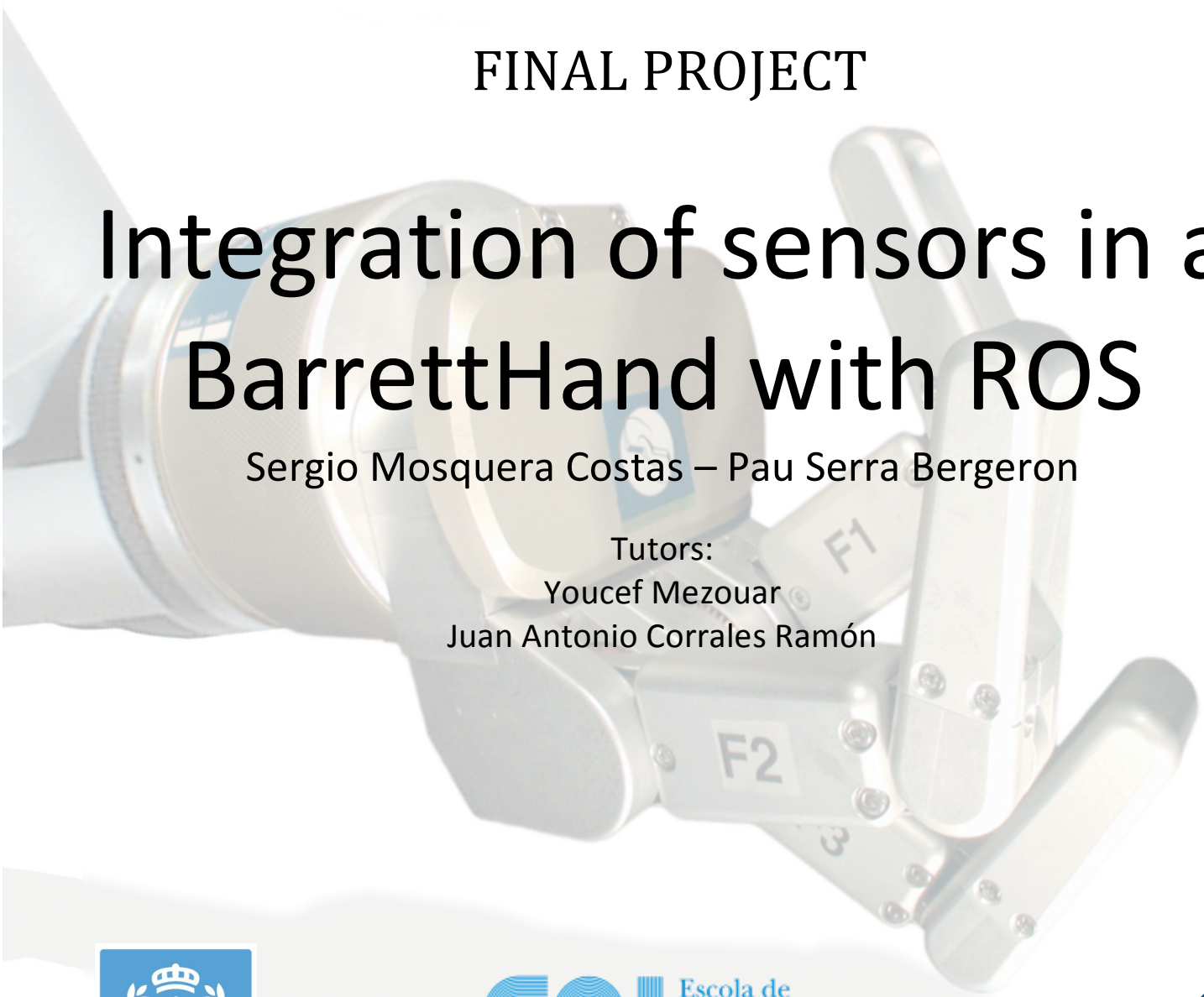
July | '15

FINAL PROJECT

Integration of sensors in a BarrettHand with ROS

Sergio Mosquera Costas – Pau Serra Bergeron

Tutors:
Youcef Mezouar
Juan Antonio Corrales Ramón



ETSEIB

Escola de Enxeñería Industrial
Universidade Vigo

IFMA

IFMA – Institut Français de Mécanique Avancée
ETSEIB – Escola Técnica Superior de Enginyeria Industrial de Barcelona
EEI – Escola de Enxeñería Industrial – Universidad de Vigo

INDEX

Index of Figures

Index of Tables

Acknowledgements

First of all, we would like to express our sincere gratitude to the Institut Français de Mécanique Avancée(IFMA), Escola Técnica Superior de Enginyeria Industrial de Barcelona(ETSEIB) and Escola de Enxeñería Industrial de Vigo(EEI) for making possible the realization of our final project and giving us the great opportunity of living the Erasmus experience in France.

Special thanks to our project tutors Youcef Mezouar and Juan Antonio Corrales Ramón for helping us and giving us advice, their availability and attention during the progress of our work.

We would like to thank the people at the Institut Blaise Pascal, for helping in the evolution of our project, especially to Laurent Lequievre and François Berry.

We are also very thankful to the people from the CTT at the IFMA, for their collaboration in the fabrication of the pieces, especially to Marta Prat Lleixá and Clement Weigel.

Thanks also to all our fellow students at the IFMA for accepting us, and especially to our project colleagues and overall, friends, for supporting us and helping us in whatever they could.

And finally, a great mention to our respective families for their gratitude, patience and help during this last six months of work under any circumstances.

1 Introduction

Since the beginning, handling tasks were performed by workers. It has always sought to improve the speed of these tasks and has sought to improve them. One of the first innovations was the arrival of mass production. The theoretical idea born with Taylorism and who had the idea of putting it into practice was Olds, who opened its assembly line in 1901. However, the assembly line took popularity a few years later, thanks to Henry Ford, who taking the idea of Ransom Olds, developed an assembly line with a higher production capacity. His idea was a way of organization of the production that delegates to each worker a specific and specialized function. This method considerably improved production rate. But the industry in its constant search for improvement has begun to include robot manipulators.

Robots manipulators are more and more widespread in current industries due to their positional precision, repeatability and durability. They are usually applied in repetitive tasks where all the components are always in the same position and only a position control of the robot is needed. For example robot arms in automotive assembly lines perform a variety of tasks such as welding and parts rotation and placement during assembly. However, this is a sector where research and innovation are paramount. The manipulation by one or more robotic arms of complex objects is certainly one of the most important challenges in the context of robotic production. This innovation will reduce the difficulty of certain tasks performed today by the operators, but also to deal with a shortage labor and competitiveness from countries where labor costs are lower.

These robotic manipulators are normally formed by a robotic arm and the end effector, which is normally called robotic hand. A robotic arm is a type of mechanical arm, usually programmable, with similar functions to a human arm; the arm may be the sum of the mechanism or may be part of a more complex robot. The end effector, or robotic hand, can be designed to perform any desired task. In some circumstances, close emulation of the human hand is desired, as in robots designed to perform surgery or bomb disarmament.

As part of the national project ROBOTEX, the Pascal Institute (CNRS/UBP/IFMA) has acquired two manipulator arms KUKA LWR4+ (arms from a technology transfer between DLR and KUKA) and dexterous hands of the company Shadow Robotics and society Barrett Technology. This project was born of the necessity of the Pascal Institute to incorporate the BarrettHand in one of the two manipulators.

So the first thing that we have to do is integrating the BarrettHand in the framework ROS, to make the control and to simplify the interaction of the BarrettHand with his environment. We will create a graphic interface that we will integrate in ROS. This GUI will be intuitive so we will make a simple design of the widget. It has to be easy to use by the future users.

We will also integrate tactile sensors in the fingertips of the hand to improve the control of the grasping of objects. And we will integrate a force/torque sensor in the wrist. This sensor will permit the user detecting the characteristics of the object that we have grasp, for example, when we take a bottle and we want to pour the content of the bottle, we will be able to control the amount being poured.

2 Description of the BarrettHand

2.1 BarrettHand™

The BarrettHand is a multi-fingered grasper with the dexterity to secure target objects of different sizes, shapes, and orientations. It uses industry-standard serial communications. The compactness and low weight of the BarrettHand assures that the enhanced dexterity does not compromise arm payload. The control-electronics package is contained entirely within the palm shell, reducing electrical wiring to a simple cable carrying all communications and motor power.

2.1.1 Flexibility and durability in a compact package

The flexibility of the BarrettHand is based on the articulation of eight joint axes. Only four brushless DC servomotors are needed to control all eight joints. The resulting 1.18 kg grasper is completely self-contained with only an 8 mm diameter cable supplying DC power and establishing a two-way serial communication link to the main robot controller of the workcell. The grasper's communications electronics, brushless servomotors, sensors, current amplifiers, signal processing electronics and electronic commutation are all packed neatly inside the palm body of the grasper.

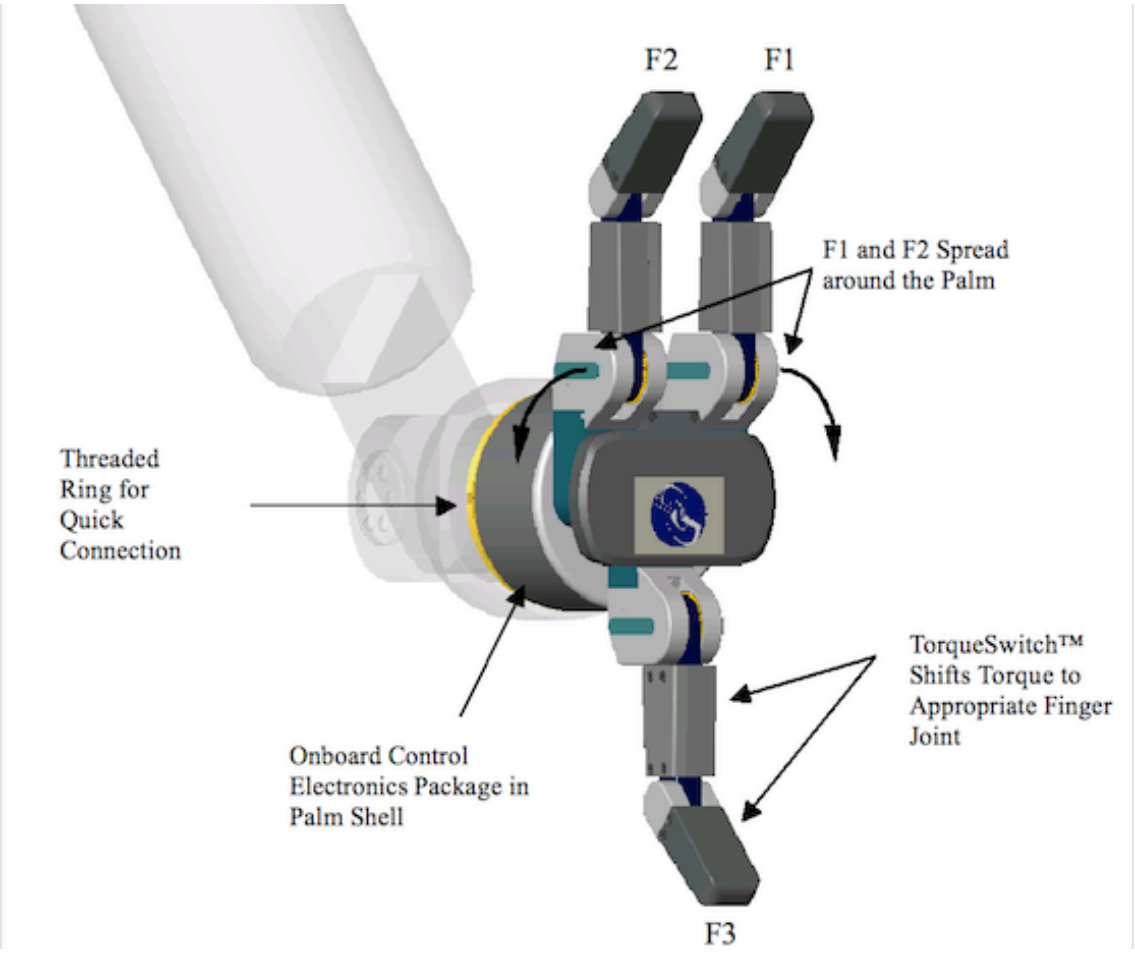


Figure 1: BarrettHand

The BarrettHand has three articulated fingers and a palm which act in concert to trap the target object firmly and securely within a grasp consisting of

seven coordinated contact vectors, one from the palm plate and one from each link of each finger. Each of the BarrettHand's three fingers is independently controlled by one of three servomotors. Except for the spread action of fingers F1 and F2, which is driven by the fourth and last servomotor, the three fingers, F1, F2 and F3, have inner and outer articulated links with identical mechanical structure. The spreading action of fingers F1 and F2 on the BarrettHand increases the dexterity of the entire unit with only one additional actuator.

The position, velocity and acceleration can all be processor controlled over the full range of 17,500 encoder positions. The maximum force that can be actively produced is 2kg, measured at the tip of each finger. Once the grasp is secure, the link automatically lock in place allowing the motor currents to be switched off to conserve power until commanded to readjust or release their grasp. The spread can be controlled to any of 3000 positions over its full range in either direction within 0.5 seconds. Unlike the mechanically lockable finger-curl motions, the spread motion is fully backdrivable, allowing its servos to provide active stiffness control in addition to control over position, velocity, acceleration and torque.

2.1.2 Arm adapter

The BarrettHand is equipped with a threaded base for compact and secure mounting. Threaded base is fully compatible with the BarrettArm. With the arm adapter, it can be mounted on virtually any robot with an arm adapter provided by Barrett Technology. This lightweight arm adapter is made to work with the end-effector bolt pattern on the robot, allowing quick, easy mounting and wire management for a BH8-SERIES System. The arm adapter is bolted to the end of the robot arm and the BarrettHand is secured to the arm adapter with its standard threaded locking ring. The arm adapter is also aquipped with an anti-rotation feature to prevent rotation during operations. In our case we will mount the hand in a KUKA LWR4+, this robotic arm derive from a technologic transfer between DLR and KUKA. The plan of the arm adapter is in the Annex .1

2.1.3 TorqueSwitch™ mechanism

Each of the three finger motors must drive two joint axes. The torque is channelled to these joints through a patented TorqueSwitch mechanism, whose function is optimized for maximum grasp security. The TorqueSwitch consists of a threaded shaft; a pair of Belleville spring washers and a spur gear with a threaded bore.

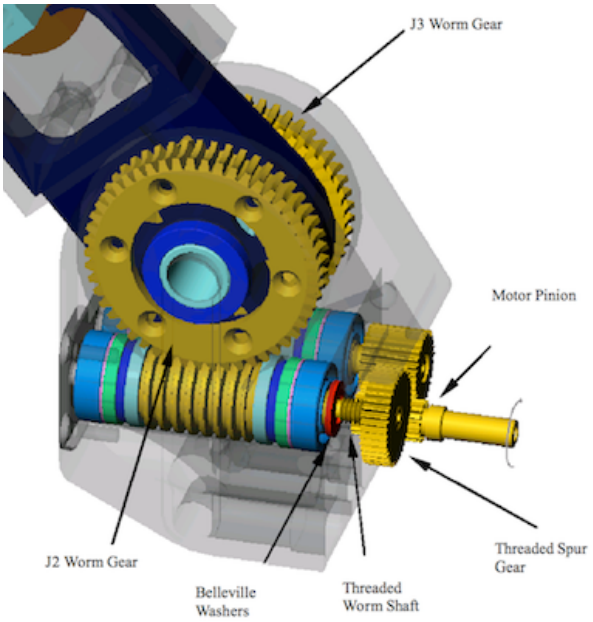


Figure 2: TorqueSwitch mechanism

The following description follows the progression of Figure 3. When the clutch is engaged, both worm gear drives and their corresponding finger links are coupled to the geared servomotor pinion. In this state, the ratios of motor position to joint position for the 1st and 2nd finger joints are 93.75:1 and 125:1, respectively. When a finger opens against its motion stop, the threaded spur gear is tightened against the Belleville spring washers with a known motor torque; thereby setting the threshold torque for disengaging the spur gear. If the inner finger link, while closing, contacts a target object of enough stiffness to increase the torque in the gear train above the threshold torque, the clutch will disengage from the Belleville spring washers. When the clutch is disengaged, the threaded spur gear "free-wheels" on the threaded shaft, allowing the motor pinion to turn without inducing motion in the inner link. Instead, only the smaller spur gear, solidly fixed to its shaft, is driven. This fixed spur gear actuates the worm gear drive for the fingertip. Thus, when the clutch is disengaged, the inner finger link remains motionless while the fingertip continues to move allowing the fingers to form-fit around any shape.

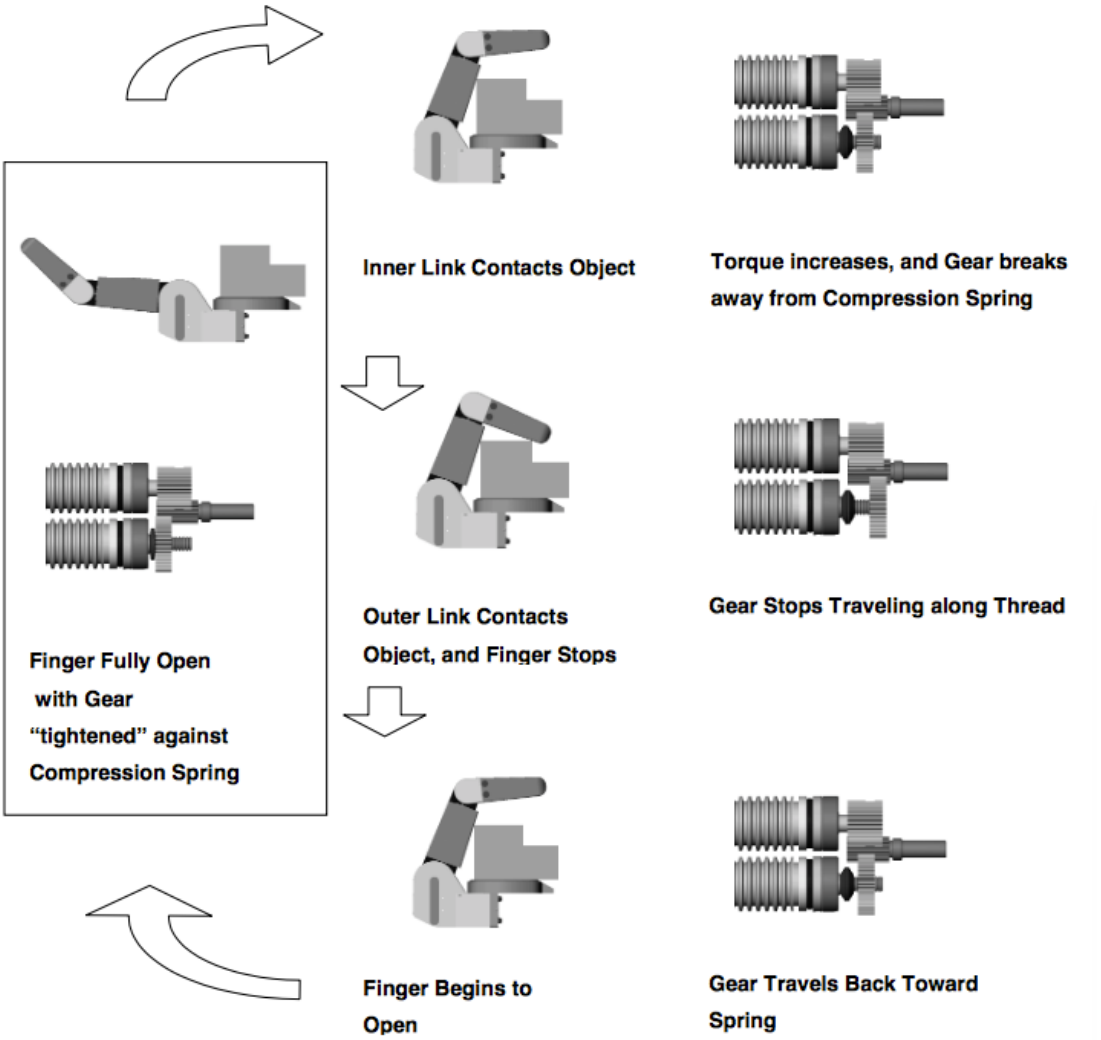


Figure 3: TorqueSwitch Mechanism

2.2 Control Modes

The BarrettHand can be used in either of the two modes: high-level Supervisory mode or low-level RealTime mode. Supervisory mode leverages the Motorola microprocessor onboard the Hand. This processor interprets incoming Supervisory Commands and then applies control signals across the set of four motion-control microprocessors. Supervisory mode allows commanding individual or multiple motors to close, open, and move to specific positions; it also provides the possibility of setting various configuration parameters and repotting positions. To automate grasping applications, you can write programs, scripts, or macros that sends and receive these text characters through the serial port. RealTime Mode extends the capability of Supervisory Mode. RealTime Mode enables users to close control loops in real time from their host PC or robot controller. The RealTime mode is interesting when we want to make real-time force control or change the speed of the motors.

In our case, we will work with the Supervisory mode for simplicity in the control of the hand, but it would be better to work with the RealTime mode because we will integrate the tactile sensors in the fingers, and we will have to program the detection of the object that we will grasp. And it is better to make this

detection in real time, in our case will have to make incremental close with little steps and see if we have touch something.

2.3 Commands for Supervisory Control mode

When the BarrettHand firmware is ready to process a command, it prints a prompt of "=>" to the host computer. The commands are entered as a single line terminated by a carriage return character. Once the firmware receives the carriage return it processes the line, executes the command, prints any error result and then prints a new prompt. Once a command has been started no configuration changes can be made until the command has completed or has been aborted. The commands can be preceded of a motor prefix for indicate which motor or motors are affected by the command that we desire: <motor prefix><command>.

VALUE	MOTOR
1	Finger 1
2	Finger 2
3	Finger 3
4	Spread
G	Fingers 1, 2 & 3
S	Spread
<none>	Finger 1, 2 & 3 and Spread

Table 1: Motor prefixes

COMMAND	NAME	PURPOSE	EXAMPLE
HI	Hand Initialize	Initializes the selected motor controller(s)	HI
O	Open	Opens the selected motor(s)	GO
C	Close	Closes the selected motor(s)	SC
IO	Incremental Open	Opens the selected motor(s) the given number of counts	12IO 500
IC	Incremental Close	Closes the selected motor(s) the given number of counts	GIC 2000
M	Move	Moves the selected motor(s) to the given position	13M 700
T	Terminate Power	Turns the selected motor(s)'s power off	ST
HOME	Home	Moves the selected motor(s) to position 0	SGHOME
FGET P	Get position	Gets and prints the current position's value for the selected motor(s)	SFGET P

Table 2: Movement Commands

3 Robot Operating System (ROS)

3.1 Utility

Robot Operating System (ROS) is an operating system for robots. As an operating system for PCs, servers or stand-alone devices, ROS is a complete operating system for service robotics. ROS is a meta operating system, something between the operating system and middleware. Middleware is a third-party software that creates a network for the exchange of information between different computer applications. It provides services close to an operating system (hardware abstraction, management of concurrency, processes,…) but also high-level features (asynchronous calls, synchronous calls, centralized database, system of parameterization of the robot,…). The main idea of a robotic OS is to avoid reinventing the wheel each time and offer standardized functionality by abstracting the hardware, like a classic OS for PCs. The philosophy of ROS is summarized in the five main principles:

- Peer to Peer: A complex robot consists of several computers or embedded boards connected via Ethernet as well as occasionally external computers to the robot for computer-intensive tasks. A peer to peer architecture coupled with a buffer system and a lookup system (a service name called master in ROS) allows each actor to interact directly with another actor, synchronously or asynchronously.
- Based on tools: Rather than a monolithic runtime ROS adopted a microkernel design that uses a large number of small tools to do the build and run the various ROS components. Each command is actually an executable. The advantage of this solution is that a problem on an executable does not affect the other, making the system more robust and scalable than a system based on a centralized runtime.
- Multi languages: ROS is not dependent on a language. To date, three main libraries have been defined for ROS; it can be programmed in Python, Lisp or C++. In addition to these three libraries, two experimental libraries are available that allow you to program in Java or Lu.
- Lightweight: To fight against the development of algorithms more or less linked with the OS robotics and therefore difficult to reuse subsequently, ROS developers would like that the drivers and other algorithms are contained in separate binaries. This ensures maximum reusability and especially keeps a reduced size. This mechanism makes it easy to use ROS because the complexity is only found in bookstores.
- Free and open source

The basic principle of a robotic OS is to run in parallel a large number of executables that need to exchange information synchronously or asynchronously. Moreover, the robotic OS will manage the competition in order to ensure effective access to robot resources.

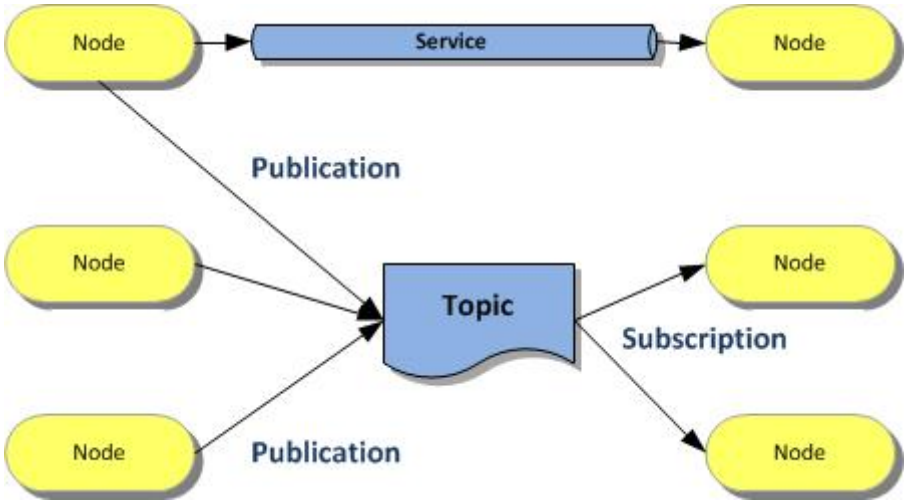


Figure 4: ROS Computation Graph

A node is an instance of an executable. A node may be a sensor, a engine, a treatment algorithm, a monitoring algorithm… Each node that we start has to be declared in the Master. Here we find the microkernel architecture where each node is an independent resource.

