RAPPORT DE MODAL

PHY43M06EP: Robotique

FRAMBOURT MATEÏS

TABLE DES MATIÈRES

INTRODUCTION

Je suis arrivé dans ce modal un peu par hasard. Au départ je m'étais inscrit pour le ModAL FPGA car c'était quelque chose qui me semble très versatile mais un peu obscur pour le découvrir en autodidacte. C'est donc avec surprise que j'apprends la disparition de ce ModAL dans une fusion avec le ModAL de Robotique. Finalement, ça ne m'a pas trop gêné car c'était mon deuxième vœu. J'avais une légère expérience en robotique. En effet, j'ai participé de 2012 à 2021 aux compétitions de robotique junior (robot radioguidés) de planètes sciences et j'aide encore marginalement dans l'équipe familiale. J'avais donc une certaine connaissance des composants et de leurs utilisations mais je ne connaissais presque rien dans le contrôle et la commande haut niveau. Mon objectif dans ce cours était donc d'apprendre cette partie pour pouvoir ensuite l'appliquer au Binet X-Robot qui est en manque crucial de RH et de compétence.
D'un point de vue plus académiques, l'objectif final est de réaliser un robot « tank » pour pouvoir travailler sur l'anticipation de mouvement de l'adversaire. L'objectif était donc de réussir à pointer une cible sur l'adversaire tout en se déplaçant dans un environnement encombré. Pour cela nous avions à notre disposition :

- Roues holonomes
- Moteurs avec encodeurs
- Raspberry pi 4
- Cartes moteurs I²C Groves
- Caméra grand angle 5MP
- Servomoteurs

Cependant, avant de réaliser le projet final il est nécessaire de réaliser un suivi de ligne fonctionnel. Dans ce sens j'ai passé (malheureusement) la majeure partie des séances (8/10) sur ce projet initial.

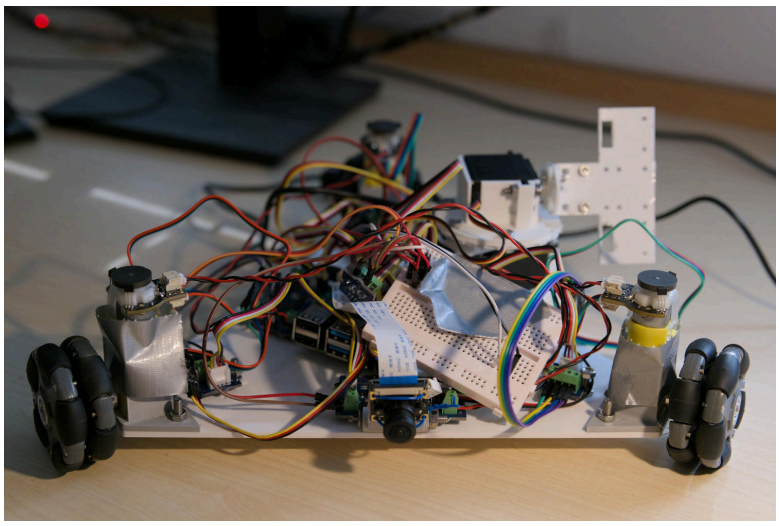

Fig. 1. – Vue d'ensemble du robot final

1
BASE COMMUNE

1.1 - Base mécanique

Dans l'optique de pouvoir réutiliser un maximum de composants (tant mécaniques que logiciels), J'ai commencé par modéliser précisément les composants à ma disposition. Ensuite j'ai dessiné les supports des moteurs pour avoir une emprise au sol minimale (ils ont ensuite été imprimés au DrahiX). Enfin j'ai dessiné la plaque principale qui devait permettre d'avoir tous les composants à plats avec l'ensemble des trous déjà percés (elle a été découpée au laser du DrahiX). Cependant, la breadboard et la batterie ont été oubliées ce qui a mené à un montage relativement complexe à suivre et à modifier. L'objectifs n'a donc pas complètement été rempli.
J'ai choisi d'utiliser 3 moteurs en triangles pour 2 raisons :

- Il y a 3 degrés de liberté sur le robot donc avoir 3 moteurs pourrait donc nous permettre de facilement faire de la cinématique inverse exacte (ce n'est pas la méthode qui a finalement été utilisée)
- Les placers à 120° permets d'avoir une symétrie et donc on peut avoir plusieurs « face avant » du robot et il suffit juste de faire un décalage au niveau des consignes pour utiliser par exemple un autre outil

D'autre part, la caméra est montée sur un pivot car elle possède un grand angle de vue. Il est donc possible de régler l'étendue du champ de vision en inclinant la caméra pour qu'elle ne soit pas perturbée par des objets extérieurs.

