28/07/2021
BETAILLE
Antonin

Compilation des comptes-rendus hebdomadaire

Projet : Bras manipulateur

SECTION : Niryo One

Déploiement du Niryo One

Mise en place du Niryo One

Préparation du robot :

- Téléchargement du manuel et suivi des instructions :

- connexions possibles de moteurs débranchés en suivant les instructions d'assemblage :

https://niryo.com/docs/niryo-one/assembly-guide/assemble-niryo-one/

- Téléchargement de NyrioOneStudio pour les intéractions avec le robot

- Connexion de Niryo au réseau Livebox-8d62 depuis Windows car la machine virtuelle ne le permet pas

- Update de l'image de la raspberry pi 3 du robot pour être sur la même version que le logiciel Studio

Prise en main du Niryo One

Utilisation de NiryoOneStudio et découverte des fonctionnalités :

→ 3 façons de programmer : via Blocky, API Python ou ROS

→ Différents outils avec de ID spécifiques

→ Un workspace pour les applications liées à la vision

→ Un mode d'apprentissage pour apprendre des différentes positions/configurations en positionnant le robot à la main

→ Calibration du Niryo conseillée manuellement

Premiers tests depuis l'application

→ Possibilité de contrôler les articulations ou les positions de l'effecteur avec gestion de l'orientation RPY (attention, le robot peut tomber s'il n'est pas fixé convenablement. De plus il ne s'arrête pas en cas de choc. Il est important de s'adapter au poste de travail)

Premier programme via l'interface Blocky

→ Le programme permet de faire du pick and place en repérant les objets à déplacer grâce à la caméra

→ Prise en main rapide et intuitive. Les fonctionnalités reprennent les fonctions de l'API

→ Il faut configurer un workspace qui doit être entièrement visible par la caméra pour y placer les objets (il faut donc prévoir une position d'observation adéquate pour repérer les objets)

→ Réalisation d'une petite vidéo montrant la séquence de fonctionnement :

https://youtu.be/Eiu68N-znYI

Recherches sur les fonctionnalités du Niryo One

Compréhension d'un démonstrateur simple en ROS:

→ Bibliothèque ROS du Niryo One : https://github.com/NiryoRobotics/niryo_one_ros

→ Démonstrateur avec le ROS stack : https://github.com/smaassen/niryo_one_tester

→ Prise en charge des déplacements compréhensible avec la bibliothèque des messages : https://github.com/NiryoRobotics/niryo_one_ros/tree/master/niryo_one_msgs

→ Pas d'exemple utilisant la caméra et le repérage d'objet avec la caméra

Reflexion sur l'utilisation de l'API :

https://github.com/NiryoRobotics/niryo_one_ros/tree/master/niryo_one_python_api

→ L'API Python permet d'aller plus vite qu'en codant en ROS mais cache les fonctionnalités liées à ROS

→ Possibilité d'avoir recours à tensorFlow pour repérer et saisir toutes sortes d'objets

https://niryo.com/docs/niryo-one/niryo-one-industrial-demonstrators/visual-picking-with-artificial-intelligence/

Problème lié au fonctionnement de la vision :

→ Tous les démonstrateurs utlisant la vision ont besoin du « workspace », c'est à dire une zone rectangulaire délimitée par des cibles faisant office d'arêtes. Cette zone peut être de différentes tailles et orientations (même à la verticale). Il faut la calibrer manuellement avec le mode d'apprentissage sur l'application NiryoOneStudio.

→ Les programmes ont besoin de vérifier la présence et la visibilité des cibles pour permettre le repérage des objets placés dans le « workspace ».

Cependant, à quoi sert réellement le workspace ? Est-il possible de s'en passer ? En effet le Niryo One n'est pas censé être embarqué sur un robot mobile donc la présence obligatoire de workspaces peut poser problème dans le fonctionnement de l'ensemble dans les scénarios en conditions réelles.

Dans le cas où nous voulons nous passer du workspace, quelles sont les méthodes de traitement de l'image à adopter pour identifier l'objet à attraper et comment positionner le bras pour l'attraper ?

Utilité et fonctionnement du workspace

Recherches concernant l'utilisation du « workspace » dans les fonctions liées à la vision → présent partout en tant qu'argument dans les définitions de fonctions
niryo_one_ros/**niryo_one_tcp_server/**

D'après le package niryo_one_ros/niryo_one_camera **et les fonctions que nous pouvons trouver pour permettre de repérer les objets, il est nécessaire de repérer les marqueurs du workspace en amont**

En contactant le service communication du Niryo One, on m'a affirmé qu'il est nécessaire d'utiliser le workspace pour être en mesure d'utiliser les fonctions liées à la vision :

« Bonjour Antonin,

Merci de votre email! J'espère que vous avez apprécié votre test de notre Niryo One.
Pour notre vision, il est indispensable et nécessaire d'avoir au moins les quatre marqueurs afin que le robot puisse se repérer.

J'imagine que cela n'est pas vraiment la réponse que vous attendiez, mais n'hésitez pas à revenir vers nous si besoin!
Je vous souhaite une très belle journée, et bon courage dans votre projet.

