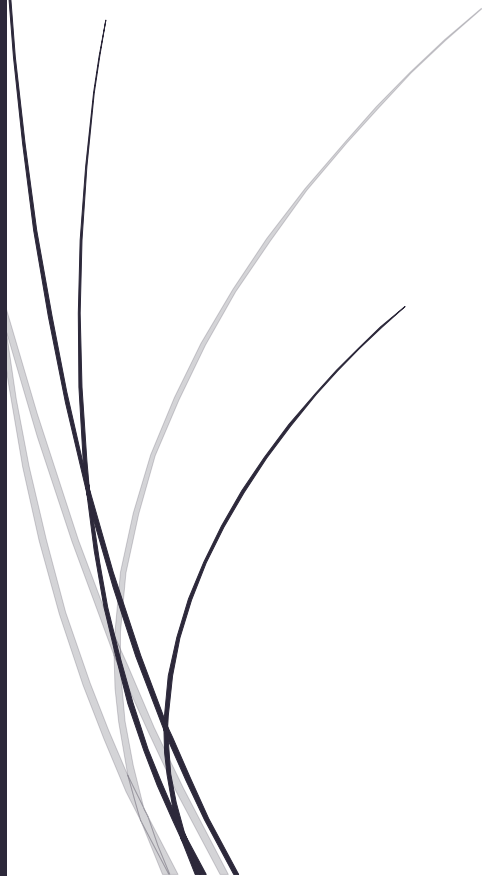
TIPE - MP

SIMULATION DE L'EVACUATION D'UNE FOULE

Etude de certains facteurs influençant l'efficacité d'une évacuation

Benbaki Riade



SIMULATION DE L'EVACUATION D'UNE FOULE

Etude de certains facteurs influançant l'efficacité d'une evacuation

OBJECTIF

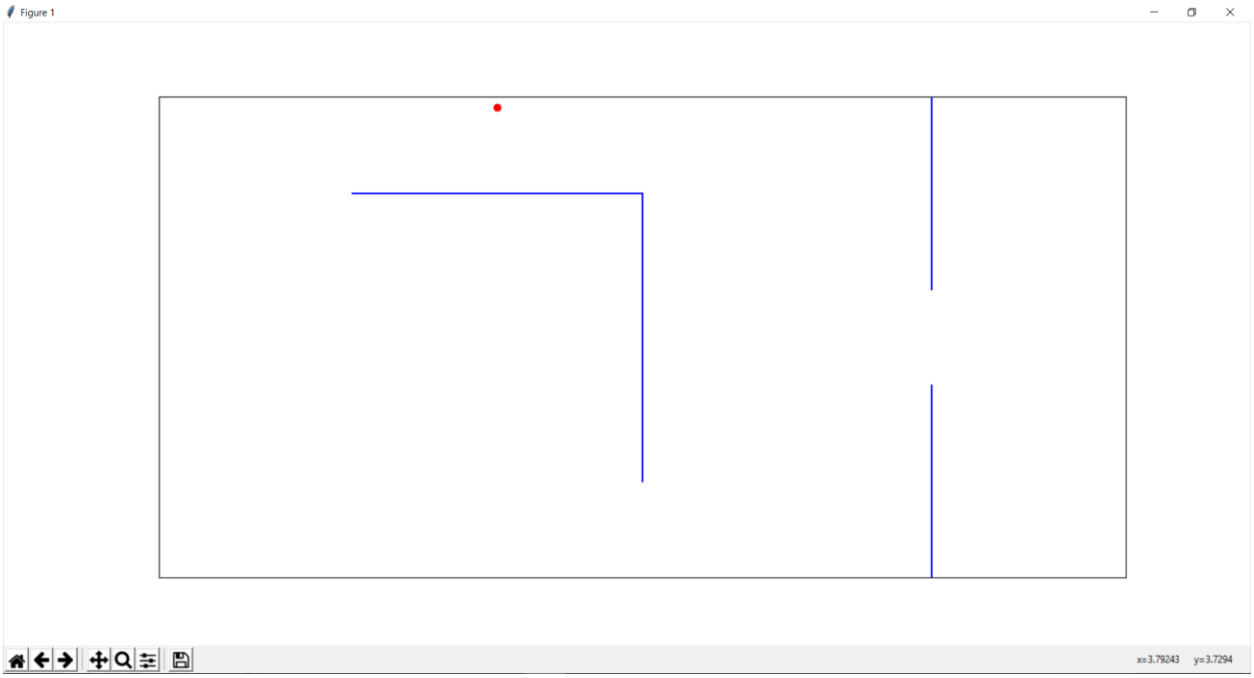
Créer un algorithme qui simule l'évacuation d'une foule d'une pièce rectangulaire. La simulation doit s'approcher le plus possible du comportement réel des individus pour que les résultats et les données finales soit proches de la réalité.

