
MGM652 Cours

Version 1.0.0

Ludovic Charleux

17 February 2013

Table des matières

| | | |
|----------|--------------------------|-----------|
| 1 | Installation | 3 |
| 2 | Introduction | 5 |
| 2.1 | Execution | 5 |
| 2.2 | Bases | 5 |
| 2.3 | Modules | 9 |
| 3 | Images Numériques | 15 |
| 3.1 | Formation | 15 |
| 3.2 | Structure | 15 |
| 3.3 | Operations | 18 |
| 4 | Optimisation | 31 |

Cette partie du cours propose une introduction à Python et

Contenu :

Python présente plusieurs avantages à l'origine de son choix pour ce cours :

- C'est un langage généraliste présent dans de nombreuses domaines : calcul scientifique, web, bases de données, jeu vidéo, graphisme, etc. C'est un outil polyvalent qu'un ingénieur peut utiliser pour toutes les tâches numériques dont il a besoin.
- Il est présent sur la majorité des plateformes : Windows, Mac OS, Linux, Unix, Android
- C'est un langage libre et gratuit avec une grande communauté d'utilisateurs. Vos programmes sont donc échangeables sans contraintes. Vos questions trouveront toujours des réponses sur internet. Enfin, très peu de bugs sont présents dans Python.

Installation

A Polytech' Annecy-Chambéry, Python est présent sur toutes les machines et toutes les plateformes :

- Sous Windows (XP/7) : vous le trouverez dans le dossier **R :\Python26** dans sa version 2.6.
- Sous Fédora/Linux : il est installé de base et est accessible dans un terminal (Konsole par exemple) *via* la commande **python**.

Sur les autres machines, voir la section dédiée sur le site python.org/download.

Introduction

2.1 Execution

Python s'utilise comme une “grosse” calculatrice, les commandes peuvent y être tapées une par une dans un terminal :

```
>>> print 'Hello World !'  
Hello World !  
>>> a = 5.  
>>> b = 7.  
>>> a + b  
12.0
```

Ou grâce à un fichier texte, de préférence terminé par .py, comme le fichier `script.py` contenant :

```
# Ceci est un programme Python tres basique  
a = 5 # On definit une variable "a"  
b = 7  
print a + b
```

Pour exécuter un script Python, deux options sont possibles :

- L'exécuter directement dans un terminal avec la commande `python script.py`.
- Lancer Python puis les utiliser la commande Python :

```
>>> execfile('script.py')
```

La première méthode est adaptée à un programme fonctionnant déjà correctement alors que la seconde est plus adaptée à la période d'écriture du programme car elle permet d'accéder après coup aux variables générées par le programme.

Note : Ces considérations sur le mode d'exécution des programme est essentiellement importante sous Linux.

2.2 Bases

Cette partie fournit quelques informations basiques sur Python, pour aller plus loin, se référer à la très complète documentation officielle <http://docs.python.org/2/tutorial/>.

2.2.1 Nombres

```
>>> a = 3.          # On définit un flottant (64 bits par défaut)
>>> b = 7           # On définit un entier (32 bits par défaut)
>>> type(a)        # On demande le type de a
<type 'float'>
>>> type(b)        # On demande le type de b
<type 'int'>
>>> a + b          # On additionne a et b, on remarque que le résultat est un flottant.
10.0
>>> c = a + b      # On assigne à c la valeur a + b
```

2.2.2 Chaînes de caractères

```
>>> mon_texte = 'salade verte' # Une chaîne de caractères
>>> mon_texte[0] # Premier caractère
's'
>>> mon_texte[1] # Second caractère
'a'
>>> mon_texte[-1] # Dernier caractère
'e'
>>> motif = 'Les {0} sont {1}' # Une comportant des balises de formatage
>>> motif.format('lapins', 'rouges') # Formatage de la chaîne
'Les lapins sont rouges'
>>> motif.format('tortues', 5)
'Les tortues sont 5'
```

2.2.3 Listes et dictionnaires

```
>>> ma_liste = [] # On crée une liste vide
>>> ma_liste.append(45) # On ajoute 45 à la fin de la liste.
>>> mon_texte = 'Les lapins ont des grandes oreilles' # On définit une chaîne de caractères nommé mon_texte
>>> ma_liste.append(mon_texte) # On ajoute mon_texte à la fin de ma_liste.
>>> ma_liste # On demande à voir le contenu de ma_liste
[45, 'Les lapins ont des grandes oreilles']
>>> ma_liste[0] # On demande le premier élément de la liste (Python compte à partir de 0)
45
>>> ma_liste[1]
'Les lapins ont des grandes oreilles'
>>> ma_liste[0] = a + b # On écrase le premier élément de ma_liste avec a + b
>>> ma_liste
[10.0, 'Les lapins ont des grandes oreilles']
>>> mon_dict = {} # On définit un dictionnaire
>>> mon_dict['lapin'] = 'rabbit' # On associe à la clé 'lapin' la valeur 'rabbit'
>>> mon_dict[1] = 'one' # On associe à la clé 1 la valeur 'one'
>>> mon_dict
{1: 'one', 'lapin': 'rabbit'}
>>> mon_dict[1]
'one'
>>> mon_dict.keys() # Liste des clés
[1, 'lapin']
>>> mon_dict.values() # Liste des valeurs
['one', 'rabbit']
```

2.2.4 Boucles

On crée un fichier boucles.py :

```
# Boucles en Python

# Boucle FOR
print 'Boucle FOR'
ma_liste = ['rouge', 'vert', 'noir', 56]

for truc in ma_liste:
    print truc # Bien remarquer le decalage de cette ligne (ou indentation) qui delimit le bloc de code

# Boucle IF
print 'Boucle IF'
nombre = raw_input(' 2 + 2 = ')
if nombre == 4:
    print 'Bon'
else:
    print 'Pas bon'

# Boucle WHILE
print 'boucle WHILE'
nombre = 3.
while nombre < 4.:
    nombre = raw_input('Donnez un nombre plus petit que 4: ')
```

2.2.5 Fonctions

On crée un fichier fonctions.py :

```
# Definition d'une fonction
def ma_fonction(x, k = 1.): # On declare la fonction et ses arguments
    """
    Renvoie k*x**2 avec k ayant une valeur par defaut de 1.
    """
    out = k * x**2 # On fait les calculs necessaires
    return out # La commande return permet de renvoyer un resultat

>>> execfile('fonctions.py')
>>> ma_fonction(3)
9.0
>>> ma_fonction(5.)
25.0
>>> ma_fonction(5., k = 5)
125.0
>>> help(ma_fonction)
```

