Projet Robotique

TINRA est l'équipe composé de :
Theret Ingrid, Langlais Rafaël, Bouyoucef Assirem, Koskas Ilan, Kerdouche Nabil

Récapitulatif du projet

Le but final du projet est de concevoir un programme contrôlant un robot roulant et de lui faire faire une série d'objectifs simples tels que : suivre des chemins prédéfinis, accélérer le plus possible puis freiner pour éviter une collision avec un mur ou encore suivre une balise de couleur. Néanmoins pour anticiper et prévoir des problèmes, il est demandé de concevoir un simulateur qui permet de tester le contrôleur du robot ainsi que les différents codes des objectifs dans un environnement virtuel avant de les tester dans le monde réel.
Le simulateur est séparé en trois parties : le contrôleur qui est le cerveau du robot, les objets qui sont la simulation de différents objets dans l'environnement virtuel (le robot, l'arène, les obstacles...) et l'affichage. Le contrôleur est un code pouvant être exécuté dans le simulateur comme dans le robot physique , et ce sans modification pour permettre une comparaison et des tests valides.

Premières étapes

La première étape du projet a été de créer les différents objets pour notre environnement virtuel :

- Arene : le code nous permet de construire l'arène en choisissant ses dimensions et en y ajoutant des obstacles et un (ou plusieurs) robot.
- Robot : pour ajouter un robot dans l'arène, on a besoin de sa position initiale x et$y$ et de sa vitesse initiale. La vitesse est un vecteur, donc en plus de sa valeur v on a besoin de sa direction θ (la direction est obtenu grâce à un angle en radian, la conversion en degrés est faite automatiquement dans le code).
- Obstacle : la création d'un obstacle nécessite sa dimension, longueur et largeur, ainsi que le coordonnées de son centre.

Le robot va bouger, on a ainsi ajouté à la classe Robot des fonctions permettant de contrôler ses mouvements. En fonction du pas de temps dt :

- celui-ci va pouvoir avancer avec la fonction *move(dt)* qui va recalculer la position du robot

$$x(t + \mathrm{d}t) = x(t) + v(t)\ \cos\theta\ \mathrm{d}t$$
$$y(t + \mathrm{d}t) = y(t) + v(t)\ \sin\theta\ \mathrm{d}t$$

— il va aussi accélérer d'un facteur a avec *accelerate(a,dt)* qui modifiera la norme du vecteur vitesse

$$v(t + \mathrm{d}t) = v(t) + a\ \mathrm{d}t$$

— le faire tourner d'un angle α en degrés avec *turn(α)* qui modifiera la direction de ce vecteur qui est en radian (modulo 2π)

$$\theta(t + \mathrm{d}t) = \theta(t) + \alpha\ \frac{\pi}{180}$$

On a ensuite travaillé sur l'affichage : on a séparé l'affichage texte et l'affichage graphique. L'affichage graphique est fait en openGL (la suite du projet aurait été en 3D, ce qui est fait avec celui-ci) et on a tenté plusieurs fois de séparer l'affichage graphique et le *main* pour faire des tests tout en contrôlant cet affichage, c'est-à-dire l'activer et le désactiver sans interrompre le programme. Pour le moment, on n'a pas réussi car cela prenait trop de temps et on a préféré tourner nos efforts vers les stratégies à mettre en place pour notre robot.
Nous avons aussi réalisé un affichage texte qui lui est activable et désactivable à tout moment lors de simulation.

Deux stratégies ont été implémentés au début : pour la première, on demande au robot d'avancer puis de s'arrêter devant un obstacle et pour la deuxième, au lieu de s'arrêter, il tournera d'un angle de 90 degrés. pour implémenter ces stratégies, il a fallu créer plusieurs fonctions afin de calculer la distance entre le robot et l'obstacle le plus proche dans sa direction

— *distancePointRobot* : fonction permettant de calculer la distance entre le robot et un point : on connaît la position du robot mais ce n'est pas le cas pour le point de collision avec l'obstacle, on doit donc le calculer.
— *findIntersection* : une fonction pour calculer le point d'intersection entre deux droites : un obstacle est rectangulaire et donc constitué de 4 segments. La fonction permet de calculer le point d'intersection d'un de ces segments qui sera ici une droite et le vecteur vitesse du robot qui sera considéré comme une demi-droite. Comme la bordure de l'obstacle est une droite ici, on peut avoir des points d'intersection qui ne feront pas parti de l'obstacle.
— *pointInObstacle* : une fonction pour déterminer si le point calculé appartient à un obstacle
— *pointsCollision* : fonction déterminant si, pour un obstacle, il y a une collision (l'obstacle peut être derrière le robot et dans ce cas, il n'y aura aucun point) et donnera les coordonnées du point de collision. Cette fonction utilisera *findIntersection* et *pointInObstacle* .
— *distanceRobotObs* : Calcule les points de collisions avec tous les obstacles présents dans l'arène et nous donne le point le plus proche et dans la direction du robot. Fera une boucle pour calculer ces points en utilisant *pointsCollision*.

