

Python et incertitudes par simulation

MONTÉ-CARLO

Sommaire

I Survivre en Python	1
I/A Bonnes pratiques	1
I/B Les bases	2
I/C Gestion de données	2
I/D Automatisation	3
I/E Tracé de graphiques	3
II Simulations MONTÉ-CARLO	5
II/A Principe	5
II/B Application : mesure d'une distance focale	5
II/C Application : régression linéaire	6
II/D En pratique	7

Capacités exigibles

- ☐ Simuler, à l'aide d'un langage de programmation ou d'un tableur, un processus aléatoire permettant de caractériser la variabilité de la valeur d'une grandeur composée.
- ☐ Simuler, à l'aide d'un langage de programmation ou d'un tableur, un processus aléatoire de variation des valeurs expérimentales de l'une des grandeurs – simulation MONTÉ-CARLO – pour évaluer l'incertitude sur les paramètres du modèle.

I Survivre en Python

I/A Bonnes pratiques

À retenir

- 1) Choisir des noms de variables cohérents et logiques.
- 2) Écrire les unités des valeurs.
- 3) Comme sur les copies, on ne fait **pas de calcul numérique** ($0.1/35.5$ par exemple)¹. Les calculs se font sur les variables.
- 4) Les $2.7*10**(-3)$ sont à **absolument** proscrire. Il existe $2.7e-3$ pour cela. Pareil pour les tableaux, voir plus loin.
- 5) On n'**importera pas** `math` en physique.
- 6) On n'**importera jamais avec *** (`from numpy import *`). Il faut connaître l'origine des fonctions et écrire explicitement `np.pi` si on veut utiliser π .

1. Vous pouvez bien sûr en faire au « brouillon » dans des cellules vides par exemple, mais le risque est de se

I/B Les bases

I/B)1 Calcul et affichage basiques



```

1  # Ce qui est après un # est un commentaire, et non traité dans le code
2
3  a = 2          # affecte la valeur 2 à la variable globale a
4  b = 3*a        # b vaut 3*2 = 6. Si on change la valeur de a, on devra recalculer b
5  c = a**3       # ** indique une puissance, ici puissance 3 (donc c = 8)
6  d = 5.5e-3     # eNBRE est un raccourci pour 10^(NBRE). Ici, d = 0.0055.

```

I/B)2 Fonctions basiques



```

1  print(d)          # affiche la valeur de d
2  print(f'd = {d}') # affiche "d = 0.0055"
3  print(f'd = {d:.2f}') # affiche "d = 0.01" : décimal 2 chiffres après ','
4  print(f'd = {d:.2e}') # affiche "d = 5.50e-03" : scientifique 2 décimales
5
6  type(a)           # donne le type d'objet de la variable a : int (entier)
7  type(d)           # donne le type d'objet de la variable d : float (décimal)
8
9  abs(-3)           # donne la valeur absolue d'un nombre : 3
10
11 len([1, 2, 3])    # donne la longueur d'une liste : 3
12 min([1, 2, 3])    # donne la valeur minimale d'une liste : 1
13 max([1, 2, 3])    # donne la valeur maximale d'une liste : 3

```

I/C Gestion de données

I/C)1 Listes



```

1  L = [1, 2, 3]     # créé la liste L contenant les valeurs 1, 2 et 3
2  print(L[0])       # extrait la première valeur de L : 1
3  print(L[-1])      # extrait la dernière valeur de L : 3
4  print(L[:2])      # extrait les deux premières valeurs de L : 1 et 2
5  print(L[1:])      # extrait toutes les valeurs à partir de la deuxième : 2 et 3
6
7  L.append(42)       # ajoute l'élément 42 à la fin de la liste
8  L2 = L + [5, 6]   # concatène la liste L et la liste [5, 6] dans une nouvelle
9  print(L2)         # [1, 2, 3, 42, 5, 6]

