Étude et modélisation de mécanismes régissant une foule

Colin Coërchon

Candidat n°4271

Session 2021

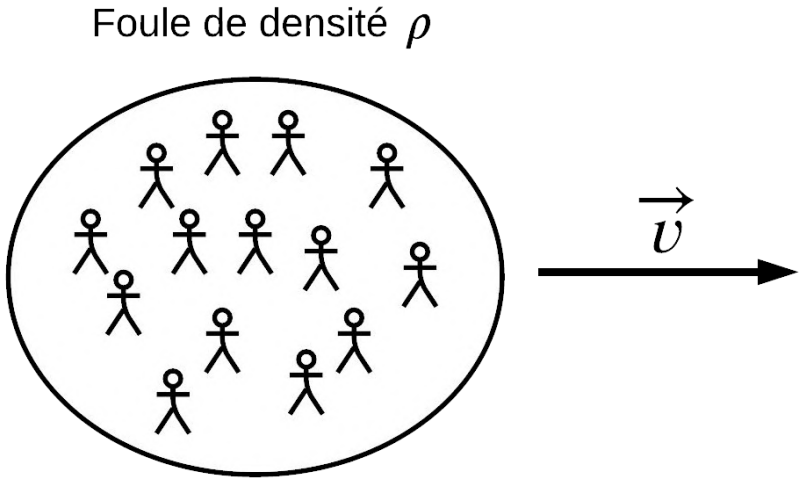
Sommaire

I - Positionnement du problème et objectifs

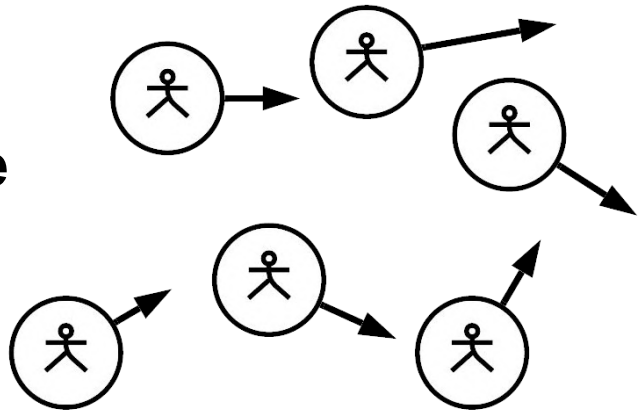
- ➔ De graves incidents assez récurrents.
- ➔ Mécanismes de foule compliqués à appréhender
- ➔ Manque de données de réels mouvements de paniques
- ➔ Nécessité de l'outil informatique

II - Interactions entre individus et précision de l'étude

➔ Modèle **macroscopique**



➔ Modèle **microscopique**



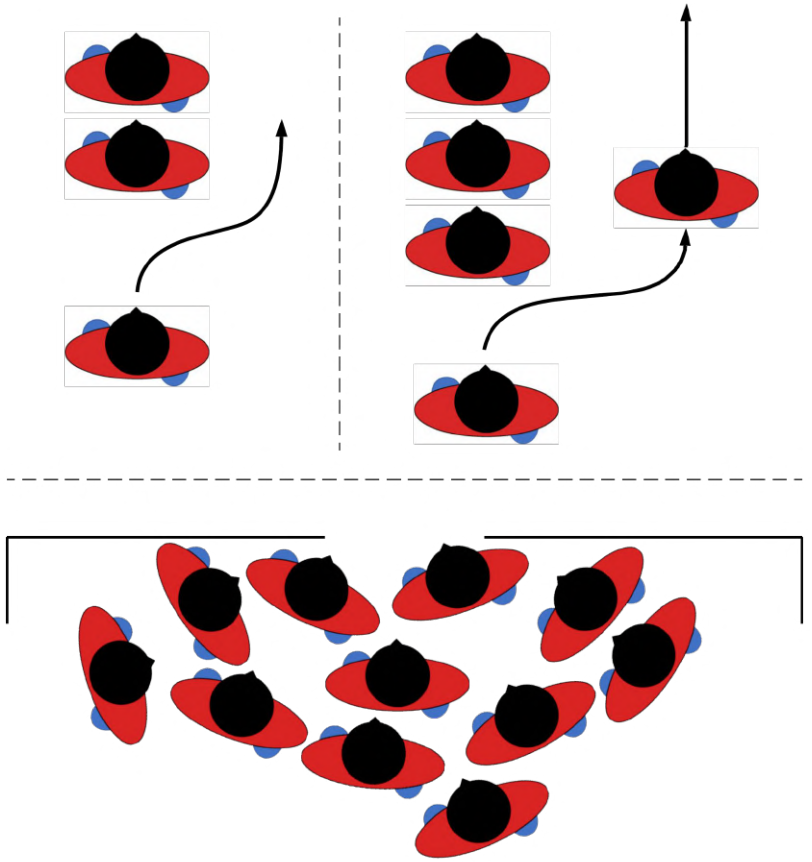
Vitesse + **position** pour chaque individu

1. Interactions locales

Pluralité des interactions entre individus dans une foule. Elles influencent directement leurs déplacements.

- L'**évitement**
- L'**imitation**
- Les **bousculades**

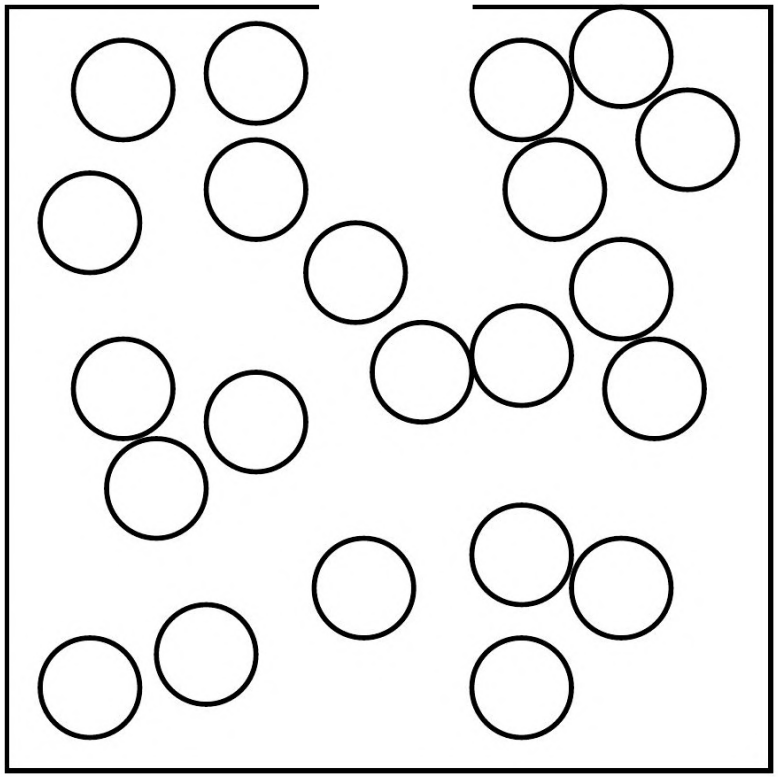
On omettra ici les interactions indirectes (ligne de désir, etc…)



2. Notre étude

- Salle carrée avec une unique porte, accueillant P personnes.
- Un piéton seul est totalement libre de ces mouvements et suivra un déplacement quasi-linéaire à une vitesse v_0 .
- L'essentiel repose alors dans les **interactions entre piétons**.

Salle initialisée



III - Discrétisation et Automates Cellulaires

1. Modèle de Vicsek

Chaque individu est soumis à 2 règles :

1. Conserver une vitesse constante v_0.
2. S'aligner en permanence avec les autres individus dans un rayon proche.

Sa direction est alors donnée par :

$$\theta(t) = \underbrace{\langle \theta(t-1) \rangle_r}_{\text{Direction moyenne de déplacement des individus se trouvant dans un rayon r}} + \underbrace{\varepsilon}_{\text{Terme de fluctuation aléatoire}}$$

Le modèle de Vicsek place l'**imitation** comme interaction principale.

Nécessité de prendre en compte une **répulsion** minimale pour éviter les chevauchements.

→ Importance de la variabilité, intrinsèque au comportement du piéton.

→ Ce modèle s'opère dans le cas d'une évacuation d'urgence.

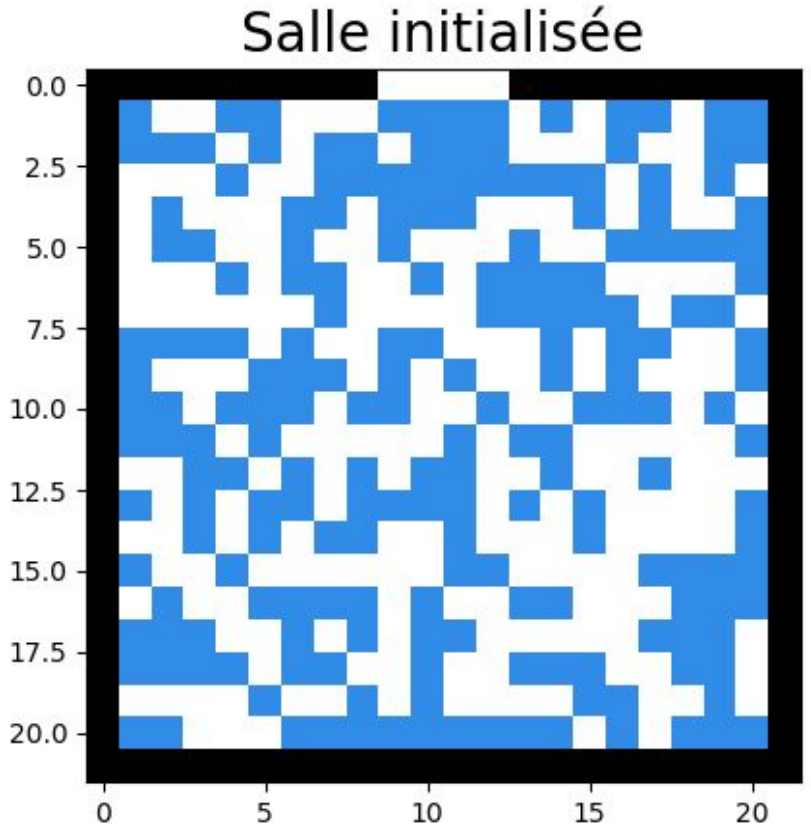
Ce modèle est une première approche de **discrétisation** de l'espace.

2. Par les Automates Cellulaires

- Discrétisation spatiale :
  - Salle de côté N
  - P personnes
- Chaque individu assimilé à un carré de 40x40 cm²

4 déplacements possibles :

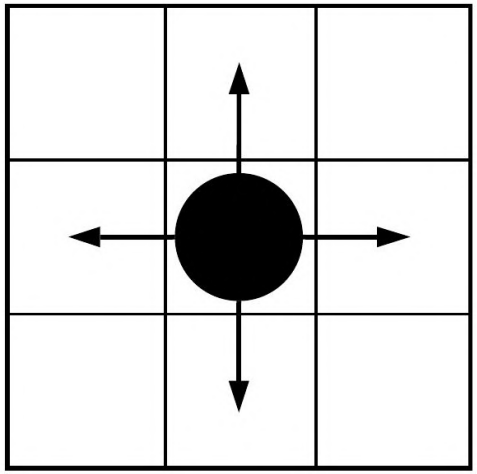
[Haut, Gauche, Droite, Bas]



N=20 et P=200

Probabilités de transitions

Théorie :



$\longrightarrow$ 4 directions possibles

La probabilité est régie selon :

- Un **champ statique S** : dépend de la distance à la sortie
- Un **champ dynamique D** : dépend du temps. À t=0,

$$\forall (i,j) \in [|1;N|]^2,\ D_{ij} = 0$$

	$p_{i-1,j}$	
$p_{i,j-1}$	$p_{i,j}$	$p_{i,j+1}$
	$p_{i+1,j}$	

Ainsi, on définit la probabilité p_{ij} pour aller sur la case (i,j) :

$$p_{ij}(t) = N \exp\left(k_S S_{ij}\right) \exp\left(k_D D_{ij}\right) \delta_{ij}$$

où δ_{ij} vaut 1 si (i,j) est libre, 0 sinon

S_{ij} et D_{ij} les valeurs des deux champs sur (i,j).

k_S et k_D sont >0 et varient pour chaque piéton

N sert de facteur de normalisation

Dans notre modèle

[Haut, Gauche, Droite, Bas]

