Implementation of an Interactive Autonomous Guard Robot with Humanoid Features

Project Report

Written by

Fredrik Lagerstedt

Zhanyu Tuo

Terje Stenström

Nipun C. Gammanage

OCTOBER 2019

Chalmers University of Technology - Gothenburg

University of Gothenburg - Gothenburg

Table of Contents

Chapter 1

Introduction

One thing that we have learned from the study of humanoids is that there is a vast number of possible applications for them. Security is one field where these humanoid robots have potential to be applied. One of the advantages of having a robot guard is a lack of emotional nuance that can be perceived as more authoritative than a human guard.

In this project we have developed a humanoid guard robot that interacts with people that approach it. Its sole purpose is to grant access to people with proper credentials and keep intruders off. More specifically, face detection is used to detect an approaching person. It then initiates a dialogue, asking for credentials. If the person is authorized, the robot will greet the them. Otherwise, the robot will ask the person to leave. If the person refuses to leave, the robot will become hostile, point a firearm at them and eventually shoot the intruder.

The purpose of this project has been to design and construct this robot to get a better understanding of robotics hardware and software such as Robot Operating System (ROS) and to apply basic algorithms from computer vision and mechanical dynamics.

The final version of the guard robot is shown in figure 1.1. A serious face is added on top of the body and a laser pen with a gun-shape model is used instead of a real weapon.

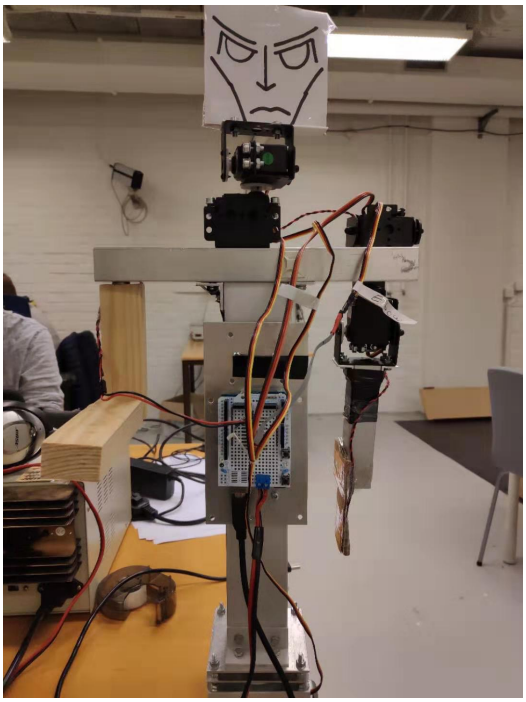


Figure 1.1: The final version of the guard robot

1.1 Goal statement

Before the end of this course, test a guard robot design, based on currently widely available technology, that can gate access to restricted areas. This work will explore the viability of such a system as well as deepen and broaden the knowledge of the team within key areas such as: face-detection, humanoid kinematics and decision making software.

1.2 Deliverables and boundaries

The project was divided into segments and deliverables according to Table 1.1. Given the time constraints to complete each of these deliverables we had to impose a few boundaries:

- Documentation
  No technical documentation will be written for this project. A written report will be compiled and an oral presentation will be given at the end of the course.
- Hardware
  The robot will be able to aim and rotate in place but it is always stationary. In addition; should a suitable toy gun not be found a substitute will be made so the robot can demonstrate this part of the project.
- Language Processing
  Only a limited set of phrases will be considered valid, otherwise the robot will default to a dismissive stance. The code will be based on available open source code.
- Testing
  Only one person will be present in the robots field of view at a time. The number of interaction scenarios tested will be limited to only a few.
- Face Detection
  Some open source codes will be utilized.
- Decision processing and FSM
  Limit number of possible actions the robot can take in regards to a scenario such that the number states does not increase uncontrollably.

Table 1.1: Project segments and deliverables.

Segment	**Deliverables**
Documentation	A planning report.
	A final report.
	A presentation.
Hardware	A Hubert robot modified for our purpose.
	A testing environment appropriate for face detection.
Computer vision	A face detection algorithm.
Kinematics	Code for controlling Hubert's arm.
Communication	A language processing algorithm.
Decision making	A finite state system for keeping dialogues and making decisions.

Chapter 2

Hardware and Servo control

The hubert robot platform was used as the base for the project as it can rotate both body and head and has an arm, on which the "gun" was attached. For control, the Arduino Uno microcontroller was used and a small add-on board was soldered up to split power to the servos and connect each of them up to a pin of the controller. A generic key-ring laser-pointer was disassembled and reverse engineered to act as a gun substitute, the Arduino can control that directly via the add-on board, turning the laser on and off by digital writes.

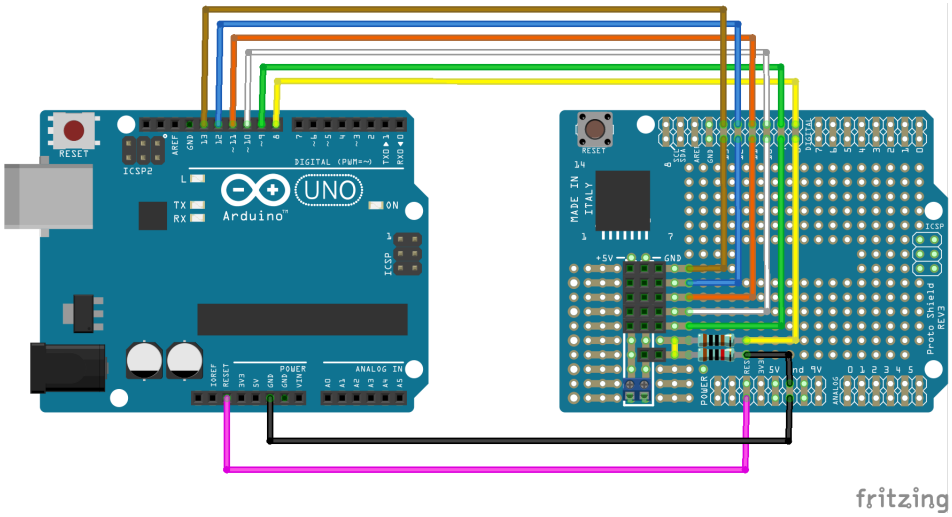


Figure 2.1: Overview of the connection between the Arduino and control board.

