Informatique

**Matrices de pixels et images**

***Résumé***

|  |  |  |
| --- | --- | --- |
| Format sous python | Images au format BMP, gérées sous Python avec plt.imread sous forme d’array (numpy) de triplets d’entiers uint8 codés sur 8 bits (3 octets/pixel). Attention aux problèmes d’overflow dans les opérations +/-. Une image est donc un array ( pour l’exemple) de lignes et colonnes. Attention, est le pixel en haut à gauche de l’image… Ne pas utiliser , mais bien – Sinon erreurs avec les slices | |
| Pixel | Un pixel de l’image est un array contenant 3 entiers uint8 (éventuellement ). Les bits « de gauche » des nombres entiers R, G et B sont appelés bits de poids forts, et ceux « de droite » les bits de poids faible | |
| Exemple | Ci-dessous, une image de 18 pixels bleus, rouges et verts   |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | | 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 | | |
| Création d’une image vierge | Attention, ne créer qu’un array ne fonctionne pas, il faut spécifier le type int8 :   |  | | --- | | import numpy as np  Nl**,**Nc **=** 100**,**100  Image\_Noire **=** np**.**zeros**((**Nl**,**Nc**,**3**),**dtype**=**'uint8'**)**  Image\_Blanche **=** 255**\***np**.**ones**((**Nl**,**Nc**,**3**),**dtype**=**'uint8'**)** | | |
| Quelques outils | | |
| On oublie scipy pour matplotlib 😊 | | **import** matplotlib**.**pyplot **as** plt |
| Format .bmp RGB ou RGBA transformé en RGB  « read only » : copier pour modifier : image**.**copy**()** | | image **=** plt**.**imread**(**"Nom.bmp"**)**  image **=** image**[:,:,:**3**]** |
| Ouverture/Sauvegarde au format array | | import numpy as np  image = np.load("Nom.npy")  np.save('Nom',image) |
| Affichage  Penser à balader la souris sur les pixels pour avoir les infos ligne (y), colonne (x) et triplet [R,G,B] | | **import** matplotlib**.**pyplot **as** plt  def f\_affiche(image):  plt**.**figure**()**  plt**.**imshow**(**image**)**  plt**.**axis**(**'off'**)**  plt**.**show**()**  plt**.**pause**(**0.00001**)**  f\_affiche(image) |
| Dimensions : lignes et colonnes | | Nl**,**Nc **=** image**.**shape**[**0**:**2**]** |
| Accès à un pixel à la ligne l (int) et colonne c (int) | | Pixel = image[l,c] |
| Parcours des pixels  Utiliser (l,c) plutôt que (i,j) ou (y,x) | | **for** c **in** range**(**Nc**):**  **for** l **in** range**(**Nl**):** |
| Utilisation des slices (gain de temps vis-à-vis d’un for² | | image**[:,:]** **=** **[**127**,**127**,**127**]** |
| Accès aux entiers R, G et B | | R**,**G**,**B **=** Pixel |
| Attention : manipulant des array, modifier des pixels modifie l’image en mémoire. Si on veut créer une copie pour garder l’image d’origine, écrire : image\_2 = np.copy(image\_1) | | |
| Enregistrement plt.imsave("Image.bmp",image) sans afficher (dimensions de l’image = lignes colonnes) ou plt.savefig("Image.png") en 640x480 après l’avoir affichée | | |

