1 Interface

L'interface proposée à l'oral devrait être Pyzo ou Spyder. Bien penser à s'habituer à l'interface avant l'oral, en particulier l'éditeur et la console. Savoir comment interpréter le script complet, ou bien seulement une partie de son code. Ne pas oublier d'enregistrer régulièrement son script.

2 Les aide-mémoire

Il est important de se familiariser avec les aide-mémoire, et de ne pas les découvrir le jour de l'oral. Comprendre en particulier ce que sont les arguments et les valeurs renvoyées par les différentes fonctions. Ne pas avoir de doute sur la façon d'importer un module, ou simplement une fonction de module.

3 Utiliser l'aide intégrée

Savoir utiliser l'aide intégrée à l'aide de `help`. Indiquer la fonction entre guillemet permet de ne pas charger le module. S'habituer aux différentes rubriques de l'aide, et savoir ce qu'est un argument optionnel d'une fonction.

```
>>> help("numpy.random.randint")

Help on built-in function randint in numpy.random:

numpy.random.randint = randint(...) method of numpy.random.mtrand.RandomState instance
    randint(low, high=None, size=None, dtype=int)

    Return random integers from `low` (inclusive) to `high` (exclusive).

    Return random integers from the "discrete uniform" distribution of
    the specified dtype in the "half-open" interval [`low`, `high`). If
    `high` is None (the default), then results are from [0, `low`).

    Parameters
    ----------
    low : int or array-like of ints
        Lowest (signed) integers to be drawn from the distribution (unless ``high=None``, in which
        case this parameter is one above the *highest* such integer).
    high : int or array-like of ints, optional
        If provided, one above the largest (signed) integer to be  drawn from the distribution
        (see above for behavior if ``high=None``). If array-like, must contain integer values
    size : int or tuple of ints, optional
        Output shape.  If the given shape is, e.g., ``(m, n, k)``, then ``m * n * k`` samples are drawn.
        Default is None, in which case a single value is returned.
    dtype : dtype, optional
        Desired dtype of the result. Byteorder must be native. The default value is int.

    Returns
    -------
    out : int or ndarray of ints
        `size`-shaped array of random integers from the appropriate
        distribution, or a single such random int if `size` not provided.

    Examples
    --------
    >>> np.random.randint(2, size=10)
    array([1, 0, 0, 0, 1, 1, 0, 0, 1, 0]) # random

    Generate a 2 x 4 array of ints between 0 and 4, inclusive:
    >>> np.random.randint(5, size=(2, 4))
    array([[4, 0, 2, 1], # random
           [3, 2, 2, 0]])
```

Réalisation de tracés

- Import du module de tracé — `import matplotlib.pyplot as plt`
- Tracé de la grille — `plt.grid()`
- Repère orthonormal — `plt.axis('equal')`
- Tracé de ligne brisé, de fonctions, de courbe paramétrée — `plt.plot(X,Y)`

plt.show()

plt.savefig("monfichier.pdf")

plt.figure()

922.1

(a) Pour $p = \frac{1}{50}, \frac{2}{50}, \dots, \frac{49}{50}$, représenter les points de coordonnées : $\big(p, 1 - 0.15\sin(\pi p)\big)$.

(b) Sur le même graphe, pour $p \in]0,1[$, tracer la courbe de la fonction :

$$p \mapsto \frac{2 - 2p + p^2}{2 - p}$$

X = np.arange .. np.linspace

Y = f(X)

plt.plot(X, Y, "+")

