RAPPORT DE PROJET

Bras robotique collaboratif 6 DDL

Commande, automatisation ROS2 et vision par IA pour le pick-and-place autonome

Date de remise : 18 mai 2026

Étudiant(e)s : Arnaud JUNCKER - Theresia SALLOUM - Hashem AL-GAFRI - Daniel BAL

Encadrants : M. Joël FRITSCH

M. Pierre-Paul ZEIL

M. Dominique KNITTEL

RÉSUMÉ / ABSTRACT

Résumé

Ce projet a pour objectif la conception et l'implémentation d'un système autonome de *pick-and-place* basé sur le bras robotique collaboratif igus Rebel 6DOF. Le système intègre une chaîne complète de perception (caméra Intel RealSense D435, réseau de neurones YOLO11) permettant la détection et la localisation 3D d'objets en temps réel. La planification et l'exécution des trajectoires s'appuient sur ROS 2 Humble et MoveIt 2, tandis que le protocole CRI assure le contrôle bas niveau du bras via TCP/IP. Une pince SCHUNK EGP 25 pilotée par sorties numériques complète l'architecture matérielle. Une bibliothèque Python orientée objet a été développée pour centraliser la logique de contrôle. Les résultats démontrent la faisabilité d'un système entièrement automatisé, de la détection visuelle jusqu'au dépôt de l'objet, transformant ainsi le robot Igus en un véritable cobot intelligent, flexible et adapté aux exigences de l'Industrie 4.0.

Abstract

This project aims to design and implement an autonomous pick-and-place system based on the igus Rebel 6DOF collaborative robotic arm. The system integrates a complete perception pipeline (Intel RealSense D435 camera, YOLO11 neural network) enabling real-time 3D object detection and localization. Trajectory planning and execution rely on ROS 2 Humble and MoveIt 2, while the CRI protocol provides low-level control of the arm via TCP/IP. A SCHUNK EGP 25 gripper, controlled via digital outputs, completes the hardware architecture. An object-oriented Python library was developed to centralize the control logic. The results demonstrate the feasibility of a fully automated system, from visual detection to object placement, thereby transforming the standard Igus robot into a truly intelligent and flexible cobot, fully adapted to the requirements of Industry 4.0.

REMERCIEMENTS

Nous tenons en premier lieu à adresser nos sincères remerciements à nos encadrants, M. J. FRITSCH, M. PP. ZEIL et M. D. KNITTEL, pour leur suivi, leurs conseils avisés et la confiance qu'ils nous ont accordée tout au long de ce projet. Leur expertise scientifique a été déterminante dans l'aboutissement de ce travail.

Nos remerciements s'adressent également aux techniciens du Hall de Technologie de la faculté de Physique de l'Université de Strasbourg, M. P. BRAUN, M. L. GODOY et M. M. KOPF. Leur disponibilité et leur accompagnement matériel ont offert un cadre de travail privilégié pour la réalisation des essais expérimentaux.

Enfin, nous souhaitons exprimer notre gratitude à l'entreprise igus GmbH pour la qualité de sa documentation concernant le protocole de communication CRI, ainsi qu'à la communauté open-source, et tout particulièrement au laboratoire AIRLab-POLIMI, pour la mise à disposition de leurs travaux sur l'intégration ROS 2 du bras robotique.

Sommaire

LISTE DES ABRÉVIATIONS

- **CNN :** Convolutional Neural Network *(Réseau de neurones convolutif)*
- **CRI :** Commonplace Robot Interface *(Protocole de communication propriétaire d'igus)*
- **DIO :** Digital Input/Output *(Entrées/Sorties numériques)*
- **DOF :** Degrees of Freedom *(Degrés de liberté)*
- **IK :** Inverse Kinematics *(Cinématique inverse)*
- **MoveIt 2 :** Motion Planning Framework for ROS 2 *(Bibliothèque de planification de mouvement)*
- **QoS :** Quality of Service *(Qualité de service)*
- **ROS 2 :** Robot Operating System 2 *(Système d'exploitation open-source pour la robotique)*
- **TCP :**
  - ***En réseau :*** Transmission Control Protocol *(Protocole de contrôle de transmission)*
  - ***En robotique :*** Tool Center Point *(Point de centre d'outil / Extrémité du bras)*
- **URDF :** Unified Robot Description Format *(Format de description unifié de robot)*
- **XACRO :** XML Macros *(Générateur de macros pour simplifier les fichiers URDF)*
- **YOLO :** You Only Look Once *(Algorithme de détection d'objets en temps réel)*
- **RGB-D :** *Red Green Blue - Depth*
- **FPS :** *Frames Per Second*
- **GPU :** *Graphics Processing Unit*
- **OpenCV :** *Open Source Computer Vision Library*
- **D435 :** *Intel RealSense Depth Camera*
- **mAP :** *mean Average Precision*

1. INTRODUCTION & CADRE DU PROJET (Daniel)

1.1 Contexte général

Le virage de l'automatisation industrielle flexible

L'Industrie 4.0 a marqué un tournant décisif dans les processus de fabrication, imposant une transition d'une production de masse rigide vers une automatisation flexible et reconfigurable. Aujourd'hui, les chaînes de production doivent être capables de s'adapter rapidement à des séries plus courtes, à des changements de produits fréquents et à des environnements de travail dynamiques. Cette nécessité d'agilité pousse les industriels à repenser leurs outils de production pour y intégrer davantage d'intelligence et de capacité d'adaptation en temps réel.

L'essor des cobots au sein des PME

Au cœur de cette transition, les bras robotiques collaboratifs (cobots) s'imposent comme une solution technologique et économique de premier plan, tout particulièrement pour les Petites et Moyennes Entreprises (PME). Contrairement aux robots industriels traditionnels, qui sont lourds, complexes à déployer et nécessitent des cages de sécurité onéreuses, les cobots sont conçus pour partager l'espace de travail avec les opérateurs en toute sécurité. Leur compacité, leur coût d'acquisition plus abordable et leur polyvalence permettent désormais aux PME d'automatiser des tâches répétitives ou à faible valeur ajoutée (comme l'emballage, l'assemblage ou le tri), améliorant ainsi leur compétitivité sans pour autant bouleverser toute leur ligne de production.

Les limites de la programmation classique

Cependant, l'intégration de ces systèmes collaboratifs se heurte encore à un obstacle majeur qui est le temps et la rigidité de la programmation. Traditionnellement, l'apprentissage d'une tâche de manipulation nécessite l'intervention d'un technicien pour enregistrer manuellement chaque point de passage du robot (méthode du *Teach Pendant* ou guidage manuel). Cette approche séquentielle est chronophage et strictement inadaptée aux environnements où les pièces arrivent de manière aléatoire ou non structurée sur un convoyeur ou une table de travail.

Problématique du projet

Face à cette contrainte de rigidité logicielle, il devient indispensable de doter le robot d'une capacité d'analyse et de décision. C'est dans ce contexte que s'inscrit notre projet, articulé autour de la problématique suivante :

> **Comment rendre un bras robotique collaboratif totalement autonome dans l'exécution de tâches de *pick-and-place*, en s'affranchissant de toute programmation manuelle de ses trajectoires ?**

Pour répondre à ce défi, notre démarche consistera à coupler le bras manipulateur à un système de perception par intelligence artificielle (vision 3D) et à un planificateur de mouvements dynamique, permettant ainsi au robot de "voir" son environnement, de localiser les objets et de calculer lui-même la trajectoire optimale pour les saisir.

1.2 Présentation du sujet

Description globale du projet

Notre projet s'appuie sur une base mécanique industrielle préexistante qui est le bras robotique collaboratif igus Rebel 6DOF. Notre travail ne porte donc pas sur la conception mécanique du robot, mais sur le développement et l'intégration d'une surcouche matérielle et logicielle pour transformer ce bras standard en une cellule de *pick-and-place* totalement autonome. L'objectif est de s'affranchir de la programmation manuelle classique. Nous avons conçu un système intelligent où le robot est capable d'analyser **son espace de travail** en temps réel pour repérer un objet cible, de calculer lui-même la meilleure trajectoire pour l'atteindre, et de le déplacer en toute sécurité vers une zone de destination prédéfinie.

Pour accomplir cette tâche complexe et donner cette autonomie au bras igus, notre architecture repose sur l'intégration de quatre briques technologiques fondamentales :

- **1. Vision (Perception) :** Avant de manipuler quoi que ce soit, le robot doit pouvoir "voir" et analyser sa zone d'intervention. Cette première étape est assurée par une caméra 3D (Intel RealSense D435) couplée à un modèle d'intelligence artificielle (YOLO11n). Ce duo permet d'identifier l'objet cible posé sur la table et d'extraire ses coordonnées tridimensionnelles (X, Y, Z) par rapport à la base du robot.

- **2. Planification de trajectoire (Décision) :** Une fois la position spatiale de l'objet connue, le "cerveau" du système prend le relais. À l'aide de l'environnement ROS 2 et de la bibliothèque MoveIt 2, le système calcule la cinématique inverse (IK) et génère dynamiquement le chemin optimal que le bras doit emprunter. Cette planification garantit des mouvements fluides tout en évitant les singularités mécaniques et les collisions avec l'environnement.
- **3. Exécution (Contrôle bas niveau) :** Les trajectoires mathématiques validées sont ensuite traduites en consignes articulaires. Ces commandes sont envoyées aux moteurs du bras igus Rebel 6DOF via une connexion réseau TCP/IP, en utilisant le protocole de communication propriétaire CRI, assurant ainsi un déplacement fluide et sécurisé vers la cible.
- **4. Préhension (Action) :** Une fois le Tool Center Point (TCP) positionné exactement sur l'objet, le système déclenche l'actionneur terminal. Une pince industrielle SCHUNK EGP 25, pilotée par les sorties numériques (DIO) du contrôleur, se referme pour saisir fermement la pièce avant d'amorcer la trajectoire de retour vers la zone de dépose.

Périmètre du projet (Scope) Afin de mener à bien cette preuve de concept dans le temps imparti, il a été nécessaire de définir précisément les limites de notre étude.

Ce qui est traité (In-scope) :

- L'intégration logicielle et matérielle de bout en bout (caméra, IA, ROS 2, robot, pince).
- La détection et la localisation spatiale d'objets posés statiquement dans le champ de vision de la caméra.
- La création d'une architecture logicielle modulaire en Python orienté objet pour coordonner l'ensemble du cycle.
- L'évitement d'obstacles statiques définis virtuellement dans la scène de planification (*Planning Scene* de MoveIt).

Ce qui n'est pas traité (Hors-scope) :

- **Le contrôle par retour d'effort :** Le protocole réseau de IGUS ne remontant pas les valeurs de couple mécanique (Effort) des moteurs en temps réel, la détection physique de collision par la force n'est pas implémentée.

- **L'évitement dynamique :** Le système ne recalcule pas sa trajectoire en temps réel si un obstacle mobile inattendu coupe sa route pendant l'exécution du mouvement.
- **L'optimisation des temps de cycle :** L'objectif prioritaire est la faisabilité, la précision et la fiabilité du mouvement autonome, et non l'atteinte de vitesses de production industrielles à haute cadence.

1.3 Cahier des charges

Exigence	Critère	Valeur cible
Détection objet	Précision YOLO11n	> 80% mAP
Localisation 3D	Erreur de position	< 10 mm
Planification	Temps de calcul IK	< 5 s
Cycle complet	Durée pick-and-place	< 20 s
Fiabilité	Taux de succès	> 90%
Sécurité	Arrêt d'urgence	< 100 ms réaction

Tableau 1 : Cahier des charges

Matériel nécessaire

Pour la mise en place et la validation du projet, nous utilisons le matériel suivant :

- **Robot collaboratif igus ReBeL à six degrés de liberté :** il s'agit du bras robotisé qui sert de base à notre projet et sur lequel nous testons les différentes stratégies de commande et d'automatisation.
- **Gripper Schunk :** monté à l'extrémité du bras, il permet de saisir et de manipuler différents objets dans l'espace de travail du robot.
- **Caméra 3D :** elle est utilisée pour la détection et la localisation des objets dans l'environnement de travail, afin de fournir au robot les informations nécessaires pour adapter sa trajectoire.
- **Objets de test :** une roulette de skateboard est utilisée comme support pour les scénarios de manipulation et de détection.
- **Support mécanique du robot :** un nouveau support est prévu pour l'installation du robot, du gripper et de la caméra. La conception de ce support s'appuie sur des profilés et des éléments issus du catalogue Norcan, afin d'obtenir une structure stable et modulable.

1.4 Situation de départ et moyens disponibles

Pour mener à bien la transformation de ce bras robotique en une cellule autonome, nous nous sommes appuyés sur un écosystème technique précis. La conception du système a nécessité de faire dialoguer plusieurs composants matériels hétérogènes au sein d'une architecture logicielle unifiée, tout en respectant des contraintes techniques strictes.

1.4.1 Ressources matérielles

L'architecture physique de notre cellule robotisée s'articule autour des éléments suivants :

- **Actionneurs et mécanique :** La base du système est le bras collaboratif **IGUS Rebel 6DOF**, sur lequel est montée une pince industrielle **SCHUNK EGP 25**, permettant la saisie physique des objets.
- **Perception :** L'acquisition des données visuelles et de la profondeur de champ est assurée par une caméra stéréoscopique **Intel RealSense D435**.
- **Calcul et réseau :** L'intelligence du système est distribuée. Un **PC sous Ubuntu** (faisant office de maître ROS 2) se charge des calculs lourds (IA et planification), tandis que plusieurs **Raspberry Pi** agissent comme relais ou nœud distant. L'ensemble de ces équipements communique via un **switch réseau** dédié.

Catégorie	Composant matériel	Fonction principale dans le système
Bras robotique	igus Rebel 6DOF	Exécution des mouvements spatiaux (6 axes)
Préhenseur	SCHUNK EGP 25	Saisie et maintien des objets (contrôle numérique)
Vision	Intel RealSense D435	Capture du flux vidéo et de la carte de profondeur (3D)
Calcul IA & Planification	PC (Ubuntu)	Exécution de ROS 2, MoveIt 2 et des réseaux de neurones
Calcul embarqué	Raspberry Pi	Nœud ROS 2 distant / interface matérielle
Infrastructure	Switch Ethernet	Interconnexion TCP/IP locale des composants

Tableau 2 : Matériels software et hardware

1.4.2 Ressources logicielles

Pour orchestrer ces différents éléments matériels, nous avons déployé une suite logicielle open-source standardisée pour la robotique :

- **ROS 2 Humble :** Le *middleware* (système d'exploitation robotique) qui gère la communication asynchrone entre tous les composants (nœuds, topics, services).
- **MoveIt 2 :** Le framework de planification de mouvement, responsable du calcul de la cinématique inverse (IK) et de la génération des trajectoires sans collision.
- **Python :** Le langage de programmation principal choisi pour développer notre bibliothèque orientée objet, liant la perception à l'action.
- **Gazebo & MATLAB :** Outils utilisés en amont pour la simulation physique du bras (Gazebo).

1.4.3 Contraintes initiales et défis techniques

Le développement de cette architecture n'a pas été sans obstacles. Nous avons dû composer avec trois contraintes majeures qui ont orienté nos choix techniques :

1. **Le protocole propriétaire CRI :** Le bras igus communique nativement via le *Commonplace Robot Interface* (CRI), un protocole propriétaire. L'interfacer avec l'écosystème open-source ROS 2 a nécessité l'adaptation de paquets spécifiques (notamment ceux issus du dépôt de l'AIRLab-POLIMI) pour traduire les commandes ROS en trames compréhensibles par le contrôleur igus.
2. **L'architecture réseau fermée :** Le contrôleur du bras exigeant une communication stricte par TCP/IP, nous avons dû configurer un sous-réseau local statique et dédié (plage d'adresses 192.168.3.10). Cela impose de relier physiquement le PC, le Raspberry Pi et le contrôleur Igus via le switch réseau pour garantir une transmission de données sans latence ni perte de paquets.
3. **Une documentation parcellaire :** La documentation officielle fournie (notamment sur l'intégration du protocole CRI et le paramétrage bas niveau)

étant partielle, une part significative de rétro-ingénierie et de tests empiriques a été nécessaire pour stabiliser la communication réseau et le pilotage des entrées/sorties numériques (DIO) gérant la pince SCHUNK.

2. ÉTAT DE L'ART

2.1 Robotique de manipulation et pick-and-place

2.1.1 Des automates rigides aux bras articulés modernes

La robotique industrielle naît à la fin des années 1950, lorsque George Devol dépose le brevet d'une machine programmable de transfert d'objets, le *Programmed Article Transfer* [1]. En 1961, Joseph Engelberger et Devol industrialisent ce concept à travers l'Unimate #001, premier bras robotisé installé en environnement de production sur la chaîne de moulage sous pression de General Motors à Trenton, dans le New Jersey [2]. Cette machine hydraulique de plusieurs tonnes, dédiée à des tâches répétitives et dangereuses (manutention de pièces brûlantes, soudage par points), marque le point de départ de la robotique de manipulation telle qu'elle est connue aujourd'hui.

La décennie 1970 voit l'émergence des constructeurs européens qui dominent encore le marché. En 1973, le suédois ASEA — qui deviendra plus tard ABB — commercialise l'IRB 6, premier bras industriel entièrement électrique contrôlé par microprocesseur [3]. La même année, l'allemand KUKA introduit le FAMULUS, premier robot doté de six axes entraînés électro-mécaniquement [3], [4]. Cette architecture à six degrés de liberté devient rapidement le standard de la robotique articulée car elle permet d'atteindre n'importe quelle pose dans l'espace de travail (position 3D + orientation 3D). Le PUMA (*Programmable Universal Machine for Assembly*) développé par Victor Scheinman pour Unimation en 1978, puis le SCARA japonais (1981), complètent ce paysage et étendent le domaine d'application à l'assemblage de précision [4].

À partir des années 1980, l'intégration de capteurs de vision et de force enrichit les capacités décisionnelles des bras. Les industriels FANUC, ABB et KUKA proposent alors des cellules robotisées capables de détecter, trier et manipuler des objets de manière partiellement autonome. Ces systèmes restent toutefois confinés dans des cages de sécurité, leur fonctionnement présentant un danger immédiat pour tout opérateur humain à proximité.

2.1.2 L'émergence des robots collaboratifs

Le concept de robot collaboratif — ou *cobot* — est introduit en 1996 par J. Edward Colgate et Michael Peshkin, qui le définissent comme « un dispositif et une méthode permettant l'interaction physique directe entre une personne et un manipulateur contrôlé par ordinateur » [5]. Il faudra cependant attendre 2004 pour qu'apparaisse le premier cobot commercial : le KUKA LBR 3, premier bras léger équipé de capteurs de couple sur chacun de ses axes, capable de détecter une collision et de s'arrêter en quelques millisecondes [6].

L'année 2008 marque un tournant majeur avec la sortie du Universal Robots UR5, premier cobot véritablement abordable et facilement programmable par des opérateurs non spécialistes. Le succès du modèle danois ouvre la voie à une multiplication des acteurs :

ABB lance YuMi en 2015, Franka Emika commercialise le Panda en 2017 (issu de la recherche académique de la TU Munich), Kinova développe la gamme Gen3 orientée recherche et services, et plus récemment, igus introduit la famille ReBeL, premier cobot dont la quasi-totalité des éléments structuraux et mécaniques sont en polymères techniques [7].

Les robots collaboratifs se distinguent de leurs prédécesseurs industriels par trois caractéristiques essentielles. **La sécurité intrinsèque** d'abord : la limitation des vitesses et des couples articulaires, combinée à des surfaces externes arrondies et à des capteurs de couple répartis, permet une cohabitation directe avec un opérateur humain sans cage de protection. Les normes ISO 10218-1/2 et la spécification technique ISO/TS 15066 encadrent précisément cette interaction homme-robot, en définissant quatre modes de collaboration (arrêt surveillé, guidage manuel, surveillance de la distance et limitation de puissance et de force). **La flexibilité de redéploiement** ensuite : un cobot peut être déplacé d'un poste à l'autre en quelques minutes, là où un robot industriel classique demande une réimplantation lourde et un retraçage des zones de sécurité. **La simplicité de programmation** enfin : la plupart des cobots modernes proposent un apprentissage par démonstration (*hand-guiding*) qui permet à l'opérateur de guider physiquement le bras le long de la trajectoire souhaitée, sans écrire la moindre ligne de code.

Ces évolutions répondent à un besoin croissant des PME et des laboratoires de recherche, pour qui les robots industriels traditionnels — coûteux, encombrants et rigides — restaient inaccessibles. Le marché des cobots affiche d'ailleurs un taux de croissance annuel composé supérieur à 30 % depuis 2018, contre environ 10 % pour la robotique industrielle classique [8].

2.1.3 Comparaison des paradigmes : robot industriel vs cobot

Le tableau 2.1 synthétise les différences fondamentales entre robots industriels classiques et robots collaboratifs. Cette distinction conditionne directement le choix d'architecture pour un projet de pick-and-place.

Critère	**Robot industriel classique**	**Robot collaboratif (cobot)**
Charge utile	10 à 300+ kg	0,5 à 15 kg
Vitesse de l'effecteur	Jusqu'à 8 m/s	Limitée à ~1 m/s (ISO/TS 15066)
Précision de répétabilité	±0,02 à ±0,1 mm	±0,03 à ±1 mm
Sécurité	Cage de confinement obligatoire	Cohabitation directe avec opérateurs
Détection de collision	Externe (rideaux laser, scanner)	Intrinsèque (capteurs de couple)
Programmation	Langage propriétaire, programmeur formé	Apprentissage par démonstration, interface graphique
Temps de redéploiement	Plusieurs jours	Quelques minutes à quelques heures
Coût d'acquisition	50 000 € à 200 000 €	5 000 € à 50 000 €
Coût d'intégration	Élevé (cellule complète)	Faible à modéré
Cas d'usage privilégiés	Soudage, peinture, manutention lourde, séries longues	Pick-and-place, assemblage léger, inspection, recherche, éducation

Tableau 2.1 : Comparaison robot industriel classique vs robot collaboratif

2.1.4 Panorama des solutions commerciales de cobots 6DOF

Pour positionner l'igus ReBeL retenu dans ce projet, le tableau 2.2 confronte quatre cobots représentatifs de différents segments du marché : un cobot polymère économique (igus ReBeL), un cobot industriel grand public (Universal Robots UR5e), un cobot orienté recherche avec retour de force (Franka Emika), et un cobot ultra-léger 7 axes pour applications mobiles et de service (Kinova Gen3).

Caractéristique	igus ReBeL 6DOF	Universal Robots UR5e	Franka Emika Production 3	Kinova Gen3 (7 DOF)
Nombre d'axes	6	6	7	7
Charge utile	2 kg	5 kg	3 kg	4 kg
Portée	664 mm	850 mm	855 mm	902 mm
Masse propre	8 kg	20,6 kg	18 kg	8,2 kg
Répétabilité	±1 mm	±0,03 mm	±0,1 mm	±0,1 mm
Capteurs de couple intégrés	Non	Estimation indirecte	Oui (7 axes)	Oui (couple sur chaque axe)
Interface de contrôle	CRI (TCP/IP propriétaire), Modbus, ROS	URScript, RTDE, ROS	FCI, libfranka, ROS 2	Kortex API, ROS 2
Compatibilité ROS 2	Oui (driver communautaire)	Oui (officiel)	Oui (officiel)	Oui (officiel)
Prix indicatif	~5 000 €	~30 000 €	~25 000 €	~30 000 €
Cible	Éducation, PME, prototypage	Industrie, intégrateurs	Recherche, manipulation fine	Recherche, service, mobile

Tableau 2.2 : Comparatif de cobots 6/7 DOF commerciaux

Ce panorama justifie le choix de l'igus ReBeL pour le présent projet : son rapport coût/fonctionnalités en fait une plateforme particulièrement adaptée à un cadre académique, où l'investissement initial doit rester maîtrisé et où le caractère expérimental autorise une répétabilité inférieure aux standards industriels. Sa compatibilité ROS, bien que reposant sur des contributions communautaires non officielles (notamment le dépôt AIRLab-POLIMI ros2-igus-rebel [9]), permet par ailleurs de s'inscrire dans l'écosystème logiciel le plus répandu en robotique de recherche. En contrepartie, la précision millimétrique du ReBeL exige une stratégie de saisie tolérante, ce qui a directement orienté nos choix de conception, notamment l'utilisation d'une pince à mors parallèle large et d'une approche descendante (*top-down grasp*) limitant l'impact des erreurs de localisation latérale.

