Bikou Angie-Sofia

Billon Martin

Takali Aryj

LU2IN013 - Projet robotique

Compte-rendu

Equipe Wall-E

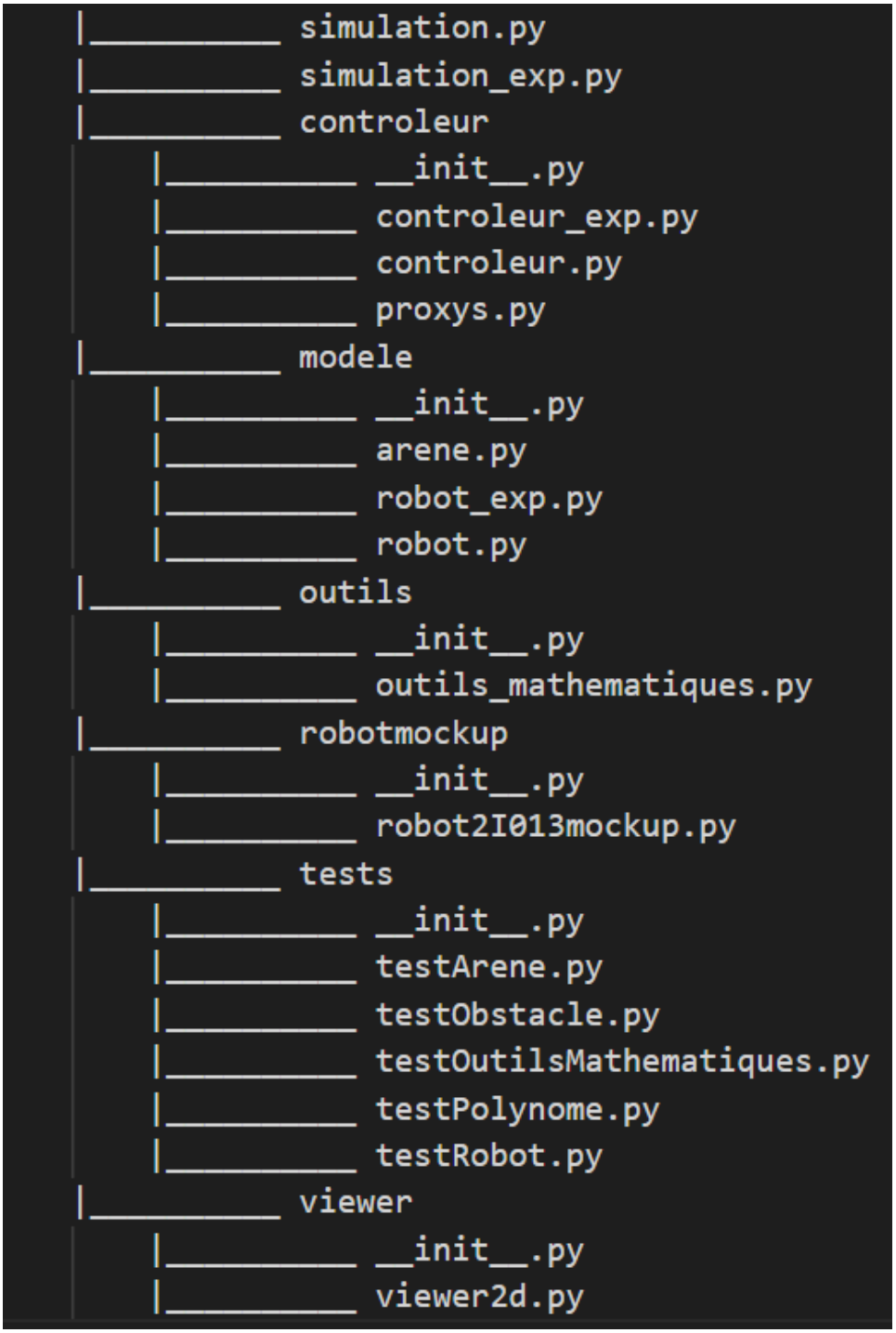
Mardi 25 mai 2021

Sommaire

I. Architecture du projet

1. Architecture globale

Ce projet consiste en la réalisation d'un environnement de simulation pour un robot Dexter GoPiGo. Notre environnement est construit de la manière suivante :



Les scripts contenant les instructions pour le robot se trouvent à la racine et il y a 6 modules disponibles.

