Auteurs : DUMAS Valentin et SARRAMEA Camille

PPC : The Circle of Life

I. Introduction

Ce projet a pour but de modéliser un écosystème dynamique où l'équilibre naturel repose sur l'interaction entre prédateurs, proies et ressources végétales. L'objectif est de simuler le cycle de la vie : les proies consomment l'herbe pour survivre, tandis que les prédateurs chassent les proies pour maintenir leur propre énergie. Chaque individu est autonome, gérant ses besoins vitaux comme l'alimentation, la reproduction et faisant face à la mort. Le système permet d'observer en temps réel l'évolution de ces populations face à des aléas climatiques comme la sécheresse ou des crises sanitaires telles que les épidémies. Cette simulation offre ainsi une base de laboratoire virtuel pour explorer le comportement de la biodiversité face aux changements environnementaux.

II. Conception et choix techniques

Notre projet s'appuie sur une organisation modulaire répartie en cinq fichiers distincts afin de garantir une séparation claire des responsabilités entre chaque composant de la simulation. Nous avons conçu quatre types de processus indépendants que sont l'affichage (*display_process2.py*), l'environnement (*env_process.py*), les prédateurs (*predator_process.py*) et les proies (*prey_process.py*), tout en centralisant les variables de contrôle dans une classe de configuration dédiée (*config.py*).

Le fichier display sert de point d'entrée principal à l'application et c'est par celui-ci que l'on doit lancer les simulations. Ce choix structurel permet de capturer les entrées clavier de l'utilisateur de manière fluide, une interaction qui aurait été compromise si l'affichage avait été relégué au rang de processus enfant d'un autre composant.

Concernant les individus de la simulation, nous avons poussé la parallélisation au niveau individuel. Contrairement à une approche où une seule boucle gérait tous les animaux, ici, chaque prédateur et chaque proie est un processus autonome, totalement indépendant de ses congénères. Cela signifie que les fichiers *predator_process.py* et *prey_process.py* servent de modèles pour générer autant de processus qu'il y a d'individus dans l'écosystème. Concrètement, si la simulation compte cinquante proies, cinquante processus distincts s'exécutent simultanément, chacun possédant sa propre boucle de vie, ses propres variables de survie et son propre identifiant. Nous avons d'ailleurs enrichi leur cycle de vie par l'ajout d'un paramètre d'âge qui provoque l'arrêt naturel du processus après un certain temps, apportant ainsi un réalisme biologique supplémentaire par rapport aux consignes de base.

Cette conception fait du processus environnement le point d'ancrage de tout le reste. Puisqu'il occupe une place centrale, il fait le lien entre l'interface de contrôle et tous les individus en circulation. Il centralise la gestion de la reproduction, de la mort et du recensement des populations tout en supervisant l'état des ressources végétales. Le temps y est modélisé par des unités appelées ticks, ce qui permet de synchroniser les actions de tous ces processus animaux de manière cohérente et de stabiliser la vitesse de la simulation pour l'observateur. En plus des sécheresses, nous avons pris l'initiative d'intégrer un système d'épidémies. Celles-ci agissent comme des crises sanitaires temporaires affectant les proies et les prédateurs, offrant ainsi de nouvelles perspectives d'étude sur la résilience de l'écosystème.

Un autre point important de notre modélisation concerne la gestion de la ressource primaire qu'est l'herbe. Contrairement aux animaux qui sont des processus actifs, l'herbe est traitée comme une ressource globale gérée par le processus environnement. Nous avons implémenté un taux de croissance constant en période normale, mais ce paramètre devient dynamique lors des crises. Par exemple, lors d'une sécheresse, nous ne nous contentons pas d'arrêter la croissance ; nous avons programmé une décroissance de la ressource, ce qui simule l'assèchement des sols et peut rendre plus compliquée l'alimentation des proies.

De plus, l'aspect aléatoire a été pris en compte pour que la simulation ne soit pas une simple boucle répétitive. Qu'il s'agisse de la probabilité de capture d'une proie par un prédateur ou de la réussite d'une reproduction, l'introduction de facteurs de chance permet de simuler l'imprévisibilité de la nature. Ainsi, chaque lancement de notre programme peut produire un équilibre biologique différent (parfois même de manière flagrante ou une extinction des espèces avait lieu avec une différence de 500 ticks en partant des mêmes conditions de départ).