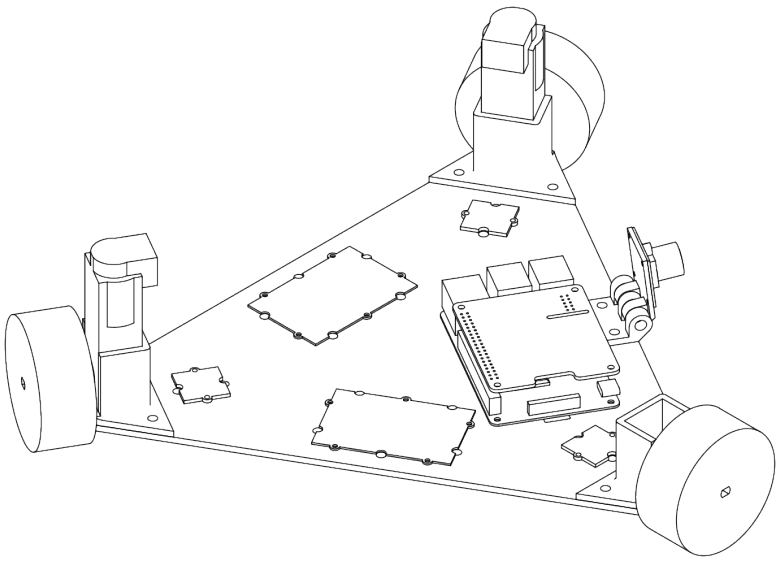

Fig. 2. – Vue isométrique de l'assemblage basique

1.2 - Encodeurs

Rapidement, les encodeurs sont des paires de capteur à effet hall en quadrature de phase autour d'un aimant à 14 pôles radiaux de manières à ce qu'ils donnent 7 périodes de la Fig. 3 par rotation. Pour interpréter le signal, j'ai fait une librairie CMake qui attache une interruption aux 2 pins de l'encodeur et appelle une fonction très simple :

```
uint8_t phase = identifier_phase();
if (phase == (lastPhase + 1) % 4)
{
    pos++;
}
else if (phase == (lastPhase - 1) % 4)
{
    pos--;
}
lastPhase = phase;
```

et pour identifier la phase on fait appel à une table de correspondance :

```
uint8_t const LUT[] = {0, 1, 3, 2};
return LUT[2 * digitalRead(pinA) + digitalRead(pinB)];
```

J'ai essayé pendant plusieurs séances d'implémenter cette librairie de manière conforme à la POO mais je n'arrivais pas à faire une fonction statique qui appelle une méthode. Finalement, j'ai réussi à l'implémenter pour une autre librairie dont je parlerais plus loin.
Globalement, le problème principal de cette mesure est qu'elle donne précisément la position mais pas la vitesse. Il faut ensuite dériver numériquement ce qui ne donne pas de très bon résultat car la position est une fonction en escalier. Or je souhaitais asservir les moteurs en vitesse ce qui rend l'asservissement un peu instable.

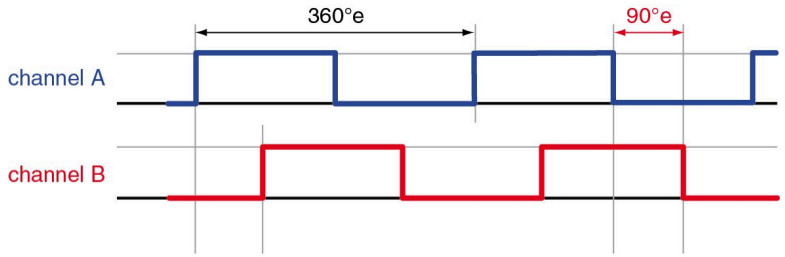

Fig. 3. – Signal en sortie des encodeurs (© Maxon)

