ENSEA

Beyond Engineering

SIGNAL

Théorie de l'information et Compression Multimédia

Compte rendu de TP 1 & 2 : JPEG

2ème année | 2G1 TD3 TP5

Erwan BROUDIN

31 mars 2023

Préambule

Ce TP porte sur la compression d'image telle qu'elle existe avec le format JPEG. Avant de nous lancer dans le TP, nous avons donc besoin d'expliquer pourquoi cette compression est si importante.
Le JPEG permet de compresser une image de deux manière :

- Compression **sans perte** : cette compression (de l'ordre 2, pouvant parfois monter à 8) permet de compresser l'image, sans pour autant perdre d'informations sur celle -ci. Elle sera restituée après la compression /. décompression à l'identique qu'elle était à l'origine.
- Compression avec perte : cette compression (de l'ordre 3 à 100) permet d'économiser énormément de place, en revanche elle à un cout : cette compression fait perdre de l'information et l'image restituée n'est pas la même que l'image d'origine.

Même si l'image restituée n'est pas exactement la même que celle que nous avions à l'origine, une forte compression permet tout de même de stocker 100 fois plus d'informations que nous ne devrions le faire. De plus, pour un image telle qu'une vignette sur un site, cela peut être intéressant car il n'est pas forcément nécessaire de charger l'image en grand si l'utilisateur ne clique pas dessus (cela permet aussi de faire gagner du temps de chargement de la page).

Durant ce TP, nous avons commencé par nous pencher sur la compression d'image en nuance de gris. Ce type d'image possède un gros avantage : il n'y a qu'une dimension sur laquelle travailler, là ou les images RGB ou YCbCr en possèdent trois.
Une fois que le code pour les images en nuance de gris a été réalisé, nous avons codé des fonctions similaires pour encoder des fonctions en trois dimensions.

Nous avons ainsi réussi à réaliser :

- L'encodage et le décodage d'une image en nuance de gris
- L'encodage et le décodage d'une image en RGB
- L'encodage et le décodage d'une image en YCbCr
- Le calcul du taux de compression d'une image en nuance de gris
- Le calcul du taux de compression d'une image en RGB
- Le calcul du taux de compression d'une image en YCbCr
- Le calcul du MSE pour toutes les images possibles

Nous avons testé l'ensemble de nos fonctions, cependant par soucis de clarté, nous n'avons pas affiché l'ensemble des tests dans ce compte-rendu. Il y a une partie de test dans notre code dans laquelle des tests peuvent être rajoutés autant que nécessaire.

Dans le compte-rendu de ce TP se trouve également un fichier Notebook qui contient tout le code que nous avons produit.
Si nous voulons poursuivre ce TP plus loin, il serait intéressant de se pencher sur les valeurs du MSE et du PSNR en fonction du facteur de qualité et de faire des analyses plus poussées du résultat que nous obtenons alors.

1 Codage de Huffman

Dans ce TP, nous aurons besoin de plusieurs fonctions afin de compresser nos images qui se basent sur le codage de Huffman. Les différentes fonctions qui permettent de convertir une liste en binaire grâce à ce type de codage sont déjà fournies, et nous n'allons donc pas les présenter.

2 Conversion d'une image en nuance de gris, en luminance/chrominance

Dans un premier temps, nous décidons de travailler sur nos images en nuance de gris. Nous avons ainsi codé les fonctions suivantes :

- Transformer une image RGB en niveau de gris
- Transformer une image RGB en YCbCr
- Transformer une image YCbCr en RGB

Voici les équations sur lesquelles nous nous sommes basés pour réaliser ces fonctions :

$$\begin{aligned} Y &= 0.299R + 0.587G + 0.114B \\ Cb &= -0.1687R - 0.3313G + 0.5B + 128 \\ Cr &= 0.5R - 0.4187G - 0.0813B + 128 \end{aligned}$$

$$\begin{aligned} R &= Y + 1.14020(Cr - 128) \\ G &= Y - 0.34414(Cb - 128) - 0.71414(Cr - 128) \\ B &= Y + 1.77200(Cb - 128) \end{aligned}$$

Par ailleurs, une image en niveau de gris ne contient que la luminance, soit la caractéristique Y. Voici alors les deux fonctions codées en Python qui nous permettent de transformer une image couleur :

```
def rgbToGray(img):
  """
  Convert a colored image to a grayscale one.
  """
  red, green, blue = img[:,:,0], img[:,:,1], img[:,:,2]
  # Formula of the Luminance
  return 0.299*red + 0.587*green + 0.114*blue
```

Code 1 – Fonction rgbToGray

```
def rgbToYCbCr(img):
  """
  Convert a rgb image to the yCbCr version.
  """
  yCbCr = img.copy()                                          # To create an oject of the
     same size
  red, green, blue = img[:,:,0], img[:,:,1], img[:,:,2]
  yCbCr[:,:,0] = 0.299*red + 0.587*green + 0.114*blue         #Formula of the Luminance
  yCbCr[:,:,1] = -0.1687*red - 0.3313*green + 0.5*blue + 128  #Formula of the chrominance
     red
  yCbCr[:,:,2] = 0.5*red - 0.4187*green - 0.0813*blue + 128   #Formula of the chrominance
     blue
  return yCbCr
```

Code 2 – Fonction rgbToYCrCb

```
def ycbcrToRGB(img):
  """
  Convert a yCbCr image to the rgb version.
  """
  rgb = img.copy()                                        # To create an oject of the
    same size
  y, cb, cr = img[:,:,0], img[:,:,1], img[:,:,2]
  rgb[:,:,0] = y + 1.14020*(cr-128)                       #Formula of the red
  rgb[:,:,1] = y - 0.34414*(cb-128) - 0.71414*(cr-128)    #Formula of the green
  rgb[:,:,2] = y + 1.77200*(cb-128)                       #Formula of the blue
  return rgb
```

Code 3 – Fonction ycbcrToRGB

En fait, pour tout le TP nous avons travaillé sur des fonctions qui prennent des images "d'une seule couleur". Ainsi, si l'on veut essayer de voir la compression avec l'image en couleur, il nous suffit d'extraire les trois couleurs, appliquer l'ensemble des fonctions de codage / décodage à chacune des composantes avant de reconstituer l'image originale en mettant les trois couleurs originales dans le bon ordre.

