Analyse des modèles de propagation d'épidémies

Travail de Nils Xhoffray n°53899 en collaboration
avec Ibrahim El Shourbagi n°11595

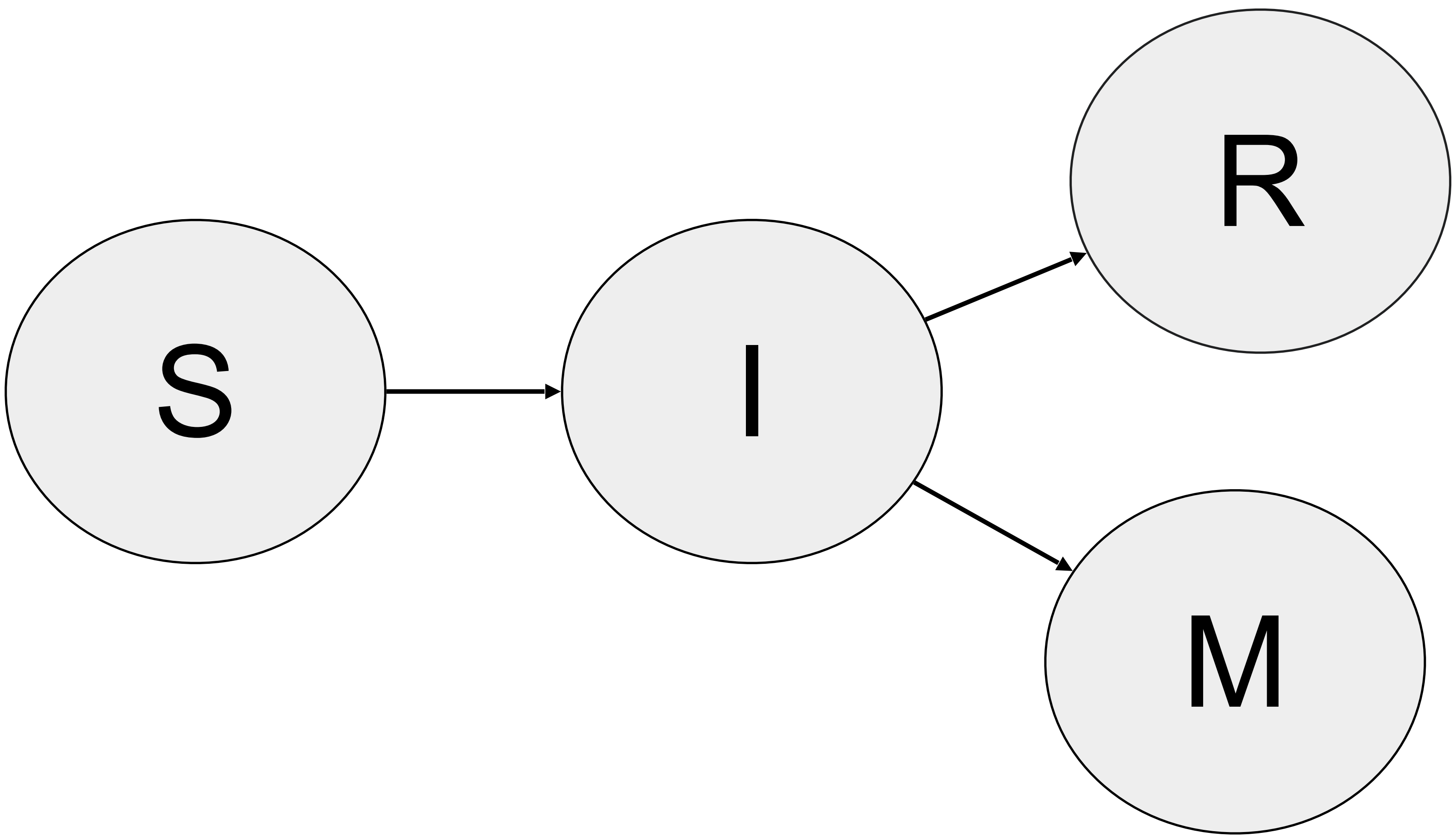
Paramètres d'une épidémie

- **Taux de reproduction** : $R_0 = \rho c d$

  - ρ probabilité de transmission
  - c nombre de contacts par unité de temps
  - d durée de la période contagieuse

- **Temps de doublement** : $T_d = \dfrac{d \ln(2)}{R_0 - 1}$

Modèle compartimental



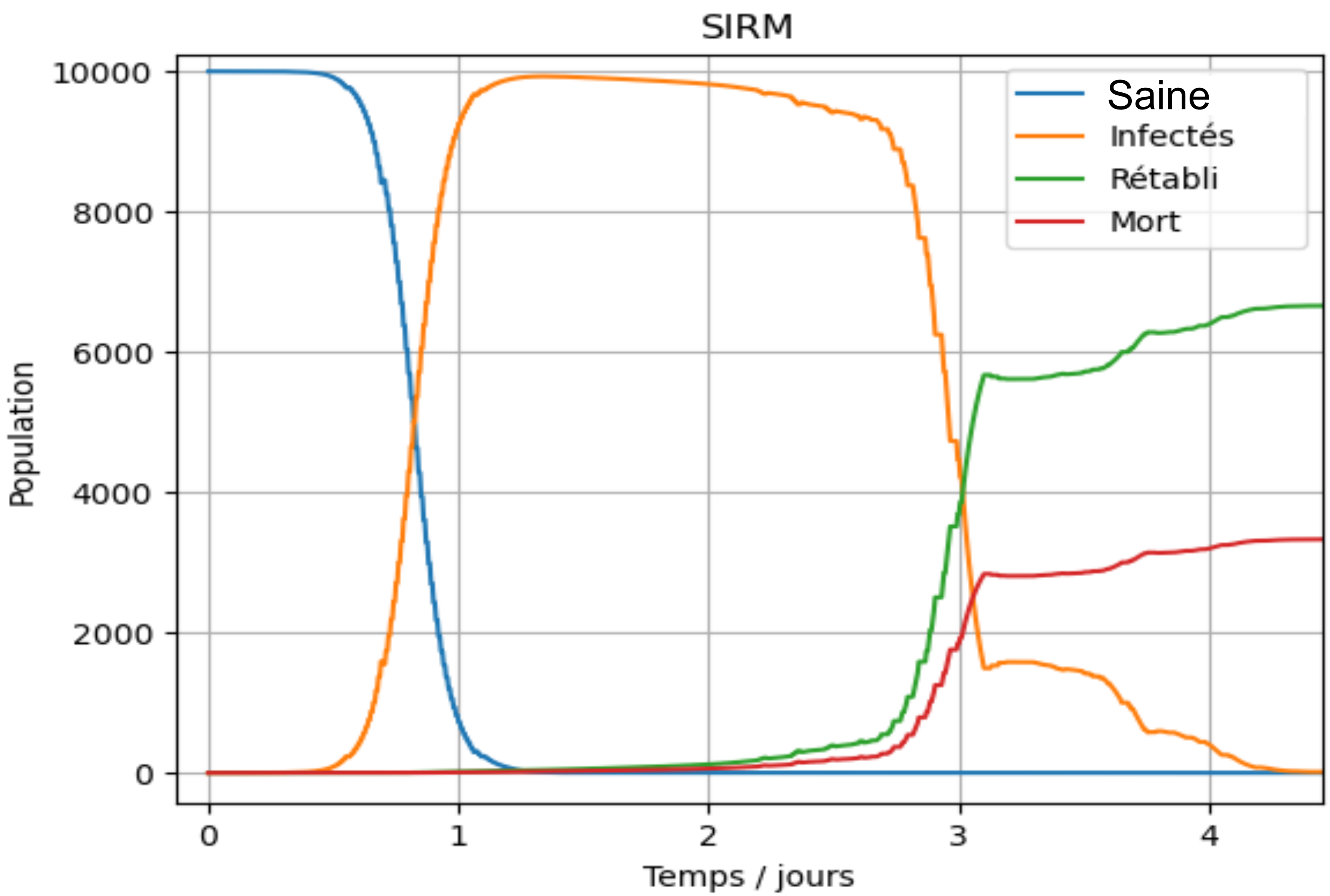
- S : Sain
- I : Infectés
- R : Rétablis
- M : Décédés

Modèle compartimental

$$
\begin{cases}
\dfrac{\mathrm{d}S}{\mathrm{d}t} = -\alpha \cdot S \cdot I \\[2ex]
\dfrac{\mathrm{d}I}{\mathrm{d}t} = \alpha \cdot S \cdot I - \beta \cdot I - \gamma \cdot I \\[2ex]
\dfrac{\mathrm{d}R}{\mathrm{d}t} = \beta \cdot I \\[2ex]
\dfrac{\mathrm{d}M}{\mathrm{d}t} = \gamma \cdot I
\end{cases}
$$

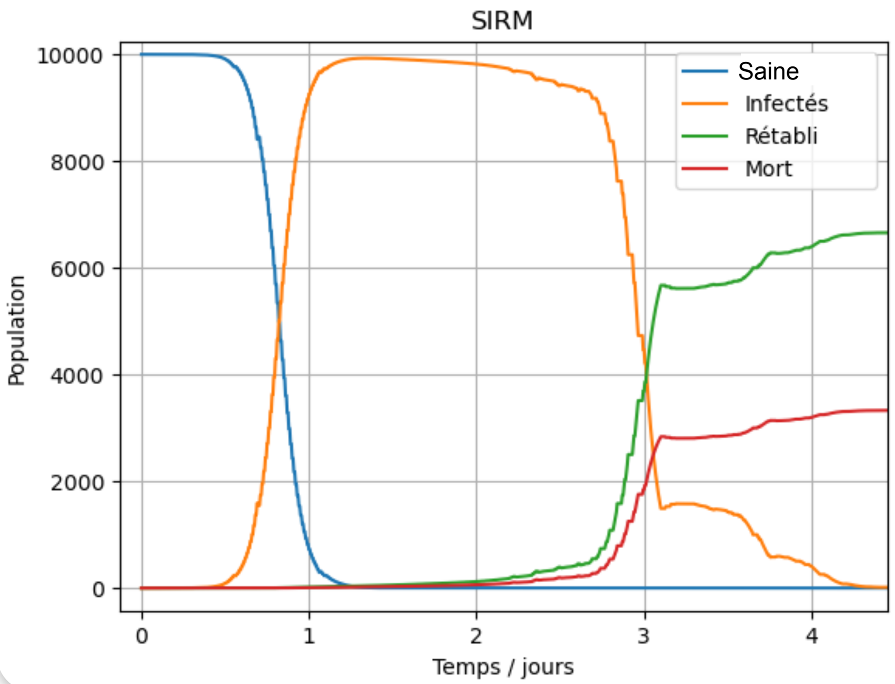
- S : Sain
- I : Infectés
- R : Rétablis
- M : Décédés

Modèle compartimental

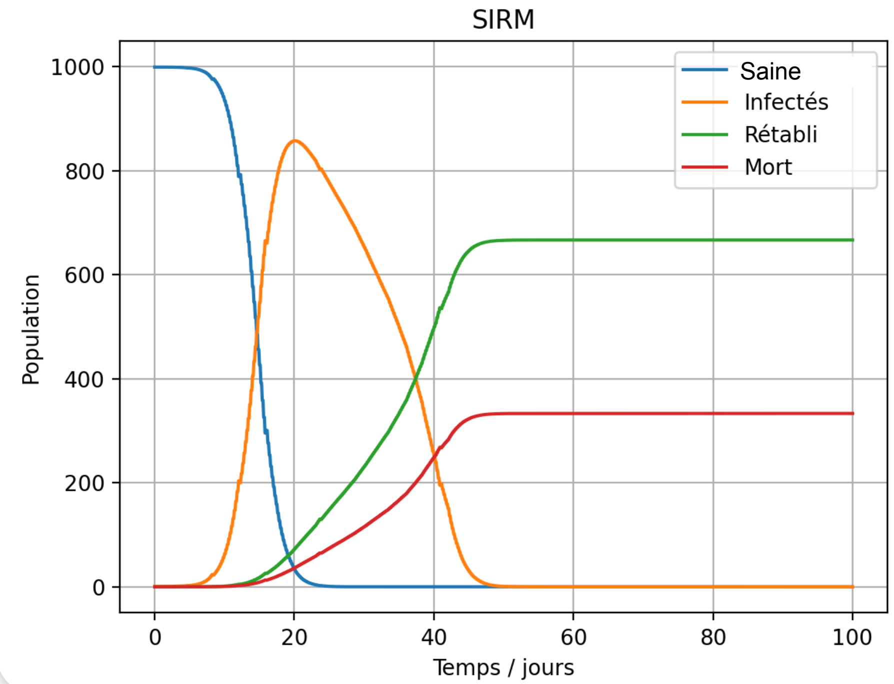


- Contamination : 0.1
- Guérison : 0.6
- Mortalité : 0.3
- Population : 10000
- Temps maximum : 50

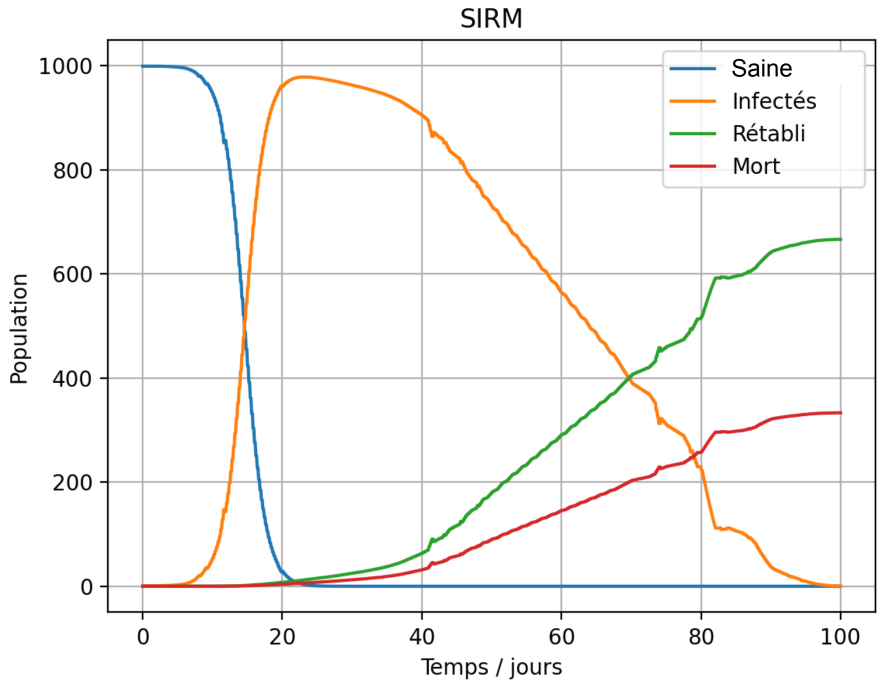
Modèle compartimental



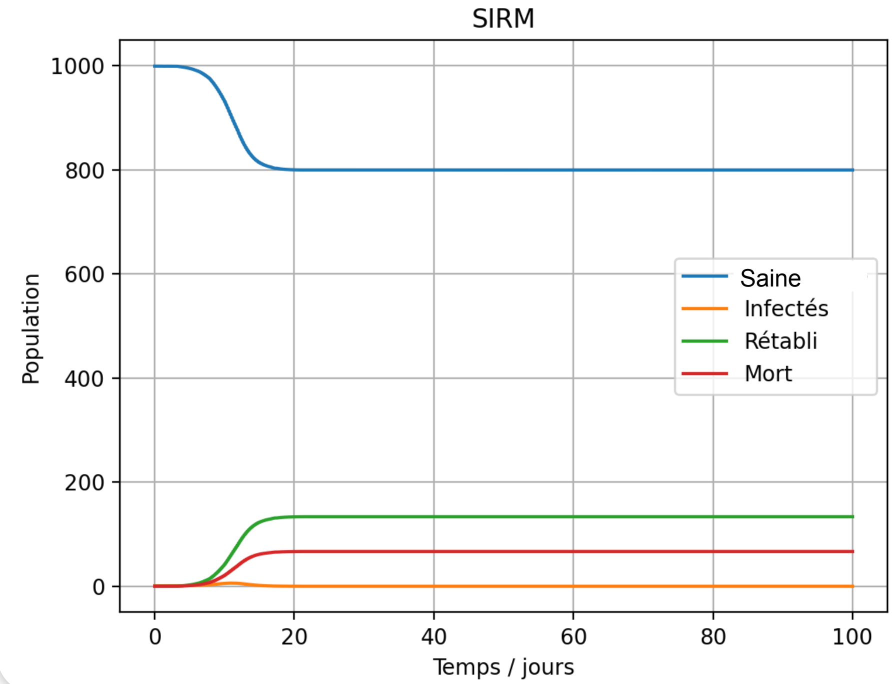
•contamination : 3.28
•guérison : 0.6
•mortalité : 0.3
•population : 10000
•temps maximum : 50