ETAPES DE LA REALISATION DE LA SIMULATION

- La gestion du comportement individuel
- La gestion des interactions entre individus

ENVIRONNEMENT DE LA SIMULATION :

La simulation est réalisée avec Python, en utilisant les modules **MATPLOTLIB** pour l'affichage de l'animation et **Numpy** pour la gestion des données.



1 – Comportement individuel

Les paramètres du mouvement de chaque individu :

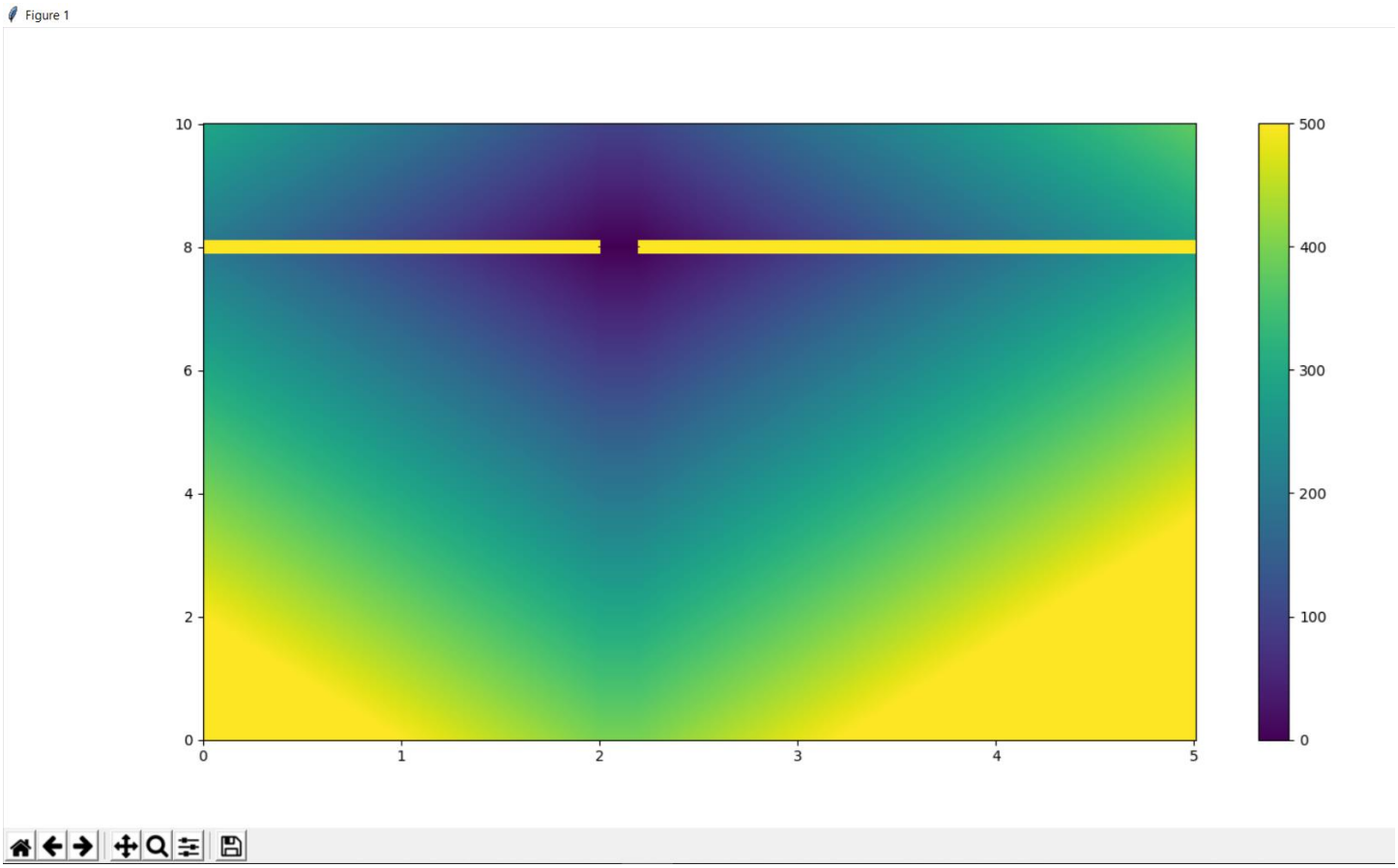
- **La norme de sa vitesse**
- **La direction et le sens de cette vitesse**

```
class individu:
    def __init__(self,x,y,vitesse,angle):
        self.x = x
        self.y = y
        self.vitesse = vitesse
        self.angle = angle
        line, = ax.plot(self.x,self.y,marker='o', color='r',lw=2)
        self.line = line
    def affichage(self):
        self.line.set_data(self.x,self.y)
    def deplacement(self):
        self.x += interval*self.vitesse*np.cos(self.angle)
        self.y += interval*self.vitesse*np.sin(self.angle)
```

