Teleoperation of miniaturized humanoid robots

Adrian Llopart Maurin (s131358)

Kongens Lyngby 2015

Technical University of Denmark
Department of Electrical Engineering
Ørsteds Plads, building 348,
2800 Kongens Lyngby, Denmark
Phone +45 4525 3800
elektro@elektro.dtu.dk
www.elektro.dtu.dk

Abstract

The primary purpose of the present paper is to investigate, build and prove the viability of different low-budget teleoperation mechanisms for humanoid robotics. Telerobotics deals with the operation and control of robots, by a human, from a certain distance. The end result of this thesis is a robot that mimics the users movements and actions wirelessly while, simultaneously, providing visual feedback. To accomplish this project, the *Bioloid* robot, commercialised by *Robotis*, will serve as a mechanical framework on top of which various devices for wireless communication (*Xbee*, *Zigbee* and *WLAN*) and telerobotics (*Kinect* camera, *Oculus Rift*, *Arduino* boards, sensors ...) will be built.

The thesis is structured into 5 parts, each one designed as a small project by itself, thus all include theoretical background, design, prototyping, result evaluation and conclusions.
The first section consists of the control of the robots arms using a *Kinect* camera. With this, the joint positional data of the users hands is retrieved, then inverse kinematics and some processing are applied and the information is then sent via *Zigbee* onto the robots actuators.
The next part deals with the acquisition of data from flex sensors, located on the users hands, which is sent through *Xbee* communication to a designed and laser cut robotic hand/gripper for precision grasping.
Similarly, the third part uses joystick information to steer the robot in a desired direction.
Fourthly, the virtual Reality headset *Oculus Rift* will be integrated as a way to provide visual feedback from the robot to the user. To do so, a webcam or smart phone will livestream what the robot is seeing at all times and this footage will be visualized simultaneously on the headset by the user. Additionally, the orientation of the *Rift* will be obtained and transferred via *Xbee* communication

to a designed head support on the robot, therefore achieving yaw and pitch rotations.
Finally, a voice recognition and speech synthesizer software will be implemented to add more control mechanisms in case of failure from previous systems.

A deep understanding of the functionalities and limitations of each technology was key to the fulfilment of this project. These mechanisms were then tested, evaluated and several conclusions for real life applications were drawn.

Preface

This thesis was prepared at DTU Elektro in fulfilment of the requirements for the acquisition of an M.Sc. in Electrical Engineering, following the study line of Automation and Robotics.

The work has been carried out from April 2015 to August 2015 in the University of Tokyo, Japan, under the tutelage of Takeo Igarashi (Professor, Department of Computer Science, Tokyo University), Daisuke Sakamoto (Associate professor, Department of Computer Science, Tokyo University) and David Johan Christensen (Associate professor, Department of Electrical Engineering, DTU) as an exchange program between DTU and Tokyo University.

I want to thank all my supervisors for giving me the unique opportunity of writing a thesis in a country so far away and different from what I had experienced so far. I have been overwhelmed positively by it, which not only has helped me become so much more professionally but also personally, opening my eyes to something so drastically distinctive, but at the same time comfortably familiar. Special thanks go to my family, friends and partner for keeping me on track, focussed and happy, lifting me in my down moments but keeping me on the ground when I was full of it.
Thank you so much.

Lyngby, 15-August-2015

Adrian Llopart Maurin (s131358)

List of abbreviations

Abbreviation	Full name / Description
DoF	Degrees of Freedom
IMU	Inertial Measurement Unit sensor
IDP	Integrated Development Platform
API	Application Program Interface
DH	Denavit-Hartenbergs
IK	Inverse Kinematics
OR	Oculus Rift
SDK	Software Developers Kit
OSC	Open Sound Cloud
UART	Universal Asynchronous Receiver/Transmitter
VR	Virtual Reality
Servo	Servomotor
UNO	*Arduino UNO* microcontroller

Contents

Chapter 1

Introduction

This thesis will deal with the design, implementation and evaluation of teleoperation of humanoid robots. To do so, a *Bioloid* robot will be remotely controlled by a human. The user will issue commands through movement and voice whilst also receiving feedback from the robot simultaneously. The concept of the project is to have a humanoid automaton mimicking the precise movements of the human and follow their every command exactly to the point the user feels the robot is part of his own body, even though they will be in different locations. The remote control will be done wirelessly and will include head tracking and visual feedback, exact end-effector positioning for arms, gripper government, robot direction and orientation control and, finally, voice activated commands.

1.1 Background of the problem

Not a week goes by before we hear on the news about the multiple disasters and human fatalities that have happened lately, and many more are not even shown on TV. Fires, collapsing buildings, earthquakes, floods, explosions of nuclear reactors, tsunamis, terrorism and many more catastrophes, which are very difficult to prevent, claim hundreds, sometimes thousands, of lives every day and there is little to nothing that humanity can do about it. What we can do, however, is act fast to prevent the loss of further lives thanks to rescue

missions and humanitarian aid. In spite of the effort and sacrifice rescuers put whilst doing their jobs, sometimes it is not enough, and in doing so they risk their own lives to an extent where it is simply not worth it. This is due primarily because of the perilous and threatening environments they work in. Therefore, finding and innovative way to replicate the service these people provide and, at the same time, not putting their lives on the line, becomes essential in a world were danger lurks in every corner. What if we could somehow maintain the work of these heroes without placing them in dangerous situations.

Lately, technological advances have lead to the creation of autonomous robots that are very suited for rescue operations, the best example of this is the *Atlas* robot designed by *Boston Dynamics* [3]. In spite of this, it is evident that they still have a long way to go, specially when it come to dealing with objects designed for humans. A situation that clearly exemplifies this is when robots need to open doors or use tools: the world has been created by humans and only human-like robots will be able to have a full advantage of its surroundings. But this is not enough, rescue robots must also face and solve immediate problems in a fast and precise way, they must be prepared for anything at any given time. This kind of artificial intelligence does not exist currently, and it will be years before humankind develops it. There is, however, an easier and faster solution: the human mind. Imagine a robot that has the fastness of though of a human and the robustness of body of a machine. This is known as *Robot teleoperation*, controlling a robot from a safe distance, and it would certainly be of use for saving lives.

But rescue missions is not the only field that would benefit from robotic teleoperation. *Telepresence* is the union of technologies that allow a person to feel, appear and/or have the effect of being in a different place to where they actually are. Currently, the best examples of robotic telepresence are wheeled robots with a screen sitting on top of them, such as the ones from *Double Robotics* [6]. Even though being so simple, their price tag ranges from 2.000 to 70.000 $ (13.600 to 478.000 dkk). Not only that, but their limitations are quite obvious: to open doors one must "bump" into them so that other people notice and open them for you. So much for telepresence when you can't even open a single door. The main reason behind this is that these robots lack arms or legs and so the human isn't really transmitting his/her presence, but actually is just remote controlling a car (like a game) with a big screen on top showing their face.

Finally, one last aspect of *telerobotics* is that robots can do all those laborious and tedious tasks some humans have to endure nowadays in a whim of an eye. Some examples are the driving of trucks in hazardous environments [9] or other construction/industry related heavy lifting tasks. But others are more exotic like *NASA*s *Robonaut* telemanipulation, to replace astronauts, or even, in a near future, the concept of *Surrogates*, where teleoperated robots act as substitutes

for the social or pastoral role of humans, very much like the *Geminoid* robots designed by Hiroshi Ishiguro [11]. The possibilities are endless.

1.2 Purpose and significance of the study

The object of this study is to develop a simple and inexpensive device, accessible to anyone, that allows the teleoperation and telepresence of a humanoid robot.

With this, the general concept of developing more and more autonomous robots is challenged since the idea is to create and artificial and mechanical body for the human mind: the technological emphasis is thus transferred from artificial intelligence development to the design of a humanoid robot that fully mimics the desires and movements of the human user.
One might easily think this concept is taking a step backwards in teleoperated robotics, when instead its actually doing the contrary. As of right now, artificial intelligence is nowhere near its full potential; and implementing something which is not fully finished only stresses the limitations it has. It is true that when artificial intelligence finally reaches its required level, teleoperated robots will become obsolete; but until that day comes, having a robot which can't respond to all and any stimuli the environment provides, may very well have catastrophic and dangerous results, specially for the humans involved with it.
As for the telepresence aspect, the benefits of using a human-like robot instead of a car-like one are obvious. Studies suggest that body language (body posture, gestures, handshakes...) and other non-verbal communication represent about 50% of the message we want to convey [22]. This is essential in working environments where huge quantities of money are based upon single meetings and conferences, and where giving a good impression is of utmost importance. Additionally, domestic environments can also benefit from this technologies if they become cheaper and easier to use.

To sum up, this study aims to prove that low cost teleoperation of humanoid robots (and telepresence through them) is feasible, easy to use and can become widely beneficial for society, even though this means going against the general direction robotics has been taking these past few years.

1.3 Research design

The project will be divided into different sections, each one of them dealing with a specific field of robotics. This is done to ensure a coherent development of the

project. These sections are shown below and have specific chapters focussed on them.

- *Kinect motion capture and arm control (Chapter 2):* skeletal detection of user, joint tracking, inverse kinematics, command of arm actuators.
- *Flex sensors and grip control (Chapter 3):* gripper design and laser cut, flex sensor data retrieval and processing, gripper control.
- *Joystick for directional control (Chapter 4):* Joystick data retrieval and processing, defining robot movements, storing them in Flash memory instead of RAM.
- *Oculus Rift for visual feedback and to orient the robots head accordingly (Chapter 5):* Head and neck design and laser cut, integration with unity, transfer of live video footage from robot to *Oculus Rift*, detection of orientation of the headset, command of neck actuators.
- *Voice recognition to enhance directional control (Chapter 6):* Implementation of *Voce* library and set up of gram files, speech recognition and processing, issuing commands to robot.
- *General overview of the project (Chapter 7):* Description of how each individual component fits into the project as a whole and how all the hardware is implemented.
- *Pick and place test (Chapter 8):* Description of several tests conducted to prove the efficiency of the system.
- *Final Conclusions (Chapter 9):* Overall conclusions with respect to the project and further improvements for its real-life application.

1.4 Assumptions, limitations and scope

The design of robots requires knowledge of a lot of engineering fields (mechanical, electrical, control, programming....), thus generally multiple teams need to work on them for a long period of time to produce something decent, resulting in an overall high expense. Since the Master thesis has a very limited time span and no funds/grants are received, certain assumptions must be made.

The intention of building a robot from scratch is certainly unachievable in the time frame provided, but this is not necessarily a drawback. It has been decided to base the project off an existing humanoid robot, and considering human sized

robots are impractical and too expensive, the logical solution was working with a miniaturized humanoid robot. Some of these robots which were taken into consideration were the *Darwin-OP*, *Nao* and *Bioloid* because they are the most renowned on the market and are very accessible.

	Darwin-OP	**Nao**	**Bioloid**
Price ($)	12.000	12.000*	1.200
Height (mm)	454.5	580	189
Weight (kg)	2.9	4.3	1.7
CPU	Intel Atom (1.6 GHz)	Intel Atom (1.6 GHz)	Atmega2561 (16 MHz)
Memory	4GB RAM	1G RAM, 2GB Flash	56KB Flash 8KB SRAM, 4KB EEPROM
Sensors	Accel(x3), Gyro(x3), HD camera, mic(x2)	Accel(x3), Gyro(x3), HD camera (x2), mic(x4)	Gyro(x2), DMS
Connectivity	Wi-Fi, Ethernet	Wi-Fi, Ethernet, Bluetooth, IR	ZigBee, Bluetooth
Actuators	20 MX-28T	21 motors (3 types)	18 AX-12A
Ports	USB2.0 (x2), mini HDMI, Ethernet, mic/audio I/O	USB, Ethernet	Serial

(*) *Note*: the price range of the Nao goes from 4.000 $ to 16.000 $ depending on the amount of motors included.

After a careful evaluation of these 3 robots, its was easily seen that the *Darwin-OP* and *Nao* are quite similar and are way more competitive than the *Bioloid*. In spite of this, it was the latter the one selected because of its cheapness and the multiple enhancement options it had. Since the project aims at creating the most economically competitive teleoperated robot possible, starting with the cheapest robot available makes a lot of sense, specially because the characteristics that make *Darwin-OP* and *Nao* better than the *Bioloid* can be, up to a certain level, replicated at a lower cost. An example of this are the grippers or the neck movements. Therefore, special attention must be given to the fact that the final product/prototype has to be easy to use and fairly low cost, using materials and components that one can easily buy at any electronic store.
The size of memory of the *Bioloid* might become an issue, and if so, must

Figure 1.1: Darwin

Figure 1.2: Nao

Figure 1.3: Bioloid

be dealt with accordingly. As for the connectivity, *Zigbee* communication will suffice, even though it is slower than Wi-Fi and Bluetooth, it is designed for low power consumption and its range may even reach the 50 meter mark, which is more than enough to showcase its usefulness in this project.
Finally, one of the main reasons the *Bioloid* was selected was because it has been very present in academic institutions since its release. Most universities have bought already some of these robots or have easy access to them in some way, which drastically lowers the overall cost of the project.

A small comment needs to be made concerning the *Bioloid* provided. Even though the aim of the project is to eliminate all cables and make every device fully wireless, the lack of a functional battery pack given by the university meant that the robot had to be plugged into a power source continuously. This affected in no way any of the other components and is not at all decisive because if a battery were available, it could be immediately connected and everything would still work the same. In spite of this, it is for this reason that in some images of the report, a black cable can be seen coming out of the robot.

The scope of the project will be transforming a commercial humanoid robot (with certain limitations) to a fully teleoperated robot without spending great amounts of money and the result being an easy to control, comfortable and user friendly device. It is this difference with current robots that makes this project so unique.

Chapter 2

Bioloid arm control through Kinect camera

The first part of the project will consist of the robots' arms movement control through the replication of the position of the users upper body. The idea is that when the operator moves his/her arms in a specific direction, the robot will do exactly the same. This must be done in real-time and with little to no delay, being as precise as possible, to ensure that the robots arm is placed correctly where the user desires it to be.
With arm control, the robot will separate from other robots in the sense that it will be able to do more "humanly" tasks: opening doors, moving objects around (pick and place) or simply conveying messages through body language (gestures), taking teleoperation and telepresence to another level.

2.1 Components needed

The fulfilment of this part of the project will be accomplished through the following hardware and software components:

- *Bioloid robot*: Only the upper body, that is servomotors from 1 to 6, corresponding to the servos for the roll and yaw angles of the shoulder

and the yaw angle for the elbow, in both arms.

- *Kinect camera*: motion sensing input device, designed by Microsoft for the Xbox 360 console. The sensor includes an RGB camera, a microphone array and a depth sensor, which will allow motion capture as described in following sections.
- *Kinect SDK (Software Development Kit) Toolbox 2.0*: grants the developers with the basic software infrastructure to create applications.
- *Processing 3.0a5*: latest version of a this programming language that promotes software literacy within the visual arts.
- *Atmel Studio 6.2*: Integrated development platform (IDP) for developing and debugging microcontroller applications.

The initial idea was to use Inertial Measurement Unit sensors (IMUs) which are a fusion of gyroscopes, accelerometers and sometimes magnetometers to calculate the velocity, orientation and gravitational forces (most smartphones nowadays have one IMU integrated in them to obtain data of the position and orientation of the device). This would have the advantage of allowing the user to be able to freely roam around the room without needing to stay constantly in front of a camera. However, it presents a major disadvantage: the human would have to carry at least two of this devices on each arm and that would be uncomfortable and tedious. Thus, this idea was not considered further.

2.2 Kinect motion capture and joint position retrieval

The first part of this section corresponds to the implementation of skeleton tracking and joint position retrieval thanks to the *Kinect* sensor. The steps followed and algorithms used will be slightly based of the book *Making Things See* [2] and a relevant paper [14]. Additionally, the SimpleOpenNI library [10] provides an excellent wrapper of the *OpenNI* (framework that defines APIs to establish applications that use natural interaction, that is when communication between human and device is done based on the humans senses, normally hearing and sight) and *NITE* (an open source software for Processing), which allows a faster and simpler implementation of specific functionalities for the *Kinect* in Processing.

The program will start by importing the *SimpleOpenNI* library and declaring a global object to access the camera. Then, it is necessary to enable both

the DepthMap image generation provided by the Depth sensor and the skeleton generation for all joints when a user is detected. This is done quite easily thanks to the API provided by the library. Despite this, visualizing the results requires further programming.

A critical issue when receiving the data from the DepthMap and the UserMap (skeleton tracking) is that it must be rotated 180 degrees around the X axis of the *Kinect* frame to be displayed correctly (Figure 2.1). The point cloud (representation of the distance to every point the camera sees) can then be accurately presented, and becomes useful when visualizing the drawn skeleton over the users depth shape (Figure 2.2). To draw the skeleton itself, *SimpleOpenNI* provides various functions such as *getJointPositionSkeleton* (which obtains the 3D position of the specified joint), *drawlimb* (which draws a line between two specified joints) and *drawSkeleton* (which draws all the required limbs), that simplify greatly the task. Additional functions that help visualize the skeleton tracking are: *getBodyDirection* (which draws the centre of mass and its orientation) and *drawJointOrientation* (which draws the orientation of each joint).

Hence, visualizing the skeleton becomes quite easy and retrieving the joint positions and orientation of each and every joint turns into a trivial matter thanks to *SimpleOpenNI*. The full and detailed *Processing* sketch can be seen in Annex E.

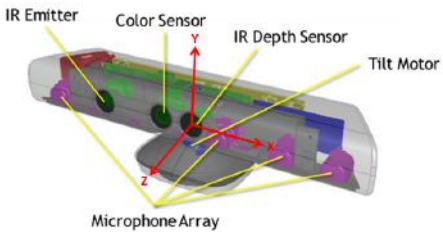


Figure 2.1: *Kinect* camera parts and frame

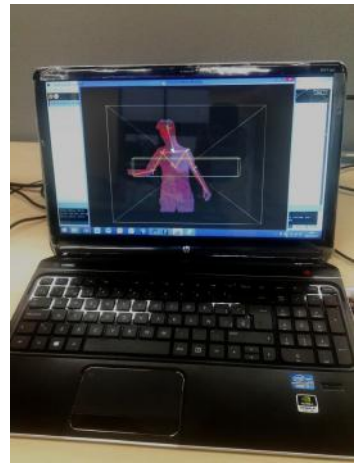


Figure 2.2: Users skeleton tracking

2.3 Inverse kinematics

Once the position of the end effector is successfully retrieved in a 3D space, it is time to transform this data into information the robot can process. This means going from end-effector position to servomotor angles, which is the definition of inverse kinematics. To do so, the Denavit-Hartenberg (DH) parameters (four parameters associated with a particular convention for attaching reference frames to the links of a spatial kinematic chain, or robot manipulator) and forward kinematics must be calculated first, and then the inverse kinematics can be derived [19].

2.3.1 Denavit-Hartenberg parametrization and Forward kinematics

To obtain the DH parameters, one must first apply the DH conventions to describe the axis of all the joints of the robot. Once the parameters are found, deriving the forward kinematics becomes very simple. The laws for a correct orientation of the axis are as follows:

- DH1: The axis x_1 is perpendicular to the axis z_0.
- DH2: The axis x_1 intersects the axis z_0.

Once these laws are applied, the resulting frames of the *Bioloids* arms will look like in Figure 2.3.

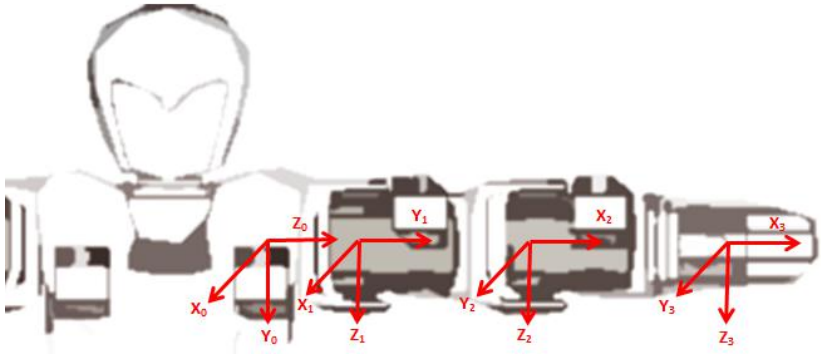


Figure 2.3: Orientation of the frames for each joint using the DH convention.

Note: Frames 0 and 1 are separated for visualization purposes; in reality they share the same origin located on the shoulder joint. Frame 2 resides on the elbow joint and frame 3 on the end-effector (hand). Additionally, the length between frames 1 and 2 and between 2 and 3 is of 70 mm

After establishing the frames, the value of the DH parameters must be found. The set of rules to find the DH parameters are the following:

- Link length a: distance between z_{i-1} and z_i along x_i.
- Link offset d: Distance from O_{i-1} to the intersection of x_i with z_{i-1} along z_{i-1}.
- Link twist α: angle from z_{i-1} to z_i around x_i.
- Joint angle θ: angle from x_{i-1} to x_i around z_{i-1}.

Applying these rules, the DH parameters, for a 3 DoF anthropomorphic arm, are found, shown in Table 2.1.

Joint i	a	d	α	θ
1	0	0	$-\frac{\pi}{2}$	θ_1
2	0.7	0	0	θ_2
3	0.7	0	0	θ_3

Table 2.1: DH parameter values of *Bioloid* arm

Forward kinematics describes the transformation matrix (rotation ad translation) from origin to end-effector. Applying a series of kinematic equations and inputting specific values for the joint parameters, the position of the robots end-effector is derived. This method is greatly simplified once the DH parameters are known. The Denavit-Hartenberg matrix unifies the matrices $Trans_{z_i}(d_i)$, $Rot_{z_i}(\theta_i)$, $Trans_{x_i}(a_{i-1,i})$ and $Rot_{x_i}(a_{i-1,i})$, which are transformation matrices for every parameter from one joint to the next, under one only transformation $(T_{i-1,i})$, which represents both the translation and rotation of a point from the frame of an initial joint to the frame of the following joint.

$$T_{i-1,i} = \begin{bmatrix} cos(\theta_i) & -sin(\theta_i)cos(\alpha_i) & sin(\theta_i)cos(\alpha_i) & \alpha_i cos(\theta_i) \\ sin(\theta_i) & cos(\theta_i)cos(\alpha_i) & -cos(\theta_i)cos(\alpha_i) & \alpha_i sin(\theta_i) \\ 0 & sin(\alpha_i) & cos(\alpha_i) & d_i \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Once the transformation matrices from every joint to the next one are found, the *Homogeneous transformation matrix* (the transformation from origin to end-effector) can be found post multiplying all of them.

$$T^{origin}_{end-effector} = T^0_3 = T^0_1 T^1_2 T^2_3 = \begin{bmatrix} c_{23}c_1 & -s_{23}c_1 & -s_1 & c_1(L_2c_{23} + L_1c_2) \\ c_{23}s_1 & -s_{23}s_1 & -c_1 & s_1(L_2c_{23} + L_1c_2) \\ -s_{23} & -c_{23} & 0 & -L_2s_{23} - L_1s_2 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Note: Te syntax has been simplified for visualization purposes: c_i represents the $cos(\theta_i)$, c_{ij} is the $cos(\theta_i + \theta_j)$ and the same happens with the sinus. Also, L_1 and L_2 are equal and have a value of 70 mm (which are the lengths of the robots upper arm and forearm, respectively)

This matrix represents the rotation (first 3x3 matrix) and the position (last 3x1 column) of the end-effector with respect to the origin's frame, given any arbitrary joint angles.

2.3.2 Inverse kinematics through DH convention and implementation problems

The idea is to find the necessary mathematical equations to do the opposite as before, that is, finding the joint angles for any given position. To do so, the elements of the Homogeneous transformation matrix will be named as $T_{i,j}$.

$$\begin{aligned} T_{1,1} &= c_{23}c_1 \\ T_{2,1} &= c_{23} \\ T_{3,1} &= -s_{23} \\ T_{1,4} &= c_1(L_2c_{23} + L_1c_2) \\ T_{2,4} &= s_1(L_2c_{23} + L_1c_2) \\ T_{3,4} &= -L_2s_{23} - L_1s_2 \end{aligned}$$

Before applying mathematical modifications to these equations, it is worth mentioning the existence of the *atan2(s,c)* function. The purpose of using two arguments instead of one is to gather information on the signs of the inputs in order to return the appropriate quadrant of the computed angle, which is not possible for the single-argument *arctangent* function. The first argument corresponds to the sine part, and the second to the cosine. Additionally, these arguments can be multiplied both by a common element since it will not change the returning result of the function. The process of finding the IK is as follows.

To find θ_1:

$T_{1,4} - L_2 T_{1,1} = c_1 L_1 c_2$
$T_{2,4} - L_2 T_{2,1} = s_1 L_1 c_2$
$\theta_1 = atan2(T_{2,4} - L_2 T_{2,1}, T_{1,4} - L_2 T_{1,1}) = atan2(s_1 L_1 c_2, c_1 L_1 c_2)$

where $c_1 L_1$ is a common multiplier that does not affect the result. Now that the value of θ_1 is known, it can be used to calculate the other angles.

$-(T_{3,4} - L_2 T_{3,1}) = L_1 s_2$
$c_1 T_{1,4} + s_1 T_{2,4} - (c_1 T_{1,4} + s_1 T_{2,4}) = L_1 c_2$
$\theta_2 = atan2(-(T_{3,4} - L_2 T_{3,1}), c_1 T_{1,4} + s_1 T2,4 - (c_1 T_{1,4} + s_1 T_{2,4})) = atan2(L_1 s_2, L_1 c_2)$

And finally, θ_3 is derived.

$c_1 T_{1,1} + s_1 T_{2,1} = (s_1^2 + c_1^2)(c_{23}) = c_{23}$
$-T_{3,1} = s_{23}$
$\theta_{2,3} = atan2(-T_{3,1}, c_1 T_{1,1} + s_1 T_{2,1})$
$\theta_3 = \theta_{2,3} - \theta_2$

And so now, knowing only the location coordinates of the end-effector and the x-component of the orientation, the joint positions can be derived.

Despite this robust methodology, after several tests and posterior evaluation, obtaining the inverse kinematics based off Denavit-Hartenbergs convention was considered inefficient. This is due in great measure to the low confidence values the Kinect calculates when providing the orientation matrix for the wrist. Therefore, a new method to retrieve inverse kinematics had to be found. Available options included Euler angles, using quaternions and basic trigonometry. Finally the latter mechanism was implemented due to its simplicity in a 3 DoF robotic arm.

2.3.3 Inverse kinematics through trigonometry

Before deriving the trigonometric equations, it is important to mention that since the frame of reference will be the shoulders position and not the world reference (as set by the *Kinect*), it is necessary to modify the end-effector's position accordingly. This is accomplished simply with the formula:

$$\vec{P}^{shoulder}_{end-effector} = \vec{P}^{0}_{end-effector} - \vec{P}^{0}_{shoulder}$$

Additionally, and so that the final program is suited for any sized user, the values of L_1 (user's upper arm) and L_2 (user's forearm) will also be calculated.

$$L_1 = \left|\left|\vec{P}^0_{elbow} - \vec{P}^0_{shoulder}\right|\right|$$

$$L_2 = \left|\left|\vec{P}^0_{end-effector} - \vec{P}^0_{elbow}\right|\right|$$

The known data that we have is the location of the end-effector (p_x, p_y and p_z) with reference to the shoulder frame and the values of L_1 and L_2.

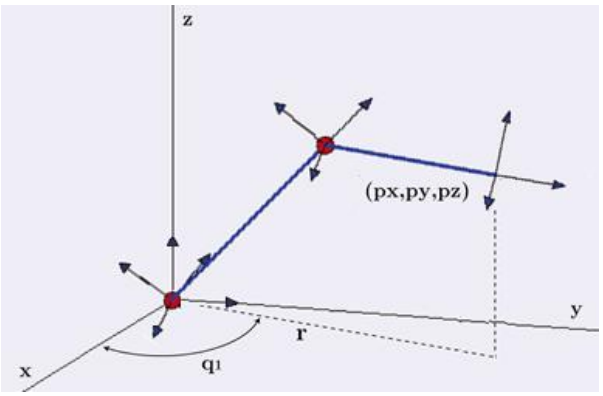


Figure 2.4: Schematic of an arbitrary position of the robotic arm

The value of q_1 is easily found as:

$$q_1 = atan2(p_y, p_x)$$

Knowing that the upper arm and forearm will always be concealed in the same plane, we can apply the cosine theorem to obtain q_3.

$$r^2 = p_x^2 + p_y^2$$

$$r^2 + p_z^2 = L_1^2 + L_2^2 + 2L_1L_2cos(q_3)$$

$$Rearranging: \qquad cos(q_3) = \frac{p_x^2 + p_y^2 + p_z^2 - L_1^2 - L_2^2}{2L_1L_2}$$

$$Therefore: \qquad q_3 = atan2(\pm\sqrt{1 - cos(q_3)^2}, cos(q_3))$$

The $\pm$ symbol when calculating the value of q_3 represents the two options of reaching a specific point; these are known as elbow up or elbow down. The positive value is chosen because it creates more space between the robots body and its arm, hence the collision probabilities decrease drastically.

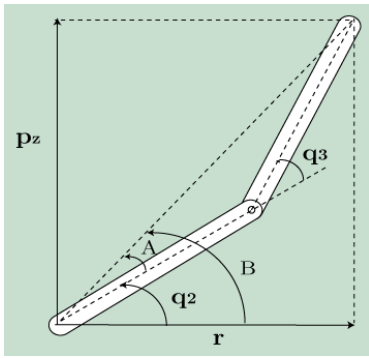


Figure 2.5: Elbow Up

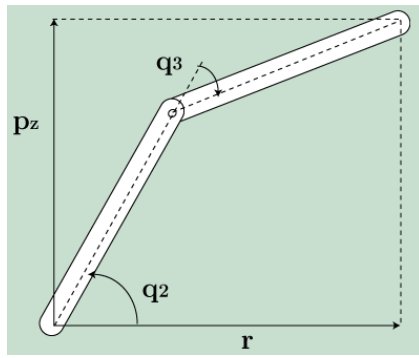


Figure 2.6: Elbow Down

Lastly, the derivation of q_2 will be accomplished as $q_2 = B - A$.

$$B = atan2(p_z, r) = atan2(p_z, \pm\sqrt{p_x^2 + p_x^2})$$
$$A = atan2(L_2 sin(q_3), L_1 + L_2 cos(q_3))$$
$$Hence,$$
$$q_2 = atan2(p_z, \pm\sqrt{p_x^2 + p_x^2}) - atan2(L_2 sin(q_3), L_1 + L_2 cos(q_3))$$

After further testing, it was apparent that a generalised solution would not work and so to select which $\pm$ sign is needed to calculate q_2, the value of p_x (X value of the end-effectors position) must be checked. If $p_x > 0$, the positive solution will be implemented, and vice versa. This is shown in Appendix E.

2.4 Data transfer through serial

Now that the values of the joint positions have been calculated through inverse kinematics, it is time to send this information from the *Processing* sketch. onto the *CM-510* micro controller of the robot. However an important issue arises when doing so. One same sketch is not able to detect when a COM Port is available if the Kinect camera is also connected to the computer. Therefore, two sketches must be run simultaneously: the former will obtain the joint positions of the user, calculate the IK and send the information to the latter, which will in turn rebuild the data into small packages, easier to send through serial, that the robot will detect and process.
The communication between sketches is easily done with the libraries *oscP5* and *netP5* [17], setting up the right ports and locations between sketches and

creating an adequate message with all the data to share. A detailed description of how it was set up and implemented can be found in Appendix E, F and G. This *Processing* library deals with the *Open Sound Control* (*OSC*) protocol. It is was originally used for networking sound synthesizers, computers, and other multimedia devices for purposes such as musical performance or show control. In spite of this, many applications of this protocols have emerged, in particular the library created for *Processing*, that supports TCP, UDP and Multicast network protocols. It is ideal for sending and receiving messages between *Processing* sketches, which is the objective aimed for.

2.4.1 Data package creation

The joint positional data received will be in degrees. Thus, some slight modifications must be made. All angles must have the initial position of the servo added to them (the reference we consider to be 0) and then the format must be changed from degrees to positional values (0-1023).
The most crucial part of this section is creating a well-defined package to be sent through serial that can be understood easily by the robot. To do so, the information will be stored as strings, which the robot will later decompose. The format of these strings will be as follows:

$$X_{id}\ X_1X_2X_3X_4$$

Where X_{id} tells the ID of the servo to be moved, and $X_{1...4}$ represents the position to which the servo has to move. The spacing between both numbers is essential since it will help the robot part the message into two integers as will be seen in the next section. Also, the *OSC* library automatically adds a new line character at the end of every message sent.
To send this packages through serial, importing the *processing.serial* library [7] is necessary. This sets up the serial connection once the object for the serial class, the port and baudrate have been defined. The *Bioloid* manual [16] recommends a baudrate of 57600 bps so that was the value selected. The hardware connexion is quite simple: *Robotis* provides a device called *USB2Dynamixel* that, when connected to the serial cable of the robot, allows an immediate communication between USB port and the microcontroller *CM-510* mounted on the robot.

