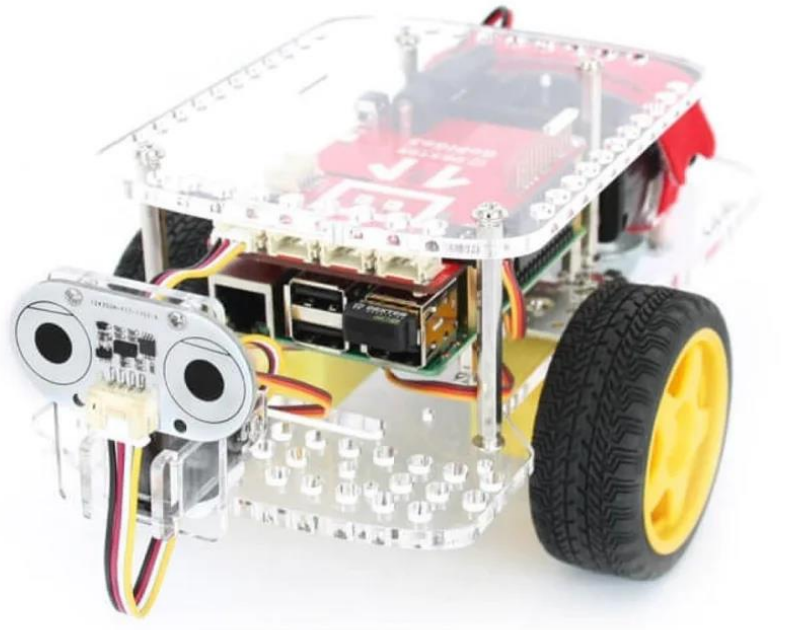
Projet de Robotique

Groupe iRobot
Grégoire Toudon et David Kitoko



2

LU2IN013 : Nicolas Baskiotis et Olivier Sigaud

SORBONNE UNIVERSITÉ

TABLES DES MATIÈRES

A. Introduction

A. Présentation

Nous sommes le groupe iRobot composé de Grégoire Toudon et David Kitoko. Pendant ce semestre, nous avions un projet à réaliser dont le but était d'appréhender un nouvel environnement, Python, mais aussi d'apprendre à gérer un projet en groupe. Pour cela, nous devions créer un environnement de simulation pour le robot Gopigo de Dexter Industries en implémentant les fonctions de ce robot. Et une fois la simulation fonctionnelle, nous pourrons passer sur le robot réel et le faire fonctionner de manière autonome, en lui donnant des ordres simples et en lui faisant éviter les collisions en détectant les obstacles.

Afin de réaliser tout cela, nous avons procédé en deux étapes :

- Créer la simulation du robot et représenter ses déplacements à travers une interface graphique.

- Adapter notre simulation au robot réel

Avant toute chose, nous avons dû apprendre à utiliser deux plateformes, Trello et GitHub. Trello est une plateforme qui nous a permis d'organiser la répartition des tâches entre les membres du groupe, et nous a également permis de voir l'évolution de chaque tâche avant la fin de la semaine.
Tandis que GitHub est une plateforme très pratique et complète pour organiser notre code et son arborescence, mais aussi pour garder un historique de notre projet tout au long du semestre.
Il nous a également fallu créer un groupe WhatsApp pour pouvoir discuter tout au long de la semaine de l'avancement du projet et pour pouvoir nous envoyer certains fichiers ou vidéos qui nous ont aidés.
Nous avons également convenu d'une réunion par visionnage sur Zoom chaque mardi. Afin de vérifier que notre code soit fonctionnel et préparer l'entretien avec les clients.

