Compte rendu final LU2IN013

Participants:

FISZBIN Vincent

ACHEK Ranya

MAJDOUB Bilel

TIGHIDET Yasmine

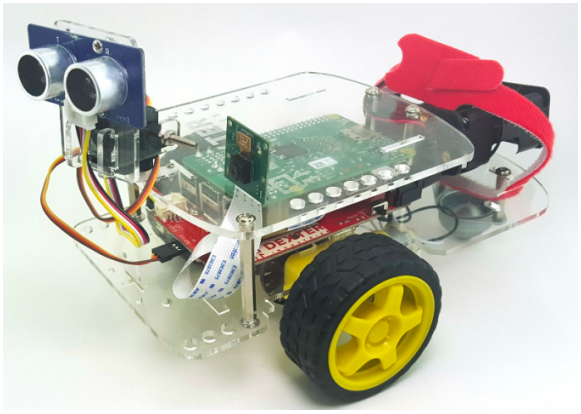
BOUCHAAL Samia

Sommaire

Introduction

Demandes du client

Dans le cadre de ce projet de robotique, nous devons rendre un code en Python, répondant à une problématique simple : permettre au robot Dexter d'effectuer différentes tâches plus ou moins complexes.



Robot Dexter composé d'un Raspberry Pi, d'une carte contrôleur Arduino, de deux moteurs encodeurs et de 3 senseurs

Le projet est composé de deux parties, une première partie comprenant 3 tâches : tracer un carré, s'approcher le plus vite possible et le plus près d'un mur sans le toucher et suivre une balise. La seconde partie est libre et nous devons proposer des tâches au client.

Objectifs

Afin de gérer notre projet efficacement, nous nous basons sur les principes de l'Agile et utilisons Trello et Git pour concevoir et répartir les tâches dans l'équipe. Le but initial, que l'on sépare en plusieurs micro-objectifs (Stories), est de créer à partir de Python, et de différents modules, un environnement de simulation suffisamment réaliste, mais le plus simple possible, avant de mettre en place le code pour contrôler le robot.

Ainsi, avant de coder, il fallait définir nos décisions initiales.

Décisions initiales

Au niveau de la modélisation du robot : nous avons vu en cours la géométrie nécéssaire à une modélisation optimale ainsi que la façon de simuler une arène (que nous appelons environnement) et des obstacles existant dans cet environnement.

En parallèle, il fallait rechercher quelle bibliothèque graphique utiliser pour représenter notre simulation. Nous avons rapidement décidé de ne pas nous lancer dans une simulation 3D pour ce début de projet et avons donc opter pour une simulation 2D avec la bibliothèque TKinter, qui a le mérite d'être simple d'utilisation, et présente dans la plupart des distributions de Python, y compris dans celle de l'installation Linux du robot.

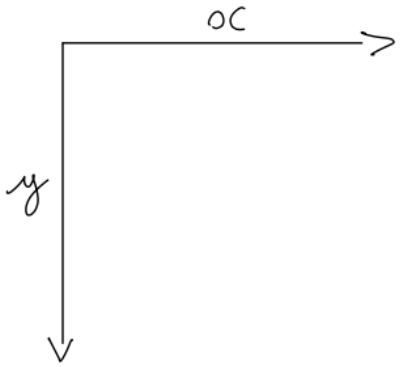
Création du robot et modélisation de l'environnement

On crée un fichier robot.py et un fichier environment.py pour simuler l'arène.

Classe Robot : robot.py

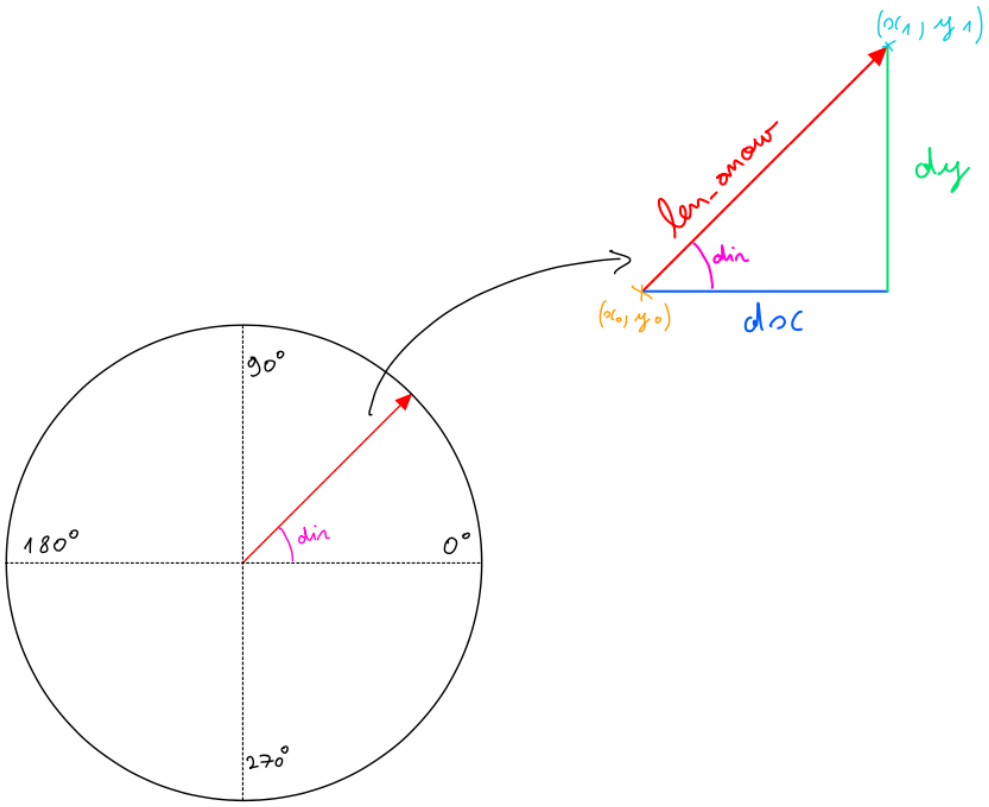
On simplifie au maximum le modèle du robot. On néglige donc au début la physique du robot et les roues qu'on rajoute plus tard. Dans notre simulation, on représente un robot par un cercle dont le centre est situé aux coordonnées (x,y), sa position initiale. Pour gérer les déplacement du robot, on introduit un attribut direction qui va servir de base pour les calculs de direction et les déplacements.

