Filière Électronique - module PR214
Projet thématique - RoboCup ligue @Work

AW-2 : Mise en œuvre du bras XArm6 pour la ligue AtWork

Élèves :
Marius BARRAIS
Thomas NIEDDU

Enseignant :
Pierre MELCHIOR

29 mai 2021

Table des matières

1 Introduction

Dans le cadre de la ligue AtWork de la RobotCup, la base roulante est accompagnée d'un bras robot autonome équipé d'un préhenseur ainsi que d'une caméra à profondeur de champ. Ces deux outils (bras + préhenseur et caméra) sont exploitables via les logiciels fournis par les deux différents constructeurs, appelés IDE.

Malheureusement, dans notre cas, ces IDE ne permettent pas directement de traiter les données captées par la caméra et d'envoyer au bras robot les ordres précis dans le but de le faire se déplacer dans l'espace et effectuer des actions.

Nous allons donc devoir coordonner ces deux systèmes en utilisant les SDK (Software Development Kit ou Kit de Développement Logiciel en français) fournis par chacun des constructeurs. Un SDK est un ensemble d'outils logiciel à destination des développeurs dans le but de leur faciliter la tâche.

- Un SDK est conçu pour :
  - un ou plusieurs langages de programmation
  - une ou plusieurs cibles (plateforme, jeux vidéo, etc.)
- Un SDK comporte au moins :
  - Un traducteur (compilateur, interpréteur, etc.) qui traduit le fichier source en langage de programmation vers un fichier destination en langage machine
  - un éditeur de liens (lieur) qui lie ensemble les différents modules et les bibliothèques de routines en un seul fichier exécutable (programme)
  - des bibliothèques contenant des routines déjà prêtes à l'utilisation.

Le SDK du bras de chez UFactory est disponible en Python, en R.O.S. (Robot Operating System) ou en `C++`. Le SDK Realsense de chez Intel quant à lui est bien plus polyvalent (Python, R.O.S, Unity, Unreal Engine, `C++`, `C#`, ...). Nous avons opté pour le langage de programmation `C++` car il est plus bas niveau et permet de piloter bien plus aisément le matériel. Aussi la documentation de la part d'Intel ainsi que les ressources partagées par la communauté des développeurs en `C++` sont bien plus fournies.

Le but de ce rapport va donc être de détailler les différentes étapes de l'installation des ces SDK et leur fonctionnement ainsi que d'expliquer le code que nous avons élaboré dans le but de déplacer le bras robotique afin qu'il saisisse un objet devant lui et le dépose à un autre endroit.

La passation de connaissances et de compétences étant l'un des principaux enjeux de ce projet, nous adopterons une approche pédagogique et didactique. Notre objectif majeur sera donc d'accompagner les prochains étudiants travaillant sur ce projet par nos avancées afin qu'ils puissent aller encore plus loin.

2 Description du matériel

2.1 Le bras robotique de chez UFactory

Le bras robotique XArm6 est un bras comportant 6 axes indépendants ayant un rayon d'action de 70cm, supportant une charge de travail de 5kg. Il est associé à un boîtier de contrôle pouvant être alimenté sur le secteur ou sur batterie. Ce boîtier de contrôle communique par la biais d'une liaison ethernet à un contrôleur. Dans nôtre cas, ce contrôleur sera un ordinateur sous Linux.

Ce bras robotique est équipé d'un préhenseur, ou XArm Gripper, permettant d'attraper des objets pour les déplacer avec le bras. Cette pince possède un écartement maximal de 86mm et une force maximale de préhension de 30N. Cet outil est connecté au bras par le biais d'un câble et se pilote de la même façon que le XArm6.

FIGURE 1 – Bras robotisé XArm6 de chez UFactory

2.2 La caméra de profondeur de chez Intel

La Caméra est une Intel RealSense D435 à profondeur de champ. Cette caméra est équipée de 3 capteurs d'image :

- Un projecteur infrarouge
- Deux récepteurs infrarouges d'une part et d'autre de la caméra
- Un module RGB Full-HD (1920*1080 pixels à 30 images par secondes)

Ces 3 capteurs permettent à la D435 d'obtenir une information de distance sur un champ de vision de 86°x57° avec une définition d'image maximale de 1280*720 pixels à 90 images par secondes. Concernant les capteurs infrarouges, ils permettent de mesurer des distances allant de 28cm à 10m avec une erreur de précision inférieure à 2% à 2m.

FIGURE 2 – Caméra D435 Realsense de chez Intel

2.3 La pièce support de la caméra

Une fois le modèle 3D designé par les enseignants de l'école ESTIA en notre possession, nous l'avons imprimé au FabLAB de l'école : un atelier mettant à notre disposition une découpeuse laser, des imprimantes 3D et pléthore d'outils. Cette impression a duré 3h45 et a malheureusement nécessité une modification afin qu'elle puisse tenir sur le bras robotisé. En effet, en l'état, la pièce en bleue sur la Figure 3 ne peut pas est fixée sur le bras. Il nous a fallu percer le bas de la pièce de manière à le maintenir en place avec une vis plus longue que celle initialement installée sur la pièce circulaire à l'interface entre le bras et le gripper.

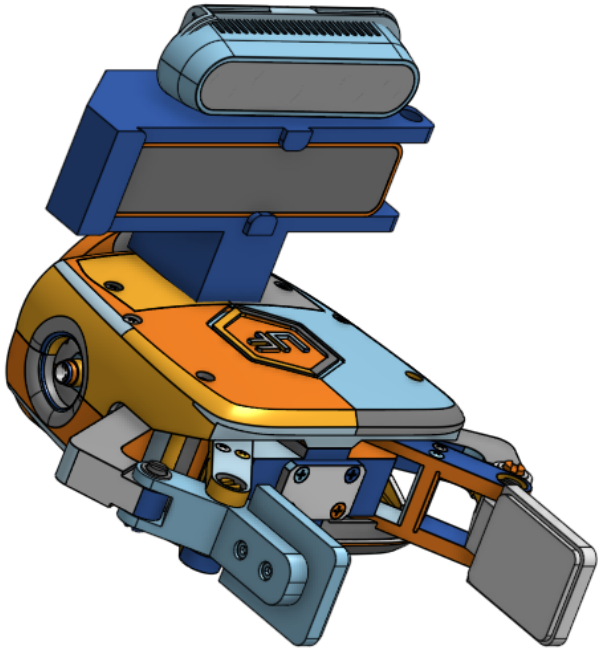


FIGURE 3 – Pièce imprimée en 3D au FabLAB de l'école (bleue) servant de support à la caméra

3 Pré-requis

L'installation des librairies pour contrôler ces équipements nécessite d'avoir exécuter la commande suivante :

```
» sudo apt update && sudo apt upgrade && sudo apt autoremove
-Mise à jour de l'ordinateur-
```

Il est aussi nécessaire d'installer le logiciel CMake. Le téléchargement et le descriptif de la procédure d'installation est disponible à l'adresse suivante :

```
https://cmake.org/install/
```

4 Bras robotique XArm6

4.1 Communication avec le PC

Comme dit précédemment, le bras XArm6 ainsi que le préhenseur (ou gripper), se commandent par le biais d'une liaison ethernet. La procédure pour connecter le bras et le PC est donc la suivante :

- Mettre sous tension le bras en allumant la station de contrôle et en réarmant de l'arrêt d'urgence (bouton "coup-de-poing" rouge de sécurité)
- Brancher un câble ethernet entre la prise LAN de la station et le port ethernet de l'ordinateur
- Configurer l'adresse IP de carte ethernet de l'ordinateur en statique à une des adresses du réseau "192.168.1.0" (Toutes devraient être disponibles sauf 208 (bras robot) et de 255 (adresse de broadcast))

Une fois ces étapes de réalisées, le programme qui tentera de se connecter au bras robot le fera au travers de cette connection ethernet. Il faudra cependant retenir l'adresse IP du bras robot : 192.168.1.208 dans le cas du bras robotique présent à l'école. Si cette adresse ne fonctionne pas, utiliser la fonction "Search server" du logiciel XArm Studio ou bien trouver manuellement l'adresse du bras robotique (scan exhaustif du réseau à l'aide de commande ping).