2.2.6 Classes

On crée un fichier classes.py dans lequel on définit une classe de vecteurs qui permet de faire très facilement les opérations usuelles sur les vecteurs :

```
# Creation d'une classe de vecteurs

class vecteur:
    """
    Classe vecteur: decrit le comportement d'un vecteur a 3 dimensions.
    """

    def __init__(self, x = 0., y = 0., z = 0.): # Constructeur: c'est la fonction (ou methode) qui est
        self.x = float(x)
        self.y = float(y)
        self.z = float(z)

    def norme(self): # Une methode qui renvoie la norme
        x, y, z = self.x, self.y, self.z
        return (x**2 + y**2 + z**2)**.5

    def __repr__(self): # On definit comment la classe apparait dans le terminal
        x, y, z = self.x, self.y, self.z
        return '<vecteur: ({0}, {1}, {2})>'.format(x, y, z)

    # Addition
    def __add__(self, other): # On definit le comportement de la classe vis-a-vis de l'addition
        x, y, z = self.x, self.y, self.z
        if type(other) in [float, int]: # Avec un nombre
            return vecteur(x + other, y + other, z + other)
        if isinstance(other, vecteur): # Avec un vecteur
            return vecteur(x + other.x, y + other.y, z + other.z)
    __radd__ = __add__ # On definit l'addition a gauche pour garantir la commutativite

    # Multiplication:
    def __mul__(self, other): # On definit le comportement de la classe vis-a-vis de la multiplication
        x, y, z = self.x, self.y, self.z
        if type(other) in [float, int]: # Avec un nombre
            return vecteur(x * other, y * other, z * other)
        if isinstance(other, vecteur): # Avec un vecteur: produit vectoriel
            x2, y2, z2 = other.x, other.y, other.z
            xo = y * z2 - y2 * z
            yo = z * x2 - z2 * x
            zo = x * y2 - x2 * y
            return vecteur(xo, yo, zo)
    __rmul__ = __mul__ # On definit le produit vectoriel a gauche

    def scalaire(self, other):
        """
        Effectue le produit scalaire entre 2 vecteurs.
        """

        x, y, z = self.x, self.y, self.z
        x2, y2, z2 = other.x, other.y, other.z
        return x * x2 + y * y2 + z * z2

    def normaliser(self):
        """
        Normalise le vecteur.
        """

        x, y, z = self.x, self.y, self.z
        n = self.norme()
        self.x, self.y, self.z = x / n, y / n, z / n
```

```

>>> execfile('classes.py')
>>> v = vecteur(1, 0, 0)
>>> v
<vecteur: (1.0, 0.0, 0.0)>
>>> v + 4
<vecteur: (5.0, 4.0, 4.0)>
>>> w = vecteur(0, 1, 0)
>>> v + w
<vecteur: (1.0, 1.0, 0.0)>
>>> v * w
<vecteur: (0.0, 0.0, 1.0)>
>>> v.scalaire(w)
0.0
>>> q = v + w
>>> q
<vecteur: (1.0, 1.0, 0.0)>
>>> q.norme()
1.4142135623730951
>>> k = vecteur(2, 5, 6)
>>> k.normaliser()
>>> k
<vecteur: (0.248069469178, 0.620173672946, 0.744208407535)>
>>> k.norme()
1.0

```

2.2.7 Fichiers

On veut lire le fichier de données `data.txt` qui comporte 2 colonnes et un nombre de lignes inconnues. La première colonne contient le temps et la seconde une amplitude qui évolue avec le temps. On peut construire une fonction dans le fichier `fichier.py` qui permet de charger les données contenues dans le fichier comme suit :

```

def lire_fichier(nom_fichier):
    """
    Lit le fichier dont le nom est fourni, y cherche 2 colonnes et convertit le tout en nombres.
    """

    fichier = open(nom_fichier, 'r') # Ouverture du fichier. Le second argument specifie le mode d'ouverture.
    lignes = fichier.readlines() # On demande de lire toutes les lignes une par une et de stocker le résultat dans une liste.
    col_0, col_1 = [], [] # On cree 2 listes vides pour recevoir le contenu des colonnes
    for ligne in lignes: # On cree une boucle FOR dont le nombre d'iterations est adapte au nombre de lignes.
        mots = ligne.split() # On casse chaque ligne sur les espaces et tabulations pour obtenir des mots.
        col_0.append(float(mots[0])) # On convertit les mots en nombre avec la commande float et on les ajoute à la liste.
        col_1.append(float(mots[1]))
    return col_0, col_1

nom_fichier = 'data.txt'
temps, amplitude = lire_fichier(nom_fichier) # Et voila...

```

2.3 Modules

Les outils présents dans Python ont vocation à être très généralistes. De très nombreux outils orientés vers des applications particulières existent mais ils ne font pas directement partie du Python, ils sont alors disponibles sous forme de modules ou de packages. A titre d'exemple, si on cherche à utiliser des fonctions mathématiques de base dans Python, on remarque qu'elles n'existent pas :

```
>>> pi
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'pi' is not defined
>>> sin(0.)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'sin' is not defined
```

Cependant, elles sont disponibles dans le module math qui est toujours présent dans Python sans installation supplémentaire (math est un module *built-in*). On peut donc aller chercher ces outils de la manière suivante :

```
>>> from math import sin, pi
>>> pi
3.141592653589793
>>> sin(pi)
1.2246467991473532e-16
>>> import math
>>> math.pi
3.141592653589793
>>> math.sin
<built-in function sin>
```

Dans cette partie, on présente les modules indispensables à une bonne utilisation de Python dans une optique d'ingénieur.

2.3.1 Numpy

Numpy est un module de calcul matriciel, il rend Python comparable à *Matlab*. Il est téléchargeable depuis son site officiel et ou installable depuis les dépôts de paquets sous Linux. Numpy est incontournable pour toutes les opérations qui peuvent être mises sous forme matricielle ou vectoriel pour lesquels il offre une facilité et une vitesse impossible autrement sous Python. L'essence de numpy est la classe array qui ressemble à une liste avec un comportement très différent vis-à-vis des opérateurs mathématiques. De plus, elle est très adaptée au stockage massif de donnée de même type.

```
>>> ma_liste = [1., 3., 5., 10.] # Une liste
>>> ma_liste + ma_liste
[1.0, 3.0, 5.0, 10.0, 1.0, 3.0, 5.0, 10.0] # Somme de liste = concaténation
>>> ma_liste*2
[1.0, 3.0, 5.0, 10.0, 1.0, 3.0, 5.0, 10.0] # Produit liste * entier: concaténation
>>> import numpy as np # On import numpy et on le renomme np par commodité.
>>> mon_array = np.array(ma_liste) # On crée un array à partir de la liste.
>>> mon_array
array([ 1.,  3.,  5., 10.])
>>> mon_array * 2 # array * entier = produit terme à terme
array([ 2.,  6., 10., 20.])
>>> mon_array +5 # array + array = somme terme à terme
array([ 6.,  8., 10., 15.])
>>> mon_array.sum() # Somme du array
19.0
>>> mon_array.mean() # Valeur moyenne
4.75
>>> mon_array.std() # Ecart type
3.344772040064913
>>> np.where(mon_array > 3., 1., 0.) # Seuillage
array([ 0.,  0.,  1.,  1.])
```

2.3.2 Scipy

Scipy (pour Scientific Python) contient un grand nombre d'algorithme classiques en méthodes numériques implémentés en Python de manière performante.