Dans certaines bibliothèques graphiques (comme TKinter), l'axe des abscisses x et des ordonnées y sont représentés ainsi :



Nous adoptons ici cette convention d'axe des abscisses/ordonnées.

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



Pour représenter graphiquement la flèche de direction du robot, nous devons calculer les coordonnées (x1,y1) de fin de flèche en ayant connaissance de la longueur de flèche (len_arrow) ainsi que l'angle dir. Pour cela, nous utilisons les règles trigonométrique ci-dessous :

$$dx = len\_arrow * cos(dir)$$

$$dy = len\_arrow * sin(dir)$$

$$x1 = x0 + dx$$

$$y1 = y0 - dy$$

Nous simulons les propriétés des roues avec les attributs speedLeftWheel, speedRightWheel, radius_of_wheels, half_dist_between_wheels et wheelMode ,

respectivement la vitesse de la roue gauche, celle de la roue droite, la taille des roues, l'espacement entre elles et le mode des roues du robot : un premier mode indique que les deux roues tournent dans la même direction et à la même vitesse, ce qui permet d'avancer ou de reculer. Dans le deuxième mode, les roues tournent toujours à la même vitesse mais dans des directions opposées, ce qui fait tourner le robot sur lui même. Avec le troisième, les roues ont la même direction mais des vitesses différentes, le robot décrit alors une trajectoire en arc de cercle.

Les coordonnées du robot sont stockées dans les attribus positionX et positionY, celles ci sont mises à jour par la fonction update du Robot (elle-même appelée continuellement par updateModele) qui calcule les nouvelles coordonnées du Robot en fonction de son mode de roue, de sa vitesse et de sa direction. La distance parcourue par le robot est calculée en fonction du temps écoulé entre chaque update qui est mesuré par l'attribut last_time.

A chaque update, le robot met aussi à jour les attributs angle_rotated_left_wheel et angle_rotated_right_wheel qui garde en mémoire l'angle dont chacune des roues a tourné depuis le dernier "reset". C'est le contrôleur qui utilise ces valeurs pour mesurer l'avancement du robot dans une stratégie et qui reset ces variables à la fin de chaque stratégie. C'est une façon de simuler l'offset de l'encodeur du robot réel.

Les fonctions changeDir, changeWheelMode, updateDir, changeSpeed, setSpeedRightWheel, setSpeedLeftWheel, deplacerRobot et deplacerEnArcRobot assurent la mise à jour de la position du robot dans la simulation.

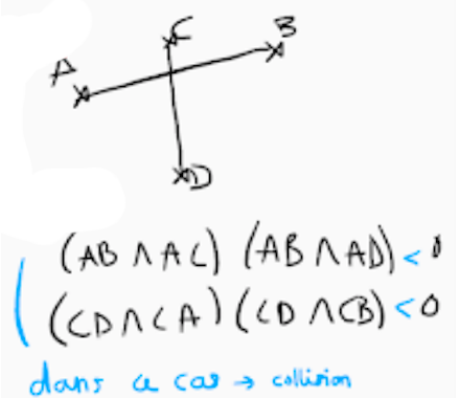
Classes Obstacle et Wall : obstacle.py

Avant de mettre en place notre arène et notre simulation, nous avons besoin de créer les obstacles que notre robot devra éviter. C'est pour ce faire que l'on crée dans obstacle.py les classes Obstacle et Wall qui hérite de Obstacle et qui sert de limites d'environnement au robot.

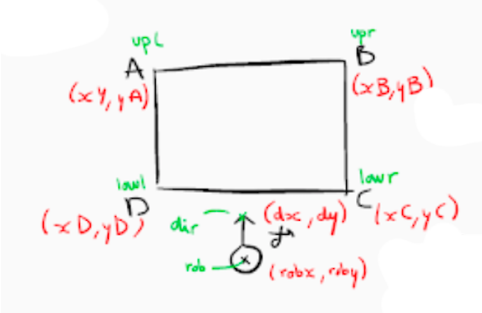
Les obstacles sont des rectangles caractérisés par la position (x,y) du point en haut à gauche.

Dans robot.py, on crée des fonctions qui permettent au robot de détecter la collision avec les obstacles.

Les calculs de detecteCollision sont réalisés en vérifiant la collision de deux vecteurs. Si ils sont de part et d'autre, l'un des produit vectoriel est positif, l'autre négatif.



La fonction se base sur cette représentation des attributs de l'obstacle et du robot.



On rajoutera plus tard un fichier lib.py contenant des fonctions réalisant des calculs mathématiques rappelant les méthodes de la classe Vecteur du cours.

Pour calculer la distance entre le robot et l'obstacle le plus proche faisant face au robot, nous avons codé une fonction getDistance qui réplique le fonctionnement du get_distance de l'API du robot réel. Pour ce faire, nous projetons des "pas" d'une certaine distance dans la direction du robot jusqu'à entrer en collision avec un obstacle de l'environnement, la fonction retourne alors le nombre de pas jusqu'au prochain obstacle. Nous utilisons principalement getDistance pour savoir si le robot est sur le point d'entrer en collision avec un obstacle. Par exemple, si le nombre de pas est inférieure à une certaine "safe distance" le robot est stoppé.

