Simulation microscopique de foule grâce aux forces sociales en respectant les contraintes de distances sanitaires

GENEST
Hugo
Numéro d'inscription : 46185

ANNEXE

```
def topologie_mur(M,B):
    if 1 < M[0] < 1.2  and  0.5 < M[1] < 1.5:
        return np.inf
    else:
        return distance(M,B)

def search_pente(M,B,topologie,h):#M:coordonnées de départ, h:taille du pas,
topologie:focntion donnant le potentiel
    x ,y = M[0], M[1]
    cercle = np.linspace(0,2*np.pi,100)
    pente = topologie(M,B)
    N = M
    for theta in cercle:
        if pente >= topologie([x+h*np.cos(theta),y+h*np.sin(theta)],B):
            pente = topologie([x+h * np.cos(theta),y+ h * np.sin(theta)],B)
            N = [x+h * np.cos(theta),y+h * np.sin(theta)]
    return N# N:coordonnées du nouveau point

def recherche_chemin(A,B,topologie,N):# A point de départ et B d'arrivée
    Ms = [A]#    liste des coordonnées parcourues
    compteur =0
    while True and compteur < N:
        compteur += 1
        pente = search_pente(Ms[-1],B,topologie,0.02)
        Ms.append(pente)
        if len(Ms) > 2:
            if N == Ms[-3]:
                break
    return Ms
```

```python
importance_potent = 0.75
importance_deplacement = 0.25/(2**(1/2))

dim = int
repulsifs = {'inv':set(coordonneés), 'plateau':set(coordonnées)}
murs = set(coordonnées)

def potent(pt,repulsifs,murs):
    if pt in murs:
        return np.inf
    V = 0
    for r in repulsifs['inv']:
        V += 1/(1+distance(r,pt))
    if pt in repulsifs['plateau']:
        V += 1
    return V

def calcul_poids(dim,repulsifs,murs):
    poids = [[[[importance_potent*potent((i+l-1,j+c-1),repulsifs,murs) +
importance_deplacement*distance((i,j),(i+l-1,j+c-1)) for c in range(3)] for l in range(3)]
for j in range(dim)] for i in range(dim)]
    return poids
```

```python
def dijkstra(dim,depart,arrivees,poids):#"arrivees" est un ENSEMBLE de points
    poids_tot = [[np.inf for j in range(dim+1)] for i in range(dim+1)]# dim+1 pour inclure les contours de l'expérience et ne pas avoir un index out of range dans le while
    chemins = [[None for j in range(dim)] for i in range(dim)]

    poids_tot[depart[0]][depart[1]] = 0
    pt = depart
    poids_atteint = 0
    chemin = [pt]
    pts_passes = {pt}

    while pt not in arrivees:
        x,y = pt[0]-1, pt[1]-1
        for l in range(3):
            for c in range(3):
                if poids_tot[x+l][y+c] >  poids_atteint + poids[x+1][y+1][l][c]:
                    poids_tot[x+l][y+c] = poids_atteint + poids[x+1][y+1][l][c]
                    chemins[x+l][y+c] = chemin + [(x+l,y+c)]
        poids_min = np.inf
        pt_min = None
        for i in range(dim):
            for j in range(dim):
                if (i,j) not in pts_passes:
                    if poids_tot[i][j] < poids_min:
                        poids_min = poids_tot[i][j]
                        pt_min = (i,j)
        pt = pt_min
        pts_passes.add(pt)
        poids_atteint = poids_min
        chemin = chemins[pt[0]][pt[1]]

    return chemin
```

```python
def directions_souhaitees(dim,arrivees,repulsifs,murs):
    poids = calcul_poids(dim,repulsifs,murs)
    directions = [[(i,j) for j in range(dim)] for i in range(dim)]
    points_fleches = arrivees.copy()
    classement = classement_distance(dim,arrivees,murs)
    for depart in classement:
        if depart not in points_fleches:
            chemin = dijkstra(dim,depart,arrivees,poids)
            for i in range(len(chemin)-1):
                if chemin[i] in points_fleches:
                    break
                else:
                    points_fleches.add(chemin[i])

                    print(len(points_fleches))

                    directions[chemin[i][0]][chemin[i][1]] = (chemin[i+1][0],chemin[i+1][1])
    return directions

def classement_distance(dim,arrivees,murs):
    map = []
    for i in range(dim):
        for j in range(dim):
            if (i,j) not in murs:
                map.append((i,j))
    map = tri_fusion(map,arrivees)
    return map
```

```python
def vitesse_souhaitee(dim,position,directions):
    if position == None:
        return (0,0)
    coeff = int(dim/len(directions))
    x,y = position
    i = int(x//coeff)
    j = int(y//coeff)
    dx = directions[i][j][0] - i
    dy = directions[i][j][1] - j
    d = ((dx**2)+(dy**2))**(1/2)
    if d == 0:
        return (0,0)
    return (dx/d,dy/d)

def vitesses_souhaitees(dim,positions,directions):
    vitesses = []
    for i in range(len(positions)):
        vitesses.append([vitesse_souhaitee(dim,positions[i],directions)
[0],vitesse_souhaitee(dim,positions[i],directions)[1]])
