TP 12 : étude des suites avec PYLAB

N.B. On importera `pylab` avec `import pylab as pl` ce qui permet d'avoir à la fois les fonctions de `numpy` et de `matplotlib.pyplot`. Voir l'appendice pour des commandes graphiques utiles.

1 Révisions et compléments sur les tracés de graphes de fonctions, recherches de zéros

Pour les commandes que vous ne connaissez pas sur plot, voir l'appendice du TP

a) Tracez le graphe de la fonction tan sur $]-\pi/2, 5\pi/2[$.

Rappel : on doit définir un tableau `X` d'abscisse, puis un tableau `Y=pl.tan(X)` puis faire `pl.plot(X,Y)` puis enfin `pl.show()`.

N.B. Bien vous souvenir que `pl.tan` s'applique directement à un tableau (ou liste) de `x` et renvoie un tableau de valeurs obtenues en appliquant tangente entrée par entrée.

Si vous êtes déçu du résultat, recadrez votre figure pour éviter que l'axe des ordonnées ne monte trop haut.

Remarque : PYLAB va joindre les points à travers les discontinuités, comment éviter ce phénomène ?

b) Tracez sur la même figure la première bissectrice, d'une autre couleur.

c) Déterminez graphiquement les coordonnées des points d'intersection entre les deux courbes dans $[0, 2\pi]$ (la figure PYLAB donne les coordonnées du curseur).

d) Déterminez numériquement ces points d'intersections avec la fonction `fsolve` du module `scipy.optimize` et rajoutez les sur la figure avec le symbole `o` d'une autre couleur.

N.B. La fonction `fsolve` de ce module s'utilise comme suit : `fsolve(func,x0)` cherche un zéro de la fonction `func` au voisinage d'une valeur `x0`. Nous reviendrons sur les méthodes de recherche de zéros de fonctions : nous en avons déjà rencontrée deux au T.P. 5, lesquelles ?

e) **Cas d'une fonction définie par cas :** Tracer le graphe de la fonction $f : x \mapsto \begin{cases} x^2, \text{ si } \lfloor x \rfloor \text{est paire,} \\ 1 - x^3 \text{ sinon} \end{cases}$ pour $x \in [-10, 10]$.

2 Etude de familles de fonctions et de suites définies implicitement

2.1 La suites des polynômes de Taylor de l'exponentielle : programmation et tracé d'une famille de fonctions

On note $f_n : x \mapsto \sum_{k=0}^{n} \frac{x^k}{k!}$. On démontrera bientôt que pour chaque x, $f_n(x) \underset{n\to+\infty}{\longrightarrow} e^x$.

a) Ecrire une fonction `Taylor_exp(x,n)` qui prend comme arguments un entier `n` et un flottant `x` et renvoie la valeur de $f_n(x)$.

Essayer de garder les bonnes habitudes de construction d'une boucle (calculs à ne pas refaire à chaque fois !)

b) Tracer une des fonctions obtenues par exemple pour $n = 5$, en l'appliquant à un tableau `x` représentant des abscisses entre -5 et 5.

N.B. Normalement, toutes les fonctions que vous avez utilisées pour le calcul de `Taylor_exp(x,n)` peuvent s'appliquer avec comme entrée un tableau `x`.

2.2 La suites des polynômes de Taylor de l'exponentielle : tracés, zéros

a) Tracez sur une même figure les graphes des fonctions f_n pour $n \in [\![1, 20]\!]$ et $x \in [-5, 5]$.

b) Recadrez, zoomez, pour voir les éventuels zéros des f_n.

c) Exercice de mathématique (les maths sont à faire à la maison) :

i) Démontrer que les f_{2n} ne s'annulent pas sur $\mathbb{R}$ et que les f_{2n+1} ont un unique zéro dans $\mathbb{R}$.

ii) On note x_n l'unique zéro de f_{2n+1}. A l'aide de PYTHON faire une conjecture sur le comportement de la suite (x_n) (on pourra utiliser la fonction `fsolve` de `scipy.optimitze` pour le calcul approché des zéros). Puis démontrer cette conjecture en faisant des mathématiques.

