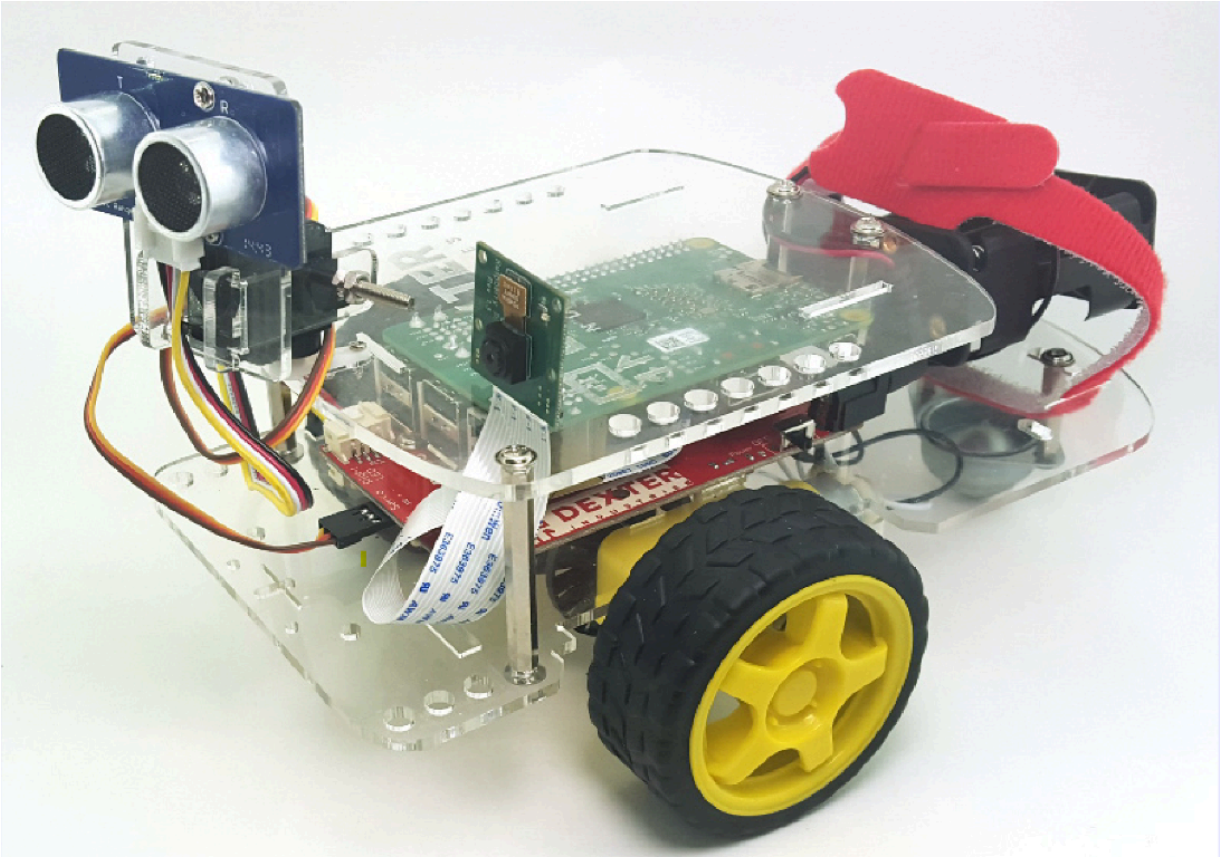

Rapport de projet : Contrôle Autonome d'un Robot Mobile

- *Bourenane lina*
- *Chen guanqi*
- *Efe SOZMEN*
- *Ari karakas*

1. Introduction

Ce projet a pour objectif de développer une plateforme de simulation et de contrôle pour un robot mobile autonome. Il s'agit d'un véritable banc d'essai pour développer, tester et valider des stratégies de déplacement dans un environnement sécurisé, avant de les appliquer sur un robot physique réel.

Trois modes d'exécution sont proposés :

- Simulation 2D via une interface graphique Tkinter,
- Simulation 3D avec caméra embarquée via Ursina,
- Contrôle réel du robot GoPiGo3 via une interface CLI.

Le cœur du projet repose sur une architecture modulaire, basée sur le modèle MVC et l'utilisation du design pattern Adapter

2. Contexte et motivation

Tester des comportements robotiques directement sur du matériel physique est coûteux, risqué et difficile à déboguer. Développer une couche de simulation permet :

- d'expérimenter sans danger,
- d'itérer rapidement,
- d'assurer une compatibilité directe avec le monde réel grâce à l'abstraction matérielle.

