Fabrication et programmation d'un robot bipède Bolt

Paul Rouanet

▶ To cite this version:

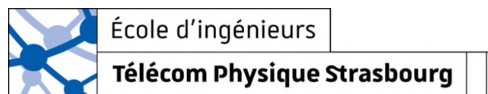
Paul Rouanet. Fabrication et programmation d'un robot bipède Bolt. Robotique [cs.RO]. 2021. hal-03347986

HAL Id: hal-03347986
https://laas.hal.science/hal-03347986

Submitted on 17 Sep 2021

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.



ROUANET, Paul
Promotion 2021
2020/2021

Diplôme d'ingénieur Télécom Physique Strasbourg
Mémoire de stage de 3e année

"Fabrication et programmation d'un robot bipède Bolt"

LAAS-CNRS
7 Avenue du Colonel Roche,
31400 Toulouse

STASSE, Olivier
ostasse@laas.fr

Du 01/03/2021 au 01/09/2021

Remerciements

Je tiens à adresser un grand merci à mon tuteur, Olivier Stasse, qui m'a accordé sa confiance et son temps, ainsi que de l'aide et des conseils qui m'ont été fort utiles. Il a été un grand élément de la réussite personnelle que j'ai tirée de ce stage. Je remercie aussi grandement Thomas Flayols, qui m'a formé au montage du robot et qui a donc fait que cette partie se soit idéalement déroulée.

Je remercie l'ensemble du personnel du LAAS-CNRS que j'ai côtoyé durant ces six mois de stage, pour leur accueil et leur sympathie. Plus particulièrement les membres de l'équipe Gepetto, stagiaires, doctorants, post-doctorants et bien entendu chercheurs permanents avec qui j'ai pu échanger autour de notions intéressantes, et avec qui j'ai partagé l'espace de travail.

Je remercie enfin l'ensemble de la communauté Open Dynamic Robot Initiative, notamment Felix Grimminger, pour leurs réponses et le suivi de mes avancements depuis l'étranger.

Résumé français

Fabrication et programmation d'un robot bipède Bolt

Ce projet de fin d'étude s'est déroulé au LAAS-CNRS à Toulouse, au sein de l'équipe Gepetto, qui travaille sur l'analyse et la génération de mouvements des robots humanoïdes et de l'Homme. L'objectif du stage était de fabriquer pour cette équipe un robot bipède open-source nommé Bolt afin d'en faire un intermédiaire de tests d'algorithmes, entre les simulateurs informatiques et des systèmes réels plus imposants. Bolt a été développé par le groupement de chercheurs Open Dynamic Robot Initiative (ODRI).

La première phase a été une partie de conception matérielle du robot. Elle a commencé par une prise en main des explications stockées sur GitHub à propos de Bolt : plans 3D, listes de pièces et méthodes de montage. La réalisation technique s'est faite en réutilisant les éléments d'un robot et en fabriquant les pièces à l'aide d'une imprimante 3D. Cela a permis une fabrication rapide et robuste du bipède.

La seconde phase a été une partie logicielle de mise en fonctionnement du robot à partir des codes déjà disponibles. Pour cela, il a fallu comprendre le rôle de nombreux codes regroupés en librairies, les installer et les faire fonctionner sur un ordinateur du laboratoire. Cette partie beaucoup plus informatique a été complexe et riche en apprentissages sur l'environnement d'un ordinateur.

Enfin, la troisième partie a consisté à adapter l'ensemble à une infrastructure logicielle de ROS 2 en cours de développement : ros2_control. Là encore il a fallu comprendre de nombreux codes afin d'accorder au mieux le comportement de Bolt à ces librairies : allumage, lecture des capteurs, envoi de commandes et arrêt. Ainsi, Bolt est devenu le premier robot à fonctionner via cette infrastructure logicielle.

Finalement, l'équipe Gepetto peut maintenant tester ses algorithmes de commande sur un robot réel simplifié, validant ainsi l'objectif du stage.

English summary

Manufacturing and programming of a Bolt bipedal robot

This graduation project took place at LAAS-CNRS, at Toulouse, in the Gepetto team, who work on the analysis and generation of humanoid robots and human movements. The objective of the internship was to build for this team an open-source bipedal robot named Bolt that will be used as an intermediary for testing algorithms, between computer simulators and larger real systems. Bolt was designed by the Open Dynamic Robot Initiative (ODRI) research group.

The first step was robot hardware design. It was necessary to get familiar with all the explanations stored on a GitHub repository about Bolt : 3D plans, parts lists, and assembly methods. The technical realization was done by reusing the elements of another robot and by making the parts with a 3D printer. This allowed a fast and robust manufacturing of the biped.

The second step was a software part to put the robot in operation from codes already available. For that, it was necessary to understand the role of many codes grouped in packages, to install them and to make them work on a laboratory computer. This part was much more complex and enriching about computer environment.

Finally, the third step consisted in adapting the whole to a software infrastructure of ROS 2 under development : ros2_control. Here again, many codes had to be understood in order to adjust as well as possible Bolt's behaviour to those packages : switching on, reading the sensors, sending commands and shutting off. Hence, Bolt became the first robot in the world to run through this software infrastructure.

In conclusion, the Gepetto team can now test their control algorithms on a simplified real robot, thus validating the objective of the internship.

Table des matières

Introduction 1
Contexte . . . 1
Objectifs du stage . . . 2

1 Présentation de l'organisme d'accueil 4
1.1 Le LAAS-CNRS . . . 4
1.2 L'équipe Gepetto . . . 5

2 Hardware 7
2.1 État de l'art . . . 7
2.2 Étapes de montage . . . 9
2.2.1 Recyclage d'un robot précédent - Solo 8 . . . 9
2.2.2 Fabrication des pièces . . . 10
2.2.3 Mécanique . . . 14
2.2.4 Électronique . . . 19
2.3 Difficultés rencontrées et solutions . . . 25
2.3.1 Impression 3D . . . 25
2.3.2 Commandes . . . 26
2.3.3 Électronique . . . 27
2.3.4 Présentiel / Distanciel . . . 27

3 Software 28
3.1 État de l'art et architecture . . . 28
3.2 Compréhension et compilation . . . 29
3.2.1 Compréhension . . . 29
3.2.2 Compilation . . . 30
3.2.3 Tests . . . 32
3.2.4 Difficultés . . . 33
3.3 Passage à ros2_control . . . 35
3.3.1 Prise en main . . . 35
3.3.2 Modifications . . . 37
3.3.3 Tests . . . 39
3.3.4 Difficultés . . . 40

Conclusion 41

Bibliographie 44

Annexes 45

A Organigrammes du LAAS-CNRS (2021) 45

B Arbre de dépendance simplifié des codes du Bolt 47

C Arbre de dépendance simplifié de ros2_control 48

D Architecture de ros2_control 49

E Exemple fichier URDF 50

Table des figures

1 Robots TIAGo (à gauche) et TALOS (à droite) de chez PAL Robotics 1
2 Schéma bloc du fonctionnement de TALOS 2

1.1 Logo du LAAS-CNRS 4
1.2 Photo du bâtiment d'accueil du LAAS-CNRS 4
1.3 Logo de l'équipe Gepetto 5

2.1 Capture de la CAO de Bolt (sur eDrawing Viewer) 8
2.2 Un robot Solo 12 (à gauche) et un Solo 8 (à droite) 9
2.3 Imprimante 3D Ultimaker S5 Pro Bundle et ensemble des coques en CAO . . 10
2.4 Impression de pièces en Tough-PLA (rouge) et en PVA (nacré) 11
2.5 Pièces issues de l'imprimante 3D, avec un moteur assemblé 11
2.6 Pré-visualisation d'une impression sur Ultimaker Cura 12
2.7 Support du robot 14
2.8 Helicoils pour des vis de diamètre 2,5 mm et 3 mm 14
2.9 Outils de pose des helicoils 15
2.10 Outils de pose des roulements. De haut en bas : presse manuelle, graisse, fraises et roulements 16
2.11 Actionneur ouvert du Solo 8 16
2.12 Montage d'un actionneur de hanche du Bolt et CAO de la même pièce . . . 18
2.13 Montage de test des actionneurs 18
2.14 Partie basse des jambes 19
2.15 Jambe droite en CAO et assemblée. Réglet de 30cm (pour l'échelle) 20
2.16 Capture de la CAO du tronc de Bolt 20
2.17 Schéma général de l'électronique et câblage du robot Bolt 21
2.18 Mise en évidence de la miniaturisation du micro-driver utilisé (en bas) 21
2.19 Masterboard fixée au robot avec les nappes de communication avec les micro-drivers et l'alimentation 22
2.20 Schéma de connexion aux micro-drivers 24
2.21 Schéma de câblage de l'IMU 25

3.1 Bolt en position de calibrage 32
3.2 Tests de fonctionnement des codes du Max Planck Institute 33
3.3 Schéma bloc de l'utilisation de ros2_control pour TALOS 36
3.4 Schéma bloc de l'utilisation de ros2_control pour Bolt 39
3.5 Nouvelles versions des pieds de Bolt (travail d'un autre stagiaire) 42

Table des abréviations

ABS acrylonitrile butadiène styrène
API interfaces de programmation d'applications
CAO Conception Assistée par Ordinateur
CLK Clock
CNRS Centre National de la Recherche Scientifique
CS Chip Select
DDL degrés de liberté
FDM Fused Deposition Modeling
GND la masse
IMU centrale inertielle
INS2I Institut National des Sciences de l'Information et de leurs Interactions
INSIS Institut National des Sciences de l'Ingénierie et des Systèmes
LAAS Laboratoire d'Analyse et d'Architecture des Systèmes
MISO Master In Slave Out
MJM Multijet Modeling
MOSI Master Out Slave In
MPI Max Planck Institute
ODRI Open Dynamic Robot Initiative
OS systèmes d'exploitation
PD correcteur proportionnel dérivé
PLA acide polylactique
PVA alcool polyvinylique
ROS Robot Operating System
SDK kit de développement logiciel
SLA Photopolymérisation Stéréolithographie
SPI Serial Peripheral Interface
STI inserts de filetage
UPR Unité Propre de Recherche

Introduction

Contexte

L'équipe Gepetto du LAAS-CNRS travaille sur l'analyse et la génération de mouvements des systèmes anthropomorphes, et est reconnue pour ses travaux dans la robotique humanoïde. Pour cela, elle utilise des simulateurs de robot sur ordinateur, et possède aussi de vrais systèmes robotisés tels qu'un TIAGo ou un TALOS [1] de chez PAL Robotics, ou encore des quadripèdes open-source Solo [2]. Malgré des systèmes différents, ce sont les mêmes logiciels qui sont utilisés pour les commander, et ce sur toutes les plate-formes du laboratoire.

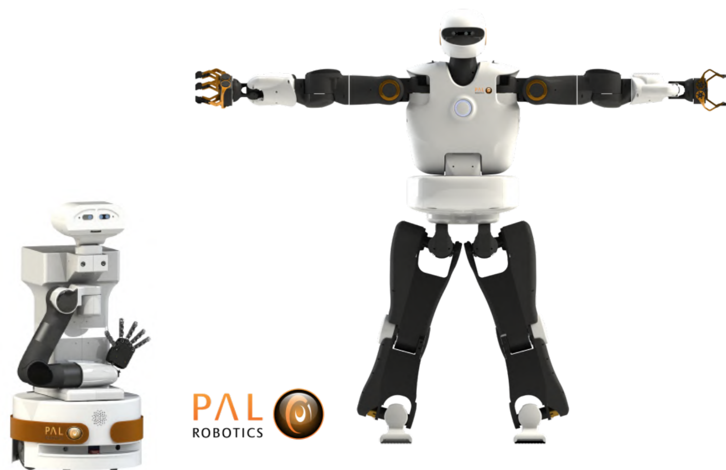


FIGURE 1 – Robots TIAGo (à gauche) et TALOS (à droite) de chez PAL Robotics

Afin de faire exécuter des tâches particulières à ces systèmes, les chercheurs élaborent des programmes, puis les testent sur simulateur avant de les tester sur les robots. Cependant, dans certains cas, ce schéma n'est pas réalisable et les actions souhaitées ne sont pas fonctionnelles sur le système réel. C'est le cas du robot TALOS qui possède une forme humanoïde. Il est d'une grande complexité physique notamment à cause de sa forte masse. Celle-ci se traduit par des équations différentielles non-linéaires complexes qui pourraient être évitées avec un modèle idéalisé, c'est-à-dire un robot plus léger et plus simple mécaniquement. Utiliser un modèle simplifié afin de résoudre des problèmes avant de passer à un système plus complexe est un processus très souvent utilisé en ingénierie, notamment en informatique, et que l'on

appelle "Divide-and-conquer" ("Diviser pour régner" en français).

De plus, la mise en œuvre sur TALOS est plus complexe car le robot étant plus puissant, il s'avère aussi plus dangereux : il est facile de le casser en cas d'erreur. Avoir des systèmes où la mise en œuvre est plus simple serait un vrai plus pour les étudiants (doctorants et futurs stagiaires). Le robot Bolt a cette vocation.

Objectifs du stage

L'objet de mon stage au sein de l'équipe Gepetto est basé sur la réalisation matérielle d'un robot bipède de type Bolt, et sur la réalisation d'une partie software assurant la communication bas-niveau en temps réel avec le robot.

Ce modèle de robot bipède léger, avec de faibles facteurs de réduction, et d'autres caractéristiques détaillées plus tard, peut être vu comme un système idéalisé des jambes du robot TALOS. Une fois fabriqué et piloté informatiquement, ce robot sera un intermédiaire de test entre la partie simulateur sur ordinateur et le robot TALOS. Les algorithmes pourront être testés sur un système à équations différentielles linéaires (Bolt) avant de passer sur un système à équations différentielles non-linéaires (TALOS). Cette étape intermédiaire facilitera les tests aux chercheurs en ayant plus d'éléments de réponse sur leurs expériences non-fructueuses, et notamment en infirmant ou confirmant des hypothèses. Vous trouverez en références des articles liés à des tests d'algorithmes de commande prédictive [3], de marche bipédique [4] [5], de modélisation d'objets 3D [6], ou de modélisation de l'environnement [7].

Ceci fait partie d'une démarche scientifique qui est celle d'un chercheur ou d'un ingénieur. La réalisation de ce stage m'a permis de m'y confronter mais aussi de créer un système qui vise à faciliter les avancées des autres membres de l'équipe.

Pour illustrer cela, vous trouverez en figure 2 le schéma bloc du fonctionnement du robot TALOS.

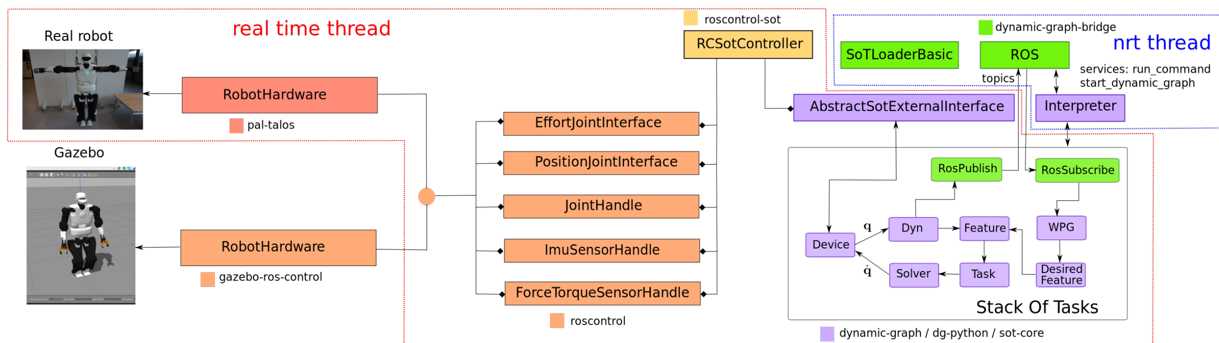


FIGURE 2 – Schéma bloc du fonctionnement de TALOS

De nombreux membres de l'équipe travaillent sur les blocs de droite (Dyn et WPG notamment) et cette partie est donc utilisée selon diverses formes. Nous n'allons pas rentrer dans les détails, vous pouvez retrouver leurs travaux en référence [8] [9]. TALOS étant un robot difficile à utiliser au niveau expérimental, l'équipe souhaite pouvoir utiliser un système robotisé plus simple, sans devoir tout changer. Ainsi le but de ce stage est de mettre en place ce nouveau système et son interfaçage informatique.

On peut ainsi segmenter mon travail selon trois axes :

1. **Assemblage physique du robot (partie 2),**
2. **Compréhension et test des codes existants (parties 3.1 et 3.2),**
3. **Adaptation à un paquet en cours de développement : ros2_control (partie 3.3).**

L'organisation temporelle de chaque partie n'a pas été fixée dès le début du stage compte tenu de l'utilisation de la méthode Agile : planification adaptative, flexibilité au changement et donc pas de cheminement figé. Les aléas dûs à la COVID-19 n'ont pas rendu le cheminement linéaire. Certaines phases, notamment de construction, ont dû être décalées à cause du télétravail. En fin de stage, on peut dire que chacun des trois axes a été réalisé en deux mois.

