ROYAL INSTITUTE
OF TECHNOLOGY

Mechatronics Advanced course MF2059

Alan Project

Sandra Aidanpää
Erik Bergdahl
Freddi Haataja
Jacob Johansson
Victor Karlsson
Daniel Ohlsson
Joel Schröder
Elif Toy
Anders Åström

January 18, 2016

Abstract

This report covers the design and implementation of two humanoid robots made in collaboration with the artist Tove Kjellmark. The robots are to be part of the exhibition "The Imitation Game" at the Manchester Art Gallery during the spring of 2016. The robots were designed and manufactured using the requirements provided by the artist. These requirements described two robots able to move their arms in a human-like manner, having a conversation with each other and interacting with bystanders by addressing them and looking them in the eyes. The resulting robots have arms with three degrees of freedom moving randomly while they speak. The head consists of a solution developed by Furhat Robotics: a back-projected face connected to a head with two degrees of freedom. If a bystander gets into a predefined range, the robots will turn and face them. This is achieved using a Microsoft Kinect alongside irisTK, the open source software that runs the head.

Glossary

Table 1: Glossary

Abbreviation	Explanation
3D-CAD	Computer-aided design software for 3D-modelling
Arduino Mega 2560	Board with Atmel AVR microcontroller chip
Cura	3D printer software
DC	Direct Current
DOF	Degrees of freedom
Ethernet	Communication protocol for TCP/IP
Furhat	Social robot platform with voice and gestures
GUI	Graphical User Interface
I/O	Input/Output interface
IrisTK	Dialog system framework
Java	Programming language
JSON	JavaScript Object Notation
Kinect	Motion controller from Microsoft for Xbox One and Windows
KTH	Kungliga Tekniska Högskolan, Royal Institute of Technology
MATLAB	Programming language for technical computing
openCV	Open source Computer Vision software
PI	Proportional Integral controller
PID	Proportional Integral Derivative controller
Python	Programming language
PWM	Pulse Width Modulation
Simulink	Graphical block diagramming tool
SVN	Apache Subversion, file documentation
Ultimaker	Manufacturer of 3D printers
XML	Extensible Markup Language
XSD	XML Schema Definition

Contents

List of Figures

List of Tables

1 Introduction

This section presents a brief introduction and background for the project.

1.1 Background

In the spring of 2016, an art exhibition is going to be held at the Manchester (European City of Science 2016) Art Gallery in the UK in memory of computer scientist Alan Turing. The exhibition will be named "The Imitation Game" after a phrase used in one of Turing's most famous and influential works. The exhibition will feature contributions made by ten artists. These ten installations all express in different ways what happens in the meeting between humanity and technology. The Swedish artist Tove Kjellmark will contribute with a piece consisting of two humanoid robots, containing both human and robotic features and attributes. These robots will be displayed while having a conversation with each other and shall be able to interact with their spectators. In order to realize this setting, a mechatronic design and solution have to be created.

1.2 Project description

The main purposes and goals of the project are identified as:

- Realizing the artist's vision and express her ideas in a successful way
- Possessing human/humanoid robot qualities and behavior in both aesthetics and movement
- Interacting with the visitors, i.e. both sensing and addressing them in a satisfactory manner
- Being a well-designed, well-produced and well-documented complete product that will function in a real environment with all its challenges for a long period of time

The installation should consist of two humanoid robots seated in chairs, having a conversation that can be interrupted and resumed. The robots will show human-like behavior, gesticulating with their arms and hands in a human way as they talk, with randomly generated movements. If a spectator approaches the robots and comes in a certain range, the robots will sense this and turn their heads and eyes to make eye contact with the spectator and one of the robots should then ask the spectator not to disturb their conversation. The robots will have a conversation about existential questions that will be pre-written by the artist. As the robots will be active and functioning continuously in a public area for many hours a day and several months, they must be built in a robust way. Since they will be transported, they should also be easy to assemble and disassemble and should also be easy to repair. The robots will eventually have integrated aesthetic features made by the artist.

1.3 Delimitation

The moving pattern of a human arm is complex and involves many degrees of freedom. To achieve a reliable and robust control system for these movements in the relatively short time-span during which the robots will be designed and manufactured, these movements had to be limited. Discussions with the stakeholder concluded that the robots should be limited to three degrees of freedom in the arms, one for the shoulder, one for the elbow and for the hand.

The head was originally planned to be animatronic, as in utilizing mechanics to realize android features. This concept would have contained eyes with two degrees of freedom as well as a head with an additional two. The concept was proven to work during the pre-study conducted during the spring semester. However, it was also established that it would be too time-consuming to get this kind of mechanical solution to work in a robust and satisfactory manor during the scope of this project. To overcome this obstacle and still provide complex human-like faces for the robots, a solution developed by Furhat Robotics was incorporated into the project. This solution consists of a back-projected face able to produce the full spectrum of human facial movements along with two servos for the pan and tilt of the head.

One of the requirements from the stakeholder was that the robots should be able to interact with people around them by looking into their eyes. This was solved during the spring using openCV, an open source computer vision software, and face detection. Further development of this software was not needed due to the fact that this feature was included in the Furhat software when using it in conjunction with a Microsoft Kinect.

1.4 Method

Project Method

During the execution of the project, the team has used the agile development tool Scrum. In the beginning of each working day the team had short Scrum meetings to clarify current goals, analyze project progress and eliminate potential obstacles. The scrum sprint periods were set to two weeks. In the beginning of each new sprint the project progress was analyzed and new tasks for the sprint where determined.

The limitation of the interference and input of the stakeholder in the autumn semester resulted in less interruptions of the work-flow and had a positive effect on the project progress.

Development Method

Primarily, a model-based development method was implemented during the design process of the arms of the robots. The resulting models were then used as a basis when evaluating and ordering appropriate motors. The models were also used for the design of the motion control system to satisfy the requirements stated in 1.5, 1 (b). A central part of using this method has also been to test and verify the software and communication before actuating the arms to negate potential hazards.

Various software tools has been used, e.g. MATLAB, Simulink, IrisTK, Cura, Apache Subversion and Solid Edge.

Validation and Verification

The head and the arm are separate systems and were tested and validated individually. After the integration of the head and arm movement, the robots were tested for robustness by running the system for longer time durations (the longest period was 4 hours). The results validated the robustness of the system. The results of the project are verified by the stakeholder and the teaching team.

1.5 Requirements

The requirements for the robots have been derived and elaborated during meetings with the stakeholder of the project, Tove Kjellmark.

Requirements:

1. The robots shall be able to:
   (a) have a conversation with each other, easily distinguishable by spectators within a radius of five meters.
   (b) move its arms, with at least three degrees of freedom.
   (c) move its head with at least two degrees of freedom.
   (d) interact with bystanders when the bystanders get closer than two meters by looking them in the eyes and addressing them with predefined phrases.
   (e) move in a stable and human-like way without noticeable vibrations.
2. The robots shall be durable and robust.
3. The robots shall be easy to assemble and dissemble with clear guidelines provided on how to do so.
4. Documentation shall be provided to every specific feature in respect to installation, start-up and to troubleshooting.
5. Extra parts shall be provided for critical parts that has a risk of breaking or risk of being worn out.

2 Theory

This section presents the theory upon which the various design decisions were made during the project.

2.1 Dynamics

This section will cover the modeling and design of the control system used to operate the motors moving the arms located in the shoulder, elbow and hand.

2.1.1 Arm modelling

Achieving a natural arm movement similar to a humans can be quite complex. Process models of the whole arm will enable and simplify the possibility of simulating and testing such a dynamic system as well as laying the foundation for the motor control design presented in section 2.1.2. The two designed models are dynamic models where the first is a model of the entire arm with an angular position output of the shoulder joint and a voltage input to the shoulder motor. The second model is a model of the lower arm with an angular position output of the elbow joint and a voltage input to the elbow motor.

Before creating a dynamic process model of the arm consisting of the arm itself, a hand and two motors, the mass and the center of mass have to be derived in order to calculate the minimum required torque needed for the shoulder and elbow motor respectively. Using the designed CAD model of the arm, these can be extracted directly. Consequently, the minimum required torque can be calculated as

$$T_s = m_s g r_s \tag{1}$$

$$T_e = m_e g r_e \tag{2}$$

where T_s and T_e represent the minimum required torque needed to rotate the arms for the shoulder and elbow motors respectively, m_s and m_e are the estimated masses at the center of mass at distance r_s and r_e from the shoulder and elbow joints respectively and where g is the gravitational acceleration. The parameters are calculated based on a worst case scenario as can be seen in Figure 1 where the arms are perpendicular to the gravitational vector and are also used as a basis for ordering appropriate motors.

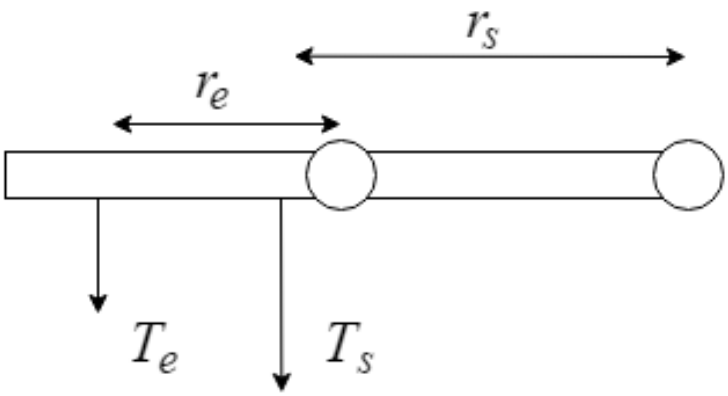


Figure 1: Arm fully extended perpendicular to the gravitational vector

When setting up the model to achieve an appropriate response and behavior, a set of motor and system parameters have to be calculated using the provided motor and belt drive parameters from the manufacturers as well as the arms' dynamic parameters presented in equations (1) and (2). The parameters provided by the manufacturer are depicted in Table 2.

Table 2: Parameters provided by manufacturer.

Abbreviation	Description
b	Belt drive gear ratio
n	Motor gear ratio
I_{nl}	No load current
I_{stall}	Stall current
T_{stall}	Stall torque
ω_{nl}	No load speed
J_m	Motor inertia

Using the stated parameters, the remaining system parameters of interest depicted in Table 3 can be derived.

Table 3: Sought variables to be calculated

Abbreviation	Description
K_t	Motor torque constant
d	Linear friction (damping)
K_{emf}	Motor velocity constant
R	Internal electrical motor resistance
J_l	Load inertia
J_x	Inertia on motor input shaft

The stated parameters in Table 3 are calculated as follows:

$$K_t = \frac{T_{stall}}{I_{stall} n} \tag{3}$$

$$d = \frac{K_t I_{nl}}{\omega_{nl}} \tag{4}$$

$$K_{emf} = \frac{R(I_{stall} - I_{nl})}{\omega_{nl}} \tag{5}$$

$$R = \frac{U_{motor}}{I_{stall}} \tag{6}$$

$$J_l = mr^2 \tag{7}$$

$$J_x = J_m + \frac{J_l}{n^2} \tag{8}$$

where U_{motor} is the applied voltage to the motor and m and r are the generic terms for the masses and distance units stated in equation 1 and 2. When modelling the system, all motor shafts are considered stiff but with a damping d thus eliminating the spring constant term. When modelling the lower arm which has a belt drive adding an additional gear in the elbow joint with gear ratio b. The belt drive increases the outgoing torque from the elbow motor and decreases the outgoing velocity with a factor b.

With the obtained parameters, an initial Simulink model depicted in Figure 2 is given with a voltage input and a velocity output. In order to verify and obtain a more accurate set of parameter values for the model, a set of empirical tests are performed in Simulink.

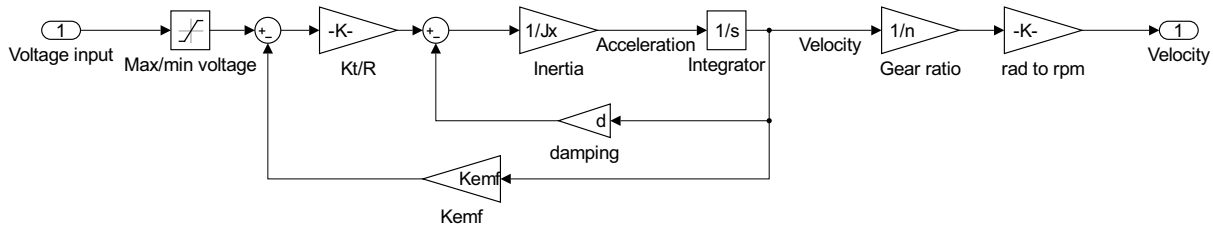


Figure 2: Initial Simulink model of the arm

By applying different voltage step inputs and studying the corresponding velocity response of the motor process model and the actual motor, the parameters of the model can be calibrated until achieving the same response as the real motor. This way, the model can be adjusted and tuned to more accurately correspond to reality. Using the external mode tool in Simulink, the corresponding step responses can be plotted and compared in real-time.

By studying the motor equation

$$\frac{d\omega}{dt} = \frac{1}{J}(\frac{K_t}{R}(U - K_{emf}\omega) - d\omega - sgn(\omega)T_c) \tag{9}$$

where U is the input voltage, it is sensible that obtaining a proper acceleration can be achieved solely by calibrating the inertia J while the linear friction d and static friction T_c correspond to the steady state velocity of the model. These dynamic frictions are calculated from the equations (derived from equation 9)

$$0 = \frac{K_t}{R}(U_1 - K_{emf}\omega_1) - d\omega_1 - sgn(\omega_1)T_c \tag{10}$$

$$0 = \frac{K_t}{R}(U_2 - K_{emf}\omega_2) - d\omega_2 - sgn(\omega_2)T_c \tag{11}$$

by studying the steady state velocity response of the real motor (ω_1 and ω_2) resulting from the two different input voltages ($U_1 = 12V$ and $U_2 = 24V$) and setting the inertia J to 0 considering steady state. The resulting Simulink model depicted in Figure 3 includes the static friction T_c.

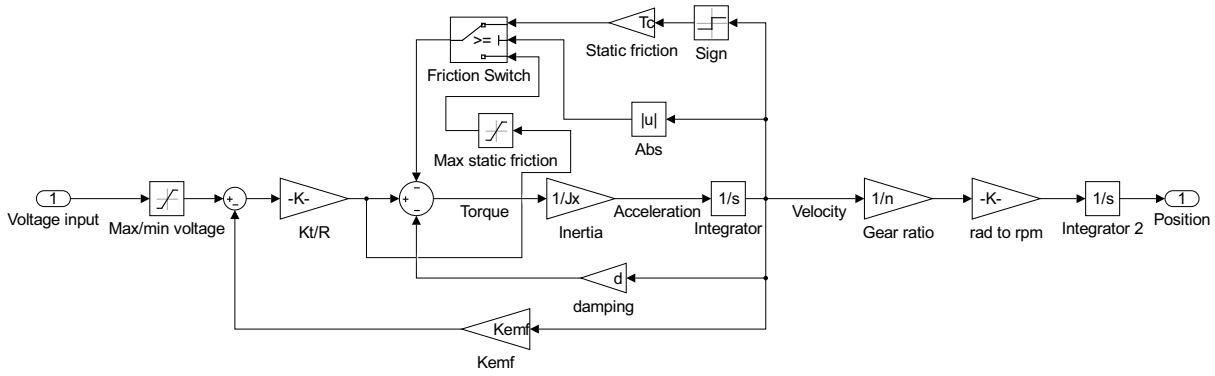


Figure 3: Resulting Simulink model with a position output incorporating the static friction

2.1.2 Motor control

A precise controller is necessary in order to fulfil requirement 1.(e) regarding precise arm movement. The positional movement pattern of the upper and lower arms are determined by a trajectory planner (presented in section 2.1.3) generating a position reference and the corresponding velocity and acceleration signals.

In order to follow the position reference properly, a position controller is designed. By generating a feed-forward control signal, the corresponding voltage inputs to the motors can be predicted as soon as a new position reference is generated. The feed-forward part of the controller depicted in Figure 4 consists of a model following system which is the inverted process model of the arm. The model following system generates the feed forward signal based on the calculated acceleration and velocity signals corresponding to a specific position reference signal from the trajectory planner. The result of the feed-forward control is a reduced delay of the whole system and an increased control accuracy by reducing the over and undershoot.

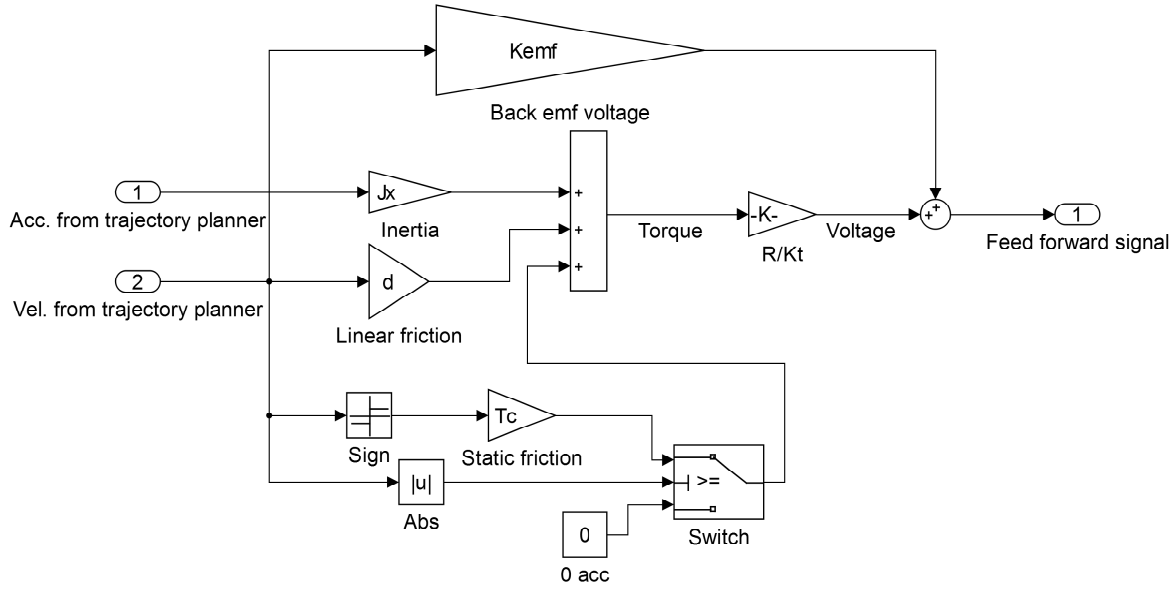


Figure 4: Model following part generating the feed-forward control signal

Achieving a perfect model of the arms as well as accounting for differences in load and disturbances due to external factors is impossible; a positional error-feedback system is required in combination with the feed-forward system to correct any angular position errors. The designed error feedback system consists of a PI-controller to eliminate the steady state error and overshoot. Using the built-in PID-tuner in Simulink and some manual tuning, the P- and I-parameters are tuned to meet the requirement 1.(e). A plot of the feed-forward and feedback control signals resulting from a trajectory position reference can be seen in Figure 5. It is noticeable that the feed-forward control signal is significantly larger and it also displays how the feedback signal only corrects for small errors in the control.

