

APPRENTISSAGE ET GÉNÉRALISATION SUR UNE TÂCHE DE NAVIGATION D'UN ROBOT À ROUES

Encadrant : **DONCIEUX Stéphane**
KADHI Youssef, LIM Vincent et ZHENG Denis
Prandroide
UE MU4IN205

Table des matières

Table des figures	2
1 Remerciements	3
2 Contexte	3
3 Outils de communication	5
3.1 Discord	5
3.2 Trello	5
3.3 GitHub	5
4 Créations des environnements	6
4.1 Les scènes	6
4.1.1 Libfastfim	6
4.1.2 PyBullet	6
4.2 Le robot	7
4.2.1 Libfastsim	7
4.2.2 PyBullet	7
5 OpenAI Gym	8
5.1 Environnements	8
5.2 Actions	8
5.3 Observations	8
6 Homogénéisation des simulations	9
7 Les contrôleurs	10
7.1 Forward	10
7.2 Follow wall	10
7.3 Rule based	11
7.4 Braitenberg	11
7.5 Résultats	12
8 Apprentissage	15
8.1 Principe	15
8.2 Résultats	15
9 Conclusion	18

Table des figures

1	Evaluation de la performance, par rapport au nombre d'échantillons, d'un robot sur la reconnaissance d'objets physiques [1]	4
2	Scènes sous Libfastsim	6
3	Scènes sous PyBullet	6
4	Modèle de robot utilisé	7
5	Représentation des capteurs du robot	7
6	Boucle agent-environnement	8
7	Diagramme du contrôleur Forward	10
8	Représentation d'un véhicule multisensoriel de Braitenberg de type 3c	11
9	Comparaison des contrôleurs	12
10	Echantillonnage d'une trajectoire à l'aide de portes	13
11	Comparaison des trajectoires Braitenberg	13
12	Comparaison des trajectoires Rule based	14
13	Violin-plot : Mesure de la différence de trajectoires sur 3 politiques	16
14	Mesure d'écart de trajectoire	17
15	Comparaison selon la précision	18

1 Remerciements

Nous souhaitons tout d'abord adresser nos remerciements à M. Stéphane DONCIEUX pour avoir accepté notre candidature à ce sujet ainsi que pour nous avoir accompagné tout au long de ce travail.

2 Contexte

Au XXI^e siècle, les robots sont de plus en plus performants et occupent une place d'autant plus importante dans nos vies. En effet, on les retrouve dans les usines, dans les champs, sur les mers, dans nos maisons, et même dans l'espace.

Nous pouvons alors nous demander quel est leur atout majeur. Quelles sont les raisons pour leur utilisation de plus en plus répandue. Bien que nous, en tant qu'humains, sommes capables d'apprendre et à nous perfectionner sur des tâches et mouvements complexes, que ce soit par la pratique ou par l'anticipation, maîtriser complètement une tâche requiert énormément de pratique. C'est pour cela que nous apprenons de nos expériences et que nous sommes en constant apprentissage, tout au long de notre vie.

Entraîner un robot afin qu'il maîtrise une tâche pendant des années n'est évidemment pas efficace. Il faudrait qu'il soit exécuté en permanence et de manière autonome. Au début de l'apprentissage, le robot serait évidemment peu efficace sur la tâche à accomplir, et il s'améliorerait au fil des exécutions. Néanmoins, les robots ont à leur disposition un puissant outil : **les simulations**.

Avec l'évolution des puissances de calcul des ordinateurs et de la complexité du matériel informatique, il est possible, en un temps limité, de simuler plusieurs années d'apprentissage pour un robot. Malgré cet atout, l'inconvénient des environnements de simulation est que ces derniers, même les plus performants, ont souvent des difficultés à reproduire parfaitement le monde réel, que ce soit dû à des éléments à représenter ou à des propriétés physiques à appliquer. Ce problème est souvent accentué par le fait que des algorithmes de type essai et erreur, dont ceux de deep learning, peuvent exploiter ces imperfections pour "tricher" et accomplir la tâche demandée. Et de ce fait, lorsque le robot serait déployé dans le monde réel, ce dernier n'aura pas le comportement attendu, observé dans les simulations.

Ce problème de transfert d'algorithmes basés sur de l'apprentissage, d'un environnement de simulation au monde réel, est appelé **reality gap** [6]. C'est un phénomène qui, dû à des légères, mais importantes, imperfections dans une simulation, empêche le déploiement d'un robot immédiatement après des simulations.

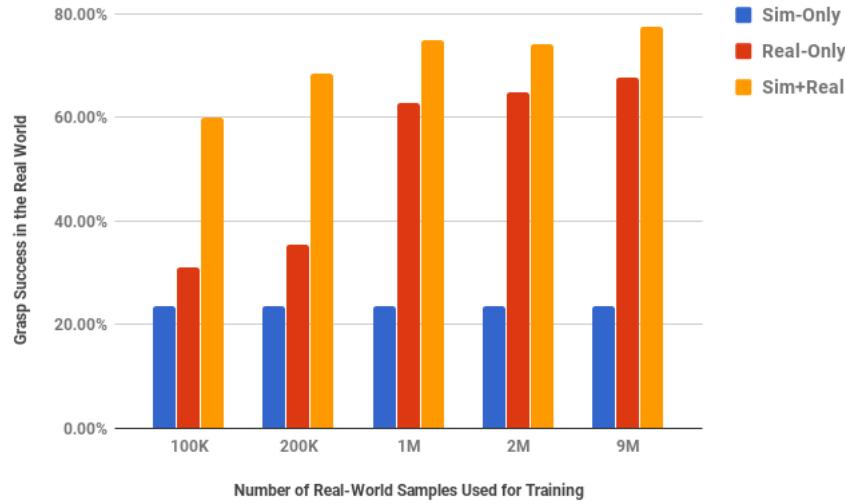


FIGURE 1 – Evaluation de la performance, par rapport au nombre d'échantillons, d'un robot sur la reconnaissance d'objets physiques [1]

Konstantinos Bousmalis et Sergey Levine, tout deux membres du projet Google Brain, ont effectué des travaux sur le reality gap (fig. 1). Ce dernier consistait à permettre à un robot, dans un environnement PyBullet, à reconnaître des objets physiques quelconques à l'aide d'algorithmes d'apprentissage machine. On observe qu'importe le nombre d'échantillons utilisés, les performances sur un apprentissage sans simulations restent constantes (dû à des contraintes de temps) ; alors qu'avec de la simulation, les performances augmentent avec le nombre d'échantillons. Enfin, nous observons qu'en travaillant dans un environnement de simulation et en appliquant des méthodes d'adaptation de domaine traitant du passage de simulation à la réalité (combinaison simulation et réalité), les résultats obtenus sont bien meilleurs.