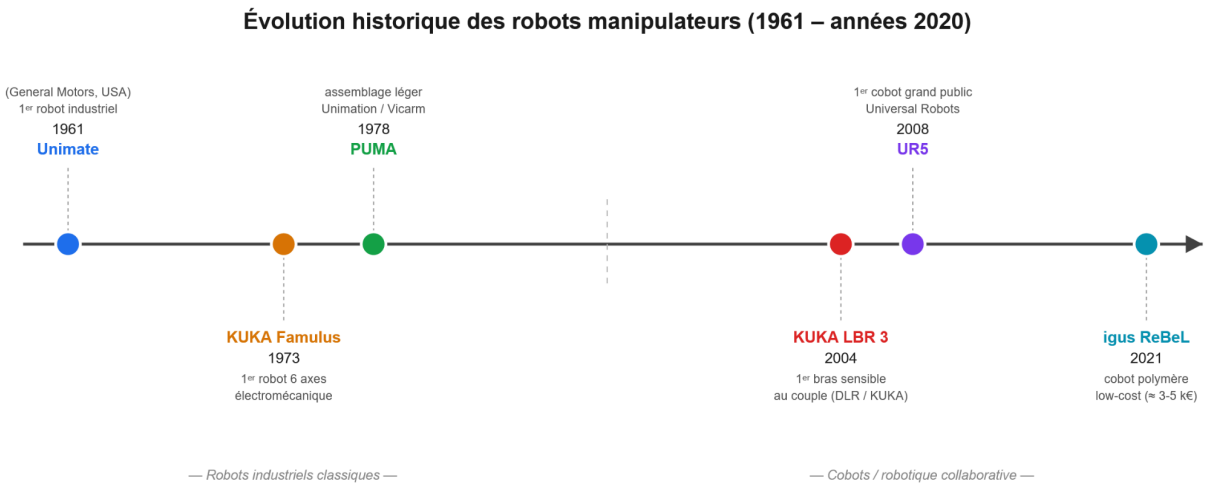


Figure 2.1 : Frise chronologique des robots

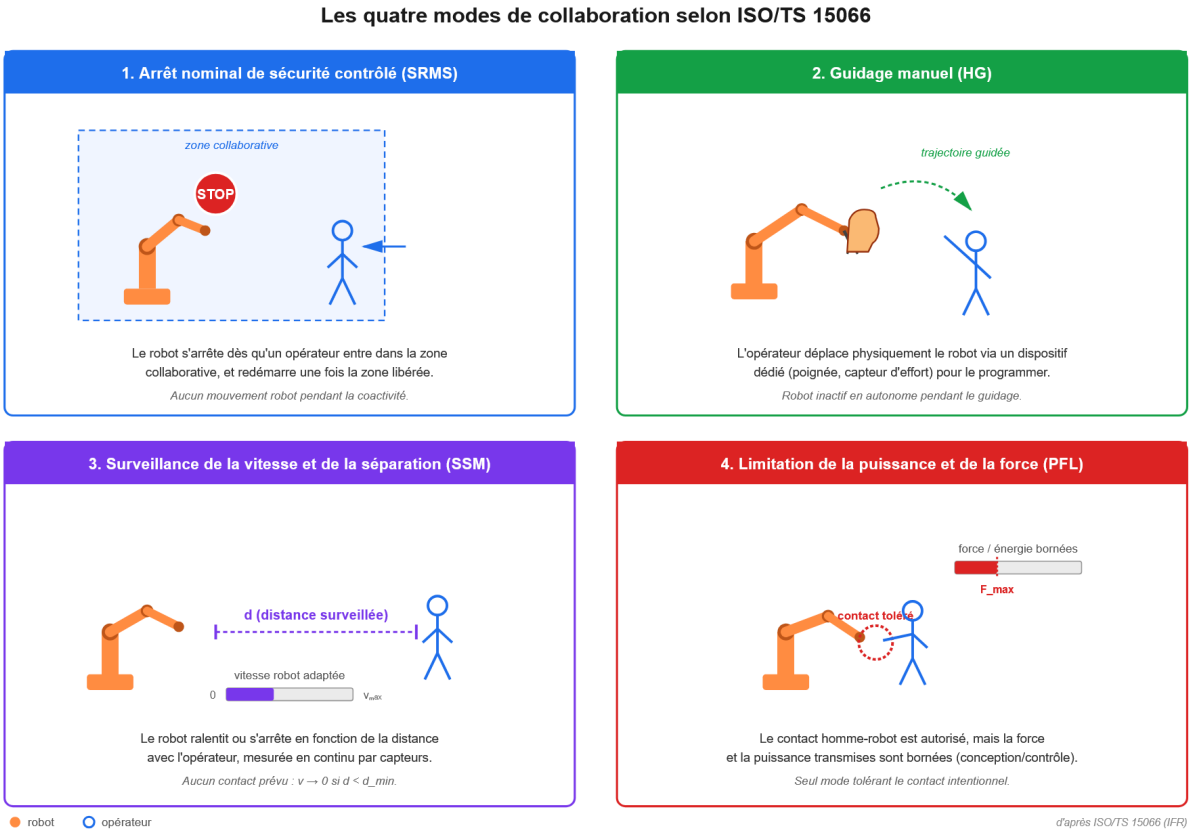


Figure 2.2 : Modes de collaboration

2.2 Détection d'objets par vision artificielle

2.2.1 De la détection à deux étages aux détecteurs en une passe

La détection d'objets — qui consiste à localiser et classifier simultanément les instances présentes dans une image — connaît une rupture méthodologique en 2014 avec l'apparition du R-CNN (*Region-based Convolutional Neural Network*) de Girshick *et al.* [10]. Le principe de ce détecteur, dit « à deux étages », repose sur la génération préalable d'environ 2 000 régions d'intérêt par algorithme de *selective search*, suivie d'une classification individuelle de chacune de ces régions par un réseau convolutif (CNN). Si l'approche atteint pour la première fois des performances acceptables sur le jeu de données PASCAL VOC, son temps d'inférence — plusieurs dizaines de secondes par image — la rend strictement incompatible avec toute application temps réel.

Les évolutions Fast R-CNN [11] (2015) et Faster R-CNN [12] (2015) corrigent progressivement ce défaut en mutualisant l'extraction de caractéristiques et en remplaçant la *selective search* par un *Region Proposal Network* entraînable de bout en bout. Faster R-CNN atteint ainsi 5 à 7 images par seconde sur GPU, mais reste structurellement limité par la nécessité d'une étape de proposition de régions distincte de l'étape de classification.

Ce verrou est levé en 2016 par deux familles concurrentes de détecteurs « à un étage » (*single-stage*), qui suppriment l'étape de proposition de régions et formulent directement la détection comme une tâche de régression sur l'image entière. Le **SSD** (*Single Shot MultiBox Detector*) de Liu *et al.* [13] s'appuie sur des cartes de caractéristiques à plusieurs résolutions pour détecter simultanément des objets de tailles variées, atteignant 59 FPS sur GPU pour une précision comparable à Faster R-CNN. La même année, Redmon *et al.* introduisent **YOLO** (*You Only Look Once*) [14], qui divise l'image en une grille S × S et prédit, pour chaque cellule, les boîtes englobantes et les probabilités de classe en une unique passe avant. La philosophie de YOLO — vitesse maximale même au prix d'une précision légèrement moindre — en fait rapidement le détecteur de référence pour les applications embarquées et robotiques.

2.2.2 Évolution de la famille YOLO

Depuis 2016, la famille YOLO a connu une succession d'itérations menée par différentes équipes, chaque version apportant des optimisations architecturales

spécifiques. YOLOv2 (2017) introduit les boîtes d'ancrage et le *batch normalization*. YOLOv3 (2018) adopte un *backbone* Darknet-53 et la détection multi-échelle. YOLOv4 (Bochkovskiy *et al.*, 2020) intègre le module CSPDarknet et le *bag of freebies* d'optimisations d'entraînement [15]. YOLOv5 (Ultralytics, 2020) marque la transition vers l'écosystème PyTorch et une distribution sous forme de variantes échelonnées (n, s, m, l, x). YOLOv7 (Wang *et al.*, 2022) introduit le réseau E-ELAN, puis YOLOv8 (2023) abandonne définitivement le mécanisme des boîtes d'ancrage au profit d'une approche *anchor-free* avec tête de détection découplée.

YOLOv9 et YOLOv10, publiés au premier semestre 2024, raffinent encore l'efficacité paramétrique, mais c'est **YOLO11**, publié par Ultralytics le 27 septembre 2024 sous la conduite de Glenn Jocher et Jing Qiu, qui établit le nouvel état de l'art en détection temps réel [16], [17]. Le tableau 2.3 récapitule cette évolution.

Version	**Année**	**Contribution principale**	**mAP COCO (val)**
YOLOv1	2016	Détection en une passe sur grille	63,4 % (VOC)
YOLOv3	2018	Backbone Darknet-53, prédictions multi-échelles	33,0 %
YOLOv4	2020	CSPDarknet, *bag of freebies*	43,5 %
YOLOv5	2020	PyTorch, variantes échelonnées n/s/m/l/x	50,7 % (l)
YOLOv7	2022	E-ELAN, optimisations gradient	51,4 % (l)
YOLOv8	2023	Anchor-free, tête découplée, multi-tâches	52,9 % (l)
YOLO11	**2024**	**Backbone affiné, −22 % paramètres vs v8**	**53,4 % (l) / 54,7 % (x)**

Figure 2.2.1 : Versions Yolo

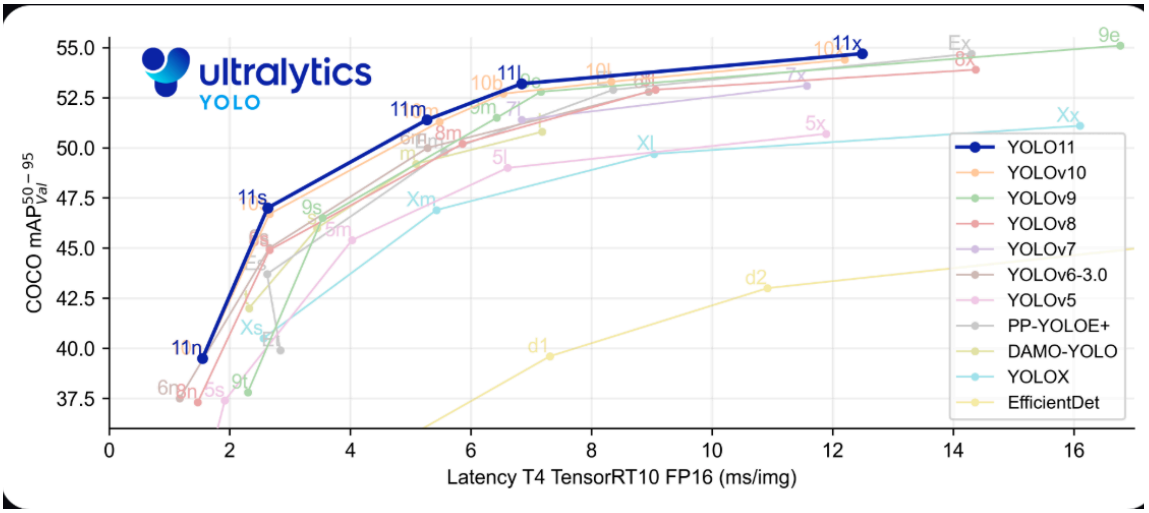


Figure 2.2.2 : Précision moyenne

2.2.3 Spécificités de YOLO11

YOLO11 conserve l'architecture en trois blocs caractéristique de ses prédécesseurs — *backbone* d'extraction de caractéristiques, *neck* d'agrégation multi-échelle, et tête de détection découplée — mais introduit des optimisations structurelles importantes qui justifient son adoption pour le présent projet.

Le premier apport est une **réduction substantielle de la complexité paramétrique**. Le variant YOLO11m atteint sur le jeu COCO une précision moyenne (mAP) supérieure à celle de YOLOv8m tout en utilisant 22 % de paramètres en moins [16]. Cette efficacité accrue se traduit par une empreinte mémoire faible à l'entraînement comme à l'inférence, élément déterminant pour un déploiement sur cible embarquée. YOLO11 est distribué en cinq variantes (n, s, m, l, x) couvrant un spectre de compromis vitesse/précision : la version YOLO11n (nano), de moins de 3 millions de paramètres, est destinée aux microcontrôleurs et systèmes ARM ; la version YOLO11x (extra-large) vise les applications cloud à précision maximale.

Le second apport est une polyvalence multi-tâches native. Là où ses prédécesseurs nécessitaient des architectures dédiées par tâche, YOLO11 unifie cinq tâches de vision dans un même framework : détection de boîtes englobantes, segmentation d'instance, classification d'image, estimation de pose, et détection de boîtes

orientées (oriented bounding boxes). Cette unification simplifie le pipeline logiciel et autorise des extensions futures sans changement de modèle.

Le troisième apport, particulièrement pertinent en robotique, est l'optimisation pour le déploiement embarqué. YOLO11 supporte nativement l'export vers TensorRT, ONNX, OpenVINO, CoreML et TFLite, et atteint des temps d'inférence aussi bas que 1,5 ms par image sur GPU NVIDIA T4 après quantification TensorRT [16]. Sur Raspberry Pi 4, le variant nano maintient une cadence supérieure à 5 FPS sans accélération matérielle, ce qui le rend exploitable même sur les plateformes les plus contraintes. Cette adaptabilité a directement orienté le choix de YOLO11 dans notre projet.

Dans notre cas, le choix s'est porté plus précisément sur la variante YOLO11n, car elle offre le meilleur compromis pour une exécution embarquée. Le système de vision étant déployé sur Raspberry Pi 5 avec Docker, ROS 2 Humble, OpenCV et la caméra Intel RealSense D435, il était nécessaire d'utiliser un modèle léger afin de limiter la charge CPU et de préserver un FPS acceptable. En effet, l'utilisation simultanée de Docker et des flux RGB-D de la caméra RealSense réduit les performances globales du système, notamment la fréquence d'acquisition des images. YOLO11n présente un faible nombre de paramètres et un coût de calcul réduit par rapport aux variantes YOLO11s, YOLO11m, YOLO11l et YOLO11x, ce qui le rend plus adapté à notre architecture embarquée temps réel.

Model	size (pixels)	mAP^{val} 50-95	Speed CPU ONNX (ms)	Speed T4 TensorRT10 (ms)	params (M)	FLOPs (B)
YOLO11n	640	39.5	56.1 ± 0.8	1.5 ± 0.0	2.6	6.5
YOLO11s	640	47.0	90.0 ± 1.2	2.5 ± 0.0	9.4	21.5
YOLO11m	640	51.5	183.2 ± 2.0	4.7 ± 0.1	20.1	68.0
YOLO11l	640	53.4	238.6 ± 1.4	6.2 ± 0.1	25.3	86.9
YOLO11x	640	54.7	462.8 ± 6.7	11.3 ± 0.2	56.9	194.9

Figure 2.1 : Précision moyenne

2.2.4 Localisation 3D par caméra RGB-D

La détection 2D, aussi performante soit-elle, ne suffit pas à un système de pick-and-place : pour saisir un objet, le robot doit connaître sa position dans son repère cartésien tridimensionnel. Cette information de profondeur est apportée par les caméras dites **RGB-D**, qui fournissent simultanément une image couleur (canaux R, G, B) et une carte de profondeur où chaque pixel encode la distance entre la caméra et l'objet observé [18].

Trois familles de technologies permettent d'acquérir cette profondeur. La **lumière structurée** (Microsoft Kinect v1, première génération de RealSense) projette un motif infrarouge connu sur la scène et déduit la profondeur de sa déformation. Le **temps de vol** ou ToF (Kinect v2, Azure Kinect) mesure le temps mis par une impulsion infrarouge pour effectuer un aller-retour vers l'objet. La **stéréovision infrarouge active** (Intel RealSense D435, retenue pour ce projet) combine deux capteurs infrarouges en stéréo avec un projecteur IR qui ajoute de la texture artificielle sur les surfaces homogènes, levant ainsi l'ambiguïté de mise en correspondance qui handicape la stéréovision passive en environnement faiblement texturé.

Quelle que soit la technologie d'acquisition, la transformation d'un pixel détecté en coordonnées 3D repose sur le **modèle géométrique sténopé** (*pinhole camera model*). Soit un point P de coordonnées (X, Y, Z) exprimées dans le repère caméra, sa projection sur le plan image en coordonnées pixel (u, v) s'écrit, en l'absence de distorsion :

$u = f_x \cdot \frac{X}{Z} + c_x \quad ; \quad v = f_y \cdot \frac{Y}{Z} + c_y u = f_x \cdot \frac{X}{Z} + c_x \quad ; \quad v = f_y \cdot \frac{Y}{Z} + c_y$

où (f_x, f_y) sont les distances focales en pixels et (c_x, c_y) les coordonnées du point principal, regroupés dans la matrice des paramètres intrinsèques :

$$K = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix}$$

Le problème inverse — appelé **rétroprojection** — consiste à retrouver les coordonnées 3D d'un pixel (u, v) connaissant sa profondeur Z = d(u, v) fournie par le capteur. Par inversion directe des équations précédentes :

$X = (u - c_x) \cdot \frac{Z}{f_x} \; ; \; Y = (v - c_y) \cdot \frac{Z}{f_y} \; ; \; Z = d(u,v) \; X = (u - c_x) \cdot \frac{fx}{Z} \; ; \; Y = (v - c_y) \cdot \frac{fy}{Z} \; ; \; Z = d(u,v)$

Dans notre architecture, cette opération est effectuée pour chaque centre de boîte englobante produit par YOLO11n, fournissant la pose 3D de la cible dans le repère caméra. Une transformation rigide T_world ← camera supplémentaire, calibrée hors ligne et publiée via l'arbre TF de ROS 2, permet enfin d'exprimer cette pose dans le repère de base du robot — référentiel attendu par MoveIt 2 pour la planification inverse.

Les caméras RGB-D grand public présentent toutefois plusieurs limitations bien documentées dans la littérature [18], [19]. Les surfaces brillantes, transparentes, ou trop éloignées génèrent des trous ou du bruit dans la carte de profondeur ; le bruit suit typiquement un modèle quadratique en fonction de la distance, atteignant ±2 cm à 1 m pour la D435. La synchronisation temporelle entre flux RGB et flux de

profondeur peut introduire des décalages spatiaux si la scène est en mouvement. Enfin, l'éclairage ambiant notamment la lumière solaire directe riche en infrarouge perturbe sensiblement les capteurs à projection IR. Ces limites doivent être prises en compte dans la stratégie de saisie, ce qui justifie en pratique l'usage de marges de sécurité dans la planification (pose de pré-saisie surélevée, approche verticale descendante).

2.3 Middleware robotique : ROS 2 et MoveIt 2 (Theresia)

2.3.1 De ROS à ROS 2 : organisation et communication

ROS, ou *Robot Operating System*, est un ensemble d'outils utilisé pour développer des applications robotiques. Il permet d'organiser un système robotique en plusieurs parties qui communiquent entre elles. ROS est souvent appelé un *middleware*, car il fait le lien entre les capteurs, les programmes de traitement, les commandes du robot et l'utilisateur.

Dans ROS 2, le système est divisé en plusieurs **nœuds**. Chaque nœud est un petit programme qui réalise une tâche précise. Par exemple, un nœud peut gérer la caméra, un autre peut détecter un objet, un autre peut calculer sa position, et un autre peut commander le mouvement du robot.

Les nœuds communiquent entre eux grâce aux **topics**. Un topic est un canal de communication. Un nœud peut publier des données sur un topic avec un **publisher**, et un autre nœud peut recevoir ces données avec un **subscriber**. Cette structure rend le système plus clair, plus modulaire et plus facile à modifier.

ROS 1, développé à partir de 2007, utilisait un élément central appelé roscore. Ce nœud maître était nécessaire pour faire communiquer les différents nœuds. Le problème est que si ce nœud central tombe en panne, toute la communication peut être bloquée [20].

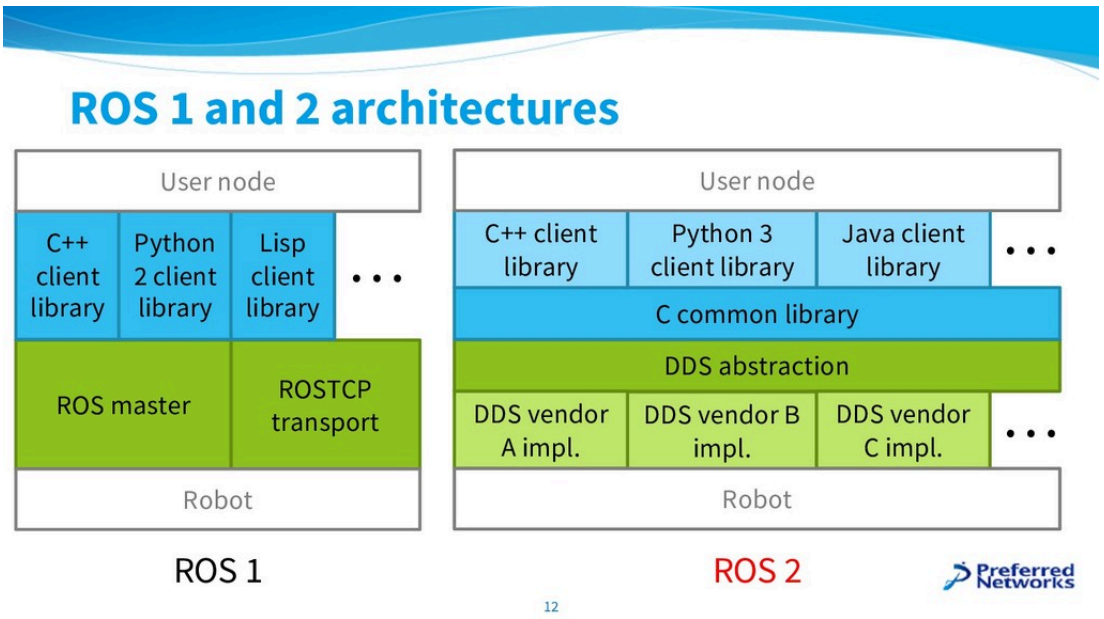


Figure 2.9 : Schéma comparatif des architectures ROS 1 (avec roscore central) et ROS 2

ROS 2 corrige cette limite en utilisant le standard DDS (*Data Distribution Service*). Grâce à DDS, les nœuds peuvent se découvrir automatiquement sur le réseau, sans avoir besoin d'un nœud maître central. Cela rend ROS 2 plus robuste et plus adapté aux systèmes embarqués et industriels [21].

ROS 2 permet aussi d'utiliser des réglages de communication appelés **QoS** (*Quality of Service*). Ces réglages permettent d'adapter l'échange des données selon le besoin. Par exemple, une image caméra doit être transmise rapidement, tandis qu'une information de sécurité doit être transmise de manière fiable.

Dans notre projet, ROS 2 nous permet donc de séparer le système en plusieurs nœuds : perception, calcul de position, transformation de repère, sécurité et déplacement du robot. Cette organisation rend le projet plus facile à tester, à comprendre et à améliorer.