3 Suites récurrentes $u_{n+1} = f(u_n)$: première partie

3.1 Représentation des itérées d'une fonction $f : x \mapsto x^2 + c$

Notation : Pour une fonction $f : \mathbb{R} \to \mathbb{R}$, et un entier $n \in \mathbb{N}^*$, on note $f^{\circ n} = \underbrace{f \circ \cdots \circ f}_{n \text{ fois } f}$.

On a vu au chapitre sur les suites récurrentes que même pour une simple fonction polynomiale du second degré f, les suites $u_{n+1} = f(u_n)$ donnent des comportements assez riches... en fait la richesse (et la complexité) des ces suites va bien au delà des exemples que nous avons étudiés.

Une première façon de comprendre cette complexité est de tracer à quoi ressemble $f^{\circ n}$.

Exercice : Tracez le graphe de $f^{\circ n}$ pour $f : x \mapsto x^2 + c$ pour $c = -1,39$ et $n = 6, 10, 15$, notamment pour $x \in [-1, 1]$. Remarque : quel est le degré de cette fonction polynomiale ?

3.2 Histoire d'un germe

Pour chaque point x_0 (appelé *germe*) on considère la suite (x_n) définie par ce x_0 et $\forall\, n \in \mathbb{N}$, $x_{n+1} = f(x_n)$.

Avec la notation du paragraphe précédent, on a $x_n = f^{\circ n}(x_0)$.

Pour avoir une représentation graphique de la suite (x_n) on va tracer les points (n, x_n) pour $n \in \mathbb{N}$ (bien sûr en fait pour une partie de $\mathbb{N}$!). Ce graphe sera appelé *l'histoire du point* x_0.

On choisit ici $x_0 = -1$ et toujours $f : x \mapsto x^2 + c$. Tracez les points (n, x_n) dans les différents cas suivants :

a) si $c = -1$. Justifier le résultat visible sur le tracé.

b) si $c = -1,3$. Commentez le résultat. Regardez des valeurs numériques plus précises pour préciser votre analyse : a-t-on oui ou non un comportement périodique A.P.C.R. ?

c) si $c = -1,8$. Commentez ?

3.3 Etude mathématique précise : des calculs faciles au début

On considère toujours $f_c : x \mapsto x^2 + c$.

a) Déterminer la CNS sur c pour que f_c ait un point fixe (réel !). Bien sûr cela se fait à la main, avec papier crayon, puisqu'il s'agit de pouvoir résoudre une équation du second degré. On suppose désormais cette condition réalisée.

b) Déterminer la CNS sur c pour qu'un de ces deux points fixes soit *attractif*.

c) Déterminer la CNS sur c pour qu'en outre f_c ait des points périodiques de période 2 i.e. il existe des $x \in \mathbb{R}$ tels que $(f \circ f)(x) = x$ et $f(x) \neq x$.

N.B. Cette question peut encore se faire avec papier crayon comme un exercice de mathématiques. Profitons en néanmoins pour introduire un module de calcul *formel* en PYTHON appelé `sympy`.

3.4 Utilisation d'un module de calcul formel : `sympy`

Si `sympy` n'est pas installé sur vos machines, passez ce paragraphe !

Dans un nouvel shell (pour éviter les conflits avec numpy), essayer :

```
a) from sympy import *
x=Symbol('x')
c=Symbol('c')
solution=solve(x**2-x+c,x)
x0,x1=solution
```

Qu'a-t-on obtenu ?

```
b) y=x**2+c
yp=diff(y,x)
```

Même question ?

```
c) f=x**2+c
f2=compose(f,f)
```

En déduire une façon de faire le calcul du 3.3 c) avec une machine.

d) On dira qu'un point périodique de période deux x est *attractif* si, et seulement si, $|(f \circ f)'(x)| < 1$.

Parmi les deux points périodiques de période deux trouvés au c), qui ne sont pas des points fixes, on pourrait essayer de déterminer ceux qui sont *attractifs* à l'aide de `sympy`. Cela demande déjà des calculs compliqués. En réalité, on va plutôt revenir au *calcul numérique* car en pratique les seuls points fixes que l'on *voit* sont les attractifs comme on va l'expérimenter maintenant.

