PRO3600: Mars Lander

Augustin Bresset | Zacharie March

2023 - 2024

Table des matières

1 Introduction

1.1 Présentation du problème : Mars Lander

Mars Lander est un problème d'optimisation proposé sur la plateforme *CodinGame* s. d. Ce jeu consiste en le contrôle du moteur d'un vaisseau spatial (puissance des moteurs, angles de rotation) afin de le faire atterrir en toute sécurité et en douceur sur une surface plane de la planète Mars. L'idée est donc de trouver une trajectoire fonctionnelle. Ce problème est un classique de l'optimisation et de l'intelligence artificielle, on lui retrouve des solutions basées sur de nombreuses approches très différentes.

1.1.1 Environnement

La surface de Mars est représenté localement par une suite continue de segments. Parmis ces segments, un seul est rigouresement vertical, celui-ci définit la zone d'atterissage que la navette doit viser. Au moindre contact avec la surface, le vaisseau atterit et le jeu est terminé.

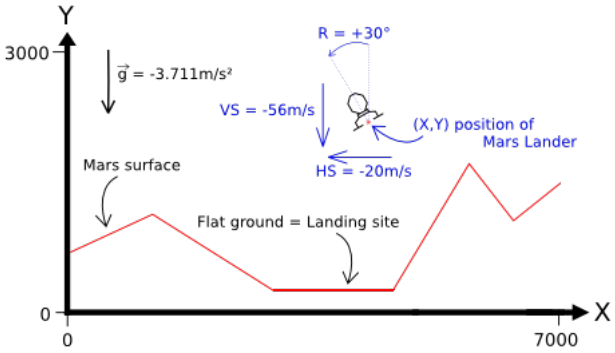


FIGURE 1 – Schéma du problème
CodinGame s. d.

1.1.2 Espace des Etats

Le vaisseau est représenté comme un propulseur à L'état du vaisseau est représenté par un vecteur de 7 valeurs :

Paramètre	Description	Bornes
x	position horizontale	$[0, 7000]$
y	position verticale	$[0, 3000]$
$hSpeed$	vitesse horizontale	-
$vSpeed$	vitesse verticale	-
$fuel$	quantité de carburant restante	$[0, \infty[$
$rotate$	angle de rotation	$[-90, 90]$
$power$	puissance des moteurs	$[0, 4]$

1.1.3 Espace des Actions

Toutes les secondes, en fonction des paramètres d'entrée (position, vitesse, fuel, etc.), le programme doit fournir le nouvel angle de rotation souhaité ainsi que la nouvelle puissance des fusées de Mars Lander. Mais les commandes de puissance des fusées et de l'angle de rotation sont bornés. La rotation est limitée à $[-15, 15]$ degrés et la variation de puissance des moteurs est limitée à $\{-1, 0, +1\}$.

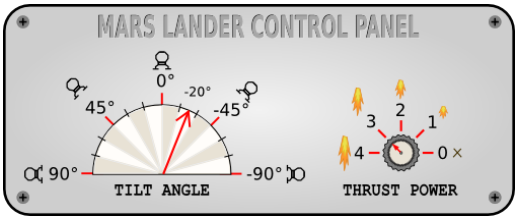


FIGURE 2 – Figure du tableau de commande
CodinGame s. d.

1.1.4 Conditions d'atterrissage

Afin de considérer qu'un attérissage est réussi, il faut respecter les conditions suivantes :

— atterrir dans la **zone d'atterissage**
— atterrir en **position verticale** (angle d'inclinaison = 0°)
— la **vitesse verticale** doit être limitée ($\leq 40m.s^{-1}$ en valeur absolue)
— la **vitesse horizontale** doit être limitée ($\leq 20m.s^{-1}$ en valeur absolue)

1.2 Algorithme Génétique

Afin de résoudre ce problème, nous avons choisi d'utiliser une approche heuristique, les algorithmes génétiques.

Les algorithmes génétiques sont des algorithmes d'optimisation stochastique inspirés de la théorie de l'évolution naturelle. Ils sont basés sur le principe de **sélection naturelle**.
Dans un premier temps on génère une **population** de solution aléatoire, on simule ensuite ces solutions et on calcule un **score** pour chacune d'entre elles. On fait ensuite évoluer cette population à travers des phénomènes semblables à la sélection naturelle tel que la **sélection**, la **reproduction** et la **mutation**. On répète ce processus jusqu'à ce qu'une solution satisfaisante soit trouvée. Dans notre problème, une solution est définie par une trajectoire, c'est à dire une suite de commande à effectuer par le vaisseau.

