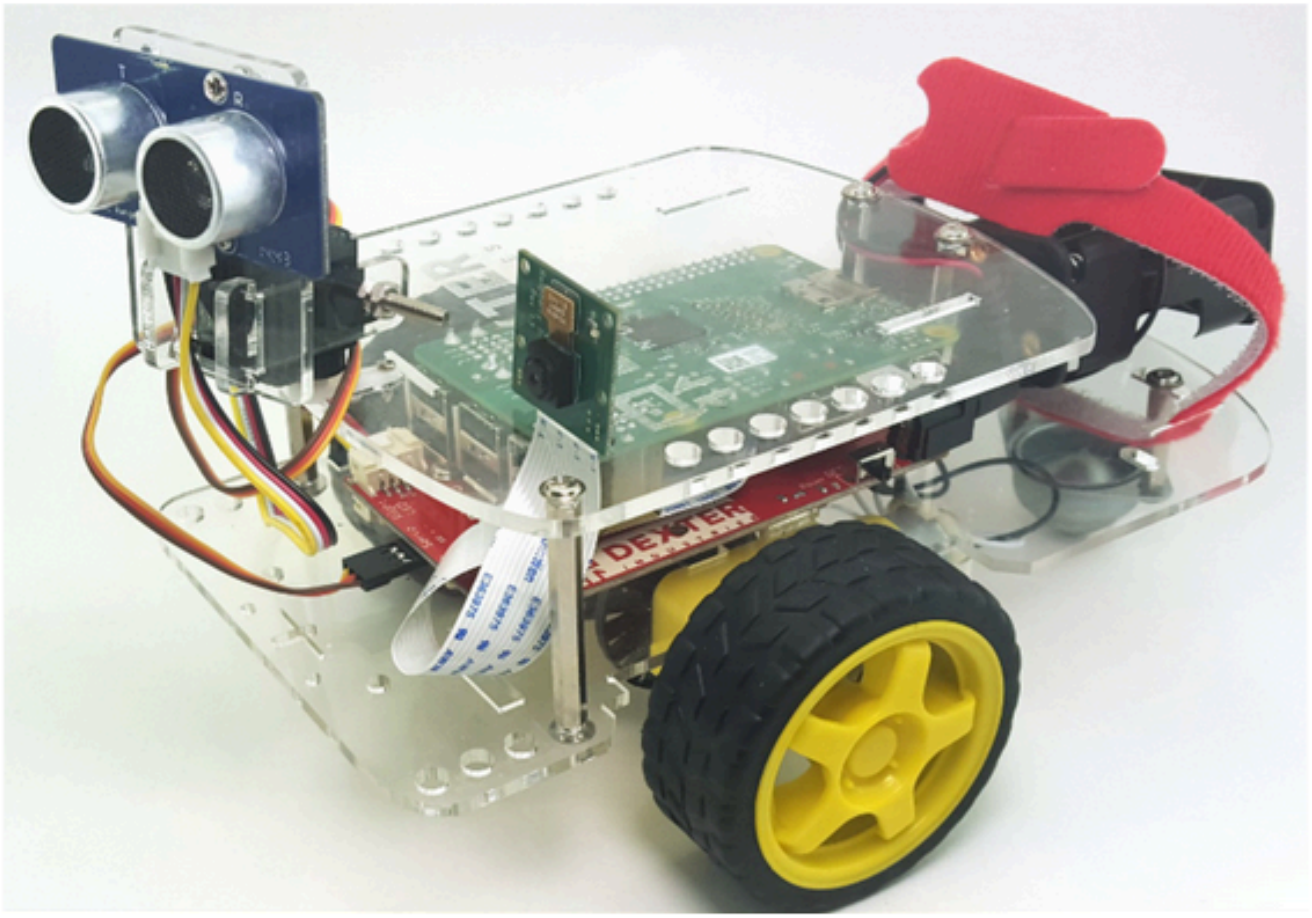
PROJET ROBOTIQUE - LU2IN013

COMPTE RENDU

DATAYANA BOT CORP



Réalisé par :

MANOUBI Taysir
SI MOHAMMED Yaniss
GU David
YAN Nanlin

Encadré par :

BASKIOTIS Nicolas
SIGAUD Olivier

Année Universitaire 2023-2024

Introduction

Cadre du projet

Le projet se déroule dans la cadre de l'UE de Projet de développement LU2IN013, de Sorbonne Université, pour notre 2ème année de licence. Encadré par nos professeurs de TD, que l'on remercie au passage pour leur disponibilité, leur sollicitude et leur patience, nous devions mettre en place une modélisation 2D (puis 3D) d'un robot fourni par l'Université en vue de son utilisation IRL.

