Project 88 Report

Laser Triangulation SLAM

Author	*Samuel Martin*
Author ID	*C1319546*
Module	*One Semester Individual Project*
Module Code	*CM3203*
Module Credits	*40*
Project Supervisor	*David Marshall*

Abstract

By using a inexpensive laser triangulation scanner this project aims to create an affordable robotic mapping solution suitable for building 3D models of interior environments. This project covers the physical design and build of the robot as well as the software implementation.

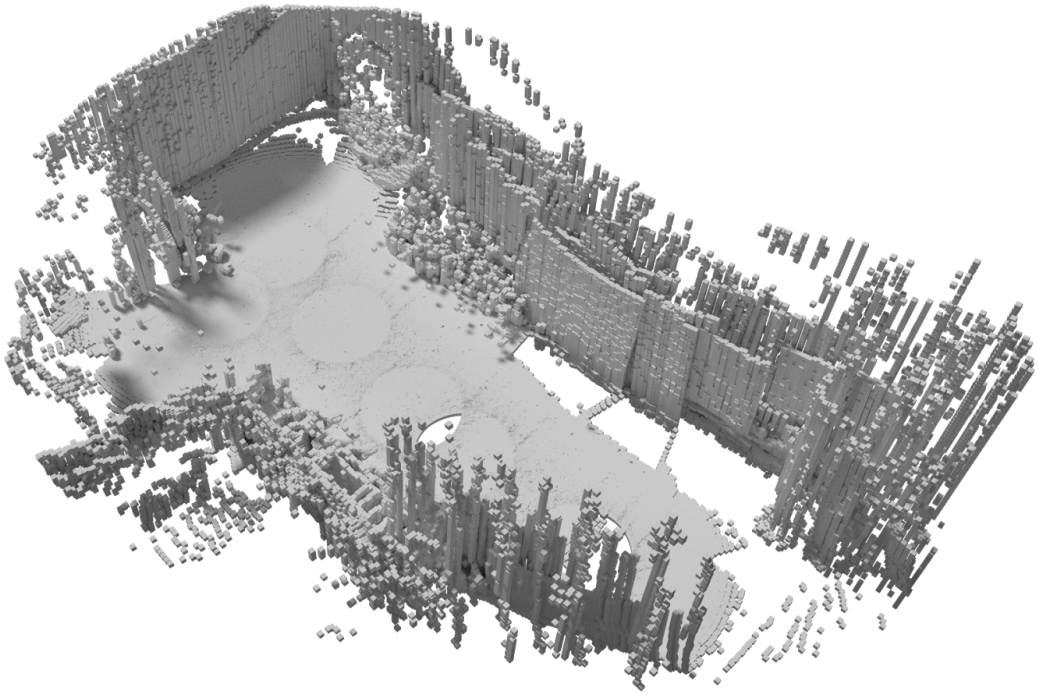


Figure 1: Artistic representation of captured scan data

Contents

1 - Introduction

2 - Background research

3 - System Overview
- 3.1 - Software Overview
- 3.2 - Hardware Overview:

4 - Hardware Implementation
- 4.1 - Mechanical Parts
- 4.2 - Electronic Parts

5 - Software Implementation
- 5.1 - Robot Module
- 5.2 - Scanner Module
- 5.3 - Point Cloud Module
- 5.4 - Voxel and Voxel Grid Modules
- 5.5 - A* Route planner & Navigation Module
- 5.6 - Motor Controller Modules
- 5.7 - Networking
- 5.8 - Intrinsic Parameter

6 - Test Results
- 6.1 - Module Tests
- 6.2 - Real world testing
- 6.3 - System Speed
- 6.4 - System Price

7 - Limitations
- 7.1 - Environmental factors
- 7.2 - Trade offs

8 - Future development

9 - Critical Reflection

10 - Conclusion

1 - Introduction

1.1 - Problem Overview

The overall objective is to create a robot which can safely navigate an environment whilst building up a comprehensive 3D model of its surroundings. The scanner is regularly the most expensive component when it comes to mobile robotics. Because of this, the key objective of this project is to create a cheap 3D scanner which has a high point density resolution to use to navigate an unknown environment safely.
LIDAR is the most commonly used scanner for small robotics platforms, however most scanners are far outside of budget. Laser triangulation scanners are easy to build and cheap making them an inexpensive substitute to LIDAR.
The second project objective was to analyse this sensor data and use it in conjunction with route planning algorithms to explore an environment to build up a 3D model safely and efficiently.

