



Projet de Fin d'Etudes
Localisation collaborative pour robot humanoïde
de Kid-Size League (RoboCup)

L. Delavois, L. Mathé
Encadrant: L. Hofer
Master 2 Informatique (ASPIC)
Université de Bordeaux

9 juillet 2020



Table des matières

Introduction	4
1 Contexte	5
1.1 Rhoban	5
1.2 Le challenge de la RoboCup	5
1.3 Réalisation du projet	6
1.3.1 Challenge simplifié : approche à l'odométrie	6
1.3.2 Solution pour le challenge technique	7
1.3.3 Expérimentations	7
1.3.4 Traitements des données	8
2 Architecture du projet	9
2.1 Description de l'existant	9
2.1.1 Kid-Size	9
2.1.2 Environnement	10
2.1.3 hl_communication	10
2.2 Traitement de données	10
3 Fonctionnalités implémentées	12
3.1 L'observateur – TCCollaborativeObserver	12
3.1.1 La vision	12
3.1.2 Le mouvement de la tête	12
3.1.3 Observation : la position de la balle	13
3.1.4 Observation : la position du robot kicker	13
3.1.5 Positionnement du robot observateur	14
3.1.6 Déplacement	14
3.2 Le kicker	15
3.2.1 La position de la balle	15
3.2.2 La position du robot	16
3.2.3 L'approche	16
3.2.4 Le placer – la V2	16
3.3 Les logs	17
3.4 Le Vive	18
3.5 Traitement des fichiers CSV	18
3.5.1 Récupération et écriture des données	18
3.5.2 Affichage des points sur un graphique	19
3.5.3 Découpage des trajectoires	20
3.5.4 Comparaison entre deux sources	20
3.5.5 Arguments	21

4 Tests effectués	22
4.1 Outils de tests	22
4.2 Simulation	23
4.2.1 Script de test	23
4.2.2 Approche	23
4.2.3 Coopération	24
4.3 Odométrie	25
4.4 Détection des faux positifs	26
4.4.1 La balle et les poteaux	27
4.4.2 Les robots	27
4.5 Test robot seul	28
4.6 Test du challenge	29
4.6.1 ApproachPotential	29
4.6.2 Placer	30
4.6.3 Résultats	30
4.7 Tests sur les fichiers CSV	31
4.7.1 Fichiers de tests	31
4.7.2 Création d'un script	31
5 Problèmes rencontrés et limites	32
5.1 Intégration au code	32
5.2 Les tests	32
5.3 Les comportements	32
5.3.1 Le robot observateur	32
5.3.2 Le robot kicker	33
5.4 Limites du traitement des fichiers CSV	33
6 Conclusion	34
Bibliographie	35
A Localisation sur le terrain	36
B La Vision	37
B.1 Détection du vert	37
B.2 Détection des éléments du terrain	38
B.3 Image synthétisée	39
C La création d'un mouvement	40
D Simulation	41
D.1 Créer des robots en simulation	41
D.2 Rejouer un log de vision	41

Introduction

En 1997, le superordinateur Deep Blue a battu Kasparov aux échecs [Hsu, 1999] et le robot Mars Pathfinder fut le premier robot autonome à atterrir sur Mars [Stevenson, 1997]. Cette année-là, un nouveau challenge est apparu : créer des robots autonomes dotés d'intelligences artificielles qui concourent dans différentes épreuves. C'est ainsi qu'eut lieu la première compétition internationale de robotique appelée RoboCup. Différentes équipes s'affrontent dans différentes ligues dans le but de faire évoluer les nouvelles technologies.

Le football est la ligue phare de la compétition, notamment la Standard Platform League avec les robots NAO [D.Gouaillier et al., 2009] mais on y retrouve aussi la Rescue, @Home, Industrial ou Junior¹.

La ligue humanoïde de football a été créée en 2002 avec des tirs aux buts, les premiers matchs n'apparurent qu'en 2005. À l'origine, cette ligue avait été créée suite à l'objectif lancé par H. Kitano en 1996 : “by the year 2050 develop a team of fully autonomous humanoid robot that can win against the human world soccer champions” [Kitano and Asada, 1999].

Au-delà de la seule intelligence, le sport nécessite des capacités motrices dans un environnement dynamique et seulement partiellement observable. Un robot qui joue au football rencontre des challenges pour se déplacer, observer et coopérer avec ses coéquipiers [Burkhard et al., 2002] (section Humanoid League).

Nous ne sommes pas encore prêts à battre des humains mais les robots de Kid-Size jouent à 4 contre 4 sur un terrain de 9x6m, les premiers matchs à lumière naturelle ont eu lieu l'année dernière et l'évolution est prévue pour compléter ce challenge [Baltes et al., 2019].

L'équipe de Rhoban, avec laquelle nous avons travaillé pour ce Projet de Fin d'Études (PFE), a été créée en 2011. Elle évolue dans la ligue humanoïde en taille Kid-Size (robot entre 40 et 90cm). Dès sa sixième participation, elle a su se hisser au sommet et en 2019, elle a obtenu son quatrième titre consécutif.

L'objectif de ce PFE sera d'utiliser les robots fournis par Rhoban pour effectuer le challenge technique de catégorie Kid-Size de la RoboCup : Collaborative Localisation². Le but de ce challenge est de faire marquer un but à un robot qui ne possède pas de perception de son environnement. Ce PFE sera effectué au sein de l'équipe de Rhoban, nous devons donc réutiliser le code existant et adapter notre projet afin de l'ajouter à terme au code des robots.

1. www.robocup.org

2. Règles de la RoboCup : <http://humanoid.robocup.org/wp-content/uploads/RCHL-2020-Rules-Dec23.pdf>, p66-67 pour le challenge

Chapitre 1

Contexte

1.1 Rhoban

Nous avons effectuer ce PFE au sein du projet Rhoban encadrés par des membres du LaBRI¹, nous permettant d'évoluer dans un monde professionnel, similaire à ce que nous allons rencontrer en stage prochainement.

Pour nos tests en conditions réelles, nous utilisons les robots mis à disposition par l'équipe. Ces robots sont appelés Sigmaban + [Ly et al., 2019], ils sont conçus pour évoluer dans la ligue Kid-Size et mesurent environ 65cm pour 6kg.

À partir de fin février, l'équipe a déménagé dans de nouveaux locaux sur le campus de l'IUT informatique à Gradignan. Nous avons perdu un peu de temps à déménager mais nous avons pu faire des tests réels une fois l'installation du terrain terminée.

L'équipe nous a mis à disposition un HTC Vive qui, grâce à un "tracker" posé sur la tête du robot, nous permet d'avoir sa position avec une meilleure précision que manuellement (maximum 5 cm).

1.2 Le challenge de la RoboCup

La finalité du travail que nous allons effectuer dans ce Projet de Fin d'Études est donc d'accomplir le challenge technique "Collaborative Localisation" de la RoboCup

Le concept du challenge est de faire marquer un but à un robot aveugle guidé par un autre. Nous avons donc le robot tireur, que nous appellerons robot kicker, placé manuellement au milieu du rond central et le robot observateur, placé manuellement n'importe où dans le terrain. La balle est ensuite déposée par l'arbitre dans le rectangle rouge situé devant les cages sur la Figure 1.1 (entre les poteaux des cages, dans l'alignement du point de penalty). Le robot kicker a ensuite deux minutes pour marquer, guidé par l'observateur.

Le robot observateur a la possibilité de se déplacer sur le terrain mais il ne doit absolument pas toucher la balle.

Pour valider le challenge, nous faisons trois essais. Un but représente un succès total. Si le robot ne marque pas mais touche la balle, cela correspond à un succès partiel.

1. Laboratoire Bordelais de Recherche en Informatique

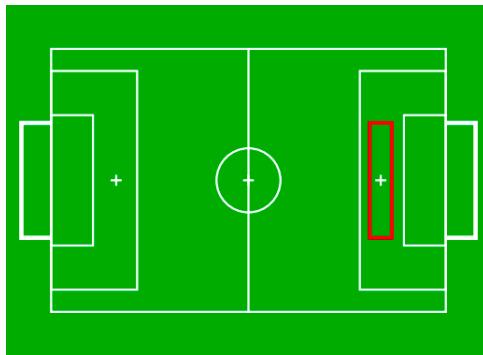


FIGURE 1.1 – Position de la balle au commencement du challenge

1.3 Réalisation du projet

1.3.1 Challenge simplifié : approche à l'odométrie

Dès le premier rendez-vous, nous avons pu voir une évolution claire de ce que pouvait apporter ce projet au code de Rhoban. Nous avons plusieurs étapes dans la réalisation du challenge technique.

La première étape donnée ici est de tester l'approche et l'odométrie² du robot [Roussel et al., 2016]. L'idée est donc dans un premier temps de placer la balle à un lieu donné au robot et de lui faire approcher cette balle sans que celui-ci n'utilise la vision, seulement l'odométrie.

Le but de cette démarche est d'effectuer des tests de fiabilité sur l'approche déjà existante voire de modifier ce mouvement. En effet, si l'approche est concluante, il nous suffira de trouver la position de la balle grâce au robot observateur et alors le robot kicker n'aura qu'à avancer et tirer.

En regardant dans le code de Rhoban, nous avons trouvé deux mouvements d'approches différents : `ApproachPotential` et le `Placer`. `ApproachPotential` a semblé être le plus complexe. Alors que le `Placer` va simplement positionner le robot en position voulue, l'`ApproachPotential` permet d'optimiser la trajectoire afin d'avoir la meilleure position du robot pour pouvoir ensuite tirer.

L'`ApproachPotential` a déjà résolu beaucoup de problèmes relatifs à l'approche : évitement des obstacles, placements optimisés pour le tir. Le problème majeur avec ce mouvement est la complexité de compréhension de ce qui a été fait. Il nous faut comprendre comment utiliser le mouvement avec les contraintes du challenge sans faire de modifications qui pourraient gêner le jeu en match. En revanche, si nous utilisons le `Placer`, nous allons devoir résoudre certains problèmes pas encore pris en compte (ici l'essentiel est de déterminer l'orientation du robot afin d'optimiser les chances de but).

Nous voyons ici sur l'exemple 1.2 que la trajectoire du robot (en rouge) est différente en fonction de l'approche choisie. La position de la balle (en noir) reflète l'orientation du tir. On peut voir qu'avec l'`ApproachPotential` la balle passe au centre des cages alors qu'avec le `Placer` le robot tire tout droit face à lui (la variation en Y est faible). Il est important de noter que les cages se situent en indice en x de 4.5 et un y compris entre -1.2 et 1.2.

2. l'odométrie est la mesure de la position d'un robot à partir des mouvements de ses moteurs.

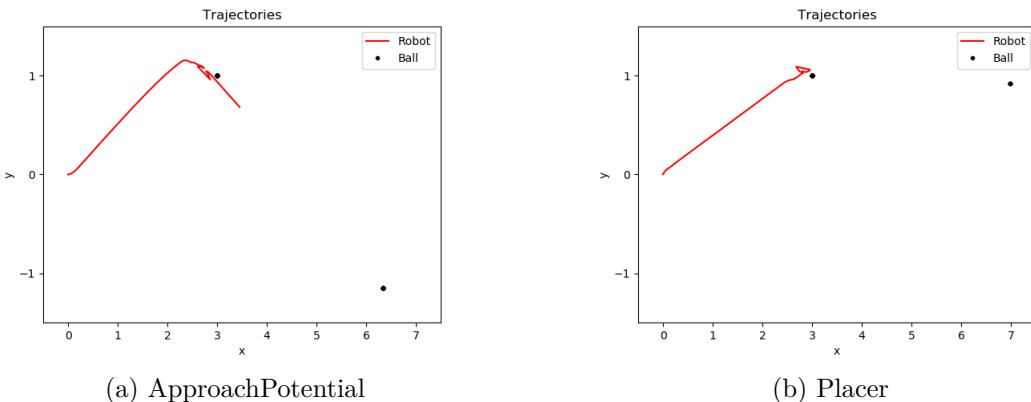


FIGURE 1.2 – Résultats en simulation des différentes approches

1.3.2 Solution pour le challenge technique

L'idée développée pour réussir ce projet est de faire deux comportements, un pour chacun des deux robots. Il y a trois essais pour valider le challenge, il est donc nécessaire que le comportement puisse se réinitialiser lorsque le robot est soulevé (ce qui deviendra l'état HANDLE dans nos comportements).

