Programmation Orientée Objet (POO) à destination de la robotique

Travaux Pratique : Structurer une application robotique avec le modèle MVC

Enseignant : Adam GOUGUET
Mail : adam.gouguet@imt-nord-europe.fr
Année : 2025–2026

Table des matières

1 Objectifs du TP

Ce TP a pour objectif de vous donner les bases sur la programmation orienté objet en **Python**, en ayant une thématique d'application robotique. À l'issue de cette séance, vous devrez être capables de :

— structurer un programme en utilisant une architecture logicielle claire,
— implémenter le patron de conception **Modèle – Vue – Contrôleur**,
— séparer les responsabilités entre logique métier, affichage et interaction utilisateur,
— concevoir un environnement simulé contenant des entités interactives,
— gérer des collisions entre objets dans une simulation,
— utiliser la bibliothèque `pygame` pour créer une visualisation graphique temps réel.

2 Patron de conception MVC

Jusqu'à présent, nous avons construit un robot mobile capable de se déplacer en fonction de commandes, avec une séparation claire entre :

— le **robot**, qui stocke l'état (position, orientation),
— le **moteur**, qui applique la cinématique et la dynamique.

Cependant, un programme de simulation complet ne se limite pas au calcul du mouvement. Il doit également :

— afficher l'état du robot (visualisation),
— permettre à un utilisateur de contrôler le robot (clavier, terminal, manette, etc.).

Pour organiser proprement ces responsabilités, nous allons introduire un patron de conception fondamental en programmation orientée objet : le **modèle MVC**.

Un **design pattern** (ou patron de conception) est une **solution générale et réutilisable** à un problème de conception courant.

2.1 Principe du MVC

Le modèle MVC (Modèle - Vue - Contrôleur) repose sur une séparation claire des rôles :

— **Modèle (Model)** : contient les données et la logique métier,
— **Vue (View)** : responsable de l'affichage,
— **Contrôleur (Controller)** : gère les interactions utilisateur.

Cette séparation permet :

— de modifier l'affichage sans changer la logique du robot,
— de changer le mode de contrôle sans modifier le modèle,
— de rendre le code plus lisible, maintenable et réutilisable.

2.2 Application du MVC à notre robot mobile

Dans notre projet, les rôles sont naturellement répartis :

— **Modèle** :
  — `RobotMobile`

  - `Moteur` et ses sous-classes
- **Vue** :
  - `VueTerminal`
  - `VueSimulation`
- **Contrôleur** :
  - `ControleurClavier`
  - `ControleurTerminal`

On rajoutera la vue simulation et le controleur clavier avec le module Pygame plus tard dans le TP. Dans un premier temps on se concentre sur le terminal.

Chaque composant a une responsabilité unique :
- le **modèle** ne connaît ni l'affichage ni les entrées utilisateur,
- la **vue** ne modifie jamais l'état du robot,
- le **contrôleur** ne calcule pas le mouvement du robot.

2.3 Flux d'exécution d'une simulation

Le déroulement typique d'une boucle de simulation est le suivant :

1. le contrôleur lit les entrées utilisateur,
2. il génère une commande (vitesses, consignes),
3. la commande est transmise au modèle,
4. le modèle met à jour l'état du robot,
5. la vue affiche le nouvel état.

`Contrôleur` → `Modèle` → `Vue`

2.4 Intérêt du MVC dans ce TP

L'utilisation du modèle MVC permet notamment :
- d'utiliser **plusieurs vues** pour un même robot (terminal ou graphique),
- de changer de contrôleur sans modifier le reste du code,
- de tester le modèle indépendamment de l'interface graphique,
- de se rapprocher des architectures utilisées dans les applications réels.

Dans la suite du TP, vous pourrez contrôler le même robot soit depuis le terminal, soit via une interface graphique `pygame`, sans modifier le code du robot.

