



UNIVERSITÉ DE ROUEN NORMANDIE

Rapport de Stage

Master 1 :

Systèmes Intelligents Mobiles pour l'Embarqué (SIME)



Développement du robot Niryo Ned² pour l'automatisation de parties d'échecs humain vs robot avec algorithmes d'apprentissage par renforcement et vision par ordinateur

Réalisé par :

Sous direction de :

Encadré par :

Eliot BERTHOMIER *Maxime BERAR* *Jason PIQUENOT*

Stage réalisé au sein de l'organisme :



Laboratoire d’Informatique, de Traitement de l’Information et des Systèmes.

Historique du document

Version	Date	Option	Auteur
Première version	16 juin 2025	Document initial	E.B
Deuxième version	19 juin 2025	Modif liens et table	E.B

Remerciements

Avant toute chose, je souhaiterais remercier toutes les personnes et institutions dont le soutien a été précieux tout au long de ce stage et lors de la rédaction de ce rapport.

Un grand merci à **Jason PIQUENOT** qui a encadré ce stage de la meilleure manière qu'il soit. Merci pour sa disponibilité, son aide et soutien constant auprès des étudiants participant à ce projet que ce soit en accompagnant les étudiants du master A2IA ou SIME.

Je tiens également à remercier **Maxime BERAR** et **Sébastien ADAM** respectivement pour la direction et l'élaboration de ce projet ainsi que tous les membres du LITIS et de l'UFR Sciences et Techniques pour leur collaboration et soutien au projet.

Je remercie par ailleurs **Youssouf SAIDALI** et **Lilian ROBERT** respectivement pour leurs conseils dans la partie matérielle du projet et le service d'impression permettant une progression constante tout au long du stage.

J'exprime une grande gratitude envers les membres de l'entreprise Niryo développant les robots que nous utilisons pour mener à bien notre projet de développement en mentionnant leur rapidité de réponse et leur intérêt dans ce stage.

Je tiens à remercier les membres du jury pour le temps consacré à la lecture et à l'évaluation de ce rapport : leurs observations contribueront à en améliorer la qualité. J'adresse également ma gratitude à l'ensemble des enseignants pour leur soutien tout au long de ma formation ; leurs conseils et leur disponibilité ont été précieux pour mener à bien ce stage. Merci à tous.

Introduction

Contexte et problématique

Le laboratoire [LITIS](#) (Laboratoire d'Informatique, de Traitement de l'Information et des Systèmes) au sein de l'Université de Rouen Normandie propose régulièrement des stages de fin d'année pour les étudiants de master. En concevant ce stage de deux mois, le LITIS crée une véritable passerelle entre deux masters de la branche [Sciences des Données](#). En effet, le croisement des masters [SIME](#) (Systèmes Embarqués Mobiles pour l'Embarqué) et [A2IA](#) (Apprentissage Automatique pour l'Intelligence Artificielle) permet de regrouper trois étudiants aux compétences complémentaires pour la réalisation du projet suivant : doter un bras robotique à six axes [Niryo Ned 2](#) de la faculté de jouer aux échecs contre un adversaire humain.

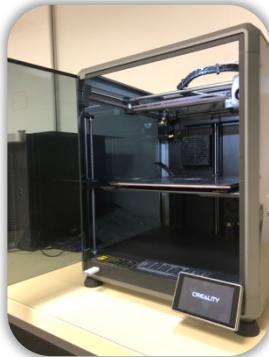
Pour ce faire, l'étudiante en A2IA sera chargée de développer une intelligence artificielle basée sur le renforcement permettant de déterminer le prochain coup à jouer du robot. Les étudiants en SIME dont je fais partie, devront quant à eux prendre en charge l'intégration matérielle et logicielle garantissant que le robot puisse interpréter l'échiquier, localiser et saisir les pièces avec précision, et exécuter des trajectoires fluides tout en intégrant l'algorithme de génération du coup suivant du robot par renforcement. Le système embarqué pourra donc jouer aux échecs contre un humain en détectant ses coups en temps réel, offrant ainsi une expérience de jeu à la fois convaincante et réactive.

Au-delà d'une simple démonstration technologique, ce stage vise à servir lors d'expositions pour les futures représentations de la fête de la science, se déroulant à [l'UFR Sciences et Techniques](#). Ces démonstrations permettront de valoriser les compétences du master SIME et A2IA mais aussi la promotion du laboratoire LITIS dans l'encadrement de stages préparant à l'insertion professionnelle. Dans cette même lignée, une collaboration avec [l'entreprise Niryo](#) (Spécialiste des bras-robots) basée à Lille, pourrait ouvrir la voie à plusieurs opportunités d'alternances ou [PICs \(Projets d'Innovation et Conception\)](#) confirmant ainsi l'ambition de ce stage : établir un lien durable entre recherche académique et innovation industrielle.

Présentation de l'organisme d'accueil



Le stage s'est déroulé au Laboratoire d'Informatique, de Traitement de l'Information et des Systèmes (LITIS) de l'Université de Rouen Normandie, une unité de recherche reconnue pour ses travaux en apprentissage automatique, vision par ordinateur et systèmes embarqués. Au sein de cet environnement, j'ai pu m'appuyer sur des équipements robotiques et informatiques pour développer notre bras robot 6 axes Niryo Ned 2 capable de jouer aux échecs. Le LITIS nous a également permis l'accès à la salle UniverTech, comportant divers outils à notre disposition tels que des imprimantes 3D de type FDM et SLA, outils de coupe portatifs et abrasifs. Ces outils et matériaux seront présentés plus en détail dans la partie suivante, « Présentation du matériel ».



Les compétences croisées des équipes du LITIS, associant ingénierie logicielle, traitement d'images et intelligence artificielle, ont offert un cadre idéal pour la mise en œuvre de ce projet. Le soutien constant des chercheurs, qu'il s'agisse d'expertise technique pour la calibration du robot ou de conseils méthodologiques pour l'entraînement du modèle de reconnaissance des pièces, a été déterminant pour mener à bien ce travail et atteindre les objectifs.

Rappel des règles des échecs

Une partie d'échecs se joue à 1 contre 1 où chaque joueur occupe le camp blanc ou le camp noir. Le jeu se déroule sur un échiquier, quadrillage de 64 cases alternant entre clair et foncé (usuellement noir et blanc). Ce quadrillage constitue également 8 rangées (les « rangs » numérotés de 1 à 8) et 8 colonnes (appelées « files » notées de *a* à *h*). Chaque case porte ainsi un nom unique (par exemple e4 ou a7).

Chaque joueur débute avec seize pièces : un roi, une reine, deux tours, deux fous, deux cavaliers et huit pions, répartis sur les rangs 1 et 2 (Blancs) ou 7 et 8 (Noirs). Les tours se déplacent en ligne droite (files ou rangs), les fous uniquement en diagonale, la reine combine les deux, le cavalier exécute un déplacement en « L » (deux cases dans une direction puis une case perpendiculaire) et le roi se déplace d'une case dans n'importe quelle direction. Les pions avancent d'une case vers l'avant (file inchangée), peuvent sauter de deux cases lors de leur premier coup, et capturent en diagonale (une case en avant à gauche ou à droite).

L'objectif ultime est de placer le roi adverse en « échec et mat », c'est-à-dire dans une position où il serait capturé au coup suivant sans issue de secours. La partie peut également se clore par une nulle (accord mutuel, répétition de positions, impasse, etc.).

Au fil de la partie, des règles spéciales entrent en jeu : la « prise en passant » permet à un pion de capturer un pion adverse qui vient de se déplacer de deux cases et se retrouve à côté de lui ; le « roque » associe déplacement du roi et d'une tour pour protéger le monarque ; la promotion transforme un pion arrivé sur la dernière rangée adverse en une autre pièce (généralement une reine).

Avec un nombre de configurations théoriques estimé à l'ordre de 10^{120} les échecs sont l'un des jeux combinatoires les plus riches jamais créés, offrant une profondeur stratégique infinie malgré la simplicité apparente de ses règles.

Règles du jeu en détail :

<https://rcc.fide.com/wpcontent/uploads/2020/05/LawsOfChessFrenchTranslation1.pdf>

Objectifs du Stage

Ce stage a pour but le développement d'un programme permettant les déplacements réfléchis du bras robot six axes Niryo Ned 2 afin que ce dernier puisse jouer une partie d'échecs contre un humain. Pour ce faire, deux étudiants en SIME et une étudiante en A2IA coopèrent :

Pour la partie A2IA, l'objectif est de créer une IA d'apprentissage par renforcement qui permet de générer et d'indiquer le prochain coup à jouer au robot. Cette pièce du projet ne sera pas développée dans ce rapport car elle est sujette à un document à part entière réalisé par **Thiziri FERAKA**, étudiante en A2IA et collaboratrice sur ce projet durant ce stage.

Pour notre partie, en SIME, nous devons être capables de réaliser plusieurs tâches faisant appel à des connaissances en conception orientée par ordinateur, électronique, machine learning, vision par ordinateur et développement logiciel :

- **Création de l'échiquier et des pièces :** La partie d'échecs se déroule sur un échiquier de 64 cases alternées sur lequel il sera nécessaire de placer 32 pièces (16 blanches et 16 noires). Cet échiquier et les pièces seront créés par impression 3D grâce aux imprimantes FDM et SLA à notre disposition.
- **Prise en charge des déplacements :** Le robot doit pouvoir de lui-même déplacer des pièces, effectuer des captures de ces dernières, effectuer les roques etc. Il doit également pouvoir se mettre en position d'attente lorsque c'est au joueur de jouer. Ces positions doivent être calculées en temps réel.
- **Choix du type de prise :** Lors du déplacement de pièces, le robot doit utiliser un outil de saisie (pinces, électro-aimant, pompe à vide et ventouse etc.). Nous devons veiller à utiliser un de ces outils assurant à la fois performance et précision.
- **Détection de l'échiquier :** Nous devons être capables de repérer et d'enregistrer les positions des cases de l'échiquier dans l'espace de travail du robot à n'importe quel instant.
- **Détection des pièces :** Il est nécessaire de pouvoir détecter les pièces présentes sur l'échiquier en temps réel et de déterminer leurs positions sur ce dernier. De cette manière nous pourrons récupérer un échiquier complet avec des cases vides ou munies d'une pièce spécifique si la case est occupée. Il faudra donc utiliser un réseau de neurones convolutifs pour l'analyse d'images.
- **Intégrer le modèle d'apprentissage par renforcement :** Cette partie consiste à mettre en commun le travail réalisé par Thiziri et le nôtre. En effet, il est question d'intégrer à la boucle de jeu l'appel au modèle nous permettant de générer le prochain coup que le robot va devoir jouer. La boucle de jeu se constitue en cycles où le joueur joue, puis le robot répond en capturant ou déplaçant une pièce, jusqu'à la fin de partie.

