Capacités numériques

Boîte à outils

I Numpy

Nous allons être amenés à utiliser les tableaux de la bibliothèque `numpy`. Leurs manipulations ressemblent à celles des listes, avec quelques spécificités : pas de `.append`, pas de concaténation, mais les opérations sont très simples dessus.

```
import numpy as np # importation de la bibliothèque numpy avec l'alias np
```

I.1 Création de tableaux prédéfinis

Tableau à 1D de n zéros	`t=np.zeros(n)`
Tableau à 2D de n lignes et m colonnes	`t=np.zeros((n,m))`
Tableau à 1D de n valeurs régulièrement espacées, entre a (inclus) et b (inclus)	`t=np.linspace(a,b,n)`
Tableau à 1D de valeurs régulièrement espacées, entre a (inclus) et b (exclus) par pas de p	`t=np.arange(a,b,p)`

```
>>> A=np.zeros(10) # création d'un tableau de 0  à une dimension, de 10 valeurs
>>> A
array([0., 0., 0., 0., 0., 0., 0., 0., 0., 0.])
>>> A=np.zeros((3,5)) # création d'un tableau de 0 à deux dimensions de 3 lignes (1iere dimension) et 5 colonnes (2è dimension)
>>> A
array([[0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0.]])

>>> B=np.ones(5)
>>> B
array([1., 1., 1., 1., 1.])

>>> C=np.linspace(1,10,4) # tableau qui contient 4 valeurs régulièrement espacées, entre 1 (inclus) et 10 (inclus) :
>>> C
array([ 1.,  4.,  7., 10.])

>>> D=np.arange(1,10,2) # tableau des valeurs régulièrement espacées, par pas de 2, entre 1 (inclus) et 10 (exclu) :
>>> D
array([1, 3, 5, 7, 9])
```

I.2 Opérations sur les tableaux

Récupérer l'élément de rang `i` d'un tableau 1D	`t[i]`
Récupérer les éléments entre les rangs `a` (inclus) et `b` (exclus) d'un tableau 1D	`t[a:b]`
Récupérer les éléments à partir du rang `a` (inclus) d'un tableau 1D	`t[a:]`
Récupérer les éléments jusqu'au rang `b` (exclus) d'un tableau 1D	`t[:b]`
Récupérer les éléments entre les rangs `a` (inclus) et `b` (exclus) par pas `p` d'un tableau 1D	`t[a:b:p]`
Récupérer l'élément de la ligne `i` et colonne `j` d'un tableau 2D	`t[i,j]` ou `t[i][j]`
Récupérer la ligne `i` d'un tableau 2D	`t[i,:]`
Récupérer la colonne `j` d'un tableau 2D	`t[:,j]`

```
>>> A[:,0]=[1,2,3] # modification de toutes les lignes  de la colonne 0
>>> A
array([[1., 0., 0., 0., 0.],
       [2., 0., 0., 0., 0.],
       [3., 0., 0., 0., 0.]])

for i in range(0,len(A[0])):  # len(A[0]) : nbre d'éléments dans la ligne 0 = nbre de colonnes
    A[:,i+1]=A[:,i]*2 # la colonne i+1 est égale à 2 fois la colonne i
>>> A
array([[ 1.,  2.,  4.,  8., 16.],
       [ 2.,  4.,  8., 16., 32.],
       [ 3.,  6., 12., 24., 48.]])
>>> A[0,:] # je récupère toute la ligne 0
array([ 1.,  2.,  4.,  8., 16.])

>>> A[1:3,1:4] # je récupère les lignes de rangs 1 à 3 exclu, et les colonnes de rangs 1 à 4 exclu
array([[ 4.,  8., 16.],
       [ 6., 12., 24.]])

>>> np.sqrt(A) # Toutes les fonctions mathématiques usuelles sont disponibles dans le module numpy et s'appliquent à l'ensemble des termes du tableau placé en argument.
array([[1.        , 1.41421356, 2.        , 2.82842712, 4.        ],
       [1.41421356, 2.        , 2.82842712, 4.        , 5.65685425],
       [1.73205081, 2.44948974, 3.46410162, 4.89897949, 6.92820323]])
```

