Mémo Python

Recommandations générales

Édition : Utiliser un IDE (Integrated Development Environment) pour interpréter du Python, comme Pyzo ou Spyder. Pour nommer les fichiers, variables ou fonctions ne pas utiliser de caractères spéciaux : accents, espaces, etc. Utiliser des noms de variables et de fonctions qui facilitent la programmation, sans toutefois l'alourdir.
Penser à commenter les programmes à l'aide du symbole #.

Exécution : Il faut exécuter le fichier .py afin de pouvoir utiliser les fonctions qui apparaissent dedans souvent. Toute modification d'une fonction ne sera effective qu'après nouvelle exécution du fichier la contenant.

Deboguage : Pour la résolution d'une **erreur**, commencer par l'identifier !
La commande `print(x)` glissée dans une fonction ou un programme permet de renvoyer la valeur de la variable `x` et ainsi de mieux comprendre l'erreur du programme.
Dans la recherche d'une erreur affectant le résultat d'un algorithme, une **stratégie de deboguage** efficace consiste à décliner une série de tests élémentaires et progressifs qui permettent de valider les différentes étapes de l'algorithme indépendamment les unes des autres. Cette validation peut/doit être effectuée au fil de la programmation qui est alors elle-même progressive.

Attention : l'indentation en Python est très importante, une mauvaise indentation renverra des erreurs. Durant les TP, il est important d'aérer le code afin de faciliter la lecture (pour tout le monde), par exemple en mettant des espaces autour des opérations +, =, etc.

Environnement à utiliser

`from math import *`	importe toutes les fonctions mathématiques du module `math`
`import numpy as np`	importe le module qui manipule les vecteurs et matrices
`import matplotlib.pyplot as plt`	importe le module qui trace les courbes

Quelques fonctions et variables

`sqrt, log, exp, cos, sin, tan, cosh, sinh, tanh, abs, floor, round`
Il faut rajouter `np.` devant les fonctions afin de les appliquer à des vecteurs composante par composante.
`x.real` et `x.imag` renvoient la partie réelle et la partie imaginaire de l'élément `x`
`pi`, `e`, `1j` (pour le complexe i)

Booléens et opérateurs logiques

`True, False`	Booléens VRAI et FAUX, respectivement
`==, >, <, >=, <=, !=`	Tests $=, >, <, \geq, \leq, \neq$ respectivement
`and, or, not`	Opérateur logique ET, OU, NON respectivement

Structures de contrôle

`def f(x1, x2):`
`  ...`
`  return y1, y2`
	Fonction `f` d'arguments `x1` et `x2` et qui renvoient `y1` et `y2`.

`for i in range (a,b,p):` `  ...`	Boucle pour `i` allant de `a` à `b-1` avec le pas `p`. (`p = 1` par défaut)
`while condition :` `  ...`	Boucle tant que la `condition` est vraie (avec `condition` un booléen)

`if cond1 :`
`  ...`
`elif cond2 :`
`  ...`
`else :`
`  ...`
	Conditionnelle. On peut utiliser plusieurs `elif` successifs. `elif` est la contraction de `else if` du langage courant.

Définition de vecteurs et matrices

`np.array([2,5,7])`	Vecteur à une dimension $(2, 5, 7)$
`np.array([[1,2],[4,5]])`	Matrice $\begin{pmatrix} 1 & 2 \\ 4 & 5 \end{pmatrix}$
`np.arange(a,b,p)`	Vecteur des valeurs entières allant de `a` à `a+kp` avec le pas `p` (`p = 1` par défaut) et où `a+kp` est le plus proche de `b` par valeurs strictement inférieurs à `b`
`np.linspace(a,b,N)`	Vecteur de `N` valeurs équiréparties entre `a` et `b` (inclus)
`np.zeros(n)`	Vecteur nul de taille `n`
`np.zeros((m,n))`	Matrice nulle de taille `m`×`n`
`np.ones(n)`	Vecteur de `1` de taille `n`
`np.ones((m,n))`	Matrice de `1` de taille `m`×`n`
`np.eye(m,n)`	Matrice de type identité de taille `m`×`n`
`np.diag(X)`	Construction de la matrice carrée dont la (`0`-ième) diagonale est le vecteur `X`
`np.diag(X,i)`	Construction de la matrice carrée dont la `i`-ième diagonale est le vecteur `X`

Manipulation de vecteurs et matrices

`len(X)`	Longueur d'un vecteur (ou nombre de lignes d'une matrice `X`)
`np.shape(A)` ou `A.shape`	Taille `m`×`n` de la matrice `A`
`np.size(A)` ou `A.size`	Nombre d'éléments de la matrice `A`
`np.shape(A)[0]`	Nombre de lignes de la matrice `A`
`np.shape(A)[1]`	Nombre de colonnes de la matrice `A`
`A[a:b,A:B]`	Extraction de la matrice composée des lignes `a` à `b-1` et des colonnes `A` à `B-1`
`A[:,0]`	Extraction de la première colonne de `A`
`A[1:-1,0]`	Extraction de la première colonne de `A` excepté le premier et le dernier coefficient
`np.diag(A)`	Extraction du vecteur formant la diagonale de `A`
`np.diag(A,i)`	Extraction du vecteur formant la `i`-ième diagonale de `A`
`A+x`	Rajoute la valeur `x` à tous les éléments de `A`
`A#B`	où l'opérateur binaire # $\in \{+,*,/,**\}$, Opération composante par composante entre les `np.array` `A` et `B`
`A.dot(B)`	Produit matriciel de `A` avec `B` (si `A` et `B` sont des `np.array`)
`np.transpose(A)` ou `A.T`	Transposée de la matrice `A`

