Programmation et Algorithmes numériques 2
Introduction

D. Cornu

S4-2019

Organisation de l'enseignement

Partie 1 (S3) : Bases de la programmation

- Représentation des nombres en mémoire .
- Programmation impérative : conditions, boucles, fonctions, ...
- Algorithmique : algorithmes simples, notions de complexité algorithmique, ordre des erreurs ...

Partie 2 (S4) : Méthodes numériques

- **Principes de diverses méthodes numériques :** système linéaire, interpolation, équation différentielle ordinaire,...
- **Mise en oeuvre :** programmation, usage de bibliothèques numériques.
- **Interprétation physique** des résultats, maitrise de la précision de calcul...

Organisation de l'enseignement

Enseignement réparti entre :

D. Cornu (remplace V. Ballenegger) (4 cours + 3 TP) Février à début Mars
J. Montillaud (4 cours + 3 TP) Mars à mi-Avril

Modalités d'évaluation :

Comptes rendus de TP à la fin de chaque séance , papier quand necessaire et codes sources par mail. DS sur table en fin de semestre partagé entre les deux parties du cours.

Supports de Cours et TP

Tous les supports de cours (slides, sujets de TP, ...) pour la première partie (D. Cornu) seront disponibles sur **GitHub** et mis à jour progressivement :

https://github.com/Deyht/PAN2_L2

Quelques commandes utiles :

```
sudo apt-get install git
git clone https://github.com/Deyht/PAN_L2
git pull
```

Aussi accessible depuis GitHub Desktop ou directement depuis le site

Pour ne pas risquer de perdre votre travail en cas de mise à jour copiez ces fichiers dans un autre répertoire !

Point sur les outils utilisés

Language **Python**
Modules usuels **Matplotlib, numpy** **voir le memento**

scipy pour (**sci**entific **py**thon)
Boîte à outils très répandue pour le calcul scientifique

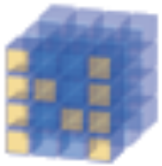
Point sur les outils utilisés

Language **Python**
Modules usuels **Matplotlib,**
numpy **voir le memento**

scipy pour (**sci**entific **py**thon)
Boîte à outils très répandue pour le calcul scientifique

La compréhension des méthodes numériques pour le calcul scientifique ainsi que la maitrise de ces outils sont des compétences hautement valorisables dans tous les domaines liés au numérique.

SciPy (pronounced "Sigh Pie") is a Python-based ecosystem of open-source software for mathematics, science, and engineering. In particular, these are some of the core packages:



NumPy
Base N-dimensional array package

SciPy library
Fundamental library for scientific computing

Matplotlib
Comprehensive 2D Plotting

IP[y]: IPython

IPython
Enhanced Interactive Console

Sympy
Symbolic mathematics

pandas
Data structures & analysis

Détail du module scipy

- Clustering package (**`scipy.cluster`**)
- Constants (**`scipy.constants`**)
- Discrete Fourier transforms (**`scipy.fftpack`**)
- Integration and ODEs (**`scipy.integrate`**)
- Interpolation (**`scipy.interpolate`**)
- Input and output (**`scipy.io`**)
- Linear algebra (**`scipy.linalg`**)
- Miscellaneous routines (**`scipy.misc`**)
- Multi-dimensional image processing (**`scipy.ndimage`**)
- Orthogonal distance regression (**`scipy.odr`**)
- Optimization and Root Finding (**`scipy.optimize`**)
- Signal processing (**`scipy.signal`**)
- Sparse matrices (**`scipy.sparse`**)
- Sparse linear algebra (**`scipy.sparse.linalg`**)
- Compressed Sparse Graph Routines (**`scipy.sparse.csgraph`**)
- Spatial algorithms and data structures (**`scipy.spatial`**)
- Special functions (**`scipy.special`**)
- Statistical functions (**`scipy.stats`**)
- Statistical functions for masked arrays (**`scipy.stats.mstats`**)
- Low-level callback functions

