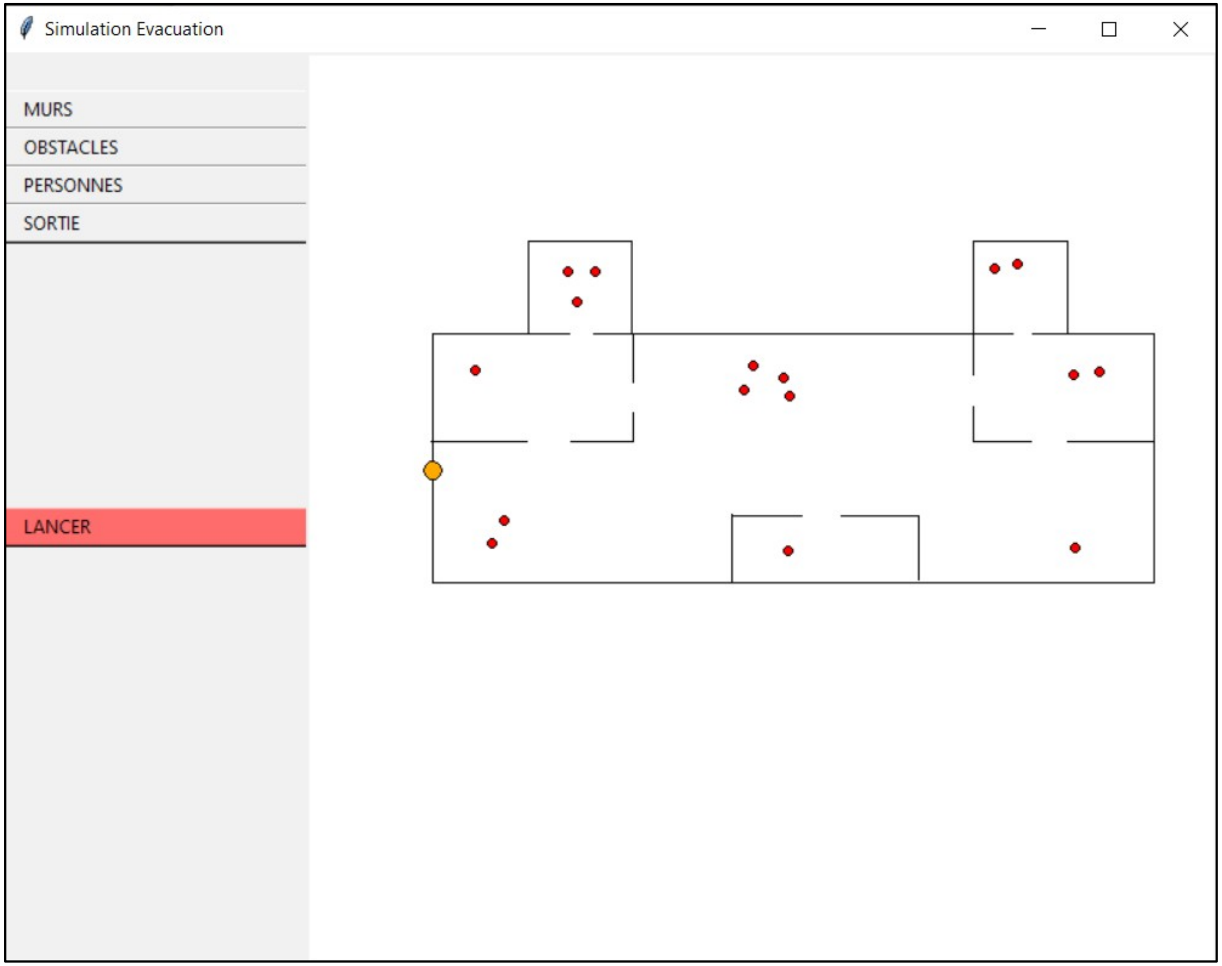
Modélisation et étude du mouvement de foule lors d'une évacuation d'un bâtiment dans le cas d'une situation d'urgence

Félix FOUCHER DE BRANDOIS, numéro de candidat : 45389
En collaboration avec Léna LACOMME, numéro de candidat : 21940

Session 2021-2022

Objectifs et enjeux



Problématique

Quels sont les facteurs sur lesquels agir afin d'optimiser l'évacuation d'une foule ?

Sommaire

- I – Présentation du simulateur
- II – Approche générale :

  → Modèle Helbing

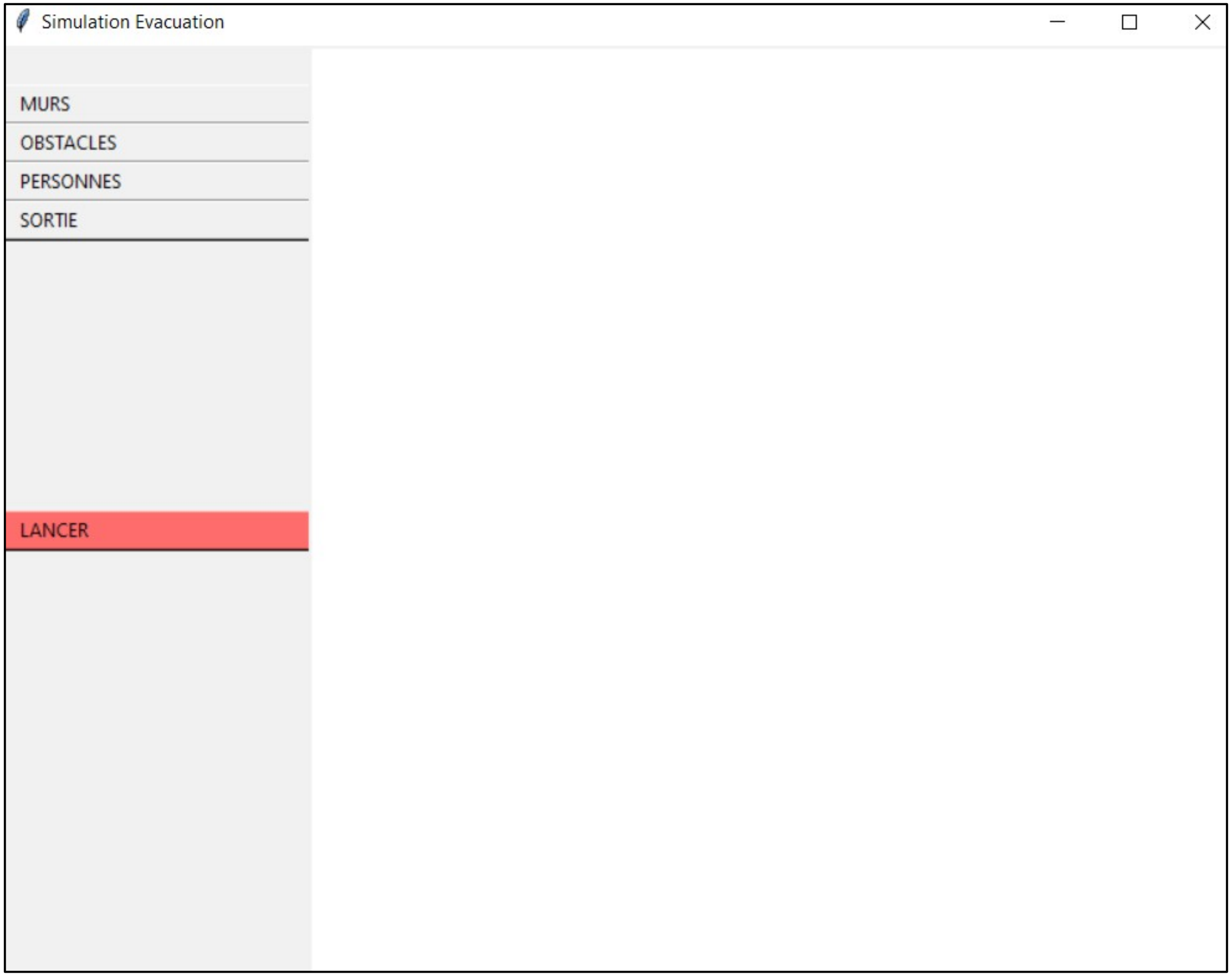
  → Essais avec le simulateur et mise en évidence des lacunes du principe
- III – Modèle CEPABS :

  → Effets pris en compte

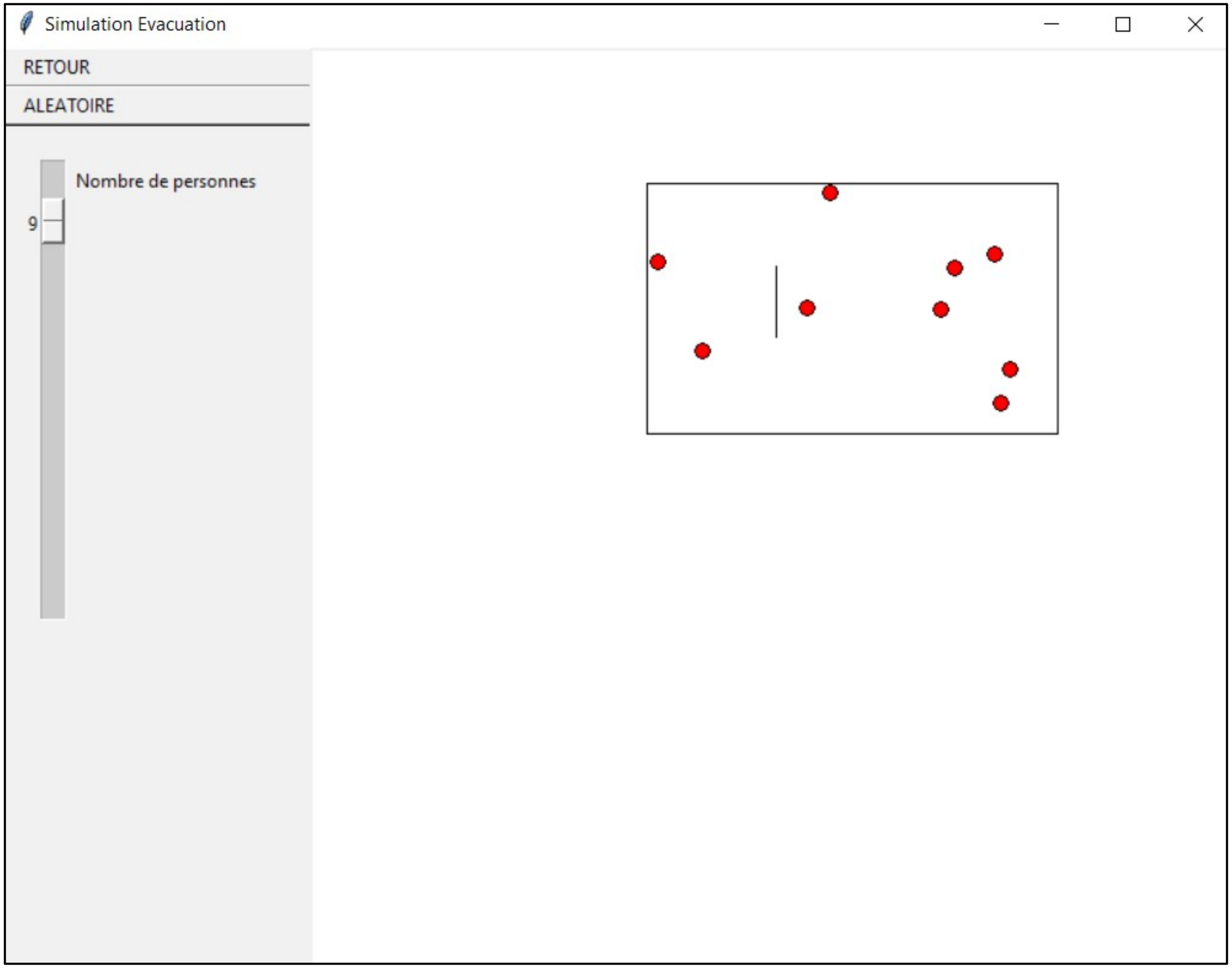
  → Forces considérées
- IV – Expérimentations effectuées

  → Facteurs pris en compte et leurs effets sur la rapidité d'un évacuation
- V – Bilan

