

# Informatique

## **Matrices de pixels et images**

*Cours*

<b>Gestion des images BMP sous Python .....</b>	<b>4</b>
<b>1.I. Principe du codage des images BMP .....</b>	<b>4</b>
1.I.1 Les triplets RGB.....	4
1.I.2 Bits de poids forts et faibles .....	5
<b>1.II. Représentation matricielle .....</b>	<b>6</b>
1.II.1 Images en couleur.....	6
1.II.2 Images en nuances de gris.....	7
1.II.3 Image en noir et blanc .....	8
<b>1.III. Prise en main des images avec Python .....</b>	<b>9</b>
1.III.1 Ouverture d'une image au format BMP .....	9
1.III.1.a Introduction.....	9
1.III.1.b Ouverture de l'image .....	9
1.III.1.c Gestion des erreurs .....	10
1.III.1.d Conversion d'une image RGBA .....	10
1.III.1.e Le format array .....	11
1.III.1.e.i Types .....	11
1.III.1.e.ii Overflow .....	11
1.III.1.e.iii Récupération de valeurs .....	12
1.III.1.e.iv Ouverture/enregistrement d'un array .....	13
1.III.1.e.v Avantage des slices .....	13
1.III.2 Affichage d'une image.....	14
1.III.3 Exploration graphique.....	14
1.III.4 Enregistrement d'une image .....	15
1.III.5 Traitement des autres formats.....	15
<b>1.IV. Travail sur les images .....</b>	<b>15</b>
1.IV.1 Parcours d'une image .....	15
1.IV.2 Modification de pixels.....	16
1.IV.3 Création d'une image noire ou blanche.....	16
<b>1.V. Convolution.....</b>	<b>17</b>
1.V.1 Principe .....	17
1.V.2 Gestion des bords.....	18
1.V.3 Normalisation.....	18
1.V.4 Limitation des valeurs résultat .....	18
1.V.5 Quelques exemples .....	19
<b>1.VI. Transformations.....</b>	<b>20</b>
1.VI.1 Rotation .....	20
1.VI.1.a Matrice de rotation.....	20
1.VI.1.b Algorithme naïf.....	21
1.VI.1.c Algorithme inverse.....	22
1.VI.1.c.i Méthode classique.....	22
1.VI.1.c.ii Méthode bilinéaire.....	23
1.VI.2 Réduction.....	24
1.VI.2.a Suppression de lignes/colonnes .....	24
1.VI.2.b Moyenne par paquets.....	25
1.VI.3 Agrandissement.....	26
1.VI.3.a Interpolation par plus proche voisin .....	26

1.VI.3.b Interpolation bilinéaire .....	28
1.VI.3.c Interpolation bicubique.....	30
1.VI.3.d Comparaison .....	31






# Gestion des images BMP sous Python

Le programme d'informatique nous propose d'aborder la gestion des images sous Python. Voici donc un paragraphe dédié à cela.

## 1.I. Principe du codage des images BMP


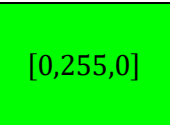
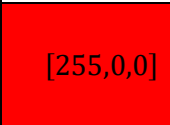
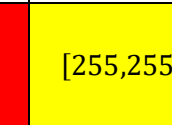

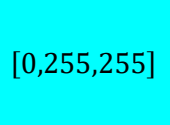
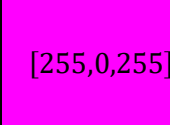
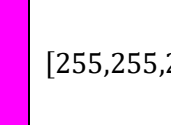
### 1.I.1 Les triplets RGB

La couleur de chaque pixel d'une image BMP est codée sur un certain nombre de bits. Nous nous intéressons ici à un codage en 24 bits (3 octets) par pixel, soit la possibilité d'avoir 16,7 millions de couleurs. Ces 24 bits sont divisés en 3 fois 8 bits codant chacun pour une couleur R (red) G (green) B (blue). 8 bits permettent de définir un nombre entre 0 et 255, ce qui nous permet finalement de dire qu'à chaque pixel est associé un triplet d'entiers, par exemple [85,1,255], correspondant aux couleurs RGB. Par exemple :

	R=0 G=0 B=255
	R=255 G=0 B=0
	R=0 G=255 B=0

Pour chaque couleur, le nombre utilisé en machine est le binaire de l'entier, par exemple : 0b01001110 (0b n'est qu'une notation pour dire que le nombre est décrit en binaire, les bits suivent).

Voici pour information ce que donnent les couleurs en prenant les valeurs extrêmes 0 et 255 pour R, G et B :

$R = 0$			$R = 255$		
	$G = 0$	$G = 255$		$G = 0$	$G = 255$
$B = 0$	 [0,0,0]	 [0,255,0]	$B = 0$	 [255,0,0]	 [255,255,0]
$B = 255$	 [0,0,255]	 [0,255,255]	$B = 255$	 [255,0,255]	 [255,255,255]

## 1.I.2 Bits de poids forts et faibles

On appelle :

- Bits de poids forts : les bits « de gauche » **01001110** d'un nombre codé en binaire (en gras, les deux premiers bits de poids fort)
- Bits de poids faible : les bits « de droite » 010011**10** d'un nombre codé en binaire (en gras, les deux derniers bits de poids faible)

Remarquons que 11111111 et 11110000 correspondent à des entiers assez proches : 255 et 240 et que tous les nombres commençant par 1111\_\_\_\_ codés sur 8 bits sont contenus dans l'intervalle [240,255].

Dans mon projet « Traitement d'images », on joue avec ces bits de poids fort et faible et on peut voir que les bits de poids fort définissent quasiment entièrement l'image alors que les bits de poids faible ne viennent que finaliser la qualité par des nuances de couleurs.

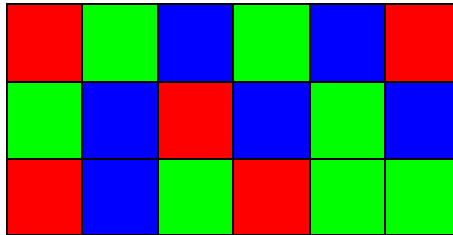
Voici une image originale et la même image où, à gauche, on ne garde que les 4 bits de poids fort, et à droite, que les 4 bits de poids faible.



## 1.II. Représentation matricielle

### 1.II.1 Images en couleur

Imaginons l'image suivante composée de 3 lignes et 6 colonnes de pixels. Pour l'exemple, nous n'avons pris que 3 couleurs, Rouge, Vert et Bleu (RGB). En fait, toutes les couleurs sont une combinaison de ces 3 couleurs.



Ainsi, on peut en fait représenter cette image avec le tableau suivant :

R=255 G=0 B=0	R=0 G=255 B=0	R=0 G=0 B=255	R=0 G=255 B=0	R=0 G=0 B=255	R=255 G=0 B=0
R=0 G=255 B=0	R=0 G=0 B=255	R=255 G=0 B=0	R=0 G=0 B=255	R=0 G=255 B=0	R=0 G=0 B=255
R=255 G=0 B=0	R=0 G=0 B=255	R=0 G=255 B=0	R=255 G=0 B=0	R=0 G=255 B=0	R=0 G=255 B=0

Sous Python, nous allons manipuler des matrices d'un format assez particulier, mais qui finalement se représente exactement comme des matrices. On peut donc représenter l'image ci-dessus par la matrice ci-dessous de 3 lignes et 6 colonnes :

$$\begin{bmatrix} [255 & 0 & 0] & [0 & 255 & 0] & [0 & 0 & 255] & [0 & 255 & 0] & [0 & 0 & 255] & [255 & 0 & 0] \\ [0 & 255 & 0] & [0 & 0 & 255] & [255 & 0 & 0] & [0 & 0 & 255] & [0 & 255 & 0] & [0 & 0 & 255] \\ [255 & 0 & 0] & [0 & 0 & 255] & [0 & 255 & 0] & [255 & 0 & 0] & [0 & 255 & 0] & [0 & 255 & 0] \end{bmatrix}$$

Attention, lignes et colonnes se lisent ainsi :

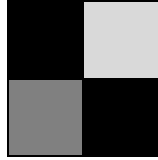
Colonnes  
→

↓  
Lignes

l=0 c=0	l=0 c=1	l=0 c=2	l=0 c=3	l=0 c=4	l=0 c=5
l=1 c=0	l=1 c=1	l=1 c=2	l=1 c=3	l=1 c=4	l=1 c=5
l=2 c=0	l=2 c=1	l=2 c=2	l=2 c=3	l=2 c=4	l=2 c=5

## 1.II.2 Images en nuances de gris

Une couleur grise est représentée par un triplet où les trois valeurs R, G et B sont égales. Ainsi, les 4 pixels ci-dessous se représente avec la matrice associée :



R=0	R=200
G=0	G=200
B=0	B=200
R=100	R=0
G=100	G=0
B=100	B=0

Remarque : il est possible de ne mettre qu'un entier au lieu d'un triplet, mais je reste sur la version triplet.

