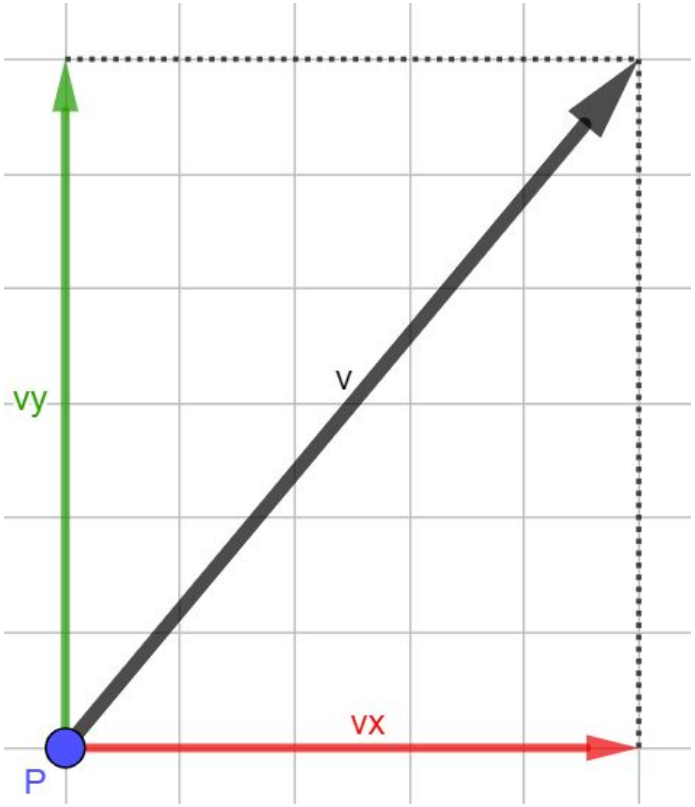
Optimisation d’une évacuation

GAUTIER Arthur
N°candidat : 36917

lfgroupe.com

Sommaire

Modélisation des points




```
particule = [ x , y , vx , vy]
```

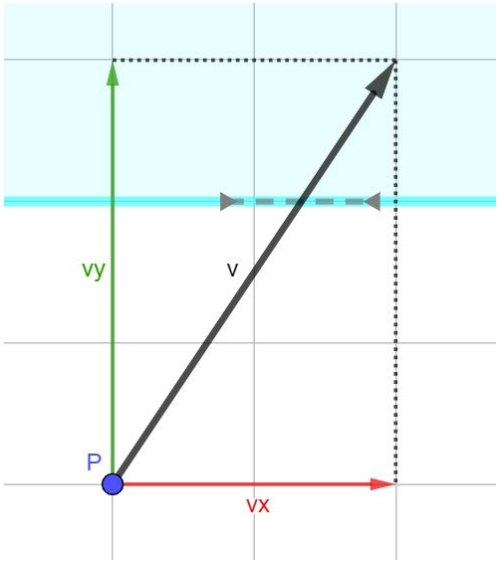
```
foule =
[ particule , particule ,
particule ]
```

```
creer_particules(n)
```

Faire évacuer les particules

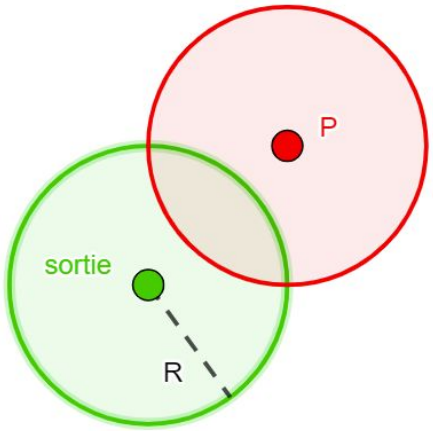
1ere version

```
est_sortie(particule)
```



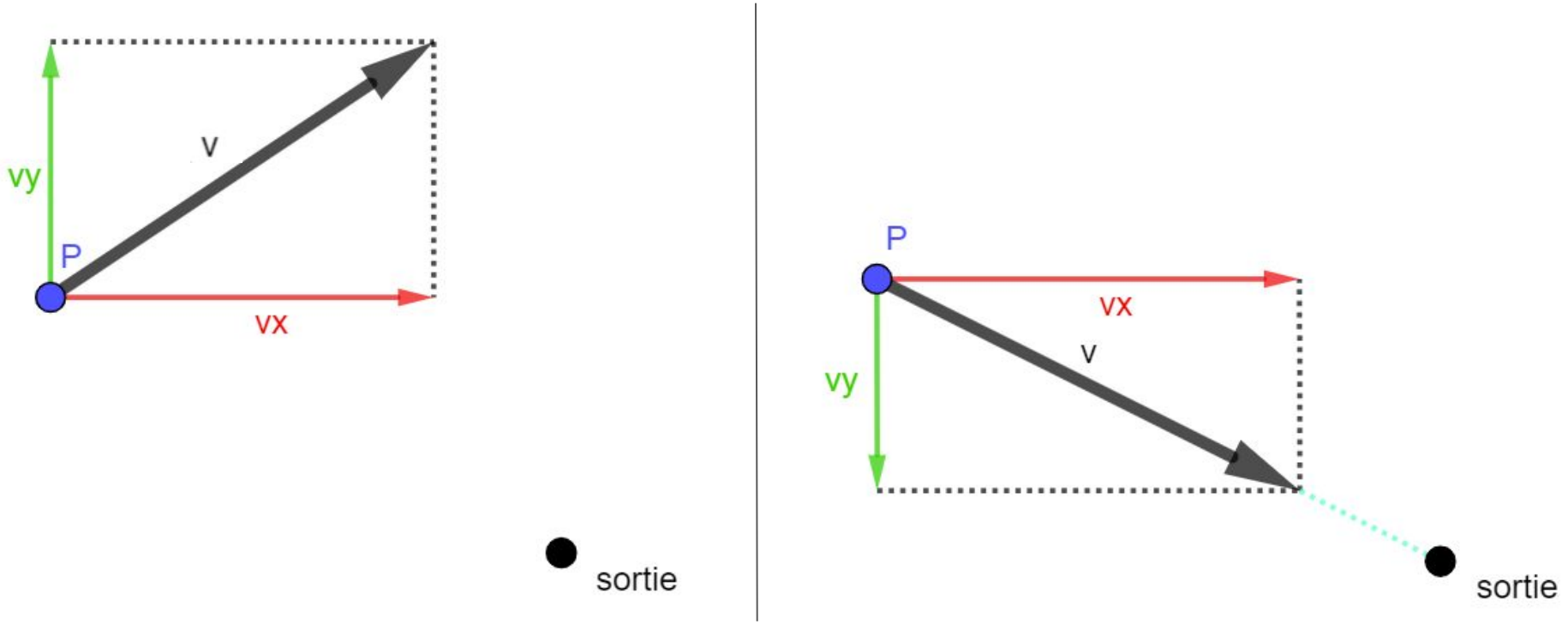
2e version

```
est_sortie2(particule,liste_sortie)
```



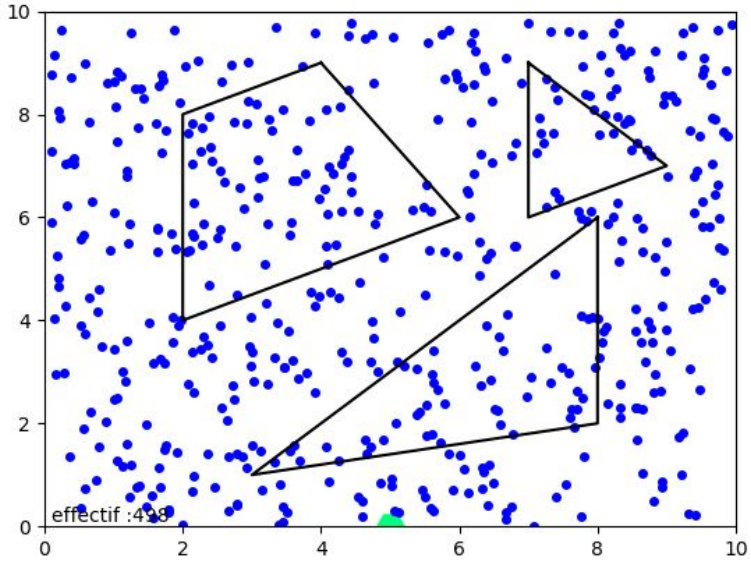

```
collision(point1,point2)
```

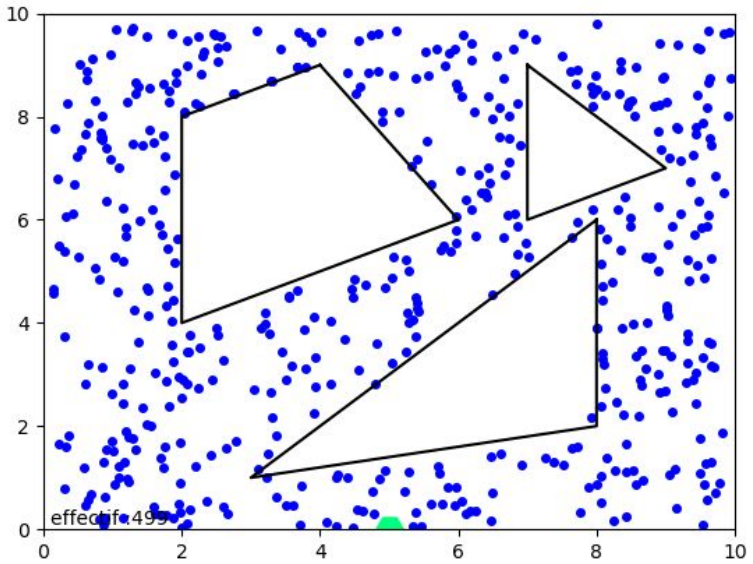
Orienter les particules




```
orientation_vers_sortie(particule,sortie)
```

