

Résumé du cours : Simulation Numérique

HOUCMEDDINE FILALI

HOUSSEMEDDINE BOUZIDI

Python offre des modules permettant de résoudre des problèmes de simulation numérique. Ces modules sont :

numpy : contient principalement la classe **numpy.ndarray** permettant de représenter des vecteurs (à plusieurs dimensions). Il comporte aussi des fonctionnalités mathématiques basées sur les **ndarrays** (algèbre linéaire, transformée de fourrier, simulation de champs aléatoires etc.).

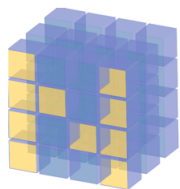
scipy : basé sur **numpy** contient des fonctionnalités de résolution approchée d'équations, équations différentielles, intégrales etc.

matplotlib : basé sur **numpy** permet de visualiser les données contenues dans un **ndarray** sous forme de graphique.



Ces modules sont importés sous les alias suivants :

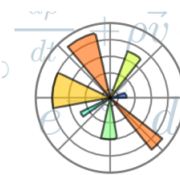
```
import numpy as np
import matplotlib.pyplot as plt
from scipy import integrate
from scipy import optimize
```



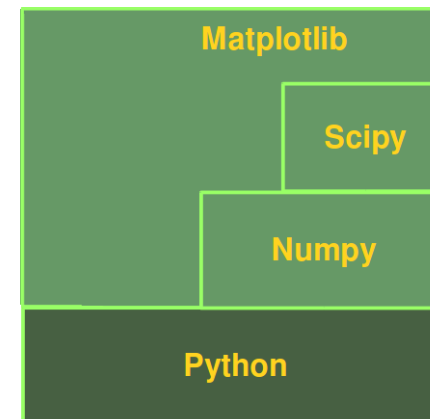
NumPy



SciPy



matplotlib





fonctions fabriques de ndarrays (tableaux)

Ces fonctions permettent d'instancier des formes particulières de tableaux multidimensionnels. Le résultat de l'invocation de ces fonctions est toujours une instance de la classe `np.ndarray`.

Description	Syntaxe	Exemple(s) & Remarques
Création à partir d'un itérable	<code>np.array(itérable source)</code>	<pre>v = np.array([1,2,3]) ##vecteur 1-D m = np.array([[1,2],[2,1]]) ##matrice 2x2</pre>
Discrétisation d'intervalles	<pre>np.arange(start = 0, end_ex, step = 1) np.linspace(start, end, num = 50)</pre>	<pre>np.arange(0, 1, 0.2) #donne np.array([.0, .2, .4, .6, .8]) np.linspace(0,1,5) #donne np.array([.0, .2, .4, .6, .8, 1.])</pre>
ndarrays homogènes	<pre>np.zeros(shape, dtype=float) np.ones(shape, dtype=float) np.full(shape, fill_value, dtype = None)</pre>	<pre>np.zeros(3) #np.array([.0,.0,.0]) np.ones((2,3), np.int) #np.array([[1, 1, 1], [1,1,1]]) np.full((3,3), 5) #array([[5, 5], [5, 5]])</pre>
matrices particulières	<pre>np.identity(ordre, dtype=float) np.diag(itérable)</pre>	<pre>m = np.diag([1,2,3]) #np.array([[1, 0, 0], [0, 2, 0], [0, 0, 3]]) np.diag(m) #np.array([1,2,3])</pre>
remplissage en fonction des indices	<code>np.fromfunction(func, shape)</code>	<pre>np.fromfunction(lambda i,j :i+j, (2,2)) #np.array([[0, 1], [1, 2]])</pre>
Création d'un tableau non initialisé	<pre>np.empty(shape,dtype = np.float) ou bien np.ndarray(shape, dtype = np.float)</pre>	



Propriétés utile des ndarrays

Les instances de la classe ndarray peuvent être manipulées à travers leurs propriétés. Dans la suite, on suppose que `a` est une instance de la classe `np.ndarray`

Description	Syntaxe	Exemple(s) & Remarques
Donne un tuple contenant la forme du <code>np.ndarray</code> : — le nombre de dimensions est égal à la taille du tuple. — la $i^{\text{ème}}$ valeur du tuple indique la taille de la $i^{\text{ème}}$ dimension du <code>np.ndarray</code>. — accessible en lecture/écriture (l'opération d'écriture doit impérativement conserver le nombre total d'éléments dans le <code>np.ndarray</code>, sinon ValueError).	<code>a.shape</code> <i>## object form</i> Ou bien <code>np.shape(a)</code> <i>## procedural form</i>	<pre>a = np.array([1,2,3]) a.shape # (3,) b = np.array([[1,2,3], [0,1,0]]) np.shape(b) # (2,3) b.shape = (3,2) b# np.array([[1,2],[3,0],[1,0]])</pre>
Donne le type des éléments du <code>np.ndarray</code>. Cette propriété est accessible en lecture seule.	<code>a.dtype</code>	<pre>a = np.array([1,2,3]) a.dtype #np.int a = np.array([1, 3+2j, 4]) a.dtype #np.complex</pre>
Donne le nombre total des éléments du ndarray <code>a.size = len(a.shape)-1</code> $\prod_{i=0}^{a.size-1} (a.shape[i]).$ Cette propriété est accessible en lecture seule.	<code>a.size</code> ou bien <code>np.size(a)</code>	<pre>a = np.array([1,2,3]) a.size # 3 b = np.array([[1,2,3], [0,1,0]]) np.size(b) # 6</pre>
Donne le nombre total de dimensions du <code>np.ndarray</code> : <code>a.ndim=len(a.shape)</code>. Cette propriété est accessible en lecture seule.	<code>a.ndim</code> ou bien <code>np.ndim(a)</code>	<pre>a = np.array([1,2,3]) a.ndim # 1 b = np.array([[1,2,3], [0,1,0]]) np.ndim(b) # 2 len(b.shape) #2</pre>
Donne la transposée d'un <code>np.ndarray</code> à plus que deux dimensions.	<code>a.T</code> ou bien <code>np.transpose(a)</code>	<pre>a = np.array([1,2,3]) a.T #array([1, 2, 3]) b = np.array([[1,2,3], [0,1,0]]) np.transpose(b) #array([[1, 0],[2, 1],[3, 0]])</pre>
Donne un <code>np.ndarray</code> contenant les parties réelles respectivement imaginaires des éléments du <code>np.ndarray</code> source.	<code>a.real</code> ou <code>np.real(a)</code> <code>a.imag</code> ou <code>np.imag(a)</code>	<pre>a = np.array([[1, 2+3j], [1j, 3-2j]]) a.real # array([[1., 2.],[0., 3.]]) np.imag(a) # array([[0., 3.],[1., -2.]])</pre>