3.5 Suite de l'étude par expérimentation numérique : la figure de la cascade

Pour différentes valeurs de $c \in [-2, 1/4]$ (disons 10, puis 50 puis 100), on va :

a) Calculer tous les termes de la suite (u_n) ayant comme valeur initiale $u_0 = 0$ (important) et telle que $u_{n+1} = f_c(u_n)$, pour $n \in [\![1, 100]\!]$.

b) Tracer les points $(u(n), c)$ pour $n \in [\![50, 100]\!]$ dans un cadre avec des abscisses dans $[-2, 2]$ et des ordonnées $c \in [-2, 1/4]$. (Le fait de commencer à 50 permet de ne pas tenir compte des premiers termes qui sont ce qu'on pourrait appeler le *régime transitoire*).

On utilisera `plot` avec l'argument `'k.'` pour avoir des points noirs (lettre `k`) ayant des formes de points non reliés `'.'` et le keyword argument `markersize=1` pour que ces points soient petits.

Autrement dit : pour chaque valeur de c en ordonnée, on trace 50 points sur la droite horizontale d'ordonnée c qui sont 50 valeurs de la suite (u_n) pour cette valeur de c.

c) Comment interpréter la zone du graphe obtenu pour $c > -0,75$?

d) Même question pour $c \in]-5/4, -3/4[$?

e) A partir du graphe précédent, en zoomant, en rajoutant éventuellement des valeurs de c déterminer à partir de quelle valeur de c la suite a 4 points périodiques attractifs.

4 La même suite récurrente dans le monde complexe

On considère la fonction définie par la même formule $f : z \in \mathbb{C} \mapsto z^2 + c \in \mathbb{C}$ avec $c \in \mathbb{C}$.

L'itération de cette fonction permet de mettre en évidence des fractales célèbres. Pour cela, on va d'abord introduire un outil graphique commode :

4.1 Préliminaire : l'outil `pl.imshow(T)`

La fonction `imshow` prend en argument un tableau bidimensionnel dont les entrées peuvent être des entiers et va afficher une image découpée en zones où la couleur de chaque zone correspondra à une entrée de la matrice. Avec des `0` et des `1` dans la matrice, on aura une image bicolore, comme dans l'exemple suivant :

```
import pylab as pl
T=pl.array([[0,0,0,1],[1,0,1,0],[0,0,0,1],[1,0,0,1]])
pl.imshow(T)
pl.show()
```

Avec des nombres plus variés, essayez, la fonction `pl.colorbar()` avec le `pl.show()` permettra de savoir à quel nombre correspond chaque couleur suivant une graduation « continue » :

```
 T=pl.array([[0,3,0,1],[1,4,1,0],[0,0,0,1],[1,0,8,12]])
pl.imshow(T)
pl.colorbar()
pl.show()
```

Plus la matrice sera grande, plus la zone correspondant à chaque entrée de la matrice sera petite : on se rapproche alors d'une correspondance entre entrée de la matrice `T` et couleur d'un pixel.

4.2 Ensembles de Julia

On fixe un $c \in \mathbb{C}$ et on cherche à dessiner dans le plan complexe l'ensemble des $z_0 \in \mathbb{C}$ tels que la suite (z_n) définie par ce germe z_0 et $\forall\, n \in \mathbb{N}, z_{n+1} = z_n^2 + c$ reste bornée.

Pour $c = 0$, il s'agit bien sûr du disque unité fermé, mais si $c \neq 0$, l'ensemble correspondant, noté J_c, prend des formes étonnantes.

Avec des maths : on peut montrer que si pour un n_0, $|z_{n_0}| > |c| + 1$ alors la suite est non bornée.

Pour le dessiner :

On voudrait représenter pour chaque valeur de $z_0 = a + ib$ avec $a \in [-3, 3]$ et $b \in [-3, 3]$, le point d'affixe z_0 avec une couleur qui va dépendre du plus petit entier n_0 tel que $|z_{n_0}| > |c| + 1$ si ce n_0 existe. Pour cela on va tester disons jusqu'à $n_0 = 30$.