2.5 Bioloid program

The microcontroller of the robot *CM-510* (Figure 7.10) is based off an *ATMega2561*, programmed in *C/C++* and compiled using *Atmel Studio 6.2*.

Now that the message has been sent through serial, the robot must be able to process it and send the positional information to the arm joints. The string received will be partitioned in two where the blank space between both numbers acts as delimiter. Both strings will then be cast into integers with the *atoi* function. Finally, the information will be sent to the corresponding servo through *dxl_write_word(ID, 30, Pos)*. Where the value 30 tells the servo it will be receiving a goal position and not an angular velocity. The result of this process is positioning the arms of the robot exactly as those of the human, via *Kinects* joint position retrieval and inverse kinematics.

2.6 Wireless

To make a humanoid robot truly teleoperable, wireless communication between robot and user is a must. Thus, removing the serial cable previously used to transfer the joint positional data from PC to robot is necessary, and needs to be substituted by a wireless device. The company *Robotis* provides the devices Zig 100 and Zig-110A, which use *Zigbee* communication, making serial communication (*UART*) possible. The former ()Zig-100), will be connected to the PC using the devices *USB2Dynamixel* (shown in Figure 2.7) and the *Zig2Serial* (apparatus that connects the USB2Dynamixel to the Zig-100, shown in Figure 2.8).

Figure 2.7: USB2Dynamixel devices provided by *Robotis*

Before any communication using *Zigbee* can be done, each device must have its own ID and the ID to whom it is communicating to, set correctly. The own ID (also referred to *MyRemote ID*) is non-variable and is shown at the back of the device. The only thing left to change is the value of *Remote ID* which must be the same as the *MyRemote ID* of the device to which you are communicating to. This process is done thanks to the *RoboPlus Manager* software provided by *Robotis*.

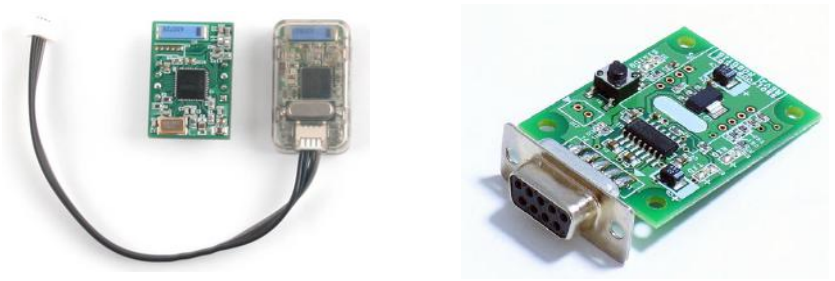


Figure 2.8: Zig-100, Zig-110A and Zig2Serial devices, respectively, provided by *Robotis*

However, this system has a significant problem. The *Zigbee* communication has been altered by the company in the sense that the firmware of the robot can only receive 8 bit integers. This is a big issue due to the fact that the PC is sending strings (which include, as mentioned before, the ID of the servo to be moved, followed by a space and the positional value). For this reason, the firmware of the robot needs to be modified so that every time there is a USART_RX event (a new integer is received from the *Zigbee* communication) the characters are stored in a buffer until the character for a new line $\backslash n$ (value 10 in ASCII) is detected, at which point a flag (*receive_ready*) will be set, telling the robot a new string is ready to be processed. The values of the buffer (up until the value for new line) will then be saved into a char array which will be processed exactly the same way as if the communication were done through serial. A detailed description of this system can be seen in the function *GetData()*, displayed in Appendix K.2.

2.7 Results and evaluation

After correctly setting up the equipment and running the software, some tests were made to evaluate the success of the application. Figure 2.9 shows various images depicting how the robots arms mime those of the user. The results are very conclusive: the arms reposition themselves with little to no delay, achieving a correct location. Despite this, the end-effector position is not optimal due to the disturbances and the level of precision the *Kinect* camera has. Additionally, the inverse kinematic equations fail in very specific areas (these disturbances are so small that the effect is almost insignificant). Further improvements would be a more optimal positioning of the end-effector thanks to a more developed mathematical model. A feedback control loop could be implemented to achieve this, however creating a communication from robot back to computer would only

delay the operation and its benefits might not be as significant as expected since the position is updated almost instantaneously with huge differences between one reference and the next.

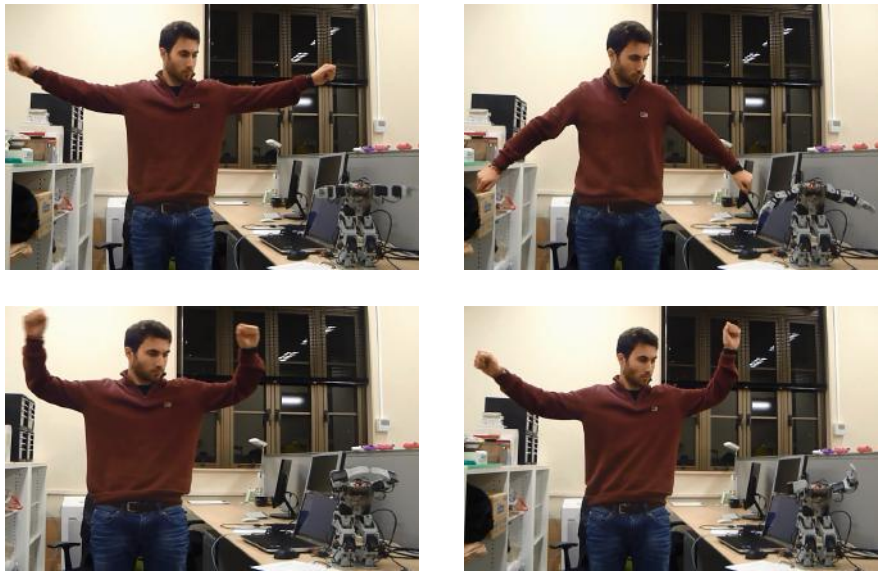


Figure 2.9: Images of one of the tests

Lastly, a very important issue with this system is the fact that the arms have no collision avoidance system and they do, at times, crash into each other or into the body of the robots itself. To solve this, the movements of the robot could be first simulated (for instance using a Python environment) and if no collision was found, then the robot would actually be allowed to reproduce them [15]. A simple way to slightly avoid this problem is by adding a small control when calculating the new arms position. This is done by taking the old value, multiplied by a fraction, and added to the new value, also multiplied by the complementary fraction. More information on the subject can be found in Section 8.3.2. This becomes essential in the final stages of the project, specifically in the *Pick and Place* test (Section 8.1.4), which will be explained later, where one must position the robots arms very precisely. Having a very fast system that occasionally overshoots the goal position is of no use. Therefore, combining both old and new values (in a certain proportion) leads to a slower and more precise response, thus achieving better results.

2.8 Conclusion

Several Lab peers where asked to test this section of the project. It was immediately seen that the learning curve was very steep. Users had, initially, a very difficult time getting to know the sensitivity of the device and realising how their gestures affected the final placement of the robots arm. But as time past, the control expertise grew significantly to a point where very precise movements could be achieved. Those users that only spent a few minutes on the technology ended up with the sensation that it did not work properly. On the other hand, those who worked with it for around half an hour managed to become fairly proficient with its use.

Another highly competitive system for skeleton tracking would have been the highly expensive *Motion capture* suits [20], used generally for film animations and CGI (Computer Generated Imagery). Even though their effectiveness and usefulness have been proven over and over again, their high expense made them inaccessible for this project.

Overall, as with any newly tested technology, time needs to be allocated to achieve the necessary expertise and dexterity to control it correctly. Even though a proper feedback positional loop and an collision avoidance system could have been implemented if more resources were available, the results for this section were quite reasonable.

Chapter 3

Bioloid Gripper control

For this part of the project, a robotic gripper will be designed and laser cut. It will be a 1 DoF hand, thus controlled only by one servo motor, which will close and open as a function of the users hand movement. To achieve this, a flex sensor will be attached to each of the humans hands and the information will be sent to a *Arduino UNO* chip which will in turn convey the signal wirelessly to a secondary *Arduino UNO* located on the robot itself, that will control the grippers servo motors. The decision to make it a 1 DoF gripper and not 2 DoF (adding a possible rotation motor) was taken because the *Kinect* is not able to calculate precisely the orientation of the wrist (as mentioned in the previous section) and thus it would generate more problems than solutions.

3.1 Components needed

The following hardware and software components will be essential to the fulfilment of this task:

- *Arduino UNO*: A microcontroller board based on the *ATmega328*. It has 14 digital input/output pins (of which 6 can be used as PWM outputs),

6 analog inputs, a 16 MHz ceramic resonator, a USB connection, a power jack, an ICSP header and a reset button.

- *Arduino Wireless Shield*: Allows the *Arduino* board to communicate wirelessly using *Xbee*.
- *Xbee S1*: Also known as *Xbee* 802.15.4 module; it is an embedded solution that provides wireless end-point connectivity to devices. This module uses the IEEE 802.15.4 networking protocol for fast point-to-multipoint or peer-to-peer networking.
- *Flex Sensor*: Sensor that increases its resistance when it is flexed.
- *Arduino programming language*: It is an implementation of Wiring, a similar physical computing platform, which is based on the *Processing* multimedia programming environment. Published as an open source tool, thus extended by experienced programmers, its libraries are well-defined and easy to use, making the process of coding very straightforward.
- *RB-90 mini servo motors:* A small (22.5 x 11.5 x 27 mm^3) and weightless (9 g) servo motor, with a required power supply of 4.8 V that provides 0.12 s/ 60 degrees of speed and 1.6 kg-cm of torque.

Since the gripper will consist of only two fingers that open and close with a single motor, only one flex sensor per hand will be used to control the respective servos. Additionally, the selection of an *Arduino UNO* board over others like the *LaunchPad* or the *Teensy 2.0*, was decided due to the vast resources available for it and the easy access the university had to them. This includes also the various components compatible with the board, for instance the *Wireless Shield* and *Xbee* modules. Even though this device has its limitations, like the lack of memory, they are totally negligible for the scope of this project.

3.2 Flex Sensor data retrieval

The first part of this small project is obtaining the information from the flex sensor (*Sparksfun* 4.5" sensor, as seen in Figure 3.1) and processing it. To do so, firstly a small 22 kΩ resistor (fixed resistance that can be used for comparison) must be soldered to the flex sensor (as seen in Figure 7.2) to create a voltage divider. Now, the analog read on the board is basically a voltage meter: at 5V (the maximum) it would read 1023, and at 0v it would read 0. Thus, the amount of that 5V that each part gets is proportional to its resistance. So if the flex sensor and the resistor have the same resistance, the 5V is split evenly

(2.5V) to each part.
Later, a small *Arduino* sketch will be written in which the analog ports connected to the sensors are set. Then, the information is read through the *Analog.Read(Pin)* command and it is processed by the *map(value, fromLow, fromHigh, toLow, toHigh)* function that re-maps a value from one range to another (this is because the servo motors can only turn 180°). Finally, the data is sent through the serial port with the command *Serial.print(Variable)*. It is important to mention that the sent message will be written as a string. For the flex sensor of the right hand, the string will be: "FSR 'Value' \n"; and similarly, for the flex sensor located on the left hand, the string will be: "FSL 'Value' \n". This is done because in following sections, many messages will be sent across the Arduino-Pc-Robot network (for instance the Joystick values used later) and using strings helps understand what is sent and where. A detailed description of the code is found in Appendix H.

Figure 3.1: *Sparksfun* 4.5" Flex Sensor

Figure 3.2: Xbee 802.15.4 (S1) module connected to Breakboard and mini USB

3.3 Xbee 802.15.4 module setup

The *Xbee module* (Figure 3.2), together with the *Arduino Wireless Shield*, allows an easy out-of-the-box wireless solution between *Arduinos* or between one *Arduino* and the PC. In this case, the messages need to be sent from one *Arduino*, situated on the user, that will read the hands movements, to a secondary board, located on the robot, that will control the grippers servo motors.
The *Xbee* modules used are Series 1, which means that, even though being very similar to the Series 2 modules (they both have ave the same footprint and for

the most part are pin for pin compatible), they do not provide mesh networking capabilities (allows the signal to be relayed through multiple nodes before it gets to your final destination). However, for this project, this will not be a requirement and, all in all, Series 1 is used for point to point or star topology (which is what is required) and are significantly faster and cheaper than the latter modules.

To correctly set up the parameters of the *Xbee*, the configuration platform *X-CTU*, developed by *Digi* (the same company that created the *Xbees*) will be used. Once the *Xbee* is connected to the PC via a *Breakout Board* and mini USB cable (as seen in Figure 3.2), the software immediately recognizes the *Xbee* and allows the user an easy parameter modification thanks to its simple GUI.
The parameters that need to be fine-tuned are the following:

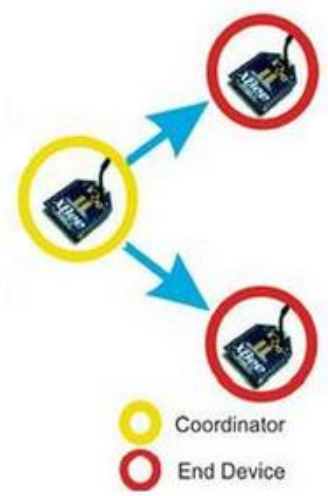


Figure 3.3: Star Topology for Xbee S1 (802.15.4 module)

- *Network* (**ID**): Both *Xbees* must share the same network.
- *Channel* (**CH**): They also have to have a common channel.
- *Modules own address* (**MY**): The 16-bit address of the module.
- *Modules destination address* (**DH**/**DL**): Determines which modules on its network and channel will receive the data the *Xbee* transmits. For a specific configuration where **DH** is zero (high 32 bits of destination address), then the data transmitted will be received by any module with a **MY** parameter equal to **DL** (low 32 bits of destination address).

Out-of-the-box *Xbees* have all the same parameters: **ID** 3332, **CH** 0, **DH** 0, **DL** 1 and **MY** 0. And so the only thing that must be changed to allow two-way communication between both of them is modifying the receiver *Xbee*'s **DL** and **MY** parameters to 0 and 1 respectively. Note that if two *Xbees* are modified to act as a receiver, while having a third one acting as a sender, a Start topology is obtained, as seen in Figure 3.3. Lastly, the biggest advantage of using a *Wireless Xbee Shield* and an *Xbee* module on top of the *Arduino UNO* board, is

that it allows a wireless communication using just the standard *Arduino* serial commands. Therefore, no modification to the previous code is needed.

3.4 Servo motor control

Once the flex sensor data is sent wirelessly from the sender *Xbee*, it is time to allow the second *Arduino* to receive the data and act accordingly. The library *Servo.h* provides an excellent resource for dealing with servo motors [1]. Firstly, two servo objects must be created (*Servo ServoName*), corresponding to the right and left gripper motor, and their specific digital pins must be selected through the command *ServoName.attach(Pin)*.
As mentioned before, the reading of wireless data is done simply with the command for reading the serial. However, the information needs to be processed a bit since it comes in string form. To do so, the info is read until a new line is found (*Serial.readStringUntil(10)*), where 10 corresponds to a new line in the ASCII table. This string is then divided into two substrings, the former called *First* coincides with the first 5 characters of the string (*First=Data.substring(0, 4)*) and will be either "FSR: " or "FSL: " (representing Flex sensor right or left, respectively); whilst the latter, called *Second*, is the value in string data type of the flex sensor (*Second = Data.substring(4)*). This command reads the data from the fifth position until a new line character is found.
The code basically checks the string *First* to see if it contains flex sensor data, and if so, converts the *Second* data into a char array and then casts it into an integer so that it can be easily sent to the servo through the command *ServoName.write(value)*. A detailed description of the program can be found in Appendix I.

3.5 Gripper design and laser cutting

The original idea was to 3D print a 1 DoF robotic hand made out of 3 fingers to provide a better grip of whatever object needed to be grasped. However, due to the fact that no 3D printer was available at *Tokyo University*, it was decided to laser cut the gripper from a 4 mm wood plank, thus limiting the possible forms of it. Basically, all the parts that built the gripper could now be only in 2D since the laser cutter produces pieces in that way. And so the design of the robotic hand had to be modified to a pincer-like gripper.

To simplify the workload, a schematic of a gripper was downloaded from *Thingiverse* and was later modified to suit the *Bioloid* using *SolidWorks*. It consisted

of two gears (one connected to the motor and the other simply following the former) which in turn were attached to two fingers. Due to the configuration, these fingers always stayed parallel to each other, allowing an easier and more precise grip due to a wider contact surface. The rendered result is shown in Figure 3.4 and the drawings for the full assembly and its individual parts can be found in Annex A.

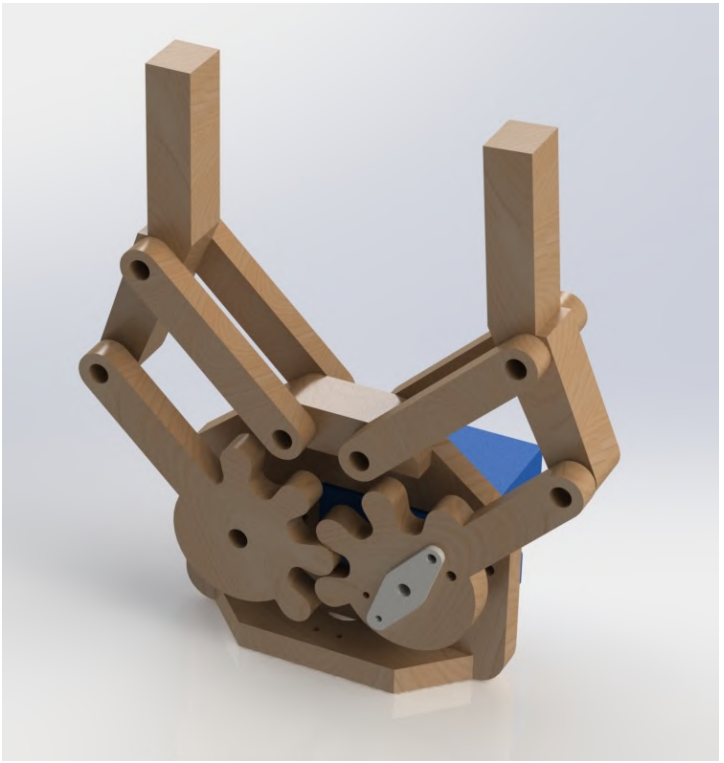


Figure 3.4: Render of the designed 1 DoF Gripper

3.5.1 Laser cutting procedure

Once the pieces were fully designed and tested through the *SolidWorks* assembly program, the files had to be saved as .DXF files and uploaded to the computer connected to the laser cutting machine, called *Commax Laser System Value Direct 7050-60W* [4] (Figure 3.5). The schematics for all pieces were copied into one *CorelDRAW* file and sent to the laser cutter.
The procedure for operating the machine was not disclosed and could only be

used in the presence of someone who knew how it worked. From observation and further investigation it was seen that laser cutters work by directing a very powerful laser beam, at a precise focal length, onto a material which they either cut or etch, operating very much like a printer.

Figure 3.5: Laser cutting machine cutting pieces

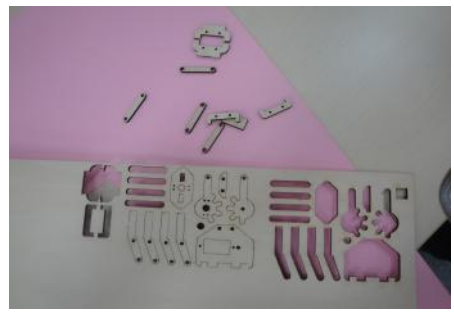


Figure 3.6: Resulting pieces after laser cutting

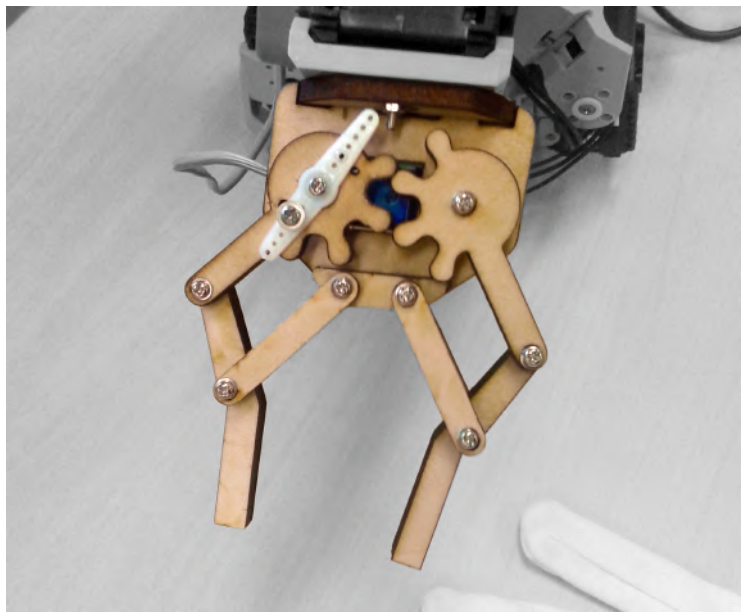


Figure 3.7: Final result: 1 DoF gripper

Two settings are very important depending on the precision and material one wants to cut: power and speed. The former controls how much power will be applied to the laser while printing. The more power - the more heat, and the more heat - the greater the chance of fire. The latter determines how fast the

laser will travel while cutting. The slower the speeds, the longer the laser sits in each spot, which yields more heat. It also means that the slower the speed, the deeper the cut or engraving will be.

After continuous testing, it was determined that a speed of 35 mm/s and a power of 85% for this specific machine would suffice and give fairly good results. The final result is displayed in Figure 3.7.

3.5.2 Servo motor and *Arduino* board power supply

Figure 3.8: RB-90 mini servo motors

To operate the gripper, a servo motor needs to be connected to the *Arduino*. The *RB-90 mini servos* (Figure 3.8) are very adequate because of their low weigh and small dimensions, even though the torque it provides is not that great. These always have 3 connexions: a digital signal that represents the position commanded, a power supply (in this case 4.8V) and a ground. Considering that the *Arduino* board outputs 5V, the servo motor could be connected easily to it. Nonetheless, the current produced by the board is only capable of handling 2 servo motors simultaneously, at best, because it overheats the power regulator on the *Arduino* board, which in turn causes it to fail. Considering that 2 servo motors will be used for the hands and 2 others will be used for the pitch and yaw movements of the neck (showed in later sections), it is necessary to find an external power supply for all servos.

Figure 3.9: 9V battery clip soldered to a 2.1 mm plug

Figure 3.10: Connections between 4 servo motors and *Arduino* board

A set of 4 AA Ni-MH rechargeable batteries produces 1.25 V each battery and therefore a total of 5V, perfect for use with these servos. Additionally, the current generated is also sufficient to move all 4 servo motors since the power source of the board and of the motors will be different. To power the board itself a 9V PP3 battery will be used because of its compact size and simplicity of use, even though there is a great loss of efficiency (around 40%) in the linear regulator to change the 9V input into the 5V needed by the *Arduino*. To connect it to the board, a 9V battery clip needs to be soldered onto a 2.1 mm power plug, the result is seen in Figure 3.9.

Figure 3.10 depicts the connections between al 4 servo motors and the *Arduino* board on the back of the robot. The 5V power supply from the 4 AA rechargeable battery pack comes in the red cable and the ground is the green cable. All 4 servo motors are connected in parallel. The ground of the servo motors is also connected to the *Arduinos* ground through the blue cable. If this connection was not made, the digital signal going from the *Arduinos* digital pins to each servo motor through the white wire would not function. These electrical connections can be better observed in Figure 3.11.

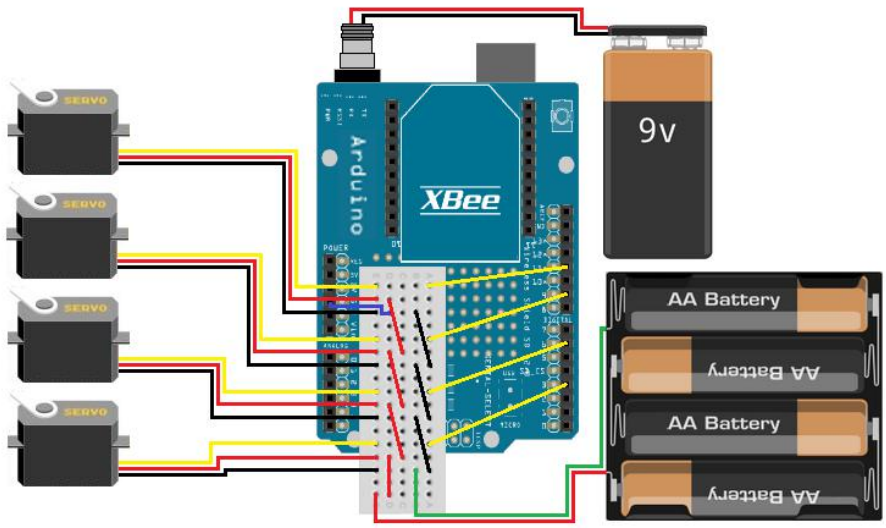


Figure 3.11: Schematic of *Arduino UNO* board on the robot

3.6 Results and evaluation

The whole procedure mentioned above results in a robot gripper closing and opening as the human user closes and opens his/her fingers. All of this is done wirelessly through the *Xbee* communication and the delay is insignificant. Additionally, it was observed that increasing or decreasing the resolution of the flex sensor data would allow for a different precision of the gripper and that this could be beneficial depending on the kind of object to be grasped.
Nonetheless, when testing the gripper further, it became evident that not all cases worked correctly. The robot easily picked up objects that were small and soft (sponges or Styrofoam items), but struggled with other types of materials, specially if the contact area was not adequate because of it being too big, hard or slippery. Also, mechanically speaking, the gripper was not at all robust due to screws often coming loose and wooden parts breaking because of fatigue.

3.7 Conclusion

All in all, the problems aforementioned and the fact that the gripper did not have rotation entails the problem that some pieces could not be grasped because they were inaccessible (as detailed in section 8.3.4). Despite this, the gripper did serve it purpose, but its improvement is more than obvious.
For real-life applications, a precise measurement of the positions of each finger (maybe using *LeapMotion* [12]) and a robust and flexible robotics hand (not gripper) would be much more adequate for picking or grabbing objects and for a better transmission of information through gestures, at a higher expense, of course.

Chapter 4

Robot directional movement control through joystick

This part of the project will use the positional information of two joysticks located on the users hands to determine where the robot has to move and how. The joystick on the right hand will be used to tell the robot that it needs to move forwards, backwards, strafe left or right. The joystick on the left hand will simply be used to turn the robot right or left, on the spot.

When the robot moves, the problem of balancing arises. The *Bioloid* is very unstable and it does not have any control algorithms to prevent falling down. It does however have a gyroscope to know if it has fallen (forwards or backwards) and a series of movements to get up if it happens. To solve the possible problems of falling, and after several tests, it was determined that when the robot moved, its arms were to remain at the side of the body, without moving or mimicking the user, to lower the centre of mass and thus add more stability.

4.1 Components needed

For this section, several accessories from previous projects will be used, such as the *Arduino* boards, *Xbees* and *Processing* sketches. The only new component needed is:

- *Sparksfun Joystick*: Made up of two potentiometers obtaining positional values in the X and Y directions.

Other methods to control the *Bioloids* walking were devised such as using the *Kinect* sensor to measure the legs position and transfer it to the robot or using voice activated commands.
The former quickly proved to be inefficient and counter-intuitive compared to the joysticks because the distribution of mass is different from human to robot. Thus, a movement that is stable for a certain human is almost always not suitable for the robot. Also, the human has automatic balancing controls which the robot has not. The latter will be discussed in coming sections of the project.

4.2 Obtaining joystick data

The Joysticks used will be provided by *Sparksfun* (Figure 4.1) and consists of two 10k Ω potentiometers, where one will read the X-axis value and the other one the Y-axis value.

Similarly to the process of obtaining flex sensor data, the analog pins must be specified and the command *analogRead(Pin)* is used. Each joystick has two potentiometers, thus a total of 4 analog port pins will be needed. The main difference is that now the values have to be treated slightly. A small function is created (*treatValue(int data)*) that transforms the sensor's measurements into a value between 0 and 15. This is done to simplify posterior calculations when figuring out the direction the joystick heads towards. The resulting values will have a minimum of 63, a maximum of 48 and a zero-state (when the joystick is not pushed in any direction) of 56.

Figure 4.1: *Sparksfun* Joystick

The data is sent through serial and, since the *Xbee Wireless shield* and the *Xbee* modules are already up and running, it will automatically send the information wirelessly. The messages sent will be similar to the flex sensors ones where the first 5 characters of the string will be "J1X: " for the first Joystick measuring the X-axis value and "J1Y: ", "J2X: " and "J2Y: " respectively for the rest of the potentiometers, followed by the value obtained. A detailed description of the program can be found in Appendix H.

4.3 Xbee to PC communication

To transmit the joystick information to the robot, the data must flow from the *Arduino* on the user, to a computer that processes it and sends it via *Zigbee* onto the robot. As mentioned earlier, the sender *Xbee* broadcasts all the data to all receiver *Xbees*. In this case, to communicate the *Xbee* that sends the sensor information (both flex and joystick) to the PC, a third *Xbee* must be set up as receiver, thus completing a Star Topology as seen in Figure 3.3. Said *Xbee* is connected to the PC via a breadboard and a mini USB cable (Figure 3.2). As always, the wireless data can be read as if it were through serial.
To do so, a third and final *Processing* Sketch is coded. The idea is quite similar to the *Arduino* code for servo control (Appendix I). The data received is in String type and must be partitioned for a better understanding of the information it conveys. What this program does is divide the strings to determine the sensor to which the header makes reference to and then store the values in corresponding variables. This means that if a received string is of the form "J2Y: 50", then a variable with the name "J2Y" will have the integer number 50 assigned to it. It is a simple process but essential nonetheless for following steps.

4.4 Joystick data processing and sending to robot

Now that the variables and values have been correctly stored, comes the tricky part. As mentioned in earlier chapters, the *oscP5 Processing* library allows for data communication between *Processing* sketches. This will be useful once again to send the joystick information to the *Kinect Processing* sketch, shown in Appendix E, which will in turn send it to the *Processing* sketch that send data via *Zigbee* communication.

The question logically emerges of why not send it directly to the second sketch. The reason is that this last sketch expects to receive integer values in an or-

derly way and from only one source. If it were to receive different messages, at different times from two different sources, the program might end up assigning information to wrong variables and thus nothing would work.
For this reason, a set of 4 integer values corresponding to the 4 potentiometers on the 2 joysticks is initially sent to the First *Processing* sketch (*Kinect and IK*). This can be better understood through the UML Deployment Diagram (Figure 7.1). This program in turn, has an OSC event triggered by the reception of the set of joystick values which constantly updates a series of global variables. Once the inverse kinematics are derived, as seen in the corresponding chapter, a package of messages is sent to the *Processing* sketch in charge of the *Zigbee* communication.

Now that all the information ha been gathered by the last *Processing* sketch (the one that sends data to the robot), the data will be processed and acted upon accordingly. As mentioned earlier, the *Bioloid* robot is extremely unstable. Hence, for a correct directional movement, all servos must follow exactly a precise pattern provided by the company *Robotis* (developers of the robot itself), specific to each and every movement; and these can be seen in the *RoboPlus Motion* software also created by the same company. This means that the robot will not be able to move in a certain direction if its arms are all over the place trying to mimic those of the user, without falling to the ground, as mentioned before. This is why, when the joysticks are moved by the user in a direction, the joint positional data must not be sent to the robot; and vice versa, if the joysticks are not pressed, only inverse kinematic information should be sent to the *Bioloid*. And this is precisely what the program does, it checks the joysticks values and determines whether they are in zero-state or not. If so, it will create strings of joint positional data and send it to the robot as it had always done. However, if the joystick values are different from the zero-state, then it must compute which joystick is pointing at which direction.