2. Module « modele »

Le modèle définit les caractéristiques du robot simulé à travers robot.py ou robot_exp.py ainsi que l'environnement de simulation à travers arene.py.

Le robot simulé principal est le robot Robot comme défini dans le fichier robot.py. Ce robot connaît sa position, sa direction, sa taille, la taille de ses roues et la vitesse de ses roues. Il est capable de :

- donner instantanément une vitesse à chacune de ses roues (méthode set_vitesse)
- se souvenir d'un point dans le temps (attribut self.last_update et méthode reset_time) qui permettra de calculer la distance/l'angle qu'il a parcouru depuis en fonction de la vitesse de ses roues (méthodes distance_parcourue, angle_parcouru_droite et angle_parcouru_gauche)
- mettre à jour les informations sur sa position/direction en fonction de cette distance calculée/cet angle calculé précédemment (méthode se_deplacer)

Le projet comporte un deuxième robot, expérimental, Robot_Exp qui repose sur une modélisation physiquement plus poussée que celle du robot principal. Malheureusement, ce robot n'a pas pu être terminé à temps et ne peut donc pas être utilisé de manière interchangeable avec le robot principal car il lui manque des fonctionnalités pour le rapprocher du robot principal et du fonctionnement global de la simulation. On peut cependant avoir un très bon aperçu de ce que cela aurait pu donner. Son fonctionnement sera décrit dans la deuxième partie de ce compte-rendu.

Dans tous les cas, ce que l'on veut c'est placer un robot dans une arène qui va donc permettre d'observer l'évolution du modèle.

On veut que le modèle soit indépendant du reste du projet donc elle possède son propre thread. On le lance avec la méthode start qui elle-même lance la méthode run redéfinie pour l'arène. La méthode run contient la boucle principale du modèle qui est exécutée toutes les 0.01 secondes environ. A chaque tour de boucle, la méthode update est appelée. Elle demande au robot de mettre à jour les informations concernant sa position et elle appelle aussi get_distance() qui est une des méthodes primordiales de l'arène. Appeler get_distance() constamment en la plaçant dans la méthode update n'est pas nécessaire mais par facilité on a décidé de l'y laisser car la méthode get_distance() permet accessoirement de visualiser la direction actuelle du robot même si ce n'est pas son but premier.

En effet le but de get_distance() est de simuler le capteur de distance du robot réel. Pour ce faire, on calcule le nombre de « pas » que l'on peut faire dans la direction du robot avant d'atteindre un obstacle. Ce pas correspond ici arbitrairement à un diamètre du robot et demi. A chaque fois que l'on tente de faire un nouveau pas, les fonctions est_exterieur_mur() et est_interieur_obstacles() sont appelées. Est_exterieur_mur() compare les coordonnées de l'extrémité avant du vecteur qui représente le dernier pas effectué avec les coordonnées des bords de l'arène.

Est_interieur_obstacles() compare ces mêmes coordonnées avec la position de chaque obstacle placé dans l'arène. Si le pas se trouve en dehors de l'arène ou à l'intérieur d'un obstacle, on ne fait pas de nouveau pas et on renvoie le nombre de pas que l'on a pu effectuer depuis le centre du robot.

Puisque get_distance() calcule donc les coordonnées de point se trouvant dans la direction du robot, on décide de garder en mémoire par les attributs self.robot_dirx et self.robot_diry les coordonnées de ces points pour pouvoir visualiser plus tard l'action de get_distance() et la direction actuelle du robot par la même occasion.

