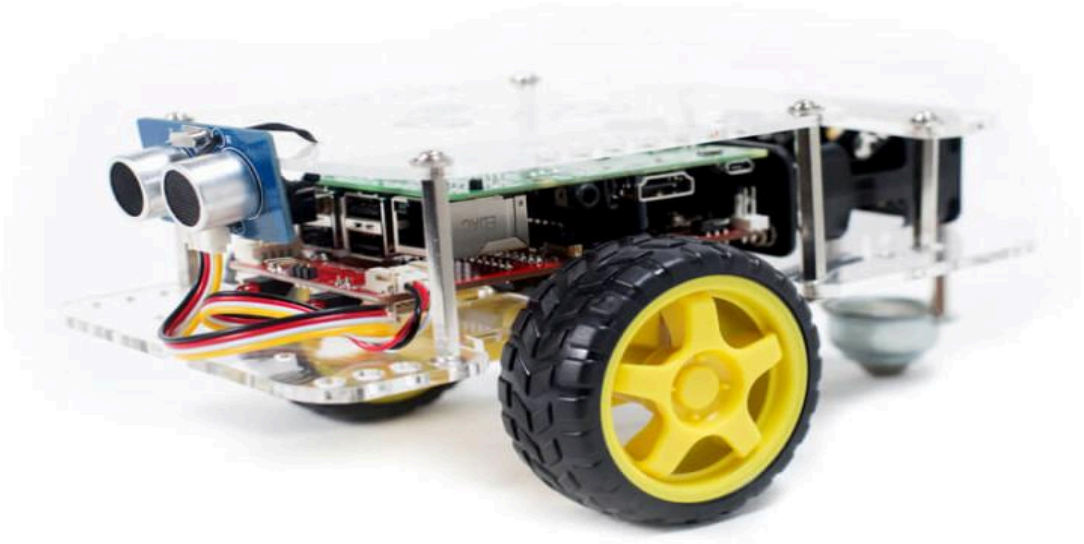
SCIENCES SORBONNE UNIVERSITÉ

2023 2024

RAPPORT FINAL
LU2IN013
PROJET ROBOTIQUE



NOMELESS®

Younes ARJDAL 21203492
Louisa OULD BOUALI 21100589
Lisa MOULA 21215357
Guillaume DUPART 28710554

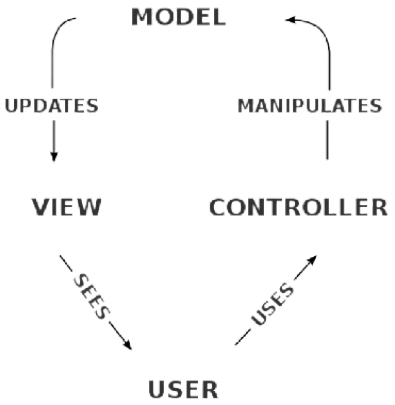
Coordinateurs d'U.E
Nicolas BASKIOTIS, Olivier SIGAUD

Table des matières

1.Présentation

1.1 Présentation du projet

L'objectif de notre projet est de créer une simulation pour un robot virtuel, capable de communiquer avec le robot réel Dexter via un proxy et d'exécuter différentes stratégies. Nous avons utilisé le modèle de conception Model-View-Controller (MVC) pour structurer notre application, ce qui améliore la modularité et la maintenabilité du code. Ce projet a été réalisé en Python, un langage de programmation qui permet une grande flexibilité et une gestion efficace des différents modules de notre application.



1.2 Organisation

- **GitHub**: Nous avons utilisé GitHub pour ce projet afin de faciliter la gestion du code source, la collaboration entre les membres de l'équipe, et le suivi des modifications grâce au contrôle de version.
- **Trello**:Nous avons utilisé Trello pour ce projet afin de gérer les tâches, suivre l'avancement du projet et organiser la gestion du temps. Trello permet une collaboration efficace en offrant une vue claire des responsabilités, des échéances et des priorités pour chaque membre de l'équipe.
- **Discord:** Nous avons utilisé Discord pour ce projet afin de faciliter la communication en temps réel entre les membres de l'équipe. Discord permet des discussions instantanées, des appels vocaux et le partage de fichiers, ce qui améliore la coordination et la collaboration tout au long du projet.