Knowing that there are 4 available directional movements that the robot can do, plus two additional turning on spot movements, the program must be able to differentiate all of them. In the case of the right joystick, the one that controls the directional movements of the robot, the orientation of the joystick will be divided into 8 areas depending on the X and Y values of the potentiometers. For instance, if the X-axis value is positive and the Y-axis value is zero, then the robot will strafe right. Similarly, if the Y-axis value is negative and the X-axis value is zero, then it will move forward. Additionally, if both X and Y axis values are different from zero, the robot will not move, this is to avoid possible data processing errors or that the joystick is held incorrectly. Likewise, for the left hand joystick, there will be two possibilities: if the X-axis value is positive, then the robot will turn right on the spot, and if the value is negative, it will turn left. A clear example of the aforementioned is found in Figure 4.2.

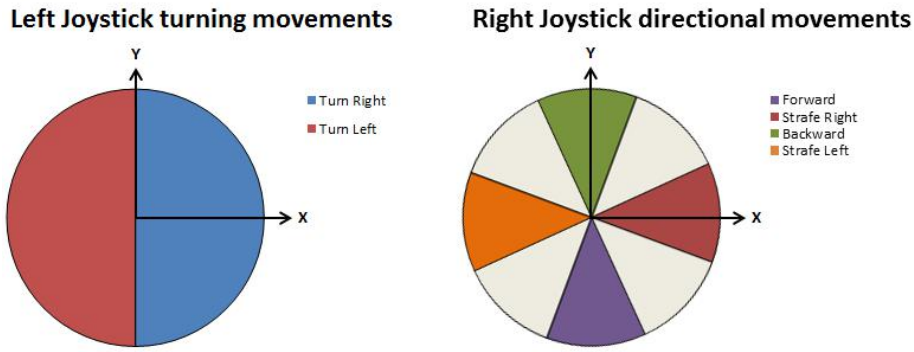


Figure 4.2: Concept of right and left joystick values and their respective movements. *Note:* Blank spaces signify that no movement is done for those values.

The strings sent to the robot via *Zigbee* with the information will be as follows:

9 ***Direction***

The 9 value is a simple way of knowing that the information must not be confused with the arms positional values, it is simply a selected reference value (for arm control, this value ranges from 1 to 6 as a function of which servo to move). When the software of the robot decomposes the string, it will easily see that the data provided will be for the leg movements and not the arms. The field "Direction" will include a 4 character long description of the movement to be made. For instance, moving Forwards will be referenced with the string "WFOR" and turning left with the string "TRNL", and so on. Between the number 9 and the directional word there must be a blank space, which will be used (like in previous programs) to decompose the string into two parts. More information on this specific code can be found in Annex K.3.

Finally, for a detailed description of how this mechanism was physically implemented, please refer to chapter 7

4.5 Storing and use of Motion Pages

Once the robot has understood that a lower body movement is required it is time to execute a series of commands that allow the servos to be set to specific positions, in a certain order and in a determined amount of time, with the goal of moving the robot in the desired direction.

The *RoboPlus* software has a series of motion pages that serve this exact purpose, and are very easily used if the robot still had its original firmware. However, now that the firmware used has been modified, one cannot access those motion files that easily. A simple way to solve this would be saving all the servo positions for all possible movements in a file that could be uploaded to the robot, and when a motion is required, simply select the values accordingly. Nonetheless, a memory problem appears. The robot is not capable of storing such a big quantity of data in its RAM and so the upload is not permitted.

Peter Lanius developed in 2013 a new Bioloid firmware (called *BioloidCControl*) [13] that replaces the original and makes it open source. Among the diverse characteristics of this software, it supports:

- Executing motions based on a RoboPlus Motion file.
- Walking and shifting between walk commands.
- Gyro and DMS sensor and the other ADC channels.
- Serial communication via cable or ZigBee and Zig2Serial.
- Basic control loop and finite state machine for executing motions without waiting for completion.
- Static balancing using gyro and accelerometer (based on Extended Kalman Filter).

It is obviously a very complete program. However many of the characteristics are not specially relevant to this particular thesis right now and so only a small part of the code is of use. Specifically the idea of storing the different motion pages in the Flash memory (PROGMEM) to minimize the weight of the code. Hence, applying some of the concepts developed by this program, a series of functions were created. The most important ones are *motionPageInit()* (Annex K.4) which initializes the motion pages by creating a table of pointers to each page, *unpackMotion(int StartPage)* (Annex K.5) which transfers the data in the Page stored in the program memory to a *struct* in the RAM for execution and *executeMotion(int StartPage)* (Annex K.6) which executes a single motion page. The motion page not only includes positional data, but also joint flexibility, time to reach a pose, pauses, and sometimes even loops, as seen in the code snippet in Annex K.7.

4.6 Results and evaluation

Several tests of the robot moving on different surfaces and with different arm movement velocities were done to measure stability and correct functioning of the robot, as mentioned in section 8.1.3.

The immediate extracted result is that once the robot starts doing a leg movement, all upper body motions must be disabled. This is because the motion Pages require specific positions of the arms to keep balance, which is very important due to the high instability of the robot. If this is not done correctly, and the arms are set to move freely mimicking the users movements, the fail rate increases drastically. Of all the taken tests, the robot fell to the ground in all cases, independently of the surface or the velocity at which the arms moved. To solve this, a control system could be designed which would use a sensor fusion of accelerometer and gyro data (IMU) to determine the general position and orientation of the robot and modify certain servos values accordingly, both of the legs and arms, with the objective of not falling over. This could also solve the problem the robot faces when stepping on an obstacle or tripping over an object. However coding this would be very time consuming and would make a project by itself. Furthermore, it would not solve completely all the limitations of the robot and would probably do more harm than good in an automaton as simple as the *Bioloid*.

After solving this issue (setting the arms to specific locations when moving the legs), it was observed that whilst the robot is moving in a certain direction it is still receiving arm positional data (even if it ignores it) via *Zigbee*. When the movement has finished, the robot will try to access that data from the buffer, working as a FIFO (First In, First Out) system. Since the majority of information is from past user movements, it makes no sense to reproduce it now. Therefore, the buffer must be flushed every time a leg movement has been finished so as to store only the current arm positional information.

Lastly, the tests also demonstrated that the defined movements of the *Bioloid* do not work for every surface (section 8.3.3). Thus, the different parameters for every movement (such as servo positions, flexibility and waiting time) had to be fine tuned. This was a very tedious and long work, and the resulting new movements only worked for one surface (in this case a carpet floor). If the robot were to walk on other surfaces, the results would not be as good.

4.7 Conclusion

Considering the multiple limitations the *Bioloid* framework has, a simple and intuitive method to direct the robot was designed and correctly implemented. This approach showcases the functionalities of a teleoperated robot in a closed and controlled environment. However, multiple improvements have to be carried out in order to use the robot in a real-life situation. Balance control and multiple poses are required for the automaton to move freely and safely among everyday obstacles, debris and other objects, such as stairs and slopes.

Feedback from multiple testers stated that even being a fairly straightforward approach, moving the robot with only one flick of a finger was very intuitive and user friendly. Further development in stability was a must, but considering the economical cost and simplicity of the project, this solution was highly competent.

Chapter 5

Camera feedback to Oculus Rift and head control

This section of the project deals with one of the most important characteristics of the whole project: sending live footage of what the robot sees to the *Oculus Rift* headset so that the user can observe what the robot is seeing in real time. Furthermore, using the sensor fusion built in the *Oculus Rift* (and IMU consisting of a gyroscope and an accelerometer), one can determine the orientation of the users head and send this data onto the robot so that it turns its head accordingly.

5.1 Components needed

Once again, to achieve this implementation, the following new components will be needed, aside from the ones already in use (Servomotors, *Arduinos* and *Xbees*):

- *Webcam* However, for the sake of easiness, a smart phone will be used instead; any phone with built in camera and Wi-Fi connection will suffice.
- *Oculus Rift SDK2*: Virtual reality head-mounted display with a resolution of 1080x1200 per eye, a 90 HZ refresh rate and a wide field of view.

- *Unity 5*: A cross-platform game engine used to develop video games for PC, consoles, mobile devices and websites (Figure 5.1).

5.2 Camera set up

The initial idea was to use an IP wireless web-camera, very similar to a surveillance camera, which connects to the Internet and uploads live video to it; then one can access this stream using any device with Internet connection. Despite this initial concept, the idea was soon scrapped because of the size of said cameras and specially the price range. The most competitive one was determined to be *Ai-ball* [23], a ultra-portable, wireless camera module that allows users to view and record audio and video wirelessly on their PC or smartphones. Due to its minimal size and highly competitive features, this device could be used to further improve the project. Nonetheless, for the requirements of this project, any smart phone with a decent camera could be used as a simpler and cheaper way of obtaining satisfactory results.

Thus, a smart phone would be used to supply the live footage. An application on the *Android Play Store* and *Apple Store* called *IP Webcam* immediately modifies your phones camera to stream the images onto the internet. A wireless connection is needed, hence the Wi-Fi of the Laboratory was used (an additional feature to add to the robot would be an Internet connexion of it own so it does not have to rely on external routers to do this job). The video can be seen through any device simply setting the correct IP address.

5.3 Unity 5 and Oculus Rift integration

The basic interface of the Unity engine (Figure 5.1) incorporates a series of windows: the Project Browser (used to access and manage the assets), the Hierarchy (contains all the used *GameObjects* and their *Parenting* relations), the Toolbar (for basic controls), the Scene View (interactive sandbox used to position *GameObjects*), the Game View (rendering of the final game from your cameras) and the Inspector (to display detailed information of the selected *GameObject*).

The official *Oculus Rift* webpage allows the download of a software developers kit (SDK) to integrate *Oculus* with Unity. This can be implemented by drag-and-dropping the package onto the existing Unity project. What this does is create a camera (OVR-Camera) in the Scene view that immediately modifies what an

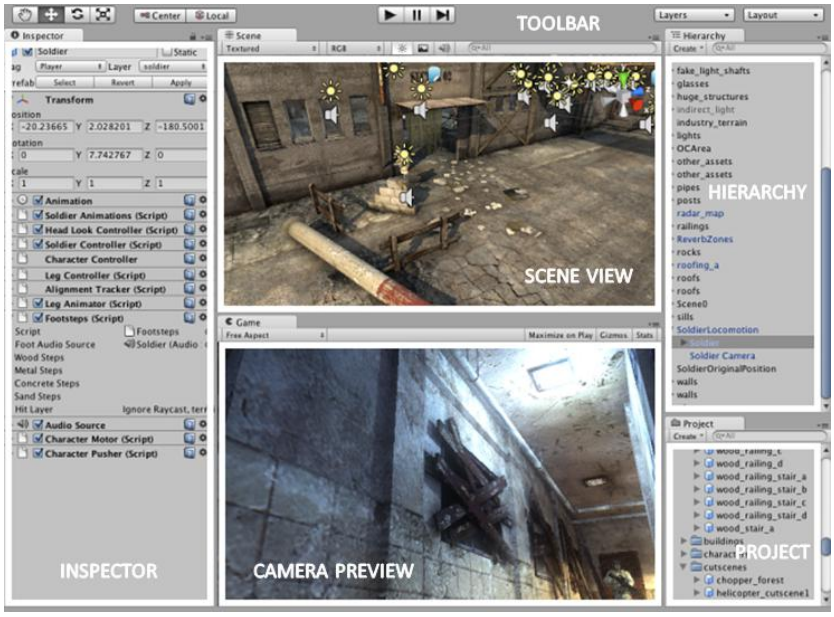


Figure 5.1: Unity 5's interface

original camera would see to be compatible with the *Oculus Rift*. Specifically, this OVR-camera consists of two cameras, which are slightly offset, next to each other, representing the two eyes of a human, to create a stereoscopic viewing of the scene. What each camera sees is then sent to each eye of the *Oculus* so that the human can visualize correctly the images.

To display the images of the *IP Webcam*, a special feature of Unity 5 is used: the *WWW* class. Through a small script, shown in Appendix J.1, attached to a plane (that acts as a screen, like in a cinema), Unity is able to download continuous images from a specified IP address and render them as material textures for the screen, as seen in Figure 5.2. The screens size is set so that it covers as much area as possible of the camera projection, without affecting resolution or scaling of the depicted images (i.e. no proportion is lost).

Since the idea is that the servomotors of the neck of the robot must turn when the *Oculus Rift* is reoriented, the screen seen in the *Oculus* needs to be stationary in relation with the OVR-Camera in the Virtual Reality World. If this does not happen, every time the user turns its head, it would stop facing the screen and look elsewhere, into the emptiness of the VR World. To solve this issue, the rotation matrices that affect the OVR-Camera (when the user turns the head), must also alter the screen in the same way. The documentation of the

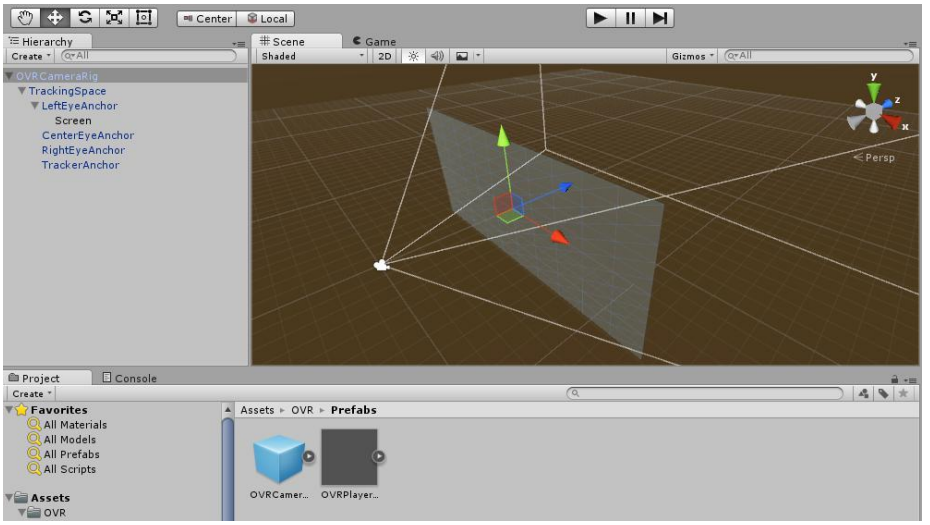


Figure 5.2: Screenshot of Unity 5 showing the OVR Camera and the screen

Oculus Integration for Unity states that the variable *CennterEyeAnchor*, which determines the orientation of the whole OVR-Camera, is actually getting its values from the *LeftEyeAnchor*. This means that the original orientation values are found in the *LeftEyeAnchor* and so making the virtual screen a child of this anchor enables it to rotate in the same way as the *Oculus Rift*; a simple drag-and-drop allows this feature. These variables are found in the OVRCameraRig script (Annex J.2), which is created automatically when downloading the integration pack, and its function is to run the interface between the *Oculus Rift* and the virtual cameras (OVR-Camera) in Unity; thus handling the *Anchor* information (orientation of device in yaw, pitch and roll angles).

5.4 Orientation data retrieval and neck servo motors control

As mentioned before, the camera orientation can be obtained through the *LeftEyeAnchor*, specifically under *localRotation*. However, the values displayed here are represented as w, x, y and z quaternions which are easier to use for matrix calculations but are irrelevant for human visual comprehension. Thus they need to be converted into the yaw, pitch and roll Euler angles. Unity does this for us, simply calling *LeftEyeAnchor.localRotation.EulerAngles.x* (yaw), and respectively for the y (pitch) and z (roll) parameters.

Once the values are derived, they must be sent to the neck servo motors connected to the *Arduino* on the robot. For ease of use, the various *Processing* sketches, *Xbee* protocols and *Arduino* scripts created in earlier sections, will be used to achieve this wireless transfer of data. Unity must therefore use the *Xbee* connected to the PC to send that data. Since the port to which the *Xbee* is connected will already be used by the *Processing* sketch that receives the joystick data, it is necessary to send the *Oculus* orientation data through OSC protocol (which has been previously described) to said *Processing* sketch.

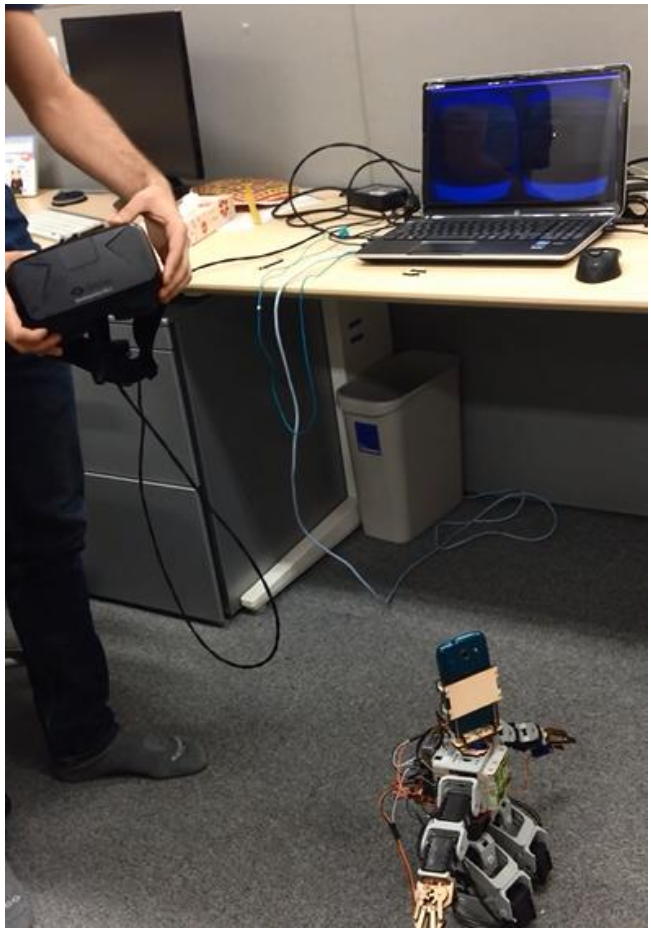


Figure 5.3: *Oculus Rift* and Unity 5 integration receiving the images of the robot and displaying them on the *Rift*; and retrieving the orientation of the headset and sending the data through wireless *Xbee* communication to the robots neck servomotors.

With the *UnityOSC* [8] C# class developed by Jorge Garcia, the sending and receiving of messages between Unity and, in this case specifically, *Processing* sketch, is made possible. As his documentation states, the first part is setting up correctly the client and server information inside the *OSCHandler.init* script (Annex J.3). Since the data will only flow in one direction (from Unity to *Processing*) it is only necessary to start a client (or transmitter) with its specific IPAddress and port. Then this initialisation is called in the OVRCameraRig script with *OSCHandler.instance.init()*. The *Anchor* information mentioned earlier (yaw, pitch and roll values of the *Oculus*) is then sent with the command *OSCHandler.Instance.SendMessageToClient("Unity", IPAdress, values)* and received directly on the *Processing* sketch through events; after a correct set up of the listening ports for the OSC libraries in said sketch.

The data was then processed and sent via *Xbee* onto the *Arduino* on the robot that controlled the neck servos, resulting in a perfect sincronization between the orientations of the *Oculus Rift* and the robots head. Figure 5.3 depicts how the orientation of the *OR* (which is supposedly mounted on the users head but for display purposes was held in the hands in this picture) is mimicked onto the robots head through *Xbee* wireless connection. Additionally, the images seen by the phone (which acts as the robots eyes) can be seen live both on the *Oculus Rift* and the computer. An additional image showing this same process can be observed in Figure 7.7.

5.5 Head design: yaw and pitch servos

Now that the software part is finished, it is time to design and laser cut some pieces to physically create the head of the robot. It must consist of three parts: one that connects the body of the robot with the yaw servo, one that supports the pitch servo on top of the yaw servo and one that holds the phone steadily on top of the robot but at the same time allows for it to turn. The drawings for the different pieces can be found in Annex D. For a better understanding of which piece is which, the following description will use numbers next to each part for a better identification in the drawings section.

Firstly, the hole in the main frame of the robot (1) will be used as mounting for the whole head. Inside it, a support for the yaw servo (3) and an anchor between it and the robots body (2) will be embedded. Once the yaw servo (4) is firmly in place, a connection (5) is used between it and the base of the head (6). With this setup, the yaw rotation of the head is enabled.
Next, the pitch servo (7) is secured onto the head base thanks to a pair of anchors (9). Two U-shaped pieces will work as the lower supports for the phone

(11), which not only holds it into place, but also allows for the pitch rotation. These supports are attached to the base of the head through a pair of anchors (10).
A small case for the upper part of the phone needs to be designed, as a means of connecting the servo movement to the phone whilst, simultaneously, avoiding that the phone applies to much force onto its lower supports and ends up braking them. Two rectangular pieces (13 and 15), parallel to one another thanks to a series of separators (14), will press on both sides of the phone and do the job.
Finally, a way to hook up the servo and the phone case needs to be implemented. this is accomplished using an elongated piece (12), joined on the one end to the servo through a small connection (9), and on the other side to the case via an L-shaped piece (16). This system will transmit the rotational movement of the servo onto the phone, obtaining the pitch movement.

The final result is shown in Figure 5.4, a system that allows the imitation of users head movements (yaw and pitch) by a phone acting as the robots head. Additional pictures of it can be seen in Figures 5.3, 7.7 and 7.9.

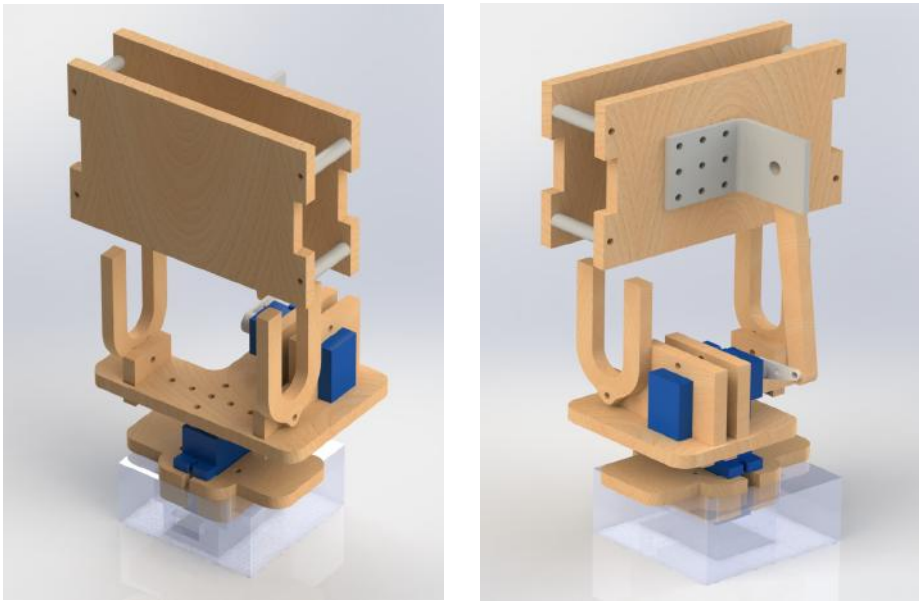


Figure 5.4: Full assembly of head system used for yaw and pitch rotation

5.6 Results and evaluation

Several users tested this mechanism, that is receiving visual feedback from the robot and controlling its head via their own, and the opinions about it were quite similar.

The speediness and precision of the system shocked them positively, achieving the desired orientation without any noticeable delay. The resolution of the servo turning was so rigorous that when facing one of the physical limits of the human head, the robotic head would actually stop moving in that same position, making it a very functional device.
In spite of this, the feedback suggested two main improvements. The former, the image quality was below average which added to the low frames per second created a sickening blur. The latter, due to only having one recording camera, and thus sending the same image to each eye (even though it was slightly offset), resulted in a loss of stereoscopic vision and therefore depth perception.

5.7 Conclusions

All in all, the communication mechanism between *Oculus Rift* and a robotic head can be fully achieved without a great expense, whilst at the same time, giving it a natural feel. Additionally, the camera used should be of enough high quality so that when the images are processed and showed on the *Oculus Rift*, the quality is not lost or the images become distorted. Finally, having two offset cameras instead of one, sending images to each individual eye, would allow for a better stereo vision of the user and a more comfortable experience.

Chapter 6

Voice recognition and voice activated commands

In this section of the project, the robot must be able to be commanded by voice. This is an additional way of adding control to it and can be used to add robustness and safeguard against the malfunctions of other programs.

6.1 *Voce* library for *Processing*

Voce is a speech synthesis and recognition *Processing* library which uses *CMU Sphinx4* and *FreeTTS* internally [21]. It is a very simple API that allows the functionalities of the libraries aforementioned to be used in *Processing*.
The former, *CMU Sphinx4*, is a Java speech recognition library that converts speech recordings into text using its own acoustic models. Two attributes must be set initially for it to work: the language and the grammar model. The grammar model is a script that determines which words the library will be able to recognize inside a specific language. But not only that, it is possible to set is so that it recognizes full sentences or half sentences mixed with simple words. The fact that it has thousands of acoustic models allows the library to recognize almost any word in the dictionary. However, it does sometimes fail when trying to recognize speech in noisy backgrounds due to the disturbance generated. The

latter, *FreeTTS*, is a speech synthesiser system in Java that transforms a text into understandable speech.

6.2 Usage of the library

To use the *Voce* library in *Processing*, one must first correctly paste it into the correct *Processing* installation directory, import it as *import guru.ttslib.*;* and then initialise it as follows:

voce.SpeechInterface.init("location of library", true to enable speech recognition, true to enable speech synthesis,"location of grammar file","name of grammar file");

Additionally, one can also set specific values for the TTS (Text to Speech) library such as the pitch, pitch range and pitch shift. When the library recognizes a word or sentence from the grammar file, the queue size increments with its character length. And so when this happens,

voce.SpeechInterface.getRecognizerQueueSize()>0

then the char array can be popped into a string:

String word = voce.SpeechInterface.popRecognizedString();

which can then be used for whatever purpose the application has. For this project in particular, the recognized speech strings were compared to see if they matched a specific request or not. Due to the instability of the library, sometimes the recognition failed or showed up a different result than the one expected. For this reason, a method (*AreYouSure()*) was created to ask if the recognized command was really what the user desired. This was accomplished using the TTS library and the order *tts.speak(input);*, which synthesises the selected text into speech. Note that when this is done, the speech recognizer must be disable for a moment so that it does not recognize the same text that is being synthesised. The question *Are you sure you want to "command"?* was inquired, to which the user must answer yes or no. A more comprehensive description can be found in at the end of Annex G

Lastly, to define the grammar file from which the library knows which words or phrases to recognize, the following structures had to be specified, with any and all combinations possible:

public <name of set> = <subset> word <additional subsets>
public <subset> = (word1 | word2 | ...);

The sets can be organized in whichever fashion necessary, having subsets first and then words, or viceversa, or everything mixed up. The following script exemplifies the grammar file used for the voice recognition:

grammar voicecontrols;

public <directionalmovements> = <address> move <direction>;
public <turnmovements> = <address> turn <direction>;
public <hands> = <address> <actionhand> <direction> hand;
public <activation> =<address> turn <onoff>;
public <stop> = <address> stop;

public <address> = (robot);
public <direction> = (forward | backward | right | left);
public <actionhand> = (open | close);
public <onoff> = (on | off);
public <others> = (yes | no);
public <VR> = voice recognition on;
public <Controls> = controls on;

6.3 Implementation with joystick control

One of the biggest problems to be faced when using voice recognition is that if you also want to add vocal output from the robot (the robot says the same things as the user), there must be a way to differentiate what are commands and what is simply speech.

To solve this issue, a mechanism for switching back and forth between joystick control (where all speech can be said by the robot because none of it are commands) and voice control was deigned. Basically, when in joystick mode, the system not only deals with the joystick data, but also keeps listening to the sentence *"Voice recognition ON"*. When this order is recognised, voice recognition mode is activated and the joystick data flow is stopped, whilst the program keeps listening for additional voice commands. To go back to joystick mode, the simple use of the command *"Controls ON"* is necessary and everything goes back to its original state. This process is detailed in Annex G

6.4 Results and evaluation

The overall impression is that this control method looks and feels splendid when using to move the robot around with only your voice. It is true that the available commands are certainly limited and could be extended, but that is not a difficult or time-consuming upgrade.
The only real problem the *Voce* library had is that there is no background noise filtering. Hence, if the surroundings are not silent enough, the recognition of commands will output something erroneous or simply nothing at all. Even the own sounds the robot made while moving disrupted this recognition. Possible noise reduction techniques include Wiener (Scalart & Filho 1996), the log minimum mean square error logMMSE (Ephraim & Malah 1985) or the exception of the logMMSE-SPU (Cohen & Berdugo 2002), among many other. Even fusion techniques from the aforementioned individual algorithms could have worked [18]. Nonetheless, due to time constraints, these filters could not be designed nor tested, but they would surely be a great upgrade for the current system.
All in all, adding speech recognition and synthesis to the project came at no cost and little effort, specially considering the good results obtained.

6.5 Conclusions

Teleoperation through Voice activated commands is surely a great concept to add to any project on top of other control mechanisms. It is very useful for increased robustness and looks very nice. However, special care should be taken if the system has to deal also with voice output, specially in telepresence. This means that if you want to make the robot say the same words as those you are speaking, but also send commands to it verbally, there must be a way for the robot to differentiate between one case and the other, and that is precisely why a switching mechanism between both controls was designed.

Chapter 7

General overview of the whole project

All the previous sections of this thesis consisted of almost standalone parts; even though some relations amongst these sections have been referenced, the time has come to unite all of them into one final project and test and evaluate the obtained results.

7.1 UML Deployment diagram

To describe the general view of the project, a UML (Unified Modelling Language) Deployment Diagram will be used. This figure models the *physical* distribution of *artefact instances* (software) and *nodes* (hardware) and how all different pieces are connected.
The *nodes*, represented as boxes, show the hardware components of the project. These might need to be, in turn, divided into smaller parts and this is achieved using nested boxes called *component instances*. Additionally, the *artefact instances* are shown as sheets of paper. Finally there exists two types of relations between instances: the former is a *link* (shown as a full arrow, which can be directional or not) that joins *nodes* and/or *component instances* depicting how they communicate with each other (in this case, over *Xbee*, *Zigbee* or *WLAN*).

The latter, known as *dependency* (dotted arrow), represents how one element requires, needs or depends on another; very much like a client-supplier relationship. Said *dependency* includes a small description of what information is exchanged and in which direction (where the arrow points to).

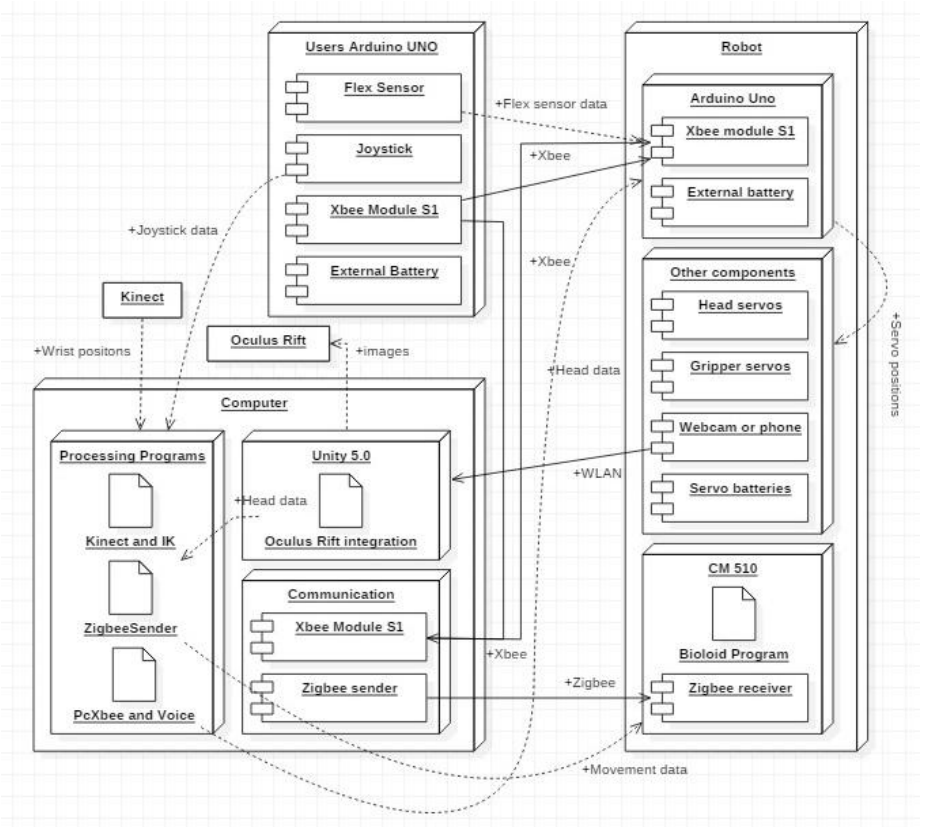


Figure 7.1: UML Deployment Diagram

7.2 Description of the Deployment Diagram

Figure 7.1 represents the interconnections between hardware and software components for this project. Three main blocks or *nodes* can be identified easily: the user, the robot and the computer.
The user/human will wear an *Arduino UNO* board and some sensors, as shown in figure 7.6 to precisely communicate what movements need to be mimicked by the robot. The computer will be used as an information processing node of

all the data, either received from the user and sent to the robot, or vice versa, or obtained by the *Kinect* and *Oculus Rift* and sent to the user. Finally, the robot will be the end actuator of the whole project, reproducing all the desired movements of the user which have been handled by the computer.