Objectif de l'UE

Le projet, constituante principale de cette unité d'enseignement, n'est pas pour autant pas une UE de code à proprement parler, ou d'approfondissement d'un langage de programmation spécifique. L'objectif visé concerne principalement :

- le développement d'un travail d'équipe
- la gestion d'un projet dans un temps déterminé, sur l'architecture de la méthode AGILE/SCRUM
- le maintien d'un travail régulier et constant
- l'appréhension d'un nouvelle environnement de travail (Python)

Objectif du projet

L'objectif du projet se résume à effectuer trois tâches simple avec le robot :

- Tracer un carré
- S'approcher le plus vite d'un obstacle puis s'arrêter
- Suivre une balise (fournie par l'Université)

Pour mener à bien ce projet, deux grande partie se dégageait:

- La première partie visait uniquement à construire un environnement de simulation pour le robot
- La seconde partie concernait l'application des stratégies testé dans l'environnement virtuelle sur le robot IRL

Une troisième pouvait voir le jour, si les deux précédentes étaient terminées. Cette dernière partie, nous laissait le libre choix quant à l'élaboration de nouvelles tâches à fournir au robot (chasse au trésor, système de patrouille, etc.)

Contrainte du projet

Plusieurs contraintes, inhérente au projet, pouvait être soulevé :

- Le travail d'équipe :
  - Pour la plupart d'entre nous, nous n'avions jamais **véritablement** travailler en équipe au-delà du binôme.
  - L'application de la méthode SCRUM/AGILE qui était une nouvelle composante à incorporer dans notre méthode de travail
- L'appropriation de nouveaux outils :
  - Bien nous ayons quelques notions de python à l'issue de notre cursus, nous n'avions qu'utiliser la version impératif, et non la version objet que nous imposait le projet
  - L'utilisation d'un Kanban pour appliquer la méthode AGILE/SCRUM et structurer notre travail de groupe, nous était complètement étranger
- Ressource limité :
  - Deux robots disponible seulement, pour plusieurs groupes différents

Projet

Formation de l'équipe

Pour ce présent projet, le groupe était constitué de 4 membres :

- YAN Nanlin
- GU David
- SI MOHAMMED Yaniss
- MANOUBI Taysir

Présentation du code

Le package principale se situe dans le folder `./src` et sa structure est calquée sur le modèle MVC (*Model-View-Controler*).

Structure du projet/package `./src`

Le folder `./src` contient les modules nécessaires à l'exécution de la simulation 2D et 3D de notre environnement simulé. Comme énoncé précédemment, nous avons essayé de suivre le modèle MVC pour structurer notre projet.

- Ce dossier contient 3 principaux modules distincts:
  - un module `controller`
  - un module `model`
  - un module `view`

Nous décrirons ici, les deux derniers modules cités, soit le module `controller` et le module `model`. Le module `view` sera décrit plus en détails plus tard dans le rapport lorsque nous parlerons des stratégies créer.

Description des modules

Module `model`

Ce module définit principalement les deux fichiers contenant les classes essentielles de notre simulation que sont le robot et l'environnement.

Fichier `robot.py`

Le fichier `robot.py`contient les classes décrivant les objet :

- *Robot*() :
  - Décrit le robot dans l'environnement simulé. C'est une copie très approximative de notre robot IRL pour notre simulation.
- *RobotAdapter*():

- ➢ Représente la classe permettant de traduire les commande destiné à notre robot simulé en commande intelligible pour le robot IRL

- ❖ *RobotFake*():
  - ➢ Définit une instance qui mime les réponses du robot IRL pour vérifier de la bonne traduction des commande faites par l'Adaptateur

Fichier `environment .py`

Le fichier `environment.py` renferme deux classes :

- ❖ *Obstacle*() :
  - ➢ Permet l'instanciation d'obstacle dans l'environnement (seulement des forme rectangulaire)
- ❖ *Environment*() :
  - ➢ Instancie un environnement sur lequel se déroulera la simulation. L'instance est une composition de d'objet *Robot* et *Obstacle*

Module `view`

Ce module définit deux classes permettant un affichage des éléments constituants la simulation :

- ➢ *Interface2D.py*
- ➢ *Interface3D.py*

Fichier `interface2D.py`

Le ficher interface2D.py contient la classe Interface qui est une classe permettant l'affichage en 2D de notre simulation, effectué grâce à la librairie graphique tkinter, l'affichage est composé de 2 frames principales, une frame comportant une représentation de notre environnement, la view, et une frame ou se trouvent des boutons et un ensemble de Labels permettant d'obtenir des données tels que les vitesses des roues.