Caractéristiques

Techniques & Fonctionnelles

- **Langage principal** : Python 3
- **Paradigme** : Programmation orientée objet (OOP)
- **Architecture logicielle** : MVC + Design Pattern *Adapter*
- **Exécution asynchrone** : Boucles de simulation temps-réel (50 Hz)

- **Stratégies autonomes** :
    - Avancer, Tourner
    - Suivre une balise visuelle
    - Exécuter des formes géométriques (carré)
- **Détection visuelle** : caméra embarquée (virtuelle), analyse OpenCV

Simulation

- **Mode 2D** : Interface graphique avec Tkinter
- **Mode 3D** : Simulation immersive avec Ursina
- **Caméra virtuelle embarquée** : flux image en direct
- **Trace du robot** : visualisation du chemin parcouru

Robot réel compatible

- **Modèle physique** : GoPiGo3 / Robot2IN013
- **Contrôle moteur** : set_motor_dps (degrés/seconde)
- **Mesures capteurs** : encodeurs, distance, inertie (IMU), caméra
- **Adaptateur logiciel** : permet réutilisation des stratégies sans modifier le code

Interopérabilité

- Simulation et exécution réelle **partagent les mêmes stratégies**
- Basculer de la simulation au réel se fait **sans changer la logique**

Grandes étapes

I. Analyse des besoin

Définir les objectifs :
→ contrôler un robot réel via des stratégies testées en simulation

Identifier les contraintes (temps réel, interface, modularité

II. Conception de l'architecture

Définition d'une interface **commune (RobotAdapter)** pour simulation/réel

Planification des différents **modes d'exécution** : CLI, GUI, 3D

III. Implémentation de la simulation 2D

Création de la carte interactive (Tkinter)

Déplacement du robot simulé avec calculs physiques

Interface utilisateur pour placer obstacles / départ / balise

IV. Implémentation des stratégies de mouvement

Avancer, Tourner, Arreter

PolygonStrategy pour dessiner une forme

FollowBeaconByCommandsStrategy pour suivi visuel

V. Ajout de la simulation 3D (Ursina)

Modélisation 3D du robot et environnement

Caméra embarquée virtuelle

Flux d'images récupéré pour traitement

VI. Traitement d'image et détection visuelle

Intégration d'OpenCV

Analyse de l'image pour détecter une balise

Extraction de coordonnées pour guider le robot

VII. Connexion au robot réel

Création du ***RealRobotAdapter*** (interface matos)

Test des mêmes stratégies sur le **GoPiGo3**

Boucles de contrôle asynchrones via CLI

Robot Simulé

Le robot du projet est conçu pour naviguer de manière autonome sur une carte définie par un modèle externe. Ce rapport présente les aspects techniques et fonctionnels du robot, basé sur le modèle fourni (*RobotModel*), ainsi que sur son environnement représenté par le modèle de carte (*MapModel*).

Description technique du robot

Le robot possède les caractéristiques techniques suivantes :

- **Largeur de la base des roues :** 20.0 cm
- **Diamètre des roues :** 5.0 cm

Fonctionnalités Principales du robot

1. **Initialisation :**
    - Position initiale définie selon les coordonnées fournies par *MapModel*
    - Angle initial de direction réglé à 0 degré.
2. **Contrôle des moteurs :**
    - Vitesse réglable individuellement pour chaque roue (droite et gauche).
    - Mise à jour automatique des positions des moteurs selon la vitesse définie et le temps écoulé.
3. **Calcul de la distance parcourue :**
    - Distance calculée à partir des déplacements individuels des roues gauche et droite.
    - Calcul basé sur la moyenne des déplacements angulaires convertis en distance réelle parcourue.
4. **Gestion des virages :**
    - Capacité à décider automatiquement du sens du virage selon l'angle demandé.
    - Ajustement de la vitesse des roues pour effectuer des virages précis et contrôlés.
5. **Mise à jour de l'état du robot :**
    - Vérification et mise à jour régulière de la position actuelle et de l'angle directionnel.
    - Vérification des collisions et du dépassement des limites via les méthodes fournies par *MapModel*

Explication détaillée de la fonction calculer_distance_parcourue

La fonction **calculer_distance_parcourue** est responsable du calcul de la distance parcourue par le robot à partir de la rotation des roues. Ce calcul se base sur les concepts physiques de rotation et de déplacement linéaire liés aux roues d'un véhicule robotisé.

