Outils Numeriques Cours

Release

Ludovic Charleux, Fabien Formosa

Jan 18, 2018

Contents

Cette partie du cours porte sur le traitement d'image et l'optimisation. Si vous avez des questions ou des remarques, posez les sur les forums du module ou contactez moi directement.

ludovic.charleux@univ-savoie.fr

Contenu du cours:

CHAPTER 1

Python

Python présente plusieurs avantage 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 un grande communauté d'utilisateurs. Tous vos programmes sont donc échangeables sans contraintes. Vos questions trouveront toujours des réponses sur internet. Enfin, très peu de bugs sont présents dans Python.

1.1 Installation

Sous Windows, nous vous conseillons la distribution Anaconda qui permet d'installer tous les outils nécessaires aux exemples traités ici.

1.2 Introduction

1.2.1 Premiers pas

Vous pouvez voir Python comme une grosse calculatrice, vous pouvez y taper des commandes et voir le résultat instantanément:

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

```
Hello World !
```

```
12.0
```

1.2.2 Nombres

```
a = 3.      # On définit un flottant (64 bits par défaut)
b = 7       # On définit un entier (32 bits par défaut)
type(a), type(b)  # On demande le type de a et b
```

```
(float, int)
```

```
a + b       # On additionne a et b, on remarque que le résultat est un flottant.
c = a + b   # On assigne à c la valeur a + b
```

1.2.3 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 chaine
```

```
'Les lapins sont rouges'
```

```
motif.format('tortues', 5)
```

```
'Les tortues sont 5'
```

1.2.4 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 chaine 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']
```

1.2.5 Structures de contrôle (ou boucles)

```
# 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␣
↪delimite le bloc de code qui appartient a la boucle. Dans Python, les blocs sont␣
↪toujours definis par une indentation.

# 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: ')
```

```
Boucle FOR
rouge
vert
noir
56
Boucle IF
 2 + 2 = 4
Pas bon
boucle WHILE
Donnez un nombre plus petit que 4: 4
```

1.2.6 Fonctions

On crée un fichier :download:`fonctions.py <Python/Example_code/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


ma_fonction(3)
```

```
9.0
```

```
ma_fonction(5.)
```

```
25.0
```

```
ma_fonction(5., k = 5)
```

```
125.0
```

```
help(ma_fonction)
```

```
Help on function ma_fonction in module __main__:

ma_fonction(x, k=1.0)
    Renvoie k*x**2 avec k ayant une valeur par defaut de 1.
```

1.2.7 Classes

```
# 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 lancee lors de la creation d'un exemplaire de la classe.
    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
```

```
v = vecteur(1, 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
```

1.2.8 Fichiers

```
f = open("fichier.txt", "wb") # On ouvre un fichier en ecriture
f.write("Very important data") # On ecrit
f.close() # On ferme de fichier
```

```
f = open("fichier.txt", "r") # On ouvre le fichier en lecture
data = f.read()
data
```

```
'Very important data'
```

1.2.9 Modules

Les outils présents dans Python ont vocation à être très généralistes. De très nombreux outils orientés vers des applications particulières existes 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:

```
sin(0)
```

```
---------------------------------------------------------------------------

NameError                                 Traceback (most recent call last)

<ipython-input-33-b5f5b12908fe> in <module>()
----> 1 sin(0)


NameError: name 'sin' is not defined
```

Cependant, elles sont disponibles dans plusieurs libraires (modules ou packages dans Python). Il en existe beaucoup ce qui permet de faire à peu près tout. Ici, nous nous focalisons sur les plus indispensables:

1.2.9.1 Numpy: l'indispensable outil du calcul numérique

```
ma_liste = [1., 3., 5., 10. ] # Une liste
ma_liste + ma_liste # Somme de liste = concaténation
```

```
[1.0, 3.0, 5.0, 10.0, 1.0, 3.0, 5.0, 10.0]
```

```
ma_liste*2 # Produit = concaténation aussi....
```

```
[1.0, 3.0, 5.0, 10.0, 1.0, 3.0, 5.0, 10.0]
```

```
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 avec le très puissant where
```

```
array([ 0.,  0.,  1.,  1.])
```

1.2.9.2 Scipy

Scipy ou scientific Python est une bibliothèque qui contient la majorité des agorithmes classiques en méthodes numériques.

1.2.9.3 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
%matplotlib nbagg

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


# DEFINITIONS DIVERSES
x = np.linspace(1., 10., 500) # On demande un array contenant 100 points equirepartis␣
↪entre 0 et 5.
y = ma_fonction(x) # Grace a numpy, on applique la fonction a tous les points x d'un␣
↪coup

# TRACE DE LA COURBE
fig = plt.figure() # On cree 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 definit 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 definit le titre et on utilise la syntaxe␣
↪de LaTeX pour y introduire des maths.
plt.show()
```

```
<IPython.core.display.Javascript object>
```

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

```
# FONCTIONS
def ma_fonction(x,y):
  '''
  Une fonction a 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 equirepartis␣
↪entre 0 et 5.
y = np.linspace(1., 5., 100)
X, Y = np.meshgrid(x,y) # On cree des grilles X, Y qui couvrent toutes les␣
↪combinaisons de x et de y
Z = ma_fonction(X, Y) # Grace a numpy, on applique la fonction a tous les points x d
↪'un coup

# TRACE DE LA COURBE
niveaux = 20
fig = plt.figure() # On cree 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 syntaxe de LaTeX pour y introduire des maths.
plt.show()
```

```
<IPython.core.display.Javascript object>
```

CHAPTER 2

ODEs

Tutorial (TD) and practical work (TP) notebook file can be downloaded here:

- `ODE_pendulum.ipynb`
- `ODE_harmonic_oscillator.ipynb`

```
import numpy as np
import matplotlib.pyplot as plt
import matplotlib as mpl
%matplotlib nbagg
```

2.1 Ordinary differential equations (ODE)

2.1.1 Scope

- Widely used in physics
- Closed form solutions only in particular cases
- Need for numerical solvers

2.1.2 Ordinary differential equations vs. partial differential equation

2.1.2.1 Ordinary differential equations (ODE)

Derivatives of the inknown function only with respect to a single variable, time t for example.

- Example: the 1D linear oscillator equation

$$\frac{d^2x}{dt^2} + 2\zeta\omega_0\frac{dx}{dt} + \omega_0 x = 0$$

2.1.2.2 Partial differential equations (PDE)

Derivatives of the unknown function with respect to several variables, time t and space (x, y, z) for example. Special techniques not introduced in this course need to be used, such as finite difference or finite elements.

- Example : the heat equation

$$\rho C_p \frac{\partial T}{\partial t} - k\Delta T + s = 0$$

2.1.3 Introductive example

Point mass P in free fall.

Required data:

- gravity field $\vec{g} = (0, -g)$,
- Mass m,
- Initial position $P_0 = (0, 0)$
- Initial velocity $\vec{V_0} = (v_{x0}, v_{y0})$

Problem formulation:

$$\begin{cases} \\ \end{cases} \quad \begin{aligned} \ddot{x} &= 0 \\ \ddot{y} &= -g \end{aligned}$$

2.1.4 Closed form solution

$$\begin{cases} \\ \end{cases} \quad \begin{aligned} x(t) &= v_{x0}t \\ y(t) &= -g\frac{t^2}{2} + v_{y0}t \end{aligned}$$

```
tmax = .2
t  = np.linspace(0., tmax, 1000)
x0, y0   = 0., 0.
vx0, vy0 = 1., 1.
g = 10.
x = vx0 * t
y = -g  * t**2/2. + vy0 * t
fig = plt.figure()
ax.set_aspect("equal")
plt.plot(x, y, label = "Exact solution")
plt.grid()
plt.xlabel("x")
plt.ylabel("y")
plt.legend()
plt.show()
```

```
<IPython.core.display.Javascript object>
```

2.1.5 Reformulation

Any ODEs can be reformulated as a first order system equations. Let's assume that

$$X = \begin{bmatrix} X_0 \\ X_1 \\ X_2 \\ X_3 \end{bmatrix} = \begin{bmatrix} x \\ y \\ \dot{x} \\ \dot{y} \end{bmatrix}$$

As a consequence:

$$\dot{X} = \begin{bmatrix} \dot{x} \\ \dot{y} \\ \ddot{x} \\ \ddot{y} \end{bmatrix}$$

Then, the initialy second order equation can be reformulated as:

$$\dot{X} = f(X,t) = \begin{bmatrix} X_2 \\ X_3 \\ 0 \\ -g \end{bmatrix}$$

Generic problem

Solving $\dot{Y} = f(Y,t)$

2.1.6 Numerical integration of ODE

Generic formulation

$$\dot{X} = f(X,t)$$

- approximate solution: need for error estimation
- discrete time: $t_0, t_1, \ldots$
- time step $dt = t_{i+1} - t_i$,

2.1.7 Euler method

- Intuitive
- Fast
- Slow convergence

$$X_{i+1} = X_i + f(X,t_i)dt$$

```
dt = 0.02 # Pas de temps
X0 = np.array([0., 0., vx0, vy0])
nt = int(tmax/dt) # Nombre de pas
ti = np.linspace(0., nt * dt, nt)

def derivate(X, t):
  return np.array([X[2], X[3], 0., -g])
```

```
def Euler(func, X0, t):
  dt = t[1] - t[0]
  nt = len(t)
  X  = np.zeros([nt, len(X0)])
  X[0] = X0
  for i in range(nt-1):
    X[i+1] = X[i] + func(X[i], t[i]) * dt
  return X

%time X_euler = Euler(derivate, X0, ti)
x_euler, y_euler = X_euler[:,0], X_euler[:,1]

plt.figure()
plt.plot(x, y, label = "Exact solution")
plt.plot(x_euler, y_euler, "or", label = "Euler")
plt.grid()
plt.xlabel("x")
plt.ylabel("y")
plt.legend()
plt.show()
```

```
CPU times: user 0 ns, sys: 0 ns, total: 0 ns
Wall time: 227 µs
```

```
<IPython.core.display.Javascript object>
```

2.1.8 Runge Kutta 4

Wikipedia

Evolution of the Euler integrator with:

- Multiple slope evaluation (4 here),
- Well chosen weighting to match simple solutions.

$$X_{i+1} = X_i + \frac{dt}{6}(k_1 + 2k_2 + 2k_3 + k_4)$$

With:

- k_1 is the increment based on the slope at the beginning of the interval, using $ X $ (Euler's method);
- k_2 is the increment based on the slope at the midpoint of the interval, using $ X + dt/2 :raw-latex:‘times ‘k_1 $;
- k_3 is again the increment based on the slope at the midpoint, but now using $ X + dt/2:raw-latex:‘times ‘k_2 $;
- k_4 is the increment based on the slope at the end of the interval, using $ X + dt :raw-latex:‘times ‘k_3 $.

```
def RK4(func, X0, t):
  dt = t[1] - t[0]
  nt = len(t)
  X  = np.zeros([nt, len(X0)])
  X[0] = X0
  for i in range(nt-1):
    k1 = func(X[i], t[i])
    k2 = func(X[i] + dt/2. * k1, t[i] + dt/2.)
    k3 = func(X[i] + dt/2. * k2, t[i] + dt/2.)
    k4 = func(X[i] + dt    * k3, t[i] + dt)
```

```
    X[i+1] = X[i] + dt / 6. * (k1 + 2. * k2 + 2. * k3 + k4)
  return X

%time X_rk4 = RK4(derivate, X0, ti)
x_rk4, y_rk4 = X_rk4[:,0], X_rk4[:,1]

plt.figure()
plt.plot(x, y, label = "Exact solution")
plt.plot(x_euler, y_euler, "or", label = "Euler")
plt.plot(x_rk4, y_rk4, "gs", label = "RK4")
plt.grid()
plt.xlabel("x")
plt.ylabel("y")
plt.legend()
plt.show()
```

```
CPU times: user 0 ns, sys: 0 ns, total: 0 ns
Wall time: 444 µs
```

```
<IPython.core.display.Javascript object>
```

2.1.9 Using ODEint

http://docs.scipy.org/doc/scipy-0.16.0/reference/generated/scipy.integrate.odeint.html

```
from scipy import integrate

X_odeint = integrate.odeint(derivate, X0, ti)
%time x_odeint, y_odeint = X_odeint[:,0], X_rk4[:,1]

plt.figure()
plt.plot(x, y, label = "Exact solution")
plt.plot(x_euler, y_euler, "or", label = "Euler")
plt.plot(x_rk4, y_rk4, "gs", label = "RK4")
plt.plot(x_odeint, y_odeint, "mv", label = "ODEint")

plt.grid()
plt.xlabel("x")
plt.ylabel("y")
plt.legend()
plt.show()
```

```
CPU times: user 0 ns, sys: 0 ns, total: 0 ns
Wall time: 11.9 µs
```

```
<IPython.core.display.Javascript object>
```

2.1.10 Tutorial (TD)

In this example, you have to model and animate a pendulum.

1. Write the constitutive equations.
2. Reformulate the equations as a first order system of ODEs.

3. Solve the problem using Euler, RK4 and ODE integrators.
4. Compare the results.

2.2 Tutorial: The simple pendulum

2.2.1 Introduction

This tutorial aims at modelling and solving the yet classical but not so simple problem of the pendulum. A representation is given bellow (source: Wikipedia).

Fig. 2.1: The simple pendulum

On a mechanical point of view, the mass m is supposed to be concentrated at the lower end of the rigid arm. The length of the arm is noted l. The angle between the arm and the vertical direction is noted θ. A simple modelling using dynamics leads to:

$$\Gamma = \vec{P} + \vec{T}$$

Where:

- $\vec{\Gamma}$ is the acceleration of the mass,
- $\vec{P}$ if the weight of the mass,
- $\vec{T}$ if the reaction force of the arm.

A projection of this equation along the direction perpendicular to the arm gives a more simple equation:

$$\ddot{\theta} = \frac{g}{l} \sin \theta$$

This equation is a second order, non linear ODE. The closed form solution is only known when the equation is linearized by assuming that θ is small enough to write that $\sin \theta \approx \theta$. In this tutorial, we will solve this problem with a numerical approach that does not require such simplification.

```
# Setup
%matplotlib nbagg
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from scipy.integrate import odeint
```

2.2.1.1 Numerical values

```
l = 1.    # m
g = 9.81 # m/s**2
```

2.2.2 Part 1: Reformulation of the problem

- This problem can be reformulated to match the standard formulation $\dot{X} = f(X, t)$:

$$X = \begin{bmatrix} \theta \\ \dot{\theta} \end{bmatrix} = \begin{bmatrix} x_0 \\ x_1 \end{bmatrix}$$

$$\dot{X} = \begin{bmatrix} x_1 \\ -\frac{g}{l} \sin x_0 \end{bmatrix} = f(X, t)$$

- Write the function f in Python:

```
def f(X, t):
    """
    The derivative function
    """
    # To be completed
    return
```

2.2.3 Part 3: Numerical solution

Solve the problem with Euler, RK4 and ODEint integrators and compare the results. First assume that the pendulum is released with no speed ($\dot{\theta} = 0^o/s$) at $\theta = 10^o$. The time discretization will be as follows:

- duration: 10 s,
- time step: 0.01 s.

```
def Euler(func, X0, t):
    """
    Euler integrator.
    """
    dt = t[1] - t[0]
    nt = len(t)
    X  = np.zeros([nt, len(X0)])
    X[0] = X0
    for i in range(nt-1):
        X[i+1] = X[i] + func(X[i], t[i]) * dt
    return X

def RK4(func, X0, t):
    """
    Runge and Kutta 4 integrator.
    """
    dt = t[1] - t[0]
    nt = len(t)
    X  = np.zeros([nt, len(X0)])
    X[0] = X0
    for i in range(nt-1):
        k1 = func(X[i], t[i])
        k2 = func(X[i] + dt/2. * k1, t[i] + dt/2.)
        k3 = func(X[i] + dt/2. * k2, t[i] + dt/2.)
        k4 = func(X[i] + dt    * k3, t[i] + dt)
        X[i+1] = X[i] + dt / 6. * (k1 + 2. * k2 + 2. * k3 + k4)
    return X

# ODEint is preloaded.
```

```
# Define the time vector t and the initial conditions X0
```

2.2.4 Part 4: Energies an errors

Calculate and plot the kinetic energy E_c, the potential energy E_p and the total energy $E_t = E_c + E_p$ for all 3 cases, plot it and comment.

2.3 Ordinary Differential Equations : Practical work on the harmonic oscillator

In this example, you will simulate an harmonic oscillator and compare the numerical solution to the closed form one.

2.3.1 Theory

Read about the theory of harmonic oscillators on Wikipedia

2.3.1.1 Mechanical oscillator

The case of the one dimensional mechanical oscillator leads to the following equation:

$$m\ddot{x} + \mu\dot{x} + kx = m\ddot{x}_d$$

Where:

- x is the position,
- $\dot{x}$ and $\ddot{x}$ are respectively the speed and acceleration,
- m is the mass,
- μ the
- k the stiffness,
- and $\ddot{x}_d$ the driving acceleration which is null if the oscillator is free.