Si $\alpha = 0,\ p = [0.9\ ;\ 0.05\ ;\ 0.05\ ;\ 0]$

Si $\alpha = +\frac{\pi}{2},\ p = [0.05\ ;\ 0\ ;\ 0.9\ ;\ 0.05]$

Si $\alpha = -\frac{\pi}{2},\ p = [0.05\ ;\ 0.9\ ;\ 0\ ;\ 0.05]$

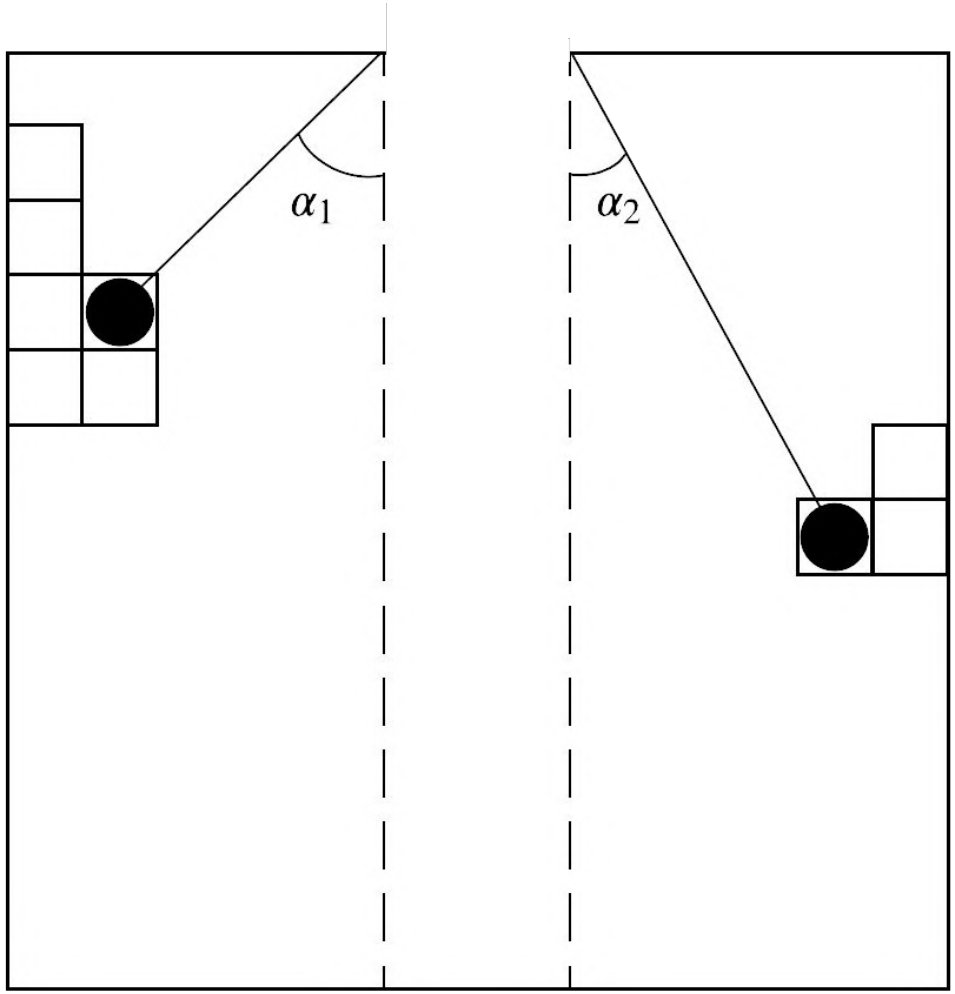
Si $\alpha \in \left[0\ ; \frac{\pi}{4}\right],\ p = [0.9\ ;\ 0\ ;\ 0.1\ ;\ 0]$

Si $\alpha \in \left[-\frac{\pi}{4}; 0\right],\ p = [0.9\ ;\ 0.1\ ;\ 0\ ;\ 0]$

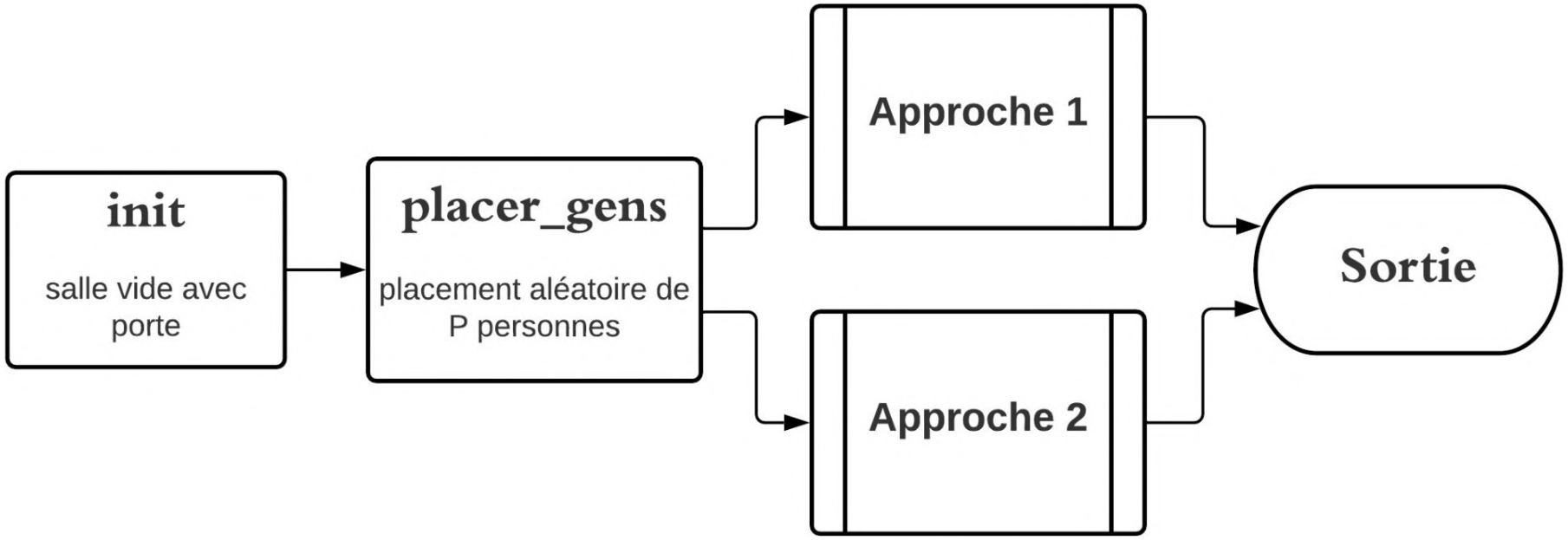
Si $\alpha \in \left[\frac{\pi}{2}, \frac{\pi}{4}\right], p = [0.55\ ;\ 0\ ;\ 0.45\ ;\ 0]$

Si $\alpha \in \left[-\frac{\pi}{4}, -\frac{\pi}{2}\right],\ p = [0.55\ ;\ 0.45\ ;\ 0\ ;\ 0]$

Si $\alpha = 0$ et $i = 1$ ou $2,\ p = [1\ ;\ 0\ ;\ 0\ ;\ 0]$



Principe des deux algorithmes



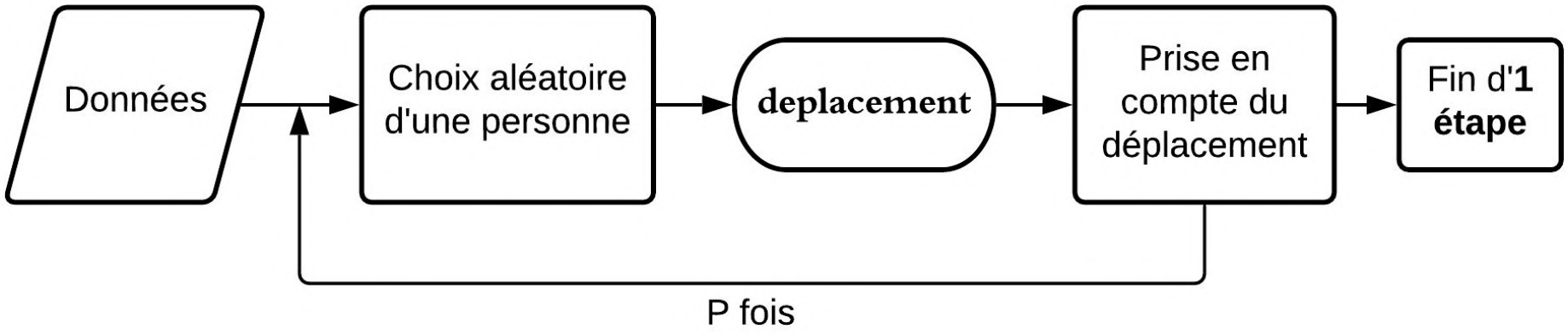
Approche 1

Mise à jour des positions **individu par individu**.

On note **1 étape** comme étant le déplacement unique de toutes les personnes.

⟶ 1 étape correspond donc à un appel de deplacement_total

Schéma d'une étape :



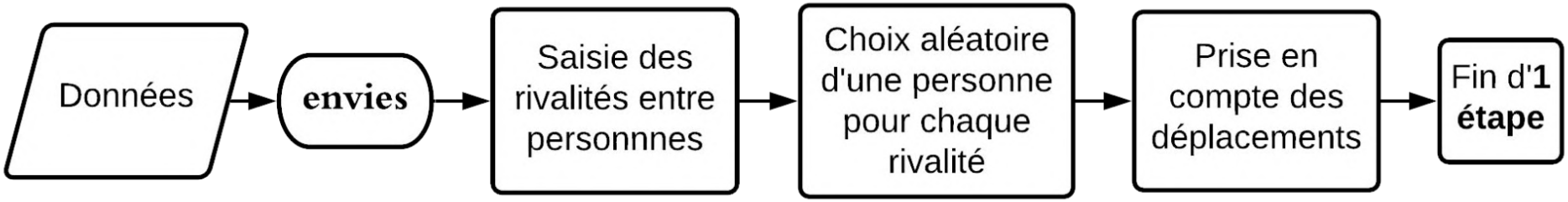
Approche 2

Mise à jour **globale** des positions.

On note **1 étape** comme étant le déplacement unique de toutes les personnes.

⟶ 1 étape correspond donc à un appel de deplacement_global

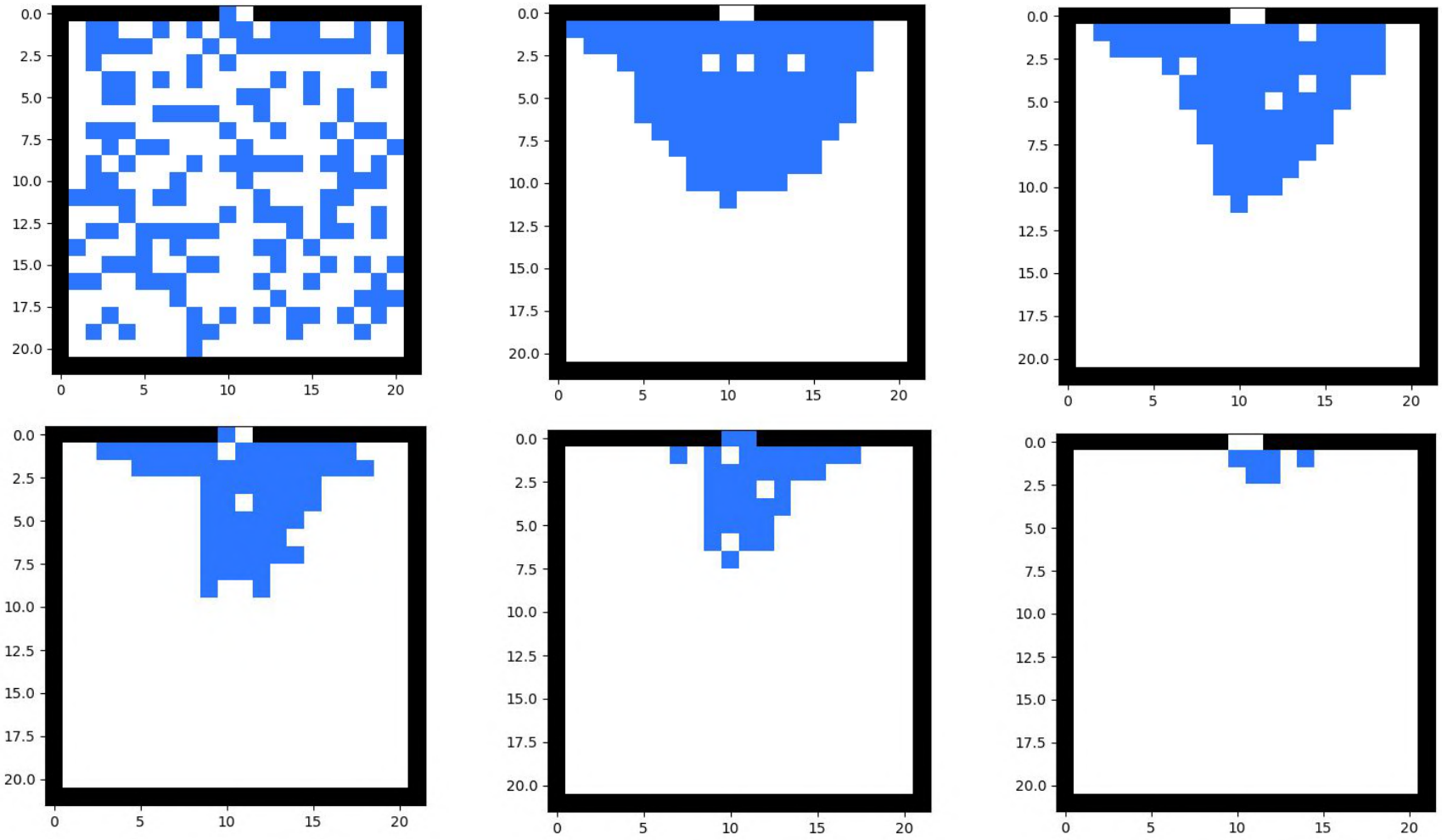
<u>Schéma d'une étape</u> :



