



IMT Mines Albi-Carmaux
École Mines-Télécom

Algorithmique & programmation : manipulation d'images


Paul Gaborit

Centre de Génie Industriel – IMT Mines Albi
2022/2023


- Connaître et utiliser les images *matricielles* (ou *bitmap*).
- Connaître et utiliser le *modèle de couleur RVB*.
- Utiliser une *bibliothèque externe* de manipulation d'images.
- Concevoir un *algorithme de zoom*.

- En informatique, les images sont :



vectérielles composées d'objets géométriques (segments, polygones, courbes...) caractérisés par des attributs (forme, position, couleurs, flous...), elles supportent un changement d'échelle en restant parfaites.

 Formats : **ps**, **pdf**, **svg**, **ai**

matricielles constituées d'une matrice de pixels où chaque pixel est défini par une couleur, elle sont adaptées aux photos mais plus on les agrandit, plus les pixels sont visibles.

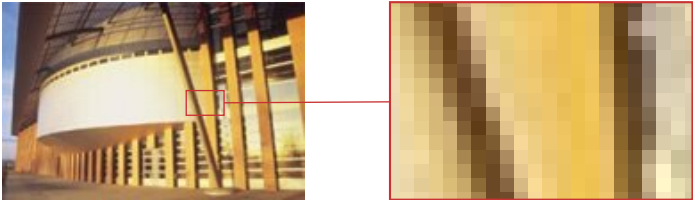
 Formats sans perte : **bmp**, **png**, **pnm**, **ppm**, **tiff**

 Formats avec perte : **gif**, **jpg**

-  Les images vectorielles peuvent contenir des images matricielles.
-  Sauf exception, ne *jamaïs* convertir une image vectorielle en image matricielle pour l'intégrer dans un document.

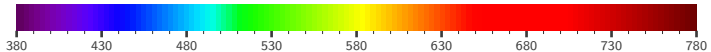
- Dans la suite de ce sujet, nous allons nous intéresser aux images matricielles.

- Une **image matricielle** est une matrice ou un tableau de pixels.
- Chaque **pixel** (abréviation de *picture element*) ou point élémentaire est un carré (parfois un rectangle) entièrement couvert par une unique couleur.



- La **définition** d'une image matricielle indique le nombre de pixels qui la composent. Exemple du Full HD : 1920×1080 (largeur \times hauteur).
- La **résolution** d'une image matricielle, lorsqu'elle est connue, indique la densité souhaitée/prévue de pixels ou de points par unité de longueur. Ex : 72 ppp = points par pouce (ou *dpi* = *dots per inch*).
 - 👉 En fait, la résolution réelle dépend de la taille finale de l'image produite.




- Physiquement, une couleur est liée à une *longueur d'onde* de la lumière :






- Mais la couleur perçue n'est souvent qu'une *illusion* obtenue par mélange de plusieurs couleurs.
- 👉 L'illusion produite dépend des capteurs utilisés : l'œil d'un être humain, d'un oiseau, d'un taureau, les capteurs d'un appareil photo, d'un satellite.
- 👉 Même chez l'humain, la perception des couleurs varie d'un individu à l'autre... et parfois de manière importante.
Daltoniens en France $\approx 8\%$ des hommes et 0.4% des femmes.

- Pour créer l'illusion, deux types de synthèse de couleurs :

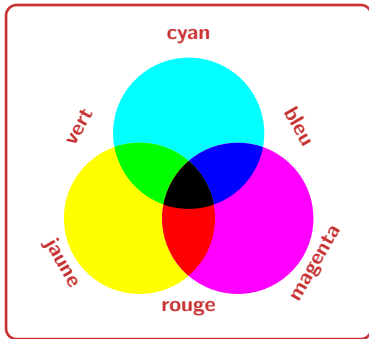
soustractive applique au blanc des filtres successifs qui soustraient de plus en plus de couleur.

