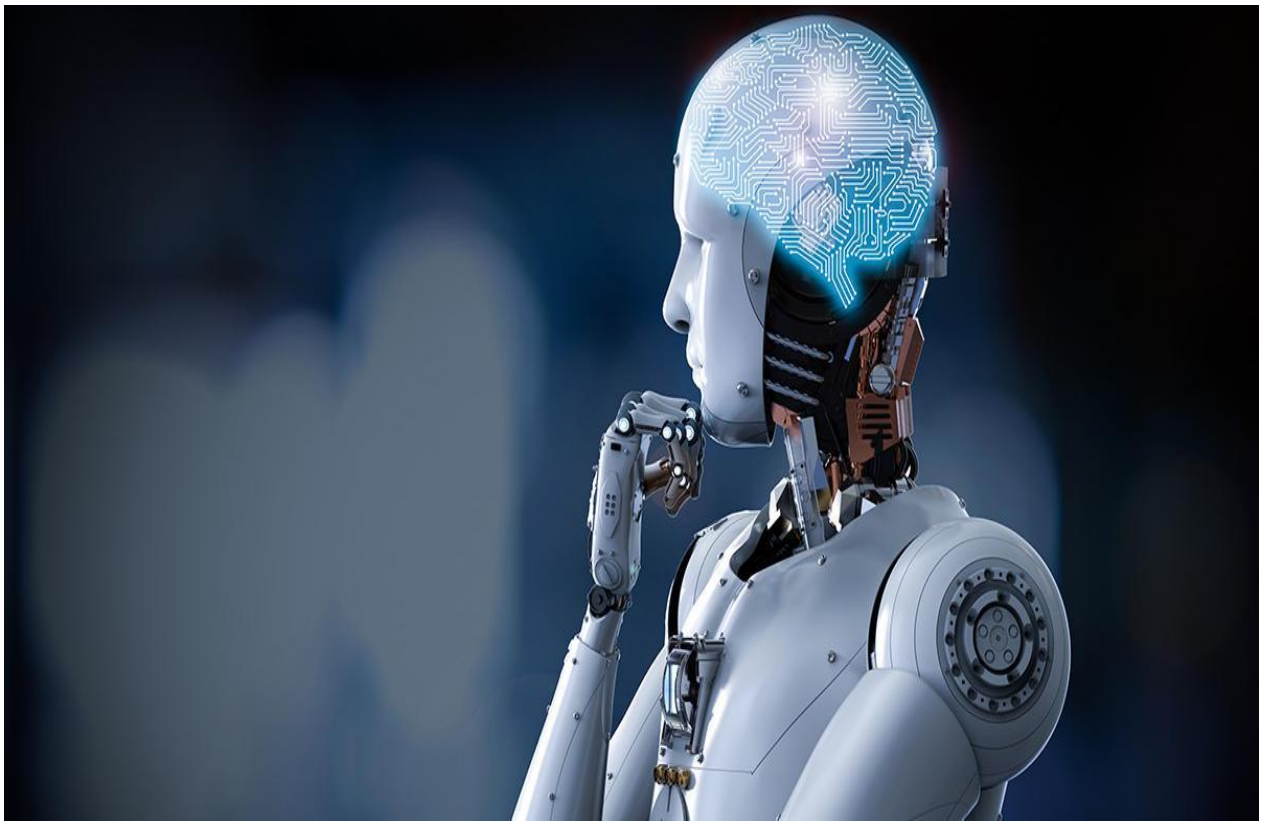
LU2IN013 : Projet Robotique

Réalisé par Cyborg Dance Troupe



Ben Salah Adel

Kone Daba

Makhlouf Djida

Boudouaour Akram Amine

Varatharajah Sanjai

Sommaire :