4.2 Installation du SDK

La méthode décrite içi concerne l'installation sur le système d'exploitation UBUNTU. Si vous souhaitez installer le SDK sur un autre support, veuillez vous référer à la page suivante : `https://github.com/xArm-Developer/xArm-CPLUS-SDK`.

L'installation des librairies, des exemples et des binaires du SDK est un passage obligé pour permettre à l'ordinateur de commander le bras. Sur linux, les fichiers nécessaires à l'installation sont téléchargeables via la commande :

```
» git clone https://github.com/xArm-Developer/xArm-CPLUS-SDK.git
```

Une fois le téléchargement terminé, ouvrez un terminal dans le dossier <xArm-CPLUS-SDK-dev> et tapez les commandes suivantes pour construire les librairies et les installer sur l'ordinateur :

```
» make
» sudo make install (nécessite les droits super-utilisateur)
```

Après cette étape, les librairies devraient être correctement installées et disponible pour le compilateur GCC.

4.3 Programmation et Compilation

Une fois l'étape d'installation des librairies réalisée, vous allez pouvoir envoyer des ordres au bras XArm6 par le biais de codes écrits en `C++`.

Des exemples disponible localisés dans le dossier "example" du SDK peuvent être compilés à l'aide de la commande suivante :

```
» make test
```

Cette commande créera un dossier "build" dans le dossier de la SDK qui contiendra les librairies et les codes exemples compilés.
Enfin, lancez l'exemple "0002-get_property" pour tester la communication avec le bras :

```
» sudo ./build/example/0002-get_property 192.168.1.208
(nécessite les droits super-utilisateur)
```

Vous devriez retrouver les résultats obtenus en Figure 4

```
terarys@marius-PC:~/Projet_S8/Xarm/xArm-CPLUS-SDK-dev$ sudo ./build/example/0002-get_property 192.168.1.208
SDK_VERSION: 1.6.0
Tcp control connection successful
is_old_protocol: 0
version_number: 1.4.1
Tcp report connection successful
[set_state], xArm is ready to move
========================================
default_is_radian: 0
version: v1.4.1
state: 2
mode: 0
cmd_num: 0
error_code: 0
warn_code: 0
collision_sensitivity: 1
teach_sensitivity: 1
world_offset: 0.000 0.000 0.000 0.000 0.000 0.000
gravity_direction: 0.000 0.000 -1.000
==========TCP==============
* position: 193.951 -115.863 439.081 -4.040 -50.260 154.620
* tcp_jerk: 7000.000000
* tcp_load: 0.000 0.000 0.000 0.000
* tcp_offset: 0.000 0.000 0.000 0.000 0.000 0.000
* tcp_speed_limit:0.100 1000.000
* tcp_acc_limit:1.000 50000.000
===========JOINT==============
* angles: -30.957 -23.707 -12.556 -0.255 -93.360 3.548 0.000
* joint_jerk: 11459.156250
* joint_speed_limit:0.573 1145.916
* joint_acc_limit:0.573 180.000
* joints_torque:-0.000 -5.195 -9.498 -0.107 0.198 0.009 0.000
```

FIGURE 4 – Test de connexion et lecture de télémétrie

5 Caméra de profondeur Intel RealSense D435

5.1 Communication avec le PC

La caméra de profondeur Intel RealSense D435 est pilotée via USB. Pour cela, il est nécessaire que l'ordinateur soit équipé d'un port USB de dernière génération, à savoir, un port USB 3.0 (minimum). La caméra utilisera donc un port USB et le bras robotique utilisera le port Ethernet.

5.2 Installation du SDK

Plusieurs solutions s'offrent à nous pour le choix du SDK. Le SDK de la caméra a été développé pour une multitude de plateformes telles que :

- Ubuntu 16.04/18.04/20.04 LTS
- Windows 10 (Build 15063 ou plus récent)
- Windows 8.1
- Windows 7
- Mac OS (High Sierra 10.13.2)
- Android 7, 8

Le système d'exploitation et le langage de programmation ayant déjà été choisis lors de l'installation du SDK du bras robotique, nous installerons donc le SDK prévu pour un système d'exploitation Linux pour un langage `C++`.

Toutes les ressources se trouvent à l'url suivante : `https://github.com/IntelRealSense/librealsense/releases`.

Voici donc les étapes pour installer le SDK version `C++` pour Ubuntu 18.

5.2.1 Pré-requis

1. Se placer à la racine du dossier *librealsense*. (La caméra n'est pas branchée pour l'instant)
2. Installer les packages principaux requis pour construire les binaires librealsense et les modules du kernel concernés:

— `» sudo apt-get install git libssl-dev libusb-1.0-0-dev pkg-config libgtk-3-dev`

— `» sudo apt-get install libglfw3-dev libgl1-mesa-dev libglu1-mesa-dev`

3. Installer les scripts de permission d'Intel Realsense situés dans le répertoire source de librealsense :

— `» sudo cp config/99-realsense-libusb.rules /etc/udev/rules.d/`

— `» sudo udevadm control - -reload-rules && udevadm trigger`

4. Construire et appliquer les modules du noyau corrigés.

— `» ./scripts/patch-realsense-ubuntu-lts.sh`

5.2.2 Compilation du SDK librealsense2

1. Se placer dans le répertoire racine de librealsense (si vous n'y êtes plus) et exécuter la commande suivante:

— `» mkdir build && cd build`

2. Utiliser CMake avec les options suivantes :

— `» cmake ../ -DBUILD_EXAMPLES=true -DBUILD_GRAPHICAL_EXAMPLES=false`

3. Recompiler et installer les binaires de librealsense.

— `» sudo make uninstall && sudo make clean && sudo make && sudo make install`

5.3 Programmation et Compilation

Maintenant que l'environnement logiciel est configuré nous allons pouvoir commencer à programmer.

