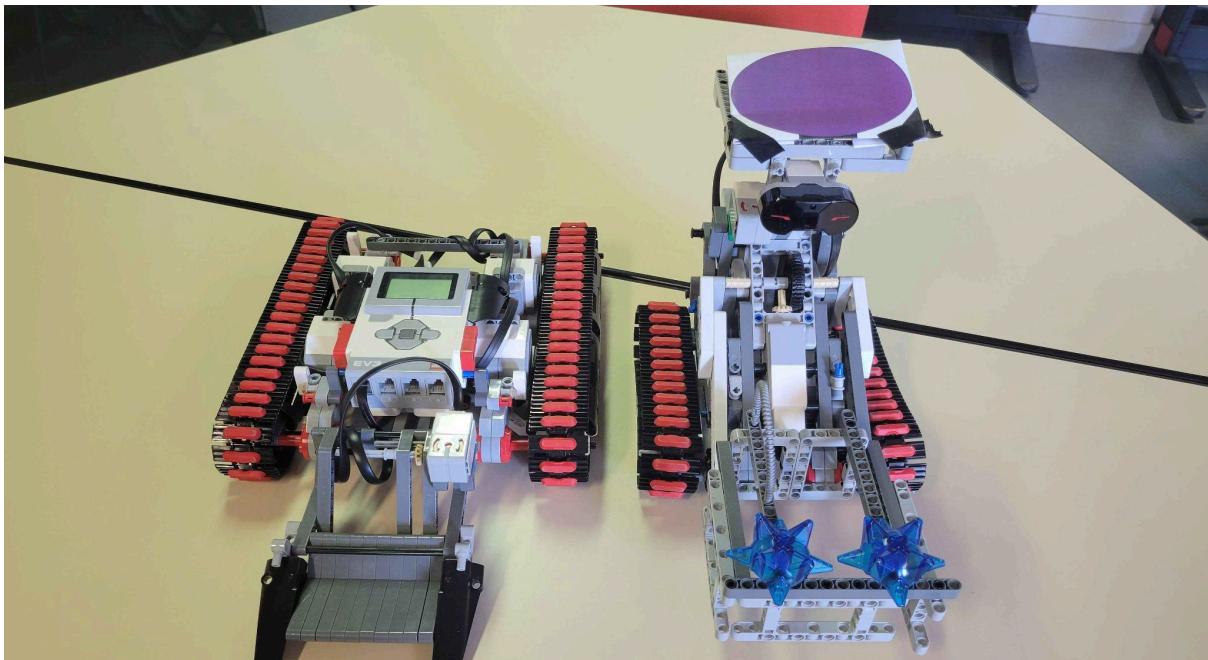


# Rapport Idefix



Robot défenseur      Robot attaquant

## Introduction

Le projet de robotique consiste à faire coopérer une équipe de robot pour accomplir une tâche, ici de jouer au jeu “Capture the flag”. Nous disposons d'une équipe de deux robots: le premier robot a pour rôle de récupérer les balles du camp ennemi. Le second robot quant à lui va défendre le goal de notre camp en repoussant les adversaire en dehors du goal.

Les robots ont été construits à l'aide des modèles disponibles sur le site LEGO que nous avons modifiés pour répondre le mieux possible à nos besoins.