Nous avons également porté une attention particulière à la robustesse de la fermeture du système. Nous avons ainsi conçu une procédure de sortie où le processus d'affichage envoie un signal d'arrêt général, relayé par l'environnement à tous les individus via un indicateur de fermeture partagé. Cette structure garantit que chaque entité libère ses ressources et ferme ses connexions proprement, assurant ainsi la stabilité de la machine hôte même après plusieurs cycles de simulation et permettant de libérer de l'espace en mémoire.

Enfin, pour l'interface utilisateur, nous avons privilégié une approche textuelle soignée au sein du terminal plutôt qu'une interface graphique complexe que nous ne serions pas capables de réaliser seuls. Nous avons construit un affichage lisible et ergonomique en intégrant manuellement des emojis pour identifier les différentes entités. Cette solution permet d'obtenir un rendu visuel clair et immédiat des statistiques de survie tout en garantissant la robustesse technique du programme.

III. Architecture et protocoles d'échange

L'architecture de notre simulation est décrite dans la Figure 1. Elle repose sur un système de communication hybride qui exploite les différentes fonctionnalités offertes par la bibliothèque multiprocessing de Python afin de garantir la fluidité des échanges entre les processus. Nous avons suivi les recommandations indiquées dans les consignes du projet pour la mise en place des protocoles d'échanges, en adaptant chaque outil au besoin spécifique du flux d'information concerné. La mémoire partagée constitue le socle de l'état global de la simulation, permettant aux processus des prédateurs et des proies de consulter instantanément la disponibilité des ressources comme le nombre de proies ou la quantité d'herbe avant de tenter de se nourrir. Cette structure permet également au processus environnement de centraliser le recensement des individus en temps réel. Pour éviter toute corruption de données lors d'accès simultanés par des dizaines de processus, nous avons intégré des verrous de synchronisation (*count_lock*, *grass_lock*, *state_lock*) qui sécurisent la manipulation des compteurs. Nous y avons également inclus des indicateurs critiques tels que *epidemy_active*, permettant de notifier instantanément chaque agent d'une crise sanitaire en cours, ainsi qu'un flag de shutdown garantissant une extinction propre de tous les processus animaux sur commande de l'utilisateur.

