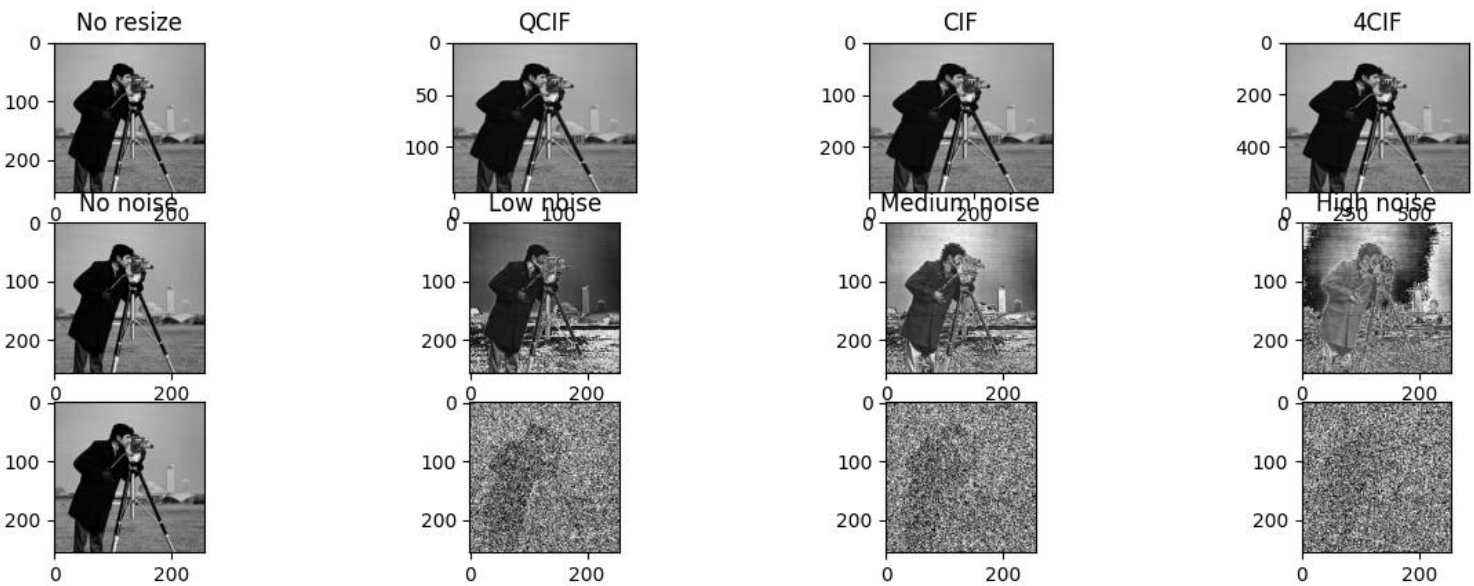
```
def simpleQualityScalable(img, lowNoise, mediumNoise, highNoise):
    image = misc.imread(img, mode='L')

    #add noise by piling many image
    low_image    = image * lowNoise
    medium_image = image * mediumNoise
    high_image   = image * highNoise

    return {"low": low_image,
                "medium": medium_image,
                    "high": high_image}

def qualityScalable(img, lowNoise, mediumNoise, highNoise):
    image = Image.open(img)
    image = np.array(image)

    # add gaussian noise (3 levels)
    noise_low_gaussian    = util.random_noise(image,mode="gaussian", var=lowNoise)
    noise_medium_gaussian = util.random_noise(image,mode="gaussian", var=mediumNoise)
    noise_high_gaussian   = util.random_noise(image,mode="gaussian", var=highNoise)

    return {"low":noise_low_gaussian,
                "medium": noise_medium_gaussian,
                    "high": noise_high_gaussian}
```

Now, let's make a simple test (I wrote the main function to test). The results will be shown by using subplots

```
if __name__ == '__main__':
    image = "Lena.jpg"

    qCif = (176, 144)
    Cif  = (352, 288)
    fCif = (704, 576)

    spatialDict = spatialScalable(image, qCif, Cif, fCif)
    simpleQualityDict = simpleQualityScalable(image, 2, 4, 8)
    qualityDict = qualityScalable(image, 2, 4, 8)


    # print(np.load("smallImage.npy"))
    # 3 lines, 4 columns
    f, axarr = plt.subplots(3, 4)

    # load spatial
    axarr[0,0].imshow(Image.open(image), cmap=plt.get_cmap('gray'))
    axarr[0,0].set_title("No resize")
    axarr[0,1].imshow(spatialDict.get("small"), cmap=plt.get_cmap('gray'))
    axarr[0,1].set_title("QCIF")
    axarr[0,2].imshow(spatialDict.get("medium"), cmap=plt.get_cmap('gray'))
    axarr[0,2].set_title("CIF")
    axarr[0,3].imshow(spatialDict.get("high"), cmap=plt.get_cmap('gray'))
    axarr[0,3].set_title("4CIF")

    # load simple quality
    axarr[1,0].imshow(Image.open(image), cmap=plt.get_cmap('gray'))
    axarr[1,0].set_title("No noise")
    axarr[1,1].imshow(simpleQualityDict.get("low"), cmap=plt.get_cmap('gray'))
    axarr[1,1].set_title("Low noise")
    axarr[1,2].imshow(simpleQualityDict.get("medium"), cmap=plt.get_cmap('gray'))
    axarr[1,2].set_title("Medium noise")
    axarr[1,3].imshow(simpleQualityDict.get("high"), cmap=plt.get_cmap('gray'))
    axarr[1,3].set_title("High noise")

    # load gaussian noise
    axarr[2,0].imshow(Image.open(image), cmap=plt.get_cmap('gray'))
    axarr[2,1].imshow(qualityDict.get("low"), cmap=plt.get_cmap('gray'))
    axarr[2,2].imshow(qualityDict.get("medium"), cmap=plt.get_cmap('gray'))
    axarr[2,3].imshow(qualityDict.get("high"), cmap=plt.get_cmap('gray'))

    plt.show()
```

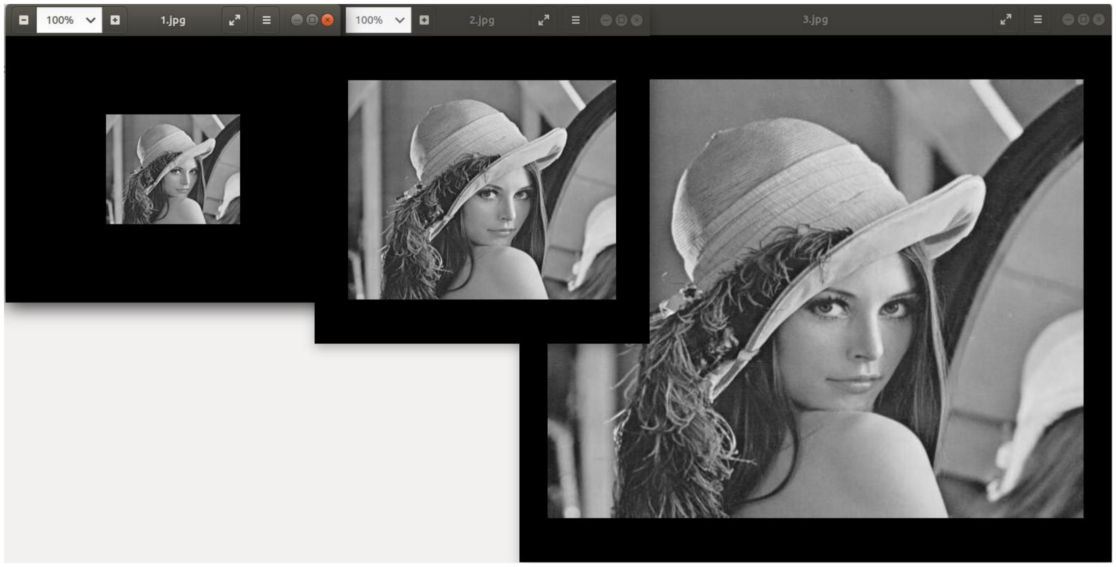



First line : Spatial Scalability Coding / Second line : Simple noise adding / Third line : Gaussian noise adding




First line : Spatial Scalability Coding / Second line : Simple noise adding / Third line : Gaussian noise adding

3 levels of spatial scalability (Lena.jpg and cameraman.tif)




Now, we will come to the main implementation of this project, compression-coding/decompression-decoding of image. I decided to choose Huffman algorithm and Run-Length Encoding algorithm for one-dimension (1D) for this project.

