ABC Wheelchair

by

Bryce Ozzie Beaumont

Project Supervisor: Dr. Nasser Asgari

May 2020

Submitted to the College of Science and Engineering in partial fulfillment of the requirements for the degree of Bachelor of Engineering (Robotics) / Master of Engineering (Electronics) at Flinders University, Adelaide, Australia

Declaration

I certify that this thesis does not incorporate without acknowledgment any material previously submitted for a degree or diploma in any university; and that to the best of my knowledge and belief it does not contain any material previously published or written by another person except where due reference is made in the text.

Signed: Dated: 30th May 2020

Acknowledgements

I would like to extend my appreciation and gratitude towards my supervisor Dr. Nasser Asgari for his support and guidance throughout my project and time at Flinders University.

I would like to thank RajKunwar Kukreja for his previous work on the ABC Wheelchair and his assistance when first starting the project.

Finally, I would like to thank my family and friends for the support, patience and confidence they have provided me with on this journey.

Abstract

Wheelchairs can provide a sense of freedom to people who are unable to walk as many others can. Producing a smarter wheelchair is a necessary step for extending this freedom to individuals who possess motor and/or visual impairments that cause operating a wheelchair difficult or impossible. The ABC Wheelchair at flinders university is pursuing the development of an addon technology for powered wheelchairs that can provide autonomous navigation and allow for multiple input methods for users to utilise.

This project researches the installation of SLAM on the ABC wheelchair and the development of a guided doorway navigator that can guide the chair through narrow doorways. The doorway navigator used a LiDAR to return range information on the environment and recognise the position of door frames by measuring the range differences between data. Assumptions were made when ranking the potential doorways detected to allow for successful detection, such as the navigator being triggered by an external process so doors would already be close to the chair.

Using the LiDAR the system was capable of performing accurate SLAM using the Cartographer system by Google, mapping and localising the position of the chair over time. Furthermore the doorway navigator could successfully distinguish open doorways from LiDAR range data and a velocity controller provided safe travel through the opening.

Additional research areas were focussed on the development of alternative input methods to provide accessible technology to a diverse audience through the means of developing a pupil detector for future gaze trackers and testing the capacity of the MyCroft voice assistant to control the wheelchair.

Pupil detectors built need further development before they can be used for gaze trackers. The first detector is highly accurate, finding pupil location within 1 pixel of error, however lacks responsiveness. The second detector is highly responsive although has higher levels of error.

The MyCroft voice assistant could control chair behaviour and allowed for multiple keyword or phrases to be used as triggers but required network access. A local speech recogniser may be run on the machine to remove the need of network access however is limited by system resources to run the deepnet speech recognition locally, slowing down the execution of real-time systems required for safe autonomous navigation when approaching obstacles.

Table of Contents

Table of Figures

Table of Tables

1 Introduction

With over one billion people in this world having some form of disability [1], it is important that researchers develop systems to assist the physically impaired to improve mobility and reduce the strain of tasks that fully able-bodied people take for granted. The research topic for this thesis is an autonomous wheelchair that will be controllable from multiple inputs to assist people who are unable to use standard powered wheelchairs.

Wheelchairs play an important role in mobility for people who are unable to walk. Research has shown that the supply of wheelchairs to people with disabilities promotes social development alongside improved mobility and reduced dependency [2, 3], and provides better quality of life to persons who previously did not possess this mobility aid [4]. The study by [5] showed increased mobility, dependence, and increased mood state among 519 participants over three different countries after 12 months of receiving a wheelchair. By adapting a powered wheelchair to accept alternative inputs to a joystick such as brain control or voice prompts, it will allow disabled populations who are unable to use joysticks to attain increased movability and improve quality of life. It has been noted that there are many individuals of whom are unable to easily use manual or powered wheelchairs and require alternate mobility aids such as smart wheelchairs (SW), particularly for people with impaired vision, motor impairments or cognitive deficits [6]. Powered wheelchairs can be difficult for people with motor and visual impairments to operate in confined spaces [3, 7], which is where autonomous technology can play a pivotal role in improving independence and reducing the strain of wheelchair operation on individuals.

Research into autonomous wheelchairs can be translated into products for able-bodied individuals also. Such technology can be used as a mobility aid to guide people in places they are unfamiliar with such as airports or public malls that can be confusing to navigate and do not always have signage that is easily discerned, especially if you are unfamiliar with the local language or are illiterate.

The project aims to deliver a system that can be installed onto new or existing electric wheelchairs to navigate its environment. The system is to be capable of mapping its environment and navigating the generated map whilst avoiding obstacles to reach a destination, and accept input from a user.

2 Background

Initial SW were mobile robots with seats whereas modern chairs are now typically modified power wheelchairs with computers and sensors incorporated [6, 8] to perceive their environment. SW developed in the past thirty years are commonly characterised as seated mobile robots, integrated prototypes or equipped wheelchairs [6, 9]

- *Seated Mobile robots:* these were the initial smart wheelchairs, composed of mobile robots that were equipped with a seat to provide alternative mobility to users.
- *Integrated prototypes:* prototype chairs start from an experimental wheelchair frame upon which a wheelchair is built around to construct a fully customised smart chair.
- *Equipped wheelchairs:* equipped chairs are modified powered wheelchairs with computers and sensors attached to provide 'smart' features to the wheelchair. These are a popular research topic and typically work by emulating the signals sent from the joystick on the chair to the motor controller and using this to control the chair movement via a computer.

Current research into modern SW have led to the development of systems with varied input devices, improved navigation implementing obstacle avoidance, map generation and positional localisation utilising an array of sensors and data fusion to produce accurate and reliable systems for wheelchair users to utilise [10].

Current development of SW can be categorised into two main branches [9, 10]

- Human-machine navigation
- Predefined machine navigation

Human-machine navigation can be described as machine assisted human navigation. The human will use an input device to control direction of the wheelchair and a machine will alter the trajectory if it detects obstacles to avoid accidents. This method of navigation provides more freedom to the user whilst easing the strain of maintained navigation and improving safety through recognition and avoidance of possible hazards.

Predefined machine navigation is a setup that utilises a series of set destinations or tasks for users to select for navigation. The machine will handle all navigation without any further input from the user. Predefined navigation is ideal for users with high impact disabilities that make

accurately maintained manual navigation difficult, such as people with conditions that cause muscle spasms in arms/hands.

Research from the past 13 years into SW have produced results with varying input methods paired with assisted or completely autonomous navigation. A number of these ventures have avoided the standard joystick controller and opted for innovations such as speech recognition or brain signal reading, summarised in Table 1.

Table 1: Summary of Smart Wheelchair Technology

Wheelchair	*Input Method(s)*	*Description of System*
Robotic Wheelchair with Dialog Manager [11]	Speech Recognition	A dialog managed wheelchair used for navigation to recognised keywords. The dialog manager setup on the chair uses a learning model (Partially Observable Decision Process) to improve recognition and provide a more natural human interaction with the chair.
ARTS Self Docking Wheelchair [12]	Touch-screen Interface	Autonomously docking wheelchair using an attached LIDAR to recognise a designated lift and navigate onto it. The LIDAR recognition system uses filtering and two reflector panels permanently affixed to limit the points of detection so the chair can recognise the lift and control the motor to navigate onto it.
EEG-based Brain Controlled Robotic Wheelchair [13]	Brain Controlled Interface	Prototype hardware for controlling wheelchair motors using a wireless Electroencephalograph (EEG) device fitted to the scalp. Brain waves are captured by the EEG device are processed through a neural network on an external PC to remove signal noise and translate signals into movement commands that are sent to the motor via an ARM 7 based CPU.
Fuzzy EKF Controlled	Computer Controlled	Intelligent wheelchair utilising fuzzy logic to execute flexible and non-linear obstacle

Intelligent Wheelchair [14]	Execution of Autonomous Navigation	avoidance. The wheelchair uses an Extended-Kalman Filter (EKF) approach to fuse data from wheelchair sensors to enhance localisation capabilities and allow for autonomous navigation to pre-defined navigation goals.
Gaze Driven Power Wheelchair Addon [15]	Gaze Tracking Interface	An addon gaze-tracking system for power wheelchairs providing alternative input control to traditional joystick. The addon system controls the wheelchair motor using an interface with directional arrows superimposed on a real-time camera feed of the environment in front of the wheelchair. The gaze-tracker determines desired movement based on what arrows the user is looking at. The wheelchair avoids obstacles by staying within a real-world boundary constructed of reflective tape on the ground. The tape is detected using an optical sensor and provides a safety mechanism for users with motor impairments that may affect gaze control.
Smart Wheelchair for Autonomous Navigation in Urban Environments [16]	Computer Controlled Execution of Autonomous Navigation	Powered wheelchair with equipped 3D Lidar for localisation and mapping. The chair is designed for autonomous navigation in urban settings. Localisation is generated using Extended Kalman Filtering to determine location based on recognition of recorded landmarks (trees) after first taking a single GPS reading before navigation is executed. The wheelchair using a cloud-based mapping service to generate landmark-based maps for the chair to use for path planning and localisation.

Research into smart wheelchairs are branching away from the standard joystick approach and turning towards alternative input utilising human speech, movements, or brain waves to assist

individuals with severe motor impairments, though this is not necessarily enough to provide safe mobility. Aiding wheelchair users with autonomous path planning and environmental recognition can provide safety measures to assure the chair can alter the user input to avoid obstacles and provide totally autonomous navigation to desired locations, easing the strain on wheelchair users with high level motor impairments. The research into the gaze driven power wheelchair addon [15] reflects the ideas discussed here to enhance safety, supplying alternative gaze tracking input for wheelchair control paired with obstacle avoidance for altering the trajectory of the chair controlled by the user to avoid collisions. This addon system is a shared control system between human and machine to provide safer and more efficient mobility to the user. This is different to the chairs described in [14, 16] where a computer controls all navigation to a pre-determined goal. For complete autonomous navigation the user will only interact with the chair to supply a destination, upon which the chair will plan the path towards the goal based on a generated map of the environment and control all components of the trip to safely navigate to the goal.

For systems that use autonomous navigation, it is essential that they maintain an accurate bearing of their location on a generated map. This is difficult as the chairs are primarily used indoors and common tracking methods such as GPS are not always accurate enough to capture the chair coordinates within buildings [16, 17]. To overcome these issues time is spent incorporating a suite of sensors to detect chair movement (Odometry, IMU) and recognise features and landmarks (Camera, Image Recognition, LIDAR). Each sensor on its own is subject to noise and does not always provide an accurate feed for predicting location, as the sensing equipment is often based on calculations for *estimating* position and not a direct reading. Recognised landmarks may trigger false positives and identify the wrong location or wheels may experience slippage which will affect odometry readings. To increase accuracy of estimated position using sensors, the readings are commonly combined using data fusion techniques to reduce the impact of noise on the system and account for the build up of errors to a degree. One data fusion technique is with Extended Kalman Filtering (EKF) as is used in [14] to combine multiple sensor readings for joining however particle filters (PF) are also This common for use in data fusion for localisation tasks [17].

2.1 DIRECTION FOR ABC WHEELCHAIR

Literature studied incorporates numerous input systems and sensing technologies to achieve a common goal of assisting individuals with motor impairments. The goal for the ABC wheelchair is to deliver an installable system for powered chairs that assists navigation. To deliver a system that is accessible to a large range of users with differing levels of motor impairments, it is important that there exist differing user input methods to control the chair. One method will be the brain control interface allowing users to control the chair via thoughts. Not all individuals will find such a system comfortable to control though and could be deterred from using the system based on the input method. To prevent this from occurring it is important to focus on providing alternative methods of input for controlling the chair as well as developing the mapping and navigation capacities of the wheelchair.

Particular points of interest for mapping and navigation at this point were to use a budget LiDAR to replace the Kinect Camera Sensor to map the environment and provide informed navigation through doorways that the Kinect Sensor was not capable of handling. To summarise components of this project:

- Mapping and Localisation with LiDAR technology
- Doorway Navigation
- Alternative User Input Methods
    - Gaze Tracking
    - Voice Control

Expanding on alternative user input methods, gaze tracking and voice control were the chosen methods as they can be utilised with the current technology available on the wheelchair and to provide a spectrum of input control. By providing visual, vocal and thought input methods later on, users with high level impairments will have a suite of options to choose from for controlling chair operations.

3 Theory

3.1 ROS ENVIRONMENT

The platform used for the development of the ABC wheelchair is the Robotic Operating System (ROS). ROS is an asynchronous framework composed where all processing is completed within nodes that can subscribe and publish to topics for controlling aspects of a system such as navigation, localisation and recognition [18]. Topics contain information like the system pose, state, control variables, navigation plans, landmarks, etc. Any ROS based system requires a roscore to be executed which is the collection of nodes and programs necessary for communication via topics [19].

ROS utilises a graph like framework where nodes subscribe and publish to any number of topics to transmit the necessary data to control and monitor the system they are built for. An example of this can be seen below in Figure 1 below, illustrating the communication between three nodes (contained by ellipse border) using three topics (contained by square border). In this example the node handling odometry of the system (/odom_node) publishes to a topic */distance*. The */motor_controller* node is subscribed to the distance topic and obstacle topics that is used to control the motor by publishing velocity information to a */cmd_vel* topic.

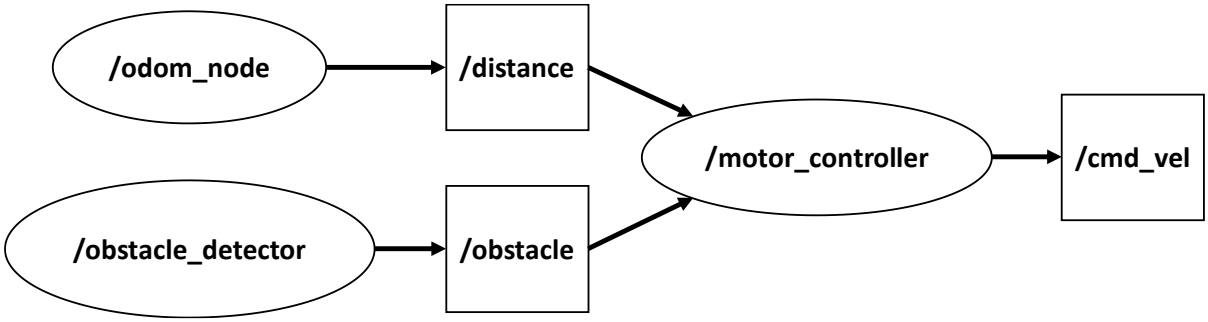


Figure 1: Example of ROS nodes communicating via topics

ROS itself is not a real-time framework however can be setup to work like one. Real-time hardware can be setup to work with nodes processing code in real-time, subscribing and publishing to topics with little delay to control and monitor hardware [18, 19]. This setup allows ROS to work as middleware for real-time robotic systems and is a popular choice for modern robotics.

3.2 LOCALISATION AND MAPPING

For robotics research looking into autonomous navigation for indoor use, it is important that the robotic system be able to recognise the environment as well as where it is in the environment. A common method for accomplishing this is to use simultaneous localisation and mapping (SLAM) algorithms to create maps of the environment and match sensor readings to locations on the map for determining location[17, 20]. There are many types of SLAM algorithms with their own benefits and trade-offs. The common trade with SLAM algorithms is between accuracy and speed.

Common algorithms for SLAM are based on variations of the EKF and particle filter (PF). EKF is typically not used on its own for SLAM due to the inaccuracy of the filter that occurs during the linearization process. Due to this particle filtering is often utilised as it is non-linear in nature and is an effective tool for localisation purposes. The particle filter works by comparing a sample of particles to the expected result and refactoring samples down to the most likely occurrence [20].

SLAM algorithms are typically focused on map generation and localising a robot on within a map, but a variant of SLAM exists called Pose SLAM which calculates only the trajectory of the robot to navigate environments. The map being generated does not contain information on obstacles but instead the trajectory is used to record the pose of the robot. The trajectory of the robot used when building the map is free of obstacles and hence does not require obstacle data be recorded as the poses stored in the map have already been used and deemed safe and free of collisions [21].

Common SLAM software for the ROS environment are GMapping and Cartographer. Both software generate maps of unknown environments by taking in sensor readings from optical or directional sensors to estimate location and obstacle positions. Generated maps can be used by path planners for autonomous navigation and support is available for exploration and mapping of unknown environments.

3.3 ODOMETRY

Odometry is a technique that uses sensor data from a machine to estimate a change in position over time. The most basic form of odometry is wheel odometry using encoders attached to wheels of a machine to estimate change in position based on the number of revolutions the wheels have performed. The issue with wheel odometry is that errors build up over time due to

wheel slippage that may occur on slippery or uneven surfaces. Wheel odometry is dependent on the previous odometry calculation to estimate position so when errors occur they stack and result in declining accuracy of pose estimations [17].

A more reliable form of odometry is visual odometry that uses a system of one to several cameras to estimate trajectory using feature extraction. This is similar to SLAM however focuses on estimating the local trajectory as opposed to estimating the global map and trajectory that are consistent with one another. Visual odometry still has issues with error stacking contributing to drift in trajectory however is seen to perform more reliably than wheel odometry [17, 22].

3.4 VIRTUAL ASSISTANTS – MYCROFT

Virtual assistants have become a common piece of technology for modern day life, with major companies such as Amazon and Google developing their assistants to assist with virtual tasks and control devices connected to an Internet of Things (IoT). Virtual assistants are commonly controlled through speech input, where typically a trained neural network will process recorded speech and decipher to text for determining actions to perform.

An open source voice based virtual assistant is called ‘Mycroft’ which offers developers the ability to create custom ‘skills’ [23]. Skills in this context are python programs executed by the assistant when key words or phrases are recognised. The process works by supplying an intent, an action, and a dialog. The intent is a set of phrases or keywords to be associated with the skill, which when detected will perform the action associated with the intent. Skills can have multiple intents and actions associated to them. The dialog is a collection of phrases for the assistant to speak back for each intent. The assistant will cycle through dialog responses to provide more human like speech flow.

