



# Rapport de Bureau d'Etude

INF-tcl

BE n°3

Algorithmes et structures de données :

*Compression d'image*

**Groupe Ab2**

Marijan SORIC  
Rémi LAURENS-BERGE

**Encadrant**

Stéphane DERRODE

# Table des matières

<b>1</b>	<b>Introduction</b>	<b>1</b>
<b>2</b>	<b>Optimisation de la compression</b>	<b>1</b>
<b>3</b>	<b>Ouverture</b>	<b>11</b>

## 1 Introduction

L'objectif du rapport est de compresser une image au choix, en appliquant la méthode de quadripartition et en l'optimisant au travers des questions suivantes.



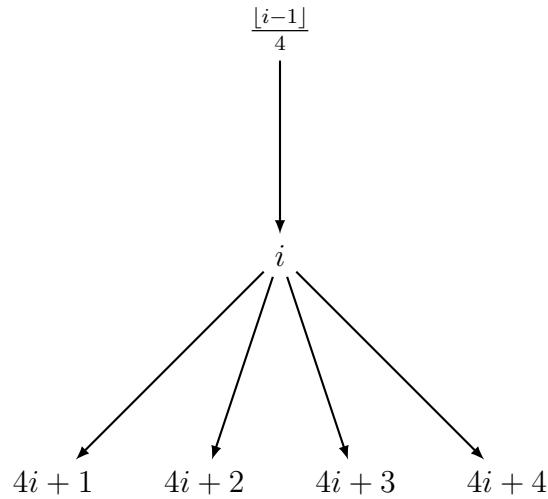
FIGURE 1 – Image à compresser

*Source :* [https://www.google.com/url?sa=i&url=https%3A%2F%2Fwww.tripadvisor.fr%2FTourismg187265-Lyon\\_Rhone\\_Auvergne\\_Rhone\\_AlpesVacations.html&psig=AOvVaw3AtbbStMIQN17bfGamDt1&ust=1651949346453000&source=images&cd=vfe&ved=0CAwQjRxqFwoTCMjk8YvFy\\_cCFQAAAAAdAAAAABAA](https://www.google.com/url?sa=i&url=https%3A%2F%2Fwww.tripadvisor.fr%2FTourismg187265-Lyon_Rhone_Auvergne_Rhone_AlpesVacations.html&psig=AOvVaw3AtbbStMIQN17bfGamDt1&ust=1651949346453000&source=images&cd=vfe&ved=0CAwQjRxqFwoTCMjk8YvFy_cCFQAAAAAdAAAAABAA)

Nous nous sommes intéressés à l'image ci-dessus. Cette image est caractérisé par des zones à la fois homogènes avec le ciel que nous voudrons compresser et des zones avec de fortes variations de couleurs (qui seront moins compressées) au niveau des habitations, du ponts ... Grâce à ces différentes zones, nous pouvons observer des noeuds à la fois grand et petit en fonction d'un seuil d'homogénéité.

## 2 Optimisation de la compression

**Exercice 3.1** Par analogie avec les arbres binaire, pour les arbres de quadripartition les formules pour calculer pour un noeud à l'indice  $i > 0$  les indices de son parent et de ses 4 enfants sont :



En numérotant depuis  $i=0$ , avec 0 le numéro de la racine.

**Exercice 3.2** La classe *Noeud* est modifiée pour générer un arbre quaternaire sous forme implicite. Dans cette question, le but est de compresser l'image à l'aide de la méthode de quadripartition. Nous avons créer l'arbre des noeuds de façon implicite sous de tas c'est-à-dire que la valeur de tout noeud sera inférieure à celle de ses enfants. Tous les noeuds seront stockés dans une liste nommée *arbre\_implicit*. Cette manière de stocker les noeuds a pour avantage de déterminer la position d'un noeud uniquement à partir de sa position dans la liste sans inclure de référence explicite entre les noeuds.

