REPUBLIQUE ALGERIENNE DEMOCRATIQUE ET POPULAIRE
MINISTERE DE L'ENSEIGNEMENT SUPERIEUR ET DE LA RECHERCHE SCIENTIFIQUE

Université de Mostaganem Abdelhamid Ibn Badis
Faculté des Sciences et de la Technologie
Département de Génie Électrique

Mémoire de Master 2

Filière : Automatique
Option : Automatique et Informatique Industrielle

Présenté par :
BOUCHTARA Rachid et BOUZIANE Mohamed

Thème

Simulation et contrôle par apprentissage par renforcement d'un bras manipulateur industriel Thor à 6 DoF dans Gazebo avec ROS2

Devant le jury composé de :

Président : Dr. M. BENTOUMI

Examinateur : M^elle^ D. BENDANI

Encadreur : Dr. M. L. AARIZOU

Année universitaire 2024/2025


بسم الله الرحمن الرحيم

Dédicaces

Bouzzian

À mes très chers parents, pour tous leurs sacrifices et leur soutien tout au long de mes études. Que Dieu les protège !

À ma sœur Imane, mon frère Aboubaker, et à toute ma famille

À mes amis et ma deuxième famille : Yasser, Abderrahim, Rachid, Bilal, Mohcine, Ilyas, Dhiae Eddine, Abd El-Mouiz, Tayebe, El-qalaa et Salah Eddine,

À toute personne qui travaille pour la renaissance de la nation des musulmans et à tous ceux que j'aime.

Bouchetara

À ma chère maman, pilier inébranlables de ma vie, dont les sacrifices et l'amour inconditionnel ont pavé le chemin de mes réussites. Que ca bienveillance soit éternellement récompensée.

À mon cher père, Que Dieu ait pitié de lui et lui accorde une place au paradis , pour ces laissons dans la vie et le guidance dans mon chemin de pour succès

À mes frères et sœurs, mansour, nourhan, meriem, benouda. pour leur présence précieuse et leur soutien indéfectible, qui ont fait de chaque étape un partage.

À mes professeurs et mentors, spécialement DR M L Aarizou, dont la sagesse et la passion ont éclairé mon esprit et guidé mes pas sur la voie du savoir. Leur dévouement restera une source d'inspiration.

À mes amis, ma deuxième famille, Mohamed, ilyas, abdel-basset, chawki. compagnons de route des moments joyeux et des défis, pour leur amitié sincère et leur complicence inestimable.

À tous ceux qui, par leur bienveillance et leur soutien, ont contribué de près ou de loin à mon parcours. Que la paix et la prospérité les accompagnent toujours

Remerciements

Nous remercions Allah, Le Tout Puissant et Le Miséricordieux, Qui nous a donné courage, volonté, santé et patience ainsi que les moyens de mener à terme ce modeste travail.

Nous souhaitons également exprimer notre profonde gratitude à tous les professeurs qui ont contribué à notre formation, ainsi qu'aux membres du laboratoire de recherche LSS pour leur soutien et les ressources précieuses qu’ils nous ont procurées.

Résumé

Ce mémoire de fin d'études explore la simulation et le contrôle d'un bras manipulateur industriel à six degrés de liberté (6-DoF), le robot Thor, en utilisant le Reinforcement Learning (RL) et la plateforme Robot Operating System 2 (ROS2). Face aux défis de la conception et du contrôle des manipulateurs complexes, le projet vise à démontrer l'efficacité de ROS2 pour la gestion complète d'un système robotique, de la modélisation virtuelle à la commande en temps réel, en intégrant des méthodes d'apprentissage basées sur le RL.

Ce mémoire est structuré en trois chapitres :

- Le premier chapitre présente une vue d'ensemble de ROS2 ;
- Le deuxième chapitre aborde les méthodologies du Reinforcement Learning appliquées aux manipulateurs, et ;
- Le troisième chapitre détaille les résultats de la simulation et du contrôle du robot Thor, notamment pour des tâches de manipulation d'objets comme "*pick and place*", en utilisant l'algorithme *Deep Deterministic Policy Gradient* (DDPG) dans un environnement simulé dans *Gazebo*.

Table des matières

Table of Contents

Table des Figures

Liste des tableaux

Introduction générale

La robotique industrielle a révolutionné les processus de fabrication et d'automatisation, offrant des solutions pour des tâches répétitives, dangereuses ou nécessitant une grande précision. Au cœur de cette révolution se trouvent les manipulateurs industriels, des systèmes mécaniques complexes conçus pour interagir avec leur environnement et effectuer une multitude

d'opérations. Parmi eux, les manipulateurs à six degrés de liberté (6-DoF) sont particulièrement polyvalents, capables d'atteindre n'importe quelle position et orientation dans leur espace de travail, ce qui les rend indispensables dans des applications allant de l'assemblage à la soudure, en passant par la peinture et la manipulation de matériaux.

Cependant, la conception, la simulation et le contrôle de ces systèmes posent des défis significatifs. La complexité cinématique et dynamique des manipulateurs 6-DoF exige des méthodes de modélisation précises et des algorithmes de contrôle robustes pour garantir des mouvements fluides, précis et sûrs. Avant tout déploiement physique, la simulation joue un rôle crucial en permettant de tester et d'optimiser les stratégies de contrôle dans un environnement virtuel sûr et rentable. Cela inclut la validation des trajectoires, la détection des collisions et l'évaluation des performances du système dans diverses conditions.

Dans ce contexte, le Framework Robot Operating System 2 (ROS2) émerge comme une plateforme logicielle de choix pour le développement de systèmes robotiques avancés. Successeur de ROS, ROS2 apporte des améliorations substantielles en termes de performance, de sécurité, de fiabilité et de support multiplateforme, le rendant particulièrement adapté aux applications industrielles critiques. Sa modularité fait de lui un environnement idéal pour aborder les défis liés à la simulation et au contrôle des manipulateurs industriels.

De plus, l'apprentissage par renforcement (RL) est devenu une approche prometteuse pour l'entraînement des robots manipulateurs. En permettant aux systèmes d'apprendre des comportements complexes par essais et erreurs dans un environnement simulé, le RL offre la possibilité d'acquérir des compétences de manipulation fines, adaptatives, et difficiles à programmer manuellement. L'intégration du RL avec ROS2 et les environnements de simulation permet de développer des agents robotiques capables de s'adapter à des situations imprévues et d'optimiser leurs performances au fil du temps.

Ce mémoire de fin d'études explore la simulation et le contrôle par l'entraînement d'un manipulateur industriel à 6 degrés de liberté, intégré dans un environnement industriel et opéré via ROS2, en mettant particulièrement l'accent sur l'application de l'apprentissage par renforcement. L'objectif principal est de démontrer la faisabilité et l'efficacité de l'utilisation de ROS2 pour la gestion complète d'un tel système robotique, depuis sa modélisation virtuelle jusqu'à sa commande en temps réel, en passant par des méthodes d'apprentissage et de formation basées sur le RL.

Le premier chapitre de ce mémoire sera consacré à une présentation approfondie de ROS2, en détaillant son architecture, ses concepts fondamentaux et les outils pertinents pour la robotique

industrielle. Le deuxième chapitre abordera les méthodologies d'apprentissage par renforcement appliquées à la robotique, en explorant comment ces techniques peuvent améliorer l'autonomie et l'adaptabilité du système. Enfin, le troisième chapitre présentera le travail effectué ainsi que les résultats obtenus de la simulation et du contrôle du manipulateur.

Chapitre 1 : Vue d'ensemble sur ROS2

1. Introduction

Ce premier chapitre a pour objectif de présenter les bases nécessaires pour comprendre le système Robot Operating System 2 (ROS2) , qui est le cœur de notre projet. Nous allons donc, dans ce chapitre, découvrir les concepts clés de ROS2, tels que les nœuds, les sujets (topics) , les services , et leurs interactions grâce à l'architecture DDS (Data Distribution Service). Enfin, nous présenterons quelques outils essentiels que nous avons utilisés dans ce projet, comme RViz2 pour la visualisation, et MoveIt2 pour la planification de mouvement.

2. Évolution historique de ROS et l'émergence de ROS2

ROS2 [1] est une plateforme open-source qui regroupe un ensemble de bibliothèques logicielles et d'outils conçus pour faciliter le développement d'applications robotiques. Bien qu'il ne soit pas un système d'exploitation au sens traditionnel, il agit comme un framework ou un middleware qui gère la complexité des systèmes distribués en robotique, permettant la communication entre les différents composants d'un robot, tels que les capteurs, les actionneurs et les algorithmes de décision. En tant qu'évolution de ROS (première version du framework), il apporte des améliorations pour répondre aux nouveaux besoins de la robotique, notamment la collaboration entre robots, les contraintes temps réel et les environnements de production.

Le système ROS [2], le prédécesseur de ROS 2, a été initialement lancé en 2010, en tant que projet open-source conçu pour faciliter le développement des applications robotiques. Avec les années, il est devenu très populaire auprès des chercheurs et des développeurs grâce à sa flexibilité et au soutien actif de la communauté scientifique. Cependant, avec l'augmentation des besoins dans les domaines industriels et commerciaux, certaines limites de ROS1 sont apparues. Parmi ces limites, on peut citer l'absence de sécurité le manque d'une gestion centralisée des communications entre les nœuds, et un support limité pour le système Windows. Au lieu de résoudre ces limites dans l'architecture de ROS, les chercheurs et développeurs ont choisi d'implémenter une nouvelle architecture pour résoudre les problèmes rencontrés dans ROS et une nouvelle version appelée ROS2 a été développée à partir de 2014. L'objectif était de garder les fonctionnalités essentielles de ROS1, tout en apportant des améliorations majeures, notamment l'intégration de l'architecture DDS (Data Distribution Service) qui permet une communication plus efficace et fiable entre les différents composants du système

3. Architecture de ROS2

ROS2 repose sur une architecture modulaire et distribuée, où chaque fonctionnalité est implémentée et encapsulée dans une unité appelée Node [3] (nœud en français). Les nodes communiquent entre eux à l'aide de différents mécanismes tels que les Topics [4] (Sujets en français), les Services [5] et les Actions [6].

3.1. NODES

Dans ROS2, un node constitue une unité fondamentale de calcul. Cela permet d'organiser les applications en modules plus petits et plus faciles à gérer. Chaque node dans ROS2 est dédié à une action spécifique, comme le contrôle des moteurs d'un robot mobile ou la transmission des données d'un capteur. Chaque node a la capacité d'envoyer des données à un autre node ou d'en recevoir. Cette interaction s'effectue par le biais de topics, de services, d'actions ou de paramètres.

Un système robotique complet est donc constitué de plusieurs nodes qui collaborent. Un programme en C++ ou un script en Python dans ROS2 peut initier un ou plusieurs nodes dans l'environnement d'exécution ROS2. Il existe différentes formes de communication entre les nodes. Les topics et les services sont les plus usuels que nous expliquerons dans ce qui suit.

3.2. TOPICS

Les topics constituent des éléments essentiels pour la communication entre les nodes. Ils servent à transmettre des informations, telles que les données des capteurs ou les instructions de mouvement, sans qu'un node ait à connaître les spécificités des autres nodes. Dans ce type de communication, les nodes adoptent une approche appelée publisher/subscriber (diffuseur/souscripteur en français) qui définit la manière par laquelle sont utilisés les topics pour communiquer. Si un node veut transmettre une information à un autre node, il publie ou diffuse un message, qui a une forme et un type spécifiques, sur un topic, on appelle ce node un publisher. Le deuxième node à qui le message est destiné reste à l'écoute sur ce topic et reçoit chaque message qui y est publié. On appelle ce tfype de node un subscriber. De cette manière, un node peut envoyer à un ou plusieurs autres nodes des messages et peut en recevoir d'autres de la part d'autres nodes. La Figure 1 illustre plusieurs cas de communication entre différents nodes en utilisant les topics. Dans cet exemple, quatre nodes sont lancés dans l'environnement ROS2: Node2 publie des messages sur le topic TopicA, et Node1 ainsi que Node3 s'abonnent à ce topic pour recevoir ces messages. Pendant ce temps, Node3 et Node4 publient des messages sur le topic TopicB, et Node2 et Node3 s'abonnent à ce topic pour recevoir ces messages. Autrement dit, il est techniquement possible pour un node d'envoyer des messages à lui-même, si nécessaire.

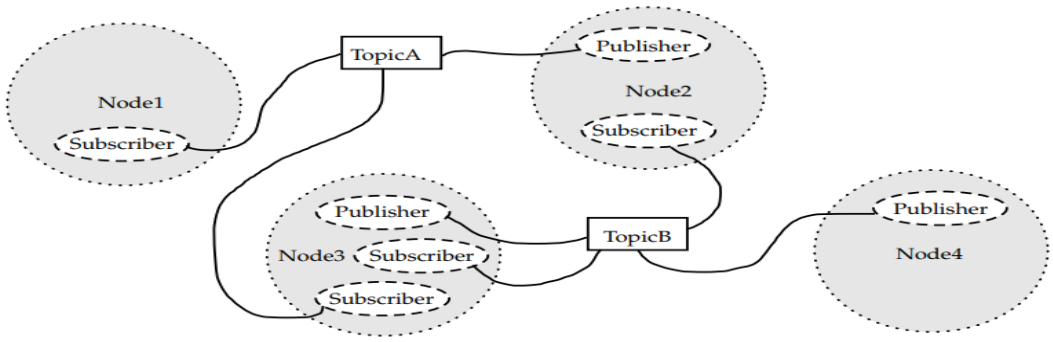

Figure 1. Exemple de communication de nodes via des topics[4].

3.3. Services

Les service [5] offrent une forme de communication légèrement plus élaborée entre les nodes. Ce type de communication suit l'architecture service-client, où un node implémentant unservice traite les requêtes d'un ou plusieurs nodes clients. Un node client envoie une requête au node service en appelant une fonction de ce service. Celui-ci envoie le résultat de la fonction en réponse à la requête du client tout de suite après. Par cette approche, plusieurs nodes clients peuvent utiliser le même service dans l'environnement ROS 2. La Figure 2 illustre le mode de communication service/client dans ROS2.

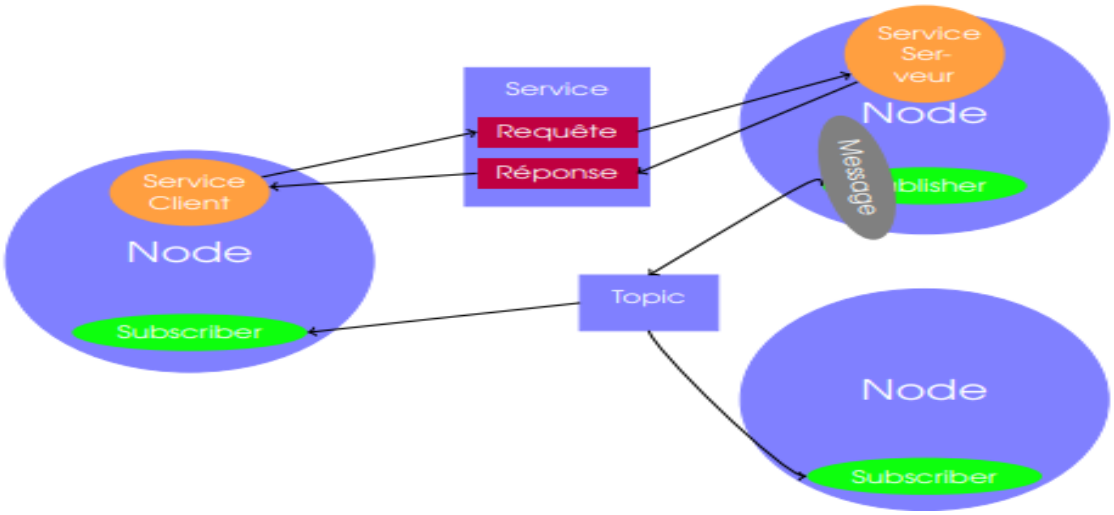

Figure 2. Échange de données correspondant au mécanisme de service et de topic[5].