Équation pour trouver le point d'intersection entre deux droites en connaissant deux points appartenant à chaque droite (notés 11 et 12 pour la première droite, 21 et 22 pour la seconde). On calcule d'abord ces facteurs :

$$\alpha_1 = x_{11}\ y_{12} - y_{11}\ x_{12}$$
$$\alpha_2 = x_{21}\ y_{22} - y_{21}\ x_{22}$$
$$\beta = (x_{11} - x_{12})(y_{21} - y_{22}) - (y_{11} - y_{12})(x_{21} - x_{22})$$

et on obtient les coordonnées du point de collision :

$$x\ \beta = \alpha_1\ (x_{21} - x_{22}) - \alpha_2\ (x_{11} - x_{12})$$
$$y\ \beta = \alpha_1\ (y_{21} - y_{22}) - \alpha_2\ (y_{11} - y_{12})$$

Réalisé récemment : Contrôleur

Du fait des difficultés rencontrées dernièrement, les objectifs ont dû être réduit. Nous avons néanmoins réussi à réaliser l'objectif premier qui était de faire suivre au robot un chemin prédéfini. Le robot est en effet capable de se déplacer selon un carré tout en prenant en compte son environnement.

Afin de réaliser ce contrôleur, plusieurs contraintes doivent être respectées. En commençant par le MVC (Model View Controler) exigeant la séparation entre le modèle (robot), l'affichage (arène virtuelle) et le contrôleur (IA). Cela implique qu'on pourra contrôler plusieurs modèles de robots, réels et virtuels. Ainsi qu'être capable d'activer ou désactiver l'affichage durant les tests. Une autre contrainte est que tous les contrôles (fonctions) doivent être asynchrones, cela veut dire que le robot pourra effectuer plusieurs tâches au même temps sans être bloqué, attendre que l'une des fonctions se termine pour pouvoir en effectuer d'autres. Cela complique les choses du point de vue programmation, nous obligeant à renoncer à toute boucle d'instructions impératives.

Nous avons donc mis en place des stratégies, en commençant par des petites permettant d'aller en avant puis de s'arrêter ou de tourner d'un certain angle dans une direction donnée. Ensuite en terminant par la grande stratégie englobant les autres et permettant de faire un carré avec le robot.

Principe des stratégies : une stratégie est codée dans une classe, donc sera utilisée avec une instance. Le constructeur prend le robot (le modèle) ainsi que d'autres paramètres. Les fonctions clés sont *start()*, *step()* et *stop()* qui déterminent dans l'ordre : le début, la mise a jour (en cours d'exécution) et la fin de la stratégie. Voici les stratégies mises en place dans l'ordre :

1. *go_ ahead_ strategy :* permet d'avancer d'une distance donnée.

- *start()* : sauvegarde la position de départ, et met la vitesse à zéro (facultatif).
- *step()* : accélérer, bouger le robot (virtuel), met à jour la distance parcourue.
- *stop()* : renvoie vrai si la distance demandée a été parcourue ou s'il y a un obstacle devant.

2. *turn_right_strategy :* permet de tourner à droite d'un angle donné.
   - *start()* : sauvegarde la direction de départ et la direction visée.
   - *step()* : faire tourner le robot à droite : bouger le robot virtuel, et faire accélérer la roue gauche concernant le réel.
   - *stop()* : renvoi vrai si la direction demandée est atteinte.
3. *turn_left_strategy :* permet de tourner à gauche d'un angle donné (même principe que la précédente dans la direction inverse (angle positif).
4. *square_right_strategy :* permet de faire un parcours en forme de carré en utilisant les stratégies précédentes à tour de rôle. Quatre instances de chaque sont créées puis mises dans une liste selon leurs d'exécution :
   go_ahead → turn_right → go_ahead → turn_right ...
   - *start()* : extrait la première stratégie de la liste (pour commencer par avancer), et initialise la vitesse à zéro (facultatif).
   - *step()* : si la stratégie courante est finie on passe à la suivante dans la liste. Sinon on fait juste step() de la courante.
   - *stop()* : renvoi vrai si la liste de stratégies est vide, donc le carré est fait.

Conclusion

La réalisation de ce projet d'introduction à la robotique était une riche expérience de travail en groupe. Le thème de la robotique et de l'intelligence artificielle n'est qu'un plus. Le plus important dans un projet est la gestion, nous avons donc appris la méthode agile SCRUM, très pratique pour gérer les idées, les tâches à faire et surtout le temps. L'utilisation de GitHub qui est un outil de gestion de versions de code a été ainsi primordiale au sein du groupe. Nous avons aussi découvert et mis en pratique une architecture qui est le MVC design-pattern. En addition à toutes les compétences techniques acquises tel que la programmation objet avec python, la simulation et la stratégie asynchrone.