Pour cela, nous avons changé la classe *Noeud* en ajoutant un attribut *terminal* qui est un bouléen égal *True* si le noeud est un terminal.

La fonction pour créer l'arbre implicite est le suivant :

---

```

1 def arbre_implicit(px, x, y, w, h, seuil, im):
2     arbre = []
3     mr, mv, mb = moyenne(px, x, y, w, h)
4     couleur = (round(mr), round(mv), round(mb))
5     arbre.append(NoeudImplicit(x, y, w, h, im, couleur)) # ajout du premier noeud
6     i = 0
7     while i == 0 or i < len(arbre)-1: # i ==0 pour l'initialisation
8         x, y, w, h = arbre[i].getDim()
9         if homogene2(px, x, y, w, h, seuil, im)==True or w < 2 or h < 2:
10             arbre[i].setTerminal() # le noeud est homogène donc on le met terminal
11             mr, mv, mb = moyenne(px, x, y, w, h)
12             couleur = (round(mr), round(mv), round(mb))
13             arbre[i].setCouleur(couleur)
14             i += 1
15         else:
16             for x1, y1, w1, h1 in partition(x, y, w, h):
17                 arbre.append(NoeudImplicit(x1, y1, w1, h1, im)) # ajout des quatre enfants à la su
18             i += 1
19     return arbre

```

---

Ce code permet ajouter à la fin de la liste *arbre\_ implicite* les enfants de chaque noeud (s'il y en a) au fur et à mesure qu'on la parcourt. Cette façon de trier la liste de noeud permet de vérifier que l'indice de tout noeud doit être inférieur à celui de ses enfants

**Exercice 3.3** Le changement de structure (désormais implicite) a un impact sur la performance du programme. En effet, elle se mesure par la vitesse d'exécution et la consommation de mémoire de l'application.

Afin de déterminer le temps d'exécution, nous important le module *time*. En faisant appel à la méthode *time* (en claire *tps1 = time.time()*) ce module au début du programme principal et à la fin, par soustraction, nous obtenons le temps d'exécution du programme en secondes. Concernant la partie spatiale (mémoire), au niveau du Terminal (Invite de Commande), on import *pip3 install memory – profiler*. En amont de la fonction à surveiller, on écrit *@profile*, en l'ayant importé avant depuis la bibliothèque *memory\_profiler*. Enfin, on exécute la ligne de commande *python – mmemory \_profiler FicherTD3.py*.

Nous avons mesuré les effets de l'introduction de la méthode implicite sur la performance de notre programme. Nous avons comparé la vitesse d'exécution et la consommation de mémoire de l'application.

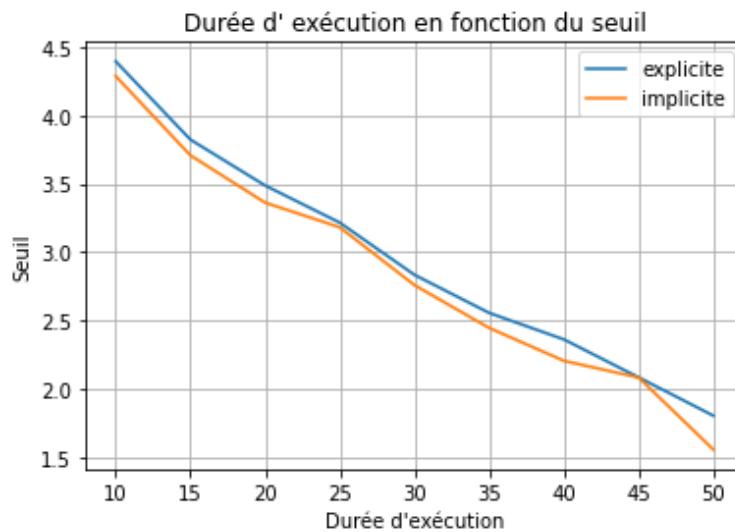


FIGURE 2 – Comparaison des vitesses d'exécution des deux méthodes

On observe que la méthode implicite ne permet pas de rendre le programme plus rapide, en revanche il permet de faire un gain dans l'espace mémoire.