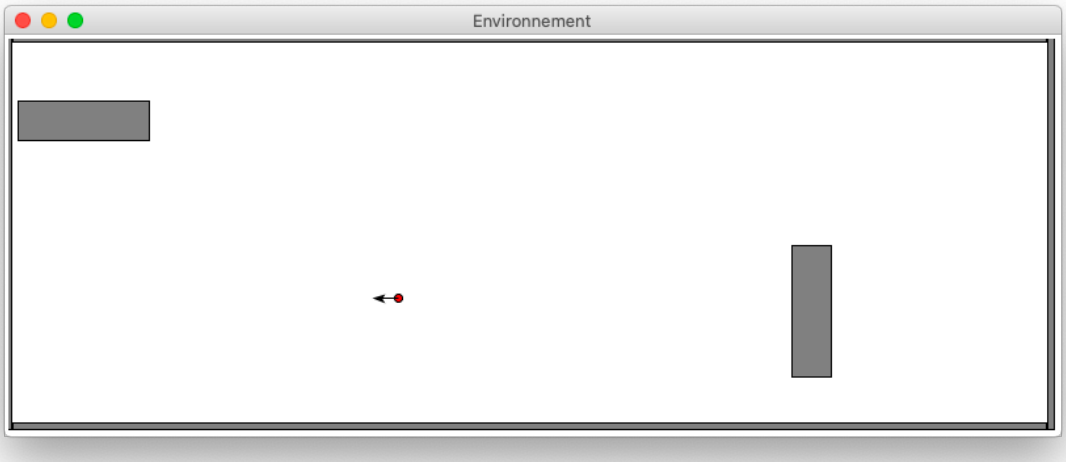
Classe Environment : environment.py

Le fichier environment.py importe les fichiers suivants:

```
from math import *
from .robot import Robot
from .obstacle import Obstacle
from ..lib import *
```

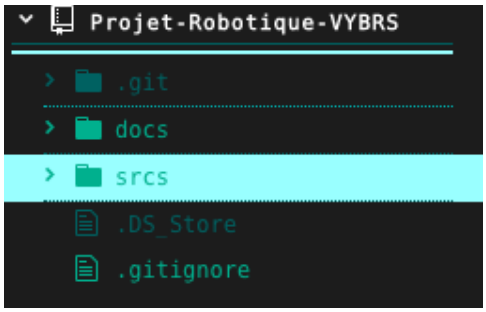
La classe environment contient les attributs height et width, définissant la taille de l'arène et une liste d'objet. Cette liste d'objet correspond à tous les éléments contenus dans l'arène, qui vont être au fur et à mesure du temps être ajoutés, supprimés et déplacés, c'est-à-dire le(s) robot(s), les obstacles et les murs de l'arène (classe Wall). De façon à créer un environnement plus ou moins réaliste, on fait attention aux conflits de position dès le placement initial des éléments. On ne peut pas, par exemple, positionner un robot dans un obstacle.

Une fois cette l'environnement créé, il peut être representé graphiquement avec notre classe View2D qui utilise la bibliothèque graphique Tkinter. View2D va continuellement itérer sur la liste d'objets de l'environnement pour les dessiner sur le canvas Tkinter.

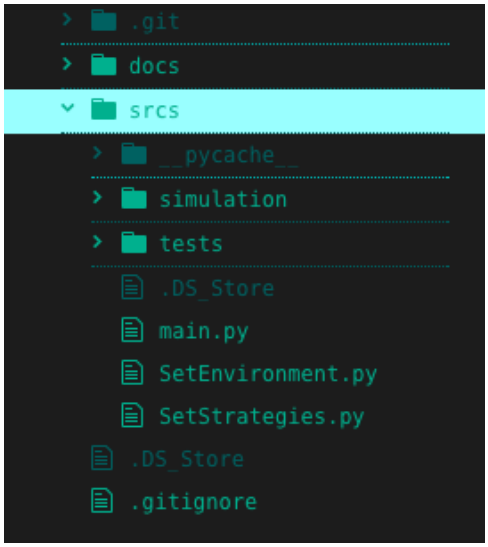


Organisation du code

Le code est contenu dans le dossier srcs de notre répertoire.



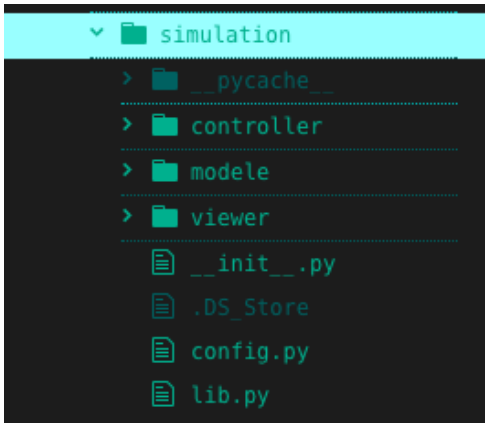
Les sous-répertoires de srcs sont les répertoires simulation et tests. Le répertoire contient aussi les fichiers main.py , SetEnvironment.py et SetStrategies.py .



Comme le suggère son nom, main.py permet de mettre en place l'environnement dans le cas de notre simulation à l'aide de SetEnvironment.py et de mettre en place les stratégies, dont on parle dans la partie suivante, pour la simulation et le robot matériel.

