Filière MP - ENS de Paris-Saclay, Lyon, Rennes et Paris - Session 2019

Page de garde du rapport de TIPE

NOM : BENNOUNA	Prénoms : Omar
Classe : MP*4	
Lycée : Louis-le-Grand	Numéro de candidat : 20360
Ville : Paris	

Concours auxquels vous êtes admissible, dans la banque MP inter-ENS (les indiquer par une croix) :

ENS Cachan	MP - Option MP	✓	MP - Option MPI	
	Informatique			
ENS Lyon	MP - Option MP	✓	MP - Option MPI	
	Informatique - Option M		Informatique - Option P	
ENS Rennes	MP - Option MP	✓	MP - Option MPI	
	Informatique			
ENS Paris	MP - Option MP		MP - Option MPI	
	Informatique			

Matière dominante du TIPE (la sélectionner d'une croix inscrite dans la case correspondante) :

Informatique		Mathématiques	✓	Physique	

Titre du TIPE : Modélisation de mouvements de groupes

Nombre de pages (à indiquer dans les cases ci-dessous) :

Texte	4	Illustration	6	Bibliographie	1/2

Résumé ou descriptif succinct du TIPE (6 lignes, maximum) :

Ce travail a pour but de créer un algorithme réalisant une gestion efficace de mouvements d'un ensemble d'unités. Pour cela, on commence par un modèle initial simple puis nous l'améliorons progressivement en s'inspirant des mouvements observés dans la nature.

À Paris

Le 8/06/2019

Signature du professeur responsable de la classe préparatoire dans la discipline

Cachet de l'établissement

Lycée Louis le Grand
075 0655 E
123, rue St Jacques
75231 PARIS CEDEX 05
Tél. 01 44 32 82 00

Signature du (de la) candidat(e)

La signature du professeur responsable et le tampon de l'établissement ne sont pas indispensables pour les candidats libres (hors CPGE).

Modélisation de mouvements de groupe

TIPE mathématiques appliquées et informatique pratique

Omar Bennouna

11 juin 2019

Le but de ce tipe est de pouvoir modéliser un mouvement de groupe semblable à ceux observé dans la nature (bancs de poissons, flottes d'oiseaux, etc.). Ce comportement peut être résumé en trois lois fondamentales proposées par Craig Reynolds [1] que doit respecter chaque agent du troupeau

1 **Séparation :** Garder une distance convenable des autres agents pour éviter les collisions et la surcharge
2 **Alignement :** Se diriger vers la direction moyenne du troupeau
3 **Cohésion :** Rester proche du troupeau pour éviter la dispersion du troupeau

Au cours de la réalisation du TIPE, nous nous sommes inspirés de plusieurs articles tels que [3] [2] [4] [1].

I Algorithme naïf

I.1 2 agents, force en $\frac{1}{r}$ et force en r

Dans tout ce qui suit, on va considérer que la masse de tous les agents considérés est égale à 1. On utilise un modèle discret avec des valeurs entières du temps, la position sera donc mise à jour avec $q_{n+1} = q_n + V_n$ et la vitesse $V_{n+1} = V_n + a_n$ avec respectivement V_n et a_n la vitesse et l'accélération de l'agent. Les positions et les vitesses seront stockés dans l'array créé via la commande : `pos=np.zeros((n,2))`.

On commence par essayer de respecter la première et troisième loi, pour cela on commence par faire une simulation avec une force proportionnelle à l'inverse de la distance entre les deux agents et une autre à la distance entre eux, caractérisés respectivement par deux constantes λ et μ. La simulation a donné le résultat suivant :

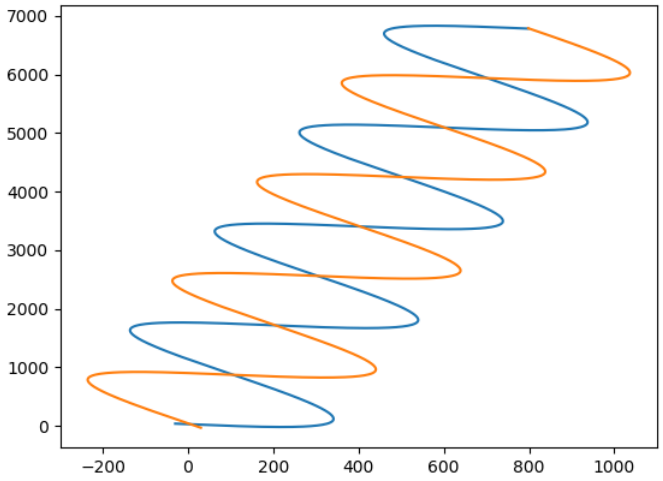

FIGURE 1 – Simulation algorithme naïf
$\lambda = 5 \times 10^{-4}$ $\mu = 4.5 \times 10^{-4}$

1. Voir [3]

I.2 Amélioration

On observe que les lois 1 et 3 sont à peu près respectées, mais la loi 3 ne l'est pas à cause des oscillations venant des forces attractives et répulsives. Pour remédier à ce problème, on fait tendre les vitesses des deux agents vers une vitesse finale $\overrightarrow{V_{lim}}$. Pour cela, on définit un coefficient de frein f et une direction finale $\overrightarrow{v_{inf}} = \frac{\overrightarrow{V_{lim}}}{\left\|\overrightarrow{V_{lim}}\right\|}$ et à chaque itération on fait :

— Si $\left\|\vec{V}\right\| \geq \left\|\overrightarrow{V_{lim}}\right\|$ Alors $\vec{V} = \vec{V} \times f + Forceattractive + Forcerpulsive$

— Sinon $\vec{V} = \vec{V} \times f + Forceattractive + Forcerpulsive + (\left\|\overrightarrow{V_{lim}}\right\| - \left\|\vec{V}\right\|) \times \overrightarrow{v_{inf}}$