The exchange of information is carried out either asynchronously via a topic or synchronously via a service. A topic is an information transport system based on the system of the subscription/publication (subscribe/publish). One or more nodes can publish information on a topic and one or more nodes can read information on this topic. The topic is somehow an asynchronous bus much like an RSS feed. The type of information that is published (the message) is always structured in the same way.

A message is a composite data structure. A message consists of a combination of primitive type (string, Boolean, integer, float …) and message (the message is a recursive structure).

In opposition of the topic, the service has a synchronous communication between two nodes. This concept is similar to the concept of remote procedure call.

3.2 RQT

RQT is a software framework of ROS that implements the various GUI tools in the form of plugins. This software makes easier to manage the various windows of each GUI tools on the screen at one moment. RQT allow making easier the implementation of interfaces in ROS, instead of open each interface in different windows. The advantages of RQT framework are standardized common procedures for GUI, dockable multiple widget in a single window, easily turn your existing Qt widgets into RQT plugins and support multi-platform (QT and ROS) and multi-language (Python, C++). To simplify the way to design the interface we can use Qt Designer and generate a .ui file. We load the file in our program:

```
$ loadUi(%PATH_UI_FILE%, self._widget)
```

Work with .ui file provides advantages like a good maintainability (because we are separating GUI design from source code) and make easier to change design (we have only to change the design with Qt Designer).

3.3 MoveIt

Moveit is a set of software tools for building mobile manipulation; it helps to integrate perception, kinematics, motion planning, trajectory processing and execution. This tool lets us create our own environment for a robot, allowing us to simulate its interaction with the world.

For the integration of the robot in MoveIt, we need a package with the meshes of the robot and an URDF (Universal Robotic Description Format) file. The URDF file contains all the position and relationships between each part of the robot, in this file we create the visualization of the robot, either creating it with geometric forms or loading the correspondent meshes.

3.3.1 URDF file

Before getting into MoveIt, we first have to talk about the URDF file, as we said before, this file contains all the positions and kinematic relationships of every part, but not just that, they have too the visual content of the files, such as meshes and color configuration.

As there was not URDF files of the BarrettHand 262, we had to create it ourselves. We could find a URDF file of the BarrettHand 282 in GitHub. This file helped us to understand the URDF files and how they define the positions, the relationships, and how they load the CAD files and all the visualization parameters. Otherwise, this file was for the BarrettHand 282, which is the newest BarrettHand model, which is a bit different from the previous 262, so all the meshes where different that our model.

In order to have a more accurate model of our hand, we contacted with BarrettHand laboratories, who sent us the 3D model of the BarrettHand 262. This was a Solidworks model of the hand, so we had to convert each piece of the hand into dae files, as it is the format used in the URDF. The original urdf of the 282 hand used stl format, but the dae files where much better, as the stl files gave a lot of collisions when defining the joints in MoveIt.

Once we had all the meshes and a URDF reference file, we start to creating the BarrettHand 262 urdf model. The dimensions and reference axes of each part are different that the used in the BarrettHand 282, so we had to remake all the URDF from zero. First we define some variables that will be used later.

bh262.urdf.xacro

```
7  <xacro:property name="dist_joint_offset" value="0.75" />
8  <xacro:property name="dist_joint_multiplier" value="0.3" />
9  <xacro:property name="spread_joint_offset" value="0" />
10 <xacro:property name="spread_joint_multiplier" value="1" />
11
12 <xacro:property name="PI" value="3.1415926535897931"/>
13 <xacro:property name="PI_2" value="1.570796327"/>
14 <xacro:property name="joint_damping" value="100.0" />
15 <xacro:property name="joint_friction" value="1.0" />
16 <xacro:property name="joint_effort_limit" value="60.0" />
17 <xacro:property name="joint_velocity_limit" value="2.0" />
18 <xacro:property name="mechanical_reduction" value="1.0" />
19 <xacro:macro name="bhand_macro" params="parent name *origin">
```

Once the variables are defined, we can load the first part of the hand. We will use the palm of the hand as the base, as it is the part on which all the other parts will be attached.

bh262.urdf.xacro

```
23 <!-- HAND BASE_LINK (RED) -->
24 <link name="${name}_base_link">
25   <inertial>
26     <mass value="1.0" />
27     <origin xyz="0 0 0" />
28     <inertia     ixx="1.0"   ixy="0.0"     ixz="0.0"     iyy="1.0"
   iyz="0.0"  izz="1.0" />
29   </inertial>
30   <visual>
31     <origin xyz="0 0 0" rpy="1.570796327 0 0" />
32     <geometry>
33       <mesh  filename="package://bhand_description/meshes/palm_26
   2.stl"/>
34     </geometry>
35     <material name="Gold">
36       <color rgba="199 154 48 1.0"/>
37     </material>
38   </visual>
39   <collision>
40     <origin xyz="0 0 0" rpy="0 0 0" />
41     <geometry>
42       <mesh  filename="package://bhand_description/meshes/palm_26
   2.stl"/>
43     </geometry>
44     <contact_coefficients kd="1.0" kp="1000.0" mu="0"/>
45   </collision>
46 </link>
```

Here we load the base link, it consists of three parts. First we have the inertial properties of the link, this section is optional, and as our movements will not be at high speeds, or for example in this case, there will not be movement; they are not really necessary, in any case, we have used the inertial values from the BarrettHand 282. The mass value represents the mass of the hand, while the origin refers to the position of the inertial reference frame of the link, this origin refers to the center of gravity of the piece. In the line 28 are the inertial values, these values refer to the 3x3 rotational inertia matrix, as this matrix is symmetrical, we just have to define the 6 values above diagonal of the matrix: ixx, ixy, ixz, iyy, iyz, izz.

The second group is where we define the visualization of the link, defining the shape of the object or using a created mesh. First we specify the origin, this sets the position of the piece to its reference frame, the xyz values refer to the positions while the rpy are the rotation values (in radians) in the same xyz axis. All this values where set to zero, as we prefer to move the pieces in the joints, simplifying all the work. The geometry property is where we define how the piece looks like, this property is necessary for the URDF, in this case we are using the stl files, so we just have to define the route where it can find the piece in the same package the

URDF file is hosted. Finally in this section, we have the material property, which is simply the color of the link written in rgba code.

The last section sets the collision properties, here we can define a form that will be the responsible of setting the auto-collision properties later in MoveIt, it will limit the movement of the joints so they do not touch between them. This form can have different origin and form than the visual form. For example, if there is any delicate part of the robot, which we do not want to be touched by anything, we can define a big cylinder instead of the mesh of the part, therefore, the piece will have a security ratio around it.

After this will create a first joint between the palm and the world, so the hand will be fixed in one place.

bh262.urdf.xacro

```
48 <joint name="${name}_base_joint" type="fixed">
49   <insert_block name="origin"/>
50   <parent link="${parent}"/>
51   <child link="${name}_base_link"/>
52 </joint>
```

At the first line, we define the name of the joint and which type of joint it is, in this case a fixed joint. In the second line we insert the block origin, this refers to the macro we previously created at the beginning with the other properties.

After we have settled the base of the hand, we can start mounting the first finger, the easiest one to begin with is the finger number three, as it does not move with the spread, it is fixed to the palm. In the barrethand262, each finger has four parts: the knuckle, the middle finger, the phalanges and the fingertip, so will have three joints and an extra joint the finger to the palm.

bh262.urdf.xacro

```
48 <!-- finger 3 -->
49 <!-- finger 31 link -->
50 <link name="${name}_finger_31_link">
51   <inertial>
52     <mass value="0.1"/>
53     <origin xyz="0 0 0" rpy="0 0 0"/>
54     <inertia ixx="0.01" ixy="0" ixz="0" iyy="0.01" iyz="0"
   izz="0.01" />
55   </inertial>
56   <visual>
57     <origin xyz="0 0 0" rpy="0 0 0" />
58     <geometry>
59       <mesh filename="package://bhand_description/meshes/knuckle
   _fixed.stl"  />
60     </geometry>
61     <material name="Black"/>
62   </visual>
63   <collision>
64     <origin xyz="0 0 0" rpy="0 0 0" />
65     <geometry>
66       <mesh filename="package://bhand_description/meshes/knuckle
   _fixed.stl"/>
```

```
67      </geometry>
68      <contact_coefficients kd="1.0" kp="1000.0" mu="0"/>
69    </collision>
70  </link>
```

This is the link that will attach the finger to the hand, as we have seen before, there are three different groups of properties: inertial, visual and collision. In this case the collision has a new parameter, contact_coefficients, which refers to the contact parameters such as friction coefficient(mu), stiffness coefficient(kp) and dampening coefficient(kd). This piece will be fixed to the palm of the hand.

bh262.urdf.xacro

```
77  <!-- joint between BH_palm_link and BH_finger_31_link -->
71  <!-- finger3 is the center finger and is fixed -->
72  <joint name="${name}_j31_joint" type="fixed">
73    <parent link="${name}_base_link"/>
74    <child link="${name}_finger_31_link"/>
75    <origin xyz="0 0.05 0.037" rpy="1.57079632679 0 0"/>
76  </joint>
```

This is another fixed joint, as we have seen at the beginning when attaching the palm to the environment. This joint is called j31, as it is the first joint of the third finger. The child link is the piece we have just added before and the parent is the palm of the hand. The origin in the joints set the position of the child link in reference to the parent link origin, this distances are in meters, it has also been rotated 90º in the x axis, so it fits the palm of he hand.

bh262.urdf.xacro

```
85  <!-- finger 32 link -->
86  <link name="${name}_finger_32_link">
87    <inertial>
88      <mass value="0.1"/>
89      <origin xyz=".5 0 0" rpy="0 1.57079632679 0"/>
90      <inertia ixx="0.01" ixy="0" ixz="0" iyy="0.01" iyz="0"
    izz="0.01" />
91    </inertial>
92    <visual>
93      <origin xyz="0 0 0" rpy="0 0 0" />
94      <geometry>
95        <mesh
    filename="package://bhand_description/meshes/finger.stl" />
96      </geometry>
97      <material name="Blue" />
98    </visual>
99    <collision>
100     <origin xyz="0 0 0" rpy="0 0 0" />
101     <geometry>
102       <mesh
    filename="package://bhand_description/meshes/finger.stl" />
103     </geometry>
104     <contact_coefficients kd="1.0" kp="1000.0" mu="0"/>
105   </collision>
```

```
106 </link>
```

Now, the middle part of the finger is loaded, this is the first part of the finger that can rotate so the hand can close, therefore the joint will be different that the fixed joints we have done before.

bh262.urdf.xacro

```
108     <!-- joint between BH_finger_31_link and BH_finger_32_link
    -->
109 <joint name="${name}_j32_joint" type="revolute">
110   <parent link="${name}_finger_31_link"/>
111   <child link="${name}_finger_32_link"/>
112   <origin  xyz="0 0 0"
113            rpy="3.1415 0 0"/>
114   <axis xyz="1 0 0"/>
115   <limit effort="${joint_effort_limit}" lower="0.0"
    upper="2.44" velocity="${joint_velocity_limit}"/>
116   <dynamics damping="${joint_damping}" friction="${joint_fricti
    on}"/>
117 </joint>
```

In this case we specify at the beginning that this is a revolution joint, as we have seen, the parent link is the previously loaded link, the knuckle that is attached to the palm. This is a joint that has rotation movement, so we have to specify the axis around it does, as we do in the axis code line.

At the end of the joint definition, we can set a limit of speed and effort that the joint can reach, as well as the friction and damping of it. These values were set at the beginning o the URDF as general properties and are the same values as in the bh282.

The next link connects the middle finger and the last phalange. The fingertip of the BarrettHand 262 is quite different to the bhand282, in the second case, all the fingertip is formed by just one piece, while in the BarrettHand 262, it consists of two parts. The first piece is the responsible of the revolution movement and the second one is just the fingertip, with a fixed joint.

bh262.urdf.xacro

```
129 <!-- finger 33 link -->
130 <link name="${name}_finger_33_link">
131   <inertial>
132     <mass value="0.1"/>
133     <origin xyz="0 0 0" rpy="0 0 0"/>
134     <inertia ixx="0.01" ixy="0" ixz="0" iyy="0.01" iyz="0"
    izz="0.01" />
135   </inertial>
136   <visual>
137     <origin xyz="0 0 0" rpy="0 0 0" />
138     <geometry>
139       <mesh filename="package://bhand_description/meshes/finger
    _tip.stl" />
140     </geometry>
141     <material name="Blue" />
```

```
142   </visual>
143   <collision>
144     <origin xyz="0 0 0" rpy="0 0 0" />
145     <geometry>
146       <mesh filename="package://bhand_description/meshes/finger
    _tip.stl" />
147     </geometry>
148     <contact_coefficients kd="1.0" kp="1000.0" mu="0"/>
149   </collision>
150 </link>
```

This link has a revolute joint too, but with a particular property. The BarrettHand can close the two phalanges of the, but they are dependent between them, this means, the fingertip of the three fingers close according to the phalange that is attached to the palm, so they can not move separately.

bh262.urdf.xacro

```
152 <!-- joint between BH_finger_32_link and BH_finger_33_link -->
153 <joint name="${name}_j33_joint" type="revolute">
154   <parent link="${name}_finger_32_link"/>
155   <child link="${name}_finger_33_link"/>
156   <origin xyz="0 0 .07" rpy="0 0 0" />
157   <axis xyz="1 0 0"/>
158   <limit effort="${joint_effort_limit}" lower="0.0"
    upper="0.84" velocity="${joint_velocity_limit}"/>
159   <dynamics damping="${joint_damping}"
    friction="${joint_friction}"/>
160   <mimic joint="${name}_j32_joint" multiplier="${dist_joint_mul
    tiplier}" offset="${dist_joint_offset}"/>
161 </joint>
```

The main difference between this revolute joint and the previous one is the second line from the bottom. This mimic joint property will take the movement of a reference joint and use it in a second one. In this case, it will take the rotation of the j32 joint, the one between the finger and the knuckle, the multiplier parameter sets the number of revolutions that will do the child joint per revolution in the parent joint, it is defined as dist_joint_multiplier, which is set at the beginning of the file as 0,3 on the other side. Moreover, is the offset parameter, which is the fixed difference of angle between the two joins, in this case is the angle the fingertips has when the hand is completely open, in the BarrettHand it is 43º, 0,75 rads

Finally, the last piece of the fingers is the fingertip that has a simple fixed joint, as it is attached to the previous link with four screws.

bh262.urdf.xacro

```
173 <!-- fingerprint 3 link -->
174 <link name="${name}_fingerprint_3_link">
175   <inertial>
176     <mass value="0.1"/>
177     <origin xyz="0 0 0" rpy="0 0 0"/>
```

```
178     <inertia ixx="0.01" ixy="0" ixz="0" iyy="0.01" iyz="0"
    izz="0.01" />
179   </inertial>
180   <visual>
181     <origin xyz="0 0 0" rpy="0 0 0" />
182     <geometry>
183       <mesh
    filename="package://bhand_description/meshes/fingerprint.stl"/>
184     </geometry>
185     <material name="Black"/>
186   </visual>
187   <collision>
188     <origin xyz="0 0 0" rpy="0 0 0" />
189     <geometry>
190       <mesh
    filename="package://bhand_description/meshes/fingerprint.stl"/>
191     </geometry>
192     <contact_coefficients kd="1.0" kp="1000.0" mu="0"/>
193   </collision>
194 </link>
195
196 <!-- joint between fingerprint_3 and finger_33_link -->
197 <joint name="${name}_fp3_joint" type="fixed">
198   <parent link="${name}_finger_33_link"/>
199   <child link="${name}_fingerprint_3_link"/>
200   <origin xyz="0 0 0.04" rpy="0 3.14159265359 0"/>
201 </joint>
```

All this joints and links are referred to the finger number three, which is the finger placed in the middle of the other two with the spreads. Now we have to load the spread supports of the fingers one and two, so the fingers can move around the hand doing the different grasping positions.

bh262.urdf.xacro

```
205 <!-- finger 1 spread link -->
206 <link name="${name}_finger_1_spread_link">
207   <inertial>
208     <mass value="0.1"/>
209       <origin xyz="0 0 0" rpy="0 0 0"/>
210       <inertia ixx="0.01" ixy="0" ixz="0" iyy="0.01" iyz="0"
    izz="0.01" />
211   </inertial>
212   <visual>
213     <origin xyz="0 0 0" rpy="0 0 0" />
214     <geometry>
215       <mesh filename="package://bhand_description/meshes/knu
    ckle.stl" />
216     </geometry>
217     <material name="Green" />
218   </visual>
219   <collision>
220     <origin xyz="0 0 0" rpy="0 0 0" />
221     <geometry>
```

```
222       <mesh filename="package://bhand_description/meshes/knuck
    le.stl" />
223     </geometry>
224   </collision>
225 </link>
```

Once the spread knuckle is loaded we will proceed to do the joint between the palm and the knuckle, both have the same type of joint with a few differences.

bh262.urdf.xacro

```
227 <!-- joint between BH_palm_link and BH_finger_1_spread_link -->
228 <joint name="${name}_j1_spread_joint" type="revolute">
229   <parent link="${name}_base_link"/>
230   <child link="${name}_finger_1_spread_link"/>
231   <origin xyz="-0.025 0 0.003" rpy="1.5708 0 0" />
232   <axis xyz="0 -1 0"/>
233   <limit effort="${joint_effort_limit}" lower="0"
    upper="3.1416" velocity="${joint_velocity_limit}"/>
234   <dynamics damping="${joint_damping}"
    friction="${joint_friction}"/>
235 </joint>
236
237 <joint name="${name}_j2_spread_joint" type="revolute">
238   <parent link="${name}_base_link"/>
239   <child link="${name}_finger_2_spread_link"/>
240   <origin xyz="0.025 0 0.0035" rpy="1.5708 0 0" />
241   <axis xyz="0 1 0"/>
242   <limit effort="${joint_effort_limit}" lower="0"
    upper="3.1416" velocity="${joint_velocity_limit}"/>
243   <dynamics damping="${joint_damping}"
    friction="${joint_friction}"/>
244   <mimic joint="${name}_j1_spread_joint"
    multiplier="${spread_joint_multiplier}"
    offset="${spread_joint_offset}"/>
245 </joint>
```

Both joints are revolute joints around the same axis, but in different directions, the finger two rotates in the positive direction of the Y-axis, while the finger one rotates in the negative direction. As both of the spreads rotate together but in opposite ways, we have defined a mimic joint so it can oth will rotate solidary, which means, the same angle at the same time. Notice that the rotation of the two spreads could be defined in the same axis and use a value of -1 in the mimic multiplier. It only lefts to put together the spread to the knuckle of each finger.

bh262.urdf.xacro

```
269 <!-- joint between finger_1_spread and BH_finger_11_link -->
270 <joint name="${name}_j11_joint" type="fixed">
271   <parent link="${name}_finger_1_spread_link"/>
272   <child link="${name}_finger_11_link"/>
273   <origin xyz="0 0.0337 0.050" rpy="0 3.1415 0" />
274 </joint>
```

Finally, when the two spread supports are installed, it only lefts to place the fingers, first the fixed joint of the knuckles, and then the rest of the finger, which uses exactly the same references and values as the finger number three.

3.3.2 MoveIt set up

MoveIt has a setup assistant, in which we can settle all the self-collisions between the parts of the robot, the joints that we will control and all their positions. All this features will be really useful in the simulation, like for example the self-collisions, as they will avoid that parts mount over other parts, creating awkward situations in the graphical simulation.

For running the the moveit setup assistant we will have to run a it from the terminal, first we will have open a terminal and introduce:

```
$ roslaunch moveit_setup_assistant setup_assistant.launch
```

This will open the MoveIt setup assistant window in the start page tab, so now the first step is to load the URDF file we have previously created with all the pieces and joints.

The second step in the configuration of the robot is to create a virtual joint between the robot and the environment around him, it is important to know how the robot is going to interact with the environment so we know if it will be fixed to the ground or on the other side it will be able to move around it.

Once we have created the joint to the world and we know how it will related to it, we will have to create the planning groups. This groups will contain different parts of the robot that work together, for example an arm, a finger,... Each group will have its own rotation plans and its kinematic group, with two revolute joints. We can define the planning groups as a chain of elements from the beginning to the top, or as a list of joints or links. The links are each part of the robot, while the joints are the position relation between each part or link of the robot. For each group, we have to define a kinematic solver; the kinematic solver is the calculator that will help to calculate all the acceleration and movement problems of the robot.

Here we will define the three fingers, we just have to set one for each finger, as they have solidary phalanges, and one for the spread. This is enough for moving all the hand thanks to the mimic joints we have introduced in the URDF file.

After defining the groups is the robot poses tab. In this step we can define different positions for each group, for example, we can settle a home position, standby position,... All this positions will be useful while programming the hand, in order that we will just have to call the position to make the hand move to that final pose, instead of ordering the movement of each element separately. To create a pose, we have to choose one group (only one pose per group can be defined at a time) and click add pose, this will show us an interface with sliders, with which we will settle the desired position. In this tab we can create different positions as fingers opened and closed, and so to for the spread.

The next tab is the end effectors, although we have already defined the kinematic groups before, we can create end effectors group, this property will allow more functions to the chosen parts such as more interaction capacities to the environment we will create for the robot.

The last group of joints and links we can define is the passive joints tab. These are the joints that are unactuated. These joints are specified separately because they will not be used during the kinematics calculation, so they cannot be

directly controlled. If the robot has any kind of unactuated joints, they should be labeled as passive joints, but in this case there are not.

Finally, once we have defined all the joints, links and positions of our robot, so we just have to generate the package. This package will contain a set of launch and configuration files in the folder you choose. This is the basic package to use later with Rviz, another tool that will help for future works with planning and interaction of the hand.

3.3.3 RViz

All the MoveIt setup is intended to improve the RViz work, we will not get much into it. Our intention with all these settings is to leave everything ready for the upcoming people that want to use the BarrettHand in RViz.

Anyway, we there is a specific feature in Rviz where we must pay attention before doing anything. We will first parameter we have to settle is the MotionPlanning parameters, we must take care of the planning scene and the fixed frame, as this is the plane that support the robot and its environment, we have to check its motion planning frame is world, if it is not, then change it.

Once it is settled, we can finally start interacting with the robot. In the top menu of Rviz, there is the Interact button, after pressing it you will see a few interactive markers out of the planning group you have chosen in the planning group. These markers will allow us to move the different links with their position relationships and joints. When moving the links around, it will be noticed that if any parts touch, they will appear in red that is thanks to the collisions and configurations we have specified previously with MoveIt. Now everything is ready for programming and interact with the hand.

4 Barrett Hand Control

To control the Barrett Hand we have to create several programs. The first program that will have to create is that which allows us to connect with the hand. It would connect through the serial port, it would allow us to obtain the position of the fingers and send commands to the hand. It would be at the same time a publisher and a server. It will publish the positions in a topic so they are available to be processed by other programs. The second program that will have to create is the one that allows creating the interface, send the commands and show the positions. It would be a listener and a client of the first program. It would have to be subscribed at the topic generated by the first program.

4.1 Server program

4.1.1 Procurement of the program

We have to make a program that allows the connection with the hand through the serial port. We have received from our tutor the basic package of programs that distribute Barrett Company to control its product. This package contains a server program and a client program. We will use this server program to make the connection with the hand.

This package was created in rosbuild so the first job that we have do is migrate from rosbuild to catkin, that is much easy to use because it is adhered to FHS layout that allows easier installation of ROS on a variety of operating systems and architectures. To realize this migration, we have first of all sought for a tutorial that explains the way to make properly this migration. The first thing that we have to change is the package layout of the workspace. The layout of workspaces is not very different between rosbuild and catkin packages. There are some differences as packages no longer contain a manifest.xml file, this has been superseded by package.xml and packages no longer contain a Makefile. Packages still contain a CMakeLists.txt file, but the content and format of this file has changed drastically. In the tutorial of the ROS Organization wiki we can found a quick guide showing the mapping between rosbuild CMake macros and catkin CMake macros.

Once we have made the migration from rosbuild to catkin, we can begin with the tests to familiarize with the basic package programs. Within the basic package, we can found the server program that allows other programs to send commands that we want that the hand execute. This server program is already compiled and ready to use. The same program publishes on a topic the positions of each finger and spread. The message published is four floats with names S, F1, F2 and F3, which belong to spread and three fingers. This allows any program that subscribes to this topic to get the motors positions; this permits to control the hand optimally. In the package also comes a basic client program that sends the command that we want to the server. We will use this program to inspire us to create our client program, which should be enable to send the command we want. But instead of receiving the command from the terminal, our command is requested by the graphical interface that we will create.

4.1.2 Description of the server program

The server program is written in C++ language. We will describe this program. First of all, this program has to include basic libraries and specific libraries with the basic commands of Barrett Control.

BHand_Server.cpp

```
1     #include <ros/ros.h>
2     #include <std_msgs/String.h>
3     #include “handif.h”
4     #include “../srv_gen/include/BHand_Control/CommandBH.h”
5     #include “../msg_gen/include/BHand_Control/BarrettState.h”
6     #include <boost/thread/mutex.hpp>
7     #include <boost/thread/thread.hpp>
8     #include <ecl/threads.hpp>
```

We have to include the library *handif.h* because is the library that contain all the commands that we can use to connect with the hand. *Command.h* and *BarrettState.h* are the libraries that contain the program of the service and the message. We include the library *mutex.hpp* because when two or more threads need simultaneous access to a shared resource, the system needs a synchronization mechanism to ensure that only one thread at a time uses the resource. Mutex is a synchronization primitive that grants to a single thread exclusive access to the shared resource. If a thread acquires a mutex, the other thread that wants to acquire the mutex is interrupted until the first thread releases the mutex. In our case we have two threads, one that send the commands to the hand and the second one that take the positions of the fingers.

