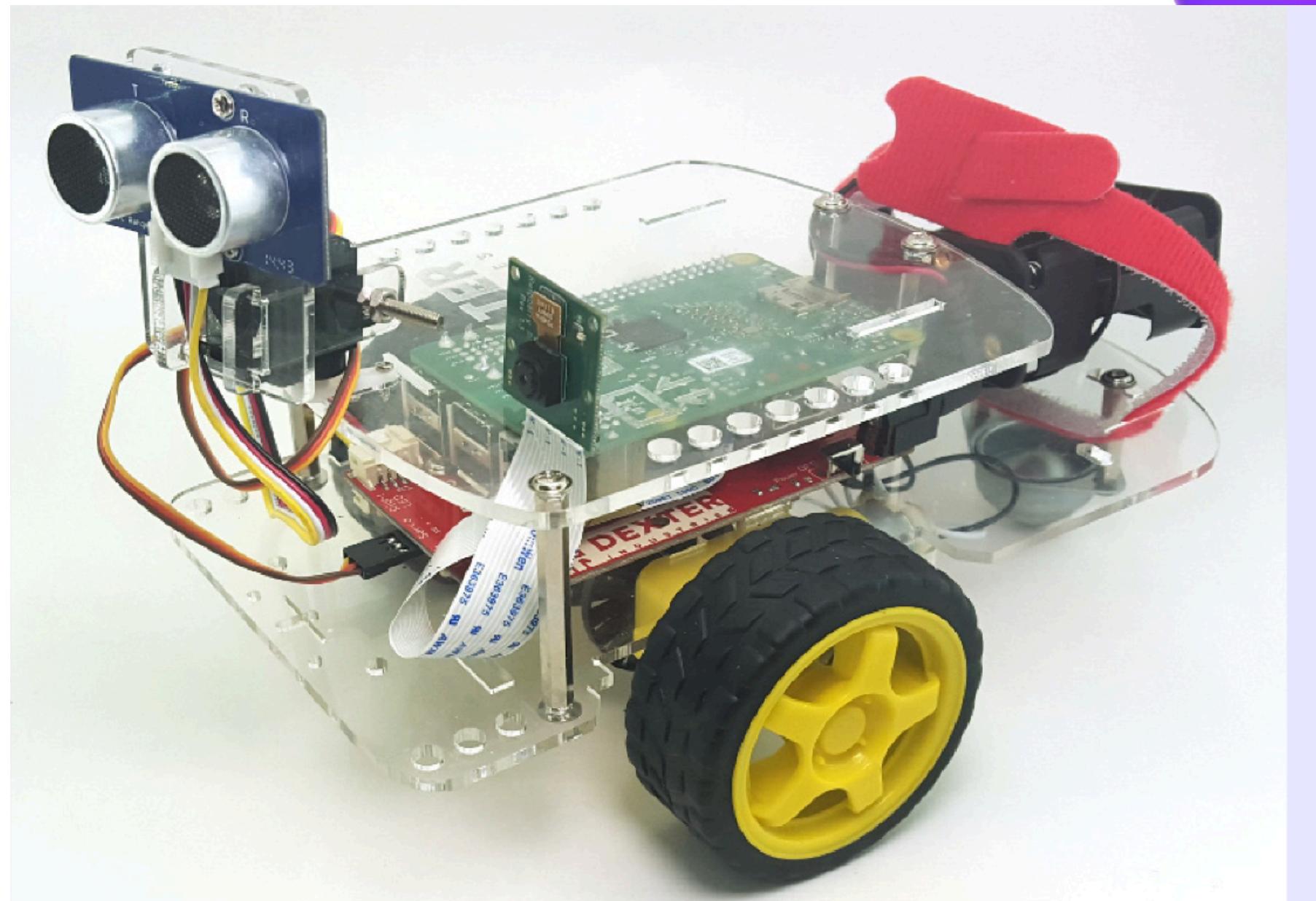


Projet : Contrôle autonome d'un robot mobile via simulation et stratégie asynchrone

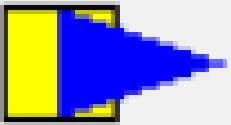
Réalisé par :

- Bourenane lina
- Chen guanqi
- E SOZMEN
- Ari karakas

Étudiants à la Sorbonne Université
Année universitaire : 2024–2025



Pourquoi ce projet ?