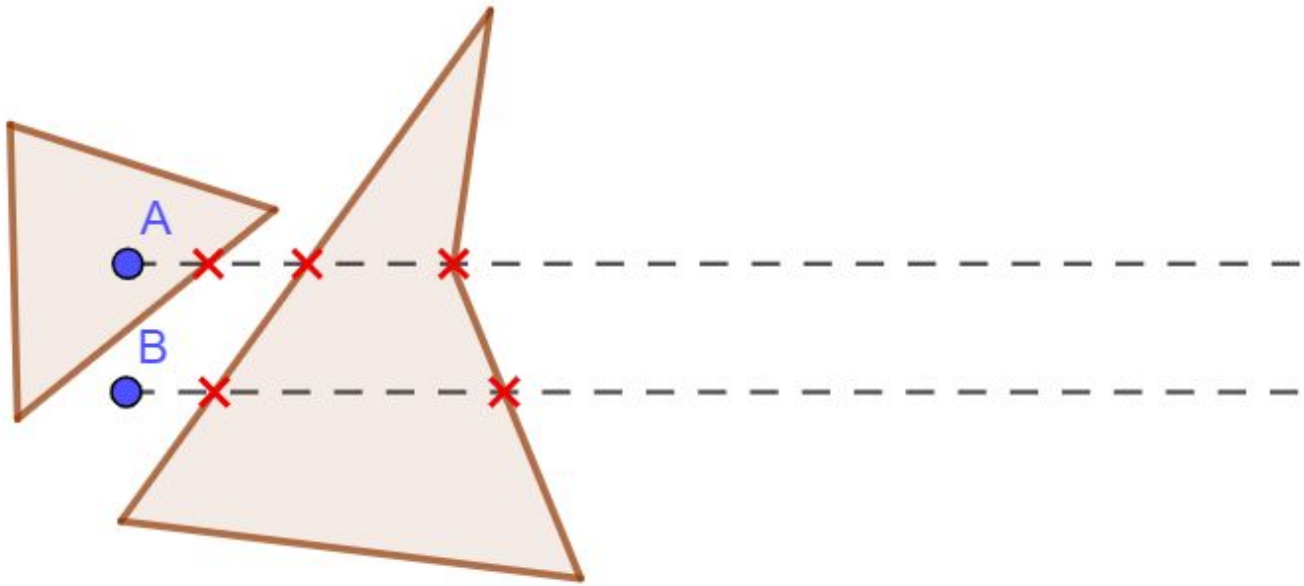
Génération des points





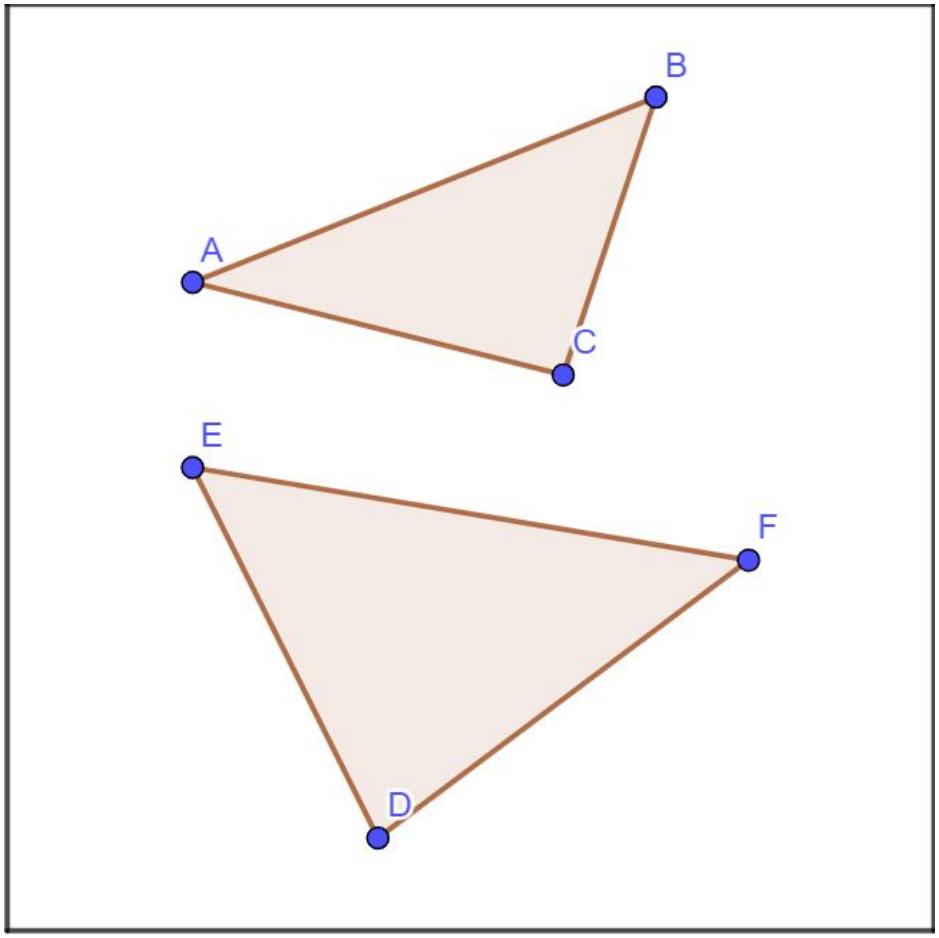
Génération des points

```
inobstacle(particule)
```



Modélisation du graphe : Etape 1

`matrice_obstacle(obstacle,liste_sortie)`



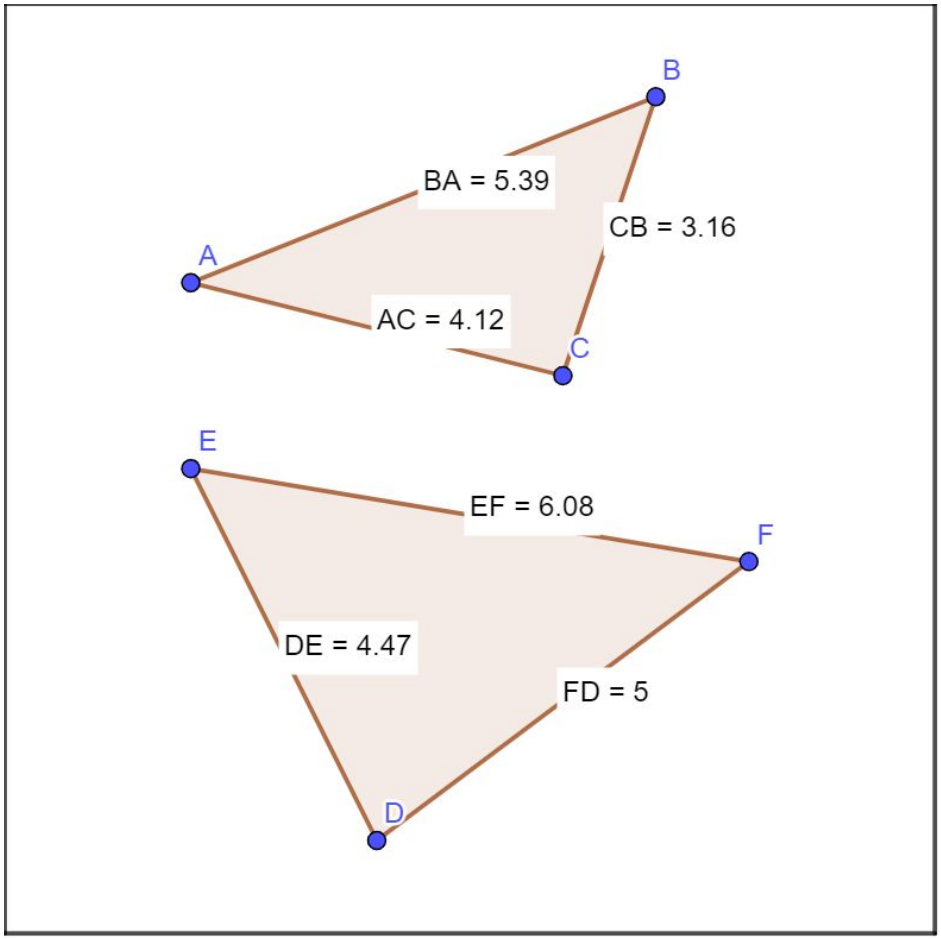
$$
\begin{array}{r}
A \rightarrow \\ B \rightarrow \\ C \rightarrow \\ D \rightarrow \\ E \rightarrow \\ F \rightarrow \\ sortie \rightarrow \\ particule \rightarrow
\end{array}
\begin{pmatrix}
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0
\end{pmatrix}
\begin{array}{cccccccc}
\uparrow & \uparrow & \uparrow & \uparrow & \uparrow & \uparrow & \uparrow & \uparrow \\
A & B & C & D & E & F & sortie & particule
\end{array}
$$

```
obstacle = [  [ [x,y] , [x,y], [x,y] ],
              [ [x,y] , [x,y], [x,y] ]  ]
```