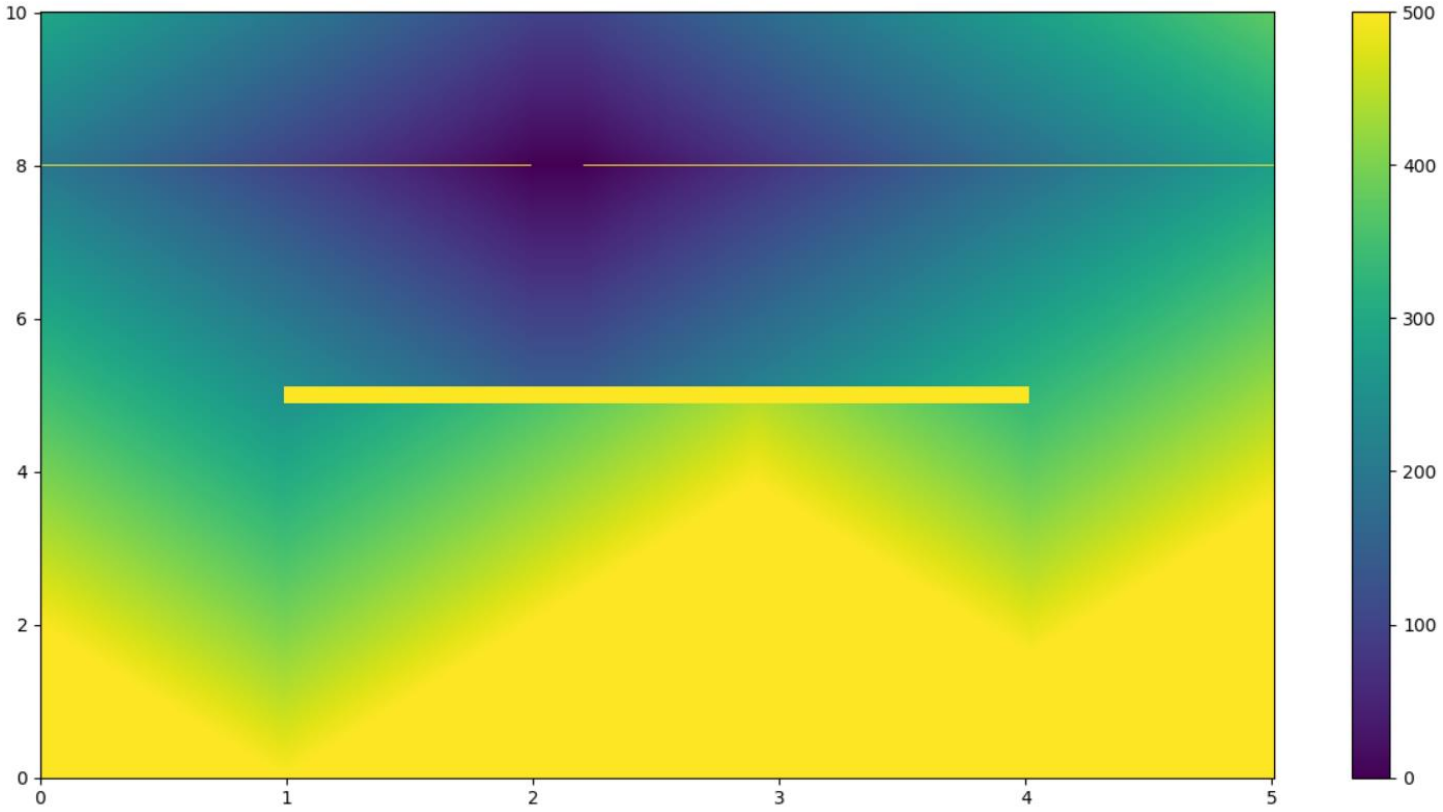
$$dx = dt \times Vx = dt * V * \cos\theta$$