```

I/C)2 Tableaux et numpy



```

1  import numpy as np
2
3  tab = np.array([1, 2, 3]) # créé le tableau [1, 2, 3]
4  tab+1                     # ajoute 1 à toutes les valeurs de tab
5  tab*2e-3                  # multiplie toutes les valeurs de tab par 0.002
6  np.sqrt(tab)              # applique racine carré à tous les éléments de tab
7  np.exp(tab)               # exponentielle
8  np.log(tab)               # logarithme NÉPÉRIEN (ln français)
9  np.log10(tab)             # logarithme décimal (log français)
10
11 np.linspace(min, max, nbre) # découpe [min, max] en nbre parties égales

```

tromper dans les unités.

```

12 np.mean(tab)                # donne la valeur moyenne de tab
13 np.std(tab, ddof=1)         # donne l'écart-type de tab
14
15 np.polyfit(X, Y, 1)         # donne les coefficients a et b de Y = a*X+b

```

I/D Automatisation

I/D) 1 Fonctions personnelles



```

1 def puissance(arg1, arg2): # définit la fonction puissance de 2 arguments
2     resultat = arg1**arg2  # variable locale de calcul
3     return(resultat)      # fin de la fonction, résultat final
4
5 print(puissance(2, 3))    # donne le résultat du calcul : 8
6
7 def comparaison(x,y):
8     if x > y:              # condition d'exécution
9         print("argument 1 supérieur au second") # si oui, exécute
10    elif x < y:             # sinon, autre condition
11        print("argument 1 inférieur au second") # si oui, exécute
12    else:                  # pour tous les autres cas,
13        print("argument 1 égal au second")      # exécute
14
15 print(comparaison(1,2))  # "argument 1 inférieur au second"

```

I/D) 2 Boucles for



```

1 for i in range(10):        # crée i qui commence à 0, terminera à 9, et augmente
2                             # de 1 à chaque réalisation des lignes en-dessous
3     print(i)               # affichera 0, puis 1, puis 2, et jusqu'à 9
4
5 L = []                    # liste vide
6 for i in range(3):         # exécute la suite 3 fois : i=0, puis 1, puis 2
7     L.append(3*i)          # ajoute 3*i à la fin de la liste
8     print(L)               # [0, 3, 6]
9
10 for i, k in enumerate(L): # i compte à partir de 0, k prend les valeurs de L
11     print(f'L[{i}] = {k}') # L[0] = 0, L[1] = 3, L[2] = 6
12
13 L2 = [3*i for i in range(3)] # crée L d'une manière plus compacte
14 print(L2)                  # même résultat

```

I/E Tracé de graphiques

I/E) 1 Minimal (Figure 5.1)



```

1 import matplotlib.pyplot as plt
2
3 abscisse = np.linspace(0, 10, 8) # définit les abscisses qu'on tracera
4 ordonnee = abscisse**2           # et les ordonnées
5
6 plt.plot(abscisse, ordonnee)     # place les points, les relie par des segments
7 plt.xlabel('t (s)')              # nomme l'abscisse
8 plt.ylabel('x (m)')              # nomme l'ordonnée
9
10 plt.show()                       # obligé pour afficher

```

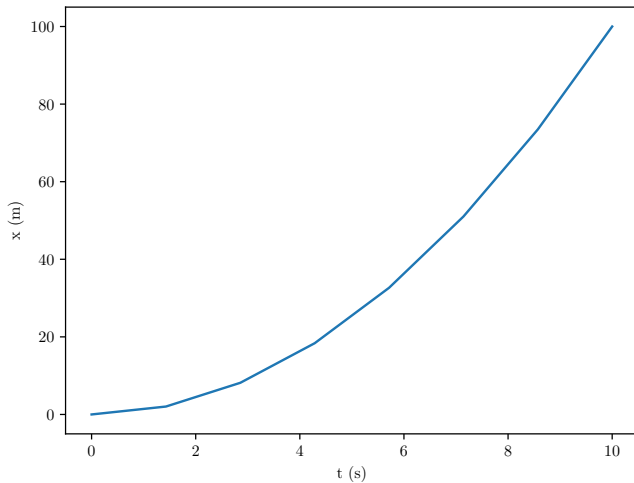


FIGURE 5.1 – Figure minimale.