2.3.3 Matplotlib

Matplotlib est un module de graphisme qui produit des figures de grande qualité dans tous les formats utiles. Voici quelques exemples :

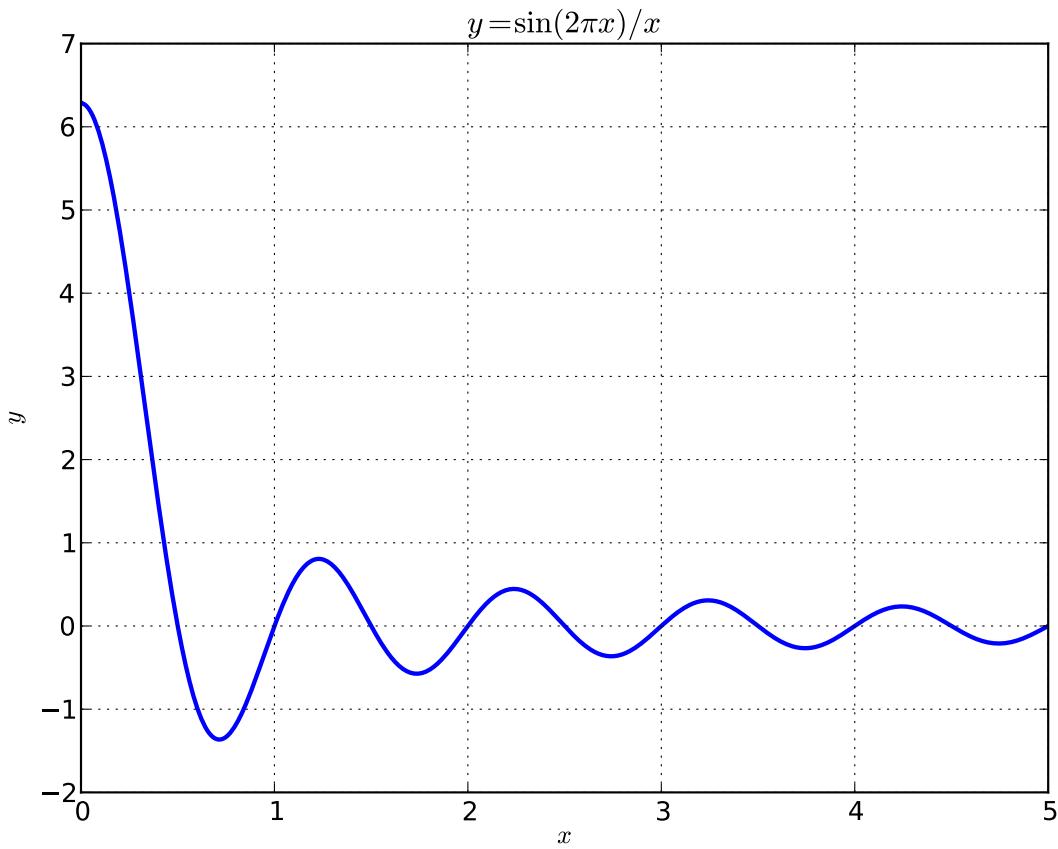
– Tracé d'une courbe $y = \frac{\sin(2\pi x)}{x}$:

```
# PACKAGES
import numpy as np
from matplotlib import pyplot as plt # On import pyplot (un sous module de Matplotlib) et on le renomme plt

# FONCTIONS
def ma_fonction(x):
    """
    Une fonction à tracer.
    """
    return np.sin(2 * np.pi * x) / x

# DEFINITIONS DIVERSES
x = np.linspace(0., 5., 500) # On demande un array contenant 100 points équidistribués entre 0 et 5.
y = ma_fonction(x) # Grâce à numpy, on applique la fonction à tous les points x d'un coup

# TRACE DE LA COURBE
fig = plt.figure() # On crée une figure
plt.clf() # On purge la figure
plt.plot(x, y, 'b-', linewidth = 2.) # On trace y en fonction de x
plt.xlabel('$x$') # On définit le label de l'axe x
plt.ylabel('$y$')
plt.grid() # On demande d'avoir une grille
plt.title(r'$y = \sin(2\pi x) / x$') # On définit le titre et on utilise la syntaxe de LaTeX pour les symboles
plt.show()
```



– Tracé d'un champ $z = \sin(2\pi x) \sin(2\pi y) / \sqrt{x^2 + y^2}$:

```
# PACKAGES
import numpy as np
from matplotlib import pyplot as plt # On import pyplot (un sous module de Matplotlib) et on le renomme plt

# FONCTIONS
def ma_fonction(x,y):
    """
    Une fonction à tracer.
    """
    return np.sin(np.pi * 2 * x) * np.sin(np.pi * 2 * y) / (x**2 + y**2)**.5

# DEFINITIONS DIVERSES
x = np.linspace(1., 5., 100) # On demande un array contenant 100 points équidistribués entre 0 et 5.
y = np.linspace(1., 5., 100)
X, Y = np.meshgrid(x,y) # On crée des grilles X, Y qui couvrent toutes les combinaisons de x et de y
Z = ma_fonction(X, Y) # Grace à numpy, on applique la fonction à tous les points x d'un coup

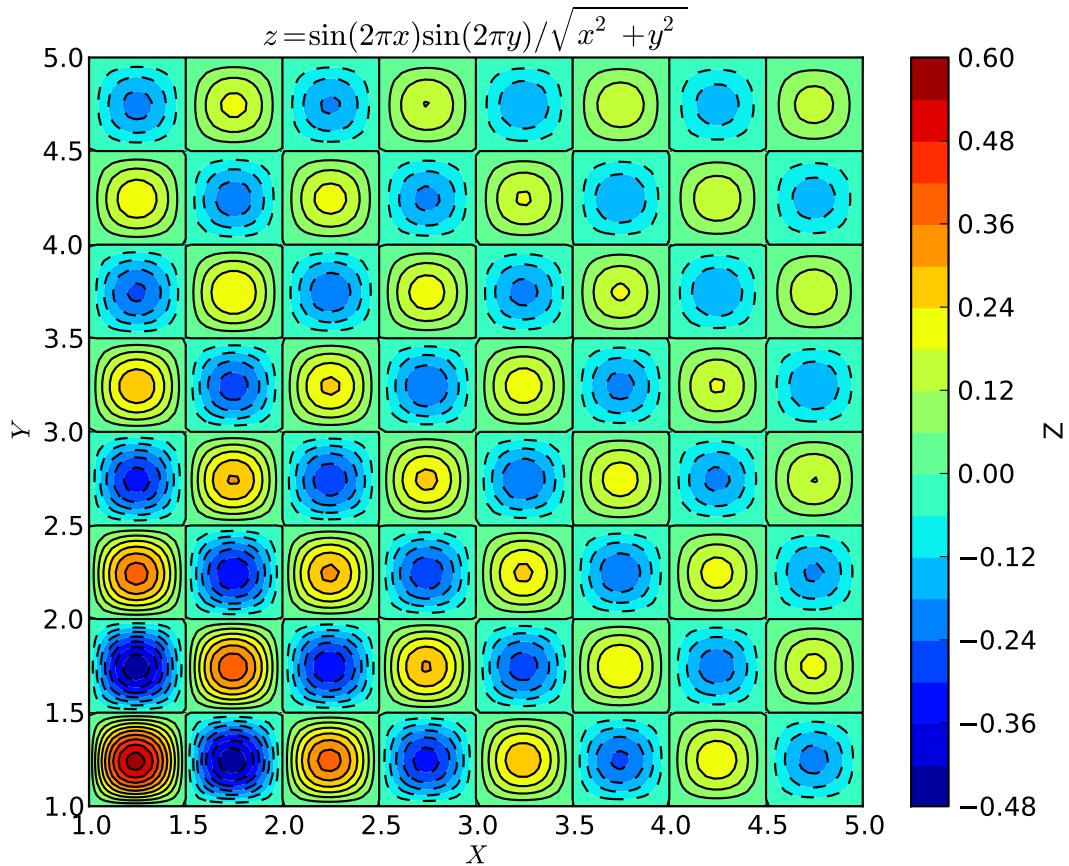
# TRACE DE LA COURBE
niveaux = 20
fig = plt.figure() # On crée une figure
plt.clf()          # On purge la figure
plt.contourf(X, Y, Z, niveaux)
cbar = plt.colorbar()
```

```

plt.contour(X, Y, Z, niveaux, colors = 'black')

cbar.set_label('Z')
plt.xlabel('$X$')      # On definit le label de l'axe x
plt.ylabel('$Y$')
plt.grid()             # On demande d'avoir une grille
plt.title(r'$z = \sin(2\pi x) \sin(2\pi y) / \sqrt{x^2 + y^2}$') # On definit le titre et on utilise la formule
plt.show()