Il est donc assez simple de transformer une image en couleur en image nuances de gris en calculant la valeur à imposer à  $R=G=B=Val$ . On pourra procéder ainsi :

$$Val = \text{int}(rR + gG + bB) \in [0,255] \text{ avec } r + g + b = 1 \quad (r, g, b) \in [0,1]^3$$

Avec  $r$ ,  $g$  et  $b$  des coefficients à choisir pour mettre en avant une couleur par rapport aux autres. Par exemple :



$r = 0.33 \ g = 0.34 \ b = 0.33$ Pas de couleur privilégiée	$r = 0 \ g = 0 \ b = 1$ Accent sur le bleu	$r = 1 \ g = 0 \ b = 0$ Accent sur le rouge

Attention : risque d'overflow si vous calculez par exemple  $Val = (R + G + B)/3$  et que  $R + G + B > 255$ . Il faut donc bien passer par la formule avec les produits  $rR$ ,  $gG$  et  $bB$  car chaque produit d'un entier avec un flottant crée un flottant.

```
import numpy as np
Mat = np.array([200,200,200],dtype='uint8')
Res = Mat[0]+Mat[1]+Mat[2]
print(Res)
```

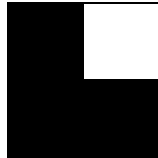
```
>>> (executing file "<tmp 1>")
<tmp 1>:4: RuntimeWarning: overflow
encountered in ubyte_scalars
Res = Mat[0]+Mat[1]+Mat[2]
88
```

### 1.II.3 Image en noir et blanc

On pourra ensuite transformer l'image en noir et blancs en utilisant un seuil, à 127 par exemple (on peut évidemment jouer dessus pour garder une certaine partie de l'image):

$$\begin{cases} Val = 0 \text{ si } Val < Seuil \\ Val = 255 \text{ si } Val \geq Seuil \end{cases}$$

Voici ce que cela donnerait sur l'exemple vu précédemment :



R=0	R=255
G=0	G=255
B=0	B=255
R=0	R=0
G=0	G=0
B=0	B=0

Remarque : il est ici aussi possible de ne mettre qu'un entier au lieu d'un triplet, mais je reste sur la version triplet.

Et en partant d'une photo en nuances de gris :



<i>Seuil = 50</i>	<i>Seuil = 127</i>	<i>Seuil = 200</i>



## 1.III. Prise en main des images avec Python

### 1.III.1 Ouverture d'une image au format BMP

#### 1.III.1.a Introduction

Je tiens à préciser ici que pendant plusieurs années, j'ai utilisé le module `imread` de `scipy` en écrivant :

```
import scipy.misc as scm
image = scm.imread("nom.bmp")
```

Malheureusement, `imread` de `scipy` est devenu obsolète récemment, et la solution consistait, quand cela fonctionnait, à downgrader `scipy` en écrivant « `conda install scipy==1.1.0` », avec son lot de soucis.

J'ai découvert qu'il était possible d'utiliser le module `matplotlib` pour réaliser l'ouverture des images, alors je ne parlerai plus que de cela !

#### 1.III.1.b Ouverture de l'image

Soit une image `bmp` de nom « `Nom.bmp` », sur le bureau par exemple.

Travaillons en chemin relatif. Pour que cela fonctionne, il faut :

- Enregistrer votre fichier python dans le même répertoire que votre image au format `bmp`, ici sur le bureau
- Activer l'option « `Change directory when executing file` » du menu « `run` » pour que `pyzo` cherche au bon endroit lors de l'exécution du fichier
- Exécuter votre code avec la touche `F5` afin que `pyzo` cherche...

Importer le module `matplotlib` comme vous en avez l'habitude :

```
import matplotlib.pyplot as plt
```

Pour créer l'objet « image » dans python, et y importer une image `BMP`, il faut écrire :

```
image = plt.imread("Nom.bmp")
```

L'image devient un array : 

```
>>> type(image)
<class 'numpy.ndarray'>
```

Chaque entier de chaque pixel est un entier codé sur 8 bits (`uint8`) : 

```
>>> type(image[0,0,0])
<class 'numpy.uint8'>
```

En cas de problèmes, préciser le chemin absolu en doublant les `\` en les remplaçant par un `/` :

```
image = plt.imread("C:\\Users\\denis\\Desktop\\Nom.bmp")
image = plt.imread("C:/Users/denis/Desktop/Nom.bmp")
```

A savoir: `imread` envoie une image « `read only` », non modifiable donc. Pour la modifier, la copier d'abord : `im = image.copy()`

### 1.III.1.c Gestion des erreurs

Le module imread de matplotlib fait appel à un module nommé Pillow. Au départ, seules les images de format png fonctionnaient. Il est donc possible que vous obteniez l'erreur suivante :

**Only know how to handle extensions: ['png']; with Pillow installed matplotlib can handle more images**

La solution est simple si vous avez la main sur l'ordinateur 😊, essayez :

```
conda install pillow
```

Ceci installera ou mettra à jour Pillow, et les BMP seront acceptés.

Si cela ne fonctionne pas, essayez :

```
upgrade pip (pas forcément nécessaire)
pip install pillow
```

Sinon, il est possible d'ouvrir les images au format png. Dans ce cas, ouvrez-les images bmp avec Paint par exemple, puis enregistrez-les au format png. Attention alors après import sous Python, ce ne seront plus des triplets mais des quadruplets qui seront composés de flottants 32 bits entre 0 et 1, le 4<sup>e</sup> codant une transparence (cela arrive avec les BMP, cf paragraphe suivant)... Je vous propose une fonction qui convertira alors l'image afin de la remettre dans le format d'entiers 8 bits :

```
def conversion(Im) :
    Nl,Nc = Im.shape[0:2]
    Im = Im[:,:,:3]
    Im_convert = np.zeros((Nl,Nc,3),dtype = np.uint8)
    for c in range (Nc) :
        for l in range(Nl) :
            R,G,B = Im[l,c]
            R = int(R*255)
            G = int(G*255)
            B = int(B*255)
            Im_convert[l,c] = [R,G,B]
    return Im_convert
```

### 1.III.1.d Conversion d'une image RGBA

Avec scipy, il n'y avait jamais de problème. Toutes les images importées étaient des array de l lignes et c colonnes, dans lesquels chaque case contenait des triplets.

Avec matplotlib, selon l'image bmp importée, il arrive que des quadruplets soient créés. On parle de forma RGBA au lieu de format RGB. A est un entier qui définit une transparence. Nous ne travaillerons pas sur ce A. c'est pourquoi, à chaque import, je vous propose d'écrire :

```
image = image[:,:,:3]
```

Cette instruction permettra de fonctionner, que l'image importée comporte des triplets comme des quadruplets, pour n'obtenir que des triplets RGB 😊

### 1.III.1.e Le format array

Vous avez à disposition un cours dans les bases Python qui vous permettra d'en savoir plus sur cet outil. Je vais toutefois rappeler ici les choses importantes à ce propos.

#### 1.III.1.e.i Types

L'ouverture d'une image avec `imread` renvoie un array contenant des array de 3 (triplets RGB) ou 4 (quadruplets RGBA) valeurs contenant des entiers codés sur 8 bits (entre 0 et 255).

```
>>> type(im)
<class 'numpy.ndarray'>
```

```
>>> type(im[0,0])
<class 'numpy.ndarray'>
```

```
>>> type(im[0,0,0])
<class 'numpy.uint8'>
```

#### 1.III.1.e.ii Overflow

Attention, danger ! Les entiers étant codés sur 8 bits, il n'est pas possible de sortir de l'intervalle [0,255]. Exemple :

```
>>> M = np.array([200,200],dtype='uint8')
>>> M[0]+M[1]
<console>:1: RuntimeWarning: overflow encountered in ubyte_scalars
144
```

En effet,  $200+200=400$ , c'est 256 (0 en 8 bits)+144... Cette erreur s'appelle « Overflow » et nous en reparlerons dans le chapitre sur le codage des nombres.

Si le résultat de l'opération sort de l'intervalle [0,255] et si cela est voulu, il faudra veiller à limiter sa valeur, par exemple 300 sera associé à 255, -10 sera associé à 0 :

```
>>> M = 255*np.ones([50,50,3],dtype='uint8')
>>> R,G,B = M[0,0]
>>> S = R+G
<console>:1: RuntimeWarning: overflow encountered in ubyte_scalars
>>> S
254
>>> R,G,B = int(R),int(G),int(B)
>>> S = R+G
>>> S = max(0,min(255,S))
>>> S
255
```

A savoir : Quand on effectue des opérations entre deux types différents sous Python, l'opération est réalisée en transformant le format le plus contraignant en l'autre. Illustrations :

```
>>> M[0]+200          >>> M[0]+200.0
400                  400.0

>>> type(M[0]+200)    >>> type(M[0]+200.0)
<class 'numpy.int32'> <class 'numpy.float64'>
```

Lorsque vous aurez à effectuer des sommes ou différences en TP entre des valeurs R, G et B des pixels, il faudra veiller à ce que l'un au moins des deux termes de l'opération soit transformé en flottant (par exemple). S'il faut ensuite affecter le résultat dans une image, il faudra le retransformer en int.

