TD 9 : Automates cellulaires

PC* - Lycée Thiers

Problème 1 : Epreuve type Mines-Ponts

Enoncé

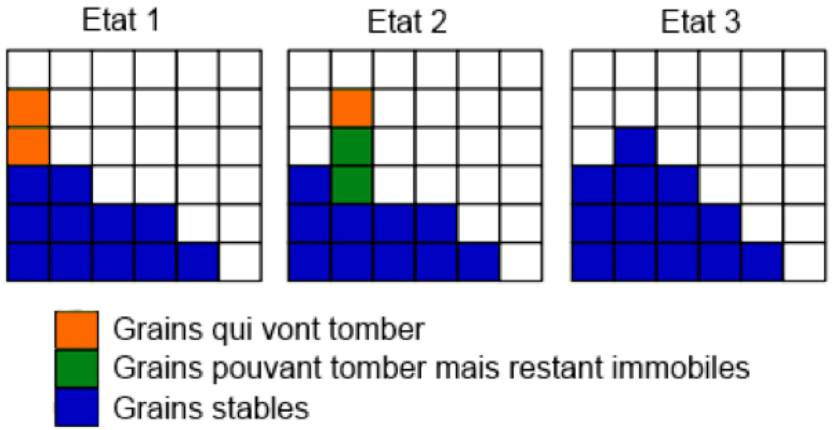
Problème 2 : X-ENS PSI-PT 2016

Problème 1 : Epreuve type Mines-Ponts

L'étude des écoulements granulaires est un domaine scientifique en pleine expansion. Assez solides pour soutenir le poids d'un immeuble, ils peuvent couler comme de l'eau dans un sablier ou être transportés par le vent pour sculpter les dunes et les déserts.
On appelle milieu granulaire une collection de particules solides macroscopiques, typiquement de taille supérieure à 100 μm. Les particules plus fines sont appelées poudres (entre 1 μm et 100 μm ou colloides (entre 1 nm et 1 μm. Pour étudier ces milieux, on néglige en général les interactions de van der Waals, les effets d'humidité et l'agitation thermique.

I. Calcul de la forme d'un tas par automates cellulaires L'objet de cette partie est la simulation numérique par un modèle de type automate cellulaire. Un automate cellulaire se présente généralement sous la forme d'un quadrillage dont chaque case peut être occupée ou vide. Un grain y est symbolisé par une case occupée. La configuration des cases, qu'on appelle état de l'automate, évolue au cours du temps selon certaines règles très simples permettant de reproduire des comportements extrêmement complexes. La physique n'intervient pas directement mais les règles d'évolution sont choisies de façon à reproduire au mieux les lois naturelles. Dans ce qui suit, nous allons simuler la formation d'un demi-tas de sable situé à droite de l'axe de symétrie vertical en appliquant les règles énoncées ci-après.

Figure 1. Exemple de l'automate cellulaire à implémenter



Une *tour* est une pile verticale de cases pleines consécutives (de grains de sables). Sa *hauteur utile*, notée h, est le nombre de cases voisines de droite qui sont vides. Par exemple la tour à gauche dans l'état 1 a pour hauteur utile $h = 2$, et dans l'état 2, hauteur utile $h = 0$.

Si $h > 1$, on détermine arbitrairement le nombre n de grains du sommet de la tour qui vont tomber avec l'instruction python :

```
n = int((h+2.0)/2 * random()) + 1
```

Dans l'exemple figure 3, lors du passage de l'état 1 à l'état 2, le hasard introduit par la fonction `random()` (qui renvoie de manière aléatoire un flottant dans l'intervalle semi-ouvert $[0.0, 1.0[$) fait que toute la tour se décale apparemment vers la droite. Ensuite, pour le passage à l'état 3 seul un grain sur les trois possibles tombe. L'état 3 est stable, plus aucun grain ne tombe.