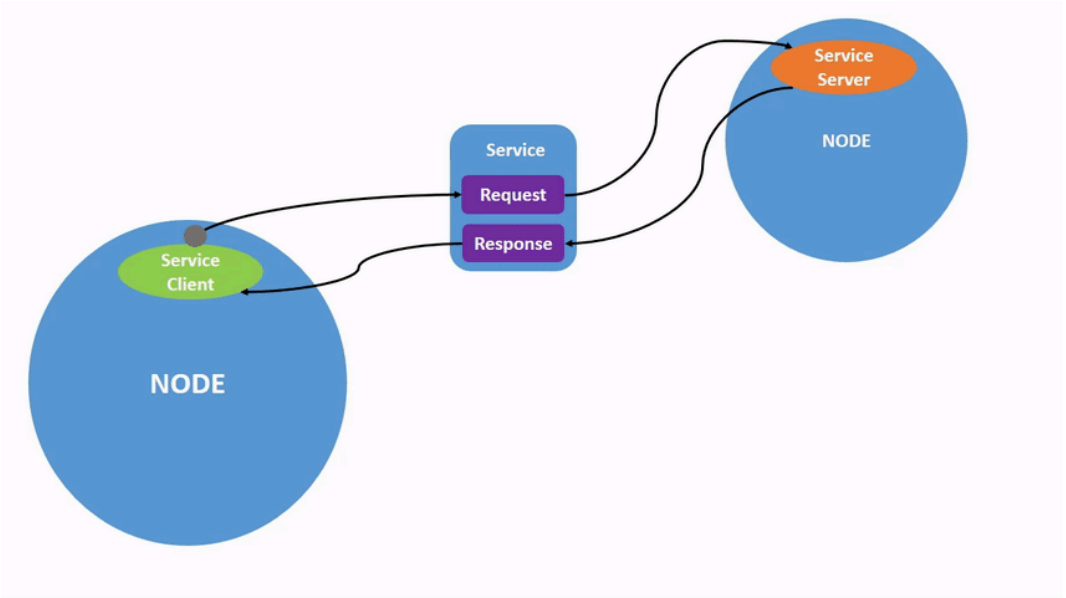


Figure 2.3.1 : Schéma illustratif

2.3.2 Avantages d'utiliser Ubuntu avec ROS 2

L'utilisation d'Ubuntu avec ROS 2 présente plusieurs avantages, notamment en termes de stabilité et de sécurité, deux éléments importants pour le développement d'applications robotiques. Ubuntu fournit un environnement fiable et adapté au développement, ce qui facilite l'installation, l'exécution et la maintenance des outils nécessaires.

- Les distributions de ROS 2 sont généralement construites pour fonctionner avec des versions d'Ubuntu régulièrement mises à jour.
- Un autre avantage concerne les paquets au format Debian, appelés paquets ".deb". Ces paquets facilitent l'installation de ROS 2, car ils permettent une gestion automatique des dépendances et une bonne intégration avec le système:
- Enfin, Ubuntu est une plateforme principale de développement pour ROS 2. Cela signifie que ROS 2 est largement testé sur Ubuntu et bénéficie d'un support important de la part des mainteneurs.

2.3.3 Pourquoi nous avons utilisé ROS 2 Humble

Le choix de ROS 2 Humble repose sur deux critères importants : la stabilité et l'écosystème. Par rapport à ROS 1, ROS 2 Humble est plus avantageux, car il offre un environnement stable et un large support des paquets tiers disponibles sur Ubuntu.

ROS 2 Humble bénéficie aussi d'un support standard LTS, ce qui permet de l'utiliser dans des projets sur une durée plus longue avec plus de confiance. Son écosystème est également très développé : de nombreux pilotes, plugins et outils maintenus par la communauté sont

compatibles avec cette version. C'est le cas par exemple de MoveIt, Nav2, ainsi que de plusieurs nœuds et composants proposés par les fournisseurs de matériel.

Cette version est distribuée avec Ubuntu 22.04 Jammy, qui est très utilisée dans les systèmes industriels et dans plusieurs robots existants. Cela rend ROS 2 Humble plus pratique à utiliser, car il est souvent considéré comme une solution plus directement compatible, ou plus « plug and play », avec les plateformes robotiques actuelles.

Un autre avantage important est la documentation. ROS 2 Humble est très bien documenté et largement utilisé dans les projets disponibles sur GitHub, ce qui facilite la recherche d'exemples, de solutions et de ressources lors du développement.

Le tableau suivant présente une comparaison entre quelques distributions de ROS 2 :

Caractéristique	**Humble Hawksbill**	**Jazzy Jalisco**	**Iron Irwini**
Type de version	LTS (Long Term Support)	LTS (Long Term Support)	Regular Release
Date de sortie	23 mai 2022	23 mai 2024	23 mai 2023
Date de fin de support	Mai 2027	Mai 2029	Décembre 2024
Ubuntu principal	22.04 (Jammy)	24.04 (Noble)	22.04 (Jammy)
Durée de support	5 ans	5 ans	Environ 18 mois

2.3.4 Installation de ROS 2 Humble

L'installation détaillée de ROS 2 Humble sur Ubuntu 22.04 suit les étapes décrites dans la documentation officielle de ROS 2. Plutôt que de recopier toutes les commandes dans ce rapport, nous renvoyons directement le lecteur vers le guide complet :

https://docs.ros.org/en/humble/Installation/Ubuntu-Install-Debs.html

Ce tutoriel explique comment :

- préparer le système, notamment la mise à jour, les dépôts et les clés nécessaires ;
- installer la variante desktop de ROS 2 Humble ;
- configurer l'environnement afin que les commandes ros2 soient disponibles dans le terminal ;
- vérifier que l'installation est correcte.

Une fois ces étapes suivies, l'environnement ROS 2 est opérationnel sur la machine virtuelle. Nous pouvons alors passer à la création d'un premier workspace dédié au bras igus ReBeL.

2.3.5 Politiques de QoS et communication ROS 2

Dans ROS 2, les QoS sont des réglages qui permettent de choisir comment les messages sont envoyés entre les nœuds.
Elles permettent d'adapter la communication selon le type de donnée : image, commande, sécurité, etc.

- Reliability
  - RELIABLE : le message doit arriver correctement.
  - BEST_EFFORT : le message peut être perdu, mais la communication est plus rapide.
  - Exemple : pour une caméra, on préfère parfois perdre une image plutôt que recevoir une image en retard.
- Durability
  - Permet de garder les derniers messages pour les nouveaux nœuds qui arrivent après.
  - Utile si un subscriber démarre après le publisher.
- History et Depth
  - Permettent de choisir combien de messages sont gardés en mémoire.
  - Exemple : KEEP_LAST(5) garde seulement les 5 derniers messages.
- Deadline
  - Permet de vérifier si un message arrive trop tard.
  - Si le message n'arrive pas dans le temps prévu, le système peut détecter un problème.
- Liveliness
  - Permet de savoir si un publisher est toujours actif.
  - Si un nœud ne répond plus, les autres nœuds peuvent le détecter.

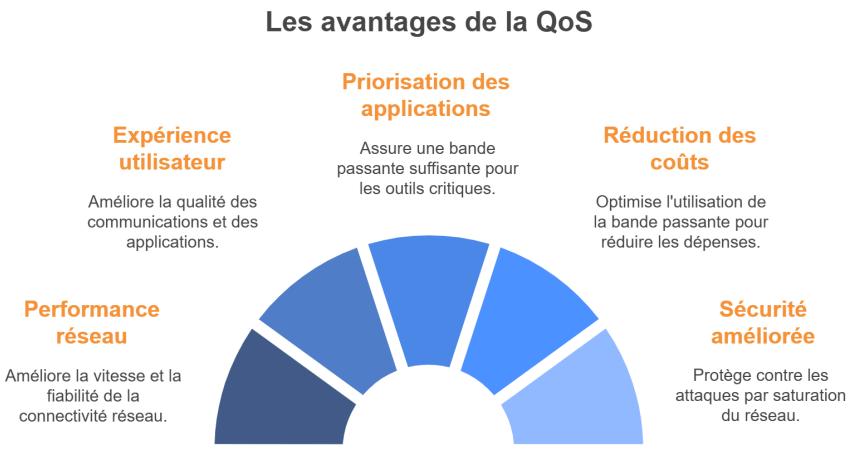


Figure 2.10 : Diagramme illustrant les politiques de QoS principales

Dans notre projet :

- Les messages de sécurité doivent être fiables.
  - On utilise donc un mode RELIABLE.
  - Le but est de ne pas rater une information importante comme un arrêt d'urgence.
- Les images de la caméra RealSense sont envoyées rapidement.
  - On utilise plutôt BEST_EFFORT.
  - Ce n'est pas grave si une image est perdue, car une nouvelle image arrive juste après.

2.3.6 MoveIt 2 et planification de mouvement

MoveIt 2 est un outil utilisé avec ROS 2 pour commander les mouvements d'un robot.

Il permet de :

- charger le modèle du robot à partir des fichiers URDF/SRDF ;
- connaître les articulations et les limites du robot ;
- calculer une trajectoire entre une position de départ et une position d'arrivée ;
- éviter les collisions ;
- visualiser les mouvements dans RViz ;
- envoyer la trajectoire aux contrôleurs du robot.

MoveIt 2 utilise aussi une bibliothèque appelée OMPL pour calculer les trajectoires.

- OMPL teste plusieurs chemins possibles.
- Il cherche un chemin qui permet au robot d'atteindre la cible sans collision.
- Un des algorithmes utilisés est RRT-Connect.
- RRT-Connect est souvent utilisé car il est rapide et efficace pour les bras robotiques.

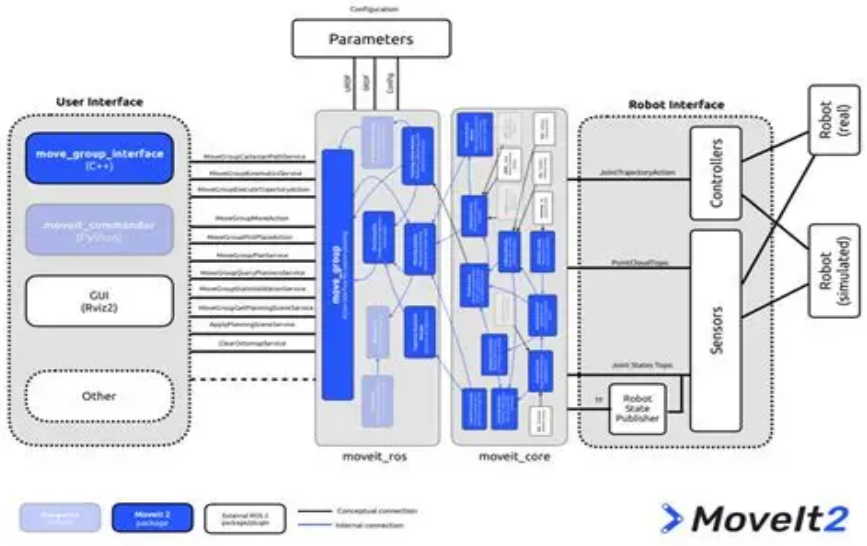


Figure 2.11: Architecture interne de MoveIt 2

Dans notre projet, MoveIt 2 sert donc à préparer le déplacement du robot vers une position cible de manière plus sûre et plus organisée.

2.3.7 Cinématique inverse : KDL et TRAC-IK

Pour déplacer le robot vers une position, il faut transformer une position dans l'espace en angles articulaires.

Cela s'appelle la cinématique inverse.

Exemple :

- on donne une position cible : x, y, z ;
- le solveur calcule les angles : q1, q2, q3, etc. ;
- le robot utilise ces angles pour se déplacer.

MoveIt 2 peut utiliser plusieurs solveurs de cinématique inverse.

- KDL
  - C'est le solveur par défaut.
  - Il fonctionne avec beaucoup de robots.
  - Il est simple à utiliser.
  - Mais il peut échouer si le robot est proche de ses limites articulaires.
- TRAC-IK
  - C'est une alternative plus robuste.
  - Il gère mieux les limites articulaires.
  - Il trouve souvent une solution quand KDL échoue.

Dans notre projet, nous avons gardé **Trac_ik** avec l'option position_only_ik: true.

Cela veut dire que le robot cherche surtout à atteindre la position de la pince, sans imposer une orientation précise.

Ce choix simplifie le problème, car pour notre cas, la position de la pince est plus importante que son orientation exacte.

2.3.8 Communication avec le contrôleur du robot

ROS 2 et MoveIt 2 fonctionnent sur l'ordinateur.
Mais le robot réel possède aussi son propre contrôleur.

Il faut donc une communication entre :

- l'ordinateur ;
- ROS 2 ;
- MoveIt 2 ;
- le contrôleur du robot ;
- les moteurs du robot.

Cette communication permet d'envoyer les ordres de mouvement au robot.

Chaque robot peut utiliser un protocole différent selon le constructeur.
C'est pour cela que l'intégration dépend du robot utilisé.

Dans notre projet, cette partie est importante car elle permet de passer de la simulation dans RViz à une commande réelle du robot.

Protocole	**Constructeur**	**Transport**	**Fréquence de contrôle**	**Modèle d'interaction**	**Documentation**
URScript / RTDE	Universal Robots	TCP/IP, sockets	Jusqu'à 500 Hz (RTDE)	Langage script + échange de registres temps-réel	Publique, complète
KUKA RSI	KUKA	UDP/XML, EtherCAT	250 Hz (4 ms)	Échange XML cyclique, correction de trajectoire en temps réel	Sous licence (XML schema documenté)

CRI	igus	TCP/IP, texte ASCII	~20 Hz (heartbeat 50 ms)	Commandes ligne par ligne, état renvoyé en continu	Partielle, complétée par la communauté

2.3.9 Protocoles de communication avec le contrôleur robot

Pour commander un robot réel, l'ordinateur doit communiquer avec le contrôleur du robot. Chaque constructeur utilise son propre protocole, ce qui influence directement la difficulté d'intégration.

Universal Robots — URScript / RTDE

URScript est le langage de programmation utilisé par les robots Universal Robots. Il ressemble à Python et permet d'envoyer des commandes simples au robot, comme :

- movej pour un mouvement articulaire ;
- movel pour un mouvement linéaire ;
- set_digital_out pour activer une sortie numérique.

Universal Robots utilise aussi RTDE, qui permet d'échanger des données avec le robot en temps réel. Ce protocole est très performant pour lire l'état du robot et envoyer des consignes rapidement.

KUKA — RSI

KUKA utilise le protocole RSI, qui permet de corriger la trajectoire du robot en temps réel. Il est souvent utilisé avec des capteurs externes, comme une caméra ou un capteur de force.

Ce protocole est très puissant pour des applications comme le soudage ou l'usinage, mais il est plus complexe à intégrer et dépend des outils KUKA.

igus ReBeL — CRI

Dans notre projet, le robot igus ReBeL utilise le protocole CRI. C'est un protocole plus simple, basé sur des commandes texte envoyées par TCP/IP.

Une commande peut par exemple avoir cette forme :

CMD Move Joint J1 J2 J3 J4 J5 J6 velocity acceleration

Le contrôleur renvoie aussi l'état du robot, comme les positions des articulations, les entrées/sorties et les erreurs.

Une particularité importante du protocole CRI est la commande ALIVEJOG. Cette commande doit être envoyée régulièrement pour indiquer au contrôleur que la connexion est toujours active. Si elle n'est plus envoyée, le robot arrête le mouvement. Pour cela, nous avons utilisé un thread Python dédié dans notre programme.

Une autre difficulté concerne les sorties numériques. La commande CMD DOUT permet de commander les sorties externes, alors que CMD SetOutput concerne les sorties internes du contrôleur. Cette différence est importante pour piloter le préhenseur SCHUNK EGP 25, qui est connecté à un module d'entrées/sorties externe.

Enfin, contrairement à Universal Robots ou KUKA, igus ne possède pas encore un écosystème ROS 2 aussi complet. Nous avons donc utilisé une base communautaire pour l'intégration du robot avec MoveIt 2, puis nous avons ajouté les parties nécessaires pour notre projet, comme la gestion du préhenseur et l'intégration des capteurs.

2.3.9.a Nœuds ROS 2 prévus pour le projet

À partir de ce premier paquet Python, l'objectif est de définir une architecture de nœuds ROS 2 qui couvre les différents besoins du projet : commande du bras, sécurité, modes de fonctionnement, préhension et perception.

Pour l'instant, un premier nœud est déjà envisagé pour simuler une trajectoire simple du robot. Les autres nœuds sont planifiés et seront développés progressivement.

- **Nœud de simulation de trajectoire :** ce nœud publie des positions et des orientations spécifiques pour chaque articulation du bras. Il permet de tester des trajectoires prédéfinies dans l'environnement virtuel avant de les exécuter sur le robot réel.
- **Nœud de sécurité et d'arrêt d'urgence :** ce nœud surveille l'état du système, comme les erreurs, les limites ou les zones interdites. Si une condition dangereuse est détectée, il doit pouvoir déclencher un arrêt d'urgence logiciel et envoyer une commande d'arrêt au contrôleur du robot.
- **Nœud de contrôle de la vitesse :** ce nœud ajuste la vitesse globale des mouvements du bras. Par exemple, il peut appliquer un facteur de mise à l'échelle sur les trajectoires afin de ralentir ou d'accélérer l'exécution selon le mode de fonctionnement ou l'environnement.
- **Nœuds de modes de fonctionnement manuel et automatique :**
    - **Nœud joystick, mode manuel :** ce nœud permet de téléopérer le bras à l'aide d'un joystick ou d'une manette. Il envoie directement des commandes à certaines articulations ou au point terminal du robot.
    - **Nœud de fonctionnement automatique :** ce nœud exécute des séquences de mouvements prédéfinies ou générées par un planificateur, par exemple des trajectoires de type pick-and-place.

- **Nœud de sélection de mode :** ce nœud reçoit les commandes provenant du mode manuel et du mode automatique, puis choisit le mode actif. Cette sélection peut se faire à l'aide d'un service ou d'un topic de configuration.

- **Nœuds de préhension et de contrôle de l'effecteur terminal :**
  - **Nœud de contrôle du gripper :** ce nœud est dédié à l'ouverture et à la fermeture du préhenseur Schunk. Il peut aussi permettre le contrôle de la force de serrage si cette fonction est disponible.
  - **Nœud de contrôle de l'effecteur terminal :** ce nœud gère l'état de l'outil situé au bout du bras, comme la position d'approche, l'orientation de la pince ou l'état ouvert/fermé. Il fonctionne en coordination avec les trajectoires du bras.
- **Nœuds de perception et de détection d'objets :**
  - **Nœud caméra 3D :** ce nœud récupère en temps réel les données de la caméra, comme les images couleur, la profondeur ou le nuage de points, puis les publie dans ROS 2.
  - **Nœud de détection d'objets 2D :** ce nœud utilise un modèle déjà entraîné, par exemple YOLOv11 compatible avec ROS 2, pour détecter les objets dans les images de la caméra et produire des boîtes englobantes 2D.
  - **Nœud d'estimation de position 3D :** ce nœud combine les détections 2D avec les informations de profondeur ou de nuage de points. Il permet de calculer la position 3D des objets dans le repère du robot et de publier ces positions pour les nœuds de planification.
- **Nœud d'exécution de trajectoires synchronisées :** ce nœud est chargé d'exécuter des trajectoires en synchronisant le mouvement du bras, du gripper et, si nécessaire, d'autres composants comme la caméra ou un convoyeur. Il permet de garantir une séquence cohérente comprenant l'approche, la prise, le dépôt et le retrait.

2.3.9.b Création du workspace ROS 2

```
/Desktop
mkdir igus_rebel_ws
cd igus_rebel_ws
mkdir src
colcon build
```

Une fois ROS 2 Humble installé, nous créons un premier workspace dédié au projet igus ReBeL. Ce workspace sert de dossier de travail pour tous les paquets liés au robot.

Dans ces commandes :

- cd ~/Desktop place le terminal sur le Bureau ;
- mkdir igus_rebel_ws crée un dossier qui sera notre workspace ROS 2 ;
- cd igus_rebel_ws permet d'entrer dans ce dossier ;
- mkdir src crée un sous-dossier src qui contiendra nos futurs paquets ;
- colcon build lance une première compilation du workspace.

À ce stade, le dossier src est encore vide. La compilation se termine donc avec le message Summary: 0 packages finished. Cela est normal, car aucun paquet n'a encore été ajouté au workspace.

2.3.9.c Récupération du premier paquet : description du robot

Pour décrire le bras igus ReBeL dans ROS 2, nous avons choisi d'utiliser des fichiers URDF au format XACRO. Au lieu de recréer cette description depuis zéro, nous avons cherché des ressources existantes. Nous avons trouvé un dépôt GitHub proposant une description complète du robot, ainsi que plusieurs paquets ROS 2 associés, comme la configuration MoveIt 2, la simulation, le contrôleur matériel et les interfaces de commande.

Ce dépôt est compatible avec les distributions ROS 2 Humble et Iron. Cependant, certaines parties sont spécifiques à ROS 2 Iron, ce qui peut créer des erreurs de compilation si l'ensemble des paquets est utilisé directement avec Humble. Pour rester cohérents avec notre choix de ROS 2 Humble, nous avons donc décidé de récupérer uniquement la branche prévue pour Humble.

Concrètement, nous clonons le dépôt dans le dossier src de notre workspace :

```
cd ~/Desktop/igus_rebel_ws/src
git clone -b humble https://github.com/AIRLab-POLIMI/ros2-igus-rebel.git
cd ..
colcon build --symlink-install
```

Dans ces commandes :

- cd ~/Desktop/igus_rebel_ws/src permet d'entrer dans le dossier src du workspace ;
- git clone -b humble permet de récupérer directement la branche adaptée à ROS 2 Humble ;
- cd .. permet de revenir à la racine du workspace ;
- colcon build --symlink-install lance la compilation du workspace en créant des liens symboliques, ce qui facilite les modifications pendant le développement.

À l'issue de cette étape, plusieurs paquets ROS 2 sont disponibles dans notre workspace, notamment :

- **igus_rebel_description :** contient les macros XACRO et les fichiers URDF décrivant le robot et ses différentes configurations, comme le bras seul, le bras monté sur une base ou avec des capteurs ;
- **igus_rebel_moveit_config :** contient la configuration MoveIt 2 utilisée pour la planification et l'exécution des trajectoires du bras ;
- **igus_rebel_gazebo_ignition :** permet de lancer un environnement de simulation Gazebo Ignition pour tester le robot et certains capteurs ;
- **igus_rebel_hw_controller :** contient l'interface ros2_control et le contrôleur matériel permettant de communiquer avec le robot réel via Ethernet ;
- **igus_rebel_commander** et **igus_rebel_gripper_controller :** regroupent des paquets de démonstration et des interfaces de commande pour le bras et le préhenseur.

Dans la suite du projet, nous nous appuyons principalement sur ces paquets pour charger le modèle URDF du robot, configurer MoveIt 2, lancer la simulation, puis établir la communication avec le robot réel.

2.3.9.d Préparation de l'environnement pour la visualisation du robot

Avant de lancer les fichiers launch fournis par le dépôt ros2-igus-rebel, il est nécessaire d'installer plusieurs dépendances. Ces dépendances permettent notamment d'utiliser ros2_control, les contrôleurs ROS 2, les transformations TF, Gazebo et MoveIt 2. Sans ces paquets, certains fichiers de lancement, comme les fichiers de démonstration, peuvent ne pas fonctionner correctement.

Les principaux paquets à installer sous ROS 2 Humble sont les suivants :

sudo apt install ros-$ROS_DISTRO-ros2-control

sudo apt install ros-$ROS_DISTRO-ros2-controllers

sudo apt install ros-$ROS_DISTRO-tf-transformations

sudo apt install ros-$ROS_DISTRO-ros-gz

En complément, il est nécessaire d'installer MoveIt 2 et les outils associés pour la visualisation et la planification de trajectoires :

- **MoveIt 2 :** installation depuis les sources en suivant la documentation officielle ;
- **moveit_visual_tools :** installation depuis les sources à partir du dépôt MoveIt 2, par exemple dans le même workspace que le robot.

Une fois ces dépendances installées, les fichiers launch du dépôt ros2-igus-rebel peuvent être exécutés. Cette étape permet de vérifier que le modèle URDF du robot s'affiche correctement dans RViz ou en simulation avant de passer à des scénarios de manipulation plus avancés.

2.4 Synthèse et positionnement du projet

Les trois sections précédentes ont mis en évidence trois domaines technologiquement matures : la robotique collaborative, la détection d'objets par apprentissage profond, et les middlewares de contrôle robotique. Pourtant, leur intégration conjointe autour d'une plateforme low-cost comme l'igus ReBeL reste très peu documentée dans la littérature. Cette section positionne notre travail dans ce paysage et justifie la combinaison technologique retenue.