Table des matières

Contexte et problématique	4
Présentation de l'organisme d'accueil	5
Rappel des règles des échecs.....	6
Objectifs du Stage.....	7
1. Présentation du matériel	12
1.1 Le robot Niryo Ned 2	12
1.2 Outils d'impression 3D	13
1.3 Outils de développement	15
2. Modélisation 3D.....	16
2.1 L'échiquier	16
2.1.1 Contraintes	16
2.1.2 Design de l'échiquier.....	17
2.1.3 Assemblage et Rendu	17
2.2 Les pièces.....	18
2.2.1 Contraintes et choix de préhension.....	18
2.3 Conclusion	19
3. Détection de l'échiquier en temps réel	20
3.1 Le Workspace	20
3.2 Les NiryoMarkers	21
3.3 Détection des cibles	21
3.4 Calcul de la position de l'échiquier.....	22
3.5 Conclusion	23
4. Programmation du robot Niryo Ned 2.....	24
4.1 L'API pyniryo.....	24
4.1.1 Le pose object	24
4.2 Quelques contraintes.....	25
4.2.1 Récupération des poses de calibration.....	25
4.3 Configuration et initialisation du robot.....	25
4.4 Calcul des poseObject pour chaque case	26
4.5 Fonctions de capture et déplacement.....	26
4.5.1 Fonctions de prise et dépôt.....	26
4.5.2 Exécution principale des coups.....	27
4.6 Conclusion	28
5. Détection des pièces	29
5.1 L'objet board()	29
5.2 Processus de détection.....	30
5.2.1 Prospection.....	30

5.2.2	Choix du modèle et Dataset	31
5.3	De l'image aux positions des pièces sur l'échiquier	33
5.4	Détection du camp du joueur	33
5.5	Conclusion	34
6.	Intégration de l'IA d'apprentissage.....	35
6.1	Calcul du prochain coup.....	35
6.2	Mise en place de la pipeline.....	35
6.2.1	Préparation.....	35
6.2.2	Boucle du jeu.....	36

Table des figures

Figure 1 : Photo du robot Niryo Ned 2	12
Figure 2 : Différents outils de prise interchangeables du robot.....	13
Figure 3 : Schéma du back-panel du robot.....	13
Figure 4 : Ultimaker S2+(à gauche) et S5 (à droite)	14
Figure 5 : Imprimante 3D SLM Creality K1 Max	14
Figure 6 : FormLabs form 3+.....	14
Figure 7 : Schéma du Niryo Ned 2 déplié au maximum	16
Figure 8 : quart de l'échiquier sous fusion 360	17
Figure 9 : Colle Araldite pour coller l'échiquier	17
Figure 10 : Résultat du prototype assemblé de l'échiquier	18
Figure 11 : Schéma 3D final des pièces.....	19
Figure 12 : Prototypes de Fou et Pion (v1).....	19
Figure 13 : Exemple du workspace fourni par Niryo	20
Figure 14 : Cibles Niryo (NiryoMarkers)	21
Figure 15 : Exemple de détection de cibles en python.....	21
Figure 16 : Schéma du calcul de l'échiquier à partir des NiryoMarkers.....	22
Figure 17 : Démonstration du calcul de l'échiquier en temps réel	22
Figure 18 : Définition de la fonction get_workspace_poses_ssh()	25
Figure 19 : Algorithme d'initialisation des poses des cases de l'échiquier.....	26
Figure 20 : Exemple de la fonction de prise	27
Figure 21 : Principe d'occultation du fou par la reine.....	30
Figure 22 : Représentation des pièces avec étiquettes de couleur	31
Figure 23 : Principe de fonctionnement de YOLO	31
Figure 24 : Annotation des pièces d'échecs.....	32
Figure 25 : métriques d'évaluation du modèle.....	32
Figure 26 : Output de la fonction board_state_from_yolo()	33
Figure 27 : Algorithme de la fonction player_has_white()	34
Figure 28 : Algorithme de la boucle du jeu	36
Figure 29 : Cases assemblables	38
Figure 30 : Échiquier final (à gauche) contre prototype (à droite)	38

Chapitre 1

1. Présentation du matériel

Cette partie vise à présenter le matériel mis à disposition pour mener à bien ce projet. En effet, le master SIME s'inscrit dans une démarche d'ingénierie et d'innovation à la fois physique et matérielle. Au cours de notre licence et de notre master, nous avons multiplié les travaux pratiques : soudure et test de circuits, montage de capteurs, modélisation 3D etc. Comparé au parcours A2IA, qui privilégie l'étude des algorithmes avancés et des réseaux de neurones, le master SIME nous forme à la mise en œuvre concrète de systèmes embarqués, tout en abordant également la partie logicielle et IA, puisque la programmation des firmwares et l'orchestration des briques logicielles restent au cœur de nos projets. C'est cette polyvalence, entre intervention matérielle et lignes de code, qui nous permet de prendre en charge l'ensemble de la chaîne de fabrication et d'intégration de ChessBot.

1.1 Le robot Niryo Ned 2

Pour le bras robot, nous utilisons le Ned 2 de chez Niryo. Il comporte six axes de rotation et peut saisir des objets jusqu'à 49 centimètres. Il peut être équipé de différents objets de saisie : une pince, un électro-aimant ou une ventouse reliée à une pompe à vide.



Figure 1 : Photo du robot Niryo Ned 2

Le panneau arrière du Niryo Ned 2 regroupe tous les connecteurs essentiels : ports USB 2.0/3.0, Ethernet Gigabit, Wi-Fi dual-band et un bloc d'entrées/sorties (numériques et analogiques). Il permet de raccorder directement la caméra du Vision Set, une pompe à vide, un électro-aimant ou tout autre capteur, offrant ainsi une modularité rapide pour des tâches de vision, de pick & place ou d'expérimentations industrielles.

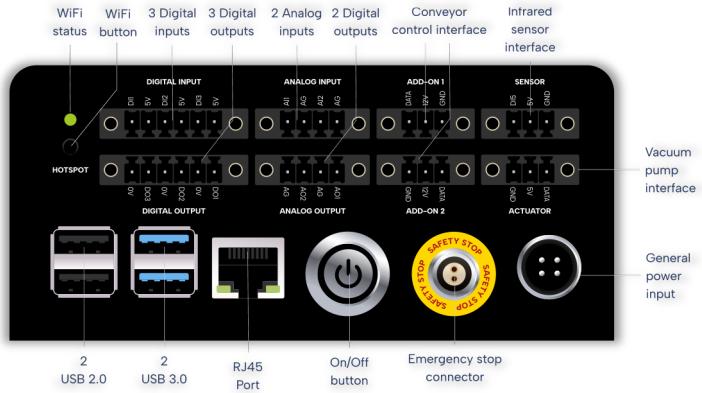


Figure 3 : Schéma du back-panel du robot



Figure 2 : Différents outils de prise interchangeables du robot

1.2 Outils d'impression 3D

Pour la création des pièces et de l'échiquier, nous avons à disposition plusieurs imprimantes 3D comportant des spécificités différentes. Le premier type d'imprimante que nous utilisons sert à la réalisation de prototypes ou d'objets de grande surface comme des tests pour les pièces ou l'impression monobloc de l'échiquier. Ces imprimantes impriment par dépôt de fil fondu (ou FDM, Fused Deposition Modeling) reposant sur un extrudeur qui pousse un filament thermoplastique chauffé à haute température dans une buse, pour le déposer couche après couche sur un plateau chauffant et bâtir progressivement la pièce. Le processus commence par la modélisation 3D dans un logiciel de CAO (Fusion 360, FreeCAD, Blender, etc.), puis l'export du modèle au format STL, OBJ ou 3MF, avant le tranchage dans un slicer (Ultimaker Cura, PrusaSlicer, Simplify3D...) qui convertit la géométrie en instructions G-code. Ce G-code est ensuite envoyé à l'imprimante, il pilote les moteurs pas à pas, contrôle la température et les ventilateurs pour réaliser l'objet selon les réglages de vitesse, hauteur de couche et densité de remplissage définis.

Nous avons à notre disposition deux imprimantes FLM de marque [Ultimaker](#) : la [S2+](#) et la [S5](#) :



Figure 4 : Ultimaker S2+ (à gauche) et S5 (à droite)

Nous avons également une autre imprimante FLM de marque [Creality](#) : la K1 Max :



Figure 5 : Imprimante 3D SLM
Creality K1 Max

Pour l'impression des pièces, nous avons la possibilité d'utiliser des imprimantes par stéréolithographie (SLA) utilisant un réservoir de résine photosensible qu'un faisceau laser ou un projecteur UV vient durcir sélectivement couche après couche. Le fichier 3D est, comme pour le FLM, exporté en STL ou 3MF puis tranché par un logiciel propriétaire (PreForm dans notre cas) qui génère des commandes de scan laser. Le plateau s'abaisse progressivement dans la résine, chaque passe de lumière polymérise une fine couche (de 25 à 100 µm d'épaisseur), offrant un niveau de détail et une finition bien plus fins qu'en FLM, au prix d'un post-traitement (nettoyage à l'alcool isopropylique et polymérisation UV supplémentaire). Nous utilisons la FormLab form 3+



Figure 6 : FormLabs Form 3+

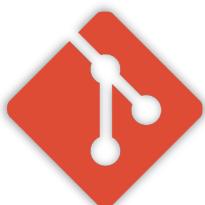
Pour la création des pièces et de l'échiquier, nous utiliserons le logiciel de CAO (conception assistée par ordinateur) [Autodesk : Fusion 360](#).



1.3 Outils de développement



Le développement des déplacement et logique du robot se fera principalement dans le langage python. Pour ce faire, nous allons utiliser l'IDE « [Visual Studio Code](#) » qui est un éditeur de code source gratuit et multiplateforme développé par Microsoft. Léger et rapide au démarrage, il intègre des fonctionnalités avancées comme l'IntelliSense (auto-complétion contextuelle), un débogueur intégré, un terminal embarqué et un gestionnaire d'extensions très fourni. Grâce à son support natif de nombreux langages et à sa personnalisation par thèmes et réglages, VS Code s'est imposé comme un outil de référence pour le développement web et logiciel.