Opérations d'algèbre linéaire

`np.linalg.norm(X,p)`	$\|X\|_p$ (p$\in [1, +\infty[$ ou p= inf)
`np.linalg.norm(A,p)`	$\|\|A\|\|_p$ (norme subordonnée lorsque p$\in \{1,2,$'inf'$\}$, de Frobenius si p='fro')
`np.linalg.det(A)`	Déterminant de la matrice `A`
`np.linalg.matrix_rank(A)`	Rang de la matrice `A`
`np.trace(A)`	Trace de la matrice `A`
`np.linalg.inv(A)`	Inverse de la matrice `A`
`np.linalg.solve(A,b)`	Résolution d'un système linéaire $Ax = b$
`np.linalg.eigvals(A)`	Valeurs propres de la matrice `A` (avec multiplicité)
`np.linalg.eig(A)`	Couple contenant les valeurs propres et les vecteurs propres de la matrice `A` (vecteurs propres rangés en colonne)
`np.vdot(X,Y)`	Produit scalaire entre les vecteurs `X` et `Y`

Un exemple de programme

Remarque. L'exemple suivant utilise très peu de fonctionnalités intégrées au logiciel et est plutôt un pretexte pour illustrer les usages de syntaxe et les possibilités de vectorisation.

Un problème d'interpolation

Étant données différentes valeurs d'un entier $n \in \mathbb{N}$, on souhaite éprouver la qualité de l'interpolation de Lagrange de la fonction $f : x \mapsto (1+25x^2)^{-1}$, en $n+1$ points uniformément répartis dans l'intervalle $[-1, 1]$: $x_j = j/n$, pour $0 \leq j \leq n$.

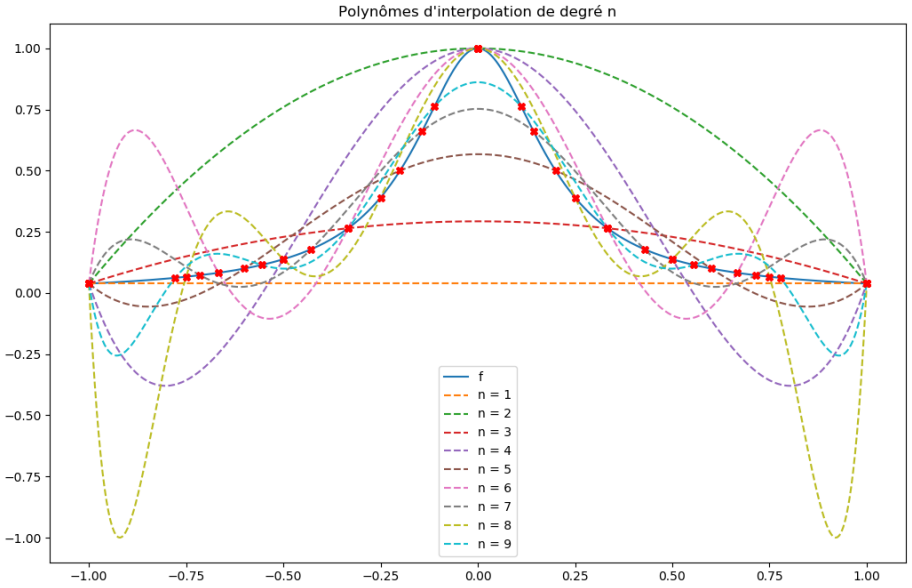
Le polynôme d'interpolation est calculé sous sa forme de Newton :

$$p(X) = f(x_0) + f[x_0, x_1](X - x_0) + \ldots + f[x_0, x_1, \ldots, x_n](X - x_0)\ldots(X - x_{n-1}),$$

pour laquelle les coefficients sont obtenus en calculant récursivement les différences divisées (généralisation du taux d'accroissement aux ordres supérieurs) :

$$f[x_0, x_1, \ldots, x_j] = \frac{f[x_1, x_2, \ldots, x_j] - f[x_0, x_1, \ldots, x_{j-1}]}{x_j - x_0}.$$

Le polynôme p est alors évalué par exponentiation rapide (algorithme de Horner).




```
1  from math import *
2  import numpy as np
3  import matplotlib.pyplot as plt
4
5  def f(x):
6      y = 1 / (1 + 25 * x**2)
7  return y
8
9
10 def diffdiv(x, y):
11     n = len(x) - 1
12     D = np.zeros((n+1, n+1))
13     D[:,0] = y
14     for j in range (1, n+1):
15         for i in range (n-j+1):
16             D[i,j] = (D[i+1,j-1] - D[i,j-1]) / (x[i+j] - x[i])
17     return D
18
19 def fastexp(x, D, t):
20     n = len(x) - 1
21     y = D[0, n] * np.ones(len(t))
22     for k in range (n-1, -1, -1):
23         y = (t-x[k]) * y + D[0, k]
24     return y
25
26
27 x = np.linspace(-1, 1, 400)
28 y = f(x)
29 plt.plot(x, y, label = "f")
30
31 for n in range (1, 10):
32     X = np.linspace(-1, 1, n+1)
33     Y = f(X)
34     D = diffdiv(X, Y)
35     y = fastexp(X, D, x)
36     plt.plot(x, y, '--', label = "n = " + str(n))
37     plt.plot(X, Y, 'Xr')
38 plt.legend(loc = "best")
39 plt.title("Polynomes d'interpolation de degre n")
40 plt.show()
```