
TP N° 1 : TP Noté

Pour ce travail vous devez envoyer par mail un lien **unique** du dépôt `github`, contenant l'intégralité du travail effectué. Le travail sera effectué par groupe de 2 ou 3 (les groupes de 1 ou de 4 ne sont pas acceptés).

Le dépôt `github` contiendra obligatoirement :

- un fichier `README.md` pour la page d'accueil décrivant succinctement votre travail.
- un fichier `notebook_tp_nom1_nom2.ipynb` contenant un notebook avec les réponses aux questions (à la fois code / questions de mathématiques).
- les fichiers aux formats `.py` pour certaines des fonctions python demandées.

Le projet doit être envoyé avant Dimanche 26 Mai 23h59. Ce qui sera fait sur le dépôt après cette date ne sera pas pris en compte pour la note.

De plus on veillera à équilibrer l'effort du groupe (nombre de commits, nombre de ligne de codes ajoutées/soustraites). En cas de contributions trop inégales entre les membres d'un même groupe, les notes seront ajustées par élèves.

La note totale est sur **20** points, répartis comme suit :

- qualité des réponses aux questions : **14** pts,
- qualité du dépôt `git` : **1** pts,
- qualité de rédaction et d'orthographe : **1** pts,
- qualité des graphiques (légendes, couleurs, titres, etc.) : **1** pt
- style PEP8 valide : **1** pts,
- qualité d'écriture 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

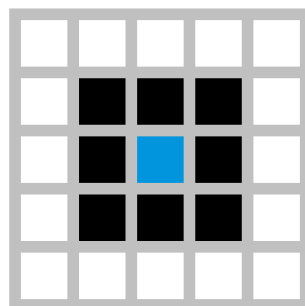


FIGURE 1 – Cellule (au centre, en bleu), et ses voisins (autour, en noir)

Le **jeu de la vie** 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 une grille orthogonale bidimensionnelle infinie de cellules carrées (dans la suite du projet on supposera 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), chacune d'entre elles se trouvant dans l'un des deux états possibles :

- vivant
- mort

Chaque cellule interagit avec ses huit voisins, qui 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 vit inchangée jusqu'à la génération suivante (**équilibre**), cf. Figure 2b
- c) Toute cellule vivante ayant 4 voisins vivants (**mort par étouffement**), cf. Figure 2c
- d) Toute cellule vivante ayant 0 ou 1 voisin vivant décède (**mort par isolement**), cf. Figure 2d.

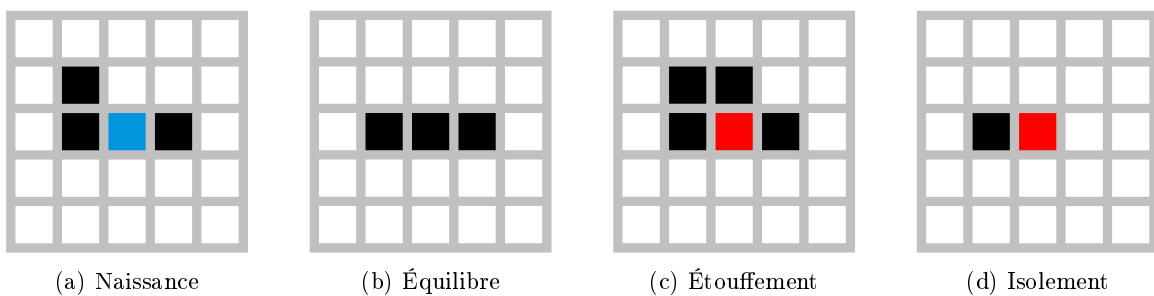


FIGURE 2 – Quatre configurations possibles pour la cellule centrale : naissance (bleu), équilibre, mort par étouffement et par isolement (rouge).

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 générations¹.