Git est un système de gestion de versions décentralisé qui permet de suivre l'historique des modifications de notre code, de revenir à n'importe quelle étape et de travailler simultanément sur différentes fonctionnalités grâce aux branches. [GitHub](#), quant à lui, est une plateforme en ligne qui héberge nos dépôts Git, facilite le partage et la synchronisation du code et offre des outils collaboratifs comme les pull requests, la revue de code et le suivi des issues. Pour notre projet à trois, l'association de Git et GitHub garantit une coordination fluide : chacun peut développer sa partie sans écraser le travail des autres, proposer ses changements pour validation, et documenter clairement les tâches ou problèmes rencontrés. Cette organisation assure à la fois traçabilité, transparence et efficacité dans la conduite de ChessBot.

Chapitre 2

2. Modélisation 3D

Pour concevoir l'échiquier et les pièces, nous avons d'abord réalisé une phase de prospection pour prendre en compte à la fois les contraintes mécaniques du bras Niryo Ned 2 (portée, précision de préhension) et l'ergonomie pour le joueur (taille des cases, lisibilité des pièces). Sur cette base, nous avons modélisé plusieurs prototypes 3D sous Fusion 360 afin de valider la prise en main du robot et la manipulation humaine. Chaque itération a été soumise à des tests de robustesse et de précision, garantissant un workflow fiable avant passage à la version finale. En accélérant ces phases de prototypage, nous avons pu disposer du matériel définitif et ainsi démarrer le développement des algorithmes de déplacement et les essais de manipulation robotisée.

2.1 L'échiquier

2.1.1 Contraintes

La conception de l'échiquier se précède d'une phase de prospection très importante pour la suite du projet. En effet, il est une pièce maîtresse dans l'environnement global du robot, ses dimensions se doivent d'être préconisées au préalable en fonction de la distance à laquelle le robot peut se déplacer et saisir des objets. Le robot peut s'allonger jusqu'à 49 centimètres mais cela n'implique pas le fait de pouvoir prendre des objets.

Dans un premier temps, il est important de comprendre, comment est constitué le robot pour ensuite calculer sa distance de préhension efficace.

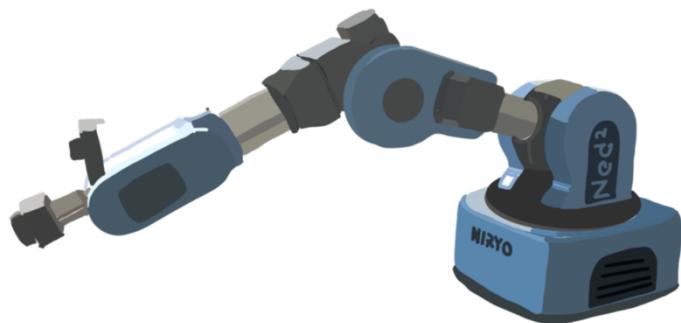


Figure 7 : Schéma du Niryo Ned 2 déplié au maximum

En tenant compte de l'extension fixe de 10 cm apportée par le dernier axe du robot et d'une hauteur moyenne des pièces de 7 cm posées sur un plateau de 1 cm d'épaisseur, on obtient une portée utile d'environ 43 cm. Pour dimensionner un échiquier de 8 cases de côté x bordé de chaque côté par une margelle de même largeur x , sa largeur totale L doit satisfaire :

$$L = 8x + 2x = 10x < 43 \quad \Rightarrow \quad x < \frac{43}{10} = 4,3 \text{ cm.}$$

Chaque case ne peut donc pas dépasser 4,3 cm de côté afin de respecter les contraintes de distance du robot. Pour le prototype, nous allons créer un échiquier comportant des cases de 3,25 cm de côté.

2.1.2 Design de l'échiquier

La taille du plateau de l'Ultimaker S5 (280×350 mm) ne nous permet pas d'imprimer l'échiquier en une seule pièce, dont la largeur totale atteint 325 mm. Nous avons donc choisi de le découper en quatre quarts, chacun d'environ 162,5 mm de côté, qui tiennent aisément sur le plateau. Grâce à la double extrusion de l'imprimante, nous pouvons toutefois imprimer chaque segment en deux couleurs simultanément, alternant directement cases claires et foncées sans avoir à réaliser autant de petites pièces individuelles. Cette configuration simplifie grandement l'assemblage final et préserve la continuité visuelle du damier.

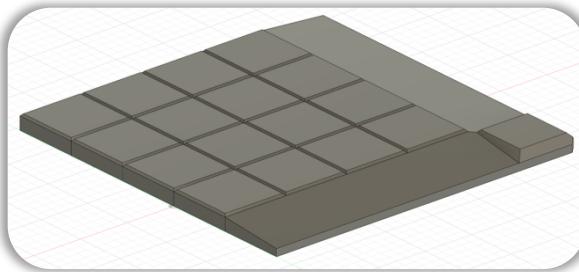


Figure 8 : quart de l'échiquier sous fusion 360

2.1.3 Assemblage et Rendu

Pour l'assemblage des quatre parties de l'échiquier, nous avons opté pour de la colle de type Araldite qui est une colle époxy bi-composant. Une fois résine et durcisseur mélangés, elle offre une adhérence et une résistance élevées sur de nombreux matériaux (plastique, métal, bois). Elle dispose d'un court temps de travail avant prise et atteint sa résistance maximale en 24 h, idéale pour assembler ou renforcer nos pièces imprimées en 3D.



Figure 9 : Colle Araldite pour coller l'échiquier



Figure 10 : Résultat du prototype assemblé de l'échiquier

2.2 Les pièces

2.2.1 Contraintes et choix de préhension

Avec des cases de 3,25 cm de côté, nous avons tout d'abord ajusté les proportions de chaque pièce selon les préconisations de la [FIDE](#) : la hauteur du roi a été fixée à environ 6,9 cm (soit $2,125 \times 3,25$ cm) et le diamètre de sa base à 40 % de cette hauteur, soit près de 2,8 cm. Les autres pièces ont été dessinées en gardant ces mêmes rapports, garantissant cohérence visuelle et équilibre sur le plateau. Chaque modèle a été précisément modélisé sous Fusion 360, puis imprimé pour vérifier à la fois la stabilité au repos et la fiabilité de la prise par le bras Niryo Ned 2.

De plus, nous devons choisir quel type d'outil nous allons utiliser pour déplacer les pièces et assurer leur capture : Le premier outil est la pince, elle est précise mais volumineuse, ce qui peut poser problème lors du déplacement de pièces entourées par d'autres. En effet, la pince peut décaler ou faire tomber certaines pièces aux alentours provoquant un risque de fausse interprétation par notre modèle de reconnaissance. Le second outil est la pompe à vide, elle est précise mais la ventouse à son extrémité peut parfois avoir des ratés, compliquant lourdement la préhension des pièces pour leurs déplacements. De plus, son temps de fonctionnement est supérieur à notre dernier outil et celui que nous utiliserons : L'électro-aimant. Ce dernier peut être activé ou désactivé de manière instantanée. En plus d'être précis, sa fiabilité de préhension est très satisfaisante. Il peut se glisser entre les pièces sans les toucher afin d'extirper des pièces entourées de la meilleure des façons. Son utilisation requiert néanmoins des pièces spéciales qui ont une rondelle en métal au-dessus afin que l'aimant puisse les attirer.



Figure 12 : Prototypes de Fou et Pion (v1)

Nous connaissons désormais les mesures des pièces ainsi que le type de saisie que le robot va effectuer. Nous utilisons une rondelle en acier zingué que nous fixons sur le dessus de la pièce afin que cette dernière soit magnétiquement soulevable et transportable. Ci-contre le schéma final des pièces sous Fusion 360 :

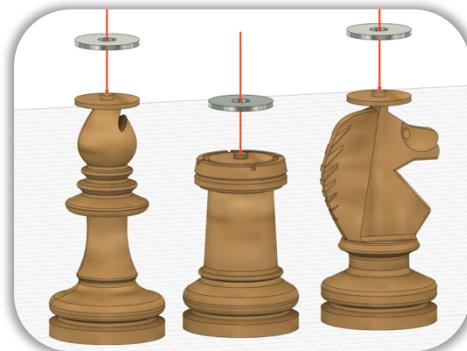


Figure 11 : Schéma 3D final des pièces

2.3 Conclusion

Ce chapitre a mis en évidence l'ensemble des étapes et des choix techniques qui ont conduit à la réalisation concrète de l'échiquier et de ses pièces : analyse des contraintes mécaniques du robot, définition des dimensions selon les préconisations de la FIDE, découpage du plateau en quatre segments optimisés pour l'Ultimaker S5 et sélection d'adhésifs et d'outils de préhension adaptés. Grâce à un processus itératif de modélisation sous Fusion 360, d'impression résine et de tests de prise et de stabilité, nous disposons désormais d'un matériel fiable, à la fois cohérent visuellement et parfaitement calibré pour le bras Niryo Ned 2. L'obtention de ce workflow va nous permettre de réaliser d'autres tâches qui en dépendent comme la reconnaissance de l'échiquier dans la partie suivante ou la reconnaissance des pièces, sans oublier le déplacement physique des pièces par programmation python.

Chapitre 3

3. Détection de l'échiquier en temps réel

Il est important de pouvoir détecter l'échiquier et ses attributs en temps réel pour diverses raisons. La première étant que le robot doit pouvoir connaître la position de chaque case en 6-DOF pose (On parle généralement de la pose d'un objet, c'est-à-dire de sa configuration rigide dans l'espace. Elle regroupe ses trois coordonnées de position x, y, z et ses trois angles d'orientation *pitch*, *yaw*, *roll* et se formalise souvent par une matrice de transformation). Par exemple la case e4 sera liée à une 6-DOF ($x, y, z, pitch, yaw, roll$).

3.1 Le Workspace

Niryo définit des « espaces de travail » (workspaces) qui délimitent une zone opérationnelle par la pose de quatre cibles visuelles (les Niryo Markers) aux coins du périmètre. Cette zone correspond à la surface plane sur laquelle le robot peut exécuter ses mouvements, déplacements de l'outil, prises et dépôts d'objets, avec la précision et la sécurité requises. Lors de l'étape de calibration, la vision embarquée identifie les quatre markers pour établir l'origine et l'orientation du repère XY, puis mesure la hauteur Z de référence, garantissant ainsi que tous les points atteignables restent à l'intérieur des limites mécaniques du robot et hors de portée d'éventuels obstacles extérieurs. Grâce à cette approche, on obtient un environnement de travail reproductible, facilement reconfigurable en repositionnant simplement les markers.