1. Présentation de l'organisme d'accueil

1.1 Le LAAS-CNRS

Le Laboratoire d'Analyse et d'Architecture des Systèmes (LAAS) est une Unité Propre de Recherche (UPR) du Centre National de la Recherche Scientifique (CNRS) créée en 1968 et située à Toulouse. Conventionnellement nommé LAAS-CNRS, nous utiliserons seulement l'acronyme LAAS dans le corps de ce rapport. Le laboratoire est rattaché à l'Institut National des Sciences de l'Ingénierie et des Systèmes (INSIS) et à l'Institut National des Sciences de l'Information et de leurs Interactions (INS2I).

Figure 1.1 – Logo du LAAS-CNRS

Le LAAS mène des recherches qui visent à une compréhension fondamentale des systèmes complexes tout en considérant l'usage qui peut en découler. Les chercheurs sont donc à la fois défricheurs de problématiques nouvelles et promoteurs de solutions, et ce autour de quatre grandes disciplines : l'informatique, la robotique, l'automatique et les micro et nano systèmes. De ces disciplines ont découlé quatre axes stratégiques : l'intelligence ambiante, le vivant, l'énergie et l'espace.

Figure 1.2 – Photo du bâtiment d'accueil du LAAS-CNRS

Au 1er janvier 2021, le LAAS était constitué de 580 personnes (hors stagiaires), de 6 départements scientifiques et de 25 équipes de recherche (cf Organigrammes en Annexe A).

1.2 L'équipe Gepetto

L'équipe Gepetto, créée en 2006, fait partie du département Robotique du LAAS. Son activité de recherche est centrée sur l'analyse et la génération de mouvements des systèmes anthropomorphes. C'est une des équipes phares en robotique humanoïde, internationalement reconnue pour son expertise en génération de mouvements.

Figure 1.3 – Logo de l'équipe Gepetto

Les deux objets de recherche de Gepetto sont les robots humanoïdes et l'Homme. Grâce à des études interdisciplinaires, l'équipe travaille sur divers problèmes de commande de ces systèmes anthropomorphes. Trois niveaux sont à distinguer au sein de l'équipe :

- Le niveau fondamental qui fournit un travail théorique à propos de la modélisation et de la génération de mouvements. La modélisation inclut la mécanique des robots, des mathématiques sur de nouvelles représentations et opérateurs, ou encore l'enregistrement et la représentation du mouvement humain. La génération de mouvements part de la planification globale de trajectoires jusqu'à la commande locale du mouvement. Cela est réalisé sous des contraintes de différents types. Ce niveau se base sur des disciplines diverses telles que les mathématiques, la mécanique, l'automatique, ou l'informatique ; mais également la biomécanique ou les neurosciences du mouvement, en collaboration avec des spécialistes des sciences du vivant.
- Le niveau intégratif constitue le cœur du travail de l'équipe. Il s'agit de l'intégration logicielle des développements théoriques. Les paquets logiciels avancés issus de cette intégration sont au maximum mis à jour et rendus accessibles à l'ensemble de la communauté scientifique.

- Le niveau applicatif concerne les contributions faites auprès de la robotique de service ou industrielle. Diverses applications y sont inclues comme l'usine du futur, l'animation graphique, la conception de nouveaux actionneurs, la compréhension et l'imitation du mouvement humain (entre autres).

Lors de mon arrivée, l'équipe Gepetto se composait de 31 personnes allant des chercheurs aux doctorants. Le responsable de l'équipe est Olivier Stasse, qui est aussi mon tuteur de stage. Une petite douzaine de stagiaires s'y sont intégrés au cours de mes 6 mois de stage.

2. Hardware

2.1 État de l'art

Le choix du robot Bolt a été fait par l'équipe Gepetto, et m'a été proposé avant mon arrivée dans l'équipe. Ils ont suivi les conseils de Felix Grimminger, ancien ingénieur de Boston Dynamics ayant participé à la conception de BigDog. La remise en question de celui-ci n'était donc pas envisagée, c'est pour cette raison que je n'ai pas fait de dimensionnement, ni de conception électronique. Nous allons maintenant détailler les nombreux avantages qui font de lui le candidat idéal.

Comme expliqué dans l'introduction, l'équipe possède déjà des robots Solo qui sont des quadripèdes issus de la même famille que Bolt (c'est-à-dire que c'est la même équipe de chercheurs/ingénieurs qui les ont conçus en gardant une architecture similaire entre eux). Cependant les Solo ne sont pas bipédaux et ne peuvent donc pas être utilisés pour modéliser un robot bipède tel que TALOS. Malgré tout, les robots développés par Felix Grimminger et al [2] [10] [11] sont commandés en couple, et possèdent d'énormes avantages :

- Ils sont open-sources, c'est-à-dire qu'ils sont facilement reproductibles.
- Ils sont reproduisibles à faible coût de fabrication, environ 10 000 €. Cela facilite leur développement par différents laboratoires à travers le monde afin de réaliser un travail collaboratif sur les mêmes systèmes. De plus, ce critère permet de moins craindre la casse matérielle car un remplacement des pièces est peu onéreux.
- Ils sont légers (1,3 kg pour Bolt) car conçus avec une coque plastique imprimée en 3D, de petits éléments de transmission souvent imprimés en 3D aussi, et des moteurs légers mais assez puissants pour l'utilisation souhaitée.
- Ils sont robustes aux chocs physiques extérieurs ou à toute autre perturbation extérieure sur le système.
- Ils sont commandés en couple ce qui offre comme avantage de mieux prendre en compte les interactions en force non prévues.
- Ils ont un faible rapport de réduction : il est de 1/9 par moteur. Cela permet des pointes de couple raisonnables ainsi qu'une vitesse élevée au niveau des articulations.
- Ils ont un rapport poids/puissance faible car le centre de masse du robot est autour du bassin.

- Ils ont une conception qui permet d'avoir une bonne réversibilité. Cela est dû aux faibles rapports de réduction et à des frottements faibles qui permettent d'avoir une bonne corrélation entre couple en sortie de l'actionneur et courant moteur.

Ainsi, en tenant compte de tous ces avantages, on peut considérer le robot Bolt comme un robot bipède idéal d'un point de vue physique. Lorsqu'on réalise les équations physiques du modèle, tous les points cités permettent d'obtenir un modèle plus simple à gérer, car plus proche d'un modèle idéal. Cela améliore grandement les conditions pour réaliser des tests d'algorithme avant de passer sur un système plus complexe. Il vérifie donc les critères pour en faire un intermédiaire de test.

D'un point de vue conception, la totalité des éléments et des étapes de fabrication réalisés par le groupement Open Dynamic Robot Initiative (ODRI), dont le Max Planck Institute (MPI), sont disponibles en open-source sur GitHub [11]. Un aperçu du robot Bolt, issu du GitHub, et tiré d'un logiciel de Conception Assistée par Ordinateur (CAO) est disponible dans la figure 2.1.

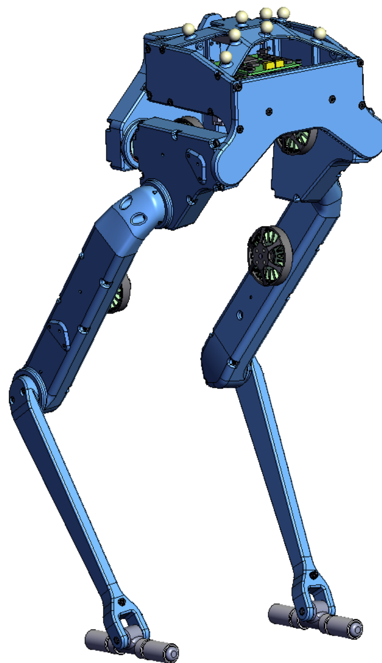


FIGURE 2.1 – Capture de la CAO de Bolt (sur eDrawing Viewer)

2.2 Étapes de montage

La totalité des éléments présents dans cette partie a donné lieu à la rédaction d'un document de notice qui explicite toute la fabrication hardware. Il sera très utile dans le futur pour l'équipe Gepetto qui pourra s'y référer en cas de casse d'une pièce, ou de tout autre besoin. En plus de toutes les étapes de montage, il donne les liens utiles, la liste des pièces et des fournisseurs, les techniques de base et les problèmes possibles lors de la réalisation des étapes. Nous allons détailler les étapes de montage en se référant à chaque fois à des éléments de la figure 2.1.

2.2.1 Recyclage d'un robot précédent - Solo 8

L'équipe Gepetto avait déjà travaillé sur des quadripèdes de la même famille que le bipède Bolt : il y a le Solo 8 et le Solo 12 (cf figure 2.2).

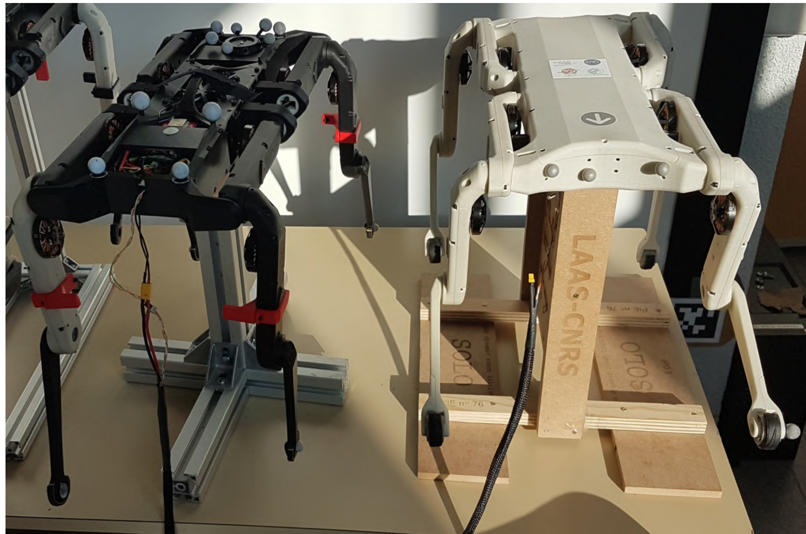


FIGURE 2.2 – Un robot Solo 12 (à gauche) et un Solo 8 (à droite)

Les nombres associés aux robots représentent le nombre de degrés de liberté (DDL) qu'ils possèdent. Les Solo 12 possèdent donc quatre DDL de plus que le Solo 8, représentant les rotations latérales des jambes (roulis), assimilables à des mouvements de hanche. Le Solo 8 ne possède pas ces mobilités là, et n'a donc que des rotations de tangage. Il ne peut donc que sauter ou avancer tout droit, sur des terrains peu accidentés.

L'équipe Gepetto a pu utiliser au mieux le Solo 8 jusqu'à la conception du Solo 12 qui offre plus de possibilités. Ainsi Solo 8 n'étant plus utilisé, il a été décidé de le recycler afin d'utiliser ses éléments internes pour le Bolt.

2.2.2 Fabrication des pièces

Dans la construction du robot, certains éléments sont à commander dans des magasins spécialisés, et d'autres doivent être fabriqués sur mesure. Nous allons parler de cette dernière catégorie dans cette partie.

Impression 3D

Malgré le recyclage de nombreuses pièces du Solo 8, les coques des membres sont différentes, et doivent donc être fabriquées. Pour cela, j'ai utilisé la méthode de l'impression 3D.

L'équipe Gepetto possède une imprimante 3D utilisant la méthode du dépôt de fil fondu (ou Fused Deposition Modeling (FDM) en anglais). Le but de cette méthode est de déposer la matière de la pièce par couche. Des matières polymères sont fondues puis extrudées pour former des couches. Ce dépôt de matière est réalisé par une buse d'extrusion.

L'imprimante 3D sera utile pour les grosses pièces du robot. Celle dont je me suis servi est une Ultimaker S5 Pro Bundle (cf figure 2.3).

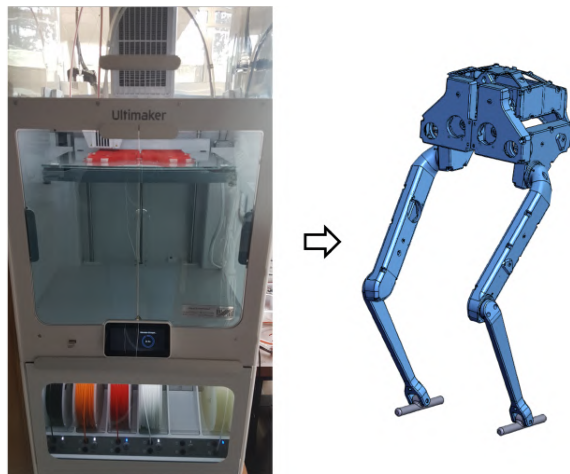


Figure 2.3 – Imprimante 3D Ultimaker S5 Pro Bundle et ensemble des coques en CAO

Ce modèle d'imprimante possède un plateau d'impression se déplaçant verticalement, et deux buses d'extrusion se déplaçant selon les deux autres dimensions. Le fait d'avoir deux buses permet le dépôt de deux matières différentes lors d'une même impression. Avant d'expliquer pourquoi cela a été très pratique, nous devons parler des matières utilisées.

Comme expliqué précédemment, l'imprimante 3D utilise des matières polymères. Il en existe différentes sortes, mais nous allons ici parler des deux dont je me suis servi lors de mon stage.

La première matière est le Tough-PLA. Celle-ci est un dérivé de l'acide polylactique (PLA) mais dont la dureté est similaire à celle de l'acrylonitrile butadiène styrène (ABS).

Nous n'allons pas rentrer dans les détails physiques ou chimiques de ces deux polymères. Il est toutefois nécessaire de savoir que le PLA est le principal matériau utilisé en impression 3D, tout comme l'ABS qui possède cependant une meilleure tenue au choc et une rigidité supérieure au PLA. Ainsi le Tough-PLA est un bon entre-deux, c'est pour cela que nous le choisissons pour constituer les pièces de la coque du Bolt. Par la suite, ces pièces seront les rouges et noires sur les illustrations.

La seconde matière est le Natural PVA. Elle est composée d'alcool polyvinylique (PVA) qui est un polymère solide dans un milieu sec et qui est soluble dans l'eau. De par cette propriété, nous l'utilisons pour créer le support de la pièce à imprimer. Ainsi son utilisation permet l'impression de pièces complexes et une récupération plus facile de celles-ci après un passage dans un bain d'eau. Par la suite, le PVA apparaîtra de couleur nacré sur certaines illustrations.

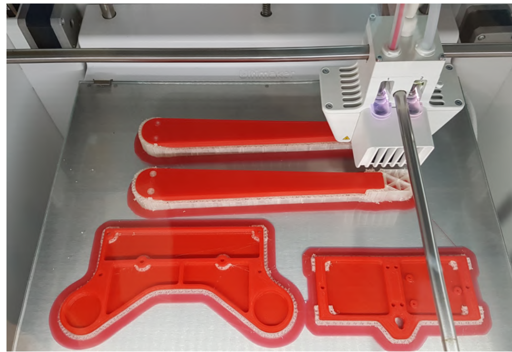


FIGURE 2.4 – Impression de pièces en Tough-PLA (rouge) et en PVA (nacré)

Grâce à ce processus, j'ai imprimé l'ensemble des pièces composant la coque du Bolt avec l'Ultimaker. Pour rappel, les plans de toutes les pièces sont sur le GitHub [11] en format STL.

FIGURE 2.5 – Pièces issues de l'imprimante 3D, avec un moteur assemblé

D'un point de vue logiciel, c'est "Ultimaker Cura" qui est associé a l'imprimante de l'équipe Gepetto. Il s'agit d'un planificateur de mouvements de l'imprimante. Celui-ci prend

en entrée les dessins en CAO qui sont des fichiers ".stl". Ils s'enregistrent comme projet Cura en ".3mf" et il en sort des projets en ".ufp" lisibles et interprétables par l'imprimante.

C'est aussi sur ce logiciel que sont spécifiés les paramètres propres à l'impression des pièces. Nous allons uniquement détailler ceux dont je me suis servi :

- La sélection des matières à utiliser par chacune des buses d'extrusion.
- Horizontal Expansion (en mm) : ce paramètre permet d'ajuster la distance entre deux couches consécutives, donc joue sur l'écrasement de la matière, et par conséquent sur la précision finale de l'impression. Des pièces de test avaient été imprimées en amont de mon stage, ce qui a permis de déterminer qu'une réduction de -0.07mm était le meilleur paramétrage pour nos impressions. Les premières pièces ont été imprimées avec -0.10mm et ainsi ont du être retravaillée afin que des pièces mécaniques y rentrent en force et sans colle. C'est le fait qu'on ait appliqué une réduction qui justifie l'emploi de distances négatives.
- Infill : il s'agit du pourcentage de remplissage des parties pleines des pièces. Ainsi afin d'alléger les structures, certaines parties ne sont pas obligatoirement pleines de matière mais peuvent être partiellement creuses, selon une géométrie précise.
- La température des buses d'extrusion et du plateau.
- La présence (ou non) d'une matière de support.
- L'orientation 3D des pièces à imprimer.

