Devoir d'informatique 5 – Corrigé

D.Malka – MPSI 2018-2019 – Lycée Jeanne d'Albret

07.06.2019

Durée de l'épreuve : 2h00

L'usage de la calculatrice n'est pas autorisé.

L'énoncé de ce devoir comporte 7 pages.

1 Modélisation de la propagation d'une épidémie par automate cellulaire

1. la fonction `grille(n)` renvoie une grille de taille $n \times n$ dont toutes les valeurs sont nulles.
2. Fonction `init(n)` qui construit une grille G de taille $n \times n$ ne contenant que des cases saines, choisit aléatoirement une des cases et la transforme en case infectée, et enfin renvoie G.

```
def init(n):
    G=grille(n)
    i,j=rd.randrange(0,n),rd.randrange(0,n)
    G[i][j]=1
    return G
```

3. Fonction `compte(G)` qui a pour argument une grille G et renvoie la liste `[n0, n1, n2, n3]` formée des nombres de cases dans chacun des quatre états. En remarquant que les indices de la liste `[n0, n1, n2, n3]` sont les valeurs des états des éléments de la grille, on évite d'écrire du code lourd à base de `if...elif...else`. A retenir car c'est une structure de données récurrente.

```
def compte(G):
    n=len(G)
    c=[0,0,0,0]
    for i in range(n):
        for j in range(n):
            e=G[i][j]#la valeur de G[i,j] donne l'indice de l'elt de c a incrementer
            c[e]=c[e]+1
    return c
```

4. La fonction `est_exposee` renvoie un booléen.
5. Fonction `est_exposee`.

```
def est_exposee(G,i,j):
    n=len(G)
    #coins
    if i==0 and j==0:
        return (G[0][1]-1)*(G[1][1]-1)*(G[1][0]-1)==0
    elif i==0 and j==n-1:
        return (G[0][n-2]-1)*(G[1][n-2]-1)*(G[1][n-1]-1)==0
    elif i==n-1 and j==0:
        return (G[n-1][1]-1)*(G[n-2][1]-1)*(G[n-2][0]-1)==0
    elif i==n-1 and j==n-1:
        return (G[n-2][n-1]-1)*(G[n-2][n-2]-1)*(G[n-1][n-2]-1)==0
    #bords
    elif i==0:
        return (G[0][j-1]-1)*(G[0][j+1]-1)*(G[1][j-1]-1)*(G[1][j]-1)*(G[1][j+1]-1)==0
    elif i==n-1:
```

```
        return (G[n-1][j-1]-1)*(G[n-1][j+1]-1)*(G[n-2][j-1]-1)*(G[n-2][j]-1)*(G[n-2][j+1]-1)==0
    elif j==0:
        return (G[i-1][0]-1)*(G[i+1][0]-1)*(G[i-1][1]-1)*(G[i][1]-1)*(G[i+1][1]-1)==0
    elif j==n-1:
        return (G[i-1][n-1]-1)*(G[i+1][n-1]-1)*(G[i-1][n-2]-1)*(G[i][n-2]-1)*(G[i+1][n-2]-1)==0
    #case ni aux bords, ni aux coins
    else:
        return (G[i-1][j-1]-1)*(G[i-1][j]-1)*(G[i-1][j+1]-1)*(G[i][j-1]-1)*(G[i][j+1]-1)*(G[i+1][j-1]-1)*(G[i+1][j]-1)*(G[i+1][j+1]-1)==0
```

6. Fonction `suivant(G, p1, p2)` qui fait évoluer toutes les cases de la grille G à l'aide des règles de transition et renvoie une nouvelle grille correspondant à l'instant suivant.

```
def suivant(G,p1,p2):
    n=len(G)
    Gc=copie_grille(G)#il faut faire une copie de G a t pour ne pas que les modfications a t+1 soit auto-impactantes
    for i in range(n):
        for j in range(n):
            if G[i][j]==0 and est_exposee(G,i,j):#sain et expose
                Gc[i][j]=bernoulli(p2)
            elif G[i][j]==1:#est infecte
                Gc[i][j]=2+bernoulli(p1)# vaut 3 donc mort pour bernoulli(p2)=1; vaut 2 sinon (retabli)
            else:#deja mort ou retabli ou sain mais pas expose
                Gc[i][j]=G[i][j]
    return Gc
```