Fichiers et classes

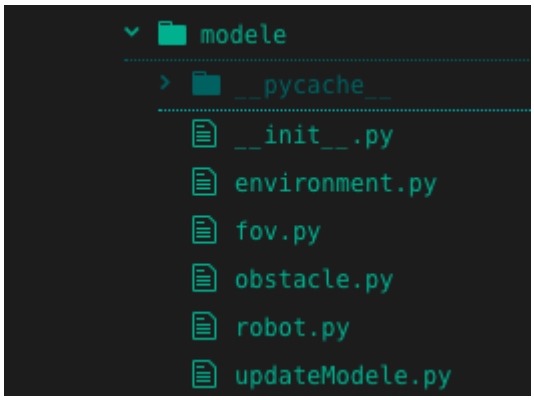
Simulation



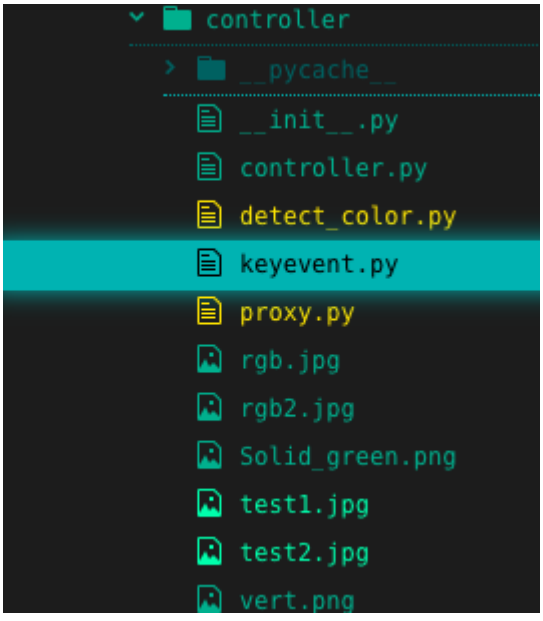
Le sous-répertoire simulation est constitués de 3 répertoires, controller, modele et viewer, ainsi que des fichiers config.py et lib.py, ce dernier mentionné plus haut.

Le fichier config.py permet de configurer les variables globales à tous les modules et de les initialiser (on a donc un état initial comme paramètres par défaut).

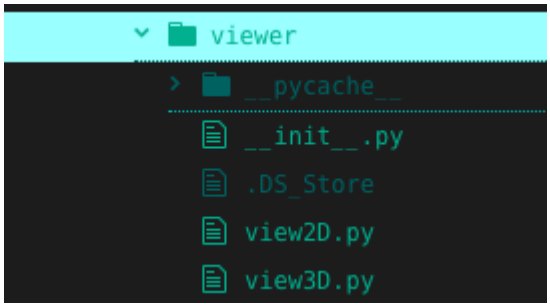
Le repertoire modele contient les fichiers dont on a parlé dans la partie précédente ainsi que les fichiers fov.py qui gère le champ de vision et updateModele.py qui permet la mise à jour continue de l'environnement dans un thread à part.



Dans controller, le fichier controller.py contient les instructions données au robot pour mener à bien l'exécution de diverses stratégies, tandis que le fichier et proxy.py contient les "traductions" faisant la passerelle entre les instructions que nous envoyons au robot réel ou simul xé. On parle plus en détails de ces fichier dans les parties suivantes. Dans le répertoire controller, on trouve aussi detect_color.py qui regroupe les fonctions permettant la détection de balise de couleur (rouge, verte ou bleue) dans une image.



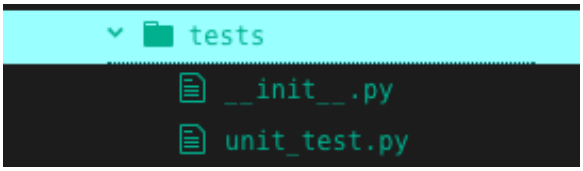
Dans viewer, enfin, on a les fichiers permettant les affichages graphiques.



Le fichier view2D.py gère l'affichage 2D à l'aide de l'interface graphique Tkinter et le fichier view3D.py gère la 3D à l'aide de l'interface graphique ursina.

Tests

Le sous-répertoire tests contient le fichier unit_test dans lequel on utilise le module de tests unitaires unittest.py pour tester chaque méthode de chaque classe individuellement.



On utilise le module unittest. On teste séparément nos sous-modules et ce fichier permet d'être plus rapide dans le debug de nos méthodes ainsi que d'automatiser un maximum les tests.

Fonctionnement du projet

L'interaction entre les différentes classes décrites plus haut, et les threads dans lesquels elles s'executent peut être résumée par le schéma ci-dessous :