•contamination : 0.028
•guérison : 0.6
•mortalité : 0.3
•population : 10000
•temps maximum : 50

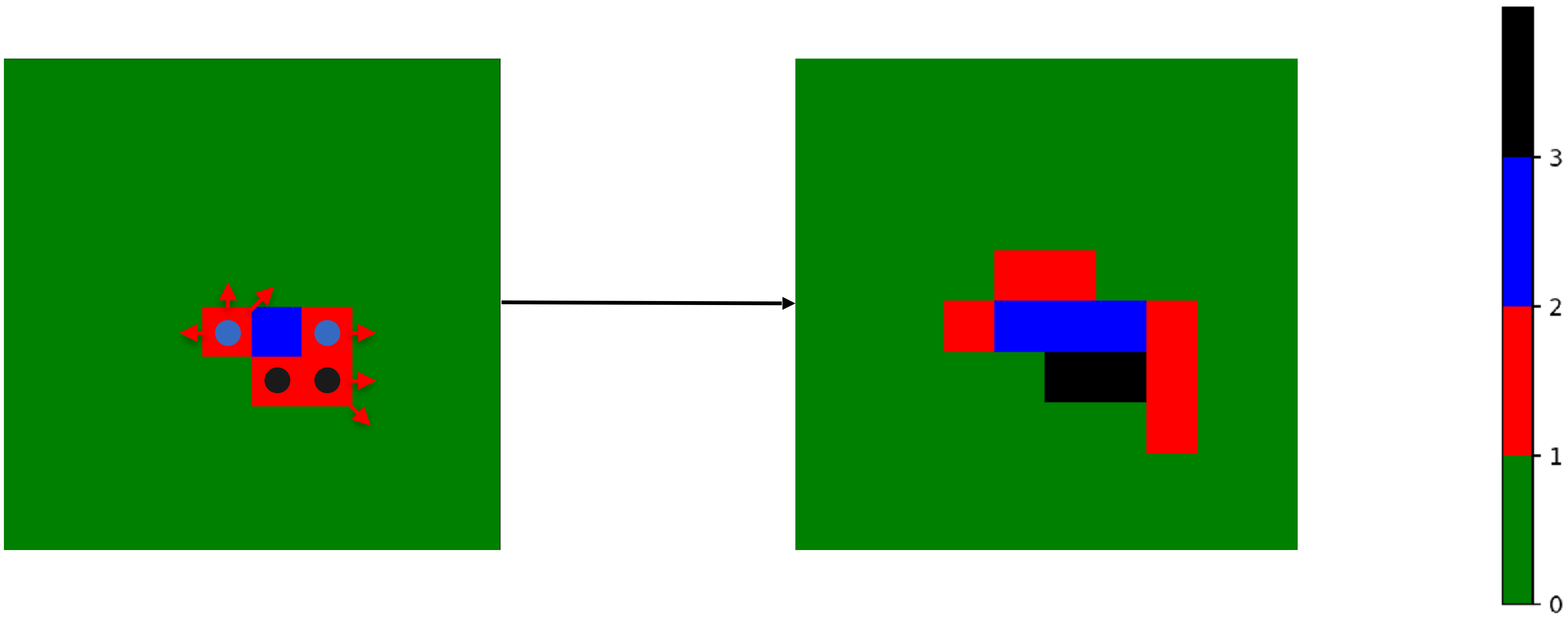


•contamination : 0.28
•guérison : 0.6
•mortalité : 0.3
•population : 10000
•temps maximum : 50

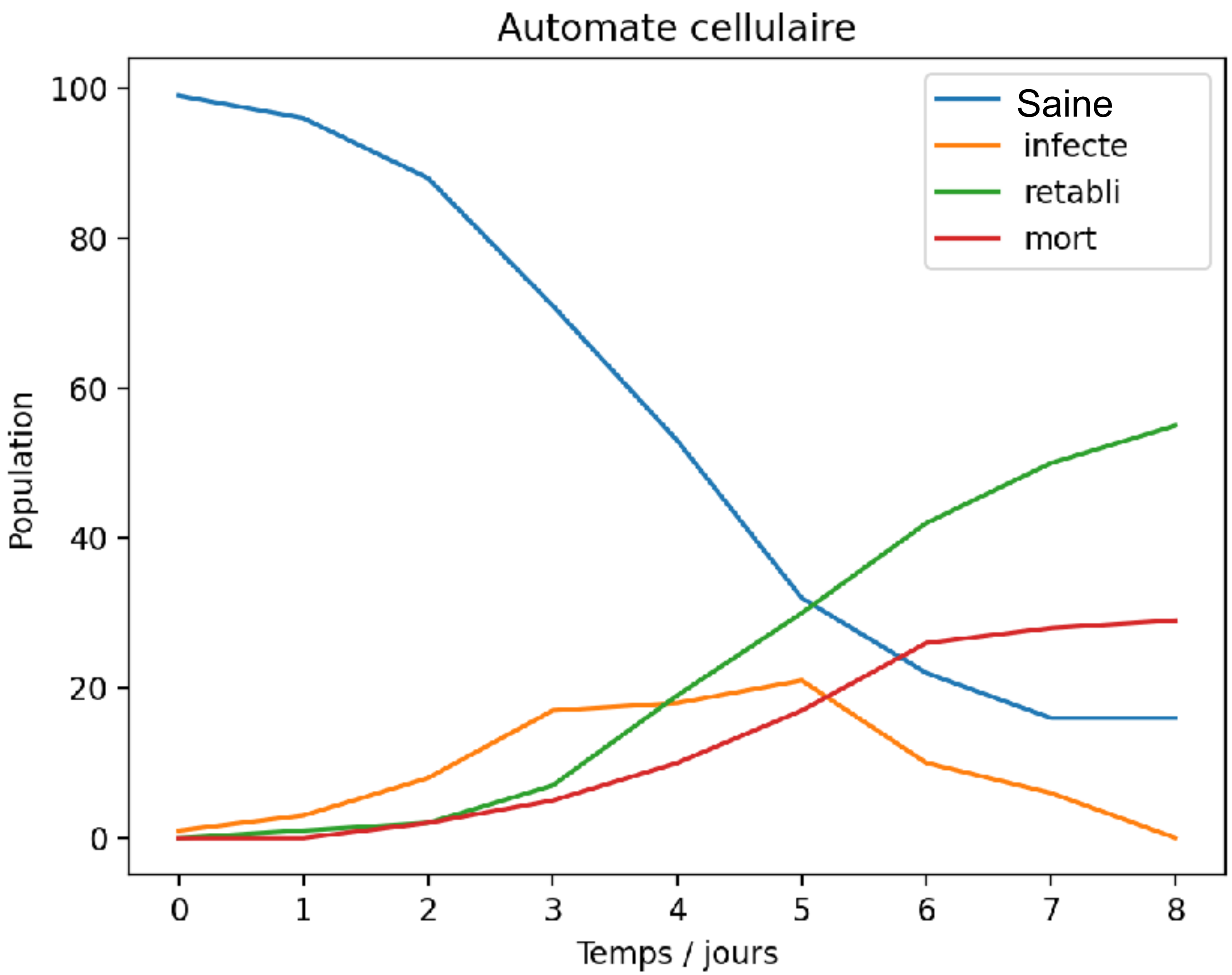


•contamination : 0.0011
•guérison : 0.6
•mortalité : 0.3
•population : 10000
•temps maximum : 50

Automate cellulaire

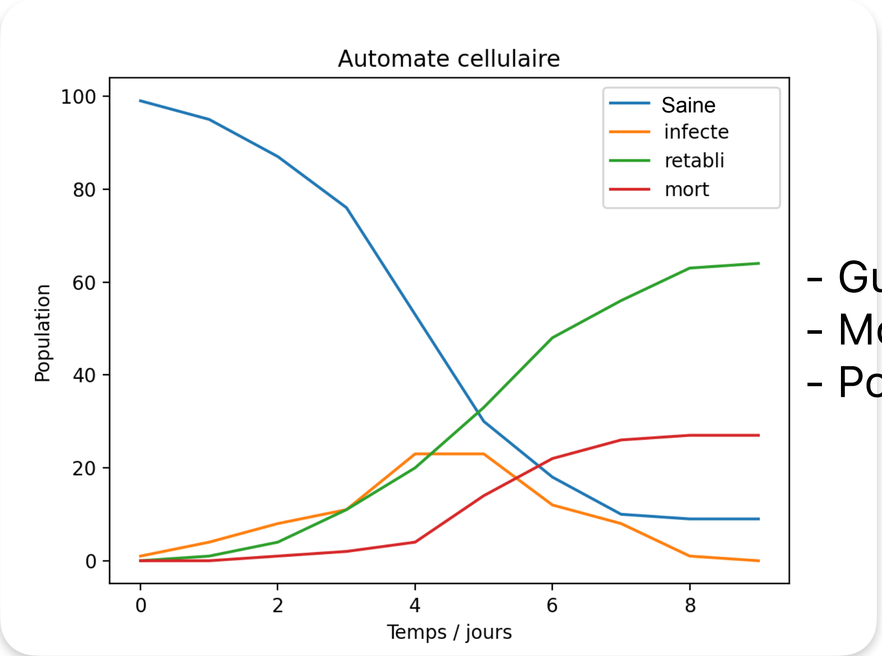


Automate cellulaire

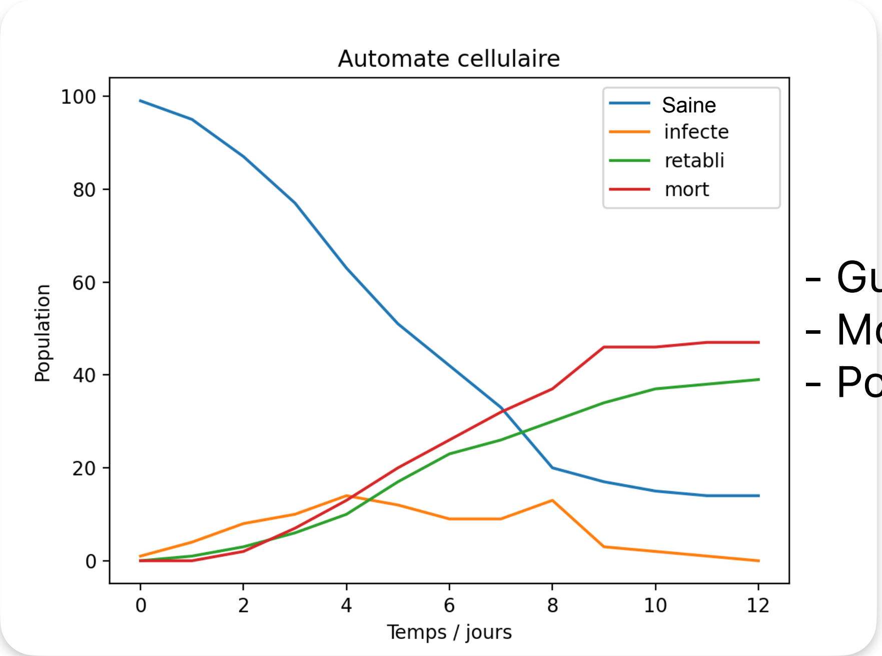


- Guérison : 0.6
- Mortalité : 0.5
- Population : 100

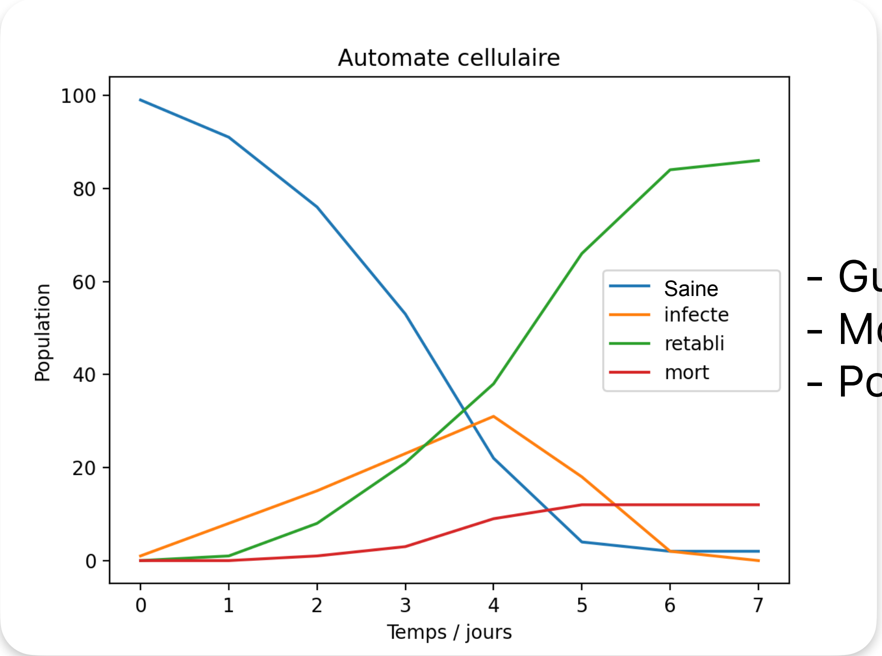
Automate cellulaire



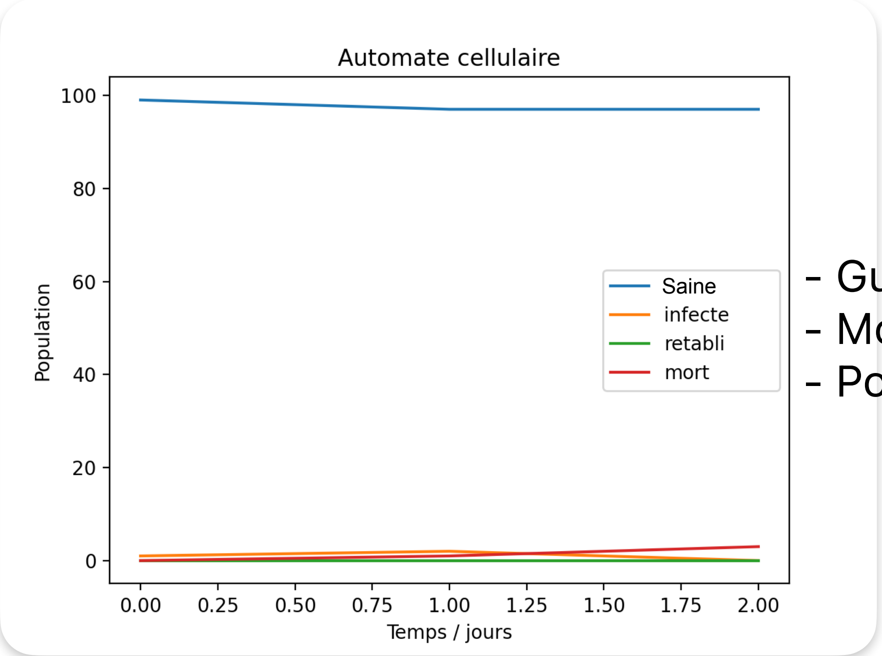
- Guérison : 0.6
- Mortalité : 0.3
- Population : 100