3 Fonctions utiles

Fonction rlencode

Afin d'encoder nos listes par la suite, notre objectif sera de réduire le nombre de bits que prends une photo. Ainsi, si nous retrouvons plusieurs symboles consécutifs nous allons les concaténer.
Prenons la liste **[1, 2, 0, 3, 0, 0, 0, 4, 0, 0]**. Nous remarquons qu'il y a de nombreux 0 consécutifs. L'objectif est alors de dire que si l'on rencontre 3 zéros consécutifs, ils seront ainsi remplacés par **[257, 3]**. Nous avons ainsi codé la fonction **rlencode** suivante :

```
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'.
  '''
  a, out = 0, []
  for i in data:
    if i==symbol:
      a+=1
    else:
      if a != 0:
        if a>1:
          out.extend((escape, a))
        else:
          out.append(symbol)
        a=0
      out.append(i)
  if data[-1]==symbol:
    if a>1:
      out.extend((escape, a))
    else:
      out.append(symbol)
  return out
```

Code 4 – Fonction rlencode

Par ailleurs, nous définissons dans cette fonction deux paramètres optionnels :

- **symbol**, valant 0 par défaut. Cela nous permettrait d'utiliser la fonction avec un autre symbole à encoder si ce n'était pas le 0 la valeur la plus récurrente.
- **escape**, valant 257 par défaut. Cela nous permettrait de changer la valeur de l'encodage qui indique le nombre de bits identiques successifs.

Fonction count_symbols

La fonction **count_symbols** a pour objectif de créer un dictionnaire à partir d'une liste. le dictionnaire possède pour clefs les différentes valeurs de la liste, et pour valeur leurs occurrences dans cette liste. Voici le code Python que nous avons réalisé :

```
def count_symbols(data:list) -> Dict[Any, int]:
  '''
  Compute the frequency of each value in the list.
  '''
  freq = {}
  for i in data:
    if freq.get(i) is None:
      freq[i]=1
    else:
      freq[i]+=1
  return freq
```

Code 5 – Fonction count_symbols

En testant cette fonction sur la liste **[0, 0, 0, 0, 1, 1, 1, 2, 2, 3]**, nous obtenons bien en sortie le dictionnaire **{0 : 4, 1 : 3, 2 : 2, 3 : 1}**.

4 Encodage d'une image

Fonction padding

Avant d'encoder une image au format JPEG, il faut que sa hauteur ainsi que sa largeur soient toutes deux multiples de 8, afin de créer des blocs de 8px par 8px par la suite. Si cette image ne satisfait pas cette condition, nous dupliquons la dernière ligne / la dernière colonne jusqu'à ce que ce soit le cas. Nous avons ainsi codé cette fonction :

```
def padding(img):
  """
  Complete the image to have weight and height multiples of 8 by copying last row/column.
  """
  while img.shape[1] % 8 != 0:
    img = np.insert(img, img.shape[1], img[:, -1], axis=1)
  while img.shape[0] % 8 != 0:
    img = np.insert(img, img.shape[0], img[-1, :], axis=0)
  return img
```

Code 6 – Fonction padding

Fonction numberOfBlock

Nous aurons aussi besoin, de connaître le nombre de blocs dans une image (pour nos boucles), ainsi nous avons codé la fonction **numberOfBlock** qui retourne un tuple qui contient le nombre de bloc en largeur et le nombre de bloc en hauteur :

```
def numberOfBlock(img):
  """
  Compute a tuple with two element : width and height the number of 8 by 8 block of the
    original image.
  img should be a "one colour" element (gray, only red, only blue, ...).
  """
  blockSize = 8

  width, height = len(img[0]), len(img)

  block_width = width//8 + 1 if width%8 else width//8
  block_height = height//8 + 1 if height%8 else height//8

  return (block_width, block_height)
```

Code 7 – Fonction numberOfBlock

Fonction to_block

Afin d'appliquer certaines fonctions, nous auront parfois besoin d'avoir un bloc. Nous utilisons alors la fonction **to_block** qui transorme une liste de taille 64 en un "bloc". Ce bloc est en réalité une liste de 8 éléments, dont chaque élément est lui même une liste de 8 éléments.

```
def to_block(data):
  """
  For a given data list of 64 elements, it returns the block.
  """
  out = [[0 for _ in range(8)] for _ in range(8)]
  for i in range(8):
    out[i]=data[i*8:i*8+8]
  return out
```

Code 8 – Fonction to_block

Fonction which_quantification_matrix

Dans le sujet, on nous indique qu'il existe deux matrices (que nous avons déclaré comme des variables globales) qui nous permettent de faire la compression. Ces deux matrices sont la matrice **qY** de luminance, ainsi que la matrice **qC** de chrominance.
Nous avons donc plusieurs cas à distinguer :

— Le bloc à quantifier est-il un bloc de luminance ou de chrominance ? Afin de distinguer ces deux cas, l'un des argument sera le booléen **isCrCb**

— Voulons nous appliquer un facteur de qualité ? Si nous ne voulons pas, nous prenons le cas par défaut choisie arbitrairement à -1, qui renvoie dans ce cas la matrice fournie dans l'énonce. Sinon, il faut calculer la matrice de quantification selon la valeur du facteur de qualité.