Une autre chose à noter est que le modèle ne va pas évoluer tout seul. On donne à l'arène un contrôleur qui va déterminer les actions que le robot de l'arène doit effectuer. Le modèle arrête sa mise à jour et le thread termine seulement lorsqu'on le contrôleur indique qu'il n'a plus d'ordre à donner au robot.

3. Module controleur

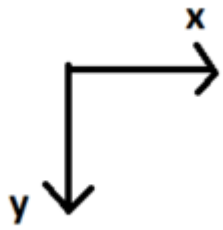
Le module controleur contient les contrôleurs (controleur.py et controleur_exp.py) et les proxys (proxys.py). Les contrôleurs servent à donner des ordres au robot. Pour que le robot simulé d'une arène bouge il faut donc donner un contrôleur à l'arène. Le contrôleur est codé pour qu'il soit très simple d'y ajouter des stratégies. Il est composé de la manière suivante :

- La classe Controleur, qui engendre son propre thread de manière à ce que les ordres soit indépendants du modèle, est la classe principale du fichier. On lui donne une action et elle permet de la démarrer, l'effectuer et arrêter l'exécution du thread lorsqu'elle est terminée.
- Les actions de bases que l'on peut donner à un Controleur sont ParcourirAction, TournerDroiteAction, TournerGaucheAction et StopAction. Ces actions ne sont pas disponibles côté client mais on peut les utiliser côté développeur pour programmer d'autres stratégies plus complexes.
- Pour ces stratégies plus complexes, on passe d'abord par des stratégies qui permettent de manipuler d'autres actions. ConditionAction permet d'effectuer une action tant qu'une certaine condition n'est pas vérifiée et déclenche une action alternative si jamais cette condition finit par être validée. BoucleAction permet d'effectuer une action indéfiniment. En effet, dès que l'action est terminée, cette stratégie fait redémarrer l'action et sa condition done ne sera jamais vérifiée. Enfin, SequenceActions permet de faire une liste d'actions à donner au robot et de les effectuer une à une.
- A partir de ces petites actions de base et de ces stratégies qui permettent de manipuler ces petites actions on peut construire des contrôleurs de plus en plus complexes. Ces stratégies seront décrites en deuxième partie du compte-rendu.

Les proxys quant à eux servent à traduire les ordres du contrôleur vers un robot en particulier. En effet, pour être flexible, le contrôleur doit utiliser un langage universel pouvant s'appliquer à n'importe quel robot réel ou simulé. Nous avons ici un proxy qui assure la liaison avec le robot principal et un autre avec le robot réel.

4. Module viewer

Le module viewer sert à avoir un rendu graphique des informations contenues dans le modèle. Nous avons un unique viewer viewer2d.py. Il permet la visualisation en 2d grâce à Tkinter. Le repère est dans ce sens ci et la direction qui suit l'axe x est l'angle 0 pour le robot.



Le robot est représenté par un carré orange dont le centre correspond aux coordonnées x et y. Les sommets du carré sont alors calculés en fonction du centre et du diamètre du robot par la fonction trouver_sommets().

On peut voir grâce à la flèche marron la direction du robot ainsi que les pas que get_distance() engendrent.

Le robot possède un attribut crayon et la méthode outil_crayon du viewer permet de l'exploiter en laissant une trace verte sur le passage du robot si l'attribut crayon est à True.

On trouve aussi dans le viewer la méthode update qui permet de redessiner la toile après 50 ms grâce à la fonction after de tkinter qui permet de retrouver un comportement similaire aux boucles de nos fonctions run des modules précédents (l.66) Les appels à la fonction update cessent et l'affichage se termine en fermant la fenêtre.

Ainsi pour lancer l'interface graphique il suffit de lancer d'abord la méthode update puis d'utiliser la méthode mainloop() de tkinter. Son appel est blocant, c'est pour cela que le lancement de l'interface graphique est la dernière chose que l'on fait dans le script de simulation.

5. Module robomockup

Le module robotmockup permet de s’assurer que les instructions d’un script seraient comprises par le robot réel.