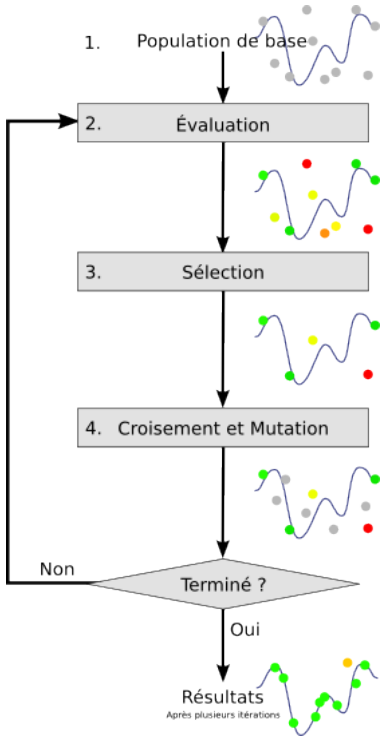


FIGURE 3 – *Schéma d'un algorithme génétique* s. d., Schéma du fonctionnement de l'algorithme génétique

1.3 Objectif

L'objectif de ce projet est de développer un support sur lequel il sera possible de simuler, manipuler et visualiser les trajctoires de notre navette sur différentes cartes. De plus on souhaite pouvoir appliquer nos propres solutions informatiques à ce problème et notamment l'algorithme génétique et lui dédier une visualisation adaptée afin de bien étudier le problème et l'influence des hyperparamètres de la solution. Pour cela on à définit plusieurs points afin de mener à bien notre projet :
— Création d'une **interface graphique** sur pygame
— Mise en place d'une **interface client** permettant de piloter la navette
— Calcul de **solution** notamment à l'aide d'algorithme génétique
— Regrouper les fonctionnalités à l'aide d'un **menue**

1.3.1 Cahier des charges

Développer une applicattion avec interface graphique. Les fonctionnalités attendues sur l'application sont les suivantes :
— Interface graphique avec menue
— Choix de la solution à utiliser
— Choix de la carte à utiliser
— Visualisation adapté à la solution utilisée
— Exécution de la simulation

Notre application sera de sorte à ce que l'on puisse facilement ajouter des solutions, des cartes et autres tel que des obstacles dynamiques. Pour cela l'architecture devra s'inscrire dans une logique agile et modulaire. On compte tester notre

environnement à l'aide de tests unitaires et fonctionnels afin de vérifier la bonne dynimique de notre environnement en le comparant à des environments déjà existants tel que celui issu de *CodinGame* s. d., CodinGame.

L'objectif final étant d'avoir une plateforme sur laquelle il est facile de développer et tester des solutions à ce problème. C'est pourquoi le langage **Python** est proposé pour ce projet, il permet de développer facilement des solutions et de les tester rapidement.

1.4 Organisation du projet

1.4.1 Répartition des tâches

En claire la répartition des tâches est la suivante :

— **Augustin Bresset**
Développement de l'environnement et des solutions
— **Zacharie March**
Développement de l'interface graphique et de l'interface client

Dans les deux cas, un apprentissage de la librairie pygame est nécessaire. Que ce soit pour la création de la solution "manuelle" (contrôle par l'utilisateur) ou pour la création de l'interface graphique.

Table 1 – Répartition des heures dans la création d'un projet informatique

Phase	**Augustin Bresset (heures)**	**Zacharie March (heures)**
Organisation	2	2
Apprentissage	4	12
Développement	22	20
Débogage et tests	14	11
Rédaction des livrables et autres documents	8	6
Total	50	51

On a passé beaucoup de temps a coder ensemble car on a un niveau très hétérogène et le pair programming nous a permis de progresser plus rapidement sans en laisser un derrière. D'où la raison des commits qui sont toujours fait pas l'un des membres du groupe, cela n'est pas représentatif du travail fourni par chacun.

1.4.2 Outils de travail

Afin de communiquer, nous nous reposons pour la communication active sur plusieurs outils, étant deux, nous n'étions pas restreint par le besoin de créer un groupe sur un outil de communication particulier tel que **Discord** ou **Slack**. Pour la commun passive nous avons créée une organization sur Github dans laquelle on peut trouver deux repertoire :

— **backend** : Contient le code source du projet
— **meta** : Contient la documentation du projet (et ce livrable)

Nous souhaitons aussi travailler en **pair programming** le plus possible afin de faciliter la communication et la compréhension du code.