De plus, différents menus de pré-visualisation sont disponibles. Lors de l'enregistrement du projet, le logiciel calcule une estimation du temps d'impression et de la quantité de matière nécessaire. Enfin, le projet est enregistré sur une clé USB qui sera mise dans l'imprimante afin de le récupérer et de lancer l'impression.

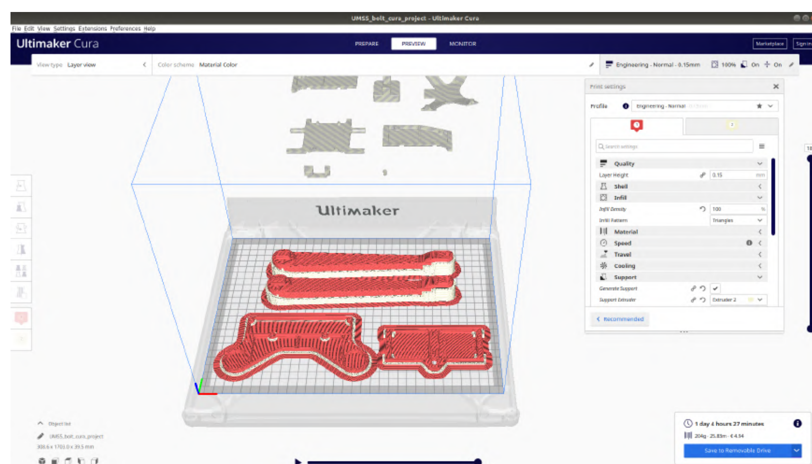


FIGURE 2.6 – Pré-visualisation d'une impression sur Ultimaker Cura

D'autres pièces, plus petites, nécessitent un niveau de précision plus élevé. Pour cela, deux autres méthodes sont envisageables : le Multijet Modeling (MJM) ou la Photopolymérisation Stéréolithographie (SLA) :

- Le MJM est un procédé de fabrication additive utilisant une technologie proche d'une imprimante 2D à jet d'encre, mais avec des centaines de buses de jets. Les gouttelettes de matière, résine polymère ou cire, sont de quelques microns ce qui permet des épaisseurs de couches de 16 microns.
- La SLA est un procédé développé dans les années 1980 reposant sur les propriétés qu'ont certaines résines à se polymériser sous l'effet de la lumière et de la chaleur. Il nécessite une cuve de résine liquide et un laser fixe qui balaie la résine selon la forme désirée. Sous l'effet du laser, il y a polymérisation de la résine, et cela est fait tranche par tranche afin d'obtenir un objet 3D par superposition. Le reste de matière est ensuite dissous.

Dans notre cas, le LAAS n'ayant pas le matériel nécessaire pour utiliser ces deux méthodes, la fabrication des pièces est faite par l'institut Max Planck, qui les envoie au LAAS par la suite. Dans mon cas, j'ai réutilisé celles du Solo 8.

Usinage de pièces métalliques

Certaines pièces de transmission sont métalliques et nécessitent de l'usinage pour leur fabrication. C'est le cas pour les axes de sortie des moteurs et pour des poulies de transmission : une sur l'axe moteur et une entre deux étages de réduction. Des plans d'usinage PDF, précis à la dizaine de microns prêt, sont fournis sur GitHub [11]. Ainsi, selon l'outillage nécessaire, elles peuvent être faites par le LAAS ou commandées à une entreprise spécifique.

Pour l'axe de sortie des moteurs, la matière utilisée est de l'acier inoxydable commandé auprès d'une compagnie allemande. Elle est fournie par l'institut Max Planck mais elle est usinée par le tourneur-fraiseur du LAAS. Ainsi les pièces ont été réalisées pour le robot Solo 8 en amont de mon stage, et j'ai pu les réutiliser.

Pour les poulies, la matière utilisée est l'aluminium et l'usinage est fait avec une fraise particulière. Ensuite, il faut faire appel à une entreprise spécialisée qui les fabrique avec les plans et la fraise spéciale. C'est ce qui a été fait pour le Solo 8 et donc j'ai réutilisé les pièces déjà existantes.

La seule partie d'usinage de pièce métallique que j'ai réalisée a été la conception d'un portique de support pour le robot. Celui-ci est composé de barres en aluminium qui ont été découpées à l'atelier d'usinage du LAAS, et qui sont fixées entre elles grâce à des équerres. Le seul modèle que j'avais, était une photo sur le GitHub [11]. J'ai donc évalué les dimensions de chaque élément du portique afin qu'il puisse porter le robot sans lui faire toucher le sol, sans que ses jambes puissent toucher le portique et sans qu'il puisse tomber sous l'effet des

mouvements du robot. Une pièce imprimée en 3D et maintenue par une vis permet de lier le portique au robot.

FIGURE 2.7 – Support du robot

Le portique sera très utile car il permettra de stocker le robot sans le laisser au sol (lieu poussiéreux à éviter pour les moteurs) et de réaliser des tests hors-sol sur le robot.

2.2.3 Mécanique

Usinage des pièces imprimées en 3D

La partie mécanique a commencé avec la fin de l'usinage des pièces imprimées en 3D. La première étape a été de s'occuper des coques.

Les coques sont des pièces maintenues fermées grâce à des vis, qui seront amenées à être ré-ouvertes plusieurs fois. Ainsi pour les ouvrir et les fermer sans abîmer la coque en plastique, on utilise des pas de vis métalliques qui s'appellent des inserts de filetage (STI), ou helicoils. Il s'agit de petites pièces ayant la forme d'une bobine de fil se finissant d'un côté par une partie horizontale au milieu de l'enroulement (cf figure 2.8).

FIGURE 2.8 – Helicoils pour des vis de diamètre 2,5 mm et 3 mm

Pour poser les helicoils, on utilise des outils spéciaux et uniques selon le diamètre de ceux-ci. Ils sont en photo sur la figure 2.9. On a de haut en bas et de gauche à droite : helicoils, mèche, taraud, fil dénudé, insert fileté, mandrin, tenon et second taraud.

FIGURE 2.9 – Outils de pose des helicoils

La première phase pour poser les helicoils consiste à ajuster la taille des trous dans les pièces 3D. Pour cela, on utilise un mandrin et une mèche, et on s'assure d'être bien perpendiculaire à la pièce. Ensuite on ré-utilise le mandrin mais cette fois avec un taraud. Le taraudage du trou se fait aussi de façon perpendiculaire à la pièce. Il permet de donner au trou une première forme filetée dans laquelle va s'insérer l'helicoil. Puis on utilise un insert fileté afin de mettre l'helicoil à sa position finale. L'insert fonctionne comme une vis avec un embout qui permet d'accrocher le bout de l'helicoil. Ainsi il peut être vissé mais jamais dévissé. Il est en place lorsqu'il se situe à l'intérieur de la pièce, à fleur du début du trou. Enfin, la dernière étape consiste à couper le bout de l'helicoil. Cela est fait grâce à un découpe tenon. Il faut faire attention à bien récupérer et jeter le bout coupé car comme il est en métal, il peut s'aimanter au rotor d'un moteur et très fortement l'endommager. En cas de difficulté pour le récupérer, notamment pour des trous non traversants, on utilise un fil dénudé pour le débloquer. Certaines vis sont directement rentrées dans la pièce 3D, sans helicoil. Ce sont des vis qui n'ont pas pour vocation d'être souvent enlevées et remises à leur place. Un taraudage des trous est tout de même nécessaire.

La seconde phase d'usinage est la pose des roulements. Ceux-ci sont rentrés en force dans la pièce 3D. Un ajustement du diamètre de leur trou d'accueil peut être nécessaire selon le paramètre d'impression choisi. Pour cela on utilise des fraises de la taille du diamètre extérieur du roulement. Ensuite, notamment pour des roulements de 8mm ou moins, on utilise une presse manuelle afin de les enfoncer au mieux dans la pièce 3D (cf partie 2.3.1 pour plus de détails).

L'ultime travail est de graisser les plus gros roulements, c'est-à-dire ceux assurant les jonctions entre les différents membres (bassin-hanches, hanches-cuisses, cuisses-tibias). Cela se fait en introduisant de la graisse sur les billes à l'aide d'une petite spatule, et de la faire

FIGURE 2.10 – Outils de pose des roulements. De haut en bas : presse manuelle, graisse, fraises et roulements

diffuser en le faisant rouler. Un roulement est bien graissé lorsqu'il ne fait plus de bruit lors de sa rotation.

Démontage et remontage des actionneurs

Pour la suite de l'assemblage de Bolt, la majeure partie des éléments sont issus du robot Solo 8. Il faut donc les démonter puis les remonter sur Bolt. Cela est valable pour les moteurs et l'électronique.

Commençons par les moteurs. Quelle que soit la configuration ou la forme du membre à mettre en action, les éléments sont exactement les mêmes. Ainsi on peut utiliser les moteurs, capteurs et éléments de transmission de Solo 8 pour les hanches ou les cuisses de Bolt même s'il n'ont pas la même forme (membres plus long ou moins longitudinaux). Une fois que l'on sait cela, on peut détacher un membre du Solo 8 et l'ouvrir pour observer ce qu'il y a à l'intérieur.

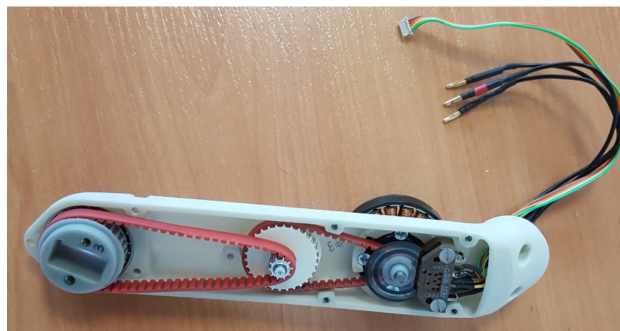


FIGURE 2.11 – Actionneur ouvert du Solo 8

Sur la figure 2.11, on trouve de gauche à droite :

- Deux étages de réduction constitués de poulies dentées imprimées en 3D ou usinées dans du métal. Des roulements assurent la rotation entre ces pieds et la coque. Il y a aussi deux courroies dentées. Ce n'est pas visible sur la photo mais la courroie de gauche est tendue grâce à des tendeurs de courroie (rondelles imprimées en 3D sur des roulements) une fois l'actionneur fermé. Chaque étage a un facteur de réduction de 1/3, soit un gain global de 1/9.
- Un moteur brushless triphasé de KV = 300 avec un axe modifié pour y mettre une poulie et fixer une roue codeuse. Le moteur est hors de la coque, tandis que la partie capteur est dedans. Un roulement assure la connexion entre l'axe et la coque. La roue codeuse a une précision de 5000 pulsations par tour.
- Un capteur optique : il s'agit d'un encodeur incrémental en quadrature à trois canaux de sortie. Il est monté avec la roue codeuse à l'intérieur afin de récupérer les informations de rotation du moteur.
- Les fils des trois phases du moteur et la nappe de connexion du codeur avec respectivement leurs connecteurs et boîtiers de connexion.

Chaque élément doit être dévissé et retiré avec minutie pour ne pas abîmer ou décoller la roue codeuse qui est fragile. On commence par retirer l'encodeur, puis la courroie présente sous la roue codeuse. Ainsi, les autres éléments de réduction peuvent être retirés sans difficulté. Ensuite, il faut séparer l'axe moteur du rotor et du stator. Cette étape n'est pas simple car ces éléments sont usinés pour être parfaitement assemblés. Pour faire cela, il faut retirer deux vis situées dans le rotor et qui le fixe à l'axe du moteur, puis sortir le rotor de l'axe. On peut alors sortir l'axe (et les éléments dessus) du stator. Enfin, on dévisse le stator de la pièce 3D.

Les étapes du remontage des actionneurs sur les coques du robot Bolt sont exactement les mêmes que celles détaillées ci-dessus, mais réalisées dans le sens inverse.

Enfin il faut fermer la coque de l'actionneur. Cette étape n'est pas facile car il faut que les différents axes et poulies rentrent dans les roulements du couvercle. De plus, comme on peut le voir sur la figure 2.12, il y a des tendeurs de courroie fixés au couvercle (les pièces blanches). Ils permettent d'augmenter le nombre de dents de la poulie en contact avec la courroie. Il faut donc bien aligner le couvercle et grâce à un petit tournevis, faire rentrer la courroie entre les deux tendeurs. On ferme le tout avec des vis et l'actionneur est monté.

L'ultime étape est une phase de test de l'actionneur. Celui-ci est connecté à des microcontrôleurs identiques à ceux qui le commanderont une fois assemblé. Ces microcontrôleurs sont reliés à un ordinateur sur lequel se trouve un code de test.

Une fois les branchements des phases et de l'encodeur réalisés sur le micro-driver (carte verte), on vérifie le comportement de l'actionneur. Pour cela, on place la pièce de tibia qui

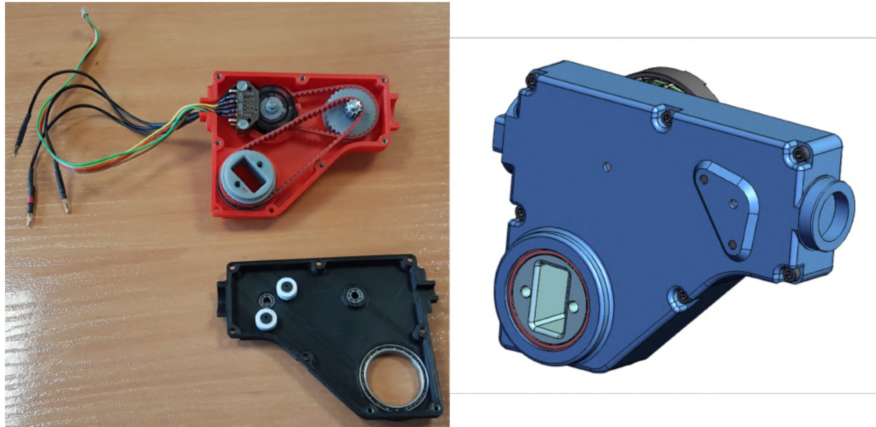


FIGURE 2.12 – Montage d'un actionneur de hanche du Bolt et CAO de la même pièce

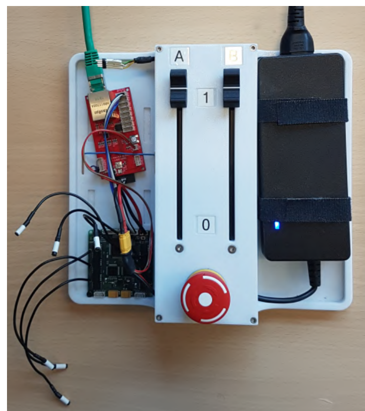


FIGURE 2.13 – Montage de test des actionneurs

est non motorisée en sortie et on l'écarte de sa position d'origine. Le bon comportement du moteur doit être un rejet de la perturbation, c'est-à-dire que le tibia doit retourner à sa position d'origine.

En rentrant dans les détails du code de test, celui-ci commence par récupérer un nom de l'appareil connecté, puis impose des caractéristiques (fréquence et amplitude du sinus, gains proportionnel et dérivé). Il réalise ensuite une calibration des encodeurs et une initialisation en temps du sinus. Il y a alors une réalisation de la boucle de commande : calcul des positions et vitesses de référence, puis des positions et vitesses d'erreur et enfin couple à appliquer selon les valeurs d'erreur et les gains. Ces valeurs de couple sont ensuite envoyées à l'appareil. Cette boucle est réalisée à une fréquence de 1kHz. Une fois cette partie réalisée, l'actionneur est prêt à être monté sur le robot.

Montage final

Une fois que toutes les pièces sont imprimées et que tous les actionneurs sont montés, on peut réaliser le montage complet des jambes et du tronc du Bolt.

Expliquons le tout étape par étape en commençant par le bas. Le pied et le tibia sont deux pièces non motorisées qui sont reliées ensemble par une goupille cylindrique en acier de 5mm de diamètre. Il faut donc utiliser un mandrin et une mèche de 5mm pour retailler les trous, notamment celui du pied afin qu'il n'y ait pas de frottement de la goupille dessus. De plus, pour éviter les frottements entre le pied et le tibia, il faut limer le plastique du pied. Ensuite, on positionne la goupille avec un marteau. L'élément est volontairement passif, donc sous-actionné, pour gagner en légèreté. Pour cela, on utilise un élastique qui maintiendra l'angle entre le pied et le tibia. Un tube en plastique assure le contact entre le pied et le sol. En parallèle de mon stage, un autre stagiaire a travaillé sur la conception de nouveaux pieds passifs plus stables.

