

Simulation Multi-agents : Nuées d'oiseaux (Boids)

Boumenad Zakariya
Da Silva Nolan

Date : 15 Décembre 2024



Table des matières

1	Introduction	2
1.1	Objectifs du Projet	2
1.2	Analyse des Problèmes à Traiter	2
1.2.1	Problèmes Techniques	2
1.2.2	Problèmes Conceptuels	2
2	Prise en main, Traitement des bordures et Gestion du hasard	3
2.1	Visualisation	3
2.2	Monde torique	4
2.3	Prise en compte de la seed	5
3	Gestion du voisinage	5
3.1	Détection	5
3.2	Distance torique	7
3.3	Mise à jour de la classe de vecteur	8
4	Règles de déplacement	8
4.1	Groupement/Evitement/Alignement	8
4.2	Modification du Flock et Boid	10
4.3	Restriction de la vitesse	13
4.4	Lecture des nouveaux arguments	14
5	Pour aller plus loin	15
5.1	Arrêt automatique	15
5.2	Force du vent	16
5.3	Cône de vision	16
5.4	Normalisation de l'angle	17
5.5	Rotation maximale	18
5.6	Mettre en place un unique programme	20

1 Introduction

Ce document décrit les étapes et les règles associées à la simulation multi-agents des nuées d'oiseaux.

1.1 Objectifs du Projet

L'objectif principal de ce projet est de simuler le comportement collectif d'une nuée d'oiseaux (ou banc de poissons) à l'aide d'une simulation multi-agents. Chaque entité, appelée *boid*, suit trois règles principales :

1. **Évitement** : Les boids évitent les collisions.
2. **Groupement** : Les boids cherchent à se rapprocher de leurs voisins.
3. **Alignement** : Les boids adaptent leur direction en fonction des autres.

Ces règles sont appliquées dans un espace continu en 2D, avec des bordures gérées de manière torique (un boid qui sort d'une bordure réapparaît à l'opposé).

1.2 Analyse des Problèmes à Traiter

1.2.1 Problèmes Techniques

- **Gestion des bordures** : Implémenter un comportement toriquement cohérent.
- **Détection des voisins** : Identifier efficacement les boids à proximité sans surcharger la simulation.
- **Appliquer les forces** : Implémenter un comportement d'interaction entre les boids

1.2.2 Problèmes Conceptuels

- **Trouver des coefficients d'équilibrage** pertinents pour les forces d'évitement, groupement et alignement afin d'obtenir un comportement réaliste.
- **S'assurer que la simulation reste fluide** avec un nombre élevé de boids.

2 Prise en main, Traitement des bordures et Gestion du hasard

2.1 Visualisation

Le projet est organisé en quatre fichiers principaux, chacun ayant un rôle spécifique et interconnecté pour former le programme complet. L'exécution commence par le fichier **main.py**, qui appelle **flock.py**. Ce dernier, à son tour, fait appel à **boid.py**. Les fichiers **boid.py** et **flock.py** peuvent, selon leurs besoins, utiliser les fonctionnalités définies dans la classe de vecteurs contenue dans **vector.py**.

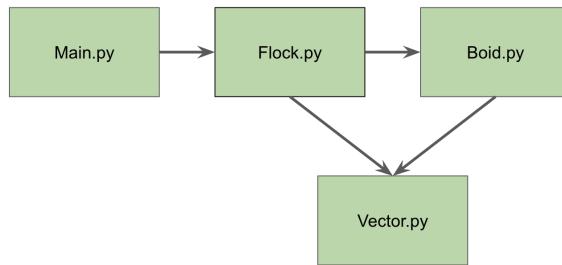


FIGURE 1 – Schéma des dépendances entre les fichiers Python

Rôle de chaque fonction

main.py : Programme de lancement de la simulation permettant de récupérer les éléments fournis par l'utilisateur.

flock.py : Programme de mise en place du terrain contenant la nuée, elle l'initialise et la met à jour par la fonction `update`.

boid.py : Programme de mise en place de la classe des oiseaux, elle les initialise et les met à jour par les fonctions `interact` et `update`.