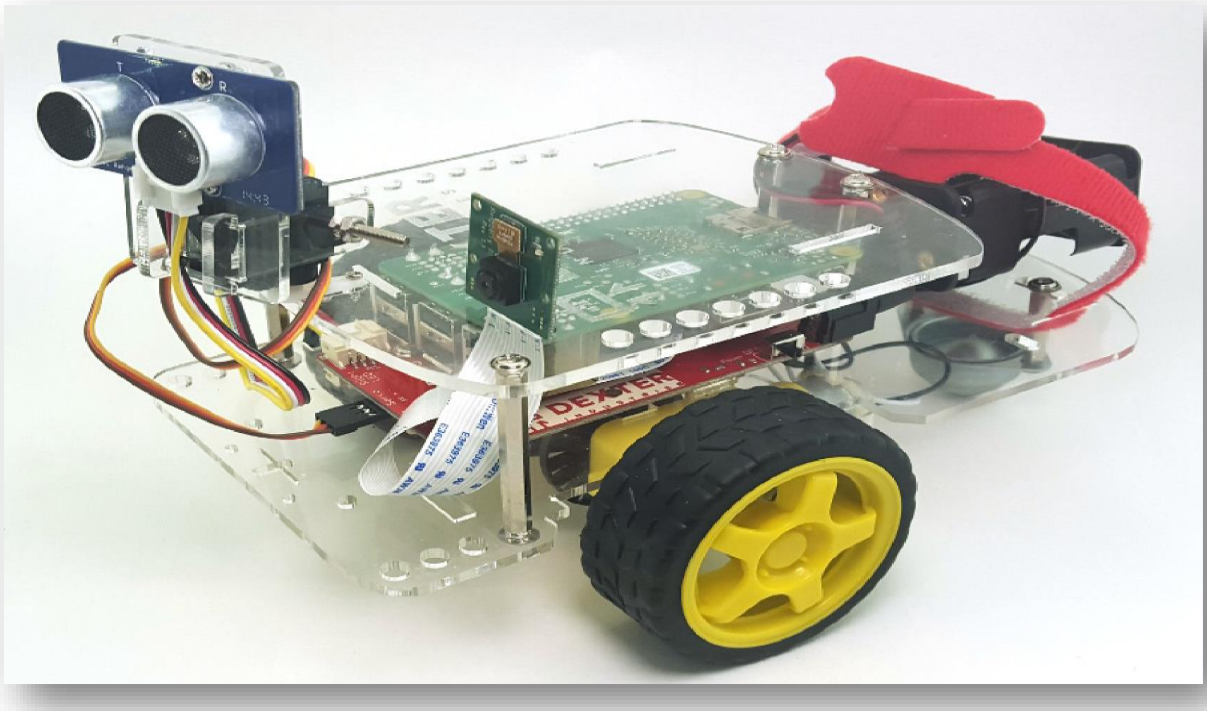
Introduction

Demandes du Client

Dans le cadre de ce projet basé sur la robotique, nous devons rendre un code organisé dans le langage Python répondant à une problématique données : Permettre au robot nommé Dexter d'effectuer différentes taches plus ou moins compliqué.

Le Projet est composé de deux parties, la 1ere est de réalisée des tâches données, comme réaliser un carré, détecter une balise et également avancer le plus vite possible et s'arrêter devant un mur. La 2nde est une partie libre où nous somme libre d'effectuer n'importe quelle tache et les proposer au Client.

Le Robot Dexter est composé d'un Raspberry PI, d'une carte contrôleur Arduino et de deux moteur encodeur pour les roues et de 3 senseurs, une caméra, un capteur de distance et un accéléromètre.



Objectifs

Afin de réussir à gérer ce projet efficacement, nous avons opté sur les principes de l'Agile et nous utiliseront Trello et Git pour créer et répartir les taches dans le groupe. Le but de base,

que l'on sépare en plusieurs sous objectifs, est de créer en utilisant Python et de différents modules, un environnement de simulation assez réaliste, mais le plus simple possible avant d'utiliser et contrôler le Robot grâce aux simulations que l'on aura effectuée.

Avant donc de démarrer, il fallait définir comment nous allions commencer tout cela, nos idées de départ.

Idées de Départ

Pour la modélisation du robot : nous avions vu en cours la géométrie nécessaire à une modélisation du robot et également une façon de simuler une arène et des obstacles existant dans cette arène.

Il fallait également trouver quelle bibliothèque graphique utiliser pour effectuer, au départ nous avions choisi Matplotlib, après quelques tests et simulations que l'on précisera plus tard, nous avons finalement décidé de changer et d'opter finalement pour Tkinter. Cette bibliothèque est très facile d'accès et d'utilisation et présente dans beaucoup de distributeur Python. Également dans celle de l'installation Linux de la fac.