### 1.III.1.e.iii Récupération de valeurs

Avec les « array », il faut utiliser la syntaxe `M[i,j]` et non `M[i][j]` comme on le ferait avec des listes de listes ! Mais oui, tant que l'on n'utilise pas les slices, les deux fonctionnent de manière équivalente... Ce n'est toutefois pas une bonne idée de prendre des mauvaises habitudes. J'illustre la différence de comportement dans le tableau suivant :

	Array	Liste de listes
A partir de	<pre>&gt;&gt;&gt; M = np.array([[1,2],[3,4]])</pre>	<pre>&gt;&gt;&gt; M = [[1,2],[3,4]]</pre>
Récupération d'une valeur	<pre>&gt;&gt;&gt; M[0,0] 1  &gt;&gt;&gt; M[0][0] 1</pre> <p>Certes, les deux fonctionnent, mais il ne faut pas utiliser <code>[]</code> !!! La preuve avec les slices ci-dessous</p>	<pre>&gt;&gt;&gt; M[0,0] Traceback (most recent call last):   File "&lt;console&gt;", line 1, in &lt;module&gt; TypeError: list indices must be integers or slices, not tuple  &gt;&gt;&gt; M[0][0] 1</pre>
Récupération de plusieurs valeurs <b>Slices</b>	<pre>&gt;&gt;&gt; M[:,0] array([1, 3])  &gt;&gt;&gt; M[:,][0] array([1, 2])</pre> <p>Voilà le problème, on croit que l'on va récupérer la première colonne, mais on récupère la première ligne..... En effet, <code>M[:,]</code> renvoie <code>M</code> entière !</p> <pre>&gt;&gt;&gt; M[:,] array([[1, 2],        [3, 4]])</pre>	<pre>&gt;&gt;&gt; M[:,0] Traceback (most recent call last):   File "&lt;console&gt;", line 1, in &lt;module&gt; TypeError: list indices must be integers or slices, not tuple  &gt;&gt;&gt; M[:,][0] [1, 2]  &gt;&gt;&gt; M[0,:][0] [1, 2]  Quoi qu'il arrive, cela ne fonctionne pas ! En effet, <code>M[:,]</code> renvoie <code>M</code> entière !  &gt;&gt;&gt; M[:,] [[1, 2], [3, 4]]  Solution si besoin : &gt;&gt;&gt; [M[0][i] for i in range(2)] [1, 2]  &gt;&gt;&gt; [M[i][0] for i in range(2)] [1, 3]</pre>
<b>Conclusion</b>	<b>Avec les array on utilise <code>M[,,,]</code> Les slices fonctionnent</b>	<b>Avec les listes on utilise <code>M[][][]</code> Les slices ne fonctionnent pas</b>

Avec les array, on peut par ailleurs récupérer des portions d'array avec les slices, ce qui peut être pratique :

```
>>> M = np.array([[1,2,3,4],[5,6,7,8]])

>>> M[:,2:2]
array([[1, 2],
       [5, 6]])
```

Ou encore avec un array de dimension 3 (chaque terme est un doublet) :

```
>>> M = np.array([[[1,1.1],[2,2.1]],[[3,3.1],[4,4.1]]])

>>> M[:, :, 0]
array([[1., 2.],
       [3., 4.]])

>>> M[:, :, 1]
array([[1.1, 2.1],
       [3.1, 4.1]])
```

### 1.III.1.e.iv Ouverture/enregistrement d'un array

Pour ouvrir une image sauvegardée en format npy, et présente dans le répertoire de travail, on écrira :

```
import numpy as np
Image = np.load("Nom.npy")
```

On peut sauvegarder une image en format « npy » avec le module numpy. Pour cela, il suffit d'écrire :

```
import numpy as np
np.save('Nom', image)
```

Avec :

- « image » l'image importée ou créée
- « Nom » le nom de l'image, sans extension !

Les images sont alors enregistrées avec l'extension « .npy » dans votre répertoire de travail (où selon les ordi, quelque part... il faudra chercher).

### 1.III.1.e.v Avantage des slices

Même si on attendra souvent de vous des doubles boucles for « pour faire de l'algorithmique », sachez que l'utilisation de slices améliore grandement les temps d'exécution des algorithmes.

Exemple :

```
N = 1000
import numpy as np
A = np.ones([N,N])

from time import perf_counter as tps
t1 = tps()
for l in range(N):
    for c in range(N):
        A[l,c] = 0

t2 = tps()
Temps_1 = t2-t1
print(Temps_1)

t1 = tps()
A[:, :] = 1
t2 = tps()
Temps_2 = t2-t1
print(Temps_2)

k = Temps_1/Temps_2
print(k)
```

0.45546730000000935  
0.0017794000000032183  
255.96678655264222

L'utilisation améliore grandement le temps d'exécution (facteur 255 dans cet exemple).

## 1.III.2 Affichage d'une image

Commencer par importer la librairie suivante.

```
import matplotlib.pyplot as plt
```

Créer ensuite une fonction qui affiche les images :

```
def f_affiche(image):  
    plt.figure()  
    plt.imshow(image)  
    plt.axis('off')  
    plt.show()  
    plt.pause(0.00001)
```

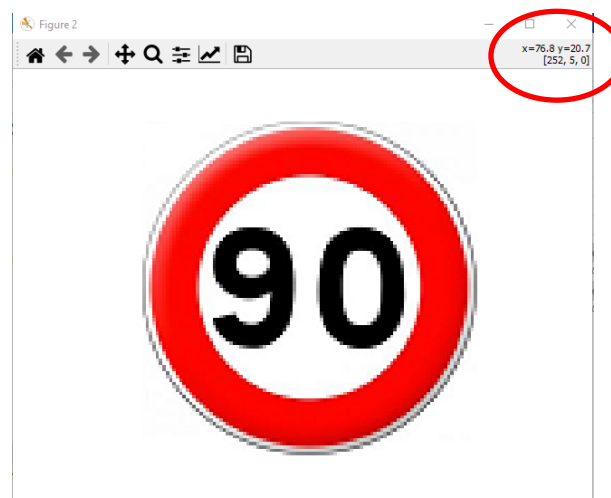
Remarque : Pause permet d'afficher les images en cours de route dans des programmes qui traitent plusieurs images d'affilée.

Enfin, pour afficher l'image, il suffit d'écrire :

```
f_affiche(image)
```

## 1.III.3 Exploration graphique

Lorsque vous affichez une image avec Python, remarquez quelque chose d'intéressant en passant la souris sur différents pixels :



On voit la ligne (y), la colonne (x), et le contenu du pixel, le triplet [R,G,B] 😊

Remarque : cela peut être localisé autre part !

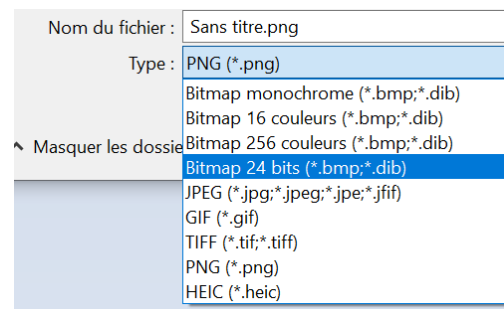
### 1.III.4 Enregistrement d'une image

Lorsque l'image `image` existe sous python (format array), on peut la sauvegarder de deux manières :

Instruction	<code>plt.imsave("Image.bmp", image)</code>	<code>plt.savefig("Image.png")</code>
Remarques	Il suffit d'avoir l'array, sans l'afficher Formats png et bmp possibles Image enregistrée dans ses dimensions en pixels (lignes, colonnes)	Nécessite d'ouvrir l'image et de l'afficher avant de l'enregistrer Format png ok, bmp non pris en charge Image redimensionnée en 640x480
Elle sera alors enregistrée sous le nom « Image.xxx » dans le répertoire de travail Rappel : au format png, les images sont stockées avec des flottants en 32 bits compris en 0 et 1		

### 1.III.5 Traitement des autres formats

Si vous possédez une image d'un autre format, je vous recommande de l'ouvrir avec Paint, ou de réaliser une capture d'écran que vous collerez dans Paint, puis de l'enregistrer au format BMP 24 bits.



## 1.IV. Travail sur les images

### 1.IV.1 Parcours d'une image

Je vous recommande d'utiliser les indices `l` (ligne) et `c` (colonne) pour les boucles `for` afin d'être lisibles des correcteurs de vos copies. Evitez `(i,j)` ou `(y,x)`.

Voici un exemple de rédaction à privilégier :

```
def f(im):  
    Nl, Nc = im.shape[0:2]  
    for c in range(Nc):  
        for l in range(Nl):  
            ...
```

Remarque : lorsque rien ne vous est imposé, il peut être très intéressant d'utiliser les slices pour modifier une portion d'image (gain de temps important) :

```
image[:, :, :] = [127, 127, 127]
```

## 1.IV.2 Modification de pixels

Après avoir chargé et copié ou créé une image sous python, on peut appeler la liste  $[R, G, B]$  d'un pixel en écrivant :

```
Pixel = image[l,c]
```

ATTENTION : Les variables  $l$  et  $c$  doivent être de type `Int`.

Le nombre de lignes et colonnes de *image* est obtenu ainsi :

```
Nl,Nc = image.shape[0:2]
```

On obtient finalement les 3 couleurs RGB du pixel concerné en écrivant :

<code>R = Pixel[0]</code> <code>G = Pixel[1]</code> <code>B = Pixel[2]</code>	<code>R = image[l,c,0]</code> <code>G = image[l,c,1]</code> <code>B = image[l,c,2]</code>	<code>R,G,B = Pixel</code> <code>R,G,B = image[l,c,:]</code>
---	---	---

Remarques importantes :

- Attention : ne pas égaliser deux images, qui pointeraient comme des listes, vers les mêmes adresses mémoire. On pourra copier une image en passant par la même procédure que pour une liste, par exemple en écrivant : `image_2 = np.copy(image_1)`
- Le test d'égalité de deux array ne fonctionne pas ! (cf ci-dessous)

```
>>> a=np.array([1,2,3])
>>> b=np.array([1,2,3])

>>> a==b
array([ True,  True,  True], dtype=bool)

>>> a==b==True
Traceback (most recent call last):
  File "<console>", line 1, in <module>
ValueError: The truth value of an array with more th
an one element is ambiguous. Use a.any() or a.all()
```

## 1.IV.3 Création d'une image noire ou blanche

Voici le code pour créer une image de toute pièce :

```
# Import des librairies
import numpy as np

# Création d'une image vierge
Nc = 100
Nl = 100
Image_Noire = np.zeros((Nl,Nc,3),dtype='uint8')
Image_Blanche = 255*np.ones((Nl,Nc,3),dtype='uint8')
```

On crée un array numpy rempli de 0 (image noire) ou de 1 (image blanche), de dimensions  $Nl$  lignes,  $Nc$  colonnes, formée de triplets (3), et dont le type est `uint8`. En effet, les entiers doivent être codés en « entiers » sur 8 bits...