Aurélie »

Tentative de prise en main d'un démonstrateur ROS

Lien vers le dépôt github du démonstrateur :
https://github.com/smaassen/niryo_one_tester

Malheureusement la procédure de lancement des programmes n'est pas fonctionnelle. « catkin build » n'est pas une commande connue et catkin_make ne fonctionne pas à cause d'un problème qui (à mon avis) est lié à niryo_one_msgs

```
-- +++ processing catkin package: 'niryo_one_debug'
-- ==> add_subdirectory(niryo_one_ros/niryo_one_debug)
-- Using these message generators: gencpp;geneus;genlisp;gennodejs;genpy
-- Boost version: 1.58.0
-- Found the following Boost libraries:
--   system
--   program_options
-- Architecture: x86_64
-- wiringPi library not required
-- +++ processing catkin package: 'niryo_one_modbus'
-- ==> add_subdirectory(niryo_one_ros/niryo_one_modbus)
-- Using these message generators: gencpp;geneus;genlisp;gennodejs;genpy
-- +++ processing catkin package: 'niryo_one_rpi'
-- ==> add_subdirectory(niryo_one_ros/niryo_one_rpi)
-- Using these message generators: gencpp;geneus;genlisp;gennodejs;genpy
-- +++ processing catkin package: 'niryo_one_tester'
-- ==> add_subdirectory(niryo_one_tester)
-- Using these message generators: gencpp;geneus;genlisp;gennodejs;genpy
CMake Error at /opt/ros/kinetic/share/catkin/cmake/catkin_package.cmake:305 (message):
  catkin_package() include dir 'include' does not exist relative to
  '/home/chaire/catkin_ws/src/niryo_one_tester'
Call Stack (most recent call first):
  /opt/ros/kinetic/share/catkin/cmake/catkin_package.cmake:102 (_catkin_package)
  niryo_one_tester/CMakeLists.txt:24 (catkin_package)


-- Configuring incomplete, errors occurred!
See also "/home/chaire/catkin_ws/build/CMakeFiles/CMakeOutput.log".
See also "/home/chaire/catkin_ws/build/CMakeFiles/CMakeError.log".
Invoking "cmake" failed
chaire@chaire-VirtualBox:~/catkin_ws$
```

Illustration 1: Message d'erreur obtenue après un catkin_make

La méthode développée sur ce site : https://www.theconstructsim.com/ros-5-mins-032-compile-ros-action-messages/

bien que prometteuse, n'a pas porté ses fruits. Notamment à cause des dépendances à niryo_one_ros/niryo_one_msgs

Problème réglé en commentant le contenu de catkin_package( ) :

```
catkin_package(
#  INCLUDE_DIRS include
#  LIBRARIES robot_tester
#  CATKIN_DEPENDS roscpp
#  DEPENDS system_lib
)
```

Mise en place du node :

1) Télécharger le dossier du démonstrateur dans **~/catkin_ws/src** du Niryo

2) revenir dans **~/catkin_ws**

puis pour compiler, taper la commande: catkin_make -j2 -l2

(**https://niryo.com/docs/niryo-one/developer-tutorials/get-started-with-the-niryo-one-ros-stack/** → voir « Develop directly with ROS (if you want to get your hands dirty) »)

3) Executer : roslaunch niryo_one_bringup rpi_setup.launch
et attendre la ligne :

« [INFO] [1622191669.945850]: Action Server started (Robot Commander) »

4) Dans un nouveau terminal du Niryo, executer :
rosrun niryo_one_tester simple_command_node

5) Appuyer sur ENTER comme il est démandé après lancement du rosrun pour effectuer les actions programmées dans **simple_command.cpp**

Remarque 1 : Pour lancer l'autre programme il suffit de remplacer l'étape 4) par
rosrun niryo_one_tester tower_staking_demo

Remarque 2 : L'adaptative gripper a pour id le n°13

Nouveau problème :

Rien ne se passe, le robot n'effectue pas les commandes. Mais il y a bien un échange de données puisque ceci s'affiche quand on appuye sur enter :

[INFO] [1622187256.283988365]: Ping Tool : ping result for id (13) : 0
[INFO] [1622187256.284223724]: Set tool with id : 13
[INFO] [1622187260.383489]: Robot Action Server - Received goal. Check if exists

Cette ligne en est peut être la raison :

[WARN] [1622191665.981653477]: No geometry is associated to any robot links

Point de vue d'un ingénieur :

Lors de la réunion avec Maxime nous avons modifié le fichier /etc/hosts de la machine virtuelle en ajoutant la ligne 192.168.1.130 Niryo_MaD. De cette façon nous avons transféré le ROS Master sur la raspberry du Nyrio et nous avons la possibilité d'ouvrir rviz et rqt_graph sur la machine. De cette façon nous avons pu observer les tf des articulations du robot et constater que l'information de position est bien accessible. Avec RQT nous avons les liens entre les nodes et les topics. Nous avons pu voir que certains nodes de communiquent pas entre eux alors qu'ils le devraient. Comment cela se fait-il ? Cela reste à déterminer.

Connexion au Niryo one avec ROS

Depuis l'ordinateur :

Paramétrage du .bashrc inchangé par rapport à l'utilisation avec le turtlebot car l'ordinateur joue le rôle du ROS_MASTER

$ ssh niryo@192.168.1.130

mdp : robotics

Depuis le Niryo :

Paramétrage du .bashrc similaire à celui du turtlebot. Il faut néanmoins changer l'adresse IP du HOSTNAME par **192.168.1.130**

Raccourcis disponibles dans les terminaux

Création de 2 alias :

alias ns='cd NiryoOneStudio-linux-x64 && ./NiryoOneStudio'
→ Lancement de Niryo One Studio

alias niryo='gnome-terminal -x sshpass -p 'robotics' ssh -t niryo@192.168.1.130'
→ Connexion au Niryo One via ssh

alias niryo='gnome-terminal -x sshpass -p 'robotics' ssh -t niryo@192.168.1.130'
→ Connexion au Niryo One via ssh

alias cm='cd /home/niryo/catkin_ws && catkin_make -j2 -l2'
→ Compilation du workspace ROS

Problèmes liés à l'utilisation du Niryo

Erreur de positionnement systématique :

Pour rappel, le robot est calibré manuellement à chaque allumage. Cependant, même s'il est très précis, il est arrivé que le Nyrio se dérègle et ajoute une erreur systématique d'orientation. Il semblerait que cela concerne un offset au niveau de la première articulation (au niveau de la

base). Je ne sais pas quelle peut être son origine exacte, mais cela est lié à son déplacement entre mon bureau et le hall de l'ENSIBS.