Afin d'illustrer la démarche nous allons nous servir d'un "example" fournis avec le SDK. Executons la commande suivante :

— `» rs-distance`

Comme on peut le voir en Figure 5, nous arrivons à extraire la distance entre la caméra et l'objet au centre de l'image qu'elle capture.

```
terarys@marius-PC:~/Projet_S8/CAMERA/librealsense$ rs-distance
There are 1 connected RealSense devices.

Using device 0, an Intel RealSense D435
    Serial number: 913522071718
    Firmware version: 05.12.10.00

The camera is facing an object 0.000 meters away.
The camera is facing an object 0.000 meters away.
The camera is facing an object 0.000 meters away.
The camera is facing an object 0.000 meters away.
The camera is facing an object 0.000 meters away.
The camera is facing an object 0.000 meters away.
The camera is facing an object 0.000 meters away.
The camera is facing an object 0.000 meters away.
The camera is facing an object 0.000 meters away.
The camera is facing an object 0.000 meters away.
The camera is facing an object 2.034 meters away.
The camera is facing an object 2.041 meters away.
The camera is facing an object 2.041 meters away.
The camera is facing an object 1.981 meters away.
The camera is facing an object 1.975 meters away.
The camera is facing an object 1.975 meters away.
The camera is facing an object 1.975 meters away.
The camera is facing an object 1.968 meters away.
The camera is facing an object 1.981 meters away.
```

FIGURE 5 – Test de l'exemple **rs-distance** fournis avec le SDK

6 Prise en main des deux SDK

6.1 Premier programme pour manipuler le robot

La prise en main du SDK du bras a été assez intuitive. Afin d'illustrer quelques uns des outils mis à disposition dans le SDK du bras robot, nous n'allons pas décortiquer un des programmes exemples fournis avec le SDK mais expliquer en détail comment fonctionne notre programme `init-bras.cc` qui met le bras en position pour les étapes suivantes qui sont : détection d'objet, saisie de l'objet et dépose de l'objet à un autre endroit.

Le code donné en Listing 2 (voir Annexe A.1 en page 18) permet d'illustrer les mécaniques de connexion et d'envoi d'ordres au bras robotique.

Nous connectons au bras en instanciant la classe XArmAPI définie dans dans la librairie `xarm\wrapper\xarm_api.h` du SDK et en lui donnant comme paramètre l'adresse IP du bras robot. Ensuite on efface les codes d'erreurs et d'avertissements qui n'auraient pas été nettoyés et nous paramétrons le robot pour pouvoir interagir avec le gripper et positionner le bras. Enfin le bras reçoit 4 commandes successives :

1. Ordre de retour à la position d'origine avec en paramètre l'attente avant de pouvoir exécuter un autre ordre
2. Ordre de fermeture du gripper (Ouvert complètement à 850 et fermé complètement à 0), sans attente
3. Ordre déplacement du bras suivant une trajectoire linéaire en lui fournissant X, Y, Z, Pitch, Yaw, Roll (X, Y et Z en millimètres et Pitch, Yaw, Roll en degrés)
4. Ordre de rotation du bras en lui fournissant les valeurs des angles des 7 articulations (6 existantes sur le Xarm6 d'où le 0 en 7^{eme} position)

Le déroulement de cet algorithme est présenté en sous forme de diagramme en Annexe B.1 (page 28).

Dans les documents `xarm_cplus_api.md` et `xarm_api_code.md` fournis avec le SDK, nous avons la description de toutes les fonctions mises à notre disposition pour interagir avec le bras robot. La plupart d'entre elles sont présentes dans ce codes. bigskip

Nous savons désormais comment nous connecter au robot, lui demander de se déplacer de plusieurs manières et d'interagir avec le gripper. Il ne nous manque plus que d'exploiter la caméra afin de repérer un objet, extraire sa position et la distance caméra-objet et déplacer le bras en conséquence.

6.2 Programme d'exemple de la caméra

Lors de la prise en main du SDK de la caméra, qui a été un peu plus longue, nous avons essayé plusieurs codes d'exemples fournis par Intel. Afin d'illustrer une des principales fonctionnalités exploitée lors des travaux réalisés, nous allons parcourir et décrire le programme simple : `rs-hello-realsense.cpp`.

Le code donné en Listing 3 (voir Annexe A.2 en page 19) permet d'illustrer les mécaniques de création de pipeline, de synchronisation avec le stream (le flux d'images venant de la caméra) et enfin celui du traitement de l'image reçue (ici nous allons mesurer une distance).

Nous instancions un pipeline définie dans `librealsense2\rs.hpp` et nous le démarrons (par analogie, cette étape est équivalente à l'instanciation de la classe d'API du bras robot en lui passant l'adresse IP en paramètre).

Nous somme donc "connectés" à la caméra. Nous allons pouvoir commencer le traitement d'image :

1. Nous nous synchronisons avec elle en forçant le programme à attendre l'arrivée de la prochaine image dans le pipeline
2. Nous récupérons une image depuis un des capteurs de la caméra : ici nous nous intéressons au capteur de profondeur car nous souhaitons mesurer une distance caméra-objet.
3. Nous récupérons les dimensions de l'image afin de pouvoir calculer les coordonnées du point en son centre.
4. Nous appelons la méthode *get_distance(float x, float y)* en lui donnant en paramètres les coordonnées du centre de l'image. Cette méthode nous retourne l'estimation de la distance entre le capteur de la caméra de profondeur et l'objet qui est en coordonnées (x, y) sur l'image que l'on traite.
5. Nous affichons finalement dans la console la distance estimée que *get_distance(float x, float y)* nous a retourné.

Le déroulement de cet algorithme est présenté en sous forme de diagramme en Annexe B.2 (page 29).

Nous savons désormais comment effectuer un traitement d'image basique sur les images que nous stream les capteurs de la caméra D435 d'Intel. Maintenant nous allons chercher à faire fonctionner ensemble le bras et la caméra.

7 Faire fonctionner un code avec les deux SDKs

Maintenant que nous savons utiliser le bras d'une part et la caméra de l'autre, il nous faut pouvoir utiliser les deux sur un même programme pour pouvoir gérer la position du bras par rapport à ce que voit la caméra.

Si tout les étapes d'installation des SDKs ont été suivies, la librairie RealSense2 a été installée de façon système. Tandis que la librairie xArm6 est une librairie locale. Cela implique que l'on peut utiliser le Makefile de la SDK xArm pour pouvoir compiler des codes faisant appel à la librairie RealSense2.

Pour cela, il faut rajouter un *-lrealsense2* à la règle `test-` (ligne *57*), dans le fichier *Makefile* présent dans le dossier de la SDK du bras xArm6 :

```
1 test-%:
2         mkdir -p $(BUILD_EXAMPLE_DIR)
3         $(CXX) $(addprefix ./$(EXAMPLE_DIR)/, $(subst test-, , $@)).cc $(C_FLAGS)
4         -L$(BUILD_LIB_DIR) -lrealsense2 -lxarm
5         -o $(addprefix $(BUILD_EXAMPLE_DIR), $(subst test-, , $@))
```

Listing 1 – Aperçu après modification du Makefile

Cette action va nous permettre d'avoir une commande simple pour compiler nos programmes. Il ne reste pour qu'à coder en C++ dans le dossier *example* présent dans le dossier de la SDK et de ensuite lancer la commande :

— » `sudo make test-`*`nom_du_code`*

Par exemple, si l'on veut compiler un code se nommant *test_code.cpp* présent dans le dossier *example*, on tapera le commande :

— » `sudo make test-test_code`

Il ne reste ensuite plus qu'à lancer l'exécutable qui sera créé lors de la compilation dans le dossier *build/example*.

8 Exploiter les données de la caméra pour piloter le bras

Cette est consacrée au programme qui utilise les deux SDK à la fois. Ce programme est lancé une fois le programme `init-bras.cc` exécuté et terminé. Nous allons explorer et décrire ce que fait ce programme.