Bien sûr, pour l'affichage, on ne choisit qu'un nombre fini de valeurs de a et de b. On considère 300 valeurs de (a_k) et 300 valeurs (b_l) entre $[-10, 10]$,

On initialise un tableau 300×300 remplis de zéros, avec la commande `T=pl.zeros((300,300))` (noter les doubles parenthèses).

Pour chaque couple (k, l), on calcule la valeur de $n_0 \leq 50$ correspondante, pour $z_0 = a_k + ib_l$.

Puis on utilise `pl.imshow(T)`

4.3 Ensemble de Mandelbrot

Problème : on cherche à savoir pour quelle valeur de $c \in \mathbb{C}$ la suite (z_n), définie par $z_0 = 0$ (ce choix de valeur initiale est important) et $z_{n+1} = f_c(z_n)$, reste bornée.

Rappel du paragraphe précédent : on peut montrer que si pour un n_0, $|z_{n_0}| > |c| + 1$ alors la suite est non bornée.

Travail à faire :

a) On voudrait représenter pour chaque valeur de $c = a + ib$ avec $a \in [-2, 2]$ et $b \in [-2, 2]$, le point d'affixe c en rouge s'il existe un $i \leq 20$ tel que $|z_i| > 4$ (dans ce cas on admet que la suite est non bornée) et en bleu si $|z_{20}| \leq 4$.

(La valeur 4 est ici un majorant de $|c| + 1$).

Bien sûr, pour l'affichage, on ne choisit qu'un nombre fini de valeurs de a et de b. On considère 300 valeurs de (a_k) et 300 valeurs (b_l) entre $[-2, 2]$,

On initialise un tableau 300×300 remplis de zéros, avec la commande `T=pl.zeros((300,300))` (noter les doubles parenthèses).

Pour chaque couple (k, l), on calcule la valeur de $|z_{20}|$ avec la fonction f_c où $c = a_k + ib_l$.

Ensuite il reste seulement à remplir le tableau `T` en mettant dans la case `T[k,l]` : 1 si $|z_{20}| > 4$ et 0 si $|z_{20}| \leq 4$.

Puis on utilise `pl.imshow(T)` et `pl.colorbar()` pour la légende.

b) En fait le comportement de la suite est plus subtil : ce n'est pas parce que la valeur $|z_{20}|$ est inférieure à quatre qu'on est sûr que $|z_n|$ est non bornée, bien sûr. Mais mieux, il s'avère qu'on peut avoir $|z_{20}| < 4$ et pourtant qu'il y ait un $k < 20$ tel que $|z_k| > 4$ et donc que, d'après le résultat mathématique donné précédemment, que la suite soit en fait non bornée.

Pour visualiser cela, on peut faire une boucle `while abs(z)<4 and iter<20` : on sortira de la boucle dès que le `abs(z) >=4` et on met ainsi en évidence une zone de c pour lesquels en fait $|z_{20}| < 4$ mais entre-temps on avait déjà obtenu $|z_k| > 4$.

On peut choisir de remplir `T` en faisant afficher `iter` s'il est plus petit que `20`. On aura alors un affichage coloré avec 20 couleurs.

Appendice : documentation sur `plot`

A Premiers tracés

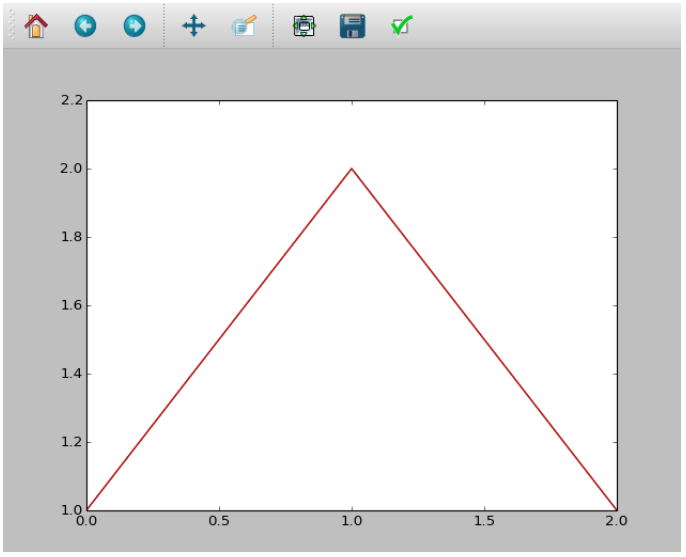
A.1 Rappel : comment marche `plot` ?

