Tableaux : application au traitement d'image

Cours 18

Structure de données - v1.0

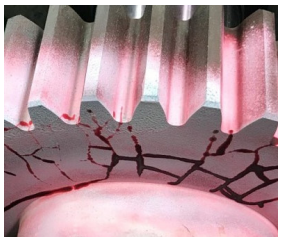
Lycée polyvalent Franklin Roosevelt, 10 Rue du Président Franklin Roosevelt, 51100 Reims

Table des matières

1 Introduction

1.1 Contexte

Depuis l'avènement de la technologie de l'imagerie numérique, et les progrès effectués sur les systèmes de traitement de l'information avec l'apprentissage automatique (*machine learning*), le traitement d'images est en plein essor et est de plus en plus développé dans des contextes très variés : contrôle non destructif, robots et véhicules autonomes, imagerie médicale, astronomie ou encore la déformation des matériaux.



(a) Détection de fissure par ressuage

(b) Reconstruction 3D d'un crâne

(c) Voiture autonome

FIGURE 1 – Exemple d'utilisation de traitement d'image

1.2 Propriétés des formats d'images

Les formats d'images sont nombreux. Le choix d'utiliser l'un ou l'autre dépend des propriétés souhaitées. Les principales limitations sont sur le nombre de couleurs, la gestion de la transparence et la compression.

Gestion de la transparence Le format `JPG` ne gère pas du tout la transparence. Les formats `PNG24` et `GIF` ont la possibilité d'avoir une seule couleur totalement transparente. Les formats `PNG32` et `JPEG 2000` ajoutent un canal, appelé canal alpha, permettant la gestion de différents niveaux de transparence.

Nombre de couleurs Le nombre de couleurs différentes (appelé profondeur) pouvant être obtenues dépend du codage de chaque pixel :

- 8 bits permettent d'avoir 256 couleurs (format souvent utilisé pour les images en nuance de gris) ;
- 16 bits permettent d'avoir 65 536 couleurs (appelé « haute couleur ») ;
- 24 bits permettent d'avoir 16 777 216 couleurs (appelé abusivement « vraies couleurs ») ;
- 32 bits permettent d'ajouter un codage sur 8 bits du canal de transparence alpha.

Les formats `GIF`, `PNG8` et `BMP8` se limitent au codage sur 8 bits : une table de correspondance permet alors d'assigner à chaque valeur une couleur différente.

Compression On distingue 3 grands types de compression des images :

- pas de compression : très souvent le cas du format `BMP` ;
- compression sans perte (ou non destructive) : il utilise un algorithme de compression complètement réversible : c'est le cas des format `PNG` et `GIF` ;

- compression avec perte : la compression n'est pas totalement réversible (perte de détails et apparition d'artefacts) : c'est le cas du format `JPEG`.

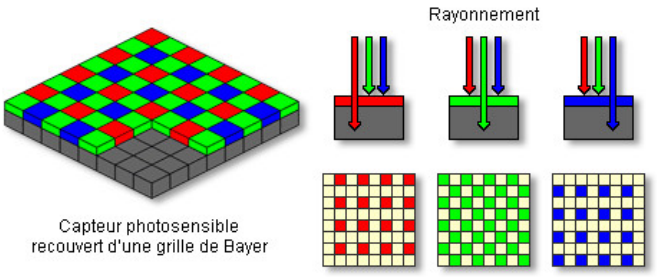
Pour l'échange de fichier sur internet, on préférera une compression importante, mais pour le traitement de données, on préférera un format sans perte.