8.1 La récupération des informations

Nos deux programmes écrits en langage `C++` sont appelé par un programme en Python. Le programme principal en Python sera décrit dans la partie 9.

Notre premier programme `init-bras.cc` n'a besoin d'aucune information venant de l'extérieur pour fonctionner. Son rôle est d'amener le bras dans une position déterminée pour pouvoir lancer la phase de recherche d'objet.

Le programme que nous allons maintenant parcourir permet d'aller saisir l'objet détecté par la caméra et d'aller le déposer à un autre endroit. Ce programme est appelé plusieurs fois par le code principal en Python avec des paramètres qui changent au cours du temps. C'est pour cela que nous allons devoir récupérer et stocker ces informations.

Comme on peut le voir en Listing 4 en Annexe A.3 (page 20), notre programme va faire appelle aux fonctionnalités offertes par les deux SDK ce qui explique que les deux bibliothèques mentionnées dans les parties 6.1 et 6.2 soient importées.

Deux coefficients à virgules flottantes sont déclarés : `RATIO_x` et `RATIO_y`. Ces coefficients estimés par nos soins, nous permettent de convertir une distance en pixels à une distance en millimètres à une hauteur fixe de 60 cm. En effet le repère 3D du robot est en millimètres et le programme Python qui va venir exécuter ce second code va lui fournir des coordonnées en pixels.
Ces coefficients sont donc entièrement corrélés à la position dans laquelle se trouve le bras robot après l'exécutions du code `init-bras.cc`. Si jamais vous souhaitez modifier cette position et notamment la hauteur du gripper (et donc de la caméra), les coefficients mentionnés précédemment seraient à déterminer de nouveau.

Nous enregistrons donc les 6 paramètres suivants, renseignés par le code principal en Python, utiles pour la suite de notre second programme :

- PosX : Correspond à la position de l'objet détecté par la caméra sur son axe des X.
- PosY : Correspond à la position de l'objet détecté par la caméra sur son axe des Y.
- calibration : Si calibration est à une autre valeur que 0, alors nous allons aligner la pince du bras robot avec l'objet.
- descente : Si descente est à une autre valeur que 0, alors nous allons saisir l'objet qui est en dessous de la pince.
- depot : Si depot est à une autre valeur que 0, alors nous allons déposer l'objet précédemment saisi à un autre endroit.
- offsetZ : Cette valeur est la distance en millimètre que le bras va devoir parcourir afin de saisir l'objet lors de la phase de dépôt (récupérée lors de la phase de descente).

Selon ce qu'on passe en argument lorsque l'on lance ce code, nous allons réaliser soit la phase de d'alignement, soit la phase de saisie, soit celle de dépôt.

8.2 Récupération de la distance à l'objet

Dans cette partie nous allons nous servir de la caméra de profondeur pour mesurer la distance par rapport à un objet aux coordonnées (posX, posY) passées en paramètres. Le code est donné en Listing 5 en Annexe A.3 (page 21).

Tout d'abord nous créons le pipeline mais nous allons le configurer avant de démarrer le streaming des images. Ici nous précisons que nous nous servirons du stream venant du capteur de profondeur, avec un format d'image 720p (1280x720), 16 bits pour définir la profondeur et en 30 images par secondes. Nous démarrons ensuite le streaming.
Afin de laisser le temps au capteur de faire son focus et de se stabiliser, on réalise l'acquisition de 30 images (soit une temporisation d'une seconde) que nous n'utiliserons pas.

Nous allons ensuite nous servir de la méthode `get_distance(float X, float Y)` que nous avions découvert grâce au code d'exemple (voir la partie 6.2). Cependant cette fois-ci nous allons effectuer 5 mesures d'affilée et en faire la moyenne, pour augmenter la précision.

Nous pouvons maintenant arrêter le streaming du pipeline. Nous avons mesuré la distance à l'objet grâce au capteur de profondeur qui nous produisait des images dont la résolution était de 1280x720. Maintenant que cette étape est fini, nous appliquons un offset à `PosX` et à `PosY` afin de transposer ces coordonnées à celles de l'image RGB qui elle est en 640x480 (caméra qui nous sert à la détection d'objet que nous verrons dans la partie 9).

8.3 Connexion au bras

Dans cette partie nous allons nous connecter au bras. Le code est donné Listing 6 en Annexe A.3 (page 22). Lors de chacune des phases nous devons nous connecter au bras et enregistrer la position actuelle du bras. Le code est très similaire à `init-bras.cc` décrit précédemment.

8.4 Alignement de la pince et de l'objet

Si le code Python appelle notre programme `calibration-bras.cc` avec comme paramètres :

- PosX : Une coordonnée en X
- PosY : Une coordonnée en Y
- calibration : 1
- descente : 0
- depot : 0
- offsetZ : Une valeur non utilisé pour cette étape

alors après nous être connecté au bras, nous allons exécuter le morceau de code donné en Listing 7 en Annexe A.3 (page 23).

Tout d'abord nous faisons correspondre aux coordonnées `posX` et `posY` en pixels des valeurs en millimètres que nous enregistrons dans les variables `offsetX` et `offsetY`.

Nous avions plus tôt enregistré la position du bras dans un vecteur nommé `position`. La position de recherche de l'objet est spécifique car nous demande peu de calcul afin de faire correspondre le repère 3D du robot et celui de la caméra. Comme nous avons demandé au bras d'effectuer une rotation de 180°, l'axe des X de la base du robot correspond à l'axe des -Y de la caméra et l'axe des Y de la base du robot correspond à l'axe des X de la caméra. Nous allons donc soustraire à `posX` et `posY` respectivement 320 et 240 pixels afin d'avoir les coordonnées de l'objet par rapport au centre de l'image et non plus par rapport au coin en haut à gauche et appliquer le coefficient déterminé précédemment.

Nous appliquons à `position[0]` et à `position[1]`, qui correspondent aux positions X et Y dans le vecteur, l'offset que nous venons de calculer ainsi qu'un deuxième résultant de l'écart entre l'objectif de la caméra RGB et le centre de la pince (écart en X et en Y).

Enfin nous demandons au bras de se déplacer aux coordonnées que nous venons de calculer en passant par la méthode `set_position(fp32 vecteur, bool wait)` avec comme paramètre notre vecteur et l'ordre d'attendre la fin du mouvement avant de pouvoir passer à la suite.

8.5 Descente et saisie de l'objet

Si le code Python appelle notre programme `calibration-bras.cc` avec comme paramètres :

- PosX : Non utilisé pour cette étape (car le bras est déjà au dessus de l'objet)
- PosY : Non utilisé pour cette étape (car le bras est déjà au dessus de l'objet)
- calibration : 0
- descente : 1
- depot : 0
- offsetZ : L'offset calculé à lors de l'étape "calibration"

alors après nous être connecté au bras, nous allons exécuter le morceau de code donné en Listing 8 en Annexe A.3 (page 23).

Ce bout de code est assez simple et reprend beaucoup des mécaniques des codes précédents. Nous venons tout d'abord convertir en millimètres la distance mesurée en mètres par la caméra à laquelle nous retirons 11,5cm correspondant à la distance entre l'objectif de la caméra RGB et le bas de la pince du bras. Nous ajoutons ensuite 1,5cm afin que l'objet soit bien saisi par le gripper.