## Rôles

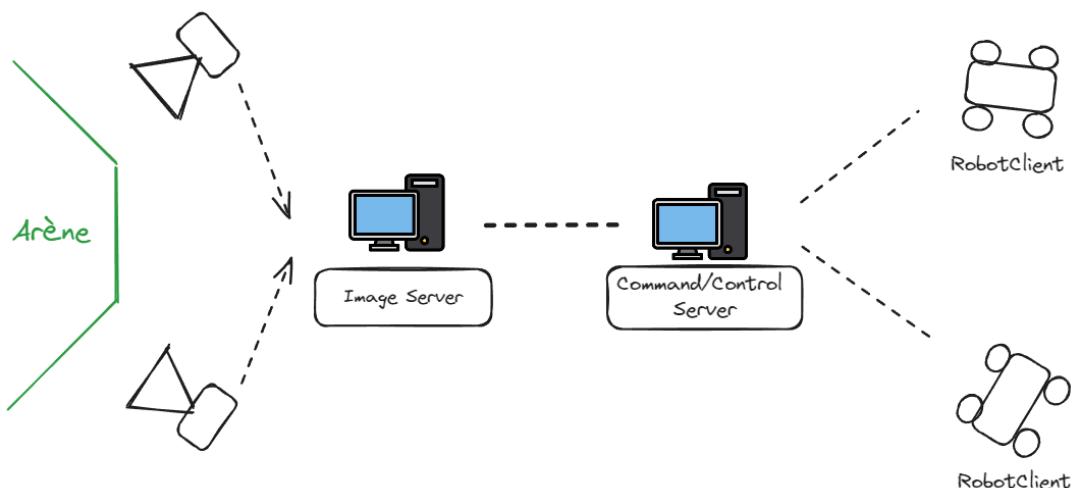
Construction du robot	Yanisse, Koureich
Reconnaissance d'image	Théo G., Yanisse
Programmation du robot	Théo B., Koureich

# Robot

## Client / Serveur

Le projet est construit autour d'un serveur principal qui est chargé d'informer les robots de leur environnement. Le serveur va continuellement traiter les images provenant des caméras de l'arène afin de déduire les positions des robots et objets figurant, puis informer les robots de l'état actuel de l'arène en effectuant un broadcast sur les clients connectés.

Lorsqu'un robot se connecte au serveur, un message handshake est envoyé identifiant le type de robot essayant de se connecter. Après un handshake réussi de la part du serveur, le robot envoie à chaque pas de temps un message informant de son état local (diffère selon le robot). L'état local est ensuite traitée par le serveur dans la boucle principale. Cette disposition permet d'effectuer de la logique côté serveur ou client du robot selon les besoins de notre application. On pourrait imaginer que le serveur effectue un algorithme sur une cohorte de robots pour leur envoyer les stratégies à exécuter en local. Dans le cadre de ce projet nous n'avions pas de traitement supplémentaire à effectuer côté serveur.



Un système de ligne de commande permet de commander à distance les robots. Taper une commande tel que "move x y" envoie message au robot destinataire va traiter de manière adéquate le message. La ligne de commande était surtout pratique pour tester la réception, le traitement du message et tester le comportement du robot. (Mettre image cmd).

```
Choose a socket to connect the command line: (r to reload)
r
r
Choose a socket to connect the command line: (r to reload)
0 : RobotType 0 | socket 4
0
0
move 10 0
```

Enfin, comme les clients / serveurs traitent les messages dans des threads séparés, on assure également l'intégrité des données avec des mutexes. Cela est utile lorsqu'on met à jour le robot depuis le serveur mais que le robot est actuellement en train d'opérer sur ses données.

# Déplacements du robot

Pour les déplacements de nos robots, nous avons implémenté deux modes de déplacements, l'un étant simple et précis mais ne pouvant pas se déplacer et tourner en même temps et l'autre étant plus dynamique. Les robots ayant été programmé séparément, ils n'ont pas exactement les même comportement au niveau du mouvement.

## Déplacement Basique

### Attaquant [\[lien démo\]](#)

Dans les déplacements basiques du robot attaquant, on se dirige simplement vers la cible en effectuant d'abord une rotation en direction de la cible puis on se dirige vers elle jusqu'à l'atteindre. Cela se fait donc en deux temps bien distincts.

### Défenseur [\[Lien démo\]](#)

Dans les déplacements basiques du robot défenseur, on poursuit simplement l'intrus le plus proche du centre sans anticipation de la direction de l'intrus. Si aucun intrus n'est détecté dans la zone on retourne vers le centre de la zone

## Déplacement Intelligent

### Attaquant [\[Lien démo\]](#) / Défenseur [\[Lien démo\]](#)

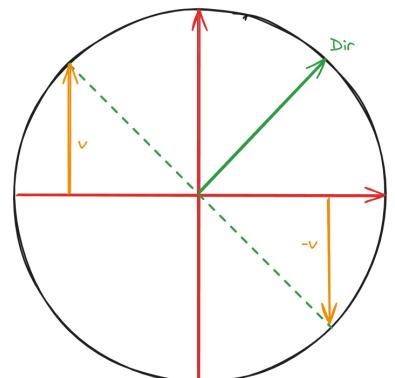
La méthode intelligente permet de faire avancer le robot sans devoir le réorienter à chaque changement de direction. L'idée ici est de calculer quel est la différence de vitesse nécessaire à appliquer au deux roues pour orienter le robot à  $\theta$  degré après  $x$  seconde de déplacement.