1.3 - PID

J'ai ensuite créé une librairie CMake générique pour les PID, c'est-à-dire un contrôleur générique à trois composante (proportionnelle, intégrale, dérivée). Celui-ci devait permettre d'avoir un contrôle précis et réactif mais les premiers moteurs était trop peu puissant et ne reposait presque que sur la composante intégrale. En effet, même en réglant la composante proportionnelle à la limite de l'instabilité, la valeur atteinte

en vitesse était très éloignée de la valeur désirée. D'autre part, l'entrée du moteur étant proportionnelle à sa vitesse, il fallait « intégrer » la commande envoyée au moteur. Dans le cas contraire, dès que l'erreur est nulle, la consigne envoyée au moteur est nulle et donc le moteur s'arrête. Ce n'est bien sûr pas le comportement voulu. Dans ce sens, l'ajout d'une commande constante au moteur permet au PID de compenser l'erreur que celle-ci aurait seule. (On approxime donc le moteur à une réponse linéaire à la consigne et on corrige l'erreur d'approximation par PID). Cependant, même avec toute ces modifications, le moteur ne répondait pas correctement à faible commande. En effet, les effets de frictions sèches étaient trop importants. Le modèle a donc été adapté pour les prendre en comptes (en augmentant la consigne artificiellement à faible commande). Pour limiter l'inertie de la composante intégrale, l'intégrale sature à partir d'une limite arbitraire.

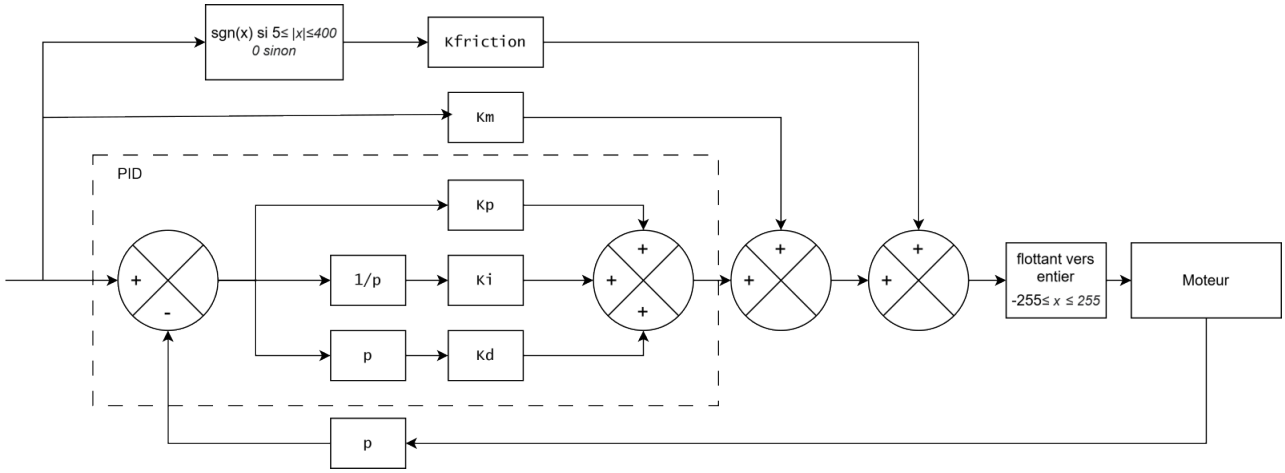

Fig. 4. – Asservissement de chaque moteur

1.4 - Roues holonomes

Les roues holonomes sont des roues qui n'imposent pas de contrainte normale à la roue. C'est dire qu'elle glisse sur la direction normale à la roue. On peut donc considérer uniquement comme un ajout de vecteur (il faut cependant prendre en compte que chaque roue adhère sur la direction tangentielle). Le robot est en liaison équivalente à une liaison plane, il y a donc 3 degrés de libertés or on a 3 moteurs donc on pourrait écrire les équations directe du mouvement pour ensuite tenter de les inverser. Cependant ici, vu le faible nombre de moteur, j'ai choisi d'exhiber 3 vecteurs de bases du mouvement (Fig. 5). Par la suite, il est donc possible de partir directement des vecteurs du mouvement désirés. Pour autant, les moteurs ont une valeur maximale de vitesse de rotation possible. Il faut donc limiter le vecteur de commande de sorte que chacune de ses composantes reste inférieures à la limite. En pratique, j'ai réalisé la transformation avec des flottants $\in [-1, 1]$. J'ai exploré 3 méthodes :