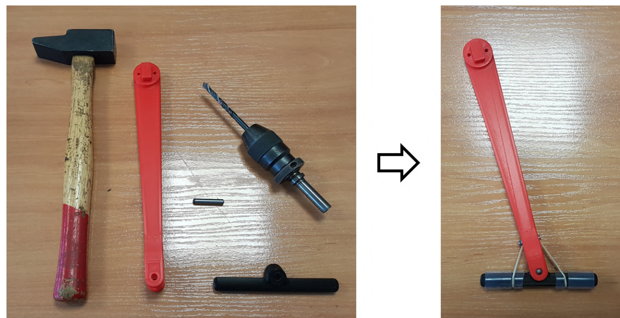


Figure 2.14 – Partie basse des jambes

Il faut ensuite assembler les trois actionneurs qui forment une jambe. Pour joindre la cuisse et la partie basse de la hanche, il faut faire passer les câbles de la cuisse dans la hanche. Il est donc nécessaire de ré-ouvrir l'actionneur de la hanche. L'ensemble s'attache sans difficulté à la partie haute de la hanche. La jambe est alors terminée (cf figure 2.15).

On peut alors assembler les deux jambes via le tronc (cf CAO figure 2.16) imprimé en 3D et qui sert de support pour l'électronique. Ainsi on peut considérer que le montage est terminé.

2.2.4 Électronique

La partie mécanique donne au robot son architecture et ses DDL. Cependant, pour piloter les mouvements, on a besoin d'une partie électronique. Les éléments qui composent cette partie sont fixés au sommet du Bolt (sur son tronc), sur des pièces imprimées en 3D. Pour Bolt, on peut diviser la partie électronique en cinq axes : les micro-drivers, la masterboard, le câblage, l'alimentation et la centrale inertielle (IMU). La figure 2.17 illustre le montage

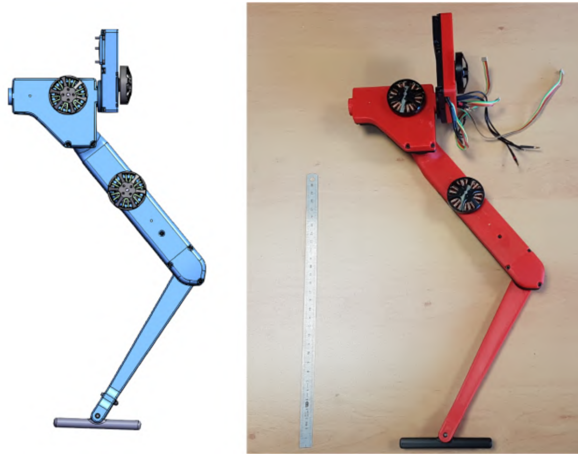


FIGURE 2.15 – Jambe droite en CAO et assemblée. Réglet de 30cm (pour l'échelle)

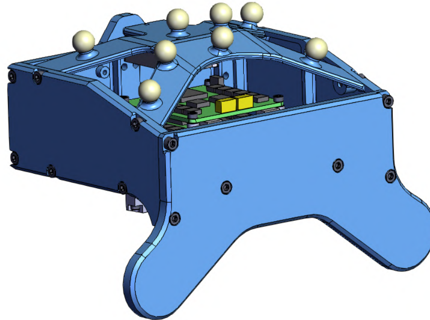


FIGURE 2.16 – Capture de la CAO du tronc de Bolt

électronique complet du Bolt.

Nous allons à chaque fois détailler leur rôle et expliquer comment les mettre en place.

Micro-drivers / électronique de puissance

Les micro-drivers sont des éléments d'électronique composés de microcontrôleurs qui permettent le contrôle en couple des moteurs. Une carte alimente et contrôle deux moteurs à 10kHz grâce à leurs câbles de phase, et échange des informations avec la masterboard à 1kHz via une liaison dite Serial Peripheral Interface (SPI). La version utilisée est une version miniaturisée d'une carte d'évaluation Texas Instruments. Elle fait 51mm x 50mm pour une masse de 13g. Elle est donc compacte et légère afin de s'intégrer au mieux au robot (cf figure 2.18). La fabrication de cette carte est faite au LAAS, grâce aux instructions situées sur GitHub [11]. Dans mon cas, j'ai réutilisé trois des micro-drivers du Solo 8.

Comme j'ai réutilisé des cartes, il a fallu mettre à jour le code qui se trouvait à l'intérieur. C'est ce que l'on appelle "flasher" les cartes. Cela doit être fait régulièrement afin

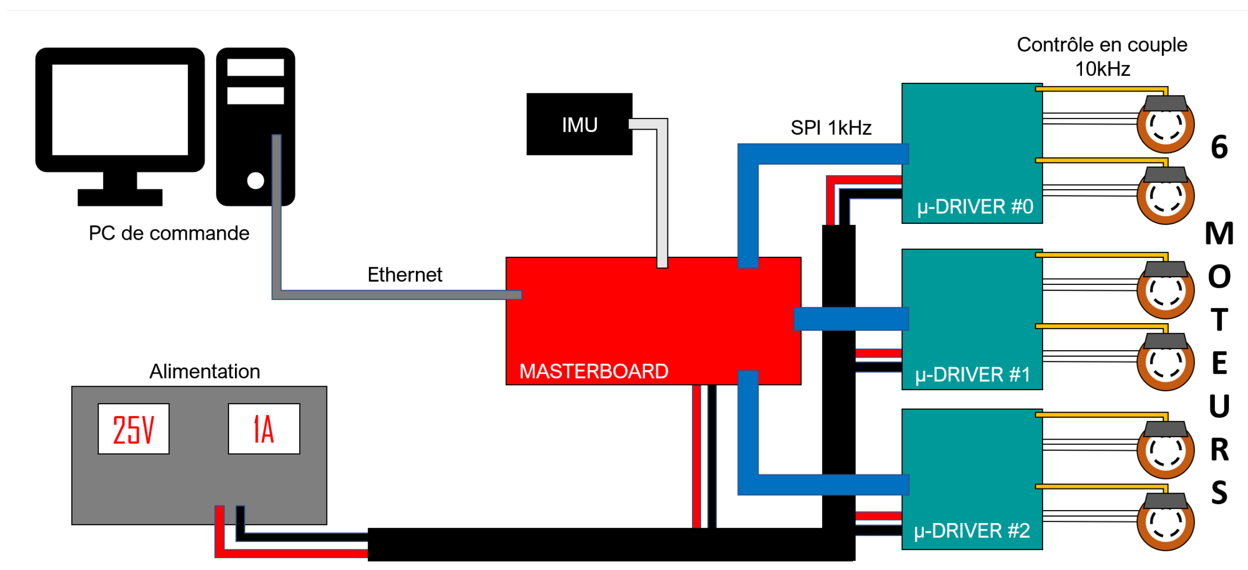


FIGURE 2.17 – Schéma général de l'électronique et câblage du robot Bolt

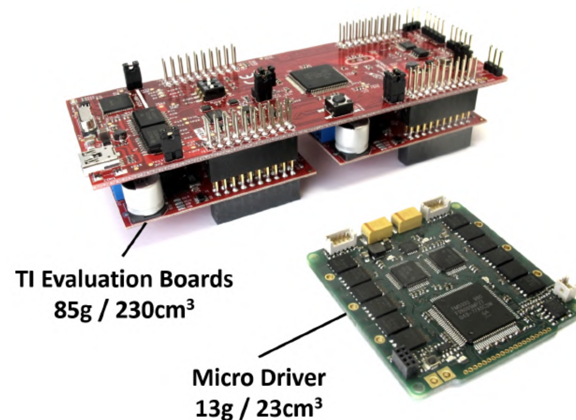


FIGURE 2.18 – Mise en évidence de la miniaturisation du micro-driver utilisé (en bas)

d'avoir la meilleure version du programme d'exécution. Pour les micro-drivers, le flash se fait grâce à un logiciel de Texas Instruments (Code Composer Studio), en y important un code disponible sur GitHub. On utilise un adaptateur spécifique pour connecter la carte à l'ordinateur.

Une fois cette opération réalisée sur chaque micro-driver, ils sont prêts à être utilisés.

Masterboard

La masterboard est aussi une carte électronique, composée d'un microcontrôleur ESP32. Elle centralise les données issues de l'IMU et des actionneurs (encodeurs et moteurs) via une

liaison SPI 1kHz avec les micro-drivers. Elle assure aussi une connexion en temps-réel avec l'ordinateur de commande via un câble Ethernet. Ses dimensions sont 61mm x 31mm pour une masse de 19g. Tout comme les micro-drivers, elle s'intègre au mieux au robot. Sa fabrication est aussi faite au LAAS mais j'ai réutilisé celle du Solo 8.

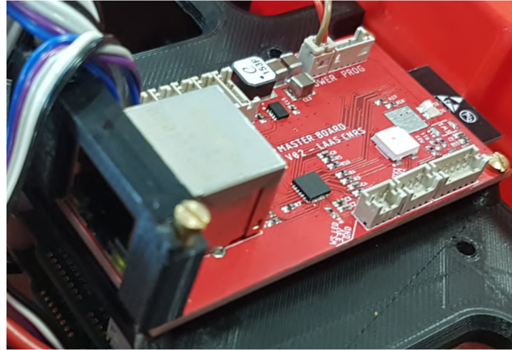


FIGURE 2.19 – Masterboard fixée au robot avec les nappes de communication avec les micro-drivers et l'alimentation

Tout comme pour les micro-drivers, il a fallu flasher la masterboard. Cette opération est réalisée sans logiciel particulier, grâce à un code disponible sur GitHub et via un adaptateur spécifique pour connecter la carte à l'ordinateur. La masterboard est prête à être utilisée.

Câblage

La partie connexion a un rôle prépondérant pour l'électronique car c'est elle qui assure la communication des données entre chaque bloc. Il y en a entre les actionneurs et les micro-drivers, entre les micro-drivers et la masterboard, et entre la masterboard et l'ordinateur de commande. Avant de détailler chaque niveau, il faut déterminer les longueurs de câble à utiliser pour chaque type (SPI et phases). Maintenant que toutes les parties du robot sont assemblées, on peut déterminer ces longueurs en faisant passer un fil entre les deux points de connexion, puis en le mesurant. Celles-ci sont aussi disponibles sur le GitHub. Il faut faire attention à prendre une marge de sécurité car des fils trop courts endommageront les éléments lors du fonctionnement du robot. Avoir des fils trop longs n'est pas non plus une solution car il peut y avoir des interférences entre eux (cf partie 2.3.3).

Commençons par détailler les deux connexions entre les actionneurs et les micro-drivers : pour le moteur et pour l'encodeur.

- Le moteur est connecté par l'intermédiaire de ses trois phases. La connexion s'établie en soudant, puis protégeant avec une gaine thermorétractable, des fils ayants des propriétés électriques adéquates. Ces fils se situent entre les câbles de phase du moteurs et les broches du micro-driver. Afin de pouvoir démonter facilement les éléments, on utilise des connecteurs de puissance mâle et femelle. Ils sont soudés au câble puis protégés par

une gaine thermorétractable. Cette connexion des phases est une sortie du micro-driver car c'est lui qui envoie les informations de commande (courant) au moteur.

- L'encodeur est connecté via une nappe de cinq câbles, chacun soudé à une de ses broches. Dans les cinq broches, il y a des éléments d'entrée pour alimenter l'encodeur en courant (5V et GND), mais aussi des éléments de sortie vers le micro-driver. Ce sont ces éléments qui permettent à la carte de connaître la position du moteur à une fréquence de 1 kHz. Pour le montage côté encodeur, il suffit de souder les câbles à l'étain sur les broches, tout en respectant un ordre précis. Côté micro-driver, on utilise des boîtiers de connecteur mâle de taille 5. Pour cela, il faut poser des contacts à sertir sur chaque fil à l'aide d'une pince à sertir. La manipulation n'est pas simple mais une fois les contacts posés, ils se clipsent facilement dans le boîtier et forment une connexion robuste. Il n'y a plus qu'à connecter la nappe de câble sur un boîtier femelle situé sur le micro-driver.

Détaillons maintenant la connexion entre les micro-drivers et la masterboard. Il s'agit d'une nappe de cinq câbles dont les deux extrémités sont des boîtiers mâles qui rentrent dans des boîtiers femelles sur les deux cartes. La procédure de montage est la même que décrite précédemment. Les cinq câbles correspondent à une liaison SPI (Serial Peripheral Interface). Il s'agit d'un bus de données série synchrone qui opère en full-duplex. Les communications s'opérent selon un schéma maître-suiveur. Dans notre cas, le maître est la masterboard, et les micro-drivers sont des suiveurs. Seulement quatre fils sont nécessaires pour une liaison SPI : Master Out Slave In (MOSI), Master In Slave Out (MISO), Clock (CLK) et Chip Select (CS). Pour nous, le 5ème est la masse (GND).

Concernant le montage final entre les éléments décrits précédemment, il suivra le schéma de la figure 2.20.

Enfin détaillons la connexion entre la masterboard et l'ordinateur de commande. C'est un câble Ethernet qui assure l'envoi et la réception des données en temps-réel par l'ordinateur. C'est sur celui-ci que sont exécutés les algorithmes de commande du robot et la seule communication qu'il a avec lui se fait par l'intermédiaire d'un câble Ethernet relié à la masterboard.

Il y a donc toujours un câble qui relie le robot à un ordinateur. Il n'est donc pas "sans-fil". Ce n'est d'ailleurs pas le seul câble, c'est ce que nous allons voir dans la partie alimentation.

Alimentation

L'alimentation en électricité est faite via deux câbles supportant jusqu'à 500V et reliés à chaque carte (micro-drivers et masterboard). Un transformateur permet une alimentation du robot en 24V. Pour ce qui est des connexions, elles sont différentes selon les cartes. Pour les micro-drivers, ce sont des fiches mâles-femelles pour batterie, une partie étant soudée aux emplacements d'alimentation sur la carte. Pour la masterboard, c'est un boîtier de connecteur femelle à deux contacts qui se plugge directement sur celle-ci. L'ensemble des câbles est relié à

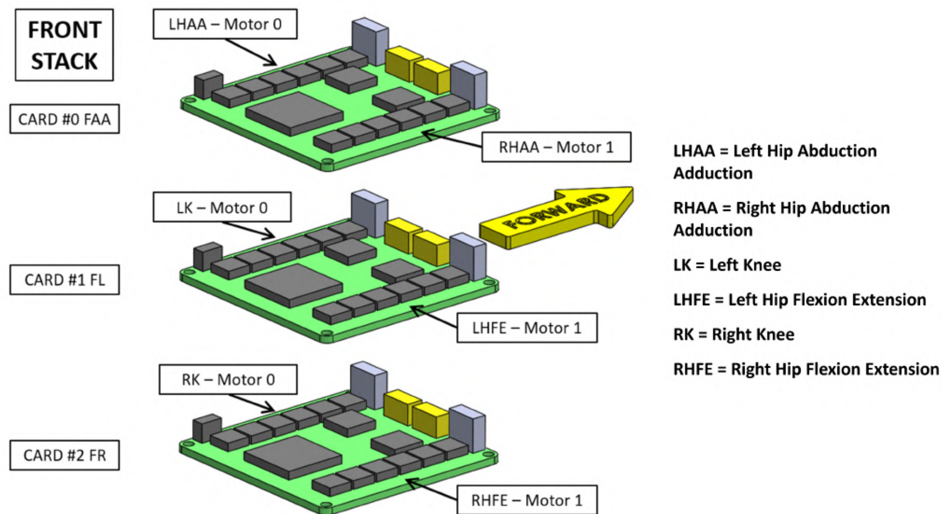


FIGURE 2.20 – Schéma de connexion aux micro-drivers

un câble principal, lui-même en série avec un bouton d'arrêt d'urgence et un transformateur. Avec le câble Ethernet, ce câble principal est toujours relié au robot.

Centrale inertielle (IMU)

En plus de tout ce qui vient d'être décrit, le robot possède une centrale inertielle, ou IMU (Inertial Measurement Unit). Il s'agit d'un instrument mesurant l'accélération, le champ magnétique et la vitesse angulaire. Ses composants intègrent un estimateur d'état fournissant la vitesse linéaire et la position. L'IMU utilisée ici est une 3DMCX5-IMU de chez LORD Corporation.

Dans notre cas, l'objet mobile est le robot Bolt, et l'IMU utilisée possède les éléments suivants :

- Un accéléromètre triaxial qui est un ensemble de trois capteurs qui mesurent les accélérations linéaires selon trois axes orthogonaux. De ces données, on en déduit les vitesses et les déplacements du robot.
- Un gyroscope qui mesure les vitesses angulaires.
- Un magnétomètre qui mesure l'intensité ou la direction d'un champ magnétique (pas utile dans notre cas).
- Un capteur de température (pas utilisé).
- Un filtre de Kalman estimateur qui améliore les performances en roulis, tangage et lacet.

Toutes ses données seront utiles lors de la commande du robot.

Le câblage de l'IMU s'est fait selon le plan de montage de la figure 2.21.