En utilisant ceci, on simule une seconde fois et on obtient :

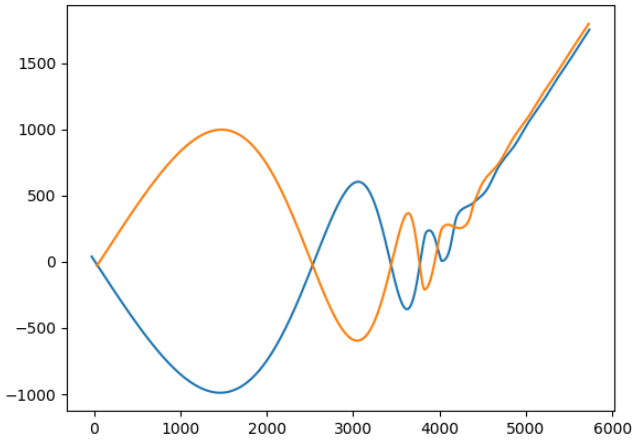

FIGURE 2 – Simulation algorithme naïf modifié
$\lambda = 5 \times 10^{-4}$ $\mu = 1$, $f = 9.9 \times 10^{-1}$

On observe que l'algorithme fait bien converger les vitesses et que les 3 lois sont respectées. On peut aisément montrer que la distance finale entre les 2 agents est $\sqrt{\frac{\mu}{\lambda}}$.

I.3 Algorithme pour N agents

On applique cette fois l'algorithme pour N agents avec N un entier. On stocke les N tableaux de positions et de vitesse en utilisant une boucle for (pour éviter les problèmes de listes clonées). En faisant un premier test pour 10 agents et pour les mêmes valeurs de λ et μ qu'avant, on obtient un comportement similaire, même pour des vitesses initiales très grandes [2] :

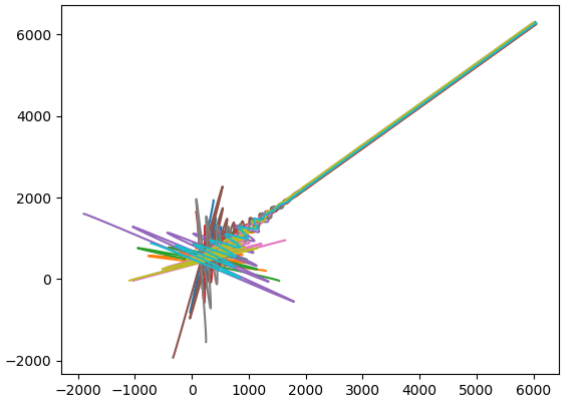

FIGURE 3 – Simulation algorithme naïf pour 10 agents
$\lambda = 5 \times 10^{-4}$ $\mu = 1$

2. pour V_x et V_y initiaux pris entre -2000 et 2000 pour la figure 3, et -100 et 100 pour les figures 4 et 5

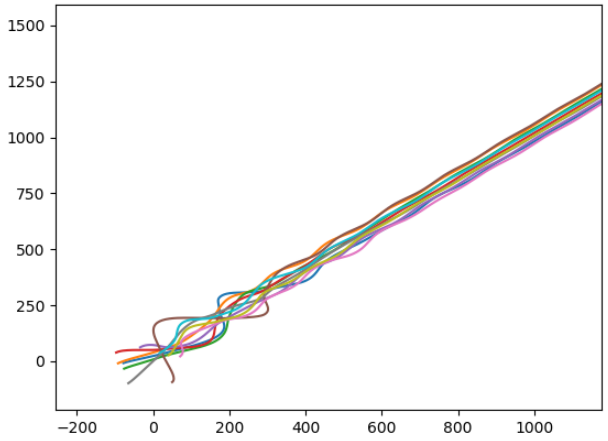

FIGURE 4 – Simulation algorithme naïf pour 10 agents
$\lambda = 5 \times 10^{-4}$ $\mu = 1$

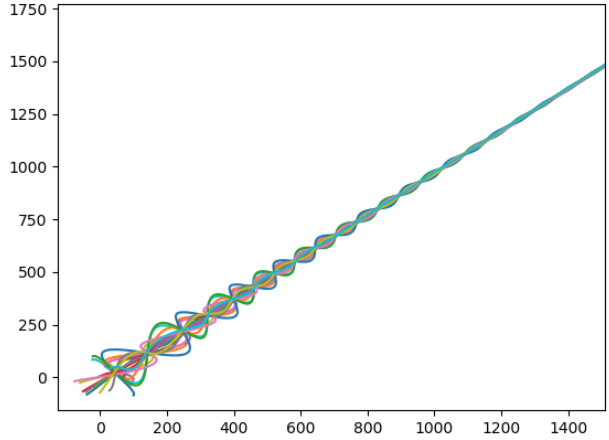

FIGURE 5 – Simulation algorithme naïf pour 20 agents
$\lambda = 5 \times 10^{-4}$ $\mu = 5 \times 10^{-3}$

II Modèle plus réaliste

On a considéré jusque là que la portée des agents étaient infinie, car ils pouvaient interagir entre eux peu importe la distance inter-agents. On va maintenant modifier le modèle pour passer à quelque chose de plus réaliste.

II.1 Graphes de proximité

On Commence par définir un graphe $G(t) = (V, E)$ et $r > 0$ qu'on appellera portée, tel que les sommets $V = v_{i\in\{1,..,k\}}$ soient les coordonnées des agents à l'instant t avec k le nombre d'agents et E représenté par la matrice d'adjacence $[a_{i,j}]_{i,j\in V}$, telle que $\forall (i,j) \in V$ $a_{i,j} = 1$ si $\|v_i - v_j\| \leq r$ et $a_{i,j} = 0$ sinon. on note dans toute la suite N_i l'ensemble des voisins de v_i.
Un agent n'interagira alors que avec l'ensemble de ses voisins N_i.