Utilisation d'un champ de potentiel pour déterminer le chemin le plus court

Le champ de potentiels sans obstacles :



Avec obstacle :



Discrétisation du plan

```
nombredesubdivisionsselonx = 200
nombredesubdivisionsselony = 200
pasx = larg/nombredesubdivisionsselonx
pasy = haut/nombredesubdivisionsselony
maillage = np.zeros((nombredesubdivisionsselonx,nombredesubdivisionsselony))
maillage.fill(1000)
def convertirencoef(x,y): #Convertir coordonnee en i et j
    i = int(x*nombredesubdivisionsselonx/larg)
    j = int(y*nombredesubdivisionsselony/haut)
    return i,j
```

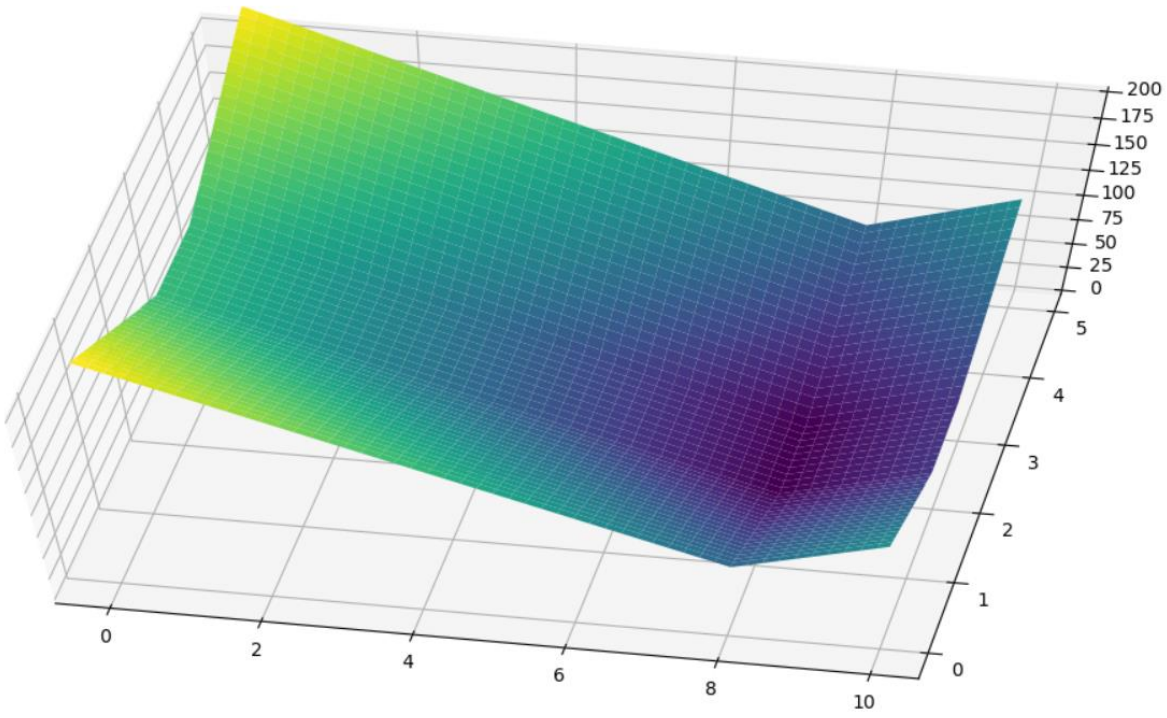
Utilisation de la programmation dynamique