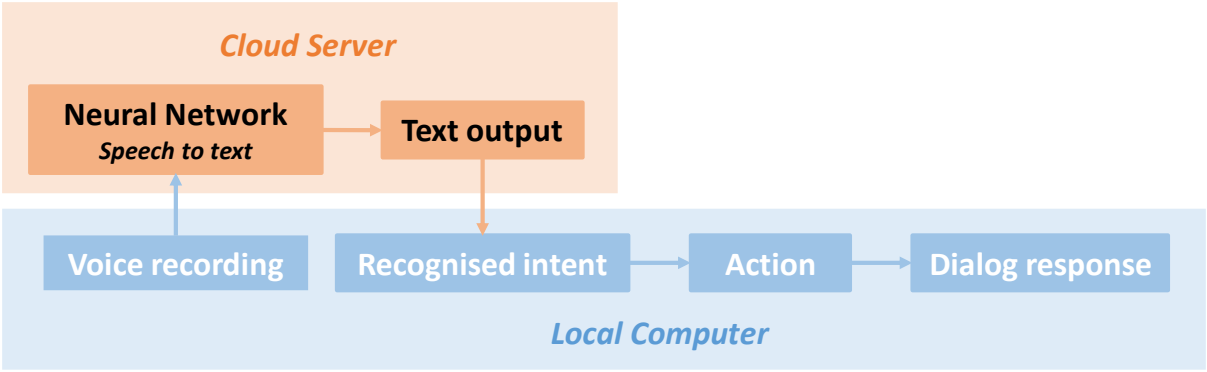


Figure 2: Flow of Mycroft Skill Activation

Mycroft handles speech to text through the cloud-based Google Speech To Text (STT) engine. Due to this one limitation with the assistant is that it requires internet access by default to recognise speech [24]. The assistant can be used without an internet connection by feeding text directly to the assistant or by installing a local speech to text converter however these converters can be very large and require additional processing power.

3.5 GAZE TRACKING

Eye-gaze tracking is a method of discerning ocular movements to determine where an individual is looking, offering an alternative Human to Computer Interface (HCI) that does not rely on the use of limbs from the neck down [25]. Modern applications of eye-gaze tracking involve analysis of video streams to identify eye movement between frames, offering a non-invasive method of measuring movement unlike earlier systems that made use of the placement of solids onto the eye or the attachment of electrodes around the eyes [26, 27]. Most video based trackers rely on infra-red sensors and multiple camera setups that are very accurate and provide fast response rates but the systems are complex and have high costs associated [26]. Modern research exists on the extraction of ocular movement from video streams using single cameras and image processing techniques, based largely on the extraction of features within the eye to discern pupil position and track movements. Such research employs binarisation of video stream frames through edge or threshold filtering before employing feature detectors to find elements such as circles or corners within the eye as a base of reference when determining ocular position and movement [25, 26, 28].

3.5.1 Open Source Computer Vision Library - OpenCV

OpenCV is an open source library for computer vision and machine learning, intended for the provision of common infrastructure for computer vision and machine perception [29]. The

library contains applications for image processing, providing edge detection, morphological operations, binarization of images and more to provide an ideal suite of tools as well as feature extraction and identification of faces and eyes within images [29-31].

3.6 ELECTROENCEPHALOGRAPHY (EEG) FOR BRAIN CONTROL INTERFACE (BCI)

An electroencephalogram is a recording of brain activity, often taken using a series of metal pads equipped to an individual's scalp to measure the electromagnetic radiation produced by the brain. The use of EEG as in interface for controlling devices is now possible as seen by research into the control of artificial limbs and wheelchair systems using headsets to record and process brain activity to execute commands to such technology [32, 33].

In order to control a device the brain activity must be isolated for a particular thought. This is achieved by recording the signals produced by the brain when thinking of a specific image, action, or word to be associated with a command, which is then used as a reference for recognising the thought. The signals measured are susceptible to noise and are filtered to isolate the brain waves from external noise for use in BCI technology. Using the filtered brain wave for a particular thought, the measured brain activity of an individual can be analysed and matched against the particular thought as an input for a BCI [32, 34].

4 Method and Results

4.1 SIMULATING THE ABC WHEELCHAIR

To simulate the wheelchair for testing outside of the lab and provide a virtual framework for SLAM software track wheelchair location, a Universal Robot Description File (URDF) was created. The URDF represents the layout of the wheelchair components, specifying how pieces of the chair join and the dimensions and positions of components on the chair. An URDF file is used to detail the robot kinematics to ROS, detailing the relationships between joint and link connections that is required for representing a robot in a virtual environment or real-world environment.

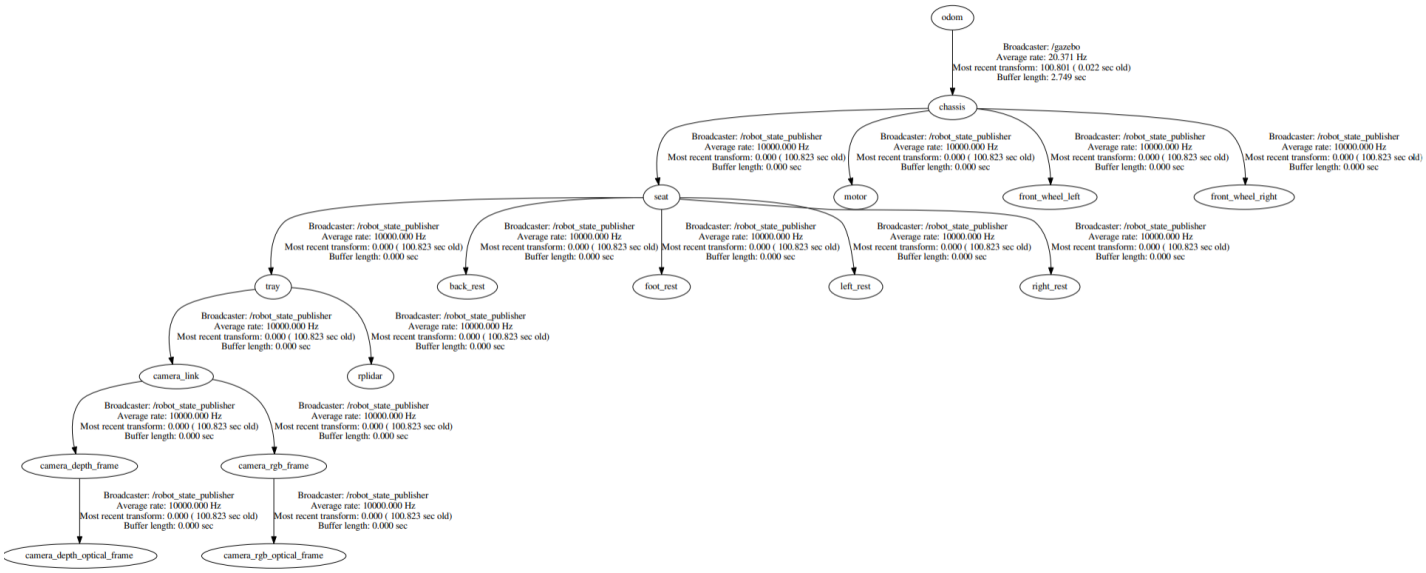


Figure 3: Wheelchair Frames - virtual representation of wheelchair for simulations and SLAM software

4.2 SLAM USING CARTOGRAPHER AND LIDAR

Previously the SLAM mapping produced by the chair was generated with a SLAM based mapping algorithm (Gmapping) utilising wheel odometry, IMU feedback and depth scan data produced by a Kinect 3D camera. Due to noise present in wheel odometry and IMU the localisation capabilities of the chair were hindered. To overcome these limitations visual odometry was generated using the Kinect sensor data which improved the localisation, though maps were reported as being choppy and position was not always accurately recorded due to the small frame and depth ranges that the Kinect is capable of detecting. To improve the capacity for the ABC Wheelchair to sense and record the environment and location within

recordings, a LiDAR was integrated onto the chair to provide depth sensing of up to twelve metres in three-hundred and sixty degrees of view.

4.2.1 Setting up Cartographer (SLAM) for ROS

The Cartographer platform is available for installation within the ROS environment. The resources required to install the SLAM platform are made available to the public by Google [35].

The Cartographer SLAM algorithm needs access to data on the environment from a range finder or similar measurement device to report depth information, specified within the configuration files of the ROS workspace cartographer was installed in.

The LiDAR range output was setup to provide data on a ROS topic named *'scan_lidar'*, which cartographer was pointed at for range data. The wheel odometry topic generated by the chair, *'wheel_odom'*, was also supplied to cartographer for additional odometry information to assist with estimating odometry metrics.

To provide tracking frames for the SLAM platform to establish pose and estimate odometry, cartographer was setup to use the chassis of the wheelchair as the tracking frame. This was performed by using the URDF that provided a reference to all components of the chair, supplying cartographer with details on the location that range data was being generated from.

Running Cartographer alongside the wheelchair generated the SLAM output, providing mapping and localisation as the wheelchair moved around the room. The SLAM algorithm was tuned within the configuration file, systematically adjusting settings affecting the behaviour of localisation and mapping to enhance performance.

4.2.2 Transition from Reliance on Wheel Odometry to Visual Odometry for Localisation

The wheel odometry provided by the physical chair was not ideal for tracking position. The system could not accurately determine a resting position, wheel readings did not detect turns in most cases, and translations within a global map were not reflected accurately as error built up over time. Identified errors that contributed to the unreliable odometry include wheel slippage on the vinyl lab floors, worn tires that have led to unevenly shaped wheels, and IMU sensor data containing high noise data for acceleration forwards and sideways. The odometry

data generated through the encoders and IMU were seen to hinder SLAM algorithms due to the high level of inaccuracy present in data generated. The impact can be noticed visually when comparing the generated outputs in Figure 4 and Figure 5, both recorded in the same room following similar paths though the algorithm is heavily disrupted by the wheel odometry and IMU readings that lead to wheelchair pose estimates being heavily off target and surroundings not being recognised as a result.

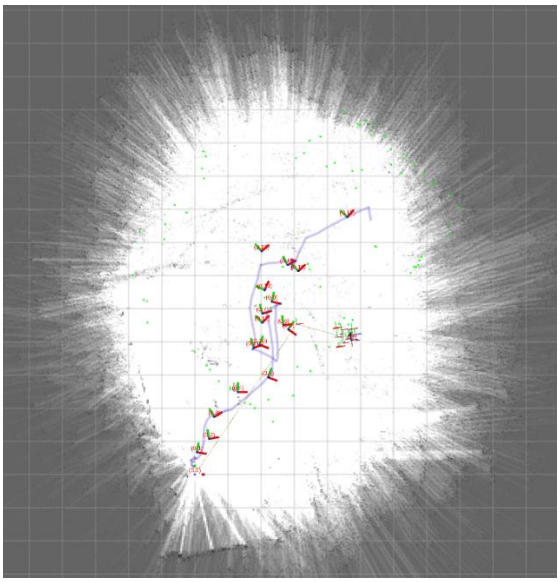


Figure 4: SLAM with Cartographer using LiDAR and Wheel Odometry to map the laboratory

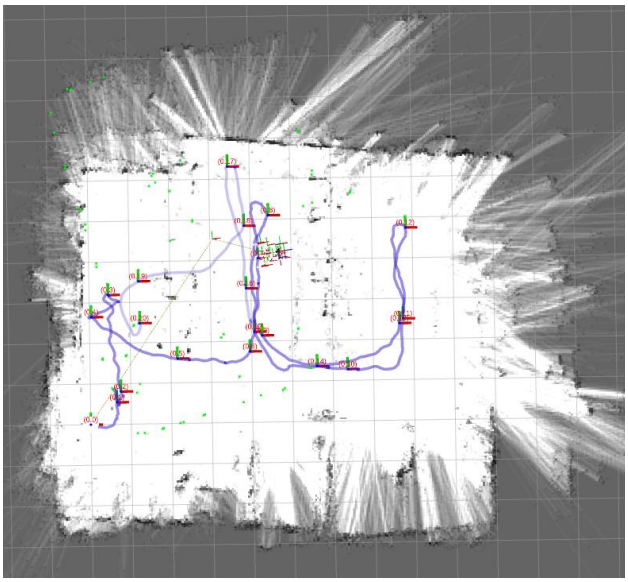


Figure 5: SLAM with Cartographer using LiDAR relying on generated Visual Odometry to map the laboratory

The map and estimated poses in Figure 4 integrated the odometry produced using the wheel encoders and IMU readings. The Cartographer SLAM algorithm could not map the room accurately. This was due to the conflicting data present between the odometry generated through the encoders and IMU, and the odometry generated visually by the Cartographer SLAM algorithm using the LiDAR readings. When the same room is mapped again without the IMU or wheel odometry, the output in Figure 5 was produced. The output generated was a far more realistic representation of the path followed by the wheelchair and the boundaries of the room.

4.2.3 Effect of LiDAR resting angle on SLAM algorithm

The LiDAR used (RPLiDAR A1) detects obstacles within a 12m radius on a single plane of vision. Unfortunately the world does not operate in a single plane of view and as a result obstacles are often missed due to an obstacle resting above or below the search range of the

LiDAR. The LiDAR was tested in several positions between 0 and 45 degrees to test the operation of the SLAM system with a tilted range finder.

4.2.3.1 LiDAR with 0 degree tilt

Tuning the Cartographer SLAM algorithm, the output in Figure 6 was produced. The tuned algorithm can maintain an accurate representation of the wheelchair pose utilising the lidar input to create visual odometry. The boundaries of the room are well defined and crisp although details are missing on desks and tables that were too low for the lidar to detect.

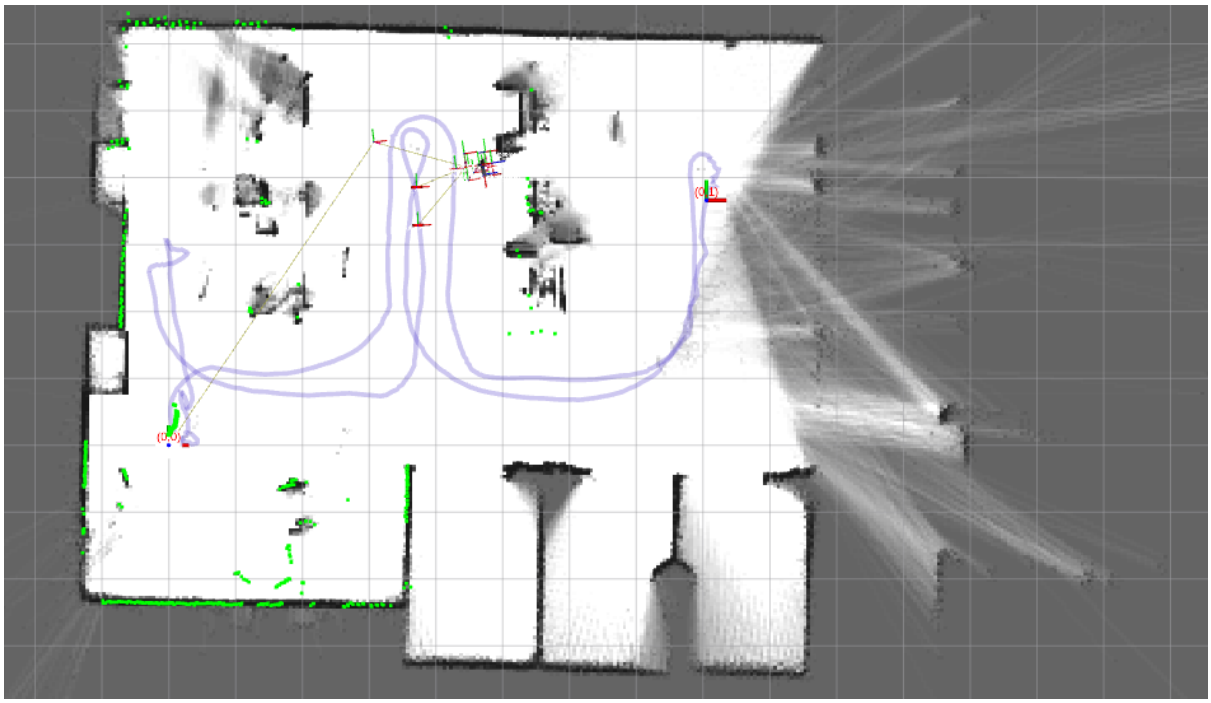


Figure 6: Tuned SLAM output for Lidar with 0 degree tilt

To achieve an accurate representation of pose, the algorithm was tuned to allow for a larger maximum rotation when matching range data received by the lidar. The lidar used had a slow refresh rate which contributed to greater changes of rotation and translations between lidar scans. By increasing the maximum rotation allowed for matching range data, turns were more accurately recorded and sharp turns could be recognised by the SLAM algorithm. This was also performed for translation by increasing the maximum translation allowed when matching ranges and heavily reducing the cost of performing such rotations and translations as the range recognition in the SLAM algorithm was now the sole means of determining pose changes, allowing the algorithm to compensate for the slow refresh rate of range data.

Additional changes were made to improve performance of mapping capabilities and range matching to provide smoother transitions between poses and more accurately detail boundaries of the room and obstacles. This included reducing the cost of mapping features, allowing points that were nor consistently caught by the lidar to appear on the map output such as the tops of desks and chairs, as well as increasing the number of local maps utilised to build the global map and adjusting constraints on how well points needed to be matched. Allowing inconsistent ranges to be mapped resulted in areas containing glass to be mapped with less certainty such as the right hand side of the room in Figure 6, allowing glass to be represented in some detail unlike the default algorithm which reported glass as empty space between the window frames detailed on the right hand side of the room in Figure 5.

4.2.3.2 Increasing LiDAR tilt up to 30 degrees

Increasing the tilt of the LiDAR to position the detection region closer to the ground in front of the wheelchair yielded promising results. As the tilt increased from 0 up to 30 degrees, local maps generated included more details of obstacles in front of the chair, though as the tilt increased the default settings for the SLAM algorithm were not able to estimate pose or provide an accurate representation of the global map, unlike that for LiDAR readings no tilt.

