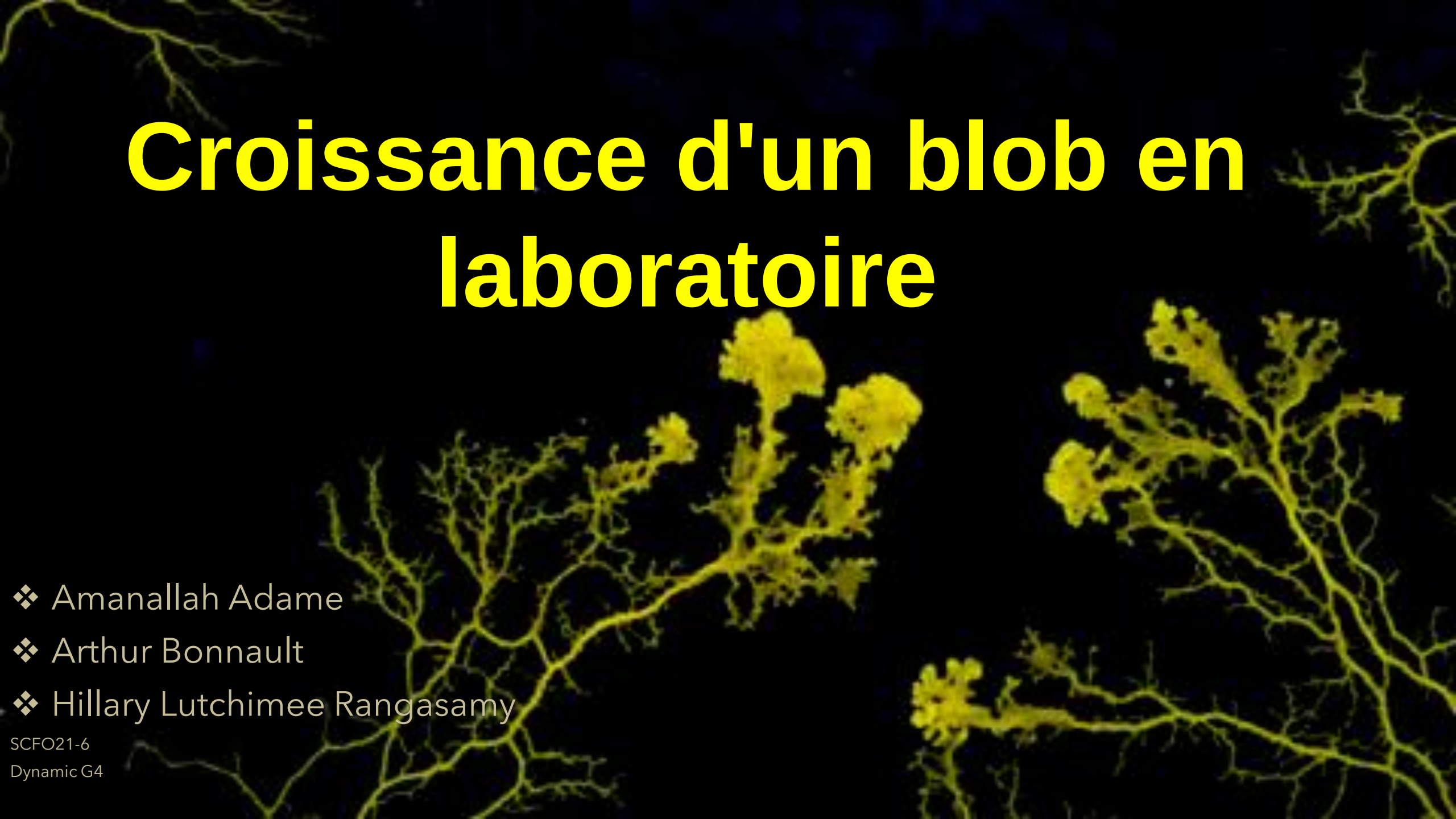


Croissance d'un blob en laboratoire

- Amanallah Adame
- Arthur Bonnault
- Hillary Lutchimee Rangasamy

SCFO21-6

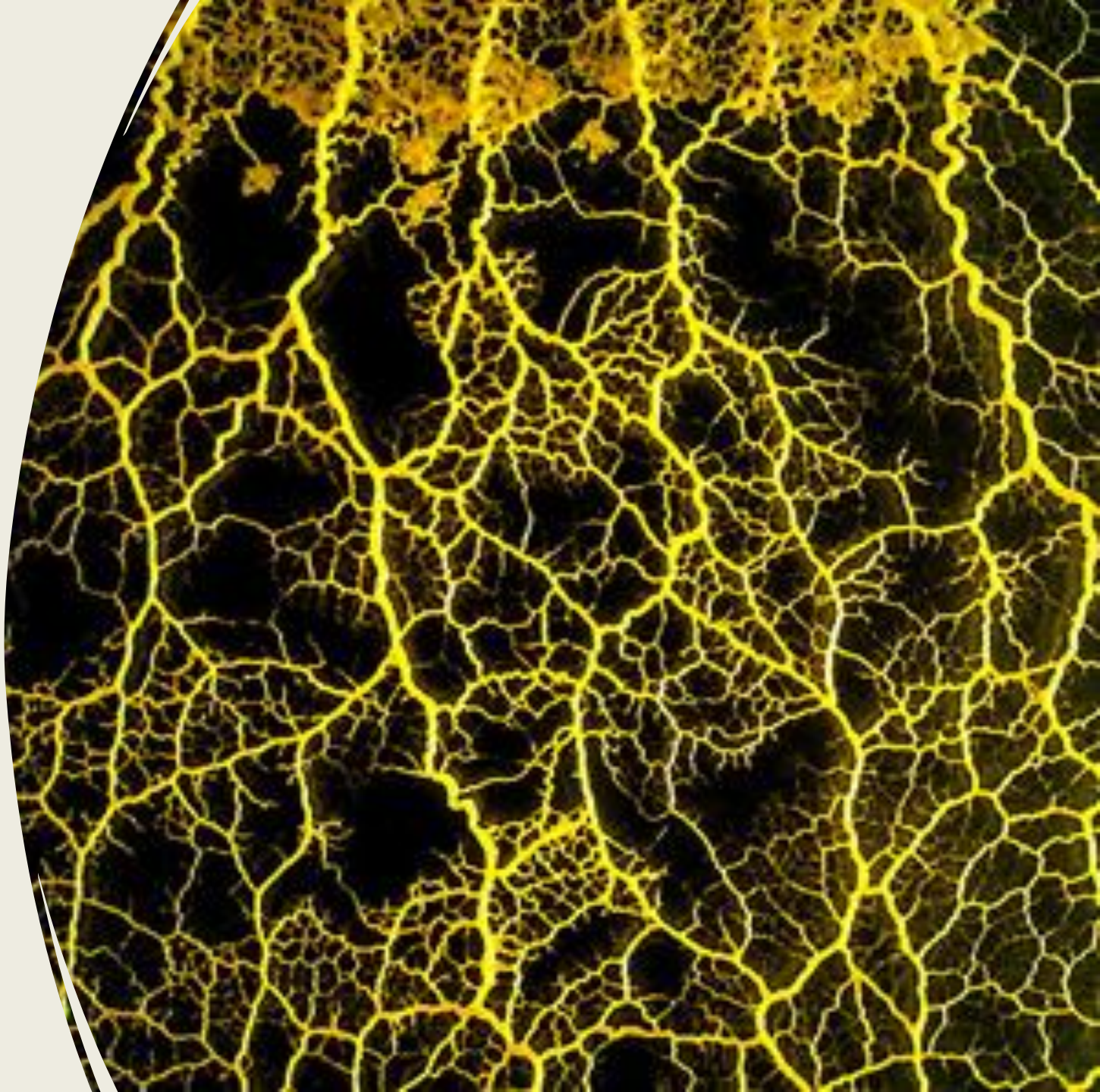
Dynamic G4

Sommaire

- Thématique et problématique
- Modèle
- Simulation
- Résultats et analyse des résultats