Afin de choisir la bonne matrice quand nous ferons les calculs sur les blocs, nous avons donc créé la fonction **which_quantification_matrix** :

```
def which_quantification_matrix(quality_factor=-1, isCbCr=False):
  """
  Return the quantification matrix, in function of the compression factor
  and if whe should take the chrominance or luminance matrix.
  """
  if quality_factor==-1:
    return (qC if isCbCr else qY)

  if quality_factor<50:
    s = 5000/quality_factor
  else:
    s = 200 - 2*quality_factor

  return (np.floor((s*qC+500)/500) if isCbCr else np.floor((s*qY+500)/500))
```

Code 9 – Fonction which_quantification_matrix

Fonction calculateBlock

Nous savons que pour réaliser la compression JPEG, il est nécessaire de réaliser les calculs suivants :

— Centrage : Il faut centrer les différentes valeurs. Pour une couleur donnée, il faut ainsi lui retrancher la valeur 128.

— Transformée en Cosinus : grâce à la bibliothèque cv2, nous appliquons la transformée, qui est un processsus réversible.

— Quantification : On fait la division du bloc trouvé par la matrice déterminée dans la fonction précédente.

Enfin, comme nous n'avons que des entiers qui caractérisent une image, nous devons reprendre chaque valeur après ces calculs et forcer son type en entier. Nous avons alors codé la fonction **calculateBlock** suivante :

```
def calculateBlock(block, quality_factor=-1, isCbCr=False):
  """
  For a given block, it will center it, apply the dct, quantify.
  """
  f_block = block + -128
  F_block = cv2.dct(np.float32(f_block),1)

  Q = which_quantification_matrix(quality_factor, isCbCr)
  Fc_block = F_block/Q

  for l in range(8):
    for h in range(8):
      Fc_block[l][h] = int(Fc_block[l][h])
  return Fc_block
```

Code 10 – Fonction calculateBlock

Fonction zigZag

Afin de parcourir les blocs en zig-zag, nous utilisons les variables globales **lig** et **col** déjà fournies dans le sujet. Nous avons donc codé la fonction **zigZag** qui, pour un bloc donné, retourne sa version parcourue en zigzag :

```
def zigZag(block):
  """
  For a given block, it will return the zigzaged version.
  """
  out = [0]*64
  for i in range(64):
    out[i] = block[lig[i]-1][col[i]-1]
  return out
```

Code 11 – Fonction zigZag

Fonction construct_huffman_coding

Bien que les fonctions codage de Huffman soient fournies dans le sujet, nous avons créé une fonction **construc_Huffman_coding** qui prends en argumnet une liste de data, qui appelle les idfférentes fonctions fournies et qui retourne un tuple contenant :

- La chaîne de caractères contenant la liste data, codé par le codage d'Huffman.
- Le dictionnaire contenant les clefs / valeurs afin de pouvoir décoder chaque mots plus tard.

Voici le code réalisé :

```
def construc_Huffman_coding(data):
  """
  For a given data, compute into a tuple :
  - the Huffman code of this data
  - the dictionnary of this huffman code
  """
  out=""
  block_huffman_dict = construct_huffman_table(data)
  block_huffman_encoded = encode_huffman(data, block_huffman_dict)
  out+=block_huffman_encoded
  return (out, block_huffman_dict)
```

Code 12 – Fonction construc_Huffman_coding

5 Décodage d'une image

Afin de décoder une image, nous aurons besoin de parcourir les étapes précédedntes dans le sens inverse. Nous aurons donc besoin de prendre un bloc parcouru en zig-zag afin de le remettre "à l'endroit", de faire les calculs précédents dans le sens inverse, ainsi que de décoder une liste à laquelle on a appliqué la fonction **rlencode**

Fonction undo_rlencode

À l'inverse de la fonction **rlencode**, nous avons ici créé la fonction **undo_rlencode**. Celle-ci prends en argument une liste, par exemple **[1, 2, 0, 3, 257, 3, 4, 257, 2]**, afin de retourner la version que nous avions avant de l'avoir encodée, c'est à dire **[1, 2, 0, 3, 0, 0, 0, 4, 0, 0]** (en supposant que le symbole est resté 0 et la valeur d'escape à 257). Voici alors le code réalisé :

```
def undo_rlencode(code, escape=257, symbol=0):
  """
  Transform a coded "escape symbol" by the real number of symbols.
  """
  for k in range(len(code)):
    code[k] = int(code[k])
  out = []
  skip = False
  for i in range(len(code)):
    if skip:
      skip = False
    elif code[i] == escape:
      out.extend(symbol for _ in range(round(code[i+1])))
      skip = True
    else:
      out.append(code[i])
  return out
```

Code 13 – Fonction undo_rlencode

Fonction undo_ZigZag

Nous appliquons "l'inverse" de la fonction zig-zag ici, et voici le code que nous avons créé pour la fonction **undo_ZigZag**

```
def undo_ZigZag(block):
  """
  For a given zigzaged list, it will return the block.
  """
  out = np.zeros((8,8))
  for k in range(64):
    out[lig[k]-1,col[k]-1] = block[k]
  return out.astype(int)
```

Code 14 – Fonction undo_ZigZag

Nous avons réalisé des tests dans le notebook afin de prouver que nos fonctions zigZag et undo_ZigZag fonctionnent bien.

