

EL ATIA Youssef

ARLAIS Julien

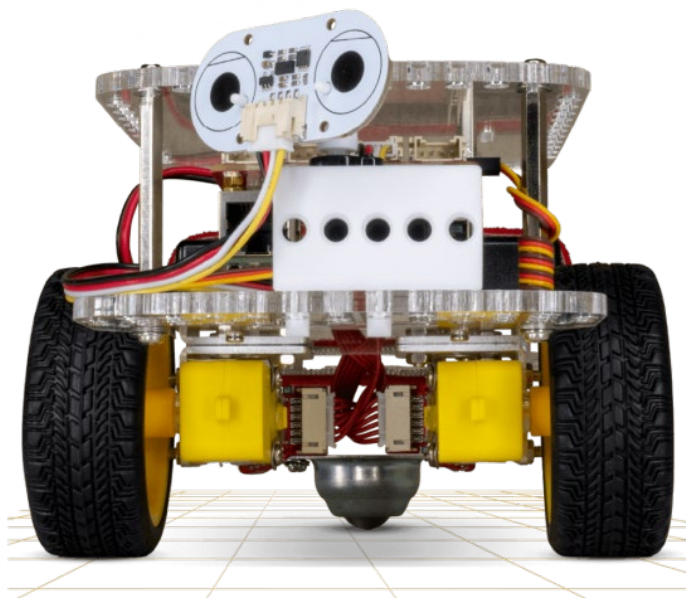
LIN David

MASSET Mael



RoboTech – LU2IN013

## Rapport final du projet Robotique



30 mai 2023

## Table des matières

<b>Introduction du Projet .....</b>	<b>3</b>
<b>1 Description global notre code .....</b>	<b>3</b>
1.1 Architecture du code .....	3
1.2 Description des fichiers .....	4
1.3 Tests.....	6
<b>2 Choix de conception .....</b>	<b>6</b>
2.1 Robot, Objet et Environnement .....	6
2.2 Interface Graphique.....	7
2.3 Gestion du temps.....	8
2.4 Threading .....	8
<b>3 Fonctionnalités du robot .....</b>	<b>9</b>
<b>4 Stratégies disponibles.....</b>	<b>9</b>
<b>5 Robot Réel.....</b>	<b>10</b>
<b>6 Conclusion.....</b>	<b>11</b>

# Introduction du Projet

Le but de notre projet est de réussir à implémenter une simulation pour notre robot virtuel afin de communiquer ultérieurement avec le robot réel Dexter en utilisant un proxy et d'exécuter des stratégies. Notre projet a été réalisé en suivant le design pattern Model-View-Controller (MVC), qui sépare les responsabilités de l'application en trois composants : le modèle (gestion des données), la vue (affichage) et le contrôleur (gestion des interactions). Cela favorise la modularité et la maintenabilité du code.

## 1 Description global notre code

### 1.1 Architecture du code

Dans le dossier RoboTech nous retrouvons les fichiers et dossiers :

**projet.py**

**camera\_test:** fichiers de test pour la caméra

**module :**

**contrôleur :**

**contrôleur.py**

**modele :**

**element\_simulation.py**

**proxy.py**

**robot\_api.py**

**robot\_mock\_up.py**

**vue :**

**affichage\_2D.py**

**camera.py**

**constante.py**

**core.py**

**test :** contient des fichiers de tests pour nos modules

**Compte rendu des séances** : contient tous les comptes rendus réalisés

Nous avons décidé d'utiliser des modules, plus précisément une arborescence de modules. En effet un répertoire peut être un module python à condition de disposer d'un fichier `__init__.py` qui est présent. Nous avons privilégié l'utilisation de module car ils nous permettent une certaine modularité et réutilisation du code.

Certains modules peuvent répondre à deux usages qui sont ceux qui sont destinés à être importés et ceux qui sont destinés à être le point d'entrée d'une application. Et c'est pour cette raison que nous devons faire la distinction qui est possible grâce au fait qu'un module a un nom contenu dans la variable `__name__` et que ce dernier est celui qu'on lui a donné lorsqu'il est importé, mais porte le nom `__main__` lorsqu'il est exécuté.