7.2.1 Users node

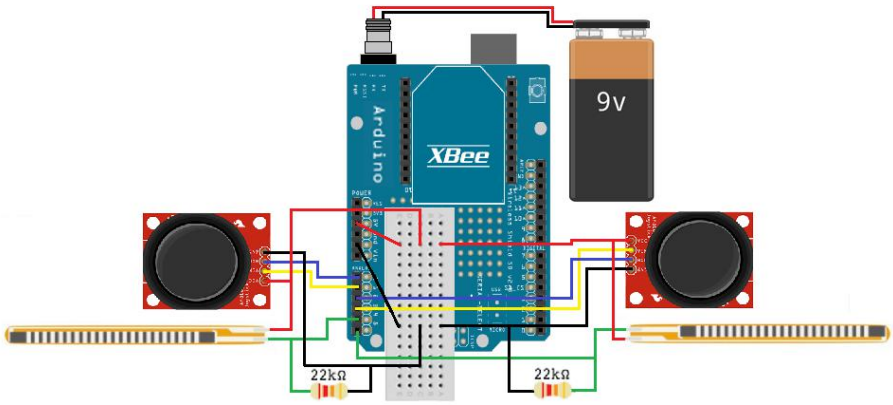


Figure 7.2: Schematic of *Arduino UNO* board on the user

This node is composed of an *Arduino* micro controller mounted on the user. The nested hardware components will be the flex sensors, for gripper control (described in chapter 3), the joysticks, for directional movement control (described in chapter 4), and the *Xbee* module S1, for communication between the user and the computer (Joystick data); and the user and the robot (Flex sensor data) resulting in a Star topology, like in figure 3.3. An external battery will be connected to the board so that the *Arduino* does not need any cabling for power supply. The electrical schematic of the *Arduino* board of the user, connected to the flex sensors, joysticks, voltage dividers and the external power supply can be seen in Figure 7.2.

An essential step for the accomplishment of this project is to design comfortable ways in which to wear all the technology, mainly the *Arduino* board, the flex sensors and the joysticks, on the user. To do so, a glove will be used as a base on which to fit all sensing components and a laser cut box will be used to store the microcontroller.

7.2.1.1 Sensing components on gloves

Since both the flex sensors and joysticks must be controlled by the fingers, it is logical to somehow attach those components onto each hand of the user, but making it so that it is easy to put them on and take them off. The most viable solution is fitting those components onto a glove, which the user can effortlessly put on and remove at any time.

Each part of the hand will have its own role. The index finger will control the flex sensors: stretching the finger will result in an open gripper and vice versa. The joystick handling is a bit trickier. The obvious control is to use the thumbs, however, that requires the joystick body to be perfectly still so that only the head moves. This became an inconvenience all along the project since the joysticks kept wiggling around and so the readings became corrupted. To firmly secure them onto the gloves, additional pieces needed to be laser cut to form some sort of handle on to which the joystick would be mounted (Figure 7.3). This handle could be held by the 3 remaining fingers (middle, ring and pinky fingers) on each hand whilst the thumb played with the joysticks head and the index fingers controlled the flex sensors. All the remaining components, such as the 22 KΩ resistor used as a voltage divider and the soldered wiring, were secured onto the glove via duct tape. Not the most visually attractive solution but probably the easiest and most practical since the components could be removed individually if necessary. The results are shown in figure 7.4.

Figure 7.3: Laser cut Joystick handle

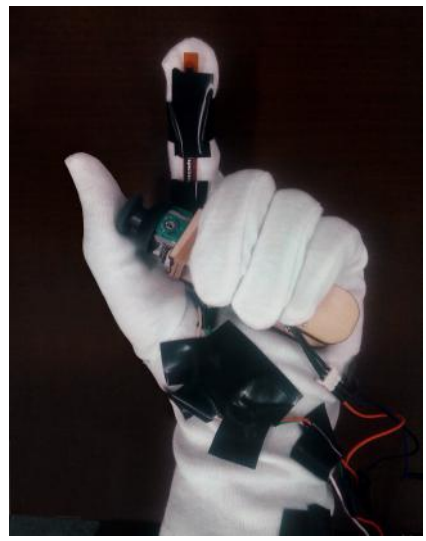


Figure 7.4: Glove worn by user

7.2.1.2 Laser cut box for *Arduino*

Figure 7.5: Render of the case

Figure 7.6: Final design on user

Considering that the *Arduino UNO* micro controller is a very fragile hardware component (specially because it has a *Wireless shield* and *Xbee* module attached to it) and that it must somehow be secured onto the user (where the risk of it moving around frantically and breaking is very high), it is necessary to store the board somewhere to prevent any possible damage. The envisioned solution is a box roughly the size of the board, with enough space to fit some paper balls in it to cushion the potential blows (Figure 7.5). Said box should have a series of openings and indents for an more accessible fitting of the external battery, the wiring and a neckband to wear by the user. Figure 7.6 depicts the resulting laser cut box and how it fits onto the user, including the gloves used for sensing. The schematics for the whole box assembly and its individual parts can be found in Appendix C.

7.2.2 Computers node

This node is used as external processing power which transmits the signals from the user onto the robot and vice versa. It also handles the *Kinect* and *Oculus Rift* nodes. The computer node has 3 differentiated nested nodes: *Processing* programs, Unity and the communication devices (*Xbee* and *Zigbee*).

7.2.2.1 Processing programs

This is probably the most complicated and convoluted part of the whole project because it deals with multiple input and outputs of information through diverse channels.

The first program deals with the *Kinect* depth sensor and pointcloud set up, the tracking of the users skeleton, the procurement of the wrist positions from the *Kinect* and the application of inverse kinematics to it. Additionally it receives the joystick data from the third *Processing* script (discussed in later paragraphs) and sends it, plus the Inverse Kinematics data, to the second *Processing* program, all done via OSC messaging. The full code can be seen in Appendix E.
The second program receives IK and joystick data from the first program via OSC and processes it to create servo positional data and directional movement commands, respectively, to be sent via *Zigbee* as strings to the *CM-510* controller on the robot, as shown in Appendix F.
The third script receives raw joystick data directly from User via *Xbee* communication and processes it before sending it to the first program via OSC. Additionally, it recognizes voice and acts accordingly. If the correct words are perceived, the joystick values are modified appropriately before being sent. Finally, it also receives via OSC the pitch and yaw values of the *Oculus Rift* orientation and after processing them, sends via *Xbee* to the *Arduino* board of the robot. The is described in Appendix G

7.2.2.2 Unity 5.0

Through the *Oculus Rift* integration, Unity is able to download a video stream on a Wireless Local Area Network (*WLAN*) as images, uploaded by a webcam or mobile phone (with the *IPWebcam* app) and send them to the *Oculus Rift* (after processing them to appear as stereoscopic vision). It also detects the headsets orientation and obtains the yaw and pitch angles which are later sent via OSC to the third *Processing* program.

Figure 7.7 depicts the robot mounted with the head servos and phone (acting as eyes/webcam). The design and building process of this part has been mentioned in Chapter 5. The Unity program showing what each eye of the *Oculus* sees can also be observed. These show the user wearing the headset, and if he were to turn the head in one direction, the robot would also move it simultaneously.

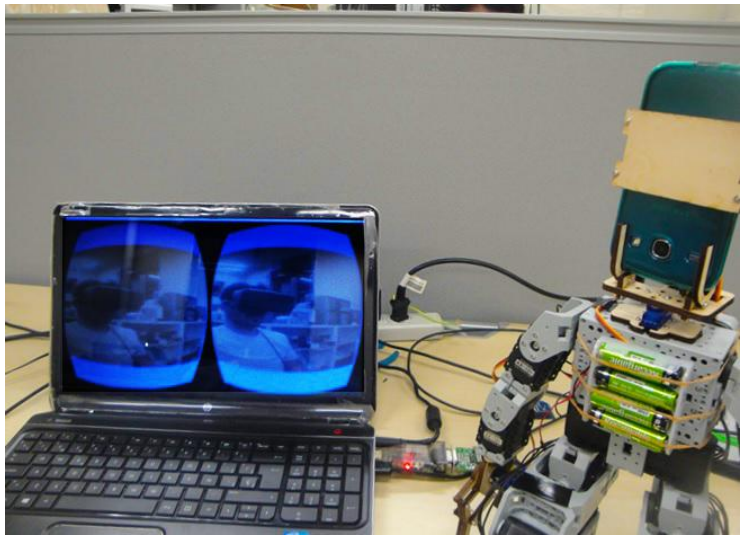


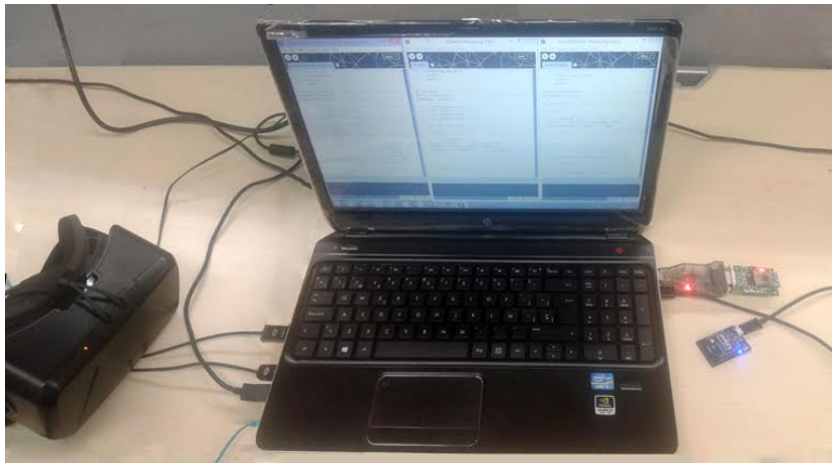
Figure 7.7: Visualization of the images seen by the robot on the *Oculus Rift*, by the user, over *WLAN*

7.2.2.3 Input and output devices

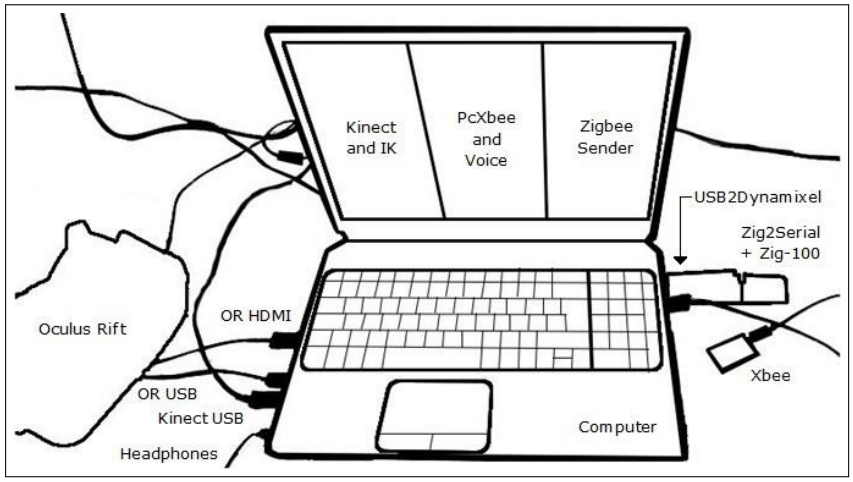
Various connections from and to the computer must happen at the same time; these are displayed in Figure 7.8.
First af all, the three *Processing* programs (discussed in previous sections and shown in the Deployment Diagram in Figure 7.1) can be seen on the computers screen. Additionally, on the left side of it, the connections between *Oculus Rift* (USB and HDMI), the *Kinect* (USB) and the headphones can be identified. Finally, on the right side, the connectivity devices are viewed: the *Zigbee* connection is accomplished using a Zig-100 module mounted on the *Zig2Serial* board, which in turn is connected to the computer via the *USB2Dynamixel* device; and the *Xbee* module is also connected to the computer via a breakboard and a miniUSB cable.

7.2.3 Robot node

This is the final actuator of the whole project, all data that has been obtained and processed flows into this one node. Once again, there are three nested nodes inside: the *Arduino UNO* board, the *CM-510* microcontroller and some additional components.



Original picture



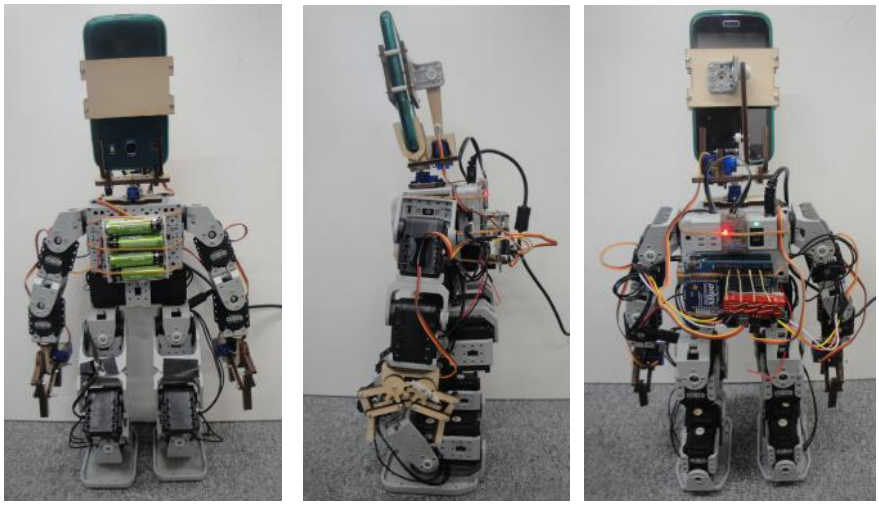
Schematics

Figure 7.8: Illustration of the different connections amongst devices (*Kinect*, *Oculus Rift*, *Zigbee* module and *Xbee* module) and Computer

7.2.3.1 *Arduino* board of the robot

Powered by an external battery so that no cables are needed and equipped with a *Wireless shield* and *Xbee* module for communication, this controller receives the flex sensor data directly from the users node and the yaw and pitch angles

from third *Processing* program.
Some problems arose when trying to attach the microcontroller and the external batteries (for the board and for the servos) because it resulted in a loss of stability. The concept was to add as much weight in the front of the robot as in the rear to mitigate the negative effects of these perturbances. To do so, the 4 AA batteries that powered the servos were allocated in the front and the *Arduino* plus its own battery were assembled on the rear, as shown in Figure 7.9.



Frontal view Lateral view Back view

Figure 7.9: Different views of *Arduino* microcontroller and external batteries (for board and servos) onto the robot

7.2.3.2 CM-510 controller

The final nested node consists of the *CM-510* controller mounted on the robot. Thanks to its *Zigbee* module, it is able to receive the joint positional data and directional movement information (as strings) from the second *Processing* program and process it to finally act appropriately. The result is a robot that imitates the users hand movements and moves according to the users desires. Figure 7.10 depicts several views of the controller and notes its different parts, where the most important ones are the *PC Link*, to upload the coded firmware, the *3 PIN Dynamixel Port*, used to activate and control all servomotors, and the *Power Jack*, which, due to the lack of battery, makes the robot need to be connected to a power source continuously.

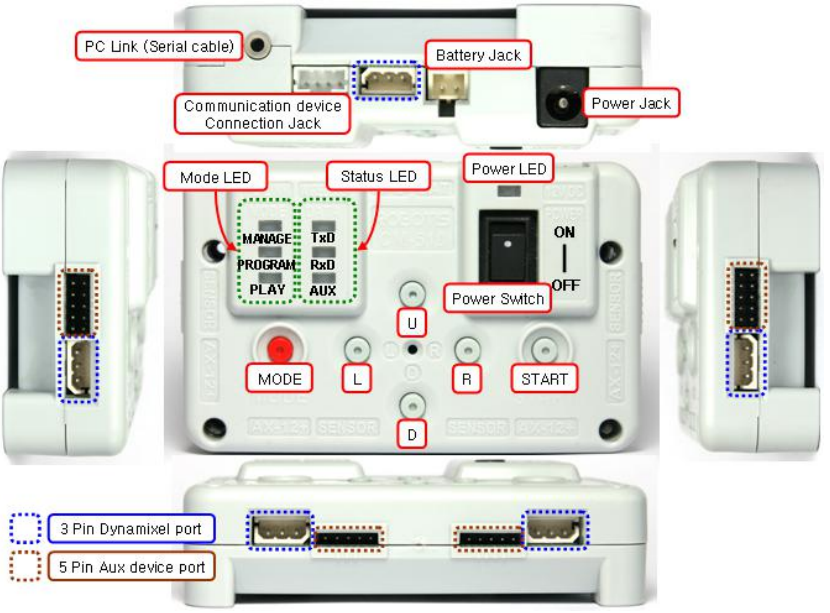


Figure 7.10: *CM-510* controller of the *Bioloid* robot

7.2.3.3 Additional components

Here we find the Gripper and Head (yaw and pitch) servos, controlled by the *Arduino*, the batteries that power them and the Webcam/phone that uploads real time video stream onto a specific IP address.

Chapter 8

Pick and Place test

The entirety of the project has finally been assembled together, all parts working cohesively and in unison. The best way to evaluate if the device works appropriately will be to run a series of tests on the whole system to see how it performs; and considering we are using a teleoperated miniaturized humanoid robot, what better way is there to test its virtues and defects than by applying a *Pick and place* task.

8.1 The challenges

This section describes the tests the robot must endure to evaluate its full capabilities and limitations. To do so, an adequate challenge had to be devised. Therefore, a modified *Pick and place* task will be used, which will be described later in section 8.1.4. However, prior to its implementation, a series of smaller exercises will be used to check other specific attributes of the robot, equally important:

8.1.1 Teleoperability range

As the title of the thesis suggests, teleoperation is one of the key aspects of this whole project. therefore, the initial logical step was to experiment how teleoperable the *Bioloid* actually was, by finding out at what ranges the automaton could still be controlled. Hence, the *Xbee* (for exchange of information between *Arduinos* and computer) and *Zigbee* (data transfer from computer to robot) communications were tested.
To do so, the user was located next to the computer, the *Kinect* and the *Oculus Rift* (because that is how the setup would look like in a real world application), wearing both the designed *Arduino* box and the sensing gloves. The robot would be located somewhere different, at a larger and larger distance with every task. The only limitation confronted was that no battery wa available for the *Bioloid*, thus it had o be connected to power sockets, which not always were distributed every 5 meters as intended. This does not limit in any way the performance of the robot since what is tested is the wireless communications and not the battery life or how it affects the movements of the robot after it has been partially drained.
Accordingly, the aim of this exercise is to be able to move correctly the arms, turn the robot in any direction and control the grippers without any major delay or disturbance, at long ranges.

8.1.2 Arm movement precision

Either for transmitting non verbal communication through gestures during telepresence, or for grabbing human-made tools, opening doors or anything of the sort; having a great control of where the robots arm is located and in what way becomes essential for the success of the operation. This task will deal specifically with the accuracy and speed of the designed arm control.
The amount of data extracted of the users arm position from the *Kinect* is truly large. Most of the information is lost during the processing since the robot can only keep up with so much due to physical restrictions. Considering the transmission rates, around 5 out of 6 data inputs are totally neglected, and that is only happening during communication. This number increases drastically during robot movement because it obviates all inputs while doing so.
This leads to a somewhat fast but jittery response of the robots arm position; which in some cases is irrelevant, but in others becomes a great inconvenience. Hence a small control algorithm must be designed as mentioned in chapter 2. Since no feedback control loop made sense for the scope and competences of this project, other ways had to be discovered.

The resulting method takes one measurement and adds it in a certain proportion to the previous one. This is not to be confused with a *Proportional Controller* because the error between input and output is not used, but rather the difference between one input and the next. The percentage of old versus new measurements is to be tested to find a suitable value, fast and precise enough, before moving onto the *Pick and Place* task.

8.1.3 Stability on different surfaces

One of the major problems one faces when dealing with miniaturized humanoid robots is that the friction and steepness of the ground affects, in high measure, the stability and movement of it. Since the robots leg movements had been predefined (hard coded), it was necessary to see in which surface they worked best and which were just disadvantageous.
Small tests were conducted, where the robot simply walked on different surfaces and the quickness and effectiveness of its response was assessed.

8.1.4 *Pick and place* final task

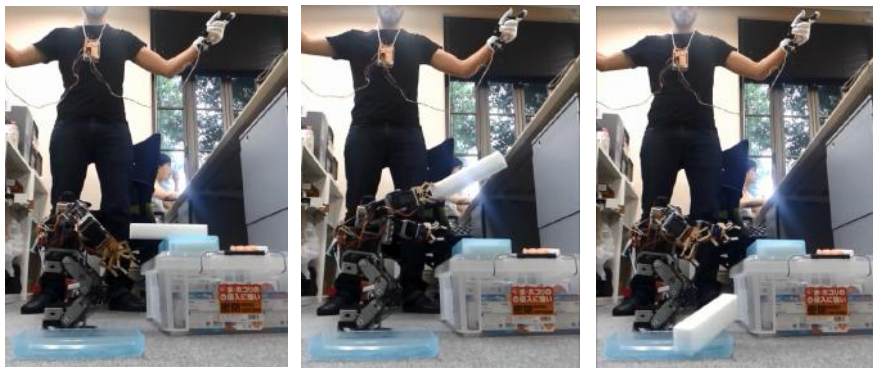


Figure 8.1: *Pick and place* test completed

Once all other aspects of the robot have been taken care of, it is time to fully task the full potential of the project as a whole. The most reasonable way is to make the system undergo a very simple test for humans, but a very complicated one for teleoperated robots: the *Pick and place* task.
The objective will be to teleoperate the robot using the users body and the visual feedback on the *Oculus Rift*, and move it (using voice commands if necessary) to a certain location where a series of objects (sponges) will be positioned. With

arm positional and gripper control, the user must be able to grab one of the objects and carry it to a secondary location where it will be released.
The overall evaluation of the mission will be dependant on the time it takes to finish it, the number of attempts and how and why it has failed, if it does.

8.2 The participants and location

Two lab peers and one of the supervisors will attempt some of the tests and help with the result assessment and improvement ideas. The experiments will take place in and around the Lab, specially the teleoperable range test which requires a huge distance between user and robot.

8.3 The results

Below, the general results for the four tests will be presented. For the analysis of the outcomes, the possible improvements suggested by the users and the real-life applications (and limitations) the system has, please refer to the next chapter (Chapter 9).

8.3.1 Teleoperability range

Operating the robot from the other side of the room (10 meters) had no negative effect on the communication. Therefore, larger distances had to be tested. Using the departments main hallway with no obstacles or walls in between, and testing the robots response every 5 meters, its was concluded that:

The *Xbee* communication was correct until the 35 meter mark, then part of the signal was lost during transmission and the gripper and neck servos started behaving differently to what was expected, mainly the final position did not match the users command. After the 40 meter mark no communication was possible. Further investigation of the *Xbee* ranges on the developers page [5] confirmed the results and added that the outdoor line-of-sight rang could go up to around 100 meters.
The *Zigbee* communication resulted in about half of the *Xbees* potential. Around the 15 meter mark the *Bioloid* started loosing arm positional accuracy and joystick and voice commands had to be issued several times before they took effect. A few meters after that all communication stopped.

The visual feedback from webcam/phone to the *Oculus Rift* was subject to how the *WLAN* covered the area. Basically, after moving far enough, the phone sent worse quality images and lower frames per second until it suddenly disconnected from the network and ceased all communication.

Obviously, the addition of walls would be detrimental for all communication bounds. However, this could not be measured due to inaccessibility to other laboratories, but a test like this would provide very useful information as to how the designed system performs in real world environments.

8.3.2 Arm movement precision

As shown in the last part of Annex E, the variable *lambda*, going from 0 to 1, represents the percentage of old data added onto the new data. The closer this value is to 1, the less effect the old data has over the new one, thus resulting in a quicker and less precise response, and vice versa.
After several tests, it was concluded that to obtain a positional correctness, whilst not having huge delays, was to set this value to around 0.7 - 0.8 . This meant that the user could move his/her arms very fast but the robot would take its time to get there; and if the end position was constantly being changed the robots arm would never reach the said location because it was simply to slow. However, focussing on the *Pick and place* test, this approach was more fruitful because the users movements could be rectified quite easily and the overall final position of the robotic hands was accurate enough to the users desires.

8.3.3 Stability on different surfaces

A very simple test leading to very straightforward results. Since the robots movements are hard coded, they do not perform equally on different surfaces. For instance, movements that work fine in a carpet covered floor barely produce any movement on more slippery materials like wood coated with acrylic resins, commonly used on tables.

8.3.4 *Pick and place* final task

Finally, the experiment that tested the project as a whole lead to very interesting outcomes.

First and foremost, it was observed that using such a big and heavy phone/recording device as head (compared to the robots body) increased dramatically the instability of the *Bioloids* movements; and so for further testing the head had to be removed. This solution became more evident when realizing that the depth perception obtained visually via the *Oculus Rift* had been compromised because of using only one recording camera, thus loosing stereoscopic vision. Knowing how far an object or obstacle was was extremely hard.

Regarding the joystick control and voice commands needed to move the robot around, it was seen that they worked acceptably and quite comfortably (even though sometimes the voice commands had to be repeated several times if there was too much background noise). Switching from one type of control to the next was fairly easy (just a few words) and the results were appropriate. The only problem this part had was the instability on different surfaces (as mentioned before) and the level of precision to which one could move: the size of steps was predetermined so you could not perform smaller steps when coming closer to the desired location. But, overall, the movement control was very satisfactory.

The arm control needed a bit of getting used to and experiencing for it to be painless to command, but after a few tries and setting the correct *lambda* value (referred to in the previous sections), one could actually position the arms surprisingly accurately.

Finally, the gripper control (opening and closing) through the flex sensors attached to the gloves worked very effectively due to its high resolution. The only problem faced was that since it was only a 1 DoF hand, sometimes it became very difficult to access objects positioned in a certain way. Moving the robot around and approaching the object from a different angle usually solved this issue but it still was a bit more time-consuming and required some more effort than necessary.

Chapter 9

Final conclusions

This final chapter presents the conclusion to the entirety of the project. It evaluates the outcomes of the tests and summarizes all the extracted findings. Possible future improvements or real-life applications will also be discussed. Finally, a few concluding remarks will emphasize the success of the project.

9.1 Discussion and further research recommendations

From the several individual tests and the experiments of the project as a whole, a wide variety of conclusions can be drawn. Two main topics will be discussed in this section: teleoperability and telepresence of miniaturized humanoid robots.

As far as teleoperability goes, this project has definitively proven that, without the need of extensive financial or technological resources, an exceptional, long distance remote controlled device is feasible, and that its results are more than competitive.
The overall distance from which the user was able to fully control the robot was a maximum of around 20 meters (section 8.3.1), and this was accomplished using the lowest Tier of wireless *Xbee* and *Zigbee* communication devices. Upgrading

these technologies could allow for a teleoperability of hundreds of meters or, instead, one could attach an internet connectivity device (such as a router) to stretch these boundaries even further. Of course, if this were to be done, a substantial amount of research should be undergone which escapes the scope of this project. However, the end result would be truly astonishing since you would be able to fully command the robot (almost) anywhere in the globe.

Concerning the mechanical side of the this venture, there are many aspects that certainly need an upgrade from the current prototype version for it to be of actual use in real-life.
Starting from the bottom, the initial *Bioloid* framework is an adequate base to build upon, however, its lack of robustness and stability have been clearly proven throughout the whole thesis. The *Darwin-OP* (Figure 1.1) and *Nao* (Figure 1.2) robots discussed at the beginning of the project would surely provide a more compact and cohesive structure to built upon, but that would, in turn, lead to a loss of freedom when developing and upgrading the robot. That is precisely one of the features that makes the *Bioloid* so desirable: it is an open framework that has multiple enhancing possibilities and is designed to provide those opportunities. Therefore a *Bioloid* robot is perfect for prototyping and testing out different technological approaches and devices, but other, more solid, automatons should be used if this project were to be expanded into quotidian life / working spaces.
The positional arm precision obtained with the robot, after careful testing and selection of the *lambda* value (as referred to in section 8.3.2), was exceptional, and can be clearly used for real life applications. Some upgrades need to be devised first, as mentioned before (section 2.7), like a self avoidance collision system or a positional feedback control for optimal results, but the overall outcome was highly compelling.
Furthermore, it is true that the hands of other commented robots would have been more sturdy, but they are still 1 DoF, thus lacking the rotational degree of freedom to grab objects in diverse orientations (just like the *Bioloid*), and so they do not pose a significant advantage over the laser cut grippers. in spite of this, to fully reach the true potential of robot grasping, different hands must be designed with at least 2 DoF. Moreover, applying individual control of each finger would substantially benefit the users grabbing precision, obviously at a higher expense.
The 2 DoF heads of the *Nao* and *Darwin-OP* however, are significantly more advanced than the one created with a pair of servos, some wooden planks and a phone. Even though, the results from the various tests showed that the designed head and yaw servomotors worked perfectly and smoothly, both *Nao* and *Darwin-OP* heads have inbuilt HD cameras and head tracking software, not to mention they are proportional to the rest of the body and therefore do not cause instability of movements. All problems faced because of the image blur and depth perception loss (section 8.3.4) would be greatly avoided. This is

essential before disclosing any similar product to the public.

The use of *Arduino* boards and other specific devices became of great use for ease of prototyping and testing. Even though this board is definitely not the most competitive one on the market, at higher price ranges that is, its usefulness has become more than evident throughout the whole project. This is due to all the documentation and resources available, and because of the effortless access most universities have to it and its additional components.

Joysticks might not be the fanciest of devices for moving a robot around, but they do provide a natural feel to it and allow the user to move around freely without that affecting the robot itself.

This is not completely accomplished though because the *Kinect* camera requires the user to be located n a certain area inside its field of view. The user can still move around but is quite limited.
This is precisely one of the biggest upgrades this system needs to make it more applicable in real life. The arm positional tracking device needs to be modified: instead of using a stationary *Kinect* camera, other options would be more advisable since they allow the free movement of the user; such as an IMU based suit [12] which does not restrict the user in any way. A direct consequence of this approach is that more software coding and signal processing would be required but the final product would be worth it.

One of the last components of this project that needs to be remarked is the *Oculus Rift*. The visual feedback onto the user and the head tracking accomplished by this device are probably the most conclusive parts of this project due to their great results (section 5.6). Its implementation on real life scenarios is definitely possible even in its early stage of development. This system is certainly what makes the project genuinely teleoperable.
Further ideas to transmit more information onto the user would be to include a tactile feedback mechanism, to feel when you are grabbing or touching something.

The use of a miniaturized humanoid robot for telepresence is a sound idea. It does depend on which scenario telepresence would want to be used, but in general, work environments (offices) require a small, compact and easy to use device. And the *Bioloid* robot (or any humanoid of that size) would certainly come in handy because it has the necessary degrees of freedom to operate in those surroundings whilst at the same time provide the user with a wide range of telepresence capabilities. Other magnitudes of robots would also be applicable to certain scenarios since they can reproduce human movement in a more accurate manner (they would be able to open doors or carry heavy machinery/-tools), but their cost would be considerably larger. Thus to select the size, a

trade off has to be made, compromising more features and functionality for a lower price tag and ease-of-use; or the other way around.
One vital aspect this project has not been able to deal with, due to time restrictions, was the transmission of sound from the user to the robot and vice versa. For a legitimately telepresence robot to exist, it must be able to reproduce the sounds the user says and to listen to others speaking. Even though this competence was not handled directly, an algorithm to differentiate between voice activated commands and actual speech was presented in section 6.3 in case that aspect would have been developed.