As can be seen in figure 2.1, each servo is given a header connected to the Arduino via the control board. The power for the servos comes from an external PSU, and is split to each one. There is also a connector for the laser, connected directly to a digital pin of the Arduino via a voltage divider to achieve the 4.5V used for the laser-pointer and limit current so as to not overload the output of Arduino (which can only supply 20mA on a single pin).

The Arduino is connected via USB to the main laptop and communicates with the other subsystems via a ROS serial node. This node accepts servo angles sent as integers via the ROS std_msgs package. The messages are written to the servos via the Arduino servo library, which also handles the PWM signal generation and position maintenance.

Chapter 3

Face Detection and Tracking

In this chapter the *face_detection* package in *Hubert's* work-space will be discussed. This package is of course build in collaboration with ROS, however the main function, face detection and tracking are developed as separate library, allowing future developers to build upon them. Furthermore for integration purpose with the real robot a coordinate conversion was made and was integrated with the *HUbert's* brain.

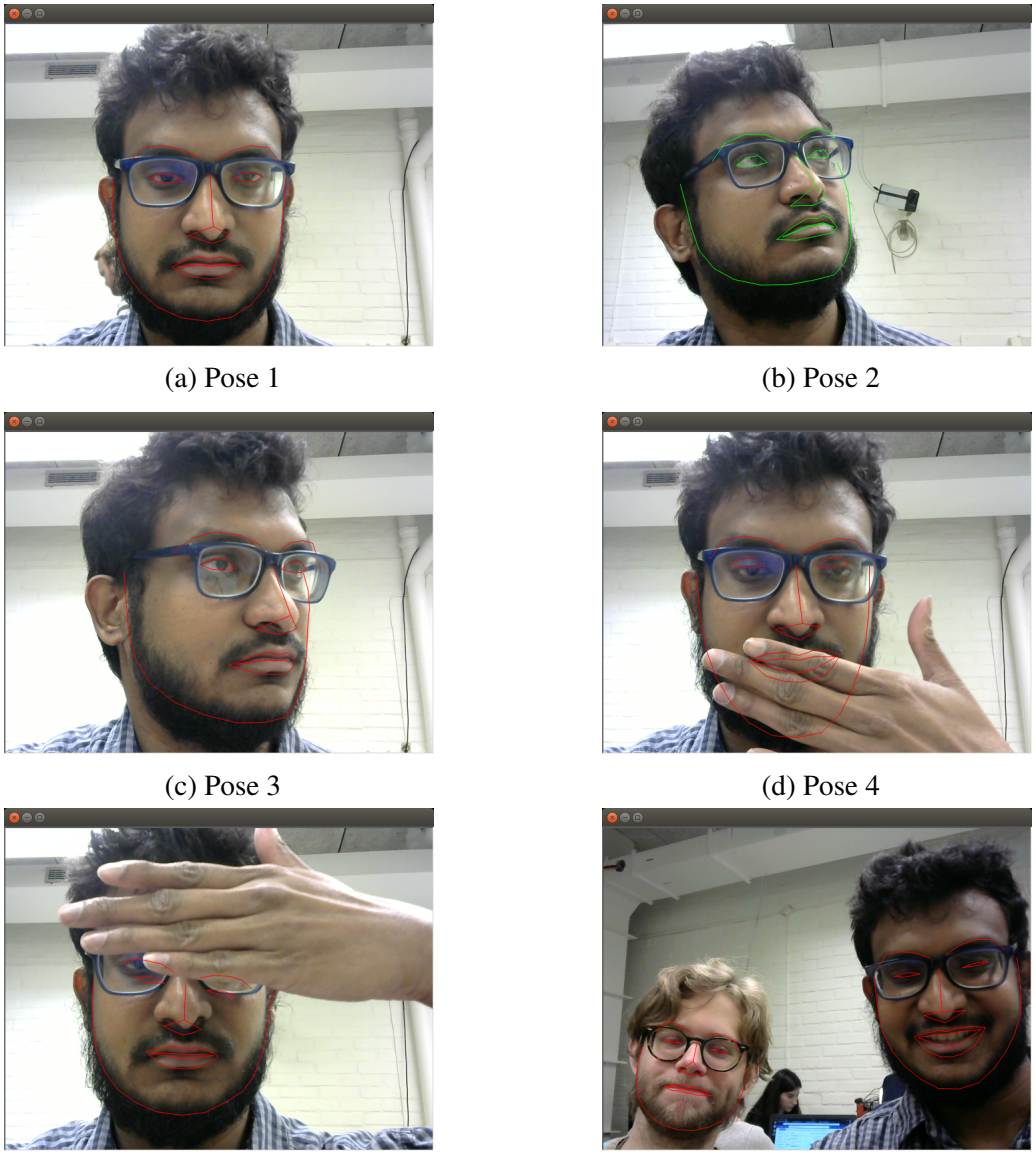
The entire process from face detection to moving the neck and base motors contains mainly three parts,

1. The Face detection and Tracking library
2. ROS node to handle smooth operation
3. Coordinate conversion method to convert the image points to a servo angle.

3.1 Face detection

This library is written so that it can do both the face detection and face tracking. For face detection the open source *C++* libraries *OpenCV* and *dlib-19.17* were used. Initially the *input image* from the camera was converted to an *OpenCV* type, *cv::Mat*. Then this matrix was feed into a face detector provided by the *dlib* pose model, which return 68 feature points will be outputted. All these faces will then be save into a vector define by dlib. The faces are define as rectangles, this rectangles are define under dlib library but can be access in the same ways as *cv::Rect*, the OpenCV rectangle. We then feed all these rectangles(one vector which includes all the rectangles) to the system to do the tracking.

With the above mentioned 68 points one could actually detect face quite accurately. In the following figure 3.1 it is illustrated the initial face detection results, without *ROS* implementation. This of course will analysis all the 68 points provided by the detector. It can be clearly seen that from the figures 3.1a, 3.1b and 3.1c that one could use different poses and the detector will still be able to detect ones face correctly even though the face has extra features like spectacles or a face(s) with a beards. In the figure 3.1d and 3.1e the face was half covered and still the detector was able to detect the face and predict the missing part pose as well. The last figure was inserted to illustrate that the detector can of course detect multiple faces.