3.4. Actions

Les actions [6] sont un moyen plus évolué pour communiquer entre les nodes: elles sont conceptuellement composées de services et de topics. Un Action Server (ou serveur d'action en français) fournit un service en deux phases pour les objets action client (clients d'action en français) avec un retour de résultats facultatif.

Dans la première étape, le client demande une tâche spécifique via le Goal Service et obtient une réponse quant à sa faisabilité. Si la réponse est affirmative, le client passe à la deuxième phase en demandant le résultat de cette tâche. Pendant l'exécution de la tâche, le serveur a la capacité d'envoyer des messages de retour au client via un topic (feedback topic). L'action est conclue quand le serveur envoie le résultat final au client. La Figure 3 illustre ce principe de communication en plusieurs phases.

3.5. Messages

Dans ROS2, les messages [7] sont des structures de données utilisées pour l'échange d'informations entre les nodes via les topics, services et actions. Un message est composé de

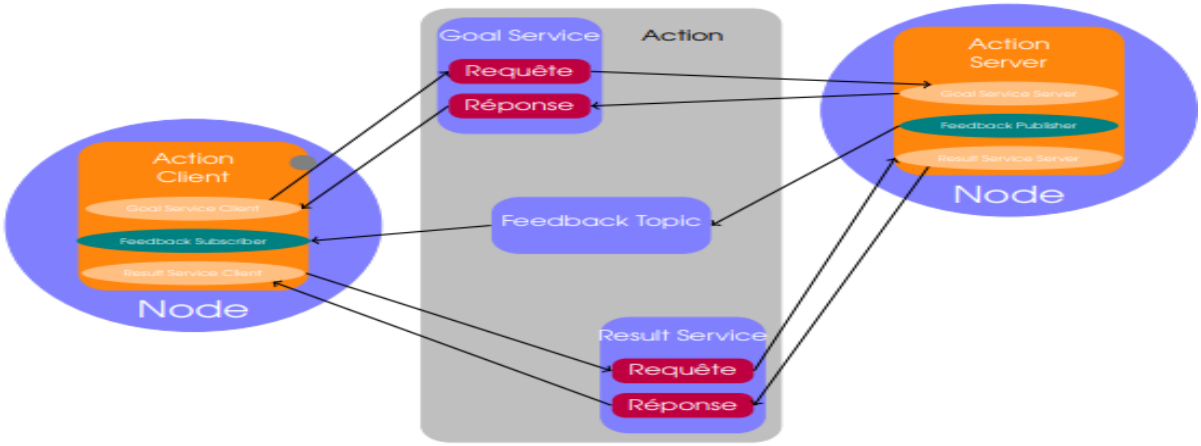

Figure 3. Échange de données correspondant au mécanisme d'action[6].

champs typés (int, float, string, etc.) et est défini dans un fichier situé dans le répertoire d'un projet ROS. Les utilisateurs peuvent créer leurs propres messages personnalisés, nécessaires pour représenter des données spécifiques à leur application robotiques. ROS2 fournit aussi des messages standards, comme *std_msgs/String*, *geometry_msgs/Twist*, ou *sensor_msgs/Image*.

3.6. Paramètres

Dans ROS 2, la gestion des paramètres [8] constitue un mécanisme fondamental pour le stockage des données de configuration. Les paramètres sont utilisés pour stocker une grande variété d'informations pertinentes telles que le modèle du robot, les trajectoires de mouvement, les gains des contrôleurs et toute autre donnée de configuration essentielle. Pour une gestion flexible, ces paramètres peuvent être chargés et sauvegardés de manière persistante grâce à des fichiers au format YAML, ou encore modifiés dynamiquement via la ligne de commande.

4. Système de fichier de ROS2

4.1. Espace de travail (Workspace)

Un workspace [9] ROS2 est un répertoire où sont créés, modifiés et compilés les projets ROS2. Il constitue l'environnement principal dans lequel sont développés des applications robotiques. La structure d'un workspace ROS 2 est illustrée dans la Figure 4.

Un espace de travail ROS2 peut inclure plusieurs packages, chacun étant situé dans un dossier distinct, et regroupés tous dans le dossier *src*. Ces packages peuvent être de divers types de construction, tels que CMake (pour les programmes C++) ou Python. Il est important de noter qu'il n'est pas permis d'avoir des packages imbriqués, c'est-à-dire un package à l'intérieur d'un autre.

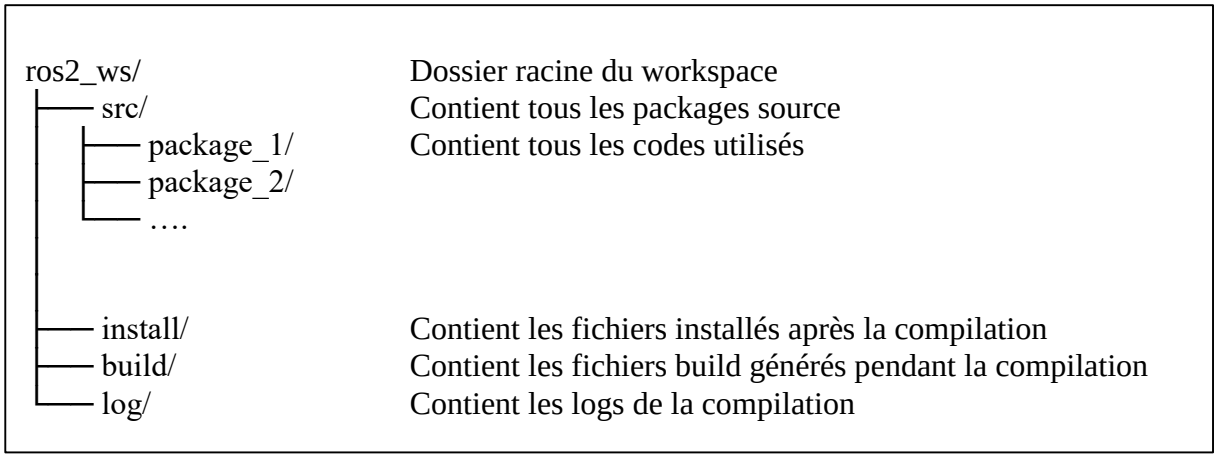

Figure 4. Forme général du workspace dans ROS2.

4.2. Package ROS2

Un package [10] représente le répertoire où est structuré le code ROS 2. Cette structure facilite l'installation et le partage du code source, ce qui permet une large diffusion des travaux réalisés avec ROS 2. Les packages contiennent plusieurs élément qui peuvent différer les uns des autres, mais on distingue deux dossiers qui sont généralement présents dans la plus part des package : le dossier des nodes (qui porte généralement le même nom que celui du package) et le dossier des lancements des exécutions appelé généralement : *launch*, regroupant des fichiers appelés : *fichiers launch.*

4.3. Les fichiers launch

Un fichier launch [11] est un fichier de lancement d'exécution dans ROS 2 qui facilite l'exécution de plusieurs nodes et configurations dans des applications robotiques. Il permet aux développeurs de définir comment leurs applications doivent fonctionner en spécifiant des paramètres, définissant des topics et en configurant l'environnement. Les fichiers launch facilitent la gestion de systèmes robotiques complexes en encapsulant les configurations nécessaires dans un seul fichier exécutable. Cette fonctionnalité améliore l'efficacité et l'organisation du développement d'applications robotiques.

5. Environnement de simulation dans ROS2

Un environnement de simulation est un outil essentiel dans le développement des systèmes robotiques. Il permet de tester et valider les comportements des robots dans un monde virtuel, avant leur déploiement dans le monde réel. Il permet également d'avoir une alternative au manque de matériel dans les systèmes robotiques complexes.

Dans ROS2, il existe plusieurs environnements de simulation. Dans ce travail, notre choix s'est porté sur l'outil Gazebo [12] , qui est l'un des environnements de simulation les plus populaires dans la communauté ROS. Gazebo offre une modélisation réaliste des phénomènes physiques, comme la gravité, les collisions et les interactions avec l'environnement.

À Gazebo s'ajoute deux autres outils forts de ROS2 : RVIZ2 [13] essentiel pour la visualisation 3D et temps-réel des données de simulation et du mouvement ; et MoveIt! , outil indispensable pour calcul temps-réel de la géométrique inverse et de la planification du mouvement de robot.

5.1. Présentation de GAZEBO

Gazebo est un simulateur robotique 3D open-source qui offre un environnement de développement et de test virtuel sophistiqué. Développé initialement en 2002 par l'Université de Californie du Sud, Gazebo est maintenant maintenu par Open Robotics et constitue l'un des outils de simulation les plus utilisés dans la communauté robotique mondiale. Comme illustré dans la Figure 5, son interface est divisée en cinq zones principales :

- **Le Panneau Latéral Gauche (Zone 1)** : Ce panneau permet la gestion des éléments de

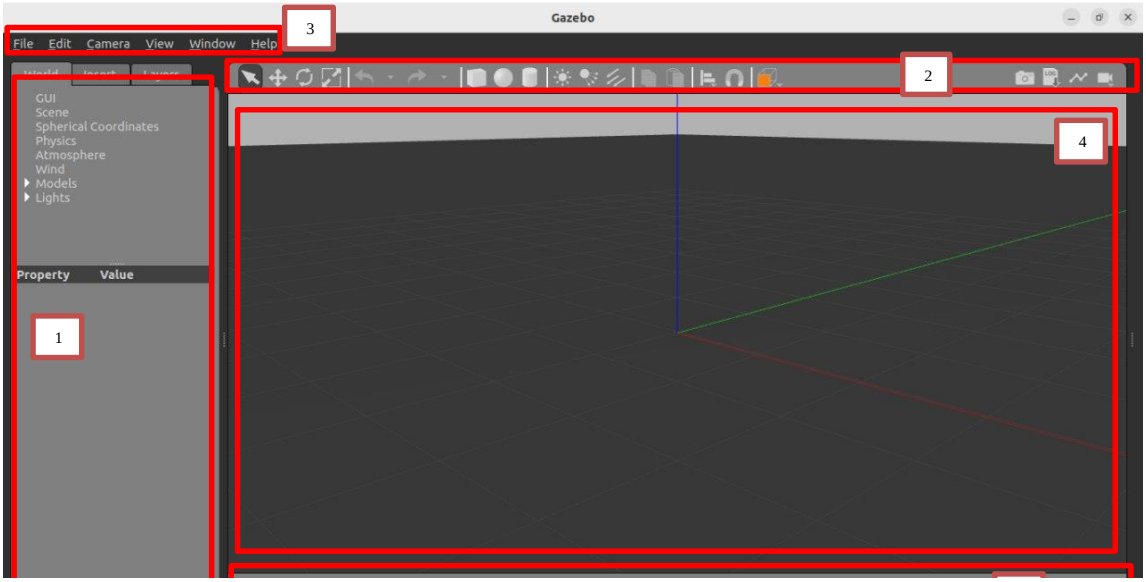

Figure 5. Fenêtre de l'interface de GAZEBO.

l'environnement simulé. Il est divisé en trois onglets principaux : l'onglet "World" (Monde) qui permet de visualiser et de modifier les propriétés globales de l'environnement et de ses entités, telles que les paramètres physiques, ou de l'atmosphère. L'onglet "Insert" (Insérer) qui permet d'ajouter facilement de nouveaux modèles (objets, robots) dans la scène à partir d'une liste catégorisée. Et enfin, l'onglet "Layers" (Couches) qui est dédié à l'organisation et à la gestion de la visibilité des groupes d'objets, simplifiant ainsi la visualisation de scènes complexes.

- **La Barre d'Outils Principale (Zone 2) :** Cette barre permet à l'utilisateur d'interagir avec la simulation. Elle comprend des outils de sélection et de manipulation permettant de déplacer, faire pivoter et redimensionner les objets dans la scène. Des icônes d'insertion rapide facilitent l'ajout de formes primitives et de sources de lumière. De plus, elle offre des contrôles de simulation pour démarrer, arrêter, avancer pas à pas ou réinitialiser l'environnement, ainsi que des options de visualisation pour gérer l'affichage d'éléments comme les ombres, les textures ou les grilles.

- **La Barre de Menus Supérieure (Zone 3) :** Elle est située en haut de la fenêtre et regroupe les fonctionnalités classiques que l'on retrouve dans la plupart des applications logicielles.
- **La Scène 3D Centrale (Zone 4) :** La Scène 3D Centrale constitue la zone la plus vaste et la plus cruciale de l'application. C'est dans cet espace que la simulation 3D prend vie, permettant aux utilisateurs de visualiser en temps-réel les robots, leurs environnements et toutes les interactions qui s'y déroulent. Cette zone offre des capacités de navigation complètes, où la souris permet de faire pivoter, de zoomer et de déplacer la vue de la caméra pour une exploration détaillée.
- **La Barre d'État Inférieure (Zone 5) :** Enfin, cette barre fournit les informations sur l'état actuel de la simulation et de l'application. Elle affiche généralement des messages d'état, des informations en temps-réel sur la performance de la simulation, le temps simulé par rapport au temps réel, et l'état de la connexion avec le moteur physique

5.2. Présentation de RViz

RViz est un outil de visualisation puissant et personnalisable au sein de l'écosystème ROS. Il permet aux développeurs de visualiser une multitude de données robotiques en temps réel, ce qui est indispensable pour le débogage, la surveillance et la compréhension du comportement d'un robot. Son interface est modulaire, permettant aux utilisateurs d'ajouter et de configurer différents types d'affichages pour répondre à leurs besoins spécifiques. Comme on peut le voir sur la Figure 6, on distingue cinq zones sur l'interface de RViz :

- **La barre du Menu et d'outils supérieure (zone 1) :** Contient des options de fichier, des panneaux d'aide, et des outils interactifs pour la navigation dans la scène 3D (comme 'Interact', 'Move Camera', 'Select', 'Focus Camera', 'Measure', '2D Pose Estimate', '2D Goal Pose', 'Publish Point').

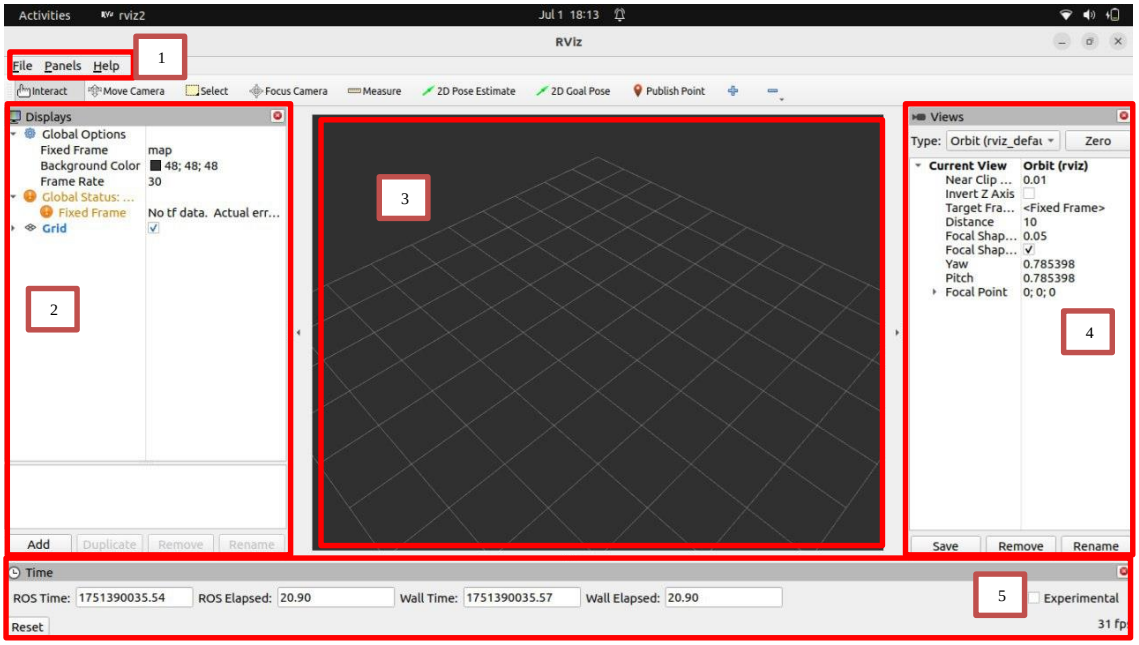

Figure 6. Fenêtre de l'interface de Rviz.

