TP N° 1 : Jeu de la vie (TP Noté)

Pour ce travail vous devez déposer un lien **unique** du dépôt `github` contenant l'intégralité du travail effectué. Le travail pour ce TP est **individuel**. Les TPs indiquant une trop grande ressemblance sur Compilatio seront pénalisés.

Le dépôt `github` contiendra obligatoirement :

— un fichier `README.md` pour la page d'accueil décrivant succinctement votre travail.

— un fichier `HMMA238_TP_prenomnom.ipynb` contenant un notebook avec les réponses aux questions (à la fois code / questions mathématiques) et où `prenomnom` sera votre prénom concaténé avec votre nom (sans majuscules, ni espaces ni accents).

— un fichier `utils.py` qui sera chargé par votre notebook, et qui contiendra certaines fonctions d'affichages ou de calcul élémentaire.

Le projet doit être terminé pour le vendredi 13 mars 22h22. Les contributions faites sur le dépôt après cette heure ne seront pas prises en compte pour la note. La note totale est sur **20** points, répartis comme suit :

— qualité des réponses aux questions : **14** pts

— qualité du dépôt `git`, commentaires-commits etc. : **1** pt

— qualité de rédaction et d'orthographe : **1** pt

— qualité des graphiques (légendes, couleurs, titres, etc.) : 1 pt

— style PEP8[1] respecté : **1** pt

— qualité du code (noms de variable clairs, commentaires adéquates, code synthétique, etc.) : **1** pt

— Notebook reproductible (i.e., “Restart & Run all” marche correctement sur la machine du correcteur) et absence de bug : **1** pt

Exercice 1. Le jeu de la vie

1) (**0.5pt**) Créer la chaîne de caractères `filename` correspondant au nom de votre fichier et qui aura le format suivant : `filename=HMMA238_TP_prenomnom.ipynb` (le tout en minuscule et sans accents ni espace).

2) (**0.5pt**) Créer une variable `taille_str` qui compte le nombre de caractères dans la chaîne de caractères `filename`.

3) (**0.5pt**) Créer une variable `ma_graine` qui vaut le reste de la division euclidienne de `taille_str` par 6 (rem : pour “Joseph Salmon” ce nombre vaut 5, et pour Benjamin Charlier il vaut 3).

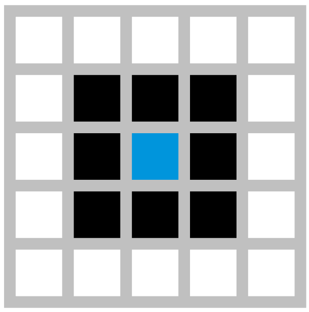


Figure 1 – Cellule (au centre, en bleu), et ses voisins (autour, en noir)

1. https://pep8.org/

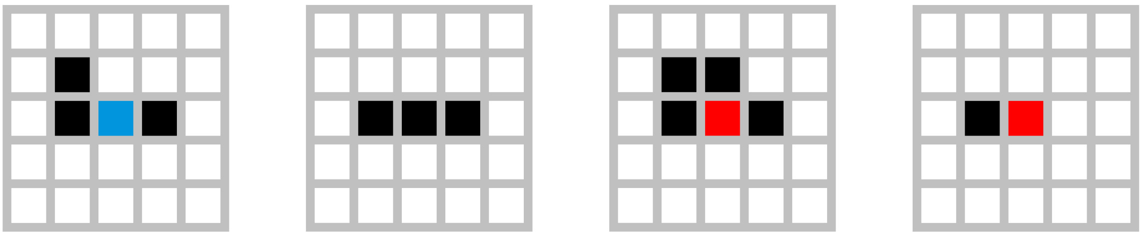
Le **jeu de la vie**[2] est un automate cellulaire mis au point par le mathématicien britannique John Horton Conway en 1970. Il constitue l'exemple le plus connu d'un automate cellulaire. Le "jeu" est en fait un jeu à zéro joueur, ce qui signifie que son évolution est déterminée par son état initial et ne nécessite aucune intervention de la part d'un humain. On interagit avec le jeu de la vie en créant une configuration initiale ; il ne reste plus alors qu'à observer son évolution.

L'univers du jeu est initialement une grille orthogonale bidimensionnelle infinie de cellules carrées. Dans la suite du projet on supposera cependant que la grille est carrée et de taille finie pour éviter toute difficulté. **On supposera aussi que le pourtour de la grille est toujours inactif/mort**.