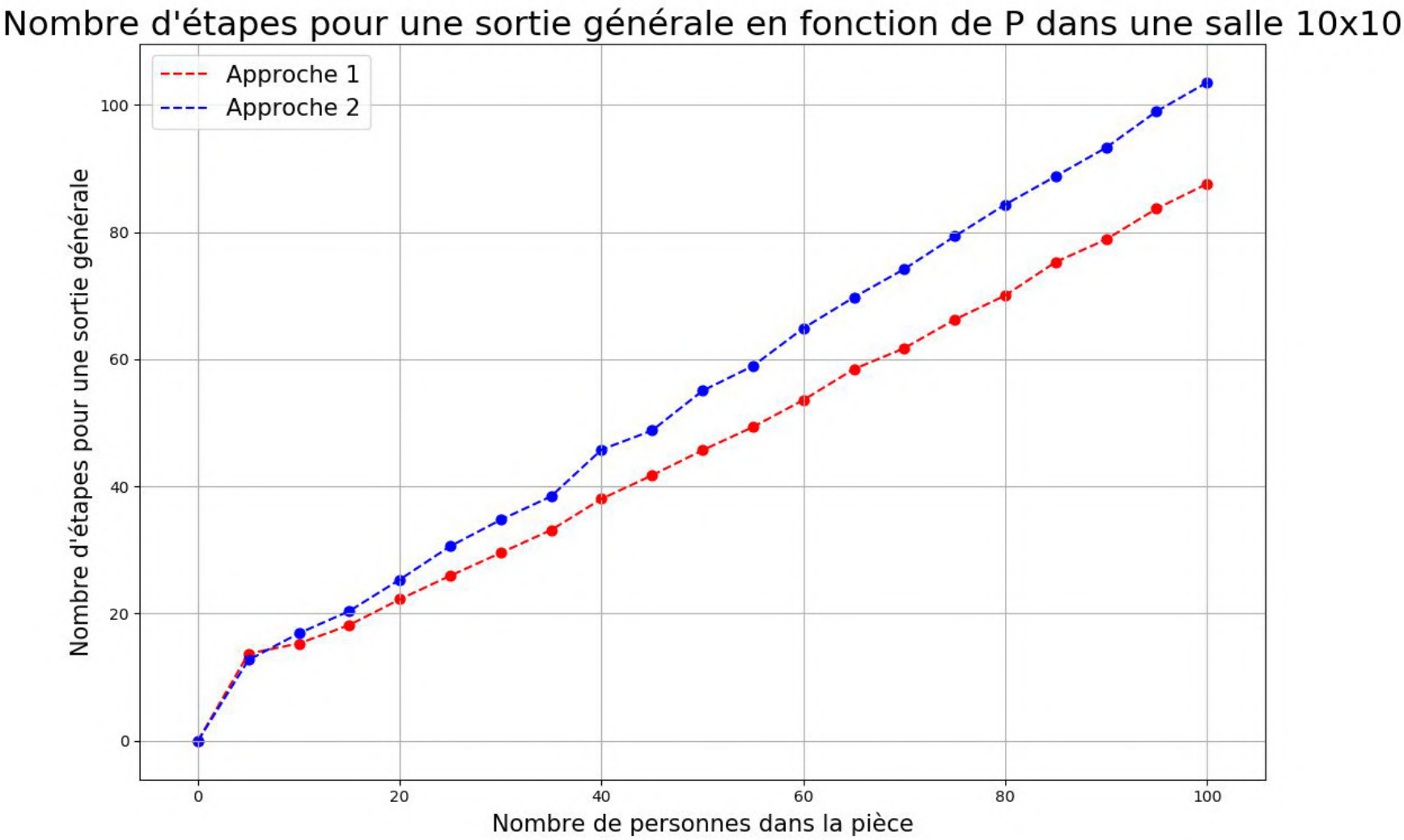
Dans les deux approches, une fonction sortie est définie.

→ Énumération du nombre d'étapes pour une sortie générale.

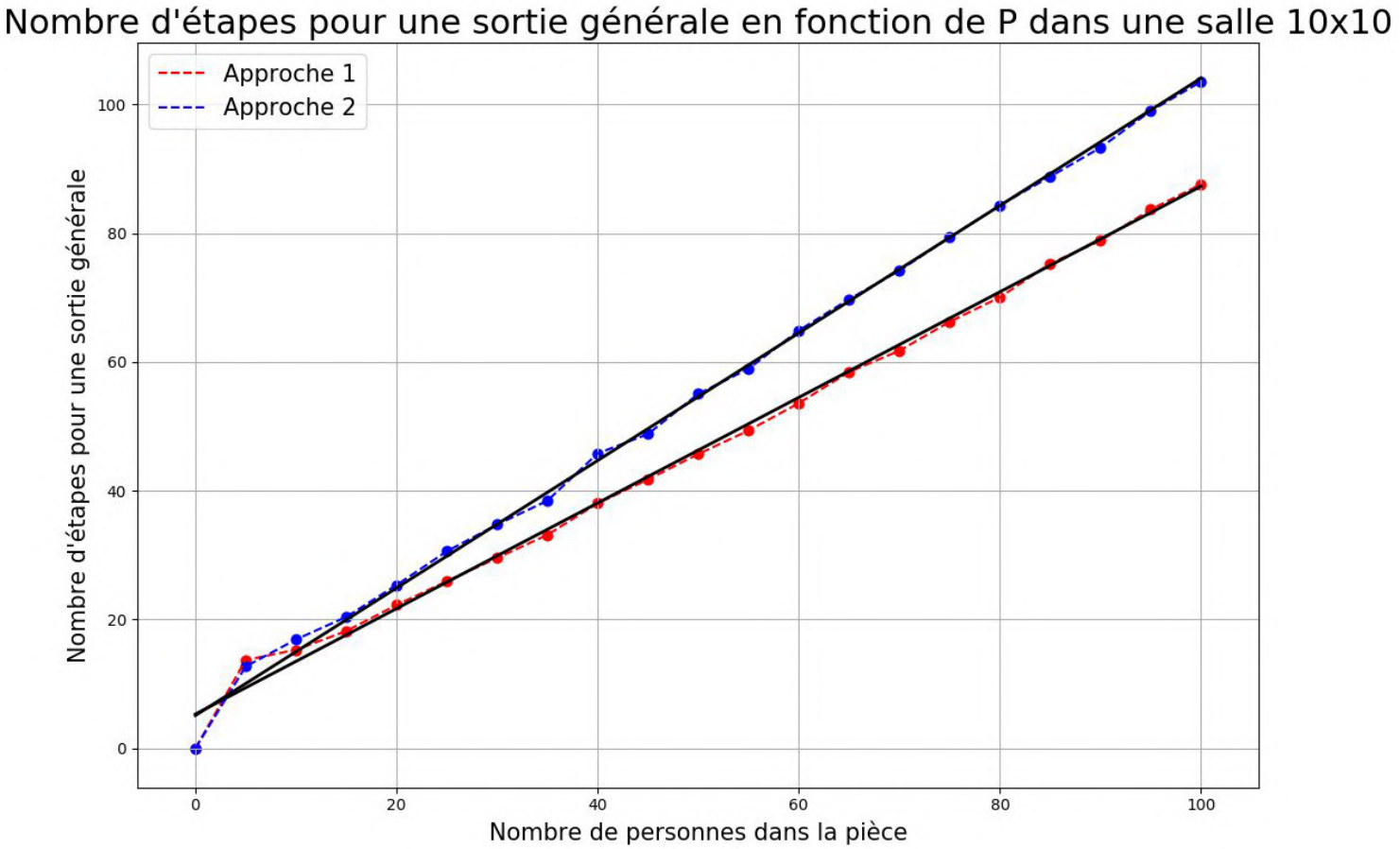
Résultats numériques :



Analyse graphique

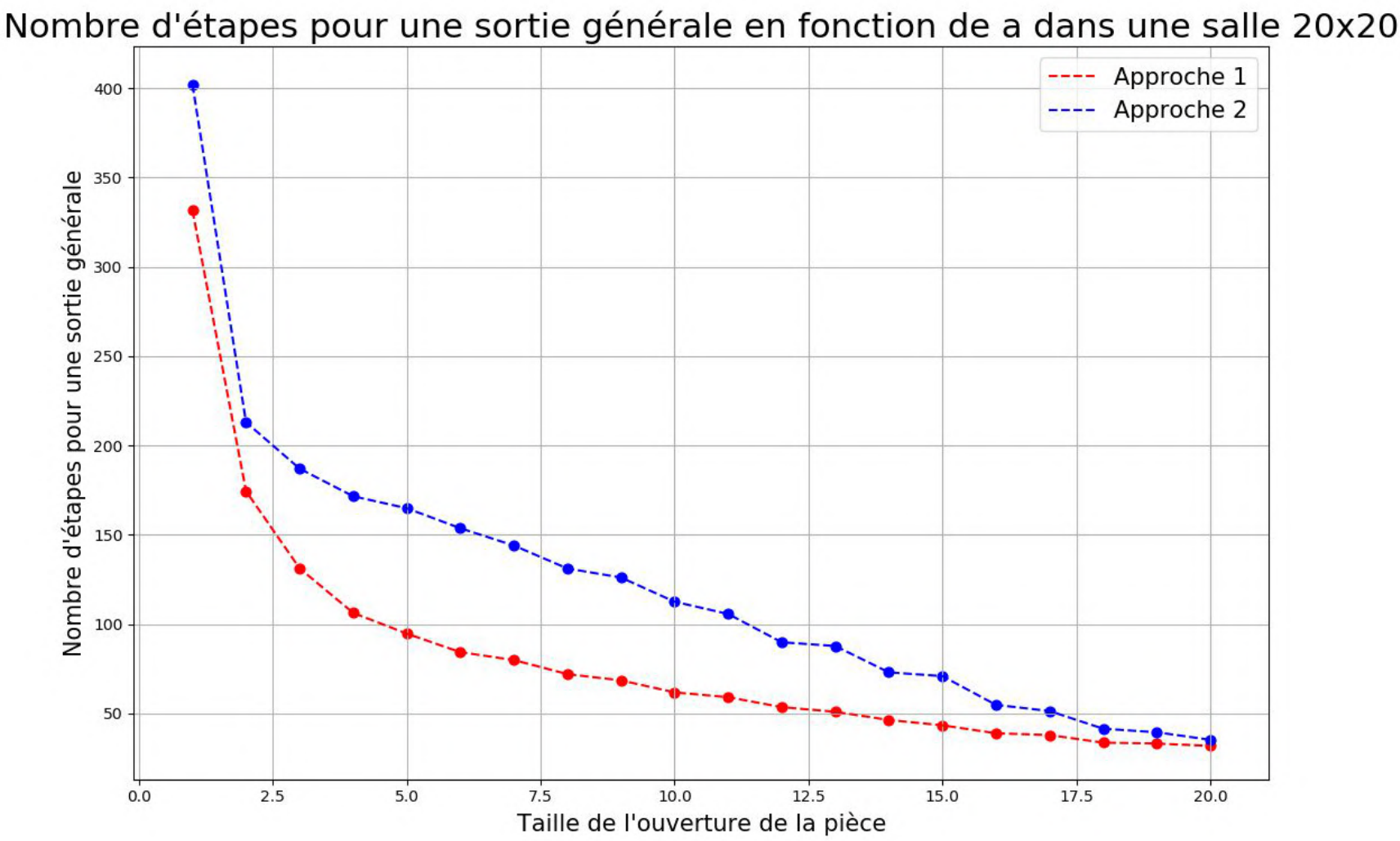


Modélisation des courbes



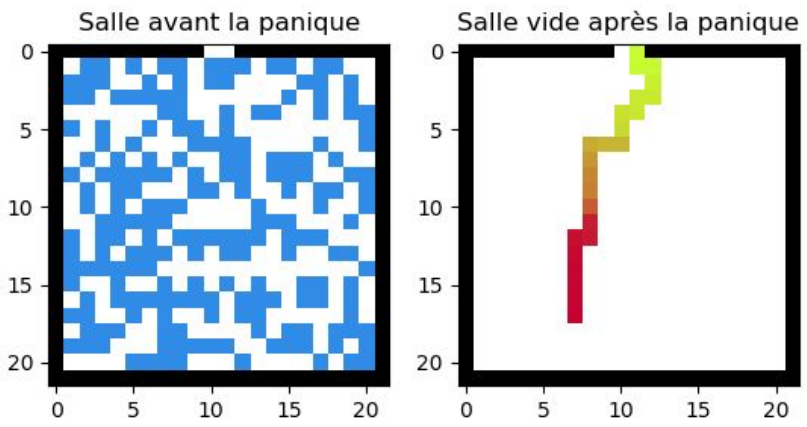
→ 2 fonctions affines :

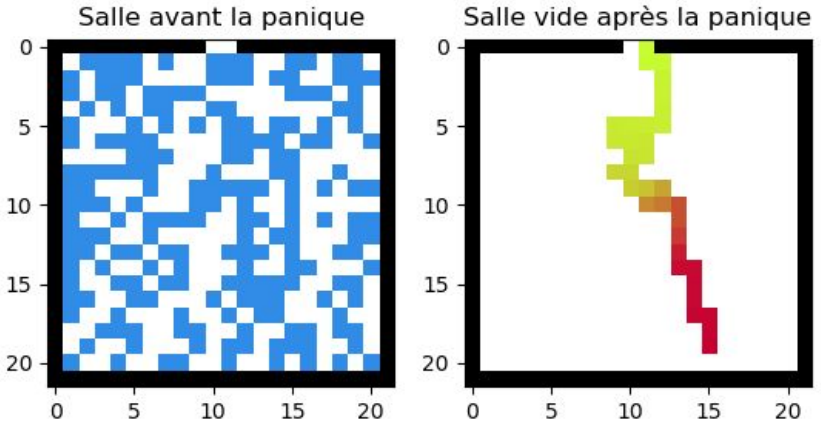
$$N_{etapes}(P) \simeq 0,82\,P + 5,3 \qquad N_{etapes}(P) \simeq P + 5,1$$


Nombre d'étapes pour une sortie générale en fonction de a dans une salle 20x20
Approche 1
Approche 2
Nombre d'étapes pour une sortie générale
Taille de l'ouverture de la pièce
400
350
300
250
200
150
100
50
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
20.0

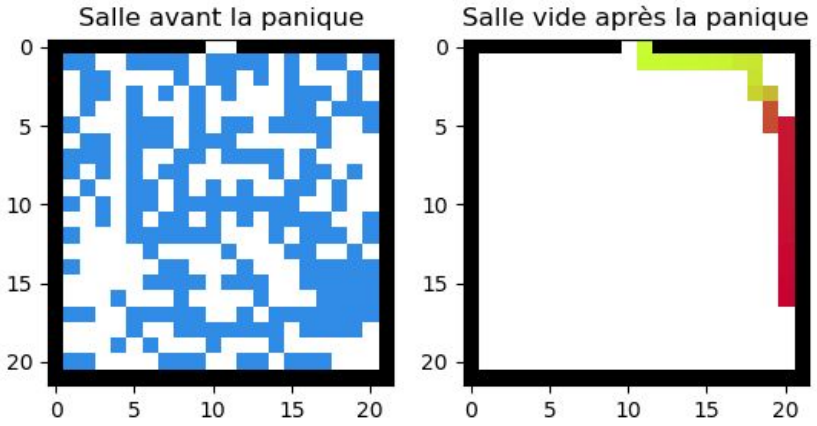
Qui sort en dernier ?