2.4.1 Ce qui existe et ce que nous apportons

Le tableau 2.6 récapitule l'état des solutions existantes pour chaque brique de notre système, et le situe par rapport à ce que nous avons développé.

Tableau 2.6 — État de l'art existant et apport du projet

Brique	Solutions existantes	Notre apport
Pilote ROS 2 pour igus ReBeL	Driver communautaire AIRLab-POLIMI (URDF, MoveIt 2 de base) [9]	Extension avec gestion temps-réel du heartbeat et nœud de sécurité
Préhenseur SCHUNK sur igus	Aucun driver public ROS 2 connu	Bibliothèque Python SchunkEGP25 via CRI DOUT + intégration URDF/SRDF
Détection 3D + ROS 2	Stacks complets pour UR, Franka, Kinova (ROS officiels)	Pipeline YOLO11 + RealSense D435 portable, agnostique du bras robot
Bibliothèque CRI Python	Implémentations partielles dispersées sur GitHub	Bibliothèque OOP en trois couches (CRI / gripper / pick-and-place) réutilisable
Pick-and-place ReBeL autonome	Démos constructeur en programmation point-à-point	Chaîne complète vision → planification → exécution sans téléopération

Notre contribution se situe donc moins dans l'innovation algorithmique que dans l'**intégration système** : nous avons construit le chaînon manquant entre un driver ROS 2 communautaire incomplet et une application robotique fonctionnelle, en levant les verrous concrets qui empêchaient cette intégration.

2.4.2 Verrous technologiques adressés

Trois verrous principaux ont structuré notre travail.

Le premier est le **verrou protocolaire** : la documentation publique du protocole CRI ne décrit pas explicitement la distinction entre CMD DOUT (modules d'E/S externes) et CMD SetOutput (sorties internes), ni l'obligation absolue du heartbeat ALIVEJOG. Ces deux points

sont des prérequis bloquants pour piloter le préhenseur et maintenir la connexion. Nous les avons identifiés et documentés par expérimentation directe.

Le deuxième est le **verrou d'intégration matérielle**. Le réseau industriel du robot (192.168.3.0/24) n'étant pas accessible depuis l'interface principale du PC Ubuntu, nous avons dû repenser l'architecture en déléguant l'exécution de la couche CRI à un Raspberry Pi connecté en pont entre les deux réseaux. Ce choix a impliqué une refonte de la répartition logicielle entre PC et nano-ordinateur.

Le troisième est le **verrou de planification**. La portée modeste du ReBeL (664 mm) et ses amplitudes articulaires restreintes rendent fréquents les échecs de convergence en cinématique inverse 6D complète. Nous avons résolu ce point en passant le solveur KDL en mode position_only_ik, compatible avec notre stratégie de saisie verticale, et en imposant une borne inférieure sur l'altitude de l'effecteur (z_min = -0,15 m) pour éviter les collisions avec la table.

2.4.3 Pertinence et originalité de la combinaison igus + ROS 2 + YOLO 11

Cette triple combinaison est pertinente pour trois raisons.

D'un point de vue économique, le ReBeL est environ six fois moins cher qu'un UR5e (Tableau 2.2). Cette accessibilité ouvre la robotique collaborative à des structures qui ne peuvent justifier l'investissement d'un cobot industriel — PME, laboratoires d'enseignement, environnements de recherche exploratoire. ROS 2 et YOLO 11, tous deux open-source, complètent cette logique d'accessibilité.

D'un point de vue technique, ROS 2 Humble apporte le déterminisme et les politiques de QoS nécessaires à la sécurité d'une application impliquant un effecteur en mouvement, là où ROS 1 restait limité à un usage purement académique. YOLO 11, avec son anchor-free design et son optimisation pour le déploiement embarqué, s'inscrit naturellement dans une architecture distribuée où le PC hôte doit gérer simultanément la perception, la planification et la communication temps-réel.

D'un point de vue méthodologique enfin, cette combinaison reste très peu couverte dans la littérature : les démonstrations existantes du ReBeL se limitent généralement à des trajectoires pré-programmées en iRC sans intégration ROS 2 complète, tandis que les démonstrations ROS 2 + YOLO se concentrent quasi exclusivement sur les bras déjà bien dotés en drivers officiels (UR, Franka, Kinova). Notre projet contribue donc à documenter et à rendre reproductible une stack qui devrait bénéficier à la communauté igus dans son ensemble — en cohérence avec les perspectives d'open-sourcing évoquées en section 8.

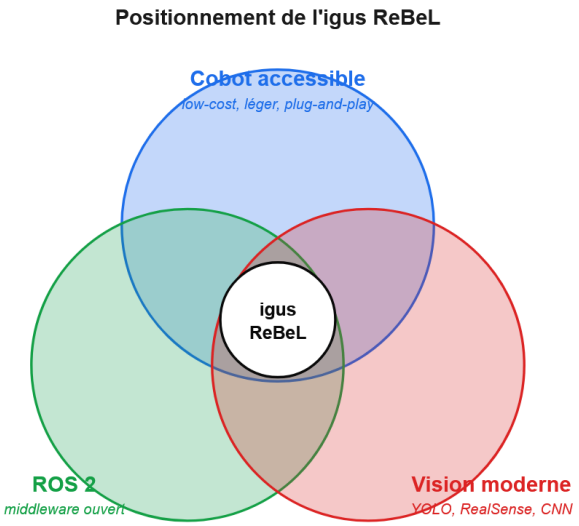


Figure 2.14 — Schéma de positionnement

3. ARCHITECTURE GLOBALE DU SYSTÈME

(Theresia+Daniel)

3.1 Vue d'ensemble fonctionnelle

3.1 Vue d'ensemble fonctionnelle

Le système développé suit une chaîne complète allant de la perception jusqu'à l'action du robot. L'objectif est que le robot puisse détecter un objet, calculer sa position, planifier un mouvement, puis exécuter une tâche de prise et de dépôt.

Le flux de données général est le suivant :

Caméra RealSense → YOLO → Calcul des coordonnées → MoveIt → CRI → Robot

La caméra RealSense fournit les images RGB, les images de profondeur et les informations de calibration. Ces données sont utilisées par le nœud de détection YOLO pour reconnaître l'objet dans l'image. YOLO publie ensuite la position de l'objet en pixels sous forme de détection 2D.

Après cela, un nœud de perception utilise la détection YOLO avec l'image de profondeur pour calculer la position 3D de l'objet par rapport à la caméra. Cette position est ensuite transformée dans le repère du robot, généralement base_link, afin que le bras puisse l'utiliser comme position cible.

Une fois la position de l'objet connue dans le repère du robot, le nœud de pick and place utilise MoveIt pour planifier le mouvement du bras. MoveIt calcule une trajectoire correcte entre la position actuelle du robot et la position cible. Ensuite, les commandes sont envoyées vers le robot à travers l'interface CRI, qui permet la communication avec le contrôleur du robot.

Les principaux modules du système sont :

- **Module caméra**
  Il fournit les données nécessaires à la perception : image couleur, image de profondeur et calibration.
- **Module YOLO**
  Il détecte l'objet dans l'image et publie les résultats de détection.
- **Module de calcul 3D**
  Il utilise la profondeur et la calibration de la caméra pour convertir la position de l'objet en coordonnées 3D.
- **Module de transformation TF**
  Il transforme la position de l'objet du repère caméra vers le repère du robot.
- **Module MoveIt**
  Il planifie les trajectoires du bras robotique à partir de la position cible.
- **Module CRI**
  Il assure la communication entre le programme et le robot réel.
- **Module pince**
  Il commande l'ouverture et la fermeture du préhenseur pendant la tâche de pick and place.
- **Modules de sécurité**
  Ils surveillent les limites articulaires, la vitesse et la distance avec la table pour éviter les mouvements dangereux.

Architecture système complèt

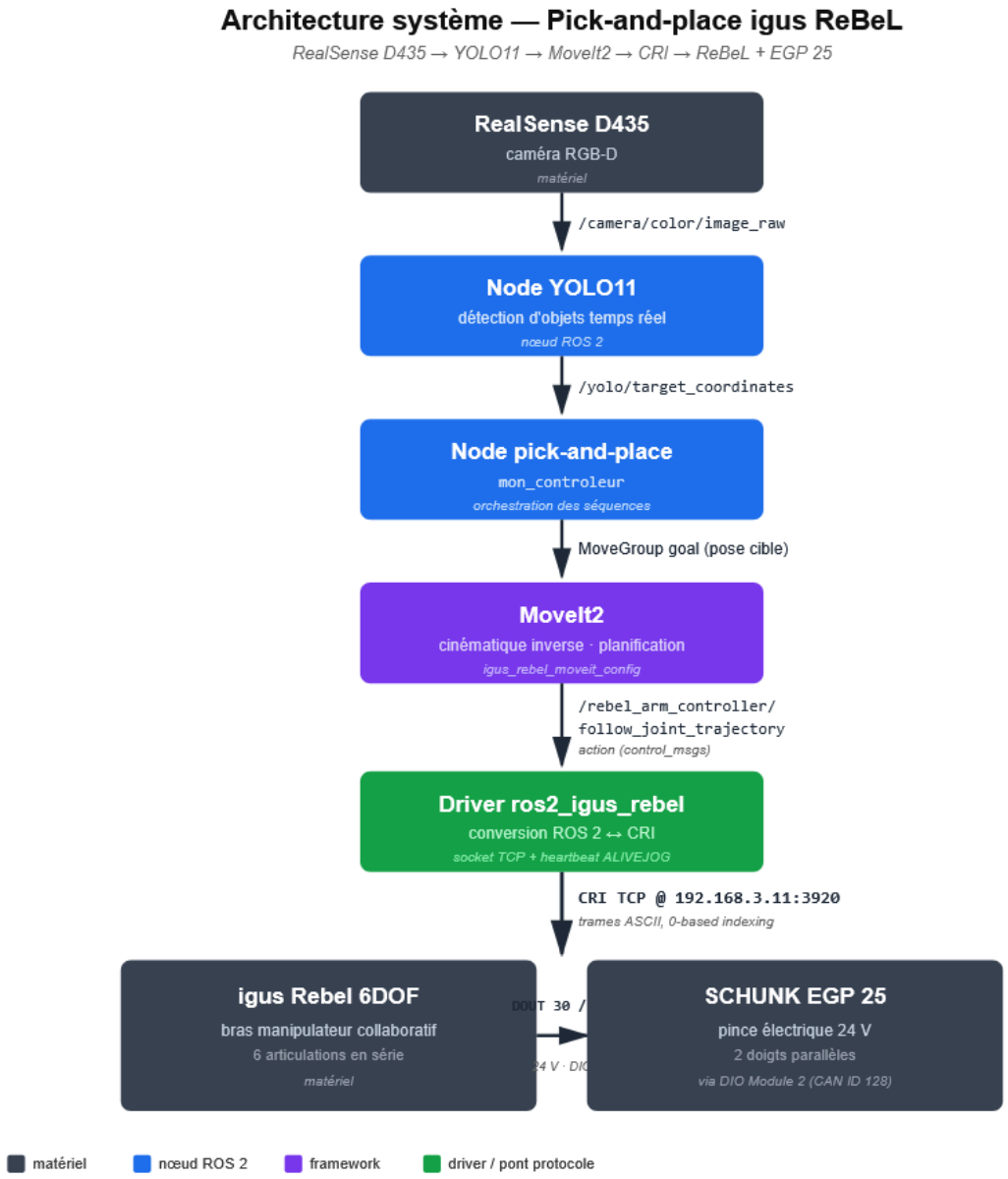


Figure 3.1 : Schéma de l'architecture globale

3.2 Architecture ROS2 détaillée

Nœud ROS2	Package	Rôle	Topics publiés	Topics souscrits
camera_node	realsense2_camera	Flux RGB-D	/camera/color/image_raw, /camera/depth/...	—

yolo_node	mon_controleur	Détection YOLO11n	/yolo/target_coordinates	/camera/color/image_raw
pick_place_node	mon_controleur	Orchestration	/safety/emergency_stop	/yolo/target_coordinates
move_group	igus_rebel_moveit_config	Planification MoveIt2	—	actions follow_joint_trajectory
safety_node	mon_controleur	Watchdog 10Hz	/safety/emergency_stop	—

3.3 Nœuds ROS 2 prévus pour le projet

À partir de ce premier paquet Python, l'objectif est de définir une architecture de nœuds ROS 2 qui couvre les différents besoins du projet : commande du bras, sécurité, modes de fonctionnement, préhension et perception.

Pour l'instant, un premier nœud est déjà envisagé pour simuler une trajectoire simple du robot. Les autres nœuds sont planifiés et seront développés progressivement.

- **Nœud de simulation de trajectoire :** ce nœud publie des positions et des orientations spécifiques pour chaque articulation du bras. Il permet de tester des trajectoires prédéfinies dans l'environnement virtuel avant de les exécuter sur le robot réel.
- **Nœud de sécurité et d'arrêt d'urgence :** ce nœud surveille l'état du système, comme les erreurs, les limites ou les zones interdites. Si une condition dangereuse est détectée, il doit pouvoir déclencher un arrêt d'urgence logiciel et envoyer une commande d'arrêt au contrôleur du robot.
- **Nœud de contrôle de la vitesse :** ce nœud ajuste la vitesse globale des mouvements du bras. Par exemple, il peut appliquer un facteur de mise à l'échelle sur les trajectoires afin de ralentir ou d'accélérer l'exécution selon le mode de fonctionnement ou l'environnement.
- **Nœuds de modes de fonctionnement manuel et automatique :**
  - **Nœud joystick, mode manuel :** ce nœud permet de téléopérer le bras à l'aide d'un joystick ou d'une manette. Il envoie directement des commandes à certaines articulations ou au point terminal du robot.
  - **Nœud de fonctionnement automatique :** ce nœud exécute des séquences de mouvements prédéfinies ou générées par un planificateur, par exemple des trajectoires de type pick-and-place.
- **Nœuds de préhension et de contrôle de l'effecteur terminal :**
  - **Nœud de contrôle du gripper :** ce nœud est dédié à l'ouverture et à la fermeture du préhenseur Schunk. Il peut aussi permettre le contrôle de la force de serrage si cette fonction est disponible.

    - **Nœud de contrôle de l'effecteur terminal :** ce nœud gère l'état de l'outil situé au bout du bras, comme la position d'approche, l'orientation de la pince ou l'état ouvert/fermé. Il fonctionne en coordination avec les trajectoires du bras.
- **Nœuds de perception et de détection d'objets :**
    - **Nœud caméra 3D :** ce nœud récupère en temps réel les données de la caméra, comme les images couleur, la profondeur ou le nuage de points, puis les publie dans ROS 2.
    - **Nœud de détection d'objets 2D :** ce nœud utilise un modèle déjà entraîné, par exemple YOLOv11 compatible avec ROS 2, pour détecter les objets dans les images de la caméra et produire des boîtes englobantes 2D.
    - **Nœud d'estimation de position 3D :** ce nœud combine les détections 2D avec les informations de profondeur ou de nuage de points. Il permet de calculer la position 3D des objets dans le repère du robot et de publier ces positions pour les nœuds de planification.
- **Nœud d'exécution de trajectoires synchronisées :** ce nœud est chargé d'exécuter des trajectoires en synchronisant le mouvement du bras, du gripper et, si nécessaire, d'autres composants comme la caméra ou un convoyeur. Il permet de garantir une séquence cohérente comprenant l'approche, la prise, le dépôt et le retrait.

3.4 Architecture réseau et communication bas niveau

L'ensemble des composants matériels est interconnecté autour d'un switch Ethernet dédié, formant un unique réseau local configuré sur la plage d'adresses du robot (192.168.3.0).

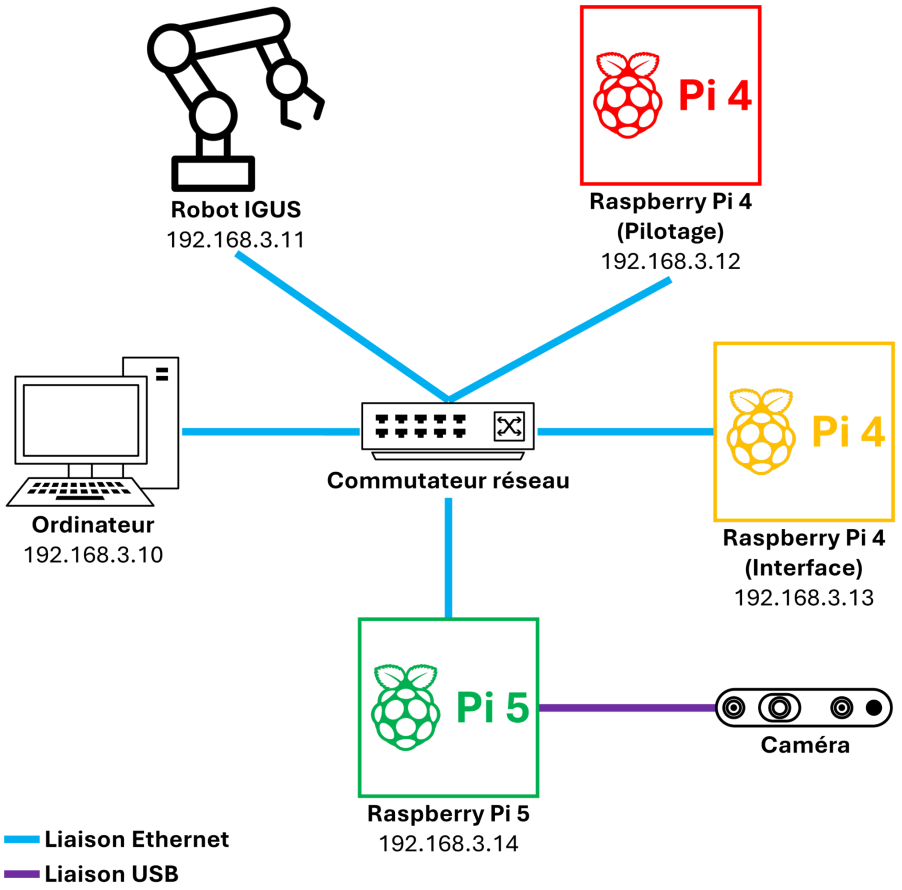


Figure 3.4 : Schéma de l'architecture réseau et de l'interconnexion des équipements

Pour faciliter la compréhension de cette topologie, les rôles et les fonctions spécifiques de chaque équipement au sein de ce réseau sont détaillés ci-dessous :

- **Ordinateur :** Il centralise la logique globale du projet et prend en charge les calculs lourds comme la planification de trajectoire.
- **Raspberry Pi 4 (Pilotage) :** Cette unité gère la communication matérielle et assure la liaison directe avec le robot.
- **Raspberry Pi 4 (Interface) :** Il gère uniquement l'affichage graphique de l'IHM permettant à l'opérateur de piloter manuellement le bras depuis l'écran tactile.
- **Raspberry Pi 5 :** Connecté en USB à la caméra, il s'occupe exclusivement de la détection et de la localisation des objets.
- **Robot IGUS ReBeL :** Le bras articulé à 6 axes et son contrôleur reçoivent et exécutent instantanément les consignes de mouvement transmises depuis le réseau.

L'utilisation d'adresses IP statiques sur l'ensemble des équipements permet d'éviter tout conflit au démarrage et de garantir la stabilité des communications.

Equipement	Liaison	Adresse IP	Masque de sous-réseau
Ordinateur	Ethernet	192.168.3.10	255.255.255.0
Robot IGUS	Ethernet	192.168.3.11	255.255.255.0
Raspberry Pi 4 (Pilotage)	Ethernet	192.168.3.12	255.255.255.0
Raspberry Pi 4 (Interface)	Ethernet	192.168.3.13	255.255.255.0
Raspberry Pi 5	Ethernet (+ USB)	192.168.3.14	255.255.255.0

Tableau 3.3 : Plan d'adressage IP

3.4.2 Isolation du réseau ROS 2

Par défaut, toutes les machines connectées à un même réseau local partagent le même identifiant de domaine ROS (le domaine 0). Afin d'isoler strictement nos communications et d'éviter toute interférence avec les autres équipements ou projets du laboratoire, la variable d'environnement suivante a été configurée sur l'ensemble de nos cartes Raspberry Pi et de nos ordinateurs : export ROS_DOMAIN_ID=10

3.4.3 Choix du middleware DDS

Une fois l'infrastructure physique et le plan d'adressage IP définis, il a fallu configurer le middleware DDS (le logiciel intermédiaire de ROS 2). Son rôle est d'agir comme une passerelle transparente pour distribuer les flux de données (coordonnées, commandes de la pince, détection de la caméra) entre les différentes machines.

Pour sélectionner la solution la plus adaptée à notre topologie (liaison Ethernet filaire et envoi rapide de coordonnées), un comparatif a été réalisé entre les trois outils principaux compatibles avec ROS 2 :

Middleware	Avantages	Inconvénients
FastDDS	Intégré par défaut	Ne transmet pas les commandes au robot lors des tests
Zenoh	Adapté pour les connexions sans fil	Nécessite plus de mémoire RAM et d'intégration
CycloneDDS	Adapté pour les connexions Ethernet	Perd en efficacité sur un réseau Wi-Fi instable

Tableau 3.4 : Comparaison des middleware DDS

Validation du choix : Le choix final s'est porté sur CycloneDDS. Ce choix se justifie par sa parfaite adéquation avec notre topologie réseau Ethernet et par les résultats de nos tests en conditions réelles.

4. IMPLÉMENTATION ET RÉALISATION

4.1 Perception : Détection et localisation 3D des objets

(Hashem AL-GAFRI + Theresia)

4.1.1 Nœud de perception : détection d'objet avec YOLO

Après la mise en place de la partie sécurité, nous avons travaillé sur la perception du robot. L'objectif de cette partie est de permettre au robot de voir son environnement et de détecter les objets présents devant la caméra.

Cette partie constitue la première étape de la chaîne de pick-and-place. Elle permet au robot de détecter automatiquement une roulette, de calculer sa position dans l'image, d'estimer sa distance grâce à la caméra Intel RealSense D435 et de préparer la transformation de ses coordonnées vers le repère du bras robotique.

Pour cela, nous avons utilisé une caméra Intel RealSense D435 et un modèle YOLO entraîné pour reconnaître les objets. Le nœud ROS 2 créé s'appelle object_detection. Il récupère l'image RGB publiée par la caméra sur le topic :

/camera/camera/color/image_raw

Ensuite, l'image ROS est convertie en image OpenCV grâce à cv_bridge. Cette conversion est nécessaire, car le modèle YOLO travaille avec des images au format OpenCV.

Une fois l'image convertie, le modèle YOLO analyse l'image et détecte les objets présents. Pour chaque objet détecté, le programme récupère plusieurs informations : le nom de l'objet, le score de confiance, la boîte de détection et le centre de l'objet dans l'image.

Le modèle utilisé est YOLO11n, entraîné sur un dataset personnalisé de roulettes. Ce choix a été motivé par les contraintes de calcul du Raspberry Pi 5 et par l'exécution simultanée de Docker, ROS 2 et des flux RGB-D de la caméra.

Le développement a été réalisé sur Raspberry Pi 5 dans un conteneur Docker basé sur ROS 2 Humble. Cette solution a été choisie car ROS 2 Humble est principalement compatible avec Ubuntu 22.04, alors que le Raspberry Pi 5 fonctionnait sous Ubuntu 24.04. Docker a donc

permis de conserver un environnement logiciel stable pour l'exécution de ROS 2, OpenCV, YOLO11n et des pilotes de la caméra Intel RealSense D435.

Cependant, l'utilisation simultanée de Docker, des flux RGB-D de la caméra RealSense et de l'inférence YOLO entraîne une diminution du FPS sur le Raspberry Pi 5. Pour cette raison, le choix d'un modèle léger comme YOLO11n était nécessaire afin de maintenir des performances temps réel acceptables.

Le centre de l'objet est calculé à partir des coordonnées de la boîte détectée :

cx = (x1 + x2) / 2

cy = (y1 + y2) / 2

Ces deux valeurs donnent la position de l'objet en pixels dans l'image. Cette information est importante, car elle sera ensuite utilisée par un autre nœud pour estimer la position réelle de l'objet par rapport à la caméra puis par rapport au robot.