En représentant l'angle de la direction sur un cercle trigonométrique, (ci contre) et en traçant la perpendiculaire représentant l'axe de rotation des roues, on peut connaître la différence à appliquer entre les deux forces sur les roues gauches et droites en calculant  $v = \cos(\theta)$ . Soit  $F_g$  et  $F_d$  les forces à appliquer sur les roues gauches et droites, la vitesse de rotation à appliquer sur les roues sont:

$$F_g = \text{vitesse} + v * \text{max}$$

$$F_d = \text{vitesse} - v * \text{max}$$

Avec max correspondant à la force maximale à appliquer lorsque la cible se trouve à droite ( $\leq 0$  deg) du robot. Cette équation fonctionne uniquement lorsque la target se trouve à +/- 90 deg de la direction du robot. Lorsque la direction et la target sont de sens opposé (Fourni par l'équation:  $\text{dot}(t - pos, direction) \leq 0$ ), le robot va opter pour la méthode basique jusqu'à que l'orientation et la target soient dans l'intervalle de définition [0, 180] deg.



Dans le cas du robot défenseur nous avons également besoin d'intercepter les robots en déplacements afin de les empêcher de voler des points à l'équipe. Au lieu de poursuivre une target ayant pour position celle du robot cible, le robot va anticiper ses mouvements : selon la dérivée de la position au temps t et la direction du robot r cible, la target finale est égale à  $target = r.pos + \Delta r.pos$

### Commandes clavier

Il est possible de pouvoir déplacer le robot défenseur à l'aide des touches Z,Q,S,D du clavier depuis la connexion SSH au robot. La détection des touches a été faite à l'aide d'une structure termios en c/c++. Cette fonctionnalité a été intégrée au code pour permettre dans un premier temps de nous faciliter la coordination du robot ainsi que le calibrage des angles avec le gyroscope.

La touche V du clavier permet d'effectuer des rotation d'angle de 90°. La touche A, de dessiner un arc de cercle. La touche P de mettre en pause l'action en cours et la touche M de reprendre l'action.

## Comportements des robots

La boucle de simulation des robots est constitué en 4 étapes:

- Calibration du robot (Effectué au démarrage du robot)
- Exécution de la logique propre au robot (si calibré)
- Mise à jour des vitesses des moteurs
- Calibration en temps réel du robot

La boucle de simulation est faite pour être non bloquante pour permettre l'évaluation de toute la logique en temps réel. Initialement, les déplacements étaient codés de manière bloquante tel que si le robot effectue un mouvement, aucune autre action peut être enclenchée jusqu'à la fin du déplacement.

### Calibration du robot

Lorsque le robot est initialisé, il n'a aucune connaissance sur son environnement et reste bloqué à l'étape de calibration initiale. Le but de la calibration est de connaître la position et l'orientation du robot. La calibration est constitué de 3 étapes:

Étape 1: Attente de la mise à jour provenant du serveur sur l'environnement.

Le serveur connaît la position de l'ensemble des objets de l'arène, le robot va attendre jusqu'à qu'un message du serveur contient sa position.

Étape 2: Le robot avance d'une distance  $d$  et attend un nouveau message du serveur.

En faisant avancer le robot et sans changer de direction, on peut connaître son orientation grâce à l'équation  $\text{atan2}(newpos - oldpos)$

Étape 3: Le robot fait marche arrière à sa position initiale.

Le robot est prêt à démarrer et la logique interne est enclenché.