- **Panneau 'Displays' (zone 2) :** C'est le cœur de la configuration de RViz. Il permet d'ajouter, de dupliquer, de supprimer et de renommer des composants. Chaque composant est configuré pour visualiser un type spécifique de données (par exemple, un nuage de points, l'état d'un robot, une grille).
- **Scène 3D Centrale (zone 3) :** C'est la zone où toutes les données sont visualisées en 3D. L'utilisateur y observe le robot, son environnement, les trajectoires planifiées, les données de capteurs, etc.
- **Panneau 'Views' (zone 4) :** Permet de gérer les différentes vues de la caméra dans la scène 3D. Les utilisateurs peuvent sauvegarder, charger et basculer entre différentes configurations de caméra (par exemple, 'Orbit', 'FPS', 'ThirdPerson').
- **Barre d'État Inférieure (zone 5) :** Cette barre affiche, enfin, les informations en temps réel sur le temps ROS, le temps écoulé, et le taux de rafraîchissement de la visualisation.

5.3. Présentation de MoveIt!

MoveIt![14] est une plateforme logicielle open-source pour la planification de mouvement robotique, la manipulation et l'exécution de tâches. Il s'intègre parfaitement avec ROS et RViz, offrant une solution complète pour contrôler des bras robotiques, des manipulateurs mobiles et d'autres systèmes robotiques complexes. MoveIt! gère des aspects cruciaux tels que la cinématique inverse et directe, la détection de collisions, la planification de trajectoires sans collision, et l'intégration avec les contrôleurs matériels. L'interface de Moveit! est divisée en deux parties (voir Figure 7):

- **Panneau MotionPlanning (à gauche)** : Ce panneau intégré dans RViz permet de configurer et de contrôler le processus de planification de mouvement. Il est divisé en plusieurs onglets :

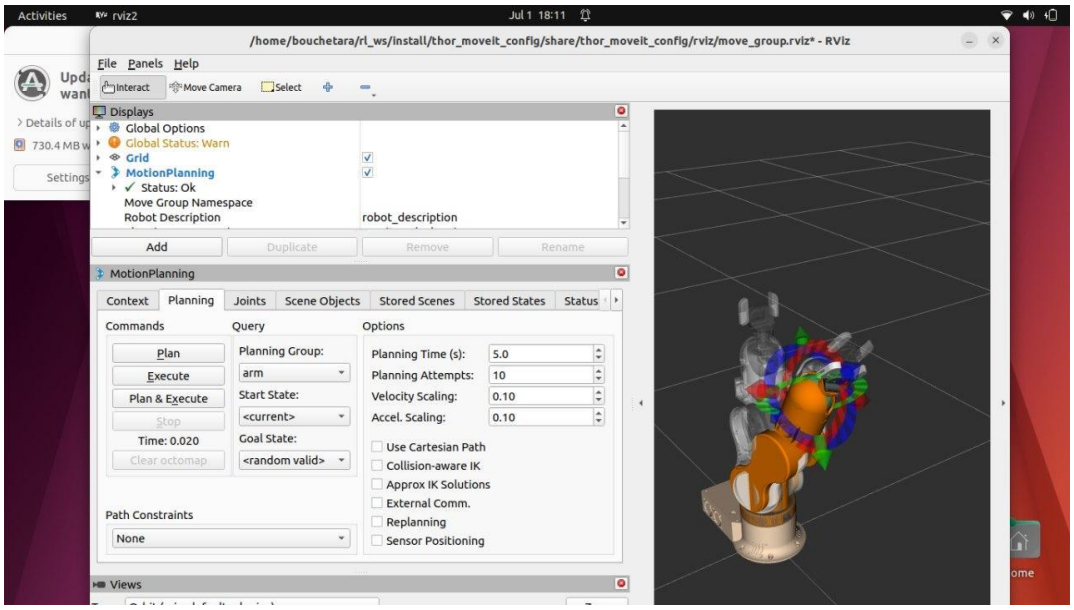

Figure 7. Fenêtre de interface de MoveIt!

- **Context** : Définit le groupe de planification (par exemple, arm pour un bras robotique), le namespace du groupe de mouvement, et la description du robot.
- **Planning** : Permet de définir les états de départ et d'arrivée, de spécifier les contraintes de chemin, et de lancer la planification et l'exécution du mouvement. On y trouve des paramètres importants comme le temps de planification (Planning Time), le nombre de tentatives de planification (Planning Attempts), et les facteurs d'échelle pour la vitesse et l'accélération.
- **Joints** : Permet d'afficher et de modifier les valeurs des articulations du robot.
- **Scene Objects** : Permet de gérer les objets dans la scène de planification, y compris les objets à manipuler et les obstacles.
- **Stored Scenes et Stred States** : Pour sauvegarder et charger des configurations de scène ou des états de robot prédéfinis.
- **Status** : Fournit des informations sur l'état actuel de la planification.

- **Visualisation du Robot et de la Planification (Scène 3D Centrale) :** Dans cette scène 3D, le robot est affiché avec ses différentes pièces et articulations. Lorsque MoveIt! planifie un mouvement, la trajectoire est visualisée en temps réel, permettant à l'utilisateur de voir le chemin que le robot empruntera. Les zones de collision potentielles ou les obstacles sont également affichés, garantissant que la trajectoire planifiée est sûre et sans collision. Sur la Figure 7, on peut voir le bras robotique Thor avec des indicateurs visuels pour son état actuel et les objectifs de planification. Plus de détails de ce robot seront présentés dans les sections à venir.

6. Le bras robotique Thor et implementation dans ROS2

Thor est un bras robotique open source et imprimable doté de six degrés de liberté. Censé pour faire plusieurs tache complexe dans ce espace de travail, il est composé de 6 articulation retative, pour le implémenté on a utilisé 3 package pour sa [15][:

6.1.thor_description

Ce package est responsable de la description du modèle 3D du robot Thor. Il contient les fichiers URDF (Unified Robot Description Format) ou XACRO (XML Macros for URDF) qui définissent la géométrie du robot, ses articulations, ses liens, ses capteurs et ses propriétés physiques. C'est la base pour visualiser le robot dans RViz et le simuler dans Gazebo.

Fichiers et répertoires clés (inférés):

- urdf/: Contient les fichiers .urdf ou .xacro décrivant le robot.
- meshes/: Contient les fichiers de maillage 3D (par exemple, .dae, .stl) utilisés dans la description URDF/XACRO.
- launch/: Peut contenir des fichiers de lancement pour afficher le modèle dans RViz.

6.2.thor_gazebo

Ce package fournit l'intégration du robot Thor avec le simulateur Gazebo. Il contient les fichiers de lancement et les configurations nécessaires pour faire apparaître le robot dans un environnement Gazebo, appliquer des propriétés physiques, et charger des plugins Gazebo spécifiques au robot (par exemple, pour la simulation des moteurs, des capteurs, ou l'interface avec ROS).

Fichiers et répertoires clés (inférés):

- launch/: Contient les fichiers de lancement pour démarrer Gazebo avec le robot Thor (simple.launch, thor_robot_in_world.launch). Ces fichiers incluent souvent le chargement du modèle URDF/XACRO et des plugins Gazebo.
- worlds/: contenir des fichiers de monde Gazebo (.world) qui définissent l'environnement de simulation (sol, obstacles, éclairage).
- config/: contenir des fichiers de configuration pour les plugins Gazebo.

6.3 thor_moveit_config

Ce package est pour la configuration MoveIt! et contient tous les fichiers de configuration nécessaires pour la planification de mouvement et la manipulation du robot Thor avec MoveIt!. Il permet à MoveIt! de comprendre la cinématique du robot, ses limites articulaires, ses groupes de planification, et de générer des trajectoires sans collision.

Fichiers et répertoires clés (inférés):

- config/: Contient la majorité des fichiers de configuration MoveIt!:
    - thor.srdf: Le fichier SRDF (Semantic Robot Description Format) qui décrit les groupes de planification, les états de pose, les paires de liens en collision, etc.

    - joint_limits.yaml: Définit les limites de vitesse et d'accélération pour chaque articulation.
    - kinematics.yaml: Spécifie le solveur cinématique à utiliser.
    - ompl_planning.yaml: Configuration pour le planificateur OMPL (Open Motion Planning Library).
    - ros_controllers.yaml: Configuration pour les contrôleurs ROS.
- launch/: Contient les fichiers de lancement pour démarrer le nœud move_group, RViz avec le plugin MoveIt!, et intégrer MoveIt! avec Gazebo (group_move.launch).

Relation entre les packages
Ces trois packages sont interdépendants pour une simulation et une planification de mouvement complètes du robot Thor dans ROS et Gazebo:

- **thor_description** est le fondement. Il fournit la description géométrique et cinématique du robot, qui est utilisée par les deux autres packages.
- **thor_gazebo** utilise la description du robot de thor_description pour faire apparaître le robot dans le simulateur Gazebo. Il ajoute les aspects physiques et les interactions avec l'environnement de simulation.
- **thor_moveit_config** s'appuie sur la description du robot de thor_description pour configurer MoveIt! pour la planification de mouvement. Pour la simulation, thor_moveit_config interagit souvent avec thor_gazebo via des contrôleurs ROS pour exécuter les trajectoires planifiées dans l'environnement simulé de Gazebo. Les fichiers de lancement dans thor_moveit_config peuvent démarrer Gazebo (via thor_gazebo) et MoveIt! ensemble.

7. Conclusion

Dans ce chapitre, on a parlé é sur tous l'élément utilisé pour notre simulation (environnement, logiciel …) on a cité tous les fondamentaux de ROS2 et on a parlé sur implémentation de notre robot dans le environnement

Chapitre 2: Apprentissage par Renforcement

1. Introduction

Le Reinforcement Learning (RL) est un domaine interdisciplinaire de l'apprentissage automatique et du contrôle optimal, qui s'intéresse à la manière dont un agent intelligent doit prendre des actions dans un environnement dynamique afin de maximiser un signal de récompense. De manière plus pratique, un agent apprenant n'est pas informé à l'avance des actions à effectuer ; il doit découvrir, par essais et erreurs, quelles actions rapportent le plus de récompenses. Dans les cas les plus intéressants et complexes, les actions influencent non seulement la récompense immédiate, mais aussi la situation suivante, et, à travers elle, toutes les récompenses futures. Ces deux caractéristiques, la recherche par essais-erreurs et la récompense différée, sont les deux traits les plus distinctifs de l'apprentissage par renforcement. Dans ce chapitre, nous nous intéressons à définir le principe du Reinforcement Learning et de ses notions fondamentales. Nous parlerons de ses composants et de ses applications dans la robotique.

2. Principe du Reinforcement Learning

Le Reinforcement Learning [16] fonctionne de manière à développer un agent autonome pour qu'il puisse apprendre à prendre des décisions de manière optimale en interagissant avec un environnement inconnu. Ce principe repose sur un mécanisme d'essais-erreurs, où l'agent apprend par l'expérience, en recevant des récompenses ou des punitions en fonction des actions qu'il entreprend. L'objectif est d'optimiser (généralement appliquer une fonction de maximisation) la somme des récompenses que l'agent peut obtenir. Cette capacité d'auto-apprentissage par interaction rend le RL particulièrement adapté aux problèmes de contrôle séquentiel, comme ceux rencontrés en robotique.

Pour illustrer ce concept, prenons l'exemple d'un robot mobile qui doit décider s'il entre dans une nouvelle chambre pour collecter plus de déchets des corbeilles, ou va chercher la station électrique pour recharger ses batteries . Cette décision doit être prise en fonction du niveau et de l'état de charge de la batterie et de sa capacité à avoir trouvé la station de rechargement dans le passé.

Dans cet exemple, le robot, qu'on considère un agent décisionnel actif, interagit avec un environnement qui lui est inconnu et dans lequel il cherche à atteindre un objectif. Par les actions qu'il entreprend, il a la possibilité d'influencer l'état futur de l'environnement (sa position suivante ou le niveau de charge de sa batterie, par exemple), modifiant ainsi les actions et opportunités disponibles dans le futur. Faire les bons choix implique donc de prendre en compte les conséquences des actions dans le futur, ce qui peut nécessiter de la planification ou de la prédiction. Cela dit, ces prédictions ne peuvent pas être faites avec certitude ; l'agent doit donc surveiller fréquemment son environnement et réagir en conséquence de manière appropriée. Ceci mène l'agent à évaluer son progrès vers l'objectif à partir de ce qu'il perçoit directement de son environnement: le robot mobile sait quand ses batteries sont déchargées.

L'agent peut également utiliser son expérience pour améliorer ses performances au fil du temps. Le robot pourra donc rapidement retrouver la station de recharge en se basant sur son expérience passée déjà acquise.

3. Composantes du Reinforcement Learing

Un système de RL repose donc sur l'interaction cyclique entre un agent et un environnement: À chaque étape de temps, l'agent observe l'environnement, effectue une action, reçoit une récompense, et observe l'état suivant. Ces éléments constituent les composantes fondamentales du processus d'apprentissage. Ce rapport cyclique est appelé politique ou policy en anglais (voir Figure 8).

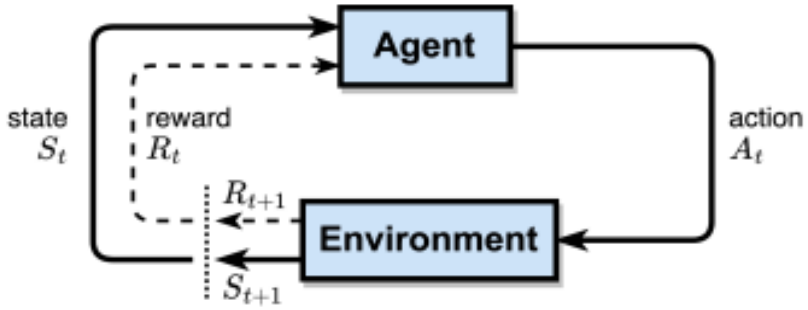

Figure 8. Politique d'un système RL.