2.3.1.2 Canonical equation

Most 1D oscilators follow the same canonical equation:

$$\ddot{x} + 2\zeta\omega_0\dot{x} + \omega_0^2 x = \ddot{x}_d$$

Where:

- ω_0 is the undamped pulsation,
- ζ is damping ratio,
- $\ddot{x}_d$ is the imposed acceleration.

In the case of the mechanical oscillator:

$$\omega_0 = \sqrt{\frac{k}{m}}$$

$$\zeta = \frac{\mu}{2\sqrt{mk}}$$

2.3.1.3 Undampened oscillator

First, you will focus on the case of an undamped free oscillator ($\zeta = 0$, $\ddot{x}_d = 0$) with the following initial conditions:

$$\begin{cases} x(t=0) = 1 \\ \dot{x}(t=0) = 0 \end{cases}$$

The closed form solution is:

$$x(t) = a \cos \omega_0 t$$

```
# Setup
%matplotlib inline
import numpy as np
import matplotlib.pyplot as plt
from scipy.integrate import odeint

# Setup
f0     = 1.
omega0 = 2. * np.pi * f0
a      = 1.
```

2.3.2 Part 1: theoretical solution

Plot the closed form solution of the undamped free oscillator for 5 periods.

Steps:

1. Create an array t reprenting time,
2. Create a function x_{th} representing the amplitude of the closed form solution,
3. Plot x_{th} vs t.

```
# Complete here
#t =
#xth =
```

2.3.3 Part 2: Numerical solution with Euler integrator

Solve the problem introduced in question 1 with the Euler integrator.

Steps:

1. Rewrite the canonical equation as a system of first order ODEs depending of the variable $X = [x, \dot{x}]$,
2. Code the derivative function $f(X, t) = \dot{X}$,
3. Define initial conditions X_0,
4. Solve the problem.
5. Plot the position x along and compare it with the theoretical solution.

2.3.4 Part 3: Energies an errors

Calculate and plot the kinetic energy E_c, the potential energy E_p and the total energy $E_t = E_c + E_p$, comment the result.

Steps:

1. Calculate E_c,
2. Calculate E_p,
3. Calculate E_t,
4. Plot the evolution of the 3 energies. You can use *plt.fill_between* instead of *plt.plot*,
5. Use the results to define a relative error estimator base on energies.

2.3.5 Part 4: Numerical solution convergence

Plot the effect of the number time steps n_t on the error e.

Steps:

1. Create an array containing the different number of time steps from 100 to 100000,
2. Loop over this array and calculate the the error for each configuration,
3. Plot the error as a function of n_t.

2.3.6 Part 5: integrator benchmark

Rewrite the code of part 4 in order to compare the RK4 and ODEint solvers with the Euler solver. Comment the efficiency of each solver.

2.3.7 Part 6: Error *vs.* time

Modify the code of part 5 in order to measure the computing time of each method in each case. Plot the error *vs.* the computing time.

2.4 Ordinary differential equations: solar system

```
import numpy as np
import matplotlib.pyplot as plt
import matplotlib as mpl
%matplotlib inline
from scipy import integrate
from matplotlib import animation, rc
#from IPython.display import HTML
rc('animation', html='html5')
```

```
# -*- coding: utf-8 -*-
"""
A Solar System class
"""
```

```
import numpy as np
from scipy.integrate import odeint

class System(object):
  def __init__(self, m, p, v, nk = 10000, G = 6.67e-11):
    """
    Syteme solaire en 2D:

    * m: masse de chaque objet
    * p: position de chaque objet
    * v: vitesse de chaque objet
    * G: constance de gravitation universelle
    """
    n = len(p)
    self._n = n
    self.Y = np.zeros([nk, 4 * n])
    self.Y.fill(np.NAN)
    self.Y[-1, :2 * n] = np.array(p).flatten()
    self.Y[-1, 2 * n:] = np.array(v).flatten()
    self.m  = np.array(m)
    self.nk = nk
    self.G  = G

  def derivative(self, y, t):
    """
    Acceleration de chaque masse !
    """
    m, G = self.m, self.G
    n = len(m)
    p = y[:2 * n ].reshape(n ,2)
    v = y[ 2 * n:].reshape(n ,2)
    a = np.zeros_like(p) # vecteur plein de zeros dans le mm format que p
    n = len(m) # nombre de masses
    for i in range(n): # On s'intéresse à la masse i
      for j in range(n): # Les masses j agissent dessus
        if i != j: # i ne doit pas agir sur i !
           PiPj = p[j] - p[i]
           rij = (PiPj**2).sum()**.5
           if rij != 0. :
             a[i] += G * m[j] * PiPj / rij**3
    y2 = y.copy()
    y2[:2*n ] = v.flatten()
    y2[ 2*n:] = a.flatten()
    return y2


  def solve(self, dt, nt):
    time = np.linspace(0., dt, nt + 1)
    Ys = odeint( self.derivative, self.Y[-1], time)
    nk = self.nk
    Y = self.Y
    Y[:nk - nt] = Y[nt:]
    Y[-nt-1:] = Ys
    self.Y    = Y

  def xy(self):
    n = self._n
    p = self.Y[-1,:2 * n].reshape(n, 2)
```

```
    return p[:,0], p[:,1]

  def trail(self, i):
    n = self._n
    Y = self.Y
    return Y[:, 2*i], Y[:, 2*i +1 ]
```

```
G = 1.

nm = 6
m = np.ones(nm)*1.e0
ms = 100. # Mass of the sun
theta = np.linspace(0., 2. * np.pi, nm)
r = np.ones(nm)
v = (G * ms / r)**.5
v *= .6
v[1:] *= np.random.normal(loc = 1., scale = .002, size = nm-1)
x  =   r * np.cos(theta)
y  =   r * np.sin(theta)
vx = - v * np.sin(theta)
vy =   v * np.cos(theta)
P = np.array([x,   y]).transpose()
V = np.array([vx, vy]).transpose()
colors = "yrgbcmk"
# Setting up a sun
m[0]  = ms
P[0] *= 0.
V[0] *= 0.

nm = len(m)
s = System(m, P, V, G = G, nk = 5000)
dt = 0.005
nt = 100
s.solve(dt, nt)


from matplotlib import animation
fig = plt.figure("Le systeme solaire")
plt.clf()
ax = fig.add_subplot(1,1,1)
ax.set_aspect("equal")
plt.xlim(-2., 2.)
plt.ylim(-2., 2.)
plt.grid()
planets = []

msize = 10. * (s.m / s.m.max())**(1./6.)
for i in range(nm):
  lc = len(colors)
  c = colors[i%lc]
  planet, = ax.plot([], [], "o"+c, markersize = msize[i])
  planets.append(planet)
  trail, = ax.plot([], [], "-"+c)
  planets.append(trail)

def init():
  for i in range(2 * nm):
    planets[i].set_data([], [])
```

```
    return planets

def animate(i):
    s.solve(dt, nt)
    x, y = s.xy()
    for i in range(nm):
      planets[2*i].set_data(x[i:i+1], y[i:i+1])
      xt, yt = s.trail(i)
      planets[2*i+1].set_data(xt, yt)
    return planets

anim = animation.FuncAnimation(fig, animate, init_func=init, frames=200, interval=20,␣
↪blit=True)

#anim.save('basic_animation.mp4', fps=30, extra_args=['-vcodec', 'libx264'])

plt.close()
anim
```

CHAPTER 3

Interpolation

3.1 1D interpolation

3.1.1 Scope

- Finite number N of data points are available: $P_i = (x_i, y_i)\,, i \in \{0, \ldots, N\}$
- Interpolation is about filling the gaps by building back the function $y(x)$

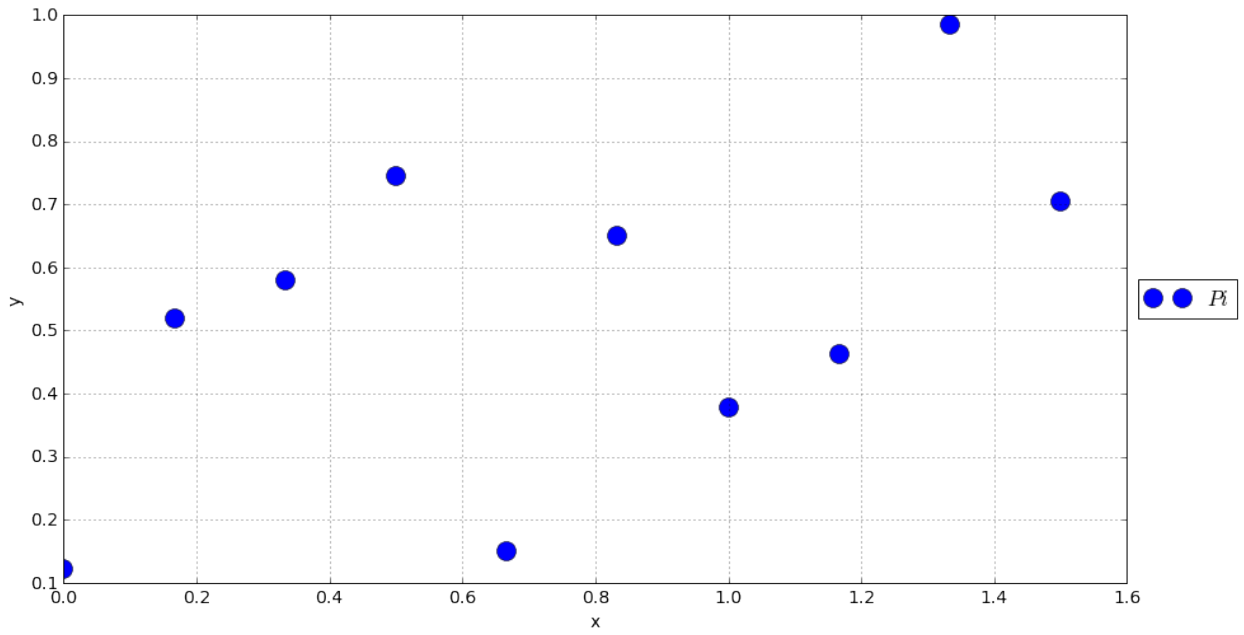
https://en.wikipedia.org/wiki/Interpolation

```
# Setup
%matplotlib inline
import numpy as np
import matplotlib.pyplot as plt
import matplotlib
params = {'font.size'     : 14,
          'figure.figsize':(15.0, 8.0),
          'lines.linewidth': 2.,
          'lines.markersize': 15,}
matplotlib.rcParams.update(params)
```

3.1.2 Let's do it with Python

```
N = 10
xmin, xmax = 0., 1.5
xi = np.linspace(xmin, xmax, N)
yi = np.random.rand(N)

plt.plot(xi,yi, 'o', label = "$Pi$")
plt.grid()
plt.xlabel("x")
plt.ylabel("y")
```

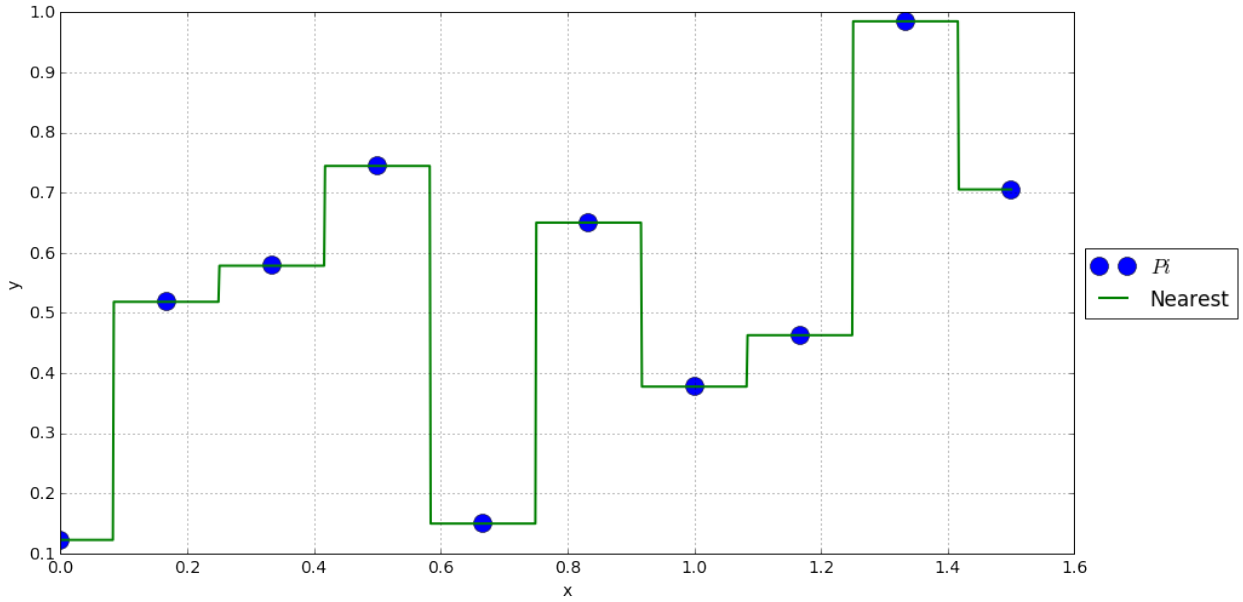
```
plt.legend(loc='center left', bbox_to_anchor=(1, 0.5))
plt.show()
```



3.1.3 Nearest (*aka.* piecewise) interpolation

Function $y(x)$ takes the value y_i of the nearest point P_i on the x direction.

```
from scipy import interpolate
x = np.linspace(xmin, xmax, 1000)
interp = interpolate.interp1d(xi, yi, kind = "nearest")
y_nearest = interp(x)

plt.plot(xi,yi, 'o', label = "$Pi$")
plt.plot(x, y_nearest, "-", label = "Nearest")
plt.grid()
plt.xlabel("x")
plt.ylabel("y")
plt.legend(loc='center left', bbox_to_anchor=(1, 0.5))
plt.show()
```



3.1.3.1 Pros

- $y(x)$ only takes values of existing y_i.

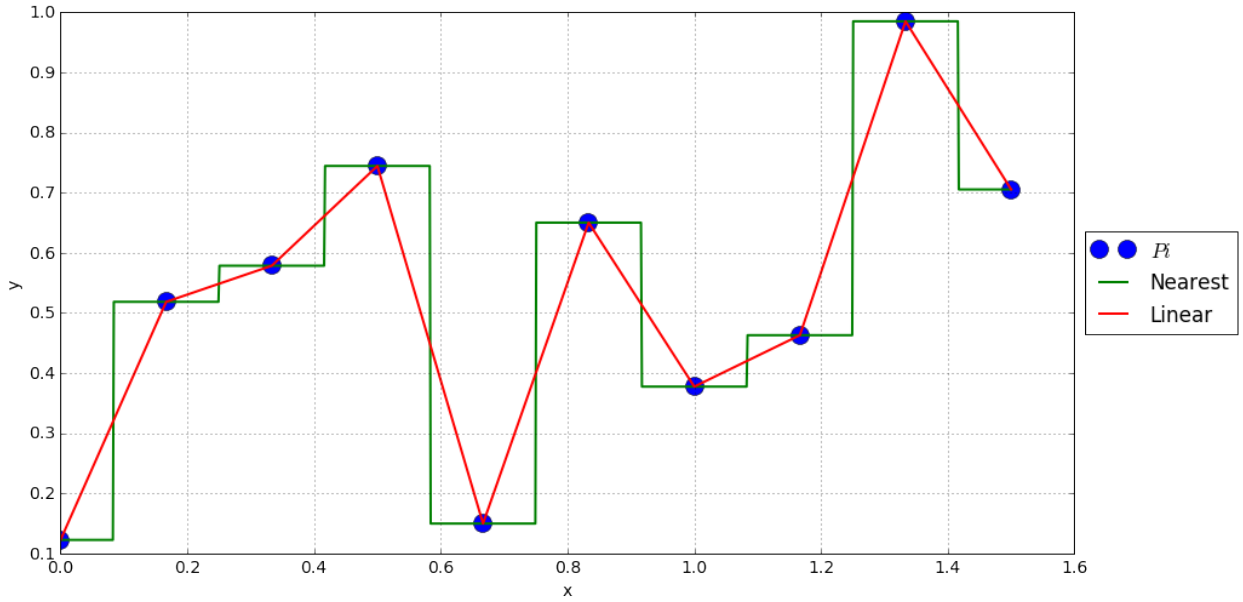
3.1.3.2 Cons

- Discontinuous

3.1.4 Linear interpolation

Function $y(x)$ depends linearly on its closest neighbours.

```
from scipy import interpolate
x = np.linspace(xmin, xmax, 1000)
interp = interpolate.interp1d(xi, yi, kind = "linear")
y_linear = interp(x)

plt.plot(xi,yi, 'o', label = "$Pi$")
plt.plot(x, y_nearest, "-", label = "Nearest")
plt.plot(x, y_linear, "-", label = "Linear")
plt.grid()
plt.xlabel("x")
plt.ylabel("y")
plt.legend(loc='center left', bbox_to_anchor=(1, 0.5))
plt.show()
```