7. Fonction `simulation(n, p1, p2)` qui réalise une simulation complète avec une grille de taille $n \times n$ pour les probabilités `p1` et `p2`, et renvoie la liste `[x0, x1, x2, x3]` formée des proportions de cases dans chacun des quatre états à la fin de la simulation (une simulation s'arrête lorsque la grille n'évolue plus).

```
def sont_identiques(G1,G2):
    n=len(G1)
    for i in range(n):
        for j in range(n):
            if G2[i][j]!=G1[i][j]:
                return False
    return True

def simulation(n,p1,p2):
    G=init(n);
    stop=False
    while not stop:
        Gc=suivant(G,p1,p2);
        stop=sont_identiques(G,Gc)
        G=Gc
    return compte(G)
```

8. A la fin de la simulation, chaque case infectée est devenue soit morte, soit rétablie. Donc $x_1 = 0$. Par conservation de la population : $x_0 + x_1 + x_2 + x_3 = n^2$. Proportion des cases qui ont été atteintes par la maladie pendant une simulation : $x_{atteinte} = (x_2 + x_3)/(x_0 + x_1 + x_2 + x_3)$.

   On fixe `p1` à 0,5 et on calcule la moyenne des résultats de plusieurs simulations pour différentes valeurs de `p2`. On obtient la courbe de la figure fig.1.

9. Fonction `seuil(Lp2, Lxa)` qui détermine par dichotomie un encadrement `[p2cmin, p2cmax]` du seuil critique de pandémie avec la plus grande précision possible.

```
def seuil(Lp2,Lxa):
    n=len(Lp2)
    g=0;d=n-1
    p2cmin=Lp2[g];p2cmax=Lp2[d]
    while d-g>1:
```

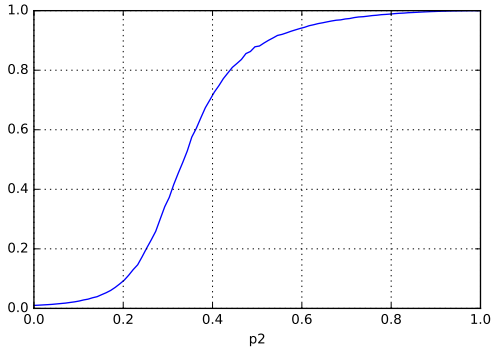

FIGURE 1 – Représentation de la proportion de la population qui a été atteinte par la maladie pendant la simulation en fonction de la probabilité p_2.

```
6           m=(d+g)//2
7           if Lxa[m]<=0.5:
8               g=m
9               p2cmin=Lp2[g]
10          else:#Lxa[m]>0.5
11              d=m
12              p2cmax=Lp2[d]
13      return [p2cmin,p2cmax]
```

Pour étudier l'effet d'une campagne de vaccination, on immunise au hasard à l'instant initial une fraction q de la population. On a écrit la fonction `init_vac(n, q)`.

```
1   def init_vac(n,q):
2       G=init(n)
3       nvac=int(q*n**2)
4       k=0
5       while k<nvac:
6           i=rd.randrange(n)
7           j=rd.randrange(n)
8           if G[i][j]==0:
9               G[i][j]==2
10              k+=1
11      return G
```

10. La case (i, j) étant choisi aléatoirement, on ne peut pas exclure qu'une même case soit tirée plusieurs fois. Si on supprime le test de la ligne 8, il est probable que l'on vaccine plusieurs fois le même individu ce qui ne fait pas sens dans le cadre du modèle. Ou bien qu'on vaccine l'individu initialement infecté.
11. L'appel `init_vac(5, 0.2)` renvoie une grille de taille 5×5 dans laquelle `int(0.2*5**2)=5` cases sont mises à 2 et une à 1. Par exemple :