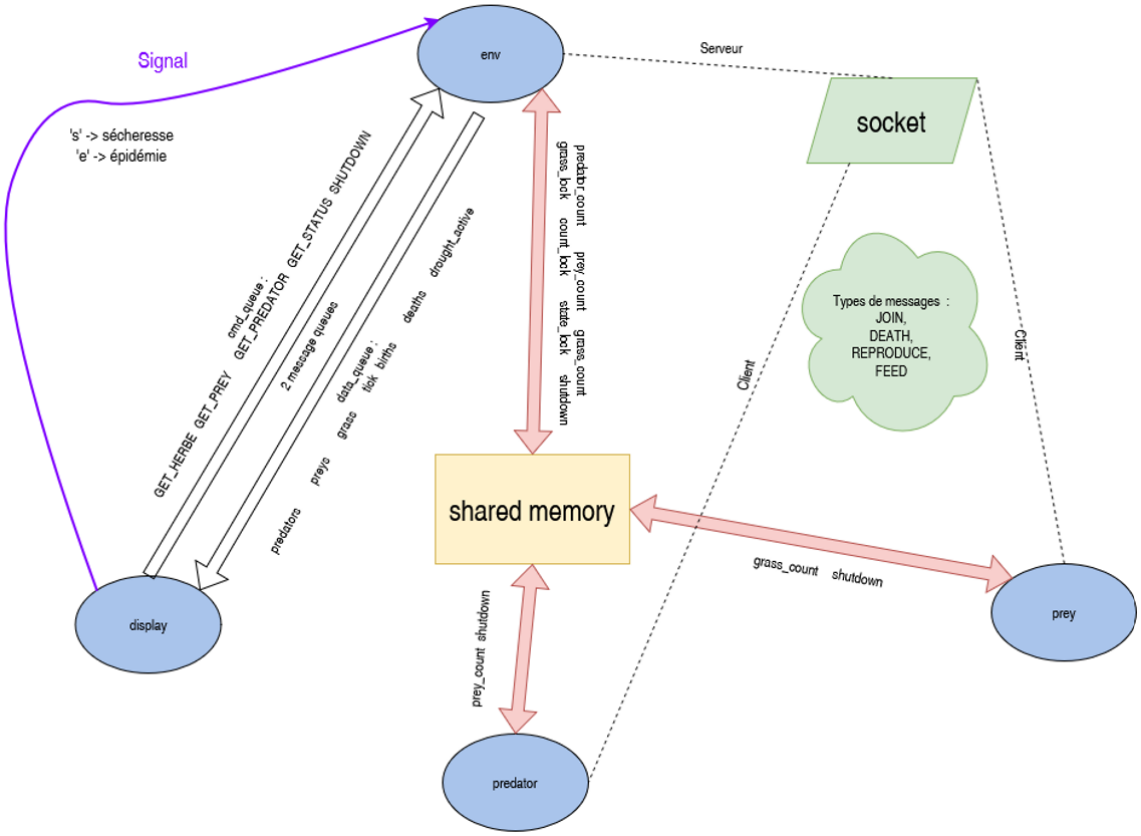

Figure 1 : Architecture générale

En complément de la mémoire partagée, nous avons implémenté un protocole de communication par sockets pour gérer les événements dynamiques de l'écosystème. Dans ce schéma, le processus environnement agit comme un serveur TCP local tandis que chaque individu, prédateur ou proie, devient un client. Cette liaison est cruciale pour transmettre des messages de type événementiel tels que l'arrivée d'un nouvel individu, la mort d'un autre ou une demande de reproduction. L'utilisation des sockets permet de traiter ces flux de manière asynchrone, l'environnement recevant les requêtes et mettant à jour les statistiques globales en conséquence, ce qui assure une grande réactivité du système même avec une population importante.

La liaison entre le processus d'affichage et l'environnement est assurée par deux files de messages distinctes afin d'éliminer tout risque de collision de données ou de blocage du flux. La première, *cmd_queue*, est dédiée à l'envoi des ordres de commande depuis le display vers l'environnement, comme les paramètres initiaux ou les demandes d'arrêt. La seconde, *data_queue*, sert exclusivement au retour d'informations de l'environnement vers l'affichage. Cette séparation des flux permet au display de demander des mises à jour de statut régulières pour rafraîchir le terminal de l'utilisateur sans interférer avec les commandes de contrôle. Enfin, nous avons utilisé les signaux système pour gérer les interruptions prioritaires, permettant ainsi à l'utilisateur de déclencher instantanément des événements climatiques ou sanitaires majeurs, tels que les sécheresses ou les épidémies, directement depuis l'interface de contrôle.