**Exercice 3.4** Pour affiner le modèle d'homogénéité, on peut prendre en compte les spécificités de l'acuité visuelle, notamment la luminance des zones de l'image, les contrastes et la position relative à l'image. Détaillons ces 3 points particuliers :

- **Luminance** : l'oeil humain est particulièrement sensible à la luminosité en effet, il perçoit mieux les nuances de couleurs claires que de celles foncées. C'est pourquoi

on distingue mieux les tonalités de vert que de bleu. Il existe un modèle (le modèle  $YC_bC_r$ ) qui définit totalement l'espace colorimétrique (il existe une bijection entre ce modèle et le  $RGB$ ) et caractérise la luminance (par  $Y$ ).

$$\begin{cases} Y = 0,299 \cdot R + 0,587 \cdot G + 0,114 \cdot B \\ C_b = -0,1687 \cdot R - 0,3313 \cdot G + 0,5 \cdot B \\ C_r = 0,5 \cdot R - 0,4187 \cdot G - 0,0813 \cdot B \end{cases}$$

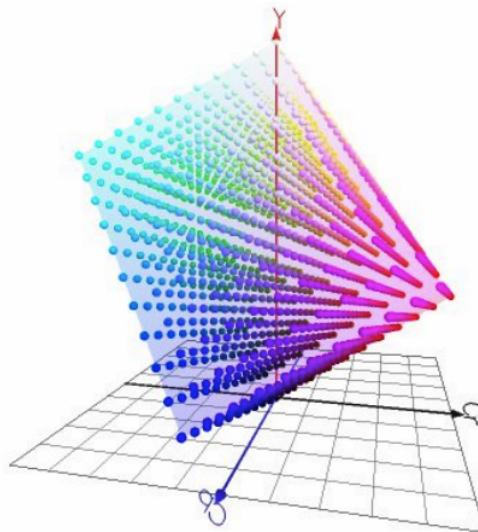


FIGURE 3 – Espace couleur  $YC_bC_r$

Source : Haytham Elghazel [https://perso.univ-lyon1.fr/haytham.elghazel/Papers/Pdf/Rapport\\_PFE.pdf](https://perso.univ-lyon1.fr/haytham.elghazel/Papers/Pdf/Rapport_PFE.pdf)

Ainsi la donnée  $Y$  est une moyenne pondérée des couleurs  $R$ ,  $G$  et  $B$  qui caractérise la luminance de l'image, tandis que l'œil humain est moins sensible à la chrominance. Alors, plus une zone de l'image est de luminance faible, plus la zone apparaît homogène pour l'œil humain.

- **Contraste** Le contraste d'une zone observée conditionne l'acuité visuelle, en particulier la luminance est liée à la notion de contraste qui permet de le définir :

$$C = \frac{L_{max} - L_{min}}{L_{max} + L_{min}}$$

C'est le contraste que nous allons utiliser pour quantifier la luminosité d'une zone ( $Y = L$ ). Voici la fonction *contraste* utilisée

---

```

1 def contraste(px, x, y, w, h):
2     Y=[]
3     for i in range(x, x+w):
4         for j in range(y, y+h):
5             R, G, B =px[i,j]
6             Y.append(0.299*R+0.587*G+0.114 * B) # utilisation de Y luminance
7     C, c = max(Y), min(Y)

```

```

8     if c+C!=0:
9         return (C-c)/(C+c)
10    else:
11        return 0

```

---

- **Position relative** Intéressons-nous maintenant au champs visuel de l'observateur : dans un premier temps, face à une image, l'œil a tendance à observer le centre de l'image plutôt que ses extrémités. Nous souhaitons donc une meilleure précision des zones centrées plutôt que celles périphérique. Pour ce faire, nous optons pour une échelle logarithmique, bien utile et efficace pour modéliser la sensibilité humaine (comme le montre l'échelle des décibels). En effet, l'échelle décimale a pour principal inconvénient d'avoir un pas irrégulier tandis qu'une échelle logarithmique reflète mieux les propriétés visuelles. Les données de la FIGURES4 ne concernent que de petits angles, mais nous choisissons d'extrapoler l'analyse à toute l'image.