2 Développement

2.1 Architecture du projet

Afin de mener notre projet à bien, nous avons décidé de découper le code en plusieurs modules.

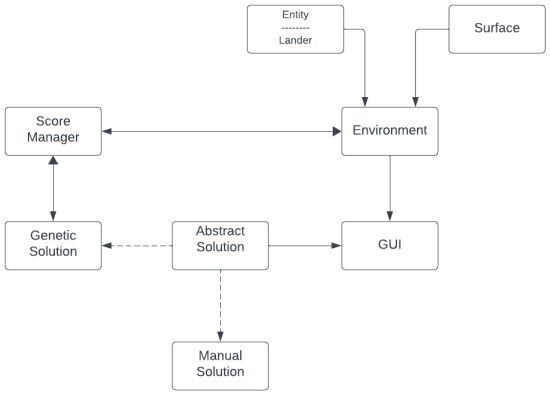


FIGURE 4 – Diagramme d'instance

Représente les interactions entre les différents objets

La figure 4, nous présente les interactions entre les différents modules du projet. Les flèches représentent les dépendances entre les modules. Par exemple *Environment* dépend de *Lander* et *Surface*. Chaque module et ces sous-modules ont une responsabilité bien définie. Cette architecture a été pensé de manière agile, l'idée est donc de séparer les responsabilités au bon moment. Tout d'abord, la séparation de l'**environnement** et de la **Surface** permet un ajout de surface plus complexe avec des formes plus variées. De même pour **Entity** qui est ici utilisé seulement pour le **Lander** mais qui pourrait être utilisé pour d'autres entités tel que des obstacles dynamiques. Ensuite on a fait de même pour les **Solutions**, on a inscrit les commandes manuelles comme un type de solution à ce *problème* afin de l'utiliser de la même manière que des solutions informatiques tel que l'algorithme génétique implémenté dans **GeneticSolution**. Enfin la séparation de l'interface graphique **GUI** et du **ScoreManager** avec l'**Environment** permet une plus grande maniabilité dans le développement de solutions qui fonctionne par apprentissage. En effet, on peut simuler des trajectoires sans avoir besoin de l'interface graphique et donc sans être limité par les performances de celle-ci et les FPS.

2.1.1 Environnement

Comme vu sur le diagramme, l'environnement instancie un **Lander** et une **Surface** et permet de simuler la trajectoire du vaisseau. Il calcule les états dynamiques du système à chaque instant et permet de savoir si le vaisseau est en collision avec la surface et si il est sur la zone d'atterissage. Il implémente les méthodes suivantes :

- **reset** : remet à zéro les paramètres dynamiques du vaisseau
- **exit_zone** : vérifie si le vaisseau est sorti de la carte
- **landing_on_site** : vérifie si le vaisseau est sur la zone d'atterissage
- **landing_angle** : vérifie si l'angle du vaisseau est correct
- **landing_vertical_speed** : vérifie si la vitesse verticale du vaisseau est correcte
- **landing_horizontal_speed** : vérifie si la vitesse horizontale du vaisseau est correcte
- **successful_landing** : vérifie si l'atterissage est réussi
- **next_dynamics_parameters** : calcule les paramètres dynamiques du vaisseau à l'instant suivant
- **step** : calcule les états du vaisseau à l'instant suivant et met à jour les paramètres dynamiques

Le **Lander** est une classe de données qui permet de représenter l'état du vaisseau à un instant donné mais ce n'est pas le cas de **Surface** qui implémente **tey_collide** qui détermine si une trajectoire est en collision avec la surface et retourne le segment de surface dans ce cas là.

Enfin on a définit trois cartes différentes stockées dans des fichiers json, par exemple la carte `level_one.json` est définie comme suit :

```
{
    "name": "Level one",
    "points" : [[0, 100], [1000, 500], [1500, 100], [3000, 100], [5000, 1500], [6999, 1000]],
    "lander_state" : {
        "x" : 2500,
        "y" : 2500,
        "h_speed" : 0,
        "v_speed" : 0,
        "fuel" : 1000,
        "rotate" : 0,
        "power" : 0
    }
}
```

Ainsi pour définir une carte il suffit de définir dans un fichier json, son **name**, les **points** qui définissent la surface et l'état initial du vaisseau.

