

Rapport du projet Robotique

Equipe HELMS

2023 - 2024



Enseignants :

- Nicolas BASKIOTIS
- Olivier SIGAUD

Étudiants :

- Sarah BEN MIMOUN
- Hugo Knecht (Absent)
- Micheal LI
- Lucie PAN
- Eric ZHENG

Table des matières

1	Introduction du Projet	3
2	Choix de conception	3
3	Description du code	4
3.1	Architecture du code	4
3.2	Description des fichiers dans le <code>src</code>	4
3.2.1	Le module <code>modele</code>	5
3.2.2	Le module <code>controller</code>	5
3.2.3	Le module <code>view</code>	5
3.2.4	Le module <code>robot</code>	6
4	Stratégie du robot	6
4.1	Strategie Basique	6
4.1.1	Go	6
4.1.2	TourneDeg	7
4.1.3	Stop	7
4.2	Méta-stratégie	8
4.2.1	StrategieSequentielle	8
4.2.2	StrategieIf	8
4.2.3	StrategieWhile	9
4.2.4	StrategieFor	10
4.2.5	StrategieTrouverBalise	10
5	Application et test des stratégies	11
5.1	Objectif 1 : Tracer un carré	11
5.2	Objectif 2 : Foncer sur un mur/obstacle sans le toucher	12
5.3	Objectif 3 : Reconnaissance de la balise	13
5.4	Autre : Éviter les obstacles	13
6	Robot IRL	15
7	Difficultés rencontrés	16
8	Conclusion	16

1 Introduction du Projet

Pour accéder au dépôt GitHub, cliquez [ici](#).

Le projet que nous avons effectué ce semestre consistait à développer un robot autonome capable d'effectuer des tâches basiques telles que : faire un carré, utiliser un capteur de distance et une caméra. Pour atteindre cet objectif, nous avons commencé par la création d'une simulation en python orientée objet. Cette simulation nous a permis de visualiser notamment grâce à **Tkinter** le comportement du robot dans un environnement virtuel, de tester différentes stratégies et de valider notre implémentation avant de passer à la phase de déploiement sur un véritable robot physique.

Nous avons appris dans un premier temps à utiliser gitHub pour pouvoir travailler en groupe sur le projet et nous nous sommes initiés au langage python orienté objet.

Nous avons dans les premières semaines opté pour un retour sur le terminal et nous avons ensuite basculé pour un retour sur l'interface graphique **Tkinter**. Et enfin nous avons ajouté la 3D avec **Panda3D**.

Nous nous sommes répartis le travail grâce à **Trello** où nous pouvions créer des tâches à effectuer pour la semaine et ainsi avoir une vue d'ensemble du travail à fournir.

2 Choix de conception

Dans le cadre de notre cours de Projet Robotique, nous avons entrepris une démarche ambitieuse visant à concevoir et développer un robot basé sur l'architecture Modèle-Vue-Contrôleur (MVC). Cette approche, largement utilisée dans le développement logiciel, offre un cadre structuré pour la conception de systèmes complexes en séparant les préoccupations liées aux données, à la logique de traitement et à l'interface utilisateur.

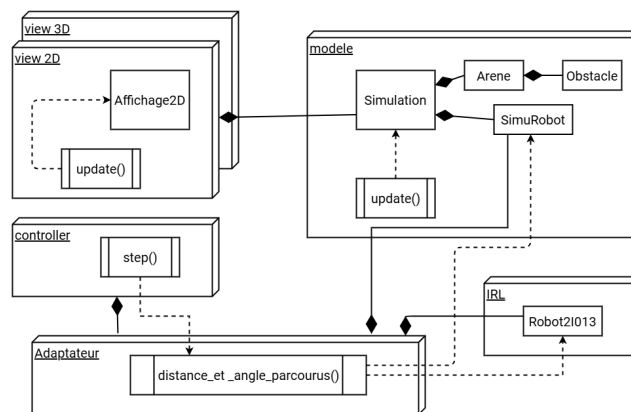


FIGURE 1 – État des lieux

Pour atteindre nos objectifs, nous avons utilisé plusieurs bibliothèques :

- L'affichage des messages : Logging permettant de garder un historique des événements et de l'état du système, et de détecter rapidement les anomalies.
- La simulation en 2d : Tkinter pour l'affichage pour sa simplicité d'utilisation et sa portabilité.
- La simulation en 3d : Panda3d pour l'affichage (simple à utiliser avec Python) et OpenCV pour l'enregistrement de image.
- La reconnaissance de la balise : Pillow et NumPy

Nous avons également utilisé des classes héritées pour structurer notre code de manière efficace et réutilisable, favorisant ainsi une meilleure modularité et maintenance du système.

