Oraux - Python

Concours

- Principalement Centrale Maths 2, et Centrale Physique 2
- 30' préparation (avec ordi), 30' oral (sans ordi)
- D'autres concours : ENSAM, Saint-Cyr, ...

Objectifs

- Python = Grosse calculatrice. Permet de conjecturer des résultats / tester.
- Vous pouvez utiliser tout de ce que vous voulez, on veut juste que ça fonctionne.

Énoncés

- Sur le site du concours : google `python centrale`
  - `https://www.concours-centrale-supelec.fr/CentraleSupelec/MultiY/C2015`
  - Calcul matriciel, Réalisation de tracés, Analyse numérique, Polynômes, Probabilités
  - Ces documents ne constituent en aucun cas une liste de connaissances exigibles dont l'ignorance serait sanctionnée. Par ailleurs, ils n'ont pas l'objectif d'être exhaustifs.
  - (sujets PC et PSI)
- Beos : `http://beos.prepas.org/ > Consulter > Informatique`.
- Les sujets de vos camarades l'année dernière :
  - compilés par Monsieur Ridde
  - disponibles sur `S:\\c9XX\donnees\python\oraux\`
  - Fiches retour des anciens, plus corrigés (parfois)
- N'hésitez pas à me poser des questions / à m'envoyer un mail.

Les erreurs à ne pas commettre

Pour éviter les erreurs classiques :

- Lire le rapport du jury. C'est important
- Lire les conseils de Mr Ridde !

Les petits trucs qui facilitent la vie :

- Savoir utiliser Pyzo en général
- Savoir lire/écrire dans un fichier sous Pyzo : essayez. Vraiment.
- `from truc import *`, peut être source d'erreurs, en particulier entre `random` et `numpy`
- `array / matrix`. Pas de multiplication matricielle `A * B` avec `array` : utiliser `A.dot(B)` ou `np.dot(A, B)`
- *log* et *ln* : notation anglosaxonnes : utiliser `log10`, `log` (pour *ln*), ou directement `log(x, base)` (module `math`)
- probas : esayer au moins une fois une loi binomiale
- binôme : `fact(n) // (fact(k) * fact(n-k))` et savoir écrire la factorielle en récursif...
- si on obtient "map", 'iterator","generator" -> utiliser `list(...)` pour afficher le contenu (python 3)
- Revoir le slicing `t[a:b, c:d]` pour les `np.array`. Pratique avec `odeint` en particulier

> Savoir utiliser 'odeint' pour des équations différentielles d'ordre supérieur. Les yeux fermés.

Python : ce qui "manque" dans les fiches

- Dichotomie : `resol.bisect(f, a, b)`
- Module d'un nombre complexe : `abs(z)`
- Polynôme caractéristique d'une matrice : `np.poly(A)` (attention, les coefficients ne sont PAS dans le même ordre que pour le module `numpy.polynomial`)

Pyzo - Remarques générales

- Utilisez les cellules de Pyzo (`## ....`) et `Ctrl+Entrée`
- Si vous avez besoin de lire/écrire dans un fichier : `Démarrer > Exécuter le script`
- Utilisez la complétion automatique de Pyzo (touche TAB quand il affiche une suggestion)
- Vous pouvez utiliser des fonctions pour tout, je trouve ça plus pratique :

```
def question1():
    ...
    plt.plot(...)
```

Matrices

Attention, `np.zeros` a un seul argument, qui peut être un couple par exemple (le deuxième argument, optionnel, permet de préciser le type des valeurs).

```
>>> np.zeros((3, 3))        # <--- il faut deux 'niveaux' de parenthèses
>>> np.array([1, 2, 3, 4])  # idem
```

Pour `alg.eig(A)` : renvoie les valeurs propres `w[i]`, et la matrice `v` telle que "The normalized (unit "length") eigenvectors, such that the column v[:,i] is the eigenvector corresponding to the eigenvalue w[i]." : chaque colonne de la matrice correspond à un vecteur propre de norme 1, associé à une valeur propre (dans l'ordre).

Polynômes

Attention erreur : pour les polynômes, avec `p = Polynomial([-3, 2, 0, 1])`, la fiche de Centrale utilise `p([1, 2, 3])` alors qu'il faut utiliser `p(np.array([1, 2, 3]))`.

> Ça peut être pratique de définir le monôme `X`, et `U` pour simplifier les notations par la suite :

```
>>> from numpy.polynomial import Polynomial
>>> U = Polynomial([1])
>>> X = Polynomial([0, 1])

>>> 3*X**7 + 2*U
Polynomial([ 2.,  0.,  0.,  0.,  0.,  0.,  0.,  3.], [-1.,  1.], [-1.,  1.])
```

Qui correspond bien au polynôme $3X^7 + 2$

Remarque : les deux termes `[-1., 1.]` ne servent à rien ici.