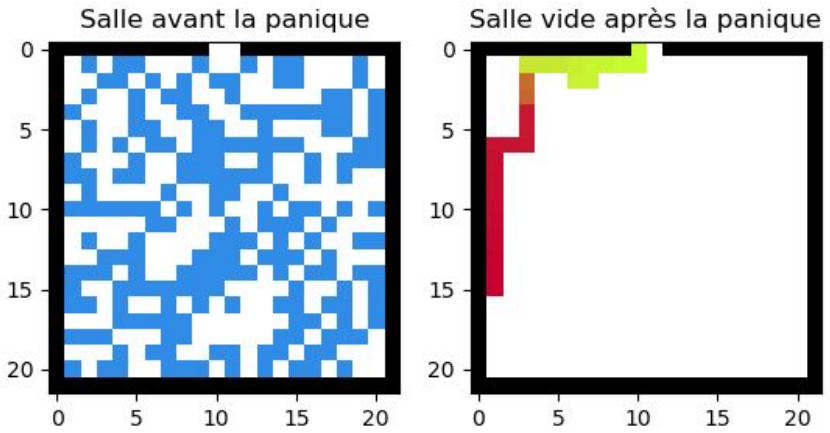
Approche 1 :





Approche 2 :





Finalement, quelle approche est la plus réaliste ?

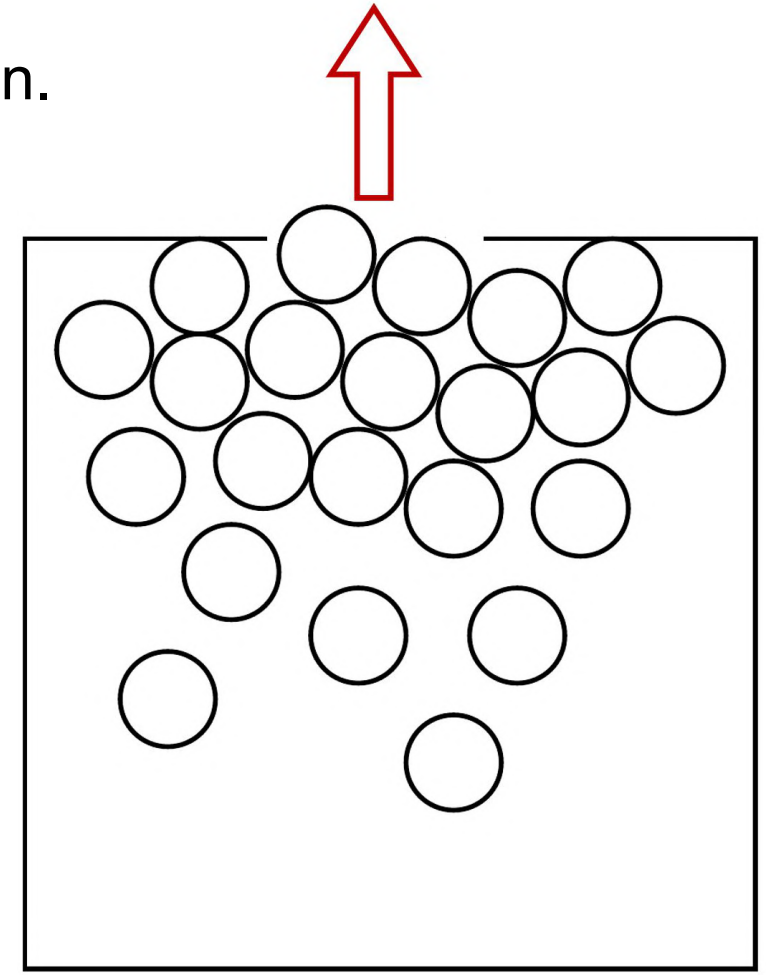
- L’approche 1 permet de mettre en évidence une certaine volonté individuelle.
- L’approche 2 propose une modélisation d’un déplacement global discret.
- L’approche 1 est plus **chaotique**.
- Une sortie avec l’approche 2 est plus **lente**.

Finalement, en analysant les derniers, l’approche 2 semble plus réaliste pour un phénomène d’évacuation d’urgence.

IV - Étude continue d'une évacuation

1. Étude empirique d'une évacuation

Expérience simulant une évacuation.
On étudie 2 vidéos.

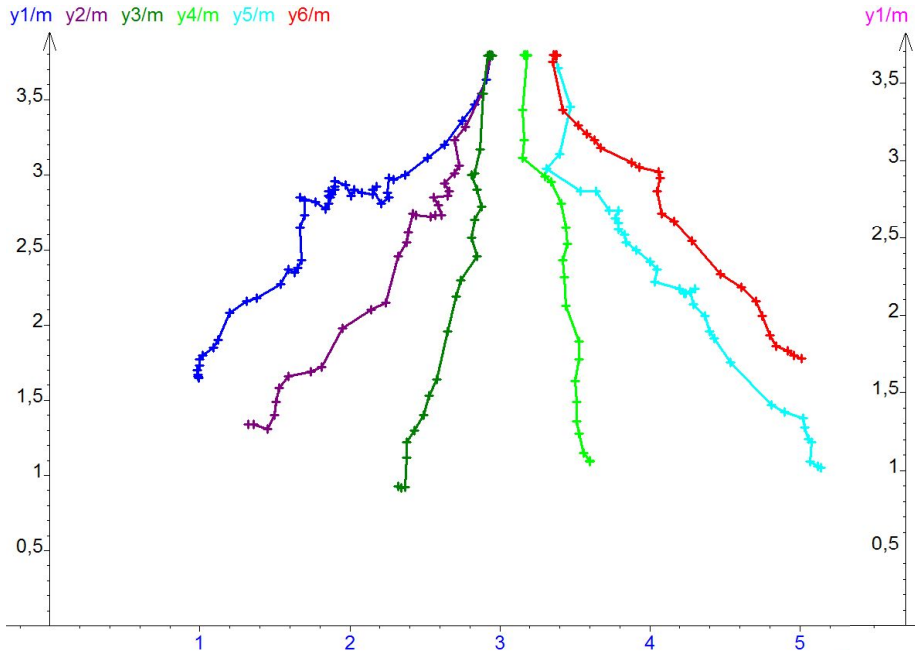


1. La foule doit sortir rapidement mais sans se bousculer
2. La foule doit se précipiter vers la sortie

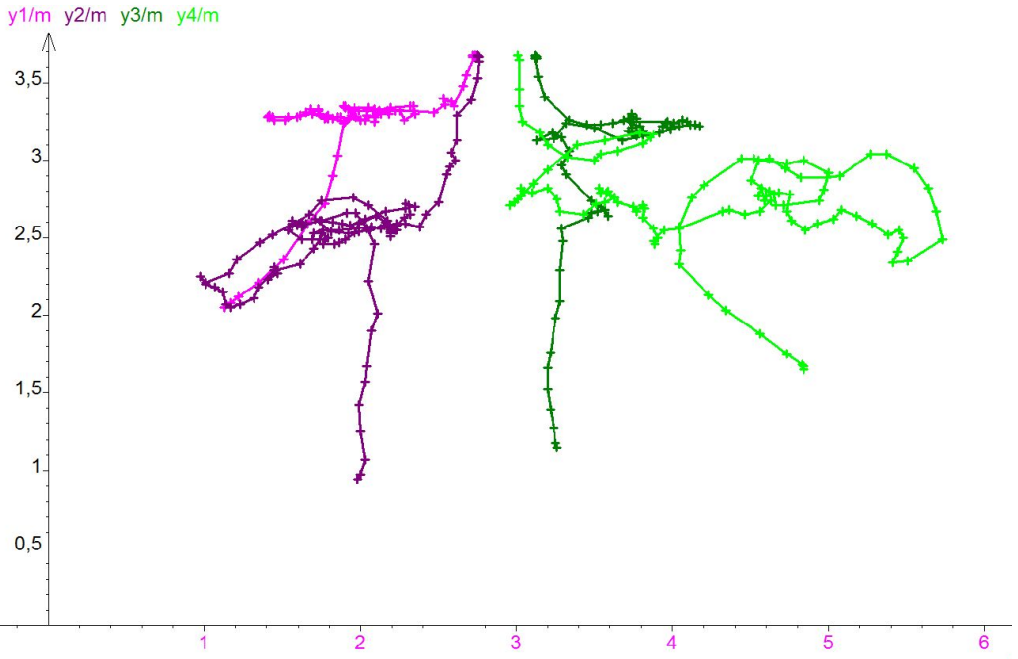
Vidéo 1 :

Vidéo 2 :

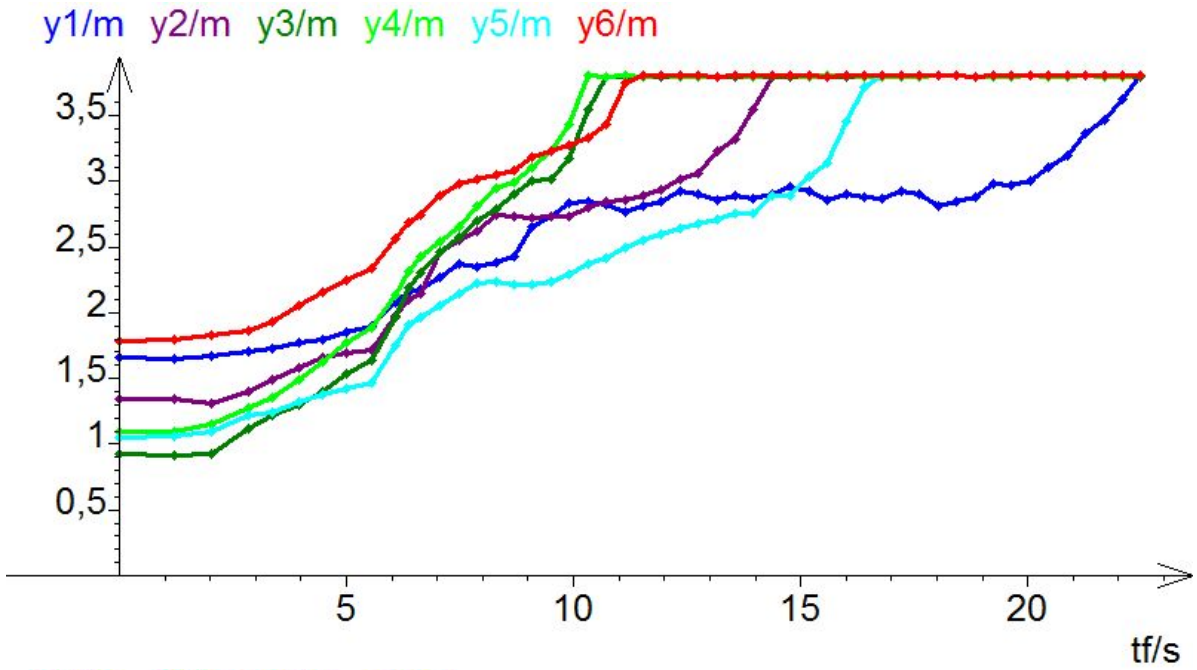
Pointage de l'expérience



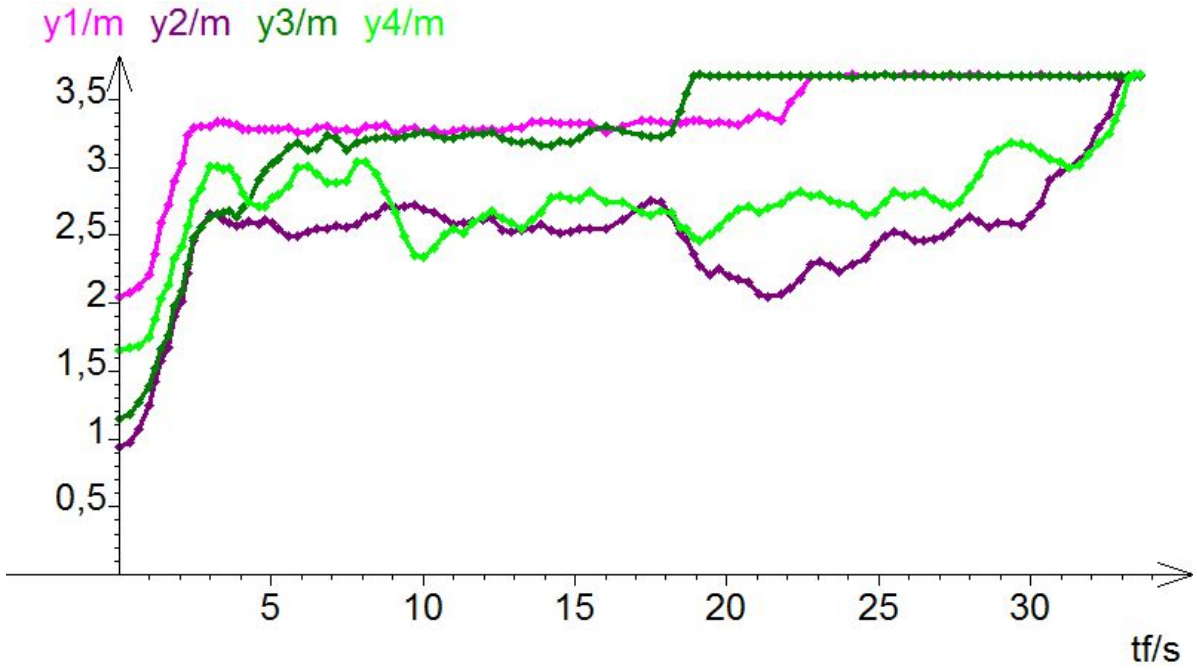
Vidéo 1



Vidéo 2



Vidéo 1



Vidéo 2

2. Le modèle des forces sociales

Chaque individu est soumis à 3 principes :

1. Conserver une vitesse constante v_0.
2. Se diriger vers un objectif.
3. Éprouver une répulsion (minimale) envers les autres.