Fonction undo_calculateBlock

Nous avions appliqué différents calculs sur les blocs, donc pour retrouver les blocs originaux, nous devons désormais :

- Multiplier le bloc par la matrice correspondante
- Appliquer la transformée en cosinus inverse
- Ajouter 128 pour compenser le centrage

Enfin, nous devons retransformer l'ensemble des valeurs en valeurs entières. Voici donc le code de la fonction **undo_calculateBlock**

```
def undo_calculateBlock(block, quality_factor=-1, isCbCr=False):
  """
  For a given block, it will multiply by the quantification matrix,
  apply the idct and uncenter it.
  """

  Q = which_quantification_matrix(quality_factor, isCbCr)

  out = block * Q
```

```
  out = cv2.idct(np.float32(out),1)
  out = out + 128
  for i in range(8):
    for j in range(8):
      out[i][j] = int(out[i][j])
  return out.tolist()
```

Code 15 – Fonction undo_calculateBlock

Fonction undo_padding

Afin de reconstruire l'image d'origine, il faut aussi supprimer les lignes et colonnes qui ont été ajoutées pour avoir des bloc e 8 par 8. Nous avons réalisé la fonction suivante qui reconstruire l'image d'origine à partir d'une image donnée, de sa hauteur et sa largeur d'origine :

```
def undo_padding(img, original_width, original_height):
  """
  Return the image from the original size, without the added row / column.
  """
  rows_to_remove = img.shape[1] - original_height
  cols_to_remove = img.shape[0] - original_width

  for _ in range(cols_to_remove):
    img = np.delete(img, -1, axis=0)

  for _ in range(rows_to_remove):
    img = np.delete(img, -1, axis=1)

  return img
```

Code 16 – Fonction undo_padding

Nous avons réalisé des tests dans le notebook afin de prouver que nos fonctions padding et undo_padding fonctionnent bien.

6 Mise en oeuvre du code

Nous avons désormais toutes les fonctions individuelles qui nous permettent d'encoder et de décoder une image bloc par bloc. Nous avons alors créé 6 fonctions :

1. encode_image_gray : encode une image en niveau de gris
2. encode_image_color : encode une image en RGB
3. encode_image_yCbCr : encode une image en YCbCr
4. decode_image_gray : decode une image en niveau de gris
5. decode_image_color : decode une image en RGB
6. decode_image_yCbCr : decode une image en YCbCr

6.1 Encodage

Nous allons créer une fonction **encode_image_gray** qui va nous permettre de d'appeler les différentes fonctions d'encodage afin de n'appeler que cette fonction avant d'encoder une image. Cette fonction prends deux arguments seulement :

- **url** : lien de l'image dont on veut faire la compression
- **quality_factor** : selon la valeur du facteur de qualité que l'on indique, on peut compresser plus ou moins lors de la division par la matrice de quantification.

Voici le code que nous réalisons pour encoder une image :

```
def encode_image_gray(url, quality_factor=-1):
  """
  Encode an image in gray scale, in function of the given quality_factor.
  """
  img_original = load_from_url(url)
  img_gray = rgbToGray(img_original)

  img_gray = padding(img_gray)
  width, height = len(img_gray[0]), len(img_gray)

  vRLC = []
  width_block, height_block = numberOfBlock(img_gray)

  for i in range(width_block):
    for j in range(height_block):
      current_block = img_gray[i*8 : i*8+8, j*8 : j*8+8]
      current_block = calculateBlock(current_block, quality_factor)
      current_block = zigZag(current_block)
      current_block = rlencode(current_block)
      vRLC.extend(current_block)

  my_image_encoded, my_image_dictionnary = construc_Huffman_coding(vRLC)

  return {"quality_factor": quality_factor,
          "image_width": width,
          "image_height": height,
          "encoded_image": my_image_encoded,
          "encoded_dictionnary": my_image_dictionnary,
          "width_block": width_block,
          "height_block":  height_block
          }
```

Code 17 – Fonction encode_image_gray

Cette fonction nous renvoie un dictionnaire avec les différentes valeurs qui seront utiles lors du décodage.
Nous ne présentons dans ce compte-rendu seulement la fonction qui permet d'encoder les niveaux de gris. Pour les couleurs et le YCbCr, il y a quelques petites différences dont voici une liste non-exhaustive :

- Il est nécessaire de faire des boucles dans les fonctions encode_image_color et encode_image_yCbCr car il faut faire chaque étape trois fois pour les trois composantes.
- Lorsque nous renvoyons "encoded_image", la valeur associée à cette clef n'est plus une chaine de caractère mais un dictionnaire. Ce dictionnaire contient trois clefs ("red" "green" et "blue" ou "y" "cb" et "cr"), dont les valeurs sont associées aux chaînes de caractères encodées.
- De même pour les dictionnaires, il y a trois dictionnaires à renvoyer donc nous renvoyons un dictionnaire avec trois paires key /value.
- Pour encode_image_yCbCr, il faut faire attention lors du codage : les trois matrices ne prennent pas la même matrice de quantification. il faut donc aussi prendre un booléen qui sera envoyé lors de l'appel à **which_quantification_matrix** lors des calculs.