1. Quel est le type de `n` ? Déterminer un encadrement de `n` pour `h` > 1.

Réponse : La variable `n` est de type entier (`int`), car obtenue par addition de deux valeurs de types entiers : `int((h+2)/2 * random())` et 1.

Lorsque $h > 1$; `random()` retourne un nombre dans $[0, 1[$ et `n = int((h+2)/2 * random()) +1`. Ainsi :
Lorsque h est pair : $1 \leqslant n \leqslant h/2 + 1$.
Lorsque h est impair : $1 \leqslant n \leqslant (h+1)/2 + 1$.
Dans tous les cas :

$$\boxed{1 \leqslant n \leqslant \left\lceil \frac{h}{2} \right\rceil + 1}$$

où $\lceil x \rceil$ désigne la partie entière par excès, $\lceil x \rceil = -\lfloor -x \rfloor$, c'est à dire le plus petit entier supérieur au réel x.

2. Ecrire une fonction nommée `calcul_n` qui prend la hauteur `h` comme argument et qui renvoie le nombre `n` de grains qui vont tomber sur la pile suivante.

Réponse :

```
from random import random

def calcul_n(h) :
    if h > 1 :
        return int((h+2.0)/2 * random()) + 1
    else :
        return 0
```

La représentation graphique est une image en deux dimensions mais l'automate est à une dimension car on considère le tas comme un ensemble de piles dont la hauteur future dépend uniquement des piles adjacentes. Le tas est complètement défini avec la variable piles qui permet de stocker la hauteur des différentes piles. Au départ le support est vide et on vient déposer périodiquement un grain sur la première pile (à gauche) qui correspondra au sommet du demi-tas. Le support peut recevoir P piles et à son extrémité droite il n'y a rien, les grains tombent et sont perdus. La pile $P+1$ est donc toujours vide.

3. Définir la fonction `initialisation(P)` renvoyant une variable `piles` de type liste contenant P piles de hauteur 0.

Réponse :

```
def initialisation(P) :
    piles = [0] * P
    return piles
```

4. Définir une fonction `actualise(piles,perdus)` qui va parcourir les piles de gauche à droite et les faire évoluer en utilisant les règles, fonctions et variables définies précédemment. Cette fonction doit renvoyer la variable `piles` actualisée ainsi que le nombre TOTAL de grains perdus (en ajoutant à `perdus` le nombre de grains tombés de la dernière pile).

```
def actualise(piles,perdus):
    P = len(piles)
    for k in range(P-1):
        n = calcul_n(piles[k]-piles[k+1])
        piles[k] -= n
        piles[k+1] += n
    n = calcul_n(piles[P-1])
    piles[P-1] -= n
    perdus += n
    return piles, perdus
```

5. Ecrire le bloc d'instructions du programme principal qui permet d'ajouter 1 grain à la première pile après chaque dizaine d'exécutions de la fonction `actualise(,)` et qui s'arrêtera lorsqu'au moins 1000 grains seront sortis du support. Le nombre de piles (taille du support P) sera demandé à l'utilisateur lors de l'exécution. Vous utiliserez les fonctions et variables définies précédemment.