Chaque piéton est alors attiré vers sa destination par une "force motrice" donnée par :

$$\overrightarrow{f_0} = \frac{1}{\tau}(v_0\,\vec{e} - \vec{v})$$

À chaque instant t, la vitesse d'un piéton varie par rapport aux interactions des autres piétons.

L'évolution de sa vitesse est alors donnée par :

$$m\frac{d\,\vec{v}(t)}{dt} = m\frac{1}{\tau}(v_0\,\vec{e} - \vec{v}) + \underbrace{\sum_i \overrightarrow{f_i}(t)}_{i} + \underbrace{\sum_k \overrightarrow{f_k}}_{k} + \underbrace{\overrightarrow{\xi}(t)}$$

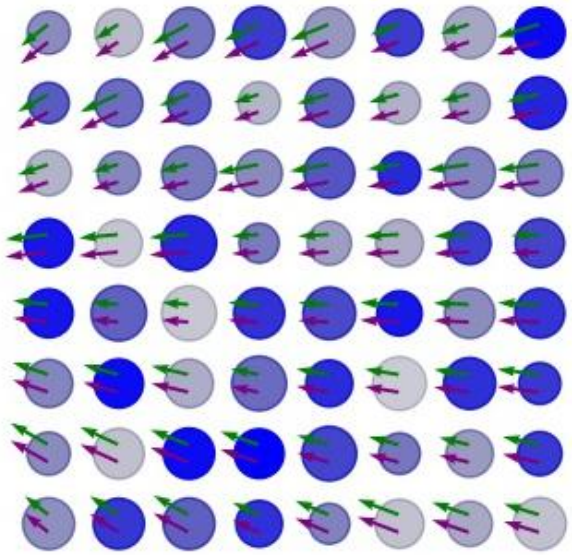
i : Les forces d'interaction auquel il est soumis avec les autres piétons

k : Les forces d'interaction auquel il est soumis avec les murs

$\overrightarrow{\xi}(t)$: Un terme de fluctuation aléatoire

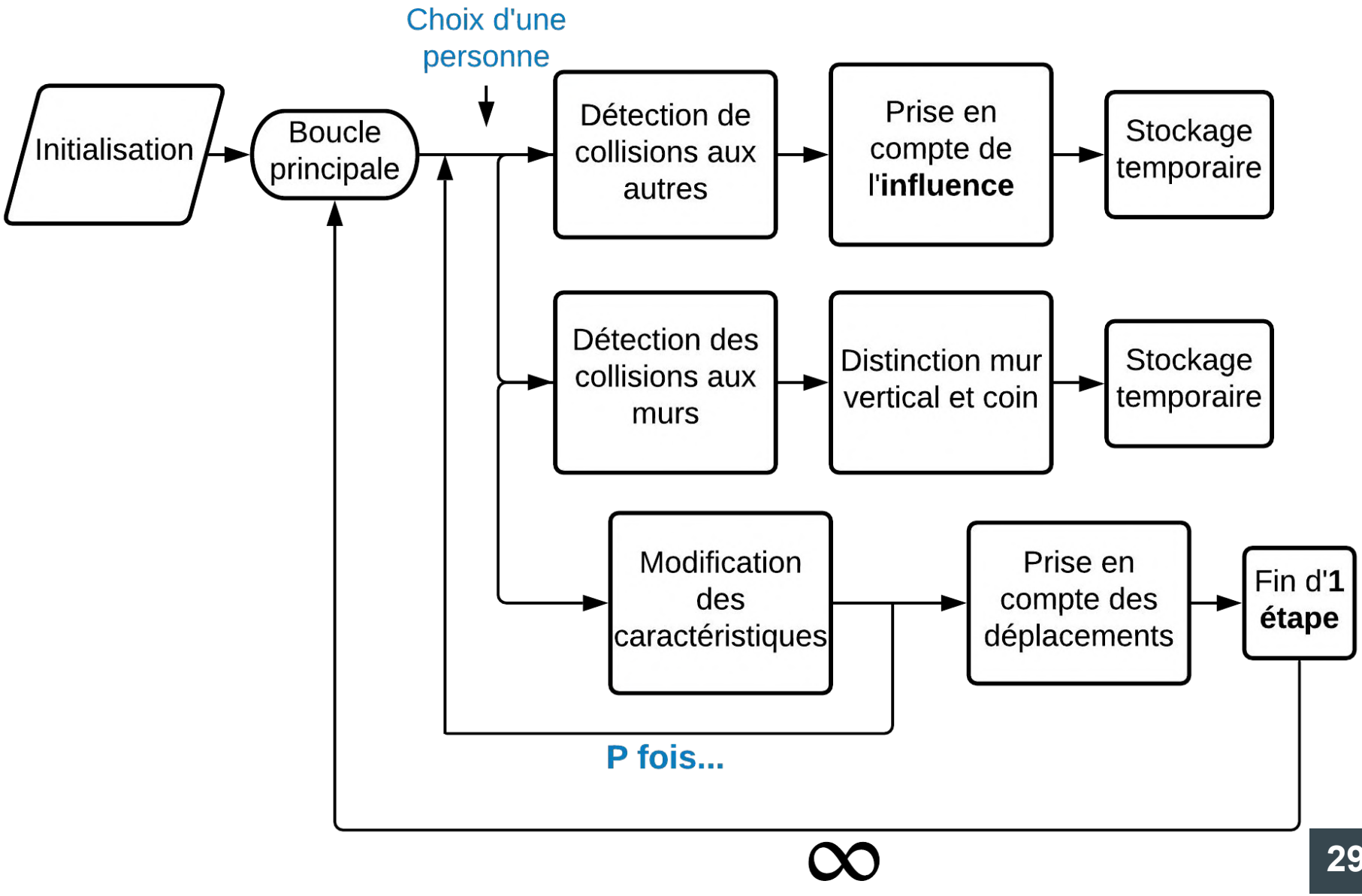
3. Implémentation de l'algorithme

- Placement aléatoire de P personnes.
- Assimilation à des cercles
- Chaque personne reçoit différentes caractéristiques de façon aléatoire :
  - Une vitesse propre
  - Une taille (rayon)
  - Une volonté
  - Une liste d'objectifs de déplacement

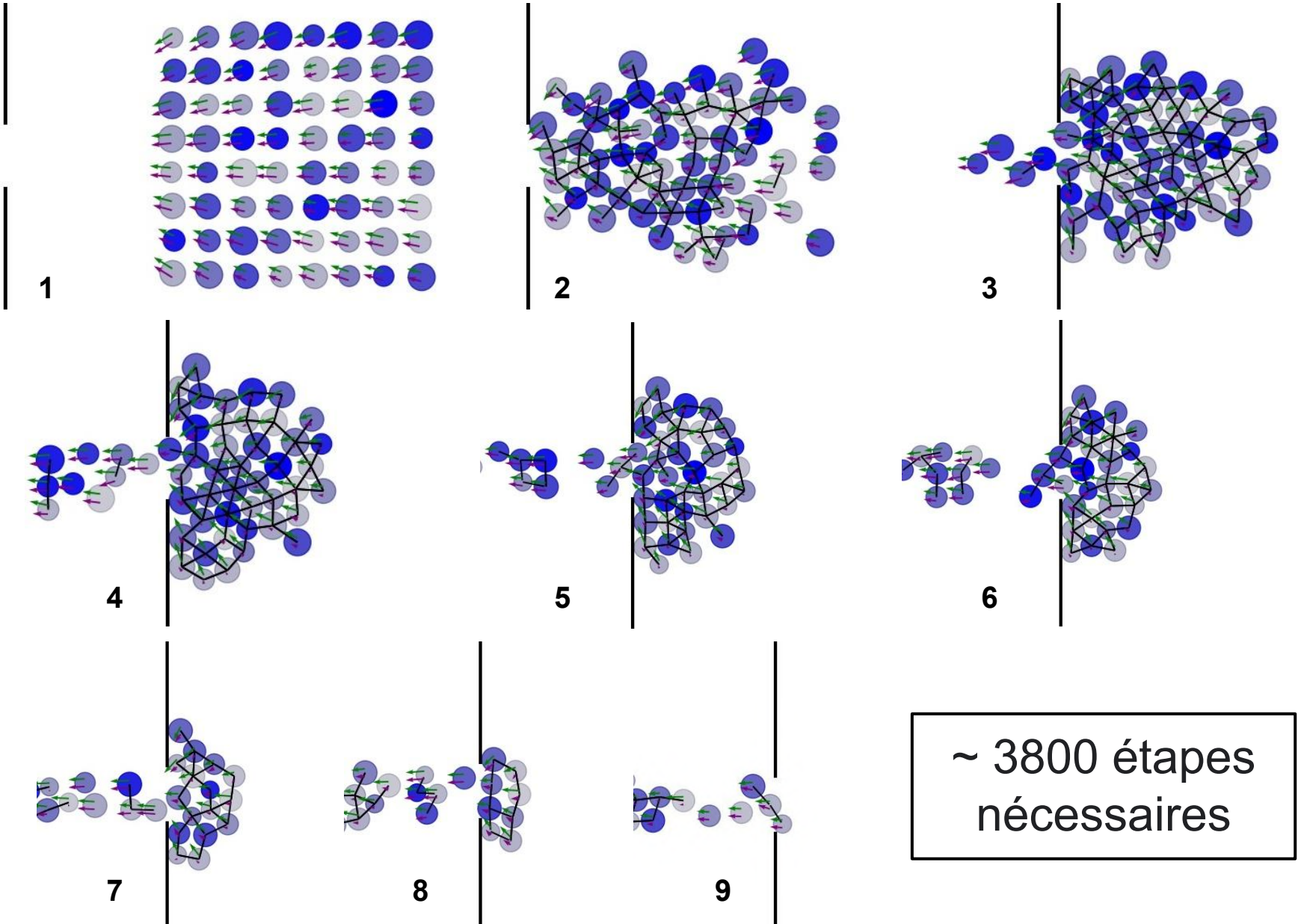


P = 64

Principe de l'algorithme



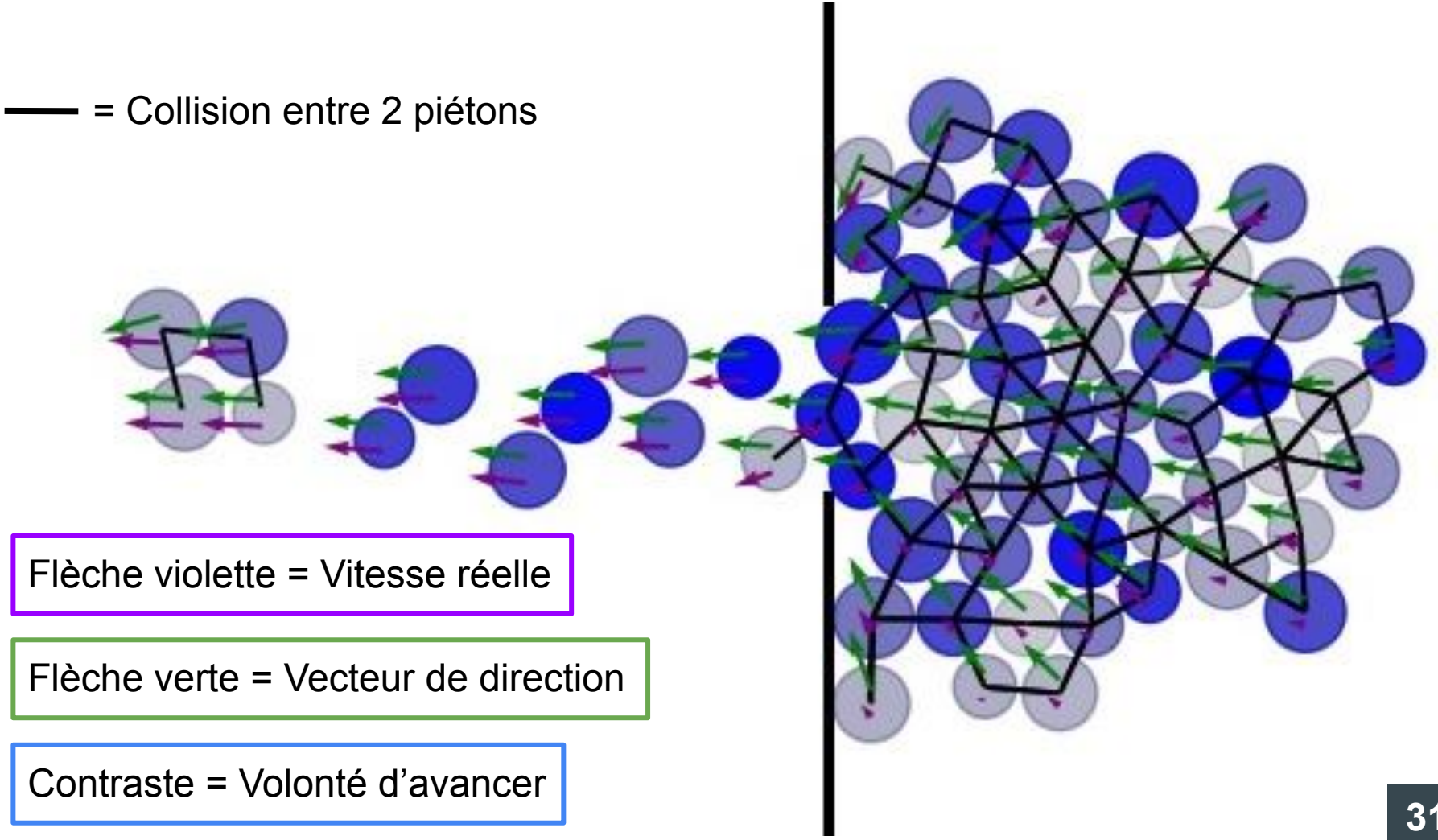
4. Résultats numériques



~ 3800 étapes nécessaires

5. Comprendre les mécanismes

Ce type d'évacuation mènent à de nombreux **chocs** entre piétons.