Dans ce qui suit, nous définirons les différents éléments d'un système RL :

- **L'agent**: C'est le système qui apprend à interagir avec son environnement dans le but de maximiser les récompenses cumulées. L'agent doit choisir les actions à entreprendre à partir des informations "observées" de l'environnement. Dans le domaine de la robotique, l'agent est généralement un robot (réel ou simulé).
- **L'environnement**: C'est l'espace extérieur avec lequel l'agent interagit. Il fournit **les états** que perçoit l'agent, réagit à ses actions, et retourne des récompenses. En robotique, l'environnement inclut le monde physique ou simulé, les objets, les obstacles, les autres agents, etc.
- **Les états**: sont l'ensemble des situations courantes de l'environnement à un instant donné. L'agent perçoit les états d'un environnement pour pouvoir décider de la prochaine action. Plus l'état possède d'informations, meilleure sera la capacité de l'agent à prendre de *bonnes* décisions. Dans une application robotique, l'état peut être : la position et l'orientation d'un robot, les valeurs de capteurs, ou une image capturée par une caméra.
- **Les actions**: représentent les décisions que doit prendre l'agent à un instant donné. L'action affecté directement l'environnement et le fait évoluer vers un autre état. Deux types d'actions sont notés dans un système RL : actions discrètes qui sont des instructions directes comme tourner à gauche/à droite ou ouvrir/fermer la pince. En actions continues représentées généralement par des valeurs comprises dans des intervalles donnés : vitesse angulaire/linéaire, position du End Effector, etc.

- **La récompense:** Un signal que l'environnement renvoie à l'agent après chaque action, indiquant la qualité de cette action par rapport à l'objectif (par exemple, une récompense positive pour atteindre une cible, une récompense négative pour une collision).

4. Le Reinforcement Learning et le Processus de Décision de Markov

Le Reinforcement Learning de base est modélisé comme un Processus de Décision de Markov (MDP) [17], qui est un cadre mathématique pour la modélisation de la prise de décision dans des situations où les résultats sont partiellement aléatoires et partiellement sous le contrôle d'un décideur.

Un MDP est défini par un tuple (S, A, P, R, γ) où

- S: Un ensemble d'états possibles de l'environnement.
- A: Un ensemble d'actions possibles que l'agent peut entreprendre.
- $P(s' \mid s, a)$: La probabilité de transition vers l'état s' à partir de l'état s après avoir pris l'action a.
- $R(s, a, s')$: La récompense immédiate reçue après la transition de l'état s à l'état s' en prenant l'action a.
- γ: Le facteur de remise (discount factor), un scalaire entre 0 et 1, qui détermine l'importance des récompenses futures

Le RL implique une interaction continue entre un agent et son environnement. À chaque pas de temps t:

- L'agent observe l'état actuel de l'environnement s_t .
- L'agent choisit une action a_t.
- L'environnement reçoit l'action a_t, passe à un nouvel état s_{t+1} , et génère une récompense r_{t+1} ,
- L'agent reçoit la récompense r_{t+1} et le nouvel état s_{t+1}, et utilise cette information pour mettre à jour sa politique.

L'objectif d'un agent de RL est d'apprendre une policy $\pi(a \mid s)$, qui est une distribution des probabilités sur les actions étant donné un état, ou une fonction déterministe $\pi(s)$ qui mappe les états aux actions. Cette policy doit maximiser la récompense cumulative attendue sur le long terme :

$$G_t = \sum \ \gamma^k r_{t+k+1} \qquad (1)$$

Où r_{t+k+1} est la récompense reçu à l'instant $t + k + 1$.

4.1. Fonctions de Valeur

Les fonctions de valeur [16] sont essentielles en RL car elles estiment la "valeur" d'un état ou d'une paire état-action en termes de récompenses futures attendues. Il existe deux types principaux de fonctions de valeur

- Fonction de Valeur d'État ($V\,\pi(s)$): Représente la récompense cumulative attendue en partant de l'état s et en suivant la politique, par la suite :

$$V\,\pi(s) = E\pi[S_t = s] = \sum_s \ \pi(a|s) \sum_{s',r} p(s', r \mid s, a)(r + \gamma V\pi(s')) \quad (2)$$

- Fonction de Valeur Action-État ($Q\,\pi(s, a)$): Représente la récompense cumulative attendue en partant de l'état s, en prenant l'action a, et en suivant ensuite la politique π, par la suite :

$$Q\,\pi(s,a) = E\pi[S_t = s, A_t = a] = \sum_{s',r} p(s', r \mid s, a)\,[r + \gamma \sum_{a'} \pi(a'|s') q\pi(s', a')] \qquad (3)$$

4.2. Le Dilemme Exploration-Exploitation

Un défi central en RL est le compromis entre l'exploration et l'exploitation [16]. L'exploration consiste à essayer de nouvelles actions pour découvrir de meilleures récompenses, tandis que l'exploitation consiste à utiliser les actions connues qui ont donné de bonnes récompenses par le passé. Un agent doit équilibrer ces deux aspects pour maximiser sa récompense cumulative à long terme. Trop d'exploration peut entraîner un apprentissage lent, tandis que trop d'exploitation peut conduire à des politiques sous-optimales si l'agent ne découvre jamais de meilleures stratégies.

5. Les types du Reinforcement Learning

Les algorithmes de RL peuvent être largement classés en plusieurs catégories, chacune ayant ses propres forces et faiblesses. Nous parlerons dans cette partie de quatre de ces méthodes :

5.1. Apprentissage par renforcement basé sur la valeur (Value-based RL)

Les méthodes basées[18] sur la valeur se concentrent sur l'apprentissage d'une fonction de valeur (souvent la *Q value-function*) qui estime la récompense cumulative attendue pour chaque état ou paire état-action. Une fois la fonction de valeur apprise, la politique optimale (*optimal policy*) peut être dérivée en choisissant l'action qui maximise la valeur Q dans chaque état. L'une de ces méthodes d'apprentissage est *Q-learning*.

Le *Q-Learning* est un algorithme d'apprentissage par renforcement **sans modèle** et **hors-politique**. "**Sans modèle**" signifie qu'il n'a pas besoin d'un modèle de l'environnement (c'est-à-dire les probabilités de transition et les fonctions de récompense) pour apprendre ; et "**hors-politique**" signifie qu'il peut apprendre la politique optimale en explorant l'environnement de manière aléatoire ou en suivant une politique différente de celle qu'il apprend. L'algorithme *Q-Learning* met à jour la fonction Q en utilisant l'équation de Bellman:

$$Q(s,a) \leftarrow Q(s,a) + \alpha[r + \gamma a'\big(Q(s',a')\big) - Q(s,a)\,] \qquad (4)$$

Où:

- $Q(s,a)$: est la valeur actuelle de la paire état-action (s,a).
- α : est le taux d'apprentissage (learning rate), $0 < \alpha \leq 10$.
- r : est la récompense immédiate reçue après avoir exécuté l'action a depuis l'état s.
- γ : est le facteur de remise (discount factor), $0 \leq \gamma < 10$, qui détermine l'importance des récompenses futures.
- s' : est le nouvel état atteint après avoir effectué l'action a.
- $max\ Q(s',a')$: est la valeur Q maximale du nouvel état s' sur toutes les actions possibles a'.

5.2. Apprentissage par renforcement base sur la politique (Policy-based RL)

Les méthodes basées sur la politique cherchent à apprendre directement une politique optimale $\pi\theta(a \mid s)$. Ces méthodes sont particulièrement adaptées aux environnements avec des espaces d'action continus ou de très grande dimension, où l'apprentissage d'une fonction de valeur pour chaque action possible serait intenable.

L'objectif est de trouver les paramètres θ de la politique qui maximisent une fonction objective $J(\theta)$ représentant la performance de la politique. Le gradient de cette fonction objective, $\nabla\theta J(\theta)$, est utilisé pour mettre à jour les paramètres de la politique. Cette méthode d'optimisation est appelée « *Monte Carlo Policy Gradient* ».

La méthode *Monte Carlo Policy Gradient* est basée sur le principe d'utiliser des trajectoires complètes (épisodes) pour estimer le gradient de la politique. Pour chaque épisode, l'algorithme collecte toutes les récompenses et utilise la récompense cumulative future pour pondérer les gradients des log-probabilités des actions prises. La mise à jour des paramètres de la politique est effectuée dans la direction du gradient estimé:

$$\theta \leftarrow \theta + \alpha.\nabla\theta.J(\theta) \qquad (5)$$

Où le gradient est estimé par : $\nabla\theta J(\theta) \approx (1/M)\sum \qquad \nabla\theta\ log(\pi\theta(a_{i,t}|s_{i,t}))G_{i,t}$

5.3. Apprentissage par renforcement basé sur le model (Model-based RL)

Les méthodes basées sur le modèle [18] apprennent un modèle de l'environnement (c'est-à-dire les probabilités de transition et les fonctions de récompense) à partir des expériences collectées. Une fois le modèle appris, l'agent peut l'utiliser pour simuler l'environnement et planifier ses actions sans avoir à interagir physiquement avec le monde réel. Cela conduit à une efficacité d'échantillonnage significativement améliorée.

5.4. Apprentissage par renforcement basé sur l'acteur-critique (actor-critic-based RL)

Les méthodes acteur-critique combinent les avantages des méthodes basées sur la valeur et des méthodes basées sur la politique. Elles maintiennent deux composants distincts:

- **L'Acteur (Actor)**: Le composant basé sur la politique, responsable de la sélection des actions.
- **La Critique (Critic)**: Le composant basé sur la valeur, responsable de l'évaluation des actions prises par l'acteur et de la fourniture d'un signal d'erreur pour la mise à jour de l'acteur.

Il existe dans la littérature plusieurs méthodes d'apprentissage basées sur l'actor-critic. Dans ce projet, notre choix se porte sur la théorie du **Deep Deterministic Policy Gradient (DDPG)**.

6. Deep Deterministic Policy Gradient (DDPG)

Le Deep Deterministic Policy Gradient (DDPG) [19], [20] est un algorithme d'apprentissage par renforcement qui a été introduit en 2015 par Lillicrap et al. . Il s'agit d'une méthode acteur-critique hors-politique conçue pour gérer des espaces d'action continus. DDPG combine les idées de l'apprentissage Q-value avec les méthodes de policy gradient, ce qui lui permet d'apprendre des politiques déterministes dans des environnements complexes avec des actions continues.

Avant DDPG, les algorithmes de RL profonds comme QL avaient montré un succès remarquable dans des environnements avec des espaces d'action discrets. Cependant, ces méthodes ne pouvaient pas être directement appliquées aux problèmes avec des espaces d'action continus (par exemple, le contrôle robotique, la conduite autonome), où l'agent doit choisir une valeur numérique continue pour chaque action (par exemple, l'angle d'un joint de robot, la force d'un moteur) .

Les méthodes basées sur le policy gradient existantes pour les espaces d'action continus souffraient souvent d'une variance élevée dans leurs estimations de gradient, ce qui rendait l'apprentissage lent et instable. DDPG a été développé pour surmonter ces limitations en combinant la stabilité de l'apprentissage Q avec l'efficacité des policy gradient pour les actions continues.

DDPG est un algorithme acteur-critique, ce qui signifie qu'il utilise deux réseaux de neurones principaux: un acteur et un critique. De plus, pour améliorer la stabilité de l'apprentissage, il utilise des réseaux cibles et un *replay buffer* (voir Figure 9).

6.1. Le Réseau Actor

Le réseau actor, noté $\mu(s \mid \theta\mu)$, est responsable de la détermination de l'action à prendre dans un état donné. C'est un réseau de neurones qui prend l'état s comme entrée et produit une action déterministe a comme sortie. La politique est déterministe car elle associe directement un état à une action spécifique, contrairement aux politiques stochastiques qui produisent une distribution de probabilité sur les actions.

L'objectif du réseau actor est de trouver la politique qui maximise la fonction Q. La mise à jour des poids du réseau actor se fait en utilisant le gradient de la fonction Q par rapport aux actions, puis en multipliant ce gradient par le gradient de la politique par rapport à ses propres poids. Cela lui permet de se déplacer dans la direction qui augmente la valeur Q estimée par le Réseau critic.

La fonction de perte pour l'acteur est généralement définie comme la négation de la valeur Q moyenne des actions produites par l'acteur :

$$\boldsymbol{L_{actor}}(\Theta^{\boldsymbol{\mu}}) = -\boldsymbol{E_{s\sim D}}[\boldsymbol{Q}(\Theta^{\boldsymbol{Q}})] \qquad \textbf{(6)}$$

Où D est le replay buffer, et Θ^Q sont les poids du réseau critic.

6.2. Le Réseau Critic

Le réseau critic, noté $Q(s, a|\Theta^Q)$, est chargé d'évaluer la qualité des actions prises par le réseau actor. C'est un réseau de neurones qui prend à la fois l'état s et l'action a comme entrées, et produit une estimation de la valeur Q, c'est-à-dire la récompense cumulative attendue si l'agent prend l'action a dans l'état s et suit ensuite la politique optimale.

Le réseau critic est entraîné à minimiser l'erreur de Bellman, similairement au Q-Learning. La cible pour l'apprentissage du réseau critic est calculée en utilisant les réseaux targets du réseau actor et du réseau critic, ce qui contribue à la stabilité de l'apprentissage.

La fonction de perte pour le réseau critic est l'erreur quadratique moyenne (MSE) entre la valeur Q prédite par le réseau critic et la cible Q :

$$L_{critic}(\Theta^Q) = -E_{(s,a,r,s')\sim D}[Q(\ y)^2] \qquad \textbf{(7)}$$

Où y est la cible calculé par :

$$y = r + \gamma Q'(s', \mu'(s'|\theta^{\mu'})|\theta^{Q'}) \qquad \textbf{(8)}$$

Où Q' et μ' sont les réseaux targets des réseaux critic et actor, respectivement, et γ est le facteur de remise.

6.3. Les Réseaux Targets

DDPG utilise des copies "retardées" des réseaux actor et critic, appelées réseaux targets. Les poids des réseaux targets sont mis à jour lentement en effectuant une mise à jour douce (soft update) à chaque pas de temps, plutôt que de copier directement les poids des réseaux principaux. Cela empêche les oscillations et la divergence qui peuvent survenir si la cible de l'apprentissage change trop rapidement.

Les mises à jour des réseaux targets se fait par :

$$\theta^{Q'} \leftarrow \tau\theta^{Q'} + (1-\tau)\theta^{Q'};\ \theta^{\mu'} \leftarrow \tau\theta^{\mu'} + (1-\tau)\theta^{\mu'} \qquad \textbf{(9)}$$

6.4. Le Replay Buffer

Le *Replay Buffer* est utilisé pour stocker les transitions (s, a, r, s') que l'agent collecte en interagissant avec l'environnement. Pendant l'entraînement, des mini-lots de transitions sont échantillonnés aléatoirement à partir de ce buffer. Cela présente deux avantages majeurs :

- Réduction de la corrélation : L'échantillonnage aléatoire rompt la corrélation temporelle entre les expériences consécutives, ce qui est crucial pour l'entraînement des réseaux de neurones, car ils supposent généralement que les données sont indépendantes et identiquement distribuées.
- Réutilisation des Données : Les expériences passées peuvent être réutilisées plusieurs fois pour l'entraînement, ce qui améliore l'efficacité de l'échantillonnage et permet à l'algorithme d'apprendre de manière plus efficace à partir d'un nombre limité d'interactions avec l'environnement. C'est ce qui rend DDPG un algorithme hors-politique

6.5. Bruit d'Exploration (Exploration Noise)

Puisque la politique de l'acteur est déterministe, l'exploration est gérée en ajoutant du bruit aux actions produites par l'acteur pendant l'entraînement. Le bruit d'Ornstein-Uhlenbeck (OU) est couramment utilisé pour cette tâche. Le bruit OU est un processus stochastique qui génère des échantillons corrélés dans le temps, ce qui est plus adapté aux problèmes de contrôle continu où l'agent doit effectuer des mouvements fluides et cohérents.

L'action à exécuter dans l'environnement est donnée par :

$$a_t = \mu(s_t|\theta^\mu) + N_t \qquad \textbf{(10)}$$

Où N_t est le bruit généré par le processus OU. Ce bruit diminue généralement au fil du temps pour permettre à l'agent d'exploiter davantage sa politique apprise.