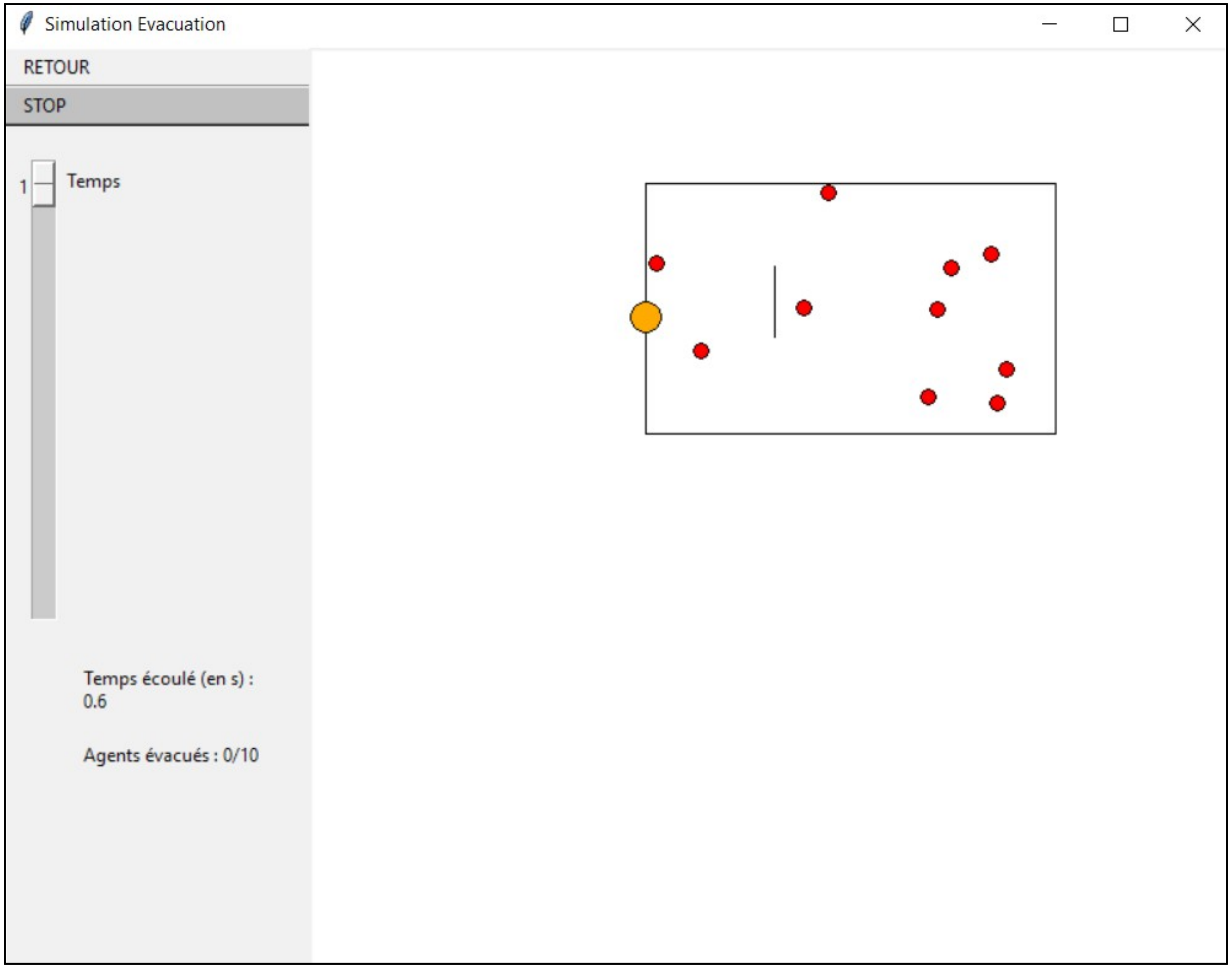
I – Présentation du simulateur



I – Présentation du simulateur

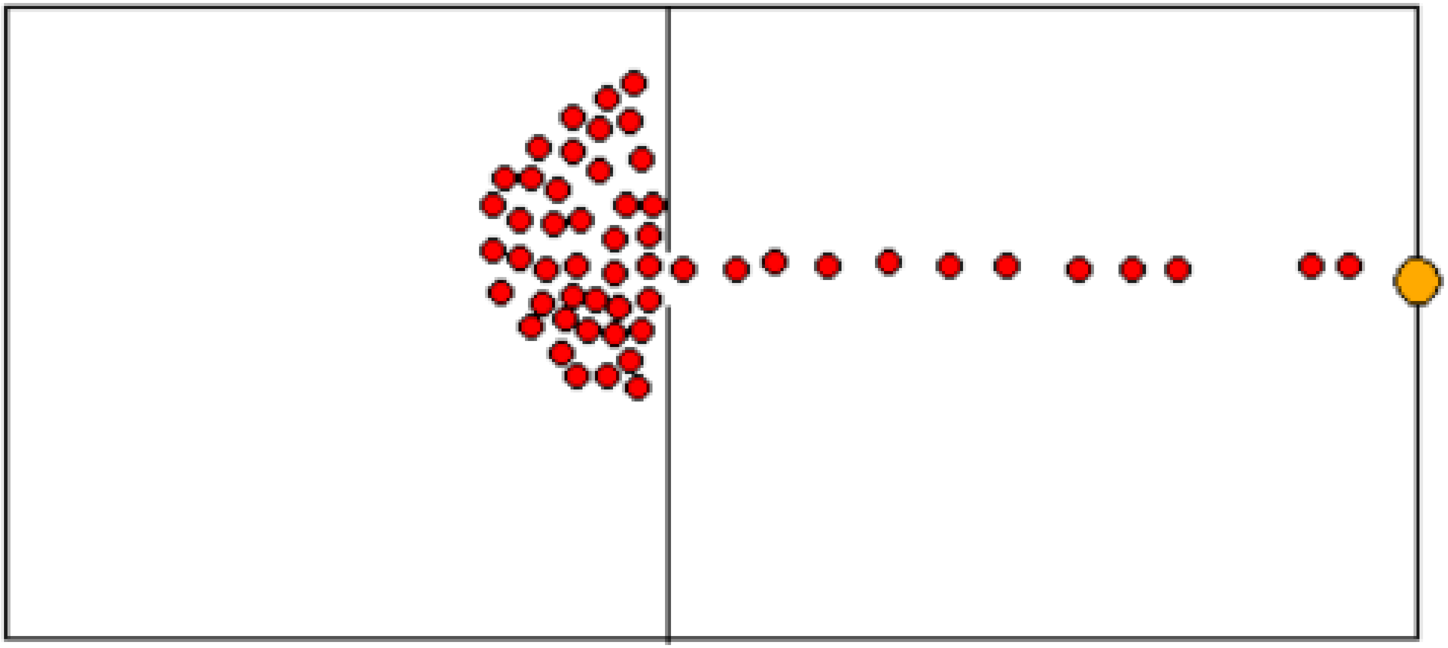


I – Présentation du simulateur

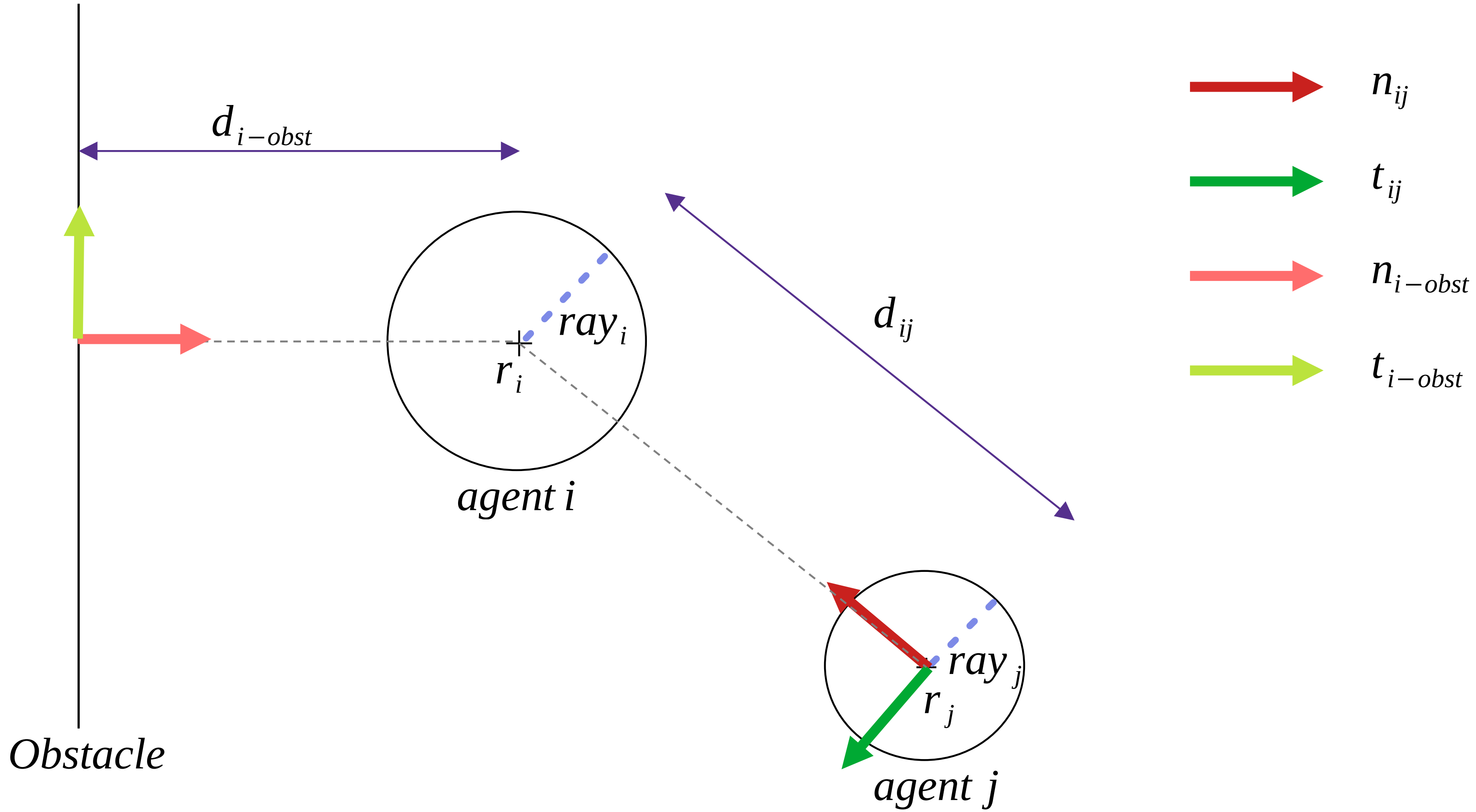


II - Approche générale

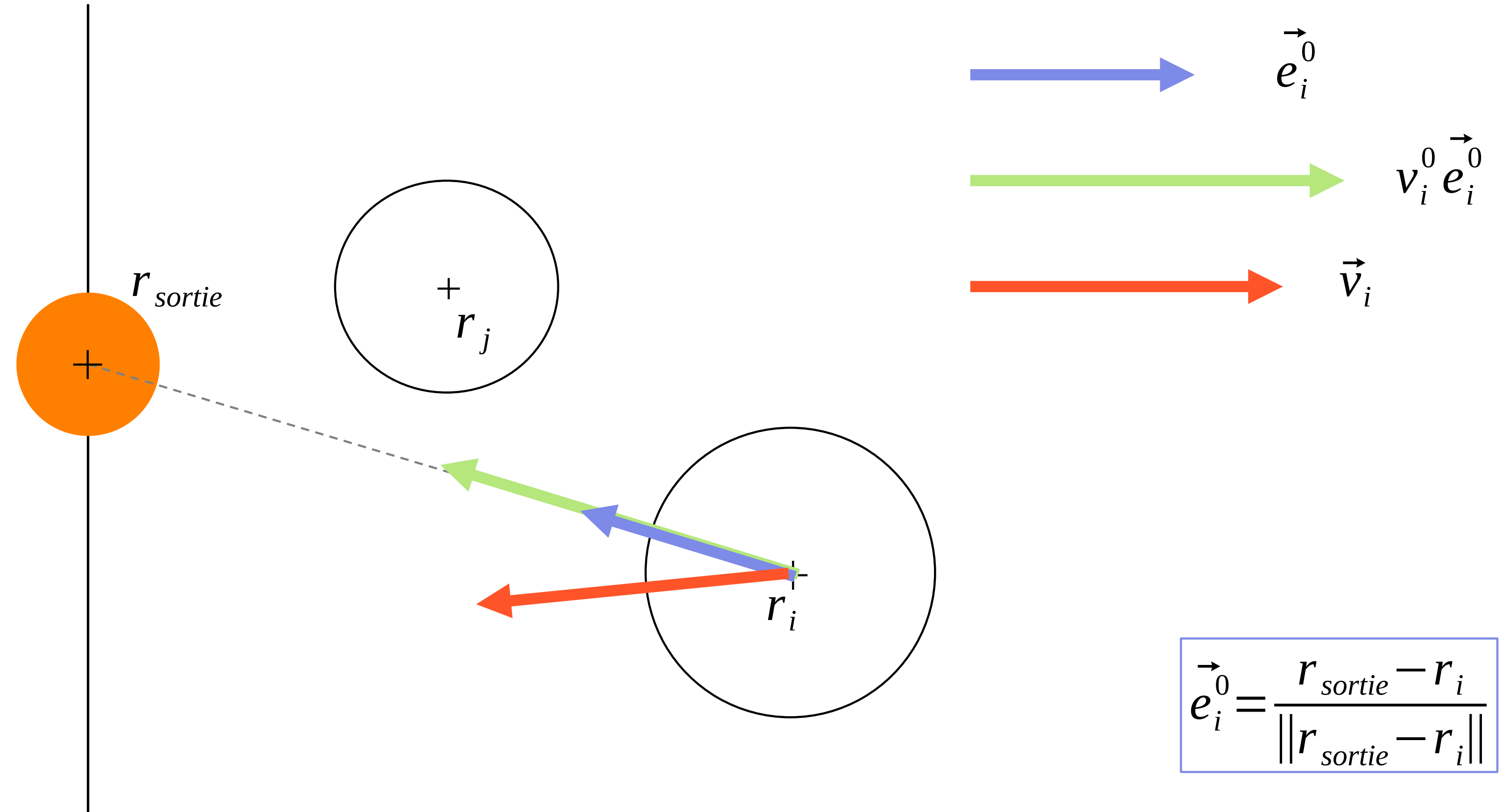
Modélisation Microscopique



Représentation des agents et des murs



Vitesse et direction désirées



$$\vec{e}_i^0 = \frac{r_{sortie} - r_i}{\| r_{sortie} - r_i \|}$$

$$v_i^0 = 1.5\, m.s^{-1}$$

Modèle Helbing : SFM

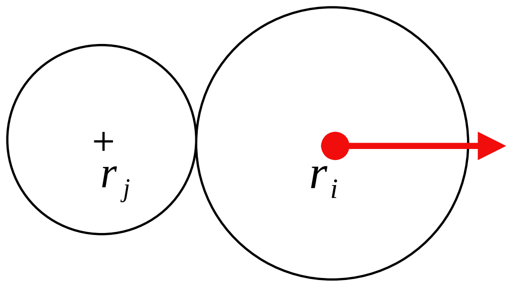
Social Force Model

$$m_i \vec{a}_i = \sum_{j \neq i} \vec{f}_{j \to i} + \sum_{obst} \vec{f}_{obst \to i} + m_i \frac{1}{\tau_i} \left(v_i^0 \vec{e}_i^0 - \vec{v}_i \right)$$

Avec :

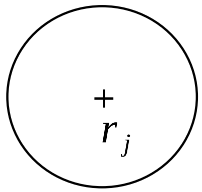
- m_i La masse de l'agent i
- $\vec{a}_i$ L'accélération de l'agent i
- $\vec{f}_{j \to i}$ Les forces exercées par les autres agents
- $\vec{f}_{obst \to i}$ Les forces exercées par les obstacles
- $\tau_i = 0.5\, s$ Le temps de réaction

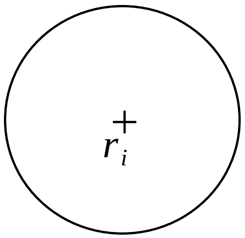
Force entre deux agents



Composante normale pour repousser

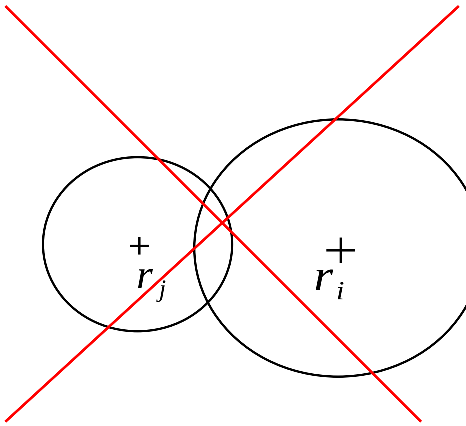
$$\boxed{\vec{f}_{j \to i} = A_i h_1(r_i, r_j, \vec{v}_i, \vec{v}_j)\vec{n}_{ij}}$$