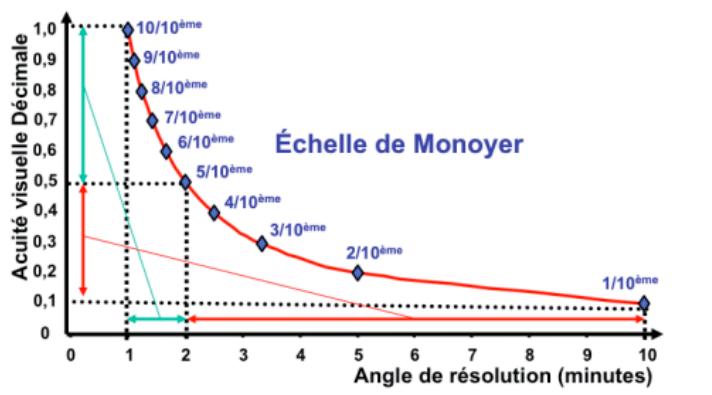


FIGURE 4 – Acuité visuelle en fonction de l'angle périphérique de vision

Source : <https://www.cahiers-ophtalmologie.fr/media/437fdb39c4b9ed3f2a78eada5fc5c019.pdf>

Pour modéliser cette sensibilité spatiale de l'œil, nous introduisons une fonction  $f$ , appelée fonction de précision. Graphiquement,

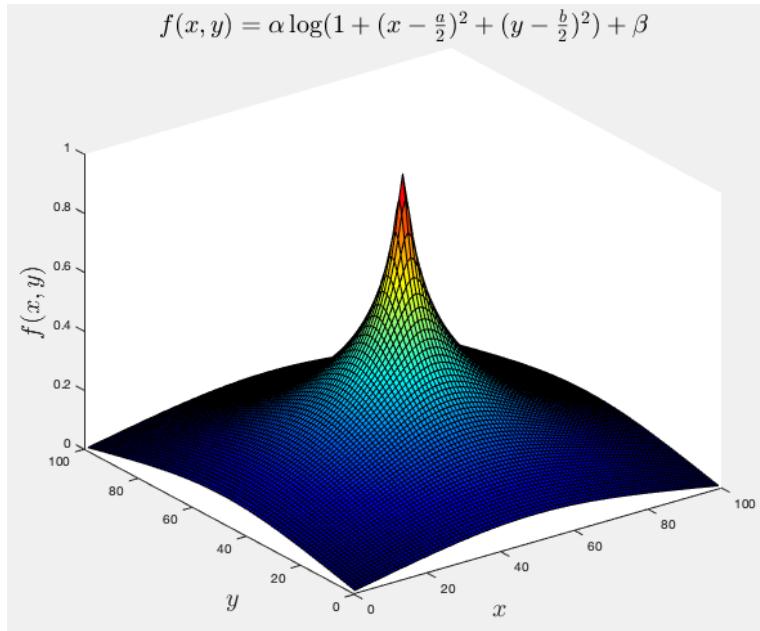
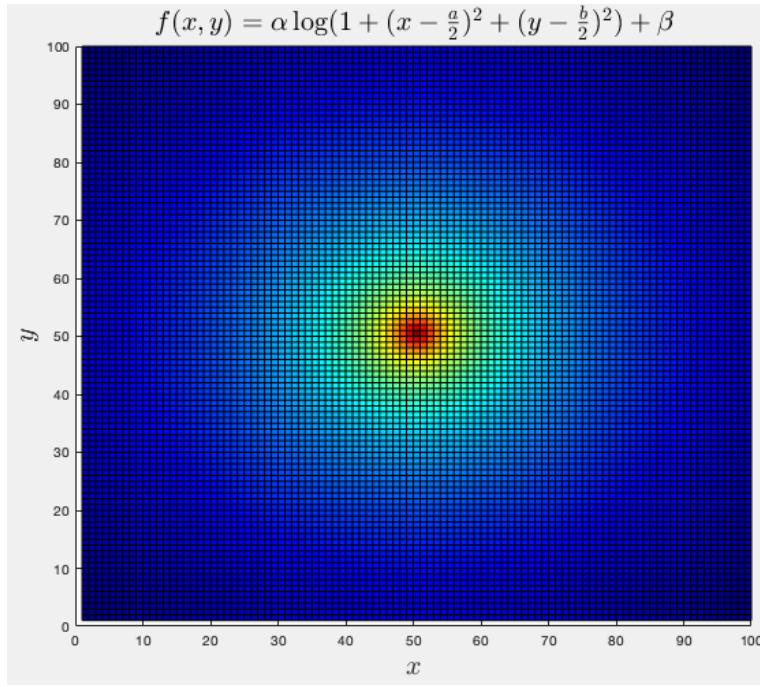