2 Représentation des images

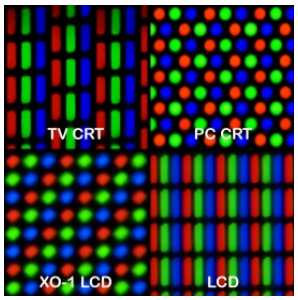
Les images numériques sont représentées matriciellement sous la forme d'un grand tableau comportant les informations de niveaux de gris ou de niveaux de couleurs élémentaires : typiquement, on a :

- une seule valeur codée sur 8 bits pour une image en niveau de gris ;
- trois valeurs codées chacune sur 8 bits pour une image en couleur : on parle de représentation `RGB` (pour `Red`, `Green` et `Blue`) ;
- quatre valeurs codées chacune sur 8 bits avec l'ajout du canal alpha : on parle alors de représentation `RGBA`.

D'un point de vue pratique, un capteur photosensible CCD ou CMOS est constitué d'une seule matrice photosensible qui est recouverte d'un filtre coloré appelé une grille de Bayer. Contenant des éléments de différentes couleurs, elle permet de sensibiliser sélectivement les pixels à une seule des 3 couleurs primaires : le rouge, le vert ou le bleu. De la même façon, pour la reconstruction des pixels sur un écran, trois sources différentes sont placées les unes à côté des autres pour reconstruire le pixel.



(a) Capteur photosensible recouvert d'une grille de Bayer



(b) Zoom sur plusieurs types d'écran

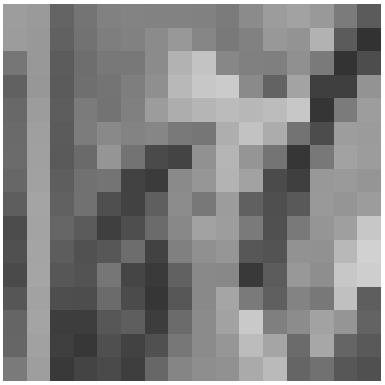
3 Tableaux et listes imbriquées : initialisation et copie

3.1 Ouverture fichier avec `imageio`

Pour convertir un fichier image en tableau de valeur et inversement, il existe de nombreux modules Python : nous allons introduire ici l'un des plus simples d'utilisation `imageio` :

```
import imageio
image = imageio.imread("Lenna.png")
array = np.array(image)
listOfList = image.tolist() # ou array.tolist()
```

Cela permet de créer un tableau `numpy` appelé `array` à partir de l'image `Lenna8.png`. On crée à partir du tableau une liste de listes (de listes) appelé `listOfList` à partir de ce tableau.



(a) Image 16x16 en niveau de gris

```
[[157 154 100 116 129 131 130 130 129 123 139 156 162 153 123  91]
 [156 152  97 114 126 129 139 152 134 121 129 152 147 178  91  50]
 [118 152  92 110 120 120 139 178 195 135 128 127 144 122  51  77]
 [ 97 151  91 111 116 126 143 184 199 203 138  99 163  63  63 147]
 [104 156  93 123 115 128 156 173 181 187 183 188 199  53 123 157]
 [106 159  91 124 137 131 134 121 118 174 194 172 115  76 157 155]
 [107 160  90 106 149 115  74  68 145 181 149 116  54 120 162 156]
 [103 161  94 113 116  66  59 138 150 179 162  75  63 152 154 149]
 [ 94 164  97 113  79  66  95 140 120 156  96  79  89 153 149 156]
 [ 76 163 102  91  63  78 107 136 162 157 122  80 121 149 159 199]
 [ 80 164  93  81  87 109  66 128 144 150  87  83 146 145 184 209]
 [ 74 165  87  83 118  69  58  97 137 136  57  92 152 142 199 206]
 [ 86 162  78  77 107  75  54  87 137 165 122  95 131 120 192  94]
 [101 164  61  61  87  94  61 107 139 162 201 127 140 162 149  99]
 [102 161  63  55  79  66  80 119 139 150 188 165 107 167 101  89]
 [122 158  57  68  74  62 103 133 142 145 170 186 101 113  87  80]]
```

(b) Tableau correspondant

3.2 Placement spatial

On peut accéder au pixel de coordonnées `(j, i)` de l'image en lecture et en écriture par `array[i][j]` ou `listOfList[i][j]`. **Pour les tableaux numpy uniquement**, on peut utiliser la notation simplifiée : `array[i,j]`. Le position du pixel par rapport à ses indices est expliquée par la FIGURE 4 : l'indice de la ligne est toujours le premier, après l'indice de la colonne.