Graphiques

Pour `import` les modules, on suit les fiches de Centrale !

```
import matplotlib.pyplot as plt
import numpy as np

les_x = np.linspace(-1, 1, 100) # 100 points, ne pas mettre 0.001 !
les_y = [f(x) for x in les_x]  # la fonction 'f' doit être déjà définie hein
plt.plot(les_x, les_y)
```

et si besoin (pas forcément dans Pyzo) :

```
plt.show()                  # Pour afficher la fenêtre
plt.savefig("mafigure.pdf")  # OU, pour enregistrer
```

Légende

```
plt.plot(les_x, les_y, '+', label="Etiquette")  # variante
plt.legend()
plt.grid()
```

Plusieurs courbes (vraiment pratique) :

```
x = np.linspace(0, 1, 100)
for n in range(3):
    y = x**n
    plt.plot(x, y, label="n="+str(n))
plt.legend()
plt.show()
```

Utilisation de `np.vectorize`

Attention, en général pour tracer des graphiques on suppose que les fonctions utilisées sont compatibles avec l'utilisation de tableaux `numpy`. Si ce n'est pas le cas, on peut utiliser `f = np.vectorize(f)` comme précisé dans la deuxième page de "Réalisation de tracés" (ou un décorateur...).

Équations

Je garde la notation de la fiche de Centrale "Analyse numérique" :

```
import scipy.optimize as resol

def f(x):
    """ On construit f telle que : x**2=2 <=> f(x)=0 """
    return x**2 - 2
```

Méthode de Newton (attention...) : fsolve

```
>>> resol.fsolve(f, 1) # N'importe quel x différent de 0, ça fonctionne !
array([ 1.41421356])
>>> resol.fsolve(f, 0) # la dérivée est nulle en 0...
array([ 0.])
```

Méthode de Newton aussi, avec `root`

Comme `fsolve`, mais avec beaucoup plus d'informations, en particulier `blabla.success` et `blabla.x`, comme dans la fiche de Centrale.

```
>>> resol.root(f, 1)
        : ....
 success: True                    # <--- ça a marché !
       x: array([ 1.41421356])
>>> resol.root(f, 0)
        : ...
 success: False                   # <--- ne pas faire confiance à la solution
       x: array([ 0.])
```


Dichotomie : `bisect`, avec un seul `s`

Remarque : la dichotomie n'apparaît pas dans la fiche de Centrale, mais c'est pratique si on veut se restreindre à un intervalle $[a; b]$ particulier

```
>>> resol.bisect(f, 0, 2)
1.4142135623715149
```

Intégration

```
import scipy.integrate as integr
```

- La fonction `integr.quad` renvoie deux valeurs : le résultat, et un estimation de l'erreur, on utilisera souvent `integr.quad(a, b)[0]`, avec `a` ou `b` qui peuvent valoir `-np.inf` ou `np.inf`
- Il existe aussi une fonction pour la méthode des trapèzes, qui fonctionne un peu différement : `integr.trapz(les_y, les_x)`.
  - Elle n'apparaît pas dans les fiches, mais peut être donnée en indication
  - Même syntaxe que `plt.plot`
  - Attention, c'est `les_y` avant `les_x` ! C'est très très trompeur

Équations différentielles

Il FAUT savoir utiliser `odeint(F, y0, temps)` où `F` décrit l'équation différentielle, `y0` la ou les conditions initiales en `t0` et `temps = [t0, t1, ...]`.

- Sur un intervalle $[a; b]$ ($a < b$), si les conditions initiales sont "à gauche" de l'intervalle, tout va bien, on peut utiliser `temps = np.linspace(a, b, nb_points)`.
- Si les conditions initiales sont en `b`, on utilisera `temps = linspace(b, a, nb_points)`, même avec $a < b$!
- Si les conditions initiales sont "au milieu", on coupe l'intervalle en deux...

Attention à : return `np.array((yp, ...))` : il faut deux niveaux de parenthèses

Intégration

Calculs

- Essayez de travailler au maximum avec des entiers :

```
>>> 10**100 * (1/2)**100 # Avec des flottants : approximation
0  # pas du tout la réponse attendue !
>>> 10**100 / 2**100
788860905221011805411728565282786229673206435109023004770278930664062 5L
>>>
```

Calcul formel / symbolique

- Il est possible de faire du calcul formel en python, avec le module `sympy`.
- Si ce module peut être utile, des indications sont données systématiquement
- Vous pouvez voir à quoi ça ressemble sur <live.sympy.org>