Modélisations de vol d'oiseaux, avec les règles de Boids ou un système suivant des lois physiques

Physique Numérique

Louis MIROUZE, Matthieu PELISSIER

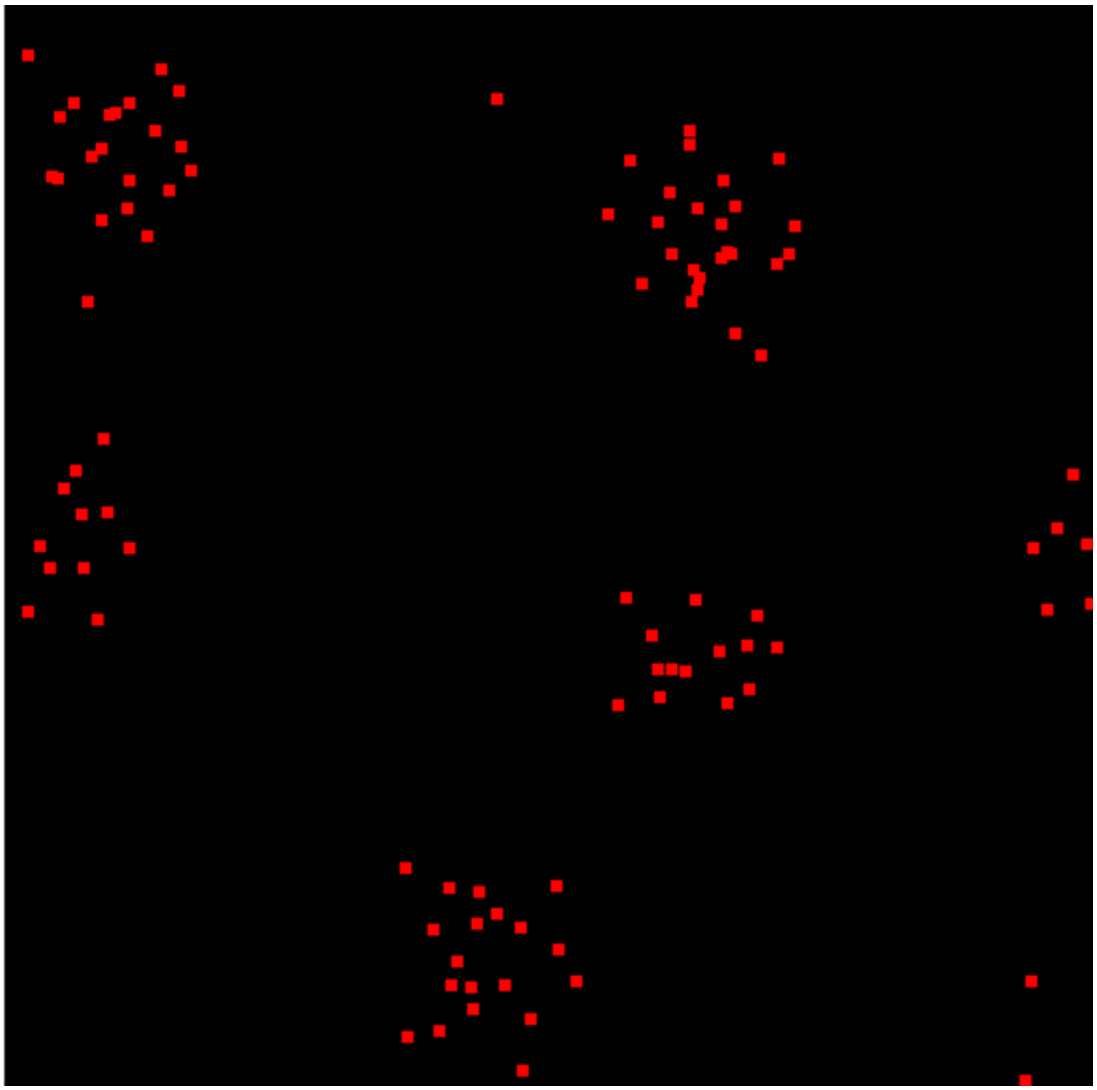


Table des matières

Introduction

La compréhension de comportements collectifs est un défi scientifique auquel s'attellent la biologie, la physique et les sciences sociales. Sa modélisation peut rapidement devenir compliquée de par le grand nombre de paramètres du système impliqué. Pour répondre à ce défi technique, Craig W Reynolds propose en 1987 trois règles simples pour simuler un vol d'oiseaux. Le programme de vie artificielle, nommé Boids, sera entre autre utilisé dans le cinéma pour simuler des foules.
Lors de ce projet, nous avons simulé des essaims d'oiseaux en 2D dans un ciel infini, en utilisant ces règles. Par la suite, nous avons modélisé un modèle suivant des lois physiques, en cherchant à s'approcher d'un comportement collectif d'oiseaux.

1 Simuler des essaims d'oiseaux avec des règles simples

Les trois règles proposées dans [Rey87] sont les suivantes :

1. Les boids cherchent à s'approcher dès qu'ils se voient.
2. Ils ne peuvent pas se retrouver au même endroit.
3. Ils cherchent à s'aligner les uns avec les autres.

Pour utiliser ces règles, nous avons utilisé la librairie Pygame pour l'interface graphique, ainsi que sa classe mère "pygame.sprite.Sprite". Sur ces points nous nous sommes inspiré de [Dow07]. Cette inspiration initiale nous fut utile pour trouver les outils graphiques, mais nous avons bien produit le code final. A partir de cela nous créons la classe oiseau, regroupant toutes les caractéristiques des objets simulés.

```
class Oiseau(pygame.sprite.Sprite):

    def __init__(self, x, y):

        super().__init__() #Appel init de la classe mère pygame.sprite.Sprite
        # et l'applique à Oiseau, donc self
        self.image = pygame.Surface([6, 6]) #crée surface de dimension [a,b]
        self.image.fill("red")  # Remplit la surface de couleur

        #positionne l'image aux coord (x,y)
        self.rect = self.image.get_rect(center=(x,y))

        #genere vitesse alétoire bornée par Max_velocity selon chaque axe
        self.velocityX = uniform(-MAX_VELOCITY, MAX_VELOCITY)
        self.velocityY = uniform(-MAX_VELOCITY, MAX_VELOCITY)

        # On définit une position pouvant prendre des valeurs décimales:
        self.positionX=x
        self.positionY=y
```

L'introduction de $self.positionX$ et $self.positionY$ est nécessaire pour ne pas perdre de précision dans les calculs à cause de Pygame. En effet, l'affichage s'effectue via un tableau quadrillé, où chaque carré est positionné par des coordonnées entières. Ainsi, "self.rect" arrondit toujours les positions à l'entier inférieur, causant donc des problèmes de conservation de grandeurs physiques. Ce qui serait problématique lors de la modélisation d'un système physique. Ainsi, "self.rect" ne sera utilisé que pour l'affichage, tandis que $self.positionX/Y$ le sera pour les calculs.
Nous introduisons également le paramètre $MAX\_VELOCITY$, qui norme les vitesses initiales selon chaque axe.
Nous devons également définir deux autres paramètres fondamentaux pour appliquer ces lois. $VISION$ est la distance maximale à laquelle un oiseau peut voir ses congénères, et donc la distance à partir de laquelle il peut attirer/être attiré. $DISTCRIT$ correspond à la distance à partir de laquelle les oiseaux se repoussent. Ces deux quantités interviennent dans les premières lois, qui dépendent de la distance entre les oiseaux. La distance est l'un des problèmes principaux de ce système, en particulier dans le cadre d'un ciel infini (ou bord périodique). Nous définissons au sein de la classe un moyen pour chaque oiseau de calculer sa distance avec ses congénères.

```
def distanceX(self, oiseau):
    #distance vers oiseau
    distX = oiseau.positionX - self.positionX
    #verifie si oiseau pas plus proche via le bord
    if abs(distX)>=SCREEN_WIDTH-2*BORDER-abs(distX):
        #si c'est le cas pointe vers à travers le bord
        return distX - (SCREEN_WIDTH-2*BORDER)*np.sign(distX)
    else: return distX
```

Cette fonction renvoie Δx : la composante du vecteur pointant de l'oiseau étudié (self) vers son congénère (oiseau), projeté selon $\vec{x}$. En combinant cette fonction avec son analogue selon $\vec{y}$, on obtient les coordonnées du vecteur. À partir de celles-ci, on obtient la distance algébrique séparant les oiseaux $D = \sqrt{(\Delta x)^2 + (\Delta y)^2}$.

```
def distance(self, oiseau):
    return sqrt(self.distanceX(oiseau)**2+self.distanceY(oiseau)**2)
```

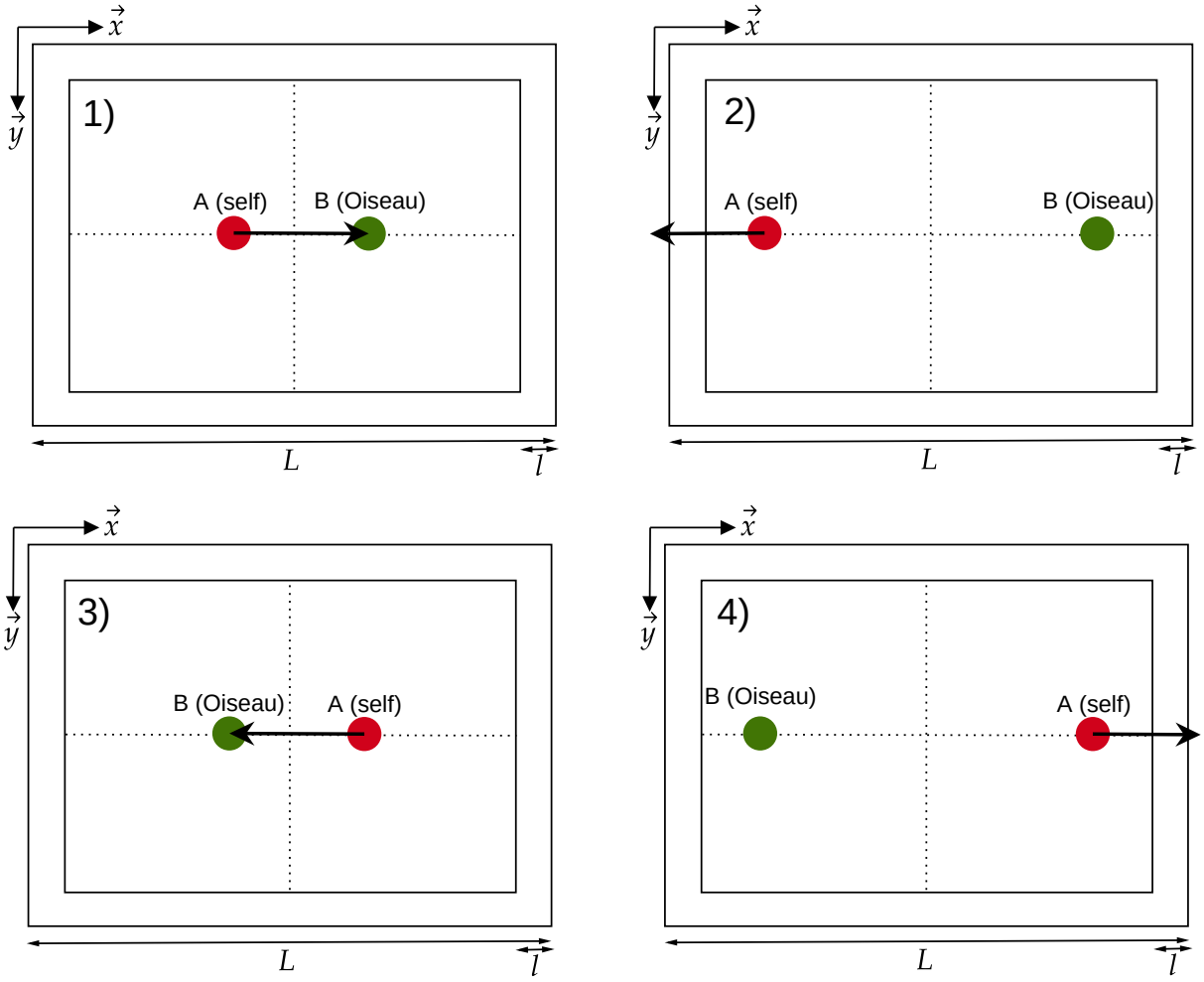


FIGURE 2 – Schéma explicatif du calcul des longueurs orientées pour différentes situations. Les cas illustrés correspondent à des oiseaux alignés sur une même coordonnée y. L est la longueur de l'écran, et l celle des bords.