Fichier `interface3D.py`

Le fichier interface3D.py contient la classe Interface3D qui est une classe permettant l'affichage 3D de notre simulation, effectuée sous Panda3D, nous avons un ensemble de modèles créer sous Blender dans le dossier dénommé "assets", nous avons alors 4 assets, un asset d'environnement, un asset du robot, un asset d'obstacle et un asset de balise, Interface3D charge à l'initialisation l'environnement et le robot et réajuste leur taille de telle sorte à ce que les coordonnées de l'affichage et de l'interface graphique soit similaires, l'ajout des obstacles lui se fera directement à la fin de l'initialisation.

L'affichage 3D possède 3 points de vue différents, pouvant être actionné sur l'appui d'une touche :

- ➔ Dessus :
  - ◆ À l'appui de la touche "z", la caméra sera de tel sorte à ce qu'elle soit directement au dessus du robot
- ➔ Derrière :
  - ◆ À l'appui de la touche "a", la caméra sera de tel sorte à ce qu'elle soit directement derrière le robot
- ➔ Robot :
  - ◆ À l'appui de la touche "e", la caméra suivra directement les mouvements du robots et aura un point de vu première personne

Présentation des stratégies et de leurs structures

Description du module `controller`

Pour l'élaboration des stratégies, nous avons décidé de distingué les tâches simples des tâches complexes du lanceur de stratégie qui se traduit par la création :

- ➢ D'un dossier `strategies` composé de deux fichier `metastrat.py` et `unitstrat.py`
- ➢ D'un fichier `seqstrat.py`

Dossier `strategies`

Le dossier `strategies` distingue deux types de stratégies : les stratégies dites `meta` et les stratégies dites `unitaire`. Chacunes est consignée et définie dans des fichiers séparés.

Fichier `unitstrat.py`

Ce fichier contient la description d'une stratégie de type `*unistrat*` qui est une stratégie abstraite. Une stratégie unitaire se décompose en trois fonctions:

- ➢ un start
    - Qui décrit son comportement, ou les actions à effectuer au démarrage de la stratégie.
- ➢ un stop
    - Qui décrit la condition d'arrête de la stratégie
- ➢ un step
    - Qui constitue le corps d'exécution de la stratégie jusqu'à valider la condition `stop`.

Ce fichier contient des stratégies prédéfinies basées sur la structure définie ci-dessus. Voici la liste des stratégies unitaires élaboré :

Stratégie MoveForward

Cette stratégie unitaire à pour seul et unique objectif de faire avancer le robot sur une distance déterminée avant de s'arrêter.

Stratégie RotateInPlace

Cette stratégie permet de faire tourner le robot sur place selon un angle

Stratégie MoveForwardWithSensor

Cette stratégie oblige le robot à avancer jusqu'à rencontrer un obstacle puis de s'y arrêter à une distance donné du capteur