Nous venons ensuite ouvrir à fond la pince du robot et sans attendre entamer la descente. Une fois le mouvement de descente terminé nous refermons la pince (nous n'avons pas de retour de capteur du préhenseur mais la pince saisi l'objet et gère de manière autonome la pression qu'elle applique sans détériorer ni l'objet ni la pince elle-même). Enfin nous soulevons l'objet et nous plaçons en position pour le dépôt.

8.6 Dépôt de l'objet

Si le code Python appelle notre programme `calibration-bras.cc` avec comme paramètres :

— PosX : Correspond à la position de la cible détectée en X.
— PosY : Correspond à la position de la cible détectée en Y.
— calibration : 0
— descente : 0
— depot : 1
— offsetZ : L'offset calculé à lors de l'étape "descente"

alors après nous être connecté au bras, nous allons exécuter le morceau de code donné en Listing 9 en Annexe A.3 (page 24).

L'étape finale de dépôt de l'objet est elle aussi assez simple et utilise les mécaniques déjà décrites précédemment. Nous allons descendre en Z de notre position moins la valeur calculée précédemment lors de la saisie (`offsetZ`) à laquelle nous retirons 10 millimètres par sécurité (nous allons lâcher l'objet à 1 cm de la table).

9 OpenCV, détection d'objet et Démontration

Le choix du python comme langage "Maître" s'est fait naturellement lorsque nous avons tenté de réaliser de la détection de formes. Nous avons ainsi opté pour la librairie OpenCV, qui nous permet, sous Python3, de disposer d'un très grand nombre de fonctions pour réaliser cette tâche.

Enfin, nous avons créé une fonction python nous permettant de détecter la forme par détection de seuil sur une image en nuances de gris (voir fonction *threshold()* dans la documentation OpenCV.

Le principe de cette fonction est le suivant : nous récupérons tous les contours détectés sur l'image, sélectionnons celui ayant la plus grande aire, puis nous calculons la position du centre de la forme sur l'image.

Pour obtenir une plus grande précision, nous réitérons cette opération sur 15 images et calculons la moyenne des positions X et Y obtenues. Nous renvoyons finalement ces deux entiers (voir Listing 11 en Annexe A.4 page 26).

Un offset est appliqué car les deux caméras utilisées lors la démonstration n'ont pas la même résolution (et donc le même nombre de pixels). La caméra RGB est en 640x480 alors que la caméra à profondeur de champ est en 1280x720. Cette différence est volontaire, elle annule le fait que les deux caméras n'ont pas le même angle de vue et permet de s'assurer qu'un pixel sur la caméra RGB représente la même surface qu'un pixel de la caméra à profondeur de champ.

Avant la détection d'objet, nous devons passer par une phase d'initialisation, afin importer les librairies nécessaires et exécuter `init-bras` (voir Listing 10 en Annexe A.4 page 25).

Nous exécutons enfin la boucle principale du code (voir Listing 12 en Annexe A.4 page 27). Cet algorithme permet de réaliser la séquence suivante :

— Étape 1 : Mise en position initiale du bras
— Étape 2 : Détection de l'objet
— Étape 3 : Mise en position du bras au-dessus de l'objet
— Étape 4 : Saisie de l'objet
— Étape 5 : Remise en position initiale du bras
— Étape 6 : Détection de la croix, cible pour déposer l'objet
— Étape 7 : Mise en position du bras au-dessus de la croix
— Étape 8 : Dépôt de l'objet
— Étape 9 : Mise en position initiale du bras

Algorithme disponible Annexe B.4 (page 31)

10 Conclusion

Cette démonstration que conclut notre avancée sur ce projet. Grâce à ce projet, nous espérons grandement faciliter l'appropriation du sujet par les futurs étudiants travaillant sur le bras robotisé accueilli par la base roulante développée dans le cadre ligue AtWork de la RobotCup.

Pour pouvoir participer à cette ligue, le bras doit être en mesure de reconnaître un objet plus précisément. Il reste donc à réaliser une reconnaissance d'objet plus poussée afin de correspondre totalement au cahier des charges de la coupe AtWork.

Nous déplorons bien évidemment les conditions sanitaires dans lesquelles nous avons du mener ce projet. Nous espérons que les prochains étudiants qui auront l'opportunité de continuer ce projet ambitieux aient tout le matériel à leur disposition et puisse s'approprier notre travaille afin de monter en puissance et exploiter toutes les possibilités offertes par ces deux systèmes.

A Programmes

A.1 Prise en main du bras

```cpp
#include "xarm/wrapper/xarm_api.h"

int main(int argc, char **argv) {
    // IP fixe du robot
    char IP_bras[] = "192.168.1.208";
    std::string port(IP_bras);

    // Connection au bras
    XArmAPI *arm = new XArmAPI(port);
    sleep_milliseconds(500);

    // Clear éventuelles erreurs
    if (arm->error_code != 0) arm->clean_error();
    // Clear éventuels avertissements
    if (arm->warn_code != 0) arm->clean_warn();

    // Permet le déplacement du bras
    arm->motion_enable(true);
    // Mode : Position control
    arm->set_mode(0);
    // Etat : Ready
    arm->set_state(0);
    sleep_milliseconds(500);
    // Permet d'utiliser le gripper
    arm->set_gripper_enable(true);
    // Réglage vitesse gripper
    arm->set_gripper_speed(5000);

    printf("=========================\n");

    int ret;
    // Vecteur de positions X, Y ,Z, Pitch, Yaw, Roll (montée de 600mm en z)
    fp32 poses[6] = {207, 0, 600, 180, 0, 0};
    // Vecteur des positions angulaires articulations 1, 2, ,... (rotation de 180°)
    fp32 angles[] = {180, -36.6, -69.6, 0, 106.1, 0, 0};

    // Déplacement position d'origine
    arm->reset(true);
    // Fermeture gripper sans attente
    arm->set_gripper_position(10, false);
    // Déplacement (linéaire)
    ret = arm->set_position(poses, true);
    printf("set_position, ret=%d\n", ret);
    // Déplacement (rotation)
    ret = arm->set_servo_angle(angles, true);
    printf("set_position, ret=%d\n", ret);
    return 0;
}
```

Listing 2 – Fonction de mise en position initiale du bras

A.2 Prise en main de la caméra

```cpp
// API cross-plateforme d'Intel RealSense
#include <librealsense2/rs.hpp>

// Création du pipeline (sert au streaming et au traitement des images)
rs2::pipeline p;

// Configure et démarre le pipeline
p.start();

// Attente pour garantir la synchronisation avec les images
rs2::frameset images = p.wait_for_frames();

// Récupère l'image depuis le capteur de profondeur
rs2::depth_frame profondeur = images.get_depth_frame();

// Récupère les dimensions de l'image reçue
float largeur = profondeur.get_width();
float hauteur = profondeur.get_height();

// Demande la distance entre la caméra et l'objet au centre de l'image
float distance_camera_objet = profondeur.get_distance(largeur / 2, hauteur / 2);

// Ecrit la distance à l'objet dans la console
std::cout << "Distance camera-objet : " << distance_camera_objet << "m \r";
```