FIGURE 5 – Fonction de précision normée pour une photo carrée 100x100

Cette fonction vient moduler la précision souhaitée pour l'homogénéisation des zones de l'images, avec pour  $f \rightarrow 1$ , on désir une meilleure précision que pour  $f \rightarrow 0$ . Voici la fonction  $f$  utilisée (avec  $(W, H)$  la largeur et hauteur de l'image) :

$$f(x, y) = 1 - \left| \frac{\log \left[ 1 + \left( \frac{x - \frac{W}{2}}{W} \right)^2 + \left( \frac{y - \frac{H}{2}}{H} \right)^2 \right]}{\log \left[ 1 + \frac{1}{4W^2} + \frac{1}{4H^2} \right]} \right|$$

Détaillons comment nous avons construit (arbitrairement) cette fonction : dans un premier temps, on utilise le logarithme pour rendre compte du phonème expliqué précédemment. La formule  $(x - W/2)^2 + (y - H/2)^2$  calcul la distance (son carré plus précisément) d'un point situé à la position  $(x, y)$  par rapport au centre de l'image qui permet de déterminer des lignes de niveau circulaires (cas de la FIGURE 5). Cependant pour une image rectangulaire, pour rendre compte de la géométrie de l'image, on préférera utiliser des lignes de niveau elliptique, d'où les 2 divisions par  $W$  et  $H$ . Afin de d'obtenir un logarithme croissant, on incrémente de 1 l'argument qu'il prend. Enfin, nous normalisons en divisant par le logarithme évalué pour le point le plus lointain  $(x, y) = (0, 0)$ . Enfin en prenant  $\beta = 1$ , nous obtenons une surface comprise entre  $[0;1]$ .

*NB : Dans le code nous avons utilisé la  $f_{cerclle}$  (simplifiée) donnée ci-dessous.*

$$f_{cerclle}(x, y) = 1 - \left| \frac{\log \left[ 1 + \left( x - \frac{W}{2} \right)^2 + \left( y - \frac{H}{2} \right)^2 \right]}{\log \left[ 1 + \left( \frac{W}{2} \right)^2 + \left( \frac{H}{2} \right)^2 \right]} \right|$$

La fonction pour *homogene2* est la suivante :

---

```

1 def homogene2(px,x,y,w,h,seuil,im):
2     W, H = im.size
3     c=contraste(px, x, y, w, h)
4     xp, yp= x+w/2, y+h/2 # position au centre de la zone
5     f = 1-abs(log(1+((xp-W/2)**2+(yp-H/2)**2))/(log(1+(W/2)**2+(H/2)**2)))
6     return (c*f**0.5<1/seuil) #mondulation du contraste par la fonction de précision