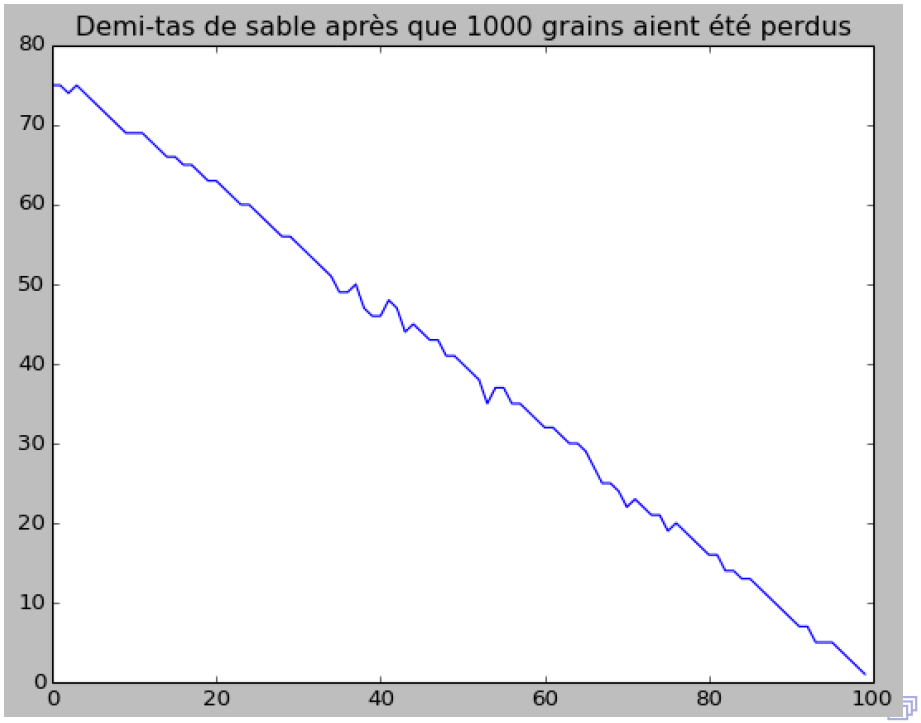
```
P = int(input('Saisissez le nombre de piles : '))
piles = initialisation(P)
perdus = 0
while perdus < 1000:
    piles[0] += 1
    for k in range(10):
        piles, perdus = actualise(piles, perdus)
```

6. Comment tracer simplement la forme (allure) du demi-tas à la fin de la simulation précédente.

```
import matplotlib.pyplot as plt

plt.figure(1)
plt.title('Demi-tas de sable après que 1000 grains aient été perdus')
plt.plot(piles)
plt.show()
```

Ce qui donnera par exemple, avec 100 piles :



Problème 2 : X-ENS PSI-PT 2016

Détection de collisions entre particules

On considère un ensemble de n particules en mouvement dans un espace à deux dimensions, délimité par un rectangle de dimensions (non nulles) $largeur \times hauteur$. L'objectif est de faire évoluer le système jusqu'à ce que deux particules entrent en collisions.

I. Simulation du mouvement des particules On considère que le temps est discret. La simulation commence au temps $t = 0$, et à chaque étape, on calcule la configuration au temps $t + 1$ en fonction de la configuration au temps t.
A tout instant t donné, chaque particule est définie par un quadruplet (x, y, vx, vy), où (x, y) sont ses coordonnées réelles représentées par des nombres flottants et où (vx, vy) est son vecteur vitesse, lui aussi constitué de deux nombres flottants.
Dans tout le sujet, on suppose que la norme de la vitesse de toute particule est majorée par une constante $vmax$.

Pour calculer les paramètres au temps $t+1$ d'une particule qui, au temps t, est en position (x, y) avec un vecteur vitesse (vx, vy), on procède successivement aux traitements suivants (un exemple d'exécution est donné en fin de partie I) :

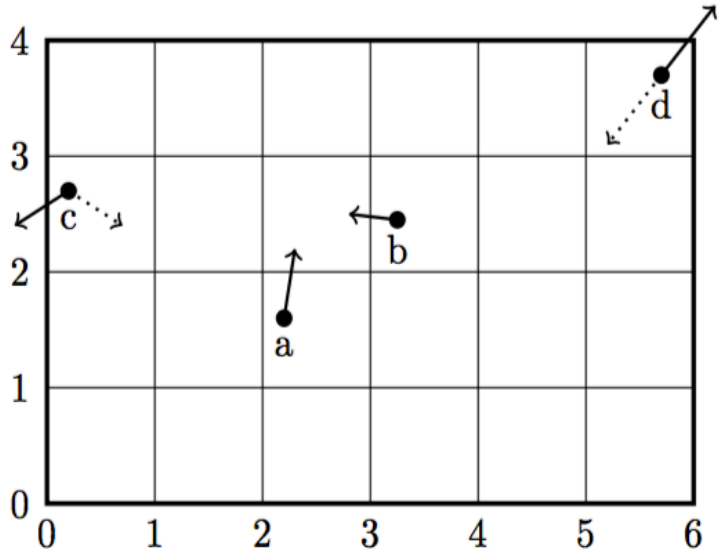
(i) si $x + vx$ atteint ou dépasse une paroi verticale, vx est changé en $-vx$ pour simuler le rebond ;