0	0	1	2	0
0	0	0	2	0
0	2	0	2	0
0	0	0	0	0
0	0	0	2	0

2 Contrôle et prédiction de la position d'un véhicule automatisé

1. Fonction `position_odometrie(delta_d,delta_g, p0)` renvoyant le tableau `p1`.

```
1   def position_odometrie(delta_d,delta_g,p0):
2       p1=np.zeros(3)
3       x,y,theta=p0[0],p0[1],p0[2]
4       m=(delta_d+delta_g)/2
5       d=(delta_d-delta_g)/2
```

```
    delta_x=r*m*np.cos(theta+r*d/2(2*e))
    delta_y=r*m*np.sin(theta+r*d/2(2*e))
    delta_theta=2*r*m

    p1[0]=x+delta_x
    p1[1]=y+delta_y
    p1[2]=theta+delta_theta

    return p1
```

2. La fonction `calcul_sigma_2(mat_sigmap,mat_jacob)` calcule $\Sigma_2 = J_p \Sigma_p J_p^T$. Il faut donc définir une fonction `transpose(M)` renvoyant la transposée de la matrice passée en entrée et une fonction `dot(A,B)` renvoyant le produit `C` des matrices `A` et `B` passées en arguments. La complexité de `transpose(M)` est $O(n^2)$, celle de `dot(A,B)` est $O(n^3)$ donc celle de `calcul_sigma_2(mat_sigmap,mat_jacob)` qui opère une transposition et deux produits matriciels est $O(n^3)$.s

```
def transpose(M):
    n,p=M.shape
    tM=np.zeros((p,n))
    for i in range(p):
        for j in range(n):
            tM[i,j]=M[j,i]
    return tM

def dot(A,B):
    n,p=A.shape
    p,m=B.shape
    C=np.zeros((n,m))
    for i in range(n):
        for j in range(m):
            for k in range(p):
                C[i,j]=C[i,j]+A[i,k]*B[k,j]
    return C

def calcul_sigma2(mat_sigmap,mat_jacob):
    t_mat_jacob=transpose(mat_jacob)
    temp_mat=dot(mat_jacob,mat_sigmap)
    mat_sigma2=dot(temp_mat,t_mat_jacob)
    return mat_sigma2
```

3. Fonction `prediction(p1,cible_map)` qui prend en arguments le tableau `p1` de dimension 3, associé à la position estimée ($\hat{p}(k+1|k)$)), le tableau `cible_map` et renvoie le tableau `zpred` de dimensions 12×2, contenant les coordonnées locales (r_i, φ_i) des 12 cibles (12 vecteurs $\hat{z}_i(k+1)$).

```
def global_local(xi,yi,p1):
    pass #suppose deja implemente
    return ri,phii

def  prediction(p1,cible_mat):
    """
    cible_mat=[[x0,y0],[x1,y1]....[x11,y11]]
    """
    zpred=[]
    for coord in cible_mat:
        r,phi=global_local(coord[0],coord[1],p1)#coord[0]=xi,coord[1]=yi
        zpred.append([r,phi])
    return zpred
```

4. Fonction `comparaison(zj,zpred)` conduisant à identifier dans le tableau `zzpred` les coordonnées `(r_pp,phi_pp)` de la cible C_{i0} plus proche voisin d'une cible C_j détectée par le laser soit minimisant la distance :

$$d_{ij} = \sqrt{r_i^2 + r_j^2 - 2 r_i r_j \cos(\varphi_j - \varphi_i)}$$

```
1  def comparaison(zj,zpred):
2      index=1
3      rj,phij=zj[0],zj[1]
4      ri,phii=zpred[0][0],zpred[0][1]
5      dmin=np.sqrt(ri**2+rj**2-2*ri*rj*np.cos(phij-phii))
6      for zi in zpred:
7          ri,phii=zi[0],zi[1]
8          d=np.sqrt(ri**2+rj**2-2*ri*rj*np.cos(phij-phii))
9          if d<=dmin:
10             d=dmin
11             index+=1
12     return zj[index]
```