(a) Pose 1

(b) Pose 2

(c) Pose 3

(d) Pose 4

(e) Pose 5

(f) Pose 6

Figure 3.1: Results with different poses of face

However this accuracy in detection does comes with a quite an expensive computational cost. Of course if one is planning to user a PC with better performances then this is a perfect solution. But for this project the PC chosen to handle the entire process (which includes other decision making) was a *core-i5* laptop with no *GPU* support, hence the output was extremely slow, running at just 10 Hz. The below figure 3.2 will show how much memory it took from RAM just to process an image of a size 640 by 480.

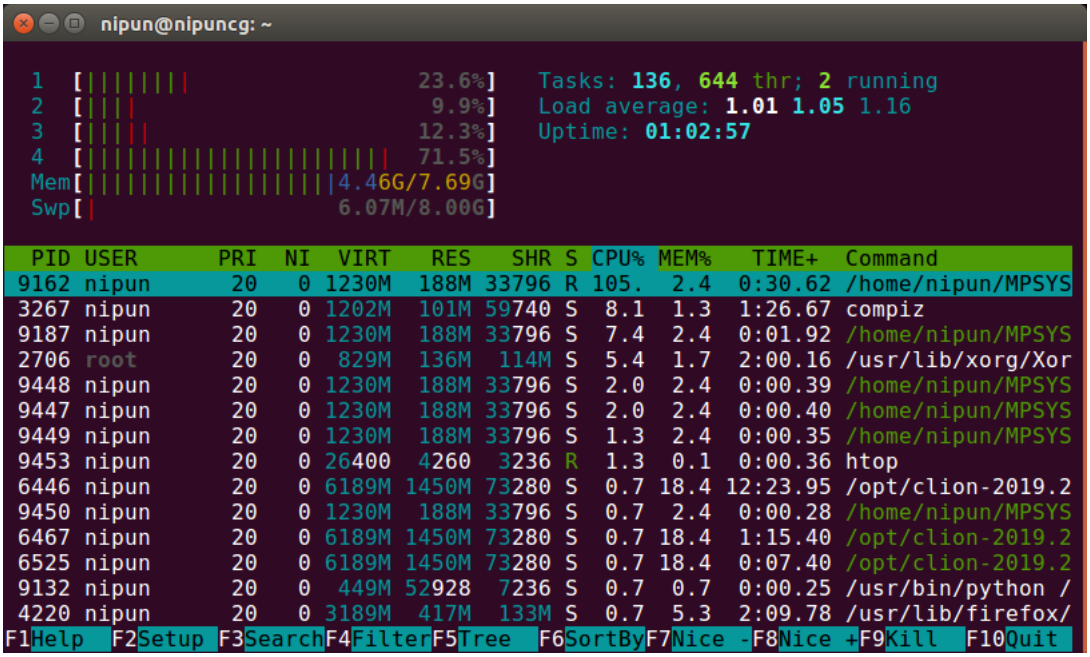


Figure 3.2: RAM usage

3.2 Face tracking

As mentioned in he previous subsection, the face detection was quite computationally expensive. Combining this with he tracking method and the other processes make it impossible to do other tasks parallel. Therefore it was decided to reduce the quality of the frame by reducing the size of the image, even though it was improved a little bit the overall results were still the same. The next solution suggested was to *skip* some frames. Again results improved but was not up to the expected speed. Lastly it was decided to reduce the number of points that describe the face. What *dlib* is providing is 68 points, then it also fit these points in to a rectangle. But for this fitting it uses all the 68 points and it was decided instead of using all these points, one could use only few points and fit it into a rectangle. Of course it will not give the same results as the initial rectangle but it was observed that for this particular instance it was enough.

The number of point which was use to get an accurate rectangle was then reduce from 68 to 16 which represents only the *jaw line* of the face. Surprisingly enough the results improved. It was also observed that one may still keep the screen ratio and by just reducing the processed number of points the whole process can speed up. This however will comes up with downsides, once the points are reduced we were not able to do pose detection like the once shown in 3.1d and 3.1e.

The idea was to get the the 16 jaw line points and using the *fitEllipse* method to get the best

fit ellipse of those points. Once these rectangles are obtained we then execute the tracking algorithm. For the purpose of this project we have used a method called *First landmark tracking*. The idea behind this is to only track the first person who comes to the frame.

The process can be explained in the following steps:

1. Once the faces are detected, all these rectangles will be compared with the *previously tracked rectangle*, which is named as the *ground truth*. To obtain a error value the method *Root mean square error* was used as explained in the equation below.

$$\text{RMSE} = \sqrt{\frac{\sum_{i=0}^{N} \left[\left(X_{i,\text{filtered}} - X_{i,\text{ground}} \right)^2 + \left(Y_{i,\text{filtered}} - Y_{i,\text{ground}} \right)^2 \right]}{\text{Number of points}}}$$

   In the equation one will get more than one filtered shape which includes the details of a face in the form of a *std::vector<cv::Point>*. It should be also noted that all the filtered shapes will contain the same number of points, if not it means the fitting function does not produce the values correctly, thus those vectors will be discarded. The distance between the all current rectangles and the ground truth rectangle will be compared and the rectangle with the minimum distance will be selected as the rectangle of interest.

2. In the initial case, where we don't have the previous face, we choose the one that is closest to the camera. This task is achieved by calculating the area of each rectangle. The rectangle with the largest area will be taken as the closest face.

3. This face which entered at the initial stage will be in the tracking loop, regardless of the area change, unless that face, go out of the frame or, go too far from the frame, then we will start tracking the largest face in the current frame.

Following figure 3.4 describe the *first landmark tracking method* in a visual manner. In the initial case as explained it will get the closest face. This will be decided by the area cover by the *eclipse* which represents the face. The more close to the camera bigger the eclipse will be. According to this method the algorithm will then decide which face is closer to the camera. Then if that person disappear (assuming he/she walks away), algorithm will then track the next face which has the largest area. This is shown in the figure 3.4b. Then in the figure 3.4c, it can be seen that a third person enters the frame but is closer to the camera than the already tracking one. However the algorithm will still track that same person and will ignore the new person. Please note that in the figure 3.4c the face is red due to the fact that it is in the corner of the frame. This detail was use to decide whether the base should move or not. The color red means the person is move away from the frame and therefore the robot needs to rotate the camera to left or right. The entire process of the tracking algorithm is explained in the next figure 3.3.