A. L'arborescence

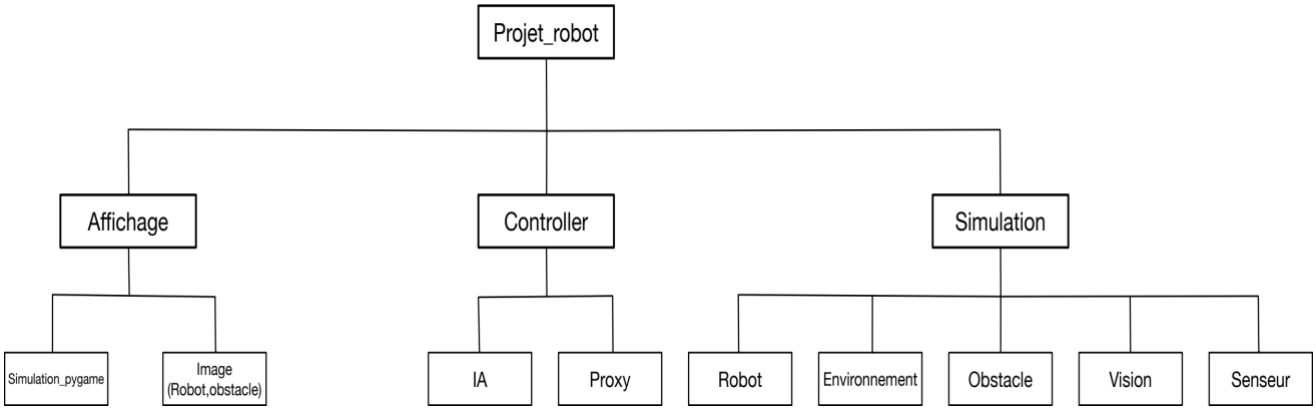
Pour organiser notre projet, nous avons divisé le code en trois parties, la partie Simulation, la partie Affichage et la partie Controller.

- La partie Simulation contient les classes Robot, Environnement, obstacle, Senseur et Vision.

- La partie Controller contient la classe IA ainsi que la classe Proxy.

•La partie Affichage est composé des images (robot et obstacles) ainsi que de la classe Simulation_pygame permettant la création de l'interface graphique.

Voici une représentation de l'arborescence de notre projet :



II) La Simulation

A. Le robot

Etant donné que nous partions de zéro, il a fallu créer notre robot virtuellement.

Pour cela, nous avons imaginé une classe Robot qui puisse se rapprocher le plus possible du robot réel.

Notre robot virtuel évolue dans un environnement, ainsi, il doit posséder comme attributs :
- Un angle et une position (x et y)
- Une vitesse (pour la roue gauche et la roue droite)

Il a également fallu définir les constantes propres au robot : diamètre et périmètre d'une roue, distance séparant la roue gauche de la roue droite, et le périmètre du cercle de rotation.

Notre classe Robot doit également contenir :
- Un constructeur (qui initialise les attributs)
- Les méthodes de l'API (set_motor_dps, offset_motor_encoder, ...)
- Une méthode « move » qui permet au robot de se déplacer en fonction d'un temps donné.
- Plusieurs accesseurs qui permettent d'obtenir certaines informations du robot.

2) Les obstacles

Afin de pouvoir simuler la présence d'obstacles dans notre simulation, nous avons créé une classe Obstacle qui permettra la création de ces obstacles. Pour pouvoir représenter plusieurs formes d'obstacle, on a défini 6 attributs pour cette classe :
- Une position (x et y)
- Une taille (largeur et longueur)
- Une vitesse qui permettra à l'obstacle d'être mobile ou non.
- Un angle initialisé aléatoirement entre 0 et π (utile pour la direction du déplacement)

Ainsi, nos obstacles pourront être carré ou bien rectangulaire selon la taille attribuée.
La classe Obstacle contient uniquement un constructeur et une méthode « move » qui déplacera cet obstacle en fonction d'un temps donné.

3) Le senseur

Pour le senseur dans la simulation, nous avons décidé de le représenter par une ligne droite partant de la tête du robot et allant droit devant lui (prenant en compte l'angle du robot).
Nous voulions avoir une classe senseur simple qui pourrait permettre au robot de voir devant lui selon une longueur bien définie.
Pour cela, la classe Senseur doit posséder un seul attribut qui sera sa portée.
Il aura donc un constructeur qui initialisera cet attribut et une méthode « get_distance » qui renvoie la distance entre un objet et le senseur (donc le robot étant donné qu'il est fixé dessus) si et seulement si l'objet est détecté par le senseur.