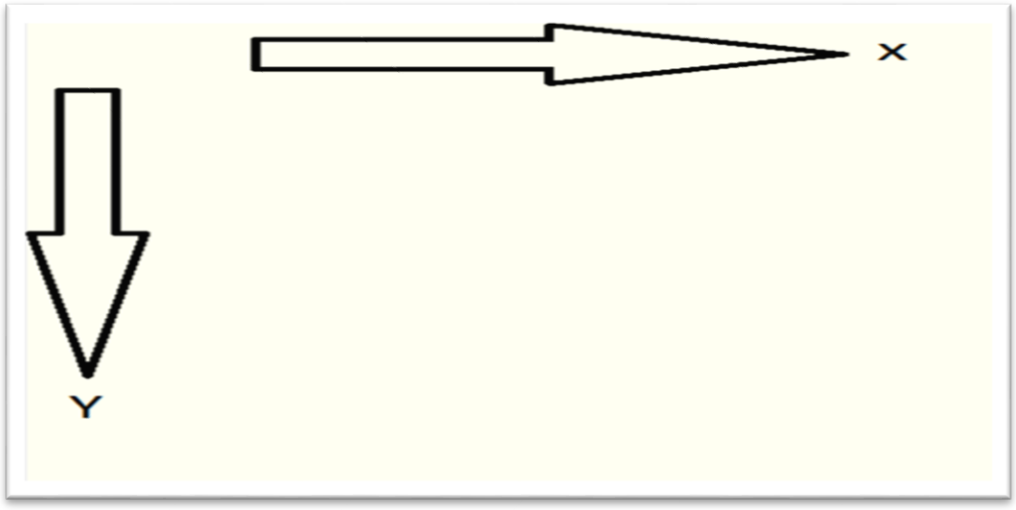
Création du Robot et modélisation de l'Arène

On créer un fichier *robot.py* et un fichier *arene.py* pour simuler l'arène

Classe Robot : *robot.py*

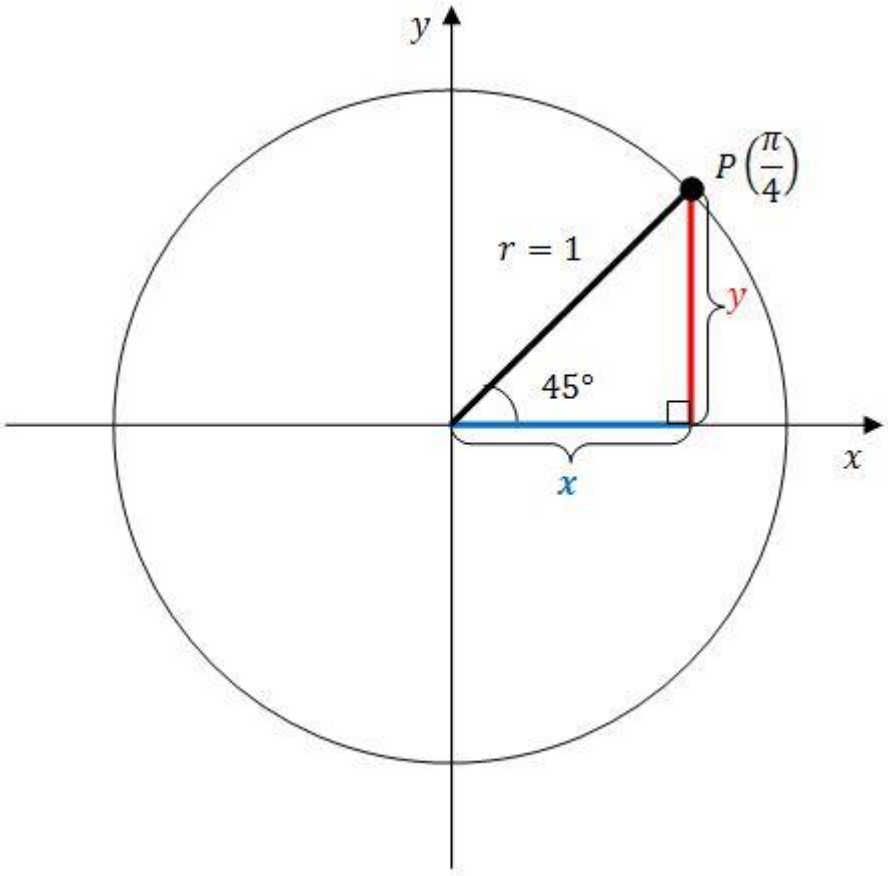
On simplifie le modèle du robot au maximum. On oublie pour l'instant le physique du robot et les roues qu'on aura rajoutées plus tard dans l'équation. Dans notre simulation, le robot est représenté par un cercle de couleur bleu aux coordonnées (x,y), sa position initiale. Pour pouvoir gérer les déplacements du robot, on ajoute un attribut orientation qui va servir de base pour les calculs de direction et déplacement.