Physique impliquée

La conversion du déplacement angulaire des roues (rotation en degrés) vers un déplacement linéaire réel (en centimètres) est au cœur de cette méthode. La relation physique utilisée ici est la suivante :

distance parcourue=angle de rotation (radians)×rayon de la roue (cm)

- **Angle de rotation :** Mesuré en radians. Un tour complet (360 degrés) correspond à 2π2\pi2π radians.
- **Rayon de la roue :** Défini par le diamètre de la roue divisé par deux

Détail des calculs physiques dans le code

Calcul de la différence angulaire :

delta_left = new_positions["left"] - old_positions["left"]

delta_right = new_positions["right"] - old_positions["right"]

- La différence *(delta_left et delta_right)* représente l'angle total (en degrés) parcouru par chaque roue depuis le dernier calcul.

Conversion de degrés en radians et en distance linéaire :

left_distance = math.radians(delta_left) * self.WHEEL_RADIUS

right_distance = math.radians(delta_right) * self.WHEEL_RADIUS

- Cette conversion est essentielle, car les moteurs donnent leur rotation en degrés, tandis que la distance parcourue est une mesure linéaire.

Calcul de la moyenne des distances des roues :

self.distance += (left_distance + right_distance) / 2

En général, le robot avance en ligne droite lorsque les deux roues parcourent des distances similaires. En prenant la moyenne, on obtient une approximation fiable de la distance réelle parcourue par le centre du robot.

Explication détaillée de la fonction calcule_angle

La fonction **calcule_angle** calcule l'angle total que le robot a tourné en utilisant les différences d'angles parcourues par les roues. Cette fonction repose sur une relation physique directe entre la différence de déplacement des deux roues et l'angle de rotation du robot autour de son centre.

Physique impliquée

Le principe physique clé est la **cinématique différentielle** d'un robot à deux roues. La rotation du robot autour de son axe vertical dépend de la différence entre les distances parcourues par les deux roues.

La relation physique utilisée ici est la suivante :

angle de rotation (rad)=(Distance gauche - Distance droite) / distance entre les roues

Où :

- **Distance gauche/droite** = circonférence de la roue × (angle parcouru / 360)
- **Largeur base roue** = distance horizontale entre les deux roues.

Détail des calculs physiques dans le code

1. Calcul des différences angulaires des roues :

delta_left = positions["left"] - self.left_initial

delta_right = positions["right"] - self.right_initial

- Ces lignes permettent de déterminer de combien chaque roue a tourné depuis le début de la mesure, exprimée en degrés.

2. Conversion de l'angle en distance parcourue :

wheel_circumference = 2 * math.pi * (self.WHEEL_DIAMETER / 2)

- Ici, la circonférence de la roue est calculée :
  Circonference roue=2π×(Diamètre roue)/ 2=π×Diamètre roue

distance_difference = (delta_left - delta_right) * wheel_circumference / 360

- Conversion des angles (en degrés) en fractions de tour complet (en divisant par 360), puis multiplication par la circonférence pour obtenir la différence réelle de distance linéaire parcourue par chaque roue.

3. Calcul de l'angle de rotation du robot :

angle = distance_difference / self.WHEEL_BASE_WIDTH

- L'angle de rotation du robot est directement proportionnel à la différence de distances linéaires entre les deux roues. On divise par la largeur de la base pour obtenir l'angle en radians, car plus la base est large, plus le même écart de distances entraîne un petit angle de rotation.

Description de l'environnement

Le robot évolue sur une carte modélisée par la classe *MapModel.* Cette carte comprend :

- **Obstacles :** Représentés par des polygones définis par des points spécifiques, permettant la détection de collision.
- **Positions initiales et finales :** Points de départ et d'arrivée configurables.
- **Gestion dynamique des obstacles :** Ajout, suppression, et déplacement possible des obstacles pendant l'exécution.

Méthodes clés du modèle de carte

- **set_start_position(position)** : définit la position initiale du robot.
- **set_end_position(position)** : définit la position finale visée.
- **add_obstacle(obstacle_id, points, polygon_id, line_ids)** : permet d'ajouter un obstacle.
- **remove_obstacle(obstacle_id)** : permet de supprimer un obstacle.
- **move_obstacle(obstacle_id, new_points)** : repositionne un obstacle existant.
- **is_collision(x, y)** : vérifie si une position spécifiée est en collision avec un obstacle.
- **is_out_of_bounds(x, y)** : vérifie si une position spécifiée est hors des limites (implémentation à compléter).

