TP 5 - Corrigé partie 2
Jeu de la vie

Les solutions données dans ce corrigé ne sont bien sûr que des propositions, et sont sans nul doute perfectibles. Parfois on proposera même plusieurs solutions en discutant de leur pertinence.

1 Un peu de statistiques

Q7 Il suffit de prendre grade à réinitialiser à zéro les différents compteurs au début chaque itération. Il peut être judicieux de distinguer d'abord les cas « cellule morte » et « cellule vivante », puis les conditions sur le nombre de voisins, pour être sûr de n'oublier aucun cas.

Listing 1 – iterer_jeu

```
def iterer_jeu(grille, nb_iter, afficher = False):
    lPop = []
    lNaissances = []
    lMorts = []

    for k in range(nb_iter):
        pop = 0
        naissances = 0
        morts = 0

        for i in range(1,grille.shape[0]-1):
            for j in range(1,grille.shape[1]-1):
                c = compter_voisins(grille, i , j)
                if (grille[i,j] == 0 and c == 3):
                    grille[i,j] = 1
                    pop = pop+1
                    naissances = naissances+1
                elif (grille[i,j] == 1):
                    if (c != 2 and c != 3):
                        morts = morts+1
                        grille[i,j] = 0
                    else:
                        pop = pop+1

        lPop.append(pop)
        lNaissances.append(naissances)
        lMorts.append(morts)
        if (afficher):
```

```
29            afficher_jeu(grille)
30            plt.title('${:3d}$'.format(k))
31            plt.pause(interval)
32    return (np.array(lPop), np.array(lMorts), np.array(
        lNaissances))
```

Q8 Aucune difficulté en utilisant la fonction `plot`. Il peut être une bonne idée de tracer dans une autre figure, à l'aide de la fonction `figure`. *Par exemple* après un appel à `plt.figure(800)`, les fonctions `plot`, `legend`, `title`, etc., agiront sur la figure 800.

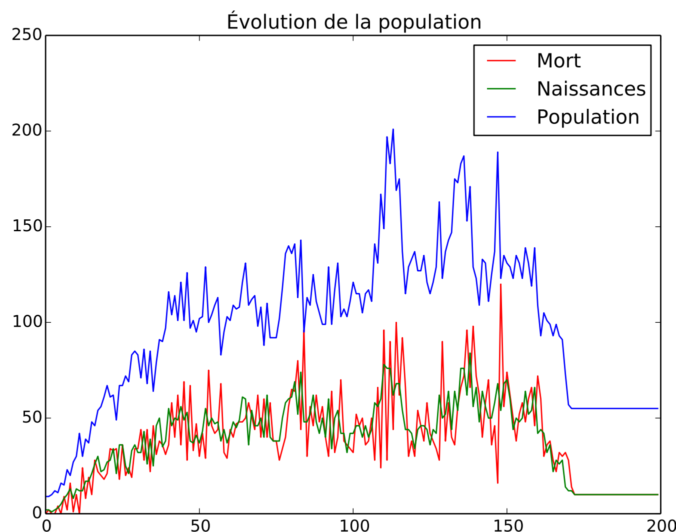
Penser à ajouter une légende et un titre appropriés.

Remarque : sans appel préalable à la fonction `figure`*, les tracés se font par défaut dans la figure* 0

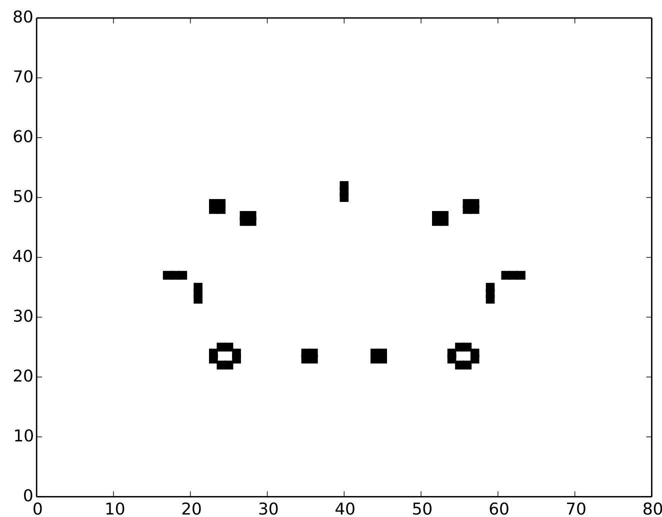
Listing 2 – Q8

```
1  G = init_jeu_u(N)
2
3  (vPop, vMorts, vNaissances) = iterer_jeu(G, 200, afficher =
      False)
4
5  plt.figure(800)
6  plt.plot(vMorts, 'r-')
7  plt.plot(vNaissances, 'g-')
8  plt.plot(vPop, 'b-')
9  plt.legend(['Mort', 'Naissances', 'Population'], loc='best')
10 plt.title('Evolution␣de␣la␣population')
```

On obtient le graphique ci-dessous.