II.2 Tests sur le modèle naïf

On commence par tester sur le modèle naïf, et on compare avec les résultats précédents :

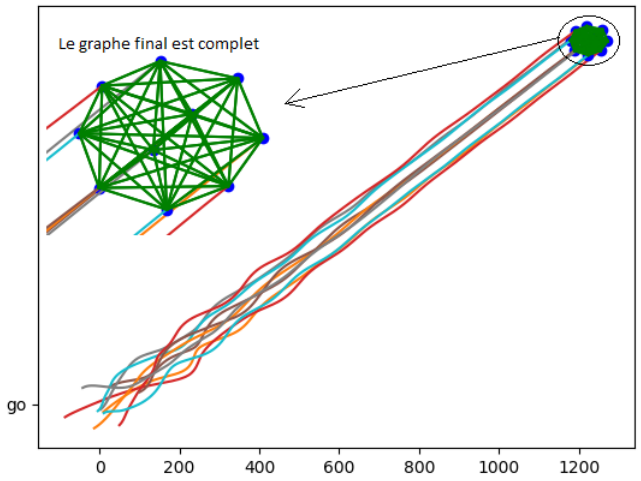

FIGURE 6 – Test algorithme naïf pour 10
$\lambda = 5 \times 10^{-4}$ $\mu = 1$ $r = +\infty$

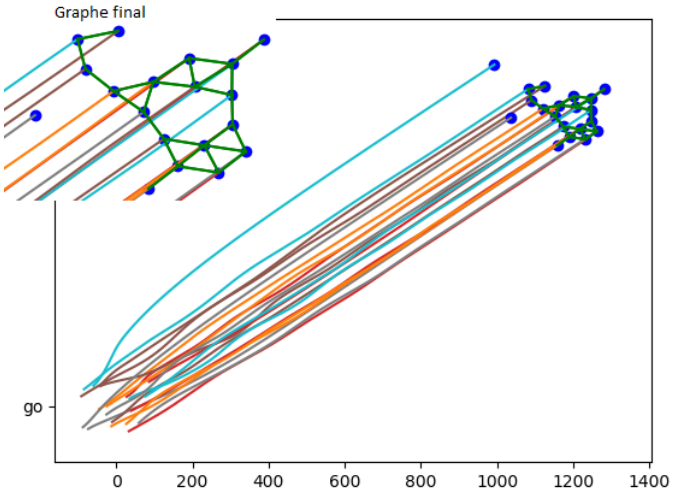

FIGURE 7 – Test algorithme naïf pour 20 agents
$\lambda = 5 \times 10^{-4}$ $\mu = 1$ $r = 100$

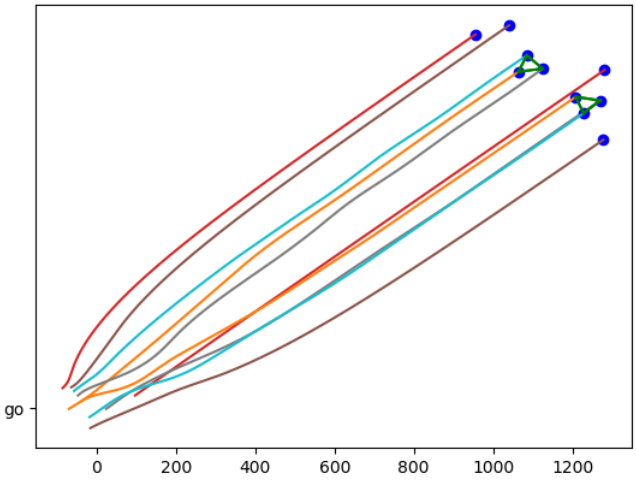

FIGURE 8 – Test algorithme naïf pour 10 agents
$\lambda = 5 \times 10^{-4}$ $\mu = 1$ $r = 70$

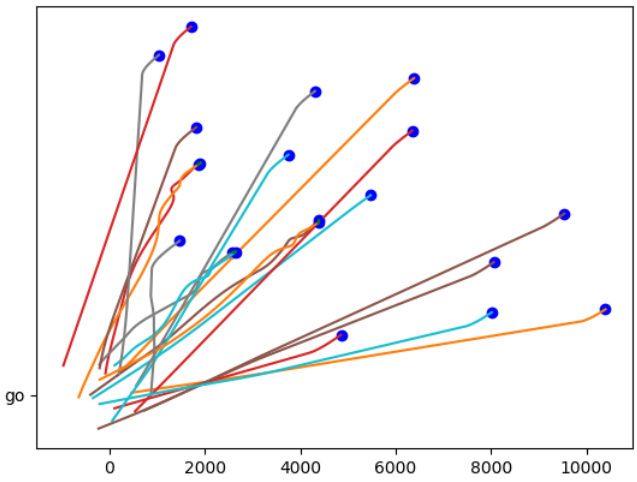

FIGURE 9 – Test algorithme naïf pour 20 agents
λ=0.0005 μ=1 $r = 500$

II.3 Exploitation des résultats

On remarque que le premier modèle ne permet pas, sauf coïncidence ; de réaliser le comportement voulu, même pour r raisonnablement grand : les agents se regroupent mais en petits troupeaux isolés les uns des autres. Il suffit de prendre des positions initiales assez éloignées pour complètement perturber le fonctionnement de l'algorithme [3] : les agents convergent indépendamment vers la vitesse limite dans interagir entre eux. Cela est dû au fait que si les agents ne sont pas assez proches initialement, il n'y a aucune interaction entre eux et donc aucune raison de se regrouper. on va donc opter pour un algorithme plus efficace.

II.4 Changement de stratégie

On va changer de moyen pour faire converger les trajectoires vers une disposition convenable : aulieu de faire converger les vitesses, on va introduire un agent-γ virtuel, ou agent leader qui sera suivi par les autres agents, c'est à dire une position but (éventuellement variable) vers laquelle se dirigeront tous les agents à tout instant.