Les robots autonomes doivent prendre des décisions en temps réel, dans un environnement souvent imprévisible.

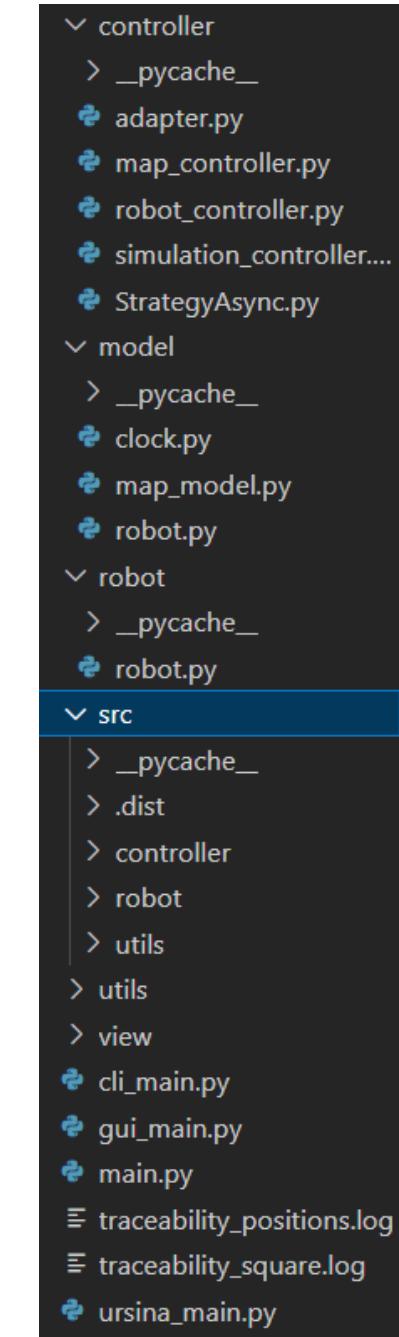
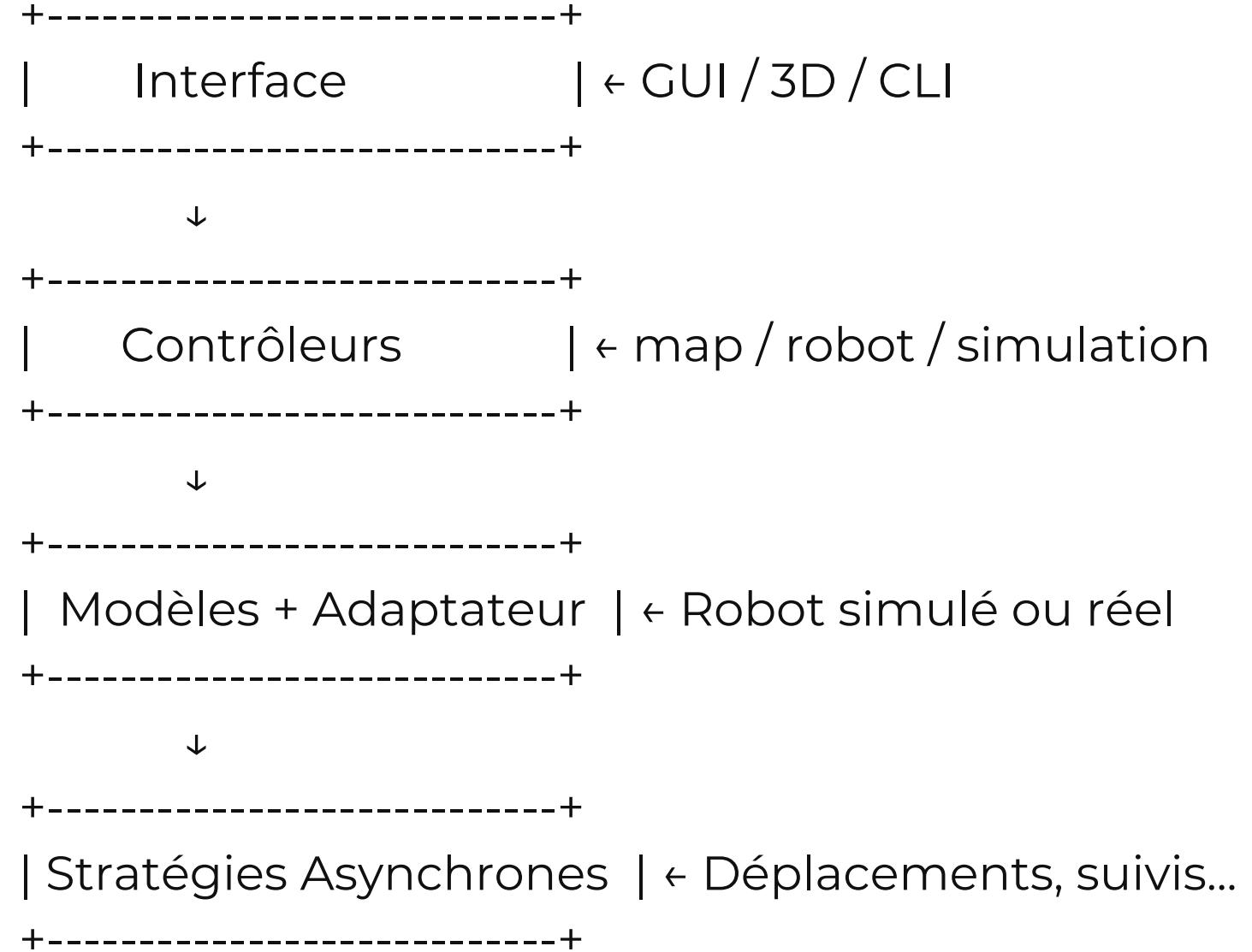
Tester directement sur un robot réel peut être :