Détail du module scipy

- Clustering package (**scipy.cluster**)
- Constants (**scipy.constants**)
- Discrete Fourier transforms (**scipy.fftpack**)
- Integration and ODEs (**scipy.integrate**)
- Interpolation (**scipy.interpolate**)
- Input and output (**scipy.io**)
- Linear algebra (**scipy.linalg**)
- Miscellaneous routines (**scipy.misc**)
- Multi-dimensional image processing (**scipy.ndimage**)
- Orthogonal distance regression (**scipy.odr**)
- Optimization and Root Finding (**scipy.optimize**)
- Signal processing (**scipy.signal**)
- Sparse matrices (**scipy.sparse**)
- Sparse linear algebra (**scipy.sparse.linalg**)
- Compressed Sparse Graph Routines (**scipy.sparse.csgraph**)
- Spatial algorithms and data structures (**scipy.spatial**)
- Special functions (**scipy.special**)
- Statistical functions (**scipy.stats**)
- Statistical functions for masked arrays (**scipy.stats.mstats**)
- Low-level callback functions

Utilisation de SciPy :

```
#import all scipy
import scipy as sp
    ...
#import specific package
from scipy import linalg
#ou
import scipy.linalg as la
    ...
#exemple
import math as m
import numpy as np
import scipy as sp
m.sqrt(-1)
# math domain error
np.sqrt(-1)
# nan (not a number)
sp.sqrt(-1)
# 1j
```

On préférera généralement scipy à numpy quand des fonctions identiques sont présentes dans les deux modules.

Savoir se documenter

Le site **SciPy.org** documente très précisément les différents sous-modules et décris avec une très grande précision l'usage de chaque fonction.

C'est également le cas pour **numpy**, **matplotlib**, ...

La documentation regorge d'**exemples** pour presque tout ce que vous pourriez immaginer. **Elle contient également le code source !**

Au moindre doute ou problème, consultez la documentation !

Présentation de la méthode de Newton-Raphson

En partant d'un point de départ x_n la tangente en ce point de la courbe s'écrit :

$$y = f'(x_n)(x - x_n) + f(x_n) \quad (1)$$

On définit alors x_{n+1} comme le point ou $y = 0$. Ce qui nous permet de définir la suite suivante :

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)} \quad (2)$$

qui converge vers la solution voulue $f(x) = 0$.

Utilisation de la méthode précédente

Implementation simple :

```
def NewtonRaphson(f,fprime, x,
    eps=1e-8, max_iter=50):
  i = 0
  h = f(x)/fprime(x)
  while abs(h) >= eps and i <=
      max_iter:
    h = f(x)/fprime(x)
    x = x - h
    #x(i+1) = x(i) - f(x)/f'(x)
    i += 1
  print x
  ...
1.414213562373095
```

Utilisation de la méthode précédente

Implementation simple :

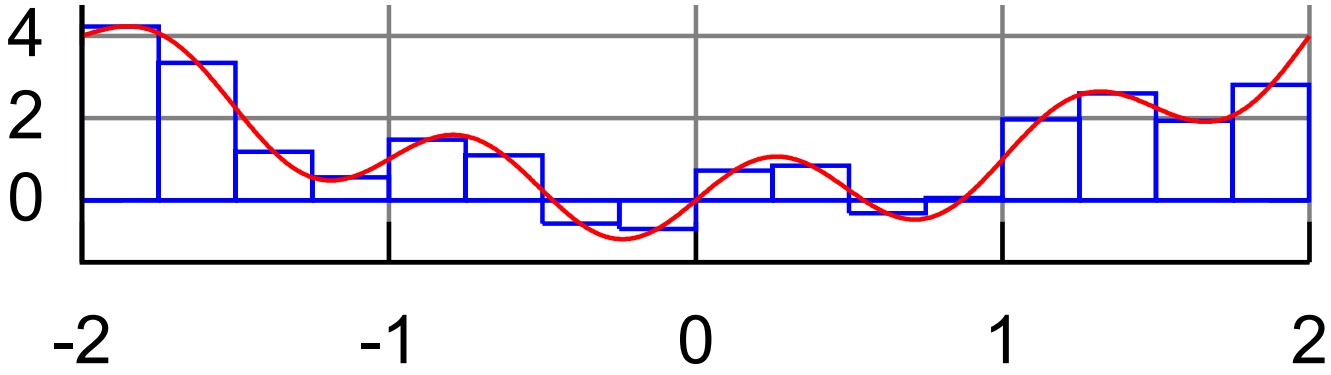
```
def NewtonRaphson(f,fprime, x,
    eps=1e-8, max_iter=50):
  i = 0
  h = f(x)/fprime(x)
  while abs(h) >= eps and i <=
      max_iter:
    h = f(x)/fprime(x)
    x = x - h
    #x(i+1) = x(i) - f(x)/f'(x)
    i += 1
  print x
  ...
1.414213562373095
```