- Conclusion et extensions possibles

Thématique

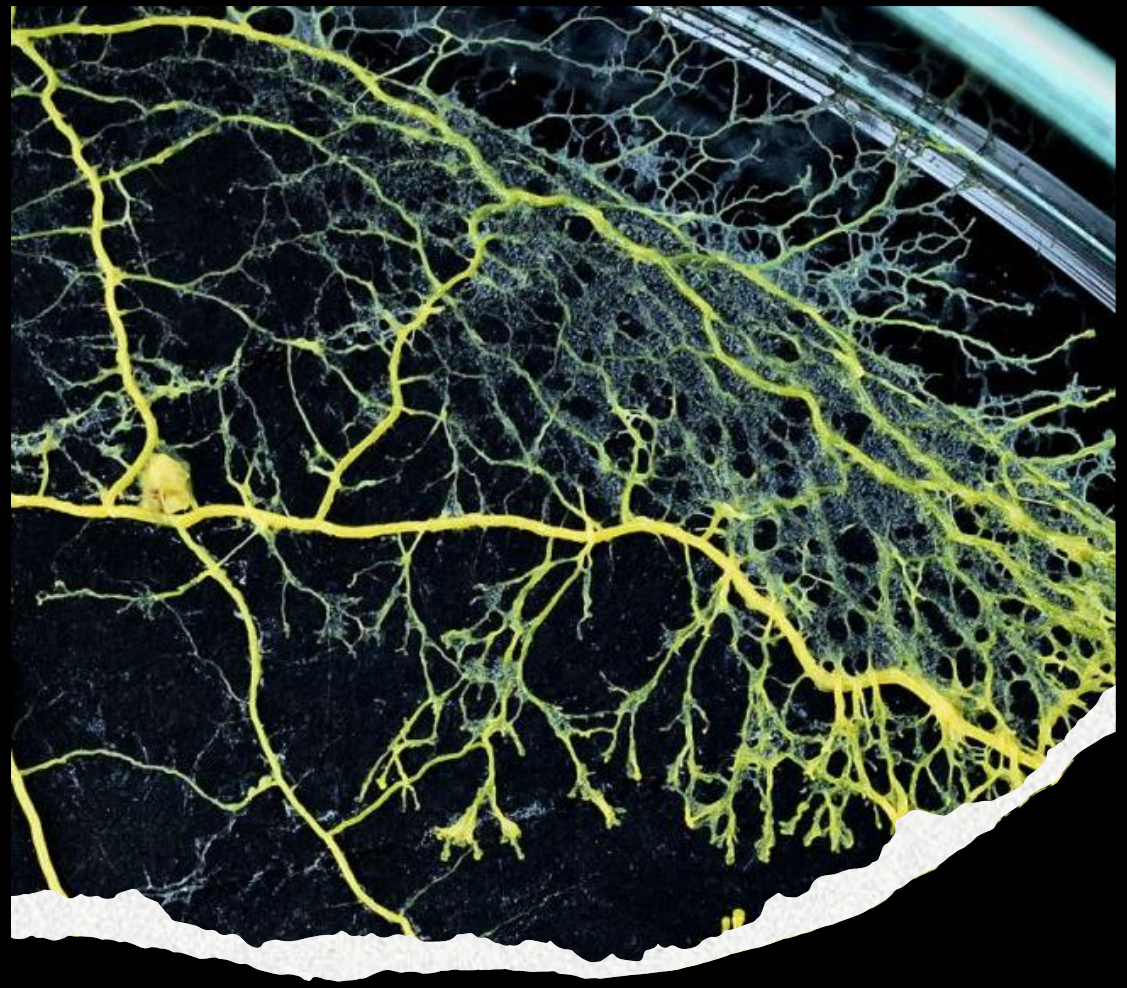
- Physarum polycephalum
- Unicellulaire
- Milieux frais et humides
- Pas de système nerveux
- Peut partager de l'information avec d'autres blobs
- S'étend et crée des réseaux complexes.