- **✗** dangereux (chutes, collisions...)
- **✗** chronophage (recharges, erreurs matérielles)
- **✗** difficile à déboguer

Quelle solution ?

- Concevoir une plateforme de simulation interactive,
- Tester des stratégies de navigation (avancer, tourner),
- Puis les appliquer au robot physique sans modifier le code.

Architecture du système :



```
controller  
└ __pycache__  
    └ adapter.py  
    └ map_controller.py  
    └ robot_controller.py  
    └ simulation_controller....  
    └ StrategyAsync.py  
model  
└ __pycache__  
    └ clock.py  
    └ map_model.py  
    └ robot.py  
robot  
└ __pycache__  
    └ robot.py  
src  
└ __pycache__  
    └ .dist  
    └ controller  
    └ robot  
    └ utils  
    └ utils  
    └ view  
    └ cli_main.py  
    └ gui_main.py  
    └ main.py  
    └ traceability_positions.log  
    └ traceability_square.log  
    └ ursina_main.py
```

Calculs physiques du robot

Objectif :

Simuler précisément la position et l'orientation du robot à chaque instant, en fonction des vitesses moteurs.

Méthode utilisée :

- Utilisation du modèle différentiel à deux roues (comme une voiturette) :
 - Entrées : vitesses des moteurs gauche/droite (en degrés/sec)
 - Sortie : position (x, y) et angle θ du robot
- Calculs faits toutes les 20 ms (50 Hz)

⚙ Formules principales :

1. Conversion en vitesses linéaires (cm/s)

$$v = (\text{dps} / 360) \times (2\pi R)$$

(avec R = rayon de la roue)

2. Vitesse linéaire et angulaire globale

$$v_{\text{lin}} = (v_{\text{gauche}} + v_{\text{droite}}) / 2$$

$$\omega = (v_{\text{droite}} - v_{\text{gauche}}) / L$$

(L = distance entre les roues)