Dans certaines bibliothèques graphiques dont Tkinter en fait partie, l'axe des abscisse x et ordonnée y sont représenté comme ceci :



Nous utiliserons donc cette convention d'axe par la suite.

L'attribut orientation d'un robot est un angle allant de 0° à 360°, on représente cet angle tel que sur un cercle trigonométrique.



Nous simulons les propriétés des roues avec les attributs *vitesse_roue_gauche, vitesse_roue_droite, rayon_roue et distance_roue et distance_sens*. Correspondant respectivement à la vitesse de la roue gauche, de la roue droite, le rayon des deux roues et la distance entre les deux roues et le sens des roues.

En ayant initialiser les roues, nous avons pu utiliser la vitesse moyenne des roues pour calculer a l'aide des formules trigonométriques, les coordonnées (x1,y1) après déplacement du robot.

vitesse_moyenne = (self.vitesse_roue_gauche + self.vitesse_roue_droite) / 2

*dx= vitesse_moyenne*cos(self.orientation)*delta_time*

*dy= vitesse_moyenne*sin(self.orientation)*delta_time*

x1=self.x+dx

y1=self.y+dy

Avec *delta_time* qui vaut un pas de temps spécifié grâce l'attribut tmp et la fonction time() venant du module time.

Les vitesses des roues sont initialiser par la fonction *set_vitesse* avec une vitesse passée en paramètres pour chacune des roues. Si on réutilise la fonction plusieurs fois on met à jour la vitesse des deux roues.

Les coordonnées du robot sont de base stocker dans des constantes x et y et sont ensuite grâce à la fonction *deplacement* modifier, stocker toujours dans les mêmes variables mais avec des coordonnées différentes grâce aux formules si dessus. Les attributs à chaque utilisation de la fonction *deplacement* sont mis à jour, l'orientation change également grâce à un enchainement de formule prévu à cet effet :

*vitesse_angulaire = (self.vitesse_roue_droite - self.vitesse_roue_gauche) / (2 * math.pi * self.distance_roues)*

*delta_orientation = vitesse_angulaire * delta_time*

Nouvelle_orientation=self.orientation + delta_orientation

Où la vitesse angulaire représente la vitesse de rotation autour de son axe centrale.

Classe Obstacles : ***Obstacle.py***

Avant de bien préparer notre arène, il faut y rajouter des obstacles, actuellement il nous fallait juste créer une classe avec des attribut pour pouvoir l'utiliser dans notre future classe Arène et également pour notre fichier *robot.py* expliqué plus haut.

Cette classe étant composée au minimum de coordonnée x,y est d'un rayon pour la taille de l'obstacle qui peut être différent d'un obstacle à l 'autre.

Classe Arène : ***arene.py***

Le fichier est constitué des valeurs de la longueur et largeur de l'arène. Nous avons décidé de les fixés à 400m.