Fig. 2 schématise comment calculer Δx, dans le cas particulier du ciel infini. On souhaite que le vecteur parte de l'oiseau étudie (self) et pointe vers l'oiseau avec le lequel il calcule sa distance. Il faut également que la distance les séparant soit minimale.

Situation 1) Les oiseaux sont proches du centre, dans ce cas-là $\Delta x = x_b - x_a > 0$

Situation 2) On a toujours $x_b - x_a > 0$. Mais $|x_b - x_a| > (L - 2l) - |x_b - x_a|$, ce qui signifie la distance les séparant en passant par le centre est plus grande que celle en passant par les bords. Il faudra donc utiliser $-(x_b - x_a) < 0$ pour que A pointe dans la bonne direction. Pour obtenir la bonne longueur les séparant, il faut corriger par la taille du ciel visible, soit $\Delta x = -(x_b - x_a) + (L - 2l) < 0$.

Situation 3) Les oiseaux sont de nouveaux proches de centres, et $\Delta x = x_b - x_a < 0$. Donc le vecteur est bien de longueur minimale et pointe vers la bonne direction.

Situation 4) Les oiseaux sont proches des bords. Il faudra ici utiliser $-(x_b - x_a) > 0$ pour pointer dans la bonne direction. On obtient donc $\Delta x = -(x_b - x_a) - (L - 2l)$.
On peut résumer toutes les situations avec :

$$\Delta x = \begin{cases} x_b - x_a \text{ si } |x_b - x_a| < (L - 2l) - |x_b - x_a| \\ -(x_b - x_a) + sign(x_b - x_a) \times (L - 2l) \text{ sinon} \end{cases} \quad (1)$$

On obtient la même expression analogue selon y, en remplaçant L par le largueur de l'écran.

Pour appliquer les règles d'attraction et de répulsion des oiseaux, on définit la fonction "acceleration".

```
def acceleration(self, oiseau_list):
    ForceX = 0
```

```
        ForceY = 0
        for oiseau in oiseau_list:
            xdiff = self.distanceX(oiseau)
            ydiff = self.distanceY(oiseau)
            distance = sqrt(xdiff**2+ydiff**2)

            #stocke force exercée par chaque oiseau sur self
            ForceX -= -xdiff*(1-DISTCRIT/distance)
            ForceY -= -ydiff*(1-DISTCRIT/distance)

        #modification de la vitesse par la force
        self.velocityX += ForceX * FORCE
        self.velocityY += ForceY * FORCE
```

On considère que tous les oiseaux de la liste "oiseau_list", exercent une force $\vec{F} = (F_x, F_y)$ sur self. Celle-ci est attractive au delà de DISTCRIT, et répulsive pour des oiseaux plus proches. Sa forme sera justifiée dans Sec. 2. Comme toute force, cela revient à une modification du vecteur vitesse de l'oiseau étudié. On introduit le paramètre $FORCE$, permettant choisir l'ordre de grandeur de la force exercée et subie par les oiseaux.

On crée une fonction "move_with" permettant d'aligner les oiseaux les uns avec les autres (3ème règle).

```
    def move_with(self, oiseau_list):
        avgX = 0
        avgY = 0
        for oiseau in oiseau_list:
            avgX += oiseau.velocityX
            avgY += oiseau.velocityY

        # avgX est la moyenne des composantes des vitesses selon X des oiseaux
        # de oiseau_list. Idem selon Y pour avgY.
        avgX /= len(oiseau_list)
        avgY /= len(oiseau_list)

        #On modifie la vitesse de self pour l'aligner avec le groupe.
        self.velocityX += (avgX/40)
        self.velocityY += (avgY/40)
```

On ajoute à la vitesse de self la moyenne des vitesses des oiseaux de "oiseau_list". Ainsi, self aura tendance à se diriger dans la même direction que ces oiseaux. La division par 40 dans les dernières ligne est arbitraire, et permet simplement un rendu plus esthétique.
On crée une fonction "update" permettant à l'oiseau de se déplacer dans le ciel en fonction de sa vitesse et position.

```
    def update(self):
        #Necessaire pour eviter la divergence de la vitesse avec move_with.
        #On vérifie que la vitesse ne dépasse pas MAX_VELOCITY.
        if abs(self.velocityY) >= MAX_VELOCITY or abs(self.velocityX) >= MAX_VELOCITY:
             scaleFactor = MAX_VELOCITY / max(abs(self.velocityY), abs(self.velocityX))
             #on ajuste la composante trop grande à Max velocity,
             #et l'autre garde la proportion (pour garder direction).
             self.velocityY *= scaleFactor
             self.velocityX *= scaleFactor

        #On déplace l'oiseau selon sa vitesse.
        self.positionX += self.velocityX
        self.positionY += self.velocityY

        #Si l'oiseau dépasse un bord on modifie sa position de l'autre côté.
        if self.positionX < BORDER: #bords
            self.positionX += SCREEN_WIDTH - 2*BORDER
        if self.positionX > SCREEN_WIDTH - BORDER:
            self.positionX -= SCREEN_WIDTH - 2*BORDER
        if self.positionY < BORDER:
```

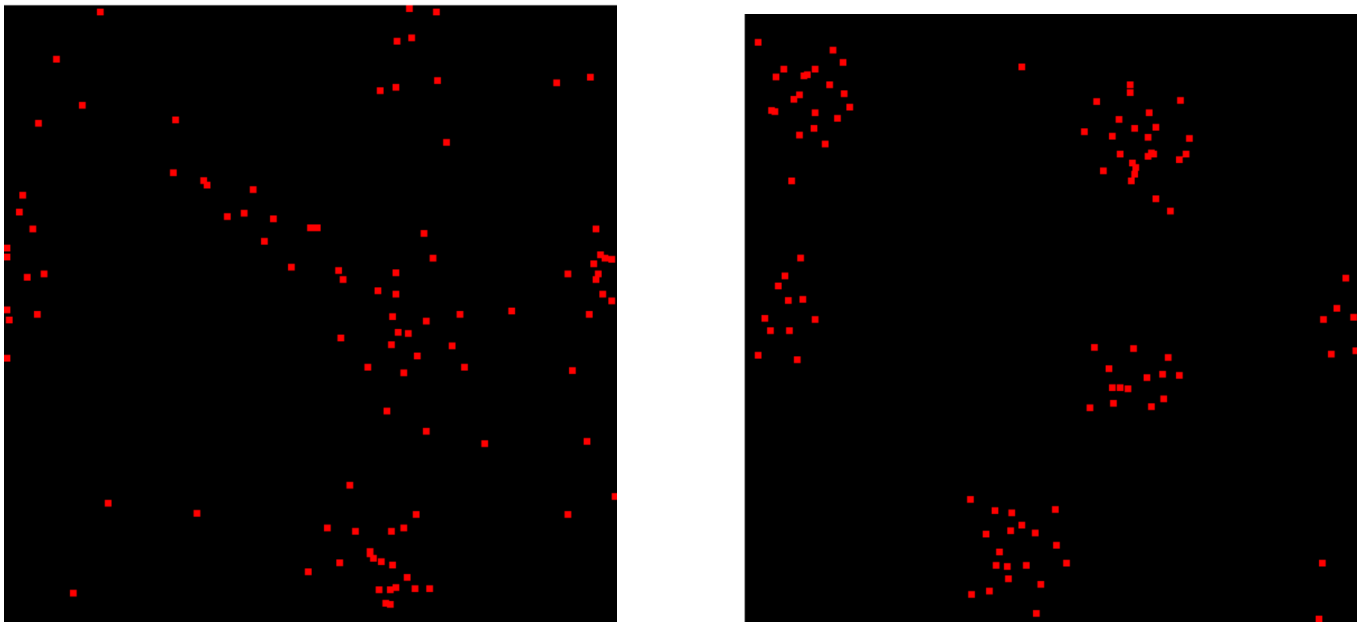
```
            self.positionY += SCREEN_HEIGHT -2* BORDER
        if self.positionY > SCREEN_HEIGHT - BORDER:
            self.positionY -= SCREEN_HEIGHT - 2*BORDER
        # On modifie la position de l'image de l'oiseau sur l'écran.
        self.rect.x = round(self.positionX)
        self.rect.y = round(self.positionY)
```

La fonction "move_with" n'est pas conservative et fait donc diverger la vitesse. Pour pallier à cela, les premières lignes de la fonction "update" limitent la norme de la vitesse à $MAX\_VELOCITY$, sans pour autant changer de direction. Une fois la vitesse ajustée, on peut modifier la position décimale de l'objet. Il se peut que ce déplacement fasse traverser les bords à l'oiseau. On vérifie que cela ne soit pas cas pour chacun des bords. Dans le cas contraire, on envoie l'oiseau de l'autre côté du ciel. C'est notamment sur cette partie du code que se base le ciel infini. On peut ensuite modifier la position de l'oiseau sur l'écran. On utilise "round" pour arrondir à l'entier le plus proche, évitant l'arrondi systématique de "self.rect" vers l'entier inférieur.
Toutes ces fonctions définissent la classe "oiseau". On peut ensuite générer un certain nombre "NUM_BOIDS" d'oiseaux à des positions aléatoires dans le ciel, et les stockons dans "oiseau_list". Après avoir initialisé Pygame, on effectue à chaque instant (i.e à chaque affichage d'une frame) la boucle suivante.

```
 for oiseau in oiseau_list:
    closeBoids=[]
    #Pour chaque oiseau on trouve ceux qu'il peut voir
    for otherBoid in oiseau_list:
        #il ne se voit pas lui même
        if otherBoid == oiseau: continue
        #Si les autres oiseaux sont plus près que VISION alors il les voit
        if oiseau.distance(otherBoid) < VISION:
            closeBoids.append(otherBoid)

    #Si l'oiseau ne voit personne alors il ne subit pas de force
    if len(closeBoids)>0:
        #Sinon on lui applique acceleration pour tous les oiseaux proches
        oiseau.acceleration(closeBoids)
        #On peut aussi lui appliquer move_with (système non physique)
        oiseau.move_with(closeBoids)
#Une fois toutes les vitesses des oiseaux modifiées, on update
# leurs positions simultanément.
oiseau_list .update()
```

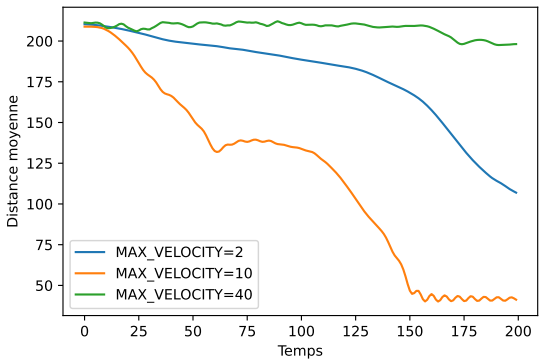
Pour chaque oiseau créé, on calcule sa distance avec tous ses congénères. Lorsque celle-ci est inférieure à $VISION$, cela signifie qu'ils interagissent entre eux. On stocke donc tous les oiseaux visibles dans "closeBoids". On peut donc appliquer les 3 lois présentées précédemment à l'oiseau et toute la liste. L'oiseau étudié va donc être attiré par les oiseaux présents dans la liste avec "acceleration" lorsque $D > DISTCRIT$, et être repoussé par ceux trop près, i.e $D < DISTCRIT$. Il cherchera également à aligner sa vitesse avec tous les oiseaux visibles via "move_with". Une fois la modification de vitesse calculée pour tous les oiseaux créés, on peut effectuer la fonction update simultanément pour chaque oiseau. Une manière plus esthétique pour simuler un vol d'oiseau, mais moins rigoureuse, serait se placer une ligne "oiseau.update()" dans la boucle. Les oiseaux seraient donc déplacés un à un. Cependant, cela signifierait que le déplacement du premier à être mis à jour influencerait les suivants. Au vu de l'objectif de modélisation d'un système physique, nous avons opté pour un "update" collectif et simultané.