- Guérison : 0.5
- Mortalité : 0.5
- Population : 100

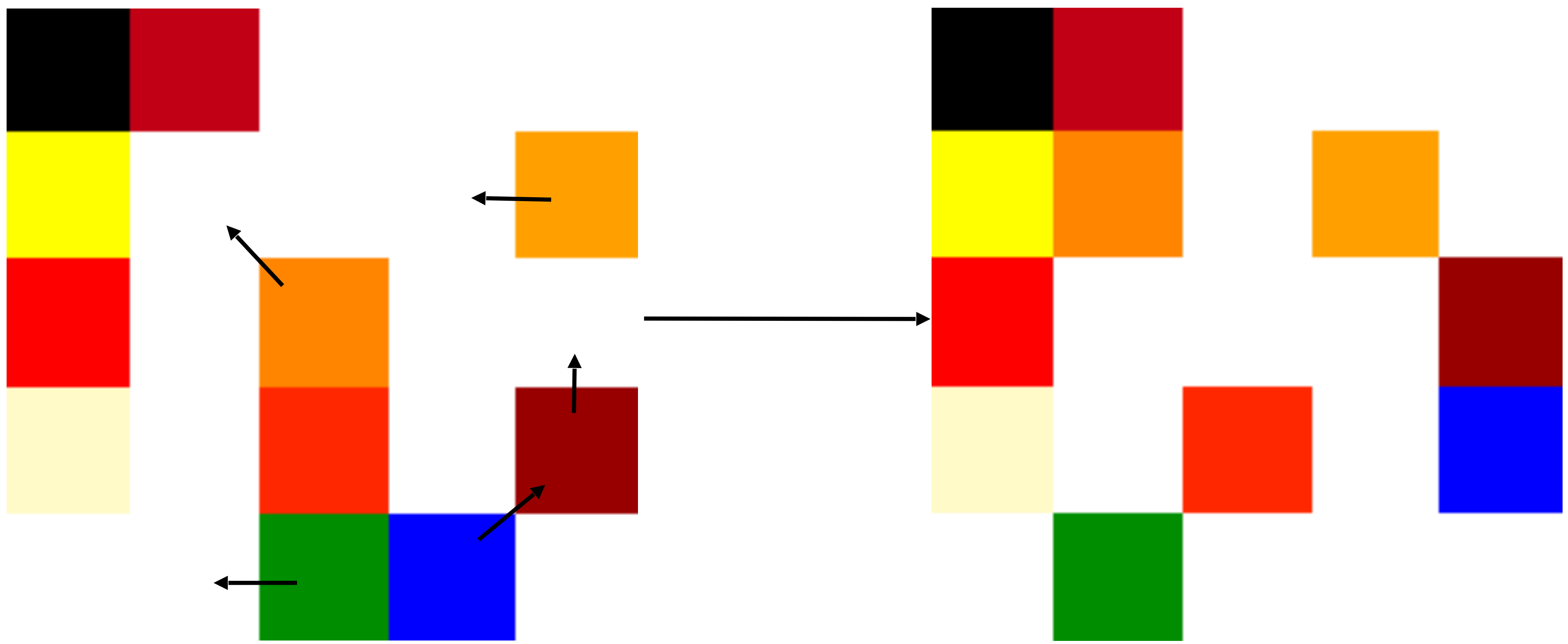


- Guérison : 0.9
- Mortalité : 0.1
- Population : 100

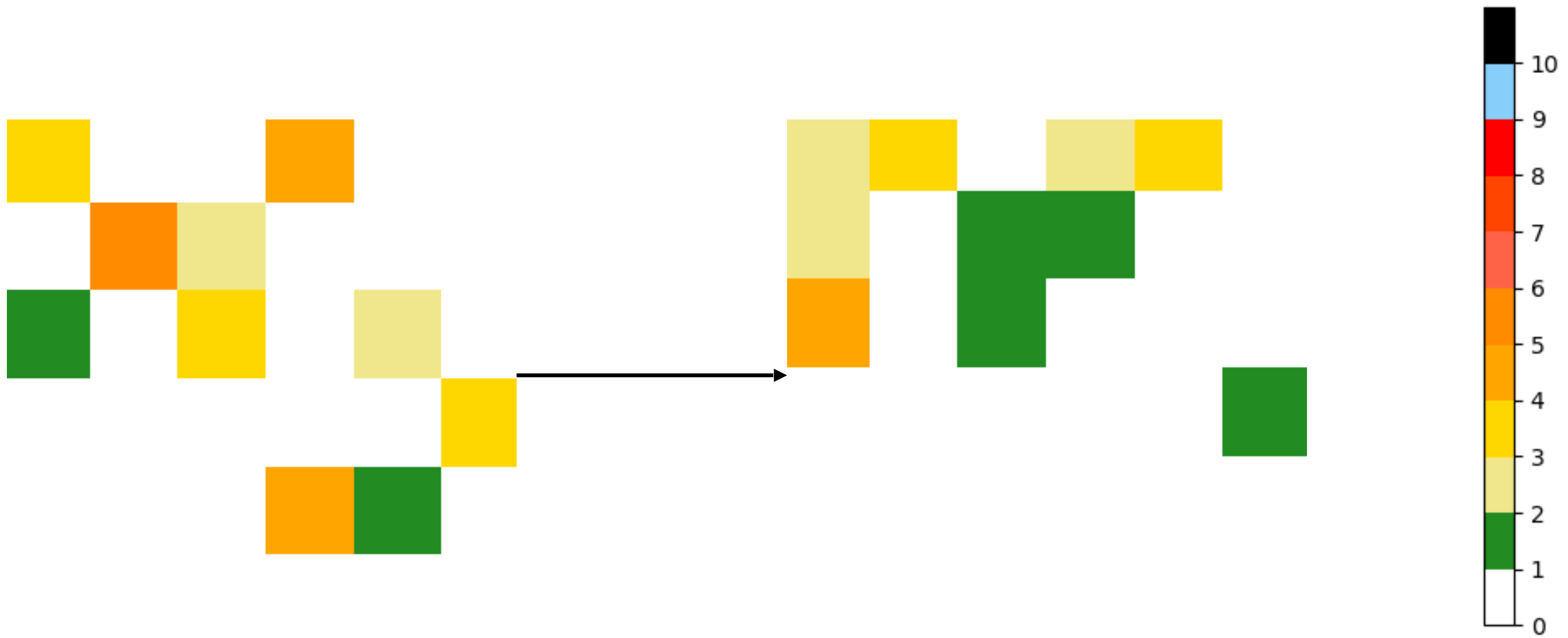


- Guérison : 0.9
- Mortalité : 0.1
- Population : 100

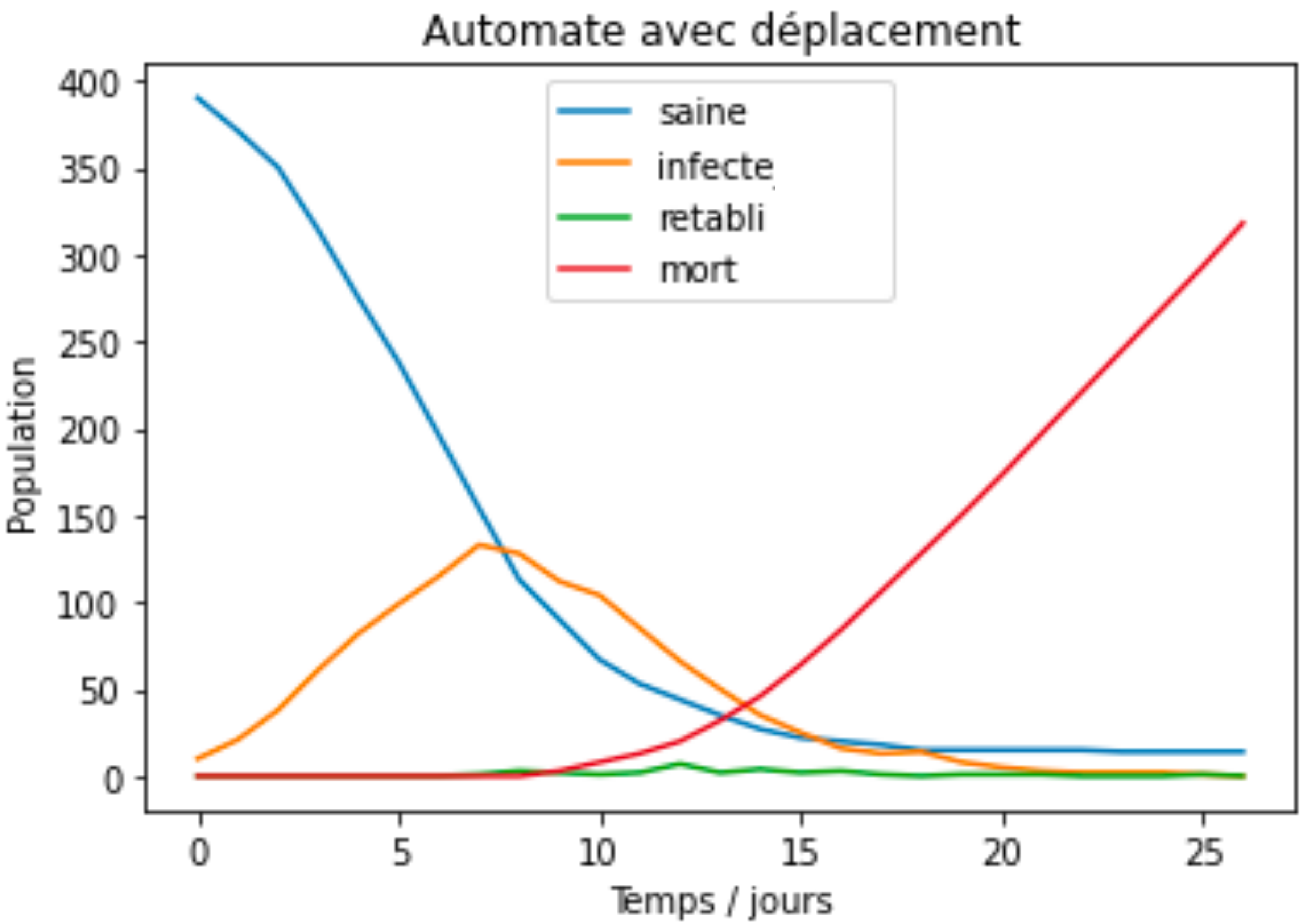
Automate cellulaire avec déplacement



Automate cellulaire avec déplacement

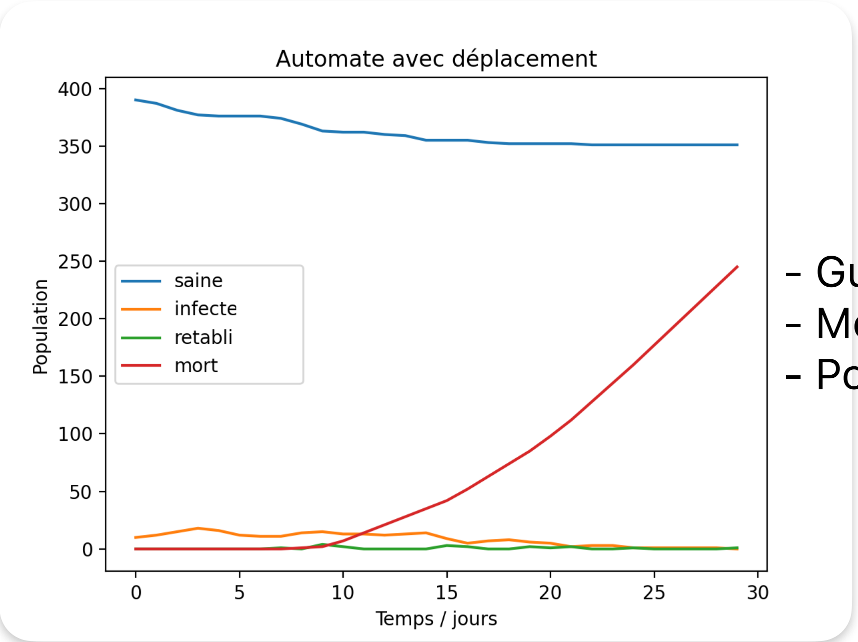


Automate cellulaire avec déplacement

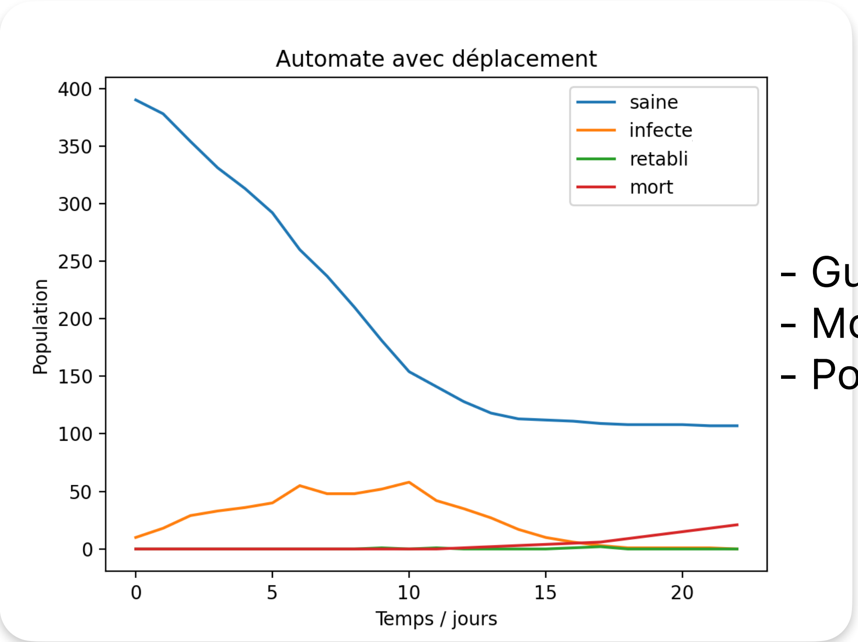


- guérison : 0.6
- mortalité : 0.3
- population : 400

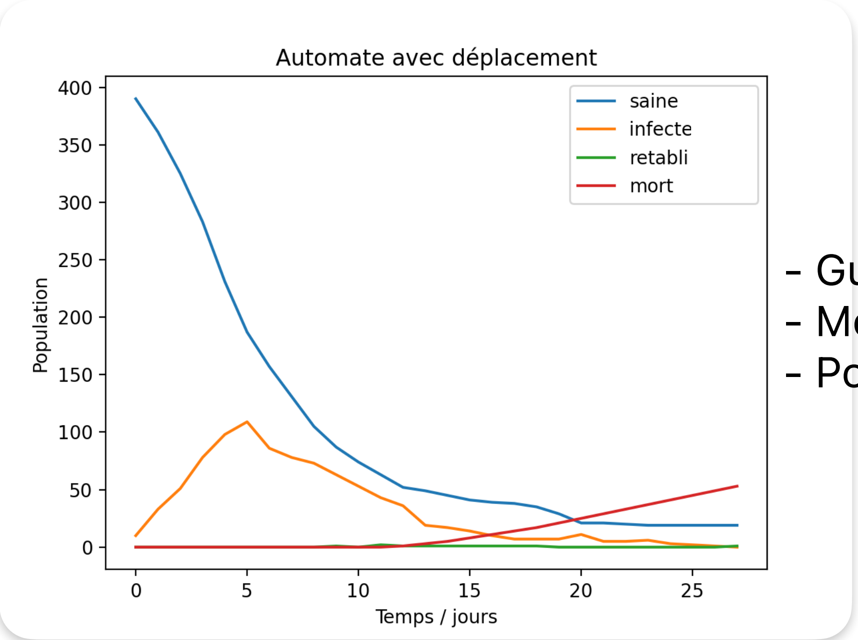
Automate cellulaire avec déplacement



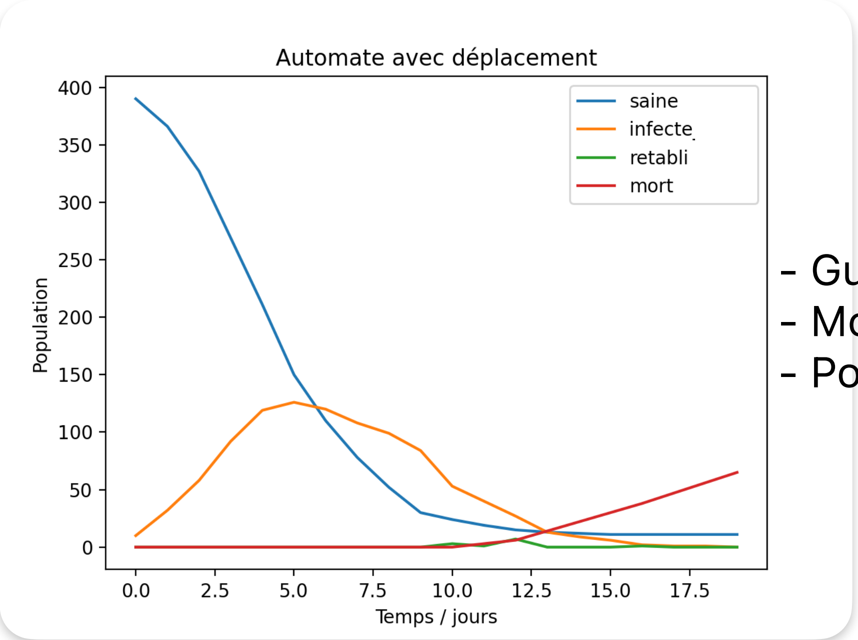