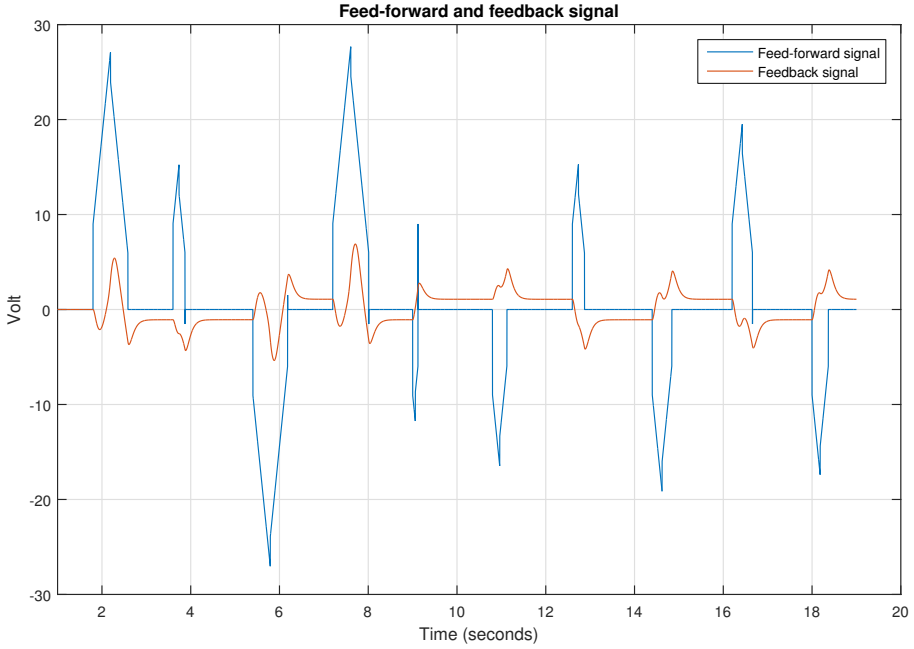


Figure 5: Model following part generating the feed-forward control signal

In contrary to shoulder and elbow control, the control of the 180-degree hand rotation actuated by a servo motor is only supplied by a position reference signal from the trajectory planner using no external control system other than the internal position control of the servos.

2.1.3 Trajectory planner

The robot is supposed to change the position of its arms continuously as long as it is speaking, thus the position reference is updated as soon as the arm has reached its designated position. Giving the motors a reference step input will introduce unnecessary tear on the parts that holds the arm together due to rapid acceleration/deceleration. Moreover there is a need to not only control the arm's positioning but also the way it reaches that position. The trajectory planner enables control of the target position as well as control of the acceleration and velocity of each joint. The trajectory planner is part of the feed-forward controller that together with a feedback controller constitutes the main control.

The maximum acceleration of the system is given by the equation below:

$$a_{max} = \frac{T_{max}}{J_l} \tag{12}$$

Where T_{max} is the maximum outgoing torque of the motor and J_l is the moment of inertia of the rotating system.
Moreover, the maximum velocity is given by the equation below:

$$v_{max} = \frac{T_{sat}}{d + T_c} \tag{13}$$

where M_{sat} is the saturated torque, d the linear friction and T_c is the static friction.
When maximum velocity is reached with maximum acceleration the corresponding time is given by the equation below:

$$t = \frac{v_{max}}{a_{max}} \tag{14}$$

where v_{max} is maximum velocity and a_{max} is maximum acceleration.
The position of the system with maximum acceleration is derived from the equation below:

$$\theta(t) = a_{max} \int_0^t t \mathrm{d}t \tag{15}$$

Where $\theta(t)$ is the position, a_{max} is the maximum acceleration and t the time.

An example of the appearance of the three outgoing signals resulting from a one radian position change is displayed in Figure 6.

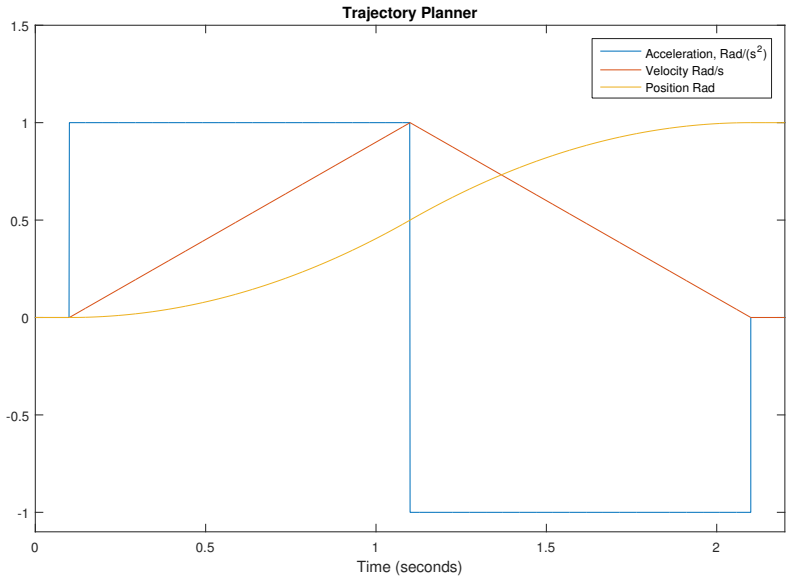


Figure 6: Trajectory planner for angular position of a motor

By introducing a trajectory planner to the motor control model a smooth reference signal is achieved. Figure 7 displays a 25 second moving sequence output from the trajectory.

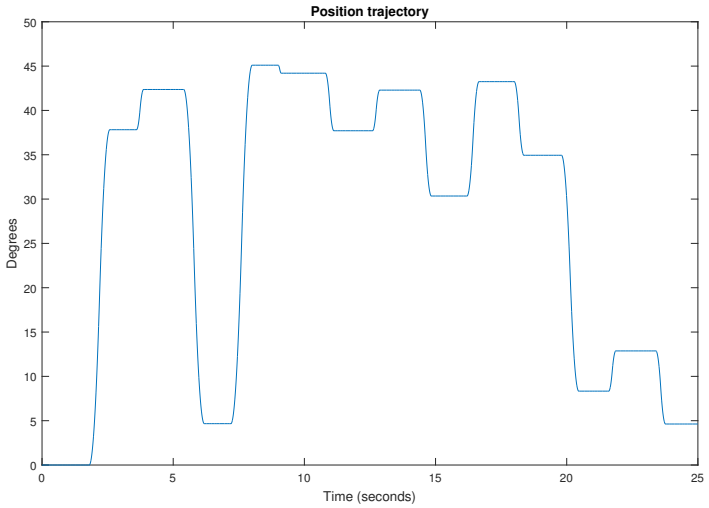


Figure 7: Trajectory positon reference

2.2 Furhat

Furhat is a robotic head developed by Furhat Robotics which aims to implement spoken dialog systems and to improve human-machine interaction. They have shared their design of the physical robotic head as well as the Furhat specific plugins for the open source software IrisTK, which is a Java based platform developed by Gabriel Skantze. Following the design provided by Furhat Robotics, two complete heads were manufactured and assembled.

The irisTK features that were used are:

- Detection of the visitors in the view of the Microsoft Kinect.
- Establishment of eye contact by turning the neck and eyes in the direction of the bystander in case of interruption.
- Text to speech synthesis.
  - When this is performed, the lips in the projection moves accordingly.

The software was altered to suit the needs of the team by adding the following features:

- Interaction between two Furhats.
- Interruption of the speech when a visitor enters the predefined proximity area of the robots.
  - When this occurs, the robot addresses the bystander with a predefined phrase after which it turns back to address the other robot again, starting from the beginning of the last sentence.

A distributed system was built. The arm movement of the robots as well as the head of one robot were controlled by one computer; the master computer, and the head of the second robot was controlled by a second computer; the slave computer. A broker socket was used to connect two computers via Ethernet cables and a router. The system can be seen in Figure 8.

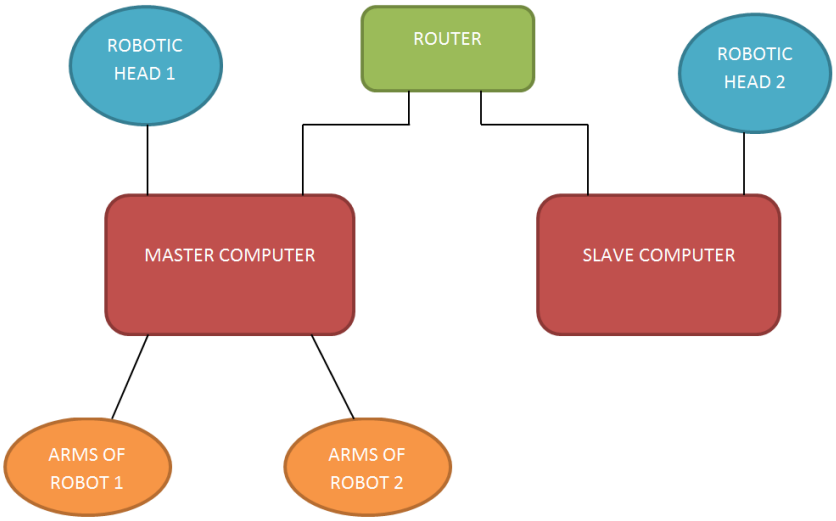


Figure 8: Distributed System of the two robots

The Java application is implemented on two computers. The initialization of the first robotic head (agent1) is made in the Java application running on the master computer and the initialization of the second robotic head (agent2) is made in the Java application running on the second computer. The behavior of both robotic heads are controlled from the Java application on the master computer and required information regarding the behavior of the second agent is sent to the slave computer via a broker service established between the two computers. The Java code on both computers is explained in detail in the software section under the Implementation chapter. For the complete Java code see Appendix B.

3 Implementation

This section covers the implementations of the different parts of the robot.

3.1 Design and manufacturing

Design decisions for the manufactured and implemented parts are described here. They are divided into three mechanical areas; Frame, Arms and Head.

3.1.1 Frame

This section is divided into three parts; Shoulder plate, Metal-frame and Speakers.

Shoulder plate

The shoulder plate was an important part of the robot since the integration of the arms, the head and the frame were dependent on it. The shoulder plate was made out of wood and is smaller than the width of human shoulders. This is because the motors are rather big, and together with the plate, as wide as human shoulders, see Figure 9.

The plate was designed and manufactured using CAD and a CNC milling machine. This method was used because the plate contains a lot of details since the head and the shoulder motors were supposed to be mounted on it.

In the middle of the plate, a hole was made so that the Furhat head could be mounted. After various configurations with the head mounted on top of the plate and sunken into the plate, the best solution for a generic human look proved to be sinking the head into the plate. This was due to the length of the neck, which became too long with the Furhat head mounted on top of the plate, see Section 3.1.3.

For the shoulder motor mounting, tracks were milled giving the possibility to add one degree of freedom for the arms. These tracks enabled a rotation of the shoulder motors in the horizontal plane, allowing the arms to be directed inwards or outwards, providing the option to individually adjust each arm and lock them at the desired angle.

Metal frame

Sturdy frames were constructed to act as backbones for the robots. This made their upper bodies robust and stable enough to stand on their own. The frame consisted of four metal tubes attached to a wooden plate at the bottom to provide stability. On top of the tubes, the shoulder plate was attached to enable integration of head and arms, see Figure 9. The plates were connected to the pipes using M8 bolts.

Figure 9: Frame of robot

Since the robots were supposed to be human-sized, the metal tubes have the same length as an average human spine. The width between the tubes was set so that the integration of the speakers could be done with ease. Both of the robots were installed with two speakers each, which were placed on top of each other inside the metal frame. A ribcage and a spine were attached to the frame to enhance the aesthetics of the robots. At the exhibition, the robots are supposed to sit in chairs facing each other. To make the assembly between the robot and the chair as easy as possible, the bottom plate and the robot were put on the chair and a few cable ties were utilized to strap the robots to the backrest of the chair, fixating the robot to the chair.

Speakers

For the audience to hear the robots speaking, a speaker system was needed. This speaker system should be able to produce a sound that is clear and loud enough so the audience can hear the robots without difficulties. It was also necessary to distinguish from which robot the sound originated from. The most desired solution would be speakers mounted in the head so that the direction of sound follows the head movement, and that can mimic the human voice in regard to clarity, frequency and sound level. For the sound level to be as loud as or louder than the human voice, the speaker would need to be bigger than what could fit within the Furhat-head. Therefore the decision was made to mount the speakers within the chest of the robots. An investigation was made to find a solution for making voice-mimicking speakers with pre-recorded voices, but this was proved to be too time-consuming.

As in the case of this project the sound was produced by a text-to-speech software and not recorded voices, therefore the quality of the sound and the need for special voice-mimicking speakers was small. When trying out different solutions, common speakers for computers were deemed well enough for our purpose. They are compact, cheap, can play loud enough and have all the necessary components included in the package. The other solution, manufacturing speakers, amplifier, power supply and connections, would increase the complexity of the system and take much more time.

The text-to-speech synthesis for each robot was produced on two different computers, each with its own power supply. There was only need for two sound channels, one from each computer that could be connected to one set of speakers where one computer would play the sound of one robot on one channel (left/right). The problem then was that two sound signals produced on two different computers needed to share ground. Without using a ground loop isolator a humming noise was present. To solve this problem two sound systems were bought, one for each robot/computer, thus keeping the ground loops separated.

3.1.2 Arms

The arms were designed with regards to robustness, weight and function. Too heavy arms would require strong, big and expensive motors. Since the robots will be displayed in a museum they should be able to withstand disturbances from visitors that might interfere and apply force to the arms during the exhibition. To reduce the torque needed from the motors the arms should be lightweight while still being rigid and esthetically designed. During the design, emphasis have been placed on achieving low backlash from motors and low friction in joints. The arms must be designed to hold together during at least three months, which is the duration of the exhibition. The upper arms are 250 mm long, manufactured from 40x40x2 mm aluminum tubes and consists of milled features to reduce weight as well as fasteners for the elbow joint, the shoulder joint, the motor mounting and the motor fasteners. The lower arm has the same dimensions as the upper arm. Because of the lesser amount of parts connected to the lower arm and to lower the moment applied to the arm more material has been milled to further reduce the weight of the arm.

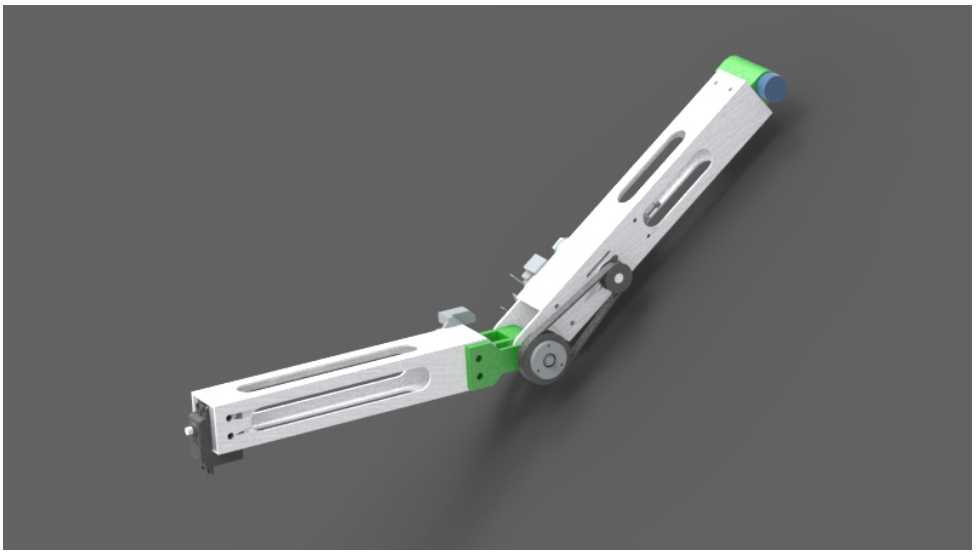


Figure 10: Arm assembled

Figure 10 shows the assembly of the robot arms. The lower and upper arms are manufactured in aluminum with milled features. The milled features are implemented to reduce the weight while still allowing the arm to remain rigid. Fastener holes are also drilled in the arms for fastening of motors, joints and motor tighteners. Because of the complex manufacturing design of the arm joints and motor attachment parts, 3D-printing was chosen. 3D printing produces low-weight parts as well as a quick way of altering and updating the design at a low cost.

Since the shoulder motor did not have any encoder a potentiometer was used to give the absolute position of the upper arm. To mount the potentiometer to the outgoing shaft of the motor it was placed on the opposite side of the output shaft to be able to follow the rotation of the shoulder motor.

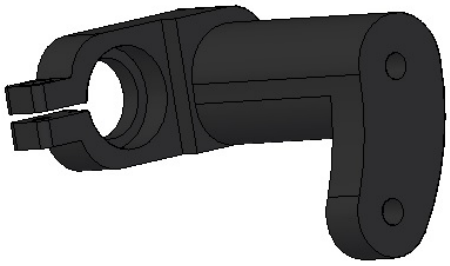


Figure 11: Potentiometer mounting

A 3D-printed part was designed to hold the potentiometer and fasten its position by a clamping designed within the part, as seen in Figure 11. By tightening the clamp around the potentiometer body with a nut and screw the potentiometer is firmly attached, while the potentiometer shaft is

attached to the shoulder joint. With the additional, vacant threaded holes on the motor it was possible to design the potentiometer holder to be mounted directly on the motor and shoulder plate.

To attach the upper arm to the shoulder motor a joint was designed. The shoulder joint is mounted on the outgoing shaft of the shoulder motor. The outgoing shaft has both a threaded hole in the center and a wedge. The design idea is implementing these features for the shoulder joint to give a robust joint, and because of the complexity of the design, the part was 3D-printed. A groove is placed in the part to give room for the wedge; this will ensure that the torque is transferred from the motor shaft to the joint and not entirely relying on the screw.

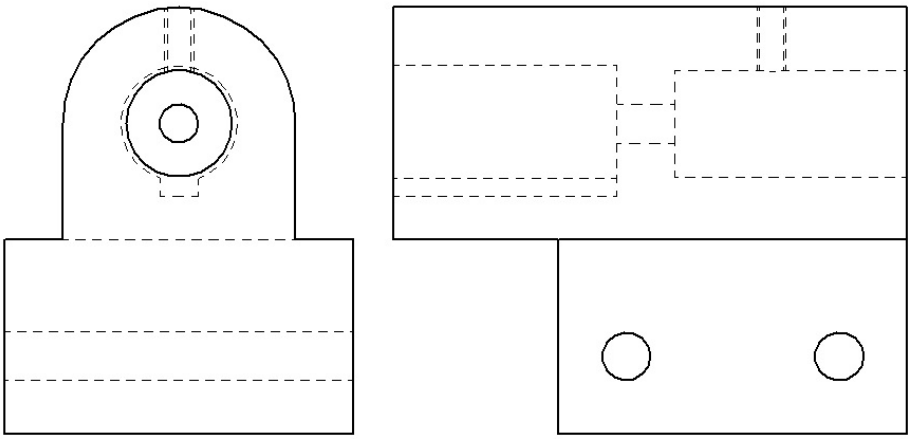


Figure 12: Shoulder joint drawing

A drawing of the shoulder joint is shown in Figure 12. The part is then mounted on the outgoing shaft with a wedge and a screw to apply force towards the outgoing shaft in order to reduce the possibility of drift on the joint. The hole where the screw connects to the outgoing shaft is also designed to accommodate the potentiometer. Since the screw head is larger than the potentiometer an adapter is applied to the potentiometer shaft that fills the void between the potentiometer shaft and shoulder joint. To ensure that the potentiometer rotates in accordance to the motor a stop screw is put in the shoulder joint that applies force on the potentiometer shaft. The upper arm then slides on the joint and is fastened with two screws and nuts.

The elbow joint is designed to give low friction and ensuring a robust joint between the upper and lower arm. Ball bearings with an inner diameter and outer diameter of 1/4" and 5/8" respectively was implemented. A 1/4" shaft was used to give a good, tight fit between the bearings and shaft to give low backlash of the joint.

Figure 13 shows the elbow joint assembly in an exploded view. The elbow connector that is connected from the shaft to the lower arms is 3D printed and designed to enclose the whole inside of the lower arm in order to achieve a rigid system. The elbow connector is mounted with

two ball bearings on the rod to give low friction and rigid support. It locks to the outer ring of the bearings while a washer locks to the inner ring. To further reduce the weight and keep the elbow connector rigid, a hole was placed in the middle of the connector. The extenders from the upper arm are water cut from 2 mm steel plate to fixate the shaft and not interfere with the elbow connector. Two washers are then used to lock these extenders to ensure that they do not get bend, and so that they stay in position. The bearing for the sprocket is locked on both sides of the inner ring with washers. Grooves where manufactured in the rod where the washers are connected.