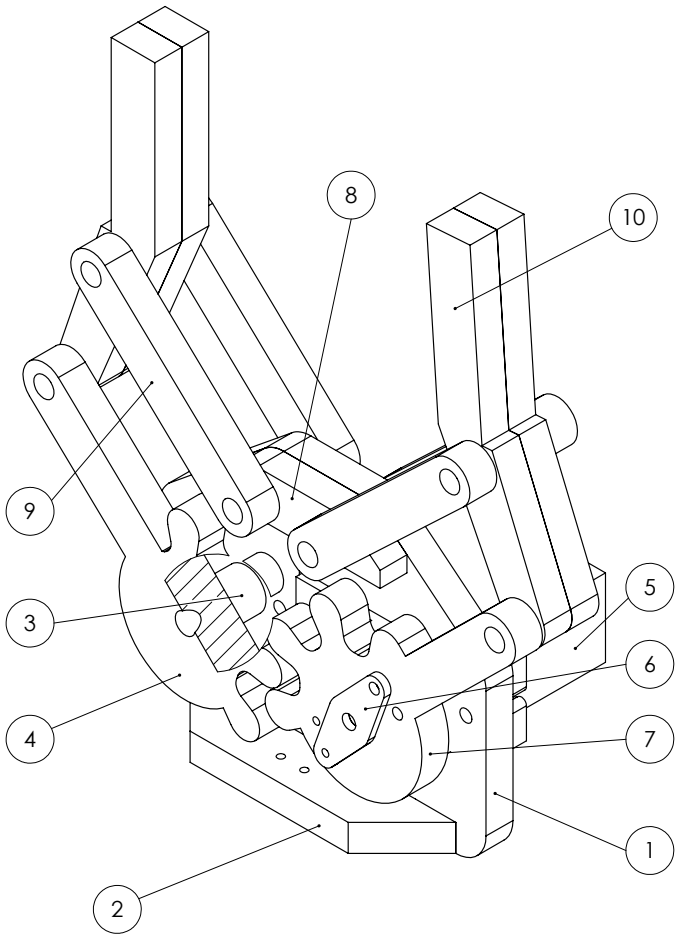
Finally, the software used throughout the project is worth noticing. The *Processing* and *Arduino* programs utilized have been very well documented and came with a wide range of useful libraries. The implementation and testing of different systems has been made possible thanks to such effortless programs. The *Atmega Studio 6.2* used to code the *Bioloids* firmware, on the other hand, required a higher degree of involvement, since it supported a lot more in depth functionalities than the other two, but is still a common and reliable software.

9.2 Concluding remark

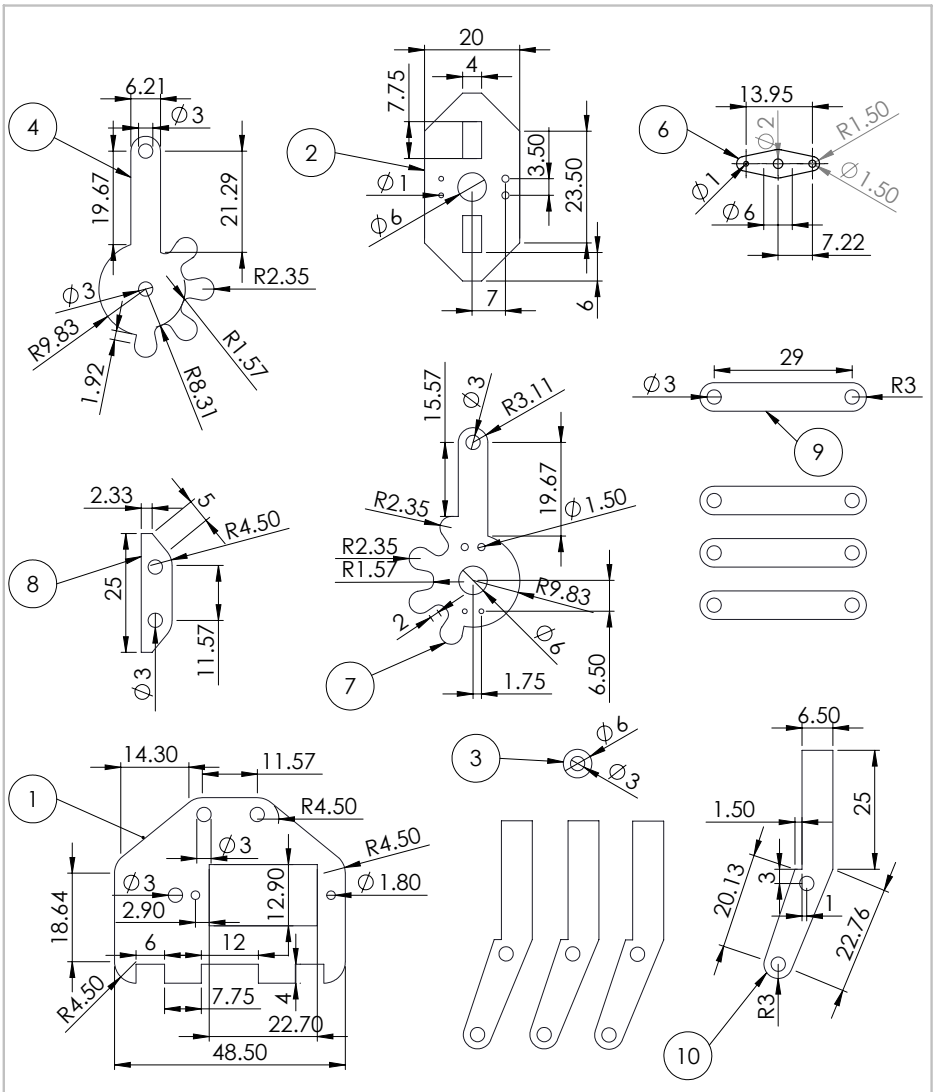
The whole project can be summarized in the following key statement: the viability of teleoperated miniaturized humanoid robots has been certainly proven all through the project. While it is true that multiple upgrades can and should be made to the current system, it is obvious that it would require a bigger investment. This venture has balanced correctly good results and practical functionalities with a low and affordable budget.

Appendix A

SolidWorks Drawings of Gripper Assembly and its individual parts



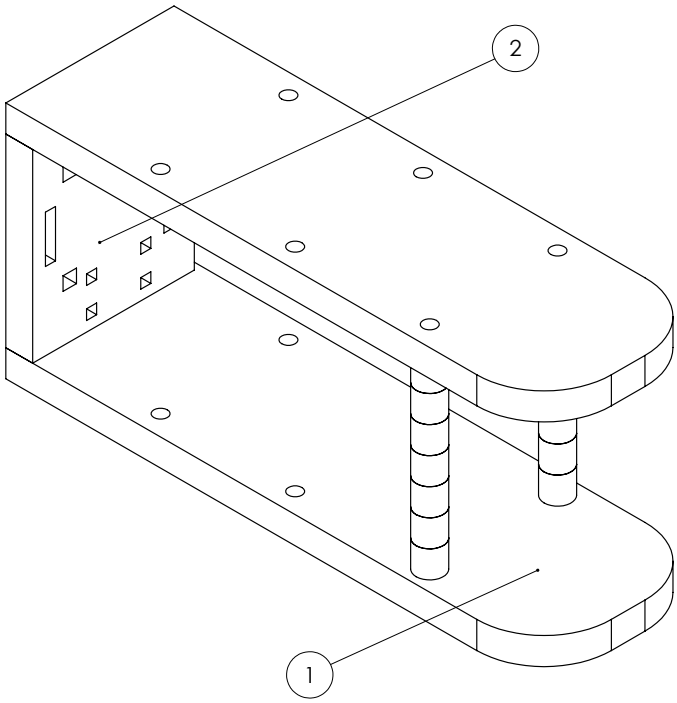
UNLESS OTHERWISE SPECIFIED: DIMENSIONS ARE IN MILLIMETERS SURFACE FINISH: TOLERANCES: LINEAR: ANGULAR:	FINISH:		DEBUR AND BREAK SHARP EDGES	DO NOT SCALE DRAWING	REVISION
	NAME	SIGNATURE	DATE	TITLE:	
DRAWN				Gipper Assembly	
CHK'D					
APPV'D					
MFG					
Q.A				DWG NO.	A4
			WEIGHT:	SCALE:2:1	SHEET 1 OF 1



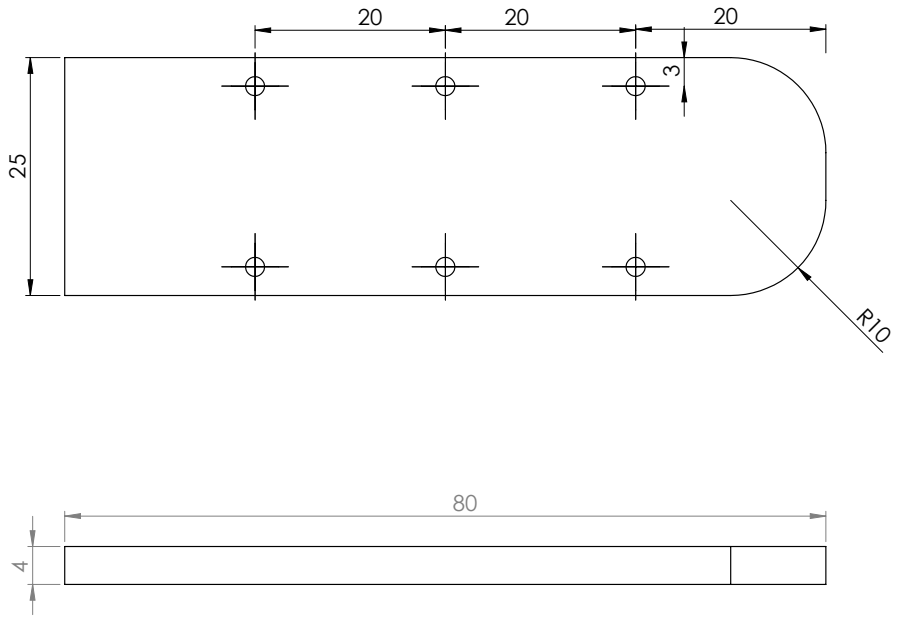
UNLESS OTHERWISE SPECIFIED: DIMENSIONS ARE IN MILLIMETERS SURFACE FINISH: TOLERANCES: LINEAR: ANGULAR:	FINISH:		DEBUR AND BREAK SHARP EDGES	DO NOT SCALE DRAWING	REVISION
	NAME	SIGNATURE	DATE	TITLE:	
DRAWN				Gripper Pieces	
CHK'D					
APPV'D					
MFG					
Q.A				DWG NO.	A4
			WEIGHT:	SCALE:1:1	SHEET 1 OF 1

Appendix B

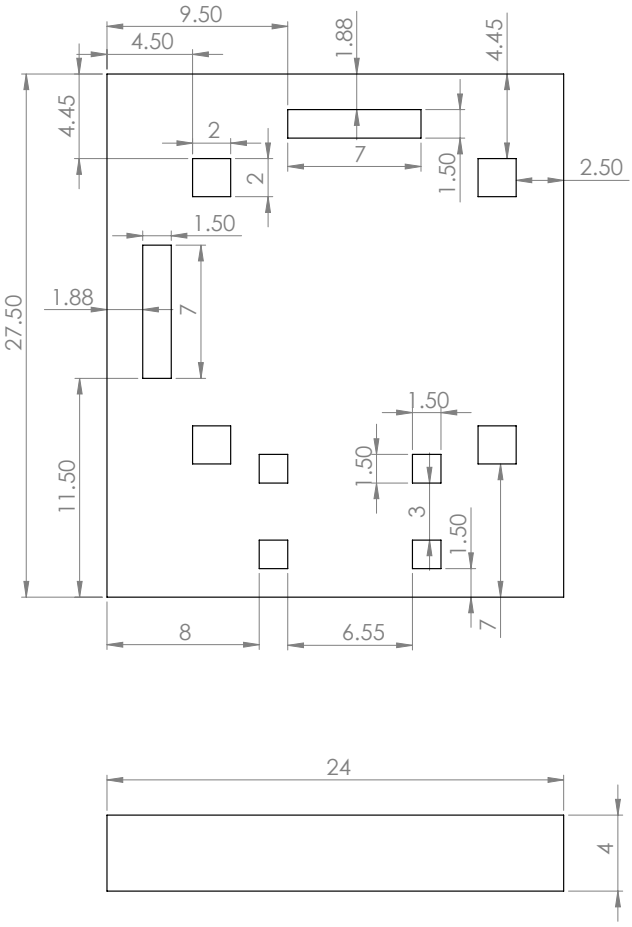
SolidWorks Drawings of Joystick Assembly and its individual parts



UNLESS OTHERWISE SPECIFIED: DIMENSIONS ARE IN MILLIMETERS SURFACE FINISH: TOLERANCES: LINEAR: ANGULAR:	FINISH:		DEBUR AND BREAK SHARP EDGES	DO NOT SCALE DRAWING	REVISION
	NAME	SIGNATURE	DATE	TITLE:	
DRAWN				Full Joystick Assembly	
CHK'D					
APPV'D					
MFG					
Q.A				DWG NO.	A4
			WEIGHT:	SCALE:2:1	SHEET 1 OF 1



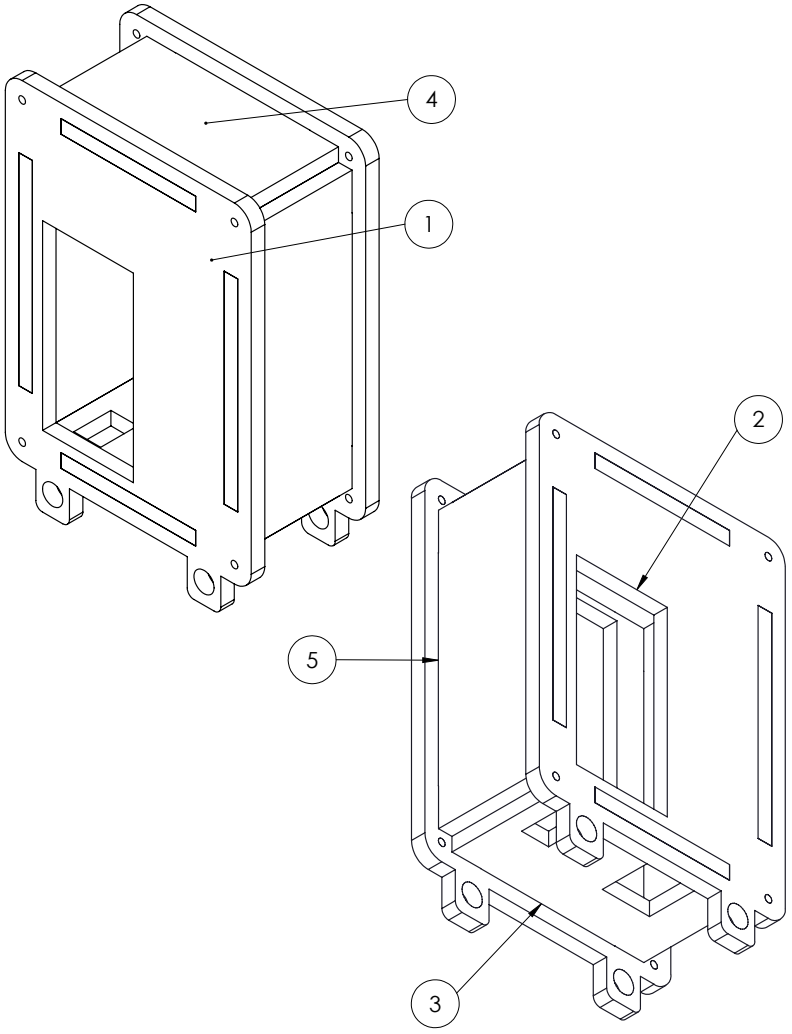
UNLESS OTHERWISE SPECIFIED: DIMENSIONS ARE IN MILLIMETERS SURFACE FINISH: TOLERANCES: LINEAR: ANGULAR:	FINISH:		DEBUR AND BREAK SHARP EDGES	DO NOT SCALE DRAWING	REVISION	
	NAME	SIGNATURE	DATE	TITLE:		
DRAWN				Joysticks External Piece		
CHK'D						
APPV'D						
MFG						
Q.A				DWG NO.		A4
				1		
				WEIGHT:	SCALE:2:1	SHEET 1 OF 1



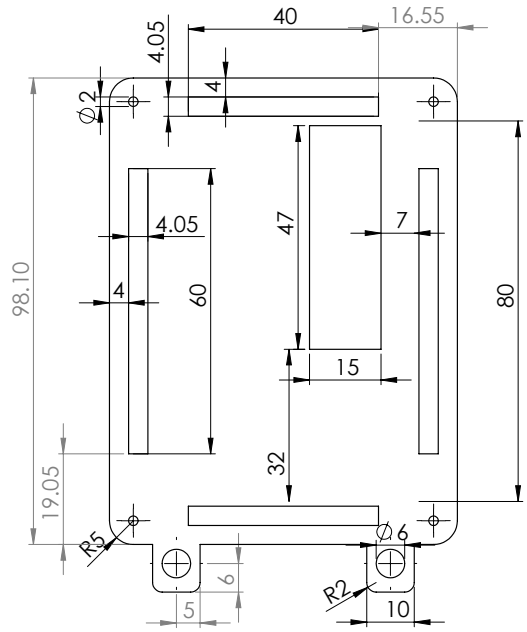
UNLESS OTHERWISE SPECIFIED: DIMENSIONS ARE IN MILLIMETERS SURFACE FINISH: TOLERANCES: LINEAR: ANGULAR:	FINISH:		DEBUR AND BREAK SHARP EDGES	DO NOT SCALE DRAWING	REVISION
	NAME	SIGNATURE	DATE	TITLE:	
DRAWN				Joysticks Internal Piece	
CHK'D					
APPV'D					
MFG					
Q.A				DWG NO. 2	A4
			WEIGHT:	SCALE:4:1	SHEET 1 OF 1

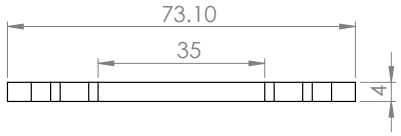
Appendix C

SolidWorks Drawings of *Arduino UNO* case Assembly and its individual parts

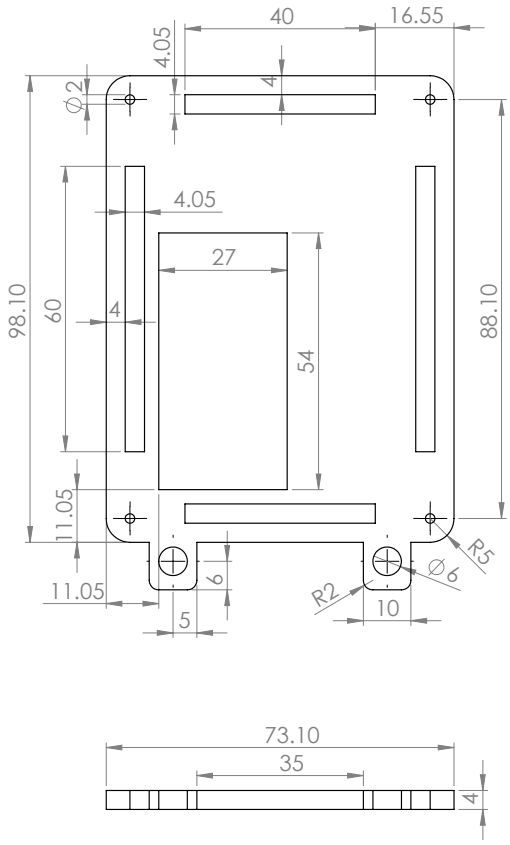


UNLESS OTHERWISE SPECIFIED: DIMENSIONS ARE IN MILLIMETERS SURFACE FINISH: TOLERANCES: LINEAR: ANGULAR:	FINISH:		DEBUR AND BREAK SHARP EDGES	DO NOT SCALE DRAWING	REVISION
	NAME	SIGNATURE	DATE	TITLE:	
DRAWN				UNO Box Assembly	
CHK'D					
APPV'D					
MFG					
Q.A				DWG NO.	A4
			WEIGHT:	SCALE:1:1	SHEET 1 OF 1

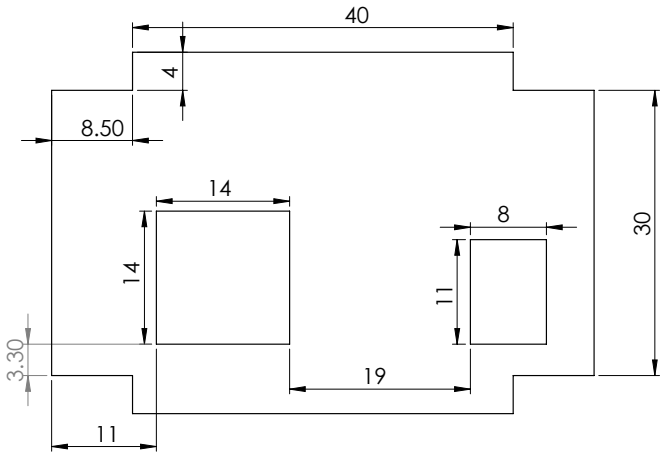


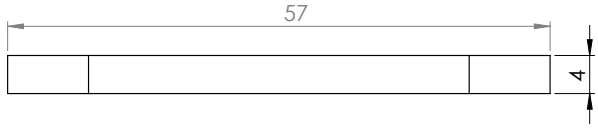


UNLESS OTHERWISE SPECIFIED: DIMENSIONS ARE IN MILLIMETERS SURFACE FINISH: TOLERANCES: LINEAR: ANGULAR:		FINISH:		DEBUR AND BREAK SHARP EDGES		DO NOT SCALE DRAWING	REVISION
	NAME	SIGNATURE	DATE			TITLE:	
DRAWN						Upper UNO Box	
CHK'D							
APPV'D							
MFG							
Q.A				MATERIAL		DWG NO. 1	A4
				WEIGHT:		SCALE:1:1	SHEET 1 OF 1



UNLESS OTHERWISE SPECIFIED: DIMENSIONS ARE IN MILLIMETERS SURFACE FINISH: TOLERANCES: LINEAR: ANGULAR:	FINISH:		DEBUR AND BREAK SHARP EDGES	DO NOT SCALE DRAWING	REVISION
	NAME	SIGNATURE	DATE	TITLE:	
DRAWN				Lower UNO Box	
CHK'D					
APPV'D					
MFG					
Q.A				DWG NO. 2	A4
			MATERIAL		
			WEIGHT:	SCALE:1:1	SHEET 1 OF 1





UNLESS OTHERWISE SPECIFIED: DIMENSIONS ARE IN MILLIMETERS SURFACE FINISH: TOLERANCES: LINEAR: ANGULAR:		FINISH:				DEBUR AND BREAK SHARP EDGES	DO NOT SCALE DRAWING	REVISION
	NAME	SIGNATURE	DATE					
DRAWN								
CHK'D								
APPV'D								
MFG								
Q.A								

SolidWorks Student Edition.
For Academic Use Only.

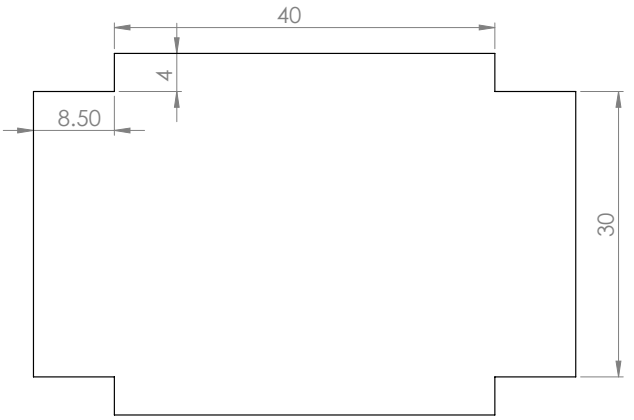
WEIGHT:	SCALE:2.1	SHEET 1 OF 1	

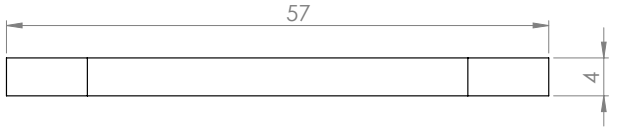
TITLE:

Front UNO Box

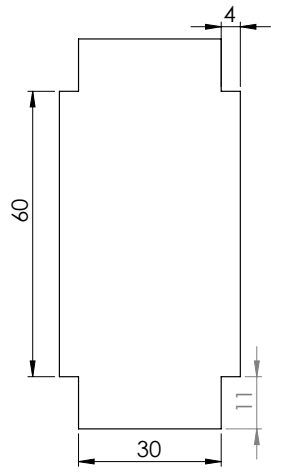
DWG NO. 3

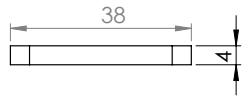
A4





UNLESS OTHERWISE SPECIFIED: DIMENSIONS ARE IN MILLIMETERS SURFACE FINISH: TOLERANCES: LINEAR: ANGULAR:	FINISH:		DEBUR AND BREAK SHARP EDGES	DO NOT SCALE DRAWING	REVISION	
	NAME	SIGNATURE	DATE	TITLE:		
DRAWN				Back UNO Box		
CHK'D						
APPV'D						
MFG						
Q.A				DWG NO.		A4
				4		
				WEIGHT:	SCALE:2:1	SHEET 1 OF 1

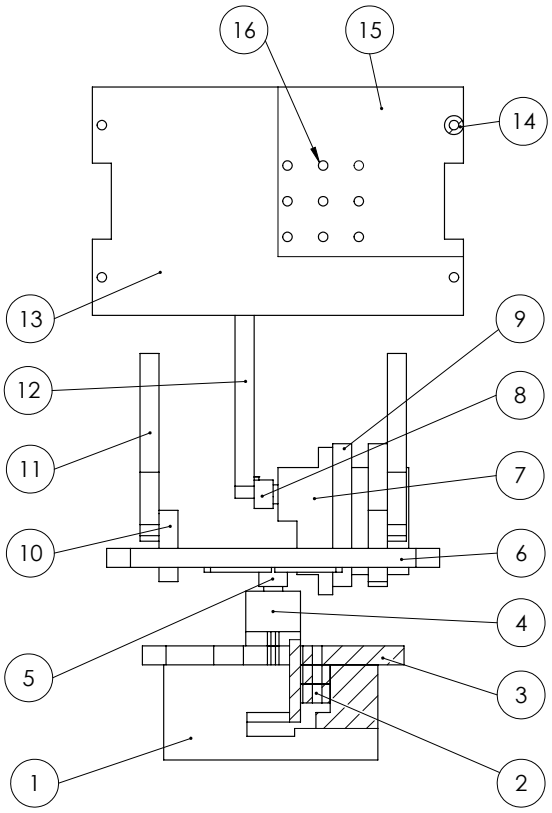




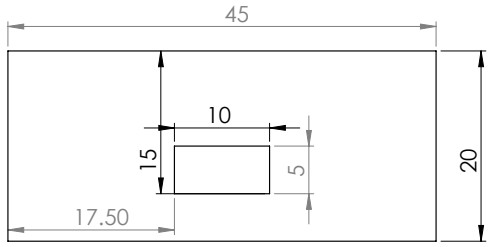
UNLESS OTHERWISE SPECIFIED: DIMENSIONS ARE IN MILLIMETERS SURFACE FINISH: TOLERANCES: LINEAR: ANGULAR:		FINISH:			DEBUR AND BREAK SHARP EDGES	DO NOT SCALE DRAWING	REVISION
	NAME	SIGNATURE	DATE			TITLE:	
DRAWN						Lateral UNO Box	
CHK'D							
APPV'D							
MFG							
Q.A						DWG NO.	A4
				MATERIAL		5	
				WEIGHT:		SCALE:1:1	SHEET 1 OF 1

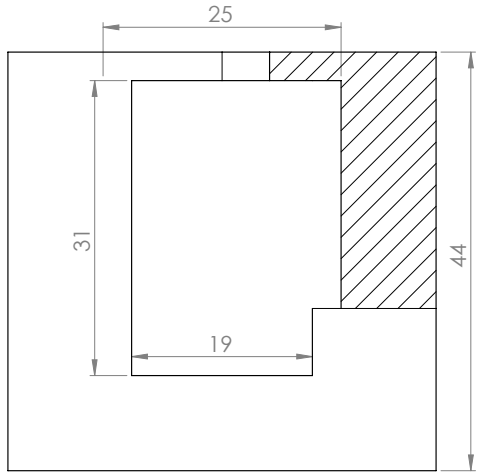
Appendix D

SolidWorks Drawings of robots Head Assembly and its individual parts

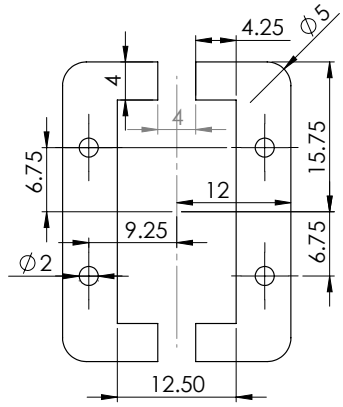


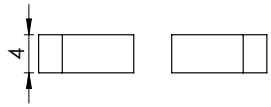
UNLESS OTHERWISE SPECIFIED: DIMENSIONS ARE IN MILLIMETERS SURFACE FINISH: TOLERANCES: LINEAR: ANGULAR:	FINISH:		DEBUR AND BREAK SHARP EDGES	DO NOT SCALE DRAWING	REVISION
	NAME	SIGNATURE	DATE	TITLE:	
DRAWN				Full Neck Assembly	
CHK'D					
APPV'D					
MFG					
Q.A				DWG NO.	A4
			WEIGHT:	SCALE:1:1	SHEET 1 OF 1



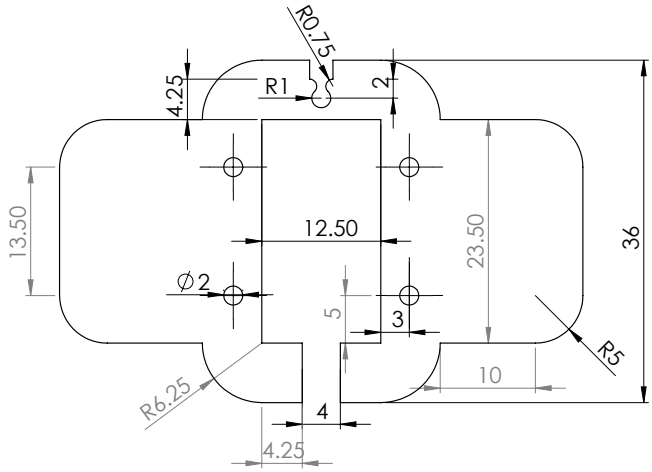


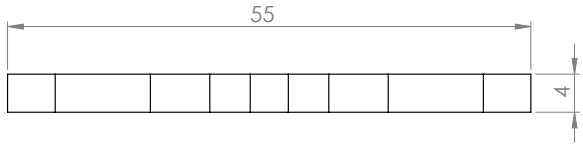
UNLESS OTHERWISE SPECIFIED: DIMENSIONS ARE IN MILLIMETERS SURFACE FINISH: TOLERANCES: LINEAR: ANGULAR:		FINISH:			DEBUR AND BREAK SHARP EDGES	DO NOT SCALE DRAWING	REVISION
	NAME	SIGNATURE	DATE			TITLE:	
DRAWN						Bioloid Body	
CHK'D							
APPV'D							
MFG							
Q.A						DWG NO.	A4
						1	
				WEIGHT:		SCALE:2:1	SHEET 1 OF 1



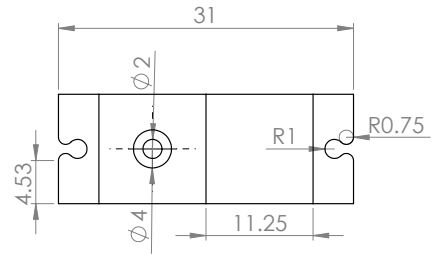


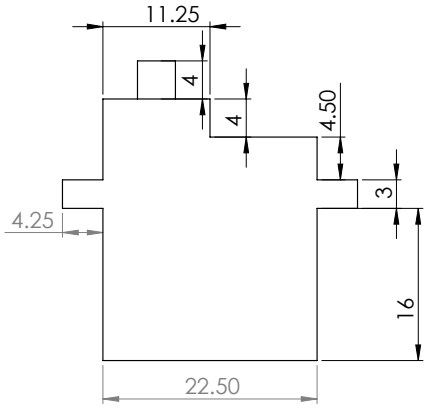
UNLESS OTHERWISE SPECIFIED: DIMENSIONS ARE IN MILLIMETERS SURFACE FINISH: TOLERANCES: LINEAR: ANGULAR:		FINISH:			DEBUR AND BREAK SHARP EDGES	DO NOT SCALE DRAWING	REVISION
	NAME	SIGNATURE	DATE				
DRAWN						TITLE: Yaw Servo Anchor	
CHK'D							
APPV'D							
MFG							
Q.A						DWG NO. 2	A4
				MATERIAL			
				WEIGHT:		SCALE:2:1	SHEET 1 OF 1



