Feuille de TP Python 06

Suites récurrentes, algorithmes numériques et équations différentielles

1 Introduction

L'objet de ce TP est de travailler le thème des suites récurrentes, de nombres, de vecteurs, de matrices ou d'objets plus généraux...

Il permet, d'une, part de passer en revue des thèmes classiques du calcul en y présentant quelques unes des méthodes les plus élémentaires et d'autre part d'aborder des thèmes de probabilité ou d'autres domaines.

1.1 Mathématiquement

Résumons mathématiquement la situation abstraite. On dispose d'un ensemble X d'objets (des nombres, des matrices,etc..) et d'une méthode de transformation de ces objets, *a.k.a* [1] une application $\phi : X \to X$.

En prenant $x_0 \in X$, le principe de récurrence affirme l'existence d'une unique suite d'objets de X, $(x_n)_{n\in\mathbb{N}}$ vérifiant

$$\forall n \in \mathbb{N}, x_{n+1} = \phi(x_n)$$

1.2 Informatiquement

C'est ce principe général que nous allons implémenter informatiquement. Les mathématiques nous permettent d'affirmer l'existence immédiate d'une infinité d'objets, ce que l'informatique ne permet pas. Il va donc falloir nous limiter à ne calculer qu'un nombre fini d'objets x_n.

Il faut décider pour chaque problème à traiter si on ne conserve que le dernier objet calculé ou toute la suite partielle calculée.

Une telle suite partielle sera placée dans une `list` au sens Python. Le tableau 4.3 en annexe récapitule quelques commandes, fonctions et méthodes pour gérer les listes.

Des codes Python typiques pour cela sont donc

```
#La fonction f effectuant la récurrence
# et l'objet initial X0 préexistent
#n donné, on calcule Xn
Xn=X0
for i in range(n): #n tours de boucle
    Xn=f(Xn)
#A ce point Xn est le terme d'indice n
```

```
#La fonction f effectuant la récurrence
# et l'objet initial X0 préexistent
#n donné, on calcule X=[X0,..,Xn]
X=[X0] #La liste qui contient le résultat
for i in range(n): #n tours de boucle
    Xi=f(X[-1]) #X[-1] dernier élément de  X
    X.append(Xi)
#A ce point X est une liste indicée de 0 à n
```

2 Exemples élémentaires

2.1 L'algorithme de Babylone

L'algorithme de Babylone est une méthode employée pour calculer « à la main » des racines carrées. Soit $A = a^2$ un nombre dont on recherche la racine carrée. L'algorithme de Babylone est la récurrence

$$x_0 \text{ donné}, \neq 0, \forall n \in \mathbb{N}, x_{n+1} = \frac{1}{2}\left(x_n + \frac{A}{x_n}\right).$$

On démontre (on le fera en TD) que si A un nombre réel strictement positif, si $x_0 \in \mathbb{R}$, $x_0 \neq 0$ alors, lorsque $n \to +\infty$,

$$x_n \to \begin{cases} +|a| & \text{si } x_0 > 0 \\ -|a| & \text{si } x_0 < 0 \end{cases}$$

1. "also known as"

1. Programmer cet algorithme dans une fonction `RacineCarree(A,x0=1,n=10)` pour calculer une racine carrée d'un nombre pour un nombre $A = \texttt{A}$ donné à l'avance en calculant x_n, le terme d'indice $n = \texttt{n}$ de la suite décrite précédemment partant de $x_0 = \texttt{x0}$. Tester sur un certain nombre d'exemples et comparer les réultats avec ceux de la fonction Numpy/Python `np.sqrt`.
2. Ecrire une une fonction `RacineCarreeSuite(A,x0=1,n=10)` retournant en liste les termes $x_0, \ldots, x_n$, $n = \texttt{n}$, de la suite décrite précédemment partant de $x_0 = \texttt{x0}$ pour un nombre $A = \texttt{A}$ donné en paramètre. Partant de $A = -1$ et $x_0 = 1.1111111$, tracer le graphe $n \mapsto x_n$ de cette suite (les indices n en abscisses, les termes de la suite x_n en ordonnée. Y-a-t'il une convergence à conjecturer ?
3. (Ouvert) Que se passe-t-il si l'on part de x_0 nombre complexe non réel, par exemple $x_0 = 1 + 3.i$? Rappel : en Python cette affectation se fait par `x0=1+3j`