obstacle 1

obstacle 2

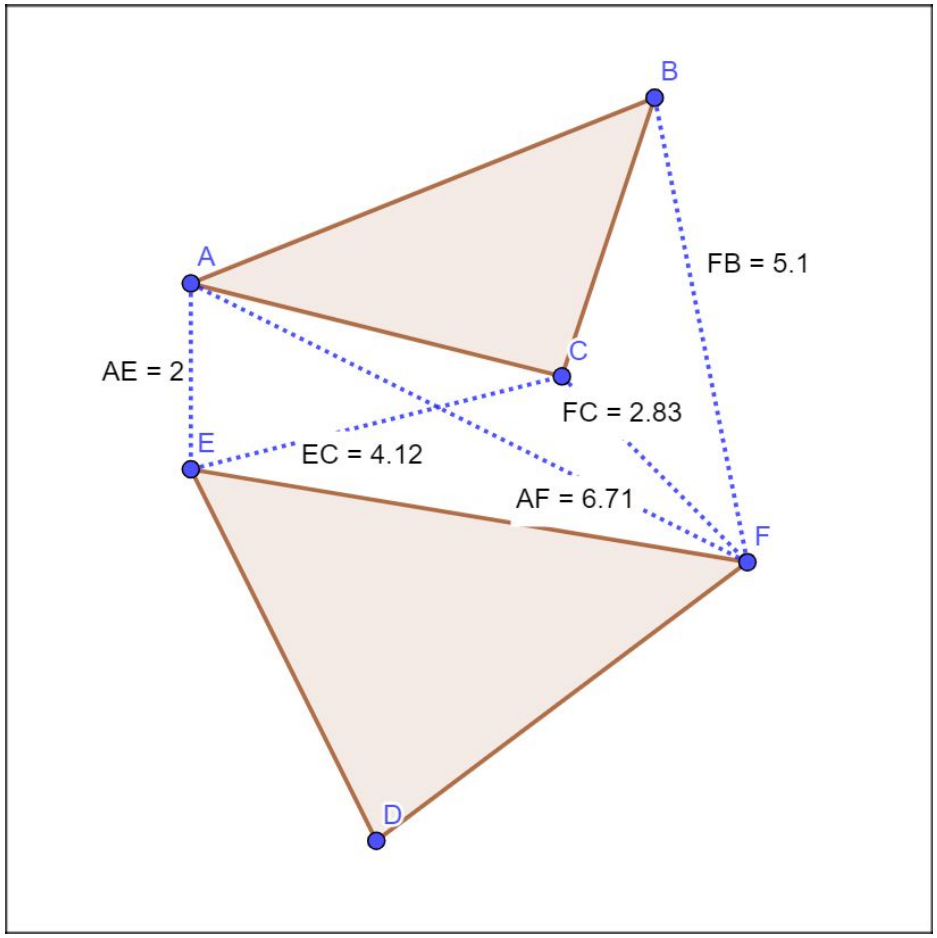
Modélisation du graphe : Etape 2



$$\begin{array}{r} A \rightarrow \\ B \rightarrow \\ C \rightarrow \\ D \rightarrow \\ E \rightarrow \\ F \rightarrow \\ sortie \rightarrow \\ particule \rightarrow \end{array} \begin{pmatrix} 0 & 5.39 & 4.12 & 0 & 0 & 0 & 0 & 0 \\ 5.39 & 0 & 3.16 & 0 & 0 & 0 & 0 & 0 \\ 4.12 & 3.16 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 4.47 & 5 & 0 & 0 \\ 0 & 0 & 0 & 4.47 & 0 & 6.08 & 0 & 0 \\ 0 & 0 & 0 & 5 & 6.08 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

```
distance(point_1,point_2)
```

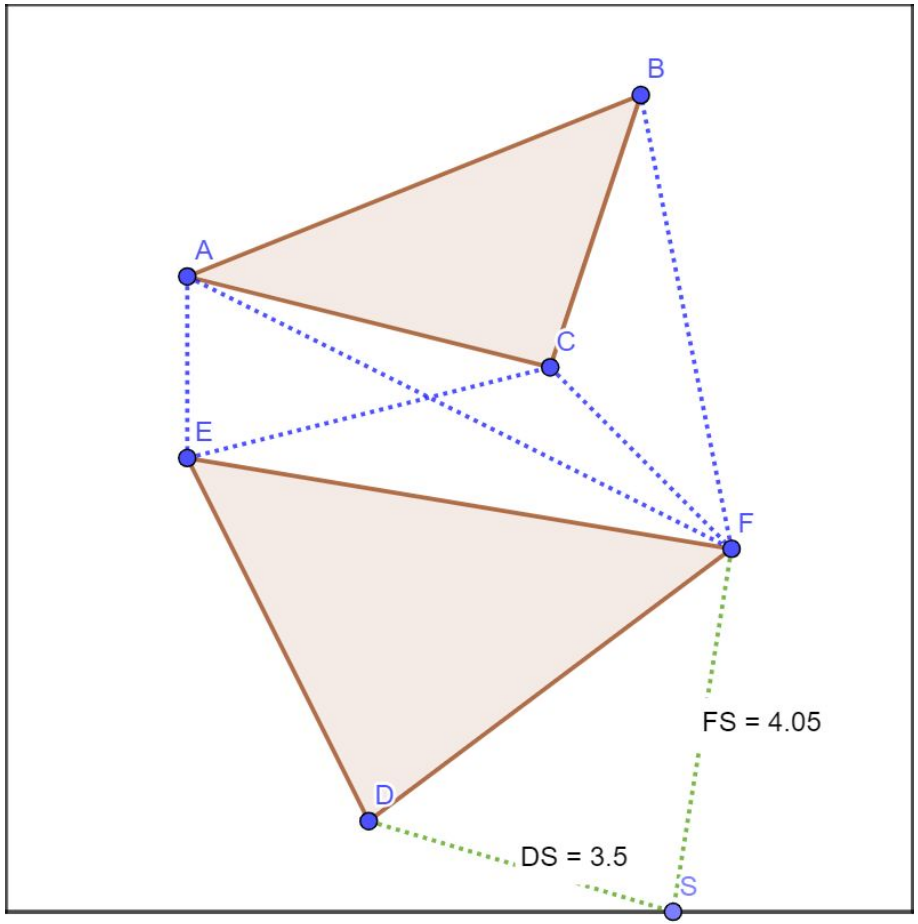
Modélisation du graphe : Etape 3



$$
\begin{array}{r}
A \rightarrow \\
B \rightarrow \\
C \rightarrow \\
D \rightarrow \\
E \rightarrow \\
F \rightarrow \\
sortie \rightarrow \\
particule \rightarrow
\end{array}
\begin{pmatrix}
0 & 5.39 & 4.12 & 0 & 2 & 6,71 & 0 & 0 \\
5.39 & 0 & 3.16 & 0 & 0 & 5.1 & 0 & 0 \\
4.12 & 3.16 & 0 & 0 & 4.12 & 2.83 & 0 & 0 \\
0 & 0 & 0 & 0 & 4.47 & 5 & 0 & 0 \\
2 & 0 & 4.12 & 4.47 & 0 & 6.08 & 0 & 0 \\
6,71 & 5.1 & 2.83 & 5 & 6.08 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0
\end{pmatrix}
$$

```
presence_obstacle(point1,point2)
```

