RoboTech – LU2IN013

Youssef El Atia

David Lin

Julien Arlais

Mael Masset

25 mai 2023

Projet Robotique



**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 disponible** 7](#_Toc136264354)

[**5** **Robot Réelle** 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**

**stop.py:**

**camera\_test**

**module :**

**controleur :**

**controleur.py :**

**modele :**

**element\_simulation.py :**

**proxy.py :**

**robot\_api.py :**

**root\_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.

## **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.

# **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 …

Voici la liste des widgets utiliser dans notre affichage graphique :

**Canvas** : zone de dessins graphiques.

**Scale** : glissière à plusieurs positions.

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**

# **Fonctionnalités du robot**

Donnez les fonctionnalités du robot (capteur, modifiez la couleur du robot par exemple …) et dire comment on a décidé de l’implémentez

# **Stratégies disponible**

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().

**Stratégie Stop :**

**Stratégie Avance :**

**Stratégie Angle :**

**Stratégie Arrêt Mur :**

**Stratégie Séquentielle :**

**Stratégie Suivre Balise :**

# **Robot Réelle**

# **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.