II.4.1 Les fonctions à utiliser

Pour que le modèle soit réaliste, on va opter pour des fonctions lisses partout :

σ-norme La norme euclidienne usuelle n'est pas différentiable en 0, pour éviter de vitesses tendant vers l'infini au voisinage de 0, on va la remplacer par une σ-norme $\|.\|_\sigma$ (pas une norme) différentielle partout telle que pour tout vecteur x,
$\|x\|_\sigma = \frac{\sqrt{1+\epsilon\|x\|^2}-1}{\epsilon}$ avec $\epsilon > 0$.

"Bump function" [4] On va introduire une fonction lisse ρ_h, $h \in]0,1[$ (de classe C^1) qui varie entre 0 et 1 lorsque l'argument varie entre 0 et 1, l'introduction de cette fonction sera justifié par ce qui suivra.
On définit la fonction par :

$$\rho_h(x) = \begin{cases} 1 & \text{si } x \in [0,h[\\ \frac{1}{2}(1+cos(\pi(\frac{x-h}{1-h})) & \text{si } x \in [h,1] \\ 0 & \text{sinon} \end{cases}$$

3. Pour le test de la figure 9, x et y ont été pris assez variés : choix aléatoire entre -1000 et 1000 ce qui a causé la fragmentations des troupeaux d'agents.
4. Fonction inspirée de [2]

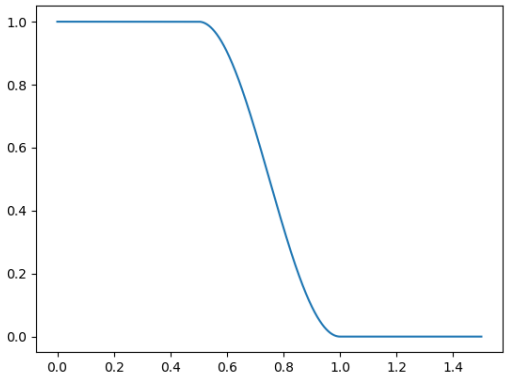

FIGURE 10 – Fonction ρ_h pour $h = \frac{1}{2}$

Graphes de proximité V2.0 On modifie la matrice d'adjacence : les valeurs de la matrice varieront continuement entre 0 et 1 pour modéliser l'effet de la distance sur l'interaction entre agents, et pour que les fonctions modélisant les interactions soient lisses. On pose $r_\sigma = \|r\|_\sigma$
On a alors $\forall(i,j)\ a_{i,j} = \rho_h(\frac{\|q_i - q_j\|_\sigma}{r_\sigma})$. On a bien $a_{i,j} \neq 0 \Leftrightarrow \|q_i - q_j\| < r$.

II.4.2 La stratégie

On pose pour tout i q_i le vecteur position de l'agent i et $u_i = \ddot{q}_i$.
On pose alors $u_i = f_g(q_i) + f_d(q_i) + f_\gamma(q_i)$. f_g sera la force qui permetra le regrouppement des agents, f_d sera une force d'amortissement pour éliminer tout oscillation parasite au cours du mouvement et f_γ sera la force qui imposera aux agents de se diriger vers le γ-agent.

III Changement de modèle

III.1 Critique du modèle précédent

Le modèle naïf a plusieurs inconvégnents :
— Fonctions discontinues ou non différentiables en certains points particuliers
— Non réaliste : Convergence forcée (V_{inf}), porté infinie etc.
— Modèle plutôt adapté aux oiseaux voyageurs.
— N'est pas cohérent avec ce qui est observé en général
On va donc se réorienter vers un modèle plus efficace et le perfectionner.

III.2 Nouveau modèle

La matrice d'adjacence ne contient plus que des 1 et des 0, les coefficients de la matrices seront donnés par $a_{i,j} = C\rho_h(\frac{\|q_i - q_j\|_\sigma}{r_\sigma})$ comme précisé auparavant, avec C une fonction qui fait que la matrice soit bistochastique.

On va laisser tomber la convergence forcée des vitesses, et on va ajouter une loi d'interaction entre les agents qui aura pour but d'illustrer ce qui se passe dans la nature comme pour les bancs de poissons et les oiseaux. La loi est définie comme ce qui suit : Pour un agent i, la vitesse à l'instant t+1 est donnée par $V_i(t+1) = V_i(t) + \sum_{j \in N_i}(V_i(t) - V_j(t)) \times a_{ij} + F$ avec a_{ij} le coefficient de position ij dans la matrice d'adjacence du graphe de proximité défini auparavant et F le terme des forces attractives et répulsives permettant de garder une distance convenable des autres agents, l'ajout de ce terme permet d'homogénéiser les vitesses sans avoir à forcer leur convergence.
On commence par faire des tests avec une portée infinie. Dans tout la suite on désigne par d_p et d_v deux réels positifs tels que les positions initiales et vitesses initiales soient respectivement dans $[-d_p, d_p]$ et $[-d_v, d_v]$. Pour des distances et portées raisonnables, on obtient de petits troupeaux isolés les uns des autres. Dans la figure qui suit, on a $d_p = 150$ $d_v = 10$ $h = 1/2$ $\epsilon = \frac{1}{10}$.