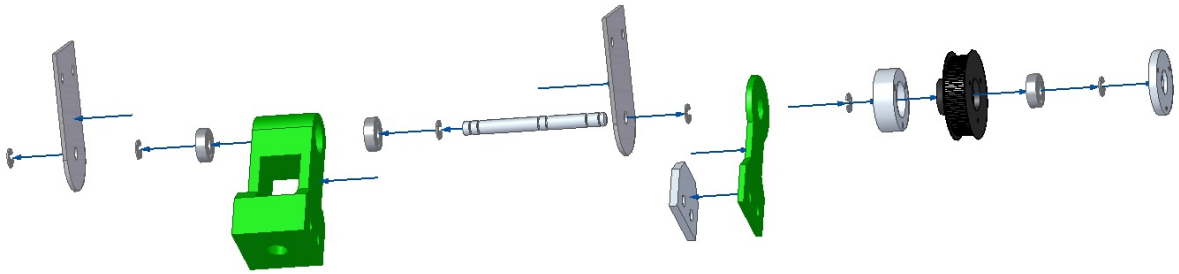


Figure 13: Elbow joint assembly

The motors originally used had 36 rpm, more then what was needed, while the torque they applied was low when concerning the safety factor. This also created a challenge of mounting the motor at the elbow joint. Therefore, a belt drive was implemented, since the motor would then be able to be mounted on the upper arm and this would also reduce the required moment from shoulder motor, as well as give a higher output torque on the elbow joint. The smaller sprocket that is mounted on the motor has 40 teeth and the larger sprocket connected to the elbow joint has 22 teeth which gives a ratio of 1.82. New motors were bought with higher output torque that was able to achieve 40-50 rpm at 3-4 Nm. The rpm is approximately the same as the old motors, so the belt drive thus needed no re-design to function with the new motors. The belt that is used is of rubber material since it will not be subjected to chemicals or rough environments during its operation.

The outgoing shaft from the motor has a flattened part, and the size of the rod and smaller sprocket invoked the design decision to fasten the sprocket with two stop screws: one applying force on the flat part and another one rotated 90 degrees to give good friction force for the sprocket upon the shaft. The sprockets are made of hard plastic and a more robust way of fastening them to the rod was designed.

An aluminum sprocket adapter was designed as shown in Figure 14. A larger hole was drilled in the small sprocket to give room for the adapter. The adapter was then epoxied together with the sprocket to ensure that the rotational force was not only transferred from the stop screw. Once epoxied two threaded holes were drilled through the sprocket and adapter. The circumference of the adapter was designed so that the stop screw would thread at least 4 revolutions in the aluminum adapter as to reduce the wear on the plastic sprocket.

Figure 14: Sprocket mounting

Due to the choice of using a belt-driven gearing for the elbow a tensioner is to be preferred. It is possible to do without one, but then it becomes difficult to tension the belt when tightening the two small screws that mount the motor to the arm. Without these tensioners all the load on the elbow motor would go through two small screws connecting the motor to the upper arm, and if these comes loose the belt will lose all its tension and thus lose all the torque transferred to the lower arm. Two of these tensioners are mounted on each arm. They consist of a part that slides on to the motor body, a part that is mounted to the arm and a threaded rod connecting the parts. A nut connects the threaded rod to the part holding the motor and a nut above the part mounted to the arm makes it possible to make the adjustments. Adding another nut above this locks the desired adjustment.

The motor tightener is shown in Figure 15. No struggles were encountered creating this part other than that the choice of motors were changed during the project and thus the dimensions needed to be altered.

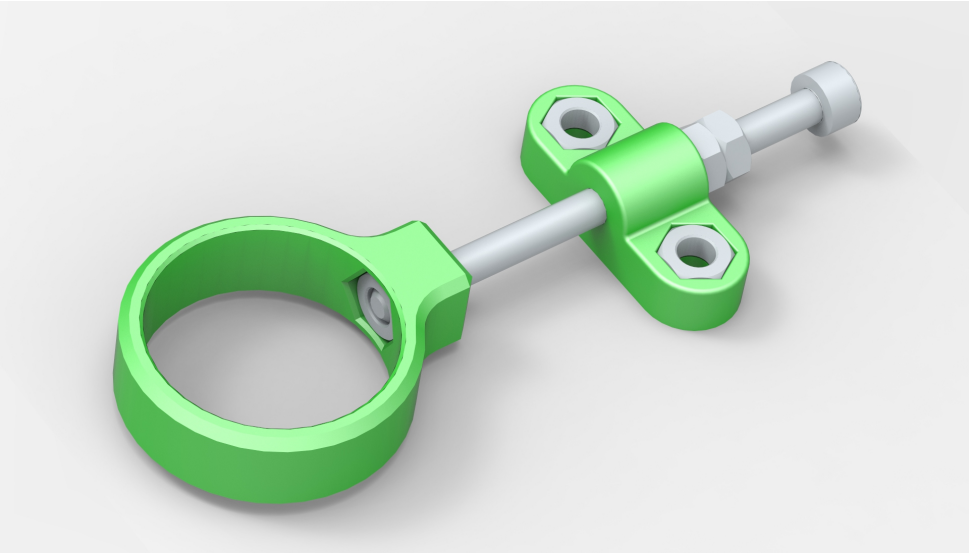


Figure 15: Motor adjustment assembly for belt tension

The larger sprocket is connected to the elbow joint rod with a ball bearing to give low friction. A lever is connected to the sprocket with a plastic adapter, a stop plate and screws which transfers the torque of the elbow motor to the lower arm. The sprocket was post-processed to fit on the ball bearing and to enable the screws to fasten to the lever. The plastic adapter is designed to fit inside the groove of the sprocket and ensure that the lever comes into the right position to be parallel to the lower arm.

Figure 16 shows how the torque lever is mounted on the larger sprocket. The lever was first manufactured with 3D-printing, but during testing it broke because of the high applied torque, which led to redesigning the lever. The lever is now manufactured from a piece of water-cut steel plate to ensure that the lever will not break. The distance mounted on the outside of the lever is used to ensure that the lever is parallel to the lower arm. The distance is designed so that the lever get some distance between the joint and lower arm to ensure there is no unwanted friction.

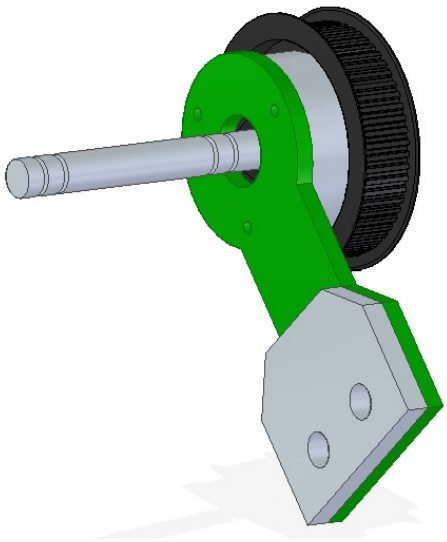


Figure 16: Torque lever

Figure 17 shows a stop plate that was 3D-printed and that is mounted on the outside of the sprocket. When the screws are fastened this stop plate and the large sprocket applies pressure between the sprocket and bearing that gives the sprocket a good fitting on the bearing.

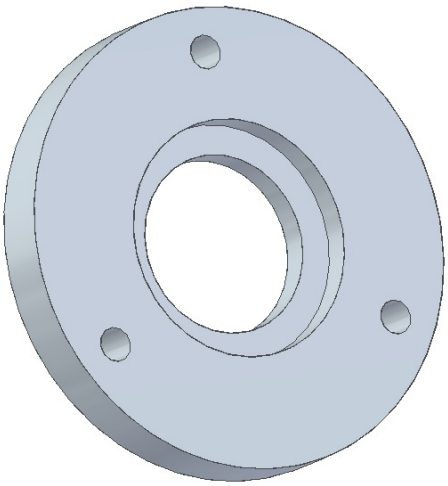


Figure 17: Stop plate between the bearing and sprocket

The servo mount for the hand servo is a 3D-printed part with nuts on the inside in a hexagon form so that the nuts are fixated when the screws are tightened. The servo is mounted onto the servo mount with the usage of its pre-existing design.

The combination of the servo mount and servo gives a tight fit and rigid design as shown in

Figure 18. The servo comes with a threaded outgoing shaft and pre-existing cross part to connect to the outgoing shaft with a screw. The 3D-printed hand is then connected with screws to the cross part which is firmly attached to the servo with four screws.

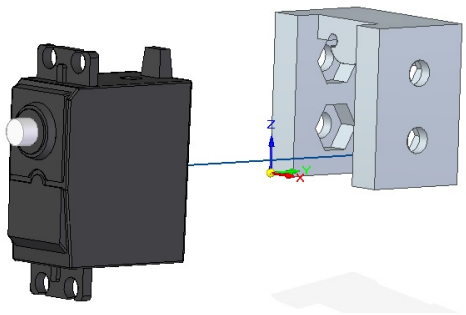


Figure 18: Hand servo mounting

The elbow should not rotate more than full extension and should not rotate so much that the lower arm comes in contact with the upper arm and produces a mechanical stop, which gives unnecessary tear on the elbow motors. The shoulder end point should be reached when the arms are in a downright position and not rotate to a too high arm position. Micro-switches are used to define these end positions. Mountings for the micro-switches are designed around the micro-switches to get a rigid fitting.

Figure 19 shows the mounting for the micro-switches for the elbow. It is 3D-printed and the design makes the micro-switches fit with their legs and gives a good fit without using extra fastening on the same part. The mounting is also designed to be fitted on the screws that fastens the elbow joint to the upper arm. Mirrored parts are also manufactured in order to mirror the switches according to the other arm.

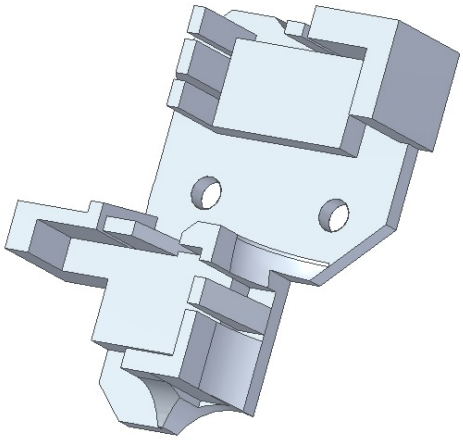


Figure 19: Mounting for elbow micro-switches

To trigger the micro-switches a 3D-printed part is manufactured as shown in Figure 20. The design gives the possibility to trigger both of the micro-switches with the same part. They were designed with the arm assembly to evaluate the length of the trigger arms to ensure that correct end points for the arm are achieved. The trigger is fastened with the screws used to fasten the elbow joint to the lower arm.

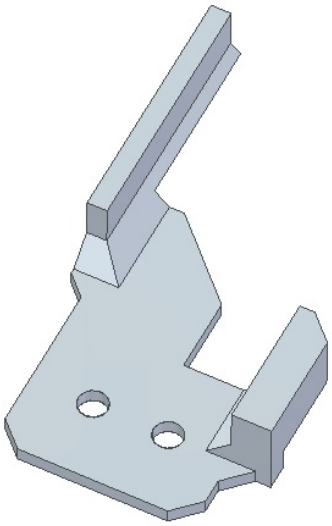


Figure 20: Elbow micro-switch triggers

The micro-switch holders for the shoulder is shown in Figure 21. The switches are mounted using the holes in the switches with the help of extensions. Both mountings are 3D-printed and attached to the extra mounting holes of the shoulder motor on the outside of the shoulder plates.

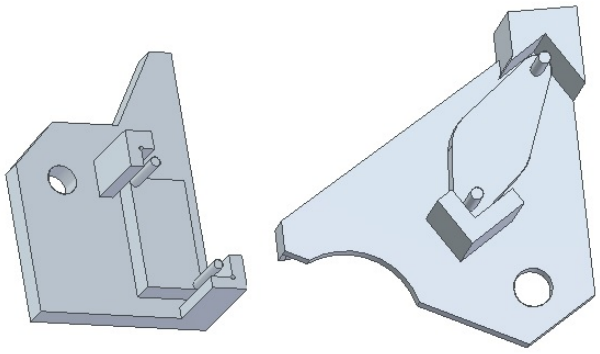


Figure 21: Micro-switches for the shoulder

Figure 22 shows the 3D-printed part that is used to trigger the shoulder switches. Two of them are mounted on both sides of the upper arm with the screws that attach the upper arm to the shoulder joint.

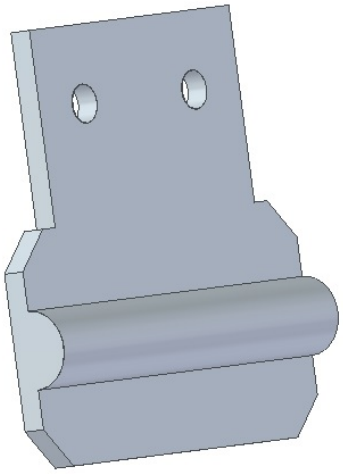


Figure 22: Shoulder micro-switch triggers

3.1.3 Head

This section is divided into two parts; production and assembly of the head, and projector.

Production and Assembly of the Robotic Head

Most of the parts used for the head are designed by the Furhat team and 3D printed by members of the Alan Project.

Both the face and head were vacuum formed from milled models provided by the Furhat team, see Figure 23 below.

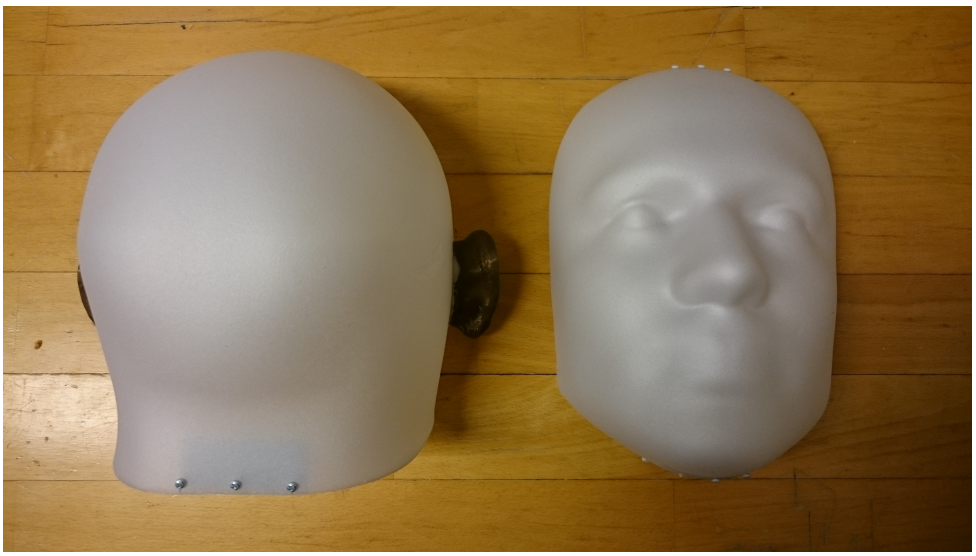


Figure 23: Front and Back Mask

Since a real face was supposed to be projected on the mask, the mask itself had the form of a human face. To enhance the picture strength of the projected face, the masks were sprayed with a white matter. This made the surface dim, which was essential when using back projection. The result of the back projected face can be seen in Figure 24 below.

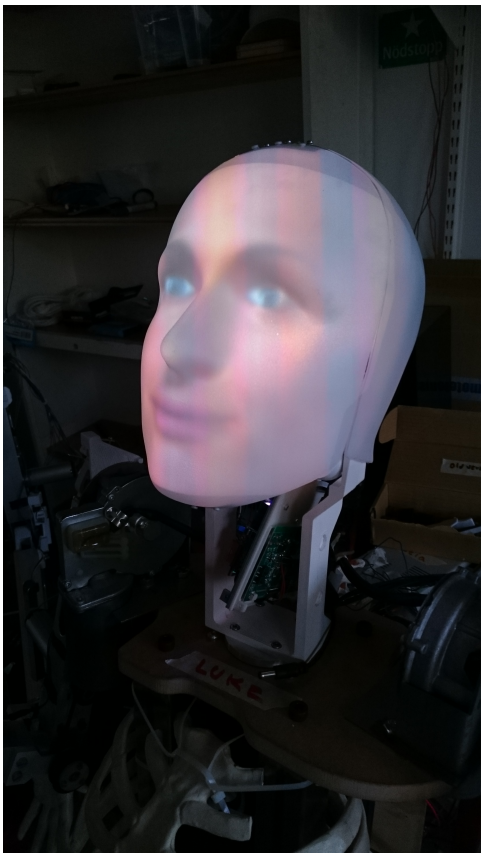


Figure 24: Back projected face

As mentioned earlier, a complete model of the head were bought from the Furhat team. The parts that had to be 3D-printed were printed during the project. To enable a smooth movement of the head, the moving parts were connected with bearings. The projector which will be discussed later, was attached at the bottom of the head. Since it was directed almost straight upward, a mirror mounted in the back of the head was used to angle the picture onto the face. An assembly of the head can be seen in Figure 25 below.

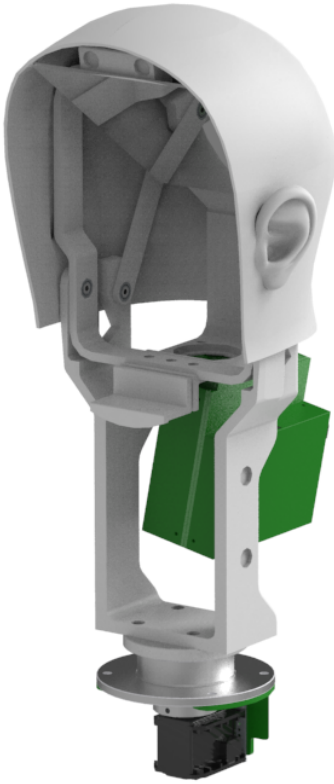


Figure 25: Head assembly

Since the robots should be able to move their heads and direct them towards the "disturbing" audience, servomotors were used enabling both a pan and tilt movement. For the pan movement a Dynamixel RX-28 servomotor was used. This motor allows the robot to look side-to-side and since the movement does not have to exceed 180, a servo is a good choice. For the tilt movement a Dynamixel MX-64 was used. These motors were selected by the Furhat team and were a part of the package bought from them. In their solution the face was mounted on a base where the electronics and speakers were mounted within a case. In this project the head needed to be mounted on the robots' shoulders so that it was anatomically correct. To do this another mount for the pan-servo was needed. The new mount made it possible to have the servo within the chest and underneath the shoulder plate. The servo was fastened to this mount and the mount was attached to the shoulder plate with the same screws as the ones holding the bottom part of the neck bearing in place, see Figure 26.

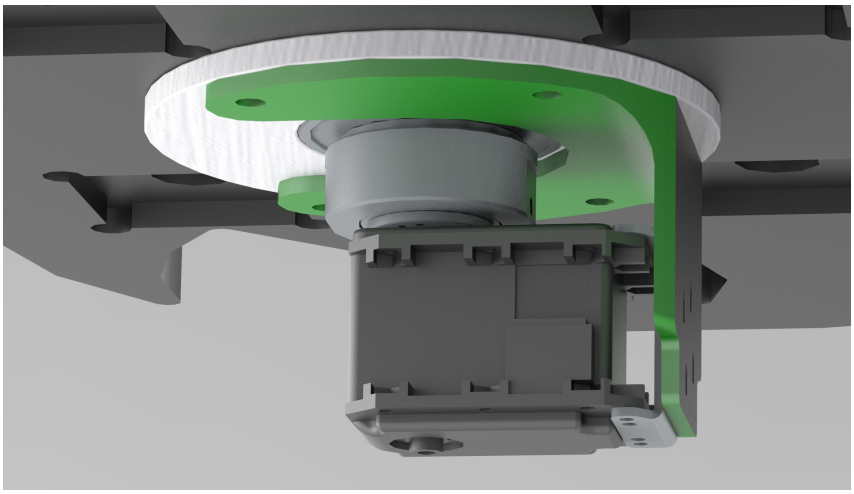


Figure 26: Pan-servo with mounting underneath shoulder plate

Projector

The developers of Furhat recently decided to change the current projector for a smaller one. This new projector is bought without a case and mount, which is why a case and a mounting had to be designed during the project.