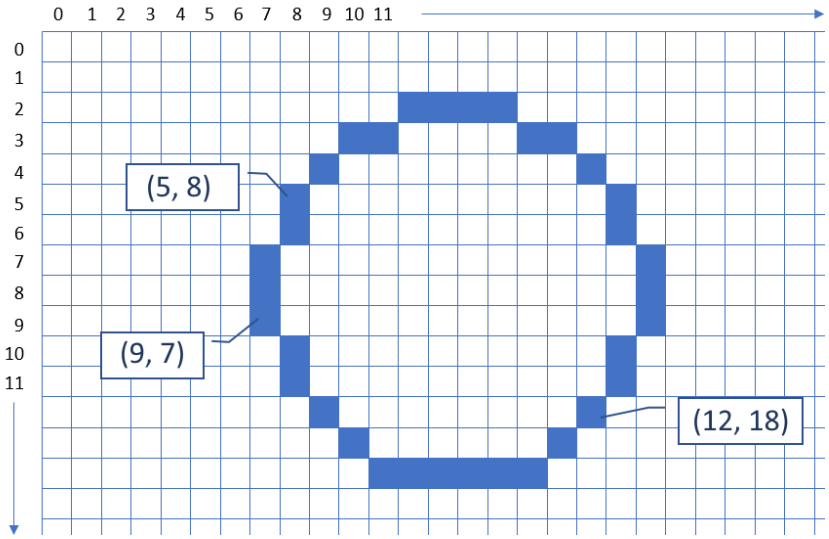


FIGURE 4 – Illustration de la numérotation des lignes (de haut en bas) et des colonnes (de gauche à droite)

3.3 Taille

Pour connaitre la taille d'une image : on peut utiliser les commandes suivantes :

- `m = len(listOfList)` pour connaitre le nombre de lignes ;
- `p = len(listOfList[0])` pour connaitre le nombre de colonnes ;
- `q = len(listOfList[0][0])` si l'image est en couleur : retourne 3 pour une représentation `RGB` et 4 pour une représentation `RGBA`.

Pour les tableaux numpy uniquement, il existe une méthode qui retourne sous forme de `tuple` la taille du tableau : `m, p = array.shape()` pour une image en niveau de gris ou `m, p, q = array.shape()` pour une image couleur.

Pour que `(j, i)` corresponde aux coordonnées d'un pixel valide, il faut donc que : `0 <= i < m` et `0 <= j < p`.

3.4 Initialisation

On va ici chercher à créer une image d'une taille donnée à partir d'un `tuple` correspondant à la taille souhaitée (`tlen = (600, 800, 3)` par exemple).

Pour créer une image remplie de zéros, on peut utiliser par exemple des codes suivants :

```
def imageNoire(tlen):
    m, p, q = tlen
    image = []
    for i in range(m):
        ligne = []
        for j in range(p):
            pixel = []
            for k in range(q):
                pixel.append(0)
            ligne.append(pixel)
        image.append(ligne)
    return image
```

Ou alors, en plus court, mais moins explicite :

```
def imageNoire(tlen):
    m, p, q = tlen
    return [[[0 for k in range(q)] for j in range(p)] for i in range(m)]
```

Mais d'autres versions doivent être utilisées pour une image en nuance de gris (cas où `tlen` ne contient que 2 éléments). Il existe alors d'autres solutions, plus généralistes, utilisant par exemple la récursivité :

```
def imageNoire(tlen):
    if len(tlen) == 1:
        return [0 for i in range(tlen[0])]
    else:
        return [ imageNoire(tlen[1:]) for i in range(tlen[0]) ]
```

Attention : la concaténation répétée (avec l'opérateur `*`) ne crée pas plusieurs listes, mais copie une référence à une même liste. Cette opération est très dangereuse dans son utilisation avec des listes imbriquées.