J'ai mis beaucoup de temps à savoir quoi mettre en `dtype` afin que ça fonctionne !



## 1.V. Convolution

### 1.V.1 Principe

Soit une matrice carrée  $K$  appelée noyau, de  $n$  lignes et  $n$  colonnes, par exemple :

$$K = \begin{bmatrix} K_{11} & K_{12} & K_{13} \\ K_{21} & K_{22} & K_{23} \\ K_{31} & K_{32} & K_{33} \end{bmatrix}$$

Soit une image  $Im$  de  $Nl$  lignes et  $Nc$  colonnes de pixels RGB, par exemple :

$$Im = \begin{bmatrix} P_{11} & P_{12} & P_{13} & P_{14} & P_{15} \\ P_{21} & P_{22} & P_{23} & P_{24} & P_{25} \\ P_{31} & P_{32} & P_{33} & P_{34} & P_{35} \\ P_{41} & P_{42} & P_{43} & P_{44} & P_{45} \\ P_{51} & P_{52} & P_{53} & P_{54} & P_{55} \end{bmatrix} ; \quad P_{ij} = [R_{ij}, G_{ij}, B_{ij}]$$

Appliquer un produit de convolution  $Im * K$  consiste à remplacer chaque pixel  $P_{ij}$  intérieur de  $Im$  par le résultat d'un calcul afin d'obtenir une nouvelle image  $Im'$  :

$$Im' = \begin{bmatrix} P'_{11} & P'_{12} & P'_{13} & P'_{14} & P'_{15} \\ P'_{21} & P'_{22} & P'_{23} & P'_{24} & P'_{25} \\ P'_{31} & P'_{32} & P'_{33} & P'_{34} & P'_{35} \\ P'_{41} & P'_{42} & P'_{43} & P'_{44} & P'_{45} \\ P'_{51} & P'_{52} & P'_{53} & P'_{54} & P'_{55} \end{bmatrix} ; \quad P'_{ij} = [R'_{ij}, G'_{ij}, B'_{ij}]$$

Intérieurs

En définissant  $k = \left\lfloor \frac{n}{2} \right\rfloor$ , les pixels intérieurs sont les pixels non compris dans les  $k$  premières et dernières lignes et colonnes. Pour tous les pixels « intérieurs », on applique donc un produit de convolution (somme des produits termes à termes des deux matrices), par exemple pour  $k = 3$  :

$$P'_{ij} = K * M_{ij} ; \quad K = \begin{bmatrix} K_{11} & K_{12} & K_{13} \\ K_{21} & K_{22} & K_{23} \\ K_{31} & K_{32} & K_{33} \end{bmatrix} ; \quad M_{ij} = \begin{bmatrix} P_{i-1,j-1} & P_{i-1,j} & P_{i-1,j+1} \\ P_{i,j-1} & P_{i,j} & P_{i,j+1} \\ P_{i+1,j-1} & P_{i+1,j} & P_{i+1,j+1} \end{bmatrix}$$

$$P'_{ij} = K_{11}P_{i-1,j-1} + K_{12}P_{i-1,j} + K_{13}P_{i-1,j+1} + K_{21}P_{i,j-1} + K_{22}P_{i,j} + K_{23}P_{i,j+1} + K_{31}P_{i+1,j-1} + K_{32}P_{i+1,j} + K_{33}P_{i+1,j+1}$$

Avec :

$$K_{ij}P_{ij} = K_{ij}[R_{ij}, G_{ij}, B_{ij}] = [K_{ij}R_{ij}, K_{ij}G_{ij}, K_{ij}B_{ij}]$$

Sous Python, en définissant une matrice locale  $Im\_Loc$  de mêmes dimensions que  $K$  centrée sur le pixel à modifier, cela revient à remplacer ce pixel par la somme (un triplet RGB) de tous les triplets RGB du produit termes à termes des entiers de  $K$  avec les triplets de  $Im\_Loc$ .

### 1.V.2 Gestion des bords

Nous en resterons à des applications simples, les bords ( $k$  premières et dernières lignes et colonnes) ne seront pas modifiés.

### 1.V.3 Normalisation

La normalisation consiste à diviser chaque terme de  $K$  par la somme de tous ses termes, ce qui permet d'avoir une matrice  $K$  normalisée dont la somme des termes vaut 1. Cela permet de garder une moyenne des pixels avant et après convolution au même niveau de couleur.

Attention : On ne normalise pas une matrice si la somme de ses termes vaut 0 !

### 1.V.4 Limitation des valeurs résultat







Après convolution, des valeurs de R, G et B peuvent sortir de l'intervalle  $[0,255]$ . Pour éviter l'overflow, on limite donc le résultat du produit de convolution :

- à 0 si le R, G ou B est inférieur à 0
- à 255 si le R, G ou B est supérieur à 255

## 1.V.5 Quelques exemples

Image originale ([source](#)) :



Moyenneur – Floutage	$K = \frac{1}{9} \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$	
Repoussage	$K = \begin{bmatrix} -2 & -1 & 0 \\ -1 & 1 & 1 \\ 0 & 1 & 2 \end{bmatrix}$	
Laplacien - Contours	$K = \begin{bmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{bmatrix}$	
Réhausseur	$K = \begin{bmatrix} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 0 \end{bmatrix}$	
Gaussien 3x3	$K = \frac{1}{16} \begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix}$ $V = \begin{bmatrix} 1 \\ 2 \\ 1 \end{bmatrix} \Rightarrow K = \frac{V * V^T}{16}$	
Gaussien 5x5	$K = \frac{1}{256} \begin{bmatrix} 1 & 4 & 6 & 4 & 1 \\ 4 & 16 & 24 & 16 & 4 \\ 6 & 24 & 36 & 24 & 6 \\ 4 & 16 & 24 & 16 & 4 \\ 1 & 4 & 6 & 4 & 1 \end{bmatrix}$ $V = \begin{bmatrix} 1 \\ 4 \\ 6 \\ 4 \\ 1 \end{bmatrix} \Rightarrow K = \frac{V * V^T}{256}$	

Les images ci-dessus sont issues d'un calcul que j'ai réalisé sur l'image source

## 1.VI. Transformations

Voyons enfin comment appliquer un algorithme de rotation, réduction ou agrandissement d'une image.

### 1.VI.1 Rotation

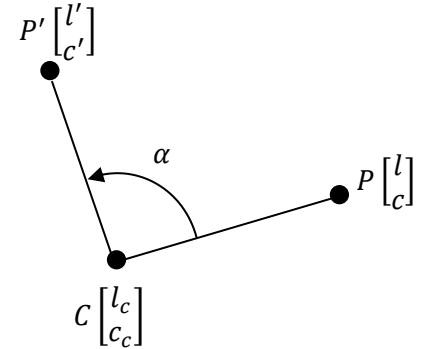
A chaque pixel est associé un couple d'entiers de sa ligne et sa colonne  $(l, c)$ .

$P(l, c)$  est un pixel de l'image à transformer.

On définit  $C(l_c, c_c)$  le centre de la rotation et  $\alpha$  l'angle de rotation souhaité.

$P'$  est le pixel image du pixel  $P$  par la rotation d'angle  $\alpha$ .

$P$  est le pixel antécédent du pixel  $P'$  par la rotation d'angle  $\alpha$ .

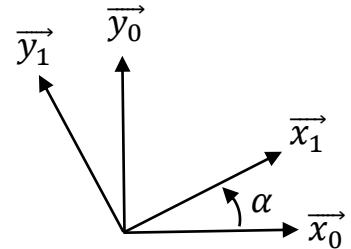


#### 1.VI.1.a Matrice de rotation

Soit la rotation d'angle  $\alpha$  entre les bases 0 et 1 :

$$\vec{x}_1 = \cos \alpha \vec{x}_0 + \sin \alpha \vec{y}_0 = \begin{bmatrix} \cos \alpha \\ \sin \alpha \end{bmatrix}$$

$$\vec{y}_1 = -\sin \alpha \vec{x}_0 + \cos \alpha \vec{y}_0 = \begin{bmatrix} -\sin \alpha \\ \cos \alpha \end{bmatrix}$$



On définit la matrice de rotation  $R_\alpha$  suivante :

$$R_\alpha = \begin{bmatrix} \cos \alpha & -\sin \alpha \\ \sin \alpha & \cos \alpha \end{bmatrix}$$

Remarquez que :

$$R_\alpha \vec{x}_0 = \begin{bmatrix} \cos \alpha & -\sin \alpha \\ \sin \alpha & \cos \alpha \end{bmatrix} \begin{bmatrix} 1 \\ 0 \end{bmatrix} = \begin{bmatrix} \cos \alpha \\ \sin \alpha \end{bmatrix} = \vec{x}_1$$

$$R_\alpha \vec{y}_0 = \begin{bmatrix} \cos \alpha & -\sin \alpha \\ \sin \alpha & \cos \alpha \end{bmatrix} \begin{bmatrix} 0 \\ 1 \end{bmatrix} = \begin{bmatrix} -\sin \alpha \\ \cos \alpha \end{bmatrix} = \vec{y}_1$$

Et d'une manière générale, on obtient le vecteur  $\vec{v}$ , image du vecteur  $\vec{u}$  par la rotation d'un angle  $\alpha$ , par le produit :