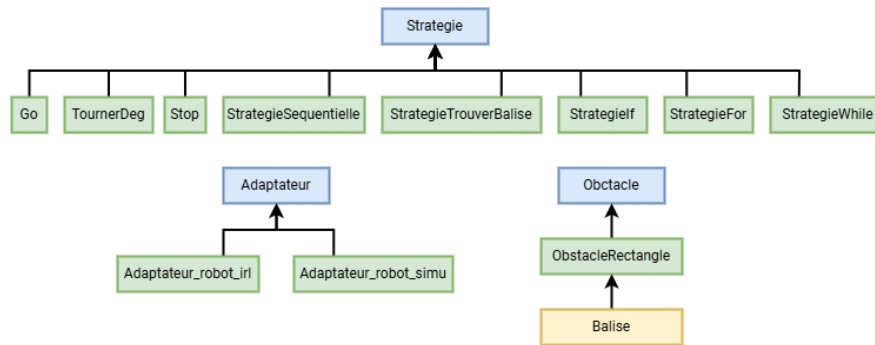


FIGURE 2 – Héritages existant dans le projet

3 Description du code

3.1 Architecture du code

Dans le dépôt GitHub, nous pouvons observer cinq dossiers principaux et plusieurs fichiers :

- Le fichier `main.py` permet d'exécuter la simulation du robot virtuel.
- Le fichier `main_robot_irl.py` permet d'exécuter la simulation du robot réel.
- Le dossier `strategies_prefaites` contient des stratégies déjà pré-écrites, comme demander au robot d'avancer ou de faire un carré.
- Le dossier `autre_documents` contient la preuve physique utilisée dans notre simulation et une fiche de renseignement sur les membres de l'équipe.
- Le dossier `videos_demo_strategie` contient les vidéos des tests réalisés sur le robot réel.
- Le dossier `docs` contient la docstring du projet en `html`.

3.2 Description des fichiers dans le `src`

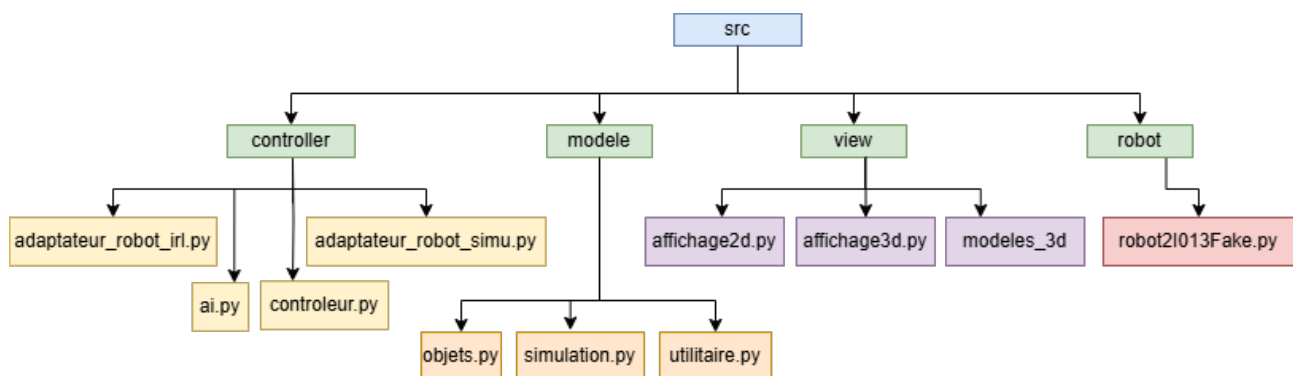


FIGURE 3 – Architecture du dossier `src`

Le dossier `src` est subdivisé en quatre modules : un module pour les préoccupations liées aux données (`modele`), un module pour la logique de traitement (`controller`), un module pour l'interface utilisateur (`view`) et un module pour l'API du robot (`robot`).

3.2.1 Le module `modele`

Le module `modele` possède un sous-module contenant toutes les classes d'objets (`robot`, `obstacles`, `arène`, etc.), un sous-module `utilitaires.py` contenant les fonctions mathématiques, et un sous-module `simulation.py` qui simule le robot.

3.2.2 Le module `controller`

Le module `controller` possède un sous-module `controleur.py`, un sous-module `ai.py` où se trouvent les stratégies de haut niveau et de bas niveau, un sous-module `adaptateur_robot_irl.py` (resp. `adaptateur_robot_simu.py`) qui fait l'intermédiaire entre le robot réel (resp. simulé) et le contrôleur.

3.2.3 Le module `view`

Le module `view` est composé d'un dossier `modeles_3d` et de deux sous-modules : `affichage2d.py` et `affichage3d.py`. Dans le dossier `modeles_3d`, vous trouverez les modèles 3D de la simulation (`robot`, `obstacles`, `arène` et `balise`) qui sont soit en `.glb` soit en `.egg`. Il y a également les patrons de ces objets dans `modeles_3d/blender`.

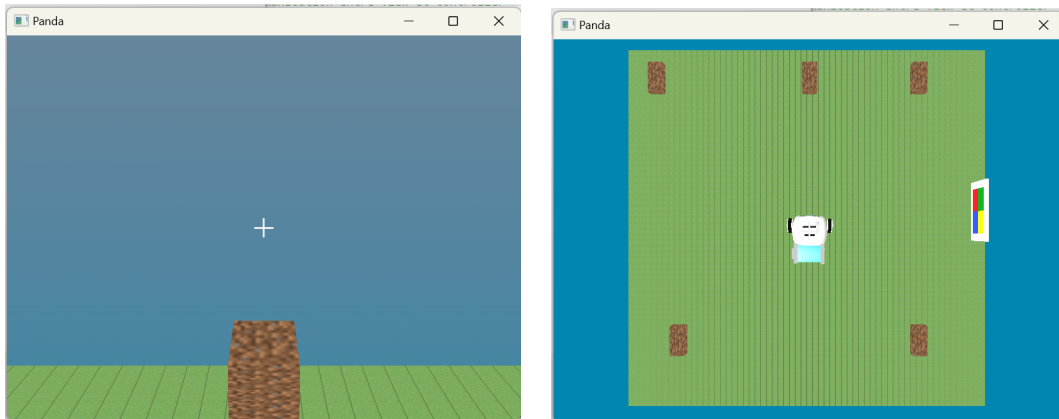
- Le sous-module `affichage2d.py` permet d'afficher en temps réel une simulation 2D à l'écran (cf. FIGURE 4). Les données du robot sont affichées en haut à gauche. Vous pouvez exécuter différentes stratégies en saisissant des valeurs pour les différentes variables. Certaines valeurs sont interdites et l'interface préviendra l'utilisateur par un message rouge. Il y a également un bouton permettant de remettre le robot à sa position initiale. Si la stratégie que vous souhaitez exécuter n'est pas présente, référez-vous à la section Application et test des stratégies