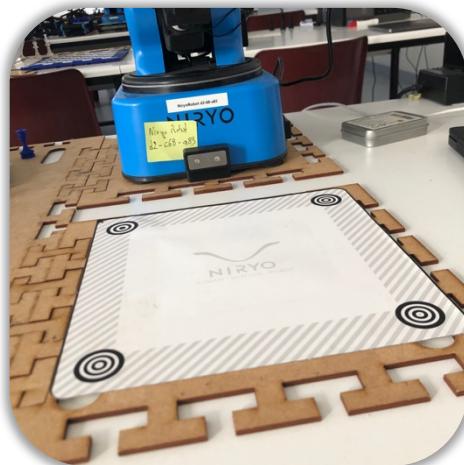


Figure 13 : Exemple du workspace fourni par Niryo

3.2 Les NiryoMarkers

Les NiryoMarkers sont des sortes de cibles qui fonctionnent à la manière d'aprils tags. Ils sont utilisés par Niryo pour délimiter le workspace et calibrer le robot. Le logiciel Niryo Studio, centralise la calibration, la simulation et le pilotage du bras robotique, simplifiant la mise en place d'espaces de travail et le contrôle des trajectoires. Il utilise des fonctions de [l'API pyniryo](#) faisant passerelle entre ROS2 (RobotOS2 est un middleware qui fournit une couche d'abstraction des communications entre les différents modules du robot (capteurs, actionneurs, algorithmes) pour faciliter son développement) et python pour faire fonctionner le robot. Ces cibles par lot de 4 ne sont pas toutes identiques, l'une d'entre elles diffère visuellement et permet de calculer l'orientation du workspace et d'attribuer un ordre aux cibles de l'espace de travail.

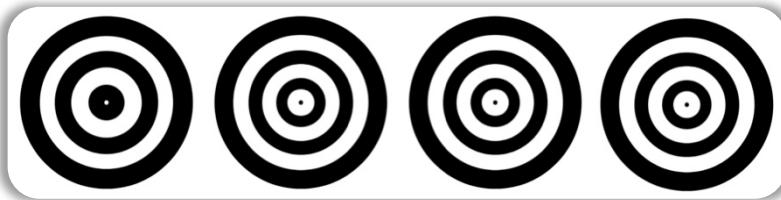


Figure 14 : Cibles Niryo (NiryoMarkers)

3.3 Détection des cibles

Nous avons vu que le logiciel Niryo Studio permet la détection des cibles en passant par l'API pyniryo. Cette dernière est disponible sur le [GitHub](#) de l'entreprise et ouvert à tous. Nous pouvons donc récupérer le code fourni par Niryo pour détecter en python les cibles du workspace. Le module NiryoMarkers sur GitHub implémente en un pipeline de traitement d'images bâti autour d'OpenCV et NumPy : l'image est d'abord binarisée par seuillage, puis les contours détectés sont approximés par des cercles (`cv2.minEnclosingCircle`) filtrés selon leur rayon pour isoler des `PotentialMarkers`, lesquels sont fusionnés s'ils sont suffisamment proches pour former de véritables marqueurs valides (au moins trois cercles et rayon minimal). Chaque marqueur est ensuite identifié ("A" ou "B") via la moyenne d'intensité dans une petite zone centrée, et l'ensemble des quatre marqueurs est ordonné et validé par une transformation de perspective (`cv2.getPerspectiveTransform`) dont la matrice dont le déterminant, proche de 1, garantit la configuration la moins déformée.

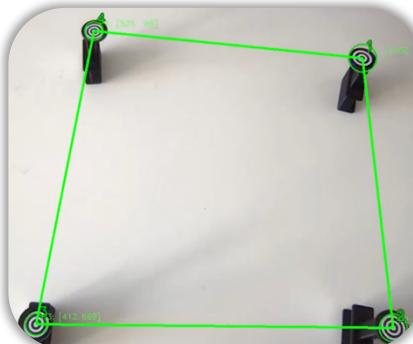


Figure 15 : Exemple de détection de cibles en python

3.4 Calcul de la position de l'échiquier

Nous savons dorénavant détecter les cibles que fournit Niryo. Nous devons alors trouver un moyen de simuler un échiquier digital à partir de ces cibles. Pour créer un échiquier numérique à partir des quatre NiryoMarkers détectés, on procède ainsi : on identifie d'abord les coins TL, TR, BR, BL et on trace les segments bordant l'échiquier : $TL \rightarrow TR \rightarrow BR \rightarrow BL$. Chaque segment est ensuite divisé en huit intervalles égaux (paramètre $t = \frac{i}{8}$ pour $i = 0 \dots 8$), ce qui définit les lignes verticales du damier en reliant le point $TL + t$ au point $BL + t$. De la même façon, on génère les lignes horizontales en interpolant entre $TL \rightarrow BL$ et $TR \rightarrow BR$. L'intersection de ces axes donne les 81 sommets du quadrillage et les 64 cases de l'échiquier.

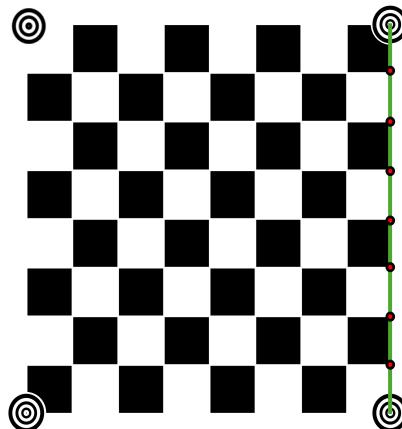


Figure 16 : Schéma du calcul de l'échiquier à partir des NiryoMarkers

Nous pouvons alors utiliser le code python de la détection des markers pour générer cet échiquier en créant deux fonction python :

`extract_img_markers_with_margin` : repère d'abord les quatre NiryoMarkers dans l'image, calcule la transformée de perspective qui redresse et normalise la vue du plateau dans un carré de taille connue (avec une petite marge car les cibles sont en dehors de l'échiquier), et renvoie l'image « warpée » prête à être découpée.

`get_cell_boxes` : subdivise cette image redressée en un quadrillage 8×8 homogène en calculant, pour chaque rangée et colonne, les coordonnées pixel des coins de la case, et assigne à chacune son nom d'échiquier (de “a8” à “h1”), en gérant éventuellement l'inversion en fonction de la couleur des pièces que le robot joue.

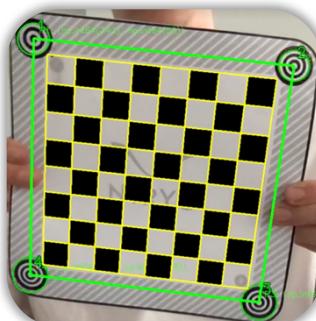


Figure 17 : Démonstration du calcul de l'échiquier en temps réel

3.5 Conclusion

En somme, cette démarche de détection et de modélisation numérique de l'échiquier constitue un point important de notre interaction robot-humain dans ChessBot. À partir des NiryoMarkers, nous calibrons l'espace de travail du bras et redressons la vue du plateau pour en extraire un quadrillage précis de 8×8 cases, chacune identifiée par son nom (a1–h8) et localisée en pixels. Cette acquisition en temps réel, couplée à la génération de cases, permet non seulement de cartographier précisément la position de chaque case dans l'espace, mais aussi de préparer le terrain pour la phase suivante : la détection des pièces et la commande robotique des déplacements vers ces emplacements prédéfinis. Ainsi, le robot dispose d'une représentation numérique du plateau parfaitement alignée avec son repère physique, permettant fiabilité lors des prochaines étapes.

Chapitre 4

4. Programmation du robot Niryo Ned 2

Cette partie vise à expliquer comment le robot est contrôlé à distance à partir d'un ordinateur. Il est important de comprendre comment le robot fonctionne et quels mécanismes ce dernier utilise. Comment construire petit à petit nos fonctions pour prendre en charge les différents coups que proposent les échecs ?

4.1 L'API pyniryo

L'API pyniryo joue un rôle central dans le développement de ChessBot : elle fournit en Python une interface de haut niveau pour piloter le bras Niryo Ned 2, orchestrer la calibration des workspaces (via la détection des NiryoMarkers), et exécuter les séquences de mouvements, de capture et de déplacements. Grâce à ses fonctions abstraites (connexion, commande des axes, contrôle des outils comme l'électro-aimant ou la caméra Vision Set), nous pouvons par la suite intégrer plus facilement nos algorithmes de vision et de planification de coups sans plonger dans les détails bas-niveau du protocole ROS2. En effet, la programmation avec ROS2 se révèle plus complexe que l'utilisation directe de Python car ROS2 n'est pas qu'une bibliothèque. Il faut composer un réseau de nœuds asynchrones qui communiquent par publications/souscriptions à des topics, services ou actions. En résumé, pyniryo nous offre l'outillage indispensable pour passer du prototypage logique et visuel à l'action robotisée fiable et répétable, tout en conservant la flexibilité d'un développement en Python.

4.1.1 Le pose object

Le `PoseObject` de l'API pyniryo est une représentation 6-DOF de la position et de l'orientation de l'outil du Niryo Ned 2 dans l'espace cartésien : il regroupe les coordonnées linéaires `x`, `y`, `z` (en mètres) et les angles `roll`, `pitch`, `yaw` (en radians) qui décrivent la rotation autour de chaque axe. On l'utilise dès qu'il faut indiquer au robot un point précis dans son workspace, par exemple pour positionner la pince ou l'électro-aimant au-dessus d'une case d'échiquier ou pour définir un trajet en espace libre. À la différence du `JointsPosition`, qui stocke simplement un tableau d'angles propres à chacun des six servomoteurs, le `PoseObject` permet de commander directement le bras en coordonnées cartésiennes et de déléguer à l'ordonnanceur cinématique le calcul des angles de chaque articulation nécessaires pour atteindre cette pose.

4.2 Quelques contraintes

4.2.1 Récupération des poses de calibration

Afin que le robot puisse se déplacer correctement au-dessus des cases pour la saisie des pièces, nous devons récupérer la position (x, y, z) des cibles de calibration. De ce fait, comme nous connaissons la distance entre deux cases, il sera possible de calculer toutes les positions de ces dernières dans l'échiquier.