2.2 Une suite intervenant en dynamique des populations

Soit $p \in]0,1[$, $d \in \mathbb{N}^*$. On considère la suite récurrente donnée par

$$p_0 \in]0,1[, \forall n \in \mathbb{N}, p_{n+1} = (1 - p + p.p_n)^d$$

Cette suite récurrente est liée à la probabilité de disparition d'une population dont la régénération au jour n se déroule de la façon suivante :

chaque femelle f engendre, indépendamment des autres, $X_{n,f}$ rejetons femelles suivant une loi binomiale ($\mathscr{B}(d,p)$) avant de mourir.

1. Ecrire une fonction Python `ProbaDisparition(n,p0=0,p=0.5,d=2)` calculant/retournant la liste des termes $p_0, \ldots, p_n$ calculés suivant cette récurrence.
2. Ecrire une fonction Python `Graphe(p=0.5,d=2)` traçant le graphe de la fonction itérée ainsi la droite d'équation $y = x$ en reprère orthonormé.
3. Effectuer quelques expériences en graphant la suite par dessus le graphe précédent. Conjecturer la convergence de cette suite et la position de la limite par rapport au point fixe 1.

   Indication:Distinguer les cas $p.d > 1$ (auquel cas la probabilité de disparition tend vers une limite < 1) $p.d = 1$ (auquel cas la limite est 1).
4. (Facultatif, à faire quand tout le reste de cette feuille est fini).
   (a) Ecrire une fonction effectuant la simulation du modèle probabiliste proposé sur N générations en partant d'une population ayant une femelle de départ et retournant `True` si la population a disparu avant la génération N, `False` sinon.
   (b) En effectuant un grand nombre de tirages, évaluer la probabilité de disparition et comparer avec la limite de la suite (p_n) précédemment obtenue.

2.3 Travail demandé

Traiter l'un des deux exemples proposés (Questions 1 (fichier `babylone.py`) ou 2. (fichier `proba-extinction.py`)).

3 Recherche de solutions d'équations

Utiliser le script `solutionfxy.py`.

Décompressez l'archive `.zip` associée à ce TD, ouvrez le fichier `python/solutionfxy.py` dans Spyder ou Pyzo/IEP. Sous Pyzo/IEP, faites de `python/` votre répertoire de travail en exécutant le script avec `SHIFT-CTRL-E`

On s'intéresse au problème suivant : Etant donnés une fonction f réelle de variable réelle, définie par exemple par une formule, un nombre $y \in \mathbb{R}$, on cherche à trouver **une** [2] solution de l'équation $f(x) = y$ d'inconnue $x \in \mathbb{R}$.

On propose deux méthodes : la méthode par dichotomie et la méthode NEWTON. Cette dernière est plus complexe mais beaucoup plus rapide.

On va écrire des méthodes génériques, *i.e.* la fonction f fait partie des arguments de la méthode lors de son appel. On va par ailleurs placer ces méthodes dans le module `solutionfxy.py`.

2. toutes, c'est beaucoup plus compliqué

3.1 Dichotomie

La méthode de dichotomie pour résoudre l'équation $f(x) = y$ d'inconnue x se décrit de la manière suivante.
On construit deux suites $(a_n)_{n\in\mathbb{N}}$ et $(b_n)_{n\in\mathbb{N}}$ par récurrence.