Pendant l'exécution du robot, nous essayons également de calibrer en temps réel le robot. La méthode consiste à évaluer comme à l'étape de calibration initiale la différence entre la position actuelle et la position précédente. Le calcul est effectué sur la position au temps  $t-2s$  et effectué uniquement si l'orientation du robot n'a pas évolué au dessus d'un seuil  $\epsilon$

### Boucle logique des robots

La logique du robot attaquant est assez simple:

- Récupérer la balle la plus proche de soi
- Fermer la cage lorsque  $d < \epsilon$
- Se déplacer vers le goal allié
- Lever la cage, puis répéter

Le robot défenseur quant à lui effectue la logique suivante:

- Déetecter le robot le plus proche du goal
- Si le robot est dans le goal
  - Poursuivre et pousser le robot
- Revenir au centre du goal lorsqu'il n'y a aucun robot proche du but

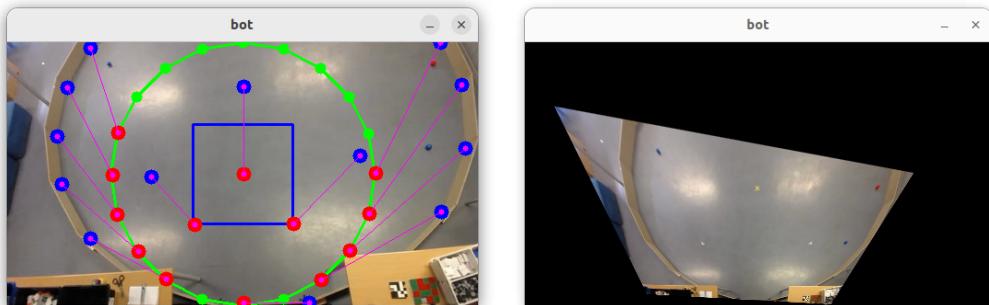
# Détection

La partie analyse d'image, pour la détection des balles et des robots, a été réalisée en C++ avec OpenCV. Cette détection a pour objectifs de fournir la position dans le monde (en cm), des balles rouges et bleues dans l'arène, ainsi que la position des robots. Le centre du monde (0,0), correspond au centre de l'arène.

## Homographie

La première chose à mettre en place pour pouvoir la partie détection est l'homographie. Cela permet d'avoir une vue du dessus pour que l'on puisse calculer la position des objets détectés.

L'arène est filmée par quatre caméras, soit quatre images/vignettes à traiter, de sorte à avoir une vue complète de l'arène. L'arène est un icosagone (polygone à 20 côtés). Pour l'homographie, on a besoin de deux vecteurs de points : les points sources et destinations. Les points sources sont les coordonnées des points de repère de l'arène (sommets de l'icosagone, centre et sommets du carré central) dans les vignettes reçues par les caméras. Les points destinations représentent là où doivent être mappés les points sources pour simuler une vue du dessus.

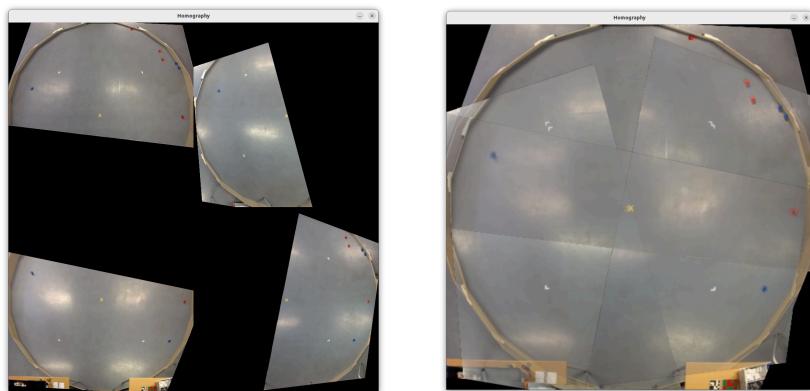


## Construction des données d'homographie

Dans un premier temps, nous avons développé une application pour construire les données d'homographie. L'image reçue contient quatre vignettes, correspondant chacune à une caméra. On les sépare pour pouvoir effectuer les opérations d'homographie. Pour chaque vignette, on récupère les positions source avec le clic gauche de la souris et les positions destination avec le clic droit. Ces coordonnées en pixel sont normalisées en utilisant la dimension des images : on divise la valeur en x par la largeur et y par la hauteur. Les données sont sauvegardées dans quatre fichiers vertices-norm-000x.txt. Ces fichiers contiennent les positions normalisées de l'arène dans l'image (positions source), ainsi que celles de l'arène dans le monde (positions destination). Ces données sont ensuite lues et stockées dans des vecteurs à chaque frame. L'homographie est effectuée avant de passer à la partie détection.