1.3 Présentation du code

1. Main_python :
    i. **Controller**:
        1. **Controlleur.py**: Ce fichier contient la classe Controleur, qui gère les stratégies d'un robot. Le contrôleur peut ajouter des stratégies à une liste, sélectionner et démarrer une stratégie spécifique, et exécuter ou arrêter les étapes de la stratégie courante.
        2. **Strategies.py**: Ce fichier contient des classes Python pour gérer différentes stratégies de contrôle d'un robot. Chaque classe définit des méthodes pour initier des actions, effectuer des étapes, et vérifier si une action est terminée, permettant ainsi au

robot d'avancer, tourner, suivre des balises, et détecter des obstacles.

ii. **Graphique**:

1. interface.py: Ce fichier contient le code utilisant Pygame pour créer et gérer une fenêtre de simulation robotique, affichant le robot, ses déplacements et les obstacles environnants. Il inclut des fonctions pour la rotation, le redimensionnement de l'image du robot, et la gestion des événements utilisateur pour contrôler la simulation.

iii. **irl:**

1. RobotReel.py**:** Ce fichier définit la classe `Robot2IN013`, qui encapsule les fonctionnalités d'un robot et de ses capteurs, permettant de contrôler les moteurs, lire les capteurs, et capturer des images à l'aide de Pygame et divers capteurs. Il inclut des méthodes pour démarrer et arrêter l'enregistrement, contrôler les moteurs et le servo, et lire les données des capteurs de distance et de mouvement.
2. mockup.py: Ce fichier contient la classe `Robot2I013Mockup`, qui simule les fonctionnalités de l'API d'un robot pour tester l'utilisation correcte des fonctions sans avoir besoin du matériel réel.
3. robotadaptateur.py: Ce fichier définit la classe **RobotAdaptateur**, qui adapte les commandes d'un robot réel pour fonctionner avec une simulation, en gérant les déplacements, la rotation, et la détection d'obstacles, tout en suivant la position et la direction du robot. Il inclut des méthodes pour mettre à jour la position, la distance parcourue, et capturer des images.

iv. **Model**:

1. Robot.py: Ce fichier définit la classe **Robot**, qui simule les déplacements, les rotations et la détection d'obstacles pour un robot dans un environnement. Il maintient un historique des positions, calcule les distances parcourues, et ajuste les directions et vitesses des roues pour modéliser le comportement du robot.

2. reel.py: Ce fichier définit la classe **Reel**, qui gère l'exécution des threads pour contrôler un robot réel. Il inclut des méthodes pour démarrer et exécuter un contrôleur de robot en boucle, en mettant à jour le robot à chaque étape, tout en permettant l'exécution parallèle grâce aux threads.

3. environnement.py: Ce fichier définit la classe **Environnemet**, qui gère les propriétés et comportements d'un environnement simulé pour un robot. La classe inclut des méthodes pour ajouter des obstacles, vérifier les positions et les collisions des objets, et mettre à jour les positions du robot et des obstacles en fonction du temps écoulé.
4. obstacle.py: Ce fichier définit la classe `Obstacle`, qui gère les coordonnées, les dimensions et la détection de collision pour simuler des obstacles dans un environnement.
5. simulation.py: Ce fichier définit la classe `Simulation`, qui gère l'exécution parallèle des threads pour le contrôleur, l'environnement et l'interface graphique d'une simulation robotique.
6. constante.py: Ce fichier définit des constantes pour la simulation, telles que les taux de rafraîchissement, les couleurs, les dimensions de l'environnement et de la fenêtre, ainsi que les paramètres initiaux du robot.

v. **Simu3D**:
   1. interface3d.py: Ce fichier initialise et exécute une simulation 3D utilisant Ursina pour modéliser un environnement robotique, où un robot et des obstacles sont visualisés, et des contrôles de caméra permettent différentes vues.

vi. Traitement_image.py: Ce fichier contient des fonctions OpenCV pour détecter des objets colorés dans une image, nettoyer les masques, et localiser les balises en trouvant le centre des plus grandes formes détectées.

vii. screen.py: Ce fichier enregistre une image, et utilise des fonctions de traitement d'image pour détecter et afficher la position d'une balise dans l'image.

```
compte rendu/
divers/
main python/
├── controller/
│   ├── controleur.py
│   └── strategies.py
├── graphique/
│   └── interface.py
├── irl/
│   ├── mockup.py
│   ├── robotadaptateur.py
│   └── robotreel.py
├── model/
│   ├── constante.py
│   ├── environnement.py
│   ├── reel.py
│   ├── obstacle.py
│   ├── robot.py
│   └── simulation.py
├── simu3d/
│   └── interface3d.py
traitement_image.py
static/
tests/
main_backup/
main_3d.py
main.py
```