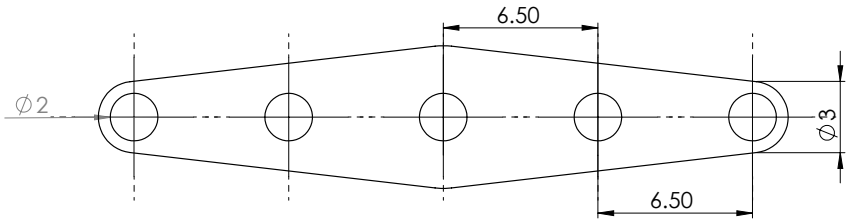


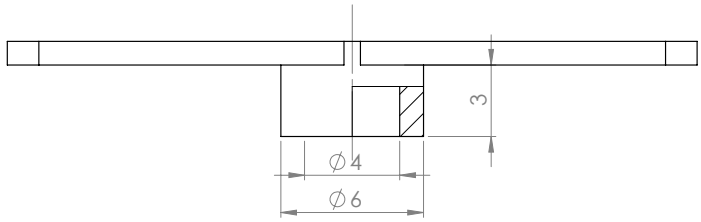
UNLESS OTHERWISE SPECIFIED: DIMENSIONS ARE IN MILLIMETERS SURFACE FINISH: TOLERANCES: LINEAR: ANGULAR:	FINISH:		DEBUR AND BREAK SHARP EDGES	DO NOT SCALE DRAWING	REVISION	
	NAME	SIGNATURE	DATE	TITLE:		
DRAWN				Yaw Servo Support		
CHK'D						
APPV'D						
MFG						
Q.A				DWG NO.		A4
				3		
			WEIGHT:	SCALE:2:1	SHEET 1 OF 1	



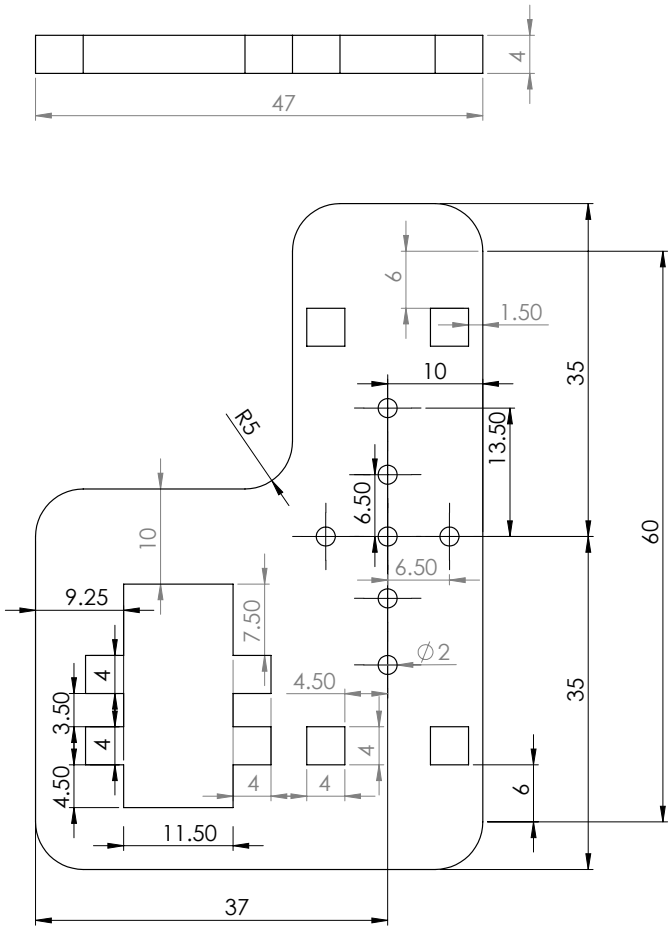


UNLESS OTHERWISE SPECIFIED: DIMENSIONS ARE IN MILLIMETERS SURFACE FINISH: TOLERANCES: LINEAR: ANGULAR:		FINISH:			DEBUR AND BREAK SHARP EDGES	DO NOT SCALE DRAWING	REVISION
	NAME	SIGNATURE	DATE			TITLE:	
DRAWN						RB-90 Yaw Servo	
CHK'D							
APPV'D							
MFG							
Q.A				MATERIAL		DWG NO. 4	A4
				WEIGHT:		SCALE:2:1	SHEET 1 OF 1

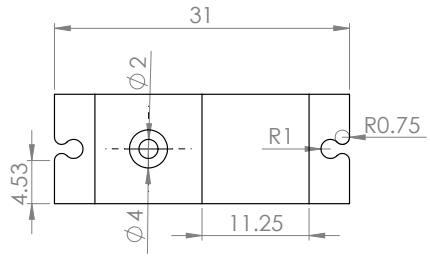


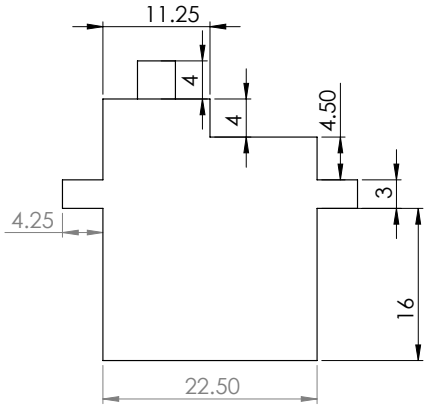


UNLESS OTHERWISE SPECIFIED: DIMENSIONS ARE IN MILLIMETERS SURFACE FINISH: TOLERANCES: LINEAR: ANGULAR:		FINISH:			DEBUR AND BREAK SHARP EDGES	DO NOT SCALE DRAWING	REVISION
	NAME	SIGNATURE	DATE			TITLE:	
DRAWN						Full Servo Connection	
CHK'D							
APPV'D							
MFG							
Q.A						DWG NO. 5	A4
				WEIGHT:		SCALE:5:1	SHEET 1 OF 1

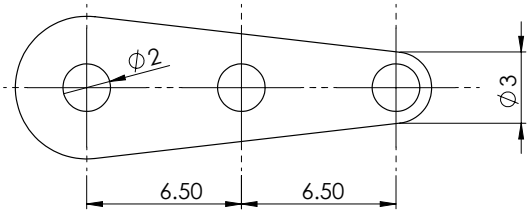


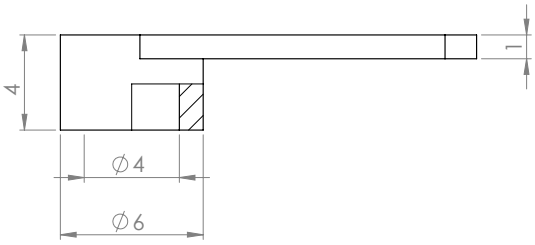
UNLESS OTHERWISE SPECIFIED: DIMENSIONS ARE IN MILLIMETERS SURFACE FINISH: TOLERANCES: LINEAR: ANGULAR:	FINISH:		DEBUR AND BREAK SHARP EDGES	DO NOT SCALE DRAWING	REVISION
	NAME	SIGNATURE	DATE	TITLE:	
DRAWN				Head Base	
CHK'D					
APPV'D					
MFG					
Q.A				DWG NO. 6	A4
			MATERIAL		
			WEIGHT:	SCALE:2:1	SHEET 1 OF 1



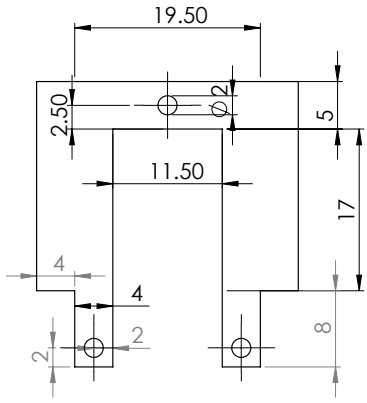


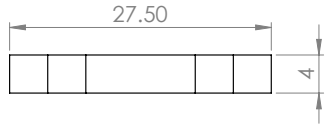
UNLESS OTHERWISE SPECIFIED: DIMENSIONS ARE IN MILLIMETERS SURFACE FINISH: TOLERANCES: LINEAR: ANGULAR:		FINISH:			DEBUR AND BREAK SHARP EDGES	DO NOT SCALE DRAWING	REVISION
	NAME	SIGNATURE	DATE			TITLE:	
DRAWN						RB-90 Pitch Servo	
CHK'D							
APPV'D							
MFG							
Q.A				MATERIAL		DWG NO. 7	A4
				WEIGHT:		SCALE:2:1	SHEET 1 OF 1



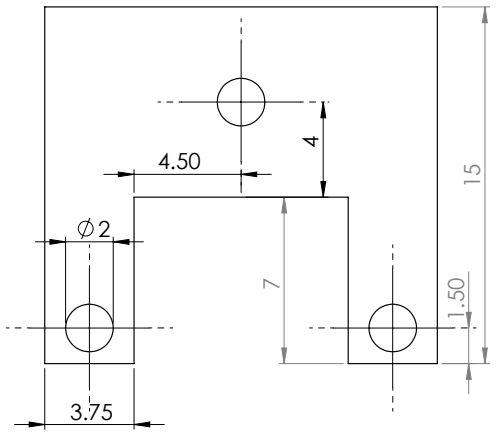


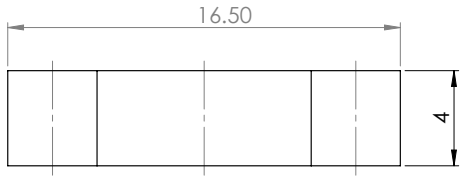
UNLESS OTHERWISE SPECIFIED: DIMENSIONS ARE IN MILLIMETERS SURFACE FINISH: TOLERANCES: LINEAR: ANGULAR:	FINISH:		DEBUR AND BREAK SHARP EDGES	DO NOT SCALE DRAWING	REVISION	
	NAME	SIGNATURE	DATE	TITLE:		
DRAWN				Half Servo Connection		
CHK'D						
APPV'D						
MFG						
Q.A				DWG NO.	8	A4
				WEIGHT:	SCALE:5:1	SHEET 1 OF 1



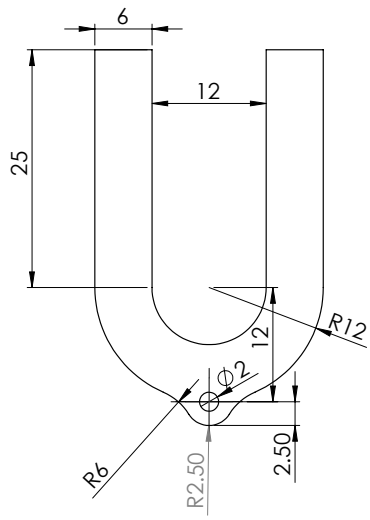


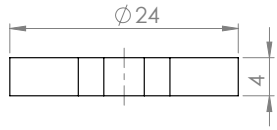
UNLESS OTHERWISE SPECIFIED: DIMENSIONS ARE IN MILLIMETERS SURFACE FINISH: TOLERANCES: LINEAR: ANGULAR:		FINISH:			DEBUR AND BREAK SHARP EDGES	DO NOT SCALE DRAWING	REVISION
	NAME	SIGNATURE	DATE			TITLE:	
DRAWN						Pitch Servo Anchor	
CHK'D							
APPV'D							
MFG							
Q.A						DWG NO. 9	A4
				WEIGHT:		SCALE:2:1	SHEET 1 OF 1



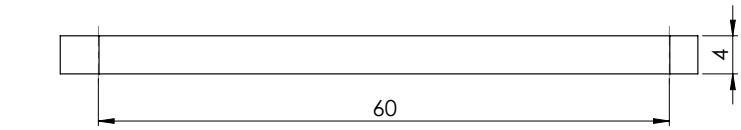


UNLESS OTHERWISE SPECIFIED: DIMENSIONS ARE IN MILLIMETERS SURFACE FINISH: TOLERANCES: LINEAR: ANGULAR:	FINISH:		DEBUR AND BREAK SHARP EDGES	DO NOT SCALE DRAWING	REVISION	
	NAME	SIGNATURE	DATE	TITLE:		
DRAWN				Phone Anchor		
CHK'D						
APPV'D						
MFG						
Q.A				DWG NO.	10	A4
				WEIGHT:	SCALE:5:1	SHEET 1 OF 1

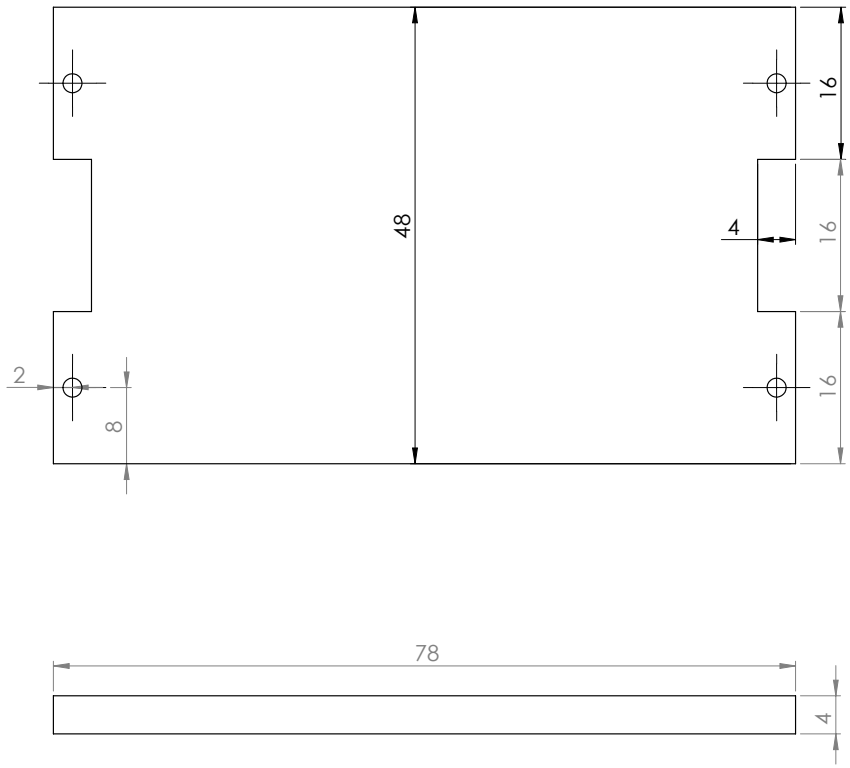




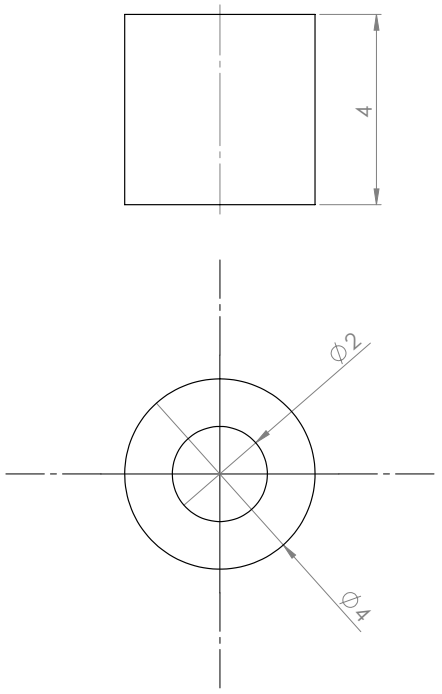
UNLESS OTHERWISE SPECIFIED: DIMENSIONS ARE IN MILLIMETERS SURFACE FINISH: TOLERANCES: LINEAR: ANGULAR:	FINISH:		DEBUR AND BREAK SHARP EDGES	DO NOT SCALE DRAWING	REVISION	
	NAME	SIGNATURE	DATE	TITLE:		
DRAWN				Phone Support		
CHK'D						
APPV'D						
MFG						
Q.A				DWG NO.		A4
				11		
			WEIGHT:	SCALE:2:1	SHEET 1 OF 1	



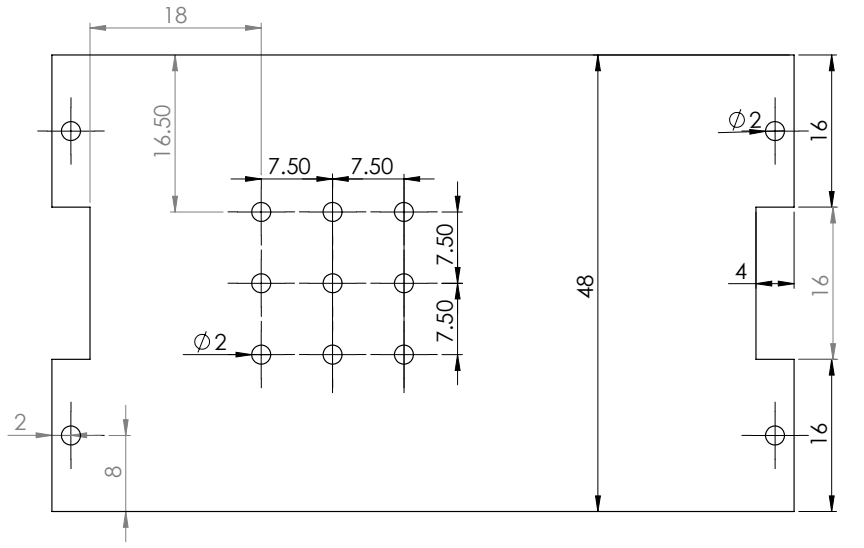
UNLESS OTHERWISE SPECIFIED: DIMENSIONS ARE IN MILLIMETERS SURFACE FINISH: TOLERANCES: LINEAR: ANGULAR:			FINISH:			DEBUR AND BREAK SHARP EDGES	DO NOT SCALE DRAWING	REVISION
	NAME	SIGNATURE	DATE				TITLE:	
DRAWN							Pitch Support	
CHK'D								
APPV'D								
MFG								
Q.A							DWG NO.	A4
							12	
				WEIGHT:			SCALE:2:1	SHEET 1 OF 1

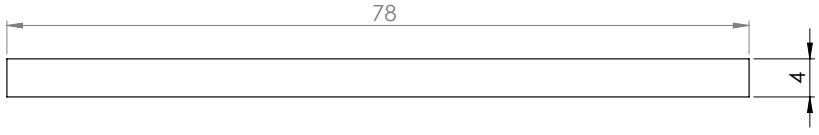


UNLESS OTHERWISE SPECIFIED: DIMENSIONS ARE IN MILLIMETERS SURFACE FINISH: TOLERANCES: LINEAR: ANGULAR:	FINISH:		DEBUR AND BREAK SHARP EDGES	DO NOT SCALE DRAWING	REVISION
	NAME	SIGNATURE	DATE	TITLE:	
DRAWN				Phone Support Front	
CHK'D					
APPV'D					
MFG					
Q.A				DWG NO. 13	A4
			WEIGHT:	SCALE:2:1	SHEET 1 OF 1

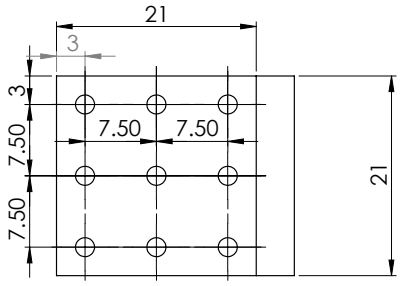


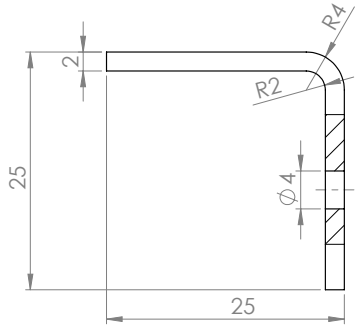
UNLESS OTHERWISE SPECIFIED: DIMENSIONS ARE IN MILLIMETERS SURFACE FINISH: TOLERANCES: LINEAR: ANGULAR:	FINISH:		DEBUR AND BREAK SHARP EDGES	DO NOT SCALE DRAWING	REVISION	
	NAME	SIGNATURE	DATE	TITLE:		
DRAWN				Separator		
CHK'D						
APPV'D						
MFG						
Q.A				DWG NO.		A4
				14		
				WEIGHT:	SCALE:10:1	SHEET 1 OF 1





UNLESS OTHERWISE SPECIFIED: DIMENSIONS ARE IN MILLIMETERS SURFACE FINISH: TOLERANCES: LINEAR: ANGULAR:	FINISH:		DEBUR AND BREAK SHARP EDGES	DO NOT SCALE DRAWING	REVISION	
	NAME	SIGNATURE	DATE	TITLE:		
DRAWN				Phone Support Back		
CHK'D						
APPV'D						
MFG						
Q.A				DWG NO.		A4
				15		
			WEIGHT:	SCALE:2:1	SHEET 1 OF 1	





UNLESS OTHERWISE SPECIFIED: DIMENSIONS ARE IN MILLIMETERS SURFACE FINISH: TOLERANCES: LINEAR: ANGULAR:	FINISH:		DEBUR AND BREAK SHARP EDGES	DO NOT SCALE DRAWING	REVISION
	NAME	SIGNATURE	DATE	TITLE:	
DRAWN				L Piece	
CHK'D					
APPV'D					
MFG					
Q.A				DWG NO. 16	A4
			WEIGHT:	SCALE:2:1	SHEET 1 OF 1

Appendix E

First Processing sketch: Kinect and IK

```
/*
Script that sets up the Kinect depth sensor and pointcloud. Thne deals with the tracking of the users skeleton, the procurement of the wrist positions from the Kinect and the application of inverse kinematics to it. Additionally it receives the Joystick data from the third Processing script and sends it, plus the Inverse Kinematics data, to the second Processing program, all done via OSC messaging.
*/

//Import libraries
import SimpleOpenNI.*;
import oscP5.*;
import netP5.*;

OscP5 oscP5Location1; //Used to share information between processing sketches
NetAddress location2;
SimpleOpenNI context; //Declare global object to access the camera


int           J1X, J1Y, J2X, J2Y;
float         zoomF =0.5f;
float         rotX = radians(180);  // By default rotate the hole scene 180deg around the x-axis,
                                    // because the data from openni comes upside down
```

```
19 float          rotY = radians(0);
   boolean        autoCalib=true; //autocalibration is done
       automatically
21 float          pi=3.14159;
   PVector        bodyCenter = new PVector(); //To show orientation of
       body
23 PVector        bodyDir = new PVector();
   PVector        com = new PVector();
25 color[]        userClr = new color[]{ color(255,0,0), //If new users
       join they will be diplayed in other colors
                                         color(0,255,0),
27                                       color(0,0,255),
                                         color(255,255,0),
29                                       color(255,0,255),
                                         color(0,255,255)
31                                      };

33 void setup()
   {
35   //Set up OSC communication
     oscP5Location1 = new OscP5(this, 1001);
37   location2 = new NetAddress("127.0.0.1", 2001);

39   //Tells Processing I'm going to draw in 3D
     size(1024,768,P3D);
41
     //Instantiate a new context that communicates with kinect
43   context = new SimpleOpenNI(this);

45   //If context can't initialize
     if(context.isInit() == false)
47   {
        println("Camera not plugged in");
49      exit();
        return;
51   }

53   context.setMirror(false); //Disable mirror
     context.enableDepth(); //Enable depthMap image generation
55   context.enableUser(); //Enable skeleton generation for all joints

57   //Setup for skeleton lines
     stroke(0,0,255); //Blue color only
59   strokeWeight(6);
     background(0,200,0);
61   smooth();
   }
63
   void draw()
65 {
     //Update the camera
67   context.update();

69   //Delete past images to not create a mess on every update
     background(0,0,0);
```

```
71
   //Needed to see the camera input correctly
73 translate(width/2, height/2, 0);
   rotateX(rotX); //Turns output image 180 degrees in the X axis
75 rotateY(rotY);
   scale(zoomF);
77 int[]   depthMap = context.depthMap();
   int[]   userMap = context.userMap();
79 int     steps   = 3;  //To speed up the drawing, draw every third
      point
   int     index;
81 PVector realWorldPoint;
   translate(0,0,-1000);  //Set the rotation center of the scene
     1000
83                        //infront of the camera

85 //Draw the pointcloud
   beginShape(POINTS);
87 for(int y=0;y < context.depthHeight();y+=steps)
   {
89   for(int x=0;x < context.depthWidth();x+=steps)
     {
91     index = x + y * context.depthWidth();
       if(depthMap[index] > 0)
93     {
         //Draw the projected point
95       realWorldPoint = context.depthMapRealWorld()[index];
         if(userMap[index] == 0)
97         stroke(100);
         else
99         stroke(userClr[ (userMap[index] - 1) % userClr.length ]);

101      point(realWorldPoint.x,realWorldPoint.y,realWorldPoint.z);
       }
103  }
   }
105 endShape();

107 //Draw the skeleton if it's available
   int[] userList = context.getUsers();
109 for(int i=0;i<userList.length;i++)
   {
111   if(context.isTrackingSkeleton(userList[i]))
       drawSkeleton(userList[i]);
113
   //Ddraw the center of mass
115 if(context.getCoM(userList[i],com))
   {
117     stroke(100,255,0);
       strokeWeight(1);
119    beginShape(LINES);
         vertex(com.x - 15,com.y,com.z);
121      vertex(com.x + 15,com.y,com.z);

123      vertex(com.x,com.y - 15,com.z);
```

```
        vertex(com.x,com.y + 15,com.z);

        vertex(com.x,com.y,com.z - 15);
        vertex(com.x,com.y,com.z + 15);
      endShape();

      fill(0,255,100);
      text(Integer.toString(userList[i]),com.x,com.y,com.z);
    }
  }

  //Draw the kinect cam
  context.drawCamFrustum();
}

////////////////////////////////////////////////////////////
// Events
////////////////////////////////////////////////////////////

void onNewUser(SimpleOpenNI curContext,int userId)
{
  println("onNewUser - userId: " + userId);

  context.startTrackingSkeleton(userId);
}

void oscEvent(OscMessage theOscMessage) {
  // Obtains data from previous sketch
  J1X = theOscMessage.get(0).intValue();
  J1Y = theOscMessage.get(1).intValue();
  J2X = theOscMessage.get(2).intValue();
  J2Y = theOscMessage.get(3).intValue();
}

////////////////////////////////////////////////////////////
// Functions
////////////////////////////////////////////////////////////
void drawSkeleton(int userId)
{
  strokeWeight(6);

  // to get the 3d joint data
  drawLimb(userId, SimpleOpenNI.SKEL_HEAD, SimpleOpenNI.SKEL_NECK);
  drawLimb(userId, SimpleOpenNI.SKEL_NECK, SimpleOpenNI.
    SKEL_LEFT_SHOULDER);
  drawLimb(userId, SimpleOpenNI.SKEL_LEFT_SHOULDER, SimpleOpenNI.
    SKEL_LEFT_ELBOW);
  drawLimb(userId, SimpleOpenNI.SKEL_LEFT_ELBOW, SimpleOpenNI.
    SKEL_LEFT_HAND);
  drawLimb(userId, SimpleOpenNI.SKEL_NECK, SimpleOpenNI.
    SKEL_RIGHT_SHOULDER);
  drawLimb(userId, SimpleOpenNI.SKEL_RIGHT_SHOULDER, SimpleOpenNI.
    SKEL_RIGHT_ELBOW);
  drawLimb(userId, SimpleOpenNI.SKEL_RIGHT_ELBOW, SimpleOpenNI.
    SKEL_RIGHT_HAND);
```

```
173   drawLimb(userId, SimpleOpenNI.SKEL_LEFT_SHOULDER, SimpleOpenNI.
        SKEL_TORSO);
      drawLimb(userId, SimpleOpenNI.SKEL_RIGHT_SHOULDER, SimpleOpenNI.
        SKEL_TORSO);
175
      //Every time we draw the Skeleton, we calculate the inv
        kinematics
177   RetrieveJointPositions(userId);

179   //Draw body direction
      getBodyDirection(userId,bodyCenter,bodyDir);
181   bodyDir.mult(200);  // 200mm length
      bodyDir.add(bodyCenter);
183   stroke(255,255,255); //white
      line(bodyCenter.x,bodyCenter.y,bodyCenter.z,
185        bodyDir.x ,bodyDir.y,bodyDir.z);
      strokeWeight(3);
187 }

189 void drawLimb(int userId,int jointType1,int jointType2)
    {
191   PVector jointPos1 = new PVector();
      PVector jointPos2 = new PVector();
193   float  confidence;

195   //Draw the joint position
      confidence = context.getJointPositionSkeleton(userId,jointType1,
        jointPos1);
197   confidence = context.getJointPositionSkeleton(userId,jointType2,
        jointPos2);

199   stroke(255,0,0,confidence * 200 + 55);

201   //Creates line from one joint to another
      line(jointPos1.x,jointPos1.y,jointPos1.z,
203        jointPos2.x,jointPos2.y,jointPos2.z);

205   drawJointOrientation(userId,jointType1,jointPos1,50);
    }
207
    void drawJointOrientation(int userId,int jointType,PVector pos,
        float length)
209 {
      //Draw the joint orientation
211   PMatrix3D  orientation = new PMatrix3D();
      float confidence = context.getJointOrientationSkeleton(userId,
        jointType,orientation);
213
      //Nothing to draw, orientation data is useless
215   if(confidence < 0.001f)
        return;
217
      pushMatrix();
219   translate(pos.x,pos.y,pos.z); //Move reference to the joint
        position
```

```
  applyMatrix(orientation); //Set the local coordsys

  //X axis is red
  stroke(255,0,0,confidence * 200 + 55);
  line(0,0,0,
       length,0,0);
  //y axis is green
  stroke(0,255,0,confidence * 200 + 55);
  line(0,0,0,
       0,length,0);
  //Z axis is blue
  stroke(0,0,255,confidence * 200 + 55);
  line(0,0,0,
       0,0,length);

  popMatrix();
}

void getBodyDirection(int userId,PVector centerPoint,PVector dir)
{
  PVector jointL = new PVector();
  PVector jointH = new PVector();
  PVector jointR = new PVector();
  float  confidence;

  //Draw the joint position
  confidence = context.getJointPositionSkeleton(userId,SimpleOpenNI
    .SKEL_LEFT_SHOULDER,jointL);
  confidence = context.getJointPositionSkeleton(userId,SimpleOpenNI
    .SKEL_HEAD,jointH);
  confidence = context.getJointPositionSkeleton(userId,SimpleOpenNI
    .SKEL_RIGHT_SHOULDER,jointR);

  //Take the neck as the center point
  confidence = context.getJointPositionSkeleton(userId,SimpleOpenNI
    .SKEL_NECK,centerPoint);

  PVector up = PVector.sub(jointH,centerPoint);
  PVector left = PVector.sub(jointR,centerPoint);

  dir.set(up.cross(left));
  dir.normalize();
}

////////////////////////////////////////////////
//Retrieving Joint Positions + Inverse Kinematics
////////////////////////////////////////////////
boolean Wristinit=false;
int q1Lisave;
int q2Lisave;
int q3Lisave;
int q1Risave;
int q2Risave;
int q3Risave;
```

```
271 void RetrieveJointPositions(int userId)
{
273   //Obtain joint poisitions in world frame
   PVector jointPosLS = new PVector();
275   context.getJointPositionSkeleton(userId,SimpleOpenNI.
     SKEL_LEFT_SHOULDER,jointPosLS);
   PVector jointPosRS = new PVector();
277   context.getJointPositionSkeleton(userId,SimpleOpenNI.
     SKEL_RIGHT_SHOULDER,jointPosRS);
   PVector jointPosLE = new PVector();
279   context.getJointPositionSkeleton(userId,SimpleOpenNI.
     SKEL_LEFT_ELBOW,jointPosLE);
   PVector jointPosRE = new PVector();
281   context.getJointPositionSkeleton(userId,SimpleOpenNI.
     SKEL_RIGHT_ELBOW,jointPosRE);
   PVector jointPosLH = new PVector();
283   context.getJointPositionSkeleton(userId,SimpleOpenNI.
     SKEL_LEFT_HAND,jointPosLH);
   PVector jointPosRH = new PVector();
285   context.getJointPositionSkeleton(userId,SimpleOpenNI.
     SKEL_RIGHT_HAND,jointPosRH);

287    // Calculate the positions in shoulder frame
   PVector WristL = new PVector();
289   WristL.x=-1*(jointPosLH.x - jointPosLS.x);
   WristL.y=-1*(jointPosLH.y - jointPosLS.y);
291   WristL.z=-1*(jointPosLH.z - jointPosLS.z);

293   PVector WristR = new PVector();
   WristR.x=jointPosRH.x - jointPosRS.x;
295   WristR.y=jointPosRH.y - jointPosRS.y;
   WristR.z=jointPosRH.z - jointPosRS.z;
297
   //Start the inverse kinematics
299   float L1L, L2L, q1L, q2L, q3L, c3L, L1R, L2R, q1R, q2R, q3R, c3R;
   int q1Li, q2Li, q3Li, q1Ri, q2Ri, q3Ri;
301
   // Left arm
303   L1L=sqrt(sq(jointPosLE.x-jointPosLS.x)+sq(jointPosLE.y-jointPosLS
     .y)+sq(jointPosLE.z-jointPosLS.z)); // Calculates length of
     upper arm
   L2L=sqrt(sq(jointPosLH.x-jointPosLE.x)+sq(jointPosLH.y-jointPosLE
     .y)+sq(jointPosLH.z-jointPosLE.z)); // Calculates length of
     forearm
305
   q1L=atan2(WristL.y,WristL.x);
307   c3L=((sq(WristL.x)+sq(WristL.y)+sq(WristL.z)-sq(L1L)-sq(L2L)))/((
     2*L1L*L2L));
   q3L=atan2(sqrt(1-sq(c3L)),c3L);
309
   if (WristL.x>0){ //Inv Kinematics with trigonometry has the
     problem of two possible solutions, this solves it.
311     q2L=atan2(WristL.z,sqrt(sq(WristL.x)+sq(WristL.y)))+atan2(L2L*
     sin(q3L),L1L+L2L*cos(q3L));
   }else
```

```
313   {
        q2L=atan2(WristL.z,-sqrt(sq(WristL.x)+sq(WristL.y)))+atan2(L2L*
        sin(q3L),L1L+L2L*cos(q3L));
315   }

317   q1Li=int(q1L*180/pi); //Convert rad to deg
      q2Li=int(q2L*180/pi);
319   q3Li=int(q3L*180/pi);

321   //Limitations to avoid self collision:
      if (q1Li<-50){
323     q1Li=-50;
      }
325   if (q1Li>50){
         q1Li=50;
327   }
      if (q3Li>70){
329      q3Li=70;
      }
331
      //Right arm
333   L1R=sqrt(sq(jointPosRE.x-jointPosRS.x)+sq(jointPosRE.y-jointPosRS
        .y)+sq(jointPosRE.z-jointPosRS.z)); // Calculates length of
        upper arm
      L2R=sqrt(sq(jointPosRH.x-jointPosRE.x)+sq(jointPosRH.y-jointPosRE
        .y)+sq(jointPosRH.z-jointPosRE.z)); // Calculates length of
        forearm
335
      q1R=atan2(WristR.y,-WristR.z);
337   c3R=((sq(WristR.x)+sq(WristR.y)+sq(WristR.z)-sq(L1R)-sq(L2R)))/((
        2*L1R*L2R));
      q3R=atan2(sqrt(1-sq(c3R)),c3R);
339
      if (WristR.x>0) //Inv Kinematics with trigonometry has the
        problem of two possible solutions, this solves it.
341   {
        q2R=atan2(WristR.z,sqrt(sq(WristR.x)+sq(WristR.y)))-atan2(L2R*
        sin(q3R),L1R+L2R*cos(q3R));
343   }else
      {
345     q2R=atan2(WristR.z,-sqrt(sq(WristR.x)+sq(WristR.y)))-atan2(L2R*
        sin(q3R),L1R+L2R*cos(q3R));
      }
347
      q1Ri=int(q1R*180/pi); //Convert rad to deg
349   q2Ri=int(q2R*180/pi);
      q3Ri=int(q3R*180/pi);
351
      //Limitations to avoid self collision:
353   if (q1Ri<-45){
        q1Ri=-45;
355   }
      if (q1Ri>45){
357      q1Ri=45;
      }
```

```
359
   //Controller for speed and precision of the arm movements
361   float lambda=1;
   if (!Wristinit){
363     q1Lisave=q1Li;
     q2Lisave=q2Li;
365     q3Lisave=q3Li;
     q1Risave=q1Ri;
367     q2Risave=q2Ri;
     q3Risave=q3Ri;
369     Wristinit=true;
     }else {
371       q1Lisave=int((1-lambda)*q1Lisave+lambda*q1Li);
       q2Lisave=int((1-lambda)*q2Lisave+lambda*q2Li);
373       q3Lisave=int((1-lambda)*q3Lisave+lambda*q3Li);

375       q1Risave=int((1-lambda)*q1Risave+lambda*q1Ri);
       q2Risave=int((1-lambda)*q2Risave+lambda*q2Ri);
377       q3Risave=int((1-lambda)*q3Risave+lambda*q3Ri);
     }
379

   println( "q1L:" + q1Li + ", q2L:" + q2Li + ", q3L:" + q3Li + ",
     q1R:" + q1Ri + ", q2R:" + q2Ri + ", q3R:" + q3Ri + ", J1X:" + J
     1X + ", J1Y:" + J1Y + ", J2X:" + J2X + ", J2Y:" + J2Y );
381   ExchangeData(q1Lisave,q2Lisave,q3Lisave, q1Risave, q2Risave, q3
     Risave, J1X, J1Y, J2X, J2Y); //Send data to another processing
     sketch
}
383

////////////////////////////////////////////////////////////
385 //// Data exchange through OSC
////////////////////////////////////////////////////////////
387 void ExchangeData(int q1Li, int q2Li, int q3Li, int q1Ri, int q2Ri,
      int q3Ri, int J1X, int J1Y, int J2X, int J2Y)
{
389    OscMessage myMessage = new OscMessage("/test"); //Creates a
      message with all the variables
    myMessage.add(q1Li);
391    myMessage.add(q2Li);
    myMessage.add(q3Li);
393    myMessage.add(q1Ri);
    myMessage.add(q2Ri);
395    myMessage.add(q3Ri);
    myMessage.add(J1X);
397    myMessage.add(J1Y);
    myMessage.add(J2X);
399    myMessage.add(J2Y);

401    oscP5Location1.send(myMessage, location2); //Sends the message
}
```