Il fallait d'abord observer le rendu non modifié de la nuée. Pour ce faire, il fallait, pour une visualisation possible, diviser `dt` par un nombre quelconque pour diminuer la rapidité de la simulation.

2.2 Monde torique

Un monde torique est un espace où les bords sont connectés, ce qui permet à un objet de réapparaître de l'autre côté lorsqu'il les dépasse. La fonction **bounce** gère ce comportement en ajustant la position de l'objet dès qu'il sort des limites définies (largeur et hauteur). Par exemple, si l'objet dépasse la droite, il réapparaît à gauche, et s'il franchit le haut, il revient en bas.

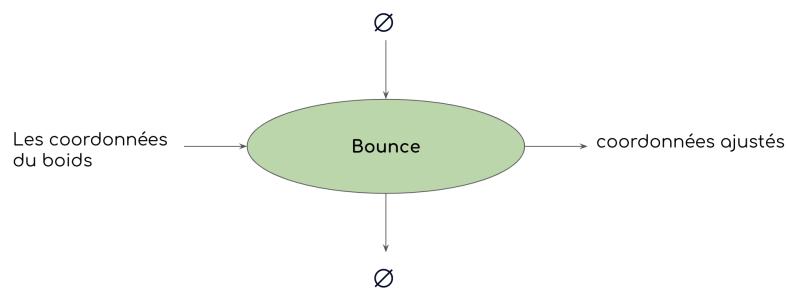


FIGURE 2 – Schéma bulle de la méthode bounce

Paramètres : (x, y) : couple donnant la position du boid en cartésien
(vecteur) $width$: largeur de l'image **(réel)** $height$: hauteur de l'image
(réel)

Résultat : (x, y) : position corrigée par la condition torique.

Fonction *Bounce(self, width, height)* :

Début

Si $x > width$ **Alors**

$x \leftarrow 0$

Fin si

Si $x < 0$ **Alors**

$x \leftarrow width$

Fin si

Si $y > height$ **Alors**

$y \leftarrow 0$

Fin si

Si $y < 0$ **Alors**

$y \leftarrow height$

Fin si

Fin

2.3 Prise en compte de la seed

La deuxième modification la plus simple était **d'implémenter la seed**. Il fallait seulement initialiser la seed donnée de cette façon :

```
random.seed(self.seed)
```

3 Gestion du voisinage

3.1 Détection

La méthode detect identifie les boids voisins d'un boid donné en fonction d'un rayon de détection spécifié.

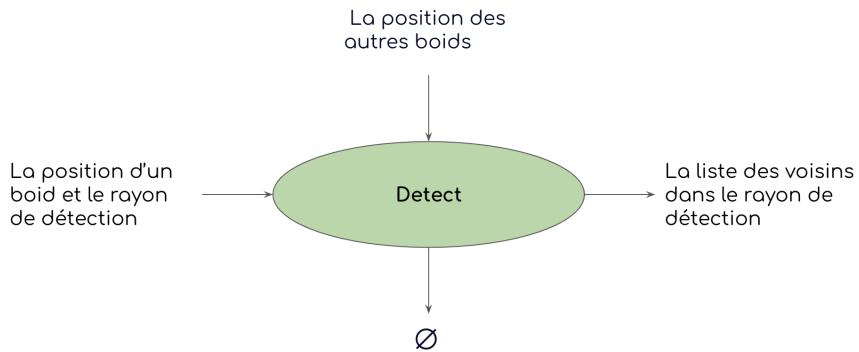


FIGURE 3 – Schéma bulle de la méthode detect

Paramètres :

boid : le **boid** cible

detection_radius : rayon de détection (**réel**)

Variable : *d* (*vecteur*)

Résultat : *neighbour* : **ensemble** des boids voisins détectés

Fonction *detect(boid,detection_radius)*

Début

Initialiser *neighbour* $\leftarrow \emptyset$

Pour chaque *other* dans *self.boids* **faire**

Appeler *d* = *distance_tore(boid, other)*

Si $0 < \text{magnitude}(d) < \text{detection_radius}$ **alors**

Ajouter *other* à *neighbour*

Fin si

Fin pour

Retourner *neighbour*

Fin

La condition **Si** $0 < \text{magnitude}(d)$ est nécessaire pour ne pas avoir le boid observé dans sa propre liste de voisin, ce qui menait à des incohérences.