FIGURE 4 – Représentation en 2D du robot et de son environnement

- Le sous-module `affichage3d.py` permet d'afficher en temps réel une simulation 3D à l'écran (cf. FIGURE 5). Vous pouvez changer de point de vu (passer de la 3D en 2D) en appuyant sur la touche `esc` du clavier. Vous pouvez également pivoter (resp. incliner) la camera avec les touches `droite/gauche` (resp. `haut/bas`). Les stratégies déjà présentes

sur l'interface 2D peuvent csimuler le robot. Si la stratégie que vous souhaitez exécuter n'est pas présente, référez-vous à la section Application et test des stratégies



(a) Point de vue du robot

(b) Vue de dessus (2D)

FIGURE 5 – Représentation en 3D du robot et de son environnement

3.2.4 Le module robot

Le module `robot` contient une API. Cette API est déployée lorsque l'API du robot réel n'a pas été trouvée lors d'une exécution.

4 Stratégie du robot

Nous avons développé plusieurs stratégies permettant de donner des ordres au robot réel ou simulé. Il existe des stratégies basiques et des stratégies imbriquées (des méga-stratégie). Chaque stratégie est une classe fille de la classe mère de `Strategie`. Leur structure sont identiques :

- `def __init__` permettant d'initialiser la stratégie.
- `def start` permettant de lancer la stratégie.
- `def stop` permettant de savoir quand elle est finie
- `def step` permettant de faire un pas dans la stratégie

Les stratégies sont exécutées par le `Contrôleur` avec la fonction `start`.

Pour créer sa propre stratégie et l'exécuter vous pouvez combiner les différentes classes de stratégie au niveau du `main` et utiliser la fonction `add_strat` du `Contrôleur` pour dire au robot d'effectuer la stratégie.

4.1 Strategie Basique

4.1.1 Go

Lorsque la stratégie `Go` est lancée avec les variables `adaptateur`, `distance`, `v_ang_d`, `v_ang_g`, et `active_trace`, l'adaptateur va mettre à jour le moteur gauche (resp. droit) du robot à `v_ang_d` (resp. `v_ang_g`) avec la fonction `set_vitesse_roue`. Puis, à chaque pas de temps, l'adaptateur vérifie si la `distance` à parcourir est atteinte avec (`distance_parcourue`). Par

ailleurs, si la variable `active_trace` est vraie, alors le robot tracera son parcours (cela ne fonctionnera qu'en 2D). (cf FIGURE 6.a)

4.1.2 TourneDeg

Lorsque la stratégie `TournerDeg` lancée avec les variables `adaptateur`, `angle`, `v_ang`, et `active_trace`, l'adaptateur va mettre à jour le moteur gauche et droit du robot à `v_ang`. Puis à chaque pas de temps l'adaptateur vérifie si l'angle à faire est atteint avec (`distance_parcourue`). Idem pour le traçage. (cf FIGURE 6.b)

4.1.3 Stop

La stratégie `Stop` est une stratégie qui arrête le robot. Elle est utilisée pour arrêter le mouvement des roues du robot en mettant leur vitesse à zéro. Cette stratégie est toujours considérée comme terminée dès qu'elle commence. (cf FIGURE 6.c)

```

24 class Go(Strategie):
25     def start(self):
26         """commencer la strategie"""
27         self.logger.info("Starting Go strategy")
28         self.adaptateur.active_trace(self.active_trace)
29         self.adaptateur.set_vitesse_roue(self.v_ang_d, self.v_ang_g)
30         self.pos_ini = self.adaptateur.distance_parcourue()
31         self.parcouru = 0
32
33     def stop(self):
34         """Verifier si la strategie est fini ou non
35         :return: True si la strategie est finie, False sinon"""
36         self.logger.info(f"distance parcourue {self.parcouru}, distance {self.distance}")
37         return math.fabs(self.parcouru) >= math.fabs(self.distance)
38
39     def step(self):
40         """pas de la strategie """
41         self.parcouru += self.adaptateur.distance_parcourue()
42         if self.stop():
43             self.adaptateur.stop()
44             self.logger.info("Go strategy finished")
45             return