The program has two threads, the first one is the responsible of execute the command that we wish.

BHand_Server.cpp

```
15 bool    executeCommand(BHand_Control::CommandBH::Request    &req,
   BHand_Control::CommandBH::Response &res)
16 {
17     mutex.lock();
18     double error= handif_hand_raw(req.command.c_str(), true);
19     mutex.unlock();
20     res.error= (int) error;
21     return true;
22 }
```

We define the function *executeCommand* that has as input the requested command (*req*) and as output a response (*res*) that can be an error message. We use a mutex to ensure that this thread is the only one that uses the port serial. The function *handif_hand_raw* (line 18) send a raw command to the hand, the first input is the command that we want to send and the second one is a boolean that in the case that is True it will make the program wait until the end of the process. Once the process is finished we have to unlock the mutex to allow an other thread to use the port serial. We give to the variable *res.error* the value of the output of the function *handif_hand_raw*.

The second thread that we have in this program is the responsible of getting the positions of the motors of the hand and publish them in the topic.

BHand_Server.cpp

```
24 void publisherThread()
25 {
26     ros::NodeHandle n;
```

```
27      ros::Publisher  pub=  n.advertise<BHand_Control::BarrettStat
    e>("BarrettState",1000);
28      ROS_INFO("Ready to publish positions");
29      ros::Rate loop_rate(2);
30      int jointValues[4];
31      while(ros::ok())
32      {
33          mutex.lock();
34          handif_getjoints(jointValues);
35          mutex.unlock();
36          BHand_Control::BarrettState msg;
37          msg.F1=jointValues[0];   msg.F2=jointValues[1];   msg.F3=
    jointValues[2]; msg.S=jointValues[3];
38          pub.publish(msg);
39          ros::spinOnce();
40          loop_rate.sleep();
41      }
42  }
```

We define the function *publisherThread*. The first thing that we have to do is create a handle to this process's node, after we will define the topic that we have call *"BarrettState".* We are going to publish a message of type *BarrettState*, that we have previously define in the file *BarrettState.msg,* on the topic chatter. This lets the master tell any nodes listening on *BarrettState* that we are going to publish data on that topic. The second argument is the size of our publishing queue. In this case if we are publishing too quickly it will buffer up a maximum of 1000 messages before beginning to throw away old ones. We print a message on the screen to see if the program has effectively created the topic. A `ros::Rate` (line 29) object allow you to specify a frequency that you would like to loop at. It will keep track of how long it has been since the last call to `Rate::sleep()`, and sleep for the correct amount of time. We define a list called `jointValues` (line 30) of four variables. We define a restriction that only the process is done when it is available ROS. `ros::ok()` will return false if:

- a SIGINT is received (Ctrl-C)
- we have been kicked off the network by another node with the same name
- `ros::shutdown()` has been called by another part of the application
- all `ros::NodeHandles` have been destroyed

When ROS is available we will use the function `handif_getjoints()` that returns a list of four arguments with the value of the four motors. We use a mutex to ensure that this thread is the only one that uses the port serial. We define *msg* like a BarrettState's message, and we give to each variable of *msg* the values that we have got previously of the positions of the motors. With `pub.publish()` we broadcast the message to anyone who is connected to the topic. We use the `ros::Rate` object to sleep for the time remaining to let hit our 2 Hz publish rate.

We define the main function:

BHand_Server.cpp

```
44  int main(int argc, char **argv)
45  {
46      ros::init(argc,argv,"BHand_Server");
47      ros::NodeHandle n;
```

```
48        handif_hand_connect();
49        handif_graspinitialize();
50        ros::ServiceServer service=n.advertiseService("CommandBH",
      executeCommand);
51        ROS_INFO("Ready to receive commands.");
52        boost::thread t(&publisherThread);
53        ros::spin();
54        return 0;
55    }
```

In the main function we have to initialize ROS, we will specify the name of our node. The name that we have use is *"BHand_Server"*. We have to create also a handle to this process's node. The function `handif_hand_connect()` execute the connection of the hand and `handif_graspinitialize()` initialize the hand. This two functions are defined in the library *handif.h*. In the line 50, we create and advertise over ROS the service, and we call the function *executeCommand.* We print a message in the screen to see when the program is ready to receive an other command. Finally we will execute the publisher in a new thread.

Now that we have created the server program, we can begin to create our graphic interface that will be at the same time the client and subscriber of the server program.

4.2 Graphic interface

As explained above, we will create a graphic interface for faster hand control. We have sought to create a simple and intuitive interface to allow a person with little knowledge in robot control to execute maneuvers easily. The graphical interface has to be integrated in ROS too, so we will use RQT that allows to open multiple interfaces at the same time and that they are all connected to ROS. For the interface design we will use QT Designer that allows a simple way to create complete interfaces, its extensive library of plugins offers a wide range of possibilities to create our interface. We have chosen to write our program in python, which offers a simpler language and that we are more familiar. QT Designer generates a .ui file containing the graphic interface. We'll have to load this .ui file in our python program.

4.2.1 Design of the graphic interface

Our window is divided into three tabs, the first one, called "Start" (Figure 1), it contains only a start button, which executes the initialization of the hand, an action that is done automatically when the program is executed but you have to redo each time the engine stops running or when there was a problem that has blocked a motor.

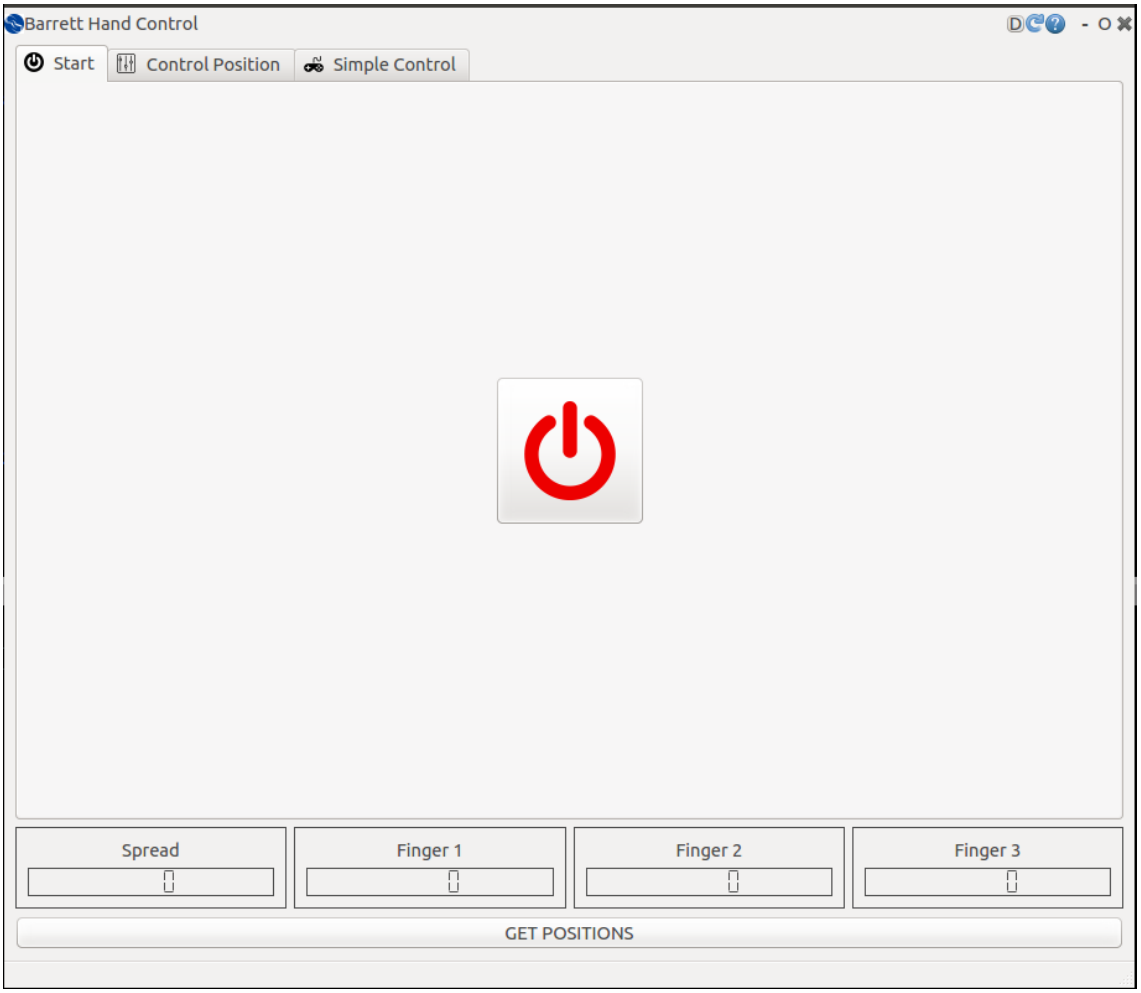


Figure 5: Start Tab of the BarrettHand Control Interface

The second of the tabs, called "Control Position" is the one that contains the hand control by sliders (indication 1 - Figure 2). We have opted for using sliders to make the control because it is very easy to select a value and allows visualizing the value we have asked intuitively. There is a slider for each finger and the spread. We also have a checkbox (indication 2 - Figure 2) that allows the simultaneous movement of the three fingers, when we move one finger the other fingers copy its position. There is also a LCD (indication 3 - Figure 2) for each engine to display the value we desire. In this same tab, we have also another way to move the fingers by steps, whose value can be modified by the spin box (indication 4 - Figure 2). In the case of the spread its values range goes from 0 to 200 and in the case of the fingers it goes from 0 to 2000. The default values are 20 for the spread and 200 for the fingers. To execute the steps in the value we have to click on the + and - buttons (indication 5 - Figure 2). There is also a button (indication 6 - Figure 2) that allows disconnecting all motors if there is any problem during the movement of any of the motors. We also have a light (indication 7 - Figure 2) indicating the status of the motors, green is when the motors are initialized and red when the motor is disconnected.

In the last tab, we have a simple control of the hand, allowing performing simple movements with buttons. The moves you can perform are as follows in the order from left to right:

- Close: Closes the three fingers to its maximum position. If it finds opposition it would stop at that position.
- Open: Opens the three fingers to its minimum position. If it finds opposition it would stop at that position.
- Shut: It moves every finger 2000 steps each time from the position that it was. But if the fingers were at different positions it would close every finger of the same steps.
- Unfold: It moves every finger 2000 steps each time from the position that it was. But if the fingers are at different positions it would open every finger of the same steps.
- Shake hand: It made a series of moves to end up getting a position to shake hands with a user. It is a movement of demo, which can be modified to suit the user from the python program.
- Demo: It is a movement of demo, which can be modified to suit the user from the python program.
- Pinch: It moves the fingers to put them all on the same side of the palm, to make a stronger grasp.
- Spread: It moves the fingers to oppose them, to make a precision grasp.

Finally in our GUI we have a LCD (indication 8 - Figure 2) for each motor that shows the actual position of the four motors. To start getting the values we have to click on the "GET POSITIONS" button.

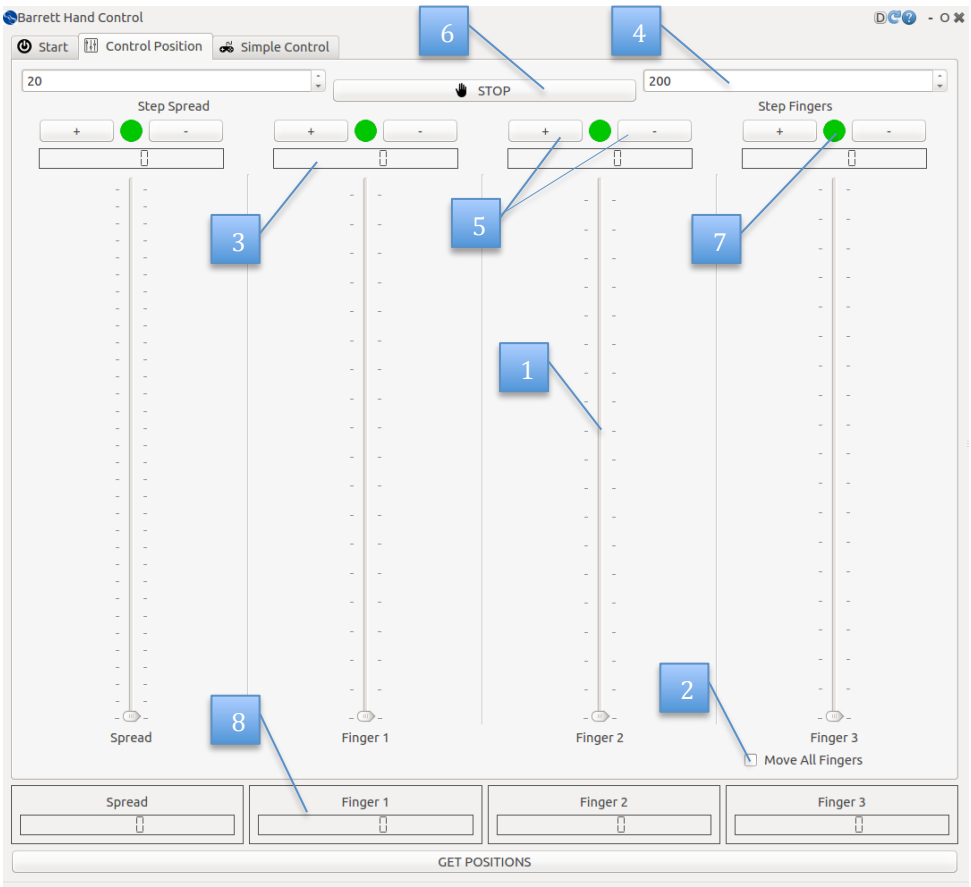


Figure 6: Control Position Interface

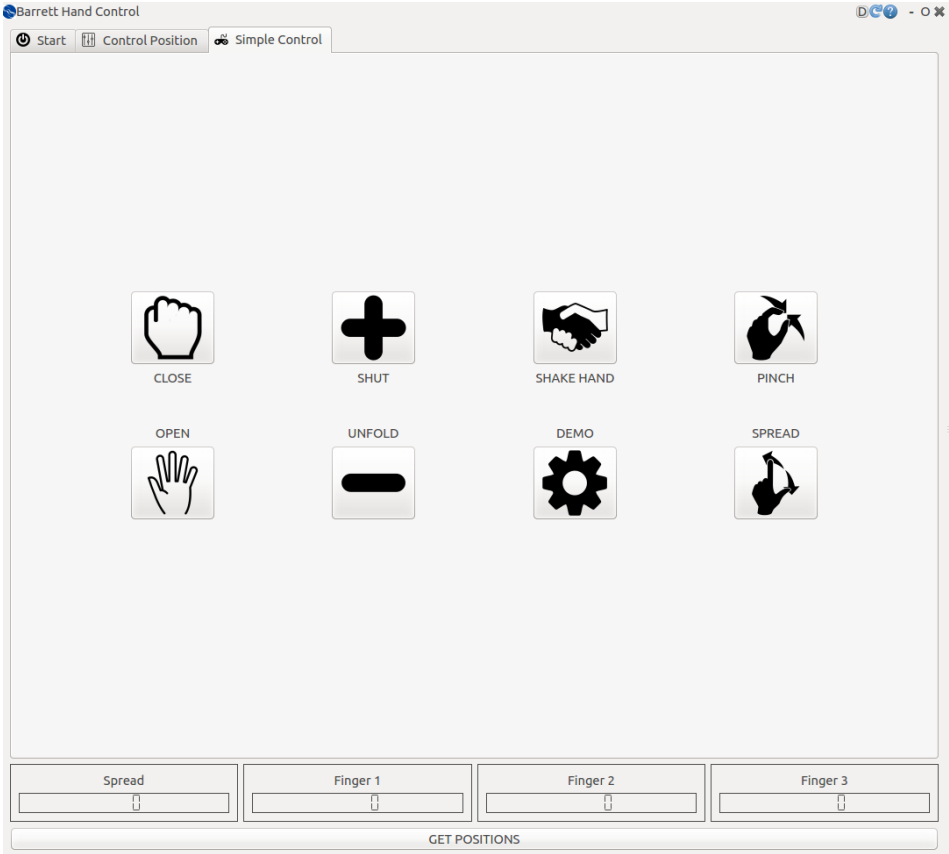


Figure 7: Simple Control Interface

4.2.2 Integration of the program in ROS

We have to integrate our custom interface into ROS GUI framework rqt. The first thing that we have done is create an empty package: `$ catkin_create_pkg rqt_bhand_control rospy rqt_gui rqt_gui_py`. The next step is declaring the plugin in *package.xml.*

package.xml

```
51 :
52 <export>
55   <rqt_gui plugin="${prefix}/plugin.xml"/>
58 </export>
59 :
```

The next step has been the creation of the *plugin.xml* file that contains additional meta information regarding the plugin.

plugin.xml

```
1  <library path="src">
2    <class name="BHand Control Interface"
   type="rqt_bhand_control.bhand_control_module.BHandControlPlugin"
   base_class_type="rqt_gui_py::Plugin">
3      <description>
4        Interface to control Barrett Hand.
5      </description>
6      <qtgui>
7        <group>
```

```
8           <label>Barrett Hand Tools</label>
9           <icon type="theme">folder</icon>
10        </group>
11        <label>Control Interface</label>
12        <icon type="theme">applications-utilities</icon>
13        <statustip>User interface to control BHand.</statustip>
14      </qtgui>
15    </class>
16 </library>
```

Finally we can start writing the plugin for RQT in Python. We have follow the tutorial of ROS wiki that show how to write a plugin for RQT in Python to integrate a custom user interface.

4.2.3 Description of the program of the GUI

We have created our python file with the template that we have found in the ROS wiki. The first thing that we have to do is import all the libraries that we will need.

bhand_control_module.py

```
1  import os
2  import sys
3  import rospy
4  import rospkg
5  import time
6
7  from qt_gui.plugin import Plugin
8  from python_qt_binding import loadUi
9  from python_qt_binding.QtGui import QWidget, QPixmap
10 from PyQt4 import QtCore, QtGui
11 from BHand_Control.msg import BarrettState
12 from BHand_Control.srv import CommandBH
```

We have to import also the type of messages *BarrettState* and the type of the service *CommandBH*, to use them in our program when we will process the message that we will receive and to send the command to the server program.

BarrettState.msg

```
1  int32 S
2  int32 F1
3  int32 F2
4  int32 F3
```

CommandBH.srv

```
1  string command
2  ---
3  uint32 error
```

The objective of the file is defining the plugin that makes up our GUI. We will describe the main function of this plugin.

bhand_control_module.py

```
22  def __init__(self, context):
23    super(BHandControlPlugin, self).__init__(context)
24    # Give QObjects reasonable names
25    self.setObjectName('BHandControlPlugin')
[…]
36    # Create QWidget
37    self._widget = QWidget()
38    # Get path to UI file which should be in the "resource"
   folder of this package
39    ui_file = os.path.join(rospkg.RosPack().get_path('rqt_bhand_c
   ontrol'), 'resource', 'interface.ui')
40    # Extend the widget with all attributes and children from UI
   file
41    loadUi(ui_file, self._widget)
42    # Give QObjects reasonable names
43    self._widget.setObjectName('MyPluginUi')
[…]
116   if context.serial_number() > 1:
117     self._widget.setWindowTitle(self._widget.windowTitle()    +
    ('( %d)' % context.serial_number()))
118   # Add widget to the user interface
119   context.add_widget(self._widget)
```

We use the function `super()` to call the method `__init__()` that expects no arguments at all. In line 25, we give a name to the plugin. The next step is to create the widget that we will joint in the plugin. We will load the widget that we have previously created in QT Designer. The first thing that we have to do is extend the widget with all attributes and children from UI file (lines 39 & 41). The last step is to add the widget to the user interface (line 119).

Now that we have the plugin created and all the widgets loaded, we will create the main functions of the GUI.

We have to connect this program with the server to send the commands that we will generate with the GUI.

bhand_control_module.py

```
[…]
121   try:
122     self._service_bhand_actions=rospy.ServiceProxy('CommandBH',
    CommandBH)
123   except ValueError, e:
124     rospy.logerr('BHandGUI: Error connecting service (&s)' %e)
[…]
300 def _send_bhand_action(self, action):
301   try:
302     self._service_bhand_actions(action)
303   except ValueError, e:
304     rospy.logerr('BHandGUI::send_bhand_action: (%s)' %e)
305   except rospy.ServiceException, e:
306     rospy.logerr('BHandGUI::send_bhand_action: (%s)' %e)
[…]
```

In the `__init__()` function (line 121 to 124), we have created a handle for calling the service *CommandBH*. We can use this handle just like a normal function and call it. Is what we do in the function `_send_bhand_action(action)` that we have create. This is the function that we will use to send commands to the hand. For example: `self._send_bhand_action('1C')`, that will close the finger 1.

To get the positions of the motors to display them in the GUI, we have to subscribe to the topic *Barrett_State*. As we have explained before the obtaining of the positions began when we click on the button below the screen.

bhand_control_module.py

```
[…]
42  self._init_value_spread = 0
43  self._init_value_finger1 = 0
44  self._init_value_finger2 = 0
45  self._init_value_finger3 = 0
[…]
85    self._widget.pushButton_get_positions.clicked.connect(self._s
   ubscribeToTopic)
[…]
278 def _subscribeToTopic(self):
279   rospy.Subscriber("Barrett_State",BarrettState,self._callback)
280   rospy.loginfo('Subscribe to topic <Barrett_State>.')
281
282 def _callback(self, data):
283   if data.S != self._init_value_spread:
284     #rospy.loginfo('Position of the spread: %s', data.S)
285     self._init_value_spread = data.S
286     self._widget.lcdNumber_pos_spread.display(data.S)
287   if data.F1 != self._init_value_finger1:
288     #rospy.loginfo('Position of the finger1: %s', data.F1)
289     self._init_value_finger1 = data.F1
290     self._widget.lcdNumber_pos_finger1.display(data.F1)
291   if data.F2 != self._init_value_finger2:
292     #rospy.loginfo('Position of the finger2: %s', data.F2)
293     self._init_value_finger2 = data.F2
294     self._widget.lcdNumber_pos_finger2.display(data.F2)
295   if data.F3 != self._init_value_finger3:
296     #rospy.loginfo('Position of the finger3: %s', data.F3)
297     self._init_value_finger3 = data.F3
298     self._widget.lcdNumber_pos_finger3.display(data.F3)
[…]
```

With the command of the line 85, we have fixed that when we click in the button "Get Positions" we execute the function `_subscribeToTopic()`. The function `_subscribeToTopic()` is the responsible of make the subscription to the topic, and it calls the function `_callback()` that is the responsible of change the value of the LCDs with the new value of the motors. To avoid saturating the program we have scheduled the LCD update only when the value changes. Therefore we had to create new variables (`self._init_value_spread`, `self._init_value_finger1`,…) that we have to update when the value change.

We have also created the function that takes the value that we have choice with the sliders and it sends it with the function that we have previously create.

bhand_control_module.py

```
[…]
46    self.fixed_fingers = 0
[…]
77    self._widget.checkBox_all_fingers.stateChanged.connect(self._
    fingers_check_box_pressed)
[…]
83    self._widget.verticalSlider_spread.sliderReleased.connect(sel
    f._handle_spreadSlider)
84    self._widget.verticalSlider_finger1.sliderReleased.connect(se
    lf._handle_finger1Slider)
85    self._widget.verticalSlider_finger2.sliderReleased.connect(se
    lf._handle_finger2Slider)
86    self._widget.verticalSlider_finger3.sliderReleased.connect(se
    lf._handle_finger3Slider)
[…]
236   def _handle_spreadSlider(self):
237     value = self._widget.verticalSlider_spread.value()
238     self._send_bhand_action('SM %i' % value)
239     #rospy.loginfo('spread value = <%i>' % value)
240
241   def _handle_finger1Slider(self):
242     value = self._widget.verticalSlider_finger1.value()
243     if self.fixed_fingers == 1:
244       self._widget.verticalSlider_finger2.setValue(value)
245       self._widget.verticalSlider_finger3.setValue(value)
246       #rospy.loginfo('finger 1 value =  <%i>.' % value)
247       #rospy.loginfo('finger 2 value =  <%i>.' % value)
248       #rospy.loginfo('finger 3 value =  <%i>.' % value)
249       self._send_bhand_action('GM %i' % value)
250     else:
251       #rospy.loginfo('finger 1 value =  <%i>.' % value)
252       self._send_bhand_action('1M %i' % value)
253
254   def _handle_finger2Slider(self):
255     value = self._widget.verticalSlider_finger2.value()
256     if self.fixed_fingers == 1:
257       self._widget.verticalSlider_finger1.setValue(value)
258       self._widget.verticalSlider_finger3.setValue(value)
259       #rospy.loginfo('finger 1 value =  <%i>.' % value)
260       #rospy.loginfo('finger 2 value =  <%i>.' % value)
261       #rospy.loginfo('finger 3 value =  <%i>.' % value)
262       self._send_bhand_action('GM %i' % value)
263     else:
264       #rospy.loginfo('finger 2 value =  <%i>.' % value)
265       self._send_bhand_action('2M %i' % value)
266
267   def _handle_finger3Slider(self):
268     value = self._widget.verticalSlider_finger3.value()
269     if self.fixed_fingers == 1:
270       self._widget.verticalSlider_finger1.setValue(value)
271       self._widget.verticalSlider_finger2.setValue(value)
```

```
272       #rospy.loginfo('finger 1 value =  <%i>.' % value)
273       #rospy.loginfo('finger 2 value =  <%i>.' % value)
274       #rospy.loginfo('finger 3 value =  <%i>.' % value)
275       self._send_bhand_action('GM %i' % value)
276     else:
277       #rospy.loginfo('finger 3 value =  <%i>.' % value)
278       self._send_bhand_action('3M %i' % value)
279
280   def _fingers_check_box_pressed(self, state):
281     '''
282      Handler call when clicking checkbox to control all fingers
283     '''
284     if state == 0:
285       self.fixed_fingers = 0
286     elif state == 2:
287       self.fixed_fingers = 1
[…]
```

The function `_fingers_check_box_pressed()` change the value of the variable `self.fixed_fingers` depending on whether the check box is marked. It only called when the state of the check box change (line 77). With the lines 83 to 86, we have fixed that when the value of the slider change and it is released we execute the function that correspond. This functions take the value that we have choose with the slider and according of the value of the variable `self.fixed_fingers` it sends the command for the three fingers or only for this finger. If we are moving the three fingers at the same time we have also to change the value of the other slider to the same value.