- Guérison : 0.1
- Mortalité : 0.1
- Population : 400



- Guérison : 0.5
- Mortalité : 0.5
- Population : 400

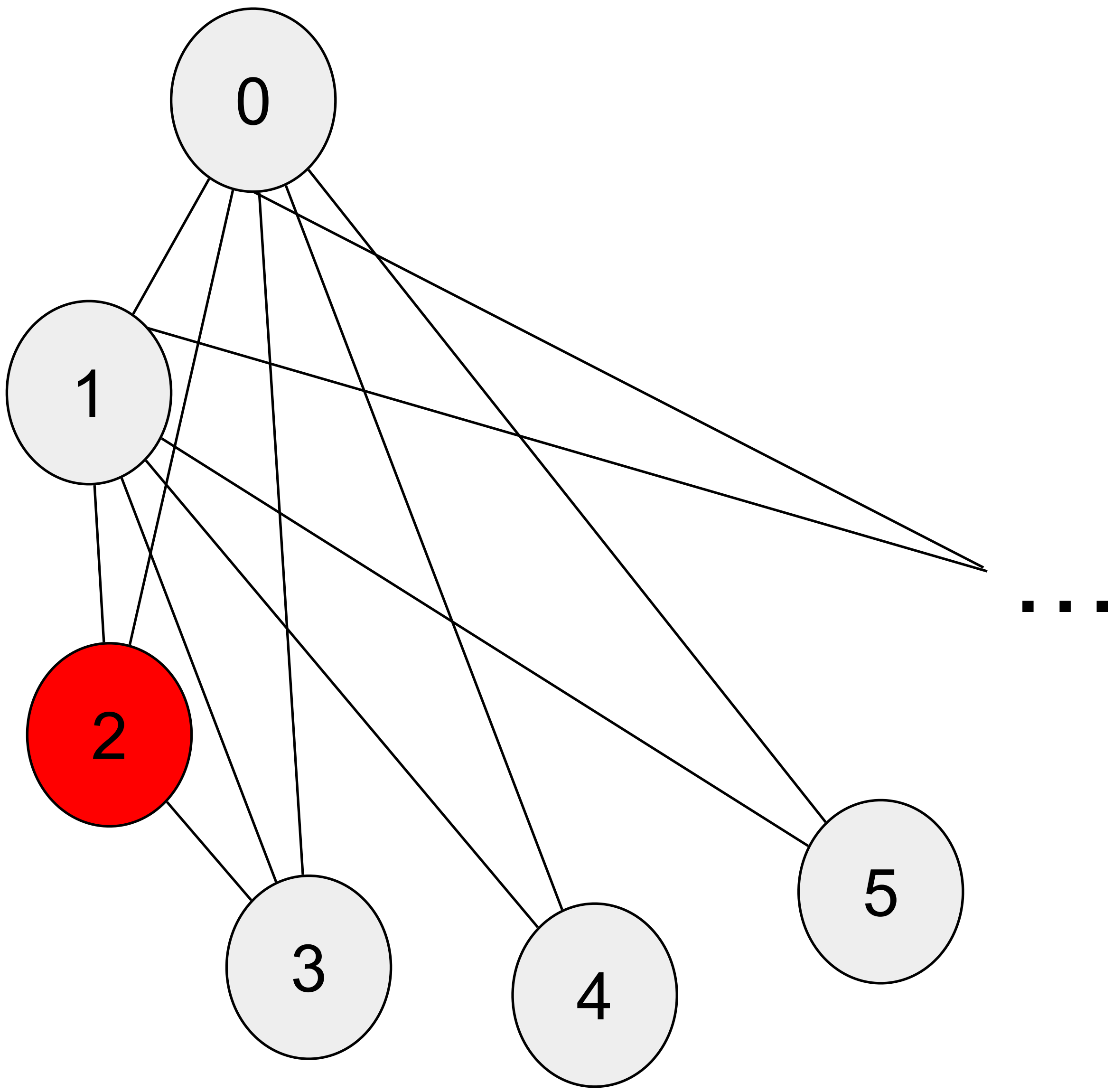


- Guérison : 0.4
- Mortalité : 0.8
- Population : 400



- Guérison : 0.9
- Mortalité : 0.4
- Population : 400

Modèle avec voisinage

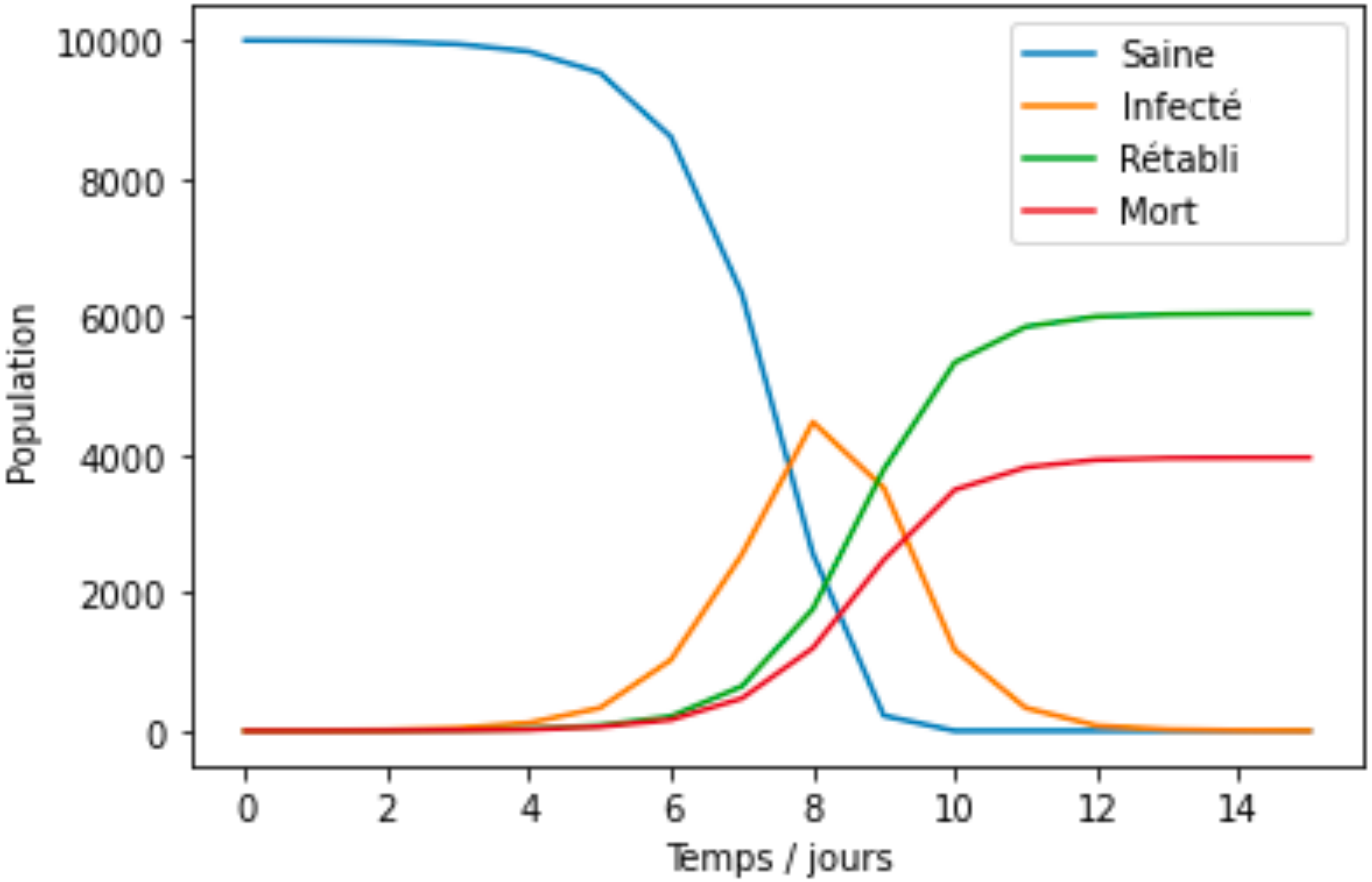


Listes d'adjacences:

$$\left[\left[1,\ 2,\ \ldots\right],\ \left[0,\ 2,\ \ldots\right],\ \ldots\right]$$

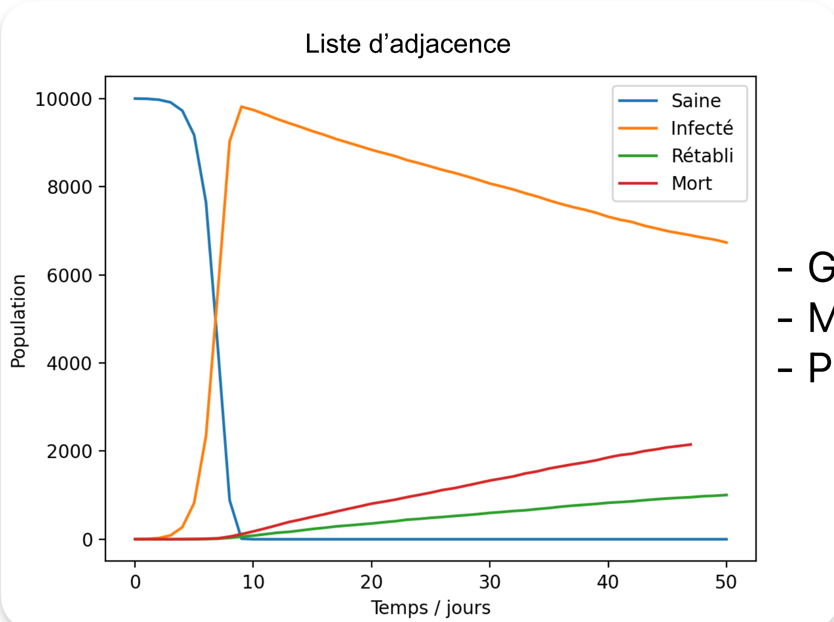
Modèle avec voisinage

Liste d'adjacence

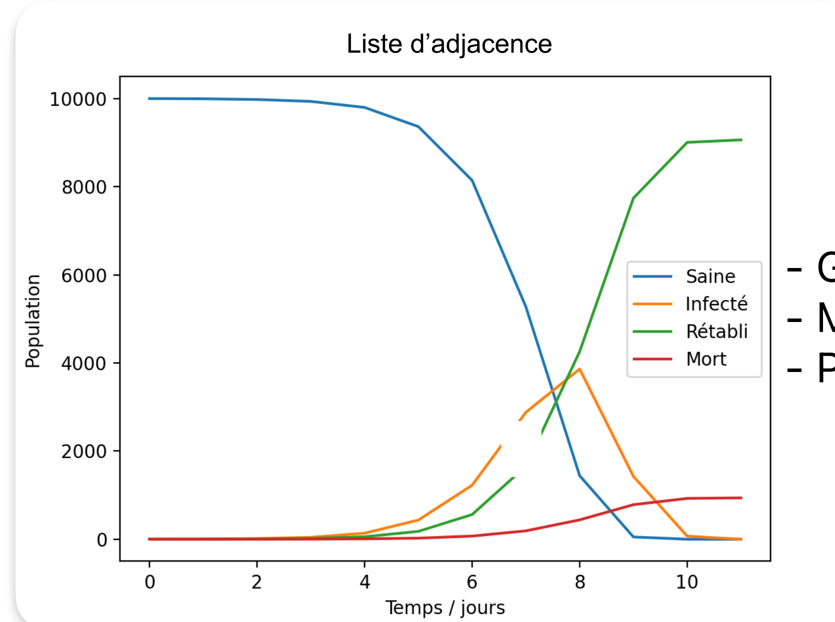


- guérison : 0.63
- mortalité : 0.29
- population : 10000
- nombres de voisins moyen : 3