A mock-up of the different parts was created in the 3D assembly due to the complicated design of the projector consisting of non-symmetric mounting holes, cylindrical parts and non-orthogonal angles, see Figure 38. This mock-up model was needed to prevent non-working prototypes of the case due to interference with the projector and correct hole placement for the power and video inputs. The mounting plate for the components was split in two to be easier to print with the used Ultimaker 3D-printer. On one side of the plate the projector with cooling was mounted, this side also consisted of the mounting bracket and a hole for the wide-angle lens to be mounted in. Due to the projection being at an angle, with the lower edge of the image being projected parallel to the projector and the upper edge at an angle of 22 degrees, a mount with an angle of 11 degrees was therefore created, see Figure 40. The other side of the plate was where the main board with all the connections and buttons was mounted. These plates together made a rigid and easy to print assembly, see Figure 27 and 37b.

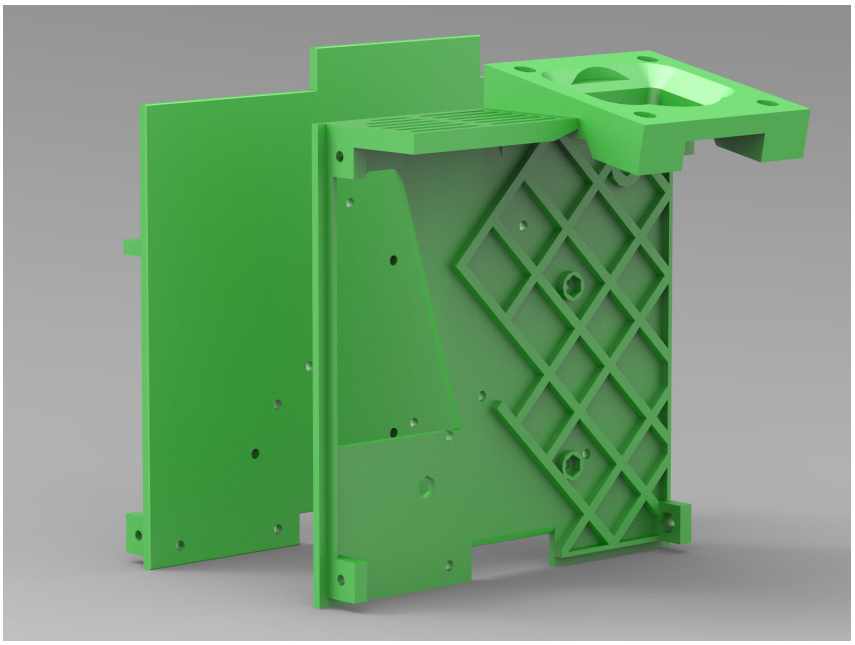


Figure 27: Mounting plates for projector

Another problem that needed to be considered was the exposed and fragile components that needed to be protected with a case. Considerations were made so that the buttons and connections still were accessible, see Figure 28. On each plate there were brackets for mounting the protective casings. Due to the heat created by the projector the included heat-sink needed to be cooled, the included fan needed to be mounted and air vents needed to be incorporated into the case design, see Figure 37a. Due to that the head was able to move in two degrees of freedom (pan/tilt), the rigidity of the case needed to be considered and improved. The cables attached to the projector would otherwise bend the case and move the projected image, distorting the face. Because of this and the possible need to remove and reinstall parts and covers during the design process, screws and nuts had been chosen as the fastening method and would also increase the rigidity. Some parts had been made thicker, removing sharp edges and moved to increase the final rigidity of the whole assembly. Space within the neck of the Furhat was also limited and thus the projector with its case was made small and clever so that it would fit and would not create any issues while the head was moving.

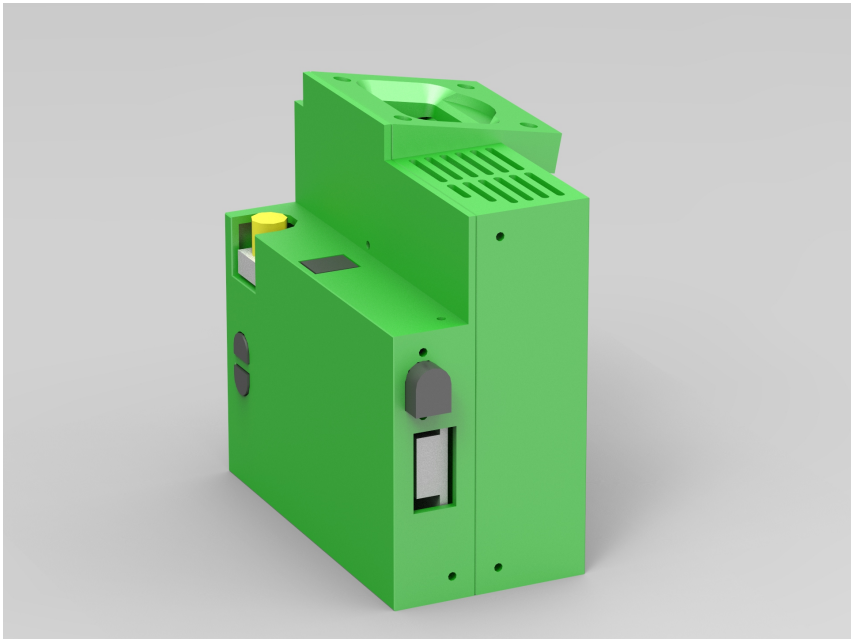


Figure 28: Complete projector assembly

Aesthetics

Since the exterior of the robots were supposed to be a mix between machine and human, paraphernalia such as ears were added to the head, see Figure 23 and Figure 25.

3.2 Hardware

This section presents the hardware used to construct the robots.

3.2.1 Arms

Shoulder motors

One of the main challenges in order to achieve a functional arm is to choose sufficient motors according to the specifications listed in section 1.5 for the shoulder. Firstly, a rough estimate of the forces that affects the shoulder was conducted. Subsequently a 3D-CAD model was developed and analyzed. From the 3D-model, with the right dimensions and materials a mass center could be deducted along with its distance from the rotational axis. Accordingly, the torque needed to rotate the arm could be calculated.

The torque needed to successfully lift the arm from the shoulder in the most demanding position was thus approximated to 2 Nm. As a security measure, motors with a continuous torque capacity higher than 5 Nm was searched for.

A great number of motors were analyzed in order to compare parameters versus price; this way the best suited motors according to specifications and budget could be found. The result of the analysis is listed in Table 4.

Table 4: Analyzed shoulder motors.

No:	Name	Torque (Nm)	Speed (rpm)	Current (A)	Cost (USD)
1	Transmotec PD52103	10,6	13	1,6	466
2	DOGA DO16841453B00	32	22	11	659
3	DOGA DO25937103B00	20	22	6	549
4	DOGA DO31936603B00	8	45	6	175
5	DOGA DO31938622B00	9	45	3	180
6	DOGA DO25837123B00	12	40	6	325
7	Drive System Europe hg192599-62	10	8,4	2,1	270

Motor No: 5 shown in Figure 9 was chosen due to its good specifications in comparison with its price. The torque provided by the motor exceeds the minimum requirements stated by earlier analysis. The relative high speed of the motor enable a wider spectrum of movement, aiding the aim to achieve human-like movements. Motor No: 4 share most of the abilities of the chosen motor, but the doubled current consumption would have provided unnecessary large costs in terms of power supply costs. All in all a more than sufficient motor with low current consumption and price was therefore chosen.

Elbow motors with encoders

Due to the versatile solution obtained by the power supply and voltage regulator, a greater freedom in the choice of elbow motors was enabled. The main requirements for the choice of elbow motor was silent drive and the ability to provide sufficient torque in order to move the lower arm. Motors with and without encoders were analyzed. A design decision to use motors with encoders was taken relatively early in the process in order to enable feedback-control.

Initially a 12V motor with encoder was chosen and evaluated. It was sufficient in terms of moving the arm, however it was too noisy. Therefore a new type of motor was ordered that was less noisy and had a higher safety margin in terms of maximum torque. This enabled a silent drive and increased the robustness of the system. Finally a design decision was taken to use the 24V motor with encoder in order to reduce current consumption and enable feedback control.

The first motors that was implemented on the robots was *Pololu 25D mm Metal Gearmotors* which had a specified torque output at 1,4Nm with 36rpm at 6V. It also consisted of a encoder with 8246 pulses per revolution of the gearbox's output shaft. During testing of the elbows it got apparent that the motor where not able to perform, the angular velocity got well beneath the specified rpm for the torque it was needed to apply while reaching its stall current. And with a robust solution in mind, if the motors would deteriorate by a slight margin the elbow rotation might have stopped working at its point of largest moment. It was thus decided to order new motors that would be able to deliver enough torque and with a higher safety factor. Because of the old motors ran at 6V there was a need for a voltage regulator to reduce the robots otherwise 24V power supply. To reduce the complexity a motor that ran at 24V was also a criteria. The new motors *DC3176-2744* was ordered from Promoco and had a rated voltage of 24V with 40rpm at 3-4Nm with an encoder that had 2048 pulses per revolution of the gearbox's output shaft. The new motors was then tested and performed very well and was far from reaching its stall current at the point where the moment was at maximum.

Hand servo motors

Due to the low weight of the 3D-printed hands a relatively small motor is sufficient to provide the necessary torque. Therefore a design decision was taken to use a motor of standard servo type. These motors are small, lightweight and does not require additional sensors for position control. Moreover they are low in cost and uncomplicated to implement.
The Parallax Standard Servo 900 shown in Figure 18 was therefore decided to be implemented as hand servo motor.

Drivers

The H-bridge-type drivers enable PWM-control of the motors. A critical aspect of choice of drivers is to obtain drivers that can operate with high currents and with more than one inde-

pendent channel. The result of a thorough search and analysis of appropriate drivers is presented in the table below.

Table 5: Analyzed H-bridge drivers.

No:	Name	Type	Continuous Current(A)	Voltage(V)	Cost(USD)
1	MD14	Single	5	24	47,99
2	MD22	Dual	5	24	97
3	MD49	Dual	5	24	52
4	RB-CYT-153	Dual	10	5-25	23

Driver No: 4 is chosen due to its good and versatile specifications in comparison to its low price. The dual ability also provides possibilities for a slimmer design.

Microcontroller Platform

The design of the arms with their power supply, sensors and actuators require a versatile platform. The platform is not only required to have a numerous amount of I/O-pins but also to have tools for serial communication. The Arduino Mega 2560 microcontroller provides such a platform. The main advantage of the chosen platform is moreover that it is compatible with Simulink and the code-generating tools thereof.

Microswitches

Microswitches are used to find end positions of the arms, initialize encoder parameters for the arm movements and sending a stop signal to the microcontrollers in case some external event triggers the arms to move to unrequested positions. The microswitches are dimensioned for maximum 125 volts DC, maximum 5 amperes (which is well above the 5 volt operating value of this application), uses a lever arm and can be operated with either normally closed or normally open function. The microswitches are mounted on the robots by the help of small 3D-printed plastic casings designed and manufactured by the project team. There are 16 switches in total that are distributed evenly to the arms of the two robots. The microswitches are used in normally closed mode to ensure that the switches are electrically connected to the microcontroller during operation conditions and opens the switch (closes the signal circuit) when the arms are heading for unwanted positions.

Voltage regulator

Four switched voltage regulators are used, two for each robot. In order to provide sufficient current to the hand servos without affecting the logic circuits one regulator per robot was used to convert the power supply of 24 volts down to 5 volts.

Potentiometers

Since the shoulder motors are not equipped with built-in encoders, rotating potentiometers are used as a simple yet effective solution to provide feedback from the arm movement to the control. Each potentiometer are mounted to register change of angle position in accordance with the rotation of the outgoing axis of the shoulder motor. With the use of absolute position, the arm's current position remain regardless if the robot is shut down.

3.3 Software

3.3.1 Java Application running on the Master Computer

The main Java application "interaction" runs on the master computer. The code for this application has been written by the Alan Project team and can be found in Appendix B.

The application includes the following Java classes, XML-, text- and XSD-files:

- *InteractionSystem.java:* This file includes the InteractionSystem class and the main function which calls the InteractionSystem class. The InteractionSystem class initializes:
  - Two system agents and their interaction distances
  - The language which the agents are speaking
  - The Kinect
  - The broker service (network)
  - The faces of the robots and the GUI
  - The voice of the agent connected to the main computer
  - The initial positions of the heads
  - The flow module: InteractionFlow

  Moreover the external text files are read and the starting signal of the motors in the head are also sent in this class.
- *Sentence.java:* This file includes the Sentence class. In the Sentence class the sentence, the agent number (including the information about which agent will read the sentence on that line) and the gesture number (including the information concerning which gesture, such as laughing or coughing, that will be made by the agent before it reads the sentence), which all are found on the same line, are separated and returned as either strings or integers.
- *SentenceSet.java:*

  This file includes the SentenceSet class. This class reads sentences from a text file and goes to the next sentence whenever the next function found in this class is called.
- *InterruptionSentenceSet.java:*

  This file includes the InterruptionSentenceSet. This class reads sentences from a text file and returns a random line from the text file.
- *InteractionFlow.java:*

  This file is made from InteractionFlow.xml by flow compiling.
- *sentences.txt:* This text file includes the sentences read during the dialog in order (not randomly chosen). An information line includes three columns separated by ";". The

first column includes the sentence, which will be read by the agent. The second column indicates which agent will read the sentence. Since only two agents exist this column can take values "1" or "2". The third column shows which gesture will be made by the agent before reading the sentence. Two different text to speech voices are implemented, of which only one of them (William, the male voice) has gesture audio files. Thus only the male voice (agent1) can make gestures such as coughing or laughing. The gesture information can be found in the flow file "InteractionFlow.xml".

Example of a line in this text file:

How do you mean?;1;2

Sentence = How do you mean?; Agent Number = 1 (First agent will read this sentence); Gesture = 2 (agent 1 will cough before reading this sentence.)

- *interruptionsentences.txt:* This file includes seven sentence lines, which will be read during the interruption of the robots by a visitor. These sentences are randomized by the InterruptionSentenceSet class. This text file includes the same information format as senteces.txt file.

  Example of a line in this text file:

  Shut up;1;1

  Sentence = Shut up ; Agent Number = 1 (First agent will read this sentence) ; Gesture = 1 (agent 1 will not do any gesture.)

- *InteractionFlow.xml:*

  This file includes the state chart of both agents and the flow of their behaviors, such as when the first robot will speak and when the robots will get interrupted. See Figure 29 for the Behavior State Chart. This XML file is compiled to the InteractionFlow.java file.

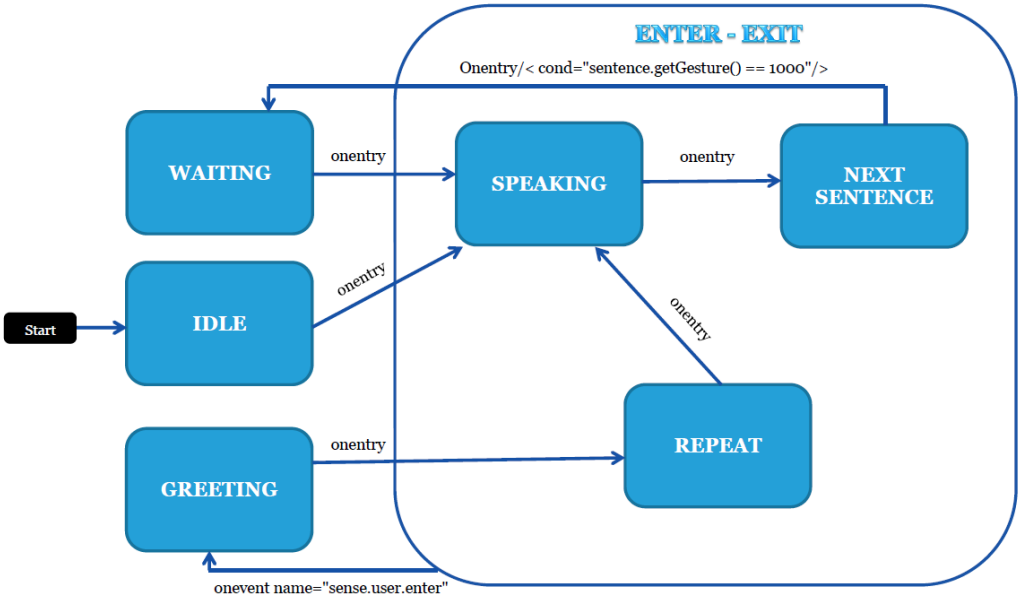


Figure 29: The State Chart of the Robotic Heads

The starting state of the behavior flow is the "Idle" state. In this state the textures of the faces and the head locations of the robots are initialized and then it moves to "Speaking" state. The "Speaking" state is the internal beginning state after the start of the program. (This state is required not to initialize the program features each and every time the dialog starts over). The agents moves to the state "NextSentence", in which the next sentence is executed. In the "sentence.txt" file the end of the dialogue is pre-coded as Gesture Number = 1000. Thus if the gesture number is equal 1000, the agents move to "Waiting" state. According to the gesture inputs in the "sentence.txt" file, firstly the gesture is made and the sentence is then read by the agent pre-coded in the "sentence.txt" file.

Example from an input line in the "sentence.text file":

How do you mean?;1;2

Sentence = How do you mean?; Agent Number = 1 (First agent will read this sentence) ; Gesture = 2 (agent 1 will cough before reading this sentence.)

The related code in the XML file:

```
<state id="NextSentence" extends="EnterExit">
  <onentry>
        <exec>sentence = sentences.next(); number = 0</exec>
          <if cond="sentence.getGesture() == 1000"/>
            <goto state="Waiting" />


          <else/>
```

```
9                 <if cond="sentence.getAgentNumber() == 1">
10
11              <if cond="sentence.getGesture() == 2" />
12              <agent1:say > <spurt audio="g0001_003">COUGH</spurt> </agent1:say >
13              <agent1:say text="sentence.getFullSentence()" />
14              ...
15    </onentry>
16 </state>
```

After reading the sentence the agents move to the "Speaking" state. There are 16 different gestures such as clearing throat, laughing or coughing. These gestures are read from a pre-recorded audio file. To see the gestures see "InteractionFlow.xml" file in Appendix B.

The EnterExit state is extented by the states "Speaking", "NextSentence" and "Repeat". In this state entrance of a visitor is detected and the speech of the agents are interrupted. Both agents will stop speaking (their sentences remain unfinished), turn their heads to the visitor and look into the eyes of the visitor. The agents will then move to the "Greeting" state. In this state, an interruption sentence is chosen randomly from the "interruption-sentences.txt" file and is read by the agent pre-coded in the same text file. The agents then move to the "Repeat" state. In the "Repeat" state the agents turn their heads and eyes to each other and the agent with the unfinished sentence says "What was I saying?" and repeats its unfinished sentence from the beginning. The agents will then move to "Speaking" state. The "Waiting" state allows the agents to wait for 15 seconds in silence till the next sequence of the dialog starts. The agents will then move to the "Speaking" state.

- *SystemAgentFlow.xsd:* This file includes the attributes of the agents. The file is taken from Furhat Robotics.
- *flow.xsd:* This file includes the attributes of the flow. The file is taken from Furhat Robotics.

3.3.2 Slave Computer Code "robot2"

The program in the slave computer includes only the files "Robot2System.java" and "SystemAgentFlow.xml". This program creates a system class which connects to the broker on the master computer to share the states and events created in the "interactionFlow.xml" file, which runs on the master computer. Moreover, the system class of this program initializes the graphical interface and the neck motor positions of the second robot head. The codes for this application, written by the Alan Project team, can be found in Appendix B.

3.3.3 Integration

With the usage of IrisTK [1] a lot of the functions used within the scope of the project are present, e.g. the possibility of interruption and receiving information about which robot is currently speaking. These functions needs to generate appropriate response in the arms by

communicating this to the Arduino's. IrisTK software has the ability to send JSON messages for a large amount of triggered events. The events that are needed to control the arms in a correct manner were identified.