Le comportement du robot observateur est relativement simple, une fois posé, celui-ci observe son environnement afin de repérer la balle et les adversaires. Si on admet qu'il se déplace alors il nous faudra trouver la distance optimale pour reconnaître les objets sur le terrain. Il suffira alors de le faire se déplacer à la position optimale.

Le robot kicker est un peu plus complexe. Il doit attendre que le robot observateur lui envoie une balle. Ensuite, il peut utiliser un mouvement d'approche pour avancer vers la balle. La vision de l'observateur va lui permettre de confirmer sa position afin de s'assurer qu'il peut tirer.

1.3.3 Expérimentations

La phase de tests est un élément très important pour notre projet. Le but final étant de faire le challenge technique à la RoboCup, la preuve de faisabilité est importante.

Pour les tests des comportements, nous définissons trois différentes étapes :

1. En simulation
 2. Robot seul sur le terrain (à l'odométrie – robot kicker seulement)
 3. En collaboration, comme pour le challenge technique.

Les comportements ont donc été développés pour être capables de fonctionner dans ces trois différentes conditions.

On nous a demandé d'avoir des protocoles d'expérimentation clairs, puis de prendre le temps de traiter les données à travers des affichages graphiques. Nous devons donc trouver une solution pour récupérer des données lors des déplacements et choisir quels éléments sont importants à montrer.

Le but sera aussi de tester différents éléments du robot (comme l'odométrie ou la vision) et de faire des statistiques sur la réussite du challenge.

1.3.4 Traitements des données

Lors de nos tests en simulation ou avec les robots, des fichiers au format CSV³ sont créés et remplis de manière automatique avec diverses informations, essentiellement la position du robot et de la balle. Chaque ligne du fichier est écrite toutes les 0.2s et contient donc une information pour chaque champ.

L'objectif ici est de pouvoir convertir ces fichiers CSV en courbes sur des graphiques, pour les rendre plus compréhensibles. On pourra également comparer ces courbes issues des données de l'approche avec celles du Vive afin d'estimer la précision de la localisation du robot.

Avoir un affichage visuel nous permet de traiter rapidement une grande quantité d'informations et nous pourrons analyser efficacement les résultats des tests effectués sur les robots.

3. Comma-separated values

Chapitre 2

Architecture du projet

2.1 Description de l'existant

La majeure contrainte est l'intégration au code de Rhoban. Ce code est complexe et possède plus de 4000 fichiers. Il est important de comprendre l'existant et son fonctionnement afin de se lancer dans le projet. Nous n'utiliserons pas tout le code Rhoban mais nous devons quand même mélanger Vision, Localisation et Mouvement, ce qui représente une quantité de code importante. Il est nécessaire de prendre du temps pour assimiler les différents éléments dont nous allons avoir besoin.

Notre travail est disponible dans la branche `collaborative_localisation` du Github de Rhoban¹.

2.1.1 Kid-Size

Le package **Kid-Size** est celui qui contient tout le code permettant de faire fonctionner le robot, il se partage entre **Motion** et **Vision**.

Nous pouvons retrouver dans la partie **Motion** tous les mouvements des robots et la partie décisionnelle où il y a les stratégies de haut niveau (Annexe C). Nous avons donc dans la partie **Vision** le traitement de l'image (Annexe B) et la localisation.

Dans le package **Kid-Size**, nous pouvons également trouver des services. Les services sont lancés dès le lancement du robot et ont différentes utilités :

- **CapitaineService** : utilisé par le capitaine pour élaborer les stratégies de jeu.
- **DecisionService** : contient divers éléments utiles à la décision, par exemple sur l'état du robot, la qualité des informations de sa localisation.
- **LocalisationService** : gère toute la localisation du robot : sa position, celle de la balle et celle des autres robots vus.
- **LogService** : crée des fichiers CSV permettant de récupérer des informations sur les robots.
- **TeamPlayService** : utilisé pour la création et la lecture des messages.

Pour vous donner un ordre d'idée, vous pouvez retrouver en figure 2.1, l'architecture du package en utilisant seulement les éléments essentiels que nous avons utilisés. En bleu, vous retrouverez les noms des dossiers, en vert les fichiers (par soucis de simplification, il y a des "regroupements" de fichiers en orange).

1. <https://github.com/Rhoban/> : il faut demander les droits d'accès au client pour pouvoir récupérer notre code.

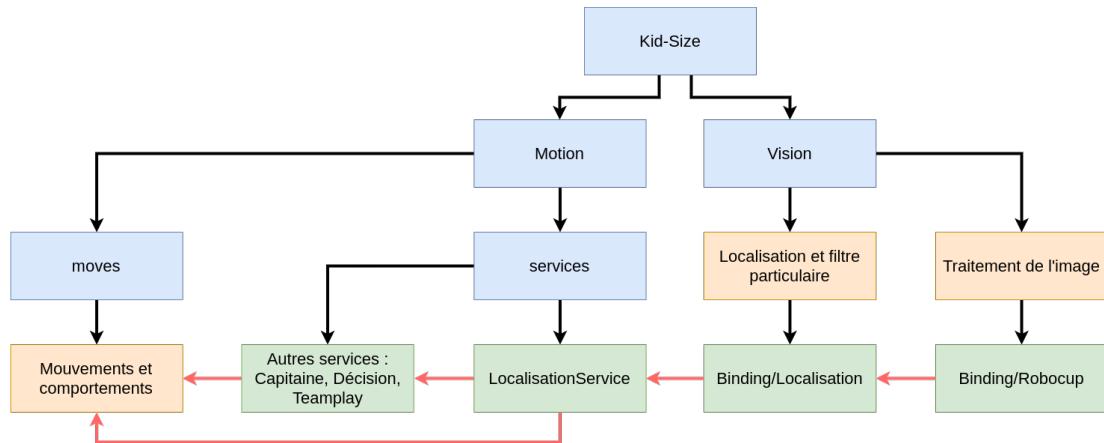


FIGURE 2.1 – Architecture simplifiée du package Kid-Size

Les flèches noires montrent la hiérarchie alors que les rouges expliquent la transmission de l'information.

La majeure partie de notre contribution se trouve donc dans ce package. C'est là que nous avons développé nos comportements, tout en modifiant certains services. Il nous a fallu surtout du temps pour comprendre et savoir où récupérer chaque information dont nous avons besoins.

2.1.2 Environnement

Le package **environnements** contient quant à lui tous les fichiers de configurations des robots comme les moteurs ou les paramètres de la vision. Nous y retrouvons un dossier pour chaque robot et un dossier commun. Il permet également de lancer les simulations de jeu grâce au robot fake. C'est ici que nous avons créé les scripts de simulations.

2.1.3 hl_communication

La communication entre les deux robots est essentielle dans ce challenge. Nous n'avons pas effectué de modification dans le package **hl_communication** mais c'est grâce à lui que transitent les informations entre les robots. Les messages sont envoyés sous forme de protobuf.

Nous avons du en revanche décortiquer les messages et nous avons utilisé le **TeamPlayService** de **Kid-Size** pour envoyer et récupérer les informations nécessaires.

2.2 Traitement de données

Afin de pouvoir traiter correctement les données issues du **LogService**, nous avons créé un dépôt Github intitulé **PFE_datatreatment**. Ce dernier contient notamment les programmes Python de traitement de données au format csv et les programmes de tests. Python est un langage simple et efficace pour générer des graphiques à partir de données grâce à la bibliothèque **matplotlib**. Notre choix s'est donc naturellement porté vers cette bibliothèque pour répondre à notre besoin.

Nous avons partagé notre travail en deux fichiers. Le premier, `divisionCsv`, va nous permettre d'effectuer un premier traitement du CSV, pour séparer le fichier en sous-fichiers représentant chacun une trajectoire (figure 2.2).

Il sera quand même possible d'afficher ou d'enregistrer les graphiques de ces trajectoires.

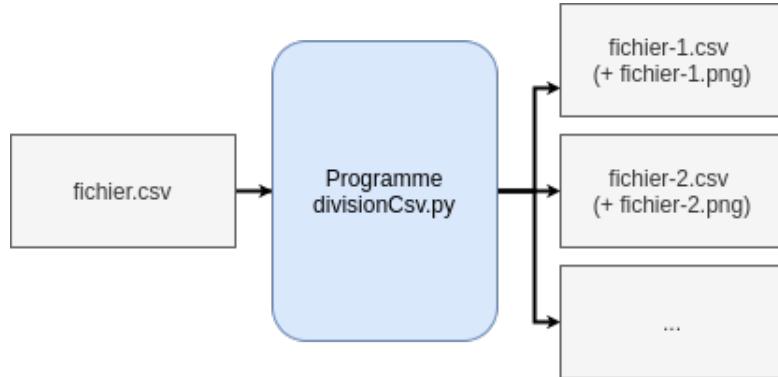


FIGURE 2.2 – Architecture du programme `divisionCsv.py`

Le second fichier, `comparePlots`, sert quant à lui à comparer deux fichiers CSV sur un même graphique. Cela nous permet principalement de confronter les résultats du Vive à ceux récupérés depuis le robot (figure 2.3).

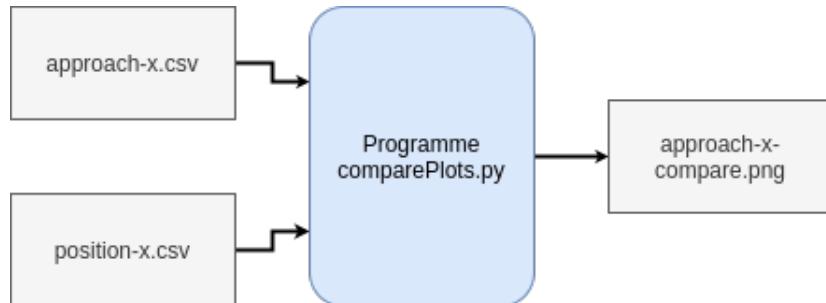


FIGURE 2.3 – Architecture du programme `comparePlots.py`

Chapitre 3

Fonctionnalités implémentées

3.1 L'observateur – TCCollaborativeObserver

Le but du robot observateur est principalement d'envoyer les données extraites dans la Vision au robot kicker. Les deux éléments principaux sont donc la position de la balle et du robot.

3.1.1 La vision

Contrairement à d'autres éléments du code, nous n'avions aucune connaissance de base sur le fonctionnement de la vision des robots. La vision de Rhoban se base sur une détection du non-vert. Comme le terrain est en herbe synthétique, il est majoritairement vert. Nous détectons les régions d'intérêts en reconnaissant les éléments présents sur le terrain qui ne sont pas vert. Un réseau de neurones se charge ensuite de la reconnaissance de l'objet. Une description plus précise de la Vision est décrite en Annexe B.

L'arrivée dans un nouveau terrain est également synonyme de réglages de la vision afin que le robot s'adapte à son nouvel environnement. Nous avons demandé de l'aide pour les réglages de la perception du vert, de la luminosité et de la détection des robots (au début confondu avec les poteaux car le blanc était trop pris en compte).

Nous avons donc bien vérifié que le robot Observateur pourrait voir le robot Kicker et la balle sans trop de problème dans ce nouvel environnement (tests en 4.4).

3.1.2 Le mouvement de la tête

Le mouvement Head définit le balayage de la tête du robot pour qu'il observe tout autour de lui afin de se localiser et de repérer les zones d'intérêts.