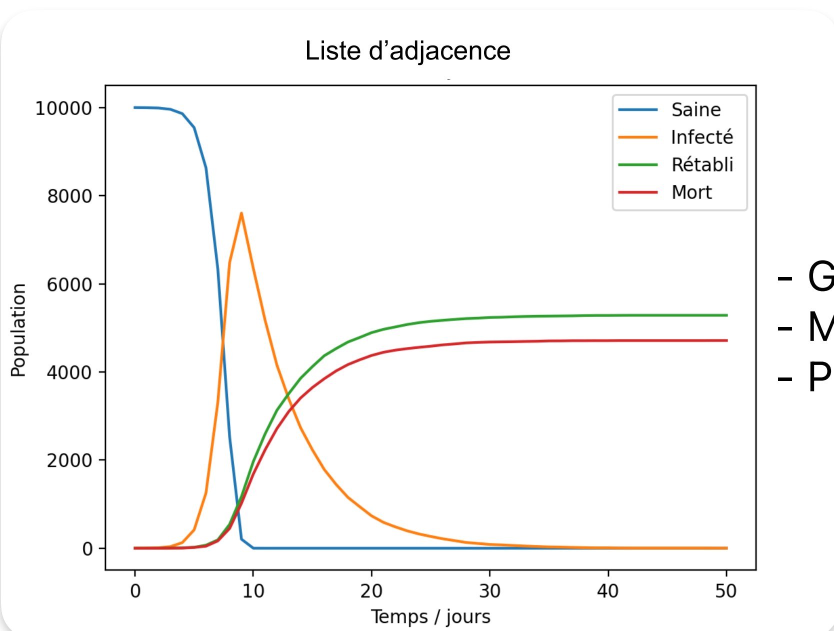
Modèle avec voisinage



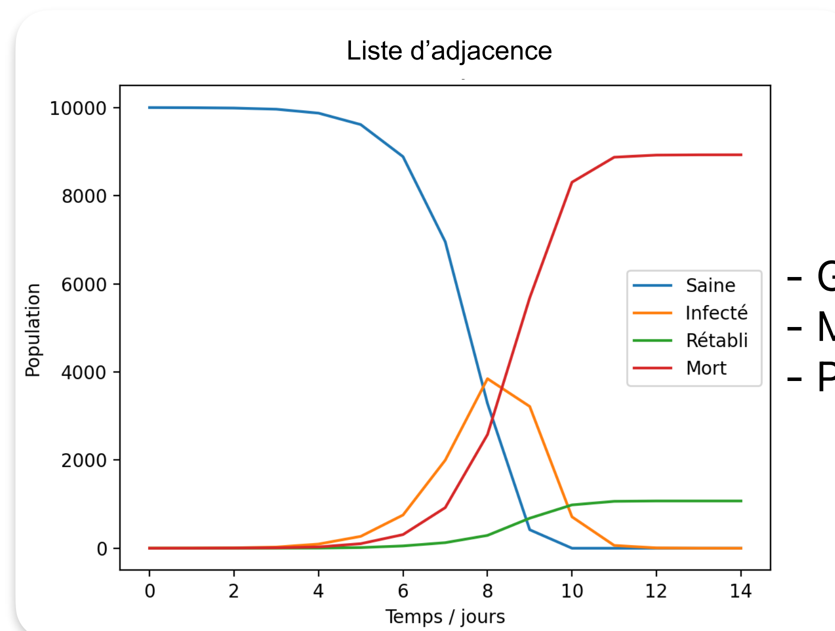
- Guérison : 0.0063
- Mortalité : 0.0029
- Population : 400



- Guérison : 0.9
- Mortalité : 0.9
- Population : 400

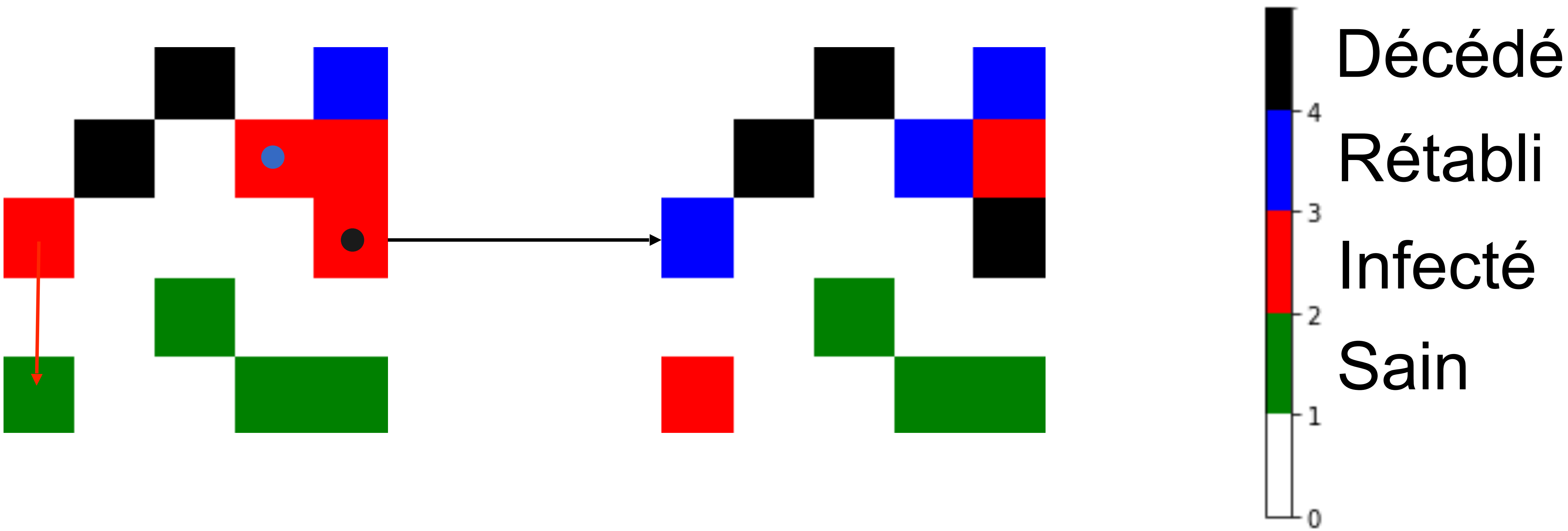


- Guérison : 0.1
- Mortalité : 0.1
- Population : 400

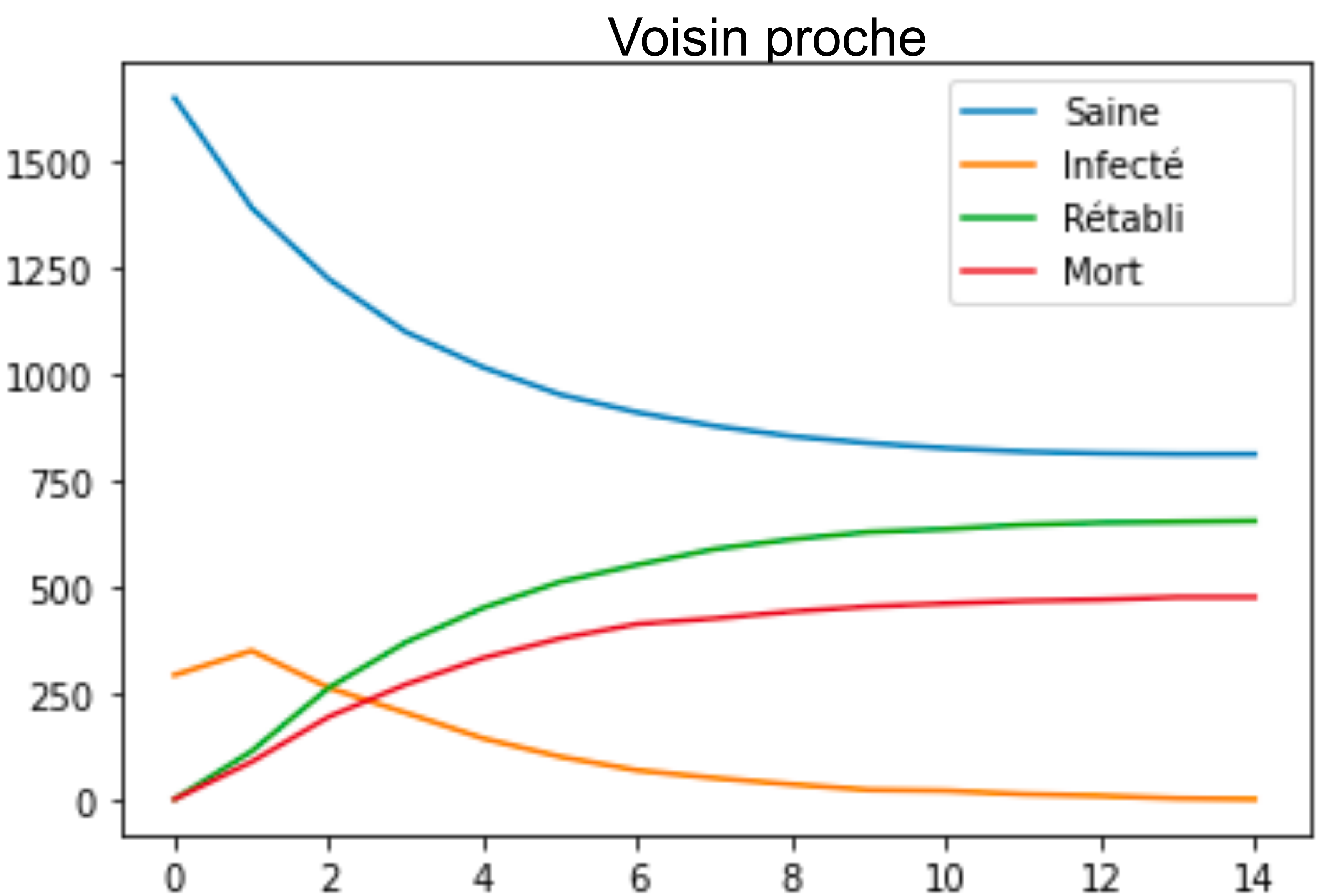


- Guérison : 0.9
- Mortalité : 0.1
- Population : 400

Modèle avec voisin proche

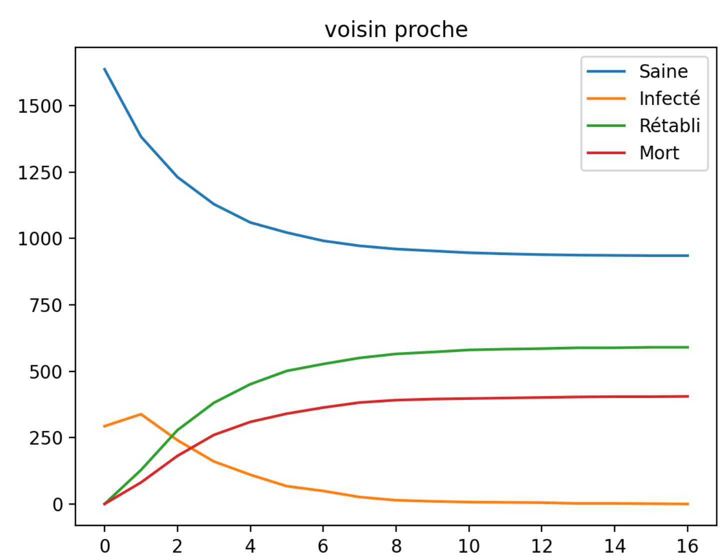


Modèle avec voisin proche

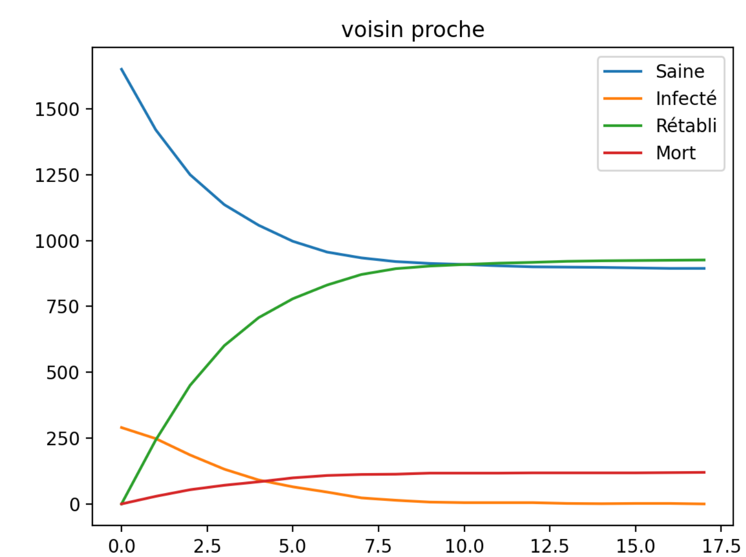


- taille de la grille : 60 x 60
- guérison : 0.6
- mortalité : 0.3
- population : 2500
- infecté initiaux : 300
- nombres de voisins : 1

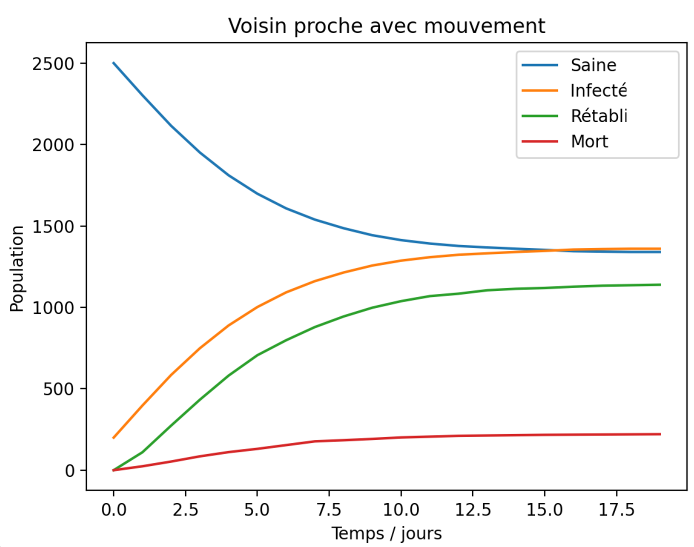
Modèle avec voisin proche



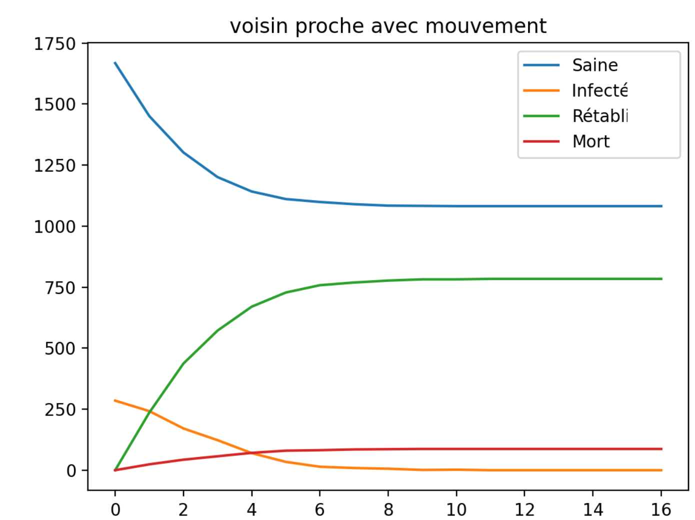
- taille de la grille : 60 x 60
- guérison : 0.6
- mortalité : 0.3
- population : 2500
- infecté initiaux : 300
- nombres de voisins : 1



- taille de la grille : 60 x 60
- guérison : 0.9
- mortalité : 0.1
- population : 2500
- infecté initiaux : 300
- nombres de voisins : 1