6.6. Applications de DDPG

DDPG est particulièrement bien adapté aux problèmes de contrôle continu dans des environnements complexes. Ses applications sont vastes et couvrent plusieurs domaines.La robotique est l'un des domaines où DDPG a démontré un impact significatif. Les tâches robotiques impliquent souvent des actions continues (par exemple, le couple appliqué à une articulation ou la vitesse d'un moteur) et des environnements dynamiques. DDPG permet aux

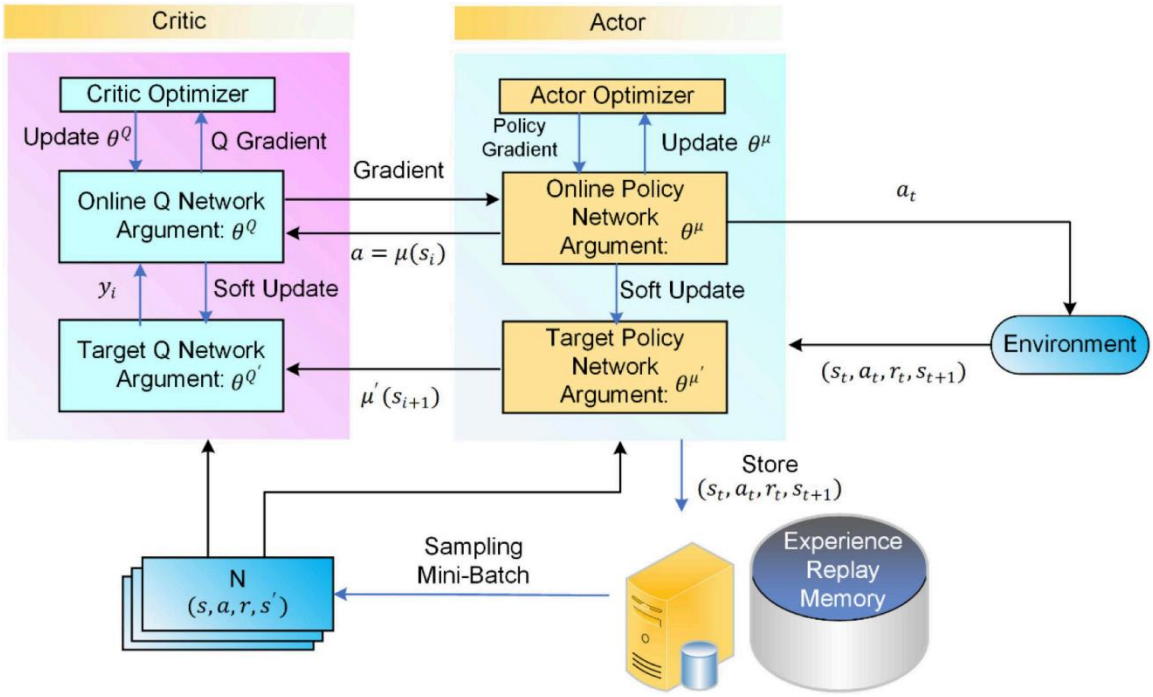

Figure 99. Exemple d'application de DDPG.

robots d'apprendre des politiques de contrôle complexes directement à partir de l'expérience. Un exemple d'utilisation de DDPG est illustré dans la Figure 9.

Ci-dessous, des exemples d'application de DDPG dans différent cas de figure dans le domaine de la robotique.

a. ***Manipulation d'Objets***

Les bras robotiques peuvent apprendre à saisir, manipuler et assembler des objets de différentes formes et tailles. DDPG peut être utilisé pour entraîner des politiques qui contrôlent les mouvements précis des effecteurs finaux du robot.

b. Locomotion

Les robots humanoïdes ou quadrupèdes peuvent apprendre à marcher, courir et maintenir l'équilibre sur des terrains variés. DDPG peut optimiser les signaux de contrôle envoyés aux moteurs des jambes pour une locomotion stable et efficace.

c. Navigation

Les robots mobiles peuvent apprendre à naviguer dans des environnements complexes, à éviter les obstacles et à atteindre des objectifs. DDPG peut être utilisé pour apprendre des politiques de direction et de vitesse.

7. Conclusion

Dans ce chapitre, on a parlé sur les élément fondamentale de reinforcement learning (state, action, reward …), on a cité tous les élément utilisé pour notre application (environnement et agent) ainsi notre algorithme de DDPG pour utiliser comme agent

Chapitre 3 : Simulation et Contrôle du Robot Thor par Apprentissage par Renforcement Profond

1. Introduction

Ce chapitre détaille l'implémentation pratique de la simulation et du contrôle du robot Thor à l'aide de l'apprentissage par renforcement profond, en se concentrant spécifiquement sur l'algorithme DDPG. Le projet vise à démontrer la capacité d'un agent DRL à apprendre des tâches complexes de manipulation d'objets, telles que "*pick and place*", dans un environnement simulé basé sur ROS2 et Gazebo. Nous explorerons la conception de l'environnement de simulation, la structure de l'agent DDPG, et l'intégration de ces composants au sein de l'écosystème ROS2, offrant ainsi une solution robuste pour le contrôle autonome de robots.

2. Scénario de la tâche

Le scénario choisi pour ce projet se concentre sur une tâche fondamentale de manipulation robotique : "*pick and place*" (prendre et placer). Le robot Thor est chargé de localiser un objet spécifique (une petite boîte en carton) dans son environnement simulé, de le saisir à l'aide de sa pince, puis de le déplacer et de le déposer à un emplacement prédéfini (un cube plat). Cette tâche, bien que simple en apparence, requiert une coordination précise des mouvements du bras robotique et de la pince, ainsi qu'une perception adéquate de l'environnement. Le succès de la tâche est défini par la capacité du robot à saisir l'objet et à le relâcher avec succès à l'emplacement désigné. L'environnement de la simulation est illustré dans la Figure 1.

3. Éléments de l'Environnement d'Apprentissage par Renforcement

L'environnement d'apprentissage par renforcement est défini par trois composantes principales: l'espace états, l'espace actions et la fonction de récompense.

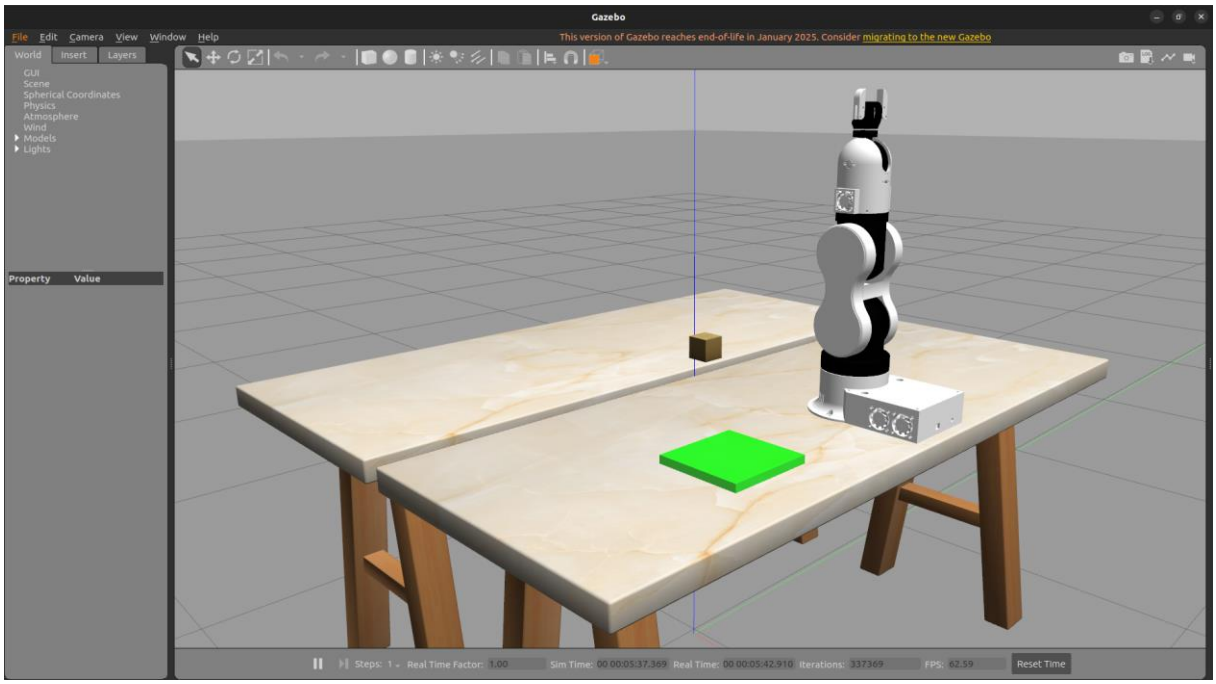

Figure 100. L'environnement simulé sur GAZEBO.

3.1. Espace d'États (State Space)

L'espace états est un vecteur de 14 dimensions qui fournit à l'agent les informations nécessaires pour prendre des décisions. Il est composé des éléments suivants :

Dimension	Description
3	Position de l'organe terminal "pince" (x, y, z)
4	Orientation de l'organe terminal (ox, oy, oz, ow)
1	État de la pince (0: ouvert, 1: fermé)
3	Position de la boîte par rapport à l'organe terminal (vecteur [x, y, z])
3	Position de la destination par rapport à l'organe terminal (vecteur [x, y, z])

Tableau 1. Description du vecteur d'état.

3.2. Espace d'Actions (Action Space)

L'espace des actions est un vecteur de quatre dimensions qui définit les actions que l'agent peut entreprendre à chaque pas de temps.

Dimension	Description
3	Position ciblée par l'organe terminal (x, y, z)
1	Commande de la pince (0: ouvrir, 1: fermer)

Tableau 2. Description du vecteur d'action.

3.3. Fonction de Récompense (Reward Function)

La fonction de récompense est conçue pour guider l'agent vers l'accomplissement de la tâche de "pick and place". Elle est composée de plusieurs éléments, positifs et négatifs, qui sont additionnés pour former la récompense totale à chaque pas de temps.

Récompense/Pénalité	Valeur	Description
step_penalty	-0.1	Pénalité à chaque pas de temps pour encourager l'efficacité.
approach_box_scale	0.3	Récompense pour s'être approché de la boîte.
approach_target_scale	0.3	Récompense pour s'être approché de la cible une fois la boîte saisie.
height_alignment_scale	0.2	Récompense pour aligner la hauteur de la pince avec la destination.
pick_reward	3.0	Récompense pour avoir saisi la boîte avec succès.

place_reward	10.0	Récompense pour avoir placé la boîte avec succès à la destination.
next_box_bonus	5.0	Bonus pour être prêt à saisir la boîte suivante (dans un scénario multi-boîtes).
task_completion_bonus	20.0	Bonus pour avoir terminé la tâche avec succès.
drop_penalty	-5.0	Pénalité pour avoir laissé tomber la boîte.
workspace_violation_penalty	-2.0	Pénalité pour avoir tenté de déplacer le bras en dehors de l'espace de travail.
ik_failure_penalty	-1.0	Pénalité en cas d'échec du solveur de cinématique inverse (IK).

Tableau 3. Description du fonction de récompense.

4. L'Agent DDPG (Deep Deterministic Policy Gradient)

4.1. Architecture de l'agent

L'agent DDPG implémenté pour le robot Thor se compose des éléments suivants :

- **Réseau Actor** : Ce réseau prend en entrée le vecteur d'état et produit en sortie le vecteur d'action. Il détermine donc les mouvements du bras robotique et de l'état de la pince. Ce réseau utilise des couches entièrement connectées avec des fonctions d'activation ReLU et une normalisation par lots (BatchNorm1d) pour améliorer la stabilité de l'entraînement. La sortie est mise à l'échelle pour correspondre aux limites de l'espace d'actions (fonction $tanh$ pour les positions, et sigmoïde pour la commande de la pince).
- **Réseau Critic**: Ce réseau prend en entrée l'état actuel du robot et l'action proposée par le réseau actor, et produit en sortie une estimation de la valeur Q de cette paire état-action. Le réseau est entraîné à minimiser l'erreur entre sa prédiction de valeur Q et la valeur Q cible calculée à l'aide de la récompense et de la valeur Q du prochain état. Comme le réseau actor, ce réseau utilise des couches entièrement connectées avec ReLU et normalisées par lots.
- **Réseaux Targets** : Pour stabiliser l'entraînement, DDPG utilise des copies "cibles" des réseaux Actor et Critic. Ces réseaux cibles sont mis à jour lentement à partir des réseaux principaux (via un mécanisme de "soft update" avec le paramètre tau), ce qui permet de fournir des valeurs Q cibles plus stables pour l'entraînement du critique.
- **Replay Buffer** : Un buffer est utilisé pour stocker les transitions (état, action, récompense, prochain état, terminé) collectées lors de l'interaction de l'agent avec l'environnement. Lors de l'entraînement, des mini-lots de transitions sont échantillonnés aléatoirement à partir de ce tampon. Cela permet de briser les corrélations temporelles dans les séquences d'expériences et d'améliorer l'efficacité de l'apprentissage. Le buffer implémente également une normalisation des états pour stabiliser l'entraînement.

- **Exploration Noise** : Pour encourager l'exploration de l'environnement, un bruit gaussien est ajouté aux actions produites par le réseau acteur pendant la phase d'entraînement. L'échelle de ce bruit diminue progressivement au fil des épisodes pour permettre à l'agent de converger vers une politique déterministe optimale.

4.2. Processus d'Entraînement

Le processus d'entraînement de l'agent DDPG suit les étapes suivantes :

- **Collecte d'Expériences** : À chaque pas de temps, l'agent observe l'état actuel de l'environnement, sélectionne une action (avec du bruit pour l'exploration), exécute cette action dans l'environnement, et observe la récompense, il passe au prochain état si l'épisode est terminé. Cette transition est stockée dans le replay buffer

- **Mise à Jour du réseau Critic** : Un mini-lot de transitions est échantillonné du replay buffer. Le réseau critic est mis à jour en minimisant l'erreur quadratique moyenne entre sa prédiction de valeur Q et la valeur Q cible. La valeur Q cible est calculée en utilisant la récompense actuelle et la valeur Q du prochain état est estimée par le réseau critic cible et le réseau actor cible.

- **Mise à Jour du réseau actor** : Le réseau actor est mis à jour en utilisant le policy-gradient, qui vise à maximiser la valeur Q prédite par le réseau critic pour les actions choisies par l'acteur. Cela encourage l'acteur à produire des actions qui mènent à des récompenses plus élevées.

- **Mise à Jour des réseaux targets** : Les poids des réseaux targets sont mis à jour en effectuant une moyenne pondérée des poids des réseaux principaux. Cela assure une mise à jour progressive et stable des cibles.

- **Décroissance du Bruit** : L'échelle du bruit d'exploration est réduite au fil du temps pour permettre à l'agent de passer de l'exploration à l'exploitation de la politique apprise.

5. Implémentation dans l'Environnement ROS2

L'intégration de l'environnement de simulation et de l'agent DDPG dans ROS2 est cruciale pour permettre une communication fluide entre les différents composants du système robotique. Le cadre de communication fourni par ROS2 facilite l'implémentation de ce système dans un environnement distribué[21].

5.1. Mise en place de l'Environnement (thor_rl_env_clean.py)

Le script thor_rl_env_clean.py regroupe la simulation du robot Thor dans Gazebo via ROS2 et la définition de l'agent d'apprentissage par renforcement via Gymnasium. Ce script définit la classe ThorRLEnv, qui est une implémentation de l'environnement Gymnasium (anciennement OpenAI Gym) adaptée au robot Thor et intégrée à ROS2. Cette classe gère les interactions avec le simulateur Gazebo et le système de contrôle du robot via les interfaces ROS2.