L'enjeu du projet est donc de pouvoir mesurer la capacité de différents algorithmes à franchir le Reality Gap en définissant un environnement capable de mesurer et affecter une valeur numérique à ce phénomène. Pour cela, nous travaillerons avec deux environnements de simulations : **Libfastsim**[7] et **PyBullet**[4]. Libfastsim est un environnement de simulation, développé par Jean-Baptistes Mouret, léger et rapide, permettant l'apprentissage de politiques pour le déplacement de robot à roues.

PyBullet est un module Python, utilisé pour de la simulation en robotique et de l'apprentissage machine. C'est un environnement de simulation plus réaliste qui prend en compte divers paramètres tels que la gravité et les frottements.

Dans un premier temps, nous allons modéliser les éléments (robot et scènes) et paramétrier les deux environnements afin qu'ils soient le plus similaires possible. Nous allons ensuite implémenter plusieurs politiques de déplacement pour un robot à roues dont nous mesurerons la transférabilité entre Libfastsim et PyBullet. Enfin, nous effectuerons le même procédé pour un algorithme d'apprentissage machine que nous implémenterons.

3 Outils de communication

Afin de pouvoir travailler efficacement en distanciel, nous avons utilisé divers outils de communications afin de bien pouvoir s'organiser et se répartir les tâches.

3.1 Discord

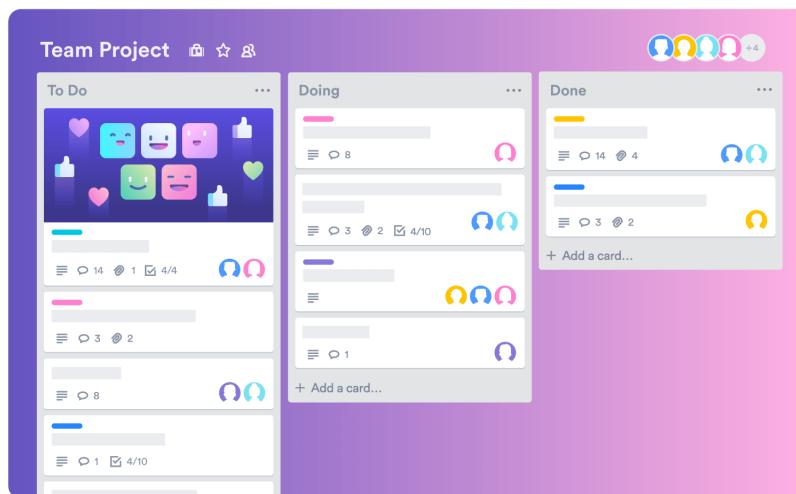


DISCORD

Discord est une plate-forme de communication collaborative qui facilite les échanges au sein d'une équipe. La possibilité de créer différents canaux permet d'organiser les conversations en fonction des sujets abordés sur le projet. Il est alors plus facile d'échanger à distance. Discord permet d'avoir un meilleur flux d'informations

3.2 Trello

Trello est un service permettant de lister des tâches. Très utile pour un projet, il est alors plus facile d'organiser des tâches et sous-tâches et de les répartir entre chaque membre de l'équipe tout en gardant un oeil sur l'avancement du projet.



3.3 GitHub



GitHub

GitHub est un service de gestion de dépôts git qui permet de centraliser des projets, faciliter la collaboration entre les développeurs sur un même code source tout en permettant d'avoir une vision claire sur l'avancement des projets.

4 Créations des environnements

4.1 Les scènes

4.1.1 Libfastfim

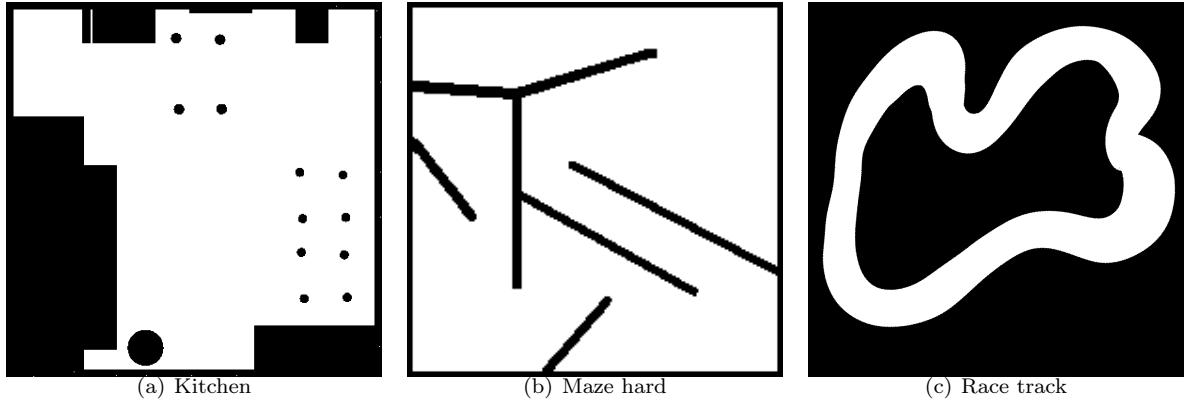


FIGURE 2 – Scènes sous Libfastsim

Pour nos mondes nous avons récupéré les fichiers `kitchen.pbm` et `mazehard.pbm` des dépôts de libfastsim. Ces scènes étant ouvertes, elles ne sont pas appropriées pour tester des comportements dans lequel le robot doit suivre les murs qu'il détecte. De ce fait, nous avons dessiné la scène `race_track` que nous avons ensuite converti en fichier `pbm`. L'autre avantage de cette carte est l'unique chemin possible qui va nous permettre de faire des comparaisons très précises.

4.1.2 PyBullet

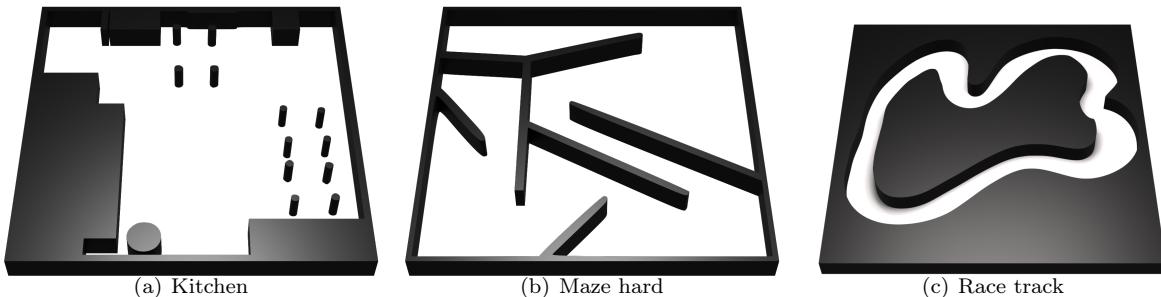


FIGURE 3 – Scènes sous PyBullet

Bullet étant un moteur physique en 3D il nous a fallu convertir les images de la figure 2 en objet 3D. Une modélisation à la main aurait été chronophage et imprécise ce qui aurait pu fausser les résultats. Nous avons donc utilisé le logiciel Blender, qui à partir d'une image, permet de modéliser l'objet en 3D. Blender permet notamment l'écriture de script python pour automatiser le processus. Ainsi, à l'exécution, celui-ci récupère tous les fichiers `.pbm`, les transforme en fichier `.svg` (`.pbm` n'étant pas accepté par Blender) puis les transforme en mesh qui l'extrude en un objet de taille $1m \times 1m \times 0.5m$ (ces valeurs sont arbitraires et modifiables). Enfin, afin d'importer ces objets dans l'environnement PyBullet, nous créons un fichier `.urdf` contenant l'objet 3D ainsi que des propriétés physiques obligatoires (simple ici s'agissant de murs).