    return vitesses
```

```python
r = 1
h = 1

def uzawa_initialisation(positions,listes_murs):
    global r

    n = len(positions)
    n_obst = len(listes_murs)

    D = []
    E = []
    for q1 in range(n):
        D.append([])
        E.append([])

        for q2 in range(n):
            d = distance(positions[q1],positions[q2])
            D[q1].append(d - 2*r)
            if d != 0:
                E[q1].append(((positions[q2][0] - positions[q1][0])/d , (positions[q2][1] - positions[q1][1])/d ))
            else:
                E[q1].append((0,0))

        for obst in range(n_obst):
            pt_proche = (np.inf,np.inf)
            d = distance(positions[q1],pt_proche)
            for mur in listes_murs[obst]:
                if distance(positions[q1],mur) < d:
                    pt_proche = mur
                    d = distance(positions[q1],pt_proche)
            D[q1].append(d-r)
            if d !=0:
                E[q1].append( ((pt_proche[0]-positions[q1][0])/d, (pt_proche[1]-positions[q1][1])/d ) )
            else:
                E[q1].append( (0, 0) )

    epsilon = 0.1*r
    iter_max = 5000
    rho = 1# constante sélectionnée après l'essai de plusieurs valeurs

    return D,E,n,n_obst,epsilon,iter_max,rho
```

```python
def phi(v,E,n,n_obst):
    global h

    retour = []
    for i in range(n):
        retour.append([])
        for nul in range(i+1):
            retour[i].append(0)
        for j in range(i+1,n):
            G = [[0,0] for _ in range(i)] + [[-1*E[i][j][0], -1*E[i][j][1]]] + [[0,0] for _ in
range(i+1,j)] + [[E[i][j][0],E[i][j][1]]] + [[0,0] for _ in range(j+1,n)]
            retour[i].append(dot(np.array(G),v))# dot correspond au produit scalaire canonique
        for l in range(n_obst):
            G = [[0,0] for _ in range(i)] + [[-1*E[i][n+l][0], -1*E[i][n+l][1]]] + [[0,0] for _ in
range(i+1,n)]
            retour[i].append(dot(np.array(G),v))
    return -1*h*np.array(retour)
def phi_star(v,E,n,n_obst):
    global h
    retour = np.array([[0.,0.] for i in range(n)])
    for i in range(n):
        for j in range(i+1,n):
            G = [[0,0] for _ in range(i)] + [[-1*E[i][j][0], -1*E[i][j][1]]] + [[0,0] for _ in
range(i+1,j)] + [[E[i][j][0],E[i][j][1]]] + [[0,0] for _ in range(j+1,n)]
            retour -=  v[i,j] * np.array(G)
        for l in range(n_obst):
            G = [[0,0] for _ in range(i)] + [[-1*E[i][n+l][0], -1*E[i][n+l][1]]] + [[0,0] for _ in
range(i+1,n)]
            retour -= v[i,n+l] * np.array(G)
    return h*retour
```

```
def projete(mu):
    mu_retour = mu.copy()
    for i in range(len(mu_retour)):
        for j in range(len(mu_retour[0])):
            if mu_retour[i,j] < 0:
                mu_retour[i,j] = 0
    return mu_retour

def uzawa(positions,u,D,E,n,n_obst,epsilon,iter_max,rho,listes_murs):
    global r
    global h

    positions_candidat = positions + h*u

    v = u.copy()
    k = 0

    liste_distances = [distance(tuple(positions_candidat[i]),tuple(positions_candidat[j])) -2*r for i in range(n) for j in 
range(i+1,n)]
    for i in range(n):
        for l in range(n_obst):
            liste_distances.append(min([distance(tuple(positions_candidat[i]), mur)-r for mur in listes_murs[l]]))
    Dmin = min(liste_distances)
    mu = np.array([[0 for j in range(n+n_obst)] for i in range(n)])

    while (k<iter_max) and (Dmin<-1*epsilon):
        v = u - phi_star(mu,E,n,n_obst)
        mu = projete(mu+rho*(phi(v,E,n,n_obst) - np.array(D)))

        positions_candidat = positions + h*v

        liste_distances = [distance(tuple(positions_candidat[i]),tuple(positions_candidat[j])) -2*r for i in range(n) for j 
in range(i+1,n)]
        for i in range(n):
            for l in range(n_obst):
                liste_distances.append(min([distance(tuple(positions_candidat[i]), mur)-r for mur in listes_murs[l]]))
        Dmin = min(liste_distances)
        k += 1

    print(Dmin)
```

```python
departs = set(coordonnées)
listes_murs = list[list[coordonnées]]


def mouvement_foule(dim,departs,listes_murs,directions):
    etapes_max = 1000
    nbr_individus = len(departs)
    positions = []
    etapes = []
    mouvement_fini = []
    for depart in departs:
        positions.append(depart)
    etapes.append(positions)
    vitesses_souhaitees_brutes = vitesses_souhaitees(dim,positions,directions)
    for i in range(len(positions)):
        if vitesses_souhaitees_brutes[i] == [0,0]:
            mouvement_fini.append(True)
        else:
            mouvement_fini.append(False)

    while False in mouvement_fini and len(etapes)< etapes_max:
```