1.2 - Project Brief

The official success criteria from the Initial Plan[1] are as follows. These will form the majority of the testing criteria.

1.2.1 - Accuracy

“The SLAM world model needs to be accurate enough to navigate around messy environments. For example a desk covered in books and papers.”

1.2.2 - Adaptability

“The system should adapt to new environments automatically.”

1.2.3 - Speed

“The robotic platform should be able to scan an environment within a reasonable amount of time. For example, one scan and one 30cm route plan traversal should take no more than two minutes to complete.”

1.2.4 - Cost

“The entire system’s component value should be no more than £100”

1.3 - Technology Overview

The following chapter should give a concise overview of the technologies used in this project.

1.3.1 - Laser Triangulation Scanning

Laser triangulation scanning works by firing a laser into a scene at an angle and analysing the reflected beam shape with a sensor to determine depth information. This is classically used in measuring known objects or scanning isolated objects. This is done by mounting the object on a rotating or moving platform. However the same methodology can be applied to interior environments if the scanner is placed on the rotating platform instead of the object to be scanned.

1.3.2 - SLAM Methodology

SLAM stands for Simultaneous Location And Mapping. This is used regularly by mobile robotics platforms to ascertain its surroundings. The first step of SLAM is to retrieve a map of the local area using some form of scanner, be that lidar, sonar or structured light. As the robot moves around it re-calculates its position by using a mixture of built in odometry and localisation techniques. This step is where it matches its current world model to previous world models to calculate its position. This relies on the environment having enough interest points to match the two correctly.

1.3.3 - ICP Registration

ICP stands for Iterative Closest Point and is a registration algorithm used for matching point clouds together. Registration will form a key step in the SLAM methodology implemented in this project.

1.3.4 - Odometry Based Position Triangulation

ICP converges much faster if the two scans have a relatively accurate initial transformation. This can be achieved by using the robot’s built in odometry to transform the world models to match the traversal made by the robot before ICP. The built in odometry is also key for maneuvering the robot around obstacles in the world.

2 - Background research

This chapter gives a brief overview of the background research required for this project. This includes hardware and software research from academic and industrial sources.

2.1 - Scanner

Laser Triangulation scanning was first developed by the National Research Council of Canada back in 1978 and has been used for many industrial measurement systems.[2] These scanners have become widely available due to their drop in component price. They can be fabricated cheaply with readily available parts. These provide a high resolution depth map using infrared or visible light lasers combined with any form of digital camera.

2.1.1 - Laser Triangulation Trigonometry Overview

Laser triangulation scanners use laser light to probe the environment. This is done by mounting some form of light sensor a certain distance from the laser and as the laser gets fired into the scene the sensor can measure where it falls within its field of view.

As long as the distance between the sensor and the laser is known as well as their relative angles, we can form an angle-side-angle triangle as shown in Figure 2. A full breakdown of these equations can be found later in this chapter.