Pour ce mouvement, nous connaissons la zone d'intérêt du robot et le besoin de se localiser est faible car ses déplacements sont minimes. Nous avons fait quelques modifications sur le mouvement de la tête. La première a été de réduire la vitesse de scan afin d'avoir des informations plus précises. En effet, en extrayant la position des objets vus par le robot, nous devons nous baser sur l'angle de la caméra. Avec une vitesse plus rapide, nous avons donc plus d'imprécision sur l'orientation de la tête (retard entre la position réelle du moteur et celle envoyée par les capteurs de

position des moteurs). Également, nous avons réduit l'angle de scan afin de focaliser le robot sur ce qu'il voit devant lui : c'est là que devrait se situer la balle et le robot kicker. Il y aura donc moins d'images inutiles (sans robot ni balle) et nous pourrons les voir à plus haute fréquence, permettant plus de précision.

3.1.3 Observation : la position de la balle

Décision

Afin d'éviter les faux positifs (test de la vision pour les faux positifs section 4.4), nous faisons une confirmation de la balle observée. Lorsque nous devons récupérer une nouvelle position de balle, nous enregistrons plusieurs positions de la balle et nous acceptons la position seulement si le robot la voit plusieurs fois au même endroit.

Transmission

Il y a deux positions de balles partagées par les robots. La première se trouve dans la `DecisionService` et correspond à la `SharedBall`. Pour déterminer quelle position de la balle on choisissait, on prenait celle dont la distance avec le robot qui l'a vu était la plus petite. Ce n'était pas optimal car elle ne nous protégeait pas forcément des faux positifs. Elle était majoritairement utilisée avant la création du `CapitaineService`.

Maintenant Rhoban utilise donc la `common_ball` issue du `CapitaineService`. Pour déterminer celle-ci, il faut prendre toutes les balles envoyées dans les messages et les regrouper par cluster. Chaque cluster a un score en fonction du nombre de balles, la distance entre chacune de ces balles et la distance par rapport l'ancienne balle si possible. La balle choisie est la balle représentant la moyenne du cluster ayant le meilleur score. Ici, étant donné qu'un seul robot envoie sa balle, on est sûr que la balle fournie est bien celle de l'observateur.

Pour que la `common_ball` soit partagée, il faut que le robot qui l'a envoyée soit entrain de jouer. Or, comme ici le challenge ne fait pas partie d'un match, le robot ne joue pas. Nous avons donc simplement changer cette condition par `not penalized` qui indique que le robot n'est pas hors jeu. Il faudra ici vérifier que cette modification n'altère pas le jeu habituel (mais c'est ce qui est utilisé pour définir le capitaine).

Comme le robot kicker ne voit pas de balle, la balle sera forcément celle vue par le robot observateur. Le robot kicker pourra donc récupérer cette balle grâce aux informations du capitaine. Par défaut, nous avons forcé le robot observateur en capitaine pour savoir facilement si le robot kicker est seul ou non. Si le robot kicker est capitaine, alors il ne reçoit pas de message de la part de l'observateur, en cas contraire, le capitaine contiendra la balle de l'observateur.

3.1.4 Observation : la position du robot kicker

La position du robot kicker perçue par le robot observateur est importante car afin d'avoir des informations les plus précises possibles, nous nous plaçons dans le même repère pour la balle et le robot. En effet, de toutes les informations que nous possédons, nous avons la balle perçue par l'observateur B_o , la position du kicker perçue par l'observateur, K_o et la position du kicker obtenue par l'odométrie donc

lui-même, K_k . Ici, le choix a été d'utilisé K_k seulement pour vérifier que la position K_o n'est pas un faux positif. Nous nous situons donc exclusivement dans le repère du robot observateur et nous évitons les changements de repère pour passer de l'un à l'autre. Ainsi, même si ces positions ne sont pas exactes, l'écart entre le robot et la balle sera plus précis que celui entre la balle du repère observateur avec la position récupérée à l'odométrie du robot kicker.

Les informations transitent entre les robots grâce à un système de messages (protobuf). Les robots vus sur le terrains sont appelés automatiquement *opponent* (car c'est surtout utilisé pour éviter les autres robots). Les robots vus par les coéquipiers sont reçus par messages et stockés dans *sharedOpponent*.

Pour que le robot kicker se localise, il récupère la liste *sharedOpponent* et retrouve dans celle-ci la position qui lui semble être le plus probablement la sienne.

3.1.5 Positionnement du robot observateur

Nous avons essayé deux positions pour le robot observateur. Pour la première, le robot est positionné au milieu des cages, regardant vers le terrain et placé en (4.5, 0, 180). La seconde, le robot est aligné en x avec le point de penalty , à côté de la zone de placement de la balle afin de pouvoir la repérer facilement, en (3, -1.75, 90). Pour l'explication de la localisation sur le terrain et des positions des robots, voir l'annexe A.

L'avantage de la première position est d'avoir une bonne précision quant à l'alignement du robot avec la balle en y , c'est-à-dire si le robot et la balle sont bien alignés vers les cages. En revanche, pour le robot se situant dans les cages, il existe une possibilité que celui-ci arrête la balle (c'est ce qu'il s'est passé lors d'un de nos tests en collaboration), surtout avec le mouvement de l'**ApproachPotential** qui vise le centre des cages.

La seconde position permet d'être plus précis sur l'écart entre le robot et la balle, pour savoir si celui-ci peut tirer, le robot étant reculé. Il est quand même possible de bien estimer l'alignement en y . C'est donc cette dernière solution qui a été retenue.

Nous pouvons également choisir une autre position en changeant simplement les valeurs de `start_robot_x`, `y` et `angle`.

3.1.6 Déplacement

Dans les règles du challenge, le robot observateur a la possibilité de se déplacer, nous pouvons changer la position du robot en fonction de la position de la balle. En admettant que le robot se situe dans la seconde position (celle qui a été retenue), c'est-à-dire sur le côté de la balle, celle-ci peut se situer au début du challenge à une distance de 50cm à 3m. La précision des informations du robot observateur ne sera pas uniforme selon la distance. Il a donc été nécessaire de trouver la position optimale pour observer le robot qui tire dans la balle.

Le problème du déplacement du robot est une perte de précision dans sa position – par conséquent sur la position de la balle et du robot observé. La perte de précision sur la localisation est due à une imprécision sur l'odométrie (voir les tests 4.3 sur l'odométrie). Comme nous utilisons la position relative du robot kicker par rapport à la balle pour savoir si un tir est possible, leur position sur le terrain ne sert qu'à

décider l'orientation du tir. Comme la balle se situe initialement à 1m des cages, l'angle de tir du robot est assez large pour permettre un but.

Le déplacement du robot observateur représente donc une optimisation de la localisation.

Après avoir cherché la distance optimale entre l'observateur et la balle (pour l'instant à la main, des tests plus précis sont à effectuer), nous avons décidé de placer le robot à 1m. Pour cela, nous utilisons le **Placer** car il ne sert à rien d'utiliser un mouvement complexe pour simplement se placer, ici, il n'y a pas de tir ensuite.

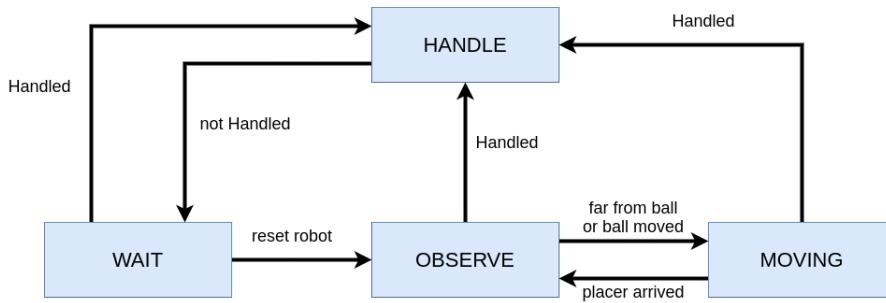


FIGURE 3.1 – Comportement du robot Observateur

Dans l'optique d'avoir un déplacement optimisé avec la position choisie du robot (position 2, annexe A), le déplacement du robot le place toujours aligné en x avec la balle, à 1m en moins en y regardant vers la balle. Depuis la position 2, le robot n'a qu'à avancer et reculer.

Si nous voulons tester une autre position et nous ne voulons pas que le robot bouge, il suffit de mettre le booléen `moving` à `false` ce qui empêche de passer à l'état MOVING (figure 3.1).

3.2 Le kicker

3.2.1 La position de la balle

La position de la balle est donc récupérée à partir du capitaine. Malgré cela, le système de vision du robot kicker essayait quand même de récupérer une balle et celui-ci étant aveugle (cache devant la caméra), il n'en voyait pas. Si aucune balle n'est perçue par un robot, celui-ci passe en mode "NoBall" et la qualité de la balle passe automatiquement à 0. Nous avons donc dû empêcher cela et désactiver la liaison entre la vision et la localisation.

Le robot n'ayant plus d'informations de la vision, nous devons donc directement utiliser la fonction `setBallWorld` (qui définit la position de la balle directement dans `LocalisationService`) et non `customBallReset` (qui définit la position de la balle dans la vision, souvent utilisée pour aider le robot à repérer la balle, la vision appelant par la suite `setBallWorld` pour confirmer la position).

Nous avons également rajouté une condition à chaque ré-initialisation du robot qui oblige que la première balle reçue se trouve dans la zone autorisée par les règles du challenge technique. En attendant d'avoir une balle respectant ces conditions le robot reste immobile.

Également, si le robot ne reçoit plus de balle de la part de l'observateur alors il garde la dernière position reçue.

3.2.2 La position du robot

Nous recevons donc une liste des *opponents* vus par le robot observateur. Il nous faut ensuite savoir si le robot observateur nous a vu. Pour cela, nous récupérons l'opposant le plus proche de la position déduite à l'odométrie et nous vérifions si cela peut correspondre à une position proche du robot. Par exemple si le robot observateur ne voit que des faux positifs sur des objets au loin, cela ne sera pas pris en compte et le robot kicker gardera sa position trouvée à l'odométrie.

3.2.3 L'approche

Dans le TCCollaborativeKicker, nous utilisons le mouvement **ApproachPotential** pour aller vers la balle. C'est le mouvement d'approche complexe de Rhoban permettant d'approcher la balle.

Nous avons fait des premiers essais sur l'approche où le robot connaît d'avance la position de la balle. Il peut faire le mouvement kicker en autonomie en utilisant seulement l'odométrie. Nous avons réussi à avoir des résultats plutôt satisfaisants (but), ayant quand même parfois de légers décalages, empêchant donc le robot de marquer un but. Dans ce cas, n'ayant pas d'observateur pour appuyer, l'essai est irrécupérable.

Lors des premiers tests à deux, le problème majeur que nous avons eu est lorsque l'on essaye de réinitialiser la position du kicker d'après la vision de l'observateur. Celui-ci ne prend pas en compte l'orientation du kicker et même en essayant de reprendre l'orientation du robot, celui-ci tourne en rond.

Après avoir résolu ce problème d'orientation (angle en radian non degrés). Nous avons eu des résultats corrects sur ce mouvement.

Le mouvement **ApproachPotential** étant complexe, le comportement TCCollaborativeKicker est simple (figure 3.2). À noter que la position de la balle et du kicker sont réactualisées toutes les 5 secondes.

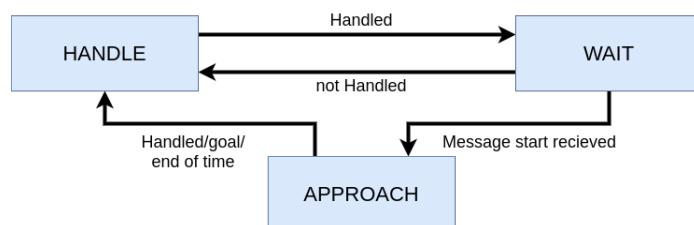


FIGURE 3.2 – Comportement du robot Kicker avec ApproachPotential

3.2.4 Le placer – la V2

Le mouvement **ApproachPotential** a été codé par Rhoban, le mouvement étant peu documenté, il est difficile d'identifier les causes de dysfonctionnements encore plus de modifier le mouvement. Nous pouvons difficilement savoir si les modifications vont impacter les performances en match. N'ayant pas de bon résultats au début à cause des problèmes d'orientation, nous voulions réussir à avoir un mouvement fonctionnel. Le plus simple pour nous a donc été de reprendre le mouvement TCCollaborativeKicker en remplaçant l'approche par un mouvement plus simple,

en utilisant le **Placer** et en essayant de réduire au maximum les variations d'orientation.

