

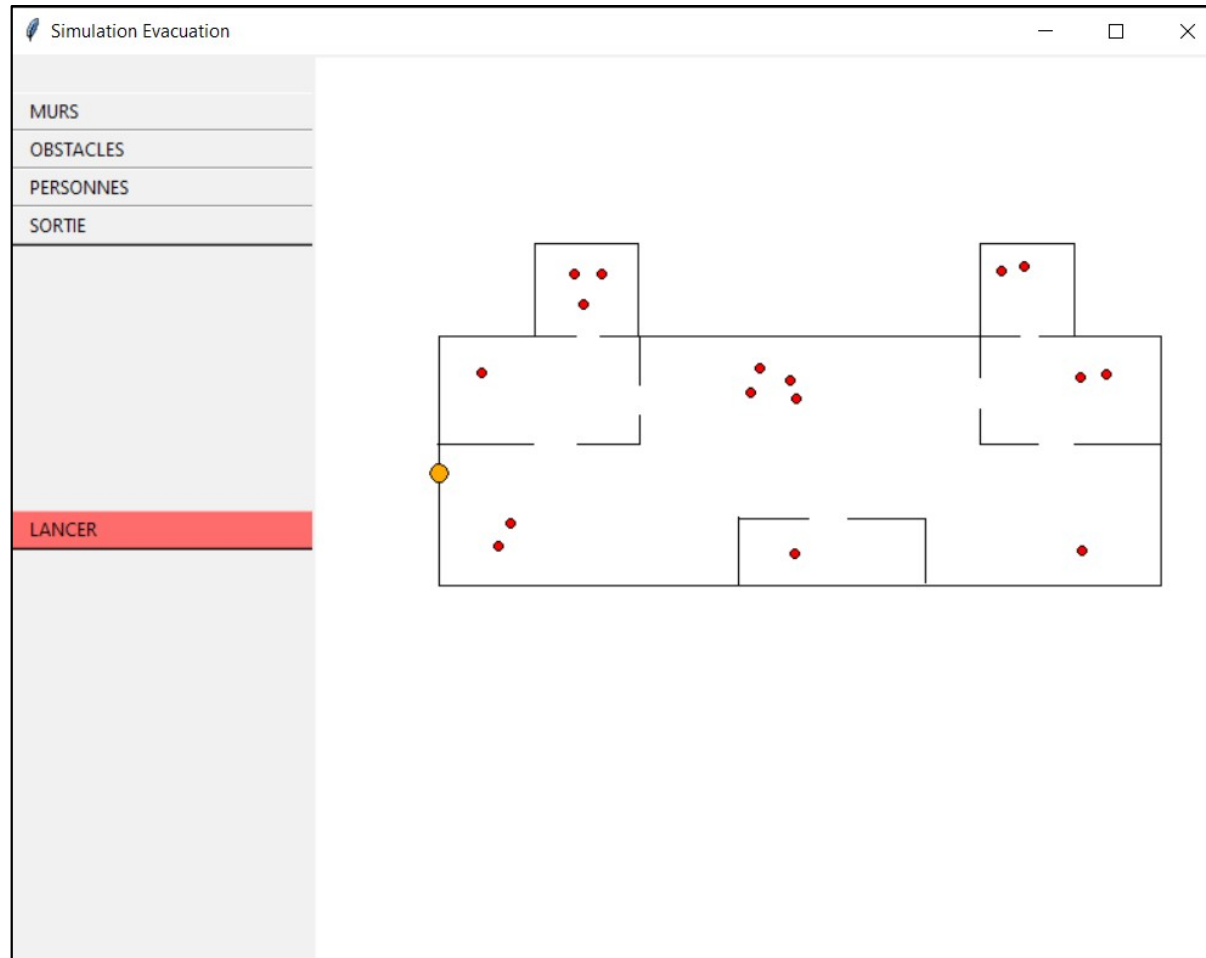


Modélisation et étude du mouvement de foule lors d'une évacuation d'un bâtiment dans le cas d'une situation d'urgence

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

Session 2021-2022

Objectifs et enjeux



Problématique

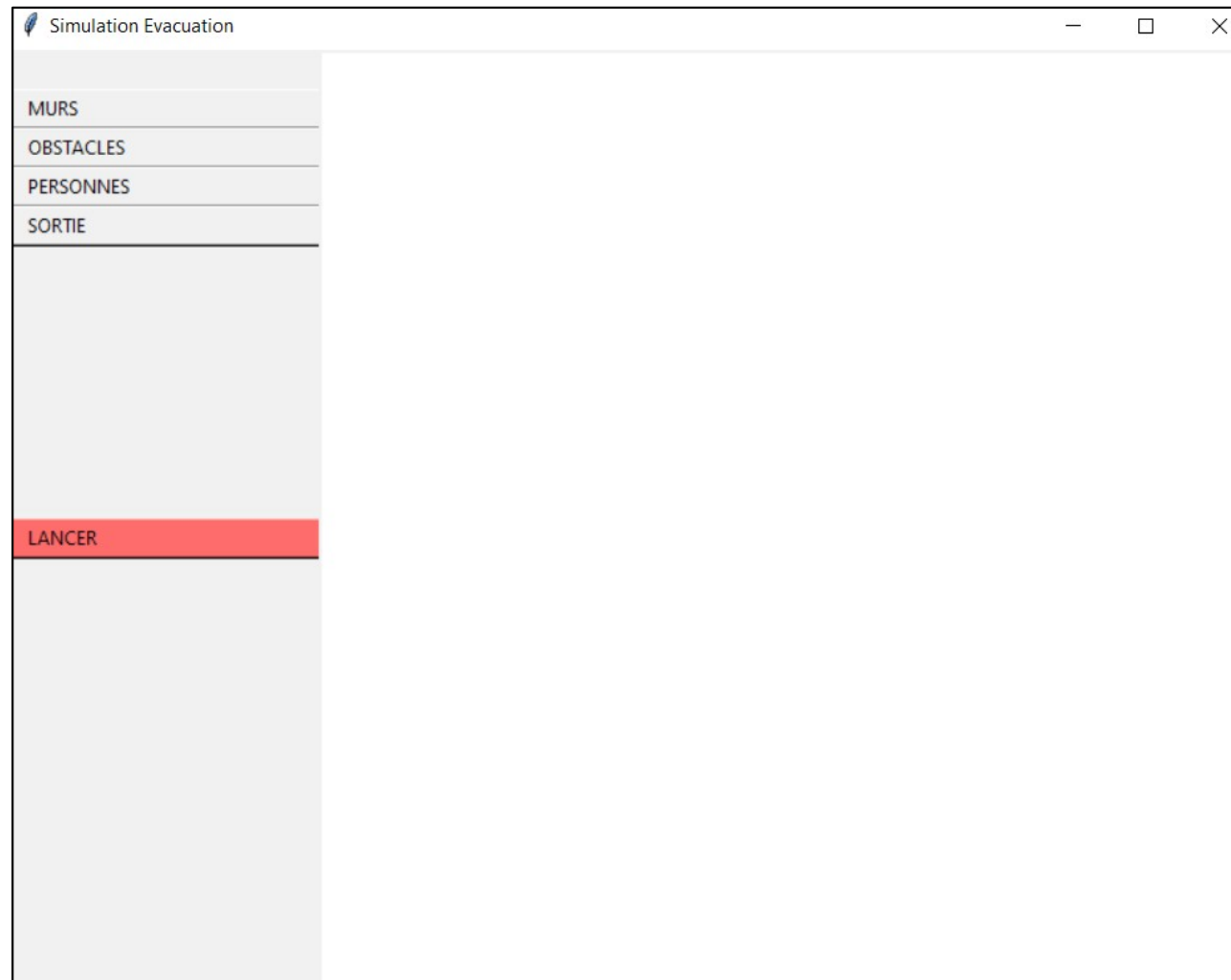
Quels sont les facteurs sur lesquels agir afin d'optimiser l'évacuation d'une foule ?



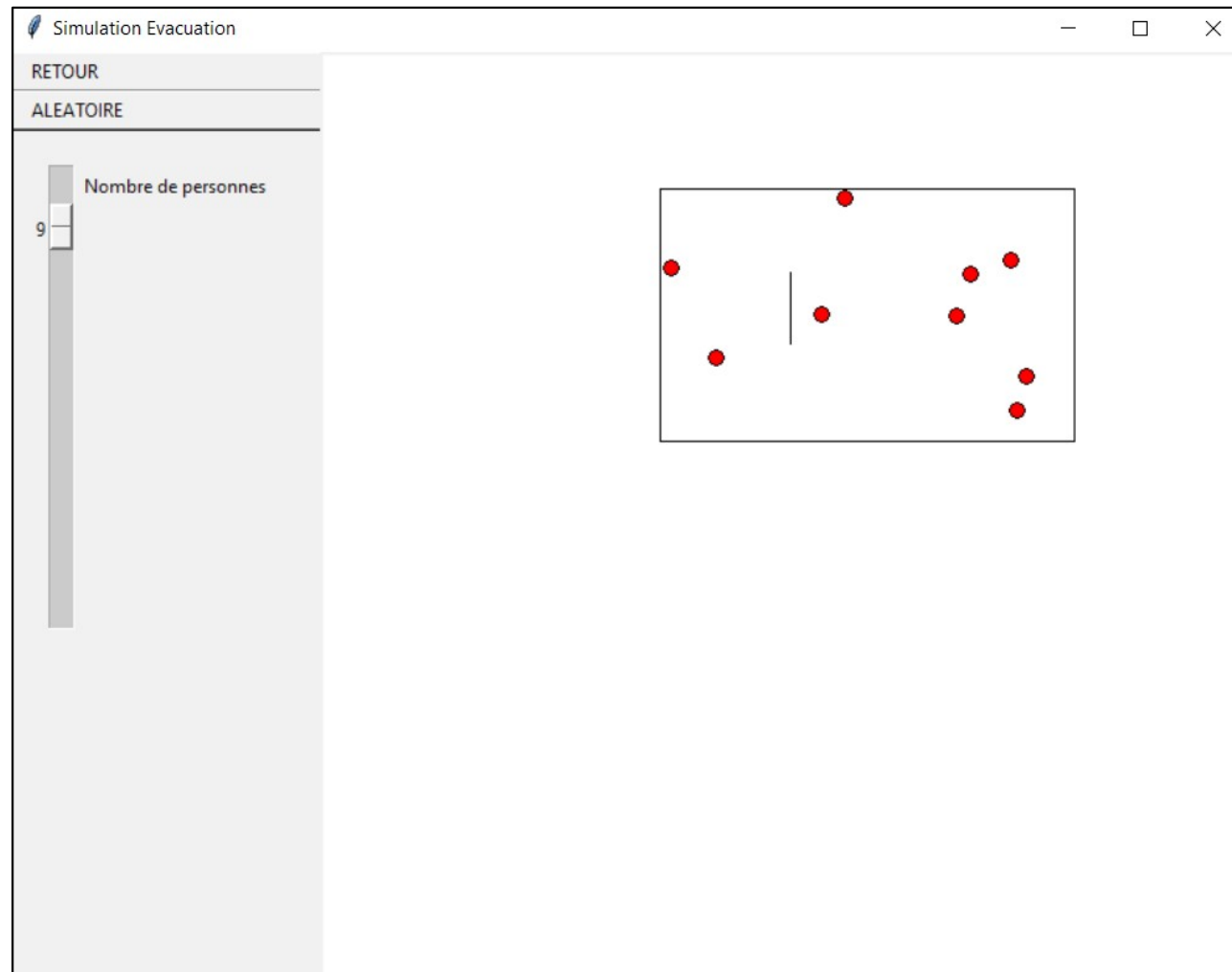
Sommaire

- [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'une évacuation
- [V – Bilan](#)

