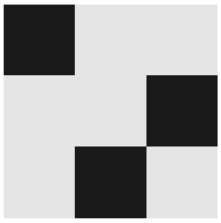
Devoir à la Maison n°2

Le jeu de la vie

Le *jeu de la vie* est une simulation imaginée par John H. Conway (1937-2020) en 1970.

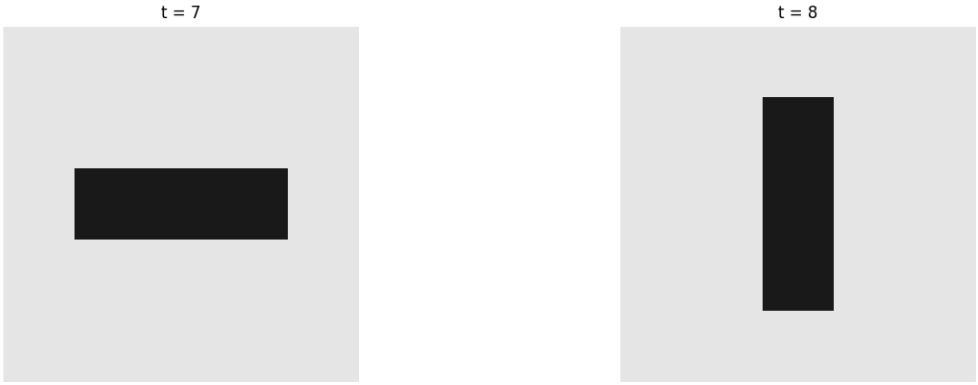
Il se joue sur une grille rectangulaire. Chaque case de cette grille, appelée *cellule*, peut prendre deux états distincts : *vivante* ou *morte*. En général, les cellules vivantes sont représentées en noir et les cellules mortes en blanc.

Chaque cellule possède huit voisines ; ce sont les cellules adjacentes horizontalement, verticalement et diagonalement :



À chaque étape, l'état d'une cellule dépend de celui de ses voisines :

- une cellule morte ayant exactement trois voisines vivantes devient vivante (elle *naît*),
- une cellule vivante ayant deux ou trois voisines reste vivante ; sinon, elle meurt.



Le but de ce devoir est de programmer et de tester ce jeu.

Le langage utilisé doit être Python, il faut donc tester les fonctions sur un ordinateur sur lequel une distribution de Python est installée. Consulter pour ceci le site `prepabellevue.org`, rubrique `PCSI2/Informatique/Installation de Python`.

Il est important de commenter les lignes de code.

Partie A. Créer la vie

1. Importer les modules `matplotlib.pyplot` et `random`, ainsi que la fonction `deepcopy` du module `copy`.

Cette dernière fonction permet de copier des listes de listes sans créer d'interférence.

▷ Comparer le résultat de ces deux scripts :

```
A=[[0,0],[1,1]]
B=A.copy()
A[1][1]=5
print(A,B)
```

```
A=[[0,0],[1,1]]
B=deepcopy(A)
A[1][1]=5
print(A,B)
```

Commençons par créer le temps et l'espace. Le temps est simplement un nombre entier `t`, et l'espace est une liste de listes `M` de taille $N \times N$, dont chaque élément est une cellule.

Une cellule morte est représentée par une case blanche `b=[0.9,0.9,0.9]`, et une cellule vivante est représentée par une case noire `n=[0.1,0.1,0.1]` (les couleurs sont ici codées en RGB). Lorsque toutes les cellules sont mortes, on dit que l'espace est *vide*. Ainsi, l'espace vide de taille 2×2 est `M=[[b,b],[b,b]]`.

2. Définir `t=0`, et `M` un espace vide de taille $N \times N$, où $N = 100$.

Pour représenter `M`, on utilise la commande `imshow` du module `matplotlib.pyplot` :

```
imshow(M)
axis("off")  #pour effacer les axes
show()
```

3. Afficher l'espace vide.

4. Changer 1000 cellules mortes, choisies aléatoirement, en cellules vivantes. On utilisera la fonction `randint` du module `random`. Afficher le résultat.

On souhaite pouvoir changer l'état d'une cellule en cliquant dessus. On utilise pour cela la fonction `connect` du module `matplotlib.pyplot` (à placer avant `imshow`) :

```
def souris(clic):
    i, j = int(clic.ydata), int(clic.xdata)
    #pour des raisons d'affichage, abscisses et ordonnées sont inversées
    print(i,j)
connect('button_press_event', souris)
```

5. Modifier la fonction `souris` pour qu'elle change l'état de la cellule `M[i][j]` (de vivante à morte ou inversement). Tester.