To receive the events from IrisTK a program has been developed in Python. A socket is opened to connect to the IrisTK broker (which connects all running IrisTK software) and the events that are of interest are being subscribed too. When one of these events that the Python program is subscribing to triggers, a JSON message is sent from IrisTk and is received inside the Python program. The message is then deconstructed to find the relevant parts in the message and to give appropriate action. By subscribing to several events it was possible to limit the events that needs subscription down to two events, *action.speech* and *action.speech.stop*. The *action.speech* event is triggered when a new sentence is started on one of the robots, and in this message information about which agent that will conduct the sentence is contained. With the help of this information arm movement will start for the corresponding robot conducting a sentence. Because of how the IrisTK script is designed, the *action.speech.stop* which stops the current sentence will only occur when the robots are being interrupted.

When an interrupt occurs a special arm movement is triggered from the Python program. To get the robots' regular arm movement to start again after the interruption sentence, the sentences that are going to be spoken are extracted from the JSON message. If the sentence is one of the two that are spoken when the interrupt is completed the arm movement will return to its randomized pattern. The sentences are found in the *action.speech* event.

To send information to the Arduino's a serial communication is started on each COM-port of the Arduino's with a baud rate of 57600. The Arduino's are programmed to work in a specific mode corresponding to the signal received. Because of how the Arduino blocks are implemented in Simulink, a received byte is interpreted as a *char* character. The different modes are: randomized arm movement, resting position, interruption routine and individually positioning of each motor.

The program then follows the following routine: if an *action.speech* event is received, the agent that will be speaking is identified, the randomization mode byte is sent to that robot's Arduino's and the other robot's Arduino's receives a resting mode byte. When the interruption event *action.speech.stop* is received, robot 1 will receive the interruption routine pattern on its right arm Arduino, while all the other Arduino's receive a resting mode message. Once the corresponding sentence that is triggered when the interruption routine is over, the robots will continue with their arm movement.

To get a visualization of each event that is triggered the program prints out information about the events in form of which agent is conducting speech, that an interruption has occurred and that the interruption sequence is finished.

3.3.4 Code generation

In order to run the system controlling the arms, which is developed and modeled in Simulink, the program has to be able to run on the Arduinos. Simulink Coder is a feature in Simulink able to generate C and C++ code from Simulink models and Matlab functions. The generated code can be flashed onto the Arduino using the Arduino support package available as a download for Simulink. Using the support package enables the internal functions used in the Arduino and the possibility to run the Simulink program in external mode, which enables interactive real-time parameter tuning and monitoring as the program runs on the Arduino board. In order to use and implement the necessary internal Arduino functions, the S-function Simulink block has to be implemented. In order for the code to be flashed on to the Arduinos the Simulink solver has to be discrete along with integration calculations and time monitoring. The discrete sample time 0.002 is used for all calculations in the program.

3.3.5 Arm Movement Software

The software designed to obtain control of the arm is Simulink model-based. The main advantages with this approach are simulation possibilities, robustness and flexibility for change in design parameters and useful code generation possibilities. An overview of the system controlling one arm is depicted in Figure 30.

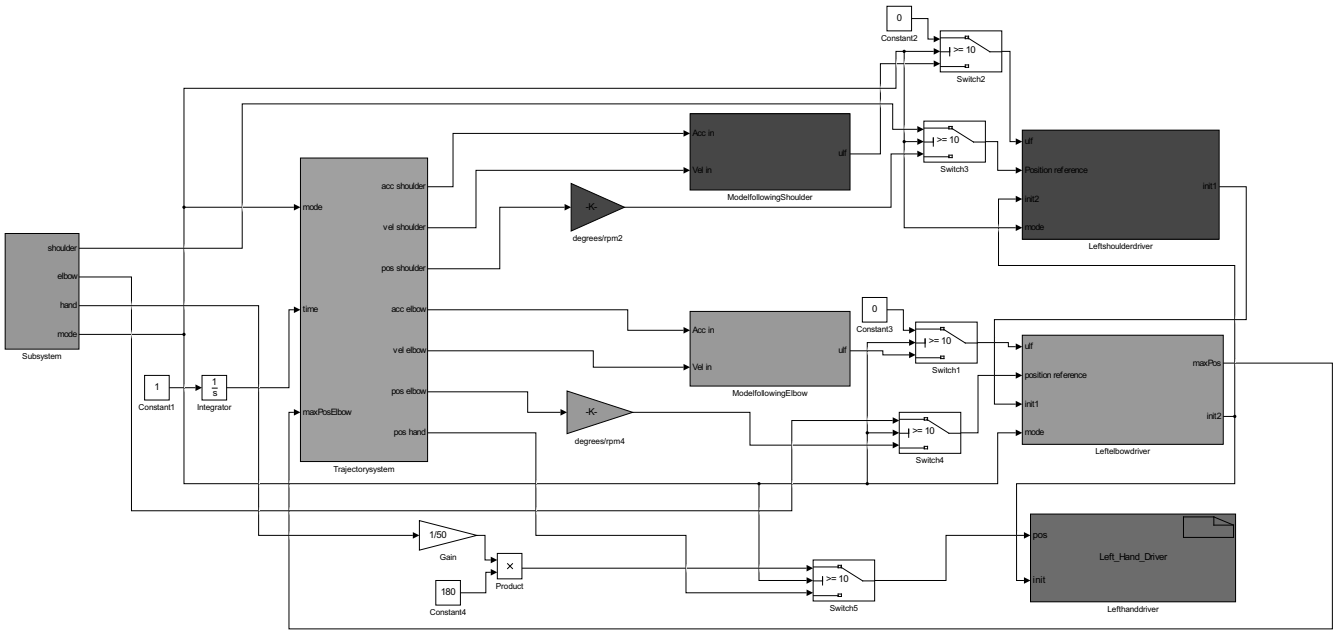


Figure 30: Simulink model of one arm

The system consists of a number of sub-systems handling serial communication, trajectory planning, model following systems, PID-controllers, sensor monitoring, motor actuation and error

handling needed to obtain a fully functional arm. Only slight adjustments of a small set of parameters allow the model to be applied to the left or right arm on one of the robots. The three differently shaded blocks on the right in Figure 30 represent the subsystems handling shoulder, elbow and hand movement and control respectively as well as the corresponding sensor monitoring. The corresponding code to each embedded MATLAB function and S-function in Figure 30 - 34 are shown in Appendix B.3.

The communication subsystem module depicted in Figure 31 handles serial communication with the main program controlling the communication between the two robots.

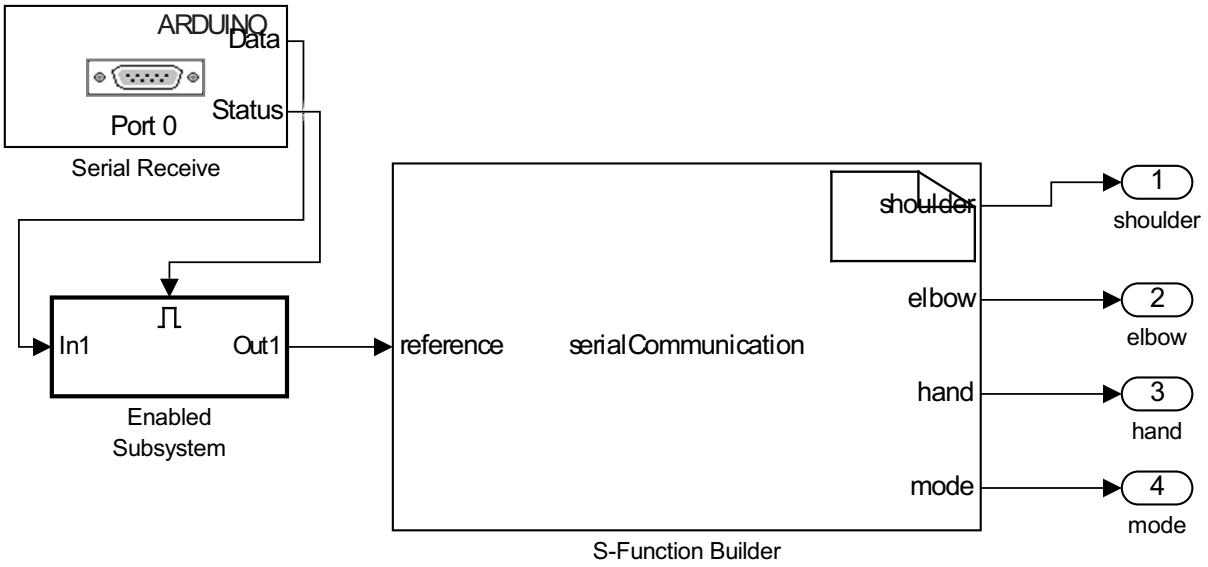


Figure 31: Serial communication module for one arm

The communication module uses the built in Arduino Serial Receive block in order to receive messages from the main computer. The received message is evaluated in the serialCommunication S-Function which thereafter sends commands to the rest of the system. The command to commence movement from the communication module is received by the trajectory subsystem which is depicted in Figure 32:

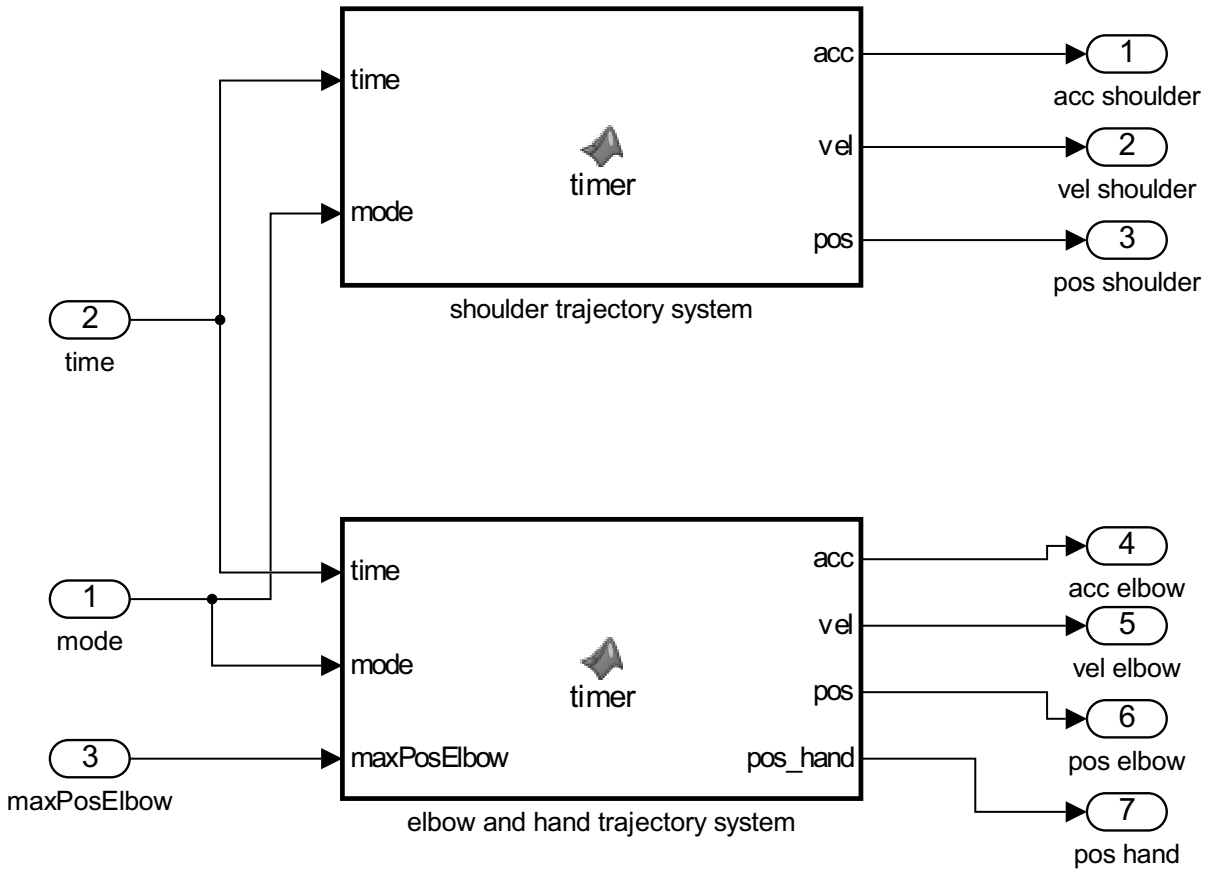


Figure 32: Trajectory system for one arm

Figure 32 consists of two embedded Matlab Functions containing the code of the trajectory planners for the shoulder, elbow and hand respectively. The main tasks of the functions are to continuously generate the position reference and the corresponding velocity and acceleration. The acceleration and velocity signals are the inputs of the model following subsystems generating the feed-forward control signal while the position references are reference inputs to the feedback control system.

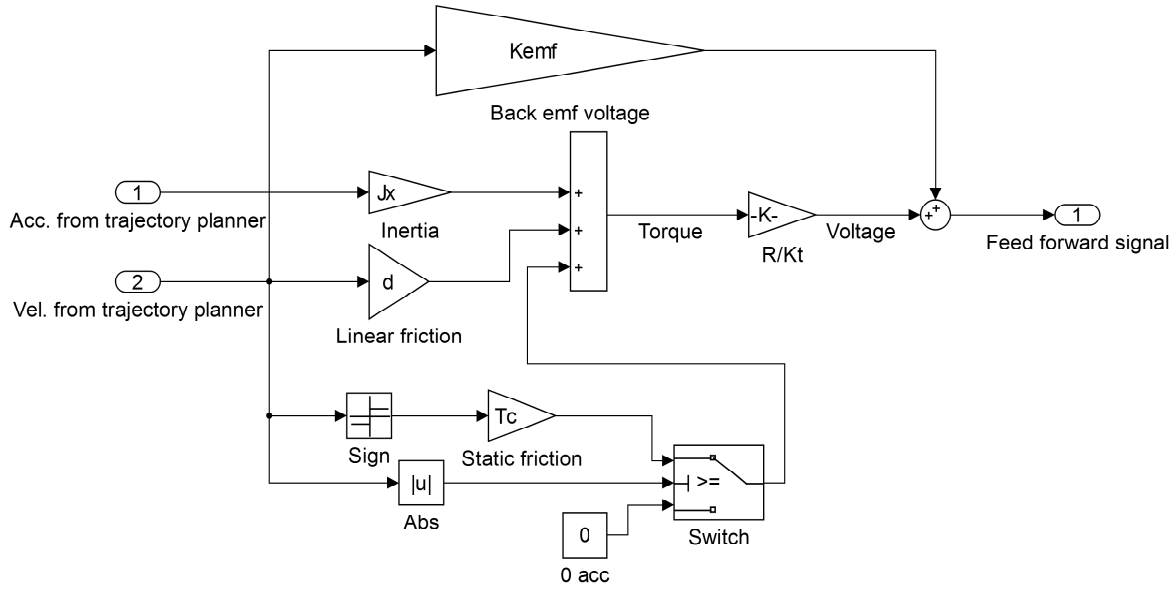


Figure 33: Model following sub system

The model following part depicted in Figure 33 is derived from the dynamic equations describing the arms dynamic behavior (see section 2.1.1 and 2.1.2). The different model following sub systems (shoulder and elbow) only differ in terms of parameter values and are otherwise identical.

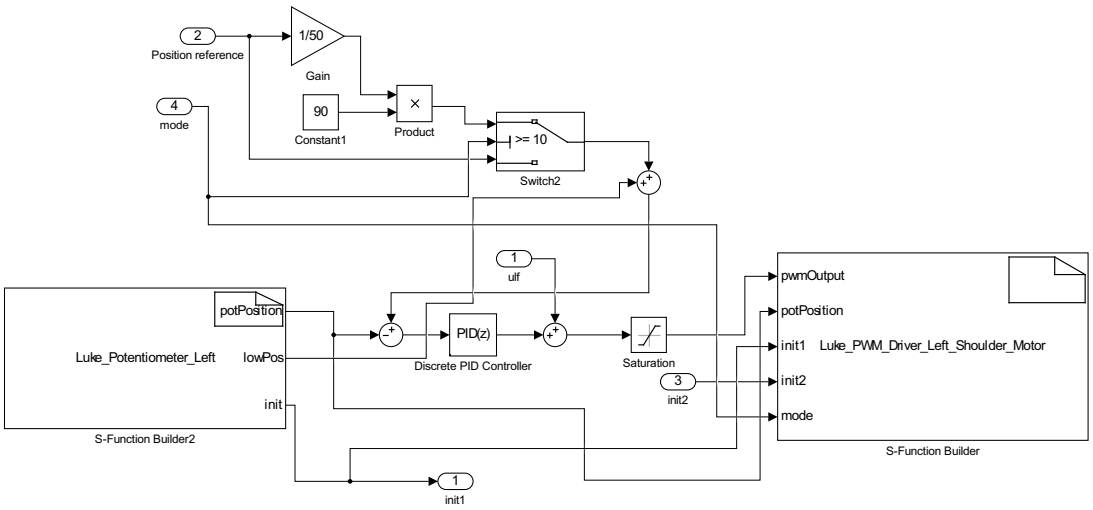


Figure 34: Left arm shoulder driver block containing sensor monitoring, feedback control and motor actuation

Figure 34 displays the final step of the system where the reference signal is compared with the signal from the sensor thus generating an error-signal to be processed by the PID-controller and driver block that updates the actuator.

Two different sets of S-function blocks are implemented in the program; one assigned as a motor driver function block and the second as a sensor monitoring function block. Each sensor (potentiometer or rotary encoder) has its own S-function block assigned to it. The same goes for each motor having its own S-function assigned. The sensor S-function assigned for the potentiometers defines and assigns the analog input pins for the potentiometer as well as outputting a calculated degree value based on the absolute position of the potentiometer. The corresponding sensor S-function assigned for the encoders assigns the two different digital input pins corresponding to the two encoder channels A and B, and outputs a degree value based on the number of calculated pulses relative to the starting position of the encoder. A separate header file is included in the S-function containing the code calculating the correct pulse count based on the two encoder channels.

The motor driver S-function blocks defines and assigns the PWM and direction output pins on the Arduino corresponding to the associated motor as well as setting the register defining the PWM frequency (20 kHz on the shoulder and elbow motors). The input to the driver S-function blocks is the output from the control system. Based on these values the corresponding duty cycle of the PWM signal output is calculated as well as the direction output.

The motor driver block also incorporates the initialization phase of the motors which is designed to locate the starting and ending position of the upper and lower arm using the microswitches. The initialization phase is run when the robots are started and restarted to ensure the correct position of the arm which negates potential hazards due to incorrect positioning.

4 Result

The project has resulted in two human-sized robots able to achieve what is specified in requirements *one a-e*. The arms where provided with an additional degree of freedom, resulting in three degrees of freedom in total, instead of the minimum of two specified in the second requirement. In Figure 35, a picture of the resulting robots can be viewed.