Listing 3 – Code simple utilisant la fonctionnalité de mesure de distance

A.3 Saisie de l'objet et dépose

```
#include "xarm/wrapper/xarm_api.h"
#include <cstdlib>
#include <iostream>
#include <librealsense2/rs.hpp>

// Coefficients estimés pour le ratio mm/pixels
float RATIO_y = 485.0 / 480.0;  //48.5cm  = 480 pixs  @z = 60 cm
float RATIO_x = 587.5 / 640.0;  //58.75cm = 640 pixs  @z = 60 cm

//---------------------------- //
// Récupération des paramètres //
//----------------------------//
int main(int argc, char **argv) {

  if (argc < 9) {
      printf("No enought arguments\n");
      return 0;
  }

  int posX = atoi(argv[1]);
  int posY = atoi(argv[2]);
  int calibration = atoi(argv[3]);
  int descente = atoi(argv[4]);
  int depot = atoi(argv[5]);
  int offsetZ = atoi(argv[6]);

  ...

}
```

Listing 4 – Récupération des paramètres du code python

```
...

//----------------------------------------//
// Récupération de la distance de l'objet //
//----------------------------------------//
rs2::pipeline p;
rs2::config cfg;
cfg.enable_stream(RS2_STREAM_DEPTH, 1280, 720, RS2_FORMAT_Z16, 30);
p.start(cfg);

// Temporisation nécessaire à la stabilisation de la caméra
rs2::frameset frames;
for(int i = 0; i < 30; i++)
{
   // Attente de l'arrivée des images dans le pipeline
   frames = p.wait_for_frames();
}

// Récupération et estimation de la distance à l'objet
// à d'une moyenne sur 5 images
float dist_sum = 0;
int n_frames = 0;
int n_frames_tot = 5;
while(n_frames < n_frames_tot){
  frames = p.wait_for_frames();
  rs2::depth_frame depth = frames.get_depth_frame();
  float tmp = depth.get_distance(posX, posY);
  if(tmp != 0){
    dist_sum += tmp;
    n_frames++;
  }
}

float dist_tmp = dist_sum/n_frames_tot;

p.stop();

// L'image étant en 640*480, nous souhaitons
// exprimer les coordonnées par rapport au centre de l'image
posX -= 320;
posY -= 120;

...
```

Listing 5 – Récupération de la distance à l'objet

```
1
2   ...
3
4
5   //-------------------------------------------------//
6   // Déplacement du bras pour s'approcher de l'objet //
7   //-------------------------------------------------//
8
9   // Connexion au bras et sauvegarde de sa position
10  if(calibration || descente || depot)
11  {
12
13     // Connection au bras et paramétrage par défaut
14     char IP_bras[] = "192.168.1.208";
15     std::string port(IP_bras);
16     XArmAPI *arm = new XArmAPI(IP_bras);
17     sleep_milliseconds(500);
18     arm->motion_enable(true);
19     arm->set_mode(0);
20     arm->set_state(0);
21     if (arm->error_code != 0) arm->clean_error();
22     if (arm->warn_code != 0) arm->clean_warn();
23     sleep_milliseconds(500);
24     arm->set_gripper_enable(true);
25     arm->set_gripper_speed(5000);
26
27     // On enregistre sa position actuelle
28     fp32 position[6];
29     arm->get_position(position);
30
31     ...
32
33  }
34  std::cerr << offsetZ;
35  return offsetZ;
36
37  ...
```

Listing 6 – Connexion au bras

```
1
2   ...
3
4   // Déplacement en X et Y du bras pour se positionner au dessus de l'objet
5   if(calibration && !descente && !depot)
6   {
7     fp32 offsetX = (posX-320)*RATIO_y;
8     fp32 offsetY = (posY-240)*RATIO_x;
9     position[0] += offsetY;
10    position[1] += offsetX;
11
12    // Distance (en x et en y) entre le capteur de la caméra
13    // et le centre de la pince
14    position[0] -= 140;
15    position[1] -= 40;
16
17    arm->set_position(position,true);
18  }
19
20  ...
```

Listing 7 – Alignement de la pince et de l'objet

```
1
2   ...
3
4   //déplacement sur l'axe Z pour attrapper l'objet
5   else if(!calibration && descente && !depot)
6     {
7        offsetZ = dist_tmp*1000 - 115 + 15;
8        fp32 angles[] = {180, -36.6, -69.6, 0, 106.1, 0, 0};
9        arm->set_gripper_position(850, false);
10       position[2] -= offsetZ;
11       arm->set_position(position, true);
12       arm->set_gripper_position(10, false);
13       sleep_milliseconds(1000);
14       arm->set_servo_angle(angles, true);
15    }
16
17  ...
```

Listing 8 – Déscente et saisie de l'objet

```
...

//déplacement sur l'axe Z pour déposer l'objet
else if(!calibration && !descente && depot)
  {
     fp32 angles[]  = {180, -36.6, -69.6, 0, 106.1, 0, 0};
     position[2]   -= offsetZ;
     position[2]   += 10;
     arm->set_position(position, true);
     arm->set_gripper_position(850, false);
     sleep_milliseconds(1000);
     arm->set_servo_angle(angles, true);
     offsetZ = 0;
  }

  ...
```

Listing 9 – Dépot de l'objet

A.4 Programme principal en Python : détection de l'objet

```python
# import the necessary packages
import numpy as np
import cv2
import subprocess
from subprocess import PIPE

def change(x):
    # print(x)
    pass

# load the image and resize it to a smaller factor so that
# the shapes can be approximated better
cap = cv2.VideoCapture(6)

cv2.namedWindow("params")

cv2.createTrackbar('max_val', 'params', 255, 255, change)
cv2.createTrackbar('th', 'params', 60, 255, change) #réglage du threshold
cv2.createTrackbar('start', 'params', 1, 1, change) #Curseur pour démarrer la démo

#on place le bras dans sa position de repérage
subprocess.run(["./init-bras"])
```

Listing 10 – Phase d'initialisation du code Python

```python
def reconnaissance(): #retourne la position de l'objet
    x = 0
    y = 0
    nbrMesureMax = 15
    nbrMesure = 0
    while nbrMesure < nbrMesureMax:
        # image 480*640*3
        ret1, img = cap.read()
        imgGry = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
        #imgGry = cv2.bitwise_not(imgGry)

        max_val = cv2.getTrackbarPos('max_val', 'params')
        th = cv2.getTrackbarPos('th', 'params')
        start = cv2.getTrackbarPos('start', 'params')
        ret, thrash = cv2.threshold(imgGry, th , max_val, cv2.THRESH_BINARY_INV)
        contours , hierarchy = cv2.findContours(thrash, cv2.RETR_TREE,
                               cv2.CHAIN_APPROX_SIMPLE)

        i = 0
        indexMaxArea = 0
        maxArea = 0
        for c in contours:
            area = cv2.contourArea(c)
            if(area > maxArea):
                indexMaxArea = i
                maxArea = area
            i = i+1

        if(len(contours)):
            contour = contours[indexMaxArea]
            rect = cv2.minAreaRect(contour)
            ((cX, cY), (w, h), angle) = rect


            box = cv2.boxPoints(rect)
            box = np.int0(box)
            cv2.drawContours(img, [box], 0, (0, 0, 0), 5)

            cv2.circle(img, (int(cX),int(cY)), 7, (0, 0, 255), -1)
            if(start):
                x = x + cX
                y = y + cY
                nbrMesure = nbrMesure + 1

        k = cv2.waitKey(1) & 0xFF
        if k == 27:
            break
        cv2.imshow("Shape", img)
        cv2.imshow("thrash", thrash)

    y_offset = 120
    x_offset = 320
    x = int(x/nbrMesureMax) + x_offset
    y = int(y/nbrMesureMax) + y_offset
    return (x, y, k)
```