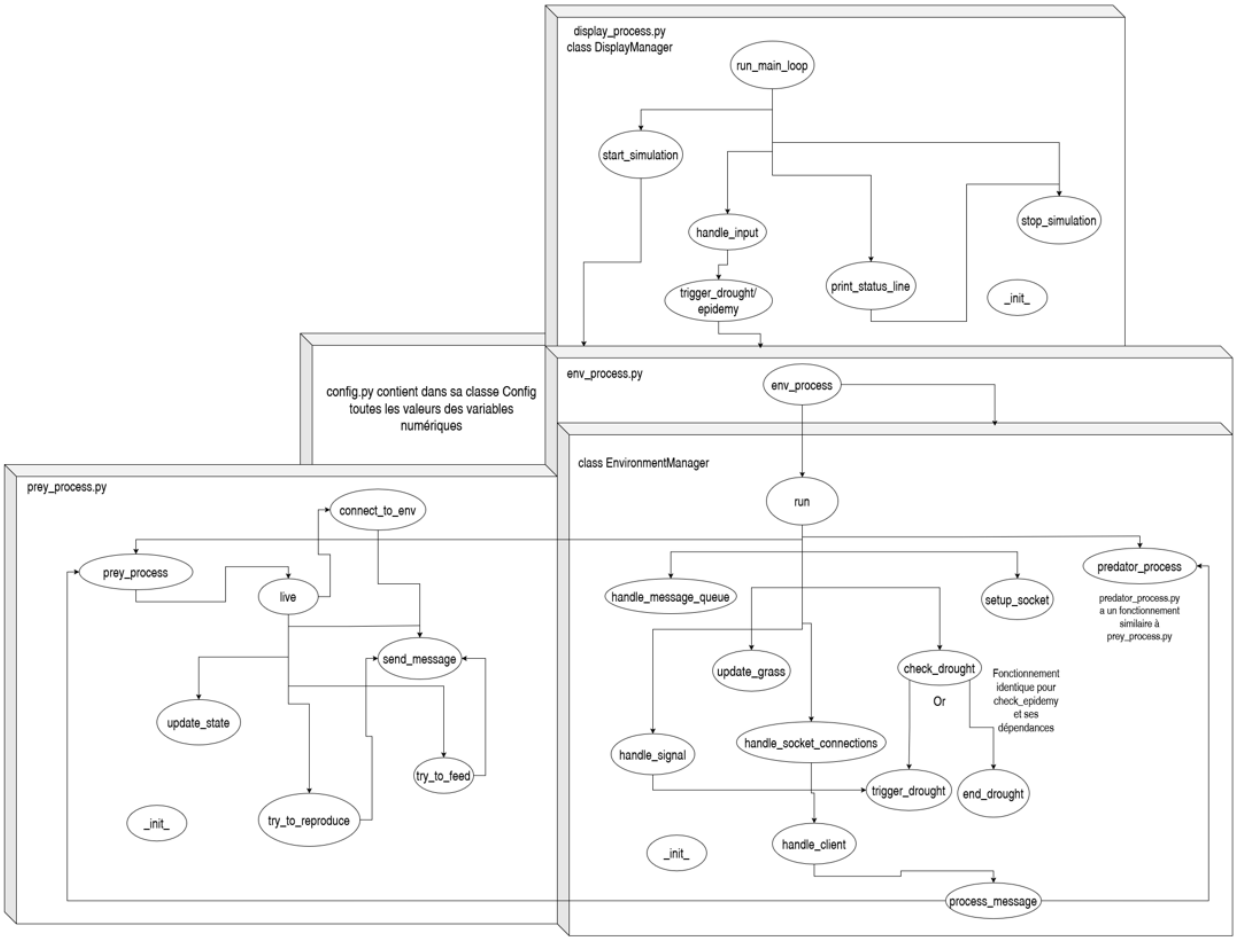

Figure 2 : Architecture des fonctions

IV. Algorithmes principaux

La réalisation de notre simulation repose sur trois algorithmes fondamentaux. Chaque processus exécute sa propre logique tout en restant synchronisé avec l'ensemble du système, garantissant ainsi un comportement cohérent de l'écosystème.

1 - Le gestionnaire d'environnement (env):

```
DÉBUT
  // 1. Initialisation
  Initialiser la mémoire partagée et les verrous
 Démarrer le serveur de Socket et son thread d'écoute (JOIN, DEATH…)

  // 2. Déploiement de la population
  Pour chaque prédateur et proie initial :
      Lancer un processus individuel (predator_process ou prey_process)
```

```
        Enregistrer le processus dans une liste de suivi
        Attendre 0.1s (évite la saturation au démarrage)
    Fin pour

    // 3. Boucle de simulation (Ticks)
  Tant que la simulation est active :
        Incrémenter tick_count
        Lire cmd_queue -> Activer sécheresse ou épidémie si demandé

        Si sécheresse : diminuer herbe | sinon : augmenter herbe
        Si fin épidémie : désactiver l'indicateur épidémie en mémoire partagée

        Envoyer l'état global (stats + météo) vers data_queue
        Attendre la durée du Tick
    Fin tant que

    // 4. Fermeture propre
    Activer shutdown en mémoire partagée (stoppe tous les processus animaux)
    Fermer le serveur de socket
    Attendre la fin de tous les processus enregistrés
    Libérer les ressources
FIN
```