Before implementing these 2 algorithms, I implement 2 methods to save (`saveEncodedImage`) and load file (`loadEncodedImage`)

```
### Store encoded image in binary file (I also store encoded image into numpy array > np.array)
### encode_image is a binary string (I will show you) >>> binary file
def saveEncodedImage(file, string):
    with open(file,'wb') as f:
        for k in range(0,len(string),8):
            f.write(struct.pack('c',bytes([int(string[k:k+8],2)])))
        f.write(struct.pack('c',bytes([len(string)-(int(len(string)/8)*8)]))) # convert infos to byte >> write to bin file

### Load encoded image from binary file to rebuild (I also load encoded image from numpy array >> np.load)
### decoded_image is a binary string
def loadEncodedImage(file):
    with open(file,'rb') as f:
        Outstring = [bitMapping(int(x), 255) for x in f.read(os.path.getsize(file))] #convert a string of bit -> raw infos
        string2 = ""
        for x in range(len(Outstring)-2):
            string2 = string2 + Outstring[x]
        remainString = Outstring[len(Outstring)-2][(8-int(Outstring[(len(Outstring)-1):][0],2)):]
        Outstring = string2 + remainString
        return Outstring  # return raw infos in form of string
```

In addition, I also built 2 functions bitMapping and reverseBitMapping.

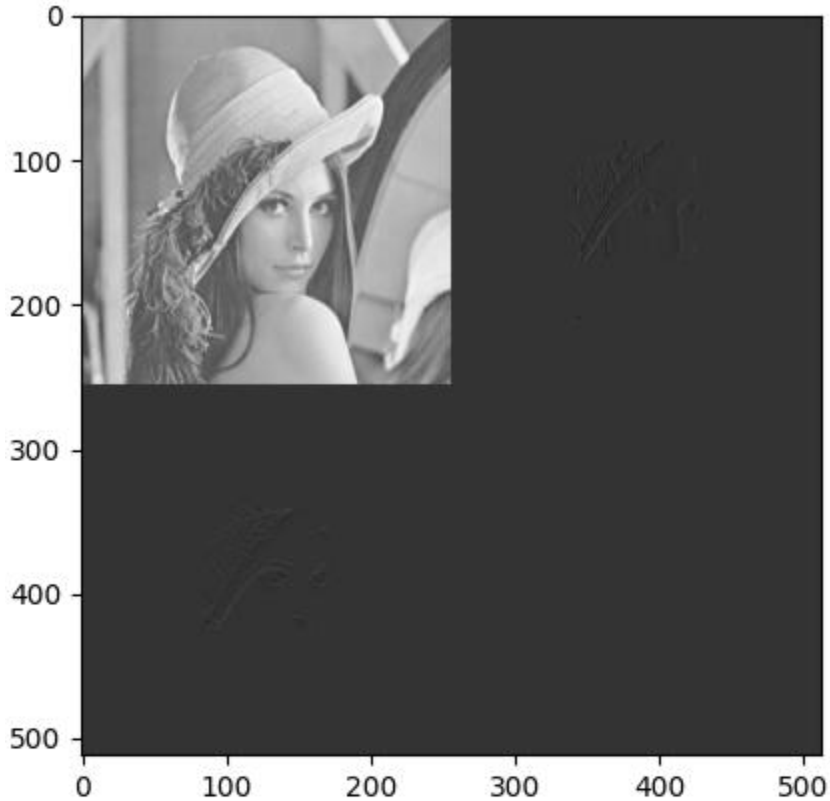
1. `bitMapping` convert a number of bits to a binary string (e.g bitMapping(10, 20) returns 01010)
2. `reverseBitMapping` return an integer corresponding a byte (e.g: 101010101010 = 2730)

```
### convert a number of bits -> a binary string, e.g bitMapping(10, 20) -> 01010
def bitMapping(bitNumber, bitsMax):
    if(bitsMax < bitNumber):
        print("bitsMax to encode < bitNumbers >> Error to encoded")
    val = "{}".format(bitNumber)
    valMax = "{}".format(bitsMax)
    maxNum = "{}".format(bin(int(valMax,10))[2:])
    maxFormat = "{}".format(bin(int(val,10))[2:])
    bitNumber = ""
    for i in range(len(maxFormat),len(maxNum)):
        bitNumber = bitNumber + "0"
    bitNumber += maxFormat

    return bitNumber

### convert a string bit (encoded) -> a integer
def reverseBitMapping(bitNumber):
    return int(str(bitNumber),2)
```

Subsequently, I will split an image into 4 **quadrants** (decompose), each quadrant is transformed into numpy array

```
### fourQuadrantSplit original image -> 4 quadrants, each quadrant is transformed into a numpy array
### return a dict
def fourQuadrantSplit(image):
    image1 = np.array([image[x][:int(len(image)/2)] for x in range(int(len(image)/2))]).astype(int)
    image3 = np.array([image[x][int(len(image)/2):] for x in range(int(len(image)/2))]).astype(int)
    image2 = np.array([image[x][int(len(image)/2):] for x in range(int(len(image)/2), len(image))]).astype(int)
    image4 = np.array([image[x][:int(len(image)/2)] for x in range(int(len(image)/2), len(image))]).astype(int)
    return {"first": image1, "second": image2,"third": image3, "fourth":image4}
```

Relying in the function `.decomposition` (in JPEG2000), I rebuild a same function (of course, I modify a little bit) `.fourQuandrantSplit()`




4 quadrants of an image

To implement Huffman algorithm, I will setup 4 methods as belows:

1. `huffmanTreeInsert(list, element)` return a new list after inserting a new element

```
### insert an element into huffman tree
#### input : a list + a element => return a list after inserting
def huffmanTreeInsert(list, element):
    rang = 0
    while(rang < len(list) and (element[0] > list[rang][0]) ):
        rang += 1
    list.insert(rang, element)
    return list
```

2. `buildHTree(occursIndexList, occursValList)` return a Huffman tree (type: List)

```
### build a huffman tree ==> print the tree
##### return a tree
def buildHTree(occursIndexList, occursValList):
    huffmanTree = [[y,x] for x,y in zip(occursIndexList,occursValList)]

    # build tree: in a Huffman tree, left side < right side (always)
    while(len(huffmanTree) != 1):
        if(huffmanTree[0][0] < huffmanTree[1][0]):  # left side < right side
            leftSide = huffmanTree[0]
            rightSide = huffmanTree[1]
        else:                                        # swap left and right
            leftSide = huffmanTree[1]
            rightSide = huffmanTree[0]
        huffmanTree.pop(0)          # pop first element
        huffmanTree.pop(0)          # next, pop the second element
        huffmanTree = huffmanTreeInsert(huffmanTree, [leftSide[0]+rightSide[0], leftSide, rightSide])
    print(huffmanTree)              # print huffman tree, e.g: [342 - root, [170 - root, [82, 367] - left side, [88, 360] - right side]
    return huffmanTree[0]
```