1. a_0 et b_0 sont donnés au départ et vérifient $(f(a_0)-y)(f(b_0)-y) \leq 0$,
2. connaissant a_n et b_n, on calcule leur moyenne c et
   (a) si $(f(c)-y)(f(b_n)-y) \leq 0$, on pose $a_{n+1} = c$, $b_{n+1} = b_n$,
   (b) si $(f(a_n)-y)(f(c)-y) \leq 0$, on pose $a_{n+1} = a_n$, $b_{n+1} = c$,

Théorème 1. *Si*

1. *Les nombres $a, b \in \mathbb{R}$, $a < b$ sont donnés ;*
2. *La fonction $f : [a,b] \to \mathbb{R}$ est continue sur l'intervalle $[a,b]$;*
3. *Le nombre $y \in \mathbb{R}$ est entre $f(a)$ et $f(b)$,*

alors

1. *les suites $(a_n)_{n\in\mathbb{N}}$ et $(b_n)_{n\in\mathbb{N}}$ construites par la méthode de dichotomie convergent vers une limite commune que l'on note c. On a*
$$\forall n \in \mathbb{N},\ a_n \leq c \leq b_n \text{ et } |b_n - a_n| = \frac{|b-a|}{2^n}$$
2. *Le nombre c est une solution de l'équation $f(x) = y$ d'inconnnue $x \in [a,b]$.*

3.2 Méthode de NEWTON

La méthode de NEWTON de résolution d'équation du type $f(x) = 0$ se décrit de la manière suivante.
On suppose que

1. il existe une solution de cette équation, notée x_∞,
2. La fonction f est de classe $\mathscr{C}^1$ sur un voisinage de x_∞,
3. f' ne s'annule pas sur un voisinage de x_∞.

On prend x_0 « suffisamment proche » de x_∞ et on construit par récurrence la suite $(x_n)_{n\in\mathbb{N}}$ par

$$\forall n \in \mathbb{N},\ x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

Géométriquement, cela revient à prendre pour x_{n+1} l'abscisse du point d'intersection entre l'axe des abscisses et la tangente au graphe de f en x_n. On a plus précisémment le théorème de convergence (HP)

Théorème 2. *Dans les conditions exposées précédemment, il existe $1 > \varepsilon > 0$, $K > 0$ (dépendants de f), tels que si*

1. *$K.|x_0 - x_\infty| < \varepsilon$*

alors

1. *la suite $(x_n)_{n\in\mathbb{N}}$ est bien définie, elle converge vers x_∞,*
2. *avec l'estimation*
$$\forall n \in \mathbb{N},\ K.|x_{n+1} - x_\infty| \leq (K.|x_n - x_\infty|)^2 \leq (K.|x_0 - x_\infty|)^{2^{n+1}}$$

3.3 Travail demandé

1. Dichotomie
   (a) Compléter le fichier `solutionfxy.py` donné en annexe en écrivant le corps de la fonction `dichotomie` suivant les spécifications indiquées. Executer le fichier pour faire les tests prévus.
   (b) On considère, pour chaque $n \in \mathbb{N}$, $n \geq 4$, l'équation $x^n - \frac{n}{2}x^2 + 1 = 0$ d'inconnue $x \in [0,1]$. On admet (c'est un exercice assez facile) que cette équation admet une unique solution α_n qui vérifie de plus $\sqrt{2}.n^{-\frac{1}{2}} \leq \alpha_n \leq 2.n^{-\frac{1}{2}}$.
   Ecrire, dans le fichier `td06.py`, un programme résolvant cette équation pour $4 \leq n \leq N = 100$ en utilisant la fonction `dichotomie` définie dans le module `solutionfxy.py`. On veut que le programme trace le graphe de la suite des nombres $(\alpha_n)_{4\leq n\leq N}$. Executez ! *Faire en sorte que les résultats des tests du module* `solutionfxy.py` *n'apparaissent plus lors de l'exécution de* `td06.py` *et reexécuter.*
2. NEWTON.
   (a) Corriger l'algorithme théorique pour qu'il résolve une équation du type $f(x) = y$ et l'implémenter dans `newton1d`. Décommenter les lignes de tests pour `newton1d` et exécuter. Les résultats sont-ils cohérents avec ceux obtenus par dichotomie ?
   (b) Reprendre le fichier `td06.py` et compléter le programme pour qu'il résolve aussi l'exercice en utilisant la fonction `newton1d` définie dans le module `solutionfxy.py`