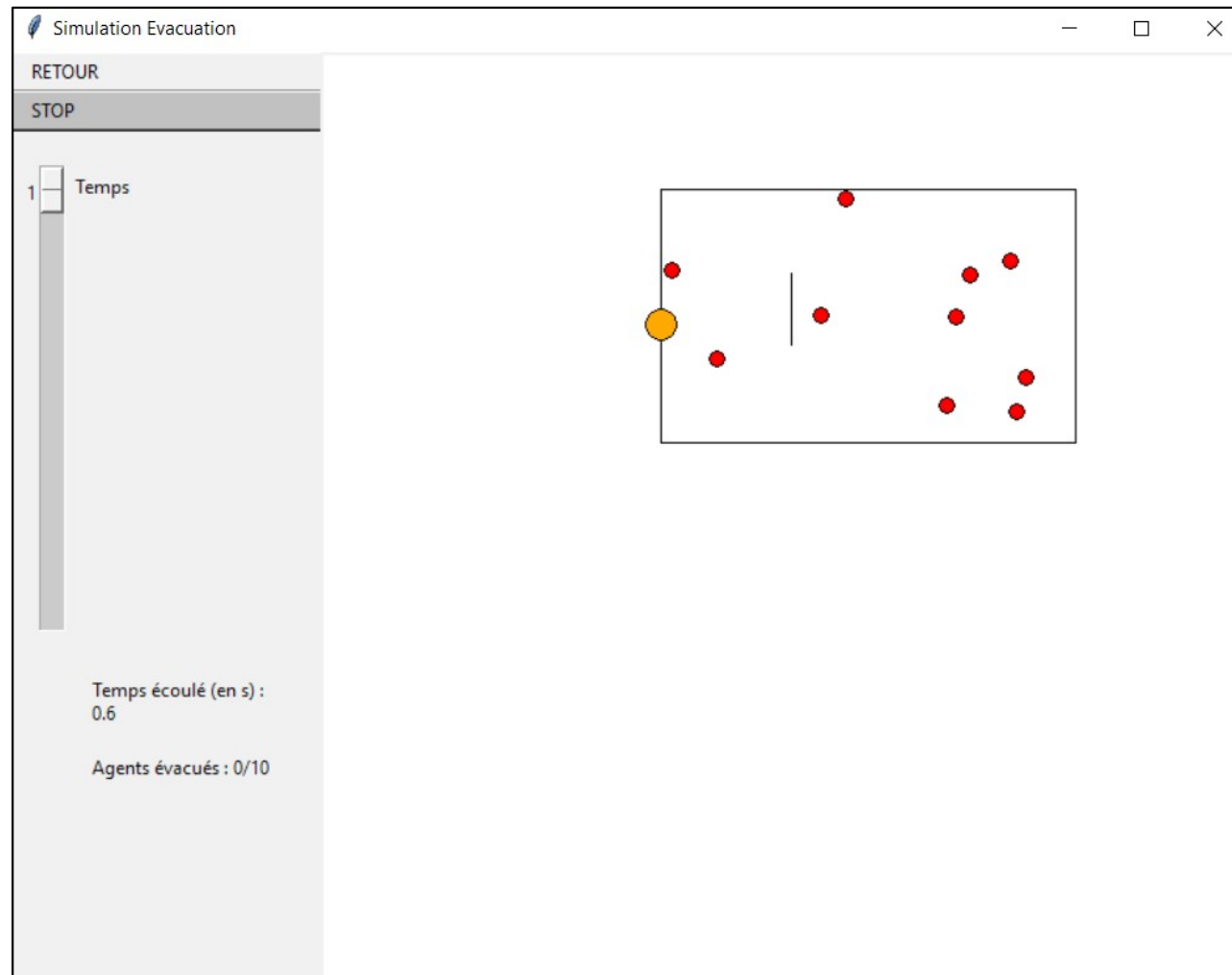
I – Présentation du simulateur



I – Présentation du simulateur

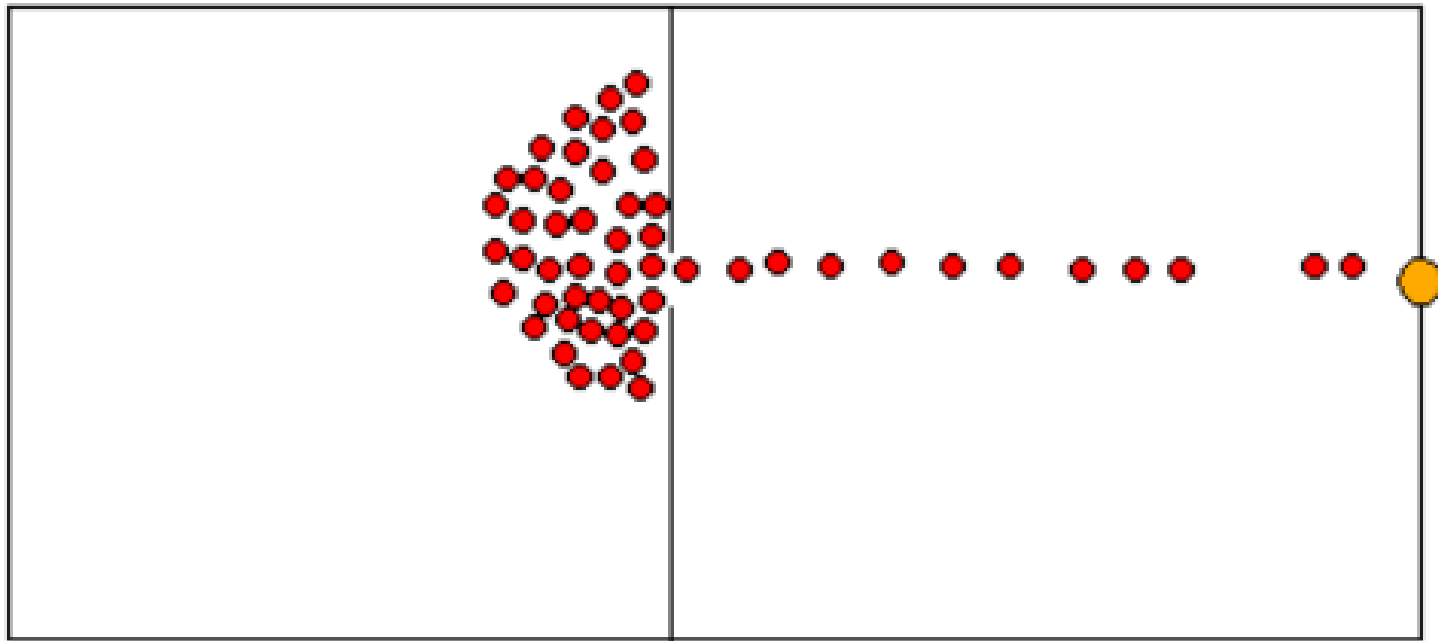


I – Présentation du simulateur

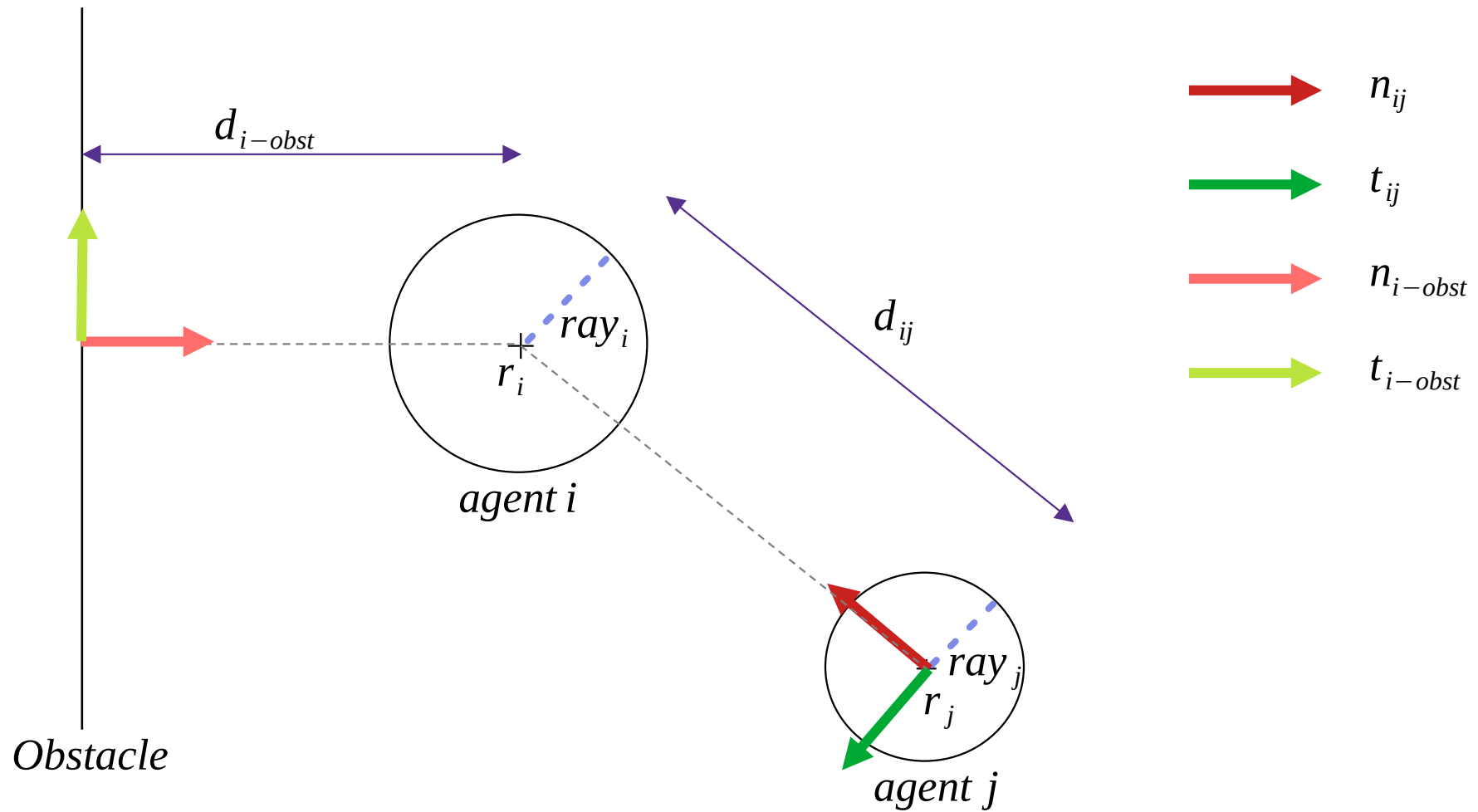


II - Approche générale

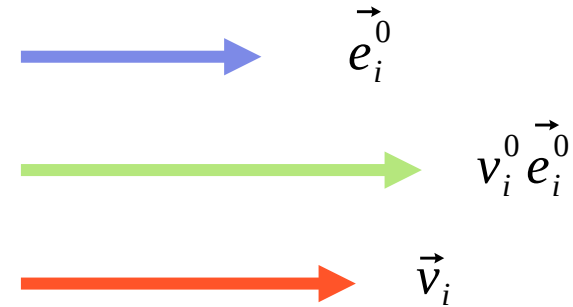
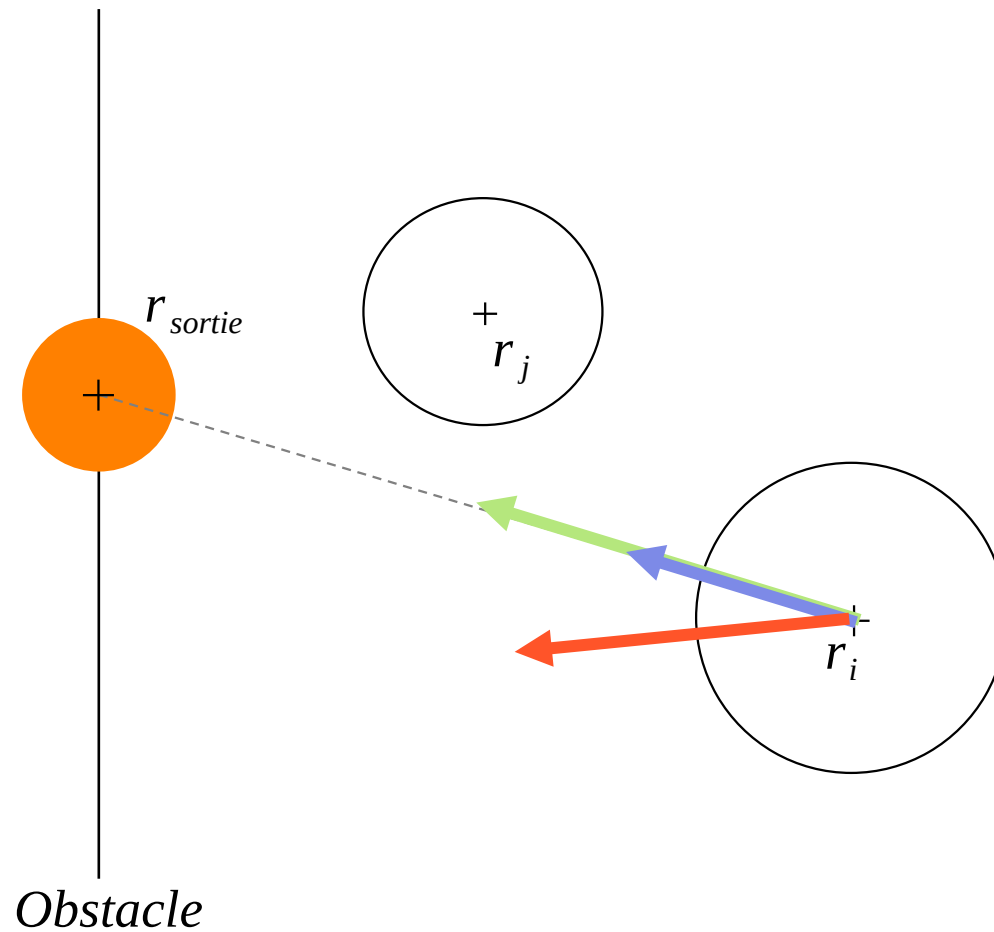
Modélisation Microscopique



Représentation des agents et des murs



Vitesse et direction désirées



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

$$v_i^0 = 1.5 \text{ m} \cdot \text{s}^{-1}$$

Modèle Helbing : SFM

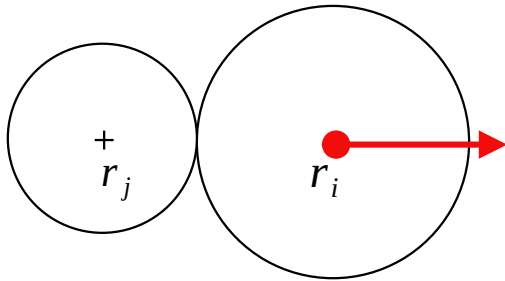
Social Force Model

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

Avec :

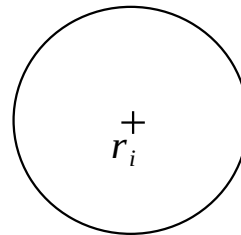
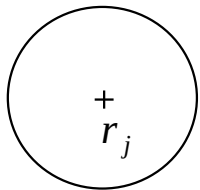
- m_i La masse de l'agent i
- \vec{a}_i L'accélération de l'agent i
- $\vec{f}_{j \rightarrow i}$ Les forces exercées par les autres agents
- $\vec{f}_{obst \rightarrow i}$ Les forces exercées par les obstacles
- $\tau_i = 0.5s$ Le temps de réaction