Le mouvement est donc simplifié, il prend plus de temps pour que le robot kicker puisse réinitialiser sa position sans qu'il soit en mouvement. Le but est que lorsqu'il reçoit la position de la balle, celui-ci se rapproche grâce au **Placer**. Lorsque le robot est arrivé à sa position, il s'arrête, récupère sa position et celle de la balle. S'il peut tirer, alors il s'exécute. Sinon, celui-ci se replace selon les observations reçues. Un des problèmes que nous avons eu à résoudre a été d'éviter que le robot se bloque entre la marge de placement de **Placer** et la marge que nous avons donné pour notre zone de tir. En effet, le robot peut s'estimer assez bien placé pour la position donnée au **Placer** mais pas assez pour tirer. Nous avons donc dû réduire les marges de placement du **Placer**.

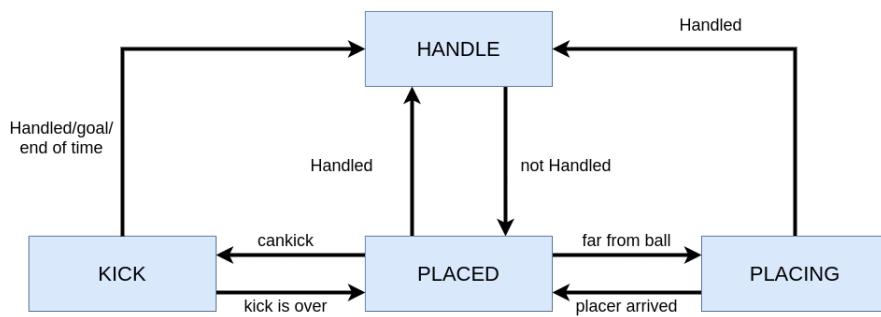


FIGURE 3.3 – Comportement du robot Kicker avec Placer

L'avantage du **Placer** c'est qu'on peut choisir d'éviter au robot de changer d'orientation lors du placement jusqu'à la balle. Ceci permet de limiter les erreurs d'orientation et donc d'être mieux placé.

Finalement, ces deux approches différentes nous ont donné des résultats satisfaisants lors des tests à deux, section 4.6.

3.3 Les logs

Les services sont des classes exécutées automatiquement lorsque le robot est lancé. Le **LogService** a pour but de créer des logs sous la forme de fichiers CSV afin de faire des relevés sur certaines informations du robot pour pouvoir comprendre ce qu'il se passe. Il est utilisé par Rhoban pour faire des relevés sur l'état continu des moteurs afin de repérer facilement la casse et de comprendre la cause.

Il nous a donc semblé naturel d'utiliser ce service afin de récupérer les informations du robot lors de l'approche. Nous avons donc déplacé le relevé des moteurs dans une fonction et puis nous en avons créé une autre ouvrant un autre fichier. Cette dernière permet de sauvegarder les éléments nécessaires au traitement des données de l'approche.

Nous avons décidé de sauvegarder le temps depuis le lancement du programme du robot, la position du robot, de la balle, la position du robot vu par l'autre robot et l'état dans lequel la machine à état du robot kicker (figure 3.3).

La fonction de log que nous avons créée n'est adaptée qu'au comportement du kicker (car c'est sa position qui nous intéresse).

3.4 Le Vive

Le Vive a donc été installé par les membres de Rhoban. Nous avons fixé le tracker sur la tête du robot kicker afin d'avoir sa position.

Pour utiliser le Vive, nous avons travaillé sur l'ordinateur de Rhoban qui a été rattaché au matériel.

Le package `vive_provider`¹ contient déjà tous les éléments nécessaires à l'utilisation du Vive. Après avoir allumé le Vive avec Steam, le fichier `vive_field_calibration` permet de recréer le terrain en simulation dans l'univers Vive afin d'avoir les bonnes coordonnées de tracker.

Ensuite, nous lançons le `vive_server` qui récupère les informations sur le tracker et les envoie sur le réseau. Nous avons ensuite modifier le `vive_client` afin de pouvoir enregistrer les informations reçues. Par des soucis de cohérence et de simplicité, nous utilisons le format CSV et nous ne gardons que la position du tracker que nous allons comparer à celle du robot.

Le `vive_client_to_csv` est disponible dans le dossier `vive` de notre Git `PFE_datatreatment`.

3.5 Traitement des fichiers CSV

Pour effectuer le traitement automatique des fichiers CSV, nous avons commencé par écrire un programme simple premièrement. Ensuite, nous l'avons complexifié au fil du temps en fonction de nos attentes.

3.5.1 Récupération et écriture des données

Nos fichiers de données contiennent plusieurs champs :

- `time` : indique à quel moment les positions ont été sauvegardées depuis le début de la simulation
- `pos_ballX` et `pos_ballY` : la position de la balle
- `pos_robotX` et `pos_robotY` : la position du robot tireur
- `seen_robotX` et `seen_robotY` : la position du robot tireur vue par le robot observateur
- `state` : l'état de la machine à état de TCCollaborativeKicker dans lequel le robot kicker est.

La première étape était donc d'ouvrir un fichier à lire et un autre fichier dans lequel écrire. Les chemins de ces deux fichiers ont d'abord été écrits en dur dans le code. Les données d'une même ligne sont séparées d'une virgule et chaque ligne est séparée d'un retour chariot. Ces caractéristiques permettent de considérer les données stockées sous forme d'un tableau que nous pourrons manipuler par la suite. Une simple boucle permet alors de lire chaque ligne et de l'écrire dans le fichier de destination. Une fois la boucle terminée, il faut bien penser à fermer la lecture et l'écriture des fichiers.

1. https://github.com/Rhoban/vive_provider

3.5.2 Affichage des points sur un graphique

Pour vérifier que notre fichier source a bien été copié dans un autre, nous pouvons afficher ces deux fichiers dans un graphique grâce à la bibliothèque `matplotlib`. L'écriture des données et l'affichage étant indépendants, nous pouvons créer deux fonctions distinctes. Les points que nous affichons sont les coordonnées (x, y) du robot. La première modification du code a effectué était de sauvegarder l'en-tête dans une variable car nous souhaitons l'écrire mais pas l'afficher dans le graphique. En précisant le nom des axes, le titre principal et les étiquettes correspondant au type de fichier, voici le rendu du programme à la figure 3.4.

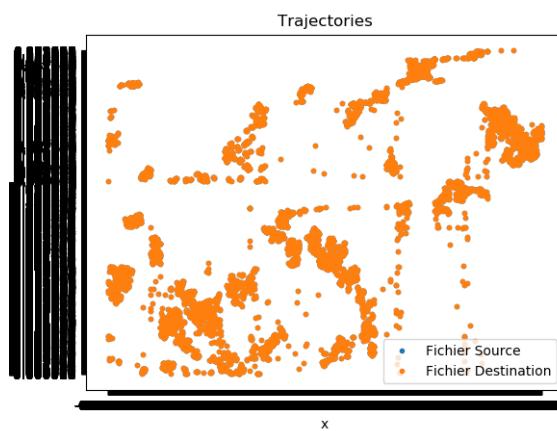


FIGURE 3.4 – Premier rendu d'affichage des points

On constate que les points issus du fichier source se superposent avec ceux du fichier de destination. On admet ainsi que notre copie s'est correctement effectuée. Cependant, notre graphique ne correspond pas encore à ce que l'on souhaite. En effet, notre fichier contient beaucoup de points trop proches les uns des autres ce qui ne représente pas une information utile et ralentit le programme. Ce problème n'existe pas lorsque nous testions notre code avec un fichier CSV court et écrit à la main. De plus, `matplotlib` marque par défaut chaque point sur les axes, qui deviennent rapidement illisibles.

La première amélioration du rendu graphique a été donc d'alléger le rendu en comparant chaque ligne avec la précédente et de ne la copier que si la différence entre les deux lignes dépasse un certain seuil (ici 0.03). Cela nous a permis de ne pas sauvegarder les lignes où le robot ne se déplace pas assez (ou robot à l'arrêt). On choisit de ne comparer que les valeurs en x car le robot avance essentiellement vers les cages donc en x (voir annexe A). Ensuite, nous avons déclaré nos axes et leurs marqueurs associés de manière à ce qu'ils ne dépendent pas des points affichés. Il manquait aussi l'affichage de la position de la balle sur le graphique. Pour cela, nous vérifions que l'en-tête du fichier contient bien les champs concernant la balle. Ensuite, de la même manière que pour la position du robot, nous stockons à chaque ligne la position de la balle dans des variables.

Comme illustré sur la figure 3.5, nous pouvons distinguer les différentes trajectoires suivies par le robot kicker. On peut différencier les trois cibles différentes en $[3, -1]$, $[3, 0]$ et $[3, 1]$. Ces cibles correspondent au placement à la main de la balle sur le terrain lorsque le robot la détecte. La position par défaut de la balle dans le fichier est $[0, 0]$ lorsqu'il n'y a pas de détection.

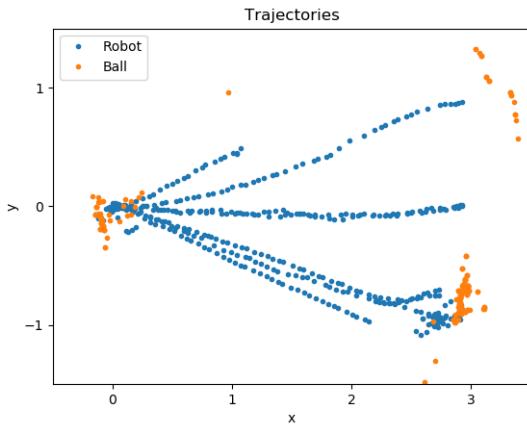


FIGURE 3.5 – Réduction du nombre de points à afficher

3.5.3 Découpage des trajectoires

Une fois arrivés à afficher correctement les données, il faut pouvoir isoler chaque trajectoire sur un graphique différent. Pour cela, il faut modifier les deux fonctions déjà écrites. Nous avons défini comme nouvelle trajectoire un changement en x d'une distance supérieure à 2m. Alors nous écrivons les lignes suivantes dans un nouveau fichier. Ainsi à partir de la source `fichier.csv`, on obtient les fichiers de destination `fichier-1.csv`, `fichier-2.csv` etc. La fonction de division du fichier va compter le nombre de trajectoires dans une variable qui est lue par la fonction d'affichage des points. Nous avons alors décidé d'afficher chaque trajectoire sous forme de ligne comme illustré sur les figures 3.6.

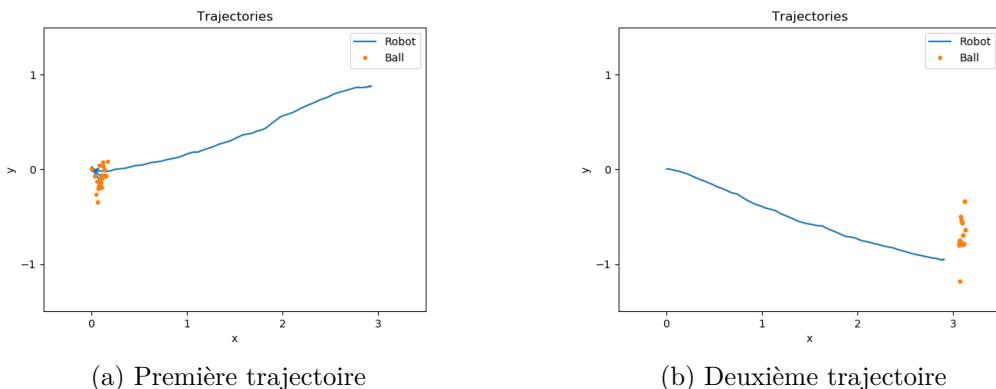


FIGURE 3.6 – Affichage d'une trajectoire par graphique

3.5.4 Comparaison entre deux sources

Nous pouvons ensuite comparer les trajectoires provenant de l'odométrie avec celles du Vive. Pour cela, nous avons écrit un nouveau programme `comparePlots.py`. Celui ci, prend en argument le chemin du fichier de l'approach puis de celui du vive. De la même manière que le programme précédent, une boucle permet d'ajouter la position à chaque ligne dans des tableaux puis de les afficher dans un graphique.