Pas d'interactions

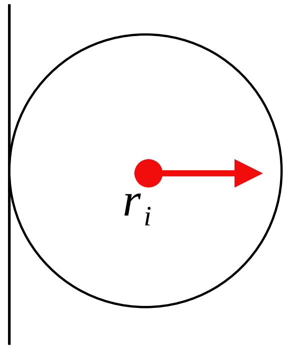
$$\vec{f}_{j \to i} = \vec{0}$$



Situation à éviter

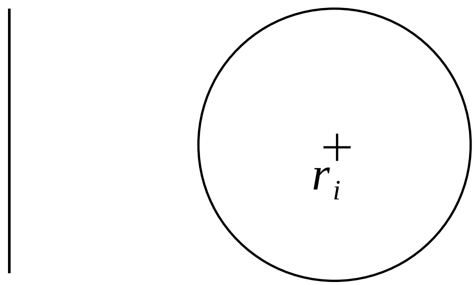
$$\boxed{A_i = 2 * 10^3\, N}$$

Force entre un obstacle et l’agent



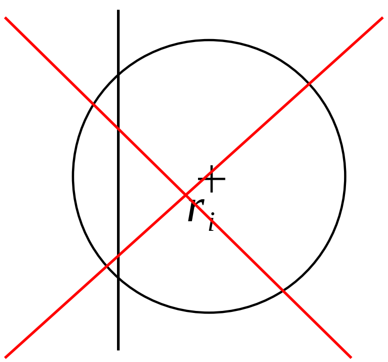
Composante normale pour repousser

$$\boxed{\vec{f}_{obst \to i} = A_i h_2(r_i, \vec{v}_i)\, \vec{n}_{i-obst}}$$



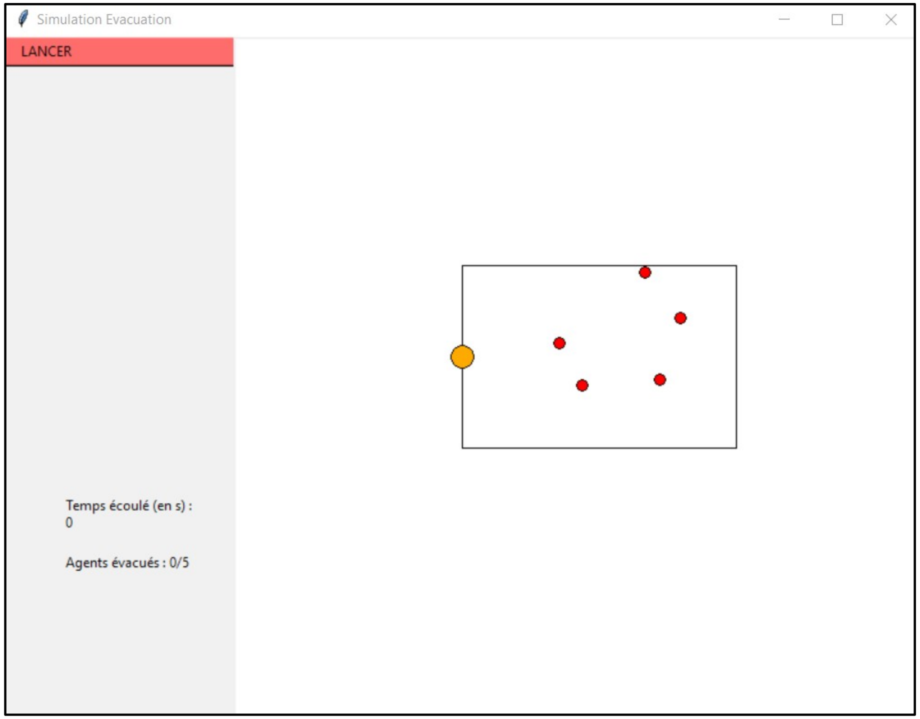
Pas d’interactions

$$\vec{f}_{obst \to i} = \vec{0}$$

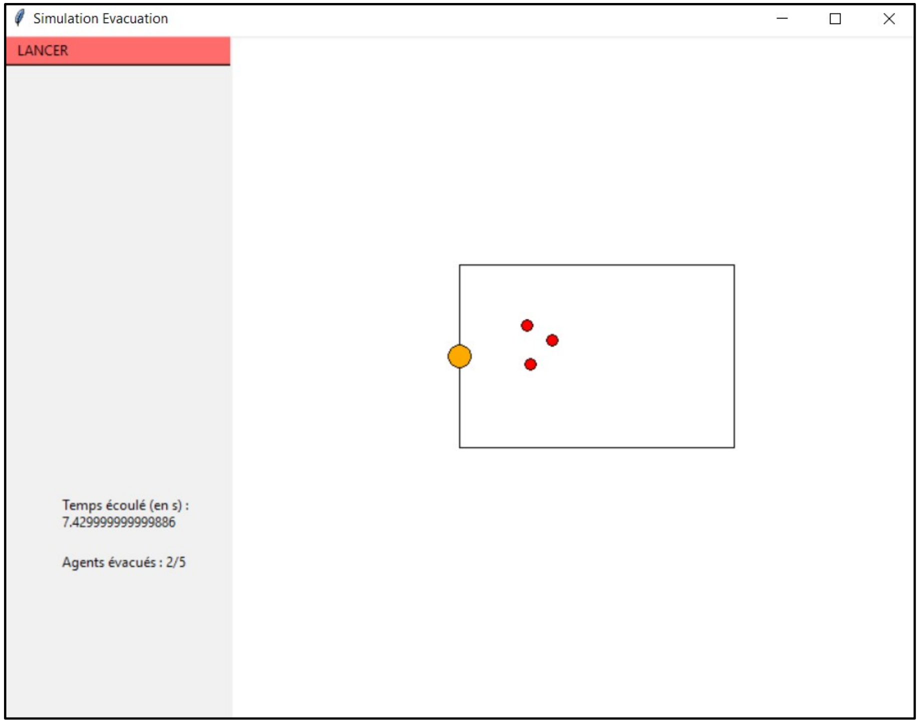


Situation à éviter

Essais à partir du modèle

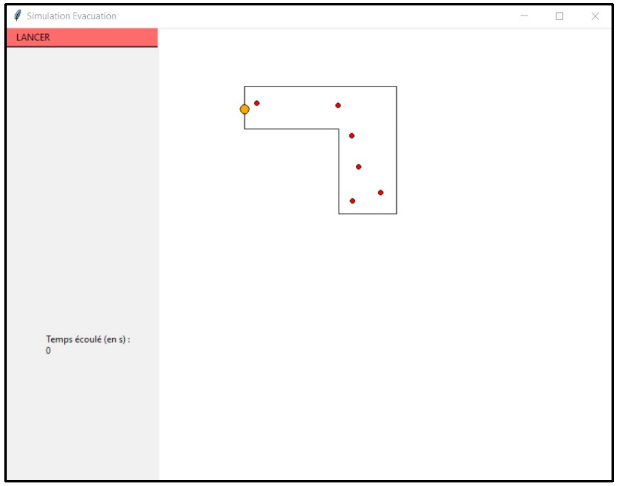


1

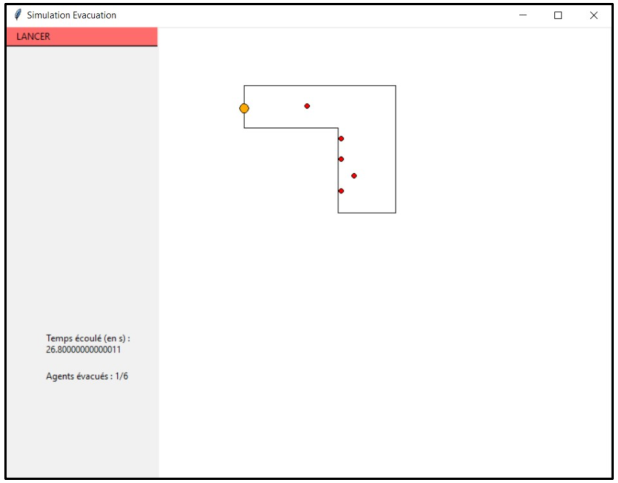


2

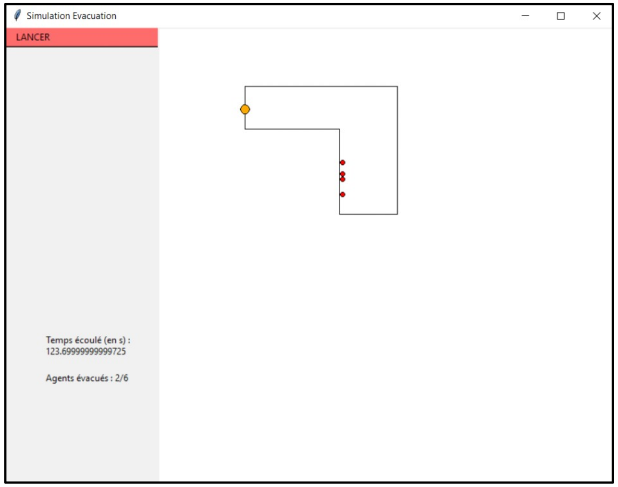
Problèmes physiques observés



1



2



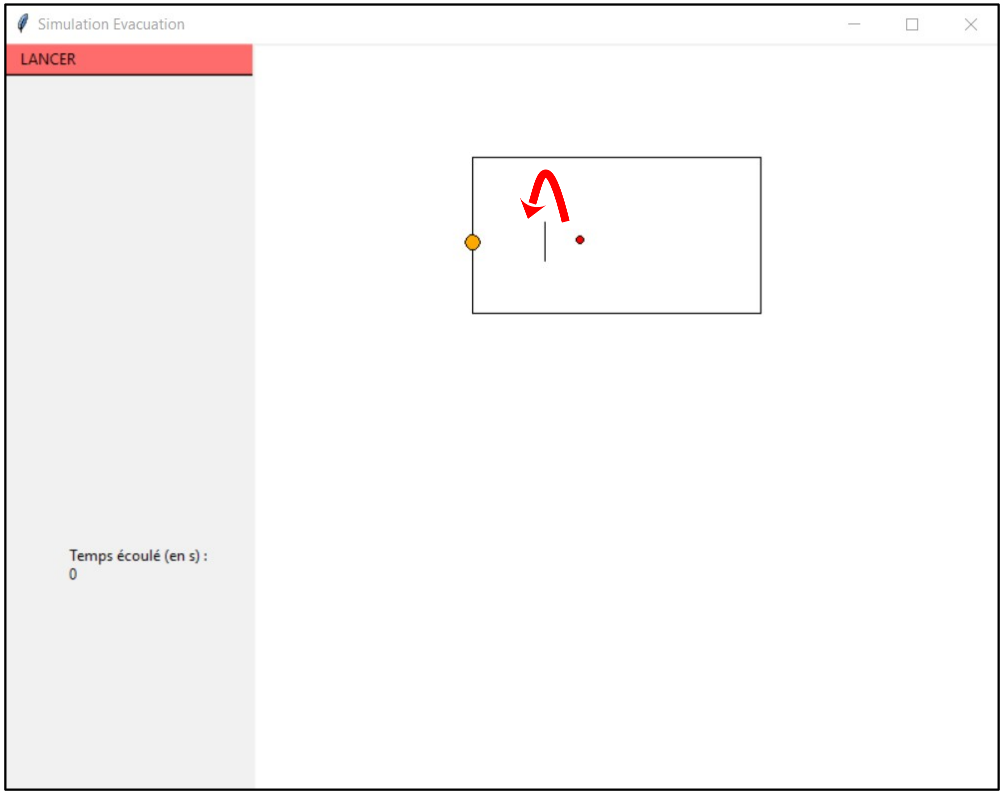
3

II – Modèle CEPABS

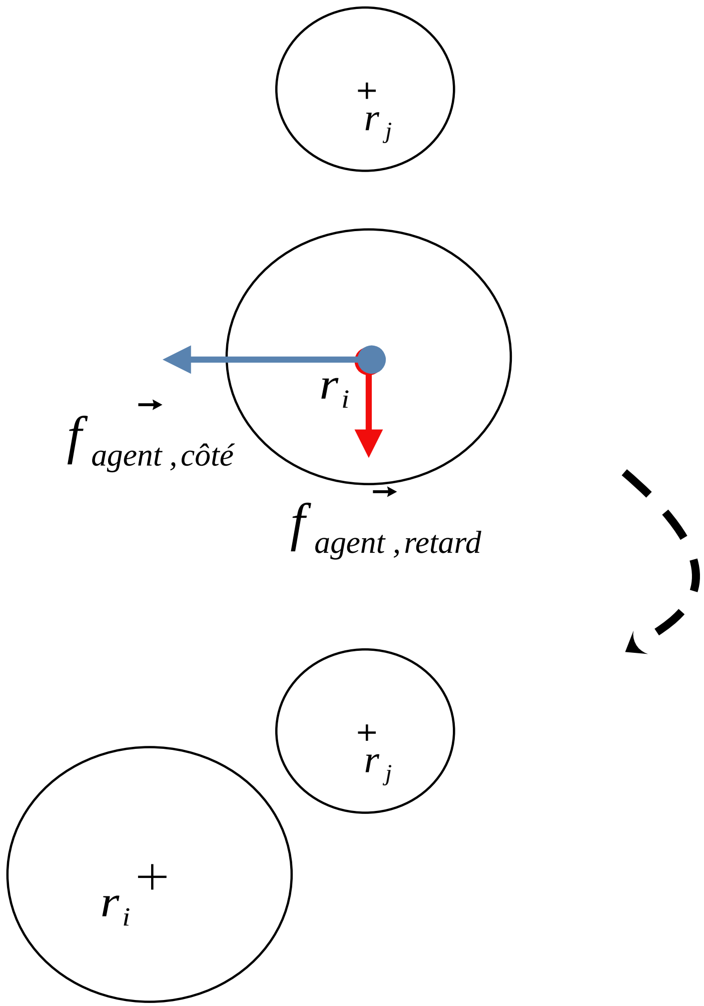
Crowd Evacuation Model Agent Building Simulation

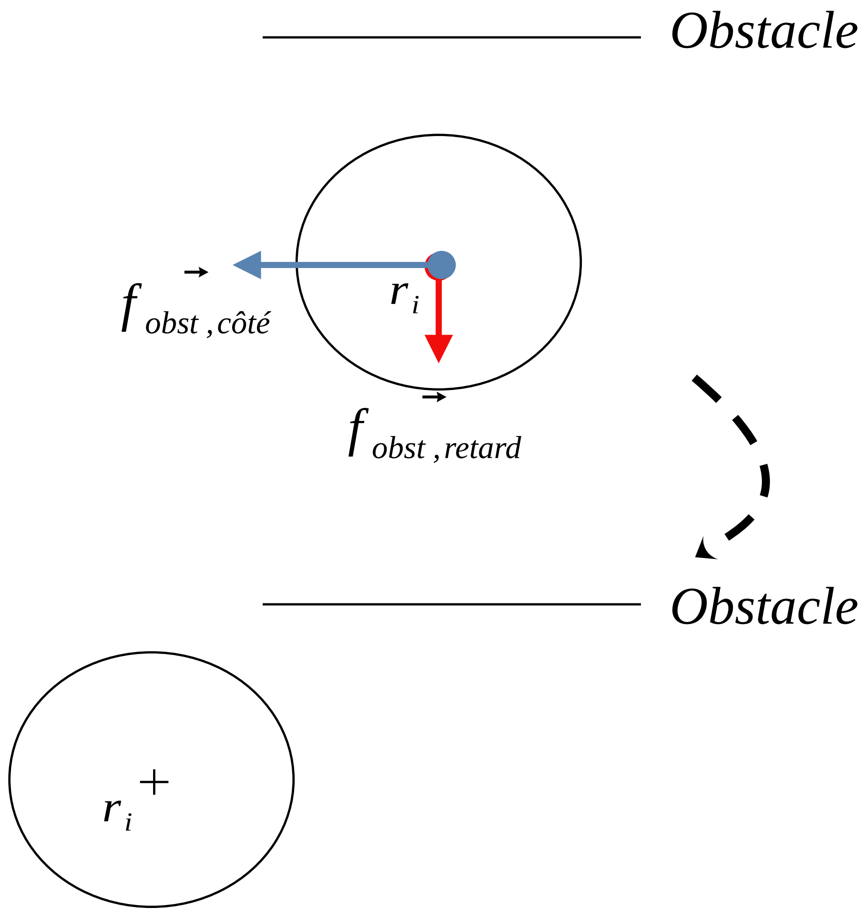
- **Permet de simuler des déplacements de foule plus complexes**
- **Plus approprié pour représenter des foules**

→ Il prends en compte les obstacles



II – Modèle CEPABS





Obstacles

Forces qui ralentissent l'agent :

$$\vec{f}_{obst,retard} = \gamma\, A_{obst,retard}\, h_3(r_i, \vec{v}_i)\, \vec{n}_{i-obst}$$

$$\vec{f}_{agent,retard} = \gamma\, A_{agent,retard}\, h_4(r_i, \vec{v}_i)\, \vec{n}_{ij}$$

$$\gamma = \begin{cases} 1\ si\ l'obstacle\ est\ devant \\ 0\ sinon \end{cases}$$

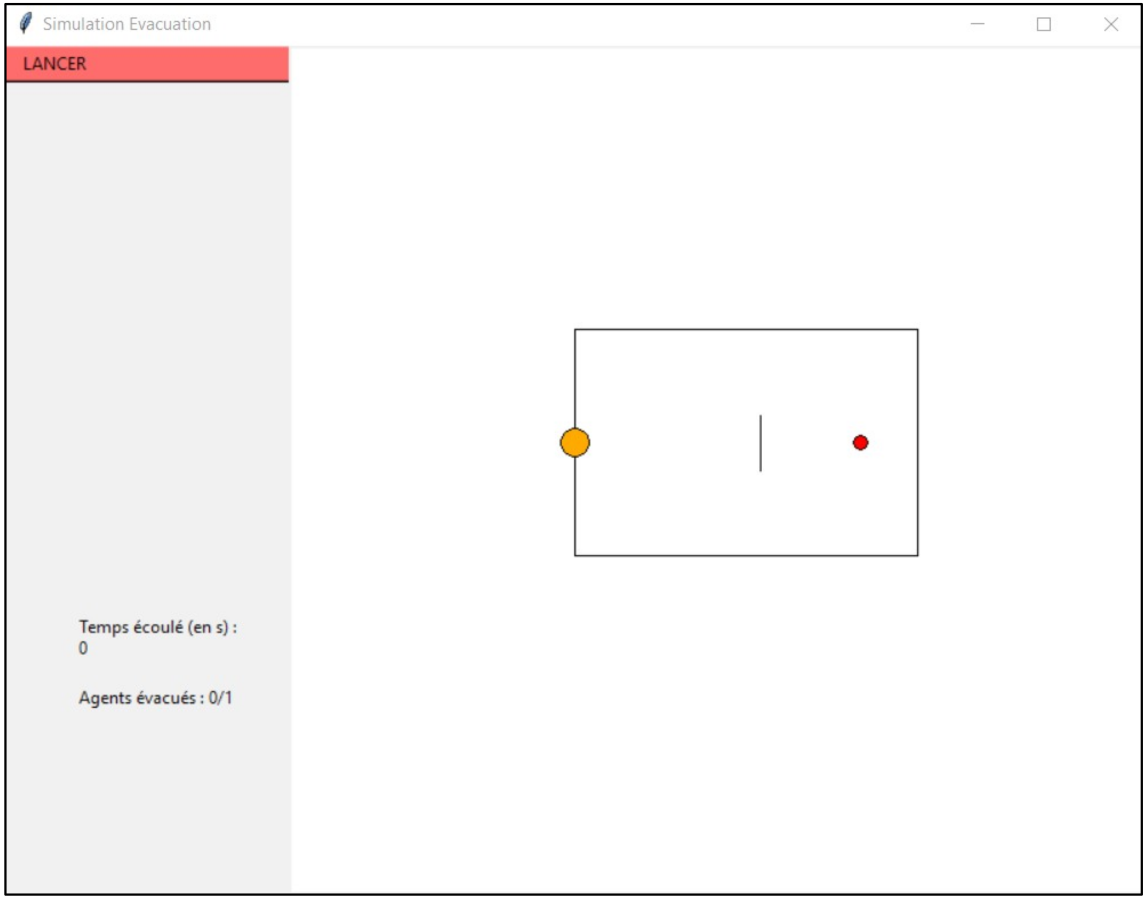
Forces qui obligent l'agent à se déplacer sur le côté :

$$\vec{f}_{obst,côté} = \gamma\, A_{obst,côté}\, h_5(r_i, \vec{v}_i)\, \vec{t}_{i-obst}$$

$$\vec{f}_{agent,côté} = \gamma\, A_{agent,côté}\, h_6(r_i, \vec{v}_i)\, \vec{t}_{ij}$$