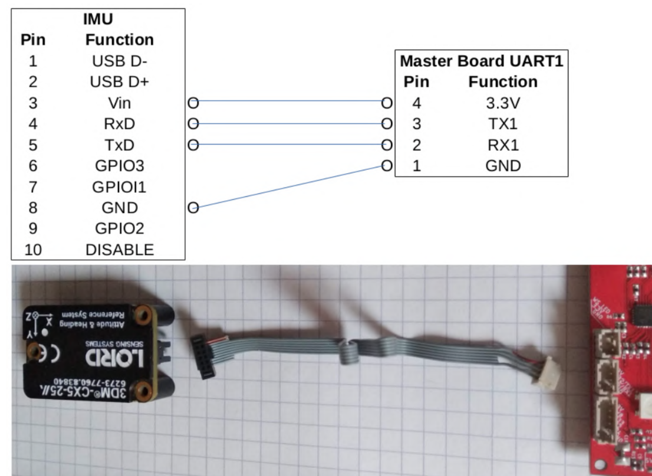


FIGURE 2.21 – Schéma de câblage de l'IMU

On observe que l'IMU possède 10 broches, mais seulement quatre (dont l'alimentation et la masse) seront utilisées. Le câblage a été une phase délicate car le composant est cher et la nappe de fil est assez fine. Elle doit donc être découpée avec précision, puis j'ai dû vérifier qu'il n'y avait pas de mauvaises connexions ou de court-circuit à l'aide d'un multi-mètre. Une fois le câblage terminé, on fixe l'IMU à un emplacement défini sur le haut du robot, puis on le branche à la masterboard. Ainsi on termine le montage électronique du robot.

2.3 Difficultés rencontrées et solutions

2.3.1 Impression 3D

Un des premiers problèmes rencontré a été avec l'imprimante 3D. J'ai utilisé la matière déjà disponible au laboratoire pour faire les impressions. Cependant cela faisait un certain temps que l'imprimante n'avait pas été utilisée. Et cela a eu deux conséquences. Tout d'abord la matière de support, le PVA, est une matière qui se dégrade à l'air libre. Pour limiter ce problème, des bobines de matière sont mises dans un caisson sous hotte aspirante. Malgré ces précautions, le fil de matière devient friable au bout d'un moment et peut donc se casser dans le tube d'arrivée de la machine, avec pour conséquence d'arrêter l'impression en cours. Cette rupture du fil peut aussi être liée à un dépôt de micro-particules (poussières ou reste de matière) dans les buses d'impression.

Ce problème est apparu à quelques reprises. La première solution a été de retirer une longueur de PVA usé de la bobine et d'éviter un rachat de matière. La seconde solution a été de nettoyer les buses. L'imprimante a une fonction qui permet de réaliser ce nettoyage.

Il peut arriver que l'imprimante ne reconnaisse plus la matière insérée, ou ait une mauvaise estimation de la matière restante après de telles manipulations. Il faut alors retirer puis intervertir deux bobines, en faisant attention de ne pas avoir fait un noeud dans la bobine d'enroulement avant de la replacer. La reconnaissance est alors bonne. Parfois il suffit d'éteindre l'imprimante puis de débrancher l'alimentation de la hotte et de rallumer le tout pour régler le problème. De manière générale, un QR code dirigeant vers une notice spécifique de l'erreur est affiché sur la machine. Il suffit de suivre les étapes pour la réparer.

Une autre difficulté liée à l'impression des pièces a été la précision d'impression. Comme expliqué précédemment, on désire insérer en force dans la matière certains éléments, tels que les roulements. Cela implique que la pièce 3D ait été imprimée aux dimensions exactes du plan CAO des pièces afin que le roulement rentre dans la pièce en forçant un peu, et n'en ressorte plus une fois dedans. On évite notamment l'utilisation de colle. Face à cette contrainte, deux problèmes peuvent subvenir : soit la pièce a un seuil d'ajustement trop bas et les roulements "nagent" dans leur emplacement, soit le seuil est trop haut et donc les roulements ne peuvent pas y rentrer. Dans le premier cas, la meilleure solution est de réimprimer la pièce. Ce cas ne s'est pas produit. Dans le second cas, il est possible de résoudre le problème en utilisant des fraises du diamètre du roulement. J'ai rencontré ce problème, notamment sur les premières pièces. J'ai donc rogné légèrement la matière, ou bien forcé avec une presse manuelle afin de faire rentrer les roulements.

De manière générale, la solution réside dans le fait d'imprimer des pièces de test pour différents niveaux de seuil d'ajustement. C'est le paramètre "horizontal expansion" du logiciel de l'imprimante 3D qui permet cet ajustement. Les pièces de tests avaient déjà été réalisées avant mon arrivée, notamment pour les robots Solo. Les premières impressions n'étant pas parfaites, nous avons modifié les paramètres jusqu'à trouver le bon : "horizontal expansion" = - 0.07mm. Il est donc important de ne pas lancer trop d'impression de pièces avant d'avoir déterminé ce paramétrage sous peine de tout ré-usiner, voire de tout réimprimer.

2.3.2 Commandes

Une autre difficulté a été dans la commande des éléments. J'ai commencé par faire une liste complète des pièces nécessaires pour le montage du robot. Mais étant donné que j'allais utiliser les éléments du Solo 8, je ne savais pas exactement quelles pièces allaient être récupérées, ni dans quel état (réutilisable ou non). De plus, l'équipe possède un atelier dans lequel certains éléments sont déjà présents : stock restant de commandes précédentes. Le LAAS possède aussi un FabLab et un magasin dans lequel sont stockés des éléments de visserie ou de soudure que je pouvais utiliser. Au final, faire la liste des pièces à commander n'a pas été simple à établir.

Une fois celle-ci réalisée, il a fallu trouver les bons fournisseurs. Pour cela, il faut savoir que le laboratoire a des partenariats avec certains. Il faut donc favoriser les commandes chez eux pour deux raisons : il y a un tarif préférentiel dû au partenariat et l'expédition (et donc la réception) des pièces est plus rapide. Cependant, certaines références de produits n'existent pas chez ces fournisseurs. Il faut alors, soit trouver un équivalent quitte à adapter

très légèrement le robot (poids ou matière de vis par exemple), soit les commander chez un autre fournisseur. Là encore, il existe une liste de fournisseurs extérieurs à qui le laboratoire a déjà fait des commandes de matériel, notamment pour les robots Solo. Il faut donc les favoriser, ou pas, dans nos choix selon comment se sont passées les précédentes commandes. Une commande chez un fournisseur non partenaire peut prendre jusqu'à un mois et demi à cause des formalités administratives.

L'ensemble de ces contraintes, louables d'un point de vue partenariat et qualité, a complexifié la réalisation de la liste des pièces. Celle-ci a été mise à jour pendant environ un mois et demi, soit jusqu'à la fin de la conception matérielle du robot.

2.3.3 Électronique

La première difficulté a été de flasher les micro-drivers. L'adaptateur utilisé est très sensible et empêche le téléversement du code. Après de nombreuses tentatives, il a fallu débrancher le clavier de l'ordinateur pour pouvoir y arriver.

Il faut ensuite faire attention à la longueur des câbles SPI entre les micro-drivers et la masterboard. Ceux-ci doivent être les plus court possible car dans le cas contraire, ils sont sujets à des interférences.

La dernière difficulté de cette partie a été de rallonger tous les câbles. Ceux récupérés du Solo 8 étaient adaptés pour ce robot et étaient tous trop courts pour le Bolt. Je n'ai pensé à ce point qu'après avoir assemblé les actionneurs. J'ai dû les démonter afin de remplacer et souder des câbles d'une longueur suffisante.

2.3.4 Présentiel / Distanciel

Une ultime difficulté en rapport à la période de pandémie de COVID-19 durant laquelle s'est déroulé le stage, s'est rajoutée. J'ai pu être en présentiel au laboratoire le premier mois, ce qui m'a permis de bien avancer la partie hardware. Mon objectif était d'en faire un maximum tant que je pouvais accéder au LAAS. Il y a ensuite eu une série de travail en distanciel (cluster et confinement). L'assemblage ne pouvant se faire qu'en présentiel, il y a eu une pause de quelques semaines pour cette partie.

Aucune solution n'étant possible compte-tenu de la situation sanitaire, j'ai dû m'adapter en travaillant en parallèle sur la partie software du robot.

3. Software

L'objectif de cette partie software est double : faire fonctionner le robot grâce aux codes déjà réalisés par le Max Planck Institute, et ce en s'appropriant les éléments déjà disponibles. Puis adapter le tout, codes et robot, aux paquets de "ros2_control" [12] via Robot Operating System (ROS) 2 [13].

3.1 État de l'art et architecture

La partie Software du robot est directement liée au développement Hardware de celui-ci, notamment à l'électronique. Ainsi, certains programmes qui permettent un contrôle du Bolt via ordinateur et câble Ethernet ont déjà été développés par des chercheurs de l'institut Max Planck en amont de mon stage [14]. Ceux-ci sont principalement en C++ et en CMake, un système de construction logicielle multi-plateforme auquel j'ai dû me former [15]. Les informations sont directement transmises à la masterboard qui s'occupe de la gestion desdits programmes en parallèle des informations issues des capteurs et des moteurs.

Un élément clé de la compréhension d'un ensemble de code est la réalisation d'un arbre de dépendance (ou arborescence) de ceux-ci. Cela permet de savoir rapidement quel code dépend de quel autre code ou librairie. Ainsi hiérarchiser leur utilisation permet de mieux comprendre l'utilité de chacun. J'ai donc réalisé l'arbre de dépendance associé à l'utilisation de Bolt. Pour cela, j'ai examiné tous les fichiers d'en-tête (.h ou .hpp) dans lesquels on trouve les inclusions de codes ou de librairies, repérables par "#include". Ainsi je suis remonté à divers répertoires nécessaires au fonctionnement global du robot.

Une fois toutes les dépendances listées, je les ai regroupées sous la forme d'un arbre en utilisant le langage DOT. Celui-ci est assez simple à prendre en main et permet d'établir des arborescences facilement. J'ai réalisé une version complète qui était beaucoup trop complexe car de nombreux codes n'étaient pas utiles à la compréhension générale. J'ai donc fait une version simplifiée bien plus utile (cf Annexe B).

Avec tous ces éléments, il est possible de prendre en main et donc de comprendre les codes déjà réalisés.

3.2 Compréhension et compilation

Cette partie est le deuxième grand axe de ce stage. Ici, l'objectif a été d'analyser chaque code et groupement de code pour comprendre leur rôle dans le fonctionnement du robot. Suite à cela, il fallait les tester sur le bipède afin d'identifier ce qui contrôle le robot et ce qui récupère les données des capteurs. Cette partie a sûrement été la plus complexe et chronophage de mon stage. J'expliquerai pourquoi dans la partie 3.2.4.

3.2.1 Compréhension

Comme cela est visible sur l'arbre de dépendance de l'Annexe B, le fonctionnement du robot nécessite de nombreux codes regroupés par dossiers, selon leur utilité. On remarque d'abord deux grandes familles : open-dynamic-robot-initiative (en bleu) et machines-in-motion (en vert). Il s'agit de deux organisations (ou entités) qui contiennent respectivement les codes proches des éléments de robot, et les codes-outils de ces éléments : gestion temps-réel par exemple. Au sein de celles-ci, on trouve lesdits dossiers que l'on nomme des dépôts (ou *repository* en anglais). Ils regroupent les codes par fonctionnalité. Chaque dépôt regroupe les fichiers d'en-tête (.hpp) dans un dossier "include" et les fichiers source (.cpp) dans un dossier "src". D'autres fichiers utiles à la compilation s'y trouvent aussi mais nous en parlerons dans la sous-partie suivante.

Faisons un point sur le rôle des différents dépôts :

- machines-in-motion :
  - yaml_utils : offre une compatibilité entre deux librairies : "Eigen3" et "yaml-cpp". De manière plus générale, les fichiers ".yaml" (Yet Another Modeling Language) fournissent les caractéristiques physiques du robot (nombre et numéros des moteurs, courant maximal, limites des moteurs, offsets, etc.).
  - mim_msgs : s'occupe des messages et services ROS pour l'ensemble de l'organisation machines-in-motion.
  - real_time_tools : ce paquet est un "wrapper" (c'est-à-dire un programme d'appel de fonction) qui fournit des interfaces de programmation d'applications (API) sur des systèmes d'exploitation (OS) temps-réel comme Xenomai ou RT-PREEMPT. Cela permet d'avoir des interfaces logicielles similaires pour des OS temps-réels ou non (Ubuntu par exemple).
  - time_series : ce paquet est utilisé pour stocker des variables doubles avec un timestamp. C'est très utile pour analyser les problèmes.

- open-dynamic-robot-initiative :
  - master_board_sdk : c'est un kit de développement logiciel (SDK) qui fournit une interface en C++ pour envoyer des commandes à la masterboard via le câble Ethernet (ou via Wifi, mais pas utile dans notre cas).

- ◦ odri_control_interface : c'est une interface de contrôle C++ commune à tous les robots utilisant une masterboard.
- ◦ blmc_drivers : c'est un protocole de communication entre un ordinateur de commande et les cartes de contrôle de l'électronique de puissance qui commandent les moteurs.
- ◦ bolt : il s'agit des drivers du robot Bolt.

Après lecture de chaque code et une compréhension de leur rôle, l'objectif est de tous les compiler pour générer un code bas niveau compréhensible par le robot. C'est la génération de cet exécutable que nous allons voir dans la partie suivante.

3.2.2 Compilation

Parlons à présent plus en détail de ROS 2. Il s'agit de la nouvelle version de ROS (Robot Operating System), un ensemble de bibliothèques logicielles et d'outils permettant de créer des applications robotiques. On y trouve des pilotes, des algorithmes de pointe, ou encore des outils de développement puissants, le tout en open-source. ROS 2 est une version améliorée qui tient compte des changements faits par la communauté robotique sur ROS. Ainsi elle s'adapte à ces changements, et tire parti des avantages de la version antérieure.

Pour compiler des paquets avec ROS 2, j'ai utilisé l'outil "colcon" via la commande "colcon build". Celle-ci s'applique dans des répertoires à l'architecture particulière qu'on appelle des workspaces. Un workspace est un dossier souvent nommé avec le suffixe "_ws". À l'intérieur on trouve un dossier "src" où il y a tous les dépôts évoqués dans le paragraphe précédent. Cependant, pour pouvoir compiler un dépôt, celui-ci doit contenir deux fichiers particuliers, aux rôles différents :

- CMakeLists.txt : c'est un fichier de configuration écrit en langage CMake, contenant un ensemble d'instructions décrivant les fichiers sources et les cibles du projet. Il permet de générer des fichiers binaires, c'est-à-dire compréhensibles par un processeur avec un OS (exécutables, librairies, etc.) à partir de dépendances établies.
- package.xml : il définit les propriétés du paquet telles que son nom, le numéro de version, les auteurs et les dépendances avec d'autres paquets. Il est écrit en langage XML (eXtensible Markup Language).

À partir de ces documents et du dossier src, l'exécution de la commande "colcon build" (avec ou sans options) à la racine du workspace crée trois dossiers : "build", "install" et "log". Elle permet aussi de voir s'il y a des erreurs de compilation le cas échéant. Toutes les explications et des exemples d'utilisation sont disponibles en référence [13] [15].

Pour la compilation du dépôt "bolt", un certain nombre de dépôts ont dû être compilés en plus de ceux déjà évoqués :

- mpi_cmake_modules : définit une liste de documents cmake utiles.
- googletest-distribution : c'est le framework (l'infrastructure logicielle) de test de Google.
- shared_memory : un wrapper pratique et qui a pour but d'être simple d'utilisation. Il est adapté au temps-réel (sans allocation dynamique).
- signal_handler : fournit une classe statique qui enregistre un gestionnaire de signal et qui vérifie si le signal est reçu.
- robot_properties_bolt : contient les propriétés physiques du robot Bolt dans des fichiers URDF ainsi que des tests en Python.

Les origines de ces dépôts sont diverses, tout comme leur rôle dans la compilation, mais leur nécessité a été mise en évidence par la commande "colcon build". Il faut un niveau avancé dans l'utilisation de ce type de projet pour bien saisir le besoin de ces dépôts dans la compilation.

Une partie parallèle à la compilation est la manière de versionner les codes, et de facilement les récupérer sur différents ordinateurs. Pour faire cela, j'ai utilisé Git et GitLab. Git est un logiciel libre de versionnage de code. Il permet de sauvegarder des modifications de fichier et d'y joindre un commentaire explicatif des modifications faites. On peut ainsi avoir un historique de toutes les modifications faites sur un projet, et éventuellement de récupérer une version antérieure. Git s'utilise sur un ordinateur, en local, et donc pas sur plusieurs ordinateurs en collaboration. Cependant, de gros projets sont souvent le résultat de travaux collaboratifs entre plusieurs personnes, ou alors nécessitent l'utilisation des mêmes codes sous plusieurs ordinateurs. Ainsi entre en jeu l'utilisation des dépôts distants. Les plus connus sont GitHub et GitLab. GitHub, nous en avons déjà parlé dans la première partie. Je m'en servais pour seulement consulter les informations que contenaient certaines pages, et non pas comme une plateforme de partage de code. Cependant pour les divers dépôts dont nous avons parlé, je les ai extraits en local depuis GitHub. Pour ce qui est de ma gestion personnelle des codes et des modifications futures, j'ai utilisé GitLab. C'est un dépôt distant similaire à GitHub mais dont le LAAS possède un serveur. Ce logiciel m'a ainsi permis de stocker les codes modifiés, et surtout de les récupérer sur les divers ordinateurs connectés au dépôt distant. Ainsi toutes les modifications étaient centralisées (GitLab), puis exportées sur tous les dépôts locaux (ordinateurs).