4.2 Le robot

4.2.1 Libfastsim

Dans ce projet le robot qui va naviguer est un simple robot circulaire à roues. Dans l'environnement Libfastsim, nous pouvons choisir son rayon, sa position et son orientation (x, y, θ), ainsi que les capteurs dont il sera équipé. Nous avons à notre disposition des lasers, des capteurs de lumière, des bumpers, ainsi qu'un radar, mais dans le cadre de ce projet nous n'utilisons que les lasers, dont il est possible de spécifier la portée ainsi que l'angle.

4.2.2 PyBullet



FIGURE 4 – Modèle de robot utilisé

Pour PyBullet, nous avons décidé de partir sur un modèle de robot à roues existant, le iRobot create, qui est basé sur les roombas et destiné à la robotique. Pour ce faire nous avons trouvé un modèle 3D dans le simulateur de robot Gazebo que nous avons légèrement modifié sur Blender.

De même que pour les scènes, le robot a un fichier iRobot.urdf auquel nous avons ajouté les propriétés physiques, pour être le plus proche de la réalité, telles que le poids de chaque pièce, leur inertie et les jointures.

Pour avoir la même fonctionnalité de laser que sous Libfastsim, nous avons ajouté au centre du robot le modèle d'un hokuyo laserrange, un capteur souvent utilisé en robotique. Puis grâce à PyBullet, nous effectuons une multitude de "raycast" pour trouver les informations d'intersection du premier objet touché par chaque rayon, ce qui nous permet d'évaluer la distance nous séparant aux objets détectés, ici, les murs.

Pour avoir une certaine flexibilité sur les paramètres du robot sans avoir à rentrer dans le code nous avons ajouté un fichier de configuration permettant de changer, entre autres, le modèle 3D, la vitesse du robot, les caractéristiques des lasers.

Pour nos expériences nous avons choisi d'utiliser 10 lasers d'une portée de 1 mètre et séparé entre eux de 20° .

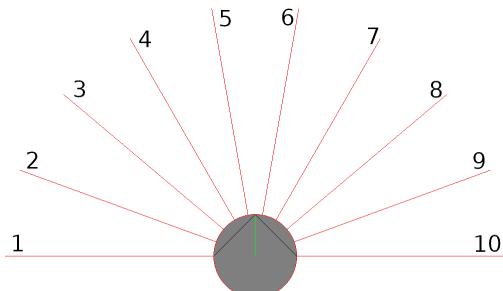


FIGURE 5 – Représentation des capteurs du robot

5 OpenAI Gym

OpenAI Gym [2] est un framework permettant une standardisation du développement et la comparaison d'algorithmes d'apprentissage par renforcement. Pour le simulateur Libfastsim, une interface avec Gym que nous avons légèrement modifiée, a été créée par [Alex Coninx](#). Nous avons essayé de nous rapprocher de son travail pour notre interface sous PyBullet.

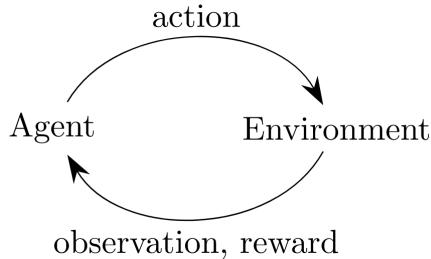


FIGURE 6 – Boucle agent-environnement

5.1 Environnements

À l'installation, un environnement gym est créé pour chaque fichier de configurations dans le dossier scénario. Dans notre cas cela consiste à avoir le robot ainsi qu'un monde : kitchen, maze hard, race track et un monde vide pour des tests.

5.2 Actions

À chaque pas de notre simulation s'effectue une action. Ici notre action prend la forme $[f_1, f_2]$, où f_1 et f_2 sont des flottants représentant la vitesse des roues gauche et droite du robot. Ces valeurs doivent être en -1 et 1 et seront limitées au-delà. Ainsi l'action $[1, 1]$ fera avancer le robot tout droit, $[-1, 1]$ le fera tourner, ainsi de suite. Sous PyBullet, nous récupérons le joint correspondant aux roues de notre robot et nous appliquons à ce dernier une vitesse proportionnelle, ce qui va faire bouger le robot ($[1, 1]$ permet d'avoir la vitesse max de 0.5 m/s)

5.3 Observations

Comme nous pouvons le voir sur la figure, chaque action effectuée entraîne une réponse de l'environnement. On note 4 informations distinctes retournées :

- **observation** : une liste contenant la valeur retournée par chaque laser. Ces informations sont récupérées et utilisées par les contrôleurs.
- **reward** : le montant de la récompense obtenue par l'action précédente. L'utilisateur peut choisir différents types de fonctions pour la récompense, les 3 premières ont été reprises de fastsim gym :
 - *reward binary goal based* : une récompense de 1 est accordée lorsque l'on s'approche suffisamment de l'objectif.
 - *reward displacement* : la récompense correspond à la distance par rapport à la position précédente.
 - *no reward* : pas de récompense.
 - *reward rapprochement goal* : la récompense correspond à la distance par rapport à l'objectif.

- **done** : un booléen indiquant si la simulation est terminée. L'utilisateur peut paramétrer la terminaison de deux manières, la 1^{er} est de choisir un rayon d'activation du goal (le robot doit-il juste toucher le goal ? Se trouver au centre ? etc...). La 2^e méthode consiste à imposer une limite de temps.
- **info** : un dictionnaire contenant les informations sur la simulation et le véhicule : temps de la simulation, vitesse, accélération, position et distance à l'objectif du robot.

6 Homogénéisation des simulations

PyBullet étant le simulateur le plus complet des deux et servant comme représentation de la vie réelle, nous avons essayé de nous calquer le plus possible sur les caractéristiques du vrai robot. Ainsi nous avons une gravité de 9.81 N/kg, les dimensions, le poids ainsi que l'inertie de chaque pièce ont été récupérés des caractéristiques d'un vrai roomba, une vitesse maximum fixée à 0.5m/s, une portée de laser de 1m, un scan toutes les 100ms, et une précision de 99%.