3.1.4.1 Pros

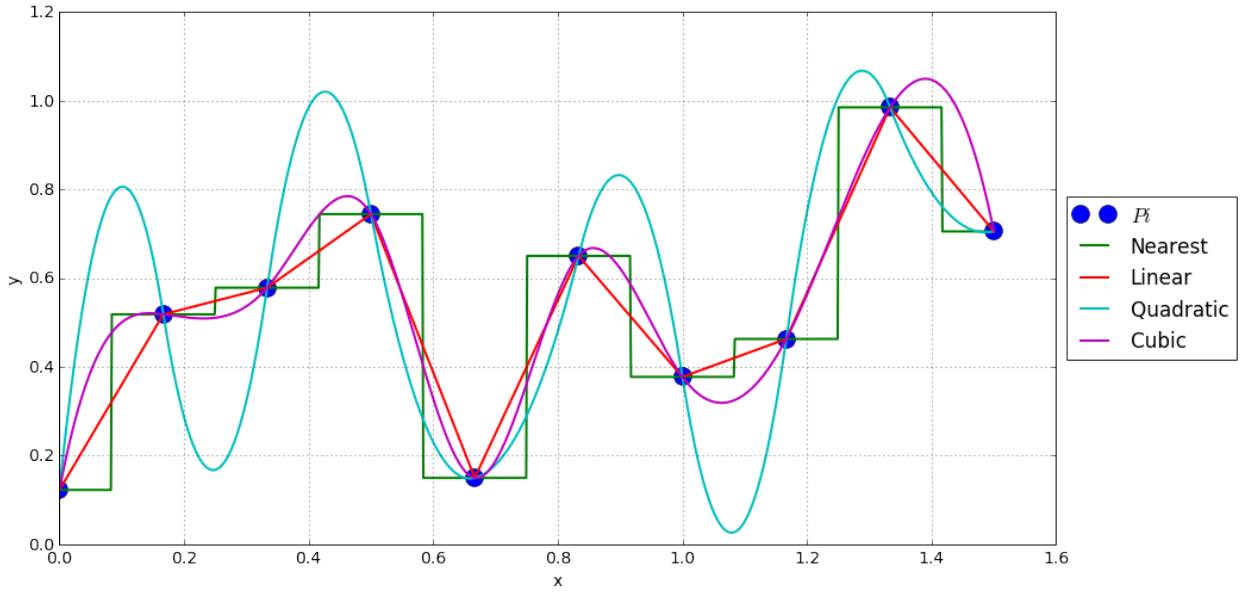
- $y(x)$ stays in the limits of y_i
- Continuous

3.1.4.2 Cons

- Discontinuous first derivative.

3.1.5 Spline interpolation

```
from scipy import interpolate
x = np.linspace(xmin, xmax, 1000)
interp2 = interpolate.interp1d(xi, yi, kind = "quadratic")
interp3 = interpolate.interp1d(xi, yi, kind = "cubic")
y_quad = interp2(x)
y_cubic = interp3(x)

plt.plot(xi,yi, 'o', label = "$Pi$")
plt.plot(x, y_nearest, "-", label = "Nearest")
plt.plot(x, y_linear,  "-", label = "Linear")
plt.plot(x, y_quad,    "-", label = "Quadratic")
plt.plot(x, y_cubic,   "-", label = "Cubic")
plt.grid()
plt.xlabel("x")
plt.ylabel("y")
plt.legend(loc='center left', bbox_to_anchor=(1, 0.5))
plt.show()
```



3.1.5.1 Pros

- Smoother
- Cubic generally more reliable that quadratic

3.1.5.2 Cons

- Less predictable values between points.

```
from IPython.core.display import HTML
def css_styling():
    styles = open('styles/custom.css', 'r').read()
    return HTML(styles)
css_styling()
```

3.2 2D Interpolation (and above)

3.2.1 Scope

- Finite number N of data points are available: $P_i = (x_i, y_i)$ and associated values z_i , $i \in \{0, \ldots, N\}$
- ND interpolation differs from 1D interpolation because the notion of neighbourhood is less obvious.

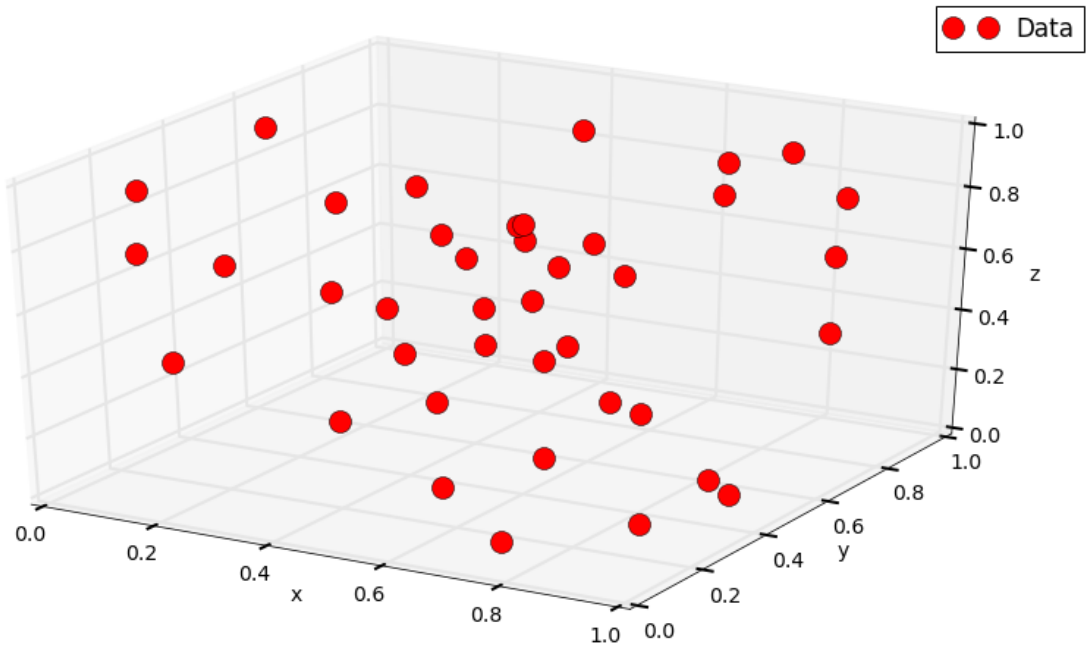
https://en.wikipedia.org/wiki/Interpolation

```
# Setup
%matplotlib inline
import numpy as np
import matplotlib.pyplot as plt
import matplotlib
params = {'font.size'      : 14,
          'figure.figsize':(15.0, 8.0),
```

```
          'lines.linewidth': 2.,
          'lines.markersize': 15,}
matplotlib.rcParams.update(params)
```

3.2.2 Let's do it with Python

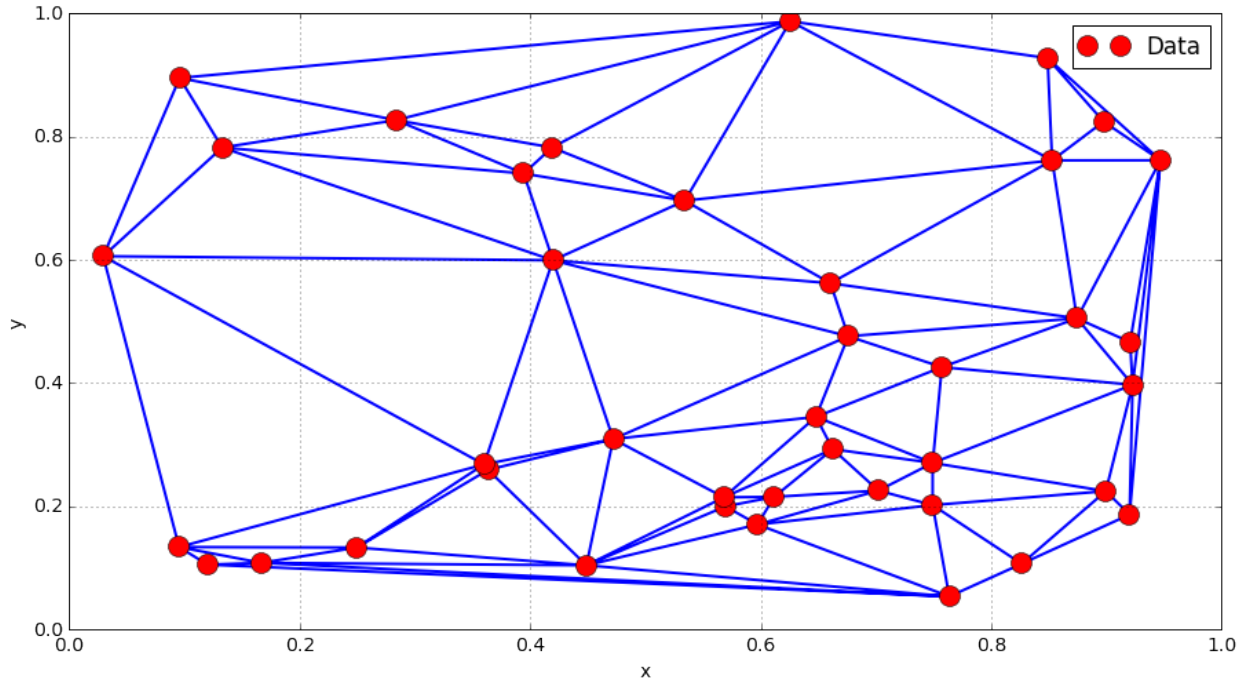
```
Ni = 40
Pi = np.random.rand(Ni, 2)
Xi, Yi = Pi[:,0], Pi[:,1]
Zi = np.random.rand(Ni)


import matplotlib as mpl
from mpl_toolkits.mplot3d import Axes3D
import numpy as np
import matplotlib.pyplot as plt
fig = plt.figure()
ax = fig.gca(projection='3d')

ax.plot(Xi, Yi, Zi, "or", label='Data')
ax.legend()
ax.set_xlabel('x')
ax.set_ylabel('y')
ax.set_zlabel('z')
plt.show()
```



3.2.3 Neighbours and connectivity: Delaunay mesh

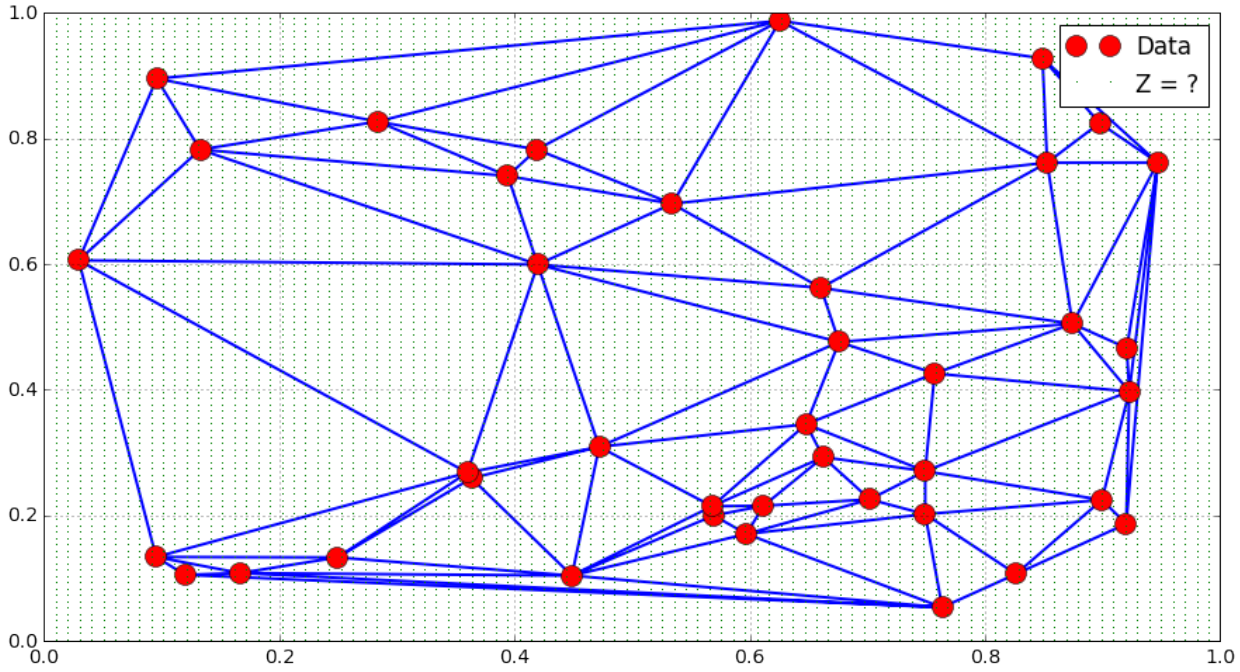
Triangular mesh over a convex domain

```
from scipy.spatial import Delaunay
Pi = np.array([Xi, Yi]).transpose()
tri = Delaunay(Pi)
plt.triplot(Xi, Yi , tri.simplices.copy())
plt.plot(Xi, Yi, "or", label = "Data")
plt.grid()
plt.legend()
plt.xlabel("x")
plt.ylabel("y")
plt.show()
```



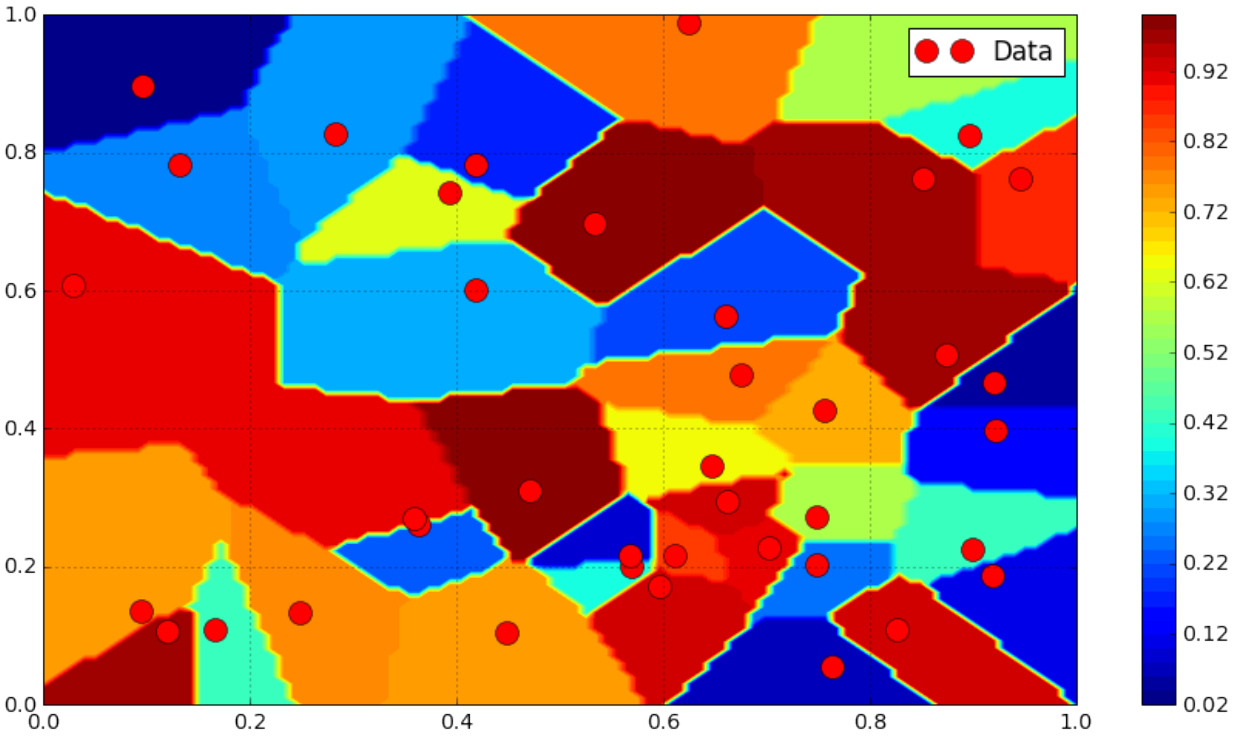
Interpolation

```
N = 100
x = np.linspace(0., 1., N)
y = np.linspace(0., 1., N)
X, Y = np.meshgrid(x, y)
P = np.array([X.flatten(), Y.flatten() ]).transpose()
plt.plot(Xi, Yi, "or", label = "Data")
plt.triplot(Xi, Yi , tri.simplices.copy())
plt.plot(X.flatten(), Y.flatten(), "g,", label = "Z = ?")
plt.legend()
plt.grid()
plt.show()
```