ATTENTION : La "carte" (on utilisera une liste de liste pour la représenter) est équipée d'une frontière de 0 : cela permet d'accélérer légèrement les calculs en évitant d'avoir des tests spécifiques pour les frontières lors du comptage du nombre de voisins. Ainsi la "couronne" du tour de la carte sera fixée à zéro.

Implémentation sans numpy

On va fournir dessous le code pure Python pour coder ce jeu. 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` :

```
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
```

- 4) (1pt) Appliquer la fonction précédente à la liste (de liste) Z suivante, et expliquer ce que représente la sortie obtenue $N=\text{calcul_nb_voisins}(Z)$. Cette fonction, et toutes les fonctions définies dans la suite seront placées dans un fichier `utils.py` que l'on appellera depuis le notebook.

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

```
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 il suit, en rajoutant une *docstring* pour cette fonction décrivant les entrées / sorties et ce que retourne la fonction :

```
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) (1pt) 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 numba

- 7) (1pt) Reprendre la section précédente avec des fonctions utilisant `numba` et la compilation "jit". On proposera en particulier un protocole expérimentale pour comparer les temps de calculs avec ou sans cet apport.
- 8) (2pts) Créer un *widget* dont le curseur permet de contrôler les itérations (par exemple de 0 à 30) 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 diminuer la taille des matrices en jeux pour rendre fluide le *widget* si besoin s'en fait sentir sur votre machine.

EXERCICE 2. Régression logistique

On va s'intéresser dans la suite à la base de données MNIST qui représente des chiffres numérisés, dont on connaît une "étiquette" associé parmi les chiffres (0, 1, ..., 9).


- 1) (0.5pt) Charger la base de données MNIST avec la commande `from sklearn.datasets import fetch_mldata` (ou bien avec `from sklearn.datasets import fetch_openml` si votre version de `sklearn` est plus récente). On pourra par exemple utiliser dans le premier cas :

```
mnist = fetch_mldata('MNIST original')
X = mnist.data.astype('float64')
y = mnist.target
```

- 2) (1pt) Transformer X et y pour ne garder que les cas des chiffres 3 et 7. Visualiser avec `imshow` un exemple de chaque classe d'image (un X donc), en utilisant un `reshape` adapté.
- 3) (1.5pt) Utiliser la fonction `LogisticRegression` pour apprendre un modèle de classification sur l'intégralité des données (on choisira un cas sans ordonnée à l'origine, *i.e.*, l'option `fit_intercept=False`). Le modèle prédit alors la classe d'une image en considérant une image comme un vecteur x et en choisissant l'une des deux classes selon le signe de $w^\top x$, où w est le vecteur appris par la méthode et stocké dans l'attribut `.coef_`.
- 4) (2pt) En utilisant le vecteur w de cette manière, proposer un widget qui investigate l'impact de la transformation de l'image par l'opération :

$$x_{mod} = x - \alpha \frac{w^\top x}{\|w\|^2} w . \quad (1)$$

On prendra pour x l'image associé au chiffre 7 de la question précédente, et le widget fera varier α de 0.1 à 100 de 0.1 en 0.1. Pour cela on définira une fonction `fig_digit` (à ajouter de nouveau dans le fichier `utils.py`), qui contient une *docstring* décrivant son fonctionnement.

- 5) (1pt) Créer un film que l'on importera en HTML dans le notebook, et qui représente l'évolution de l'image ainsi créer en fonction de α . On pourra s'inspirer de la page <http://jakevdp.github.io/blog/2012/08/18/matplotlib-animation-tutorial/> pour plus d'information concernant la création de films.
- 6) (1pt) Décrire mathématiquement ce que paramétrise le paramètre $\alpha > 0$.
- 7) (1pt) Proposer une analyse en composante principale ( : *Principal Component Analysis (PCA)*), pour visualiser la base de données dans un espace de dimension 2, en ajustant les couleurs selon la classe des données. On pourra s'inspirer de l'exemple : https://scikit-learn.org/stable/auto_examples/decomposition/plot_incremental_pca.html#sphx-glr-auto-examples-decomposition-plot-incremental-pca-py.