## Utilisation des données d'homographie

Tout d'abord, on lit les données que l'on stocke dans une structure *HomographyVertices* qui se compose de deux vecteurs. On redimensionne l'image en 512x512. Grâce aux valeurs récupérées depuis le fichier et les nouvelles dimensions, on calcule les coordonnées des points dans l'image (on inverse l'opération de normalisation en utilisant les nouvelles dimensions). Enfin, on appelle les méthodes d'homographie d'OpenCV sur les données calculées. On fait cette opération sur les quatre vignettes. Ensuite, il y a plusieurs possibilités : faire la détection d'objets sur les vignettes concaténées, sur les vignettes fusionnées, ou séparément. Nous discuterons de leurs avantages / inconvénients dans les parties suivantes.



## Détection des balles

La détection se fait en trois phases. Une phase de prétraitement où on applique un filtre bilatéral à l'image afin de réduire le bruit tout en conservant le contour des objets. Une phase de détection des couleurs qui fournit une image binaire correspondant aux pixels dont les valeurs HSV sont comprises dans l'intervalle fourni. Sur l'image binaire, on effectue une ouverture pour séparer les balles détectées, suivie de trois dilatations qui permettront de rendre les balles plus facilement détectables. Enfin, une phase permettant de calculer le centre des objets détectés. Pour cette dernière phase, deux approches ont été testées. Une qui utilise les moments pour le calcul du centre de gravité des objets avec en entrée les vignettes fusionnées et une qui utilise la transformé de Hough pour détecter les cercles dans les vignettes séparées.

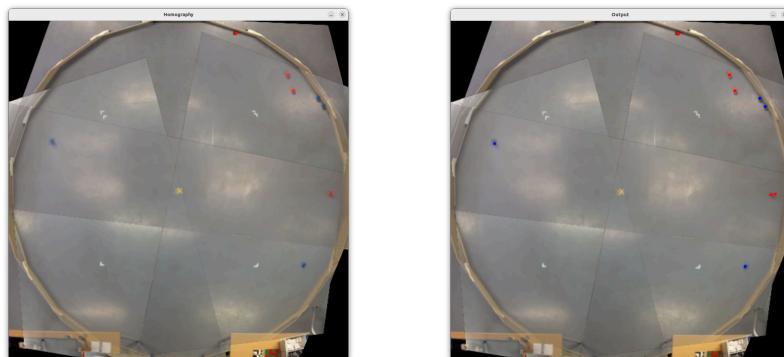
## Détection du centre des objets

### Première approche : utilisation des moments

Pour cette approche, on prend en entrée l'image binaire obtenue après la détection des couleurs sur les vignettes fusionnées (après homographie).

Le calcul du centre de gravité des formes dans une image utilise les moments pour déterminer le point central de chaque contour détecté. Le code commence par détecter les contours dans l'image source en utilisant l'algorithme de *Canny* pour la détection des bords, et stocke les contours détectés dans un vecteur de vecteurs de points. Ensuite, pour chaque

contour trouvé, les moments géométriques sont calculés et stockés dans un vecteur. À partir de ces moments, on calcule le centre de gravité (centroïde). On stocke le point seulement s'il se trouve à l'intérieur du cercle de même rayon que l'arène.



Le problème majeur de cette approche est que les objets sont "floutés" et les couleurs ressortent moins ce qui pose problème pour la détection des robots et des cubes. Aussi, comme on ne se soucie pas de la forme de l'objet détecté, certaines parties des robots de couleur rouge ou bleu sont détectées comme des balles.

