Calcul scientifique avec `python` – 2023-2024

Feuille d'exercices n° 5

Exercice 0. On veut représenter graphiquement la fréquence d'obtenir une certaine face du dé en fonction du nombre de lancées.

1) Étudier la fonction `numpy.random.randint(low, high, size)`. Noter son **comportement** par rapport à la borne `high`.

2) Écrire une fonction `frequenceDe(face, nbLancees)` qui prend en arguments la face à étudier et le nombre de lancées (deux entiers) et qui renvoie le vecteur des fréquences d'apparition de la face. Il faut commencer par simuler un vecteur représentant les différentes lancées du dé.

3) Représenter graphiquement l'évolution de la fréquence. Donner un titre et utiliser une grille en arrière-plan.

Exercice 1 (intégrales)**.** On veut approximer la valeur de

$$I = \int_0^1 f(x)\,dx \quad \text{avec} \quad f(x) = x\,(1-x)\sin^2(200\,x\,(1-x))$$

par une méthode de Monte-Carlo (c'est-à-dire à l'aide de simulations de variables aléatoires).

1) Représenter la fonction f sur $[0,1]$.

2) Expliquer le principe de l'algorithme ci-dessous.

```
import numpy.random as npr
def intMonteCarlo(f, xmin, xmax, n):
    w = xmax - xmin
    Y = [f(xmin + npr.rand()*w) for k in range(n)]
    S = sum(Y)
    return (w/n)*S
```

3) On a en fait $I \approx 0.080498$ (on pourra tenter de retrouver cette valeur avec la commande `quad` du module `scipy.integrate`). Évaluer la qualité de la méthode sur quelques simulations.

4) Effectuer des tentatives avec d'autres fonctions.

Exercice 2 (Aiguille de Buffon)**.** L'aiguille de Buffon est une expérience statistique proposée dès le XVIII$^{\text{e}}$ siècle par le mathématicien français Buffon. Le concept est assez simple : en répétant un grand nombre de fois une situation aléatoire bien particulière que l'on va décrire, on peut aboutir à une approximation de π. On considère un parquet composé de planches parallèles sur lequel une aiguille est lancée un grand nombre de fois. On compte le nombre de fois où l'aiguille touche au moins deux planches du parquet. Les lames du parquet sont de même largeur $L > 0$ et l'aiguille est modélisée par un segment de longueur $0 < a \leq L$. À chaque lancer, on note $0 \leq d \leq \frac{L}{2}$ la distance du centre de l'aiguille à la rainure la plus proche et $\theta \in [0, \frac{\pi}{2}]$ l'angle formé par l'aiguille avec les rainures, et ces deux quantités sont considérées comme générées uniformément et indépendamment.

1) Faire un schéma pour bien saisir le problème.

2) Montrer que la condition pour que l'aiguille coupe une rainure est $d \leq \frac{a}{2}\sin\theta$. Sans entrer dans les détails techniques, on peut montrer que la probabilité qu'une aiguille coupe une rainure vaut $\frac{2a}{\pi L}$.

3) Écrire une fonction `tracerparquet(N, L)` qui représente graphiquement un parquet composé de N lames parallèles de largeur L.

4) Écrire une fonction `lanceraiguilles(n, a, N, L)` qui simule le lancer de n aiguilles de longueur a et qui les représente graphiquement sur un parquet de N lames de largeur L.

5) Étudier l'évolution de la suite $(\pi_n = \frac{2a}{LF_n})_{n\geq 1}$ où F_n est la fréquence avec laquelle les n aiguilles ont coupé une rainure du parquet.

Exercice 3. Considérer une (ou plusieurs) des courbes dans la liste ci-dessous et essayer de réaliser sa construction étape par étape. Éventuellement animer la construction. Il faudrait choisir entre une programmation récursive et programmation linéaire (basée sur des boucles).
courbe de Hilbert / flocon de Koch / courbe du dragon

Exercice 4 (ensembles de Julia)**.** On considère pour cet exercice la suite de nombres complexes définie par récurrence $z_0 = a$ et $z_{n+1} = z_n^2 + c$ où a et c sont des paramètres complexes.

1) On peut définir un nombre complexe $a + ib$ dans `python` en tapant `a+bj`, ou `complex(a,b)`.

a) Écrire une fonction prenant en paramètre un nombre complexe et renvoyant son module. (Étudier les commandes `z.real` et `z.imag`.

b) Écrire une fonction prenant en argument deux nombres complexes z, c, et renvoyant le résultat de l'opération complexe $z^2 + c$.

c) Écrire une fonction prenant en paramètre deux nombres complexes a, c et un entier N , et donnant la représentation graphique de la suite $(|z_n|)_n$, dont le n-ème terme correspond au module du nombre z_n. Tester cette fonction avec $a = 0$, $c = -0.8 + 0.156i$ et pour N valant 30, 250, 252, 253. Que se passe-t-il à partir de ces dernières valeurs ?

2) On fait maintenant varier le nombre complexe a, et on s'intéresse, pour un a fixé, à l'indice n à partir duquel la suite $(z_n)_n$ n'est plus bornée, c'est à dire quand son module commence à tendre vers l'infini.

a) Écrire une fonction `max_iter` qui prend en entrée des entiers N_{max}, s, et deux nombres complexes a, c, et qui renvoie le premier rang à partir duquel le module de la suite $(z_n)_n$ paramétrée par a et c dépasse la valeur s, ou N_{max} si le module ne dépasse pas s.

b) Écrire une fonction `julia` qui prend en entrée des entiers N_{max}, s, N_{div}, un réel r et un nombre complexe c, qui calcule pour chaque point (x, y) du carré $[-r, r] \times [-r, r]$, la valeur

$$\min\left(\frac{1}{255}\,\texttt{max\_iter}(N_{max}, s, x + iy, c), 1\right)$$

et qui renvoie une matrice contenant toutes ces valeurs. *Indice :* Utiliser le paramètre N_{div} pour subdiviser l'intervalle $[-r, r]$ en N_{div} intervalles, puis créer une matrice carrée nulle M de taille $N_{div} \times N_{div}$ et pour chaque x_i et x_j dans la subdivision, remplacer $M[i][j]$ par le minimum entre $\frac{1}{255}\,\texttt{max\_iter}(N_{max}, s, \texttt{complex}(x_i, x_j), c)$ et 1.

c) Tester la fonction avec $N_{max} = 1000$, $s = 100$, $N_{div} = 150$, $r = 1.5$ et $c = -0.8 + 0.156i$. Utiliser la fonction `plt.imshow` qui prend en paramètre une matrice contenant des valeurs entre 0 et 1 et renvoyant l'image associée.

Zen of `python`

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
[...]
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.