...Page suivante →

```
    while False in mouvement_fini and len(etapes)< etapes_max:
        positions_nettoyees = []
        vitesses_souhaitees_nettes = []
        vitesses_souhaitees_brutes = vitesses_souhaitees(dim,positions,directions)
        for i in range(len(vitesses_souhaitees_brutes)):
            if vitesses_souhaitees_brutes[i] == [0,0]:
                mouvement_fini[i] = True
            else:
                positions_nettoyees.append(list(positions[i]))
                vitesses_souhaitees_nettes.append(vitesses_souhaitees_brutes[i])
        if len(positions_nettoyees)==0:
            break

        D,E,n,n_obst,epsilon,iter_max,rho = uzawa_initialisation(positions_nettoyees,listes_murs)
        vitesses_souhaitees_nettes = np.array(vitesses_souhaitees_nettes)
        positions_nettoyees = np.array(positions_nettoyees)

        nouvelles_positions =
uzawa(positions_nettoyees,vitesses_souhaitees_nettes,D,E,n,n_obst,epsilon,iter_max,rho,listes_murs)

        positions = []
        j = 0
        for i in range(nbr_individus):
            if mouvement_fini[i] == True:
                positions.append(None)
            else:
                positions.append(tuple(nouvelles_positions[j]))
                j += 1

        etapes.append(positions)

    return etapes
```

```python
r = 1
R = 3
h = 1
def uzawa_initialisation_sanitaire(positions,listes_murs):
    global r
    global R

    n = len(positions)
    n_obst = len(listes_murs)

    D = []
    E = []
    for q1 in range(n):
        D.append([])
        E.append([])

        for q2 in range(n):
            d = distance(positions[q1],positions[q2])
            D[q1].append(d - 2*R)
```

[...]

```python
        for obst in range(n_obst):
            pt_proche = (np.inf,np.inf)
            d = distance(positions[q1],pt_proche)
            for mur in listes_murs[obst]:
                if distance(positions[q1],mur) < d:
                    pt_proche = mur
                    d = distance(positions[q1],pt_proche)
            D[q1].append(d-r)
```

[...]

```python
    epsilon_murs = 0.1*r
    epsilon_sanitaire = 0.5*R
    iter_max = 5000
    rho = 1

    return D,E,n,n_obst,epsilon_sanitaire,epsilon_murs,iter_max,rho
```

```python
def
uzawa_sanitaire(positions,u,D,E,n,n_obst,epsilon_sanitaire,epsilon_murs,iter_max,rho,listes_murs):
    global r
    global R
    global h

    positions_candidat = positions + h*u

    v = u.copy()
    k = 0

    liste_distances_sanitaire = [distance(tuple(positions_candidat[i]),tuple(positions_candidat[j]))
-2*R for i in range(n) for j in range(i+1,n)]
    if len(liste_distances_sanitaire) >= 1:
        Dmin_sanitaire = min(liste_distances_sanitaire)
    else:
        Dmin_sanitaire = np.inf

    liste_distances_murs = []
    for i in range(n):
        for l in range(n_obst):
            liste_distances_murs.append(min([distance(tuple(positions_candidat[i]), mur)-r for mur in
listes_murs[l]]))
    Dmin_murs = min(liste_distances_murs)
```

[...]