On a déjà vu en T.P. que `plot` s'utilise avec la syntaxe :

```
pl.plot(x,y)
```

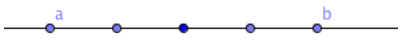
où `x` et `y` sont ou bien des *listes* (ou *tuple*) python, ou bien des *tableaux* numpy *de même taille* et qu'à partir de ces données, `plot` trace la ligne brisée qui joint les point M_I de coordonnées `x[i],y[i]` dans l'ordre des `i` croissants. Ainsi :

```
x=[0,1,2]
y=[1,2,1]
pl.plot(x,y)
pl.show()
```



A.2 Pour le tracé de fonctions : comment fabriquer le tableau des abscisses

Admettons qu'on veuille tracer le graphe de la fonction $x \mapsto x^2$ sur $[0,1]$. On va pour cela découper le segment $[0,1]$ en disons en 11 points espacés régulièrement autrement dit avec un pas $p = 0.1$



Ici $n = 5$, le pas est $\dfrac{b-a}{4}$.

Plusieurs méthodes sont possibles :

- créer à la main une liste PYTHON.
- utiliser la commande `linspace` de `numpy` (incluse dans `pylab`) qui crée un *array* :

L'acronyme `linspace` est pour *linear space*.

```
x=pl.linspace(0,1,11)
```

D'une manière générale, `linspace(a,b,n)` subdivise le segment $[a,b]$ en n points régulièrement espacés, donc avec un pas $(b-a)/(n-1)$.

• utiliser la commande `arange` de `numpy` (incluse dans `pylab`) qui crée un *array* :

Il s'agit encore d'un acronyme pour `array range`. Elle s'utilise comme le `range` des listes sauf qu'elle crée un `array` et permet des *pas* qui sont des flottants.

Ainsi `pl.arange(a,b,p)` crée le tableau des $a+kp$ jusqu'au plus grand k tel que $a+kp<b$.

Ainsi pour obtenir la même subdivision de $[0,1]$ on entrera :

```
x=pl.arange(0,1.1,0.1)
```

Reste ensuite à définir le vecteur `y` :

Les fonctions de `numpy` (ou `pylab`) opèrent directement sur les tableaux

Ainsi à partir du *tableau* `x` (pas d'une liste), on peut créer la liste `y` dont les entrées `y[i]` sont les `x[i]**2` simplement via :

```
y=x**2
```

Puis enfin faire `plot(x,y)`.

A.3 Effacement, gestion de plusieurs fenêtres :

Pour effacer le graphique précédent :

```
pl.clf() # clf pour clear figure
```

Pour afficher dans une autre fenêtre :

```
pl.figure(1) # on crée une figure qu'on appelle 1
pl.plot(x,x)
pl.figure("Ma jolie parabole")  #  on crée une figure qu'on appelle ...
pl.plot(x,y)
```

A.4 Commandes pour les axes, la couleur, le style

Reprenons à tire d'exercice un autre exemple déjà vu : celui du sin sur $[0,2\pi]$.

A.4.1 Quelques commandes pour les axes

Par défaut, le cadrage ne collera pas forcément à ce qu'on voudrait. On peut déclarer

```
pl.xlim(0,2*pl.pi)
```

Plus commode, on peut définir directement les extrémités des deux axes comme suit

```
pl.axis([-4,5,-10,10]) # les x seront entre -4 et 5 et les y entre -10 et 10
```

On peut aussi faire apparaître une grille avec

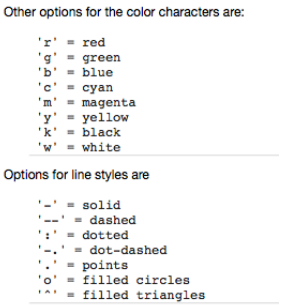
```
pl.grid()
```

On peut donner faire afficher des étiquettes sur les axes :

```
pl.xlabel(" temps t")
pl.ylabel(" tension u(t)")
```

A.4.2 Quelques commandes pour les couleurs et les styles

Les huit couleurs de base : avec leur première lettre.