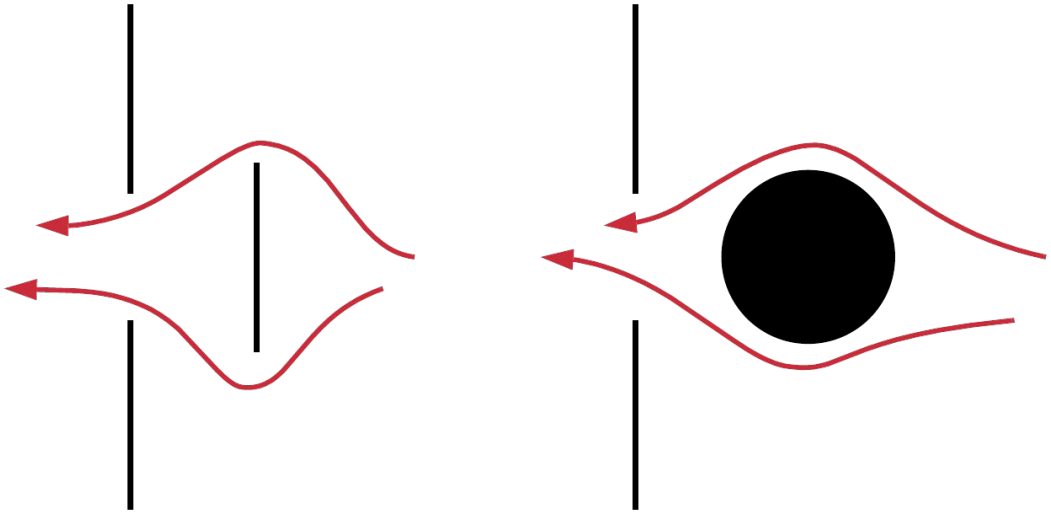
6. Recherche de solutions

L'idée est de placer un obstacle devant la sortie pour limiter ce réseau de forces d'interactions.

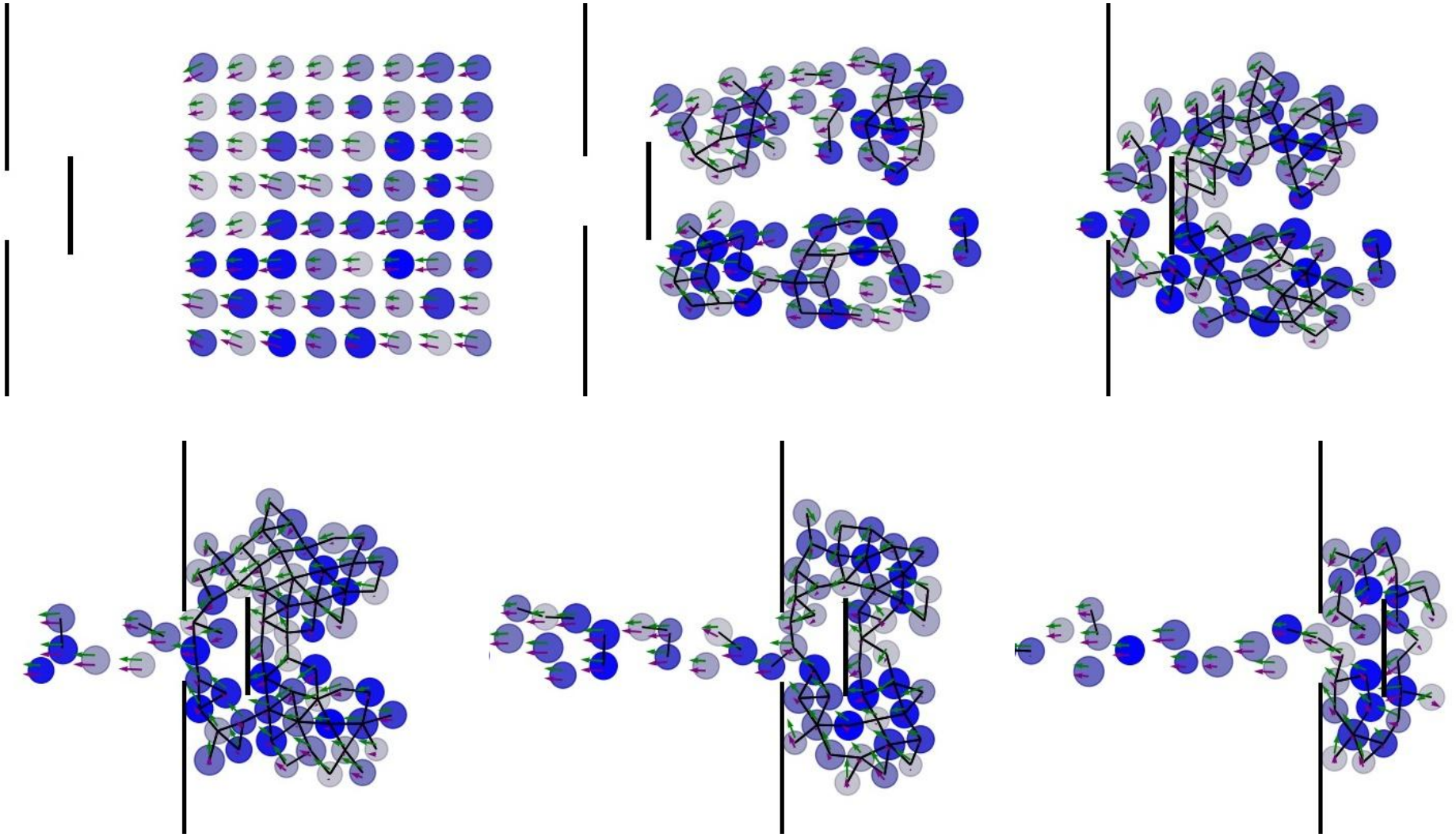
Différentes idées s'offrent à nous, on en traite 2 :

→ Placer un mur devant la sortie

→ Placer un large poteau devant la sortie

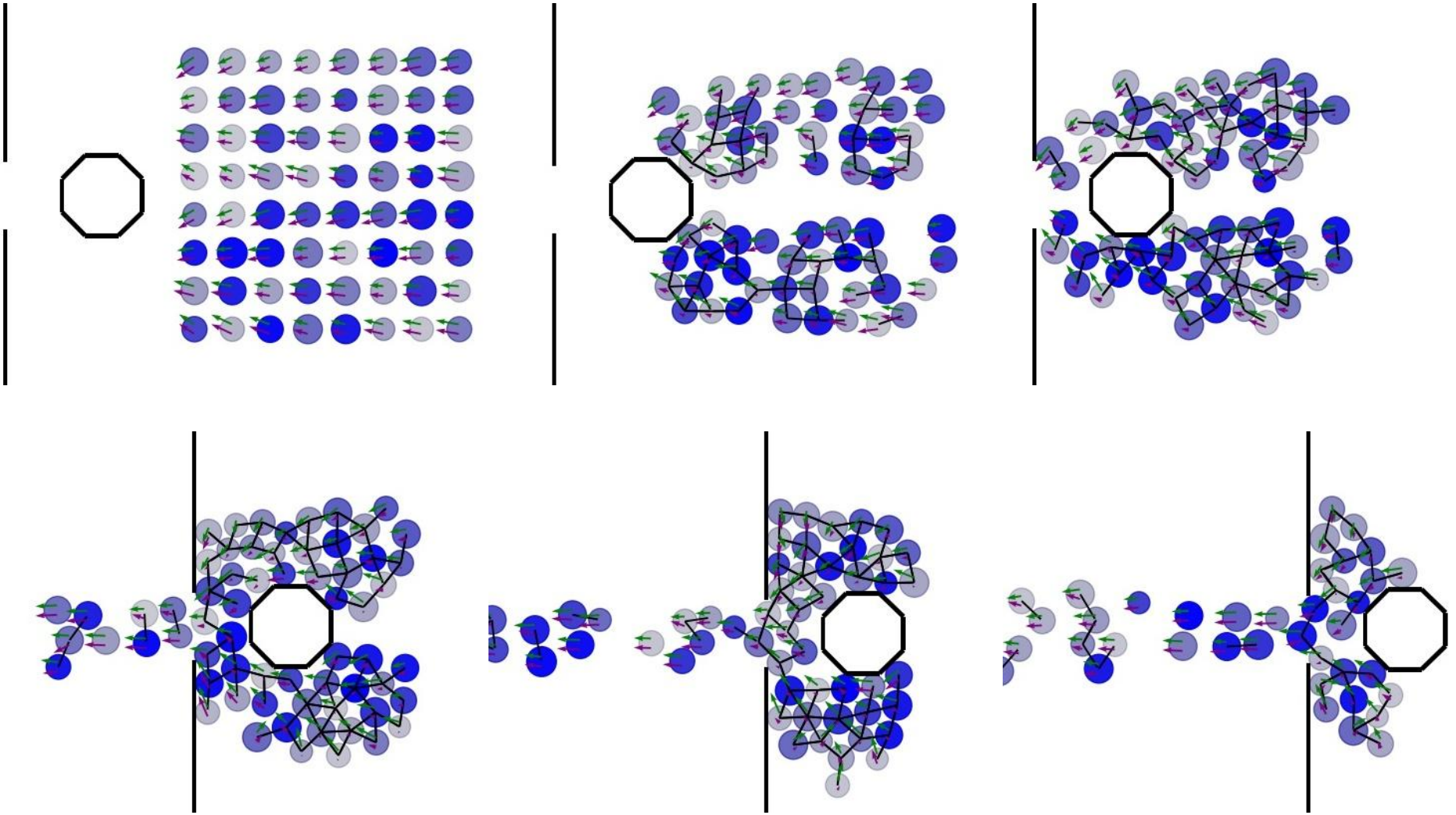


Solution d’un mur devant la sortie



~ 3550 étapes nécessaires

Solution d’un hexagone devant la sortie



~ 3200 étapes nécessaires

Analyses des résultats

- Lorsque la foule se précipite vers la porte, les bousculades provoquent une sortie 25% plus lente. Apparition d'une forme de **chaos**.
- Les dernières personnes à sortir sont, en majorité, celles sur les côtés de la porte.
- La modélisation informatique est cohérente par rapport à l'expérience.
- La sortie est plus rapide en divisant le flux de personnes.

V - Conclusion

- En cas de panique, les bousculades ont pour fondement un réel dilemme social.
- Nos modèles informatiques permettent une meilleure compréhension des phénomènes d'évacuation.
- Diviser le flux de sortie devient une idée simple, mais essentielle pour éviter des potentielles catastrophes.
- D'autres facteurs rentrent en jeu comme le stress, une trop forte densité, etc...

Annexes :

➔ Fonctions communes aux approches 1 et 2 des automates cellulaires

```
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.animation as animation
import random as r
import math as m
import time

# Variable globale #

p=0.9 #proba de se déplacer dans le bon sens
C=0.7 #couleur des personnes dans la salle

# # # # # # # # # # #

## Importation des fonctions classiques de AC 1 et AC 2

# # # # # # # # # # #     Fonctions classiques     # # # # # # # # # # #

def init0(n):
    tab=np.zeros((n+2,n+2))
    for k in range(n+2):
        tab[0,k]=1
        tab[n+1,k]=1
        tab[k,0]=1
        tab[k,n+1]=1
    return(tab)

def ouverture(tab,n):
    t=np.copy(tab)
    d,_=np.shape(tab)

    assert n<=(d-2)         #Taille de la porte maximale
    for i in range((d-n)//2,(d+n)//2):
        t[0,i]=0
    return(t)

def init(taille,taille_ouverture):
    return(ouverture(init0(taille),taille_ouverture))
```

Annexes :

```
def placer_gens(tab,P):
    t=np.copy(tab)
    d,_=np.shape(tab)
    l_pers=[]
    assert P<=(d-2)**2      #Le nombre maximal de personnes

    m=P
    while m!=0:
        i,j=r.randint(1,d-2),r.randint(1,d-2)
        if t[i,j] != C:
            t[i,j]=C
            l_pers.append((i,j))
            m-=1
    return(t,l_pers)

def angle(tab,personne):
    n0=(tab[0]!=1).sum()
    d,_=np.shape(tab)
    a,b=(d-n0)//2,(d+n0)//2
    j_ouv=[k for k in range(a,b)]
    i,j=personne

    if (j in j_ouv):
        alpha=0
    elif i==1 and j<a:
        alpha=m.pi/2
    elif i==1 and j>=b:
        alpha= -m.pi/2
    elif j<a:
        alpha=m.atan((a-j)/(i-1))
    else:
        alpha=m.atan(((b-1)-j)/(i-1))

    return(alpha)
```