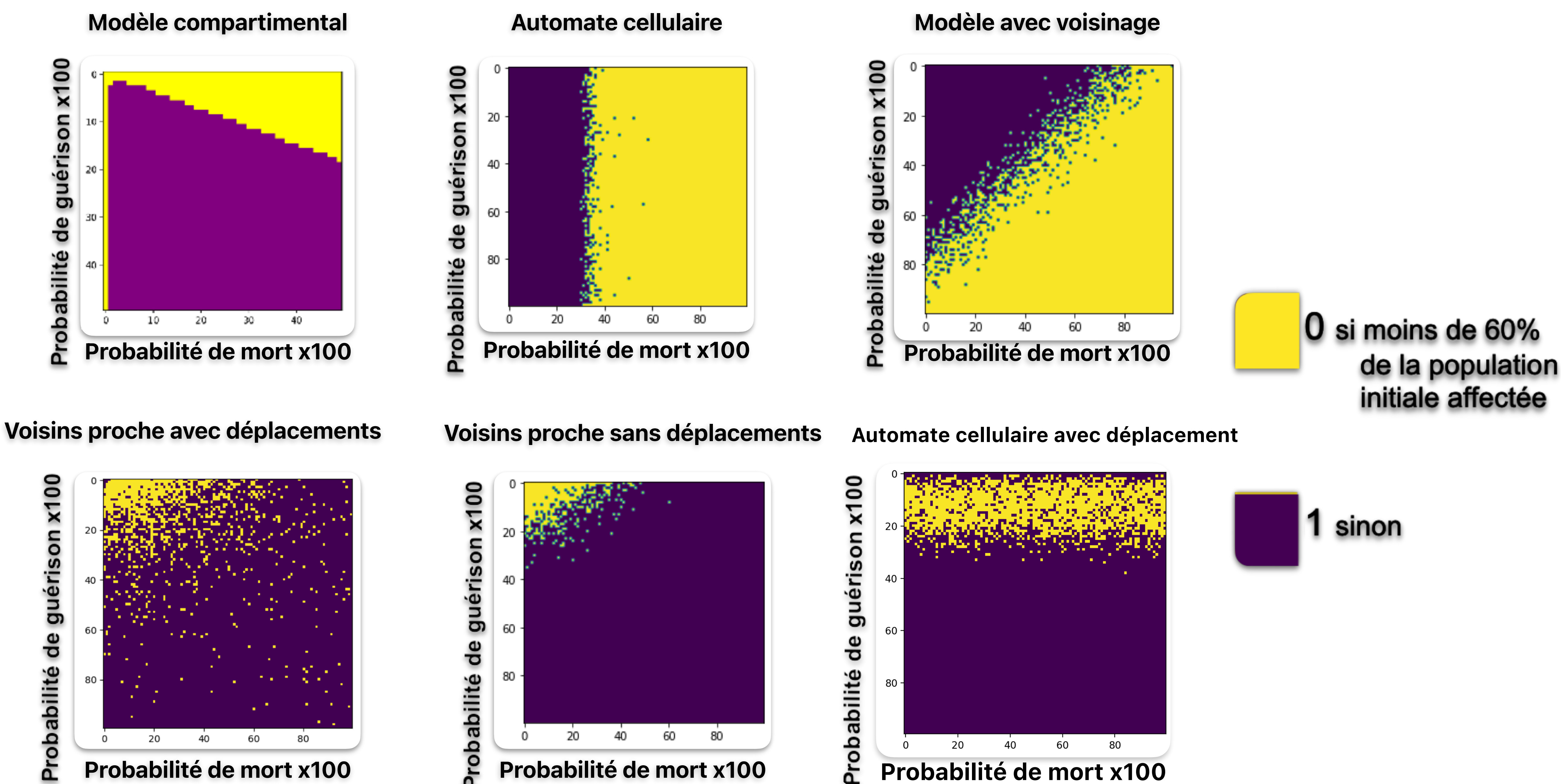


- taille de la grille : 60 x 60
- guérison : 0.6
- mortalité : 0.3
- population : 2500
- infecté initiaux : 300
- nombres de voisins : 1

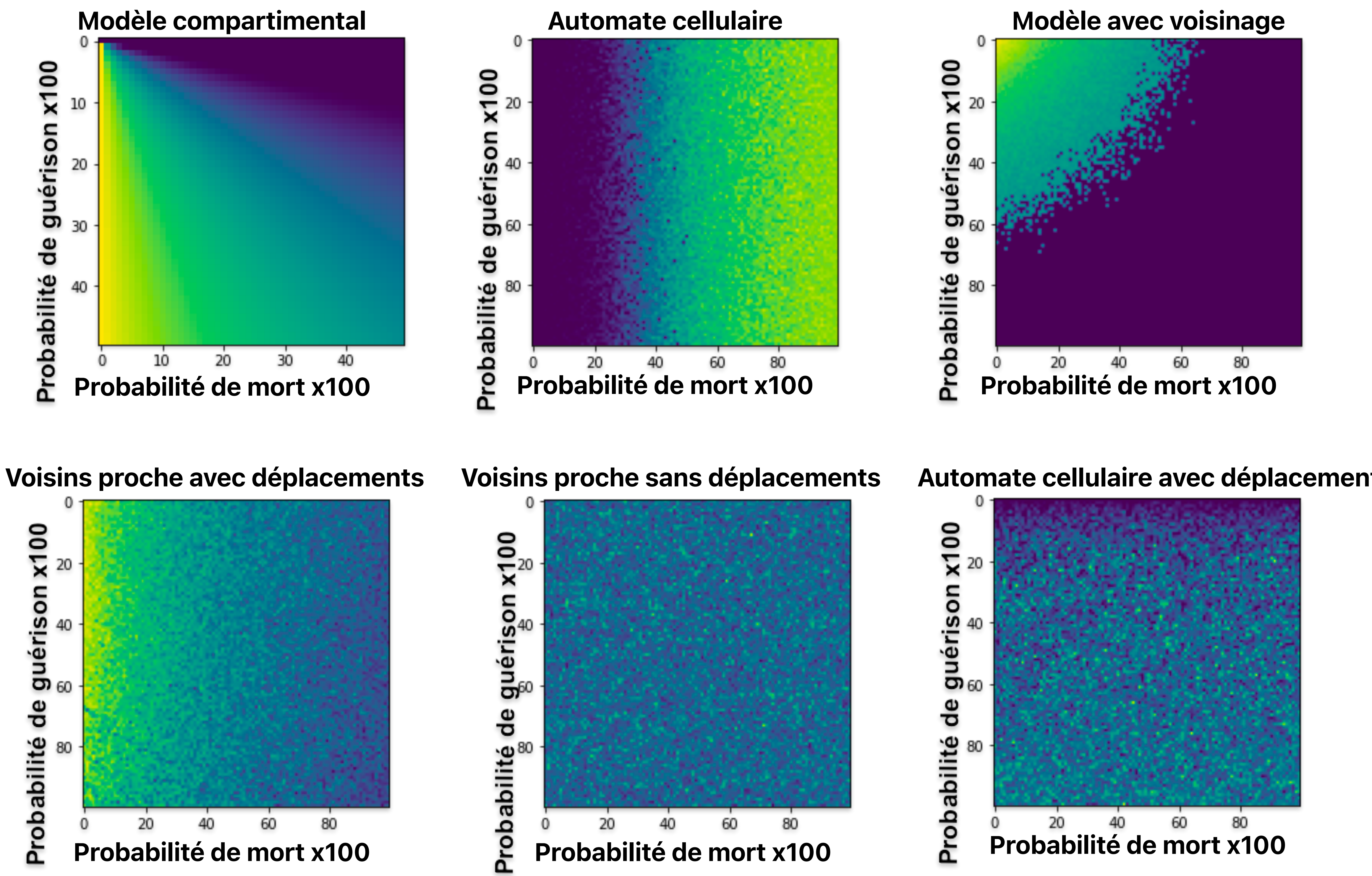


- taille de la grille : 60 x 60
- guérison : 0.9
- mortalité : 0.1
- population : 2500
- infecté initiaux : 300
- nombres de voisins : 1

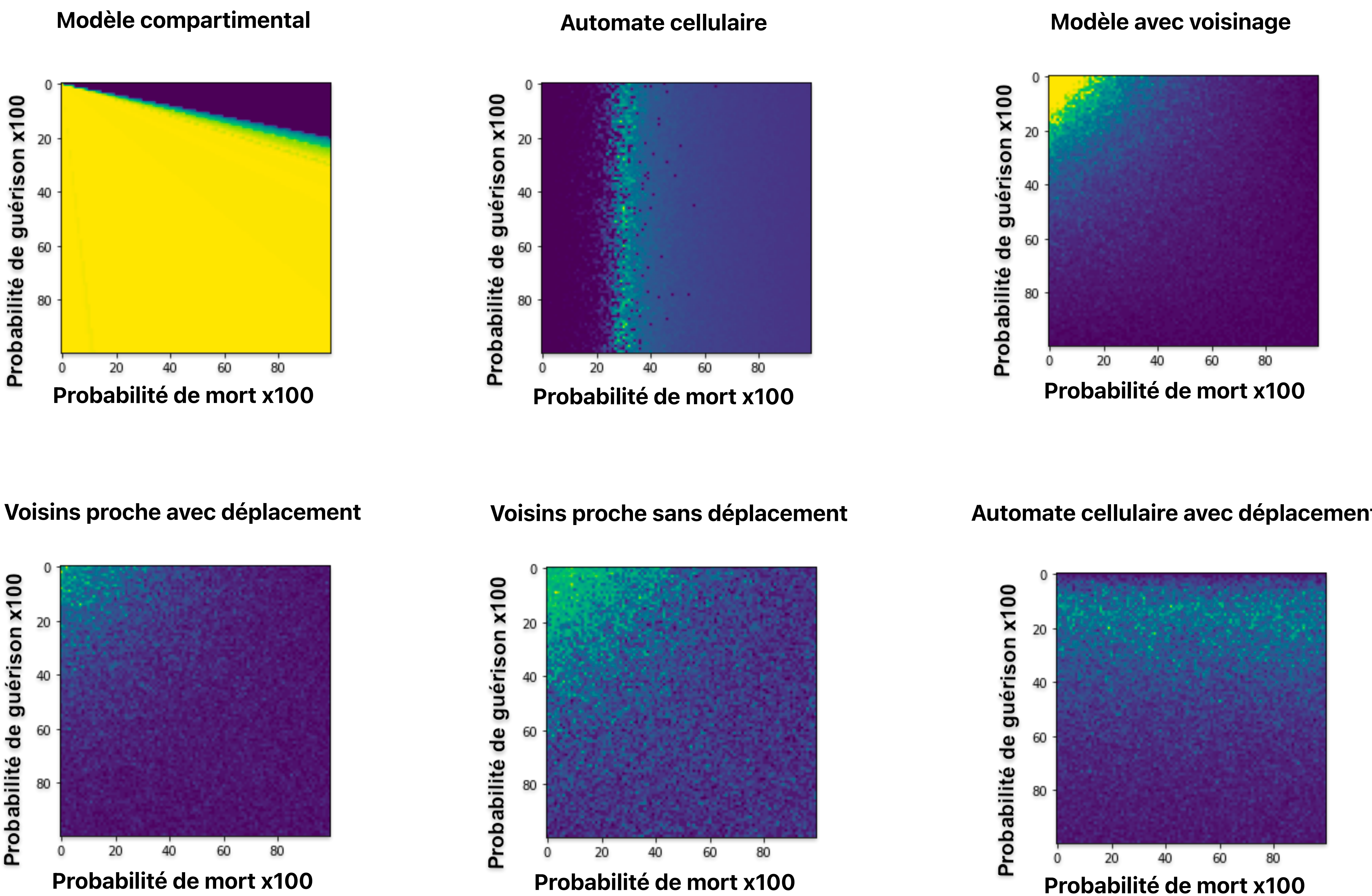
Contamination en fonction des probabilités



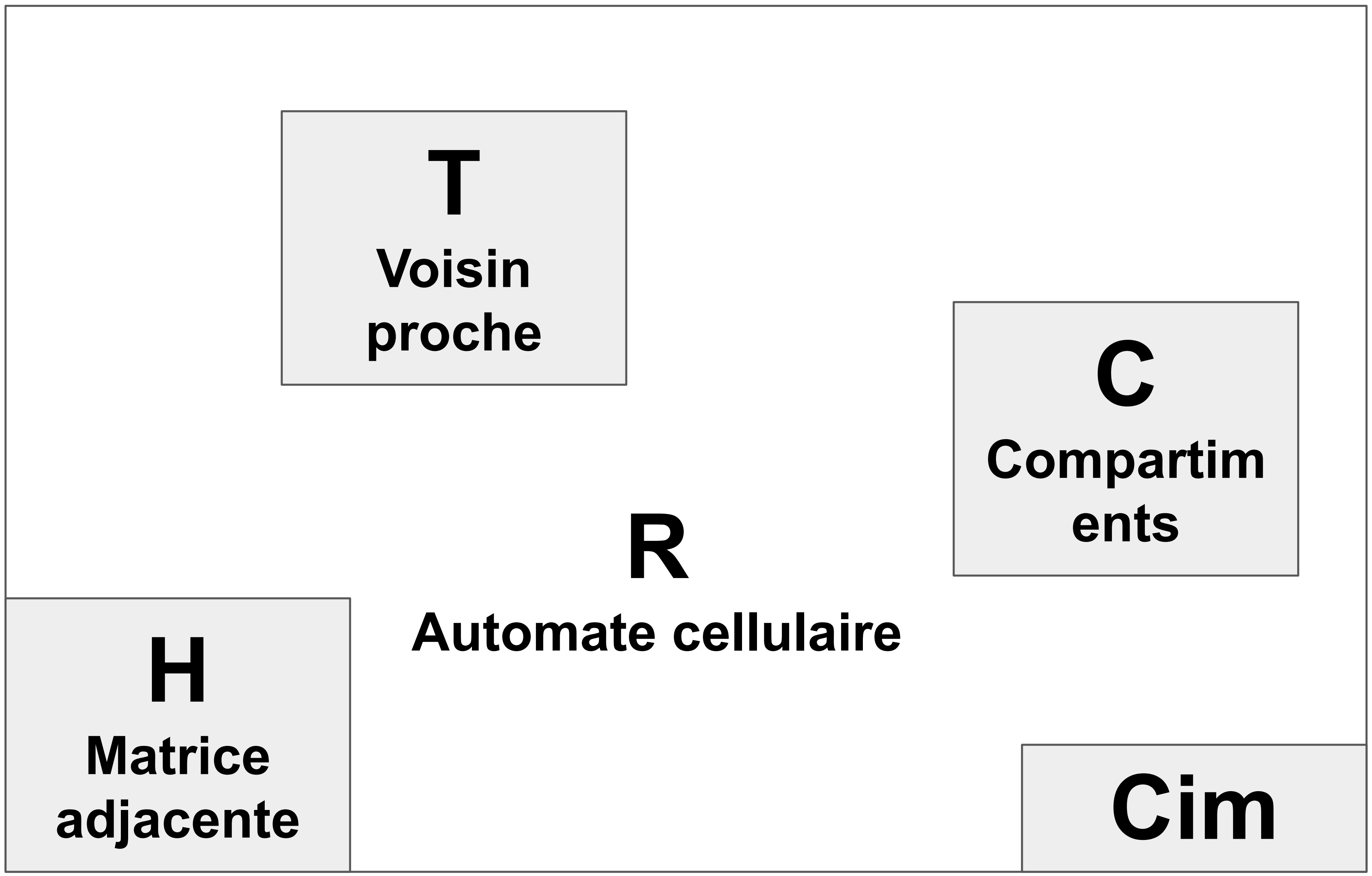
Contamination en fonction des probabilités