## 1.2 Description des fichiers

**projet.py** : sert à lancer notre projet. On importe l'api du robot si on y est connecté, le `mock_up` sinon pour ne pas causer d'erreurs. Dans le `main`, on n'a gardé que les éléments importants : on a initialisé un environnement, un robot, un proxy réel ou virtuel selon le besoin et fait appel à la fonction `run` appartenant à `core.py`.

**core.py** : fait office de sous-main, particulièrement la fonction `run` : on y déclare une simulation, les stratégies et les threads. Il y a aussi une implémentation de l'affichage 2D qui peut être désactivée en peu de modifications. Ce fichier comporte aussi toutes les importations nécessaires au fonctionnement du `main` et les fonctions `run_strategie` et `run_simulation` appelées dans `run` au niveau des threads et qui mettent à jour respectivement la stratégie à exécuter et la simulation. `run_strategie` lance la fonction `update` de la stratégie en paramètre puis celle du proxy, lui-même argument de la stratégie, afin d'effectuer la tâche requise puis de mettre à jour le temps, la distance parcourue et l'angle parcourue du proxy tant que la condition d'arrêt de la stratégie, sa fonction `stop`, n'est pas remplie. `run_simulation` quant à elle fait appel à la fonction `update` de la simulation en paramètre ainsi qu'à celle de GUI pour les synchroniser lors du lancement d'une simulation avec affichage 2D.

**contrôleur.py** : contient nos différentes stratégies. Tout d'abord nous avons `StrategieAvance` qui fait avancer le robot de la distance donnée. `StrategieArretMur` est identique excepté que le robot s'arrête avant de rencontrer un obstacle. Ensuite, `StrategieAngle` fait tourner le robot de l'angle donné. Enfin, nous disposons d'une stratégie séquentielle intitulée `StrategieSeq`.

**element\_simulation.py** : comprend nos classes, telles que `Objet` qui n'a pas de correspondance avec l'expérience réelle, mais surtout `Robot`, `Environnement` et `Simulation`. L'environnement gère la génération d'obstacles et la collision ainsi que les déplacements du robot. Nous avons choisi de représenter le robot par un cercle afin de simplifier la gestion des collisions. Un robot est défini par des coordonnées  $x$  et  $y$ , son orientation  $\theta$  qui est en radians mais qui prend en paramètre une valeur en degrés pour son initialisation, les vitesses de ses roues en radians par seconde et ses dimensions en millimètres : son rayon, la distance entre les roues et le rayon d'une roue. La fonction `get_distance` simule le capteur du robot, elle regarde dans la direction du robot et renvoie la distance le séparant du mur ou d'un obstacle. La simulation est constituée d'un environnement et d'un robot, s'occupe de leur mise à jour et renvoie une exception si une collision a lieu.

**affichage\_2D.py** : on dispose d'une classe GUI, contenant une initialisation de la fenêtre d'affichage et de la représentation du robot à sa position initiale, ainsi que d'éventuels obstacles mais aussi une fonction `update` qui s'occupe de modéliser le déplacement du robot en fonction de la simulation en cours.


**proxy.py** : ce fichier sert de traducteur entre notre code et le robot, il contient deux classes : `Proxy_Virtuel` et `Proxy_Reel`. Chacune contient les mêmes fonctions, pour que l'on ait juste à interchanger ces proxies dans le main pour faire fonctionner la simulation ou le robot réel. Cela permet donc d'avoir une seule version de chaque stratégie gérant à la fois le réel et le virtuel. Chaque proxy est constitué d'un constructeur, de `set_vitesse` qui donne une vitesse à chaque roue, de `update_distance` et `update_angle` pour mettre à jour la distance et l'angle parcourus, de `reset_distance` et `reset_angle` pour les réinitialiser, de `get_capteur_distance` qui sert de capteur et renvoie la distance séparant le robot de l'obstacle le plus proche en ligne droite, de `get_vitAng` qui renvoie les valeurs de vitesses angulaires des roues du robot, de `tourner` qui fait tourner le robot sur lui-même à une certaine vitesse, et enfin de `reset` et `update` pour qui combinent respectivement les autres fonctions `reset` de la classe et les autres fonctions `update`.