Problématique

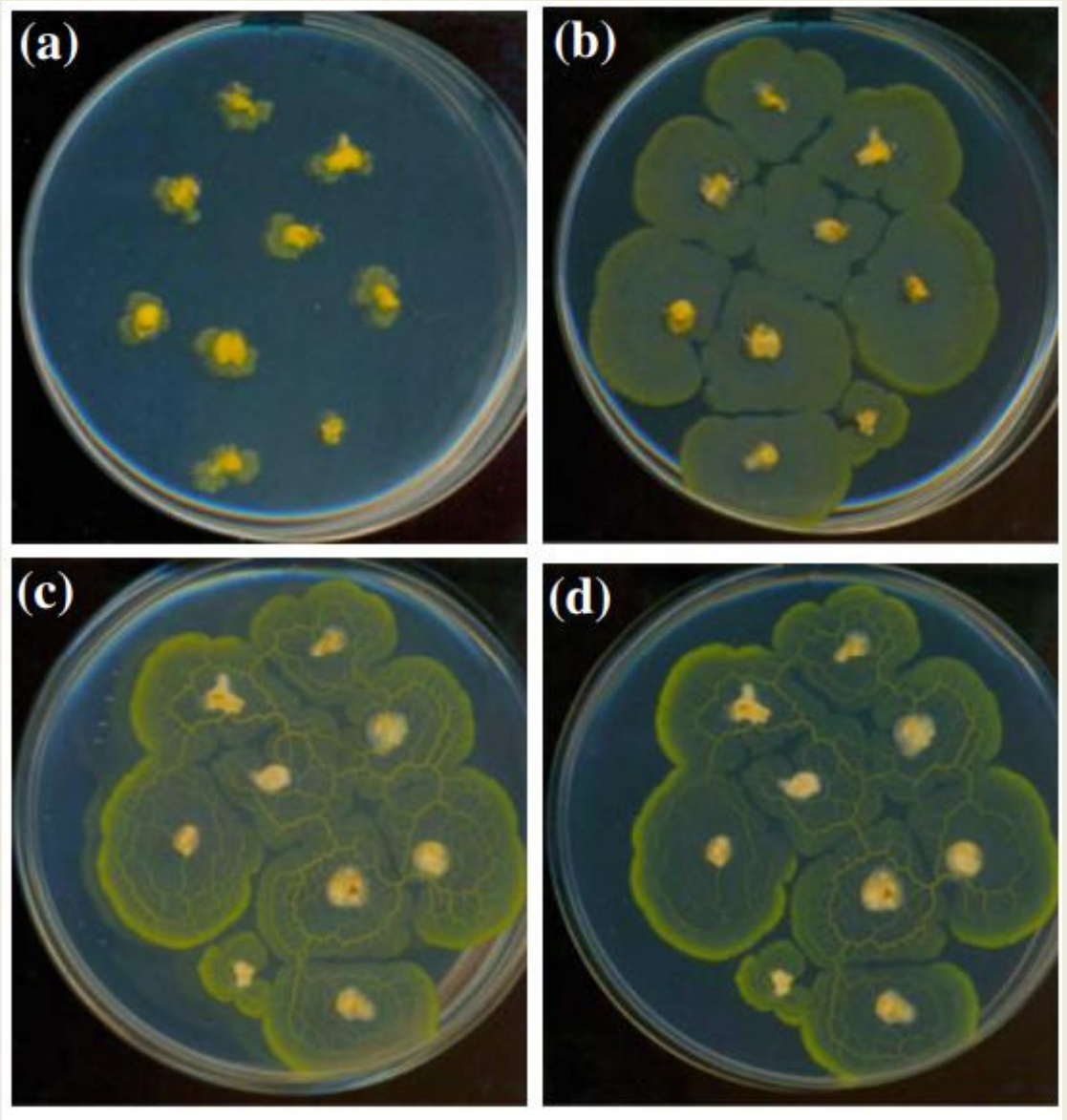
- On se questionnera sur les réseaux du blob.

- Hypothèse : Le blob s’étendra d’abord dans différentes directions puis formera des réseaux qui s’optimiseront dans sa recherche de nourriture.



Modèle utilisé

- On aura pour but de répliquer la seconde variation du modèle d’automate cellulaire de Tsompanas et al.



1. Initialiser le modèle

Les paramètres des équations de diffusion sont définis et la topologie des points de départ et des sources de nourriture est introduit grâce à un tableau depuis une image

```
CON = 0.95  #Le pourcentage de consommation des CHA par le plasmodium
thPM = 0.2  #Masse de slime minimale sur source de nourriture pour faire parti du réseau
PMP1 = 0.08 #parametre diff slime 1
PMP2 = 0.01 #parametre diff slime 2
CAP1 = 0.05 #paramètre diff CHA 1
CAP2 = 0.01 ##paramètre diff CHA 2
```

1. Initialiser le modèle

```python
@staticmethod
def image_to_color_matrix(image_name):
    im = Image.open(image_name)
    matrice = asarray(im)
    return matrice

@staticmethod
def normalise(color_matrix):
    """
    arrayIm de taille (x, y, z=valeur rgb), on veut return une matrice de taille (x, y), première étape
    """
    x, y, _ = color_matrix.shape
    new_array = []

    for i in range(x):
        row = []
        for j in range(y):
            if ((color_matrix[i][j][0] == 0) and (color_matrix[i][j][1] == 0) and (
                    color_matrix[i][j][2] == 0)):  # noir
                row.append(Cell(dispo=0))  # représente un mur
            elif color_matrix[i][j][0] == 255 and color_matrix[i][j][1] == 255 and color_matrix[i][j][2] == 255:  # blanc
                row.append(Cell(dispo=1))  # est un espace libre
            elif color_matrix[i][j][0] == 255 and color_matrix[i][j][1] == 255 and color_matrix[i][j][2] == 0:  # jaune
                row.append(Cell(dispo=1, mass=100, slime=True))  # représente une cellule du slime
            elif color_matrix[i][j][0] == 0 and color_matrix[i][j][1] == 0 and color_matrix[i][j][2] == 255:  # bleu
                row.append(Cell(dispo=1, CHA=100, food=True))  # source de nourriture
        new_array.append(row)
    return new_array
```