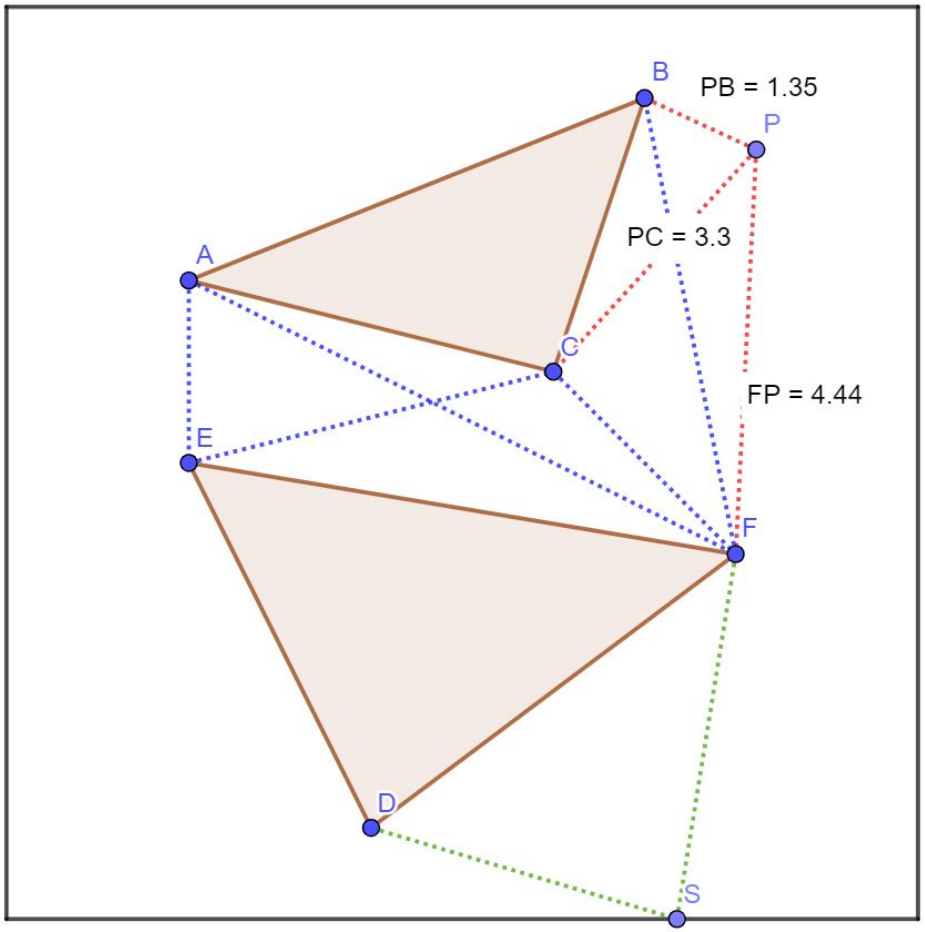
Modélisation du graphe : Etape 4



$$
\begin{array}{r}
A \rightarrow \\
B \rightarrow \\
C \rightarrow \\
D \rightarrow \\
E \rightarrow \\
F \rightarrow \\
sortie \rightarrow \\
particule \rightarrow
\end{array}
\begin{pmatrix}
0 & 5.39 & 4.12 & 0 & 2 & 6,71 & 0 & 0 \\
5.39 & 0 & 3.16 & 0 & 0 & 5.1 & 0 & 0 \\
4.12 & 3.16 & 0 & 0 & 4.12 & 2.83 & 0 & 0 \\
0 & 0 & 0 & 0 & 4.47 & 5 & \boxed{3.5} & 0 \\
2 & 0 & 4.12 & 4.47 & 0 & 6.08 & 0 & 0 \\
6,71 & 5.1 & 2.83 & 5 & 6.08 & 0 & \boxed{4.05} & 0 \\
0 & 0 & 0 & \boxed{3.5} & 0 & \boxed{4.05} & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0
\end{pmatrix}
$$

Modélisation du graphe : Etape 5

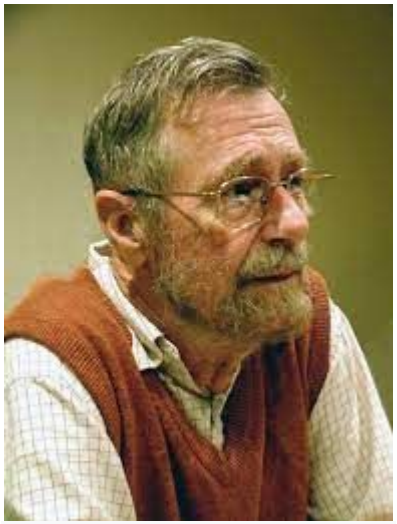
```
matrice_point(matrice_obstacle,particule)
```



$$
\begin{array}{r}
A \rightarrow \\
B \rightarrow \\
C \rightarrow \\
D \rightarrow \\
E \rightarrow \\
F \rightarrow \\
sortie \rightarrow \\
\mathit{rticule} \rightarrow
\end{array}
\begin{pmatrix}
0 & 5.39 & 4.12 & 0 & 2 & 6,71 & 0 & 0 \\
5.39 & 0 & 3.16 & 0 & 0 & 5.1 & 0 & \boxed{1.35} \\
4.12 & 3.16 & 0 & 0 & 4.12 & 2.83 & 0 & \boxed{3.3} \\
0 & 0 & 0 & 0 & 4.47 & 5 & 3.5 & 0 \\
2 & 0 & 4.12 & 4.47 & 0 & 6.08 & 0 & 0 \\
6,71 & 5.1 & 2.83 & 5 & 6.08 & 0 & 4.05 & \boxed{4.44} \\
0 & 0 & 0 & 3.5 & 0 & 4.05 & 0 & 0 \\
0 & \boxed{1.35} & \boxed{3.33} & 0 & 0 & \boxed{4.44} & 0 & 0
\end{pmatrix}
$$

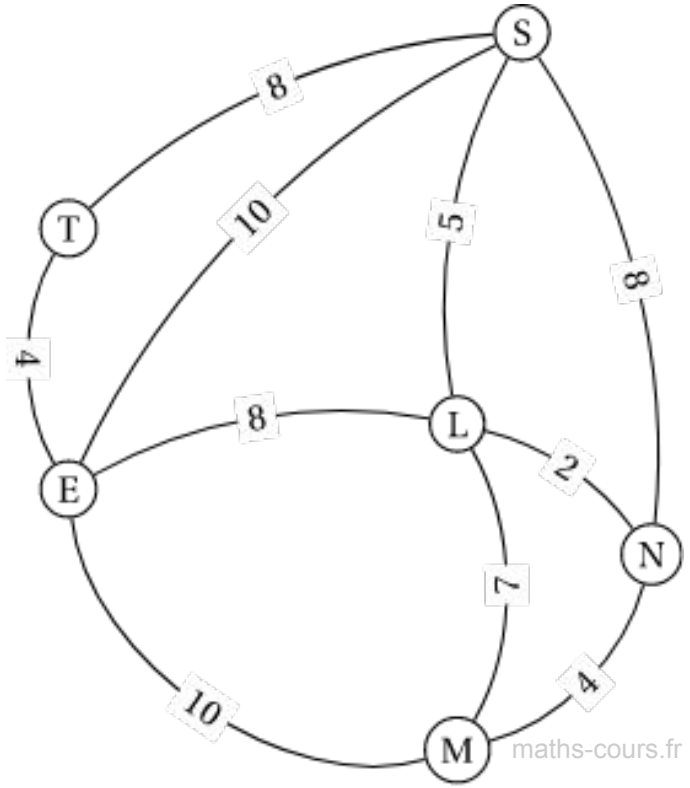
Etape 6

-Utilisation de Dijkstra



manuscripts of Edsger W. Dijkstra, University Texas at Austin

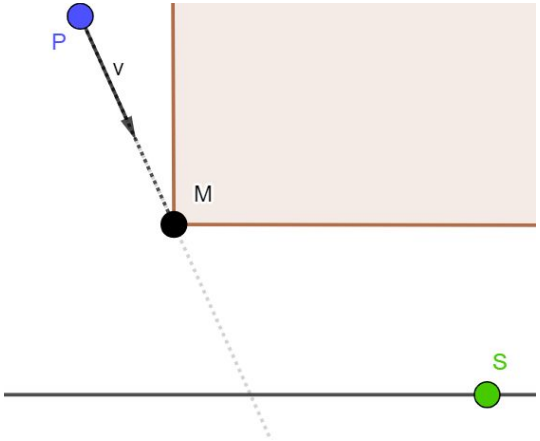
1930 - 2002



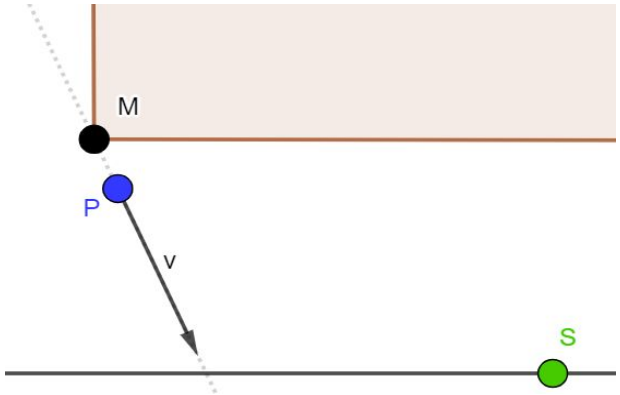

```
dijkstra(M,s)
```

```
pluscourtchemin(M,entree,sortie)
```

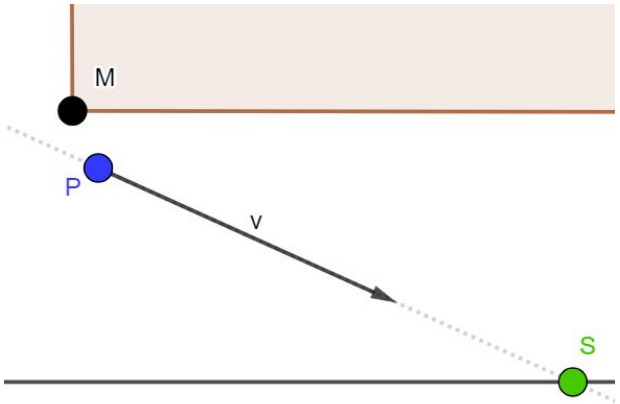
Suivi du parcours : version 1



[M,S]