(ii) si $y + vy$ atteint ou dépasse une paroi horizontale, vy est changé en $-vy$ pour simuler le rebond ;

(iii) (x, y) est changé en $(x + vx, y + vy)$.

Les points (i) et (ii) simulent de façon simplifiée les rebonds sur les parois : on considère que la particule rebondit à l'endroit où elle est au temps t, ce qui nous permet d'éviter de calculer le véritable point de collision avec la paroi. Il y a rebond lorsqu'une particule arrive exactement sur la paroi ou qu'elle la dépasse. Il est possible qu'une particule rebondisse sur une paroi verticale et une horizontale pendant une même mise à jour, ce qui correspond au rebond sur un coin.

Important : Au départ, aucune particule n'est sur la paroi. On suppose de plus que $vmax < \frac{1}{2}\min(largeur, hauteur)$, ce qui garantit que les particules restent toujours strictement à l'intérieur des parois.



Dans l'exemple ci-dessus, le rectangle est de dimension *largeur* $\times$ *hauteur* $= 6 \times 4$. Les particules a et b se déplacent sans rebondir au temps $t+1$. La particule c est sujette au point (i), comme $x + vx \leqslant 0$, elle rebondit sur la paroi, ce que l'on simule en changeant vx en $-vx$ avant d'effectuer le déplacement (le nouveau vecteur vitesse est représenté en pointillés). La particule d est sujette aux deux points (i) et (ii), puisque $x + vx \geqslant$ *largeur* et $y + vy \geqslant$ *hauteur*, on change donc vx en $-vx$ et vy en $-vy$ avant de déplacer cette particule.

1. Ecrire une fonction `deplacerParticule(particule, largeur, hauteur)` **qui prend en paramètre une particule à l'instant** t **ainsi que les dimensions du rectangle et renvoie la particule à l'instant** $t+1$**, en tenant compte des rebonds.**

Exemples d'exécution de cette fonction :

```
>>> deplacerParticule((2.2, 1.6, 0.1, 0.6), 6, 4)
(2.3, 2.2, 0.1, 0.6)
>>> deplacerParticule((3.25, 2.45, 0.45, 0.05), 6, 4)
(3.7, 2.5, 0.45, 0.05)
>>> deplacerParticule((0.2, 2.7, -0.5, -0.3), 6, 4)
(0.7, 2.4, 0.5, -0.3)
>>> deplacerParticule((5.7, 3.7, 0.5, 0.6), 6, 4)
(5.2, 3.1, -0.5, -0.6)
```

```
def deplacerParticule(particule, largeur, hauteur):
    x,y,vx,vy = particule
    if not(0 < x+vx < largeur):
        vx = -vx
    if not(0 < y+vy < hauteur):
        vy = -vy
    return (x+vx,y+vy,vx,vy)
```

II. Représentation par une grille

Dans un premier temps, on décide de représenter un ensemble de particules par une grille. Une particule de coordonnées (x,y) se trouvera obligatoirement dans la case d'indices $\lfloor x \rfloor$, $\lfloor y \rfloor$; en Python, on obtient la partie entière d'un flottant x positif ou nul en utilisant la fonction `int(x)`.