2.5 Vue Terminal

Pour définir nos différentes vue, créeons un fichier `vue.py` contenant notre `VueTerminal` :

```
class VueTerminal:
    def dessiner_robot(self, robot):
        # affiche les informations du robot
```

2.6 Contrôleur Terminal

Pour définir nos différents contrôleurs, créons un fichier `controleur.py` contenant une classe abstraite pour tous les contrôleur (`Controleur`) et le contrôleur avec le terminal (`ControleurTerminal`) :

```
from abc import ABC, abstractmethod

class Controleur(ABC):
    @abstractmethod
    def lire_commande(self):
        """Retourne une commande pour le robot"""
        pass

class ControleurTerminal(Controleur):
    def lire_commande(self):
        print("Commande differentiel : v omega (ex: 1.0 0.5)")

        # Utiliser input pour recuperer les entree clavier
        ...
        return {"v": v, "omega": omega}
```

2.7 Test

Pour tester, nous avons dans le `main.py` :

```
from robot.robot_mobile import RobotMobile
from robot.moteur import *
from robot.controleur import *
from robot.vue import *

robot = RobotMobile(moteur=MoteurDifferentiel())
controleur = ControleurTerminal()
vue = VueTerminalRobot()

dt = 1.0 # 1 second passe entre chaque mise a jour

running = True
while running:
    # La vue dessine
    ...
    # Le controleur recupere les commande de l'utilisateur
    ...
```

```
    # On envoie les commandes au robot
    ...
    # On met a jour le robot
    robot.mettre_a_jour(dt)
```

Vous devriez avoir un résultat ressemblant à cela :

```
Robot -> x=0.00, y=0.00, orientation=0.00
Commande differentiel : v omega (ex: 1.0 0.5)
> 1.0 0.5
Robot -> x=1.00, y=0.00, orientation=0.50
Commande differentiel : v omega (ex: 1.0 0.5)
> 10 0
Robot -> x=9.78, y=4.79, orientation=0.50
Commande differentiel : v omega (ex: 1.0 0.5)
>
```

3 Simulation avec Pygame

Pygame est une bibliothèque Python dédiée à la création d'applications graphiques interactives en 2D. Elle fournit des outils simples pour ouvrir une fenêtre, dessiner des formes (cercles, segments, rectangles), gérer le temps et afficher des animations.

Vous pouvez retrouver une liste de projet utilisant Pygame ici et la doc.

Nous allons utiliser Pygame comme un outil de visualisation permettant d'observer le comportement d'un robot mobile dans un environnement simulé.

3.1 Vue Pygame

Cette partie introduit une classe de visualisation basée sur Pygame. Elle a pour rôle de représenter graphiquement l'état du système (position et orientation du robot) à partir des variables calculées par le simulateur. La classe `VuePygame` encapsule l'ensemble des fonctionnalités graphiques. La fenêtre graphique est initialisée avec une largeur et une hauteur exprimées en pixels. Un facteur d'échelle permet de convertir les coordonnées exprimées en mètres vers des coordonnées écran.

```
class VuePygame:
    def __init__(self, largeur=800, hauteur=600, scale=50):
        pygame.init()
        self.screen = pygame.display.set_mode((largeur, hauteur))
```

```
        pygame.display.set_caption("Simulation Robot Mobile")
        self.largeur = largeur
        self.hauteur = hauteur
        self.scale = scale # metres -> pixels
        self.clock = pygame.time.Clock()

    def convertir_coordonnees(self, x, y):
        px = # largeur / 2 + (x * scale) (attention on veut un entier)
        py = # hauteur / 2 - (y * scale) (attention on veut un entier)
        return px, py

    def dessiner_robot(self, robot):
        x, y = # coordonne du robot dans l'affichage
        r = # son rayon
        # dessiner un cercle representant le robot (pygame.draw.circle)
        x_dir = x + int(r * math.cos(robot.orientation))
        y_dir = y - int(r * math.sin(robot.orientation))
        # dessiner un trait representant l'orientation du robot (pygame.draw.line)

    def tick(self, fps=60):
        self.clock.tick(fps)
```

Le robot est représenté par un cercle correspondant à son rayon et un segment indiquant son orientation. L'affichage est rafraîchi à chaque itération de la boucle de simulation, ce qui permet de visualiser l'évolution du système en temps réel.

Ajouter l'appel de la méthode `tick` dans la boucle du main.

FIGURE 1 – Simulation pygame d'une navigation de robot mobile

3.2 Contrôleur Pygame

Le contrôleur Pygame permet de piloter le robot à l'aide du clavier, via la fenêtre graphique. Il constitue une alternative au contrôleur basé sur l'entrée texte dans le terminal. À chaque itération de la boucle de simulation, l'état du clavier est interrogé afin de produire une commande de déplacement. Les touches directionnelles sont associées à :

— une vitesse linéaire pour l'avance et le recul,
— une vitesse angulaire pour la rotation du robot.