- La peinture et l'imprimerie reposent sur cette synthèse.
- Couleurs primaires :  cyan  magenta  jaune.
... et le noir car les encres ne sont pas des filtres parfaits.

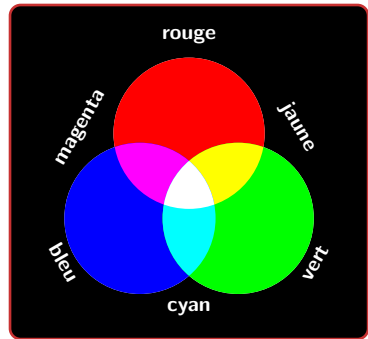
additive sur fond noir, projette différentes couleurs qui s'ajoutent en se mélangeant.

- L'éclairage au théâtre, les écrans de télévision, d'ordinateurs ou de terminaux mobiles reposent sur cette synthèse.
- Couleurs primaires :  rouge  vert  bleu

👉 Les couleurs primaires choisies pour la synthèse additive correspondent aux longueurs d'ondes perçues par les trois types de cônes de l'œil humain (les bâtonnets perçoivent la luminosité).

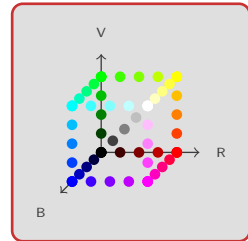


Synthèse soustractive



Synthèse additive

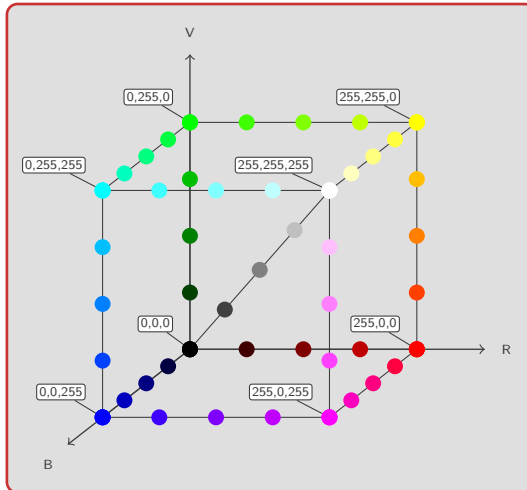
- La couleur d'un pixel s'exprime généralement via le modèle RVB (ou RGB) pour rouge, vert, bleu. Chacune des composantes peut varier entre 0 à 255 proposant ainsi une palette de 16 millions de couleurs ($256^3 = 16\,777\,216$).



Cube des couleurs RVB

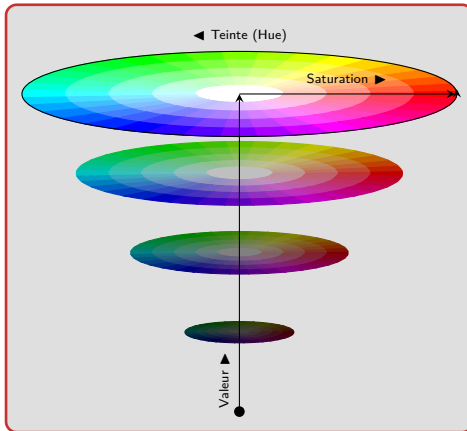
- Il existe d'autres modèles de couleur : TLS, TSV, CMJ(N)... La quasi totalité utilise un espace à 3 dimensions adapté aux dimensions de la perception humaine des couleurs
(Nb de dimensions – taureaux : 1, certains oiseaux : 4, satellites : 6 et plus)
- Algorithmiquement et informatiquement, nous considérerons qu'un pixel est une structure composée de trois champs nommés respectivement rouge, vert et bleu.

✚ Pour en savoir plus [Rouge vert bleu](#) [sur Wikipedia](#).