2.1.2 Gui

L'interface graphique dépend de la solution adoptée, c'est pourquoi on a une class abstraite **GuiAbstract** qui permet de définir les méthodes communes à toutes les interfaces graphiques. Les méthodes implémenté sont les suivantes :

- **screen_reset** : Réinitialise l'écran de l'interface graphique.
- **render_reset** : Réinitialise effaçant tous les éléments affichés.
- **reset** : Réinitialise les paramètres dynamiques du vaisseau et de la surface.
- **display_text** : Affiche du texte à l'écran de l'interface graphique.
- **draw_surface** : Dessine la surface sur l'écran de l'interface graphique.

Méthodes abstraites :

- **write_parameters** : Écrit les paramètres de la solution dans un fichier.
- **step** : Effectue une étape de simulation de la trajectoire du vaisseau.
- **pygame_step** : Effectue une étape de simulation de la trajectoire du vaisseau spécifiquement pour l'interface graphique pygame.
- **run** : Lance l'interface graphique et démarre la simulation.

Cette classe abstraite a permet l'implémtation d'une classe pour l'interface graphique **Gui** pour le pilotage manuel du vaisseau et une autre classe **GuiTrajectory** qui permet de visualiser plusieurs trajectoires simultanément, fonctionnalité utile pour visualiser par population d'individus dans le cas de l'algorithme génétique.

2.1.3 Solution

Une solution est définie par une classe abstraite **AbstractSolution** qui permet de définir les méthodes communes à toutes les solutions. Elle demande l'implémentation des méthodes suivantes :

- **get_parameters** : retourne les paramètres de la solution
- **set_parameters** : définit les paramètres de la solution
- **use** : envoi une instance d'action en fonction de l'envirronement

On a implémenté deux types de solutions :

- **ManualSolution** : Solution manuelle permettant de piloter le vaisseau à l'aide du clavier
- **GeneticSolution** : Solution génétique permettant de piloter le vaisseau à l'aide d'un algorithme génétique

La solution manuelle est implémenté sans méthode supplémentaire, mais la solution génétique a besoin de plusieurs méthodes supplémentaires pour fonctionner. Elle emploi notamment l'objet **Population** qui implémente toutes les méthodes liés à l'évolution : **selection**, **mutation**, ... ainsi que des méthodes supplémentaires tel que la création d'une population aléatoire et une fonction de roulette cumulative. Cette population créer un tableau de genome qui repondent à la classe abstraite **AbstractChromosome** qui permet de définir les méthodes communes à tous les chromosomes. Une classe **ActionChromosome** a été implémenté pour ce problème, elle permet de définir les actions à effectuer par le vaisseau à un instant donné. L'idée derrière l'emploi de classe abstraite est l'application d'algorithme génétique à d'autres paramètres du problème.

2.1.4 Score

Le module **ScoreManager** permet de calculer le score d'une trajectoire. Cette classe n'a aucun intérêt pour le mode manuel du vaisseau, mais permet à l'aide d'un algorithme génétique de trouver la meilleure trajectoire possible. Il implémente les méthodes suivantes :

- **landing_distance** : calule la distance au sol entre la zone de collision et la zone d'atterissage
- **scoring_distance_off_site** : détermine un score en fonction de la distance au sol
- **scoring_distance_on_site** : donne le score si le vaisseau est sur la zone d'atterissage
- **scoring_speed_on_site** : détermine le score en fonction de la vitesse du vaisseau lors de l'atterissage dans le cas où le vaisseau est sur la zone d'atterissage
- **scoring_speed_off_site** : détermine le score en fonction de la vitesse du vaisseau lors de l'atterissage dans le cas où le vaisseau n'est pas sur la zone d'atterissage
- **scoring_angle** : détermine le score en fonction de l'angle du vaisseau lors de l'atterissage

2.1.5 Utils

Le module **Utils** contient des fonctions utilitaires et des constantes utilisées dans les autres modules et notamment les classes **Point** et **Segment**.

Point

La classe Point permet de représenter un point dans un espace à deux dimensions. Il lui ait associé des fonctions permettant de calculer la distance et de gérer les égalités entre deux points. On a considérer ici que deux points étaient égaux si leurs coordonnées étaient assez proches.

Segment
La classe Segment permet de représenter un segment dans un espace à deux dimensions. Sa méthode la plus utile est la méthode **collision** qui à l'aide d´une fonction *CCW* s. d., **CCW** qui permet de savoir si deux segments se croisent.