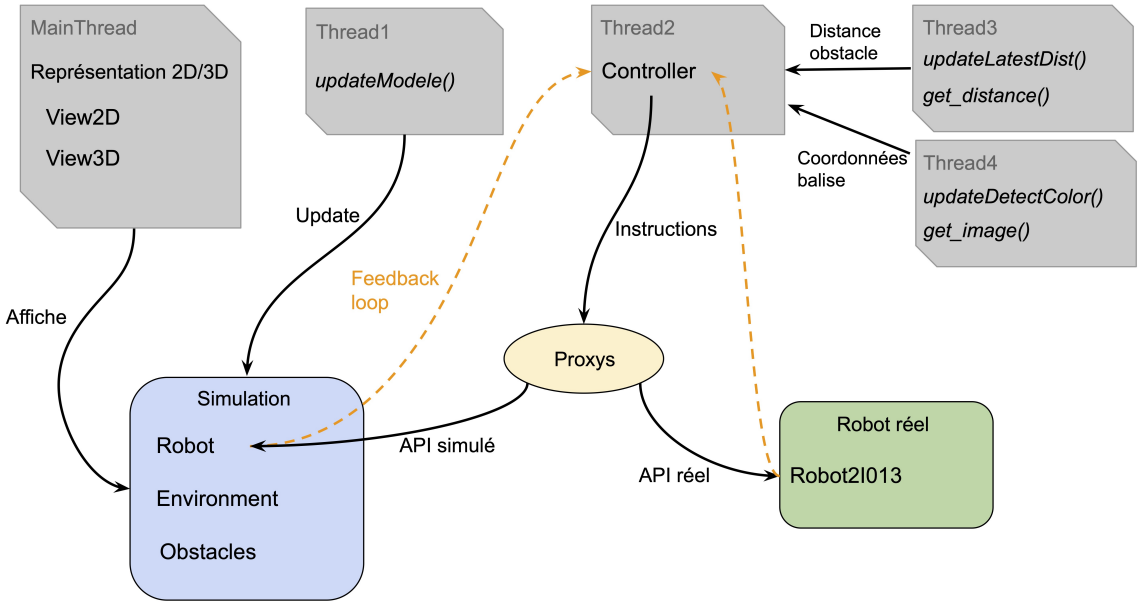


Schéma fonctionnement du projet

Branches

Sur le github du groupe, https://github.com/mKartofel/Projet-Robotique-VYBRS, on travaille sur la branche master tout en mettant à jour une branche demo sur laquelle on a fait notre présentation hebdomadaire.

Strategies

Les stratégies sont des '*behavioural patterns*' ou modèles comportementaux en français. Ceux-ci permettent de faire lancer nos algorithmes indépendamment du type des instructions données (mode réel/ simulation)

Ces stratégies correspondent à des classes dans le fichier controller.py du répertoire controller.

On distingue trois types de stratégies:

Les stratégies simple: ce sont des stratégies qui effectuent des instruction relativement simple comme avancer le robot(moveFowardStrategy), faire tourner le robot (TurnStrategy).

Les stratégies trigonométriques : elles héritent de la classe StrategySeq (classe qui organise une séquence de stratégies) afin de créer de instructions

permettant au robot de tracer des formes en faisant appel aux stratégies élémentaires comme tracer un carré(SquareStrategy) ou tracer un triangle équilatérale(TriangleEquiStrategy) .

Les méta-Stratégies : elles permettent de manipuler d'autre stratégies avec des conditions/ instructions un peu plus compliqué qui nous permet d'avoir un robot plus autonome comme:

La Stratégie Navigate :

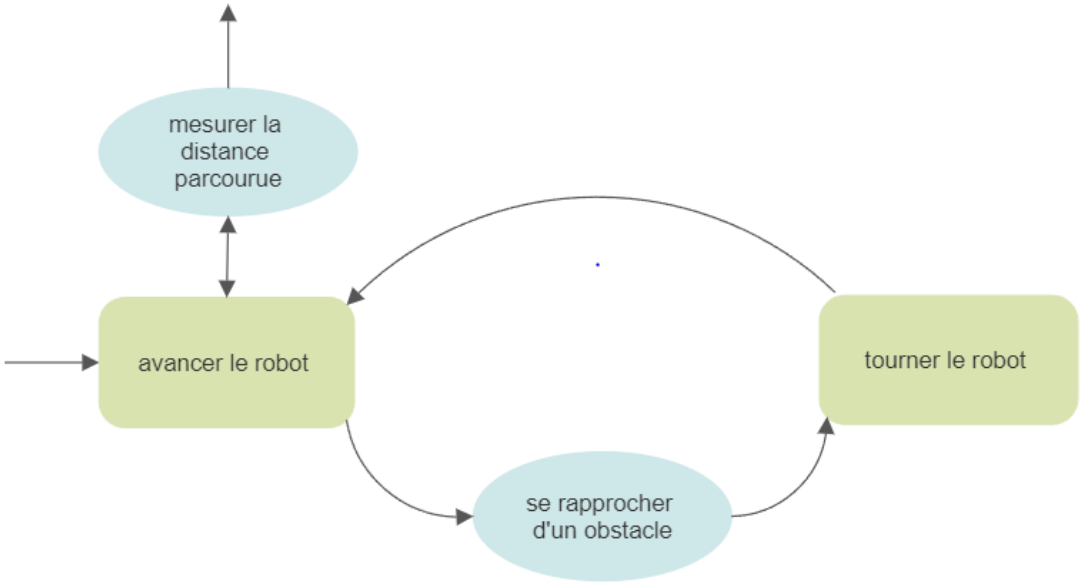


schéma du fonctionnement de la Stratégie Navigate

Cette méta-stratégie alterne l'exécution des stratégies '*moveForwardStrategy*' et '*TurnStrategy*'. Le robot avance tout droit jusqu'à détecter un obstacle, il tourne alors de 90 degrés pour éviter l'obstacle avant de reprendre sa course. La stratégie prend fin lorsque le robot a parcouru la distance passée en argument.

La Stratégie ArcStrategy :

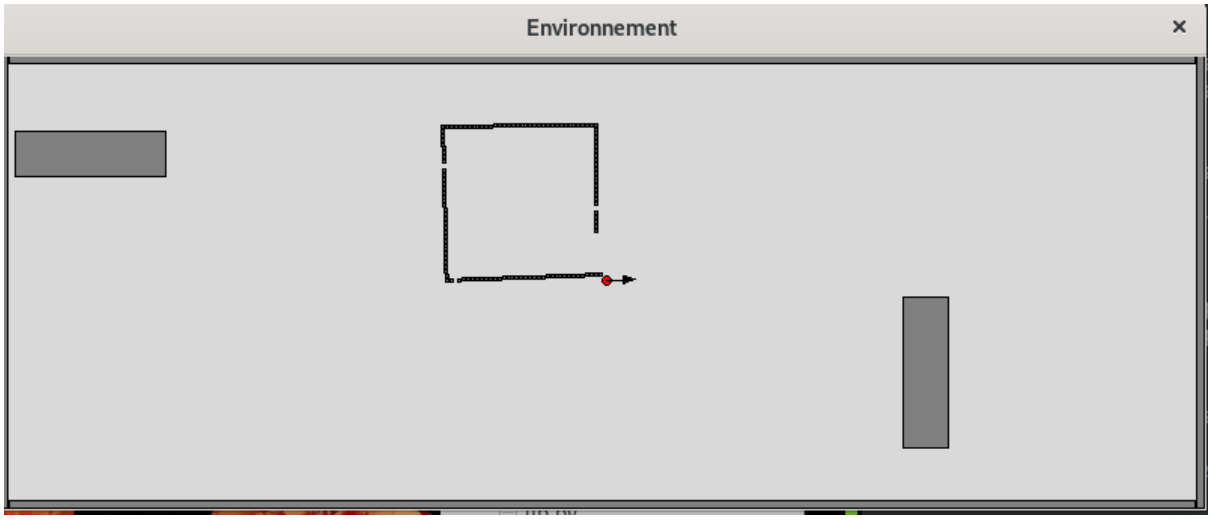
Permet au robot de tracer un arc de cercle d'un angle et d'un diamètre donnés en paramètre. L'argument to_left_or_right de cette stratégie permet au robot de décrire l'arc de cercle soit à sa droite, soit à sa gauche.