3. `huffmanTreeFinding(huffmanTree, pix` (it is pixel), `string)` finds a pixel in a string into Huffman tree. This function return the result or NIL

```
#### this method will return a boolean value after finding a pix in a string into a huffman tree
def huffmanTreeFinding(huffmanTree, pix, string):
    if(len(huffmanTree) > 0):
        if(len(huffmanTree) > 1 and not(isinstance(huffmanTree[1], list))):     # a leaf
            if(huffmanTree[1] == pix):
                return string
        else:   # 1 child
            _result1 = huffmanTreeFinding(huffmanTree[1], pix, string + "0")
            if(_result1 != "nil"):
                return _result1         # empty
            if(len(huffmanTree) > 2):   # 2 childs
                _result1 = huffmanTreeFinding(huffmanTree[2], pix, string + "1")
                if(_result1 != "nil"):
                    return _result1
    else:   # len(huffmanTree <= 0)
        print("Tree invalid")
    return "nil"
```

4. `HuffmanEncode(image, inputSize)` returns an encoded image (the encoded result is a string). This function have 2 parameters: image (input image) and inputSize (how many bit you want to encode? In this case, I will hardcode 16000000). The output of Huffman encode is a binary string (0,1) and the size of this string

```python
###### Huffman encoding ############
###### return an encoded image #######
def HuffmanEncode(image, inputSize):
    beginBitNumber = len(bitMapping(inputSize,inputSize))
    reshapedImg = np.reshape(image, (-1,len(image)*len(image)))[0] # reshape input image
    FramedImg = pd.DataFrame(reshapedImg)   # transformr 2D size-mutable (tabular data structure)

    occursList = FramedImg[0].value_counts() # len of tabular (horizontal)

    occursValList = occursList.values.tolist()
    occursIndexList = occursList.index.tolist()

    huffmanTree = buildHTree(occursIndexList, occursValList)    #build a huffman tree using index list and value list

    indexMax = max(occursIndexList)
    maxVal = max(occursValList)
    indexBitNumber = len(bitMapping(indexMax, indexMax))
    valueBitNumber = len(bitMapping(maxVal, maxVal))

    finalString = bitMapping(indexBitNumber, inputSize)
    finalString += bitMapping(valueBitNumber, inputSize)
    finalString += bitMapping(beginBitNumber*3+(indexBitNumber+valueBitNumber)*len(occursIndexList), inputSize)

    for rang in range(len(occursIndexList)):
        finalString += bitMapping(occursIndexList[rang], indexMax)
        finalString += bitMapping(occursValList[rang], maxVal)

    res = [huffmanTreeFinding(huffmanTree, reshapedImg[x], "") for x in range(len(reshapedImg)) ]
    _result = "".join(map(str, res))
    _result = finalString + _result

    return _result  # result is a string -> store to bin file
```

5. `HuffmanDecode(_result, inputSize)` returns a decoded image (the decode result is an array, and it will be load by using np.array). This function also have 2 parameters: image (input image) and inputSize (how many bit you want to decode?). The output of this function is a list of integer value.

```python
###### Huffman decoding ########
########## return "original" image ###########
def HuffmanDecode(_result, inputSize):
    beginBitNumber = len(bitMapping(inputSize,inputSize))
    indexBit = reverseBitMapping(_result[:beginBitNumber])
    indexVal = reverseBitMapping(_result[beginBitNumber:beginBitNumber*2])
    minVal = reverseBitMapping(_result[beginBitNumber*2:beginBitNumber*3])

    occursIndexList = []
    occursValList = []
    for i in range(beginBitNumber*3, minVal,indexBit+indexVal):
        occursIndexList.append(reverseBitMapping(_result[i:i+indexBit]))
        occursValList.append(reverseBitMapping(_result[(i+indexBit):(i+indexBit+indexVal)]))

    huffmanTree = buildHTree(occursIndexList, occursValList)    #build a huffman tree using index list and value list

    image = []  # output is a list that I will use np.load() to load

    clonedHuffmanTree = huffmanTree     # clone a new huffman tree
    bit = minVal
    while (bit < len(_result)):
        if(len(clonedHuffmanTree) > 0):
            if(len(clonedHuffmanTree) > 1):
                if(not(isinstance(clonedHuffmanTree[1], list)) ) :  # if fist element of huffman tree is not a list
                    image.append(clonedHuffmanTree[1])               # make a new list by appending image (list) this first element -> return image (list)
                    clonedHuffmanTree = huffmanTree                  # re-assign cloned tree by current tree -> if skip this line, the memory leaking by recursive
                else:
                    if(_result[bit] == "0"):
                        clonedHuffmanTree = clonedHuffmanTree[1]
                        bit += 1
                    else:
                        if(len(clonedHuffmanTree) > 2 and _result[bit] == "1"):
                            clonedHuffmanTree = clonedHuffmanTree[2]
                            bit += 1
    if(not(isinstance(clonedHuffmanTree[1], list)) ) :
        image.append(clonedHuffmanTree[1])

    return image # image is a list of integer value -> loaded with np.array
```

Now, the Huffman algorithm (3 activities: finding, building, inserting and 2 phases : encoding, decoding) is finished. We will come to Run-Length Encoding (RLE) algorithm, in this project, I will use one-dimensional (1D) approach with 2 methods: HorizontalEncode() and HorizontalDecode(). Actually, RLE algorithm is similar to Huffman, so its implementation is quite simple.

1. `HorizontalEncode(image, inputSize)` returns an encoded image (the encoded result is a string). This function have 2 parameters: image (input image) and inputSize (how many bit you want to encode? In this case, I will hardcode 16000000). The output of RLE encode is a binary string (0,1) and the size of this string

```python
############ RLE algorithm is builded from Huffman algorithm ===> that means that I will modify Huffman algorithm above to implement RLE algorithm ###########
############ Horizontal Encode ###########
############## like Huffman Encode, this method will return an encoded image in form of a binary string ############
def HorizontalEncode(image, inputSize):
    image = np.reshape(image, (-1,len(image)*len(image)))[0]
    nbFact = 0
    pixList = []
    orderList = []
    i = 0
    while (i < len(image)):
        count = i
        while (i < len(image) - 1 and image[i] == image[i + 1]):
            i += 1
        pixList.append(int(image[i]))
        orderList.append(i-count+1)
        nbFact += i-count+1
        i += 1
    minVal = -min(pixList)
    pixList = [i+minVal for i in pixList]

    i = 0
    maxVal = 10*int(mean(orderList))
    while(i < len(pixList)):
        if(orderList[i] > maxVal):
            orderList.insert(i+1, orderList[i]-maxVal)
            orderList[i] = maxVal
            pixList.insert(i+1, pixList[i])
        i += 1
    beginBitNumber = len(bitMapping(inputSize,inputSize))
    indexMax = max(orderList)
    maxVal = max(pixList)
    indexBitNumber = len(bitMapping(indexMax, indexMax))
    valueBitNumber = len(bitMapping(maxVal, maxVal))
    stringFinale = bitMapping(indexBitNumber, inputSize)
    stringFinale += bitMapping(valueBitNumber, inputSize)
    stringFinale += bitMapping(minVal, inputSize)

    for j in range(len(orderList)):
        stringFinale += bitMapping(orderList[j], indexMax)
        stringFinale += bitMapping(pixList[j], maxVal)
    return stringFinale     # output is a binary string -> use np.load() to load
```