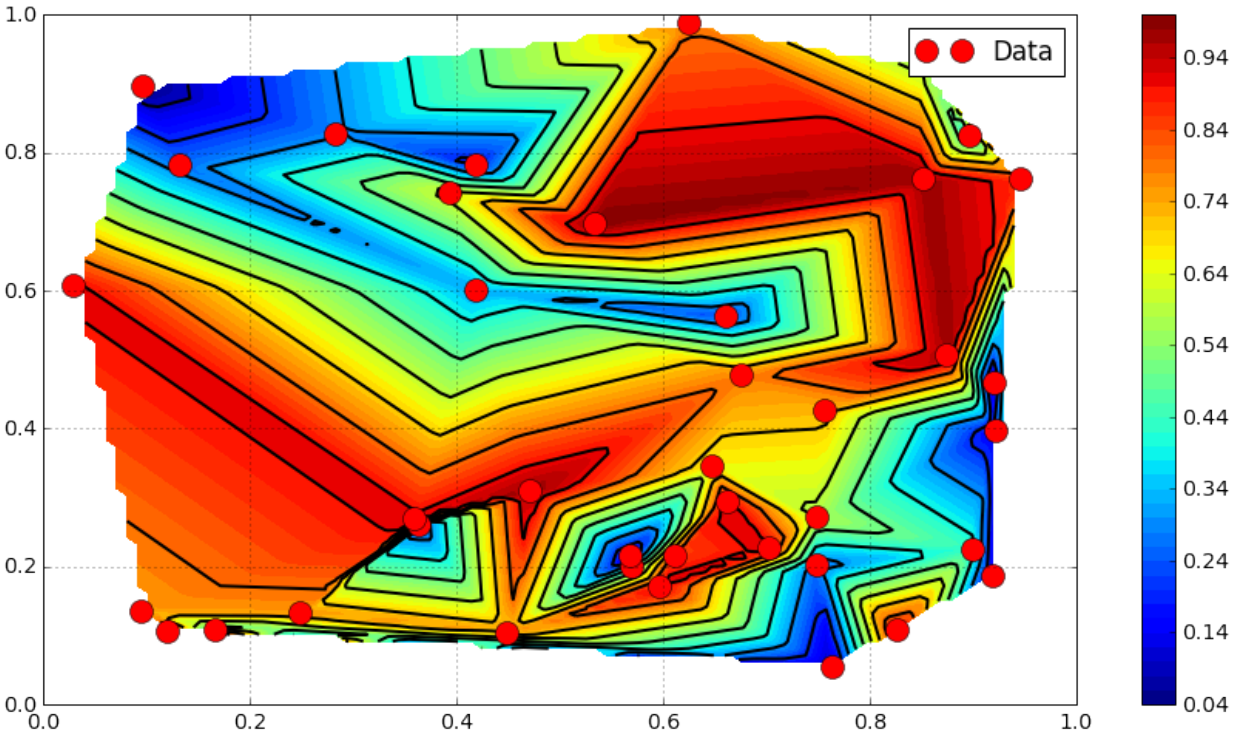
3.2.4 Nearest interpolation

```
from scipy.interpolate import griddata
Z_nearest = griddata(Pi, Zi, P, method = "nearest").reshape([N, N])
plt.contourf(X, Y, Z_nearest, 50)
plt.plot(Xi, Yi, "or", label = "Data")
plt.colorbar()
plt.legend()
plt.grid()
plt.show()
```



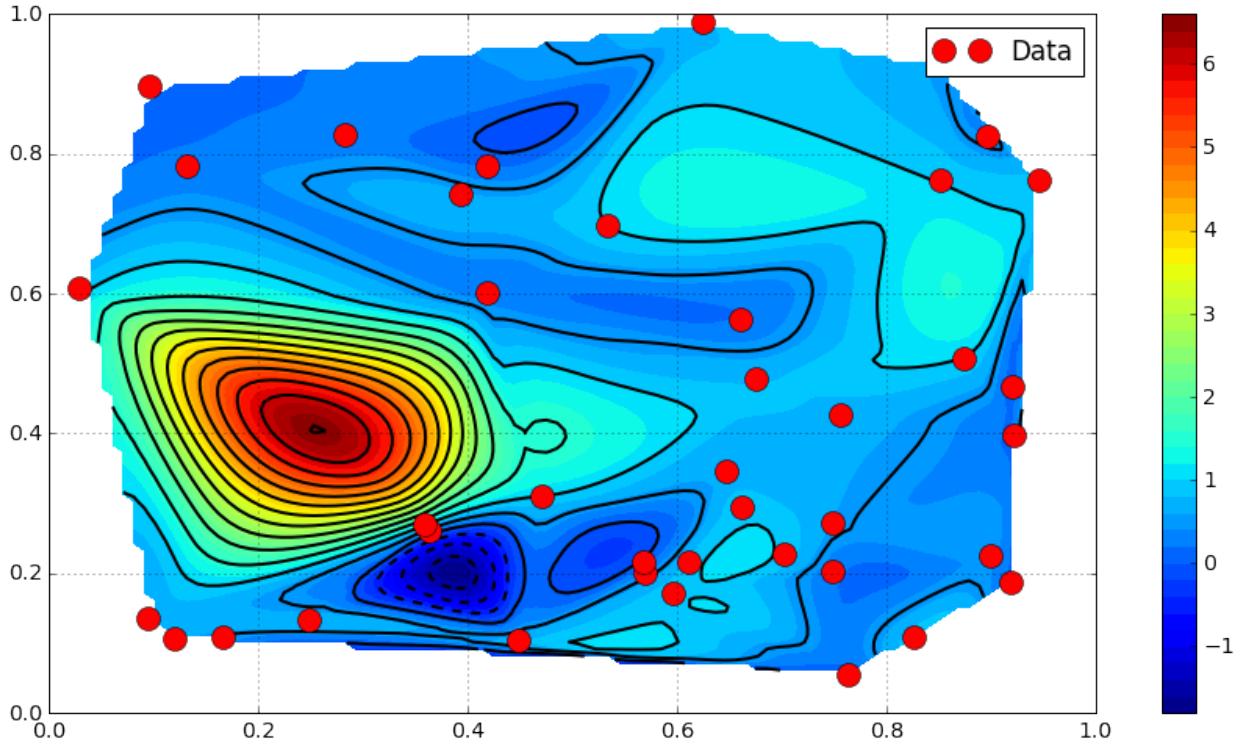
3.2.5 Linear interpolation

```
from scipy.interpolate import griddata
Z_linear = griddata(Pi, Zi, P, method = "linear").reshape([N, N])
plt.contourf(X, Y, Z_linear, 50, cmap = mpl.cm.jet)
plt.colorbar()
plt.contour(X, Y, Z_linear, 10, colors = "k")
#plt.triplot(Xi, Yi , tri.simplices.copy(), color = "k")
plt.plot(Xi, Yi, "or", label = "Data")
plt.legend()
plt.grid()
plt.show()
```



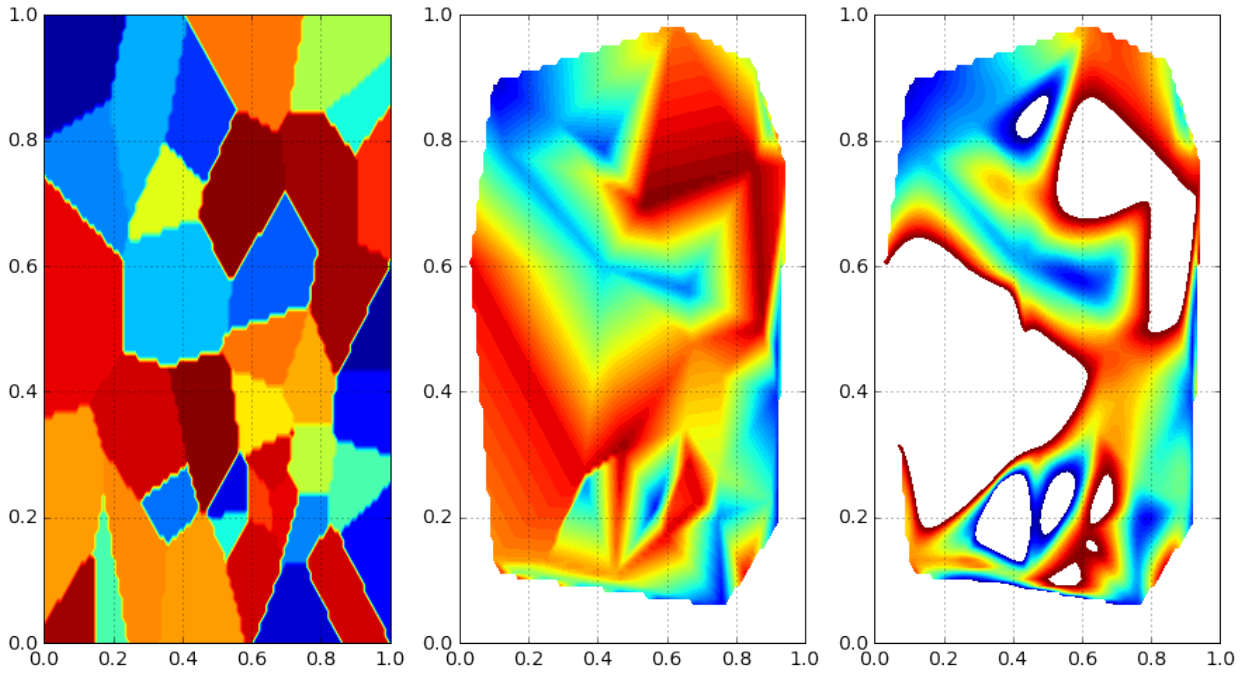
3.2.6 Higher order interpolation

```
from scipy.interpolate import griddata
Z_cubic = griddata(Pi, Zi, P, method = "cubic").reshape([N, N])
plt.contourf(X, Y, Z_cubic, 50, cmap = mpl.cm.jet)
plt.colorbar()
plt.contour(X, Y, Z_cubic, 20, colors = "k")
#plt.triplot(Xi, Yi , tri.simplices.copy(), color = "k")
plt.plot(Xi, Yi, "or", label = "Data")
plt.legend()
plt.grid()
plt.show()
```



3.2.7 Comparison / Discussion

```
levels = np.linspace(0., 1., 50)
fig = plt.figure()
ax =  fig.add_subplot(1, 3, 1)
plt.contourf(X, Y, Z_nearest, levels)
plt.grid()
ax =  fig.add_subplot(1, 3, 2)
plt.contourf(X, Y, Z_linear, levels)
plt.grid()
ax =  fig.add_subplot(1, 3, 3)
plt.contourf(X, Y, Z_cubic, levels)
plt.grid()
```



3.3 Tutorials

3.3.1 1D interpolation

- Define a mathematical function of your choice.
- Sample it at a chose rate.
- Try all known interpolation methods on the sampled points.
- Estimate errors

3.3.2 2D interpolation

Proceed as with 1D interpolation with a mapping.

CHAPTER 4

Traitement de signal

Cette partie du cours aborde les notions basiques sur:

- Ce qu'est un signal.
- Les conséquences de son enregistrement dans un format numérique.
- L'analyse fréquentielle de son contenu.

Les points abordés ici sont détaillés plus finement dans le cours: `Traitement_Signal_slides.pdf`

4.1 Signal

Un signal undimensionnel peut être vu comme une fonction mathématique du type:

$$x : \ t \mapsto x(t)$$

Le signal sera périodique si il existe une période T telle que:

$$\forall t, \ x(t+T) = x(t)$$

On définit alors sa fréquence $f = 1/T$. Dans le cas général, un signal quelconque pourra toujours être vu comme une somme de plusieurs signaux (ou composantes) périodiques.

4.2 Observation du signal

Pour des raisons pratiques, il est impossible d'observer un signal pour toutes les valeur de t. On observe donc uniquement le signal entre t_0 et t_1. On définit alors la durée d'observation $D = t_1 - t_0$.

4.3 Echantillonnage

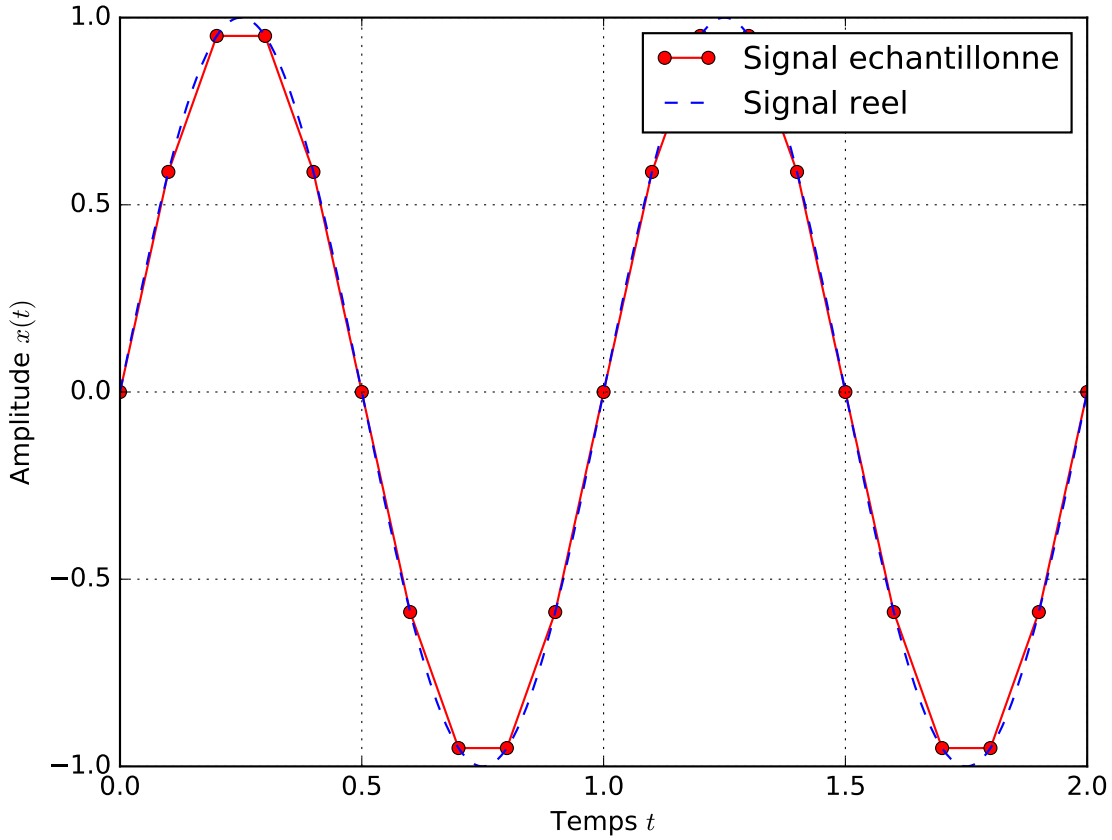
4.3.1 Principe

Pour enregistre un signal dans un format numérique, on mesure ses valeurs sur une grille de points $[t_i]$ avec $i \in [0, N-1]$, c'est l'échantillonnage. On enregistre donc N valeurs. Une grande quantité d'information est donc perdue par ce procédé. On définit aussi la fréquence d'échantillonnage:

$$f_e = \frac{N-1}{D}$$

Dans l'exemple ci-dessous, les pastilles rouges représentent les points pour lesquelles les valeurs du signal réel sont enregistrées. On remarque les pastilles rouges décrivent bien la forme du signal réel, l'échantillonnage est donc réussi.

```
import numpy as np
import matplotlib.pyplot as plt

# Signal
T = 1.
def signal(t): return np.sin(2. * np.pi * t / T)
# Echantillonnage
D = 2. # Duree d'observation
fe = 10. # Frequence d'echantillonnage
N = int(D * fe) + 1 # Nombre de points enregistres
te = np.linspace(0., (N-1)/fe, N) # Grille d'echantillonnage
tp = np.linspace(0., D, 1000) # Grille plus fine pour tracer l'allure du signal␣
↪parfait
# Trace du signal
plt.plot(te, signal(te), 'or-', label = u"Signal echantillonne")
plt.plot(tp, signal(tp), 'b--', label = u"Signal reel")
plt.grid()
plt.xlabel("Temps $t$")
plt.ylabel("Amplitude $x(t)$")
plt.legend()
plt.show()
```