Comme nous avons vu qu'une particule se trouve toujours à l'intérieur du rectangle et jamais sur les parois, cette simplification n'entraine aucun débordement de tableau. Une case de la grille ne peut contenir qu'une seule particule : si deux particules ou plus devaient aboutir dans la même case, on considère qu'il y a une collision et la simulation se termine. Pour indiquer qu'une case est vide (sans particule), on utilisera `None`.

En Python, cette grille sera représentée par un tableau à deux dimensions (le nombre de lignes correspond à la largeur et le nombre de colonnes à la hauteur de la grille). Il s'agit donc d'un tableau de tableaux, ou plus précisément d'un tableau de longueur *largeur* contenant dans chaque case une colonne, qui est elle-même un tableau de longueur *hauteur*. La case d'indices `[i][j]` dans ce tableau correspondra ainsi à la case de cordonnées (i, j) dans la grille.

2. Ecrire une fonction `nouvelleGrille(largeur, hauteur)` qui renvoie une nouvelle grille vide de dimensions *largeur* × *hauteur*.

```
def nouvelleGrille(largeur,hauteur):
    G = []
    for i in range(largeur):
        L = []
        for j in range(hauteur):
            L.append(None)
        G.append(L)
    return G
```

ou de façon plus concise, par compréhension de liste :

```
def nouvelleGrille(largeur,hauteur):
    return [[None * hauteur] for i in range(largeur)]
```

ATTENTION, SURTOUT PAS :

```
    return [[None * hauteur]] * largeur
```

autrement toutes les lignes seraient identiques : la modification de l'une modifierait les autres.

3. Pour cette partie, on considère qu'une collision entre deux particules survient si elles arrivent dans la même case de la grille à un instant donné.

Ecrire une fonction nommée `majGrilleOuCollision(grille)` qui prend en paramètre une grille contenant des particules à l'instant t et renvoie une nouvelle grille contenant ces particules à l'instant $t+1$ s'il n'y a pas eu de collision.

Si une collision survient, la fonction renvoie `None`.

Remarque : Attention à ne pas confondre les particules à l'instant t avec celles à l'instant $t+1$.

```
def majGrilleOuCollision(grille):
    L, H = len(grille), len(grille[0])
    G = nouvelleGrille(L, H)
    for i in range(L):
        for j in range(H):
            particule = grille[i][j]
            x,y,vx,vy = deplacerParticule(particule, L, H)
            if G[int(x)][int(y)] != None:
                return None
            else:
                G[int(x)][int(y)] = (x,y,vx,vy)
    return G
```

4. Ecrire une fonction `attendreCollisionGrille(grille, tMax)` qui prend une grille de particules en paramètre et renvoie le temps où a eu lieu la première collision entre deux particules. S'il n'y a pas de collision avant le temps *tMax*, la fonction renvoie `None`.

```
def attendreCollisionGrille(grille, tMax):
    for t in range(1,tMax):
        grille = majGrilleOuCollision(grille,tMax)
        if grille == None:
            return t
    return None
```

5. Quelle est la complexité de la fonction `attendreCollisionGrille(grille, tMax)` en fonction des dimensions (*largeur* et *hauteur*) de la grille et de *tMax* ? La réponse devra être justifiée.

- `deplacerParticule(particule, largeur, hauteur)` est de complexité $O(1)$ dans tous les cas.
- `nouvelleGrille(largeur, hauteur)` est de complexité $O(\mathit{largeur} \times \mathit{hauteur})$ dans tous les cas.
- `majGrilleOuCollision(grille)` est de complexité $O(\mathit{largeur} \times \mathit{hauteur})$ dans tous les cas.

Ainsi : `attendreCollisionGrille(grille,tMax)` est de complexité :

- $O(\mathit{largeur} \times \mathit{hauteur})$ dans le meilleur des cas (celui d'une collision au temps $t = 1$).
- $O(\mathit{tMax} \times \mathit{largeur} \times \mathit{hauteur})$ dans le pire des cas (celui ou il n'y a pas collision avant $t = \mathit{tMax}$).