En revanche, la fonction de l'API qui nous permet de récupérer les positions des cibles de calibration n'a pas été implémenté.... Il faut donc trouver un moyen de récupérer ces positions pour définir des `PoseObjects` pour chaque case. Pour ce faire, nous créons la fonction `get_workspace_poses_ssh` nous permettant, en une seule commande, de récupérer directement depuis le robot les quatre poses 6-DOF correspondant aux NiryoMarkers définissant notre workspace. Concrètement, elle établit une connexion SSH/SFTP vers l'adresse IP du Niryo Ned 2 (par défaut "10.10.10.10" avec les identifiants "niryo"/"robotics"), lit le fichier JSON de l'espace de travail nommé (par exemple `ChessBoard.workspace`) dans le répertoire `~/niryo_robot_saved_files/niryo_robot_workspaces/`, puis en extrait la liste des quatre positions stockées sous forme de paires position/orientation. Chaque paire est convertie en un `PoseObject` (x, y, z, roll, pitch, yaw), et la fonction renvoie un tuple ordonné de ces quatre objets.

```
def get_workspace_poses_ssh(
    hostname: str = "10.10.10.10",
    username: str = "niryo",
    password: Optional[str] = "robotics",
    key_filepath: Optional[str] = None,
    workspace_name: str = "ChessBoard"
) -> Tuple[PoseObject, PoseObject, PoseObject, PoseObject]:
    """
    Se connecte en SSH/SFTP au robot, lit
    | ~/niryo_robot_saved_files/niryo_robot_workspaces/{workspace_name}.workspace
    et retourne un tuple de 4 PoseObject(x,y,z,rx,ry,rz).
    """

```

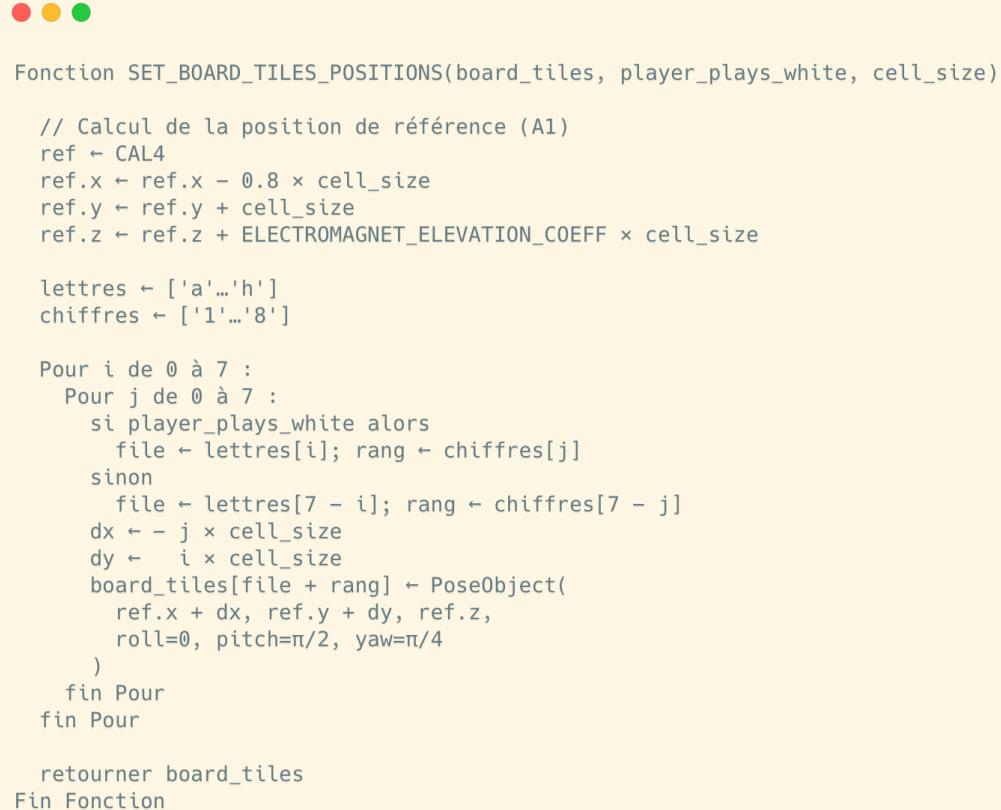
Figure 18 : Définition de la fonction `get_workspace_poses_ssh()`

4.3 Configuration et initialisation du robot

Avant le début des déplacements et logique du jeu, nous devons mettre en place un bloc d'initialisation pour que le Niryo Ned 2 soit opérationnel. On commence par créer une instance Python de la classe `NiryoRobot` en lui passant l'adresse IP du robot, puis on réinitialise tout signal de collision éventuellement présent (`clear_collision_detected`) afin d'éviter les blocages intempestifs. L'appel à `calibrate_auto()` déclenche la séquence de calibration automatique des six axes, garantissant une précision optimale des mouvements. Ensuite, on configure l'outil électromagnétique via `setup_electromagnet` en spécifiant le numéro de broche GPIO auquel il est connecté. Enfin, pour affiner la détection visuelle future des pièces, on règle les paramètres de la caméra embarquée : luminosité, contraste et saturation afin d'obtenir un rendu d'image suffisamment net et contrasté pour nos algorithmes de vision.

4.4 Calcul des poseObject pour chaque case

Comme énoncé précédemment, nous devons attribuer pour chaque case une position (x,y,z). Pour ce faire, nous créons la fonction `set_board_tiles_positions()` qui prend en paramètres la taille des cases et la couleur jouée par le robot. En effet, la numérotation des cases va changer en fonction de l'orientation du plateau ou la couleur jouée par le robot. Il faut donc adapter les positions 6-DOF pour les cases. Ci-dessous l'algorithme en pseudo-code :



```
Fonction SET_BOARD_TILES_POSITIONS(board_tiles, player_plays_white, cell_size)

// Calcul de la position de référence (A1)
ref ← CAL4
ref.x ← ref.x - 0.8 × cell_size
ref.y ← ref.y + cell_size
ref.z ← ref.z + ELECTROMAGNET_ELEVATION_COEFF × cell_size

lettres ← ['a'...'h']
chiffres ← ['1'...'8']

Pour i de 0 à 7 :
    Pour j de 0 à 7 :
        si player_plays_white alors
            file ← lettres[i]; rang ← chiffres[j]
        sinon
            file ← lettres[7 - i]; rang ← chiffres[7 - j]
        dx ← - j × cell_size
        dy ← i × cell_size
        board_tiles[file + rang] ← PoseObject(
            ref.x + dx, ref.y + dy, ref.z,
            roll=0, pitch=π/2, yaw=π/4
        )
    fin Pour
fin Pour

retourner board_tiles
Fin Fonction
```

Figure 19 : Algorithme d'initialisation des poses des cases de l'échiquier

4.5 Fonctions de capture et déplacement

Nous avons désormais les positions des cases de l'échiquier associées à des PoseObjects dans un dictionnaire. Grâce à ce dictionnaire, nous pouvons déplacer le bras robot au-dessus de n'importe quelle case. À partir de cela, il nous faut créer des fonctions de prise et dépôt de pièces.

4.5.1 Fonctions de prise et dépôt

`pickup_with_electromagnet`, `place_with_electromagnet` et `capture_piece` sont trois fonctions que nous créons encapsulent toute la logique de préhension et de dépôt avec notre électro-aimant, car l'API pyniryo ne propose pas de primitives toutes faites pour cet outil (contrairement à la pince). Pour chaque action, on remonte d'abord l'outil à une hauteur sûre

calculée à partir de la taille spécifique de la pièce (`PIECE_HEIGHTS[piece_type]`) et d'un multiple de `CELL_SIZE`, on active (ou désactive) ensuite l'électro-aimant, puis on descend et remonte de la même distance pour assurer une prise (ou une dépose) propre. Le passage du `piece_type` est essentiel : chaque modèle de pièce a une hauteur différente, qu'il faut prendre en compte pour positionner correctement l'aimant au-dessus et obtenir une force d'adhérence suffisante sans heurter ni rayer les autres pièces.

```
def pickup_with_electromagnet(robot, pick_pose, piece_type):
    h = PIECE_HEIGHTS[piece_type]
    p = copy.copy(pick_pose)
    p.z += h + 3*CELL_SIZE
    robot.move_pose(p)
    robot.activate_electromagnet(ELECTROMAGNET_PIN)
    robot.shift_pose(RobotAxis.Z, -3*CELL_SIZE)
    robot.shift_pose(RobotAxis.Z, 3*CELL_SIZE)
```

Figure 20 : Exemple de la fonction de prise

4.5.2 Exécution principale des coups

Nous créons la fonction `play_move` qui orchestre l'exécution de n'importe quel coup d'échecs : déplacement simple, prises et roques, en combinant les primitives de préhension et de dépôt que nous avons définies. Elle prend en entrée le robot, le coup à jouer `from_sq`, `to_sq`, `move_type` ainsi que deux dictionnaires de poses pour les cases du plateau et du buffer, et l'état courant de celui-ci. Selon `move_type`, elle traite d'abord les quatre variantes de roque en déplaçant tours et roi dans l'ordre approprié, puis gère les coups normaux en saisissant la pièce à l'origine et en la déposant sur sa destination. Pour les prises, elle repère une case libre dans le buffer (zone délimitée à côté du robot), y stocke la pièce capturée, met à jour l'état du buffer, puis déplace la pièce attaquante vers la case désormais vacante.

```
● ● ●

Fonction PLAY_MOVE(robot, from_sq, to_sq, move_type, tiles_pos, buffer_pos, buffer_state)

Si move_type est un roque (0-0, 0-0-0, o-o, o-o-o) alors
    Pour chaque (orig, dest) dans la liste des déplacements du roque
        pickup_with_electromagnet(robot, tiles_pos[orig], type_de_piece(orig))
        place_with_electromagnet(robot, tiles_pos[dest], type_de_piece(orig))
    Retourner

Sinon si move_type est None (coup normal) alors
    pickup_with_electromagnet(robot, tiles_pos[from_sq.case], from_sq.piece)
    place_with_electromagnet(robot, tiles_pos[to_sq.case], from_sq.piece)
    Retourner

Sinon (prise)
    // 1. trouver une case libre dans le buffer
    buffer_tile ← première clé dans buffer_state avec valeur None
    Si pas de buffer_tile alors erreur "Buffer plein"
    // 2. stocker la pièce capturée
    capture_piece(robot,
                  tiles_pos[move_type],
                  buffer_pos[buffer_tile],
                  piece_type(to_sq.piece))
    buffer_state[buffer_tile] ← to_sq.piece

    // 3. déplacer la pièce attaquante
    pickup_with_electromagnet(robot, tiles_pos[from_sq.case], from_sq.piece)
    place_with_electromagnet(robot, tiles_pos[to_sq.case], from_sq.piece)
    Retourner

Fin Fonction
```

4.6 Conclusion

En définitive, ce chapitre détaille la chaîne logicielle qui nous permet de déplacer le Niryo Ned 2 depuis un ordinateur, de la calibration automatique jusqu'à l'exécution des différents coups d'échecs. Nous nous appuyons sur l'API pyniryo pour abstraire les communications ROS2 et accéder à des objets de haut niveau comme le `PoseObject`, tandis qu'une fonction dédiée (`get_workspace_poses_ssh`) récupère en toute transparence les quatre repères de workspace stockés sur le robot. Après avoir initialisé l'appareil (calibration, électro-aimant, réglages caméra), nous calculons dynamiquement les positions 6-DOF de chaque case avec `set_board_tiles_positions`, puis encapsulons les séquences de prise et dépôt magnétiques dans des fonctions génériques `capture_piece`, `pickup_with_electromagnet`, `place_with_electromagnet`.