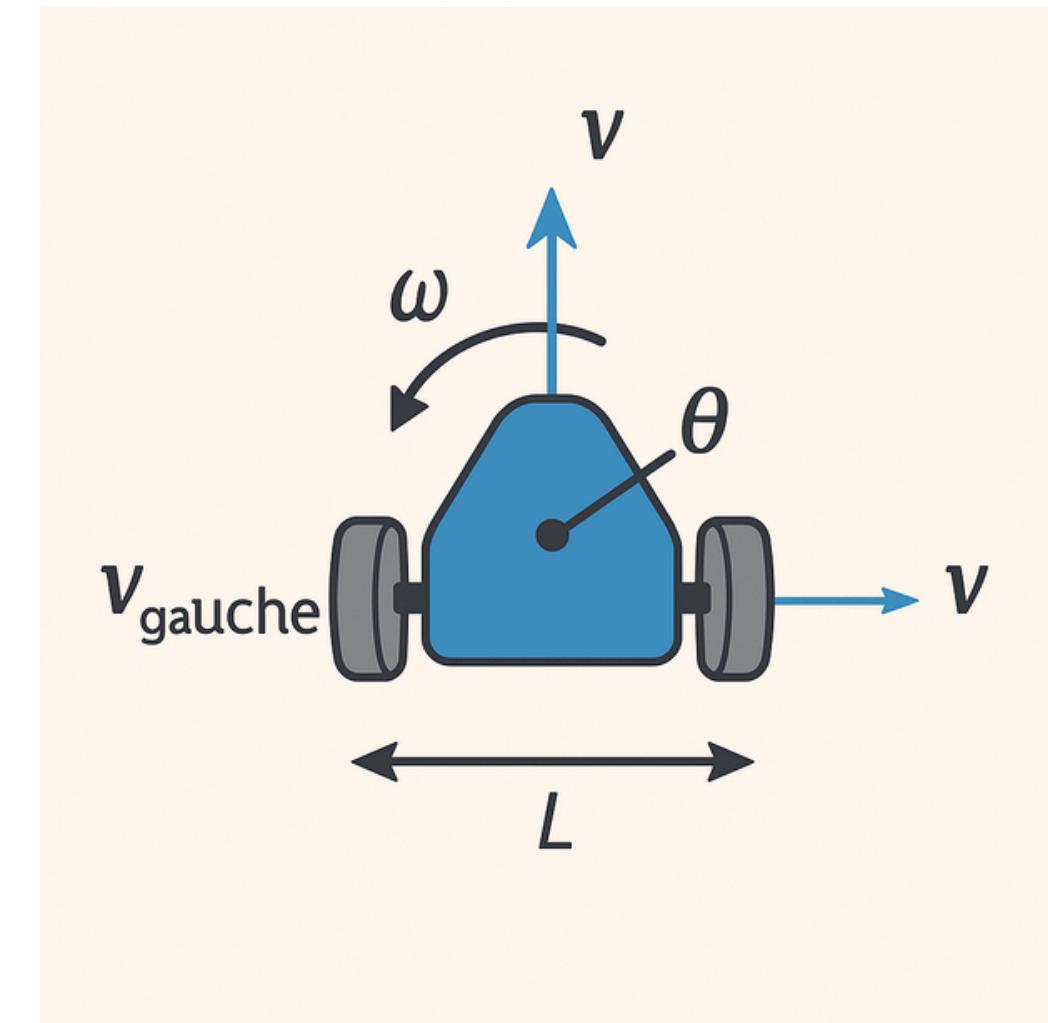
3. Mise à jour de la position

- Si les roues vont à la même vitesse → déplacement en ligne droite :

$$x += v_{\text{lin}} \times \cos(\theta) \times \Delta t$$

$$y += v_{\text{lin}} \times \sin(\theta) \times \Delta t$$

- Sinon → rotation autour d'un centre (mouvement en arc)



Mouvement en arc (rotation autour d'un centre)

Quand se produit-il ?