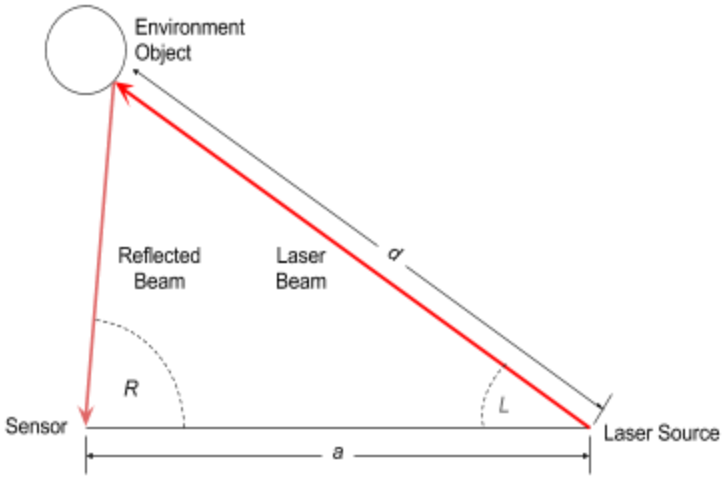


Figure 2: Diagram of generic laser triangulation layout

Known

a = *Distance between sensor and laser*

L = *Angle between laser and camera*

R = *Reflected laser angle in relation to the sensor*

Unknown

d = *Unknown distance between the laser and the environment object*

Relationship

$$d = \frac{a(sin(R))}{sin(180-L-R)}$$

A 1D sensor and single dot laser can measure along a 1D intersection with the environment. However a 2D sensor such as a camera and a line laser can capture a 2D plane intersection with the environment. We can use the incoming horizontal pixel index (i_u) for each row of the camera to determine the angle R. This assumes that the FOV of the camera is known. Figure 3 shows how a camera can be utilised as the laser detection sensor.

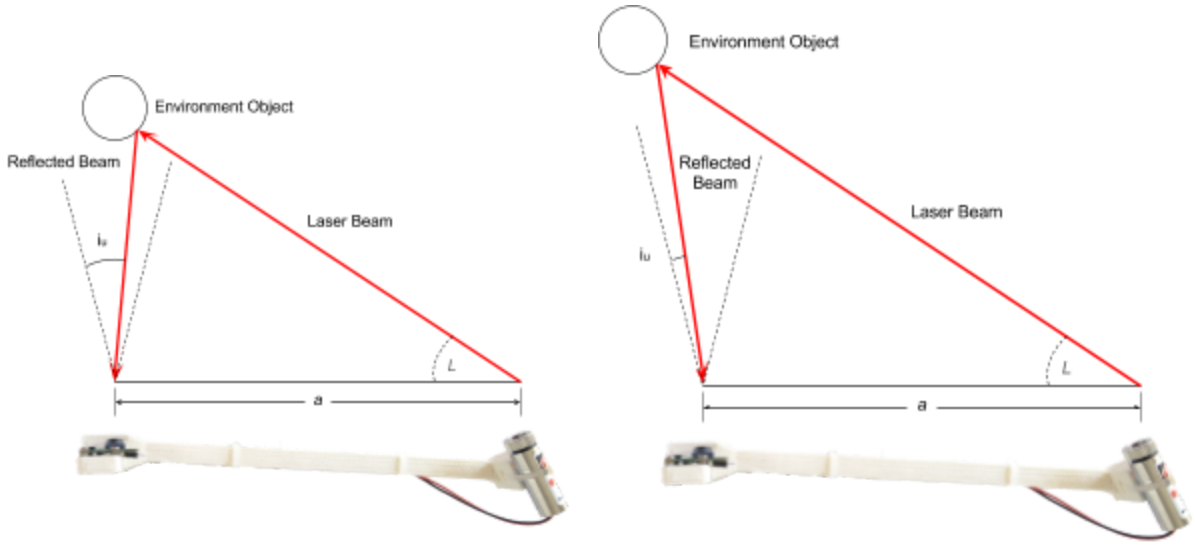


Figure 3: Diagrams showing change in environment distance

If we visualise this with a semi-complex primitive monkey head, we can visualise how the laser would fall on it. Figure 4 shows how the laser intersects with its environment from the third person view of the object and the scanner.

