Psi 999 – 2024/2025 ψ **Maths 2 – Centrale**

Quelques consignes/conseils...

et un peu d'idéologie/de dogmatisme

Voici une liste de consignes générales puis particulières à lire dès maintenant, puis quelques jours avant votre oral Pythonisé à Centrale (maths 2), 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... »*

1 Maîtrise de l'environnement

Pour 2024 vous disposerez normalement au choix (le votre !) de Pyzo ou Spyder (vous choisirez donc ce dernier !).

1. Avant toute chose, ON CRÉE UN NOUVEAU FICHIER Python QU'ON SAUVE IMMÉDIATEMENT. Cinq minutes après, c'est trop tard. Et le nom termine avec le suffixe `.py` bien entendu. Il va sans dire, mais...
2. On importe `numpy` sous sa forme préférée : `from numpy import *` ou `import numpy as np` ou encore autre chose que vous maîtrisez. 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. Au pire vous recopiez caractère par caractère ce qu'il y a sur les notices. Parfois ça marche.
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 ?
8. La gestion globale de l'environnement est importante. Au choix :
   — Avoir un script Python pensé dès le début pour être exécuté globalement. À chaque nouvelle chose écrite (une fonction, typiquement), vous exécutez globalement le script (F5 par exemple, ou « la flèche verte ») et testez le nouvel élément. Si jamais votre script contient un gros calcul hors fonction (c'est mal et/ou rare), vous pouvez le commenter une fois testé et les résultats copiés/collés.
   — Si vous savez faire, vous pouvez décider de tester des boûts de scripts. Vous ne savez pas comment faire ? Il est trop tard pour apprendre : adoptez le premier point de vue.
9. Le premier point ?
10. Sachez commenter une ligne/partie de code.
11. Je conseille une dernière fois de copier/coller de la console vers le script python le résultat de vos exécutions. Quand, fatalité, plus rien ne marchera face à l'examinateur, il vous restera des traces de ce qui a marché à un moment.
12. Quand vous avez quelque chose qui marche (même vaguement), ne le modifiez pas (spoiler : la nouvelle version ne va pas marcher) sans avoir copié/collé/commenté le code qui marchait.

2 Graphiques

1. On vérifie qu'on a bien importé `matplotlib.pyplot`.
2. Même si l'énoncé vous réclame plusieurs graphes sur une même représentation, on ne fait JAMAIS plusieurs graphes 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 [1] 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 magique 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 [2].
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).
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 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 (et qui est heureusement dans la notice) est : `plt.xlim(x1, x2)` et bien entendu `plt.ylim(y1, y2)`
11. 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.
12. Après avoir fait un graphique il convient évidemment de l'INTERPRÉTER, sans quoi il était inutile ! *« il semblerait que... » « on vérifie bien ce qu'on a montré avant : ... » « la convergence semble rapide/lente... » « ça semble franchement incohérent avec ce qui a été établi, mais je ne comprends pas l'erreur probable »*, etc.

1. En général, le `plt.show()` est inutile : le graphique existe probablement dans une fenêtre... éventuellement cachée. Mais comme vous aurez testé l'affaire avant le jour J, ça ne posera pas de problème !
2. Et lancer ça chez des informaticiens, c'est ouvrir un troll dont vous n'imaginez pas l'ampleur !

3 Probabilités

1. Beaucoup d'exercices de probabilités ont pour composante informatique l'estimation d'une espérance ou bien d'une probabilité (vu comme cas particulier de l'espérance d'une Bernoulli [3]). On réalise des tas d'expériences et on fait une moyenne. Il est de bon ton de préciser que l'espérance et la moyenne expérimentale sont reliées par la loi faible des grands nombres qui raconte vaguement que l'un approche l'autre, non ? On peut même réfléchir à la vitesse de convergence : vous avez ce qu'il faut dans le programme !
2. En pratique, j'écris presque toujours les mêmes choses : j'ai une expérience à simuler, en général avec un paramètre, disons p. Je vais alors écrire une fonction réalisant l'expérience (une seule, c'est important !), une autre estimant l'espérance via la moyenne, et pourquoi pas quelques lignes pour visualiser ça.

```
def une_experience(p):
    ...
    return le_resultat

def esperance(p, nb_exp):
    return sum(une_experience(p) for _ in range(nb_exp)) / nb_exp

les_p = linspace(0, 5, 100) # 100 paramètres différents entre 0 et 5
les_esp = [esperance(p, 10**3) for p in les_p]
plt.plot(les_p, les_esp)
plt.grid()
plt.savefig('plouf.pdf')
```

   Pour le nombre d'expériences, 10 est certainement trop faible et 10^{10} trop grand... Réfléchir aussi au nombre de paramètres différents pour lesquels vous ferez les estimations. Pensez à *la règle du milliard* : sensiblement moins (resp. plus) qu'un milliard d'opérations ça passe crème (resp. : c'est mort).
3. Vous trouverez dans la notice le fonctionnement des fonctions permettant de simuler des variables aléatoires suivant les différentes lois (uniforme, Bernoulli vue comme un cas particulier d'une binomiale [4], Poisson, géométrique). Ces fonctions fournissent même une liste de tirages indépendants [5].

