Quelques consignes/conseils...

et un peu d'idéologie

Voici une liste de consignes générales puis particulières à lire dès maintenant, puis quelques jours avant votre oral Pythonisé à Centrale, puis la veille, puis quelques minutes avant. En fonction de votre psychologie, vous pouvez également les relire quelques minutes après : « Mais quel gros blaireau, c'était pourtant écrit dans cette foutue liste de consignes... »
C'est parti :

1. Avant toute chose, ON CRÉE UN NOUVEAU FICHIER Python QU'ON SAUVE IMMÉDIATEMENT. Cinq minutes après, c'est trop tard.
2. On importe `numpy` sous sa forme préférée : `from numpy import *` ou `import numpy as np`. Vous n'avez jamais trop su la différence, et d'une manière générale vous ne savez pas trop comment gérer les imports de bibliothèques ? C'est bien dommage ; il vous reste quelques semaines pour comprendre ce qu'on vous raconte depuis 2 ou 3 ans.
3. Vérifiez le premier point (la sauvegarde).
4. Si une lecture rapide de l'énoncé vous fait penser qu'il faudra faire un graphique, alors immédiatement : `import matplotlib.pyplot as plt`.
5. Non mais vérifiez VRAIMENT le premier point.
6. En cours de TP, prenez l'habitude de sauvegarder (évidemment...) et tester TOUT DE SUITE ce que vous écrivez : si vous écrivez une fonction qui va être utilisée par une autre fonction, il est suicidaire de se lancer dans la deuxième sans avoir d'abord testé la première. Pour vérifier, on sauvegarde (...), on exécute, et on passe dans la fenêtre d'interprétation/exécution/la console en demandant la valeur de la fonction en tel ou tel point.
7. Au fait, le premier point ?

1 Graphiques

1. On vérifie qu'on a bien importé `matplotlib.pyplot`.
2. Même si l'énoncé vous réclame plusieurs dessins, on ne fait JAMAIS plusieurs dessins avant d'avoir testé que le premier fonctionne !
3. Je conseille d'utiliser `plt.grid()` qui fournit une grille facilitant la lecture. On peut aussi tracer les axes, mais personnellement je trouve la syntaxe un peu plus pénible, et ça m'est moins utile. À vous de voir.
4. Avant même de faire un `plt.show()` (à la fin du script Python en général) je commence toujours par sauver le graphique via par exemple `savefig('mon_exo.pdf')`. Un fichier sera créé et sauvé dans le répertoire courant (si vous êtes sous Spyder et probablement Idle... mais peut-être aussi n'importe où si vous êtes sous Pyzo ; dans ce dernier cas, l'exécution via la combinaison CTRL SHIFT E expliquera à Pyzo qu'il est prié de sauvegarder le graphique dans le répertoire courant du script Python en cours.
5. Pensez au `plt.clf()` entre deux graphiques différents, pour effacer la fenêtre graphique en cours avant d'en commencer une nouvelle.
6. Le principe général pour faire un graphique est de donner comme paramètres à `plt.plot` les listes d'abscisses et d'ordonnées des points à relier. Il s'agit donc d'abord de créer ces listes. Notons que « liste » ou « `array` de `numpy` », c'est en première approximation la même chose [1].