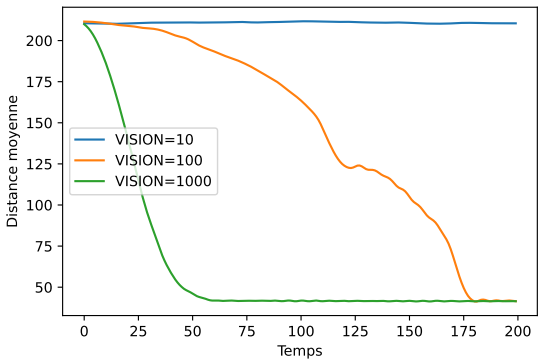
(a) t_1 (b) t_2

FIGURE 3 – Exemple de simulation obtenable en utilisant les règles de boids présentées précédemment. Fig.7a correspond à un temps t_1, et Fig.6c à un temps t_2 ultérieur . Paramètres globaux : $Move\_with$: *activé*, $Update$: $version\ oiseaux$, $NUM\_BOIDS = 100$. Pour la figure 7a : $DISTCRIT = 50$, $FORCE = 0.01$, $VISION = 100$, $MAX\_VELOCITY = 5$.

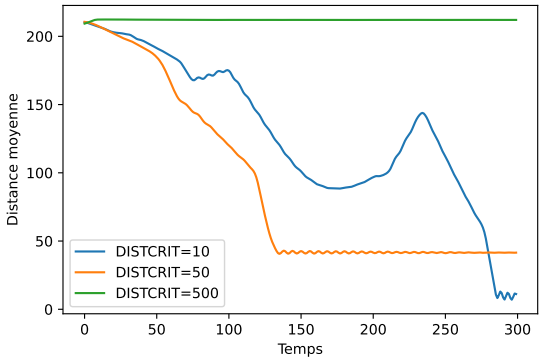
Les éléments présentés au-dessus permettent la simulation d'un vol d'oiseaux, illustré par Fig. 3. Les oiseaux vont avoir tendance à former des essaims, jusqu'à la fusion en un essaim final regroupant tous les boids. On peut étudier graphiquement l'impact des différents paramètres sur la formation d'essaims, permettant de dégager des valeurs optimale pour la modélisation de vol d'oiseaux.



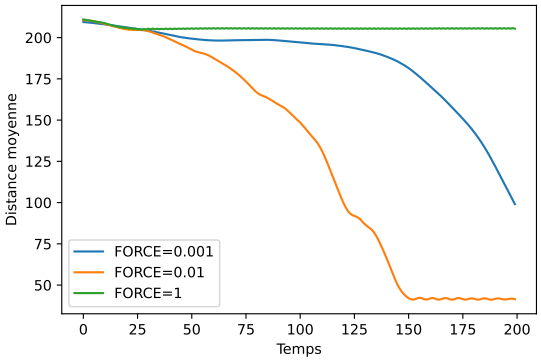
(a) Differents VELOCITY_MAX



(b) Differents VISION



(c) Differents DISTCRIT



(d) Differents FORCE

FIGURE 4 – Évolution temporelle de la distance moyenne entre les oiseaux pour différents sets de paramètres. Paramètres globaux : *Move_with* : *activé*, *Update* : *version oiseaux*, $NUM\_BOIDS = 100$. Pour la figure 7a : $DISTCRIT = 50$, $FORCE = 0.01$, $VISION = 100$, $MAX\_VELOCITY$ variable. Pour la figure 6c : $MAX\_VELOCITY = 5$, $DISTCRIT = 50$, $FORCE = 0.01$, $VISION$ variable. Pour la figure 6d : $MAX\_VELOCITY = 5$, , $FORCE = 0.01$, $VISION = 100$, $DISTCRIT$ variable. Pour la figure 4d : $MAX\_VELOCITY = 5$, $VISION = 100$, $DISTCRIT = 50$, $FORCE$ variable.

Fig. 4 présente l'évolution temporelle de la distance moyenne entre les oiseaux, pour différents paramètres. Chaque courbe correspond à une nouvelle simulation, donc avec des conditions initiale de position et vitesse différentes. La distance moyenne est un bon indicateur de l'évolution du système, puisque sa diminution est synonyme d'agroupement entre les oiseaux, donc de mouvements collectifs. On remarque premièrement que la distance moyenne débute à $t = 0$ légèrement au-dessus de 200, pour tous les paramètres. Ceci signifie que la distribution des positions est bien aléatoire et uniforme pour tous les oiseaux.

Fig. 7a met en évidence que si la vitesse maximale est trop élevée, la distance moyenne ne diminue pas. En effet, les oiseaux iront trop vite pour avoir le temps de s'attirer suffisamment les uns les autres. On remarque que pour une faible vitesse maximale = 2, la distance moyenne diminue plus lentement que pour une vitesse maximale plus élevée = 10. Les oiseaux prennent plus de temps à se regrouper, car plus de temps pour parcourir la distance les séparant. Il faut donc choisir une vitesse ni trop faible, ni trop élevée, pour pouvoir observer une évolution collective intéressante des oiseaux. Pour $MAX\_VELOCITY = 10$, la distance moyenne tend vers une valeur limite. Dans ce cas, cela correspond à l'essaim final, où les oiseaux forment une sphère, à cause de l'expression de la force dans "acceleration". La distance moyenne est de l'ordre de $DISTCRIT$, puisque les oiseaux ne voudront pas s'approcher beaucoup plus proche que cela.

Fig. 6c fixe $MAX\_VELOCITY$ à 5, mais calcule la distance moyenne pour différentes valeurs de $VISION$. Lorsque l'on choisit ce paramètre trop petit, les oiseaux ne voient pas leurs congénères, et n'interagissent que très rarement avec eux. Il leur est donc impossible de s'agglomérer, ce qu'on observe pour le cas $VISION = 10$. Pour une valeur très élevée, $VISION = 1000$, chaque oiseau voit les autres (la dimension de l'écran était alors de 650X650). Ils vont donc tous rapidement être attirés les uns par les autres avant d'atteindre la distance DISTCRIT et se repousser. Pour d'une valeur intermédiaire de $VISION = 100$, les oiseaux se peuvent se regrouper en plusieurs essaims. Ils finissent aussi par former un essaim unique, tendant donc de nouveau vers la même valeur $DistanceMoyenne \simeq DISTCRIT$ que pour $VISION = 1000$.

Fig. 6d présente l'évolution de la distance moyenne pour différente valeur de DISTCRIT. Pour des valeurs trop élevées,

les oiseaux se repoussent constamment. Apparait donc une sorte de réseau cristallin vibratoire, où quelques oiseaux sont regroupés en amas se repoussant. Ce comportement ne correspond pas à un vol d'oiseaux. Pour des valeurs plus faibles $DISTCRIT = 50$ *ou* 10, les oiseaux parviennent à se regrouper, et la distance moyenne tend vers leur valeur de DISTCRIT respective. La courbe bleue permet de facilement retracer l'évolution des essaims secondaires jusqu'au final. En effet, le choc entre deux essaims peut les faire se séparer (la distance augmente donc), jusqu'à ce qu'ils se recroisent pour peut-être fusionner (la distance diminue). C'est ce à quoi correspondent le premier et second pic.
Dans Fig. 4d, on impose un coefficient de force différent. Pour une fort coefficient, les oiseaux s'attirent rapidement en plusieurs essaims. Si ceux-ci sont plus éloignés que $VISION$, la force sera plus importante que l'effet de "move_with", ainsi ils se stabilisent et ne se rapprochent pas. C'est ce qui est observable pour $FORCE = 1$. Pour des valeurs plus faibles, les oiseaux parviennent à se regrouper, puis à fusionner les essaims. Cependant une force très faible, comme $FORCE = 0.001$, implique une plus faible accélération et donc un temps plus long pour former l'essaim final.

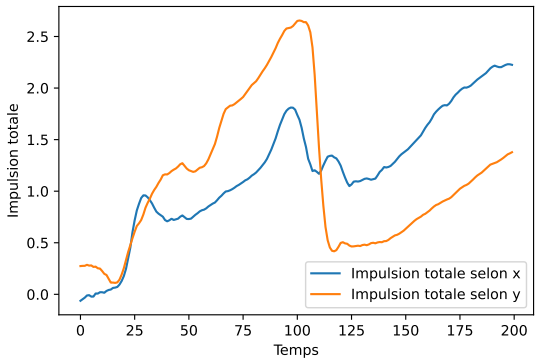


FIGURE 5 – Évolution temporelle de l'impulsion totale selon x et y. Paramètres utilisés : $Move\_with$: *activé*, $Update$: *version oiseaux*, $NUM\_BOIDS = 100$, $MAX\_VELOCITY = 5$, $DISTCRIT = 50$, $FORCE = 0.01$, $VISION = 100$.

En choisissant le bon set de paramètres, nous pouvons donc bien simuler un vol d'oiseaux. Cependant, cette simulation ne correspond pas à un système suivant des lois physique. C'est ce que montre Fig. 5. L'impulsion totale n'est pas conservée selon $\vec{x}$ et $\vec{y}$. L'énergie n'est pas conservée, et diverge, à cause de "move-with". Ceci est dû à l'utilisation de "move_with", qui est une force non conservative. En effet, l'influence des autres oiseaux impacte directement la vitesse de self, en moyennant sur le nombre d'oiseaux visible. Ceci signifie que la force dépend de la vitesse et du nombre d'oiseaux visibles. C'est cette force qui nous force à limiter la vitesse à $MAX\_VELOCITY$ dans "update", car sans cela les oiseaux accélèrent à une vitesse infinie, jusqu'à sortir de l'écran.
Il serait donc intéressant d'obtenir des comportements similaires en utilisant cette fois des règles de physique.

2 Simuler des essaims d'oiseaux avec un système physique

2.1 Choix système physique

Notre simulation de vol d'oiseaux se concentre uniquement sur le mouvement de particules ponctuelles, sans prendre en compte le vol individuel des oiseaux d'un point de vue mécanique des fluides. Ainsi, la situation se rapproche d'un gaz de particules avec interaction.
Pour simuler un système suivant des lois physiques, les forces exercées par les oiseaux doivent être conservatives, donc dérivant d'un potentiel. Les contraintes sur ce potentiel sont qu'il doit être répulsif sous $DISCRIT$ et attractif au delà. Un candidat correspondant à cela est le potentiel harmonique :

$$V(D) = \frac{k}{2} \times (D - DISTCRIT)^2 \tag{2}$$

où D est la distance entre self et l'oiseau exerçant son potentiel sur self. L'avantage du potentiel harmonique est que n'importe quel potentiel de la forme recherchée peut s'y rapporter en première approximation.

