TP 04

Percolation

1 Présentation

Pour ce TP, on utilisera le fichier `Percolation_Sujet.py`

La percolation [1] désigne le passage d'un fluide à travers un solide poreux. Ce terme fait bien entendu référence au café produit par le passage de l'eau à travers une poudre de café comprimée, mais dans un sens plus large peut aussi bien s'appliquer à l'infiltration des eaux de pluie jusqu'aux nappes phréatiques ou encore à la propagation des feux de forêt par contact entre les feuillages des arbres voisins.

L'étude scientifique des modèles de percolation s'est développée à partir du milieu du XXe siècle et touche aujourd'hui de nombreuses disciplines, allant des mathématiques à l'économie en passant par la physique et la géologie.

2 Choix d'un modèle

Nous allons aborder certains phénomènes propres à la percolation par l'intermédiaire du modèle suivant : une grille carrée $n \times n$, chaque case pouvant être aléatoirement ouverte ou fermée. La question à laquelle nous allons essayer de répondre est la suivante : est-il possible de joindre le haut et le bas de la grille par une succession de cases ouvertes adjacentes ?

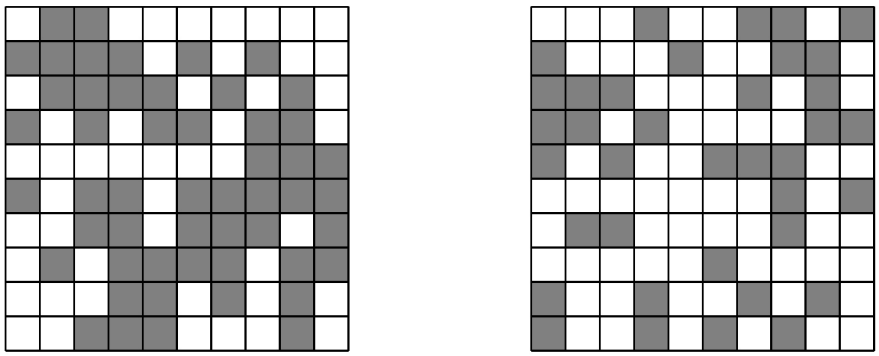

FIGURE 1 – Deux exemples de grilles 10×10. La percolation n'est possible que dans le second cas (les cases ouvertes sont les cases blanches).

Création et visualisation de grilles

La grille de percolation sera représentée par le type `list`. Cette grille sera elle-même composée de `n` listes de `n` flottants (tableau $n \times n$). Une fois la grille `grille` créée, la case d'indice (i, j) est référencée par `grille[i][j]`.

Si la case `grille[i][j]` est vide on lui affectera la valeur 0 (case ouverte, le fluide peut passer). Si la case `grille[i][j]` empêche le fluide de passer, on lui affectera la valeur 1 (case fermée).

Question 1 *Implémenter la fonction* `creation_grille_vierge(n:int) -> list` *permettant de créer un tableau de* `n` *lignes,* `n` *colonnes de 0. On utilisera pour cela deux boucles* `for` *imbriquées.*

1. du latin *percolare* : couler à travers.

L'instruction `creation_grille_vierge(3)` renverra le résultat suivant `[[0, 0, 0], [0, 0, 0], [0, 0, 0]]`.

On dispose de la fonction `afficher_grille(grille : list) -> None` permettant d'afficher une grille.

Question 2 *Donner les instructions permettant d'afficher une grille vierge de taille 3.*

Question 3 *Afficher la grille suivante* `[[1, 0, 1], [0, 1, 0], [1, 0, 1]]`.

Question 4 *Implémenter la fonction* `creation_zebreh(n:int) -> list` *permettant de créer une grille constituée de bandes horizontales noires puis blanches.* `creation_zebreh(3)` *renverra* `[[1, 1, 1], [0, 0, 0], [1, 1, 1]]`. *Valider votre résultat en utilisant la fonction* `afficher_grille`.