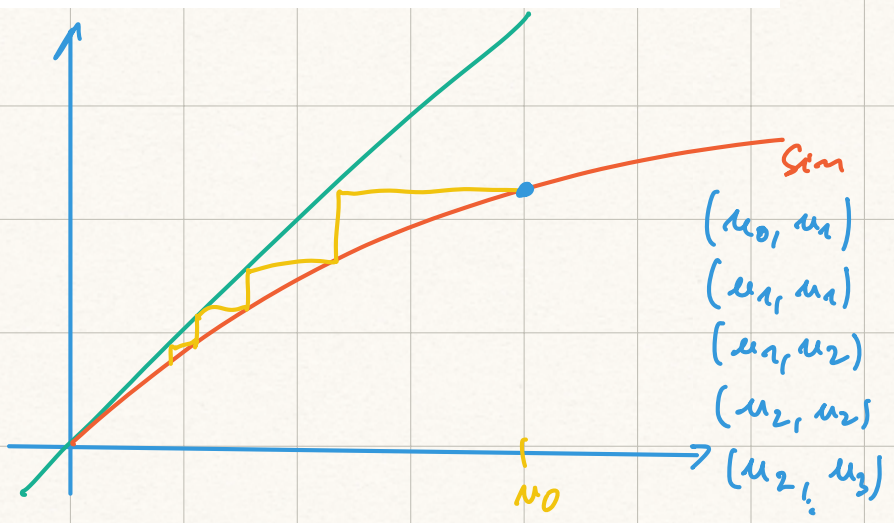
"o-"

922.2

(a) Représenter dans un repère orthonormal, pour $x \in \left[0, \frac{\pi}{2}\right]$, le graphe de la fonction sin, ainsi que la première bissectrice.

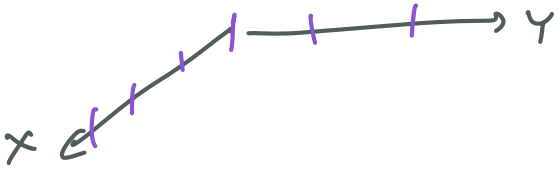
(b) Sur le même graphe, représenter les 10 premiers termes de la suite définie par :

$$\begin{cases} u_0 = \dfrac{\pi}{2} \\ u_{n+1} = \sin(u_n) \quad \forall n \in \mathbb{N} \end{cases}$$



Zoom sur la 3d

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

def f(x,y):
    return ...

X = np.linspace(1, 49, 4)   # [ 1. 17. 33. 49.]
Y = np.linspace(0, 50, 3)   # [ 0. 25. 50.]

X, Y = np.meshgrid(X, Y)
                            # [[ 1. 17. 33. 49.]   [[ 0.  0.  0.  0.]
                            #  [ 1. 17. 33. 49.]    [25.  25. 25. 25.]
                            #  [ 1. 17. 33. 49.]]   [50. 50. 50. 50.]]
Z = f(X, Y)
                            # [[1.49494949 1.39759036 1.25373134 1.01960784]
                            #  [0.7006237  0.97372742 0.97372742 0.7006237 ]
                            #  [1.01960784 1.25373134 1.39759036 1.49494949]]

fig = plt.figure()
ax = fig.add_subplot(projection='3d')

ax.plot_surface(X, Y, Z)
# ax.set_aspect('equal')

plt.savefig("...")          # ou plt.show()

plt.figure()
plt.contour(X, Y, Z, [1.0, 1.1, 1.2])
plt.show()
```



import numpy.random, numpy

Probabilités

- Simuler 20 lancers de dé à 6 faces — `numpy.random.randint(1,7,20)`
- Simuler 20 valeurs d'une va suivant $\mathscr{U}([\![1,7[\![)$ — `numpy.random.randint(1,7,20)`
- Simuler une épreuve de Bernoulli de paramètre .2 — `numpy.random.binomial(1,.2)`
- Simuler 42 valeurs d'une va suivant $\mathscr{G}(.7)$ — `numpy.random.geometric(.7,42)`
- Obtenir un réel « au hasard » dans $[0,1[$ — `X= numpy.random.random()` ; `X<0.7`

La liste des opérations proposée ici est très incomplète, se référer à l'aide-mémoire
Savoir dire : « La loi faible des grands nombres nous permet de dire que si l'on calcule la moyenne d'un grand nombre d'épreuves, la probabilité qu'on soit loin de l'espérance est faible. »

