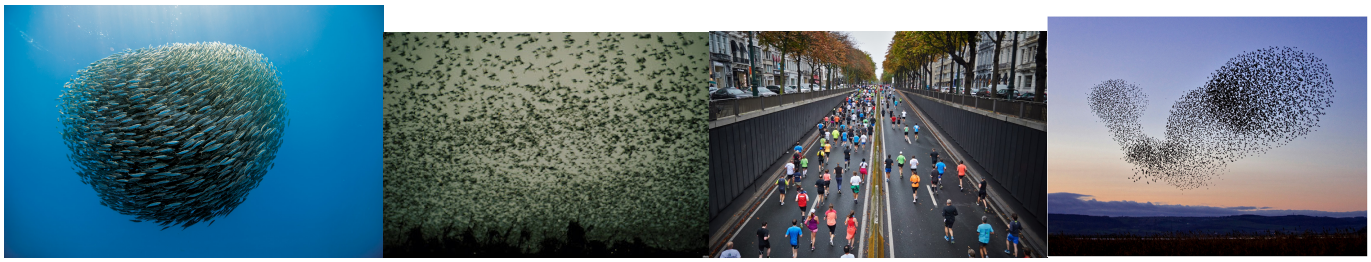
# Mini projet : simulation d'une nuée d'oiseaux (ou d'un banc de poissons, ou d'une foule, d'un essaim d'insectes etc.)



Comme indiqué dans le titre, l'objectif de ce mini-projet est de réaliser une simulation de mouvements de masse. Le type d'algorithme que vous allez coder est utilisé dans le cinéma, pour les mouvements d'oiseaux, d'armées etc. Historiquement la 1ère utilisation au cinéma a été pour des nuées de chauve-souris dans un des Batman des années 90. Il y a des utilisations « non ludiques » de ce type d'algorithmes ceci dit.
Vous allez utiliser le paradigme de la programmation objet pour réaliser ce projet.
Les programmes à compléter sont ici : http://www.maths-info-lycee.fr/programmes/essaims_objet.zip

## 1. Préliminaire : la classe `Vecteur`

Cette classe permet de faire plus simplement des opérations sur des vecteurs du plan. On l'utilisera ensuite pour calculer positions et trajectoires des oiseaux.
*Bien sûr, il existe déjà des bibliothèques puissantes pour traiter les vecteurs (numpy notamment). Mais l'objectif de ce devoir n'est pas de vous apprendre à utiliser des bibliothèques…*
On complétera le fichier `classe_vecteur.py` pour :

### a. Écrire les méthodes de la classe

Enlever les instructions `pass` au fur et à mesure. Elles ne font rien, et permettent d'éviter des erreurs de syntaxe qui seraient sinon signalées.
Sont à faire ou à compléter :

- `est_nul()`
- `vect_nul()`
- `norme()`
- `diff(v)`
- `oppose()`
- `prodk(k)`
- `affectation(v)`

Vous disposez d'une batterie de tests, qui volontairement ne sont pas sous forme d'assertions : cela vous permettra de faire quelques calculs de révision sur les vecteurs. Vous pouvez également tester les angles si vous le souhaitez.

### b. Une remarque importante pour la suite

Avec cette classe, testez les deux blocs instructions suivantes :

```
u = Vecteur(1, 2)
v = Vecteur(3, 4)
u.somme(v).prodk(5)
```

Et

```
u = Vecteur(1, 2)
v = Vecteur(3, 4)
u.somme(v)
u.prodk(5)
print(u)
```

Comment s'explique ce comportement ? Réponse : la méthode `u.somme(v)` ne renvoie rien, elle modifie `u`. On ne peut donc pas appliquer une autre méthode sur un résultat qui n'existe pas !

## 2. La classe `Animal`

Cette classe représente un seul animal.
Elle est à compléter dans le fichier `classe_animal.py`.
Vous remarquerez qu'on importe dans ce programme la classe `Vecteur`, sous la syntaxe `from classe_vecteur import *`, en omettant le `.py` du fichier `classe_vecteur.py`. Ce fichier doit être dans le même dossier que `classe_nuee.py`.