2 - Le cycle de vie des individus (predator/prey) :

```
DEBUT
    // 1. Initialisation et connexion
    se connecter au serveur via socket
    envoyer message "join" (type d'entité, id)

    tant que vivant et énergie > 0 et shutdown == 0 et âge < âge_max:
        vieillir de 1 an
        diminuer l'énergie (coût de survie)

        // gestion de la faim
        si energie < seuil_faim :
            consulter la mémoire partagée (herbe ou proies dispo)
            si nourriture disponible :
                tenter de manger (probabilité de succès pour les prédateurs)
                si succès : augmenter l'énergie
```

```
        // gestion de la descendance
        si énergie > seuil_reproduction :
            si tirage aléatoire favorable :
                envoyer message "reproduction" via socket
                réduire l'énergie (coût de naissance)

        // gestion des risques
        si epidemie_active en mémoire partagée :
            si tirage aléatoire < taux_mortalité :
                mourir (sortir de la boucle)
    fin tant que

    envoyer message "death" via socket pour libérer la place
    fermer la connexion
FIN
```

3 - L'affichage et contrôle utilisateur (display) :

```
DEBUT
    lancer le processus environnement (env)

    tant que la simulation tourne :
        // lecture des entrées utilisateur (non-bloquant)
        si touche pressée == 's' : envoyer signal sécheresse à env
        si touche pressée == 'e' : envoyer signal épidémie à env
        si touche pressée == 'q' : envoyer message shutdown dans cmd_queue

        // mise à jour de l'écran
        si message disponible dans data_queue :
            lire les statistiques (nombre d'individus, état herbe, météo)
            calculer l'état de santé global (stable, critique, extinction)
            afficher la ligne de statut

        attendre un court instant (rafraîchissement visuel)
    fin tant que

    nettoyer et arrêter tous les processus
FIN
```

V. Plan de réalisation

Notre stratégie d'implantation a suivi une progression logique permettant de valider la stabilité des échanges de données à chaque nouvelle étape. Nous avons commencé par établir un socle commun via la mémoire partagée afin de permettre aux processus des proies et des prédateurs de communiquer leurs statistiques de base à l'environnement. L'affichage (display) a aussi été connecté à cette mémoire dans un premier temps, pour plus de facilité. Une fois ce partage d'informations réalisé, nous avons intégré les sockets pour gérer les événements ponctuels comme les naissances et les décès, transformant ainsi l'environnement en un véritable serveur centralisé capable de traiter les requêtes de dizaines de clients simultanés.

L'interface de contrôle a ensuite été connectée à ce noyau via l'implémentation d'une première file de messages. Cependant, nos tests ont rapidement révélé que l'utilisation d'une file unique pour les ordres de l'utilisateur et le retour des données statistiques créait des collisions et des problèmes dans l'affichage (le programme semblait attendre quelque chose sans jamais le recevoir). Pour résoudre ce problème technique, nous avons fait le choix de doubler cette infrastructure en créant deux files distinctes, la *cmd_queue* pour les commandes descendantes et la *cmd_data_queue* pour les flux remontants, garantissant ainsi une réactivité optimale du terminal.

Pour améliorer la réactivité du système face aux événements d'urgence comme les catastrophes climatiques, nous avons ensuite ajouté la gestion des signaux permettant un déclenchement instantané des crises sans attendre le prochain cycle de lecture des files. Enfin, une fois que tout fonctionnait, nous avons retiré l'accès à la mémoire partagée du processus d'affichage. Nous avons donc transféré l'intégralité des flux de données utilisant auparavant cette mémoire vers les files de messages, isolant ainsi totalement l'interface graphique du cœur logique de la simulation.

Tout au long de ce processus, nous avons effectué des tests afin de vérifier la bonne implémentation des fonctionnalités.