On constate qu'à la 173e itération, la population se stabilise à 55 cellules vivantes. En effet, on obtient la configuration périodique (de période 2) suivante :



Q9 On peut écrire une fonction « à la main » en utilisant la fonction `rand`, ou utiliser la fonction `choice` qui permet de simuler directement une variable aléatoire de Bernoulli. On propose donc deux corrigés.

Remarque : on a choisi de n'initialiser que l'intérieur de la grille, en évitant les bords, pour être cohérent avec le fait qu'on ignore les bords pour mettre à jour la grille. Mais c'est un détail.

Listing 3 – init_jeu_aleat à la main

```
1  def init_jeu_aleat(N, p):
2      grille = np.zeros((N, N))
3
4      for i in range(1,N-1):
5          for j in range(1,N-1):
6              x = rd.rand()
7              if (x <= p):
8                  grille[i,j] = 1
9              else:
10                 grille[i,j] = 0
11
12     return grille
```

On peut même astucieusement écrire cela en une ligne en utilisant les opérations sur les tableaux.

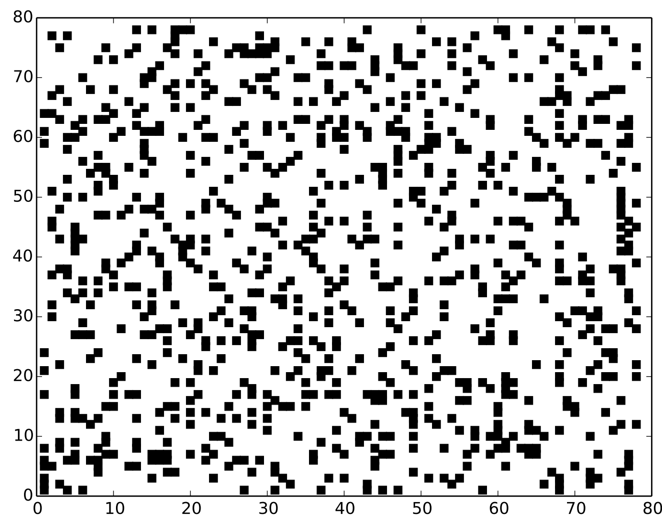
Listing 4 – init_jeu_aleat à la main en une ligne

```
1 def init_jeu_aleat(N, p):
2     grille = np.zeros((N, N))
3 
4     # Plus efficacement
5     # On multiplie par 1 pour obtenir un vecteur de nombres et
         pas de booleens
6     # (meme si en Python False est toujours assimile a 0 et True
          a 1)
7     grille[1:-1,1:-1] = 1*(rd.rand(N-2, N-2) <=p)
8 
9     return grille
```

Listing 5 – init_jeu_aleat avec la fonction choice

```
1 def init_jeu_aleat(N, p):
2     grille = np.zeros((N, N))
3     grille[1:-1,1:-1] = rd.choice(2, (N-2, N-2), p=np.array([1-p
         , p]))
4     return grille
```

On peut obtenir par exemple la configuration initiale suivante pour $p = 0.15$:



Q10 Aucune difficulté ici. On peut utiliser une boucle *for* et utiliser la fonction `figure` pour tracer dans une figure différente pour chaque valeur de p. Le temps de calcul peut être assez long !

Remarque : la fonction `enumerate` *permet de parcourir facilement une liste en donnant à la fois l'élement courant de la liste et l'indice (numéro) de cet élément à chaque itération.*

Listing 6 – init_jeu_aleat à la main en une ligne

```
vp = [0.15, 0.5, 0.6, 0.8]

for (i,p) in enumerate(vp):
    G = init_jeu_aleat(N, p)
    (vPop, vMorts, vNaissances) = iterer_jeu(G, 800)

    plt.figure(1000+i)
    plt.plot(vMorts, 'r-')
    plt.plot(vNaissances, 'g-')
    plt.plot(vPop, 'b-')
    plt.legend(['Mort', 'Naissances', 'Population'], loc='best')
    plt.title('Evolution variable de Bernoulli\n$p = {:.3f}$'.
        format(p))
```

On peut observer qu'au bout d'un certain nombre d'itérations la population se stabilise. Si on exécute plusieurs fois le code précédent, on voit que les courbes obtenues sont en fait très variables, selon la configuration initiale (aléatoire), et qu'il est difficile d'en tirer des conclusions. C'est pourquoi on va répéter un certain nombre de fois les calculs précédents pour obtenir une évolution *moyenne* de la population, du nombre de morts et de naissances.