### a. Compléter le constructeur de la classe.

Lire les spécifications de la classe pour y voir les attributs déjà crées. Il en reste à compléter :
La position initiale est un vecteur, dont les coordonnées aléatoires (en pixels) sont limitées par la taille de l'univers (en pixels également), à laquelle on ajoute. Prévoyez une marge de 10 pixels au minimum, plus la taille de l'animal (également en pixels). Si l'animal est au point *A*, ce vecteur est égal à $\overrightarrow{OA}$.
La vitesse est également un vecteur aléatoire, qui ne doit pas être nul. Chaque coordonnée de la vitesse est un nombre flottant compris entre -1 et 1, sachant que les deux coordonnées ne peuvent pas être nulles en même temps. La vitesse générée initialement est transformée de manière à avoir la norme de la vitesse initiale, on calcule $\vec{v} = \vec{v}_{\text{générée aléatoirement}} \times \frac{v\_\text{init}}{||\vec{v}||}$

Remarques :

- On rappelle que `random.random()` renvoie un nombre flottant aléatoire entre 0 et 1.
- Vous avez le droit de demander au professeur l'astuce qui permet de passer de $[0;1]$ à $[-1;1]$ – au risque de subir les moqueries de vos camarades qui auront trouvé comment faire –.
- Cette classe contient des attributs de classe. Ces attributs sont identiques pour toutes les instances de la classe, on ne les modifie pas. Ils sont déclarés avant le constructeur.
- Vérifier que les attributs suivants ont bien les valeurs indiquées :

```
v_max = 4              # tester avec 3
v_init = 2             # tester avec 6
force_max = 0.5        # tester avec 0.5
self.perception = [25, 50, 100]
```

### b. Les méthodes : faire se déplacer un animal.

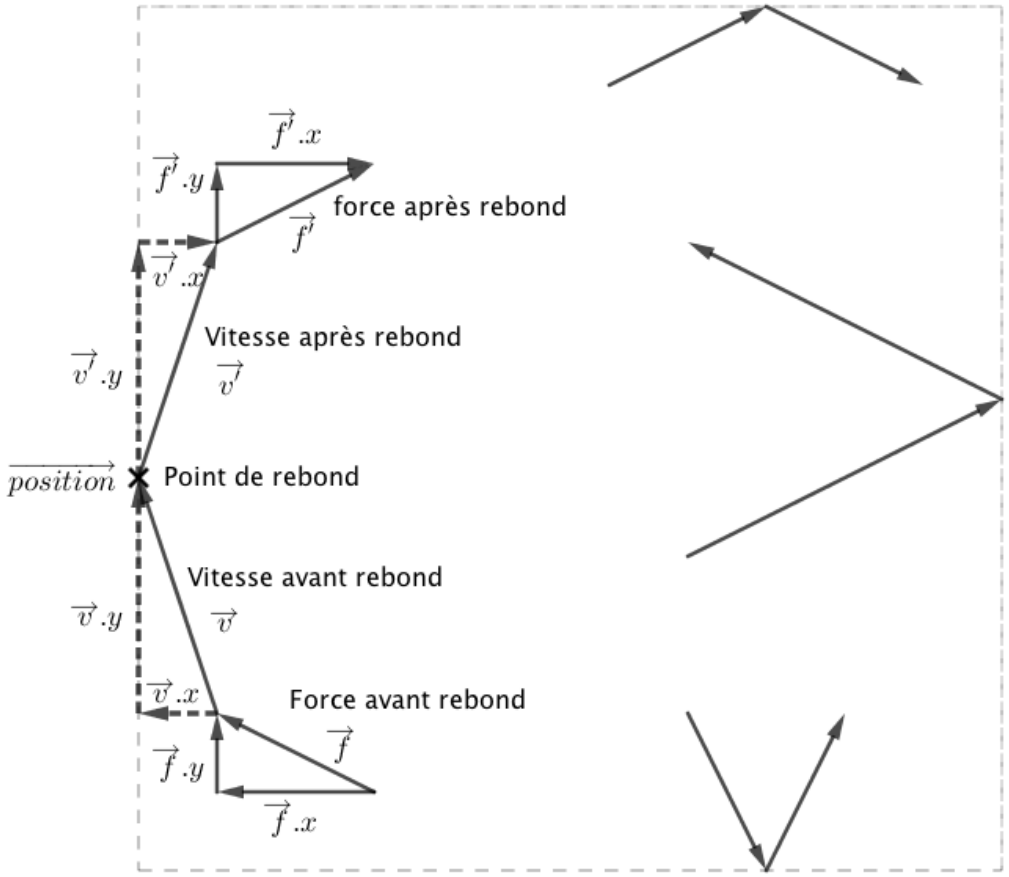
i. L'explication théorique.
On raisonne de manière discrète, par modification de la position à chaque tick d'horloge.
Un animal va passer de la position $A_n$, à l'instant *n,* à la position $A_{n+1}$, à l'instant *n* + 1, grâce à sa vitesse $\vec{v}_n$ : on a $\overrightarrow{OA}_{n+1} = \overrightarrow{OA}_n + \vec{v}_n$.
De même, la vitesse est modifiée à chaque tick par la force –l'accélération[1]– qui lui est appliquée. On a $\vec{v}_{n+1} = \vec{v}_n + \vec{a}_n$.