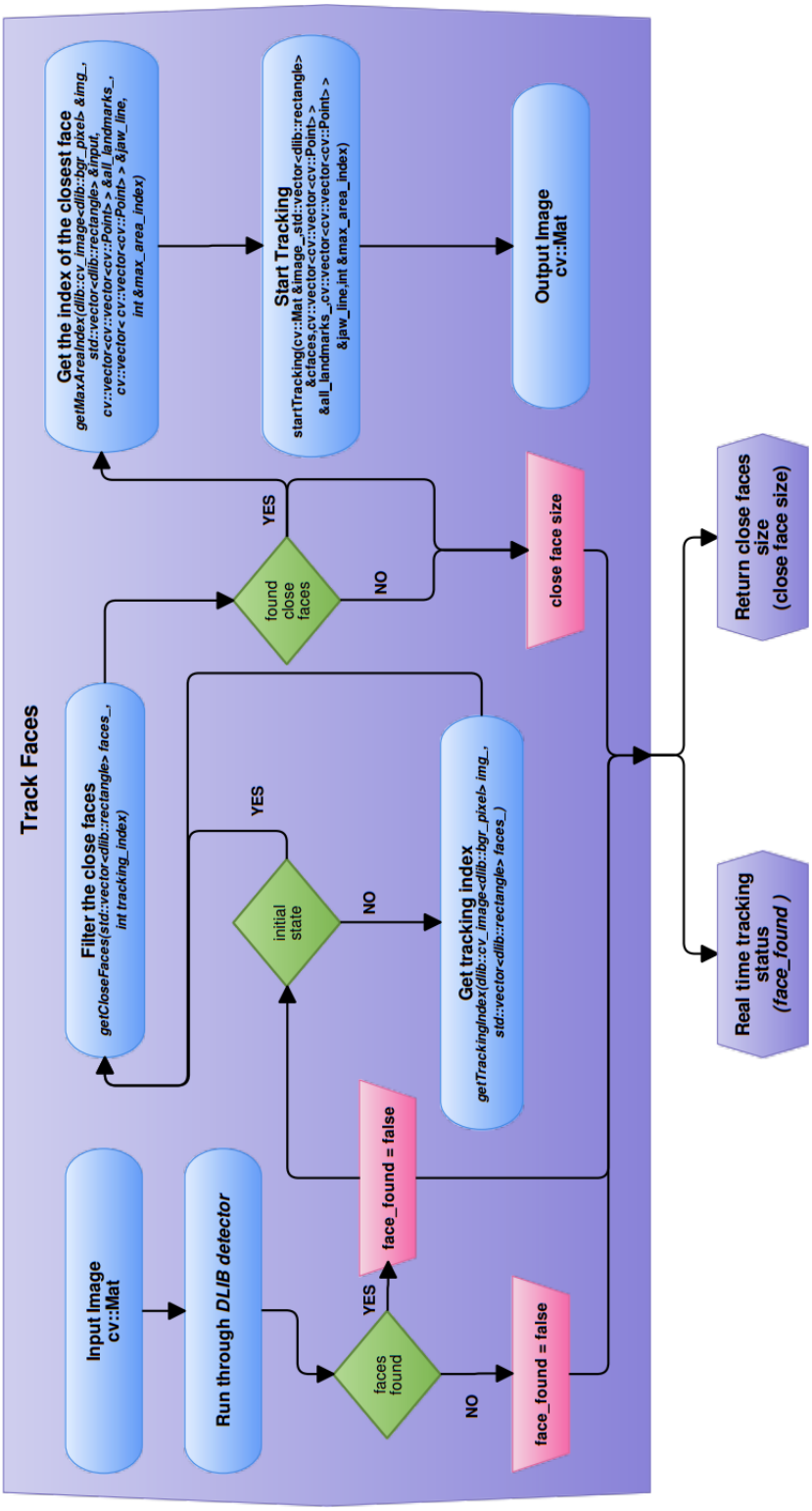
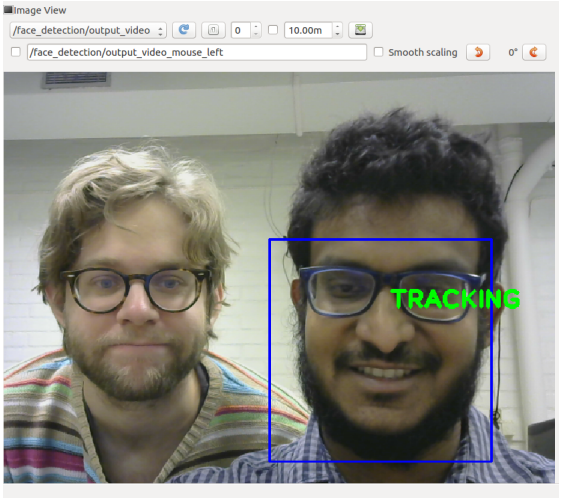
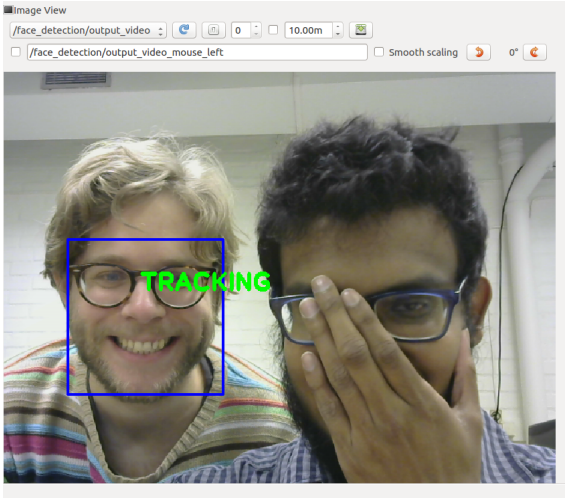


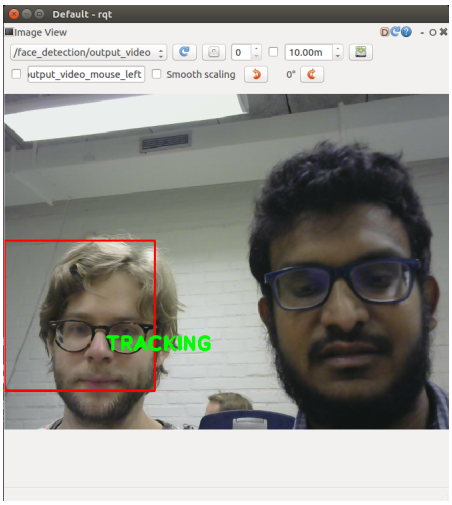
Figure 3.3: Flow chart of face tracking function



(a) Initial scenario



(b) Scenario 2



(c) Scenario 3

Figure 3.4: First landmark tracking method

3.3 Integration with ROS

The above mentioned two processes were then integrated with ROS to facilitate communication with the other parts of the robot smoothly. In the below figure 3.5 explains the main process in the *ROS Face Detection* node.