Q11 Là encore pas de difficulté. Le symbole *'+'* permet d'additionner deux tableaux (`array`) de même taille. Attention car appliqué à des listes ce symbole *concatène* les listes au lieu de les additionner ! La fonction `iterer_jeu` doit donc bien retourner des tableaux, et non des listes.

```
plt.figure(1100)
vPopMoy = np.zeros(250)
p = 0.15

for j in range(10):
    G = init_jeu_aleat(N, p)
    (vPop, vMorts, vNaissances) = iterer_jeu(G, 250)
    vPopMoy = vPopMoy + vPop
    vPopMoy = vPopMoy / nb_samples

plt.plot(vPopMoy)
plt.legend('$p = {:.3f}$'.format(p), loc='best')
plt.title('Evolution population moyenne')
```

Q12 Comme les calculs sont très longs, on cherche à optimiser notre code. Plutôt que de compter le nombre de voisines de chaque cellule vivante, on va partir d'un tableau `T` de taille $N \times N$ initialisé à zéro, et pour chaque cellule vivante (i, j) de la grille, ajouter 1 aux éléments de `T` voisins de (i, j). Une fois que toute la grille a été parcourue, `T[i, j]` contient le nombre de voisines vivantes de la cellule (i, j).

On voit que globalement on effectue dans le pire cas le même nombre d'opérations que précédemment (le pire cas est obtenu lorsque toutes les cellules sont vivantes), mais qu'en pratique, si la grille contient peu de cellules vivantes, on en effectue moins.

Remarque : on peut vérifier ce gain de performance en mesurant le temps d'exécution de `iterer_jeu` *pour un grand nombre d'itérations, à l'aide du module* `time` *par exemple.*

Listing 7 – calculer_matrice_voisins

```
1  def calculer_matrice_voisins(grille):
2      T = np.zeros_like(grille)
3      for i in range(1, grille.shape[0]-1):
4          for j in range(1, grille.shape[1]-1):
5              if (grille[i, j] != 0):
6                  # On aurait aussi pu utiliser une boucle ici
                       plutot que d'ecrire
7                  # une suite d'instructions assez similaires
8                  T[i-1,j-1] += 1
9                  T[i-1,j] += 1
10                 T[i-1,j+1] += 1
11                 T[i,j-1] += 1
12                 T[i,j+1] += 1
13                 T[i+1,j-1] += 1
14                 T[i+1,j] += 1
15                 T[i+1,j+1] += 1
16     return T
```

On modifie la fonction `iterer_jeu` en conséquence. Ci-dessous les « ... » indiquent que rien n'a changé par rapport à la version précédente.

Listing 8 – iterer_jeu avec calculer_matrice_voisins

```
1  def iterer_jeu(grille, nb_iter, afficher = False):
2      ...
3
4      for k in range(nb_iter):
5          voisins = calculer_matrice_voisins(grille)
6          ...
7
8          for i in range(1,grille.shape[0]-1):
9              for j in range(1,grille.shape[1]-1):
10                 c = voisins[i, j]
11                 ...
12
```

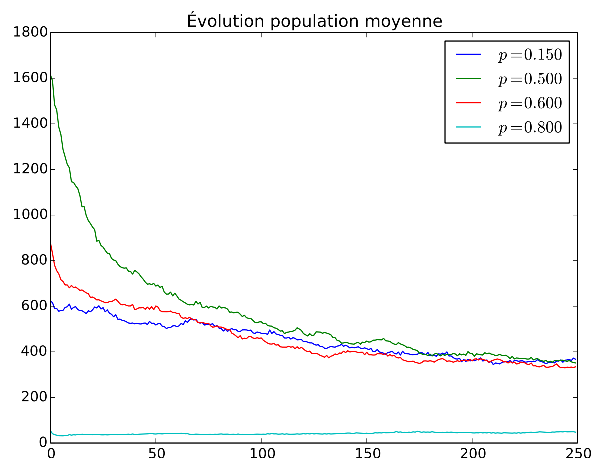
```
13            ...
14        return (np.array(lPop), np.array(lMorts), np.array(
              lNaissances))
```

Q13 Pas de difficulté ici. S'armer de patience pour le temps de calcul !