La structure du problème est parfaitement adaptée pour être traité en utilisant la programmation dynamique :

- **Structure optimale**
- **Chevauchement des sous problèmes**

```
##### calcul de la distance - Programmation dynamique
file = list()
buts = [convertirencoef(mur,2),convertirencoef(mur,2.2)]
point1,point2 = buts[0],buts[1] #Extremites du mur
for i in range(point1[0],point2[0]+1):
    for j in range(point1[1],point2[1]+1):
        maillage[i,j] = 0 #Le but recoit la valeur 0
        file.append((i,j))

while len(file) > 0:
    i,j = file.pop() #Point actuel
    voisins = [(i-1,j),(i+1,j),(i,j-1),(i,j+1)]
    for iv,jv in voisins:
        if iv >= nbrdivx or jv >= nbrdivy or iv < 0 or jv < 0: continue
        if siobstacle(iv,jv) : continue
        nouvellevaleur = maillage[i,j] + 1
        if nouvellevaleur < maillage[iv,jv]:
            maillage[iv,jv] = nouvellevaleur
            file.insert(0,(iv,jv))
np.savetxt("file",maillage,delimiter=" ; ",newline="\n")
```

Courbes 3D de la grille remplies :

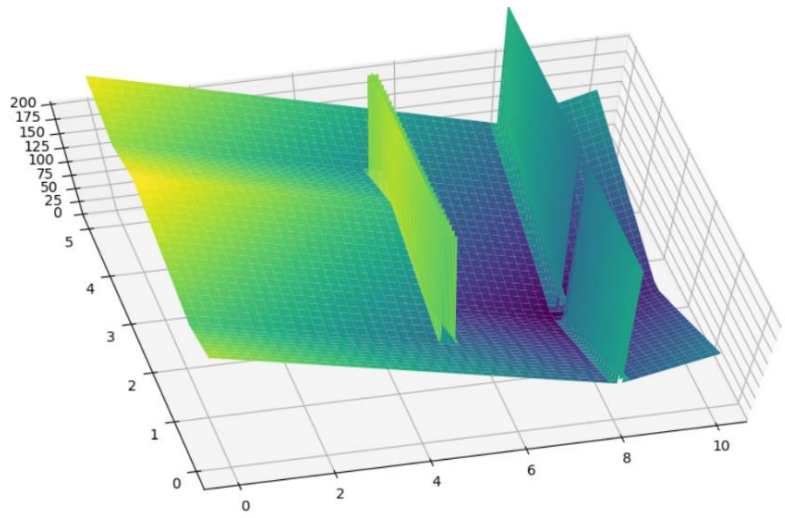


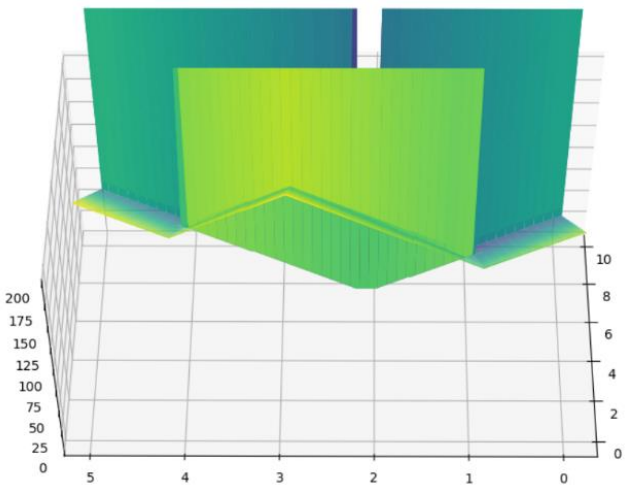
Le chemin du minimum mène à la sortie, qu'on peut suivre en utilisant l'algorithme de descente de gradient.

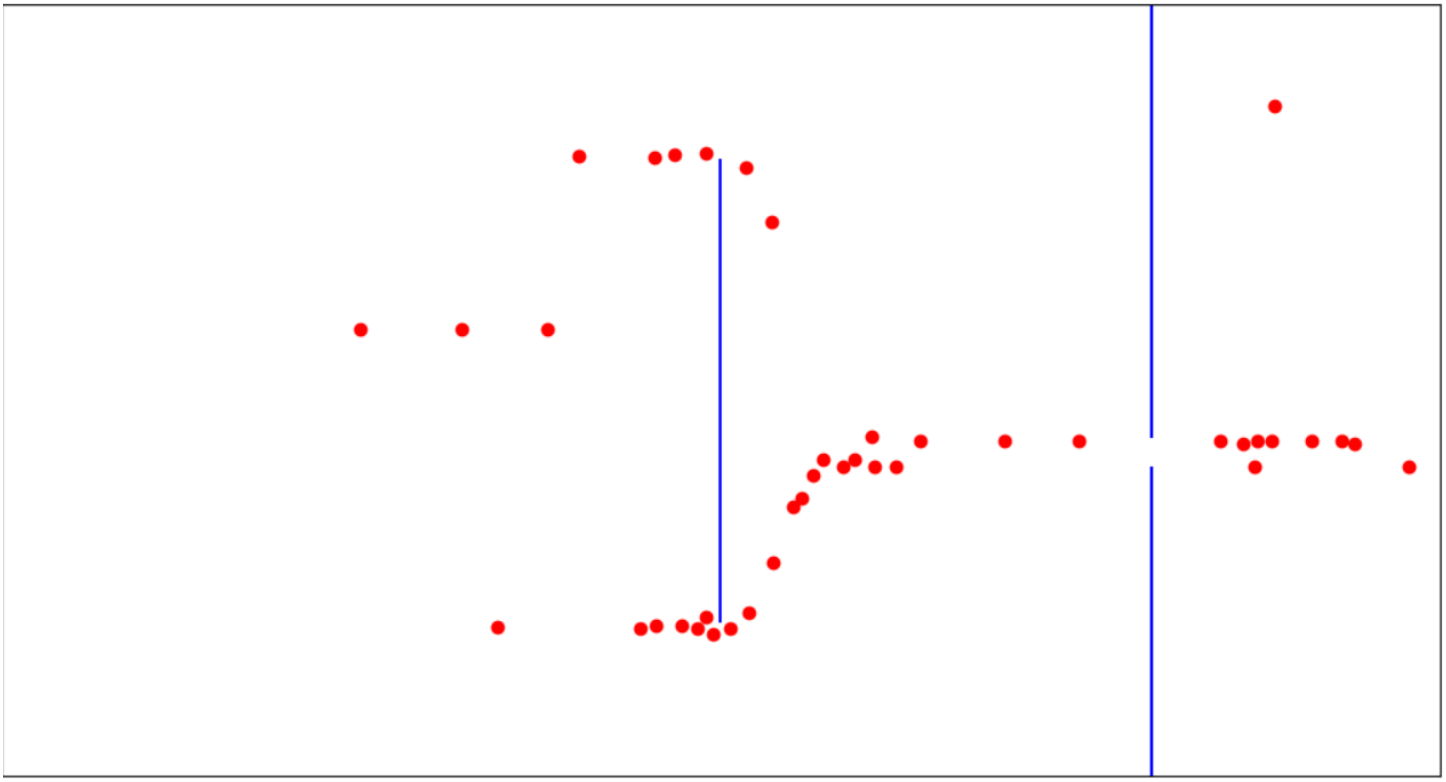
```
### Calcul de langle que fait le vecteur gradient
gradient = np.zeros_like(maillage)
for i in range(nombredesubdivisionsselonx):
    for j in range(nombredesubdivisionsselony):
        left = i-1 if i > 0 else i
        right = i+1 if i < nombredesubdivisionsselonx -1 else i
        up = j+1 if j < nombredesubdivisionsselony -1 else j
        down = j-1 if j > 0 else j

        y = maillage[i,up]-maillage[i,down]
        x = maillage[right,j]-maillage[left,j]
        x /= pasx
        y /= pasy
        if x != 0:
            gradient[i,j] = np.arctan(y/x)
        elif y ==0: gradient[i,j] = 0
        else: gradient[i,j] = np.pi*np.sign(y)
######
```