2. `RLEDecode(_result, inputSize)` returns a decoded image (the decode result is an array, and it will be load by using np.array). This function also have 2 parameters: image (input image) and inputSize (how many bit you want to decode?). The output of this function is a list of integer value.

```python
############ Horizontal Decode ###########
############## like Huffman Decode, this method will return a decoded image in form of list ############
def HorizontalDecode(_result, inputSize):
    beginBitNumber = len(bitMapping(inputSize,inputSize))
    indexBit = reverseBitMapping(_result[:beginBitNumber])
    indexVal = reverseBitMapping(_result[beginBitNumber:beginBitNumber*2])
    minVal = reverseBitMapping(_result[beginBitNumber*2:beginBitNumber*3])

    orderList = []
    pixList = []
    for i in range(beginBitNumber*3, len(_result),indexBit+indexVal):
        orderList.append(int(reverseBitMapping(_result[i:i+indexBit])))
        pixList.append(int(reverseBitMapping(_result[(i+indexBit):(i+indexBit+indexVal)]))-minVal)
    image = []
    for i in range(len(orderList)):
        for count in range(orderList[i]):
            image.append(pixList[i])

    return image    # output is a list of integer
```

Lastly, I would like to present how to implement Wavelet Transform. To implement it, I relied the section of "Seuillage des Coefficient d'Ondelettes" of JPEG2000. From that, I rebuild 4 methods as follows:

1. `applyContourByBias(image, discretZone, deathZone)` aims to apply the contour with a bias (bias = seuil) to an image

```
def applyContourByBias(image, discretZone, deathZone):
    contour = np.array(image[int(image.shape[0]/3):2*int(image.shape[0]/3),int(image.shape[0]/3):2*int(image.shape[0]/3)])
    for rangX in range(len(image)):
        for rangY in range(len(image[0])):
            if(image[rangX][rangY] != 0.0):
                image[rangX][rangY] = int(image[rangX][rangY]/discretZone)*discretZone
    image = np.where(np.abs(image) > deathZone, image, 0.0)
    image[int(image.shape[0]/3):2*int(image.shape[0]/3),int(image.shape[0]/3):2*int(image.shape[0]/3)] = contour
    return image # return an image applied by contour with bias
```

2. `biasDecomposing(img, LO_D, HI_D, compteur, deathZone, discretZone)` aims to decompose an image which is applied Wavelet Transform. This method is quite similar to the function .decomposer() (predefined in JPEG2000)

```
def biasDecomposing(img, LO_D, HI_D, compteur, deathZone, discretZone):
    compteur -= 1
    if(compteur == 0):
        a1 = jp2.decompose(LO_D, LO_D, img)
    else:
        a1 = biasDecomposing(jp2.decompose(LO_D, LO_D, img), LO_D, HI_D, compteur, deathZone, discretZone)
    im_out=np.double(np.zeros(img.shape))
    d1 = jp2.decompose(HI_D, HI_D, img)          # similar to decompose() -> find depth
    h1 = jp2.decompose(LO_D, HI_D, img)          # similar to decompose() -> find horizontal
    v1= jp2.decompose(HI_D, LO_D, img)           # similar to decompose() -> find vertical

    h1 = applyContourByBias(h1, discretZone, deathZone)     # apply contour by bias on h1 to find horizontal
    v1 = applyContourByBias(v1, discretZone, deathZone)     # apply contour by bias on v1 to find vertical
    d1 = applyContourByBias(d1, discretZone, deathZone)     # apply contour by bias on d1 to find depth

    im_out[0:a1.shape[0],0:a1.shape[1]] = a1
    im_out[0:a1.shape[0],a1.shape[0]:2*a1.shape[0]] = h1
    im_out[a1.shape[0]:2*a1.shape[0],0:a1.shape[1]] = v1
    im_out[a1.shape[0]:2*a1.shape[0],a1.shape[0]:2*a1.shape[0]] = d1
    return im_out # im_out is a string -> but you can store the returned value by array -> return [a1, h1, v1, d1]
```

3. `decomposeImageWithWavelet(cpt,cheminImage,deathZone,discretZone)` aims to load an input image with Wavelet Transform. This function returns a decomposed image (e.g: quadrant) transformed by Wavelet Transform. This function has 3 steps:
    a. Load an image and store in "img" variable
    b. Apply function of (2) to decompose an image to many sub-images
    c. Return a decompose of image

```
##### decompose Image with Wavelet -> load input image + decompose #####

def decomposeImageWithWavelet(cpt, cheminImage, deathZone, discretZone):
    img = np.double(plt.imread(cheminImage))
    img = biasDecomposing(img, LO_D, HI_D, cpt, deathZone,discretZone)
    return img # return a decomposed image -> many sub-images (quadrants)
```

4. `recomposeImageWithoutWavelet(cpt,compressedImage)` aims to recompose a decomposed image (e.g: 4 quadrants >> 1 image) without Wavelet Transform. This function returns a recomposed image (full image) without Wavelet Transform. This function has 2 steps:
    a. Recompose compressed image (image after decoding). *Image after decoding is the form of "quadrant". This step will "join" all to only one image
    b. Return the "original" image (we can use np.load to load it)

```
##### recompose Image without Wavelet #####

def recomposeImageWithoutWavelet(cpt, compressedImage):
    img = recomposer(compressedImage, LO_R, HI_R, cpt)
    return img # return the recomposed image
#
```

Before to implement the test / the deployment, I would like to present some ways / criteria to evaluate the model. In this project, I used entropy (before encoding and after encoding) as well as compression ratio (before encoding and after encoding). Actually, you can use 1 in 2 these criteria, but I prefer to use both of them.

1. Firstly, to calculate entropy, I will import `scipy.stats` with the function `entropy` (e.g: `from scipy.stats import entropy`). I will calculate the entropy of image before and after encoding. Before to do that, I will store the decompressed image into numpy array (I save this array into npy file to load)

```
# 2 ways
# save rebuiltImaage -> jpg (using scipy.misc)
scipy.misc.imsave('outfile.jpg', np.array(rebuiltImg, dtype = np.float32))

# save rebuiltImage -> numpy array
np.save("rebuiltImage.npy", np.array(rebuiltImg, dtype = np.float32))
```

Next, I calculate the entropy after and before compressing. The way of calculation of both of them is similar (I only change the image after and before compressing). Lastly, I calculate the difference between these two entropies (to make sure, you can use ABS to eliminate all negative values). I will show you some results in the next part.

```
# --------------- Evaluation ----------------> Entropy (Before and After) --------------->
# entropy before encoding
imgReshaped = Image.open(image)
X = imgReshaped.size[0]
Y = imgReshaped.size[0]
imag = np.array(imgReshaped.getdata()).reshape(X,Y)

sumDCT = np.array([np.zeros(8) for x in range(8)])
for x in range(0,X,8):
    for y in range(0,Y,8):
        sumDCT+=abs(jpeg.DCT(imag[x:x+8,y:y+8]))
sumDCT = sumDCT / np.sum(sumDCT)
print("Entropy before encoding", entropy(np.reshape(sumDCT, (64))))

# entropy after encoding

# load by Image.open()
#imgAfter = Image.open("outfile.jpg")

# load from numpy array
imgAfter = Image.fromarray(np.load("rebuiltImage.npy"))
XAfter = imgAfter.size[0]
YAfter = imgAfter.size[1]
imagAfter = np.array(imgAfter.getdata()).reshape(XAfter,YAfter)

sumDCTAfter = np.array([np.zeros(8) for x in range(8)])
for x in range(0,XAfter,8):
    for y in range(0,YAfter,8):
        sumDCTAfter+=abs(jpeg.DCT(imagAfter[x:x+8,y:y+8]))
sumDCTAfter = sumDCTAfter / np.sum(sumDCTAfter)
print("Entropy after encoding", entropy(np.reshape(sumDCTAfter, (64))))

print("Entropy BEFORE - Entropy AFTER", entropy(np.reshape(sumDCT, (64)))-entropy(np.reshape(sumDCTAfter, (64))))
```