Force entre deux agents



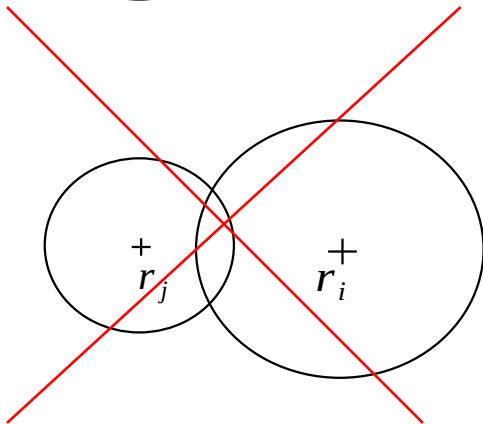
Composante normale pour repousser

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



Pas d'interactions

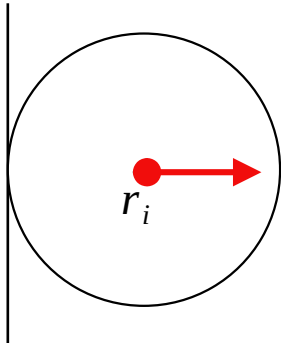
$$\vec{f}_{j \rightarrow i} = \vec{0}$$



Situation à éviter

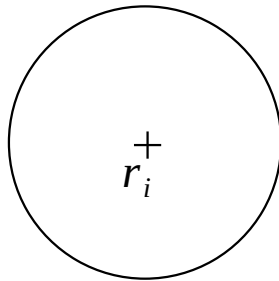
$$A_i = 2 * 10^3 N$$

Force entre un obstacle et l'agent



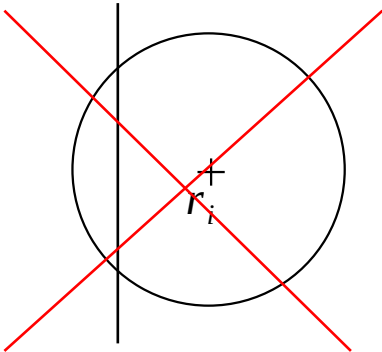
Composante normale pour repousser

$$\vec{f}_{obst \rightarrow i} = A_i h_2(r_i, \vec{v}_i) n_{i-obst}$$



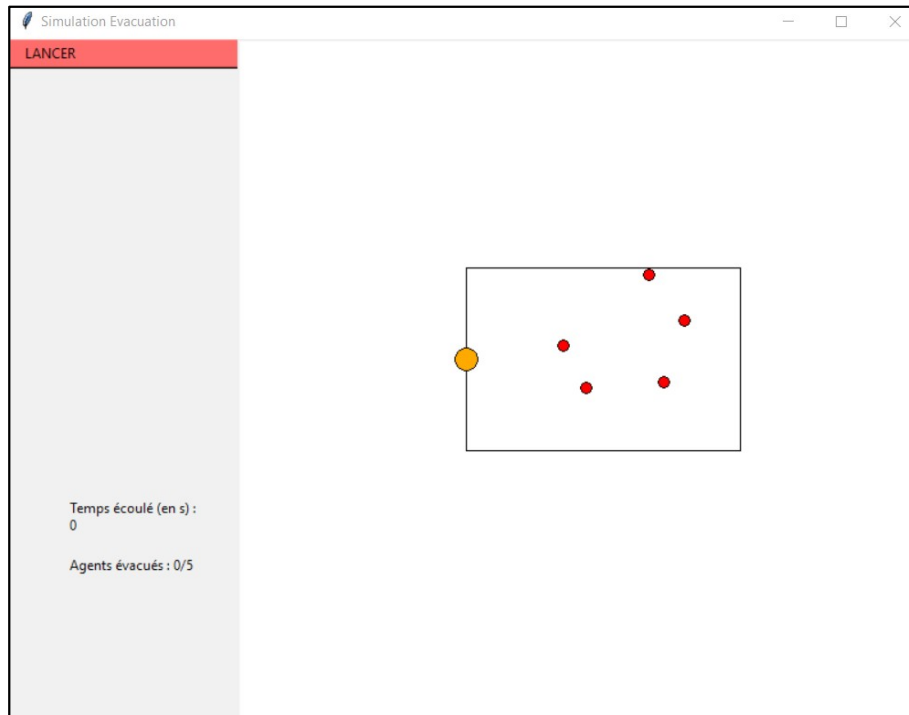
Pas d'interactions

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

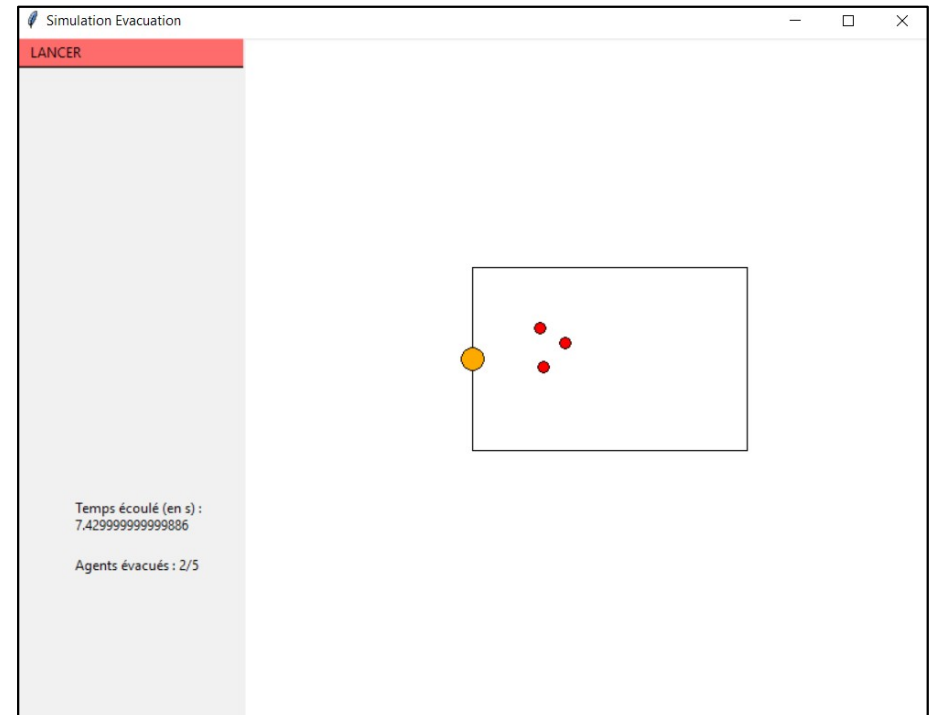


Situation à éviter

Essais à partir du modèle

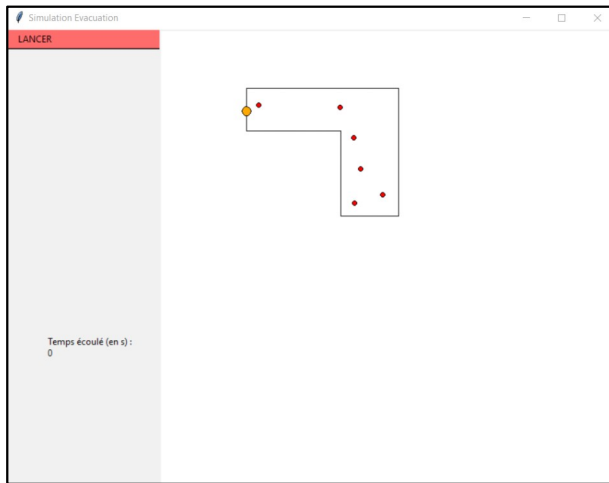


1

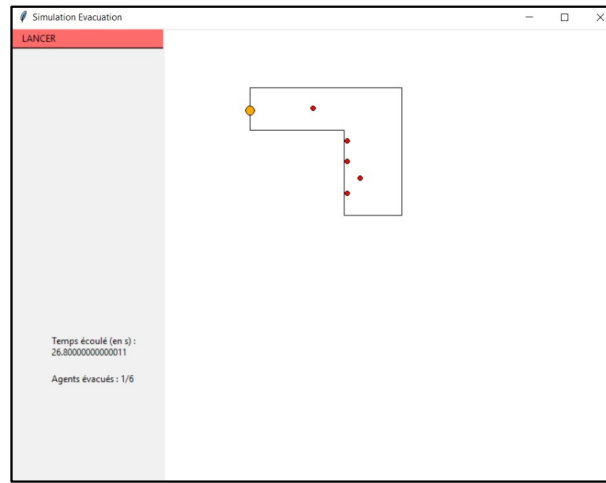


2

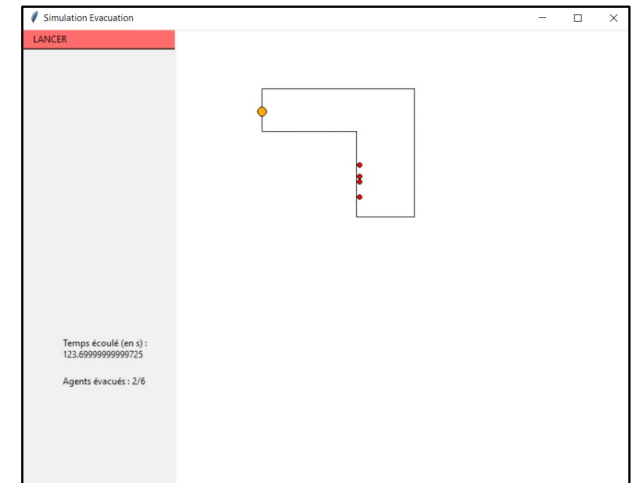
Problèmes physiques observés



1



2



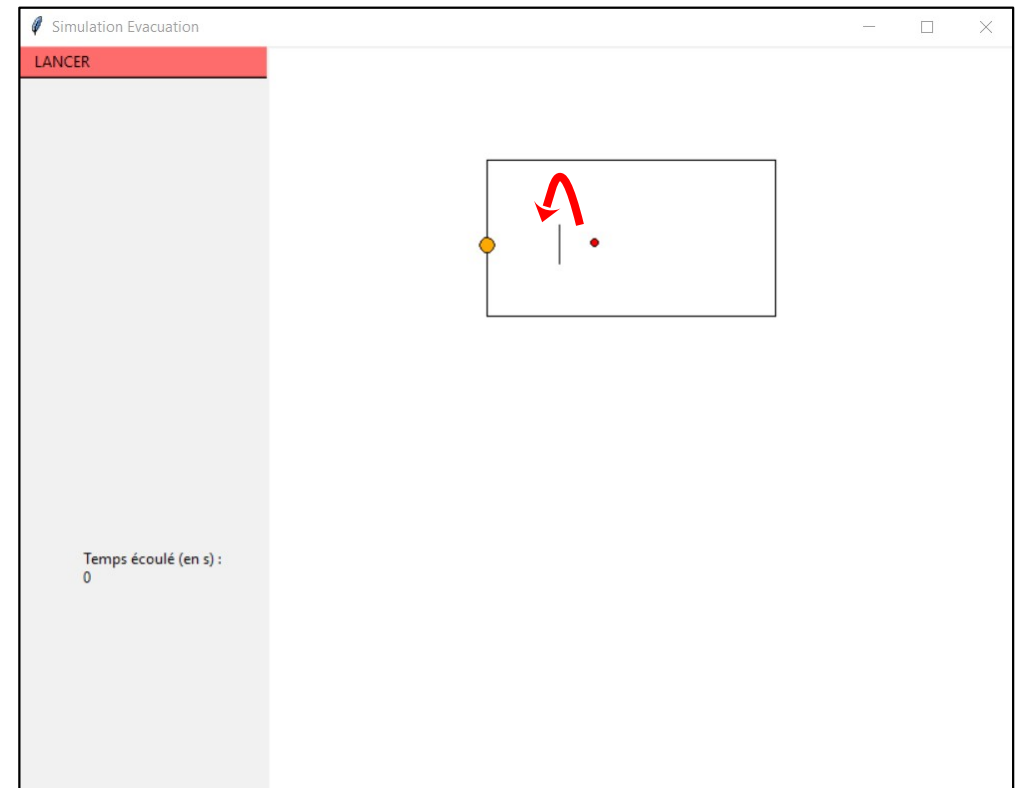
3

II – Modèle CEPABS

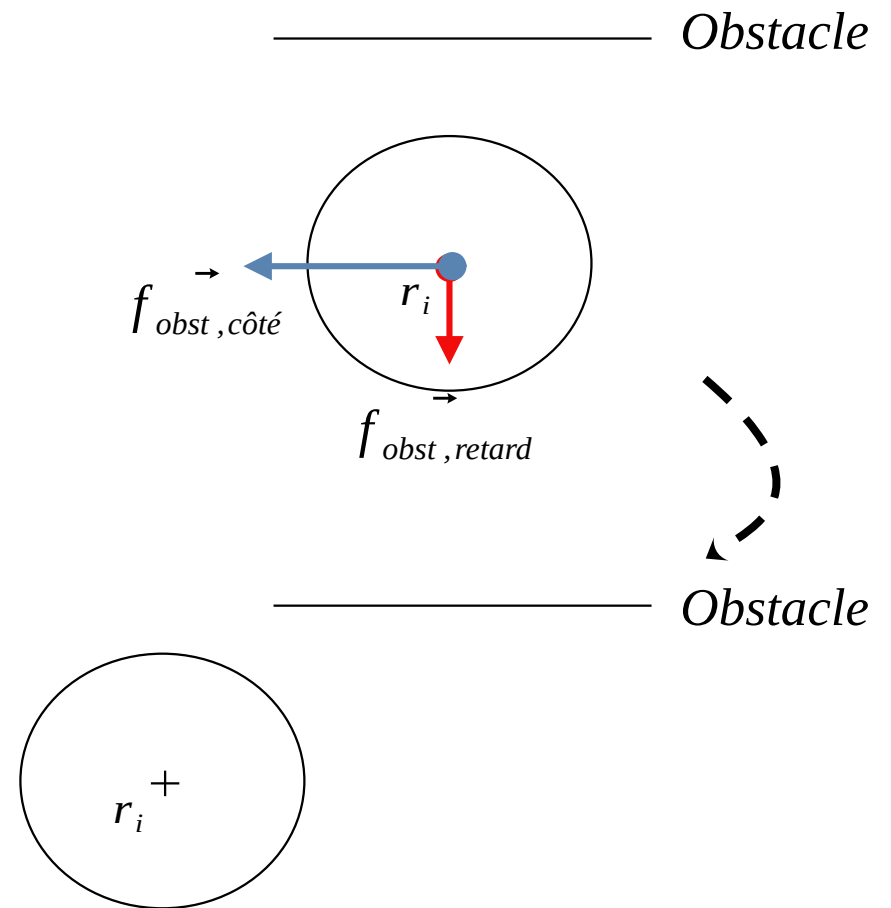
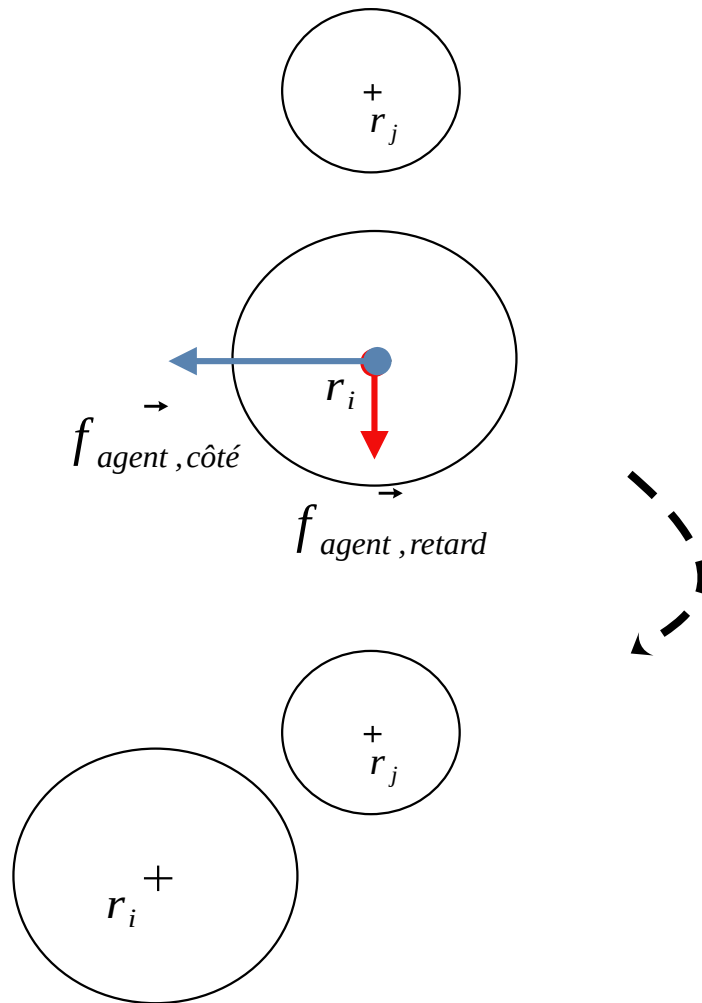
Crowd Evacuation Model Agent Building Simulation

- 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 :

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

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

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

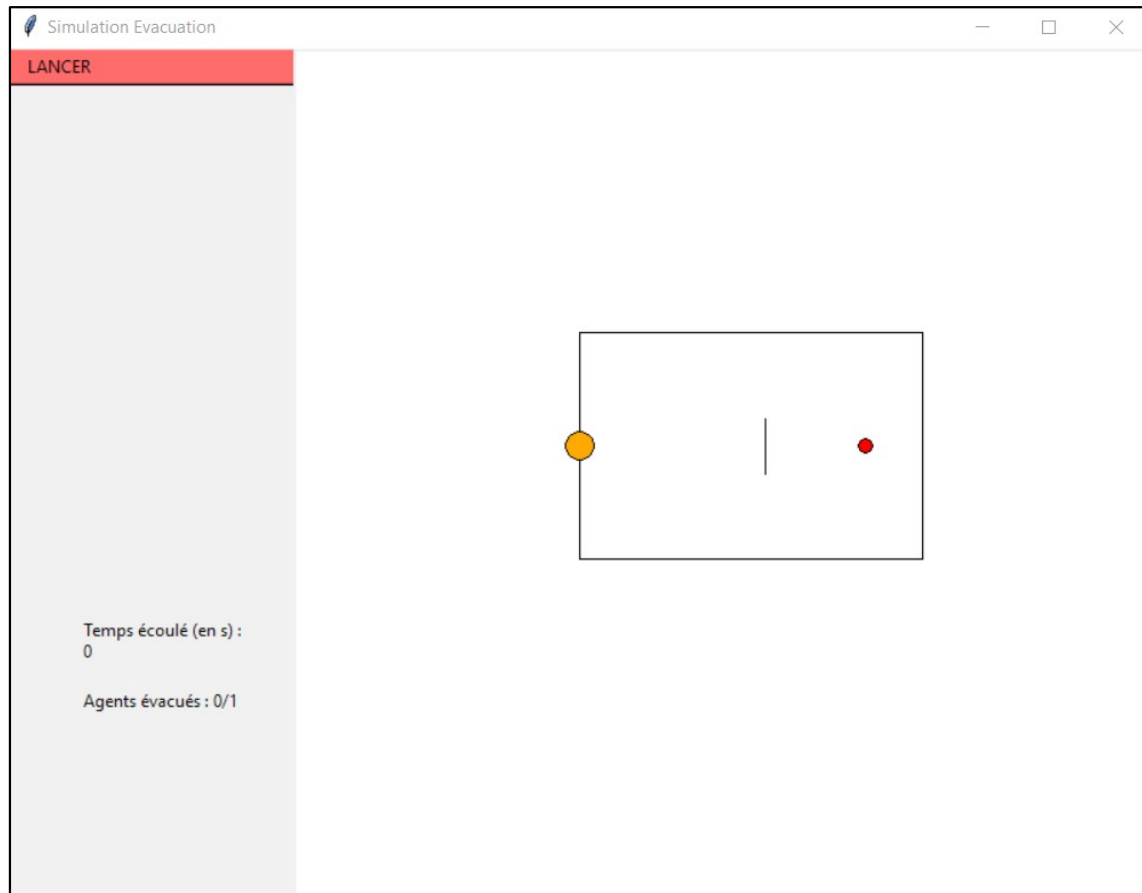
Forces qui obligent l'agent à se déplacer sur le côté :