En considérant un instant de trajectoire comme étant un segment, on peut vérifier si il ne rentre pas en collision avec la surface grâce à cette méthode.

Enfin d'autre scripts ont permis le débuggage tel que **display_map** ou bien **load_map** qui permet de charger la carte.

2.2 Tests

Afin de vérifier le bon fonctionnement de notre code, nous avons mis en place des tests. Ces tests se divisent en trois catégorites qui composent ces parties. On va voir dans un premier temps les **tests unitaires** qui permettent de tester les fonctions et méthodes importantes et de manières indépendantes au reste du code. Il enn suivra les **tests d'intégration** qui permettent de tester le bon fonctionnement des modules entre eux. Et enfin les **tests fonctionnels** qui permettent de tester le bon fonctionnement de l'application dans son ensemble.

Nous utiliserons la librairie **unittest** pour l'écriture de nos tests.

2.2.1 Tests unitaires

Environnement Afin de tester l'environnement, nous avons récuperer les réponses dynimiques de l'environnement présent sur *CodinGame* s. d., CodinGame et nous avons comparer les états à chaque instant à ceux calculer par notre environnement. De plus on a fait des tests pour vérifier que les collisions avec la surface étaient bien détectées. Et notamment que les collisions avec avec le site d'atterissage est bien détecté et que si le vaisseau sort de la carte, ce soit bien détecté par l'environnement.

Score Les fonctions étant simplement des fonctions mathématiques dépendant d'un paramètre du problème, nous n'avons pas jugé nécessaire de faire des tests unitaires excepté pour la fonction qui permet de calculer la distance entre la zone de collision et la zone d'atterissage. En effet, ce n'est pas une simple distance euclidienne, mais plutot la distance en suivant la surface de l'environnement. On a pu vérifier le bon fonctionnement de cette fonction à l'aide de cas caculé à la main. Et les trois tests nous permettent de nous laisser penser que :

— La distance est bien décroissante si l'on se dirige vers la zone d'atterissage
— La distance est bien calculée

Algorithme Génétique Ces tests unitaires sont destinés à évaluer le fonctionnement de certaines fonctionnalités de l'algorithme génétique mise en œuvre.

On test tout d'abord la génération d'une population. Il crée une population de 10 individus, chacun ayant 100 gènes du type "ActionChromosome". Les assertions comparent la longueur de la population, la longueur des gènes du premier individu, et s'assurent que la longueur des gènes est la même pour tous les individus, tout en vérifiant qu'il y a des différences entre les gènes des deux premiers individus.

Puis on test la **cumulative wheel**, ce test évalue la fonction qui génère une roue cumulative pour la sélection d'individus. Une population de 10 individus est créée avec des scores aléatoires. La population est triée en fonction des scores, puis la roue cumulative est générée. Les assertions vérifient que la longueur de la roue cumulative est correcte, et que chaque élément de la roue est une paire d'individus de type "ActionChromosome", avec des identifiants différents.

Enfin on vérifie **la sélection** du/des meilleur individu dans une population.Ce test évalue la fonction de sélection du meilleur individu dans une population. Une population de 10 individus est créée, chaque individu ayant un score égal à son index. La population est triée en fonction des scores, et la meilleure solution est sélectionnée. Les assertions comparent le score de la meilleure solution avec l'index attendu, soit 9 dans ce cas. En résumé, ces tests garantissent le bon fonctionnement de différentes parties de la solution génétique, y compris la génération de populations, la création de roues cumulatives pour la sélection, et la sélection du meilleur individu dans une population.

2.2.2 Tests d'intégration et fonctionnels

Les tests d'intégration et fonctionnels on plutôt été fait de manière empirique, c'est à dire que l'on a testé le bon fonctionnement des modules entre eux à l'aide de l'interface graphique. On a pu ainsi vérifier que les solutions fonctionnaient bien avec l'environnement et que l'interface graphique fonctionnait bien avec les solutions notamment grâce aux logs ajoutés dans le prompt.

```
(mars-lander) (base) smaug@smaug-TM1703:~/pro3600-MarsLander/backend$ python3 src/launcher.py
pygame 2.5.2 (SDL 2.28.2, Python 3.10.12)
Hello from the pygame community. https://www.pygame.org/contribute.html
Run MarsLander on map Reversed Cave by Genetic

Welcom to Mars Lander challenge !

Here are some usefull commands :
 - RIGHT ARROW : evolve your population
 - SPACE : continue the evolution even if a good trajectory has been found
 - Q : Quit the simulation
```

FIGURE 5 – Exemple de logs

3 Résultats

3.1 Menue

L'application est divisée en deux parties, le **menue** et la **partie**.