ii. La programmation : mise à jour de la position.
Ecrire la méthode `maj_position()`, qui :

- applique les règles expliquées ci-dessus au i ;
- vérifie que la vitesse n'est pas supérieure à la vitesse maximale autorisée. Si c'est le cas, on limitera cette vitesse en calculant et affectant : $\vec{v} \leftarrow \vec{v} \times \frac{v\_\max}{||\vec{v}||}$ ;

[1] La force appliquée sur un système correspond à un vecteur d'accélération appliquée à ce système, pour simplifier.

- vérifie que la vitesse n'est pas inférieure à la vitesse intiale. Si c'est le cas, on augmentera cette vitesse en calculant et affectant : $\vec{v} \leftarrow \vec{v} \times \frac{v\_\text{init}}{\|\vec{v}\|}$ ;
-
- dans un deuxième temps (éventuellement après avoir fait un premier test du mouvement à la question suivante iii), on vérifiera que l'animal ne sort pas de l' « univers », mais qu'il rebondit sur les côtés. Les attributs `self.l_univers` et `self.h_univers` représentent la largeur (en *x*) et la hauteur (en *y*) de l'univers.
  Le schéma en page suivante est utile pour comprendre ce qui se passe lors d'un rebond.
- On peut prendre une marge de 50 pixels de chaque côté pour les rebonds.




Remarque : après avoir fait se déplacer l'animal, on peut, si on trouve que rester proche de la vitesse initiale est mieux, multiplier la vitesse courante par 0,9 si elle est supérieure à la vitesse initiale.

iii. La programmation : test
Utiliser le fichier `gui_animal_aleatoire.py` pour tester que votre animal (ici une mouche ou un moustique…) se déplace correctement, sans sortir de la fenêtre.

iv. La programmation : méthode `force_alea()`
Cette méthode applique une force aléatoire sur l'animal, en modifiant l'attribut self.force. Chaque composante de la force est comprise entre -1 et 1. La force est ensuite maximisée à la force maximale autorisée, en utilisant le même principe que pour la vitesse au ii (décommenter les lignes comme précisées dans le code).
Tester à nouveau le déplacement de l'animal avec `gui_animal_aleatoire.py` p
Remarque : on ne se servira plus de cette méthode après le 3d.

v. La programmation : méthode `distance(autre)`

A programmer. Se fait en une ligne –rappelez-vous knn–. Il est déconseillé d'utiliser la classe Vecteur qui ici rend les choses plus compliquées.

## 3. La classe Nuee

Cette classe représente un groupe d'animaux. Elle se trouve dans `classe_nuee.py`.

### a. Compléter le constructeur de la classe.

Créer l'essaim, qui est une liste d'objets de type `Animal` de longueur `nombre` (passé en paramètre du constructeur) et les attributs `self.l_univers`, `self.h_univers`.

### b. Les méthodes : faire se déplacer un essaim d'animaux.

Compléter la méthode `mouvement()`, qui met à jour la position de tous les animaux de la nuée en parcourant l'attribut `essaim` de la `Nuee`.