```



Images Numériques

Les images numériques sont des images décrites dans un format numérique. On peut les utiliser pour interpréter quantitativement certaines phénomènes qu'elles mettent en évidence. Cette partie dresse un rapide tableau des différentes approches basiques qui permettent d'effectuer ces tâches.

3.1 Formation

Une image est issue de la mesure d'un phénomène physique particulier par un dispositif adapté. On trouve ainsi des images de différentes natures :

- Lumière visible : [photographie](#) , [microscopie optique](#).
- Lumière infrarouge : [thermographie](#).
- Electrons : [microscopie électronique](#).
- Topologie : [microscopie à force atomique](#).
- Et de nombreux autres...

3.2 Structure

Deux grandes familles d'images numériques existent :

- [Images vectorielles](#) : elles constituées de figures géométriques (droites, polygones, ...). Elles sont idéales pour représenter des schémas et des courbes.
- [Images matricielles](#) : elles sont constituées d'une matrice de **pixels**. Chaque pixel d'une même image porte le même type d'informations. Ces informations sont scindées en **canaux** chacun contenant un nombre qui peut être entier (généralement 8 bits) ou des flotant dans le cas d'images scientifiques. Il est important de noter que la couleur d'un pixel tel qu'il apparaît quand on représente une image n'est pas associé de manière unique à l'information contenue dans le pixel. Par exemple, dans une photographie, on cherche à ce que la représentation du pixel soit fidèle à la vision humaine et donc on va donc la décomposer en 3 canaux (rouge, vert, bleu par exemple) avec éventuellement un quatrième canal destiné à coder la transparence. On parle alors d'image polychrome. A l'opposé dans une image à vocation scientifique, on cherchera généralement à quantifier un phénomène scientifique par un seul canal, si possible sous forme flottante. On parle alors d'image monochrome.

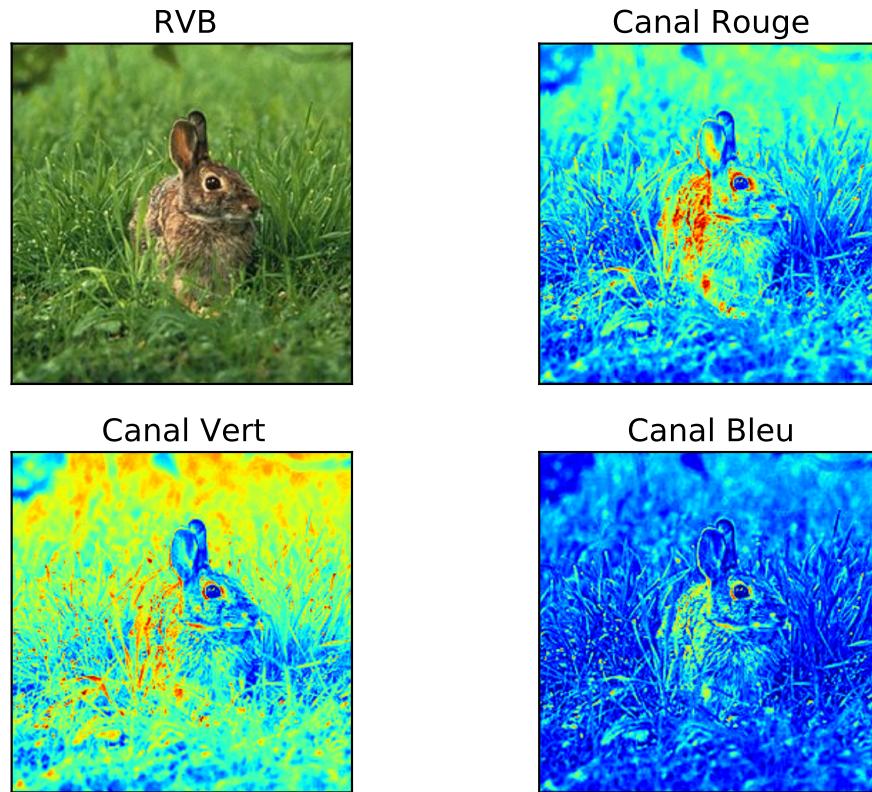
On prend un exemple de photographie : `laping.jpg`

```
#-----
# Decomposition d'une image a polychrome en 3 canaux
#-----
```

```
# PACKAGES
from PIL import Image # On charge Python Image Library
import numpy as np     # On charge Numpy
from matplotlib import pyplot as plt # On charge pyplot (un sous module de Matplotlib) et on le renomme plt

# LECTURE IMAGE
im = Image.open('lapin.jpg') # PIL permet de lire tous les formats d'images
rouge, vert, bleu = im.split()

# AFFICHAGE IMAGE
fig = plt.figure() # On cree une figure
fig.add_subplot(221) # On cree une sous figure
plt.xticks([])
plt.yticks([])
plt.imshow(im,origin='lower')
plt.title('RVB')
fig.add_subplot(222) # On cree une sous figure
plt.xticks([])
plt.yticks([])
plt.imshow(rouge,origin='lower')
plt.title('Canal Rouge')
fig.add_subplot(223) # On cree une sous figure
plt.xticks([])
plt.yticks([])
plt.imshow(vert,origin='lower')
plt.title('Canal Vert')
fig.add_subplot(224) # On cree une sous figure
plt.xticks([])
plt.yticks([])
plt.imshow(bleu,origin='lower')
plt.title('Canal Bleu')
plt.show()
```