→ Il arrive que cette erreur se résolve d'elle-même avec une recalibration automatique. Un autre moyen de corriger cette erreur consiste à manipuler le robot lorsque que le mode d'apprentissage est inactif. Il faut forcer la première articulation à revenir sur la bonne orientation.

Pince un peu faible :

Même si la pince n'est pas censée blesser l'utilisateur, il est quand même important qu'elle puisse attraper les objets du quotidien. Il faut donc que la force qu'elle exerce permette de maintenir un objet en place sans qu'il se retourne ou qu'il glisse. Or, d'après les tests que j'ai pu faire, le robot ne pourrait pas écrire avec un stylo sans que ce stylo ne glisse. Qu'en serait-il avec un mug alors ? Cela reste à tester. Si en effet la pince n'est pas assez forte, il faudra déterminer si une fonction bride la puissance des moteurs.

Streaming caméra endommagé :

Bien que je ne l'aie pas mentionné dans un rapport antérieur, la caméra installée sur le Niryo pose quelques problèmes. Jusque-là l'utilisation de la caméra était certes possible mais perturbée par des déconnexions intempestives. Mais depuis cette semaine, la situation s'est dégradée, jusqu'à Vendredi, où la caméra ne répondait plus du tout.

Le streaming vidéo n'étant plus disponible, les fonctionnalités de pick and place ne sont plus possibles. Ce qui est très problématique pour la suite du projet. Cependant je suis à peu près sûr de connaître la cause du dysfonctionnement. C'est le câble USB reliant la raspberry à la caméra qui est endommagé. Étant positionné à un endroit où il est constamment plié à cause des mouvements du robot, il s'est usé à un endroit spécifique où sa gaine est moins résistante.

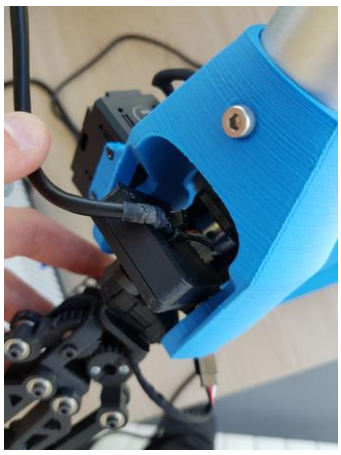

Illustration 2: Câble de la caméra abîmé

Le logiciel du robot étant capable de détecter une connexion avec la caméra, tous les branchements ne sont pas HS. Néanmoins il va falloir déterminer quel est le fil endommagé en regardant la documentation et en dénudant les fils afin d'utiliser un multimètre pour tester la continuité. Suite à quoi il faudra intervenir et arranger les branchements.

Réparation de la caméra

1ère tentative de réparation :

Étant donné que les fils du câble USB étaient seulement sectionnés, il était possible de les ressouder ensemble. Malheureusement, alors que je dénudait un fil, celui ci est sorti du connecteur PH4 dans lequel il était fixé. Ne possédant pas de cosse ni de pince permettant de remettre le fil à sa place, je me suis retrouvé avec un connecteur inutilisable.

2ème tentative de réparation :

Dans un des cartons du Niryo j'ai trouvé un connecteur identique à celui utilisé pour la caméra. Ce connecteur était lui-même relié à un autre connecteur par l'intermédiaire de 4 fils. Il était donc possible de relier le second connecteur au âble USB en faisant le lien avec des fils utilisés pour le prototypage.

J'ai donc raccordé les fils du câbles à des fils avec des embouts mâles pour les brancher au connecteur.

En testant la caméra sur le Niryo, je n'ai eu aucun résultat. Cependant, en passant sur une application web permettant de tester les caméra branchées à l'ordinateur j'ai pu constater que la caméra fonctionnait en partie. En effet il fallait que le connecteur soit orienté d'une certaine façon pour éviter les glitchs et les messages d'erreur. Or cette solution ne pouvait pas être définitive

3ème tentative de réparation :

Le connecteur utilisé pour la 2ème tentative semblait être à l'origine des problèmes de connexion. La solution s'est donc tournée vers la soudure des fils fixés au connecteur sur les rajouts du câble USB. Cela a nécessité de se séparer du deuxième connecteur qui devait être utilisé pour faire fonctionner le convoyeur livré avec le Niryo.