1. Et lancer ça chez des informaticiens, c'est ouvrir un troll dont vous n'imaginez pas l'ampleur !

7. Pour tracer le graphe d'une application f, il s'agit de relier les points de coordonnées $(x, f(x))$, pour x prenant disons 200 valeurs entre a et b (prenez $(a, b) = (-5, 5)$ si vous n'avez pas de meilleure idée au début ; vous pourrez changer ça ensuite). S'il s'agit d'un arc paramétré, il faut relier les points de coordonnées $(f_1(t), f_2(t))$ pour certaines valeurs de t ; dans ce cas, on va commencer par créer la liste des t pour créer ensuite les listes d'abscisses/ordonnées... et la liste des t ne sera PAS donnée à `plt.plot`.
8. Pour créer une liste de valeurs entre a et b, on peut utiliser `np.arange(a, b, delta)` (attention, b ne sera pas dans la liste, ce qui n'est en général pas un gros problème si vous avez pris `delta` petit) ou (ce que je préfère) : `np.linspace(a, b, nb_points)`. Ensuite, il faut choisir le nom de cette liste. Une pratique commune (très majoritaire) et que je réprouve consiste (comme dans la notice Centrale) à l'appeler `X`... alors que ce sera une liste de tels x. Je préfère appeler ça `les_x`, comme ça je sais de quoi je parle.
9. Dans le cas $y = f(x)$, vous commencez donc par `les_x = np.linspace(-2, 4, 200)` par exemple, puis pour la seconde liste, vous avez le choix entre :
   — `les_y = [f(x) for x in les_x]` qui va créer la liste des $f(x)$ pour x décrivant `les_x` : c'est ma version préférée, même si on crée ainsi quelque chose qui est une liste, alors que formellement `les_x` est un tableau...
   — `les_y = f(les_x)` si f est une fonction vectorisée. Je suis certain que vous allez aimer cette version même si vous ne savez pas ce qu'est une fonction vectorisée : « ouais mais en pratique ça marche bien ». Voila, *la plupart du temps ça marche bien.* Puis un jour (le jour de l'oral de Centrale, au hasard), ça ne marche pas. C'est parce que votre fonction f n'est pas vectorisée. Ce jour là, il faudra que vous vous souveniez que c'est très facile de la vectoriser : `fv = np.vectorize(f)`. Mais quelque chose me dit que ce jour là, vous aurez oublié (et quelques minutes après, vous vous en voudrez VRAIMENT de ne pas avoir suivi mon conseil...).

   Vous pouvez alors lancer le `plt.plot(les_x, les_y)`
10. Dans le cas d'un arc paramétré, ça donnera donc :

```
les_t = np.linspace(-3, 3, 200)
les_x = [f1(t) for t in les_t]  # f1 a évidemment été définie avant
les_y = [f2(t) for t in les_t]
plt.plot(les_x, les_y)
plt.savefig('mon_super_arc.pdf')
```

11. Dans le cas d'arc ou graphe avec asymptote, il peut être utile de réduire le domaine des abscisses ou ordonnées. J'oublie très rapidement la commande, ce qui n'est pas un problème quand on a Google, mais... Or donc, la commande que vous recherchez alors est : `plt.xlim(x1, x2)` et bien entendu `plt.ylim(y1, y2)`
12. Attention, encore une fois, c'est très pratique de pouvoir faire plusieurs graphiques superposés, mais ne faites JAMAIS le deuxième avant d'avoir vérifié que le premier est correct.

2 Équations différentielles

1. Les équations que vous allez pouvoir résoudre numériquement sont de la forme $X'(t) = F(X(t), t)$. J'INSISTE : F possède bien DEUX paramètres. DEUX, comme dans « POUCE+INDEX, ÇA FAIT DEUX DOIGTS. ». Par exemple, si votre équation est $x' = 3x + 2$, alors vous commencez par sortir un papier et un crayon (étape que certains n'arrivent pas à franchir, c'est bien dommage) pour la réécrire $x'(t) = F(x(t), t)$, avec F l'application $(a, t) \mapsto 3a + 2$.
2. Vous avez trouvé l'exemple précédent compliqué, donc vous êtes passé à la suite en regardant ailleurs et en sifflotant ? Dommage. Je vous conseille de reprendre l'exemple précédent.
3. Non mais vraiment en fait...
4. Vous pouvez considérer que c'est vraiment un truc de blaireau de prendre une deuxième variable qui n'intervient pas dans le résultat de la fonction, et donc ne pas l'utiliser. En expliquant vos arguments à Python, il va probablement comprendre et finir par être d'accord avec vous...

5. Comme en maths, TOUTES LES ÉQUATIONS DIFFÉRENTIELLES SONT EN FAIT D'ORDRE UN. Par exemple, l'équation $y'' = -\sin y - y'^2 + \cos t$ devient en posant $Y = (y, y')$:

$$Y'(t) = (y'(t), y''(t)) = F(Y(t), t) \quad \text{avec} \quad F : ((a, b), t) \mapsto (b, -\sin a - b^2 + \cos t)$$

6. Vous avez trouvé l'exemple précédent compliqué, donc vous êtes passé à la suite en regardant ailleurs et en sifflotant ? Dommage. Je vous conseille de reprendre l'exemple précédent.
7. Non mais vraiment en fait...
8. Finalement, dans l'exemple précédent, vous pourrez définir F de la façon suivante :

```
def F(Y, t)  # celui qui écrit ça sait que Y DOIT être une liste/un vecteur
    (a, b) = Y  # on met dans a et b les deux valeurs qui sont dans Y
    return (b, -sin(a)-b^2+cos(t))
```