- **Tests du cycle de vie individuel :** Nous avons d'abord isolé une proie seule pour vérifier qu'elle consultait bien la mémoire partagée et consommait l'herbe disponible. Ce test a permis de valider que l'énergie augmentait correctement, déclenchant soit une reproduction via socket, soit une mort naturelle une fois l'âge maximum atteint.
- **Tests de prédation et survie** : Un protocole similaire a été appliqué aux prédateurs pour confirmer que leur survie dépendait strictement de la capture de proies et que le coût métabolique entraînait bien leur disparition en cas de famine.
- **Validation des signaux** : Nous avons testé l'envoi des signaux système pour déclencher manuellement les sécheresses et épidémies. Nous avons ainsi vérifié que l'arrêt de la croissance de l'herbe et l'augmentation de la mortalité étaient instantanément perçus par l'ensemble des processus actifs. Nous avons vérifié que l'envoi de plusieurs signaux de même nature d'affilée n'entraînent pas de problème. Dans ce cas précis, un nouvel évènement remplace seulement le précédent.
- **Tests de l'affichage et des flux de données** : Nous avons contrôlé la cohérence entre les données réelles de l'environnement et leur rendu dans le terminal. L'objectif était de s'assurer que le passage par les files de messages (queues) ne créait pas de décalage et que les alertes visuelles restaient fluides même avec une population élevée.

VI. Exécution

L'utilisation de notre simulateur a été conçue pour être immédiate et transparente, reposant sur un terminal unique qui fait office de centre de commande. Avant le lancement de la boucle de simulation, le programme invite l'utilisateur à saisir certains paramètres initiaux directement dans la console. Cette phase de configuration manuelle permet de définir le nombre de proies et de prédateurs qui composeront la population de départ, ainsi que la quantité initiale de ressources végétales. Ces entrées sont importantes car elles déterminent le point d'équilibre de l'écosystème : une population de prédateurs trop élevée dès le départ pourrait, par exemple, mener à une extinction rapide avant même que l'utilisateur n'ait pu interagir avec le système.

Une fois ces données validées, le fichier principal dédié à l'affichage prend en charge l'initialisation de la hiérarchie des processus. Dès le démarrage, l'utilisateur voit apparaître une interface textuelle dynamique où les populations de prédateurs, de proies et l'état de la végétation sont mis à jour en temps réel à chaque tick de simulation et affiche une nouvelle interface uniquement lorsqu'il y a eu du changement. Cette interface a été optimisée pour rester lisible même lorsque le système gère plusieurs centaines de processus simultanés, en utilisant des symboles clairs pour identifier chaque catégorie d'acteurs. L'interaction avec l'écosystème se poursuit ensuite de manière non-bloquante. L'utilisateur peut déclencher manuellement des crises environnementales en pressant la touche “s” pour une sécheresse ou la touche “e” pour une épidémie, influençant instantanément le comportement autonome de chaque processus animal via les signaux et la mémoire partagée.

Pendant que la simulation tourne, l'utilisateur observe la dynamique des populations évoluer. Les naissances et les morts se succèdent, créant des modifications constantes dans les statistiques affichées. On remarque souvent rapidement que le système s'auto-régule : une explosion de la population de proies est presque toujours suivie d'une augmentation des prédateurs, avant qu'une phase de famine ne vienne stabiliser l'ensemble. L'utilisateur devient alors un observateur actif capable de tester les limites de cet équilibre. Lorsqu'il décide d'arrêter le programme avec la touche “q”, la procédure de sortie assure qu'aucun calcul ne reste en suspens.

VII. Prise de recul

La finalisation de ce projet nous a permis d'évaluer la pertinence de nos choix techniques face aux contraintes du multiprocessus. Bien que le simulateur soit opérationnel, l'analyse de notre travail révèle des points qui pourraient être améliorés.

Nous avons attribué à chaque individu un identifiant unique généré aléatoirement lors de la reproduction (lignes 125 et 135 du fichier environnement). Si ces identifiants ne sont pas réellement exploités actuellement, ils ont constitué un outil de débogage précieux pour suivre la trace de processus spécifiques durant la phase de test. Nous sommes conscients qu'une génération purement aléatoire comporte un risque théorique de collision, où deux individus pourraient partager le même ID. Toutefois, au vu de la taille de la population et de la plage de valeurs choisie, ce risque demeure statistiquement négligeable et sans impact sur la stabilité actuelle, l'identifiant n'étant pas utilisé comme clé primaire de synchronisation.