Méthodes utiles des ndarrays

Dans la suite, on suppose que a et b sont deux instances de la classe np.ndarray

Description	Syntaxe	Exemple(s) & Remarques
Donne la trace d'un np.ndarray avec un nombre de a.ndim >= 2	a.trace() ou bien np.trace(a)	a = np.array([1,2,3]) a.trace() <i>#ValueError</i> b = np.array([[1,2,3], [0,1,0]]) np.trace(b) <i>#2</i>
Donne la transposée d'un np.ndarray	a.transpose() ou bien a.transpose(permutation_axes)	
Donne un np.ndarray obtenu à partir du np.ndarray source en modifiant sa forme : — Le nombre total des éléments du np.ndarray source doit être conservé. — La modification du np.ndarray obtenu engendre la modification du np.ndarray source.	a.reshape(shape) ou np.reshape(a, shape)	a = np.array([[1,2,3], [1,2,3]]) b = a.reshape((3,2)) <i>#b = array([[1, 2], [3, 1], [2, 3]])</i> b[0,0] = 0 b <i># array([[0, 2],[3, 1],[2, 3]])</i> a <i># array([[0, 2, 3], [1, 2, 3]])</i>
retourne une copie indépendante du np.ndarray source.	a.copy() ou bien np.copy(a)	a = np.array([[1,2,3], [1,2,3]]) b = a.reshape((3,2)).copy() <i>#array([[1, 2], [3, 1], [2, 3]])</i> b[0,0] = 0 b <i># array([[0, 2],[3, 1],[2, 3]])</i> a <i># array([[1, 2, 3], [1, 2, 3]])</i>
retourne un np.ndarray en convertissant les éléments du np.ndarray source vers le type indiqué en paramètre.	a.astype(type)	a = np.array([[1,2,3], [1,2,3]]) c = a.astype(np.float) <i>#array([[1., 2., 3.],[1., 2., 3.]])</i>
Donne le produit scalaire ou matriciel entre deux np.ndarray. — produit scalaire pour deux np.ndarray à une seule dimension. — produit matriciel, sinon.	a.dot(b) ou bien np.dot(a,b) ou bien a@b	v1 = np.array([1,2,3]) v2 = np.array([2,0,-1]) v1@v2 <i>#-1</i> m1 = np.array([[1,2,3], [0, 1, 0]]) m2 = np.array([[1,2], [1, 1], [0, 1]]) m1.dot(m2) <i>#array([[3, 7],[1, 1]])</i> np.dot(m2,m1) <i>#array([[1, 4, 3],[1, 3, 3], [0, 1, 0]])</i> v1.dot(m1) <i>#Errordimensionsincompatibles</i> v1.dot(m2) <i>#array([3, 7])</i>
Méthodes d'agrégation	a.sum() ou np.sum(a) a.mean() ou np.mean(a) a.max() ou np.max(a) a.min() ou np.min(a) a.argmax() ou np.argmax(a) a.argmin() ou np.argmin(a)	m1 = np.array([[1,2,3], [0, 1, 0]]) m1.sum() <i>#7</i> m1.sum(axis = 0) <i>#array([1, 3, 3])</i> m1.sum(axis = 1) <i>#array([6, 1])</i>



Description	Syntaxe	Exemple(s) & Remarques
Les opérations arithmétiques et relationnelles natives sont appliquées terme à terme pour le cas des np.ndarray.	+ - * ** / // % == != < > <= >=	<pre>>>> m1 = np.array([[1,2,3], [0, 1, 0], [0,1,1]]) >>> m1+2 #array([[3, 4, 5],[2, 3, 2],[2, 3, 3]]) >>> m1*m1 #array([[1, 4, 9],[0, 1, 0],[0, 1, 1]]) >>> m1.dot(m1) #array([[1, 7, 6],[0, 1, 0],[0, 2, 1]]) >>> m1 < 2 >>> _ array([[True, False, False], [True, True, True], [True, True, True]])</pre>

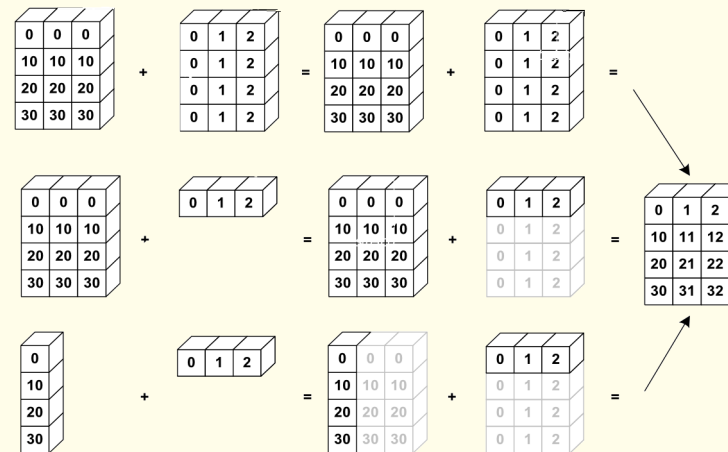
Diffusion « broadcasting » des np.ndarray : numpy interprète les opérations de la forme **a op b** (des instance np.ndarray) terme à terme. Si **a.shape != b.shape**, alors le vecteur ayant la taille la plus petite est diffusé dans le vecteur le plus large.

Règles de diffusion :

- numpy remplit la forme du vecteur de dimension plus petite par des 1 à gauche afin d'avoir deux vecteurs de mêmes dimensions.
- numpy commence à comparer les paires d'entiers dans **a.shape** et **b.shape** à partir de la droite *i* (indice -1 → 0 : deux dimensions **a.shape[i]** et **b.shape[i]** sont compatibles si et seulement si :

$$\begin{cases} -a.shape[i] == 1 \text{ or } b.shape[i] == 1 \\ -a.shape[i] == b.shape[i] \\ \text{sinon} \Rightarrow \text{ValueError} \end{cases}$$

Shape: (3, 2)		Shape: (2,)		Shape: (3, 2)
0 1		4 5		-4 -4
2 4	-	4 5	=	-2 -1
10 10		4 5		6 5



Description	Syntaxe	Exemple(s) & Remarques
Indexage simple/par slicing : - Le nombre d'indices doit toujours être $\leq a.ndim$. - Le même principe que les séquences Python. Indexage à gauche vers la droite pour des indices positifs et de la droite vers la gauche pour les indices négatifs. - Les cases et les morceaux indexés sont accessibles en lecture écriture. - Python applique le mécanisme de diffusion si un scalaire est affecté à un morceau.	<i>#i, j entier ou slice</i> <pre> a[i] a[i,j] a[b:e:s, j] <i>#etc.</i> a[i] = expr a[i,j] = expr a[:,i:j] = expr <i>#etc.</i> </pre>	<pre> m1 = np.array([[1,2,3], [0, 1, 0], [0,1,1]]) m1[0] <i>#première ligne</i> m1[0] = 0 <i>#mettre à zéro la première ligne</i> <i>#placer le vecteur 1,2,3 au niveau de la première ligne</i> m1[0] = [1,2,3] m1[-1,-1] <i>#dernière ligne dernière colonne</i> <i>#m1[1,1,1] IndexError</i> m1[:,0] <i>#première colonne array([1, 0, 0])</i> m1[:2, 1:] <i>#array([[2, 3],[1, 0]])</i> m1[:,0] = 0 <i>#mettre à zéro tous les éléments # de la première colonne</i> </pre>

Indexage par des list / ndarray :

- chaque **list** / **ndarray** doit exclusivement contenir des int valables comme indices pour le tableau source.
- le nombre de **list** / **ndarray** passés est impérativement $\leq$ au nombre de dimensions du tableau source et doivent avoir la même taille.

```

>>> m1[[1,1,0,0]] #donne
array([[0, 1, 0],
       [0, 1, 0],
       [1, 2, 3],
       [1, 2, 3]])
>>> m1[[1,-1],[0,-2]] #donne
array([0, 1])

```

Le masquage des ndarrays : il est possible d'indexer un **np.ndarray** *a* par un autre **np.ndarray** *b* (de même shape et de type booléen). *b* est appelé masque. Dans ce cas, le résultat est un **np.ndarray** à une seule dimension contenant les valeurs qui se superposent avec **True** dans *b*.