Partie B. Animer la vie

On considère qu'une cellule `M[i][j]` a pour voisines les huit cellules de coordonnées ($i \pm 0$ ou 1, $j \pm 0$ ou 1) *modulo* N. Ceci revient à connecter les bords de notre espace.

6. Créer une fonction `nbvoisines(A,i,j)` prenant en argument un espace `A` de taille $N \times N$, et renvoyant le nombre de voisines vivantes de la cellule `A[i][j]`.

7. Créer une fonction `generation()` qui fait passer l'espace `M` d'une étape à la suivante en suivant les règles du jeu de la vie. On commencera par créer une copie de `M`, sur laquelle on comptera les voisines vivantes. Cette fonction ne renvoie rien.

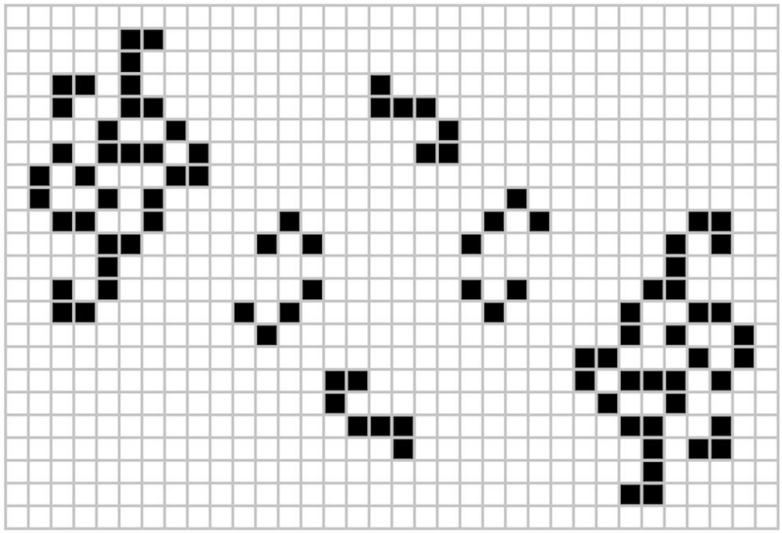
On peut alors afficher quelques étapes du jeu à l'aide du script suivant :

```
for t in range(10):
    generation()
    clf()    #efface l'affichage courant
    imshow(M)
    title('t = '+str(t))    #titre de l'étape
    axis("off")
    pause(10**-2)    #temps de pause entre deux étapes
show()
```

8. Implémenter ce script. Observer.

Malgré leur simplicité, les règles du jeu de la vie permettent l'apparition de structures complexes. On parle d'*émergence*.

Certaines de ces structures se déplacent, d'autres sont périodiques, comme le *clignotant* présenté en page 1, de période 2. Depuis juillet 2023, on sait construire des structures n-périodiques pour tout n !



La structure *Cribbage*, 19-périodique, découverte par Mitchell Riley en juillet 2023.

Partie C. Contrôler la vie

On souhaite pouvoir contrôler notre simulation à l'aide du clavier de l'ordinateur. On utilise à nouveau la fonction `connect` :

```
def clavier(tape):
    if tape.key=='q':    #si on appuie sur la touche q
        close()
connect('key_press_event', clavier)
imshow(M)
axis("off")
show()
```

▷ Tester ce script.

En plus de la fonction `close()`, déjà définie dans Python, on crée trois fonctions :

- la fonction `disparition()` vide l'espace `M`,
- la fonction `apparition()` remet le temps à 0 et remplit l'espace `M` de 1000 cellules vivantes aléatoires,
- la fonction `evolution()` incrémente le temps et applique la fonction `generation()`.

Chacune de ces fonctions suivra le schéma suivant :

```
def ...():
    global t    #indique que les variables t et M sont globales
    global M
    ...
    clf()
    imshow(M)
    title('t = '+str(t))
    axis("off")
    pause(10**-2)
```

9. Écrire ces trois fonctions, et les associer aux touches `z`, `r` et `e` du clavier.

10. Créer une variable globale `stop`. Associer :

- à la touche `espace` l'assignation `stop=`**True**,
- à la touche `d` l'assignation `stop=`**False** puis la fonction `evolution()` en boucle jusqu'à ce qu'on appuie sur la touche `espace`.

11. Rajouter la connexion de la souris définie à la question 5. Profiter !