ARLAIS Julien

EL ATIA Youssef

LIN David

MASSET Mael

RoboTech – LU2IN013

Rapport final du projet Robotique



30 mai 2023

**Table des matières :**

[**1.** **Introduction du Projet** 3](#_Toc136264344)

[**1** **Description global notre code** 3](#_Toc136264345)

[**1.1** **Architecture du code** 3](#_Toc136264346)

[**1.2** **Description des fichiers** 4](#_Toc136264347)

[**1.3** **Tests** 4](#_Toc136264348)

[**2** **Choix de conception** 5](#_Toc136264349)

[**2.1** **Threading** 5](#_Toc136264350)

[**2.2** **Interface Graphique** 6](#_Toc136264351)

[**2.3** **Gestion du temps** 6](#_Toc136264352)

[**3** **Fonctionnalités du robot** 6](#_Toc136264353)

[**4** **Stratégies disponibles** 7](#_Toc136264354)

[**5** **Robot Réel** 7](#_Toc136264355)

[**6** **Conclusion** 7](#_Toc136264356)

# **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.

# **Description global notre code**

## **Architecture du code**

Dans le dossier RoboTech nous retrouvons les fichiers :

**projet.py**

**camera\_test**

**module :**

**controleur :**

**controleur.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 :**

**Compte rendue des séances**

Nous avons décider 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’entré 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.

## **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.

**controleur.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, StratégieAngle 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 proxys 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, d’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.

## **Tests**

Dans le dossier test nous retrouvons les fichiers :



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 unittestTestCase, et nous définissions 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**

# **Choix de conception**

## **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 bibliothèques 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 1er 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 2ème 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 3ème Thread est le thread principal et il s’agit de l’interface graphique.

## **Interface Graphique**

Nous avons fait le choix d’utiliser le module Tkinter intégré à la bibliothèque standard de Python.

import tkinter as tk

Après avoir importé la bibliothèque, la démarche consiste à créer, configurer et positionner les éléments graphiques (widgets) utilisés, à définir les fonctions /méthodes associés aux widgets, puis à entrer dans une boucle chargée de récupérer et traiter les différents événements pouvant se produire au niveau de l’interface graphique : mises à jour graphiques …

Pour cela, on créer une fenêtre Tkinter avec le titre « Interface Graphique », on créer un canevas dans la fenêtre en spécifiant les dimensions du canevas en fonction des dimensions de l’environnement. Ensuite, on créer un cercle de couleur rouge avec les coordonnées et le rayon du robot en utilisant la fonction « create\_cercle », une ligne aux coordonnées et dans la direction du robot avec la fonction « create\_line », le cercle représente le robot et la ligne représente la direction du robot. On parcourt tous les objets de l’environnement et on créer un cercle de couleur noir aux coordonnées de l’objet. Enfin, le canevas est empaqueté (pack) pour l'afficher à l'intérieur de la fenêtre.

Pour la mise à jour de l’interface graphique, on créer un cercle noir aux coordonnées du robot qui représente le tracer du robot. Ensuite on supprime le cercle et la ligne représentant le robot du canevas avec la fonction « delete » et on les recréer aux nouvelles coordonnées du robot.

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

t1 = Thread(target=run\_simulation, args=(s, gui))

## **Gestion du temps**

Pour la gestion du temps, on utilise un sleep dans le thread du contrôleur, on donne en argument un pas de temps dt présent dans constante.py. Le thread du contrôleur doit être mise à jour moins souvent que le thread de la simulation. Ensuite, Pour le calcul du temps écoulé on fait la différence entre le temps actuel avec « 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.

# **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 autre de reconnaître un obstacle ou suivre un chemin.

# **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().

**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.

**Strategie Suivre Balise :**

# **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.

# **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.