A =

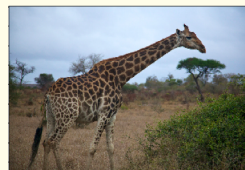
5	4	5
4	4	4
5	4	5

A < 5 →

F	T	F
T	T	T
F	T	F

Base

Binary Mask



```

m1 = np.array([[1,2,3], [0, 1, 0], [0,1,1]])
b = m1 > 0
m1[b] #array([1, 2, 3, 1, 1, 1])

```

Description	Syntaxe	Exemple(s) & Remarques
Le module <code>numpy.linalg</code> contient des fonctions permettant d'appliquer des opérations de l'algèbre linéaire	<pre>from numpy import linalg ou bien import numpy.linalg as linalg</pre>	$a = \begin{pmatrix} 2 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & -2 \end{pmatrix} b = \begin{pmatrix} 1 \\ 2 \\ -1 \end{pmatrix}$ <pre>a = np.diag([2,1,-2]) b = np.array([1,2,-1])</pre>
Calcul du déterminant d'une matrice carrée	<code>linalg.det(a)</code>	<code>linalg.det(a) #-4.0</code>
Le rang d'une matrice	<code>linalg.matrix_rank(a)</code>	<code>linalg.matrix_rank(a) ##3</code>
L'inverse d'une matrice carrée (exception lorsque le déterminant est nul)	<code>linalg.inv(a)</code>	<pre>linalg.inv(a) #array([[0.5, 0. , 0.], # [0. , 1. , 0.], # [-0. , -0. , -0.5]])</pre>
Puissance nième d'une matrice carrée	<code>linalg.matrix_pow(a,k)</code>	<pre>linalg.matrix_power(a,2) #array([[4, 0, 0], # [0, 1, 0], # [0, 0, 4]])</pre>
Revoie <code>numpy.ndarray</code> à une dimension contenant les valeurs propres d'une matrice	<code>linalg.eigvals(a)</code>	<pre>linalg.eigvals(a) #array([2., 1., -2.])</pre>
Renvoie un <code>tuple</code> contenant les valeurs propres puis les vecteurs propres d'une matrice	<code>linalg.eig(a)</code>	<pre>pvals, pvecs = linalg.eig(a) pvals #array([2., 1., -2.]) pvecs #array([[1., 0., 0.], # [0., 1., 0.], # [0., 0., 1.]])</pre>
Résoudre un système d'équations linéaires <code>a x = b</code>	<pre>x = linalg.solve(a,b) ou bien x = linalg.inv(a).dot(b) ou bien x = np.dot(linalg.inv(a), b)</pre>	<pre>x = linalg.solve(a,b) x #array([0.5, 2. , 0.5]) a.dot(x) #array([1., 2., -1.]])</pre>



Description	Syntaxe	Exemple(s) & Remarques
Fonctions mathématiques vectorisées	<pre>np.cos np.sin np.exp np.tan np.acos np.log np.sqrt etc.</pre>	$a = \begin{pmatrix} \frac{\pi}{4} & 0 & 0 \\ 0 & \frac{\pi}{2} & 0 \\ 0 & 0 & \frac{\pi}{3} \end{pmatrix}$ <pre>a = np.diag([np.pi/4, np.pi/2, np.pi/3]) #array([[0.78539816, 0., 0.], # [0., 1.57079633, 0.], # [0., 0., 1.04719755]]) np.sin(a) #array([[0.70710678, 0., 0.], # [0., 1., 0.], # [0., 0., 0.8660254]]) def f(x): return 1 if x != 0 else 0 f = np.vectorize(f) f(a) #array([[1, 0, 0], # [0, 1, 0], # [0, 0, 1]]) #vérifier si deux matrices sont identiques np.all(a == b) np.all(a >= 0) # ∀?v ∈ a : v ≥ 0 = True np.all(a > 0) # ∀?v ∈ a : v > 0 = False np.all(a > 0) # ∃?v ∈ a : v > 0 = True np.all(a <= 0) # ∃?v ∈ a : v ≤ 0 = True np.all(a <= 0) # ∃?v ∈ a : v < 0 = False a = np.random.rand(3,3) ai = np.linalg.inv(a) id3 = ai.dot(a) id3 #array([[1.00000000e+00, -2.59935694e-17, 1.62067702e-16], # [-1.73697592e-17, 1.00000000e+00, -6.08078299e-17], # [2.12959035e-17, 5.05495793e-17, 1.00000000e+00]]) np.allclose(id3, np.identity(3)) #True</pre>
Vectoriser une fonction f . La version vectorisée de f est appliquée terme à terme lorsqu'elle est invoquée avec une instance de la classe <code>np.ndarray</code> .	La classe <code>np.vectorize</code> permet de vectoriser une fonction : <code>fv = np.vectorize(f)</code>	
Vérifier si tous les éléments d'un <code>np.ndarray</code> valent <code>True</code>	<code>np.all(p)</code>	
Vérifier s'il existe au moins un élément du <code>np.ndarray</code> qui vaut <code>True</code>	<code>np.any(p)</code>	
Vérifier si deux <code>np.ndarrays</code> contiennent des valeurs très proches.	<code>np.allclose(a,b)</code>	

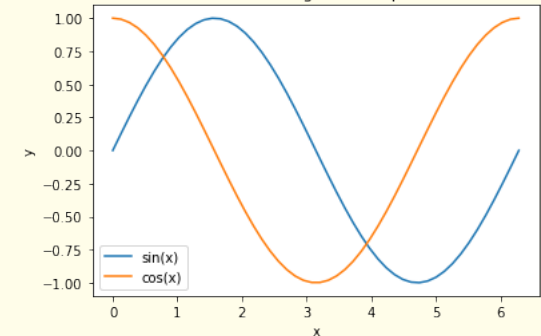


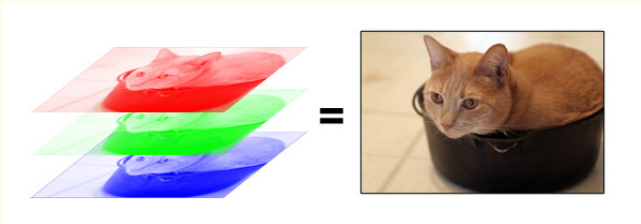


Description	Syntaxe	Exemple(s) & Remarques
Valeurs aléatoires uniformes sur l'intervalle $[0, 1]$	<code>np.random.rand(d1, d2, ..., dn)</code>	<pre>>>> np.random.rand(3,3) array([[0.34696974, 0.08364107, 0.25945056], [0.18289212, 0.27384236, 0.19379444], [0.05854545, 0.56170746, 0.97233758]])</pre>
Valeurs aléatoires selon la loi normale standard $\mu = 0, \sigma = 1$	<code>np.random.randn(d1, d2, ..., dn)</code>	<pre>>>> np.random.randn(3,3) array([[0.59513685, -1.65090605, 0.1103868], [1.34707164, 1.57841324, -0.05837151], [0.86024789, -1.60313356, 1.19716067]])</pre>
Générer une permutation aléatoire	<code>np.random.permutation(taille)</code>	<pre>>>> np.random.permutation(10) array([9, 8, 4, 0, 7, 2, 1, 3, 6, 5])</pre>