```

(a)Code de la stratégie Go

```

69 class TournerDeg(Strategie):
70     def start(self):
71         """commencer la strategie"""
72         self.logger.info("Starting TournerDeg strategy")
73         #self.logger.info(f"adapateur angle {self.adaptateur.angle}")
74         self.adaptateur.active_trace(self.active_trace)
75         if self.angle is None :
76             if self.adaptateur.angle < 90:
77                 self.angle = 90 - self.adaptateur.angle
78             else :
79                 self.angle = self.adaptateur.angle-90
80         if self.angle > 0:
81             self.v_ang_d, self.v_ang_g = self.v_ang, -self.v_ang
82             self.adaptateur.set_vitesse_roue(self.v_ang, -self.v_ang)
83         else:
84             self.v_ang_d, self.v_ang_g = -self.v_ang, self.v_ang
85             self.adaptateur.set_vitesse_roue(-self.v_ang, self.v_ang)
86         self.pos_ini = 1. * self.adaptateur.angle_parcouru()
87         self.parcouru = 0
88
89     def stop(self) -> bool:
90         """Verifier si la strategie est fini ou non
91         :return: True si la strategie est finie ou False sinon"""
92         self.logger.info(f"angle parcourue {self.parcouru}, angle {self.angle}")
93         if self.angle == 0 or math.fabs(self.parcouru) >= math.fabs(self.angle):
94             self.adaptateur.stop()
95             return True
96         return False
97
98     def step(self):
99         """pas de la strategie"""
100         self.parcouru += self.adaptateur.angle_parcouru()
101         if self.stop():
102             self.logger.info("TournerDeg strategy finished")
103             return

```

(b)Code de la stratégie TournerDeg

```

162 class Stop(Strategie):
163     def start(self):
164         """commencer la strategie"""
165         self.logger.info("Starting STOP strategy")
166         self.adaptateur.set_vitesse_roue(0, 0)
167
168     def stop(self) -> bool:
169         """Verifier si la strategie est finie
170         :return: True si la strategie est finie, False sinon"""
171         return True
172
173     def step(self):
174         pass

```

(c)Code de la stratégie Stop

FIGURE 6 – Code des stratégies de base

4.2 Méta-stratégie

4.2.1 StrategieSequentielle

Cette approche permet de combiner plusieurs stratégies de base en une séquence, offrant ainsi au robot la possibilité de suivre une série d'actions (**steps**) prédéfinies de manière ordonnée. Cela rend la stratégie modulable et flexible, chaque action ou groupe d'actions étant en-capsulé dans des objets de stratégie distincts.

La fonction **stop** vérifie si toutes les étapes de la séquence ont été exécutées. Elle renvoie **True** si l'index de l'étape actuelle (**current_step**) est supérieur ou égal à la longueur de la liste des étapes (**len(self.steps)**), ce qui signifie que toutes les étapes ont été exécutées. **False** sinon.

La fonction **step** exécute un pas de la stratégie séquentielle. Si la stratégie est terminée (**self.stop()** retourne **True**), elle ne fait rien. Sinon, elle exécute un pas de l'étape actuelle (**self.steps[self.current_step].step()**). Si l'étape actuelle est terminée (**self.steps[self.current_step].stop()** retourne **True**), elle passe à l'étape suivante (**self.current_step += 1**) et démarre cette nouvelle étape si elle existe (**self.steps[self.current_step].start()**). (cf FIGURE 7)

```

122 class StrategieSequentielle(Strategie):
145     def stop(self) -> bool:
146         """Verifier si la strategie est finie
147         :return: True si la strategie est finie, False sinon"""
148         return self.current_step >= len(self.steps)
149
150     def step(self):
151         """pas de la strategie sequentielle """
152         if self.stop():
153             return
154         self.steps[self.current_step].step()
155         if self.steps[self.current_step].stop():
156             self.current_step += 1
157             if self.current_step < len(self.steps):
158                 self.steps[self.current_step].start()

```

FIGURE 7 – Code de la stratégie **StrategieSequentielle**

4.2.2 StrategieIf

La stratégie **StrategieIf** permet de conditionner l'exécution entre deux stratégies en fonction d'une condition spécifique, permettant une flexibilité dans le comportement du robot basé sur ses capteurs.

La fonction **start** initialise la stratégie conditionnelle en choisissant la stratégie à exécuter en fonction de la condition. Nous vérifions la distance mesurée par l'adaptateur (avec **get_distance()**) et nous comparons cette valeur avec la condition. Si la distance est supérieure ou égale à la condition, **stratA** est choisie. Sinon, **stratB** est choisie. (cf FIGURE 8)


```

187 class StrategieIf(Strategie):
188     def start(self):
189         """commencer la strategie"""
190         self.logger.info("Starting condition strategy")
191         #self.logger.info(f"capteur distance : {self.adaptateur.get_distance()}")
192         if self.adaptateur.get_distance() >= self.condition:
193             self.strat_a_faire = self.stratA
194         else:
195             self.strat_a_faire = self.stratB
196         self.strat_a_faire.start()
197
198     def stop(self) -> bool:
199         """Verifier si la strategie est finie
200         :return: True si la strategie est finie, False sinon"""
201         # Verifier si strat_a_faire n'est pas vide
202         if self.strat_a_faire is not None:
203             return self.strat_a_faire.stop()
204
205     def step(self):
206         """pas de la strategie sequentielle """
207         if self.strat_a_faire is not None:
208             self.strat_a_faire.step()

```

FIGURE 8 – Code de la stratégie StrategieIf

4.2.3 StrategieWhile

La stratégie **StrategieWhile** permet de répéter une stratégie tant qu'une condition spécifique est vérifiée. Lorsque la condition n'est plus vérifiée, la stratégie s'arrête. Cette structure permet de gérer des boucles de comportement basées sur des conditions spécifiques, les mesures de capteurs.

Le **stop** vérifie si la condition est toujours vérifiée pour continuer la stratégie. Elle renvoie **True** si la distance mesurée par l'adaptateur (**get_distance()**) est inférieure à la condition. Nous arrêtons les moteurs du robot (**self.adaptateur.stop()**). Sinon si la stratégie **strat** est terminée (**self.strat.stop()** retourne **True**), elle est redémarrée (**self.strat.start()**). **False** sinon.