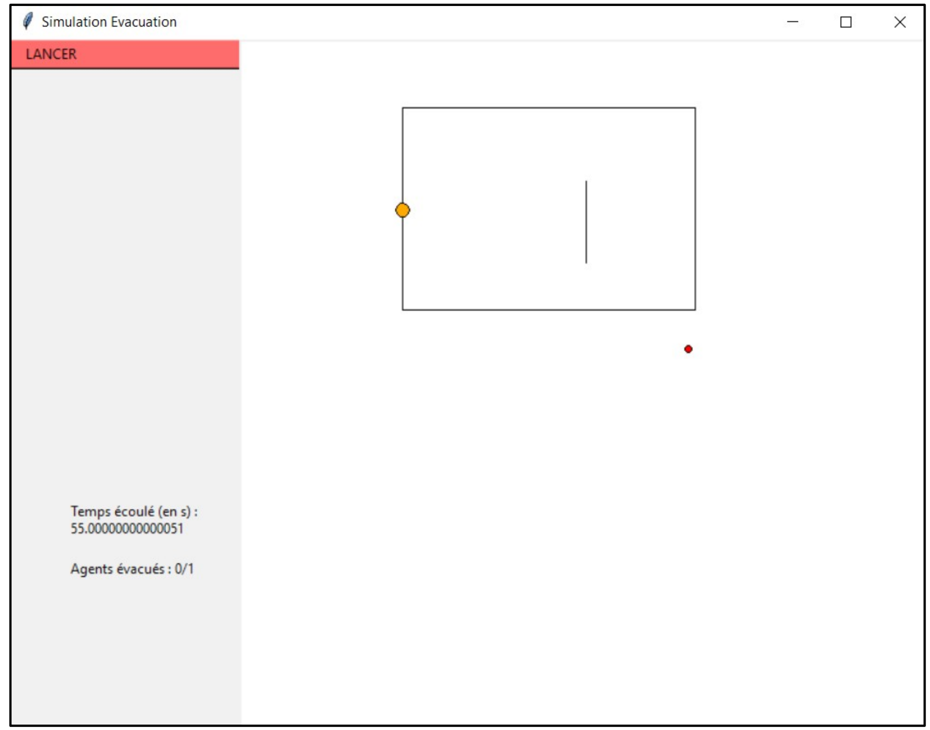
Recherche des constantes

Exemple sur $A_{obst,retard}$:

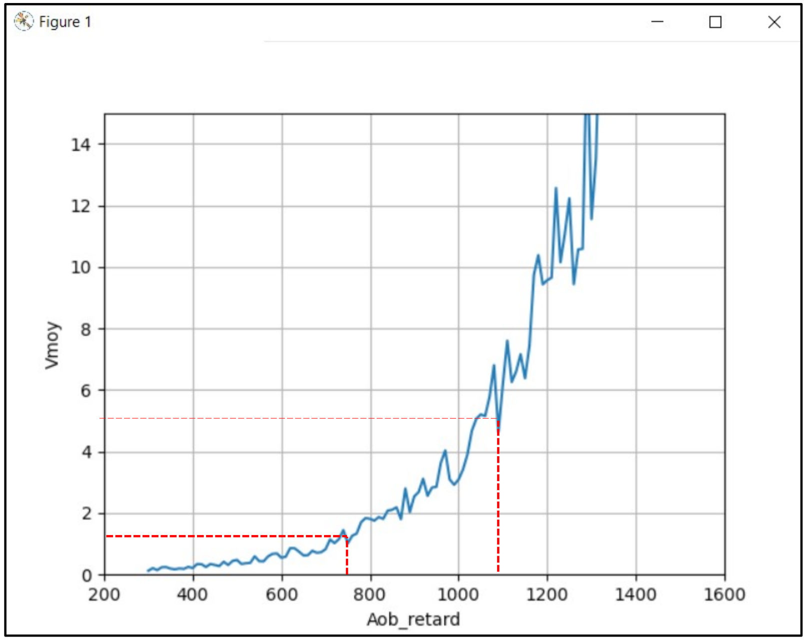


Recherche des constantes

Exemple sur $A_{obst,retard}$:



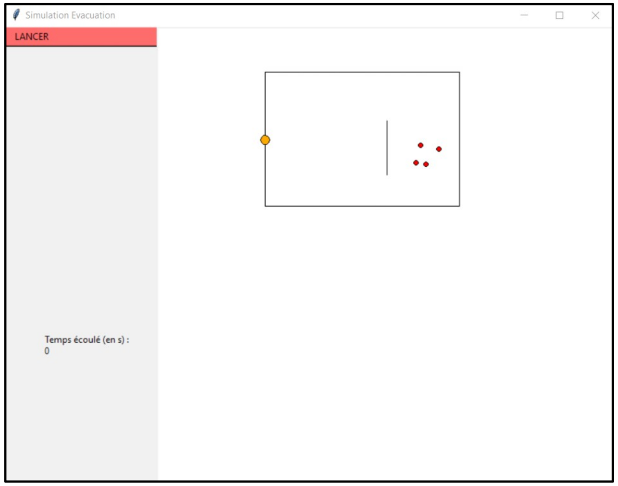
Recherche des constantes

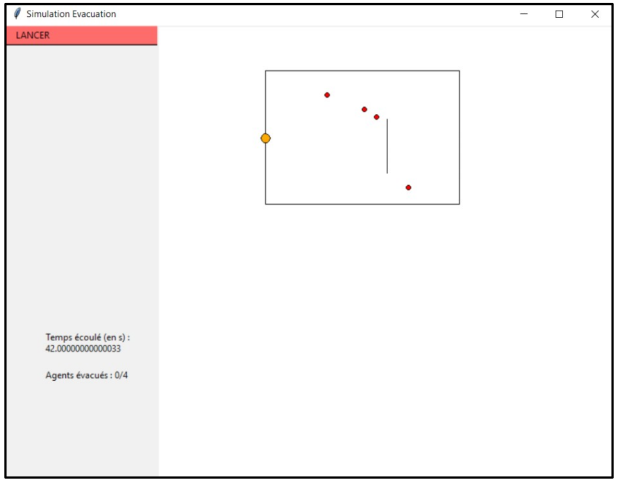


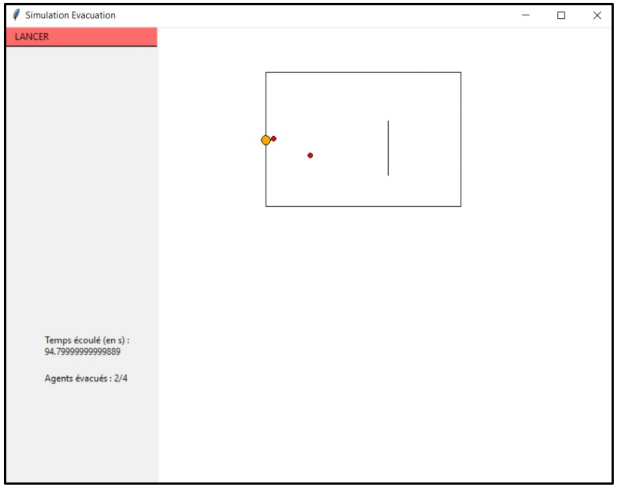
Vitesse moyenne de l'agent en fonction de A

On choisit donc : $A_{obst,retard} = 900\, N$

Essais avec le simulateur





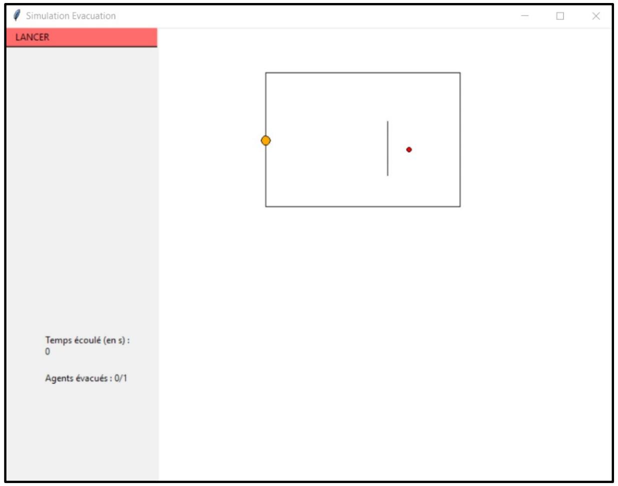


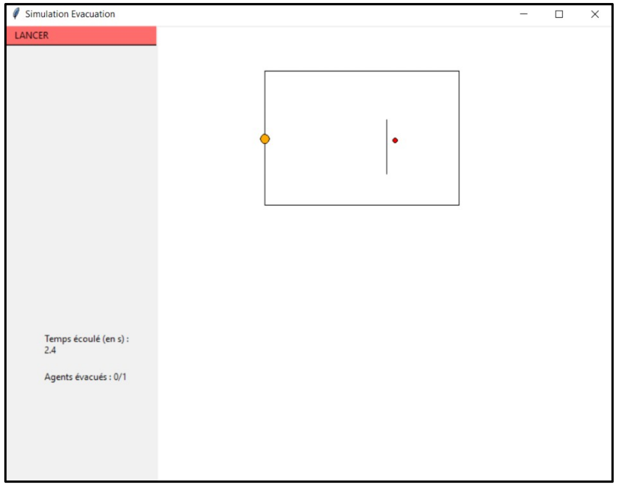
1

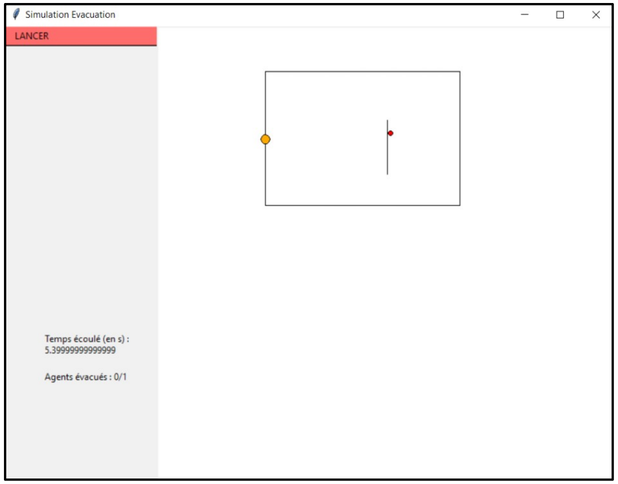
2

3

Comportement inattendu



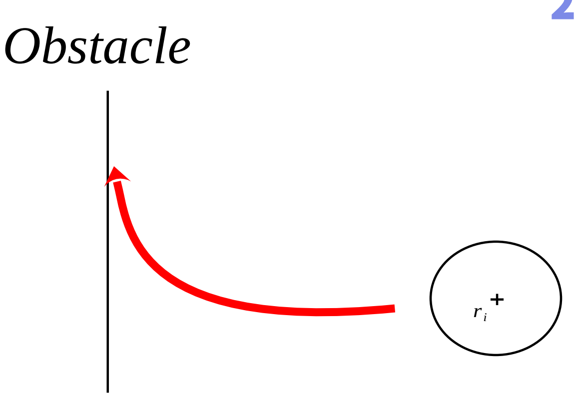




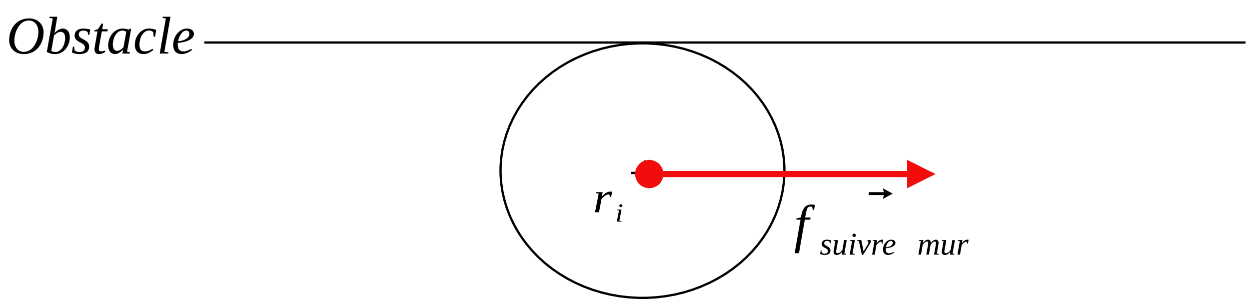
1

2

3



Solution



Ajustements

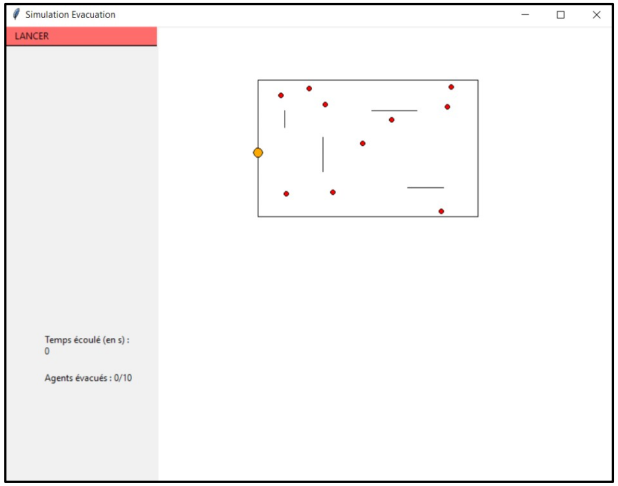
Force qui aide l'agent à contourner l'obstacle :

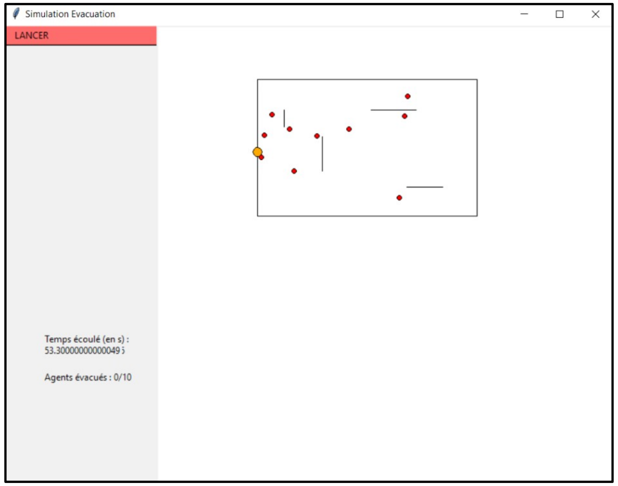
$$\overrightarrow{f_{suivre\ \ mur}} = A_{suivre\ \ mur}\, h_7(r_i, \vec{v}_i)\, \overrightarrow{t_{i-obst}}$$

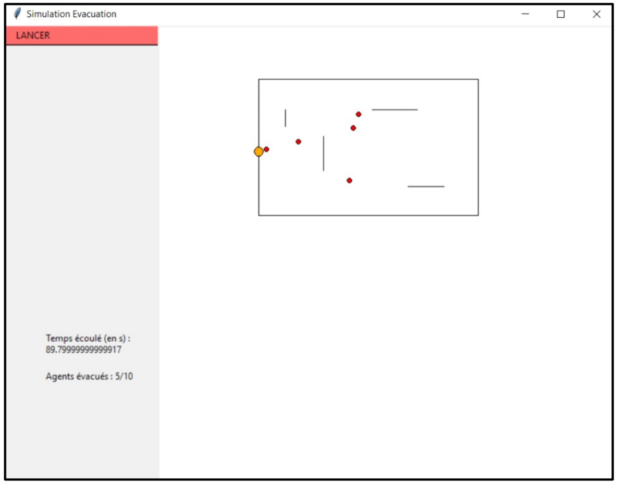
Force qui réduit une trop grande variation de vitesse (~frottements) :

$$\overrightarrow{f_{v,amortissement}} = -\sqrt{\|\vec{v}_i\|}\,\vec{v}_i$$

Résultats sur le simulateur







1 2 3

Ajout du modèle comportemental

Panique :

$$\boxed{\dot{p}_i = c_1(\bar{p}_j - p_i) \ + \ c_2(v_i^0 - \|\vec{v}_i\|) \ + \ c_3 Q \ + \ c_4}$$

$$p_i \in [0, 100]$$

1. Panique des agents autour
2. Frustration provenant de la différence entre la vitesse désirée et la vitesse actuelle
3. Solitude
4. Frustration initiale

Conséquence sur la vitesse désirée :