Un autre argument (`--target`) demande de spécifier les coordonnées (x, y) de la cible que le robot kicker est censé atteindre. Cette cible est ensuite affichée sous forme d'étoile sur le graphique comme illustré sur la figure 3.7.

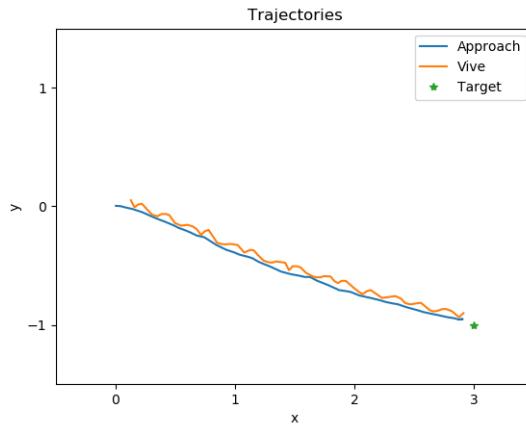


FIGURE 3.7 – Comparaison des deux trajectoires

3.5.5 Arguments

La dernière étape du programme de traitement du fichier CSV a été de définir des arguments à écrire en ligne de commande pour pouvoir l'exécuter et faciliter son utilisation par l'utilisateur. En premier, l'utilisateur doit désigner avec `-f` le fichier CSV à traiter. Si le fichier n'a pas la bonne extension, alors le programme s'arrête en indiquant l'erreur. Ensuite, les options `-save` et `-show` permettent de sauvegarder les graphiques dans le même dossier où se trouve les données puis de les afficher.

Enfin, une autre fonction lit chaque fichier csv généré pour créer des graphiques clairs. Un jeu de couleurs sert à différencier la position du robot par rapport à celle de la balle.

Chapitre 4

Tests effectués

4.1 Outils de tests

De nombreux outils de travail sont fournis dans le code de Rhoban. Le plus utilisé est l'interface **RhIO** (Rhoban Input/Output)¹ permettant de se connecter sur un robot lorsque celui-ci fonctionne².

L'interface **RhIO** utilise une architecture client/serveur [Rouxel et al., 2015]. Le client est le côté utilisateur, sous un interface de type terminal, il permet d'inspecter l'état de variables, de les modifier à la volée et de déclencher des mouvements. Il est également possible de voir la vision du robot avec les différents filtres. Le serveur contenu à bord du robot, permet de stocker des paramètres et de répondre aux services demandés par l'utilisateur.

La modification des variables se fait par le *bind*. Pour pouvoir faire des réglages lors des phases de tests on utilise **Push**. **Push** sert à afficher l'état des variables au cours du mouvement. Cela sert à rendre le code modulaire et changer rapidement les variables *bindées* pendant les phases de tests.

Par exemple, si nous voulons tester le **Throw-In** (la touche à la main, implantée l'an passé) il suffit de changer le nom du tir avec la commande **moves/kick/kickName = throwin** puis d'exécuter le tir avec **kick**.

Il est possible de se déplacer dans l'interface **RhIO** comme dans un vrai terminal avec des commandes similaires (**cd**, **ls** par exemple).

Également, nous avons le **BehaviorViewer** (Figure 4.1) qui permet d'observer le comportement d'un robot sur le terrain. Il représente un terrain vu de dessus avec un robot, une balle et si besoin des obstacles. Nous pouvons les déplacer manuellement ou lancer des mouvements et des comportements. Il sert beaucoup lorsque l'on développe des stratégies de jeu. Nous pouvons l'utiliser lorsque nous lançons des robots en réel ou en simulation. À noter que le **BehaviorViewer** est relié à **RhIO**, une commande sur l'un sera répercutée sur l'autre.

Nous avons également modifié le **BehaviorViewer** pour pouvoir afficher les *sharedOpponent* sur le robot kicker. (voir section 3.1.4).

1. <https://github.com/Rhoban/RhIO>

2. vidéo explicative : <https://www.youtube.com/watch?v=M0izgXYENLc&feature=youtu.be>



FIGURE 4.1 – BehaviorViewer

4.2 Simulation

Le lancement d'une simulation comme décrite en annexe D nous permet de tester nos comportements avant de les lancer sur les robots et de pouvoir commencer les tests avant le déménagement et le montage du terrain. Nous pouvions ainsi vérifier que les machines à états implémentées transitent bien aux bons endroits.

Nous avons pu repérer quelques erreurs sur la simulation mais rien ne compromettant le test des mouvements. Un exemple flagrant est sur le tir. En effet, quelque soit la position du robot et de la balle sur le terrain, si l'on lance la commande `kick`, la balle se déplace dans la direction du robot comme si le robot était en position de tir à côté de la balle.

4.2.1 Script de test

Créer un script nous permet donc de lancer plusieurs robots sans avoir de multiples terminaux lancés en simultané. En effet, pour tester le comportement, il nous faut lancer la simulation, l'interface RhIO et le BehaviorViewer.

4.2.2 Approche

La première idée a été de créer un script permettant de tester les deux différents mouvements d'approche, `ApprochePotential` et `Placer`. Le but sera ici de pouvoir lancer plusieurs robots qui vont aller se placer et tirer dans la balle.

Le script se compose d'une boucle `for` qui va créer différents dossiers `tmp_robot_log_x` chacun représentant une simulation d'un robot qui va tester un mouvement d'approche puis l'autre. La balle est placée de façon aléatoire sur la zone autorisée de placement de balle (figure 4.2).

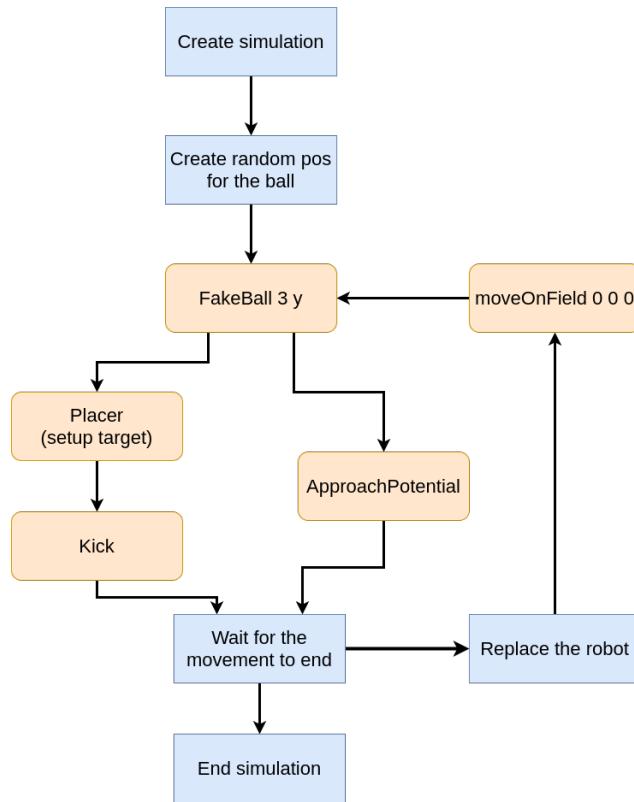


FIGURE 4.2 – Test en simulation de l’approche

Le résultat de ce test a été détaillé dans la partie contexte, section 1.3.1. Il nous a simplement servi à visualiser les différents types d’approche. C’est pourquoi notre première idée a été d’utiliser le mouvement ApproachPotential car il semblait plus efficace.

4.2.3 Coopération

Le challenge technique que nous devons réaliser lors de ce projet se compose de deux robots. Il nous a donc paru essentiel que l’on puisse créer une simulation où l’on pouvait tester que deux robots puissent communiquer et interagir en dehors d’un match de football.

Le but ici a donc été de créer deux robots sur deux ports différents et de lancer automatiquement les deux comportements créés, *TCCollaborativeKicker* et *TCCollaborativeObserver* sur chacune des simulations. Le Behavior Viewer nous permet d’observer en temps réel les deux comportements et de vérifier que tout fonctionne correctement avant de lancer les programmes en réel.

Afin de vérifier que les informations transitent bien entre les deux joueurs, nous avons simulé la vision. Pour cela, nous lisons depuis le mouvement un fichier CSV contenant les positions où le robot kicker devrait être.

Par exemple, sur la figure 4.3, nous pouvons voir à gauche le robot kicker qui reçoit sa position vue par l’autre robot (cercle gris clair).



FIGURE 4.3 – Test en simulation à deux joueurs

4.3 Odométrie

Avant de tester les comportements à 1 joueur, il nous a paru essentiel de tester la fiabilité de l’odométrie (et cela nous a été fortement recommandé par le client).

Les premiers tests à l’odométrie ont donc été fait rapidement dès l’installation du terrain. En effet, le robot kicker étant aveugle, il ne peut pas se repérer grâce aux informations de sa vision mais seulement à l’odométrie.

Pour effectuer ces tests sur l’odométrie, nous avons utilisé le mouvement du **Placer**. Nous définissons une position cible en x et y puis nous réinitialisons la position du robot en $(0,0)$ dans le repère du terrain une fois que le robot a été placé manuellement au centre du terrain. En lançant le mouvement, le robot avançait alors jusqu’à la position demandée. Nous pouvions ensuite calculer l’écart à la position.

Lors des premiers tests, le Vive n’était pas encore installé. Nous avons dans un premier temps fait les tests en prenant des mesures manuelles entre la position du robot et la target donnée. Nous avons testé différentes positions sur les deux robots, des positions où il est possible qu’il se déplace lors du challenge. Plus tard, nous avons confirmé nos conjectures grâce aux tests effectués avec le Vive.

Par exemple, sur la figure 4.4, nous pouvons voir le tracé de l’approche du robot jusqu’au point target $(3,1)$ et la comparaison avec le relevé du Vive.

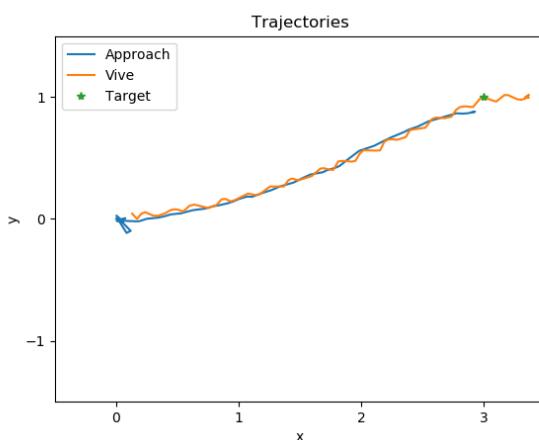


FIGURE 4.4 – Test d’odométrie

Nous pouvons voir un léger décalage au début – sans doute lié à l'imprécision du positionnement du robot au démarrage. Ensuite, nous voyons que la trajectoire du Vive oscille, c'est simplement car le robot alterne entre ses deux pieds et le tracker, placé sur sa tête, est donc impactée par la marche du robot.

Lors de nos tests manuels, nous avions remarqué que le robot était souvent trop avancé (une quinzaine de centimètres en moyenne) et quelques fois trop décalé à droite ou à gauche. Ici, nous voyons que malgré un placement plutôt correct par le robot (trajectoire bleue). Le Vive (en orange) nous confirme que celui-ci a largement dépassé le point à atteindre.

Également, faire des tests sur l'odométrie nous a permis de nous rendre compte à quel point le placement du robot au départ du challenge pouvait influencer le résultat. En effet, dans la figure 4.5a, le robot se place relativement proche de la position souhaitée (en (3,0)), toujours un peu devant.