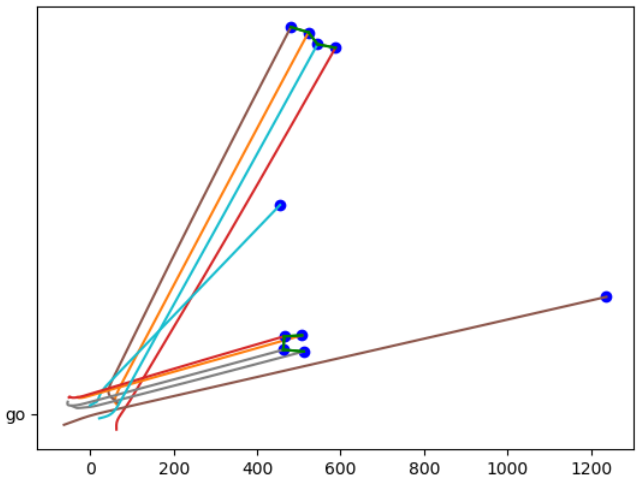

FIGURE 11 – Test algorithme réaliste pour 10 agents
$\lambda = 5 \times 10^{-4}$ $\mu = 1$ $r = 50$

III.3 Ajout d'une cible

Pour assurer la formation d'un seul troupeau, nous allons maintenant ajouter une position cible vers laquelle tous les agents vont se diriger.

III.3.1 Premier modèle et premiers tests

On va commencer par définir une loi d'attraction simple : pour tout unité de position q_i et c le vecteur position du but, la force exercée sur l'unité est donnée par $\gamma(c - q_i)$, avec gamma une constante définie au préalable.

Les premiers tests donnent des résultats très insatisfaisants.

Test1 : but avec vitesses rectiligne uniforme La courbe en pointillés représente la trajectoire du but.

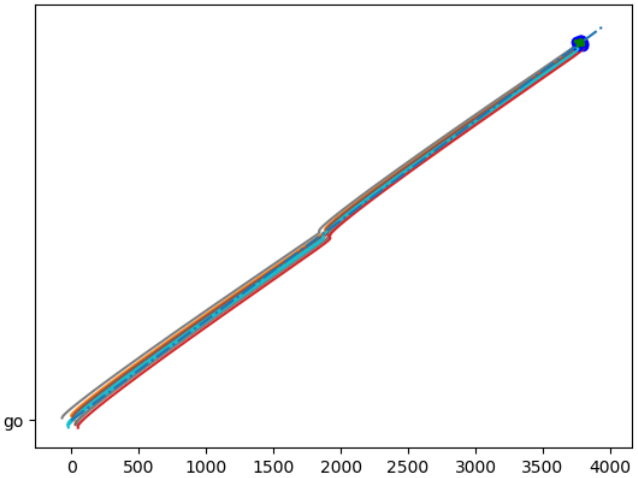

FIGURE 12 – Test ajout de but pour 10 agents

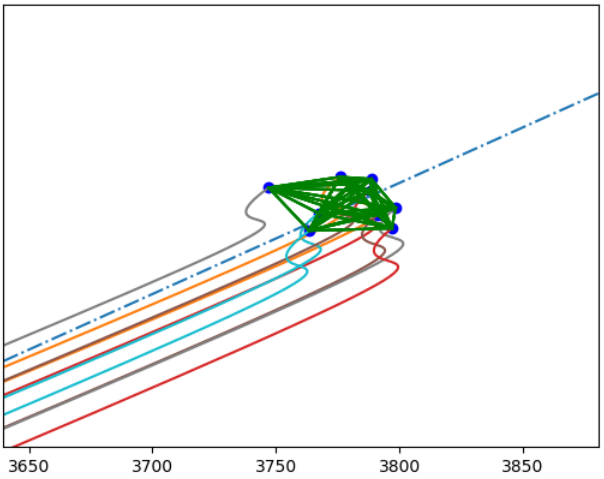

FIGURE 13 – Zoom sur la figure précédente

On remarque que la vitesses des agents croit jusqu'à ce qu'ils atteignent le but, ce qui les fait dévier et ralentir, ce qui n'est pas voulu

Test2 : but avec vitesse circulaire de norme constante Ici encore, la courbe en pointillés représente la trajectoire du but.