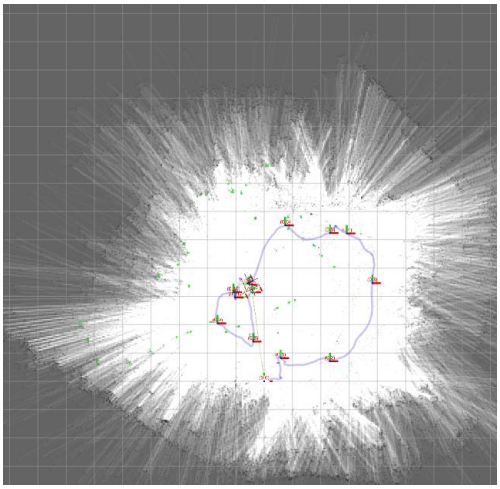


Figure 7: Initial SLAM output for Lidar with 20 degree tilt

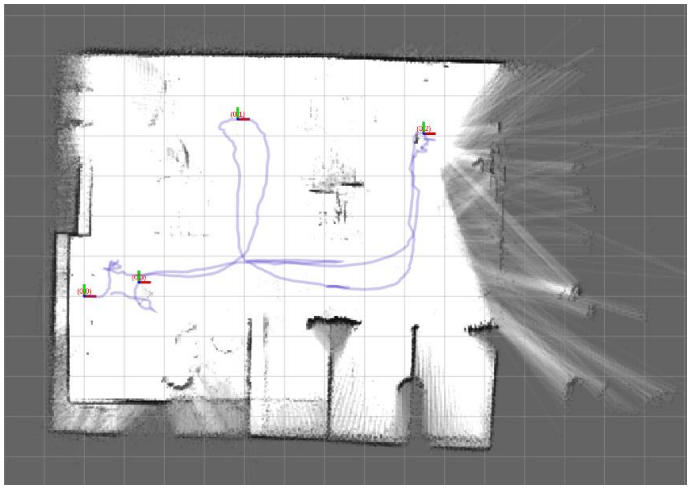


Figure 8: Tuned SLAM output for Lidar with 20 degree tilt

Tuning the algorithm to be more forgiving when matching range data from the LiDAR, and increasing the maximum rotation and translation allowed when estimating pose, the output began to produce results similar to Figure 8 with a level LiDAR. The global map produced by the Cartographer SLAM algorithm contained less details on obstacles around the room as the tilt of the lidar increased. A trade was made in detail present on the global and local maps,

where the presence of obstacles detailed on the global map was inversely proportional to the increased tilt and was directly proportional to the obstacles detailed within local maps.

Using a tilt of 20 degrees with a tuned algorithm produced the most promising outputs. Maps generated held the shape of the rooms being mapped, local maps contained more information on obstacles in front of the chair than that of a level LiDAR, and the pose of the wheelchair was still estimated within a high degree of accuracy. The algorithm could estimate the pose of the chair as it circled through the room and returned to the starting position. This functionality was lost as the tilt on the LiDAR was increased beyond 20 degrees.

4.2.3.3 Increasing LiDAR tilt beyond 30 degrees

Beyond a tilt of 30 degrees the quality of range data from the LiDAR began being reduced. The maximum useable range captured in front of the chair was heavily impacted as the LiDAR scans began to coincide with the ground closer to the chair as the tilt was increased. To prevent the ground from being detected as an obstacle, the maximum useable range for the Cartographer SLAM algorithm had to be reduced from the 12-metre distance that the LiDAR is capable of detecting. A tilt beyond 30 degrees reduced the number of matchable points in the range data for the algorithm to detect, impacting visual odometry and the quality of the global map. At this point only the local maps were useful for detecting obstacles in front of the wheelchair.

4.3 EDGE BASED DOORWAY DETECTION AND NAVIGATION

Taking the concept of tracked navigation utilised in research by ARTS Self Docking Wheelchair [12]

Using range data from the LiDAR, the layout of the environment around the wheelchair can be visualised and processed to distinguish features in the environment. By identifying significant changes in range data between points, openings within obstacles can be isolated to estimate the location of a doorway and provide a reference point for the wheelchair to navigate towards and pass through narrow spaces.

For the purposes of section 4.3 of this thesis, an edge is referring to a distinct change in range between two adjacent points. Using edges to recognise a doorway requires understanding

where an ‘opening’ exists between two edges and checking whether it fits the conditions for a doorway. The conditions for a doorway were defined as follows:

1) A doorway would have two edges, where one edge is ‘rising’ and the other ‘falling’:
    a. A rising edge is defined as a positive difference between the current range and the next.
    b. A falling edge is defined as a negative difference between the current range and the previous.
2) No other obstacle would be present in the triangular area formed by the two doorway edges and the wheelchair.
3) The doorway being detected would be of a nominated width and would only be 1.5x larger than the nominated width as standard path planning algorithms would be capable of traversing a space that wide. The nominated width would be larger than the width of the wheelchair.
4) A standard doorway is more likely to be a similar width to the wheelchair.
5) The algorithm to detect a doorway using LiDAR output would be called upon by image recognition of camera feed, so it is safe to assume that the doorway would be close to the wheelchair.

4.3.1 Finding edges in LiDAR output

Doorways in the LiDAR output are distinguishable by looking for a sudden changes in range that leaves a gap between two regions of consistent range readings, noticeable when plotted against the angle of the LiDAR measurement as in Figure 9.

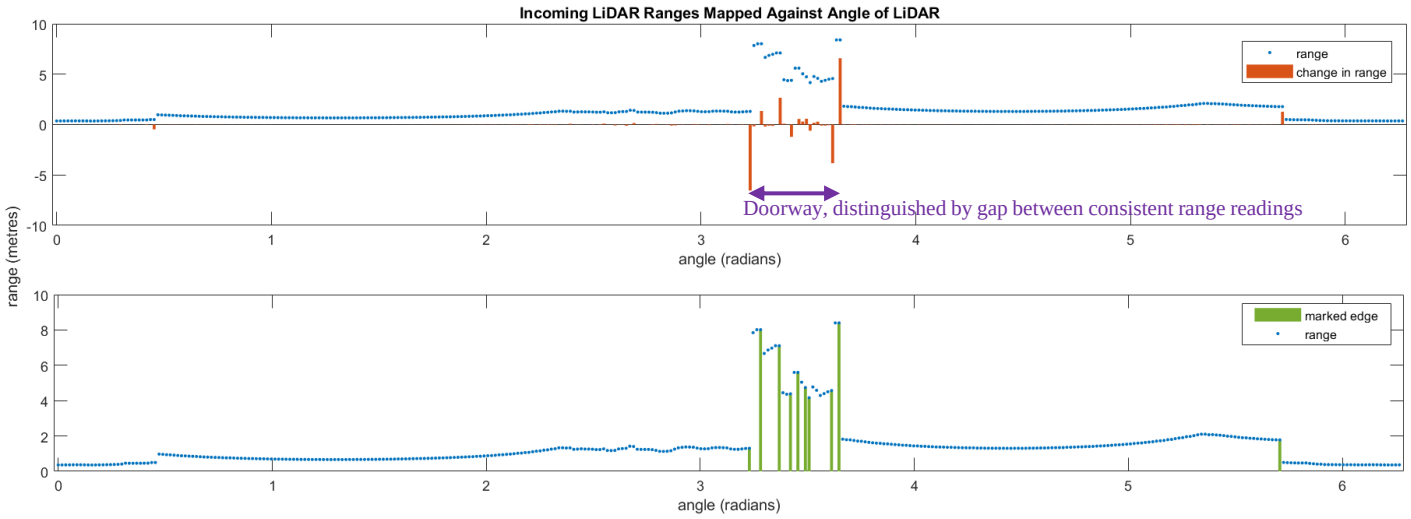


Figure 9: Finding Doorway Boundary by searching for Changes in Range above a specified Threshold of 0.5 metres

To find all potential doorways in the LiDAR range data, the difference in range between one point and the next were taken and used to determine changes in range that were greater than a given threshold. The data was iterated through to find and store the locations of all data points that started with a falling edge, then storing this point with all rising edges beyond that met the threshold and had no obstacles located between the two edges.

Obstacles were found by taking the range with the largest distance, then comparing all ranges between the stored points to see if obstacles were present in the LiDAR output. The result was a list containing pairs of edges representing possible doorways in the LiDAR output.

To filter down the number of potential doorways, each point was converted into x,y coordinates and the width of the potential doorway calculated. If the width was less than the width of the wheelchair then it was discarded, as were doorways with widths greater than 150% of the wheelchair width. Using the same LiDAR output from Figure 9 produced the set of potential doorways in Figure 10 on the left side.

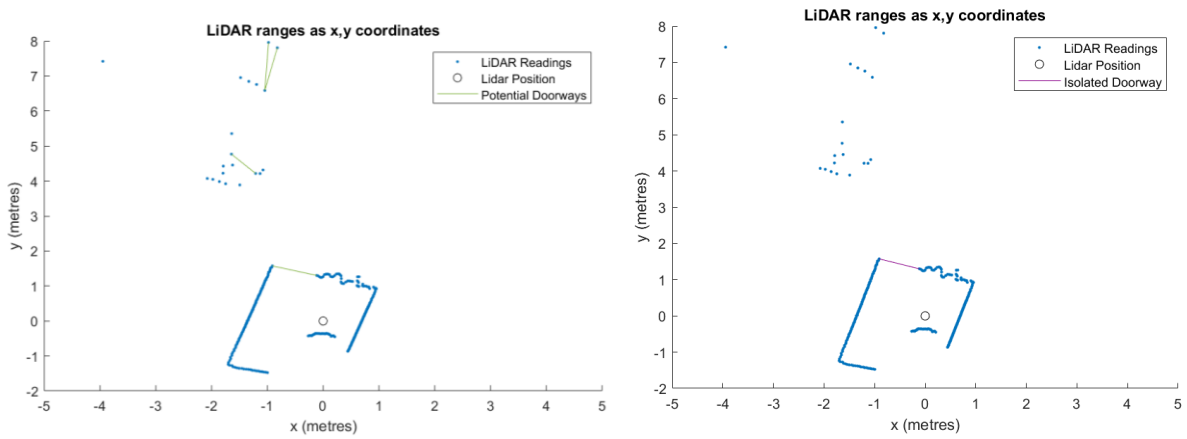


Figure 10: Potential Doorways found using the detector (left) and Isolated Doorway after ranking (right)

4.3.2 Ranking Potential for a Doorway

Doorway likelihood was ranked based on assumptions 4 and 5 that the door is most likely to be near the wheelchair and of a similar width to the wheelchair. The ranking system was designed as follows, where the potential doorway with the largest ranking is chosen as the navigation goal:

$let\ w_d\ be\ width\ of\ doorway$
$let\ w_c\ be\ width\ of\ the\ wheelchair$
$let\ r_d\ be\ distance\ of\ doorway\ centre\ from\ the\ wheelchair$
$let\ k_d\ be\ gain\ for\ distance\ ranking$
$let\ k_w\ be\ gain\ for\ width\ ranking$

$$let\ distance\ rank\ be\ L_d, width\ rank\ be\ L_w$$
$$L_w = (w_d - w_c) * k_d$$
$$L_d = \frac{k_d}{k_w}$$
$$rank = (L_w + L_d)^{-1}$$

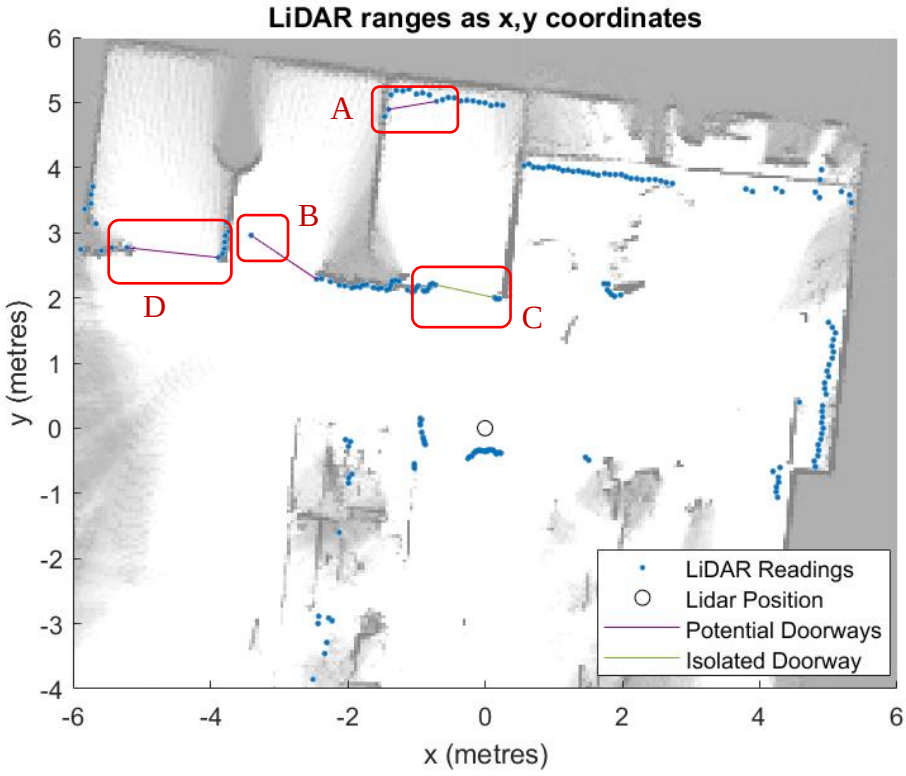
This ranking system punishes doorways for being wider than the wheelchair width and for being far away from the chair. The gains in place are used to balance the weighting of width and distance ranks, allowing control over the process of selecting the doorway. The result was the isolated doorway seen on the graph on the right in Figure 10.

4.3.3 Doorway Detector Performance and Adjustments

Testing the doorway detection functionality in more diverse locations showed promise for use in larger rooms containing more details. The wheelchair was taken through a lab environment containing many features that were partially undetectable by the LiDAR and could provide false positive detections for the doorway detector.

The detector was able to distinguish where doorways were majority of the time, one case example shown in Figure 11 where three doorways were detected and a fourth false positive

detected at the corner of a small room. Due to noise present in LiDAR measurements, data did not always represent the true details in the room. Sometimes points were detected where no data existed or weak laser reflections returning as an infinite result, leaving regions with edges for the detector to trigger on though no edge existed in the real world. This allowed corners of rooms to be recognised as doorways and occasionally doorways were detected with offsets due to false readings on obstacles that do not exist.



A – False doorway detection due to two 'inf' detections where returning laser signal was too weak to be detected by LiDAR. Resulted in a falling and rising edge at just the right conditions to be detected as a doorway.

B – Doorway with offset due to detection of obstacle that did not exist (noise).

C & D – Doorways detected accurately by the detector. C was selected as most likely doorway by ranker.

Figure 11: Doorway Detection results overlaid onto 2d map of the room

To remove false doorways being detected between two readings with 'infinite' values, the detector was changed to use the previous detection which held a decimal range when searching for the initial rising edge of a doorway. Passing the same range data as in Figure 11 resulted in detection 'A' being removed from the data due to removing one of the false edges caused by the presence of two infinite edges near one another.

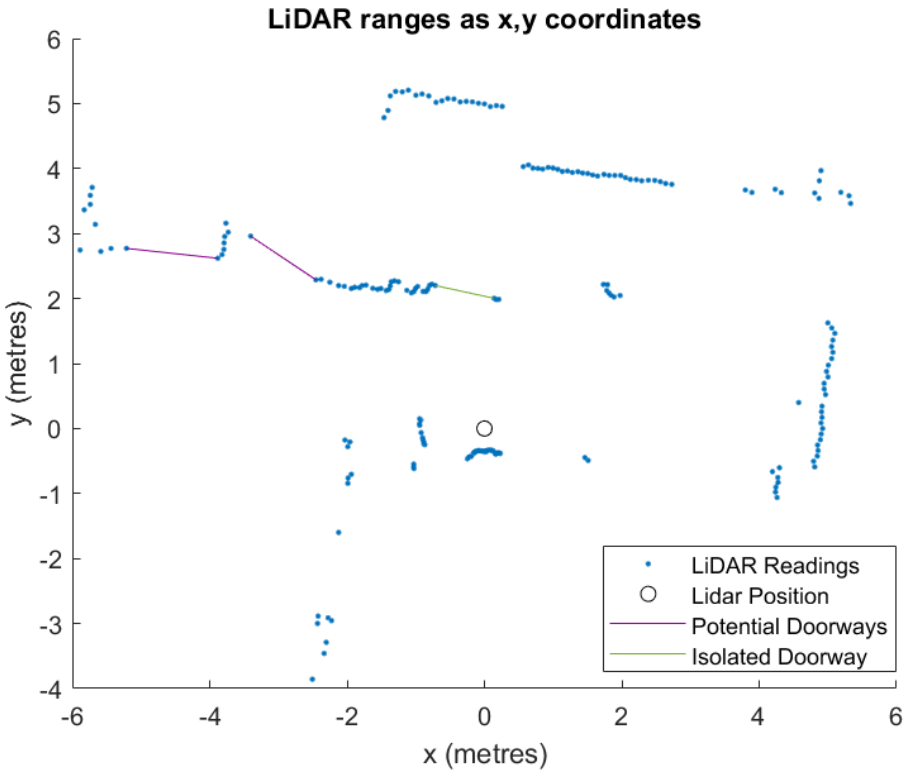


Figure 12: doorway detection results using data from previous figure, preventing infinite readings being triggered as rising edges

Preventing infinite readings being read as rising edges is beneficial for navigating between rooms where the LiDAR is no more than 12 metres from a wall visible beyond the doorway to provide a dataset where edges can be extracted from. For cases where the doorway has only infinite detections between the edges of the doorway, such as a doorway leading into a large open room, the doorway will now longer be detectable.

4.3.4 Velocity Controller

The motor on the wheelchair is controllable by sending velocity commands through ROS. Velocity commands are read by an Arduino micro-controller and converted into a signal the motor can understand.