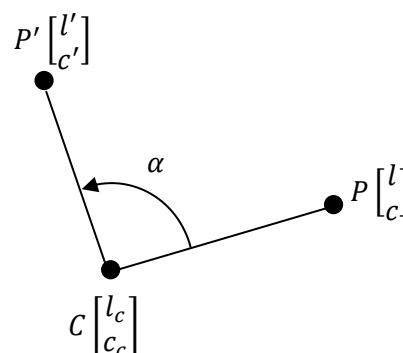
$$\vec{v} = R_\alpha \vec{u}$$

### 1.VI.1.b Algorithme naïf

Le principe de cet algorithme consiste à déterminer le couple  $P'(l', c')$  obtenu à partir du pixel de l'image source de couple  $P(l, c)$  par rotation de centre  $P(l_c, c_c)$  d'un angle  $\alpha$  :

Définissons le vecteur  $\overrightarrow{CP} = \begin{bmatrix} l - l_c \\ c - c_c \end{bmatrix}$ . On obtient l'image  $\overrightarrow{CP'}$  de ce vecteur par rotation d'angle  $\alpha$  et de centre  $C$  avec le produit :

$$\overrightarrow{CP'} = R_\alpha \overrightarrow{CP} = \begin{bmatrix} \cos \alpha & -\sin \alpha \\ \sin \alpha & \cos \alpha \end{bmatrix} \begin{bmatrix} l - l_c \\ c - c_c \end{bmatrix} = \begin{bmatrix} a \\ b \end{bmatrix}$$



On trouve alors les coordonnées de l'image  $P'$  de  $P$  :  $\begin{bmatrix} l' \\ c' \end{bmatrix} = \begin{bmatrix} l_c + a \\ c_c + b \end{bmatrix}$ . Toutefois, les lignes et colonnes étant des entiers, on trouve le couple  $(l', c')$  en utilisant la partie entière de  $a$  et  $b$  :

$$\begin{bmatrix} l' \\ c' \end{bmatrix} = \begin{bmatrix} l_c + \lfloor a \rfloor \\ c_c + \lfloor b \rfloor \end{bmatrix}$$

On vérifie alors que le couple  $(l', c')$  appartient bien à l'image issue de la rotation que nous nommerons  $Im\_Rot$ , car il est possible de sortir de l'image.

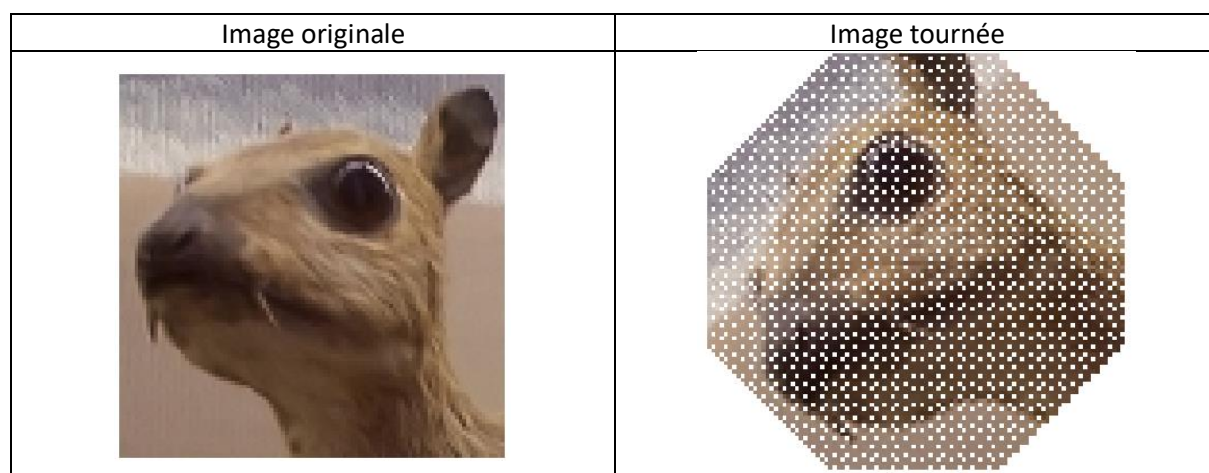
On applique enfin dans la nouvelle image au couple  $(l', c')$ , la couleur du pixel de l'image source en  $(l, c)$ .

$$Im'(l', c') = Im(l, c)$$

L'inconvénient de cette méthode est de faire apparaître des trous dans l'image résultat, car tous les pixels de l'image source n'aboutissent pas sur tous les pixels de l'image cible à cause des arrondis entiers des pixels d'arrivée.

Remarque : Les pixels dont l'antécédent est en dehors de l'image originale auront une couleur arbitrairement choisie au préalable.

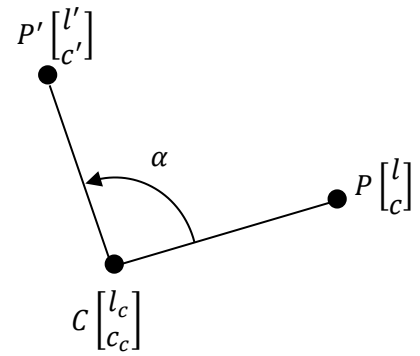
Voici un exemple d'application où au départ, tous les pixels sont fixés à la couleur blanche, et on applique une rotation de centre  $[50, 50]$  sur une image  $[100, 100]$  et d'angle  $45^\circ$  :



## 1.VI.1.c Algorithme inverse

### 1.VI.1.c.i Méthode classique

Pour améliorer le résultat précédent, l'idée de cet algorithme est de réaliser l'opération inverse. Pour chaque pixel de l'image résultat à remplir (aucun ne sera oublié) de couple  $P'(l', c')$ , on cherche le pixel de couple  $P(l, c)$  de l'image source correspondante par rotation d'angle  $-\alpha$  (s'il est bien dans l'image, la rotation pouvant conduire à sortir de l'image source), et on applique au pixel résultat la couleur du pixel source associé.



En reprenant le même formalisme qu'au paragraphe précédent, on a  $\overrightarrow{CP'} = \begin{bmatrix} l' - lc \\ c' - cc \end{bmatrix}$ . On obtient l'antécédent  $\overrightarrow{CP}$  de ce vecteur par rotation d'angle  $\alpha$  et de centre  $C$  par une rotation d'angle  $-\alpha$  et de centre  $C$  et le produit :

$$\overrightarrow{CP} = R_{-\alpha} \overrightarrow{CP'} = \begin{bmatrix} \cos \alpha & \sin \alpha \\ -\sin \alpha & \cos \alpha \end{bmatrix} \begin{bmatrix} l' - lc \\ c' - cc \end{bmatrix} = \begin{bmatrix} a \\ b \end{bmatrix}$$

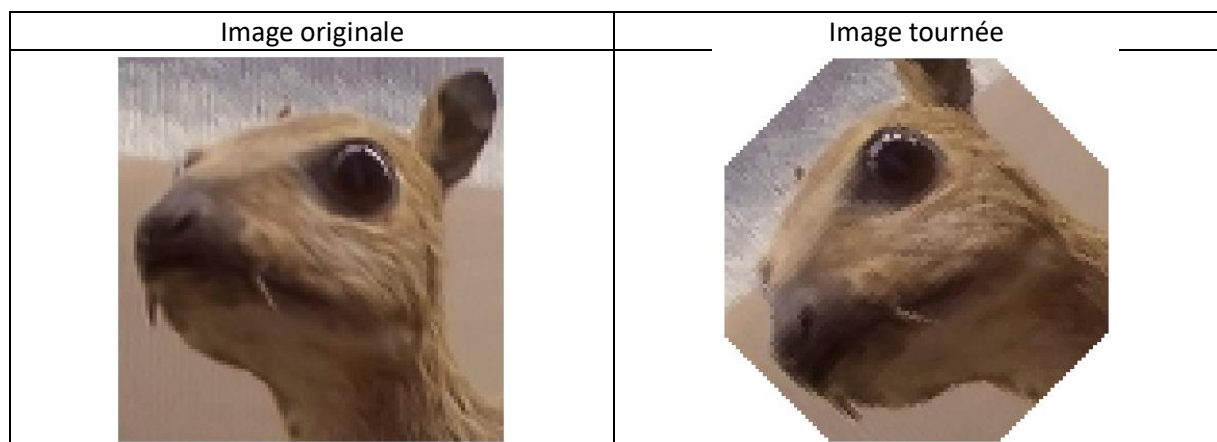
On trouve alors les coordonnées de l'antécédent  $P$  de  $P' : \begin{bmatrix} l \\ c \end{bmatrix} = \begin{bmatrix} lc + a \\ cc + b \end{bmatrix}$ . Toutefois, les lignes et colonnes étant des entiers, on trouve le couple  $(l, c)$  en utilisant la partie entière de  $a$  et  $b$  :

$$\begin{bmatrix} l \\ c \end{bmatrix} = \begin{bmatrix} lc + \lfloor a \rfloor \\ cc + \lfloor b \rfloor \end{bmatrix}$$

On vérifie alors que le couple  $(l, c)$  appartient bien à l'image originale et on applique dans la nouvelle image au couple  $(l', c')$ , la couleur du pixel de l'image source en  $(l, c)$ .

$$Im'(l', c') = Im(l, c)$$

On remarque quand même que l'image obtenue n'est pas parfaite. Pour améliorer ce résultat, on applique généralement au pixel résultat une moyenne des pixels qui avoisinent le pixel de couple  $(l, c)$  de l'image source, en appliquant par exemple au préalable une convolution d'effet moyennneur.



Cette méthode conserve les valeurs des pixels entre chaque image.