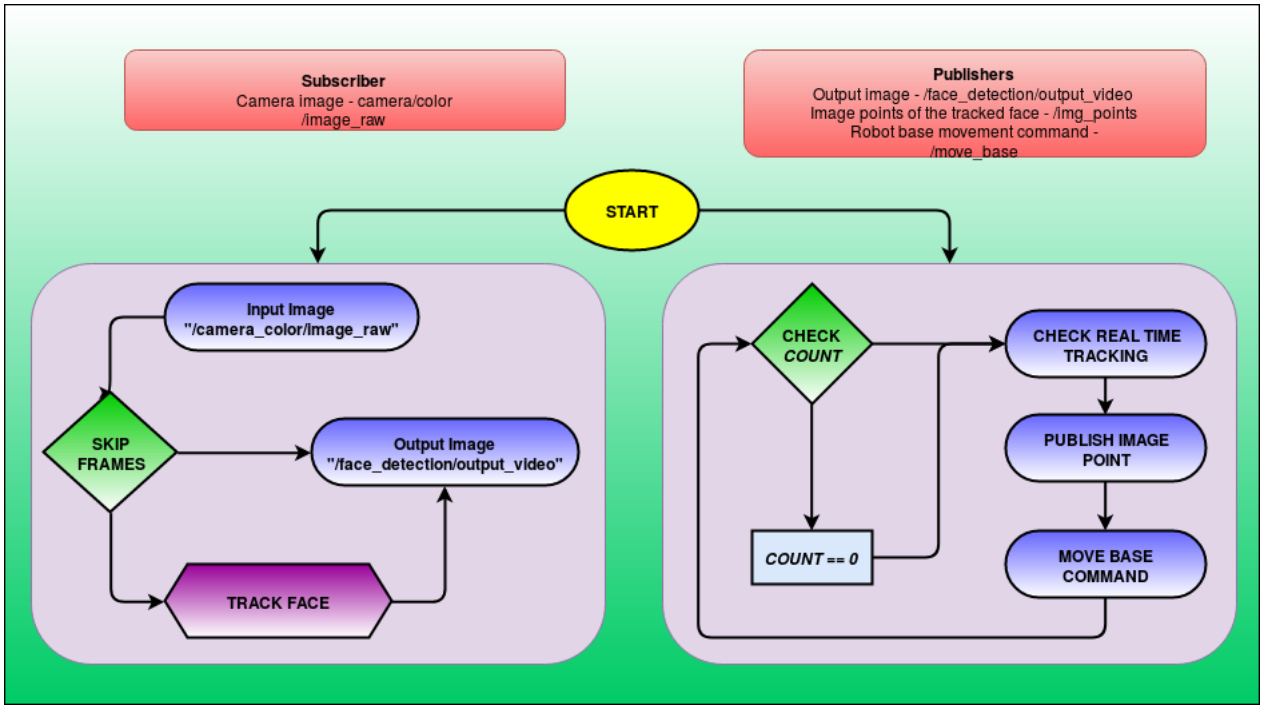


Figure 3.5: ROS node overview

This package need some prerequisites in order to produce the expected results and those are given below.

1. dlib
   The developers have provided the user with a script file in order to install and to configure the dlib library to the system. All the commands are properly explain in the script file (install_dlib.sh) itself.

2. OpenCV

   - It has been confirmed after few different tests, that the *OpenCV* version which is compatible with ROS is OpenCV-2.4. If the user has already installed OpenCV, but not by the Ubuntu repository, it might create some problems. So if the user have OpenCV version 3.0 or higher it is recommended to remove that version and install the version provide by ROS. which can be installed from the following command. (this package however was tested under OpenCV - 2.4.8)

     ```
     sudo apt-get install ros-kinect-opencv3
     ```

   - If the problem still remains, it is best advice to remove all your OpenCV version and re-install the standard OpenCV library provided by Ubuntu.
   - ***Warning*** If the user has to remove *libopencv*, Please **backup** all the data before removing anything as it may cause corrupting the file-system.

3. cv_bridge

    - This package is use to communicate with ROS and OpenCV, it will work as a bridge between the two platforms. And can be install from typing the following command,

      ```
      sudo apt-get install ros-kinect-cv-bridge
      ```

4. image_transport

    - This package is use to transport images between the system and the program. In other words, this will work as a node handler for images. All the image publishing, subscribing goes through this image handler.

      ```
      sudo apt-get install ros-kinect-image-transport*
      ```

5. Camera packages - This will depend on what camera is been used.

    - Asus-Xtion PRO LIVE/ Kinect or for any 3D camera- For this camera use the following command,

      ```
      sudo apt-get install ros-kinect-openni2-launch
      ```

    - If the user is using normal web cam, user may download this package from github, the command is given below.

      ```
      git clone https://github.com/bosch-ros-pkg/usb_cam.git
      ```

3.4 Coordinate conversion

Once a face is detected, there needs to be a boundary box around the detected face. According to the information from the boundary box, the direction and distance of the face can be obtained using the parameters of the camera.

First, some basic background and definitions should be presented so as to make the solution clearer to explain. As is shown in fig 3.6, digital images are saved as matrix of numbers. According to the original parameters in the camera, the width and height of image could be obtained, which is defined as w and h. The size of images is depended on the size of sensor in the camera. The CMOS sensor is composed of numbers of pixels. The size of each pixel is defined as e. The distance between the lens and sensor of the camera is called focal length, which is defined as f. The principle of imaging in camera is shown in fig 3.7 . According to the triangle ratio principle, the angle between the vector from the image center to the face and normal vector of image is calculated using the parameter above.