The next function of our graphic interface it to move the fingers by steps with a variable value of the steps, we change this value with a spin box.

bhand_control_module.py

```
[…]
63    QtCore.QObject.connect(self._widget.spinBox_step_spread,
    QtCore.SIGNAL(_fromUtf8("valueChanged(int)")),
    self._widget.verticalSlider_spread.setSingleStep)
64    QtCore.QObject.connect(self._widget.spinBox_step_fingers,
    QtCore.SIGNAL(_fromUtf8("valueChanged(int)")),
    self._widget.verticalSlider_finger1.setSingleStep)
65    QtCore.QObject.connect(self._widget.spinBox_step_fingers,
    QtCore.SIGNAL(_fromUtf8("valueChanged(int)")),
    self._widget.verticalSlider_finger2.setSingleStep)
66    QtCore.QObject.connect(self._widget.spinBox_step_fingers,
    QtCore.SIGNAL(_fromUtf8("valueChanged(int)")),
    self._widget.verticalSlider_finger3.setSingleStep)
67    QtCore.QObject.connect(self._widget.spinBox_step_spread,
    QtCore.SIGNAL(_fromUtf8("valueChanged(int)")),
    self._widget.verticalSlider_spread.setPageStep)
68    QtCore.QObject.connect(self._widget.spinBox_step_fingers,
    QtCore.SIGNAL(_fromUtf8("valueChanged(int)")),
    self._widget.verticalSlider_finger1.setPageStep)
```

```
69    QtCore.QObject.connect(self._widget.spinBox_step_fingers,
    QtCore.SIGNAL(_fromUtf8("valueChanged(int)")),
    self._widget.verticalSlider_finger2.setPageStep)
70    QtCore.QObject.connect(self._widget.spinBox_step_fingers,
    QtCore.SIGNAL(_fromUtf8("valueChanged(int)")),
    self._widget.verticalSlider_finger3.setPageStep)
[…]
73    self._widget.spinBox_step_spread.setValue(20)
74    self._widget.spinBox_step_fingers.setValue(200)
[…]
87    self._widget.pushButton_spread_m.clicked.connect(self._handle
    _spreadSlider)
88    self._widget.pushButton_spread_p.clicked.connect(self._handle
    _spreadSlider)
89    self._widget.pushButton_finger1_m.clicked.connect(self._handl
    e_finger1Slider)
90    self._widget.pushButton_finger1_p.clicked.connect(self._handl
    e_finger1Slider)
91    self._widget.pushButton_finger2_m.clicked.connect(self._handl
    e_finger2Slider)
92    self._widget.pushButton_finger2_p.clicked.connect(self._handl
    e_finger2Slider)
93    self._widget.pushButton_finger3_m.clicked.connect(self._handl
    e_finger3Slider)
94    self._widget.pushButton_finger3_p.clicked.connect(self._handl
    e_finger3Slider)
[…]
```

With the lines 63 to 70, we give the value of the spin box to the simple step and page step of the sliders. The default values of the spinboxes are 20 in the case of the spread and 200 in the case of the fingers. They have also the values bounded, in the case of the spread the maximum value that we can give to the steps is 200, and in the case of the fingers the maximum value is 2000. In QT Designer, we have linked the buttons + and – to the sliders, so when we click in one of the buttons the slider will change his value of the step that we have choosen. The only thing left to do is send the command with the new value of the slider. So we had to add the lines 87 to 94 that make that every time that we click in one of this buttons the new position will be send to the hand.

bhand_control_module.py

```
[…]
80    self._widget.pushButton_hand_init.clicked.connect(self._handi
    nit)
[…]
100   self._widget.pushButton_stop.clicked.connect(self._handstop)
[…]
119   pixmap_red_file=os.path.join(rp.get_path('rqt_bhand_control'),
    'resource','red.png')
120   pixmap_green_file=os.path.join(rp.get_path('rqt_bhand_control'
    ),'resource','green.png')
121   self._pixmap_red = QPixmap(pixmap_red_file)
122   self._pixmap_green = QPixmap(pixmap_green_file)
123   self._widget.label_state_spread.setPixmap(self._pixmap_green)
```

```
 124   self._widget.label_state_finger1.setPixmap(self._pixmap_green)
 125   self._widget.label_state_finger2.setPixmap(self._pixmap_green)
 126   self._widget.label_state_finger3.setPixmap(self._pixmap_green)
 […]
 139 def _handinit(self):
 140   # initialize all values to zero and send this positions to the
     hand
 141   self._send_bhand_action('HI')
 142   self._widget.verticalSlider_spread.setValue(0)
 143   self._widget.verticalSlider_finger1.setValue(0)
 144   self._widget.verticalSlider_finger2.setValue(0)
 145   self._widget.verticalSlider_finger3.setValue(0)
 146   self._widget.spinBox_step_spread.setValue(20)
 147   self._widget.spinBox_step_fingers.setValue(200)
 148   self._widget.label_state_spread.setPixmap(self._pixmap_green)
 149   self._widget.label_state_finger1.setPixmap(self._pixmap_green)
 150   self._widget.label_state_finger2.setPixmap(self._pixmap_green)
 151   self._widget.label_state_finger3.setPixmap(self._pixmap_green)
 152
 153 def _handstop(self):
 154   self._send_bhand_action('GST')
 155   self._widget.label_state_spread.setPixmap(self._pixmap_red)
 156   self._widget.label_state_finger1.setPixmap(self._pixmap_red)
 157   self._widget.label_state_finger2.setPixmap(self._pixmap_red)
 158   self._widget.label_state_finger3.setPixmap(self._pixmap_red)
 […]
```

We have created with GIMP (a free and open-source raster graphics editor) two images that simulate the lights of a semaphore. These images will show the state of the motors if they are initialized the light will be green. We join these images to our interface and transform them in a QPixMap class. We change the pixmap of the labels in function of the state of the motors. To make the initialization of the hand, we have created the function `_handinit()` that send the initializing command to the hand, it puts all the sliders to zero, reset the values of the steps of the sliders to the default values and change the color of the pixmaps to green. The function `_handstop()` is the responsible of disconnecting the motors and it change the color of the *pixmaps* to red to advertise the user of the state of the motors.

The last thing that we have done is the functions of the simple control of Barrett Hand, we have fixed every button with his function.

bhand_control_module.py

```
 […]
 101 # Open the hand
 102 self._widget.pushButton_open.clicked.connect(self._hand_open)
 103 # Close the hand
 104 self._widget.pushButton_close.clicked.connect(self._hand_close)
 105 # Joint the 3 fingers to make a power grasp
 106 self._widget.pushButton_pinch.clicked.connect(self._hand_pinch)
 107 # Separe the 3 fingers to make a precision grasp
 108 self._widget.pushButton_spread.clicked.connect(self._hand_sprea
     d)
```

```
109 # Make a hand shake
110 self._widget.pushButton_shake_hand.clicked.connect(self._hand_s
    hake)
111 # Make a demostration of all the movements that the hand can do
112 self._widget.pushButton_demo.clicked.connect(self._hand_demo)
113 # Close the hand of 2000 motor steps
114 self._widget.pushButton_plus.clicked.connect(self._hand_increme
    ntal_close)
115 # Open the hand of 2000 motor steps
116 self._widget.pushButton_minus.clicked.connect(self._hand_increm
    ental_open)
[…]
160 def _hand_open(self):
161   self._send_bhand_action('GO')
162   self._widget.verticalSlider_finger1.setValue(0)
163   self._widget.verticalSlider_finger2.setValue(0)
164   self._widget.verticalSlider_finger3.setValue(0)
165
166 def _hand_close(self):
167   self._send_bhand_action('GC')
168   self._widget.verticalSlider_finger1.setValue(18500)
169   self._widget.verticalSlider_finger2.setValue(18700)
170   self._widget.verticalSlider_finger3.setValue(18300)
171
172 def _hand_pinch(self):
173   self._send_bhand_action('SC')
174   self._widget.verticalSlider_spread.setValue(3150)
175
176 def _hand_spread(self):
177   self._send_bhand_action('SO')
178   self._widget.verticalSlider_spread.setValue(0)
179
180 def _hand_shake(self):
181   self._send_bhand_action('SC')
182   time.sleep(2)
183   self._send_bhand_action('GM 10000')
184   time.sleep(1)
185   self._send_bhand_action('GM 15000')
186   self._send_bhand_action('13M 16000')
187   self._send_bhand_action('1M 17000')
188
189 def _hand_demo(self):
190   self._send_bhand_action('GO')
191   self._send_bhand_action('SC')
192   self._send_bhand_action('GIC 500')
193   self._send_bhand_action('1IC 2000')
194   self._send_bhand_action('3IC 1000')
195   self._send_bhand_action('1IC 2000')
196   self._send_bhand_action('3IC 1000')
197   self._send_bhand_action('GIC 1000')
198   self._send_bhand_action('GIC 1000')
199   self._send_bhand_action('GIC 1000')
200   self._send_bhand_action('GIC 1000')
201   self._send_bhand_action('GIC 1000')
```

```
202   self._send_bhand_action('GIC 1000')
203   self._send_bhand_action('3IC 1000')
204   self._send_bhand_action('2IC 2000')
205   self._send_bhand_action('1IO 2000')
206   self._send_bhand_action('3IO 1000')
207   self._send_bhand_action('1IO 2000')
208   self._send_bhand_action('3IO 1000')
209   self._send_bhand_action('GIO 1000')
210   self._send_bhand_action('GIO 1000')
211   self._send_bhand_action('GIO 1000')
212   self._send_bhand_action('GIO 1000')
213   self._send_bhand_action('GIO 1000')
214   self._send_bhand_action('GIO 1000')
215   self._send_bhand_action('3IO 1000')
216   self._send_bhand_action('2IO 2000')
217
218 def _hand_incremental_close(self):
219   self._send_bhand_action('GIC 2000')
220   value_f1 = self._widget.verticalSlider_finger1.value()
221   self._widget.verticalSlider_finger1.setValue(value_f1+2000)
222   value_f2 = self._widget.verticalSlider_finger2.value()
223   self._widget.verticalSlider_finger2.setValue(value_f2+2000)
224   value_f3 = self._widget.verticalSlider_finger3.value()
225   self._widget.verticalSlider_finger3.setValue(value_f3+2000)
226
227 def _hand_incremental_open(self):
228   self._send_bhand_action('GIO 2000')
229   value_f1 = self._widget.verticalSlider_finger1.value()
230   self._widget.verticalSlider_finger1.setValue(value_f1-2000)
231   value_f2 = self._widget.verticalSlider_finger2.value()
232   self._widget.verticalSlider_finger2.setValue(value_f2-2000)
233   value_f3 = self._widget.verticalSlider_finger3.value()
234   self._widget.verticalSlider_finger3.setValue(value_f3-2000)
[…]
```

5 NetBox Sensor

5.1 What it is

The Network Force/Torque(Net F/T), is multi-axis system, capable of measuring forces and torques simultaneously. The Net F/T system sends the information via EtherNet/IP, CAN Bus, Ethernet, and is compatible with DeviceNet. The Net F/T has its own web page, where it can set up and monitor all the parameters easily.

5.2 Net F/T features:

The Net F/T box has many features that can be set; it can change the units, the frequency ratio, and its IP direction. Also it can use different source of power and way of sending information, as we will see in detail:

Multiple calibrations:

The Net F/T can hold up to sixteen different calibrations, each one with its own sensibility range. These different calibrations are useful for different purposes, using larger or finer calibrations depending on the adjustments required.

Multiple configurations:

Net F/T sensor can have sixteen different configurations, each one configuration linked to a calibration setting, making the Net F/T more versatile.

Force and torque values:

The Net F/T outputs scaled numbers or counts, representing the loading supported by the sensor in the different axis. The value of the counts per force and torque units is defined in the calibration, also it can display all the values in different units: newtons, pounds, newtons-meter, pounds-inches,...

System status code

The net f/T sensor has a code that indicates the health of the transducer and net-box, indicating id it is in good condition (healthy) or it it needs to be checked.

Thresholding:

The Net F/T can monitor the force and torque values of each axis and set an output code if a received value crosses a defined threshold. The Net F/T can hold up to sixteen thresholds, each threshold can be enabled individually or as a group.

Tool transformation:

The Net F/T can measure forces from a reference different than the original one set in factory, this change is called a tool transformation, which can be defined in each configurations

Multiple interfaces:

As we have said before, the Net f/T can communicate using many different ways, like EtherNet/IP, CAN bus, Ethernet and even DeviceNet, each one can be configured in the Net F/T's web page.

Power supply:

This system can use power trough PoE(Power over Ethernet) or from a DC power source with a voltage between 11V and 24V.

These features can be managed in the sensors web page, but before that, we will see the tools that form the sensor system, how it receives and sends the information to the computer.

Figure 8: NetBox Sensor

This system consists of three main parts: the Net F/T transducer, the net box and the transducer cable:

- Net F/T transducer: It is a cylindric box with the system able to transform torque and forces into electrical signals so it can be sent through the cable to the Net Box. In this model, the signals are digital.
- Net Box: Is an aluminium case that contains the power supplies and network basic interfaces for setting the parameters. It contains 4 connections: Transducer connection, which receives the digital signals from the transducer; power connector, the threshold relay connector and an ethernet port which sends the information to the computer
- Transducer cable: Is a replaceable cable for digital transmission

5.3 Connecting to the NetBox

The first step for connecting to the sensor is establishing the IP of the NetBox and our computer. If the Ip of the NetBox is unknown, we should change its Ip to the default one. In the NetBox case, there are different switches available, if the switch 9 is turned to ON position, it will reestablish the NetBox's IP to the default one which is 192.168.1.1.

Once the Ip of the NetBox is known, we can connect to it. For this, we must first connect the NetBox to the power supply, and connect the Ethernet cable between the box and the computer. After everything is connected, we just have to write the sensor's Ip in our web browser, and it will open the parameters and settings interface of the NetBox.

First we will have the welcome page, which gives us a quick overview of the sensors functions; on the left we have a list of tabs with the different options settings.

The snapshot page shows us information about the maximum and minimum peaks and the thresholding conditions, this information is static and it only updates when the page is refreshed. In this page we can also see the status of the sensor, if it's healthy or on the other side it needs to be repaired. If it is not healthy, it will give less precise values.

The demo page, allows us to download an example application in java that displays all the force and torque values. This application show the values, and uses a progress bar too, also, it indicates us the units used in the sensor.

In the settings page, we can start to set the properties that the sensor will use, there are some previous configurations saved, that we could use, choosing in the active configuration list. We can specify too the Low-Pass filter cutoff, that allows us to attenuate signals with frequencies over the limit specified. We can enable the peak monitoring, that updates the top value of each parameter to the highest registered.

In the thresholding page, we can set the conditions in the different thresholds, and choose the counts, that are the units we have previously settled.

The configuration page allows us to choose the calibration to use in each configuration, each transducer must have at least one calibration, we also can set the force and torque units and the counts per force and torque.

Communication settings

In the section of the communication settings, we will set the ip that the sensor will use to communicate to the computer, notice, that be able to save the ip we set, the switch 9 must be off, if it is on, it will keep its ip in 192.168.1.1. There is too the option to specify the ratio of output information and the buffer size, that controls the queue size of traffic data.

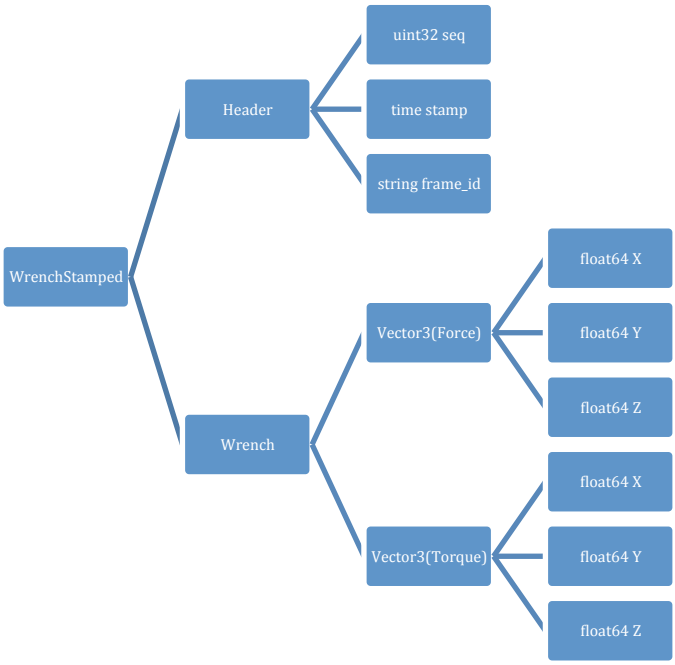
5.4 Receiving data in real time

For receiving the data from the Net F/T sensor, we have created two ROS packages. The first package is the responsible of receiving the data from the Net F/T sensor, and organizes it, creating a topic and a message to send trough the topic. The second package will subscribe to this topic and display all the values with in an interface we have created.

The message sent through the topic is a Geometry message, called Wrench Stamped message. This kind of messages consists of two different types of messages, a Header and a Wrench message.

First of all there is a Header, which is a standard message; it consists of a uint32 sequence, a timestamp and a string frame_id. The uint32 variable displays the number of messages displayed since the subscriber is activated. The second value, the time stamp, displays the time at which value is received, counting from zero at the activation of the subscriber.

On the other hand is the Wrench message, which is divided in two groups. Both groups are vector3 type of messages, this means, and each one consists of three values, one per axis. The two groups are force and torque, each one formed by three float64 value, each one for the different axis (x, y or z) of the correspondent force or torque.



5.5 Socket package

This is the package that connects gets the measures from the Ethernet port and creates a topic with the information we are going to use later on the interface. Our tutor gave us this package, and it is a package that was made in the institute E. Piaggio that we have found on Github.

This package is called "force_torque_sensor", and it is the first package to open when we want to receive the info from the Netbox, as this is the package that will take the data from the sensor to the computer. It is a ROS package that consists on two C++ executable, one main executable that will create the message and topic, and a second one that is called by the first one and is the responsible of creating the socket and send all the data to the first C++ executable. In order to execute these two C++ executable, we first will have to run roscore in one terminal, so then, the executables can create the topic with all the data. Once the roscore is running, the netbox reader can be executed, as it is a package, it has to be called like:

```
$ rosrun force_torque_sensor ftsensor_CLIENT
```

Once the package is running, if everything is well connected, it will start publishing the force and torque values in the terminal. As we said before, there are two executable: one main C++ that calls a second one. So we will see the code in the order it does execute.

The main C++ is the *FT_sensor_node.cpp*, this file starts defining the classes and creating the nodes and the topic where the measurements will be published, it also sends its first order to the second C++ executable, sending it the order of creating the socket. The *FT_sensor_node.cpp* calls the *FTSensor.cpp*, which creates the socket when initializing.

FTSensor.cpp

```
42 // create the socket
43 socketHandle_ = socket(AF_INET, SOCK_DGRAM, 0);
44 if (socketHandle_ == -1) {
45   printf("failed to ini sensor");
46 }
```

```
47 // parse the configuration
48  xmlDoc         *doc = NULL;
49 xmlNode        *root_element = NULL;
50 char     Filename[100];
51 sprintf(Filename, "http://%s/netftapi2.xml?index=15", ip_);
52 doc = xmlReadFile(Filename, NULL, 0);
53 if (doc == NULL) {
54   printf("error: could not parse file %s\n", Filename);
55  }
56 else {
57   root_element = xmlDocGetRootElement(doc);
58   getElementNames(root_element);
59   xmlFreeDoc(doc);
60  }
61  xmlCleanupParser();
62  // set the socket parameters
63 hePtr_ = gethostbyname(ip_);
64 memcpy(&addr_.sin_addr,      hePtr_->h_addr_list[0],      hePtr_-
   >h_length);
65 addr_.sin_family = AF_INET;
66 addr_.sin_port = htons(PORT);
67  // connect
68 int  err  =  connect(  socketHandle_,  (struct  sockaddr  *)&addr_,
   sizeof(addr_) );
69 if (err == -1) {
70   printf("Error to conect with sensor");
71   return;
72 }
73 }
```

Here the sensor is created and it connects to the Ethernet port, it uses the parameters that are defined in the *FT_sensor_node.cpp* such as the ip direction or the NetBox sensor. Once the socket is created and connected, it will go back to *FT_sensor_node.cpp* and go execute the next function, the publish measurements thread, which sends another order to the second C++ file and receives all the measures so it can publish them.

FT_sensor_node.cpp

```
74 void FTSensorPublisher::publishMeasurements()
75 {
76  geometry_msgs::WrenchStamped ftreadings;
77 float measurements[6];
78 ftsensor_->getMeasurements(measurements);
79 ftreadings.wrench.force.x = measurements[0];
80 ftreadings.wrench.force.y = measurements[1];
81 ftreadings.wrench.force.z = measurements[2];
82 ftreadings.wrench.torque.x = measurements[3];
83 ftreadings.wrench.torque.y = measurements[4];
84 ftreadings.wrench.torque.z = measurements[5];
85 ftreadings.header.stamp = ros::Time::now();
86 ftreadings.header.frame_id = frame_ft_;
87 pub_sensor_readings_.publish(ftreadings);
```

```
88 ROS_INFO("Measured Force: %f %f %f Measured Torque: %f %f %f",
   measurements[0],        measurements[1],        measurements[2],
   measurements[3], measurements[4], measurements[5]);
89 }
```

In this function, the WrenchStamped message called ftreadings is created. In the line 78 it calls a function called getMeasurements, this function is in the second C++ executable. Once the measurements from the FTSensor.cpp are received, it just copies the values into the message that will be published in the topic. It also does a ROS_INFO print, so we can visualize all the values in the terminal. Now we will proceed to see what happens in the FTSensor.cpp when the getMeasurements function is called in FTSensor.cpp. As the socket is already working, it just has to get the values from it.

FTSensor.cpp

```
108 void FTSensor::getMeasurements(float measurements[6])
109 {
110   // set the standard request
111   *(short*)&request_[0] = htons(0x1234);
112   *(short*)&request_[2] = htons(COMMAND);
113   *(unsigned int*)&request_[4] = htonl(NUM_SAMPLES);
114   send( socketHandle_, request_, 8, 0 );
115   recv( socketHandle_, response_, 36, 0 );
116   resp_.rdt_sequence = ntohl(*(unsigned int*)&response_[0]);
117   resp_.ft_sequence = ntohl(*(unsigned int*)&response_[4]);
118   resp_.status = ntohl(*(unsigned int*)&response_[8]);
119   int i;
120   for( i = 0; i < 6; i++ ) {
121     resp_.FTData[i] = ntohl(*(unsigned int*)&response_[12 + i *
    4]);
122     if (i<3)
123       measurements[i]=(float)           resp_.FTData[i]/(float)
    counts_per_force_;
124     else
125       measurements[i]=(float)           resp_.FTData[i]/(float)
    counts_per_torque_;
126   }
127   return;
128 }
```

First it sets the mode the data will be used, and them it sends a request to the socket which is already connected and gets all the measures form the NetBox. When the response is received with the values, they are saved in the measurements array which is the message send to the FT_sensor_node.cpp.

If we want to see how the info is sent trough the topic, we can see it using a terminal. There we can see the list of topics that are running by writing “rostopic list”, and we will see all the topics that are currently being executed.

We can see the *sensor_readings* topic is running, so now we can see the information that is being sent. Using the command `rostopic echo sensor_readings`, we will see the message of the topic displayed in the terminal.

5.6 Interface package

Before getting into the interface, first we have to understand another tool used. ROS has a tool called rqt, rqt is a software framework of ROS that implements the various GUI tools in the form of plugins, so all the existing GUI tools can be run as dock-able windows in rqt, simplifying all the work with the interfaces.

The interface package consists of two main parts, which reside in two different folders, the resource folder and the src folder. In the src folder we have the code that the interface will be using, a python code where the subscribing and displaying functions are hold. The other folder, the resource folder, is where the interface resides, along with other images that we are using with her.

The interface was made with QTdesigner, a useful program that simplifies the creation of interfaces in a very intuitive and graphical way. This program allows us to create the interfaces placing different widgets, the widgets are different tools with, multiple functions that make easier the interaction between user and computer so the user can see values or info sent from the program he is using, or the other way around, send data to the program, and so it can work and process it.