Une des caractéristiques de PyBullet est le timeStep, le temps entre chaque action de la simulation, que nous avons fixé à $\frac{1}{60}$ s. Ainsi, pour que le robot avance de 1m, il faudra 2s, soit 120 timesteps. Tandis que sur fastsim, le robot, à la vitesse maximum, parcourt 1m à chaque step. En plus d'être différent de PyBullet, avoir une telle vitesse crée des situations où lorsque le robot se trouve à une distance d'un mètre d'un mur à un step donné, il peut se trouver collé à celui-ci au step suivant étant donné qu'il se déplace d'un mètre par step.

Afin d'avoir des vitesses similaires dans les deux simulations, nous avons fait parcourir au robot 100 mètres sur PyBullet et nous avons compté le nombre de pas nbstep. Ainsi, en fixant la vitesse du robot sur fastsim à $\frac{100}{nbstep}$, nous avons une vitesse et un nombre de pas similaire sur les deux simulateurs, ce qui nous permet alors de les comparer efficacement.

7 Les contrôleurs

Afin de paramétriser les déplacements du robot, nous avons implémenté divers contrôleurs respectant l'architecture de subsomption proposée par Rodney Brooks dans les années 80 [3]. Une architecture de subsomption est un algorithme réactif dans lequel la prise de décision par le robot est dirigée par les informations récupérées par ses senseurs. Les comportements possibles sont implémentés par des couches distinctes et hiérarchisées, les couches supérieures pouvant subsumer celles qui lui sont inférieures. Ces contrôleurs sont paramétrables et permettent d'avoir des actions répétables et comparables entre les deux environnements de simulation. Nous allons vous présenter les quatre contrôleurs implémentés dans le cadre de ce projet.

7.1 Forward

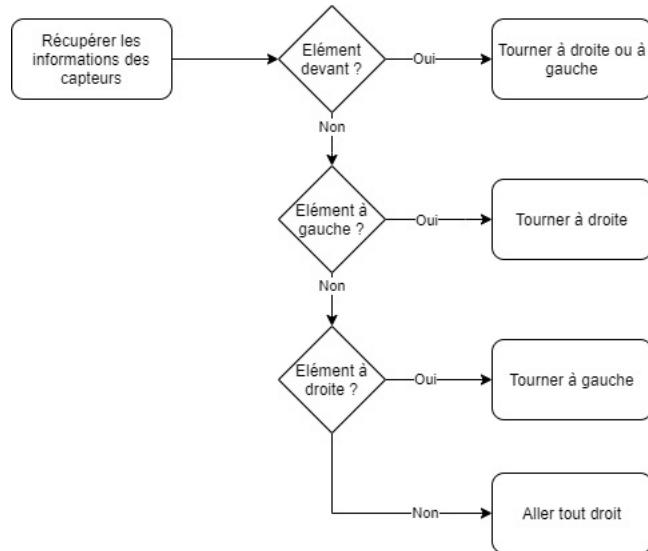


FIGURE 7 – Diagramme du contrôleur Forward

Le premier contrôleur que nous avons implémenté est Forward. C'est un contrôleur basique qui a pour objectif de faire avancer le robot tout droit, ou de le faire tourner lorsqu'il rencontre des obstacles. Le principe est le suivant :

- Interroger en priorité les capteurs avant, si un élément est détecté, tourner à droite ou à gauche avec des probabilités 1/2.
- Interroger les capteurs de gauche, si un élément est détecté, tourner à droite.
- Interroger les capteurs de droite, si un élément est détecté, tourner à gauche.

7.2 Follow wall

Le contrôleur Follow wall a une architecture similaire au premier, on interroge également les capteurs dans les trois directions dans le même ordre et les comportements adoptés sont également proches. Afin de permettre au robot de suivre un mur, deux constantes numériques **distTooClose** et **distTooFar** ont été ajoutées avec $distTooClose \leq distTooFar$. Le principe pour suivre un mur sur la gauche, par exemple, est

le suivant :

Algorithm 1 : Follow wall

```

Entrées : vitesse du robot.
Entrées : distMur, distTooClose, distTooFar.
1 si distTooFar > distMur > distTooClose alors
2   | retourner [vitesse, vitesse] ; // aller tout droit
3 si distMur < distTooClose alors
4   | retourner [vitesse, -vitesse] ; // tourner à droite
5 si distMur > distTooFar alors
6   | retourner [-vitesse, vitesse] ; // tourner à gauche

```

Les deux contrôleurs suivants ont été récupérés de l'article suivant [8].

7.3 Rule based

Le premier contrôleur est rule-based, il consiste à récupérer et sommer, pour chaque côté du robot, les valeurs des lasers, et si cette somme dépasse un seuil, le robot tourne de l'autre coté, la vitesse du robot est fixe. Ce contrôleur est très similaire à notre contrôleur Forward, la différence étant qu'ici, on vérifie si la somme des capteurs atteint un seuil, alors qu'avec Forward, on vérifie un à un si le capteur a atteint le seuil. On remarque qu'avec ce contrôleur le robot ne peut qu'avancer.

Algorithm 2 : Rule based

```

Entrées : vitesse du robot.
Entrées : seuil à partir duquel on tourne.
1 si  $\sum_{1}^{n/2} Sensor_i > seuil$  alors
2   | retourner [vitesse, -vitesse] ; // tourner à droite
3 si  $\sum_{n/2}^n Sensor_i > seuil$  alors
4   | retourner [-vitesse, vitesse] ; // tourner à gauche
5 retourner [vitesse, vitesse] ; // aller tout droit

```

7.4 Braitenberg

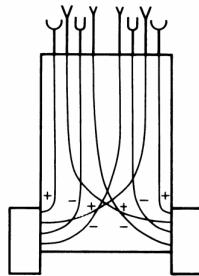


FIGURE 8 – Représentation d'un véhicule multisensoriel de Braitenberg de type 3c

Le deuxième contrôleur suit le concept de **véhicule de Braitenberg** émis par le chercheur en cybernétique autrichien-italien Valentino Braitenberg. Un véhicule de Braitenberg est un agent qui peut se déplacer de manière autonome en se basant sur les retours de ses capteurs. En effet, les roues du robot étant connectées aux lasers, le robot modifie automatiquement sa vitesse en fonction des obstacles. Dans notre cas le contrôleur correspond au véhicule de Braitenberg multisensoriel de type **3c** (chaque capteur affecte chaque

roue).