Enfin, `play_move` orchestre le déplacement, la prise et le roque, garantissant une unification pour tous les types de coups. Cette architecture constitue une fondation robuste et évolutive de notre projet ChessBot.

Chapitre 5

5. Détection des pièces

Dans ce chapitre, nous allons aborder la méthode utilisée pour détecter les pièces et retranscrire leur positions de manière logiciel. Cette partie fait appel à des connaissances en vision pour ordinateur et constitue un point clé de ce projet. En effet, si l'on n'est pas capables de récupérer la position en temps réel des pièces sur l'échiquier, nous ne pouvons pas actualiser l'avancée du jeu. Dans ce cas, il nous sera donc impossible d'utiliser le modèle par la suite afin d'indiquer quels coup le robot doit jouer. Dans un premier temps, il est question de comprendre comment est structuré l'élément qui accueille l'état d'une partie à un instant t . Par la suite, nous verrons en détail le processus de détection de pièces et de transcription sur l'échiquier. Il est donc essentiel de communiquer avec Thiziri pour savoir quelle structure elle utilise pour simuler ses parties. En analysant son code, on observe qu'elle utilise un objet de type `board`, objet qui sera expliqué dans la partie suivante.

5.1 L'objet `board()`

L'objet `board` encapsule l'état complet d'une partie d'échecs : il initialise une matrice 8×8 de chaînes de caractères pour représenter la position de chaque pièce (minuscules pour les pièces noires, majuscules pour les blanches) selon la configuration standard, puis stocke toute une série de compteurs et de marqueurs pour suivre l'avancement du jeu. On y trouve le nombre total de coups joués, le décompte de l'avancée des pions pour la règle des cinquante coups, les occurrences de répétition pour chaque camp, ainsi que les droits de roque (via des compteurs de mouvements pour les tours et les rois) et la possibilité de prise en passant (indice de colonne et tour). Des attributs de copie permettent de sauvegarder puis restaurer l'état à volonté, et un indicateur `player` définit à qui revient le trait. Grâce à cette structure, toutes les règles spéciales (roque, en passant, répétitions, progression sans prise) peuvent être gérées et l'historique des déplacements enregistré pour un retour en arrière ou une analyse ultérieure.

5.2 Processus de détection

5.2.1 Prospection

Le processus de détection s'accompagne au préalable d'une phase d'étude. Comment faire en sorte de détecter les pièces avec la meilleure précision possible ?

Pour capturer la position des pièces de l'échiquier, il est important de prendre en compte quelques contraintes : la première est le risque d'occultation. En effet, si la caméra capture une photo de l'échiquier de face, il est possible qu'une pièce soit cachée par une autre se situant devant cette dernière.



Figure 21 : Principe d'occultation du fou par la reine

Pour pallier ce problème, nous optons pour placer la caméra au-dessus de l'échiquier en vision d'oiseau. Cette position permet à la fois une capture globale des pièces de l'échiquier tout en laissant la place nécessaire au joueur pour qu'il puisse déplacer ses pièces facilement. Cette position crée néanmoins un nouveau problème, la prise d'image par le dessus rend difficile la distinction des pièces. En effet, avec la rondelle au-dessus des pièces, ces dernières se ressemblent confortant une homogénéité des données. La capacité à séparer clairement les classes dans l'espace des caractéristiques est cruciale en machine learning : elle détermine la facilité avec laquelle un modèle peut tracer une frontière de décision fiable. Des données bien séparables réduisent le risque de surapprentissage et améliorent la généralisation, tandis qu'un manque de séparabilité invite à enrichir ou transformer les données (sélection de caractéristiques, projection non linéaire, etc.) pour rendre le problème plus tractable.

Pour remédier à cette faible séparabilité des données, nous optons pour l'ajout d'autocollants de couleur différentes pour chaque type de pièces. Les couleurs sont codées via un triplet de composantes (R,G,B) généralement sur 8 bits chacune, ce qui les place dans un espace tridimensionnel où la distance euclidienne ΔE entre deux points RGB quantifie leur différence perceptuelle. Par exemple, les six « primaires » que l'on peut choisir rouge (255, 0, 0), jaune (255, 255, 0), vert (0, 255, 0) etc. occupent des positions bien séparées dans cet espace. Cette distance vaut par exemple entre rouge et vert :

$$\sqrt{255^2 + 255^2} \approx 360$$

Ces grands écarts garantissent que, même avec un simple seuillage ou un petit réseau de neurones, on peut facilement regrouper chaque pixel autour de son centre de couleur et séparer nettement ces six teintes sans confusion.



Figure 22 : Représentation des pièces avec étiquettes de couleur

5.2.2 Choix du modèle et Dataset

Nous avons désormais les pièces finalisées avec leurs étiquettes. Nous pouvons élaborer un jeu d'images prises avec la caméra du robot qui nous servira à entraîner un modèle détection. Nous avons choisi d'utiliser le modèle YOLOv8S (Small). C'est un puissant algorithme de détection d'objets développé par [Ultralytics](#). Son approche est assez novatrice car elle permet d'effectuer simultanément la détection et la classification d'objets en une seule passe à travers un réseau de neurones convolutifs, combinant vitesse et précision pour des applications en temps réel. Son architecture en pipeline est associée à des mécanismes de régions d'intérêts ce qui est différent des CNN classiques. L'image d'entrée est dans un premier temps divisée en une grille de cellules. Chacune de ces cellules est responsable de prédire les coordonnées des bounding boxes pour les objets détectés ainsi que leurs probabilités d'appartenance à différentes classes.

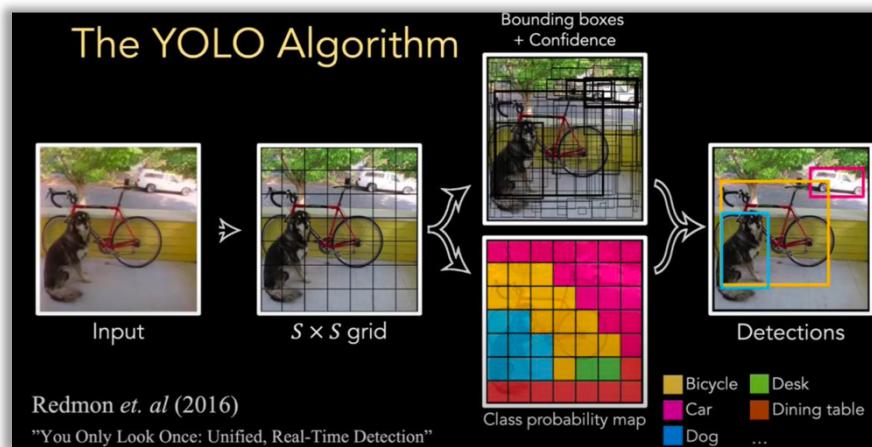


Figure 23 : Principe de fonctionnement de YOLO

Pour l'entraînement de ce modèle, il est nécessaire d'annoter les images prises par le robot. Pour ce faire, nous utilisons un logiciel OpenSource d'annotation : [LabelStudio](#).



Figure 24 : Annontation des pièces d'échecs

Une fois cette tâche effectuée, on peut séparer notre dataset en deux parties : une partie comprenant 80% des images pour l'entraînement du modèle et les autres 20% pour la validation et test. Pour l'entraînement, j'utilise la puissance GPU des instances [Google colab](#) ce qui me permet d'accélérer sa vitesse. Le dataset se compose d'environ 300 images et s'entraîne sur 128 epochs. Après entraînement, on sauvegarde les poids du modèle pour l'utilisation réelle du robot.

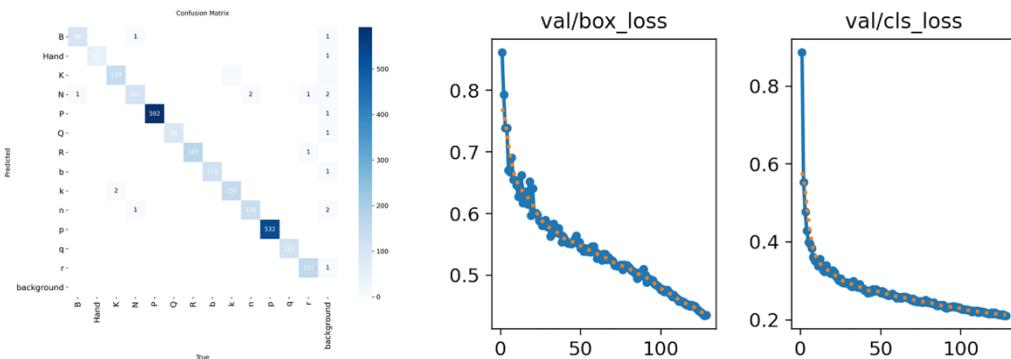


Figure 25 : métriques d'évaluation du modèle

Les résultats sont plutôt bons, le modèle fait peu d'erreurs et confusions. On note quand même que plus d'éPOCHS pourraient être bénéfiques à la précision du modèle car la perte en validation continue de descendre et ne semble pas tendre vers une valeur.