**robot\_api.py** : contient la classe `Robot2IN013` contenant toutes les fonctions du robot réel, dans son langage.


**robot\_mock\_up.py** : contient la même chose que `robot_api` mais sans les corps de fonctions des méthodes de la classe, remplacés par des `pass`, pour fonctionner dans la simulation.


## 1.3 Tests


Dans le dossier test nous retrouvons les fichiers :

 test\_2D

 test\_controleur

 test\_element\_simulation

 test\_proxy

 test\_toolbox

Le fichier **test\_2D.py** contient les tests pour l'interface graphique du fichier `./module/vue/affichage2D/py`

Le fichier **test\_element\_simulation.py** contient les tests du fichier `./module/modele/element_simulation.py`

Le fichier **test\_controleur.py** qui contient les tests du fichier `./module/controleur/controleur.py`

Le fichier **test\_toolbox.py** contient les tests du fichier `./module/toolbox.py`

Nous avons importé le module `unittest` de la bibliothèque standard de Python qui inclut le mécanisme des tests unitaires.

Pour cela nous définissons une classe héritant de `unittest.TestCase`, et nous définissons ensuite une méthode dont le nom commence par `test`.

Pour exécuter tous les tests unitaires on exécute la commande : **`python -m unittest discover test -v`**

## 2 Choix de conception

### 2.1 Robot, Objet et Environnement

On a choisi de représenter l'environnement sous la forme d'un terrain continu afin d'obtenir une plus grande précision. Cette représentation est légèrement plus complexe que celle d'un tableau discontinu mais elle représente bien mieux l'environnement pour la simulation.

Concernant le robot et les objets de l'environnement, on a opté pour une représentation sous forme de cercle, cette décision facilite leur manipulation et notamment la gestion des collisions.

De plus, on a décidé d'utiliser des coordonnées réelles pour représenter la position du robot et des objets afin d'assurer une précision optimale dans un environnement continu.

Cette combinaison d'un terrain continu avec des positions à valeur réelles, offre une représentation plus détaillée pour la simulation

## 2.2 Interface Graphique

On a choisi d'utiliser Tkinter pour créer une interface graphique en raison de sa simplicité d'utilisation et de sa compatibilité avec Python. Tkinter est une bibliothèque standard de Python qui offre une variété d'outils et de widgets pour créer des interfaces graphiques.

On a utilisé une fenêtre Tkinter pour afficher l'interface graphique, ce qui permet de créer un conteneur visuel pour les différents éléments de l'interface. La fenêtre est personnalisée avec le titre "Interface Graphique" pour une meilleure compréhension.

Ensuite, on a ajouté un canevas à l'intérieur de la fenêtre, on a dimensionné le canevas en fonction des valeurs de largeur, hauteur et échelle de l'environnement, on utilise également un multiplicateur « mult » défini dans le fichier « constante.py » pour multiplier tous les éléments présents sur le canevas. Cela permet de s'assurer que le canevas a une taille appropriée pour afficher les éléments de manière proportionnelle et clair.

En ce qui concerne la représentation graphique du robot, on a utilisé un cercle rouge pour le représenter avec la fonction « create\_cercle ». Le choix de la couleur rouge a été fait pour le distinguer des autres objets sur le canevas. On a utilisé les coordonnées et le rayon du robot pour positionner et dimensionner correctement le cercle sur le canevas.

En plus du cercle du robot, on a ajouté une ligne pour représenter la direction du robot avec la méthode « create\_line » du canevas pour créer une ligne à partir des coordonnées du robot et on ajoute une flèche à l'extrémité de la ligne pour indiquer la direction du robot.