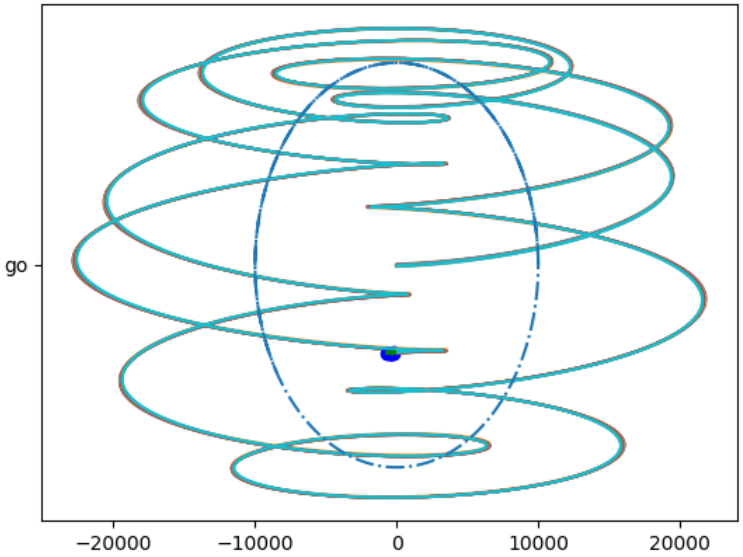

FIGURE 14 – Test ajout de but pour 10 agents

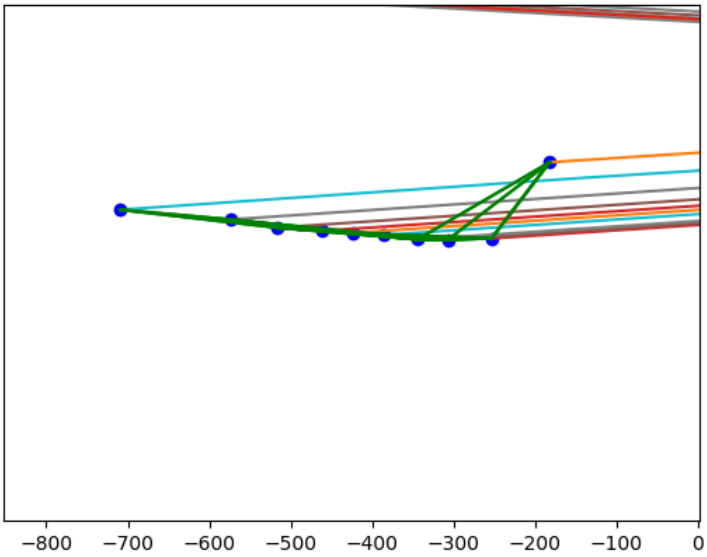

FIGURE 15 – Zoom sur la figure précédente

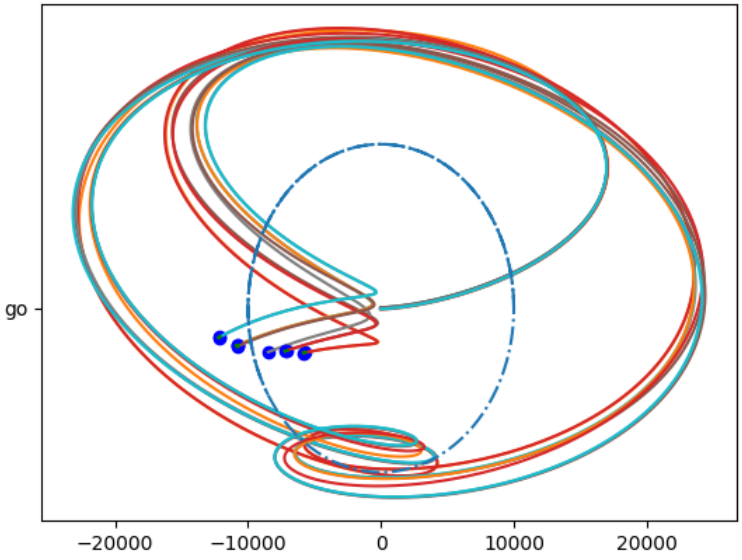

FIGURE 16 – Autre test avec paramètres similaires

Les agents n'arrivent pas à stabiliser leur trajectoire pour qu'elle soit identique à celle du but.

III.3.2 Remarques et améliorations

Pour pouvoir avoir une vitesse identique à celle du but en régime permanent, on va ajouter une force attractive dans l'espace des vitesses. Cette force sera sous la forme $F_{StabVitesse} = -k \times \|V_{but} - V_{agent}\|^{\beta} \times (V_{but} - V_{agent})$ avec, $\vec{v}$ la vitesse de l'agent divisée par sa norme et β un réel positif.
Les résultats obtenus sont relativement satisfaisants : les lois 1, 2 et 3 sont respectées.

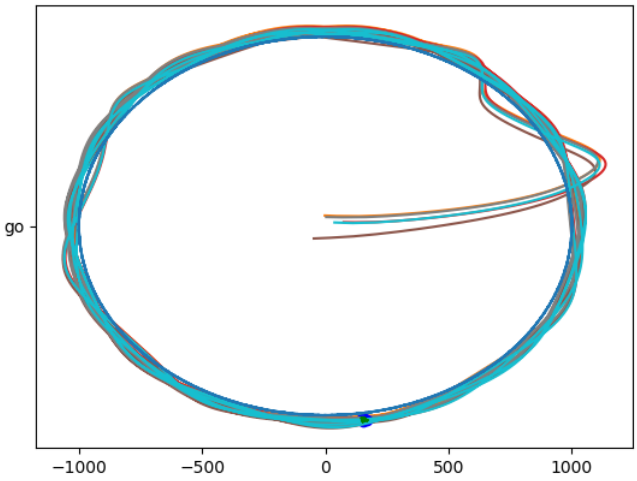

FIGURE 17 – Simulation avec nouvelle règle
$\beta = 1.5$

IV Obstacles

On modélise les obstacles par des disques exerçant sur les agents une force répulsive en $\frac{1}{d}$ avec d la distance entre l'agent et la frontière du disque.
La force sera donc de la forme $F_{obs} = -\omega \times \frac{q_{obs2}-q_i}{\|q_{obs2}-q_i\|^2} \times 1_{\|q_{obs}-q_i\|<\delta}$ avec $\omega > 0$, $q_{obs2} = q_{obs} - \frac{q_{obs2}-q_i}{\|q_{obs2}-q_i\|} \times r_{obs}$, δ une distance arbitrairement choisie pour ignorer l'obstacle à grande distance et r_{obs} le rayon de l'obstacle.
Les résultats obtenus sont satisfaisants, les obstacles perturbent raisonnablement le mouvement.