Question 5 *Implémenter la fonction* `creation_zebrev(n:int) -> list` *permettant de créer une grille constituée de bandes verticales noires puis blanches.* `creation_zebrev(3)` *renverra* `[[1, 0, 1], [1, 0, 1], [1, 0, 1]]`. *Valider votre résultat en utilisant la fonction* `afficher_grille`.

Question 6 *Implémenter la fonction* `creation_damier(n:int) -> list` *permettant de créer une grille en forme de damier.* `creation_damier(3)` *renverra* `[[1, 0, 1], [0, 1, 0], [1, 0, 1]]`. *Valider votre résultat en utilisant la fonction* `afficher_grille`.

On va maintenant créer une grille pour simuler la percolation. Afin de remplir la grille, on dispose de la fonction `rand()` de la bibliothèque `random` permettant de renvoyer un nombre aléatoire compris dans l'intervalle $[0,1[$.

On souhaite remplir une grille de taille `n` de la façon suivante :

- l'utilisateur choisit un nombre `p` compris entre 0 et 1 via un paramètre d'une fonction ;
- création d'une grille vide de taille `n` ;
- pour chaque élément de la grille :
  - on génère un nombre aléatoire `nb` compris dans l'intervalle $[0,1[$;
  - si `nb>p` alors l'élément est remplacé par la valeur 1.

Ainsi, si `p` vaut 1, la case aura 100 % de chance d'être ouverte. Si `p` vaut 0,25, la case aura 25 % de chance d'être ouverte.

Question 7 *Définir une fonction de signature* `def creation_grille(p: float, n: int) -> list` *à deux paramètres : un nombre réel p (qu'on supposera dans l'intervalle $[0,1]$) et un entier naturel n, qui renvoie un tableau (n,n) dans lequel chaque case sera ouverte (valeur 0) avec la probabilité p et fermée sinon (valeur 1).*

3 Percolation

Une fois la grille créée, les cases ouvertes de la première ligne sont remplies par un fluide, ce qui sera représenté par la valeur 2 dans les cases correspondantes. Le fluide pourra ensuite être diffusé à chacune des cases ouvertes voisines d'une case contenant déjà le fluide jusqu'à remplir toutes les cases ouvertes possibles.

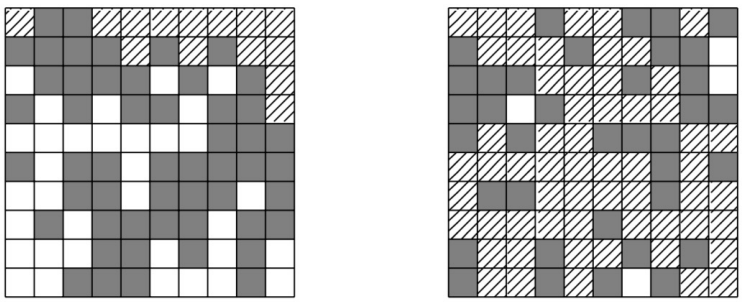

FIGURE 2 – les deux grilles de la figure 1, une fois le processus de percolation terminé (le fluide est représenté par des hachures).

3.1 Percolation « naïve »

Dans un premier temps, nous allons créer une fonction intermédiaire permettant de savoir si, pour une case de coordonnées `[i,j]` de la grille, un de ses voisins est rempli (valeur 2) ou non.

Question 8 *Écrire une fonction* `a_un_voisin_rempli(grille:list,i:int,j:int) -> bool` *retournant* `True` *si un des voisins est rempli,* `False` *si aucun des voisins n'est rempli.*

Question 9 *Écrire une fonction* `percolation(grille : list) -> None` *qui prend en argument une grille et qui remplit de fluide celle-ci, en appliquant l'algorithme exposé ci-dessous :*

- *pour la première ligne, si la case est ouverte on la remplit (passage de la valeur* `0` *à la valeur* `2`*);*
- *pour les cases des lignes suivantes : si la case est ouverte et qu'un voisin est rempli, alors on remplit cette case.*

Question 10 *En affichant la grille avant et après percolation avec différentes valeurs de* `p`*, visualiser que l'algorithme ne répond pas complètement à la problématique. Modifier la fonction précédente en conséquence.*