3.2 Distance torique

La présence du monde torique a obligé à modifier le calcul standard de la distance de telle sorte à prendre en compte les bords de notre simulation.

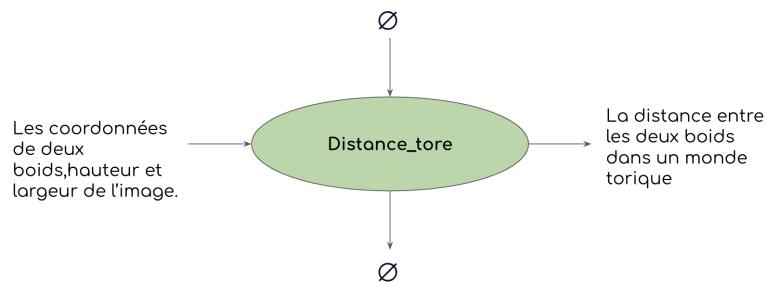


FIGURE 4 – Schéma bulle de la méthode distance torique

Paramètres : other : La position de l'autre boid (**Vecteur** contenant les coordonnées (x,y)). width : La largeur de la zone de simulation (**réel**) height : La hauteur de la zone de simulation (**réel**).

Variable : d (*vecteur*)

Résultat : Vecteur *Distance entre self et other*

Fonction : *Distance torique(self,other,width, height)*

Début

$d \leftarrow other - self$

$x1 \leftarrow d.x$

$x2 \leftarrow width - x1$

$y1 \leftarrow d.y$

$y2 \leftarrow height - y1$

Retourner $(Vector(\min(x1, x2), \min(y1, y2)))$

Fin

3.3 Mise à jour de la classe de vecteur

Quelques fonctions simples ont été ajoutées à la classe vecteur pour ne pas avoir les erreurs courantes du fait de mettre par habitude Vecteur1 - Vecteur2 car la classe de vecteur ne possédait ni de soustraction ni de négation. C'est pourquoi il a fallu ajouter les fonctions ci-dessous pour éviter l'écriture répétée de Vecteur1 + (-1) * Vecteur2

1 : Méthode de soustraction

Paramètres :

self : Vecteur courant

other : Vecteur à soustraire

Résultat : Vecteur différence entre *self* et *other*

Fonction `__sub__(self, other)`

Début

Calculer *result* \leftarrow *self* + (-*other*)

Retourner *result*

Fin

2 : Méthode de passage à l'opposé

Paramètres : *self* : Vecteur courant

Résultat : L'opposé de *self*

Fonction `__neg__(self, other)`

Début

Calculer *result* \leftarrow -*self*

Retourner *result*

Fin

4 Règles de déplacement

4.1 Groupement/Evitement/Alignement

Au départ, on calculait les forces pour chaque boid avec sa liste de voisins comme sur la formule.