Algorithme 3 : Braitenberg

```

Entrées : vitesse
Entrées : reactivite
1  $Sl = \sum_0^{n/2} Sensor_i$ 
2  $Sr = \sum_{n/2}^n Sensor_i$ 
3 gauche = vitesse  $\times (1 + reactivite \times (Sr - Sl))$ 
4 droit = vitesse  $\times (1 + reactivite \times (Sl - Sr))$ 
5 retourner [gauche, droit]

```

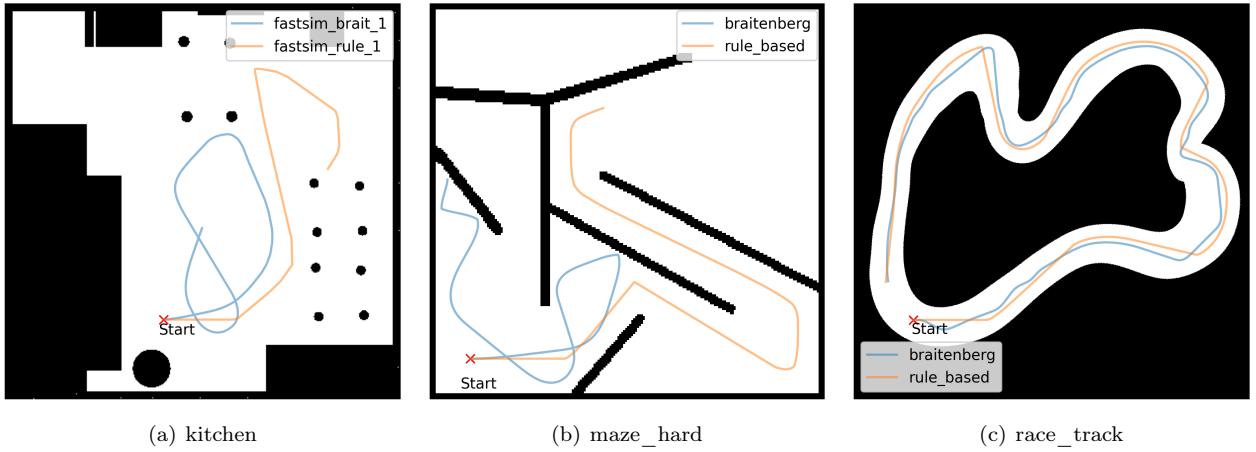


FIGURE 9 – Comparaison des contrôleurs

En observant la figure, nous pouvons remarquer les différences de comportement entre les contrôleurs :

- Rule based a une vitesse constante et ne peut qu'avancer, il fait donc du tout droit jusqu'à que les laser atteignent un seuil ce qui fonctionne assez pour suivre un mur ce qui est adapté pour race track et maze hard.
- Pour le contrôleur de type Braitenberg, nous avons une vitesse qui varie constamment et s'ajuste. Nous avons donc un contrôleur plus souple, ce qui peut être plus intéressant pour kitchen, par exemple, du fait que cette scène a des obstacles qui ne sont pas des murs. Cependant sur maze hard il est moins intéressant à ne pas suivre les murs.

7.5 Résultats

Dans le cadre de nos comparaisons, nous avons travaillé avec le contrôleur Rule-based et le contrôleur de type véhicule 3c de Braitenberg. Afin de pouvoir comparer des trajectoires, ces dernières doivent avoir des taux d'échantillonnages permettant d'avoir autant de points dans les deux simulations. De ce fait, nous avions initialement voulu utiliser les portes d'échantillonnages, présentées dans l'article [9].

L'idée est de placer des portes orthogonales aux murs tout le long du chemin. Les croix, indiquant les intersections entre la trajectoire d'un robot et les portes, correspondent aux points utilisés lors de l'analyse. Cette approche est adaptée à la scène race_track car le robot suit forcément le seul chemin existant. Cependant, cette approche est moins appropriée pour les autres scènes kitchen et maze_hard, qui sont plus ouvertes. En effet, le robot peut aller dans n'importe quelle direction, de ce fait, le placement des portes n'est pas évident.

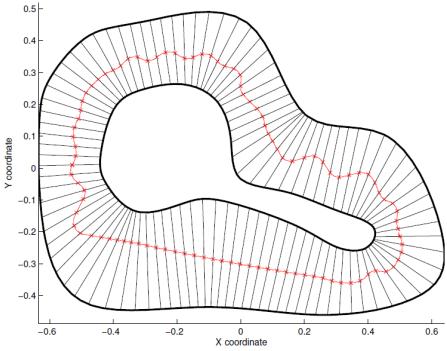


FIGURE 10 – Echantillonage d'une trajectoire à l'aide de portes

Nous avons donc abandonné cette approche, et comme indiqué dans la partie 5, nous avons effectué des tests pour avoir des vitesses similaires dans les deux environnements. Afin d'enregistrer les trajectoires, la position du robot à chaque instant est enregistrée dans un fichier .csv. De ce fait, sa trajectoire est représentée par l'ensemble des coordonnées présentes dans le fichier .csv. Et afin de comparer la différence entre deux trajectoires, nous les superposons dans un premier temps sur un plot afin d'observer les éventuels décalages. Une observation visuelle n'est parfois pas possible lorsque les trajectoires se croisent régulièrement. Nous avons donc aussi tracé la distance à l'objectif selon le pas et nous calculons une valeur représentative de l'écart de trajectoire entre les deux simulations à l'aide de la méthode des moindres carrés.