- Normaliser : $\boldsymbol{t_{out}} = \frac{\boldsymbol{t_{in}}}{\|\boldsymbol{t_{in}}\|}$ on a donc $\|\boldsymbol{t_{out}}\| = 1$ donc toute les composantes $\in [-1, 1]$
- Tronquer : pour tout composante $t_{in} \in \boldsymbol{t_{in}}$, $t_{out} = \min(\max(t_{in}, -1), 1)$ donc $-1 \leq t_{out} \leq 1$
- Mettre à l'échelle : soit $t_{max} \in \boldsymbol{t_{in}}$ tel que $\forall t \in t_{in}, |t_{max}| \geq |t|$ alors $\boldsymbol{t_{out}} = \frac{\boldsymbol{t_{in}}}{|t_{max}|}$

L'avantage de tronquer, c'est que le calcul est extrêmement rapide mais on perd de la direction originelle du vecteur. La normalisation permet de conserver la direction mais réduit plus les composantes que nécessaire

(on est dans la boule unitaire) alors que mettre à l'échelle permet de conserver la direction (en étant dans le cube unité complet). Au final, l'algorithme choisi est donc :

```
float avant[3] = {0, -1, 1};
float droite[3] = {-1, 1 / sqrt2, 1 / sqrt2};
float rotation[3] = {1, 1, 1};
scale(input);
for (int i = 0; i < 3; i++)
{
    output[i] = avant[i] * input[0] + droite[i] * input[1] + rotation[i] * input[2];
}
scale(output);
```

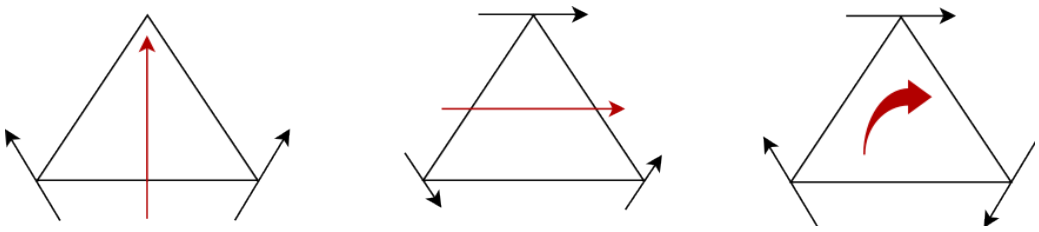

Fig. 5. – Vecteurs considérés pour les roues holonomes

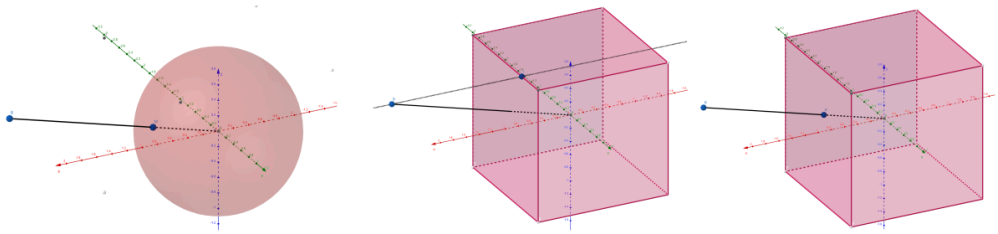

Fig. 6. – Les différentes méthodes de projections explorées

1.5 - Carte Moteur

Cette section est courte car l'utilisation est relativement simple mais a demandé du temps de calibration. En effet, les cartes reçoivents 2 bytes pour les vitesses et 1 bytes pour les directions. Cependant, les valeurs semblent aléatoires :

```
uint8_t DirLut[4] = {0x0a,0x06,0x09,0x05};
wiringPiI2CWriteReg16(this->m_fd,0xaa,DirLut[(dirA+2*dirB)]);
```

et pour la vitesse :

```
this->setDirection(this->_M1_direction,this->_M2_direction);
wiringPiI2CWriteReg16(this->m_fd,0x82,((uint16_t)this->_speed1)<<8|this->_speed2);
```

On peut donc remarquer que c'est ici qu'intervient la limitation à 8 bit par moteur qui était affiché sur la Fig. 4