Interface graphique

L'affichage graphique est assuré par les classes *MapView* et *RobotView* utilisant Tkinter :

- **MapView :**
  - Utilise un canevas graphique pour représenter visuellement la carte et ses éléments.
  - Gestion dynamique permettant d'afficher, d'ajouter ou de supprimer visuellement les obstacles.
  - Affichage distinct des positions initiale (jaune) et finale (verte).
  - Capacité de nettoyer entièrement la carte pour une réinitialisation visuelle complète.
- **RobotView :**
  - Visualisation en temps réel du robot, représenté par un triangle bleu indiquant clairement sa direction.
  - Traçage dynamique du trajet suivi par le robot à l'aide de lignes grises sur le canevas.
  - Affichage en temps réel des données clés, telles que la vitesse des roues gauche et droite et l'angle d'orientation du robot.
  - Mise à jour fluide de l'affichage via la gestion d'événements intégrée à Tkinter.

Simulation 3D et Caméra Virtuelle (Ursina)

But

Permettre une immersion visuelle et une validation réaliste des stratégies de déplacement du robot grâce à une simulation 3D, incluant une caméra embarquée virtuelle.

Fonctionnement:

- Moteur 3D utilisé : Ursina (Python)
- Modélisation : Le robot et l'environnement (sol, obstacles) sont modélisés en 3D.
- Caméra embarquée virtuelle :
    - Une caméra est attachée au robot dans la scène 3D.
    - Elle simule la vue qu'aurait une caméra réelle montée sur le robot.
    - Le flux d'images généré par cette caméra virtuelle peut être traité comme s'il provenait d'un vrai capteur (par exemple, pour la détection de balises avec OpenCV).

- Utilisation :
    - Permet de tester les algorithmes de vision et de navigation dans un environnement contrôlé.
    - Les stratégies de déplacement peuvent être observées en temps réel sous différents angles.

Exemple de code (simplifié)

```
from ursina import *


class Robot3D(Entity):
   def __init__(self):
      super().__init__(model='cube', color=color.blue, scale=(1,0.5,2))
      self.camera = Camera(parent=self, position=(0,1,0), rotation=(20,0,0))


app = Ursina()
robot = Robot3D()
app.run()
```

Stratégies de Déplacement

But

Définir des comportements autonomes pour le robot, réutilisables en simulation et sur le robot réel.

- Implémentation :
  - Chaque stratégie est une classe Python qui reçoit un “adapter” (interface robot).
  - Elle utilise des méthodes génériques (avancer, tourner, arreter, etc.) pour commander le robot.
  - Les stratégies peuvent être composées (ex: dessiner un carré = avancer + tourner x4).

- Exemples de stratégies :
  - Avancer/Tourner : Commande directe des moteurs pour avancer ou tourner d’un certain angle.
  - PolygonStrategy : Fait avancer le robot sur les côtés d’un polygone, puis tourner à chaque sommet.
  - FollowBeaconByCommandsStrategy : Utilise la vision pour suivre une balise détectée dans l’image.

Exemple de code (simplifié)

```
class PolygonStrategy:
    def __init__(self, adapter, sides=4, length=20):
        self.adapter = adapter
        self.sides = sides
        self.length = length

    def execute(self):
        for _ in range(self.sides):
            self.adapter.avancer(self.length)
            self.adapter.tourner(90)
```

- Cette stratégie dessine un carré en alternant avancer et tourner.

Adapter

But

Permettre à une même stratégie de fonctionner aussi bien en simulation qu'avec le robot réel, sans modifier le code de la stratégie

Fonctionnement

- Interface commune : L'Adapter définit les méthodes génériques (avancer, tourner, get_camera_image, etc.).
- Implémentations :
    - SimulationAdapter : Connecte les commandes aux objets simulés (Tkinter/Ursina).
    - RealRobotAdapter : Traduit les commandes en instructions pour le vrai robot (GoPiGo3).

- Avantage : Les stratégies n'ont pas à connaître les détails d'implémentation (simulation ou réel).

Exemple de code (simplifié)

```
class RobotAdapter:
    def avancer(self, distance): pass
    def tourner(self, angle): pass
    def get_camera_image(self): pass
class SimulationAdapter(RobotAdapter):
    def avancer(self, distance):
        # Commande le robot simulé
        pass
class RealRobotAdapter(RobotAdapter):
    def avancer(self, distance):
        # Commande le robot réel via GoPiGo3
        pass
```

- Les stratégies utilisent uniquement l'interface RobotAdapter.