- Quand $v_{gauche} \neq v_{droite}$
- Le robot ne va pas tout droit : il suit une trajectoire circulaire
- Il tourne autour d'un Instantaneous Center of Rotation (ICR)

Paramètres du mouvement :

- v_{lin} = vitesse moyenne des roues

$$v_{lin} = (v_{droite} + v_{gauche}) / 2$$

- ω (vitesse angulaire)

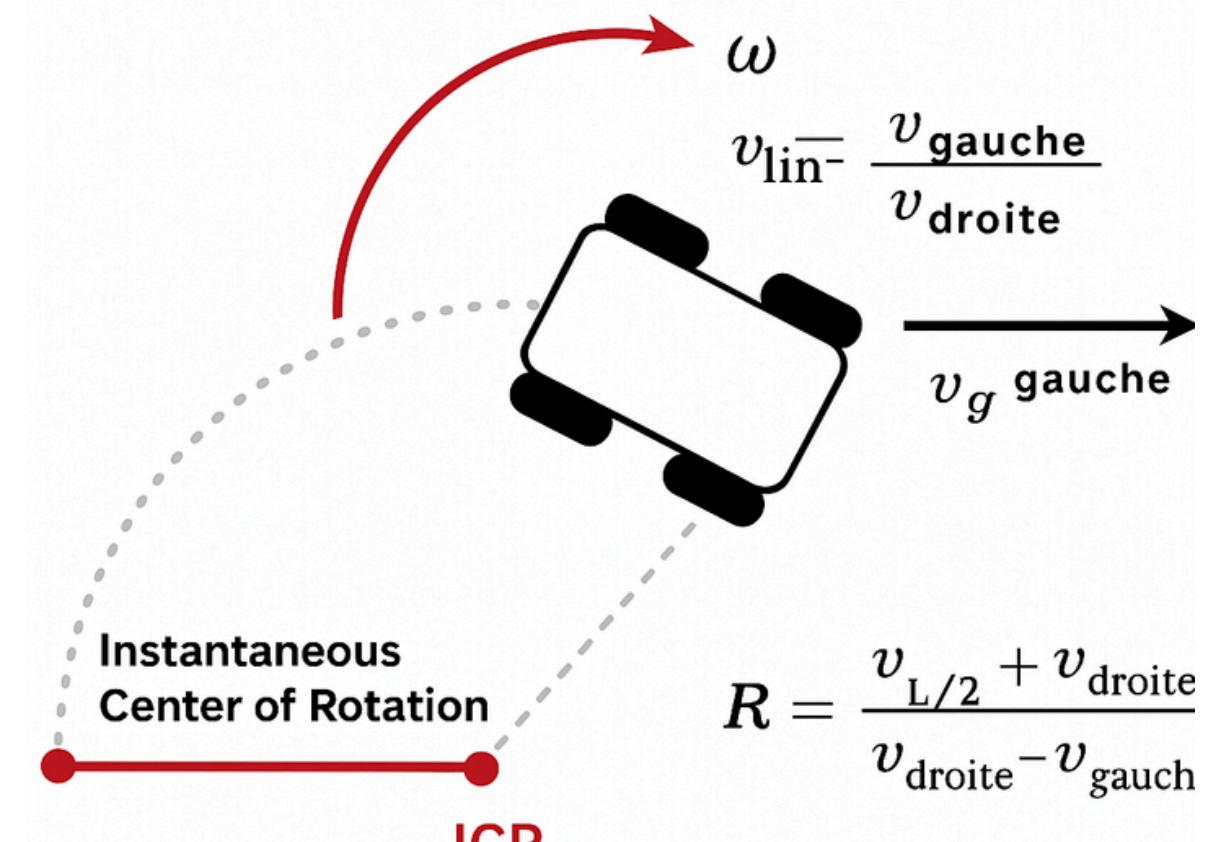
$$\omega = (v_{droite} - v_{gauche}) / L$$

- Rayon de courbure R :

$$R = L/2 \times (v_{gauche} + v_{droite}) / (v_{droite} - v_{gauche})$$

- Plus la différence entre les vitesses est grande, plus le robot tourne serré.
- Si $v_{gauche} = -v_{droite}$, il fait du **sur-place** (rotation pure).