2
ROBOT SUIVEUR

2.1 - Estimation du chemin

Le chemin est un scotch rouge ou bleu faisant un cycle sans embranchement. On peut donc essayer d'estimer le chemin en ne prenant que le barycentre des pixels de la bonne couleur. Dans un premier temps j'ai essayé de prendre le barycentre bleu et rouge sur toute l'image mais le résultat était trop influencé par les pixels proches de la caméra et le robot oscillait. J'ai donc séparé l'image en 2, une partie supérieure et inférieure sur lesquels on calcule le barycentre des 2 couleurs. Pour cela, on retourne l'image et on convertit l'image de RGB vers HSV pour avoir accès à la teinte qui est plus facile à interpréter. Ensuite, on filtre en ne gardant que les pixels entre 2 bornes. Enfin on calcule les moments pour enfin avoir les barycentres. Les moments sont $M_{\text{ij}} = \sum_{p\in \text{ image}}(x^i y^j p)$ donc $\mathbf{bary} = \begin{pmatrix} \frac{M_{10}}{M_{00}} \\ \frac{M_{01}}{M_{00}} \end{pmatrix}$

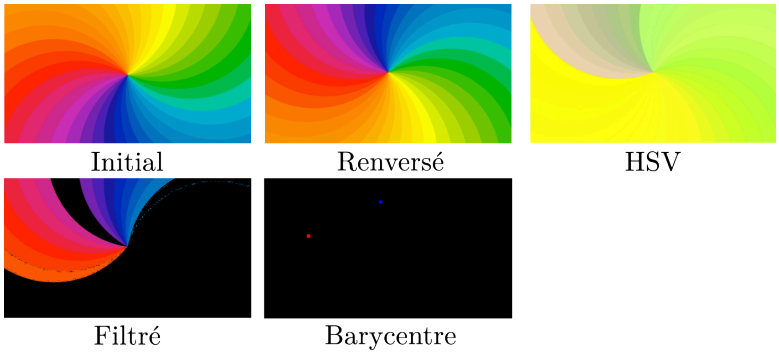

Fig. 7. – Filtre appliqué

2.2 - Choix entre bleu et rouge

Il faut ensuite choisir s'il faut suivre les barycentres bleus et rouges. Pour cela, j'ai considéré qu'on pouvait estimer comme fiable s'il y a plus de 2% de pixel de l'image sont de cette couleurs. Il est nécessaire d'avoir un hystérésis pour ne pas être sensible à un verrouillage bref d'un obstacle. Finalement, j'ai utilisé une machine à 3 états : Soit on suit le rouge, soit le bleu soit le robot est perdu.

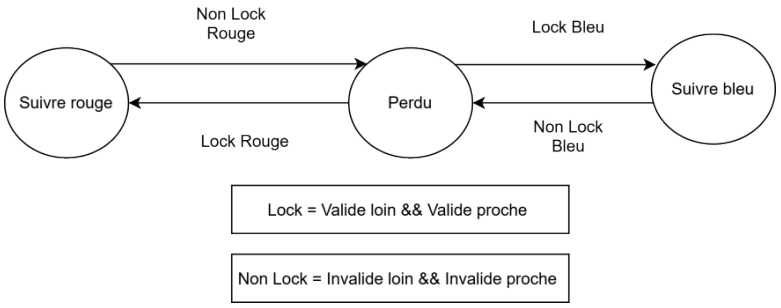

Fig. 8. – Logique de choix des couleurs

2.3 - Loi de commande

Le robot a maintenant 2 barycentres sélectionnés qu'il doit suivre. Il y a 3 degrés de libertés à commander. J'ai choisi de mettre directement la commande « avancer » à 0.5 pour donner une direction globale. Si le robot est perdu il tourne en alternant le sens de rotation pour parcourir un arc de plus en plus grand pour trouver le chemin le plus proche de la direction initiale. Dans le cas contraire, on vérifie si on un verrouillage proche alors on prend en compte ce barycentre et on applique un PID pour maintenir le barycentre au centre horizontalement. D'autre part, s'il y a un verrouillage loin valide, alors on le prend en compte et on calcule l'angle entre le centre du bord bas de l'image et le barycentre avec l'horizontale. On applique alors un PID sur l'angle pour le maintenir à 90°. Pour résumer en un schéma :