En revanche, dans la figure 4.5b, le robot a été placé rapidement avec un décalage de quelques degrés (pas totalement droit face au cage). Finalement, sur 3m de marche en ligne droite, le décalage du kicker passe de quelques degrés à près de 20cm.

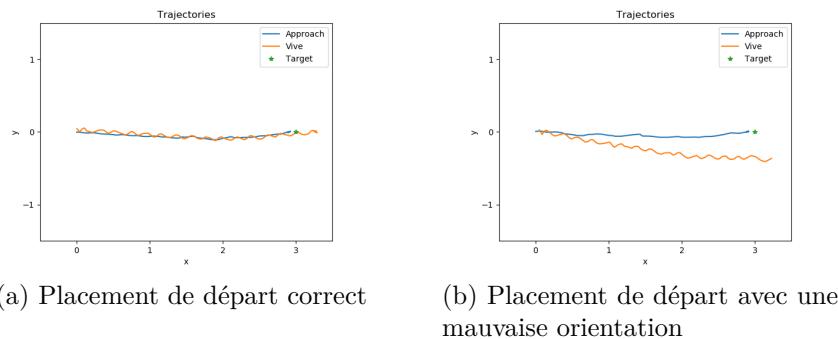


FIGURE 4.5 – Test d'odométrie – placement initial du robot

Suite à ce test, nous avons essayé de prévoir cette avancée trop importante, surtout pour la version du kicker utilisant le `Placer`. En effet, au lieu de faire avancer directement le robot jusqu'au ballon, nous procédons par étapes, lui faisant d'abord faire la moitié du chemin afin qu'il puisse se recaler grâce aux informations de l'observateur.

Il est important de noter que Rhoban s'est ensuite penché sur ces problèmes d'odométrie afin de la rendre plus fiable. Utilisant le Vive et le replay en simulation, le but est d'identifier les erreurs afin de minimiser ces problèmes sur l'odométrie. D'ici la réalisation du challenge en RoboCup, il est donc fortement possible que l'odométrie devienne plus fiable.

4.4 Détection des faux positifs

La base de notre PFE repose sur la détection de la balle et du robot sur le terrain. La vision n'est pas parfaite – et peut être que nos réglages sur le vert et la perception ne sont pas optimaux – donc il a semblé essentiel de regarder de plus près quels sont les problèmes récurrents.

Pour tester la vision, nous avons utilisé le script de `start_vision_log_machine` qui permet de lancer un robot sur le terrain pour un temps t de notre choix. Dans ce script, le robot se déplace et prend le temps d'observer son environnement, il est habituellement utilisé pour la prise de logs lorsque nous arrivons dans un nouveau terrain.

Nous avons posé l'autre robot et deux balles sur un demi-terrain et nous avons laissé le robot observateur se déplacer pendant deux minutes.

Une fois la log machine terminée, nous avons récupéré les logs sur le robot afin de les rejouer sur l'ordinateur (voir Annexe D). Nous pouvons alors prendre le temps d'étudier la vision afin de comprendre certains problèmes rencontrés.

4.4.1 La balle et les poteaux

Nous n'avons pas vraiment de problème avec la reconnaissance de ces deux éléments. La balle est placée devant l'observateur et les poteaux ne sont pas utilisés du tout dans notre PFE car ils servent pour la localisation (même pour le robot observateur, on fonctionne essentiellement à l'odométrie). Nous avions beaucoup de faux positifs sur les poteaux mais nous avons changé un paramètre sur l'impact du blanc dans la classification et maintenant les résultats sont davantage corrects.

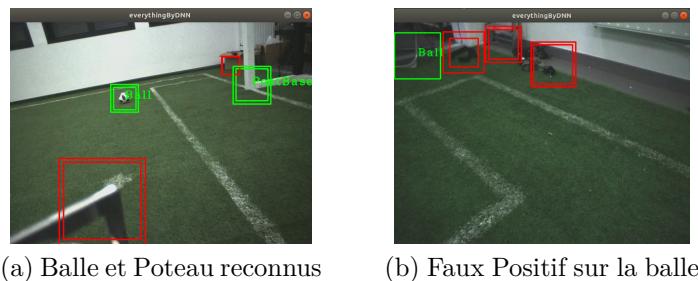


FIGURE 4.6 – Reconnaissance des objets du terrain – balle et poteaux

Nous voyons sur la figure 4.6a que la balle et les poteaux sont bien repérés. En revanche, le robot repère un faux positif sur une des chaises sur la figure 4.6b. Le robot garde la balle avec le score le plus important à la sortie du réseau de neurones. En présence d'une balle plus sûre (comme dans le challenge où le robot observateur a une balle devant lui), ce faux positif sera sûrement ignoré.

4.4.2 Les robots

À l'inverse, nous avons eu énormément de mauvaises reconnaissances de robot. Il y a souvent de nombreux opposants sur le terrain et nous devons donc filtrer quelle est la position la plus probable pour le placement du robot.

Les figures 4.7 nous montrent les trois cas majeurs où l'on retrouve des faux positifs : les humains, les valises et le dos du robot.

C'est le dernier cas qui est plus difficile pour nous. Il y a peu d'écart de position entre l'arrière du robot et le centre du robot, l'une apparaît juste un peu plus loin que l'autre.

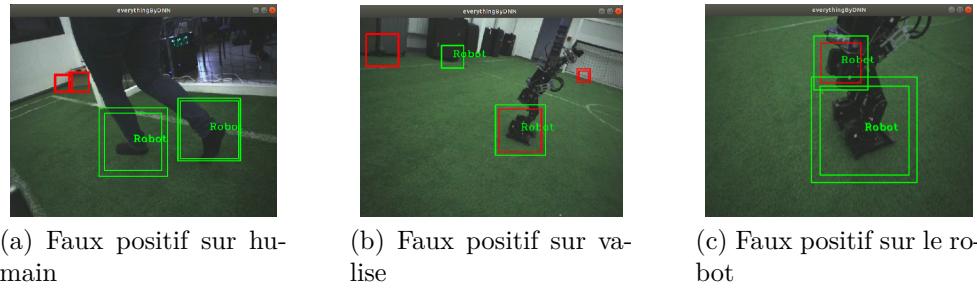


FIGURE 4.7 – Reconnaissance des objets du terrain – robot

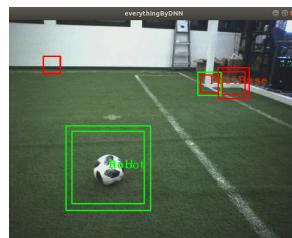


FIGURE 4.8 – Faux positif d'un robot sur la balle

Nous avons eu également une autre erreur pendant les vraies phases de jeu (mais pas pendant les tests) : voir un robot à la place de la balle 4.8. Dans ce cas là, sauf si le robot kicker arrive à revenir dans le champ de l'observateur, il est impossible de se sortir de cette situation.

Nombreuses de ces erreurs peuvent être dues au réseau de neurones qui n'a pas été ré-entraîné dans les nouveaux locaux. Le terrain et les robots restent identiques donc le changement reste mineur.

Nous ne pouvons pas changer grand chose de notre côté pour améliorer la vision. C'est juste que connaissant la cause des problèmes, nous pouvons comprendre pourquoi un essai a raté. Nous pouvons également essayer de filtrer l'information pour récupérer que ce qui nous intéresse. Par exemple, nous ignorons les faux positifs de robot qui sont loin du robot et de la balle.

4.5 Test robot seul

Comme vu section 1.3.1, le premier test réel du challenge à effectuer est de faire avancer le robot kicker seul sur le terrain.

Pour cela, nous avons créé un mode *alone*. Ce mode est actif tant que le robot kicker est capitaine. Comme expliqué précédemment, nous avons changé l'ordre du choix du capitaine afin que le robot observateur soit premier, dès qu'il se connecte il devient capitaine. On peut donc en déduire que le robot kicker est capitaine seulement s'il est seul.

Pour lancer le test sur le robot, nous avons donc simplement lancé le *TCCCollaborativeKicker* sur un robot. Dans le mode *alone*, la balle est automatiquement positionnée en (3,0). Les tests sur le robot seul restent donc relativement simple (le robot allant toujours tout droit).

Les résultats de ce test étaient binaires. Souvent, le robot jouant tout seul réussissait à marquer le but. En revanche, si le robot se décalait, il était alors impossible

de réussir le challenge.

Dans tous les cas, si nous observions un comportement relativement conforme à ce qui était attendu, nous lancions la phase de test suivante : le vrai challenge.

4.6 Test du challenge

La dernière phase de tests fonctionnels a été de tester les deux comportements en même temps sous forme de challenge.

Pour tester le challenge, nous n'avions qu'à lancer un mouvement sur chacun des robots. Également, pour enregistrer la vision, nous n'avons pas réussi à faire fonctionner la fonction `autologGames` et nous avons donc utilisé des logs manuels pour récupérer la vision (`logLocal`).

Grâce à l'enregistrement de la Vision, nous avons pu ensuite revisualiser les essais du challenge que nous avions effectué.

4.6.1 ApproachPotential

Nous avons donc eu deux mouvements à tester. En premier lieu nous avons donc testé le mouvement qui nous a semblé le plus prometteur. Nous avons donc fait une dizaine d'essais en plaçant la balle à différents endroits.³

Lors des réussites, le temps de l'essai est de moins d'une minute (environ 50 secondes), souvent avec un premier tir raté (à côté de la balle) puis un autre tir pour marquer le but. Le taux de réussite sur nos derniers tests n'est que de 60% mais nous avons essayé des positions difficiles de la balle (proche des poteaux). Nous avions déjà précédemment réussi à faire trois essais validés de suite comme demandé dans le challenge.

En revanche, si le robot ne marquait pas, nous avons pu remarquer différents problèmes.

Perte de vision de la balle

Le robot est passé devant la balle car il a trop avancé (voir test section 4.3) sans se relocaliser. Comme on le voit figure 4.9, le robot observateur n'arrive pas à repérer la balle derrière le robot car ils font partie de la même région d'intérêt reconnue comme un robot. Ici le robot ne bougeait pas et continuait à tirer car il se croyait toujours en bonne position. N'ayant pas d'informations sur la position de la balle, il gardait la dernière balle qu'il avait eu en mémoire, celle pour laquelle il était placé.

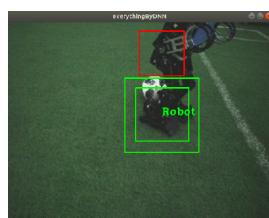


FIGURE 4.9 – Balle non repérée derrière un robot

3. Vidéos de nos tests collaborative_localisation – Google Photos : [lien](#)

Perte de vision du robot

Suite à un faux positif sur la balle (similaire à la figure 4.8), le robot observateur ne scannait pas assez large pour détecter l'autre robot (comme il en voyait déjà un sur la balle). Le robot kicker qui était loin de la balle et hors du champ de vision de l'observateur s'arrêtait alors pour tirer, se croyant sur la balle.

Autres

Le robot kicker est tombé en tirant (sûrement déstabilisé à cause du tracker du Vive posé sur sa tête). Nous avons alors décidé d'arrêter le challenge si le robot tombe car, la localisation a souvent du mal à être juste suite à une chute.

Également, le dernier échec est dû à un temps de jeu trop long. Le robot kicker a fait un premier tir mais en effleurant la balle sur le côté, celle-ci est donc partie vers la gauche. Le robot kicker a pu se repositionner derrière la balle mais il n'a pas tiré dans le temps imparti.

4.6.2 Placer

Le *TCCollaborativeKickerV2* est donc un système d'approche que l'on a développé nous-même, se basant sur le **Placer**. Il reste quelques problème à régler (notamment l'orientation du robot) ce qui le rend moins efficace que le mouvement de l'**ApproachPotential**.

Nous avons fait seulement 6 essais avec cette version du kicker mais en ayant 4 buts, nous avons donc 67% de succès. Ce mouvement semble plus efficace pour taper la balle : le robot a réussi par deux fois à se recaler alors qu'il était passé devant la balle. En revanche, il est possible qu'en se recalant il touche la balle ce qui complique l'accomplissement du challenge en moins de deux minutes.