6. Module outils

Ce module contient les objets Points, Vecteur et Polynome qui sont utilisés avec le robot expérimental.

7. Module tests

Au cours de la réalisation de notre programme, il est très important de tester quelques portions de code pour s'assurer du bon fonctionnement de celles-ci (pour que le code ne régresse pas après des modifications sur des fonctions).

La préparation nécessaire au déroulement des tests :

- D’abord, on a créé un répertoire séparé pour les tests que l'on appelle : tests.
- Ensuite, on y crée des fichiers pour tester nos « classes",
  (testNomDuFichier.py).
- Après dans chaque fichier on importe le module unittest fournissant une classe de base, TestCase « une classe fille » pour permettre l'assertion (import unittest), et pouvant être utilisée pour créer de nouveaux scénarios de test (sans oublier les imports des classes nécessaires pour assurer le bon fonctionnement des tests). Le module sys fournit un accès à certaines variables système utilisées et à des fonctions interagissant fortement avec ce dernier
- On créé, par la suite, une classe qui hérite de ce module et de la classe TestCase qui permettant de tester des cas pour des scénarios bien précis (class TestNomDeLaClasse(unittest.TestCase)).
- On prépare les données qui seront exécutées dans la méthode setUp( ) avant chaque méthode test.
- Puis, on crée les méthodes de manière isolée. Ces tests individuels sont définis par des méthodes dont les noms commencent par test.
- Le cœur de chaque test est un appel à:

  assertEqual() pour vérifier un résultat attendu ;

  assertTrue() ou assertFalse() pour vérifier une condition ;

  assertRaises() pour vérifier qu'une exception particulière est levée.

- Enfin, on appelle à la méthode main pour exécuter tous les tests un à un et éventuellement présenter les résultats de tous les tests sur une ligne de commandes
- Et pour nettoyer des données on pourrait utiliser la méthode tearDown( ) après chaque méthode test.