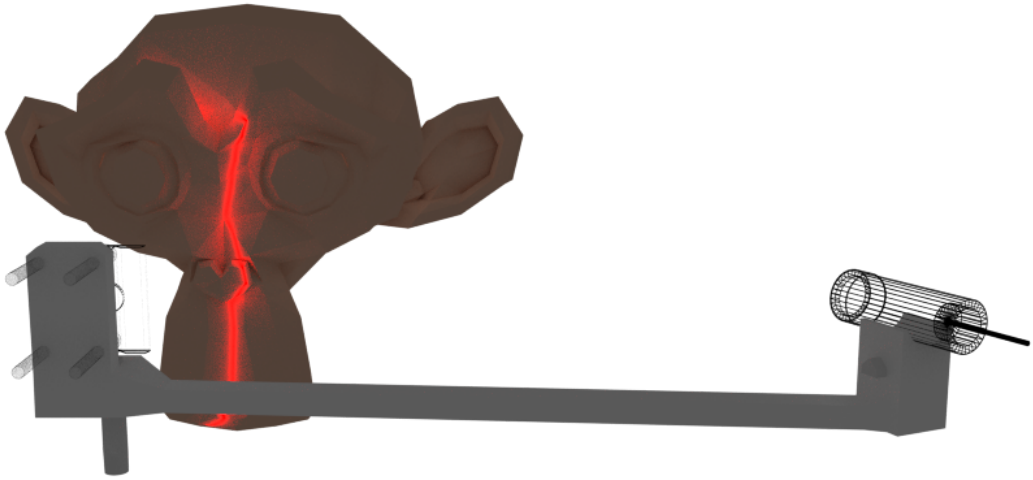


Figure 4: 3D Visualisation of Figure 3

Figure 5 shows how the laser intersects with its environment from the camera's point of view.

Figure 5a: 3D Visualisation of the camera view in Figure 4

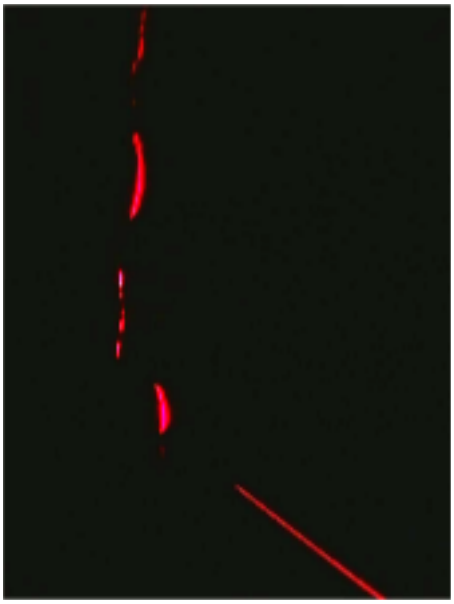


Figure 5b: Image captured from a real world example

As we need a 3D scan of the environment, we need to introduce a new dimension to these one dimensional slices. We can do this by moving the scanner. This can be done as a linear sweep or a rotation. For this project a turntable is more suitable as it complies a panorama style scan which will give the robot 360 degree vision. Due to mechanical constraints it is easier to mount the camera at the center of the turntable to reduce unwanted camera motion between slice captures. As the scanner rotates, it captures 2D intersecting slices of the environment.

Figure 6 shows an artistic representation of how these slices are captured from the camera's point of view. This was created by overlaying multiple captures whilst rotating the scanner. Each capture has been shifted by 6% so that each slice is identifiable.