2. Secondly, I also calculate compression ratio with the formula `Ratio = Before_Compressing / After_Compressing`

The compression ratio is quite simple, I just apply the simple formula Ration = Size_Before / Size_After (with Size is the file size of two images after and before compressing)

```
# --------------- Evaluation ----------------> Compression rate (Before and After) --------------->
# image size before compressiong
originalSize = os.path.getsize("Lena.jpg")

# image size before decompressing
afterSize = os.path.getsize("outfile.jpg")

# compression rate (%)
print(afterSize/originalSize * 100)
```

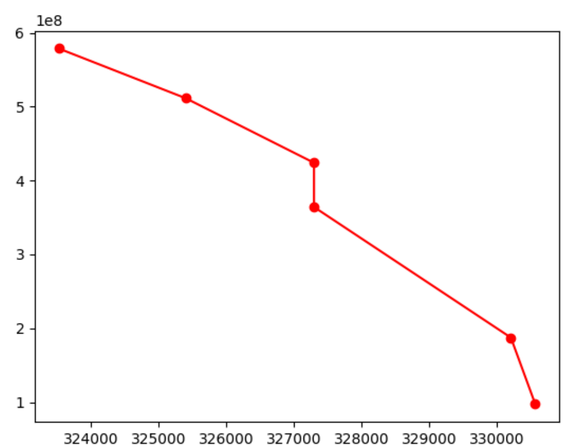
3. Lastly, I will give the compression curve for different bias (e.g: I choose 20, 50, 100, 200, 500, 1000). Since the curve is made from `matplotlib.pyplot`, it requires to calculate the plot size (Axis X, Axis Y). Hence, I have to modify a little bit the function `decomposeImageWithWavelet()` and `biasDecomposing()` by adding the size parameter (as well as return its value).

```
def biasDecomposing(img, LO_D, HI_D, compteur, deathZone, discretZone, size, energie):
    compteur -= 1
    if(compteur == 0):
        a1 = jp2.decompose(LO_D, LO_D, img)
    else:
        a1, size, energie = biasDecomposing(jp2.decompose(LO_D, LO_D, img), LO_D, HI_D, compteur, deathZone, discretZone, size, energie)
        size += size
        energie += energie
    im_out=np.double(np.zeros(img.shape))
    d1 = jp2.decompose(HI_D, HI_D, img)        # similar to decompose() -> find depth
    h1 = jp2.decompose(LO_D, HI_D, img)        # similar to decompose() -> find horizontal
    v1= jp2.decompose(HI_D, LO_D, img)         # similar to decompose() -> find vertical

    h1 = applyContourByBias(h1, discretZone, deathZone)     # apply contour by bias on h1 to find horizontal
    v1 = applyContourByBias(v1, discretZone, deathZone)     # apply contour by bias on v1 to find vertical
    d1 = applyContourByBias(d1, discretZone, deathZone)     # apply contour by bias on d1 to find depth

    size += (h1.size - np.count_nonzero(h1) + v1.size - np.count_nonzero(v1) + d1.size - np.count_nonzero(d1))
    energie += ((h1**2).sum() + (v1**2).sum() + (d1**2).sum())

    im_out[0:a1.shape[0],0:a1.shape[1]] = a1
    im_out[0:a1.shape[0],a1.shape[0]:2*a1.shape[0]] = h1
    im_out[a1.shape[0]:2*a1.shape[0],0:a1.shape[1]] = v1
    im_out[a1.shape[0]:2*a1.shape[0],a1.shape[0]:2*a1.shape[0]] = d1
    return im_out, size, energie # im_out is a string -> but you can store the returned value by array -> return [a1, h1, v1, d1]
    # return energy to plot

##### decompose Image with Wavelet -> load input image + decompose #####

def decomposeImageWithWavelet(cpt, cheminImage, deathZone, discretZone):
    img = np.double(plt.imread(cheminImage))
    img, size, energie = biasDecomposing(img, LO_D, HI_D, cpt, deathZone, discretZone, 0, 0)

    return img,size,energie # return a decomposed image -> many sub-images (quadrants)
    # return plot, energy to plot
```

The implementation of compression curve will be:

```
# --------------- Evaluation ----------------> Compression courbe using subplot (Before and After) --------------->
# img, plot1,energie1 = tp2.decomposeImageWithWavelet(multipleQuadrantRecursive, image, 100, 20)
img, plot2,energie2 = tp2.decomposeImageWithWavelet(multipleQuadrantRecursive, image, 100, 50)
img, plot3,energie3 = tp2.decomposeImageWithWavelet(multipleQuadrantRecursive, image, 100, 100)
img, plot4,energie4 = tp2.decomposeImageWithWavelet(multipleQuadrantRecursive, image, 100, 200)
img, plot5,energie5 = tp2.decomposeImageWithWavelet(multipleQuadrantRecursive, image, 100, 500)
img, plot6,energie6 = tp2.decomposeImageWithWavelet(multipleQuadrantRecursive, image, 100, 1000)

plt.plot([plot,plot2,plot3,plot4,plot5,plot6], [energie,energie2,energie3,energie4,energie5,energie6], color="red", marker="o")
plt.show()
```

I will show you the curve of 6 levels (20, 50, 100, 200, 500, 1000) (of course, you can change the parameter to generate a new curve)

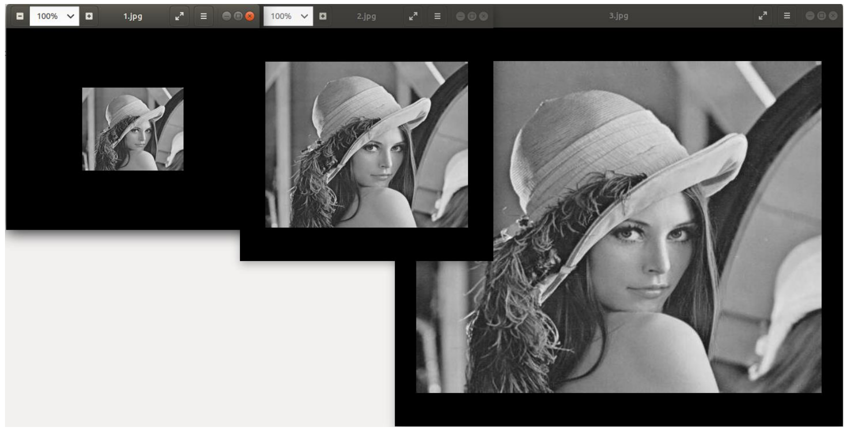



Compression curve

The results of ROI (Region of Interest) will be shown in the next part, the value of wavelet (that you chose) affects on the ROI (for example: I just detect the face of Lena)