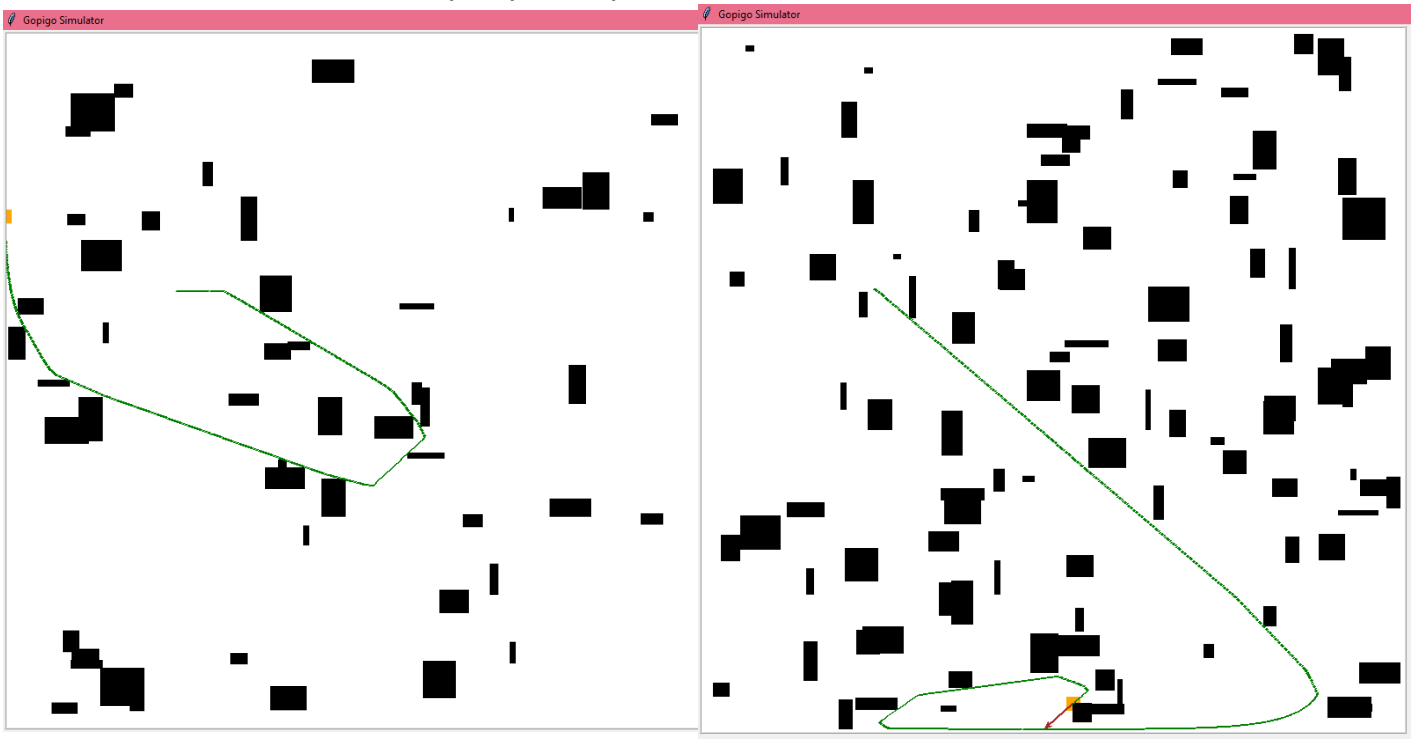
II. Capacités de la simulation

1. Capacités du robot principal

Les capacités de ce robot sont les suivantes :

- Carre : il permet au robot de dessiner un carré (sans se soucier des obstacles)
- AvanceJusquAuMur fait avancer le robot jusqu'au premier obstacle qu'il rencontre
- TourneAvanceStop fait effectuer une rotation au robot dans le sens des aiguilles d'une montre s'il elle est positive, dans le sens contraire si elle est négative, puis fait avancer le robot jusqu'au mur
- AvanceJusquAuMurPuisNouvelleDirection est le premier pas vers une stratégie autonome. Elle permet au robot de trouver une nouvelle direction lorsqu'il se trouve coincé devant un obstacle
- BoucleSurTourne permet comme son nom l'indique de tourner en continu
- En combinant AvanceJusquAuMurPuisNouvelleDirection et BoucleSurTourne, on arrive à une première stratégie autonome qui est appelée Parcours Autonome. Cette stratégie pourrait être grandement améliorée en rendant par exemple le sens de rotation aléatoire lors de la recherche d'une nouvelle direction (pour l'instant c'est toujours vers la droite arbitrairement) ou encore en changeant de direction aléatoirement même lorsqu'il n'y a pas d'obstacles à proximité. Enfin la fonction get_distance() de l'arène n'est pas parfaite puisqu'elle ne permet de vérifier que strictement dans la direction en face du robot alors qu'on aimerait pouvoir envoyer des pas à 45° à gauche et à droite par exemple pour mieux éviter de se retrouver dans le mur.

Ci-dessous le tracé de quelques expériences avec ParcoursAutonome.



2. Capacités du robot expérimental

Dans ce projet nous avons deux modélisations différentes pour le robot : une plus simple et plus directe et une un peu plus complexe qui permet de visualiser la trajectoire en fonction de l'accélération et en fonction des paramètres de trajectoires précédentes. Nous allons à présent parler de cette seconde modélisation.

Tout d'abord, ce robot prend en paramètre ses coordonnées dans l'arène pour savoir où le placer, il a également quelques méthodes pour les changer directement et les afficher. La méthode qui est la plus importante ici c'est changePosition qui prend en paramètre l'accélération et l'angle et qui va être invoquée pour chaque nouvelle trajectoire, quand on veut changer la vitesse ou la direction du robot.