2. Appliquer les équations de diffusion

Les équations de diffusion représenteront « l'étalement » du blob et des CHA

$$PMvNN^{t}_{(i,j)} = (1 + PA^{t}_{(i,j),(i-1,j)}) \times PM_{(i-1,j)} - AA_{(i-1,j)} \times PM^{t}_{(i,j)} + (1 + PA^{t}_{(i,j),(i,j-1)}) \times PM_{(i,j-1)} - AA_{(i,j-1)} \times PM^{t}_{(i,j)} + (1 + PA^{t}_{(i,j),(i+1,j)}) \times PM_{(i+1,j)} - AA_{(i+1,j)} \times PM^{t}_{(i,j)} + (1 + PA^{t}_{(i,j),(i,j+1)}) \times PM_{(i,j+1)} - AA_{(i,j+1)} \times PM^{t}_{(i,j)} \quad (12)$$

$$PMeMN^{t}_{(i,j)} = (1 + PA^{t}_{(i,j),(i-1,j-1)}) \times PM_{(i-1,j-1)} - AA_{(i-1,j-1)} \times PM^{t}_{(i,j)} + (1 + PA^{t}_{(i,j),(i+1,j-1)}) \times PM_{(i+1,j-1)} - AA_{(i+1,j-1)} \times PM^{t}_{(i,j)} + (1 + PA^{t}_{(i,j),(i-1,j+1)}) \times PM_{(i-1,j+1)} - AA_{(i-1,j+1)} \times PM^{t}_{(i,j)} + (1 + PA^{t}_{(i,j),(i+1,j+1)}) \times PM_{(i+1,j+1)} - AA_{(i+1,j+1)} \times PM^{t}_{(i,j)} \quad (13)$$

$$PM^{t+1}_{(i,j)} = PM^{t}_{(i,j)} + PMP1 \times \left[PMvNN^{t}_{(i,j)} + PMP2 \times PMeMN^{t}_{(i,j)}\right] \quad (14)$$

$$CHAvNN^{t}_{(i,j)} = (CHA^{t}_{(i-1,j)}) - AA_{(i-1,j)} \times CHA^{t}_{(i,j)} + (CHA^{t}_{(i,j-1)}) - AA_{(i,j-1)} \times CHA^{t}_{(i,j)} + (CHA^{t}_{(i+1,j)}) - AA_{(i+1,j)} \times CHA^{t}_{(i,j)} + (CHA^{t}_{(i,j+1)}) - AA_{(i,j+1)} \times CHA^{t}_{(i,j)}$$

$$CHAeMN^{t}_{(i,j)} = (CHA^{t}_{(i-1,j-1)}) - AA_{(i-1,j-1)} \times CHA^{t}_{(i,j)} + (CHA^{t}_{(i+1,j-1)}) - AA_{(i+1,j-1)} \times CHA^{t}_{(i,j)} + (CHA^{t}_{(i-1,j+1)}) - AA_{(i-1,j+1)} \times CHA^{t}_{(i,j)} + (CHA^{t}_{(i+1,j+1)}) - AA_{(i+1,j+1)} \times CHA^{t}_{(i,j)}$$

$$CHA^{t+1}_{(i,j)} = CON \times \{CHA^{t}_{(i,j)} + CAP1 \times (CHAvNN^{t}_{(i,j)} + CAP2 \times CHAeMN^{t}_{(i,j)})\}$$

3. Création de tubes

Si une cellule de nourriture est encapsulée par le blob (le taux de masse de blob minimum sur sa cellule est atteint), on peut créer un tube entre cette cellule de nourriture et le point de départ

```
self.grid[i][j].tube = True
```

4. Arrivée à la 5000ème itération

Redéfinir les cellules "d'intérêt" (points de départ et sources de nourriture) en tant que sources de nourriture, sauf l'avant dernière source de nourriture encapsulée dans les 5000 dernières itérations, qui est transformée en point de départ

Répéter les équations de diffusion et la création de tubes pendant 5000 itérations

On s’arrête à la 10000ème itération ou avant si tous les points ont été connectés

On choisit l’avant dernier point capturé sur la base du fait qu’il est assez éloigné du point de départ

Création de cellules

```python
class Cell:
    #cell[i,j] = (Espacelibre, MasseSlime, MasseFood, DA, Tube (si la cellule fait partie d'un tube ou non))
    def __init__(self, dispo = 0, mass = float(0), CHA = float(0), da = 0, tube = False, food = False, slime = False, count = 0):
        self.dispo = dispo #la cellule est dispo ou non, 0 ou 1
        self.mass = None #masse du slime présente sur la cellule
        self.CHA = None #attractivité de la nourriture sur notre cellule CHemoAttractant en pourcentage
        self.da = da #direction entre 0 et 8, 0 si pas de direction, va vers le voisin avec la plus grande masse de slime
        self.tube = tube #la cellule fait partie du réseau de tube ou non, bool
        self.food = food #si la cellule est un NS "nutrient source"
        self.slime = slime #si la cellule est un SP "starting point"
        self.color = None #code hexa de la couleur associée à la cellule
        self.refresh(mass, CHA)
```

Création de la carte

```python
class Map:
    def __init__(self, image_name, diffSlime1, diffSlime2, diffNour1, diffNour2, step_diff=50):

        self.diffSlime1 = diffSlime1  # parametre 1 de l'équation de diffusion du slime
        self.diffSlime2 = diffSlime2  # parametre 2 de l'équation de diffusion du slime
        self.diffNour1 = diffNour1  # parametre 1 de l'équation de diffusion de la nourriture
        self.diffNour2 = diffNour2  # parametre 2 de l'équation de diffusion de la nourriture
        self.image_name = image_name  # nom du fichier de l'image
        self.color_matrix = self.image_to_color_matrix(self.image_name)
        self.grid = self.normalise(self.color_matrix)
        self.height = len(self.grid)
        self.width = len(self.grid[0])
        self.check_matrix()
        self.t = 0 #itérations t
        self.step_diff = step_diff #le nombre d'itérations
        self.two_last_encapsulated = [] #les deux derniers NS englobées dans le slime, l'avant d
        self.other_encapsulated = [] #les autres valeurs de NS déjà encapsulées
        self.ns_list = self.list_pos_ns() #liste des coordonnees des NS sources de nourriture
        self.sp_list = self.list_pos_sp() #liste des coordonnes desSP starting point
        self.crash_counter = 0
```