The navigation goal for the wheelchair is defined by the location of isolated doorway. The x and y coordinates are found for the centre of the doorway and this used to determine the forward and angular velocity to be sent to the chair. An initial velocity controller was constructed using the distance from the doorway in the y axis as forward velocity and the distance in the x axis as the angular velocity. The maximum forward velocity of the chair was found to occur when a value of ± 2 was sent using the velocity command for linear movement along the x axis,

where positive values responded to forward movement and negative values responded to movement backwards. Maximum angular velocity occurred when a value of ± 2.8 was sent using velocity command for angular movement on the z axis.

The velocity controller was setup with conditions to modify performance under different conditions after testing and adjusting the controller output over a series of runs, with simulations initially and then with the real-world wheelchair. The logic flow of the final velocity controller is presented in Figure 13.

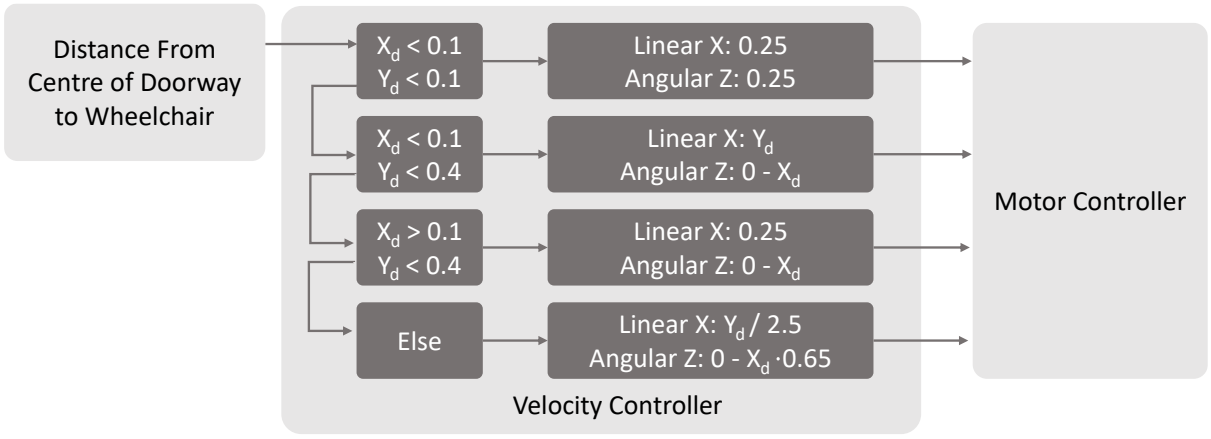


Figure 13: Velocity Controller Logic for Doorway Navigator

4.3.5 Navigation Performance and Adjustments

Tests using the initial version of the velocity controller to navigate the chair through doorways were not compatible with real-world conditions. It was discovered that the simulated LiDAR was producing range data up to ten times faster than the physical LiDAR. Additionally, there was notable lag between velocity commands being produced and the motor controller on the wheelchair moving the wheels. The lag was in the range of 1-2 seconds which provided issues alongside the slower feed of range data from the LiDAR. The result was a system that would overcompensate itself by turning heavily and the path of the chair up to the doorway was near constantly sinusoidal in nature with no evidence of the system settling.

The system needed to be slowed down to allow the velocity controller enough time to dampen oscillation and allow the wheelchair to cross through the centre of the doorway without colliding with the doorway frame. This was achieved by reducing the maximum velocity of the wheelchair when moving forward, slowing the overall time taken for the chair to reach the doorway. This version of the velocity controller allowed for navigation through doorways, so

long as the wheelchair was not on too great an angle to the doorway opening as with the case in Figure 14. The velocity controller in Figure 14 is underdamped, evident by the mild oscillation occurring on the x axis distance from the wheelchair which begins to be compensated for by increasing the turn velocity of the wheelchair as the wheelchair moves forward and never stops turning inwards until the system has compensated too much at roughly the 90 second mark.

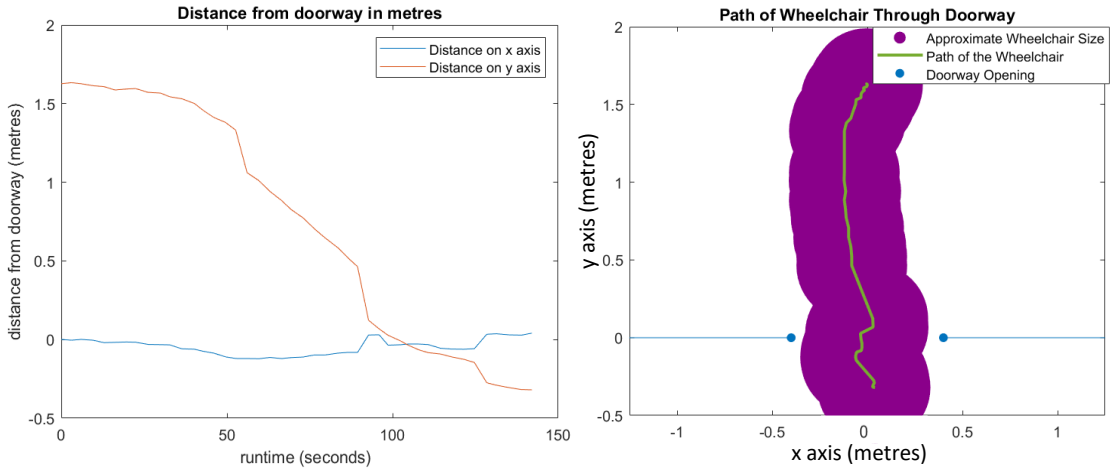


Figure 14: Results of Autonomous Wheelchair Navigation through Doorways with small misalignment from doorway

To allow the wheelchair navigation through doorways under conditions where the chair was more heavily misaligned with the doorway than in Figure 14, the x axis compensation was increased by allowing the controller to send faster angular velocity commands. The result was a controller that autonomously guided the wheelchair through doorways even when misaligned from the door so long as the wheelchair was within 1 metre of the centre of the doorway along the x axis and at least 0.8m away from the door on the y axis. A set of results are shown in Figure 15 depicting the path of the wheelchair through a doorway when more heavily misaligned.

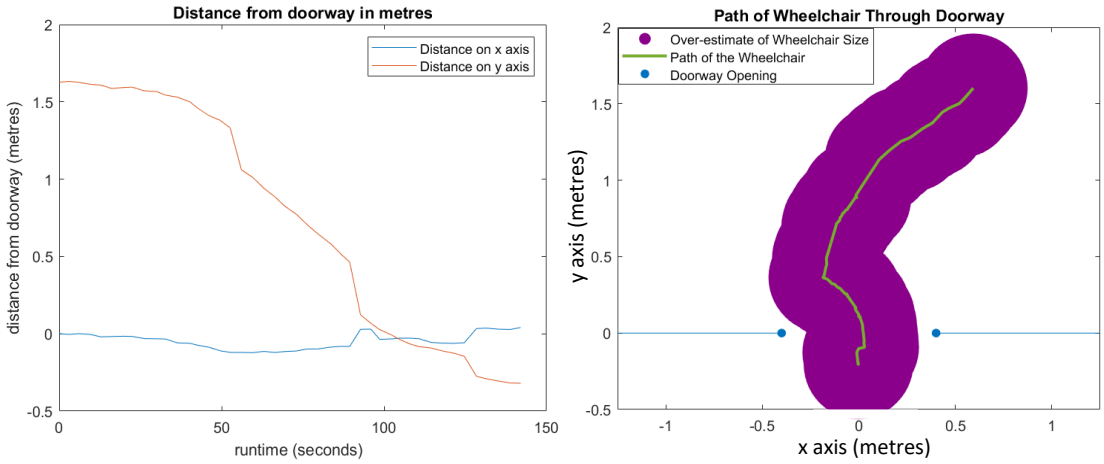


Figure 15: Results of Autonomous Wheelchair Navigation through Doorway Misaligned with Wheelchair

Sharp turns and sudden stops were seen to occur when navigating. This occurred due to the motor suddenly stopping when too great a change in wheel rotation commands were sent to the motor, or when the front swivel wheels on the chair are not aligned appropriately. Under these conditions it would take 1-3 seconds for the motor to resume movement or for the front wheels to be pushed back into alignment from the force of the motor. These occurrences were seen to recur randomly when the chair was navigated both autonomously and manually and was the cause of sharp changes in the wheelchair direction present in results.

The produced navigator does suit the requirements of navigating narrow doorways without collision through the aid of the doorway detector that provides input to the velocity controller which guides the chair towards the doorway and corrects the system to allow the chair to pass through the centre of the opening without touching the door frame.

4.4 PUPIL DETECTION FOR GAZE TRACKING

Gaze tracking can provide an alternative form of input for individuals whom are incapable of controlling the chair via the joystick controller. Setting up gaze tracking alongside virtual controllers laid out on a video display can be used to control wheelchair movements as well as offer further functionality to control the environment should the chair be capable of connecting with devices outside the internal environment, such as automated homes or mobile phones.

Testing the application of web cameras installed on most laptops for the use as a gaze tracker, the OpenCV library was utilised to generate python scripts that could detect the location of pupils from webcam feeds.

4.4.1 Removing unnecessary features from images

Processing an image to determine the location of pupils required the process of first discovering the face, then eyes before being able to apply blob or circle detectors to find pupils. Lighting changes and unique facial attributes can confuse detectors by presenting as pupil-like features after transformations are applied to the image, confusing detectors and reducing success rates of finding pupils within images.

To reduce an input image down to the region containing face and then eyes, cascade classifiers are employed. The cascade classifiers are pretrained machine learning algorithms available open source for use with OpenCV. Using the face cascade classifier, faces are found in the image and returned as locations. For the purpose of finding the pupil of a user seated in front of a webcam it is assumed that the largest face region would be where the pupils are positioned, so only the largest detected face was kept when searching the image. Then within the face region an eye cascade classifier was used to locate the positions of the eyes. Occasionally the chin or nose was detected as an eye so an assumption was made regarding where eyes would be located within the face:

- If the face is separated into two halves, bottom and top, then eyes will only ever be present on the top half of the face.

Using this assumption, all eyes detected with pixels on the lower half of the face were ignored, allowing accurate detection of eyes from an input image as in Figure 16 below.

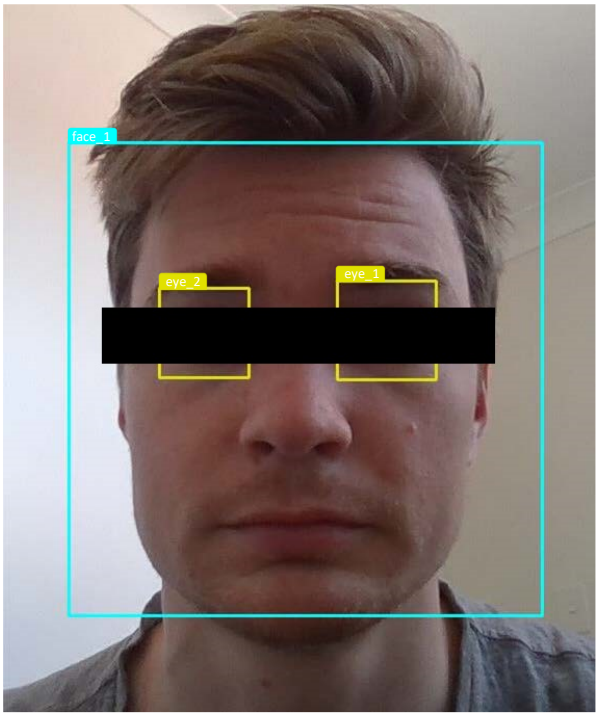


Figure 16: face and eye detection using cascade classifiers with OpenCV

The regions found by the eye cascade classifier contained quite a bit of space above and below the eyes. To remove features such as eyebrows and freckles present in the upper and lower regions, a set of 30 images of random individuals from a google photos search of 'person face' were passed through the cascade classifiers and used to define the portions of unnecessary space within the returned regions. Each image was measured individually to determine the how many pixels in height the unnecessary portions above and below the eye were, then was divided by the total height of the region to find percentages that can be used to remove features regardless of region size.

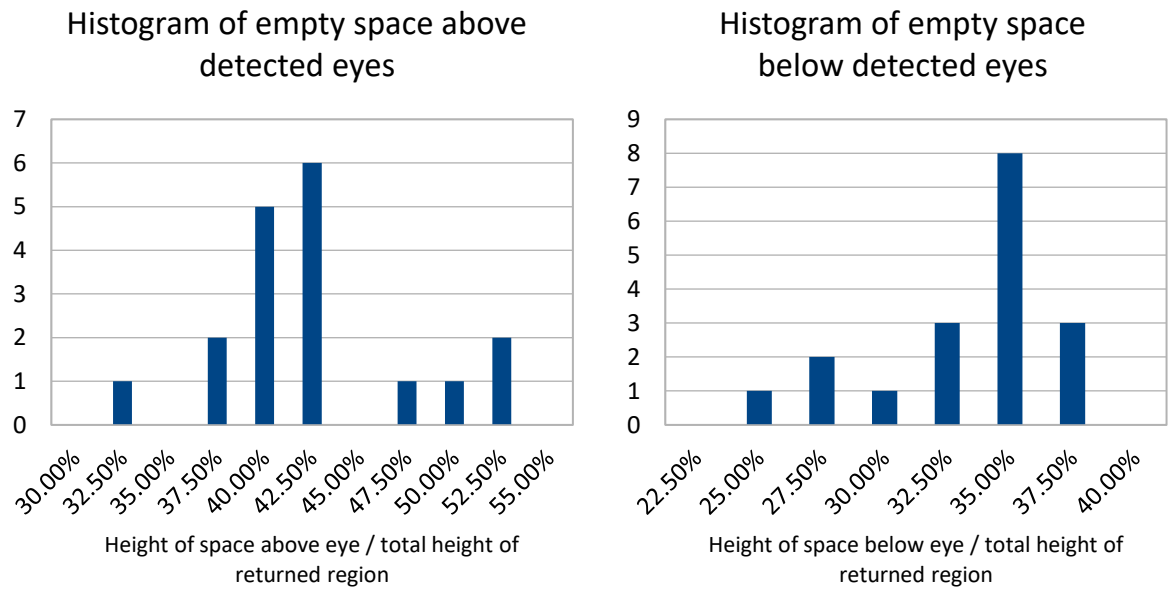


Figure 17: Histograms of the percentage of unnecessary space above and below the eye within returned eye regions

Table 2: Results of measuring 30 images to determine percentages of unnecessary space above/below returned eye region

Unnecessary Space	Std Dev	Smallest Bin	(Smallest Bin - Std Dev)
Above eye	4.47%	30.00%	25.53%
Below eye	3.25%	22.50%	19.25%

The data in Table 2 was calculated based on the percentage of images containing unnecessary space. The smallest bin minus the standard deviation of the data was used as a guide for the largest amount of trimmable space within the returned regions of the eye. To remove as much unnecessary space as possible, the regions were manipulated to crop the top 25% and the bottom 20%. These levels left eyes intact whilst removing the eyebrows and most facial features below the eye.

4.4.2 Pupil Isolation Method 1: Morphological Operations and Blob Detection

The OpenCV library has an inbuilt function for detecting dots or 'blobs' within images called the 'blob detector'. Using thresholding to produce a binary representation of an image combined with morphological operations, it is possible to convert an image of an eye to a black circle where the pupil is located surrounded by a series of lines and blocks where features such as eyelids, hair are present.

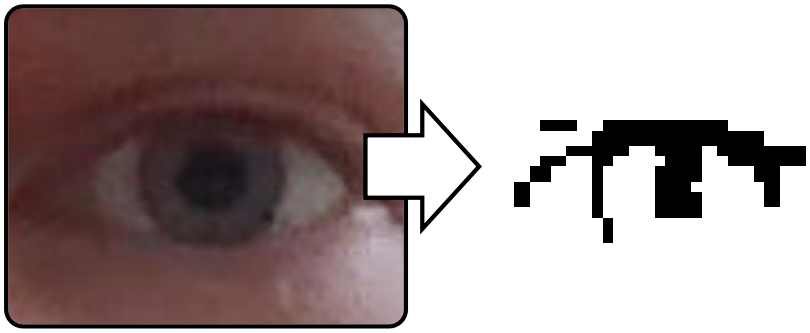


Figure 18: Binary Conversion of RGB image of eye, preparing for detection of pupil

4.4.2.1 Removing unnecessary features

Preparing the eye for blob detection, the eye was converted from colour to greyscale. The OpenCV library has functions for converting colour formats of images as well as reducing to binary representation of pixels using thresholds. The greyscale eye was passed through a histogram equaliser to enhance the contrast of the image for better visual analysis, as lighting can often be poor using a standard webcam.

The greyscale image is converted to binary representation (white and black pixels) using thresholding, and then filtered using morphological operations to assist with removing horizontal lines from the image to purge features like eyelid lines and hairs.