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

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

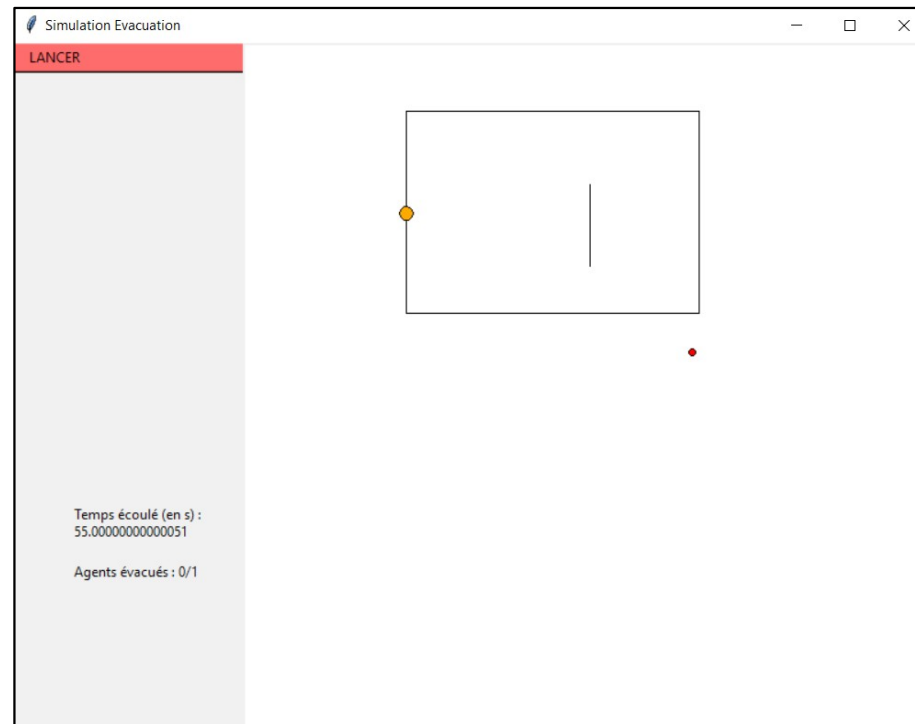
Recherche des constantes

Exemple sur $A_{\text{obst}, \text{retard}}$:

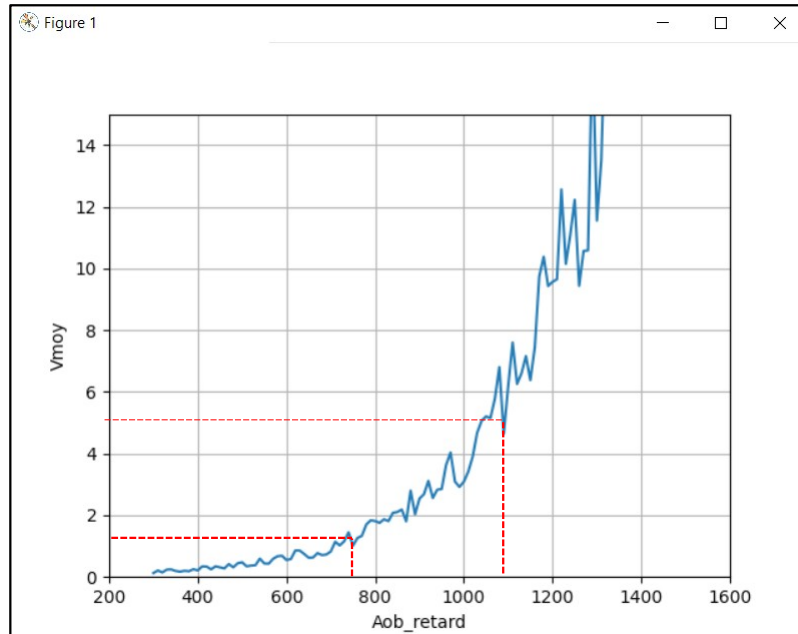


Recherche des constantes

Exemple sur $A_{\text{obst}, \text{retard}}$:

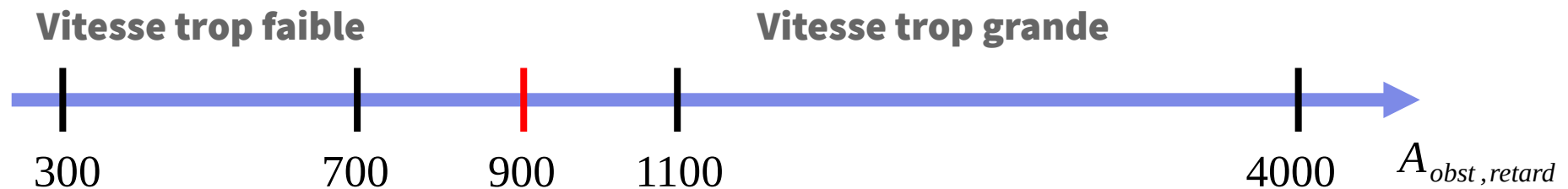


Recherche des constantes

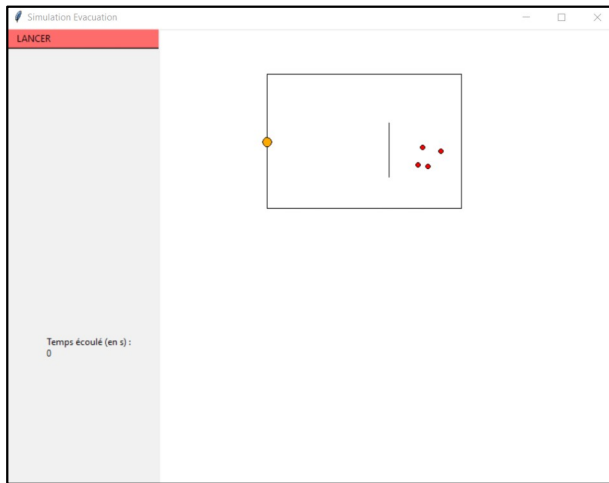


On choisit donc : $A_{obst,retard} = 900 N$

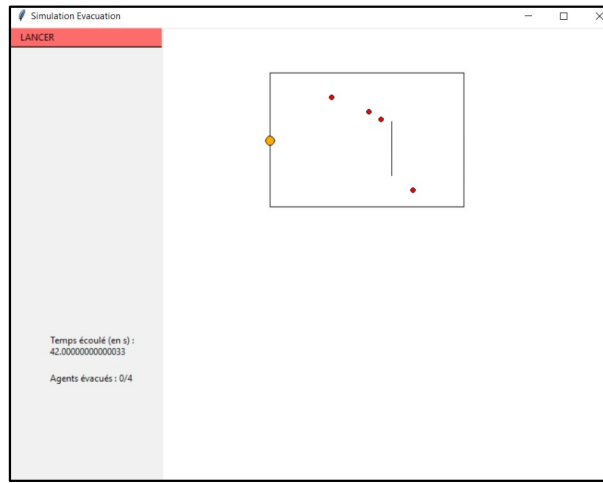
Vitesse moyenne de l'agent en fonction de A



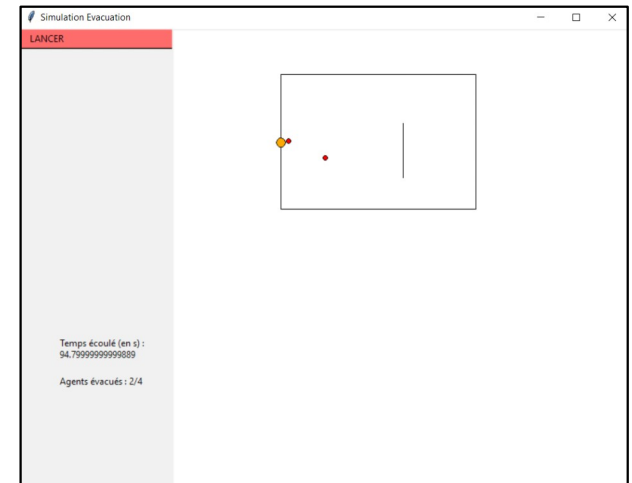
Essais avec le simulateur



1

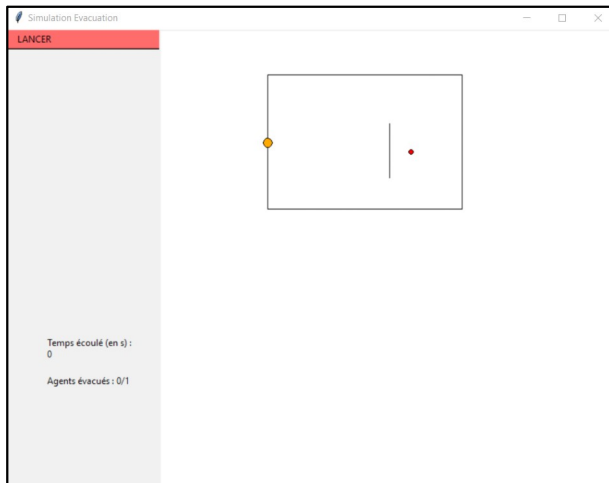


2

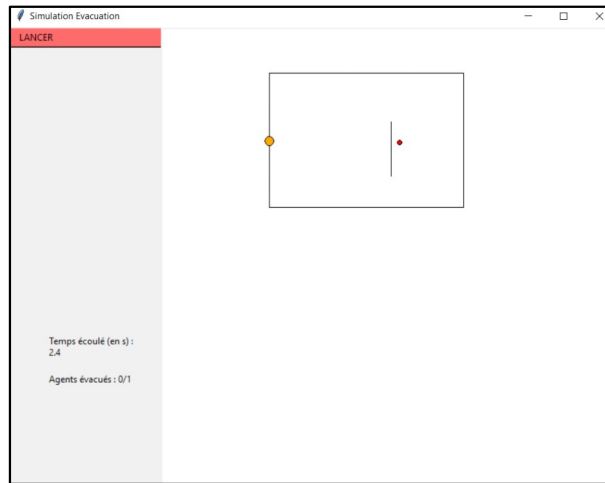


3

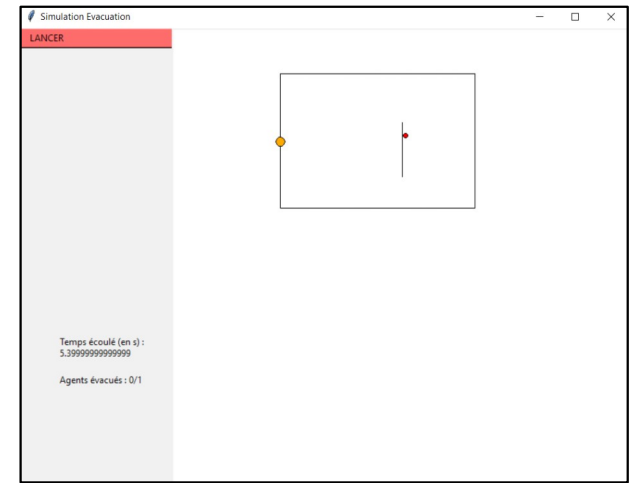
Comportement inattendu



1

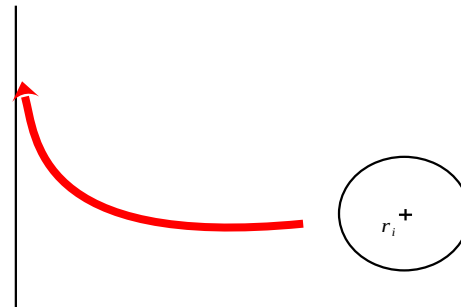


2

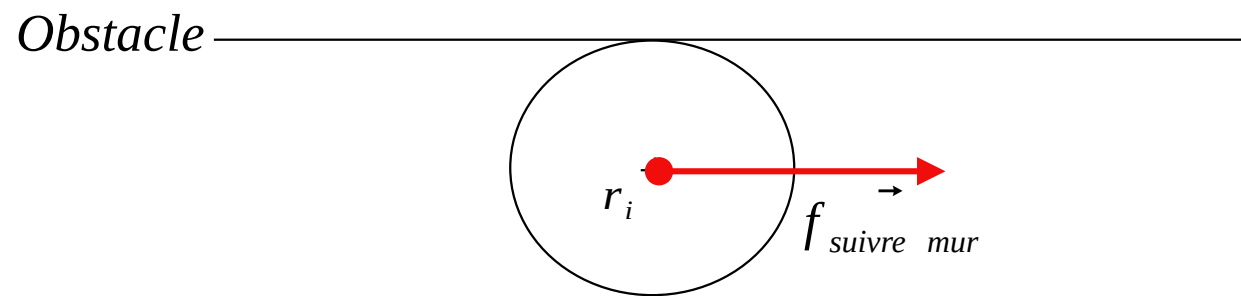


3

Obstacle



Solution



Ajustements

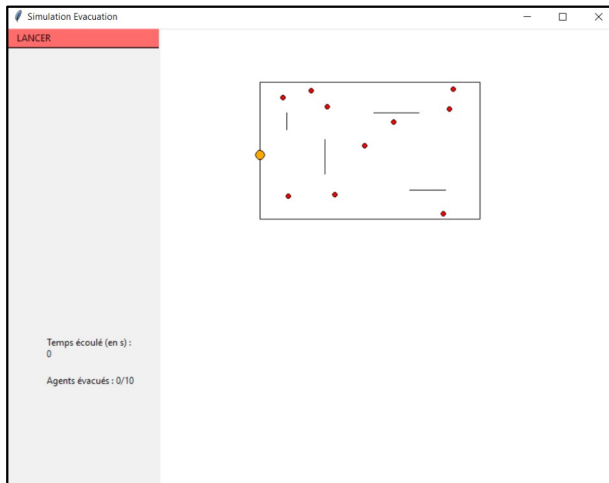
Force qui aide l'agent à contourner l'obstacle :

$$\vec{f}_{\text{suivre mur}} = A_{\text{suivre mur}} h_7(r_i, \vec{v}_i) \vec{t}_{i-\text{obst}}$$

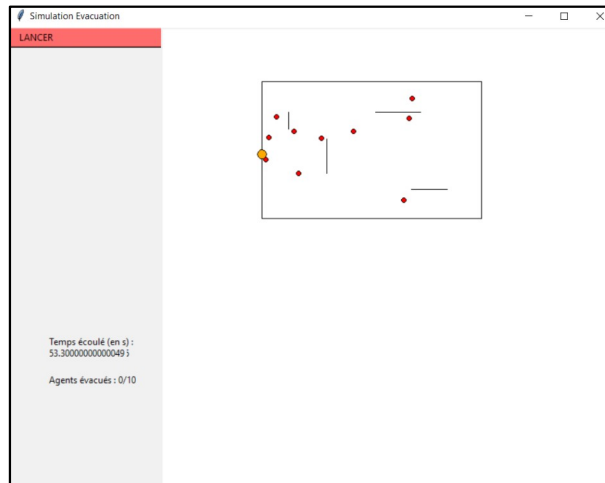
Force qui réduit une trop grande variation de vitesse (~frottements) :

$$\vec{f}_{v, \text{amortissement}} = -\sqrt{\|\vec{v}_i\|} \vec{v}_i$$

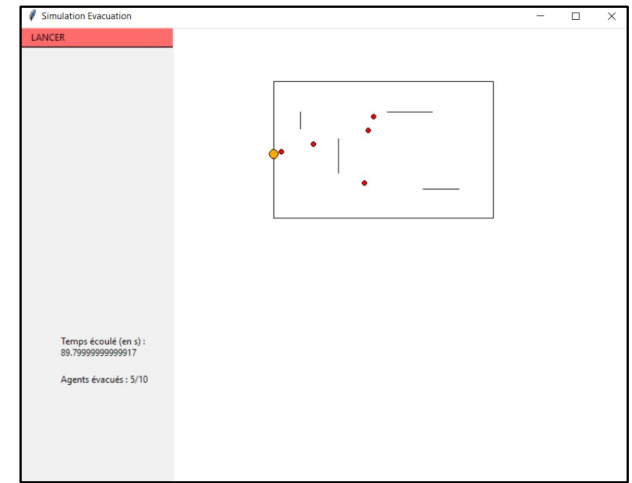
Résultats sur le simulateur



1



2



3

Ajout du modèle comportemental

Panique :

$$\dot{p}_i = c_1(\bar{p}_j - p_i) + c_2(v_i^0 - \|\vec{v}_i\|) + c_3Q + c_4 \quad p_i \in [0, 100]$$