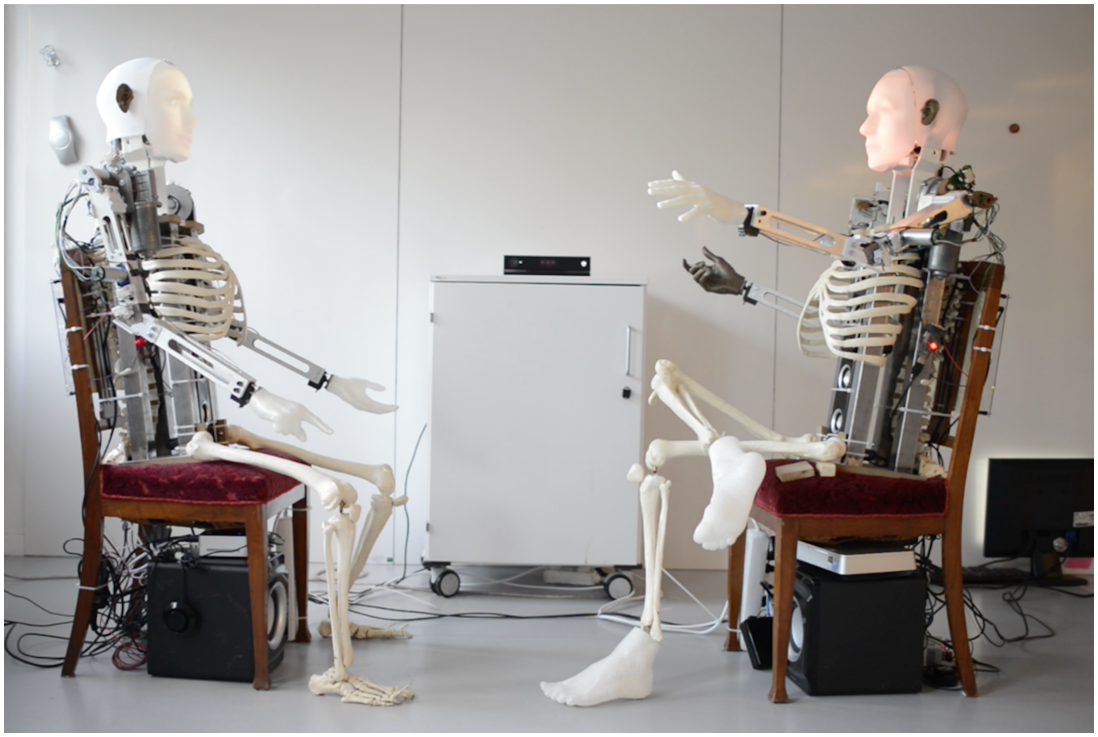


Figure 35: Picture of the two robots during the final presentation of the project at KTH

The sixth requirement, that the robots shall be durable and robust is a requirement that is hard to verify, but the team has considered this during the design and manufacturing stage of the robots. The electronics that deliver voltage, current and control signals to the motors controlling the arms are gathered on a plate at the back of the chair to prevent unnecessary exposure, and protected by a plastic plate mounted a few centimeters from the hardware and the related cabling, see Figure 36.

In the final design, as can be seen in Figure 35, the robots as well as most of the hardware are fastened to the chair. This makes it easy to transport and unnecessary to disassemble. The lower arms should however be disassembled to prevent them from being harmed during transportation, and a detailed exploded view of how this joint is assembled can be seen in Figure 13. Requirement 3 is thereby fulfilled, with detailed instructions on how to assemble or disassemble the one part in need of this.

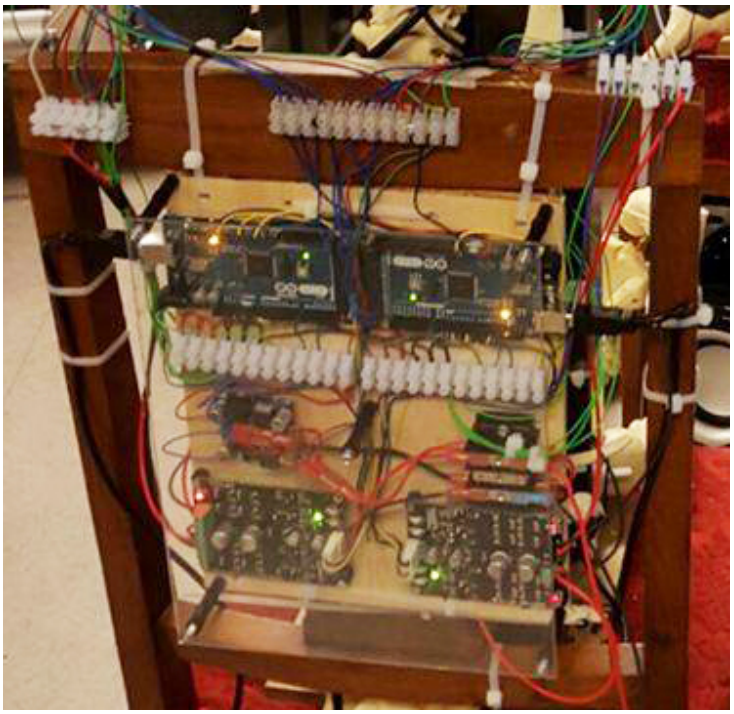


Figure 36: Plate where the motor-controlling hardware are gathered and protected

Documentation on how to operate, install and trouble shoot the robot software is exhaustively explained in appendix A.

Duplicates for all the parts that could possibly wear out have been manufactured. Extra motors and projector are also included as backup parts in order to prevent long delivery time if something would break down.

5 Discussion and Conclusion

5.1 Arms

The arms have some flexibility in them, but with a good control of the motor this is not too visual. With the usage of a belt drive some extra flexibility is built into the lower arm, so when there is a sudden movement in the arm there is some backlash of the lower arm. The upper arm extensions to the elbow joint are flexible and must be mounted and adjusted to be parallel to the arms, otherwise they can come into contact with the lower elbow connector and give friction and noise in the movement. The elbow connector could be made smaller on the rod to eliminate this risk of unwanted friction. The arms were at first designed for the first elbow motors that was bought, and to reduce the manufacturing time and cost the same arms were used with the new motors. The different motors had different ways of fastening them to the arm. The first motor had two threaded holes while the new motors had four, and the holes where placed on different circumferences. To accommodate this design towards the new motors only two screws could be mounted on the new motors, with a redesign it would be possible to tighten the motors with four screws. However, with the implementation of motor fasteners that aligned the motors orthogonally to the arms it may be insufficient to use more than two screws. One more possible improvement for this is that the screws currently interfere with the adapter for the lower sprocket. By extending the screw holes for the motor away from the output shaft it would be possible to get these screws not to interfere with the sprocket adapter. The elbow joint is also a bit cumbersome to assemble and disassemble. When the belt between the sprockets is tightened the larger sprocket mounted on the elbow joint got slightly misaligned when the lever was 3D printed. With the lever manufactured in plate instead this misalignment got reduced.

5.2 Motor control

The early decision to implement the arm as a Simulink control model is concluded to have been successful. The model provided a good environment and enabled the design of the motor control model. In the end this also contributed to decrease the overall effort even though a great amount of time was spent on the start-up of the system, including the design of a the arm model.

5.2.1 Main Design

The main advantages of the chosen method is listed below:

- Adaptability to new conditions
- Simulation ability
- Controllable system
- Integration of additional C Code
- Code generation

The model enabled a good tool to cope with changing parameters of the modeled system. For example, a design decision had to be remade in order to adapt to a heavier arm than expected, which only added a minor change of parameters within the model as additional work effort. The simulation ability that was enabled by choosing Simulink as design environment had a positive effect, especially during the early design phases. By simulating different events and behaviors a greater understanding of the system as a whole could be obtained. Moreover, dangerous behaviors of the implementation could be identified and taken into account for safety precautions.
Due to budget reasons the shoulder motors could not be bought with positioning encoders. The design and implementation of an encoder based on a potentiometer enabled positioning feedback of the shoulder motor, thus enabling control-ability of the whole arm.
The implementation of the motor control was not only designed to be a continuous model of movement. Therefore additional C-code needed to be implemented within the model in order to cope with interrupt routines such as the occurrence of individuals too close to the robots. S-functions were therefore implemented successfully in order to handle those interrupts.
The code generation ability of Simulink which enabled code to be flashed to Arduino hardware, is concluded to be a successful method mainly due to the correctness of the code generated, thus saving time of error-handling.
Even though the advantages overtake the disadvantages of the implementation of a motor control model in Simulink it can be argued that it may result in complications with optimization. If implemented right, perhaps a more efficient code could have saved computing capacity, thus enabled savings in hardware costs. However, due to the concept of only two customized robots that are not planned to go into mass-production, a lesser focus on optimization in terms of efficiency could be accepted.
The servos mounted to actuate the hand movements lack feedback to the control system. The design decision could however be motivated since the actuator is of standard-servo type and therefore is self-controlled. It could be discussed whether a motor model of the non-standard servo could have been more appropriate to implement in order to achieve more human-like movements.

5.2.2 Hardware

The choosing of the motors was an iterative process. The set of motors that finally was chosen and implemented is concluded to cope with the requirements stated in chapter 1.5 regarding stakeholder and system requirements. The initial set of elbow motors had enough torque to move the arms as required. However, the final choice of motors with higher torque have an increased level of robustness, thus decreasing the probability of breakdown during the exhibition. Moreover the final choice of elbow motors are less noisy adding a more natural appearance for spectators.
The shoulder motors first ordered were less noisy than expected, that in combination with pure aesthetic aspects generated a second iteration in the choosing of shoulder motors. Thus a smaller model was chosen since it still provided the torque needed for the applications of the project. Another aspect thereof was that the cost of the motors was reduced significantly, resulting in

more freedom within other design decisions.

The circuitry designed to enable the necessary connection and control of the arms as a system is concluded to be both robust and durable. Due to the demands of long-term durability a lot of effort was put into a robust solution. The cabling could from an engineering perspective be improved significantly in terms of discreetness. However, the decision to maintain the cabling visible was taken due to aesthetic reasons after discussions with the the main stakeholder Tove Kjellmark.
The implementation of micro-switches introduced safety aspects, especially in order to avoid self-harming behavior due to unexpected power-loss or faulty signals sent from the system. These micro-switches are concluded to fulfill their mission and enhances durability and robustness according to requirements in chapter 1.5.

5.2.3 Control method

The applied method of combining a feed-forward trajectory-planner with a PID feedback controller, generating an output-feedback control structure serves its purpose well. The trajectory planner was introduced to control the reference signal of the system in order to obtain smooth human-like movements. The method is suitable for this application and the PID-controller follows the reference signal without overshoot, thus coping with the requirements stated in chapter 1.5.

5.3 Software

The integration python software can be run once the broker is started and will continue running even if the IrisTK software for each robot is restarted. The Arduino's are also able to be reset to restart the initialization routine without interfering with the integration software.

Known bugs in the program are:

- Exiting the program with ctrl+c only triggers when a new event occurs.
- If an Arduino is plugged out when the program is running, the program will not be able to restart until exiting the program and re-inserting the Arduino that was unplugged prior.
- The JSON message is sometimes not received as a whole message. Some bytes are received in an earlier receive routine, and the rest of the message is received in the new iteration. The impact is that an event is not received correctly, and occurs approximately 1% of the times.

With the exception that an event is sometimes not received correctly, the program works satisfactory and without problems concerning the communication with the Arduino's. With the implementation of individual control of the motors through the python software, a third party

would be able to set their own arm-moving patterns without interfering with the programming of the Arduino.

6 Future work

There are some possible improvements that could be done if further work would be conducted in this project:

- Improved arm movement (model, trajectory)
- Additional DOF in arms
- New vacuum formed faces

  The faces were rather difficult to produce. Since the face is very uneven it is hard to make the plastic follow the contour exactly, which is why the nose and lips are a bit wider than they are intended to be. This is almost not noticeable, but can still be improved.

- Sound level measurement

  The idea was initially to have a sound measuring system that would be a part of the interacting system of the robots. The measuring system would send a signal to the robots if the surrounding noise exceeded a certain level, in which case the robots would tell the audience to keep it down.

- Design of arms

  A redesign of the elbow joint should be used to make the assembly easier to disassemble if a part needs to be fixed. Also a tighter tolerance between the large sprocket and bearing would ensure that the sprocket is parallel to the belt and orthogonal to the elbow joint shaft.

- Projector
  - The parts could be 3D-printed as one piece or molded. This in an effort to save time and effort during assembly and lower the number of needed parts. For this project there is only two assemblies required, thus it would not be cost or time effective to make this change in manufacturing.
  - To lower the time and effort spent when starting the robots a computer-controlled switch of the projector-power could be installed. It would then be possible to power on the projector as a part of the prescripted startup procedure and also turn of the projectors when not needed.
  - A fan speed regulator could be implemented for the cooling system so that the life span of the fans could be increased and the noise reduced. This requires a system that monitors the temperature so that the projector never gets too hot. The other benefit of a temperature monitor would be being able to detect malfunction of the fans or excessive heat of the projector itself, in turn the projector could be turned off automatically.

  - As a fastening method snap mountings could be designed instead of screws. As a result this would lower the price and make the projector easier and faster to assemble.
  - Due to problems with one projector a new one needed to be bought from the Furhat-developers. This new projector is the same as the earlier but with another projection angle, the correct angle and position of the new projector could therefore be achieved by redesigning the projector mount. The projection still works but does not cover the face as good as the earlier projector.
  - Calculate and design the best way to mount the projector, the wide-angle lens and the mirror in relation to each other and the face so that a high resolution and coverage of the face is possible. Also it might be possible to place the projector within the Furhat head and thus not having it sticking out of the neck, as it needs to with the current placement. As of now some of the projection is spread wider than the face, resulting in a lower resolution of the face than what could be achieved by the projector.

- A more robust development of the python program.

  The python software should be developed further to receive the JSON messages in a correct manner each time. Since the Arduino's are assumed to never change their COM-port, the COM-ports of each Arduino are hard coded in the program. This could be developed further with the software asking the Arduino which arm of which robot it belongs to.

  Since there is some drift in the elbow position over a long time of running, a serial reset should be implemented. This will reset the Arduino's, making them rerun the initialization routine and finding their edge positions again. Once the whole dialog is completed the robots pause for 15 seconds, so a resting arm position should be implemented and after approximately three iterations a serial reset can occur.

7 Bibliography

[1] Furhat ROBOTICS, Guide to IrisTK.
`http://iristk.net/overview.html`, Dec. 2015

A Software Manual

A.1 Starting Guide of the Head Programs

Before you start be sure ethernet cables are connected to router and to the computers.

1. Start Broker service on the Master Computer(Luke):

   - Open “command promt” by pressing windows button and writing cmd to the search box.

   - Write “iristk broker” to command prompt and press enter

2. Start serial communication to the Arduino’s

   - Open ”command prompt” and write ”py -3 furhatClass.py” and hit enter.

3. Start Robot2 program on the Slave Computer (Leila):

   Option 1: Open “command promt” and write “iristk robot2”

   Option 2: Open “eclipse”, find “app/robot2/src” from the package explorer(usually on the left.). Expand the files and Find “Robot2System.java”. Press run button. Be sure the face is started before going to next step.

4. Start Interaction program on the Master Computer(Luke):

   Option 1: Open another “command prompt” window, without closing the one on, which you started the broker and write “iristk interaction”

   Option 2: Open “eclipse”, find “app/interaction/src” from the package explorer(usually on the left.). Expand the files and Find “InteractionSystem.java”. Press run button.

Trouble Shooting Menu

- If you get any error related to broker,

  - Be sure Ethernet cables are connected to the router and the computers

  - Check if the router is plugged in.

  - Close the programs and the broker and repeat the steps mentioned in the program run manual.

- If you get any error related to speech package,

  - Check if the speakers are connected to computers and plugged in.

- If the face seem to be misaligned on the mask,

  - Check the orientation of the screens from the screen resolution.

- Close the programs and the broker and repeat the steps mentioned in the program run manual.

A.2 Installation Guide of the Programs from Backup Folder

These steps should be followed both on the master computer and the slave computer.

1. You can find a backup files on the Flash Disk .
2. Uninstall all the Java versions from the Control Panel.
3. Delete Eclipse and IrisTK folder from C:/.
4. (Be sure Microsoft Visual Studio 2010 is installed on the computer)
5. Install Java 8 file " jre-8u65-windows-x64.exe"
6. Install the java 7 file " jre-7u79-windows-x64.exe"
7. Setup the Java path of both Java 8 and Java 7 to the environment variables.
8. Copy "Eclipse" folder to C:/ from the backup folder.
9. Copy "IrisTK" folder to C:/.
10. Open "command prompt".
11. Go to folder C:/IrisTK.
12. Write "iristk install" to install iristk
13. Write "iristk eclipse" to setup the eclipse path
14. Import IrisTK files to Eclipse (Open eclipse and go to $File \rightarrow Import \rightarrow General \rightarrow ExistingProjects$. Find IrisTK folder from the select root directory.)
15. Adjust Java 8 as the JRE to Eclipse ($OpenWindow \rightarrow Preferences \rightarrow Java \rightarrow InstalledJREs$. Add and find Java 8 and choose it as the default)
16. Add Libraries ($OpenProject \rightarrow Properties \rightarrow JavaBuildPath \rightarrow Libraries$ →Add Jars. Expand $IrisTK \rightarrow core \rightarrow lib$. Add the file org.apache.commons.collection.jar)
17. Install Dynamixel Wizard "RoboPlusWeb (v1.1.3.0).exe"
18. Note down the motor positions, when the head is in the middle.
19. Open command prompt and write "iristk furhat-server"
20. Adjust face and neck parameters from the furhat server and save.
21. Install the Voice "CereVoiceLauren.exe"

Additional Steps should be done on the master computer

- Install Kinect " KinectSDK-v2.0˙1409-Setup.exe "

- Install Python “ python-3.5.0 ”
- Install PySerial for “ python 3.5.0 ”

B Software

B.1 Java Application on the Master Program: Interaction

InteractionSystem.java:

```
/*******************************************************************************
 * Copyright (c) 2014 Gabriel Skantze.
 * All rights reserved. This program and the accompanying materials
 * are made available under the terms of the GNU Public License v3.0
 * which accompanies this distribution, and is available at
 * http://www.gnu.org/licenses/gpl.html
 *
 * Contributors:
 *     Gabriel Skantze - initial API and implementation
 *     Elif Toy- implementation of the java application: interaction
 ******************************************************************************/
package iristk.app.interaction;

import iristk.situated.SituatedDialogSystem;
import iristk.situated.Situation;
import iristk.situated.SituationPanel;
import iristk.situated.SystemAgent;
import iristk.situated.SystemAgentFlow;
import iristk.speech.*;
import iristk.system.IrisGUI;
import iristk.util.Language;
import iristk.app.interaction.InteractionFlow;
import iristk.app.interaction.SentenceSet;
import iristk.app.interaction.InterruptionSentenceSet;
import iristk.cfg.SRGSGrammar;
import iristk.flow.FlowModule;
import iristk.furhat.FurhatModule;
import iristk.kinect.*;
import iristk.speech.cereproc.*;


//Definition of the System Class
public class InteractionSystem {

  private WindowsSynthesizer synth;

  public InteractionSystem() throws Exception {
    SituatedDialogSystem system = new SituatedDialogSystem(this.getClass());

    //SystemAgentFlow systemAgentFlow1 = system.addSystemAgent();
    SystemAgentFlow systemAgentFlow1 = system.addSystemAgent("agent1");
    //Adjusts the interaction area(distance)
    systemAgentFlow1.getSystemAgent().setInteractionDistance(1.5);
    systemAgentFlow1.getSystemAgent().setMaxUsers(1);
    SystemAgentFlow systemAgentFlow2 = system.addSystemAgent("agent2");
    //Adjusts the interaction area(distance)
    systemAgentFlow2.getSystemAgent().setInteractionDistance(1);
    systemAgentFlow2.getSystemAgent().setMaxUsers(1);
    //new MyApplicationFlow(systemAgentFlow1, systemAgentFlow2);

    //Language
    system.setLanguage(Language.ENGLISH_US);

    //Graphical Interface
```

```
55      system.setupGUI();
56
57      //KinectSetup
58      system.setupKinect();
59
60      //Network Setup
61      system.connectToBroker("myticket", "127.0.0.1", 1932);
62
63      //AgentSetup
64      system.setupFace("bertil", new CereVoiceSynthesizer(), "agent1");
65
66      //synth = new WindowsSynthesizer();
67      //system.setupFace("beata", synth, "agent1");
68      //synth.setVoice("lauren");
69
70      //Read of external files: sentences.txt
71      InterruptionSentenceSet interruptionsentences = new InterruptionSentenceSet(getClass
        ().getResourceAsStream("interruptionsentences.txt"));
72      SentenceSet sentences = new SentenceSet(getClass().getResourceAsStream("sentences.txt
        "));
73
74      system.addModule(new FlowModule(new InteractionFlow(sentences,interruptionsentences,
        systemAgentFlow1,systemAgentFlow2)));
75
76      //Position of Head
77      system.loadPositions(system.getFile("situation.properties"));
78      system.sendStartSignal();
79      }
80
81    //main Program
82    public static void main(String[] args) throws Exception {
83      new InteractionSystem();
84    }
85  }
```