4.3.2 Théorème de Shannon-Nyquist

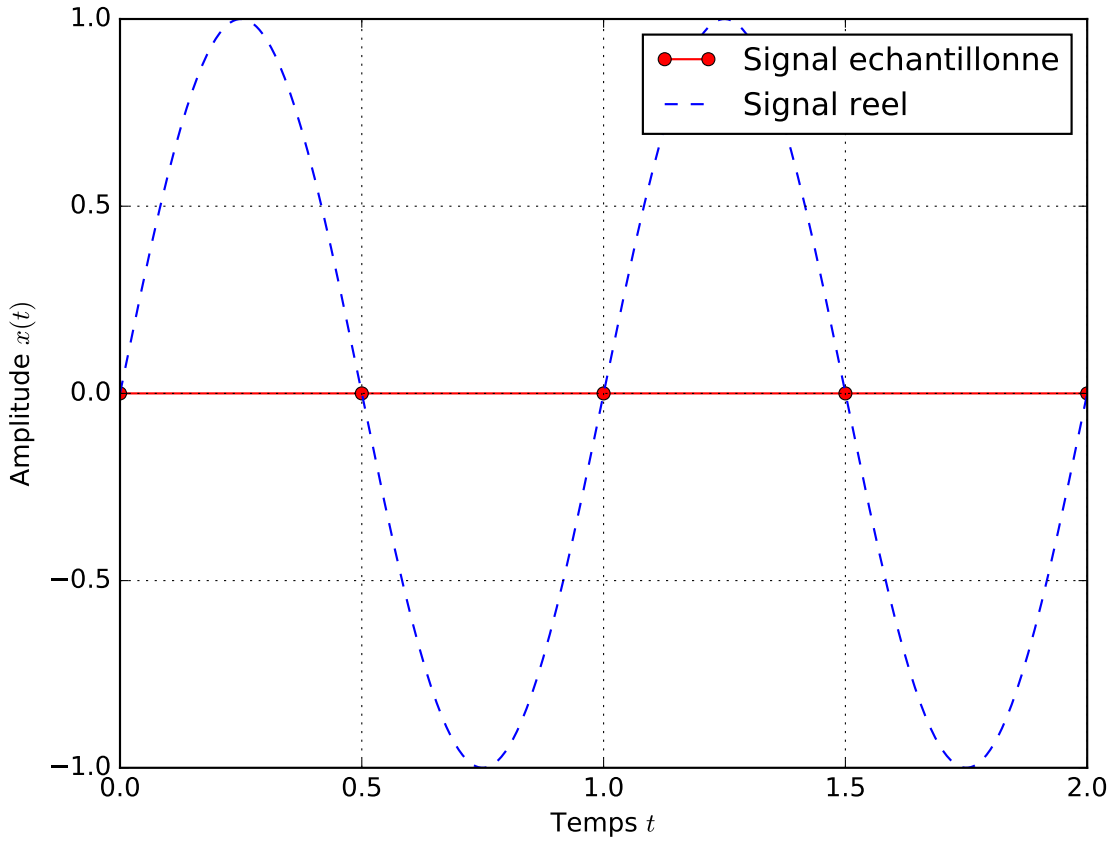
Le théorème de Shannon Nyquist indique que la fréquence f du signal (ou celles de ses composantes) doit vérifier:

$$f < \frac{f_e}{2}$$

Le cas extrême où $f = f_e/2$ est représenté ci-dessous.

```
import numpy as np
import matplotlib.pyplot as plt

# Signal
T = 1.
def signal(t): return np.sin(2. * np.pi * t / T)
# Echantillonnage
D = 2. # Duree d'observation
fe = 2. # Frequence d'echantillonnage
N = int(D * fe) + 1 # Nombre de points enregistres
te = np.linspace(0., D, N) # Grille d'echantillonnage
tp = np.linspace(0., D, 1000) # Grille plus fine pour tracer l'allure du signal␣
↪parfait
# Trace du signal
plt.plot(te, signal(te), 'or-', label = u"Signal echantillonne")
plt.plot(tp, signal(tp), 'b--', label = u"Signal reel")
```

```
plt.grid()
plt.xlabel("Temps $t$")
plt.ylabel("Amplitude $x(t)$")
plt.legend()
plt.show()
```

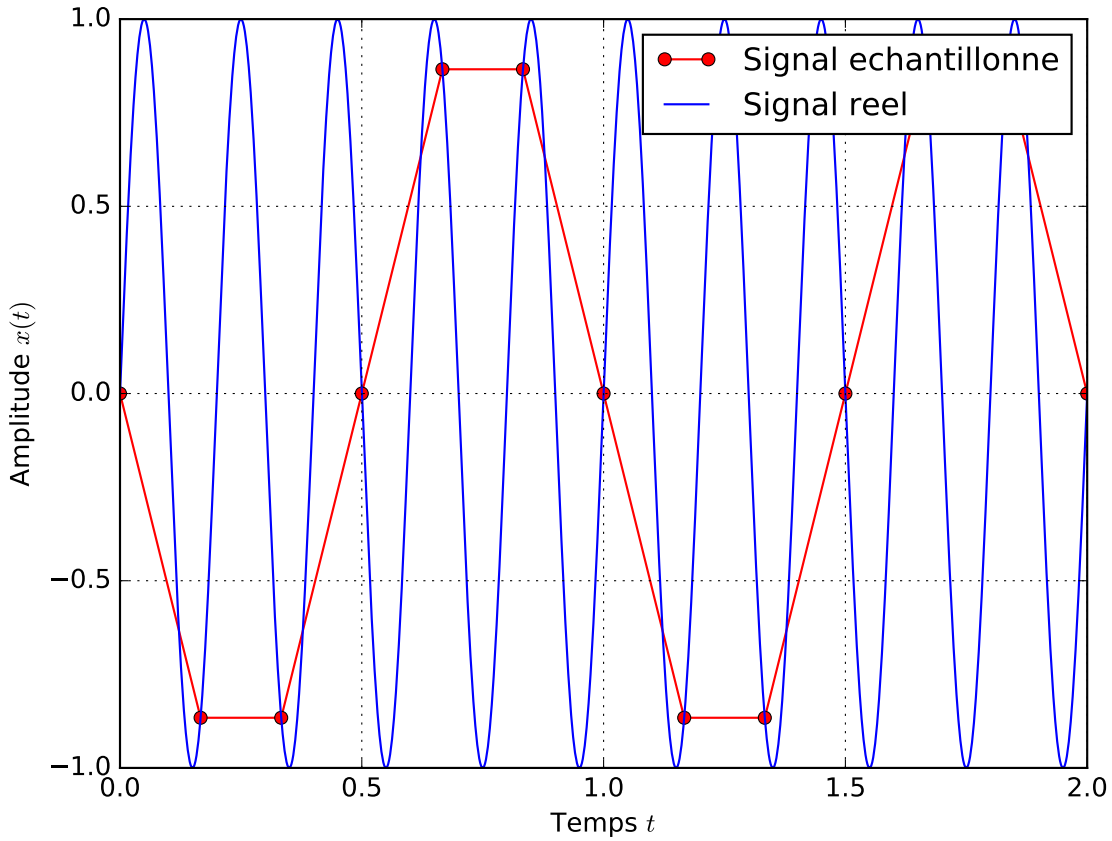


4.3.3 Repliement de spectre

La figure ci-dessus laisse penser que si la fréquence du signal ne respecte pas le théorème de Shannon-Nyquist, alors le signal est perdu lors de l'échantillonnage. En réalité, le signal apparait comme ayant une fréquence différente comme le montre la figure ci-dessous.

```
import numpy as np
import matplotlib.pyplot as plt

# Signal
T = .2
def signal(t): return np.sin(2. * np.pi * t / T)
# Echantillonnage
D = 2. # Duree d'observation
fe = 6. # Frequence d'echantillonnage
N = int(D * fe) + 1 # Nombre de points enregistres
te = np.linspace(0., (N-1)/fe, N) # Grille d'echantillonnage
```

```
tp = np.linspace(0., D, 1000) # Grille plus fine pour tracer l'allure du signal␣
↪parfait
# Trace du signal
plt.plot(te, signal(te), 'or-', label = u"Signal echantillonne")
plt.plot(tp, signal(tp), 'b-', label = u"Signal reel")
plt.grid()
plt.xlabel("Temps $t$")
plt.ylabel("Amplitude $x(t)$")
plt.legend()
plt.show()
```



Pour échantillonner un signal, il est donc essentiel de retirer préalablement les composantes de fréquence $f \geq f_e/2$ à l'aide d'un filtre anti repliement.

4.4 Analyse Spectrale

4.4.1 Principe

L' analyse spectrale d'un signal consiste à construire son spectre, c'est-à-dire sa décomposition sous forme d'une somme fonctions périodiques. Plusieurs outils existent selon le type de signal étudié. Dans la pratique, nous allons travaillons toujours avec des signaux apériodiques échantillonnés, l'outil de base de base pour construire le spectre est la Transformé de Fourier Discrète (DFT) ou son implémentation rapide, la FFT . En python, le moyen le plus simple

pour accéder aux algorithmes de FFT est scipy . L'algorithem FFT impose que le nombre d'échantillon N soit une puissance de 2.

```
>>> N = len(x)
>>> fe = 1. / (t[1] - t[0])
```

Dans la pratique la FFT d'un signal x se présente de la manière suivante:

```
>>> from scipy import fftpack
>>>  X = fftpack.fft(x)
```

Le vecteur X est composé de N coefficients complexes. La première moitié des coefficients du vecteur X correspondent aux fréquences positives et la seconde aux fréquences négatives.

```
>>> Xpos = X[0:N/2] # Coefficients correspondant aux frequences positives
>>> Xneg = X[N/2:N] # Coefficients correspondant aux frequences negatives
```

Dans notre cas, le signal x étant réel, les coefficients correspondant aux fréquences négatives sont les conjugués des coefficients correspondant aux fréquences positives, ils n'apportent donc pas d'information utile.

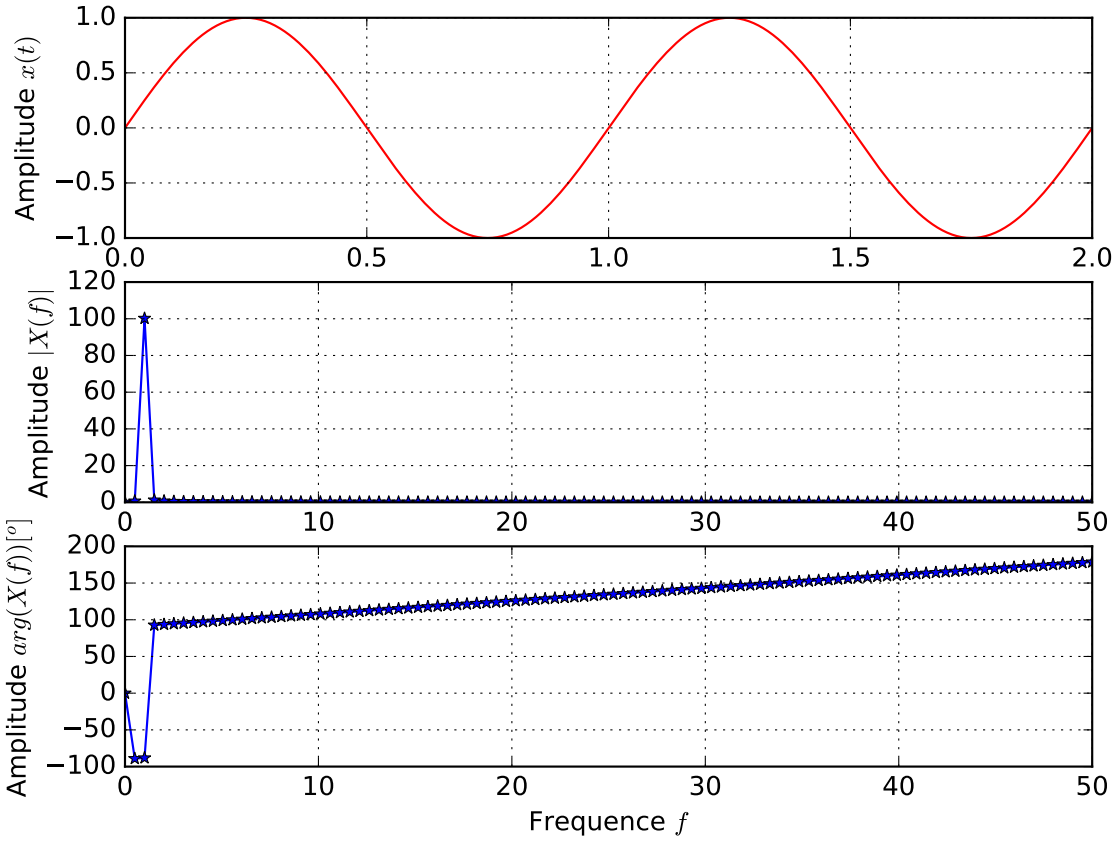
Le vecteur fréquence f correspondant au vecteur X comporte N coefficients se répartissant entre $-f_e/2$ et $f_e/2$. Dans la pratique, il n'est pas intéressant de tracer les fréquences négatives, nous pouvons donc tracer un signal et son spectre de la manière suivante:

```
>>> f = np.linspace(0., fe/2., N/2)
```

Mise en pratique:

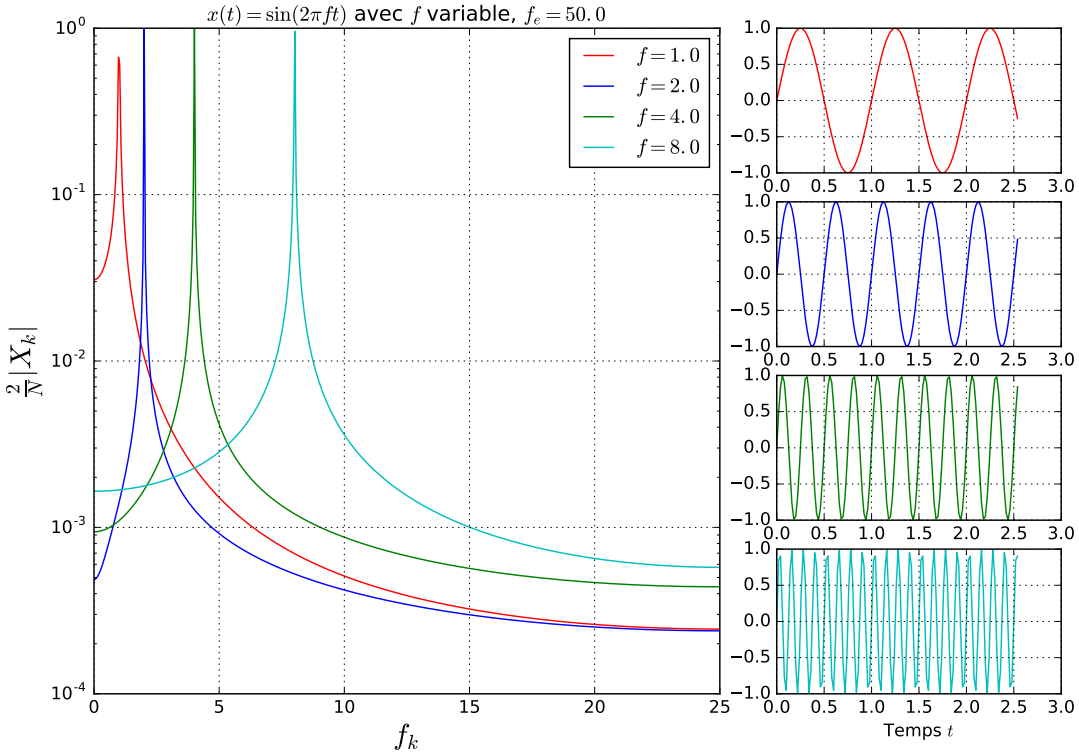
```
import numpy as np
import matplotlib.pyplot as plt
from scipy import fftpack

# Signal
T = 1.
def signal(t): return np.sin(2. * np.pi * t / T)
# Echantillonnage
D = 2. # Duree d'observation
fe = 100. # Frequence d'echantillonnage
N = int(D * fe) + 1 # Nombre de points enregistres
t = np.linspace(0., (N-1)/fe, N) # Grille d'echantillonnage
x = signal(t)
# FFT
X = fftpack.fft(x)
fpos = np.linspace(0., fe/2., N/2)
Xpos = X[0:N/2]
# Trace du signal et de son spectre
fig = plt.figure(0)
ax = fig.add_subplot(3,1,1)
plt.plot(t, x, 'r-')
plt.grid()
plt.xlabel("Temps $t$")
plt.ylabel("Amplitude $x(t)$")
ax = fig.add_subplot(3,1,2)
plt.plot(fpos, abs(Xpos), 'b*-')
plt.grid()
plt.ylabel("Amplitude $|X(f)|$")
ax = fig.add_subplot(3,1,3)
plt.plot(fpos, np.degrees(np.angle(Xpos)), 'b*-')
```

```
plt.grid()
plt.xlabel("Frequence $f$")
plt.ylabel("Amplitude $arg(X(f)) [^o]$")
plt.show()
```



4.4.2 Interprétation

- Effet de la fréquence:



4.5 Travaux dirigés

Ce sujet est une introduction aux questions abordées dans ce cours. On vous demande d'écrire un (ou plusieurs) scripts qui pour effectuer les tâches suivantes:

1. Signal sinusoidal

1. Générer un signal sinusoidal de la forme $x(t) = a\sin(2\pi ft + \phi)$.
2. Construire une grille d'échantillonnage t pour laquelle on peut contrôler la fréquence d'échantillonnage f_e et la durée d'observation D.
3. Tracer le signal échantillonné.
4. Que se passe-t-il quand on augmente la fréquence du signal f en laissant f_e constante.
5. Calculer la transfromée de Fourier par FFT X des coefficients x.
6. Calculer les fréquences positives.
7. Tracer le spectre du signal.
8. Expliquer l'influence de a , f et ϕ sur le spectre.

2. Autres signaux

1. Réutilisez le code produit dans les questions précédentes et appliquez le à un signal carré.
2. Même démarche pour un signal constant.
3. Même démarche pour une gaussienne.