Limites du modèle :

Le problème principal est l'existence potentielle de minimum locaux ou de maximum.







Pour pallier à ce problème, on vérifie à chaque étape, dans le gradient est nul, si la zone est un maximum. Dans ce cas, on ajoute une légère perturbation a la trajectoire de l'individu pour qu'il quitte cette zone de maximum.

```
def simaximum(i,j):
    voisins = [(i-2,j),(i+2,j),(i,j-2),(i,j+2)]
    return maillage[i,j] >= max([maillage[iv,jv] for iv,jv in voisins])
```

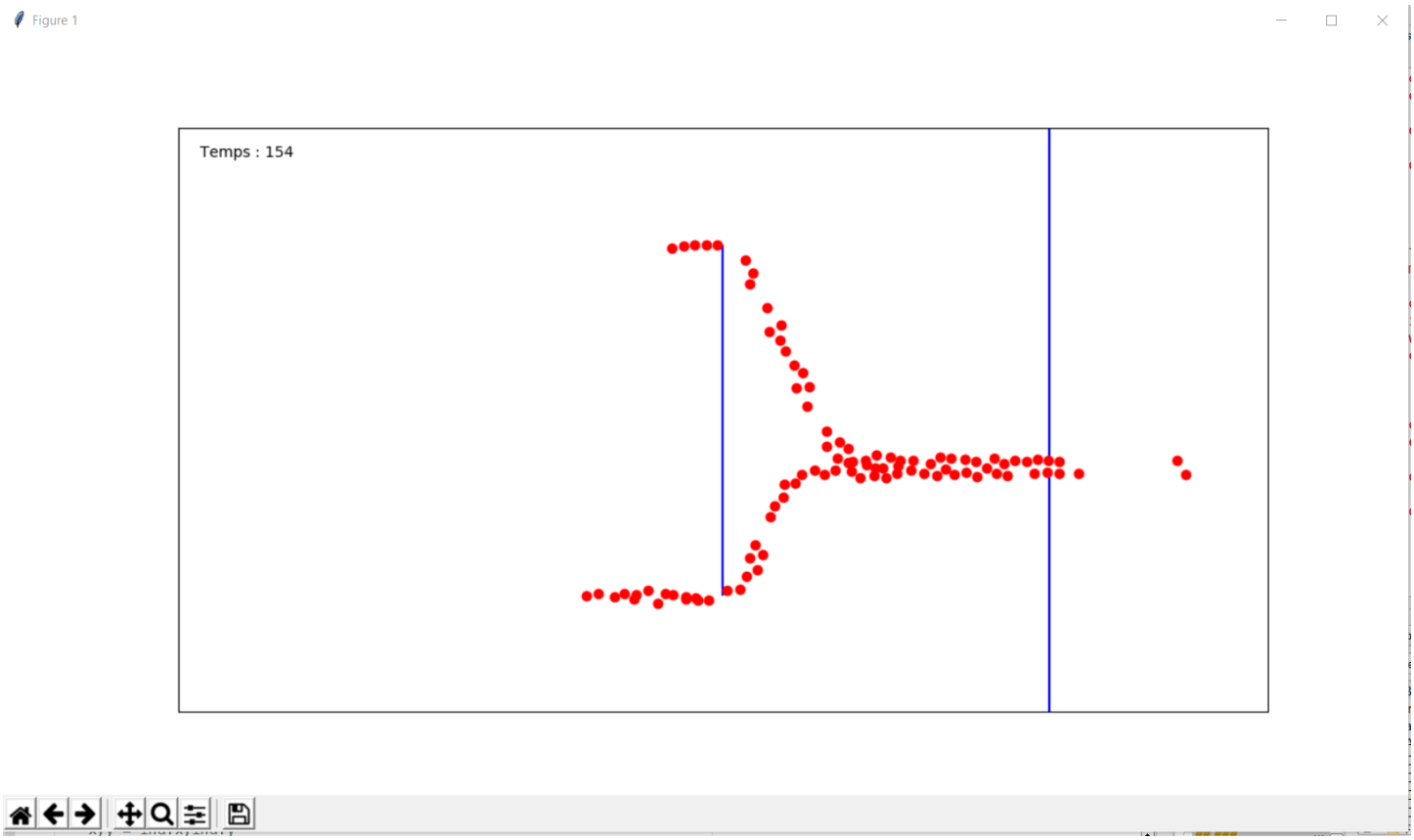
```
if i< convertirencoef(mur,0)[0] and j < nombredesubdivisionsselony - 3 and int(ind.angle) == 0 and simaximum(i,j)
    ind.x += pasx
    ind.y +=  pasy
```

2 – Gestion des interactions

```
def choc(ind1,ind2):
    d = np.sqrt((ind1.x-ind2.x)**2 + (ind1.y-ind2.y)**2 )
    if d < 0.1:
        ind1.nbrchoc += 1
        ind2.nbrchoc += 1
        return True
    else: return False
def rebond(self,ind2):
    v1 = np.array([self.vitesse*np.cos(self.angle),self.vitesse*np.sin(self.angle)])