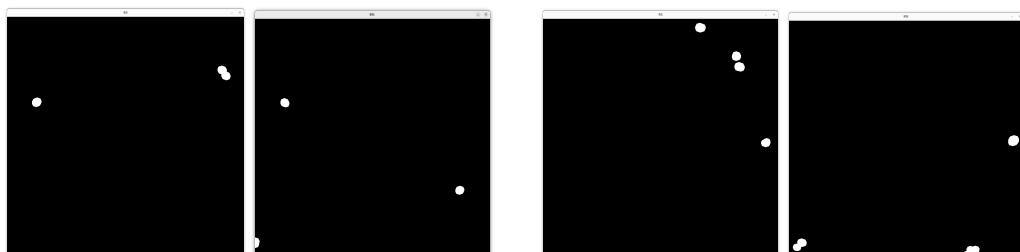
### Seconde approche : utilisation de Hough Circle

Pour cette seconde approche, la détection se fait sur deux des quatre vignettes. On prend les deux vignettes qui permettent d'avoir une vue complète sur l'arène.



La détection de couleur nous donne les images binaires suivantes :

Détection couleur bleue et rouge :



À la place du calcul des moments, on fait de la détection de cercle avec la transforme de Hough. Comme pour l'approche précédente, on fait de la détection de contour avec l'algorithme *Canny*. On utilise la méthode *HoughCircles* d'OpenCV pour récupérer la position et le rayon des cercles détectés. En sortie, on obtient le vecteur contenant la position de chaque balle.

## Calcul des positions dans le monde

Après la détection, on récupère des vecteurs de position de balle. Leurs coordonnées sont en pixel et on veut les convertir en position dans le monde (position en cm par rapport au centre de l'arène). Tout d'abord, on transforme les coordonnées en prenant comme (0, 0) le centre de l'image (centre de l'arène). Ensuite, on multiplie la nouvelle position par le ratio cm/pixel.

Toutes les deux frames, on compare les positions courantes avec les positions précédentes. On ne garde que les balles qui n'ont pas bougé d'un certain delta par rapport aux frames précédentes.

## Détection des robots

La détection des robots est similaire à la détection de balle puisqu'elle se base sur une détection de couleur et de cercle. On utilise les mêmes méthodes que pour la détection de balles, en passant l'intervalle HSV correspondant à la couleur de la pastille des robots.



## Détection des cubes

Nous avons également mis dans le code un algorithme de détection de cubes qui n'a malheureusement pas été utilisé, car les images fusionnées par l'homographie ont subi une baisse de qualité, et avec l'éclairement les cubes (de couleurs gris) sont devenues très peu visibles.

Cet algorithme consistait à pré-traiter l'image avec dans un premier temps un filtre permettant d'obtenir une image binaire contenant seulement les contours, les contours représentant une forme sont ensuite stockés dans un tableau de points.

Dans un deuxième temps, l'algorithme de Douglas-Peucker est utilisé pour vérifier que la forme est convexe, et si c'est le cas nous conservons celles qui ont 4 côtés.

Dans un troisième temps, nous vérifions les angles des quadrilatères conservés et nous nous assurons que leurs angles mesurent environ  $90^\circ$ .

Et enfin, une fois toutes les conditions remplies, nous faisons la moyenne des 4 points du quadrilatère pour récupérer son centre.

## Difficultés et améliorations possible

Dans la réalité, notre robot n'est pas capable de se calibrer correctement, dû aux imprécisions de la détection. Après coup, nous pensons que cette imprécision vient d'un manque de mutex pour assurer l'intégrité l'image venant de la capture des caméras, entre la réception de celle et de son traitement dans le thread principal.

Le robot a également des soucis à opérer avec le déplacement intelligent lorsque la vitesse d'une des deux roues est faible (< 100 tacho/s), faisant dévier le robot de sa cible..

Une des difficultés qu'on a rencontrées lors de la programmation du client / serveur concernait l'architecture ARM. Lorsque le client est exécuté sur une machine ARM et connecté à un serveur, le client reçoit des valeurs incohérentes. La cause de ce problème vient de plusieurs endroits où l'accès à la mémoire avec un pointeur était effectué de manière non alignée. ( Exemple: accéder au bytes d'un entier en faisant un cast char\* )

L'exécution du serveur sous WSL causait également des soucis empêchant l'accès au serveur. La couche networking entre WSL et la machine hôte devait être configurée différemment avec un fichier de config pour pouvoir s'exécuter normalement..