Algorithmes précédents pour Groupement, Évitement et Alignement :

Groupement

Paramètres : *boid* : le **boid** cible *voisins* : **ensemble** des voisins détectés

Résultat : Vecteur de groupement *f_cohesion*

Fonction *cohesion(boid, voisins)*

Début

```
n ← taille(voisins)  
posboid ← boid.position  
sum ← Vector(0, 0)
```

```
Pour chaque other.position dans voisins faire  
    sum ← sum + (other.position – posboid)
```

Fin Pour

```
f_cohesion ←  $\frac{1}{n} \times$  sum  
Retourner f_cohesion
```

Fin

Évitement

Paramètres : *boid* : le **boid** cible *voisins* : **ensemble** des voisins détectés

Résultat : Vecteur d'évitement *f_avoidance*

Fonction *avoidance(boid, voisins)*

Début

```
n ← taille(voisins)  
posboid ← boid.position  
sum ← Vector(0, 0)
```

```
Pour chaque other.position dans voisins faire  
    dist ← other.position – posboid  
    sum ← sum +  $\frac{1}{\|dist\|^2} \times dist$ 
```

Fin Pour

```
f_avoidance ←  $\frac{1}{n} \times$  sum  
Retourner f_avoidance
```

Fin

Alignement

Paramètres : *voisins* : ensemble des voisins détectés

Résultat : Vecteur d'alignement *f_alignment*

Fonction *alignment(voisins)*

Début

n \leftarrow *taille(voisins)*

sum_vel \leftarrow *Vector(0, 0)*

Pour chaque *other.velocity* dans *voisins* **faire**

sum_vel \leftarrow *sum_vel* + *other.velocity*

Fin Pour

f_alignment $\leftarrow \frac{1}{n} \times \text{sum_vel}$

Retourner *f_alignment*

Fin

Ainsi les calculs étaient effectués dans Flock ce qui était une mauvaise initiative. Ceci ne fonctionnait pas, nous avons réalisé plus tard que c'était surtout la valeur des coefficients qui entraînait que les boids ne réagissaient pas entre eux, mais j'avais déjà modifié la forme de l'implémentation des forces pour ne pas faire appel à des fonctions supplémentaires et ajouter directement ce qui était attendu à chaque tour de boucle en le fournissant à la fonction interact.

4.2 Modification du Flock et Boid

Retour en arrière, il a fallu supprimer l'erreur du fait que les différents calculs étaient dans Flock, la simplification aura alors été de mettre l'obtention de l'ensemble des voisins dans Flock et de fournir la distance et la vitesse de ces derniers à la fonction interact de Boid

Algorithme : Mise à jour des boids

Paramètres :

dt : intervalle de temps pour la mise à jour (**réel**)

self : classe des boids

Variable : *neighbour* (*ensemble*), *n* (*entier*), *d* (*vecteur*)

Fonction *update(self, dt)*

Début

Pour chaque *boid* dans *self.boids faire*

neighbour \leftarrow *self.detect(boid, self.detection_radius)*

n \leftarrow *taille(neighbour)*

Pour chaque *other* dans *neighbour faire*

d \leftarrow *boid.position.distance_to(other.position, self.width, self.height)*

boid.interact(d, other.velocity, n, self.coeff, self.optional)

Fin Pour

Fin Pour

Pour chaque *boid* dans *self.boids faire*

boid.bounce(self.width, self.height)

Fin Pour

Pour chaque *boid* dans *self.boids faire*

boid.update(dt, self.optional)

Fin Pour

Fin

Ce choix a été fait pour ne pas rechercher chaque donnée dans la liste de voisins une fois de plus et on effectue le même calcul que précédemment, mais d'une façon plus lisible.

Ainsi la forme de l'interact a été modifié pour correspondre à cela :

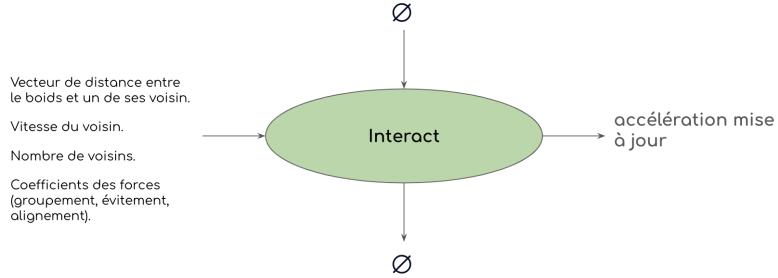


FIGURE 5 – Schéma bulle d'interact

Algorithme : Interaction des forces

Paramètres :

d : vecteur de distance *boid* à *other*

v : vitesse de *other* (vecteur)

n : nombre de voisins détectés (entier)

coeff : coefficients des forces (3-uplet)