Résolution avec scipy :

```
import scipy.optimize

def f(x):
  return x**2 - 2

def fprime(x):
  return 2*x

scipy.optimize.newton(f, 3,
    fprime)
  ...
1.414213562373095
```

Résultat identique dans un cas simple. Néanmoins, la fonction du module scipy est capable de prendre en compte la dérivée seconde ou encore de faire cette opération facilement sur des vecteurs, de gérer automatiquement le nombre d'itérations necessaires ... $\rightarrow$ **Voir la doc !**

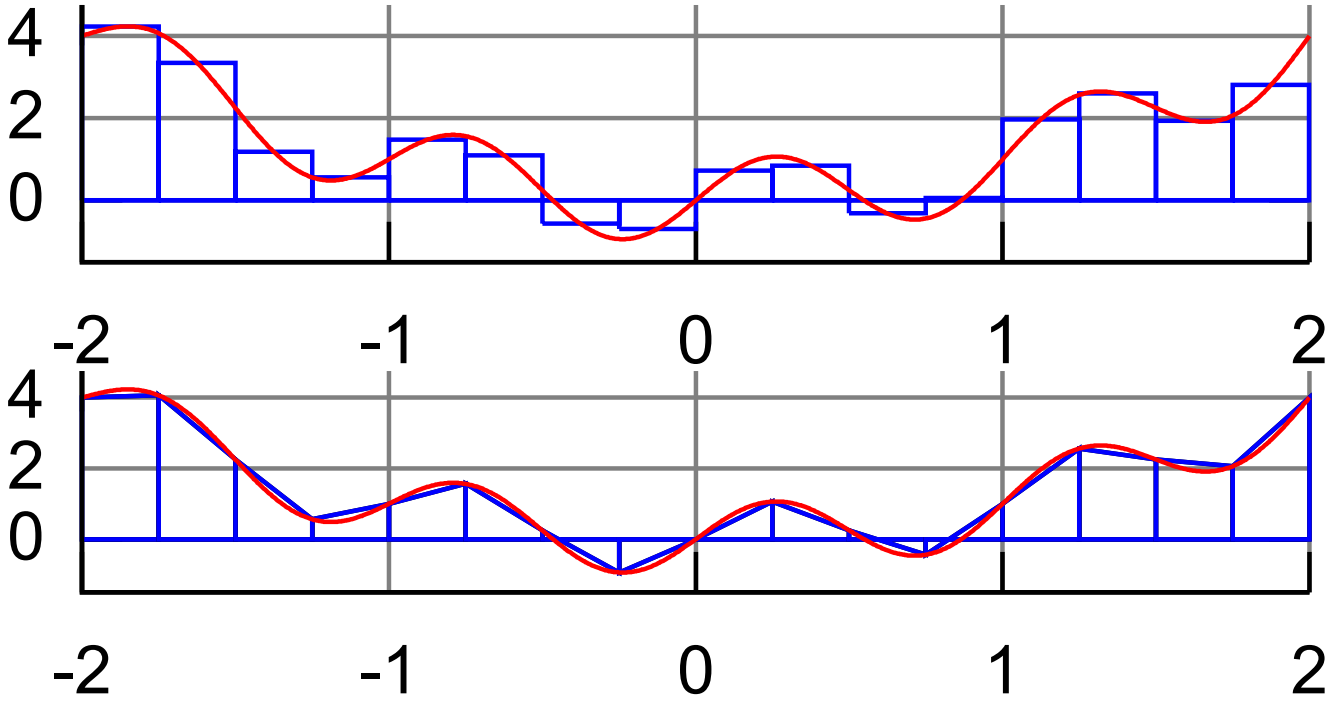
Exemple 2 : Intégration numérique de fonctions 1D