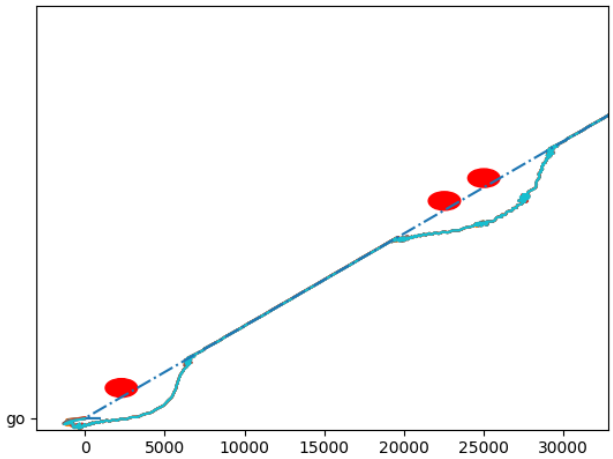

FIGURE 18 – Simulation avec vitesse rectiligne uniforme

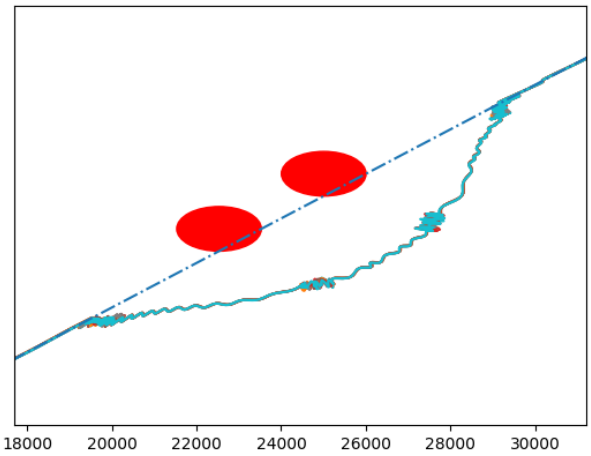

FIGURE 19 – Zoom sur la figure précédente

Références

[1] Jamila Sam et FRIEDEMANN ZENKE. *Simulation de Boids Modèle et Compléments Mathématiques.* URL : icwww.epfl.ch/~sam/prjinfoSV/projet/latex/projet-infosv3-1011.pdf.

[2] Reza OLFATI-SABERS. *IEEE TRANSACTIONS ON AUTOMATIC CONTROL, Flocking for Multi-Agent Dynamic Systems : Algorithms and Theory : VOL. 51, NO. 3, MARCH 2006, 401-420.* URL : http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.121.7027&rep=rep1&type=pdf.

[3] Craig REYNOLDS. *Boïds.* URL : https://www.red3d.com/cwr/boids/.

[4] LETUVEE Nicolas et ROHMER DAMIEN. *Modélisation du mouvement des poissons par des Boïds.* URL : http://imagecomputing.net/damien.rohmer/publications/2002_2006_student_work/2004_boids/ra%20pport/boids_3eti_rohmer.pdf.