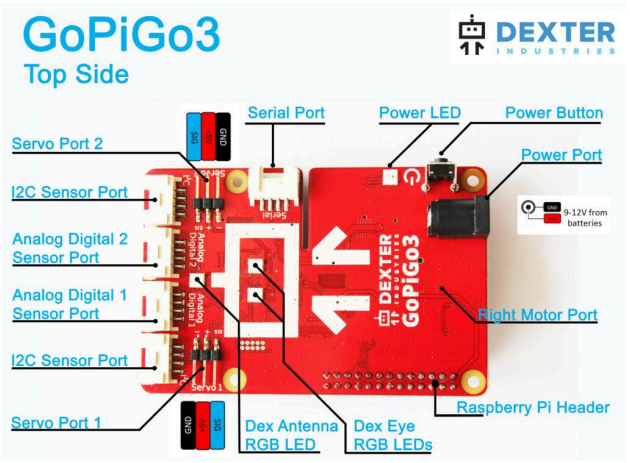
1.4 Présentation du robot

Le robot est équipé de deux roues motrices, une à gauche et une à droite, ainsi que d'un capteur de distance et d'une caméra. Les roues peuvent être contrôlées indépendamment, ce qui permet au robot d'avancer, de reculer ou de tourner sur place ou en mouvement. Il est possible de régler leur vitesse angulaire et de mesurer l'angle de rotation. Le capteur de distance détecte les obstacles situés devant le robot jusqu'à une portée de 8 mètres. La caméra, quant à elle, peut prendre des photos, facilitant ainsi la reconnaissance d'obstacles ou le suivi de trajectoire.

1.5 Robot IRL

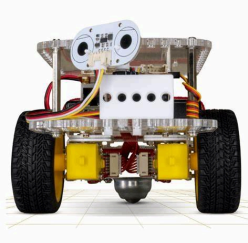
1. Fonctionnalité du robot

La classe **Robot2IN013** encapsule les fonctionnalités d'un robot **GoPiGo3**, intégrant divers capteurs et actionneurs. Les constantes définissent les dimensions et les caractéristiques des roues du robot.



Le constructeur **__init__** initialise le robot, la caméra, le servo, le capteur de distance et l'IMU, avec gestion des exceptions pour chaque composant. La méthode **stop** arrête le robot et éteint les LEDs.

Pour la capture d'images, **get_image** retourne la dernière image et **get_images** fournit les images capturées récemment. La méthode **set_motor_dps** fixe la vitesse des moteurs, **get_motor_position** lit leur position en degrés, et **offset_motor_encoder** ajuste les encodeurs des moteurs.



Le capteur de distance est lu avec **get_distance**, et **servo_rotate** contrôle l'angle du servo. **start_recording** lance l'enregistrement d'images en continu dans un thread séparé, avec **_stop_recording** pour l'arrêter. La méthode interne **_start_recording** gère la capture des images à une fréquence spécifiée.

La méthode **__getattr__** permet d'accéder aux fonctionnalités de la classe **EasyGoPiGo3**, comme le contrôle des LEDs.

2. Connexion au robot & execution du code

Pour nous connecter au robot, il est d'abord nécessaire de se connecter via un réseau Wi-Fi émis par un robot "Gopigo(1-2)" puis d'effectuer une connexion ssh (Secure Shell) à partir du terminal de commande. Par la suite nous utiliserons ipython qui est un environnement interactif via le terminal de commande et qui permet d'envoyer des requêtes au robot (définir les roues à une certaine vitesse et effectuer une rotation de x degrés de la tête du robot).
Une fois connecté au robot, il est nécessaire d'importer la classe **Robot2IN013** pour pouvoir utiliser les fonctionnalités du robot (vitesses et positions des roues…).
Notre but maintenant est d'envoyer nos propres lignes de code au robot. Pour cela on utilise la commande scp (Secure Copy Protocol) qui permet de copier à distance des fichiers d'un endroit vers un autre. Une fois le dossier envoyé, il nous faut nous re-connecter par ssh afin d'exécuter le code.

2. Implémentation de la 2D

2.1 Choix du FrameWork 2D

Nous avons hésité entre trois interfaces graphiques : Pygame, Kivy et Tkinter. En raison des limitations de performance de Kivy, notamment pour des applications non mobiles, et de sa lourdeur potentielle, nous avons écarté cette option. Le choix s'est ensuite restreint à Pygame et Tkinter.

Bien que Tkinter soit facile à utiliser, multiplateforme et ne nécessite pas l'installation de packages supplémentaires, ses limitations en termes de vitesse d'exécution nous ont conduits à privilégier Pygame. De plus, certains membres de notre équipe possèdent déjà des connaissances en Pygame, facilitant ainsi l'apprentissage pour ceux qui ne sont pas encore familiers avec cette interface graphique.

En conclusion, nous avons opté pour Pygame pour la réalisation de la simulation de notre projet de robotique.