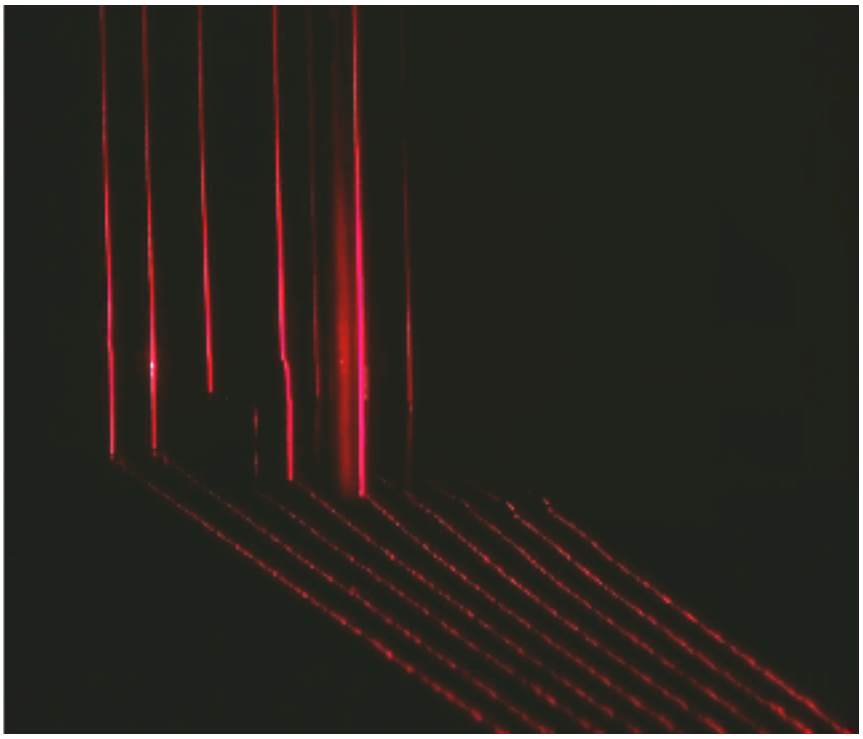


Figure 6: Artistic representation of turntable scanning

2.1.2 - Alternative Scanning Solutions

Impact and sonar based mapping is the cheapest solution but has the lowest resolution. This only provides a 2D map and is also susceptible to large amounts of drift. As well as this it struggles to identify non-solid obstacles, oblique obstacles, low lying surface changes and cliffs.

LIDAR is one of the best options for small robotic platforms. These modules usually come in the form of small turntables which spin at around 1 - 3Hz[4] which provide a high accuracy, dense 2d point cloud. However it is also the most expensive scanning solution listed.

Camera motion has the simplest hardware implementation as all it requires is a camera and a processor. However this method requires a fast processor to capture and analyse data in real time. As well as this the platform has to be moving to build up a map of its surroundings. This poses severe problems when the environment is initially unknown.

Structured Light scanners such as the Microsoft Kinect provides an excellent real time scanning solution. The main drawback with most off the shelf systems though is the physical size and minimum scan distance. The Kinect has a minimum scan distance of around 50cm[3]. Mounting this to a robotic platform would create a one meter dead zone which would cause issues when navigating close-quarter obstacles.

2.2 - A* Route Planner

The A* route inspection algorithm was coined by Peter Hart, Nils Nilsson and Bertram Raphael of Stanford Research Institute back in 1968. Originally this was an extension of Dijkstra's algorithm.[5] To build an A* route inspection algorithm in 2D space, each position needs to be classified as a node within a graph with weighted connections to each adjacent area. This can then be searched until the path connects the source with the target node.

Using a square grid is the most common way to convert a 2D environment into a route inspection graph. However because diagonal movements are longer than cartesian movements, the edges connecting them are weighted slightly higher. A common estimation of diagonal weighting is set to a factor of 1.4
The figure to the right shows a representation of this weighting factor.

14	10	14
10	0	10
14	10	14

A* uses an iterative approach for determining which path to explore. It does so by sorting each partial path by the cumulative path heuristic. The algorithm aims to minimise the following equation:

Known

$n = The\ last\ node\ on\ the\ path$

$g(n) = cost\ from\ source\ to\ n$

$h(n) = heuristic\ estimation\ of\ cost\ from\ n\ to\ target$

Unknown

$f(n) = minimisation\ function$

Relationship

$$f(n) = g(n) + h(n)$$

The heuristic estimation is implementation specific. The most common heuristic used on 2D environment grids is the real world distance. This can be estimated using the manhattan or euclidean distance.
On each iteration of the algorithm, it chooses the next reachable node with the lowest $f(x)$ value and removes it from the open set. Once chosen, the neighbours f and g values are updated and these neighbours are added to the open set. This is repeated until the goal node is in the open set. This gives us the minimum distance from the source to the goal.

To rebuild the path, the algorithm needs to be adjusted slightly to include node parenting. Each node's parent is set to the node which it was put in the open set by. This allows us to follow the nodes parents from the goal back to the source. Reversing this path will give us the shortest path from source to target.

2.3 - SLAM

SLAM stands for Simultaneous Location And Mapping. It's aim is to create a system to adjust a robot's position using landmarks in the real world[6]. This is to compensate for error in the robots built in odometry. It was originally developed by Hugh Durrant-Whyte and John J. Leonard based on earlier work by Smith, Self and Cheeseman[7]. The four main steps to SLAM are landmark extraction, data association, state estimation and state/landmark update. Once all these steps are complete, the robot has a new map of its surroundings and a new adjusted position. These steps can then be repeated for each new scan the robot takes.

2.3.1 - Landmark Extraction

Landmarks are physical attributes inside the environment which are easily identifiable and distinguishable. These help the robot triangulate its position when traversing the environment. These can be captured using scanners such as LIDAR, Sonar or impact based scanning.