(Exemples d'application : la 2D et la 3D)

```

232 class StrategieWhile(Strategie):
233     def stop(self) -> bool:
234         """Verifier si la strategie est finie
235         :return: True si la strategie est finie, False sinon"""
236         self.logger.info(f"capteur distance : {self.adaptateur.get_distance()}")
237         if self.adaptateur.get_distance() < self.condition :
238             self.logger.info(f"STOP !!! ")
239             self.adaptateur.stop()
240             return True
241         else :
242             if self.strat.stop():
243                 self.strat.start()
244             return False
245
246     def step(self):
247         """pas de la strategie sequentielle """
248         if self.stop():
249             return
250         self.strat.step()

```

FIGURE 9 – Code de la stratégie StrategieWhile

4.2.4 StrategieFor

La stratégie **StrategieFor** encapsule une stratégie et la répète plusieurs fois de manière séquentielle. Elle suit et contrôle le nombre de répétitions, assurant que la stratégie est exécutée le nombre de fois demandé avant de s'arrêter.

```

161 class StrategieFor(Strategie):
177 def start(self):
178     """commencer la strategie"""
179     self.logger.info("Starting For strategy")
180     self.cpt = 0
181     self.strategie.start()
182
183 def stop(self) -> bool:
184     """Vérifier si la stratégie est finie
185     :return: True si la stratégie est finie, False sinon"""
186     if self.strategie.stop():
187         self.cpt += 1
188         if self.cpt >= self.nombre_repetition:
189             return True
190         self.strategie.start()
191     return False
192
193 def step(self):
194     """pas de la strategie sequentielle """
195     if self.stop():
196         return
197     self.strategie.step()

```

FIGURE 10 – Code de la stratégie **StrategieFor**

4.2.5 StrategieTrouverBalise

La stratégie **StrategieTrouverBalise** est une stratégie pour rechercher une balise en capturant et en analysant des images à différents angles. Elle utilise un adaptateur pour contrôler le robot et un **servo_moteur** pour ajuster l'angle de vision (**angle**). La stratégie s'arrête lorsqu'elle trouve la balise aide de la fonction **_process_image** ou lorsque le nombre maximal de captures est atteint.

```

272 class StrategieTrouverBalise(Strategie):
291 def start(self):
292     """Commencer la strategie"""
293     self.logger.info("Starting StrategieTrouverBalise strategy")
294     self.adaptateur.start_recording()
295     self.adaptateur.servo_rotate(self.angle)
296
297 def stop(self) -> bool:
298     """Vérifier si la stratégie est finie
299     :return: True si la stratégie est finie, False sinon"""
300     if self._process_image():
301         self.logger.info(f"Balise trouvée: {self.num}")
302         return True
303     self.logger.info(self.cpt)
304     return self.cpt > self.seuil_cpt
305
306 def step(self):
307     """Pas de la stratégie"""
308     if self.stop():
309         self.adaptateur.stop_recording()
310         return
311     self.image = self.adaptateur.get_image()
312     if self.image is not None:
313         self.angle += self.pas_angle
314         self.adaptateur.servo_rotate(self.angle)
315         self.cpt += 1

```

FIGURE 11 – Code de la stratégie **StrategieTrouverBalise**

5 Application et test des stratégies

5.1 Objectif 1 : Tracer un carré

Le premier objectif de cette UE était de demander au robot de tracer un carré. Pour cela nous avons défini une stratégie (`StrategieWhile`) pour avancer jusqu'à ce qu'une collision soit détectée, où la détection de collision est déterminée par la condition `seuil_collision`.

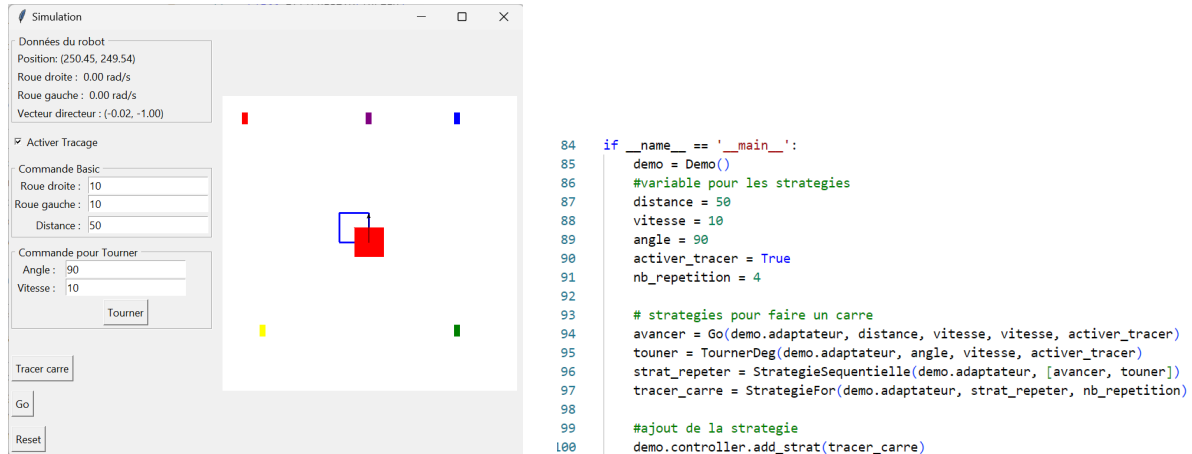
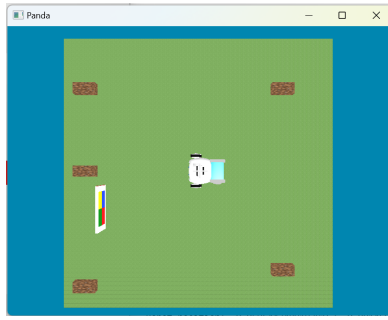
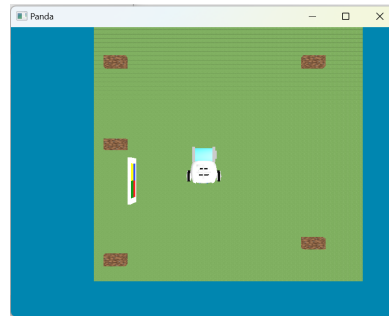
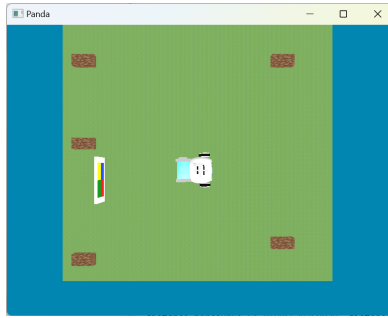