En fin de compte, cette solution n'a rien amélioré. Les tests de la caméra se sont soldés par un échec. Pourtant, le multimètre ne semble pas indiquer de discontinuité ou de court-circuit. Il semble cependant qu'une tension non détectée lors de la 2ème tentative soit maintenant présente au niveau du fil vert (DP)

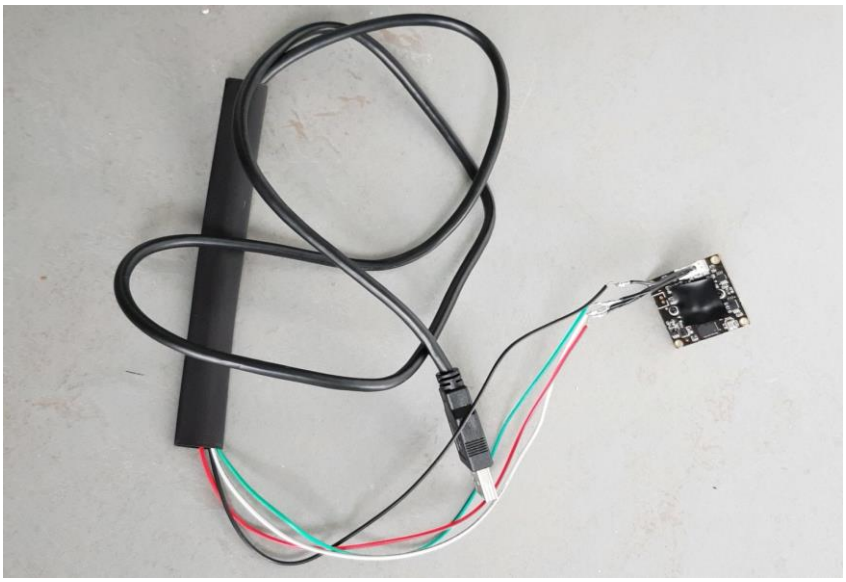

Illustration 3: Câble après la 3ème tentative de réparation (les gaines ont été retirées pour tester les soudures)

<u>Future tentative de réparation :</u>

Pour commencer il va falloir retirer les soudures proches du connecteur. Ensuite il faudra peut être revenir à la configuration de la 2ème tentative pour vérifier que ce n'est pas la caméra qui est maintenant défectueuse. Ensuite il faudra sûrement prévoir l'achat d'un nouveau câble si le SAV du Niryo ne nous en fournit pas un gratuitement.

→ Le SAV a finalement répondu de façon favorable mais le câble endommagé est toujours disponible dans un des cartons de matériel du Niryo One

Point sur les différents préhenseurs

La grande pince / « large gripper » :

Avantages :

- meilleur écartement que la petite pince

- prise légèrement plus solide que la pince adaptative

Inconvénients :

- moins bon écartement que la pince adaptative

L'électroaimant :

Avantages :

- Permet d'explorer de nouvelles opportunités dans le scénario

Inconvénient :

- Aimant bien trop faible pour tenir un mug

La petite pince / « small gripper » :

Avantages :

- aucun

La ventouse :

Inconvénient :

- Non disponible, il faudrait penser à la commander si ses caractéristiques sont satisfaisantes

La pince adaptative / « adaptative gripper »:

Avantage :

- Meilleur écartement parmi les outils

Inconvénients :

- Prise moyennement convaincante

Mise en place des fonctionnalités

Ajout de deux démonstrateurs

Encore une fois dans le cadre de la présentation pédagogique. J'ai mis en place deux démonstrateurs via l'interface blocky sur Niryo One Studio.

Le premier est un pierre-feuille-ciseau en 3 manches où les points sont comptabilisés par le robot. Pour cela le robot attrape et dépose un carré bleu lorsqu'il gagne (puis me fait un high five) et sinon il attrape et dépose un rond bleu

Le second est un petit tour où le robot fait semblant de deviner dans quelle main se situe l'objet. Il fait un petit décompte pour laisser le temps de cacher l'objet (en faisant comme une aiguille qui compte les secondes) puis pointe à gauche ou à droite. Il attend ensuite que le joueur place l'objet dans sa pince puis va le déposer devant lui.

Premiers tests de l'API avec ROS

Communication entre robots :

En observant les programmes des démonstrateurs de l'API Python j'ai pu remarquer qu'une déclaration de node est faite au début des scripts. J'ai pu vérifier par moi-même que le nom du node est bien présent lorsque que j'effectue la commande **rosnode list** après avoir fait le

bringup du Niryo. Il est donc en théorie possible de créer des topics entre le robot et l'ordinateur ou le robot et le bras

Pour l'heure j'ai voulu tester qu'il était possible de déclencher l'activation du bras depuis un évènement lié au turtlebot. Par exemple à l'activation du bouton d'arrêt d'urgence, j'ai demandé de lancer un script capable de fair le bringup du Niryo et de démarrer le démonstrateur « which_hand ». Pour faire cela j'ai écrit un fichier bash (sur l'ordinateur) activé par un roslaunch, disponible dans le package **emergency_activation**.

Résultats :

Après avoir tout mis en place j'ai pu tester le lancement d'un démonstrateur. Celui-ci met un peu de temps avant de se lancer (à cause du bringup) mais il fonctionne.

Cependant le terminal où s'affichent normalement les commandes envoyées au Niryo est pollué par des messages d'erreurs envoyés à la chaine. Je n'ai pas fait de capture d'écran mais cela ressemblait à « error : left_wheel_joint is not found » et « error : right_wheel_joint is not found ». D'après les recherches que j'ai pu faire sur ces messages, il semblerait que cela provienne du **fichier URDF du Niryo** qui ne traite pas la présence des roues. L'idéal serait de pouvoir modifier ce fichier pour faire disparaître les messages. Mais pour le moment je ne sais pas comment le trouver. Je dois faire appel à Maxime pour m'aider **(cela n'a jamais été fait)**

Remarque sur la création des scripts :

Tous les programmes réalisés avec l'interface Blocky peuvent être traduits pour être exploitables par l'API Python. Il ne reste alors plus qu'à ajouter les fonctionnalités de ROS telle que la création d'un node, d'un publisher ou d'un subscriber afin de rendre le script compatible avec le fonctionnement général du système.