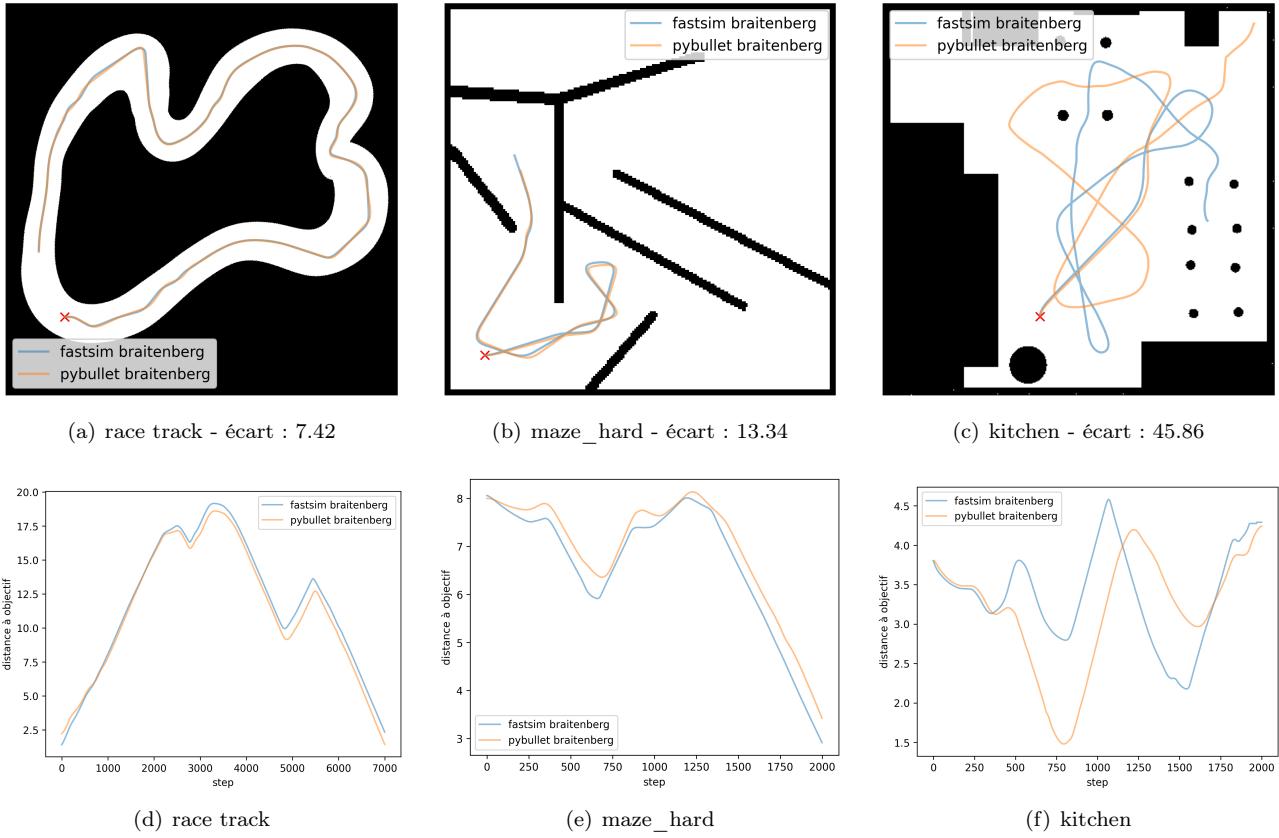
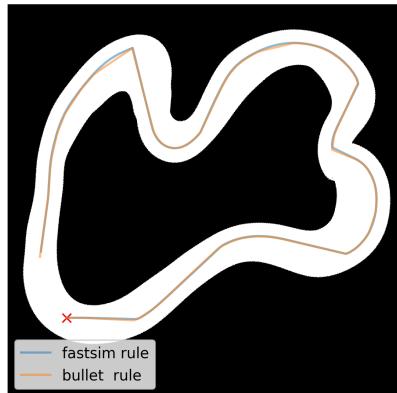
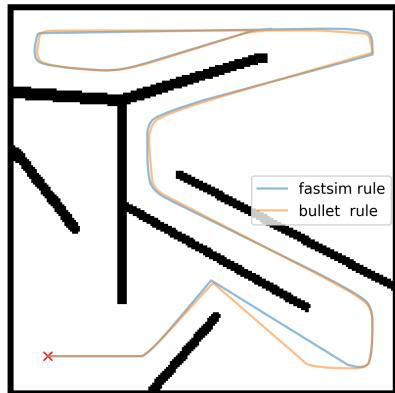


FIGURE 11 – Comparaison des trajectoires Braitenberg

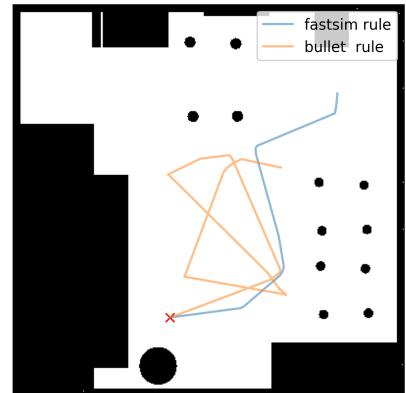
Nous pouvons voir que le contrôleur de type 3c de Braitenberg se transfert extrêmement bien d'un simulateur à l'autre pour la carte race track et maze hard. Pour kitchen la trajectoire initiale est similaire, mais à partir du moment où il rencontre un certain obstacle les trajectoires se séparent et ne se ressemblent plus du tout.



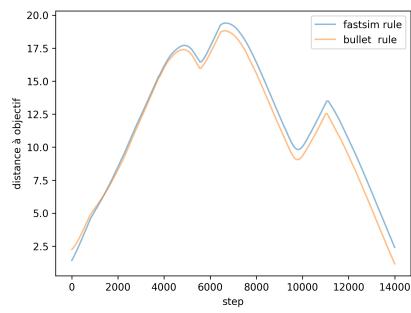
(a) race track - écart : 6.28



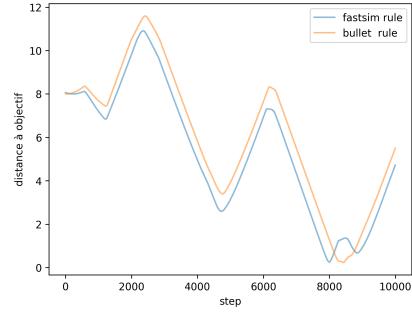
(b) maze_hard - écart : 15.81



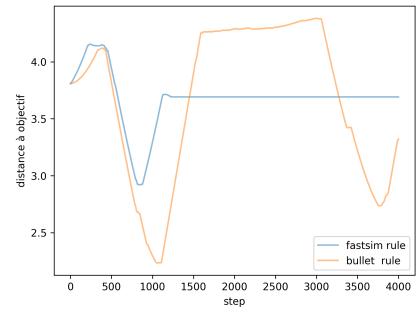
(c) kitchen - écart : 47.16



(d) race track



(e) maze_hard



(f) kitchen

FIGURE 12 – Comparaison des trajectoires Rule based

Le même constat peut être observé pour le contrôleur rule based, un très bon transfert pour race track et maze hard (on remarque une divergence de trajectoire pour maze hard qui refusionne à la rencontre du mur suivant). Par contre pour kitchen nous avons une séparation bien plus tôt et nous observons que sous fastsim le robot est bloqué et reste immobile lorsqu'il a un mur en face de lui.

8 Apprentissage

Dans cette partie, nous allons vous présenter les algorithmes d'apprentissage implémentés ainsi que leur capacité à franchir le reality gap entre les deux environnements de simulation.

8.1 Principe

Nous avons repris le travail effectué en TME dans le cadre de l'UE Robotique et Apprentissage. De ce fait, nous avons implémenté un algorithme génétique NSGA-II et un réseau de neurones à l'aide du framework **DEAP**. Le principe est le suivant, DEAP s'occupe de trouver le meilleur individu possible après n générations, cet individu va "paramétriser" le réseau de neurones qui va guider le contrôleur (et donc le robot) dans fastsim. On utilise ce même individu sur PyBullet et nous comparons la trajectoire obtenue sur fastsim et celle obtenue sur PyBullet. Trois critères sont disponibles pour NSGA-II.

Le critère de fitness, qui représente la distance à l'objectif : l'algorithme garde après chaque génération les individus les plus proches de l'objectif.

Le critère de nouveauté, présenté par John LEHMAN et Kenneth O. Stanley dans leur papier "*Abandonning Objectives : Evolution through the search for Novelty Alone*"[5]. Dans cette variante, au lieu de nous concentrer sur la distance à l'objectif pure et simple, nous privilégions les individus avec les comportements les plus "nouveaux". Pour ce faire, nous avons, comme expliqué dans le papier, quantifié la nouveauté de chaque individu en faisant la somme de la distance dans l'espace comportemental au k plus proches voisins parmi une archive des individus déjà explorés, que nous aurons stockée au préalable, et le reste de la population. Plus cette somme est grande, meilleure est la nouveauté. Cette approche est particulièrement adaptée sur des scènes telles que maze_hard, dans lequel un algorithme glouton qui vise à minimiser la distance à l'objectif coincera le robot dans une impasse.