    v2 = np.array([ind2.vitesse*np.cos(ind2.angle),ind2.vitesse*np.sin(ind2.angle)])
    u = np.array([ind2.x - self.x,ind2.y - self.y])
    v1nouv = v1 - np.vdot(v1-v2,u)*u
    v2nouv = v2 + np.vdot(v1-v2,u)*u
    self.angle = np.arctan2(v1nouv[1],v1nouv[0])
    ind2.angle = np.arctan2(v2nouv[1],v2nouv[0])
    tang = np.arctan2(ind2.y-self.y,ind2.x - self.x)
    angle = 0.5 * np.pi + tang
    self.x += 0.1*np.sin(angle)
    self.y -= 0.1*np.cos(angle)
    ind2.x -= 0.1*np.sin(angle)
    ind2.y += 0.1*np.cos(angle)
```

3 – Données

Certains comportements observés expérimentalement dans les foules, comme la formation spontanée de files, sont présents.



Comparaison avec et sans obstacles

On utilise deux paramètres pour comparer :

- **Le temps d'évacuation**
- **Le nombre de chocs durant l'évacuation**

Données à partir de 50 simulations (100 individus) :

	Sans obstacles / Avec obstacles
Temps moyen d'évacuation	**0.78**
Nombre de chocs moyen	**1.17**