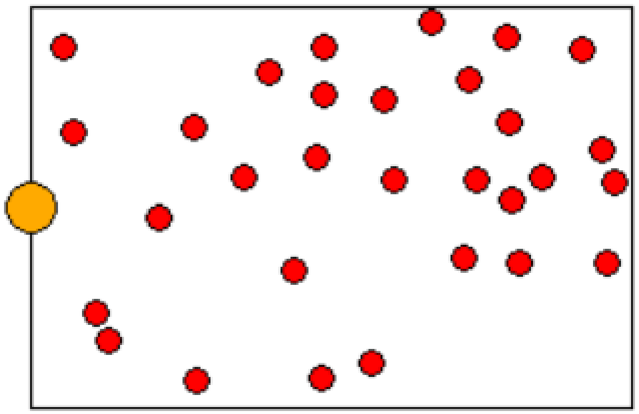
$$\boxed{v_i^0(p_i) = v_{i,initial}^0 \left(1 \ + \ \frac{p_i}{p_{max}}\right)}$$

$$p_{max} = 100$$

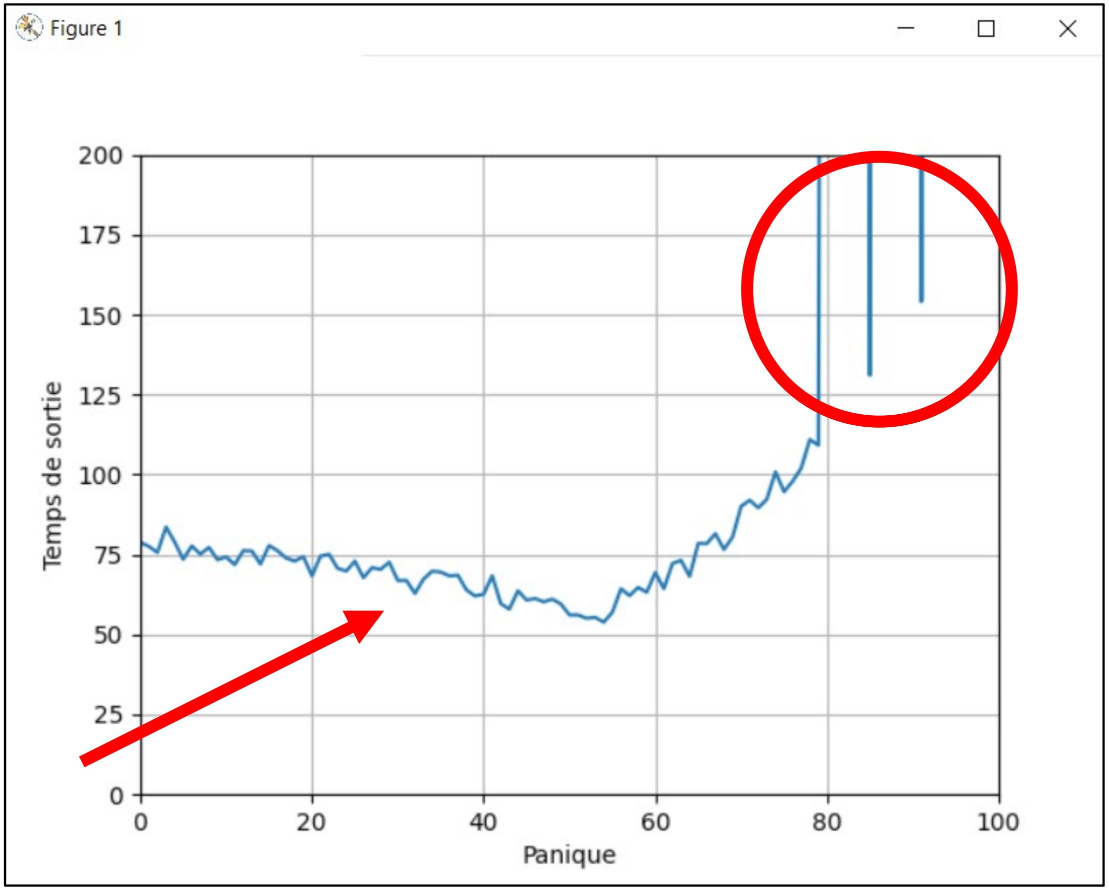
III – Influence des différents paramètres

- Niveau de panique
- Disposition des obstacles

Niveau de panique



On fixe le niveau de panique et on mesure le temps de sortie



Durée de l'évacuation en fonction du niveau de panique

Annexe 10

<u>Non respect des distanciations sociales :</u>

On veut :

$$A_i(p_i) = a * p_i + b$$

$$A_i(0) = 2*10^3$$

$$A_i(100) = \frac{1}{10} A_i(0) = 2*10^2$$

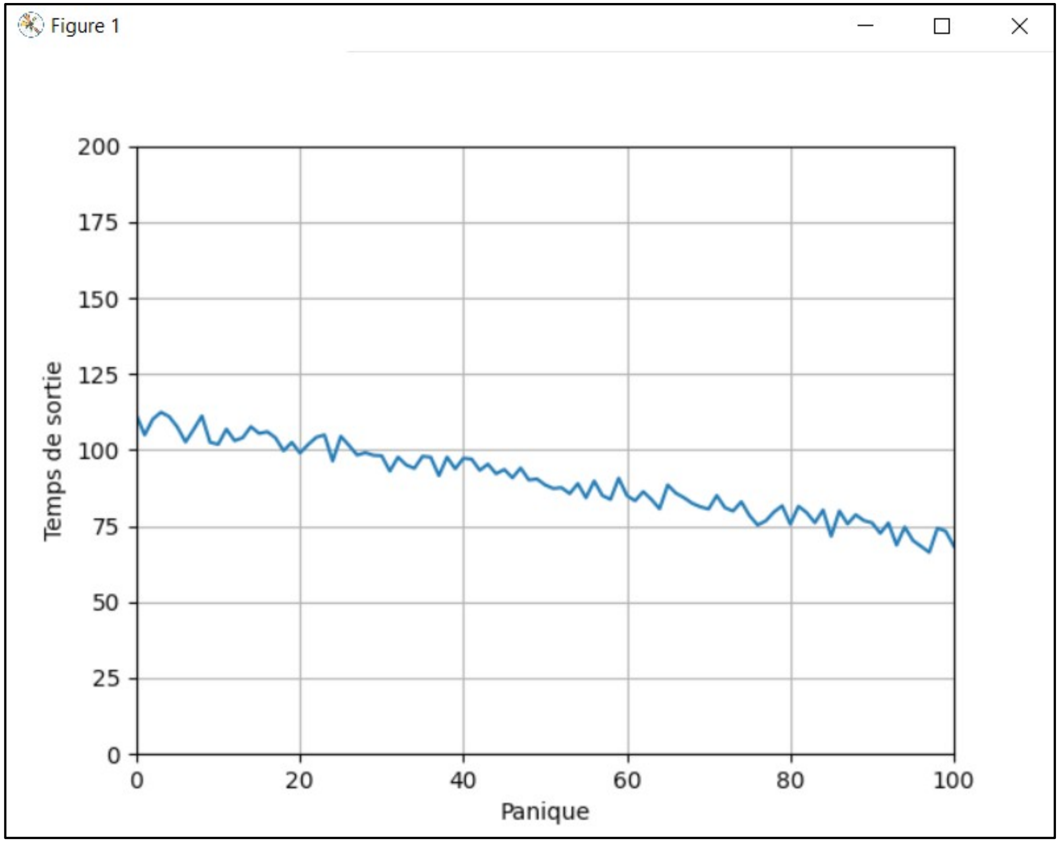
$$\boxed{A_i(p_i) = -18\, p_i + 2*10^3}$$

<u>Forces modifiées :</u>

$$\vec{f}_{j \to i} = {\color{red}A_i}\, h_1(r_i, r_j, \vec{v}_i, \vec{v}_j)\, \vec{n}_{ij}$$

$$\vec{f}_{obst \to i} = {\color{red}A_i}\, h_2(r_i, \vec{v}_i)\, \vec{n}_{i-obst}$$

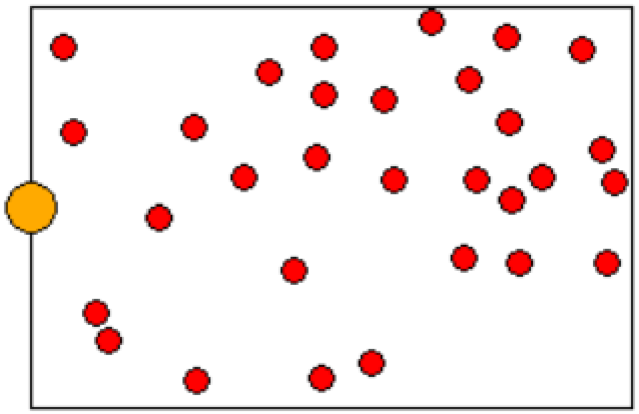
Niveau de panique



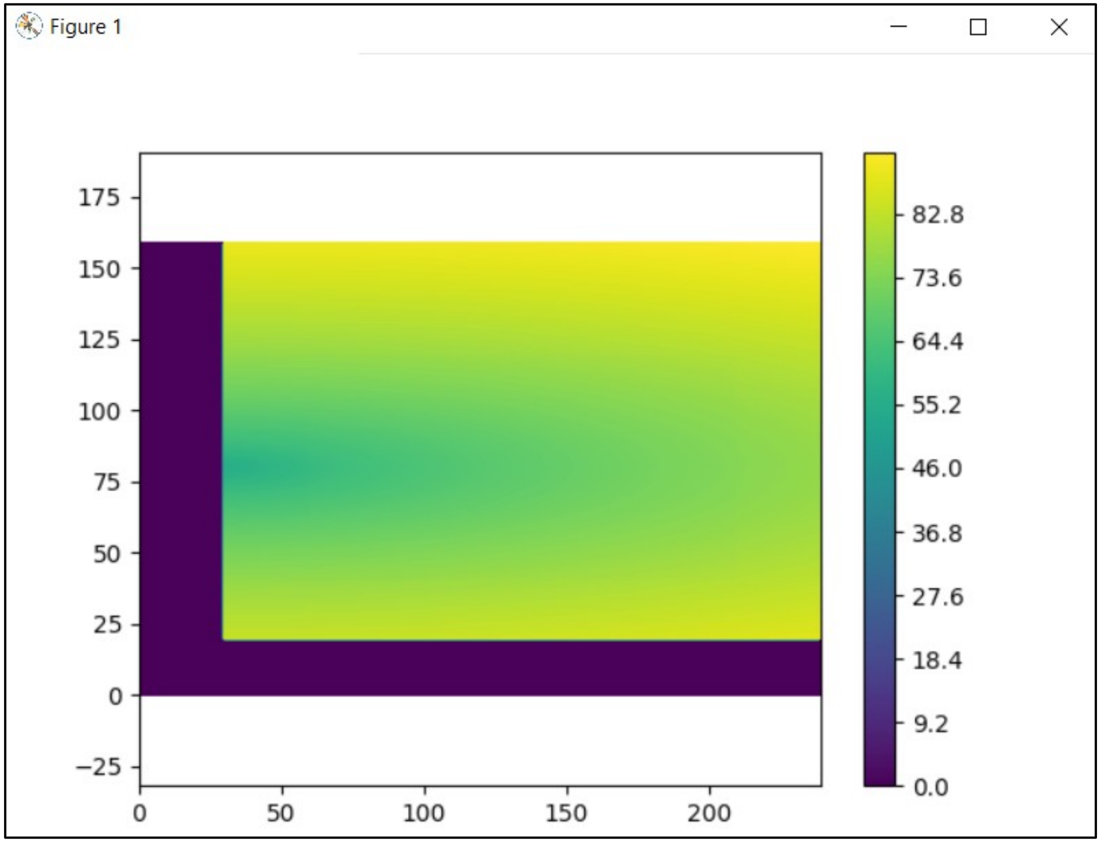
La panique améliore le temps de sortie

Durée de l'évacuation en fonction du niveau de panique

Disposition des obstacles



Un obstacle devant la sortie diminue la durée de l'évacuation



Durée de l'évacuation en fonction de l'emplacement de l'obstacle

IV – Bilan

Objectifs :

- Construire un logiciel permettant de simuler de manière adéquate les évacuations

- Étudier les facteurs sur lesquels agir afin de diminuer le temps de sortie

Panique :

La panique améliore le temps de sortie

Impossible de simuler les comportements incohérents

Obstacles :

Un obstacle devant la sortie diminue la durée de l'évacuation

FIN

Annexe 1

Simulateur :

```python
class Animation(tk.Tk):

    def __init__(self):
        tk.Tk.__init__(self)
        self.tool = tk.Frame(self, width = 200, height = 600)
        self.tool.pack(side = 'left')

        self.graph = tk.Frame(self, width = 600, height = 600, bg = 'white')
        self.graph.pack(side = 'right')

        self.can = tk.Canvas(self.graph,width = 600, height = 600, bg = 'white')
        self.can.pack()

        tk.Button(self.tool, text = "   LANCER", command = self.debut, width = 30, anchor = 'w', bg = '#ff6666', activebackground = '#bdbdbd').place(x = 0, y = 0)



    def debut(self):
        self.supprimer_parametres()
        self.menu()

    ##MENU##

    def menu(self):
        tk.Button(self.tool, text = "   MURS", command = self.placer_murs, width = 30, anchor = 'w', bg = '#f0f0f0', activebackground = '#bdbdbd').place(x = 0, y = 25)

        tk.Button(self.tool, text = "   OBSTACLES", command = self.placer_obstacle, width = 30, anchor = 'w', bg = '#f0f0f0', activebackground = '#bdbdbd').place(x = 0, y = 50)

        tk.Button(self.tool, text = "   PERSONNES", command = self.placer_personnes, width = 30, anchor = 'w', bg = '#f0f0f0', activebackground = '#bdbdbd').place(x = 0, y = 75)

        tk.Button(self.tool, text = "   SORTIE", command = self.placer_sortie, width = 30, anchor = 'w', bg = '#f0f0f0', activebackground = '#bdbdbd').place(x = 0, y = 100)


        tk.Button(self.tool, text = "   LANCER", command = self.lancer_simulation, width = 30, anchor = 'w', bg = '#ff6666', activebackground = '#bdbdbd').place(x = 0, y = 300)


    def supprimer_parametres(self):
        L = self.tool.winfo_children()
        for i in L:
            i.destroy()
```

Annexe 1

```python
    ##MURS##

    def placer_murs(self): #Bouton 'nettoyer' pour remettre à zero la pièce
        self.supprimer_parametres()
        self.can.delete('all')
        tk.Button(self.tool, text = "   RETOUR", command = self.retour_murs, width = 30, anchor = 'w', bg = '#f0f0f0', activebackground = '#bdbdbd').place(x = 0, y = 0)

        self.peut_placer_murs = False

        self.murs = np.array([[[0, 0], [0, 0]]])
        self.L_murs_dessins = [0]

        self.can.bind("<Enter>", self.dessiner_ligne_mur)
        self.can.bind("<Motion>", self.dessiner_ligne_mur)
        self.can.bind("<Button-1>", self.cliquer_mur)


    def cliquer_mur(self, event):
        x1, y1 = int(event.x), int(event.y)
        k = False
        if not self.peut_placer_murs:
            self.peut_placer_murs = True
            self.x, self.y = x1, y1

        elif self.peut_placer_murs:
            if abs(self.x - x1) < abs(self.y - y1):
                for i in self.murs:
                    if abs(y1 - i[0, 1]) < 20:
                        self.murs = np.vstack([self.murs,[[[self.x, self.y], [self.x, i[0, 1]]]]])
                        self.y = i[0, 1]
                        k = True
                        break
                if not k:
                    self.murs = np.vstack([self.murs,[[[self.x, self.y], [self.x, y1]]]])
                    self.y = y1
            else :
                for i in self.murs:
                    if abs(x1 - i[0, 0]) < 20:
                        self.murs = np.vstack([self.murs,[[[self.x, self.y], [i[0, 0], self.y]]]])
                        self.x = i[0, 0]
                        k = True
                        break
                if not k:
                    self.murs = np.vstack([self.murs,[[[self.x, self.y], [x1, self.y]]]])
                    self.x = x1

            self.L_murs_dessins.append(0)

            if self.murs[0].all() == 0:
                self.murs = np.delete(self.murs, 0, axis = 0)
```

Annexe 1

```python
        if abs(self.murs[-1, 1, 0] - self.murs[0, 0, 0]) <= rayon and abs(self.murs[-1, 1, 1] - self.murs[0, 0, 1]) <= rayon and len(self.murs)>=2:
            self.peut_placer_murs = False


    #Dessiner les lignes du mur quand la souris bouge
    def dessiner_ligne_mur(self, event):
        x1, y1 = int(event.x), int(event.y)
        if self.peut_placer_murs:
            self.can.delete(self.L_murs_dessins[-1])

            if abs(self.x - x1) < abs(self.y - y1):
                self.L_murs_dessins[-1] = self.can.create_line(self.x, self.y, self.x, y1)
                for i in self.murs:
                    if abs(y1 - i[0, 1]) < 20:
                        self.can.delete(self.L_murs_dessins[-1])
                        self.L_murs_dessins[-1] = self.can.create_line(self.x, self.y, self.x, i[0, 1])

            else :
                self.L_murs_dessins[-1] = self.can.create_line(self.x, self.y, x1, self.y)
                for i in self.murs:
                    if abs(x1 - i[0, 0]) < 20:
                        self.can.delete(self.L_murs_dessins[-1])
                        self.L_murs_dessins[-1] = self.can.create_line(self.x, self.y, i[0, 0], self.y)


    def retour_murs(self):
        self.supprimer_parametres()
        self.can.unbind("<Enter>")
        self.can.unbind("<Motion>")
        self.can.unbind("<Button-1>")
        self.dmin = np.linalg.norm(self.murs[0, 1, :] - self.murs[0, 0, :])
        for i in self.murs:
            d = np.linalg.norm(i[1] - i[0])
            if d < self.dmin :
                self.dmin = d
        self.menu()
```

