Corrigé

Jeu de la vie

1. Génération de la configuration initiale

Question 1. Le script ci-dessous calcule dans un tableau les 10 000 premières valeurs de la suite $(u_n)_{n\in\mathbb{N}}$:

```
import numpy as np

u = np.zeros(10000, dtype=int)
u[0] = 42
for i in range(1, 10000):
    u[i] = (16383 * u[i-1]) % 59047
```

On en déduit les valeurs demandées : $u_{996} = 58\,034$ et $u_{9996} = 8\,178$.

Question 2. Le script ci-dessous :

```
s = 0
for i in range(10000):
    if u[i] % 3 != 0:
        s += 1
print(s)
```

affiche le nombre de valeurs pour lesquelles $v_i = 0$, à savoir 6 693.

Question 3. On génère l'univers initial à l'aide de la fonction :

```
def genere_univers(k):
    univers = np.zeros((k,k), dtype=int)
    for i in range(k):
        for j in range(k):
            if u[i+j*k] % 3 == 0:
                univers[i, j] = 1
    return univers
```

Pour calculer le nombre de cellules vivantes, il peut être utile de définir la fonction :

```
def nb_vivantes(univers):
    k = univers.shape[0]
    s = 0
    for i in range(k):
        for j in range(k):
            if univers[i, j] == 1:
                s += 1
    return s
```

À l'aide de cette dernière on obtient que le nombre de cellules vivantes à la date $t = 0$ est de 128 pour $k = 20$ et de 815 pour $k = 50$.

2. Évolution de l'univers

Question 4. On commence par écrire une fonction qui calcule le nombre de voisins d'une case qui contiennent une cellule :

```
def nb_voisins(univers, i, j):
    k = univers.shape[0]
    s = 0
    for u in (-1, 0, 1):
        for v in (-1, 0, 1):
            if (u, v) == (0, 0):
                continue
            if univers[(i+u)%k, (j+v)%k] == 1:
                s += 1
    return s
```

Il reste à suivre la description des règles de mort et de naissance pour chaque case de l'univers :

```
def evolue(univers):
    k = univers.shape[0]
    nouvel_univers = univers.copy()
    for i in range(k):
        for j in range(k):
            if univers[i, j] == 1:                          # la case contient une cellule
                if nb_voisins(univers, i, j) not in (2, 3):
                    nouvel_univers[i, j] = 0                # mort
            else:                                           # la case est vide
                if nb_voisins(univers, i, j) == 3:
                    nouvel_univers[i, j] = 1                # naissance
    return(nouvel_univers)
```

Pour calculer le nombre de cellules vivantes à la date $t = 10$, on réalise le script suivant :

```
for k in (20, 50):
    univers = genere_univers(k)
    for _ in range(10):
        univers = evolue(univers)
    print(nb_vivantes(univers))
```

Pour $k = 20$ on obtient 102 cellules vivantes, et 583 pour $k = 50$.

Question 5. L'ensemble des univers possibles étant fini (de cardinal 2^{k^2}) il existe $t_0 < t_1 \leqslant 2^{k^2}$ tel que $u_{t_0} = u_{t_1}$, où u_t désigne l'état de l'univers à la date t. L'évolution de l'univers à la date $t+1$ ne dépendant que de l'état de l'univers à la date t, on démontre par récurrence que la suite u est périodique de période $t_1 - t_0$ à partir du rang t_0.

3. Calcul du temps d'attraction et de la période de l'attracteur

Question 6. On calcule les caractéristiques de l'attracteur à l'aide des fonctions :

```
def periode(univers):
    i, ui = 0, univers
    j, uj = 1, evolue(univers)
    while (ui != uj).any():
        if j == 2 * i + 1:
            i, ui = j, uj
        j, uj = j+1, evolue(uj)
    return j - i
```

```
def attraction(univers):
    p = periode(univers)
    i, ui = 0, univers
    up = univers.copy()
    for _ in range(p):
        up = evolue(up)
    while (ui != up).any():
        i, ui = i+1, evolue(ui)
        up = evolue(up)
    return p, i
```

Pour $k = 20$, la période est de 1 et le temps d'attraction vaut 373 (valeurs obtenues en 3 secondes sur mon ordinateur) ; pour $k = 50$ la période est égale à 2 et le temps d'attraction à 701 (valeurs obtenues en 42 secondes).

4. Îles et îlots de l'univers

Question 7. Nous allons écrire une fonction `taille_ilot` qui, à partir d'une case contenant une cellule, calcule la taille de l'îlot auquel la cellule appartient.