4) L'environnement

L'environnement doit être un espace simple où évoluera notre robot ainsi que les obstacles.
Il doit posséder plusieurs attributs tel que :
- Des limites (ici, on représentera l'environnement comme un graphe c'est-à-dire avec un axe des abscisses et un axe des ordonnées)
- Un running qui définit si la simulation tourne ou non.
- Un robot et un senseur
- Une liste d'obstacles mobiles et une autre d'obstacles immobiles
- Une liste d'obstacles qui regroupera les deux listes afin d'alléger le code en parcourant une seule liste dans les méthodes de la classe Environnement.

La classe Environnement doit également avoir :
- Un constructeur
- Des méthodes de détection :

.detection_collision_bord_map_robot (qui vérifie que le robot ne dépasse pas les limites de l'environnement)

.detection_collision_bord_map_obstacle (qui vérifie que les obstacles ne dépasse pas les limites de l'environnement)

.detection_obstacle (pour savoir si le senseur détecte un obstacle)

.detection_collision (pour renvoyer les collisions)
- Une méthode « generer_obstacles » qui prend en paramètre un nombre d'obstacles à générer et la vitesse à leur attribuer.
- Une méthode « move_obstacles » qui déplacera les obstacles mobiles.
- Une méthode « update » qui met à jour la simulation.

5) La vision

La classe Vision que nous avons créée, a pour but de récupérer les images capturées par la caméra afin de les traiter.
L'objectif étant de détecter une balise disposant d'un écran composé de 4 carrés de couleurs différentes (rouge, vert, bleu et jaune).
Pour cela, on récupère l'image qui est un tableau 2d où chaque case est un triplet contenant les valeurs RGB des pixels. Puis, on détermine la couleur dominante afin de tester si cette couleur est l'une des 4 qui composent la balise.
Le traitement de l'image se fait via Pillow, une bibliothèque capable de traiter ce format de fichier ainsi qu'avec OpenCV qui nous a permis entre autres de découper l'image étant donné qu'il s'agit de 4 carrés à analyser comme expliqué précédemment.

Cette partie du projet est la dernière, nous avons rencontré plusieurs problèmes notamment car il est compliqué de récupérer l'image et de plus il faut réussir à la traiter correctement après.

III) L'Affichage

Une fois tous les éléments de la simulation créés, nous avons dû choisir comment les représenter. Après s'être renseigné sur la bibliothèque Pygame, qui est une bibliothèque facilitant le développement de jeux vidéo ou encore utilisé pour des applications nécessitant du graphisme, nous avons décidé de choisir cette interface graphique 2D. Pygame possède de nombreuses fonctions qui permettent l'affichage de formes, de lignes ou encore de charger des images.
Ainsi, nous avons créé la classe Simulation_pygame qui permet l'affichage de notre simulation.

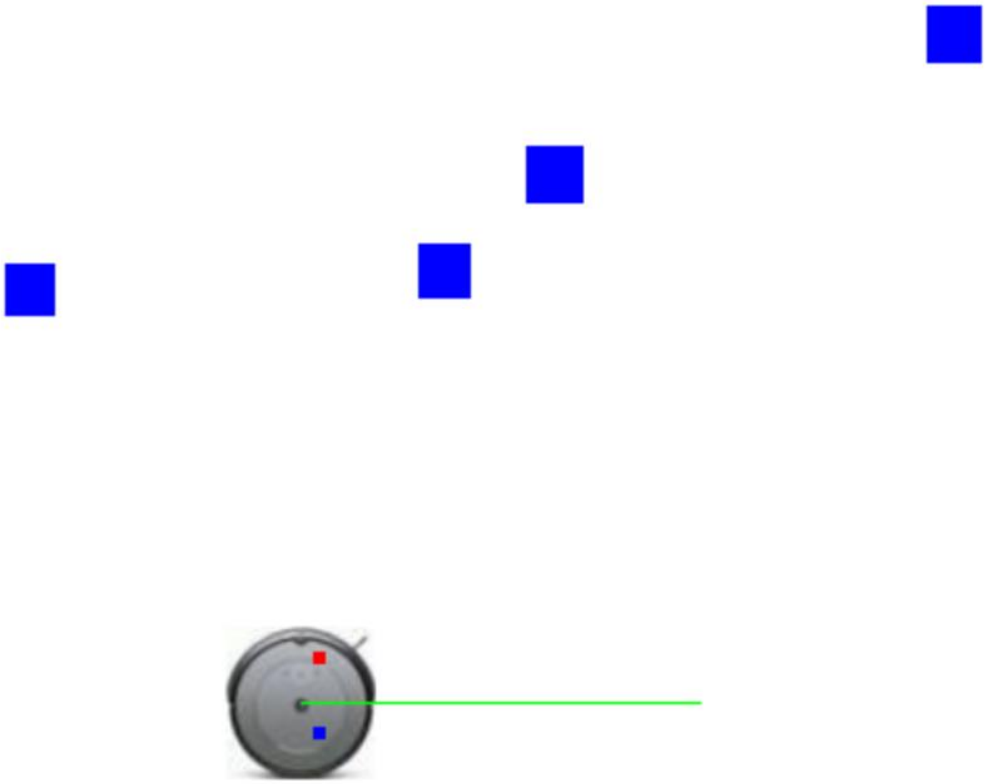
Voici comment nous l'avons codé :
Il faut d'abord un constructeur qui prend en paramètre les limites de notre environnement pour afficher une interface graphique qui aura les mêmes dimensions.
Puis il faut créer différentes méthodes qui permettront l'affichage de nos divers éléments :