Stratégie StratIf

Cette stratégie permet de choisir entre deux stratégie selon une condition

Stratégie Stop

Cette stratégie définit comment arrêter le robot

Fichier `metastrat.py`

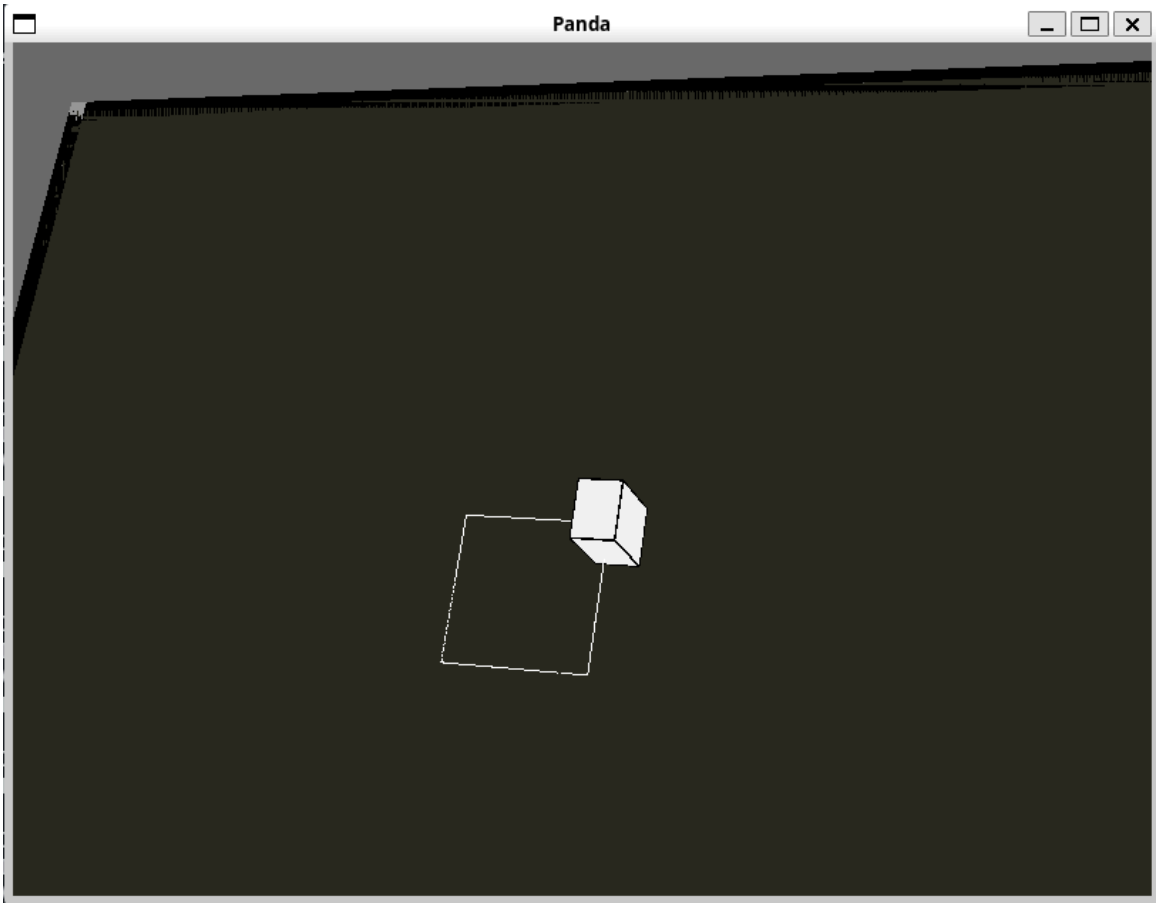
Le fichier `metastrat.py` possède des fonctions qui renvoient une liste de stratégie unitaire. Les meta-stratégies amalgament des suite de stratégie unitaires pour en faire des stratégie plus ou moins complexes.

Voici la liste des stratégies élaboré :