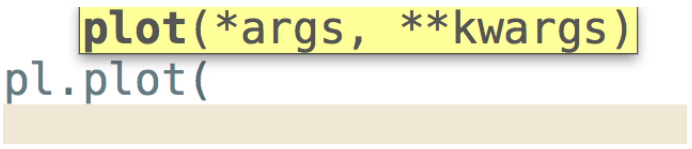
Ainsi `plot(x,y,"r--")`.

Pour des couleurs plus compliquée, on va définir `color= " "` dans `plot`, notamment pour :

Les couleurs en RGB : même syntaxe qu'en HTML : avec trois nombres en hexadécimal : exemple `pl.plot(x,y,color='#eeee00')` donnera la courbe en ...

A.4.3 Remarque générale sur les arguments de `plot`, et de beaucoup de fonctions Python :

Lorsque vous tapez `plot` vous voyez :



Qu'est-ce que cela signifie ?

• Le premier `*args` signifie que le *nombre d'arguments* de `plot` n'est pas toujours le même. Ainsi, on a utilisé `plot(x,y)` avec deux arguments, et `plot(x,y,'r-')` avec trois arguments. On verra même plus tard qu'on peut même ne donner *qu'un* argument !

Comment fabriquer soi-même des fonctions avec des `*arg` ? Avec un `*` devant votre nom d'argument. Dans ce cas, tous les arguments donnés entre virgules seront stockés dans un tuple, comme expliqué ici :

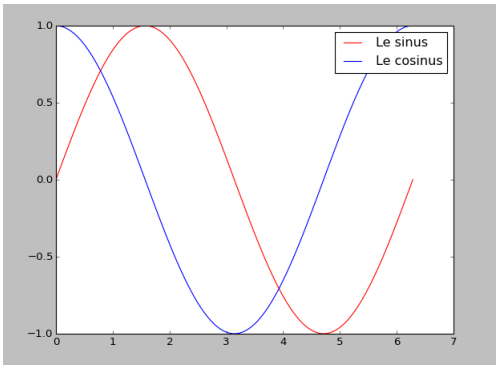
```
def mafonction(*mesarguments):
    return mesarguments
```

• Le second `kwargs` signifie *key word arguments* : ce sont des arguments définis par un *mot-clef*. Comme `color` dans notre exemple. Là aussi, on va voir que `plot` peut en admettre beaucoup.

A.5 Deux tracés sur la même figure avec une légende

```
x=pl.linspace(0,2*pl.pi,100)
y1=pl.sin(x)
y2=pl.cos(x)
pl.clf()
pl.plot(x,y1,"r",label="Le sinus")
pl.plot(x,y2,"b-",label="Le cosinus")
pl.legend()
```

Avec le résultat :



B Cas particuliers des suites, un plot à un argument !

Les suites $n \mapsto u_n$ sont bien sûr des fonctions particulières, dont la variable n est un entier.

On peut bien sûr tracer les par exemple les 1000 premières valeurs de la suite $(\sin(n))$ en faisant :

```
X=list(range(1000))
Y=pl.sin(X)
pl.plot(X,Y)
pl.show()
```

Mais on peut faire la même chose en faisant simplement `pl.plot(y)` :

s'il n'y a qu'un argument, `plot` prendra par défaut comme tableau des abscisses le tableaux des entiers successifs, à partir de 0, de même longueur que y.
Ceci peut paraître une simple curiosité, mais cela peut vous jouer des tours parfois... si vous appliquer votre `plot` à certaines valeurs de retour d'une fonction...

Remarque : comment peut-on illustrer la *densité* de $\{\sin(n), n \in \mathbb{N}\}$ dans $[-1, 1]$, en dessinant tous ces points $\sin(n)$ sur le segment $[-1, 1]$?

Voici ci-dessous le dessin pour seulement les 100 premières valeurs de n : la densité n'est pas évidente, elle le devient pour 1000 :