J'ai alors pu supposer que le Nyrio serait capable de couper les moteurs du turtlebot lorsqu'il serait en fonctionnement et pourrait les réactiver à la fin de son application de pick and place.

Fonctionnement et utilisation du package niryo_control

Le node /niryo_control

Le node **/niryo_control** est le node qui permet de communiquer avec le niryo et l'interface web. Il repose sur un fonctionnement similaire à celui de **/simple_navigation_goals**. D'une part, des subscribers permettent de recevoir les informations des boutons de l'interface web et du robot. D'autre part, des publishers permettent de communiquer les messages sur l'interface web et de mettre sur pause le robot.

Un **switch** avec des étapes correspondant à chaque interaction avec les boutons permet de gérer l'envoi et la réception des messages.

Lancement des scripts des actions

Les scripts en question sont situés directement sur le Niryo :

Chemin vers les scripts : catkin_ws/src/niryo_one_python_api/examples

Il faut donc être en mesure d'activer ces programmes depuis le node **/niryo_control** qui lui est situé dans le catkin_ws de la machine virtuelle.

Pour cela j'ai donc créé des fichiers **.launch** pour lancer des **scripts bash** executant les programmes voulus à l'aide de la fonction **sshpass**

Il suffit donc d'utiliser la commande

```
system("gnome-terminal -x roslaunch niryo_control activ_arm_servir.launch");
```

ou

```
system("gnome-terminal -x roslaunch niryo_control activ_arm_ranger.launch");
```

En fonction de la fonction demandée

Remarques sur les lancements des nodes

→ Il a été nécessaire de modifier le node de navigation du turtlebot pour surpprimer les terminaux lorsque nous tuons le node. Sinon les terminaux s'accumulaient

→ Après le bringup du Niryo, le terminal de lancement se remplit de messages d'erreurs liés à l'URDF. J'en avais déjà fait mention mais cette fois-ci j'ai pu constater que ces messages pouvaient saturer la capacité des logs . Il faut alors effectuer un **rosclean purge** pour supprimer les logs. Cependant cette solution n'est que temporaire puisque les logs se remplissent en permanence dès que le système général est en marche.

Principe des « actions »

Les scripts en Python

Ces scripts correspondent à des programmes réalisés avec blocky. Il en existe 2 pour le moment :

- **ranger.py** : Le bras saisit un objet sur le workspace et le dépose dans le meuble

- **servir.py** : Le bras sort un objet du meuble pour le déposer sur le workspace

Les deux programmes réalisent une fonction différente mais fonctionnent sur le même principe. Ils ont été modifiés pour fonctionner avec ROS. Ce sont donc des nodes avec des publishers et des subscriber. Cela permet à la fois de recevoir les informations concernant les pauses et les arrêts mais aussi d'informer le node de contrôle que l'action est terminée.

Principe de fonctionnement

Tout d'abord le programme commence par une calibration automatique. Si la calibration a déjà été faite elle ne recommence pas. Ensuite entre chaque mouvement du robot, une fonction permet d'analyser l'état de la consigne pour savoir s'il faut mettre le robot en pause ou l'arrêter. Si La classe **Listener** de change pas de valeur d'élément alors le robot passe au mouvement suivant jusqu'à la fin du programme où la fonction **talker** annonce la fin de l'action.

Mise en place des « actions »

Reprogrammation des actions

En prévision de la visite des représentants d'Orange, j'ai dû redéfinir les mouvements du robot pour intégrer les actions de pick and place. Les scripts ranger.py et servir.py réalisent maintenant réellement les fonctions qui leur sont attribuées. Je suis donc passé par l'interface

blocky pour programmer puis j'ai enregistré la séquence pour que Niryo One Studio me propose d'afficher la conversion en python qui peut être utilisée par l'API.

J'ai notamment ajouté la fonction suivante :

```
while not (n.vision_pick('default_workspace', 0, "CIRCLE", "BLUE")[0]):
        n.wait(1)
        pause(n,interaction)
```

Qui me permet de récupérer un palet bleu sur le workspace par défaut (que j'ai défini) avec un offset de 0 mm

Lancement de /niryo_control après un arrêt d'urgence

Pour une raison qui m'échappe, l'arrêt d'urgence devant concerner le turtlebot uniquement arrêtait le node de contrôle du Niryo.

J'ai donc ajouté un rosrun dans le node /emergency_activation pour faire en sorte que /niryo_control se relance à tous les coups

Activation à l'initialisation

J'ai créé un nouveau programme intégré au Niryo qui est lancé de la même façon que les autres à partir d'un roslaunch activant un fichier bash qui contacte le Niryo pour exécuter le programme **activation.py.**

Ce petit programme est lancé dans le fichier **turtle.sh** après le bringup du niryo. Il permet de réaliser la calibration automatique du robot au démarrage

→ Ancienne version de fonctionnement

Améliorations à suivre

Actuellement, les actions sont dans des programmes distincts. Pour les exécuter il faut effectuer une connexion ssh pour communiquer avec le Niryo. Pour plus de fluidité, il

faudrait fusionner les deux programmes et les transformer en fonctions qui seraient activées grâce à un topic. De cette façon il n'y aurait plus besoin de quitter le programme à chaque fin d'action.

Correction de la trajectoire du bras

Utilisation du workspace

Amélioration attendue depuis longtemps, la correction de la trajectoire du bras en cas de mauvais positionnement de la base mobile, a enfin sa première version.