On peut en déduire la force exercée par ce potentiel, en dérivant par rapport à $-\Delta X$. Le signe moins est dû au

fait que nous avons défini la distance comme orientée de l'objet percevant la force, vers celui qui l'exerce.

$$\begin{aligned} F_x &= -\frac{\partial V}{\partial(-\Delta X)} \\ &= k\frac{\partial\sqrt{(\Delta X)^2+(\Delta Y)^2}}{\partial \Delta X} \times (D - DISTCRIT) \\ &= k\frac{\Delta X}{D}(D - DISTCRIT) \\ &= FORCE \times \Delta X\big(1 - \frac{DISTCRIT}{D}\big) \end{aligned}$$

avec $FORCE = k$. On peut effectuer de la même manière la dérivée selon ΔY, on trouve :

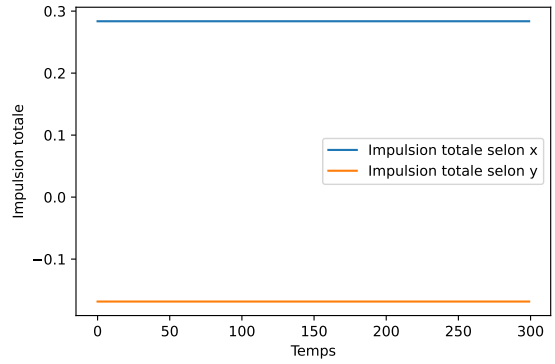
$$\vec{F} = FORCE \times \vec{D}\big(1 - \frac{DISTCRIT}{D}\big) = \begin{pmatrix} FORCE \times \Delta X(1 - \frac{DISTCRIT}{D}) \\ FORCE \times \Delta Y(1 - \frac{DISTCRIT}{D}) \end{pmatrix} \tag{3}$$

En choisissant $FORCE > 0$, on a bien $\vec{F}$ orientée vers l'objet exerçant la force pour $D > DISTCRIT$ (même signe que ΔX *ou* Y), soit une force attractive. A l'inverse la force est répulsive, i.e orientée à l'opposé de l'objet exerçant la force pour $D < DISTCRIT$. Cette force est celle utilisée à la partie précendente, et satisfait bien les deux premières règles des Boids.
Cette force a l'avantage de ne pas diverger en $D = 0$, ce qui nous permet d'avoir une meilleure résolution numérique, qu'un potentiel en $1/D$ par exemple.

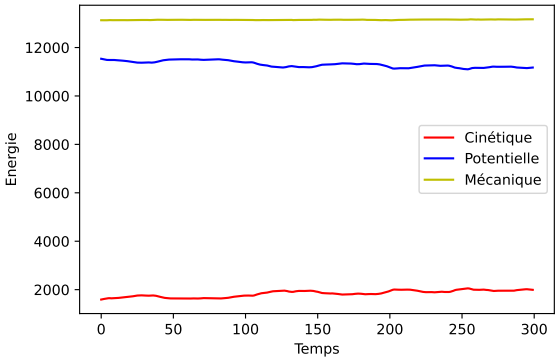
2.2 Ordre 1

Nous pouvons simuler un gaz de particules générant individuellement un potentiel harmonique sur les particules visibles ($D < VISION$), à partir du code présenté dans Sec. 1. Nous devons tout de même modifier quelques aspects. Il faut désactiver la fonction "move_with", car celle-ci n'est ni conservative, ni physique. De plus le fait de renormaliser la vitesse à $MAX\_VELOCITY$ lorsqu'elle devient trop importante n'est pas physique non plus. Ainsi on supprime les lignes à ce propos dans "update", comme présenté dans Annexe. A.2.
On peut désormais obtenir l'énergie totale (ou plutôt son double) :

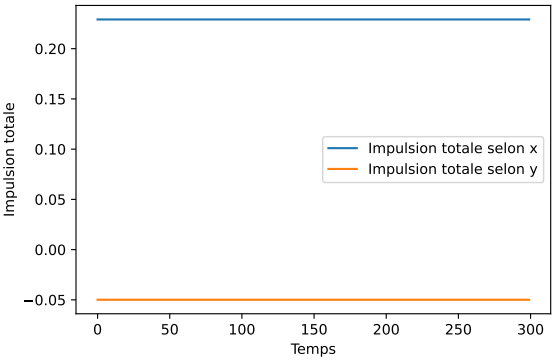
$$2 \times E_{tot} = \sum_{boids} \big(V(D) + 2 \times E_{cin}\big) = \sum_{boids} \left(\sum_{boids \in close\_boids} \frac{FORCE}{2} \times (D_i - DISTCRIT)^2 + V_x^2 + V_y^2 \right) \tag{4}$$



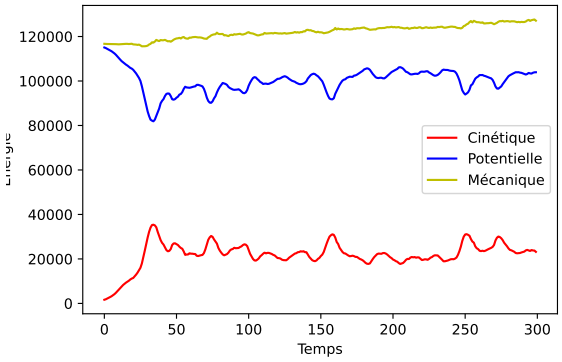
(a) Impulsion pour FORCE =0.001



(b) Energie pour FORCE =0.001



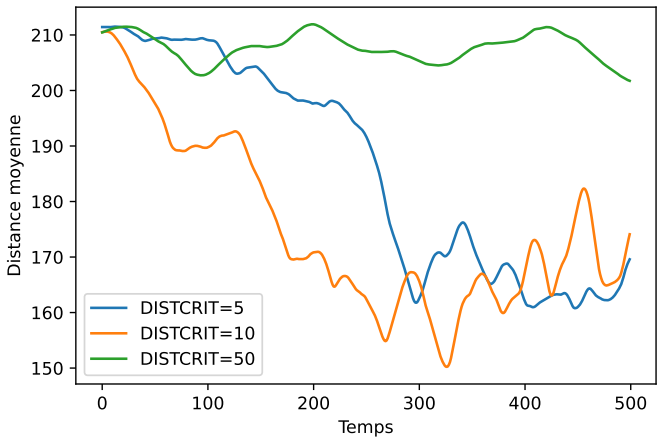
(c) Impulsion pour FORCE =0.01



(d) Energie pour FORCE =0.01

FIGURE 6 – Évolution temporelle de l'impulsion totale selon x et y, et de l'énergie potentielle, cinétique et totale. Paramètres utilisés : $Move\_with$: *désactivé*, $Update$: *version physique ordre* 1, $NUM\_BOIDS = 100$, $MAX\_VELOCITY = 5$, $DISTCRIT = 50$, $VISION = 100$. Les graphiques du haut correspondent à $FORCE = 0.001$, ceux du bas à $FORCE = 0.01$

Fig. 6 illustre l'évolution de l'impulsion totale, de l'énergie totale cinétique ,mécanique, et potentielle pour un système où $FORCE = 0.001$ et $FORCE = 0.01$. Dans les deux cas l'impulsion est parfaitement conservée. La force est donc bien physique. Pour un faible coefficient de force, l'énergie mécanique totale du système est conservée à l'échelle temporelle macroscopique. L'énergie potentielle est bien antisymétrique avec la potentielle, synomyme de transfert d'énergie entre particules : la force est bien conservative ainsi qu'elle est censée l'être. Cependant, pour une force plus importante l'énergie totale croît. Les variations d'énergie potentielle et cinétique ne se compensent plus parfaitement. C'est un problème numérique dû au fait que nos calculs sont d'ordre 1, et ne permettent donc pas une précision assez importante pour la conservation de l'énergie.



(a) Distance moyenne pour FORCE =0.001

FIGURE 7 – Évolution temporelle de la distance moyenne entre les oiseaux. Paramètres utilisés : *Move_with* : *désactivé*, *Update* : *version physique ordre* 1, $NUM\_BOIDS = 100$, $MAX\_VELOCITY = 5$, $VISION = 100$, $FORCE = 0.001$. $DISTCRIT$ variable

On peut tout de même obtenir des résultats similaire à la section précédente, même avec une force faible. Fig. 7 montre que des essaims se forment bien sur des temps longs. En effet, la distance diminue avec le temps pour de faibles distances critiques, car la force est majoritairement attractive. Cependant, on voit que le système ne finit pas par former un unique groupe, car la force est trop faible. Des particules s'échappent et rejoignent le groupe constamment. Si on choisit une faible distance critique, on parvient facilement à la formation de groupe, tout en conservant l'énergie. En Physique ce système pourrait correspondre à un gaz de particules en interaction. Le regroupement de particules lui correspondrait à une transition de phase, de gazeux vers solide par exemple. Le système physique ayant un comportement proche de celui des oiseaux présentés dans la section précédente, nous pourrions faire l'analogie inverse, c'est-à-dire en considérant que le passage d'oiseaux solitaires à des essaims serait une transition de phase de l'espèce.

Pour obtenir un système où l'énergie est conservée, tout ayant un comportement similaire aux oiseaux, il faudrait avoir une force plus importante. Pour cela, il faudrait avoir une meilleure précision de résolution des équations différentielles régissant le problème.

2.3 Ordre 4

Afin de pallier à ce problème, nous avons donc décidé de modifier notre programme pour implémenter une méthode plus précise d'ordre 4. Il s'agit de la méthode de Runge-Kutta avec dérivée seconde.
La méthode va consister à venir calculer des points intermédiaires successifs afin de pouvoir trouver les vraies positions et vitesses finales avec une plus grande précision. Voici plus précisément les calculs de la méthode :

Soient x_t et v_t la position et vitesse à l'instant précédent, et x_{t+1} et v_{t+1} celles que nous voulons calculer pour l'instant suivant. On a de plus une fonction qui nous donne l'accélération comme étant $a = f(x, v)$. On pose h comme étant le pas de temps.
On va commencer par calculer 4 accélérations différentes à différentes positions successives de l'espace des phases :

$$a_1 = f(x_t, v_t)$$
$$a_2 = f(x_t + h/2 \cdot v_t, v_t + h/2 \cdot a_1)$$
$$a_3 = f(x_t + h/2 \cdot v_t + h^2/4 \cdot a_1, v_t + h/2 \cdot a_2)$$
$$a_4 = f(x_t + h \cdot v_t + h^2/2 \cdot a_2, v_t + h \cdot a_3)$$

À partir desquelles on peut alors obtenir :

$$x_{t+1} = x_t + h \cdot v_t + h^2/6 \cdot (a_1 + a_2 + a_3)$$
$$v_{t+1} = v_t + h/6 \cdot (a_1 + 2a_2 + 2a_3 + a_4)$$

Afin d'implémenter ceci dans notre programme, la fonction accélération est modifiée dans un premier temps pour

retourner les valeurs de l'accélération selon x et y, au lieu de les ajouter directement à la vitesse comme elle le faisait auparavant. On introduit alors une nouvelle liste "acc" qui va venir stocker les valeurs de "accelération" à chacune des 4 itérations de Runge-Kutta. Pour faire évoluer les oiseaux dans la temps, il va désormais falloir appeler 4 fois la fonction accélération dans chacune des 4 configurations successives de l'espace des phases. Ceci sera réalisé comme suit (les nouvelles lignes sont commentées ensuite) :

```
for oiseau in oiseau_list: oiseau.acc=[] # initialisation accélérations

for i in range(4): # on appelle 4 fois la fonction
        for oiseau in oiseau_list:
            closeBoids=[]
            for otherBoid in oiseau_list:

                if otherBoid == oiseau: continue

                if oiseau.distance(otherBoid) < VISION:
                    closeBoids.append(otherBoid)

            if len(closeBoids)>0:
                oiseau.acc.append(oiseau.acceleration(closeBoids)) # stocke la valeur a_i
            else: oiseau.acc.append((0,0))

        all_sprites_list.update(i) # on update tous les paramètres des oiseaux
```