```

---

Nous choisissons comme condition d'homogénéité  $c \times \sqrt{f} < 1/seuil$  qui rend compte à la fois du contraste de la zone mais aussi de la précision par une simple multiplication. Voici le résultat pour différentes valeurs de seuils :

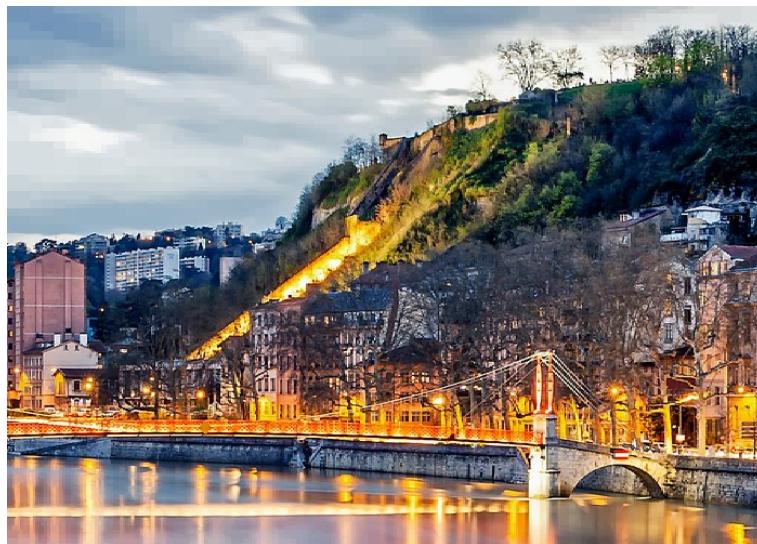


FIGURE 6 – Traitement avec le nouveau critère d'homogénéité  $seuil = 100$

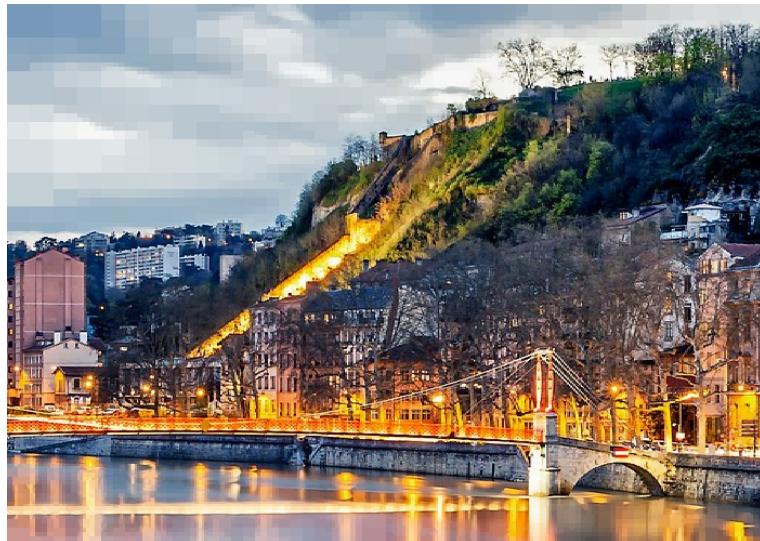


FIGURE 7 – Traitement avec le nouveau critère d'homogénéité  $seuil = 50$

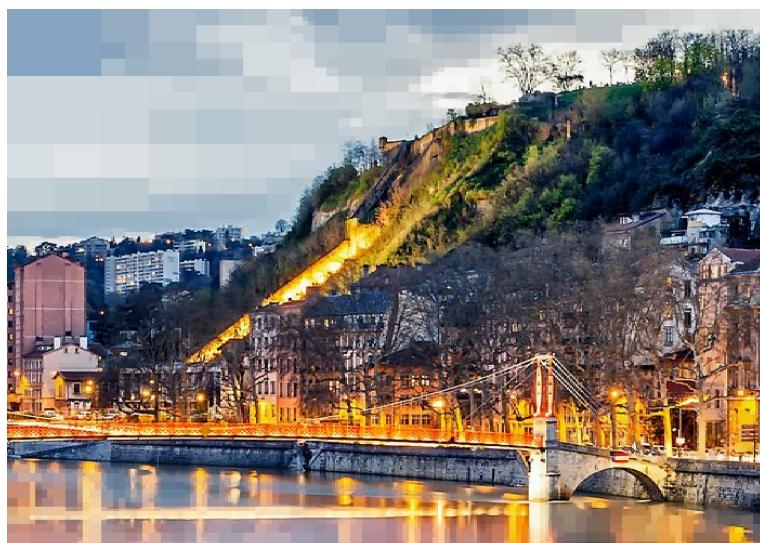


FIGURE 8 – Traitement avec le nouveau critère d'homogénéité  $seuil = 25$

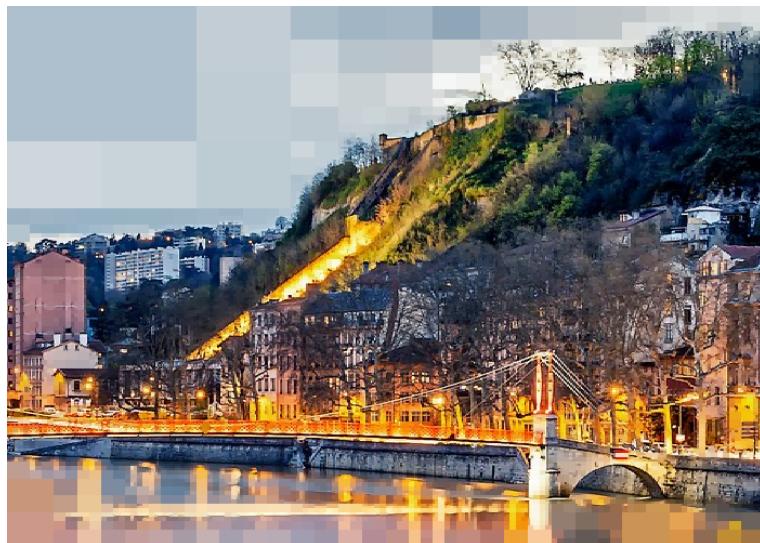


FIGURE 9 – Traitement avec le nouveau critère d'homogénéité  $seuil = 15$

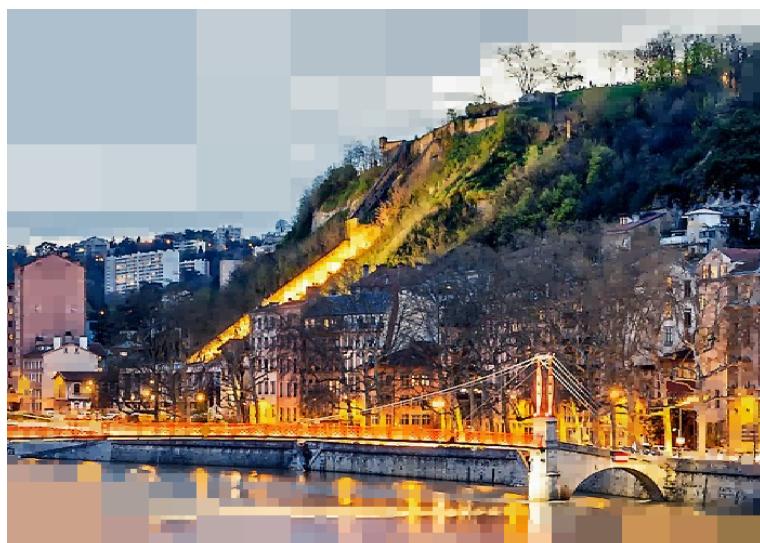


FIGURE 10 – Traitement avec le nouveau critère d'homogénéité  $seuil = 10$

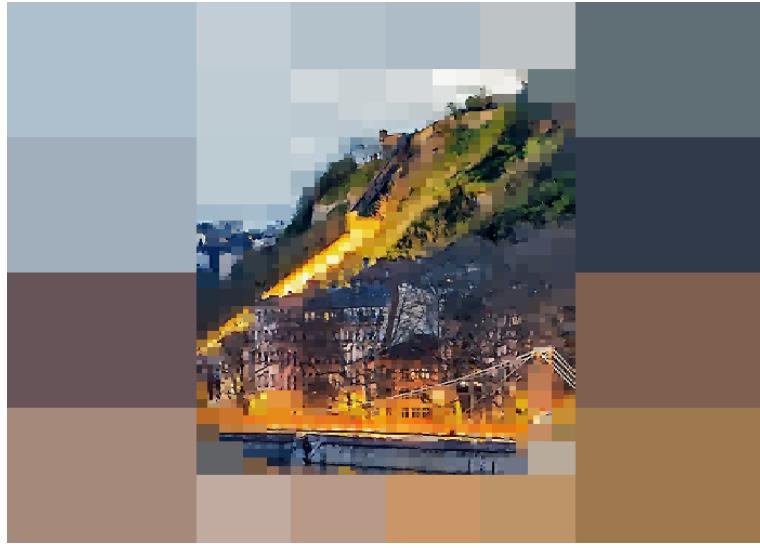


FIGURE 11 – Traitement avec le nouveau critère d'homogénéité  $seuil = 3$