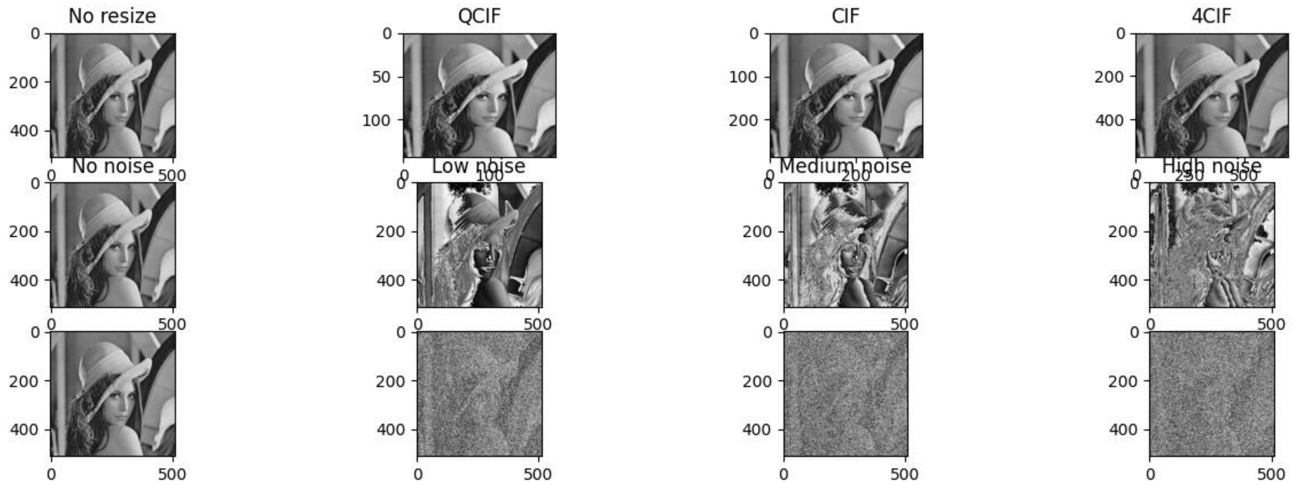
# Part 4 : Results of version 1

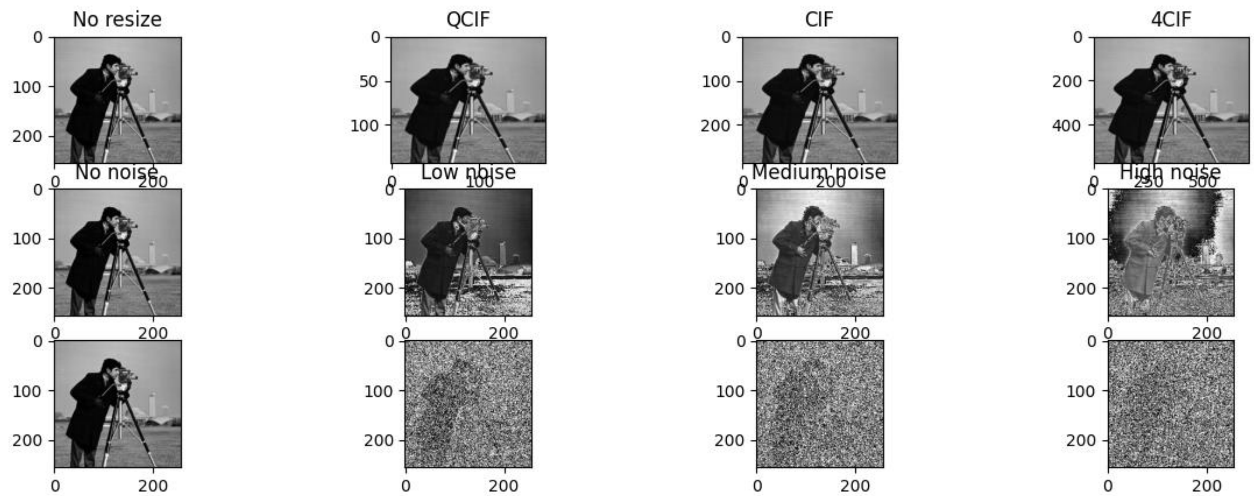
3 levels of spatial scalability (I make the test with 3 pictures: “Lena.jpg”, “cameraman.tif”, “barbara.jpg”)





3 levels of quality scalability (I make the test with 3 pictures: “Lena.jpg”, “cameraman.tif”, “barbara.jpg”)



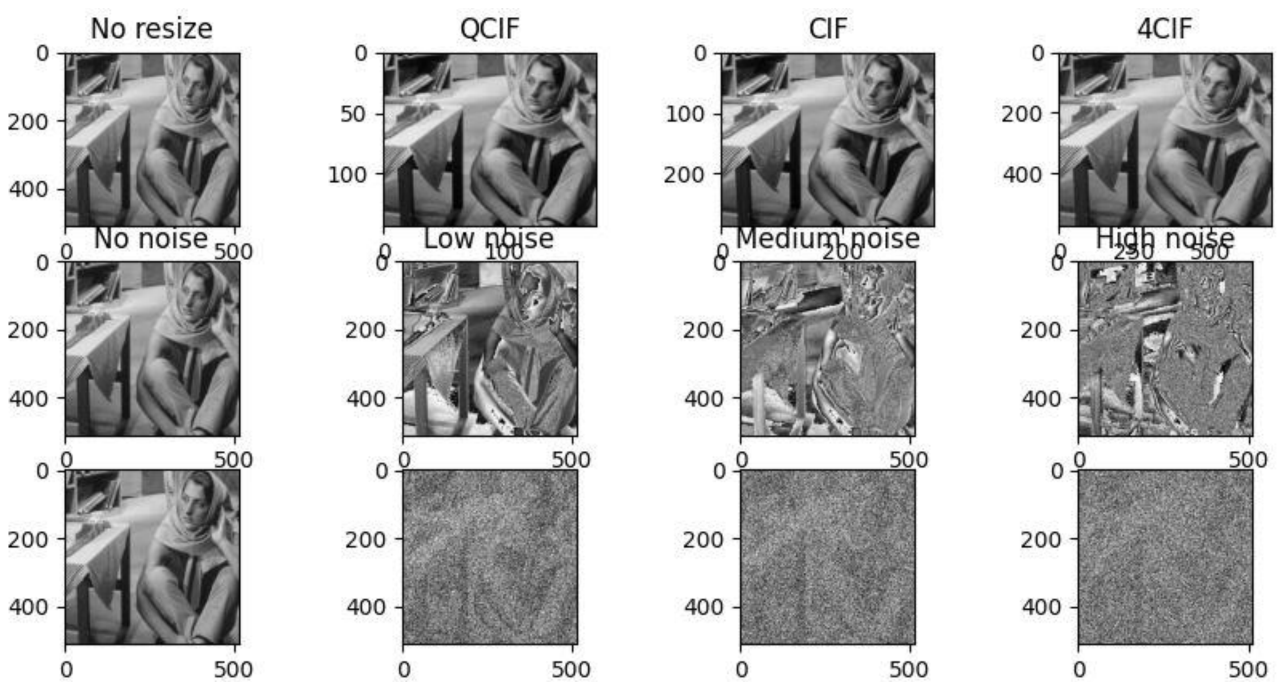
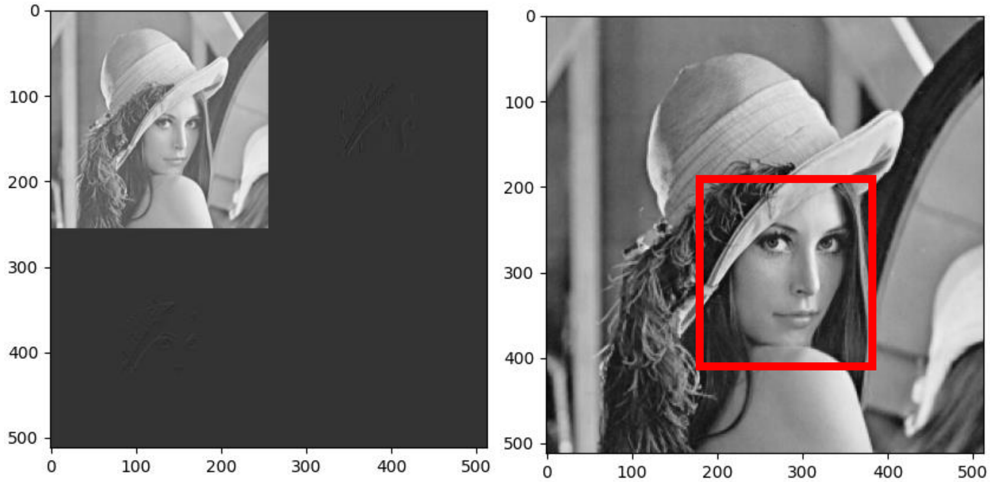
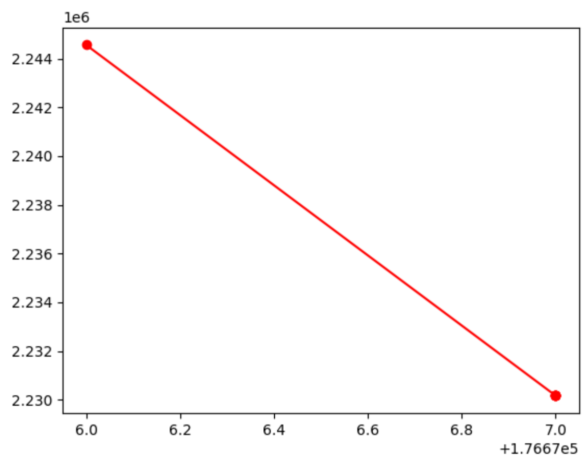



Image compression – decompression (no losing)

Firstly, I set the number of 4 quadrants recursive is 1, it means 4^1 = 4 (quadrants) et let's see what happens?