2.3.2 - Data Association

Data association is the process where the same landmarks are extracted multiple times and therefore can be assumed to be the same physical point in the world. Ensuring that the landmarks are unique is crucially important as matching two different landmarks together will introduce error in the state update step. Other issues can be caused by landmark occlusion where a landmark is seen once and never seen again or is only seen sporadically.

2.3.3 - State Estimation

The state estimation formulates the robot's position using only the built in odometry of the drive system. This is regularly inconsistent and very susceptible to drift. For instance if the robot hit a small grain of sand and the heading changes by one degree, this would result in huge error in the robot's final position after a long traversal.

2.3.4 - State and Landmark Update

The state and landmark update step takes the robot's state and the landmarks from the landmark association steps and re-estimates the robot's state based on the landmark's relative position. This step is highly dependent on the Extended Kalman Mark's filter[28], however to reduce development time of this project this will be substituted by an

Iterative Closest Point matching algorithm. This step also updates the landmark's positions to minimise any error in the data association and landmark extraction stages.

2.4 - Iterative Closest Point

Iterative closest point is an algorithm used to align misaligned rigid point clouds together. The algorithm takes two point clouds, a source and a target. The target point cloud stays in place and is used as reference for the source point cloud to align to. The algorithm iteratively translates and rotates the source to minimise the difference between the two scans. This is known as the error metric.[8]
The error metric is calculated by iterating through each point in the source point cloud and measuring the distance to the nearest neighbor in the target point cloud. This also forms a set of vectors which can be used to estimate the transformation or the entire point cloud after filtering out statistical outliers. The output of the ICP algorithm is usually a 4x4 homogeneous coordinate transformation which maps the source's original position to its new iteratively refined position.

Figure 7 shows two scans which have been purposely misaligned. The orange scan is the source and the black scan is the target. We can see here how the ICP registration corrects this.

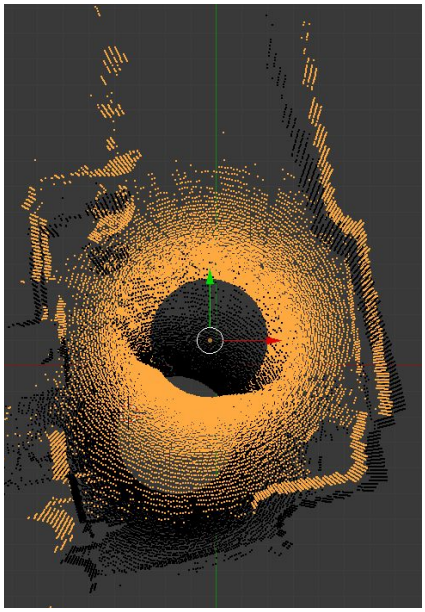


Figure 7a: Before Registration

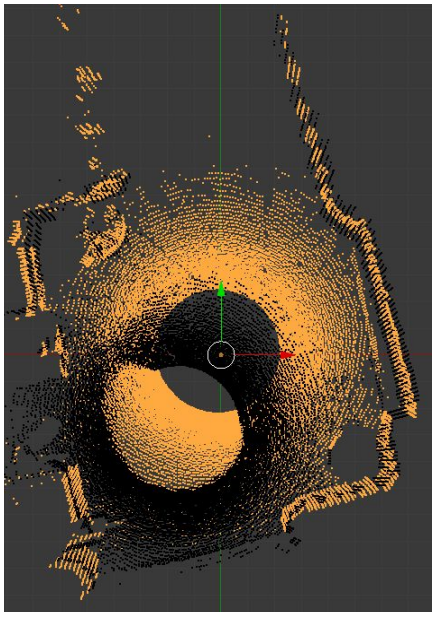


Figure 7b: After Registration

2.5 - Processor Choice

Processing solutions are usually a trade-off between price and computing power. Other factors to consider include power consumption, availability and physical footprint.

Arduino[19] is a popular brand of microprocessor. It has a small footprint, inexpensive to source and has a very low current draw. The main drawback of the Arduino is that it has very limited processing power. This makes 3D transformation and image analysis very slow. It also has limited connectivity lacking networking options and camera ports.