6.2 Décodage

Nous allons créer une fonction **decode_image_gray** qui va nous permettre de d'appeler les différentes fonctions de décodage afin de n'appeler que cette fonction avant de décoder une image. Cette fonction prends un seul argument, qui est un dictionnaire. Ce dictionnaire est celui retourné par la fonction **encode_image_gray** et qui contient les différentes informations utiles. Voici le code que nous avons alors réalisé :

```
def decode_image_gray(encode_dict):
  """
  Decode a grayscale image already encoded. All the informations are on the
  dictionnary that is returned by encode_image_gray.
  """
  image_compressed = np.zeros((encode_dict.get("image_width"), encode_dict.get("
    image_height")))

  huffman_decoded = decode_huffman(encode_dict.get("encoded_image"), encode_dict.get("
    encoded_dictionnary"))
  rlc_decoded = undo_rlencode(huffman_decoded)

  for i in range(encode_dict.get("width_block")*encode_dict.get("height_block")):
    indice_width = i%encode_dict.get("width_block")
    indice_height = i//encode_dict.get("height_block")

    current_list = rlc_decoded[i*64:i*64+64]
    current_block = undo_ZigZag(current_list)
    current_block = undo_calculateBlock(current_block, encode_dict.get("quality_factor"))

    for j in range(8):
      for k in range(8):
        image_compressed[indice_width*8+j, indice_height*8+k] = current_block[k][j]
    image_compressed = image_compressed.astype(np.uint8)
  image_compressed = undo_padding(image_compressed, encode_dict.get("image_width"),
    encode_dict.get("image_height"))

  image_compressed = np.rot90(image_compressed, 3)
  image_compressed = np.fliplr(image_compressed)
  return (image_compressed)
```

Code 18 – Fonction decode_image_gray

Il y a un seul point sur lequel nous aimerions revenir : l'appel aux fonctions **np.rot90** et **np.flip**. Lors de nos calculs, nous devons faire une légère erreur d'indice à un endroit, ce qi fait que sans ces deux lignes, l'image se retrouve sur le côté et avec un effet miroir. Nous avons résolu ainsi ce problème grâce aux fonctions pré-conçues par la bibliothèque **numpy**, ce qui fait que la différence de temps de calcul est réellement négligeable.

Nous ne présentons dans ce compte-rendu seulement la fonction qui permet de décoder les niveaux de gris. Pour les couleurs et le YCbCr, il y a quelques petites différences dont voici une liste non-exhaustive :

- Il est nécessaire de faire des boucles dans les fonctions decode_image_color et decode_image_yCbCr car il faut faire chaque étape trois fois pour les trois composantes.
- Pour decode_image_yCbCr, il faut faire attention lors du décodage : les trois matrices ne prennent pas la même matrice de quantification. il faut donc aussi prendre un booléen qui sera envoyé lors de l'appel à **which_quantification_matrix**.

7 Résultats

Les différents tests de chaque fonction ayant été réalisés au préalable, nous pouvons désormais afficher une image selon différents critères. Nous allons prendre dans la partie suivante l'image de base, celle de lenna, disponible ici. Nous utilisons cette image pour l'ensemble nos tests pour plusieurs raisons :

- L'image est assez petite (256*256), ainsi les calculs sont amoindries par rapport à une image en HD qui est bien plus grande.
- Sa petite taille nous permet d'afficher 3 images côte à côte dans ce compte rendu, ce qui nous permet un gain de place et de lisibilité.
- L'image contient déjà suffisamment d'informations que nous pouvons comparer grâce aux différentes nuance de couleur, mais aussi de changement radical de colorimétrie d'une zone à l'autre (entre ses cheveux et son épaule par exemple).

7.1 Affichage de l'image sous toutes ses coutures

Une image existe donc dans trois modes principaux :

- RGB : Le mode "classique", chaque pixel est codé sur 3 octets : un pour le rouge, un pour le vert et un pour le bleu.
- Nuance de gris : Ce mode ne contient donc pas de couleur, chaque bit est représenté par un octet d'intensité de noir/blanc.
- YCrCb : Ce mode est caractérisé par la luminosité Y et les chrominances bleu et rouge.

Voyons comment la même image se représente donc selon ces trois mode de couleur :

IMAGE 1 – Image originale, en RGB

IMAGE 2 – Image originale, en nuance de gris

IMAGE 3 – Image originale, en YCrCb

Nous avons aussi affiché les images en nuance de rouge / vert / bleu en les isolant et voici l'affichage que nous obtenons alors :

IMAGE 4 – Image en ne gardant que le rouge

IMAGE 5 – Image en ne gardant que le vert

IMAGE 6 – Image en ne gardant que le bleu

De même, on peut décider d'afficher les images en intensité de chrominance et nous obtenons alors l'affichage suivant :

IMAGE 7 – Image en ne gardant que la luminance (qui est en réalité le niveau de gris)

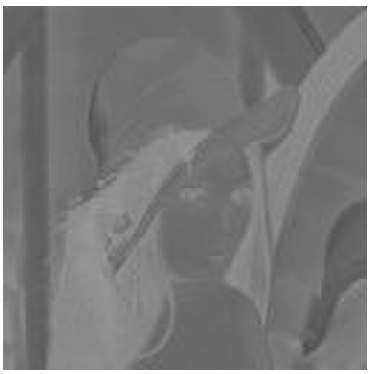

IMAGE 8 – Image en ne gardant que la chrominance bleue (en niveau d'intensité)

IMAGE 9 – Image en ne gardant que la chrominance rouge (en niveau d'intensité)