4 Suites récurrentes et EDO

4.1 Problème de CAUCHY et schéma d'EULER

Soit $d \in \mathbb{N}^*$ (d comme *dimension*). Une équation différentielle ordinaire (scalaire pour $d = 1$, vectorielle pour $d \geq 1$) du premier ordre se présente sous la forme

$$\frac{dY}{dt} = F(Y,t) \qquad (*)$$

et résoudre un problème de CAUCHY associé à l'équation (*) c'est

1. se donner un « instant initial » t_0, une valeur initiale Y_0, dans $\mathbb{R}^d$,
2. déterminer un intervalle I de $\mathbb{R}$ non trivial contenant t_0 (l'intervalle de résolution)
3. déterminer la(les) fonction(s) $Y : t \in I \mapsto Y(t)$, de classe $\mathscr{C}^1(I)$, à valeurs dans $\mathbb{R}^d$, telle(s) que
   (a) $Y(t_0) = Y_0$
   (b)
   $$\forall t \in I, Y'(t) = F(Y(t),t)$$

Une telle EDO est l'analogue continu d'une suite récurrente, ce qui peut se comprendre du point de vue de la physique.

1. On pose un réel positif, Δt, un « petit accroissement » de la variable t, le *pas* du schéma d'EULER. On peut par exemple prendre $\Delta t = \frac{1}{N}$ où N est un entier naturel.
2. On obtient alors une suite de points $(t_k)_{k\in\mathbb{Z}}$ donnée par $t_k = t_0 + k.\Delta t$.
3. l'EDO (*) et le problème de CAUCHY associé à (*) et une valeur initiale $Y_0 \in \mathbb{R}^d$ ont pour analogue la détermination d'un segment K d'entiers relatifs contenant 0, d'une suite $(Y_k)_{k\in K}$ telle que
   $$\forall k \in K, k+1 \in K \Rightarrow Y_{k+1} = Y_k + \Delta t.F(Y_k,t_k)$$

Appliquer le schéma d'EULER pour résoudre l'EDO (*) et le problème de CAUCHY associé, c'est calculer une suite vérifiant cette récurrence et espérer que Y_k est une « bonne approximation » de $Y(t_k)$.

Une étape de l'algorithme d'EULER, prolongeant d'un pas des listes `Y` et `t` déjà construites se traduit donc par

```
Y0=Y[-1] ; t0=t[-1]
Y1=Y0+dt*F(Y0,t0) ; t1=t0+dt
Y.append(Y1) ; t.append(t1)
```

Il s'agit évidemment d'itérer cette construction via une boucle `while` ou `for`.

Une autre façon de coder ce schéma de résolution approximative est de spécifier la fonction F, le point de départ Y_0 et la plage de temps discrétisée voulue par un `ndarray t` (obtenu par exemple via `linspace`) :

```
N = t.shape[0] #Le nombre de points
Y = np.zeros( N )
Y[0] = Y0
for i in range(1,N):
    Y[i] = Y[i-1]+(t[i]-t[i-1])*F(Y[i-1],t[i-1])
```

4.2 Travail demandé

Le script `edo.py` contient un canevas de fonction `my_odeint` (que vous devez compléter) de résolution d'EDO dont la signature [3] est similaire à celle de la fonction `odeint` du module `scipy.integrate`, a savoir :

```
F: func/A, y0: ndarray/A, t: ndarray/B -> ndarray/AB
```

où

- `ndarray/A` est un `ndarray` unidimensionnel, de forme (`np.shape`) `(d,)` où d est la dimension de l'espace d'arrivée de la fonction inconnue Y ;
- `func/A` est une fonction Python de signature `y: ndarray/A, t: float -> ndarray/A` ;