Durée des épidémies



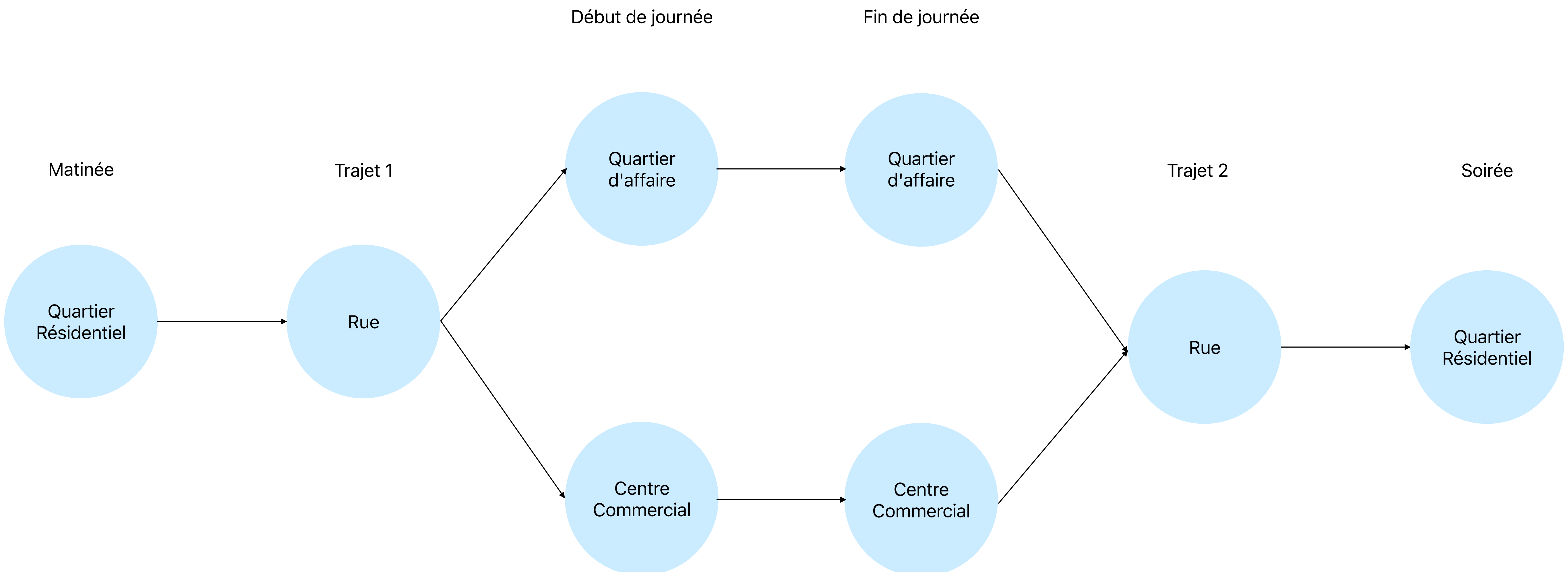
Simulation d’une ville



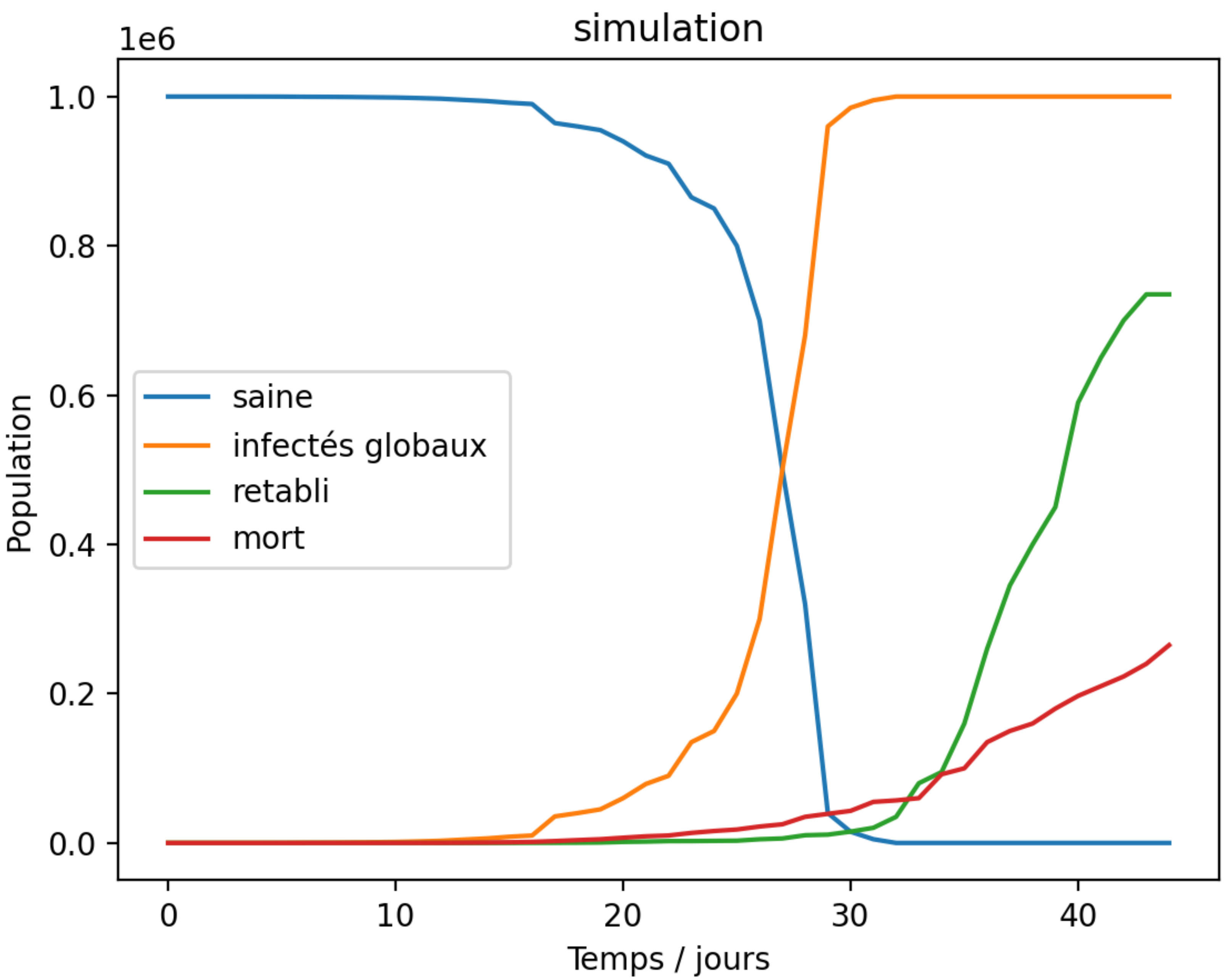
lieux:

- H : quartier résidentiel
- R : rue
- T : quartier d'affaire
- C : centre commercial
- Cim : Cimetière

Simulation d’une ville

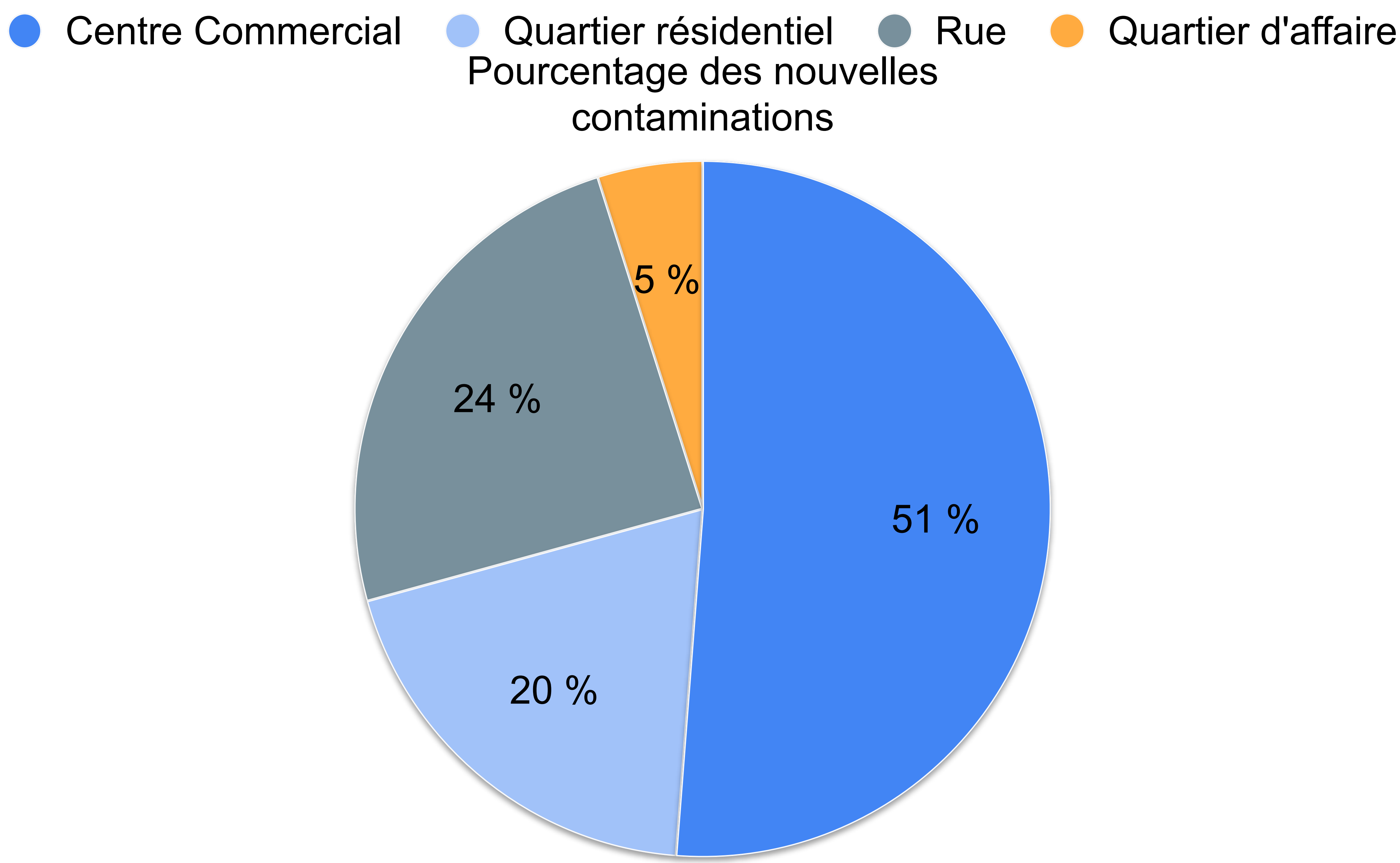


Simulation d'une ville



- population initiale : 1000000
- guérison : 0.63
- mortalité : 0.3
- infecté initiaux : 1