Pour chaque objet dans l'environnement, on crée un cercle noir pour représenter chaque objet sur le canevas, ça permet de visualiser tous les objets présents dans l'environnement.

Pour la mise à jour de l'interface graphique, on crée un cercle noir aux coordonnées actuelles du robot pour représenter sa trace sur le canevas de l'interface graphique. Puis on supprime le cercle précédent et la ligne représentant le robot du canevas en utilisant la fonction « delete ». Cela nous permet de nettoyer le canevas avant de mettre à jour la position du robot.

On recrée le cercle et la ligne du robot aux nouvelles coordonnées du robot, afin de les afficher à leur nouvelle position sur le canevas. Cette approche permet de mettre à jour visuellement la position du robot sur l'interface graphique en supprimant les éléments graphiques précédents et en les recréant aux nouvelles coordonnées. Ainsi, la trace du robot est mise à jour au fur et à mesure que le robot se déplace dans l'environnement simulé.

Nous pouvons également nous séparer de l’affichage graphique en remplaçant gui par None lors de la création du 1<sup>er</sup> thread t1 :

```
t1 = Thread(target=run_simulation, args=(s, None))
```

## 2.3 Gestion du temps

Pour gérer le temps dans le code, on a utilisé une fonction « sleep » dans le thread du contrôleur, en lui passant un pas de temps « dt » défini dans le fichier « constante.py ». Le thread du contrôleur doit être mis à jour moins fréquemment que le thread de la simulation, c'est pourquoi on a ralenti délibérément la fréquence de mise à jour du contrôleur afin que la simulation reste plus rapide.

Ensuite, pour calculer le temps écoulé, on fait la différence entre le temps actuel obtenu grâce à la fonction « time.time() » et le temps de la dernière mise à jour. Ce calcul est nécessaire pour les calculs d'angle, de distance et de position.

En ralentissant la mise à jour du contrôleur par rapport à la simulation, on permet à la simulation de s'exécuter plus fréquemment. Le calcul du temps écoulé est essentiel pour effectuer des calculs basés sur le temps, notamment pour les angles, les distances et les positions.

## 2.4 Threading

Nous avons un thread pour l’IA un autre pour la simulation et le thread principal est celui pour l’interface graphique.

Les threads nous permettent d’exécuter plusieurs instructions en même temps et donc d’être indépendant.

Les threads sont créés dans la fonction run du fichier core.py qui se trouve dans le dossier module.

Pour pouvoir les utiliser nous importons la bibliothèque threading :

```
from threading import Thread
```

Nos threads prennent en paramètre une cible target qui correspond à la fonction à exécuter et les arguments de ces fonctions.

```
t1 = Thread(target=run_simulation, args=(s, gui))
t2 = Thread(target=run_strategie, args=(strat_carre,))
t1.start()
```



```
t2.start()
```

Le 1<sup>er</sup> thread t1 prend en paramètre la fonction run\_simulation et a comme paramètre s qui correspond à la simulation créer et gui à l'interface graphique (qu'on peut remplacer par None si l'on veut utiliser le robot réel).

Le 2<sup>ème</sup> thread prend en paramètre la fonction run\_strategie à exécuter qui a comme paramètre la stratégie que l'on souhaite faire. Ici il s'agit de la stratégie strat\_carre.

Le 3<sup>ème</sup> Thread est le thread principal et il s'agit de l'interface graphique.

### 3 Fonctionnalités du robot

Le robot possède deux roues motrices, une gauche et une droite, un capteur de distance et une caméra.

Les roues sont contrôlables séparément, permettant d'avancer, reculer ou tourner en avançant ou sur place. Il est possible de choisir leur vitesse angulaire et de connaître de combien de degrés elles ont tournée.

Le capteur de distance permet de connaître à quelle distance est l'obstacle le plus proche, en face du robot, jusqu'à une distance de 8 mètres.

La caméra peut prendre des photos, ce qui peut permettre entre autres de reconnaître un obstacle ou suivre un chemin.