J'ai dû prendre en main ces logiciels afin d'en tirer le meilleur, gagner en efficacité dans les diverses phases de compilations, et pour corriger les erreurs. Cela a été très pratique pour mettre en place la partie compilation entre les ordinateurs du LAAS, et mon ordinateur personnel, notamment avec le télétravail : je pouvais faire des modifications depuis chez moi et les récupérer facilement au laboratoire. Le dépôt distant donnait une vision rapide et à distance de mes codes à mon tuteur, et donc a facilité l'aide qu'il m'a fournie dans cette partie. Cette aide est relative aux éléments développés dans la partie 3.2.4.

3.2.3 Tests

La phase de test sur le système réel a eu lieu assez tard, après avoir commencé les éléments développés dans la partie 3.3. Des retards de livraison d'éléments de calibrage et une longue mise en place d'un nouveau poste de commande en ont été la cause.

Avant de parler des codes, évoquons le montage physique nécessaire au fonctionnement du robot. On a évoqué dans la partie Hardware un câble Ethernet et des câbles d'alimentation. Au bout du câble Ethernet doit se trouver un ordinateur à noyau Linux 20.04 afin de pouvoir utiliser l'ensemble des paquets qui seront utiles au projet. Cet ordinateur doit avoir deux cartes réseau : une reliée au réseau local du laboratoire (pour Internet, etc.), et une seconde qui sera consacrée au robot. C'est sur celle-ci que sera connecté le câble Ethernet. Pour l'alimentation, on utilise un générateur de tension capable de fournir 25V et environ 1A, courant continu suffisant pour faire fonctionner le robot. Cependant on ne connecte pas le robot directement sur l'alimentation. Entre les deux, on dispose un bouton d'arrêt d'urgence. Il a un double rôle : arrêter le robot en cas de problème de code lui faisant faire des actions dangereuses pour son intégrité, mais aussi pouvoir avoir un interrupteur afin de laisser l'alimentation allumée entre différents tests, sans avoir le robot toujours alimenté (limite une usure ou un échauffement inutile du robot). Une fois cette configuration faite, on peut commencer les premiers tests du système.

Les premiers tests ont permis de vérifier que le robot n'avait pas de problème de conception. Pour cela, j'ai lancé un code Python de calibrage des moteurs. Il s'agit d'un code fait pour les robots Solo (8 et 12) et pour le banc de test. J'ai donc dû les prendre en main, puis les adapter au robot Bolt, en créant notamment une classe. Lors de l'assemblage du robot, les roues codeuses sont positionnées arbitrairement sur le moteur. Lorsque le système associe une valeur à une certaine configuration initiale du robot, on parle de calibrage. Dans notre cas, j'ai imprimé des pièces en 3D pour positionner le robot comme sur la figure 3.1, i.e. les jambes tendues et droites. Les valeurs récupérées des capteurs dans cette position seront soustraites afin de la définir comme position initiale, avec tous les encodeurs à 0.

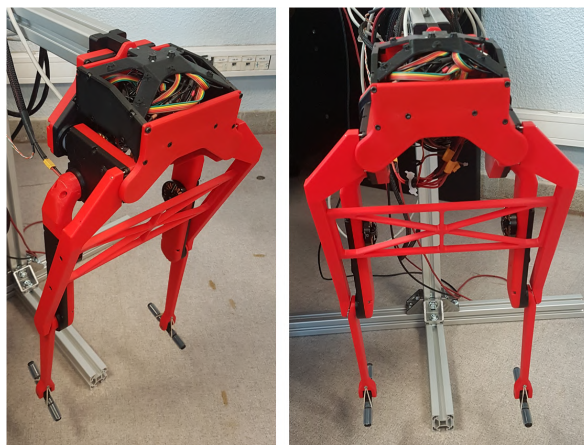


FIGURE 3.1 – Bolt en position de calibrage

Ces tests ont été concluants et m'ont donc permis de valider la bonne conception du robot. Après des difficultés dues à l'architecture de mon ordinateur (cf partie 3.2.4), j'ai pu exécuter trois codes de démonstration du Max Planck Institute qui ont été compilés via l'outil "colcon". Un premier permettait de tester le calibrage, un second d'envoyer une consigne en utilisant un correcteur proportionnel dérivé (PD), et un troisième d'afficher les valeurs de tous les capteurs. Cela m'a permis de bien comprendre les codes réalisés par le MPI, mais aussi d'en créer un qui regroupe les fonctionnalités de chacun. Ainsi une fois ce nouveau code exécuté sur le robot, il commence par tester le calibrage, puis il exécute des mouvements de jambes périodiques (cf figure 3.2), tout en affichant à l'écran les valeurs de tous les capteurs chaque seconde pour limiter la perte du temps CPU.

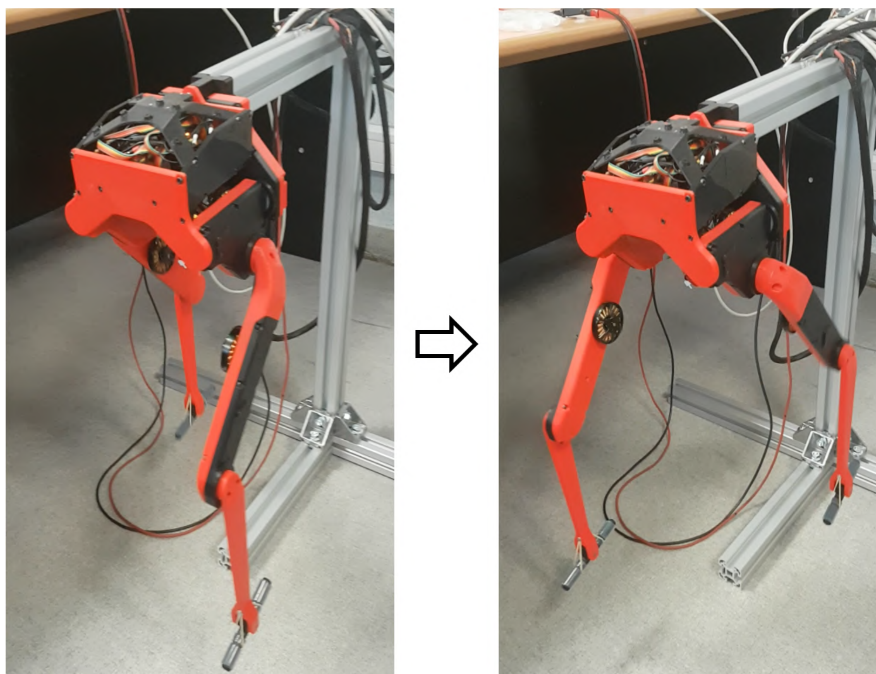


FIGURE 3.2 – Tests de fonctionnement des codes du Max Planck Institute

En réussissant à m'approprier les codes du MPI, j'ai bien compris quel était le rôle de chaque variable : celles permettant de récupérer les données des capteurs, celles permettant de commander indépendamment chaque actionneur, et de gérer le tout à une fréquence de 1 kHz. Ainsi, après deux mois de travail, cela met fin à la partie de compréhension et d'exécution de travaux précédents, et prépare au mieux la suivante, qui est l'adaptation à ros2_control.

3.2.4 Difficultés

Les étapes de prise en main et de compilation des codes existants ont été des parties complexes pour moi, et ce pour diverses raisons.

Tout d'abord, il n'est jamais simple de se mettre à la place des personnes qui ont écrit les divers codes. Ils ont une grande connaissance du domaine, des systèmes en question et

une grande maîtrise des langages employés. En tant que stagiaire pendant six mois, j'ai uniquement pu m'approprier le système lors de sa fabrication, durant les deux premiers mois. À cela s'ajoute une expérience uniquement scolaire des systèmes et de leur fonctionnement général, ainsi que de certains langages utilisés (C++ pour les codes, et les bases de Linux pour la compilation).

Avec ces éléments, je me suis retrouvé face à une très grande diversité de codes, ayant tous une utilité différente et bien précise, mais dont la documentation ou les commentaires étaient très peu présents. Malgré mes connaissances qui ont été utiles à une compréhension en surface, j'ai dû me former à de nouveaux langages (ROS 2, CMake, ...) afin d'avoir une compréhension et une productivité palpable. Cela m'a pris du temps et cela n'a pas été simple de passer directement de tutoriels à une application complexe. De même pour la compilation. Celle-ci nécessite une bonne connaissance des ordinateurs (environnement, dépendances, paquets ...) et des lignes de commande pour réaliser des actions. Je n'avais pas ces connaissances en début de stage. C'est en tombant sur de longues successions d'erreurs et grâce à l'aide fournie par mon tuteur de stage que j'ai pu comprendre et apprendre. Mon tuteur est très pointu dans le domaine, et cela grâce à ses années d'expérience. Il comprenait vite les problèmes, avait en tête diverses causes de ceux-ci, ainsi que des solutions adaptées. Je cherchais aussi de mon côté avant de voir avec lui, mais il était facile de se perdre sur Internet, vu la diversité des solutions proposées. Ses conseils ont été précieux et grâce à ses explications et mon écoute active, j'ai pu les réutiliser dans d'autres situations similaires, et ainsi faire de ces problèmes une partie instructive.

Les problèmes de compilation ont été nombreux et souvent liés à des paquets et librairies absentes ou mal appelées. Savoir lesquels il était bon d'installer et lesquels il ne fallait pas installer était très complexe, voire impossible pour moi sans avoir l'avis de mon tuteur. Dans ce cadre-là, seule l'expérience et une bonne connaissance des logiciels est utile.

Une autre grande difficulté a été dans la compréhension de l'utilisation du fichier CMakeLists.txt. C'est celui-ci qui assure notamment la récupération de librairies dans un projet et leur assemblage. Il est écrit en langage cmake. Une compréhension du rôle des fonctions est nécessaire car leur utilisation dépend de l'ordre d'appel de celles-ci. Cela demande du temps et le passage de l'apprentissage théorique à la pratique n'a pas été simple.

D'autres problèmes, liés à la structure de mon ordinateur du laboratoire, ont ralenti mes avancées. Je ne possédais pas toujours les droits, certains chemins d'accès ne fonctionnaient pas, des priorités n'étaient pas toujours les bonnes, etc. Deux éléments étaient en cause : le type d'utilisateur et la définition de l'environnement de l'ordinateur. Je ne vais pas plus rentrer dans les détails, mais j'aurais été incapable de trouver les causes de ces erreurs seul. Mon tuteur m'a débloqué et m'a appris de nombreuses choses dans ce domaine. Malgré tout, ces blocages ont fortement limité mes avancées pendant environ deux semaines.

Même si sur cette partie, mon tuteur est souvent intervenu pour résoudre des problèmes que je n'aurais pas réussi à régler seul dans tous les cas, j'ai beaucoup appris de cela, ou par moi-même lors de mes recherches. Ainsi malgré les difficultés évoquées, j'en ai retiré quelque

chose de positif.

3.3 Passage à ros2_control

Cette partie entame le troisième et dernier grand axe de ce stage. L'objectif est d'utiliser la compréhension du système et des codes de ce dernier afin de les assembler à une infrastructure logicielle (framework) en cours de développement lors du stage : ros2_control. Plus précisément, cela a consisté à créer un document spécifique à l'architecture du Bolt (URDF), puis à réaliser des actions proches du Hardware du robot : le démarrer, récupérer les données de ses capteurs, envoyer des commandes aux moteurs et l'arrêter. Ces actions doivent s'adapter aux dépôts de ros2_control et de open-dynamic-robot-initiative (odri_control_interface notamment, cf partie 3.2.1).

3.3.1 Prise en main

La commande et l'architecture globale du robot se font via ROS 2, la version la plus récente de ROS (Robot Operating System). Il s'agit d'un ensemble d'outils informatiques libres, permettant de développer des logiciels pour la robotique. En parallèle de mon stage, des chercheurs ont travaillé sur la mise en place d'un framework comprenant les interfaces et les gestionnaires des contrôleurs, ainsi que les transmissions et les interfaces matérielles. Cette infrastructure s'appelle "ros2_control" [12] [16]. J'ai aussi dû me former à ces outils [17] afin de pouvoir les comprendre dans un premier temps, puis les utiliser.

ros2_control est un framework pour le contrôle (en temps réel) de robots utilisant ROS 2. Il est disponible avec la version ROS 2 Foxy. Ses paquets sont une réécriture des paquets ros_control utilisés dans la première version de ROS. L'objectif de ros2_control est de simplifier l'intégration de nouveaux matériels, de surmonter certains inconvénients, le tout en s'adaptant aux nouveautés de ROS 2.

L'intégralité des codes composants ros2_control se trouve sur GitHub. Étant donné la diversité de codes, et tout comme pour la deuxième partie du stage (partie 3.1), j'ai réalisé les arborescences des codes de ros2_control et de ses dérivés. Cela m'a permis de mieux cerner les dépendances entre chacun d'eux et de mieux comprendre leurs rôles. Voici une rapide présentation des dépôts concernés :

- ros2_control : les principales interfaces et composantes du framework. Composé de :
  - hardware_interface : il définit sous forme de code tous les types de composants physiques du robot : moteurs, capteurs, et leurs interfaces. Il s'agit de la brique la plus proche de la partie Hardware.
  - controller_manager : d'un côté, il gère différentes actions des contrôleurs (activation, désactivation ...) et les interfaces requises pour cela. De l'autre côté, il a accès aux interfaces des composants matériels. Il fait correspondre les interfaces

requises et donne aux contrôleurs l'accès au matériel lorsqu'il est activé, ou signale une erreur en cas de conflit d'accès.

- ○ transmission_interface : gère, sous la forme de codes informatiques, de nombreux types de transmissions physiques comme par exemple des engrenages, ou des liaisons poulie-courroie.
- ○ joint_limits_interface : contient des structures de données pour représenter les limites des articulations, des méthodes pour remplir ces structures à partir du serveur de paramètres ROS et de formats communs (ex : URDF), et des méthodes pour faire respecter les limites sur différents types d'interfaces matérielles.
- ○ ros2_control_test_assets : contient des fichiers URDF de robots afin de réaliser des tests.
- ○ d'autres sous-dépôts non développés ici.

- ros2_controllers : des contrôleurs largement utilisés, comme le contrôleur de commande en position, le contrôleur de trajectoire, le contrôleur d'entraînement différentiel. Il y a des sous-dépôts pour chaque type de contrôleur (non détaillés ici).

- ros2_control_demos : des exemples de mise en œuvre de cas d'utilisation courants, utile pour de la prise en main. C'est sur les codes d'un quadrupède de ce dépôt que je me suis basé pour adapter Bolt à ros2_control.

Devant le nombre de dépendance et la taille de l'arbre final, j'ai décidé de montrer ici uniquement l'arbre de dépendances complet de ros2_control, sans les dérivés qui en découlent. Il est disponible en Annexe C.

Vous trouverez en Annexe D un schéma de l'architecture de ros2_control. Nous allons nous baser sur celui-ci, ainsi que sur la figure 3.3, qui est est un schéma bloc du fonctionnement de ros2_control par rapport au robot TALOS, pour détailler un peu plus mes objectifs.

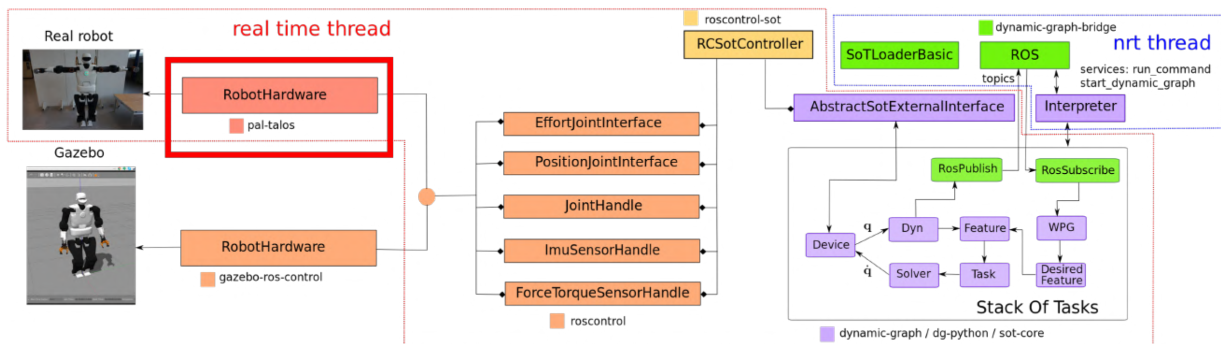


FIGURE 3.3 – Schéma bloc de l'utilisation de ros2_control pour TALOS

On voit au centre cinq interfaces correspondant aux actionneurs et aux capteurs. La moitié de droite représente les lois de contrôle et d'autres axes qui sont développées par des

doctorants. À gauche se trouvent deux parties : celle représentant le robot sur simulateur (en bas) et celle représentant le robot réel (en haut). Dans les blocs "RobotHardware" sont définis les éléments composant le système (moteurs et capteurs, réels ou sur simulateur) en lien avec leur interface sous ros2_control. Le bloc qui nous intéresse est celui relatif au robot réel, encadré en rouge sur la figure. Ce dernier lie les lois de contrôle avec les variables des composants. C'est dans ce bloc que j'ai dû adapter l'ensemble des codes des divers dépôts pour réaliser quatre fonctions :

- start() : démarrer le robot,
- read() : récupérer les données des capteurs,
- write() : envoyer des commandes issues des lois de contrôle aux moteurs,
- stop() : arrêter le robot.