Codes/FirstProcessingSketch/FirstProcessingSketch.pde

Appendix F

Second Processing sketch: ZigbeeSender

```
/*
Script that receives IK and Joystick data from the first program
    via OSC and processes it to create servo positional data and
    directional movement commands, respectively, to be sent via
    Zigbee as strings to the CM-510 controller on the robot
*/

//Import libraries
import processing.serial.*;
import oscP5.*;
import netP5.*;

//Used to share information between processing sketches
OscP5 oscP5Location2;
NetAddress location1;

// Create object from Serial class
Serial myPort;

void setup()
{
  //Allow serial communication
  String portName = "COM4";
  myPort = new Serial(this, portName, 57600);

  //Set up OSC communication
  oscP5Location2 = new OscP5(this, 2001);
```

```
}

void draw() {
 // Does nothing
}

void oscEvent(OscMessage theOscMessage) {

  float scalar= 3.25;
  //Obtains data from previous sketch
  int q1L = theOscMessage.get(0).intValue();
  int q2L = theOscMessage.get(1).intValue();
  int q3L = theOscMessage.get(2).intValue();
  int q1R = theOscMessage.get(3).intValue();
  int q2R = theOscMessage.get(4).intValue();
  int q3R = theOscMessage.get(5).intValue();
  int J1X = theOscMessage.get(6).intValue();
  int J1Y = theOscMessage.get(7).intValue();
  int J2X = theOscMessage.get(8).intValue();
  int J2Y = theOscMessage.get(9).intValue();

  //Process Arm Joint position data if Joysticks not active
  if ((J1X < 58 )&&(J1X > 54 )&&(J1Y <58)&&(J1Y > 54)&&(J2X < 58)
    &&(J2X > 54)){
    // Degrees to position for zigbee
    q1L = int(scalar*( (q1L/4 + 160/4) %90) );
    String q1Lz = new String ("2 "+q1L+"\n") ;
    q2L = int(scalar*( (q2L/4 + 150/4) % 90) );
    String q2Lz = new String ("4 "+q2L+"\n") ;
    q3L = int(scalar*( (q3L/4 + 160/4) % 90) );
    String q3Lz = new String ("6 "+q3L+"\n") ;
    q1R = int(scalar*( (q1R/4 + 160/4) % 90) );
    String q1Rz = new String ("1 "+q1R+"\n") ;
    q2R = int(scalar*( (q2R/4 + 160/4) % 90) );
    String q2Rz = new String ("3 "+q2R+"\n") ;
    q3R = int(scalar*( (q3R/4 - 170/4) % 90) );
    String q3Rz = new String ("5 "+q3R+"\n") ;

    //Sends data through serial
    myPort.write(q1Lz);
    myPort.write(q2Lz);
    myPort.write(q3Lz);
    myPort.write(q1Rz);
    myPort.write(q2Rz);
    myPort.write(q3Rz);
    println(q1Lz, q2Lz, q3Lz, q1Rz, q2Rz, q3Rz);

  //Process directional movement
  }else{
    J1X=J1X-56;
    J1Y=J1Y-56;

    if ((J1X>2)&&(J1Y==0)){ //Strafe Right
      String JM = new String ("9 STFR\n");
      myPort.write(JM);
```

```
79          println("JM:" + JM);
80        } else if ((J1X==0)&&(J1Y>2)){ //Backwards
81          String JM = new String ("9 WBAC\n");
82          myPort.write(JM);
83          println("JM:" + JM);
84        } else if ((J1X<-2)&&(J1Y==0)){ //Strafe Left
85          String JM = new String ("9 STFL\n");
86          myPort.write(JM);
87          println("JM:" + JM);
88        } else if ((J1X==0)&&(J1Y<-2)){ //Forward
89          String JM = new String ("9 WFOR\n");
90          myPort.write(JM);
91          println("JM:" + JM);
92        }
93
94        if(J2X==63){ //Joystick 2 pressed against the right -> Right
          turn
95          String JT = new String ("9 TRNR\n");
96          myPort.write(JT);
97          println("JT:" + JT);
98        } else if (J2X==48){ //Joystick 2 pressed against the left ->
          Left turn
99          String JT = new String ("9 TRNL\n");
100         myPort.write(JT);
101         println("JT:" + JT);
102       }
103     }
    }
```

Codes/SecondProcessingSketch/SecondProcessingSketch.pde

Appendix G

Third Processing sketch: PC Xbee and Voice recognition

```
/*
Script that receives raw Joystick data directly from User via Xbee
    communication and processes it before sending it to the first
    program via OSC. Additionally, it recognizes voice and acts
    accordingly, if the correct words are perceived, the Joystick
    values are modified appropriately before being sent. Finally,
    it also receives via OSC the pitch and yaw values of th Oculus
    Rift orientation and after processing them, sends via Xbee to
    the \textit{Arduino} board of the robot.
*/

 //Import the libraries
import guru.ttslib.*;
import processing.serial.*;
import oscP5.*;
import netP5.*;
import voce.*;
import oscP5.*;
import netP5.*;

//Used to share information between processing sketches
OscP5 oscP5Location3;
NetAddress location1;
```

```
Serial Xbee;  //Create object from Serial class
TTS tts; //Speech ssynthesizer

//String comparators
String response;
String stringcomparison1 = "J1X:";
String stringcomparison2 = "J1Y:";
String stringcomparison3 = "J2X:";
String stringcomparison4 = "J2Y:";

int J1X, J1Y, J2X, J2Y;
boolean VR = false;

void setup(){
  String portNameXbee = "COM3";
  Xbee = new Serial(this, portNameXbee, 9600);

  //Define locations for sketch exchange of data
  oscP5Location3 = new OscP5(this, 6001);
  location1 = new NetAddress("127.0.0.1", 1001);

  //Initialize the Voce library
  voce.SpeechInterface.init("libraries/voce-0.9.1/lib", false, true
    ,"libraries/voce-0.9.1/lib/gram","voicecontrols");

  tts = new TTS();
  //Control settings for the voice sound
  tts.setPitch( 150 );
  tts.setPitchRange( 70 );
}

void draw(){
  //If voice recognition is not active, receive Joystick data
    through Xbee
 if (!VR){
   if (voce.SpeechInterface.getRecognizerQueueSize()>0){
        String s = null;
        s = voce.SpeechInterface.popRecognizedString();
        println(s);
        if (s.equals("voice recognition on")){
           respond("voice recognition enabled");
           VR=true;
        }
    }

    if (Xbee.available() > 0) {

      for(int i=0; i<6; i++){
        String Data = Xbee.readStringUntil(10);
        if (Data != null) {
          String First = Data.substring(0, 4);
          String Second = Data.substring(4, 6);
          if (First.equals(stringcomparison1) == true){
             J1X = parseInt(Second);
```

```
70            }
              else if(First.equals(stringcomparison2) == true){
72              J1Y = parseInt(Second);
              }
74            else if(First.equals(stringcomparison3) == true){
                J2X = parseInt(Second);
76            }
              else if(First.equals(stringcomparison4) == true){
78              J2Y = parseInt(Second);
              } else{
80            }
            }
          }
          ExchangeData(J1X, J1Y, J2X, J2Y);
84      }
    }
86  else if (VR){
        ProcessSpeech();
88  }
}
90
// OSC exchange of joystick data
92 void ExchangeData(int J1X, int J1Y, int J2X, int J2Y)
 {
94   println("J1X:" + J1X + ", J1Y:" + J1Y + ", J2X:" + J2X + ", J2Y:
      " + J2Y);
     OscMessage myMessage = new OscMessage("/test"); //Creates a
      message with all the variables
96   myMessage.add(J1X);
     myMessage.add(J1Y);
98   myMessage.add(J2X);
     myMessage.add(J2Y);
100  oscP5Location3.send(myMessage, location1); //Sends the message
     delay(100);
102 }

104 // OSC retrieval of head data and posterior processing
void oscEvent(OscMessage theOscMessage) {
106   float pitch = theOscMessage.get(0).floatValue();
    float yaw = theOscMessage.get(1).floatValue();
108
    //Process data accordingly
110 if (pitch<=90.0){
      pitch = 90.0- pitch;
112 }else{
     pitch= (pitch-360.0)*(-1)+90.0;
114 }
    if (yaw<=90.0){
116   yaw = 90.0- yaw;
    }else{
118  yaw= (yaw-360.0)*(-1)+90.0;
    }
120
    //Limit the turning
122 if (yaw>270){
```

```
    yaw=0;
  } else if (yaw<270 && yaw>180){
    yaw=180;
  }
  if (pitch>270){
    pitch=0;
  }else if (pitch<270 && pitch>180){
    pitch=180;
  }

  String ORP = new String ("ORP:"+(int)pitch+"\n") ;
  String ORY = new String ("ORY:"+(int)yaw+"\n") ;
  Xbee.write(ORP);
  Xbee.write(ORY);
  println(ORP);
  println(ORY);
}

//Voice recognition function
void ProcessSpeech(){
   boolean stop=false;
   //If voce recognizes anything being said
   if (voce.SpeechInterface.getRecognizerQueueSize()>0){
     //Assign the string that voce heard to the variable s
     String input = voce.SpeechInterface.popRecognizedString();
     //Print what was heard to the debug window.
     println("you said: " + input);

     //Compare strings and act accordingly

     //Movements
     if (input.equals("move forward") == true){
        if (AreYouSure()==true){
          J1X=56; J1Y=48; J2X=56; J2Y=56;
          respond("Im moving forward");

        }
     }else if (input.equals("move backward") == true){
        if (AreYouSure()==true){
           J1X=56; J1Y=63; J2X=56; J2Y=56;
           respond("Im moving backward");
        }
     }else if (input.equals("move right") == true){
        if (AreYouSure()==true){
           J1X=63; J1Y=56; J2X=56; J2Y=56;
           respond("Im strafing right");
        }
     }else if (input.equals("move left") == true){
        if (AreYouSure()==true){
           J1X=48; J1Y=56; J2X=56; J2Y=56;
           respond("Im strafing left");
        }
     }else if (input.equals("turn right") == true){
        if (AreYouSure()==true){
           J1X=56; J1Y=56; J2X=63; J2Y=56;
```

```
178             respond("Im turning right");
            }
180         }else if (input.equals("turn left") == true){
            if (AreYouSure()==true){
182             J1X=56; J1Y=56; J2X=48; J2Y=56;
                respond("Im turning left");
184         }
        }else if (input.equals("stop") == true){
186         J1X=56; J1Y=56; J2X=56; J2Y=56;
            respond("Stopping");
188     }else if (input.equals("controls on")){
            respond("controls enabled");
190         VR=false;  //De-activate Voice recognition
        }else {}
192     ExchangeData(J1X, J1Y, J2X, J2Y);
    }
194 }

196 //Voice recognition confirmation function
    boolean AreYouSure() {
198   boolean detect = false;
      boolean yesorno;
200   String s = null;
      respond("Are you sure?");
202   while (!detect){
        delay(200);
204     println("...");
        if (voce.SpeechInterface.getRecognizerQueueSize()>0){
206         s = voce.SpeechInterface.popRecognizedString();
            if (s.equals("yes") | s.equals("no")){
208             detect=true;
                println(s);
210         }
        }
212   }
      if (s.equals("yes")) yesorno=true;
214   else yesorno=false;
      return yesorno;
216 }

218 //Speech synthesizer
    void respond(String input){
220   if (input.length() > 0){  //Check there is something to say
        //Stop listening, otherwise the robot will listen to itself
222     voce.SpeechInterface.setRecognizerEnabled(false);
        //Speech synthesizer
224     tts.speak(input);
        voce.SpeechInterface.setRecognizerEnabled(true);
226   }
    }
```

Codes/ThirdProcessingSketch/ThirdProcessingSketch.pde

Appendix H

Users *Arduino UNO* sketch

```
/*
Arduino sketch of the board mounted on the user that reads the
    analog inputs of Joysticks and Flex sensors, processes them and
     broadcasts them via Xbee to the Computers Xbee and to the
    Robots Arduino board respectively
*/

//Define constants
const int J1X=0; //Analog pin connected to X output
const int J1Y=1; //Analog pin connected to Y output
const int J2X=2; //Analog pin connected to X output
const int J2Y=3; //Analog pin connected to Y output
const int FSL=4; //analog pin 4
const int FSR=5; //analog pin 5

void setup() {
   Serial.begin(9600); //Initialize serial
}

//Joystick value processing function
int treatValue(int data) {
  return (data * 16  / 1024) + 48;
}

void loop() {

  //Read and process Joystick values
  String J1X1 = "J1X:";
  String J1X2 = J1X1 + treatValue(analogRead(J1X));
```

```
27    String J1X3 = J1X2 + "\n";
      Serial.print(J1X3);
29
      String J1Y1 = "J1Y:";
31    String J1Y2 = J1Y1 + treatValue(analogRead(J1Y));
      String J1Y3 = J1Y2 + "\n";
33    Serial.print(J1Y3);

35    String J2X1 = "J2X:";
      String J2X2 = J2X1 + treatValue(analogRead(J2X));
37    String J2X3 = J2X2 + "\n";
      Serial.print(J2X3);
39
      String J2Y1 = "J2Y:";
41    String J2Y2 = J2Y1 + treatValue(analogRead(J2Y));
      String J2Y3 = J2Y2 + "\n";
43    Serial.print(J2Y3);

45    //Read and process Flex Sensor values
      int ValueLeft = analogRead(FSL);
47    int FSLMap = map(ValueLeft, 400, 750, 0, 45);
      String FSL1 = "FSL:";
49    String FSL2 = FSL1 + FSLMap;
      String FSL3 = FSL2 + "\n";
51    Serial.print(FSL3);

53    int ValueRight = analogRead(FSR);
      int FSRMap = map(ValueRight, 400, 750, 0, 45);
55    String FSR1 = "FSR:";
      String FSR2 = FSR1 + FSRMap;
57    String FSR3 = FSR2 + "\n";
      Serial.print(FSR3);
59
      delay(100);
61  }
```

Codes/UsersArduinoSketch/UsersArduinoSketch.ino

Appendix I

Robots *Arduino UNO* sketch

```
/*
Arduino sketch from the board mounted on the robot which receives
    the yaw, pitch and gripper servo data via Xbee, processes it
    and controls the specific servos accordingly.
*/

//Include Servo Library
#include <Servo.h>

//Define Pins
int LeftPin = 11;
int PitchPin = 9;
int YawPin = 6;
int RightPin = 3;

\\Define string comparators
String stringcomparison1 = "FSR:";
String stringcomparison2 = "ORP:";
String stringcomparison3 = "ORY:";
String stringcomparison4 = "FSL:";

//Create Servo Object
Servo RHS;
Servo HP;
Servo HY;
Servo LHS;

```

```
void setup()
{
  //Initialize serial
  Serial.begin(9600);

  //Attach Servo to object
  RHS.attach(RightPin);
  HP.attach(PitchPin);
  HY.attach(YawPin);
  LHS.attach(LeftPin);

  delay(500);
}

void loop()
{
  //Partition and process incoming strings and
  //send information to servomotors
  while( Serial.available()){

    String Data = Serial.readStringUntil(10);
    String First = Data.substring(0, 4);

    if (First.equals(stringcomparison1) == true){
        String SecondR = Data.substring(4);
        char valueArrayR[SecondR.length() + 1];
        SecondR.toCharArray(valueArrayR, sizeof(valueArrayR));
        int valueR = atoi(valueArrayR);
        int ValueR=map(valueR, 0, 45, 180, 0);
        RHS.write(ValueR);
        Serial.println(ValueR);
    }

    else if (First.equals(stringcomparison2) == true){
        String SecondP = Data.substring(4);
        char valueArrayP[SecondP.length() + 1];
        SecondP.toCharArray(valueArrayP, sizeof(valueArrayP));
        int valueP= atoi(valueArrayP);
        HP.write(valueP);
        Serial.println(valueP);
    }

    else if (First.equals(stringcomparison3) == true){
        String SecondY = Data.substring(4);
        char valueArrayY[SecondY.length() + 1];
        SecondY.toCharArray(valueArrayY, sizeof(valueArrayY));
        int valueY = atoi(valueArrayY);
        HY.write(valueY);
        Serial.println(valueY);
    }

    else if (First.equals(stringcomparison4) == true){
        String SecondL = Data.substring(4);
        char valueArrayL[SecondL.length() + 1];
        SecondL.toCharArray(valueArrayL, sizeof(valueArrayL));
```

```
81            int valueL = atoi(valueArrayL);
              int ValueL=map(valueL, 0, 45, 0, 180);
83            LHS.write(ValueL);
              Serial.println(ValueL);
85        }
      }
87  }
```

Codes/RobotsArduinoSketch/RobotsArduinoSketch.ino

Appendix J

Oculus Rift codes

J.1 WWW class script

```
/*Script that renders images from a webpage to a material texture
    in Unity 5.0*/

#pragma strict
//Sets the IP to download images from
var url = "http://157.82.7.2:8080/shot.jpg";

function Start () {
        // Creates a 2D Texture in DXT1 format to render as
    material texture for the specified component
        GetComponent.<Renderer>().material.mainTexture = new
    Texture2D(4, 4, TextureFormat.DXT1, false);
        while(true) {
            // Start a download of the given URL
            var www = new WWW(url);
            // wait until the download is done
            yield www;
            // assign the downloaded image to the main texture of
    the object
            www.LoadImageIntoTexture(GetComponent.<Renderer>().
    material.mainTexture);
        }
}
```

Codes/OculusRift/WWWClass.txt

J.2 OVRCameraRig.cs script

```
/*
Copyright : Copyright 2014 Oculus VR, LLC. All Rights reserved.
Licensed under the Oculus VR Rift SDK License Version 3.2 (the "
    License"); you may not use the Oculus VR Rift SDK except in
    compliance with the License, which is provided at the time of
    installation or download, or which otherwise accompanies this
    software in either electronic or hard copy form.
You may obtain a copy of the License at
http://www.oculusvr.com/licenses/LICENSE-3.2
Unless required by applicable law or agreed to in writing, the
    Oculus VR SDK distributed under the License is distributed on
    an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND,
     either express or implied. See the License for the specific
    language governing permissions and limitations under the
    License.
*/

using System;
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

// A head-tracked stereoscopic virtual reality camera rig.
[ExecuteInEditMode]
public class OVRCameraRig : MonoBehaviour
{
  // The left eye camera.
  public Camera leftEyeCamera { get; private set; }
  // The right eye camera.
  public Camera rightEyeCamera { get; private set; }
  // Provides a root transform for all anchors in tracking space.
  public Transform trackingSpace { get; private set; }
  // Always coincides with the pose of the left eye.
  public Transform leftEyeAnchor { get; private set; }
  // Always coincides with average of the left and right eye poses.
  public Transform centerEyeAnchor { get; private set; }
  // Always coincides with the pose of the right eye.
  public Transform rightEyeAnchor { get; private set; }
  // Always coincides with the pose of the tracker.
  public Transform trackerAnchor { get; private set; }
  // Occurs when the eye pose anchors have been set.
  public event System.Action<OVRCameraRig> UpdatedAnchors;
  private bool needsCameraConfigure = true;
  private readonly string trackingSpaceName = "TrackingSpace";
  private readonly string trackerAnchorName = "TrackerAnchor";
  private readonly string eyeAnchorName = "EyeAnchor";
  private readonly string legacyEyeAnchorName = "Camera";

#region Unity Messages
  private void Awake()
  {
    EnsureGameObjectIntegrity();
```

```
44      if (!Application.isPlaying)
          return;
46      OVRManager.Created += () => { needsCameraConfigure = true; };
        OVRManager.NativeTextureScaleModified += (prev, current) => {
        needsCameraConfigure = true; };
48      OVRManager.VirtualTextureScaleModified += (prev, current) => {
        needsCameraConfigure = true; };
        OVRManager.EyeTextureAntiAliasingModified += (prev, current) =>
         { needsCameraConfigure = true; };
50      OVRManager.EyeTextureDepthModified += (prev, current) => {
        needsCameraConfigure = true; };
        OVRManager.EyeTextureFormatModified += (prev, current) => {
        needsCameraConfigure = true; };
52      OVRManager.MonoscopicModified += (prev, current) => {
        needsCameraConfigure = true; };
        OVRManager.HdrModified += (prev, current) => {
        needsCameraConfigure = true; };
54    }
      private void Start()
56    {
        EnsureGameObjectIntegrity();
58      OSCHandler.Instance.Init();
        if (!Application.isPlaying)
60        return;
        UpdateCameras();
62      UpdateAnchors();
      }
64
#if !UNITY_ANDROID || UNITY_EDITOR
66    private void LateUpdate()
#else
68    private void Update()
#endif
70    {
        EnsureGameObjectIntegrity();
72
        if (!Application.isPlaying)
74        return;
        UpdateCameras();
76      UpdateAnchors();
      }
78
#endregion
80    private void UpdateAnchors()
      {
82      bool monoscopic = OVRManager.instance.monoscopic;

84      OVRPose tracker = OVRManager.tracker.GetPose(0f);
        OVRPose hmdLeftEye = OVRManager.display.GetEyePose(OVREye.Left)
        ;
86      OVRPose hmdRightEye = OVRManager.display.GetEyePose(OVREye.
        Right);

88      trackerAnchor.localRotation = tracker.orientation;
```

```
    centerEyeAnchor.localRotation = hmdLeftEye.orientation; //
    using left eye for now
90
    leftEyeAnchor.localRotation = monoscopic ? centerEyeAnchor.
    localRotation : hmdLeftEye.orientation;
92  rightEyeAnchor.localRotation = monoscopic ? centerEyeAnchor.
    localRotation : hmdRightEye.orientation;

94  trackerAnchor.localPosition = tracker.position;
    centerEyeAnchor.localPosition = 0.5f * (hmdLeftEye.position +
    hmdRightEye.position);
96  leftEyeAnchor.localPosition = monoscopic ? centerEyeAnchor.
    localPosition : hmdLeftEye.position;
    rightEyeAnchor.localPosition = monoscopic ? centerEyeAnchor.
    localPosition : hmdRightEye.position;
98
    if (UpdatedAnchors != null)
100 {
      UpdatedAnchors(this);
102 }

104 List<object> values = new List<object>();
    values.AddRange(new object[]{leftEyeAnchor.localRotation.
    eulerAngles.x, leftEyeAnchor.localRotation.eulerAngles.y,
    leftEyeAnchor.localRotation.eulerAngles.z});
106
    //Location to which to send the yaw and pitch values through
    OSC
108 OSCHandler.Instance.SendMessageToClient("Unity", "157.82.5.240"
    , values);
  }
110 private void UpdateCameras()
  {
112   if (!OVRManager.instance.isVRPresent)
        return;
114
      if (needsCameraConfigure)
116   {
        leftEyeCamera = ConfigureCamera(OVREye.Left);
118     rightEyeCamera = ConfigureCamera(OVREye.Right);

120 #if !UNITY_ANDROID || UNITY_EDITOR
        needsCameraConfigure = false;
122 #endif
      }
124 }

126 public void EnsureGameObjectIntegrity()
  {
128   if (trackingSpace == null)
        trackingSpace = ConfigureRootAnchor(trackingSpaceName);
130   if (leftEyeAnchor == null)
        leftEyeAnchor = ConfigureEyeAnchor(trackingSpace, OVREye.Left
      );
132   if (centerEyeAnchor == null)
```

```
      centerEyeAnchor = ConfigureEyeAnchor(trackingSpace, OVREye.
    Center);
    if (rightEyeAnchor == null)
      rightEyeAnchor = ConfigureEyeAnchor(trackingSpace, OVREye.
    Right);
    if (trackerAnchor == null)
      trackerAnchor = ConfigureTrackerAnchor(trackingSpace);
    if (leftEyeCamera == null)
    {
      leftEyeCamera = leftEyeAnchor.GetComponent<Camera>();
      if (leftEyeCamera == null)
      {
        leftEyeCamera = leftEyeAnchor.gameObject.AddComponent<
    Camera>();
      }
#if UNITY_ANDROID && !UNITY_EDITOR
      if (leftEyeCamera.GetComponent<OVRPostRender>() == null)
      {
        leftEyeCamera.gameObject.AddComponent<OVRPostRender>();
      }
#endif
    }
    if (rightEyeCamera == null)
    {
      rightEyeCamera = rightEyeAnchor.GetComponent<Camera>();
      if (rightEyeCamera == null)
      {
        rightEyeCamera = rightEyeAnchor.gameObject.AddComponent<
    Camera>();
      }
#if UNITY_ANDROID && !UNITY_EDITOR
      if (rightEyeCamera.GetComponent<OVRPostRender>() == null)
      {
        rightEyeCamera.gameObject.AddComponent<OVRPostRender>();
      }
#endif
    }
  }
  private Transform ConfigureRootAnchor(string name)
  {
    Transform root = transform.Find(name);

    if (root == null)
    {
      root = new GameObject(name).transform;
    }

    root.parent = transform;
    root.localScale = Vector3.one;
    root.localPosition = Vector3.zero;
    root.localRotation = Quaternion.identity;

    return root;
  }
```

```
184   private Transform ConfigureEyeAnchor(Transform root, OVREye eye)
      {
186     string name = eye.ToString() + eyeAnchorName;
        Transform anchor = transform.Find(root.name + "/" + name);
188
        if (anchor == null)
190     {
          anchor = transform.Find(name);
192     }

194     if (anchor == null)
        {
196       string legacyName = legacyEyeAnchorName + eye.ToString();
          anchor = transform.Find(legacyName);
198     }

200     if (anchor == null)
        {
202       anchor = new GameObject(name).transform;
        }
204
        anchor.name = name;
206     anchor.parent = root;
        anchor.localScale = Vector3.one;
208     anchor.localPosition = Vector3.zero;
        anchor.localRotation = Quaternion.identity;
210
        return anchor;
212   }

214   private Transform ConfigureTrackerAnchor(Transform root)
      {
216     string name = trackerAnchorName;
        Transform anchor = transform.Find(root.name + "/" + name);
218
        if (anchor == null)
220     {
          anchor = new GameObject(name).transform;
222     }

224     anchor.parent = root;
        anchor.localScale = Vector3.one;
226     anchor.localPosition = Vector3.zero;
        anchor.localRotation = Quaternion.identity;
228
        return anchor;
230   }

232   private Camera ConfigureCamera(OVREye eye)
      {
234     Transform anchor = (eye == OVREye.Left) ? leftEyeAnchor :
        rightEyeAnchor;
        Camera cam = anchor.GetComponent<Camera>();
236
```

```
    OVRDisplay.EyeRenderDesc eyeDesc = OVRManager.display.
    GetEyeRenderDesc(eye);
238
    cam.fieldOfView = eyeDesc.fov.y;
240 cam.aspect = eyeDesc.resolution.x / eyeDesc.resolution.y;
    cam.targetTexture = OVRManager.display.GetEyeTexture(eye);
242 cam.hdr = OVRManager.instance.hdr;

244 #if UNITY_ANDROID && !UNITY_EDITOR
    // Enforce camera render order
246 cam.depth = (eye == OVREye.Left) ?
        (int)RenderEventType.LeftEyeEndFrame :
248     (int)RenderEventType.RightEyeEndFrame;

250 /* If we don't clear the color buffer with a glClear, tiling
    GPUs will be forced to do an "unresolve" and read back the
    color buffer information. The clear is free on PowerVR, and
    possibly Mali, but it is a performance cost on Adreno, and we
    would be better off if we had the ability to discard/invalidate
     the color buffer instead of clearing.
    NOTE: The color buffer is not being invalidated in skybox mode,
     forcing an additional,
252 */ wasted color buffer read before the skybox is drawn.
    bool hasSkybox = ((cam.clearFlags == CameraClearFlags.Skybox)
    &&
254                   ((cam.gameObject.GetComponent<Skybox>() !=
    null) || (RenderSettings.skybox != null)));
    cam.clearFlags = (hasSkybox) ? CameraClearFlags.Skybox :
    CameraClearFlags.SolidColor;
256 #else
    cam.rect = new Rect(0f, 0f, OVRManager.instance.
    virtualTextureScale, OVRManager.instance.virtualTextureScale);
258 #endif

260 // When rendering monoscopic, we will use the left camera
    render for both eyes.
    if (eye == OVREye.Right)
262 {
      cam.enabled = !OVRManager.instance.monoscopic;
264 }

266 // AA is documented to have no effect in deferred, but it
    causes black screens.
    if (cam.actualRenderingPath == RenderingPath.DeferredLighting)
268   OVRManager.instance.eyeTextureAntiAliasing = OVRManager.
    RenderTextureAntiAliasing._1;

270 return cam;
  }
272 }
```