Mouvement en arc



$$R = \frac{v_{L/2} + v_{droite}}{v_{droite} - v_{gauche}}$$

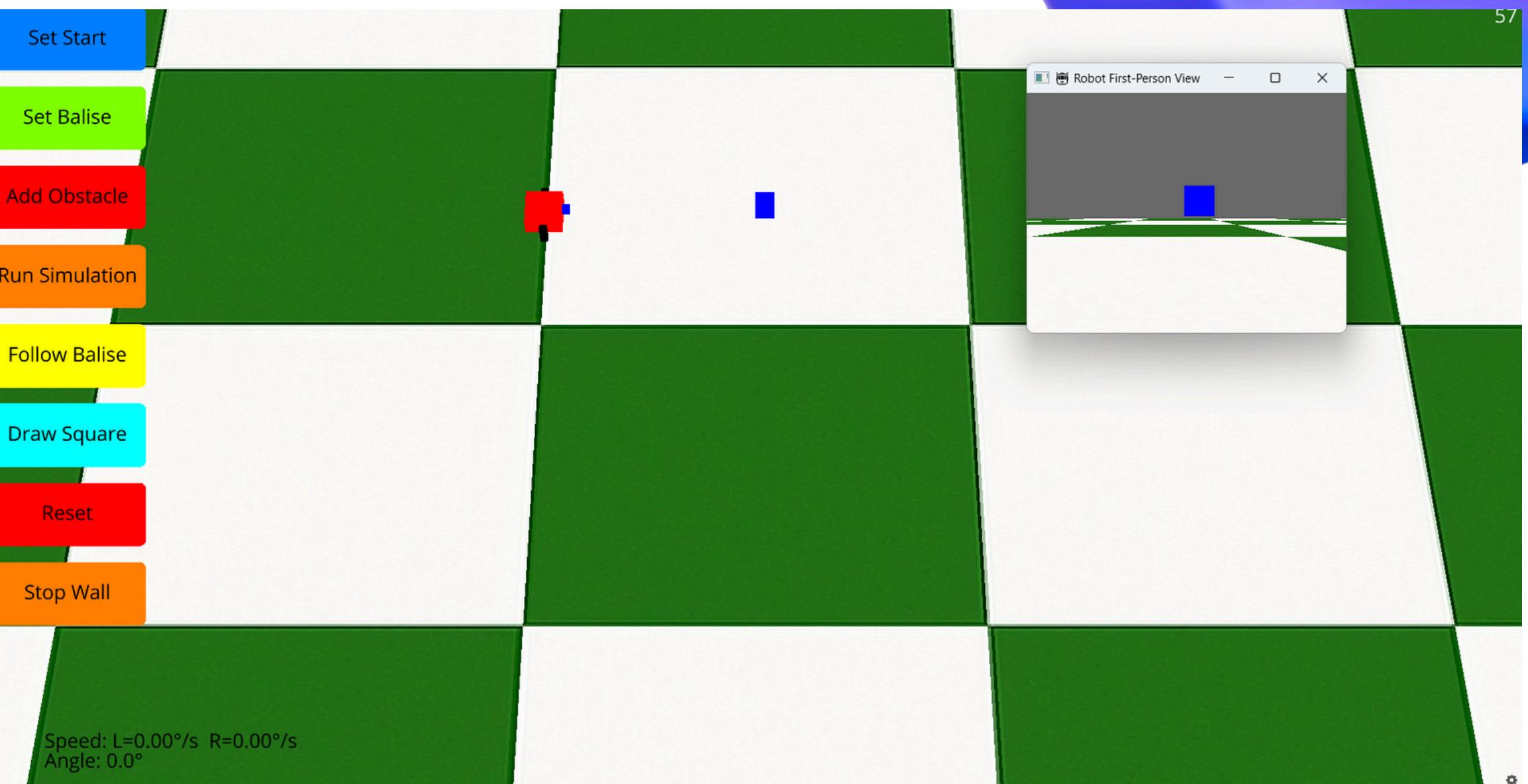
Caméra embarquée dans l'environnement 3D (Ursina)

🎥 Vue à la première personne du robot

- Caméra virtuelle fixée sur le robot (UrsinaView)
- Affiche une image temps réel depuis le robot
- Résolution : 400×300 pixels
- Utilisée pour simuler un capteur visuel embarqué