On s'intéresse maintenant uniquement à des images monochromes formées de nombres flotants. Ainsi si on dispose d'une photographie, on peut isoler un canal ou construire une combinaison quelconque de canaux comme suit.

```

#-----
# Creation d'une image monochrome a partir d'une photographie
#-----


# PACKAGES
from PIL import Image # On charge Python Image Library
import numpy as np      # On charge Numpy
from matplotlib import pyplot as plt # On charge pyplot (un sous module de Matplotlib) et on le renomme plt

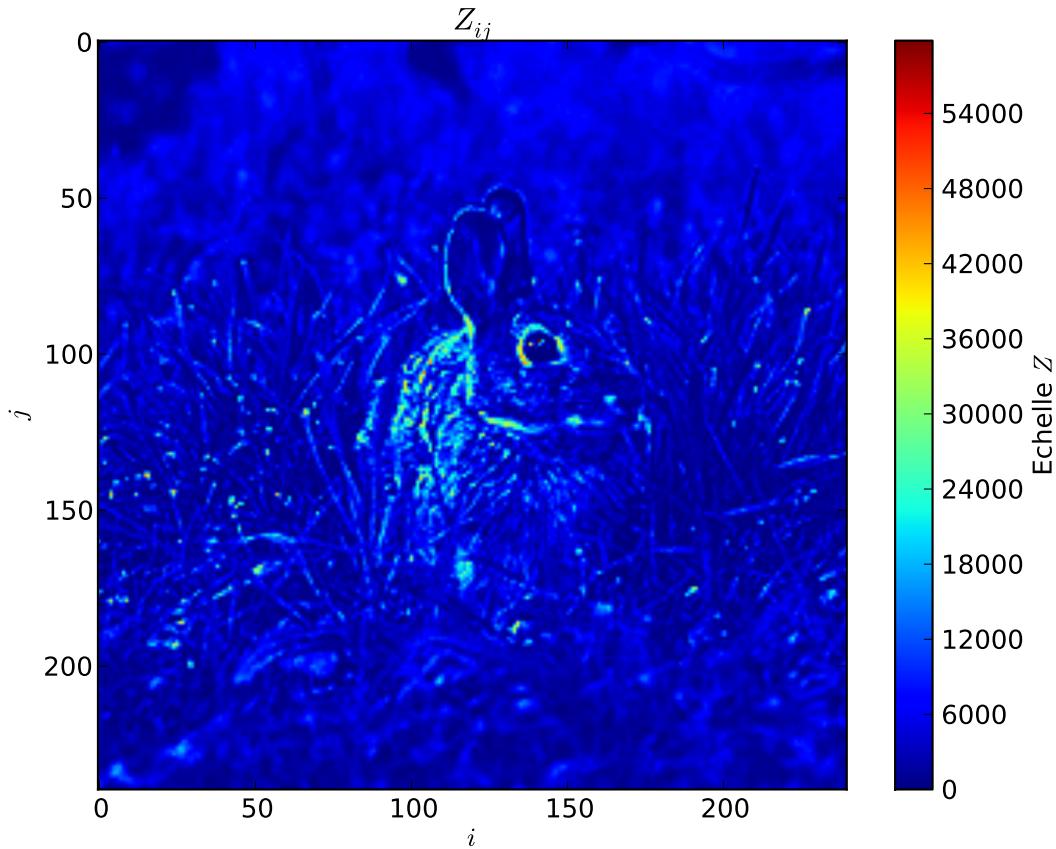
# LECTURE IMAGE
im = Image.open('lapin.jpg') # PIL permet de lire tous les formats d'images
rouge, vert, bleu = im.split()
R = np.array(rouge).astype(np.float64) # On cree des matrices contenant les informations de chaque canal
V = np.array(vert).astype(np.float64)
B = np.array(bleu).astype(np.float64)

# CONSTRUCTION IMAGE MONOCHROME
Z = R + 2 * V + B**2 # On cree un combinaison des canaux R, V et B qu'on nomme Z

# AFFICHAGE IMAGE
fig = plt.figure() # On cree une figure
plt.imshow(Z)
cbar = plt.colorbar()

```

```
cbar.set_label('Echelle $Z$')
plt.xlabel('$i$')
plt.ylabel('$j$')
plt.title('$Z_{ij}$')
plt.show()
```



Une image se résumera donc à une matrice Z_{ij} où i et j sont les indices des pixels. Dans certains cas, on pourra ajouter des informations comme les coordonnées X_{ij} et Y_{ij} des pixels. Toutes ces matrices sont décrites dans le format Python `numpy.array` avec des pixels sous forme `numpy.float64`.

3.3 Opérations

3.3.1 Histogramme

A titre d'exemple, on travaille sur une image d'altitude de l'Europe (source : European Environment Agency) dans laquelle 1 pixel = 1 km × 1 km.

```
#-----
# Carte d'altitude de l'Europe
#-----
```



```
# PACKAGES
```

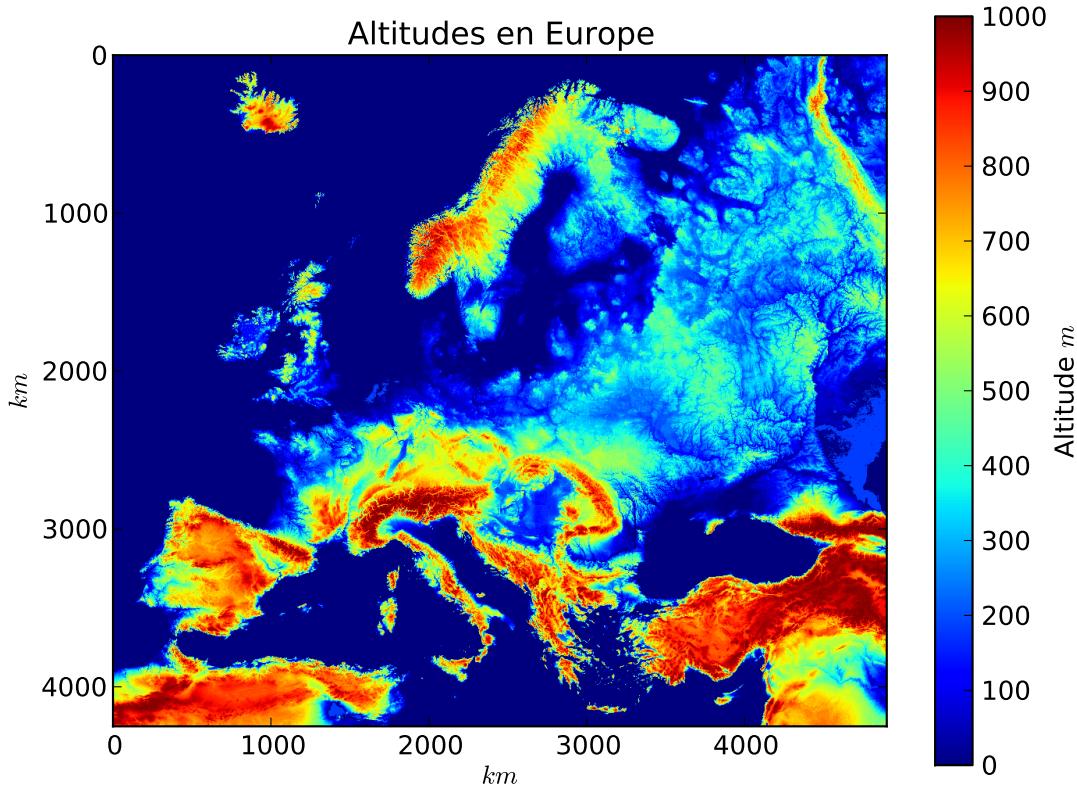
```

from PIL import Image # On charge Python Image Library
import numpy as np    # On charge Numpy
from matplotlib import pyplot as plt # On charge pyplot (un sous module de Matplotlib) et on le renomme plt

# TRAITEMENT IMAGE
im = Image.open('europe.tif')           # PIL permet de lire tous les formats d'images
Z = np.array(im).astype(np.float64)       # On convertit l'image en array
max_altitude = 1000.                     # Altitude maximale en metres, cette donnée est un peu douteuse
Z = Z / Z.max() * max_altitude          # On recalcule les altitudes

# AFFICHAGE
plt.imshow(Z)                          # Affichage de l'altitude
cbar = plt.colorbar()                  # Ajout d'une barre d'échelle
cbar.set_label('Altitude $m$')         # On spécifie le label en z
plt.xlabel('$km$')                    # On spécifie le label en x
plt.ylabel('$km$')                    # On spécifie le label en y
plt.title('Altitudes en Europe')      # On spécifie le titre
plt.show()                            # On affiche l'image