Également, le problème de ce comportement représente la zone de tir. La position du robot par rapport à la balle reste moins optimale que celle définie dans l'**ApproachPotential** (nous n'avons pas fait des recherches autant approfondies que Rhoban pour définir le placement idéal du robot). Nous avons deux tirs où l'on remarque clairement que le robot est mal placé. Pour le premier le robot tape avec le coin de son kicker et la balle sort donc du terrain sur le côté. Le second est moins grave car le robot effleure juste la balle, celle-ci n'avançant que d'une trentaine de centimètres, le robot a pu marquer avec un second tir.

4.6.3 Résultats

Les tests nous montrent des bons résultats sur nos comportements mais surtout on peut clairement comprendre les problèmes qui font échouer le robot. Nous voyons que globalement le majeur problème vient de l'observation du robot kicker et de la balle en simultané. Nous pensons qu'une des solutions pour ce problème est peut-être de faire davantage bouger le robot observateur, par exemple le faire avancer derrière le robot kicker pour qu'il puisse mieux confirmer l'alignement mais un peu décalé afin qu'il observe tout de même la balle.

Également, il nous semble essentiel d'élargir un peu plus souvent l'angle du robot observateur, pas seulement quand il ne voit plus d'autre robot mais peut-être alterner les deux modes.

4.7 Tests sur les fichiers CSV

Afin de vérifier que les fichiers en entrée sont conformes, plusieurs tests ont été réalisés. Il s'agit notamment de s'assurer que l'extension du fichier et son contenu soient corrects.

- **Le ou les fichiers ne sont pas des CSV** : On vérifie que les quatre derniers caractères des fichiers sont bien ".csv". Si ce n'est pas le cas, alors un message informant l'utilisateur de la mauvaise extension est affiché et le programme s'arrête.
- **L'en-tête n'est pas conforme** : L'en-tête doit contenir les bonnes cellules. On peut alors vérifier au minimum si, dans la liste correspondant à la première ligne du fichier, il existe les chaînes de caractères "pos_robotX" ou "pos_robotY". Dans le cas contraire, on informe l'utilisateur que l'en-tête de son fichier d'entrée n'est pas bonne puis le programme se termine.
- **Le fichier contient peu de données** : L'objectif est de pouvoir tracer des courbes significatives pour étudier la trajectoire d'un robot. S'il y a peu de données alors il est très probable que le déplacement notoire. À l'aide d'une variable, on compte le nombre de lignes du fichier. S'il est inférieur à 20 alors un message d'avertissement avec le nombre de lignes est affiché. Le programme continue ensuite normalement.
- **Les fichiers ne sont pas appelés dans le bon ordre** (pour le fichier *comparePlots.py*) : Pour la cohérence entre la légende et le type de trajectoire, nos fichiers doivent être appelés dans le bon ordre : *approach* en premier et *position* en second. On vérifie alors si ces deux chaînes de caractères sont présentes dans chaque élément respectif de la liste des fichiers passés en argument.

4.7.1 Fichiers de tests

Nous avons ensuite crée des fichiers tests d'entrée afin de vérifier si nos tests fonctionnent :

- **not_csv_file.txt** qui contient la bonne en-tête et des valeurs correctes mais n'est pas de la bonne extension. Ainsi il permet de vérifier notre test concernant l'extension du fichier en entrée.
- **under_20lines.csv**, fichier correct mais de moins de vingt lignes de données qui génère un avertissement avec le test sur les fichiers courts.
- **wrong_header.csv** où les champs de position du robot ont été supprimés dans l'en-tête. Cette erreur est gérée par notre test qui lit l'en-tête

4.7.2 Crédit d'un script

Un script **script.sh** a été crée dans le dossier **src** pour faciliter l'utilisation du programme de traitement de donnée. Il permet de traiter deux fichiers CSV (diviser en plusieurs trajectoire) et de comparer leur première trajectoire automatiquement. Il est initialisé pour afficher un résultat du test d'odométrie.

Chapitre 5

Problèmes rencontrés et limites

5.1 Intégration au code

Comprendre le code existant a été un des grand challenge de ce projet. Nous avons du nous adapter et prendre le temps de découvrir le code. La partie traitement de test a été donc essentielle afin de pouvoir créer du code en dehors du code de Rhoban.

Aucun test automatique n'étant développé, il nous est impossible savoir si les modifications que l'on a faites dans le code commun impacte les performances en match. Il a été également compliqué d'en développer pour notre travail : nous n'avions pas de base et tester l'ensemble du code paraissait une tâche bien trop lourde.

5.2 Les tests

Nous avons eu l'avantage de travailler sur du matériel en réel mais cela nous a imposé également des contraintes de travail. En effet, suite au déménagement et aux conditions de fin de PFE, le terrain n'a été disponible que 3 semaines (dont une de vacances). Le temps de préparation du terrain, des robots, de calibration et l'utilisation par les membres de Rhoban pour leur travail (changement du code des robots) nous a forcément obligé à prendre davantage de temps pour les tests.

5.3 Les comportements

5.3.1 Le robot observateur

Le robot observateur remplit son objectif de transmettre les informations. Nous avons vu dans les tests des pistes d'amélioration à suivre. Il est facile de changer la largeur de scan dans le code.

Nous pensons que le problème majeur avec l'observateur est que nous n'avions aucune main sur la Vision. Peut être que si nous entraînions le robot à reconnaître la balle dans des situations où elle est davantage proche d'un robot, cela pourrait nous aider à mieux réussir le challenge. C'était en revanche intéressant de devoir s'adapter à un existant et en modifiant de notre côté les mouvements afin les résultats soient satisfaisants.

5.3.2 Le robot kicker

L'odométrie est un problème sérieux en robotique. Suite à notre travail, l'équipe de Rhoban a relancé des tests afin d'améliorer la précision de leur localisation. Je pense qu'empêcher ce dépassement de quelques centimètres de l'odométrie améliorerait grandement la réussite du challenge car c'est le problème le plus récurrent.

Au niveau de l'approche, les deux idées ont eu des résultats convenables. Il est vrai que le `Placer` aurait pu avoir de meilleurs résultats si on avait ajouté la résolution à d'autres problématiques comme le contournement de la balle ou l'orientation pour le tir (ici par défaut toujours vers la ligne de but adverse, pas forcément vers le centre des cages comme préféré dans l'`ApproachPotential`).

Le but du comportement au `Placer` était vraiment de simplifier au maximum l'approche pour voir si c'était réalisable lorsque l'`ApproachPotential` ne marchait pas à cause des problèmes d'orientation. Il aurait été absurde de vouloir recoder totalement une approche lors de ce PFE alors que c'est une problématique compliquée que même Rhoban a mis du temps à résoudre.

Avoir essayé de recréer une approche nous a revanche fait comprendre à quel point cette problématique est complexe et que comme beaucoup de choses en robotique qui paraissent simples, ce n'est souvent pas évident.

Une des choses que nous aurions voulu tester pour éviter le problème de la reconnaissance d'un robot dans une balle est de supprimer dans les `sharedOpponents` se trouvant à moins de 10cm du centre la balle (taille de son rayon). Pour l'ajouter au projet, il aurait été nécessaire de le tester. En effet, supprimer des `sharedOpponents` peut enlever le faux positif mais il est également possible qu'il enlève la vraie position d'un robot.

5.4 Limites du traitement des fichiers CSV

Le script `script.sh` permet à l'utilisateur d'avoir un exemple de traitement des trajectoires. Cependant, notre programme rencontre des limites si l'utilisateur souhaite traiter d'autres fichiers. En effet, celui-ci serait amené à écrire lui-même les commandes ou modifier le script. L'ajout de données (ou changement de l'en-tête) dans le CSV peut être effectué depuis le code de Rhoban. Tant que les colonnes nécessaires à l'affichage sont présentes, le programme fonctionnera mais les nouvelles données seront ignorées. En revanche, si l'utilisateur change les noms de colonnes ou s'il veut traiter les données ajoutées, il devra comprendre et modifier le travail effectué.

Également, il aurait été envisageable de créer une Interface Homme Machine (IHM) plus intuitive pour l'utilisateur. Cela n'a jamais été évoqué par le client donc nous avons préféré garder une utilisation en ligne de commandes pour pouvoir l'intégrer facilement à des scripts comme nous l'avons fait pour tester.

Chapitre 6

Conclusion

Notre projet a des résultats concrets qui représentent un bon signe pour la réussite du challenge lors de la prochaine RoboCup. Une fois sur place s'ajoutera également la pression de l'épreuve : seulement 25 minutes pour réussir 4 épreuves dont celle-ci, cela rend le challenge plus difficile que dans les locaux.

En dehors du challenge, nous avons fait des améliorations qui pourront être réutilisée pour les matchs. Par exemple, la hiérarchisation des capitaines selon un ordre choisi pourra permettre de choisir un robot en particulier pour mener le jeu. Le choix pertinent semble être le gardien de but car il a moins de chance de se faire sortir du terrain et cela permettra d'éviter les changements de capitaine en cours de match.

Nous espérons que le travail fourni sera utile à l'équipe par la réussite du challenge mais aussi car nous avons amené un regard nouveau sur le code, fait des remarques et soulevé des problématiques qui, comme nous avons vu pour le cas de l'odométrie, vont être étudiées par la suite.

Ce projet de fin d'étude s'inscrit pleinement dans la fin de notre formation de Master 2 ASPIC¹. Le travail que nous avons pu effectuer avec Rhoban nous a permis de comprendre dans un cas concret comment fonctionne le développement en robotique. Travailler sur du vrai matériel a représenté ici à la fois un challenge mais aussi une chance car nous n'en avions pas eu l'occasion auparavant.

Dans ce projet, nous avons pu parfaire nos connaissances en C++ et en Python, bases essentielles pour continuer dans la robotique (même si ROS², l'outil de développement le plus répandu en robotique, n'est pas utilisé ici, il réutilise ces deux mêmes langages). Ce PFE est un bon tremplin entre les études et le stage car tout le long du projet nous avons pu collaborer avec les chercheurs du LaBRI travaillant dans l'équipe. Nous avons eu la chance de pouvoir travailler dans leur locaux donnant un cadre et un rythme de travail pour avancer dans le projet et nous remercions toute l'équipe de Rhoban d'avoir partagé avec nous non seulement leur locaux mais aussi leur savoir et de nous avoir aidé tout au long de ce projet.