La fonction update prend désormais i en paramètre : c'est elle qui va gérer la transition d'un point de l'espace des phases à un autre.
Concrètement, on va vouloir que la première fois qu'elle soit appelée, elle réalise les opérations :

$$x+ = h * v/2$$

$$v+ = h * a_1/2$$

Ainsi de suite pour chacune des étapes de Runge-Kutta. Comme précédemment, la fonction update va également gérer les bords, puis à la dernière étape va modifier la valeur "rect" avec la valeur finale de "position". Ainsi update devient finalement :

```
    def update(self,i):

        #RK4:
        if i==0:
            self.positionX += h*self.velocityX/2
            self.positionY += h*self.velocityY/2
            self.velocityX += h*self.acc[0][0]/2
            self.velocityY += h*self.acc[0][1]/2
        elif i==1:
            self.positionX += h**2 * self.acc[0][0]/4
            self.positionY += h**2 * self.acc[0][1]/4
            self.velocityX += h*(self.acc[1][0]-self.acc[0][0])/2
            self.velocityY += h*(self.acc[1][1]-self.acc[0][1])/2
        elif i==2:
            self.positionX += h*self.velocityX/2 + h**2 *(self.acc[1][0]-self.acc[0][0])/4
            self.positionY += h*self.velocityY/2 + h**2 *(self.acc[1][1]-self.acc[0][1])/4
            self.velocityX += h*(self.acc[2][0]-self.acc[1][0]/2)
            self.velocityY += h*(self.acc[2][1]-self.acc[1][1]/2)
        else:
            self.positionX += h**2 * (self.acc[0][0]-2*self.acc[1][0]+self.acc[2][0])/6
            self.positionY += h**2 * (self.acc[0][1]-2*self.acc[1][1]+self.acc[2][1])/6
            self.velocityX += h*(self.acc[0][0]+2*self.acc[1][0]-4*self.acc[2][0]+self.acc[3][0])/6
            self.velocityY += h*(self.acc[0][1]+2*self.acc[1][1]-4*self.acc[2][1]+self.acc[3][1])/6

        #Bords:
        if self.positionX < BORDER: #bords
            self.positionX += SCREEN_WIDTH - 2*BORDER
```

```
if self.positionX > SCREEN_WIDTH - BORDER:
    self.positionX -= SCREEN_WIDTH - 2*BORDER
if self.positionY < BORDER:
    self.positionY += SCREEN_HEIGHT -2* BORDER
if self.positionY > SCREEN_HEIGHT - BORDER:
    self.positionY -= SCREEN_HEIGHT - 2*BORDER

#Affichage:
if i==3:
    self.rect.x = round(self.positionX)
    self.rect.y = round(self.positionY)
```

Maintenant que la méthode a été implémentée, reste à savoir si elle fonctionne bel et bien mieux que la méthode d'ordre 1. Pour cela on va reprendre les valeurs pour lesquelles la simulation d'ordre 1 commençait à montrer des difficultés et l'on va la relancer avec RK4 :

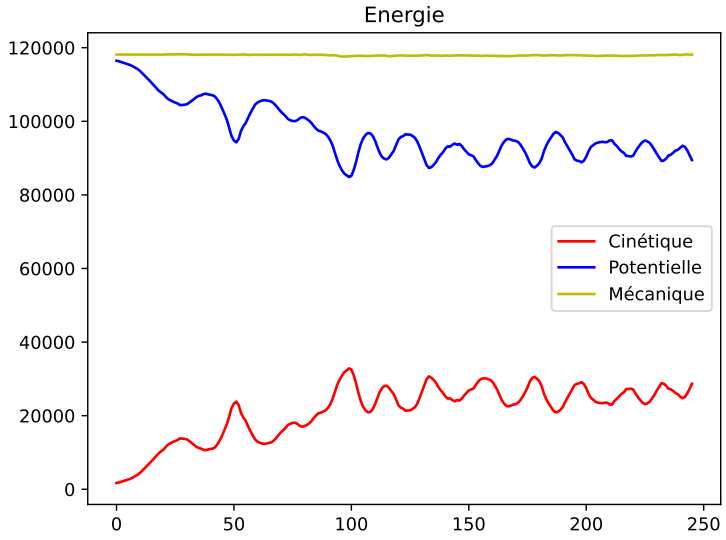
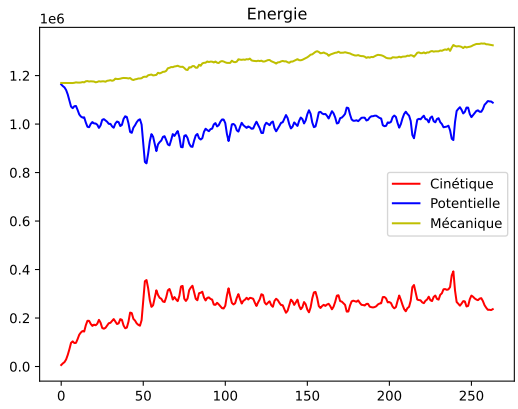
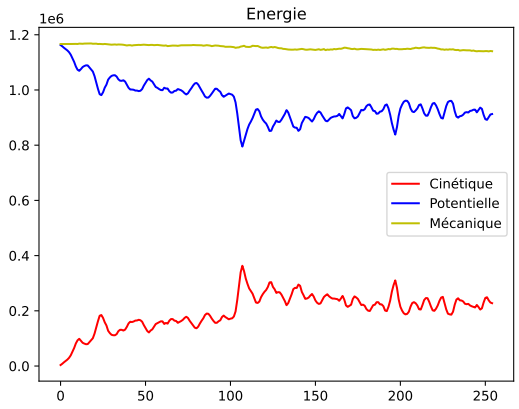


FIGURE 8 – Évolution temporelle de l'énergie potentielle, cinétique et totale. Paramètres utilisés : Move_with : désactivé, Update : version physique ordre 4, NUM_BOIDS = 100, MAX_VELOCITY = 5, DISTCRIT = 50, VISION = 100, FORCE = 0.01, h=0.9

On voit que maintenant on peut utiliser une force de 0.01 sans problème, la simulation d'ordre 4 a bien permis de régler le problème de conservation de l'énergie. On peut donc essayer de pousser un peu plus loin avec FORCE=0.1, illustré dans Fig. 9.



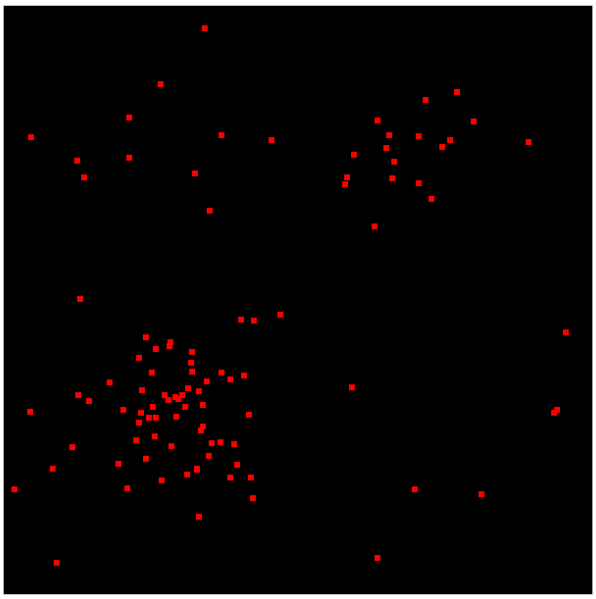
(a) h=0.9



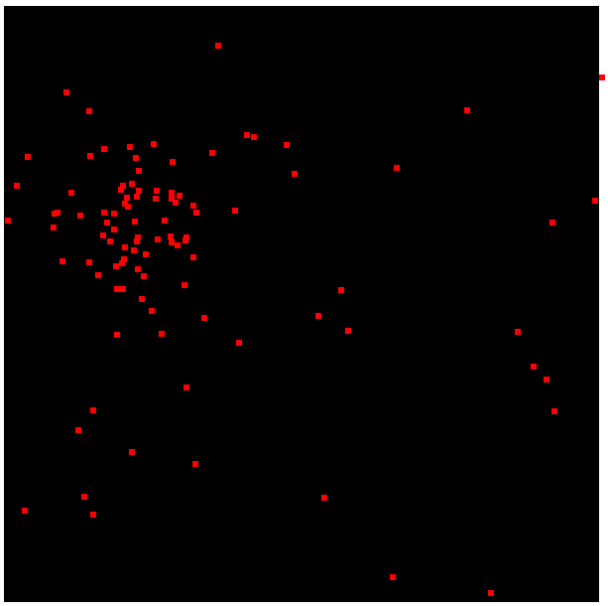
(b) h=0.5

FIGURE 9 – Évolution temporelle de l'énergie potentielle,voulu, nous avons et totale. Paramètres utilisés : Move_with : désactivé, Update : version physique ordre 4, NUM_BOIDS = 100, MAX_VELOCITY = 5, DISTCRIT = 50, VISION = 100, FORCE = 0.1

On observe qu'avec un pas de temps h=0.9, on a de nouveau un système qui ne conserve pas son énergie. En revanche, si on diminue le pas de temps h, la simulation va être plus lente à l'écran, mais la précision en est améliorée. Il devient possible d'utiliser des forces 100 fois plus intenses que ce qui était fait à l'ordre 1.
Maintenant que nous avons un programme avec de vraies lois physiques respectant les lois de conservation, et qu'il permet d'atteindre des forces non négligeables, se pose la question de savoir si nous pouvons reproduire des résultats se rapprochant du comportement d'essaims d'oiseaux. Lorsque l'on lance la simulation avec les paramètres précédents on obtient ce type de comportements :



(a) Peu après le début



(b) Après quelque temps

FIGURE 10 – Images de la simulation RK4 avec FORCE=0.1

On observe bien l'apparition d'essaims dès le départ, qui finissent par fusionner en un essaim unique. Ceci dit ces essaims sont très diffus et les particules les composant y sont liées de manière très faible. Leur vitesse notamment reste toujours un frein à la formation d'essaims plus compacts.
Cela peut aussi se voir en regardant l'évolution de la distance moyenne :

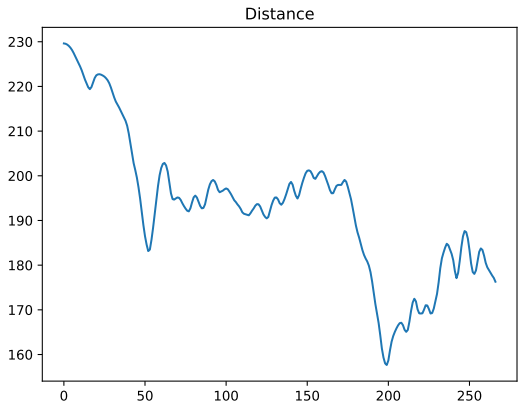


FIGURE 11 – Distance moyenne au cours du temps

La distance ne diminue pas de manière très importante, et il lui arrive de réaugmenter.

Pour obtenir le résultat voulu nous donc eu l'idée d'introduire une force de frottement qui va réduire petit à petit la vitesse des particules.

2.4 Avec du frottement

L'idée est d'introduire une force de frottement proportionnelle à la vitesse des oiseaux pour les faire ralentir. Une telle force de frottement fluide est difficilement envisageable pour des particules qui n'auraient aucun fluide dans lequel frotter, cette simulation doit donc reproduire un système macroscopique. On peut par exemple penser à des balles souples dans une piscine qui seraient attirées les unes par les autres (par exemple à l'aide d'aimants à l'intérieur), leur rayon représenterait la DISTCRIT du potentiel harmonique.
Nous allons essayer de programmer un tel système et de voir si l'on peut reproduire des comportements d'essaims d'oiseaux avec.