La classe ThorRLEnv peut être initialisée avec un node ROS2. Cela permet à l'environnement de fonctionner comme un node ROS2 à part entière, capable de communiquer avec d'autres nodes.

L'environnement utilise les services MoveIt2 (via GetPositionIK pour la cinématique inverse et GetMotionPlan pour la planification de mouvement) pour calculer les positions articulaires nécessaires pour atteindre une pose de l'organe terminal donnée.

Pour exécuter les mouvements du robot, l'environnement utilise l'action client FollowJointTrajectory pour communiquer avec le contrôleur des articulations du robot. Par exemple /arm_controller/follow_joint_trajectory permet d'envoyer des trajectoires articulaires au robot et d'attendre leur exécution.

Dans son traitement, l'environnement a besoin d'obtenir toutes les positions des articulations, et leurs orientations à un instant donné. Ces informations sont cruciales pour le calcul de la cinématique inverse et la définition du vecteur État de l'agent. Dans ROS2, ces informations sont publiées périodiquement sur le topic /joint_states. Le script est donc défini comme subscriber à ce topic.

D'autre part, l'environnement est défini comme publisher au topic/thor/gripper_command afin decontrôler l'ouverture et la fermeture de la pince du robot Thor.

L'environnement interagit directement avec le monde simulé de Gazebo via des services ROS2:

- GetEntityState : Pour obtenir la position et l'orientation des objets dans le simulateur comme , la boîte en carton et l'emplacement de destination.
- SetEntityState : Pour mettre à jour la position et l'orientation des objets dans le simulateur, notamment pour simuler la boîte tenue par la pince. Si l'environnement n'arrive pas à obtenir les informations sur l'organe terminal depuis le service GetPositionIK (il est possible que les services de Gazebo ne soient pas disponibles ou échouent), il est possible de récupérer le système des coordonnées de ROS2, TF2, à partir des modules tf2_ros.Buffer et tf2_ros.TransformListener. Ces derniers sont un outil incontournable pour suivre les systèmes de coordonnées de tous les éléments dans l'environnement Gazebo et les transformations entre eux.

5.2. Intégration de l'Agent (thor_ddpg_agent.py)

Le script Python thor_ddpg_agent.py contient l'implémentation de l'agent DDPG et l'étape d'entraînement. La classe ThorRLTrainer est un node ROS2 qui orchestre le processus d'apprentissage. Elle hérite de la classe rclpy.node.Node, ce qui en fait un node ROS2 et lui permet de gérer ses propres paramètres.

Le node peut charger sa configuration d'entraînement (hyperparamètres DDPG, fréquences d'évaluation, etc.) à partir d'un fichier YAML spécifié via un paramètre ROS2 (config_file) ce qui rend l'entraînement configurable et reproductible.

ThorRLTrainer initialise une instance de ThorRLEnv en lui passant son propre node ~~ROS2~~. Cela garantit que l'environnement et l'agent partagent le même contexte ROS2 et peuvent

communiquer efficacement. Une boule d'entraînement est définie pour le lancement de l'apprentissage. Elle interagit avec l'environnement collecte les expériences, les stocke dans le replay buffer, et met à jour les réseaux de l'acteur et du critique de l'agent DDPG.

Le node utilise le système de logging ROS2 pour suivre la progression de l'entraînement. Il gère également la sauvegarde périodique des modèles de l'agent et des statistiques d'entraînement dans des répertoires spécifiés.

L'architecture basée sur ROS2 permet une modularité et une extensibilité significatives. L'environnement ThorRLEnv peut être exécuté indépendamment ou avec différents agents, et l'agent ThorRLTrainer peut interagir avec d'autres environnements compatibles Gymnasium/ROS2. Cette séparation des tâches facilite le débogage, le développement et le déploiement des systèmes robotiques basés sur l'apprentissage par renforcement.

6. Explication Détaillée des Codes

Cette section fournit une explication détaillée des deux fichiers de code fournis, *thor_rl_env_clean.py* et *thor_ddpg_agent.py*, en mettant en évidence les fonctions clés et leur rôle dans l'implémentation de l'environnement et de l'agent DDPG.

Les deux scripts travaillent en synergie : *thor_rl_env_clean.py* fournit l'interface standardisée de l'environnement d'apprentissage par renforcement, tandis que *thor_ddpg_agent.py* implémente l'algorithme DDPG et gère le processus d'entraînement en interagissant avec cet environnement via ROS2. Cette architecture modulaire permet une grande flexibilité et facilite l'expérimentation avec différents algorithmes d'apprentissage par renforcement ou environnements robotiques.

6.1. thor_rl_env_clean.py : L'Environnement du Robot Thor

Ce fichier définit la classe ThorRLEnv, qui est l'interface entre l'agent d'apprentissage par renforcement et le robot Thor simulé dans Gazebo via ROS2. Elle hérite de gymnasium.Env, respectant ainsi la structure standard des environnements d'apprentissage par renforcement. Nous exposons dans ce qui suit la structure de la classe ThorRLEnv :

- **__init__(self, node=None)** :
  - Constructeur de la classe.
  - Initialise le ROS2 (rclpy.create_node('thor_rl_env')) si aucun node n'est fourni, permettant à l'environnement de fonctionner de manière autonome ou d'être intégré à un node existant.
  - Initialise les clients de service MoveIt2 (GetPositionIK, GetMotionPlan) pour la cinématique inverse et la planification de mouvement.
  - Crée un ActionClient pour FollowJointTrajectory, utilisé pour envoyer des trajectoires aux contrôleurs d'articulations du robot.
  - Configure les subscribers (joint_state_sub) pour recevoir les états des articulations du robot et les publishers (gripper_cmd_pub) pour contrôler la pince.

  - Initialise les clients de service Gazebo (get_entity_state_client, set_entity_state_client) pour interagir avec les objets dans le simulateur.
  - Définit l'espace d'actions (self.action_space) comme une spaces.Box de quatre dimensions (dx, dy, dz, commande de pince) et l'espace d'observations (self.observation_space) comme une spaces.Box de 14 dimensions (position/orientation de l'organe terminal, état de la pince, positions relatives des objets).
  - Définit les limites de l'espace de travail du robot (ee_workspace_limits) pour contraindre les mouvements de l'organe terminal.
- **reset(self, seed=None, options=None)** :
  - Réinitialise l'environnement au début de chaque épisode d'entraînement.
  - Réinitialise le compteur de pas et l'état de la pince (ouverte).
  - Appelle _get_entities_from_gazebo() pour obtenir les poses initiales des objets dans Gazebo.Retourne l'observation initiale et un dictionnaire d'informations.
- **step(self, action)** :
  - Exécute une action donnée par l'agent dans l'environnement.
  - Décompose l'action en deltas de position (dx, dy, dz) pour l'organe terminal et une commande pour la pince.
  - Calcule la pose cible de l'organe terminal et la contraint aux limites de l'espace de travail.
  - Utilise _inverse_kinematics_safe() pour calculer les positions articulaires correspondantes via MoveIt IK.
  - Exécute la trajectoire articulaire calculée via _execute_joint_trajectory().
  - Définit l'état de la pince via _set_gripper_state().
  - Met à jour l'état de la boîte si elle est tenue par la pince (_update_held_box_pose()).
  - Calcule la récompense, l'état de fin d'épisode (done), et les informations supplémentaires (info) via _calculate_reward().
  - Retourne la nouvelle observation, la récompense, l'état done, l'état truncated et les informations.
- **_inverse_kinematics_safe(self, target_x, target_y, target_z)** :
  - Une fonction utilitaire qui tente de calculer la cinématique inverse (IK) pour une pose cible donnée.
  - Gère les échecs d'IK en retournant les positions articulaires actuelles du robot, évitant ainsi les erreurs et permettant à l'agent de recevoir une pénalité.
- **_moveit_inverse_kinematics(self, target_x, target_y, target_z)** :
  - Appelle le service MoveIt2 compute_ik pour résoudre la cinématique inverse.
  - Tente différentes orientations de l'organe terminal pour trouver une solution IK valide, améliorant ainsi le taux de succès.
- **_execute_joint_trajectory(self, joint_positions)** :
  - Envoie une trajectoire articulaire au contrôleur du robot via l'action FollowJointTrajectory.
  - Permet au robot de se déplacer vers les positions articulaires calculées par l'IK.

- **_set_gripper_state(self, state)** : Publie une commande à la pince du robot pour l'ouvrir ou la fermer.
- **._get_end_effector_pose(self)** : Récupère la pose actuelle de l'organe terminal du robot, en essayant d'abord via les services Gazebo, puis en utilisant TF2 comme solution de secours.
- **_get_observation(self)** : Construit le vecteur d'observation complet pour l'agent.Normalise la position de l'organe terminal, inclut son orientation, l'état de la pince, et les positions relatives de la boîte et de la cible.
- **_calculate_reward(self)** : Implémente la logique de la fonction de récompense, en attribuant des récompenses et des pénalités en fonction de la progression de la tâche (approche de la boîte, saisie, approche de la cible, placement, violations de l'espace de travail, échecs d'IK, etc.).

6.2. thor_ddpg_agent.py : L'Agent DDPG et le Cadre d'Entraînement

Ce fichier contient l'implémentation de l'algorithme DDPG et la classe ThorRLTrainer qui gère le processus d'entraînement complet. Les classes et fonctions clés implémentées dans ce script :

- **RLConfig** :
  - Une classe de configuration (dataclass) qui contient tous les hyperparamètres pour l'entraînement DDPG (taux d'apprentissage, dimensions des réseaux, taille du tampon de rejeu, paramètres d'exploration, etc.).
  - Permet de charger et de sauvegarder la configuration à partir de fichiers YAML, facilitant la reproductibilité des expériences.
- **ReplayBuffer** :
  - Implémente le tampon optimisé pour stocker les transitions (état, action, récompense, prochain état, terminé).
  - Permet l'échantillonnage aléatoire de mini-lots pour l'entraînement des réseaux.
  - Intègre une normalisation des états (normalization_initialized, normalizing_stats) pour stabiliser l'apprentissage, en calculant la moyenne et l'écart-type des observations.
- **SimpleGaussianNoise** : Génère un bruit gaussien pour l'exploration des actions. L'échelle du bruit diminue linéairement au fil des épisodes (update_episode()) pour favoriser l'exploitation à mesure que l'agent apprend.
- **Actor** :
  - Le réseau acteur du DDPG, implémenté avec PyTorch.
  - Prend en entrée l'observation de l'environnement et produit les actions continues (postion et commande de pince).
  - Utilise des couches linéaires et des couches de normalisation par lots.
  - Applique la fonction $tanh$ aux actions de position et sigmoïde à la commande de pince pour les contraindre aux plages appropriées.
- **Critic** :
  - Le réseau critique du DDPG, également implémenté avec PyTorch.

  - Prend en entrée l'observation et l'action de l'acteur, et estime la valeur Q de cette paire.
  - Utilise une architecture similaire à celle de l'acteur.
- **DDPG** :
  - La classe principale qui encapsule l'algorithme DDPG.
  - Initialise les réseaux actor et critic, ainsi que leurs versions targets (actor_target, critic_target).
  - Initialise les optimiseurs (Adam) pour les deux réseaux.
  - Gère la sélection des actions (select_action()), en ajoutant du bruit pour l'exploration pendant l'entraînement.
  - Implémente la logique d'entraînement (train()) : échantillonnage du tampon de rejeu, calcul des pertes du critique et de l'acteur, rétropropagation et mise à jour des poids, et mise à jour douce des réseaux cibles.
  - Fournit des méthodes pour sauvegarder et charger les modèles (save(), load()).
- **ThorRLTrainer(Node)** :
  - La classe principale pour l'entraînement, héritant de rclpy.node.Node pour s'intégrer à ROS2.
  - Charge la configuration d'entraînement, crée les répertoires pour les modèles et les logs.
  - Initialise l'environnement (ThorRLEnv), l'agent DDPG (DDPG), et le tampon de rejeu (ReplayBuffer).
  - La méthode train() contient la boucle d'entraînement principale :Réinitialise l'environnement au début de chaque épisode. Dans chaque pas de temps, l'agent sélectionne une action, l'exécute dans l'environnement, et stocke la transition. Si le tampon de rejeu est suffisamment rempli, l'agent est entraîné en utilisant un mini-lot d'expériences. Met à jour l'échelle du bruit d'exploration. Effectue des évaluations périodiques de la politique (_evaluate_policy()) pour suivre les progrès. Sauvegarde régulièrement les modèles et les statistiques d'entraînement.
  - La méthode _evaluate_policy() exécute des épisodes sans bruit d'exploration pour évaluer la performance actuelle de la politique.
  - La méthode _save_training_stats() enregistre les récompenses d'épisode, les pas de temps, et les récompenses d'évaluation pour l'analyse post-entraînement.

6.3. train_rl_agent.py : Le Script d'Entraînement

Le script train_rl_agent.py est le cœur du processus d'entraînement de l'agent DDPG pour le robot Thor. Il orchestre l'interaction entre l'environnement de simulation ROS2 et l'agent d'apprentissage par renforcement. Ce script est conçu pour être exécuté en tant que node ROS2, ce qui lui permet de communiquer avec d'autres composants du système robotique.

Le script suit une structure typique pour un programme d'entraînement en apprentissage par renforcement :

- **Importations Nécessaires** : Il importe les bibliothèques Python requises, notamment rclpy pour l'intégration ROS2, thor_rl_env_clean pour l'environnement du robot, et thor_ddpg_agent pour l'implémentation de l'agent DDPG, ainsi que des modules pour la manipulation numérique (NumPy) et la gestion du temps.
- **Classe RLAgentTrainer** : C'est la classe principale qui encapsule la logique d'entraînement. Elle hérite de rclpy.node.Node pour fonctionner comme un node ROS2.
- **Méthode __init__** : Le constructeur de la classe RLAgentTrainer initialise le node ROS2, configure les paramètres de l'entraînement, et instancie l'environnement et l'agent.
- **Méthode train** : Cette méthode contient la boucle principale d'entraînement, où les épisodes sont exécutés et l'agent est mis à jour.

Dans la méthode __init__, plusieurs étapes clés sont réalisées :

- **Initialisation du node ROS2** : super().__init__('rl_agent_trainer_node') initialise le node ROS2 avec un nom spécifique.
- **Paramètres d'Entraînement** : Des paramètres tels que le nombre d'épisodes (num_episodes), la taille du tampon de rejeu (buffer_size), la taille du lot (batch_size), le facteur de remise (gamma), les taux d'apprentissage de l'acteur et du critique (actor_lr, critic_lr), et les paramètres de bruit d'exploration (noise_std_dev, noise_decay) sont définis. Ces hyperparamètres sont cruciaux pour la performance de l'entraînement et peuvent être ajustés pour optimiser l'apprentissage.
- **Instanciation de l'Environnement** : self.env = ThorRobotEnv() crée une instance de l'environnement du robot Thor. C'est cette instance qui interagit avec la simulation ROS2.
- **Instanciation de l'Agent DDPG** : self.agent = DDPGAgent(...) crée une instance de l'agent DDPG, en lui passant les dimensions de l'espace d'état et d'action, ainsi que les hyperparamètres d'apprentissage.
- **Chargement des Modèles (Optionnel)** : Le script peut inclure une logique pour charger des modèles pré-entraînés si l'entraînement doit reprendre à partir d'un point de contrôle précédent. Cela est essentiel pour les entraînements de longue durée.