Annexe 1

```python
    ##OBSTACLES##


    def placer_obstacle(self):
        self.supprimer_parametres()
        tk.Button(self.tool, text = "   RETOUR", command = self.retour_obstacle, width = 30, anchor = 'w', bg = '#f0f0f0',
activebackground = '#bdbdbd').place(x = 0, y = 0)

        self.peut_placer_obstacle = False
        self.L_obstacles_dessins = [0]
        self.obstacles = np.array([[[0, 0], [0, 0]]])

        self.can.bind("<Enter>", self.dessiner_ligne_obstacle)
        self.can.bind("<Motion>", self.dessiner_ligne_obstacle)
        self.can.bind("<Button-1>", self.cliquer_obstacle)


    def cliquer_obstacle(self, event):
        x1, y1 = int(event.x), int(event.y)
        k = False
        if not self.peut_placer_obstacle:
            self.peut_placer_obstacle = True
            for i in self.obstacles:
                if abs(y1 - i[0, 1]) < 20:
                    self.y = i[0, 1]
                    k = True
                    break
            if not k:
                self.y = y1
            k = False
            for i in self.obstacles:
                if abs(x1 - i[0, 0]) < 20:
                    self.x = i[0, 0]
                    k = True
                    break
            if not k:
                self.x = x1

        elif self.peut_placer_obstacle:
            self.peut_placer_obstacle = False

            if abs(self.x - x1) < abs(self.y - y1):
                for i in self.obstacles:
                    if abs(y1 - i[0, 1]) < 20:
                        self.obstacles = np.vstack([self.obstacles,[[[self.x, self.y], [self.x, i[0, 1]]]]])
                        k = True
                        break
                if not k:
                    self.obstacles = np.vstack([self.obstacles,[[[self.x, self.y], [self.x, y1]]]])
```

Annexe 1

```python
            else :
                for i in self.obstacles:
                    if abs(x1 - i[0, 0]) < 20:
                        self.obstacles = np.vstack([self.obstacles,[[[self.x, self.y], [i[0, 0], self.y]]]])
                        k = True
                        break
                if not k:
                    self.obstacles = np.vstack([self.obstacles,[[[self.x, self.y], [x1, self.y]]]])

        self.L_obstacles_dessins.append(0)

        if self.obstacles[0].all() == 0:
            self.obstacles = np.delete(self.obstacles, 0, axis = 0)


    def dessiner_ligne_obstacle(self, event):
        if self.peut_placer_obstacle:
            x1, y1 = int(event.x), int(event.y)
            self.can.delete(self.L_obstacles_dessins[-1])

            if abs(self.x - x1) < abs(self.y - y1):
                self.L_obstacles_dessins[-1] = self.can.create_line(self.x, self.y, self.x, y1)
                for i in self.obstacles:
                    if abs(y1 - i[0, 1]) < 20:
                        self.can.delete(self.L_obstacles_dessins[-1])
                        self.L_obstacles_dessins[-1] = self.can.create_line(self.x, self.y, self.x, i[0, 1])

            else :
                self.L_obstacles_dessins[-1] = self.can.create_line(self.x, self.y, x1, self.y)
                for i in self.obstacles:
                    if abs(x1 - i[0, 0]) < 20:
                        self.can.delete(self.L_obstacles_dessins[-1])
                        self.L_obstacles_dessins[-1] = self.can.create_line(self.x, self.y, i[0, 0], self.y)


    def retour_obstacle(self):
        self.can.unbind("<Enter>")
        self.can.unbind("<Motion>")
        self.can.unbind("<Button-1>")
        self.supprimer_parametres()
        self.menu()
```

Annexe 1

```
    ##PERSONNES##

#Placer les personnes "en gros" -> Pas précisément
#Curseur "intensité des interactions, ...

    def placer_personnes(self):
        self.supprimer_parametres()
        tk.Button(self.tool, text = "   RETOUR", command = self.retour_personnes, width = 30, anchor = 'w', bg = '#f0f0f0',
activebackground = '#bdbdbd').place(x = 0, y = 0)

        tk.Button(self.tool, text = "   ALEATOIRE", command = self.placer_personnes_aleatoire, width = 30, anchor = 'w', bg =
'#f0f0f0', activebackground = '#bdbdbd').place(x = 0, y = 25)

        self.nb_pers = tk.IntVar()
        tk.Scale(self.tool, orient = "vertical", label = "Nombre de personnes", length = 300, from_ = 0, to = 100, variable =
self.nb_pers).place(x = 0, y = 70)

        self.N = self.nb_pers.get()
        self.r = np.zeros((self.N, 2))

        self.L_personnes_dessin = []

        self.can.bind('<Up>', self.pers_moins)
        self.can.bind('<Down>', self.pers_plus)
        self.can.bind('<Button-1>', self.placer_personnes_manuel)


    def pers_moins(self, event):
        x = self.nb_pers.get()
        self.nb_pers.set(x-1)

    def pers_plus(self, event):
        x = self.nb_pers.get()
        self.nb_pers.set(x+1)


    def test_peut_placer_personnes(self, x, y):
        peut_placer_personnes = True
        for i in self.r:
            if (((i[0]-x)**2)+(i[1]-y)**2)**0.5 <= 2*rayon:
                peut_placer_personnes = False
                continue

        point = np.array([x, y])

        for i in self.murs :
            temp = np.dot(point - i[0, :], i[1, :] - i[0, :]) / (np.linalg.norm(i[1, :] - i[0, :]))**2
            temp = min( max( 0, temp ), 1 )
            proche = i[0, :] + (i[1, :] - i[0, :])*temp
            d = np.linalg.norm(proche - point)
            if d<=rayon :
                peut_placer_personnes = False
```

Annexe 1

```python
        for i in self.obstacles :
            temp = np.dot(point - i[0, :], i[1, :] - i[0, :]) / (np.linalg.norm(i[1, :] - i[0, :]))**2
            temp = min( max( 0, temp ), 1 )
            proche = i[0, :] + (i[1, :] - i[0, :])*temp
            d = np.linalg.norm(proche - point)
            if d<=rayon :
                peut_placer_personnes = False

        k = 0
        for i in self.murs:
            if i[0, 1] == i[1, 1]:
                if i[0, 1] < y:
                    if (i[0, 0] - x) * (i[1, 0] - x)<0:
                        k+=1
        if k%2==0:
            peut_placer_personnes = False

        return peut_placer_personnes


    def placer_personnes_manuel(self, event):
        x = int(event.x)
        y = int(event.y)
        if self.test_peut_placer_personnes(x, y):
            self.r = np.vstack([self.r, [[x, y]]])
            self.L_personnes_dessin.append(self.can.create_circle(x, y, rayon, fill = "red"))
            i = self.nb_pers.get()
            self.nb_pers.set(i+1)
            self.N += 1


    def placer_personnes_aleatoire(self):
        for i in self.L_personnes_dessin:
            self.can.delete(i)

        self.N = self.nb_pers.get()
        k = 0
        self.r = np.zeros((self.N, 2))
        self.L_personnes_dessin = [0]*self.N
        while k<self.N:
            x = np.random.randint(rayon, 600-rayon)
            y = np.random.randint(rayon, 600-rayon)

            if self.test_peut_placer_personnes(x, y):
                self.r[k, :] = [x, y]
                self.L_personnes_dessin[k] = self.can.create_circle(x, y, rayon, fill = "red")
                k+=1
```

Annexe 1

```python
    ##SORTIE##


    def placer_sortie(self):
        self.supprimer_parametres()
        tk.Button(self.tool, text = "   RETOUR", command = self.retour_sortie, width = 30, anchor = 'w', bg = '#f0f0f0', activebackground = '#bdbdbd').place(x = 0, y = 0)

        self.longueur_sortie = tk.IntVar()
        tk.Scale(self.tool, orient = "vertical", label = "Taille de la sortie", length = 300, from_ = 2*rayon, to = self.dmin/2, variable = self.longueur_sortie).place(x = 0, y = 70)

        self.s = self.longueur_sortie.get()
        self.peut_placer_sortie = True

        self.can.bind('<Up>', self.sortie_moins)
        self.can.bind('<Down>', self.sortie_plus)

        self.can.bind("<Enter>", self.bouger_sortie)
        self.can.bind("<Motion>", self.bouger_sortie)
        self.can.bind("<Button-1>", self.test_peut_placer_sortie)
        self.dot = self.can.create_circle(self.murs[0, 0, 0], self.murs[0, 0, 1], self.s, fill = "yellow")


    def sortie_moins(self, event):
        x = self.longueur_sortie.get()
        self.longueur_sortie.set(x-1)

    def sortie_plus(self, event):
        x = self.longueur_sortie.get()
        self.longueur_sortie.set(x+1)


    def bouger_sortie(self, event):
        self.s = self.longueur_sortie.get()
        if self.peut_placer_sortie:
            x, y = int(event.x), int(event.y)
            self.can.delete(self.dot)

            A = np.array([[x, y]])
            temp = np.dot(A - self.murs[0, 0, :], self.murs[0, 1, :] - self.murs[0, 0, :]) / (np.linalg.norm(self.murs[0, 1, :] - self.murs[0, 0, :]))**2
            temp = min( max(self.s/np.linalg.norm(self.murs[0, 1, :] - self.murs[0, 0, :]), temp ), 1 - self.s/np.linalg.norm(self.murs[0, 1, :] - self.murs[0, 0, :]) )
            proche = self.murs[0, 0, :] + (self.murs[0, 1, :] - self.murs[0, 0, :])*temp
            d = np.linalg.norm(proche - A)
            for j in range(1, len(self.murs)):
                temp = np.dot(A - self.murs[j, 0, :], self.murs[j, 1, :] - self.murs[j, 0, :]) / (np.linalg.norm(self.murs[j, 1, :] - self.murs[j, 0, :]))**2
                temp = min( max(self.s/np.linalg.norm(self.murs[j, 1, :] - self.murs[j, 0, :]), temp ), 1 - self.s/np.linalg.norm(self.murs[j, 1, :] - self.murs[j, 0, :]) )
                proche1 = self.murs[j, 0, :] + (self.murs[j, 1, :] - self.murs[j, 0, :])*temp
```

Annexe 1

```python
            d1 = np.linalg.norm(proche1 - A)
            if d1<d:
                d = d1
                proche = proche1
        self.dot = self.can.create_circle(proche[0], proche[1], self.s, fill = "orange")


    def test_peut_placer_sortie(self, event):
        x, y = int(event.x), int(event.y)
        if self.peut_placer_sortie:
            self.cliquer_pour_placer_sortie(x, y)
            self.peut_placer_sortie = False
        else:
            self.peut_placer_sortie = True


    def cliquer_pour_placer_sortie(self, x, y):
        i = 0

        A = np.array([[x, y]])
        temp = np.dot(A - self.murs[0, 0, :], self.murs[0, 1, :] - self.murs[0, 0, :]) / (np.linalg.norm(self.murs[0, 1, :] - self.murs[0, 0, :]))**2
        temp = min( max( self.s/np.linalg.norm(self.murs[0, 1, :] - self.murs[0, 0, :]), temp ), 1 - self.s/np.linalg.norm(self.murs[0, 1, :] - self.murs[0, 0, :]) )
        proche = self.murs[0, 0, :] + (self.murs[0, 1, :] - self.murs[0, 0, :])*temp
        d = np.linalg.norm(proche - A)

        for j in range(len(self.murs)):
            temp = np.dot(A - self.murs[j, 0, :], self.murs[j, 1, :] - self.murs[j, 0, :]) / (np.linalg.norm(self.murs[j, 1, :] - self.murs[j, 0, :]))**2

            temp = min( max(self.s/np.linalg.norm(self.murs[j, 1, :] - self.murs[j, 0, :]), temp ), 1 - self.s/np.linalg.norm(self.murs[j, 1, :] - self.murs[j, 0, :]))

            proche1 = self.murs[j, 0, :] + (self.murs[j, 1, :] - self.murs[j, 0, :])*temp
            d1 = np.linalg.norm(proche1 - A)
            if d1<d:
                i = j
                d = d1
                proche = proche1

        self.sortie = proche
        self.murs1 = np.delete(self.murs, i, 0)
        if self.murs[i, 0, 0] == self.murs[i, 1, 0]:
            if self.murs[i, 0, 1] < self.murs[i, 1, 1]:
                self.murs1 = np.vstack([self.murs1, [[[self.murs[i, 0, 0], self.murs[i, 0, 1]], [self.murs[i, 0, 0], proche[1] - self.s]]]])
                self.murs1 = np.vstack([self.murs1, [[[self.murs[i, 0, 0], proche[1] + self.s], [self.murs[i, 1, 0], self.murs[i, 1, 1]]]]])
```

Annexe 1

```
            elif self.murs[i, 0, 1] > self.murs[i, 1, 1]:
                self.murs1 = np.vstack([self.murs1, [[[self.murs[i, 0, 0], self.murs[i, 0, 1]], [self.murs[i, 0, 0], proche[1] + self.s]]]])
                self.murs1 = np.vstack([self.murs1, [[[self.murs[i, 0, 0], proche[1] - self.s], [self.murs[i, 1, 0], self.murs[i, 1, 1]]]]])
        elif self.murs[i, 0, 1] == self.murs[i, 1, 1]:
            if self.murs[i, 0, 0] < self.murs[i, 1, 0]:
                self.murs1 = np.vstack([self.murs1, [[[self.murs[i, 0, 0], self.murs[i, 0, 1]], [proche[0] - self.s, self.murs[i, 0, 1]]]]])
                self.murs1 = np.vstack([self.murs1, [[[proche[0] + self.s, self.murs[i, 0, 1]], [self.murs[i, 1, 0], self.murs[i, 1, 1]]]]])
            elif self.murs[i, 0, 0] > self.murs[i, 1, 0]:
                self.murs1 = np.vstack([self.murs1, [[[self.murs[i, 0, 0], self.murs[i, 0, 1]], [proche[0] + self.s, self.murs[i, 0, 1]]]]])
                self.murs1 = np.vstack([self.murs1, [[[proche[0] - self.s, self.murs[i, 0, 1]], [self.murs[i, 1, 0], self.murs[i, 1, 1]]]]])


    def retour_sortie(self):
        self.supprimer_parametres()
        self.murs = np.copy(self.murs1)
        self.sortie = np.array(self.sortie)
        self.can.unbind("<Enter>")
        self.can.unbind("<Motion>")
        self.can.unbind("<Button-1>")
        self.can.unbind('<Up>')
        self.can.unbind('<Down>')
        self.menu()
```

Annexe 1

```python
    ##SIMULATION##


    def lancer_simulation(self):
        self.supprimer_parametres()
        tk.Button(self.tool, text = "   RETOUR", command = self.retour_lancer_simulation, width = 30, anchor = 'w', bg = '#f0f0f0', activebackground = '#bdbdbd').place(x = 0, y = 0)

        self.pause = tk.Button(self.tool, text = "   STOP", command = self.anim, width = 30, anchor = 'w', bg = '#f0f0f0', activebackground = '#bdbdbd')
        self.pause.place(x = 0, y = 25)

        self.attente = tk.IntVar()
        tk.Scale(self.tool, orient = "vertical", label = "Temps", length = 300, from_ = 1, to = 10, variable = self.attente).place(x = 0, y = 70)

        self.temps_ecoule = tk.StringVar()
        self.temps_ecoule.set(0)
        tk.Label(self.tool, textvariable = self.temps_ecoule).place(x = 50, y = 415)
        tk.Label(self.tool, text = "Temps écoulé (en s) : ").place(x = 50, y = 400)

        self.agents_sortis = tk.StringVar()
        tk.Label(self.tool, textvariable = self.agents_sortis).place(x = 50, y = 450)

        self.evacue = []

        self.simulation_active = True
        self.rtot = np.zeros((1, len(self.r), 2))
        self.rtot[0] = self.r
        self.pos = Position(len(self.r), rayon, self.sortie, self.murs, self.obstacles, dt)
        self.Pa = np.zeros((len(self.r)))
        self.Pa_tot = np.zeros((1, len(self.r)))

        self.can.bind('<Up>', self.temps_moins)
        self.can.bind('<Down>', self.temps_plus)
        self.can.bind('<space>', self.animation_pause)
        self.deplacement()


    def retour_lancer_simulation(self):
        self.simulation_active = False
        self.supprimer_parametres()
        self.menu()


    def temps_moins(self, event):
        x = self.attente.get()
        self.attente.set(x-1)

    def temps_plus(self, event):
        x = self.attente.get()
        self.attente.set(x+1)
```