La classe Arène est composé de 3 attributs, le robot en lui-même, une liste d'obstacles, on rappelle que les deux ont été définis dans leur propre fichier. C'est pour cela que l'on rajoute tout en haut du fichier ceci :

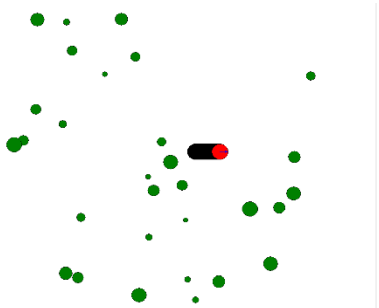
```
import math
from .robot import Robot
from .obstacle import Obstacle
from random import *
import time
```

Cela importe les classe Robot et Obstacle qui viennent de leur propre fichier dans le fichier *arene.py.*

La classe Arène est aussi composé d'un attribut dt qui correspond au pas de temps entre chaque mise à jour de l'arène. Il y a une fonction ajout_obstacle qui rajoute à chaque utilisation 1 obstacle en plus dans l'environnement. Selon la taille de l'arène, on peut ou non ajouter un grand nombre d'obstacles sans que le robot qui avance en continue puisse entrer en collision avec un Obstacle.

En parlant de collision, dans ce fichier il y a également une fonction check_collision (on aurait pu mettre cette fonction dans le fichier *obstacle.py*) qui permet de savoir si le robot a touché les coordonnées d'un obstacle ou les coordonnées des bords de l'arène durant la simulation, si c'est le cas la simulation s'arrête immédiatement. A ce moment-là du développement, c'était même la seule condition d'arrêt de la simulation.

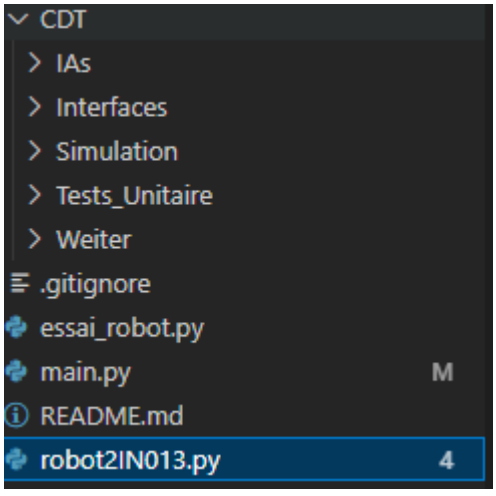
Une fois l'arène créée, il peut être représenté graphiquement avec notre classe Affichage dans un autre fichier Affichage qui utilise la bibliothèque graphique Tkinter. Cette classe initialise tous les éléments qui pourrait se trouver sur l'arène comme le robot, les obstacles, la flèche qui indique dans quelle direction avance le robot...



Le point rouge représente le robot, la ligne noir le chemin que le robot prend et les points verts représente les obstacles et la ligne bleu dans le point rouge représente l'orientation du robot.

Organisation du code

Le code est répertorié sous plusieurs couches : notre main et nos fichiers pour l'utilisation réelle du robot sont visible immédiatement dans nos fichiers.



Weiter

Ce sous-dossier contient la partie contenant nos variables globales et la modélisation du robot réelle fourni par le professeur. (*constantes.py et robotmockup.py)*

Simulation

Contient les fichiers expliqué précédemment c'est à dire le fichier robot.py qui nous permet d'instancier notre robot virtuel, arene.py qui représente l'environnement de notre simulation et obstacle.py qui nous permet d'instancier les obstacles, tous indispensables à la simulation

Interfaces

Le sous-répertoire Interfaces contient le fichier affichage.py dans lequel on utilise la bibliothèque Tkinter afin de réaliser l'affichage graphique de notre simulation en 2D et affichage3d.py qui gère la 3d avec l'aide de la bibliothèque panda3d (même si elle n'a pas été abouti)

IAs

Le sous-répertoire IAs contient différents fichiers qui ont pour but de réaliser nos stratégies et parmi elles on a les fichiers commencent par ia qui sont des fichiers contenant des classes qui ont pour but de donner des instructions au robot . Nos ias sont basé sur un décorateur avec 3 méthodes : start, step et stop . Start permet de démarrer la consigne de l'ia, step permet de mettre à jour l'état de la consigne de l'ia et stop permet de vérifier si la condition d'arrêt de la consigne a été vérifié .