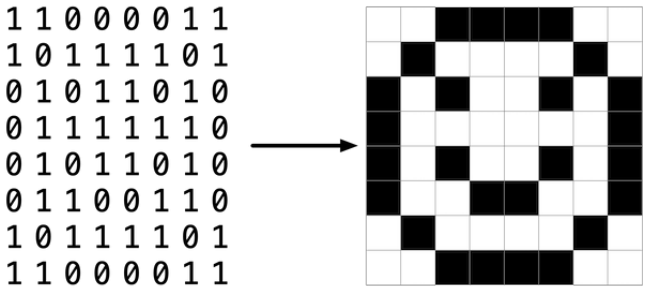


Figure 3.6: How the digital image is sorted

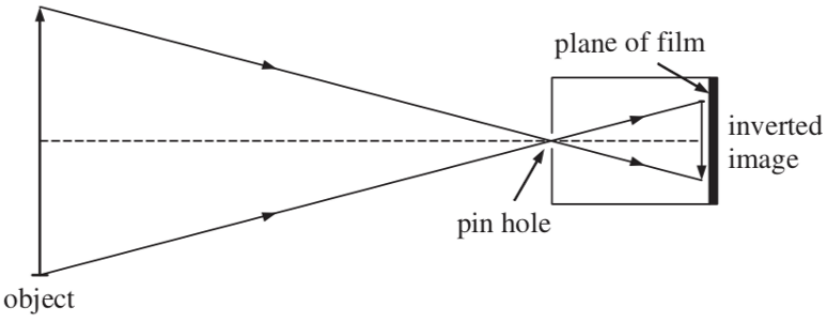


Figure 3.7: The principle of imaging in camera

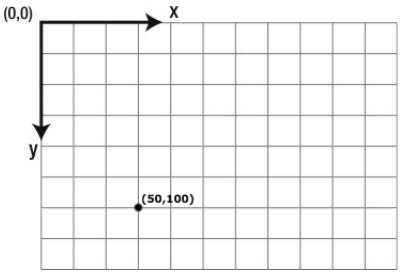


Figure 3.8: The coordinate system of image

According to the definition of image coordinate in fig 3.8, the coordinates of center point (C_x, C_y) could be easily obtained as:

$$C_x = \frac{1}{2}w \tag{3.4.1}$$

$$C_y = \frac{1}{2}h \tag{3.4.2}$$

Let's define the position of boundary box in image as x and y, and the width and length of boundary box as b_w and b_h. According to the principle of Euler Angle Rotational Sequence, there should be 12 kinds of sequences of rotation axes in total. In this project, only two rotation axes x and y is chosen. Assuming that the sequence of rotation is $x - y$, thus the rotation angles around axes x: θ_x and around axes y: θ_y is obtained:

$$\theta_x = arctan(\frac{(x - C_x)}{((f/e)^2 + (y - C_y)^2)^{0.5}}) \tag{3.4.3}$$

$$\theta_y = arctan(\frac{(y - C_y)e}{f}) \tag{3.4.4}$$

According to the size of boundary box, the distance d between the face and camera could also be estimated. To make it more reasonable, the distance could be estimated through the width and height of boundary box separately, which is defined as d_1 and d_2, and average them to obtain a more reasonable result d. Assume the average ground truth of human face's width and height is A_w and A_h, the estimated distance between camera and face could be obtained:

$$d_1 = \frac{A_w \cdot f}{b_w \cdot e} \tag{3.4.5}$$

$$d_2 = \frac{A_h \cdot f}{b_h \cdot e} \tag{3.4.6}$$

$$d = \frac{d_1 + d_2}{2} \tag{3.4.7}$$

Once the the distance of the face d and direction of the face — θ_x and θ_y are obtained, the position of the face could be located. Then the output angle for each motor could be obtained through inverse dynamic function so that the 'gun' would be pointed to that face.

Chapter 4

The Agent

To generate speech output from the robot based on user actions a dialog system or "agent" was implemented. We chose to use an agent rather than for example just sending string values directly to a speech synthesizer for pure educational purposes. The agent's structure is based on the intelligent agent Hazel (?, ?). We choose not to call our agent intelligent however since it lacks integral parts of what makes an intelligent agent "intelligent", e.g. no handling of user input since all input is handled outside the agent application. This agent thus falls into the category of a finite-state system (?, ?), in which the agent asks the user a rigid sequence of questions (in this case login information) in order to obtain the information necessary to solve a task (grant or deny the user access).

4.1 How the agent works

The agent defines a set of dialogs. A dialog is referred to as a context and contains a list of dialog items, each with a unique ID.

4.1.1 The dialog items

The dialog items are responsible for the agent's dialog. All dialog items are derived from the 'DialogItem' class and each instance has a context and ID associated with it that is unique. When starting the agent it will run the initial dialog item associated with the initial context and ID. When the dialog item has finished running, it will return the next context and ID which then becomes the current context and ID. Whenever the context or ID changes, the agent will look for the current context and ID and run the corresponding dialog item and so on. A dialog item can refer to any other dialog item. This allows for nonlinear dialogs, e.g. it is possible to repeat dialog items in a loop.

Note that the agent does not have an input item for handling user input. This means that the initial context and ID defined when starting the agent will determine the entire direction of the dialog. All user input (e.g. vision input) was handled outside the agent. To work around this problem we started the agent multiple times and changed the initial context and ID depending as desired to control the dialog.

The 'DialogItem' class does allow for an input item to be added later on however. In fact, all sorts of dialog items that weren't needed in this particular project can be added, e.g. items

for search memory items (if any) and searching the internet etc.

4.1.1.1 The output item

The 'OutputItem' class sends agent output (in the case speech output). It is associated with an output action that defines the next context and ID and takes a single pattern containing a string value. Calling the 'run' method will retrieve this value and send it to the speech synthesizer. It will also retrieve the next context and ID in the dialog.

For speech synthesis, the open source speech synthesizer eSpeak was used.

4.1.1.2 The wait item

The purpose of the 'WaitItem' class is to create a pause between speech outputs. It will pause the agent for a specified amount of time.

Chapter 5

Brain

5.1 Introduction

In order to control the different parts of the robot, and react accordingly to stimuli, the *brain* package was implemented. This implementation is limited to a finite state machine, where each state is defined as the mode the robot should currently be in. The following states were introduced in order for a smooth transaction between all the ROS nodes and the processes.

1. Idling
2. Tracking
3. Interrogation
4. FeedbackWaitingState
5. DecisionMaking

Furthermore we have introduced our own method of *action-lib* for a better output. It was observed that implementing an action-lib base packages was little bit expensive. There for *ros-service* was used in some instances along with a customize made *std_msgs::String* message called feedback. this message type was also generated in this package and was to send the feedback of processes between nodes.

5.2 State processes

5.2.1 Idling

The initial state is set as the *idling* sate, where no face is detected and the robot runs through a loop to scan the environment. The robot will move three of its motors, the pan,tilt and base servo in this idling loop. The parameters regarding boundaries of the robot movements can be set in the launch file provided for this package (brain.launch).