Annexe 1

```python
    def animation_pause(self, event):
        anim()

    def anim(self):
        if self.simulation_active:
            self.pause.config(bg = '#bdbdbd', activebackground = '#f0f0f0')
            self.simulation_active = False
        else :
            self.simulation_active = True
            self.pause.config(bg = '#f0f0f0', activebackground = '#bdbdbd')
            self.deplacement()



    def deplacement(self):
        self.v = self.pos.vit_des(self.r, self.Pa)
        while self.simulation_active:
            if len(self.evacue)>=len(self.rtot[0]):
                self.simulation_active = False

            k = self.temps_ecoule.get()
            self.temps_ecoule.set(float(k)+dt)
            r1, v1, Pa1, sortis = self.pos.euler(self.r, self.v, self.Pa)

            for i in sortis:
                self.evacue.append(i)

            self.agents_sortis.set("Agents évacués : "+str(len(self.evacue))+"/"+str(len(r1)))

            self.rtot = np.vstack([self.rtot,[r1]])
            for i in range(len(self.r)):
                self.can.move(self.L_personnes_dessin[i],(r1[i, 0] - self.r[i, 0]), (r1[i, 1] - self.r[i, 1]))
            self.r = r1
            self.v = v1
            #self.Pa = Pa1



            attente_num = self.attente.get()
            time.sleep(0.01/attente_num)
            self.update()





anim = Animation()
anim.title("Simulation Evacuation")
anim.geometry("800x600")

anim.mainloop()
```

Annexe 2

Modèle Helbings :

$$m_i \vec{a}_i = \sum_{j \neq i} \vec{f_{j \to i}} + \sum_{obst} \vec{f_{obst \to i}} + \frac{1}{\tau_i}(v_i^0 \vec{e_i^0} - \vec{v}_i)$$

$$\vec{f_{j \to i}} = [A e^{\frac{ray_i + ray_j - d_{ij}}{B}} + k * g(ray_i + ray_j - d_{ij})] \vec{n}_{ij} + \kappa * g(ray_i + ray_j - d_{ij}) \Delta v_{ij}^t \vec{t}_{ij}$$

$$\Delta v_{ij}^t = (\vec{v}_j - \vec{v}_i) \cdot \vec{t}_{ij} \qquad g(x) = \begin{matrix} x \; si \; x > 0 \\ 0 \; sinon \end{matrix}$$

$$\vec{f_{obst \to i}} = [A e^{\frac{ray_i - d_{i-obst}}{B}} + k * g(ray_i - d_{i-obst})] \vec{n_{i-obst}} + \kappa * g(ray_i - d_{i-obst}) \Delta v_{i-obst}^t \vec{t_{i-obst}}$$

$$\Delta v_{i-obst}^t = -\vec{v}_i \cdot \vec{t}_{ij}$$

Annexe 3

<u>Code modèle Helbings :</u>

```python
##Modèle de Helbings##

class Helbings():

    def __init__(self, murs, sortie, N, ray, m, tau, dt):
        self.A = 2*10**3                                  # constante (N) : int
        self.B = 0.08                                     # constante (m) : int
        self.k = 1.2*10**5                                # parametre (kg/s^2) : int
        self.kap = 2.4*10**5                              # parametre (kg/(m*s^2))

        self.murs = murs
        self.sortie = sortie

        self.v0 = np.ones((N))*1.5
        self.ray = ray
        self.m = m

        self.tau = tau
        self.dt = dt



    #Calcul de la force totale exercee sur les agents
    def f_final_Helbings(self, r, v):
        a = np.zeros((len(r), 2))
        v0e0 = self.vit_des_helbings(r)
        F_ag = self.f_agent_helbings(r, v)
        F_mur = self.f_mur_helbings(r, v)

        for i in range(len(r)):
            a[i, :] = ((v0e0[i, :] - v[i, :])/self.tau) + (F_ag[i, :]/self.m[i]) + ((F_mur[i, :])/self.m[i])
        return a



    #Calcul de la direction desiree des agents
    def dir_des(self, r):
        e0 = np.zeros((len(r), 2))
        for i in range(len(r)):
            e0[i, :] = (self.sortie - r[i, :])/ np.linalg.norm(self.sortie - r[i, :])
        return e0


    #Calcul de la vitesse desiree des agents
    def vit_des_helbings(self, r):
        v0e0 = np.zeros((len(r), 2))
        e0 = self.dir_des(r)
        for i in range(len(r)):
            v0e0[i, :] = self.v0[i]*e0[i, :]
        return v0e0
```

Annexe 3

```
    """f_agents"""

    def g(self, x):
        if x < 0:
            return 0
        return x


    #Caracteristiques agent i vers j
    def agent(self, i, j, r):
        d = np.linalg.norm(r[i, :] - r[j, :])
        n = (r[i, :] - r[j, :])/d
        t = np.array([-n[1], n[0]])
        return d, n, t


    #Force exercee par j sur i
    def f_ij(self, i, j, r, v):
        d, n, t = self.agent(i, j, r)
        dv_t = np.dot(v[j, :] - v[i, :], t)
        r_tot = self.ray[i] + self.ray[j]
        a = (self.A * np.exp((r_tot - d)/self.B)) + (self.k * self.g(r_tot - d))
        b = self.kap * self.g(r_tot - d) * dv_t
        return a*n + b*t


    #Force totale des agents
    def f_agent_helbings(self, r, v):
        F = np.zeros((len(r), 2))
        for i in range(len(r)):
            for j in range(len(r)):
                if i != j:
                    F[i, :] += self.f_ij(i, j, r, v)
        return F
```

Annexe 3

```
    """f_mur"""

    #Caracteristiques de l'agent i au segment j (murs ou obstacle)
    def dist_iM(self, i, r, seg):
        temp = np.dot(r[i, :] - seg[0, :], seg[1, :] - seg[0, :]) / (np.linalg.norm(seg[1, :] - seg[0, :]))**2
        temp = min( max( 0, temp ), 1 )
        proche = seg[0, :] + (seg[1, :] - seg[0, :])*temp
        d = np.linalg.norm(proche - r[i, :])
        n = r[i, :] - proche / d
        t = np.array([-n[1], n[0]])
        return d, n, t, proche

    #Force exercee par le mur j sur l'agent i
    def f_iM(self, i, j, r, v):
        d, n, t = self.dist_iM(i, r, self.murs[j, :, :])[:-1]
        dv_t = -np.dot(v[i, :], t)
        a = (self.A * np.exp((self.ray[i] - d)/self.B)) + (self.k * self.g(self.ray[i] - d))
        b = self.kap * self.g(self.ray[i] - d) * dv_t
        return a*n + b*t


    #Force totale des murs
    def f_mur_helbings(self, r, v):
        F = np.zeros((len(r), 2))
        for i in range(len(r)):
            for j in range(len(self.murs)):
                F[i, :] += self.f_iM(i, j, r, v)
        return F
```

Annexe 3

```python
    #Calcul de la vitesse et de la position des agents à l'instant k
    def euler_Helbings(self, r, v):
        a1 = np.zeros((len(r), 2))
        v1 = np.zeros((len(r), 2))
        r1 = np.zeros((len(r), 2))

        evacue = []

        a1 = self.f_final_Helbings(r, v)
        v1 = v + self.dt * a1
        r1 = r + self.dt * v1

        for i in range(len(r1)):
            if np.linalg.norm(r1[i, :] - self.sortie) < 2*self.ray[i]:
                r1[i, :] = np.random.rand(1, 2) * 10**6
                evacue.append(i)

        return r1, v1, evacue
```

Annexe 4

<u>Modèle CEPABS :</u>

$$\vec{f}_{obst,retard} = \gamma \frac{A_{obst,retard}\,\Delta v^{n}_{i-obst}}{e^{d_{i-obst}-ray_i}}\,\vec{n}_{i-obst}$$

$$\vec{f}_{agent,retard} = \gamma\, A_{agent,retard}\, g'\left(d_{ij}-ray_i-ray_j\right)\Delta v^{t}_{ij}\,\vec{n}_{ij}$$

$$\vec{f}_{obst,\text{côté}} = \gamma\, A_{obst,\text{côté}}\, g'\left(d_{i-obst}-ray_i\right)\Delta v^{n}_{i-obst}\,\vec{e}^{\,0}_{i,tangentielle}$$

$$\vec{f}_{agent,\text{côté}} = \gamma\, A_{agent,\text{côté}}\, g'\left(d_{ij}-ray_i-ray_j\right)\Delta v^{n}_{ij}\,\vec{t}_{ij}$$

$$\vec{f}_{suivre\ mur} = A_{suivre\ mur}\, e^{\frac{c*ray_i-d_{i-obst}}{B}}\,\vec{t}_{i-obst}$$

Annexe 5

<u>Code modèle Helbings :</u>

```
##Modèle CEPABS##


class CEPABS():

    def __init__(self, murs, obstacles, sortie, N, ray, m, tau, dt, p_suivre):

        self.Aag_retard = 900            # constante (à trouver experimentalement)
        self.Aag_evite = 10             # constante (à trouver experimentalement)
        self.Aob_retard = 900          # constante (à trouver experimentalement)
        self.Aob_evite = 10            # constante (à trouver experimentalement)
        self.Aob_contact = 2*10**3
        self.Asuivre_mur = 30          # constante (à trouver experimentalement)

        self.c1 = 0.5                     # constante (à trouver experimentalement)
        self.c2 = 0.4                     # constante (à trouver experimentalement)
        self.c3 = 4                     # constante (à trouver experimentalement)
        self.c4 = 0.5                     # constante (à trouver experimentalement)

        self.A = 2*10**3                # constante (N) : int
        self.B = 0.08                   # constante (m) : int
        self.k = 1.2*10**5              # parametre (kg/s^2) : int
        self.kap = 2.4*10**5            # parametre (kg/(m*s^2))

        self.murs = murs
        self.obstacles = obstacles
        self.sortie = sortie

        self.v0i = np.ones((N))*3
        #self.v0i = np.random.rand((N))*3 + 3*np.ones((N))      # vitesse desiree initiale (en norme)
        self.m = np.ones((N))*80        # masse
        self.ray = ray
        self.tau = 0.5                  # duree d'acceleration
        self.dt = dt                    # pas d'integration

        self.p_suivre = p_suivre


    #Calcul de la force totale exercee sur les agents
    def f_final_cepabs(self, r, v, Pa):
        a = np.zeros((len(r), 2))
        v0e0 = self.vit_des(r, Pa)
        F_ag = self.f_agent(r, v)
        F_mur = self.f_mur(r, v)
        F_ob = self.f_obstacles(r, v)
        F_aj = self.f_vitesse_amortie(v)

        for i in range(len(r)):
            a[i, :] = ((v0e0[i, :] - v[i, :])/self.tau) + (F_ag[i, :]/self.m[i]) + ((F_mur[i, :])/self.m[i]) + (F_ob[i, :]/
self.m[i]) - (F_aj[i, :] / self.m[i])
        return a
```

Annexe 5

```
    #Calcul de la direction desiree des agents
    def dir_des(self, r):
        e0 = np.zeros((len(r), 2))
        for i in range(len(r)):
            e0[i, :] = (self.sortie - r[i, :])/ np.linalg.norm(self.sortie - r[i, :])
        return e0

    #Tendance à suivre les autres
    def dir_des_moyenne(self, r, p):
        e0 = self.dir_des(r)
        e0_moyenne = np.array([0.0, 0.0])

        for i in range(len(r)):
            e0_moyenne += e0[i, :]
        e0_moyenne /= np.linalg.norm(e0_moyenne)
        e0_s = ((1-p)*e0 - p*e0_moyenne) / np.linalg.norm((1-p)*e0 - p*e0_moyenne)
        return e0_s

    #Calcul de la norme de la vitesse desiree des agents
    def vit_des_norme(self, r, Pa):
        v0 = np.zeros((len(r)))
        for i in range(len(r)):
            v0[i] = self.v0i[i] * (1 + (Pa[i]/100))
        return v0

    #Calcul de la vitesse desiree des agents
    def vit_des(self, r, Pa):
        v0e0 = np.zeros((len(r), 2))
        v0 = self.vit_des_norme(r, Pa)
        e0 = self.dir_des_moyenne(r, self.p_suivre)
        for i in range(len(r)):
            v0e0[i, :] = v0[i]*e0[i, :]
        return v0e0
```

Annexe 5

```python
    """f_agents"""

    def g(self, x):
        if x < 0:
            return 0
        return x

    #Caracteristiques agent i vers j
    def agent(self, i, j, r):
        d = np.linalg.norm(r[i, :] - r[j, :])
        n = (r[i, :] - r[j, :])/d
        t = np.array([-n[1], n[0]])
        return d, n, t

    #Force exercee par j sur i
    def f_ij(self, i, j, r, v):
        d, n, t = self.agent(i, j, r)
        dv_t = np.dot(v[j, :] - v[i, :], t)
        r_tot = self.ray[i] + self.ray[j]
        a = (self.A * np.exp((r_tot - d)/self.B)) + (self.k * self.g(r_tot - d))
        b = self.kap * self.g(r_tot - d) * dv_t
        return a*n + b*t

    #Si l'agent i voit l'agent j
    def gamma_agent(self, i, j, r):
        e0 = self.dir_des_moyenne(r, self.p_suivre)
        d, n, t = self.agent(i, j, r)
        x = np.dot(-n, e0[i, :])
        if x < 0 :
            return 0
        if d < 60 :
            y = x * d
            z = np.sqrt(d**2 + y**2)
            if self.ray[i] + self.ray[j] > z :
                return 1
        return 0


    def g_prime1(self, i, j, r):
        if np.linalg.norm(r[j, :] - r[i, :]) > self.ray[i] + self.ray[j]:
            return 1 / (np.linalg.norm(r[j, :] - r[i, :]) - (self.ray[i] + self.ray[j]))
        return 0

    #Ralentir a la vue d'agent
    def f_agent_retard(self, i, j, r, v):
        d, n, t = self.agent(i, j, r)
        dv_n = np.dot(v[j, :] - v[i, :], t)
        a = self.Aag_retard * self.gamma_agent(i, j, r) * self.g_prime1(i, j, r) * dv_n
        return a * n
```

Annexe 5

```python
    #Eviter les agents
    def f_agent_evite(self, i, j, r, v):
        d, n, t = self.agent(i, j, r)
        dv_n = np.dot(v[j, :] - v[i, :], n)
        a = self.Aag_evite * self.gamma_agent(i, j, r) * self.g_prime1(i, j, r)* dv_n
        return a * t

    #Force totale des agents
    def f_agent(self, r, v):
        F = np.zeros((len(r), 2))
        for i in range(len(r)):
            for j in range(len(r)):
                if i != j:
                    F[i, :] += self.f_ij(i, j, r, v)
                    F[i, :] += self.f_agent_retard(i, j, r, v)
                    F[i, :] += self.f_agent_evite(i, j, r, v)
        return F


    """f_mur"""

    #Caracteristiques de l'agent i au segment j (murs ou obstacle)
    def dist_iM(self, i, r, seg):
        temp = np.dot(r[i, :] - seg[0, :], seg[1, :] - seg[0, :]) / (np.linalg.norm(seg[1, :] - seg[0, :]))**2
        temp = min( max( 0, temp ), 1 )
        proche = seg[0, :] + (seg[1, :] - seg[0, :])*temp
        d = np.linalg.norm(proche - r[i, :])
        n = proche - r[i, :] / d
        t = np.array([-n[1], n[0]])
        return d, n, t, proche

    #Force exercee par le mur j sur l'agent i
    def f_iM(self, i, j, r, v):
        d, n, t = self.dist_iM(i, r, self.murs[j, :, :])[:-1]
        dv_t = -np.dot(v[i, :], t)
        a = (self.A * np.exp((self.ray[i] - d)/self.B)) + (self.k * self.g(self.ray[i] - d))
        b = self.kap * self.g(self.ray[i] - d) * dv_t
        return a*n + b*t


    def seg_intersect(self, a1,a2,b1,b2):
        da = a2-a1
        db = b2-b1
        dp = a1-b1
        dap = np.array([-da[1],da[0]])
        denom = np.dot(dap, db)
        num = np.dot(dap, dp)
        return (num / denom.astype(float))*db + b1
```