Une force de frottement peut être simplement implémentée en modifiant la fonction "acceleration", en ajoutant un terme à ce qu'elle retourne (on introduit une nouvelle grandeur FROTT pour régler l'intensité du frottement) :

```
return (ForceX*FORCE-FROTT*self.velocityX,ForceY*FORCE-FROTT*self.velocityY)
```

Ceci dit cette force non conservative qu'on ajoute va de fait faire décroitre l'énergie mécanique, donc on ne pourra plus savoir si cette décroissance est due à la force ou aux imprécisions numériques. Pour remédier à cela, il va falloir calculer le travail de cette force. Ce dernier est égal à la force multipliée par le déplacement de la balle : $W_{frottement} = \vec{f_{frottement}} \cdot \vec{dl}$. Il faut donc conserver la position de la balle à l'instant précédent : "positinit". Mais il va aussi falloir faire attention au fait qu'à chaque fois que l'on fait traverser un mur à la balle, on va brusquement changer sa position et donc fausser le $\vec{dl}$. Il faut donc à chaque fois qu'on modifie la position ainsi, modifier la position initiale de la même manière afin de conserver le bon $\vec{dl}$. La fonction pour les bords devient :

```
if self.positionX < BORDER: #bords
    self.positionX += SCREEN_WIDTH - 2*BORDER
    self.positinitX += SCREEN_WIDTH - 2*BORDER
if self.positionX > SCREEN_WIDTH - BORDER:
    self.positionX -= SCREEN_WIDTH - 2*BORDER
    self.positinitX -= SCREEN_WIDTH - 2*BORDER
if self.positionY < BORDER:
    self.positionY += SCREEN_HEIGHT -2* BORDER
    self.positinitY += SCREEN_HEIGHT -2* BORDER
if self.positionY > SCREEN_HEIGHT - BORDER:
    self.positionY -= SCREEN_HEIGHT - 2*BORDER
    self.positinitY -= SCREEN_HEIGHT - 2*BORDER
```

Désormais, c'est donc la somme de ce travail et de l'énergie mécanique qui devra être conservée pour que le modèle soit considéré valide.

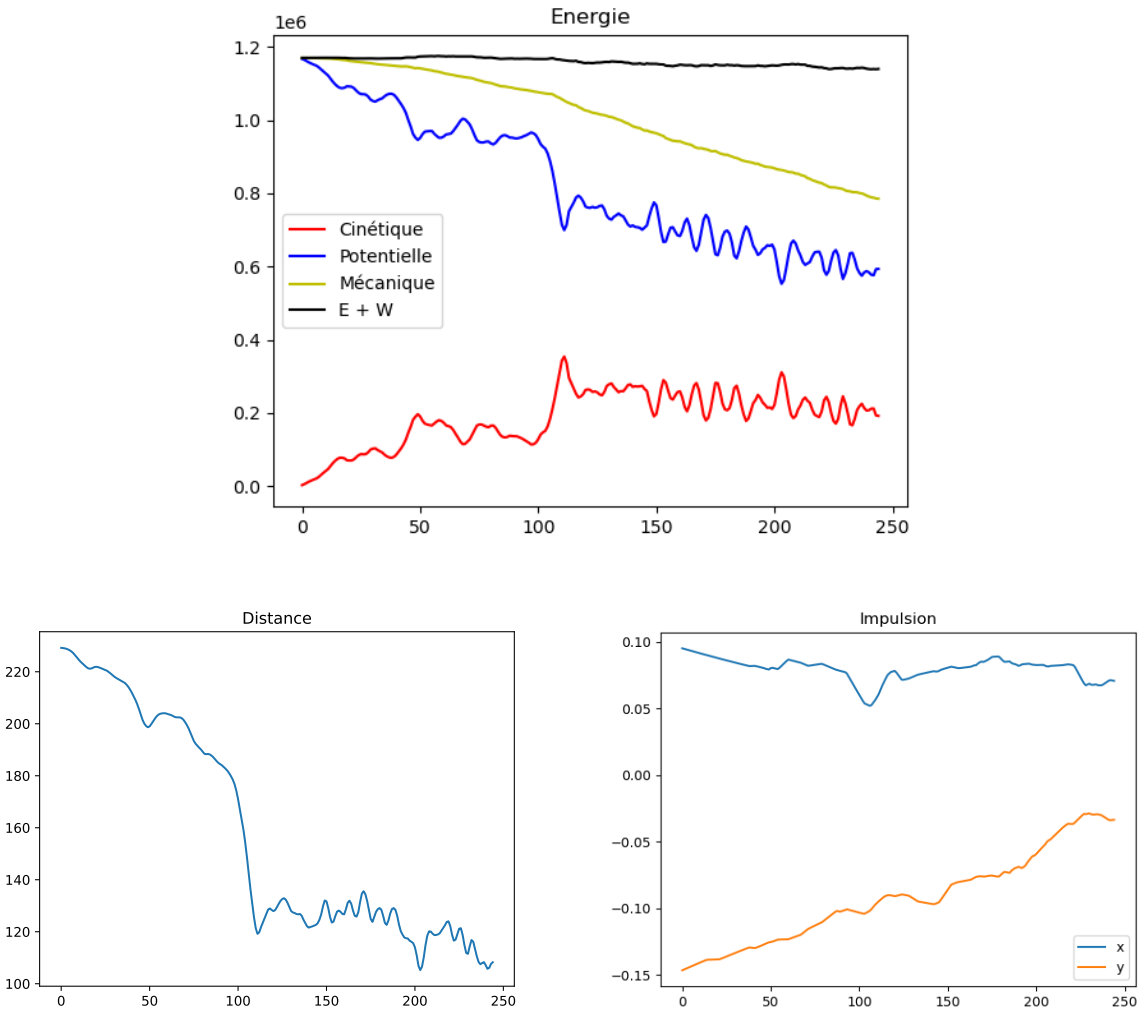
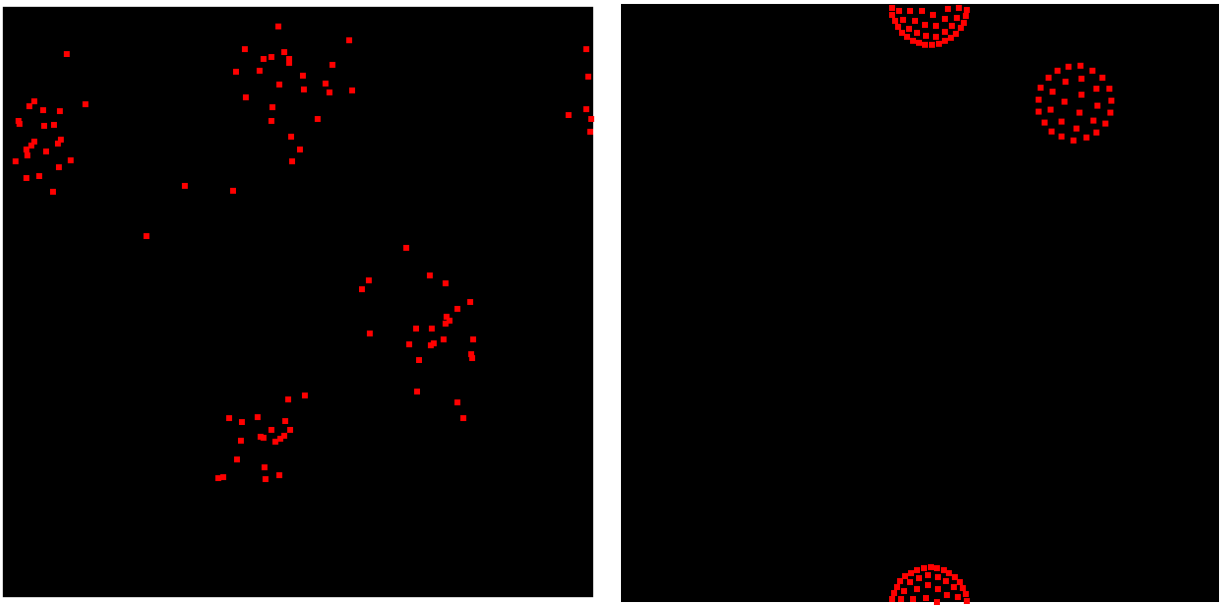


FIGURE 12 – Évolution temporelle des énergies totales, de la distance moyenne, et de l'impulsion totale. Paramètres utilisés : Move_with : désactivé, Update : version physique ordre 4, NUM_BOIDS = 100, MAX_VELOCITY = 5, DISTCRIT = 50, VISION = 100, FORCE = 0.1, h=0.4, FROTT=0.01

On observe qu'en ayant introduit un frottement relativement faible on conserve bien la somme du travail et de l'énergie mécanique. Cependant, nous avons dû réduire la valeur de h : l'introduction d'une dépendance de la force en la vitesse, déterminée par RK4, augmente les erreurs numériques. On note que la distance a diminué rapidement comme voulu : des essaims se sont formés. L'impulsion ne se conserve en revanche plus : elle va logiquement tendre vers 0.
Avec ce frottement on arrive à obtenir des essaims bien mieux délimités qui se déplacent à travers l'écran :



(a) Peu après le début (b) Après quelque temps

FIGURE 13 – Images de la simulation RK4 avec FORCE=0.1, FROTT=0.01

On observe néanmoins que les essaims tendent rapidement à devenir complètements immobiles. Ils adoptent alors la formation en boule, qui minimise leur énergie potentielle.
Une idée de développement serait de rajouter une température de fond, qu'on pourrait même rendre réglable avec un curseur, qui communiquerait ainsi constamment de l'énergie aux balles afin de leur faire conserver un minimum d'énergie et donc un mouvement. La formation en boule correspondrait alors à un état critique où la température n'est pas suffisante pour briser les liaisons potentielles.

3 Conclusion

Lors de notre projet nous avons d'abord mis en place une simulation d'oiseaux selon les règles de Boids. Ces règles arrivent à reproduire un comportement très esthétique, se rapprochant visuellement d'essaims d'oiseaux. Notre idée à partir de là a été de vérifier par simulation s'il était possible de reproduire les comportements apparents d'oiseaux en faisant appel à des règles issues plutôt de la physique.
Nous avons montré pour commencer qu'une simulation effective de règles physiques avec autant de paramètres se devait d'être assez précise. En l'occurrence il faut utiliser des approximations au moins d'ordre 4 ; la méthode de Runge-Kutta dans notre cas.
Partant de ces simulations nous avons alors montré qu'il était effectivement possible de penser des systèmes obéissant à des règles physiques et arrivant à transitionner d'un état uniforme à des formations en essaims. Ceci dit, ces essaims ne rendent pas tout-à-fait l'effet esthétique des essaims d'oiseaux. C'est notamment dû à l'absence de loi physique où les vitesses des constituants vont s'aligner avec celles de leurs voisins. Une piste à explorer serait l'ajout d'un champ de vent, où les oiseaux créeraient un mouvement d'aspiration dans leur sillage, attirant donc les oiseaux les suivant.
Nos simulations sont donc intéressantes pour étudier des systèmes où des amas apparaissent à partir d'une distribution quelconque de constituants. Nos modélisations peuvent également correspondre à des mouvements organisés de foules.
Le modèle pourrait être amélioré en ajoutant des obstacles, des effets de fluides, une troisième dimension d'espace, ou restreindre le champ de vision des oiseaux à 180°. Aussi, il spourrait être intéressant d'aborder le problème des oiseaux par le biais de l'entropie.

Références

[Dow07] Ben DOWLING. *Boids*. http ://www.coderholic.com/boids/. 2007. (Visité le 14/04/2023).

[Rey87] Craig W. REYNOLDS. « Flocks, herds and schools : A distributed behavioral model ». In : *Proceedings of the 14th annual conference on Computer graphics and interactive techniques* (1987).