Pour rappel, les contraintes que sont la présence d'un workspace et le positionnement approximatif de la base mobile influe sur la capacité des fonctions du Niryo à réaliser la préhension d'un objet. En effet, si la position relative de la base du bras par rapport au workspace ne correspond pas à celle prise en compte par le robot pendant la configuration du workspace, la préhension n'aboutit pas. La pince vient se fermer à une distance correspondant à l'erreur de positionnement du positionnement de la base mobile.

Alors, puisque le workspace est nécessaire pour utiliser les fonction du Niryo, je me suis servi de celui-ci pour aboutir à une solution. Il avait été pressenti d'estimer le déplacement du workspace pour venir ensuite modifier sa configuration avec les nouvelles positions des marqueurs. Cependant, bien que la détection des marqueurs était déjà possible, l'utilisation de ces coordonnées était soit trop complexe soit impossible pour redéfinir le workspace.

J'ai donc décidé de jouer sur la position du bras lorsque l'objet est détecté. Puisque l'erreur de position de préhension correspond à la translation du workspace par rapport au workspace de référence, il suffit de retrancher cette erreur à la position estimée pour retomber sur la bonne position.

Fonctionnement de la première correction

Pour réaliser la correction j'ai récupéré les fonctions de détection des marqueurs pour estimer la translation des marqueurs avec OpenCV. J'ai d'abord pris une photo du

workspace lorsque le bras est parfaitement positionné grâce à sa caméra embarquée : c'est la photo témoin. Ensuite, avant de pouvoir utiliser le bras j'ai demandé à prendre une nouvelle photo pour comparer la position des marqueurs. De cette façon j'ai fait une moyenne des translations pour obtenir la translation à retrancher à la position du préhenseur. Cette correction me permet de palier à des problèmes d'erreur dans le cas où l'erreur de positionnement se fait par translations dans le plan du workspace.

Problème de rotation

Lorsque l'erreur de positionnement a pour origine une rotation selon a normale au workspace, le problème est un peu plus compliqué à gérer. Il ne suffit plus seulement de calculer la translation pour corriger la position du préhenseur car le repère a été changé. Alors plutôt que de passer par une solution à base de matrice de rotation j'ai préféré opérer en deux temps. Tout d'abord, corriger la rotation au niveau du turtlebot et ensuite laisser la première correction finir le travail. Pour cela j'ai estimé l'erreur d'orientation de la base mobile à l'aide de la caméra du Niryo pour ensuite faire tourner le niryo d'un angle équivalent dans le sens inverse pour corriger l'erreur.

Fonctionnement de la correction de la deuxième erreur

L'erreur d'orientation est calculée grâce à la fonction **arctan2** qui me permet de connaître l'angle dirigé entre 2 vecteurs. Les deux vecteurs de référence sont :

- le vecteur défini par les coordonnées des deux marqueurs inférieurs du workspace

- un vecteur arbitraire (x;0)

Le calcul qui en découle est le suivant :

```
angle = atan2(vector2.y, vector2.x) - atan2(vector1.y, vector1.x);
```

J'utilise ce calcul avec la photo témoin puis avec la photo de test avant l'action du bras.

Une fois la valeur de l'angle de la photo de test connue je l'utilise dans une fonction permettant au turtlebot de se réorienter en publiant sur le topic **/cmd_vel**. En connaissant la vitesse de rotation du turtlebot je peux lui ordonner d'effectuer une rotation pendant le temps nécessaire pour effectuer la correction en orientation. Une fois la rotation terminée je demande au programme d'appliquer la correction en translation.

Fluidité de communication

Unification des scripts du Niryo One

Plus haut, j'expliquais que j'avais un script pour chaque action : ce n'est désormais plus le cas.

Le principe était le suivant :

- clic sur un bouton

- communication de l'information au node **/niryo_control**

- dans le switch, appel du roslaunch de l'action

- le roslaunch lance le fichier bash de l'action

- le fichier bash effectue une connexion ssh avec le Niryo puis demande l'exécution du programme python de l'action.

- le programme vérifie la calibration puis le bras se met en mouvement

- à la fin de l'action le programme publie sur un topic que l'action est finie

- le programme se termine, quittant le terminal et coupant la connexion ssh

Ce procédé était fonctionnel mais à chaque appel de programme, il fallait attendre plusieurs secondes avant que le bras ne se mette à bouger. En effet, les connexions ssh, bien que relativement rapides, ralentissaient quand même la mise en marche du bras.

J'ai d'ailleurs tenté d'accélérer un peu la procédure de cette façon :

https://linux-tips.com/t/disabling-reverse-dns-lookups-in-ssh/222

C'est pourquoi j'ai converti les programmes **ranger.py** et **servir.py** en un seul programme où les actions sont traduites par des fonctions. Ce changement a également rendu le programme **activation.py** (qui lance la calibration au lancement du système) obsolète puisque toutes les actions relatives au Niryo ont été centralisées.

J’ai donc changé quelques principe de fonctionnement pour que certaines communications se fassent à l’intérieur du programme et plus par l’intermédiaire du node **/niryo_control**

Implémentation de la correction

La correction a été intercalée entre la calibration et la boucle while dans laquelle le Niryo attend ses commandes.

Elle prend également la forme d’une boucle while qui commence par la prise du photo de test permettant de vérifier les erreurs de positionnement. S’il y a une erreur de rotation en fin de première boucle, la base mobile se réoriente. Puis une nouvelle photo est prise et les valeurs de translations sont recalculées avant d’être utilisée en argument des fonctions ranger et servir

Application de la correction

Afin de permettre la correction de l’erreur de position pendant la préhension il suffit d’utiliser les coefficients obtenus pendant la phase d’initialisation du programme **actions_niryo_final.py**. Ces coefficients sont la translation en x et en y dans le plan du workspace. Ils sont mis en argument des fonctions **servir()** et **ranger().**

Dans la fonction **ranger()**, ils servent à ajuster la valeur de la position relative de l’objet par rapport au workspace dans la fonction **get_target_pose_from_rel()**.