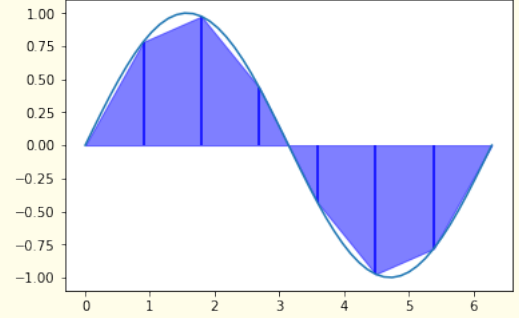
Description	Syntaxe	Exemple(s) & Remarques
importation du module	<pre>from matplotlib import pyplot as plt ou bien import matplotlib.pyplot as plt</pre>	<pre>x = np.linspace(0, 2*np.pi) y1 = np.sin(x) y2 = np.cos(x) plt.plot(x,y1, label = 'sin(x)') plt.plot(x,y2, label= 'cos(x)') plt.xlabel('x') plt.ylabel('y') plt.legend() plt.title('Fonctions trigonometriques ') plt.show()</pre>
Tracer la courbe d'une fonction $y = f(x)$	<pre>plt.plot(x,y)</pre>	produit la figure suivante 

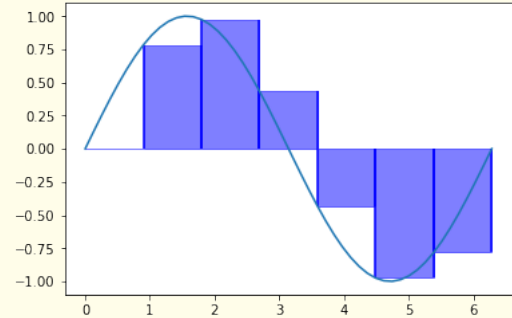
Description	Syntaxe	Exemple(s) & Remarques
Charger une image dans un <code>np.ndarray</code>. Selon la nature de l'image le <code>np.ndarray</code> récupéré peut avoir l'un des deux types suivants : — <code>np.uint8</code> où chaque élément prend des valeurs dans <code>{0...255}</code>. — <code>np.float</code> où chaque élément prend des valeurs dans <code>[0, 1]</code>. Une image en niveau de gris est représentée par un <code>np.ndarray</code> à deux dimensions de <code>shape (h,w)</code> où : — <code>h</code> est la hauteur de l'image. — <code>w</code> est la largeur de l'image. une image couleur est représentée par un <code>np.ndarray</code> <code>I</code> tel que <code>I.shape</code> renvoie <code>(h,w,p)</code> où : — <code>h</code> est la hauteur de l'image. — <code>w</code> est la largeur de l'image. — <code>p</code> est la profondeur de l'image (3 pour les images en couleur codées en RVB).	<pre>a = plt.imread(file_name)</pre> 	<pre>>>> img = plt.imread('pika.jpg') >>> img.shape (225, 225, 3) >>> img.dtype dtype('uint8') >>> img.min(), img.max() (0, 255)</pre>
Visualiser une image encodée dans un <code>np.ndarray</code> <code>a</code>. Utiliser l'option <code>cmap = plt.cm.gray</code> pour visualiser l'image en niveau de gris.	<pre>plt.imshow(a)</pre>	<pre>plt.imshow(img) plt.axis('off') plt.show()</pre>  produit la figure suivante : La relation suivante permet de calculer le niveau de gris <code>g</code> dans un pixel couleur codé en RVB : $g = 30\%r + 59\%v + 11\%b$ <pre>ig = ([0.3, 0.59, 0.11] * img).sum(axis = -1) plt.imshow(ig, cmap = plt.cm.gray) plt.axis('off') plt.show()</pre>  <pre>ib = ig > 240 plt.imshow(ib, cmap = plt.cm.gray) plt.axis('off') plt.show()</pre> 
Enregistrer l'image encodée dans le <code>np.ndarray</code> <code>a</code> dans le fichier indiqué.	<pre>plt.imsave(file_name, a)</pre>	<pre>plt.imsave("pikabw.png", ib)</pre>