Equations de diffusion du Slime

1. Von Neumann

```
def PMvNN(self, i, j):
        """
        Contribution du voisinage de von Neumann pour la masse du slime sur cellule [i][j]
        """
        haut = 0
        gauche = 0
        bas = 0
        droit = 0
        self.calcul_PA(i,j)

        if self.grid[i][j].dispo == 1: #Vérifie juste qu'on est pas sur un mur. Si c'est le cas la contribution doit être nul
            if self.inMap(i-1,j):
                haut   = (1 + PA_NORTH) * self.grid[i-1][j].mass - (self.grid[i-1][j].dispo * self.grid[i][j].mass)
            if self.inMap(i,j-1):
                gauche = (1 + PA_WEST) * self.grid[i][j-1].mass - (self.grid[i][j-1].dispo * self.grid[i][j].mass)
            if self.inMap(i+1,j):
                bas    = (1 + PA_SOUTH) * self.grid[i+1][j].mass - (self.grid[i+1][j].dispo * self.grid[i][j].mass)
            if self.inMap(i,j+1):
                droit  = (1 + PA_EAST) * self.grid[i][j+1].mass - (self.grid[i][j+1].dispo * self.grid[i][j].mass)

            #print("contribution PA von Neumann : ",(haut + gauche + droit + bas))
        total = haut + gauche + droit + bas
        return total
```

2. Voisinage de Moore

```
def PMeMN(self, i, j):
    """
    Contribution du voisinage de Moore pour la masse du slime sur cellule [i][j]
    """
    hautgauche=0
    basgauche=0
    hautdroit=0
    basdroit=0

    if self.grid[i][j].dispo == 1:
        if self.inMap(i-1,j-1):
            hautgauche = (1 + PA_NORTHWEST) * self.grid[i-1][j-1].mass - (self.grid[i-1][j-1].dispo * self.grid[i][j].mass)
        if self.inMap(i+1,j-1):
            basgauche  = (1 + PA_SOUTHWEST) * self.grid[i+1][j-1].mass - (self.grid[i+1][j-1].dispo * self.grid[i][j].mass)
        if self.inMap(i-1,j+1):
            hautdroit  = (1 + PA_NORTHEAST) * self.grid[i-1][j+1].mass - (self.grid[i-1][j+1].dispo * self.grid[i][j].mass)
        if self.inMap(i+1,j+1):
            basdroit   = (1 + PA_SOUTHEAST) * self.grid[i+1][j+1].mass - (self.grid[i+1][j+1].dispo * self.grid[i][j].mass)

        #print("contribution PA moore : ",(hautgauche + hautdroit + basgauche + basdroit))
    total = hautgauche + hautdroit + basgauche + basdroit
    return total
```

```python
def PM(self, i, j):
    """
    masse de slime totale à t+1 sur la case [i][j]
    """
    #print("masse sur la case {} {} = ".format(i,j), self.grid[i][j].mass)
    #print("mass : {}, PMvNN : {}, PMeMN : {}".format(self.grid[i][j].mass,self.PMvNN(i, j),self.PMeMN(i, j)))
    return self.grid[i][j].mass + self.diffSlime1 * (self.PMvNN(i, j) + self.diffSlime2 * self.PMeMN(i, j))
```

Self.grid[i][j]

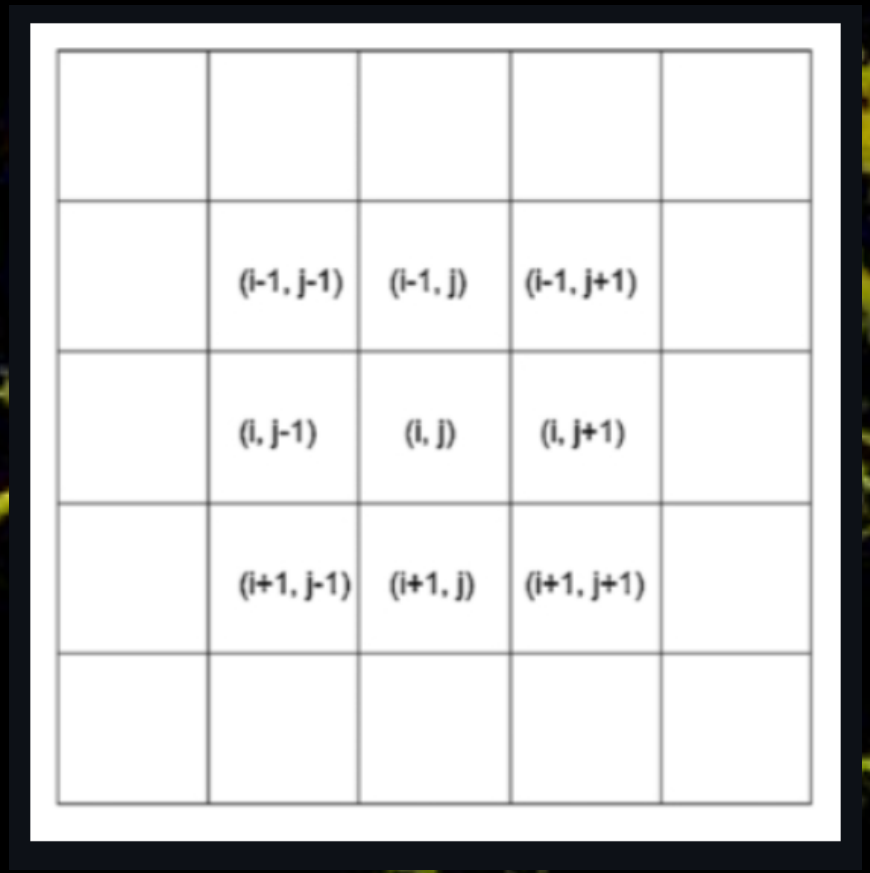
- Masse du slime sur la case [i][j]