Chercher dans la doc de Pygame comment récupérer les touches du clavier en cours d'exécution et créer la classe `ControleurClavierPygame`.

4 Ajout d'un environnement et des obstacles

Jusqu'à présent, notre robot évoluait dans un espace abstrait sans contrainte. Pour rendre la simulation plus réaliste et plus modulaire, nous allons introduire une nouvelle abstraction : **l'environnement**.

4.1 Principe

L'environnement représente le monde simulé dans lequel évolue le robot. Il devient le **gestionnaire central de la simulation** :

— il contient le robot,
— il contient les obstacles,
— il met à jour la simulation,
— il vérifie les collisions.

Cette approche permet d'éviter que le robot ait connaissance directe des obstacles ou des limites du monde. Le robot reste donc un modèle purement cinématique, tandis que l'environnement gère les interactions physiques.

On respecte ainsi le principe de responsabilité unique : le robot calcule son mouvement, l'environnement décide s'il est autorisé à bouger.

4.2 Classe `Environnement`

On introduit une classe `Environnement` qui stocke :

— la taille du monde simulé,
— une référence vers le robot,
— une liste d'obstacles.

Elle fournit également des méthodes pour :

— ajouter un robot,
— ajouter des obstacles,
— mettre à jour la simulation,
— tester les collisions.

Le principe de mise à jour est le suivant :

1. on sauvegarde l'état courant du robot,
2. on laisse le robot calculer son mouvement,
3. on teste s'il entre en collision,
4. si oui, on annule le déplacement.

```
Robot calcule -> Environnement valide -> Position acceptée ou refusée
```

Dans les simulateurs robotiques réels, le robot n'est jamais responsable des collisions : c'est toujours le moteur physique ou l'environnement qui décide de la validité du mouvement.

4.3 Gestion des collisions

L'environnement ne connaît pas la forme précise des obstacles. Il délègue donc ce calcul à chaque obstacle individuellement. Chaque obstacle doit simplement fournir une méthode `collision(position, rayon)` qui retourne `True` si le robot entre en contact avec lui. Cette approche permet d'ajouter facilement de nouveaux types d'obstacles (rectangles, polygones, murs, etc.) sans modifier le code de l'environnement.

On applique ici un principe fondamental de la POO : **le polymorphisme**. L'environnement manipule des obstacles sans connaître leur type concret.

4.4 Classe abstraite `Obstacle`

Pour garantir que tous les obstacles possèdent les mêmes fonctionnalités, on définit une classe abstraite :

— elle impose une méthode de collision,
— elle impose une méthode d'affichage,
— elle définit une interface commune.

Toutes les formes d'obstacles hériteront de cette classe.

4.5 Obstacle circulaire

Nous implémentons un premier type d'obstacle simple : le cercle.

La collision avec un robot circulaire est déterminée en calculant la distance entre les centres :

$$d \leq r_{\text{obstacle}} + r_{\text{robot}}$$

Si cette condition est vraie, alors il y a collision.

Ce type d'obstacle présente plusieurs avantages pédagogiques :

— calcul simple,
— rapide à simuler,
— facile à afficher,
— extensible vers d'autres formes.

4.6 Intégration dans l'architecture MVC

Avec l'introduction de l'environnement, l'architecture devient :

- **Modèle**
  - Robot
  - Moteur
  - Environnement
  - Obstacles
- **Vue**
  - affiche l'environnement
  - dessine robot + obstacles
- **Contrôleur**
  - génère les commandes du robot

La vue n'affiche plus directement le robot seul : elle affiche maintenant **l'état global du monde simulé**.

4.7 Avantages de cette conception

L'introduction d'un environnement apporte plusieurs bénéfices importants :

- simulation plus réaliste,
- architecture plus modulaire,
- code plus extensible,
- ajout facile de nouvelles interactions.