Sentence.java:

```
1
2  package iristk.app.interaction;
3
4  import java.util.Arrays;
5  import java.util.List;
6
7  import iristk.util.RandomList;
8
9  //Sentence Class defines where the sentence is in the text file and returns it
10 public class Sentence {
11
12   private  String id;
13   private String sentence;
14   private Integer agentnumber;
15   private String agentnumber_s;
16   private Integer gesture;
17   private String gesture_s;
18
19   //Defition of sentence at the text file
20   public Sentence(String id, String[] cols) {
21     this.id = id;
22     sentence = cols[0].trim();
23     agentnumber_s = cols[1].trim();
24     gesture_s=cols[2].trim();
25     agentnumber = Integer.parseInt(agentnumber_s);
26     gesture= Integer.parseInt(gesture_s);
27   }
28
29   public String getId() {
30     return id;
31   }
32
33   public String getFullSentence() {
34     return sentence;
35   }
36   public Integer getAgentNumber() {
37     return agentnumber;
38   }
39   public Integer getGesture() {
40     return gesture;
41   }
42
43 }
```

SentenceSet.java:

```
1
2  package iristk.app.interaction;
3
4  import iristk.util.RandomList;
5  import java.io.*;
6  import java.util.*;
7
8  public class SentenceSet extends ArrayList<Sentence> {
9
10   private int n = 0;
11
12   //Read from text file
13   public SentenceSet(InputStream sentenceFile) {
14     try {
15       BufferedReader br = new BufferedReader(new InputStreamReader(sentenceFile));
16       String line = br.readLine();
17       int qn = 0;
18       int ln = 0;
19       System.out.println("Reading sentences");
20       while ((line = br.readLine()) != null) {
21         ln++;
22         if (!line.matches("[A-Za-z0-9,\\.;\\?'\\- ]*")) {
23           System.err.println("Illegal line " + ln + ": " + line);
24           continue;
25         }
26         String[] cols = line.split(";");
27         if (cols.length < 3) {
28           System.err.println("Not enough columns in line " + ln + ": " + line);
29           continue;
30         }
31         Sentence q = new Sentence("q" + qn++, cols); // creates:  sentence (line number,
     the written sentense )
32         add(q);                    // adds one sentence to the total file that has been
     read in
33       }
34       System.out.println(qn + " sentences read");
35
36     } catch (FileNotFoundException e) {
37       e.printStackTrace();
38     } catch (IOException e) {
39       e.printStackTrace();
40     }
41   }
42
43   public SentenceSet(File file) throws FileNotFoundException {
44     this(new FileInputStream(file));
45   }
46   //next sentence
47   public Sentence next() {
48     Sentence q = get(n);
49     n++;
50     if (n >= size()) {
51       n = 0;
52     }
53     return q;
54   }
55
56 }
```

InteruptionSentenceSet.java:

```
1
2  package iristk.app.interaction;
3  import iristk.util.RandomList;
4  import java.io.*;
5  import java.util.*;
6
7  public class InterruptionSentenceSet extends ArrayList<InterruptionSentence> {
8
9    private int n = 0;
10
11   //Read from text file
12   public InterruptionSentenceSet(InputStream interruptionsentenceFile) {
13     try {
14       BufferedReader br = new BufferedReader(new InputStreamReader(
       interruptionsentenceFile));
15       String line = br.readLine();
16       int qn = 0;
17       int ln = 0;
18       System.out.println("Reading sentences");
19       while ((line = br.readLine()) != null) {
20         ln++;
21         if (!line.matches("[A-Za-z0-9,\\.;\\?'\\- ]*")) {
22           System.err.println("Illegal line " + ln + ": " + line);
23           continue;
24         }
25         String[] cols = line.split(";");
26         if (cols.length < 3) {
27           System.err.println("Not enough columns in line " + ln + ": " + line);
28           continue;
29         }
30         InterruptionSentence q = new InterruptionSentence("q" + qn++, cols); // creates:
       sentence (line number, the written sentense )
31         add(q);                    // adds one sentence to the total file that has been
       read in
32       }
33       System.out.println(qn + " sentences read");
34       randimize();
35     } catch (FileNotFoundException e) {
36       e.printStackTrace();
37     } catch (IOException e) {
38       e.printStackTrace();
39     }
40   }
41
42   public InterruptionSentenceSet(File file) throws FileNotFoundException {
43     this(new FileInputStream(file));
44   }
45
46   public void randimize() {
47     RandomList.shuffle(this);
48   }
49   //next sentence
50   public InterruptionSentence next() {
51     InterruptionSentence q = get(n);
52     n++;
53     if (n >= size()) {
54       n = 0;
55     }
56     return q;
57   }
58 }
```

InteractionFlow.xml:

```
<?xml version="1.0" encoding="utf-8"?>
<flow name="InteractionFlow" package="iristk.app.interaction "
  initial="Idle" xmlns:this="iristk.app.interaction .InteractionFlow" xmlns="iristk.flow"
  xmlns:p="iristk.flow.param" xmlns:agent1="iristk.situated.SystemAgentFlow" xmlns:agent2
    ="iristk.situated.SystemAgentFlow" xmlns:xsi="http://www.w3.org/2001/XMLSchema-
    instance"
  xsi:schemaLocation="iristk.flow flow.xsd iristk.situated.SystemAgentFlow
    SystemAgentFlow.xsd">

  <param name="sentences" type="SentenceSet"/>
  <param name="interruptionsentences" type="InterruptionSentenceSet"/>
  <param name="agent1" type="iristk.situated.SystemAgentFlow"/>
  <param name="agent2" type="iristk.situated.SystemAgentFlow"/>

  <var name="system1" type="iristk.situated.SystemAgent" value="agent1.getSystemAgent()"/
    >
  <var name="system2" type="iristk.situated.SystemAgent" value="agent2.getSystemAgent()"/
    >
  <var name="sentence" type="Sentence"/>
  <var name="interruptionsentence" type="InterruptionSentence"/>
  <var name="number" type="Integer" />

<state id="Idle">
  <onentry>
        <send event="action.face.texture" p:name="'tove'" p:agent="'agent1'"/>
          <send event="action.face.texture" p:name="'sandor'" p:agent="'agent2'"/>
        <agent1:attend location="system2.head.location"/>
        <agent2:attend location="system1.head.location"/>
        <goto state="Speaking"/>
  </onentry>
</state>

<state id="Speaking" extends="EnterExit">

   <onentry>
         <goto state="NextSentence" />
   </onentry>
</state>

<state id="NextSentence" extends="EnterExit">
  <onentry>
        <exec>sentence = sentences.next(); number = 0</exec>

          <if cond="sentence.getGesture() == 1000"/>
            <goto state="Waiting" />

          <else/>
              <if cond="sentence.getAgentNumber() == 1">



            <if cond="sentence.getGesture() == 2"/>
            <agent1:say > <spurt audio="g0001_003">COUGH</spurt> </agent1:say >
            <agent1:say text="sentence.getFullSentence()"/>

            <elseif cond="sentence.getGesture() == 3"/>
            <agent1:say > <spurt audio="g0001_006">CLEAR THROAT</spurt> </agent1:say >
            <agent1:say text="sentence.getFullSentence()"/>

            <elseif cond="sentence.getGesture() == 4"/>
            <agent1:say > <spurt audio="g0001_013">hm</spurt> </agent1:say >
```

```
57              <agent1:say text="sentence.getFullSentence()"/>
58
59              <elseif cond="sentence.getGesture() == 5"/>
60              <agent1:say > <spurt audio="g0001_014">hm</spurt> </agent1:say >
61              <agent1:say text="sentence.getFullSentence()"/>
62
63              <elseif cond="sentence.getGesture() == 6"/>
64              <agent1:say > <spurt audio="g0001_015">ehm</spurt> </agent1:say >
65              <agent1:say text="sentence.getFullSentence()"/>
66
67              <elseif cond="sentence.getGesture() == 7"/>
68              <agent1:say > <spurt audio="g0001_020">GIGGLE</spurt> </agent1:say >
69              <agent1:say text="sentence.getFullSentence()"/>
70
71              <elseif cond="sentence.getGesture() == 8"/>
72              <agent1:say > <spurt audio="g0001_025">ah</spurt> </agent1:say >
73              <agent1:say text="sentence.getFullSentence()"/>
74
75              <elseif cond="sentence.getGesture() == 9"/>
76              <agent1:say > <spurt audio="g0001_027">yeah</spurt> </agent1:say >
77              <agent1:say text="sentence.getFullSentence()"/>
78
79              <elseif cond="sentence.getGesture() == 10"/>
80              <agent1:say > <spurt audio="g0001_029">yeah</spurt> </agent1:say >
81              <agent1:say text="sentence.getFullSentence()"/>
82
83              <elseif cond="sentence.getGesture() == 11"/>
84              <agent1:say > <spurt audio="g0001_036">yay</spurt> </agent1:say >
85              <agent1:say text="sentence.getFullSentence()"/>
86
87              <elseif cond="sentence.getGesture() == 12"/>
88              <agent1:say > <spurt audio="g0001_037">oh</spurt> </agent1:say >
89              <agent1:say text="sentence.getFullSentence()"/>
90
91              <elseif cond="sentence.getGesture() == 13"/>
92              <agent1:say > <spurt audio="g0001_038">oh</spurt> </agent1:say >
93              <agent1:say text="sentence.getFullSentence()"/>
94
95              <elseif cond="sentence.getGesture() == 14"/>
96              <agent1:say > <spurt audio="g0001_050">ha ha</spurt></agent1:say >
97              <agent1:say text="sentence.getFullSentence()"/>
98
99              <elseif cond="sentence.getGesture() == 15"/>
100             <agent1:say > <spurt audio="g0001_052">GASP</spurt> </agent1:say >
101             <agent1:say text="sentence.getFullSentence()"/>
102
103             <elseif cond="sentence.getGesture() == 16">
104             <agent1:say > <spurt audio="g0001_002">tut tut</spurt> </agent1:say >
105             <agent1:say text="sentence.getFullSentence()"/>
106
107               <else/>
108               <agent1:say text="sentence.getFullSentence()"/>
109               </if>
110
111           <else/>
112           <agent2:say text="sentence.getFullSentence()"/>
113
114             </if>
115       </if>
116             <goto state="Speaking"></goto>
117     </onentry>
118 </state>
```

```
119
120 <state id="EnterExit">
121     <!--  <onevent name="sense.user.enter" cond="eq(event:agent, system1.id)">
122       <send event="action.speech.stop" p:agent="'agent1'"/>
123       <send event="action.speech.stop" p:agent="'agent2'"/>
124     </onevent>  -->
125   <onevent name="sense.user.enter" cond=" eq(event:agent, system1.id)">
126     <send event="action.speech.stop" p:agent="'agent1'"/>
127     <send event="action.speech.stop" p:agent="'agent2'"/>
128     <agent1:attend target="event:user"/>
129     <!--  <agent2:attend target="event:user"/>  -->
130     <wait msec="10"/>
131     <goto state="Greeting" />
132     </onevent>
133  </state>
134
135 <state id="Greeting">
136   <onentry>
137     <exec> interruptionsentence = interruptionsentences.next(); number = 0</exec>
138     <wait msec="2000"/>
139       <if cond="interruptionsentence.getAgentNumber() == 1">
140                 <if cond="interruptionsentence.getGesture() == 16">
141             <agent1:say > <spurt audio="g0001_002">tut tut</spurt> </agent1:say >
142
143             <elseif cond="interruptionsentence.getGesture() == 2"/>
144             <agent1:say > <spurt audio="g0001_003">COUGH</spurt> </agent1:say >
145
146             <elseif cond="interruptionsentence.getGesture() == 3"/>
147             <agent1:say > <spurt audio="g0001_006">CLEAR THROAT</spurt> </agent1:say >
148
149             <elseif cond="interruptionsentence.getGesture() == 4"/>
150             <agent1:say > <spurt audio="g0001_038">oh</spurt> </agent1:say >
151
152             <elseif cond="interruptionsentence.getGesture() == 5"/>
153             <agent1:say > <spurt audio="g0001_014">hm</spurt> </agent1:say >
154
155             <elseif cond="interruptionsentence.getGesture() == 6"/>
156             <agent1:say > <spurt audio="g0001_015">ehm</spurt> </agent1:say >
157             </if>
158
159               <agent1:say text="interruptionsentence.getFullInterruptionSentence()"/>
160               <wait msec="1500"/>
161           <else/>
162           <agent2:say text="interruptionsentence.getFullInterruptionSentence()"/>
163            </if>
164            <wait msec="1500"/>
165       <goto state="Repeat"/>
166     </onentry>
167
168 </state>
169
170 <state id="Repeat" extends="EnterExit">
171   <onentry>
172     <agent1:attend location="system2.head.location"/>
173     <agent2:attend location="system1.head.location"/>
174     <if cond="sentence.getAgentNumber() == 1">
175     <agent1:say gesture="'express_anger'"> <spurt audio="g0001_013">hm</spurt> what was i
       saying </agent1:say >
176     <wait msec="200"/>
177     <agent1:say text="sentence.getFullSentence()"/>
178     <goto state="Speaking"/>
179       <else/>
```

```
180       <agent1:say gesture="'express_anger'"> <spurt audio="g0001_038">oh</spurt> what
      were you saying? </agent1:say >
181       <wait msec="200"/>
182       <agent2:say text="sentence.getFullSentence()"/>
183       <goto state="Speaking"/>
184       </if>
185   </onentry>
186 </state>
187
188 <state id="Waiting" >
189   <onentry>
190     <agent1:attend location="system2.head.location"/>
191     <agent2:attend location="system1.head.location"/>
192     <agent1:say > , </agent1:say >
193     <wait msec="15000"/>
194     <goto state="Speaking"/>
195     </onentry>
196 </state>
197
198 </flow>
```

B.2 Java Application on the Slave Computer: robot2

Robot2System.java:

```
/*******************************************************************************
* All rights reserved. This program and the accompanying materials
 * are made available under the terms of the GNU Public License v3.0
 * which accompanies this distribution, and is available at
 * http://www.gnu.org/licenses/gpl.html
 *
 * Contributors:
 *      Gabriel Skantze - initial API and implementation
 *      Elif Toy- implementation of the java application robot2
 ******************************************************************************/
package iristk.app.robot2;

import iristk.situated.SituatedDialogSystem;
import iristk.situated.Situation;
import iristk.situated.SituationPanel;
import iristk.situated.SystemAgent;
import iristk.situated.SystemAgentFlow;
import iristk.speech.SpeechGrammarContext;
import iristk.speech.Synthesizer;
import iristk.speech.SynthesizerModule;
import iristk.speech.Voice.Gender;
import iristk.speech.windows.WindowsRecognizerFactory;
import iristk.speech.windows.WindowsSynthesizer;
import iristk.system.IrisGUI;
import iristk.util.Language;
import iristk.app.robot2.Robot2Flow;
import iristk.cfg.SRGSGrammar;
import iristk.flow.FlowModule;
import iristk.furhat.FurhatModule;
import iristk.kinect.KinectModule;
import iristk.speech.cereproc.*;

//Slave computer
//Definition of the System Class
public class Robot2System {

  private WindowsSynthesizer synth;

  public Robot2System() throws Exception {
    SituatedDialogSystem system = new SituatedDialogSystem(this.getClass());

    //Network Setup
    system.connectToBroker("myticket", "130.237.57.183", 1932);

    //Language
    system.setLanguage(Language.ENGLISH_US);

    //system.setupLogging(new File("c:/iristk_logging"), true);

    //AgentSetup
    synth = new WindowsSynthesizer();
    system.setupFace("bertil", synth, "agent2");
    synth.setVoice("lauren");

  public static void main(String[] args) throws Exception {
    new Robot2System();
  }
}
```

B.3 Arm Software

S-Function, serialCommunication, Outputs

```
if (xD[0] == 1) {
    if (reference[0] == 49) {
        mode[0] = 1;
    } else if (reference[0] == 48) {
        mode[0] = 0;
    } else if (reference[0] == 50) {
        mode[0] = 2;
    } else if (reference[0] >= 51 && reference[0] <= 100) {
        mode[0] = 10;
        elbow[0] = reference[0] -51;
    } else if (reference[0] >= 101 && reference[0] <= 150) {
        mode[0] = 10;
        shoulder[0] = reference[0] - 101;
    } else if (reference[0] >= 151 && reference[0] <= 200) {
        mode[0] = 10;
        hand[0] = reference[0] - 151;
    }
}
```

S-Function, serialCommunication, Discrete update

```
if (xD[0] != 1) {
    xD[0] = 1;
}
```

Embedded MATLAB-Function, Shoulder trajectory system

```
function [acc, vel, pos] = timer(time,mode)

Ts = 0.00068;
degOutToRadIn = ((2*pi)/360)*81;
amax = 500;                          %Max acc without saturation
%vmax = 500;                         %Max vel at zero acc
persistent t0 t1 t2 t3 t4 tstart goalPos vel_old pos_old oldGoalPos diffPos;

if mode >= 1                         %If given signal to initiate random movement
if isempty(t4)                       %Initialization of needed variables and parameters

    vel_old = 0;
    pos_old = 0;
    oldGoalPos = 0;
    diffPos = 0;
    t0 = 0;
    t1 = 0;
    t2 = 0;
    t3 = 0;
    t4 = 0;
    tstart = 1.3;
    goalPos = floor(rand*50)+1;
    goalPos = goalPos * degOutToRadIn;
    tacc  = sqrt(abs(goalPos/amax));
    tconst = 0;
    tbreak = tacc;
    tstoptime = 0;
    t0 = t0 + tstart;
    t1 = t0 + tacc;              %acceleration time
    t2 = t1 + tconst;            %constant velocity time
    t3 = t2 + tbreak;            %retardation time
    t4 = t3 + tstoptime;         %stop time
end

if time >= t4 && t4 ~= 0         %Used to generate a new trajectory from a random number
    oldGoalPos = goalPos;


    if mode == 0                                    %Normal mode
        goalPos = 5*degOutToRadIn;
        diffPos = goalPos - oldGoalPos;
    elseif mode == 2                                %Interrupt routine (silence motion)
        goalPos = 45*degOutToRadIn;
        diffPos = goalPos - oldGoalPos;
    else
        goalPos = floor(rand*45)+1;                 %Randomly generated reference
        goalPos = goalPos * degOutToRadIn;          %%deg
        while abs(goalPos-oldGoalPos) <= 8
            goalPos = floor(rand*45)+1;
            goalPos = goalPos * degOutToRadIn;
        end
        diffPos = goalPos - oldGoalPos;
    end

    tstart = 1.3;
    tacc  = sqrt(abs(diffPos/amax));
    tconst = 0;
    tbreak = tacc;
    tstoptime = 0;
    t0 = t0 + tstart;
    t1 = t0 + tacc;                                 %acceleration time
```

```
62      t2 = t1 + tconst;                           %constant velocity time
63      t3 = t2 + tbreak;                           %retard time
64      t4 = t3 + tstoptime;                        %stop time
65
66  end
67
68
69
70  if time <= t0                   %Resting state
71      acc = 0;
72
73  elseif time <= t1               %Acceleration state
74      if diffPos >= 0
75          acc = amax;
76      else
77          acc = -amax;
78      end
79
80  elseif time <= t2               %Constant velocity state
81      acc = 0;
82
83  elseif time <= t3               %Retardation state
84      if diffPos >= 0
85          acc = -amax;
86      else
87          acc = amax;
88      end
89
90  elseif time <= t4               %Resting state
91      vel = 0;
92      vel_old = 0;
93      acc = 0;
94  else
95      acc = 0;
96  end
97
98
99
100 vel = vel_old + acc*Ts;         %Update velocity ticker
101 if acc == 0
102     vel = 0;
103 end
104 pos = pos_old + vel*Ts;         %Update Position ticker
105
106 if pos >= 45*degOutToRadIn      %Safety routine
107     pos = 45*degOutToRadIn;
108 elseif pos*degOutToRadIn <= 2
109     pos = 5*degOutToRadIn;
110 end
111
112 vel_old = vel;                  %%Save velocity ticker
113 pos_old = pos;                  %Save Position ticker
114
115 else
116     acc = 0;                        %Returns to zero with step
117     vel = 0;
118     pos = 5*degOutToRadIn;
119
120 end                                 %end of switch statement
```