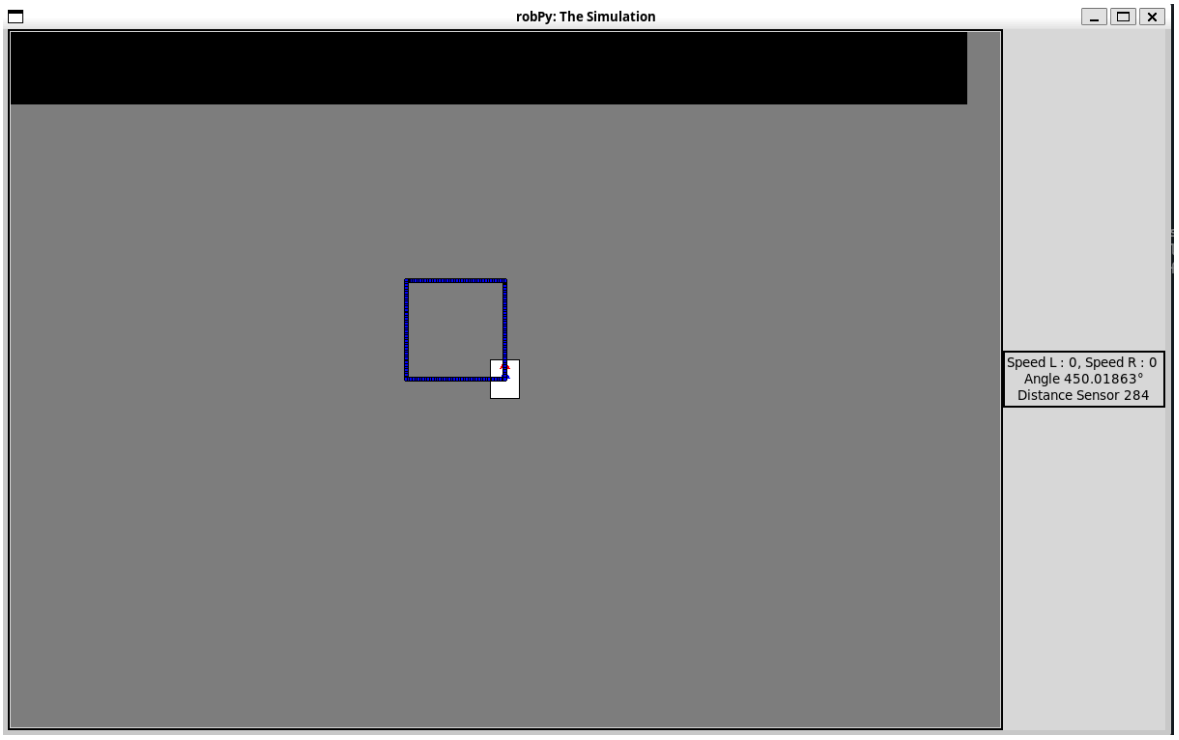
StratSquare

Renvoie une liste permettant d'effectuer un carré

Vue 3D (pour les valeur vitesse : 50 et distance : 100):



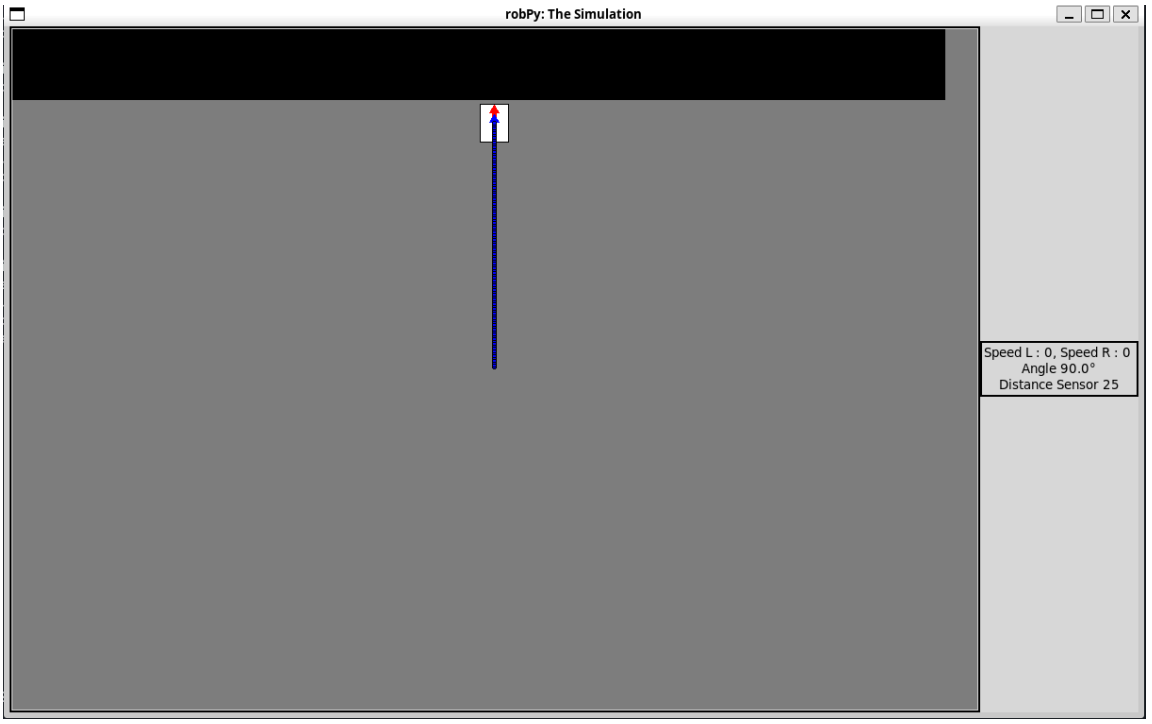
Vue 2D (pour les valeur vitesse : 50 et distance : 100):