5.3 De l'image aux positions des pièces sur l'échiquier

Désormais, il nous faut fusionner les deux résultats de détection : l'échiquier et les pièces pour reconstituer la position complète sur les 64 cases. D'une part, nous disposons des coordonnées pixel de chaque case, obtenues via la grille 8×8 générée sur l'image redressée. D'autre part, le modèle YOLOv8 renvoie pour chaque pièce une boîte englobante (bounding box) définie par ses coordonnées. Il suffit alors de calculer l'aire d'intersection entre chaque bounding box et chaque case : la case dont la surface recouvre le plus grand pourcentage de la boîte correspondra à la pièce détectée. En appliquant ce principe à toutes les détections, on remplit une matrice 8x8 où chaque élément contient la classe de la pièce présente (ou reste vide), reproduisant l'état du plateau. Pour ce faire, nous créons une fonction python `board_state_from_yolo` qui convertit les résultats bruts de YOLO (positions, classes et confiances des bounding boxes) en une matrice 8x8 lisible du plateau d'échecs. Pour chaque détection, elle calcule l'aire d'intersection avec chaque case (définie par `cell_boxes`) et choisit la case maximisant ce recouvrement ; elle ignore les classes "Hand" et ne conserve qu'une détection par case, celle à la plus haute confiance. Si le joueur n'est pas blanc, elle remet à l'endroit les noms de case pour retrouver l'ordre standard (a1-h8). Le résultat est une liste de huit listes de huit chaînes ('r', 'P', ' ', etc.) qui restitue l'état complet du plateau :

```
[ 'r', 'n', 'b', 'q', 'k', 'b', 'n', 'r'],
[ 'p', 'p', 'p', 'p', 'p', 'p', 'p', 'p'],
[ ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' '],
[ ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' '],
[ ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' '],
[ ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' '],
[ ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' '],
[ 'P', 'P', 'P', 'P', 'P', 'P', 'P', 'P'],
[ 'R', 'N', 'B', 'Q', 'K', 'B', 'N', 'R']
```

Figure 26 : Output de la fonction
`board_state_from_yolo()`

5.4 Détection du camp du joueur

Déterminer quelle couleur va être jouée par le robot ou le joueur est crucial pour la numérotation des cases de l'échiquier. En effet, si le joueur joue les blancs, la cases en bas à gauche sera 'a1' ; par contre, si le joueur joue les noirs, la case en bas à gauche sera 'h8'. Nous devons donc élaborer une technique de détection du camp du joueur en utilisant la vision par ordinateur. Nous savons récupérer les bounding boxes de toutes les pièces de l'échiquier. Ces pièces sont soit en majuscule pour les blancs ou minuscules pour les noirs. Dans ce cas nous pouvons différencier la couleur des pièces entre elles ainsi que leur positions dans l'image. Nous connaissons également la position du marker n°1 dans l'image se situant du coté robot et donc, à partir de ces informations, nous pouvons mesurer la distance à chaque pièces et retenir la distance minimale. Si cette distance minimale fait lien entre la cible et une pièce majuscule,

alors les pièces blanches sont du côté du robot, signifiant que le joueur joue les noirs. Inversement, le joueur jouera les blancs.

```

● ● ●

Fonction PLAYER_HAS_WHITE(corners, yolo_result)
    // Récupérer le point de référence : centre du marqueur top-left
    marker_tl ← centre de corners[0]

    // Initialiser les distances minimales
    min_white ← +∞
    min_black ← +∞

    // Pour chaque détection retournée par YOLO
    Pour chaque (box, cls_idx) dans yolo_result :
        name ← yolo_result.names[cls_idx]
        Si name.lower() = "hand" alors
            Continuer

        // Déterminer la couleur par la casse du nom
        Si name est majuscule alors couleur ← white
        Sinon si name est minuscule alors couleur ← black
        Sinon
            Continuer

        // Calculer le centre de la bounding box
        centre_box ← ((x1+x2)/2, (y1+y2)/2)

        // Calculer la distance au marqueur
        dist ← distance_euclidienne(marker_tl, centre_box)

        // Mettre à jour la distance minimale
        Si couleur = white et dist < min_white alors min_white ← dist
        Si couleur = black et dist < min_black alors min_black ← dist
    Fin Pour

    // Si une des couleurs n'a pas été détectée, on ne peut pas décider
    Si min_white = +∞ ou min_black = +∞ alors
        Retourner None

    // Le joueur est blanc si la pièce blanche la plus proche est plus proche que la noire
    Retourner (min_white < min_black)
Fin Fonction

```

Figure 27 : Algorithme de la fonction player_has_white()

5.5 Conclusion

En définitive, cette chaîne de détection combine judicieusement vision géométrique et apprentissage profond pour passer d'une simple image à une représentation complète et structurée de l'échiquier. Après avoir calibré et redressé la vue du plateau grâce aux NiryoMarkers, nous avons résolu la problématique de la séparabilité des pièces en ajoutant des autocollants colorés et formé un modèle YOLOv8S pour localiser et classifier chaque pièce. La fonction board_state_from_yolo fusionne ensuite ces détections avec la grille 8×8 pour remplir la matrice d'état, tandis que player_has_white identifie automatiquement le camp du joueur. Ce dispositif garantit une mise à jour fiable et temps réel de la position, indispensable pour alimenter la logique de décision et piloter le robot lors de chaque coup.

Chapitre 6

6. Intégration de l'IA d'apprentissage

Cette partie clôture la pipeline du projet. Elle vise à l'intégration de l'algorithme de génération du prochain coup élaboré par Thiziri et à la création de la boucle du jeu encapsulant toutes les fonctions utilitaires vues tout au long des derniers chapitres. La prospection du format des données est essentielle pour une bonne mise en place du pipeline permettant le jeu en continu du robot contre le joueur.

6.1 Calcul du prochain coup

La fusion entre les fonctions du modèle de prédiction du prochain coup s'effectue en python par import des modules du projet de Thiziri. Les fonctions fournies par ces modules permettent de donner en entrée un objet `board()`. Cet objet est ensuite utilisé dans la fonction `UCT_search(b, num_reads=NUM_READS, net=net)` qui lance une recherche Monte-Carlo à base de l'algorithme UCT (Upper Confidence bounds applied to Trees) sur la position courante `b`. La fonction retourne ensuite `best_idx` qui est l'indice du coup à jouer pour le robot. Il suffit ensuite d'utiliser la fonction `do_decode_n_move_pieces(b, best_idx)`, qui va traduire cet index en un coup concret (par exemple en passant de `0→e2e4`, `1→d2d4`, etc.) et mettre à jour l'objet `b` avec la nouvelle position.

6.2 Mise en place de la pipeline

6.2.1 Préparation

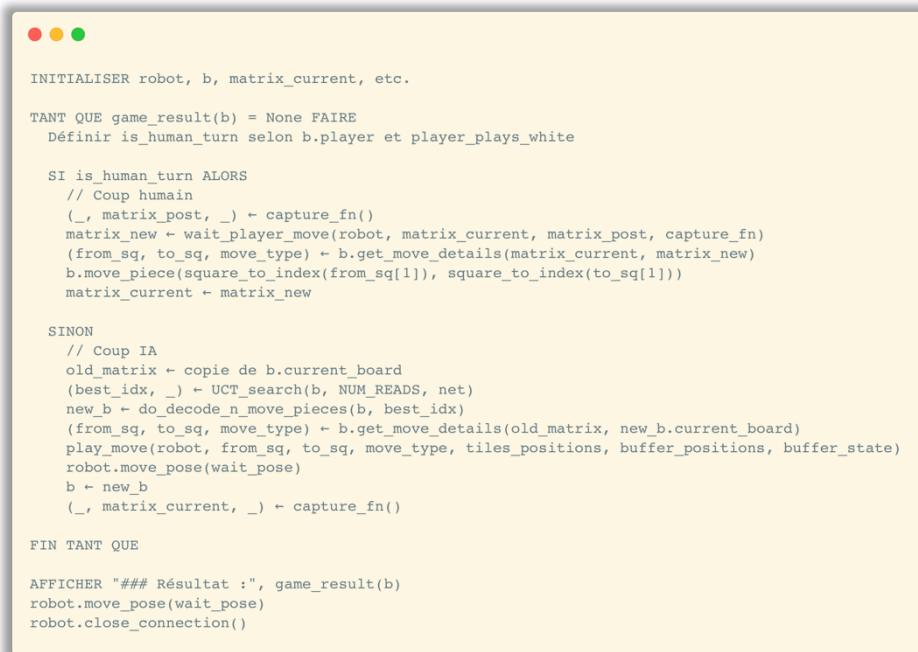
Cette phase précède la boucle de jeu. Elle permet d'initialiser tous les paramètres et fonctions d'utilitaires nécessaires pour le déplacement, logique du robot etc. Dans un premier temps, nous importons tous les modules et bibliothèques nécessaires au bon fonctionnement des fonctions (numpy, ultralytics, torch etc.). Ensuite, on définit toutes les constantes telles que l'adresse IP du robot, le numéro de broche sur lequel est branché l'électo-aimant, la taille des cases de l'échiquier, la hauteur des pièces et de l'échiquier. On définit par la suite toutes les fonctions de déplacement du robot et logique. En débutant le programme, on effectue l'initialisation du robot (connexion, calibration, setup de l'aimant, chargement des poids du modèle YOLO, calibration visuelle, initialisation de l'objet `board()`, recherche des pièces et du camp du joueur)

6.2.2 Boucle du jeu

La boucle principale de ChessBot incarne la convergence de toutes nos briques logicielles et matérielles : tantôt acteur humain, tantôt artificiel, le système alterne détection vision, mise à jour logicielle et commandes robotiques jusqu'à l'issue de la partie. À chaque itération, on commence par vérifier l'état du jeu (`game_result`) et déterminer à qui revient le trait, selon l'orientation du joueur et l'attribut `b.player`.

Si c'est au tour de l'humain, on capture deux matrices d'état avant et après son coup via `capture_fn`. On attend ensuite que le joueur joue avec `wait_player_move`, on localise la différence entre l'état de l'échiquier avant et après pour en extraire `from_sq`, `to_sq` et `move_type`. On utilise ces variables pour mettre à jour l'objet `board` avec `move_piece`. Dans le cas du coup de l'IA, on clone l'état courant, on lance notre MCTS enrichi du réseau de neurones (`UCT_search`), on décode l'indice du meilleur coup en une nouvelle configuration `new_b` via `do_decode_n_move_pieces`, puis on ordonne au robot d'exécuter physiquement la séquence complète (prise, dépôt, capture éventuelle) en faisant appel à `play_move` et aux fonctions de préhension magnétiques.