*Cube des couleurs RVB
(avec les valeurs des trois composantes pour les principales couleurs).*

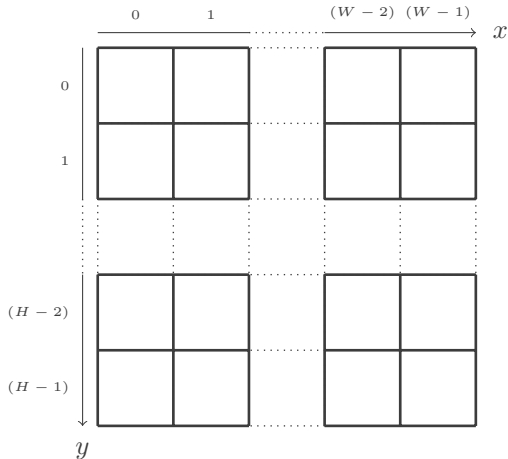
Le modèle TSV (ou HSV) utilise la teinte (*hue*), la saturation et la valeur pour définir un cône des couleurs :



- La teinte varie du **rouge** (0°) au **rouge** (360°) en passant par le **jaune** (60°), le **vert** (120°), le **cyan** (180°), le **bleu** (240°) et le **magenta** (300°).
- La valeur varie de 0 (noir) à 1 (maximum de luminosité).
- La saturation varie de 0 (sans couleur) à 1 (couleur saturée).

👉 Pour en savoir plus [Teinte Saturation Valeur](#) [sur Wikipedia](#).

- En informatique, pour des raisons historiques, le système de coordonnées conventionnel place l'origine en haut à gauche des images.
- Exemple pour une image de définition $W \times H$:



- Le module Python `simple_image` permet de créer des images matricielles, de lire et modifier leurs pixels et d'enregistrer ou lire ces images via des fichiers en différents formats (PNG, JPG...).
- 👉 Ce module écrit par nos soins est en fait une surcouche de simplification du module `PIL.Image` fournie par la bibliothèque Python `Pillow` (*Python Imaging Library*).
- 👉 Dans tous les exercices de ce module, nous vous conseillons d'enregistrer les images générées au format PNG puisque c'est un format compressé *sans* perte d'information (contrairement au format JPG qui dénature l'image pour mieux la compresser).

- Pour utiliser le module `simple_image.py` :

- 1 Créer un environnement virtuel Python3 nommé `venv-images` :

```
%% python3 -m venv venv-images
```

```
C:> python -m venv venv-images
```

- 2 Activer cet environnement virtuel :

```
%% source ./venv-images/bin/activate  
(venv-images) %%
```

```
C:> .\venv-images\Scripts\activate.bat  
(venv-images) C:>
```



Noter le changement de *prompt* !

- Pour utiliser le module `simple_image.py` (suite) :

- 3 Installer via `pip` (le *Python package installer*) les bibliothèques Python `Pillow` et `packaging` dont dépend `simple_image.py` :

```
(venv-images) %% pip install Pillow packaging
```

```
(venv-images) C:> pip install Pillow packaging
```

- 4 Extraire le fichier `simple_image.py` et l'exécuter pour vérifier que tout marche bien.

```
(venv-images) %% chmod a+x ./simple_image.py
```

```
(venv-images) %% ./simple_image.py
```

```
(venv-images) C:> python simple_image.py
```

 le fichier `simple_image.py` est une pièce-jointe de ce fichier PDF.

 Ne faire ces opérations qu'**une seule fois** !

Seule l'activation **2** peut être à refaire à l'ouverture d'un nouveau terminal (mais Visual Studio Code propose souvent de le faire).

- Une fois créé un environnement virtuel, il est possible d'indiquer à VS Code qu'il doit l'utiliser :
 - 1 En bas à droite de la fenêtre VS Code, cliquer sur la version de Python affichée.
 - 2 Cliquer sur « + Enter interpreter path... ».
 - 3 Cliquer sur « Find... ».
 - 4 Naviguer pour aller chercher `.../venv-images/bin/python3` (la plupart du temps, VS Code vous proposera tout seul le bon chemin).