I.3 Tableaux de nombre aléatoires

Tableau de `n` réalisations d'une VA uniforme dans l'intervalle `[a,b[`	`t=np.random.uniform(a,b,n)`
Tableau de `n` réalisations d'une variable aléatoire (VA) gaussienne d'espérance `es` et d'écart-type `et`	`t=np.random.normal(es,et,n)`

```
# Tableau qui contient 10 réalisations d'une VA uniforme dans l'intervalle [4,6[
>>> t2=np.random.uniform(4,6,10)
>>> t2
array([4.46432319, 4.95939463, 5.27606821, 4.11668892, 4.08891539,
       4.31635582, 5.41857959, 4.69045148, 5.41163497, 5.60475265])
# Tableau qui contient 10 réalisations d'une VA gaussienne d'espérance 3 et d'écart-type égal à 2
>>> t1=np.random.normal(3,2,10)
>>> t1
array([-0.17762886,  2.76122342,  3.74733478, -0.58493931,  7.69376754,
        2.20415206,  2.25832712,  3.92065487,  3.33449631,  5.54844443])
```

I.4 Fonctions statistiques utiles

Valeur minimale	`>>> np.min(t)`
Valeur maximale	`>>> np.max(t)`
Somme des éléments	`>>> np.sum(t)`
Moyenne	`>>> np.mean(t)`
Écart-type expérimental $\sqrt{\frac{1}{N-1}\sum_{i=0}^{N-1}(x_i-\overline{x})^2}$	`>>> np.std(t, ddof = 1)`

II Graphiques

```
import matplotlib.pyplot as plt # Pour tracer des graphiques
```

II.1 Tracé d'un graphe

Entrée des données expérimentales dans un tableau `array`, grâce à la bibliothèque `numpy` on crée un `np.array` pour chacune des grandeurs (abscisses et ordonnées).

```
x = np.array([rentrer les valeurs de x séparées par des virgules])
#contient les valeurs x auxquelles ont été mesurées celles de y
y = np.array([rentrer les valeurs de y séparées par des virgules])
#ce sont les valeurs de y correspondantes
```

Il est important de rentrer le même nombre de valeurs de `x` et de `y` et dans le même ordre.

```
plt.plot(x, y, 'bo',label='nom de la courbe') # Représenter y en fonction de x avec ronds bleus
plt.xlabel('légende à adapter') # Légende axe de abscisses
plt.ylabel('légende à adapter') # Légende axe des ordonnées
plt.title('titre à adapter') # Titre
plt.legend() # Afficher la légende (utile si plusieurs courbes sur le même graphique)
plt.grid() # Quadrillage
plt.show() # Affiche le graphique
```

II.2 Options

```
plt.plot(x,y,'couleur style_point style_trait',paramètres)
```

— `couleur` : première lettre anglaise de la couleur (b, g, y, c, r, m, w ou k pour le noir)

— `style_point` : (les principaux)

paramètre	x	X	+	P	.	o	*	d
marqueur	croix	croix plein	plus	plus plein	pixel	rond	étoile	carreau

— `style_trait` :

paramètre	:	-	-.	–	,	v
trait	pointillé	ligne continue	point tiret	ligne de tirets	pixel	triangle

II.3 Histogrammes

Pour tracer un histogramme avec Python, on utilise la fonction `plt.hist` :

```
#Représentation d'un histogramme des effectifs
plt.hist(x, bins='rice') # Représentation de l'histogramme des valeurs x / avec optimisation du nombre de classes.
plt.title('Histogramme de x')
plt.xlabel("x (préciser l'unité)")
plt.ylabel('Effectifs')
plt.show()
```

II.4 Barres d'incertitude

Pour ajouter les barres d'incertitude-type sur un graphe, on utilise la fonction `errorbar`.

```
# Renseigner les grandeurs x et y dans des array au préalable.

# Ajout des barres d'incertitude sur la grandeur portée en y
u_x=  # valeur ou tableau contenant les incertitudes-types sur x
u_y = # valeur ou tableau contenant les incertitudes-types sur y
plt.errorbar(x, y, xerr=u_x, yerr=u_y, fmt='go') # Tracé du nuage de points en vert avec les barres d'incertitude en x et y.
```

II.5 Régression linéaire

Il y a plusieurs possibilités pour faire une régression linéaire. Nous présentons ici le script permettent de faire une régression linéaire avec la fonction `np.polyfit` de la bibliothèque `numpy`.

```
# Renseigner les grandeurs x et y dans des array au préalable.

# Ajout des barres d'incertitude sur la grandeur portée en y
u_y = valeur ou tableau # Incertitude-type sur la grandeur portée en y
plt.errorbar(x, y, yerr=u_y, fmt='go') # Tracé du nuage de points en vert avec les barres d'incertitude en y.

# Régression linéaire de y en fonction de x (équation y = p[0]*x+p[1])
p=np.polyfit(x, y, 1)
pente=p[0] # pente de la droite
ord_origine=p[1] # ordonnée à l'origine

plt.plot(x,np.polyval(p,x)) # Tracé de la droite de régression (p en fonction de x)
plt.xlabel('à adapter')
plt.ylabel('à adapter')
plt.title('à adapter')
plt.show()
```

III Monte-Carlo

III.1 Utilisation pour calculer une incertitude-composée

Supposons que l'on cherche à estimer une grandeur y donnée par $y = f(x_1, x_2, ...)$ avec les x_i des données résultants d'une mesure et f une fonction connue.
Chaque x_i est caractérisé par une valeur minimale et une valeur maximale.
La valeur de y est donnée par l'application de la formule.
Pour estimer l'incertitude-type, il faut remonter à la variabilité de y, qui est elle-même une conséquence de la variabilité des x_i.