On utilise une liste `atraiter` dans laquelle toutes les cellules de l'îlot vont transiter, ainsi qu'un tableau `dejavu` qui marque les cellules déjà rentrées dans la liste `atraiter`. Le traitement d'une cellule consiste à ajouter dans la liste `atraiter` (en les marquant) les cellules voisines qui n'ont pas encore été traitées. Quand cette liste est vide, tous les éléments de l'îlot ont été vus.

```
def taille_ilot(univers, i0, j0):
    k = univers.shape[0]
    dejavu = np.zeros_like(univers, dtype=bool)
    dejavu[i0, j0] = True
    atraiter = [(i0, j0)]
    s = 1
    while atraiter != []:
        (i, j) = atraiter.pop()
        for u in (-1, 0, 1):
            for v in (-1, 0, 1):
                if (u, v) == (0, 0):
                    continue
                if univers[(i+u)%k, (j+v)%k] == 1 and not dejavu[(i+u)%k, (j+v)%k]:
                    dejavu[(i+u)%k, (j+v)%k] = True
                    atraiter.append(((i+u)%k, (j+v)%k))
                    s += 1
    return s
```

Pour $k = 20$ la cellule présente en $(0, 0)$ appartient à un îlot de taille 9 et pour $k = 50$ à un ilot de taille 4.

Question 8. De manière analogue, pour chaque cellule n'ayant pas encore été vue, on parcourt en les marquant les cellules qui appartiennent à son îlot, ce qui permet de dénombrer le nombre d'îlots :

```
def nb_ilots(univers):
    k = univers.shape[0]
    dejavu = np.zeros_like(univers, dtype=bool)
    s = 0
    for i0 in range(k):
        for j0 in range(k):
            if univers[i0, j0] == 0 or dejavu[i0, j0]:  # cellule vide ou déjà vue
                continue
            s += 1                      # on débute l'exploration d'un nouvel îlot
            dejavu[i0, j0] = True
            atraiter = [(i0, j0)]
            while atraiter != []:
                (i, j) = atraiter.pop()
                for u in (-1, 0, 1):
                    for v in (-1, 0, 1):
                        if (u, v) == (0, 0):
                            continue
                        if univers[(i+u)%k, (j+v)%k] == 1 and not dejavu[(i+u)%k, (j+v)%k]:
                            dejavu[(i+u)%k, (j+v)%k] = True
                            atraiter.append(((i+u)%k, (j+v)%k))
    return s
```

Pour $k = 20$, on obtient 18 îlots pour $t = 0$, 11 îlots pour $t = 1$ et 1 îlot pour $t = t_0$.
Pour $k = 50$, on obtient 91 îlots pour $t = 0$, 63 îlots pour $t = 1$ et 21 îlots pour $t = t_0$.

Question 9. Calculer le nombre d'îles suit la même logique : il suffit de remplacer la mesure du voisinage `(-1, 0, 1)` par la mesure plus large `(-2, -1, 0, 1, 2)`. On obtient 17 îles pour $k = 50$ et $t = t_0$. À l'aide de l'animation et une fois atteint le temps d'attraction, il est facile d'observer qu'il existe une île formée de deux îlots et une île formée de quatre îlots, qui correspondent aux deux motifs de période 2.

Vers l'infini et au-delà

Pour représenter un univers infini, nous allons utiliser le type `set` qui permet de manipuler des ensembles : l'univers à l'instant t est décrit par l'ensemble des coordonnées des cellules vivantes.
La configuration initiale est engendrée par la fonction :

```
def genere_univers_infini(k):
    univers = set()
    for i in range(k):
        for j in range(k):
            if u[i+j*k] % 3 == 0:
                univers.add((i, j))
    return univers
```

Nous aurons besoin d'une fonction qui calcule le nombre de cellules voisines d'une case :

```
def nb_voisins2(univers, i, j):
    s = 0
    for u in (-1, 0, 1):
        for v in (-1, 0, 1):
            if (u, v) == (0, 0):
                continue
            if (i+u, j+v) in univers:
                s += 1
    return s
```

et d'une fonction qui fait évoluer l'univers :

```
def evolue2(univers):
    nouvel_univers = set()
    for (i, j) in univers:
        if nb_voisins2(univers, i, j) in (2, 3):  # la cellule survit
            nouvel_univers.add((i, j))
        for u in (-1, 0, 1):                      # recherche des lieux de naissances possibles
            for v in (-1, 0, 1):
                if (i+u, j+v) in univers:
                    continue
                if nb_voisins2(univers, i+u, j+v) == 3:  # naissance
                    nouvel_univers.add((i+u, j+v))
    return(nouvel_univers)
```

On utilise le fait que les lieux de naissances possibles sont à chercher parmi les cases voisines des cellules actuelles.