Les détails sur les calculs nécessaires à l'exécution de cette stratégie peuvent être trouvés dans le répertoire docs/ du projet.

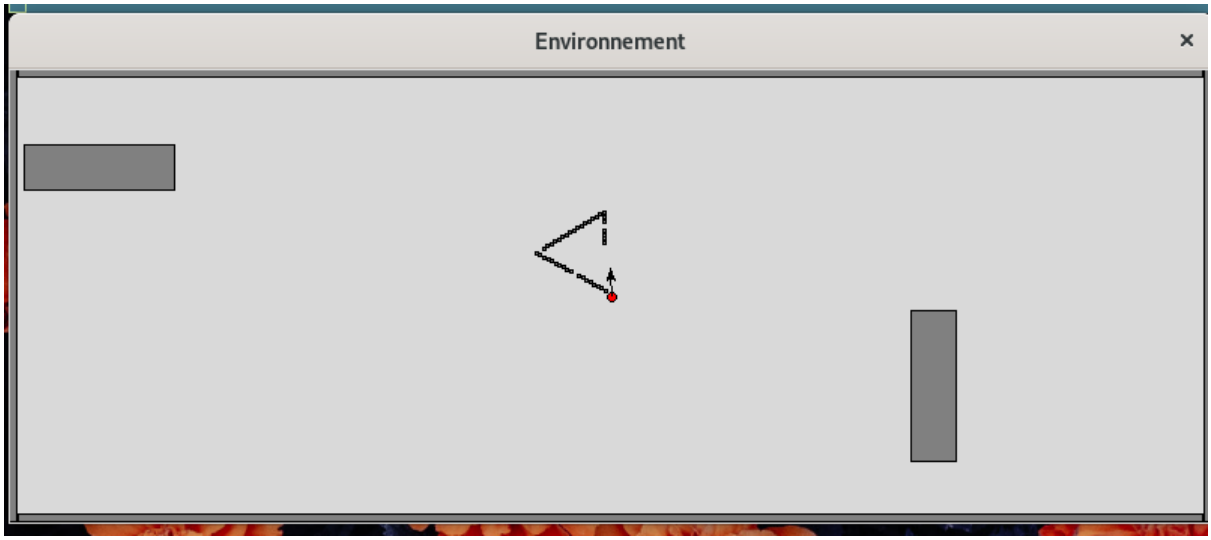
Résultats des Testes des Stratégies :

Pour tester les stratégie on a créer une classe Trace qui hérite d'obstacle et on crée des petits obstacles qu'on affiche dans updateSimulation dans view2D.py où on crée les Traces au cordonnées des robots.

Résultat de La Stratégie Carré *(SquareStrategy):*



Résultat de stratégie Triangle équilatérale(TriangleEquiStrategy):



Résultat de stratégie *Navigate*: on lance la stratégie avec une distance 1000

Résultat de stratégie Arc(*ArcStrategy)* : on lance la Stratégie pour un angle de 180° et un diamètre 50

Proxy et API du robot

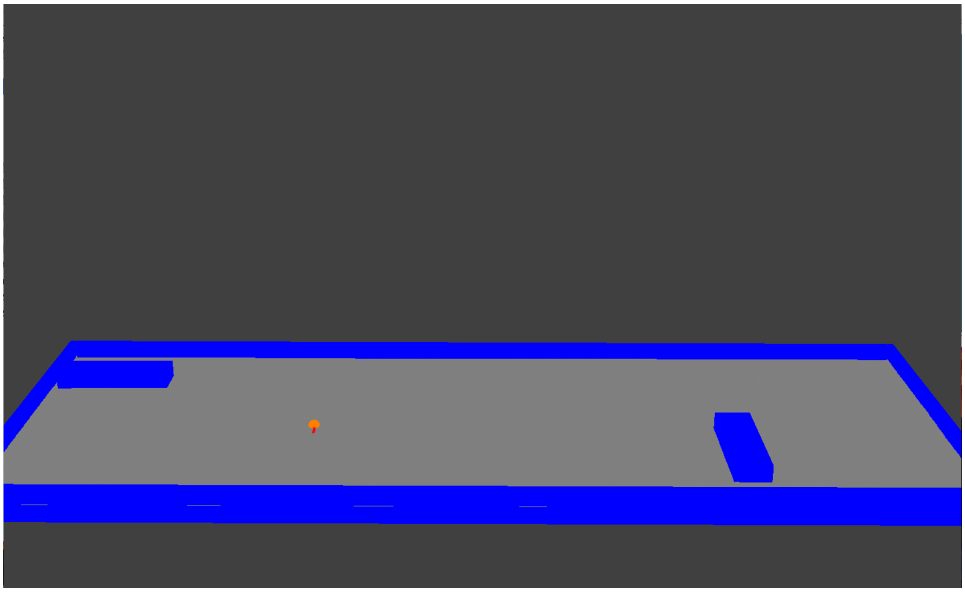
De manière à ce que l'on puisse communiquer les instructions de nos stratégies à notre robot simulé ainsi qu'à son équivalent matériel, on crée un '*adaptateur*' servant d'intermédiaire entre l'API du robot de la simulation et celui du réel.