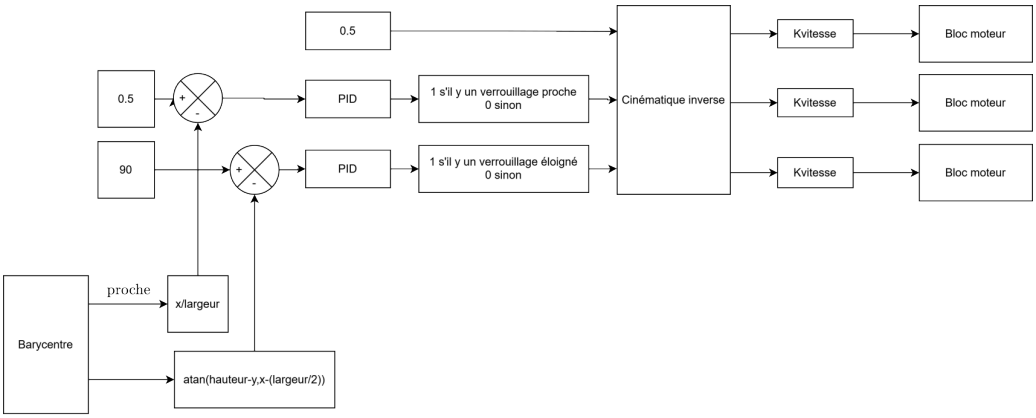

Fig. 9. – Schéma de la loi de commande

3
ROBOT TANK

3.1 - Ajout d'une tourelle

Dans un premier temps, j'ai ajouté une tourelle avec 2 axes (θ, φ) à partir de 2 servomoteurs. Les servomoteurs acceptent une chaine d'impulsion de Hz avec un temps à l'état haut $\in [250, 2000]$ µs. Pour réaliser ces signaux, j'ai utilisé 2 threads avec un « niceness » de -50 pour ne pas être trop perturbé par le temps de calcul des autres threads. Je n'ai pas eu le temps d'aboutir cette tourelle mais j'aurais voulu réaliser un système de limite de position. En effet, le bras est suffisamment long pour qu'il existe des positions avec une collision entre la tourelle et les moteurs. Pour cela j'aurais premièrement approximé l'espace admissible par un rectangle alors il suffit de borner les valeurs acceptées pour projeter la position dans l'espace admissible. Ensuite j'aurais voulu raffiner en définissant l'espace admissible comme un polygone puis en projetant sur le point le proche dans le polygone [1]. Il aurait ensuite été possible (mais sûrement trop long et trop peu rentable en termes de temps) de le définir par une courbe paramétrique et d'appliquer la méthode de l'article [2]. D'autre part, j'ai fait une erreur dans la conception de la tourelle, je n'ai pas mis la caméra sur l'axe de rotation des 2 servomoteurs, le centre de la caméra se déplace donc quand le moteur φ tourne ce qui ne permet pas de supposer la caméra comme immobile avec un large angle de vue. Il faudrait donc compenser cette erreur dans les parties suivantes.