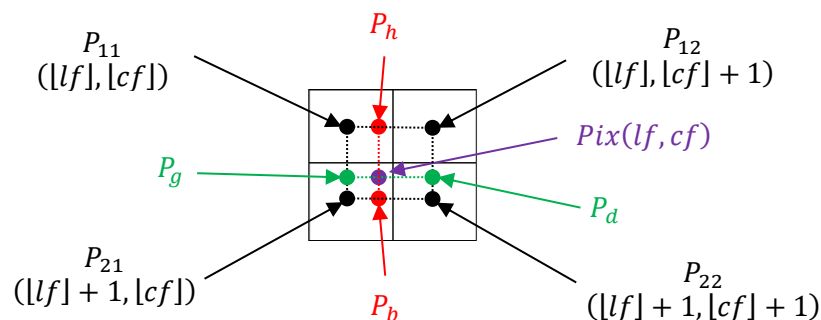
### 1.VI.1.c.ii Méthode bilinéaire

Il est possible de déterminer la valeur du triplet de  $P'$  par interpolation bilinéaire sur les 4 pixels entourant les coordonnées flottantes  $(lf, cf)$  de  $P$  afin d'être plus précis :  $\begin{bmatrix} lf \\ cf \end{bmatrix} = \begin{bmatrix} lc + a \\ cc + b \end{bmatrix}$

Introduisons une fonction  $\text{lin}(x, x_1, x_2, y_1, y_2)$  qui associe à  $x$  dans  $[x_1, x_2]$  une valeur de  $y$  dans  $[y_1, y_2]$  de manière linéaire ainsi :  $y = y_1 + \frac{x-x_1}{x_2-x_1}(y_2 - y_1)$

Supposons maintenant qu'une fonction  $\text{lin\_pix}(x, x_1, x_2, \text{pix1}, \text{pix2})$  renvoie par interpolation linéaire le triplet RGB  $\text{pix}$  calculé linéairement pour  $x$  dans  $[x_1, x_2]$  donnant  $\text{pix}$  entre  $\text{pix1}$  et  $\text{pix2}$  (application de  $\text{lin}$  sur les 3 valeurs RGB en faisant attention aux problèmes d'overflow) .

Introduisons les 4 pixels (triplets RGB) entourant  $(lf, cf)$  :



On pourra alors au choix calculer la valeur  $\text{Pix}$  par bilinéarité ainsi :

Linéarisation par lignes puis par colonnes	Linéarisation par colonnes puis par lignes
$\text{Pb} = \text{lin\_pix}(cf, [cf], [cf] + 1, P_{21}, P_{22})$ $\text{Ph} = \text{lin\_pix}(cf, [cf], [cf] + 1, P_{11}, P_{12})$ $\text{Pix} = \text{lin\_pix}(lf, [lf], [lf] + 1, \text{Pb}, \text{Ph})$	$\text{Pg} = \text{lin\_pix}(lf, [lf], [lf] + 1, P_{11}, P_{21})$ $\text{Pd} = \text{lin\_pix}(lf, [lf], [lf] + 1, P_{12}, P_{22})$ $\text{Pix} = \text{lin\_pix}(cf, [cf], [cf] + 1, \text{Pg}, \text{Pd})$



On notera que cette méthode ne conserve pas les valeurs des pixels entre chaque image. Ainsi, la rotation d'une image en noir et blanc créera des pixels en nuances de gris.



## 1.VI.2 Réduction

Pour réduire une image, il faut supprimer des lignes et des colonnes. Pour réduire  $n$  fois le nombre de lignes et/ou colonnes, on peut par exemple :

### 1.VI.2.a Suppression de lignes/colonnes

L'idée consiste à prendre 1 pixels tous les  $n$  pixels. Cette pratique peut faire apparaître un effet d'aliasing (lignes visibles). Il convient pour l'éliminer d'appliquer un filtre moyenneur par exemple, avant de supprimer les lignes et/ou colonnes. Exemple : Division du nombre de lignes et colonnes par 3 (division de nombre de pixel par 9) :

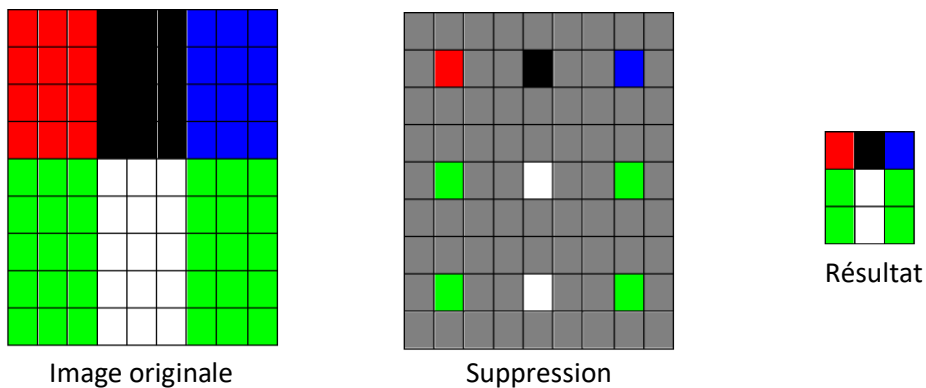







Illustration avec Python en gardant soit 1 ligne sur 5, soit 1 colonne sur 5, soit les deux transformations à la suite :

Image originale (100x100)	
	
Réduction des lignes (20x100)	Réduction des colonnes (100x20)
	
Résultat (20x20)	
	



## 1.VI.2.b Moyenne par paquets

On peut aussi imaginer récupérer une valeur moyenne par paquets de  $n \times n$  pixels, et ce tous les  $n$  pixels en ligne et en colonne. Prenons l'image précédente et appliquons-lui ce principe par bloc de  $3 \times 3$  en supposant que les pixels ont les valeurs suivantes (j'ai respecté les couleurs exactes dans mes illustrations) :

	Rouge	Vert	Bleu	Noir	Blanc
R	255	0	0	0	255
G	0	255	0	0	255
B	0	0	255	0	255

On va garder les lignes et colonnes 2, 5 et 8 et appliquer un filtre moyenneur :

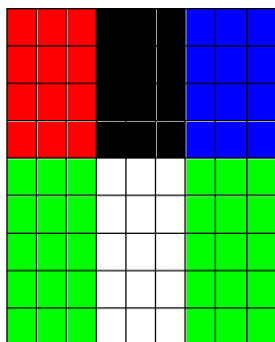
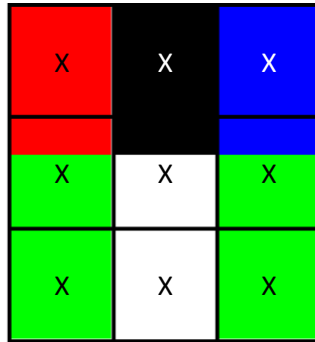


Image originale



Pixels à garder/modifier

R=255 G=0 B=0		R=0 G=0 B=255
R=85 G=170 B=0	R=85 G=85 B=85	R=0 G=170 B=85
R=0 G=255 B=0	R=255 G=255 B=255	R=0 G=255 B=0

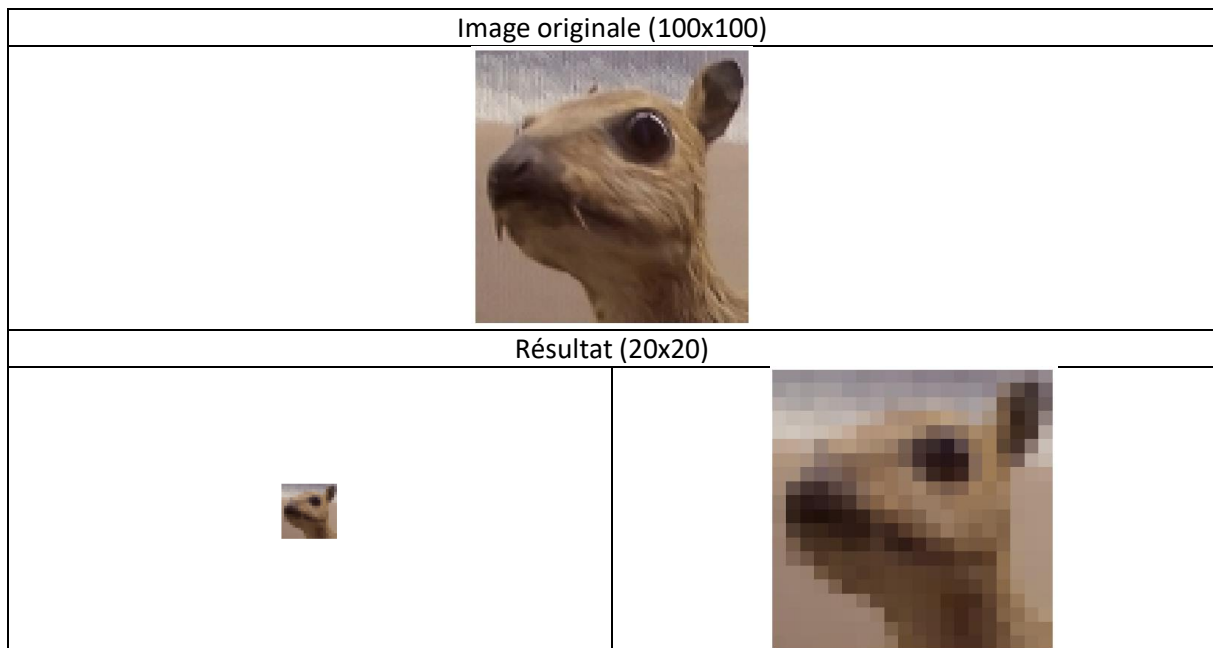
Résultat intermédiaire



Résultat

Remarque : Cette méthode appliquée à de grandes images donne des résultats plus concluants que la précédente par une transition plus douce d'une couleur à l'autre.

Illustration avec Python procédant par paquets de  $5 \times 5$  :



Rappel du résultat précédent :



## 1.VI.3 Agrandissement

Le principe de l'agrandissement consiste à ajouter des lignes et/ou colonnes à l'image, et à déterminer la couleur à imposer aux nouveaux pixels par interpolation. Citons trois méthodes.