4.6 Travaux Pratiques

1. Signaux

On a en enregistré deux signaux expérimentaux au moyen d'un accéléromètre:

- Un signal enregistré par un accéléromètre sur une cloche: `cloche.txt`.
- Un signal enregistré sur une poutre que l'on met en vibration au moyen d'un marteau de choc: `poutre_Al_flexion.txt`.

2. Etude de la poutre

La poutre est constituée d'un alliage d'aluminium. Elle est de forme parallélépipédique de longueur $l = 600\ mm$, de hauteur de $h = 15\ mm$ et de largeur de $b = 30\ mm$. La masse volumique est mesurée préalablement est vaut $\rho = 2700\ kg/m^3$. Elle est sollicitée de manière à vibrer en flexion. D'un point de vue théorique, une poutre sollicité en flexion va présenter plusieurs modes propres correspondant chacun à une fréquence propre f_n vérifiant:

$$f_n = \frac{1}{2\pi}\frac{C_n^2}{l^2}\sqrt{\frac{E}{\rho}}\sqrt{\frac{I}{S}}$$

Avec:

- I: le moment quadratique de la section qui vaut: $bh^3/12$.
- S: l'aire de la section de la poutre qui vaut bh .
- C_n: un coefficient qui dépent du numéro n: du mode considéré.

On donne:

n	C_n^2
1	22.37
2	61.67
>2	$((2n+1)\pi/2)^2$

Travail demandé: écrire un script Python qui effectue les tâches suivantes:

1. Tracer le signal de l'accélération en fonction du temps.
2. Calculer le spectre de l'accélération par FFT.
3. Tracer le module du spectre $|X|$:.
4. Identifier automatiquement les modes propres de la poutre.
5. Déterminer le module du Young E: de l'alliage utilisé.

3. Etude de la cloche.

La cloche est prévue pour sonner le *Ré*, implique de produire certaines fréquences particulières

Composante	Formule	Valeur
Hum	f_0	276.8 Hz
Fondamentale	$2f_0$	553.6 Hz
Tierce	$2.4f_0$	664.3 Hz
Quinte	$3f_0$	830.4 Hz
Octave	$4f_0$	1107 Hz

Travail demandé: écrire un script Python qui effectue les tâches suivantes. Pour ce faire vous pouvez grandement réutiliser les outils développés dans la partie précédente:

1. Tracer le signal de l'accélération en fonction du temps.
2. Calculer le spectre de l'accélération par FFT.
3. Tracer le module du spectre $|X|$.
4. Identifier automatiquement les différentes harmoniques présentes dans le signal.
5. Déterminer le niveau d'erreur sur chaque harmonique.

CHAPTER 5

Image processing

5.1 Images Numériques

Les images numériques sont des images décrites dans un format numérique. On peut les utiliser pour interpréter quantitativement certaines grandeurs. Cette partie dresse un rapide tableau des différentes approches basiques qui permettent d'effectuer ces tâches et donne des pistes pour aller plus loin sur chaque thème abordé.

Les points abordés ici sont détaillés plus finement dans le cours: `Traitement_Signal_slides.pdf`

5.1.1 Formation

Selon le dispositif qui l'a produite, le sens physique des informations contenues dans une image est différent. Voici quelques exemples d'images classées par type d'informations:

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

5.1.2 Structure

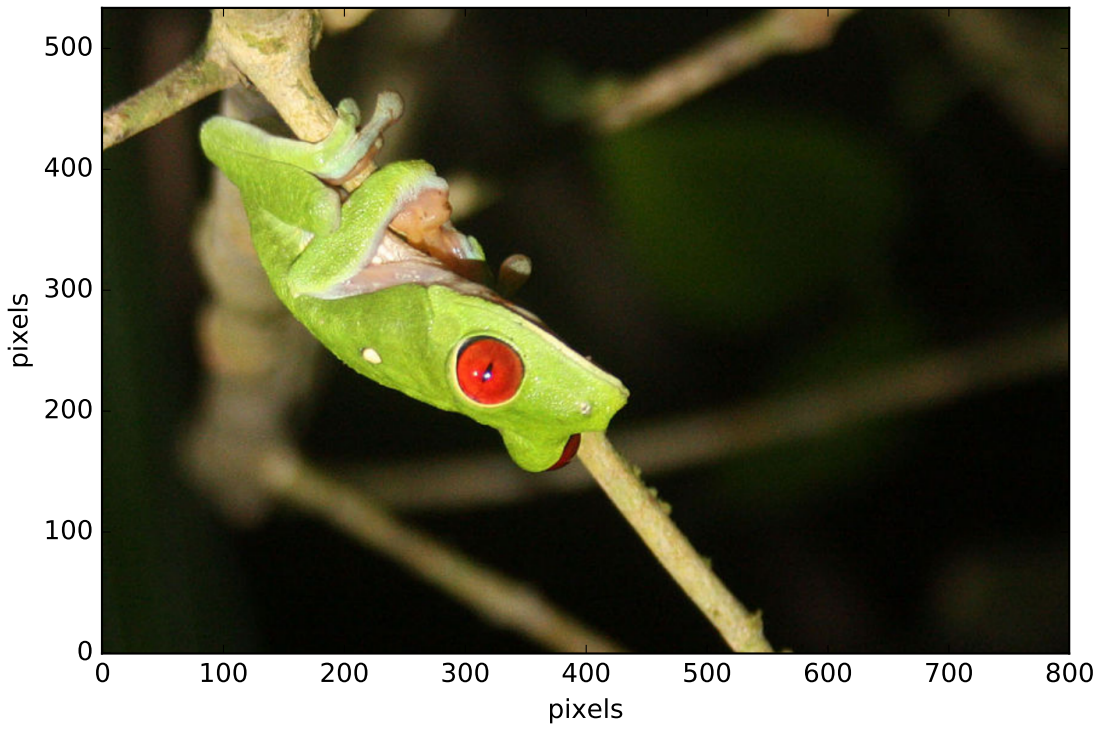
Deux grandes familles d'images numériques existent:

- 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 flottant dans le cas d'images scientifiques. Il est important de

noter que la couleur d'un pixel tel qu'il apparait 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 eventuellement 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 phénomème scientifique par un seul canal, si possible sous forme flottante. On parle alors d'image monochrome.

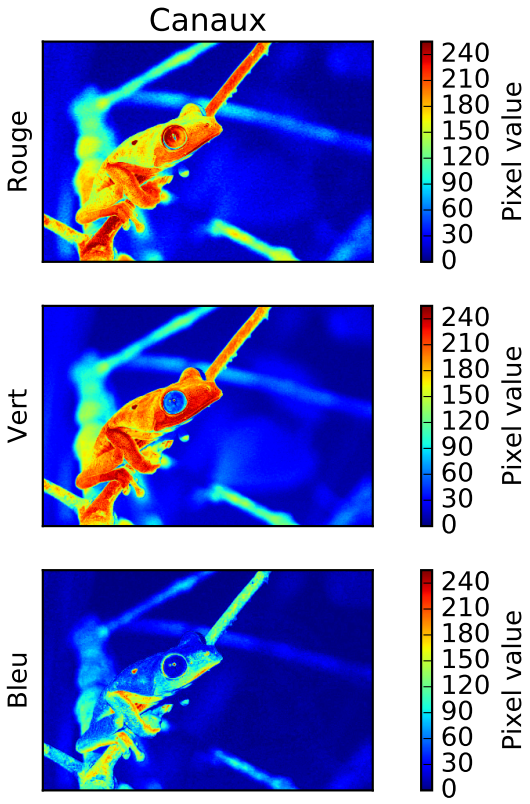
On prend un exemple de photographie: `grenouille.jpg`

```
from PIL import Image
import numpy as np
import matplotlib.pyplot as plt
im = Image.open(
  '../data/grenouille.jpg')
fig = plt.figure(0)
plt.clf()
plt.imshow(im, origin = "lower")
plt.xlabel("pixels")
plt.ylabel("pixels")
plt.show()
```



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

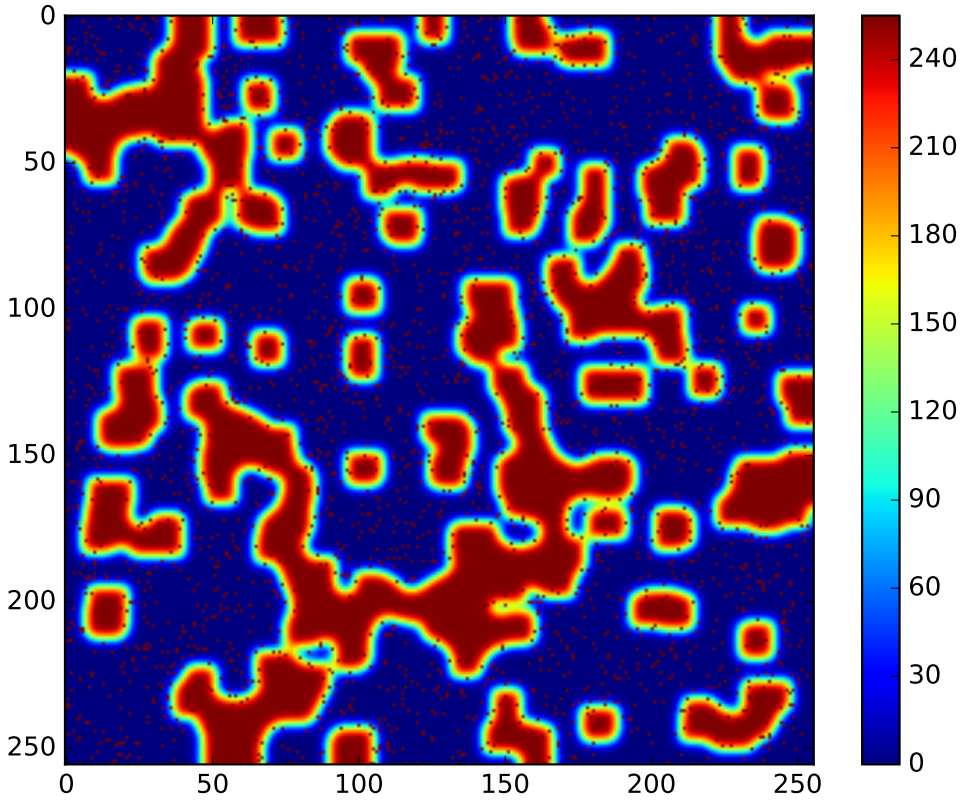
```
from PIL import Image
import numpy as np
from matplotlib import pyplot as plt
im = Image.open('../data/grenouille.jpg')
rouge, vert, bleu = im.split()
rouge = np.array(rouge)
vert  = np.array(vert)
bleu  = np.array(bleu)

fig = plt.figure(0) # On cree une figure
plt.clf()
ax1 =fig.add_subplot(3,1,1)
plt.imshow(rouge, origin = "upper")
plt.xticks([])
plt.yticks([])
plt.grid()
cbar = plt.colorbar()
cbar.set_label("Pixel value")
plt.ylabel("Rouge")
plt.title("Canaux")
ax2 =  fig.add_subplot(3,1,2)
plt.imshow(vert,  origin = "upper")
plt.xticks([])
plt.yticks([])
plt.grid()
cbar = plt.colorbar()
cbar.set_label("Pixel value")
plt.ylabel("Vert")
ax3 = fig.add_subplot(3,1,3)
plt.imshow(bleu,  origin = "upper")
plt.xticks([])
plt.yticks([])
plt.grid()
cbar = plt.colorbar()
cbar.set_label("Pixel value")
plt.ylabel("Bleu")
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`.

5.1.3 Operations

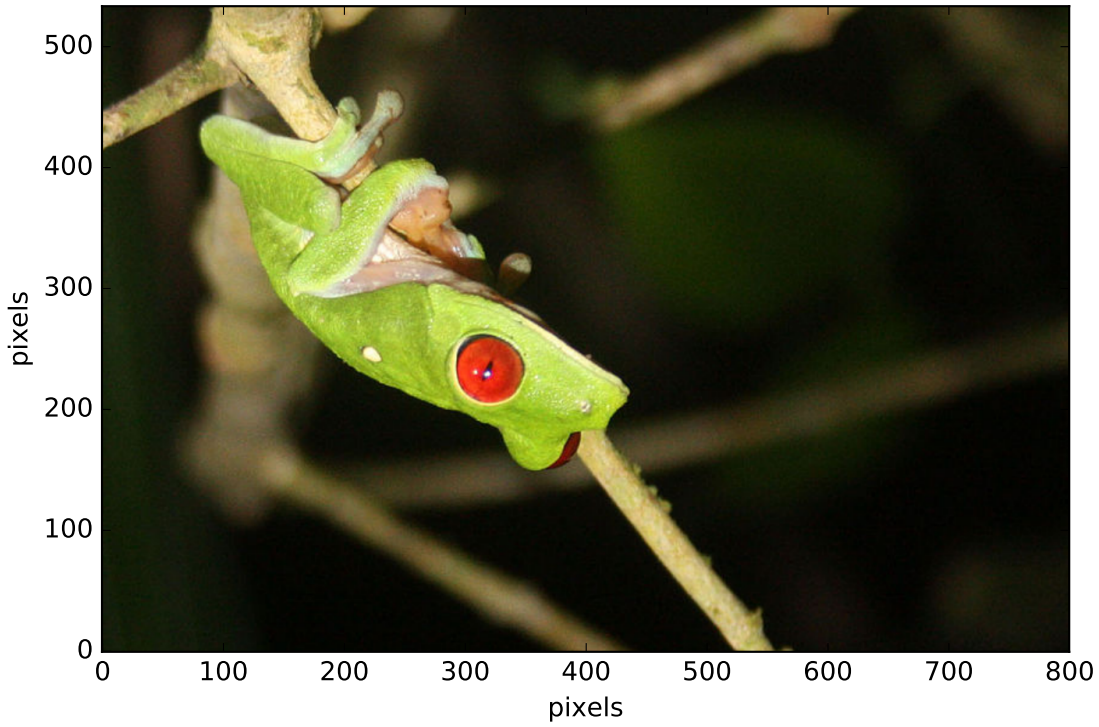
Dans cette partie, nous utiliserons aussi une image générée pour l'occasion:



Vous pouvez télécharger l'image ici: `image.jpg`

5.1.3.1 Lecture

```
from PIL import Image
import numpy as np
import matplotlib.pyplot as plt
im = Image.open(
  '../data/grenouille.jpg')
fig = plt.figure(0)
plt.clf()
plt.imshow(im, origin = "lower")
plt.xlabel("pixels")
plt.ylabel("pixels")
plt.show()
```



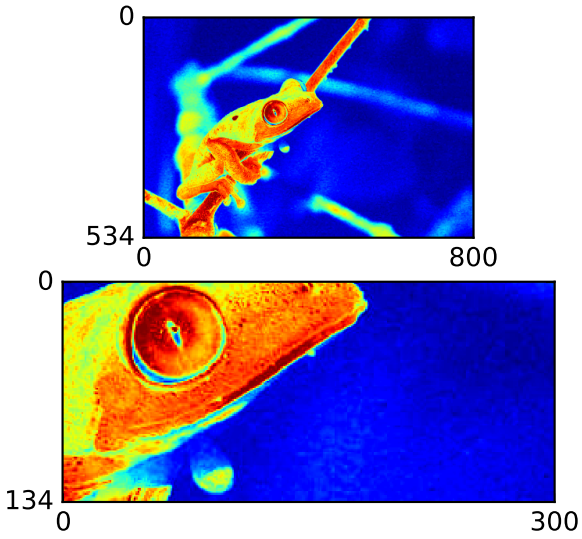
5.1.3.2 Sauvegarde

```
from PIL import Image
import numpy as np
from matplotlib import pyplot as plt
from matplotlib import cm
im = Image.open('../data/grenouille.jpg')
rouge, vert, bleu = im.split()
z = np.array(rouge)
z = np.uint8(cm.copper(z)*255)
im2 = Image.fromarray(z)
im2.save("grenouille_saved.jpg")
```

5.1.3.3 Rognage

```
from PIL import Image
import numpy as np
from matplotlib import pyplot as plt
from matplotlib import cm
im = Image.open('../data/grenouille.jpg')
rouge, vert, bleu = im.split()
z = np.array(rouge)
```

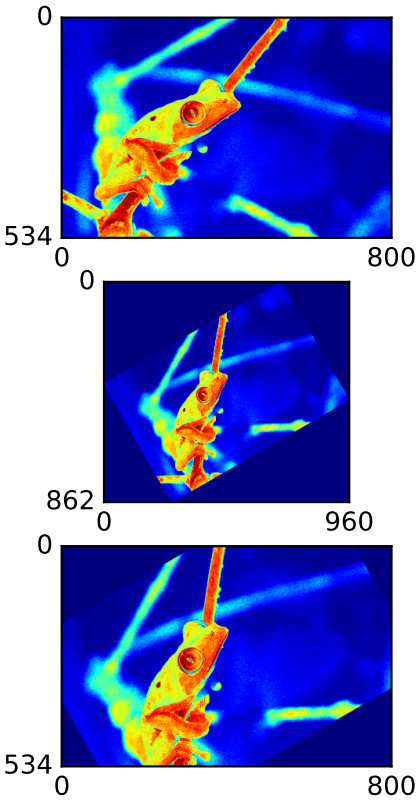
```
ny, nx = z.shape
cx, cy = 200, 250
zc = z[cx:-cx, cy:-cy]
nyc, nxc = zc.shape

fig = plt.figure(0) # On cree une figure
plt.clf()
ax1 =fig.add_subplot(3,1,1)
plt.imshow(z, origin = "upper")
plt.xticks([0, nx])
plt.yticks([0, ny])
ax2 =  fig.add_subplot(3,1,2)
plt.imshow(zc,  origin = "upper", interpolation = "nearest")
plt.xticks([0, nxc])
plt.yticks([0, nyc])
plt.show()
```



5.1.3.4 Rotations