|  |  |  |
| --- | --- | --- |
| Travail sur les images | | |
| Bits de poids forts et faibles | Image complète  C:\Users\Denis DEFAUCHY\AppData\Local\Microsoft\Windows\INetCache\Content.Word\Image_Test.bmp | |
| 4 Bits de poids forts, les 4 faibles=0  C:\Users\Denis DEFAUCHY\AppData\Local\Microsoft\Windows\INetCache\Content.Word\Prog_Que_Forts_4_Bits.bmp | 4 Bits de poids faibles, les 4 forts=0  C:\Users\Denis DEFAUCHY\AppData\Local\Microsoft\Windows\INetCache\Content.Word\Prog_Que_Faibles_4_Bits.bmp |
| Nuances de gris | On obtient une nuance de gris en égalisant pour chaque pixel à une valeur entre 0 et 255 (ou n’imposer que la valeur sans utiliser de triplets)   |  |  | | --- | --- | | R=0  G=0  B=0 | R=200  G=200  B=200 | | R=100  G=100  B=100 | R=0  G=0  B=0 |   On pourra privilégier une couleur plus qu’une autre en calculant en jouant sur :  Attention à l’overflow possible si , bien passer par la formule de   |  |  |  | | --- | --- | --- | | Pas de couleur privilégiée | Accent sur le bleu | Accent sur le rouge | |  |  |  | | |
| Noir et blanc | Pixel noir : - Pixel blanc : (ou une seule valeur)   |  |  | | --- | --- | | R=0  G=0  B=0 | R=255  G=255  B=255 | | R=0  G=0  B=0 | R=0  G=0  B=0 |   Il suffira de définir un seuil sur après avoir transformé l’image en nuances de gris :   |  |  |  | | --- | --- | --- | |  |  |  | |  |  |  | | |

|  |  |  |  |
| --- | --- | --- | --- |
| Convolution | | | |
| Principe  avec un noyau de dimensions 3x3 | Noyau à lignes et colonnes  **Intérieurs** | | |
| Filtres | Moyenneur  *Floutage* |  |  |
| Repoussage |  |  |
| Laplacien  *Contours* |  |  |
| Réhausseur |  |  |
| Gaussien  3x3 |  |  |
| Gaussien  5x5 |  |  |
| Gestion des bords | On peut les laisser à leur valeur d’origine, redimensionner l’image de sortie en les enlevant, ou traiter les formules au cas par cas | | |
| Normalisation | Afin de maintenir une moyenne des RGB par convolution, on peut normaliser la matrice en divisant chacun de ses termes par la somme de tous ses termes | | |
| Re limitation des résultats | Dans le cadre des BMP par exemple et pour éviter l’overflow, on relimite les valeurs de R, G et B afin de les maintenir dans l’intervalle [0,255] avec une formule comme : | | |

|  |  |  |
| --- | --- | --- |
| Transformations | | |
| Rotation |  |  |
| Direct | Pour chaque pixel de l’image source, on trouve l’image par rotation d’angle et de centre dans l’image cible, à laquelle on applique le triplet RGB de  :  Cette méthode crée des trous, les coordonnées ne tombant pas sur 100% des pixels de l’image cible |
| Inverse | Pour chaque pixel de l’image cible, on trouve l’antécédent par rotation d’angle et de centre dans l’image source, et on applique à le triplet RGB de : .  Il est aussi possible de déterminer la valeur de par interpollation bilinéaire sur les 4 pixels entourant les coordonnées flottantes avec |
| Réduction | Prendre 1 pixels tous les n pixels. Risque d’effet d’aliasing (lignes visibles)  Pour limiter cet effet, appliquer un filtre moyenneur avant suppression par exemple | |
| Prendre la moyenne par paquets de nxn pixels, et ce tous les n pixels tous les n pixels en ligne et en colonne    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 | |

|  |  |  |
| --- | --- | --- |
| Agrandissement | Principe | Ajouter des lignes et/ou colonnes à l’image et déterminer la couleur à imposer aux nouveaux pixels par interpolation |
| Une image contenant mammifère, regardant, fixant  Description générée automatiquementPlus proches voisins | prend la valeur du triplet du pixel le plus proche avec un choix de priorité (droite, gauche, haut, bas)      Exemple avec priorité à gauche ou au-dessus |
| Interpolation bilinéaire | On connaît les 4 triplets RGB autour d’une zone dans laquelle on ajoute des pixels aux coordonnées  Une image contenant texte, horloge  Description générée automatiquement  On détermine le triplet RBG en fonction des 4 pixels avoisinants à l’aide d’une fonction telle que :  Si est à 1/3 du bord gauche et 1/3 du bord inférieur (comme sur l’image)   |  |  | | --- | --- | |  |  | |
| Interpolation bicubique | Sans rentrer dans les détails, on propose une interpolation qui 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. |