Embedded MATLAB-Function, Elbow and Hand trajectory system

```
function [acc, vel, pos, pos_hand] = timer(time,mode,maxPosElbow)

Ts = 0.00068;
degOutToRadIn = ((2*pi)/360)*128.3; %Transformation
amax = 1000;                        %Max acceleration without saturation
tstart = 0.5;
t_hand_start = 2;
vmax = 500;                         %Max vel at zero acc
persistent t0 t0_hand t1 t2 t3 t4 goalPos vel_old pos_old pos_old_hand oldGoalPos diffPos
    ;
if mode >= 1                        %If given signal to initiate random movement
if isempty(t4)                      %Initialization of needed variables and parameters

    vel_old = 0;
    pos_old = 0;
    oldGoalPos = 0;
    diffPos = 0;
    pos_old_hand = 0;
    t0 = 0;
    t0_hand = 0;
    goalPos = floor(rand*maxPosElbow)+1;
    goalPos = goalPos * degOutToRadIn;
    tacc  = sqrt(abs(goalPos/amax));
    tconst = 0;
    tbreak = tacc;
    tstoptime = 0;
    t0 = t0 + tstart;
    t1 = t0 + tacc;             %acceleration time
    t2 = t1 + tconst;           %constant velocity time
    t3 = t2 + tbreak;           %retardation time
    t4 = t3 + tstoptime;        %stop time
end

if time >= t4 && t4 ~= 0        %Used to generate a new trajectory from a random number
    r = rand();                 %Randomly generated reference
    r = rand();
    oldGoalPos = goalPos;

    if mode == 0                                %Not speak
        goalPos = 0;
        diffPos = goalPos - oldGoalPos;
    elseif mode == 2                            %Interrupt routine
        goalPos = (maxPosElbow-5)*degOutToRadIn;
        goalPos = goalPos * degOutToRadIn;
        diffPos = goalPos - oldGoalPos;
    else
        goalPos = floor(r*maxPosElbow);         %Normal mode
        goalPos = goalPos * degOutToRadIn;
        diffPos = goalPos - oldGoalPos;

    end

    tacc  = sqrt(abs(diffPos/amax));
    tconst = 0;
    tbreak = tacc;
    tstoptime = 0;
    t0 = t0 + tstart;
    t1 = t0 + tacc;             %acceleration time
    t2 = t1 + tconst;           %constant velocity time
    t3 = t2 + tbreak;           %retardation time
    t4 = t3 + tstoptime;        %stop time
```

```
61
62 end
63
64 pos_hand = pos_old_hand;            %set position to old before time check
65
66 if time >= t0_hand + t_hand_start  %Set position for hand at hand start time
67     if mode == 2
68         pos_hand = 90;
69     else
70         pos_hand = floor(rand*180)+1;
71     end
72     t0_hand = t0_hand + t_hand_start;
73 end
74
75 if time <= t0                %Resting state
76     acc = 0;
77
78 elseif time <= t1            %Acceleration state
79     if diffPos >= 0
80         acc = amax;
81     else
82         acc = -amax;
83     end
84
85 elseif time <= t2            %Constant velocity state
86     acc = 0;
87
88 elseif time <= t3            %Retardation state
89     if diffPos >= 0
90         acc = -amax;
91     else
92         acc = amax;
93     end
94
95 elseif time <= t4            %Resting state
96     vel = 0;
97     vel_old = 0;
98     acc = 0;
99 else
100     acc = 0;
101 end
102
103 vel = vel_old + acc*Ts;                    %Update velocity ticker
104 if acc == 0
105     vel = 0;
106 end
107 pos = pos_old + vel*Ts;                    %Update position ticker
108
109 if pos >= maxPosElbow*degOutToRadIn        %Safety routine
110     pos = (maxPosElbow-5)*degOutToRadIn;
111 elseif pos <= 0
112     pos = 0;
113 end
114
115 vel_old = vel;                             %Save velocity ticker
116 pos_old = pos;                             %Save position ticker
117 pos_old_hand = pos_hand;                   %Save position hand ticker
118
119 elseif mode == 2                           %Interrupt routine
120     pos = (maxPosElbow-5)*degOutToRadIn;
121     acc = 0;
122     vel = 0;
```

```
123     pos_hand = 0;
124 else
125     acc = 0;                        %Returns to zero with step
126     vel = 0;
127     pos = 0;
128     pos_hand = 0;
129
130 end                                 %end of switch statement
```

S-Function,Luke Potentiometer Left , Output

```
if(xD[0]==1)
{
    #ifndef MATLAB_MEX_FILE
    int val;
    val = analogRead(POTPIN);
    val = val/2.84; // to degrees
    potPosition[0] = val;

    static int state = 1;
    static double lowPosition = 0;
    int micro1 = digitalRead(microSwitch3);
    int micro2 = digitalRead(microSwitch4);

    switch(state) {
        case 1:
            init[0] = 0;
            if (micro1 != 1) {
                digitalWrite(LEFT_SHOULDER_MOTOR_DIR,LOW);
                analogWrite(LEFT_SHOULDER_MOTOR_PWM, 20);
            } else {
                analogWrite(LEFT_SHOULDER_MOTOR_PWM, 0);
                digitalWrite(LEFT_SHOULDER_MOTOR_DIR,HIGH);
                lowPosition = val + 8;
                lowPos[0] = lowPosition;
                state = 2;
            }
            break;
        case 2:
            if (val < lowPosition) {
                analogWrite(LEFT_SHOULDER_MOTOR_PWM, 40);
            } else {
                analogWrite(LEFT_SHOULDER_MOTOR_PWM, 0);
                state = 3;
            }
            break;
        case 3:
            init[0] = 1;
    }


    #endif
}
```

S-Function, Luke Potentiometer Left , Discrete update

```
if(xD[0]!=1)
{
  #ifndef MATLAB_MEX_FILE
    pinMode(microSwitch3, INPUT_PULLUP);
    pinMode(microSwitch4, INPUT_PULLUP);
   xD[0]=1;
   #endif
}
```

S-Function, Luke PWM Driver Left Shoulder Motor, Output

```
#ifndef MATLAB_MEX_FILE

if(xD[0] == 1)
{
    int pwm = 0;

    static int state = 1;
    int micro1 = digitalRead(microSwitch3);
    int micro2 = digitalRead(microSwitch4);

    switch(state) {
        case 1:
            if (init1[0] == 1 && init2[0] == 1) {
                state = 2;
            }
            break;
        case 2:
            if (micro1 == 1 || micro2 == 1) {
                state = 3;
                break;
            }
            if(pwmOutput[0] <= 0){
                digitalWrite(LEFT_SHOULDER_MOTOR_DIR,LOW);
                if(mode[0] == 0){
                    pwm = abs(pwmOutput[0])/3;
                }
                else{
                    pwm = abs(pwmOutput[0]);
                }
            }
            else if(pwmOutput[0] > 0){
                digitalWrite(LEFT_SHOULDER_MOTOR_DIR,HIGH);
                if(mode[0] == 0){
                    pwm = pwmOutput[0]/3;
                }
                else{
                    pwm = pwmOutput[0];
                }
            }
            if (potPosition[0] <= 50 || potPosition[0] >= 160) {
                analogWrite(12,0);
            }
            analogWrite(12, pwm);
            break;
        case 3:
            if (micro1 != 1 && micro2 != 1) {
                state = 2;
            } else {
                pwm = 0;
                analogWrite(12, pwm);
            }
            break;
    }
}
#endif
```

S-Function, Luke PWM Driver Left Shoulder Motor, Discrete update

```
#ifndef MATLAB_MEX_FILE

```

```
3  if(xD[0] != 1)
4  {
5    TCCR1A = 0b10100000;
6    TCCR1B = 0b00010001;
7    ICR1 = 400;
8
9    pinMode(LEFT_SHOULDER_MOTOR_DIR,OUTPUT);
10   pinMode(LEFT_SHOULDER_MOTOR_PWM,OUTPUT);
11   pinMode(microSwitch3,INPUT_PULLUP);
12   pinMode(microSwitch4,INPUT_PULLUP);
13
14   analogWrite(LEFT_SHOULDER_MOTOR_PWM, 0);
15   digitalWrite(LEFT_SHOULDER_MOTOR_DIR, LOW);
16
17     xD[0] = 1;
18 }
19 #endif
```

S-Function, Luke Elbow Encoder Left, Output

```
if(xD[0]==1)
{
    #ifndef MATLAB_MEX_FILE

    static int state = 1;
    static double highPosition = 0;
    int micro1 = digitalRead(microSwitch1);
    int micro2 = digitalRead(microSwitch2);

    switch(state) {
        case 1:
            maxPos[0] = 0;
            init[0] = 0;
            if (micro1 != 1) {
                digitalWrite(LEFT_ELBOW_MOTOR_DIR,HIGH);
                analogWrite(LEFT_ELBOW_MOTOR_PWM, 35);
            } else {
                analogWrite(LEFT_ELBOW_MOTOR_PWM, 0);
                digitalWrite(LEFT_ELBOW_MOTOR_DIR,LOW);
                left_position = 0;
                state = 2;
            }
            break;
        case 2:
            if (left_position > -70) {
                analogWrite(LEFT_ELBOW_MOTOR_PWM, 40);
            } else {
                analogWrite(LEFT_ELBOW_MOTOR_PWM, 0);
                left_position = 0;
                state = 3;
            }
            break;
        case 3:
            if (micro2 != 1) {
                analogWrite(LEFT_ELBOW_MOTOR_PWM, 45);
            } else {
                analogWrite(LEFT_ELBOW_MOTOR_PWM, 0);
                digitalWrite(LEFT_ELBOW_MOTOR_DIR, HIGH);
                highPosition = left_position + 50;
                maxPos[0] = -1*highPosition/((4*4*128.3)/360.0*1.875);
                state = 4;
            }
            break;
        case 4:
            if (left_position < highPosition) {
                analogWrite(LEFT_ELBOW_MOTOR_PWM, 20);
            } else {
                analogWrite(LEFT_ELBOW_MOTOR_PWM, 0);
                state = 5;
            }
            break;
        case 5:
            init[0] = 1;
            position[0] = -1*left_position;
            break;
    }

    #endif
}
```

S-Function, Luke elbow Encoder Left, Discrete update

```
if(xD[0]!=1)
{
  #ifndef MATLAB_MEX_FILE


  pinMode(2, INPUT);    // Enable input on encoder A
  pinMode(3, INPUT);    // Enable input on encoder B
  pinMode(microSwitch1, INPUT_PULLUP);
  pinMode(microSwitch2, INPUT_PULLUP);

  pinMode(LEFT_ELBOW_MOTOR_DIR,OUTPUT);
  pinMode(LEFT_ELBOW_MOTOR_PWM,OUTPUT);

  analogWrite(LEFT_ELBOW_MOTOR_PWM, 0);
  digitalWrite(LEFT_ELBOW_MOTOR_DIR, LOW);

  attachInterrupt(0, left_encoder_interrupt, CHANGE);  //Encoder interrupt for change,
    rising/falling edge, signal A
  attachInterrupt(1, left_encoder_interrupt, CHANGE);   //Same as above, signal B
  interrupts();

   xD[0]=1;
   #endif
}
```

S-Function, Luke PWM Driver Elbow Left, Output

```
#ifndef MATLAB_MEX_FILE

if(xD[0] == 1)
{
    int micro1 = digitalRead(microSwitch1);
    int micro2 = digitalRead(microSwitch2);
    static int state = 1;
    switch(state) {
        case 1:
            if (init1[0] == 1 && init2[0] == 1) {
                state = 2;
            }
            break;
        case 2:
            if (micro1 == 1 || micro2 == 1) {
                state = 3;
                break;
            }
            int pwm = 0;
            if(duty[0] <= 0){
               digitalWrite(LEFT_ELBOW_MOTOR_DIR,HIGH);
               pwm = abs(duty[0]);
            }
            else if(duty[0] > 0){
                digitalWrite(LEFT_ELBOW_MOTOR_DIR,LOW);
                pwm = duty[0];
            }
            analogWrite(LEFT_ELBOW_MOTOR_PWM, pwm);
            break;
        case 3:
            if (micro1 != 1 && micro2 != 1) {
                state = 2;
            } else {
                analogWrite(LEFT_ELBOW_MOTOR_PWM, 0);
            }
            break;
    }
}
#endif
```

S-Function, Luke PWM Driver Elbow Left, Discrete update

```
#ifndef MATLAB_MEX_FILE

if(xD[0] != 1)
{
     TCCR3A = 0b10100000;
     TCCR3B = 0b00010001;
     ICR3 = 400;

     pinMode(LEFT_ELBOW_MOTOR_DIR,OUTPUT);
     pinMode(LEFT_ELBOW_MOTOR_PWM,OUTPUT);
     pinMode(microSwitch1, INPUT_PULLUP);
     pinMode(microSwitch2, INPUT_PULLUP);

     analogWrite(LEFT_ELBOW_MOTOR_PWM, 0);
     digitalWrite(LEFT_ELBOW_MOTOR_DIR, HIGH);
    xD[0] = 1;
}
#endif
```

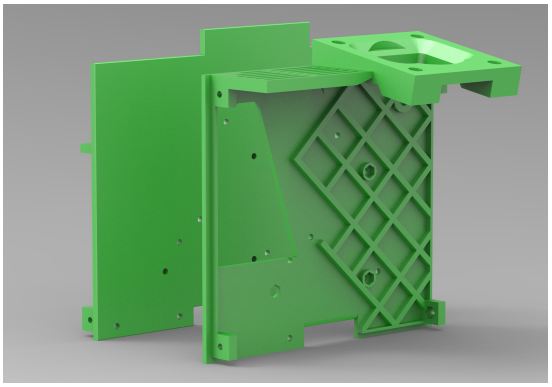
S-Function, Left Hand Driver, Output

```
#ifndef MATLAB_MEX_FILE
 if(xD[0] == 1)
{
     if(init[0] == 1){
         OCR4B = pos[0]/180.0*0.075*OCR4A+0.0375*OCR4A;
     }
}
#endif
```

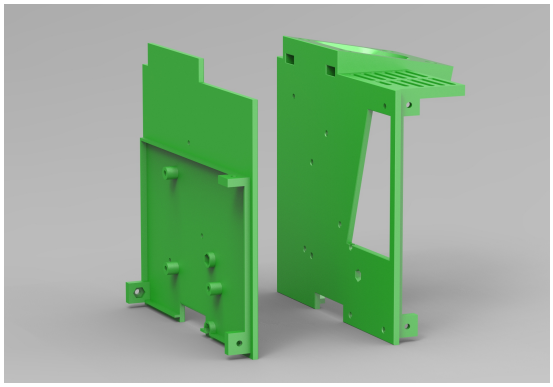
S-Function, Left Hand Driver, Discrete update

```

#ifndef MATLAB_MEX_FILE
 if(xD[0] != 1)
{
     //Servo myservo1;            // servo object
     pinMode(LEFT_HAND_MOTOR, OUTPUT);           //PWM out

     TCCR4A = _BV(WGM41) | _BV(WGM40) | _BV(COM4B1); //Fast PWM
   TCCR4B = _BV(WGM43) | _BV(WGM42) | _BV(CS42) | _BV(CS40);
   OCR4A = 311;  // 39 > Correct frequency setting of 50kHz -> need 50 Hz

     xD[0] = 1;
}
#endif
```

C Additional Figures

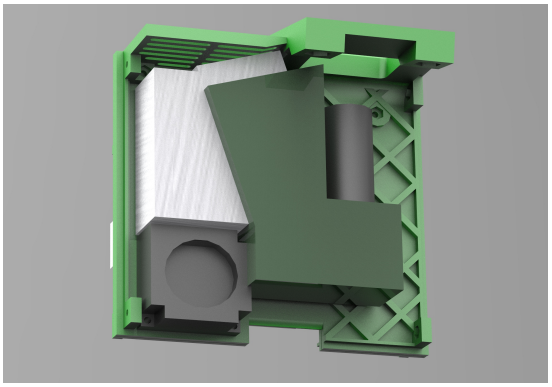


(a) Showing reinforcements

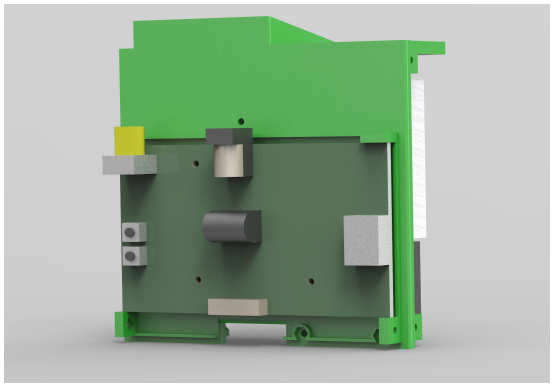


(b) Showing main board mounting and slots for mounting nuts

Figure 37: Projector plates

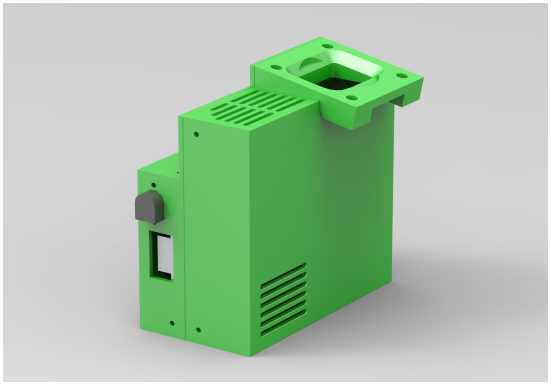


(a) Showing projector, heatsink and fan

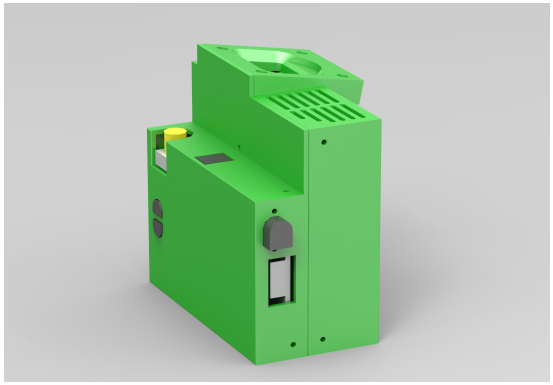


(b) Showing main board

Figure 38: Projector components



(a) Showing vents and mount



(b) Showing connections and buttons

Figure 39: Projector assembly with cover

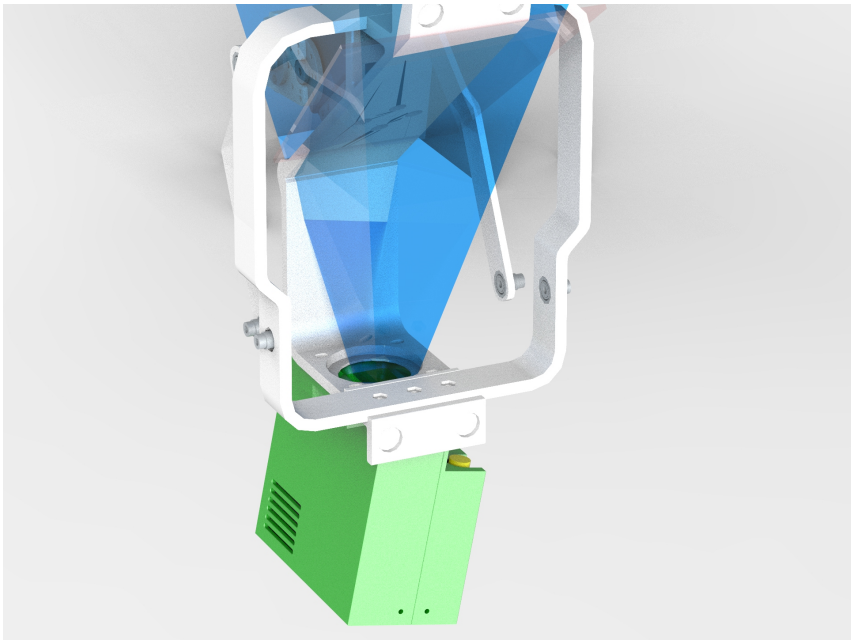


Figure 40: Projector assembly mounted to head showing projection without wide-angle lens