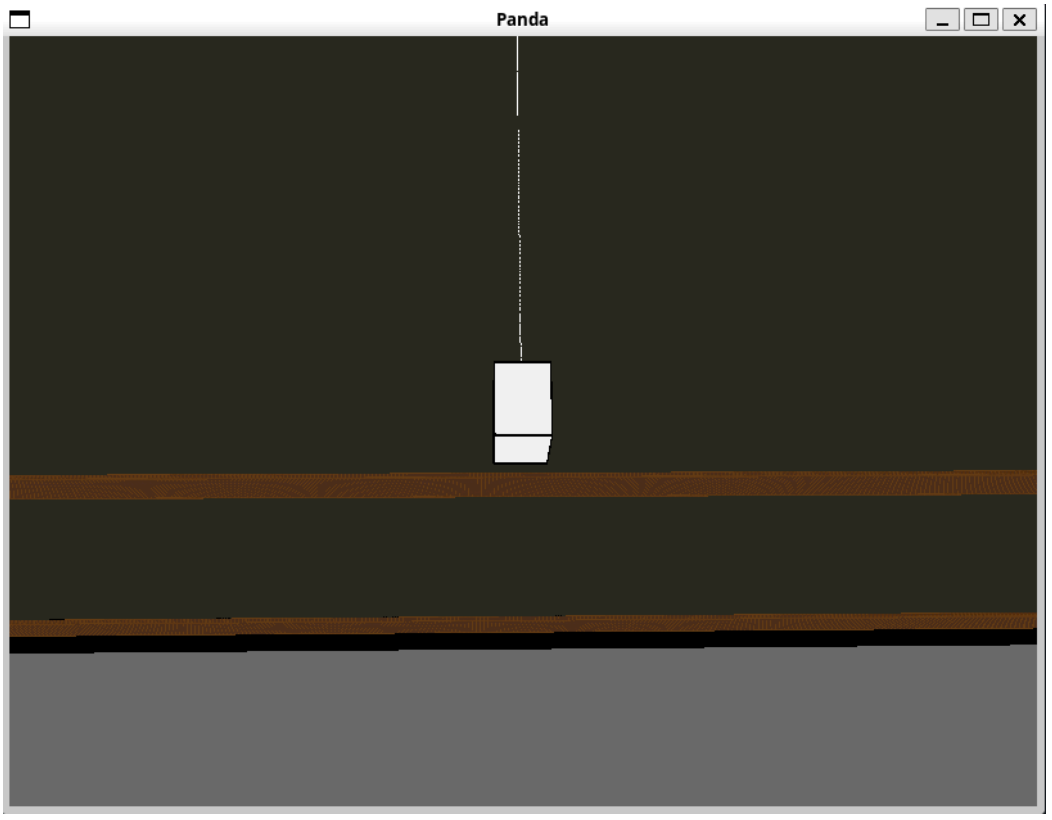
StratDontTouchTheWall

Renvoie une liste permettant d'effectuer une avancée vers un mur sans le toucher à une distance près

Vue 2D (valeur vitesse : 50 et distance au mur : 25):



Vue 3D (même valeur que pour la 2D):



Fichier `seqstrat.py`

Le fichier `seqstrat.py` abrite la définition de la classe `SequentialStrategy`. Son principal objectif est de lancer les stratégies qu'elle reçoit en paramètre. Une fois arrivée au terme de sa liste, elle s'arrête.

Conclusion

Bien que notre périple s'achève sur un petit goût d'inachevé, il n'en est pas moins resté une riche expérience. Nous avons composé avec nos faiblesses et nos avantages, et réussi, malgré un départ quelque peu chaotique et hasardeux, à parachevé deux de nos objectifs et entamé sérieusement le dernier d'entre eux.

Bien évidemment, le projet que nous avons mené ne reste qu'une ébauche qui mérite d'être poursuivie. Quelques pistes, par exemple, serait de finir de concevoir une stratégie pour trouver et suivre une balise. Ainsi que revoir une meilleure implémentation de stratégie probablement le fichier `metatstrategie.py`