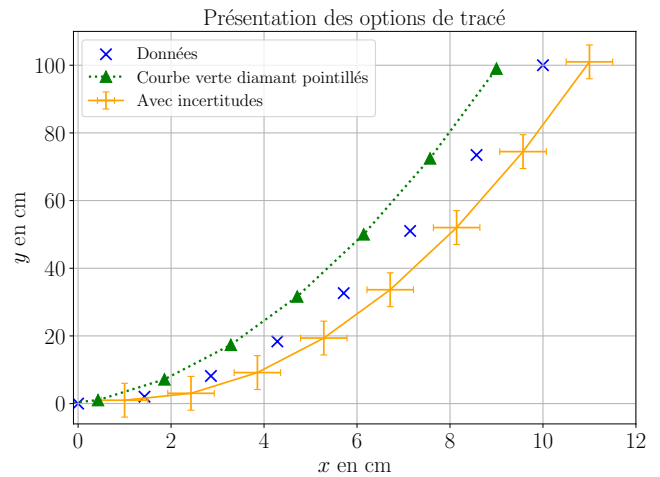


FIGURE 5.2 – Figure complexe.

I/E) 2 Complexe (Figure 5.2)



```

1 X = np.linspace(0, 10, 8)
2 Y = X**2
3
4 plt.figure(figsize=(8, 6))           # dimension horizontale, verticale
5 plt.grid()                           # affiche un quadrillage de lecture
6 plt.xticks(fontsize=20)               # affiche les nombres de l'axe x plus grand
7 plt.yticks(fontsize=20)               # affiche les nombres de l'axe y plus grand
8 plt.xlabel('$x$ en cm',               # Donne le nom de l'axe x, $ pour le mode math
9           fontsize=20)                 # en grand
10 plt.ylabel('$y$ en cm',               # Donne le nom de l'axe y, $ pour le mode math
11          fontsize=20)                 # en grand
12
13 plt.scatter(X, Y,                     # nuage de points X en abscisse et Y en ordonnée
14             marker='x', s=100,         # possibilité de customiser le tracé
15             color='blue',              # pour la couleur
16             label='Données')           # pour la légende
17
18 plt.errorbar(X+1, Y+1,                 # nuage de points X abscisse et Y ordonnée
19             xerr=.5,                    # incertitude en x
20             yerr=5,                     # incertitude en y
21             capsize=3,                  # indique la limite des erreurs
22             color='orange',             # pour la couleur
23             label='Avec incertitudes')
24
25 plt.plot(X-1, Y-1,                    # graphique relié
26          color="g",                     # couleur
27          marker="^",                     # marker
28          markersize=10,                  # taille marker
29          linestyle="dotted",             # type de ligne
30          linewidth=2,                    # épaisseur
31          label="Courbe verte diamant pointillés")
32
33 plt.title('Présentation des options de tracé',
34           fontsize=20)

```

```

35 plt.legend(fontsize=15)
36
37 plt.tight_layout()           # évite les débordements ou rognages
38 plt.xlim(min(X)-.1, max(X)+2) # pour les limites d'affichage en abscisse
39 plt.ylim(min(Y)-6, max(Y)+10) # pour les limites d'affichage en ordonnée
40 plt.show()

```

II Simulations MONTE-CARLO

II/A Principe

On dispose généralement de plusieurs jeux de données pour lesquels on a des incertitudes de mesure, et on veut calculer z qui dépend de ces données mais d'une manière complexe². On peut alors réaliser une simulation.