La méthode train contient la boucle principale qui gère le déroulement de l'apprentissage par renforcement. Elle itère sur un nombre prédéfini d'épisodes :

- **self.env.reset()** : Au début de chaque épisode, l'environnement est réinitialisé. Cela signifie que le robot est ramené à une position initiale prédéfinie, et l'état initial de l'environnement est observé et renvoyé.
- **self.agent.choose_action(state, add_noise=True)** : L'agent DDPG prend l'état actuel en entrée et utilise son réseau acteur pour produire une action. Pendant l'entraînement, un bruit d'exploration (généralement du bruit d'Ornstein-Uhlenbeck) est ajouté à l'action pour encourager l'agent à explorer de nouvelles stratégies et à éviter de rester bloqué dans des optima locaux.
- **self.env.step(action)** : L'action choisie par l'agent est envoyée à l'environnement. Dans le contexte de ROS2, cela implique la publication de commandes sur les topics

appropriés. L'environnement exécute l'action dans la simulation, met à jour son état interne, calcule la récompense obtenue pour cette transition, et détermine si l'épisode est terminé (done).

- **self.agent.remember(state, action, reward, next_state, done)** : La transition complète (état actuel, action prise, récompense reçue, nouvel état, et indicateur de fin d'épisode) est stockée dans le tampon de rejeu de l'agent. Ce tampon est essentiel pour l'apprentissage hors-politique du DDPG, permettant de réutiliser les expériences passées de manière aléatoire.
- **self.agent.learn()** : C'est l'étape où l'agent met à jour ses réseaux acteur et critique. Il échantillonne un mini-lot aléatoire de transitions à partir du tampon de rejeu et utilise ces données pour calculer les gradients et ajuster les poids des réseaux. Cette étape est cruciale pour l'apprentissage de la politique optimale.
- **rclpy.spin_once(self.env)** : Cette ligne est vitale pour l'intégration ROS2. Elle permet au node de l'environnement (self.env) de traiter les messages ROS2 en attente (par exemple, les mises à jour des capteurs du robot, les réponses aux services). Sans cela, la communication ROS2 ne fonctionnerait pas correctement et l'environnement ne pourrait pas fournir des états à jour ou exécuter les actions.
- **Décroissance du bruit d'exploration** : Après chaque épisode, l'amplitude du bruit d'exploration est réduite (self.agent.noise_std_dev *= self.agent.noise_decay). Cela permet à l'agent d'explorer davantage au début de l'entraînement et de se concentrer sur l'exploitation de sa politique apprise à mesure que l'entraînement progresse.
- **Sauvegarde des Modèles** : À intervalles réguliers (self.save_interval), les poids des réseaux de l'acteur et du critique sont sauvegardés. Cela permet de sauvegarder les progrès de l'entraînement et de reprendre l'apprentissage en cas d'interruption, ou d'utiliser le modèle entraîné pour l'évaluation

7. Conclusion

Ce chapitre a mis en forme l'implémentation pratique de la simulation et du contrôle du robot Thor via Reinforcement Learning, en se focalisant sur l'algorithme DDPG. Nous avons détaillé la conception de l'environnement d'apprentissage par renforcement, incluant les espaces d'états, d'actions et la fonction de récompense, ainsi que l'architecture et le processus d'entraînement de l'agent DDPG. L'intégration de ces composants au sein de l'écosystème ROS2 a été expliquée, soulignant la modularité et la communication fluide entre les différents éléments du système robotique simulé. Ce travail démontre la capacité du RL à permettre au robot Thor d'apprendre des tâches de manipulation complexes de manière autonome, ouvrant des perspectives prometteuses pour le contrôle robotique avancé.

Conclusion générale

Dans le domaine de la robotique, qui est un domaine en constante évolution, le Reinforcement Learning (RL) s'impose comme un outil moderne et un objet de recherche prometteur pour doter les systèmes autonomes de capacités d'apprentissage et d'adaptation.

Ce mémoire a eu pour objectif de démontrer la faisabilité et l'efficacité de la simulation et du contrôle d'un bras manipulateur industriel Thor à 6 degrés de liberté (DoF) au sein de l'environnement Gazebo, en exploitant l'architecture ROS2 et les avancées de l'apprentissage par renforcement profond, notamment l'algorithme Deep Deterministic Policy Gradient (DDPG). Notre tâche principale a consisté à entraîner ce bras robotique à exécuter des mouvements complexes de manière autonome dans un environnement simulé, en s'affranchissant des méthodes de programmation traditionnelles. Pour ce faire, nous avons configuré l'environnement ROS2 avec les packages thor_description pour la modélisation du robot, thor_gazebo pour sa simulation physique réaliste, et thor_moveit_config pour la planification de mouvement. Nous avons ainsi appris à construire un écosystème de simulation complet et à mettre en œuvre des algorithmes d'apprentissage sophistiqués pour la commande robotique.
Au cours de notre étude, nous avons développé des compétences et connaissances pertinentes en programmation. Cependant, ce parcours n'a pas été sans difficultés, notamment en ce qui concerne l'optimisation des paramètres de l'algorithme DDPG, la gestion des interactions complexes entre les différents composants ROS2 et Gazebo, et la rareté des document officiel sur le ROS2 et Gazebo. Malgré ces défis, cette étude ouvre des perspectives considérables pour l'avenir. Les prochaines étapes seraient de trouver des solutions pour résoudre les problèmes d’interaction entre les composants de la simulation, explorer d'algorithmes d'apprentissage par renforcement plus avancés et intégrer des capacités de perception plus sophistiquées pour une autonomie accrue.

Ce sujet reste au cœur de la recherche vu son utilité dans divers champs d’applications y compris le domaine industriel, comme la gestion des dépôts, la fabrication et le montage complexes.

Bibliographie

[1] « ROS 2 Documentation — ROS 2 Documentation: Humble documentation ». Consulté le: 4 juillet 2025. [En ligne]. Disponible sur: https://docs.ros.org/en/humble/index.html

[2] « ROS2 : qu'est-ce qui change par rapport à ROS ? » Consulté le: 5 juillet 2025. [En ligne]. Disponible sur: https://www.generationrobots.com/blog/fr/ros2-quest-ce-qui-change-par-rapport-a-ros/?srsltid=AfmBOortltvI3h9UtY94IR6HMFKL7uuzlxJ7nMRIIRqKQ6aLZR-d73uV

[3] « Nodes — ROS 2 Documentation: Humble documentation ». Consulté le: 4 juillet 2025. [En ligne]. Disponible sur: https://docs.ros.org/en/humble/Concepts/Basic/About-Nodes.html

[4] « Understanding topics — ROS 2 Documentation: Humble documentation ». Consulté le: 4 juillet 2025. [En ligne]. Disponible sur: https://docs.ros.org/en/humble/Tutorials/Beginner-CLI-Tools/Understanding-ROS2-Topics/Understanding-ROS2-Topics.html

[5] « Understanding services — ROS 2 Documentation: Humble documentation ». Consulté le: 4 juillet 2025. [En ligne]. Disponible sur: https://docs.ros.org/en/humble/Tutorials/Beginner-CLI-Tools/Understanding-ROS2-Services/Understanding-ROS2-Services.html

[6] « Understanding actions — ROS 2 Documentation: Humble documentation ». Consulté le: 4 juillet 2025. [En ligne]. Disponible sur: https://docs.ros.org/en/humble/Tutorials/Beginner-CLI-Tools/Understanding-ROS2-Actions/Understanding-ROS2-Actions.html

[7] « Interfaces — ROS 2 Documentation: Humble documentation ». Consulté le: 5 juillet 2025. [En ligne]. Disponible sur: https://docs.ros.org/en/humble/Concepts/Basic/About-Interfaces.html#messages

[8] « Understanding parameters — ROS 2 Documentation: Humble documentation ». Consulté le: 4 juillet 2025. [En ligne]. Disponible sur: https://docs.ros.org/en/humble/Tutorials/Beginner-CLI-Tools/Understanding-ROS2-Parameters/Understanding-ROS2-Parameters.html

[9] « Creating a workspace — ROS 2 Documentation: Humble documentation ». Consulté le: 4 juillet 2025. [En ligne]. Disponible sur: https://docs.ros.org/en/humble/Tutorials/Beginner-Client-Libraries/Creating-A-Workspace/Creating-A-Workspace.html

[10] « Creating a package — ROS 2 Documentation: Humble documentation ». Consulté le: 4 juillet 2025. [En ligne]. Disponible sur: https://docs.ros.org/en/humble/Tutorials/Beginner-Client-Libraries/Creating-Your-First-ROS2-Package.html

[11] « Launch — ROS 2 Documentation: Humble documentation ». Consulté le: 4 juillet 2025. [En ligne]. Disponible sur: https://docs.ros.org/en/humble/Concepts/Basic/About-Launch.html

[12] « Gazebo ». Consulté le: 4 juillet 2025. [En ligne]. Disponible sur: https://gazebosim.org/home

[13] « rviz2 — RViz 1.9.0 documentation ». Consulté le: 4 juillet 2025. [En ligne]. Disponible sur: https://docs.ros.org/en/rolling/p/rviz2/

[14] « MoveIt 2 Documentation — MoveIt Documentation: Rolling documentation ». Consulté le: 4 juillet 2025. [En ligne]. Disponible sur: https://moveit.picknik.ai/main/index.html

[15] MerAARIZOU, *MerAARIZOU/thor_ros2*. (19 mai 2025). Python. Consulté le: 8 juillet 2025. [En ligne]. Disponible sur: https://github.com/MerAARIZOU/thor_ros2

[16] R. S. Sutton et A. G. Barto, « Reinforcement Learning: An Introduction ».

[17] « Markov decision process », *Wikipedia*. 27 juin 2025. Consulté le: 4 juillet 2025. [En ligne]. Disponible sur: https://en.wikipedia.org/w/index.php?title=Markov_decision_process&oldid=1297575772

[18] R. S. Sutton et A. G. Barto, *Reinforcement learning: an introduction*. in Adaptive computation and machine learning. Cambridge, Mass: MIT Press, 1998.

[19] « Papers with Code - Continuous control with deep reinforcement learning ». Consulté le: 4 juillet 2025. [En ligne]. Disponible sur: https://paperswithcode.com/paper/continuous-control-with-deep-reinforcement

[20] Machine Learning with Phil, *How to Implement Deep Learning Papers | DDPG Tutorial*, (2 juillet 2019). Consulté le: 4 juillet 2025. [En ligne Vidéo]. Disponible sur: https://www.youtube.com/watch?v=jDll4JSI-xo

[21] Machine Learning with Phil, *Reinforcement Learning in Continuous Action Spaces | DDPG Tutorial (Pytorch)*, (28 juin 2019). Consulté le: 8 juillet 2025. [En ligne Vidéo]. Disponible sur: https://www.youtube.com/watch?v=6Yd5WnYls_Y

[22] « Project | Thor | Hackaday.io ». Consulté le: 8 juillet 2025. [En ligne]. Disponible sur: https://hackaday.io/project/12989/logs?sort=newest&page=2

Annexe

Catégorie	Syntaxe de la Commande	Arguments/Options	Objectif/Description
Gestion de l'Espace de Travail	mkdir -p <nom_espace_travail>/src		Crée la structure de répertoire d'un espace de travail ROS2.
	cd <nom_espace_travail>		Change le répertoire courant vers l'espace de travail.
	source install/setup.bash		Source le fichier de configuration de l'espace de travail pour rendre les paquets découvrables.
Compilation	colcon build	--build-base BUILD_BASE	Compile un ensemble de paquets ROS2. L'argument
		--install-base INSTALL_BASE	Spécifie le chemin de base pour tous les préfixes d'installation.
		--merge-install	Installe le préfixe pour tous les paquets au lieu d'un sous-répertoire spécifique au paquet.
		--symlink-install	Utilise des liens symboliques au lieu de copier les fichiers depuis la source pour une itération plus rapide.
		--test-result-base TEST_RESULT_BASE	Spécifie le chemin de base pour tous les résultats de test.
		--continue-on-error	Continue la compilation des autres paquets lorsqu'un paquet échoue.
		--executor sequential	Traite un paquet à la fois.
		--executor parallel	Traite plusieurs tâches en parallèle.
		--packages-select PKG_NAME	Sélectionne des paquets spécifiques par leur nom.
		--packages-skip PKG_NAME	Ignore des paquets spécifiques par leur nom.
		--packages-up-to <nom_paquet>	Compile le paquet sélectionné et ses dépendances.
		--packages-above <nom_paquet>	Compile le paquet sélectionné et les paquets qui en dépendent.
		--parallel-workers NUMBER	Spécifie le nombre maximal de tâches à traiter en parallèle (par défaut : cœurs logiques du CPU).

		--event-handlers console_direct+ ou console_cohesion+	Affiche directement la sortie de compilation sur stdout/stderr.
Graphe	colcon graph	--density	Génère une représentation visuelle du graphe de dépendances des paquets. L'argument
		--legend	Affiche une légende pour le graphe de dépendances.
		--dot	Affiche le graphe topologique au format DOT (langage de description de graphe).
		--dot-cluster	Regroupe les paquets par leur chemin dans le système de fichiers.
		--packages-up-to <nom_paquet>	Génère un graphe de dépendances pour le paquet sélectionné et ses dépendances
Info	colcon info	--base-paths <chemins>	Affiche des informations détaillées sur les paquets
		--paths <chemins>	Spécifie les chemins à vérifier pour un paquet.
		--packages-select <NOM_PAQUET>	Ne traite qu'un sous-ensemble de paquets.
		--packages-skip <NOM_PAQUET>	Ignore un ensemble de paquets.
Liste	colcon list	--topological-order ou -t	Énumère un ensemble de paquets. L'option
		--names-only ou -n	Affiche uniquement le nom de chaque paquet.
		--paths-only ou -p	Affiche uniquement le chemin de chaque paquet.
		--packages-up-to <nom_paquet>	Liste le paquet sélectionné et ses dépendances.
Test	colcon test	--build-base BUILD_BASE	Exécute des tests pour un ensemble de paquets. L'argument
		--install-base INSTALL_BASE	Spécifie le chemin de base pour tous les préfixes d'installation.
		--merge-install	Installe le préfixe pour tous les paquets au lieu d'un sous-répertoire spécifique au paquet.
		--test-result-base TEST_RESULT_BASE	Spécifie le chemin de base pour tous les résultats de test.
		--retest-until-fail N	Ré exécute les tests jusqu'à N fois s'ils réussissent.
		--retest-until-pass N	Ré exécute les tests échoués jusqu'à N fois.

		--abort-on-error	Interrompt après le premier paquet présentant des erreurs.
		--return-code-on-test-failure	Utilise un code de retour non nul pour indiquer tout échec de test.
		--ctest-args tests [arguments_sélection_paquet]	Passe des arguments à CTest.
		--packages-select <nom_paquet>	Exécute les tests pour un paquet spécifique.
Résultat de Test	colcon test-result	--test-result-base TEST_RESULT_BASE	Résume les résultats des tests précédemment exécutés. L'argument
		--all	Affiche tous les fichiers de résultats de test, y compris ceux sans erreurs/échecs.
		--verbose	Affiche des informations supplémentaires pour chaque erreur/échec.
		--result-files-only	N'imprime que les chemins des fichiers de résultats.
		--delete	Supprime tous les fichiers de résultats.
		--delete-yes	Identique à --delete sans confirmation interactive.

Tableau 4.Les commandes de compilation.