Le nœud publie aussi une image annotée sur le topic :

/image_yolo/image

Cette image permet de visualiser directement dans ROS ou RViz les objets détectés avec leurs cadres de détection.

En plus de l'image annotée, le nœud publie les informations des objets détectés sous deux formes. La première est un message texte au format JSON sur le topic :

/image_yolo/objects

La deuxième est un message de type Detection2DArray sur le topic :

/image_yolo/detections

Ce dernier topic est surtout utilisé pour transmettre les détections au nœud suivant, qui va s'occuper du calcul des coordonnées de l'objet.

Ce nœud est donc la première étape de la perception. Il ne déplace pas encore le robot, mais il permet de détecter l'objet dans l'image et de fournir les informations nécessaires pour la suite du système.

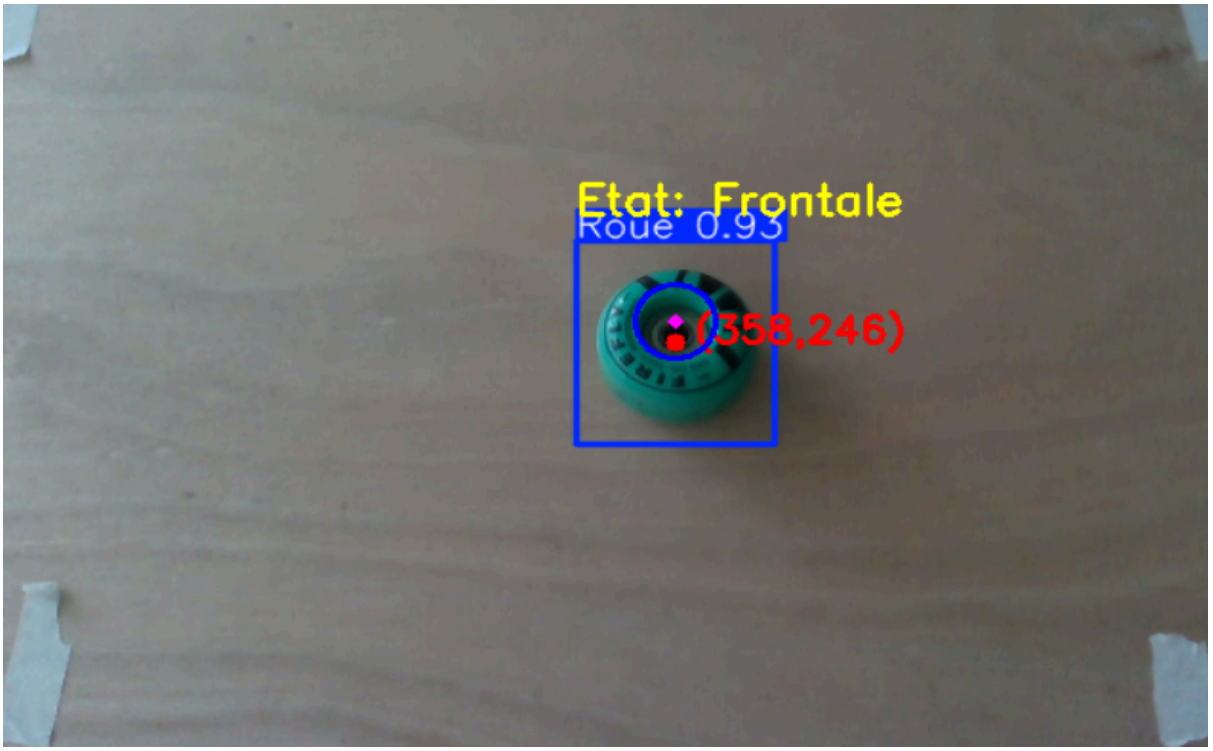


Figure 4.1.1 : Détection temps réel d'une roulette avec YOLO11n.

En plus de la détection classique, une estimation simple de l'orientation de la roulette a été ajoutée. Deux états principaux ont été considérés :

- roulette frontale
- roulette verticale

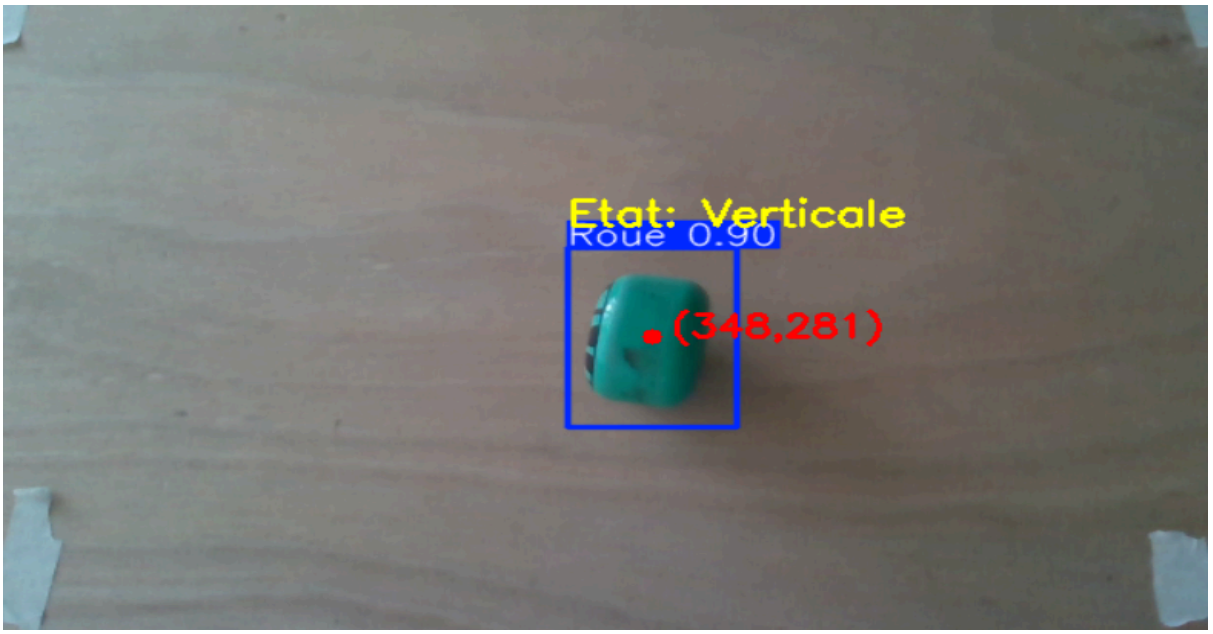


Figure 4.1.2 : Détection d'une roulette en position verticale.

4.1.2 Calcul de la position 3D de l'objet par rapport à la caméra

Après la détection de l'objet avec YOLO, nous avons ajouté un deuxième nœud de perception. Ce nœud s'appelle object_position_camera. Son rôle est de passer d'une détection 2D dans l'image à une position 3D de l'objet par rapport à la caméra.

Le nœud reçoit d'abord les détections publiées par YOLO sur le topic :

/image_yolo/detections

Chaque détection contient la position de l'objet dans l'image sous forme de boîte de détection. Le programme récupère principalement le centre de cette boîte, noté ici (u, v). Ce centre correspond à la position de l'objet en pixels dans l'image.

Le nœud utilise également l'image de profondeur publiée par la caméra RealSense sur le topic :

/camera/camera/aligned_depth_to_color/image_raw

Grâce à cette image, le programme récupère la distance entre la caméra et l'objet. Pour éviter les erreurs dues au bruit de profondeur, il ne se limite pas à un seul pixel. Il considère une petite zone autour du centre de l'objet, puis utilise la valeur médiane de profondeur.

Cette méthode permet de réduire les erreurs liées au bruit de profondeur de la caméra RealSense, notamment lorsque l'objet se trouve au bord de la boîte détectée ou lorsque certaines valeurs de profondeur sont invalides.

Le nœud reçoit aussi les informations de calibration de la caméra sur le topic :

/camera/camera/color/camera_info

Ces informations fournissent les paramètres intrinsèques de la caméra, notamment fx, fy, cx et cy. Elles sont nécessaires pour transformer une position exprimée en pixels en une position réelle dans l'espace.

Une fois le centre de l'objet et sa profondeur connus, le programme calcule la position 3D de l'objet dans le repère de la caméra. On obtient alors les coordonnées X, Y et Z, où Z représente la profondeur, c'est-à-dire la distance entre la caméra et l'objet.

Enfin, le nœud publie cette position sous forme de message PointStamped sur le topic :

/object_position_in_camera

Ce topic est ensuite utilisé par le nœud suivant pour transformer la position de l'objet du repère caméra vers le repère du robot.

En résumé, ce nœud fait le lien entre la détection YOLO et la position réelle de l'objet. YOLO donne la position de l'objet dans l'image, puis la caméra de profondeur permet de calculer sa position 3D par rapport à la caméra.

4.1.3 Transformation de la position de l'objet vers le repère du robot

Après la détection de l'objet et le calcul de sa position par rapport à la caméra, nous avons ajouté un autre nœud de perception. Son rôle est de transformer cette position pour l'exprimer dans le repère du robot.

Le nœud s'appelle object_position_transform. Il reçoit la position de l'objet publiée sur le topic :

/object_position_in_camera

Cette position est d'abord donnée dans le repère de la caméra. Cependant, pour que le robot puisse utiliser cette information, il faut connaître la position de l'objet par rapport au robot, donc dans le repère base_link.

Pour faire cette transformation, le nœud utilise le système TF de ROS 2. TF permet de connaître la relation entre deux repères, par exemple entre camera_link et base_link.

Dans notre cas, la caméra est placée à une certaine distance du robot. La distance utilisée est :

y = 0.56 m

x = 0.0 m

z = 0.40 m

Cela représente la translation entre la caméra et la base du robot.

En plus de cette distance, il faut aussi prendre en compte l'orientation de la caméra par rapport au robot. En effet, le repère de la caméra et le repère du robot ne sont pas forcément orientés dans le même sens. On peut donc définir un angle (\alpha), qui représente l'angle de rotation d'un repère par rapport à l'autre.

Dans une seconde approche expérimentale, une transformation par homographie a également été étudiée afin de convertir directement les coordonnées image en coordonnées réelles sur le plan de travail. Cette méthode est particulièrement adaptée lorsque les objets se trouvent sur une surface plane fixe observée par une caméra inclinée.

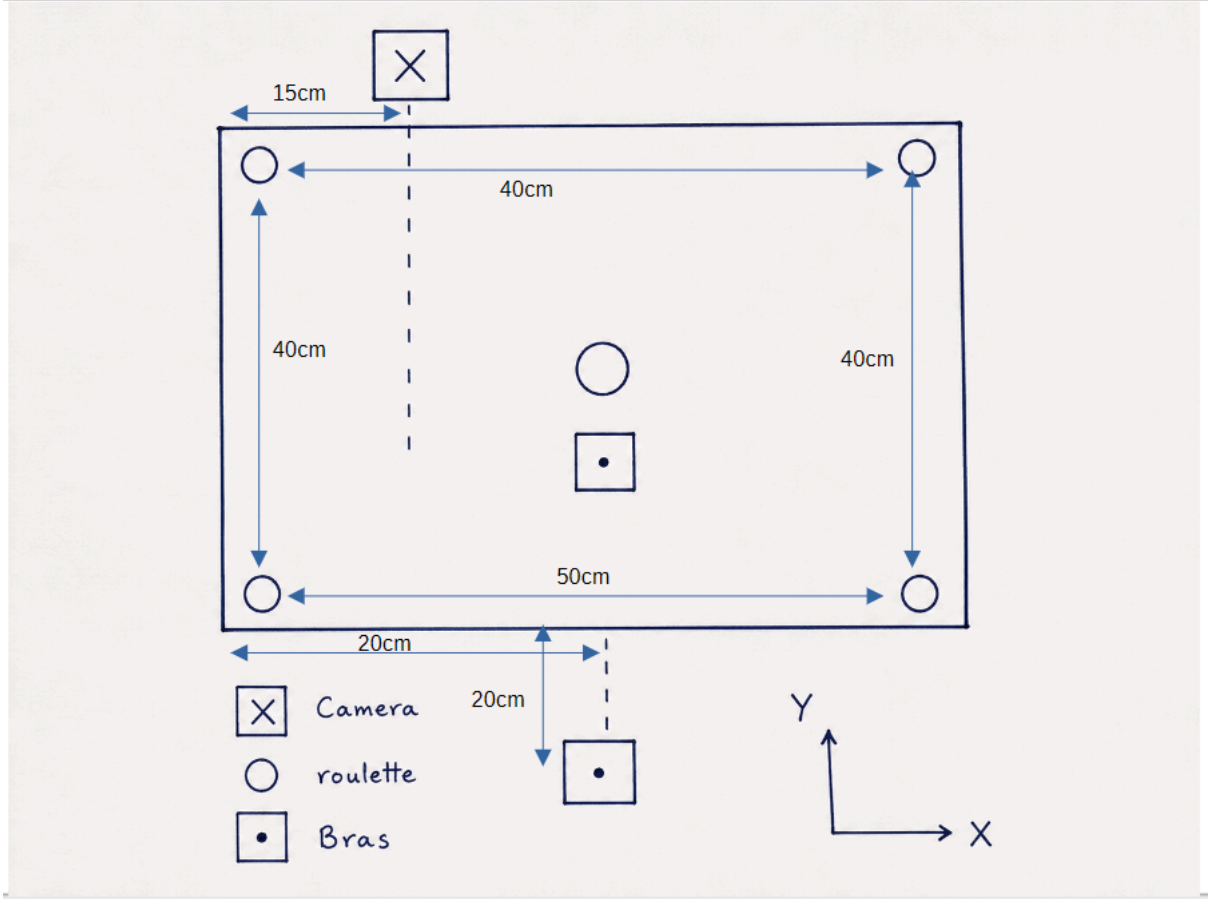


Figure 4.1.3 : Points de calibration utilisés pour la transformation par homographie entre l'image caméra et le plan de travail.

Cet angle permet de corriger la position de l'objet lorsque l'axe de la caméra n'est pas parfaitement aligné avec l'axe du robot. La transformation entre les deux repères ne dépend donc pas seulement de la distance entre la caméra et le robot, mais aussi de leur orientation relative.

Lorsque le nœud reçoit un point dans le repère caméra, il cherche la transformation disponible entre le repère de la caméra et le repère du robot. Cette transformation contient donc la translation et la rotation entre les deux repères. Ensuite, le point est converti dans le repère base_link.

Le point transformé est publié sur le topic :

/object_position_in_world

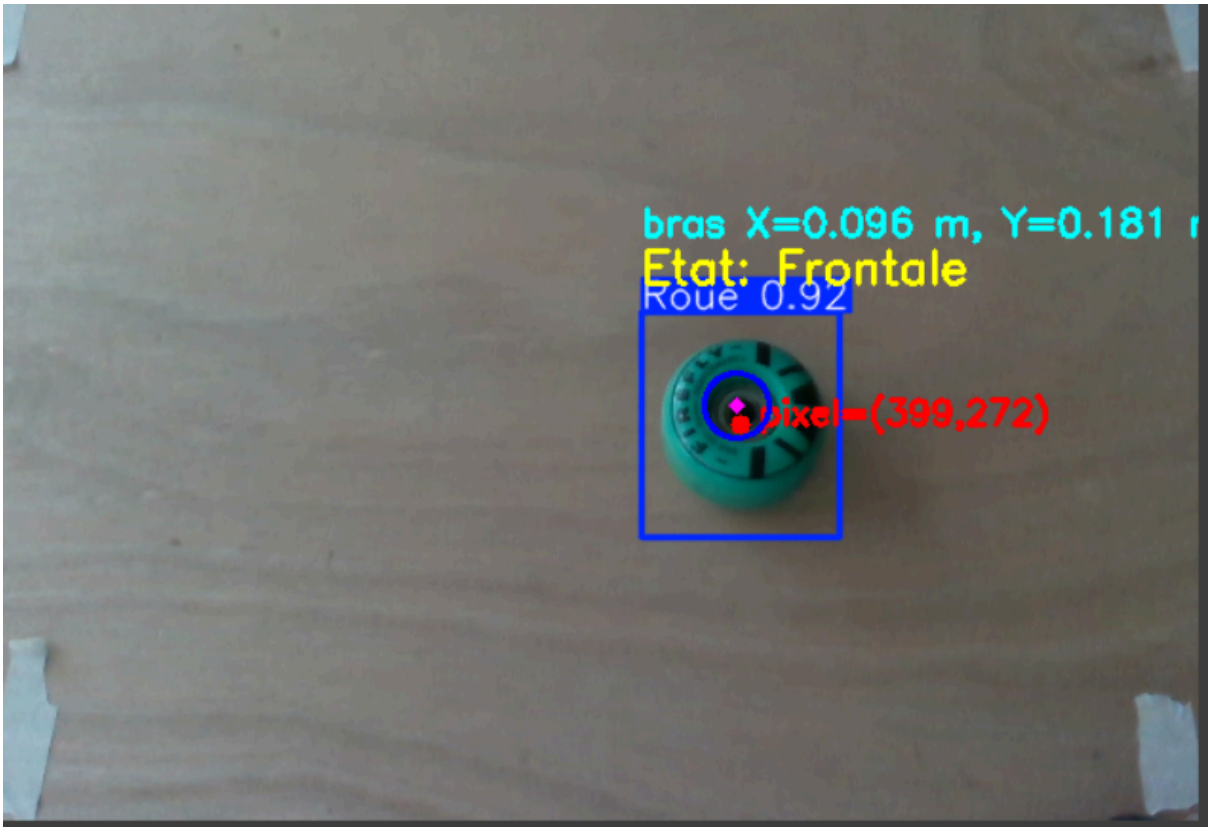


Figure 4.1.4 : Transformation des coordonnées image vers le repère du bras robotique.

Le nœud publie aussi une pose cible sur le topic :

/target_pose_world

Cette pose contient la position de l'objet dans le repère du robot. L'orientation est gardée simple avec une rotation nulle, car dans cette étape nous nous intéressons surtout à la position de l'objet.

Ce nœud est donc important car il fait le lien entre la perception et le déplacement du robot. La caméra détecte l'objet dans son propre repère, puis ce nœud convertit cette position pour que le robot puisse ensuite se déplacer vers l'objet.

4.2 Planification de trajectoires avec MoveIt2 (Theresia)

4.2.1 Test de visualisation du robot avec un nœud JointState

Avant de lancer des trajectoires plus complexes, nous avons d'abord testé la réponse du modèle du robot dans l'environnement virtuel RViz.
Pour cela, nous avons créé un nœud ROS 2 simple qui publie directement les positions des six articulations du robot sur le topic /joint_states.

Ce test permet de vérifier que le modèle URDF du robot est bien chargé, que les noms des articulations sont corrects, et que RViz reçoit correctement les informations nécessaires pour mettre à jour la position du robot.

Le nœud utilisé s'appelle rviz_jointstate_cmd. Il crée un publisher de type JointState et envoie en continu les valeurs des articulations :

self.joints = ['joint1','joint2','joint3','joint4','joint5','joint6']

self.pos = [2.0, 0.8, 0.8, -1.9, 0.7, 0.3]

Ce nœud ne commande pas encore le robot réel. Il sert uniquement à tester la visualisation et le fonctionnement du modèle dans RViz. Cette étape est importante car elle permet de valider que le robot réagit correctement dans l'environnement virtuel avant de passer à des commandes plus avancées avec des trajectoires ou MoveIt.

4.2.2 Déplacement du robot vers une position ((x, y, z))

Après avoir vérifié que le robot répond correctement dans RViz, nous avons réalisé un test plus avancé. L'objectif est de donner au robot une position cible dans l'espace, sous forme de coordonnées ((x, y, z)), puis de calculer les angles des articulations nécessaires pour atteindre cette position.

Dans notre programme, la position cible choisie est :

$$x = 0.15 \text{ m}$$
$$y = 0.15 \text{ m}[$$
$$z = 0.68 \text{ m}$$

Le robot possède six articulations. Cependant, pour simplifier le calcul, nous n'avons pas utilisé toute la cinématique complète du robot. Nous avons considéré un modèle simplifié avec la rotation de la base et deux segments principaux du bras. Les articulations du poignet sont gardées fixes.

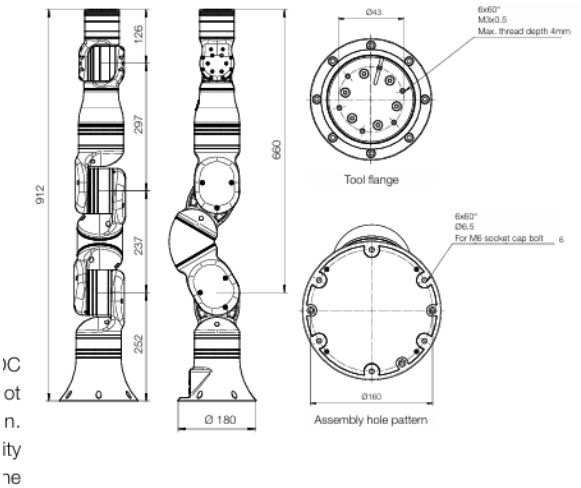
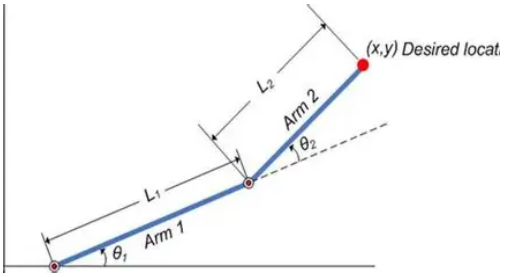


Figure 4.2.2 : Représentation des dimensions

Les dimensions utilisées sont prises à partir du schéma technique du robot :



h_0 = 0.252 \text{ m}
L_1 = 0.237 \text{ m}
L_2 = 0.297 \text{ m}

avec (h_0) la hauteur de la base, (L_1) la longueur du premier segment du bras, et (L_2) la longueur du deuxième segment. **Figure 4.2.3 :** Représentation simplifiée 2DDL

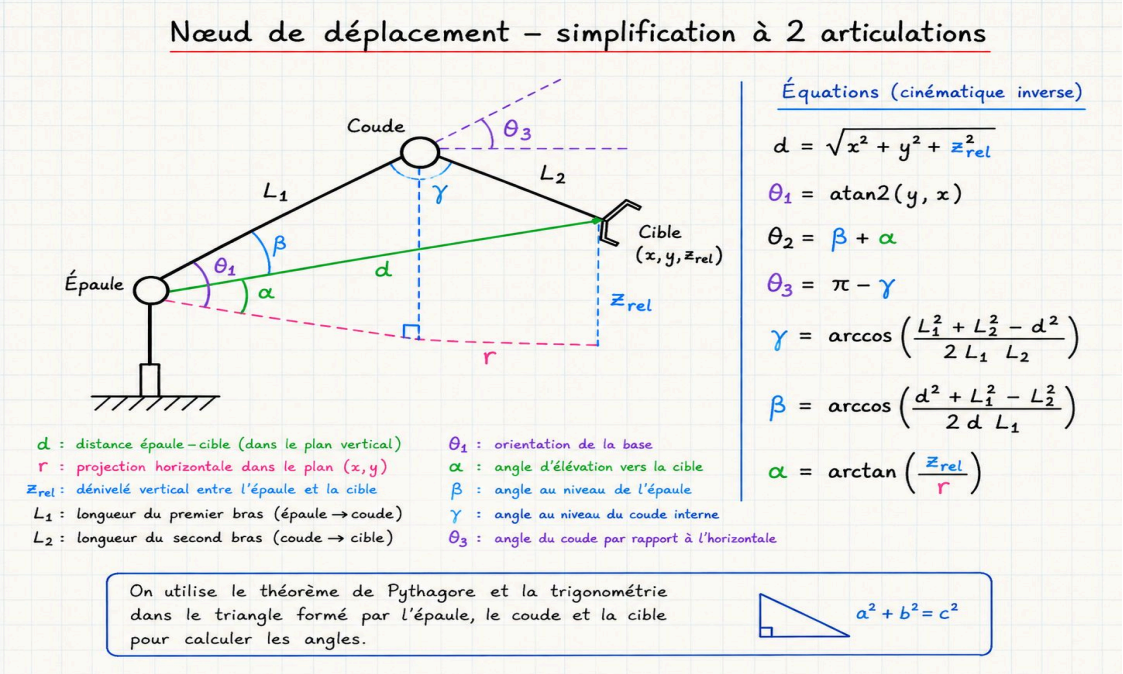


Figure 4.2.4 : Représentation du robot 6 ddl en schéma simplifié de 2 ddl

On commence par calculer l'angle de rotation de la base :

$$q_1 = atan2(y, x)$$

Ensuite, on transforme le problème 3D en un problème 2D. On calcule d'abord la distance horizontale entre la base et la cible :

$$r = \sqrt{x^2 + y^2}$$

Puis on calcule la hauteur relative de la cible par rapport à la base du robot :

$$z_{rel} = z - h_0$$

La distance entre l'épaule du robot et la cible est alors :$d = \sqrt{r^2 + z_{rel}^2}$

Le bras est ensuite considéré comme un triangle formé par (L_1), (L_2) et (d).