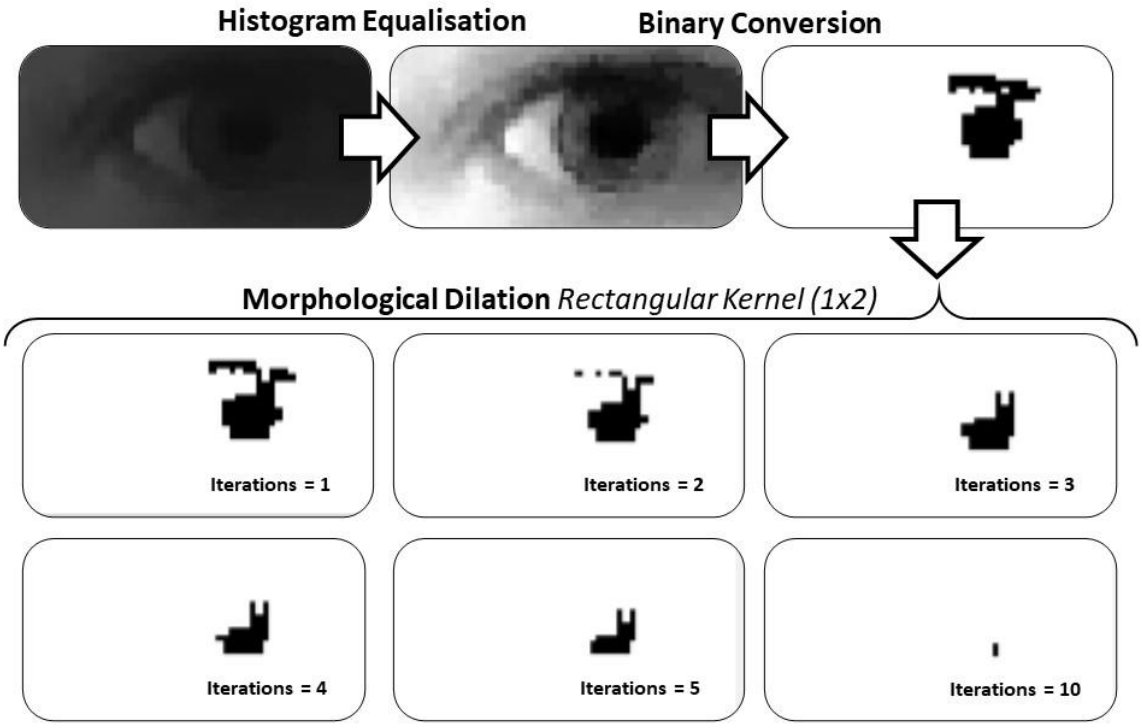


Figure 19: Reducing image of eye down to binary representation of the pupil

Performing dilation on the image with a rectangular, one by two kernel removes features with horizontal lines. Repeating the operation further reduces the presence of features with horizontal lines. This process leaves a shape representing the area where the pupil is located, as depicted in Figure 19, showing the effect repeated dilation operations with a rectangular kernel on a binary representation of an eye.

4.4.2.2 Blob detector to find pupil coordinates

Passing the processed binary image of the eye through the blob detector, the coordinates of any detected blobs are returned. Circles are drawn around the coordinates of the detected blobs on the image, as seen in Figure 20 demonstrating the output of the python application written to utilise the OpenCV library for detecting pupils using morphological operations and the blob detector.

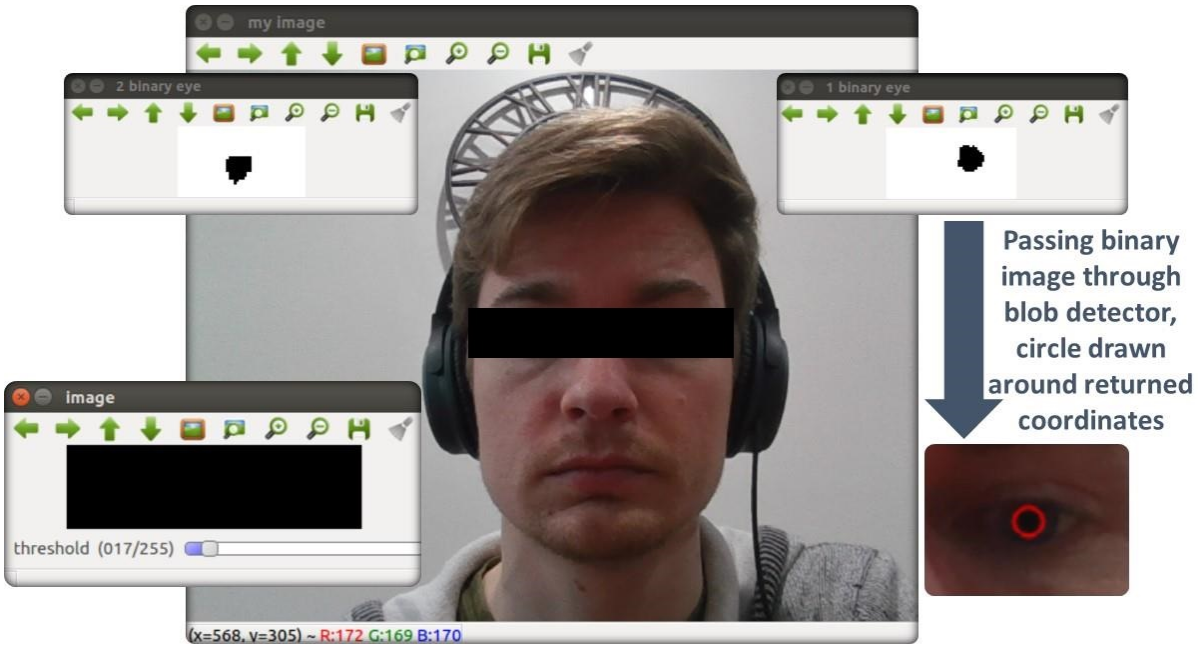


Figure 20: Using OpenCV blob detector to find location of pupil and draw circle over returned coordinates

4.4.3 Pupil Isolation Method 2: Edge Detection and Circle Detection with Hough Transform

To better detect the coordinates of pupils from the webcam feed, the Hough Transform was utilised within OpenCV. OpenCV contains functions employing the Hough Transform for feature detection, such as detecting circles in greyscale images. Although the OpenCV applications for circle detection with the Hough Transform is compatible with greyscale images, to remove the presence of circular features other than the pupil the Canny edge detector is utilised to convert the eye to a binary representation of edges within the image.

4.4.3.1 Preparing Image by Finding Edges

To prepare the image for detecting circles through the Hough Transform, a Canny edge detector was employed to find the edges of identified eyes. The OpenCV library contains a Canny edge detector built in that accepts a greyscale image and returns a binary image based on an upper and lower threshold portraying the edges detected within the greyscale image. Refer to Figure 21 for example of the edge detector output.

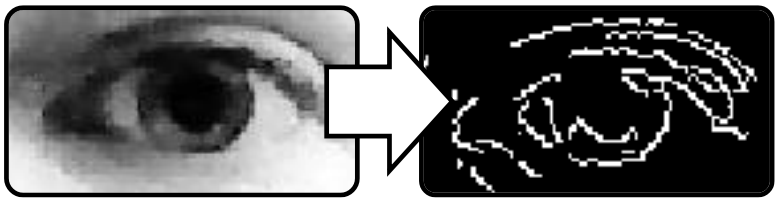


Figure 21: Edge Detection using Canny Edge Detector within OpenCV

The upper and lower threshold values are highly dependent on the lighting conditions within the image. If the thresholds are setup for a webcam capturing images of a well-lit environment, the system will not produce the same results for an environment with poor lighting. Edges may be poorly translated and result in pupils being undetectable as the contrast between the pupil, iris and/or eyelid may be too poor for the detector to distinguish edges between. To ease this issue the colouring in the greyscale image is more evenly dispersed through histogram equalisation, reducing the difference in threshold values between poor and well-lit environments.

4.4.3.2 Detecting circles from edges

Parsing the edge representation of the eye through the 'HoughCircles()' OpenCV function, circular features within the image are identified and their coordinates returned. The function uses the Hough Transform to search for circular features, allowing specification of minimum distance between features as well as a range for the radius of circles. Other options allow for further tuning by changing the upper threshold used for the greyscale image and adjusting how exact features need to match.

Once coordinates are returned, circles are drawn onto the image to indicate where pupils have been detected. Figure 22 demonstrates the output of the python application written to utilise the OpenCV library for detecting pupils using edge detection and circular feature detection.

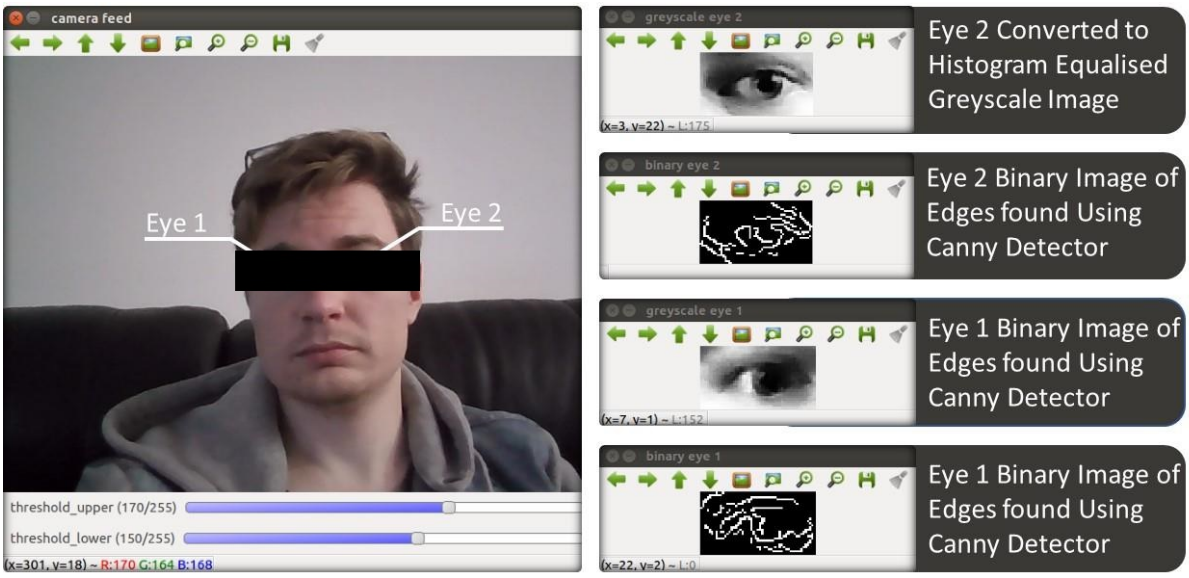


Figure 22: Pupil Detection using Hough Transform to find circles in image

4.4.4 Comparison of pupil detectors

The detectors from Method 1 and Method 2 were both able to detect pupils from a webcam feed. The detection rate and average error in pixels were analysed to understand the performance of the two detectors. These results are shown in Figure 23.

Key		
Eye Looking At:		
Top Left of Screen	Top of Screen	Top Right of Screen
Left of Screen	Centre of Screen	Right of Screen
Bottom Left of Screen	Bottom of Screen	Bottom Right of Screen

Method 1 Average Error (rounded to nearest pixel)					
Left Eye			Right Eye		
1	1	1	N/A	0	0
0	0	1	0	0	0
0	0	N/A	0	0	0

Method 1 Detection Rate					
Left Eye			Right Eye		
20%	25%	70%	0%	20%	45%
15%	25%	70%	10%	10%	60%
10%	10%	0%	35%	55%	20%

Method 2 Average Error (rounded to nearest pixel)					
Left Eye			Right Eye		
13	4	1	3	7	6
12	6	7	2	7	11
19	8	15	15	19	16

Method 2 Detection Rate					
Left Eye			Right Eye		
100%	100%	100%	100%	100%	100%
100%	100%	100%	100%	100%	100%
100%	100%	100%	100%	100%	100%

Figure 23: Average Error (in pixels, average pupil radius was 6 pixels) of Returned Pupil Coordinates and Average Detection Rate for Methods 1 and 2 when looking at specific sections of the computer display

Comparing the two pupil detectors, method 1 is considerably more accurate than method 2 in terms of average error. Method 1 consistently had an error of 1 or less pixels when looking at majority of the computer display, suggesting that pupil distance from the centre of the camera does not have a noticeable effect on average error. Method 2 results suggest that as pupils drift further from the camera the average error increases, where the left pupil is closer to the camera centre when looking at the top right of the screen and the right pupil is closer to the camera centre when looking at the top left of the screen.

When comparing detection rates, Method 2 was most reliable as it always returned coordinates unlike Method 1 with eyes being undetectable when looking at two regions of the display. While Method 1 is the most accurate, near always locating the pupils within a 1-pixel error under ideal lighting conditions, method 2 provides better feedback.

If applied to gaze tracking now, method 1 would likely pinpoint pupil location but responsiveness would be slow and not capable of using both pupils at every portion of the display to determine where a user is looking. Method 2 would be more jittery but have faster response rate, the trade-off being responsiveness for accuracy.

4.5 VOICE CONTROL USING THE MYCROFT AGENT

Voice control was tested for controlling wheelchair movement. This was executed by writing a simple script that would be triggered by the MyCroft agent when keywords in Table 3 were detected and send velocity commands to the motor controller over ROS. To reduce the occurrence of unwanted commands being sent, the MyCroft agent was chosen as it allowed setting of a custom hot word that would need to be spoken and detected before the agent began listening for new commands. The hot word was kept as ‘Hey MyCroft’ for tests.

Scripts for controlling the chair movements in a simulated environment were successfully activated by the MyCroft agent when spoken commands were recognised, given that the room was quiet. Testing the agent in a room with music playing, as the surrounding noise increased in volume the agent had trouble recognising spoken commands using the microphone built into the wheelchair computer. Switching over to a standard microphone in a headset, the agent could recognise commands better in noisy environments, but commands were still often missed or incorrectly translated.

Table 3: Speech Recognition - Keywords/Phrases and Associated Triggers

Keyword or Phrase	Spoken Response	Triggered Action
"Forward" or *"Move"*	*"Moving forward"*	Run script to send velocity command to move chair forwards
"How are you"	*"I'm doing well"* or *"Pretty well"* or *"I'm doing very well"*	No triggered action

The MyCroft agent can be programmed to recognise multiple keywords or phrases for a single action and/or response, as well as cycle through multiple responses to provide more natural conversation flow. When asked *"How are you"* the agent would respond with one of three responses. This can be utilised to provide instructions to the user on how to use the agent.

5 Limitations

Technology restraints and environmental conditions have resulted in limited success for aspects of the ABC wheelchair. Due to the quarantine restrictions in the presence of COVID-19 it became difficult to obtain users to test the operation of the pupil detector and speech recognition.

The LiDAR utilised has a view of 360 degrees, however this complete vision of the environment was blocked by the user sitting in the chair and partially by the chair itself. The LiDAR hat to be positioned in a location where it would be able to detect obstacles around the room to generate maps of rooms. If the LiDAR had been positioned above the head of the user then full range of measurements would be available however many features such as tables and chairs would be lost.

Processing power was limited on the wheelchair computer and real-time processes had to be executed relatively quickly to keep the system responses up to date. This posed issues when developing the edge based doorway navigator. Ideally the edges would be detected by converting the LiDAR readings into a 600 by 600 grid of cells representing an area of $3cm^2$ each to process for doorway detection. The grid could be treated as a picture and an edge detector used to find and pair edges in two dimensions to find potential doorways, which would provide a more thorough search. This process exponentially increases the calculations however based on the number of cells and required lengthy execution times that were not suitable to control the system. As a result, the one-dimensional solution was applied to reduce the execution time.

Real-time control of navigational control was hindered by the delay between velocity commands being sent to the motor before being executed. The delay was often larger than one second which could allow the chair to collide with obstacles before it was able to correct itself. This posed an issue implementing the real-time doorway navigator as extra caution had to be used to keep the chair from travelling too fast towards walls and doorways to prevent collisions before correction could be employed.

As the wheels on the front of the chair were connected on freely rotating hinges, the intended movement of the chair was not always executed when the front wheels were not aligned and had to be forced into place by the force of the motor on the rear wheels. This resulted in sharp turns or generally unwanted movements which caused issues when navigating through narrow

spaces such as doorways and halls. The resulted in the wheelchair needing to be slowed down when nearing narrow spaces to allow the system more time to correct itself and prevent collisions.

Testing of pupil isolators for the future development of a gaze tracking system showed that both isolators were highly dependent on lighting conditions. The system was highly dependent on external lighting to operate properly and required retuning when lighting conditions changed. The system is not useable in dark environments and if being used as a method for gaze tracking would require either a light built into the chair to illuminate the face of the user or utilise an IR camera that could operate in both poor and well lit environments.

The MyCroft agent was only capable of interpreting spoken commands when connected to the internet. To allow for offline speech recognition a local interpreter would have to be installed. The deepnet neural network the agent uses is available to be run locally on a machine however the processing power required slowed system resources down and hindered the execution times of real-time systems on the machine when ran simultaneously. As a result the agent was kept to do processing through the online server and required internet access to operate.

6 Improvements

The processing power on the chair computer provided limitations for the demand that navigational resources could utilise. By providing more processing power the edge based navigator could be processed within a 2 dimensional context and allow for thorough search algorithms to determine the locations of doorways. The range data could be treated as an image processing problem to extract points where edges end and fill in details of small gaps more accurately as well as utilising median filtering or similar techniques that preserve edges in images whilst removing salt and pepper noise present from false detections of the LiDAR.

The SLAM software can be improved by providing additional sensor data that could interpret range data in a 3d context, whether that be the integration of sonar or 3d laser finding from the Kinect Camera. Combining this with the LiDAR would allow for detection of obstacles outside the LiDAR range and may remove the need for angling the LiDAR, allowing for maps to be produced with level of detail output by the level LiDAR and providing the necessary obstacle detection for objects outside the LiDAR range. The Kinect Camera would be a good choice for this as it can provide 3d range data in 120 degrees of view in front of the chair to provide higher resolution data on obstacles in the environment as opposed to the single beam sonar ranges.

The gaze tracking requires further research to first provide accurate pupil detection of both pupils when looking at the entire range of the computer display. This could be achieved by combining the data from method 1 and 2 using a particle filter or similar. Coordinates from method 1 of pupils would supply a near guaranteed location that could set all particles to a single location and then use coordinates from method 2 better estimate the most likely position by carrying over the likelihood from each stage. This method may allow for estimations that find a happy medium between the accuracy of method 1 and the responsiveness of method 2. From this point machine learning may be employed to train a model that can correlate pupil locations with points on the display to achieve a first draft of the gaze tracker.

7 Conclusion

The work done for the wheelchair has provided a basis for future research into an addon system for powered wheelchairs that can tailor for a range of users with disabilities that prevent standard powered chairs from being useable or comfortable to use on their own.

SLAM utilising the cartographer system and LiDAR can produce viable maps of rooms and track the location of the chair as it moves around environments without the need for wheel odometry to support it. The LiDAR can handle being angled downwards towards the front of the wheelchair up to roughly 25 degrees for greater vision of obstacles closer to the ground before behaviour of the SLAM system begins to poorly track location and map rooms. The system has room for improvement by incorporating additional sensors to feed information on the environment outside the LiDAR sensing range and improve system visibility.