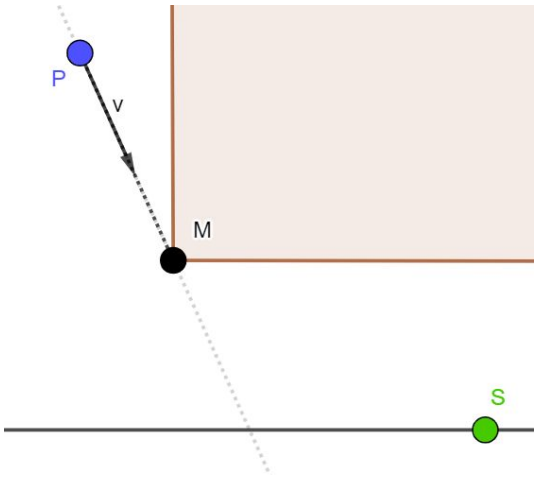


[M,S]

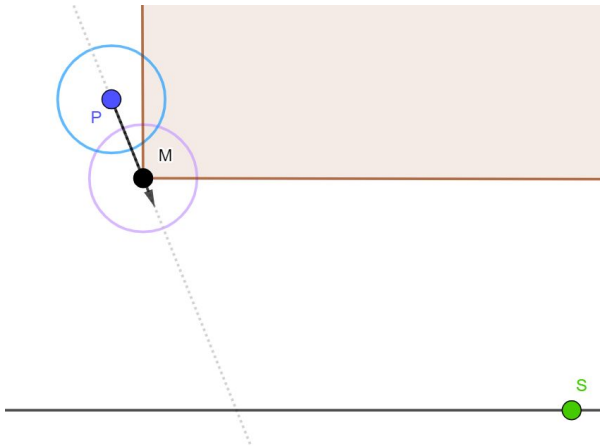


[S]

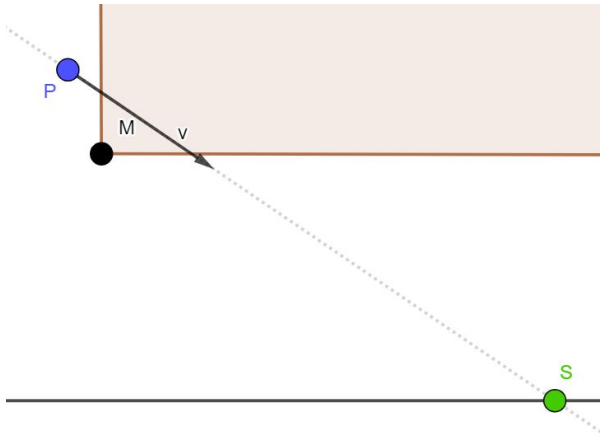
Suivi du parcours : version 2



[M,S]

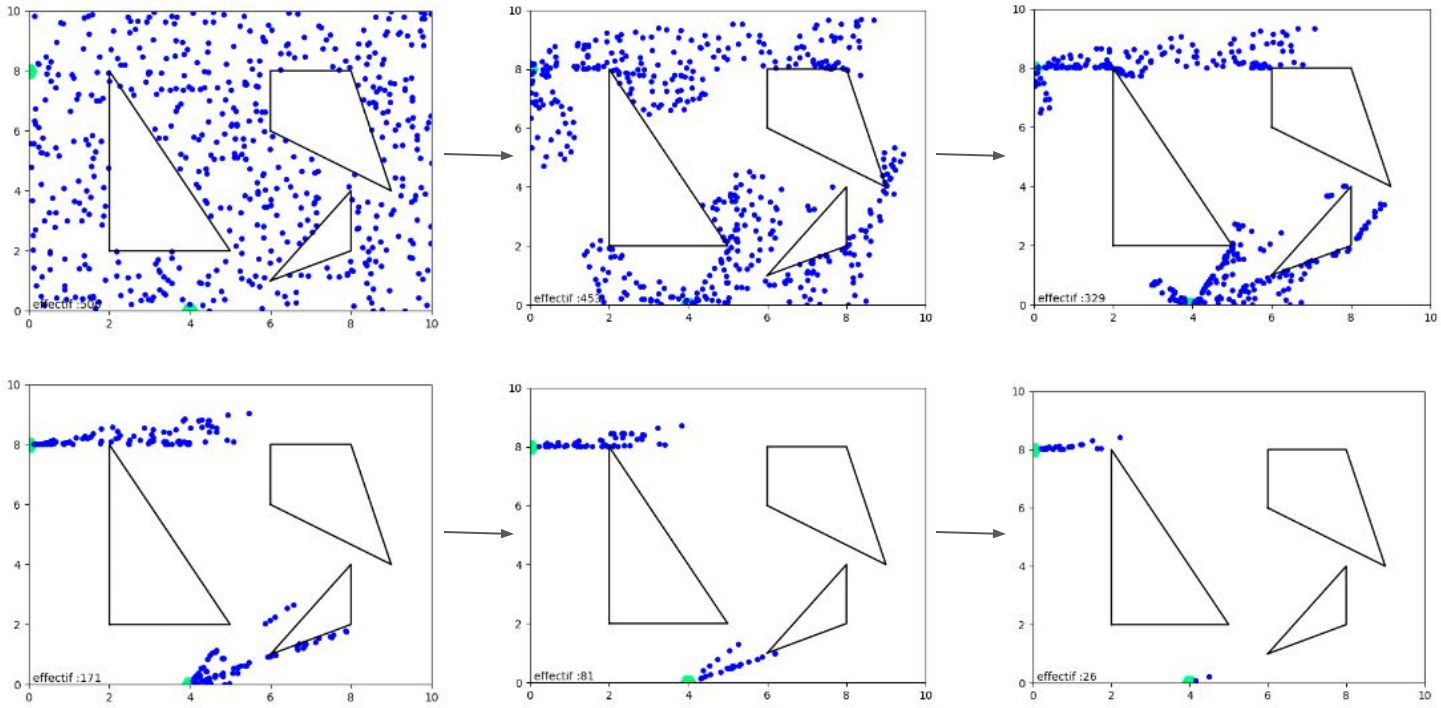


[M,S]

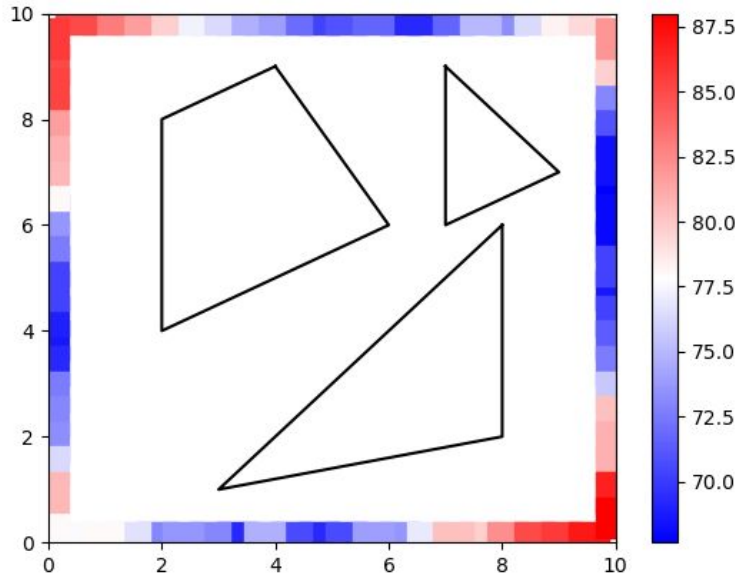


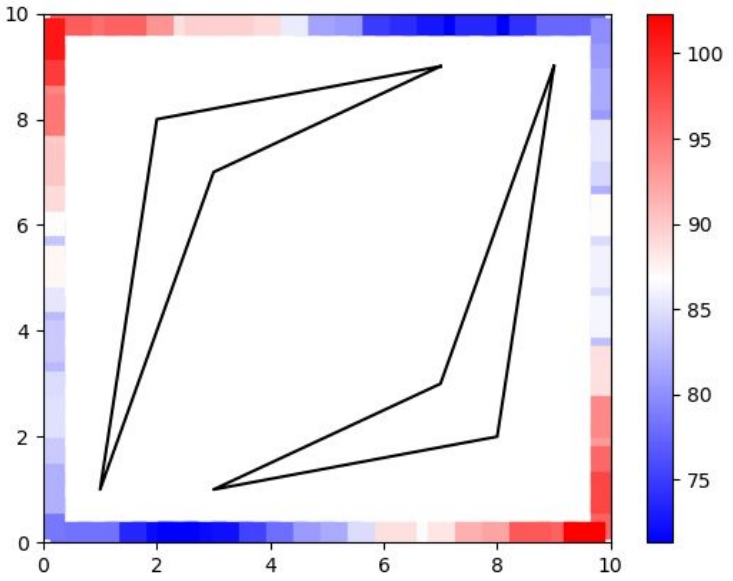
[S]