En faisant un parallèle avec l'Annexe D, le cadre rouge correspond au même niveau de travail que sur la figure 3.3. Passons à présent aux détails des modifications réalisées.

3.3.2 Modifications

Dans un premier temps, j'ai dû définir Bolt sous la forme d'un fichier URDF au format xacro. Il s'agit d'un fichier dans lequel on décrit les propriétés physiques du robot : nombre d'actionneurs, leur nom, leurs limites, leurs entrées (position, vitesse, accélération), leurs sorties (idem) ; les capteurs (IMU par exemple) et leurs propriétés. Vous trouverez en Annexe E un bout de ce fichier. Ce travail est nécessaire pour pouvoir adapter toutes les fonctions au robot en question. Le grand avantage est que si l'on désire utiliser un autre système (Solo 12 par exemple), il suffit de changer d'URDF et le reste de l'architecture s'adapte au changement.

Pour assurer cela, il est nécessaire de modifier toutes les fonctions afin qu'elles aillent récupérer l'ensemble des paramètres dans l'URDF et qu'elles réalisent le comportement désiré. Dans mon cas, j'ai créé des codes "system_robot.hpp" et "system_robot.cpp" inspirés de codes incomplets d'un système quadrupède développé par les membres de ros2_control : "rrbot_system_quadruped.cpp". Ce dernier était jusqu'à présent utilisé pour tester des fonctions de base, en simulation, sur un robot nommé rrbot qui a deux degrés de liberté (rotation-rotation-robot). Les modifications de ce code ont été de deux types :

1. L'initialisation des composants (masterboard, moteurs et IMU) afin de créer la variable définissant le robot Bolt.
2. Les modifications des quatre fonctions explicitées précédemment.

1. Étant donné qu'un des objectifs est d'adapter odri_control_interface à ros2_control, la première chose à faire est de définir le robot de la même manière dans les codes de ros2_control_bolt. Pour cela, on commence par créer une dépendance entre les deux répertoires. Puis il faut définir le robot en utilisant les données définies dans l'URDF. Ainsi,

j'ai créé une fonction "init_robot()" qui initialise le robot, par l'intermédiaire de variables globales :

- L'interface de communication Ethernet avec la masterboard,
- La masterboard,
- Chaque moteur à partir des éléments de l'URDF,
- Une variable regroupant tous les moteurs,
- Les positions de calibrage (ou d'offset) de chaque moteur à partir de l'URDF,
- L'IMU via les données de l'URDF,
- Le robot, étant une variable regroupant tous les autres éléments cités ici.

Ainsi, nous avons une carte (map) utilisable dans tout le code et étant directement liée au robot réel connecté par Ethernet. Cela est primordial pour pouvoir coder les quatre fonctions désirées. Les avantages sont que nous avons défini cette variable de la même manière que dans le répertoire odri_control_interface, et avec des données initiales issues de l'URDF. Cela permet de pouvoir s'adapter facilement à d'autres robots (changement de l'URDF), ainsi que d'utiliser, adapter et donc inclure les travaux précédents (cf partie 3.2.1) à ros2_control.

2. Une fois l'initialisation du robot faite, on peut passer aux quatre fonctions de base définies précédemment.

Tout d'abord la fonction start(). Elle a pour but de faire démarrer le robot afin qu'il soit prêt à être utilisé à la fin de celle-ci. Pour cela, nous appelons une fonction définie par ODRI et qui met en marche la masterboard et la prépare à des échanges d'information. En plus de cela, nous voulons que le robot soit calibré avant de faire quoi que ce soit avec. Ainsi j'ai créé une autre fonction, "calibration()", qui assure le calibrage de tous les moteurs du robot. Elle utilise pour cela les positions d'offset récupérées de l'URDF. Cela est fait dans une fonction à part afin de laisser les utilisateurs libres de pouvoir faire le calibrage au moment où ils le désirent. Avec mon tuteur, nous avons décidé de faire cette tâche lors du démarrage. Enfin, une lecture des capteurs est ensuite faite (cf paragraphe suivant). Après toutes ces étapes, on considère que le robot est démarré.

Passons à la fonction read(). Son but est de récupérer les valeurs de tous les capteurs (encodeurs et IMU). La partie encodeur est stockée dans une structure de type "map", qui décrit l'état du système, avec pour chaque moteur :

- Sa position,
- Sa vitesse,
- Son couple mesuré,
- Son gain de position Kp,
- Son gain de vitesse Kd.

Pour l'IMU, une autre "map" réalise la même action en stockant :

- Les données du gyroscope,
- Les données de l'accéléromètre,
- Les accélérations linéaires,
- Les attitudes d'Euler,
- Les attitudes quaternion.

Une fois toutes ces valeurs stockées, la fonction read() a rempli son rôle.

La fonction write() a un but inverse : elle envoie des valeurs à chaque actionneur. Ces valeurs sont stockées dans une autre "map" décrivant les mêmes éléments que la première (position, vitesse, couple, gains Kp et Kd). Grâce à des fonctions d'ODRI, on envoie toutes ces données au robot.

Enfin, la fonction stop() doit arrêter le robot. Pour cela, on utilise une autre fonction issue des codes d'ODRI et qui arrête la masterboard. Comme c'est elle qui gère les envois de commandes aux moteurs, son arrêt peut être considéré comme un arrêt du robot.

En adaptant le schéma du TALOS (figure 3.3) à ce que nous venons de décrire pour Bolt, on obtient le schéma de la figure 3.4.

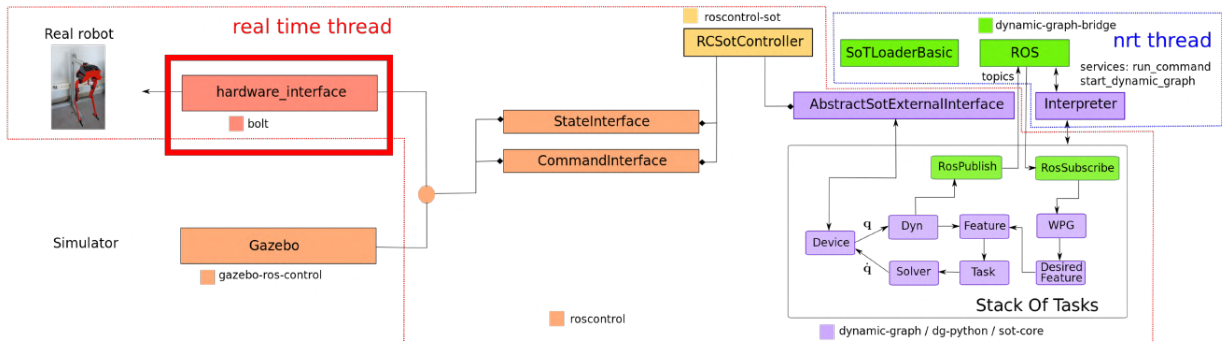


FIGURE 3.4 – Schéma bloc de l'utilisation de ros2_control pour Bolt

On observe une modification des blocs centraux passant de cinq à deux. Cela s'explique par le fait que seules deux interfaces sont nécessaires, une gérant les états lors de la lecture des capteurs, l'autre gérant les commandes des moteurs lors de l'envoi de celles-ci. La partie simulation sur ordinateur a aussi disparu car elle n'a pas été faite lors de mon stage, mais elle sera présente dans les futures utilisations faites par le LAAS. Toutes les modifications évoquées ont été faites dans le bloc "hardware_interface" encadré en rouge.

Passons à la phase de test de ces modifications.

3.3.3 Tests

La phase de test a pour but de vérifier le bon comportement des fonctions créées. Pour start(), read() et stop(), cela peut être directement fait sur le robot réel. Pour write(), un

travail d'études de commandes doit être fait. Au moment de mon stage, cela n'a été possible que via le visualisateur de ROS : rviz.

Lors de la rédaction de ce rapport, ces phases de test n'ont pas encore été réalisées. Elles devraient être faites dans les derniers jours de mon stage. Seule une compilation sans erreur et le regard avisé de mon tuteur sur ces fonctions ont joué un rôle de test jusqu'à présent, sans pour autant valider complètement cette partie de mon travail.

En tout, j'ai passé 1 mois et demi sur cette partie liée à ros2_control.

3.3.4 Difficultés

La première difficulté est proche de celles expliquées dans la partie 3.2.4, c'est-à-dire la prise en main des codes. Un des avantages cependant a été que mon tuteur fait partie du groupement de développeur de ros2_control. Je pouvais donc avoir facilement des réponses à mes questions.

Ensuite, l'accumulation des codes et des variables n'a pas été simple à gérer. Dans cette partie, j'ai dû lier l'ensemble des codes dont j'ai parlé dans ce rapport. Je ne les ai pas comptés mais on doit être proche des 150 fichiers de codes, avec de nombreuses variables, fonctions, méthodes et définitions propres à chacun. La gestion de tout cela a été complexe car cela demande une très bonne connaissance de chaque dépôt pour facilement faire des rapprochements et ne pas se perdre. C'est ce que j'ai essayé de faire lors du stage, mais j'ai manqué de temps, d'expérience et parfois de connaissances pour être à l'aise. Cela a donc été une phase d'apprentissage en conditions réelles assez rude mais enrichissante.

Une autre difficulté a été de ne pas pouvoir tester mes fonctions entre chaque modification. La seule étape de validation que j'ai eue pendant un long moment a été lorsque que la compilation se faisait sans erreur. Mais cela ne signifie en rien que le contenu des fonctions réalise bien le comportement désiré. Ce problème devrait être comblé dans les derniers jours de stage.

Conclusion

L'objectif de ce stage de fin d'étude a été de mettre à disposition des membres de l'équipe un robot bipède léger, peu coûteux, robuste et facile à réparer en cas de problème. Cela leur permettra d'avoir un intermédiaire de test d'algorithmes entre les simulateurs sur ordinateurs et des robots imposants et complexes comme TALOS par exemple. Ce robot deviendra un excellent outil permettant à Gepetto de gagner en efficacité dans leurs travaux.

Le robot en question a été choisi par les membres de l'équipe avant mon arrivée : il s'agit du bipède Bolt développé en open-source par le groupement de chercheurs de Open Dynamic Robot Initiative (ODRI). Celui-ci possède tous les critères souhaités : légèreté, commande en couple, robustesse, conception peu coûteuse et reproductible en cas de casse grâce à des composants faciles d'accès et des pièces imprimées en 3D.

La première phase du stage a donc été de fabriquer le robot (Hardware). Pour cela, je me suis basé sur les plans et toutes les indications fournies par ODRI sur son dépôt GitHub. Pour les composants internes (moteurs, cartes électroniques, ...), j'ai pu recycler le Solo 8 et ainsi avoir la quasi totalité des composants du Bolt. Pour les coques extérieures, je les ai toutes imprimées en 3D au laboratoire. L'usinage et l'assemblage m'a permis de finir la conception du Bolt en environ deux mois.

La deuxième partie a été tournée vers le Software. En plus des plans de fabrication du robot, le GitHub ODRI contient une grande quantité de codes permettant de le tester. Il a donc fallu mettre en place tous les éléments nécessaires à la commande du Bolt sur un ordinateur du LAAS. Cela est passé par une évolution de système d'exploitation, une reconfiguration de l'environnement de l'ordinateur, et une compilation de nombreux packages d'ODRI. Après deux mois de travail sur cette partie, je pouvais calibrer, lire les données des capteurs et mettre en mouvement Bolt via des codes de démonstration adaptés au système.

Enfin, la dernière partie a été l'adaptation de ces codes à une infrastructure logicielle en cours de développement : ros2_control. Celle-ci se base sur ROS 2, la dernière version d'un ensemble d'outils informatiques libres, permettant de développer des logiciels pour la robotique. Elle n'est pas encore beaucoup étendue dans l'équipe Gepetto mais la transition est en train de se faire. L'objectif de cette partie était de faire de ce Bolt le premier robot du LAAS fonctionnant sous ROS 2 et ros2_control. Pour cela j'ai du adapter les codes de la partie précédente à l'architecture de ros2_control, en définissant les caractéristiques propres du robot fabriqué. De plus, quatre fonctions étaient nécessaires pour que l'équipe

puisse facilement l'utiliser : le démarrage du robot, la lecture de tous ses capteurs, l'envoi de commandes et son arrêt. Tous ces points ont été réalisés sans pouvoir être testés avant l'écriture de ce rapport. Cependant, cela n'empêche pas la validation des objectifs après 1 mois et demi. Cette partie clôt l'ensemble du stage en validant l'objectif principal.

L'équipe Gepetto pourra à présent utiliser ce robot Bolt pour tester en conditions réelles des algorithmes avant de passer sur TALOS. De mon côté, j'ai été confronté à de nombreuses difficultés lors de ce stage, que j'ai su analyser, traiter et résoudre selon une démarche que doit être capable de fournir un ingénieur ou un chercheur. Ces problèmes ont été riches en apprentissages et m'ont fait monter en compétences, notamment dans la partie Software.

La continuité de ce stage pourrait être de mettre en fonctionnement le bipède selon des générateurs de marche et des lois de contrôle en cours d'étude par des doctorants. Des tests de commande par vision pourront aussi être réalisés dessus. En parallèle de ce stage, un autre stagiaire a travaillé sur de nouvelles versions des pieds du robot (cf figure 3.5) [18]. La mise en marche de Bolt pourra aussi être l'occasion de tester ces nouveautés. Enfin, l'équipe Gepetto pourra en disposer selon leurs besoins, ceux-ci évoluant quasi-quotidiennement au rythme des problèmes rencontrés.

FIGURE 3.5 – Nouvelles versions des pieds de Bolt (travail d'un autre stagiaire)

Bibliographie

[1] Olivier Stasse, Thomas Flayols, Rohan Budhiraja, Kevin Giraud-Esclasse, Justin Carpentier, Joseph Mirabel, Andrea Del Prete, Philippe Souères, Nicolas Mansard, Florent Lamiraux, Jean-Paul Laumond, Luca Marchionni, Hilario Tome, and Francesco Ferro. TALOS : A new humanoid research platform targeted for industrial applications. *IEEE-RAS 17th International Conference on Humanoid Robotics (Humanoids)*, pages 689–695, 2017.

[2] Felix Grimminger, Avadesh Meduri, Majid Khadiv, Julian Viereck, Manuel Wüthrich, Maximilien Naveau, Vincent Berenz, Steve Heim, Felix Widmaier, Thomas Flayols, Jonathan Fiene, Alexander Badri-Spröwitz, and Ludovic Righetti. An Open Torque-Controlled Modular Robot Architecture for Legged Locomotion Research. *IEEE Robotics and Automation Letters*, 5(2) :3650–3657, 2020.

[3] Jonas Koenemann, Andrea Del Prete, Yuval Tassa, Emanuel Todorov, Olivier Stasse, Maren Bennewitz, and Nicolas Mansard. Whole-body model-predictive control applied to the HRP-2 humanoid. *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 3346–3351, 2015.

[4] Shuuji Kajita, Fumio Kanehiro, Kenji Kaneko, Kiyoshi Fujiwara, Kensuke Harada, Kazuhito Yokoi, and Hirohisa Hirukawa. Biped walking pattern generation by using preview control of zero-moment point. *IEEE International Conference on Robotics and Automation (Cat. No.03CH37422)*, pages 1620–1626 vol.2, 2003.

[5] Olivier Stasse, Björn Verrelst, Pierre-Brice Wieber, Bram Vanderborght, Paul Evrard, Abderrahmane Kheddar, and Kazuhito Yokoi. Modular Architecture for Humanoid Walking Pattern Prototyping and Experiments. *Advanced Robotics*, 22(6-7) :589–611, 2008.

[6] Torea Foissotte, Olivier Stasse, Adrien Escande, Pierre-Brice Wieber, and Abderrahmane Kheddar. A two-steps next-best-view algorithm for autonomous 3D object modeling by a humanoid robot. *IEEE International Conference on Robotics and Automation*, pages 1159–1164, 2009.

[7] Nosan Kwak, Olivier Stasse, Torea Foissotte, and Kazuhito Yokoi. 3D grid and particle based SLAM for a humanoid robot. *9th IEEE-RAS International Conference on Humanoid Robots*, pages 62–67, 2009.