Parmi les fichiers ia on a ia.py et ia_tourner.py qui contiennent les classes IA et Ia_Tourner , IA permet de faire avancer le robot d'une certaine distance et IA_Tourner qui permet de faire tourner le robot d'un angle donné. Puis on a le fichier ia_seq.py qui est une liste d'ia à exécuter séquentiellement et ia_loop.py qui permet de répéter à l'infini l'instruction d'une ia (ia.cond n'est pas en état d'être utilisé mais avais pour but d'exécuter l'instruction d'une ia si une condition booléenne était vérifiée). Et pour faire le pont entre les instructions des ias et le robot notre sous-répertoire contient deux fichiers : intermediaire.py et inter_robot.py. Intermediaire.py contient la classe Intermediaire qui a des méthodes permettant d'obtenir la distance et l'angle que le robot a effectué sur une période donnée et aussi d'initialiser les roues à une certaine vitesses. Intermediaire.py pour le robot virtuel et inter_robot.py pour le robot réel.

Test

Ce sous répertoire contient les fichiers où on été effectué les tests unitaires pour les différentes fonctions qui ont été codés précédemment. On rappelle qu'un test unitaire sert a tester une partie du bon fonctionnement d'un programme, cela sert a vérifier a termes si tout fonctionne parfaitement bien. Avec l'utilisation du module unittest, on a pu réaliser les tests unitaires demandés. Voici pour exemple le test unitaire du fichier *robot.py*.

```python
import unittest        adelbns1, 4 months ago • Ajout du fichier test unitaire de
from CDT.Simulation.robot import Robot
from CDT.Simulation.obstacle import Obstacle
You, 2 months ago | 3 authors (You and others)
class TestRobot(unittest.TestCase):
    def setUp(self):
        self.r=Robot()
    def test_start_time(self):
        self.r.start_time()
    def test_set_vitesse(self):
        self.r.set_vitesse(6.4,5)
    def test_deplacement(self):
        self.r.deplacement(0.3)
    def test_senseur_distance(self):
        self.r.senseur_distance([Obstacle(4.0,4.0,5.0),Obstacle(8.0,6.0,2.0)])

if __name__ == '__main__':
    unittest.main()
```

Branches

Sur le Github du groupe Cyborg-Dance-Groupe, on travaille sur la branche main, qui est notre branche principale. En parallèle, nous avons une autre branche Dev servant de brouillon, lorsque nous avions des idées venant d'un coup, nous les mettions dans la branche Dev pour pas que le vrai projet soit modifié définitivement.

Stratégies

Les stratégies sont aussi connues sous le nom de "behaviour patterns" ou modèles comportementaux en Français. Celles-ci permettent au robot d'effectuer des tâches en lançant un algorithme indépendamment du type des instructions données.

En effet, nous pouvons séparer nos stratégies en deux catégories de stratégies :

- Les stratégies simples
- Les stratégies avancées

Tout comme leurs noms, nos stratégies sont implémentées différemment dans notre programme. Effectivement, on peut remarquer que les stratégies simples sont des classes du module IAs, par exemple "ia.py" ou "ia_tourner", tandis que les stratégies avancées sont en fait des fonctions directement codées dans le main qui utilisent les stratégies simples afin de faire des figures plus complexes.

Les stratégies simples

Comme dit précédemment les stratégies simples sont au nombre de deux dans notre code :

ia.py :

Il s'agit d'une ia qui permet à notre robot d'effectuer un mouvement linéaire avec une distance spécifié autrement dit d'avancer. Comme toute classe, elle possède un constructeur qui initialise les attributs tel que la vitesse du mouvement la distance parcourue et celle à parcourir. Une méthode

start qui enclenche le mouvement linéaire, une méthode step qui met à jour les distances parcourues et une méthode stop qui vérifie que le mouvement a bien été effectué.

Ia_tourner :

La classe Ia_Tourner représente une IA spécialisée dans la rotation à gauche. Elle dispose de fonctionnalités de démarrage, de progression et d'arrêt. Son constructeur initialise les attributs, tandis que la méthode start permet de lancer la rotation. La méthode step assure la mise à jour de l'angle parcouru et l'ajustement éventuel de la vitesse. Enfin, la méthode stop vérifie si la rotation est terminée. Cette IA est conçue pour ralentir la rotation une fois qu'un certain pourcentage de l'angle total est atteint.