3.2 Percolation un peu plus évoluée – Facultatif

Question 11 *Écrire une fonction* `percolation(grille : list) -> None` *qui prend en argument une grille et qui remplit de fluide celle-ci, en appliquant l'algorithme exposé ci-dessous :*

1. *Créer une liste contenant initialement les coordonnées des cases ouvertes de la première ligne de la grille et remplir ces cases de liquide.*
2. *Puis, tant que cette liste n'est pas vide, effectuer les opérations suivantes :*
   (a) *extraire de cette liste les coordonnées d'une case quelconque ;*
   (b) *ajouter à la liste les coordonnées des cases voisines qui sont encore vides, et les remplir de liquide.*

L'algorithme se termine quand la liste est vide.

3.3 Validation de la percolation

Question 12 *Rédiger un script vous permettant de visualiser une grille avant et après remplissage, et faire l'expérience avec quelques valeurs de* p *pour une grille de taille raisonnable (commencer avec* $n = 10$ *pour vérifier visuellement que votre algorithme est correct, puis augmenter la taille de la grille, par exemple avec* $n = 64$*). On pourra l'exporter et l'enregistrer sous le nom* `tp_n_q03_vos_noms.png`*.*

On dit que la percolation est réussie lorsqu'à la fin du processus au moins une des cases de la dernière ligne est remplie du fluide.

Question 13 *Écrire une fonction* `teste_percolation(p : float, n : int) -> bool` *qui prend en argument un réel* $p \in [0,1]$ *et un entier* $n \in \mathbb{N}^*$*, crée une grille, effectue la percolation et renvoie :*

- `True` lorsque la percolation est réussie, c'est-à-dire lorsque le bas de la grille est atteint par le fluide ;
- `False` dans le cas contraire.

4 Seuil critique

Nous allons désormais travailler avec des grilles de taille 128×128 [2]

Faire quelques essais de percolation avec différentes valeurs de p. Vous observerez assez vite qu'il semble exister un seuil p_0 en deçà duquel la percolation échoue presque à chaque fois, et au delà duquel celle-ci réussit presque à chaque fois. Plus précisément, il est possible de montrer que pour une grille de taille infinie, il existe un seuil critique p_0 en deçà duquel la percolation échoue toujours, et au delà duquel la percolation réussit toujours. Bien évidemment, plus la grille est grande, plus le comportement de la percolation tend à se rapprocher du cas de la grille théorique infinie.

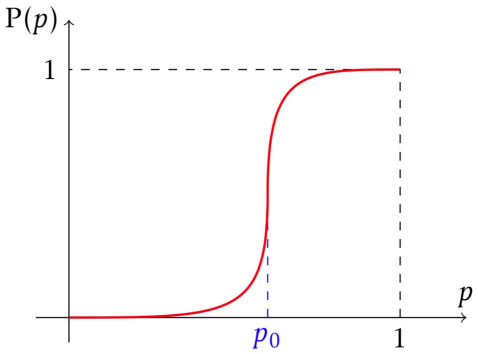

FIGURE 3 – L'allure théorique du graphe de la fonction $P(p)$.

Notons $P(p)$ la probabilité pour le fluide de traverser la grille. Pour déterminer une valeur approchée de la probabilité de traverser la grille, on se contente d'effectuer k essais pour une valeur de p puis de renvoyer le nombre moyen de fois où le test de percolation est vérifié.

Question 14 *Rédiger la fonction* `proba(p : float, k : int, n : int) -> float` *qui prend en argument le nombre d'essai k, la variable p ainsi que le nombre de cases n sur la largeur de la grille et qui renvoie* $P(p)$.

La fonction suivante prend en argument une taille n et affiche le graphe obtenu. On pourra traiter le cas d'une grille de 128×128 cases.

```
import numpy as np
import matplotlib.pyplot as plt
def tracer_proba(n):
    x=np.linspace(0,1,21)
    y=[]
    for p in x:
        y.append(proba(p,20,n))
    plt.clf()
    plt.plot(x,y)
    plt.show()
    return None
```

2. Baisser cette valeur si le temps de calcul sur votre ordinateur est trop long.