- « draw_robot » qui affichage l'image du robot que nous avons sélectionné. Pour cela, elle prend le robot en paramètre, ce qui permet de centrer l'image grâce à sa position et d'effectuer une rotation de l'image en fonction de l'angle du robot.
- « draw_obstacle » qui prend en paramètre les informations de l'obstacle (sa position et sa taille) et affiche ensuite un carré ou un rectangle selon sa taille avec la fonction « draw.rect » de Pygame d'une couleur que l'on choisit.
- « draw_senseur » qui affiche la ligne que représente le senseur avec la fonction « draw.line » en initialisant le début de la ligne sur la tête du robot et la fin de la ligne en fonction de la portée du senseur.

Et pour éviter d'avoir des appels redondants de ces méthodes dans le main, on a créé une méthode « event_update » qui centralise les méthodes d'affichage, créé l'interface graphique et ferme cette interface si la simulation cesse de tourner.
On définit également le nombre d'images par seconde que nous souhaitons afficher via la fonction de Pygame « clock.tick ».

Afin d'obtenir un affichage proche de la réalité au niveau de la vitesse, des distances parcourues ainsi que de la taille de notre environnement et des différents objets, nous avons trouvé que 1 mètre est environ égal à 3800 pixels, cela nous a permis de mieux visualiser les trajectoires et les parcours que fera le robot réel.

Voici une capture d’écran de notre simulation en plein fonctionnement :



On distingue donc les obstacles ici en bleus, le senseur en vert et le robot qui est une image importée. Il y a également deux leds sur le robot (en rouge et bleu) qui sont un ajout personnel afin de tester nos prises d’informations.

IV) Le Controller

A. Le proxy

Pour faire fonctionner la simulation, beaucoup de calculs se produisent en boucle dans notre code. Et tout cela se passait dans chaque fichier de notre projet.
Alors pour éviter d'avoir à répéter des calculs inutilement et pour simplifier la compréhension de notre code, nous avons créé un proxy (ou Décorateur).
Ce proxy comprend tous les principaux calculs que fait notre simulation ainsi que certaines fonctions qui permettent d'alléger le contenu des autres fichiers comme par exemple des accesseurs ainsi que des setteurs.

Etant donné que notre simulation est faite pour faire fonctionner le robot réel mais aussi pour simuler le robot réel, on a donc dû diviser notre proxy en 2 parties :
- Proxy simulation
- Proxy réel

Explicitement, le premier s'occupera des calculs du robot simulé et le deuxième du robot réel.
Principalement, ces différences sont dues à l'interface graphique par exemple la vitesse que l'on donne au robot est multiplié par 3800 lorsqu'on s'intéresse à la partie graphique comme expliqué précédemment, on convertit ici mètre par seconde en pixel par seconde.
Et il en est de même pour les distances, la taille des objets, la portée du senseur, etc…