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 vecteurs propres via `numpy.linalg.eig`
5. Le résultat de `numpy.linalg.eigvals` et `numpy.linalg.eig` est d'un type assez naturel que vous ne découvrirez pas non plus le jour de l'oral : quelques jours/semaines avant le jour J, 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 $\theta \mapsto \cos(20\,\theta)$... sur un intervalle de votre choix.

3. Oui. C'est bien d'y avoir réfléchi une fois...
4. Ce qui ne vous surprendra pas le jour J car vous l'aurez lu ici avant, et testé d'ici là !
5. Ce qui ne vous surprendra pas le jour J...

7. Vous pouvez rencontrer des exercices avec de l'algèbre bilinéaire. La compétence principale attendue sera alors probablement l'orthogonalisation/orthonormalisation de Schmidt. Il s'agit d'avoir compris ce qu'on fait géométriquement, de le traduire précisément par un algorithme (à deux balles : c'est une boucle !) qu'on écrit sur papier. Puis de le coder. La difficulté principale est la maîtrise du type de données : il y aura certainement deux listes de vecteurs (la vieille base et la nouvelle) ; chaque vecteur est... dépendant du contexte (une liste ou un tableau, un polynôme pourquoi pas une matrice...) et vous devez disposer d'un produit scalaire agissant sur des couples de vecteurs du contexte. Ensuite, piece of cake. Si vous êtes dans $\mathbb{R}^n$ avec des `array` vous trouverez dans la notice comment faire le produit scalaire canonique (et vectoriel dans $\mathbb{R}^3$).
8. Sachez résoudre un système linéaire, après traduction sous la forme $AX = Y$.

5 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 préférer une fonction avec une seule variable. 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_t, 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.

6 Calcul numérique

1. On peut vous demander d'estimer des intégrales, ou encore de donner une valeur approchée d'une solution de $f(x) = 0$ (dans différents contextes).
2. 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.
3. 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]`
4. Vous allez oublier le point précédent.
5. 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]
```

6. J'ai appris en lisant la notice Python qu'on peut estimer des intégrales impropres de type $\int_a^{+\infty}$ en prenant comme borne `np.inf`
7. Pour résoudre une équation de la forme $f(x) = 0$ vous pourrez utiliser `fsolve` si f est à valeurs réelles ; `roots` sinon : aide et exemples fournis dans la notice. À noter que si vous disposez par exemple d'un paramètre p et que vous devez, à p fixé, résoudre $f(p, x) = 0$. Il conviendra de se ramener au cas précédent avec quelque chose comme ça :

```
def resoudre(p):
    def g(x):
        return f(p, x)
    return fsolve(g, 0)
```

(attention, le deuxième argument de `fsolve` n'est peut-être pas ce que vous croyez !)

8. On peut aussi vous demander de coder à la main une méthode particulière de résolution de $f(x) = 0$ dont on vous fera/aura fait étudier la vitesse de convergence : dichotomie ou méthode de Newton.

7 Manipuler efficacement une suite

De nombreux exercices nécessitent de définir une suite, observer différents termes, éventuellement en fonction d'un paramètre...

1. Dans le cas (fréquent) d'une suite définie par une relation de la forme $u_{n+1} = f(u_n)$ ou encore $u_{n+1} = f(n, u_n)$, vous pouvez choisir un programme itératif (boucle) ou récursif. C'est essentiellement la même chose dans ce contexte. Si vous savez **presque** faire les deux, c'est fâcheux. Il s'agit de **vraiment** maîtriser une façon de faire. En imaginant que la fonction f ait été définie avant, ça peut donner :

```
def u(n, u0):
    if n == 0:
        return u0
    else:
        return f(u(n-1, u0))

def u(n, u0):
    uk = u0
    for _ in range(n):
        uk = f(uk)
    return uk
```

2. Dans le cas où on vous demande de calculer les 10 premiers termes, faire un programme qui les calcule et les affiche en même temps est une catastrophe : c'est de l'informatique mal comprise des années 90. Vous valez mieux que ça ! Vous allez plutôt écrire une fonction calculant u (dans l'exemple qui suit, sans paramètres) puis les placer très simplement dans une liste :

   ```
   premiers_termes = [u(n) for n in range(10)]
   ```

   Vous pouvez au choix demander à votre programme de les afficher via un `print(premiers_termes)` mais c'est très maladroit : pourquoi ne pas plutôt aller dans la console et lui demander tout simplement ?

   ```
   >>> premiers_termes
   ```

   Bien entendu aussitôt après vous recopiez la commande et le résultat vers votre script Python (et vous sauvegardez...).

3. Vous souhaitez les représenter ? Pas de problème !

   ```
   les_n = list(range(20))
   les_u = [u(n) for n in les_n]
   plt.plot(les_n, les_u)
   plt.grid()
   plt.savefig('mon_dessin.pdf')
   ```

4. Vous voulez les représenter pour différents paramètres intervenant dans la suite ? Pas de problème ! Les perfectionnistes peuvent même mettre une légende...

   ```
   les_n = list(range(20))
   for p in [1, 42, 0.15]:
       les_u = [u(p, n) for n in les_n]
       plt.plot(les_n, les_u, label = p)
   plt.grid()
   plt.legend()
   plt.savefig('mon_dessin.pdf')
   ```