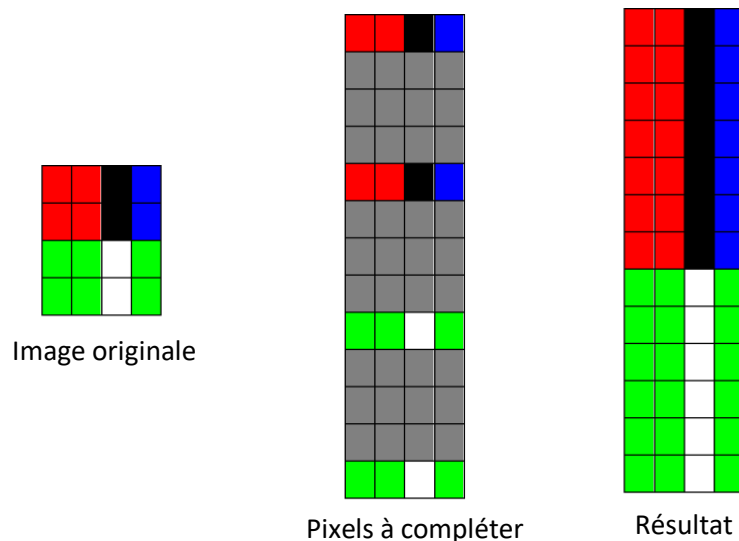
### 1.VI.3.a Interpolation par plus proche voisin

On prend la valeur du pixel le plus proche avec une priorité (gauche, droite, haut, bas) s'il y a égalité de distance.

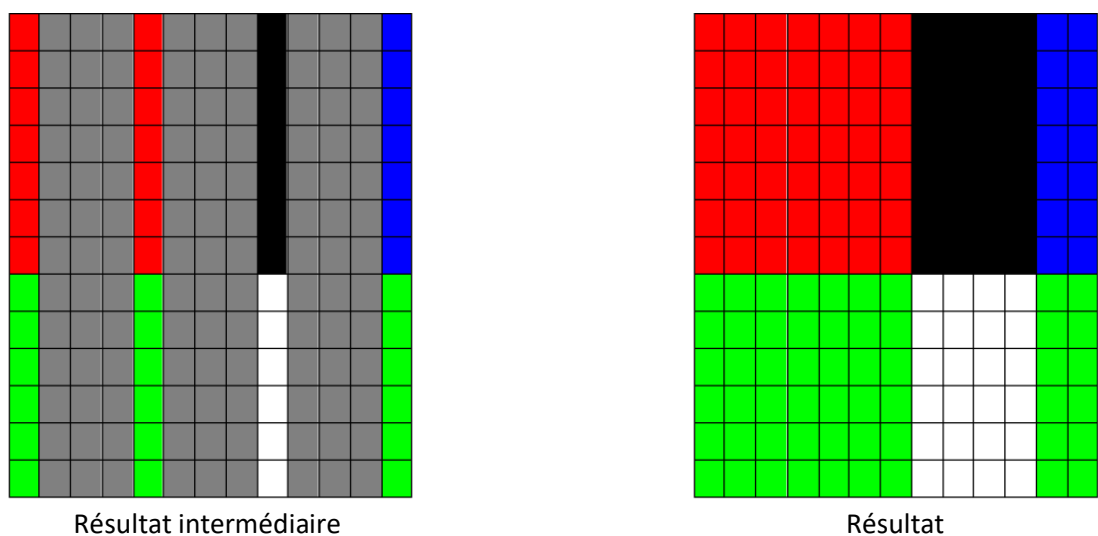
Exemple de stratégie lorsque l'on ajoute 3 lignes entre chaque ligne :

- La première prend les valeurs de celle qui est la plus proche, par exemple au-dessus
- La dernière prend les valeurs de celle qui est la plus proche, par exemple en dessous
- Celle du milieu prend la valeur du dessus (choix)

Illustration :



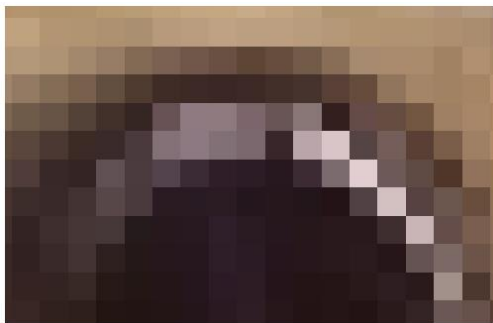


Puis si l'on ajoute 3 colonnes entre chaque colonne avec une priorité à gauche :



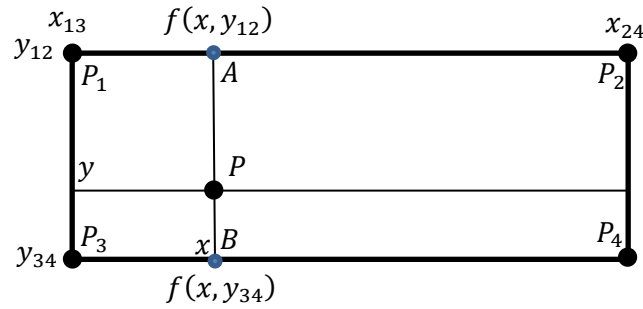
Oui, il pourrait être intéressant d'aller plus loin en appliquant des calculs de type moyenne plutôt que de réaliser des choix de priorité.

Application au lapin avec un agrandissement x5, soit ajout de 4 lignes/colonnes entre chaque ligne/colonne :

Image originale (100x100)	
	
Résultat (496x496) (Echelle non respectée)	
	

### 1.VI.3.b Interpolation bilinéaire

La couleur attribuée au pixel  $P$  dépend des 4 triplets  $P_i$  qui l'entourent dans l'image d'origine.



Supposons que l'on ajoute  $L$  lignes et  $C$  colonnes de pixels entre deux lignes et deux colonnes, alors :

- Chaque ligne aura une ordonnée  $y$  comprise dans  $[y_{12}, y_{34}]$  espacées de  $\frac{|y_{12}-y_{34}|}{L+1}$
- Chaque colonne aura une abscisse  $x$  comprise dans  $[x_{13}, x_{24}]$  espacées de  $\frac{|x_{13}-x_{24}|}{C+1}$

Les pixels  $P_i$  sont des triplets RGB qui vont servir à déterminer par interpolation linéaire les valeurs des triplets des nouveaux pixels ajoutés.

On définit la fonction  $f(x, y)$  la fonction qui renvoie le pixel RGB à attribuer à un point de coordonnées  $(x, y)$ . On a :

$$\begin{cases} f(x_{13}, y_{12}) = P_1 \\ f(x_{24}, y_{12}) = P_2 \\ f(x_{13}, y_{34}) = P_3 \\ f(x_{24}, y_{34}) = P_4 \end{cases}$$

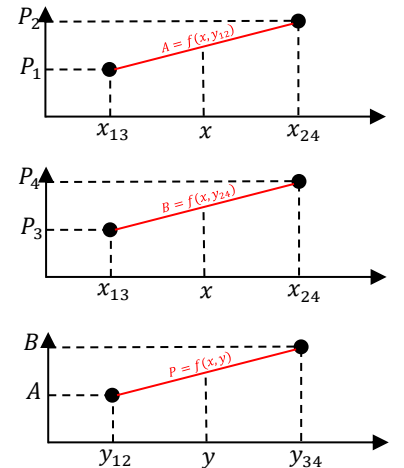
Par interpolation linéaire, on a :

$$A = f(x, y_{12}) = \frac{x_{24} - x}{x_{24} - x_{13}} P_1 + \frac{x - x_{13}}{x_{24} - x_{13}} P_2$$

$$B = f(x, y_{34}) = \frac{x_{24} - x}{x_{24} - x_{13}} P_3 + \frac{x - x_{13}}{x_{24} - x_{13}} P_4$$

Soit, par interpolation entre  $A$  et  $B$  :

$$f(x, y) = \frac{y_{34} - y}{y_{34} - y_{12}} A + \frac{y - y_{12}}{y_{34} - y_{12}} B$$



$$\begin{aligned}
f(x,y) &= \frac{y_{34} - y}{y_{34} - y_{12}} \left( \frac{x_{24} - x}{x_{24} - x_{13}} P_1 + \frac{x - x_{13}}{x_{24} - x_{13}} P_2 \right) + \frac{y - y_{12}}{y_{34} - y_{12}} \left( \frac{x_{24} - x}{x_{24} - x_{13}} P_3 + \frac{x - x_{13}}{x_{24} - x_{13}} P_4 \right) \\
&= \frac{1}{(y_{34} - y_{12})(x_{24} - x_{13})} [(y_{34} - y)[(x_{24} - x)P_1 + (x - x_{13})P_2] \\
&\quad + (y - y_{12})[(x_{24} - x)P_3 + (x - x_{13})P_4]] \\
&= \frac{1}{(y_{34} - y_{12})(x_{24} - x_{13})} \left[ (y_{34} - y) \begin{bmatrix} (x_{24} - x) & (x - x_{13}) \end{bmatrix} \begin{bmatrix} P_1 \\ P_2 \end{bmatrix} \right. \\
&\quad \left. + (y - y_{12}) \begin{bmatrix} (x_{24} - x) & (x - x_{13}) \end{bmatrix} \begin{bmatrix} P_3 \\ P_4 \end{bmatrix} \right]
\end{aligned}$$

$$f(x,y) = \frac{1}{(y_{34} - y_{12})(x_{24} - x_{13})} [y_{34} - y \quad y - y_{12}] \begin{bmatrix} P_1 & P_2 \\ P_3 & P_4 \end{bmatrix} \begin{bmatrix} x_{24} - x \\ x - x_{13} \end{bmatrix}$$