```



On peut alors se demander quelle surface du territoire européen est située dans telle bande d'altitudes de 100m de largeur..

```

#-----
# Histogramme d'altitude de l'Europe
#-----

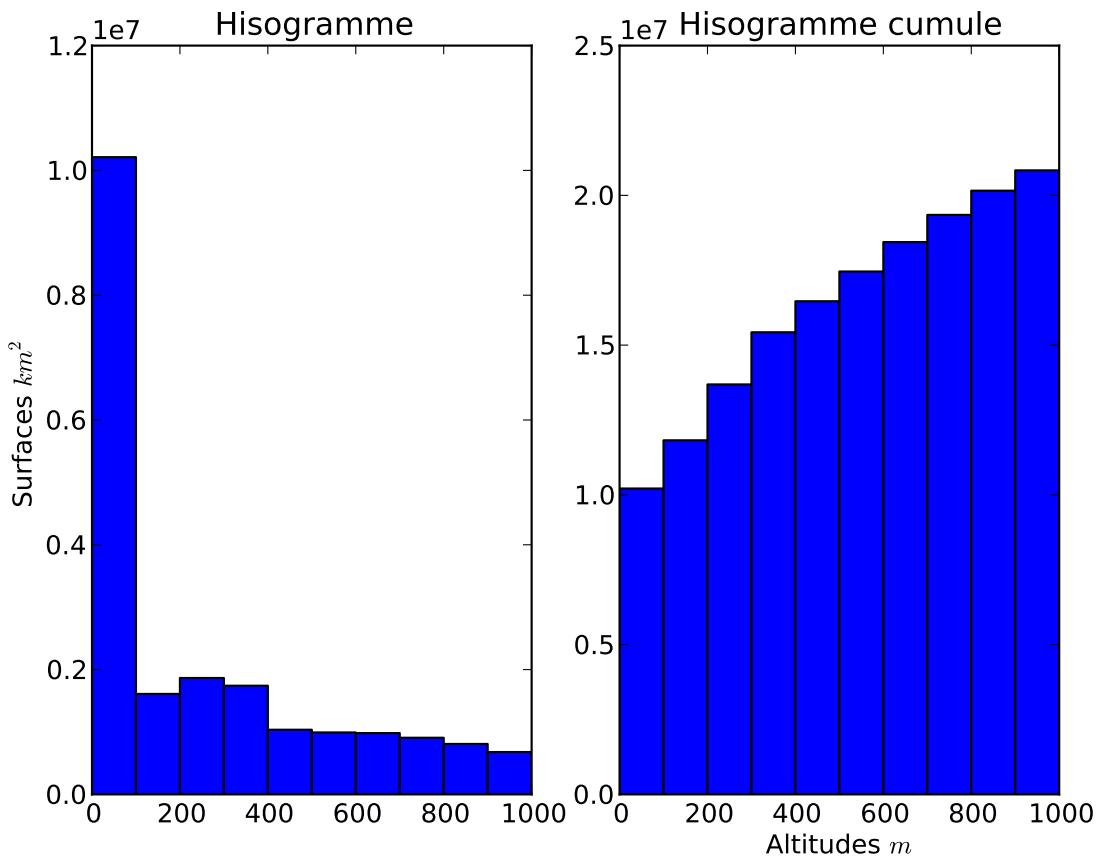
```

```
# PACKAGES
from PIL import Image # On charge Python Image Library
import numpy as np     # On charge Numpy
from matplotlib import pyplot as plt # On charge pyplot (un sous module de Matplotlib) et on le renomme en plt

# TRAITEMENT IMAGE
im = Image.open('europe.tif')           # PIL permet de lire tous les formats d'images
Z = np.array(im).astype(np.float64)       # On convertit l'image en array
max_altitude = 1000.                     # Altitude maximale en metres, cette donnée est un peu douteuse
Z = Z / Z.max() * max_altitude          # On recalcule les altitudes

# AFFICHAGE
n_classes = 10                          # Nombre de classes
fig = plt.figure()
fig.add_subplot(121)
plt.title('Histogramme')
plt.ylabel('Surfaces $km^2$')            # On spécifie le label en y
plt.hist(Z.flatten(), bins=n_classes)    # Histogramme
fig.add_subplot(122)
plt.title('Histogramme cumulé')
plt.hist(Z.flatten(), bins=n_classes, cumulative=True) # Histogramme cumulé
plt.xlabel('Altitudes $m$')              # On spécifie le label en x

plt.show()                                # On affiche l'image
```



3.3.2 Seuillage

L'histogramme montre clairement que la majorité du territoire est située à $Z \leq 100m$. En fait, les zones marines sont à une altitude $Z = 0m$ et contribuent grandement à cette grande surface. Le seuillage consiste à transformer une image monochrome en **image binaire** en appliquant un test booléen à chaque pixel. Une image binaire est ainsi formée de 0 et de 1 ou de **Vrai** et **Faux**. Dans le cas présent, on peut alors chercher à isoler la mer de la terre en effectuant un seuillage $Z > 0m$:

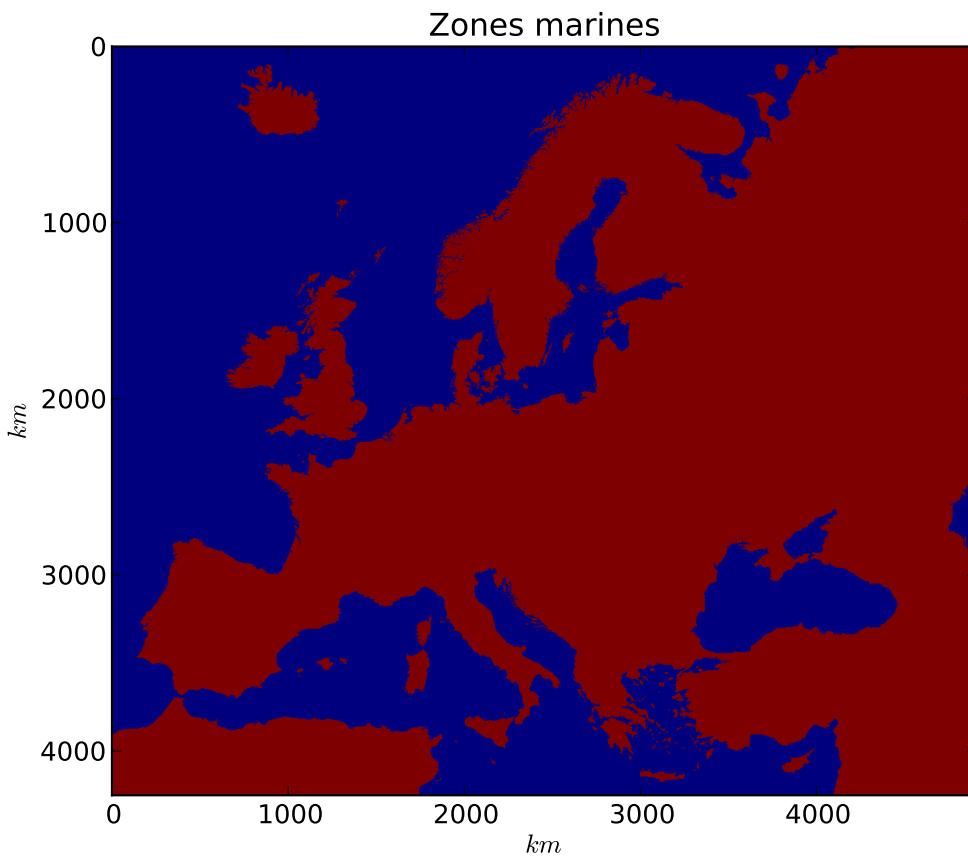
```
#-----
# Recherche des zones marines par seuillage
#-----
```

```
# PACKAGES
from PIL import Image # On charge Python Image Library
import numpy as np # On charge Numpy
from matplotlib import pyplot as plt # On charge pyplot (un sous module de Matplotlib) et on le renomme plt

# TRAITEMENT IMAGE
im = Image.open('europe.tif') # PIL permet de lire tous les formats d'images
Z = np.array(im).astype(np.float64) # On convertit l'image en array
max_altitude = 1000 # Altitude maximale en metres, cette donnée est un peu douteuse
Z = Z / Z.max() * max_altitude # On recalcule les altitudes
```

```
Z = np.where(Z > 0., True, False) # La fonction np.where permet d'appliquer un test booleen a chaque pixel

# AFFICHAGE
plt.imshow(Z) # Affichage de l'altitude
plt.xlabel('km') # On specifie le label en x
plt.ylabel('km') # On specifie le label en y
plt.title('Zones marines') # On specifie le titre
plt.show() # On affiche l'image
```



