M1 Physique Appliquée — Université Paris-Sud XI
Année 2015-2016

Programmation et données numériques

TD5 : représentation des données

Vous pouvez télécharger le sujet au format pdf sur le site du cours : `http://lptms.u-psud.fr/wiki-cours/index.php/Programmation_et_données_numériques_M1_Physique_Appliquée` si vous voulez faire des copier-coller.

Les entiers et les réels

Le site `http://baseconvert.com` pour vous aider à passer d'une écriture en décimal aux écritures en binaire.

☞ Étudier et exécuter les instructions suivantes :

```
eps=1.0
while not (1.0+0.5*eps) == 1.0:
    eps *= 0.5
print eps
```

Dans quelle base sont codés les parties fractionnaires des réels ? Interpréter le résultat. À quoi correspond la valeur affichée ?
Remplacer les deux 1.0 dans le test par 0.0. À quoi correspond la valeur affichée ?

☞ Dans une cellule, faite afficher successivement les résultats des opérations suivantes sans utiliser `print` qui met en forme le résultat :

```
0.125+0.5
0.1+0.2
```

Qu'est-ce que le résultat de la seconde opération a de surprenant ? Pour mieux comprendre, nous utilisons la bibliothèque `decimal` de Python qui permet d'afficher la représentation décimale exacte d'une variable avec un grand nombre de chiffres significatifs. Taper et interpréter les résultats suivants (*on pourra penser au problème qui apparaît lorsqu'on écrit* 1/3 *en base 10*) :

```
from decimal import Decimal

print Decimal("0.1") # affiche la représentation décimale à partir d'une string
print Decimal(0.1)   # affiche la représentation décimale du réel 0.1 codé sur un mot mémoire
print Decimal(0.1+0.2), Decimal(0.3)
print Decimal("0.1")+Decimal("0.2")

print Decimal(0.25), Decimal(0.125)
print Decimal(0.125+0.25)
print Decimal(0.35-0.1)
```

☞ Les entiers non-signés et codés sur 8 bits ou 1 octet sont utiles pour coder les lettres et les couleurs. Le type correspond est disponible dans `numpy` sur le nom `uint8`. Taper et interpréter

```
import numpy as np

print np.uint8(1000), 1000%256
print np.uint8(230)+np.uint8(120)
print np.uint8(-20), np.uint8(3.4*41)
```

Manipuler des images

On pourra débuter cette partie du TD par les lignes suivantes qui chargent les bibliothèques utiles (`numpy` a déjà été importée dans la partie précédente) :

```
import matplotlib.pyplot as pl
import matplotlib.image as mpim
%matplotlib inline
```

N'oubliez pas que la plupart des fonctions mathématiques sont disponibles dans `numpy`.
On pourra télécharger dans le dossier courant l'image `Grenouille.jpg` à partir du site du cours.

Chargement et sauvegarde

☞ On peut importer l'image et la faire afficher par les commandes

```
img = mpim.imread("Grenouille.jpg")
type(img)
```

Sous quelle forme (type d'objet) l'image est-elle importée ? Combien y a-t-il de dimensions ?

☞ Pour faire afficher l'image, vous pouvez utiliser `pyplot` via la commande

```
pl.imshow(img)
```

`pyplot` rajouter automatiquement des axes. En déduire la signification des deux premières dimensions. Dans quel ordre sont numérotés les pixels ?
Pour ne pas faire afficher les axes mais uniquement l'image, vous pouvez faire la commande suivante :

```
pl.axis('off')
pl.imshow(img)
```

☞ Faite afficher un élément (ou pixel) de l'image en choisissant des valeurs pour les deux premiers indices

```
img[100,100,:]
```

À quoi correspondent finalement les deux premières dimensions et la troisième ?
A quelle taille maximale en Ko s'attend-on pour stocker le contenu en pixels de l'image ?

☞ On peut copier et/ou sauvegarder une image par les commandes suivantes :

```
imgcopy = np.copy(img)
mpim.imsave("Grenouille_copy.jpg",imgcopy)
```

Observer les propriétés de l'image sauvegardée avec Windows.

Agir sur la structure spatiale de l'image

☞ Nous rappelons que la méthode de slicing des indices sur une liste est applicable à tous les indices d'un `array` de numpy et qu'elle prend la forme [start :stop :step] avec la possibilité de mettre des indices négatifs. En l'absence d'indication, tous les indices sont parcourus, par exemple img[:, :] renvoie l'ensemble de la matrice. À l'aide de cette méthode, proposer en une ligne de la forme

```
pl.imshow(img[?:?:?,?:?:?])
```

dans laquelle vous remplacerez les ? par un nombre bien choisi, une manière de :
— prendre la réflexion par rapport à la verticale plus par rapport à l'horizontale.
— retourner de 180° la photo.
— réduire sa taille par deux (sans se soucier d'une éventuelle interpolation).
— extraire le quadrant (quart de la photo) supérieur gauche.