A Codes

Nous avons obtenu 2 codes similaires. Le premier regroupe la résolution via les règles boids, ou bien par le système physique, avec une résolution d'ordre 1. Le second est d'ordre 4 et ajoute une force de frottement au système physique.

A.1 Pour simuler des oiseaux à l'ordre 1

```
import pygame
from random import randint, uniform
from math import sqrt
import numpy as np
import matplotlib.pyplot as plt

# === Paramètre affichage ===

FPS=100
SCREEN_SIZE = SCREEN_WIDTH, SCREEN_HEIGHT = 650, 650 #Taille écran
BORDER = 50 #taille bord
COLOR1 = (255, 255, 255) #couleur bord
COLOR2 = (0,0,0) # Couleur intérieur

# === Paramètres système ===

MAX_VELOCITY = 5 #Vitesse maximale
NUM_BOIDS = 100 #Nombres d'oiseaux simulés

VISION = 100 #distance à laquelle les oiseaux se voient
DISTCRIT = 50 #distance à partir de laquelle les oiseaux se repoussent

FORCE = 0.01 #coefficient force

# === Caractéristiques oiseaux ===

class Oiseau(pygame.sprite.Sprite):

    def __init__(self, x, y):

        super().__init__() #Appel init de la classe mère pygame.sprite.Sprite
        # et l'applique à Oiseau, donc self
        self.image = pygame.Surface([6, 6]) #crée surface de dimension [a,b]
        self.image.fill("red")  # Remplit la surface de couleur

        #positionne l'image aux coord (x,y)
        self.rect = self.image.get_rect(center=(x,y))

        #genere vitesse alétoire bornée par Max_velocity selon chaque axe
        self.velocityX = uniform(-MAX_VELOCITY, MAX_VELOCITY)
        self.velocityY = uniform(-MAX_VELOCITY, MAX_VELOCITY)

        # On définit une position pouvant prendre des valeurs décimales:
        self.positionX=x
        self.positionY=y


    #Calcule distance sur l'axe des X, orientée de self vers oiseau
    def distanceX(self, oiseau):
        #distance vers oiseau
        distX = oiseau.positionX - self.positionX
        #verifie si oiseau pas plus proche via le bord
        if abs(distX)>=SCREEN_WIDTH-2*BORDER-abs(distX):
            #si c'est le cas pointe vers à travers le bord
```

```
        return distX - (SCREEN_WIDTH-2*BORDER)*np.sign(distX)
    else: return distX


#Calcule distance sur l'axe des X, orientée de self vers oiseau
def distanceY(self, oiseau):
    distY = oiseau.positionY - self.positionY
    if abs(distY)>=SCREEN_HEIGHT-2*BORDER-abs(distY):
        return distY - (SCREEN_HEIGHT-2*BORDER)*np.sign(distY)
    else: return distY


#Calcule distance algébrique entre self et oiseau
def distance(self, oiseau):
    return sqrt(self.distanceX(oiseau)**2+self.distanceY(oiseau)**2)


#Force subie par l'oiseau par influence des oiseaux de oiseau_list
def acceleration(self, oiseau_list):
    ForceX = 0
    ForceY = 0
    for oiseau in oiseau_list:
        xdiff = self.distanceX(oiseau)
        ydiff = self.distanceY(oiseau)
        distance = sqrt(xdiff**2+ydiff**2)

        if distance !=0:
            #stocke force exercée par chaque oiseau sur self
            ForceX -= -xdiff*(1-DISTCRIT/distance)
            ForceY -= -ydiff*(1-DISTCRIT/distance)

    #modification de la vitesse par la force
    self.velocityX += ForceX * FORCE
    self.velocityY += ForceY * FORCE


#Loi non physique, utile pour simuler plus esthétiquement des oiseaux
#Permet de faire voler self aligné avec les oiseaux de oiseau_list
def move_with(self, oiseau_list):
    avgX = 0
    avgY = 0
    for oiseau in oiseau_list:
        avgX += oiseau.velocityX
        avgY += oiseau.velocityY

    # avgX est la moyenne des composantes des vitesses selon X des oiseaux
    # de oiseau_list. Idem selon Y pour avgY.
    avgX /= len(oiseau_list)
    avgY /= len(oiseau_list)

    #On modifie la vitesse de self pour l'aligner avec le groupe.
    self.velocityX += (avgX/40)
    self.velocityY += (avgY/40)


#Fonction pour update la position pour des oiseaux, lorsqu'on souhaite
#utiliser move_with.
def update(self):
    #Necessaire pour eviter la divergence de la vitesse avec move_with.
    #On vérifie que la vitesse ne dépasse pas MAX_VELOCITY.
    if abs(self.velocityY) >= MAX_VELOCITY or abs(self.velocityX) >= MAX_VELOCITY:
```

```
            scaleFactor = MAX_VELOCITY / max(abs(self.velocityY), abs(self.velocityX))
            #on ajuste la composante trop grande à Max velocity,
            #et l'autre garde la proportion (pour garder direction).
            self.velocityY *= scaleFactor
            self.velocityX *= scaleFactor

        #On déplace l'oiseau selon sa vitesse.
        self.positionX += self.velocityX
        self.positionY += self.velocityY

        #Si l'oiseau dépasse un bord on modifie sa position de l'autre côté.
        if self.positionX < BORDER: #bords
            self.positionX += SCREEN_WIDTH - 2*BORDER
        if self.positionX > SCREEN_WIDTH - BORDER:
            self.positionX -= SCREEN_WIDTH - 2*BORDER
        if self.positionY < BORDER:
            self.positionY += SCREEN_HEIGHT -2* BORDER
        if self.positionY > SCREEN_HEIGHT - BORDER:
            self.positionY -= SCREEN_HEIGHT - 2*BORDER

        # On modifie la position de l'image de l'oiseau sur l'écran.
        self.rect.x = round(self.positionX)
        self.rect.y = round(self.positionY)


# === Initialisation système ===

#On lance Pygame
pygame.init()
#On crée l'écran aux dimensions choisies
screen = pygame.display.set_mode(SCREEN_SIZE)

#Crée un groupe oiseau
oiseau_list = pygame.sprite.Group()

# Crée chaque oiseaux
for i in range(NUM_BOIDS):
    #Les positionne aléatoirement sur l'écran
    oiseau = Oiseau(randint(BORDER, SCREEN_WIDTH-BORDER), randint(BORDER, SCREEN_HEIGHT-BORDER))
    #Ajoute l'oiseau au groupe
    oiseau_list.add(oiseau)


# === Simulation ===

#Crée une horloge
clock = pygame.time.Clock()

#initialise la simulation
running = True

#Datas que l'on pourrait vouloir stocker
DIST_MOY=[]
VITX_MOY=[]; VITY_MOY=[]
NRJ=[]
E_POT=[]


while running:

    #Securité pour pouvoir arrêter le programme
```

```
    for event in pygame.event.get():
        #Si on stoppe pygame on stoppe la simulation
        if event.type == pygame.QUIT:
            running = False
        #Si on appuie sur une touche vérifie si on veut quitter le programme
        if event.type == pygame.KEYDOWN:
            #Si on appuie sur ECHAP on arrête le programme
            if event.key == pygame.K_ESCAPE:
                running = False

# ==== Degré1 ===== :

    #On regarde tous les oiseaux
    for oiseau in oiseau_list:
        closeBoids=[]
        #Pour chaque oiseau on trouve ceux qu'il peut voir
        for otherBoid in oiseau_list:
            #il ne se voit pas lui même
            if otherBoid == oiseau: continue
            #Si les autres oiseaux sont plus près que VISION alors il les voit
            if oiseau.distance(otherBoid) < VISION:
                closeBoids.append(otherBoid)

        #Si l'oiseau ne voit personne alors il ne subie pas de force
        if len(closeBoids)>0:
            #Sinon on lui applique acceleration pour tous les oiseaux proches
            oiseau.acceleration(closeBoids)
            #On peut aussi lui appliquer move_with (système non physique)
            oiseau.move_with(closeBoids)

    #Une fois toutes les vitesses des oiseaux modifiées, on update
    # leur positions simultanément.
    oiseau_list .update()


# ==== Graphes ==== :

    moy=0.
    vitx=0.; vity=0.
    nrj=0.
    e_pot=0.

    for oiseau in oiseau_list:
        closeBoids=[]
        for otherBoid in oiseau_list:
            if otherBoid == oiseau: continue
            distance = oiseau.distance(otherBoid)
            moy+= distance
            if distance < VISION:
                closeBoids.append(otherBoid)
                e_pot+= FORCE*(distance-DISTCRIT)**2/2
            else: e_pot+= FORCE*(VISION-DISTCRIT)**2/2

        vx=oiseau.velocityX; vy=oiseau.velocityY
        vitx+= vx; vity+= vy
        nrj+= vx**2 + vy**2

    DIST_MOY.append(moy)
    VITX_MOY.append(vitx); VITY_MOY.append(vity)
    NRJ.append(nrj)
    E_POT.append(e_pot)
```

```
    # === Affichage ===:

    #On affiche les bords et le fond
    screen.fill(COLOR1)
    Bord=pygame.draw.rect(screen,COLOR2,pygame.Rect(BORDER,BORDER,SCREEN_WIDTH-2*BORDER,SCREEN_HEIGHT-2*BO

    # On affiche tous les oiseaux
    oiseau_list .draw(screen)

    # On met à jour l'écran
    pygame.display.update() #met à jour tout l'écran

    # Règle la vitesse d'affichage
    clock.tick(FPS)

#=== Plots === :

DIST_MOY=np.array(DIST_MOY)
VITX_MOY=np.array(VITX_MOY); VITY_MOY=np.array(VITY_MOY)
NRJ=np.array(NRJ); E_POT=np.array(E_POT)

DIST_MOY/=(NUM_BOIDS-1)*NUM_BOIDS
VITX_MOY/=NUM_BOIDS; VITY_MOY/=NUM_BOIDS

plt.figure(1)
plt.plot(DIST_MOY)
plt.ylabel('Distance moyenne'); plt.xlabel('Temps')
plt.savefig("Oiseau_o1_dist_moy", format='pdf',dpi=1000)

plt.figure(2)
plt.plot(VITX_MOY,label='Impulsion selon x'); plt.plot(VITY_MOY,label='Impulsion selon y')
plt.legend()
plt.ylabel('Impulsion'); plt.xlabel('Temps')
plt.savefig("Oiseau_o1_impulsion", format='pdf',dpi=1000)

plt.figure(3)
plt.plot(NRJ,'r',label='Cinétique'); plt.plot(E_POT,'b',label='Potentielle')
plt.plot(NRJ+E_POT,'y',label='Mécanique')
plt.legend();
plt.ylabel('Energie'); plt.xlabel('Temps')
plt.savefig("Oiseau_o1_Energie", format='pdf',dpi=1000)

# === Fin ===
#permet de fermer pygame, si par exemple on veut  faire un autre programme
pygame.quit()
```

A.2 Pour simuler un système physique à l'odre 1

Il suffit de commenter la ligne "$oiseau.move_with(closeBoids)$" dans la section "Degré1", et de remplacer la fonction uptade par la fonction suivante :

```
    #Fonction pour update la position pour des oiseaux, lorsqu'on ne souhaite
    #pas utiliser move_with (système physique)
    def update(self):
      #On déplace l'oiseau selon sa vitesse.
        self.positionX += self.velocityX
        self.positionY += self.velocityY

         #Si l'oiseau dépasse un bord on modifie sa position de l'autre côté.
        if self.positionX < BORDER:
```

```
            self.positionX += SCREEN_WIDTH - 2*BORDER
        if self.positionX > SCREEN_WIDTH - BORDER:
            self.positionX -= SCREEN_WIDTH - 2*BORDER
        if self.positionY < BORDER:
            self.positionY += SCREEN_HEIGHT -2* BORDER
        if self.positionY > SCREEN_HEIGHT - BORDER:
            self.positionY -= SCREEN_HEIGHT - 2*BORDER

        #On modifie la position de l'image de l'oiseau sur l'écran.
        self.rect.x = round(self.positionX)
        self.rect.y = round(self.positionY)
```