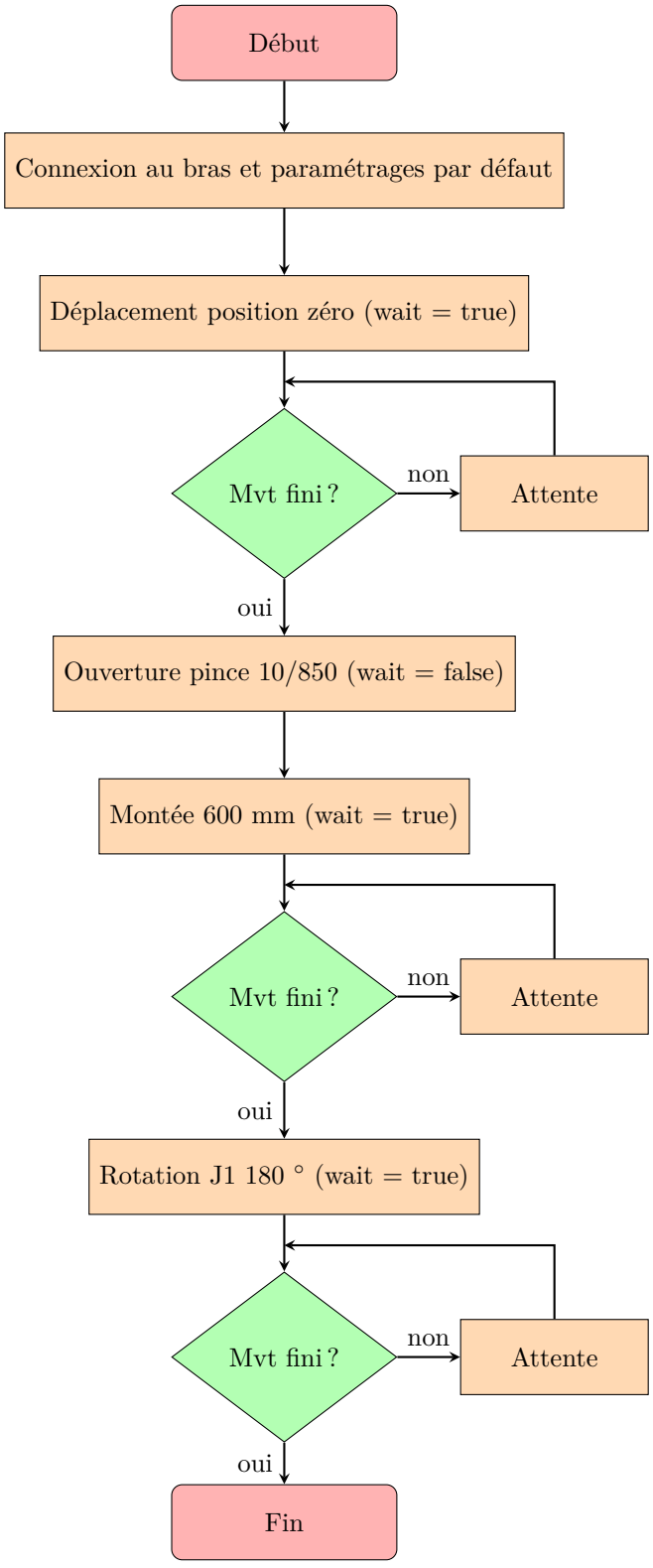
Listing 11 – Fonction python pour détecter une forme

```python
k = 0
iteration = 0
iterationMax = 3
offsetZ = 0

#La première itération pour ajuster la position du robot pour prendre l'objet
#(le bras se positionne au dessus de l'objet)
#
#La deuxième itération pour ajuster la position du robot pour déposer l'objet
#(le bras se positionne au dessus de la croix)
for j in range(2):
  #On répète 3 ajustement de la position pour contrer l'effet de distortion de
  #la lentille du capteur
  while iteration < iterationMax and k != 27:
    move = 1
    descente = 0
    depot = 0

    #On repère où est l'objet
    (posX, posY, k) = reconnaissance()

    #On lance le programme pour se calibrer (sauf si l'on a appuyer sur la touche
    #"echap"
    if k!=27:
      ret = subprocess.run(["./calibration-bras", str(posX), str(posY), str(move),
            str(descente), str(depot), str(0)])

    iteration = iteration + 1

  iteration = 2
  move = 0
  descente = 1-j
  depot = j

  #On repère finalement l'objet pour venir le saisir dans la première itération
  #Pour la deuxième itération, on repère la croix pour poser l'objet dessus
  (posX, posY, k) = reconnaissance()
  if k!= 27:
    ret = subprocess.run(["./calibration-bras", str(posX - 50), str(posY + 10),
          str(move), str(descente), str(depot), str(offsetZ)], stderr=PIPE)
    offsetZ = int(ret.stderr)
```