2) L'IA

Une fois notre simulation fonctionnelle, nous avons rencontré un nouveau problème : Comment faire pour que notre robot puisse exécuter plusieurs tâches au cours de la même simulation ?
Alors tout d'abord, il a fallu créer des stratégies qui correspondent à une tâche que le robot devra effectuer.

a) Stratégie Avancer

Pour faire avancer notre robot, on doit simplement initialiser la vitesse de ses roues. Et si on veut modifier sa trajectoire, il suffit de changer la vitesse des roues (par exemple si on met une plus grande vitesse sur la roue gauche, notre robot se déplacera vers la gauche en continu).Ce n'était pas le cas de notre simulation initialement.
Avant, notre robot se déplaçait de cordonnées en cordonnées, c'est-à-dire que son abscisse et son ordonné évoluait au fil du temps. Cela était donc problématique car notre robot réel ne se déplace pas du tout comme ça.

*distance_parcourue_roue_gauche = (vitesse_roue_gauche*temps*2π*Rayon_roue)/360*

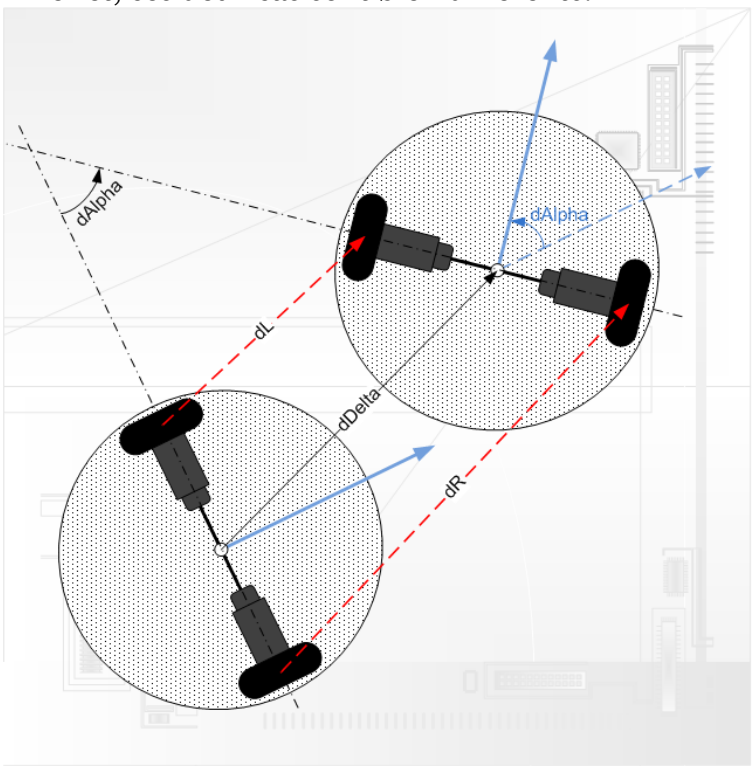
Nous avons donc suivi cette méthode de fonctionnement et avons changé le calcul des distances parcourues, au lieu de prendre une position initiale et une position finale pour ces calculs, on calcule la distance parcourue par chaque roue puis on fait la moyenne.
Et cette stratégie s'exécute tant que la distance parcourue n'est pas atteinte.

b) Stratégie Tourner

Comme expliqué juste avant, pour modifier la trajectoire du robot, il faut modifier la vitesse des roues, c'est ce principe là pour tourner.
On distingue deux cas lorsqu'il s'agit de tourner :

- Tourner sur place
- Tourner en avançant ou reculant

En effet, ces deux cas sont bien différents.



Comme le montre l'image ci-dessous, si l'on veut tourner à gauche tout en avançant, il y aura un déplacement de la roue gauche et de la roue droite.
On remarque également que la distance parcourue lors de la rotation par la roue droite est plus élevée , cela montre que la rotation à gauche se fait grâce à une vitesse supérieure de la roue droite et c'est le même principe pour la rotation à droite.
C'est donc ce principe que nous avons appliqué dans notre stratégie.
En revanche, lorsque l'on souhaite faire un carré parfait par exemple, notre robot doit tourner sur place pour faire un angle droit et non arrondi.
Nous pouvons facilement constater que si la vitesse d'une roue est nulle, prenons ici la roue gauche, son déplacement sera nulle mais le roue droite gardera son déplacement, ainsi la rotation s'effectuera.
Mais il y aura tout de même un déplacement du robot, ce que nous souhaitons éviter.