En effet, connaissant l'intervalle d'existence des mesures, on peut prendre aléatoirement d'autres valeurs possibles pour les mesures, et faire toute une série de calculs avec des valeurs légèrement modifiées. On pourra alors finalement prendre la moyenne des valeurs calculées et leur écart-type pour avoir la propagation des incertitudes !

Cœur de la simulation

Finalement, le cœur de la simulation revient (presque) à réaliser une estimation d'incertitude de type A sur les valeurs calculées !

II/B Application : mesure d'une distance focale

On peut mesurer la focale d'une lentille convergente par la méthode de BESSEL :

$$f' = \frac{D^2 - d^2}{4D}$$

avec d la plage de positions de la lentille qui garde une image nette sur l'écran, et D la distance objet-écran. S'il est possible de faire le calcul analytique ici, il peut être plus rapide de réaliser une propagation des incertitudes des valeurs d et D sur la valeur calculée de f' .

Pour cela,

- ◇ On détermine les demi-largeurs Δ_d et Δ_D . Si on possède des incertitudes-types, on aura $\Delta_d = u(d)\sqrt{3}$. Sinon, c'est la demi-largeur de la plage des mesures possibles.
- ◇ On crée une liste `liste_f` vide qui accueillera les valeurs de f' calculées ;
- ◇ On fixe $N \gtrsim 10^4$ le nombre de simulations ;
- ◇ Pour i allant de 0 à $N - 1$:
 - ▷ On tire aléatoirement une valeur de d dans l'intervalle $[d - \Delta_d, d + \Delta_d]$. On appelle cette valeur `d_simu` ;
 - ▷ On tire aléatoirement une valeur de D dans l'intervalle $[D - \Delta_D, D + \Delta_D]$. On appelle cette valeur `D_simu` ;
 - ▷ On calcule f' avec ces données **simulées** ;
 - ▷ On ajoute cette valeur simulée à la liste des valeurs de f' .

2. Comprendre : pas donnée dans la fiche Mesures et incertitudes

- ◇ On a alors N valeurs de f' . La valeur la plus probable sera la moyenne, et son incertitude-type est l'écart-type de la liste des f' simulés.

Tirage aléatoire

`np.random.uniform(min, max)` est la fonction Python qui permet de tirer aléatoirement une valeur entre `min` et `max`.

Ainsi, en Python :

```
1  import numpy as np
2
3  d = 12          # cm
4  Delta_d = 0.1   # demi-largeur de d
5  D = 50          # cm
6  Delta_D = 0.5   # demi-largeur de D
7
8  N = 10000       # nombre de régressions à effectuer
9
10 liste_f = []    # création de la liste vide pour stocker les valeurs
11 for i in range(N):
12     # on prend des valeurs de d_simu parmi toutes les valeurs possibles
13     # à l'aide d'un tirage aléatoire à l'intérieur de l'intervalle
14     # [d ± Delta_d]
15     d_simu = d + Delta_d * np.random.uniform(-1, 1)
16     # Pareil pour D
17     D_simu = D + Delta_D * np.random.uniform(-1, 1)
18
19     # On calcule les f' simulés avec ces valeurs
20     f_simu = (D_simu**2 - d_simu**2) / (4 * D_simu)
21
22     # On les sauvegarde dans a liste correspondante
23     liste_f.append(f_simu)
24
25 # La valeur moyenne est la moyenne de la liste
26 fmoy = np.mean(liste_f)
27 # L'incertitude est l'écart-type de la liste
28 uf = np.std(liste_f, ddof=1)
29 # Affichage, à modifier pour l'écriture scientifique
30 print(f"f' = {fmoy :.2f} +- {uf:.2f} cm")
```

II/C Application : régression linéaire

Prenons l'exemple de la régression linéaire :

$$y = ax + b$$

On a mesuré x et y , et on obtient a et b avec `np.polyfit(x, y, 1)`. Mais ce calcul ne donne pas l'incertitude sur a et b . Les deux valeurs étant interdépendantes, on n'a pas d'expression analytique pour les déterminer : on va donc les simuler.