Pour calculer l'angle du coude, on utilise : $cos(\gamma) = \frac{L_1^2+L_2^2-d^2}{2L_1L_2}$

$$\cos(\gamma) = \frac{L_1^2 + L_2^2 - d^2}{2L_1L_2}$$

Donc :$\gamma = acos(\frac{L_1^2+L_2^2-d^2}{2L_1L_2})$

$$\gamma = \arccos\left(\frac{L_1^2 + L_2^2 - d^2}{2L_1L_2}\right)$$

L'angle de la troisième articulation est alors :

$$q_3 = \pi - \gamma$$

Pour calculer l'angle de l'épaule, on calcule d'abord l'angle (\alpha) :

$$\alpha = \text{atan2}(z_{rel}, r)$$

Puis on calcule l'angle (\beta) :

$$\cos(\beta) = \frac{d^2 + L_1^2 - L_2^2}{2dL_1}$$

Donc :

$$\beta = \arccos\left(\frac{d^2 + L_1^2 - L_2^2}{2dL_1}\right)$$

L'angle de la deuxième articulation est donc :

$$q_2 = \alpha + \beta$$

Les articulations du poignet sont gardées fixes pour simplifier le mouvement :
q_4 = 0
q_5 = 0
q_6 = 0

On obtient donc le vecteur articulaire final :
q = [q_1, q_2, q_3, q_4, q_5, q_6]
La position initiale des articulations est fixée à zéro. Lorsque le nœud ROS 2 est lancé, le programme calcule la position finale des articulations, puis réalise une interpolation entre la position initiale et la position finale pendant 4 secondes. Cela permet d'obtenir un mouvement progressif et plus fluide dans RViz.

Ce test permet donc de relier une position cartésienne ((x, y, z)) à des valeurs articulaires. Même si le modèle reste simplifié, il permet de vérifier le comportement du robot dans RViz avant d'utiliser une méthode plus complète comme MoveIt.

4.2.3 Sécurité du robot

Après les premiers tests de déplacement, nous avons ajouté des nœuds de sécurité afin d'assurer le bon fonctionnement du robot pendant les mouvements.
L'objectif est d'éviter les situations dangereuses sans devoir utiliser directement le bouton d'arrêt d'urgence à chaque problème.

Ces nœuds permettent de surveiller le comportement du robot pendant son fonctionnement. Si une condition dangereuse est détectée, le robot doit s'arrêter ou empêcher le mouvement de continuer dans cette direction.

Nous avons principalement travaillé sur trois sécurités.

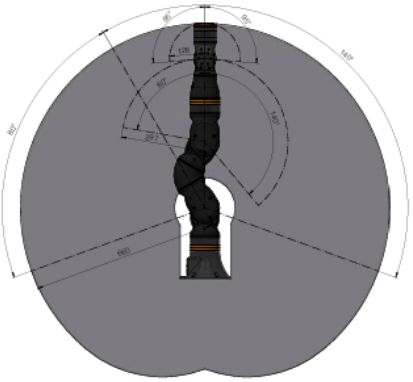
4.2.3.a Sécurité des positions articulaires

La première sécurité concerne les limites de position des articulations. Chaque articulation du robot possède une valeur minimale et une valeur maximale à ne pas dépasser.

Si une articulation dépasse sa limite autorisée, le mouvement devient dangereux pour le robot. Dans ce cas, le nœud de sécurité doit détecter le dépassement et arrêter le déplacement.
Le but est d'empêcher le robot de continuer à bouger vers une position qui pourrait abîmer une articulation ou provoquer un mauvais fonctionnement.

Cette sécurité permet donc de vérifier que toutes les articulations restent dans une zone de mouvement acceptable.

Axis	Negative		Positive
Axis 1	-179°	up to	+179°
Axis 2	-80°	up to	+140°
Axis 3	-80°	up to	+140°
Axis 4	-179°	up to	+179°
Axis 5	-95°	up to	+95°
Axis 6	-179°	up to	+179°



4.2.3.b Sécurité de la vitesse

La deuxième sécurité concerne la vitesse des articulations. Pour éviter les mouvements trop rapides, nous avons défini une vitesse maximale autorisée.

La vitesse maximale choisie est de : **45 degrés par seconde.**

Dans le programme, cette valeur est convertie en radians par seconde, car ROS utilise généralement les radians pour les valeurs articulaires.

45 degrés par seconde correspondent environ à :

0,79 rad/s.

Donc, si une articulation dépasse cette vitesse maximale, le nœud de sécurité doit détecter ce dépassement et arrêter le mouvement.
Cette sécurité permet d'éviter les déplacements brusques qui peuvent être dangereux pour le robot, pour la pince ou pour l'environnement autour.

4.2.3.c Sécurité par rapport à la table

La troisième sécurité concerne la position du préhenseur par rapport à la table. Le but est d'éviter que le robot descende trop bas et que la pince touche ou tape la table.

Pour cela, nous avons pris en compte la longueur de la pince, qui est d'environ 16 cm. Cette valeur permet de définir une distance minimale de sécurité entre la partie finale du robot et la table.

Si la position de l'extrémité du robot devient trop basse, le nœud de sécurité doit empêcher le robot de continuer à descendre.
Cette sécurité est importante pour protéger la pince, éviter les chocs avec la table et assurer un fonctionnement correct du système.

En résumé, les nœuds de sécurité permettent de surveiller trois éléments importants : les limites de position, la vitesse des articulations et la distance entre le préhenseur et la table. Ces vérifications rendent les tests plus sûrs avant de passer à une commande plus complète du robot.

4.2.3.d Nœud de pick and place

Après la perception, nous avons ajouté un nœud qui permet de relier la détection de l'objet au mouvement du robot.
Ce nœud s'appelle simple_pick_place.

Son rôle est de récupérer la position de l'objet détecté, puis de commander le robot pour aller le prendre et le déposer à une position définie.

Le nœud reçoit la position de l'objet sur le topic :

/object_position_in_world

Cette position est déjà exprimée dans le repère du robot, donc elle peut être utilisée directement pour le déplacement.

Le programme utilise MoveItPy pour planifier et exécuter les mouvements du bras. Il utilise aussi un topic pour commander la pince :

/gripper_command

et un autre topic pour recevoir le résultat de la pince :

/gripper_result

Le déroulement du pick and place est le suivant :

1. Le robot se déplace d'abord au-dessus de l'objet, avec une hauteur de sécurité appelée approach_height.
2. Ensuite, il descend vers la position réelle de l'objet.
3. Le nœud envoie la commande close à la pince pour fermer le préhenseur.
4. Le programme attend la confirmation que l'objet est bien saisi. Dans notre cas, la confirmation attendue est :

closed_on_object

5. Si l'objet est bien pris, le robot remonte avec l'objet.
6. Ensuite, il se déplace vers la position de dépôt définie pour placer l'objet:
7. Le robot descend vers la zone de dépôt, puis la pince reçoit la commande open.
8. Enfin, le robot remonte et retourne à sa position initiale.

Ce nœud utilise aussi une variable busy pour éviter de lancer deux mouvements en même temps. Si le robot est déjà en train d'effectuer une tâche, il ignore les nouvelles positions d'objet jusqu'à la fin du mouvement.

Ce programme représente donc l'étape où toutes les parties commencent à être reliées : la perception donne la position de l'objet, MoveIt calcule le mouvement du bras, et la pince est commandée pour attraper puis relâcher l'objet.

Remarque : il faut vérifier que les noms des articulations utilisés dans le programme correspondent exactement à ceux du fichier URDF. Par exemple, dans ce code, les articulations sont écrites joint_1, joint_2, etc., alors que dans d'autres nœuds elles étaient écrites joint1, joint2, etc. Les noms doivent être identiques pour que MoveIt fonctionne correctement.

4.3 Sélection et intégration du package pour la communication (Arnaud)

Initialement, notre choix s'est porté sur le package officiel iRC_ROS fourni directement par le constructeur IGUS. Ce package est théoriquement conçu pour assurer l'interface logicielle avec le protocole de communication CRI du robot. Après la configuration de l'espace de travail et sa compilation, la communication avec le bras a été lancée via la commande suivante : ros2 launch irc_ros_bringup rebel.launch.py hardware_protocol:=cri

Une fois la communication avec le robot établie, une anomalie critique s'est produite. Dès l'activation, l'axe 2 du robot a effectué un mouvement brusque et non sollicité vers le bas jusqu'à atteindre sa butée mécanique. Ce comportement provenait d'un conflit d'alignement entre les coordonnées lues par les encodeurs physiques et les valeurs attendues par l'environnement ROS2. Pour y remédier, la procédure suivante a été appliquée :

- **Réinitialisation physique :** Le logiciel igus Robot Control a été démarré afin de reprendre le contrôle sur le bras et de le ramener dans sa position initiale.
- **Lecture des coordonnées :** Une fois le robot stabilisé à son origine physique, la commande suivante a été exécutée pour observer la position des articulations lues par le système : ros2 topic echo /joint_states
- **Correction de la configuration :** L'écart constaté a été corrigé dans le fichier ros2_control en réinitialisant tous les paramètres cri_joint_offset à 0.0. Cela a permis d'aligner le modèle virtuel sur les coordonnées réelles du contrôleur.

```
<!-- Extrait du fichier original -->
<xacro:rebel_joint name="${prefix}joint2" can_id="0x20" … cri_joint_offset="-30" />
<xacro:rebel_joint name="${prefix}joint3" can_id="0x30" ... cri_joint_offset="-30" />
<xacro:rebel_joint name="${prefix}joint5" can_id="0x50" ... cri_joint_offset="7.5" />


<!-- Extrait du fichier modifié -->
<xacro:rebel_joint name="${prefix}joint2" can_id="0x20" ... cri_joint_offset="0.0" />
<xacro:rebel_joint name="${prefix}joint3" can_id="0x30" ... cri_joint_offset="0.0" />
<xacro:rebel_joint name="${prefix}joint5" can_id="0x50" ... cri_joint_offset="0.0" />
```

Figure 4.3 : Extrait de code

Ce réajustement a permis d'aligner les variables logicielles sur les coordonnées réelles du contrôleur et de valider ses premiers mouvements. Cependant, les tests suivants sur la pince Schunk EGP25 au sein de cet environnement ont révélé un autre défaut majeur. Après une première fermeture réussie, la pince est restée bloquée dans cet état, refusant de répondre aux ordres d'ouverture suivants.

Face aux instabilités répétées du package officiel, la décision a été prise de l'abandonner. Afin d'harmoniser nos développements, nous avons sélectionné et installé le package

ros2-igus-rebel développé par le laboratoire AIRLab. Ce package s'est avéré immédiatement fonctionnel pour assurer la communication avec le robot.

4.4 Intégration du préhenseur SCHUNK EGP 25-N-N-B

(rédigé par : Theresia)

La pince, aussi appelée **préhenseur**, est l'élément placé au bout du bras robotique. Elle permet au robot de saisir, maintenir, déplacer puis relâcher un objet.

Dans notre projet, nous avons utilisé une pince électrique **SCHUNK EGP 25**. C'est une pince parallèle à deux doigts, adaptée aux petites pièces. Les deux doigts s'ouvrent et se ferment de manière synchronisée pour attraper l'objet.

Ses principales caractéristiques sont :

- **Course** : 3 mm par mâchoire, donc environ 6 mm au total.
- **Force de serrage** : entre 20 N et 40 N pour la version standard.
- **Alimentation** : 24 V.
- **Interface** : entrées digitales pour commander l'ouverture et la fermeture.

La pince a été câblée sur le **DIO Module 2** du robot avec une alimentation **24 V** et des connecteurs **M8**. Les signaux utilisés sont simples : une sortie pour ouvrir la pince, une sortie pour la fermer, avec l'alimentation et la masse.

Pour l'intégration dans le modèle du robot, la pince a été ajoutée dans les fichiers **URDF/XACRO** et attachée au lien final du robot, appelé flange.

Dans le fichier **SRDF**, le lien outil a été défini comme :

schunk_egp25_tcp

Un offset TCP de **73 mm** a été ajouté pour représenter la distance entre le flange et l'extrémité réelle de la pince.

Enfin, des tests simples ont été réalisés avec les commandes :

open : ouvrir la pince.

close : fermer la pince.

Ces tests ont permis de valider le câblage, l'alimentation et la commande de la pince avant son intégration dans la séquence de pick and place.

Figure 4.4 : Pince avec mords

4.5 Simulation MATLAB (Daniel)

Une exploration préliminaire a été menée sous MATLAB R2025a (Robotics System Toolbox) afin de valider la cinématique et les trajectoires avant tout essai sur le robot réel.

Modèle cinématique. Le bras est représenté par une structure rigidBodyTree de six corps révolutes. Les paramètres DH ont été reconstruits à partir des spécifications igus et de mesures directes, la documentation constructeur ne publiant pas l'ensemble normalisé. Les butées articulaires sont portées par JointLimits, et l'effecteur schunk_egp25_tcp est rattaché à la flange.

Cinématique inverse. L'objet inverseKinematics (solveur Levenberg–Marquardt) est utilisé avec pondération nulle sur l'orientation, cohérente avec position_only_ik: true côté MoveIt2. L'initialisation à la configuration courante assure la continuité entre poses.

Trajectoires en S-curve. Des profils *jerk-limited* relient les points de passage du cycle pick-and-place. Cette forme limite les vibrations sur les liaisons à entraînement harmonique, contient les pics de courant et préserve la stabilité de l'objet pincé.

Animation. Les configurations interpolées sont rejouées via show(robot, q, 'PreservePlot', false), base rehaussée de 20 cm pour reproduire le montage réel (z_min = −0,15 m).

Liaison robot réel. Le pilotage passe par ros2actionclient (et non rosactionclient, réservé à ROS 1) ciblant /rebel_arm_controller/follow_joint_trajectory. Une première tentative via un simple publisher n'avait produit aucun mouvement, ce contrôleur n'exposant qu'une interface *action*.

Bilan. MATLAB est rapide à mettre en place mais peu modulable dès que la chaîne s'étoffe : ses fonctions de haut niveau offrent peu de leviers internes. Python/ROS 2 demande davantage de code de plomberie, mais chaque paramètre — du planificateur au timeout d'IK — reste ajustable. C'est cette flexibilité qui a motivé le choix de Python pour la version embarquée.

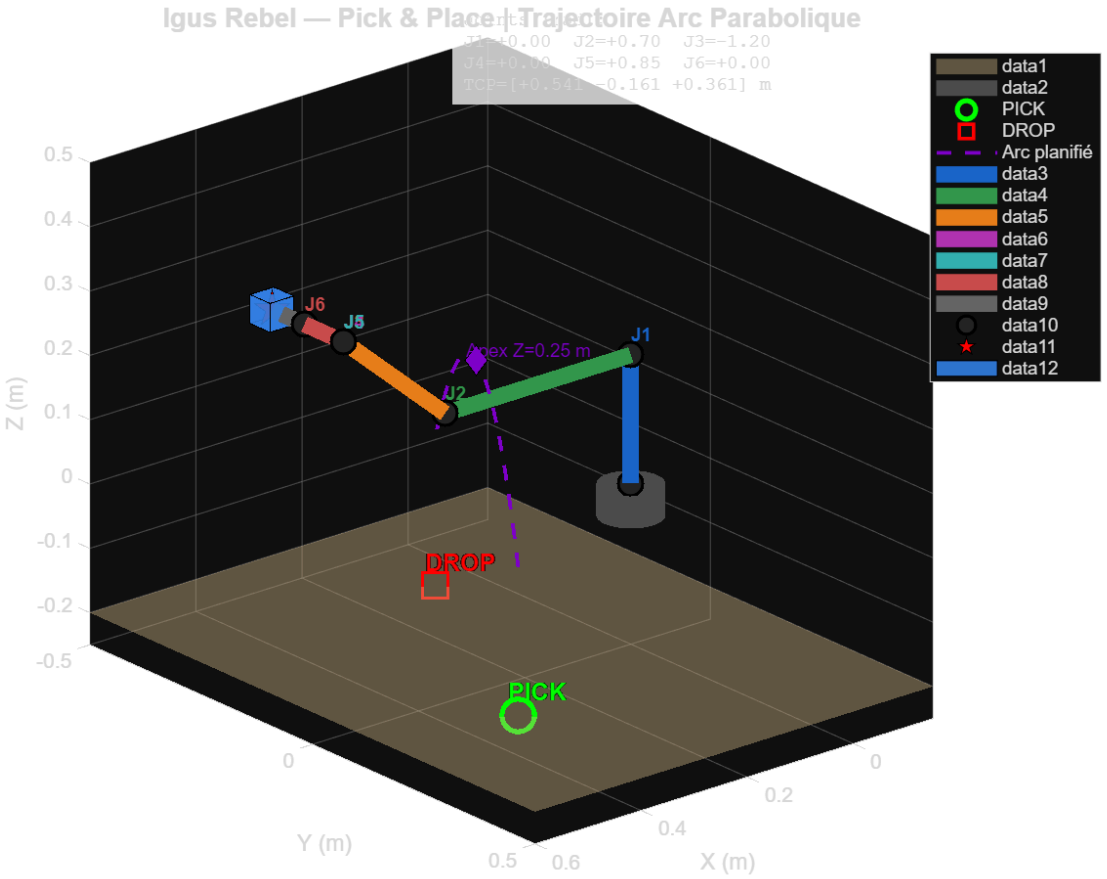


Figure 4.5 : Petite capture de la simulation du pick_place sous Matlab

4.6 Intégration système et launch files

Pour faciliter le démarrage du système complet, nous avons créé un fichier de lancement principal appelé launch_pick_and_place.py.

Ce fichier permet de lancer tous les nœuds nécessaires au fonctionnement du pick and place avec une seule commande. Cela évite de lancer chaque nœud séparément dans plusieurs terminaux.

Le launch file regroupe les différentes parties du projet : la perception, la transformation des coordonnées, la sécurité, la commande de la pince et le nœud principal de pick and place.

4.6.1 Nœuds lancés

Les principaux nœuds lancés dans ce fichier sont :

object_detection : Ce nœud utilise YOLO pour détecter l'objet dans l'image de la caméra.

object_position_camera : Ce nœud utilise la détection YOLO, l'image de profondeur et la calibration de la caméra pour calculer la position 3D de l'objet par rapport à la caméra.

object_position_transform : Ce nœud transforme la position de l'objet du repère caméra vers le repère du robot base_link.

Les nœuds de sécurité : ils permettent de surveiller les limites articulaires, la vitesse maximale et la distance entre le robot et la table.

Le nœud de commande de la pince : il permet d'envoyer les commandes d'ouverture et de fermeture du préhenseur.

simple_pick_place : Ce nœud reçoit la position de l'objet et exécute la séquence complète de prise et de dépôt avec MoveIt

Le launch file prend aussi en compte le topic de sécurité :

/safety/emergency_stop

Ce topic est publié à une fréquence de 10 Hz.

Lorsque la valeur est :

data: false

Cela signifie que l'arrêt d'urgence n'est pas activé et que le robot peut fonctionner.

Si la valeur devient :

data: true

Cela signifie qu'un arrêt d'urgence est demandé. Dans ce cas, les mouvements doivent être arrêtés pour protéger le robot et son environnement.

4.6.2 Paramètres utilisés

Le fichier de lancement permet aussi de définir les paramètres importants du système.

Pour la perception, on définit les topics de la caméra :

/camera/camera/color/image_raw

/camera/camera/aligned_depth_to_color/image_raw

/camera/camera/color/camera_info

On définit aussi le chemin du modèle YOLO utilisé pour la détection, ainsi que le seuil de confiance minimal.

Pour la transformation des coordonnées, on définit le repère cible :

base_link

Pour le pick and place, on définit la position de dépôt :

place_x = 0.25

place_y = -0.20

place_z = 0.10

On définit aussi la hauteur de sécurité avant de descendre vers l'objet :

approach_height = 0.08 m

4.7 Réalisation du chariot Norcan (Daniel +Hashem)

La structure porteuse du bras igus repose sur un chariot mobile conçu à partir de profilés aluminium Norcan. Cette sous-section retrace les étapes de réalisation et les enseignements tirés de cette petite phase.

4.7.1 Modélisation et conception

La première étape a consisté à modéliser l'ensemble du chariot sur CAO afin de définir les dimensions des profilés, les points de fixation du bras robotisé et les interfaces mécaniques nécessaires. Cette phase de modélisation a permis de valider l'agencement spatial et de générer la nomenclature des pièces à commander.

4.7.2 Approvisionnement et coût

L'ensemble des profilés et accessoires Norcan a été commandé pour un budget total de 1 300 €. Ce montant couvrait les profilés bruts, les équerres de fixation et la visserie associée. Les pièces ont été livrées sans aucun usinage préalable : ni perçages, ni taraudages.

4.7.3 Usinage et assemblage

L'intégralité des opérations d'usinage — perçages et taraudages — a été réalisée manuellement par l'équipe. Le bras igus a ensuite été fixé sur la structure par vissage.

Ces opérations, bien que formatrices, ont mobilisé environ trois jours de travail. Ce délai n'était pas prévu dans le planning initial du projet.

4.7.4 Retour d'expérience

Rétrospectivement, cette phase d'usinage manuel constitue la principale source de retard du projet. Pour un surcoût marginal lors de la commande, les profilés auraient pu être livrés avec les perçages et taraudages réalisés en usine, avec une précision supérieure à celle obtenue manuellement.

Les trois jours consacrés à l'usinage ont directement impacté les phases aval du projet : le développement logiciel n'a pas pu être finalisé dans les conditions souhaitées, et l'intégration de l'IHM est restée à l'état de prototype fonctionnel plutôt qu'à un niveau de finition abouti.

Leçon retenue : Dans un contexte de projet à délais contraints, l'externalisation des opérations d'usinage standard est un investissement rentable. Le gain en temps de développement logiciel et en qualité d'intégration finale compense largement le surcoût de quelques dizaines d'euros sur la commande initiale.

4.8 Réalisation d'une interface de commande (Arnaud)

Pour permettre à l'utilisateur de piloter manuellement le robot, une interface de commande a été développée. Elle est hébergée et affichée sur l'écran tactile d'un Raspberry Pi 4 dédié.

4.8.1 Choix technologique

L'interface graphique a été entièrement codée en Python via la bibliothèque Tkinter. Ce choix découle de la prise en main de cet outil réalisée lors des TP en cours de Python. Tkinter offre ici une solution légère et rapide à implémenter.