Listing 9 – Q13

```
1  vp = [0.15, 0.5, 0.6, 0.8]
2  legend = []
3
4  plt.figure(1300)
5  for (i,p) in enumerate(vp):
6      nb_iter = 250
7      nb_samples = 10
8      vPopMoy = np.zeros(nb_iter)
9
10     for j in range(nb_samples):
11         G = init_jeu_aleat(N, p)
12
13         (vPop, vMorts, vNaissances) = iterer_jeu(G, nb_iter)
14         vPopMoy = vPopMoy + vPop
15     vPopMoy = vPopMoy / nb_samples
16
17     plt.plot(vPopMoy)
18     legend.append('$p␣=␣{:.3f}$'.format(p))
19
20 plt.legend(legend, loc='best')
21
22 plt.title('Evolution␣population␣moyenne')
```

Il faudrait effectuer une moyenne sur un plus grand nombre de configurations initiales, mais les calculs sont déjà très longs. Ci-dessous un exemple de courbes qu'on peut obtenir :



En exécutant ce code plusieurs fois on peut remarquer qu'il y a moins de variations sur les courbes obtenues. On peut commencer à commenter quelque peu ce qu'on observe.

On remarque que pour de grandes valeurs de p, la population s'effondre presque immédiatement, ce qui n'est pas surprenant car une cellule meurt si elle a plus de 3 voisines vivantes. De même en prenant des valeurs très faibles de p, ce qui est également attendu car une cellule ayant moins de 2 voisines meurt.

On peut remarquer qu'en moyenne dans tous les cas la population a tendance à décroître avant de se stabiliser, ce qui semble indiquer que les configurations comme le « U inversé » pris en exemple précédemment sont des exceptions (on a vu que dans ce cas la population se stabilisait à 55 alors qu'elle est initialement de 7).

2 Pour s'amuser encore un peu

2.1 Ajoutons de la couleur

Q14 Rien de bien difficile. On pensera néanmoins à appliquer une fois les règles de mise à jour de la grille (en appelant `iterer_jeu`, qui sera modifiée pour prendre en compte le troisième indice de la grille), de sorte que `G[:,:,2]` contienne bien la génération suivante.

Listing 10 – init_jeu_u version couleur

```
1  def init_jeu_u(N) :
2      grille = np.zeros((N, N, 3), dtype=float)
3      c = N//2
4
5      grille[c-1,c-1:c+2,0] = 1
6      grille[c,c+1,0] = 1
7      grille[c+1,c-1:c+2,0] = 1
8      grille[:,:,1] = grille[:,:,0]
9      grille[:,:,2] = grille[:,:,1]
10     iterer_jeu(grille, 1)
11
12     return grille
```

Listing 11 – init_jeu_aleat version couleur

```
1  def init_jeu_aleat(N, p):
2      grille = np.zeros((N, N, 3))
3      grille[1:-1,1:-1,0] = 1*(rd.rand(N-2, N-2) <=p)
4
5      grille[:,:,1] = grille[:,:,0]
6      grille[:,:,2] = grille[:,:,1]
7      iterer_jeu(grille, 1)
8      return grille
```

Q15 Il suffit de créer trois listes de coordonnées, correspondant respectivement aux cellules mourrantes, naissantes, et ne vivant que pour une génération.

Listing 12 – afficher_jeu version couleur

```
1  def afficher_jeu(grille):
2      Xnaissant = []
3      Ynaissant = []
4      Xmourrant = []
5      Ymourrant = []
6      X1gen = []
7      Y1gen = []
8      Xstable = []
9      Ystable = []
10     for i in range(1,grille.shape[0]-1):
11         for j in range(1,grille.shape[1]-1):
12             if (grille[i,j,1] != 0):
13                 if (grille[i,j,0] != 0 and grille[i,j,2] != 0):
14                     Xstable.append(i)
15                     Ystable.append(j)
16                 elif (grille[i,j,0] == 0 and grille[i,j,2] != 0)
                       :
17                     Xnaissant.append(i)
18                     Ynaissant.append(j)
19                 elif (grille[i,j,0] != 0 and grille[i,j,2] == 0)
                       :
20                     Xmourrant.append(i)
21                     Ymourrant.append(j)
22                 else:
23                     X1gen.append(i)
24                     Y1gen.append(j)
25     plt.clf()
26     plt.axis([0, grille.shape[1], 0, grille.shape[0]])
27     plt.plot(Xstable, Ystable, 'ks')
28     plt.plot(Xnaissant, Ynaissant, 'gs')
29     plt.plot(Xmourrant, Ymourrant, 'rs')
30     plt.plot(X1gen, Y1gen, 'ys')
```