```
from PIL import Image
import numpy as np
from matplotlib import pyplot as plt
from matplotlib import cm
```

```
from scipy import ndimage
im = Image.open('../data/grenouille.jpg')
rouge, vert, bleu = im.split()
z = np.array(rouge)
zrr = ndimage.rotate(z, 30.)
zrn = ndimage.rotate(z, 30.,
  reshape = False)

ny, nx = z.shape
nyrr, nxrr = zrr.shape
nyrn, nxrn = zrn.shape

fig = plt.figure(0) # On cree une figure
plt.clf()
ax1 =fig.add_subplot(3,1,1)
plt.imshow(z, origin = "upper")
plt.xticks([0, nx])
plt.yticks([0, ny])
ax2 =  fig.add_subplot(3,1,2)
plt.imshow(zrr,  origin = "upper")
plt.xticks([0, nxrr])
plt.yticks([0, nyrr])
ax2 =  fig.add_subplot(3,1,3)
plt.imshow(zrn,  origin = "upper")
plt.xticks([0, nxrn])
plt.yticks([0, nyrn])
plt.show()
```

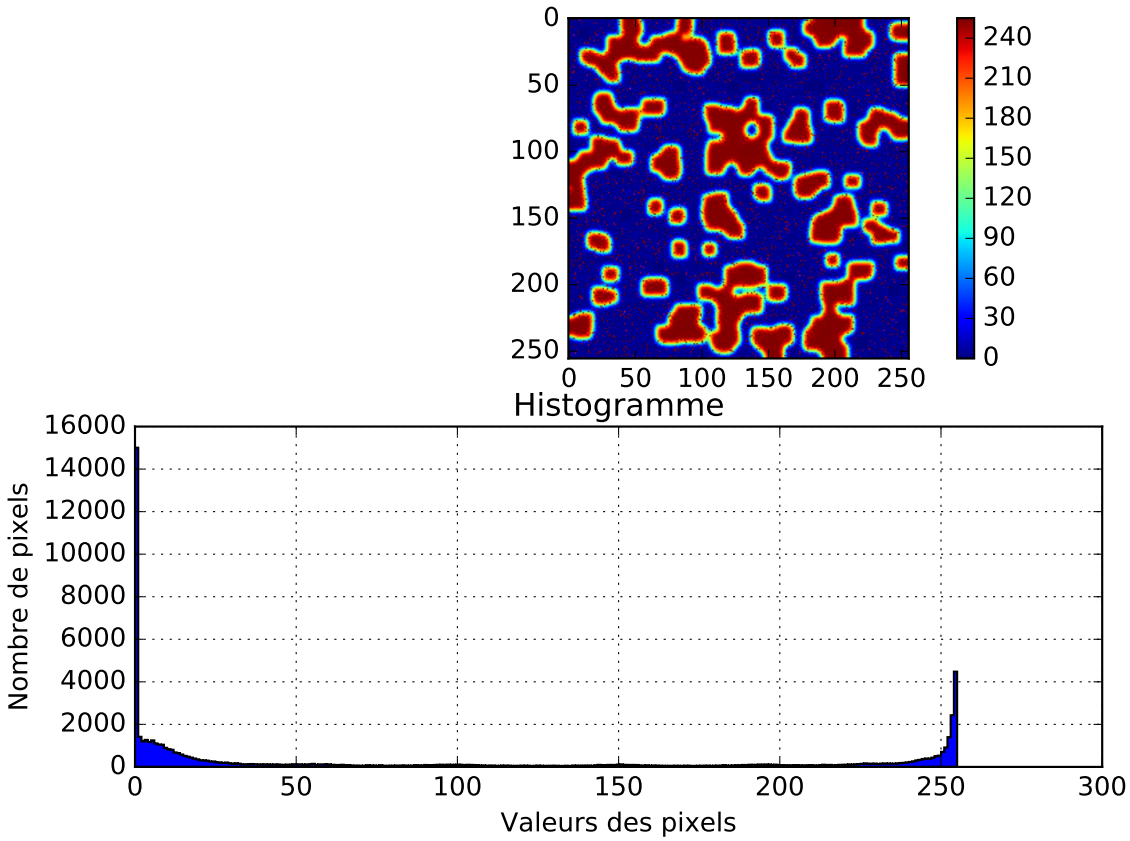


5.1.3.5 Histogramme

Un histogramme représente la répartition de la population de pixels en fonction de leur altitude. Une valeur haute dans l'histogramme indique donc qu'un grand nombre de pixels correspondent à l'altitude considérée.

```
from PIL import Image
import numpy as np
from matplotlib import pyplot as plt
im = Image.open('../Slides/figures/image.jpg')
channels = im.split()
z = np.array(channels[0])
N  = z.size
n_classes = int(N**.5)
fig = plt.figure()
plt.clf()
fig.add_subplot(2, 1, 1)
plt.imshow(z, origin = "upper")
plt.colorbar()
fig.add_subplot(2, 1, 2)
plt.title('Histogramme')
plt.ylabel('Nombre de pixels')
plt.xlabel('Valeurs des pixels')
plt.hist(z.flatten(), bins=n_classes, histtype = "stepfilled")
plt.grid()
```

```
plt.show()
```

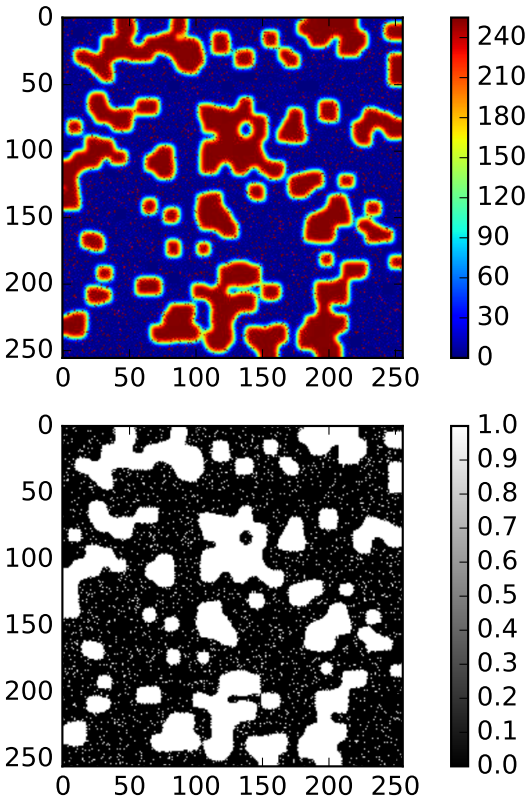


5.1.3.6 Seuillage

L'histogramme montre deux pics ($Z = 20$ et $Z = 230$) correspondant à deux populations de pixels. Le seuillage consiste à transformer une image monochrome en une **image binaire** en appliquant un test booléen à chaque pixel. Une image binaire, c'est-à-dire formée de 0 et de 1 ou de **Vrai** et **Faux** est ainsi créé. Dans le cas présent, on peut alors chercher séparer les deux populations en effectuant un seuillage $Z > 120$:

```
from PIL import Image
import numpy as np
from matplotlib import pyplot as plt
from matplotlib import cm
im = Image.open('../Slides/figures/image.jpg')
channels = im.split()
z = np.array(channels[0])
seuil = 150.
zs = z > seuil
fig = plt.figure()
plt.clf()
fig.add_subplot(2, 1, 1)
plt.imshow(z, origin = "upper")
plt.colorbar()
fig.add_subplot(2, 1, 2)
```

```
plt.imshow(zs, origin = "upper", cmap = cm.gray)
plt.colorbar()
plt.show()
```

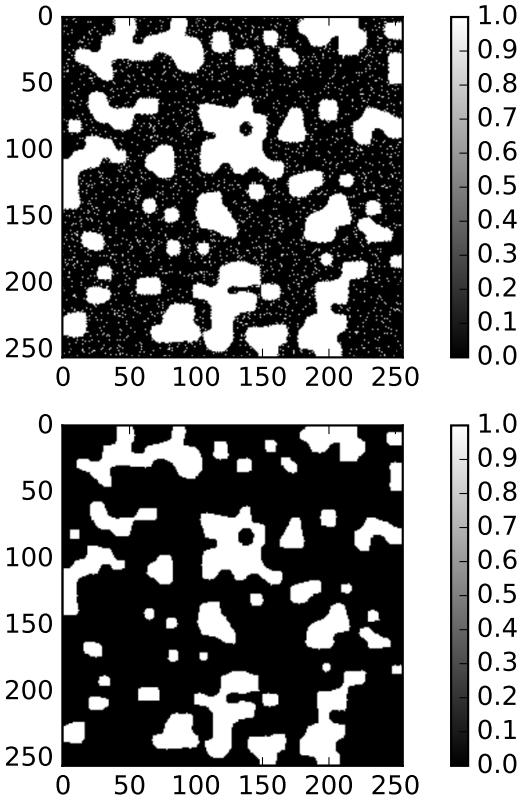


5.1.3.7 Erosion / Dilatation

On souhaite mesurer éliminer le bruit révélé par le seuillage effectué précédement. Pour ce faire, les outils issus de la morphologie mathématique tels que l'érosion et la dilatation sont particulièrement adaptés:

```
from PIL import Image
import numpy as np
from matplotlib import pyplot as plt
from matplotlib import cm
from scipy import ndimage
im = Image.open('../Slides/figures/image.jpg')
channels = im.split()
z = np.array(channels[0])
seuil = 150.
zs = z > seuil
zss = ndimage.morphology.binary_erosion(zs, structure=np.ones((3,3)))
fig = plt.figure()
plt.clf()
fig.add_subplot(2, 1, 1)
plt.imshow(zs, origin = "upper", cmap = cm.gray)
```

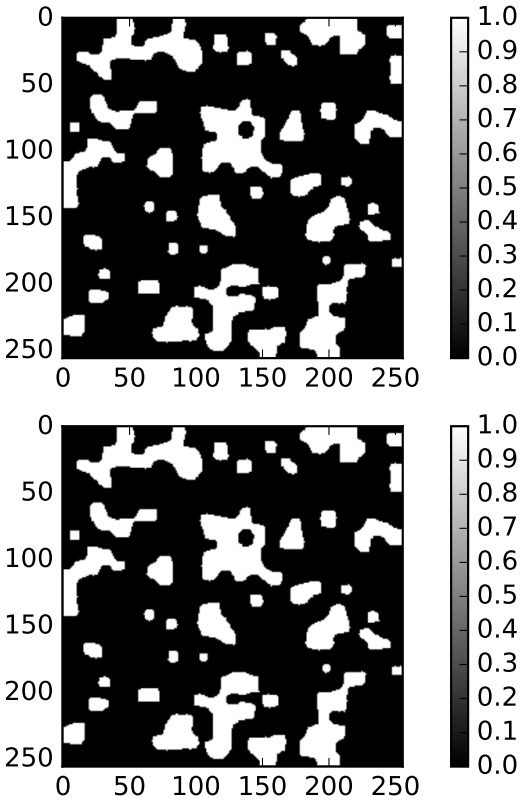
```
plt.colorbar()
fig.add_subplot(2, 1, 2)
plt.imshow(zss, origin = "upper", cmap = cm.gray)
plt.colorbar()
plt.show()
```



Pour restaurer la surface des zones partiellement érodées, on applique une dilatation:

```
from PIL import Image
import numpy as np
from matplotlib import pyplot as plt
from matplotlib import cm
from scipy import ndimage
im = Image.open('../Slides/figures/image.jpg')
channels = im.split()
z = np.array(channels[0])
seuil = 150.
zs = z > seuil
zse = ndimage.morphology.binary_erosion(zs, structure=np.ones((3,3)))
zsd = ndimage.morphology.binary_dilation(zse, structure=np.ones((3,3)))
fig = plt.figure()
plt.clf()
fig.add_subplot(2, 1, 1)
plt.imshow(zse, origin = "upper", cmap = cm.gray)
plt.colorbar()
```

```
fig.add_subplot(2, 1, 2)
plt.imshow(zse, origin = "upper", cmap = cm.gray)
plt.colorbar()
plt.show()
```

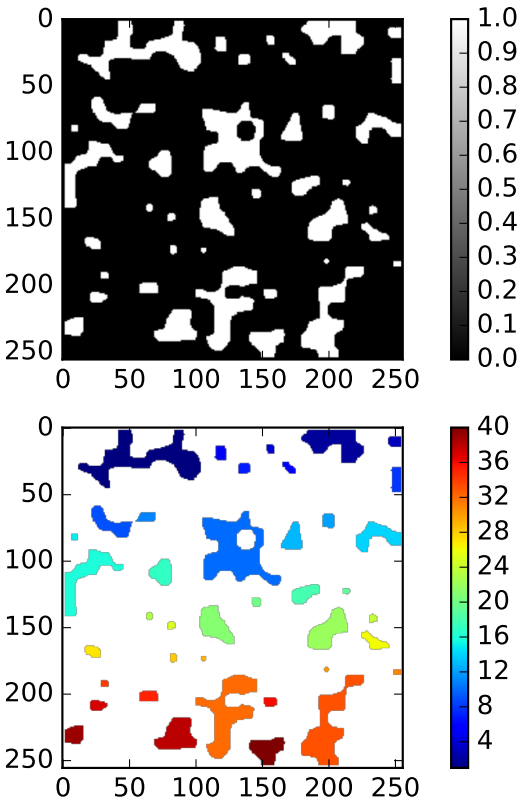


5.1.3.8 Comptage

Si on cherche maintenant a identifier individuellement les zones blanches mises en évidence lors du seuillage, 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:

```
from PIL import Image
import numpy as np
from matplotlib import pyplot as plt
from matplotlib import cm
from scipy import ndimage
im = Image.open('../Slides/figures/image.jpg')
channels = im.split()
z = np.array(channels[0])
seuil = 150.
zs = z > seuil
zse = ndimage.morphology.binary_erosion(zs, structure=np.ones((3,3)))
zsd = ndimage.morphology.binary_erosion(zse, structure=np.ones((3,3)))
zl, nombre = ndimage.measurements.label(zsd) # On compte les zones
```

```
zl = np.where(zl == 0, np.nan, zl)
fig = plt.figure()
plt.clf()
fig.add_subplot(2, 1, 1)
plt.imshow(zsd, origin = "upper", cmap = cm.gray)
plt.colorbar()
fig.add_subplot(2, 1, 2)
plt.imshow(zl, origin = "upper", cmap = cm.jet)
plt.colorbar()
plt.show()
```

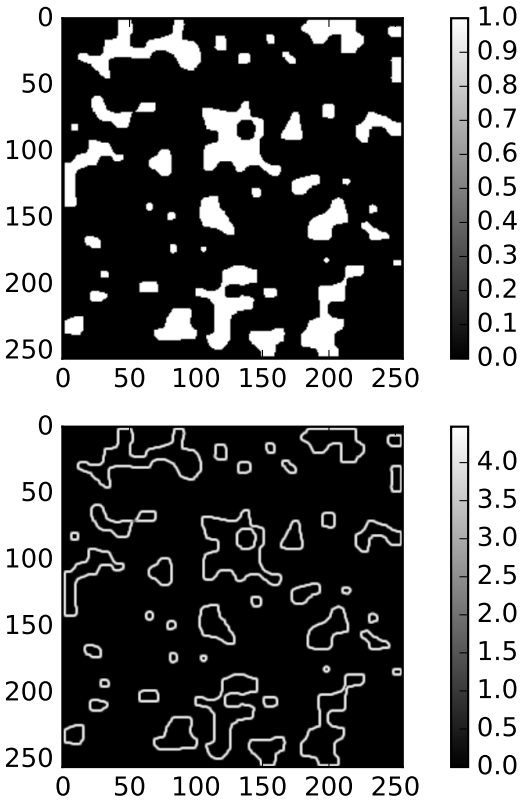


5.1.3.9 Recherche de contours

Si on cherche maintenant à trouver les contours des zones blanches. On peut combiner les opérateurs de dérivation:

```
from PIL import Image
import numpy as np
from matplotlib import pyplot as plt
from matplotlib import cm
from scipy import ndimage
im = Image.open('../Slides/figures/image.jpg')
channels = im.split()
z = np.array(channels[0])
seuil = 150.
```

```
zs = z > seuil
zse = ndimage.morphology.binary_erosion(zs, structure=np.ones((3,3)))
zsd = np.float64(ndimage.morphology.binary_erosion(zse, structure=np.ones((3,3))))
zgx = ndimage.sobel(zsd, axis=0, mode='constant')
zgy = ndimage.sobel(zsd, axis=1, mode='constant')
zsob = np.hypot(zgx, zgy)
fig = plt.figure()
plt.clf()
fig.add_subplot(2, 1, 1)
plt.imshow(zsd, origin = "upper", cmap = cm.gray)
plt.colorbar()
fig.add_subplot(2, 1, 2)
plt.imshow(zsob, origin = "upper", cmap = cm.gray)
plt.colorbar()
plt.show()
```



Les performances de la détection sont meilleures avec un filtre dédié comme le filtre de Canny .

5.1.4 Travaux Dirigés

On vous propose de travailler sur l'image suivante: (source). On vous demande de faire un programme qui effectue les tâches suivantes:

1. Lire l'image et la convertire au format *numpy.array*. Bien que d'aspect grisatre, l'image est en couleur et

comporte donc 3 canaux. Il convient donc de choisir le canal le plus avantageux.

2. Rogner séparer l'image en deux parties pour séparer le bandeau inférieur de l'image elle même.
3. Tracer l'histogramme de l'image et en déduire un moyen de séparer les deux phases.
4. Calculer la proportion de particules dans l'image.
5. Compter les particules.
6. Déterminer la taille moyenne des particules.

5.2 MECA653: Traitement d'image sur une carte de l'Europe

On vous demande de travailler une image topographique de l'Europe. Téléchargez l'image et sauvergardez la dans votre répertoire de travail. Notez qu'un pixel représente 1 km de coté.