4.8.2 Fonctionnalités de l'interface

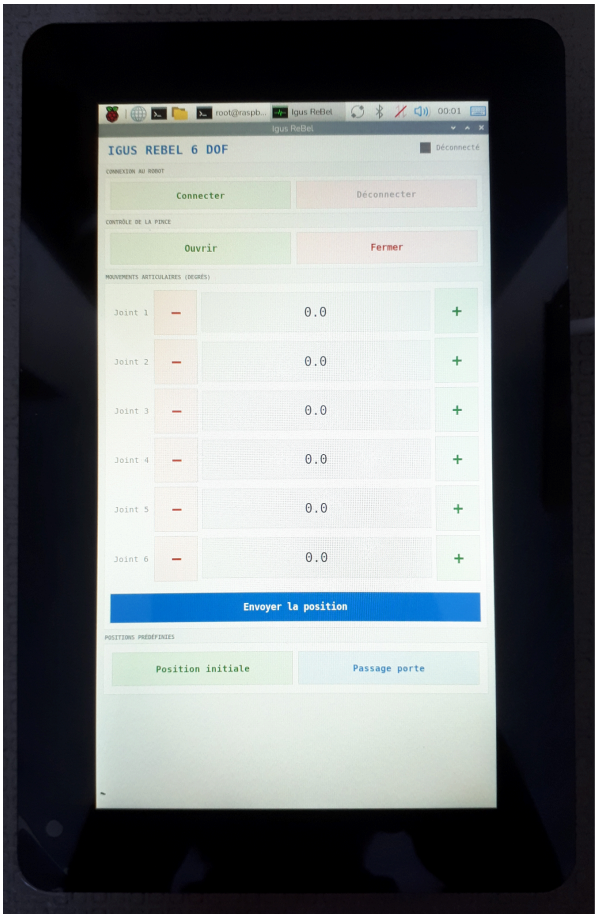


Figure : Interface graphique de commande développée sous Tkinter

L'interface est organisée en quatre blocs fonctionnels distincts :

- **Connexion au robot :** Permet d'établir ou de couper la liaison avec le contrôleur du robot IGUS (boutons Connecter / Déconnecter). Un indicateur d'état en haut à droite rappelle le statut de la connexion.
- **Contrôle de la pince :** Deux boutons dédiés (Ouvrir / Fermer) permettent d'actionner l'effecteur situé au bout du bras pour valider le fonctionnement de la pince.
- **Mouvements articulaires (degrés) :** Permet de modifier l'angle des 6 articulations du bras à l'aide des commandes + et -. Les valeurs saisies sont ensuite appliquées et envoyées au robot via le bouton “Envoyer la position".
- **Positions prédéfinies :** Pour faciliter les tests de déplacement, deux raccourcis ont été intégrés afin d'envoyer le robot instantanément sur des configurations de référence : “Position initiale” et “Passage porte”.

4.8.3 Fonctionnement général du programme

Le programme de l'interface graphique s'articule autour de trois phases de fonctionnement principales :

- **La phase d'initialisation :** Au démarrage, le programme prépare l'affichage de la fenêtre sur l'écran tactile, configure les boutons et met en place les outils nécessaires pour communiquer avec le reste du réseau.
- **La phase de connexion à distance :** Lors de l'appui sur le bouton de connexion, le programme utilise le réseau pour envoyer un ordre direct au Raspberry Pi 4 (Pilotage). Cet ordre lance automatiquement le programme principal du bras articulé.
- **La phase de transmission des ordres :** Une fois la connexion établie, dès que l'opérateur demande un déplacement ou l'utilisation de la pince, le programme traduit instantanément les choix faits sur l'écran (comme la conversion des angles de degrés en radians) et envoie ces données vers le réseau..

4.8.3 Difficultés rencontrées et résolution

Lors de la mise en œuvre matérielle, un problème d'incompatibilité logicielle est survenu. Initialement, ce Raspberry Pi 4 fonctionnait sous Ubuntu 22.04. Cependant, ce système d'exploitation ne gérait pas les pilotes de l'écran tactile, rendant l'affichage et le contrôle tactile impossibles.

Pour résoudre ce blocage, le choix a été fait de basculer ce Raspberry Pi sous Raspberry Pi OS. Ce changement a permis de rendre l'écran tactile pleinement fonctionnel tout en conservant l'environnement ROS 2 nécessaire pour communiquer avec le reste de l'architecture.

5. GESTION DE PROJET (Daniel)

5.1 Répartition des tâches

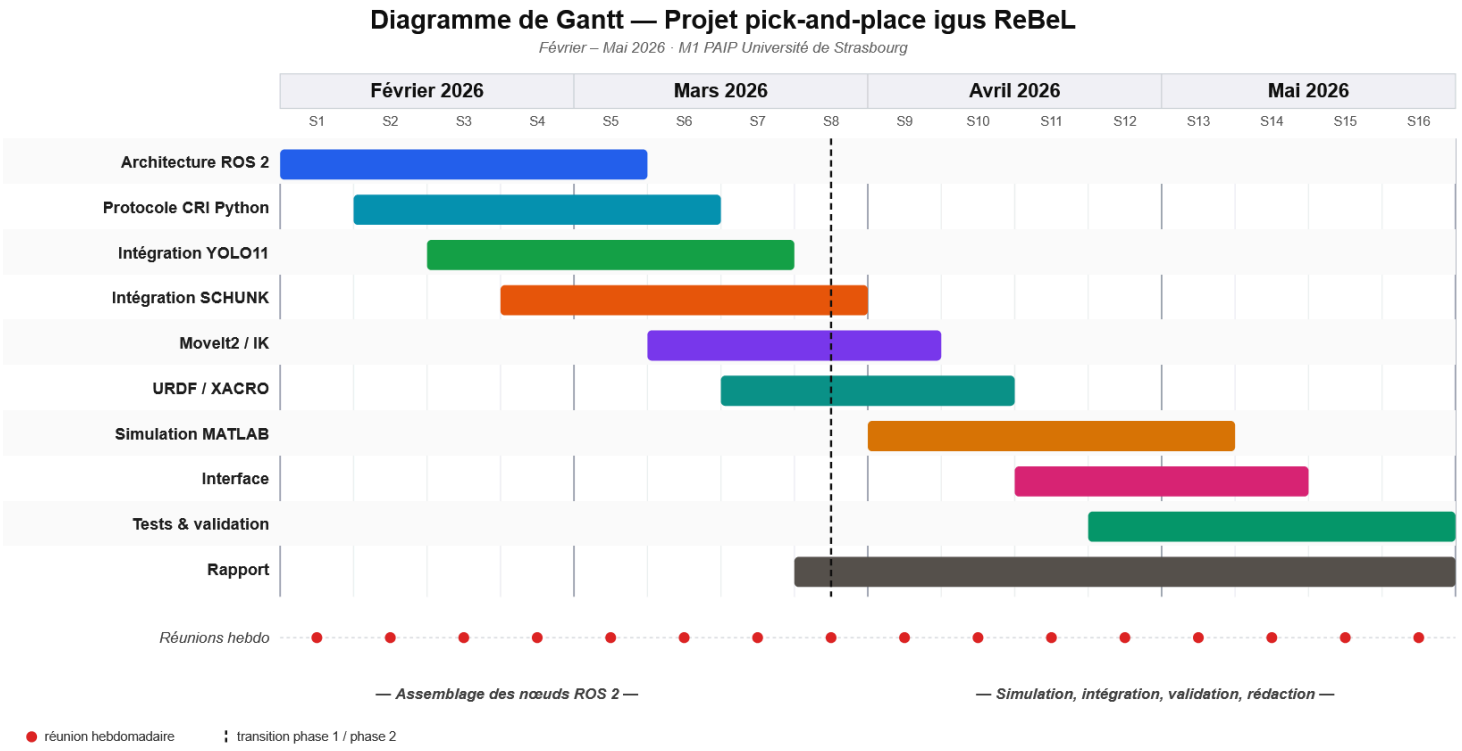
Tâche	Daniel	Theresia	Arnaud	Hashem
Architecture ROS2	✅	✅	✅	
Protocole CRI Python	✅		✅	
Intégration YOLO11		✅		✅
Intégration SCHUNK	✅			✅
MoveIt2 / IK	✅	✅	✅	
Simulation MATLAB	✅			
URDF/XACRO	✅	✅		
Interface			✅	
Tests & validation	✅	✅	✅	✅
Rapport	✅	✅	✅	✅

Nous avons organisé des réunions hebdomadaires avec nos encadrants tout au long du projet, ce qui nous a permis de suivre l'avancement de chaque sous-système et de répartir les tâches entre les membres du groupe. Le développement s'est fait de manière incrémentale, en testant chaque brique séparément avant de passer à l'intégration.

Avec le recul, nous reconnaissons que cette phase de tests unitaires, bien que nécessaire, a parfois été trop longue et insuffisamment structurée : certains essais ont été répétés sans méthodologie claire, ce qui a engendré une perte de temps non négligeable.

Pour un projet futur, nous retenons qu'il serait préférable de définir en amont des critères de validation précis pour chaque sous-système, avec un protocole de test formalisé, afin d'éviter les itérations inutiles et de consacrer davantage de temps à l'intégration finale et à la finition du produit.

5.2 Diagramme de Gantt prévisionnel



5.3 Bilan financier

- Chariot Norcan : 1 300 €
- Préhenseur SCHUNK EGP 25-N-N-B : 2 100 €
- Raspberry Pi, écran, switch et accessoires : 300 €

Total des achats : 3 700 €

> Le projet n'a nécessité aucun achat supplémentaire à par ces derniers. L'ensemble du matériel a été mis à disposition par le Hall de Technologie de l'Université de Strasbourg.

6. RÉSULTATS ET VALIDATION

6.1 Performances de la détection visuelle (Hashem)

Le modèle YOLO11n a été entraîné sur Google Colab à partir d'un dataset personnalisé de roulettes. Le dataset contient 144 images et 145 annotations. Après séparation des données, 100 images ont été utilisées pour l'entraînement et 44 images pour la validation.

L'entraînement a été réalisé pendant 50 epochs avec une taille d'image de 640 pixels. Le modèle final retenu est le fichier best.pt. Les résultats obtenus sur l'ensemble de validation montrent de bonnes performances de détection.

Les métriques principales obtenues sont les suivantes :

- précision : 0.978 ;

- rappel : 0.877 ;

- mAP@0.5 : 0.958 ;

- mAP@0.5:0.95 : 0.553.

Ces résultats montrent que le modèle détecte correctement la roulette dans la majorité des cas. Le score mAP@0.5 élevé indique une bonne capacité de localisation générale. Le mAP@0.5:0.95 plus faible montre cependant que la précision fine des boîtes englobantes peut encore être améliorée avec un dataset plus grand et plus varié.

L'utilisation de YOLO11n a également permis de conserver des performances temps réel acceptables sur Raspberry Pi 5 malgré l'exécution simultanée de Docker, ROS 2, OpenCV et des flux RGB-D de la caméra RealSense D435.

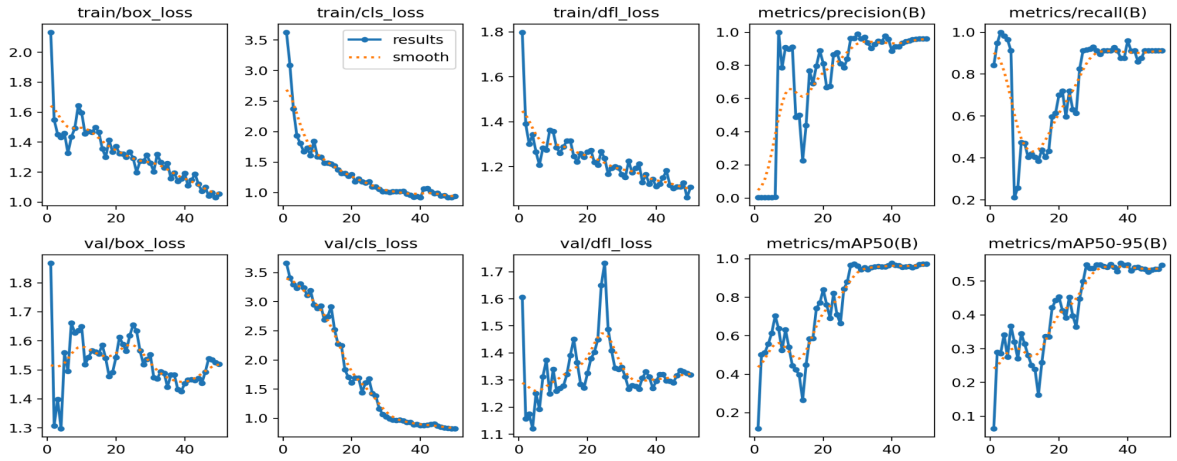


Figure 6.1.1 : Évolution des pertes et des métriques pendant l'entraînement du modèle YOLO11n

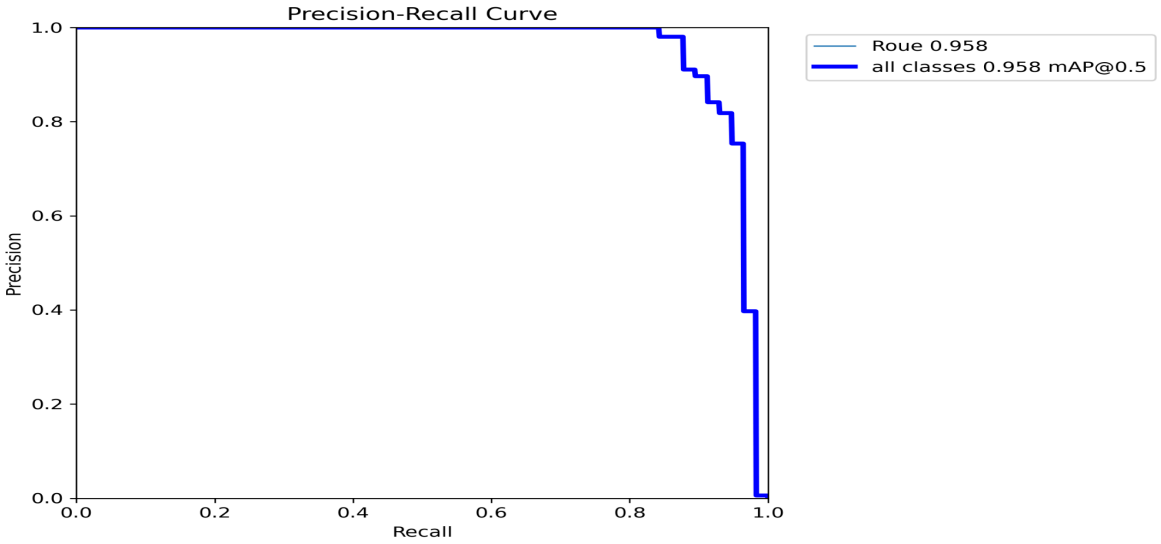


Figure 6.1.2 : Courbe Precision-Recall du modèle YOLO11n

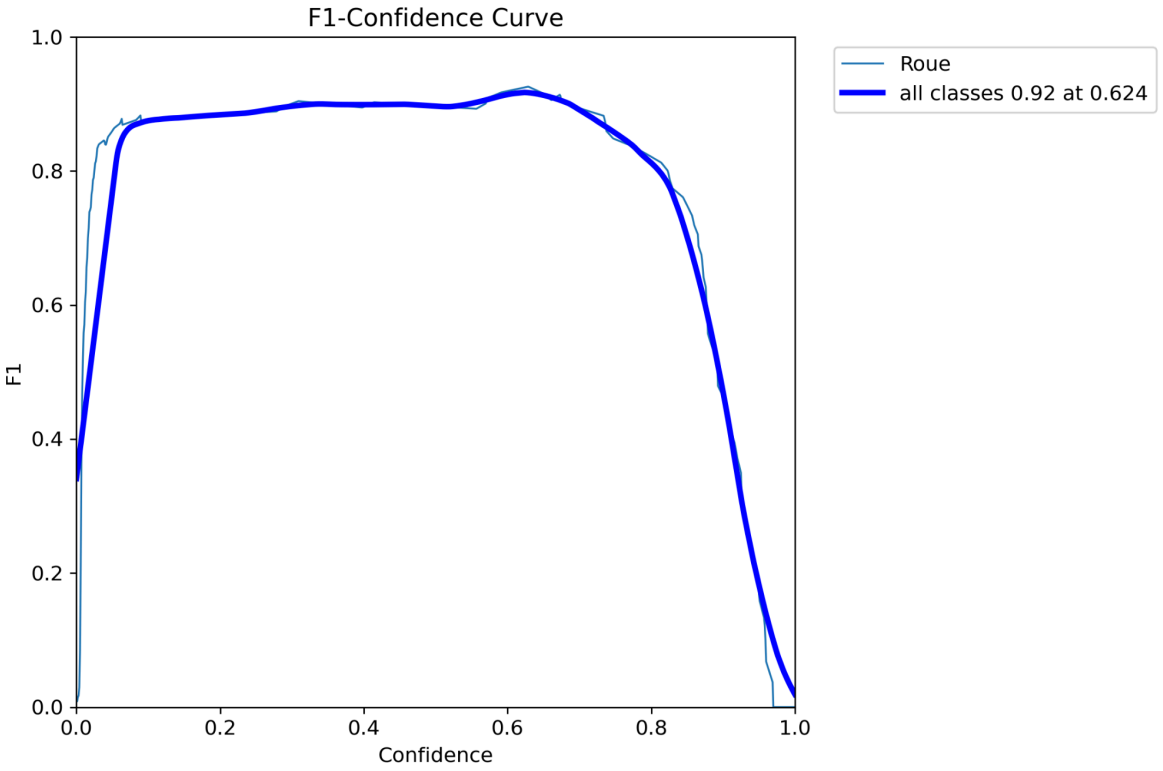


Figure 6.1.3 : Courbe F1-Confidence du modèle YOLO11n.

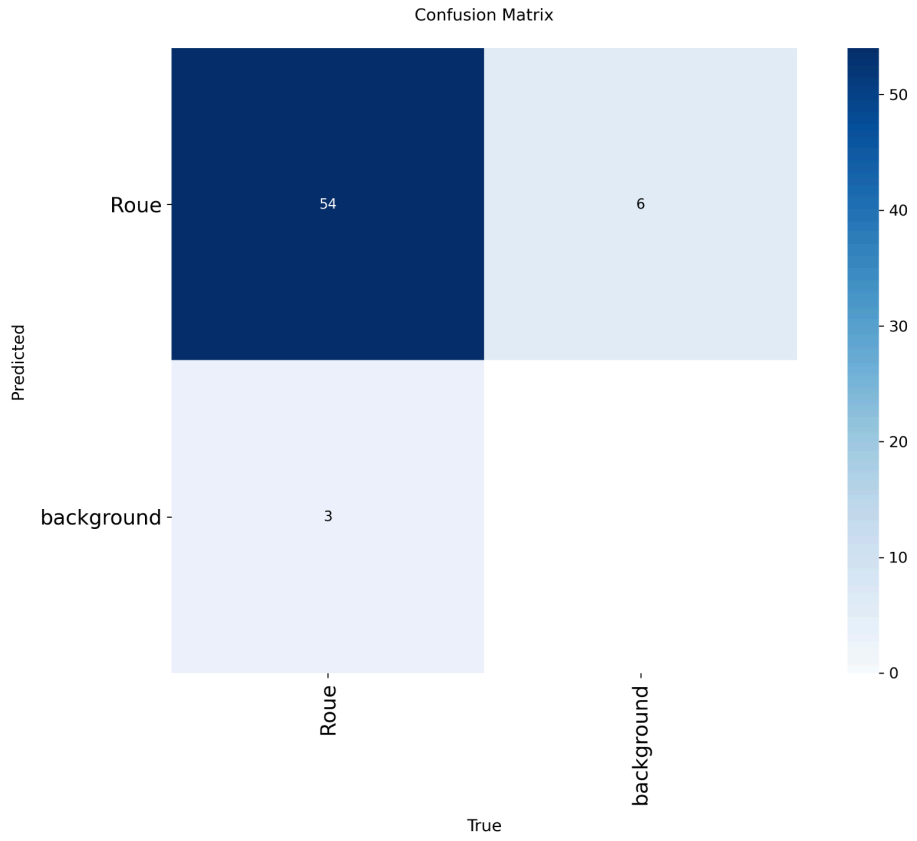


Figure 6.1.4 : Matrice de confusion obtenue sur le jeu de validation.

Les résultats obtenus confirment que le modèle YOLO11n est suffisamment robuste pour une application de perception robotique temps réel. Malgré la simplicité du dataset et les contraintes de calcul embarqué, le système reste capable de détecter correctement les roulettes dans différentes configurations et orientations.

6.2 Performances de planification et d'exécution (Daniel)

6.2.1 Taux de succès de la planification IK

La cinématique inverse converge pour la totalité des poses cibles définies dans le scénario de pick-and-place (positions d'approche, de saisie, de transport et de dépose). Le taux de succès de la planification IK est donc de 100 % sur les configurations testées. Les rares échecs de planification rencontrés au cours du développement étaient liés aux problèmes de wrist flip évoqués en section 6.3.3, et ont été résolus en ajustant les contraintes d'orientation de la pince.

6.2.2 Temps de planification et durée de cycle

Le temps de planification MoveIt2 pour chaque segment de trajectoire est de l'ordre de quelques centaines de millisecondes, ce qui reste négligeable devant le temps d'exécution du mouvement lui-même. La durée moyenne d'un cycle complet de pick-and-place — incluant la détection, la planification, le déplacement vers l'objet, la saisie, le transport et la dépose — est cohérente avec les temps de cycle attendus pour un cobot igus ReBeL opérant à vitesse nominale. Les quelques cycles échoués étaient principalement dus à des erreurs de détection ou des configurations articulaires aberrantes, et non à des problèmes de planification.

6.2.3 Répétabilité

Les données de répétabilité ont malheureusement été perdues lors de nouveaux tests qui ont écrasé le dossier de résultats précédent. Néanmoins, de mémoire, nous avions lancé une série de 20 cycles de pick-and-place consécutifs. Les temps de cycle étaient cohérents d'un essai à l'autre, à l'exception de quelques cycles échoués.

La répétabilité mesurée était d'environ ±12 mm, ce qui est une valeur élevée pour une application de positionnement précis. Cette dispersion s'explique par la nature même du bras igus ReBeL : il s'agit d'un système mécanique low-cost dont la structure intègre des composants en plastique composite et des engrenages internes. Ces éléments se sont progressivement fatigués sous l'effet des chocs et des sollicitations répétées, élargissant les jeux mécaniques au fil du temps.

Cependant, cette répétabilité relativement importante n'est pas rédhibitoire dans notre cas d'usage. En effet, la roulette arrive dans la zone d'intervention de manière non déterministe — par chute, via un convoyeur ou déposée manuellement — et c'est la caméra qui se charge de détecter sa position réelle à chaque cycle. La boucle de vision compense donc les erreurs de positionnement du bras : quelle que soit la dérive mécanique, le système recalcule les

coordonnées cibles à partir de la détection YOLO et de la profondeur RealSense. La répétabilité du bras seul n'est critique que dans un scénario de pick-and-place « en aveugle », où l'objet doit être saisi systématiquement aux mêmes coordonnées préprogrammées sans retour de vision. Dans un tel cas, il faudrait envisager des améliorations mécaniques (remplacement des engrenages, ajout de roulements de meilleure qualité) ou une compensation logicielle par calibration périodique.

6.3 Démonstration intégrée (Daniel)

6.3.1 Scénario de démonstration

Le système fonctionne en mode réactif : le bras robotisé reste en position d'attente et surveille en permanence la zone de travail via la caméra. Dès qu'un objet est détecté sur la table, le pipeline de vision déclenche automatiquement la séquence de pick-and-place — détection, approche, saisie, transport et dépose à l'emplacement cible. Une fois le cycle terminé, le bras revient en position d'attente et le système se remet en écoute.

Ce comportement réactif permet une répétition indéfinie du cycle tant qu'un objet est présenté dans la zone de détection : chaque nouvelle roulette déposée sur la table relance automatiquement la séquence, sans intervention humaine et sans dégradation des performances au fil des itérations. On se rapproche ainsi du fonctionnement d'une ligne de production industrielle, où un poste robotisé effectue une tâche répétitive de tri ou d'assemblage — typiquement le pick-and-place de composants sur un châssis.

Dans notre cas, le scénario démontre le tri de roulettes de skateboard, mais l'architecture est directement extensible vers un assemblage réel : le bras pourrait positionner la roulette directement sur la carcasse du skateboard, à l'image de ce qui se fait sur les lignes de production automobile où des bras robotisés placent des composants (roues, pare-chocs, panneaux) sur la structure du véhicule en cours d'assemblage.