⚙️ Fonctionnement

- Utilisation de Ursina + Panda3D
- Création d'un FrameBuffer → image extraite à chaque frame
- Image convertie en array NumPy RGB
- Peut être affichée, enregistrée, analysée en direct



Analyse visuelle pour détecter une balise

Objectif

Extraire automatiquement la position et la distance relative d'un objet visuel (la balise) depuis la caméra embarquée du robot

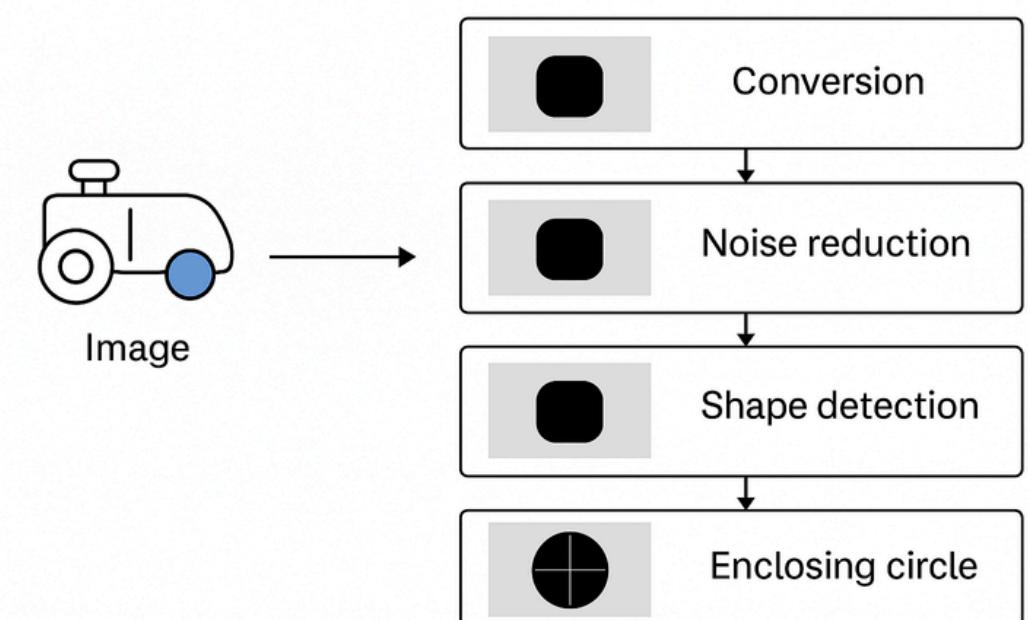
Étapes du traitement

1. Récupération d'une image depuis la caméra du robot
2. Conversion dans un espace de couleur adapté à l'analyse visuelle
3. Nettoyage de l'image pour supprimer les parasites
4. Identification de la forme dominante
5. Calcul d'un cercle englobant autour de cette forme
6. → permet de déterminer :
 - sa taille (estimée → distance)
 - sa position horizontale (centrage)

Stratégie utilisée : **FollowBeaconByCommandsStrategy**

- Le robot :
 - Pivote pour se centrer sur la balise
 - Avance quand elle est bien alignée
 - S'arrête quand il est suffisamment proche

Visual analysis to detect a beacon



L'adaptateur : Simulation \Leftrightarrow Réalité

🎯 Rôle de l'adaptateur

Permettre aux mêmes stratégies (avancer, tourner, suivre une balise) de fonctionner soit en simulation virtuelle, soit sur un robot physique réel (GoPiGo3)

Deux implémentations :

RobotModel :