Listing 12 – Boucle principale du code python

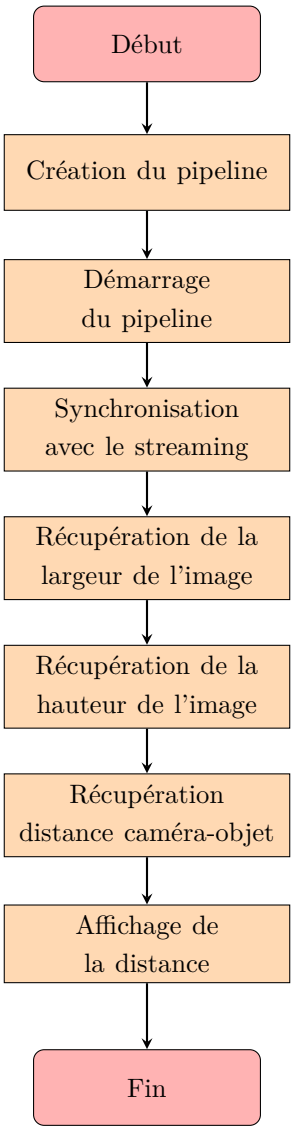
B Algorithmes

B.1 `init-bras.cc`



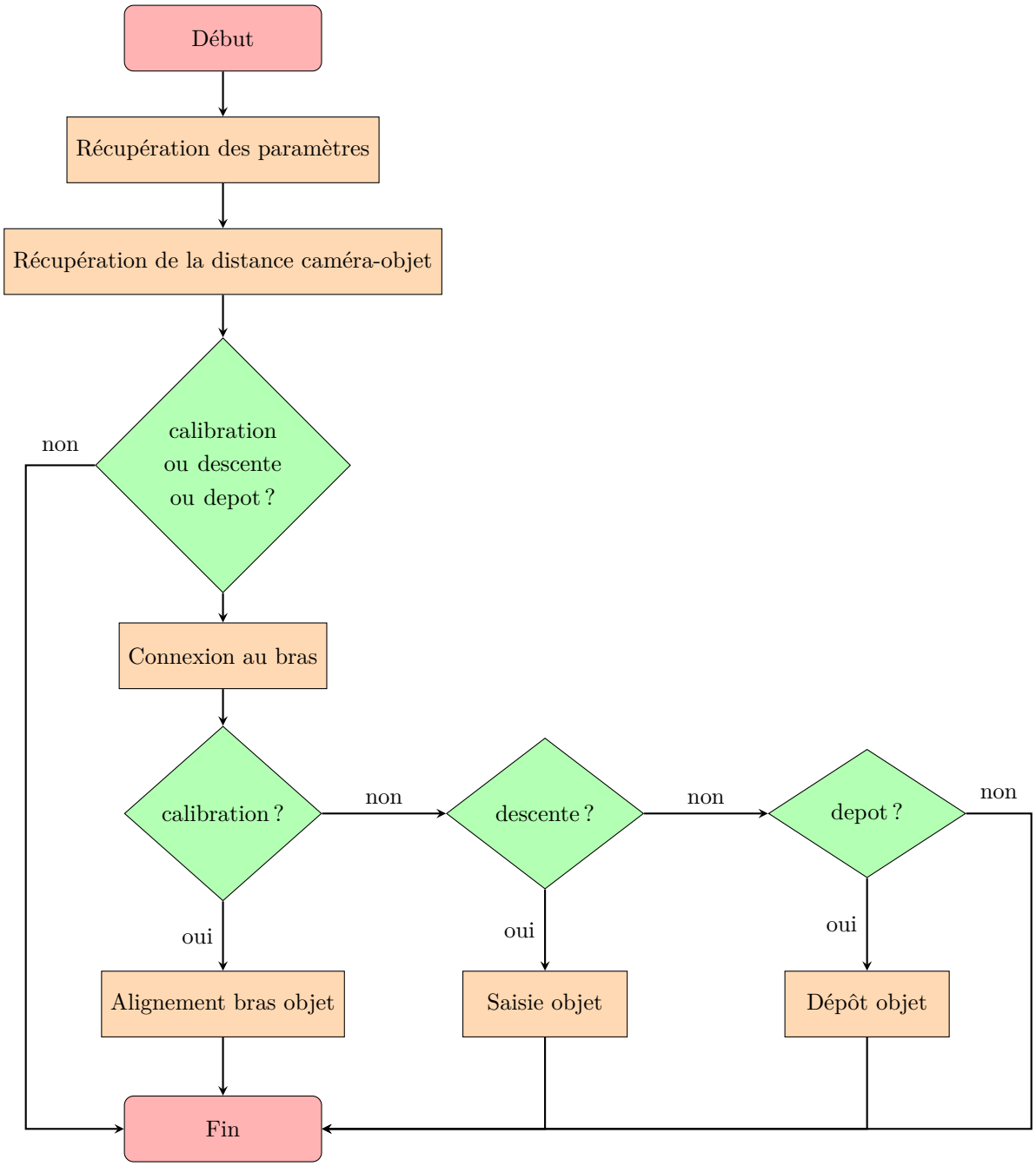
Annexe B.1.1 – Diagramme de l'algorithme de `init-bras.cc`

B.2 `rs-hello-realsense.cpp`



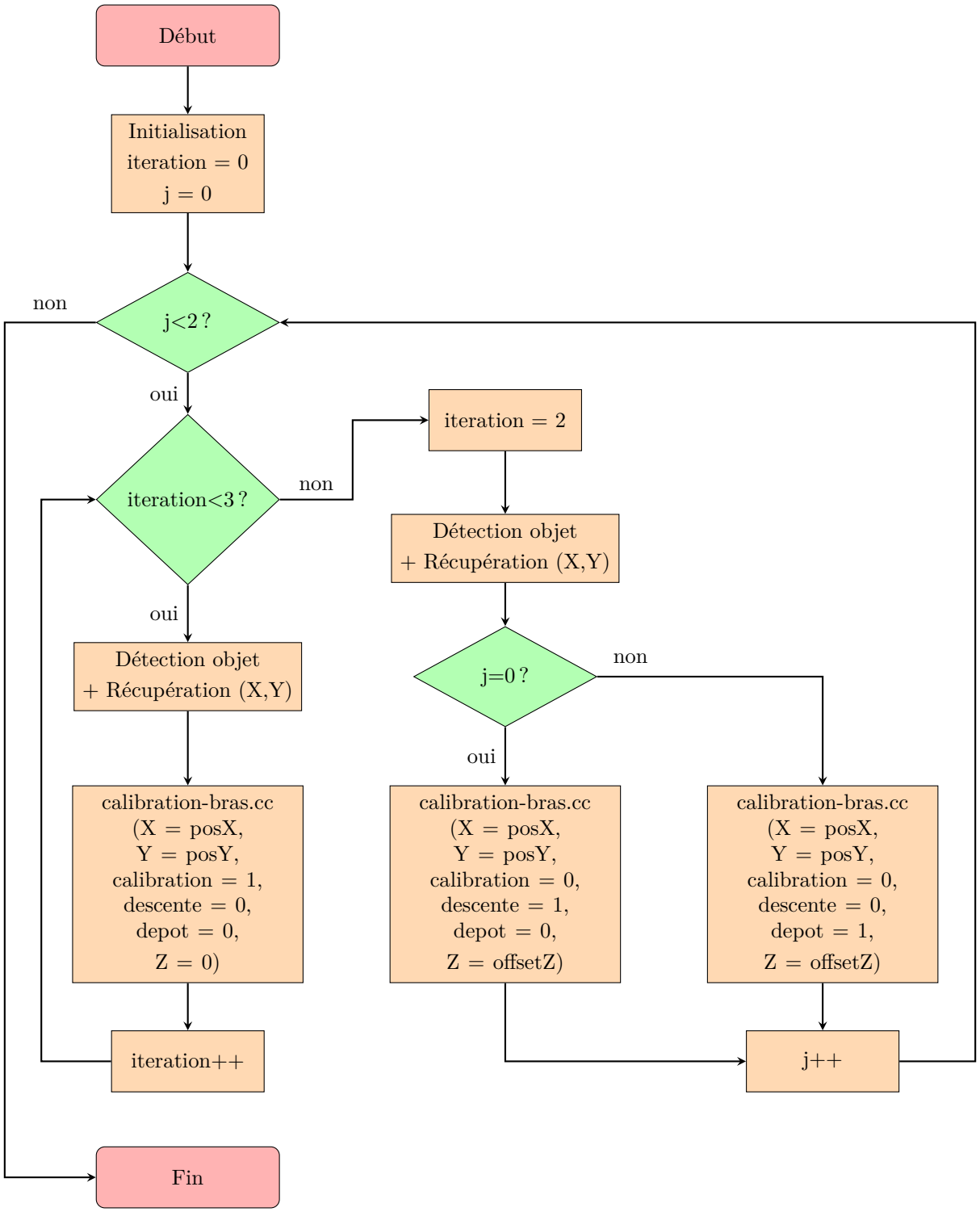
Annexe B.2.1 – Diagramme de l'algorithme de `rs-hello-realsense.cpp`

B.3 `calibration-bras.cpp`



Annexe B.3.1 – Diagramme de l'algorithme de `calibration-bras.cc`

B.4 `test_shape_detection.py`



Annexe B.4.1 – Diagramme de l'algorithme de `test_shape_detection.py`