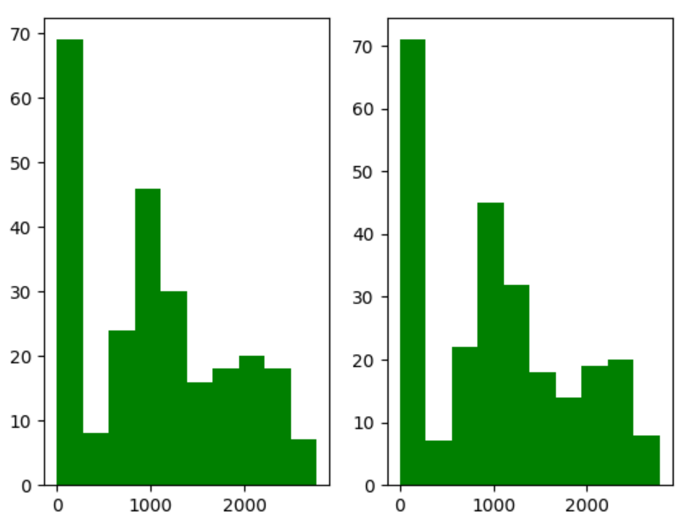

The decompressed image has the high quality 😊, and the red square is ROI (right image)




The compression curve is linear

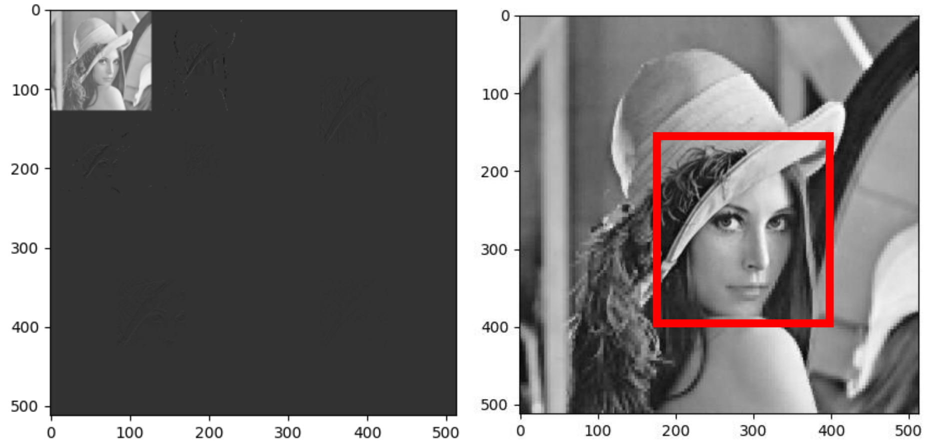
```
Entropy before encoding 1.140041037165215
Entropy after encoding 1.1883335138751765
Entropy BEFORE - Entropy AFTER -0.04829247670996151
compression ratio : 3.3409168822264377%
```

Entropy (before and after coding) as well as the compression ratio (3,34%)

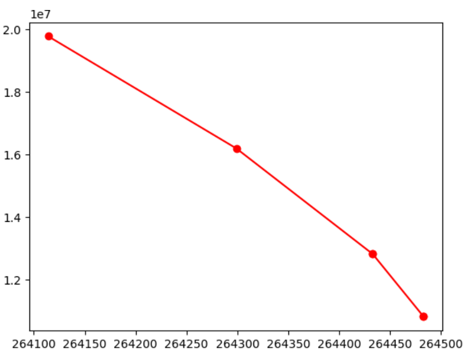



The histogram before (left) and after (right)

Now, I would like to change some parameter (coefficient) and to see what happens. Firstly, I changed the multipleQuandrantRecursive to 2 (we will get 4^2 = 16 quadrants). All parameters remaining are fixed.



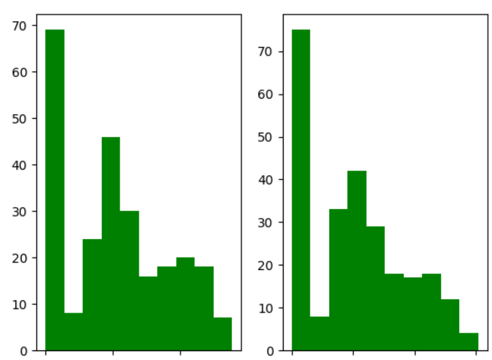

The decompressed image 😊, and the red square is ROI (right image)




The compression curve (quite linear, but not too many)

```
Entropy before encoding 1.140041037165215
Entropy after encoding 0.9209429218999929
Entropy BEFORE - Entropy AFTER 0.21909811526522205
compression ratio : 12.229251721881141%
```

Entropy (before and after coding) as well as the compression ratio (12,22%)




The histogram before (left) and after (right)

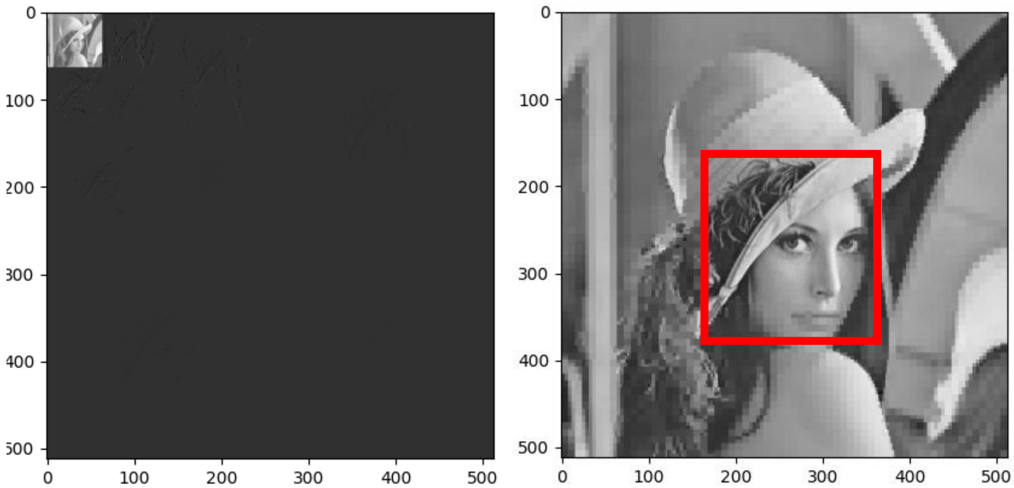
When I select multipleQuandrantRecursive = 1, the decoded image has a better quality than when I selected multipleQuandrantRecursive = 2. So multipleQuandrantRecursive affects to the quality of compression. Actually, this parameters shows how many compression times (when you compress an image many times, of course, its quality is not good).