Self.diffSlime1 - constante

- PMP1 = 0.08

Self.diffSlime2 - constante

- PMP2 = 0.01



Equations de diffusion de la source de nourriture

```python
def CHAvNN(self, i, j):
    """
    Contribution du voisinage de von Neumann pour la diffusion du chemoattractant sur la case [i][j]
    """
    haut = 0
    gauche = 0
    bas = 0
    droit = 0

    if self.grid[i][j].dispo == 1: #Vérifie juste qu'on est pas sur un mur. Si c'est le cas la contribution doit être nulle
        if self.inMap(i-1,j):
            haut   = self.grid[i-1][j].CHA - (self.grid[i-1][j].dispo * self.grid[i][j].CHA)
        if self.inMap(i,j-1):
            gauche = self.grid[i][j-1].CHA - (self.grid[i][j-1].dispo * self.grid[i][j].CHA)
        if self.inMap(i+1,j):
            bas    = self.grid[i+1][j].CHA - (self.grid[i+1][j].dispo * self.grid[i][j].CHA)
        if self.inMap(i,j+1):
            droit  = self.grid[i][j+1].CHA - (self.grid[i][j+1].dispo * self.grid[i][j].CHA)

    #print("contribution CHA von Neumann : ",(haut + gauche + droit + bas))
    total = haut + gauche + droit + bas
    return total
```

```python
def CHAeMN(self, i, j):
    """
    Contribution du voisinage de Moore pour la diffusion du chemoattractant sur la case [i][j]
    """
    hautgauche = 0
    hautdroit = 0
    basgauche = 0
    basdroit = 0

    if self.grid[i][j].dispo == 1: #Vérifie juste qu'on est pas sur un mur. Si c'est le cas la contribution doit être nulle
        if self.inMap(i-1,j-1):
            hautgauche = (self.grid[i-1][j-1].CHA) - (self.grid[i-1][j-1].dispo * self.grid[i][j].CHA)
        if self.inMap(i+1,j-1):
            hautdroit  = (self.grid[i+1][j-1].CHA) - (self.grid[i+1][j-1].dispo * self.grid[i][j].CHA)
        if self.inMap(i-1,j+1):
            basgauche  = (self.grid[i-1][j+1].CHA) - (self.grid[i-1][j+1].dispo * self.grid[i][j].CHA)
        if self.inMap(i+1,j+1):
            basdroit   = (self.grid[i+1][j+1].CHA) - (self.grid[i+1][j+1].dispo * self.grid[i][j].CHA)

    #print("contribution CHA moore : ",(hautgauche + hautdroit + basgauche + basdroit))
    total = hautgauche + hautdroit + basgauche + basdroit
    return total
```

```python
def CHA(self, i, j):
    """
    Attraction chimique sur la case [i][j] à t+1
    """
    #print("CHA : {}, CHAvNN : {}, CHAeMN : {}".format(self.grid[i][j].CHA,self.CHAvNN(i, j),self.CHAeMN(i, j)))
    return CON * ( self.grid[i][j].CHA + self.diffNour1 * ( self.CHAvNN(i, j) + self.diffNour2 * self.CHAeMN(i, j) ) )
```

CON = 0.95
- Pourcentage de nourriture absorbé par le slime.

Self.grid[i][j]
- Masse de la nourriture sur la case [i][j]

Self.diffNour1 - constante
- CAP1 = 0.05

Self.diffNour1 - constante
- CAP2 = 0.01

Partie algorithmique

```python
def start_loop(self):
    """
    boucle principale de la Map, se charge de faire tous les traitements
    """
    self.is_running.set("Simulation lancée")
    is_end = False
    copy_ns_list = self.map.ns_list
    copy_sp_list = self.map.sp_list
    while not is_end:
        self.iterate_diff() #50 itérations de diffusion
        print("t : ", self.map.t)
        self.time_var.set("Itération : " +str(self.map.t)) #tkinter
        if len(self.map.ns_list) == 0: #s'il n'y a plus de ns, ça veut dire qu'ils sont normalement tous connecté
            print("Fin de la simulation")
            is_end = True   #à modifier
            self.is_running.set("Fin de la simulation") #tkinter
        else:
            for i in (copy_ns_list):
                x=i[0]
                y=i[1]
                #print("i :",i)
                #print("i[0] :",i[0])
                #print("ns_list : ", self.ns_list)
                #print("x : {}, y : {}".format(x,y))
                if self.map.grid[x][y].food and self.map.grid[x][y].mass >= thPM:
                    self.map.connect_tube(x,y)
                    self.map.grid[x][y].ns_to_sp()
                    self.map.ns_list.remove(i) #on se retrouvera normalement avec une liste vide
                    self.map.sp_list.append(i) #on rajoute l'élément courant dans la liste des SP
```