Notre fichier proxy.py dans le répertoire controller nous sert d'adaptater. Les APIs de nos simulations ainsi que celui du robot diffèrent : la création de ce design pattern

sert à faciliter l’appel de nos fonctions pour chacune d’entre elles. Le principe est de créer deux classes ProxySimu et ProxyReal 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 classes un nom identique de façon à ce que l’appel d’une fonction marche avec chaque API.

Modélisation 3D

En plus de la représentation 2D avec Tkinter, l’état de la simulation peut être représenté en 3 dimensions via l’API graphique Ursina. Nous avons conçu les classes View2D et View3D comme des blocs à part entière et interchangeables capables de représenter en temps réel les objets de la simulation (robot et obstacles) et l’évolution de leur position.



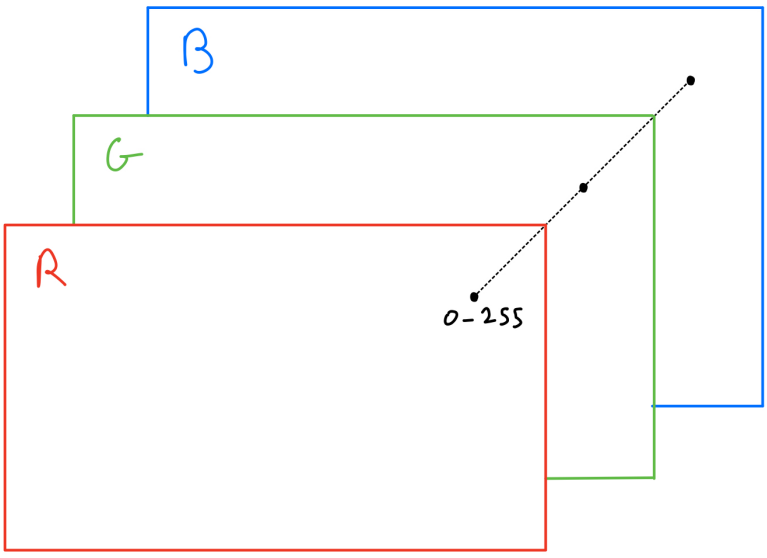
Vue de dessus de l’environnement représenté en 3D

Détection de balise

Pour que notre robot soit capable de détecter une balise dans son environnement, nous avons mis au point un système de localisation de rectangles rouges, bleus ou

vert sur une image.

Nous avons choisi de ne retenir que les trois couleurs primaires étant donné qu'une image, telle que le robot peut en capturer à l'aide de sa caméra, peut être représentée comme une matrice à trois dimensions. Les dimensions de ces matrices correspondent à la résolution de l'image, et chaque élément de matrice à un pixel. À chaque pixel sont associées trois valeurs, en raison des trois dimensions de la matrice, qui quantifient respectivement l'intensité de Rouge, Bleu et Vert (RGB) pour ce pixel, sur une échelle allant de 0 à 255.



Représentation d'une image comme matrice RGB a 3 dimensions

Notre premier prototype de détection de rectangle rouge, bleu ou vert parcourait l'ensemble d'une matrice et à chaque fois qu'un pixel de la couleur recherchée était trouvé, nous tentions de déterminer les coordonnées du plus grand rectangle de cette couleur à partir du pixel en question. Cette méthode, bien que fonctionnelle sur des images simples et de faible résolution, n'était pas utilisable en conditions réelles car trop lente.

Par la suite, nous avons raffiné ce prototype avec une nouvelle approche : la subdivision de l'image en sous-rectangles auxquelles on attribue la couleur rouge, verte ou bleue si une proportion suffisante de ses pixels sont de cette couleur.

Cette agglomération de groupes de pixels en sous-rectangles permet d'obtenir une représentation suffisamment fidèle de la carte des couleurs RGB présentes dans l'image si celle-ci est divisée en un nombre assez grand de sous-rectangles, tout en

offrant permettant une exécution considérablement plus rapide qu'avec la première approche.

Après quelques tests, nous avons abandonné le schéma RGB et avons opté pour une représentation HSV de l'image qui permet une détection plus simple et efficace des couleurs (qui peuvent être spécifiées en degrés avec la valeur Hue), le principe reste toutefois le même que celui décrit plus haut.

Prenons l'exemple de l'image ci-dessous pour illustrer le fonctionnement de notre système. On souhaite détecter le plus grand rectangle soit rouge, soit vert, soit bleu, apparaissant dans l'image. Dans cet exemple nous cherchons le plus grand rectangle vert.

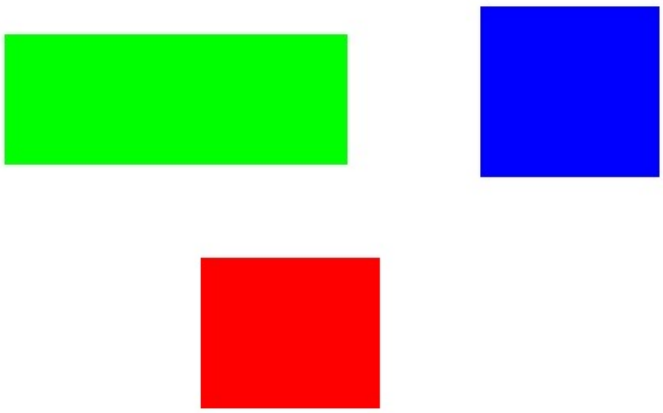


Image sur laquelle on souhaite détecter le plus grand rectangle rouge, vert ou bleu