« Given a pose (x_rel, y_rel, yaw_rel_) relative to a workspace, this function returns the robot pose in which the current tool will be able to pick an object at this pose. The height_offset argument (in m) defines how high the tool will hover over the workspace. If height_offset = 0, the tool will nearly touch the workspace. »

De plus ils permettent de modifier la position de dépôt de l’objet. Mais cette fois-ci, c’est la position dans le référentiel du robot qu’il faut modifier car **move_pose()** est utilisé. Il faut donc inverser le x et le y et multiplier par un correcteur de 1.5/1000 pour arriver à la position attendue.

Dans la fonction **ranger()** qui n’utilise pas la vision, le même fonctionnement est appliqué pour déposer l’objet sur le meuble ou sur le workspace.

Limitation de la correction

Par manque de temps, tous les tests nécessaires pour vérifier la capacité de la correction à être appliquée en toutes circonstances n'ont pas pu être réalisés. Il semblerait que certaines erreurs de positionnement empêchent de réaliser certains mouvements. Je n'ai cependant pas pu estimer dans quelle mesure le problème apparaissait. Je sais seulement que si l'estimation de la position du robot sort du cadre des positions possible, l'action est bloquée.

D'autre part, il est important de noter que si la translation en « y » est trop importante alors le bras se déploie plus loin vers l'avant et se retrouve susceptible de tomber. Après intégration à la base mobile, il serait alors possible que le bras fasse basculer tout l'ensemble

La fonction d'observation

Fonctionnement général

La fonction d'observation n'est pas nécessaire dans le scénario d'utilisation du bras. Cependant, ayant plusieurs applications possibles en cas de problème de navigation ou tout simplement pour observer son environnement à distance (observation depuis une fenêtre, chercher ses lunettes, regarder sous les meubles), j'ai finalement décidé de l'implémenter.

Pour réaliser une observation il faut passer par le menu du bras, car c'est sa caméra que nous utilisons. J'ai choisi de ne pas passer par un joystick parce qu'il est impossible d'effectuer des mouvements successifs avec une fluidité satisfaisante. J'ai donc opté pour un système de déplacements avec des flèches directionnelles sur lesquelles il faut appuyer pour réaliser des petits déplacements de la caméra.

Explications sur les commandes

Figure 1: Screenshot de l'interface web : Partie dédiée à l'observation

Il y a en tout 6 boutons :

- Un bouton d'activation, coloré et rectangulaire, situé au dessus des boutons liés au déplacement. Il y est écrit « Démarrer l'observation ». En appuyant sur ce bouton, nous activons la fonction **observation()** dans le script du Niryo. Cette fonction prend en charge les déplacements du robot en écoutant directement les informations transmises depuis l'interface web. Au début de cette fonction, le bras prend sa position d'observation, que j'appellerai « position d'origine ». Dans cette position le tangage, le roulis et l'assiette de la caméra sont proches de 0.

- Un bouton spécial, représenté par le symbole de maison du logo de la chaire. Situé au milieu des flèches, il permet de revenir à la position d'origine à n'importe quel moment

- Une flèche pointée vers le haut, pour faire tourner la 4^{e} articulation du Niryo influant sur le tangage de la caméra. Ce bouton permet d'augmenter d'augmenter l'angle de tangage.

- Une flèche pointée vers le bas pour diminuer l'angle de tangage.

- Une flèche pointée sur la gauche, pour faire tourner la 1re articulation du Niryo. Le moteur étant situé à la base du robot, il nous permet de contrôler directement l'azimut du bras. Ce bouton nous permet d'augmenter l'azimut.

- Une flèche pointée vers la droite pour diminuer le lacet

Le robot ayant une butée à 180° et à -180° sur ses articulations il n'est pas possible de laisser les angles de lacet et d'azimut sans valeurs limites. C'est pourquoi chaque flèche directionnelle permet d'effectuer un déplacement à condition que les valeurs limites ne sont pas dépassées.

Instabilité de la position entre 2 mouvements

En travaillant sur la fonction d'observation j'ai pu constater un manque de maintien de la part de la 4e articulation dans le maintient de l'angle de tangage de la caméra. En effet, à chaque clic sur une des flèches latérales, la caméra descendait de quelques degrés, d'autant plus lorsque l'angle de tangage était élevé. Cela rendait impossible le balayage visuel lorsque la caméra pointait vers le haut.

Pour palier à ce problème j'ai changé la façon dont la fonction de mouvement associée à la caméra, **move()**, reçoit et effectue le mouvement du bras. Auparavant, un clic sur une des flèches se traduisait par un incrément selon le tangage ou l'azimut réalisé par un seul moteur à la fois. La position des moteurs était récupérée au début de la fonction à partir de la position en fin du mouvement précédent. Or, puisque l'angle de tangage ne pouvait pas être maintenu, l'erreur angulaire s'accumulait sans avoir de correction. C'est pourquoi désormais la commande de position angulaire du bras est renvoyée à chaque fin de fonction. Ainsi, même si la position réelle du bras en fin de fonction n'est pas en accord avec la consigne, l'erreur est en partie compensée à chaque appel de la fonction.