```
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 import ndimage
%matplotlib nbagg
```

```
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
Z = np.array(im).astype(np.float64)      # On convertir l'image en array
max_altitude = 1000.                     # Altitude maximale en metres, cette donnee␣
↪est un peu douteuse, (a confirmer).
Z = Z / Z.max() * max_altitude           # On recale les altitudes
Z = Z[::4, ::4] # Sous échantillonage
```

```
# AFFICHAGE
plt.figure()
plt.imshow(Z, cmap = cm.terrain, interpolation = "nearest")                    #␣
↪Affichage de l'altitude
cbar = plt.colorbar()             # Ajout d'une barre d'echelle
cbar.set_label('Altitude $m$')    # On specifie le label en z
plt.xlabel('$km$')    # On specifie le label en x
plt.ylabel('$km$')    # On specifie le label en y
plt.title('Altitudes en Europe') # On specifie le titre
plt.grid()
plt.show()                        # On affiche l'image
```

```
<IPython.core.display.Javascript object>
```

5.2.1 Partie 1: Quelle est la surface de terre présente sur la carte ?

5.2.2 Partie 2: Dans cette surface de terre, quelle est la proportion d'iles ?

5.2.3 Partie 3: Parmi les iles, quelles sont les 5 plus grandes dans l'ordre ? Affichez les.

5.2.4 Partie 4: Quelle proportion de surface de terre perdrait l'Europe si le niveau de la mer montait de 10 m? Même question pour 50 m, 100m et 200m.

5.2.5 Quelle est l'ile la plus étendue d'est en ouest ?

Practical work (TP) notebook file can be downloaded here:

- `MECA653_ImageProcessing_Europe.ipynb`

CHAPTER 6

Optimization

```
# Setup
%matplotlib inline
import numpy as np
import matplotlib.pyplot as plt
import matplotlib
params = {'font.size'     : 14,
          'figure.figsize':(10.0, 6.0),
          'lines.linewidth': 2.,
          'lines.markersize': 8,}
matplotlib.rcParams.update(params)
```

6.1 Optimization

6.2 Scope

Mathematical optimization aims at solving various kinds of problems by minimizing a function of the form:

$$f(X) = e$$

Where f if the **cost function**, X is a N dimensional vector of parameters and $e \in R$. More informations about the underlying theory, the nature of the solution(s) and practical considerations can be found:

- On Wikipedia,
- On (excellent) Scipy lectures.

6.3 Solving

Scipy offers multiple approaches in order to solve optimization problems in its sub package *optimize*

6.3.1 General purpose approach

scipy.optimize.minimize allows one to use multiple general purpose optimization algorithms.

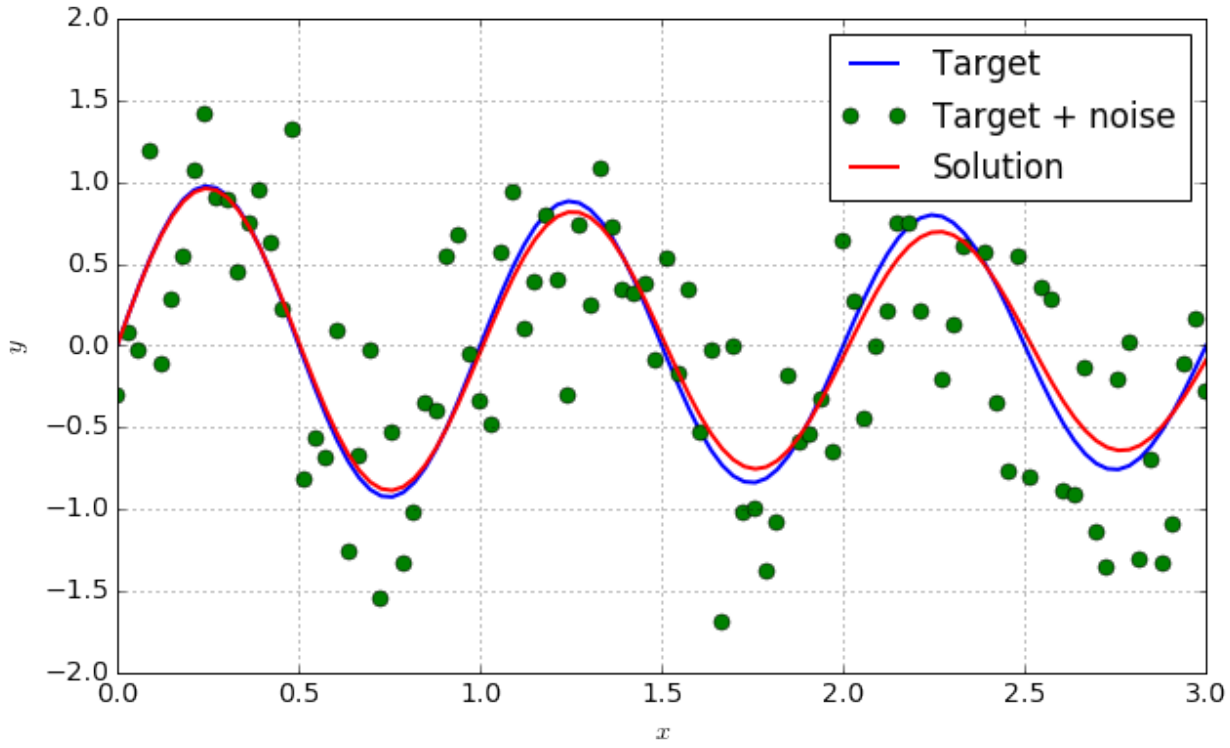
```
from scipy import optimize

def f(X):
  """
  Cost function.
  """
  return (X**2).sum()

X0 = [1.,1.] # Initial guess
sol = optimize.minimize(f, X0, method = "nelder-mead")
X = sol.x
print "Solution: ", X
```

```
Solution:  [ -2.10235293e-05   2.54845649e-05]
```

6.3.2 Curve fitting using least squares

In order to perform curve fitting in a more convenient way, scipy.optimize.curve_fit can be used.

```
def func(x, omega, tau):
    return np.exp(-x / tau) * np.sin(omega * x)


xdata = np.linspace(0, 3., 100)
y = func(xdata, omega = 2. * np.pi, tau = 10.)
ydata = y + .5 *  np.random.normal(size=len(xdata))

params, cov = optimize.curve_fit(func, xdata, ydata)
omega, tau  = params
ysol = func(xdata, omega, tau)


fig = plt.figure(0)
plt.clf()
plt.plot(xdata, y, label = "Target")
plt.plot(xdata, ydata, "o", label = "Target + noise")
plt.plot(xdata, ysol, label = "Solution")
plt.grid()
plt.xlabel("$x$")
plt.ylabel("$y$")
plt.legend()
plt.show()
```



6.4 Optimization tutorials (TD)

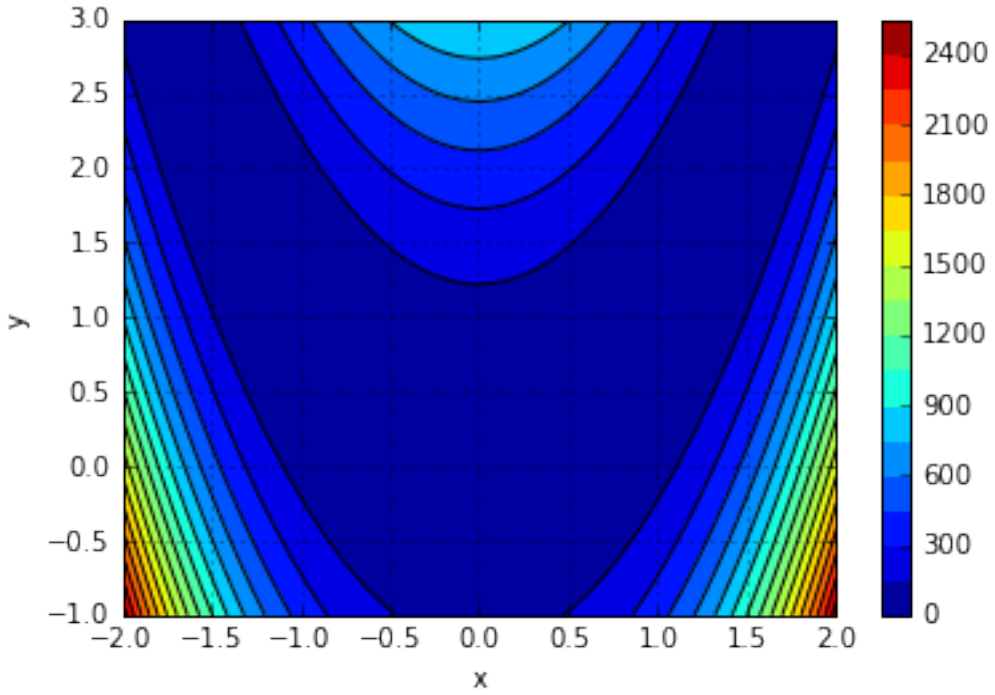
6.4.1 The Rosenbrock function

The Rosenbrock function is a classical benchmark for optimization algorithms. It is defined by the following equation:

$$f(x, y) = (1 - x)^2 + 100(y - x^2)^2$$

```
%matplotlib inline
import numpy as np
import matplotlib.pyplot as plt

def Rosen(X):
    """
    Rosenbrock function
    """
    x, y = X
    return (1-x)**2 + 100. * (y-x**2)**2

x = np.linspace(-2., 2., 100)
y = np.linspace(-1., 3., 100)
X, Y = np.meshgrid(x,y)
Z = Rosen( (X, Y) )

fig = plt.figure(0)
plt.clf()
plt.contourf(X, Y, Z, 20)
plt.colorbar()
```

```
plt.contour(X, Y, Z, 20, colors = "black")
plt.grid()
plt.xlabel("x")
plt.ylabel("y")
plt.show()
```



6.4.1.1 Questions

1. Find the minimum of the function using brute force. Comment the accuracy and number of function evaluations.
2. Same question with the simplex (Nelder-Mead) algorithm.

6.4.2 Curve fitting

1. Chose a mathematical function $y = f(x, a, b)$ and code it.
2. Chose target values of a and b that you will try to find back using optimization.
3. Evaluate it on a grid of x values.
4. Add some noise to the result.
5. Find back a and b using *curve_fit*

6.5 Practical work: optimizing a bridge structure

6.5.1 Installation of the *truss* package

For this session, you will need the Python package Truss that can be download here:

https://github.com/lcharleux/truss/archive/master.zip

Dowload it, extract the content of the archive and put it in your work directory. Once this is completed, execute the cell below to check the install is working.

```
%load_ext autoreload
%autoreload 2
import numpy as np
import matplotlib.pyplot as plt
%matplotlib nbagg
import sys, copy, os
from scipy import optimize
sys.path.append("truss-master")
try:
    import truss
    print("Truss is correctly installed")
except:
    print("Truss is NOT correctly installed !")
```

```
Truss is correctly installed
```

A short truss tutorial is available here:

http://truss.readthedocs.io/en/latest/tutorial.html

6.5.2 Building the bridge structure

In this session, we will modelled a bridge structure using truss and optimize it using various criteria. The basic structure is introduced below. It is made of steel bars and loaded with one vertical force on G. The bridge is symmetrical so only the left half is modelled.

```
E       = 210.e9   # Young Modulus [Pa]
rho     = 7800.    # Density       [kg/m**3]
A       = 5.e-2    # Cross section [m**2]
sigmay  = 400.e6   # Yield Stress  [Pa]

# Model definition
model = truss.core.Model() # Model definition

# NODES
nA = model.add_node((0.,0.), label = "A")
nC = model.add_node((3.,0.), label = "C")
nD = model.add_node((3.,3.), label = "D")
nE = model.add_node((6.,0.), label = "E")
nF = model.add_node((6.,3.), label = "F")
nG = model.add_node((9.,0.), label = "G")
nH = model.add_node((9.,3.), label = "H")

# BOUNDARY CONDITIONS
nA.block[1] = True
nG.block[0] = True
nH.block[0] = True

# BARS
AC = model.add_bar(nA, nC, modulus = E, density = rho, section = A, yield_stress =␣
↪sigmay)
CD = model.add_bar(nC, nD, modulus = E, density = rho, section = A, yield_stress =␣
↪sigmay)
```

```
AD = model.add_bar(nA, nD, modulus = E, density = rho, section = A, yield_stress =␣
↪sigmay)
CE = model.add_bar(nC, nE, modulus = E, density = rho, section = A, yield_stress =␣
↪sigmay)
DF = model.add_bar(nD, nF, modulus = E, density = rho, section = A, yield_stress =␣
↪sigmay)
DE = model.add_bar(nD, nE, modulus = E, density = rho, section = A, yield_stress =␣
↪sigmay)
EF = model.add_bar(nE, nF, modulus = E, density = rho, section = A, yield_stress =␣
↪sigmay)
EG = model.add_bar(nE, nG, modulus = E, density = rho, section = A, yield_stress =␣
↪sigmay)
FH = model.add_bar(nF, nH, modulus = E, density = rho, section = A, yield_stress =␣
↪sigmay)
FG = model.add_bar(nF, nG, modulus = E, density = rho, section = A, yield_stress =␣
↪sigmay)
GH = model.add_bar(nG, nH, modulus = E, density = rho, section = A, yield_stress =␣
↪sigmay)

# STRUCTURAL LOADING
nG.force = np.array([0., -1.e6])


model.solve()



xlim, ylim = model.bbox(deformed = False)
fig = plt.figure()
ax = fig.add_subplot(1,1,1)
ax.set_aspect("equal")
#ax.axis("off")
model.draw(ax, deformed = False, field = "stress", label = True, force_scale = 1.e-6,␣
↪forces = True)
plt.xlim(xlim)
plt.ylim(ylim)
plt.grid()
plt.xlabel("Axe $x$")
plt.ylabel("Axe $y$")
```

```
<IPython.core.display.Javascript object>
```

```
<matplotlib.text.Text at 0x7f7e4843ee48>
```

6.5.2.1 Detailed results at the nodes

```
model.data(at = "nodes")
```

6.5.2.2 Detailed results on the bars

```
model.data(at = "bars")
```

6.5.2.3 Dead (or structural) mass

```
m0 = model.mass()
m0 * 1.e-3 # Mass in tons !
```

```
14.323889603929565
```

6.5.3 Questions

Question 1: Verify that the yield stress is not exceeded anywhere, do you think this structure has an optimimum weight ? You can use the *state/failure* data available on the whole model.

```
# Example:
model.data(at = "bars").state.failure.values

#...
```

```
array([False, False, False, False, False, False, False, False, False,
       False, False], dtype=object)
```

Question 2: Modify all the cross sections at the same time in order to minimize weight while keeping acceptable stress level.

Question 3: We want to modify the position along the $\vec{y}$ axis of the points D, F and H in order to minimize the vertical displacement of the node G times the mass of the structure α:

$$\alpha = |u_y(G)|m$$

Where $u_y(G)$ is the displacement of the node G along the $\vec{y}$ axis and m the mass of the whole structure.

Do not further modify the sections determined in question 4. Comment the solution.

Question 4: Same question with displacements also along $\vec{x}$ of C, D, E and F. Is it better ?

Question 5: You can now try to perform topological optimization by removing/merging well chosen beams and nodes. In order to make the structure even more efficient.

Question 6: You are now asked to optimize the cross section along with the position of C, D, E and F in order to reach the yield stress in each individual beam.

Practical work (TP) notebook file can be downloaded here:

- `Optimization_practical_work.ipynb`