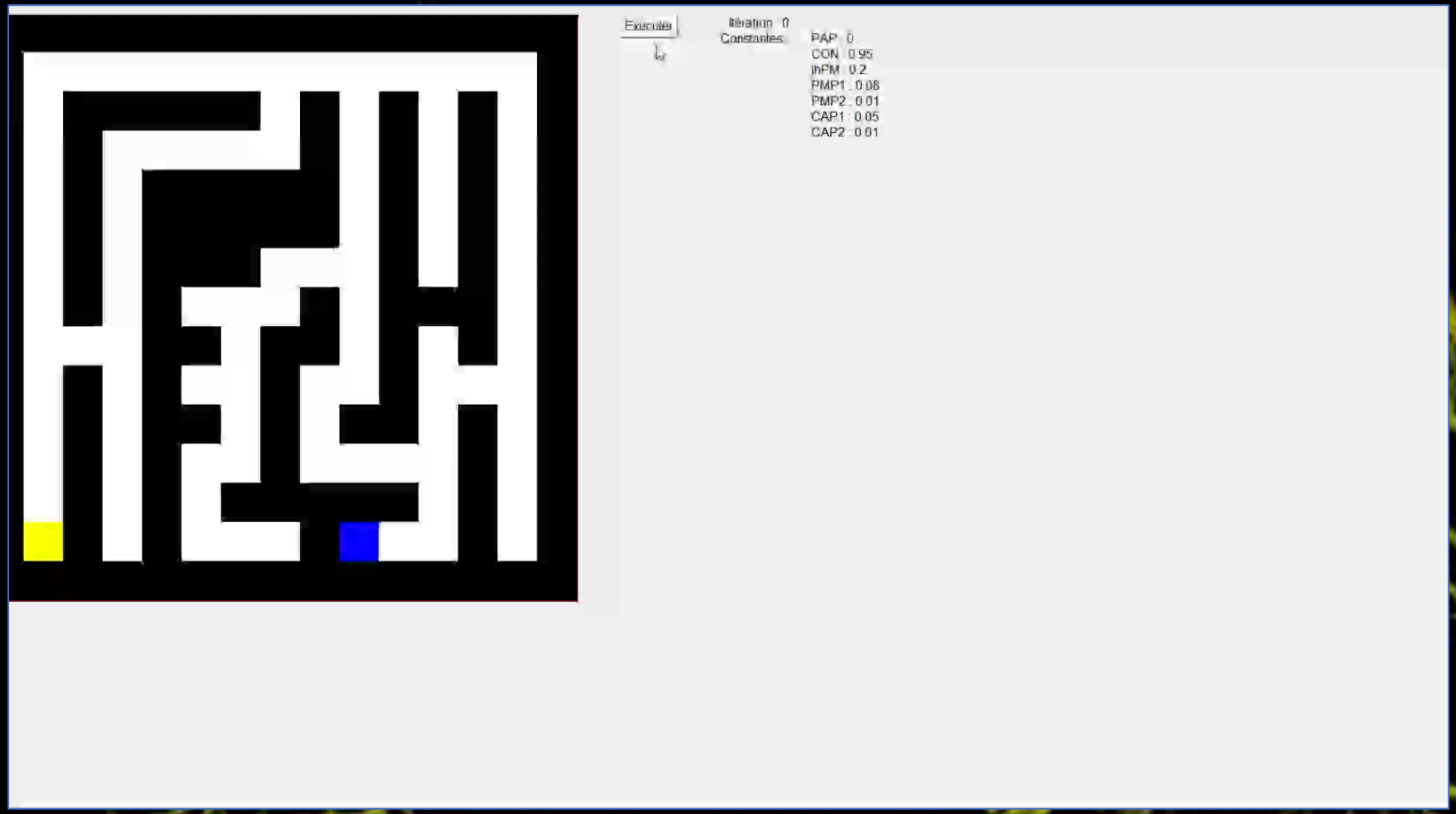
Partie algorithmique

```
        if self.map.t <= 5000:
            if self.map.t == 5000:
                print("Nous sommes à la 5000ème itération, changement des SP en NS")
                if len(self.map.sp_list) == 1:
                    print("Il n'y a qu'un seul SP restant, impossible de le changer en NS")
                else:
                    for i in (copy_sp_list):
                        x=i[0]
                        y=i[1]
                        self.map.grid[x][y].misc_to_ns()
                        self.map.sp_list.remove(i)

                    (abs_avant_dernier, ord_avant_dernier,) = self.map.two_last_encapsulated[0]
                    #changement de l'avant dernier en SP
                    self.map.grid[abs_avant_dernier][ord_avant_dernier].ns_to_sp()
                    print("sp_list :",self.map.sp_list)
                    print("ns_list :",self.map.ns_list)
                    print("abs : {}, ord : {}".format(abs_avant_dernier,ord_avant_dernier))
                    self.map.sp_list.append((abs_avant_dernier,ord_avant_dernier))
        elif self.map.t >=10000:
            is_end = True
```

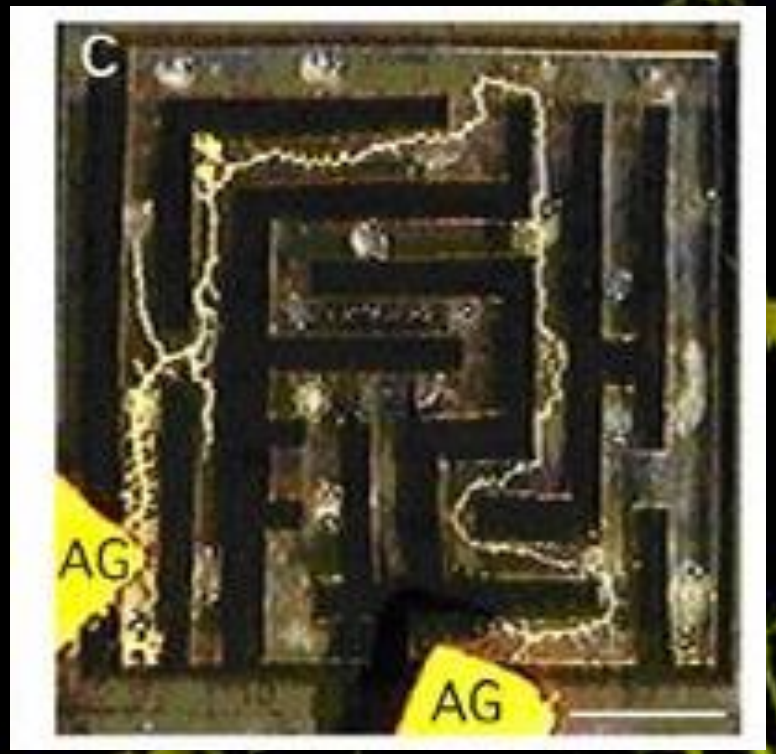
Partie algorithmique

```python
def connect_tube(self, i, j):
    """
    Commence le tube au NS et va jusqu'à un SP, guidé par la masse de slime sur les cellules voisines
    """
    self.grid[i][j].tube = True
    print("tube créé en {},{}".format(i,j))
    if not self.grid[i][j].slime:
        _,x,y = self.highest_neighbour_pm(i,j)
        if self.crash_counter < 500:
            self.crash_counter += 1
            self.connect_tube(x,y)
        else:
            print("Boucle infinie : ", self.crash_counter)

def highest_neighbour_pm(self, i, j):
    x = -1 #valeurs impossible
    y = -1

    highest_pm = float(0)

    for a in range(i - 1, i + 2):
        for b in range(j - 1, j + 2):
            if a == i and b == j:
                continue
            else:
                pm = self.val_pm(a, b)
                if pm > highest_pm:
                    highest_pm = pm
                    x = a
                    y = b

    return highest_pm, x, y
```