Les stratégies avancées

Les stratégies avancées sont des fonctions qui implémente une liste d'ia qui a comme élément de la liste les ias simples décrites au-dessus. Cette liste d'ia sera ensuite donnée en paramètre à une ia séquentielle qui dans notre code est une classe du module IAs nommé "ia_seq".
La classe ia_seq gère une séquence d'IA (Intelligence Artificielle) en utilisant une liste d'IA spécifiée lors de l'initialisation. Elle possède les méthodes start, step, et stop pour démarrer, avancer d'une étape et vérifier si la séquence est terminée respectivement. L'indice cur est utilisé pour suivre la position courante dans la séquence. La méthode start démarre l'exécution de la première IA, tandis que step passe à l'IA suivante lorsque l'IA courante est terminée. La méthode stop renvoie True si l'indice cur dépasse ou atteint la longueur de la liste d'IA, indiquant la fin de la séquence. En résumé, IA_Seq permet de gérer une séquence d'IA en effectuant les actions appropriées à chaque étape.

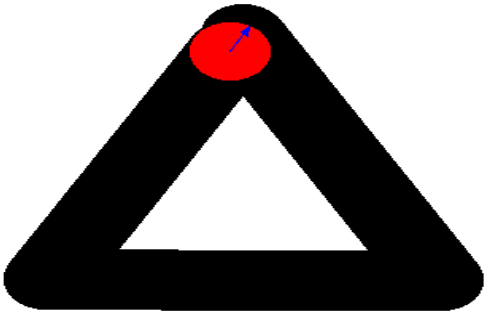
Par ailleurs notre fichier main possède quatre stratégies avancées : create_rectangle, create_carre, create_triangle_equilateral, create_hexagone.

Résultats obtenus pour les différentes stratégies

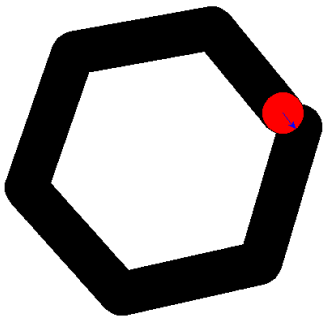
Carré :

Rectangle :

Triangle équilatéral :



Hexagone :



Proxy et API du robot

De manière à ce que l'on puisse communiquer les instructions de nos stratégies à notre robot simulé et également à son équivalent matériel, on crée un 'adaptateur' servant d'intermédiaire entre l'API du robot de la simulation et celui du réel.

Pour ce faire, nous avons dû faire deux adaptateurs, un qui permet de faire l'intermédiaire entre l'API du robot de la simulation vers celui du réel (le fichier appelé *intermediare.py* dans le sous répertoire IAs). Et un autre faisant bien évidemment l'inverse qui fait l'intermédiaire entre l'API du robot du réel et celui de la simulation (le fichier appelé *inter_robot.py* dans le sous répertoire IAs).

Le principe est de créer de classe Intermédiaire et Inter_Robot dans lesquelles on a des fonctions appelant les fonctions des APIs qui ont le même comportement mais nommées différemment. On donne aux fonctions de ces deux classe un intitulé identique de façon à ce que l'appel d'une fonction fonctionne avec chaque API.

Tests de nos simulations vers le Robot réel

Maintenant que nous pouvions utiliser le robot réel grâce à nos intermédiaires et nos stratégies. Il fallait que l'on teste nos simulations de figure sur le robot réel.

Dans notre fichier *essai_robot.py*, nous avons adapter nos créations de figure à l'API du robot réel grâce à nos classe Inter_Robot et également toutes nos stratégies.

Pour faire a notre robot une simple ligne cela n'as pas été très compliqué, la ligne étant très facile a effectuer, nous avons pensé que faire les différentes figures seraient donc plutôt facile, après plusieurs essais, nous avons vu que ce n'était pas le cas.

Carré

Pour le test du carré, le plus dur a été de le faire tourner a 90°, en effet soit il ne tournait pas du tout, ne soit pas complètement, ou soit un peu trop. Après plusieurs tests, on s'est dit qu'il fallait

modifier un petit notre classe IA_Tourner pour que cela fonctionne mieux. Après quelque modification, le robot a enfin réussi à faire un carré.