Annexes :

```
def proba(tab,personne):
    t=np.copy(tab)
    d,_=np.shape(tab)
    alpha=angle(t,personne)
    i,j=personne

    #[haut,gauche,droite,bas]
    if alpha==0 and (i==1 or i==2):
        proba=[1,0,0,0]
    elif alpha==0:
        proba=[p,(1-p)/2,(1-p)/2,0]
    elif alpha==m.pi/2:
        proba=[(1-p)/2,0,p,(1-p)/2]
    elif alpha== -m.pi/2:
        proba=[(1-p)/2,p,0,(1-p)/2]
    elif alpha>=m.pi/4:
        proba=[1-p/2,0,p/2,0]
    elif alpha<= -m.pi/4:
        proba=[1-p/2,p/2,0,0]
    elif alpha>0:
        proba=[p,0,1-p,0]
    else:
        proba=[p,1-p,0,0]

    return(proba)
```

Annexes :

➔ Fonctions relatives à l'approche 1 des automates cellulaires

```
# # # # #     Relatif à l'approche 1 des automates cellulaires     # # # # #

def est_bloque(tab,personne):           #1
    i,j=personne
    dispo=[(i-1,j),(i,j-1),(i,j+1),(i+1,j)]

    for k in dispo:
        if tab[k]==0:
            return(False)
    return(True)

#1 : Définie si une personne est entièrement entourée par d'autres
```

Annexes :

```
def deplacement(tab,personne,sortie):
    t=np.copy(tab)
    i,j=personne
    dispo=[(i-1,j),(i,j-1),(i,j+1),(i+1,j)] #1
    pr=proba(t,personne)
    sum_proba=[pr[0],pr[0]+pr[1],pr[0]+pr[1]+pr[2],1.]
    rep,res=0,0

    if sortie:
        return(t,(i,j),sortie)
    elif i==0:                          #2
        t[i,j]=0
        sortie=True
        return(t,(i,j),sortie)
    elif est_bloque(t,(i,j)):           #3
        return(t,(i,j),sortie)
    else:
        while t[i,j]!=0 and rep<10:     #4
            res=0
            x=r.random()
            while x>sum_proba[res]:
                res+=1
            if t[dispo[res]]==0:
                t[i,j]=0
                t[dispo[res]]=C
                return(t,dispo[res],sortie)
            rep+=1
        return(t,(i,j),sortie)
```

#1 : [haut,gauche,droite,bas]
#2 : La personne va sortir ! Donc sortie = True
#3 : Si elle est bloquée, elle reste à sa place
#4 : La valeur 10 est purement arbitraire et permet de contrer un aléa extrêmement défavorable. Si au bout de 10 rep, le "res" ne change pas, la personne est considérée comme têtu et elle restera donc immobile.

Annexes :

```
#Toutes les personnes se déplacent (si elles peuvent) d'une case, chacune dans un ordre
aléatoire
#l_sortie est un liste de booléens indiquant si les personnes sont sorties ou non
#l_pos est une liste des listes des différentes positions des différentes personnes

def deplacement_total(tab,l_pos,l_sortie):
    t=np.copy(tab)
    n=len(l_pos)
    l=[l_pos[i][0] for i in range(n)]
    l_ind=[i for i in range(n)]

    for k in range(n):
        numero_p=r.choice(l_ind)            #1
        q=l_ind.index(numero_p)             #2
        pers=l.pop(q)
        l_ind.pop(q)
        t,suiv,sortie=deplacement(t,pers,l_sortie[numero_p])
        l_sortie[numero_p]=sortie           #3

        if not(sortie):
            l_pos[numero_p]=[suiv]+l_pos[numero_p]    #4

    return(t,l_pos,l_sortie)

#1 : numero_p est le numero de la personne dans l_pos
#2 : q est purement utilitaire et sert uniquement à correctement vider la liste l de
manière totale et aléatoire
#3 : On indique si la personne est sortie ou non
#4 : Si c'est non, on indique la prochaine position de la personne
```

Annexes :

```
# # # # #     Sortie finale et nombre de répétitions    # # # # #

def sortie_1(tab,l_pers):       #1
    n=len(l_pers)
    l_pos=[[l_pers[i]] for i in range(n)]
    l_sortie=[False for _ in range(n)]
    t=np.copy(tab)
    rep=0

    while (False in l_sortie):      #2
        t,l_pos,l_sortie=deplacement_total(t,l_pos,l_sortie)
        rep+=1
    return(t,rep,l_pos)

#1 : l_pers = liste des emplecements initiaux de toute les personnes
#2 : La boucle continue tant qu'il y a des personnes dans la salle



# # # # #      Algorithme final    # # # # #

def simulation_1(N,a,P):
    salle,l_pers=placer_gens(init(N,a),P)
    salle_finale,duree,l_pos=sortie_1(salle,l_pers)
    return(salle,salle_finale,duree,l_pos)
```

Annexes :

➔ Fonctions relatives à l'approche 2 des automates cellulaires

```
# # # # #   Relatif à l'approche 2 des Automates Cellulaires   # # # # #

def envies_pers(tab,personne):                #1
    t=np.copy(tab)
    i,j=personne
    dispo=[(i-1,j),(i,j-1),(i,j+1),(i+1,j)] #2
    pr=proba(t,personne)
    sum_proba=[pr[0],pr[0]+pr[1],pr[0]+pr[1]+pr[2],1.]
    l=[]
    res=0
    x=r.random()
    while x>sum_proba[res]:
        res+=1
    return(dispo[res])

#1 : Renvoie la position souhaitée pour le prochain déplacement
#2 : [haut,gauche,droite,bas]

def envies(tab,l_pers):           #1
    t=np.copy(tab)
    n=len(l_pers)
    l_envies=[]

    for pos in l_pers:
        l_envies.append(envies_pers(t,pos))
    return(l_envies)

#1 : Crée la liste des envies avec l'envie de chaque personne dans la salle
```

Annexes :

```
def rivalite(l_envies):            #1
    n=len(l_envies)
    l,l_mem=[],[]
    for k in range(n):
        pos=l_envies[k]
        if pos in l_mem:
            ind=l_mem.index(pos)
            l[ind]=(pos,l[ind][1]+[k])
        else:
            l_mem.append(pos)
            l.append((pos, [k]))
    return(l)

#1 : Construit une liste de couple composé d'une position appartenant à l_envie et la
liste des personne qui souhaitent y aller

def supprimer_pers(l,l_ind):        #1
    n=len(l)
    i=0
    while i<n:
        m=len(l[i][1])
        k=0
        while k<m:
            if not(l[i][1][k] in l_ind):
                l[i][1].pop(k)
                k=k-1
                m=m-1
            k+=1

        if l[i][1]==[]:
            l.pop(i)
            i=i-1
            n=n-1
        i+=1
    return(l)

#1 : l sera dans notre cas une liste de rivalités
```

Annexes :

```
def deplacement_global(tab,l_pos,l_sortie):
    t=np.copy(tab)
    n=len(l_pos)
    l_pers=[l_pos[i][0] for i in range(n)]
    l_ind=[i for i in range(n)]
    l_envies=envies(t,l_pers)

    for i in range(n):
        if l_sortie[i]:                                    #1
            l_ind.remove(i)
        else:
            if l_pers[i][0]==0:                            #2
                l_sortie[i]=True
                t[l_pers[i]]=0
                l_ind.remove(i)
            else:
                if t[l_envies[i]]!=0:                      #3
                    l_pos[i]=[l_pers[i]]+l_pos[i]
                    l_ind.remove(i)

    l=rivalite(l_envies)
    l=supprimer_pers(l,l_ind)                              #4
    m=len(l)

    for k in range(m):
        numero_p=r.choice(l[k][1])                         #5
        l[k][1].remove(numero_p)
        t[l[k][0]]=C
        t[l_pers[numero_p]]=0
        l_pos[numero_p]=[l[k][0]]+l_pos[numero_p]          #6

        for x in l[k][1]:                                  #7
            l_pos[x]=[l_pers[x]]+l_pos[x]

    return(t,l_pos,l_sortie)
```

Annexes :

```
#1 : Si la personne est déjà sortie, on l'ignore
#2 : La personne va sortir donc sortie = True
#3 : Si son souhait de position est pris, la personne restera à sa place
#4 : On supprime dans la liste des rivalités les personnes qui n'appartiennent pas à l_ind
#5 : Choix aléatoire d'une personne parmi celles qui s'intéressent à l[k][0]
#6 : Cette personne se déplace car elle a été choisie
#7 : Les autres restent à leur place

# # # # #    Sortie finale et nombre de répétitions   # # # # #

def sortie_2(tab,l_pers):        #1
    n=len(l_pers)
    l_pos=[[l_pers[i]] for i in range(n)]
    l_sortie=[False for _ in range(n)]
    t=np.copy(tab)
    rep=0

    while (False in l_sortie):
        t,l_pos,l_sortie=deplacement_global(t,l_pos,l_sortie)
        rep+=1
    return(t,rep,l_pos)

#1 : l_pers = liste des emplecements initiaux de toute les personnes




# # # # #    Algorithme final    # # # # #

def simulation_2(N,a,P):
    salle,l_pers=placer_gens(init(N,a),P)
    salle_finale,duree,l_pos=sortie_2(salle,l_pers)
    return(salle,salle_finale,duree,l_pos)
```

Annexes :

➔ Analyse des résultats pour les deux approches

```
## Simulation graphique

#N = Nombre de cases par côté
#P = Nombre de personnes dans la salle
#e = Taille ouverture

def moyenne(l):
    n=len(l)
    S=0
    assert n>0
    for i in range(n):
        S+=l[i]
    return(S/n)

def l_duree_1(N,a,P,nombre):    #1
    L=[]
    for _ in range(nombre):
        _,_,d,_=simulation_1(N,a,P)
        L.append(d)
    return(L)

#1 : Liste des durées obtenues pour des valeurs de N,a,P fixées avec l'approche 1

def l_duree_2(N,a,P,nombre):    #1
    L=[]
    for _ in range(nombre):
        _,_,d,_=simulation_2(N,a,P)
        L.append(d)
    return(L)

#1 : Liste des durées obtenues pour des valeurs de N,a,P fixées avec l'approche 2
```

Annexes :

```
def resultats_1(N,a):           #1
    l_abs=[]
    l_ord=[]
    for P in range(0,N**2+1,5):
        print(P)
        l_abs.append(P)
        l_ord.append(moyenne(l_duree_1(N,a,P,20)))  #2
    return(l_abs,l_ord)

#1 : Retourne la liste des moyennes des durées de sortie en faisant varier le nombre de
personnes (approche 1)
#2 : Répéter 20 fois est arbitraire

def resultats_2(N,a):   #1
    l_abs=[]
    l_ord=[]
    for P in range(0,N**2+1,5):
        print(P)
        l_abs.append(P)
        l_ord.append(moyenne(l_duree_2(N,a,P,20)))  #2
    return(l_abs,l_ord)

#1 : Retourne la liste des moyennes des durées de sortie en faisant varier le nombre de
personnes (approche 2)
#2 : Répéter 20 fois est arbitraire

l_1=resultats_1(10,2)
l_2=resultats_2(10,2)
```

Annexes :

```
def resultats_porte_1(N,P):   #1
    l_abs=[]
    l_ord=[]
    for a in range(1,N+1):
        print(a)
        l_abs.append(a)
        l_ord.append(moyenne(l_duree_1(N,a,P,20)))
    return(l_abs,l_ord)

#1 : Retourne la liste des moyennes des durées de sortie en faisant varier la taille de
l'ouverture (approche 1)

def resultats_porte_2(N,P):   #1
    l_abs=[]
    l_ord=[]
    for a in range(1,N+1):
        print(a)
        l_abs.append(a)
        l_ord.append(moyenne(l_duree_2(N,a,P,20)))
    return(l_abs,l_ord)

#1 : Retourne la liste des moyennes des durées de sortie en faisant varier la taille de
l'ouverture (approche 2)

lp_1=resultats_porte_1(20,200)
lp_2=resultats_porte_2(20,200)
```

Annexes :

➔ Tracé des résultats et modélisation des courbes

```
## Affichage variation P

plt.scatter(l1[0],l1[1],s=40,c = "red")
plt.plot(l1[0],l1[1],c = 'r',linestyle='dashed',label= "Approche 1")

x = np.linspace(0, 100)
y = 0.82*x +5.3                          #La modélisation
plt.plot(x, y, c='k',linewidth=2)

plt.scatter(l2[0],l2[1],s=40,c = "blue")
plt.plot(l2[0],l2[1],c = 'b',linestyle='dashed',label= "Approche 2")

x = np.linspace(0, 100)
y = 0.99*x +5.1                          #La modélisation
plt.plot(x, y, c='k',linewidth=2)

plt.legend(loc=2,fontsize=15)
plt.title("Nombre d'étapes pour une sortie générale en fonction de P dans une salle
10x10",fontsize=22)
plt.xlabel("Nombre de personnes dans la pièce",fontsize=15)
plt.ylabel("Nombre d'étapes pour une sortie générale",fontsize=15)
plt.grid()

plt.show()
```

Annexes :

```
## Affichage variation a

plt.scatter(lp1[0],lp1[1],s=40,c = "red")
plt.plot(lp1[0],lp1[1],c = 'r',linestyle='dashed',label= "Approche 1")

x = np.linspace(0.5, 20,100)
y = 320*x**(-0.8)                    #La modélisation
plt.plot(x, y, c='k',linewidth=2)

plt.scatter(lp2[0],lp2[1],s=40,c = "blue")
plt.plot(lp2[0],lp2[1],c = 'b',linestyle='dashed',label= "Approche 2")

x = np.linspace(0.5, 20,100)
y = 390*x**(-0.6)                    #La modélisation
plt.plot(x, y, c='k',linewidth=2)

plt.legend(fontsize=15)
plt.title("Nombre d'étapes pour une sortie générale en fonction de a dans une salle
20x20",fontsize=22)
plt.xlabel("Taille de l'ouverture de la pièce",fontsize=15)
plt.ylabel("Nombre d'étapes pour une sortie générale",fontsize=15)
plt.grid()

plt.show()
```