Les cellules du jeu ne peuvent prendre qu'un état parmi l'un des deux états possibles : **vivant** (1) ou **mort** (0).

Chaque cellule interagit avec ses huit cellules voisines (ce sont les cellules directement adjacentes horizontalement, verticalement ou en diagonale), comme indiqué sur la Figure 1. À chaque étape, les transitions suivantes se produisent :

a) Toute cellule morte ayant exactement 3 voisins vivants devient une cellule vivante (**naissance**), *cf.* Figure 2a,
b) Toute cellule vivante avec 2 ou 3 voisins vivants reste vivante à la génération suivante (**équilibre**), *cf.* Figure 2b,
c) Toute cellule vivante ayant 4 voisins vivants meurt à la génération suivante (**mort par étouffement**), *cf.* Figure 2c,
d) Toute cellule vivante ayant 0 ou 1 voisin vivant décède à la génération suivante (**mort par isolement**), *cf.* Figure 2d.



(a) Naissance (b) Équilibre (c) Étouffement (d) Isolement

FIGURE 2 – Quatre configurations possibles pour la cellule centrale : a) naissance (bleu), b) équilibre, c) mort par étouffement et d) par isolement (rouge).

Le modèle initial constitue la "graine" du système. La première génération est créée en appliquant les règles ci-dessus simultanément à chaque cellule de la graine - les naissances et les décès se produisent simultanément. Ainsi chaque génération est une fonction de la précédente. Les règles continuent d'être appliquées de manière répétée pour créer d'autres générations.

Implémentation sans `numpy`

On va fournir dessous le code pure Python pour coder ce jeux. Dans la suite on va coder les cellules vivantes par des 1 et les cellules mortes par des 0. Tout d'abord on définit la fonction `calcul_nb_voisins` :

```python
def calcul_nb_voisins(Z):
    forme = len(Z), len(Z[0])
    N = [[0, ] * (forme[0]) for i in range(forme[1])]
    for x in range(1, forme[0] - 1):
        for y in range(1, forme[1] - 1):
            N[x][y] = Z[x-1][y-1]+Z[x][y-1]+Z[x+1][y-1] \
                    + Z[x-1][y] + 0          +Z[x+1][y]  \
                    + Z[x-1][y+1]+Z[x][y+1]+Z[x+1][y+1]
    return N
```

2. Plus de détails pour les curieux : https://fr.wikipedia.org/wiki/Jeu_de_la_vie

4) (**1pt**) Appliquer la fonction précédente à la liste (de liste) `Z` suivante, et expliquer ce que représente la sortie obtenue `N=calcul_nb_voisins(Z)`.

```
Z = [[0,0,0,0,0,0],
     [0,0,0,1,0,0],
     [0,1,0,1,0,0],
     [0,0,1,1,0,0],
     [0,0,0,0,0,0],
     [0,0,0,0,0,0]]
```

Définir ensuite la fonction `iteration_jeu` comme suit : et rajouter une *docstring* pour cette fonction décrivant les entrées / sorties et ce que retourne la fonction :

```python
def iteration_jeu(Z):
    forme = len(Z), len(Z[0])
    N = calcul_nb_voisins(Z)
    for x in range(1,forme[0]-1):
        for y in range(1,forme[1]-1):
             if Z[x][y] == 1 and (N[x][y] < 2 or N[x][y] > 3):
                 Z[x][y] = 0
             elif Z[x][y] == 0 and N[x][y] == 3:
                 Z[x][y] = 1
    return Z
```

5) (**2pts**) Dans cette question on se propose pour la liste `Z` ci-dessus d'afficher les étapes du jeu de 0 à 9 itérations, en utilisant une boucle `for`. On utilisera la fonction `subplot` de `matplotlib` pour afficher sur 2 lignes et 5 colonnes ces 10 matrices. De plus on devra transformer ces listes en array pour pouvoir utiliser la fonction `imshow` de `matplotlib`.

6) (**1pt**) Que remarquez-vous entre l'itération 0 et l'itération 4 ? Que se passe-t-il après l'itération 7 ?

Implémentation avec `numpy`

7) (**1pt**) Pour le vecteur `vec` suivant, exprimer ce que vaut le vecteur `nb_vect` défini comme suit.

```python
vect = np.array([0,1,0,0,1,1])
nb_vect = np.zeros(vect.shape)
nb_vect[1:-1] += (vect[:-2] + vect[2:])
```

Rem : on ne se s'intéresse pas au bord du vecteur, tout comme on ne intéresse pas au bord de la matrice représentant le jeu de la vie.