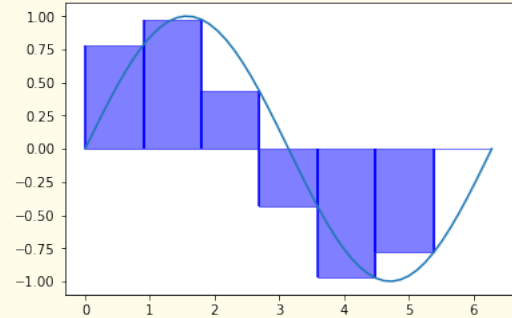


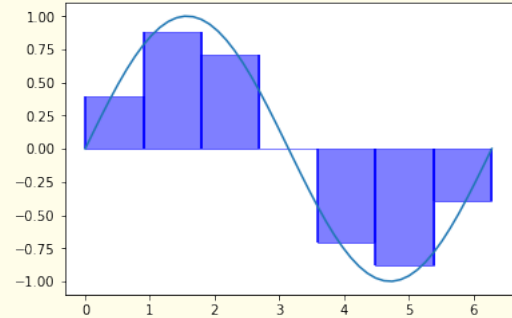
Description	Syntaxe	Exemple(s) & Remarques
importation du module	<pre>from scipy import integrate ou bien import scipy.integrate as integrate</pre>	$\approx \int_0^1 (x^2 - x + 1) dx$ <pre>>>> integrate.quad(lambda x : x**2 - x + 1, 0, 1) (0.8333333333333334, 9.251858538542972e-15) >>> 1/3 - 1/2 + 1 0.8333333333333333</pre>
Intégrale d'une fonction à une seule variable. Renvoie un couple $(\mathcal{A}, \varepsilon)$ où : $\mathcal{A}$: la valeur approchée de l'intégrale $\approx \int_a^b f(x) dx$ ε : est une majoration de l'erreur absolue. Si f est une fonction à plusieurs variables, alors f est intégrée par rapport à son premier paramètre. Les valeurs des paramètres restants sont fixés par le tuple args.	<pre>integrate.quad(f, a, b, args = ())</pre>	Définir la fonction $E_n(x) = \int_1^{+\infty} \frac{\exp(-x t)}{t^n} dt$: <pre>def E(n,x): f = lambda t,x,n : np.exp(-x*t)/t**n tpl = integrate.quad(f, 1, np.inf, args = (x,n)) return tpl[0]</pre> Vérifier que $I_n = \int_0^{+\infty} E_n(x) dx = \frac{1}{n} \forall n \leq 30$ <pre>>>> def I(n): f = lambda x,n : E(n,x) tpl = integrate.quad(f, 0, np.inf, args = (n,)) return tpl[0] >>> I = np.vectorize(I) >>> v1 = np.fromfunction(lambda i : 1/(i+1), (31,)) >>> v2 = np.fromfunction(lambda i : I(i+1), (31,)) >>> np.allclose(v1, v2) True</pre>
Intégrer un vecteur d'échantillons : Étant donné deux vecteurs $x = (x_0 \ x_1 \ \dots \ x_{n-1})$ et $y = (y_0 \ y_1 \ \dots \ y_{n-1})$ où $y_i = f(x_i) \forall i$. Il est possible d'approcher l'intégrale $\int_{x_0}^{x_{n-1}} f(x) dx$ par la somme des aires des trapèzes délimités par les 4 points $(0, x_i), (0, x_{i+1}), (x_i, f(x_i)), (x_{i+1}, f(x_{i+1})) \forall i < n - 1$.	<pre>integrate.quad(x,y) → renvoie un réel contenant l'approximation de l'intégrale comme indiqué.</pre>	Vérifier que la matrice définie par $a = (a_{ij})_{\substack{0 \leq i \leq 9 \\ 0 \leq j \leq 9}}$ où $a_{ij} = \int_{-\pi}^{\pi} \sin(n t) \sin(k t) dt \quad \forall n, i \in \{1 \dots 10\}$ vérifier que $a \approx \pi I_{10}$. <pre>def ps(x,j,k): return np.sin(j*x) * np.sin(k*x) x = np.linspace(np.pi, -np.pi) f = lambda j,k : integrate.trapz(x,ps(x,j+1,k+1)) f = np.vectorize(f) a = np.fromfunction(f, (10,10)) np.allclose(a,np.pi * np.identity(10)) # True</pre>

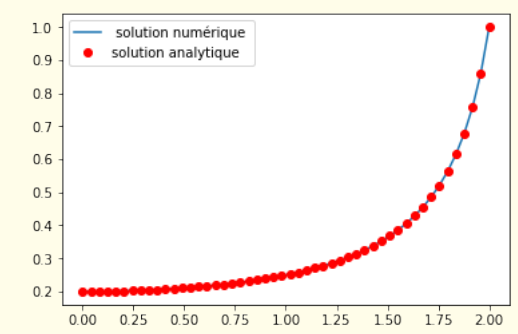


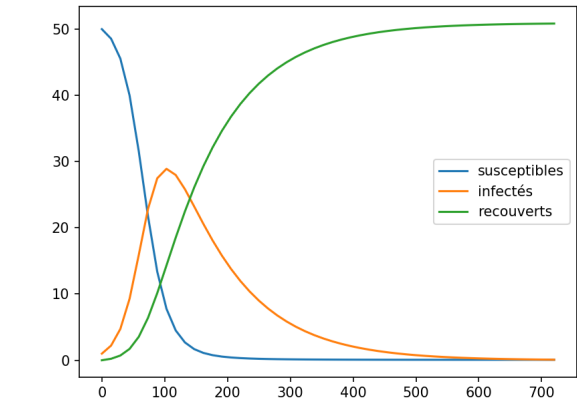
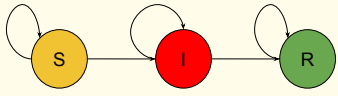
Description	Formule	Illustration graphique
Idée de base : Le domaine d'intégration $\Omega = [a, b]$ est discrétisé en un vecteur $x = (x_0 \ x_1 \ \dots \ x_{n-1})$ avec $\begin{cases} x_0 = a \\ x_{n-1} = b \\ x_{i+1} = x_i + \varepsilon \quad \varepsilon > 0 \end{cases}$	L'intégrale est découpée en une sommation d'intégrales sur les sous domaines $[x_i, x_{i+1}]$ $\int_a^b f(x)dx = \int_{x_0}^{x_{n-1}} f(x)dx = \sum_{i=0}^{n-2} \int_{x_i}^{x_{i+1}} f(x)dx$	
Méthode des trapèzes : $\begin{aligned} & \int_{x_i}^{x_{i+1}} f(x)dx \\ \approx & \text{Aire du trapèze délimité par } [(x_i, 0), (x_{i+1}, 0), (x_i, f(x_i)), (x_{i+1}, f(x_{i+1}))] \\ \approx & \varepsilon f(x_i) \frac{f(x_{i+1}) - f(x_i)}{2} \\ = & \varepsilon \frac{f(x_i) + f(x_{i+1})}{2} \end{aligned}$		<pre> n = 8 x = np.linspace(0, 2 * np.pi) xs = np.linspace(0, 2 * np.pi, n) y = np.sin(x) ys = np.sin(xs) plt.plot(x,y) plt.fill_between(xs,ys, facecolor = "b",\ edgecolor = "b", alpha = 0.5) plt.vlines(xs, 0, ys, colors="b") plt.show() </pre> 

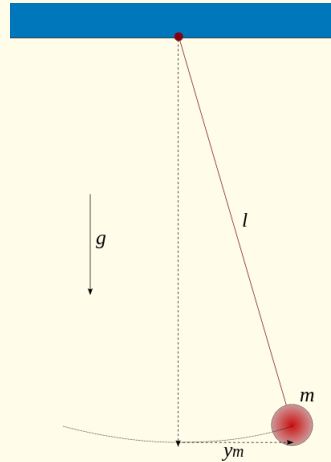
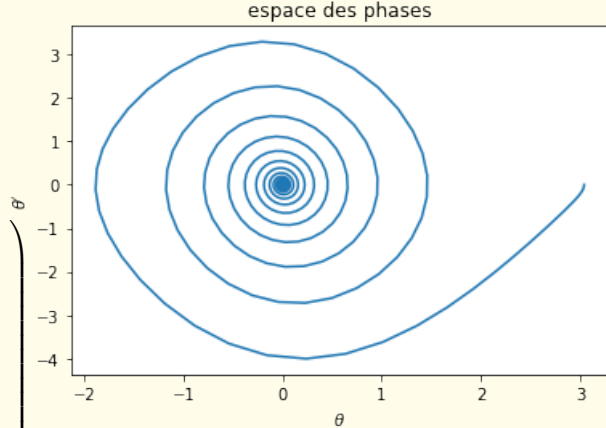
Description	Formule	Illustration graphique
Méthode des rectangles à gauche	$\int_{x_i}^{x_{i+1}} f(x)dx \approx \varepsilon f(x_i).$	<pre> x = np.linspace(0, 2 * np.pi) xs = np.linspace(0, 2 * np.pi, n) y = np.sin(x) ys = np.sin(xs) yv = ys[:-1] xm = np.empty(2+ (xs.size -2)*2) xm[:,2] = xs[:-1] xm[1::2] = xs[1:] ym = np.empty_like(xm) ym[:,2] = yv ym[1::2] = yv plt.plot(x,y) plt.fill_between(xm,ym, facecolor = "b",\ edgecolor = "b", alpha = 0.5) plt.vlines(xm, 0, ym, colors="b") plt.show() </pre> 