FIGURE 12 – Robot qui trace un carré

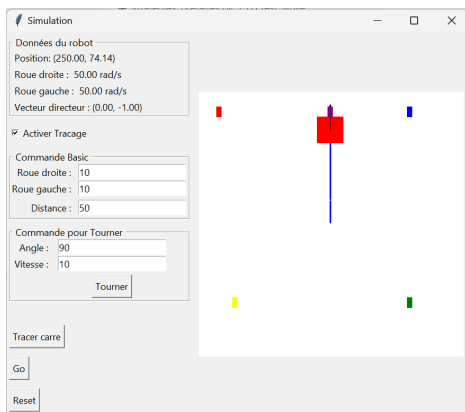
(a) Avant l'affectation de la stratégie t_0 (b) Position du robot à un $t_1 = t_0 + \Delta t$ (c) Position du robot à un $t_2 = t_1 + \Delta t$ 

(d) Fin de la stratégie

FIGURE 13 – Évolution dans le temps lorsque robot effectue un carré

5.2 Objectif 2 : Foncer sur un mur/obstacle sans le toucher

Le deuxième objectif de cette UE était de demander au robot d'avancer vers un mur et de s'arrêter juste avant de le toucher. Pour cela nous avons défini une stratégie séquentielle (*StrategieSequentielle*) qui avance puis tourne, et cette stratégie est répétée 4 de fois pour tracer un carré, (*StrategieFor*).



(a) Simulation de la stratégie en 2d

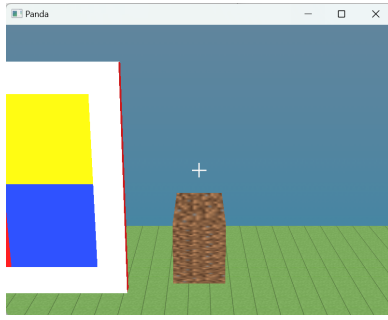
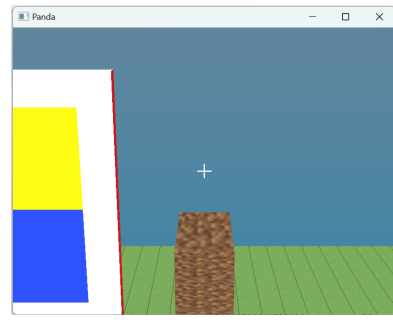
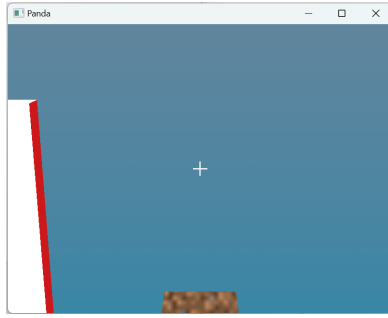
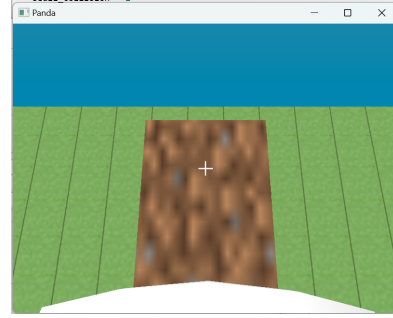
```

84 if __name__ == '__main__':
85     demo = Demo()
86     #variable pour les strategies
87     distance = 50
88     vitesse = 50
89     seuil_collision = 1
90
91     # strategies aller jusqu'au mur
92     avancer = Go(demo.controller.adaptateur, distance, vitesse, vitesse, True)
93     avancer_jusquau_mur = StrategieWhile(demo.adaptateur, avancer, seuil_collision)
94
95     #ajout de la strategie
96     demo.controller.add_strat(avancer_jusquau_mur)

```

(b) Code permettant d'effectuer la simulation

FIGURE 14 – Robot qui fonce sur un obstacle sans le toucher

(a) Avant l'affectation de la stratégie t_0 (b) Position du robot à un $t_1 = t_0 + \Delta t$ (c) Position du robot à un $t_2 = t_1 + \Delta t$ 