Dans le menue, on propose à l'utilisateur de choisir entre deux solutions : **Manual** et **Genetic** puis entre trois cartes prédéfinies : **level one**, **Reversed Cave** et **Flat Surface**.

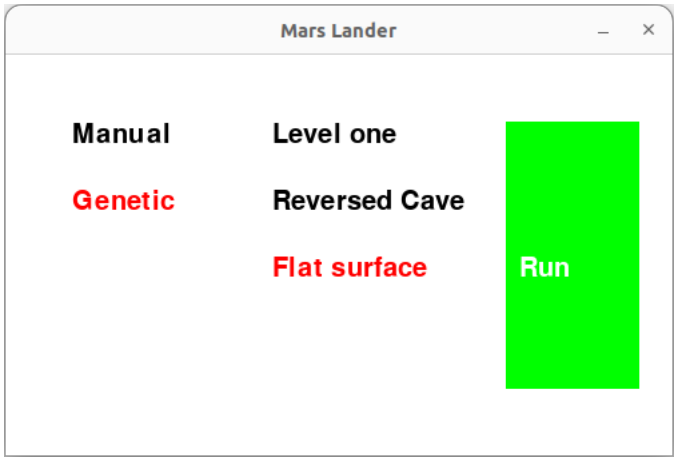


FIGURE 6 – Menue

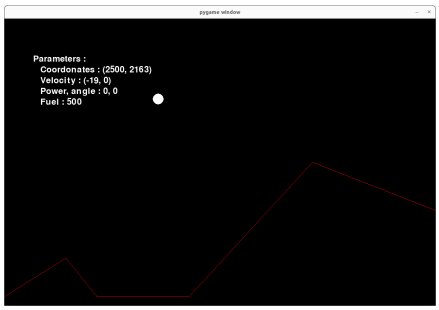
Ce menue permet de choisir la solution et la carte à utiliser pour la simulation, il suffit enfin de cliquer sur le bouton **Run** pour lancer la partie.

3.2 Partie

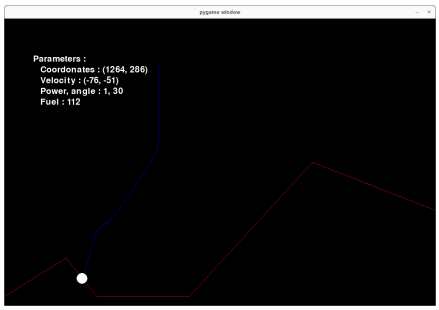
Une partie est lorsque l'on choisit la solution manuelle, elle ce joue à l'aide des flèches du clavier. Les commandes sont les suivantes :

- **Flèche du haut** : Augmente la puissance des moteurs
- **Flèche du bas** : Diminue la puissance des moteurs
- **Flèche de gauche** : Tourne le vaisseau à gauche
- **Flèche de droite** : Tourne le vaisseau à droite

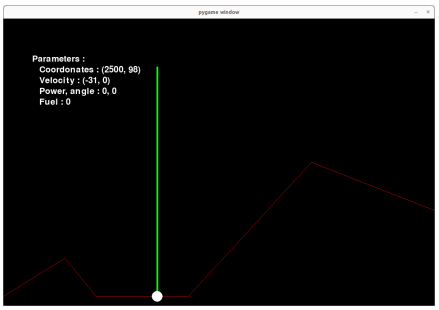
Une fois la partie lancé, on peut contrôler le vaisseau. Arriver au sol, le logiciel trace la trajectoire effectuer et sa couleur indique le succès ou non de l'atterissage, vert pour un succès et bleu pour un échec.