```
def moyenne(N=1000):
    S=0
    for i in range(N):
        S+=X()
    return S/N
```

geometric(.7) ⇝ 3 (c'est $X(\omega_1)$)

geometric(.7) ⇝ 1 (c'est $X(\omega_2)$)

geometric(.7, 200)
↑ nb de ω

geometric(.7, (200, 1000))

922.4

Au rez-de-chaussée d'un immeuble, n personnes entrent dans un ascenseur et choisissent de façon aléatoire un étage parmi p possibles. On note X le nombre d'arrêts.

(a) Écrire une fonction simulant X.

(b) Estimer $\mathrm{E}(X)$ lorsque $n = 10$ et $p \in \{15, 30, 100\}$.

ω. $X(\omega)$?

a) Simuler chq personne Y_i = n° de l'étage

$\sim \mathcal{U}(\llbracket 1, p \rrbracket)$ $\llbracket 1, p+1 \llbracket$

```
Y = numpy.random.randint(1, p+1, n)
X = nb de valeurs ≠ dans Y.
```

b) `def E(n, p, N=1000):`

`...`

```
for p in [15, 30, 100]:
    print(E(n, p))
```

Calcul matriciel

- Créer des matrices, des vecteurs comme $\begin{pmatrix} 1 & 2 & 3 \\ 2 & 3 & 1 \\ 3 & 1 & 2 \end{pmatrix}$ `numpy.array`
- Faire des produits de matrices `A.dot(B)`
- Utiliser la transposée d'une matrice `A.T`
- Obtenir les éléments propres d'une matrice `numpy.linalg.eig`

La liste des opérations proposée ici est très incomplète, se référer à l'aide-mémoire

```
a = np.array([1, 2, 3])
                ↑ liste d'initialisation

# [1 2 3]
b = np.array([[1, 2, 3],
              [4, 5, 6],
              [7, 8, 9]])

b.shape
# (3, 3)
b[1, 2]   (et pas b[1][2])
b[:, 2]
# [2 5 8]
b**2   b*2
# [4 25 64]
b.dot(b)   (ou np.dot(b, b))
# b^T.b = 4+25+64
```

```
e = np.zeros((3,))      (ou np.zeros((3,2)))
f = np.eyes(10)         I_10
d = np.diag([1,2,3])    ( 1 0 0 )
                        ( 0 2 0 )
                        ( 0 0 3 )
```

opérations : → aide mémoire

```
A + B
3 * A
A.dot(B)
```

matrice carrée ↙

```
Sp, P = numpy.linalg.eig(A)
```

↑ tableau des vp.

↑ matrice dont les colonnes "sont" les vecteurs propres associés (dans le même ordre)

Polynômes

- Définir un polynôme comme $1 + X + X^2$

```
X = numpy.polynomial.Polynomial([0,1])
P=1+X+X**2
```

La liste des opérations proposée ici est très incomplète, se référer à l'aide-mémoire

CONCOURS CENTRALE•SUPÉLEC

Oral

Python

MP, PC, PSI, TSI

Polynômes

La classe `Polynomial` du module `numpy.polynomial` ~~`.polynomial`~~ permet de travailler avec des polynômes.

```
from numpy.polynomial import Polynomial
```

Pur créer un polynôme, il faut lister ses coefficients par ordre de degré croissant. Par exemple, pour le polynôme $X^3 + 2X - 3$,

```
p = Polynomial([-3, 2, 0, 1])
```

```
X = Polynomial([0,1])
P = X**3 + 2*X - 3

P(3)

dP = P.deriv()
```

Analyse numérique

- Résoudre (numériquement) une équation comme $x^5 + x^2 + 1 = 0$ — `scipy.optimize.fsolve`
- Résoudre (numériquement) un système comme $\begin{cases} x^2 - y^2 = 1 \\ x + 2y - 3 = 0 \end{cases}$ — `scipy.optimize.root`
- Calculer (numériquement) une intégrale comme $\int_0^{+\infty} \mathrm{e}^{-t^2}\,\mathrm{d}t$ — `scipy.integrate.quad`
- Résoudre (numériquement) une éq. diff. comme $y' = x + y^2$ — `scipy.integrate.odeint`
- Résoudre (numériquement) une éq. diff. comme $y'' + xy' - y^3 = \cos x$ — `scipy.integrate.odeint`
- Résoudre (numériquement) un syst. diff. comme $\begin{cases} x' = -x - y \\ y' = x - y \end{cases}$ — `scipy.integrate.odeint`
- Calculer avec des nombres complexes comme $1 + \mathrm{i}$ — `1+1j`

① `1j` est i `4j`

```
z = 1 + 1j
help(z)
z.real
z.imag
```

Calcul d'intégrale.