### 4 Stratégies disponibles

Le but de l'IA est de donner des ordres au robot et de récupérer les éventuels données (données du capteur...), l'IA est comme le cerveau, mais elle ne doit décider en rien dans notre simulation.

Chaque stratégie est une classe composée d'un constructeur, d'une méthode update() ainsi que d'une méthode stop()

Nous avons les stratégies suivantes :

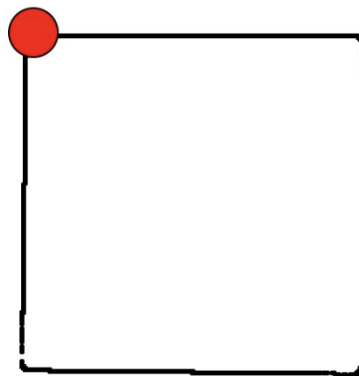
**StrategieAvance:** prend en paramètres la vitesse des roues, la distance à parcourir en millimètres et un proxy, elle se contente de d'appliquer la vitesse au roue grâce à set\_vitesse jusqu'à ce que la distance parcourue ait atteint la distance voulue.

**StrategieAngle :** est similaire mais s'arrête si le robot s'approche trop près d'un obstacle, à l'aide de la fonction get\_capteur\_distance du proxy.

**StrategieArretMur** : a pour arguments l'angle à effectuer en degrés, une vitesse et un proxy et fait tourner le robot jusqu'à ce que l'angle parcouru ait atteint l'angle souhaité.

**StrategieSeq** : est une stratégie séquentielle qui à partir d'un proxy et d'une liste de stratégies les exécute toutes l'une après l'autre. Nous l'utilisons dans notre projet pour faire un carré, en lui donnant pour liste une succession de StrategieAvance et StrategieAngle à 90 degrés.

La stratégie séquentielle nous permet de réaliser des figures géométriques et notamment un carré:



**Strategie Suivre Balise** : StrategieSuivreBalise est une classe de stratégie qui implémente un comportement de suivi de balise. Elle prend en compte les données fournies (data) et utilise un proxy (proxy) pour effectuer des actions.

## 5 Robot Réel

On accède au robot réel en se connectant à son réseau wifi avec le mot de passe GOPIGO2IN013, puis par ssh avec la commande « ssh pi@192.168.13.1 » puis en rentrant le mot de passe pi.

Ensuite, on transfère notre code au robot avec « scp -r mon\_repertoire pi@192.168.13.1: », on doit le faire à chaque fois qu'on a modifié notre code en local et qu'on veut le tester. Pour ce faire, on se place dans le fichier RoboTech puis on exécute le fichier projet.py dans le terminal correspondant au robot. Le robot réel se met donc à se déplacer en fonction de la stratégie voulue.

## 6 Conclusion

Notre projet sur la robotique, réalisé en collaboration avec notre groupe de travail en Python, a été une expérience enrichissante qui nous a permis de mettre en pratique les principes de la méthode Scrum et Agile. L'utilisation de la méthode Scrum nous a permis de structurer notre travail en sprints courts et focalisés, où nous avons défini des objectifs spécifiques à atteindre. Grâce à cette approche, nous avons pu constamment évaluer notre progression et nous ajuster en fonction des résultats obtenus. Cela nous a permis d'itérer rapidement et d'améliorer continuellement notre projet. L'approche Agile nous a permis en favorisant la communication et la collaboration au sein de notre équipe, de pouvoir exploiter au maximum les compétences de chacun et résoudre les problèmes de manière agile et flexible. Les réunions le mercredi matin, de revue et de rétrospective nous ont permis d'identifier les opportunités d'amélioration et de prendre des décisions éclairées tout au long du processus de développement. En conclusion, notre projet sur le thème de la robotique a été non seulement une occasion de développer nos compétences techniques, mais aussi d'apprendre les principes fondamentaux du travail en équipe. Ces méthodologies nous ont permis de travailler de manière plus efficace, collaborative et itérative, tout en nous adaptant aux changements. Nous sommes fiers des connaissances acquises grâce à cette expérience.