Rendu de la modélisation



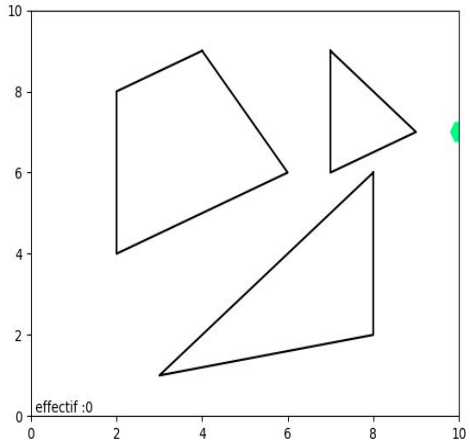
Recherche de sorties optimisées



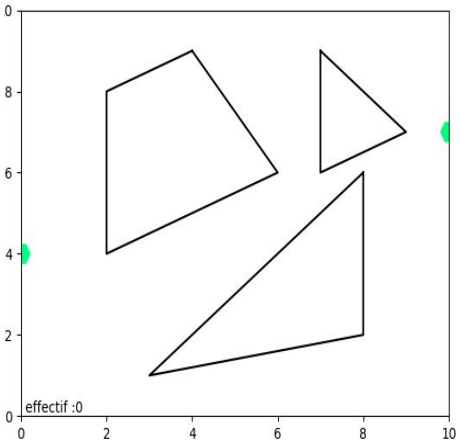


80 sorties testées avec une foule de 500 points

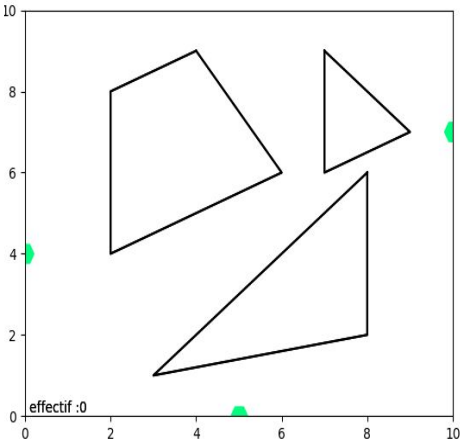
Présence de plusieurs sorties



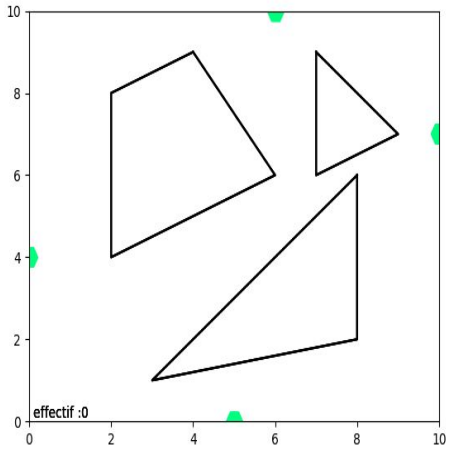
1 sortie
69 images



2 sorties
43 images



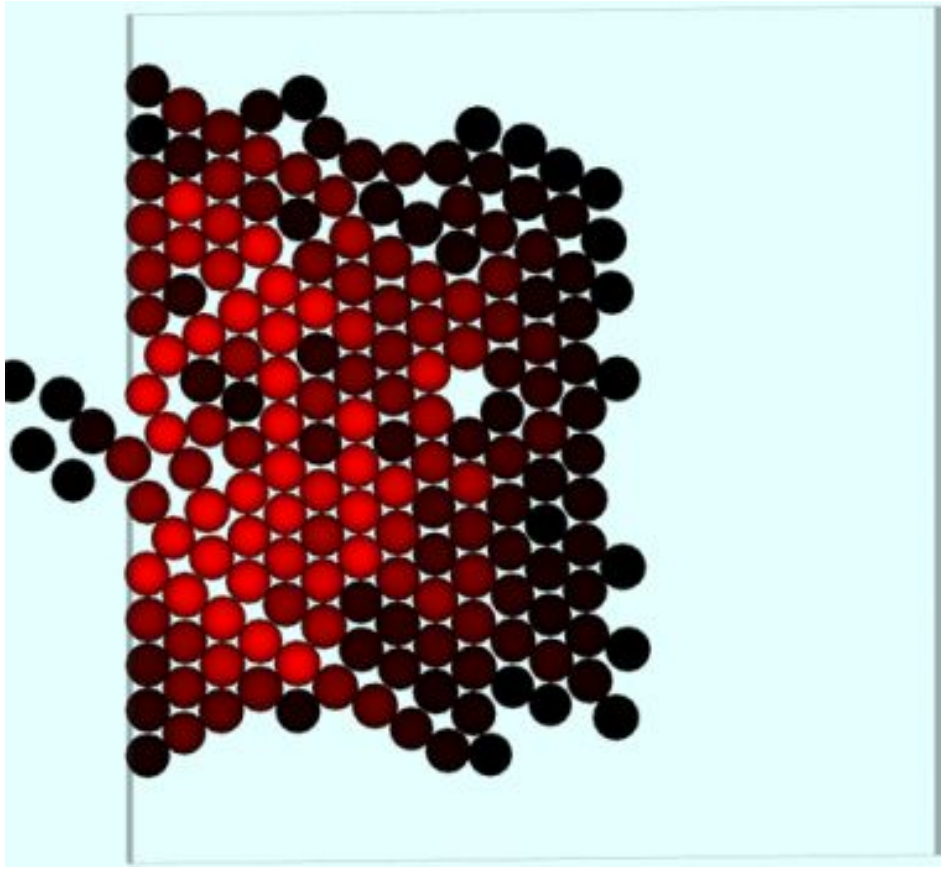
3 sorties
38 images



4 sorties
32 images

Temps obtenu avec 16 sorties : 29 images

Limite du modèle et conclusion



http://images.math.cnrs.fr/

FIN

```python
import matplotlib.pyplot as plt
import matplotlib.animation as animation
import numpy as np
from math import *
from pylab import *
import random

##Constantes :
#dimensions de la salle :
largeur = 10
hauteur = 10
taille_porte =0.1
rayon = 0.25

#temps elementaire, en seconde
t = 0.20

#obstacles et sorties
obstacle = [[[2,2],[5,2],[2,8],[2,2]],[[6,1],[8,2],[8,4],[6,1]],[[6,6],[6,8],[8,8],[9,4],[6,6]]]
liste_sortie = [[4,0],[0,8]]
```

```python
###creation matrice
"""permet de creer une matrice de particule"""
def creer_particules(n):
    particules = []

    for k in range(0,n):
        x = random.random()*largeur
        y = random.random()*hauteur
        vx = (random.random()*0.5 + 1.1)*random.choice([-1,1])
        vy = (random.random()*0.5 + 1.1)*random.choice([-1,1])
        p = [x,y,vx,vy]
        while inobstacle(p) == True:
            x = random.random()*largeur
            y = random.random()*hauteur
            vx = (random.random()*0.5 + 1.1)*random.choice([-1,1])
            vy = (random.random()*0.5 + 1.1)*random.choice([-1,1])
            p = [x,y,vx,vy]

        particules.append(p)

    return particules
```

```python
"""cree une matrice adaptee au probleme, satisfaisant l'algorithme de dijkstra"""
def matrice_obstacle(obstacle,liste_sortie):
    taille = 0
    for objet in obstacle:
        taille += len(objet)-1

    t_obstacle = taille #sauvegarde
    taille += len(liste_sortie) + 1

    matrice = []
    i = 0 #cet indice correspond a l'avancee dans la matrice,
    s = 0 #cet indice correspond au somme de points d'objets
    for k in range(taille):
        matrice.append([0 for k in range(taille)])

#cette partie permet de calculer les distances entres points objets
    for objet in obstacle:
        for k in range(len(objet)-2):
            norme = distance(objet[k],objet[k+1])
            matrice[i][i+1]=norme
            matrice[i+1][i]=norme
            i+=1
        if len(objet)>2:
            j = len(objet)-2
            norme = distance(objet[0],objet[j])
            matrice[i][i-j]=norme
            matrice[i-j][i]=norme
        i += 1
#cette partie permet de déterminer et calculer les liens entre objets et sorties
```

```
#cette partie permet de déterminer et calculer les liens entre objets et sorties
    i1=0
    for ob1 in range(len(obstacle)):
        for pt1 in range(len(obstacle[ob1])-1):

            i2 = 0

            for k in range(len(liste_sortie)):
                if presence_obstacle(liste_sortie[k],obstacle[ob1][pt1])==False:
                    A = obstacle[ob1][pt1]
                    B = liste_sortie[k]
                    norme = distance(A,B)
                    matrice[t_obstacle + k][i1] = norme
                    matrice[i1][t_obstacle + k] = norme

            for ob2 in range(len(obstacle)):
                if obstacle[ob1]==obstacle[ob2]:
                    i2 += len(obstacle[ob1])-1
                    continue
                for pt2 in range(len(obstacle[ob2])-1):
                    A = obstacle[ob1][pt1]
                    B = obstacle[ob2][pt2]
                    if presence_obstacle(A,B) == False:
                        norme = distance(A,B)
                        matrice[i1][i2] = norme
                        matrice[i2][i1] = norme
                    i2 += 1
            i1 += 1

    return matrice
```

```python
"""cree une matrice propre à chaque particules à partir d'une matrice obstacle"""
def matrice_point(matrice,particule):
    x,y,vx,xy = particule
    n = len(matrice)-1
    t_obstacle = n - len(liste_sortie)
    pos = [x,y]
    i=0
    for ob in range(len(obstacle)):
        for pt in range(len(obstacle[ob])-1):

            for k in range(len(liste_sortie)):
                if presence_obstacle(liste_sortie[k],pos)==False:
                    norme = distance(pos,liste_sortie[k])
                    matrice[t_obstacle + k][n] = norme
                    matrice[n][t_obstacle + k] = norme

            point_ob = obstacle[ob][pt]

            if presence_obstacle(pos,point_ob)==False:
                norme = distance(pos,point_ob)
                matrice[n][i] = norme
                matrice[i][n] = norme
            i += 1

    return matrice

###deplacement
"""permet de deplacer une unique particule"""
def deplacerParticule(particule) :
    x, y, vx, vy = particule

    if x+vx*t >= largeur or x+vx*t <= 0 :
        vx = -vx
    if y+vy*t >= hauteur or y+vy*t <= 0 :
        vy = -vy
    return [x+vx*t, y+vy*t, vx, vy]