Pour tester, utiliser le fichier `gui_nuee.py`. On peut dans un premier temps faire un test avec la méthode `force_alea()` de la classe `Animal`, qui sera appelée dans la boucle de parcours de l'`essaim`.

Dans un deuxième temps, ce sera la méthode `regles(animal)` qui sera appelée en début de cette fonction, pour calculer au préalable la force appliquée sur chacun des animaux.

### c. Les méthodes : trouver les voisins qui influent sur le mouvement d'un animal.

La méthode `voisins()` renvoie trois listes `vois_sep`, `vois_align`, et `vois_coh`. Ce sont des listes de listes. En indice *i* on a un animal, en indice *j*, on a un booléen qui précise si l'animal *j* influe sur l'animal *i*, respectivement à distance de séparation, d'alignement et de cohésion (cf. ci-après pour les explications correspondantes)

Compléter les trois matrices en même temps pour des raisons d'efficacité de temps de calcul.

Un animal influe sur un autre si sa distance est inférieure à la distance de perception correspondante.

*Complément* : les oiseaux, poissons, etc. grégaires ont en général un champ de vision important, mais tout de même pas à 360 degrés. On peut limiter les voisins dans un angle de 300 degrés par rapport à la direction de l'animal (+/- 150 de chaque côté). Pour cela on utilise la méthode `angle(v)` de la classe vecteur.

Vous pouvez tester les voisins en décommentant les lignes idoines dans `gui_nuee.py`. Il y a trois blocs de lignes à décommenter. Vous verrez alors apparaître les trois distances sous forme de cercle sur l'animal 0, ainsi que ses voisins sous différentes couleurs.

### d. Les méthodes : règles de déplacement.

Les animaux sont soumis à trois règles, qui s'appliquent à une distance de plus en plus grande[2] :

- Les animaux trop proches s'évitent, c'est la règle de séparation.
- Les animaux moyennement proches essaient d'aller dans la même direction, c'est la règle d'alignement.
- Les animaux dans un rayon un peu plus grand essaient de se rapprocher, c'est la règle de cohésion.

Toutes ces règles modifient la force exercée sur l'animal.

i. Règle d'alignement : les animaux essaient d'aller tous dans la même direction.
La méthode `alignement(animal)` renvoie `force_alignement`. Cette force est égale à $\vec{f}_{\text{alignement}} = \left( \frac{1}{\text{nombre de voisins}} \sum_{\text{voisin}} \vec{v}_{\text{voisin}} \right) - \vec{v}_{\text{animal}}$ . Cette formule signifie que l'on calcule la vitesse moyenne des voisins, qui sont trouvés dans la matrice `vois_align`, à laquelle on soustrait la vitesse de l'animal.

[2] De manière cumulative, c'est à dire qu'un animal très proche d'un autre subira les trois règles.

Coder cette méthode ; on peut appliquer l'algorithme suivant :

```
Fonction alignement(animal, liste_voisins)
  force_alignement ← Vecteur nul
  somme_poids ← 0
  Pour tous les animaux voisins de animal :
       force_alignement ← force_alignement  + vitesse du voisin
       somme_poids ← somme_poids + 1
  Si somme_poids ≠ 0 :
       force_alignement ← force_alignement / somme_poids
       # en fait on multiplie par 1/somme_poids
       force_alignement ← force_alignement – vitesse de l'animal
```

ii. Application des règles

Comme écrit ci-dessus au 3b, commenter la ligne qui appelle la méthode `animal.force_alea()` dans la méthode `mouvement()` de la classe `Nuee`.

On va remplacer cet appel par un appel à la méthode `regles()`. L'appel à `regles()` se fera avant de parcourir les animaux pour mettre à jour leur position.s

La méthode `regles` :

- appelle la méthode `voisins()` pour récupérer les trois matrices de voisinage ;
- parcourt l'essaim, et pour chaque animal :
  - calcule les trois forces à appliquer sur l'animal en question. Chacune des trois forces sera normée, cf. algorithme donné pour la règle d'alignement ;
  - les multiplie par un coefficient que l'on précisera ;
  - les ajoute à l'attribut force de l'animal après les avoir éventuellement limitées à la vitesse maximale.