Catégorie	Syntaxe de la Commande	Arguments/Options	Description
Action	ros2 action list	-t ou --show-types	Liste toutes les actions dans le graphe ROS.
	ros2 action info <nom_action>		Obtient des informations détaillées sur une action spécifique (clients, serveurs).
	ros2 action send_goal <nom_action> <type_action> <valeurs>	--feedback	Envoie un objectif à un serveur d'action (les valeurs doivent être au format YAML). L'argument
Bag	ros2 bag record <nom_sujet> [<nom_sujet>...]	-o <nom_fichier>	Enregistre les données publiées sur les sujets spécifiés. L'option

		-a	Enregistre tous les sujets du système.
		-e <regex>	Enregistre uniquement les sujets correspondant à une expression régulière fournie.
		-x <regex>	Exclut les sujets correspondant à une expression régulière fournie.
	ros2 bag info <nom_fichier_bag>		Affiche les détails d'un fichier bag enregistré.
	ros2 bag play <nom_fichier_bag>	--clock	Rejoue les données d'un fichier bag enregistré .
		-l	Active la lecture en boucle.
		-r <taux>	Spécifie le taux de lecture des messages.
		-s <id_stockage>	Spécifie l'identifiant de stockage à utiliser (par défaut 'sqlite3').
		--topics <nom_sujet> [<nom_sujet>...]	Sujets à rejouer, séparés par des espaces.
		--storage-config-file <chemin>	Chemin vers un fichier YAML définissant les configurations spécifiques au stockage.
Composant	ros2 component list		Liste les conteneurs et composants en cours d'exécution.
	ros2 component load <nom_nœud_conteneur> <nom_paquet> <type_composant>	-n <nom>	Charge un composant dans un nœud conteneur. -n spécifie le nom du nœud composant.

		--node-namespace <namespace>	Spécifie l'espace de noms du nœud composant.
		--log-level <niveau>	Spécifie le niveau de journalisation du nœud composant.
		-r <de:=vers>	Spécifie les règles de remappage du nœud composant.
		-p <nom:=valeur>	Spécifie les paramètres du nœud composant.
	ros2 component standalone <nom_paquet> <type_composant>		Exécute un composant dans son propre nœud conteneur autonome.
	ros2 component types		Liste les types de composants enregistrés dans l'index ament.
	ros2 component unload <nom_nœud_conteneur> <id_composant>		Décharge un composant d'un nœud conteneur.
Contrôle	ros2 control list_controller_types		Affiche les types de contrôleurs disponibles et leurs classes de base.
	ros2 control list_controllers		Affiche les contrôleurs chargés, leur type et leur statut.
	ros2 control list_hardware_interfaces	--claimed-interfaces	Affiche les interfaces matérielles chargées, leur type et leur statut. L'option
		--required-state-interfaces	Liste les interfaces d'état requises par le contrôleur.
		--required-command-interfaces	Liste les interfaces de commande requises par le contrôleur.

	ros2 control load_controller <nom_contrôleur>	-c <nom_nœud_gestionnaire_contrôleur>	Charge un contrôleur dans un gestionnaire de contrôleurs. L'argument
	ros2 control reload_controller_libraries		Recharge les bibliothèques de contrôleurs
	ros2 control set_controller_state <nom_contrôleur> <état>		Ajuste l'état du contrôleur.
	ros2 control switch_controllers --start-controllers <nom_contrôleur> [...] --stop-controllers <nom_contrôleur> [...]		Change les contrôleurs dans un gestionnaire de contrôleurs.
	ros2 control unload_controller <nom_contrôleur>		Décharge un contrôleur dans un gestionnaire de contrôleurs.
	ros2 control view_controller_chains		Génère un diagramme des contrôleurs chaînés chargés.
Démon	ros2 daemon start		Démarre le démon s'il n'est pas déjà en cours d'exécution.
	ros2 daemon status		Affiche le statut du démon.
	ros2 daemon stop		Arrête le démon s'il est en cours d'exécution.
Points d'Extension	ros2 extension_points	--all ou -a	Liste les points d'extension disponibles dans le système ROS2. L'option --all affiche les points d'extension qui n'ont pas pu être importés.
		--verbose ou -v	Affiche plus d'informations pour chaque point d'extension.

Extensions	ros2 extensions	-a	Liste les extensions d'un paquet. L'option -a affiche les extensions qui n'ont pas pu être chargées ou qui sont incompatibles.
		-v	Affiche plus d'informations pour chaque extension.
Interface	ros2 interface list		Liste tous les types d'interface disponibles.
	ros2 interface package <nom_paquet>		Affiche une liste des types d'interface disponibles au sein d'un paquet.
	ros2 interface packages		Affiche une liste des paquets qui fournissent des interfaces.
	ros2 interface show <type_interface>		Affiche la définition de l'interface.
Lancement	ros2 launch [nom_paquet][nom_fichier_launch]	-n ou --noninteractive	Lance un système ROS2. L'option -n exécute le système de lancement de manière non interactive.
		-d ou --debug	Met le système de lancement en mode débogage pour une sortie plus verbeuse.
		-p ou --print ou --print-description	Imprime la description du lancement sur la console sans le lancer.
		-s ou --show-args ou --show-arguments	Affiche les arguments qui peuvent être donnés au fichier de lancement.

Nœud	ros2 run [nom_paquet][nom_exécutable]	--prefix PREFIX	Exécute un exécutable dans un paquet. L'argument
		--ros-args	Passe des arguments lors de l'exécution d'un nœud.
		--remap <de:=vers>	Renomme les noms des sujets lors de l'exécution d'un nœud.
		--params-file <nom_fichier>	Démarre le nœud avec les valeurs de paramètres sauvegardées.
	ros2 node list		Liste tous les nœuds en cours d'exécution.
	ros2 node info <nom_nœud>		Accède à plus d'informations sur un nœud.
Paquet	ros2 pkg create --build-type <ament_python|ament_cmake> [nom_paquet]		Crée un nouveau paquet ROS2. Les types de construction incluent
	ros2 pkg list		Liste tous les paquets disponibles.
Paramètre	ros2 param list [<nom_nœud>]		Liste tous les paramètres disponibles sur un nœud donné, ou sur tous les nœuds découvrables si aucun nœud n'est spécifié.
	ros2 param get <nom_nœud> <nom_paramètre>		Obtient la valeur d'un paramètre particulier sur un nœud.
	ros2 param set <nom_nœud> <nom_paramètre> <valeur>		Définit la valeur d'un paramètre particulier sur un nœud. La valeur doit être au format YAML et le type doit correspondre.

	ros2 param delete <nom_nœud> <nom_paramètre>		Supprime un paramètre d'un nœud (ne peut supprimer que les paramètres dynamiques).
	ros2 param describe <nom_nœud> <nom_paramètre>		Fournit une description textuelle d'un paramètre.
	ros2 param dump <nom_nœud>	--output-dir <chemin>	Affiche tous les paramètres d'un nœud au format de fichier YAML. L'argument
	ros2 param load <nom_nœud> <fichier_paramètre>		Charge les valeurs des paramètres d'un fichier YAML dans un nœud.
Outils rqt	rqt_graph		Affiche les nœuds et sujets actifs.
	rqt		Affiche un écran avec des éléments de menu déroulants pour visualiser les données.
Sujet	ros2 topic list	-t ou --show-type	Liste tous les sujets actuellement actifs. L'option
	ros2 topic echo <nom_sujet>		Affiche les données publiées sur un sujet spécifié.[1]
	ros2 topic info <nom_sujet>		Fournit des informations sur un sujet spécifique (type, nombre de publicateurs/abonnés).
	ros2 topic pub <--once> <nom_sujet> <type_msg> '<args>'	-r <taux>	Publie des données sur un sujet directement depuis la ligne de commande (les arguments doivent être au format YAML). L'argument

		-p <num>	N'imprime que le N-ième message publié.
		-1 ou --once	Publie un message et quitte.
		-t <num>	Publie ce nombre de fois et quitte.
		--keep-alive <secondes>	Maintient le nœud de publication actif pendant N secondes après le dernier message.
	ros2 topic hz <nom_sujet>		Affiche le taux (fréquence) auquel les données sont publiées.
	ros2 topic bw <nom_sujet>		Affiche la bande passante utilisée par un sujet.
	ros2 topic find <type_sujet>		Liste tous les sujets disponibles d'un type de message donné.

Tableau 5.Les commandes des action sur ROS2.

les caractiristique de robote thor [22]

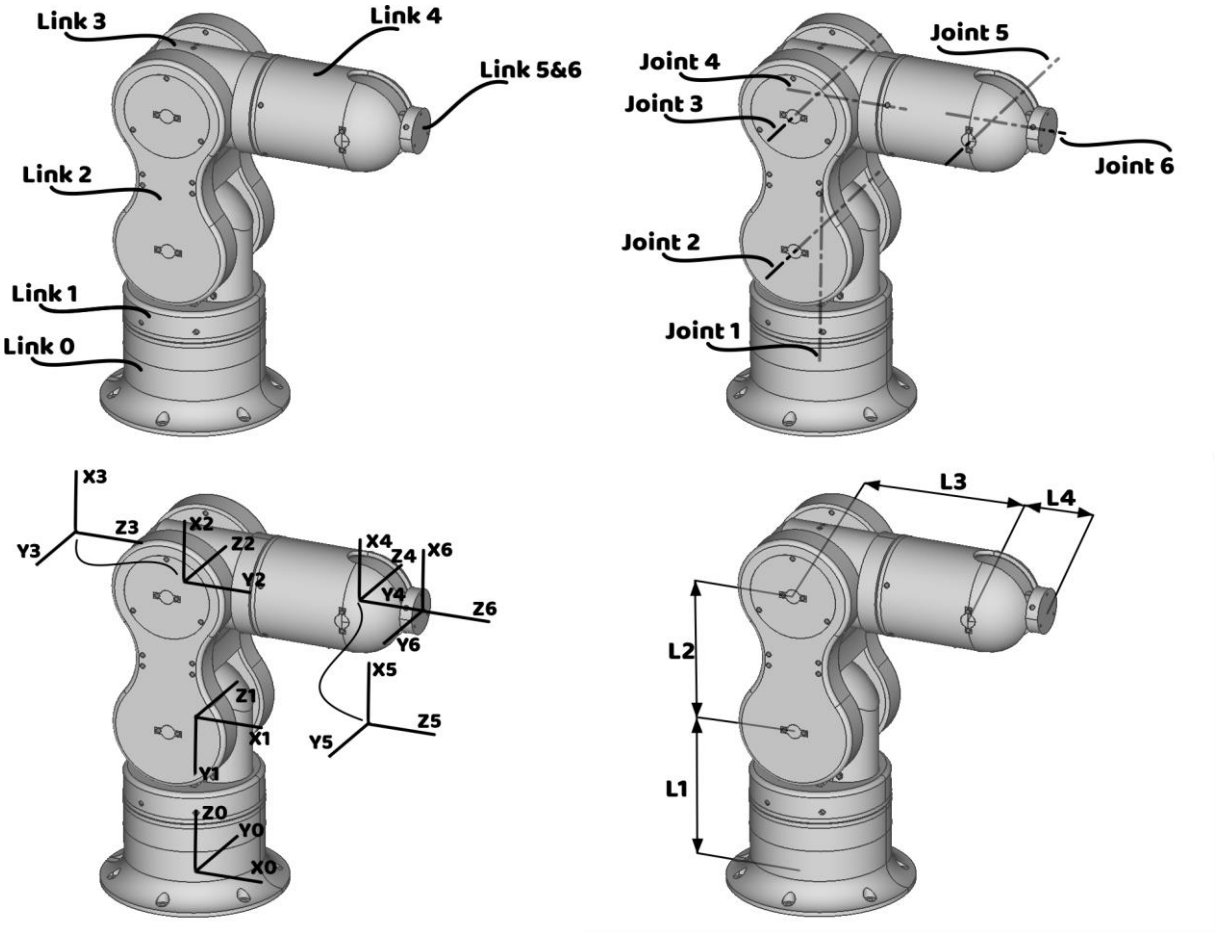

Figure 11. illustration des articulation et repaire des robot.

Link	θ	d	a	α
1	q1	L1	0	90
2	q2-90	0	L2	0
3	q3	0	0	90
4	q4	L3	0	-90
5	q5	0	0	90
6	q6	0	0	0

Tableau 6.Les donné de Denavit-hartenberg.

Les matrix de transformation

$$A_1^0=\begin{bmatrix} C1 & 0 & S1 & 0 \\ S1 & 0 & -C1 & 0 \\ 0 & 1 & 0 & L1 \\ 0 & 0 & 0 & 1 \end{bmatrix} A_2^1=\begin{bmatrix} S2 & C2 & 0 & S2L2 \\ -C2 & S2 & 0 & -L2C2 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} A_3^2=\begin{bmatrix} C3 & 0 & S3 & 0 \\ S3 & 0 & -C3 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$A_4^3=\begin{bmatrix} C4 & 0 & -S4 & 0 \\ S4 & 0 & C4 & 0 \\ 0 & -1 & 0 & L3 \\ 0 & 0 & 0 & 1 \end{bmatrix} A_5^4=\begin{bmatrix} C5 & 0 & S5 & 0 \\ S5 & 0 & -C5 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} A_6^5=\begin{bmatrix} C6 & -S6 & 0 & 0 \\ S6 & C6 & 0 & 0 \\ 0 & 0 & 1 & L4 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$\mathrm{T}=\begin{bmatrix} n_x & o_x & a_x & p_x \\ n_y & o_y & a_y & p_y \\ n_z & o_z & a_z & p_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$n_x = C_6(C_5(C_4(C_1C_2S_3+C_1C_3S_2)+S_1S_4)+S_5(C_1C_2C_3-C_1S_2S_3))+S_6(C_4S_1-S_4(C_1C_2S_3+C_1C_3S_2))$$

$$n_y = C_6(C_5(-C_1S_4+C_4(C_2S_1S_3+C_3S_1S_2))+S_5(C_2C_3S_1-S_1S_2S_3))+S_6(-C_1C_4-S_4(C_2S_1S_3+C_3S_1S_2))$$

$$n_z = C_6(C_4C_5(-C_2C_3+S_2S_3)+S_5(C_2S_3+C_3S_2))-S_4S_6(-C_2C_3+S_2S_3)$$

$$o_x = C_6(C_4S_1-S_4(C_1C_2S_3+C_1C_3S_2))-S_6(C_5(C_4(C_1C_2S_3+C_1C_3S_2)+S_1S_4)+S_5(C_1C_2C_3-C_1S_2S_3))$$

$$o_y = C_6(-C_1C_4-S_4(C_2S_1S_3+C_3S_1S_2))-S_6(C_5(-C_1S_4+C_4(C_2S_1S_3+C_3S_1S_2))+S_5(C_2C_3S_1-S_1S_2S_3))$$

$$o_z = -C_6S_4(-C_2C_3+S_2S_3)-S_6(C_4C_5(-C_2C_3+S_2S_3)+S_5(C_2S_3+C_3S_2))$$

$$a_x = -C_5(C_1C_2C_3-C_1S_2S_3)+S_5(C_4(C_1C_2S_3+C_1C_3S_2)+S_1S_4)$$

$$a_y = -C_5(C_2C_3S_1-S_1S_2S_3)+S_5(-C_1S_4+C_4(C_2S_1S_3+C_3S_1S_2))$$

$$a_z = C_4S_5(-C_2C_3+S_2S_3)-C_5(C_2S_3+C_3S_2)$$

$$p_x = C_1L_2S_2+L_3(-C_1C_2C_3+C_1S_2S_3)+L_4(-C_5(C_1C_2C_3-C_1S_2S_3)+S_5(C_4(C_1C_2S_3+C_1C_3S_2)+S_1S_4))$$

$$p_y = L_2S_1S_2+L_3(-C_2C_3S_1+S_1S_2S_3)+L_4(-C_5(C_2C_3S_1-S_1S_2S_3)+S_5(-C_1S_4+C_4(C_2S_1S_3+C_3S_1S_2)))$$

$$p_z = -C_2L_2+L_1+L_3(-C_2S_3-C_3S_2)+L_4(C_4S_5(-C_2C_3+S_2S_3)-C_5(C_2S_3+C_3S_2))$$