The doorway navigator can successfully navigate narrow doorways without collision with the door frames thanks to the velocity controller. The controller successfully corrects the wheelchair movement taking input from the edge based doorway detector based on range data received by the LiDAR to search for doorways in an environment. The navigator can operate in real-time and is capable of correcting the path of the wheelchair towards the door when not aligned with the doorway. There is space here for triggering the navigator when doorways are detected using image recognition based on the Kinect Camera feed, as well as the potential to produce a more robust detector using a two dimensional approach as mention in the improvements section.

The pupil detectors still require further work before a gaze tracking system can be finished. The detectors on their own cannot accurately detect the position of pupils as the user looks around every extremity of the computer display. Further work needs to be done to either combine the two systems as discussed previously or find an alternative method that is more ideal whilst still operating on ‘budget’ technology to reduce the cost of addon system for powered wheelchairs.

The MyCroft agent proved to be a useful tool as an input alternative to the joystick. More work will be required to develop full control of the wheelchair using the voice assistant as only basic testing was performed to prove that the technology could be used. Moving forward it may be necessary to move the speech recognition to be performed locally. This will benefit the user if

they are attempting to operate the wheelchair in environments where network connectivity is not available.

The system is progressing towards an accessible power wheelchair addon that can benefit individuals with disabilities and impairments. With research into brain control interfaces being developed by other researchers and the branches opened here on gaze tracking and speech control for the ABC Wheelchair, it has the potential to become a system that can suit a diverse range of users. When selecting technology to aid individuals it is important to consider what is comfortable, suitable and viable for the individual. By offering a powered chair with manual, assistive and autonomous systems in place that can be controlled via multiple input methods, it provides a greater audience of users the ability to attain an unknown level of freedom that was otherwise inaccessible to them and offer greater quality of life for the future.

8 References

[1] L. Sminkey, "New world report shows more than 1 billion people with disabilities face substantial barriers in their daily lives," ed: World Health Organization, 2011.

[2] K. Rousseau-Harrison and A. Rochette, "Impacts of wheelchair acquisition on children from a person-occupation-environment interactional perspective," *Disability and Rehabilitation: Assistive Technology,* vol. 8, no. 1, pp. 1-10, 2013, doi: 10.3109/17483107.2012.670867.

[3] F. Pasteau, V. K. Narayanan, M. Babel, and F. Chaumette, "A visual servoing approach for autonomous corridor following and doorway passing in a wheelchair," *Robotics and Autonomous Systems,* vol. 75, no. PA, pp. 28-40, 2016, doi: 10.1016/j.robot.2014.10.017.

[4] M. L. Toro, C. Eke, and J. Pearlman, "The impact of the World Health Organization 8-steps in wheelchair service provision in wheelchair users in a less resourced setting: A cohort study in Indonesia Health systems and services in low and middle income settings," *BMC Health Services Research,* vol. 16, no. 1, p. <xocs:firstpage xmlns:xocs=""/>, 2016, doi: 10.1186/s12913-016-1268-y.

[5] S. Shore and S. Juillerat, "The impact of a low cost wheelchair on the quality of life of the disabled in the developing world," *Medical Science Monitor : International Medical Journal of Experimental and Clinical Research,* vol. 18, no. 9, pp. CR533-CR542, 2012, doi: 10.12659/MSM.883348.

[6] R. C. Simpson, "Smart wheelchairs: A literature review," *Journal of rehabilitation research and development,* vol. 42, no. 4, pp. 423-436, 2005, doi: 10.1682/JRRD.2004.08.0101.

[7] A. Murarka, S. Gulati, P. Beeson, and B. Kuipers, "Towards a safe, low-cost, intelligent wheelchair," in *Workshop on Planning, Perception and Navigation for Intelligent Vehicles (PPNIV)*, 2009, pp. 42-50.

[8] R. Simpson, E. LoPresti, S. Hayashi, I. Nourbakhsh, D. J. J. o. R. R. Miller, and Development, "The smart wheelchair component system," vol. 41, 2004.

[9] J. Leaman and L. Hung Manh, "A Comprehensive Review of Smart Wheelchairs: Past, Present, and Future," *IEEE Transactions on Human-Machine Systems,* vol. 47, no. 4, pp. 486-499, 2017, doi: 10.1109/THMS.2017.2706727.

[10] O. Horn, "Smart Wheelchairs: past and current trends," in *2012 1st International Conference on Systems and Computer Science (ICSCS)*, 2012: IEEE, pp. 1-6.

[11] F. Doshi and N. Roy, "Efficient model learning for dialog management," ed, 2007, pp. 65-72.

[12] C. Gao, T. Miller, J. R. Spletzer, I. Hoffman, and T. Panzarella, "Autonomous docking of a smart wheelchair for the automated transport and retrieval system (ATRS)," *Journal of Field Robotics,* vol. 25, no. 4-5, pp. 203-222, 2008.

[13] B. Jenita Amali Rani and A. Umamakeswari, "Electroencephalogram-based Brain Controlled Robotic Wheelchair," *Indian Journal of Science and Technology,* vol. 8, no. S9, p. 188, 2015, doi: 10.17485/ijst/2015/v8iS9/65580.

[14] M. Njah and M. Jallouli, "Fuzzy-EKF Controller for Intelligent Wheelchair Navigation," *Journal of Intelligent Systems,* vol. 25, no. 2, pp. 107-121, 2016, doi: 10.1515/jisys-2014-0139.

[15] E. Wästlund, K. Sponseller, O. Pettersson, and A. Bared, "Evaluating gaze-driven power wheelchair with navigation support for persons with disabilities," *Journal of rehabilitation research and development,* vol. 52, no. 7, pp. 815-826, 2015, doi: 10.1682/JRRD.2014.10.0228.

[16] D. Schwesinger, A. Shariati, C. Montella, and J. Spletzer, "A smart wheelchair ecosystem for autonomous navigation in urban environments," *Autonomous Robots,* vol. 41, no. 3, pp. 519-538, 2017, doi: 10.1007/s10514-016-9549-1.

[17] K. Yousif, A. Bab-Hadiashar, and R. Hoseinnezhad, "An Overview to Visual Odometry and Visual SLAM: Applications to Mobile Robotics," *Intelligent Industrial Systems,* vol. 1, no. 4, pp. 289-311, 2015, doi: 10.1007/s40903-015-0032-7.

[18] G. K. Kalyani, Z. Yang, V. Gandhi, and T. Geng, "Using Robot Operating System (ROS) and Single Board Computer to Control Bioloid Robot Motion," in *Towards Autonomous Robotic Systems*, Cham, Y. Gao, S. Fallah, Y. Jin, and C. Lekakou, Eds., 2017// 2017: Springer International Publishing, pp. 41-50.

[19] "ROS/Introduction." http://wiki.ros.org/ROS/Introduction (accessed 10 December, 2019).

[20] J.-c. Ma, Q. Zhang, and L.-y. Ma, "A novel robust approach for SLAM of mobile robot.(Report)," *Journal of Central South University: Science & Technology of Mining and Metallurgy,* vol. 21, no. 6, p. 2208, 2014, doi: 10.1007/s11771-014-2172-4.

[21] R. Valencia, M. Morta, J. Andrade-Cetto, and J. M. Porta, "Planning Reliable Paths With Pose SLAM," *IEEE Transactions on Robotics,* vol. 29, no. 4, pp. 1050-1059, 2013, doi: 10.1109/TRO.2013.2257577.

[22] J. Cobos, L. Pacheco, X. Cufi, and D. Caballero, "Integrating visual odometry and dead-reckoning for robot localization and obstacle detection," vol. 1, ed, 2010, pp. 1-6.

[23] mycroft. "https://mycroft-ai.gitbook.io/docs/skill-development/introduction/your-first-skill." (accessed 3 January, 2020).

[24] mycroft. "https://mycroft-ai.gitbook.io/docs/using-mycroft-ai/customizations/stt-engine." (accessed 3 January, 2020).

[25] S. Cristina and K. P. Camilleri, "Unobtrusive and pervasive video-based eye-gaze tracking," *Image and Vision Computing,* vol. 74, pp. 21-40, 2018, doi: 10.1016/j.imavis.2018.04.002.

[26] L. Xia, B. Sheng, W. Wu, L. Ma, and P. Li, "Accurate gaze tracking from single camera using gabor corner detector," *An International Journal,* vol. 75, no. 1, pp. 221-239, 2016, doi: 10.1007/s11042-014-2288-4.

[27] R. S. Remmel, "An Inexpensive Eye Movement Monitor Using the Scleral Search Coil Technique," *IEEE Transactions on Biomedical Engineering,* vol. BME-31, no. 4, pp. 388-390, 1984, doi: 10.1109/TBME.1984.325352.

[28] D. Iacoviello, M. Lucchetti, G. Calcagnini, and F. Censi, "Pupil edge detection and morphological identification from blurred noisy images," vol. 1, ed, 2003, pp. 922-925 Vol.1.

[29] OpenCV. "About." https://opencv.org/about/ (accessed 23 March, 2020).

[30] Z. Zhu and Y. Cheng, "Application of attitude tracking algorithm for face recognition based on OpenCV in the intelligent door lock," *Computer Communications,* vol. 154, pp. 390-397, 2020, doi: 10.1016/j.comcom.2020.02.003.

[31] K. Pulli, A. Baksheev, K. Kornyakov, and V. Eruhimov, "Real-time computer vision with OpenCV," *Communications of the ACM,* vol. 55, no. 6, pp. 61-69, 2012, doi: 10.1145/2184319.2184337.

[32] D. Choi, Y. Ryu, Y. Lee, and M. Lee, "Performance evaluation of a motor-imagery-based EEG-Brain computer interface using a combined cue with heterogeneous training data in BCI-Naive subjects," *Biomedical engineering online,* vol. 10, no. 1, p. 91, 2011, doi: 10.1186/1475-925X-10-91.

[33] H. A. Lamti, M. M. Ben Khelifa, P. Gorce, and A. M. Alimi, "A brain and gaze-controlled wheelchair," *Computer Methods in Biomechanics and Biomedical Engineering,* vol. 16, no. sup1, pp. 128-129, 2013, doi: 10.1080/10255842.2013.815940.

[34] B. Rebsamen *et al.*, "A Brain Controlled Wheelchair to Navigate in Familiar Environments," *IEEE Transactions on Neural Systems and Rehabilitation Engineering,* vol. 18, no. 6, pp. 590-598, 2010, doi: 10.1109/TNSRE.2010.2049862.

[35] T. C. Authors. "Compiling Cartographer Ros." https://google-cartographer-ros.readthedocs.io/en/latest/compilation.html (accessed 13 November, 2019).

[36] S. Filanov. "Tracking your eyes with Python." https://medium.com/@stepanfilonov/tracking-your-eyes-with-python-3952e66194a6 (accessed 18 February, 2020).

[37] M. Wirth. "An example of the hough transform - pupil segmentation." https://craftofcoding.wordpress.com/2017/12/06/an-example-of-the-hough-transform-pupil-segmentation/ (accessed 14 March, 2020).

9 Appendix A

9.1 CODE ASSOCIATED WITH DOORWAY NAVIGATION

Code associated with edge based doorway navigation from range data.

9.1.1 controller.py

```
#! /usr/bin/env python2

import rospy
import datetime
import doorNav
import numpy as np
import time
from sensor_msgs.msg import LaserScan
from geometry_msgs.msg import Twist
import matplotlib.pyplot as plt
import lidarProcessor as lpr

# redundant variables
# front = 0
# side = 0
# bufferFront = 0
# bufferFront = 0

pub=rospy.Publisher("cmd_vel",Twist,queue_size=1)
speed=Twist()
forward=0
side=0
[theta_range_1,theta_range_2, front, side] = doorNav.setup()
global reading
global count
count = 0

global accX
global accY
global yOrient
accX=[]
accY=[]
yOrient = []

plt.ion()

#plot readings for visual of room and doorway from LiDAR
def plotReadings(point_x,point_y,x,y,edge_x,edge_y):
    plt.figure(2)
    plt.plot(x,y,'go',markersize="1.5")
```

```
    plt.plot(edge_x,edge_y,'bo',markersize="6")
    plt.plot(point_x,point_y,'ro',markersize="6")
    plt.axis([-12,12,-12,12])
    plt.pause(0.0001)
    plt.clf()

def callback(msg):
    global accX
    global accY
    global count
    count = count + 1

    measurements = doorNav.segment_measurements(msg)

    reading = doorNav.assign_angle(msg)
    result = lpr.edges(reading,0.1,0.9)

    if (len(result) > 0):
        door = result[0]
        edges = result[1]
        if (count > 40):
            accX.append((door[0][0] + door[0][1])/2)#middle of door in x plane
            accY.append((door[1][0] + door[1][1])/2)#middle of door in y plane
            yOrient.append(door[1][0] - door[1][1])#0 is perpendicular to door
            np.savetxt('accX.csv',accX,delimiter=',')
            np.savetxt('accY.csv',accY,delimiter=',')
            np.savetxt('yOrient.csv',yOrient,delimiter=',')

    else:
        door = [[0,0],[0,0]]
        edges = [0,0]

    goalx = (door[0][0] + door[0][1])/2
    goaly = (door[1][0] + door[1][1])/2

    #condition 1
    if (goaly < 0.1 and goalx < 0.1):
        forward = 0
        turn = 0

    #condition 2
    elif (goaly < 0.4 and goalx < 0.1):
        forward = goaly
        turn = 0 - goalx

    #condition 3
    elif (goaly < 0.4 and goalx > 0.1):
        forward = 0.25
```

```
        turn = 0 - goalx

    #condition 4
    else:
        forward = (goaly)/2.5
        turn = 0 - goalx*0.65

    if (count > 40):
        count = 0
        x = []
        y = []
        for meas in reading:
            x.append(doorNav.find_x(meas[1],meas[0]))
            y.append(doorNav.find_y(meas[1],meas[0]))

        plotReadings([door[0]],[door[1]],x,y,edges[0],edges[1])

    print (forward, turn)
    speed.linear.x=forward/2
    speed.angular.z=turn/2
    pub.publish(speed)


def callback2(msg):
    global prev_speed
    prev_speed = msg.linear.x
    prev_turn = msg.angular.z

rospy.init_node('scan_values')
pub.publish(speed)
sub = rospy.Subscriber('/laser', LaserScan, callback)
rospy.spin()
```

9.1.2 lidarProcessor.py

```
import math
import numpy as np
import matplotlib.pyplot as plt
import doorNav

# find edges in the captured data using change in distances
def edges(data, threshold, width):
    index = []
    degFilterStart = math.pi/2
    degFilterEnd = math.pi*1.5

    for i in range(len(data)-1):
        angle = data[i][1]
        # isolate to 90 -> 270 deg to prevent computer from being detected
        if (angle > degFilterStart and angle < degFilterEnd):
            ray = data[i][0]  # the current ray distance
            rayNext = data[i+1][0]  # the next ray distance
            changeInRay = rayNext - ray #change in dist btwn current and next

            #if there an increase ray distance, assume possible door opening
            if ( changeInRay > threshold ):
                opening = i #store possible edge of door opening
                iterator = i + 1
                while(iterator < len(data)-1 and angle < degFilterEnd):
                    iterator = iterator + 1 #increment position in data
                    angle = data[iterator][1] #update angle for end condition
                    ray = data[iterator-1][0]
                    rayNext = data[iterator][0]
                    changeInRay = rayNext - ray #for search of decreasing dist

                    if( changeInRay < (0 - threshold) ):
                        distance = calcDistance(data[opening],data[iterator])
                        print(distance)

                        if ( distance < 1.5*width and distance > 0.5* width ):
                            openingDistance = data[opening][0]
                            iteratorDistance = data[iterator][0]
                            radius = max(openingDistance, iteratorDistance)
                            safe = True
                            #if any point between edges has detection < radius
                            for point in range( opening+1,iterator ):
                                ray = data[point][0]
                                if ( ray < radius ):
                                    safe = False #don't apend points

                            # add if no detection between two edges
                            if ( safe ):
```

```
                                index.append([opening,iterator])
    #if edges found, rank them and return all valid edges for plotting
    if (len(index) > 0):
        positions = rankEdges(data, index, width)
        opening = data[positions[0]]
        closing = data[positions[1]]
        x = [doorNav.find_x(opening[1],opening[0]), doorNav.find_x(closing[1],
closing[0])]
        y = [doorNav.find_y(opening[1],opening[0]), doorNav.find_y(closing[1],
closing[0])]

        xCollated = []
        yCollated = []
        for point in index:
            hypOpening = data[point[0]][0]
            hypClosing = data[point[1]][0]
            angleOpening = data[point[0]][1]
            angleClosing = data[point[1]][1]
            xCollated.append(doorNav.find_x(angleOpening, hypOpening))
            xCollated.append(doorNav.find_x(angleClosing,hypClosing))
            yCollated.append(doorNav.find_y(angleOpening,hypOpening))
            yCollated.append(doorNav.find_y(angleClosing,hypClosing))
        return [[x,y],[xCollated,yCollated]]
    else:
        return []

#rank possible doorways
def rankEdges(data, index, expWidth):
    kw = 1 #higher = greater punishment on door width
    kd = 2 #lower = greater punishment on distance
    likelyEdges = None
    mostLikely = None
    for i in index:
        foundWidth = calcDistance(data[i[0]],data[i[1]])
        widthDev = abs(foundWidth-expWidth)
        avgDistance = (data[i[0]][0] + data[i[1]][0])/2
        likelihood = (widthDev * kw) + (avgDistance / kd)
        if(likelihood < mostLikely or mostLikely is None):
            mostLikely = likelihood
            likelyEdges = i
    return likelyEdges

#use pythag to determine distance between two points
def calcDistance(a,b):
    aHyp = a[0]
    bHyp = b[0]
    aTheta = (a[1])
    bTheta = (b[1])
```

```
ax = doorNav.find_x(aTheta, aHyp)
bx = doorNav.find_x(bTheta, bHyp)
ay = doorNav.find_y(aTheta, aHyp)
by = doorNav.find_y(bTheta, bHyp)

distance = math.sqrt(math.pow(ax-bx,2) + math.pow(ay-by,2))
return distance
```