- Importer la classe `Image` depuis le module `simple_image.py` :

```
from simple_image import Image
```

- Ensuite toutes les opérations se feront via la classe `Image` et ses objets.

- La méthode `Image.new(width, height)` crée une nouvelle image (noire) en mémoire :

```
im1 = Image.new(width=640, height=480)
```

Les paramètres obligatoires `width` et `height` indiquent respectivement la largeur et la hauteur en pixels de l'image (sa définition).

- La méthode `Image.read(filepath)` permet de charger en mémoire une image stockée dans un fichier :

```
im2 = Image.read("chemin/du/fichier/image.jpg")
```

- Les attributs non modifiables `width`, `height` et `definition` d'un objet `Image` indiquent respectivement sa largeur, sa hauteur et définition (sous la forme d'un tuple) :

```
print(f"Définition de l'image: {im2.width}x{im2.height}")  
print(f"Définition de l'image: {im2.definition[0]}x{im2.definition[1]}")
```

- La méthode `save(filepath)` permet d'enregistrer un objet `Image` dans un fichier (le format du fichier image sera choisi selon l'extension du nom de fichier) :

```
im2.save("chemin/du/nouveau/fichier/image.png")
```

- La méthode `get_color(coordinate)` permet de récupérer la couleur d'un pixel depuis un objet `Image` :

```
color = im.get_color((x, y))
```

- La méthode `set_color(coordinate, color)` permet de modifier la couleur d'un pixel d'un objet `Image` :

```
red_color = (255, 0, 0) # rouge  
im.set_color((x, y), red_color)
```

⚙ Concernant les paramètres :


- `coordinate` est un couple d'entiers `(x, y)` permettant de spécifier les coordonnées du pixel concerné.
 - 👉 Si les coordonnées sont en dehors de l'image, une exception de type `ValueError` est levée.
- `color` est un triplet d'entiers `(r, v, b)` : la composante rouge, la composante verte et la composante bleue. Les valeurs de ces composantes sont nécessairement entre 0 et 255.
 - 👉 Si l'une des composantes est en dehors de cet intervalle, une exception de type `ValueError` est levée.

Consignes


- 1 Récupérer les images `originale.png`, `image1.png` et `image2.png` et les stocker dans un sous-répertoire nommé `images`.


 l'image `originale.png` est une pièce-jointe de ce fichier PDF.

 l'image `image1.png` est une pièce-jointe de ce fichier PDF.


 l'image `image2.png` est une pièce-jointe de ce fichier PDF.

- 2 Récupérer le fichier source `exercice-1.py` (ainsi que le module `simple_image.py`).

 le script `exercice-1.py` est une pièce-jointe de ce fichier PDF.

 le module `simple_image.py` est une pièce-jointe de ce fichier PDF.

- 3 Exécuter ce script, regarder ce qu'il produit et expliquer son fonctionnement.

 Pour voir les images, utiliser la commande `display` (sur Linux), Gimp (logiciel libre de dessin), votre navigateur ou même VS Code.

```
(venv-images) %% display images/image1.png
```

```
#!/usr/bin/env python3
from simple_image import Image

def transform(im1):
    im2 = Image.new(width=im1.height, height=im1.width)
    for x in range(im1.width):
        for y in range(im1.height):
            color = im1.get_color((x, y))
            im2.set_color((y, x), color)
    return im2

if __name__ == "__main__":
    # Image.trace = False
    im_orig = Image.read("images/image1.png")
    im_res = transform(im_orig)
    im_res.save("images/resultat-ex1.png")
```

Balise anti-divulgâchage



Solution à venir !