1. Compléter les fonctions `Euler` et `my_odeint` suivant les docstrings.
2. Exécuter le script. Quelle équation différentielle est-elle résolue approximativment ? avec quelle condition initiale ?
3. Résoudre ce problème de CAUCHY à la main, coder la fonction solution dans une fonction Python `y_vraie(t)` et tracer le graphe cette fonction sur le même graphique que l'approximation. Commentaires ?
4. Pour la physique : observer le bloc où la fonction `odeint` du module `scipy.integrate` est utilisée. Commentaires

3. La signature d'une fonction c'est la spécification, en général typée, voire nommée et typée, de ses arguments (entrée) et de ses valeurs de retour. P.ex. :
   - la fonction `np.exp` du module `numpy` (importé « `as np` ») a pour signature `ndarray -> ndarray` pour signifier qu'elle prend en entrée un `ndarray` (tableau numpy) et qu'elle retourne un `ndarray`.
   - la fonction `nr.chi2.pdf` du module `numpy.random` (importé « `as nr` ») a pour signature `x:ndarray, d:int -> ndarray` pour signifier qu'elle prend en entrée un couple `ndarray x`, `int d` et qu'elle retourne un `ndarray`.

4.3 Listes

`L=[]` ou `L=list()`	fabrique une liste vide à remplir
`L=[1.0,'aa']`	fabrique une liste avec un préremplissage
`len(L)`	donne le nombre d'éléments de la liste, ils sont indicés de 0 à `len(L)-1`
`L[i]`	donne la valeur de l'élément de `L` d'indice i si $0 \leq i \leq$`len(L)-1`
`M=L[i:j]`	extrait une sous-liste M dont les éléments ceux de `L` sont indicés de i à $j-1$
`L[i:j]=M`	remplace la sous-liste des éléments de `L` indicés de i à $j-1$ par la liste `M`
`M=L`	Les deux étiquettes `M` et `L` sont sur la même liste en mémoire
`M=L[:]` ou `M = L.copy()`	créé une copie de `L` et lui affecte l'étiquette `M`
`L[-1]`	donne le dernier élément de la liste
`L.append(obj)`	ajoute l'objet `obj` en fin de liste
`L.extend(M)`	fusionne les listes `L` et `M` en plaçant `M` à la fin de `L`
`L.insert(index,obj)`	insère l'objet `obj` avant l'indice i, décale le reste de `L`
`L[i:i] = M`	insère la liste `M` à l'indice i de la liste `L`, décale le reste de `L`
`L.pop()`	efface le dernier élément de la liste `L` et le retourne
`L.pop(i)`	efface l'élément i de la liste `L` et le retourne
`L.reverse()`	inverse *en place* l'ordre des éléments de la liste
`L.sort()`	trie *en place* les éléments de la liste, pourvu que l'on puisse comparer les éléments
`M=sorted(L)`	`M` devient une copie triée de la liste `L`
`V=np.array(L)` ou `V=np.asarray(L)`	`V` devient un vecteur `numpy` composé avec les éléments de la liste de nombres `L`. Si `L` est une liste de listes de nombres, compose une matrice `numpy`.

`for obj in L :`
 	
Exemple :
`for k in L:`
`  print(k**2)`
	boucle sur les éléments de `L`, ne modifie pas `L`. Modifier `L` dans la boucle peut avoir des conséquences inattendues.
	Si on prend `L=[1,2,4]` affiche successivement `1`, `4`, `16`.

`M = [expression(obj) for obj in L]`
 	
Exemple :
`M=[k+1 for k in L]`
	fabrique une liste en bouclant sur les éléments de `L` et en leur appliquant `expression()`, ne modifie pas `L`.
	Si on prend `L=[1,2,4]` fabrique la nouvelle liste `[2,3,5]` et l'affecte à `M`.

`M = [expr(obj) for obj in L if cond(obj)]`
 	
Exemple :
`M=[k**2 for k in L if k>1]`
	fabrique une liste en bouclant sur les éléments de `L`, ne gardant que ceux vérifiant `cond()` et en leur appliquant `expr()`, ne modifie pas `L`.
	Si on prend `L=[1,2,4]` fabrique la nouvelle liste `[4,16]` et l'affecte à `M`.