Gestion du problème d'URDF

Messages d'erreur

Lorsque le bringup du Niryo est effectué après le bringup du turtlebot (fonctionnement normal lors du démarrage du système) des messages d'erreur s'affichent dans le turminal.

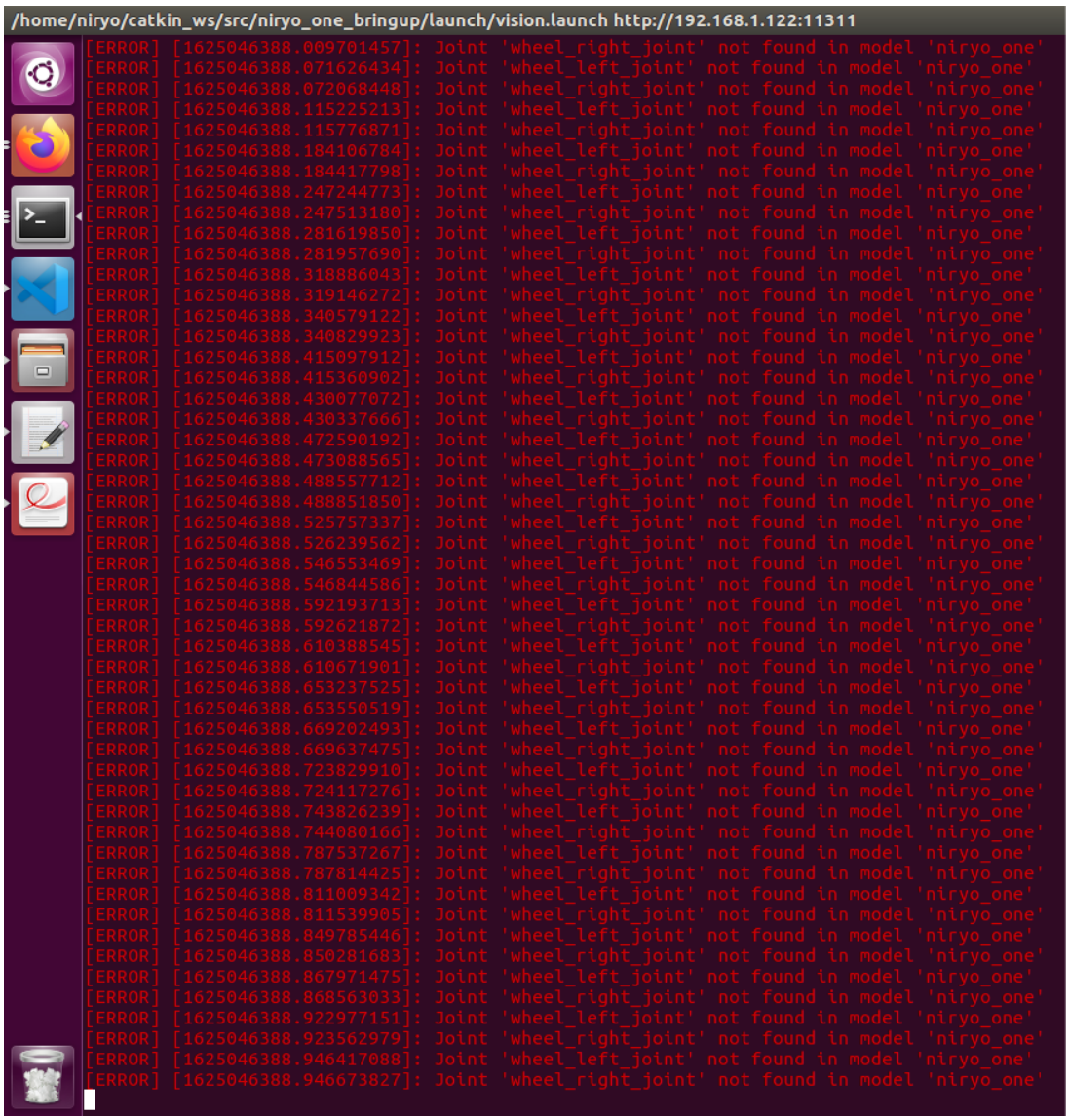

Illustration 4: Affichage des messages d'erreur dans le terminal du bringup du Niryo

Il est évident que la collaboration entre le turtlebot et le Niryo est à l'origine de ces messages.

D'après ce lien

https://answers.ros.org/question/62612/suppressing-wheel-joint-errors-when-using-moveit/

il semblerait que Moveit (qui est le package permettant de commander le bras) soit à l'origine de cette erreur.

La démarche recommandée est alors de modifier l'URDF et le SRDF du Niryo pour signaler au robot que « wheel_left_joint » et « wheel_right_joint » font partie du bras mais sont passifs.

<passive_joint name="wheel_left_joint" />

<passive_joint name="wheel_right_joint" />

→ Déclaration dans le SRDF

~/catkin_ws/src/niryo_one_moveit_config/config/niryo_one_srdf

→ emplacement du SRDF du Niryo

~/catkin_ws/src/niryo_one_description/urdf/v2/niryo_one.urdf.xacro

→ emplacement de l'URDF du Niryo

Dans l'URDF, j'ai essayé de replacer la même configuration que dans l'URDF du turtlebot (même type de chemin). J'ai donc repris les joint et leurs child link qui sont nécessaires pour la déclaration des joints.

Malheureusement même si cette solution ne posait pas de problème lorsque le bras était activé seul, le fonctionnement général du système était HS car la définition du bras n'était plus reconnue.

Je suis donc revenu à la configuration originale avec les messages d'erreur car cela n'empêche pas le système de fonctionner