Annexes :

➔ Modèle des chocs entre individus dans un espace continu

```
import matplotlib . pyplot as plt
import matplotlib . image as mpimg
import math as m
import numpy as np

from numpy import linalg as LA

np.random.seed(1000)
# Permet de conserver un même aléa pour une série de différents tests

# # # Parametre Matplotlib # # #

plt.rcParams['toolbar'] = 'None'
fig = plt.figure(facecolor='w')
plt.ion()
fig.canvas.set_window_title('Les interactions')
fig.subplots_adjust(left=0, right=1, top=1, bottom=0)
ax = fig.add_subplot(111,aspect='equal')

# # # Initialisation des tailles (personnes et salle) pour la modélisation # # #

windowsize = [9.5,9.5]
frontieres = [-10,10 , -10,10 ]
nbpixel = windowsize[0]*fig.dpi
taillepointnormal = nbpixel/ (-frontieres[0]+frontieres[1]) * 1.3

def dotsize(n) :
    return (n*taillepointnormal)**2
```

Annexes :

```
# # # Variables globales # # #

filename = "Colin_test_"
Path = "/home/colin180302/IPT/TIPE/TESTS/TEST1/"
nb = 64  #Le nombre de personnes
epsilon = 0.1
SaveFile = True
pasExp = 20
RDetect = 1
t = 0
Ic = 0
```

Annexes :

```
# # # # # Fonction de création : Initialisation # # # # #

def initialisation(mx,my,e,Lr,nb,lV,hV,lv,hv,lr,hr,aim1,aim2,col) :
    volonte = np.random.random((nb,1))      #1
    Rayon = np.random.random((nb))*(hr-lr) +(lr+hr)/2   #2
    vitesse = np.random.random((nb,1))*(hV-lV) +(lV+hV)/2 #3
    position = np.ones(shape=(0,2)) #4
    nbre_aims = len(aim1)
    aimR = np.ones(shape=(0,nbre_aims,2)) #4

    for i in range(nb): #5
        position = np.append(position , [[  e * (i% Lr) , e* (i//Lr)   ]] ,axis=0)
        if i< (nb//2) :
            aimR = np.append(aimR, [aim1], axis=0)
        else:
            aimR = np.append(aimR, [aim2], axis=0)

    position = position + [mx,my- 1/2* e * (nb//Lr-1)] #6
    aimlist = aimR.tolist()
    col = np.full( (nb,4),[col]) * (volonte*0.8+0.2) #7
    volonte = volonte*(hv-lv) +(lv+hv)/2 #8
    return(volonte,Rayon,vitesse,position,aimlist ,col)

#1 : Initialisation des volontés
#2 : Initialisation des différents rayons
#3 : Initialisation des différentes vitesses
#4 : Tableau vide
#5 : Initialisation des positions des personnes
#6 : Re-positionnement des personnes pour être disposées symétriquement devant la porte
#7 : Couleur en fonction de la volonté : Plus la volonté est haute, plus le contraste est
haut
#8 : Donne une meilleure cohérence à la répartition des volontés entre les personnes
```

Annexes :

```
def objectif(aimlist,ind):      #On détermine le tableau des objectifs pour chaque
personne
    Aim = np.zeros((len(aimlist),2),dtype=float)
    for i in range(len(aimlist)):
        Aim[i] = np.array(aimlist[i][ind[i]])
    return(Aim)

# Pour une modélisation simple : volonte,Rayon,vitesse,position,aim,couleur =
initialisation (3,0,0.8,8,nb,0.005,0.01,0.1,6,0.2,0.3,np.array([[-0.5,0],[-10,0]]),
np.array([[-0.5,0],[-10,0]]),(0,0,1,1) )

# Pour un mur devant : volonte,Rayon,vitesse,position,aim,couleur = initialisation
(4,0,0.8,8,nb,0.005,0.01,0.1,6,0.2,0.3,np.array([[1.2,-1.5],[0.5,-1.],[-0.5,0],[-10,0]]),
np.array([[1.2,1.5],[0.5,1.],[-0.5,0],[-10,0]]),(0,0,1,1) )

# Pour hexagone : volonte,Rayon,vitesse,position,aim,couleur = initialisation
(4,0,0.8,8,nb,0.005,0.01,0.1,6,0.2,0.3,np.array([[2.,-1.5],[0.72,-1.],[-0.5,0],[-10,0]]),
np.array([[2.,1.5],[0.72,1.],[-0.5,0],[-10,0]]),(0,0,1,1) )

volonte,Rayon,vitesse,position,aim,couleur = initialisation (4,0,0.8,8,nb,
0.005,0.01,0.1,6,0.2,0.3,np.array([[1.2,-1.5],[0.5,-1.],[-0.5,0],[-10,0]]),
np.array([[1.2,1.5],[0.5,1.],[-0.5,0],[-10,0]]),(0,0,1,1) )


# # # Création des murs # # #

# Pour une modélisation classique : LineList = np.array([[[0,-10],[0,-0.7]],[[0,0.7],
[0,10]]])

# Pour un mur devant : LineList = np.array([[[0,-10],[0,-0.7]],[[0,0.7],[0,10]],[[1.3,1.],
[1.3,-1]]])

# Pour un hexagone : LineList = np.array([[[0,-10],[0,-0.7]],[[0,0.7],[0,10]],[[1.2,0.35],
[1.69,0.84]],[[1.69,0.84],[2.39,0.84]],[[2.39,0.84],[2.9,0.35]],[[2.9,0.35],[2.9,-0.35]],
[[2.9,-0.35],[2.39,-0.84]],[[2.39,-0.84],[1.69,-0.84]],[[1.69,-0.84],[1.2,-0.35]],
[[1.2,-0.35],[1.2,0.35]]])

LineList = np.array([[[0,-10],[0,-0.7]],[[0,0.7],[0,10]],[[1.3,1.],[1.3,-1]]])
```

Annexes :

```
# # # # # La Procédure # # # # #

PosTemp = np.zeros(position.shape)
PastVit = np.zeros(position.shape)
direNorm = np.zeros(position.shape)
VitNorm = np.zeros(position.shape)
VitNormS = np.zeros(vitesse.shape)
IndAim = np.zeros(len(aim),dtype=int)

while True :
    direNorm = objectif(aim,IndAim) - position
    direNorm = np.divide( direNorm , np.expand_dims(LA.norm(direNorm,axis=1),axis=1) )
    ax.scatter(position[:,0],position[:,1],s = dotsize(Rayon) , c=couleur )
    PosTemp = position

    for part in range(vitesse.shape[0]) :
        partpos = position[part]
        Colldetect = position - partpos
        ColldetectNorm = LA.norm(Colldetect,axis=1)
        ColldetectNormR = ColldetectNorm-(Rayon[part]+Rayon)
        ColldetectNormR[part]=0
        indicecoll = np.argwhere(ColldetectNormR<0)
        indicenorm = indicecoll[:,0]
        volontetotal = volonte[part]
        influence = direNorm[part]* vitesse[part]*volonte[part]

        # # # # # Collisions dues aux personnes # # # # #
```

Annexes :

```
        # # # # # Collisions dues aux personnes # # # # #

        if len(indicenorm)>0 :

            ColldetectNormalized = np.divide(Colldetect[indicenorm],
np.expand_dims(ColldetectNorm[indicenorm],axis=1))

            ProduitScalaire = VitNorm[indicenorm][:,0]*ColldetectNormalized[:,0] +
VitNorm[indicenorm][:,1]*ColldetectNormalized[:,1]

            ProduitScalaire = np.expand_dims(ProduitScalaire,axis=1)

            influence = influence - np.sum( ColldetectNormalized
*np.abs(ProduitScalaire)*VitNormS[indicenorm] * volonte[indicenorm] ,axis=0 )

            volontetotal = volontetotal +
np.sum(volonte[indicenorm]*np.abs(ProduitScalaire))

            PosTemp[part] = PosTemp[part] +
np.sum(ColldetectNormalized*volonte[indicenorm]*np.expand_dims(ColldetectNormR[indicenorm]
, axis=1),axis=0)/(np.sum(volonte[indicenorm])+volonte[part])

            ax.quiver(np.repeat(partpos[0],len(indicenorm)),
np.repeat(partpos[1],len(indicenorm)), Colldetect[indicenorm][:,0], Colldetect[indicenorm]
[:,1],scale=0.8,scale_units='xy',width=0.002,headwidth=0)

        PosTemp[part] = PosTemp[part]  +  influence/volontetotal


        # # # # # Colisions dues aux murs # # # # #
```

Annexes :

```
        # # # # # Colisions dues aux murs # # # # #

        MurVec = LineList[:,1,:] - LineList[:,0,:]
        MurVecNorm = LA.norm(MurVec,axis=1)
        MurVecN = np.divide(MurVec, np.expand_dims(MurVecNorm,axis=1) )
        Mvec = np.repeat(np.expand_dims(PosTemp[part],axis=0),len(LineList),axis=0) -
LineList[:,0,:]
        Produit = Mvec[:,0]*MurVecN[:,0] + Mvec[:,1]*MurVecN[:,1]
        ColP = LineList[:,0,:] + MurVecN *  np.expand_dims(Produit,axis=1)
        MtP = ColP - PosTemp[part]
        DstTW = LA.norm(MtP,axis = 1)

        # # # Colision verticale # # #

        PremCondition = (DstTW<Rayon[part])&(Produit>0)&(Produit<MurVecNorm)
        IndiceColV = np.nonzero(PremCondition)
        MtPNorm = np.divide(MtP , np.expand_dims(LA.norm(MtP,axis=1),axis = 1))
        PosTemp[part] = PosTemp[part] +
np.sum(MtPNorm[IndiceColV]*np.expand_dims(DstTW[IndiceColV]- Rayon[part],axis=1),axis=0)

        # # # Colision d'un point bas # # #
```

Annexes :

```
        # # # Colision d'un point bas # # #

        ColPB = (Produit<=0)&( (np.power(Produit,2)+np.power(DstTW,
2))<=np.power(np.repeat(Rayon[part],len(Produit)),2) )
        IndiceColPBb = np.argwhere(ColPB)
        IndiceColPB = IndiceColPBb[:,0]

        if(len(IndiceColPB) > 0 ):
            VecColPB =
np.repeat(np.expand_dims(PosTemp[part],axis=0),len(IndiceColPB),axis=0) -
LineList[IndiceColPB,0,:]
            VecColPBNorm = LA.norm(VecColPB,axis=1)
            VecColPBN = np.divide(VecColPB,np.expand_dims(VecColPBNorm,axis=1))
            PosTemp[part] = PosTemp[part] - np.sum(VecColPBN*np.expand_dims(VecColPBNorm-
Rayon[part],axis=1),axis=0)

        # # # Colision d'un point haut # # #

        ColPH = (Produit>=MurVecNorm)&( (np.power(Produit-MurVecNorm,2)+np.power(DstTW,
2))<=np.power(np.repeat(Rayon[part],len(Produit)),2) )
        IndiceColPHb = np.argwhere(ColPH)
        IndiceColPH = IndiceColPHb[:,0]

        if(len(IndiceColPH) > 0 ):
            VecColPH =
np.repeat(np.expand_dims(PosTemp[part],axis=0),len(IndiceColPH),axis=0) -
LineList[IndiceColPH,1,:]
            VecColPHNorm = LA.norm(VecColPH,axis=1)
            VecColPHN = np.divide(VecColPH,np.expand_dims(VecColPHNorm,axis=1))
            PosTemp[part] = PosTemp[part] - np.sum(VecColPHN*np.expand_dims(VecColPHNorm-
Rayon[part],axis=1),axis=0)

        # # # Fin des tests # # #
```

Annexes :

```
        # # # Fin des tests # # #

        PastVit[part]=influence/volontetotal
        VitNormS[part] = LA.norm(PastVit[part])
        VitNorm[part] = PastVit[part]/ VitNormS[part]

        # Si l'objectif est atteint, on passe au suivant :
        if(LA.norm(PosTemp[part]-aim[part][IndAim[part]])<RDetect):
            if(IndAim[part]<(len(aim[part])-1)):
                IndAim[part] = IndAim[part]+1

    # # # # # Fin des modifications dues aux collisions # # # # #

    position = PosTemp          # Fin d'une étape, ajout des nouvelles positions

    # # # # # Affichage à chaque étape avec Matplotlib # # # # #

    ax.quiver(position[:,0],position[:,1]+epsilon,direNorm[:,0]*vitesse[:,0],direNorm[:,
1]*vitesse[:,0],color="g",scale=0.5,width=0.002)
    ax.quiver(position[:,0],position[:,1]-epsilon,PastVit[:,0],PastVit[:,
1],color="purple",scale=0.5,width=0.002)
    ax.quiver(LineList[:,0,0],LineList[:,0,1],LineList[:,1,0]-LineList[:,0,0],LineList[:,
1,1]-LineList[:,0,1],scale=1,scale_units='x',width=0.005,headaxislength=0,headlength=0)

    ax.axis(frontieres)
    ax.axis('off')
    fig.set_size_inches(windowsize)

    # # # Enregistrement des étapes en format JPEG # # #
```

Annexes :

```
    # # # Enregistrement des étapes en format JPEG # # #

    if(SaveFile & ((t % pasExp)==0)):  #On enregistre un certain nombre d'images, pas
toutes
        plt.savefig(Path+filename+str(Ic)+".jpeg",bbox_inches='tight')
        Ic+=1
    t+=1

    plt.show()
    plt.pause(0.001)
    ax.cla()

# # # # # Fin de la procédure # # # # #
```