CentraleSupélec

Projet Wolf Sheep Predation Model

Tanguy Blervacque

Côme Stephant

Paris-Saclay,

Mars 2023

Contents

1 Introduction

L'objectif de ce projet est d'explorer la stabilité des écosystèmes prédateurs-proies. Un tel système est dit instable s'il tend à entraîner l'extinction d'une ou de plusieurs espèces impliquées. En revanche, un système est stable s'il tend à se maintenir dans le temps, malgré les fluctuations de la taille des populations. Ici, nous allons modéliser via la librairie mesa un écosystème Loups - Moutons - Herbe dans lequel ces trois types d'agents se déplacent dans un environnement fixe. Les loups mangent les moutons, les moutons mangent l'herbe, et l'herbe pousse au file du temps.

2 Description du code

Le dossier est composé d'un README, d'un fichier requirements.txt, d'un fichier run.py pour faire tourner le modèle et d'un dossier prey_predator où se trouvent tous nos fichiers de code. Les consignes pour lancer le modèle sont dans le README.

3 Explication du code

3.1 model.py

Pour bien définir les attributs initiaux du modèle, il faut d'abord réfléchir au comportement que l'on cherche à obtenir dans notre environnement. Un environnement idéal serait tel que les trois agents wolf, sheep et grasspatch évoluent sur le long terme sans qu'un type d'agent disapraissent définitivement.

3.1.1 loss_from_movement

Dans le code proposé initialement, il n'y a aucun attribut définissant une façon de faire mourir les loups. Or, pour obtenir un environnement comme décrit au dessus, il est nécessaire de pouvoir faire mourir les loups pour ne pas que leur population grandisse indéfiniment et mange tous les moutons. Nous avons donc intégré des attributs loss_from_movement qui définissent l'énergie perdue par un agent lorsqu'il se déplace. Ainsi, dans la méthode random_move de la classe Wolf et Sheep, on réduit l'énergie de l'agent à partir de la valeur de loss_from_movement définie en paramètre du modèle. Si l'énergie de l'agent (Wolf ou Sheep) tombe à 0, l'agent meurt (il est retiré de l'environnement avec la méthode remove du modèle).

3.1.2 current_unique_id

Lors de la création d'un agent, on lui attribue un unique_id. Par soucis de simplicité au lieu de supprimer l'identifiant d'un agent que l'on supprime, et donc de laisser la possibilité à un nouvel agent de récupérer cet id, nous avons préféré définir un paramètre current_unique_id en entrée du modèle, qui sera incrémenté de 1 à chaque création d'un nouvel agent. Ainsi, chaque nouvel agent créé aura un nouvel id unique et on est sûr de ne pas avoir de conflit d'id. Bien sûr, cela constitue une limite du modèle car ce n'est pas optimal en terme de mémoire et de calcul d'incrémenter indéfiniment cet attribut si l'on fait tourner longtemps le modèle. Malgré tout, dans notre cas, ça ne devrait pas poser problème.

3.1.3 graphiques additionnels

Nous avons ajouté au DataCollector l'énergie moyenne des moutons, l'énergie moyenne des loups, et la quantité totale d'herbe par step, pour les afficher à partir du ChartModule et obtenir plus d'informations en temps réel.

3.2 agent.py

Nous avons implémenté trois classes distinctes d'agents : Grass, Sheep et Wolf. Les actions de Grasspatch ne dépendant d'aucun autre agent, nous avons commencé par coder cette classe, puis Sheep, dont les actions influent sur l'agent Grasspatch, puis Wolf.

3.2.1 Création d'un nouvel agent

La création d'un agent se fait par le biais de la méthode reproduce dans la classe Wolf ou Sheep. Il est important de noter qu'au lieu de créer le nouvel agent directement au sein de cette méthode et donc au sein de la classe agent, la méthode reproduce fait appel à une méthode new_wolf (respectivement new_sheep) du modèle qui crée un nouveau loup (resp. mouton). Ceci permet d'éviter que des agents "communiquent" directement entre eux sans passer par le modèle.

4 Scénarios

Nous avons ajouté des sliders dans la visualisation afin de faciliter au maximum le test des différents scénarios. Avec nos paramètres, nous avons identifié un scénario stable qui est le suivant :

- $initial\_sheep = 100$
- $initial\_wolf = 50$

- $sheep\_reproduce = 0.1$
- $wolf\_reproduce = 0.05$
- $wolf\_gain\_from\_food = 10$
- $grass\_regrowth\_time = 4$
- $sheep\_gain\_from\_fodd = 4$
- $wolf\_loss\_from\_movement = 4$
- $sheep\_loss\_from\_movement = 2$

Ce scénario est assez représentatif de la réalité. À l'équilibre, il y a beaucoup de moutons, qui se reproduisent vite et meurent vite car leur nourriture donne moins d'énergie. À l'inverse, il y a moins de loups, ils se reproduisent plus lentement mais recoivent plus d'énergie lorsqu'ils mangent, et comme les moutons sont abondants ils parviennent à survivre.

Il est intéressant de remarquer que dans ce scénario stable, il y a en réalité un léger cycle causé par la relation entre moutons et herbe. En effet, lorsqu'il y a un peu moins de moutons, l'herbe est plus abondante ce qui cause une augmentation du nombre de moutons (et donc également du nombre de loups). Mais lorsqu'il y a trop de moutons, il manque d'herbe et donc la population de moutons baisse à nouveau.

Si vous souhaitez étudier les différents types de divergences possible, vous pouvez partir de ce scénario stable et observer l'influence de chaque paramètre sur le résultat final.