8) (**1pt**) Se servir de la question précédente (et du `slicing` donc) pour créer une fonction `calcul_nb_voisins_np`, cette fois sur des `array`, qui prend en entrée une matrice `Z` et qui ressort le nombre de voisins pour chaque entrée (et qui vaut zéro sur le pourtour). On utilisera donc 8 types de *slicing* pour obtenir le nombre de voisins.

9) (**1pt**) Créer une fonction `iteration_jeu_np`, similaire à `iteration_jeu` mais qui prend comme entrée sortie des `numpy array` et non plus des listes de listes. On se servira bien sûr de la question précédente pour cela.

10) (**0.5pt**) Créer une fonction `jeu_np` qui prend en entrée une matrice initiale `Z_in` et un nombre d'itérations `nb_iter` et sort une matrice (de même taille que `Z_in`) décrivant l'état du jeu de la vie après `nb_iter` itérations.

11) (**2pts**) Afficher un film (avec la commande `animation.FuncAnimation` de `matplotlib`) qui représente les itérations du jeu de la vie quand on initialise avec la matrice `Z_huge` suivante :

```
Z_huge = np.zeros((100, 100))
Z_np = np.array(
    [[0, 0, 0, 0, 0, 0],
     [0, 0, 0, 1, 0, 0],
     [0, 1, 0, 1, 0, 0],
     [0, 0, 1, 1, 0, 0],
     [0, 0, 0, 0, 0, 0],
     [0, 0, 0, 0, 0, 0]])
Z_huge[10:16, 10:16] = Z_np
```

On pourra consulter notamment l'exemple suivant :
https://jakevdp.github.io/blog/2012/08/18/matplotlib-animation-tutorial/, ainsi que l'aide de `matplotlib`. Il pourra être utile aussi pour cette question d'utiliser la commande magique : `%matplotlib notebook` pour l'affichage des vidéos (ou bien d'utiliser `from IPython.display import HTML`).

12) (**1pt**) Reprendre la question précédente en initialisant de la manière suivante : créer un matrice aléatoire de taille 100×100, remplie de 1 et de 0, et dont la proportion de 1 est (en espérance) égale à `(1 + ma_graine) * 10 / 100` (*e.g.*, pour Joseph Salmon ce nombre vaut `prop_active = 0.6`, et pour Benjamin Charlier, ce nombre vaut `prop_active = 0.4`).

13) (**1pt**) Proposer et afficher avec `plt.imshow()` des matrices (et 10 itérations d'un jeu initialisé avec elles) de taille 50×50, ayant les propriétés suivantes :

   — trois matrices simples qui représentent des jeux qui sont fixes dans le temps (configuration stables)

   — une matrice qui représente un jeu dont l'état oscille avec une période de deux (et qui ne comporte pas uniquement des valeurs nulles).

14) (**1pt**) Reprendre la formulation précédente avec `numpy` et créer le jeu sous forme d'une classe `JeuDeLaVie`. En particulier, on utilisera comme moyen de stocker les iterations un tenseur tri-dimensionnel.

   - Attributs de la classe `JeuDeLaVie` :

     — `init_state` ($n_1 \times n_2$ ndarray),

     — `_time_T`

     — `_dimension: n_1, n_2=init_state.shape`

     — `_historic_state: (ndarray: n_1 x n_2 x (time_T+1))` (Remarque : on initialise le tensor à zéro puis on met à jour la première tranche : `self._historic_state[:, :, 0] = self.init_state`)

     — `average_life = np.zeros((n_1, n_2))`

   - Méthode de la classe `JeuDeLaVie` :

     — `play` : qui fait jouer le jeux de la vie jusqu'à `_time_T` et stocke dans la t^{e} tranche du tenseur, l'état du jeu au temps t pour tous les t de 0 à `_time_T`. On mettra aussi à jour l'attribut `average_life` qui permet de visualiser le temps de vie moyen de chaque cellule au cours du jeu.

     — `plot` : affiche la matrice `average_life` avec la palette `viridis`

15) Question bonus (**2pt**) : Comment modifier `jeu_np` et `iteration_jeu_np, calcul_nb_voisins_np` pour faire ce jeux non plus sur une grille carrée (avec une bordure "éteinte"), mais sur un tore, "à la Pac-Man". Reprendre la question 12) avec cette nouvelle règle.