Chaque valeur des x_i est comprise dans un certain intervalle $[x \pm \Delta_x]$, et de même pour y . Plutôt que de prendre la valeur centrale et de calculer a et b avec ces valeurs, on peut essayer de calculer a et b pour des valeurs de x_i et de y_i légèrement modifiées, à l'intérieur de l'intervalle des valeurs possibles.

On va donc réaliser un **grand nombre de régressions linéaires en prenant des valeurs x_i et y_i « simulées »**. On obtient une liste de valeurs possibles de a et de b ; la moyenne des a et b sont alors la valeur moyenne de la liste, et l'incertitude-type sera l'écart-type de la liste.

II/D En pratique

- ◇ On détermine les demi-largeurs Δ_x et Δ_y . Si on possède des incertitudes-types, on aura $\Delta_x = u(x)\sqrt{3}$. Sinon, c'est la demi-largeur de la plage des mesures possibles.

Attention !!

Pour la régression linéaire, les Δ_x et Δ_y doivent être des listes !
Il doit y avoir autant de valeurs de Δ que de valeurs de x !

- ◇ On fixe un nombre N très grand.
- ◇ On crée des listes vides `liste_a` et `liste_b` pour y stocker les futures valeurs des a et des b calculés.
- ◇ Pour chaque i compris entre 0 et $N - 1$:
 - ▷ on prend `x_simu` dans l'intervalle $[x - \Delta_x, x + \Delta_x]$;
 - ▷ on prend `y_simu` dans l'intervalle $[y - \Delta_y, y + \Delta_y]$;
 - ▷ on calcule `a_simu` et `b_simu` avec ces valeurs simulées ;
 - ▷ on les stocke dans `liste_a` et `liste_b`.
- ◇ On a alors N valeurs de a et de b : les valeurs les plus probables sont les moyennes, et leurs incertitudes-types sont les écarts-types des listes de a et de b .

Ainsi, en Python :

```
1 x = np.array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]) # unité
2 ux = 0.1 * np.ones(len(x)) # incertitude de 0.1 sur chaque valeur
3
4 y = np.array(
5     [2.20, 2.00, 1.60, 1.55, 1.16, 1.00, 0.95, 0.60, 0.36, 0.36, 0.18]
6 ) # unité
7 uy = 0.12 * np.ones(len(y)) # incertitude de 0.12 sur chaque valeur
8
9 Delta_x = ux * np.sqrt(3) # demi-largeur x
10 Delta_y = uy * np.sqrt(3) # demi-largeur y
11
12 N = 10000 # nombre de régressions à effectuer
13
14 liste_a, liste_b = [], [] # création des listes vides pour stocker les valeurs
15 for i in range(N):
16     # on prend des valeurs de x_simu parmi toutes les valeurs possibles
17     # à l'aide d'un tirage aléatoire à l'intérieur de l'intervalle
18     # [x ± Delta_x]
19     x_simu = x + Delta_x * np.random.uniform(-1, 1)
20     # Pareil pour y
21     y_simu = y + Delta_y * np.random.uniform(-1, 1)
22
23     # On fait une régression linéaire avec ces données « simulées »
24     a_simu, b_simu = np.polyfit(x_simu, y_simu, 1)
25
26     # On les sauvegarde dans les listes correspondantes
27     liste_a.append(a_simu)
28     liste_b.append(b_simu)
29
```

```
30 # Les valeurs moyennes sont les moyennes des listes
31 a_moy, b_moy = np.mean(liste_a), np.mean(liste_b)
32 # Les incertitudes sont l'écart-type des listes
33 ua, ub = np.std(liste_a, ddof=1), np.std(liste_b, ddof=1)
34
35 # Affichage, à modifier pour l'écriture scientifique
36 print(f"Coef.directeur = {a_moy:.3e} +- {ua:.3e}")
37 print(f"Ordonnée à l'origine = {b_moy:.3e} +- {ub:.3e}")
```