Alors logiquement, si l'on souhaite que la distance parcourue par la roue droite s"annule, il faut que la roue gauche parcourt la même distance mais dans le sens inverse.
C'est pourquoi, en initialisant une vitesse à une roue et l'inverse à l'autre, on obtiendra bel et bien la rotation sans que le robot avance ou recule.

c) L'IA séquentielle

Une fois nos stratégies fonctionnelles, nous avons réfléchis à comment les faire fonctionner l'une après l'autre.
Pour cela, nous avons créé une classe IA qui prend un seul paramètre, une liste de commandes.

Cette IA possède 3 attributs :
- « statut » qui permet à la simulation de savoir si l'IA marche ou si elle est arrêtée.
- « ia_command » : La liste de commandes.
- « curr_command » qui nous indique quelle commande est en cours d'exécution. (0 pour la première, 1 pour la deuxième ,etc…)

On a également défini quelques méthodes de base qui permettent de gérer notre IA :
- « stop » qui retourne True si sa dernière commande a fini d'être exécutée.
- « ajout_commandes » qui ajoute une commandes à l'IA.
- « select_commandes » qui permet de sélectionner la commande que l'on veut dans l'IA.
- « update » qui exécute les commandes une par une et qui, lorsque la dernière commande de cette IA se termine, arrête l'IA.

Ainsi, pour faire un carré, il suffit d'implémenter à une IA les différentes commandes nécessaires, qui seront :

```
IA = IA ([IA_avance, IA_tourne_gauche, IA_avance, IA_tourne_gauche, IA_avance, IA_tourne_gauche, IA_avance, IA_tourne_gauche])
```

IA_Avance : Stratégie qui permet d'avancer d'une distance mise en paramètre
IA_tourne_gauche : Stratégie qui permet de tourner de 90° (dans le sens trigonométrique donc à gauche)

d) L'IA conditionnelle

Nous avons principalement utilisé l'IA conditionnelle pour les collisions, lorsque notre simulation détectait qu'une collision allait se produire, alors la stratégie de l'IA en cours s'arrêtait et l'on passait directement à la suivante.
Cela permet à notre robot de se déplacer continuellement en évitant les collisions et lui permet de proposer plusieurs trajets, par exemple si notre carré s'arrête à cause d'une collision, on peut enchaîner sur un triangle.

V) Les Threads

Un thread est une partie du code qui s'exécute en parallèle.
Etant donné que notre code est déjà divisé en 3 parties assez distinctes, nous avons créé 3 threads.

Le premier permet la gestion de notre environnement, c'est-à-dire faire les différentes actions qui se passent dans l'environnement , on a notamment : les détections de collisions entre nos objets et aussi avec les limites de l'environnement, le déplacement des objets.

Le deuxième thread est pour l'affichage de notre simulation, donc effectuer les différents affichages (robot, obstacles, ...) en plus de l'interface graphique.

Et enfin, le troisième permet l'exécution des stratégies de l'IA.

Tout cela est nécessaire car sinon on peut rencontrer plusieurs problèmes lors de l'exécution comme par exemple lorsqu'on entame un processus long après avoir lancer notre programme, l'interface graphique peut cesser de réagir tant que ce processus ne sera pas fini.
Et afin d'éviter cet inconvénient, on va lancer un thread qui va exécuter ce processus. Il n'y a plus qu'à attendre la fin du thread, et pendant tout ce temps, l'interface graphique par exemple sera également capable de traiter tout autre événement provenant de l'utilisateur.

VI) Le Main

C'est dans le main que l'on crée tous les éléments de notre projet.
Tout d'abord, on choisit dans quelle situation on veut se trouver, si l'on souhaite faire tourner notre simulation avec le robot virtuel, on crée le robot avec notre classe Robot.
Puis on crée le senseur, l'environnement et l'affichage.