Variable : *c_group*, *c_avoid*, *c_align* (3 réels) séparant le 3-uplet *coeff*

Résultat : *self.acceleration* mis à jour

Fonction *interact(d, v, n, coeff, optional)*

Début

Si *n* $\neq 0$ **alors**

self.acceleration \leftarrow *self.acceleration* + *c_group* \times *d* \times $\frac{1}{n}$ (Force de groupement)

 Calculer *d_magni* \leftarrow $\|d\|$

self.acceleration \leftarrow *self.acceleration* - *c_avoid* \times *d* \times $\frac{1}{n \times d_magni^2}$ (Force d'évitement)

self.acceleration \leftarrow *self.acceleration* + *c_align* \times *v* \times $\frac{1}{n}$ (Force d'alignement)

Sinon si

self.acceleration \leftarrow *self.acceleration* + *Vector(random.gauss(0, 0.4), random.gauss(0, 0.4))* (Force aléatoire)

Fin Si

Fin

4.3 Restriction de la vitesse

Une fois que les boids interagissaient entre eux, il fallait les limiter à la vitesse maximale initialisée pour qu'il n'aille pas à une vitesse infiniment grande.

Algorithme : Mise à jour d'un boid

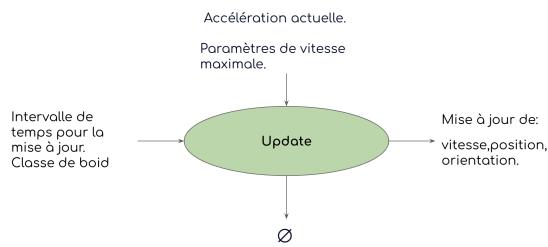


FIGURE 6 – Schéma bulle d'update

Paramètres :

dt : intervalle de temps pour la mise à jour (**réel**)

optional : indicateur pour activer les restrictions d'angle (**réel**)

Variable : *self.position* (*vecteur*), *self.velocity* (*vecteur*),
self.acceleration (*vecteur*), *self.heading* (**réel**)

Résultat : *position, vitesse et orientation mis à jour*

Fonction *update(self, dt)*

Début

self.velocity \leftarrow *self.velocity* + *self.acceleration* \times *dt*

Si $\|self.velocity\| > self.max_speed$ **alors**

self.velocity \leftarrow *Vector*(*self.max_speed*

$\times \cos(self.velocity.angle()), self.max_speed$

$\times \sin(self.velocity.angle())$)

Fin Si

self.position \leftarrow *self.position* + *self.velocity* \times *dt*

self.heading \leftarrow *self.velocity.angle()* + $\pi/2$

self.acceleration \leftarrow *Vector*()

Fin

4.4 Lecture des nouveaux arguments

Pour simplifier les tests nous avions d'abord utilisé des coefficients fixes définis au début du main.py, or il faut pouvoir les modifier à l'appel de la fonction, ceci nécessitait une modification assez simple de la lecture des arguments.

Algorithme : Lecture des arguments du programme

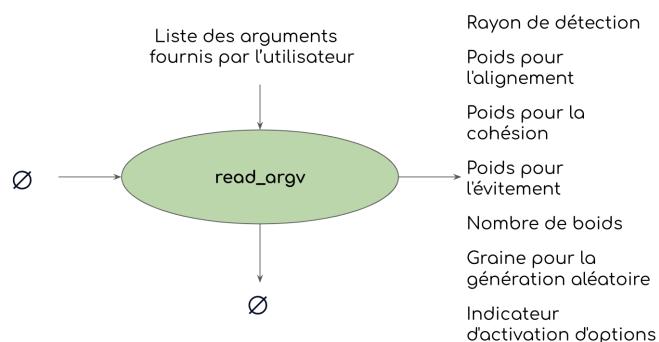


FIGURE 7 – Schéma bulle de `read_argv`

Paramètres : Aucun (les arguments sont récupérés via `sys.argv`).