Modifier le code de la méthode `mouvement()`, et programmer `regles()` de manière à appliquer la force d'alignement. On mettra un coefficient 1/8 sur cette force.

Vous pouvez tester avec `gui_nuee.py`.

iii. Règle de séparation : les animaux essaient de ne pas se rentrer dedans.

La méthode `separation(animal)` renvoie `force_separation`. Cette force est égale à $\vec{f}_{\text{séparation}} = \sum_{\text{voisin}} \overrightarrow{A_{\text{voisin}}A}$ . Cette formule signifie que l'on ajoute tous les vecteurs d'origine un voisin $A_{\text{voisin}}$ , dans la matrice `vois_sep`, et d'extrémité l'animal $A$. On utilisera les positions des animaux pour calculer ces vecteurs.

Coder cette méthode, l'ajouter dans `regles()` avec un coefficient 1/10.

La tester (il est conseillé de la tester seule d'abord, sans `alignement()`).

iv. Règle de cohésion : les animaux essaient de se rapprocher.

La méthode `cohesion(animal)` renvoie `force_cohesion`. Cette force est égale à $\vec{f}_{\text{cohésion}} = \left( \frac{1}{\text{nombre de voisins}} \sum_{\text{voisin}} \overrightarrow{Position}_{\text{voisin}} \right) - \overrightarrow{Position}_{\text{animal}}$ . Cette formule signifie que l'on calcule la position moyenne des voisins, qui sont trouvés dans la matrice `vois_coh`, à laquelle on soustrait la vitesse de l'animal.

Coder cette méthode, la tester avec un coefficient 1/100.

v. Conclusion et règles supplémentaires.

Il est très difficile de régler les différents coefficients sur les forces, et autres paramètres (vitesses maximale et initiale entre autres, distances de voisinage) afin d'avoir un comportement un tant soit peu réaliste. Vous pouvez faire vos propres essais. Dans la littérature, j'ai trouvé (entre autres) pour les forces d'alignement, de séparation et de cohésion de (1, 3/2, 1) à (1/8, 1/10, 1/100). Au mieux j'obtiens un comportement de type nuée d'oiseaux pas très loin du

réaliste, plutôt que banc de poissons extrêmement en cohésion. Il est à noter que dans la nature les animaux disposent de sens supplémentaires pour réagir extrêmement vite. Par exemple les poissons ont une ligne latérale, qui leur permet de ressentir immédiatement les changements de pression, de vibration et de vitesse de l'eau sur les côtés de leur corps.

Voir le début de la méthode `regles(animal)` pour une règle supplémentaire : une force centripète avec un coefficient 1/700. Elle fait tournoyer les animaux.

Plus intéressant, mais long à programmer, est la création d'un prédateur, qui se dirige sur l'animal le plus proche, et que les animaux ordinaires fuient. Cf. ci-après pour quelques indications sur cette classe, et les modifications que cela entraîne sur le programme.

**e. Une question importante**

Pourquoi la méthode `force_alea()` est-elle dans la classe `Animal`, alors que les méthodes `règles`, `alignement`, etc. sont dans la classe `Nuee` ?

## 4. La classe `Predateur`

*Partie totalement facultative !*

La création de cette classe est l'occasion d'appliquer la notion d'héritage. En effet l'objet `Predateur` est un type particulier `d'Animal`.

**a. Exploration du code du constructeur de la classe.**

```
0. import classe_animal as a

1. class Predateur(a.Animal):
2.     v_max = 5
3.     v_init = 1
4.     force_max = 1.5
5.     max_vit_max = 75
6.     max_vol_m_dir = 200

7.     def __init__(self, l_univers, h_univers):
8.         super().__init__(l_univers, h_univers)
9.         self.taille = 4
10.        self.duree_vit_max = 0
11.        self.en_chasse = False
12.        self.proie = None
13.        self.t_entre_chasse = randint(500, 1000)
           # randint(100, 300) pour les tests de chasse
14.        self.duree_plane = 0
15.        self.duree_vol_m_dir = 0
```

Ligne 1 : signifie que la classe `Predateur` hérite de la classe `Animal`, importée comme `a` en ligne 0

Lignes 2 à 6 : on redéfinit certains attributs de classe avec des valeurs différentes entre `Animal` et `Predateur`, mais également on ajoute d'autres variables de classe répondant à des besoins particuliers.