Sinon pour le robot réel, on crée le robot avec la classe de l'API, Robot2IN013 et on a plus besoin de l'environnement étant donné qu'on se réfère directement à la réalité.

Il nous faut ensuite créer nos stratégies , dans le cas où l'on veut un carré, on sait que deux stratégies sont nécessaires : avancer d'une distance quelconque et Tourner de 90 degrés ou -90 degrés selon l'endroit où l'on veut faire le carré.
Une fois nos stratégies initialisées, on peut créer notre IA en l'initialisant avec nos stratégies (de la même manière vue dans la partie sur l'IA séquentielle).

Il ne faut pas oublier d'initialiser une variable temps avant le début de la simulation.

Une fois toutes ces instructions réalisées, on peut donc lancer les threads et démarrer la boucle de notre simulation qui fonctionnera tant que le running de l'environnement est True.
C'est-à-dire dans ce cas-là, tant que nos stratégies ne sont pas effectuées.
A chaque tour de boucle, on calcule le temps écoulé entre le début de l'exécution de la boucle précédente et le début de la boucle actuelle pour que le déplacement du robot en fonction du temps soit calculé très précisément par la simulation.
Ainsi, la valeur obtenue des distances et angles parcourus sera très proche de la réalité.
Et dernière étape, dans la boucle, on met à jour la simulation (l'IA, l'environnement, l'affichage).

VII) Conclusion

1) Difficultés rencontrées

Dès le début du projet, il a fallu mettre en place une organisation qui puissent satisfaire tous les membres du groupe.

Au début, la répartition des tâches était compliquée étant donné que nous ne travaillions pas tous au même moment durant la semaine et que nous créions des tâches trop longues et trop complètes qui prenaient beaucoup de temps à finir (plusieurs heures).
Il a donc fallu créer des tâches plus succinctes, allant de 10 minutes à 1 heure principalement.

Le deuxième problème auquel nous avons dû faire face au début, était GitHub.
Même si nous avions compris comment utiliser GitHub, les clients nous ont fait remarqués plusieurs fois que, comme les tâches, il fallait faire des commits plus succincts.
Par exemple, il faut commit une méthode d'une classe et non un fichier contenant une classe tout entière.
Cela permet un meilleur suivi du projet et une sauvegarde plus précise.
Une fois l'organisation bien acquise, cela nous a grandement aidé dans l'avancement du projet.

Le troisième problème principal rencontré a été le proxy.
Pendant plusieurs semaines, nous avons eu du mal à percevoir exactement ce que les clients voulaient précisément, il ne nous manquait que très peu de changements pour passer sur le robot réel mais bien organiser notre code nous a pris du temps.

Le dernier problème fut le bon calcul des angles parcourus pour le robot réel.
Pour avancer, il n'y avait pas de problèmes, il respectait la distance à parcourir.
Mais pour la rotation, l'angle parcouru calculé était souvent très inférieur à la rotation réelle du robot.
Ce qui a fait que le robot tournait souvent 2 à 3 fois plus que demandé.

2) Remarques sur le projet

Ce projet nous a été très bénéfique que ce soit au niveau de l'organisation, de l'adaptation à différents outils, de la compréhension et de la gestion d'un projet.
Mais aussi il a fallu beaucoup d'imagination car créer quelque chose à partir de zéro demande beaucoup d'efforts, il faut se projeter et imaginer constamment la prochaine étape à réaliser.

Le passage au robot réel est forcément compliqué, on passe d'une simulation à la réalité alors plusieurs facteurs indépendants du code rentrent en compte et peuvent altérer la précision.
Comme par exemple, les frottements des roues sur le sol, l'état du robot, de ses composants.

3) Bilan final

Pour conclure, ce projet nous a permis une meilleure appréhension de ce que l'on pourrait nous demander en entreprise, de comment on pourrait nous le demander et comment répondre à ces demandes. Nous avons également appris à mieux utiliser l'environnement Python, en découvrant de nouveaux modules, de nouvelles bibliothèques. Nous avons pu découvrir comment peut fonctionner réellement une IA. Et nous avons pu apprendre à mieux nous organiser, ce qui est une notion fondamentale et indispensable dans la vie de tous les jours.