Description	Formule	Illustration graphique
Méthode des rectangles à droite	$\int_{x_i}^{x_{i+1}} f(x)dx \approx \varepsilon f(x_{i+1}).$	<pre> x = np.linspace(0, 2 * np.pi) xs = np.linspace(0, 2 * np.pi, n) y = np.sin(x) ys = np.sin(xs) yv = ys[1:] xm = np.empty(2+ (xs.size -2)*2) xm[:,2] = xs[:-1] xm[1::2] = xs[1:] ym = np.empty_like(xm) ym[:,2] = yv ym[1::2] = yv plt.plot(x,y) plt.fill_between(xm,ym, facecolor = "b", \ edgecolor = "b", alpha = 0.5) plt.vlines(xm, 0, ym, colors="b") plt.show() </pre> 

Description	Formule	Illustration graphique
Méthode du point milieu	$\int_{x_i}^{x_{i+1}} f(x)dx \approx \varepsilon f\left(\frac{x_i + x_{i+1}}{2}\right).$	<pre> x = np.linspace(0, 2 * np.pi) xs = np.linspace(0, 2 * np.pi, n) y = np.sin(x) ys = np.sin(xs) yv = (ys[:-1] + ys[1:])/2 xm = np.empty(2+ (xs.size -2)*2) xm[:,2] = xs[:-1] xm[1::2] = xs[1:] ym = np.empty_like(xm) ym[:,2] = yv ym[1::2] = yv plt.plot(x,y) plt.fill_between(xm,ym, facecolor = "b",\ edgecolor = "b", alpha = 0.5) plt.vlines(xm, 0, ym, colors="b") plt.show() </pre> 

Description	Syntaxe	Exemple(s) & Remarques
importation du module Intégration d'une équation différentielle ordinaire d'ordre 1 (EDO). Cette fonction prend en entrée : — <code>rel(y,t,...)</code> : une fonction avec aux moins deux paramètres (dans l'ordre : <code>y</code> la variable dépendante et <code>t</code> la variable libre). Si <code>rel</code> comporte d'autres paramètres (paramètres du modèle physique : masse, raideur, longueur, etc.). Ces paramètres supplémentaires doivent obligatoirement se situer à partir de la position 3. Leurs valeurs associées sont indiquées dans le <code>tuple args</code>. — <code>y0</code> est la condition initiale <code>y0 = y(t0)</code>. — <code>t</code> est une instance de la classe <code>np.ndarray</code> (à une dimension) contenant le domaine d'intégration de l'EDO (séquence de points ou la valeur de <code>y</code> doit être estimé). — <code>args</code> est un <code>tuple</code> (prenant par défaut <code>tuple()</code>) contient les valeurs des paramètres supplémentaires de <code>rel</code> s'il y en a. Pour $t = (t_0 \ t_1 \ \dots \ t_{n-1})$ la fonction renvoie un vecteur colonne $y^T = (y_0 \ y_1 \ \dots \ y_{n-1})$ tel que $y_i \approx y(t_i) \ \forall i \leq n-1$. L'idée principale derrière l'algorithme de résolution consiste à construire une suite récurrente. La forme de récurrence varie selon la méthode utilisée pour approcher l'intégrale suivante : $\int_{t_i}^{t_{i+1}} y'(t) dx = y(t_{i+1}) - y(t_i) = y_{i+1} - y_i$ $y_{i+1} = \underbrace{\int_{t_i}^{t_{i+1}} y'(t) dx}_{\approx \varepsilon y'(t_i) \text{ (méthode des rectangles à gauche)}} + y_i$ $\Rightarrow y_{i+1} \approx \varepsilon \text{ rel}(y_i, t_i) + y_i$ où $\varepsilon = t_{i+1} - t_i \text{ pas de discrétisation}$	<pre>from scipy import integrate ou bien import scipy.integrate as integrate</pre> <code>y = integrate.odeint(rel, y0, t, args=())</code>	Résolution numérique de l'équation $\begin{cases} y' = 2ty^2 \\ y(0) = 0.2 \end{cases} \quad \text{à intégrer sur } t = [0, 2]$ Comparer graphiquement avec la solution analytique sachant que $y(t) = \frac{1}{5 - t^2}$ <pre>import numpy as np from scipy import integrate import matplotlib.pyplot as plt rel = lambda y, t : 2*t*y**2 y0 = 0.2 t = np.linspace(0,2) y_num = integrate.odeint(rel,y0, t) y_ana = 1/(5-t**2) plt.plot(t, y_num, label=" solution numérique ") plt.plot(t, y_ana, "ro", label="solution analytique ") plt.legend() plt.show() y_num.shape #(50,1)</pre> 

Description	Syntaxe	Exemple(s) & Remarques
Intégration d'un système d'équations différentielles ordinaire d'ordre 1. Dans ce cas, le système est exprimé par une relation : $\mathbf{y}'(t) = \text{rel}(\mathbf{y}(t), t)$ où : — $\mathbf{y}'(t) = \begin{pmatrix} y'_0(t) \\ y'_1(t) \\ \vdots \\ y'_{n-1}(t) \end{pmatrix}$ est un vecteur (<code>np.ndarray</code>) contenant les dérivées premières des variables liées. — $\mathbf{y}(t) = \begin{pmatrix} y_0(t) \\ y_1(t) \\ \vdots \\ y_{n-1}(t) \end{pmatrix}$ est un vecteur (<code>np.ndarray</code>) contenant les variables liées (fonctions à déterminer). — t est un scalaire (la variable libre). Dans ce cas, la condition initiale $\mathbf{y}(t_0) = \begin{pmatrix} y_0(t_0) \\ y_1(t_0) \\ \vdots \\ y_{n-1}(t_0) \end{pmatrix}$ est exprimée par un vecteur (<code>np.ndarray</code>) contenant les valeurs respectives des fonctions au moment de départ t_0. Le principe de résolution est le même que pour une EDO d'ordre 1. Construire une suite de vecteurs $(\mathbf{y}_n)_{n \in \mathbb{N}}$ telle que : $\begin{cases} \mathbf{y}_0 = \mathbf{y}(t_0) \\ \mathbf{y}_{i+1} = \varepsilon \text{ rel}(\mathbf{y}_i, t_i) + \mathbf{y}_i \end{cases}$ Où ε est le pas de discrétisation.	<pre> y = integrate.odeint(rel, y0, t, args= ()) def rel(pop, t, beta, gamma): s,i, r = pop return np.array([-beta * s * i, beta*s*i - ↪ gamma * i, gamma * i]) pop0 = np.array([50, 1, 0]) beta = 0.0013 gamma = 0.01 t = np.linspace(0,24*30) s,i,r = integrate.odeint(rel, pop0, t, ↪ args=(beta, gamma)).T plt.plot(t,s, label="susceptibles") plt.plot(t, i, label="infectés") plt.plot(t, r, label="recouverts") plt.legend() plt.show() </pre> 	Diffusion d'une épidémie : La diffusion d'une épidémie dans une population d'individus peut être modélisée par un système de 3 équations différentielles d'ordre 1, avec 3 inconnues : — $s(t)$: le nombre d'individus susceptibles (ayant la possibilité de se contaminer) à l'instant t. — $i(t)$: le nombre d'individus qui sont infectés à l'instant t. — $r(t)$: le nombre d'individus recouverts à l'instant t. Ce modèle est dit modèle à compartiments, les individus se déplacent entre les 3 compartiments (S,I et R) selon certaines règles de transition. Le diagramme suivant illustre les états transitions du modèle SIR :  Les règles de transition sont régies par le système différentiel suivant : $\begin{cases} s'(t) = -\beta s(t)i(t) \\ i'(t) = \beta s(t)i(t) - \gamma i(t) \\ r'(t) = \gamma i(t) \end{cases}$ où : β est la probabilité d'infection d'un individu susceptible suite à un contact d'un individu infecté. γ et la probabilité de guérison (rétablissement) d'un individu infecté. Pour la simulation, nous choisissons $\beta = 0.0013$ $\gamma = 0.01$. Nous supposons qu'à l'instant $t = 0$, la population est répartie entre les 3 compartiments comme indiqué par le vecteur suivant $p_0^T = (50 \ 1 \ 0)$. Nous désirons simuler l'évolution de notre population pour la durée d'un mois où $\Delta t = 1$ heure.