(d) Fin de la stratégie, vue avec caméra qui a été inclinée

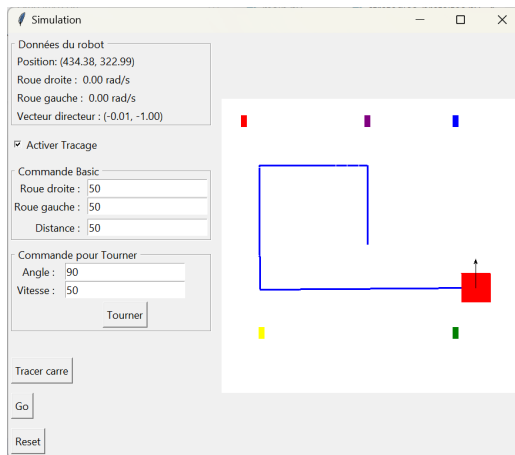
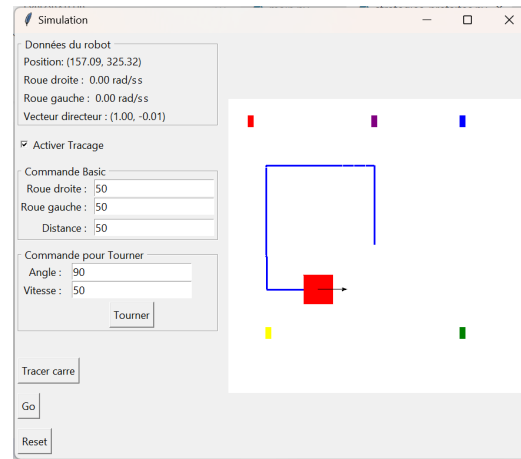
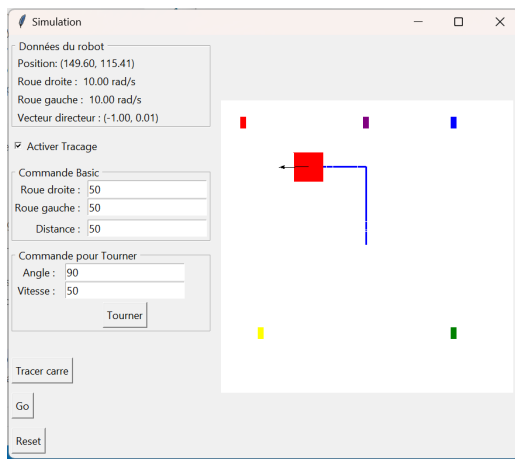
FIGURE 15 – Évolution dans le temps lorsque robot effectue une stratégie while

5.3 Objectif 3 : Reconnaissance de la balise

Le dernier objectif de cette UE était d'amener le robot à identifier une balise. À cette fin, nous avons élaboré une stratégie nommée `StrategieTrouverBalise`, qui analyse régulièrement le champ de vision du robot. À chaque intervalle défini (`pas_angle`), la stratégie vérifie la présence de la balise dans le champ de vision. Bien que cet objectif n'ait pas été accompli avec succès sur le robot simulé, il a été pleinement réalisé sur le robot réel. Vous pouvez trouver plus de détails dans la section Robot irl.

5.4 Autre : Éviter les obstacles

Nous avons décidé de définir un nouvel objectif : créer une stratégie qui guide le robot en ligne droite jusqu'à ce qu'il rencontre un obstacle, puis le tourne vers la gauche de 90 degrés. Ensuite, le robot reprend sa trajectoire en ligne droite jusqu'à ce qu'il rencontre un nouvel obstacle. Cette séquence d'actions doit être répétée quatre fois. Pour atteindre cet objectif, nous avons combiné les fonctionnalités des méta-stratégies `StrategieWhile` et `StrategieFor`, ainsi que les actions `Go` et `TournerDeg`.

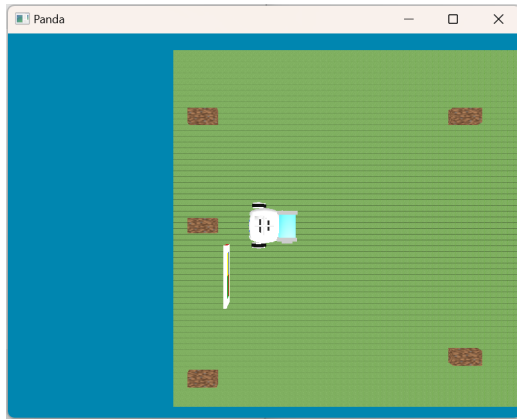
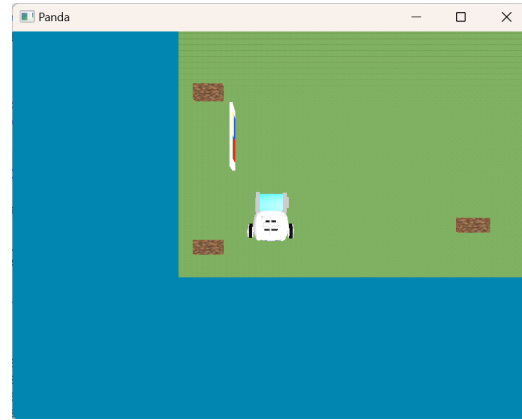
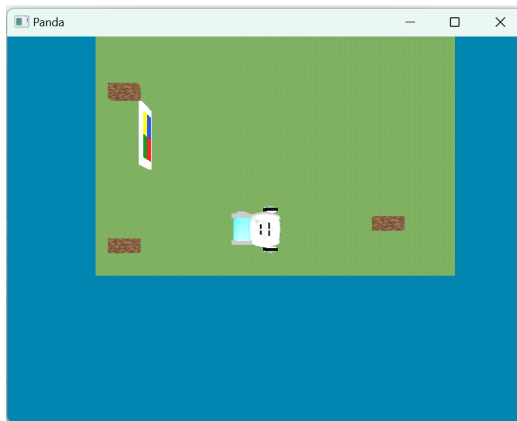
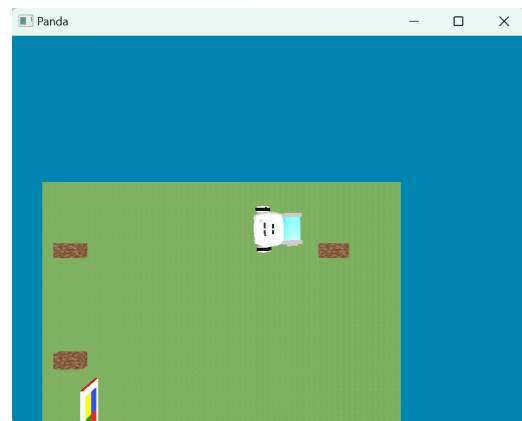


```

84 if __name__ == '__main__':
85     demo = Demo()
86     #variable pour les strategies
87     distance = 50
88     vitesse = 10
89     angle = 90
90     activer_tracer = True
91     nb_repetition = 4
92
93     # strategies pour faire un carre
94     avancer = Go(demo.adaptateur, distance, vitesse, vitesse, activer_tracer)
95     tourner = TournerDeg(demo.adaptateur, angle, vitesse, activer_tracer)
96     strat_repeter = StrategieSequentielle(demo.adaptateur, [avancer, tourner])
97     tracer_carre = StrategieFor(demo.adaptateur, strat_repeter, nb_repetition)
98
99     #ajout de la strategie
100    demo.controller.add_strat(tracer_carre)