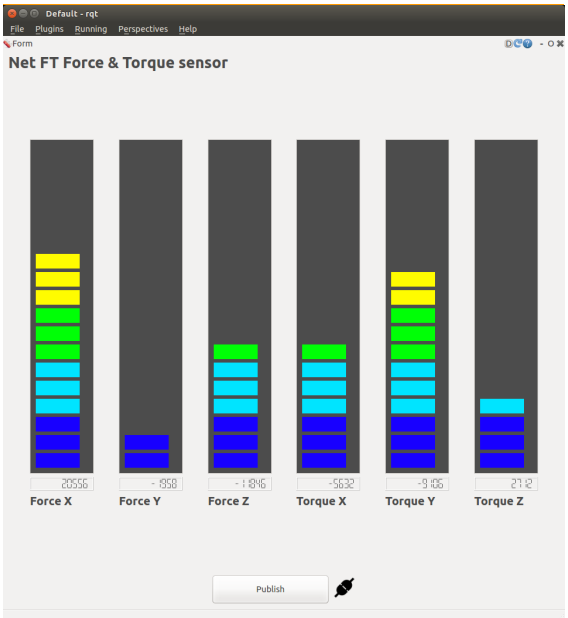


Figure 9: Interface of the sensor with progress bars and displays

As we can see in the photo, there are six vertical bars composed of eighteen colored rectangles, these bars represent the absolute value of the forces and torques, so we can see the force visually. Under each bar, there is a LCD display, which shows the value of the force measured by the sensor. Finally at the bottom of the interface, there is a button which says “Subscribe” this button will activate the subscribe function, so then the interface can show the received values.

Each widget of the interface is connected to a function in the python code, so when we create a python file, we have to declare the links between each button and its function.

The most important function is the subscribe function, which is the tool we use to get the values of force and torque from the topic.

wrist_module.py

```
108 self._wristValuesSignal = WristValuesSignal()
109 self._wristValuesSignal.signal.connect(self._wristValuesSignalH
    andler)
110 self._widget.subscribe.clicked.connect(self._subscribe)
```

Here we are defining the signals that our interface we will use. In the first line, we create our own signal, called *wristValueSignal*, this signal will be sent whenever we call during the execution, and will activate the defined function. This signal can also send information. In this case, the *wristValueSignal* is connected to the *wristValueSignalHandler* function. The last line connects a button, with a function. Some widgets have their own signals, so we do not have to define them if we want to use them. For example, the buttons can send a signal, when they are pressed or released. In our interface, we use the press signal, which will activate the subscribe function.

wrist_module.py

```
216 def _subscribe(self):
217  #Subscriber to the force sensor
218  self._topicListener = rospy.Subscriber('sensor_readings',
    WrenchStamped,self._callback)
219  rospy.loginfo('Subscribe to topic sensor_readings')
```

The subscribe function, connects the to the sensor readings topic, where all the force and torque values are being published in a defined *WrenchStamped* message. All this values are sent to second function, called callback.

wrist_module.py

```
120 def _callback(self, ftreadings):
121   #defining variables
122   self.force_x = int(ftreadings.wrench.force.x)
123   self.force_y = int(ftreadings.wrench.force.y)
124   self.force_z = int(ftreadings.wrench.force.z)
125   self.torque_x = int(ftreadings.wrench.torque.x)
126   self.torque_y = int(ftreadings.wrench.torque.y)
127   self.torque_z = int(ftreadings.wrench.torque.z)
128   self._plug()
129   #send the values to the display function
130   self._display(self.force_x,  self.force_y,  self.force_z,
    self.torque_x, self.torque_y, self.torque_z)
131   #create an array and send it to the bars function
132   wrist_list=[self.force_x,  self.force_y,  self.force_z,
    self.torque_x, self.torque_y, self.torque_z]
133   self._wristValuesSignal.signal.emit(wrist_list)
134   self._wristValuesSignalHandler(wrist_list)
135   #publisher in terminal
136   rospy.loginfo("Force : x=[%f] y=[%f] z=[%f] Torque x=[%f]
    y=[%f]  z=[%f]",self.force_x,  self.force_y,  self.force_z,
    self.torque_x, self.torque_y, self.torque_z)
```

Here we can see the callback function, which is the function that the subscribe function calls when it connects to the topic. In this function we receive the values from the topic, contained in the *WrenchStamped* message. This message is called *ftreadings*, and the first step to process all its data is defining the message values in our executable. We will define the values as integers and we will get them separately, the force and the torque values. Once the three values are defined, they can be sent to the LCD display we have in the interface, the displays have their own function, so we just have to call the function and send it the values we want to show. In the case of the bars function, it is a bit different, as we have seen before, the *wristValueSignalHandler* is activated by a signal we have previously created. The reason we have to create this signal and we cannot execute it directly as we do with the display function, is due to an error rqt shows when activating it directly. This is because the *wristValueSignalHandler* when using functions that are updating in real time, the program is saturated, and this is why we create an extra signal to call the *wristValueSignalHandler*.
Now we will analyze the two displaying threads, the one that updates the LCD widget screens and the one that updates the progress bar.
The LCD widgets are really simple to update, as they have no trouble updating while they receive the values.
wrist_module.py

```
200 def _display(self,force_x,force_y, force_z, torque_x, torque_y,
    torque_z):
201   #Force LCDs
202   self._widget.lcd_display_0.display(force_x)
203   self._widget.lcd_display_1.display(force_y)
204   self._widget.lcd_display_2.display(force_z)
205   #Torque LCDs
206   self._widget.lcd_display_3.display(torque_x)
207   self._widget.lcd_display_4.display(torque_y)
208   self._widget.lcd_display_5.display(torque_z)
```

As we can see, there is an LCD display for each axis of the force and torque values. These values are defined when the thread is initialized, as they are received from the subscribe function, where the display function is called.
For showing the values in the display is quite simple, as we just have to define the name of the LCD widget we are using, set the display property of the LCD display with the value we want him to show.

The progress bar thread uses a different structure, as it will use labels with colored bars instead of LCD displays, also, this is percentage visualization. Therefore the thread will be a succession of two for loops with different if statements. The for loops will go through all the color blocks of each column, while the if statements will define the color they are. So we will first set a limit and a step size that will define the number of labels activated for each value.
wrist_module.py

```
138 def _wristValuesSignalHandler(self, wrist_list):
139   self._cleaning()
140   self.forcetop=90000
```

```
141   self.stepvalue=int(self.forcetop/17.)
142   for i in range (0,3):
143     force=wrist_list[i]
144     if force<0:
145       =force*-1
146       if force> 0:
147         for j in range (0,18):
148         force_color = 'clab_' + str(i) +'_' + str(j)
149         if j>=0 and j<3
150
    getattr(self._widget,force_color).setPixmap(self._pixmap_blue)
151         elif j>=3 and j<6:
152
    getattr(self._widget,force_color).setPixmap(self._pixmap_baby)
153         elif j>=6 and j<9:
154
    getattr(self._widget,force_color).setPixmap(self._pixmap_green)
155         elif j>=9 and j<12:
156
    getattr(self._widget,force_color).setPixmap(self._pixmap_yellow
    )
157         elif j>=12 and j<15:
158
    getattr(self._widget,force_color).setPixmap(self._pixmap_orange
    )
159         elif j>=15 and j<18:
160
    getattr(self._widget,force_color).setPixmap(self._pixmap_red)
161         if force < self.stepvalue*(j+1):
162           break
```

The first step of this function is the cleaning function; this function goes through all the labels of the interface, setting them in black, so they are uncolored blocks. The blocks that represent the percentage of force measured will be colored in blue, baby blue, green, yellow, orange and red from lower to higher values. The code lines above represent just a small part of the code for the force bars, as it would be too long to fit here. As we have said before, we will first set a top value and a step value, in this case we have set a top value of 90000, as we considered it is high enough to measure all the possible forces. This top value is divided in seventeen, which is the number of blocks minus one, as the last block will be activated only if the top value is exceeded.

The force and torque values are content in an array of six values that our thread receives from the callback thread, the first three values are the forces, the last three are the torque values, so we will do a for loop which takes the first three values.

Now we proceed to go through the values received and compare them to the step values and activate the correspondent colored bars for each one force measured. The for loop with I, is the responsible of getting the values from the array received, we will deal the force and torque values separately so it will do loop for the first three values, and another one for the other three. Once the measure is gotten, we will make sure the value is positive with an if, that can take the negative values and make them positive. Once we got the value, we will start

running through all the labels to be colored. Here it will color the label in position (i,j), the in variable represents the force and its column while the j variable represents the block of the column that is being threaten in that moment.

The succession of if and elif statements will set the color of each label block. The last if condition, compares the measure with its different barrier value, so once the measure is not higher that its next step value, it exit the loop, and start with the next force measured.

On the other side, the called function from each time the bars are started, the cleaning bars thread, uses the same idea, but as it fills all the labels with the same color, it is much simple.

wrist_module.py

```
108 def  _cleaning(self):
109   for i in range(0,6):
110     for j in range(0,18):
111       lab_color = 'clab_' + str(i) +'_' + str(j)
112
    getattr(self._widget,lab_color).setPixmap(self._pixmap_empty)
```

It goes through all the labels leaving them empty. All these colors we have used, are defined in the init class of the program, this images are kept in the same folder as the .ui file of the interface. To load them into the program we just have to follow a simple structure.

```
pixmap_blue_file = os.path.join(rp.get_path('wrist_sensor_interface'),
'resource', 'blue.png')
self._pixmap_blue = Qpixmap(pixmap_blue_file)
```

The first line loads the image on the interface, while the second, and creates a shortcut to execute the image when we want to use it.

6 Takktile Sensors

6.1 Takktile sensors connection

The Takktile sensors hardware consists in three lines of sensors (TakkStrip) and a main interface processor (TakkFast).

6.1.1 TakkFast

The TakkFast is a processor that makes possible to transfer data from I2C to USB at approximately 100Hz to a Linux host with ROS drivers. In our case we will use it to connect the three TakkStrip to it, and send the data to the computer. The TakkFast comes with 5 solder terminals with four connections to each one, the connections are: Voltage, Ground, data and clock. On the other side of the plate is the USB connection.

The TakkFast will be placed in the adapter of the BarrettHand, then, the processor will have an USB cable that will connect directly to he computer. The USB cable will not just transfer the information, it will also is the responsible of the power supply of the sensors.

6.1.2 TakkStrip

The TakkStrip consists of two parts, the first one is a “traffic cop” microchip on the border, the second one is a line of five MEMS barometers protected with a metal case and rubber. The strips are PCB boards with a length of 62 mm and a width of 8,6. The sensors have a length of 8 mm each one.

Only five sensors can be connected to each microcontroller, and up to eight microcontrollers can be connected to the main processor (Arduino or TakkFast). In order of not having trouble confusing the address, each microchip must have its own, that is why each one has a set of three pins, named A0, A1 and A2, a binary code to set the address of each line from 0 to 7, by welding the different pins.

The strips are modular, the microchip can be separated from the sensors' line, and the five sensors can be cut separately. Once they are cut, they have to be wired, according to the instructions. In our case, we have cut the strip in three parts: the microchip, a line of four sensors and a separate sensor that is placed in the fingertip.

The microchip has three groups of pin connections. The first group is the nine pins at the top of the image, these pins are used to connect to the line of sensors, this nine connections are the voltage, ground, data, clock, and addresses of each sensor. The first line of pins are, Ground, second address, fourth address and date, while the second line consists of voltage, addresses first, third and fifth and clock.

The line of sensors has a symmetrical configuration of connections, so it is simple to connect all the sensors to the microchip. In our case, as we have just four sensors in the line, we do not have connected the fifth address pin.

The matrix of pins in the center of the board, are the optional connections, that we can use to connect all the sensors independently, each column refers to an independent sensor, and each sensor has five connections, Clock, data, ground, voltage and select, this last one is the pin that sets the address of the sensor. Each sensor has two lines with these same connections so we have two options or we can even connect two independent sensors in series (except the select pin, that must be independently connected for each one).

In the case of our sensors connections we have a fifth sensor that is connected independently, so it is connected to the fifth column of pins.

Finally at the bottom of the plate, under the microchip, are the I2C connections to the TakkFast processor, the first two pins are the power supply connections, Voltage and Ground, while the last two are Data and Clock pins. These are the pins that are connected to the TakkFast processor.

In the following image, we can see the example of how each sensor is connected to the TakkStrip.

Red – Voltage
Black – Ground
Green – Data
Blue – Clock
Orange and Yellow – Address select

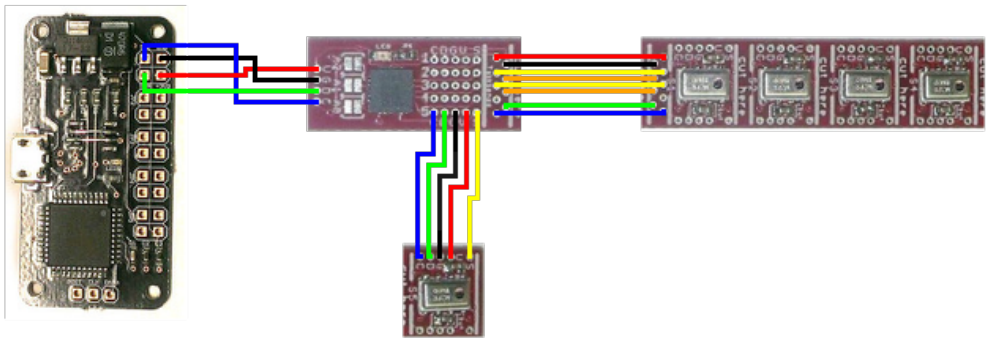


Figure 10: Connections in between the sensors and processor

6.1.3 Wiring

The wires used for all the connections are ribbon parallel wires that were cut in groups of four, five and nine for the different connections between sensors and processors. The connections between the sensors and the microchip are weld with extra thin tin thread, as they will not be removed once they are mounted in the finger and this way they will not have problems of contact in the pins. On the other side, the cable between the microchip and the TakkFast processor, is weld on the side of microchip, while on the other one, there were weld two pairs of pins, so they can be plugged in and out to the TakkFast without damaging the cable.

Finally, when all the sensors are weld and ready to be mounted, we will have to organize the cables, so they do not interfere with the finger's movements or between them. At the front face, where the sensors will be placed, the line of sensors will be placed leaving the connections pins at the lower section, so thee cables do not touch the sensors. At the top of the fingertip, will be placed the fifth sensor, this one will have the cables welded from below the sensor, so the sensor will have to be a bit displaced so it leaves space to the cables.

On the back face of the finger, will be placed the microchip. Here is where all the wires run to, so it must be careful they do not cover each others connections. The microchip is place so the wires from the line of four sensors is at the end of the finger, this cable comes laterally, but is thin enough to not be in danger when the spread closes. Second, there is the independent sensor, whose cable comes from the very top of the finger, but it has to do a C form, this is because it went straight

to the back it would interfere with the VGDC connections that go to the processor, also, they need to come by the side, because of the column of pins that are for the fifth sensor. Finally, it is the connection that goes to the TakkFast processor, this cable leaves directly from the microchip to the processor, and it is long enough to not having tight forces when the fingers close or move around.

6.2 Finger's tactile sensors

The sensors are made in a line of 5 sensors with a processor; they all together measure 62mm and a width of 8,6mm. But the Takktile sensors are modular, so they can be separated in different parts independently, with the help of cables, to connect them. In this case, we will cut the string in three different parts.

6.2.1 Finger design

In order to properly integrate the Takktile sensors in the hand, we have redesigned the fingerprints of the hand. The new model of the fingers is able to hold 5 sensors and the processor of each finger.

The finger is 3D printed in ABS plastic, as the plastic parts of the camera support. The design of this new finger was made using the base model received from Barrett Laboratories, also used in the RViz visualization. The original fingers are 46mm long, made out of aluminum with the point of the finger rounded.

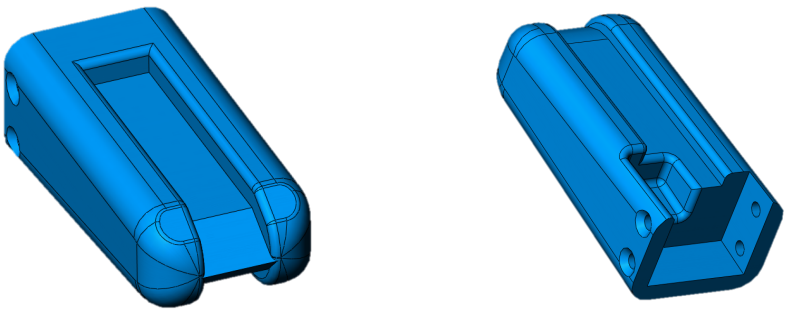


Figure 11: Top and back faces of the finger

This new fingerprint has been enlarged almost 4 mm (49.97mm), so it is long enough to hold the five sensors in a row. The first four sensors will be placed inside a furrow, of 3.5mm deep, therefore the sensors will be protected and will only let stand out the minimum required by the sensors to receive and work without interferences. The furrow is 34mm long, so it has space enough to hold the 22mm long line of sensors.

The fifth sensor will be placed at the tip of the finger, but this sensor will not be placed in the same surface as the rest of sensors, this one will be in a plane displaced an angle of 30 degrees, therefore it will have a better contact position if there is any object while closing the hand. Finally, on the back surface of the finger, there is another furrow 2.5mm deep, where the processor will be held.

6.2.2 Connections

The cables have been placed, in the position that they interfere the less to the hands movements. The cables between the four sensors row and the microchip will be placed on the side of the finger, as they are thin enough to not have

problems once the spread is fully closed. The fifth sensor will have its cables parallel to the furrow, but doing a C form, so it can leave space to the pins of the main processor. These cables have been glued so they cannot move around while the fingers are moving.

Finally, the last cable with the four main connections (Voltage, ground, data and clock) will leave directly from the pins to the main processor, going parallel to the finger's length. In the case of the fingers two and three, will have extra cable length, as they need to move around due to the spread.

6.3 Publisher Program

The company that sells these sensors gives in his web page a program to make the integration in ROS. This program works only with the TakkFast. The TakkFast is an interface that makes possible to transfer data from I2C to USB at high speeds.

6.4 Graphic interface

6.4.1 Design of the graphic interface

The interface that we have created is composed of LCDs that show the value of tactile sensors of the fingertips and change of color in function of the value. We have chosen of use color to improve the intuitive visualization of the pressure. The LCDs are organized as in the BarrettHand. To begin the obtaining of the values you have to click in the button "GET VALUES" in the bottom of the window.

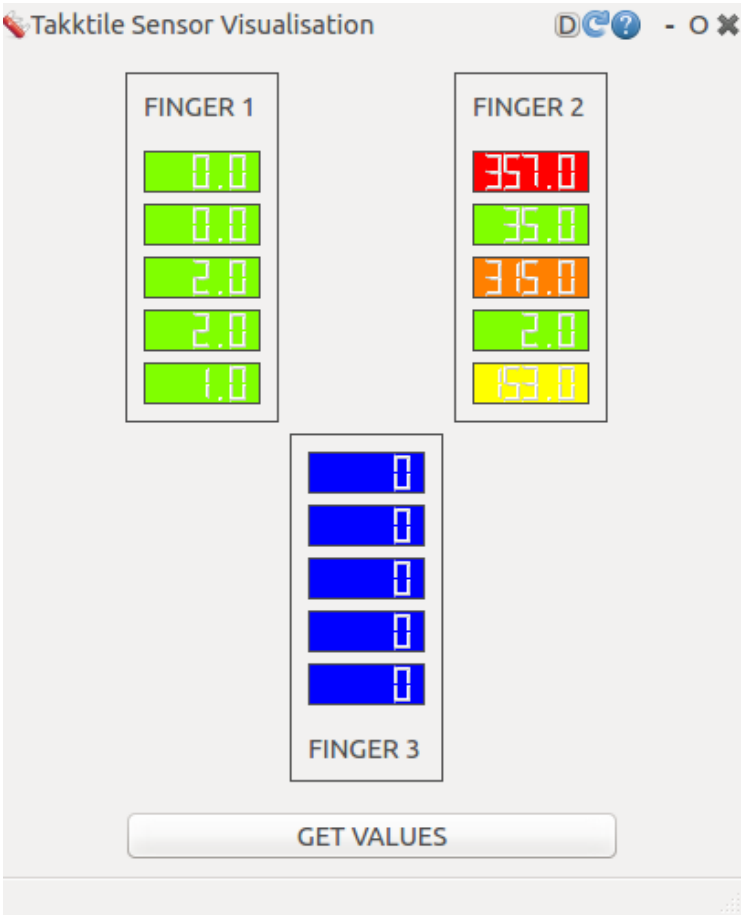


Figure 12: Takktile Sensor Interface

The LCDs can take the following colors in function of the value:

COLOR	VALUE	
Green	0	99
Yellow	100	199
Orange	200	349
Red	350	500
Blue	No initialized	

Table 3: Colors of LCDs in Takktile Sensor interface

6.4.2 Integration of the program in ROS

As we have done with the Interface for the BarrettHand's control. We have integrated our custom interface into ROS GUI framework RQT. We have create an empty package: `$ catkin_create_pkg rqt_se rospy rqt_gui rqt_gui_py`. The next step is declare the plugin in *package.xml.*

package.xml

```
53 :
54 <export>
56   <rqt_gui plugin="${prefix}/plugin.xml"/>
60 </export>
61 :
```

The next step has been the creation of the *plugin.xml* file that contains additional meta information regarding the plugin.

plugin.xml

```
1  <library path="src">
2    <class name="Sensor Takktile Interface" type="rqt_sensor_takkt
   ile.sensor_takktile_module.SensorTakktilePlugin"
   base_class_type="rqt_gui_py::Plugin">
3      <description>
4        Interface to visualize value of Takktile.
5      </description>
6      <qtgui>
7        <group>
8          <label>Barrett Hand Tools</label>
9          <icon type="theme">folder</icon>
10       </group>
11       <label>Sensor Takktile</label>
12       <icon type="theme">applications-utilities</icon>
13       <statustip>Great user interface to visualize Takktile
   Sensor.</statustip>
14     </qtgui>
15   </class>
16 </library>
```

Finally we can start writing the plugin for rqt in Python. We have follow the tutorial of ROS wiki that show how to write a plugin for rqt in Python to integrate a custom user interface.

6.4.3 Description of the program

We have created our python file with the template that we have found in the ROS wiki. The first thing that we have to do is import all the libraries that we will need.

sensor_takktile_module.py

```
1  import os
2  import sys
3  import rospy
4  import rospkg
5  import time
6  import serial
7
8  from std_msgs.msg import Float32
9  from qt_gui.plugin import Plugin
10 from python_qt_binding import loadUi
11 from python_qt_binding.QtGui import QWidget, QPixmap
12 from PyQt4 import QtCore, QtGui
13 from python_qt_binding.QtCore import QObject, pyqtSignal
14 from takktile_ros.msg import Touch
```

Touch.msg

```
1  float32[] pressure
```

We use the messages *Touch* that send a list of floats with the value of each sensor of the TakkStrip. We could use the other types of message that come with the basic program of Takktile, but we prefer to use this ones because they are calibrated, so we do not need to transform the output of the sensors.

The objective of the file is defining the plugin that makes up our GUI. We will describe the main function of this plugin.

sensor_takktile_module.py

```
[…]
25 class SensorTakktilePlugin(Plugin):
26
27   def __init__(self, context):
28     super(SensorTakktilePlugin, self).__init__(context)
29     # Give QObjects reasonable names
30     self.setObjectName('SensorTakktilePlugin')
[…]
45     # Create QWidget
46     self._widget = QWidget()
47     # Get path to UI file which should be in the "resource"
   folder of this package
48     ui_file=os.path.join(rospkg.RosPack().get_path('rqt_sensor_t
   akktile'), 'resource', 'interface_sensor_takktile.ui')
49     # Extend the widget with all attributes and children from UI
   file
50     loadUi(ui_file, self._widget)
51     # Give QObjects reasonable names
52     self._widget.setObjectName('MyPluginUi')
[…]
```

```
74      if context.serial_number() > 1:
75        self._widget.setWindowTitle(self._widget.windowTitle()+('(
    %d)' % context.serial_number()))
76      # Add widget to the user interface
77      context.add_widget(self._widget)
[…]
```

We use the function `super()` to call the method `__init__()` that expects no arguments at all. In line 30, we give a name to the plugin. The next step is to create the widget that we will join in the plugin. We will load the widget that we have previously created in QT Designer. The first thing that we have to do is extend the widget with all attributes and children from UI file (lines 48 & 50). The last step is to add the widget to the user interface (line 77).

We have to subscribe to the topic *takktile/calibrated*. As we have explained before the obtaining of the positions began when we click on the button below the screen.

sensor_takktile_module.py

```
[…]
22  class TactileValuesSignal(QObject):
23    signal = pyqtSignal(list)
[…]
34  self._tactileValuesSignal = TactileValuesSignal()
[…]
59  self._widget.pushButton.clicked.connect(self._subscribeToTopic)
60
61  self.red_string = "background-color: rgb(255,0,0)"
62  self.orange_string = "background-color: rgb(255,128,0)"
63  self.yellow_string = "background-color: rgb(255,255,0)"
64  self.green_string = "background-color: rgb(128,255,0)"
65  self.blue_string = "background-color: rgb(0,0,255)"
66
67  for i in range(1,4):
68    for j in range(1,6):
69      lcd_string = 'lcdNumber_f' + str(i) + str(j)
70      getattr(self._widget,lcd_string).setStyleSheet(self.blue_st
    ring)
71
72  self._tactileValuesSignal.signal.connect(self._tactileValuesSig
    nalHandler)
[…]
79  def _subscribeToTopic(self):
80    rospy.Subscriber("takktile/calibrated", Touch,self._callback)
81    rospy.loginfo('Subscribe to topic <takktile_calibrated>.')
82    for i in range(1,4):
83      for j in range(1,6):
84        lcd_string = 'lcdNumber_f' + str(i) + str(j)
85        getattr(self._widget,lcd_string).setStyleSheet(self.green
    _string)
86
87  def _callback(self, data):
88    values_string = repr(data)
89    values_string = values_string.replace('pressure:','')
```

```
90    values_string = values_string.replace(' ','')
91    values_string = values_string.replace('[','')
92    values_string = values_string.replace(']','')
93    line_list = values_string.split(',')
94    if len(line_list) == 15:
95      self._tactileValuesSignal.signal.emit(line_list)
96
97  def _tactileValuesSignalHandler(self, value):
98    for i in range(1,4):
99      for j in range(1,6):
100       lcd_string = 'lcdNumber_f' + str(i) + str(j)
101       lcdValue = abs(float(value[(i-1)*5+(j-1)]))
102       getattr(self._widget,lcd_string).display(str(lcdValue))
103       if 350 <= lcdValue and lcdValue < 500:
104         color_string = self.red_string
105       elif 200 <= lcdValue and lcdValue < 350:
106         color_string = self.orange_string
107       elif 100 <= lcdValue and lcdValue < 200:
108         color_string = self.yellow_string
109       else:
110         color_string = self.green_string
111       getattr(self._widget,lcd_string).setStyleSheet(color_stri
    ng)
112       #rospy.loginfo('LCDValue:<%d> ColorString:<%s>',lcdValue,
    str(color_string))
[…]
```

We define that when the interface is initialized the LCDs will be blue to show that we do not receive values. With the command of the line 59, we have fixed that when we click in the button "GET VALUES" we execute the function `_subscribeToTopic()`. The function `_subscribeToTopic()` is the responsible of making the subscription to the topic, it change the pixmap of all the LCDs to green and it calls the function `_callback()`, that is the responsible of modify the string that we receive from the publisher to have a list of values, that is the simplest way that we have found to process this values. The first thing that this function do is eliminating the characters and spaces, immediately afterwards we separate the numbers using the command split that return a list of the words of the string separate by the character that we have specify. And if the list of values has the length that we spec, we send this list with a signal. We have created a new class `TactileValuesSignal` that creates a signal to connect our principal thread with this new one because we need a new thread that will be the responsible of update the color of the LCDs. This signal will call the function `_tactileValuesSignalHandler` that is the responsible of updating the color of the LCDs in function of the value of the sensor. We have created a loop to avoid calling all the LCDs separately, the name of the LCDs has a common structure *lcdNumber_f<num.finger><position>*. So we go over all the LCDs and we change their color in function of the value that they display.