Description	Syntaxe	Exemple(s) & Remarques
Résolution d'un EDO d'ordre $k > 1$: $y^{(k)} = \text{rel}(y^{(k-1)}, y^{(k-2)}, \dots, y', y, t) \quad (1)$ avec la condition initiale : $\begin{cases} y(t_0) = y_0(t_0) \\ y'(t_0) = y_1(t_0) \\ \vdots \\ y^{(k-1)}(t_0) = y_{k-1}(t_0) \end{cases}$ Il est possible de réécrire toute équation d'ordre $k > 1$, sous forme d'un système d'équations différentielles d'ordre 1, en introduisant $k-2$ inconnus supplémentaires afin de cacher les dérivées d'ordre > 1. En posant $\mathbf{y}(t) = \begin{pmatrix} y(t) \\ y'(t) \\ y''(t) \\ \vdots \\ y^{(j)}(t) \\ \vdots \\ y^{(k-1)}(t) \end{pmatrix} = \begin{pmatrix} y_0(t) \\ y_1(t) \\ y_2(t) \\ \vdots \\ y_j(t) \\ \vdots \\ y_{k-1}(t) \end{pmatrix}$ l'équation 1 est transformée en un SEDO d'ordre 1 décrit par : $\mathbf{y}'(t) = \begin{pmatrix} y_0'(t) \\ y_1'(t) \\ y_2'(t) \\ \vdots \\ y_j'(t) \\ \vdots \\ y_{k-1}'(t) \end{pmatrix} = \begin{pmatrix} y_0'(t) \\ y_1''(t) \\ y_2^{(3)}(t) \\ \vdots \\ y^{(j+1)}(t) \\ \vdots \\ y^{(k)}(t) \end{pmatrix} = \begin{pmatrix} y_1(t) \\ y_2(t) \\ y_3(t) \\ \vdots \\ y_{j+1}(t) \\ \vdots \\ \underbrace{\text{rel}(\mathbf{y}(t), t)}_{\text{décrite dans l'équation 1}} \end{pmatrix}$	<pre>import numpy as np from scipy import integrate from matplotlib import pyplot as plt def rel(Y,t, b, c): theta, omega = Y return np.array([omega, -b * omega - c * ↪ np.sin(theta)]) Y0 = np.array([np.pi - 0.1, 0]) t = np.linspace(0, 100, num = 1000) theta, omega = integrate.odeint(rel, Y0, t, args ↪ = (0.25, 5)).T plt.plot(theta, omega) plt.xlabel(r'\theta\$') plt.ylabel(r"\$\theta'\$") plt.title('espace des phases') plt.show()</pre>	Étude d'un pendule Le mouvement d'un pendule est régi par l'équation différentielle d'ordre 2 suivante : $\theta''(t) + b \theta'(t) + c \sin \theta(t) = 0 \quad (2)$  Avec b, c des paramètres du système physique. Afin de transformer en un système d'ordre 1, nous posons : $\theta'(t) = \omega(t)$ Ainsi $(2) \Rightarrow \begin{cases} \theta'(t) = \omega(t) \\ \omega'(t) = -b \omega(t) - c \sin \theta(t) \end{cases}$ Qui est SEDO d'ordre 1 . On s'intéresse à visualiser l'espace des phases du pendule (θ' en fonction de θ). Étant donnée la configuration suivante pour les paramètres : $\begin{cases} t = [0, 100] & \text{avec } \Delta t = 0.1 \\ \theta(0) = \pi - 0.1 \\ \omega(0) = \theta'(0) = 0 \\ b = 0.25 \\ c = 5 \end{cases}$
		

Description	Syntaxe	Exemple(s) & Remarques
Méthode des bissectrices (de dichotomie). Soit $f : \begin{matrix} D \subseteq \mathbb{R} \\ x \end{matrix} \longrightarrow \begin{matrix} \mathbb{R} \\ f(x) \end{matrix}$ Approcher le zéro de f avec une précision $\varepsilon > 0$ consiste à trouver $x \in D$ tel que $f(x) < \varepsilon$. La méthode des bissectrices se base sur le théorème des valeurs intermédiaires qui justifie une recherche dichotomique dans D afin de trouver x où $f(x) < \varepsilon$. Hypothèses : — f continue sur D. — $f(a) f(b) < 0$. L'idée de cette méthode est de construire 3 suites réelles : $(a_n)_{n \in \mathbb{N}} \text{ où } \begin{cases} a_0 = a \\ a_{n+1} = \begin{cases} a_n & \text{si } f(a_n) f(x_n) < 0 \\ x_n & \text{sinon} \end{cases} \end{cases}$ $(b_n)_{n \in \mathbb{N}} \text{ où } \begin{cases} b_0 = b \\ b_{n+1} = \begin{cases} b_n & \text{si } f(b_n) f(x_n) < 0 \\ x_n & \text{sinon} \end{cases} \end{cases}$ $(x_n)_{n \in \mathbb{N}} \text{ où } x_n = \frac{a_n + b_n}{2} \quad \forall n \geq 0$ Le triplet (a_i, b_i, x_i) est évalué $\forall i = 0 \dots k$: tel que $f(x_k) < \varepsilon$. Alors x_k est la solution retournée.	<pre>optimize.bisect(f, a, b, args=(), maxiter=100, ...)</pre> <pre>>>> optimize.bisect(f, 0, 2) 1.4142135623715149 >>> optimize.bisect(f, 0, 2, full_output=True) (1.4142135623715149, converged: True flag: 'converged' function_calls: 42 iterations: 40 root: 1.4142135623715149)</pre>	<pre>f = lambda x : x**2 - 2 #implémentation de la méthode des dichotomies def bisect(f, a, b, eps = 1e-4, max_iter = 50, ↪ full_output = False): """ full_output = True si on veut avoir toutes ↪ les valeurs de la suite x_n sinon on obtient uniquement la dernière ↪ valeur """ assert f(a) * f(b) <= 0 lst = [] for i in range(max_iter): x = (a+b)/2 lst.append(x) if abs(f(x)) < eps: return x if not full_output else lst elif f(a)*f(x) < 0: b = x else: a = x raise Exception("Nombre maximal d'itérations ↪ achevé") >>> bisect(f, 0, 2) 1.4141845703125 >>> bisect(f, 0, 2, full_output=True) [1.0, 1.5, 1.25, 1.375, 1.4375, 1.40625, 1.421875, 1.4140625, 1.41796875, 1.416015625, 1.4150390625, 1.41455078125, 1.414306640625, 1.4141845703125]</pre>