Simulation

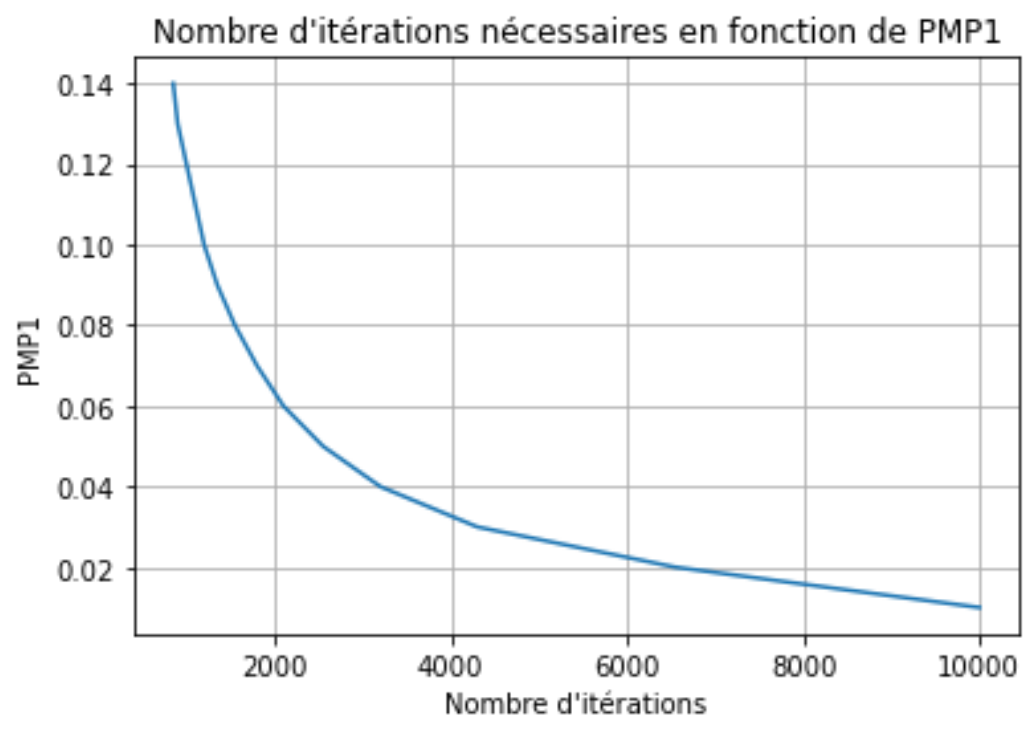

Executer
Itération 0
Constantes
PAP 0
CON 0.95
INPM 0.2
PMP1 0.08
PMP2 0.01
CAP1 0.05
CAP2 0.01

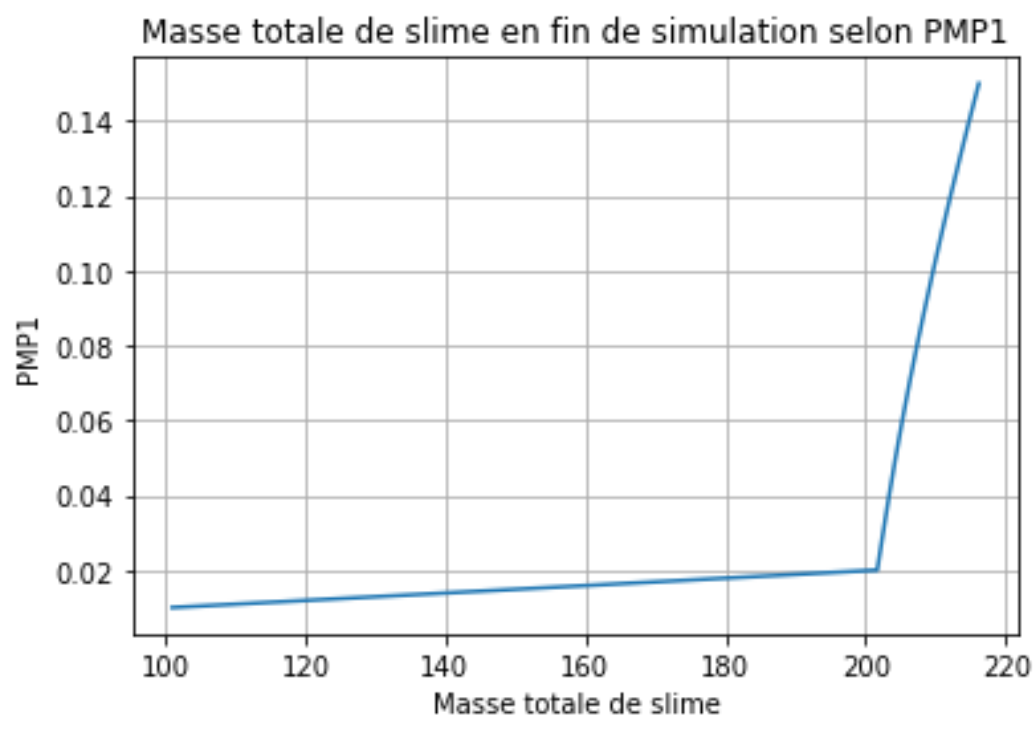
Résultats



Métriques

Changements des paramètres de diffusion





Suites possibles

- Résoudre l’algorithme pour N sources de nourritures
- N sources de Blob
- Implémenter d’autres paramètres comme des obstacles/agents chimiques/conditions environnementales
- Optimiser l’algorithme pour de plus grandes tailles