On subdivise cette image selon un nombre donné de lignes (n_lines) et de colonnes(n_col), on obtient un quadrillage de l'image comme ci-dessous :

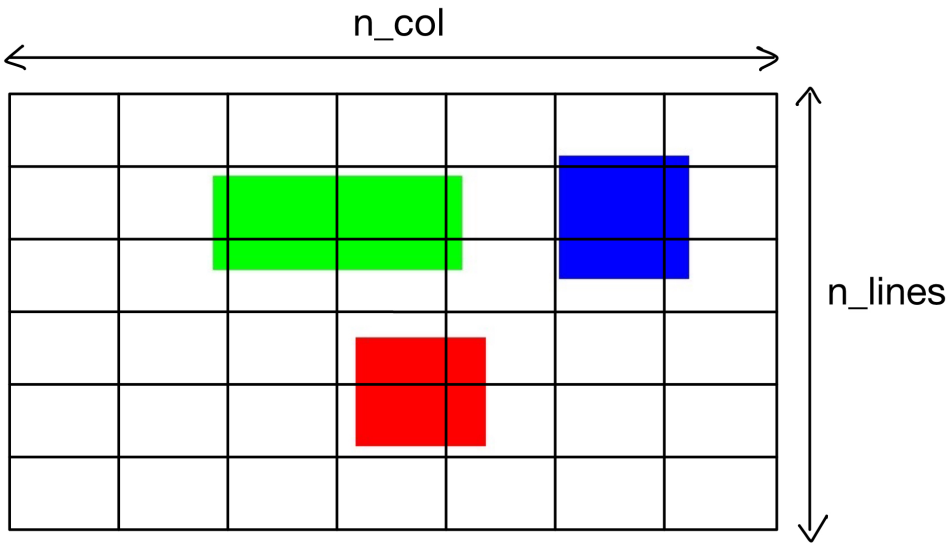
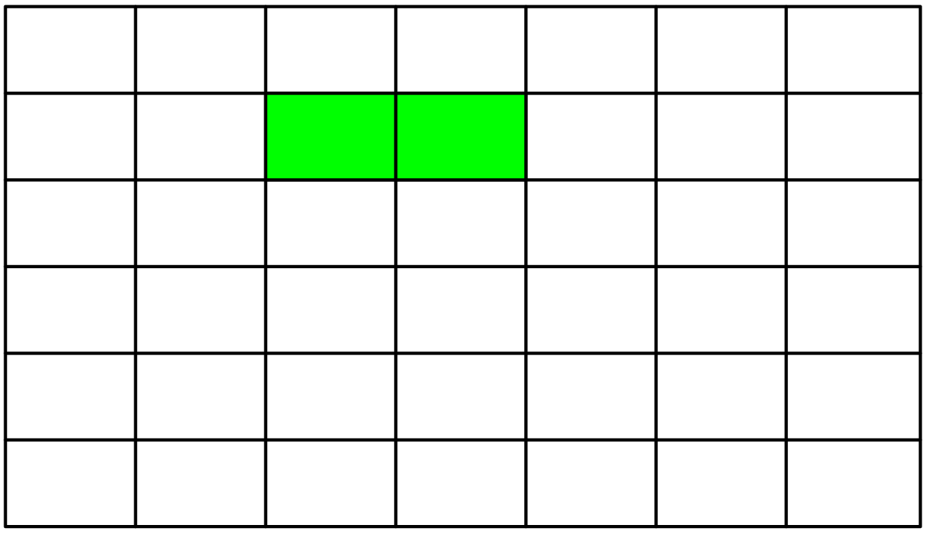


Image quadrillée en sous-rectangles

Ensuite, on utilise la fonction somme de la bibliothèque Numpy pour rapidement obtenir le nombre de pixels verts présents dans chaque sous rectangle. Si ce nombre représente plus de la moitié des pixels du sous rectangle (si le ratio fixé est de 0.5), alors on considère que ce rectangle est de couleur verte, dans notre exemple on obtiendrait donc la matrice de couleur suivante :



Rectangles à dominante verte présents dans l’image

Il ne reste alors qu’à retourner les coordonnées du sommet en haut à gauche et en bas à droite du plus grand rectangle composé de sous-rectangles verts.

Ces coordonnées sont ensuite utilisées dans la stratégie de recherche. Cette stratégie fait tourner le robot sur lui même tout en appelant continuellement la méthode getImage de l’API du robot et en analysant l’image avec le système de détection de balise. Une fois la balise détectée, le robot se dirige vers elle tant en avançant tout droit, jusqu’à rencontrer un obstacle. Si le robot perd de vue la balise, il se remet à tourner pour la retrouver.

Notre approche de détection d’une balise, bien qu’elle soit plus performante que le prototype initial, reste perfectible et ne produit pas systématiquement les résultats attendus. Une première amélioration serait de considérer la balise comme étant un ensemble prédéfini de couleurs (par exemple un rectangle composé de trois ou quatres couleurs), de cette façon, le nombre de faux positifs lors de la détection de balise serait bien moindre puisqu’on considérerait une balise comme détectée uniquement lorsque l’on détecte cet ensemble de couleur (agencé d’une façon précise) et non plus une seule.

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 ainsi qu’un programme permettant au robot d’effectuer les tâches suivantes :

- tracer un carré
- naviguer dans un environnement contenant des obstacles
- s’approcher d’un mur le plus vite possible
- suivre une balise à l’aide de la caméra du robot

Nous avons pu apprendre de nombreuses choses au cours de ces derniers mois comme, par exemple, la gestion d’un projet et les différentes phases du cycle (conception, tests), l’importance du travail régulier, le planning des tâches avec une interface type Trello, la recherche de sources etc.