V Annexes

V.1 Code Python pour les simulations

Ci-dessous le code Python permettant d'effectuer les simulations.
Nous avons amélioré progressivements le code, il y a donc plusieurs versions. Nous présentons ici le code permettant d'obtenir les résultats de la figure 17.
Il y a quelques fonctions et variables inutiles à cause de quelques essais d'implémentation échoués faute de temps.

```
#Modules a importer
import numpy as np
import matplotlib.pyplot as plt
from math import sqrt,cos,pi,sin
import scipy.integrate as integrate
import scipy.special as special
from numpy.linalg import norm as norme,norm
from random import randint,random
#Liste des matrices d'adjacences a chaque instant

tabG=[]

#constantes
```

```
lamda=0.0005
mu=1
frein=np.array([0.99]) #coef de freinage
acc=1.01 #coef d'acceleration
a=0.5
b=1
c=abs(a-b)/(2*sqrt(a*b)) # a b c constantes pour la fonction
    phi
r=100 #rayon maximal d'interaction
boolinteract=1 #Activer ou desactiver l'interaction entre
    agents
d=5 #distance but
hh=1/2 #parametre de la fonction rho
eps=1/10
rg=150 #dispersion des positions initiales
vd=10 # dispesion des vitesses initiales
GoalCoeff=0.0005 #coef d'attraction vers le but
StabVitesseCoeff=0.0005 #Coef utilise dans la fonction
    StabVitesse
gamma=10
beta=2

#Fonctions

#forces

def StabVitesse(vbut,vagent):
    return(StabVitesseCoeff*norm(vbut-vagent)**beta*(vbut-
        vagent))
def Goal(x,g):
    return(norm(g-x)*(g-x)*[GoalCoeff/10])

def Fattr(P1,P2):
    return(lamda*(P2-P1))

def Frep(P1,P2):
    return((-mu/(norm(P2-P1)**2)*(P2-P1)))

def Fobs(P1,P2):
    return((gamma/(norm(P2-P1)**2)*(P2-P1)))

def sigmanorme(epsilon,z): #sigma norme
    return(1/epsilon*(sqrt(1+epsilon*(norme(z)**2))-1))

def sigmanormereel(epsilon,z): #sigma norme
    return(1/epsilon*(sqrt(1+epsilon*z**2))-1)

def lapsigmanorme(epsilon,z): # laplacien de la sigma norme
    return(np.array([1/(1+epsilon*sigmanorme(z))])*z)

def rho(h,z): #Bump function
    if z<h and z>=0:
        return(1)
    elif z>=h and z<=1:
        return((1+cos(pi*((z-h)/(1-h))))/2)
    else:
        return(0)
```

```
def sigma1(z): #fonction sigma1 utilisee pour phi
    return(z/sqrt(1+z**2))

def phi(z):
    return(1/2*((a+b)*sigma1(z+c)+(a-b)))

def phialpha(z):
    rho(hh,(z/sigmanorme(r)))*phi(z-sigmanorme(d))

def psialpha(z): #fonction psialpha de potentiel
    res=integrate.quad(lambda x: phialpha(x), sigmanorme(d), z)


#Nombre d'iterations , tableaux de coordonees et liste d'
   obstacles
nbobs=1 #nombre d'obstacles
obstacle=[] #liste d'obstacles
for i in range(nbobs):
    a=10*random()
    obstacle.append([1000*cos(a),1000*sin(a),30])

n=3000
k=5 #nombre d'agents

but=np.array([50,50])
pos=[] #Matrice de positions

for i in range(0,k):
    pos.append((np.zeros((n,2))).copy())  #boucle creation
       tableau position

for j in range(0,k):
    v1=(random()-1/2)*rg
    v2=(random()-1/2)*rg
    pos[j][0]=[v1,v2] #boucle positions initiales


V=[] #matrice de vitesses


for i in range(0,k):
    V.append((np.zeros((n,2))).copy()) #boucle creation tableau
        vitesses

for j in range(0,k):
    v1=(random()-1/2)*vd
    v2=(random()-1/2)*vd
    V[j][0]=[v1,v2] #boucle vitesses initiales

Vlim=[2,2]
Vinf=Vlim*np.array([1/norm(Vlim)])

#Implementation du graphe par matrice d'adjacence
G=np.zeros((k,k))
butab=[[1000,0],[1000,0]]
taby=[]
#Boucle de caclcul des positions successives
```

```
for i in range(0,n-1):
    for l in range(0,k):
        coefsum=0
        for j in range(0,k):
            coef=rho(hh,(sigmanorme(eps,pos[j][i]-pos[l][i]))/
               sigmanormereel(eps,r))
            coefsum+=coef
            G[j,l],G[l,j]=coef,coef #Calcul des coeficients de
               la matrice d'adjacence
        for j in range(0,k):
            G[j,l],G[l,j]=G[j,l]/coefsum,G[l,j]/coefsum

    tabG.append(G)

    but=[1000*cos(i/100),1000*sin(i/100)]
    butab.append(but)
    for j in range(0,k): #Calcul de la vitesse a l'instant i+1
        vit=V[j][i]
        #but=[100+10000*cos(i/10),100+10000*sin(i/10)]
        for l in range(0,k):
            if l!=j:
                vit+=(V[l][i]-V[j][i])*(G[j,l])+(Frep(pos[j][i
                   ],pos[l][i])+Fattr(pos[j][i],pos[l][i]))*G[j
                   ,l]
        vit+=Goal(pos[j][i],but)+StabVitesse((np.array(butab
           [-1])-np.array(butab[-2])),V[j][i])
        for t in range(0,len(obstacle)):
            aziz=norm([obstacle[t][0],obstacle[t][1]]-pos[j][i
               ])
            if aziz<3*obstacle[t][2]:
                if aziz<obstacle[t][2]:#force desecours au cas
                   ou l'agent traverse l'obstacle
                    vit+=-10*gamma*Frep([obstacle[t][0],
                       obstacle[t][1]],pos[j][i])
                else:
                    vit+=gamma*Fobs([obstacle[t][0],obstacle[t
                       ][1]],pos[j][i])
        pos[j][i+1]=pos[j][i]+vit
        V[j][i+1]=vit
taby=[norm(V[0][i]-(np.array(butab[i])-np.array(butab[i-1])))
   for i in range(n)]
taby=[norm(pos[0][i]-butab[i]) for i in range(n)]


#Trace de la courbe
H=[]
for j in range(0,k):
    x=[pos[j][i,0] for i in range(0,n)]
    y=[pos[j][i,1] for i in range(0,n)]
    plt.plot(x[n-1],y[n-1],'bo','go')
    H.append([x[n-1],y[n-1]])
    plt.plot(x,y)
for l in range(0,k):
    for m in range(0,k):
        if m!=l and G[m,l]==1:
            P1=[H[l][0],H[m][0]]
            P2=[H[l][1],H[m][1]]
            plt.plot(P1, P2, "green")
```

```
x=[butab[i][0] for i in range(len(butab))]
y=[butab[i][1] for i in range(len(butab))]
plt.plot(x,y,'-.')

for l in range(0,k):
    for m in range(0,k):
        if m!=l and G[m,l]!=0:
            P1=[H[l][0],H[m][0]]
            P2=[H[l][1],H[m][1]]
            plt.plot(P1, P2, "green")
C=[]
for i in range(0,nbobs):
    C.append(plt.Circle((obstacle[i][0],obstacle[i][1]),
       obstacle[i][2], color='r'))



fig = plt.gcf()
ax = fig.gca()


for i in range(nbobs):
    ax.add_artist(C[i])
plt.show()

##Trace espace des vitesses
for j in range(0,k):
    x=[V[j][i,0] for i in range(0,n)]
    y=[V[j][i,1] for i in range(0,n)]
    plt.plot(x[n-1],y[n-1],'bo','go')
    H.append([x[n-1],y[n-1]])
    plt.plot(x,y)
plt.show()
## Trace progressif
for h in range(0,n,20):
    for j in range(0,k): #Trace progressif de chaque
       trajectoire
        x=[pos[j][i,0] for i in range(h,h+20)]
        y=[pos[j][i,1] for i in range(h,h+20)]
        xbut=[butab[i][0] for i in range (h,h+20)]
        ybut=[butab[i][1] for i in range (h,h+20)]
        if h>=n-20:
            plt.plot(x[-1],y[-1],'bo','go')
            H.append([x[-1],y[-1]])
        plt.plot(x,y)
        plt.plot(xbut,ybut)
    plt.show()
    plt.pause(0.2)

for l in range(0,k):
    for m in range(0,k):
        if m!=l and G[m,l]!=0:
            P1=[H[l][0],H[m][0]]
            P2=[H[l][1],H[m][1]]
            plt.plot(P1, P2, "green")
#for j in range(0,k):
 #    for l in range(0,k):
  #       if G[j,l]==1:
```

```
   #          plt.plot(pos[j][n-1], pos[l][n-1], "green")
plt.show()
##Plot a l'instant t
def plot(debut,fin):
    H=[]
    for j in range(0,k):
        x=[pos[j][i,0] for i in range(debut,fin)]
        y=[pos[j][i,1] for i in range(debut,fin)]
        plt.plot(x,y)
    xbut=[butab[i][0] for i in range (debut,fin)]
    ybut=[butab[i][1] for i in range (debut,fin)]
    plt.plot(xbut,ybut)
    plt.show()

##plot cercles
import matplotlib.pyplot as plt
import random
obstacles=[] #liste d'obstacles
for i in range(10):
    obstacles.append([random.random()*1000,random.random()
       *1000,100])
C=[]
for i in range(10):
    C.append(plt.Circle((obstacles[i][0],obstacles[i][1]),
       obstacles[i][2], color='r'))



fig = plt.gcf()
ax = fig.gca()


for i in range(10):
    ax.add_artist(C[i])

plt.show()
```