9.1.3 doorNav.py

```
import math
import datetime
import numpy as np
import json
import time
import matplotlib.pyplot as plt

with open('config.json') as json_file:
    config = json.load(json_file)

#setup the chair dimensions and lidar range
def setup():
    with open('config.json') as json_file:
        config = json.load(json_file)

        # Calculate dimensions of boundary
        front = config["front"] + config["bufferFront"]
        side = config["side"] + config["bufferSide"]

        # Calculate range for detecting objects in front
        theta_range_1 = (math.atan(front/side)) + math.pi/2
        theta_range_2 = math.pi*1.5 - math.atan(front/side)
        return [theta_range_1,theta_range_2, front, side]

#Assigns an angle to the range and returns as array
def assign_angle(msg):
    measure = msg.ranges
    data = [None]*len(measure)
    angle = float(0)
    inc = msg.angle_increment

    # Collate lidar measure and angle into array data
    for i in range(len(measure)):
        data[i] = [measure[i],angle]
        angle = angle + inc

    return data

#segment lidar ranges into 5 segments
def segment_measurements(msg):
    lidarLeft = [None]*40
    lidarFrontLeft = [None]*40
    lidarFront = [None]*40
    lidarFrontRight = [None]*40
    lidarRight = [None]*40

    for i in range(40):
```

```
        lidarLeft[i]=msg.ranges[i+80]
        lidarFrontLeft[i]=msg.ranges[i+120]
        lidarFront[i]=msg.ranges[i+160]
        lidarFrontRight[i]=msg.ranges[i+200]
        lidarRight[i]=msg.ranges[i+240]

    left = round(min(lidarLeft),3)
    frontLeft = round(min(lidarFrontLeft),3)
    front = round(min(lidarFront),3)
    frontRight = round(min(lidarFrontRight),3)
    right = round(min(lidarRight),3)
    measurements = [left, frontLeft, front, frontRight, right] #segment 1,2,3,
4,5
    return measurements


def compare_change(bookmark,data,window):
    search = []
    for i in range(window):
        search.append(data[bookmark+i][0])#-data[bookmark+(i + 1)][0])
    median = np.median(search)
    if (search[0] < 2*median):
        return True
    else:
        return False

def compare_change_2(first_edge,data,window):
    index = []
    bookmark = first_edge + 1
    while (data[bookmark][1] < math.pi*1.5):
        search = []
        for i in range(window):
            search.append(data[bookmark-i][0])#-data[bookmark-(i + 1)][0])
        median = np.median(search)
        if (search[0] < 2*median):
            radius = max(data[first_edge][0],data[bookmark][0])
            safe = True
            for point in range(first_edge+1,bookmark):
                if (data[point][0] < radius):
                    safe = False
            if (safe):
                index.append([first_edge,bookmark])
        bookmark = bookmark + 1
    return index

def find_edges_v2(msg, threshold, width):
    data = assign_angle(msg)
    index = []
    edges = []
```

```
    recordx = []
    recordy = []
    probability = None
    edge1 = None
    edge2 = None
    likelyEdge = [None,None]

    for i in range(len(data)-1):
        meas = data[i][0]
        meas_next = data[i+1][0]
        angle = data[i][1]
        if (angle > math.pi/2 and angle < math.pi*1.5):
            if(compare_change(i,data,4)):
                index = compare_change_2(i,data,4)

    for i in index:
        edge1 = data[i[0]]
        edge2 = data[i[1]]
        edges.append(calc_distance(edge1,edge2))
        recordx.append(find_x(edge1[1],edge1[0]))
        recordy.append(find_y(edge2[1],edge2[0]))

    #print("found edges:",edges)

    for i in range(len(edges)):
        stat = abs(edges[i]-width)
        #print(stat,math.degrees(data[index[i][0]][1]),math.degrees(data[index
[i][1]][1]))
        if(stat < probability or probability is None):
            probability = stat
            point1 = index[i][0]
            point2 = index[i][1]
            likelyEdge = [data[point1],data[point2]]
            #print("--------------")
    return (likelyEdge,recordx,recordy)

def find_edges_v3(msg, threshold, width):
    data = assign_angle(msg)
    index = []
    edges = []
    recordx = []
    recordy = []
    probability = None
    edge1 = None
    edge2 = None
    likelyEdge = [None,None]
```

```
    for i in range(len(data)-1):
        meas = data[i][0]
        meas_next = data[i+1][0]
        angle = data[i][1]
        if (angle > math.pi/2 and angle < math.pi*1.5 and meas < 4):
            if (meas_next - meas > threshold):
                edge1 = i
                j = i
                while(j < len(data)-1 and data[j][1] < math.pi*1.5):
                    j = j+1
                    if(data[j][0] - data[j-1][0] < 0 - threshold):
                        radius = max(data[i][0],data[j][0])
                        safe = True
                        for point in range(i+1,j):
                            if (data[point][0] < radius):
                                safe = False
                        distance = calc_distance(data[i],data[j])
                        if (safe and distance < 1.6*width and distance > 0.4 *
 width):
                            index.append([i,j])

    for i in index:
        edge1 = data[i[0]]
        edge2 = data[i[1]]
        edges.append(calc_distance(edge1,edge2))
        recordx.append(find_x(edge1[1],edge1[0]))
        recordx.append(find_x(edge2[1],edge2[0]))
        recordy.append(find_y(edge1[1],edge1[0]))
        recordy.append(find_y(edge2[1],edge2[0]))

    #print("found edges:",edges)

    for i in range(len(edges)):
        stat = abs(edges[i]-width)
        #print(stat,math.degrees(data[index[i][0]][1]),math.degrees(data[index
[i][1]][1]))
        if(stat < probability or probability is None):
            probability = stat
            point1 = index[i][0]
            point2 = index[i][1]
            likelyEdge = [data[point1],data[point2]]
            #print("--------------")
    return [likelyEdge,recordx,recordy]

def find_edges(msg, threshold, width):
    data = assign_angle(msg)
    index = []
    edges = []
```

```
    recordx = []
    recordy = []
    probability = None
    edge1 = None
    edge2 = None
    likelyEdge = [None,None]


    for i in range(len(data)-1):
        meas = data[i][0]
        meas_next = data[i+1][0]
        angle = data[i][1]
        if (angle > math.pi/2 and angle < math.pi*1.5 and meas < 4):
            if (meas_next - meas > threshold):
                edge1 = i
                j = i
                while(j < len(data)-1 and data[j][1] < math.pi*1.5):
                    j = j+1
                    if(data[j][0] - data[j-1][0] < 0 - threshold):
                        radius = max(data[i][0],data[j][0])
                        safe = True
                        for point in range(i+1,j):
                            if (data[point][0] < radius):
                                safe = False
                        if (safe):
                            index.append([i,j])

    for i in index:
        edge1 = data[i[0]]
        edge2 = data[i[1]]
        edges.append(calc_distance(edge1,edge2))
        recordx.append(find_x(edge1[1],edge1[0]))
        recordx.append(find_x(edge2[1],edge2[0]))
        recordy.append(find_y(edge1[1],edge1[0]))
        recordy.append(find_y(edge2[1],edge2[0]))

    #print("found edges:",edges)

    for i in range(len(edges)):
        stat = abs(edges[i]-width)
        #print(stat,math.degrees(data[index[i][0]][1]),math.degrees(data[index
[i][1]][1]))
        if(stat < probability or probability is None):
            probability = stat
            point1 = index[i][0]
            point2 = index[i][1]
            likelyEdge = [data[point1],data[point2]]
            #print("--------------")
```

```
    return [likelyEdge,recordx,recordy]

def calc_distance(a,b):
    aHyp = a[0]
    bHyp = b[0]
    aTheta = (a[1])
    bTheta = (b[1])
    ax = find_x(aTheta, aHyp)
    bx = find_x(bTheta, bHyp)
    ay = find_y(aTheta, aHyp)
    by = find_y(bTheta, bHyp)

    distance = math.sqrt(math.pow(ax-bx,2) + math.pow(ay-by,2))
    return distance

def find_x(theta,hyp):
    if (theta < math.pi/2):
        theta = (math.pi/2) - theta
        result = math.cos(theta)*hyp
    elif (theta < math.pi):
        theta = theta - math.pi/2
        result = math.cos(theta)*hyp
    elif (theta < 3*math.pi/2):
        theta = (3*math.pi/2) - theta
        result = -math.cos(theta)*hyp
    else:
        theta = theta - (3*math.pi/2)
        result = -math.cos(theta)*hyp
    return result

def find_y(theta,hyp):
    if (theta < math.pi/2):
        theta = (math.pi/2) - theta
        result = -math.sin(theta)*hyp
    elif (theta < math.pi):
        theta = theta - math.pi/2
        result = math.sin(theta)*hyp
    elif (theta < 3*math.pi/2):
        theta = (3*math.pi/2) - theta
        result = math.sin(theta)*hyp
    else:
        theta = theta - (3*math.pi/2)
        result = -math.sin(theta)*hyp
    return result

#prints the segment min measurements from LiDAR
def show_lidar_measurements(data):
    print datetime.datetime.now().time()
```

```
    print data[3],"|",data[2],"|",data[1]
    print data[4],"|   o   |",data[0]
    print "  x   |   x   |   x"
    print "----------------------"


def check_if_safe(data):
    w_safe = config["w"]
    r_safe = w_safe/math.cos(config["a"]) - 0.1
    #print "r_safe:",r_safe
    f_safe = 0.25
    ok = True
    if data[0] < w_safe or data[4] < w_safe:
        ok = False
    elif data[1] < r_safe or data[3] < r_safe:
        ok = False
    elif data[2] < f_safe:
        ok = False
    return ok

#Return hypotenuse given adjacent side and angle
def calc_ray(adj, angle):
    result = adj / math.cos(angle)
    return result

#Placeholder for calculating the safety range for each measurement
def calc_boundaries(data):
    print "initialising boundaries..."
    with open('config.json') as json_file:
        config = json.load(json_file)
        front = config["front"] + config["bufferFront"]/2
        side = config["side"] + config["bufferSide"]
        angle1 = (math.atan(front/side)) + math.pi/2
        angle2 = math.pi*1.5 - math.atan(front/side)
        result = []

        for i in range(len(data)):
            theta = data[i][1]
            if (theta > angle1 and theta < angle2):
                if (theta < math.pi):
                    result.append( calc_ray(front, math.pi - theta) )
                else:
                    result.append( calc_ray(front, theta - math.pi) )
            else:
                if (theta > math.pi/2 and theta <= angle1):
                    result.append( calc_ray(side, theta - math.pi/2) )
                elif (theta < math.pi*1.5 and theta >= angle2):
                    result.append( calc_ray(side, theta - math.pi*1.5) )
                else:
```

```
                    result.append(0)
        return result

def calc_chair_dimensions(data):
    print "initialising boundaries..."
    with open('config.json') as json_file:
        config = json.load(json_file)
        front = config["front"] + (config["bufferFront"]/3)
        side = config["side"] + (config["bufferSide"]/3)
        angle1 = (math.atan(front/side)) + math.pi/2
        angle2 = math.pi*1.5 - math.atan(front/side)
        result = []

        for i in range(len(data)):
            theta = data[i][1]
            if (theta > angle1 and theta < angle2):
                if (theta < math.pi):
                    result.append( calc_ray(front, math.pi - theta) )
                else:
                    result.append( calc_ray(front, theta - math.pi) )
            else:
                if (theta > math.pi/2 and theta <= angle1):
                    result.append( calc_ray(side, theta - math.pi/2) )
                elif (theta < math.pi*1.5 and theta >= angle2):
                    result.append( calc_ray(side, theta - math.pi*1.5) )
                else:
                    result.append(0)
        return result

def determine_movement(data):
    heightLeft = math.sin(config["a"]*1.5)*data[3]
    heightRight = math.sin(config["a"]*1.5)*data[1]
    widthLeft = math.cos(config["a"]*1.5)*data[3]
    widthRight = math.cos(config["a"]*1.5)*data[1]
    forward = 0
    turn = 0

    with open('config.json') as json_file:
        config = json.load(json_file)

        # Calculate dimensions of boundary
        front = config["front"] + config["bufferFront"]
        side = config["side"] + config["bufferSide"]

        forward = min(data[2],heightLeft,heightRight)
        left = data[0] - 0.4 + widthLeft - 0.4

    return [forward, turn]
```

```
def proceed_forward(reading, boundary):
    result = True
    for i in range(len(boundary)):
        if (reading[i][0] < boundary[i]):
            result = False

    return result

def navigate(reading, boundary, safety, angle1, angle2):
    frontInBounds = True
    leftInBounds = True
    rightInBounds = True

    leftMeasure = angle1
    rightMeasure = angle2
    frontMeasure = 12

    left = 0
    right = 0
    forward = 0

    for i in range(len(boundary)):
        theta = reading[i][1]
        distance = reading[i][0]
        #print reading[i]

        if (theta > math.pi/2 and theta < math.pi*1.5):
            if (distance < 0.85):
                #print(theta, "<", math.pi, theta<math.pi)
                if (theta < math.pi):
                    leftMeasure = max(leftMeasure,theta)
                    #print(leftMeasure)
                else:
                    #print(theta, ">", math.pi, theta>math.pi)
                    #print theta
                    rightMeasure = min(rightMeasure, theta)
                    #print(rightMeasure)

        if (theta < angle1):

            left = min(distance, left)
            if left:
                left = 1/left
            if (distance < safety[i]):
                leftInBounds = False

        elif (theta > angle1 and theta < angle2):
```

```
            if (distance < safety[i]):
                frontInBounds = False
            else:
                frontMeasure = min(frontMeasure,distance)

        else:
            if (distance < boundary[i] and theta < math.pi*1.5):
                right = min(distance, right)
                if right:
                    right = 1/right
            if (distance < safety[i]):
                rightInBounds = False

    #if (frontInBounds & leftInBounds & rightInBounds):
    max_speed = 0.6
    max_distance_speed = 2
    if (frontMeasure > max_distance_speed):
        frontMeasure = 2
    forward = (frontMeasure*0.4) - 0.2
    if (forward > max_speed):
        forward = max_speed

    if (not frontInBounds):
        forward = 0

    if (not leftInBounds and rightInBounds):
        right = 0.5

    if (not rightInBounds and leftInBounds):
        left = 0.5

    if (not rightInBounds and not leftInBounds):
        forward = 0
        right = 0
        left = 0

    print("measurements",leftMeasure," ", rightMeasure)
    print("turns",left," ", right)


    turn = left - right

    return[forward, turn]

def navigate2(readings, boundary, safety, angle1, angle2):
    leftMeasure = 12
    rightMeasure = 12
    forward = 0
```

```
frontMeasure = 12

for i in range(len(boundary)):
    theta = readings[i][1]
    distance = readings[i][0]

    if (theta > math.pi/2 and theta < math.pi*1.5):
        if (theta < math.pi and distance < 1.5):
            leftMeasure = min(distance, leftMeasure)
            leftAngle = max
        if (theta > math.pi and distance < 1.5):
            rightMeasure = min(distance, rightMeasure)

    if(theta > angle1 and theta < angle2):
        if (distance < safety[i]):
            frontInBounds = False
        else:
            frontMeasure = min(frontMeasure,distance)


if (rightMeasure > 1):
    right = 0
else:
    right = 1/rightMeasure

if (leftMeasure > 1):
    left = 0
else:
    left = 1/leftMeasure

if (frontMeasure > 3):
    frontMeasure = 0.6
else:
    frontMeasure = (frontMeasure/3) - 0.2


turn = left - right
forward = frontMeasure

return[forward, turn]
```

9.2 CODE ASSOCIATED WITH CARTOGRAPHER SLAM

This section contains the configuration files used to tune the Cartographer SLAM system

9.2.1 2d_lidar_0_deg_optimised_with_partial_odom.lua

```
-- Copyright 2016 The Cartographer Authors
--
-- Licensed under the Apache License, Version 2.0 (the "License");
-- you may not use this file except in compliance with the License.
-- You may obtain a copy of the License at
--
--      http://www.apache.org/licenses/LICENSE-2.0
--
-- Unless required by applicable law or agreed to in writing, software
-- distributed under the License is distributed on an "AS IS" BASIS,
-- WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
-- See the License for the specific language governing permissions and
-- limitations under the License.

include "map_builder.lua"
include "trajectory_builder.lua"

options = {
  map_builder = MAP_BUILDER,
  trajectory_builder = TRAJECTORY_BUILDER,
  map_frame = "map",
  tracking_frame = "chassis",
  published_frame = "chassis",
  odom_frame = "wheel_odom",
  provide_odom_frame = true,
  publish_frame_projected_to_2d = true,
  use_odometry = true,
  use_nav_sat = false,
  use_landmarks = false,
  num_laser_scans = 1,
  num_multi_echo_laser_scans = 0,
  num_subdivisions_per_laser_scan = 1,
  num_point_clouds = 0,
  lookup_transform_timeout_sec = 0.2,
  submap_publish_period_sec = 0.5,
  pose_publish_period_sec = 5e-3,
  trajectory_publish_period_sec = 30e-3,
  rangefinder_sampling_ratio = 1.,
  odometry_sampling_ratio = 1.,
  fixed_frame_pose_sampling_ratio = 1.,
  imu_sampling_ratio = 1.,
  landmarks_sampling_ratio = 1.,
}
```

```
MAP_BUILDER.use_trajectory_builder_2d = true

-- input stream settings
TRAJECTORY_BUILDER_2D.min_range = 0.3
TRAJECTORY_BUILDER_2D.max_range = 11.5
TRAJECTORY_BUILDER_2D.missing_data_ray_length = 11.5
TRAJECTORY_BUILDER_2D.use_imu_data= false
TRAJECTORY_BUILDER_2D.adaptive_voxel_filter.max_range = 11.5
TRAJECTORY_BUILDER_2D.loop_closure_adaptive_voxel_filter.max_range = 11.5
TRAJECTORY_BUILDER_2D.num_accumulated_range_data = 1

-- local slam
TRAJECTORY_BUILDER_2D.submaps.num_range_data = 300

-- ceres scan matcher
TRAJECTORY_BUILDER_2D.motion_filter.max_angle_radians = math.rad(3.5)
TRAJECTORY_BUILDER_2D.ceres_scan_matcher.translation_weight = 1e-3
TRAJECTORY_BUILDER_2D.ceres_scan_matcher.rotation_weight = 1e3
TRAJECTORY_BUILDER_2D.ceres_scan_matcher.occupied_space_weight = 1e-2

-- realtime scan match
TRAJECTORY_BUILDER_2D.use_online_correlative_scan_matching = true
TRAJECTORY_BUILDER_2D.real_time_correlative_scan_matcher.linear_search_window
= 1e-1
TRAJECTORY_BUILDER_2D.real_time_correlative_scan_matcher.translation_delta_cos
t_weight = 1e-3
TRAJECTORY_BUILDER_2D.real_time_correlative_scan_matcher.rotation_delta_cost_w
eight = 1e-3


--global slam
POSE_GRAPH.constraint_builder.min_score = 0.65
POSE_GRAPH.constraint_builder.log_matches = true
POSE_GRAPH.constraint_builder.fast_correlative_scan_matcher.linear_search_wind
ow = 20.0
POSE_GRAPH.constraint_builder.fast_correlative_scan_matcher.angular_search_win
dow = math.rad(10)
POSE_GRAPH.optimization_problem.ceres_solver_options.max_num_iterations = 100
POSE_GRAPH.constraint_builder.sampling_ratio = 1
POSE_GRAPH.optimization_problem.log_solver_summary = true
POSE_GRAPH.optimization_problem.huber_scale = 1e2
POSE_GRAPH.optimize_every_n_nodes = 25


-- localise weighting
POSE_GRAPH.optimization_problem.local_slam_pose_translation_weight = 1e2
POSE_GRAPH.optimization_problem.local_slam_pose_rotation_weight = 1e2
POSE_GRAPH.optimization_problem.odometry_translation_weight = 1
```

```
POSE_GRAPH.optimization_problem.odometry_rotation_weight = 0

return options
```