Ligne 8 : dans le constructeur de `Predateur`, on fait appel au constructeur de la super classe, c'est à dire la classe `Animal`, parent de `Predateur`. La classe `Predateur` aura donc tous les attributs et toutes les méthodes de `Animal`. Notamment, la méthode `maj_position` est indispensable et ne sera pas à réécrire dans `Predateur`.

Lignes 9 à 14 : on donne certaines valeurs particulières à des attributs déjà existants, et on crée les attributs nécessaires pour faire tourner le prédateur.

Le prédateur chasse avec un intervalle de temps aléatoire `self.t_entre_chasse`. Dans la nature, les observations sur un même rapace dans la journée donnent un intervalle de temps de quelques minutes à plusieurs heures. Cette imprévisibilité fait que les proies ne peuvent pas adopter un schéma de comportement simple. Comme par ailleurs les proies ne peuvent pas rester en permanence sur le qui-vive, cela augmente les chances de succès des prédateurs.
Lorsqu'il ne chasse pas, le prédateur plane, avec deux compteurs de temps. Le premier, `self.duree_plane`, sert à déclencher la chasse lorsqu'il atteint `self.t_entre_chasse`. Le deuxième, `self.duree_vol_m_dir`, est facultatif et sert à faire tourner doucement le prédateur.

**b. Les méthodes : planer entre deux chasses.**

La méthode plane(self) incrémente les deux compteurs `self.duree_plane` et `self.duree_vol_m_dir = 0`.
Si l'on souhaite faire tourner le prédateur, on peut, lorsque le compteur `self.duree_vol_m_dir` atteint `max_vol_m_dir`, modifier la force qui s'exerce sur le prédateur. On lui affecte un vecteur aléatoire non nul, de coordonnées comprises entre -1/50 et 1/50 de la force maximale (entre -1 et 1, avec `random()*2 - 1` puis multiplié par force_max/50). ensuite, on applique cette force pendant 100 ticks (jusqu'à `self.max_vol_m_dir + 100`), puis on remet cette force à 0 ainsi que le compteur de temps de vol dans la même direction.
Ensuite on met à jour la position avec la méthode de la classe parent : on appelle simplement `self.maj_position()`, puisque la classe `Predateur` dispose des méthodes de la classe `Animal`.
Vérifier que la vitesse ne dépasse pas la vitesse initiale.

**c. Les méthodes : trouver une proie.**

La proie sera l'animal de la nuée le plus proche. La méthode `trouve_proie(self, nuee)` se résume à une recherche de minimum pour mettre à jour l'attribut `self.proie`.
Pour initialiser la distance minimale, on peut importer « l'infini ». Pour une fois, le nom est trompeur et il vaut mieux l'écrire sous la forme `from math import inf as infini`.

**d. Les méthodes : chasser.**

La méthode chasse(self) teste d'abord si la chasse est finie, à l'aide du compteur de temps `self.duree_vit_max`. En effet un prédateur ne peut pas aller longtemps à sa vitesse maximale (un guépard met –je crois– plusieurs heures à se reposer entre deux chasses). Si ce compteur dépasse `self.max_vit_max` , alors on remet les attributs `en_chasse` à `False`, `proie` à `None`, `duree_plane` à `0`, `duree_vit_max` à `0`, et on réinitialise aléatoirement le temps entre deux chasses.
Sinon, la chasse peut être à son début (il n'y a pas de proie, ce qui équivaut à `en_chasse` est Faux). Dans ce cas on appelle `self.trouve_proie()` et on met `en_chasse` à Vrai.

Quand il y a chasse, la force exercée sur le prédateur est simplement le vecteur $\overrightarrow{PA}$, où $P$ est le prédateur et $A$ la proie. Cette force est mise au maximum, ainsi que la vitesse résultante une fois appliquée la force. On fait ensuite appel à `self.maj_position()`.
Ne pas oublier d'incrémenter le compteur `self.duree_vit_max`.