```

```
"""première version : permet de deplacer la foule"""
def deplacement_type1(foule):
    for k in range(0,len(foule)):
        foule[k] = deplacerParticule(foule[k])
    return foule

"""deuxième version : permet de deplacer toute la foule en considérant les sorties seulement"""
def deplacement_type2(foule,liste_sortie):
    k = 0
    taille_liste = len(foule)
    while k < taille_liste:
        if est_sortie2(foule[k],liste_sortie)==True:
            foule.pop(k)
            k = k - 1
        else :
            foule[k] = deplacerParticule(foule[k])
        k = k + 1
        taille_liste = len(foule)
    return foule
"""permet de déplacer la foule avec suivi d'un plan"""
def deplacement_type3(foule,liste_sortie,plan):
    k = 0
    taille_liste = len(foule)
    while k < taille_liste:
        if est_sortie2(foule[k],liste_sortie)==True:
            foule.pop(k)
            plan.pop(k)
            k = k - 1
        else :
            foule[k] = deplacerParticule(foule[k])
        k = k + 1
        taille_liste = len(foule)
    return foule, plan
"""determine un itinéraire pour une particule"""
def itineraire(particule,liste_sortie):
    M = matrice_obstacle(obstacle,liste_sortie)
    matrice = matrice_point(M,particule,liste_sortie)

    s = 100000000000

    itineraire_retenu = 0

    for k in range(len(liste_sortie)):
        chemin = pluscourtchemin(matrice,len(matrice)-1,t_obstacle + k)
        if trajet(chemin,particule,liste_sortie) < s:
            s = trajet(chemin,particule,liste_sortie)
            itineraire_retenu = chemin
    return itineraire_retenu
```

```python
"""determine un itineraire pour chaque particule"""
def plan_evacuation(foule,liste_sortie):
    liste_iti = []
    for k in range(0,len(foule)):
        liste_iti.append(itineraire(foule[k],liste_sortie))
    for k in liste_iti:
        k.pop(0)
    return liste_iti

##Gestion des obstacles et sorties
"""premiere version : permet de verifier si une personne est sortie en utilisant une methode geometrique"""
def est_sortie(particule):
    x, y, vx, vy = particule

    for k in liste_sortie:
        sortie_x,sortie_y = k

        if x+vx >= largeur or x+vx <=0:
            base_petit_triangle = sortie_x - x

            ptAy = y + base_petit_triangle*vy/vx

            if abs(ptAy -sortie_y) <= taille_porte/2:
                return True


        if y+vy >= hauteur or y+vy <= 0:
            base_petit_triangle = (y + vy - sortie_y)*vx/vy

            ptAx = x + vx - base_petit_triangle

            if abs(ptAx - sortie_x) <= taille_porte/2:
                return True
    return False

"""deuxieme version : permet de verifier si une particule est sorti en utilisant la methode de collision"""
def est_sortie2(particule,liste_sortie):
    x,y,vx,vy = particule

    for k in liste_sortie:
        xs,ys =k
        if collision([x,y],[xs,ys]) == True:
            return True
    return False
```

```
"""permet de choisir une sortie optimisee pour la particule sans considérer les obstacles"""
def determination_sortie_naif(particule):
    x,y,vx,vy = particule
    d = sqrt((x-liste_sortie[0][0])**2 + (y-liste_sortie[0][1])**2)
    s = 0
    for k in liste_sortie:
        x_sortie,y_sortie = k
        distance = sqrt((x-x_sortie)**2 + (y-y_sortie)**2)
        if distance <= d:
            d = distance
            s = k
    return s

"""oriente la trajectoire d'une particule vers la sortie"""

#ce programme fonctionne egalement pour orienter une particule vers n'importe quel point de la salle

def orientation_vers_sortie(particule,sortie):
    sortie_x,sortie_y = sortie
    x,y,vx,vy = particule
    #distance entre la particule et la sortie
    v = sqrt(vx**2 + vy**2)
    d = sqrt((x-sortie_x)**2 + (y-sortie_y)**2)
    #angle entre la droite (particule - sortie) et l'axe des abscisses
    teta = np.arccos((x-sortie_x)/d)
    #calcul_vx
    if y >= sortie_y :
        vx = -cos(teta)*v
        vy = -sin(teta)*v
    if y <= sortie_y:
        vx = -cos(-teta)*v
        vy = -sin(-teta)*v
    return [x, y, vx, vy]

"""detecte la presence d'obstacles entre un point et les sorties"""
def presence_obstacle(point1,point2):
    xp,yp = point1
    xs,ys = point2

    #cas où les points abscisses sont différents
    if xs!=xp:
        a1 = (ys-yp)/(xs-xp)
        b1 = ys - a1*xs

    #pour chaque paire de points obstacle je calcule les equations de droite
    for objet in obstacle:
        for k in range(len(objet)-1):
            xob1,yob1 = objet[k]
            xob2,yob2 = objet[k+1]


            if xob1 == xp and yob1 == yp:
                continue
```

```
            if xob2 == xp and yob2 == yp:
                continue
            if xob1 == xs and yob1 == ys:
                continue
            if xob2 == xs and yob2 == ys:
                continue

            if xob2 != xob1:
                a2 = (yob2 - yob1)/(xob2-xob1)
                b2 = yob1 - a2*xob1

            if xs == xp:
                if xob2 ==xob1: #TODO vérifier que le problème est souvelé
                    continue
                y = a2*xs + b2
                if (min(ys,yp) <= y <= max(ys,yp)) and (min(xob1,xob1) <= xs <= max(xob1,xob2)):
                    return True
                else:
                    continue

            if xob2 == xob1:
                y = a1*xob1+b1
                if (min(yob1,yob2) <= y <= max(yob1,yob2)) and (min(xs,xp) <= xob2 <= max(xs,xp)):
                    return True
                else :
                    continue

            if a1 != a2:
                x = (b2-b1)/(a1-a2)
                if egalite(a1*x+b1,a2*x+b2) == True:
                    y = a1*x + b1
                    if ((min(yob1,yob2) <= y <= max(yob1,yob2)) and (min(xob1,xob2) <= x <= max(xob1,xob2))) and ((min(ys,yp) <= y <= max(ys,yp))
                        return True
            #cas particulier
    return False

##affichage
"""permet d'afficher la modelisation"""
def tracer(particules,type_deplacement,liste_sortie,plan):

    if type_deplacement == 1:
        deplacement_type1(particules)

    if type_deplacement == 2:
        deplacement_type2(particules,liste_sortie)

    if type_deplacement == 3:
        deplacement_type3(particules,liste_sortie,plan)

    X = [particules[k][0] for k in range(0,len(particules))]
    Y = [particules[k][1] for k in range(0,len(particules))]
    plt.axis([0, largeur, 0, hauteur])
    plt.plot(X,Y,"ob",markersize=4)
```

```
    #permet de tracer des obstacles
    dessin(obstacle)
    #x = np.array([largeur/4, 3*largeur/4])
    #y = np.array([hauteur/2, hauteur/2])
    #plt.plot(x, y)
    plt.text(0.1,0.1,"effectif :" + str(len(particules)))
    savefig("OneDrive\Bureau\TIPE\image1.png")
    plt.show()

"""permet de representer les obstacles"""
def dessin(obstacle):
    X = []
    Y = []
    for objet in obstacle:
        x = []
        y = []
        for point in objet:
            x.append(point[0])
            y.append(point[1])
        X.append(x)
        Y.append(y)
    for sortie in liste_sortie:
        a,b =sortie
        plt.scatter(a, b, s = 200, c = 'springgreen',marker= "H")
    for k in range(len(X)):
        plt.plot(np.array(X[k]),np.array(Y[k]),c = 'black')

"""permet de visualiser les sorties pertinentes"""
def affich_sortie(sorties,temps):
    dessin(obstacle)
    min = temps[0]
    max = temps[len(temps)-1]



    for k in range(len(sorties)):
        x,y = sorties[k]
        norm=plt.Normalize(min,max)
        color = plt.cm.get_cmap('plasma')
        plt.scatter(x, y, c = temps[k], cmap='bwr',norm=norm,marker= "s",s=400)

    plt.colorbar()

    for k in range(0,0):
        a,b =sorties[k]
        plt.scatter(a, b, s = 400, c = 'c',barker= "s")

    plt.axis([0, largeur, 0, hauteur])

    plt.show()
```

```python
##Partie evacuation
"""oriente la trajectoire de chacune des particules vers la sortie, sans considerer les obstacles"""
def lancement_evacuation_naif(foule):
    for k in range(0,len(foule)):
        sortie = determination_sortie_naif(foule[k])
        foule[k] = orientation_vers_sortie(foule[k],sortie)

"""permet d'assurer le suivi de l'itinéraire pour une particule"""
def suivi_parcours(particule,chemin,liste_sortie):
    sommets = point_graph(obstacle,liste_sortie)
    x,y,vx,vy = particule

    if len(chemin) != 1 and collision([x,y],sommets[chemin[0]])==True:
        chemin.pop(0)

    return chemin

"""lance l'evacuation"""
def lancement_evacuation(foule,liste_sortie):
    plan = plan_evacuation(foule,liste_sortie)

    sommets = point_graph(obstacle,liste_sortie)

    for k in range(len(foule)):
        foule[k] = orientation_vers_sortie(foule[k],sommets[plan[k][0]])

"""permet de faire tourner le programme de maniere non optimale"""
#mouvement de foule en apparence plus realiste
def test1(foule,liste_sortie,draw):
    lancement_evacuation(foule,liste_sortie)
    if draw == True:
        tracer(foule,2,liste_sortie,plan)
    else:
        deplacement_type2(foule,liste_sortie)

"""permet de faire tourner le programme de maniere optimale"""
#mp
def test2(foule,plan,liste_sortie,draw):

    sommets = point_graph(obstacle,liste_sortie)

    for k in range(len(foule)):
        plan[k] = suivi_parcours(foule[k],plan[k],liste_sortie)
        foule[k] = orientation_vers_sortie(foule[k],sommets[plan[k][0]])
    if draw == True:
        tracer(foule,3,liste_sortie,plan)
    else :
        deplacement_type3(foule,liste_sortie,plan)
    return plan