1. Panique des agents autour

2. Frustration provenant de la différence entre la vitesse désirée et la vitesse actuelle

3. Solitude

4. Frustration initiale

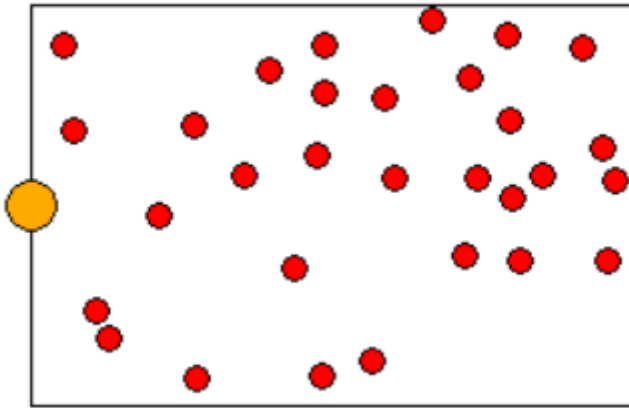
Conséquence sur la vitesse désirée :

$$v_i^0(p_i) = v_{i,initial}^0 \left(1 + \frac{p_i}{p_{max}} \right) \quad p_{max} = 100$$

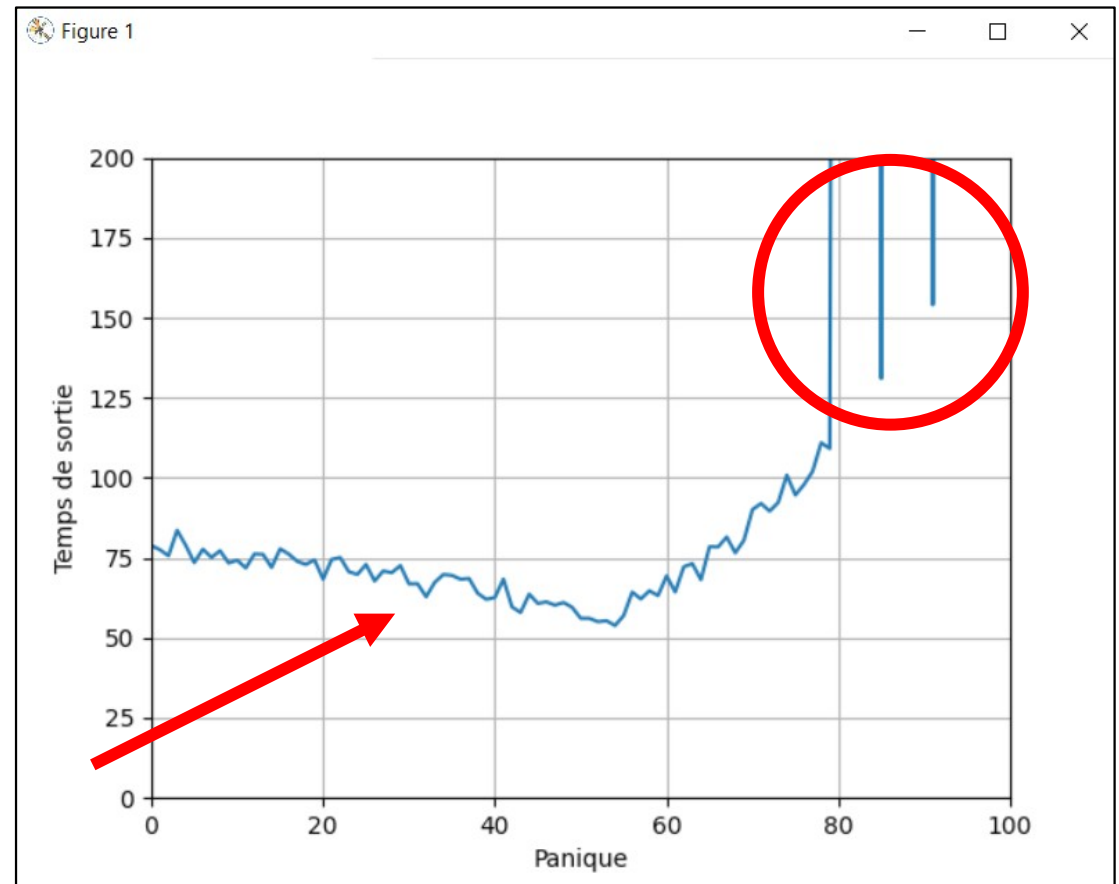
III – Influence des différents paramètres

- **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

Non respect des distanciations sociales :

On veut :

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

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

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

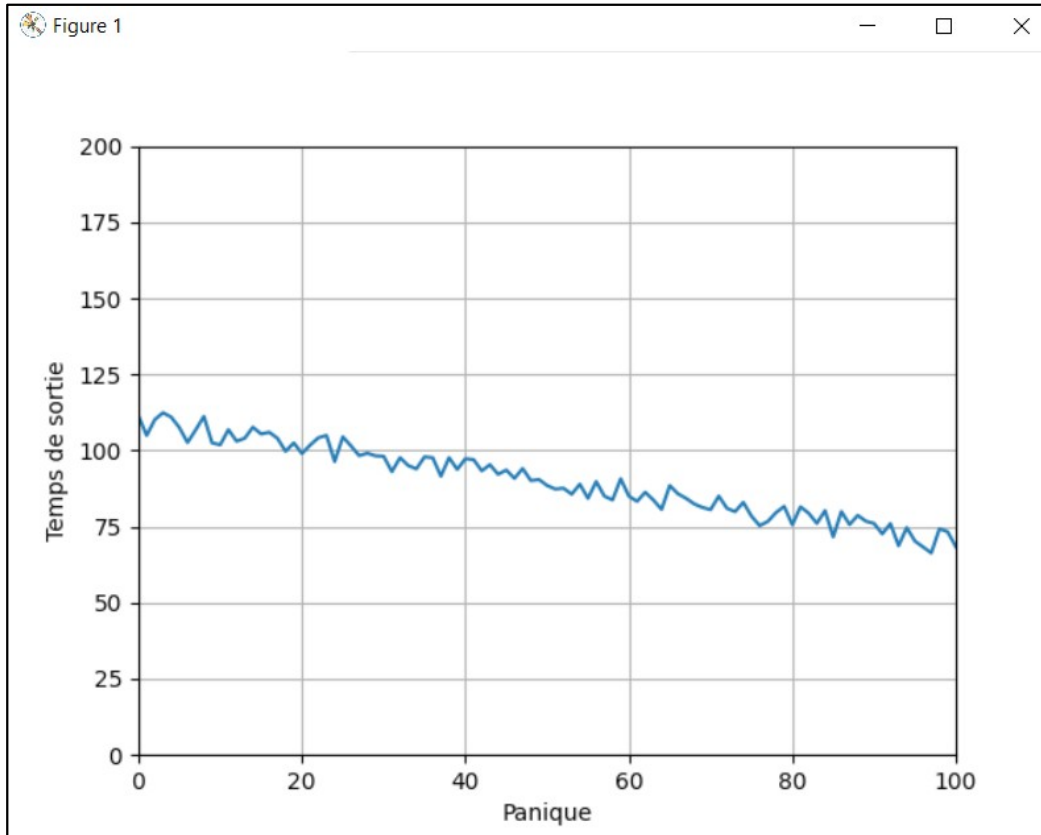
$$A_i(p_i) = -18 p_i + 2 * 10^3$$

Forces modifiées :

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

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

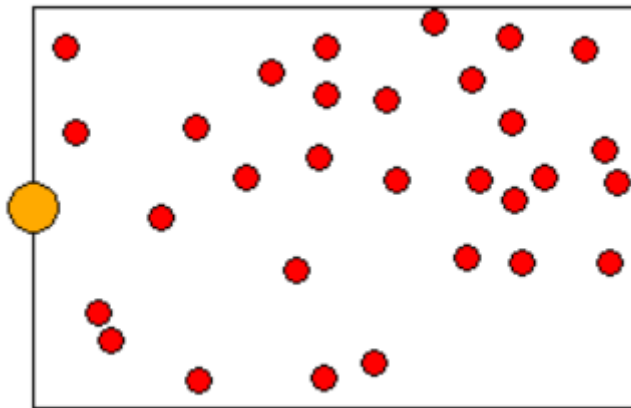
Niveau de panique



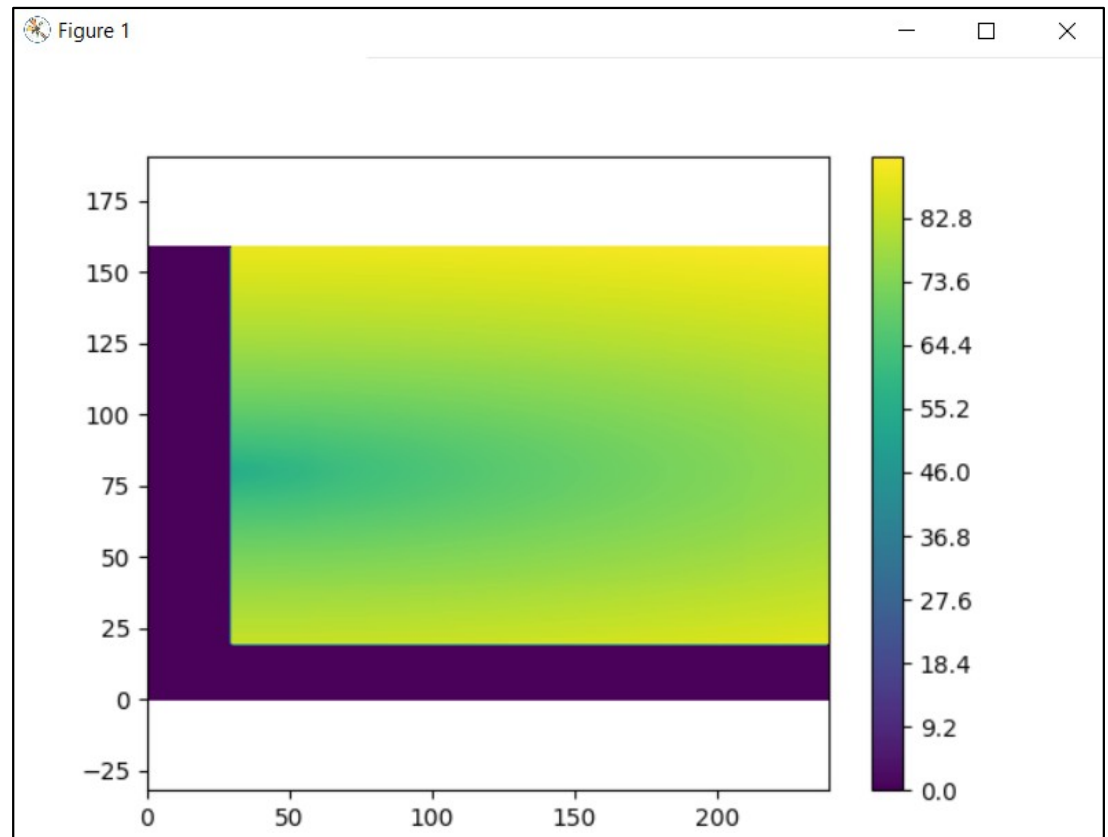
La panique améliore le temps de sortie

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 :

```
380 class Animation(tk.Tk):
381
382     def __init__(self):
383         tk.Tk.__init__(self)
384         self.tool = tk.Frame(self, width = 200, height = 600)
385         self.tool.pack(side = 'left')
386
387         self.graph = tk.Frame(self, width = 600, height = 600, bg = 'white')
388         self.graph.pack(side = 'right')
389
390         self.can = tk.Canvas(self.graph,width = 600, height = 600, bg = 'white')
391         self.can.pack()
392
393         tk.Button(self.tool, text = "    LANCER", command = self.debut, width = 30, anchor = 'w', bg = '#ff6666', activebackground
= '#bdbdbd').place(x = 0, y = 0)
394
395
396
397     def debut(self):
398         self.supprimer_parametres()
399         self.menu()
400
401     ##MENU##
402
403     def menu(self):
404         tk.Button(self.tool, text = "    MURS", command = self.placer_murs, width = 30, anchor = 'w', bg = '#f0f0f0',
activebackground = '#bdbdbd').place(x = 0, y = 25)
405
406         tk.Button(self.tool, text = "    OBSTACLES", command = self.placer_obstacle, width = 30, anchor = 'w', bg = '#f0f0f0',
activebackground = '#bdbdbd').place(x = 0, y = 50)
407
408         tk.Button(self.tool, text = "    PERSONNES", command = self.placer_personnes, width = 30, anchor = 'w', bg = '#f0f0f0',
activebackground = '#bdbdbd').place(x = 0, y = 75)
409
410         tk.Button(self.tool, text = "    SORTIE", command = self.placer_sortie, width = 30, anchor = 'w', bg = '#f0f0f0',
activebackground = '#bdbdbd').place(x = 0, y = 100)
411
412         tk.Button(self.tool, text = "    LANCER", command = self.lancer_simulation, width = 30, anchor = 'w', bg = '#ff6666',
activebackground = '#bdbdbd').place(x = 0, y = 300)
413
414
415
416     def supprimer_parametres(self):
417         L = self.tool.winfo_children()
418         for i in L:
419             i.destroy()
```

Annexe 1

```
422 ##MURS##
423
424 def placer_murs(self): #Bouton 'nettoyer' pour remettre à zero la pièce
425     self.supprimer_parametres()
426     self.can.delete('all')
427     tk.Button(self.tool, text = "    RETOUR", command = self.retour_murs, width = 30, anchor = 'w', bg = '#f0f0f0',
428         activebackground = '#bdbdbd').place(x = 0, y = 0)
429
430     self.peut_placer_murs = False
431
432     self.murs = np.array([[[0, 0], [0, 0]]])
433     self.L_murs_dessins = [0]
434
435     self.can.bind("<Enter>", self.dessiner_ligne_mur)
436     self.can.bind("<Motion>", self.dessiner_ligne_mur)
437     self.can.bind("<Button-1>", self.cliquer_mur)
438
439 def cliquer_mur(self, event):
440     x1, y1 = int(event.x), int(event.y)
441     k = False
442     if not self.peut_placer_murs:
443         self.peut_placer_murs = True
444         self.x, self.y = x1, y1
445
446     elif self.peut_placer_murs:
447         if abs(self.x - x1) < abs(self.y - y1):
448             for i in self.murs:
449                 if abs(y1 - i[0, 1]) < 20:
450                     self.murs = np.vstack([self.murs,[[self.x, self.y], [self.x, i[0, 1]]]])
451                     self.y = i[0, 1]
452                     k = True
453                     break
454                 if not k:
455                     self.murs = np.vstack([self.murs,[[self.x, self.y], [self.x, y1]]])
456                     self.y = y1
457             else :
458                 for i in self.murs:
459                     if abs(x1 - i[0, 0]) < 20:
460                         self.murs = np.vstack([self.murs,[[self.x, self.y], [i[0, 0], self.y]]])
461                         self.x = i[0, 0]
462                         k = True
463                         break
464                 if not k:
465                     self.murs = np.vstack([self.murs,[[self.x, self.y], [x1, self.y]]])
466                     self.x = x1
467
468     self.L_murs_dessins.append(0)
469
470     if self.murs[0].all() == 0:
471         self.murs = np.delete(self.murs, 0, axis = 0)
```

Annexe 1

```
473         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:
474             self.peut_placer_murs = False
475
476
477     #Dessiner les lignes du mur quand la souris bouge
478     def dessiner_ligne_mur(self, event):
479         x1, y1 = int(event.x), int(event.y)
480         if self.peut_placer_murs:
481             self.can.delete(self.L_murs_dessins[-1])
482
483             if abs(self.x - x1) < abs(self.y - y1):
484                 self.L_murs_dessins[-1] = self.can.create_line(self.x, self.y, self.x, y1)
485                 for i in self.murs:
486                     if abs(y1 - i[0, 1]) < 20:
487                         self.can.delete(self.L_murs_dessins[-1])
488                         self.L_murs_dessins[-1] = self.can.create_line(self.x, self.y, self.x, i[0, 1])
489
490             else :
491                 self.L_murs_dessins[-1] = self.can.create_line(self.x, self.y, x1, self.y)
492                 for i in self.murs:
493                     if abs(x1 - i[0, 0]) < 20:
494                         self.can.delete(self.L_murs_dessins[-1])
495                         self.L_murs_dessins[-1] = self.can.create_line(self.x, self.y, i[0, 0], self.y)
496
497
498     def retour_murs(self):
499         self.supprimer_parametres()
500         self.can.unbind("<Enter>")
501         self.can.unbind("<Motion>")
502         self.can.unbind("<Button-1>")
503         self.dmin = np.linalg.norm(self.murs[0, 1, :] - self.murs[0, 0, :])
504         for i in self.murs:
505             d = np.linalg.norm(i[1] - i[0])
506             if d < self.dmin :
507                 self.dmin = d
508         self.menu()
509
```