On peut éventuellement manger la proie si les positions prédateur/proie sont suffisamment proches. ± 1 pixel sur chaque coordonnée donne une chasse un peu trop efficace, l'égalité exacte semble donner une chasse impossible.
Dans ce cas, on récupère l'indice de la proie avec la méthode `index()`, et on la supprime dans `nuee.essaim`, en utilisant la méthode `nuee.essaim.remove(self.proie)` qui permet de supprimer un élément non pas suivant son index (cas de `pop` ou `del`), mais avec l'élément directement. On arrête la chasse aussi.
On renvoie :

- un booléen `True` qui indique ce qui indique que la proie a été capturée ;

- l'indice de la proie pour supprimer le sprite correspondant ;
- l'abscisse de la proie ;
- l'ordonnée de la proie (coordonnées à sauver avant `remove`).

Pour des raisons de code propre, renvoyer `False None None None` quand la proie n'est pas capturée.

Remarques :
- il y a un doublon volontaire sur la variable booléenne `en_chasse`, qui est égale en fait à `self.proie is not None`. Mais le nom `en_chasse` est bien plus explicite et permet une meilleure compréhension/maintenance/modification du programme
- il n'y a pas d'autres remarques finalement.

**e. Modifications induites sur la classe Nuee et sur la classe Animal.**

i. `Animal, maj_position`

On passe en paramètre le prédateur. S'il est en chasse, c'est mode panique pour tout le monde, ou bien si la distance de l'animal au prédateur est inférieure à la perception pour l'alignement (ce qui correspond au choix moyen entre cohésion et séparation), alors on norme la vitesse au maximum :

$\vec{v} \leftarrow \vec{v} \times \frac{v\_\max}{\|\vec{v}\|}$ .

ii. `Nuee, mouvement`

Modifier l'appel à maj_position pour passer le prédateur en paramètre.

iii. `Nuee, regle`

La méthode `regle` admet également comme paramètre le prédateur. Les règles deviennent :
- si le prédateur est en chasse et qu'il est à distance de perception maximale –la plus grande de la liste perception–, on appliquera une seule force, celle de fuite, au maximum possible ;
- sinon si le prédateur est à une distance inférieure à la distance moyenne de perception, on appliquera une seule force de séparation du prédateur (méthode `separation_p` ci-dessous), au maximum possible. On peut éventuellement reprendre la force de fuite précédente.
- sinon on applique les règles normales

iv. `Nuee, fuite`

La méthode `fuite` va, comme son nom l'indique, générer la force de fuite à appliquer dans les règles.

En version simple et assez efficace, la force de fuite est égale au vecteur $\overrightarrow{PA}$ normalisé (de norme 1, éventuellement de norme `self.force_max`, cf. ci-dessus iii), où $P$ est le prédateur et $A$ la proie.

En version plus sophistiquée, on peu essayer de rajouter un angle entre 30 et 90 degrés, soit en positif soit en négatif. Cela correspond à un comportement plus réaliste : les prédateurs étant un peu moins mobiles à cause de leur vitesse, les proies ne fuient pas dans l'axe mais en tournant. Ceci dit, c'est assez difficile au niveau mathématique. Il faut d'abord normaliser le vecteur $\overrightarrow{PA}$. Ses coordonnées correspondent alors au cosinus et sinus de l'angle de la trajectoire. Pour ajouter un autre angle, on applique les formules $\cos(a+b) = \cos a \cos b - \sin a \sin b$ et $\sin(a+b) = \sin a \cos b + \cos a \sin b$ . Et pour conclure, le résultat testé ne donne pas une simulation plus satisfaisante…

v. `Nuee, separation_p`

Même principe que la force de fuite. D'un point de vue assez personnel, il me paraît plus clair et plus réaliste d'avoir néanmoins deux méthodes différentes, puisque ce sont deux comportements différents que l'on peut affiner.

**5. Modifications induites sur l'interface graphique**

Tout est fait dans `nuee_total.py` … Vous pouvez regarder la gestion des sprites de l'essaim, dans le canevas lors de la chasse.