image lue



image1.png

image produite



resultat-ex1.png

- 👉 Il semble qu'on applique une rotation à 90° vers la gauche puis une symétrie d'axe horizontal sur l'image lue... mais la composition d'une rotation et d'une symétrie est aussi une symétrie !
- 👉 C'est donc une symétrie autour de l'axe allant du coin en haut à gauche au coin en bas à droite de l'image.
- 👉 C'est l'axe $x = y$ (en tenant compte du système de coordonnées des images).

```
#!/usr/bin/env python3
from simple_image import Image

def transform(im1):
    im2 = Image.new(width=im1.height, height=im1.width)
    for x in range(im1.width):
        for y in range(im1.height):
            color = im1.get_color((x, y))
            im2.set_color((y, x), color)
    return im2

if __name__ == "__main__":
    # Image.trace = False
    im_orig = Image.read("images/image1.png")
    im_res = transform(im_orig)
    im_res.save("images/resultat-ex1.png")
```

- ⚙ Dans la fonction `transform()`, l'image `im2` est créée en inversant la largeur et la hauteur de `im1` (normal pour une symétrie).
- ⚙ Les deux boucles imbriquées (sur x et y) permettent de parcourir tous les pixels de l'image `im1`.
- ⚙ On lit la couleur du pixel de coordonnées (x, y) dans `im1` puis on modifie la couleur du pixel de coordonnées (y, x) dans l'image `im2`. C'est cette **inversion de coordonnées** qui réalise la symétrie d'axe $x = y$!

- 💬 Concevoir le script `gradient.py` qui crée une nouvelle image 256x100 contenant un dégradé du blanc (à gauche) au vert (à droite). Le script enregistrera l'image produite dans un fichier nommé `degrade.png`.
- 🔧 Vérifier que l'image produite correspond aux spécifications.
- 🔧 Remplacer 256 par 800 et vérifier que le programme fonctionne toujours correctement.
- ⚙️ Pour vérifier que le programme fonctionne bien, afficher la couleur de trois pixels : un 1^{er} complètement à gauche, un 2^e au milieu et un 3^e complètement à droite de l'image.

Balise anti-divulgâchage



Solution à venir !

- Un dégradé passe de manière continue d'une couleur à une autre.
- Dans l'espace de couleur RVB, on peut utiliser une interpolation linéaire entre la couleur initiale et la couleur finale (en pratique, cette interpolation se fait sur chacune des composantes).
- Ici on part du blanc ($r = 255, v = 255, b = 255$) pour aller au vert ($r = 0, v = 255, b = 0$). Donc la composante verte ne change pas de valeur et seules les composantes rouge et bleue varient progressivement.

DÉGRADÉ DU BLANC AU VERT(*im*)

Remplit l'image *im* d'un dégradé du blanc (à gauche) au vert (à droite).

Paramètres : *im* (image) l'image à remplir.

Variables : *x*, *y* (entiers) pour parcourir les coordonnées des pixels.

rouge, *vert*, *bleu* (entiers) les 3 composantes de la couleur.

Début

vert \leftarrow 255

Pour *x* variant de 0 à *im.width* - 1 **Faire**

rouge \leftarrow arrondi($\frac{im.width-1-x}{im.width-1}$)

bleu \leftarrow *rouge*

Pour *y* variant de 0 à *im.height* - 1 **Faire**

im.set_color((*x*, *y*), (*rouge*, *vert*, *bleu*))

Fin Pour

Fin Pour

Fin

- ⚙ La composante *vert* ne varie jamais tout au long de l'image. Elle est donc définie une bonne fois pour toutes avant les boucles.
- ⚙ Le calcul de *rouge* réalise l'interpolation entre 255 (lorsque *x* vaut 0) et 0 (lorsque *x* vaut *im.width* - 1). Ce calcul n'a lieu que lorsque *x* varie (la boucle sur *y* parcourt une colonne; or dans une colonne la couleur est constante). La composante *bleu* est identique à la composante *rouge*.

```
#!/usr/bin/env python3
from simple_image import Image

def fill_with_gradient_from_wite_to_green(im):
    green = 255
    for x in range(im.width):
        red = round((im.width - 1 - x)/(im.width - 1)*255)
        blue = red
        for y in range(im.height):
            im.set_color((x, y), (red, green, blue))

if __name__ == "__main__":
    # Image.trace = False
    im = Image.new(256, 100)
    fill_with_gradient_from_wite_to_green(im)
    for x in [0, round(im.width/2), im.width - 1]:
        color = im.get_color((x, 0))
        print(x, color)
    im.save("images/degrade.png")
```