Annexe 1

```
512     ##OBSTACLES##
513
514
515     def placer_obstacle(self):
516         self.supprimer_parametres()
517         tk.Button(self.tool, text = "    RETOUR", command = self.retour_obstacle, width = 30, anchor = 'w', bg = '#f0f0f0',
activebackground = '#bdbdbd').place(x = 0, y = 0)
518
519         self.peut_placer_obstacle = False
520         self.l_obstacles_dessins = [0]
521         self.obstacles = np.array([[[0, 0], [0, 0]]])
522
523         self.can.bind("<Enter>", self.dessiner_ligne_obstacle)
524         self.can.bind("<Motion>", self.dessiner_ligne_obstacle)
525         self.can.bind("<Button-1>", self.cliquer_obstacle)
526
527
528     def cliquer_obstacle(self, event):
529         x1, y1 = int(event.x), int(event.y)
530         k = False
531         if not self.peut_placer_obstacle:
532             self.peut_placer_obstacle = True
533             for i in self.obstacles:
534                 if abs(y1 - i[0, 1]) < 20:
535                     self.y = i[0, 1]
536                     k = True
537                     break
538             if not k:
539                 self.y = y1
540             k = False
541             for i in self.obstacles:
542                 if abs(x1 - i[0, 0]) < 20:
543                     self.x = i[0, 0]
544                     k = True
545                     break
546             if not k:
547                 self.x = x1
548
549         elif self.peut_placer_obstacle:
550             self.peut_placer_obstacle = False
551
552             if abs(self.x - x1) < abs(self.y - y1):
553                 for i in self.obstacles:
554                     if abs(y1 - i[0, 1]) < 20:
555                         self.obstacles = np.vstack([self.obstacles, [[self.x, self.y], [self.x, i[0, 1]]]])
556                         k = True
557                         break
558             if not k:
559                 self.obstacles = np.vstack([self.obstacles, [[self.x, self.y], [self.x, y1]]])
```

Annexe 1

```
560         else :
561             for i in self.obstacles:
562                 if abs(x1 - i[0, 0]) < 20:
563                     self.obstacles = np.vstack([self.obstacles, [[self.x, self.y], [i[0, 0], self.y]]])
564                     k = True
565                     break
566             if not k:
567                 self.obstacles = np.vstack([self.obstacles, [[self.x, self.y], [x1, self.y]]])
568
569         self.L_obstacles_dessins.append(0)
570
571         if self.obstacles[0].all() == 0:
572             self.obstacles = np.delete(self.obstacles, 0, axis = 0)
573
574     def dessiner_ligne_obstacle(self, event):
575         if self.peut_placer_obstacle:
576             x1, y1 = int(event.x), int(event.y)
577             self.can.delete(self.L_obstacles_dessins[-1])
578
579             if abs(self.x - x1) < abs(self.y - y1):
580                 self.L_obstacles_dessins[-1] = self.can.create_line(self.x, self.y, self.x, y1)
581                 for i in self.obstacles:
582                     if abs(y1 - i[0, 1]) < 20:
583                         self.can.delete(self.L_obstacles_dessins[-1])
584                         self.L_obstacles_dessins[-1] = self.can.create_line(self.x, self.y, self.x, i[0, 1])
585
586             else :
587                 self.L_obstacles_dessins[-1] = self.can.create_line(self.x, self.y, x1, self.y)
588                 for i in self.obstacles:
589                     if abs(x1 - i[0, 0]) < 20:
590                         self.can.delete(self.L_obstacles_dessins[-1])
591                         self.L_obstacles_dessins[-1] = self.can.create_line(self.x, self.y, i[0, 0], self.y)
592
593     def retour_obstacle(self):
594         self.can.unbind("<Enter>")
595         self.can.unbind("<Motion>")
596         self.can.unbind("<Button-1>")
597         self.supprimer_parametres()
598         self.menu()
599
600
601
```

Annexe 1

```
604 ##PERSONNES##
605
606 #Placer les personnes "en gros" -> Pas précisément
607 #Curseur "intensité des interactions, ...
608
609 def placer_personnes(self):
610     self.supprimer_parametres()
611     tk.Button(self.tool, text = "    RETOUR", command = self.retour_personnes, width = 30, anchor = 'w', bg = '#f0f0f0',
activebackground = '#bdbdbd').place(x = 0, y = 0)
612
613     tk.Button(self.tool, text = "    ALEATOIRE", command = self.placer_personnes_aleatoire, width = 30, anchor = 'w', bg =
'#f0f0f0', activebackground = '#bdbdbd').place(x = 0, y = 25)
614
615     self.nb_pers = tk.IntVar()
616     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)
617
618     self.N = self.nb_pers.get()
619     self.r = np.zeros((self.N, 2))
620
621     self.L_personnes_dessin = []
622
623     self.can.bind('<Up>', self.pers_moins)
624     self.can.bind('<Down>', self.pers_plus)
625     self.can.bind('<Button-1>', self.placer_personnes_manuel)
626
627
628 def pers_moins(self, event):
629     x = self.nb_pers.get()
630     self.nb_pers.set(x-1)
631
632 def pers_plus(self, event):
633     x = self.nb_pers.get()
634     self.nb_pers.set(x+1)
635
636
637 def test_peut_placer_personnes(self, x, y):
638     peut_placer_personnes = True
639     for i in self.r:
640         if ((i[0]-x)**2)+(i[1]-y)**2**0.5 <= 2*rayon:
641             peut_placer_personnes = False
642             continue
643
644     point = np.array([x, y])
645
646     for i in self.murs :
647         temp = np.dot(point - i[0, :], i[1, :] - i[0, :]) / (np.linalg.norm(i[1, :] - i[0, :]))**2
648         temp = min( 0, temp ), 1 )
649         proche = i[0, :] + (i[1, :] - i[0, :])*temp
650         d = np.linalg.norm(proche - point)
651         if d<=rayon :
652             peut_placer_personnes = False
```


Annexe 1

```
654     for i in self.obstacles :
655         temp = np.dot(point - i[0, :], i[1, :] - i[0, :]) / (np.linalg.norm(i[1, :] - i[0, :]))**2
656         temp = min( max( 0, temp ), 1 )
657         proche = i[0, :] + (i[1, :] - i[0, :])*temp
658         d = np.linalg.norm(proche - point)
659         if d<=rayon :
660             peut_placer_personnes = False
661
662     k = 0
663     for i in self.murs:
664         if i[0, 1] == i[1, 1]:
665             if i[0, 1] < y:
666                 if (i[0, 0] - x) * (i[1, 0] - x)<0:
667                     k+=1
668     if k%2==0:
669         peut_placer_personnes = False
670
671     return peut_placer_personnes
672
673
674 def placer_personnes_manuel(self, event):
675     x = int(event.x)
676     y = int(event.y)
677     if self.test_peut_placer_personnes(x, y):
678         self.r = np.vstack([self.r, [[x, y]]])
679         self.L_personnes_dessin.append(self.can.create_circle(x, y, rayon, fill = "red"))
680         i = self.nb_pers.get()
681         self.nb_pers.set(i+1)
682         self.N += 1
683
684
685 def placer_personnes_aleatoire(self):
686     for i in self.L_personnes_dessin:
687         self.can.delete(i)
688
689     self.N = self.nb_pers.get()
690     k = 0
691     self.r = np.zeros((self.N, 2))
692     self.L_personnes_dessin = [0]*self.N
693     while k<self.N:
694         x = np.random.randint(rayon, 600-rayon)
695         y = np.random.randint(rayon, 600-rayon)
696
697         if self.test_peut_placer_personnes(x, y):
698             self.r[k, :] = [x, y]
699             self.L_personnes_dessin[k] = self.can.create_circle(x, y, rayon, fill = "red")
700             k+=1
```

Annexe 1

```
712 ##SORTIE##
713
714
715 def placer_sortie(self):
716     self.supprimer_parametres()
717     tk.Button(self.tool, text = "    RETOUR", command = self.retour_sortie, width = 30, anchor = 'w', bg = '#f0f0f0',
718 activebackground = '#bdbdbd').place(x = 0, y = 0)
719
720     self.longueur_sortie = tk.IntVar()
721     tk.Scale(self.tool, orient = "vertical", label = "Taille de la sortie", length = 300, from_ = 2*rayon, to = self.dmin/2,
722 variable = self.longueur_sortie).place(x = 0, y = 70)
723
724     self.s = self.longueur_sortie.get()
725     self.peut_placer_sortie = True
726
727     self.can.bind('<Up>', self.sortie_moins)
728     self.can.bind('<Down>', self.sortie_plus)
729
730     self.can.bind("&<Enter>", self.bouger_sortie)
731     self.can.bind("<Motion>", self.bouger_sortie)
732     self.can.bind("<Button-1>", self.test_peut_placer_sortie)
733     self.dot = self.can.create_circle(self.murs[0, 0, 0], self.murs[0, 0, 1], self.s, fill = "yellow")
734
735 def sortie_moins(self, event):
736     x = self.longueur_sortie.get()
737     self.longueur_sortie.set(x-1)
738
739 def sortie_plus(self, event):
740     x = self.longueur_sortie.get()
741     self.longueur_sortie.set(x+1)
742
743 def bouger_sortie(self, event):
744     self.s = self.longueur_sortie.get()
745     if self.peut_placer_sortie:
746         x, y = int(event.x), int(event.y)
747         self.can.delete(self.dot)
748
749         A = np.array([[x, y]])
750         temp = np.dot(A - self.murs[0, 0, :], self.murs[0, 1, :] - self.murs[0, 0, :]) / (np.linalg.norm(self.murs[0, 1, :] -
751 self.murs[0, 0, :]))**2
752         temp = min( max(self.s/np.linalg.norm(self.murs[0, 1, :] - self.murs[0, 0, :]), temp ), 1 - self.s/
753 np.linalg.norm(self.murs[0, 1, :] - self.murs[0, 0, :]) )
754         proche = self.murs[0, 0, :] + (self.murs[0, 1, :] - self.murs[0, 0, :])*temp
755         d = np.linalg.norm(proche - A)
756         for j in range(1, len(self.murs)):
757             temp = np.dot(A - self.murs[j, 0, :], self.murs[j, 1, :] - self.murs[j, 0, :]) / (np.linalg.norm(self.murs[j, 1,
758 :] - self.murs[j, 0, :]))**2
759             temp = min( max(self.s/np.linalg.norm(self.murs[j, 1, :] - self.murs[j, 0, :]), temp ), 1 - self.s/
760 np.linalg.norm(self.murs[j, 1, :] - self.murs[j, 0, :]) )
761             prochel = self.murs[j, 0, :] + (self.murs[j, 1, :] - self.murs[j, 0, :])*temp
```

Annexe 1

```
758         d1 = np.linalg.norm(proche1 - A)
759         if d1 < d:
760             d = d1
761             proche = proche1
762         self.dot = self.can.create_circle(proche[0], proche[1], self.s, fill = "orange")
763
764
765     def test_peut_placer_sortie(self, event):
766         x, y = int(event.x), int(event.y)
767         if self.peut_placer_sortie:
768             self.cliquer_pour_placer_sortie(x, y)
769             self.peut_placer_sortie = False
770         else:
771             self.peut_placer_sortie = True
772
773
774     def cliquer_pour_placer_sortie(self, x, y):
775         i = 0
776
777         A = np.array([[x, y]])
778         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)
779         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, :]) )
780         proche = self.murs[0, 0, :] + (self.murs[0, 1, :] - self.murs[0, 0, :])*temp
781         d = np.linalg.norm(proche - A)
782
783         for j in range(len(self.murs)):
784             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)
785             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, :]))
786
787             proche1 = self.murs[j, 0, :] + (self.murs[j, 1, :] - self.murs[j, 0, :])*temp
788             d1 = np.linalg.norm(proche1 - A)
789             if d1 < d:
790                 i = j
791                 d = d1
792                 proche = proche1
793
794
795         self.sortie = proche
796         self.murs1 = np.delete(self.murs, i, 0)
797         if self.murs[i, 0, 0] == self.murs[i, 1, 0]:
798             if self.murs[i, 0, 1] < self.murs[i, 1, 1]:
799                 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]]])
800                 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

```
801         elif self.murs[i, 0, 1] > self.murs[i, 1, 1]:
802             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]]))
803             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]]]))
804         elif self.murs[i, 0, 1] == self.murs[i, 1, 1]:
805             if self.murs[i, 0, 0] < self.murs[i, 1, 0]:
806                 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]]]))
807                 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]]]))
808             elif self.murs[i, 0, 0] > self.murs[i, 1, 0]:
809                 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]]]))
810                 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]]]))
811
812
813     def retour_sortie(self):
814         self.supprimer_parametres()
815         self.murs = np.copy(self.murs1)
816         self.sortie = np.array(self.sortie)
817         self.can.unbind("<Enter>")
818         self.can.unbind("<Motion>")
819         self.can.unbind("<Button-1>")
820         self.can.unbind('<Up>')
821         self.can.unbind('<Down>')
822         self.menu()
823
```

Annexe 1

```
826     ##SIMULATION##
827
828
829     def lancer_simulation(self):
830         self.supprimer_parametres()
831         tk.Button(self.tool, text = "    RETOUR", command = self.retour_lancer_simulation, width = 30, anchor = 'w', bg =
832         '#f0f0f0', activebackground = '#bdbdbd').place(x = 0, y = 0)
833
834         self.pause = tk.Button(self.tool, text = "    STOP", command = self.anim, width = 30, anchor = 'w', bg = '#f0f0f0',
835         activebackground = '#bdbdbd')
836         self.pause.place(x = 0, y = 25)
837
838         self.attente = tk.IntVar()
839         tk.Scale(self.tool, orient = "vertical", label = "Temps", length = 300, from_ = 1, to = 10, variable =
840         self.attente).place(x = 0, y = 70)
841
842         self.temps_ecoule = tk.StringVar()
843         self.temps_ecoule.set(0)
844         tk.Label(self.tool, textvariable = self.temps_ecoule).place(x = 50, y = 415)
845         tk.Label(self.tool, text = "Temps écoulé (en s) : ").place(x = 50, y = 400)
846
847         self.agents_sortis = tk.StringVar()
848         tk.Label(self.tool, textvariable = self.agents_sortis).place(x = 50, y = 450)
849
850         self.evacue = []
851
852         self.simulation_active = True
853         self.rtot = np.zeros((1, len(self.r), 2))
854         self.rtot[0] = self.r
855         self.pos = Position(len(self.r), rayon, self.sortie, self.murs, self.obstacles, dt)
856         self.Pa = np.zeros((len(self.r)))
857         self.Pa_tot = np.zeros((1, len(self.r)))
858
859         self.can.bind('<Up>', self.temps_moins)
860         self.can.bind('<Down>', self.temps_plus)
861         self.can.bind('<space>', self.animation_pause)
862         self.deplacement()
863
864
865     def retour_lancer_simulation(self):
866         self.simulation_active = False
867         self.supprimer_parametres()
868         self.menu()
869
870
871     def temps_moins(self, event):
872         x = self.attente.get()
873         self.attente.set(x-1)
874
875
876     def temps_plus(self, event):
877         x = self.attente.get()
878         self.attente.set(x+1)
```

Annexe 1