While executing this node it was needed to send the servo commands with small intervals. For this it was need to *sleep* the current process and therefor this was put into a separate thread so that main thread will runs without any interruptions and will still be able to

send/receive data via ROS communication protocols. *ros::Asyncspinner* was used to achieve this task.

5.2.2 Tracking

If a face is detected the brain switches to the tracking state, here the robot turns towards the human and follows their movements and it will also send commands to the *Agent* so that it will enable its greeting dialog. The tracking includes moving the pan and tilt angles of the neck and also it can move the its base so that it can follow if a person is moving away from the frame. After 10 seconds has elapsed and the person has not left the robots sight, the robot enters the interrogation state, asks for login credentials and starts the terminal application. This will also runs in a different thread and the internal communication (i.e. sending messages to Agent) will be done via the customized *Feedback* messages.

5.2.3 Interrogation

Once the process enters to the interrogation mode, it will first informed this to the user via an *Agent* speech output. The a terminal application will start in order for the user to enter the *credentials*. The output screen of the terminal application is given in the figure 5.1.

Figure 5.1: Terminal application

If the person correctly enters their credentials they are allowed to pass and the robot returns to its idle state and also the *Agent* have the ability to greet the user by its name (i.e. "Hello Justin"). For this authentication process a predefined file with the required details will be used. User will be given 3 chances to enter the correct credentials and with each wrong attempts warnings will appear in the terminal and the *Agent* will also give warnings via a speech output. Also note that even though the user name is visible, when typing the password the letters will not be visible. This measurement was taken in order to mimic actual authentication system.

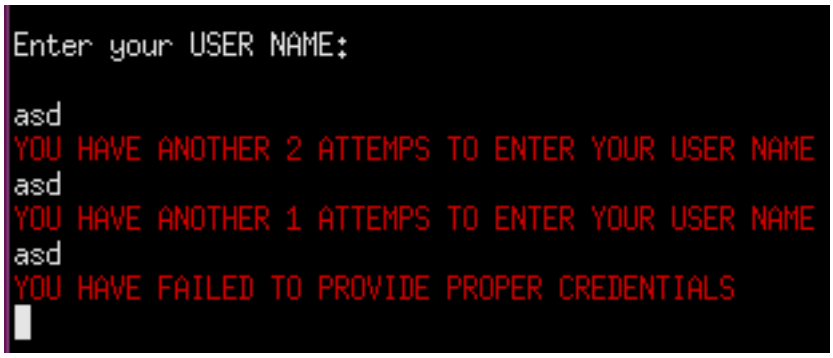


Figure 5.2: Warnings displayed in the terminal application

5.2.4 FeedbackWaitingState

The state will be moved from the *Interrogation* to *FeedbackWaitingState* once it has a decision. This could either be *access granted* or not. This state was introduced to handle the the transition section from *Interrogation* to *DecisionMaking*, as there will be few sub processes running in the background. Once the initial welcome or the warning is given the state will change to *DecisionMaking*

While waiting for the *Agent* feedback this state will command servos in the arm (the elbow and the shoulder) to move based on the results from *Interrogation*. If the access it granted the robot arms will move to its initial position, however if three wrong attempts are made that is if the user is not granted access, the robot will move its am to a *shooting* position.

5.2.5 DecisionMaking

This state will be responsible to handle the last phase of the operation. Of course if the person is given access system will be reset, and the state will move to initial idling loop, however if one was not given access then warning will be issued until that person leave the area or until it fire the gun. In here we have again introduced an warning states to handle smooth

transition. This is needed since the speech outputs from the *Agent* take different time laps for different dialogues and the warning commands to the *Agent* has to be send sequentially. One could set static time interval between each warning speech, but it will reduce the quality of the software as this time has to be change upon every single change in the dialog. The following are the warning stats define in the program.

1. NoWarning –> The no warning state
2. InitialWarning –> The initial warning state, this is triggered when the main state changes from *Interrogation* to *FeedbackWaitingState*
3. SecondWarning –> This is the second warning
4. FinalWarning –> This is the final warning

If the person move away in between any of these warning then the robot will go the initial state after resetting all the servo motors. However even after the final waring and after another 10 seconds the robot will *fire* the gun. In other words commands will be send to the *Arduino* to light up the laser.

5.3 System overview

Following figure 5.3 will describe all the nodes that will be running in parallel in order to make the whole process. This is of course a representation of the state machine which is presented in the system. In the next figure 5.4 it will display the topic transition between nodes.