Cette méthode fonctionne en utilisant un principe de la physique appelé cinématique. Dans un premier temps, on calcule la durée qui s'est écoulée depuis qu'on a appelé changePosition pour la dernière fois (last_update), donc depuis le dernier changement de trajectoire, et on la stocke dans somme_temps. Après on calcule les valeurs des autres variables à l'instant de l'appel de la méthode (donc en fonction de somme_temps) : les coordonnées x et y avec le vecteur position, la norme du vecteur position avec les valeurs à l'appel, puis les coordonnées xa et ya 0.1 secondes avant l'appel de changePosition ainsi que la norme d'un vecteur indiquant la dernière direction, le vecteur vitesse à l'appel et la vitesse du robot elle-même. Après cela un nouveau vecteur accélération est créé en fonction de la dernière direction du robot, donc que l'angle en paramètre correspond bien à celui entre l'ancienne direction et la nouvelle. C'est à ça que servent xa et ya, pour ne pas avoir la nouvelle direction en fonction de l'orientation du repère de l'arène. Dans la suite le vecteur vitesse et le vecteur position sont calculés par le principe de la cinématique, en calculant les primitives en fonction des valeurs initiales. Donc nous obtenons bien le vecteur position qui va nous permettre de modéliser la trajectoire d'un robot.

Il y a d'autres méthodes comme reset_time qui sert à réinitialiser last_update après l'appel de changePosition, distance_parcourue qui à chaque appel calcule une valeur xd et yd en soustrayant les coordonnées à l'appel et celles 0.1 seconde avant pour calculer la distance parcourue pendant ce temps et l'ajouter à une variable distance, et se_deplacer qui est invoquée dans arene.py et qui calcule les coordonnées du robot à l'appel à l'aide de positionVecteur. Cette dernière méthode calcule également l'angle de direction par rapport au repère de l'arène ainsi que la vitesse du robot.

À part Point et Vecteur ce robot utilise polynome.py qui prend en paramètre trois valeurs et permet de représenter un polynôme de degré 2 en l'affichant avec la méthode affichePolynome et aussi de calculer le résultat en fonction d'une valeur en paramètre dans la méthode calcul (qui va surtout être somme_temps). Maintenant nous allons parler des différentes stratégies liées à ce robot dans le contrôleur.

La première stratégie et la plus importante est Trajectoire, elle prend en paramètre le robot, la durée de la trajectoire, l'accélération et l'angle souhaités. La première action effectuée est l'appel de changePosition qui crée le vecteur position nécessaire puis reset_time qui réinitialise last_update, après update s'effectue jusqu'à ce que le temps écoulé atteigne la durée en paramètre. À chaque fois que Trajectoire est appelé (ou une des autres stratégies de trajectoire) le nouveau vecteur position est créé en fonction de la trajectoire précédente : coordonnées initiales, nouvel angle de direction et vitesse en fonction de la vitesse que le robot a déjà (le changement n'est donc pas instantané).

Il y a d'autres stratégies similaires en fonction d'autres paramètres : TrajectoireVitesse s'effectue jusqu'à ce que la vitesse souhaitée soit atteinte (après on peut appeler une stratégie de trajectoire avec en paramètre une accélération de 0 pour garder la vitesse constante) et TrajectoireDistance s'effectue jusqu'à ce que la distance souhaitée soit parcourue (en invoquant la méthode distance_parcourue).

La stratégie Stop est utilisée pour stopper le robot (comme il se déplace à une certaine vitesse ce n'est pas instantané) en le ralentissant avec une accélération négative passée en paramètre jusqu'à un moment juste avant qu'il commence à reculer. Sa vitesse n'est pas exactement 0 mais elle est assez basse pour qu'on puisse utiliser la stratégie Reset_vitesse qui la met à exactement 0.

Une dernière stratégie est Tourner qui est une séquence d'actions qui permet de faire un virage beaucoup plus serré en ralentissant le robot jusqu'à 10% de sa vitesse initiale, le tournant par rapport à un certain angle rapidement, ralentissant rapidement puis plus lentement pour atteindre la vitesse initiale.

III. Répartition du travail

Le travail a été fait comme suit :

- Aryj : module tests
- Martin : robot expérimental avec ses stratégies, classe Polynome de outils_mathematiques
- Angie : modules modele, controleur et viewer