(a) Début de partie en manuel



(b) Fin de partie avec crash

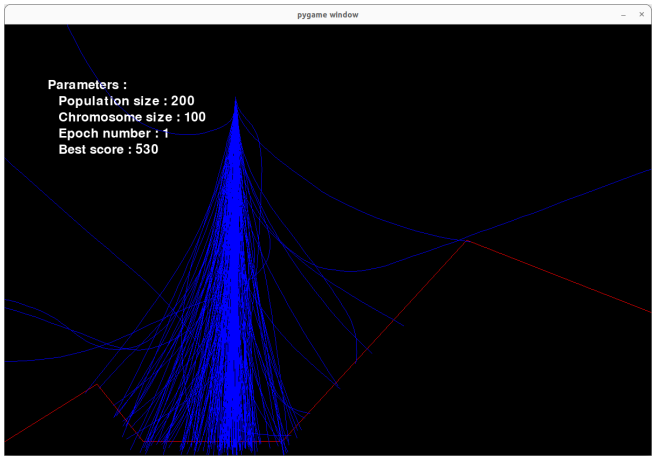


(c) Fin de partie avec succès

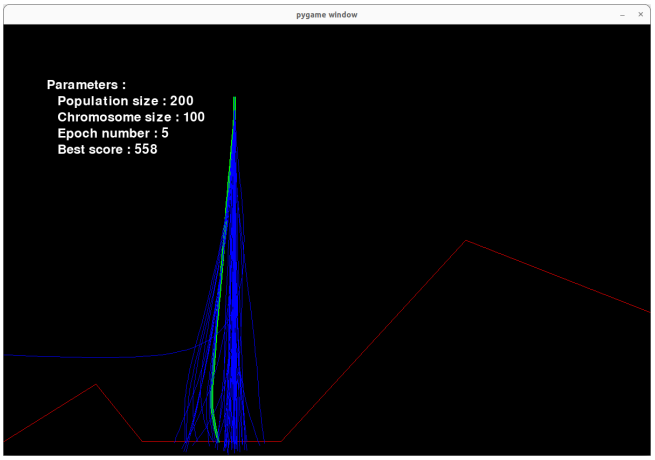
Le vaisseau est représenté par un cercle blanc et la surface par des segments rouges.

3.3 Simulation

Pour représenter les résultats de l'algorithme génétique, on affiche les trajectoires population par population. On peut ainsi voir l'évolution des trajectoires au fur et à mesure des générations.