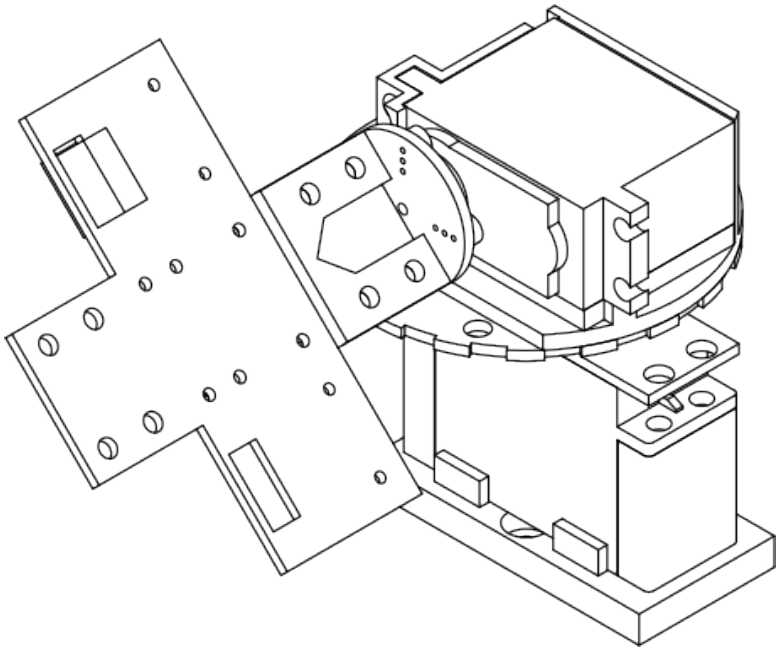

Fig. 10. – Vue isométrique de la tourelle

3.2 - Stratégie envisagée

3.2.1 • Perception et mapping de l'environnement

Pour percevoir l'environnement, j'ai essayé de rajouter 2 VL53L1 [3] de STMicroelectronic (que je n'ai pas eu le temps de les utiliser en pratique) qui sont des LiDAR pouvant donner jusqu'à 16 point par mesure à une fréquence 30 Hz (mais à une précision réduite variant de $\pm 40\%$ à $\pm 4.5\%$ en fonction du mode choisi et de la distance). Je pensais faire la reconnaissance de l'environnement en 2 étapes. La première serait d'utiliser des marqueurs ArUco sur chaque coins des boites en prenant soin d'utiliser toujours les mêmes marqueurs pour les mêmes coins. Ainsi en utilisant uniquement la caméra il aurait été possible de savoir approximativement l'environnement dans lequel il évolue. Le problème étant donc de préparer l'environnement en avances. L'autre avantage de cette méthode est qu'il n'est pas nécessaire de cartographier l'environnement dans le référentiel terrestre mais uniquement dans celui du robot car l'information lui est disponible de manière sûre à chaque image. L'autre méthode est celle référencé dans cet article [4]. J'aurais donc essayé d'implémenter la méthode de grille en prenant une grille de taille fixe. Je voulais essayer de modéliser l'incertitude de mesure en prenant les mesures comme des boules (dans la distance de Manhattan) en répartissant (uniformément) 1 point sur les cases affectées. Alors on peut définir un seuil de point par case à partir duquel on considère que celle-ci est occupée par un obstacle. Il faut ensuite retirer les anciens point, on aurait donc pu faire un sigmoïde renversée sur chaque points. On peut ensuite faire un A^* vers l'objectif visé. Pour avoir un algorithme cohérent, il faut prendre en compte le déplacement du robot et donc tourner et translater les points de manière opposée au robot. En effet, il est intéressant de rester dans le référentiel du robot pour pouvoir avoir une grille très dense sans occuper trop de mémoire.

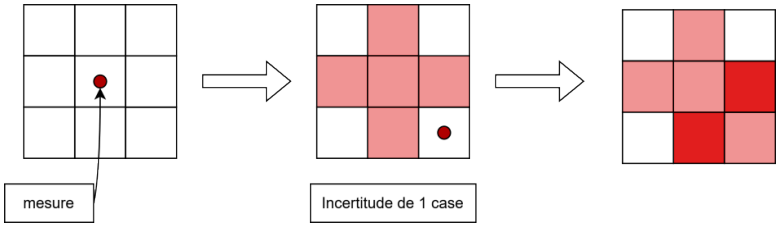

Fig. 11. – Algorithme de remplissage de la grille

3.2.2 • Détection de l'adversaire et anticipation

Le robot est donc sensé savoir où il se trouve et comment atteindre des objectifs arbitraires. Il faut maintenant trouver l'adversaire. Pour cela la solution la plus simple est de lui fixer un marqueur ArUco pour pouvoir le repérer avec la caméra. On peut ensuite mesurer la distance avec l'un des LiDAR et enfin on peut simplement supposer sa vitesse sera constante. Il y a alors plusieurs stratégies possible :

- Le suivre et en cas de perte visuelle aller à a la dernière position connue de la cible (en utilisant le modèle du terrain élaboré précédemment)
- Estimer la position de réapparition, en utilisant le modèle d'environnement, on peut supposer que la cible restera sur la bordure de l'obstacle et il suffit donc de parcourir la frontière estimée de l'obstacle jusqu'à arriver sur une face visible. On a alors une estimation du temps et de la position où l'adversaire peut venir.