Pour les tableaux numpy uniquement, il existe une fonction `zeros` permettant d'initialiser un tableau à partir de sa taille sous forme de tuple :
`imageNoire = np.zeros((m, p, q), dtype=np.uint8)`.

3.5 Copie : réalisation d'une copie profonde

Les listes étant des objets mutables, la copie d'une image `im1` ne peut pas se réaliser simplement par l'opération `im2 = im1`. Parce que la structure est imbriquée (liste de listes ou même liste de listes de listes), une copie classique (`im2 = im1[:]` ou `im2 = im1.copy()`) n'est pas suffisante, car les sous-listes réfèrent encore aux mêmes objets.

Une façon de faire une copie est d'initialiser une nouvelle image avec la même taille et de la remplir avec les mêmes valeurs en utilisant une double ou une triple boucle imbriquée :

```
def copie(im0):
    m, p, q = len(im0), len(im0[0]), len(im0[0][0])
    im1 = imageNoire(m, p, q)
    for i in range(m):
        for j in range(p):
            for k in range(q):
                im1[i][j][k] = im0[i][j][k]
    return im1
```

En plus d'être un peu long, le programme de copie n'est pas le même en nuance de gris et en couleur. On préférera une méthode plus généraliste, utilisant la récursivité :

```
def copie(L):
    if type(L[0]) != list: # pas de sous-liste
        return L.copy()
    else:
        Lcopie = []
        for elt in L:
            Lcopie.append(copie(elt))
        return Lcopie
```

Une dernière façon est d'utiliser directement la fonction `deepcopy` du module `copy`. :

```
from copy import deepcopy
def copie(L):
    return deepcopy(L)
```

Pour les tableaux numpy uniquement, il existe une fonction `np.copy` ou la méthode `.copy()` permettant de copier un tableau :
`im2 = np.copy(im1)` ou
`im2 = im1.copy()`.

4 Programmation matricielle et bibliothèque Numpy pour les aspects numériques (physique, chimie, S2I)

La programmation matricielle (ou *array programming*) est un paradigme de programmation tourné vers le traitement des tableaux et des matrices. Ces types de données sont fondamentales dans le domaine des simulations (par exemple pour la *méthode des éléments finis*) et du traitement des grands volumes d'informations (ce qu'on appelle le *big data*).

Matlab, qui est très utilisé dans le monde scientifique, est un exemple de langage de programmation tourné vers ce paradigme, mais de nombreuses bibliothèques permettent d'ajouter de la programmation matricielle à un langage. C'est typiquement le cas de la bibliothèque *numpy* dans Python avec son type *np.array*.

4.1 Création d'un tableau np.array

a) Création de vecteurs

Les vecteurs sont des tableaux de dimension 1 (un seul indice). Trois générateurs de vecteurs sont particulièrement utiles :

- `np.linspace(a, b, n)` qui crée un vecteur de `n` valeurs régulièrement espacées de `a` à `b` (bornes incluses).
- `np.logspace(a, b, n)` qui crée un vecteur de `n` valeurs logarithmiquement espacées de 10^a à 10^b (bornes incluses).
- `np.arange(a, b, dx)` qui crée un vecteur de valeurs de `a` à `b` (`b` exclus) par pas de `dx`.

b) À partir d'une liste de listes

Pour passer d'une liste de listes à un tableau, on utilise la fonction `np.array(A)` ou `np.asarray(A)`. Pour passer d'un tableau à une liste de listes, on utilise la méthode `A.tolist()`.