- ⚙ La fonction prédéfinie `round` arrondit une valeur flottante en une valeur entière.
- 👉 Quelle que soit la largeur choisie pour l'image, les 3 couleurs affichées sont (255, 255, 255), (≈ 127 , 255, ≈ 127) et (0, 255, 0) (il n'y a pas toujours de point exactement au milieu).

📎 le script `gradient.py` est une pièce-jointe de ce fichier PDF.



L'image `degrade.png`

💬 Écrire le script `negative.py` qui calcule le négatif d'une image.

👉 Le négatif d'une couleur est sa couleur complémentaire dans le modèle de couleur RVB (mathématiquement parlant, le négatif d'une couleur est son symétrique par rapport au centre du cube des couleurs RVB). Le jaune et le bleu sont donc le négatif l'un de l'autre.

🔧 Ce script recevra le nom du fichier image à lire et celui du fichier image à écrire via la ligne de commande.

Ainsi pour lire l'image `images/originale.png` et écrire le résultat dans l'image `images/negatif.png`, il faudra utiliser la commande suivante :

```
(venv-images) %% ./negative.py images/originale.png images/negatif.png
```

👉 En bonus : si l'utilisateur n'utilise pas le bon nombre d'arguments, le script devra afficher un message rappelant son usage correct.

⚙️ Le module `sys` définit la liste `sys.argv` qui contient la liste des mots composant la ligne de commande ayant servi à lancer le script.

Balise anti-divulgâchage



Solution à venir !

```
#!/usr/bin/env python3
from simple_image import Image
import sys

def usage(error):
    """
    Rappel à l'utilisateur l'usage du script.
    """
    print(f"Erreur: {error}\n"
          f"Usage: {sys.argv[0]} image_in image_out\n"
          "\n"
          "\tLit l'image 'image_in' et écrit son négatif dans\n"
          "\tle fichier 'image_out'.",
          file=sys.stderr)
    sys.exit(1)
```

- 👉 `sys.argv[0]` contient le nom du script.
- 👉 Le paramètre `file=sys.stderr` permet d'écrire sur la sortie d'erreur plutôt que sur la sortie standard (`sys.stdout`). C'est mieux pour les messages d'erreur !
- 👉 La fonction `sys.exit()` arrête brutalement le script en retournant l'entier fourni en paramètre (convention Unix : 0=tout s'est bien passé, autre valeur=c'est une erreur).


```
def negative(im1):  
    """  
    Retourne l'image négative de l'image en paramètre.  
    """  
    im2 = Image.new(width=im1.width, height=im1.height)  
    for x in range(im1.width):  
        for y in range(im1.height):  
            color = im1.get_color((x, y))  
            negative_color = tuple(255 - v for v in color)  
            im2.set_color((x, y), negative_color)  
    return im2  
  
if __name__ == "__main__":  
    # Image.trace = False  
    if len(sys.argv) != 3:  
        usage("nombre d'arguments incorrect !")  
    im_in = Image.read(sys.argv[1])  
    im_out = negative(im_in)  
    im_out.save(sys.argv[2])
```

👉 `sys.argv[1]` et `sys.argv[2]` sont respectivement le premier et le second mot qui suivent le nom du script sur la ligne de commande.