9. On donne ensuite à `odeint` la fonction F, les points en lesquels on veut des approximations de $Y(t)$, et la condition initiale, qui est la valeur $Y(t_0)$, où t_0 est la PREMIÈRE valeur de votre liste de t (je n'ai pas dit la plus petite...). Pour l'ordre, j'oublie à chaque fois ; heureusement qu'il y a la doc... Le résultat est alors la liste (le tableau) des approximations. Comme vous avez gardé sous le coude votre liste de temps, vous pouvez balancer ces deux listes à `plt.plot`...
10. Dans le cas d'une équation d'ordre 2, vous récupérez donc un tableau bidimensionnel représentant la liste des valeurs de Y (donc y et y') aux différents temps.

```
>>> valeurs = odeint(F, Y0, les_t)
>>> valeurs
[[y0, yp0],
 [y1, yp1],
 ...
 [yn, ypn]]
```

Si vous voulez récupérez puis tracer le graphe de y, commencez par slicer !

```
>>> les_y = valeurs[ : , 0]
>>> plt.plot(les_x, les_y)
```

11. Si on vous demande de tracer une solution sur $[t_1, t_2]$ avec une condition initiale en $t_0 \in]t_1, t_2[$, vous devez faire en fait deux tracés : l'un classique de t_0 à t_2, et l'autre plus dérangeant pour le cerveau (mais pas pour Python) de t_0 à t_1 : `linspace(t0, t1, 100)` vous fournira une liste décroissante de valeurs de t commençant à t_0, ce qui est exactement ce qu'on veut. Et NON, je ne vous dirai pas comment on fait pour avoir la même couleur sur les deux branches tracées successivement.

3 Intégrales

1. Pour calculer $\int_a^b f$ il suffit d'appeler : `quad(f, a, b)`, avec `quad` importé de la bibliothèque `scipy.integrate` et après avoir bien entendu défini la fonction f.
2. Attention, la valeur retournée est un couple (a, b) avec a la valeur approchée de l'intégrale, et b une estimation d'un majorant de l'erreur. La valeur de b est en général petite... et en général mise à la poubelle, donc les appels se font souvent sous la forme `quad(f, a, b)[0]`
3. On vous demande parfois de calculer $\int_a^b f_n$ pour différentes valeurs de n. La (une) bonne façon de faire consiste alors à définir f_n à l'intérieur de la fonction chargée de calculer l'intégrale :

```
def I(n):
    def f(x):
        return cos(x)**n
    return quad(f, 0, pi/2)[0]
```

4. J'ai appris en lisant la notice Python qu'on peut calculer des intégrales impropres de type $\int_a^{+\infty}$ en prenant comme borne `np.inf`

4 Algèbre linéaire

1. Définir une matrice ou un vecteur en Python ne s'apprend pas le jour J en lisant la notice.
2. Non mais vraiment, en fait !
3. Sachez donc définir de telles matrices, les multiplier, les transposer, les mettre à une puissance entière...
4. Vous avez accès au spectre d'une matrice via la fonction `numpy.linalg.eigvals` et aux valeurs propres via...
5. Le résultat de `numpy.linalg.eigvals` et `numpy.linalg.eigs` est d'un type assez naturel que vous ne découvrirez pas non plus le jour de l'oral : pendant la préparation, vous aurez testé ces fonctions sur des exemples...
6. C'est un scandale absolu, mais il est parfois question de manipulation de polynômes en Python (reliquat de temps anciens où on faisait du calcul formel). Vous ne comprendrez pas l'utilisation de la bibliothèque `numpy.polynomial` le jour de l'oral, donc lisez la notice bien avant, et tapez les exemples proposés (et allez récupérer un énoncé les utilisant, ainsi que le corrigé !). C'est pénible d'investir du temps dans un truc si anecdotique, donc concentrez-vous bien : en 10 minutes vous pouvez faire le tour de la question. Exercice : calculer le polynôme de Tchebychev T_{20}, sachant que $T_0 = 1$, $T_1 = X$, et $T_{n+2} = 2XT_{n+1} - T_n$ pour tout $n \in \mathbb{N}$. Représenter ensuite le graphe de $\theta \mapsto T_{20}(\cos\theta)$ ainsi que celui de $\theta \mapsto \cos(20\,\theta)$.