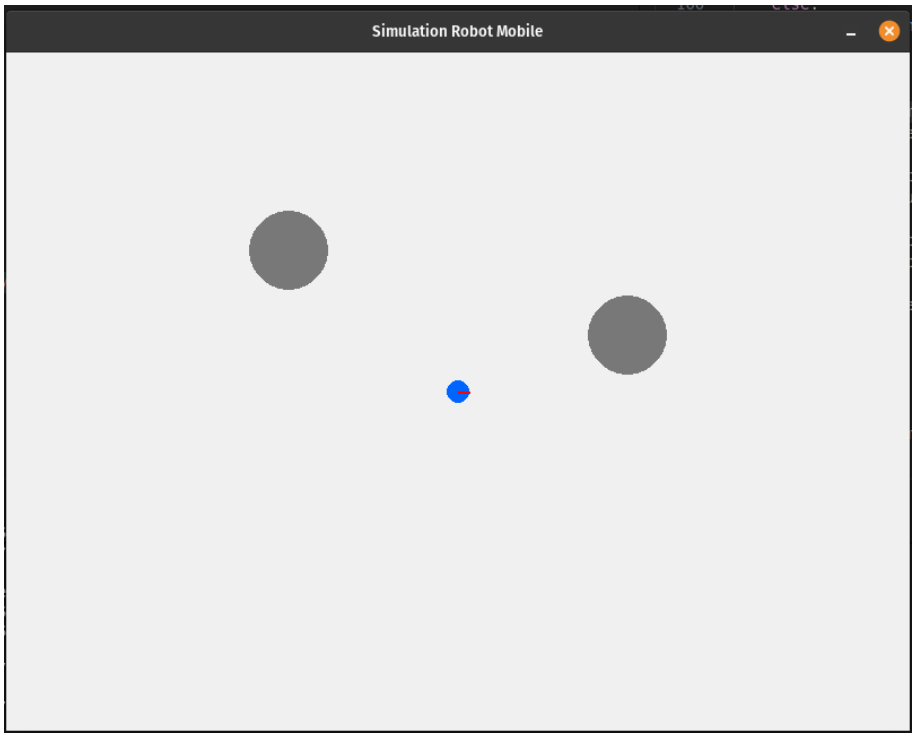


FIGURE 2 – Simulation pygame d'une navigation de robot mobile avec des obstacles

5 Github (si ce n'est pas fait)

GitHub est une plateforme de gestion de versions basée sur l'outil `git`. Elle permet de sauvegarder l'évolution d'un projet, de collaborer à plusieurs sur le même code et de partager facilement un travail.

Dans ce TP, GitHub sera utilisé comme un **outil de travail collaboratif** :

— partager le code du projet entre les membres du groupe,

— travailler à plusieurs sur les mêmes fichiers,

— conserver un historique clair des modifications.

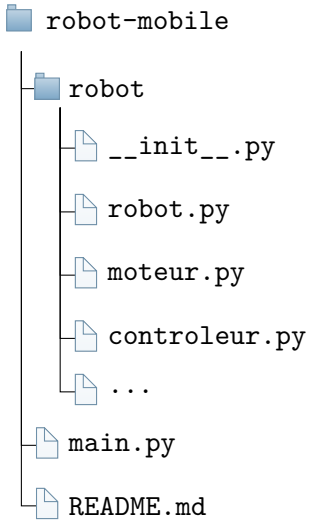
Chaque groupe devra disposer d'un dépôt GitHub unique contenant l'ensemble du projet de simulation du robot mobile.

5.1 Création du dépot

Un membre du groupe crée un dépôt sur GitHub, les autres membres du groupe sont ensuite ajoutés comme collaborateurs.

5.2 Organisation du projet

Le dépôt doit respecter l'organisation suivante :



Cette organisation vise à séparer clairement :

— la logique du robot,

— les contrôleurs,

— la visualisation via Pygame.

5.3 Travail collaboratif

Chaque membre du groupe doit contribuer activement au projet. Les bonnes pratiques suivantes sont attendues :

— effectuer des `commits` réguliers avec des messages explicites,

— éviter de travailler directement sur la branche `main`,

— utiliser une branche par fonctionnalité (ex : `controleur`, `simulation`).

Avant de fusionner une branche, le code doit fonctionner et ne pas casser la simulation existante.

5.4 Documentation

Le fichier `README.md` doit contenir au minimum :
— les noms des membres du groupe,
— une description du projet,
— les instructions pour lancer la simulation,
— les dépendances nécessaires.

Remarque L'utilisation de GitHub (structure du dépôt, documentation, répartition du travail) fait partie de l'évaluation du projet.

Modifier ou créer votre diagramme UML en prenant en considération le design pattern MVC et Pygame.