2.2 Analyse du code

Pour un robot, il est crucial que la physique de la simulation 2D soit quasiment identique à celle de la réalité. Cela garantit une reproduction précise des mouvements, des interactions et des réponses du robot dans son environnement virtuel, ce qui permet de tester et de valider son comportement de manière fiable avant une mise en pratique réelle.

1. Physique des roues

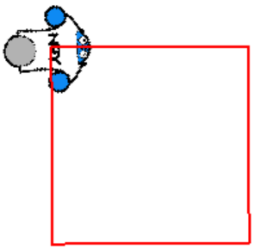
Dans la simulation 2D, afin d'être fidèle à la réalité, il nous faut simuler la rotation des roues. La fonction **update_position** assure un déplacement précis du robot en se basant sur les vitesses spécifiées pour ses roues gauche et droite. Elle réalise une série de calculs mathématiques pour mettre à jour sa position de manière cohérente. En considérant des paramètres tels que la vitesse linéaire moyenne, la rotation, la nouvelle direction et les coordonnées actualisées, cette fonction garantit un déplacement précis du robot selon les commandes des roues. La normalisation du vecteur direction est essentielle pour maintenir sa cohérence et faciliter la compréhension du mouvement du robot. Des exemples concrets sont fournis pour illustrer l'application de ces calculs, démontrant ainsi l'efficacité de cette approche mathématique pour le contrôle du déplacement du robot.

2. Utilisation du Threading

Pour la simulation 2D, nous utilisons trois threads distincts : un pour le contrôleur, un pour l'environnement et un pour l'interface utilisateur. Cette répartition permet une gestion optimale des différentes tâches, du contrôle des actions du robot à la gestion de l'environnement et à l'interaction avec l'utilisateur. En revanche, pour la simulation 3D, nous simplifions légèrement l'architecture en utilisant deux threads, dédiés au contrôleur et à l'environnement notamment en raison de la manière dont fonctionne Ursina. Cette approche permet toujours une exécution parallèle des tâches principales tout en réduisant la complexité de l'implémentation. Enfin, pour la simulation du robot réel, nous utilisons un seul thread pour le contrôleur car pas nécessaire de faire tourner un environnement en même temps et pas besoin d'interface.

3. Implémentation des stratégies

- **StratégieAvancer** : Cette méthode reçoit la vitesse des roues, la distance à parcourir en millimètres, et un proxy en tant que paramètres. Elle ajuste les roues à la vitesse spécifiée via **set_vitesse** et continue ainsi jusqu'à ce que la distance cible soit atteinte.
- **StratégieSequentiel**: Cette approche séquentielle prend un proxy et une liste de stratégies, qu'elle exécute de manière consécutive. Nous l'appliquons dans notre projet pour dessiner un carré, en utilisant une série de **StrategieAvancer** et de **StrategieTourner_reel** à 90 degrés.
  La stratégie séquentielle nous permet de réaliser des figures géométriques comme un carré:



- **StratégieTourner_reel** : Cette stratégie contrôle la rotation d'un robot en ajustant sa vitesse pour tourner à un angle spécifié, soit à droite soit à gauche, et vérifie si la rotation est complète. Elle initialise la rotation, exécute les étapes de la rotation, et arrête le mouvement une fois l'angle désiré atteint.

- **StratégieSuivre_Balise** : La classe StrategieSuivreBalise implémente un comportement de suivi de balise en utilisant les données fournies (data) et un proxy (proxy) pour effectuer des actions. Elle est conçue pour fonctionner uniquement en simulation et ne réagit qu'à certaines formes spécifiques.
- **Stratégie Cherche_Balise** : Définit une stratégie de recherche de balise dans un environnement en effectuant des rotations par étapes de 30 degrés. Si la balise est détectée, elle s'arrête et avance vers elle. Sinon, elle continue de tourner jusqu'à un angle maximal de 1000 degrés.
- **Stratégie Foncer Mur** :La stratégie FonceMur permet à un robot d'avancer en ligne droite jusqu'à détecter un obstacle devant lui, puis de s'arrêter lorsqu'il est suffisamment proche de cet obstacle.

2.3 Implémentation de tests

Les tests unitaires sont essentiels pour s'assurer que la simulation en 2D reproduit fidèlement la réalité. Ils vérifient chaque fonctionnalité individuelle, garantissant ainsi la conformité aux spécifications et la détection précoce des erreurs. En validant les mises à jour et les modifications, les tests unitaires assurent la stabilité et la fiabilité de la simulation. Dans notre cas, nous avons testé l'implémentation d'un environnement avec un robot donné ainsi que les stratégies associées, ce qui nous a permis de valider leur fonctionnement conformément aux attentes. En bref, les tests unitaires sont indispensables pour assurer la qualité et la précision de la simulation en 2D, ainsi que pour garantir son bon fonctionnement dans des situations réelles.