**Exercice 3.5** Observons l'influence du nouveau critère sur le  $PSNR$  par rapport au critère initial ? Le Peak-Signal-to-Noise Ratio est un rapport de puissance entre celle maximal du signal et celle du bruit qui corrompt l'image. C'est donc un indicateur de lisibilité d'un signal (ici d'une image). Pour  $PSNR=0$  dB, aucune information n'est transmise par l'image : la compression a totalement détérioré l'image. Plus une compression a un  $PSNR$  proche de l'image initiale, plus sa reconstitution est fidèle à l'image d'origine (en terme d'information).

**Exercice 3.6** Le but de cette question est de rendre compte de la faible sensibilité de l'oeil aux détails en ajoutant une fonction *filtre* qui permet de rendre une image floue.

---

```

1  def filtre(px, w, h):
2      I = [(-1,-1), (-1, 0), (-1, 1), (0, -1), (0, 1), (1, -1), (1, 0), (1, 1)]
3      for i in range(w):
4          for j in range(h):
5              sr = 0
6              sv = 0
7              sb = 0
8              nb_voisin = 0
9              for e1, e2 in I:
10                  if 0 <= e1 + i < w and 0 <= e2 + j < h: # on fait attention si le pixel se situe sur
11                      nb_voisin += 1
12                      r, v, b= px[e1 +i, e2+j]
13                      sr += r
14                      sv += v
15                      sb += b
16              px[i,j] = round(sr/nb_voisin), round(sv/nb_voisin), round(sb/nb_voisin)

```

---

Pour rendre floue la photo, nous avons choisi de moyenner chaque pixel avec ses voisins dans les 8 directions. Nous avons fait attention pour chaque pixel, s'il se trouverait en sélectionnant (ligne 10) seulement les voisins appartenant au dimension de l'image. On obtient ce résultat en floutant l'image de Lyon.



FIGURE 12 – Image flouté

### 3 Ouverture

Pour aller plus loin, au niveau de la fonction d'homogénéisation, il aurait intéressant de faire appel à des convolutions entre les pixels/zones de l'image pour détecter les similitudes dans l'image pour pouvoir les traiter ensemble. On pourrait aussi essayer de détecter les zones d'intérêt pour l'image (plus complexe que de prendre uniquement le centre de l'image). D'autre part, pour discréteriser un espace (comme on peut le faire pour les simulations numériques en mécanique) on utilise généralement la triangulation comme maillage (plus flexible que le rectangle).