```
876 def animation_pause(self, event):
877     anim()
878
879 def anim(self):
880     if self.simulation_active:
881         self.pause.config(bg = '#bdbdbd', activebackground = '#f0f0f0')
882         self.simulation_active = False
883     else :
884         self.simulation_active = True
885         self.pause.config(bg = '#f0f0f0', activebackground = '#bdbdbd')
886         self.deplacement()
887
888
889
890 def deplacement(self):
891     self.v = self.pos.vit_des(self.r, self.Pa)
892     while self.simulation_active:
893         if len(self.evacue)>=len(self.rtot[0]):
894             self.simulation_active = False
895
896         k = self.temps_ecoule.get()
897         self.temps_ecoule.set(float(k)+dt)
898         r1, v1, Pa1, sortis = self.pos.euler(self.r, self.v, self.Pa)
899
900         for i in sortis:
901             self.evacue.append(i)
902
903         self.agents_sortis.set("Agents évacués : "+str(len(self.evacue))+"/"+str(len(r1)))
904
905         self.rtot = np.vstack([self.rtot, [r1]])
906         for i in range(len(self.r)):
907             self.can.move(self.L_personnes_dessin[i], (r1[i, 0] - self.r[i, 0]), (r1[i, 1] - self.r[i, 1]))
908         self.r = r1
909         self.v = v1
910         #self.Pa = Pa1
911
912
913
914         attente_num = self.attente.get()
915         time.sleep(0.01/attente_num)
916         self.update()
917
918
919
920
921
922 anim = Animation()
923 anim.title("Simulation Evacuation")
924 anim.geometry("800x600")
925
926 anim.mainloop()
```

Annexe 2

Modèle Helbings :

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

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

$$\Delta v_{ij}^t = (\vec{v}_j - \vec{v}_i) \cdot \vec{t}_{ij}$$

$$g(x) = \begin{cases} x & \text{si } x > 0 \\ 0 & \text{sinon} \end{cases}$$

$$\vec{f}_{obst \rightarrow i} = \left[A e^{\frac{ray_i - d_{i-obst}}{B}} + k * g(ray_i - d_{i-obst}) \right] \vec{n}_{i-obst} + \kappa * g(ray_i - d_{i-obst}) \Delta v_{i-obst}^t \vec{t}_{i-obst}$$

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

Annexe 3

Code modèle Helbings :

```
24 ##Modèle de Helbings##
25
26 class Helbings():
27
28     def __init__(self, murs, sortie, N, ray, m, tau, dt):
29         self.A = 2*10**3          # constante (N) : int
30         self.B = 0.08             # constante (m) : int
31         self.k = 1.2*10**5        # parametre (kg/s^2) : int
32         self.kap = 2.4*10**5      # parametre (kg/(m*s^2))
33
34         self.murs = murs
35         self.sortie = sortie
36
37         self.v0 = np.ones((N))*1.5
38         self.ray = ray
39         self.m = m
40
41         self.tau = tau
42         self.dt = dt
43
44
45
46 #Calcul de la force totale exercee sur les agents
47 def f_final_Helbings(self, r, v):
48     a = np.zeros((len(r), 2))
49     v0e0 = self.vit_des_helbings(r)
50     F_ag = self.f_agent_helbings(r, v)
51     F_mur = self.f_mur_helbings(r, v)
52
53     for i in range(len(r)):
54         a[i, :] = ((v0e0[i, :] - v[i, :])/self.tau) + (F_ag[i, :]/self.m[i]) + ((F_mur[i, :])/self.m[i])
55     return a
56
57
58
59 #Calcul de la direction desiree des agents
60 def dir_des(self, r):
61     e0 = np.zeros((len(r), 2))
62     for i in range(len(r)):
63         e0[i, :] = (self.sortie - r[i, :]) / np.linalg.norm(self.sortie - r[i, :])
64     return e0
65
66
67
68 #Calcul de la vitesse desiree des agents
69 def vit_des_helbings(self, r):
70     v0e0 = np.zeros((len(r), 2))
71     e0 = self.dir_des(r)
72     for i in range(len(r)):
73         v0e0[i, :] = self.v0[i]*e0[i, :]
```


Annexe 3

```
77 """f_agents"""
78
79 def g(self, x):
80     if x < 0:
81         return 0
82     return x
83
84
85 #Caracteristiques agent i vers j
86 def agent(self, i, j, r):
87     d = np.linalg.norm(r[i, :] - r[j, :])
88     n = (r[i, :] - r[j, :])/d
89     t = np.array([-n[1], n[0]])
90     return d, n, t
91
92
93 #Force exercee par j sur i
94 def f_ij(self, i, j, r, v):
95     d, n, t = self.agent(i, j, r)
96     dv_t = np.dot(v[j, :] - v[i, :], t)
97     r_tot = self.ray[i] + self.ray[j]
98     a = (self.A * np.exp((r_tot - d)/self.B)) + (self.k * self.g(r_tot - d))
99     b = self.kap * self.g(r_tot - d) * dv_t
100     return a*n + b*t
101
102
103 #Force totale des agents
104 def f_agent_helbings(self, r, v):
105     F = np.zeros((len(r), 2))
106     for i in range(len(r)):
107         for j in range(len(r)):
108             if i != j:
109                 F[i, :] += self.f_ij(i, j, r, v)
110     return F
```

Annexe 3

```
114 """f_mur"""
115
116 #Caracteristiques de l'agent i au segment j (murs ou obstacle)
117 def dist_iM(self, i, r, seg):
118     temp = np.dot(r[i, :] - seg[0, :], seg[1, :] - seg[0, :]) / (np.linalg.norm(seg[1, :] - seg[0, :]))**2
119     temp = min( max( 0, temp ), 1 )
120     proche = seg[0, :] + (seg[1, :] - seg[0, :])*temp
121     d = np.linalg.norm(proche - r[i, :])
122     n = r[i, :] - proche / d
123     t = np.array([-n[1], n[0]])
124     return d, n, t, proche
125
126 #Force exercee par le mur j sur l'agent i
127 def f_iM(self, i, j, r, v):
128     d, n, t = self.dist_iM(i, r, self.murs[j, :, :])[:-1]
129     dv_t = -np.dot(v[i, :], t)
130     a = (self.A * np.exp((self.ray[i] - d)/self.B)) + (self.k * self.g(self.ray[i] - d))
131     b = self.kap * self.g(self.ray[i] - d) * dv_t
132     return a*n + b*t
133
134
135 #Force totale des murs
136 def f_mur_helbings(self, r, v):
137     F = np.zeros((len(r), 2))
138     for i in range(len(r)):
139         for j in range(len(self.murs)):
140             F[i, :] += self.f_iM(i, j, r, v)
141     return F
```

Annexe 3

```
148 #Calcul de la vitesse et de la position des agents à l'instant k
149 def euler_Helbings(self, r, v):
150     a1 = np.zeros((len(r), 2))
151     v1 = np.zeros((len(r), 2))
152     r1 = np.zeros((len(r), 2))
153
154     evacue = []
155
156     a1 = self.f_final_Helbings(r, v)
157     v1 = v + self.dt * a1
158     r1 = r + self.dt * v1
159
160     for i in range(len(r1)):
161         if np.linalg.norm(r1[i, :] - self.sortie) < 2*self.ray[i]:
162             r1[i, :] = np.random.rand(1, 2) * 10**6
163             evacue.append(i)
164
165     return r1, v1, evacue
```

Annexe 4

Modèle CEPABS :

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

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

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

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

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

Annexe 5

Code modèle Helbings :

```
172 ##Modèle CEPABS##
173
174
175 class CEPABS():
176
177     def __init__(self, murs, obstacles, sortie, N, ray, m, tau, dt, p_suivre):
178
179         self.Aag_retard = 900          # constante (à trouver experimentalement)
180         self.Aag_evite = 10             # constante (à trouver experimentalement)
181         self.Aob_retard = 900          # constante (à trouver experimentalement)
182         self.Aob_evite = 10            # constante (à trouver experimentalement)
183         self.Aob_contact = 2*10**3
184         self.Asuivre_mur = 30           # constante (à trouver experimentalement)
185
186         self.c1 = 0.5                   # constante (à trouver experimentalement)
187         self.c2 = 0.4                   # constante (à trouver experimentalement)
188         self.c3 = 4                     # constante (à trouver experimentalement)
189         self.c4 = 0.5                   # constante (à trouver experimentalement)
190
191         self.A = 2*10**3                # constante (N) : int
192         self.B = 0.08                  # constante (m) : int
193         self.k = 1.2*10**5              # parametre (kg/s^2) : int
194         self.kap = 2.4*10**5            # parametre (kg/(m*s^2))
195
196         self.murs = murs
197         self.obstacles = obstacles
198         self.sortie = sortie
199
200         self.v0i = np.ones((N))*3
201         #self.v0i = np.random.rand((N))*3 + 3*np.ones((N)) # vitesse desirée initiale (en norme)
202         self.m = np.ones((N))*80        # masse
203         self.ray = ray
204         self.tau = 0.5                  # durée d'accélération
205         self.dt = dt                   # pas d'intégration
206
207         self.p_suivre = p_suivre
208
209
210     #Calcul de la force totale exercée sur les agents
211     def f_final_cepabs(self, r, v, Pa):
212         a = np.zeros((len(r), 2))
213         v0e0 = self.vit_des(r, Pa)
214         F_ag = self.f_agent(r, v)
215         F_mur = self.f_mur(r, v)
216         F_ob = self.f_obstacles(r, v)
217         F_aj = self.f_vitesse_amortie(v)
218
219         for i in range(len(r)):
220             a[i, :] = ((v0e0[i, :] - v[i, :])/self.tau) + (F_ag[i, :]/self.m[i]) + ((F_mur[i, :])/self.m[i]) + (F_ob[i, :]/
221             self.m[i]) - (F_aj[i, :] / self.m[i])
222         return a
```

Annexe 5

```
224 #Calcul de la direction desiree des agents
225 def dir_des(self, r):
226     e0 = np.zeros((len(r), 2))
227     for i in range(len(r)):
228         e0[i, :] = (self.sortie - r[i, :]) / np.linalg.norm(self.sortie - r[i, :])
229     return e0
230
231 #Tendance à suivre les autres
232 def dir_des_moyenne(self, r, p):
233     e0 = self.dir_des(r)
234     e0_moyenne = np.array([0.0, 0.0])
235
236     for i in range(len(r)):
237         e0_moyenne += e0[i, :]
238     e0_moyenne /= np.linalg.norm(e0_moyenne)
239     e0_s = ((1-p)*e0 - p*e0_moyenne) / np.linalg.norm((1-p)*e0 - p*e0_moyenne)
240     return e0_s
241
242 #Calcul de la norme de la vitesse desiree des agents
243 def vit_des_norme(self, r, Pa):
244     v0 = np.zeros((len(r)))
245     for i in range(len(r)):
246         v0[i] = self.v0i[i] * (1 + (Pa[i]/100))
247     return v0
248
249 #Calcul de la vitesse desiree des agents
250 def vit_des(self, r, Pa):
251     v0e0 = np.zeros((len(r), 2))
252     v0 = self.vit_des_norme(r, Pa)
253     e0 = self.dir_des_moyenne(r, self.p_suivre)
254     for i in range(len(r)):
255         v0e0[i, :] = v0[i]*e0[i, :]
256     return v0e0
```

Annexe 5

```
260 """f_agents"""
261
262 def g(self, x):
263     if x < 0:
264         return 0
265     return x
266
267 #Caracteristiques agent i vers j
268 def agent(self, i, j, r):
269     d = np.linalg.norm(r[i, :] - r[j, :])
270     n = (r[i, :] - r[j, :])/d
271     t = np.array([-n[1], n[0]])
272     return d, n, t
273
274 #Force exercee par j sur i
275 def f_ij(self, i, j, r, v):
276     d, n, t = self.agent(i, j, r)
277     dv_t = np.dot(v[j, :] - v[i, :], t)
278     r_tot = self.ray[i] + self.ray[j]
279     a = (self.A * np.exp((r_tot - d)/self.B)) + (self.k * self.g(r_tot - d))
280     b = self.kap * self.g(r_tot - d) * dv_t
281     return a*n + b*t
282
283 #Si l'agent i voit l'agent j
284 def gamma_agent(self, i, j, r):
285     e0 = self.dir_des_moyenne(r, self.p_suivre)
286     d, n, t = self.agent(i, j, r)
287     x = np.dot(-n, e0[i, :])
288     if x < 0 :
289         return 0
290     if d < 60 :
291         y = x * d
292         z = np.sqrt(d**2 + y**2)
293         if self.ray[i] + self.ray[j] > z :
294             return 1
295     return 0
296
297
298 def g_primel(self, i, j, r):
299     if np.linalg.norm(r[j, :] - r[i, :]) > self.ray[i] + self.ray[j]:
300         return 1 / (np.linalg.norm(r[j, :] - r[i, :]) - (self.ray[i] + self.ray[j]))
301     return 0
302
303 #Ralentir a la vue d'agent
304 def f_agent_retard(self, i, j, r, v):
305     d, n, t = self.agent(i, j, r)
306     dv_n = np.dot(v[j, :] - v[i, :], t)
307     a = self.Aag_retard * self.gamma_agent(i, j, r) * self.g_primel(i, j, r) * dv_n
308     return a * n
```

Annexe 5

```
310 #Eviter les agents
311 def f_agent_evite(self, i, j, r, v):
312     d, n, t = self.agent(i, j, r)
313     dv_n = np.dot(v[j, :] - v[i, :], n)
314     a = self.Aag_evite * self.gamma_agent(i, j, r) * self.g_prime1(i, j, r) * dv_n
315     return a * t
316
317 #Force totale des agents
318 def f_agent(self, r, v):
319     F = np.zeros((len(r), 2))
320     for i in range(len(r)):
321         for j in range(len(r)):
322             if i != j:
323                 F[i, :] += self.f_ij(i, j, r, v)
324                 F[i, :] += self.f_agent_retard(i, j, r, v)
325                 F[i, :] += self.f_agent_evite(i, j, r, v)
326     return F
327
328
329 """f_mur"""
330
331 #Caracteristiques de l'agent i au segment j (murs ou obstacle)
332 def dist_iM(self, i, r, seg):
333     temp = np.dot(r[i, :] - seg[0, :], seg[1, :] - seg[0, :]) / (np.linalg.norm(seg[1, :] - seg[0, :]))**2
334     temp = min( max( 0, temp ), 1 )
335     proche = seg[0, :] + (seg[1, :] - seg[0, :])*temp
336     d = np.linalg.norm(proche - r[i, :])
337     n = proche - r[i, :] / d
338     t = np.array([-n[1], n[0]])
339     return d, n, t, proche
340
341 #Force exercee par le mur j sur l'agent i
342 def f_iM(self, i, j, r, v):
343     d, n, t = self.dist_iM(i, r, self.murs[j, :, :])[:-1]
344     dv_t = -np.dot(v[i, :], t)
345     a = (self.A * np.exp((self.ray[i] - d)/self.B)) + (self.k * self.g(self.ray[i] - d))
346     b = self.kap * self.g(self.ray[i] - d) * dv_t
347     return a*n + b*t
348
349
350 def seg_intersect(self, a1,a2,b1,b2):
351     da = a2-a1
352     db = b2-b1
353     dp = a1-b1
354     dap = np.array([-da[1],da[0]])
355     denom = np.dot(dap, db)
356     num = np.dot(dap, dp)
357     return (num / denom.astype(float))*db + b1
```


Annexe 5

```
359 #Si l'agent i voit l'obstacle j
360 def gamma_obstacle(self, i, j, r, seg):
361     e0 = self.dir_des_moyenne(r, self.p_suivre)
362     d = self.dist_iM(i, r, seg[j, :, :])[0]
363     if d < 60 :
364         proche = self.seg_intersect(r[i], r[i] + e0[i], seg[j, 0], seg[j, 1])
365         if np.dot(seg[j, 0] - proche, seg[j, 1] - proche) < 0:
366             if np.dot(e0[i], proche - r[i]) > 0:
367                 return 1
368     return 0
369
370
371 def g_prime2(self, i, j, r, seg):
372     d, n, t, proche = self.dist_iM(i, r, seg[j, :, :])
373     if d > self.ray[i]:
374         return 1 / (d - self.ray[i])
375     return 0
376
377 #Ralentir a la vue de l'obstacle
378 def f_ob_retard(self, i, j, r, v, seg):
379     d, n, t = self.dist_iM(i, r, seg[j, :, :])[:-1]
380     dv_n = -np.dot(v[i, :], n)
381     a = self.Aob_retard * self.gamma_obstacle(i, j, r, seg) * dv_n * (np.exp(self.ray[i] - d))
382     return a * n
383
384 #Stopper l'agent i au contact de l'obstacle j
385 def f_ob_contact(self, i, j, r, v):
386     d, n, t = self.dist_iM(i, r, self.obstacles[j])[:-1]
387     dv_n = -np.dot(v[i, :], n)
388     a = self.Aob_contact * dv_n * np.exp((self.ray[i] + 0.05 - d) / self.B)
389     return a * n
390
391 #Eviter l'obstacle
392 def f_ob_evite(self, i, j, r, v, seg):
393     d, n, t = self.dist_iM(i, r, seg[j, :, :])[:-1]
394     dv_n = -np.dot(v[i, :], n)
395     e0 = self.dir_des_moyenne(r, self.p_suivre)
396     e0_t = np.array([-e0[i, 1], e0[i, 0]])
397     a = self.Aob_evite * self.gamma_obstacle(i, j, r, seg) * self.g_prime2(i, j, r, seg) * dv_n
398     return a * e0_t
399
400 #Suivre le mur
401 def f_suivre_mur(self, i, j, r):
402     d, n, t = self.dist_iM(i, r, self.obstacles[j, :, :])[:-1]
403     a = self.Asuivre_mur * np.exp((1.07 * self.ray[i] - d) / self.B)
404     e0 = self.dir_des_moyenne(r, self.p_suivre)
405     return a * t
```

Annexe 5