- Adopter le mouvement de l'adversaire. Il serait aussi envisageable de suivre le mouvement de l'adversaire en miroir mais dans ce cas, les obstacles ne sont pas directement pris en compte

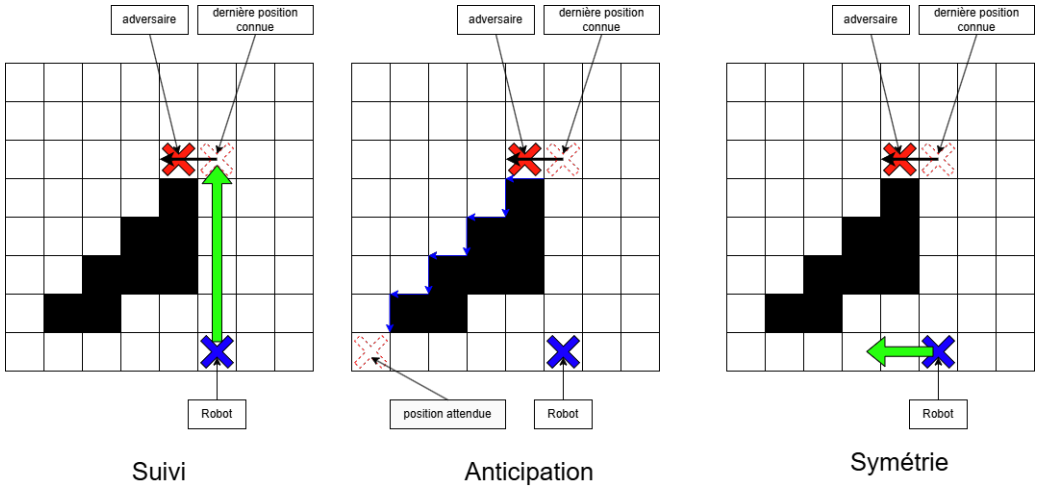

Fig. 12. – Les différentes stratégies proposées

3.2.3 • Suivi de la cible et « tir »

Une fois la cible en vue, le robot doit être capable de tirer avec précision sur la cible. Pour cela, je souhaitais réutiliser les PID, un par axe. La partie plus complexe est que la tourelle des capteurs est la même que celle de tir. Il n'est donc pas possible de cartographier l'environnement en même temps que le ciblage et le tir sur la cible. la seule particularité de cette partie est qu'il est nécessaire de savoir la vitesse du projectile (instantané pour le laser ou la photo mais finie pour un projectile d'airsoft) pour pouvoir anticiper le point de contact du projectile.

CONCLUSION

En conclusion, Ce projet a permis de réaliser un robot suiveur de ligne relativement performant avec des contraintes matérielles (particulièrement sur les moteurs qui n'avaient pas assez de couple pour pouvoir convenablement déplacer le robot) et de commencer la conception d'un robot tank plus polyvalent et plus haut niveau dans sa logique de commande. Cependant, un retard assez important a été accumulé sur partie « suiveur » qui m'a donc empêché de réaliser le robot tank. Ainsi, même si j'aurais tout de même préféré découvrir les FPGAs, je pense que ce modal m'a permis de toucher du doigt un aspect plus complet et plus haut niveau de la robotique que je pourrais ensuite appliquer sur les (potentiels) robot de X-Robot. Je pense qu'il pourrait peut-être être intéressant pour les futurs ModALs de commencer directement avec une base roulante et donc se rapprocher d'une exploration scientifique. Je pense en effet qu'il est intéressant de découvrir tous les composants pour savoir en profondeurs comment son robot fonctionne mais j'ai trouvé qu'il était particulièrement dur d'appliquer des articles de recherche car la majeur partie du temps est passée sur la base roulante.

BIBLIOGRAPHIE

[1] J. Ali, « Get Closest Point on a Polygon ». [En ligne]. Disponible sur: https://javedali-iitkgp.medium.com/get-closest-point-on-a-polygon-23b68e26a33

[2] J. Xu, W. Liu, H. Bian, et L. Li, « Accurate and Efficient Algorithm for the Closest Point on a Parametric Curve », 2008, p. 1000-1002. doi: 10.1109/CSSE.2008.618.

[3] « Time-of-Flight long-distance ranging sensor with advanced multizone and multiobject detection ». [En ligne]. Disponible sur: https://www.st.com/resource/en/datasheet/vl53l1.pdf

[4] L. Liu, X. Wang, X. Yang, H. Liu, J. Li, et P. Wang, « Path planning techniques for mobile robots: Review and prospect », *Expert Systems with Applications*, vol. 227, p. 120254, 2023, doi: https://doi.org/10.1016/j.eswa.2023.120254.

[5] « Dépot GitHub du Modal ». [En ligne]. Disponible sur: https://github.com/Docredstein/ModalRobot