Résultat : Retourne les valeurs des arguments nécessaires à l'exécution du programme :

`detection_radius, alignment_weight, cohesion_weight, avoidance_weight, num_boids, seed, optional`

Fonction `read_argv()`

Début

Si le nombre d'arguments (`len(sys.argv)`) est inférieur à 7 **alors**

Appeler `error_and_exit("Nombre d'arguments incorrect")` pour afficher une erreur et quitter le programme.

```
detection_radius ← int(sys.argv[1])
alignment_weight ← int(sys.argv[2])
cohesion_weight ← int(sys.argv[3])
avoidance_weight ← int(sys.argv[4])
num_boids ← int(sys.argv[5])
seed ← int(sys.argv[6])
```

Fin Si

Si le nombre d'arguments est exactement 7 **alors**

```
optional ← 0
```

Fin Si

Sinon

```
optional ← int(sys.argv[7])
```

Retourner (`detection_radius, alignment_weight, cohesion_weight, avoidance_weight, num_boids, seed, optional`)

Fin Si

Fin

5 Pour aller plus loin

5.1 Arrêt automatique

L'arrêt automatique après un certain temps n'était pas quelque chose de compliqué à réaliser. Il fallait seulement faire attention à l'endroit où le placer et à la manière de comptabiliser le temps écoulé en secondes. Dans le `for event in run`, cela ne fonctionnait pas. En effet, cette partie du programme correspondait non pas à un événement du programme à chaque tour de boucle, mais à un événement extérieur tel que quitter la page Pygame ou appuyer

sur Échap. Pour ce qui est de la comptabilisation en secondes, cela s'est fait principalement grâce à des tests avec plusieurs print pour vérifier si la valeur était cohérente. Cela a permis de trouver comment modifier le dt pour qu'il devienne le temps écoulé en secondes :

```
temps ecoule += dt * 30 / 1000
```

5.2 Force du vent

Aucun problème là-dessus, nous avons juste pris la liberté de mettre en place une direction du vent aléatoire au début de chaque essai.

Cf Algorithme Interact 4.2

5.3 Cône de vision

Tâche plus difficile à réaliser, pour ce faire, il fallait bien calculer la différence d'angle entre le boid visualisé et son potentiel voisin. Le programme utilise cependant distance, car ça ne change rien sur la direction/angle vers lequel regarde son potentiel voisin, il fallait faire attention à utiliser la valeur absolue pour la vérification et bien faire attention à normaliser entre -pi et pi pour le calcul puisque c'est l'intervalle que nous fournit atan2 dans la fonction angle de Vector.

Paramètres : *boid* : le **boid** cible *detection_radius* : rayon de détection (réel)

Variable : *d* (vecteur), *diff_angle* (réel)

Résultat : *neighbour* : ensemble des boids voisins détectés

Fonction *detect(boid,detection_radius)*

Début :

Initialiser *neighbour* $\leftarrow \emptyset$

Pour chaque *other* dans *self.boids faire*

Appeler *d = distance_tore(boid, other)*

Si $0 < \text{magnitude}(d) < \text{detection_radius}$ **alors**

Calculer *diff_angle* \leftarrow différence d'angle normalisée entre *boid.velocity.angle()* et *d.angle()*