Simulation Monte-Carlo :

1. Choisir un nombre N de simulations à réaliser ;
2. Réaliser N simulations par tirage aléatoire uniformément réparties dans l'intervalle défini précédemment pour les différents paramètres.
3. Déterminer les valeurs de y pour chaque expérience.
4. L'incertitude-type de y est l'écart-type de la distribution des y_k.
5. La moyenne des y_k permet de retrouver la valeur y.

Exploitation :

6. La moyenne des y_k permet de retrouver la valeur y.
7. L'incertitude-type de y est l'écart-type de la distribution des y_k.

```python
import numpy as np
# Valeurs extrêmes de x1
x1min=
x1max=
# Valeurs extrêmes de x2
x2min=
x2max=
# Nombre de simulations à effectuer
N=
# Tableaux des N tirages aléatoires uniformes
x1_MC=np.random.uniform(x1min,x1max,N)
x2_MC=np.random.uniform(x2min,x2max,N)
# Tableaux des N résultats de l'expérience
y_MC=x1_MC*x2_MC # (si y=x1*x2)
# Valeur de y
y=np.mean(y_MC)
# Incertitude-type sur y :
u_y=np.std(y_MC,ddof=1)
# Résultat :
print('y=',y,'u_y=',u_y)
```

III.2 Utilisation pour une régression linéaire

Supposons que l'on cherche à vérifier un modèle $y = ax + b$ et que l'on ait, après expérience, un ensemble de valeurs expérimentales $\{x_i\}$ et $\{y_i\}$ qui possèdent chacun une certaine variabilité. La régression linéaire simple permet, à partir de l'ensemble des points expérimentaux, de trouver UNE valeur de a et UNE valeur de b.

Pour estimer l'incertitude-type de ces paramètres, il faut réaliser des ensembles de nouvelles mesures $\{x_i\}$ et $\{y_i\}$ puis réaliser une nouvelle régression linéaire. En réalisant un grand nombre de fois cette opération, on obtiendra, en prenant les écarts-types, les valeurs des incertitudes-types sur les paramètres. On utilise pour cela la méthode de Monte-Carlo.

```
# Mesures effectuées
x=np.array([ , , , , , ])        # séries de mesures de x
y=np.array([ , , , , , ])        # séries de mesures de y
Delta_x=np.array([ , , , , , ]) # demi-largeur des intervalles de x
Delta_y=np.array([ , , , , , ]) # demi-largeur des intervalles de y

# On simule N expériences similaires à celle effectuée précédemment
N=     # nombre d'expériences simulées
liste_a_MC=[] # liste des N pentes déterminées par régression linéaire
liste_b_MC=[] # liste des N ordonnées à l'origine déterminées par régression
    linéaire
for k in range(N):
    L_xk=[] # kieme série de mesures de x
    L_yk=[] #  kieme série de mesures de y
    for i in range(len(x)):
        # pour chaque valeur de x et y précédente on génère une valeur
        # aléatoire dans l'intervalle [x-Delta_x,x+Delta_x],
        # et dans l'intervalle  [y-Delta_y,y+Delta_y]
        x_k=np.random.uniform(x[i]-Delta_x[i],x[i]+Delta_x[i])
        y_k=np.random.uniform(y[i]-Delta_y[i],y[i]+Delta_y[i])
        L_xk.append(x_k)
        L_yk.append(y_k)
    # on détermine pour l'expérience k simulée, la régression linéaire
    reg_k=np.polyfit(L_xk,L_yk,1)
    a_k=reg_k[0] # pente de l'expérience simulée k
    b_k=reg_k[1] # ordonnée à l'origine de l'expérience simulée k
    # on ajoute ces résultats aux listes
    liste_a_MC.append(a_k)
    liste_b_MC.append(b_k)

# Valeurs moyennes des N valeurs générées de a et b
a_moy=np.mean(liste_a_MC)
b_moy=np.mean(liste_b_MC)
# Incertitudes-types sur a et b (=écarts-types des N valeurs générées)
u_a=np.std(liste_a_MC,ddof=1)
u_b=np.std(liste_b_MC,ddof=1)
```