1. (Autonomous System : Perception, Interaction and Control)

2. <http://wiki.ros.org/ROS/Introduction>

Bibliographie

- [Allali et al., 2018] Allali, J., Fabre, R., Gondry, L., Hofer, L., Ly, O., N'Guyen, S., Passault, G., Pirrone, A., and Rouxel, Q. (2018). *Rhoban Football Club : RoboCup Humanoid Kid-Size 2017 Champion Team Paper*, pages 423–434.
- [Baltes et al., 2019] Baltes, J., Gerndt, R., Paetzl, M., Sadeghnejad, S., Farazi, H., Hofer, L., and Sattler, M. (13 June 2019). RoboCup Soccer Humanoid League : RoadMap. <http://www.robocuphumanoid.org/wp-content/uploads/RCHL-2019-Roadmap-Draft1.pdf>.
- [Burkhard et al., 2002] Burkhard, H., Duhaut, D., Fujita, M., Lima, P., Murphy, R., and Rojas, R. (07 August 2002). The road to RoboCup 2050. *IEEE Robotics & Automation Magazine*, 9(2) :31–38.
- [D.Gouaillier et al., 2009] D.Gouaillier, V.Hugel, P.Blazevic, C.Kilner, J.Monceaux, P.Lafourcade, B.Marnier, J.Serre, and B.Maisonnier (12-17 May 2009). Mechatronic design of NAO humanoid. In *2009 IEEE International Conference on Robotics and Automation*, pages 769–774.
- [Hsu, 1999] Hsu, F.-H. (1999). IBM's Deep Blue Chess grandmaster chips. *IEEE Micro*, 19(2) :70–81.
- [Kitano and Asada, 1999] Kitano, H. and Asada, M. (1999). The robocup humanoid challenge as the millennium challenge for advanced robotics. *Advanced Robotics*, 13 :723–736.
- [Ly et al., 2019] Ly, O., Allali, J., Gondry, L., Hofer, L., Laborde-Zubieta, P., N'Guyen, S., Pirrone, A., and Rouxel, Q. (2019). Rhoban Football Club - Robot Specification. <https://submission.robocuphumanoid.org/uploads/Rhoban-specs-5c05011864329.pdf>.
- [Rouxel et al., 2016] Rouxel, Q., Passault, G., Hofer, L., N'Guyen, S., and Ly, O. (2016). Learning the odometry on a small humanoid robot. pages 1810–1816.
- [Rouxel et al., 2015] Rouxel, Q., Passault, G., Hofer, L., N'Guyen, S., and Ly, O. (2015). Rhoban Hardware and Software Open Source Contributions for RoboCup Humanoids. In *Proceedings of 10th Workshop on Humanoid Soccer Robots, 15th IEEE-RAS International Conference on Humanoid Robots*.
- [Stevenson, 1997] Stevenson, S. (27 July-1 Aug. 1997). Mars Pathfinder Rover-Lewis Research Center technology experiments program. In *IECEC-97 Proceedings of the Thirty-Second Intersociety Energy Conversion Engineering Conference*, pages 722–727, Honolulu, HI, USA, USA.

Annexe A

Localisation sur le terrain

Les coordonnées des robots sur le terrain sont définies par un repère ayant pour origine le centre du terrain. L'axe des abscisses du terrain est défini en partant du centre du terrain jusqu'au centre des cages situées à droite du terrain. L'axe des ordonnées part du centre jusqu'à la ligne de touche opposée à la table d'arbitrage.

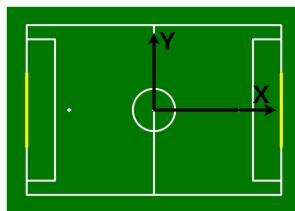


FIGURE A.1 – Axes du terrain

Dans le cas du challenge, on assume que le robot avancera toujours vers les cages situées devant lui, lorsque x augmente. Les positions des robots sont toujours données sous forme d'un vecteur à trois coordonnées : x, y, θ . L'orientation du robot est égale à 0 lorsque le robot regarde en direction de l'axe x .

La figure A.2 nous montre les trois positions essentielles pour ce PFE. La position 1 est celle du robot kicker à l'initialisation $(0,0,0)$. L'observateur a été testé aux positions 2 : $(3, -1.75, 90)$ et 3 : $(4.5, 0, 180)$.

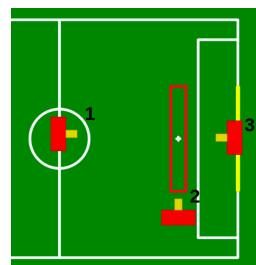


FIGURE A.2 – Positions notables des joueurs sur le terrain

Annexe B

La Vision

B.1 Détection du vert

À partir d'une caméra Point Grey Blackfly GigE [Ly et al., 2019], le robot récupère des images en YUV, immédiatement changées en RGB. Pour expliquer comment fonctionne la Vision et les différents filtres, nous allons nous baser sur un environnement simple (Figure B.1) avec un robot et une balle, éléments majeurs pour notre PFE.

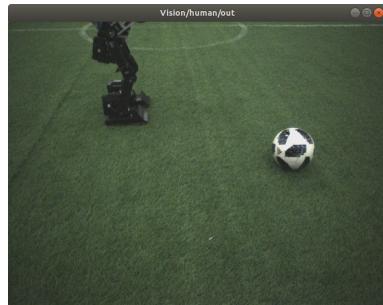


FIGURE B.1 – Image RGB perçue par le robot

Ensuite vient la calibration du vert. Grâce à RhIO , nous pouvons paramétriser la caméra pour avoir la meilleure perception du vert possible. (voir section 4.1). À chaque condition lumineuse, nous devons changer les paramètres servant à repérer le vert. Le but quand on calibre et d'avoir un terrain blanc et homogène et d'avoir les lignes du terrain qui ressortent en noir.

Sur la figure B.2a, nous pouvons voir les paramètres dans les conditions optimales : les stores sont fermés et le terrain est éclairé de façon homogène par un éclairage artificiel au plafond.

En lumière naturelle, la luminosité dépend de la météo. Malgré une image testée en milieu d'après midi (15h30), le temps était très nuageux ce jour-là et nous voyons clairement le manque de luminosité sur la figure B.2b, prise avec l'éclairage éteint et les stores ouverts.

Au contraire, un excès de luminosité peut être néfaste pour la vision. En effet, la figure B.2c a été prise avec les stores ouverts et les lumières allumées. La vision est moins efficace car le vert devient blanc.

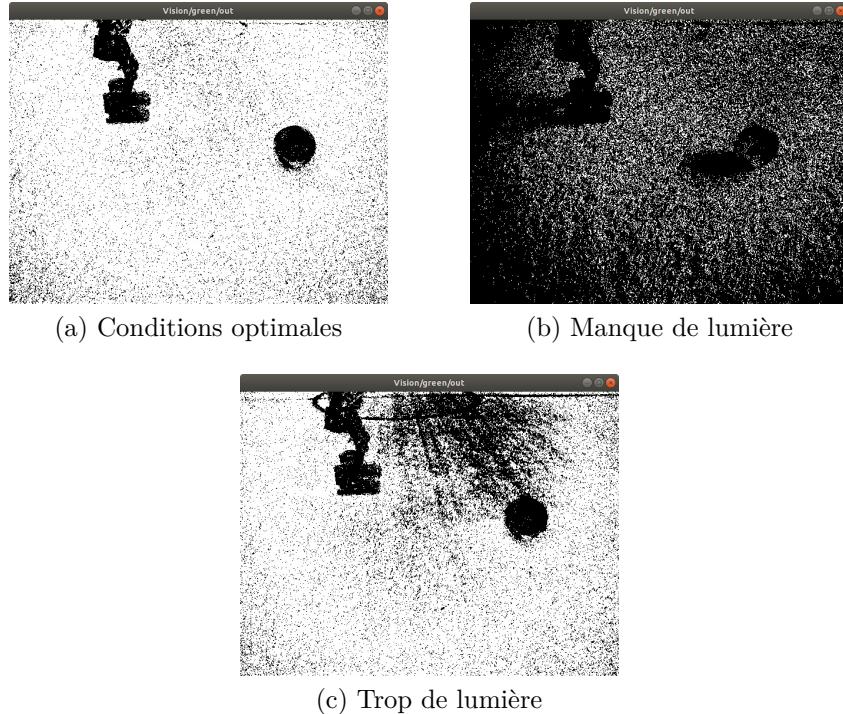


FIGURE B.2 – Impact de la luminosité sur la détection du vert

B.2 Détection des éléments du terrain

Une fois le vert bien paramétré, Rhoban utilise un algorithme des images intégrales qui calcule des régions d'intérêts en repérant ce qui est non-vert dans le terrain. Le filtre par Image Intégrale de la figure B.3a nous permet de voir que le robot et la balle sont bien reconnus dans le terrain.

Un réseau de neurones va ensuite catégoriser ces régions d'intérêts en fonction des éléments qu'il peut reconnaître. Pour l'instant le robot reconnaît essentiellement la balle, le robot et les poteaux des cages. Chaque carré représente une image passée dans le réseau de neurone. Le carré est rouge si le réseau ne reconnaît rien ou que le résultat obtenu est trop faible. Sur la figure B.3b, le robot reconnaît parfaitement la balle mais n'est pas très sûr pour le robot. L'algorithme est détaillé dans leur article suite à leur victoire en 2017 [Allali et al., 2018], section 4.

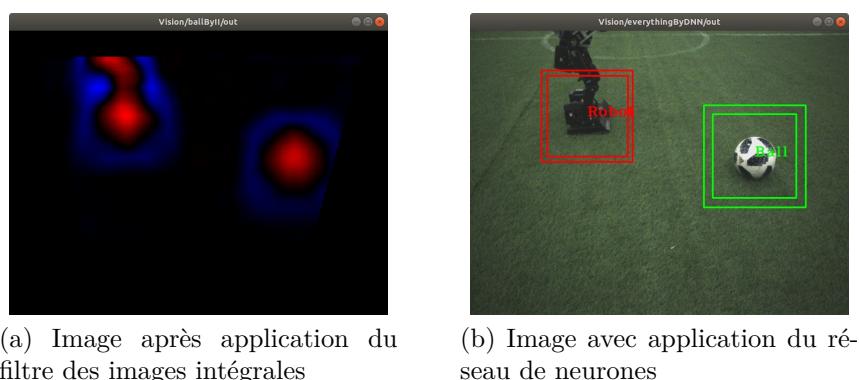


FIGURE B.3 – Reconnaissance des objets du terrain

B.3 Image synthétisée

À la fin du traitement, l'image finale est une image tagguée comme sur la figure B.4. Nous pouvons voir alors un point bleu marquant le centre de la balle et un point rose marquant la position située entre les deux pieds du robot.



FIGURE B.4 – Rendu de l'image après traitement

Annexe C

La création d'un mouvement

L'implémentation des **Moves** est déjà prédéfinie dans le code de Rhoban et se présente sous la forme de :

- **OnStart** : action effectuée au démarrage ;
- **OnStop** : action effectuée à la fin ;
- **step** : action effectuée à chaque tic d'horloge du mouvement.

OnStart nous sert souvent à initialiser l'état du robot pour le mouvement et OnStop pour rétablir l'état avant OnStart.

Les **Moves** englobent toutes les actions que va faire le robot. Les mouvements simples permettent d'activer certaines parties du corps (**Head**, **Arms**) ou d'effectuer une action (**Kick**, **StandUp**) par exemple.

Enfin, il existe des **STM** (state machine) qui s'apparentent davantage à des comportements comme (**GoalKeeper**, **Playing**, **Robocup**) qui vont utiliser d'autres mouvements pour créer une stratégie de jeu. Les comportements développés lors de notre PFE sont des **STM**. Étant créés pour un challenge technique, leurs noms commencent par **TC** (technical challenge).

Rhoban a créé une implémentation et comme pour les mouvements, nous avons deux fonctions **enterState** et **exitState** qui nous permettent d'agir à l'entrée et à la sortie de chaque état.

Annexe D

Simulation

D.1 Créer des robots en simulation

Il existait un script permettant de lancer plusieurs simulations sur des ports différents : `run_fake_team.sh`. Pour utiliser cela nous avons donc du comprendre comment se connecter à ces différents ports ainsi que la procédure de simulation.

1. Depuis le dossier `fake`, lancement de la simulation du robot : `./KidSize`
 - `-n` : noVision mode, sinon on doit donner des anciennes vidéos pour simuler la vision.
 - `-p nb_port` : permet d'en lancer plusieurs en choisissant un port.
2. Lancement de l'interface de contrôle du robot : `rhiostatic`. Si un port a été spécifié, `rhiostatic localhost:nb_port`
3. Lancement du `BehaviorViewer` pour voir le robot en 2D sur le terrain : `behav` (On assume un alias pour aller chercher `workspace/bin/BehaviorViewer`). Si un port a été spécifié `behav localhost nb_port`

D.2 Rejouer un log de vision

Pour lancer la Vision, il est nécessaire d'avoir un ancien log des robots, un match par exemple. Il est nécessaire de préparer plusieurs fichiers.

- `vision_config.json` : `ln -sf ..//common/vision_filters/all_fake.json vision_config.json`
- `calibration.json` : `ln -sf lien_vers_le_dossier_de_log/calibration.json calibration.json`
- `workingLog` : `ln -sf lien_vers_le_dossier_de_log/ workingLog`

Le robot rejouera le log choisi en adaptant la vision avec la configuration actuelle.

On peut également