Entre chaque mouvement, le bras retourne à une position d'attente (`wait_pose`), garantissant sécurité et lisibilité pour la vision. Après chaque action, qu'elle soit humaine ou robotique, on rafraîchit la représentation du plateau pour préparer le tour suivant. Lorsque `game_result` sort de la boucle, le résultat est annoncé, le robot est ramené en position neutre et la connexion est close. Cette orchestration continue illustre comment la détection d'images, la gestion d'état, la recherche de coups et le pilotage robotique s'imbriquent pour offrir une expérience de jeu autonome.



```

INITIALISER robot, b, matrix_current, etc.

TANT QUE game_result(b) = None FAIRE
    Définir is_human_turn selon b.player et player_plays_white

    SI is_human_turn ALORS
        // Coup humain
        (_, matrix_post, _) ← capture_fn()
        matrix_new ← wait_player_move(robot, matrix_current, matrix_post, capture_fn)
        (from_sq, to_sq, move_type) ← b.get_move_details(matrix_current, matrix_new)
        b.move_piece(square_to_index(from_sq[1]), square_to_index(to_sq[1]))
        matrix_current ← matrix_new

    SINON
        // Coup IA
        old_matrix ← copie de b.current_board
        (best_idx, _) ← UCT_search(b, NUM_READS, net)
        new_b ← do_decode_n_move_pieces(b, best_idx)
        (from_sq, to_sq, move_type) ← b.get_move_details(old_matrix, new_b.current_board)
        play_move(robot, from_sq, to_sq, move_type, tiles_positions, buffer_positions, buffer_state)
        robot.move_pose(wait_pose)
        b ← new_b
        (_, matrix_current, _) ← capture_fn()

    FIN TANT QUE

    AFFICHER "## Résultat :", game_result(b)
    robot.move_pose(wait_pose)
    robot.close_connection()

```

Figure 28 : Algorithme de la boucle du jeu

Conclusion Générale

En clôture de ce projet, nous avons mis en œuvre une chaîne complète combinant vision par ordinateur, apprentissage automatique et commande robotique pour permettre au bras Niryo Ned 2 de disputer une partie d'échecs contre un adversaire humain. À travers la modélisation 3D et l'impression des composants, nous avons d'abord conçu un échiquier et des pièces aux dimensions réglementaires et parfaitement adaptées aux capacités mécaniques du robot. La calibration automatique des workspaces via les NiryoMarkers et la génération d'un quadrillage numérique 8x8 ont posé les bases d'une perception fiable du plateau.

La détection des pièces, renforcée par l'ajout de repères couleur et l'entraînement du modèle YOLOv8 small, nous a ensuite permis de traduire en temps réel l'image de la caméra en une matrice d'état du jeu, tandis que la fonction de détermination du camp assurait une orientation correcte du plateau. Sur cette représentation logicielle, l'algorithme MCTS enrichi par un réseau de neurones a pu sélectionner le coup optimal, que nous avons exécuté physiquement grâce à des fonctions de préhension magnétiques soigneusement calibrées en fonction de la hauteur des pièces.

L'orchestration finale, incarnée par la boucle de jeu, illustre l'intégration de toutes ces briques : détection, simulation, décision et action robotique s'enchaînent jusqu'à l'issue de la partie, offrant une expérience interactive et autonome. Ce démonstrateur témoigne de la synergie entre les parcours SIME et A2IA, et valide les choix techniques réalisés tout au long du stage.

Au cours de ce stage, j'ai consolidé mes compétences en programmation et en résolution de problèmes, notamment grâce à la mise en place d'un workflow Git structuré pour le versioning et la collaboration. J'ai découvert la CAO sous Fusion 360 et les technologies d'impression 3D (FLM et SLA), ce qui m'a permis de prototyper et d'ajuster rapidement des pièces aux tolérances strictes du robot. L'intégration de la vision par ordinateur et de l'API pyniryo m'a familiarisé avec les défis du développement embarqué en temps réel. Enfin, piloter un projet pluridisciplinaire m'a entraîné à planifier les tâches, estimer les charges et adapter les plannings face aux imprévus, renforçant ainsi ma confiance dans la gestion de projets techniques.

Perspectives et pistes d'amélioration

L'échiquier

L'échiquier actuel est un prototype, nous l'avons imprimé rapidement pour se concentrer rapidement sur les fonctions de mouvement et récupération de l'échiquier. Récemment, nous avons réimprimé un échiquier qui ne comporte pas de contours. Cette méthode permet d'élargir les cases de l'échiquier pour atteindre des côtés de 4 cm offrant une expérience de jeu plus confortable. Nous avons également intégré les NiryoMarkers à l'intérieur des cases car après réflexion et optimisation des fonctions, nous n'avons pas besoin de les détecter en continu, mais uniquement d'enregistrer leurs positions avant le début de partie.



Figure 30 : Échiquier final (à gauche) contre prototype (à droite)

Pour le nouvel échiquier, nous avons utilisé l'imprimante 3D K1 Max de Creality. Elle permet une vitesse d'impression très rapide et une finition assez fine. En revanche, nous avons dû imprimer les cases unes par unes et les assembler ensuite via un système de clips (voir capture).

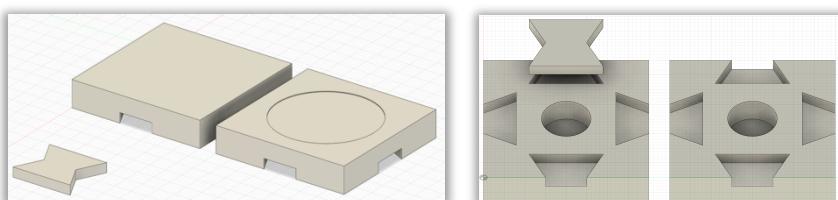


Figure 29 : Cases assemblables

Les pièces

Les pièces sont déplacées par le robot mais ce dernier peut commettre des petites erreurs de déplacement de l'ordre du millimètre. Au bout de plusieurs répétitions, il est possible que la pièce ne soit pas exactement centrée sur la case sur laquelle elle est déplacée. De la même manière, le joueur peut déplacer maladroitement une pièce et mal la placer au centre de la case.

Pour y remédier, nous avons prévu des emplacement en dessous de chaque case et chaque pièce dans lesquels viennent s'insérer des aimants fins. Les pièces seront donc toujours centrées.

Amélioration de la détection des pièces

Le modèle YOLO que nous avons entraîné n'est pas infaillible et peut commettre des erreurs. Ce problème est assez gênant car une matrice de l'échiquier sera générée comportant des emplacement des pièces vides ou erronés. La fonction estimant le coup joué par le joueur ne retournera pas la bonne valeur car plusieurs pièces seront estimées déplacées ou capturée alors qu'un seul coup a été joué. Cela est dû au manque de données pour notre dataset et à la luminosité / reflets que l'environnement apporte. Ces complications peuvent être corrigées de deux façons complémentaires :

- La première étape est d'augmenter le nombre de données pour avoir un dataset permettant au modèle de bien généraliser quel que soit l'environnement autour de lui.
- La deuxième étape est d'appliquer des micros-transformations sur l'image pour l'altérer et obtenir une nouvelle détection. En effectuant la moyenne des positions sur l'échiquier, on pourra obtenir une matrice plus fiable. En revanche cette approche rallonge légèrement le temps de réflexion du robot car elle nécessite de multiples captures photos et micro-déplacements

Amélioration de la pipeline et robustesse aux erreurs

Actuellement, certaines erreurs commises par le robot ou le joueur renvoient une erreur fatale qui coupe le programme et la partie. Par exemple si l'algorithme de vision effectue une fausse détection, la matrice des positions de l'échiquier sera erronée générant une erreur dans la fonction d'estimation du coup, ce qui coupe la partie...

Ces erreurs perturbent la pipeline en coupant le joueur dans son expérience mais peuvent être corrigées en ajoutant des alternatives si ces problèmes apparaissent.

Expérience utilisateur : vers une solution « user friendly »

En dehors des déplacements, le robot possède de nombreuses fonctionnalités qui peuvent être exploitées pour améliorer les conditions de jeu. Par exemple le robot peut émettre des sons à partir de sa base. On peut donc imaginer enregistrer des voix indiquant au joueur ce qu'il faut faire pour lancer une partie ou lui expliquant pourquoi il perd, pourquoi son coup est illégal etc. Ces retours comme la parole ou l'indication de l'état du jeu par l'anneau de led à la base du robot peuvent grandement améliorer l'expérience que l'utilisateur a avec le robot plutôt que de devoir regarder le terminal pour recevoir les informations que génère le robot.

Sources et Sitographie

https://github.com/NiryoRobotics/ned_ros/tree/master

<https://slack.com/intl/fr-fr>

<https://github.com/Errioto/chess-bot>

<https://www.youtube.com/watch?v=r0RspLG260>

<https://www.youtube.com/watch?v=svn9-xV7wjk&t=790s>

<https://www.youtube.com/@NiryoRobotics>

<https://niryorobotics.github.io/pyniryo/v1.2.0-1/api/api.html>

<https://niryorobotics.github.io/pyniryo/v1.2.0-1/api/vision.html>

<https://docs.ultralytics.com/fr/tasks/detect/>

<https://docs.ultralytics.com/fr/modes/train/>

https://simple.wikipedia.org/wiki/Pitch,_yaw,_and_roll

<https://carbon.now.sh>

https://www.youtube.com/watch?v=WKb3mRkgTwg&list=PLrZ2zKOtC_-C4rWfapngoe9o2-ng8ZBr

<https://www.google.com/url?sa=t&source=web&rct=j&opi=89978449&url=https://www.autodesk.com/akn-aknsite-article-attachments/b35ea3d7-5427-47ca-980f-250b1438f1a5.pdf&ved=2ahUKEwjcjafM4v2NAxWYU6QEHWaCO7UQFnoECBoQAQ&usg=AOvVaw37mnP7qVx7ZFhHzerUmGuc>

<https://www.autodesk.com/ca-fr/products/fusion-360/personal>

<https://ultimaker.com>

<https://www.creality.com/pages/download-software>

<https://www.litislab.fr>

<https://www.univ-rouen.fr>