Rectangle

Après avoir réussi le carré, le rectangle n'était pas du tout compliqué, la seule vraie différence a été de mettre deux valeurs différentes pour la longueur et largeur. La seule vraie difficulté ici et encore c'est vraiment mineur était de ne pas confondre la largeur et la longueur, parce que si on confondait, cela ferait un rectangle totalement différent.

Triangle Equilatéral

La différence comparée aux figures présentés précédemment concerne l'angle, un angle de 90° étant désormais interdit sinon cela n'effectuerait absolument pas un triangle équilatéral. Il fallait choisir un même angle pour toutes les fois où le robot tournerais, la seule difficulté ici était de bien choisir la direction dans laquelle le robot partirait, il pouvait très bien partir dans le bon angle mais dans une direction opposée. C'est grâce au cercle trigonométrique que nous avons pu bien réaliser ce triangle équilatéral à la perfection.

Hexagone

L'hexagone était très facile à réaliser, comme nous avions réussi les autres figures, l'hexagone n'était pas si différent à effectuer, comme toujours il fallait choisir un angle dans a bonne orientation grâce au cercle trigonométrique, et effectuer la même séquence de mouvement 6 fois pour que la figure soit créer, grâce à notre classe expliqué précédemment Ia_Seq.

Echecs du Projet

Dans notre projet, bien que nous ayons réussi dans l'ensemble nos objectifs, il y a malheureusement quelques travaux qui n'ont pas pu aboutir.

Modélisation 3D

En plus de la modélisation 2D avec Tkinter, notre groupe a essayé de réaliser la modélisation 3D de notre simulation afin de pouvoir effectuer une simulation de la caméra du robot. Cependant, ce n'a point abouti par une tentative de rendu en 3d de l'arène et robot à partir de l'API Panda3D trouvable sur le fichier *affichage3d.py* dans le sous-répertoire Interface. Les causes de cet échec peuvent être expliqué de par le commencement tardif du rendu en 3D de notre groupe suite aux multiples retards engendré par le groupe ont causé des multiples réécritures du code lors des 4-5 premières semaines afin d'avoir une base solide pour continuer le projet. Et de par notre in-expérimentation totale dans le rendu 3D qui nous a, au début pousser à prendre certaines API comme Unity,OpenGI et ModernGI qui furent trop compliquer et nous a pris beaucoup de notre temps.

On s'est mis ensuite d'accord pour travailler sur Panda3D. Cependant, même si Panda3d était moins compliquer pour nous, il n'en restait pas facile non plus, commençant a s'approcher de la fin du semestre, nous avons finalement choisi de se focaliser sur la partie Proxy et API du robot pour les tests expliqués plus tôt.

Détection de Balise

L'autre échec de notre semestre est sur la détection de balise qui était une idée de départ que nous aurions dû réaliser d'ici la fin du semestre. Cependant le début fut extrêmement compliqué pour notre groupe ce qui nous as poussé à faire l'essentiel. C'est-à-dire nos stratégies avancées de traçage de figure tel que le carré et de laisser cette idée en arrière-plan. Pour avancer dans cet objectif, on avait fait une méthode pour le robot qui devait pouvoir détecter et renvoyer la distance de l'obstacle le plus proche dans la direction du robot. Sauf qu'à ce moment-là, on avait pour objectif de réaliser notre affichage 3D qu'on a jugé plus important que la balise surtout que si on avait réussi. On aurait pu faire passer la détection de balise en 3D. Cependant on a mal géré le temps consacré à la 3D et il nous restait pas du tout de temps pour pouvoir faire la détection de balise. Nous avons donc décidé d'abandonné l'idée de la recherche de balise pour se contenter de nos stratégies avancées déjà plus ou moins abouti.

Conclusion

Dans l'ensemble, nous avons conçu un code simple à comprendre et à modifier qui affiche une simulation la plus réaliste possible des déplacements du robot et également un programme permettant au robot de réussir les tâches suivantes :

- Naviguer dans une arène contenant des obstacles.
- Tracer un carré.
- Tracer d'autre figures tel qu'un triangle ou un hexagone.

Nous avons pu apprendre de nombreuses choses à travers ces derniers mois tel que par exemple, la gestion d'un projet et les différentes phases du cycle d'un projet (conception, tests), l'importance de la régularité, le planning des tâches avec une plateforme comme Trello, la recherche de source etc.