7.2 Différence de l'image décompressée (couleur) selon le taux de compression

Nous avons codé précédemment les fonctions d'encodage en couleur. Nous pouvons alors afficher l'image originale, ainsi que les images que l'on retrouve après compression / décompression pour deux facteurs de qualité différents :

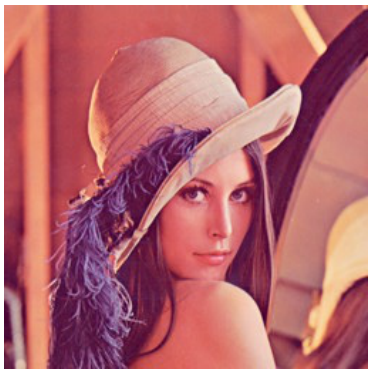

IMAGE 10 – Image originale, en RGB

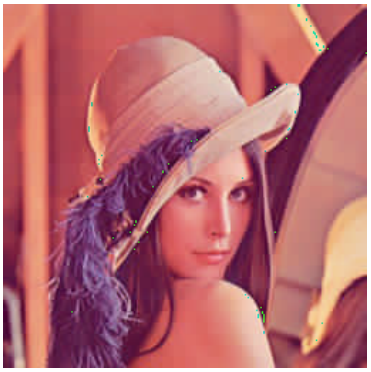

IMAGE 11 – Image compressée puis décompressée, avec $f_q = 10$

IMAGE 12 – Image compressée puis décompressée, avec $f_q = 90$

Plus le facteur de qualité f_q est grand, plus l'image que l'on obtient après décompression est proche de l'image avant compression. nous verrons dans un second temps que cela revient à obtenir en contrepartie un taux de compression plus faible. Le temps de calcul pour un plus grand taux de compression est aussi plus long, surtout sur la partie décodage de Huffman.
Nous remarquons par ailleurs que lorsque le taux de compression est bas, certains pixels sont entièrement vert pour une raison que nous n'avons pas réussi à trouver. En dehors de cela, l'image reste tout de même d'assez bonne qualité et fidèle à l'originale.

7.3 Différence de l'image décompressée (nuance de gris) selon le taux de compression

Nous avons codé précédemment les fonctions d'encodage en nuance de gris. Nous pouvons alors afficher l'image originale, ainsi que les images que l'on retrouve après compression / décompression pour deux facteurs de qualité différents :

IMAGE 13 – Image originale, en nuance de gris

IMAGE 14 – Image compressée puis décompressée, avec $f_q = 10$

IMAGE 15 – Image compressée puis décompressée, avec $f_q = 90$

De même qu'avec la couleur, plus le facteur de qualité f_q est grand, plus l'image que l'on obtient après décompression est proche de l'image avant compression. nous verrons dans un second temps que cela revient à obtenir en contrepartie un taux de compression plus faible. Le temps de calcul pour un plus grand taux de compression est aussi plus long, surtout sur la partie décodage de Huffman.
Nous remarquons par ailleurs que lorsque le taux de compression est bas, on peut commencer à distinguer les blocs notamment dans l'arrière plan. Il y a comme un léger effet de flou et les blocs deviennent des "moyennes" des pixels qui le compose. En poussant le taux de compression à son extrême, on remarque encore mieux cette effet de flou qui s'opère :

IMAGE 16 – Image compressée puis décompressée, avec $f_q = 1$

7.4 Différence de l'image décompressée en selon le taux de compression

Nous avons codé précédemment les fonctions d'encodage en yCrCb. Nous pouvons alors afficher l'image originale, ainsi que les images que l'on retrouve après compression / décompression pour deux facteurs de qualité différents :

IMAGE 17 – Image originale, en YCrCb

IMAGE 18 – Image compressée puis décompressée, avec $f_q = 10$

IMAGE 19 – Image compressée puis décompressée, avec $f_q = 40$

Nous pouvons remarquer déjà que nous n'avons pas demandé d'affichage pour $f_q = 90$, c'est parce que dans ce cas précis, le temps de réponse devient extrêmement long. C'est pourquoi nous n'affichons que pour $f_q = 40$. Il est difficile d'interpréter ces images, nous pouvons alors afficher la transformation de ces images en image couleur afin de voir ce qui a été fait, grâce à la fonction ycrcbToRGB que nous avons codé précédemment :

IMAGE 20 – Image originale, en rgb

IMAGE 21 – Image compressée puis décompressée, avec $f_q = 10$

IMAGE 22 – Image compressée puis décompressée, avec $f_q = 40$

L'affichage obtenu en sortie n'est pas satisfaisant, cependant nous ne trouvons pas pourquoi certaines couleurs apparaissent alors qu'elles ne devraient pas (le vert par exemple).

8 Le taux de compression

8.1 Fonctions afin de calculer le taux de compression

Fonction compression_rate

Nous voulons calculer le taux de compression que nous obtenons en compressant une image dont le facteur de qualité donné est f_q. nous appelons une fonction unique **compression_rate** qui calcule ce taux de compression selon la couleur indiquée. Voici le code que nous avons réalisé :

```
def compression_rate(url, color_mode, quality_factor):
  """
  Calculates the compression ratio defined by :
  the number of bit at the end divided by the original number of bit.
  """
  img_original = load_from_url(url)
  original_number_bits = img_original.shape[0] * img_original.shape[1] * 8
  final_number_bits = 0

  if color_mode=="gray":
    image_encoded_dictionnary = encode_image_gray(url, quality_factor)
    final_number_bits = len(image_encoded_dictionnary.get("encoded_image"))

  elif color_mode=="rgb":
    image_encoded_dictionnary = encode_image_color(url, quality_factor)
  else:
    image_encoded_dictionnary = encode_image_yCbCr(url, quality_factor)

  if color_mode!="gray":
    original_number_bits = original_number_bits * 3
    for key, value in image_encoded_dictionnary.get("encoded_image").items():
      final_number_bits += len(value)
  return original_number_bits/final_number_bits
```