7 Force analysis of the emptying of a bottle

As an application of the force and moment sensor that we have installed on the wrist of the robotic arm. We want to do the analysis of the forces exerted on the wrist when we serve a bottle of water. The first thing we have done is defining the experiments that will be done to get all the necessary information. There in the future make a program that is able to detect the amount of water in the bottle and serve part of their contents. We will perform these two experiments:

- collect the values of the forces in all three directions during the inclination of the bottle of water. We will collect the values every 10 ° until 90 ° and keep the bottle closed (cap screwed).
- collect the values of the forces in all three directions during pouring of bottle of water. We will collect values through a graph in which the bottle will be inclined 90 degrees and hold it in that position until no more water falls.

7.1 First experience

We have collected values of the forces in all three directions during the inclination of the bottle from vertical to horizontal. We have collected values every 10º. We have obtained the following table:

Angle	Force		
	X	Y	Z
0º	10	-1300	-3450
10º	-1400	-1250	-2600
20º	-2700	-1100	-1600
30º	-3900	-860	-450
40º	-5000	-530	820
50º	-5980	-100	2150
60º	-6700	330	3530
70º	-7260	850	4870
80º	-7530	1340	6150
90º	-7600	1680	7500

Table 4: Forces during the tilt of the bottle

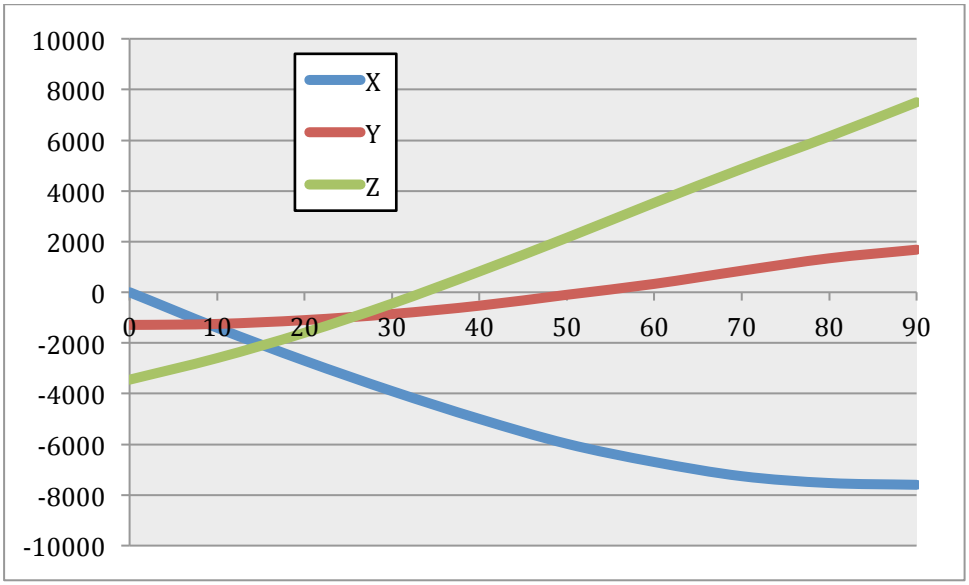


Figure 13: Graphic of the forces during the tilt of the bottle

The first problem that we find is that the reference of the axes of the force sensor spin with the hand so the weight of the object that we have grasped change of direction when we twirl the hand. We would calculate the value of the weight with the components X and Y of the sensor.

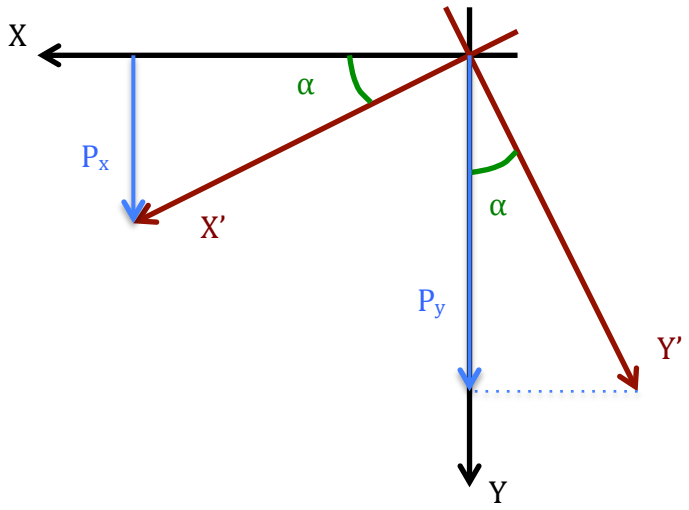


Figure 14: Calculation of the weight

$$P = P_x \cdot \sin(\alpha) + P_y \cdot \cos(\alpha)$$

Normally, the only force that is applied to the system is the weight that is effectuated in direction to the floor, so it has always the same direction. We take only the force effectuated in direction to the floor. We has obtain this table:

Angle	0º	10º	20º	30º	40º	50º	60º	70º	80º	90º
P	1300	1474	1957	2695	3620	4645	5637	6531	7183	7600

Table 5: Calculation of the weight

We see that the "theoretical" value is not constant it increases. This it can be caused because at the beginning the values of the sensor have been initialized with the weight of the hand. So we have thing to make the same experiment with the hand only and see which are the values of the forces in the inclination of the hand. And we will subtract these values to the values previously founded. These are the results:

Angle	Forces (Hand only)			Forces (Bottle only)		
	X	Y	Z	X	Y	Z
0º	0	0	0	10	-1300	-3450
10º	-760	40	420	-640	-1290	-3020
20º	-1420	140	910	-1280	-1240	-2510
30º	-2020	310	1460	-1880	-1170	-1910
40º	-2570	500	2070	-2430	-1030	-1250
50º	-3000	750	2720	-2980	-850	-570
60º	-3360	1060	3360	-3340	-730	170
70º	-3600	1400	4000	-3660	-550	870
80º	-3740	1770	4600	-3790	-430	1550
90º	-3760	2150	5180	-3840	-470	2320

Table 6: Forces of the hand and bottle separately

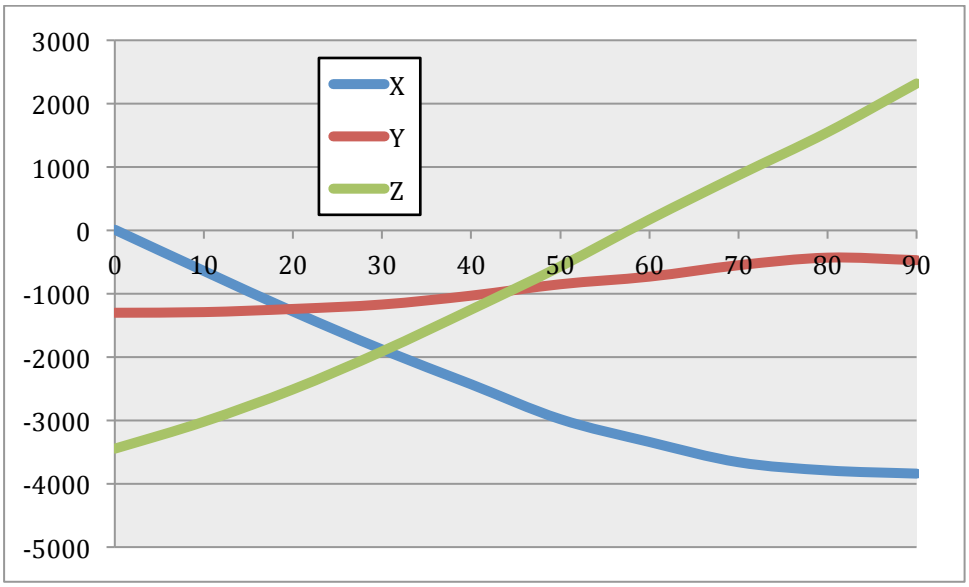


Figure 15: Graphic of the forces in the case when we have the bottle only

We see that the results continue to be illogical. We would expect to have only components X and Y and that this values would compensate between them.

We see also that the force in direction Z is not negligible; it's the one that change more his value. The force Z is caused by the action of flexion does to the weight of the hand and the bottle in the wrist. The weight of the hand is not uniform so the center of the forces is moved from the center of the sensor. This makes a force in direction Z. When the hand turn the value of the force Z change because the volume of the hand is not uniform.

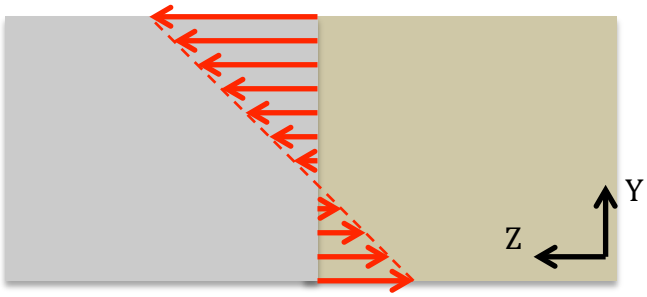


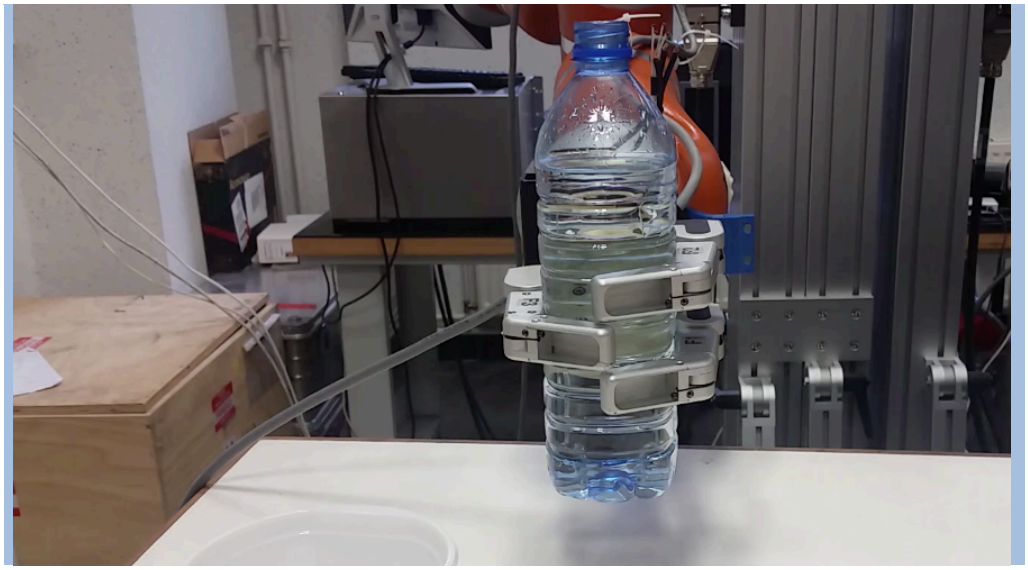
Figure 16: Flexion

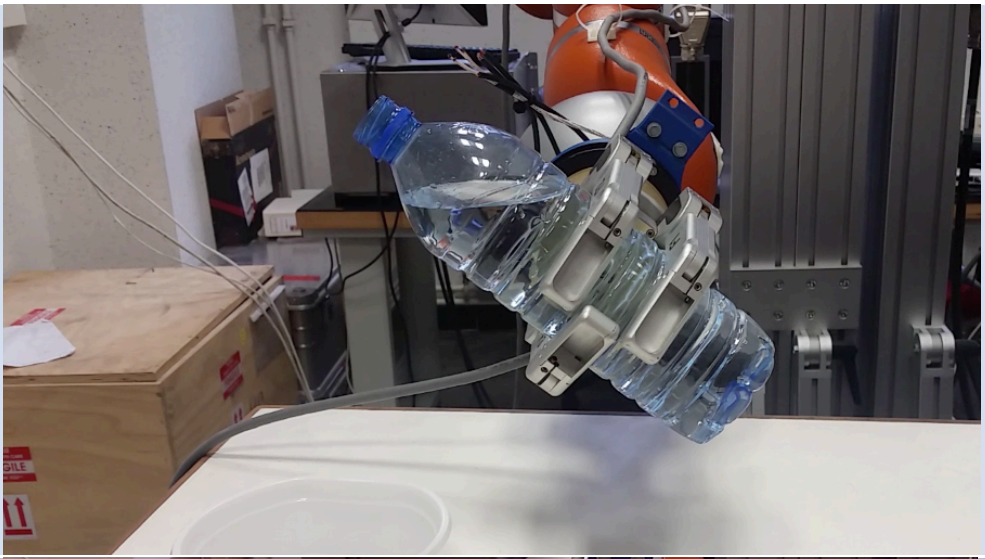
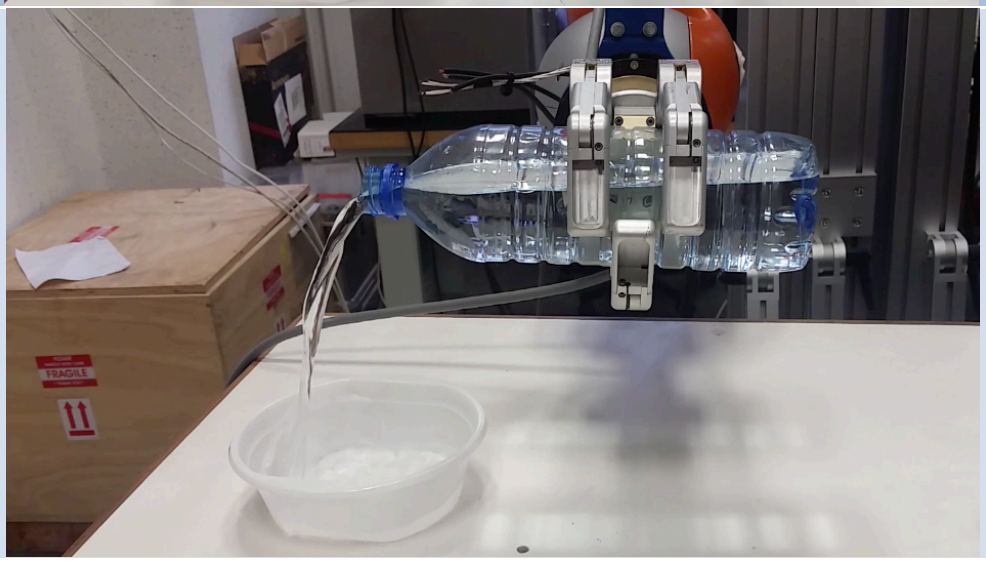
We see that if we want to find the weight of the object that we have grasp we will need to make an algorism that compensate the variations of the volume of the grasper and the object, to found the exact weight of them.

This could be another project due to the complexity of the solution. For this reason, we will only analyze the way that the force decreases when we empty the bottle.

7.2 Second experience

We have collected the values of the forces in all three directions during the pouring of a bottle of water in which the bottle will be inclined 90 degrees and hold it in that position until no more water falls. We have collected these values through a graph and we have filmed all the experience.

Number	Photo
1	

2	
3	
4	

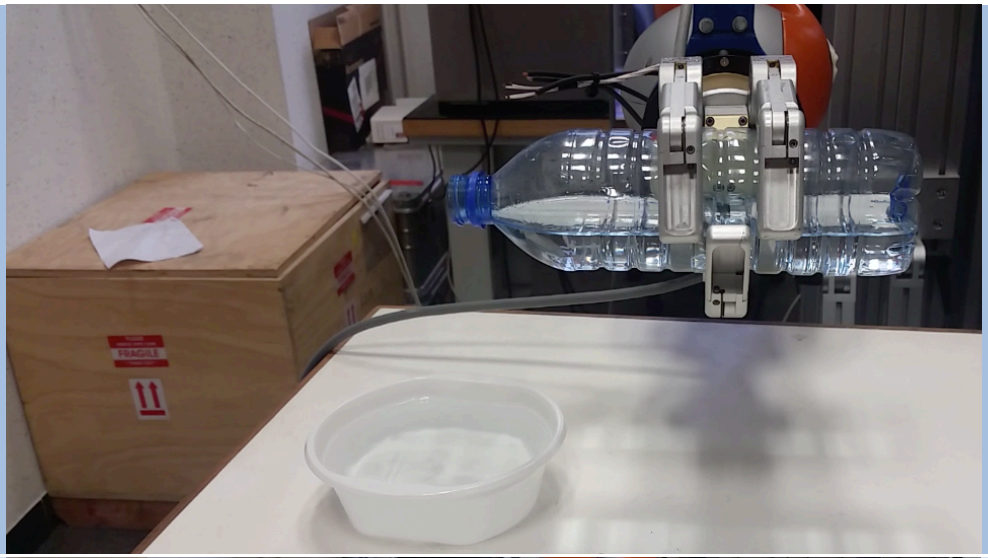
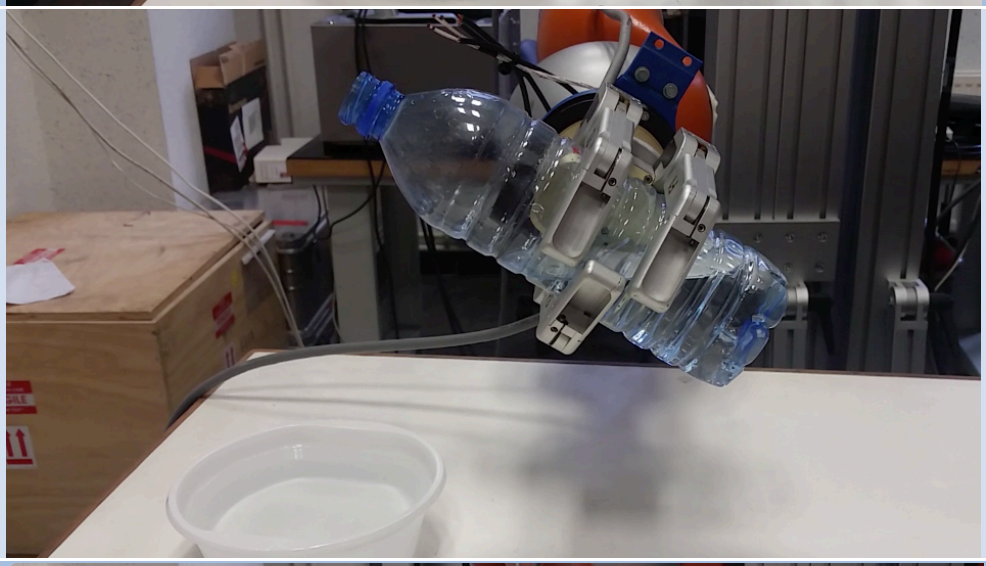
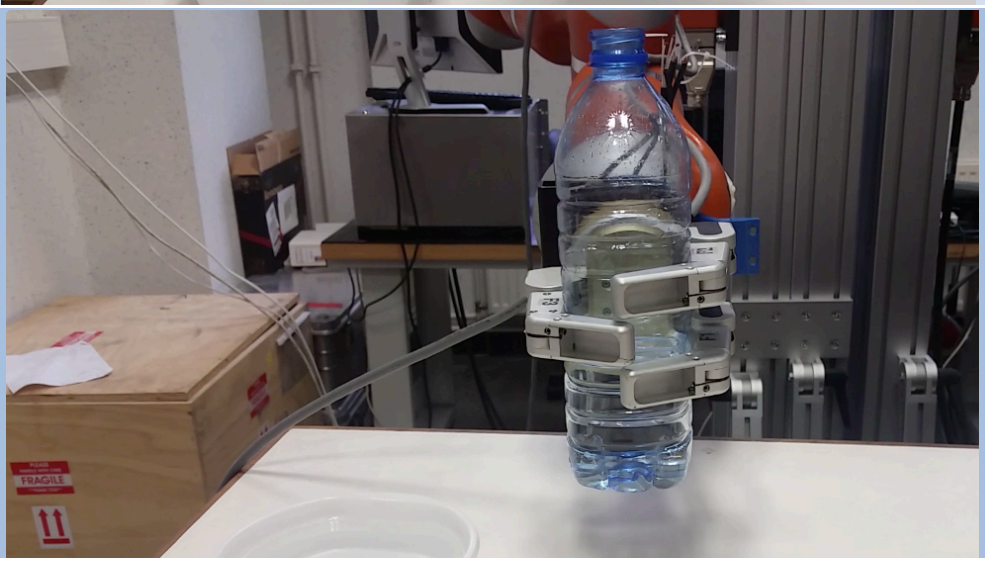
5	
6	
7	

Table 7: Photos of the second experience

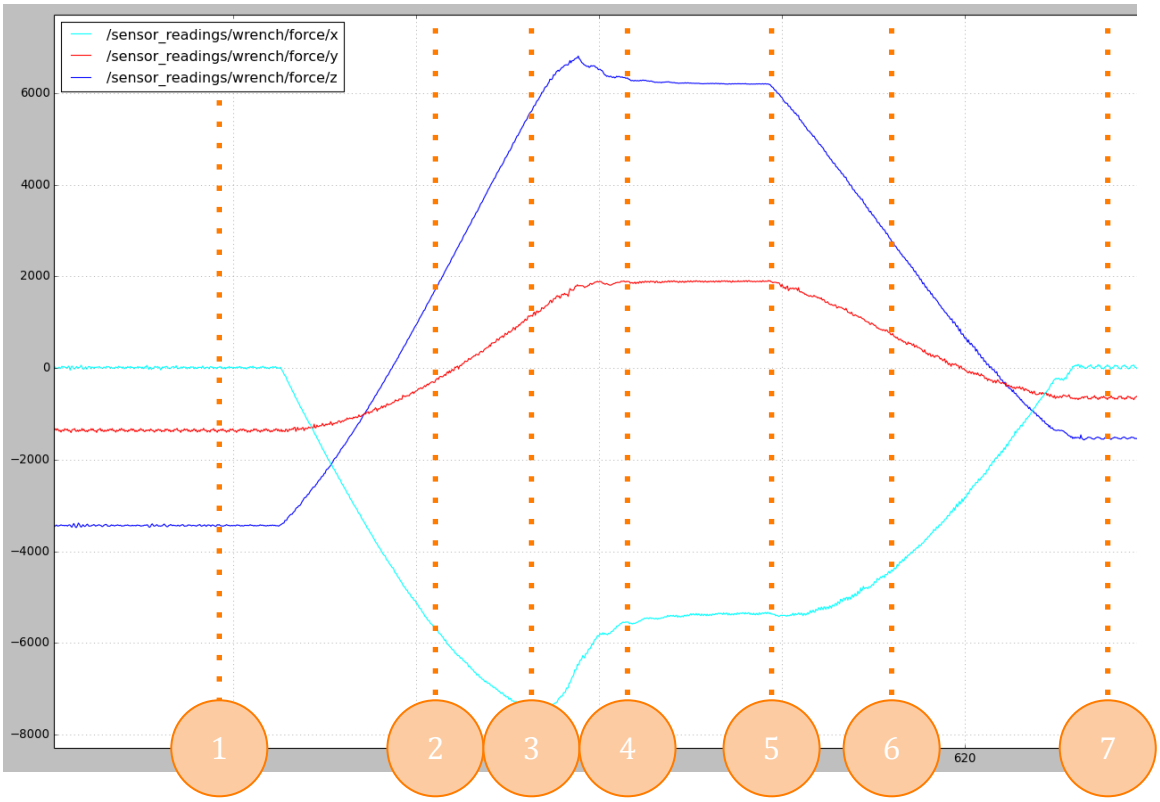


Figure 17: Graphic of the forces during the emptying

The first photo is the initial position of the bottle in the experience. We have begun to incline the bottle (photo 2) with a speed of the 10% of the Kuka Arm. The third photo is the moment when begins to fall the water from the bottle. We continue to incline the bottle until we reach the horizontal position (photo 4). We have maintained this position until no water falls from the bottle (photo 5). After, we have stood up the bottle (photo 6) until we reach its vertical position (photo 7).

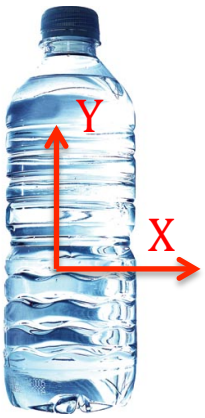


Figure 18: Axes relative to the bottle

At the beginning, we have only force in directions Y and Z. The force in the direction Z is caused by the force of flexion. When we begin to turn the hand we see that the forces change as in the first experience. When the water begins to fall we see that the force increases very fast and after it begins to stabilize slowly. The first value that changes is in direction X that in the horizontal position is the one that is perpendicular to the floor so it is the direction of the weight. We see that the force in Z also decreases because it is related to the weight that decreases because

we are emptying the bottle. The force in Y does not change because in this direction the weight does not affect. We see that when we stand up the bottle the force in X recovers its initial value, but in the other direction the values have decreased, the difference is the weight of the water that we have poured.