```

FIGURE 16 – Robot doit avancer vers l'obstacle puis tourne à gauche

(a) Avant l'affectation de la stratégie t_0 (b) Position du robot à un $t_1 = t_0 + \Delta t$ (c) Position du robot à un $t_2 = t_1 + \Delta t$ 

(d) Fin de la stratégie, vue avec caméra a été inclinée

FIGURE 17 – Évolution dans le temps lorsque robot effectue une stratégie while

6 Robot IRL

Le robot réel utilisé dans notre projet est possède :

- Un Raspberry Pi
- Une carte contrôleur (Arduino)
- Deux moteurs encodeurs pour le contrôle des roues
- 3 senseurs : une caméra, un capteur de distance, un accéléromètre

Nous l'avons utilisé pour tester les stratégies que nous avons programmé. Vous trouverez ci-dessous les vidéos :

Pour visionner le test de rotation : le robot tourne à 90 degrés vers la droite. Cliquez [ici](#).

Pour visionner le test de stratégie conditionnelle :

- Si le robot est à plus de 100 mm, il ne fait rien ; sinon, il avance de 20 mm. Cliquez [ici](#).
- Idem mais en inversant les conditions. Cliquez [ici](#).

Pour visionner le test d'une stratégie while : tant que le robot est à plus de 50 mm d'un obstacle, il avance de 20 mm.

- Cas 1 : le robot est à 150 mm de son obstacle. Cliquez [ici](#).
- Cas 2 : le robot est à moins de 50 mm de son obstacle. Cliquez [ici](#).

Pour visionner le test de la réalisation d'un carrée de 150 mm. Cliquez [ici](#).

Pour visionner le test de reconnaissance de la balise : le robot regarde de la droite vers la gauche, commençant à 20° et pivotant à chaque fois de 10°. Cliquez [ici](#).

Pour visionner le test trouver la balise et d'arrêt près d'elle, cliquez [ici](#).

7 Difficultés rencontrés

Nous avons été confrontés à plusieurs défis tout au long des semaines de travail.

Dans un premier temps, nous avons dû nous familiariser avec l'utilisation de Trello. Cela nous a demandé un certain temps pour pouvoir estimer correctement la durée de chaque tâche et organiser leur répartition de manière à éviter la présence de sous-tâches inutiles.

Ensuite, nous avons rencontré des difficultés de factorisation de notre code. Les classes que nous avons implémentées étaient trop imbriquées les unes dans les autres, ce qui compliquait la gestion du programme si nous souhaitions désactiver certaines fonctionnalités.

La perte de membres de l'équipe en cours de projet peut représenter un défi significatif pour le reste de l'équipe. Lorsqu'un membre quitte le groupe, cela peut déséquilibrer la répartition des tâches et entraîner une perte de connaissances spécifiques qu'il ou elle avait apportées au projet. Malheureusement, c'est exactement ce qui s'est produit dans notre cas.

De plus, la répartition du temps de travail sur le projet variait selon les membres du groupe. Certains avaient d'autres priorités en dehors du projet, ce qui influençait le temps qu'ils pouvaient y consacrer. Par conséquent, vers la fin, il ne restait plus beaucoup de membres impliqués dans le projet, ce qui a eu un impact sur la progression et l'achèvement des tâches.

8 Conclusion

Le projet robotique a constitué une expérience riche en enseignements. Malgré les divers défis rencontrés, nous avons réussi à en surmonter la plupart. Bien que le projet ne soit pas entièrement achevé, en particulier la reconnaissance de la balise en 3D, nous avons néanmoins acquis des compétences et des connaissances précieuses. Cette aventure nous a également offert une exploration approfondie du domaine de la robotique et de la conception de logiciels.