3.3.3 Erosion / Dilatation

On souhaite mesurer la surface continentale Européenne à l'aide du seuillage effectué précédemment. On remarque de nombreuses îles sont assimilés au continent, on souhaite les éliminer. Pour éliminer cet artefact, les outils issus de la morphologie mathématique tels que l'érosion et la dilatation sont particulièrement adaptés :

```
import numpy as np      # On charge Numpy
from matplotlib import pyplot as plt # On charge pyplot (un sous module de Matplotlib) et on le renomme plt
from scipy.ndimage import morphology # Module de morphologie mathematique de Scipy

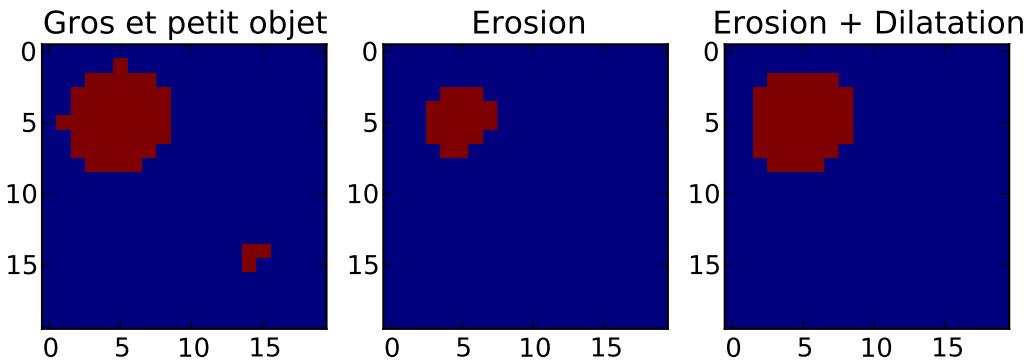
# On construit un cas test
N_pixels = 20
x = np.linspace(0., 1., N_pixels)
X, Y = np.meshgrid(x, x)
```

```

Za = np.where((X-.25)**2 + (Y-.25)**2 < .2**2, 1., 0.)
Zb = np.where((X-.75)**2 + (Y-.75)**2 < .05**2, 1., 0.)
Z = Za + Zb

# On applique l'erosion et la dilatation
structure = np.ones([3,3]) # On definit l'element structurant
Ze = morphology.binary_erosion(Z, structure = structure) + 0. # On applique l'erosion
Zed = morphology.binary_dilation(ze, structure = structure) + 0. # On applique l'erosion
fig = plt.figure()
plt.clf()
fig.add_subplot(131)
plt.title('Gros et petit objet')
plt.imshow(Z, interpolation = 'nearest')
fig.add_subplot(132)
plt.title('Erosion')
plt.imshow(ze, interpolation = 'nearest')
fig.add_subplot(133)
plt.title('Erosion + Dilatation')
plt.imshow(Zed, interpolation = 'nearest')
plt.show()

```



On applique l'érosion-dilatation à la carte de l'Europe :

```

#-----
# Erosion - Dilatation de l'Europe
#-----

```

```
# PACKAGES
from PIL import Image # On charge Python Image Library
import numpy as np     # On charge Numpy
from matplotlib import pyplot as plt # On charge pyplot (un sous module de Matplotlib) et on le renomme plt
from scipy.ndimage import morphology # Module de morphologie mathematique de Scipy

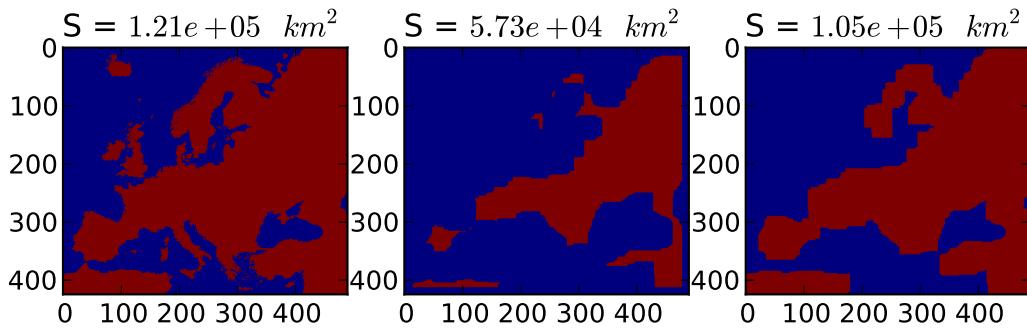
# TRAITEMENT IMAGE
im = Image.open('europe.tif')           # PIL permet de lire tous les formats d'images
Nx, Ny = im.size                      # On reduit la definition de l'image
im = im.resize((Nx/10, Ny/10), Image.ANTIALIAS)
Z = np.array(im).astype(np.float64)      # On convertit l'image en array

max_altitude = 1000.                    # Altitude maximale en metres, cette donnee est un peu douteuse
Z = Z / Z.max() * max_altitude         # On recalcule les altitudes

Z = np.where(Z > 0., 1., 0.) # La fonction np.where permet d'appliquer un test booleen a chaque pixel

structure = np.ones([30,30]) # On definit l'element structurant
Ze = morphology.binary_erosion(Z, structure = structure) + 0.    # On applique l'erosion
Zed = morphology.binary_dilation(ze, structure = structure) + 0. # On applique l'erosion

# AFFICHAGE
fig = plt.figure()
fig.add_subplot(131)
plt.title('S = ${0:1.2e} \ km^2'.format(Z.sum()))
plt.imshow(Z, interpolation = 'nearest')
fig.add_subplot(132)
plt.title('S = ${0:1.2e} \ km^2'.format(ze.sum()))
plt.imshow(ze, interpolation = 'nearest')
fig.add_subplot(133)
plt.title('S = ${0:1.2e} \ km^2'.format(Zed.sum()))
plt.imshow(Zed, interpolation = 'nearest')
plt.show()
```