In conclusion, we have observed the same behavior of the force as in the first experience when we incline the bottle. And we have seen that when we pour the water it is at the beginning that falls the most part of the water.

These two experiences have shown us that is very difficult to find the weight of the object in any position because the sensor turns with the hand, so the axes change with its orientation. Besides, the flexion caused by the distance between the center of masses and the sensor cause a force in Z. This force depends on the distance between the center of masses and the sensor. Also it depends on the moment of inertia that depends on the geometry of the hand that is on top is variable. So the solution for finding the value of the object's weight that we have grasped is to develop an algorithm that in function of the position calculates the moment of inertia and finds the weight. This algorithm could be the subject of another project.

8 Design of camera support

8.1 Objective

In order to have more precision and new control features in future works, we have designed new supports to hold cameras in the wrist of the KUKA robot. This supports must be adapted to the wrist sensor, be adjustable in height and angle and don't interfere with correct functioning and handling of the robot.

8.2 Material

All the parts of the camera support for the BarrettHand are 3D printed parts. The design of the BarrettHand support does not have to support the weight of the hand, only the camera (100g), this allows us to use a more light material for the support, saving a lot of effort in the arms. The 3D printed plastic is ABS that we used with a 30% filling in the interior, is stiff enough to safely hold the camera without vibrations or undesired inertial movements. The weight of all the support with the bolts and nuts is …., not only this parts are really light, but they are cheap to make, so we can make and modify them easily and fast(printing time is around 2 hours).

On the other hand, as the design and set up are quite different and the support has to hold the hand too, the 3D printed plastic is not enough to resist all the weight of the shadow hand (4,2kg), so this support had to be made of aluminum (specs).

8.3 Design

In order to maximize the regulation options, we have made a modular design. This design allows us to change three different aspects of the camera's point of view.

-Height between the camera and the base of the wrist

-Angle between the hand's position and the camera position

-Camera angle

These regulations are possible thanks to the different parts, now we will proceed to explain how can each parameter be adjusted to have the best position for the camera.

8.3.1 Angle between the hand and the camera

The support that attaches to the arm and which is the base from where the camera will be mounted, looking for a better angle for the hand's palm and the grasping. In this case we have two different supports, one for the BarrettHand and other for the Shadow Hand.

8.3.1.1 BarrettHand Wrist support

The BarrettHand wrist support is a fully 3D printed piece with eight holes around. This piece was designed as a ring, with an interior 80mm diameter space, that leaves space enough to hold the wrist sensor placed in the kuka robot. The support has eight holes placed in the ring, with 90mm of diameter, where the 4mm bolts are located to attach the ring support to the BarrettHand mounter. In this piece, as in the ShadowHand support, we have eight different angle positions of the camera to watch the palm or avoid collisions with the robots environment.

This ring has an overhang with two holes, where we can place the height support. The holes are 9mm away from the exterior border, and 5,4 mm form the superior wall. As the rest of holes in the other parts of the support, it is designed to locate 5mm bolts with 5,5 mm diameter perforations.

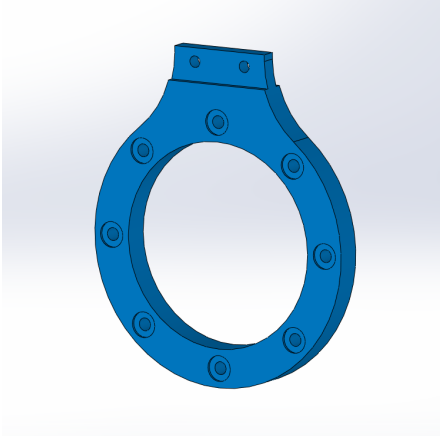


Figure 19: BarrettHand's main wrist support

8.3.1.2 ShadowHand Wrist support.

The shadow hand has a support that attaches the hand to the robot, arm, this adapter, does not have space enough for adding another matrix of holes where we could place an exterior support for the camera like we did in the BarrettHand. Therefore, in order to save weight and minimize the number of pieces, we decided to create a new link piece with an overhang, following the idea of the previous design. As this piece has to hold the weight of the hand, it has to be tougher and resistant than the 3d printed one, that's why it was made of aluminum.

This support is different to the BarrettHand support, this one has two functions, it holds the camera and the hand. This support has an internal circular hole, of 63mm of diameter that will fit in a platform that is attached to the Kuka arm.

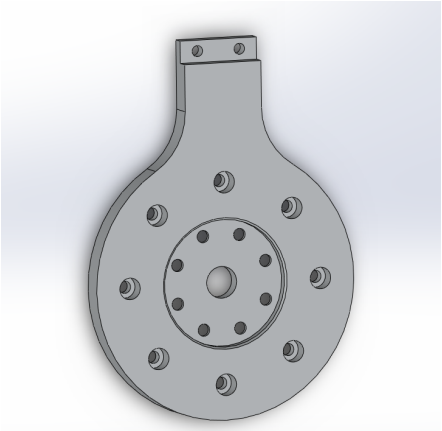


Figure 20: Wrist's main support for the ShadowHand

Once the support is placed in the arm, the bolts can be screwed. The holes are placed in a circular matrix of 50mm of diameter. The bolts used in the Kuka support are 6mm wide, and have a chamber, so the piece had to be designed so it could hold the whole bolt, because if the any bolt is not completely covered, then

the arm cannot be placed. That is why we have done a 5 mm chamber with exterior diameter of 16.11mm and interior diameter of 6.11mm.

Due to the mechanizing process, the extrusion where the height support is placed could not have been placed in the center, so it is based in the back face of the holder.

8.3.2 Height between camera and support

To regulate the height between the camera and the arm, we have designed spacers that will allow us to have better view of the hands palm. This part is specially useful for the shadow hand, as it has really long forearm (more than 30cm). These pieces are fit-table between them; therefore we can mount many of them. Each of these pieces add 21.2mm of height, but in case we want a specific distance, we can easily redesigned the piece changing the parameters with a CAD editor.

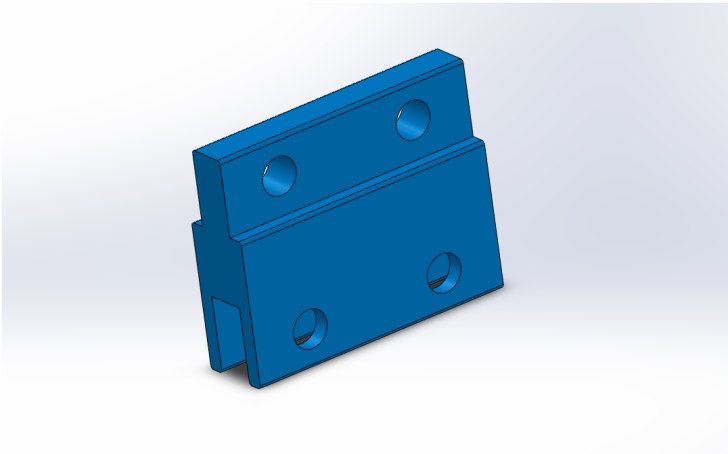


Figure 21: Height spacer

These pieces have four holes, two on the top for the upper part and other two at the bottom for the base piece. The top part of the piece, the extrusion where the two upper holes are placed, has a thickness of 6 mm. On the other side, the bottom part has two walls of 2,4 mm with and interior space of 6mm. These holes are designed for 5 mm bolts, but they have a 5.5mm diameter, we have designed them with this extra space due to lack of precision in the holes and for avoiding fatigue from the bolts. These holes are 11mm long from the center of the piece and 5.4mm away from the top/bottom border and 9mm from the side border.

8.3.3 Camera angle support

This parameter can be fixed thanks to two parts the camera base support and his holder. The base support of the camera is the piece that allows us to attach the camera, for this, the support has four holes where we introduce the bolts, so we can connect the camera to this support. The piece has two horizontal holes that we use to link the base to the following part. These holes are a bit displaced from the center; therefore the mass center of the camera is closer to the line of the support, avoiding excessive torques in the bolt that could create displacements in the camera position. This horizontal holes are where the long bolt will be placed, this bolt along with the nut, will do pressure in the connection between the two parts, fixing the angle of the camera.

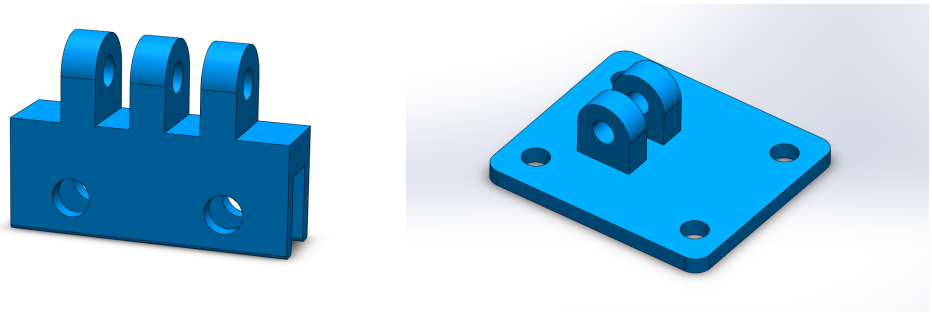


Figure 22: Fitter and camera support for the angle fixing

The other part is a holder this piece is similar to the height extension piece, but with a different top, that allows us fit it to the base support, so we can fix them two doing pressure between all the aligned horizontal holes with a bolt and a nut. This piece adds 17.10 mm of height, and as the height enlarger, the two bottom walls are 2.4mm thick with an internal space of 6mm.

9 How to use the programs

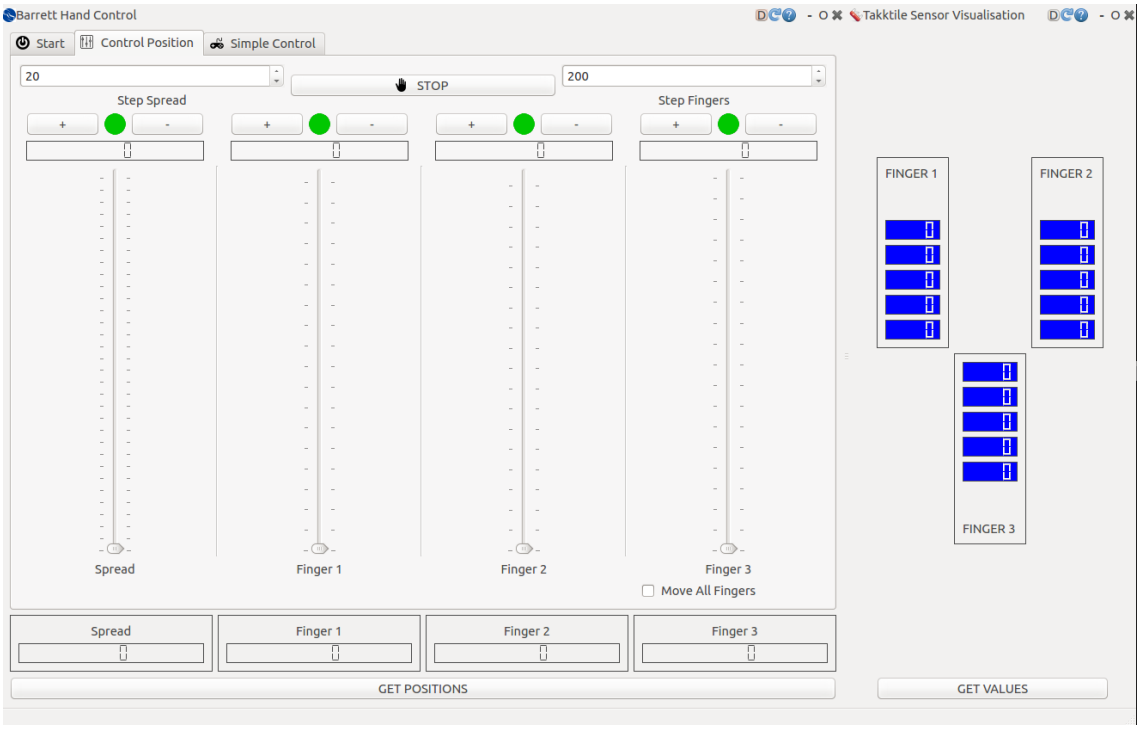


Figure 23: Programs in RQT

9.1 Server

The first thing that we have to do is connect the hand to his power supply (Figure 9). The Power Supply for the BH8-SERIES can be plugged into any regular AC power source. It provides DC motor bus power (24V), electronic component logic power (5V), and supports RS-232 communications between the host computer (via the serial cable) and the control electronics in the BarrettHand palm shell (via the Hand Cable).

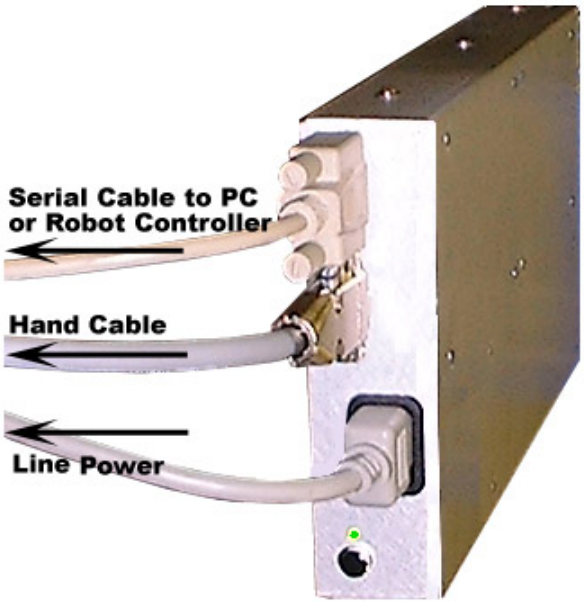


Figure 24: Cable connections

Once we made the connections, we can begin with the execution of the server program. First of all, we have to run `roscore` that is the first thing that we should run when we use ROS. After, in a new terminal, we have to open the catkin workspace where is the file.

```
$ cd catkin_barrett_ws
```

The next step is to source our workspace's setup.sh file.

```
$ source devel/setup.bash
```

The last step is to run the program.

```
$ rosrun BHand_Control BHand_Server
```

If we don't see any message is that the program is initialized and without errors. The program would create a topic called *BarrettState*, we can see what is publish in this topic, with the command:

```
$ rostopic echo /BarrettState
```

The program has also created a service called *CommandBH*.

9.2 BarrettHand Control Interface

First of all, we have to run `roscore`, if we don't have done it yet. After, in a new terminal, we have to open the catkin workspace where is the file.

```
$ cd catkin_barrett_ws
```

The next step is to source our workspace's setup.sh file.

```
$ source devel/setup.bash
```

The last step is to run RQT.

```
$ rqt
```

Once RQT is open, we open the menu *Plugins* in the top of the window. We choose in the folder *Barrett Hand Tools* the plugin called *Control Interface*.

9.3 Wrist Sensor

9.3.1 Setting the environment

For connecting the wrist sensor, first we will have to pay attention to the connections of the NetBox. There are three main connections, the power supply, the data receiver from the sensor and the Ethernet cable that connects the box to the computer.

Once everything is connected, we will have to change the IP of our computer. The Net F/T sensor box has its IP set to 192.168.1.1, and it will have it reset to it every time the switch number nine is in ON position. The default IP address of the computers is the same, so we will have to change our IP address before been able to get into the NetBox and set it to as we want it. Therefore, for changing the ip address of our computer we have to go:

System configuration > Network > Wired configuration > Options

This opens a new window with five different tabs, we click the fourth tab, IPv4 Adjustments. Here we will set the IP we want, first we have to change the method to Manual, so now we can add our IP address. It can be any IP except the first one, in this case we created the 192.168.1.100.

9.3.2 Publisher

Now the computer is set, we can run the subscriber in the terminal. First of all we have to open an independent terminal and execute roscore:

```
$ roscore
```

Once the roscore is running, we open a new terminal. In this new terminal, we open the catkin workspace.

```
$ cd catkin_ws
```

Once we are in the workspace, we update the list of packages and source its setup.bash file.

```
$ catkin_make
$ source devel/setup.bash
```

Now we can call our package

```
$ rosrun force_torque_sensor ftsensor_CLIENT
```

This will run the package, and it will start showing all the values in the terminal.

9.3.3 Interface

While the roscore and the ftsensor_CLIENT package are running, we can run the interface of the sensor, for this we have to first move to the catkin's workspaces

```
$ cd catkin_ws
```

And now source its setup.bash filename

```
$ source devel/setup.bash
```

The last step is to run rqt

```
$ rqt
```

Now that the rqt window is open, we just have to open the plugins tab, and select:

```
Barretthands tool > Wrist sensor Interface
```

This will open the interface with the bars and displays so we just have to click subscribe for start receiving all the values measured in the sensor.

9.4 Sensor Takktile Interface

9.4.1 Publisher

First of all, we have to connect the sensors to the TakkFast. We have defined a color-coding to avoid make an error when we connect the sensors to the card.

Figure 25: Position of the pins in TakkFast

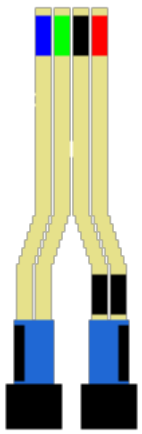


Figure 26: Color-coding to differentiate wires

The pins that we has weld in the TakkFast are all them connected to the I2C bus, so is not important the order of the sensors. But we have to watch where it is connected each cable of the sensor, for this reason we have defined a color-coding. We have paint the end of the cable to differentiate which cable is it without having to follow the wire from the base. The figure 10 indicates the position where we have to put each cable.
First of all, we have to run roscore that is the first thing that we should run when we use ROS. After, in a new terminal, we have to open the catkin workspace where is the file.

```
$ cd takktile_ros
```

To compile the catkin workspace, we will use the command:

```
$ rosmake
```

rosmake is a ROS dependency aware build tool which can be used to build all dependencies in the correct order. The next step is run the publisher:

```
$ rosrun takktile_ros takktile_node.py
```

This program publishes three types of messages. If we want to show them we will use this commands:
`$ rostopic echo /takktile/sensor_info` (check how many sensors and which are connected)
`$ rostopic echo /takktile/raw` (show raw values)
`$ rostopic echo /takktile/calibrated` (this will set the zero point for the 'calibrated' topic)

9.4.2 Interface

First of all, we have to run `roscore`, if we don't have done it yet. After, in a new terminal, we have to open the catkin workspace where is the file.

```
$ cd catkin_barrett_ws
```

The next step is to source our workspace's setup.sh file.

```
$ source devel/setup.bash
```

The last step is to run RQT.

```
$ rqt
```

Once RQT is open, we open the menu *Plugins* in the top of the window. We choose in the folder *Barrett Hand Tools* the plugin called *Sensor Takktile*.

10 Conclusions

Once the project is finished we have reach to integrate in ROS the robot and the sensors. We have created a program that executes commands that we send him in the Supervisory Control. Therefore, we can use this program in the future to control the hand by programs integrate in ROS. We have also created an interface to control the BarrettHand. It is a simple interface to expand the range of users who can use it. We have also integrated a force and torque sensor in the wrist of the robotic arm. We have also made their integration into ROS; the program publishes the values that get the sensor, and we have created an interface to display these values by means of color bars and LCDs. We have redesigned the fingers to incorporate touch sensors in the fingertips. We have also made their integration into ROS, with the creation of the respective program that reflects the values and published them on a topic. We have used these values to create a simple interface that displays these values by the way of LCDs that change of color depending on the value. The integration of these sensors will improve the ability to pick up objects and recognize them.

We have also begun a study of the forces that suffers the wrist sensor when we pick and serve a bottle of water. We have observed that due to the rotation of the sensor and that the hand geometry change depending on its orientation, the treatment of these values is not an easy task and this problem could perform another project.

We have also design two brackets to install a camera on the wrists of the two robotic arms. These brackets allow in future projects to improve the detection of new objects with a new point of view.

In conclusion, we can say that this project has been a job the integration of the robotic hand and the sensors to enable the realization of future projects by the IFMA School and the Institute Pascal, under the ROBOTEX project. All the results we have obtained will help future students and researchers that would work with Barrett hand.

11 References

[1]. **ATI Industrial Automation**, *Netbox 9105 user manual*, http://www.ati-ia.com/app_content/documents/9610-05-1022.pdf

[2]. **Barrett Technology Inc.**, *BarrettHand BH8-262*, http://support.barrett.com/wiki/Hand/262 [July 2015]

[3]. **Barrett Technology Inc.**, *BH8-Series User Manual Control Version 4.4.x*

[4]. **Génération Robots**, *Qu'est ce que ROS?*, http://www.generationrobots.com/fr/content/55-ros-robot-operating-system [July 2015]

[5]. **GitHub**, *BarrettHand 280 URDF*, https://github.com/jhu-lcsr/barrett_model [July 2015]

[6]. **GitHub**, *Netbox subscriber*, https://github.com/CentroEPiaggio/force-torque-sensor [July 2015]

[7]. **ROS.org**, *Moveit Tutorials*, http://moveit.ros.org/documentation/tutorials

[8]. **ROS.org**, *ROS Tutorials*, http://wiki.ros.org/ROS/Tutorials [July 2015]

[9]. **TakkTile LLC**, *TakkFast*, http://takktile.com/product:takkfast[July 2015]

[10]. **TakkTile LLC**, *TakkStrip*, http://takktile.com/product:takkstrip[July 2015]

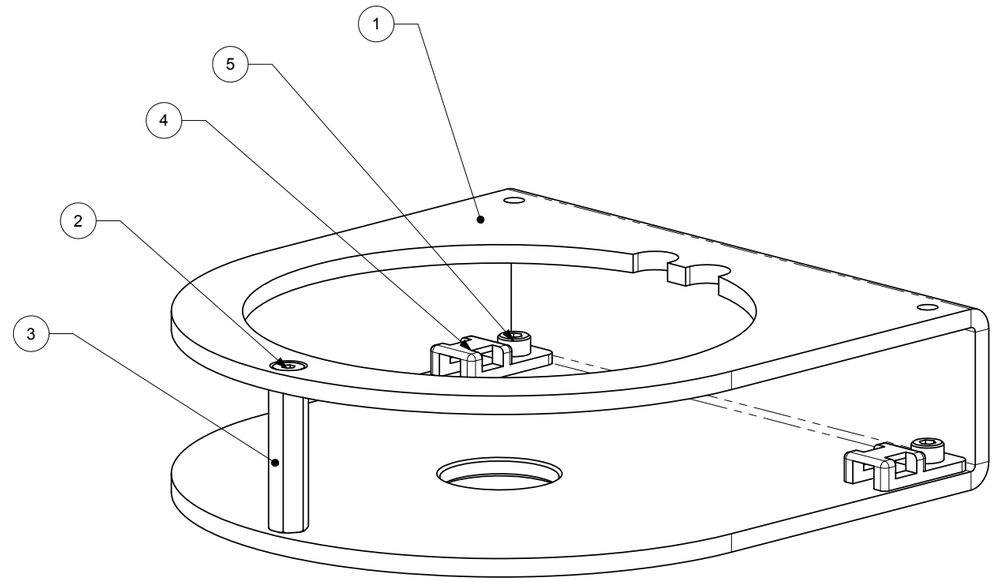
[11]. **TakkTile LLC**, *TakkTile Tutorials*, http://takktile.com/tutorial:main[July 2015]

[12]. **The QT Company**, *QT Designer User's manual*, http://doc.qt.io/qt-4.8/designer-manual.html

[13]. **William Townsend**, *The BarrettHand grasper – programmably flexible part handling and assembly*, Industrial Robot: An International Journal – Volume 27 – Number 3 – 2000 – pp. 181-188

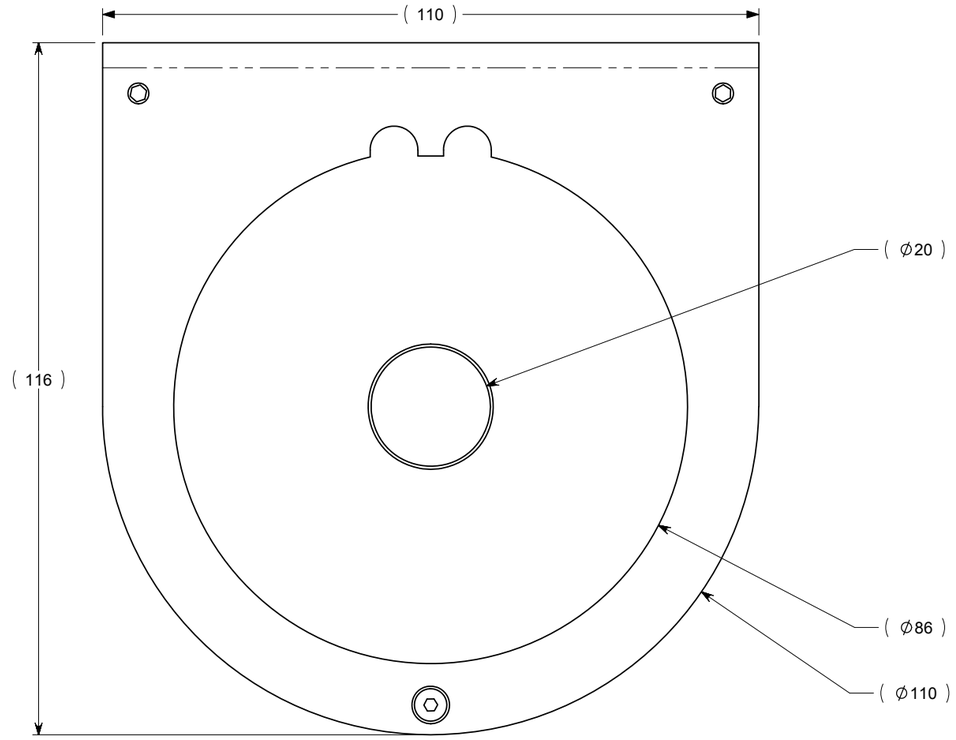
ANNEXES

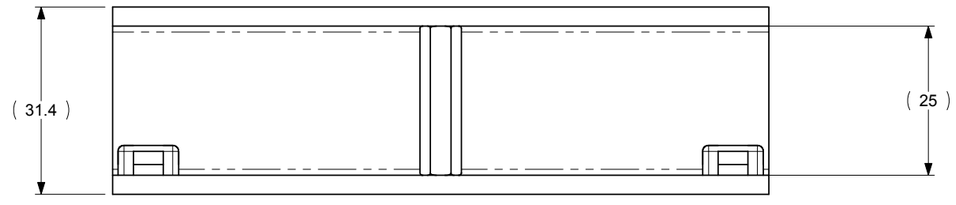
Item	Qty	P/N	ModelRev	Description	DwgRevsAccepted
1	1	B1401	AB.00	AdapterPlate	
2	2	B1071	AA.11	M3-0.5x12 FHCS, SS 18-8	
3	1	B2698	01.02	HexStandoff-M3-Fem-Fem-25lg	
4	2	B2921	01.01	zip-tieAnchor-offset	
5	2	B3214	01.01	M3-0.5x4 SHCS	