```

```
"""permet de faire tourner le programme jusqu'à ce que toutes les particules sortent"""
def evacuation(n,liste_sortie):
    L = creer_particules(n)
    lancement_evacuation(L,liste_sortie)
    while len(L)!= 0:
        lancement_evacuation(L,liste_sortie)
        deplacement_type2(L,liste_sortie)
        print(L)
    print('Done')


##Fonctions auxiliaires
"""donne une liste claire des points du graph"""
def point_graph(obstacle,liste_sortie):
    L = []
    for objet in obstacle:
        for k in range(len(objet)-1):
            L.append(objet[k])
    for sortie in liste_sortie:
        L.append(sortie)
    return L

"""permet de calculer la distance entre deux points"""
def distance(point_1,point_2):
    x1,y1 = point_1
    x2,y2 = point_2
    distance = sqrt((x1-x2)**2 + (y1-y2)**2)
    return distance
"""taille matrice obstacle"""
def taille_obstacle(obstacle):
    taille = 0
    for objet in obstacle:
        taille += len(objet)-1
    return taille

"""calcule la distance d'un trajet"""
def trajet(chemin,particule,liste_sortie):
    d = 0

    sommets = point_graph(obstacle,liste_sortie)

    sommet = copy(sommets)

    x,y,vx,vy = particule
    sommet.append([x,y])

    for k in range(len(chemin)-1):
        A = sommet[chemin[k]]
        B = sommet[chemin[k+1]]
        d += distance(A,B)
    return d

```

```
"""egalite dans le cadre de l'approximation entre flottants"""
def egalite(x,y):
    if x - 0.00001 <= y <= x + 0.00001 + y :
        return True
    return False

"""permet de copier une liste"""
def copy(list):
    L = []
    for k in list:
        L.append(k)
    return L

"""detecte une collision entre deux points"""
def collision(point1,point2):
    x1,y1 = point1
    x2,y2 = point2
    if (x2-x1)**2 + (y2-y1)**2 <= 4*rayon**2:
        return True
    return False

"""evalu si un point est contenu dans un obstacle"""
def inobstacle(particule):
    x,y,_,_ = particule
    intersec = 0
    for objet in obstacle:
        if len(objet)<2:
            continue
        count = 0
        for k in range(len(objet)-1):
526         for k in range(len(objet)-1):

528             xob1,yob1 = objet[k]
529             xob2,yob2 = objet[k+1]
530             if xob2 != xob1:
531                     a2 = (yob2 - yob1)/(xob2-xob1)
532                     b2 = yob1 - a2*xob1
533             if xob2 == xob1:
534
535                 if (min(yob1,yob2) <= y <= max(yob1,yob2)) and (x<=xob1):
536                     count += 1
537                     intersec += 1
538
539                     if yob1 == y or yob2 == y:
540                         count -= 0.5
541                         intersec -= 0.5
542                     continue
543                 else:
544                     continue
545             if a2 != 0:
546                 x_intersec = (y-b2)/a2
```

```
            if a2 != 0:
                x_intersec = (y-b2)/a2
                if (min(yob1,yob2) <= y <= max(yob1,yob2)) and x <= max(xob1,xob2) and x_intersec >= x:
                    intersec += 1
                    count += 1

                    if yob1 == y or yob2 == y:
                        intersec -= 0.5
                        count -= 0.5
        if int(count)%2 == 1 and (yob1 == y or yob2 == y):
            intersec -=1
    intersec = int(intersec)

    if intersec%2==0:
        return False
    return True

"""algorithme de dijkstra"""
def dijkstra(M,s):
    infini = 0

    #on choisit pour l'infini la somme + 1, de la longueur des arcs sur graph
    for ligne in M:
        for k in ligne:
            infini += k

    nb_sommets = len(M)
    s_connu = {s : [0,[s]]}
    s_inconnu = {k : [infini,''] for k in range(nb_sommets) if k != s}

    for suivant in range(nb_sommets):
        if M[s][suivant]:
            s_inconnu[suivant] = [M[s][suivant],s]

    while s_inconnu and any(s_inconnu[k][0] < infini for k in s_inconnu):
        u = min(s_inconnu, key = s_inconnu.get) #on choisit parmis les points le chemin pour lequel la distance est la plus petite

        longueur_u, precedent_u = s_inconnu[u]

        for v in range(nb_sommets):
            if M[u][v] and v in s_inconnu:
                d = longueur_u + M[u][v]
                if d < s_inconnu[v][0]:
                    s_inconnu[v] = [d,u]

        s_connu[u] = [ longueur_u, s_connu[precedent_u][1] + [u]]
        del s_inconnu[u] #on supprime u du dictionnaire des sommets inconnus

    return s_connu
```

```
"""trouve le plus court chemin"""
def pluscourtchemin(M,entree,sortie):
    s_connu = dijkstra(M,entree)
    for k in s_connu:
        if sortie == k:
            return s_connu[k][1]

"""tri par incertion"""
def triparincertion2(L,M):
    n = len(L)
    for i in range(n):
        #mémoriser L[i] dans memo
        memo = L[i]
        memo2 = M[i]
        #décaler vers la droite les éléments de L[0] à L[i-1] qui sont plus grands que memo en partant de L[i-1]
        j = i
        while j > 0 and L[j-1]>memo:
            L[j] = L[j-1]
            M[j] = M[j-1]
            j = j - 1
        #on place x dans le "trou" laissé par le décalage.
        L[j] = memo
        M[j] = memo2

    return L,M


##Evaluation d'une sortie optimisee
"""compte le nombre de frame necessaire pour une evacuation"""
def temps_sortie(n,liste_sortie):
    L = creer_particules(n)
    plan = plan_evacuation(L,liste_sortie)
    lancement_evacuation(L,liste_sortie)
    t = 0
    while len(L) != 0:
        test2(L,plan,liste_sortie,Fa)
        print(len(L))
        t += 1
    return t

"""compte le nombre de frame moyen necessaire pour une une evacuation"""
def temps_sortie_moyen(n,liste_sortie):
    s = 0
    for k in range(3):
        s += temps_sortie(n,liste_sortie)
    return s/3
```

```
641
642 """cree une liste de sorties placees de facon regulieres autour de la salle, nb_sorties correspondant au nombre de sortie par cotes"""
643 def sortie_successives(nb_sortie):
644     L = []
645     h = hauteur/(nb_sortie + 1)
646     l = largeur/(nb_sortie + 1)
647
648     hs = h
649     ls = l
650     for k in range(nb_sortie):
651
652         L.append([ls,0])
653         L.append([ls,hauteur])
654         L.append([0,hs])
655         L.append([largeur,hs])
656
657         hs += h
658         ls += l
659     return L
660
661 """permet de calculer les temps d'evacuation moyens de chaque sorties"""
662 def meilleure_sortie(n,nb_sortie):
663     sorties = sortie_successives(nb_sortie)
664     temps = []
665
666     for k in sorties:
667         liste_sortie = [k]
668         temps.append(temps_sortie_moyen(n,liste_sortie))
669     return temps
670
671
672
673 ##Constantes dependantes des fonctions
674 sommets = point_graph(obstacle,liste_sortie)
675 t_obstacle = taille_obstacle(obstacle)
676 M = matrice_obstacle(obstacle,liste_sortie)
677
```