Méthode des Rectangles

Interpolation par paliers

Exemple 2 : Intégration numérique de fonctions 1D



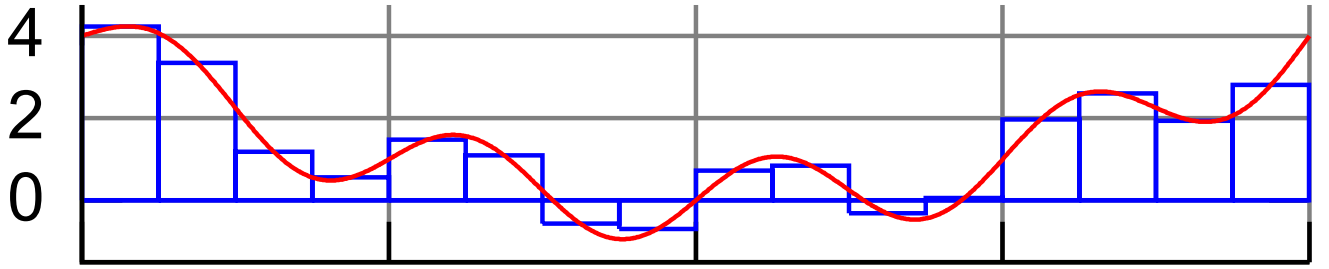
Méthode des Rectangles

Interpolation par paliers

Méthode des Trapèzes

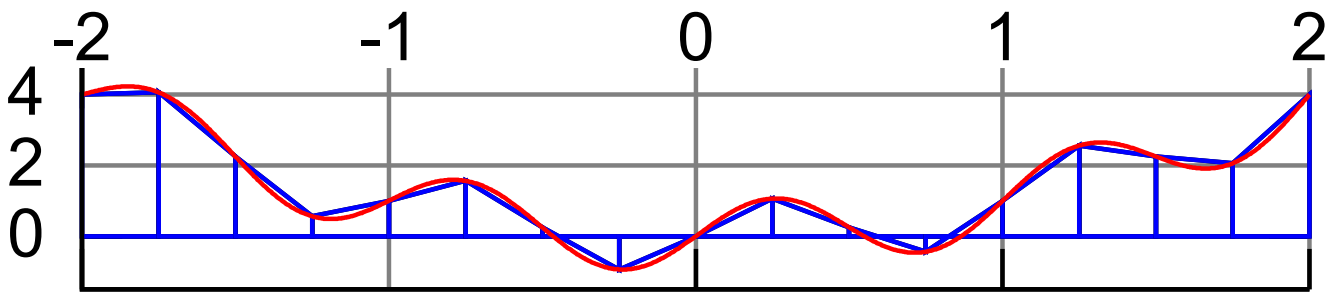
Interpolation linéaire

Exemple 2 : Intégration numérique de fonctions 1D



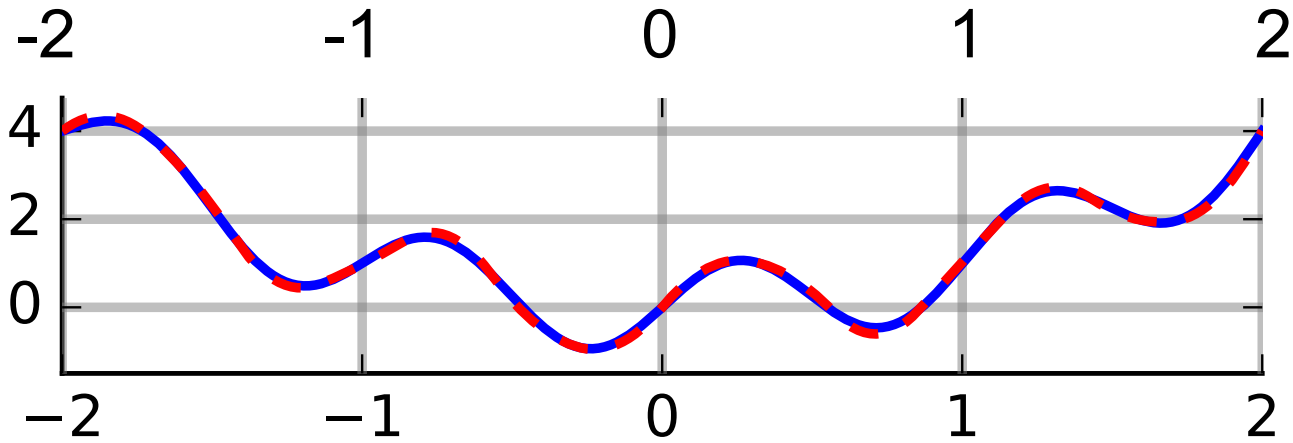
Méthode des Rectangles

Interpolation par paliers



Méthode des Trapèzes

Interpolation linéaire



Méthode de Simpson

Interpolation cubique

Exemple 2 : Intégration numérique de fonctions 1D

On appelle courament les formules d'intégration numériques des* **formules de quadrature

Quadrature avec scipy.integrate

```
>>> help(integrate)
 Methods for Integrating Functions given function object.

   quad          -- General purpose integration.
   dblquad       -- General purpose double integration.
   tplquad       -- General purpose triple integration.
   fixed_quad    -- Integrate func(x) using Gaussian quadrature of order n.
   quadrature    -- Integrate with given tolerance using Gaussian quadrature.
   romberg       -- Integrate func using Romberg integration.

 Methods for Integrating Functions given fixed samples.

   trapz         -- Use trapezoidal rule to compute integral from samples.
   cumtrapz      -- Use trapezoidal rule to cumulatively compute integral.
   simps         -- Use Simpson's rule to compute integral from samples.