Si *diff_angle < view_angle* **alors**

Ajouter *other* à *neighbour*

Fin si

Fin si

Fin pour

Retourner *neighbour*

Fin

5.4 Normalisation de l'angle

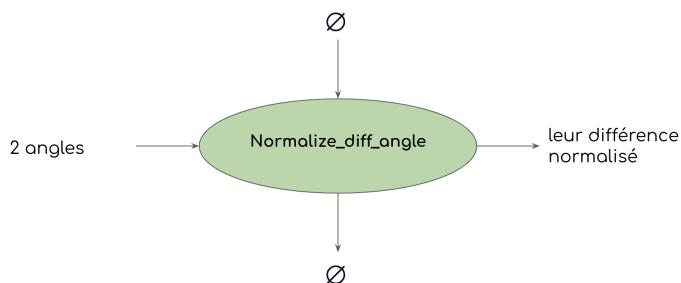


FIGURE 8 – Schéma bulle de la méthode Normalize

Ainsi pour normaliser la différence d'angle, nous avons rajouté une fonction dans Vector :

Algorithme : Normalisation de la différence d'angle

Paramètres :

a : Angle actuel en radians (**réel**)

b : Angle de référence en radians (**réel**)

Variable : *diff_angle* (*réel*)

Résultat : Différence normalisée entre les deux angles, comprise entre $-\pi$ et π

Fonction *normalize_diff_angle(a, b)*

Début

diff_angle $\leftarrow a - b$

normalized_angle $\leftarrow (\text{diff_angle} + \pi) \equiv (2 \times \pi) - \pi$

Retourner *normalized_angle*

Fin

5.5 Rotation maximale

Tâche d'autant plus compliquée, cela réutilise le même principe où nous devons faire attention à l'intervalle de l'angle, mais en plus, nous devons dissocier deux cas entre si le boid se dirige vers la “gauche” ou la “droite” de sa position initiale.

Algorithme : Mise à jour d'un boid (facultatif)

Paramètres : dt : Intervalle de temps (**réel**), $optional$: Option pour la conservation de l'angle (**booléen**), max_angle : Angle maximal permis (**réel**)

Variable : $previous_angle$, $current_angle$, new_angle , $self.heading$ (**réels**), $self.position$, $self.velocity$, $self.acceleration$ (**vecteurs**)

Résultat : Mise à jour de la position, vitesse et accélération du boid courant

Fonction $update(dt, optional)$

Début

Si $optional = 1$ **alors**

$previous_angle \leftarrow self.velocity.angle()$

$self.velocity \leftarrow self.velocity + self.acceleration \times dt$

Fin Si

Si la vitesse dépasse $self.max_speed$ **alors**

$self.velocity \leftarrow Vector(self.max_speed \times \cos(self.velocity.angle()),$

$self.max_speed \times \sin(self.velocity.angle()))$

Fin Si

Si $optional = 1$ **alors**

$current_angle \leftarrow self.velocity.angle()$

$diff_angle \leftarrow normalize_diff_angle(current_angle,$

$previous_angle)$

Si $-diff_angle \geq max_angle$ **alors**

$new_angle \leftarrow previous_angle - max_angle$

$self.velocity \leftarrow Vector(self.velocity.magnitude()$

$\times \cos(new_angle), self.velocity.magnitude() \times \sin(new_angle))$

Sinon si $diff_angle \geq max_angle$ **alors**

$new_angle \leftarrow previous_angle + max_angle$

$self.velocity \leftarrow Vector(self.velocity.magnitude()$

$\times \cos(new_angle), self.velocity.magnitude() \times \sin(new_angle))$

Fin Si

$self.position \leftarrow self.position + self.velocity \times dt$

$self.heading \leftarrow self.velocity.angle() + \pi/2$

$self.acceleration \leftarrow Vector()$

Fin

5.6 Mettre en place un unique programme

C'était la dernière étape de modification, une modification du programme tel que si le dernier coefficient était de valeur 0 on renvoyait la visualisation normale et s'il valait 1 la visualisation facultative. Il a fallu donc rajouter un terme nommé optional dans plusieurs fonctions pour cela. De plus, pour ne pas déranger la demande obligatoire d'un lancement de programme sous la forme demandé pour main.py, nous avons fait en sorte que si on oubliait de donner la valeur 0 ou 1 dans les données, par défaut celle-ci était 0, c'est une modification apportée à read_argv.

Cf Algorithme read_argv 4.4