☞ On étudie ici une méthode pour découper une forme quelconque de l'image. Pour cela, un masque est créé qui va sélectionner les pixels à mettre à 0 (en noir). Voici comment générer un masque circulaire :

```
nl, nc, s = img.shape # on récupère les dimensions dans nl et nc, s=3
rayon = 120           # rayon voulu en pixel
Y, X = np.ogrid[:nl,:nc] # crée le tableau 2D des indices de la photo
masque = (X-nc/2)**2 + (Y-nl/2)**2 > rayon**2 # définition de l'extérieur du cercle
                                              # centré sur le milieu de l'image
print masque[0,0] # le masque est une matrice de booléen
pl.imshow(masque, cmap=pl.cm.gray) # on affiche le masque, cmap est la palette de couleur
```

puis

```
img_coupee = np.copy(img)
pixel_noir = np.array ([0,0,0], dtype=np.uint8) # définition du pixel noir
img_coupee[masque] = pixel_noir # les indices du masque renvoyant True sont mis en noir
pl.imshow(img_coupee)
```

Reprendre cette technique pour créer un masque ellipsoïdal qui met en blanc les pixels correspondant au masque.

Agir sur les couleurs de l'image

☞ On souhaite prendre le négatif couleur de l'image, c'est-à-dire l'image pour laquelle chacun des pixels est la couleur complémentaire de celle de l'image originale. Pour cela, nous écrivons

```
neg = 255*np.ones(img.shape,dtype=np.uint8)
neg ????
pl.imshow(neg)
```

Que fait la première la ligne ? Enlever `dtype=np.uint8` pour comprendre son rôle, en vous référant également à la première partie. Dans la seconde ligne, remplacer les ? ? ? ? par une instruction qui va créer le négatif.

☞ On veut pouvoir changer le contraste de l'image. Cela consiste à augmenter l'intensité des couleurs au-dessus de 127.5 et diminuer l'intensité de celles en-dessous de 127.5 et ce, de façon progressive, à l'aide d'une fonction f qui envoie l'intervalle [0,255] sur lui-même. On choisit une fonction f paramétrée par s de la forme

$$f(x) = 255\frac{\tanh(s(x-x_0)/255.) - a}{b-a} .$$

Quelles valeurs prendre pour x_0, a et b ? Tracer la fonction obtenue (on utilisera `np.tanh`). Que fait la fonction `contraste` ci-dessous ?

```
def contraste(source, f):
    return np.array(f(source [:,:]), dtype=np.uint8)
```

L'utiliser pour modifier le contraste de l'image avec $s = 5$.

Compression par décomposition en valeurs singulières

Le principe est le suivant : toute matrice A de dimensions $N \times M$ est décomposable sous la forme

$$A = USV$$

où U est une matrice $N \times N$, S une matrice $N \times M$ diagonale dont on note $\{s_i\}_{i=1,\min(N,M)}$ les éléments diagonaux, appelés valeurs singulières, et enfin V une matrice $M \times M$. Cette décomposition est exacte. Si l'on ne conserve pas toutes les valeurs singulières mais un nombre $k < \min(N, M)$, alors on obtient une bonne approximation de A que l'on notera R. Du coup, on ne peut conserver que les k premières colonnes de U et les k premières lignes de V pour former des matrices tronquées $\tilde{U}$, $\tilde{S}$ et $\tilde{V}$. On peut donc écrire

$$R = \tilde{U}\tilde{S}\tilde{V}$$

qui a besoin de moins d'information que dans la formule précédente du fait de la troncation.

☞ Pour cela, commençons par simplifier l'information contenu dans la photo couleur en la transformant en photo niveau de gris avec un seul nombre par pixel de sorte que cela corresponde exactement à une matrice que nous appellerons A.

```
def rgb2gray(rgb):
    return np.dot(rgb, [0.299, 0.587, 0.144]) # convention de poids pour les niveaux de gris
A = rgb2gray(img) # matrice en niveau de gris
pl.imshow(A, cmap=pl.cm.gray) # on choisit une colormap en niveau de gris
pl.clim(0,255)              # l'échelle de la colormap est de 0 à 255
```

☞ La décomposition en valeurs singulières est calculée numériquement par la commande

```
U, s, V = np.linalg.svd(A) # U et V sont des matrices, s est un vecteur
```

Tracer le spectre des valeurs singulières en fonction de leur indice. On pourra utiliser `pl.semilogy()` pour mettre l'axe y en échelle logarithmique.

☞ Il faut enfin procéder à la construction de la matrice compressée, notée `R`, en n'utilisant qu'un nombre partiel de valeurs singulières que l'on note `kept`. Ainsi, on tronque la dimension de S et donc le nombre de colonnes dans U et le nombre de lignes dans V aux `kept` premières. Si `kept` reste petit, il y a finalement moins d'information à stocker que dans la matrice initiale. Les lignes suivantes

```
kept = 25
S = np.diag(s[?:?])
R = np.dot(U[?:?,?:?],np.dot(S,V[?:?,?:?]))
pl.subplot(1, 2, 1)
pl.imshow(R,cmap=pl.cm.gray)
pl.clim(0,255); pl.axis('off')
pl.subplot(1, 2, 2)
pl.imshow(A,cmap=pl.cm.gray);
pl.clim(0,255); pl.axis('off')
```

Remplacer les ? par un blanc ou le nom d'une variable pour que la compression soit effective. Les dernières lignes permettent de faire afficher le résultat à côté de la matrice originale. Jouer avec la valeur de `kept` pour voir l'effet visuel de la compression.

☞ On rappelle que pour un objet `array` de numpy, `size` permet d'accéder au nombre total d'éléments, par exemple `A.size` correspond au nombre d'éléments dans A. Estimer le taux de compression théorique pour `kept=25`.