```
407 #Force totale des murs
408 def f_mur(self, r, v):
409     F = np.zeros((len(r), 2))
410     for i in range(len(r)):
411         for j in range(len(self.murs)):
412             F[i, :] += self.f_iM(i, j, r, v)
413             F[i, :] += self.f_ob_retard(i, j, r, v, self.murs)
414
415     return F
416
417 #Force totale des obstacles
418 def f_obstacles(self, r, v):
419     F = np.zeros((len(r), 2))
420     for i in range(len(r)):
421         for j in range(len(self.obstacles)):
422             F[i, :] += self.f_ob_retard(i, j, r, v, self.obstacles)
423             F[i, :] += self.f_ob_contact(i, j, r, v)
424             F[i, :] += self.f_ob_evite(i, j, r, v, self.obstacles)
425             F[i, :] += self.f_suivre_mur(i, j, r)
426     return F
427
428
429 """Ajustements"""
430
431 #Amortissement de la vitesse
432 def f_vitesse_amortie(self, v):
433     v_am = np.zeros((len(v), 2))
434     for i in range(len(v)):
435         v_am[i, :] = ((np.linalg.norm(v[i, :]))**0.5)*v[i, :]
436     return v_am
437
438
439
440 """Panique"""
441
442 #Solitude
443 def solitude(self, r):
444     S = np.zeros((len(r)))
445     for i in range(len(r)):
446         for j in range(len(r)):
447             if i != j:
448                 d = self.agent(i, j, r)[0]
449                 if d < 1:
450                     S[i] +=1
451                     break
452     return S
```

Annexe 5

```
454 #Eq diff panique
455 def panique(self, r, v, Pa):
456     Pa_moyenne = 0
457     for i in Pa:
458         Pa_moyenne += i
459     Pa_moyenne /= len(Pa)
460     v0e0 = self.vit_des(r, Pa)
461     S = self.solitude(r)
462
463     Pa_prime = np.zeros((len(Pa)))
464     for i in range(len(Pa)):
465         Pa_prime[i] = self.c1*(Pa_moyenne - Pa[i]) + self.c2*np.linalg.norm(v0e0[i, :] - v[i, :]) + self.c3*S[i] + self.c4
466     return Pa_prime
467
468
469
470
471 """Calcul de la position"""
472
473 #Calcul de la vitesse et de la position des agents à l'instant k
474 def euler(self, r, v, Pa):
475     a1 = np.zeros((len(r), 2))
476     v1 = np.zeros((len(r), 2))
477     r1 = np.zeros((len(r), 2))
478
479     evacue = []
480
481     a1 = self.f_final_cepabs(r, v, Pa)
482     v1 = v + self.dt * a1
483     r1 = r + self.dt * v1
484
485     Pa_prime = self.panique(r, v, Pa)
486     Pa1 = Pa + self.dt * Pa_prime
487
488     for i in range(len(r1)):
489         if np.linalg.norm(r1[i, :] - self.sortie) < 2*self.ray[i]:
490             r1[i, :] = np.random.rand(1, 2) * 10**6
491             evacue.append(i)
492
493     return r1, v1, Pa1, evacue
```

Annexe 6

Recherche des constantes :

```
1 import numpy as np
2 import matplotlib.pyplot as plt
3
4 from Modeles import *
5
6 rayon = 3
7 dt = 0.1
8 tau = 0.5
9 p_suivre = 0
10
11
12 ##Paramètres##
13
14 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.]])
15
16 obstacles = np.array([[[330, 300],[330, 260]]])
17
18 r = np.array([400., 290.])
19
20 sortie = np.array([200., 280.])
21
22 Pa = np.zeros((len(r)))
```

Annexe 6

```
25 ##Tests##
26
27
28 X = np.arange(300, 4010, 10)
29 Y = []
30
31 for i in X:
32     evacue = []
33     position = CEPABS(murs, obstacles, sortie, len(r), rayon, tau, dt, p_suivre, i) # On pose self.Aob_retard = i dans CEPABS
34
35     while len(evacue) < len(rtot[0]):
36
37         r1, v1, Pa1, sortis = position.euler(r, v, Pa)
38         for i in sortis:
39             evacue.append(i)
40
41         rtot = np.vstack([rtot, [r1]])
42         vtot = np.vstack([vtot, [v1]])
43
44         r = r1
45         v = v1
46         Pa = Pa1
47
48         vmoy = np.mean(vtot)
49         Y.append(vmoy)
50
51 plt.clf()
52 plt.plot(X, Y)
53 plt.grid()
54 plt.axis([200, 1600, 0, 15])
55
56 plt.xlabel("Aob_retard")
57 plt.ylabel("Vmoy")
58 plt.show()
```

Annexe 7

Test Panique :

```
1 import numpy as np
2 import matplotlib.pyplot as plt
3
4 from Modeles import *
5
6 rayon = 3
7 dt = 0.1
8 tau = 0.5
9 p_suivre = 0
10
11 ##Paramètres##
12
13 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.]])
14
15 r = np.array([[305., 305.],
16               [285., 268.],
17               [217., 250.],
18               [433., 270.],
19               [266., 349.],
20               [265., 248.],
21               [375., 229.],
22               [420., 217.],
23               [317., 235.],
24               [428., 257.],
25               [314., 260.],
26               [213., 216.],
27               [341., 237.],
28               [390., 212.],
29               [295., 226.],
30               [317., 216.],
31               [251., 284.],
32               [404., 268.],
33               [316., 348.],
34               [430., 302.],
35               [336., 342.],
36               [345., 269.],
37               [378., 269.],
38               [226., 322.],
39               [231., 333.],
40               [395., 302.],
41               [392., 277.],
42               [373., 300.],
43               [360., 206.],
44               [391., 246.]])
45
46
47 sortie = np.array([200., 280.])
48
49 obstacles = np.array([[[[0, 0],[0, 0]]])
```

Annexe 7

```
52 ##Tests##
53
54 X = np.arange(0, 101, 1)
55 Y = []
56
57
58 for i in X:
59     Pa = np.ones((len(r)))*i
60     evacue = []
61     position = CEPABS(murs, obstacles, sortie, len(r), rayon, tau, dt, p_suivre)
62
63     while len(evacue)<len(rtot[0]):
64
65         r1, v1, Pa1, sortis = position.euler(r, v, Pa)
66         for i in sortis:
67             evacue.append(i)
68
69         rtot = np.vstack([rtot,[r1]])
70         vtot = np.vstack([vtot,[v1]])
71
72         r = r1
73         v = v1
74
75     Y.append(len(rtot)*dt)
76
77 plt.clf()
78 plt.plot(X, Y)
79 plt.grid()
80 plt.axis([0, 1, 0, 200])
81
82 plt.xlabel("Panique")
83 plt.ylabel("Temps de sortie")
84 plt.show()
```

Annexe 8

Test Obstacles :

```
1 import numpy as np
2 import random as rd
3 import matplotlib.pyplot as plt
4
5 from Modeles import *
6
7 tau = 0.5          # temps de réaction
8 dt = 0.1           # pas d'integration
9 rayon = 5          # rayon des agents -> 1m = rayon*2 pixels
10 p_suivre = 0       # tendance à suivre les autres
11
12
13
14 ##Paramètres##
15
16 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.]]]]]
17
18 sortie = np.array([200., 280.])
19
20 m = np.ones((len(r)))*80
21
22
23 Pa = np.zeros((len(r)))
24
25
26
27 ##Tests##
28
29
30 X = [k for k in range(240)]
31 Y = [k for k in range(160)]
32 Z = np.zeros((240, 160))
33
34
35 def test_peut_placer_personnes(x, y, r, obstacles):
36     peut_placer_personnes = True
37     for i in r:
38         if (((i[0]-x)**2)+(i[1]-y)**2)**0.5 <= 2*rayon:
39             peut_placer_personnes = False
40             continue
41
42     point = np.array([x, y])
43
44     for i in murs:
45         temp = np.dot(point - i[0, :], i[1, :] - i[0, :]) / (np.linalg.norm(i[1, :] - i[0, :])**2)
46         temp = min( max( 0, temp ), 1 )
47         proche = i[0, :] + (i[1, :] - i[0, :])*temp
48         d = np.linalg.norm(proche - point)
49         if d<=rayon:
50             peut_placer_personnes = False
```


Annexe 8

```
53     k = 0
54     for i in murs:
55         if i[0, 1] == i[1, 1]:
56             if i[0, 1] < y:
57                 if (i[0, 0] - x) * (i[1, 0] - x) < 0:
58                     k += 1
59     if k % 2 == 0:
60         peut_placer_personnes = False
61
62     return peut_placer_personnes
63
64
65
66 for i in range(21, 240):
67     for j in range(1, 140):
68         obstacles = np.array([[[200, 200], [200, 220]]]) + np.array([[i, j], [i, j]]) #longueur : 20 = 2m
69         r = np.zeros((30, 2))
70         k = 0
71         while k < 30:
72             x = np.random.randint(rayon, 600 - rayon)
73             y = np.random.randint(rayon, 600 - rayon)
74             if test_peut_placer_personnes(x, y, r, obstacles):
75                 r[k] = [x, y]
76                 k += 1
77         evacue = []
78         position = CEPABS(murs, obstacles, sortie, len(r), rayon, tau, dt, p_suivre)
79         while len(evacue) < len(rtot[0]):
80
81             r1, v1, Pa1, sortis = position.euler(r, v, Pa)
82             for i in sortis:
83                 evacue.append(i)
84             rtot = np.vstack([rtot, [r1]])
85             vtot = np.vstack([vtot, [v1]])
86             r = r1
87             v = v1
88             Pa = Pa1
89             ttot = len(rtot) * dt / (2 * rayon)
90             Z[i, j] += ttot
91
92 plt.clf()
93 x, y = np.meshgrid(X, Y)
94 plt.contourf(x, y, np.transpose(Z), 1000)
95 plt.colorbar()
96 plt.axis('equal')
97 plt.show()
```

Annexe 9

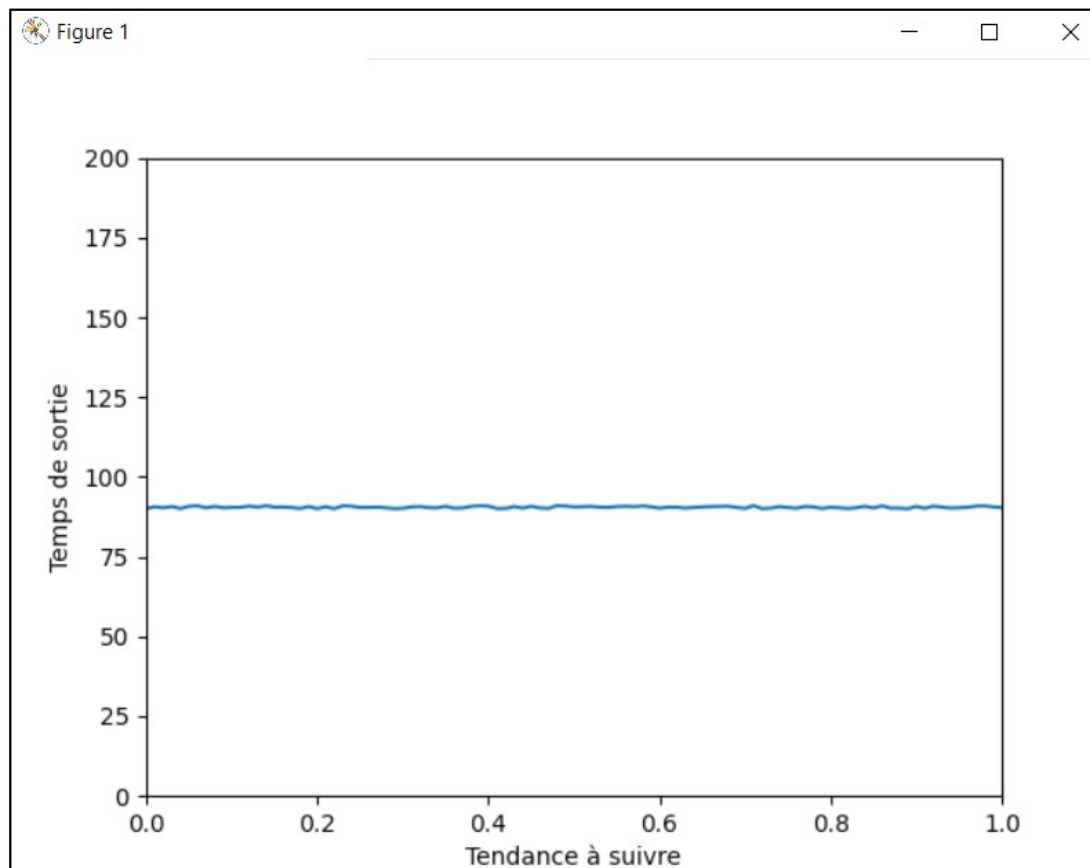
Ajout du modèle comportemental :

Tendance à suivre les autres :

$$\vec{e}_i^{0'} = \frac{(1-q)\vec{e}_i^0 + q\langle \vec{e}_j^0 \rangle}{\|(1-q)\vec{e}_i^0 + q\langle \vec{e}_j^0 \rangle\|}$$

$$q \in [0, 1]$$

Annexe 9



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

Annexe 10

Test Tendance à suivre :

```
1 import numpy as np
2 import matplotlib.pyplot as plt
3 from Modeles import *
4
5 rayon = 3
6 dt = 0.1
7 tau = 0.5
8
9 X = np.arange(0, 1.01, 0.01)
10 Y = []
11
12 murs = np.array([[156., 87.],[324., 87.],[324., 87.],[324., 172.],[324., 172.],[156., 172.],[156., 139.],
13 [[156., 127.],[156., 87.]])
14 r = np.array([[196., 114.],[209., 149.],[205., 133.], [209., 115.],[278., 106.],[269., 167.],[266., 152.],[193., 145.],[257.,
15 132.],[229., 161.]])
16 obstacles = np.array([[0, 0],[0, 0]])
17 sortie = np.array([156., 133.])
18 ray = np.ones((len(r)))*rayon
19 m = np.ones((len(r)))*80
20 Pa = np.zeros((len(r)))
21
22 for i in X:
23     L = []
24     for j in range(100):
25         evacue = []
26         position = CEPABS(murs, obstacles, sortie, len(r), ray, m, tau, dt, i)
27
28         while len(evacue)<len(rt0t[0]):
29
30             r1, v1, Pa1, sortis = position.euler(r, v, Pa)
31             for i in sortis:
32                 evacue.append(i)
33
34             rtot = np.vstack([rtot,[r1]])
35             vtot = np.vstack([vtot,[v1]])
36
37             r = r1
38             v = v1
39             Pa = Pa1
40
41             L.append(len(rt0t)*dt)
42
43     Y.append(np.mean(L))
44
45 plt.clf()
46 plt.plot(X, Y)
47 plt.grid()
48 plt.axis([0, 1, 0, 200])
49
50 plt.xlabel("Aob_retard")
51 plt.ylabel("Vmoy")
52 plt.show()
```