Le critère mélangeant fitness et nouveauté (nous avons alors un problème d'optimisation multi-objectif). Dans cette variante, au lieu d'avoir à la fin un seul individu, on en aura plusieurs constituant le front de Pareto. Nous avons choisi arbitrairement de ne garder que celui qui finit le plus près de l'objectif pour guider le contrôleur.

Avoir ces 3 variantes permettra d'avoir plus de données de tests, nous pourrons alors, par exemple, voir si ces 3 variantes se transfèrent aussi bien ou non.

Le réseau de neurones chargé de guider le contrôleur dans la simulation est constitué de 2 layers cachés de 10 neurones chacun, 1 layer d'entrée de 10 neurones (10 lasers) et 1 layer de sortie de 2 neurones pour les vitesses de la roue gauche et droite. La fonction d'activation est la tangente hyperbolique. A chaque tour de simulation, le contrôleur chargé du robot va récupérer l'état des senseurs, la donner au réseau qui va répondre en sortie une commande pour la roue gauche et droite.

8.2 Résultats

Après avoir lancé une dizaine de fois un algorithme qui génère et garde les meilleurs individus sur 200 générations ($\mu=100$, $\lambda=100$), nous avions à notre disposition approximativement 1100 individus pour le critère fitness, 300 pour le critère novelty et 400 pour fitness + novelty. Afin de limiter les temps de calculs et sachant que les individus obtenus lors des premières générations ne sont pas forcément pertinents, nous avons, pour chaque critère, retenu les 50 meilleurs individus dans le cadre de nos comparaisons. De la même manière que précédemment, nous calculons la différence entre les trajectoires à l'aide de la méthode des moindres carrés.

itérations	FIT	NS	FIT+NS
1	200+	200+	24
2	49	45	200+
3	200+	200+	59
4	200+	61	200+
5	54	200+	200+
6	200+	159	200+
7	200+	200+	45
8	200+	44	37
9	200+	58	38
10	200+	52	40

TABLE 1 – Nombre de générations nécessaires pour atteindre la sortie selon la méthode et l’itération

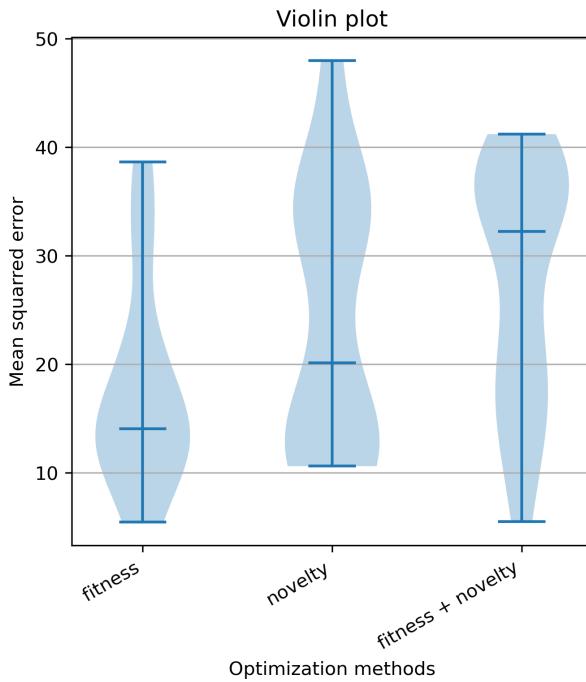


FIGURE 13 – Violin-plot : Mesure de la différence de trajectoires sur 3 politiques

Nous pouvons observer sur le violin-plot que le critère fitness a des différences de trajectoires plus faibles que les deux autres critères, qui eux ont l’air d’être équivalents vis-à-vis de cet aspect. En effet, sa médiane et ses quartiles sont tous inférieurs à ceux des 2 autres variantes. De plus, la largeur du violon nous indique que les individus ont une plus grande probabilité d’avoir un écart inférieur à 20.

En ce qui concerne les critères novelty et fitness + novelty, nous pouvons observer que la médiane pour le novelty, est située à 20, et donc que la moitié des individus comparés ont un écart inférieur à 20, alors que la médiane pour le troisième critère est approximativement à 32. Néanmoins, la largeur du violon, qui indique la probabilité qu’un individu ait cette valeur, semble être maximal entre 30 et 40 pour le critère novelty, ce qui est similaire à la variante fitness + novelty.

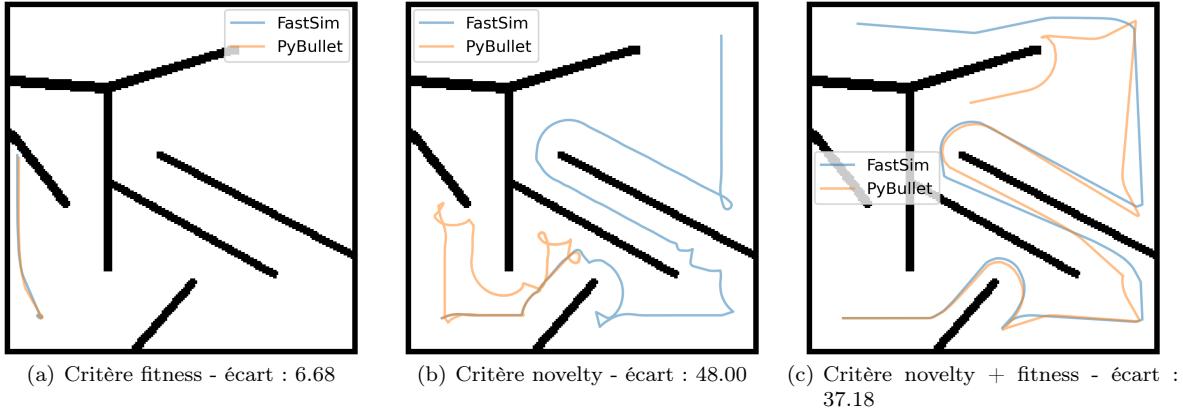


FIGURE 14 – Mesure d'écart de trajectoire

Les observations constatées sur la figure 14 semblent être cohérentes. En effet, lorsque l'algorithme cherche à optimiser la fitness en diminuant la distance à l'objectif, le robot va se coincer, dans les deux environnements, dans une impasse en haut de sa position initiale (figure 14.a). Et dans sa trajectoire vers cette impasse, le robot ne rencontre qu'un ou deux murs qu'il peut facilement longer, ce qui explique qu'il y ait peu de différences et que l'écart mesuré est faible. En revanche, avec les critères novelty et novelty + fitness, le robot parcourt une distance bien plus grande et rencontre plus de murs par rapport au critère fitness (figure 14.b et 14.c). De ce fait, les trajectoires sont plus sujettes à des écarts possibles, ce qui explique les valeurs supérieures obtenues.

