Troisième séance : probas

Pour cette dernière séance, nous allons essayer d'écrire des programmes un peu plus conséquents, et je vous laisserai déterminer les étapes (donc en gros les fonctions) intermédiaires.

1 Méthode de Monte-Carlo pour calculer π

La méthode est très simple : on tire aléatoirement et uniformément des points dans le carré $[0,1]^2$. La probabilité que le point soit dans le disque $\mathcal{C}(0,1)$ est donc π.

Par la loi forte des grands nombres, lorsque le nombre de tirages tend vers ∞, la fréquence des points tombant dans le disque a une probabilité 1 de converger vers π.

On déduit alors un programme simple quoiqu'assez inefficace pour calculer π.

Pour aller plus loin : Étant donné $\varepsilon \in \mathbb{R}^{+*}$, combien de fois faut-il répéter le tirage pour espérer obtenir π avec une précision ε ?

Commande utile : Pour tirer un nombre au hasard dans $[0,1[$, on peut utiliser `rand` de la bibliothèque `np.random`. Attention : cette fonction ne prend aucun argument, on l'utilise donc avec des parenthèses vides : `rd.rand()`, pour peu que la bibliothèque ait été chargée sous l'alias `rd`.
Remarque : J'ai pour habitude d'utiliser la bibliothèque numpy car je l'utilise dans mon cours par ailleurs. Cependant, une bibliothèque plus simple pour générer des nombres aléatoires est la bibliothèque `random` présente dans la distribution Python de base. Taper `import random as rd`. Je ne l'utilise pas car elle présente un défaut : pour $n \in \mathbb{N}$, l'appel à `random.randint(0,n)` renvoie un élément entier aléatoire dans $[\![0,n]\!]$, donc n *inclu*, ce qui est une hérésie pythonesque ! Au contraire le `randint` de `numpy.random` a bien le comportement attendu d'exclure la borne finale. Enfin, précisons que dans la bibliothèque `random` il faut utiliser `randrange` pour exclure la borne finale. Taper dans une console `import random ; help(random.randrange)`, c'est intéressant...

2 Diffusion 1D, mouvement brownien

On considère un tableau `t` de longueur n. Initialement, la case centrale contient N particules (on y enregistre le nombre N) et les autres cases sont vides (on y enregistre 0).

À chaque étape, chaque particule (sauf celles du bord) a une probabilité $\frac{1}{2}$ de se déplacer d'une case vers la droite ; sinon elle se déplace d'une case vers la gauche.

On propose d'écrire un programme qui prend en entrée n et N et un entier `nbEtapes` et qui initialise puis fait évoluer le tableau pendant `nbEtapes` étapes.

On pourra tracer l'état du tableau à différent instants grâce à des `plt.plot(range(n), t)` . Problème technique : avec la modélisation choisie, à chaque étape toutes les particules sont dans des cases de même parité... Si on veut un dessin qui ressemble à une Gaussienne, on peut utiliser `plt.plot( range(0,n,2), t[0:n:2])` pour tracer uniquement les points d'abscisse paire et `plt.plot( range(1,n,2), t[1:n:2])` pour les points d'abscisse impaire. Ou alors démarrer la simulation en remplissant deux cases adjacentes au lieu d'une seule.

Autre méthode / prolongement : Lorsqu'une case contient n cases, pour les faire toute bouger la méthode directe est d'utiliser une boucle « `for z in range(n)` ». Cependant, on pourrait aussi utiliser une loi binomiale pour déterminer combien de particules partent à droite. Ceci amène une question finalement pas évidente : y-a-t-il une manière pour programmer une loi binomiale plus directe que d'additionner n Bernoulli ? On peut tirer un flottant x dans $[0,1[$ puis trouver le plus grand entier k tel que $\sum_{j=0}^{k} \left(\frac{1}{2}\right)^n \binom{n}{j} \leqslant x$. On se retrouve alors à devoir calculer les coefficients binomiaux ... Le plus simple est de remplir un triangle de Pascal. Si vous voulez être efficace, quitte à consommer de la mémoire, créez un gros triangle de Pascal une fois pour toute au début du programme.

On voit que cet axe peut être riche de prolongements maths/info.

3 En 2D

On adapte sans difficulté le principe en dimension 2. On crée cette fois une matrice `m` nulle de format $n \times n$, on l'initialise en exécutant `m[n//2][n//2] = N`, et on la fait évoluer en tirant pour chaque particule une de ses cases voisines au hasard. Le problème du bord est un peu plus pénible car il y a 8 types de bords différents ; il serait fastidieux

de traiter chaque cas séparément... Pour le tirage de la case voisine, je propose d'écrire d'abord une fonction renvoyant la liste des cases voisines.