📎 le script `negative.py` est une pièce-jointe de ce fichier PDF.



original.png



negative.png

👉 La commande utilisée pour transformer l'image **original.png** en l'image **negative.png** :

```
(venv-images) %% ./degrade.py images/originale.png images/negative.png
```

```
(venv-images) C:> python degrade.py images/originale.png images/negative.png
```

- Concevoir le script `double.py` permettant de doubler la définition d'une image (l'image à lire et celle à produire seront fournies sur la ligne de commande).
- Question subsidiaire 1 : concevoir le script `demi.py` permettant d'appliquer un facteur de zoom $1/2$.
- Question subsidiaire 2 : généraliser l'algorithme précédent pour pouvoir utiliser un facteur de zoom quelconque (strictement supérieur à zéro).

Balise anti-divulgâchage



Solution à venir !

- 💬 L'image résultat est 2 fois plus large et 2 fois plus haute que l'image d'origine.
- 💬 Chaque pixel de l'image originale devient un pavé de 2×2 pixels dans l'image résultat.
- 👉 Pour un zoom simple, la couleur adoptée dans le pavé est la même couleur que celle du pixel original.

DOUBLE(*im*)

Retourne une image doublant la définition de l'image *im*.

Paramètre : *im* (image) image d'origine.

Résultat : *im2* (image) image résultat.

Variables : *x*, *y* (entiers) pour les coordonnées des pixels dans l'image originale.

dx, *dy* (entiers) pour décaler les coordonnées des pixels dans l'image résultat.

c (couleur) une couleur.

Début

im2 \leftarrow Image.new($2 \times im.width$, $2 \times im.height$)

Pour *x* **variant de** 0 **à** *im.width* - 1 **Faire**

Pour *y* **variant de** 0 **à** *im.height* - 1 **Faire**

c $\leftarrow im.get_color((x, y))$

Pour *dx* **variant de** 0 **à** 1 **Faire**

Pour *dy* **variant de** 0 **à** 1 **Faire**

im2.set_color(($2 \times x + dx$, $2 \times y + dy$), *c*)

Fin Pour

Fin Pour

Fin Pour

Fin Pour

Retour (*im2*)


Fin

```
#!/usr/bin/env python3
from simple_image import Image
import sys

def usage(error):
    """
    Rappelle à l'utilisateur l'usage du script.
    """
    print(f"Erreur: {error}\n"
          f"Usage: {sys.argv[0]} image_in image_out\n"
          "\n"
          "\tLit l'image 'image_in', double sa définition et\n"
          "\técrit le résultat dans le fichier 'image_out'.",
          file=sys.stderr)
    sys.exit(1)
```

```
def double(im1):
    """
    Double la définition de l'image en paramètre.
    """
    im2 = Image.new(width=im1.width*2, height=im1.height*2)
    for x in range(im1.width):
        for y in range(im1.height):
            color = im1.get_color((x, y))
            for dx in range(2):
                for dy in range(2):
                    im2.set_color((x*2+dx, y*2+dy), color)
    return im2

if __name__ == "__main__":
    # Image.trace = False
    if len(sys.argv) != 3:
        usage("nombre d'arguments incorrect !")
    im_in = Image.read(sys.argv[1])
    im_out = double(im_in)
    im_out.save(sys.argv[2])
```

 le script `double.py` est une pièce-jointe de ce fichier PDF.

- 💬 L'image résultat est 2 fois moins large et 2 fois moins haute que l'image d'origine.
- 💬 Chaque pavé de 2×2 pixels dans l'image originale devient un seul pixel dans l'image résultat.
- 👉 Pour un zoom simple, la couleur adoptée par le pixel résultat est la moyenne des 4 couleurs du pavé d'origine.

DOUBLE(*im*)

Retourne une image doublant la définition de l'image *im*.

Paramètre : *im* (image) image d'origine.

Résultat : *im2* (image) image résultat.