9 Conclusion

L'objet de ce projet a été de paramétriser deux environnements de simulations, un étant plus réaliste que l'autre, afin de pouvoir mesurer la capacité de différents algorithmes à franchir le phénomène de reality gap. Pour cela, nous avons fixé divers paramètres dans et modéliser le robot et les cartes de telle sorte que les deux environnements soient le plus proche possible.

Nous avons ensuite implémenté des contrôleurs que nous avons nous-mêmes conçus (Forward, Follow wall), ou récupéré dans un article (Rule-Based, véhicule 3c de Braitenberg). Afin de pouvoir mesurer les écarts de trajectoires, nous avons aligné les vitesses de déplacements du robot dans les deux environnements afin d'obtenir des trajectoires avec un même nombre de points. Enfin, en observant visuellement les trajectoires sur un plot, ainsi qu'à l'aide de la méthode des moindres carrées afin de mesurer les écarts de trajectoires, nous avons conclu que les contrôleurs Rule-Based et de type Braitenberg se transfèrent très bien d'un environnement à l'autre.

La dernière étape du projet a été de répéter le procédé précédent, mais avec un algorithme d'apprentissage machine. Pour cela, nous avons utilisé le framework DEAP afin d'implémenter un algorithme NSGA-II qui est un algorithme d'optimisation multi-objectif. Nous avons exécuté cet algorithme avec 3 critères d'optimisations : minimiser la distance à l'objectif (fitness), nouveauté (novelty) et un dernier critère mélangeant les 2 (fitness + novelty). Enfin, nous avons observé que parmi les individus générés à l'aide de ces critères, ceux obtenus à l'aide de fitness sont les moins sujets au reality gap (bien que le critère de fitness ne soit pas efficace pour atteindre l'objectif, notamment sur la carte maze). Par ailleurs, le calcul d'écarts de trajectoires se fait actuellement en comparant deux à deux les positions échantillonnées des robots. Nous aurions pu améliorer cette comparaison en normalisant les écarts par rapport à la distance parcourue.

Nous avons donc deux environnements qui permettent, ensemble, de mesurer à quel point un algorithme permettant à un robot à roues de se déplacer est sujet au reality gap.

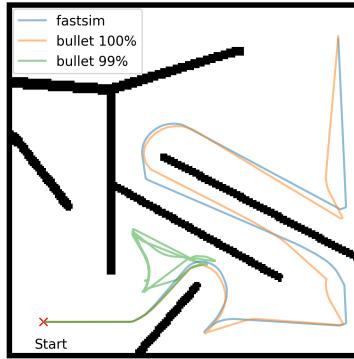


FIGURE 15 – Comparaison selon la précision

Néanmoins, notre environnement PyBullet pourrait être amélioré. En effet, dans nos expériences, nous avions fixé une précision des lasers de 100%. On remarque qu'en changeant cette précision à 99%, et donc en ajoutant un peu d'aléatoire dans les valeurs renvoyées par les lasers, nous pouvions avoir un changement drastique des trajectoires avec le contrôleur de type Braitenberg (fig. 15), qui a normalement un très bon transfert entre fastsim et PyBullet.

Par ailleurs, il se pourrait, comme proposé dans l'article [6], que les simulateurs ne soient jamais assez performants pour éliminer complètement ce phénomène. Une solution suggérée serait de créer de nouveaux

environnements de simulation capable d'attribuer un score à une politique, tout en accordant une valeur de confiance à cette prédition.

Ce type de simulateur permettrait d'éviter d'avoir de l'overfitting, ce qui diminuerait grandement l'impact du reality gap sur le transfert des algorithmes. Deux idées d'approches sont proposées dans l'article, la première consiste à utiliser des algorithmes de type Monte-Carlo, c'est-à-dire à calculer une valeur numérique approchée en utilisant des techniques probabilistes. Dans notre contexte, il s'agirait de lancer de nombreuses fois des simulations en faisant varier divers paramètres, puis de faire des mesures statistiques afin d'obtenir une politique plus robuste.

La deuxième méthode serait de recourir au crowd sourcing. En développant un ensemble d'expériences que les utilisateurs pourront lancer en ligne, on agrandit artificiellement notre puissance de calcul. Les données recueillies pourraient ensuite nous donner de nouveaux résultats sur le reality gap et augmenter notre confiance vis à vis de nos simulateurs. Toutefois, il faut noter que les résultats dépendent assez finement des caractéristiques du robot, du simulateur et de l'environnement comme on l'a montré avec l'impact du laser. Pour avoir des expériences variées et pertinentes, il faudrait donc potentiellement un ensemble d'expériences par robot, par environnement et par simulateur.

Références

- [1] K BOUSMALIS et S LEVINE. "Closing the Simulation-to-Reality Gap for Deep Robotic Learning". In : (2017).
- [2] G BROCKMAN et al. *OpenAI Gym*. 2016. eprint : arXiv:1606.01540.
- [3] R BROOKS. "A robust layered control system for a mobile robot". In : *IEEE Journal on Robotics and Automation* (1985). DOI : 10.1109/jra.1986.1087032.
- [4] E COUMANS et Y BAI. *PyBullet, a Python module for physics simulation for games, robotics and machine learning*. <http://pybullet.org>. 2016–2021.
- [5] J LEHMAN et K STANLEY. "Abandoning Objectives : Evolution through the Search for Novelty Alone". In : *Evolutionary Complexity Research Group School of Electrical Engineering and Computer Science University of Central Florida Orlando*, (2011). DOI : 10.1162/evco_a_00025.
- [6] JB MOURET et K CHATZILYGEROUDIS. "20 Years of Reality Gap : a few Thoughts about Simulators in Evolutionary Robotics". In : *Workshop "Simulation in Evolutionary Robotics", Genetic and Evolutionary Computation Conference, 2017, Berlin, Germany* (2017). DOI : 10.1145/3067695.3082052.
- [7] JB MOURET et S DONCIEUX. "Encouraging Behavioral Diversity in Evolutionary Robotics : An Empirical Study". In : *Evolutionary Computation* 20.1 (2012), p. 91-133. DOI : 10.1162/evco_a_00048.
- [8] P RODUIT, A MARTINOLI et J JACOT. "A quantitative method for comparing trajectories of mobile robots using point distribution models". In : *2007 IEEE/RSJ International Conference on Intelligent Robots and Systems* (2007). DOI : 10.1109/iros.2007.4399126.
- [9] P RODUIT, A MARTINOLI et J JACOT. "Behavioral analysis of mobile robot trajectories using a point distribution model". In : *From Animals to Animats 9* (2006), p. 819-830. DOI : 10.1007/11840541_67.