```
def fct(.):
    ...
```

`scipy.integrate.quad(fct, a, b)` (fct)

→ renvoie un couple val, erreur

```
val, _ = quad(---)
```

```
def calcul (..):
    def f(x):
        . . .
        return . .
    val, _ = quad(f, ..)
    return val
```

équadiff :

scipy.integrati.odeint (F, y_0, T)

$y' = F(y, t)$

y_0 = cond initiale

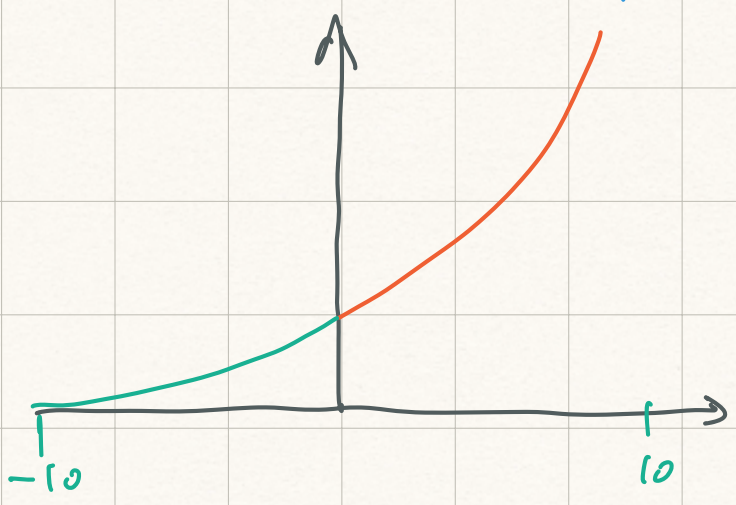
T = tableau/liste des valeurs de t

$$\begin{cases} y' = y \\ y(0) = 1 \end{cases}$$

$F : (y, t) \longmapsto y$

$y_0 = 1$

$T = [0,\ 0.1,\ 0.2,\ \dots\ 1]$



- Souvent, F ne dépend pas de t

```
def F(y, t):
```

- Y peut être un vecteur.

$$Y' = F(Y, t)$$

$$\begin{pmatrix} y_1' \\ \vdots \\ y_n' \end{pmatrix} = F\left(\begin{pmatrix} y_1 \\ \vdots \\ y_n \end{pmatrix}, t\right)$$

Cond. initiale vectorielle $Y(0) = \begin{pmatrix} y_1(0) \\ \vdots \\ y_n(0) \end{pmatrix}$

$$T = [0,\ 0.1,\ \ldots\ 1]$$

- odeint renvoie un tableau de valeurs

$$\text{Sol} = \begin{bmatrix} y_1 & y_2 & y_3 & \\ \sim & \sim & \sim & t=0 \\ \sim & \sim & \sim & t=0.1 \\ \sim & \sim & \sim & t=0.2 \\ \sim & \sim & \sim & \vdots \\ \sim & \sim & \sim & 1 \\ \sim & \sim & \sim & \vdots \\ \sim & \sim & \sim & \end{bmatrix}$$

Si on ne s'intéresse qu'à y_0 : sol[:, 0]

plt.plot(T, sol[:, 0])

- **ED 2** $\quad x'' + x = 1$

① vectorialiser. $Y' = F(Y, t)$

$$Y = \begin{pmatrix} x \\ x' \end{pmatrix}$$

$$Y' = \begin{pmatrix} x' \\ x'' \end{pmatrix} = \begin{pmatrix} x' \\ -x+1 \end{pmatrix} = F\left(\begin{pmatrix} x \\ x' \end{pmatrix}, t\right)$$

```
def F(Y, t):                          Y[0] Y[1]
    x, dx = Y                         return .. [Y[1], -Y[0]+1]
    return np.array([dx, -x+1])
```

x(0) x'(0)

```
Y0 = np.array([1, 2])

T = np.arange(0, 1, .01)
T2 = np.arange(0, -1, -0.01)
sol = odeint(F, Y0, T)
sol2 = - -
plot(T, sol[:, 0])
```