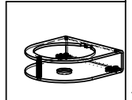
ISOMETRIC VIEW

TOP





FRONT



Barrett Technology, Inc.
625 Mount Auburn St, Cambridge, MA 02138 U.S.A.
www.barrett.com Ph +617-252-9000 Fx +617-252-9021 mfg@barrett.com

Created by: on Wednesday, November 21, 2001 6:10:19 PM Modified by: dw on Friday, January 22, 2010 2:58:10 PM Model Rev: AA.00 Date printed: 1/22/2010

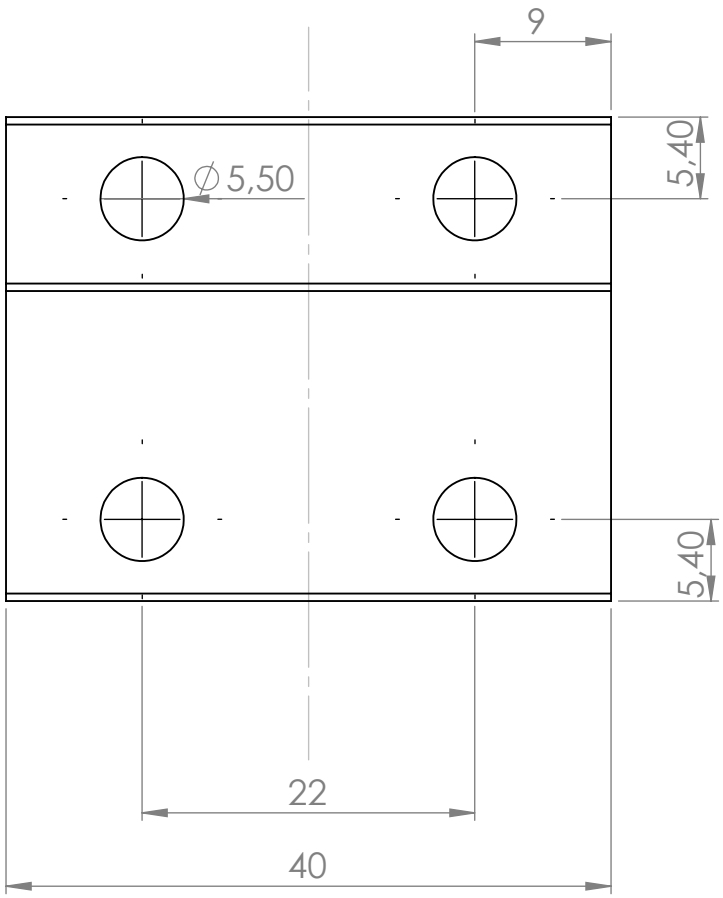
ASSEMBLY DRAWING

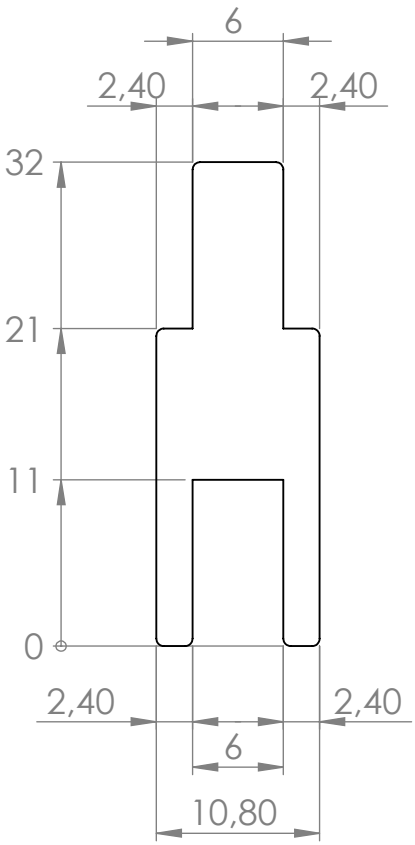
BHand-Assemblies

Drawn	DL	Jul 28, 2005
Checked	DW	Jan 22, 2010

AdapterPlateAssembly

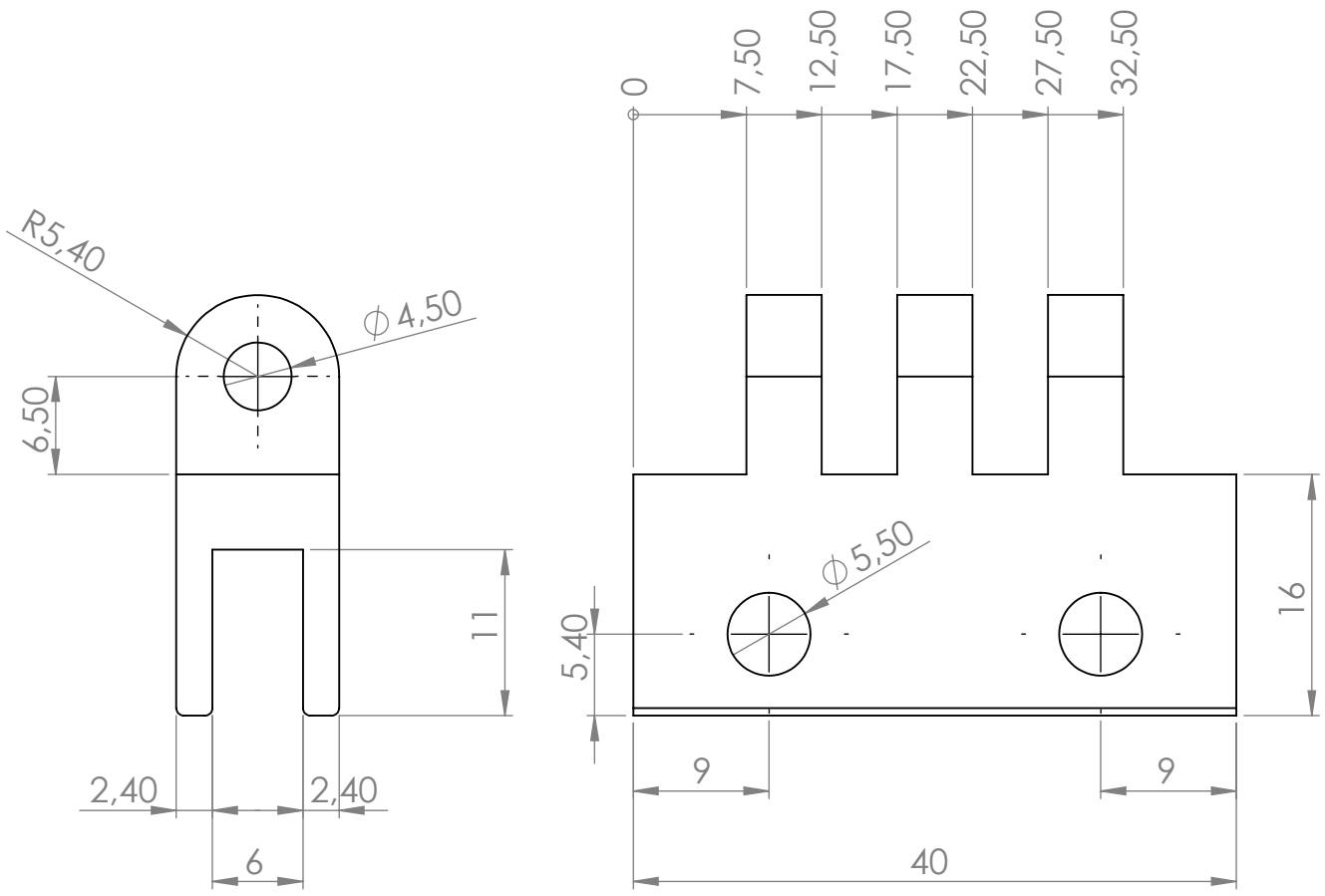
Size **B** | PN **B1402** | REV **AA.00**
Scale **1:1** | Mass **169.606 g** | Sheet **1 of 1**





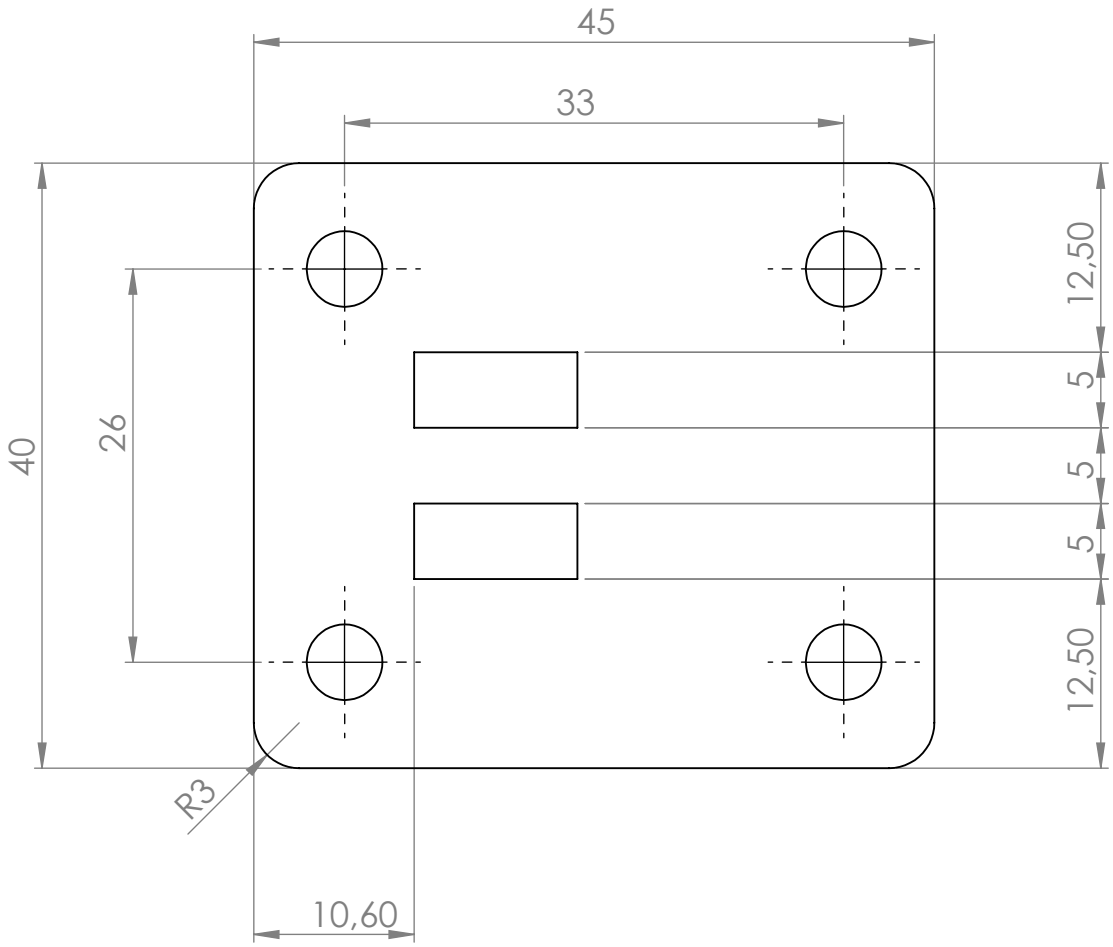
ANNEX 2

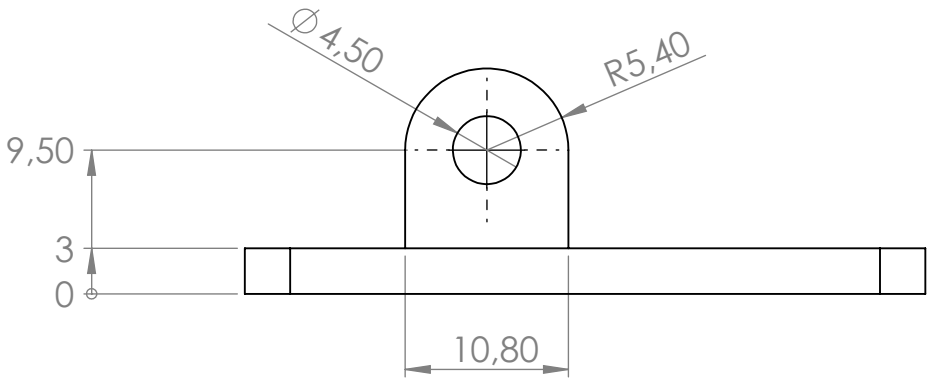
HEIGHT SPACER



ANNEX 3

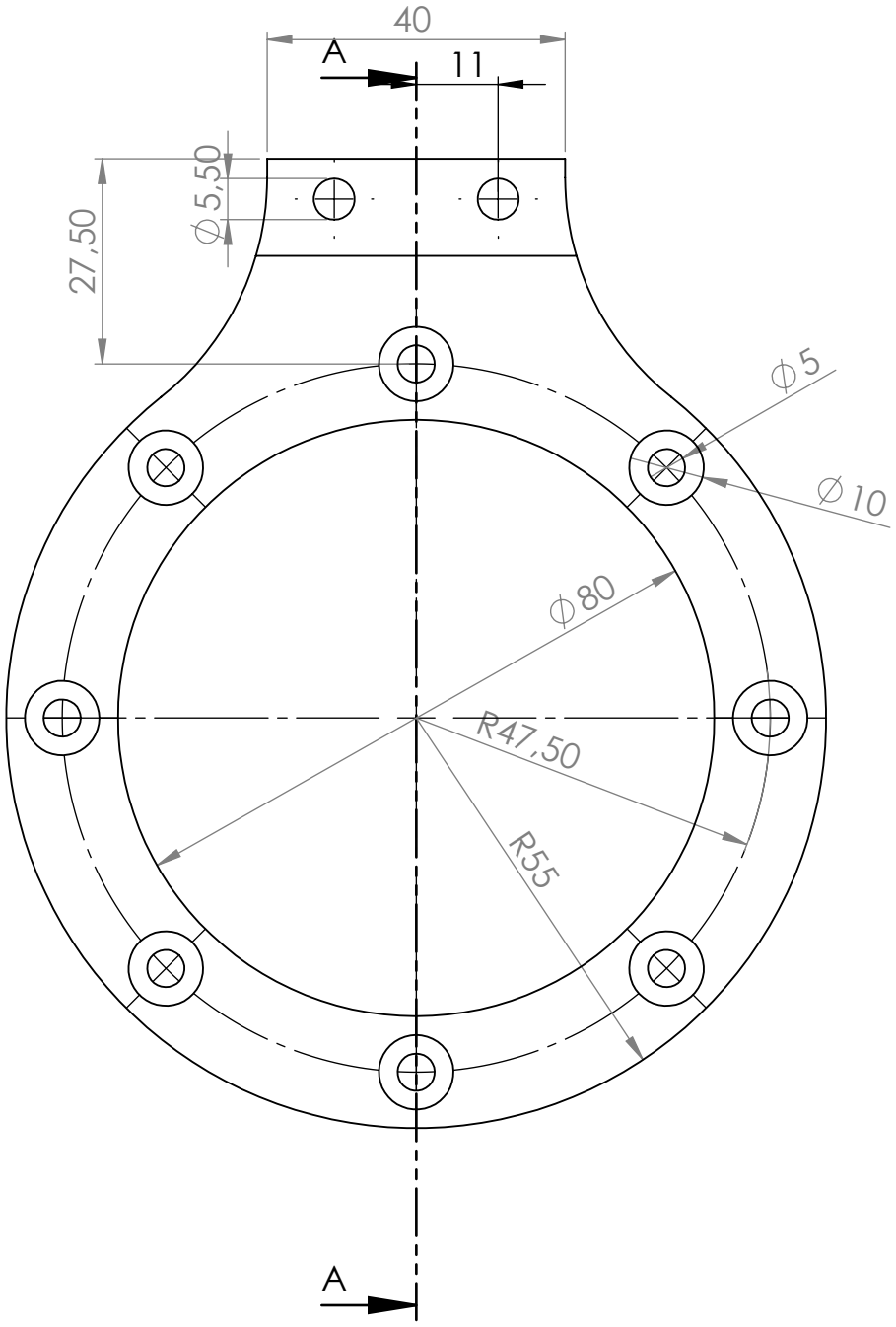
CAMERA ANGLE MOUNTER

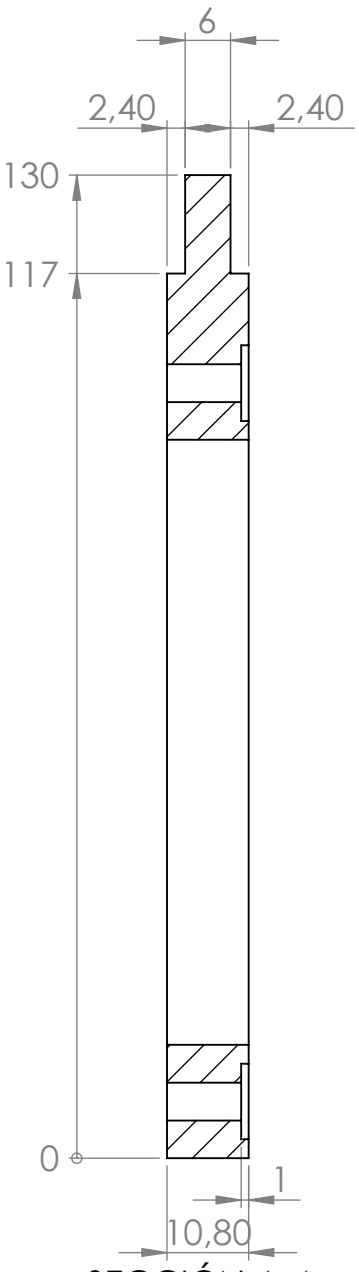




ANNEX 4

CAMERA BASE

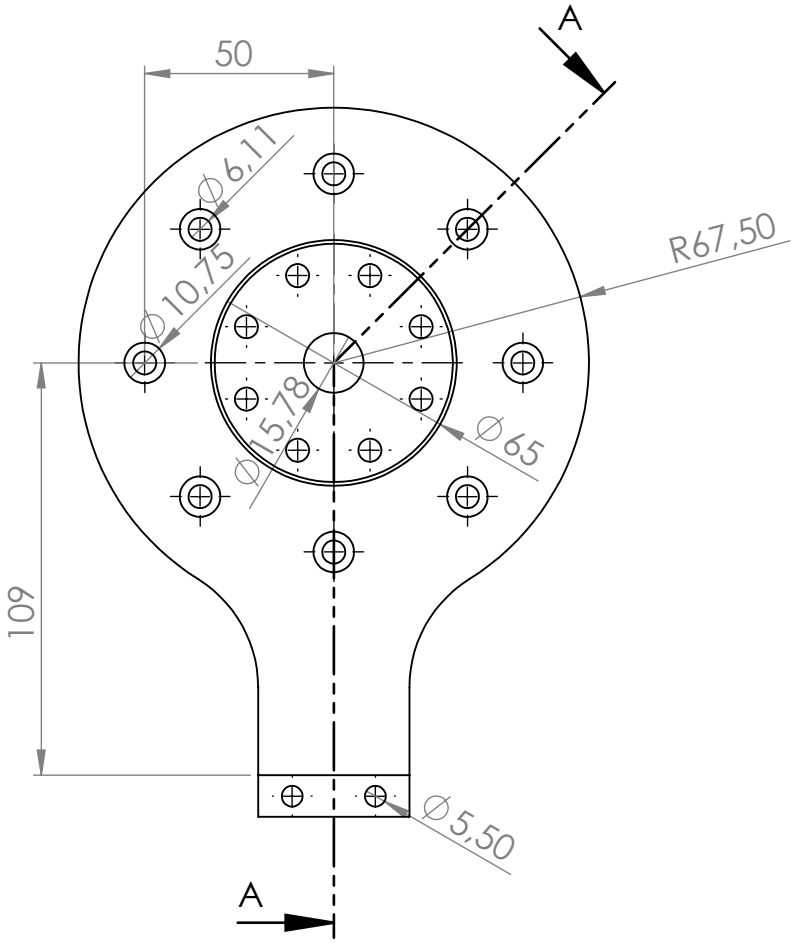


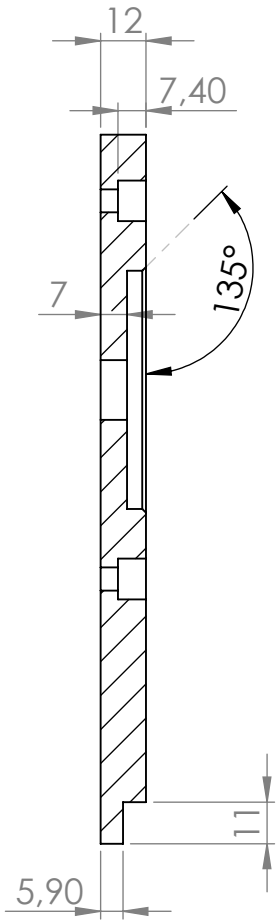


SECCIÓN A-A
ESCALA 1 : 1

ANNEX 5

BARRETTHAND BASE RING

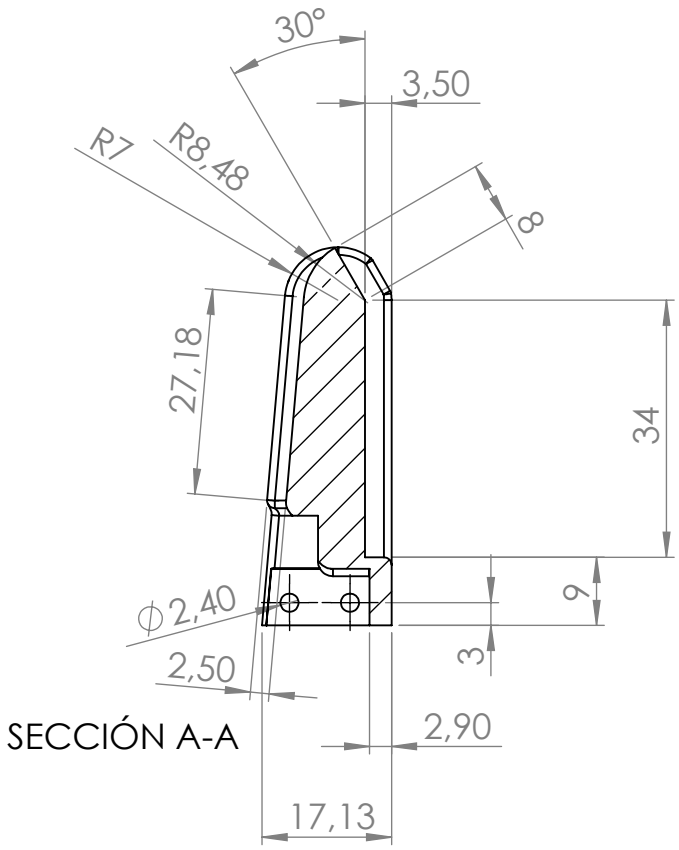




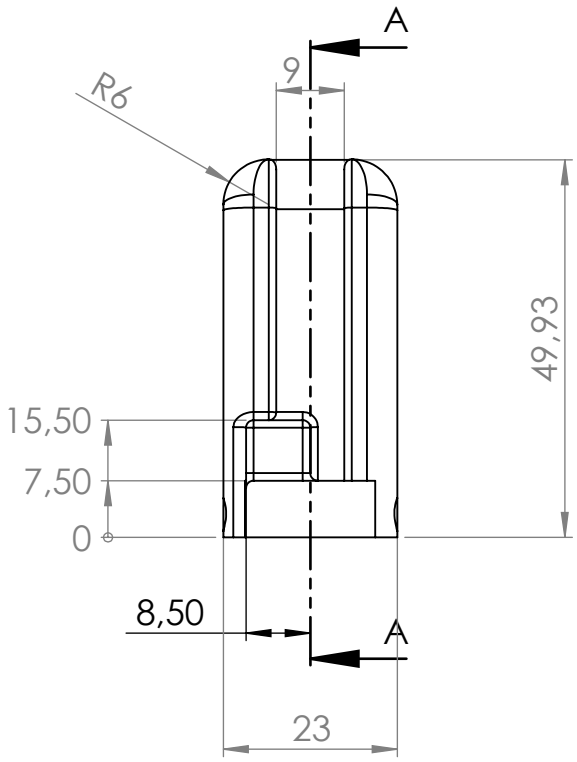
SECTION A-A
SCALE 1 : 2

ANNEX 6

SHADOW HAND RING SUPPORT



SECCIÓN A-A



ANNEX 7

FINGERTIP