Description	Syntaxe	Exemple(s) & Remarques
Approcher le zéro d'une fonction avec la méthode de newton. $f : \begin{matrix} D \subseteq \mathbb{R} \\ x \end{matrix} \longrightarrow \mathbb{R} \begin{matrix} \\ \longmapsto f(x) \end{matrix}$ Hypothèses : — f dérivable au voisinage de x_0. — $f'(x) \neq 0 \quad \forall x \in V(x_0)$. Le principe de cette méthode consiste à approcher itérativement la courbe de $f _{\mathcal{C}_f}$ par la tangente à chaque point x_i. Ainsi, au lieu de résoudre directement pour $f(x) = 0$, on résout itérativement $y = f'(x_i)(x - x_i) + f(x_i) = 0$ pour obtenir x_{i+1}. et s'arrêter pour x_k tel que $f(x) < \varepsilon$. Ceci nous permet d'obtenir la suite récurrente suivante : $(x_n)_{n \in \mathbb{N}} \begin{cases} x_0 = \text{une valeur donnée} \\ x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)} \end{cases} \quad \forall n > 0$ Si l'expression analytique de $f'(x)$ n'est pas disponible, il est possible de l'approcher par la méthode des différences finies : $f'(x) \approx \frac{f(x + \varepsilon) - f(x)}{\varepsilon}$ pour $\varepsilon > 0$ suffisamment petit.	<pre> newton(func, x0, fprime=None, args=(), maxiter=50) >>> f = lambda x : x**2 - 2 >>> fprime = lambda x : 2*x >>> optimize.newton(f,2, fprime=fprime) 1.4142135623730951 >>> optimize.newton(f,[-2, 2], fprime=fprime) array([-1.41421356, 1.41421356]) </pre>	<pre> def newton(f, x0, fprime = None, max_iter = 50, eps = 1e-5): if fprime is None: fprime = lambda x : (f(x+eps) - f(x))/eps x = x0 for i in range(max_iter): x -= f(x)/fprime(x) if abs(f(x)) < eps: return x raise Exception(" Nombre maximal ↪ d'itérations dépassé") >>> newton(f, 2) 1.4142156955270093 >>> newton(f,-2) -1.4142156770245087 </pre>

Description	Syntaxe	Exemple(s) & Remarques
Approcher le zéro d'une fonction vectorielle (champ de vecteurs). $\mathbf{f}: D \subseteq \mathbb{R}^n \longrightarrow \mathbb{R}^n$ $\mathbf{x} = \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix} \mapsto \mathbf{f}(\mathbf{x}) = \begin{pmatrix} f_1(\mathbf{x}) \\ f_2(\mathbf{x}) \\ \vdots \\ f_n(\mathbf{x}) \end{pmatrix}$ Approcher le zéro de $\mathbf{f}$ consiste à trouver x_k tel que $\ \mathbf{f}(\mathbf{x})\ < \varepsilon$ pour un $\varepsilon > 0$ donné. Cette méthode nous permet aussi d'approcher la solution d'un système d'équations non linéaires. Cette méthode est une généralisation de la méthode de Newton expliquée ci-haut, la suite de vecteurs $(\mathbf{x}_n)_{n \in \mathbb{N}} \subset \mathbb{R}^n$ qui converge vers un zéro de $\mathbf{f}$ (étant donnée un point de départ bien choisi) est formulée ci-dessous : $(\mathbf{x}_n)_{n \in \mathbb{N}} \begin{cases} \mathbf{x}_0 \in \mathbb{R}^n \\ \mathbf{x}_{n+1} = \mathbf{x}_n - \mathbf{J}_f(\mathbf{x}_n)^{-1} \mathbf{f}(\mathbf{x}_n) \end{cases}$ où $\mathbf{J}_f(\mathbf{x}_n)$ est la matrice Jacobienne de $\mathbf{f}$ définie par : $\mathbf{J}_f(\mathbf{x}) = \mathbf{J}_f \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix} = \begin{bmatrix} \frac{\partial \mathbf{f}}{\partial x_1} & \cdots & \frac{\partial \mathbf{f}}{\partial x_n} \\ \frac{f_1}{\partial x_1} & \cdots & \frac{f_1}{\partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{f_n}{\partial x_1} & \cdots & \frac{f_n}{\partial x_n} \end{bmatrix}$ Écriture de la relation de récurrence sous forme d'un système linéaire (à résoudre itérativement) : $\mathbf{J}_f(\mathbf{x}_n)(\mathbf{x}_{n+1} - \mathbf{x}_n) = -\mathbf{f}(\mathbf{x}_n)$ Approximation de la dérivée partielle par la méthode des différences finies : $\frac{f_i}{\partial x_k}(\mathbf{x}) = \frac{f_i(x_1 \quad \dots \quad x_k + \varepsilon \quad \dots \quad x_n)^T - f_i(x_1 \quad \dots \quad x_k \quad \dots \quad x_n)^T}{\varepsilon}$	<code>fsolve(func, x0, args=(), fprime=None, ...)</code> Résolution dans $\mathbb{R}^2$ du système : $(E) \begin{cases} x_1 + \frac{1}{2}(x_1 - x_2)^3 = 1 \\ \frac{1}{2}(x_2 - x_1)^3 + x_1 = 0 \end{cases} \text{ avec } \mathbf{x}_0 = 0_{\mathbb{R}^2}$ <pre>>>> def f(x): x_1, x_2 = x #dépaquetage return np.array([x_1 + 0.5 * (x_1 - x_2)**3 ↪ - 1, 0.5 * (x_2 - x_1)**3 + x_2]) >>> x0 = np.zeros((2,)) >>> optimize.fsolve(f, x0) array([0.8411639, 0.1588361])</pre>	<pre>def Jacob(f,x, eps = 1e-4): n = x.size j = np.zeros((n,n)) for i in range(n): xp = x.copy() xp[i] += eps j[:,i] = (f(xp) - f(x))/eps return j def newton2(f, x0, jacob = None, eps = 1e-4, ↪ max_iter = 50): if jacob is None: jacob = lambda x: Jacob(f,x,eps) x = x0 for i in range(max_iter): p = np.linalg.solve(jacob(x), -f(x)) x += p if np.linalg.norm(f(x)) < eps: return x raise Exception(" Nombre maximal ↪ d'itérations dépassé !") >>> newton2(f, x0) array([0.84116979, 0.15883021])</pre>