Simulation d’une ville



```
# etats:
#     1=sain
#     2=infecte j0
#     3=infecte j1
#     4=infecte j2
#     5=infecte j3
#     6=infecte j4
#     7=infecte j5
#     8=infecte j6
#     9=infecte j7
#     10=retabli
#     11=mort
# les infectés déclarent des symptômes à partir du 3eme jour donc confiné dans le quartier résidentiel
# lieux:
#     1=quartier résidentiel H propagation par matrices d'adjacence
#     2=rue R propagation de proches en proches
#     3=quartier d'affaire T propagation par knn
#     4=centre commercial C propagation en modèle compartimental
# moment de la journée:
#     1=matinée
#     2=trajet 1
#     3=début de journée
#     4=fin de journée
#     5=trajet 2
#     6=soir
```

```python
import matplotlib.pyplot as plt
from matplotlib.colors import ListedColormap, BoundaryNorm
import copy
import random as rd
import math
import numpy as np
from scipy.integrate import odeint


class Individu:
    def __init__ (self,identifiant, etat, voisins):
        self.identifiant = identifiant
        self.etat = etat
        self.voisins = []

    def position(self, lieu):
        self.lieu = lieu

    def ajouter_voisin(self, voisin):
        self.voisins.append(voisin)
```

```python
class Population:
    def __init__(self, n):
        self.individus = []
        for i in range(1,n+1):
            individu = Individu(i, 0)
            self.individus.append(individu)

    def __iter__(self):
        return iter(self.individus)

    def repartition(self, H, R, T, C, Cim, J):
        for individu in self.individus:
            if individu.etat in [5, 6, 7, 8, 9]:
                H.ajouter_individu(individu)
            elif individu.etat == 11:
                Cim.ajouter_individu(individu)
            else :
                if J.moment_journee in [1, 6]:
                    H.ajouter_individu(individu)
                elif J.moment_journee in [2, 5]:
                    R.ajouter_individu(individu)
                else:
                    r = rd.randint(3, 4)
                    if r == 3:
                        T.ajouter_individu(individu)
                    else:
                        C.ajouter_individu(individu)
```

```python
def compte(self):
    occurences = {
        1: 0,
        2: 0,
        3: 0,
        4: 0,
        5: 0,
        6: 0,
        7: 0,
        8: 0,
        9: 0,
        10: 0,
        11: 0
    }
    for individu in self.individus:
        if individu.etat in occurences:
            occurences[individu.etat] += 1
    return occurences

def ajouter_infecte(self,n):
    for i in range(n):
        r = rd.randint(0, len(self.individus)-1)
        self.individus[r].etat = 2
```

```python
class Jour:
    def __init__(self, numero_jour, moment_journee):
        self.numero_jour = numero_jour
        self.moment_journee = moment_journee

    def avancer_temps(self):
        if self.moment_journee < 6:
            self.moment_journee += 1
        else:
            self.moment_journee = 1
            self.numero_jour += 1
```

```python
class Lieu:
    def __init__(self, nom):
        self.nom = nom
        self.individus = []

    def ajouter_individu(self, individu):
        self.individus.append(individu)

    def get_individu(self,identifiant):
        for individu in self.individus:
            if individu.identifiant == identifiant:
                return(individu)
        return None

    def get_population(self):
        L=[]
        for individu in self.individus:
           L.append(individu.identifiant)
        return(L)
```

```python
class Quartier_résidentiel(Lieu):
    def __init__(self, nom):
        super().__init__(nom)

    def attribuer_voisins(self, v):
        n = len(self.individus)
        for i in range(n):
            individu = self.individus[i]
            voisins = rd.sample(self.individus[:i] + self.individus[i+1:], v)
            individu.voisins = voisins

    def retirer_voisins(self):
        for individu in self.individus:
            individu.voisins = []
```

```python
def propagation(self, v, pr, pm):
    Quartier_résidentiel.attribuer_voisins(self, v)
    for individu in self.individus:
        if individu.etat == 1:
            for voisin in individu.voisins:
                if voisin.etat in [2, 3, 4, 5, 6, 7, 8, 9]:
                    individu.etat = 2
                    break
        elif individu.etat in [2, 3, 4, 5, 6, 7, 8]:
            if bernoulli(pm/2) == 1:
                individu.etat = 11
            else:
                individu.etat += 1
        elif individu.etat == 9:
            if bernoulli(pr) == 1:
                individu.etat = 10
            elif bernoulli(pm) == 1:
                individu.etat = 11
    Quartier_résidentiel.retirer_voisins(self)
```

```
class Rue(Lieu):
    def __init__(self, nom):
        super().__init__(nom)

    def propagation(self, pr, pm):
        G = liste_vers_matrice(self.individus, math.floor(math.sqrt(4*len(self.individus))))
        n = G.taille
        for i in range(n):
            for j in range(n):
                if Grille.get_etat(G, i, j) == 1 and est_exposee(G, i, j):
                    Grille.gete(G, i, j).etat = 2
                elif Grille.get_etat(G,i,j) in [2, 3, 4 ,5, 6, 7, 8, 9]:
                    if bernoulli(pm/2) == 1:
                        Grille.gete(G, i, j).etat = 11
                    else:
                        Grille.gete(G, i, j).etat = Grille.gete(G, i, j).etat +1
```

```python
class Quartier_d_affaire(Lieu):
    def __init__(self, nom):
        super().__init__(nom)

    def propagation_epidemie(self, k, pr, pm):
        M=liste_vers_matrice(self.individus, math.floor(math.sqrt(4*len(self.individus))))

        for i in range(M.taille):
            for j in range(M.taille):
                if Grille.get_etat(M, i, j) in [2, 3, 4, 5, 6, 7, 8]:  # Cellule infectée
                    if bernoulli(pm/2) == 1:  # Probabilité de mourir
                        Grille.gete(M, i, j).etat = 11  # La cellule meurt
                    else :
                        Grille.gete(M, i, j).etat = Grille.gete(M, i, j).etat + 1
                if Grille.get_etat(M, i, j) == 9:
                    if bernoulli(pr) == 1:
                        Grille.gete(M, i, j).etat = 10
                    else:
                        Grille.gete(M, i, j).etat = 11

        for i in range(M.taille):
            for j in range(M.taille):
                if Grille.get_etat(M, i, j) == 1:  # Cellule saine
                    voisins = liste_voisins(M, (i, j), k)
                    e=etiquette_maj(M, voisins)
                    if e in [2,3,4,5,6,7,8,9]:  # La majorité des voisins sont infectés
                        Grille.gete(M, i, j).etat = 2  # La cellule devient infectée
```

```
class Centre_commercial(Lieu):
    def __init__(self, nom):
        super().__init__(nom)

    def propagation(self, alpha, beta, gamma, tmax):
        S, I, R, M, T = solve_SIRM(alpha, beta, gamma, len(self.individus), tmax)
        mI = moyenne_liste(I)
        mM = moyenne_liste(M)
        mR = moyenne_liste(R)
        rd.shuffle(self.individus)
        for individu in self.individus:
            if individu.etat in [2,3,4,5,6,7,8,9]:
                propagation = True
                break
        if propagation == True:
            for individu in self.individus[:mI]:
                if individu.etat == 1:
                    individu.etat = 2
                elif individu.etat in [2,3,4,5,6,7,8]:
                    individu.etat += 1
            for i in range(mM):
                continuer = True
                while continuer:
                    r = rd.randint(0, len(self.individus) -1)
                    if self.individus[r].etat in [2,3,4,5,6,7,8,9]:
                        self.individus[r].etat = 11
            for i in range(mR):
                continuer = True
                while continuer:
                    r = rd.randint(0, len(self.individus) -1)
                    if self.individus[r].etat in [2,3,4,5,6,7,8,9]:
                        self.individus[r].etat = 10
```

```python
class Grille:
    def __init__(self, taille):
        self.taille = taille
        self.matrice = [[0] * taille for _ in range(taille)]

    def est_individu(self, i, j):
        return isinstance(self.matrice[i][j], Individu)

    def gete(self, i, j):
        return self.matrice[i][j]

    def sete(self, i, j, valeur):
        self.matrice[i][j] = valeur

    def est_case_vide(self, x, y):
        return self.matrice[x][y] == 0

    def get_etat(self, i, j):
        if i < 0 or i >= self.taille or j < 0 or j >= self.taille:
            return None
        if self.matrice[i][j] == 0:
            return None
        return self.matrice[i][j].etat
```

```python
def mouvement(self):
    for i in range(len(self.taille)):
        for j in range(len(self.taille)):
            if not Grille.gete(self, i, j) == 0:
                dx, dy = rd.randint(-1, 1), rd.randint(-1, 1)
                if i + dy <= self.taille - 1 and i + dy >= 0 and Grille.gete(self, i + dy, j) == 0:
                    Grille.sete(self, i + dy, j, Grille.gete(self, i, j))
                    Grille.sete(self, i, j, 0)
                    if j + dx <= len(self.matrice) - 1 and j + dx >= 0 and Grille.gete(self, i + dy, j + dx) == 0:
                        Grille.sete(self, i + dy, j + dx, Grille.gete(self, i + dy, j))
                        Grille.sete(self, i + dy, j, 0)
                elif j + dx <= len(self.matrice) - 1 and j + dx >= 0 and Grille.gete(self, i, j + dx) == 0:
                    Grille.sete(self, i, j + dx, Grille.gete(self, i, j))
                    Grille.sete(self, i, j, 0)
                    if i + dy <= len(self.matrice) - 1 and i + dy >= 0 and Grille.gete(self, i + dy, j + dx) == 0:
                        Grille.sete(self, i + dy, j + dx, Grille.gete(self, i, j + dx))
                        Grille.sete(self, i, j + dx, 0)
    return self.matrice
```

```python
def compte(self):
    occurences = {
        1: 0,
        2: 0,
        3: 0,
        4: 0,
        5: 0,
        6: 0,
        7: 0,
        8: 0,
        9: 0,
        10: 0,
        11: 0
    }
    for row in self.matrice:
        for cell in row:
            if cell in occurences:
                occurences[cell] += 1
    return occurences
```

```
def liste_vers_matrice(L, taille):
    G = Grille(taille)
    for individu in L:
        continuer = True
        while continuer:
            x,y=rd.randint(0,taille-1),rd.randint(0,taille-1)
            if G.est_case_vide(x, y):
                G.sete(x ,y, individu)
                continuer = False
    return G

def afficher(M):
    n = M.taille
    M2 = np.zeros((n, n))
    for i in range(n):
        for j in range(n):
            if M.est_individu(i, j):
                e = Grille.get_etat(M, i, j)
                M2[i][j] = e
            elif not M.gete(i, j) == 0:
                M2[i][j] = M.gete(i, j)[-1]

    cmap = ListedColormap(["white", "forestgreen", "lemonchiffon", "yellow", "orange", "darkorange",
                           "orangered", "red", "firebrick", "darkred", "blue", "black"])
    bounds = np.arange(13) - 0.5
    norm = BoundaryNorm(bounds, cmap.N)

    plt.imshow(M2, cmap=cmap, norm=norm)
    plt.colorbar(ticks=np.arange(12), boundaries=bounds)
    plt.show()
```

```python
def bernoulli(pb):
    if rd.random() <= pb:
        return 1
    return 0

def moyenne_liste(L):
    H=[]
    for l in L:
        if l not in H:
            H.append(l)
    return(sum(H)/len(H))
```

```python
##knn
def generer(n):
    return [[0] * n for _ in range(n)]

def distance(p1, p2):
    return abs(p1[0] - p2[0]) + abs(p1[1] - p2[1])

def liste_voisins(M, p, k):
    def f(L):
        return L[0]

    a, b = p
    D = []
    for i in range(M.taille):
        for j in range(M.taille):
            if Grille.est_individu(M, i, j) and not (a,b)==(i,j):
                p2 = (i, j)
                D.append((distance(p, p2), (i, j)))

    return sorted(D, key=f)[:int(k)]
```

```python
def etiquette_maj(M, V):
    D = {}
    for (_, e) in V:
        x, y = e
        if Grille.get_etat(M, x, y) not in D:
            D[Grille.get_etat(M, x, y)] = 1
        else:
            D[Grille.get_etat(M, x, y)] += 1
    e_maj = None
    M_max = 0
    for e in D:
        if D[e] > M_max:
            e_maj = e
            M_max = D[e]
    return e_maj
```

```
##proche en proche
def est_exposee(G, i, j):
    n = G.taille
    if i == j == 0:
        if Grille.get_etat(G, 0, 1) in [2, 3, 4 ,5, 6, 7, 8, 9] or Grille.get_etat(G, 1, 1) in [2, 3, 4 ,5, 6, 7, 8, 9] or Grille.ger_etat(G, 1, 0) in [2, 3, 4 ,5, 6, 7, 8, 9]:
            return True
    elif i == 0 and j == n-1:
        if Grille.get_etat(G, 0, n-2) in [2, 3, 4 ,5, 6, 7, 8, 9] or Grille.get_etat(G, 1, n-2) in [2, 3, 4 ,5, 6, 7, 8, 9] or Grille.get_etat(G, 1, n-1) in [2, 3, 4 ,5, 6, 7, 8, 9]:
            return True
    elif i == n-1 and j == 0:
        if Grille.get_etat(G, n-1, 1) in [2, 3, 4 ,5, 6, 7, 8, 9] or Grille.get_etat(G, n-2, 1) in [2, 3, 4 ,5, 6, 7, 8, 9] or Grille.get_etat(G, n-2, 0) in [2, 3, 4 ,5, 6, 7, 8, 9]:
            return True
    elif i == j == n-1:
        if Grille.get_etat(G, n-1, n-2) in [2, 3, 4 ,5, 6, 7, 8, 9] or Grille.get_etat(G, n-2, n-2) in [2, 3, 4 ,5, 6, 7, 8, 9] or Grille.get_etat(G, n-2, n-1) in [2, 3, 4 ,5, 6, 7, 8, 9]:
            return True
    elif i == 0 and 1 <= j <= n-2:
        if Grille.get_etat(G, 0, j-1) in [2, 3, 4 ,5, 6, 7, 8, 9] or Grille.get_etat(G, 0, j+1) in [2, 3, 4 ,5, 6, 7, 8, 9] or Grille.get_etat(G, 1, j-1) in [2, 3, 4 ,5, 6, 7, 8, 9] or
Grille.get_etat(G, 1, j) in [2, 3, 4 ,5, 6, 7, 8, 9] or Grille.get_etat(G, 1, j+1) in [2, 3, 4 ,5, 6, 7, 8, 9]:
            return True
    elif i == n-1 and 1 <= j <= n-2:
        if Grille.get_etat(G, n-1, j-1) in [2, 3, 4 ,5, 6, 7, 8, 9] or Grille.get_etat(G, n-1, j+1) in [2, 3, 4 ,5, 6, 7, 8, 9] or Grille.get_etat(G, n-2, j-1) in [2, 3, 4 ,5, 6, 7, 8, 9] or
Grille.get_etat(G, n-2, j) in [2, 3, 4 ,5, 6, 7, 8, 9] or Grille.get_etat(G, n-2, j+1) in [2, 3, 4 ,5, 6, 7, 8, 9]:
            return True
    elif j == 0 and 1 <= i <= n-2:
        if Grille.get_etat(G, i-1, 0) in [2, 3, 4 ,5, 6, 7, 8, 9] or Grille.get_etat(G, i+1, 0) in [2, 3, 4 ,5, 6, 7, 8, 9] or Grille.get_etat(G, i-1, 1) in [2, 3, 4 ,5, 6, 7, 8, 9] or
Grille.get_etat(G, i, 1) in [2, 3, 4 ,5, 6, 7, 8, 9] or Grille.get_etat(G, i+1, 1) in [2, 3, 4 ,5, 6, 7, 8, 9]:
            return True
    elif j == n-1 and 1 <= i <= n-2:
        if Grille.get_etat(G, i-1, n-1) in [2, 3, 4 ,5, 6, 7, 8, 9] or Grille.get_etat(G, i+1, n-1) in [2, 3, 4 ,5, 6, 7, 8, 9] or Grille.get_etat(G, i-1, n-2) in [2, 3, 4 ,5, 6, 7, 8, 9] or
Grille.get_etat(G, i, n-2) in [2, 3, 4 ,5, 6, 7, 8, 9] or Grille.get_etat(G, i+1, n-2) in [2, 3, 4 ,5, 6, 7, 8, 9]:
            return True
    else:
        if Grille.get_etat(G, i-1, j-1) in [2, 3, 4 ,5, 6, 7, 8, 9] or Grille.get_etat(G, i-1, j) in [2, 3, 4 ,5, 6, 7, 8, 9] or Grille.get_etat(G, i-1, j+1) in [2, 3, 4 ,5, 6, 7, 8, 9] or
Grille.get_etat(G, i, j-1) in [2, 3, 4 ,5, 6, 7, 8, 9] or Grille.get_etat(G, i, j+1) in [2, 3, 4 ,5, 6, 7, 8, 9] or Grille.get_etat(G, i+1, j-1) in [2, 3, 4 ,5, 6, 7, 8, 9] or
Grille.get_etat(G, i+1, j) in [2, 3, 4 ,5, 6, 7, 8, 9] or Grille.get_etat(G, i+1, j+1) in [2, 3, 4 ,5, 6, 7, 8, 9]:
            return True
    return False
```

```
##Compartiments
def solve_SIRM(alpha, beta, gamma, Nt, tmax):
   x0 = 1
   y0 = 0
   z0 = 0
   w0 = Nt - x0 - y0 - z0

   wlist = [w0]
   xlist = [x0]
   ylist = [y0]
   zlist = [z0]

   def derivative(X, t, alpha, beta, gamma):
      w, x, y, z = X
      wlist.append(w)
      xlist.append(x)
      ylist.append(y)
      zlist.append(z)

      dotw = -alpha * w * x  # Susceptible
      dotx = alpha * w * x - beta * x - gamma * x  # Infecté
      doty = beta * x # Rétabli
      dotz = gamma * y  # Mort

      return np.array([dotw, dotx, doty, dotz])

   t = np.linspace(0, tmax, Nt)

   X0 = [w0, x0, y0, z0]
   res = odeint(derivative, X0, t, args=(alpha, beta, gamma))
   w, x, y, z = res.T

   if len(wlist) != Nt:
      while len(wlist) < Nt:
         wlist.append(wlist[-1])
         xlist.append(xlist[-1])
         ylist.append(ylist[-1])
         zlist.append(zlist[-1])
      while len(wlist) > Nt:
         wlist.pop()
         xlist.pop()
         ylist.pop()
         zlist.pop()

   return wlist, xlist, ylist, zlist, t
```

```
def simulation(n):
    continuer = True
    S = []
    I = []
    R = []
    M = []
    t = 0
    T = [t]
    H = Quartier_résidentiel("Quartier résidentiel")
    R = Rue("Rue")
    T = Quartier_d_affaire("Quartier d'affaires")
    C = Centre_commercial("Centre commercial")
    Cim = Cimetiere("Cimetière")
    J = Jour(0,1)
    population = Population(n)
    while continuer:
        t += 1
        T.append(t)
        population.repartition(H, R, T, C, Cim, J)
        Quartier_résidentiel.propagation(H, 4, 0.60, 0.3)
        Quartier_d_affaire.propagation_epidemie(T, 1, 0.60, 0.3)
        Rue.propagation(R, 0.60, 0.3)
        Centre_commercial.propagation(C, 0.7, 0.60, 0.3, 50)
        Jour.avancer_temps(J)
        D = Population.compte(population)
        infectes = D[2] + D[3] + D[4] + D[5] + D[6] + D[7] + D[8] + D[9]
        I.append(infectes)
        S.append(D[1])
        R.append(D[10])
        M.append(D[11])

        if infectes == 0:
            continuer = False
```

```
plt.close()
plt.plot(T,S)
plt.plot(T,I)
plt.plot(T,R)
plt.plot(T,M)
plt.title("simulation")
plt.xlabel('Temps / jours')
plt.ylabel('Population')
plt.legend(["saine","infectés globaux ","retabli","mort"])
plt.show()
```