[8] Joseph Mirabel, Florent Lamiraux, Thuc Long Ha, Alexis Nicolin, Olivier Stasse, and Sébastien Boria. Performing manufacturing tasks with a mobile manipulator : from motion planning to sensor based motion control. *IEEE International Conference on Automation Science and Engineering*, 2021.

[9] Isabelle Maroger, Noelie Ramuzat, Olivier Stasse, and Bruno Watier. Human Trajectory Prediction Model and its Coupling with a Walking Pattern Generator of a Humanoid Robot. *IEEE Robotics and Automation Letters*, 6(4) :6361 – 6369, 2021.

[10] Manuel Wüthrich, Felix Widmaier, Felix Grimminger, Joel Akpo, Shruti Joshi, Vaibhav Agrawal, Bilal Hammoud, Majid Khadiv, Miroslav Bogdanovic, Vincent Berenz, Julian Viereck, Maximilien Naveau, Ludovic Righetti, Bernhard Schölkopf, and Stefan Bauer. TriFinger : An Open-Source Robot for Learning Dexterity, 2021.

[11] Felix Grimminger. GitHub - Open Dynamic Robot Initiative - Open Robot Actuator Hardware. Consulté du 01/03/2021 au 04/06/2021 : https://github.com/open-dynamic-robot-initiative/open_robot_actuator_hardware.

[12] Sachin Chitta, Eitan Marder-Eppstein, Wim Meeussen, Vijay Pradeep, Adolfo Rodríguez Tsouroukdissian, Jonathan Bohren, David Coleman, Bence Magyar, Gennaro Raiola, Mathias Lüdtke, and Enrique Fernandez Perdomo. ros_control : A generic and simple control framework for ROS. *The Journal of Open Source Software*, 2(20) :456–456, 2017.

[13] Open Robotics. ROS 2 Documentation : Foxy. Consulté du 08/03/2021 au 20/04/2021 : https://docs.ros.org/en/foxy/Tutorials.html.

[14] Maximilien Naveau, Julian Viereck, Majid Khadiv, and Elham Daneshmand. bolt, since 2020. Consulté le 02/07/2021 : https://github.com/open-dynamic-robot-initiative/bolt.

[15] Jeremy Fix. Tutoriel CMAKE, 2015. Consulté le 09/04/2021 : http://sirien.metz.supelec.fr/depot/SIR/TutorielCMake/index.html.

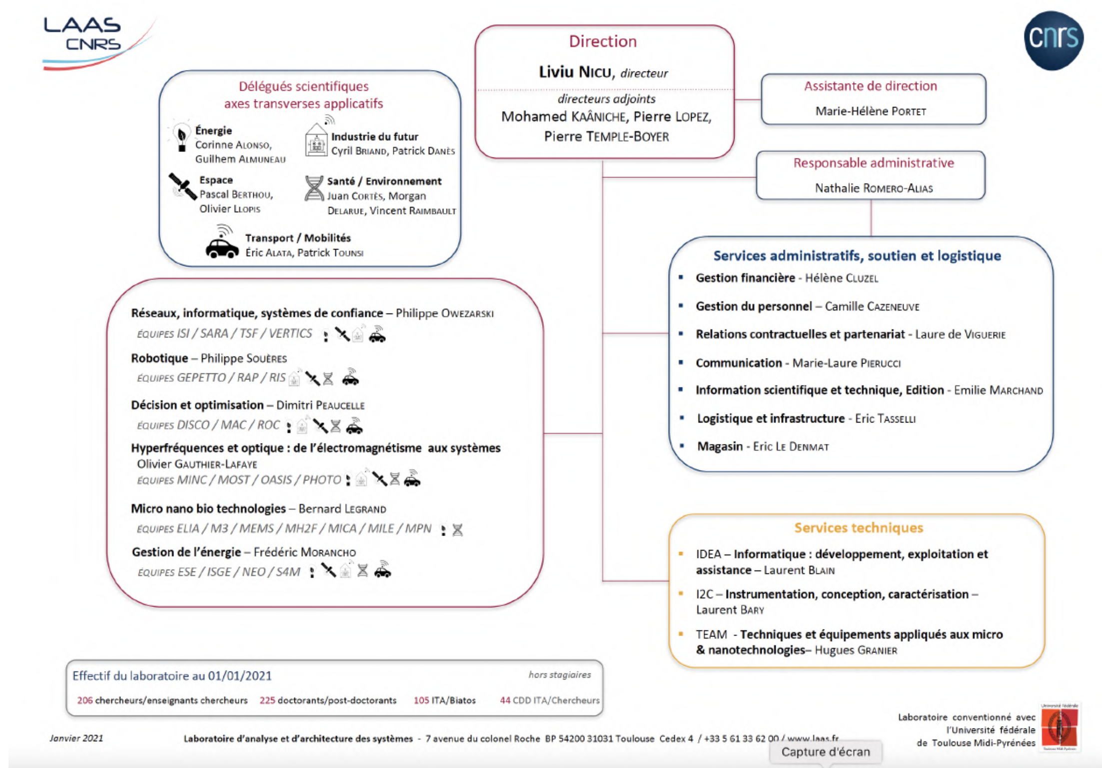
[16] ros2_control Maintainers. ros2_control Documentation, 2021. Consulté le 23/07/2021 : https://ros-controls.github.io/control.ros.org/.

[17] Olivier Stasse. Robot Operating System - Introduction. Consulté du 05/03/2021 au 09/04/2021 : https://homepages.laas.fr/ostasse/Teaching/ROS/poly-rosintro.pdf.

[18] Martin Sana. Rapport de stage de 4ème année, 2021. IPSA Toulouse.

Annexes

A. Organigrammes du LAAS-CNRS (2021)



LAAS CNRS

Informatique

Automatique

Robotique

Micro et nano systèmes

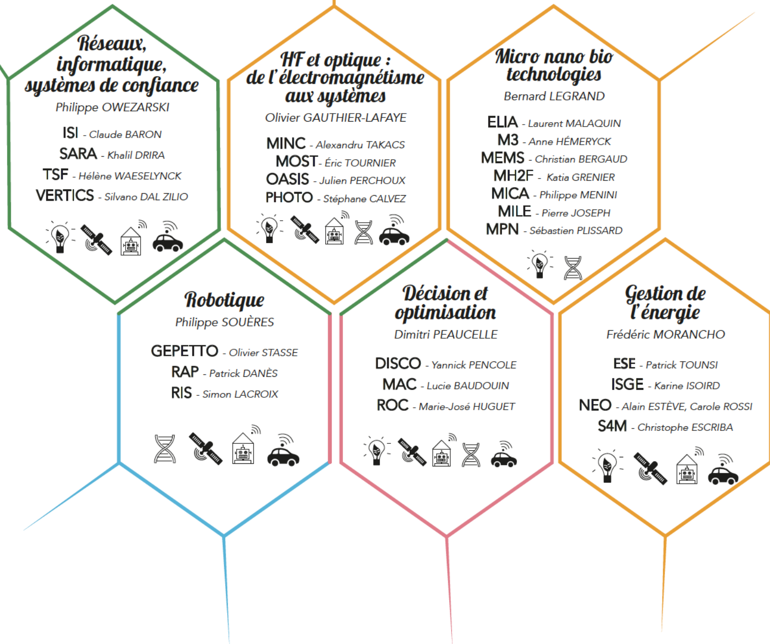
Santé / Environnement
Juan CORTÈS, Morgan DELARUE, Vincent RAIMBAULT

Industrie du Futur
Cyril BRIAND, Patrick DANÈS

Energie
Corinne ALONSO, Guilhem ALMUNEAU

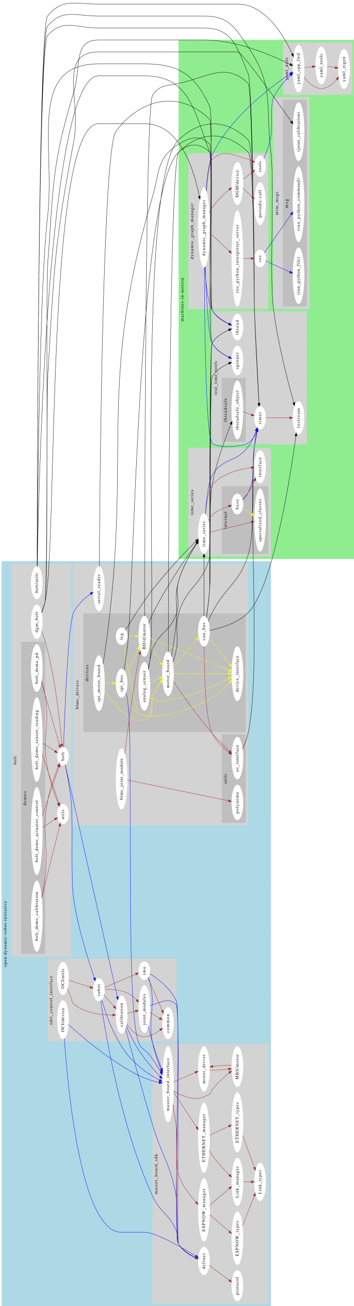
Transports / Mobilités
Éric ALATA, Patrick TOUNSI

Espace
Pascal BERTHOU
Olivier LLOPIS

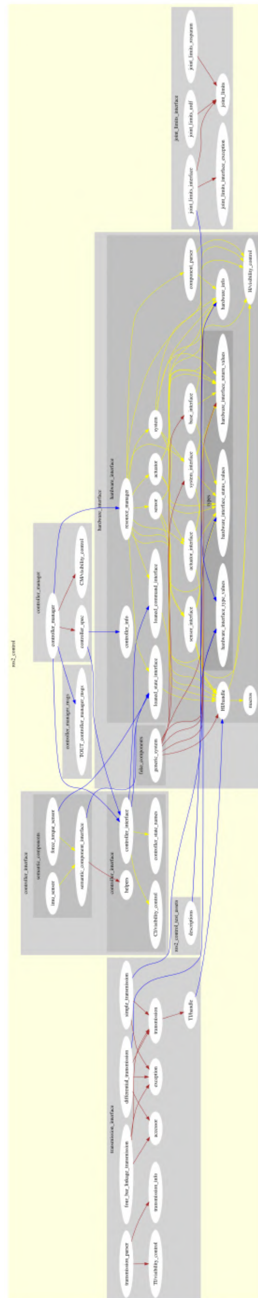


Janvier 2021

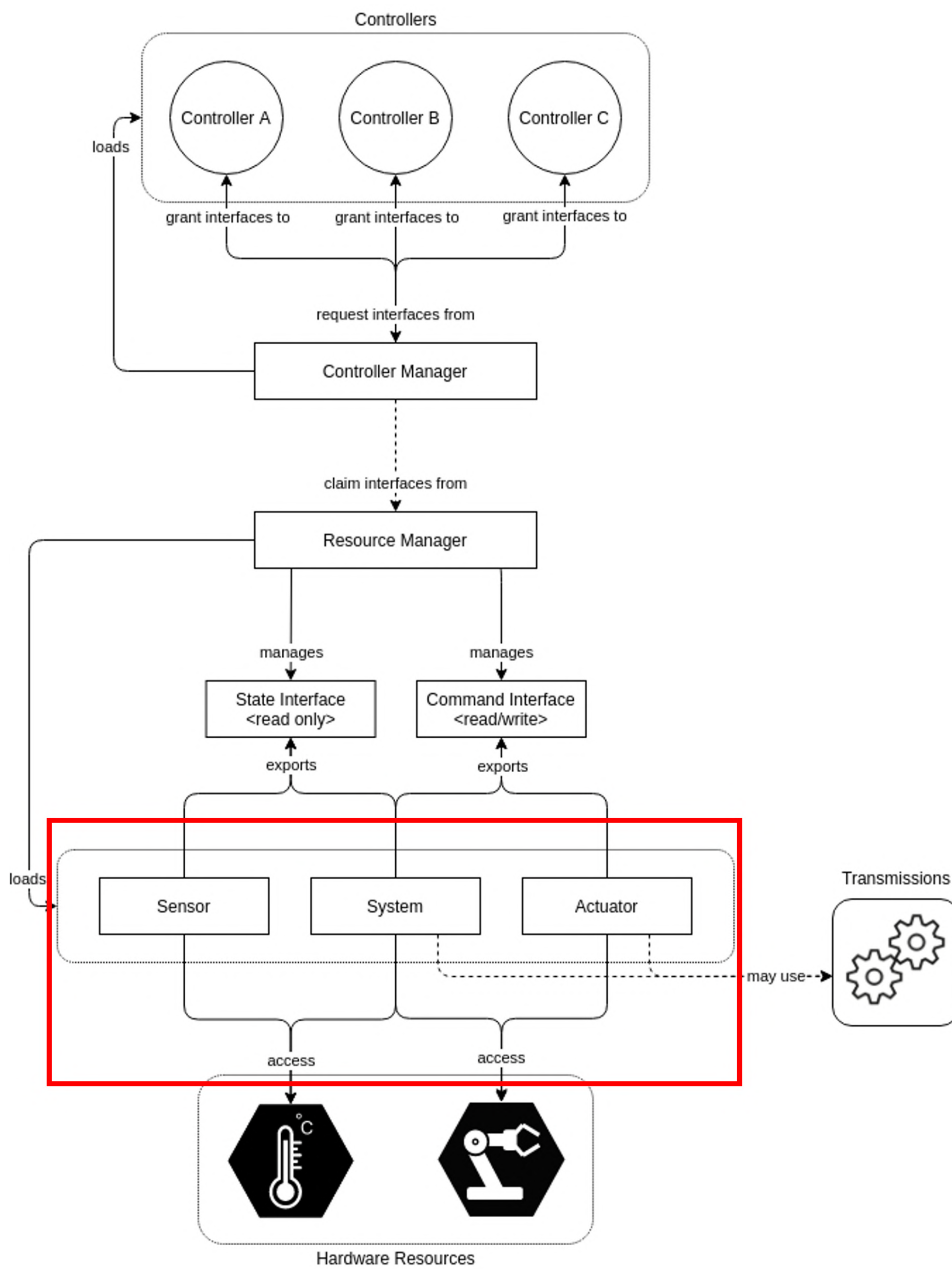
B. Arbre de dépendance simplifié des codes du Bolt



C. Arbre de dépendance simplifié de ros2_control



D. Architecture de ros2_control



E. Exemple fichier URDF

```
1   <?xml version="1.0"?>
2   <robot xmlns:xacro="http://www.ros.org/wiki/xacro">
3
4     <xacro:macro name="system_bolt" params="name prefix use_sim:=^|false use_fake_hardware:=^|true fake_sensor_commands:=^|false slowdown:=2.0">
5
6       <ros2_control name="${name}" type="system">
7
8         <xacro:if value="$(arg use_sim)">
9           <hardware>
10            <plugin>gazebo_ros2_control/GazeboSystem</plugin>
11          </hardware>
12        </xacro:if>
13        <xacro:unless value="$(arg use_sim)">
14          <hardware>
15            <xacro:if value="${use_fake_hardware}">
16              <plugin>fake_components/GenericSystem</plugin>
17              <param name="fake_sensor_commands">${fake_sensor_commands}</param>
18              <param name="state_following_offset">0.0</param>
19            </xacro:if>
20            <xacro:unless value="${use_fake_hardware}">
21              <plugin>ros2_hardware_interface_bolt/SystemBoltMultiInterfaceHardware</plugin>
22              <param name="example_param_hw_start_duration_sec">2.0</param>
23              <param name="example_param_hw_stop_duration_sec">3.0</param>
24              <param name="example_param_hw_slowdown">${slowdown}</param>
25              </xacro:unless>
26          </hardware>
27        </xacro:unless>
28
29        <joint name="FLHAA">
30          <command_interface name="position">
31            <param name="min">-0.5</param>
32            <param name="max">0.5</param>
33          </command_interface>
34          <command_interface name="velocity">
35            <param name="min">-1</param>
36            <param name="max">1</param>
37          </command_interface>
38          <command_interface name="acceleration">
39            <param name="min">-1</param>
40            <param name="max">1</param>
41          </command_interface>
42          <state_interface name="position"/>
43          <state_interface name="velocity"/>
44          <state_interface name="acceleration"/>
45          <param name="motor_number">0</param>
46        </joint>

119       <joint name="FRK">
120         <command_interface name="position">
121           <param name="min">-3.4</param>
122           <param name="max">3.4</param>
123         </command_interface>
124         <command_interface name="velocity">
125           <param name="min">-1</param>
126           <param name="max">1</param>
127         <state_interface name="position"/>
128         <state_interface name="velocity"/>
129         <state_interface name="acceleration"/>
130         <param name="motor_number">5</param>
131       </joint>
132       <sensor name="IMU">
133         <state_interface name="gyroscope"/>
134         <state_interface name="accelerometer"/>
135         <state_interface name="linear_acceleration"/>
136         <state_interface name="attitude_euler"/>
137         <state_interface name="attitude_quaternion"/>
138       </sensor>
139     </ros2_control>
140
141   </xacro:macro>
142
143 </robot>
```