Codes/OculusRift/OVRCameraRig.cs

J.3 OSCHandler.cs script

```
/*
UnityOSC - Open Sound Control interface for the Unity3d game engine

Copyright (c) 2012 Jorge Garcia Martin

Permission is hereby granted, free of charge, to any person
    obtaining a copy of this software and associated documentation
    files (the "Software"), to deal in the Software without
    restriction, including without limitation
the rights to use, copy, modify, merge, publish, distribute,
    sublicense, and/or sell copies of the Software, and to permit
    persons to whom the Software is furnished to do so, subject to
    the following conditions:

The above copyright notice and this permission notice shall be
    included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND,
    EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES
     OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND
    NONINFRINGEMENT. IN NO EVENT SHALL
THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES
    OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR
    OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE
    SOFTWARE OR THE USE OR OTHER DEALINGs IN THE SOFTWARE.

Inspired by http://www.unifycommunity.com/wiki/index.php?title=
    AManagerClass
*/

using System;
using System.Net;
using System.Collections.Generic;

using UnityEngine;
using UnityOSC;

 // Models a log of a server composed by an OSCServer, a List of
    OSCPacket and a List of strings that represent the current
    messages in the log.
public struct ServerLog
{
  public OSCServer server;
  public List<OSCPacket> packets;
  public List<string> log;
}

// Models a log of a client composed by an OSCClient, a List of
    OSCMessage and a List of strings that represent the current
    messages in the log.
 public struct ClientLog
{
```

```
    public OSCClient client;
36  public List<OSCMessage> messages;
    public List<string> log;
38 }

40  // Handles all the OSC servers and clients of the current Unity
      game/application. Tracks incoming and outgoing messages.
   public class OSCHandler : MonoBehaviour
42 {
    #region Singleton Constructors
44  static OSCHandler()
    {
46  }

48  OSCHandler()
    {
50  }

52  public static OSCHandler Instance
    {
54      get
    {
56          if (_instance == null)
    {
58          _instance = new GameObject ("OSCHandler").AddComponent<
      OSCHandler>();
            }

60
            return _instance;
62      }
    }
64  #endregion

66  #region Member Variables
    private static OSCHandler _instance = null;
68  private Dictionary<string, ClientLog> _clients = new Dictionary<
      string, ClientLog>();
    private Dictionary<string, ServerLog> _servers = new Dictionary<
      string, ServerLog>();

70
    private const int _loglength = 25;
72  #endregion

74  // Initializes the OSC Handler. Here you can create the OSC
      servers and clientes.
    public void Init()
76  {
        //Initialize OSC clients (transmitters)
78      CreateClient("Unity", IPAddress.Parse("157.82.5.240"),
      6001);

80    }

82  #region Properties
    public Dictionary<string, ClientLog> Clients
```

```
84    {
        get
86      {
          return _clients;
88      }
      }
90
      public Dictionary<string, ServerLog> Servers
92    {
        get
94      {
          return _servers;
96      }
      }
98    #endregion

100   #region Methods

102   // Ensure that the instance is destroyed when the game is stopped
        in the Unity editor. Close all the OSC clients and servers
      void OnApplicationQuit()
104   {
        foreach(KeyValuePair<string,ClientLog> pair in _clients)
106     {
          pair.Value.client.Close();
108     }

110     foreach(KeyValuePair<string,ServerLog> pair in _servers)
        {
112       pair.Value.server.Close();
        }
114
        _instance = null;
116   }
      // Creates an OSC Client (sends OSC messages) given an outgoing
        port and address.
118    // <param name="clientId">
      // A <see cref="System.String"/>
120   // </param>
      // <param name="destination">
122   // A <see cref="IPAddress"/>
      // </param>
124   // <param name="port">
      // A <see cref="System.Int32"/>
126   // </param>
      public void CreateClient(string clientId, IPAddress destination,
        int port)
128   {
        ClientLog clientitem = new ClientLog();
130     clientitem.client = new OSCClient(destination, port);
        clientitem.log = new List<string>();
132     clientitem.messages = new List<OSCMessage>();

134     _clients.Add(clientId, clientitem);
```

```
136     // Send test message
        string testaddress = "/test/alive/";
138     OSCMessage message = new OSCMessage(testaddress, destination.
        ToString());
        message.Append(port); message.Append("OK");
140
        _clients[clientId].log.Add(String.Concat(DateTime.UtcNow.
        ToString(),".",
142                                                 FormatMilliseconds(
        DateTime.Now.Millisecond), " : ",
                                                    testaddress," ",
        DataToString(message.Data)));
144     _clients[clientId].messages.Add(message);

146     _clients[clientId].client.Send(message);
    }
148 // Creates an OSC Server (listens to upcoming OSC messages) given
       an incoming port.
    // <param name="serverId">
150 // A <see cref="System.String"/>
    // </param>
152 // <param name="port">
    // A <see cref="System.Int32"/>
154 // </param>
    public void CreateServer(string serverId, int port)
156 {
          OSCServer server = new OSCServer(port);
158       server.PacketReceivedEvent += OnPacketReceived;

160       ServerLog serveritem = new ServerLog();
          serveritem.server = server;
162   serveritem.log = new List<string>();
      serveritem.packets = new List<OSCPacket>();
164
      _servers.Add(serverId, serveritem);
166 }

168   void OnPacketReceived(OSCServer server, OSCPacket packet)
      {
170   }
    // Sends an OSC message to a specified client, given its clientId
       (defined at the OSC client construction), OSC address and a
      single value. Also updates the client log.
172 // <param name="clientId">
    // A <see cref="System.String"/>
174 // </param>
    // <param name="address">
176 // A <see cref="System.String"/>
    // </param>
178 // <param name="value">
    // A <see cref="T"/>
180 // </param>
    public void SendMessageToClient<T>(string clientId, string
      address, T value)
182 {
```

```
    List<object> temp = new List<object>();
184   temp.Add(value);

186   SendMessageToClient(clientId, address, temp);
    }
188 // Sends an OSC message to a specified client, given its clientId
      (defined at the OSC client construction), OSC address and a
      list of values. Also updates the client log.
    // <param name="clientId">
190 // A <see cref="System.String"/>
    // </param>
192 // <param name="address">
    // A <see cref="System.String"/>
194 // </param>
    // <param name="values">
196 // A <see cref="List<T>"/>
    // </param>
198 public void SendMessageToClient<T>(string clientId, string
      address, List<T> values)
    {
200   if(_clients.ContainsKey(clientId))
      {
202     OSCMessage message = new OSCMessage(address);

204     foreach(T msgvalue in values)
        {
206       message.Append(msgvalue);
        }
208
        if(_clients[clientId].log.Count < _loglength)
210     {
          _clients[clientId].log.Add(String.Concat(DateTime.UtcNow.
      ToString(),".",
212                                                   FormatMilliseconds
      (DateTime.Now.Millisecond),
                                                      " : ", address, "
      ", DataToString(message.Data)));
214       _clients[clientId].messages.Add(message);
        }
216     else
        {
218       _clients[clientId].log.RemoveAt(0);
          _clients[clientId].messages.RemoveAt(0);
220
          _clients[clientId].log.Add(String.Concat(DateTime.UtcNow.
      ToString(),".",
222                                                   FormatMilliseconds
      (DateTime.Now.Millisecond),
                                                      " : ", address, "
      ", DataToString(message.Data)));
224       _clients[clientId].messages.Add(message);
        }
226
        _clients[clientId].client.Send(message);
228   }
```

```
    else
    {
      Debug.LogError(string.Format("Can't send OSC messages to {0}.
     Client doesn't exist.", clientId));
    }
  }

  // Updates clients and servers logs.
  public void UpdateLogs()
  {
    foreach(KeyValuePair<string,ServerLog> pair in _servers)
    {
      if(_servers[pair.Key].server.LastReceivedPacket != null)
      {
        //Initialization for the first packet received
        if(_servers[pair.Key].log.Count == 0)
        {
          _servers[pair.Key].packets.Add(_servers[pair.Key].server.
  LastReceivedPacket);

          _servers[pair.Key].log.Add(String.Concat(DateTime.UtcNow.
  ToString(), ".",

  FormatMilliseconds(DateTime.Now.Millisecond)," : ",
                                                    _servers[pair.
  Key].server.LastReceivedPacket.Address," ",
                                                    DataToString(
  _servers[pair.Key].server.LastReceivedPacket.Data)));
          break;
        }

        if(_servers[pair.Key].server.LastReceivedPacket.TimeStamp
           != _servers[pair.Key].packets[_servers[pair.Key].packets
  .Count - 1].TimeStamp)
        {
          if(_servers[pair.Key].log.Count > _loglength - 1)
          {
            _servers[pair.Key].log.RemoveAt(0);
            _servers[pair.Key].packets.RemoveAt(0);
          }

          _servers[pair.Key].packets.Add(_servers[pair.Key].server.
  LastReceivedPacket);

          _servers[pair.Key].log.Add(String.Concat(DateTime.UtcNow.
  ToString(), ".",

  FormatMilliseconds(DateTime.Now.Millisecond)," : ",
                                                    _servers[pair.
  Key].server.LastReceivedPacket.Address," ",
                                                    DataToString(
  _servers[pair.Key].server.LastReceivedPacket.Data)));
        }
      }
    }
```

```
272   }
      // Converts a collection of object values to a concatenated
        string.
274   // <param name="data">
      // A <see cref="List<System.Object>"/>
276   // </param>
      // <returns>
278   // A <see cref="System.String"/>
      // </returns>
280   private string DataToString(List<object> data)
      {
282     string buffer = "";

284     for(int i = 0; i < data.Count; i++)
        {
286       buffer += data[i].ToString() + " ";
        }
288
        buffer += "\n";
290
        return buffer;
292   }
      // Formats a milliseconds number to a 000 format. E.g. given 50,
        it outputs 050. Given 5, it outputs 005
294   // <param name="milliseconds">
      // A <see cref="System.Int32"/>
296   // </param>
      // <returns>
298   // A <see cref="System.String"/>
      // </returns>
300   private string FormatMilliseconds(int milliseconds)
      {
302     if(milliseconds < 100)
        {
304       if(milliseconds < 10)
            return String.Concat("00",milliseconds.ToString());
306
          return String.Concat("0",milliseconds.ToString());
308     }

310     return milliseconds.ToString();
      }
312
    #endregion
314 }
```

Codes/OculusRift/OSCHandler.cs

Appendix K

Bioloids CM-510 codes

This Appendix will show some of the most important snippets of the code run by the CM-510 controller on the robot. Displaying the whole code on this report is troublesome due to visualization reasons.

K.1 Main program

```
#define F_CPU 16000000UL  // CM-510 runs at 16 MHz.
#include <avr/io.h>
#include <stdio.h>
#include <util/delay.h>
#include <avr/interrupt.h>
#include <math.h>
#include <avr/delay.h>
#include "motion_f.h"
#include "dynamixel.h"
#include "zgb_hal.h"
#include "global.h"
#include "clock.h"



#define DEFAULT_PORTNUM 5    // COM5
#define DEFAULT_BAUDNUM 1 // 1Mbitps
#define COMMAND_BALANCE_MP  224
```

```
19
   int main (void){
21
     // Setting the ports to enable wireless communication with ZigBee
23   DDRC  = 0x7F;
     PORTC = 0x7E;
25   PORTD &= ~0x80;
     PORTD &= ~0x20;
27   PORTD |= 0x40;
     zgb_hal_open(0 , 57412); //using the ZigBee and disables the
       serial communication.
29
     // Perform high level initialization of Dynamixel bus and servos
31   dxl_initialize(0, DEFAULT_BAUDNUM);
     // Initialize motion pages and clock
33   motionPageInit();
     clock_init();
35   // enable interrupts
     sei();
37
     // assume initial pose
39   _delay_ms(1000);
     StandUp();
41   _delay_ms(1000);

43   while(1){
       GetData();
45   }
   }
```

Codes/Bioloid/BioloidProgram.c

K.2 GetData() function

```
   // Function inside zgb_hal.c file that receives data through Zigbee
        connection, processes it and if it is ment as arms joint
       positional control, sends them to the corresponding servos, and
        if its meant as movement control, executes the necessary
       motion pages.
2
   void GetData ( void )
4  {
     int armsorlegs;
6    char c1, c2, c3, c4, c5, c6, command[8], mov[4];
     int id, pos;
8
     // nothing to do, go straight back to main loop
10   if (receive_ready == 0)
     {
12     return 0;
```

```
  }

  // we have a new command, get characters
  c1 = zgb_hal_get_queue();
  if( c1 == 0xFF ) c1 = ' ';
  command[0] = c1;
  c2 = zgb_hal_get_queue();
  if( c2 == 0xFF ) c2 = ' ';
  command[1] = c2;
  c3 = zgb_hal_get_queue();
  if( c3 == 0xFF ) c3 = ' ';
  command[2] = c3;
  c4 = zgb_hal_get_queue();
  if( c4 == 0xFF ) c4 = ' ';
  command[3] = c4;
  c5 = zgb_hal_get_queue();
  if( c5 == 0xFF ) c5 = ' ';
  command[4] = c5;
  c6 = zgb_hal_get_queue();
  if( c6 == 0xFF ) c6 = ' ';
  command[5] = c6;
  command[6] = 0x00;  // finish the string


  if (strlen(command)>2){
    char * pch; //temporal hold of individual sections
    pch = strtok (command," "); //Divide string into two parts
    delimited by blank space
    for(int j=0; pch != NULL; j++)
    {
      if(j==0) id=atoi(pch); //Parses first part of string as
    integer number: ID of motor
      if(j==1) {
        strcpy(mov, pch); //Saves second string as is
        pos=atoi(pch);//Parses second part of string as integer
    number: Positional value
      }
      pch = strtok (NULL, " ,.-");
    }

    if (id<7){ //Data is meant for arm control
      armsorlegs=1;
      dxl_write_word(id, 30, 4*pos); // Sends data to servo
    }

    else if (id==9){ //Data is meant for leg control
      armsorlegs=0;
      CompareStrings(mov);
    }
  }
  receive_ready=0; //This value will be changed when the adequate
    interrupt is activated
}
```

Codes/Bioloid/GetData.c

K.3 CompareStrings() function

```
//Function located in the zgb_hal.c file. It compares the strings
    processed by the GetData() function and sees if the information
     matches certain movement commands. If so, it enables said
    movements. After the comparison (and execution of movements, if
     needed) has finished, the buffer used to store the different
    characters from the Zigbee communication must be cleared before
     receiving anything new. This is because the quantity of data
    is enormous and the transfer rate is way greater than how the
    robot can physically move; and if the instructions are not
    cleared every time, the robot would get stuck doing actions
    that should have happened some time back.

void CompareStrings(char input[]){

  _delay_ms(250);
  StandUp();
  _delay_ms(250);

  if (strcmp(input, "WFOR")==0){
    Forwards();
  }else if(strcmp(input, "WBAC")==0){
    Backwards();
  }else if(strcmp(input, "STFR")==0){
    StrafeRight();
  }else if(strcmp(input, "STFL")==0){
    StrafeLeft();
  }else if(strcmp(input, "WFRT")==0){
    ForwardsRight();
  }else if(strcmp(input, "WFLT")==0){
    ForwardsLeft();
  }else if(strcmp(input, "WBRT")==0){
    BackwardsRight();
  }else if(strcmp(input, "WBLT")==0){
    BackwardsLeft();
  }else if(strcmp(input, "TRNR")==0){
    TurnRight();
  }else if(strcmp(input, "TRNL")==0){
    TurnLeft();
  }
  _delay_ms(500);


  //initialize Zgb again (clean buffer hack)
  UDR1 = 0xFF;
  gbZgbBufferHead = 0;
  gbZgbBufferTail = 0;
  return 1;

}
```

Codes/Bioloid/CompareStrings.c

K.4 MotionPageInit() function

```
/* Initialize the motion pages by constructing a table of pointers
    to each page. Motion pages are stored in Flash (PROGMEM)*/
void motionPageInit()
{
  // Motion Page pointer assignment to PROGMEM
  motion_pointer[0] = NULL;
  motion_pointer[1] = (uint8*) &MotionPage1;
  motion_pointer[2] = (uint8*) &MotionPage2;
  motion_pointer[3] = (uint8*) &MotionPage3;

  //       .............

  motion_pointer[226] = (uint8*) &MotionPage226;
  motion_pointer[227] = (uint8*) &MotionPage227;
}
```

Codes/Bioloid/MotionPageInit.c

K.5 UnpackMotion() function

```
// This function (inside the motion.c file) unpacks a motion stored
    in program memory (Flash) in a struct stored in RAM to allow
    execution

void unpackMotion(int StartPage)
{
  uint8 i, s, num_packed_steps;
  uint32 packed_step_values;

  // first we retrieve the Compliance Slope values
  for (i=0; i<NUM_AX12_SERVOS; i++)
  {
    CurrentMotion.JointFlex[i] = pgm_read_byte(motion_pointer[
    StartPage]+i);
  }
  // next we retrieve the play parameters (each are 1 byte)
  CurrentMotion.NextPage = pgm_read_byte(motion_pointer[StartPage]+
    NUM_AX12_SERVOS+0);
  CurrentMotion.ExitPage = pgm_read_byte(motion_pointer[StartPage]+
    NUM_AX12_SERVOS+1);
  CurrentMotion.RepeatTime = pgm_read_byte(motion_pointer[StartPage
    ]+NUM_AX12_SERVOS+2);
  CurrentMotion.SpeedRate10 = pgm_read_byte(motion_pointer[
    StartPage]+NUM_AX12_SERVOS+3);
  CurrentMotion.InertialForce = pgm_read_byte(motion_pointer[
    StartPage]+NUM_AX12_SERVOS+4);
  CurrentMotion.Steps = pgm_read_byte(motion_pointer[StartPage]+
    NUM_AX12_SERVOS+5);
```

```
20
  // now we are ready to unpack the Step Values
22  // 3 values are packed into one 32bit integer - so use
      pgm_read_word twice
    num_packed_steps = NUM_AX12_SERVOS / 3;
24  for (s=0; s<CurrentMotion.Steps; s++)
    {
26    for (i=0; i<num_packed_steps; i++)
      {
28      // higher 16bit
        packed_step_values = pgm_read_word(motion_pointer[StartPage]+
      NUM_AX12_SERVOS+6+(s*4*num_packed_steps)+4*i+2);
30      packed_step_values = packed_step_values << 16;
        // lower 16bit
32      packed_step_values += pgm_read_word(motion_pointer[StartPage
      ]+NUM_AX12_SERVOS+6+(s*4*num_packed_steps)+4*i);
        // unpack and store
34      CurrentMotion.StepValues[s][3*i+2] = packed_step_values & 0
      x3FF;
        packed_step_values = packed_step_values >> 11;
36      CurrentMotion.StepValues[s][3*i+1] = packed_step_values & 0
      x3FF;
        packed_step_values = packed_step_values >> 11;
38      CurrentMotion.StepValues[s][3*i] = packed_step_values & 0x3FF
      ;
      }
40    // Sanity Check - if the values are outside the overall Min/Max
       values we are probably accessing random memory
      for (i=0; i<NUM_AX12_SERVOS; i++)
42    {
        if ( CurrentMotion.StepValues[s][i] > SERVO_MAX_VALUES[i] ||
      CurrentMotion.StepValues[s][i] < SERVO_MIN_VALUES[i] )
44      {
          // obviously have unpacked rubbish, stop right here
46        printf("\nUnpack Motion Page %i, Step %i - rubbish data.
      STOP.", StartPage, s+1);
          printf("\nServo ID%i, Step Value = %i, Min = %i, Max = %i \
      n", AX12_IDS[i], CurrentMotion.StepValues[s][i],
      SERVO_MIN_VALUES[i],SERVO_MAX_VALUES[i] );
48        exit(-1);
        }
50    }

52  }

54  // and finally the play and pause times (in ms)
    // both need to be recalculated using the motion speed rate
      factor
56  for (s=0; s<CurrentMotion.Steps; s++)
    {
58    CurrentMotion.PauseTime[s] = pgm_read_word(motion_pointer[
      StartPage]+(NUM_AX12_SERVOS+6+CurrentMotion.Steps*4*
      num_packed_steps)+(s*2));
      if(CurrentMotion.PauseTime[s] != 0 && CurrentMotion.PauseTime[s
      ] < 6500 ) {
```

```
60        CurrentMotion.PauseTime[s] = (10*CurrentMotion.PauseTime[s])
      / CurrentMotion.SpeedRate10;
          } else {
62        CurrentMotion.PauseTime[s] = 10* (CurrentMotion.PauseTime[s]/
      CurrentMotion.SpeedRate10);
      }
64    }
      for (s=0; s<CurrentMotion.Steps; s++)
66    {
        CurrentMotion.PlayTime[s] = pgm_read_word(motion_pointer[
        StartPage]+(NUM_AX12_SERVOS+6+CurrentMotion.Steps*4*
        num_packed_steps+CurrentMotion.Steps*2)+(s*2));
68      if(CurrentMotion.PlayTime[s] != 0 && CurrentMotion.PlayTime[s]
        < 6500 ) {
          CurrentMotion.PlayTime[s] = (10*CurrentMotion.PlayTime[s]) /
      CurrentMotion.SpeedRate10;
70        } else {
          CurrentMotion.PlayTime[s] = 10 * (CurrentMotion.PlayTime[s]/
      CurrentMotion.SpeedRate10);
72      }
      }
74
}
```

Codes/Bioloid/UnpackMotion.c

K.6 ExecuteMotion() function

```
1  /* This function executes a single robot motion page defined in
       motion.h. It waits for the motion to finish to return control,
       so it's no good for a command loop. StartPage is the number of
       the first motion page in the motion; the function returns (int)
        StartPage of next motion in sequence (0 - no further motions)
       */
   int executeMotion(int StartPage)
3  {
     uint8 complianceSlope;
5    int commStatus;
     uint16 goalPose[NUM_AX12_SERVOS];
7
     // temporary array of timings to keep track of step timing
9    unsigned long step_times[MAX_MOTION_STEPS];
     unsigned long pre_step_time, total_time;
11
     // set the currently executed motion page global variable
13   current_motion_page = StartPage;

15   // first step is to unpack the motion
     unpackMotion(StartPage);
17
```

```
  // now we can process the joint flexibility values
  for (uint8 i=0; i<NUM_AX12_SERVOS; i++) {
    // translation is bit shift operation (see AX-12 manual)
    complianceSlope = 1<<CurrentMotion.JointFlex[i];
    commStatus = dxl_write_byte(AX12_IDS[i],
    DXL_CCW_COMPLIANCE_SLOPE, complianceSlope);
    if(commStatus != COMM_RXSUCCESS) {
      // there has been an error, print and break
      printf("executeMotion Joint Flex %i - ", AX12_IDS[i]);
      return 0;
    }
    commStatus = dxl_write_byte(AX12_IDS[i],
    DXL_CW_COMPLIANCE_SLOPE, complianceSlope);
    if(commStatus != COMM_RXSUCCESS) {
      // there has been an error, print and break
      printf("executeMotion Joint Flex %i - ", AX12_IDS[i]);
      return 0;
    }
  }
  total_time = millis();

  // in case the motion repeats we need a loop
  for (int r=1; r<=CurrentMotion.RepeatTime; r++)
  {
    // loop over the steps and execute poses
    for (int s=0; s<CurrentMotion.Steps; s++)
    {
      // create the servo values array
      for (int j=0; j<NUM_AX12_SERVOS; j++)
      { goalPose[j] = CurrentMotion.StepValues[s][j]; }
      // take the time
      pre_step_time = millis();
      // execute each pose
      moveToGoalPose(CurrentMotion.PlayTime[s], goalPose,
    WAIT_FOR_POSE_FINISH);
      // store the time
      step_times[s] = millis() - pre_step_time;
    }
  }
  total_time = millis() - total_time;
  return (int) CurrentMotion.NextPage;
}
```

Codes/Bioloid/ExecuteMotion.c

K.7 Example of forward motion

```
//This partial script (located in the motion.c file) describes how
    the Forward motion works as a composite of smaller movements/
    poses.
```

```
void Forwards(){
  executeMotion(F_S_L);
  executeMotion(F_S_L+1);
  executeMotion(F_E_R);
  executeMotion(F_E_R+1);
  _delay_ms(500);
  StandUp();
  executeMotion(F_S_R);
  executeMotion(F_S_R+1);
  executeMotion(F_E_L);
  executeMotion(F_E_L+1);
  _delay_ms(500);
  StandUp();
}


//Since all moveemnts need to be stored in the Flash memory because
     of RAM limitations, this is the way to do it. the following
    script(located in the motion.h file) shows the structure for
    storing poses, in this case for only a small portion (one
    fourth) of the total Forward movement, Values such as teh Joint
     flexibility, the pause time, the joint positions and the
    pointers to subsequent Motion pages are essential.
const struct // F_S_L
{
   const uint8 JointFlexibility[18];
   const uint8 NextPage;
   const uint8 ExitPage;
   const uint8 RepeatTime;
   const uint8 SpeedRate10;
   const uint8 InertialForce;
   const uint8 Steps;
   const uint32 StepValues[7][6];
   const uint16 PauseTime[7];
   const uint16 PlayTime[7];
} MotionPage32 PROGMEM = {
{7,7,5,5,5,5,5,5,5,5,5,5,5,6,6,6,6},
33, 33, 1, 21, 5, 7,
{{987275543,3121508913,1502925313,2190127785,
1012427398,1582303754},
 {987275543,3121508913,1502925319,2215297702,
 1029194372,1594898960},
 {987275543,3121508913,1502925324,2236273315,
 1050153602,1607492117},
 {987275543,3121508913,1502925328,2253056672,
 1075305087,1620083225},
 {987275543,3121508913,1502925321,2282422951,
 1100501628,1590729245},
 {987275543,3121508913,1502925303,2328564408,
 1117346426,1519430176},
 {987275543,3121508913,1502925294,2349537983,
 1125765753,1490072098}},
{0,0,0,0,0,0,0}, {80,80,80,80,80,80,80}
};

```

```
const struct //
{
    const uint8 JointFlexibility[18];
    const uint8 NextPage;
    const uint8 ExitPage;
    const uint8 RepeatTime;
    const uint8 SpeedRate10;
    const uint8 InertialForce;
    const uint8 Steps;
    const uint32 StepValues[7][6];
    const uint16 PauseTime[7];
    const uint16 PlayTime[7];
} MotionPage33 PROGMEM = {
{7,7,5,5,5,5,5,5,5,5,5,5,5,6,6,6,6},
38, 42, 1, 21, 5, 7,
{{978882839,3121508913,1502925303,2328562389,
1121532545,1620093472},
 {1041828119,3121508913,1502925321,2282433230,
 1108863624,1766890013},
 {1104773399,3121508913,1502925328,2253069001,
 1087849100,1833992729},
 {1159325975,3121508913,1502925324,2236285646,
 1058503311,1821401621},
 {1205485847,3121508913,1502925319,2215316176,
 1041744532,1792031248},
 {1239056663,3121508913,1502925313,2190158543,
 1033374362,1754270218},
 {1264234775,3121508913,1502925307,2165007052,
 1041775263,1716509188}},
{0,0,0,0,0,0,0}, {80,80,80,80,80,80,80}
};
```

Codes/Bioloid/ForwardExample.c

Bibliography

[1] M. Banzi. Servo Library for Arduino board. Website. https://www.arduino.cc/en/reference/servo.

[2] G. Borenstein. *Making Things See*. O'REILLY MEDIA, INC, 1005 Gravenstein Highway North, Sebastopol, CA 95472, 2012.

[3] Boston Dynamics. Atlas - The Agile Anthropomorphic Robot. Website, 2013. http://www.bostondynamics.com/robot_Atlas.htmll.

[4] Commaxi. Laser System Value Direct 7050-60W. Website. http://www.commax.co.jp/products/7050-4.html.

[5] Digi. XBee Series 1 vs. the XBee Series 2. Website. http://knowledge.digi.com/articles/Knowledge_Base_Article/The-Major-Differences-in-the-XBee-Series-1-vs-the-XBee-Series-2.

[6] Double Robotics. Telepresence robots. Website, 2015. http://www.doublerobotics.com/.

[7] B. Fry. Implementation of the Serial communication (RS-232) for Processing. Website, 2004. https://processing.org/reference/libraries/serial/.

[8] J. Garcia. Open Sound Control (OSC) C Sharp classes interface for the Unity3d game engine. Website. https://github.com/jorgegarcia/UnityOSC.

[9] H. Hasunuma, M. Kobayashi, H. Moriyama, T. Itoko, Y. Yanagihara, T. Ueno, K. Ohya, and K. Yoko. A Tele-operated Humanoid Robot Drives

a Lift Truck. Proceedings of the 2002 Conference on Robotics and Automation, May 2002.

[10] Interaction Design Department Zurich. Simple OpenNI and NITE wrapper for Processing. Website, 2015. https://code.google.com/p/simple-openni/.

[11] H. Ishiguro. Hiroshi Ishiguro Laboratories: Geminoid robots. Website, 2015. http://www.geminoid.jp/en/index.html.

[12] Leap Motion. Leap Motion Controller. Website. https://www.leapmotion.com/.

[13] P. Linus. Alternative firmware for the Robotis Bioloid humanoid. Website. https://code.google.com/p/bioloidccontrol/.

[14] R. Mogollán-Toral, O. R. Díaz-Márquez, and A. Aceves-López. Imitación de movimientos humanos en un robot humanoide Bioloid mediante Kinect. Proceedings of the 2013 National Congress of Automatic Control, Mexico, 2013.

[15] S. Pfeiffer. Guiado gestural de un robot humanoid mediante un sensor Kkinect. Website, 2011. http://upcommons.upc.edu/bitstream/handle/2099.1/12454/74118.pdf.

[16] Robotis. Robotis e-manual. http://support.robotis.com/en/.

[17] A. Schlegel. Implementation of the OSC protocol for Processing. Website, 2012. http://www.sojamo.de/libraries/oscp5/.

[18] U. Shrawankar and V. Thakare. Noise Estimation and Noise Removal Techniques for Speech Recognition in Adverse Environment. IFIP Advances in Information and Communication Technology, 2014.

[19] M. W. Spong, S. Hutchinson, and M. Vidyasagar. *Robot Modeling and Control (First Edition)*. JOHN WILEY AND SONS, INC, 2005.

[20] C. Stanton. Teleoperation of a humanoid robot using full-body motion capture, example movements, and machine learning. In *Proceedings of the 2012 Australasian Conference on Robotics and Automation*, 2012.

[21] T. Streeter. Voce: Open Source Speech Interaction. Website. http://voce.sourceforge.net/.

[22] The Nonverbal Group. How much of communication is really nonverbalL? Website, 2015. http://www.nonverbalgroup.com/2011/08/how-much-of-communication-is-really-nonverbal/l.

[23] Trek. Ai-Ball: the worlds smallest portable Wi-Fi remote camera. Website. http://www.thumbdrive.com/aiball/.