Variables : *x*, *y* (entiers) pour les coordonnées des pixels dans l'image originale.

c1, *c2*, *c3*, *c4* (couleurs) couleurs des 4 pixels d'un pavé. *r*, *v*, *b* (entiers) pour calculer la moeyenne de chaque composante.

Début

im2 \leftarrow Image.new(arrondi(*im.width*/2), arrondi(*im.height*/2))

Pour *x* variant de 0 à *im2.width* - 1 **Faire**

Pour *y* variant de 0 à *im2.height* - 1 **Faire**

c1 \leftarrow *im.get_color*((2 \times *x*, 2 \times *y*))

c2 \leftarrow *im.get_color*((2 \times *x* + 1, 2 \times *y*))

c3 \leftarrow *im.get_color*((2 \times *x*, 2 \times *y* + 1))

c4 \leftarrow *im.get_color*((2 \times *x* + 1, 2 \times *y* + 1))

r \leftarrow arrondi($\frac{c1[0]+c2[0]+c3[0]+c4[0]}{4}$)

v \leftarrow arrondi($\frac{c1[1]+c2[1]+c3[1]+c4[1]}{4}$)

b \leftarrow arrondi($\frac{c1[2]+c2[2]+c3[2]+c4[2]}{4}$)

im2.set_color((*x*, *y*), (*r*, *v*, *b*))

Fin Pour

Fin Pour

Retour (*im2*)

Fin

Exercice 4 : zoom $\times 2$, $\times 1/2$ et au-delà...

Le script `demi.py` : imports et fonction `usage()`

19i/19

```
#!/usr/bin/env python3
from simple_image import Image
import sys

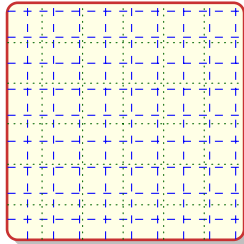
def usage(error):
    """
    Rappelle à l'utilisateur l'usage du script.
    """
    print(f"Erreur: {error}\n"
          f"Usage: {sys.argv[0]} image_in image_out\n"
          "\n"
          "\tLit l'image 'image_in', divise sa définition par 2\n"
          "\tet écrit le résultat dans le fichier 'image_out'.",
          file=sys.stderr)
    sys.exit(1)
```

```
def demi(im1):
    """
    Divise par 2 la définition de l'image en paramètre.
    """
    im2 = Image.new(width=int(im1.width/2), height=int(im1.height/2))
    for x in range(im2.width):
        for y in range(im2.height):
            c1 = im1.get_color((x*2, y*2))
            c2 = im1.get_color((x*2, y*2 + 1))
            c3 = im1.get_color((x*2 + 1, y*2))
            c4 = im1.get_color((x*2 + 1, y*2 + 1))
            c = (
                round((c1[0] + c2[0] + c3[0] + c4[0]) / 4), # rouge
                round((c1[1] + c2[1] + c3[1] + c4[1]) / 4), # vert
                round((c1[2] + c2[2] + c3[2] + c4[2]) / 4), # bleu
            )
            im2.set_color((x, y), c)
    return im2

if __name__ == "__main__":
    # Image.trace = False
    if len(sys.argv) != 3:
        usage("nombre d'arguments incorrect !")
    im_in = Image.read(sys.argv[1])
    im_out = demi(im_in)
    im_out.save(sys.argv[2])
```

 le script `demi.py` est une pièce-jointe de ce fichier PDF.

- 💬 Superposer la grille de pixels de l'image originale à celle de l'image résultat :



En déduire que chaque pixel de l'image résultat prend pour couleur la moyenne pondérée des couleurs de tous les pixels qu'il recouvre dans l'image originale.

- 😊 ... la rédaction du code correspondant est laissé comme exercice au lecteur !
- 👉 Il existe en fait de **nombreuses méthodes** [↗](#)... Les plus récentes utilisent maintenant des IA (par exemple [Upscayl](#) [↗](#)).