3.3.4 Comptage

Si on cherche maintenant à extraire les différents zones distinctes (îles et continents), il faut trouver tous les pixels appartenant à la terre $Z = 1$ qui sont voisins. Le comptage de zones dans une image binaire peut se faire par des algorithmes dédiés. Voici un exemple :

```
#-----
# Comptage des îles et continents
#-----
```

```
# PACKAGES
from PIL import Image # On charge Python Image Library
import numpy as np     # On charge Numpy
from matplotlib import pyplot as plt # On charge pyplot (un sous module de Matplotlib) et on le renomme plt
from scipy.ndimage import morphology # Module de morphologie mathématique de Scipy
from scipy.ndimage import measurements # Module de morphologie mathématique de Scipy

# TRAITEMENT IMAGE
im = Image.open('europe.tif')          # PIL permet de lire tous les formats d'images
Nx, Ny = im.size                      # On réduit la définition de l'image
im = im.resize((Nx/5, Ny/5), Image.ANTIALIAS)
Z = np.array(im).astype(np.float64)      # On convertir l'image en array
```

```
max_altitude = 1000.                                # Altitude maximale en metres, cette donnée est un peu douteuse
Z = Z / Z.max() * max_altitude                      # On recale les altitudes

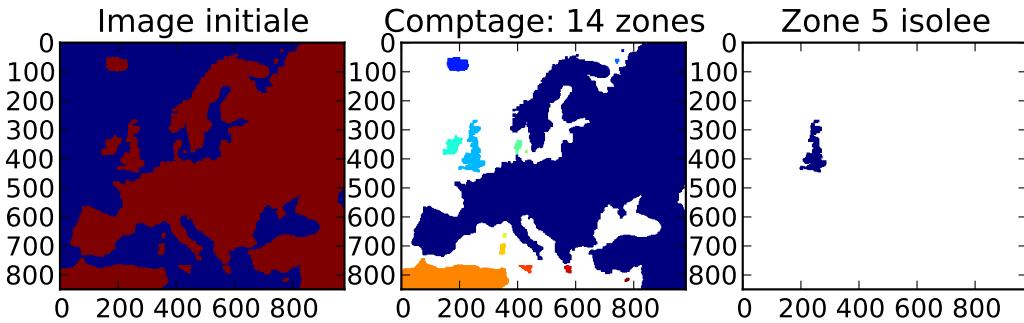
Z = np.where(Z > 0., 1., 0.) # La fonction np.where permet d'appliquer un test booleen a chaque pixel

# EROSION / DILATATION
structure = np.ones([10,10]) # On definit l'element structurant
Z = morphology.binary_erosion(Z, structure = structure) + 0.    # On applique l'erosion
Z = morphology.binary_dilation(Z, structure = structure) + 0. # On applique l'erosion

# COMPTAGE
Zl, nombre = measurements.label(Z) # On compte les zones

# ISOLEMENT D'UNE ZONE
zone = 5 # Label de la zone a isoler
Zli = np.where(Zl == zone, 1, np.nan) # On masque la mer

# AFFICHAGE
fig = plt.figure()
fig.add_subplot(131)
plt.title('Image initiale')
plt.imshow(Z, interpolation = 'nearest')
fig.add_subplot(132)
plt.title('Comptage: {0} zones'.format(nombre))
plt.imshow(np.where(Zl == 0, np.nan, Zl), interpolation = 'nearest')
fig.add_subplot(133)
plt.title('Zone {0} isolee'.format(zone))
plt.imshow(Zli, interpolation = 'nearest')
plt.show()
```



3.3.5 Recherche de contours

Si on cherche maintenant à trouver les cotes européennes, il faut rechercher les contours terres. Pour ce faire le laplacien donne de bons résultats :

```
#-----
# Comptage des îles et continents
#-----
```

```
# PACKAGES
from PIL import Image # On charge Python Image Library
import numpy as np     # On charge Numpy
from matplotlib import pyplot as plt # On charge pyplot (un sous module de Matplotlib) et on le renomme plt
from matplotlib import cm
from scipy.ndimage import morphology # Module de morphologie mathématique de Scipy
from scipy.ndimage import filters # Module de morphologie mathématique de Scipy

# TRAITEMENT IMAGE
im = Image.open('europe.tif')           # PIL permet de lire tous les formats d'images
Nx, Ny = im.size                        # On réduit la définition de l'image
im = im.resize((Nx/5, Ny/5), Image.ANTIALIAS)
Z = np.array(im).astype(np.float64)       # On convertit l'image en array
```

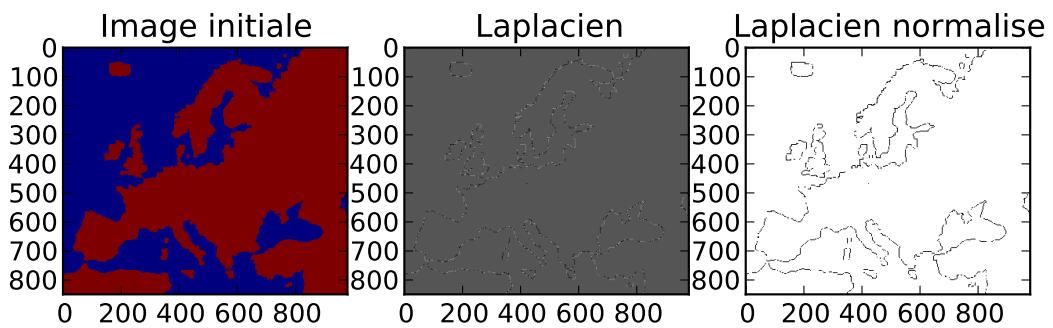
```
max_altitude = 1000.                                # Altitude maximale en metres, cette donnée est un peu douteuse
Z = Z / Z.max() * max_altitude                      # On recalcule les altitudes

Z = np.where(Z > 0., 1., 0.) # La fonction np.where permet d'appliquer un test booleen a chaque pixel

# EROSION / DILATATION
structure = np.ones([10,10]) # On definit l'element structurant
Z = morphology.binary_erosion(Z, structure = structure) + 0.    # On applique l'erosion
Z = morphology.binary_dilation(Z, structure = structure) + 0. # On applique la dilatation

# LAPLACIEN
Zl = filters.laplace(Z)
Zl2 = np.where(Zl != 0., 0, 1)

# AFFICHAGE
fig = plt.figure()
fig.add_subplot(131)
plt.title('Image initiale')
plt.imshow(Z, interpolation = 'nearest')
fig.add_subplot(132)
plt.title('Laplacien')
plt.imshow(Zl, interpolation = 'nearest', cmap = cm.gray)
fig.add_subplot(133)
plt.title('Laplacien normalise')
plt.imshow(Zl2, interpolation = 'nearest', cmap = cm.gray)
plt.show()
```



Les performances de la détection sont meilleures avec un filtre dédié comme le [filtre de Canny](#).

Optimisation

En travaux