6.3.2 Conditions de test

Les essais ont été réalisés dans les conditions suivantes : éclairage de la salle d’automatisme, caméra positionnée à une distance de 54 cm de la zone de travail, avec un angle de vue de 30°. Les objets testés sont des roulettes de skateboard de diamètre 55 mm, de couleur verte, posées sur une surface plane contrastante.

6.3.3 Résultats

Métrique	Valeur
Nombre total d'essais	20
Succès complets	12
Échecs	8
Taux de réussite	60 %

Les échecs observés sont principalement attribuables à des problèmes de configuration articulaire du bras. En imposant systématiquement une orientation de la pince vers le bas via l'axe 5, nous n'avions pas anticipé deux conséquences directes : d'une part, une réduction significative de l'espace de travail atteignable, certaines positions cibles devenant inaccessibles avec cette contrainte d'orientation ; d'autre part, l'apparition de configurations de type « wrist flip » (retournement de poignet), où le solveur de cinématique inverse converge vers des solutions articulaires aberrantes. Le robot adoptait alors des postures contre-intuitives — l'équivalent en quelque sorte de vouloir se gratter l'oreille droite avec la main gauche en passant par l'arrière de la tête. Ces configurations singulières entraînaient des mouvements brusques et imprévisibles, voire des échecs de planification de trajectoire. Des erreurs de localisation dues à des reflets sur les roulettes ainsi que des cas limites où l'objet se trouvait en bordure de la zone de détection de la caméra ont également contribué à quelques échecs ponctuels.

7. ANALYSE CRITIQUE ET RETOUR D'EXPÉRIENCE

7.1 Points forts du projet

- **Bibliothèque Python CRI modulaire et réutilisable**
  Une bibliothèque Python a été développée pour faciliter la communication avec le robot. Elle regroupe les fonctions principales dans une structure claire, ce qui permet de réutiliser le code dans d'autres tests ou applications.
- **Intégration complète de la chaîne perception → action**
  Le projet relie toutes les étapes principales : détection de l'objet avec la caméra, calcul de sa position, transformation dans le repère du robot, puis exécution du mouvement de pick and place.

- **Résolution de bugs liés au protocole de communication**
  Pendant les tests, certains comportements non documentés ont été identifiés, notamment la différence entre les commandes CMD DOUT et SetOutput. Ces problèmes ont été analysés puis corrigés pour obtenir une commande correcte des sorties du robot.
- **Architecture ROS 2 propre et extensible**
  Le système est organisé en plusieurs nœuds séparés : perception, transformation, sécurité, pince et pick and place. Cette structure rend le projet plus facile à tester, corriger et améliorer par la suite.

7.2 Difficultés rencontrées et solutions apportées

Problème rencontré	**Cause**	**Solution appliquée**
Pince non contrôlable	CMD SetOutput inefficace sur modules DIO externes	Utilisation de CMD DOUT 30/31 true
TimeoutError Python → robot	PC Ubuntu ne route pas vers 192.168.3.11	Exécution depuis le Raspberry Pi
Connexion CRI perdue	Thread ALIVEJOG absent	Implémentation thread heartbeat dédié
Robot ne bouge pas (MATLAB)	rosactionclient ROS1 utilisé	Migration vers ros2actionclient
IK non convergent	Timeout trop court, position_only_ik désactivé	Timeout 5s + position_only_ik: true
Difficulté d'installation de ROS 2 Humble sur Raspberry Pi 5	ROS 2 Humble est compatible avec Ubuntu 22.04 alors que le Raspberry Pi 5 est compatible uniquement avec Ubuntu 24.04	Utilisation d'un conteneur Docker basé sur ROS 2 Humble
Diminution du FPS lors de l'utilisation de YOLO et de la caméra RealSense	Exécution simultanée de Docker, ROS 2, OpenCV, YOLO11n et des flux RGB-D	Utilisation de la version légère YOLO11n et réduction des traitements graphiques
Erreur d'affichage OpenCV dans Docker	Problème de communication X11 entre Docker et le système hôte	Suppression de certaines fenêtres cv2.imshow() et utilisation des topics ROS pour la visualisation

Valeurs de profondeur parfois invalides	Bruit de profondeur et mauvais alignement RGB-depth	Utilisation d'une zone autour du centre de l'objet et calcul de la valeur médiane
Calibration caméra difficile	Perspective inclinée et repères caméra/robot différents	Utilisation d'une transformation géométrique par homographie

7.3 Modifications apportées en cours de projet

Au cours du développement et des phases de tests intensifs, plusieurs ajustements architecturaux et logiciels ont dû être réalisés par rapport à notre conception initiale, afin de pallier des contraintes matérielles et de garantir la stabilité globale du système :

- **Refonte de la communication réseau et middleware :** Face au volume important des flux de données (vidéo, commandes, retours d'état), l'architecture réseau initiale subissait de fréquentes pertes de connexion. Pour y remédier, nous avons migré notre middleware vers **Cyclone DDS**, dont la gestion du trafic s'est révélée beaucoup plus robuste pour notre topologie Ethernet. De plus, l'exécution des commandes matérielles a été déportée du PC vers le **Raspberry Pi** pour assurer une meilleure réactivité au plus près du robot.
- **Allègement du pipeline de vision :** Au démarrage des tests, l'étape de *pick-and-place* était totalement bloquée car l'énorme flux de données généré par la caméra saturait le système, rendant impossible la lecture des coordonnées calculées. Pour désengorger le système et stabiliser la détection, nous avons dû désactiver les flux vidéo infrarouge et de profondeur (depth) non indispensables à cette étape. Enfin, la fréquence d'acquisition de la caméra a été abaissée drastiquement de 30 Hz à 6 Hz, ce qui a permis de libérer la puissance de calcul nécessaire et de fiabiliser l'extraction des coordonnées.
- **Résolution du contrôle de la pince :** Le pilotage du préhenseur a nécessité l'abandon de la commande CMD SetOutput qui s'est avérée inefficace sur notre configuration matérielle. Après plusieurs sessions de débogage, l'analyse directe des signaux via le logiciel propriétaire **iRC** nous a permis d'identifier la nomenclature exacte des entrées, conduisant à l'adoption définitive et fonctionnelle de la commande CMD DOUT.
- **Renforcement de la sécurité (Safety Node et Workspace) :** Pour prévenir tout comportement dangereux, un nœud de sécurité (safety_node), non prévu initialement, a été implémenté. Il publie en continu sur le topic /safety/emergency_stop à une fréquence de 10 Hz. Afin de protéger l'infrastructure physique, nous avons également défini une zone d'exclusion (Workspace Scene) dans l'environnement de planification MoveIt. L'ensemble caméra et profilés aluminium y a été modélisé par des volumes simples (cubes) agissant comme des murs virtuels, interdisant à l'algorithme d'y faire passer une trajectoire.
- **Optimisation de la cinématique inverse (IK) :** Les calculs de trajectoire ont nécessité une révision complète des paramètres IK dans le fichier kinematics.yaml. Pour éviter

de surcharger le processeur avec des calculs de collision complexes, le modèle 3D détaillé (STL) de la pince a été remplacé par un simple cube virtuel aux dimensions équivalentes situé en bout de bras.

- **Compensation des erreurs de préhension :** Enfin, lors des essais réels, des imprécisions mécaniques entraînaient des saisies erronées de la roulette. Nous avons compensé ce défaut en intégrant des variables d'erreur dans notre algorithme, permettant d'ajuster numériquement la position finale du préhenseur par rapport aux coordonnées brutes renvoyées par le système de vision.

8. PERSPECTIVES ET TRAVAUX RESTANTS

8.1 Améliorations techniques à court terme

Avec une à deux semaines de développement supplémentaires, le système aurait pu atteindre un niveau d'intégration nettement plus abouti.

8.1.1 Intégration matérielle sur chariot autonome

L'objectif immédiat serait de rassembler l'ensemble du système — Raspberry Pi, switch réseau, contrôleur du robot et IHM — dans un chariot autonome clé en main, alimenté par une seule prise secteur. Tous les composants seraient intégrés dans des boîtiers propres, avec un câblage industriel, de sorte que la mise en service se résume à brancher le chariot et à lancer l'application. On passerait ainsi d'un démonstrateur de laboratoire à un poste de travail robotisé déployable.

8.1.2 IHM multi-modes

L'interface homme-machine pourrait proposer plusieurs modes de fonctionnement sélectionnables directement depuis l'écran. Nous avons d'ailleurs déjà testé plusieurs types de trajectoires de pick-and-place : un mode cartésien avec déplacement linéaire entre les points, un mode avec trajectoire à rayon de courbure imposé pour des mouvements plus fluides, ainsi que des séquences de démonstration de type « danse robotique » pour des usages pédagogiques ou événementiels. L'IHM permettrait de basculer entre ces modes en un clic, et d'en paramétrer les vitesses et les points de passage.

8.1.3 Extension du système de perception

L'ajout de caméras supplémentaires permettrait de couvrir un espace de travail plus large et de réduire les zones aveugles, améliorant ainsi la robustesse de la détection dans des configurations variées.

8.1.4 Mobilité du chariot

Une évolution particulièrement intéressante consisterait à motoriser les roues du chariot pour le rendre mobile de manière autonome. Le bras robotisé ne serait plus affecté à un poste fixe mais pourrait se déplacer d'une zone de travail à une autre : une fois le tri des roulettes terminé, le chariot se déplacerait par exemple vers un poste de peinture ou d'assemblage. Cette approche s'inscrit dans la tendance actuelle des industriels qui cherchent à construire des jumeaux numériques reconfigurables de leurs usines, où des robots mobiles autonomes se réaffectent dynamiquement en fonction des besoins de production.

8.1.5 Améliorations logicielles

Côté logiciel, deux chantiers prioritaires restent à mener. D'une part, la validation complète du modèle URDF/XACRO de la pince dans MoveIt2, avec intégration dans la planning scene et gestion des collisions, permettrait une planification de trajectoire fiable tenant compte de l'encombrement réel de l'effecteur. D'autre part, l'amélioration de la robustesse de la cinématique inverse par un mécanisme de fallback sur des poses alternatives résoudrait les problèmes de wrist flip évoqués précédemment : en cas de configuration singulière, le solveur testerait automatiquement des orientations de poignet alternatives pour trouver une solution articulaire viable.

8.2 Évolutions fonctionnelles envisagées

8.2.1 Évolution du système de perception et de tri

Pour la partie perception, une évolution fonctionnelle importante serait d'étendre le système à la détection de plusieurs types d'objets. Actuellement, le modèle YOLO11n a été entraîné pour détecter une seule classe, correspondant à la roulette. Une amélioration future consisterait donc à enrichir le dataset avec plusieurs catégories d'objets et à réentraîner le modèle afin de permettre un tri automatique par classe.

Cette évolution permettrait au système de distinguer différents objets présents dans la zone de travail et d'adapter la stratégie de préhension selon le type d'objet détecté. Par exemple, le robot pourrait choisir une trajectoire ou une orientation de prise différente selon que l'objet détecté est une roulette en position frontale, une roulette verticale ou un autre composant.

8.2.2 Intégration d'un système de Bin-Picking (objets en vrac)

Actuellement, les roulettes doivent être posées bien à plat pour être saisies. Une évolution majeure serait de permettre au robot de les attraper directement en vrac dans un bac. Grâce à un système de vision adapté, le robot pourrait repérer la roulette située tout au-dessus du tas, éviter les collisions avec les bords du bac, et vider le contenant de manière autonome.

8.2.3. Ajout d'un capteur d'effort (retour de force)

La pince Schunk fonctionne aujourd'hui en tout ou rien (ouverte ou fermée) sans retour d'information. L'installation d'un capteur d'effort sur la bride du robot apporterait une sensibilité physique au système. Ce retour d'effort permettrait de vérifier en temps réel si l'objet est bien serré, de détecter si une pièce glisse ou est manquante, et de stopper le robot en cas de résistance anormale pour protéger le matériel.

8.2.4 Évolution de l'interface et accès à distance

Actuellement, l'interface graphique s'exécute localement sur le Raspberry Pi. Une évolution consisterait à alimenter ce dernier par batterie et à utiliser sa connectivité Wi-Fi pour s'affranchir des liaisons filaires.

En parallèle, l'interface pourrait être adaptée pour être accessible directement depuis un navigateur web. Cela permettrait de déporter la supervision et le contrôle du robot sur d'autres terminaux connectés au réseau (comme un PC ou une tablette) et d'y afficher le flux vidéo en direct.

8.3 Généralisation et réutilisabilité

8.3.1 Package ROS2 de contrôle du bras igus ReBeL

La bibliothèque igus_rebel_control.py que nous avons développée est suffisamment découplée du reste du projet pour être publiée en tant que package ROS2 indépendant. Elle encapsule l'ensemble de la communication CRI avec le contrôleur igus — heartbeat ALIVEJOG, commandes de déplacement articulaire et cartésien, contrôle de la pince SCHUNK — dans une interface Python propre et documentée. Tout utilisateur disposant d'un bras igus ReBeL pourrait l'intégrer dans son propre workspace ROS2 sans avoir à reproduire les semaines de reverse engineering que nous avons investies sur le protocole.

8.3.2 Pipeline de perception transférable

Le pipeline de perception développé dans ce projet repose sur des composants modulaires basés sur ROS 2, YOLO11n et la caméra Intel RealSense D435. La détection d'objets, l'estimation de profondeur et la transformation des coordonnées dans l'espace 3D sont réalisées de manière totalement indépendante du bras robotique utilisé. Cette séparation nette entre perception et actionnement rend le système directement réutilisable sur d'autres plateformes robotiques.

Concrètement, pour porter le pipeline sur un autre bras — qu'il soit industriel ou collaboratif — il suffit de recalibrer la caméra dans sa nouvelle position et de redéfinir la transformation

statique entre le repère caméra et le repère de base du robot. Le reste de la chaîne — acquisition RGB-D, inférence YOLO, calcul du centroïde 3D et publication des coordonnées cibles — fonctionne sans modification.

Cette modularité constitue un atout majeur pour des applications de vision industrielle : on peut transférer le système vers un autre environnement de production, changer de bras robotique ou même ajouter de nouveaux types d'objets à détecter par simple réentraînement du modèle YOLO, sans remettre en cause l'architecture logicielle globale.

9. CONCLUSION

Ce projet avait pour objectif de développer un système de pick-and-place autonome en intégrant un bras collaboratif igus ReBeL, une caméra Intel RealSense D435 et une architecture logicielle complète sous ROS 2. Notre cahier des charges nous demandait de boucler la chaîne complète : détecter un objet, estimer sa position dans l'espace, planifier le mouvement du robot, saisir l'objet avec la pince SCHUNK et le déposer dans une zone cible.

Nous avons atteint les objectifs principaux. Notre système fonctionne en mode réactif et relie effectivement la perception à l'action : la caméra détecte l'objet, on calcule sa position 3D puis on la transforme dans le repère du robot, et cette information commande le déplacement du bras et l'actionnement de la pince. Le cycle est répétable, et l'architecture globale que nous avons mise en place est fonctionnelle et cohérente.

Au-delà du résultat technique, ce projet nous a fait progresser dans des domaines très variés. Sur le plan mécanique, nous avons appréhendé les contraintes cinématiques d'un bras 6 axes, les problèmes de configurations singulières et de wrist flip, ainsi que les limites de l'espace de travail atteignable. Sur le plan électronique et protocoles industriels, nous avons plongé dans le protocole CRI pour piloter le contrôleur igus et la pince, avec tout ce que cela a impliqué de reverse engineering, de débogage de trames et de gestion du heartbeat. En intelligence artificielle et vision par ordinateur, nous avons entraîné et déployé un modèle YOLO11n sur plateforme embarquée, exploité les flux RGB-D de la RealSense et implémenté les transformations géométriques entre repères caméra et robot. Enfin, ROS 2 a

été notre fil conducteur tout au long du projet : nous en avons assimilé le fonctionnement en profondeur — architecture en nœuds, communication par topics et services, fichiers de lancement, gestion des transformations TF, conteneurisation Docker — et nous avons pu constater à quel point cet écosystème structure et facilite le développement robotique.

Ce travail nous a montré qu'on peut réaliser un système de pick-and-place autonome à faible coût en s'appuyant sur un cobot collaboratif et des outils open-source. La solution que nous avons obtenue est modulaire, évolutive et transférable à d'autres plateformes robotiques ou d'autres environnements de production.

En perspective, nous pensons que ce système pourrait évoluer vers un poste de travail mobile et reconfigurable — chariot autonome intégré, IHM multi-modes, mobilité autonome — capable de se déplacer d'un poste à l'autre dans une logique de jumeau numérique reconfigurable, répondant ainsi à la demande croissante des industriels pour une automatisation flexible et adaptable.

BIBLIOGRAPHIE

[1] G. C. Devol, "Programmed Article Transfer," U.S. Patent 2 988 237, déposé en 1954, délivré en 1961.

[2] J. F. Engelberger, *Robotics in Practice: Management and Applications of Industrial Robots*. New York, NY, USA: AMACOM, 1980.

[3] M. Ceccarelli, "From the Unimate to the Delta robot: the early decades of Industrial Robotics," in *Explorations in the History of Machines and Mechanisms*, Springer, 2019, pp. 284–295.

[4] B. Siciliano, L. Sciavicco, L. Villani, and G. Oriolo, *Robotics: Modelling, Planning and Control*. London, UK: Springer-Verlag, 2009.

[5] J. E. Colgate, M. A. Peshkin, and W. Wannasuphoprasit, "Cobots: Robots for Collaboration with Human Operators," in *Proc. ASME Int. Mech. Eng. Congress and Exposition*, Atlanta, GA, USA, 1996, vol. DSC-58, pp. 433–439.

[6] R. Bischoff, J. Kurth, G. Schreiber et al., "The KUKA-DLR Lightweight Robot arm – a new reference platform for robotics research and manufacturing," in *Proc. 41st Int. Symp. on Robotics (ISR) and 6th German Conf. on Robotics (ROBOTIK)*, Munich, Germany, 2010, pp. 1–8.

[7] igus GmbH, "ReBeL® Cobot — Technical Datasheet," Cologne, Germany, 2024. [En ligne]. Disponible : https://www.igus.com/automation/rebel-cobot

[8] International Federation of Robotics, "World Robotics 2024 — Industrial Robots Report," Frankfurt, Germany, 2024.

[9] AIRLab-POLIMI, "ros2-igus-rebel: ROS 2 driver and MoveIt 2 integration for the igus ReBeL 6DOF," GitHub Repository, Politecnico di Milano, 2023. [En ligne]. Disponible : https://github.com/AIRLab-POLIMI/ros2-igus-rebel

[10] R. Girshick, J. Donahue, T. Darrell, and J. Malik, "Rich feature hierarchies for accurate object detection and semantic segmentation," in *Proc. IEEE Conf. Computer Vision and Pattern Recognition (CVPR)*, Columbus, OH, USA, 2014, pp. 580–587.

[11] R. Girshick, "Fast R-CNN," in *Proc. IEEE Int. Conf. Computer Vision (ICCV)*, Santiago, Chile, 2015, pp. 1440–1448.

[12] S. Ren, K. He, R. Girshick, and J. Sun, "Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks," in *Advances in Neural Information Processing Systems (NeurIPS)*, vol. 28, 2015, pp. 91–99.

[13] W. Liu, D. Anguelov, D. Erhan *et al.*, "SSD: Single Shot MultiBox Detector," in *Proc. European Conf. Computer Vision (ECCV)*, Amsterdam, Netherlands, 2016, pp. 21–37.

[14] J. Redmon, S. Divvala, R. Girshick, and A. Farhadi, "You Only Look Once: Unified, Real-Time Object Detection," in *Proc. IEEE Conf. Computer Vision and Pattern Recognition (CVPR)*, Las Vegas, NV, USA, 2016, pp. 779–788.

[15] A. Bochkovskiy, C.-Y. Wang, and H.-Y. M. Liao, "YOLOv4: Optimal Speed and Accuracy of Object Detection," *arXiv preprint arXiv:2004.10934*, 2020.

[16] G. Jocher and J. Qiu, "Ultralytics YOLO11," Ultralytics, version 8.3.0, Sept. 2024. [En ligne]. Disponible : https://docs.ultralytics.com/models/yolo11/

[17] R. Khanam and M. Hussain, "YOLOv11: An Overview of the Key Architectural Enhancements," *arXiv preprint arXiv:2410.17725*, 2024.

[18] S. Zollhöfer, P. Stotko, A. Görlitz *et al.*, "State of the Art on 3D Reconstruction with RGB-D Cameras," *Computer Graphics Forum (Eurographics State-of-the-Art Reports)*, vol. 37, no. 2, pp. 625–652, 2018.

[19] Intel Corporation, "Intel® RealSense™ Product Family D400 Series — Datasheet," Doc. 337029, Rev. 016, 2023. [En ligne]. Disponible : https://www.intelrealsense.com/

[20] M. Quigley *et al.*, "ROS: an open-source Robot Operating System," in *Proc. ICRA Workshop on Open Source Software*, Kobe, Japan, 2009.

[21] Object Management Group, *Data Distribution Service (DDS) Specification*, Version 1.4, OMG Document formal/2015-04-10, 2015. [En ligne]. Disponible : https://www.omg.org/spec/DDS/

[22] Open Robotics, "Quality of Service settings — ROS 2 Humble Documentation," 2022. [En ligne]. Disponible : https://docs.ros.org/en/humble/Concepts/Intermediate/About-Quality-of-Service-Settings.html

[23] D. Casini, T. Blass, I. Lütkebohle, and B. B. Brandenburg, "Response-Time Analysis of ROS 2 Processing Chains under Reservation-Based Scheduling," in *Proc. 31st Euromicro Conf. on Real-Time Systems (ECRTS)*, Stuttgart, Germany, 2019, pp. 1–23.

[24] D. Coleman, I. Sucan, S. Chitta, and N. Correll, "Reducing the Barrier to Entry of Complex Robotic Software: a MoveIt! Case Study," *Journal of Software Engineering for Robotics*, vol. 5, no. 1, pp. 3–16, 2014.

[25] I. A. Şucan, M. Moll, and L. E. Kavraki, "The Open Motion Planning Library," *IEEE Robotics & Automation Magazine*, vol. 19, no. 4, pp. 72–82, Dec. 2012.

[26] J. J. Kuffner and S. M. LaValle, "RRT-Connect: An Efficient Approach to Single-Query Path Planning," in *Proc. IEEE Int. Conf. Robotics and Automation (ICRA)*, San Francisco, CA, USA, 2000, pp. 995–1001.

[27] R. Smits, *KDL: Kinematics and Dynamics Library*, Orocos Project, 2011. [En ligne]. Disponible : http://www.orocos.org/kdl

[28] P. Beeson and B. Ames, "TRAC-IK: An Open-Source Library for Improved Solving of Generic Inverse Kinematics," in *Proc. IEEE-RAS Int. Conf. on Humanoid Robots (Humanoids)*, Seoul, South Korea, 2015, pp. 928–935.

[29] Universal Robots A/S, *The URScript Programming Language*, Manual version 5.x, Odense, Denmark, 2023.

[30] Universal Robots A/S, *Real-Time Data Exchange (RTDE) Guide*, Doc. URDoc-RTDE, 2023. [En ligne]. Disponible : https://www.universal-robots.com/

[31] KUKA Roboter GmbH, *KUKA.RobotSensorInterface 4.0 — Documentation*, Augsburg, Germany, 2021.

ANNEXES

L'ensemble du code source, des fichiers de configuration, des modèles URDF/XACRO, des launch files ainsi que les captures supplémentaires sont disponibles sur notre dépôt GitHub à l'adresse suivante :

https://github.com/danielsb67/Projet_M1_MESI_IGUS_Rebel.git

On y retrouve notamment la bibliothèque igus_rebel_control.py, le nœud pick_and_place_node.py, les fichiers de configuration cinématique, le modèle XACRO de la pince SCHUNK et le launch file principal.

Les fiches techniques des composants utilisés (igus ReBeL 6DOF, pince SCHUNK EGP 25-N-N-B, caméra Intel RealSense D435) ainsi que le diagramme de Gantt complet du projet sont joints en fin de document.