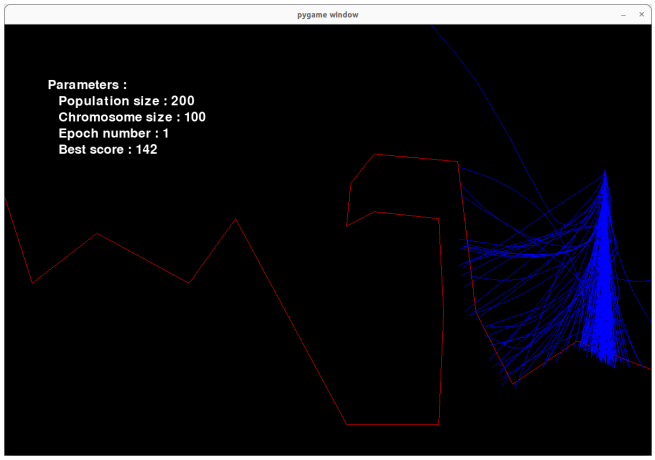


(a) Début de partie en génétique

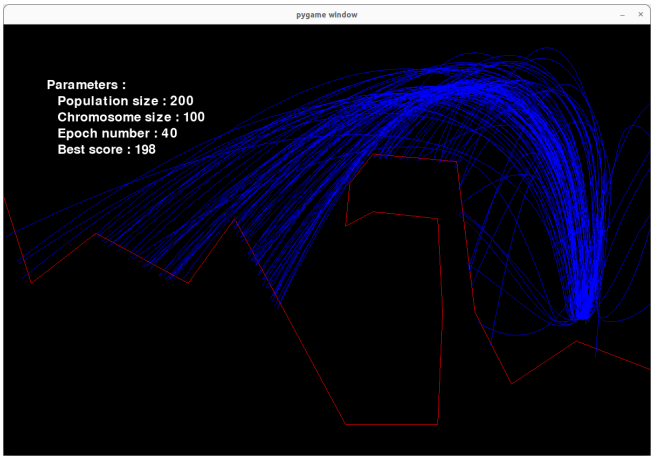


(b) Fin de partie en génétique

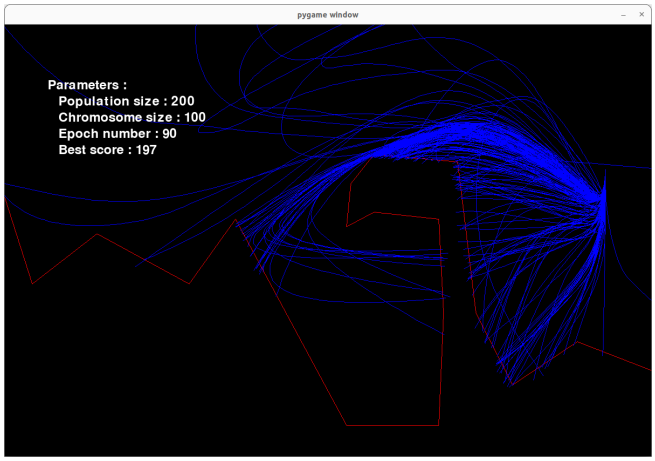
On a aussi une carte plus compliquée, la **Reversed Cave** qui permet de tester la robustesse de notre algorithme génétique.



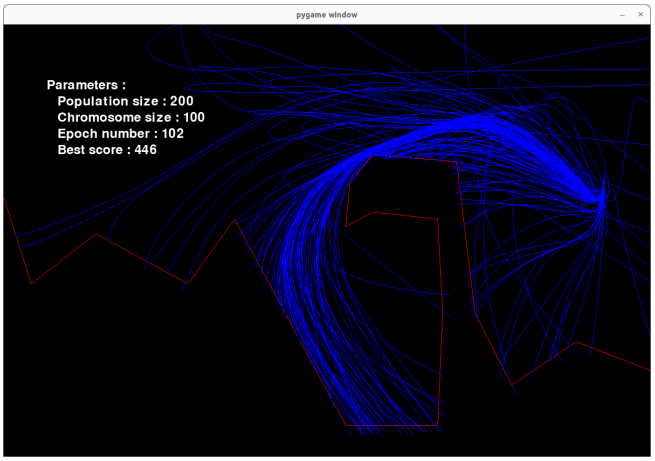
(a) Début de partie en génétique



(b) Simulation à l'époque 40



(c) Simulation à l'époque 90



(d) Simulation à l'époque 102

3.4 Manuel utilisateur

Afin de lancer le programme, il faut tout d'abord installer les dépendances du projet présentes dans le fichier **requirements.txt**. Pour cela on peut utiliser la commande suivante :

```
pip install -r requirements.txt
```

Une fois les dépendances installées, on peut lancer le programme à l'aide de la commande suivante :

```
python src/launcher.py
```

3.5 Critique et conclusion

3.5.1 Problèmes rencontrés

Tout au long du projet, on a essayé de garder une architecture de code permettant une meilleure répartition des processus tout en gardant une cohérence dans le code. Comme dit plus tôt, on a essayer de garder une structure agile, cela nous a prit du temps d'y réfléchir et de restructurer le code sans cesse.

Nous voulions diviser les repertoires github entre le frontend et le backend, mais nous avons finalement pas rencontrés ce besoins étant donné que l'on travaillait sur le même langage.

N'ayant pas de deadline claire, on a pu prendre le temps de bien réfléchir à l'architecture du code et à la répartition des tâches mais ca nous a aussi conduit a prendre plus de temps que nécessaire pour certaines tâches. Finalement, on s'est remis à faire des deadlines pour pouvoir avancer plus rapidement.

3.5.2 et amélioration à envisager

On est fière d'avoir sur créer une architecture répondant à nos attentes et on aurait aimé prendre plus de temps à développer d'autres solutions qu'elle soit sur les algorithmes génétiques ou sur d'autres types de solutions.

Dans l'interface graphique, on aurait aimé ajouté un menue supplémentaire permettant de modifier directement les paramètres du jeu sans passer par le code. On aurait aussi aimé ajouter des obstacles dynamiques pour rendre le jeu plus intéressant.

Finalement d'un point de vue optimisation, notre jeu aurait pu être bien plus performant si on avait forcé l'utilisation de numpy et numba pour les calculs mais n'étant tous les deux pas du mêmes niveau en python, on a préféré ne pas le faire et surtout que le jeu est déjà assez rapide pour être jouable.

4 Annexes

Références

CodinGame (s. d.). URL : `https://www.codingame.com/multiplayer/optimization/mars-lander`.
CCW (s. d.). URL : `https://bryceboe.com/2006/10/23/line-segment-intersection-algorithm/`.
Schéma d'un algorithme génétique (s. d.). URL : `https://fr.wikipedia.org/wiki/Algorithme_g%C3%A9n%C3%A9tique`.