Devoir d'informatique 5 – Corrigé

D.Malka – MPSI 2017-2018 – Lycée Saint-Exupéry

25.05.2018

1 Modèle à compartiments

1. Vecteur $X = \begin{pmatrix} S(t) \\ I(t) \\ R(t) \\ D(t) \end{pmatrix}$

   En posant :

$$f : X = (S(t), I(t), R(t), D(t)) \underset{\mathbb{R}_+^4}{} \to (-rS(t)I(t), rS(t)I(t) - (a+b)I(t), aI(t), bI(t)) \underset{\mathbb{R}_- \times \mathbb{R}\mathbb{R}_+^2}{}$$

   Le système d'équations différentielles s'écrit :

$$\frac{d}{dt}X = f(X)$$

2. La ligne 25 montre que l'on doit pouvoir utiliser des opérations algébriques sur X et $f(X)$ donc il faut renvoyer un objet de type `ndarray` et non pas de type `list`.

```
1  def f(X):
2      """Fonction definissant l'equation differentielle"""
3      global r,a,b
4      S=X[0];I=X[1];R=X[2];D=X[3]
5      rhsS=-r*S*I
6      rhsI=r*S*I-(a+b)*I
7      rhsR=a*I
8      rhsD=b*I
9      return np.array([rhsS,rhsI,rhsR,rhsD])
10
11 #Parametres
12 tmax=25.
13 r=1.
14 a=0.4
15 b=0.1
16 X0=np.array([0.95,0.05,0.,0.])
17
18 N=250
19 dt=tmax/N
20
21 t=0
22 X=X0
23 tt=[t]
24 XX=[X]
25
26 #Methode d'Euler
27 for i in range(N):
28     t=t+dt
29     X=X+dt*f(X)
30     tt.append(t)
31     XX.append(X)
```

3. La méthode d'Euler est d'autant plus précise que le pas `dt` est petit c'est-à-dire, à intervalle fixé, que le nombre de point `N` est élevé. La figure 1 montre, par la dispersion des points autour des courbes, que

pour $N = 7$ l'erreur locale et l'erreur de convergence globale est plus importante que pour $N = 250$. La méthode d'Euler étant de complexité linéaire, le temps de calcul nécessaire à la simulation pour $N = 250$ est plus grand que pour $N = 7$.

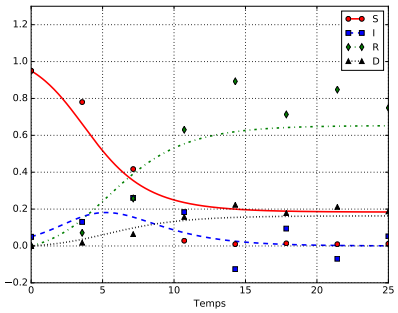

FIGURE 1 – Représentation graphique des quatre catégories S, I, R et D en fonction du temps pour $N = 7$ (points) et $N = 250$ (courbes).

4. Prise en compte de l'incubation.

```
def f(X,Itau):
    """
    Fonction definissant l'equation differentielle
    Itau est la valeur de I(t-p*dt)
    """
    global r,a,b
    S=X[0];I=X[1];R=X[2];D=X[3]
    rhsS=-r*S*Itau
    rhsI=r*S*Itau-(a+b)*I
    rhsR=a*I
    rhsD=b*I
    return np.array([rhsS,rhsI,rhsR,rhsD])

#Parametres
r=1.
a=0.4
b=0.1
X0=np.array([0.95,0.05,0.,0.])

tmax=25.
N=250
dt=tmax/N
p=50

t=0
X=X0
tt=[t]
XX=[X]

#Methode d'Euler
for i in range(N):
    t=t+dt
    if i<p:#t<tau,I(t)=I(0)=XX[0][1]
        Itau=XX[0][1]
    else:
        Itau=XX[i-p][1]
    X=X+f(X,Itau)*dt
    tt.append(t)
    XX.append(X)
```

5. `Itau` se calcule à chaque itération à l'aide de la fonction `rectangle`. En l'état, la complexité de l'algorithme est $O(Np)$ ce qui est peu efficace. En effet, on recalcule l'intégrale complètement à chaque itération i alors qu'il suffirait d'ajouter le terme `dt*XX[i-j][1]*h(j*dt)` et de soustraire `dt*XX[i-j-p][1]*h((j-p)*dt)`

à la valeur de `Itau` calculé à l'itération $i-1$. On gagnerait en complexité à prendre cet élément en compte mais on perdrait en clarté du programme.

```
def rectangle(XX,h,i,j,p,dt):
    Itau=0
    for j in range(p):
        if i-j<p:
            Itau+=dt*XX[0][1]*h(j*dt)
        else:
            Itau=dt*XX[i-j][1]*h(j*dt)
    return Itau

def f(X,Itau):
    """
    Fonction definissant l'equation differentielle
    Itau est la valeur de I(t-p*dt)
    """
    global r,a,b
    S=X[0];I=X[1];R=X[2];D=X[3]
    rhsS=-r*S*Itau
    rhsI=r*S*Itau-(a+b)*I
    rhsR=a*I
    rhsD=b*I
    return np.array([rhsS,rhsI,rhsR,rhsD])

#Parametres
r=1.
a=0.4
b=0.1
X0=np.array([0.95,0.05,0.,0.])

tmax=25.
N=250
dt=tmax/N
p=50

t=0
X=X0
tt=[t]
XX=[X]

#Methode d'Euler
for i in range(N):
    t=t+dt
    Itau=rectangle(XX,h,i,j,p,dt)
    X=X+f(X,Itau)*dt
    tt.append(t)
    XX.append(X)
```

2 Modélisation dans des grilles

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.2.

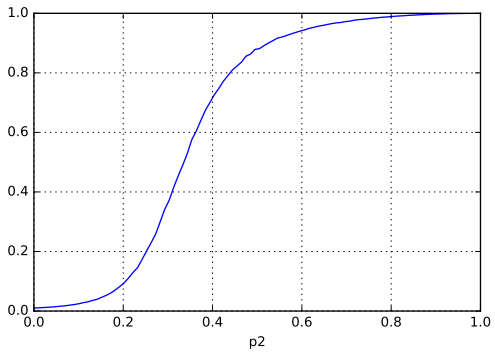

FIGURE 2 – Représentation de la proportion de la population qui a été atteinte par la maladie pendant la simulation en fonction de la probabilité p_2.

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:
        m=(d+g)//2
        if Lxa[m]<=0.5:
            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