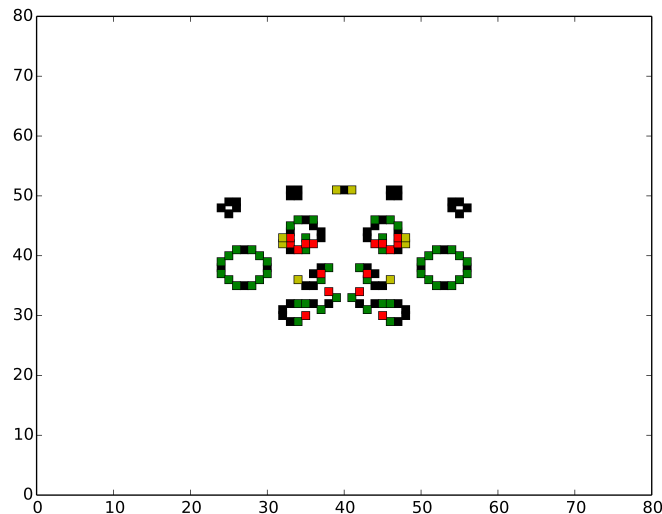
Q16 Il suffit de copier la grille N dans la $N-1$, la $N+1$ dans la N, et mettre à jour la grille $N+1$, i.e. `G[:,:,2]`, comme précédemment.

Listing 13 – iterer_jeu version couleur

```
1  def iterer_jeu(grille, nb_iter, afficher = False):
2      lPop = []
3      lNaissances = []
4      lMorts = []
5
```

```
6     for k in range(nb_iter):
7         grille[:,:,0] = grille[:,:,1]
8         grille[:,:,1] = grille[:,:,2]
9         voisins = calculer_matrice_voisins(grille[:,:,1])
10        pop = 0
11        naissances = 0
12        morts = 0
13
14        for i in range(1,grille.shape[0]-1):
15            for j in range(1,grille.shape[1]-1):
16                c = voisins[i,j]
17                if (grille[i,j,2] == 0 and c == 3):
18                    grille[i,j,2] = 1
19                    pop = pop+1
20                    naissances = naissances+1
21                elif (grille[i,j,2] == 1):
22                    if (c != 2 and c != 3):
23                        morts = morts+1
24                        grille[i,j,2] = 0
25                    else:
26                        pop = pop+1
27
28        lPop.append(pop)
29        lNaissances.append(naissances)
30        lMorts.append(morts)
31        if (afficher):
32            afficher_jeu(grille)
33            plt.title('${:3d}$'.format(k))
34            plt.pause(10**-15)
35    return (np.array(lPop), np.array(lMorts), np.array(
         lNaissances))
```

On obtient par exemple, à partir de la configuration « U inversé », après 110 itérations, la figure suivante :



Q17 Il suffit de calculer les abscisses et ordonnées minimales et maximales du motif pour qu'il soit bien centré.

Listing 14 – init_jeu_motif

```
1  def init_jeu_motif(N, motif):
2      grille = np.zeros((N, N, 3))
3      c = N//2
4  
5      grille[(N-motif.shape[0])//2:(N+motif.shape[0])//2,
6      (N-motif.shape[1])//2:(N+motif.shape[1])//2,0] = motif
7      grille[:,:,1] = grille[:,:,0]
8      grille[:,:,2] = grille[:,:,1]
9      iterer_jeu(grille, 1)
10 
11     return grille
```

Pour ensuite prendre par exemple le pentadécathlon comme configuration initiale, on peut procéder comme suit :

Listing 15 – Initialisation pentadecathlon

```
1  pentadecathlon = np.array
       ([[1,1,1],[1,0,1],[1,1,1],[1,1,1],[1,1,1],[1,1,1],[1,0,1],
2  [1,1,1]]).T
3  G = init_jeu_motif(N, pentadecathlon)
```

On peut remarquer que le pentadécathlon est, comme son nom l'indique, un motif périodique de période 15.

Le pulsar est quant à lui de période 3, comme le montrent les figures ci-dessous.