Annexe 5

```python
    #Si l'agent i voit l'obstacle j
    def gamma_obstacle(self, i, j, r, seg):
        e0 = self.dir_des_moyenne(r, self.p_suivre)
        d = self.dist_iM(i, r, seg[j, :, :])[0]
        if d < 60 :
            proche = self.seg_intersect(r[i], r[i] + e0[i], seg[j, 0], seg[j, 1])
            if np.dot(seg[j, 0] - proche, seg[j, 1] - proche) <0:
                if np.dot(e0[i], proche - r[i]) > 0:
                    return 1
        return 0


    def g_prime2(self, i, j, r, seg):
        d, n, t, proche = self.dist_iM(i, r, seg[j, :, :])
        if d > self.ray[i]:
            return 1 / (d - self.ray[i])
        return 0

    #Ralentir a la vue de l'obstacle
    def f_ob_retard(self, i, j, r, v, seg):
        d, n, t = self.dist_iM(i, r, seg[j, :, :])[:-1]
        dv_n = -np.dot(v[i, :], n)
        a = self.Aob_retard * self.gamma_obstacle(i, j, r, seg) * dv_n*(np.exp(self.ray[i] - d))
        return a*n

    #Stopper l'agent i au contact de l'obstacle j
    def f_ob_contact(self, i, j, r, v):
        d, n, t = self.dist_iM(i, r, self.obstacles[j])[:-1]
        dv_n = -np.dot(v[i, :], n)
        a = self.Aob_contact * dv_n * np.exp((self.ray[i]+0.05-d)/self.B)
        return a*n

    #Eviter l'obstacle
    def f_ob_evite(self, i, j, r, v, seg):
        d, n, t = self.dist_iM(i, r, seg[j, :, :])[:-1]
        dv_n = -np.dot(v[i, :], n)
        e0 = self.dir_des_moyenne(r, self.p_suivre)
        e0_t = np.array([-e0[i, 1], e0[i, 0]])
        a = self.Aob_evite * self.gamma_obstacle(i, j, r, seg) * self.g_prime2(i, j, r, seg) * dv_n
        return a * e0_t

    #Suivre le mur
    def f_suivre_mur(self, i, j, r):
        d, n, t = self.dist_iM(i, r, self.obstacles[j, :, :])[:-1]
        a = self.Asuivre_mur * np.exp((1.07 * self.ray[i] - d)/self.B)
        e0 = self.dir_des_moyenne(r, self.p_suivre)
        return a * t
```

Annexe 5

```python
    #Force totale des murs
    def f_mur(self, r, v):
        F = np.zeros((len(r), 2))
        for i in range(len(r)):
            for j in range(len(self.murs)):
                F[i, :] += self.f_iM(i, j, r, v)
                F[i, :] += self.f_ob_retard(i, j, r, v, self.murs)

        return F

    #Force totale des obstacles
    def f_obstacles(self, r, v):
        F = np.zeros((len(r), 2))
        for i in range(len(r)):
            for j in range(len(self.obstacles)):
                F[i, :] += self.f_ob_retard(i, j, r, v, self.obstacles)
                F[i, :] += self.f_ob_contact(i, j, r, v)
                F[i, :] += self.f_ob_evite(i, j, r, v, self.obstacles)
                F[i, :] += self.f_suivre_mur(i, j, r)
        return F


    """Ajustements"""

    #Amortissement de la vitesse
    def f_vitesse_amortie(self, v):
        v_am = np.zeros((len(v), 2))
        for i in range(len(v)):
            v_am[i, :] = ((np.linalg.norm(v[i, :]))**0.5)*v[i, :]
        return v_am



    """Panique"""

    #Solitude
    def solitude(self, r):
        S = np.zeros((len(r)))
        for i in range(len(r)):
            for j in range(len(r)):
                if i != j:
                    d = self.agent(i, j, r)[0]
                    if d < 1:
                        S[i] +=1
                        break
        return S
```

Annexe 5

```python
    #Eq diff panique
    def panique(self, r, v, Pa):
        Pa_moyenne = 0
        for i in Pa:
            Pa_moyenne += i
        Pa_moyenne /= len(Pa)
        v0e0 = self.vit_des(r, Pa)
        S = self.solitude(r)

        Pa_prime = np.zeros((len(Pa)))
        for i in range(len(Pa)):
            Pa_prime[i] =  self.c1*(Pa_moyenne - Pa[i]) + self.c2*np.linalg.norm(v0e0[i, :] - v[i, :]) + self.c3*S[i] + self.c4
        return Pa_prime



    """Calcul de la position"""

    #Calcul de la vitesse et de la position des agents à l'instant k
    def euler(self, r, v, Pa):
        a1 = np.zeros((len(r), 2))
        v1 = np.zeros((len(r), 2))
        r1 = np.zeros((len(r), 2))

        evacue = []

        a1 = self.f_final_cepabs(r, v, Pa)
        v1 = v + self.dt * a1
        r1 = r + self.dt * v1

        Pa_prime = self.panique(r, v, Pa)
        Pa1 = Pa + self.dt * Pa_prime

        for i in range(len(r1)):
            if np.linalg.norm(r1[i, :] - self.sortie) < 2*self.ray[i]:
                r1[i, :] = np.random.rand(1, 2) * 10**6
                evacue.append(i)

        return r1, v1, Pa1, evacue
```

Annexe 6

Recherche des constantes :

```
import numpy as np
import matplotlib.pyplot as plt

from Modeles import *

rayon = 3
dt = 0.1
tau = 0.5
p_suivre = 0


##Paramètres##

murs = np.array([[[200.,  200.],[440.,  200.]],[[440.,  200.],[440., 360.]],[[440., 360.],[200., 360.]],[[200., 360.],[200., 280 + 2*rayon]],[[200., 280 - 2*rayon],[200.,  200.]]])

obstacles = np.array([[[330, 300],[330, 260]]])

r = np.array([[400., 290.]])

sortie = np.array([200., 280.])

Pa = np.zeros((len(r)))
```

Annexe 6

```python
##Tests##


X = np.arange(300, 4010, 10)
Y = []

for i in X:
    evacue = []
    position = CEPABS(murs, obstacles, sortie, len(r), rayon, tau, dt, p_suivre, i) # On pose self.Aob_retard = i dans CEPABS

    while len(evacue)<len(rtot[0]):

        r1, v1, Pa1, sortis = position.euler(r, v, Pa)
        for i in sortis:
            evacue.append(i)

        rtot = np.vstack([rtot,[r1]])
        vtot = np.vstack([vtot,[v1]])

        r = r1
        v = v1
        Pa = Pa1

    vmoy = np.mean(vtot)
    Y.append(vmoy)

plt.clf()
plt.plot(X, Y)
plt.grid()
plt.axis([200, 1600, 0, 15])

plt.xlabel("Aob_retard")
plt.ylabel("Vmoy")
plt.show()
```

Annexe 7

Test Panique :

```python
import numpy as np
import matplotlib.pyplot as plt

from Modeles import *

rayon = 3
dt = 0.1
tau = 0.5
p_suivre = 0

##Paramètres##

murs = np.array([[[200.,  200.],[440.,  200.]],[[440.,  200.],[440., 360.]],[[440., 360.],[200., 360.]],[[200., 360.],[200., 280
+ 2*rayon]],[[200., 280 - 2*rayon],[200.,  200.]]])

r = np.array([[305., 305.],
       [285., 268.],
       [217., 250.],
       [433., 270.],
       [266., 349.],
       [265., 248.],
       [375., 229.],
       [420., 217.],
       [317., 235.],
       [428., 257.],
       [314., 260.],
       [213., 216.],
       [341., 237.],
       [390., 212.],
       [295., 226.],
       [317., 216.],
       [251., 284.],
       [404., 268.],
       [316., 348.],
       [430., 302.],
       [336., 342.],
       [345., 269.],
       [378., 269.],
       [226., 322.],
       [231., 333.],
       [395., 302.],
       [392., 277.],
       [373., 300.],
       [360., 206.],
       [391., 246.]])


sortie = np.array([200., 280.])

obstacles = np.array([[[0, 0],[0, 0]]])
```

Annexe 7

```python
##Tests##

X = np.arange(0, 101, 1)
Y = []


for i in X:
    Pa = np.ones((len(r)))*i
    evacue = []
    position = CEPABS(murs, obstacles, sortie, len(r), rayon, tau, dt, p_suivre)

    while len(evacue)<len(rtot[0]):

        r1, v1, Pa1, sortis = position.euler(r, v, Pa)
        for i in sortis:
            evacue.append(i)

        rtot = np.vstack([rtot,[r1]])
        vtot = np.vstack([vtot,[v1]])

        r = r1
        v = v1

    Y.append(len(rtot)*dt)

plt.clf()
plt.plot(X, Y)
plt.grid()
plt.axis([0, 1, 0, 200])

plt.xlabel("Panique")
plt.ylabel("Temps de sortie")
plt.show()
```

Annexe 8

Test Obstacles :

```python
import numpy as np
import random as rd
import matplotlib.pyplot as plt

from Modeles import *

tau = 0.5                    # temps de réaction
dt = 0.1                     # pas d'integration
rayon = 5                    # rayon des agents -> 1m = rayon*2 pixels
p_suivre = 0                 # tendance à suivre les autres



##Paramètres##

murs = np.array([[[200.,  200.],[440.,  200.]],[[440.,  200.],[440., 360.]],[[440., 360.],[200., 360.]],[[200., 360.],[200., 280
+ 2*rayon]],[[200., 280 - 2*rayon],[200.,  200.]]])

sortie = np.array([200., 280.])

m = np.ones((len(r)))*80


Pa = np.zeros((len(r)))



##Tests##


X = [k for k in range(240)]
Y = [k for k in range(160)]
Z = np.zeros((240, 160))


def test_peut_placer_personnes(x, y, r, obstacles):
    peut_placer_personnes = True
    for i in r:
        if (((i[0]-x)**2)+(i[1]-y)**2)**0.5 <= 2*rayon:
            peut_placer_personnes = False
            continue

    point = np.array([x, y])

    for i in murs :
        temp = np.dot(point - i[0, :], i[1, :] - i[0, :]) / (np.linalg.norm(i[1, :] - i[0, :]))**2
        temp = min( max( 0, temp ), 1 )
        proche = i[0, :] + (i[1, :] - i[0, :])*temp
        d = np.linalg.norm(proche - point)
        if d<=rayon :
            peut_placer_personnes = False
```

Annexe 8

```
    k = 0
    for i in murs:
        if i[0, 1] == i[1, 1]:
            if i[0, 1] < y:
                if (i[0, 0] - x) * (i[1, 0] - x)<0:
                    k+=1
    if k%2==0:
        peut_placer_personnes = False

    return peut_placer_personnes



for i in range(21, 240):
    for j in range(1, 140):
        obstacles = np.array([[[200, 200],[200, 220]]]) + np.array([[i, j], [i, j]]) #longueur : 20 = 2m
        r = np.zeros((30, 2))
        k = 0
        while k<30:
            x = np.random.randint(rayon, 600-rayon)
            y = np.random.randint(rayon, 600-rayon)
            if test_peut_placer_personnes(x, y, r, obstacles):
                r[k] = [x, y]
                k+=1
        evacue = []
        position = CEPABS(murs, obstacles, sortie, len(r), rayon, tau, dt, p_suivre)
        while len(evacue)<len(rtot[0]):

            r1, v1, Pa1, sortis = position.euler(r, v, Pa)
            for i in sortis:
                evacue.append(i)
            rtot = np.vstack([rtot,[r1]])
            vtot = np.vstack([vtot,[v1]])
            r = r1
            v = v1
            Pa = Pa1
        ttot = len(rtot)*dt/(2*rayon)
            Z[i, j] += ttot

plt.clf()
x, y = np.meshgrid(X, Y)
plt.contourf(x, y, np.transpose(Z), 1000)
plt.colorbar()
plt.axis('equal')
plt.show()
```

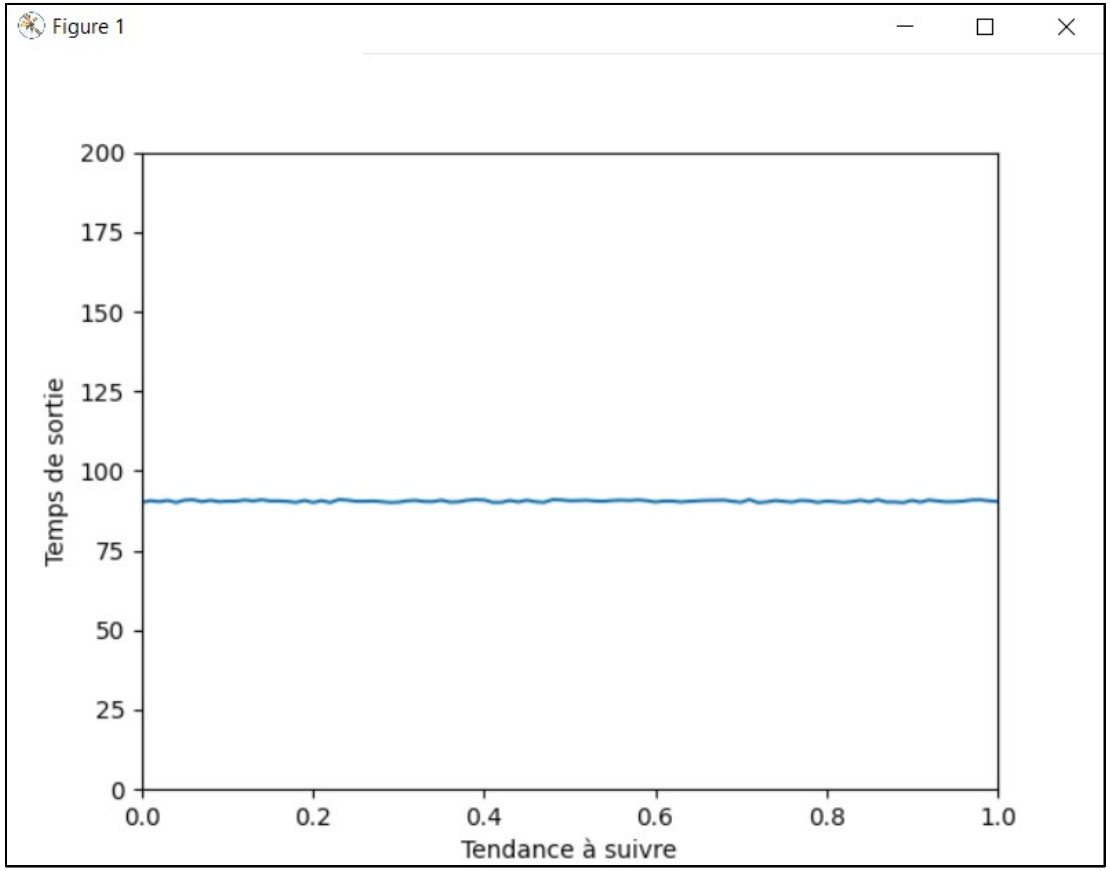
Annexe 9

Ajout du modèle comportemental :

Tendance à suivre les autres :

$$\boxed{\vec{e}_i^{\,0\,\prime} = \frac{(1-q)\,\vec{e}_i^{\,0} + q\,\langle \vec{e}_j^{\,0} \rangle}{\|(1-q)\,\vec{e}_i^{\,0} + q\,\langle \vec{e}_j^{\,0} \rangle\|}} \qquad q \in [0,1]$$

Annexe 9



Pas d’incidence sur la durée totale de l’évacuation

Annexe 10

<u>Test Tendance à suivre :</u>

```python
import numpy as np
import matplotlib.pyplot as plt
from Modeles import *

rayon = 3
dt = 0.1
tau = 0.5

X = np.arange(0, 1.01, 0.01)
Y = []

murs = np.array([[[156.,  87.],[324.,  87.]],[[324.,  87.],[324., 172.]],[[324., 172.],[156., 172.]],[[156., 172.],[156., 139.]],
[[156., 127.],[156.,  87.]]])
r = np.array([[196., 114.],[209., 149.],[205., 133.], [209., 115.],[278., 106.],[269., 167.],[266., 152.],[193., 145.],[257.,
132.],[229., 161.]])

obstacles = np.array([[[0, 0],[0, 0]]])
sortie = np.array([156., 133.])
ray = np.ones((len(r)))*rayon
m = np.ones((len(r)))*80
Pa = np.zeros((len(r)))

for i in X:
    L = []
    for j in range(100):
        evacue = []
        position = CEPABS(murs, obstacles, sortie, len(r), ray, m, tau, dt, i)

        while len(evacue)<len(rtot[0]):

            r1, v1, Pa1, sortis = position.euler(r, v, Pa)
            for i in sortis:
                evacue.append(i)

            rtot = np.vstack([rtot,[r1]])
            vtot = np.vstack([vtot,[v1]])

            r = r1
            v = v1
            Pa = Pa1

        L.append(len(rtot)*dt)
    Y.append(np.mean(L))

plt.clf()
plt.plot(X, Y)
plt.grid()
plt.axis([0, 1, 0, 200])

plt.xlabel("Aob_retard")
plt.ylabel("Vmoy")
plt.show()
```