9.2.2 2d_lidar_25_deg_optimised_with_partial_odom.lua

```
-- Copyright 2016 The Cartographer Authors
--
-- Licensed under the Apache License, Version 2.0 (the "License");
-- you may not use this file except in compliance with the License.
-- You may obtain a copy of the License at
--
--      http://www.apache.org/licenses/LICENSE-2.0
--
-- Unless required by applicable law or agreed to in writing, software
-- distributed under the License is distributed on an "AS IS" BASIS,
-- WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
-- See the License for the specific language governing permissions and
-- limitations under the License.

include "map_builder.lua"
include "trajectory_builder.lua"

options = {
  map_builder = MAP_BUILDER,
  trajectory_builder = TRAJECTORY_BUILDER,
  map_frame = "map",
  tracking_frame = "chassis",
  published_frame = "chassis",
  odom_frame = "wheel_odom",
  provide_odom_frame = true,
  publish_frame_projected_to_2d = true,
  use_odometry = true,
  use_nav_sat = false,
  use_landmarks = false,
  num_laser_scans = 1,
  num_multi_echo_laser_scans = 0,
  num_subdivisions_per_laser_scan = 1,
  num_point_clouds = 0,
  lookup_transform_timeout_sec = 0.2,
  submap_publish_period_sec = 0.5,
  pose_publish_period_sec = 5e-3,
  trajectory_publish_period_sec = 30e-3,
  rangefinder_sampling_ratio = 1.,
  odometry_sampling_ratio = 1.,
  fixed_frame_pose_sampling_ratio = 1.,
  imu_sampling_ratio = 1.,
  landmarks_sampling_ratio = 1.,
}

MAP_BUILDER.use_trajectory_builder_2d = true

-- input stream settings
```

```
TRAJECTORY_BUILDER_2D.min_range = 0.3
TRAJECTORY_BUILDER_2D.max_range = 11.5
TRAJECTORY_BUILDER_2D.missing_data_ray_length = 11.5
TRAJECTORY_BUILDER_2D.use_imu_data= false
TRAJECTORY_BUILDER_2D.adaptive_voxel_filter.max_range = 11.5
TRAJECTORY_BUILDER_2D.loop_closure_adaptive_voxel_filter.max_range = 11.5
TRAJECTORY_BUILDER_2D.num_accumulated_range_data = 1

-- local slam
TRAJECTORY_BUILDER_2D.submaps.num_range_data = 100

-- ceres scan matcher
TRAJECTORY_BUILDER_2D.motion_filter.max_angle_radians = math.rad(3.5)
TRAJECTORY_BUILDER_2D.ceres_scan_matcher.translation_weight = 1e-3
TRAJECTORY_BUILDER_2D.ceres_scan_matcher.rotation_weight = 1e3
TRAJECTORY_BUILDER_2D.ceres_scan_matcher.occupied_space_weight = 1e-1

-- realtime scan match
TRAJECTORY_BUILDER_2D.use_online_correlative_scan_matching = true
TRAJECTORY_BUILDER_2D.real_time_correlative_scan_matcher.linear_search_window
= 1e-1
TRAJECTORY_BUILDER_2D.real_time_correlative_scan_matcher.translation_delta_cos
t_weight = 1e-3
TRAJECTORY_BUILDER_2D.real_time_correlative_scan_matcher.rotation_delta_cost_w
eight = 1e-3


--global slam
POSE_GRAPH.constraint_builder.min_score = 0.65
POSE_GRAPH.constraint_builder.log_matches = true
POSE_GRAPH.constraint_builder.fast_correlative_scan_matcher.linear_search_wind
ow = 20.0
POSE_GRAPH.constraint_builder.fast_correlative_scan_matcher.angular_search_win
dow = math.rad(10)
POSE_GRAPH.optimization_problem.ceres_solver_options.max_num_iterations = 100
POSE_GRAPH.constraint_builder.sampling_ratio = 1
POSE_GRAPH.optimization_problem.log_solver_summary = true
POSE_GRAPH.optimization_problem.huber_scale = 1e2
POSE_GRAPH.optimize_every_n_nodes = 25


-- localise weighting
POSE_GRAPH.optimization_problem.local_slam_pose_translation_weight = 3
POSE_GRAPH.optimization_problem.local_slam_pose_rotation_weight = 1
POSE_GRAPH.optimization_problem.odometry_translation_weight = 1
POSE_GRAPH.optimization_problem.odometry_rotation_weight = 0

return options
```

9.3 CODE ASSOCIATED WITH PUPIL DETECTION

This section contains the code for detecting pupils using the OpenCV library.

9.3.1 blobDetector.py

The program is adapted from work performed by Stepan Filanov [36].

```
import cv2
import numpy as np
import array
import datetime
img = cv2.imread("me.png")
face_cascade = cv2.CascadeClassifier('haarcascade_frontalface_default.xml')
eye_cascade = cv2.CascadeClassifier('haarcascade_eye.xml')
detector_params = cv2.SimpleBlobDetector_Params()
detector_params.filterByArea = True
detector_params.maxArea = 1600
detector = cv2.SimpleBlobDetector_create(detector_params)

#For holding previous points of pupil values
left_pupil = [None] * 10
right_pupil = [None] * 10

threshold = 20

def cut_empty(img):
    h, w = img.shape[:2]
    upper_h = int(h / 4)
    lower_h = int(h*4 / 5)
    img = img[upper_h:lower_h, 0:w] #remove empty space above/below eye
    return img

def detect_eyes(img, classifier):
    grey = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
    eyes = classifier.detectMultiScale(grey, 1.3, 5) # detect eyes
    width = np.size(img, 1) # get face frame width
    height = np.size(img, 0) # get face frame height
    left_eye = None
    right_eye = None
    for (x, y, w, h) in eyes:
        if y > height / 2:
            pass
        eyecenter = x + w / 2  # get the eye center
        if eyecenter < width / 2:
            left_eye = img[y:(y + h), x:(x + w)]
        else:
            right_eye = img[y:(y + h), x:(x + w)]
    return left_eye, right_eye
```

```
def detect_faces(img, classifier):
    grey = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
    coords = classifier.detectMultiScale(grey, 1.3, 5)
    if len(coords) > 1:
        max = (0, 0, 0, 0)
        for i in coords:
            if i[3] > max[3]:
                max = i
        max = np.array([i], np.int32)
    elif len(coords) == 1:
        max = coords
    else:
        return None
    for (x, y, w, h) in max:
        frame = img[y:(y + h), x:(x + w)]
    return frame

def blob_process(img, threshold, detector):
    grey = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
    hist = cv2.equalizeHist(grey)
    _, img = cv2.threshold(hist, threshold, 255, cv2.THRESH_BINARY)
    horizontal_kernel = cv2.getStructuringElement(cv2.MORPH_RECT, (1, 2))

    img = cv2.dilate(img, horizontal_kernel, iterations=2)

    keypoints = detector.detect(img)
    return [keypoints, img]

def testing():
    vid = cv2.VideoCapture(0)
    while True:
        _, frame = vid.read()
        face_frame = detect_faces(frame, face_cascade)
        if face_frame is not None:
            eyes = detect_eyes(face_frame, eye_cascade)
            for eye in eyes:
                if eye is not None:
                    eye = cut_empty(eyes[0])
                    grey = cv2.cvtColor(eye, cv2.COLOR_BGR2GRAY)
                    hist = cv2.equalizeHist(grey)
                    _, img = cv2.threshold(hist, threshold, 255, cv2.THRESH_BI
NARY)
                    horizontal_kernel = cv2.getStructuringElement(cv2.MORPH_RE
CT, (1,2))
                    circle_kernel = cv2.getStructuringElement(cv2.MORPH_ELLIPS
E, (3,3))
```

```
                    square_kernel = cv2.getStructuringElement(cv2.MORPH_ELLIPS
E, (3,3))
                    preImg = img
                    img = cv2.dilate(img, horizontal_kernel, iterations=3)
                    #img = cv2.morphologyEx(img, cv2.MORPH_CLOSE, circle_kerne
l)
                    #img = cv2.morphologyEx(img, cv2.MORPH_OPEN, horizontal_ke
rnel, iterations=3)

                    # repair_kernel = cv2.getStructuringElement(cv2.MORPH_RECT
, (2,4))
                    # img = 255 - cv2.morphologyEx(255 - img, cv2.MORPH_CLOSE,
 repair_kernel, iterations=1)

                    # thresh = cv2.threshold(grey, 0, 255, cv2.THRESH_BINARY_I
NV + cv2.THRESH_OTSU)[1]

                    # detected_lines = cv2.morphologyEx(thresh, cv2.MORPH_OPEN
, horizontal_kernel, iterations=4)
                    # cnts = cv2.findContours(detected_lines, cv2.RETR_EXTERNA
L, cv2.CHAIN_APPROX_SIMPLE)
                    # cnts = cnts[0] if len(cnts) == 2 else cnts[1]
                    # for c in cnts:
                    #     cv2.drawContours(img, [c], -1, (255,255,255), 2)
                    #img = cv2.dilate(img, square_kernel, iterations = 1) #1
                    #img = cv2.erode(img, square_kernel, iterations=1) #2
                    keypoints = detector.detect(img)

        cv2.imshow('my image 1', img1)
        cv2.imshow('my image 2', img2)
        cv2.imshow('my image 3', img3)
        cv2.imshow('my image 4', img4)
        cv2.imshow('my image 5', img5)
        cv2.imshow('my image 6', img6)
        cv2.imshow('grey',grey)
        cv2.imshow('hist',hist)
        cv2.imshow('binary pre',preImg)
        if cv2.waitKey(1) & 0xFF == ord('q'):
            break
    vid.release()
    cv2.destroyAllWindows()

def nothing(x):
    pass

def main():
    frame_place = 1
    vid = cv2.VideoCapture(0)
```

```
    cv2.namedWindow('image')
    cv2.createTrackbar('threshold', 'image', 0, 255, nothing)
    while True:
        _, frame = vid.read()
        face_frame = detect_faces(frame, face_cascade)
        if face_frame is not None:
            eyes = detect_eyes(face_frame, eye_cascade)
            i = 0
            for eye in eyes:
                i = i+1
                if eye is not None:
                    threshold = 11 #cv2.getTrackbarPos('threshold', 'image')
                    eye = cut_empty(eye)
                    [keypoints,binary] = blob_process(eye, threshold, detector
)
                    eye = cv2.drawKeypoints(eye, keypoints, eye, (0, 0, 255),
cv2.DRAW_MATCHES_FLAGS_DRAW_RICH_KEYPOINTS)
                    cv2.imshow(str(i) + ' binary eye',binary)
        cv2.imshow('my image', frame)
        cv2.imwrite('./blob/br'+str(frame_place)+'.jpg',frame)
        frame_place = frame_place + 1
        if cv2.waitKey(1) & 0xFF == ord('q'):
            break
    vid.release()
    cv2.destroyAllWindows()

main()
```

9.3.2 houghCirclesDetector.py

The program is adapted from work performed by Michael Wirth [37] and work performed by Stepan Filanov [36].

```
import cv2
import numpy as np
import array
img = cv2.imread("test_1.jpg")
face_cascade = cv2.CascadeClassifier('haarcascade_frontalface_default.xml')
eye_cascade = cv2.CascadeClassifier('haarcascade_eye.xml')

threshold = 20
threshold_upper = 100
threshold_lower = 90
min_rad = 3
max_rad = 8
min_dist = 200
global accLeftEye
global accRightEye
accLeftEye = []
accRightEye = []

def cut_empty(img):
    h, w = img.shape[:2]
    upper_h = int(h / 4)
    lower_h = int(h*4 / 5)
    img = img[upper_h:lower_h, 0:w] #remove empty space above/below eye
    return img

def find_eyes(img, classifier):
    grey = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
    eyes = classifier.detectMultiScale(grey, 1.3, 5) #find eye frames
    face_w = np.size(img, 1) #face frame width
    face_h = np.size(img, 0) #face frame height
    left_eye = None
    right_eye = None
    for (x,y,w,h) in eyes:
        if y > (face_h / 2):
            pass #don't use anything on bottom of face (false detections)

        else:
            center = x+w/2 #center (between the eyes)
            if center < face_w / 2:
                left_eye = img[y:(y + h), x:(x + w)]
            else:
                right_eye = img[y:(y + h), x:(x + w)]
    return left_eye, right_eye
```

```
def find_face(img, classifier):
    grey = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
    coords = classifier.detectMultiScale(grey, 1.3, 5)
    if len(coords) > 1:
        max = (0, 0, 0, 0)
        for i in coords:
            if i[3] > max[3]:
                max = i
        max = np.array([i], np.int32)
    elif len(coords) == 1:
        max = coords
    else:
        return None
    for (x, y, w, h) in biggest:
        frame = img[y:(y + h), x:(x + w)]
    return frame

def to_binary(img, threshold):
    hist_frame = cv2.equalizeHist(gray_frame)
    __, img = cv2.threshold(hist_frame, threshold, 255, cv2.THRESH_BINARY)
    return img

def grayscale(img):
    grey = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
    #img = grey
    img = cv2.equalizeHist(grey)
    return img

def get_edges(img, threshold_lower, threshold_upper):
    img = cv2.Canny(img, threshold_lower, threshold_upper, apertureSize=3)
    return img

def find_pupils(eye, img, min_dist, min_rad, max_rad, threshold_uppper):
    circles = cv2.HoughCircles(eye, cv2.HOUGH_GRADIENT, 1,min_dist,
param1=threshold_upper, param2=5, minRadius=min_rad, maxRadius=max_rad)
    circles = np.uint16(np.around(circles))
    for i in circles[0,:]:
        # Planning: append to array the position of detected pupil
        #           if array len > 10, delete position [0]
        #           draw circle around average of positions in array

        # draw the outer circle
        cv2.circle(img,(i[0],i[1]),i[2],(0,255,0),1)
        # draw the center of the circle
        cv2.circle(img,(i[0],i[1]),1,(0,0,255),1)
    return img
```

```
def nothing(x):
    pass

def main():
    vid = cv2.VideoCapture(0)
    place = 1

    while True:
        _, camImg = vid.read()

        cv2.namedWindow('camera feed')
        cv2.createTrackbar('threshold_upper', 'camera feed', 170, 255, nothing
)
        cv2.createTrackbar('threshold_lower', 'camera feed', 150, 255, nothing
)
        #cv2.createTrackbar('min_rad', 'camera feed', 3, 5, nothing)
        #cv2.createTrackbar('max_rad', 'camera feed', 10, 15, nothing)
        #cv2.createTrackbar('min_dist', 'camera feed', 30, 50, nothing)

        face = find_face(camImg, face_cascade)
        eye_1 = None
        if face is not None:
            eyes = find_eyes(face, eye_cascade)
        for eye in eyes:
            if eye is not None:
                threshold_upper = 255#cv2.getTrackbarPos('threshold_upper', 'c
amera feed')
                threshold_lower = 82#cv2.getTrackbarPos('threshold_lower', 'ca
mera feed')
                #min_rad = cv2.getTrackbarPos('min_rad', 'camera feed')
                #max_rad = cv2.getTrackbarPos('max_rad', 'camera feed')
                #min_dist = cv2.getTrackbarPos('min_dist', 'camera feed')
                eye = cut_empty(eye)
                grey = grayscale(eye)
                edges = get_edges(grey, threshold_lower, threshold_upper)
                pupils = find_pupils(edges, eye, min_dist, min_rad, max_rad, t
hreshold_upper)
                if eye_1 is not None:
                    cv2.imshow('binary eye 2', edges)
                    cv2.imshow('greyscale eye 2',grey)
                else:
                    eye_1 = eye
                    cv2.imshow('binary eye 1',edges)
                    cv2.imshow('greyscale eye 1',grey)
        cv2.imshow('camera feed', camImg)
        cv2.imwrite('./hough/br'+str(place)+'.jpg',camImg)
        place = place + 1
```

```
        if cv2.waitKey(1) & 0xFF == ord('q'):
            break

    vid.release()
    cv2.destroyAllWindows()

main()
```