Sur le plan de l'architecture, la structure actuelle centralise une grande partie de la logique de contrôle au sein du processus environnement, conformément aux exigences initiales du projet. Avec le recul, une meilleure répartition des responsabilités entre les différents processus permettrait d'alléger la charge de

l'environnement. On pourrait imaginer déléguer davantage de calculs de probabilités ou de gestion de ressources directement aux processus agents.

On aurait aussi pu permettre à l'utilisateur de choisir plus de paramètres au début de la simulation, et de les laisser par défaut s'il ne le souhaitait pas.

La gestion des mécanismes de protection de la mémoire partagée aurait sûrement pu être améliorée, mais ne sachant pas vraiment comment, nous avons fait au mieux avec ce qui marchait. Ainsi, dans la fonction *trigger_drought* dans *env_process*, une modification de variable n'est pas protégée par un mutex ce qui ne pose normalement pas de problème dans notre cas mais qui n'est probablement pas très propre.

Enfin, le réalisme de la simulation pourrait être enrichi par une modélisation plus complexe des conditions météorologiques. L'introduction d'un cycle de pluie, par exemple, permettrait une croissance de l'herbe plus organique et dynamique, contrastant avec le modèle linéaire actuel. Une croissance aléatoire, indexée sur des variables climatiques fluctuantes, offrirait une meilleure imprévisibilité au système et obligerait les populations à s'adapter à des cycles d'abondance et de rareté moins prévisibles, rendant l'étude de l'équilibre de l'écosystème encore plus riche.

Annexe sur l'IA

Dans un premier temps nous avons utilisé l'IA de Claude sonnet 4.5 pour développer un prototype de projet.

Nous lui avons fourni l'architecture que nous avions réfléchie ainsi que les fonctions principales. Cela nous a donné un ordre d'idée du résultat attendu. Et suite à cela nous avons continué par nous-mêmes en enlevant des parties inutiles, problématiques, voire même des parties parfaitement étanches du reste du programme telles que la sécheresse ou l'entrée de données de l'utilisateur. En effet, ces dernières ne communiquaient absolument pas avec le reste du programme.

Puis, nous avons débugué le programme en suivant le plan de réalisation établi précédemment. A chaque étape, des tests étaient effectués pour vérifier que les différents moyens de communications fonctionnaient correctement.

Pendant cette phase, l'architecture du projet a été modifiée : la fonction main a été retirée et remplacée par display, ce qui a nécessité quelques modifications dans l'organisation du code. Nous avons aussi ajouté les signaux qui manquaient encore, notamment ceux liés aux situations de sécheresse puis d'inondation, ainsi que de nouvelles entrées utilisateur pour permettre une interaction plus complète avec le système.

Au final, notre projet actuel ne ressemble en aucun cas au premier jet que nous avions généré avec l'IA. Cependant certaines parties qui ont été rajoutées par la suite ont de nouveau été générées artificiellement et ont peu ou pas été retouchées :

- Nous avons utilisé la dernière version disponible de Copilot Smart GPT-5.1 pour gagner du temps dans le choix des variables numériques du fichier config.py.
- Dans le fichier *display_process2.py* nous avons utilisé sonnet 4.5 de Claude pour corriger notre fonction *handle_input* (ligne 43) car nous avions des soucis pour prendre en compte les inputs de l'utilisateur et pour corriger la lecture de la réponse du statut de ENV dans la fonction *run_main_loop* (ligne 95).
- Nous l'avons également utilisé pour la récupération ligne par ligne dans le buffer dans la fonction *handle_client* (ligne 81 et 82) du fichier *env_process* et pour nous aider à gérer les threads dans la fonction *handle_socket_connections* (ligne 54).
- Nous l'avons aussi utilisé dans *send_message* (*predator_process* et *prey_process*), ligne 53 afin de pouvoir envoyer notre message correctement, avec le bon format, via la socket.
- Enfin, l'IA (ChatGPT version GPT-5.2.) nous a parfois été utile lors des débugages, pour décrypter certaines commandes dans le terminal lorsqu'il y avait des centaines de lignes après un KeyboardInterrupt.

Nous avons également utilisé Gemini 1.5 Flash pour nous aider sur la rédaction du README.md ainsi que pour développer la rédaction du rapport afin de gagner du temps. Nous avons bien entendu réfléchi au contenu par nous-mêmes.