Code 19 – Fonction compression_rate

Fonction plot_compression_rate

Afin d'afficher le taux de compression en fonction du facteur de qualité, nous appelons simplement la fonction créé précédemment **taux_compression** dans une boucle, et nous l'affichons avec certains paramètres afin d'embellir le tracé :

```
def plot_compression_rate(url, min=1, max=100, step=1):
  """
  Plot the compression rate of the three colour modes.
  """

  # Calculate every compression rate
  gray_table, rgb_table, yCbCr_table = [], [], []
  X = [i for i in range(min, max, step)]
  for quality_factor in X:
    gray_table.append(compression_rate(url, "gray", quality_factor))
    rgb_table.append(compression_rate(url, "rgb", quality_factor))
    yCbCr_table.append(compression_rate(url, "yCbCr", quality_factor))

  fig = plt.figure()

  # Set up the grid
  ax = fig.add_subplot(1, 1, 1)
  major_ticksx = np.arange(0, 101, 10)
  minor_ticksx = np.arange(0, 101, 1)
  major_ticksy = np.arange(0, 55, 5)
  minor_ticksy = np.arange(0, 55, 1)
  ax.set_xticks(major_ticksx)
  ax.set_xticks(minor_ticksx, minor=True)
  ax.set_yticks(major_ticksy)
  ax.set_yticks(minor_ticksy, minor=True)
  ax.grid(which='minor', alpha=0.2)
  ax.grid(which='major', alpha=0.7)

  # Plotting our datas

  plt.plot(X, gray_table, label="Image en nuance de gris")
  plt.plot(X, rgb_table, label="Image en couleur RGB")
  plt.plot(X, yCbCr_table, label="Image en chrominance yCbCr")
```

```
  plt.legend(bbox_to_anchor=(1.04, 0), loc="lower left", borderaxespad=0)
  plt.ylabel("Taux de compression")
  plt.xlabel("Facteur de qualit ")
  plt.title(f"Taux de compression en fonction du facteur de qualit ")
```

Code 20 – Fonction plot_compression_rate

8.2 Résultats

Grâce à la fonction **plot_compression_rate**, nous pouvons afficher le taux de compression en fonction du facteur de qualité pour les trois versions de compression / décompression.
Voici l'affichage que nous obtenons lorsque nous appelons la fonction avec un intervalle de [1, 99] avec un pas de 1, après près de 5 minutes de temps de calcul :

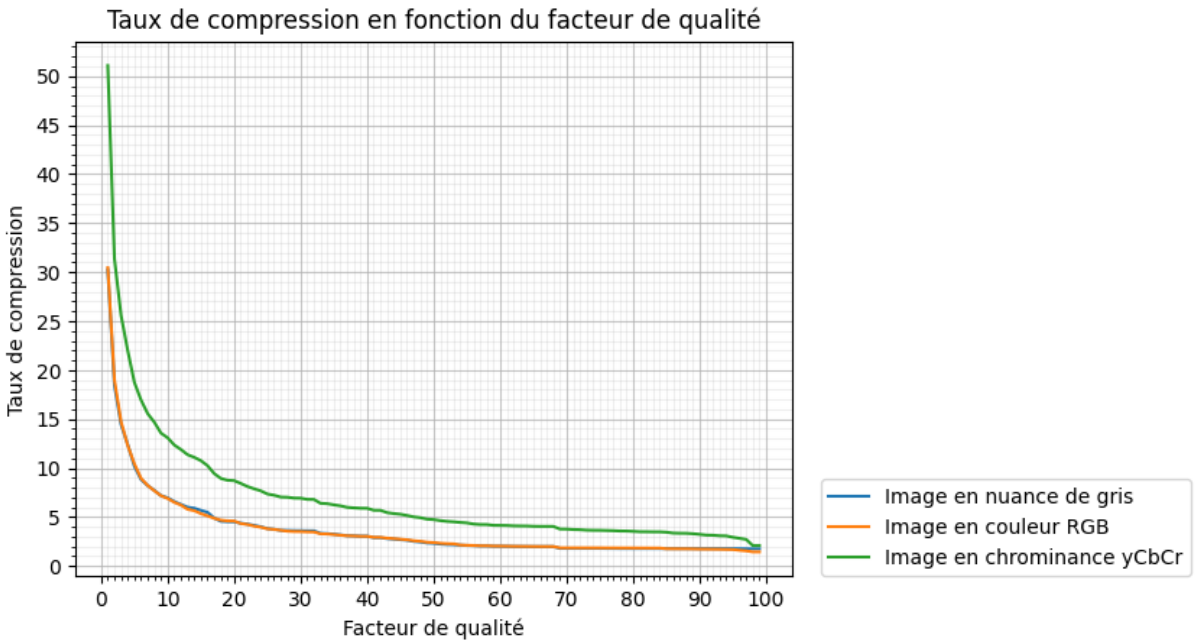

Image 23 – Taux de compression en fonction du facteur de qualité

En augmentant le facteur de qualité, nous obtenons effectivement une image image encodé dont le taux de compression est de plus en plus faible (car on perds moins d'informations). Cependant, nous pourrions considérer que nous nous plaçons à un endroit à un facteur de qualité suffisamment grand pour que l'image ne perde pas en qualité, ou que la différence ne soit pas visible à l'affichage (pour une vignette en petit pour un site par exemple).
Nous remarquons par ailleurs que la courbe du taux de compression pour l'image en nuance de gris et celle en RGB sont pratiquement superposées. Elles obtiennent donc le même taux de compression en fonction du facteur de qualité.
Enfin, nous remarquons le grand avantage de l'image en yCbCr : pour un même facteur de qualité, le taux de compression est bien meilleur. Cela permet ainsi d'obtenir une image qui prends beaucoup moins d'espace, mais qui, une fois décompressée, garde tout de même une bonne qualité.