A.3 Pour simuler un système physique à l'ordre 4 avec frottements

```
import pygame
from random import randint, uniform
from math import sqrt
import numpy as np
import matplotlib.pyplot as plt

# === Interface ===

FPS=100 # vitesse d'affichage
h=0.9 # pas de temps
SCREEN_SIZE = SCREEN_WIDTH, SCREEN_HEIGHT = 700, 700
# doit être carré si la vision des oiseaux porte loin
BORDER = 50 # épaisseur bordure de l'écran
COLOR1 = (255, 255, 255) #RGB extérieur
COLOR2 = (0,0,0) # Intérieur

# === Variables ===

MAX_VELOCITY = 5 # vitesse initiale maximale
NUM_BOIDS = 100

VISION = 100 #distance à laquelle les oiseaux se voient
DISTCRIT = 50 #distance à partir de laquelle les oiseaux se repoussent

FORCE = 0.1 # coefficient force
FROTT=0.01 # coeff frottement

# === construction oiseaux ===

class Oiseau(pygame.sprite.Sprite):

    def __init__(self, x, y):
        super().__init__() # appelle init de la classe mère, donc de pygame.sprite.Sprite
        # et l'applique à oiseau, donc self

        #self.image = pygame.image.load("oiseaux.png").convert() # image remplissant le bloc
        self.image = pygame.Surface([6, 6]) # crée une surface
        self.image.fill("red")  # couleur oiseau

        # Crée un rectangle avec les dimensions de l'image:
        self.rect = self.image.get_rect(center=(x,y))
        # rect nécessairement ce nom car utilisé par draw

        self.velocityX = uniform(-MAX_VELOCITY, MAX_VELOCITY) #genere vitesse alétoire, non entière
        self.velocityY = uniform(-MAX_VELOCITY, MAX_VELOCITY)
        # nom velocityX/Y n'importe pas
```

```
        # On définit une position non entière:
        self.positionX=x
        self.positionY=y

        # On définit l'accélération :
        self.acc=[]

        # Positions initiales pour calcul du travail
        self.positinitX=0
        self.positinitY=0

    def distanceX(self, oiseau):
        distX = oiseau.positionX - self.positionX
        if abs(distX)>=SCREEN_WIDTH-2*BORDER-abs(distX):
            return distX - (SCREEN_WIDTH-2*BORDER)*np.sign(distX)
        else: return distX

    def distanceY(self, oiseau):
        distY = oiseau.positionY - self.positionY
        if abs(distY)>=SCREEN_HEIGHT-2*BORDER-abs(distY):
            return distY - (SCREEN_HEIGHT-2*BORDER)*np.sign(distY)
        else: return distY

    def distance(self, oiseau):
        return sqrt(self.distanceX(oiseau)**2+self.distanceY(oiseau)**2)


    def acceleration(self, oiseau_list):

        ForceX = 0
        ForceY = 0

        for oiseau in oiseau_list:

            xdiff = self.distanceX(oiseau)
            ydiff = self.distanceY(oiseau)
            distance = sqrt(xdiff**2+ydiff**2)

            ForceX -= -xdiff*(1-DISTCRIT/distance) # force harmonique
            ForceY -= -ydiff*(1-DISTCRIT/distance)

        #frottements incorporés:
        return (ForceX*FORCE-FROTT*self.velocityX,ForceY*FORCE-FROTT*self.velocityY)


    def update(self,i):

        # RK4:
        if i==0:
            self.positinitX = self.positionX
            self.positinitY = self.positionY

            self.positionX += h*self.velocityX/2
            self.positionY += h*self.velocityY/2
            self.velocityX += h*self.acc[0][0]/2
            self.velocityY += h*self.acc[0][1]/2
        elif i==1:
            self.positionX += h**2 * self.acc[0][0]/4
            self.positionY += h**2 * self.acc[0][1]/4
            self.velocityX += h*(self.acc[1][0]-self.acc[0][0])/2
            self.velocityY += h*(self.acc[1][1]-self.acc[0][1])/2
```

```
        elif i==2:
            self.positionX += h*self.velocityX/2 + h**2 *(self.acc[1][0]-self.acc[0][0])/4
            self.positionY += h*self.velocityY/2 + h**2 *(self.acc[1][1]-self.acc[0][1])/4
            self.velocityX += h*(self.acc[2][0]-self.acc[1][0]/2)
            self.velocityY += h*(self.acc[2][1]-self.acc[1][1]/2)
        else:
            self.positionX += h**2 * (self.acc[0][0]-2*self.acc[1][0]+self.acc[2][0])/6
            self.positionY += h**2 * (self.acc[0][1]-2*self.acc[1][1]+self.acc[2][1])/6
            self.velocityX += h*(self.acc[0][0]+2*self.acc[1][0]-4*self.acc[2][0]+self.acc[3][0])/6
            self.velocityY += h*(self.acc[0][1]+2*self.acc[1][1]-4*self.acc[2][1]+self.acc[3][1])/6

        #Bords:
        if self.positionX < BORDER: #bords
            self.positionX += SCREEN_WIDTH - 2*BORDER
            self.positinitX += SCREEN_WIDTH - 2*BORDER
        if self.positionX > SCREEN_WIDTH - BORDER:
            self.positionX -= SCREEN_WIDTH - 2*BORDER
            self.positinitX -= SCREEN_WIDTH - 2*BORDER
        if self.positionY < BORDER:
            self.positionY += SCREEN_HEIGHT -2* BORDER
            self.positinitY += SCREEN_HEIGHT -2* BORDER
        if self.positionY > SCREEN_HEIGHT - BORDER:
            self.positionY -= SCREEN_HEIGHT - 2*BORDER
            self.positinitY -= SCREEN_HEIGHT - 2*BORDER

        # Affichage:
        if i==3:
            self.rect.x = round(self.positionX)
            self.rect.y = round(self.positionY)

# === main ===

# --- init ---

pygame.init()
screen = pygame.display.set_mode(SCREEN_SIZE)

# --- objets ---

oiseau_list = pygame.sprite.Group() # groupe de sprite: réunit tous les oiseaux

# crée les oiseaux à des positions aléatoires:
for i in range(NUM_BOIDS):
    oiseau = Oiseau(randint(BORDER, SCREEN_WIDTH-BORDER), randint(BORDER, SCREEN_HEIGHT-BORDER))
    oiseau_list.add(oiseau)

# --- mainloop ---

clock = pygame.time.Clock()

running = True

# Servent à l'affichage des graphes:
DIST_MOY=[]
VITX_MOY=[]; VITY_MOY=[]
NRJ=[]
E_POT=[]
TRAV=[]
trav=0.

while running:
```

```
    # --- events ---

    for event in pygame.event.get():
        if event.type == pygame.QUIT:  # permet de fermer la fenetre
            running = False
        if event.type == pygame.KEYDOWN: # racourcis clavier
            if event.key == pygame.K_ESCAPE: #echap
                running = False

# ======= RK4 ======= :

    for oiseau in oiseau_list: oiseau.acc=[] #initialisation accélérations

    for i in range(4):
        for oiseau in oiseau_list:

            # crée la liste des oiseaux proches:
            closeBoids=[]
            for otherBoid in oiseau_list:

                if otherBoid == oiseau: continue

                if oiseau.distance(otherBoid) < VISION:
                    closeBoids.append(otherBoid)

            if len(closeBoids)>0:
                oiseau.acc.append(oiseau.acceleration(closeBoids)) # retourne la valeur de l'accélération
            else: oiseau.acc.append((0,0))

        oiseau_list.update(i) # on update tous les paramètres des oiseaux

# ======= Graphes ======= :

    moy=0.
    vitx=0.; vity=0.
    nrj=0.
    e_pot=0.

    for oiseau in oiseau_list:
        closeBoids=[]
        for otherBoid in oiseau_list:
            if otherBoid == oiseau: continue
            distance = oiseau.distance(otherBoid)
            moy+= distance
            if distance < VISION:
                closeBoids.append(otherBoid)
                e_pot+= FORCE*(distance-DISTCRIT)**2/2
            else: e_pot+= FORCE*(VISION-DISTCRIT)**2/2

        vx=oiseau.velocityX; vy=oiseau.velocityY
        vitx+= vx; vity+= vy
        nrj+= vx**2 + vy**2
        trav+=2*FROTT*(vx*(oiseau.positionX-oiseau.positinitX)+vy*(oiseau.positionY-oiseau.positinitY))

    DIST_MOY.append(moy)
    VITX_MOY.append(vitx); VITY_MOY.append(vity)
    NRJ.append(nrj)
    E_POT.append(e_pot)
    TRAV.append(trav)
```

```
    # --- draws ---

    screen.fill(COLOR1) # fond
    Bord=pygame.draw.rect(screen,COLOR2,pygame.Rect(BORDER,BORDER,SCREEN_WIDTH-2*BORDER,SCREEN_HEIGHT-2*BO

    oiseau_list.draw(screen) # on dessine tout les oiseaux à leur nouvel emplacement sur le screen
    pygame.display.update() # met à jour tout l'écran

    clock.tick(FPS) # vitesse d'affichage
    # clock.tick_busy_loop(FPS)

DIST_MOY=np.array(DIST_MOY)
VITX_MOY=np.array(VITX_MOY); VITY_MOY=np.array(VITY_MOY)
NRJ=np.array(NRJ); E_POT=np.array(E_POT); TRAV=np.array(TRAV)

DIST_MOY/=(NUM_BOIDS-1)*NUM_BOIDS
VITX_MOY/=NUM_BOIDS; VITY_MOY/=NUM_BOIDS

plt.figure(1)
plt.plot(DIST_MOY); plt.title('Distance')

plt.figure(2)
plt.plot(VITX_MOY,label='x'); plt.plot(VITY_MOY,label='y')
plt.legend(); plt.title('Impulsion')

plt.figure(3)
plt.plot(NRJ,'r',label='Cinétique'); plt.plot(E_POT,'b',label='Potentielle')
plt.plot(NRJ+E_POT,'y',label='Mécanique')
plt.plot(NRJ+E_POT+TRAV,'k',label='E + W')
plt.legend(); plt.title('Energie')

# --- the end ---
pygame.quit() #permet de fermer pygame, si je veux par exemple faire un autre programme
```

Modélisation de poissons

On remarque qu'en changeant la couleur de l'arrière plan, nous pouvons aisément parvenir à simuler des poissons au lieu d'oiseaux. La Fig. 14 fait office d'exemple. On pourrait même imaginer comme développement possible de reproduire des troupeaux de buffles en mettant plutôt du jaune en arrière plan.

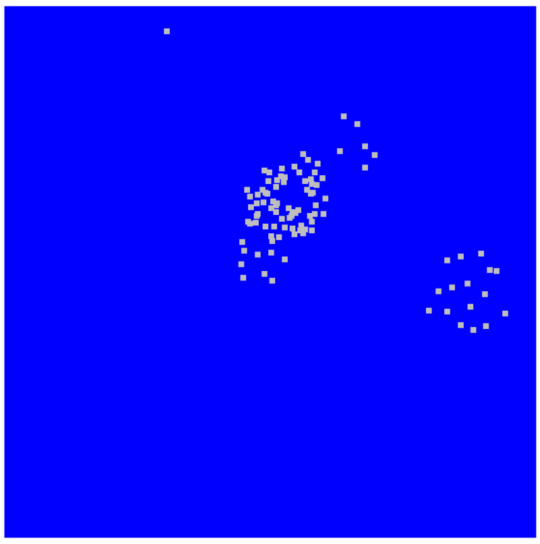


FIGURE 14 – Simulation de poisson à l'aide de la méthode des boids.