What happens next? All parameters remaining are fixed.

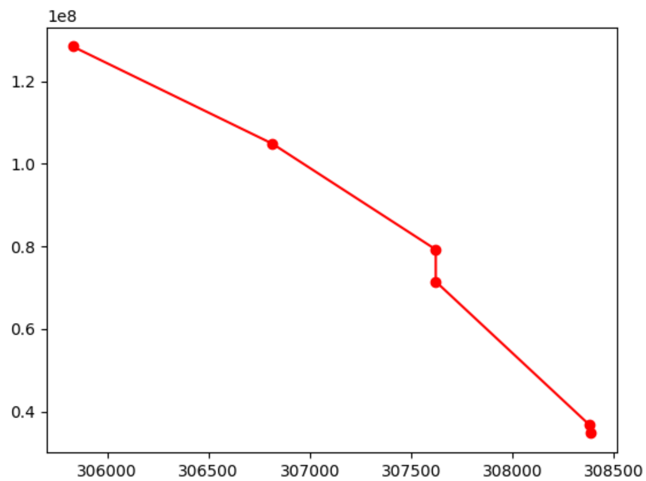
So, the compression curve which show the compression energy is linear. Moreover, the compression ratio is lower (3,34 % vs 12,22 %). So, when I decrease multipleQuandrantRecursive, the quality of decoded image is better, but the compression ration is decreased.

When I changed multipleQuandrantRecursive by 3 and 4, what happens next?

Case 1 : Value “3” => 4^3 = 64 quadrants



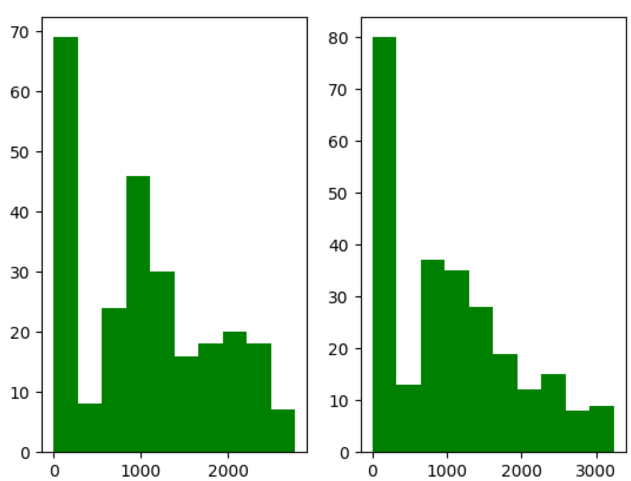

The decompressed image has the high quality 😊, and the red square is ROI (right image)




The compression curve

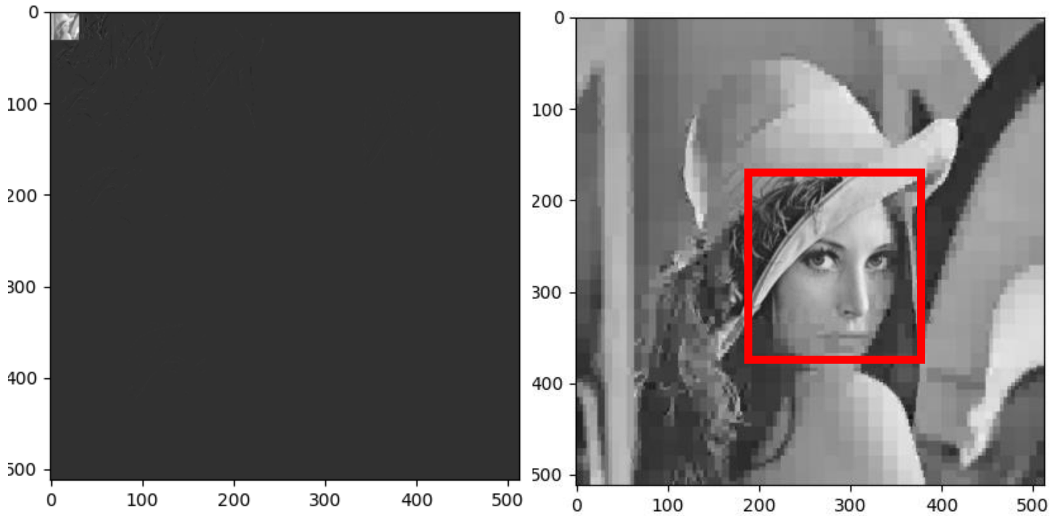
```
Entropy before encoding 1.140041037165215
Entropy after encoding 0.6040720369002065
Entropy BEFORE - Entropy AFTER 0.5359690002650085
compression ratio : 52.46361454794777%
```

The entropy before and after as well as the compression ratio (52,46%)

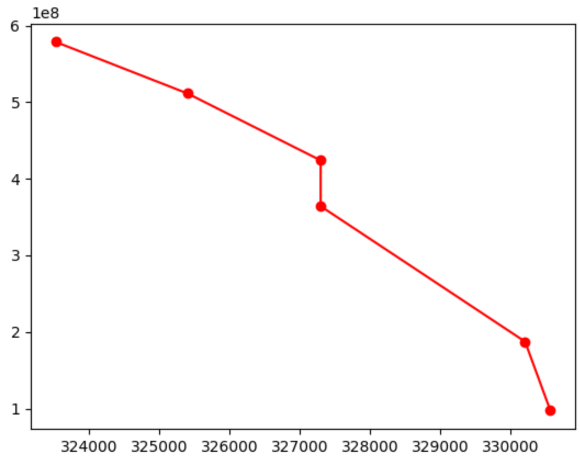



The histogram before (left) and after (right)

Case 2 : Value “4” => 4^4 = 256 quadrants



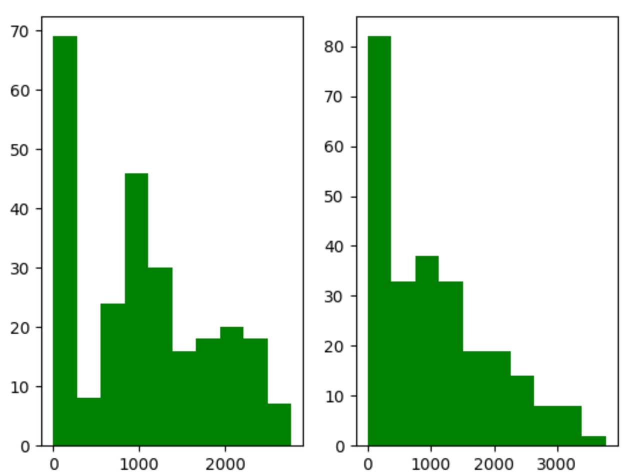

The decompressed image has the high quality 😊, and the red square is ROI (right image)




The compression curve

```
Entropy before encoding 1.140041037165215
Entropy after encoding 0.603903660171036
Entropy BEFORE - Entropy AFTER 0.536137376994179
compression ratio : 53.61672951662667%
```

The entropy before and after as well as the compression ratio (53,61%)




The histogram before (left) and after (right)

So, I can summary as follows

| multipleQuandrantRecursive | Number of quadrants (using ABS) | Compression ratio (%) |
|---|---|---|
| 0 | NULL/ERROR | NULL/ERROR |
| **1** | **4** | **3,34** |
| **2** | **16** | **12,22** |
| **3** | **64** | **52,46** |
| 4 | 256 | 53,61 |
| 5 | 1024 | 53,96 |
| 6 | 2048 | 53,97 |

For 3 and 4,5,6 and above, the compression ratio is quite similar (53,46% vs 53,96% vs 53,97%). So, I think 1,2,3 is a three ideal levels (1 is the high, 2 is the medium, 3 is the low).