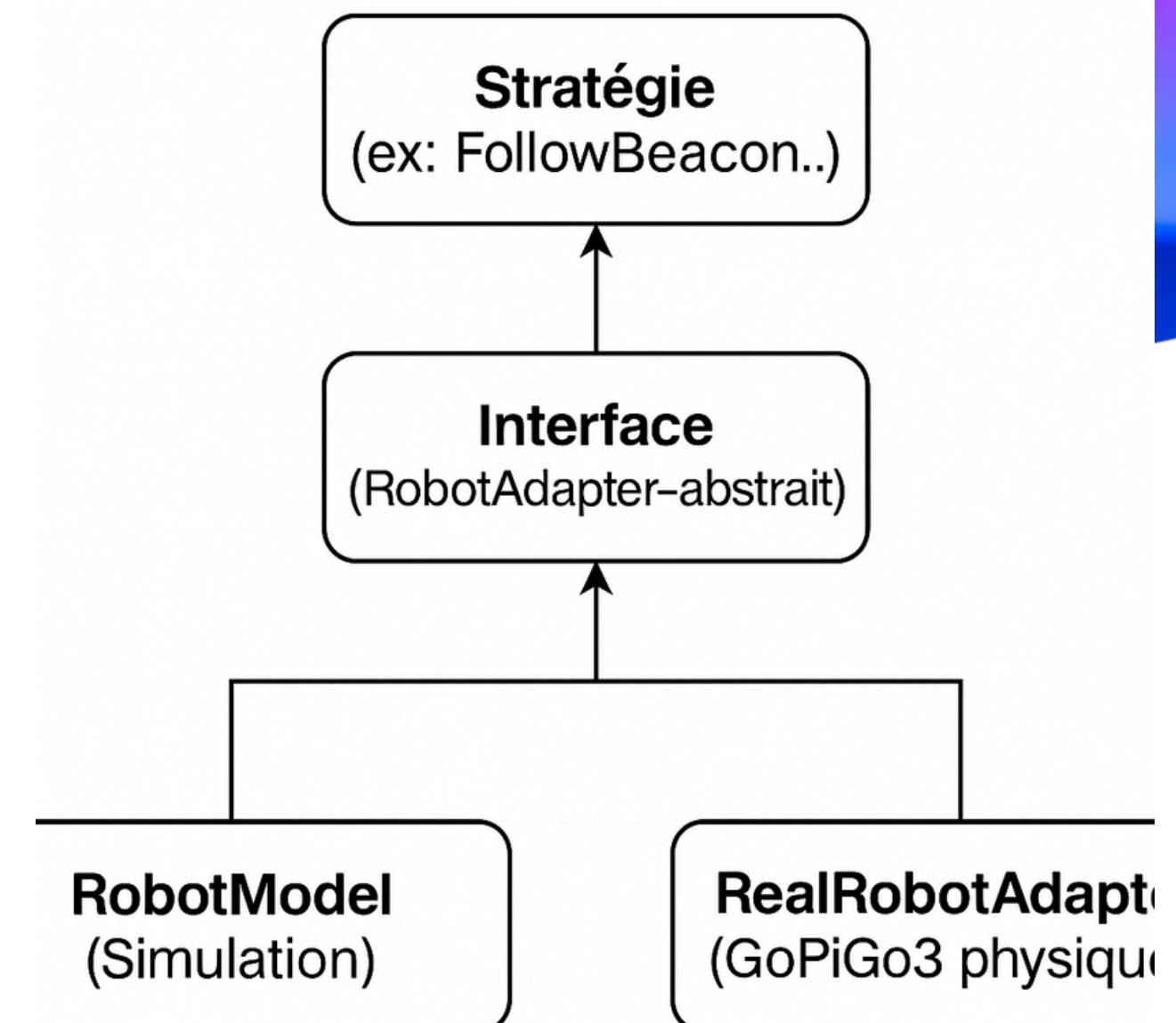
- → gère les déplacements dans la simulation 2D/3D
- → met à jour la position (x, y, angle) selon les vitesses moteurs

RealRobotAdapter :

- → traduit les ordres en commandes réelles moteurs
- → récupère les données des encodeurs, capteurs, etc

Architecture utilisée : Design Pattern “Adapter”

- Interface commune : `set_motor_speed()`, `calcule_angle()`, `calculer_distance_parcourue()`
- Le reste du code (stratégies, contrôleurs...) n'a pas besoin de savoir s'il parle à un vrai robot ou à une simulation



Conclusion

Bilan du projet

- Mise en place d'une plateforme complète de simulation et contrôle robotique
- Architecture modulaire et réutilisable

Contrôle possible en :

- Interface graphique (Tkinter)
- Environnement 3D avec vision (Ursina)
- Robot physique (GoPiGo3)