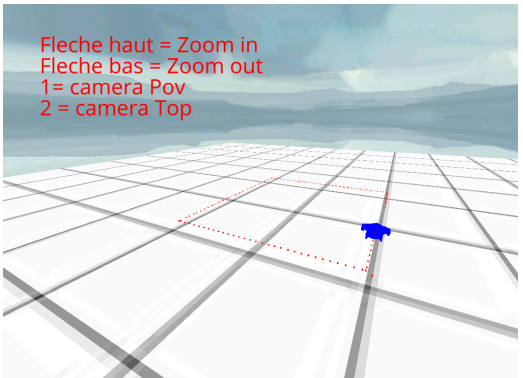
3. Implémentation de la 3D

3.1 Choix du FrameWork 3D

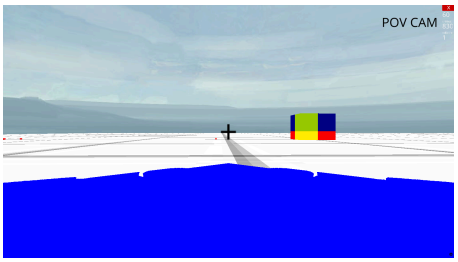
Après avoir analysé chaque interface 3D et examiné une par une les avantages et les inconvénients, nous avons constaté que Panda3D et Ursina sont deux options intéressantes. Cependant, Ursina semble être beaucoup plus conviviale et facile à utiliser, ce qui nous oriente vers ce choix.

3.2 Analyse du code

Nous avons créé un fichier nommé interface3d.py utilisant la bibliothèque Ursina en Python. Ce fichier crée un environnement de simulation où un robot peut se déplacer et interagir avec son



environnement. Il est nécessaire d'utiliser le fichier main_3d pour pouvoir avoir la simulation 3D.
L'utilisateur peut contrôler la caméra pour observer le robot sous différents angles (Vue d'en haut, vue du robot, vue éditeur).
Le fichier comprend également des fonctions pour exécuter le contrôleur et l'environnement dans des boucles séparées, permettant ainsi une interaction en temps réel avec la simulation.
Le contrôleur permet d'exécuter les mêmes stratégies que dans la simulation 2D (exemple : **faire_carre**).



Nous avons aussi implémenté une balise, permettant ainsi d'utiliser la stratégie **chercher_balise**, cette stratégie permet au robot d'effectuer un tour sur lui même, jusqu à trouver la balise (représenter dans la simulation par un cube contenant 4 couleurs : rouge, bleu, vert et jaune), puis le robot se dirige vers la balise.

4. Conclusion

Notre projet en robotique, réalisé avec notre groupe de travail en Python, a été une aventure enrichissante qui nous a permis d'appliquer les principes des méthodes Scrum et Agile. L'utilisation de Scrum a organisé notre travail en cycles courts et focalisés, avec des objectifs précis à atteindre. Cette méthode nous a permis de suivre notre progression en continu et de modifier notre approche en fonction des résultats, facilitant ainsi des itérations rapides et des améliorations constantes du projet. La méthode Agile a encouragé la communication et la coopération au sein de notre équipe, permettant à chacun de maximiser ses compétences et de résoudre les défis de manière flexible. Les réunions hebdomadaires de revue et de rétrospective, tenues chaque mercredi matin, ont été cruciales pour repérer des opportunités d'amélioration et prendre des décisions éclairées tout au long du développement.

En résumé, ce projet en robotique a été une occasion précieuse d'améliorer nos compétences techniques et 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 au cours de cette expérience.

5. Bibliographie

- **Unittest** : Cette bibliothèque est utilisée pour tester le programme.
- **pygame** : Nous utilisons cette bibliothèque externe pour créer des fenêtres 2D et afficher l'arène.
- **Ursina** : Nous utilisons cette bibliothèque externe pour créer des fenêtres 3D et afficher l'arène.
- **PIL** : Cette bibliothèque externe nous permet de traiter les images prises par le robot.
- **pygetwindow:** Cette bibliothèque est utilisée pour capturer une image de la fenêtre Ursina.
- **easygopigo3** : Il s'agit d'une bibliothèque externe pour contrôler de vrais robots. Nous ne l'avons pas installée.
- **GitHub** : https://github.com/Younesmrc/nameless.git
- **Trello** : https://trello.com/b/HxLaeJRG/lu2in013-projet