Comme nous ne pouvons pas faire de compression JPEG sans perte avec un taux de compression supérieur à 8, nous réalisons ce deuxième affichage avec un intervalle de [23, 99] seulement, puisque c'est à partir de 23 que le facteur de qualité est assez grand pour ne pas avoir de perte pour l'image au format yCbCr. Voici le tracé que nous obtenons alors :

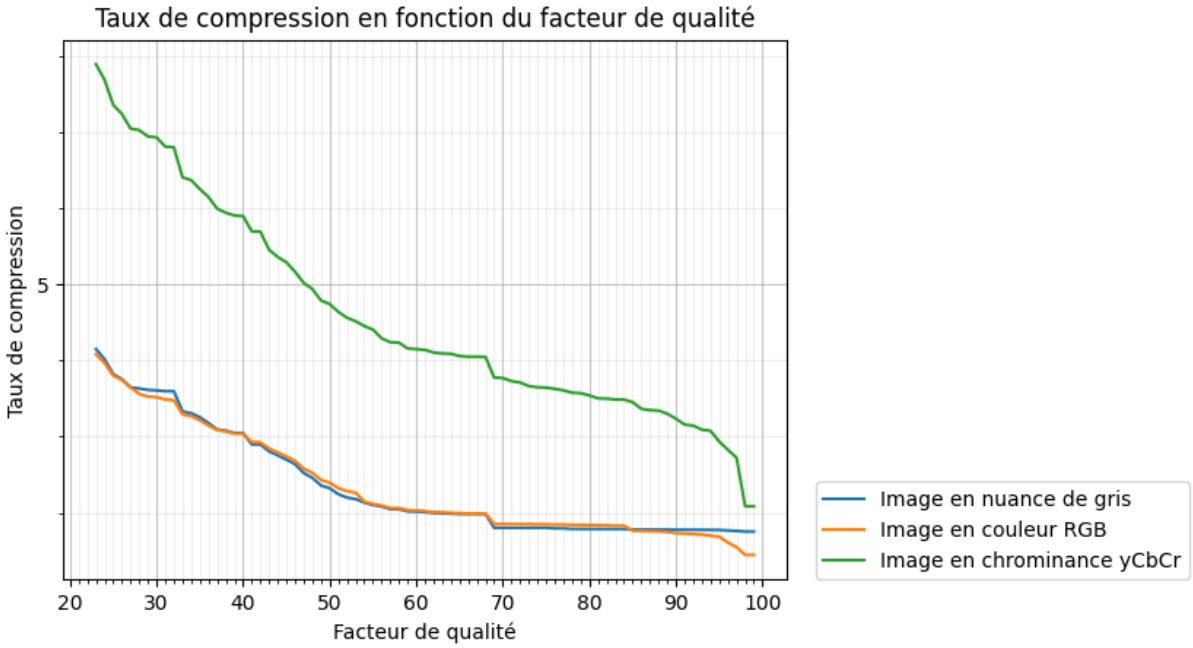

IMAGE 24 – Taux de compression en fonction du facteur de qualité, "sans-perte"

Nous avons ainsi la courbe du taux de compression telle que cette compression soit sans perte (en théorie) en fonction du facteur de qualité que l'on choisit.

9 MSE

Enfin, nous avons voulu calculer le MSE et le PSNR selon le taux de compression. Nous avons donc codé trois fonctions qui permettent de calculer ces deux données. Voici le codé réalisé pour **MSE_gray** :

```
def MSE_gray(url, quality_factor=-1):
  """
  Calculate the MSE of an image (in grayscale), taking into account the quality_factor.
  """
  original_image = load_from_url(url)
  encoded_img = encode_image_gray(url, quality_factor)
  decoded_img = decode_image_gray(encoded_img)
  D = 0
  image_gray = rgbToGray(original_image).astype(np.uint8)
  for i in range(encoded_img.get("image_width")):
    for j in range(encoded_img.get("image_height")):
     D += (int(image_gray[i][j])-int(decoded_img[i][j]))**2
  D = D/(encoded_img.get("image_width")*encoded_img.get("image_height"))
  PSNR = 10*np.log10(255**2/D)

  print(f"MSE : {D}")
  print(f"PSNR : {PSNR} \n")
```

Code 21 – Fonction MSE_gray

Nous avons aussi codé les fonctions **MSE_color** et **MSE_yCbCr**, que nous ne présentons pas dans ce compte-rendu. En faisant un test, présent sur le notebook, nous obtenons l'affichage suivant :

```
Calcul du MSE et du PSNR pour un facteur de qualité de 35 :

Pour l'image en nuance de gris :
MSE : 15.06353759765625
PSNR : 36.35153385103894

Pour l'image en RGB :
MSE : 36.69623819986979
PSNR : 32.48458814675059

Pour l'image en YCbCr :
MSE : 4398.415176391602
PSNR : 11.697841398663897
```

IMAGE 25 – Calcul des MSE et PSNR pour $f_q = 35$