Cela revient à chercher  $f(x,y)$  sous la forme :  $f(x,y) = ax + by + cxy + d$

Exemple : Si  $P$  est à  $1/3$  du bord gauche et  $1/3$  du bord inférieur (comme sur l'image) :

$$P = \frac{2}{3}B + \frac{1}{3}A \quad ; \quad A = \frac{2}{3}P_1 + \frac{1}{3}P_2 \quad ; \quad B = \frac{2}{3}P_3 + \frac{1}{3}P_4$$

Application à l'exemple précédent :

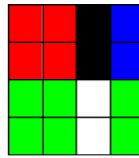
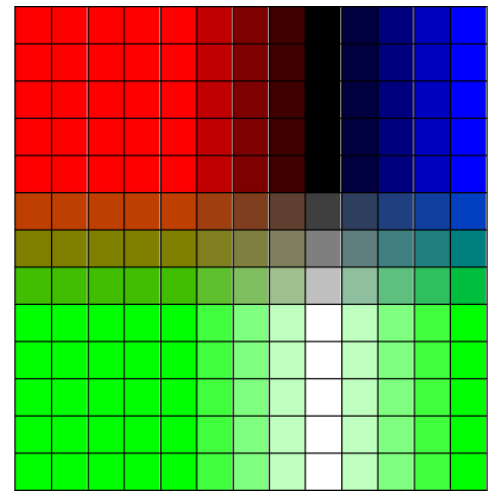


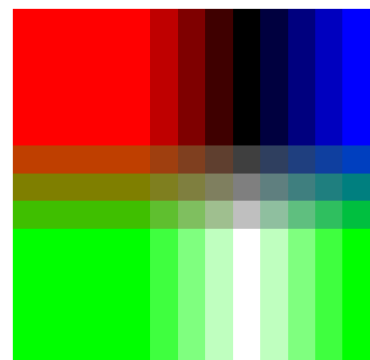
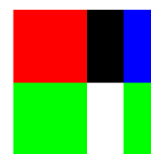
Image originale

R=255	R=255	R=255	R=255	R=255	R=191	R=127	R=63	R=0	R=0	R=0	R=0	R=0
G=0	G=0	G=0	G=0	G=0	G=0	G=0	G=0	G=0	G=0	G=0	G=0	G=0
B=0	B=0	B=0	B=0	B=0	B=0	B=0	B=0	B=0	B=63	B=127	B=191	B=255
R=255	R=255	R=255	R=255	R=255	R=191	R=127	R=63	R=0	R=0	R=0	R=0	R=0
G=0	G=0	G=0	G=0	G=0	G=0	G=0	G=0	G=0	G=0	G=0	G=0	G=0
B=0	B=0	B=0	B=0	B=0	B=0	B=0	B=0	B=0	B=63	B=127	B=191	B=255
R=255	R=255	R=255	R=255	R=255	R=191	R=127	R=63	R=0	R=0	R=0	R=0	R=0
G=0	G=0	G=0	G=0	G=0	G=0	G=0	G=0	G=0	G=0	G=0	G=0	G=0
B=0	B=0	B=0	B=0	B=0	B=0	B=0	B=0	B=0	B=63	B=127	B=191	B=255
R=255	R=255	R=255	R=255	R=255	R=191	R=127	R=63	R=0	R=0	R=0	R=0	R=0
G=0	G=0	G=0	G=0	G=0	G=0	G=0	G=0	G=0	G=0	G=0	G=0	G=0
B=0	B=0	B=0	B=0	B=0	B=0	B=0	B=0	B=0	B=63	B=127	B=191	B=255
R=191	R=191	R=191	R=191	R=191	R=159	R=127	R=95	R=63	R=47	R=31	R=15	R=0
G=63	G=63	G=63	G=63	G=63	G=63	G=63	G=63	G=63	G=63	G=63	G=63	G=63
B=0	B=0	B=0	B=0	B=0	B=15	B=31	B=47	B=63	B=95	B=127	B=159	B=191
R=127	R=127	R=127	R=127	R=127	R=127	R=127	R=127	R=127	R=95	R=63	R=31	R=0
G=127	G=127	G=127	G=127	G=127	G=127	G=127	G=127	G=127	G=127	G=127	G=127	G=127
B=0	B=0	B=0	B=0	B=0	B=0	B=31	B=63	B=95	B=127	B=127	B=127	B=127
R=63	R=63	R=63	R=63	R=63	R=95	R=127	R=159	R=191	R=143	R=95	R=47	R=0
G=191	G=191	G=191	G=191	G=191	G=191	G=191	G=191	G=191	G=191	G=191	G=191	G=191
B=0	B=0	B=0	B=0	B=0	B=47	B=95	B=143	B=191	B=159	B=127	B=95	B=63
R=0	R=0	R=0	R=0	R=0	R=63	R=127	R=191	R=255	R=191	R=127	R=63	R=0
G=255	G=255	G=255	G=255	G=255	G=255	G=255	G=255	G=255	G=255	G=255	G=255	G=255
B=0	B=0	B=0	B=0	B=0	B=63	B=127	B=191	B=255	B=191	B=127	B=63	B=0
R=0	R=0	R=0	R=0	R=0	R=63	R=127	R=191	R=255	R=191	R=127	R=63	R=0
G=255	G=255	G=255	G=255	G=255	G=255	G=255	G=255	G=255	G=255	G=255	G=255	G=255
B=0	B=0	B=0	B=0	B=0	B=63	B=127	B=191	B=255	B=191	B=127	B=63	B=0
R=0	R=0	R=0	R=0	R=0	R=63	R=127	R=191	R=255	R=191	R=127	R=63	R=0
G=255	G=255	G=255	G=255	G=255	G=255	G=255	G=255	G=255	G=255	G=255	G=255	G=255
B=0	B=0	B=0	B=0	B=0	B=63	B=127	B=191	B=255	B=191	B=127	B=63	B=0
R=0	R=0	R=0	R=0	R=0	R=63	R=127	R=191	R=255	R=191	R=127	R=63	R=0
G=255	G=255	G=255	G=255	G=255	G=255	G=255	G=255	G=255	G=255	G=255	G=255	G=255
B=0	B=0	B=0	B=0	B=0	B=63	B=127	B=191	B=255	B=191	B=127	B=63	B=0
R=0	R=0	R=0	R=0	R=0	R=63	R=127	R=191	R=255	R=191	R=127	R=63	R=0
G=255	G=255	G=255	G=255	G=255	G=255	G=255	G=255	G=255	G=255	G=255	G=255	G=255
B=0	B=0	B=0	B=0	B=0	B=63	B=127	B=191	B=255	B=191	B=127	B=63	B=0

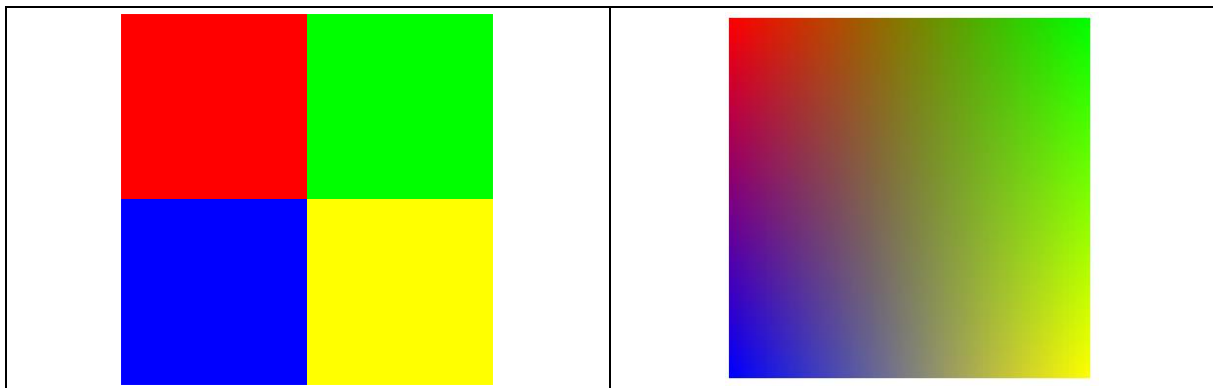
Détail des calculs



Résultat



Exemple d'application sur une image à 4 pixels à laquelle on ajoute 100 lignes et colonnes :



Application au lapin avec un agrandissement x5, soit ajout de 4 lignes/colonnes entre chaque ligne/colonne :



### 1.VI.3.c Interpolation bicubique

Plus couteuse en temps de calcul, l'interpolation bicubique prend en compte 16 pixels (4x4) au lieu de 4 (2x2) comme dans l'interpolation bilinéaire. Sans aller dans les détails comme je l'ai fait pour l'interpolation bilinéaire, le principe consiste à écrire :  $f(x, y) = \sum_{i=0}^3 \sum_{j=0}^3 a_{ij} x^i y^j$

L'interpolation va utiliser, en plus des valeurs des 4 pixels avoisinants, des valeurs plus éloignées (12 pixels de plus) **afin de tenir compte, en plus de l'interpolation bilinéaire, des évolutions des valeurs des pixels selon l'espace.**

En supposant que  $f$  et ses dérivées partielles  $f_x$ ,  $f_y$  et  $f_{xy}$  sont connues aux 4 coins, il faut déterminer les 16 coefficients  $a_{ij}$  pour connaître  $f$ .

### 1.VI.3.d Comparaison

Comparons les résultats des algorithmes par plus proches voisins et interpolation bilinéaire pour une multiplication par 5 (ajout de 4 lignes/colonnes entre chaque ligne et colonne :

Image originale (100x100)	
	
Plus proches voisins	
	
Interpolation bilinéaire	
	