Il reste à écrire deux fonctions pour calculer le nombre de cellules vivant dans l'univers, ainsi que les coordonnées des cellules les plus éloignées :

```
def nb_vivantes2(univers):
    return len(univers)

def plus_lointaine(univers):
    d = 0
    s = []
    for (i, j) in univers:
        if abs(i)+abs(j) > d:
            d = abs(i)+abs(j)
            s = [(i, j)]
        elif abs(i)+abs(j) == d:
            s.append((i, j))
    return d, s
```

Pour $k = 20$ et $t = 1\,000$ on obtient 66 cellules, les plus éloignées de l'origine étant aux coordonnées $(225, 248)$ et $(226, 247)$.
Pour $k = 20$ et $t = 10\,000$ on obtient toujours 66 cellules, les plus éloignées étant aux coordonnées $(2476, 2497)$ et $(2445, 2498)$.
En réalité, la situation s'est « stabilisée » à la date $t = 190$ (voir figure 1). À l'exception d'un seul, les différents motifs que l'on voit sont périodiques de période 1 (par exemple les pavés de quatre cellules) ou de période 2 (les alignements de trois cellules). Un seul fait exception, le motif en bas à droite, qui est un *glisseur*, nommé ainsi car il a la propriété de ce reproduire au bout de quelques itérations décalé de quelques cases. La figure 2 illustre le déplacement de ce motif au bout de quatre itérations.

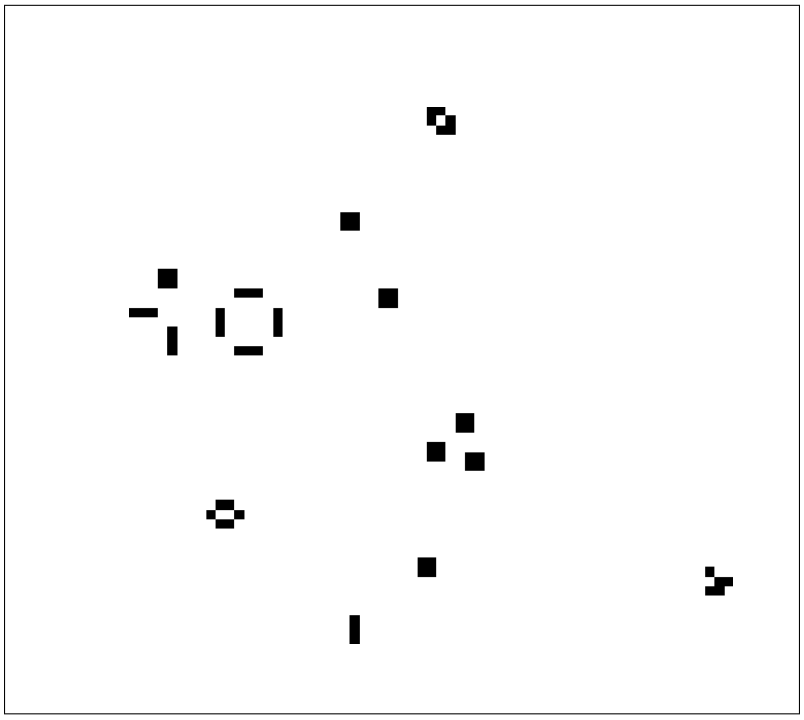

FIGURE 1 – La situation à la date $t = 190$.

Remarque. Pour afficher l'état de l'univers j'ai utilisé la fonction suivante :

```
def affiche_univers(univers):
    xmin = ymin = xmax = ymax = 0
    for (i, j) in univers:
        xmin, xmax = min(xmin, i), max(xmax, i)
        ymin, ymax = min(ymin, j), max(ymax, j)
    m = np.zeros((xmax-xmin+1, ymax-ymin+1), dtype=int)
    for (i, j) in univers:
        m[i-xmin, j-ymin] = 1
    plt.matshow(m)
    plt.show()
```

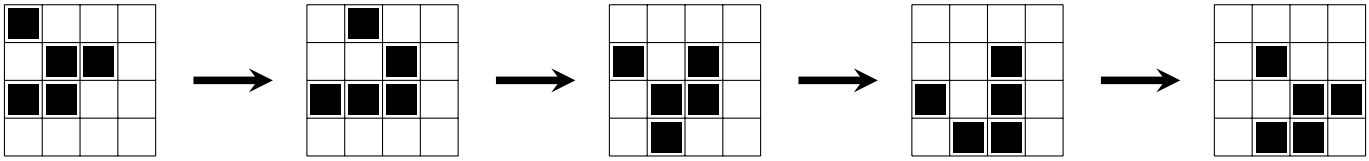

FIGURE 2 – Le déplacement du glisseur.