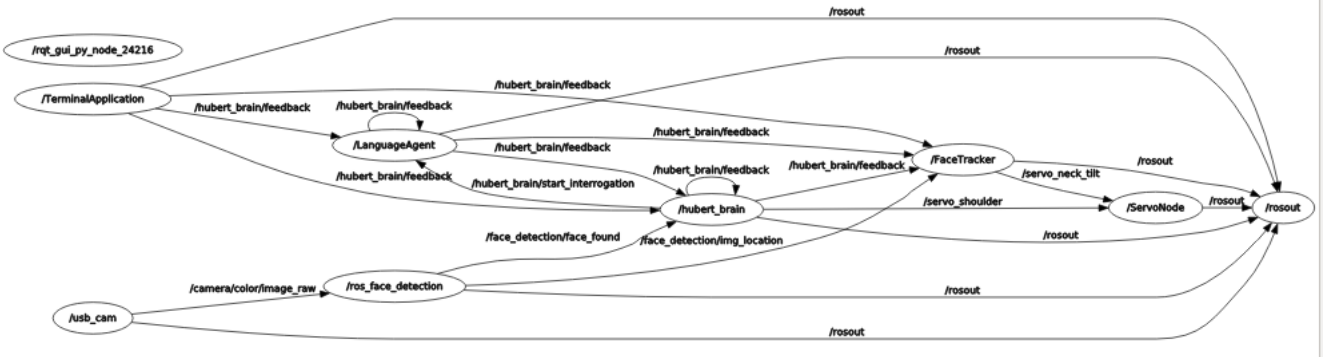


Figure 5.3: Node behaviour of the Hubert System

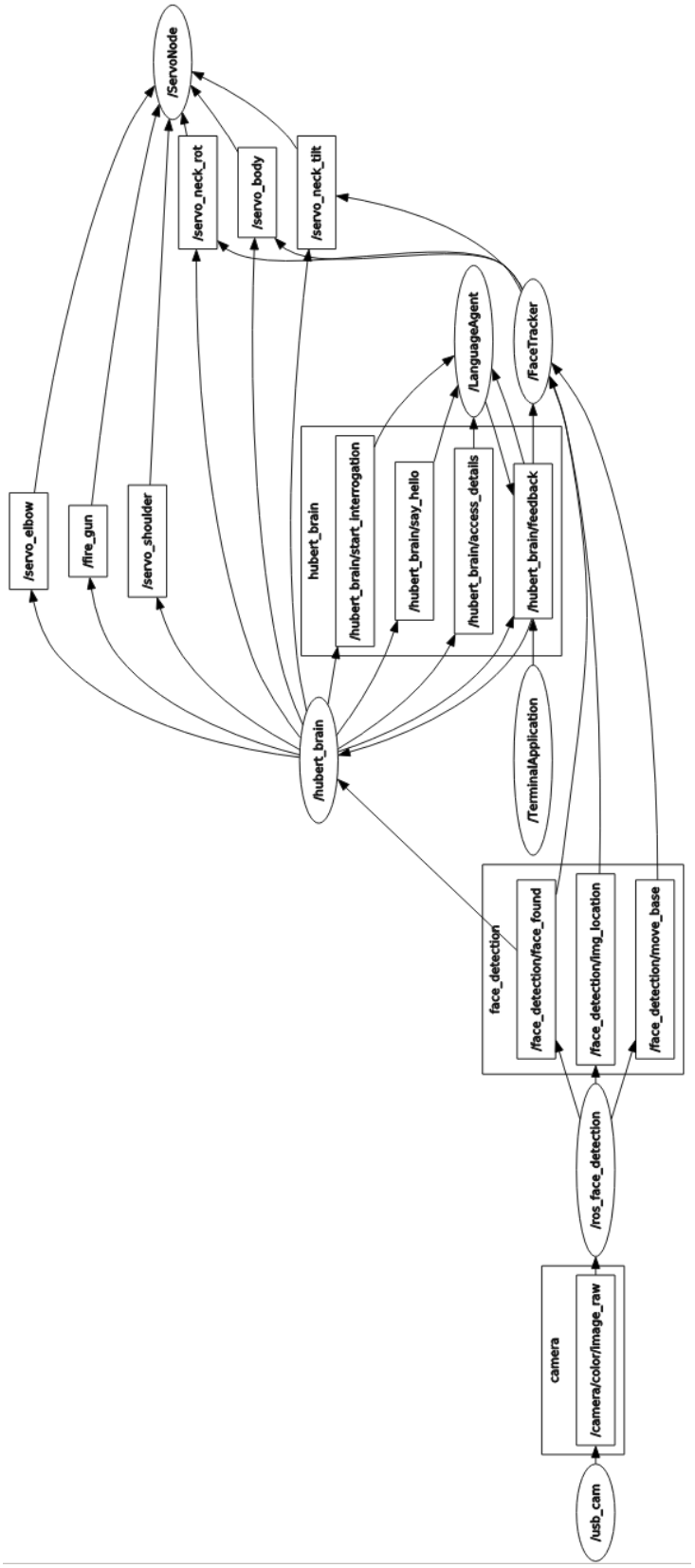


Figure 5.4: Topic transitions in the Hubert system

Chapter 6

Discussion and Conclusion

6.1 Goal statement

In the beginning of the project, the goal statement was defined as: "Before the end of this course, test a guard robot design, based on currently widely available technology, that can gate access to restricted areas. This work will explore the viability of such a system as well as deepen and broaden the knowledge of the team within key areas such as: face-detection, humanoid kinematics and decision making software." In regards to whether or not this goal has been reached, it is prudent to ask if each part of the project fulfills the requirements set by this goal statement.

Given the results obtained it is clear that the goal has been met, the robot can detect and interact with a human attempting entry, and, given certain criteria are met, it can defend the area it is guarding. Furthermore the software implementation was prioritized in this project and software are made as dynamic as possible with ROS wrapper for each package and therefore the scalability of this project is quite high. However there have been quite a few challenges throughout the project.

6.2 Challenges and issues

One of the biggest challenges in this project was the vibration of the mechanical structure. The rotation of the body is handled by a motor near its base, but the center of the gravity is located on the upper half of the robot.In addition, due to the method used to control the servos, the resolution of control is limited to $\pm 1^{\circ}$, and the rotational joint attached to the servo is able to flex. These limitations together cause some "jerky" movement when the robot is tracking a face, leading to the face detection failing due to the sudden movement. A possible solution to this would be to move the rotational joint higher up on the robot or moving the center of gravity further down towards the base joint. In addition, using a control system to only move one degree at a time, or increase the PWM of the servo signal more slowly could mitigate some of the rapid movements.

There also exists some lags in the system leading to slow response times. Some of these are thought to be limitations inherent to the Arduino, since it is a an 8-bit architecture running at 16MHz. There could be issues concerning the communication via ROS, as the Arduino may be too slow to maintain serial communication at the speed preset, whilst also controlling each servo.

6.3 Future improvements

By either switching to a faster microcontroller or switching to a more robust servo library, some of the jitters of the servos could be mitigated. The mechanical structure could also be modified so as to be more stable when moving, for example by changing the main rotational joint of the body and moving the center of gravity as has been previously discussed. This would improve the ability of the robot to perform face detection whilst moving.

Adding an input item for handling user input in the agent could allow for more flexible dialogue. With this addition it would also be possible to have a more thorough interrogation phase as the input could be passed into a parser, making a dialogue possible. In the same vein, adding a speech-to-text algorithm would give the robot the ability listen for the answers from the interrogated human, instead of relying on a terminal interface.

Another addition would be to implement a face recognition system, which would only let in people the robot recognizes. This was not attempted during this project as the issue with face recognition algorithms are their sensitivity to changing light conditions. Giving false or no recognition matches unless using advanced algorithms. Allowing the robot to affect these conditions, by controlling bright lights for example, could go some way to mitigate this issue, however the hardware might need modification to accommodate such a change.

One huge benefit of using ROS as an environment for this project is that all these improvements are possible to test and verify on their own and can then be slotted in with the existing systems.

6.4 Conclusion

Going back to the goal statement, it can be said that the project has fulfilled the specification set out at the beginning. Even if it does not live up to the features of commercially available offerings, it proves the viability of humanoids being used in this capacity.