   romb          -- Use Romberg Integration to compute integral from
                    (2**k + 1) evenly-spaced samples.
```

Exemple 2 : Utilisation des fonctions scipy.integrate

```
1 import numpy as np
2 import matplotlib.pyplot as plt
3 import scipy as sp
4 import scipy.integrate as integ
5
6 x = np.linspace(0,5,100)
7 y = sp.sin(x) + sp.cos(2*x)
8
9 print integ.simps(y, x)
10
11 def f_int(x):
12   return -sp.cos(x) + sp.sin(2*x)/2
13
14 print (f_int(5) - f_int(0))
15
16 plt.plot(x,y)
17 plt.show()
```

Résultat :
0.444329314138 # Simpson
0.444327259092 # Analytique

Simpson est une **méthode d'ordre 3** :
les termes négligés sont ceux en $O(h^3)$.

Certaines méthodes sont d'ordre plus élevé $\rightarrow$
Romberg, ordre 4

Coder à la main VS Biliothèques/Modules

- Meilleure compréhension des mécanismes du code
- Meilleure adaptation à un problème spécifique
- Souvent plus rapide si adapté à son materiel
- Permet théoriquement de résoudre n'importe quel problème

- **Temps de developpement souvent bien plus long !**

VS

- Pas ou peu de temps de developpement
- Souvent très bien documenté, grosse communauté
- Pré-optimizé, temps de calcul correct sur la majorité des machines

- Si librairie arrêtée → codes obsolètes !
- Disponibles uniquement pour les problèmes les plus courent
- Très souvent mal employé car le détail n'est pas compris