4.2 Indexation logique

On peut faire une modification sélective par indexation logique : si l'on souhaite que tous les termes dont la valeur est inférieure à 5 soient maintenant égaux à 0, on peut par exemple écrire :

```
A[A<5] = 0
```

4.3 Opérations et fonctions

Les opérations numériques et l'application de fonction se fait terme par terme lorsque les deux tableaux ont la même taille. Si on fait une opération avec un nombre seul, l'opération est effectuée sur chacun des termes du tableau.

```
x = np.linspace(0,2*np.pi,1000)
y = np.sin(x-np.pi/4)
```

Seules les fonctions `numpy` permettent de prendre en entrée des tableaux. On parle alors de vectorisation du calcul, qui a l'avantage d'être beaucoup plus rapide que l'utilisation classique de boucles (la complexité est la même, mais l'implémentation matérielle est optimisée).

4.4 Connaître les caractéristiques d'un tableau

Pour connaître les dimensions d'un tableau, on peut utiliser`T.ndim` pour connaître le nombre de dimensions (nombre d'indices), mais surtout `T.shape` qui renvoie un n-uplet qui correspond à la taille de chacune des dimensions. `T.size` correspond au nombre total d'éléments.

Un tableau est toujours d'un type unique. On peut le connaître en utilisant `T.dtype`. Les types de données classiques sont :

- `np.bool` pour les booléens ;
- `np.int32` pour les entiers signés sur 32 bits (par défaut pour les entiers) ;
- `np.uint32` pour les entiers non signés sur 32 bits ;
- `np.int16` pour les entiers signés sur 16 bits ;

- `np.uint16` pour les entiers non signés sur 16 bits ;
- `np.int8` pour les entiers signés sur 8 bits ;
- `np.uint8` pour les entiers non signés sur 8 bits ;
- `np.float32` pour les flottants sur 32 bits ;
- `np.float64` pour les flottants sur 64 bits (par défaut pour les flottants) ;
- `np.float64` pour les complexes sur 64 bits (2 fois 32 bits) ;
- `np.float128` pour les complexes sur 128 bits (2 fois 64 bits, par défaut pour les complexes).

Pour faire une conversion d'un type à un autre, on peut utiliser l'instruction suivante : `newT = T.astype(dataType)`.

4.5 Création d'un tableau initial

La création d'un tableau peut se faire par 3 grands générateurs :

- `np.empty(shp, dtype = dataType)` pour un tableau qui contient a priori n'importe quoi ;
- `np.zeros(shp, dtype = dataType)` pour un tableau rempli de 0 (ou `False` pour les booléens) ;
- `np.ones(shp, dtype = dataType)` pour un tableau qui rempli de 1 (ou `True` pour les booléens).

Dans les 3 cas, `shp` représente la forme du tableau (taille de chacune des dimensions) sous la forme d'un n-uplet (*tuple*). La précision du type de données est optionnelle.

On peut également créer des tableaux particuliers : des matrices diagonales à partir d'un vecteur `V` avec `np.diag(V)` ou une matrice identité de taille $n \times n$ avec `np.eye(n)`.

4.6 Opérations matricielles

Il est bien sûr important dans un langage matricielle de pouvoir réaliser assez directement des opérations matricielles classiques : multiplication matricielle (fonction `np.dot(A,B)`, méthode `A.dot(B)` ou opérateur `A@B`), transposée (fonction `np.transpose(A)`, méthode `A.transpose()` ou `A.T`), calcul de trace (fonction `np.trace(A)` ou méthode `A.trace()`).

4.7 Algèbre linéaire

Une sous-bibliothèque est spécialisée en algèbre linéaire. On peut par exemple l'importer ainsi : `import numpy.linalg as la` (*linear algebra*).

Elle permet de calculer des inversions matricielles (fonction `la.inv(A)`), des déterminants (fonction `la.det(A)`), des rangs de matrices (fonction `la.matrix_rank(A)`), des valeurs propres et des vecteurs propres (fonction `la.eig(A)` pour la matrice de passage et `la.eigvals(A)` pour le vecteur des valeurs propres) et permet de résoudre des équations du type $AX = B$ (fonction `la.solve(A, B)`).