The Intel Galileo[20] has much better connectivity than the Arduino, however comes with a heftier price tag. The processing power is much closer to what is needed and is a well supported platform.

The Raspberry Pi[12] has great onboard connectivity with built in WiFi, bluetooth and ethernet ports. The processor is powerful enough to deal with most image analysis tasks and has a large support platform due to its popularity. It has a small footprint and supports many off the shelf modules such as webcams, Raspberry Pi cameras and a plethora of headers for the GPIO pins. However the Raspberry Pi doesn't support many real time and analogue operations such as pulse width modulation and sensor communication. Coming in at around £30 the Raspberry Pi is the perfect affordable processor of choice for this project.

Laptops can be built into these types of project, they provide excellent power and connectivity but have a large physical footprint and a larger price tag. Using a laptop would be suitable for robots navigating larger office spaces or exterior environments but is unsuitable for smaller interior environments.

The final processing option is to pair any of the above solutions with a host machine which could process the data then send it back to the platform. This would allow the footprint and the price to be driven down. This would allow a desktop machine or server to do the majority of the processing which would provide the maximum processing power. However this would move the bottleneck to the communication between the robotic platform and the host. This also introduces a reliance on a host machine which might not be available in remote locations.

2.6 - Motor Choice

The following motors could be used in this project however considering these points the stepper motor is the obvious choice for use with the Raspberry Pi.

DC motors need to be supplied with a constant current in order to rotate. Rough odometry can be achieved by supplying the voltage over time however this can fluctuate depending on the motor's load. If you want to change the speed you'll also need a DC motor driver to supply a constantly varying voltage supply.

Servo motors require a digital signal in order to rotate. They use a pulse where the frequency determines the rotation angle or speed of the motor. However this requires you to have some form of pulse width modulator.

Stepper motors[23] require a digital signal input over multiple channels. Each channel controls a coil inside the motor known as the phase. By iterating through a set of on-off steps you can rotate the motor to a high degree of accuracy depending on the number of channels and poles.

2.7 - Language Choice and OO Reasoning

Most 3D vision systems run off fast, low level programming languages such as C, C++ or C#. However, due to the development time restraints of this project, a higher level language is a better choice. Python is an excellent choice as development is fast and efficient. It is also the primary coding language the Raspberry Pi was developed for.[11][12] The main drawback is its low level processing speed. Therefore libraries and C wrappers are required to do some of the more intensive processing. I chose an object orientated approach to ensure each component of this project could be tested and developed independently. This proved vital for utilising my time efficiently by juggling development between modules.

2.8 - External Packages

The following packages provide functionality which Python commonly utilises for 3D manipulation and visualisation.

PCL[10] is a stand alone point cloud library for 3D point cloud processing. This provides ICP implementation as well as many other useful functionality such as statistical outlier filters.

Cython[13] is an optimisation library which wraps useful C extensions in Python. This is required for Python PCL.

Python PCL[14] is a Python wrapper for PCL. However this does require Cython and PCL to be installed. Installing this on the Raspberry Pi requires alteration of the Raspbian fstack protector, dphys swapfile and compilation pipeline.[15]

Numpy[16] is a scientific library which allows more advanced matrix calculations on images inside of python. This is also required for extracting images from the Pi-Camera quickly and efficiently using Pi Camera Array as it natively supports Numpy.

Pi-Camera[17] is a library that allows us to access the Raspberry Pi camera inside of Python. This also relies on Numpy to quickly extract images without encoding.

PIL[18] is an image processing library used for saving and exporting graphical representations of the world.

3 - System Overview

This chapter gives a brief overview of the entire system. Implementation of solutions defined below can be found under the hardware and software implementation chapters.

3.1 - Software Overview

The robot's task is to build a complete 3D map of an interior environment. It does so by compiling multiple scans at different locations within the environment.
These locations are chosen by scan voxelization. This allows volumetric areas of the environment to be analysed independently to assess the optimal location of the next scan.
Once this has been acquired, the robot uses the same voxelization analysis to safely traverse the environment from its current position to the new scan location using an A* route inspection algorithm. As soon as it cannot find a good location for the next scan it exits the program and returns a full 3D point cloud map of the environment.

This exploration loop can be represented as flowchart in Figure 8.

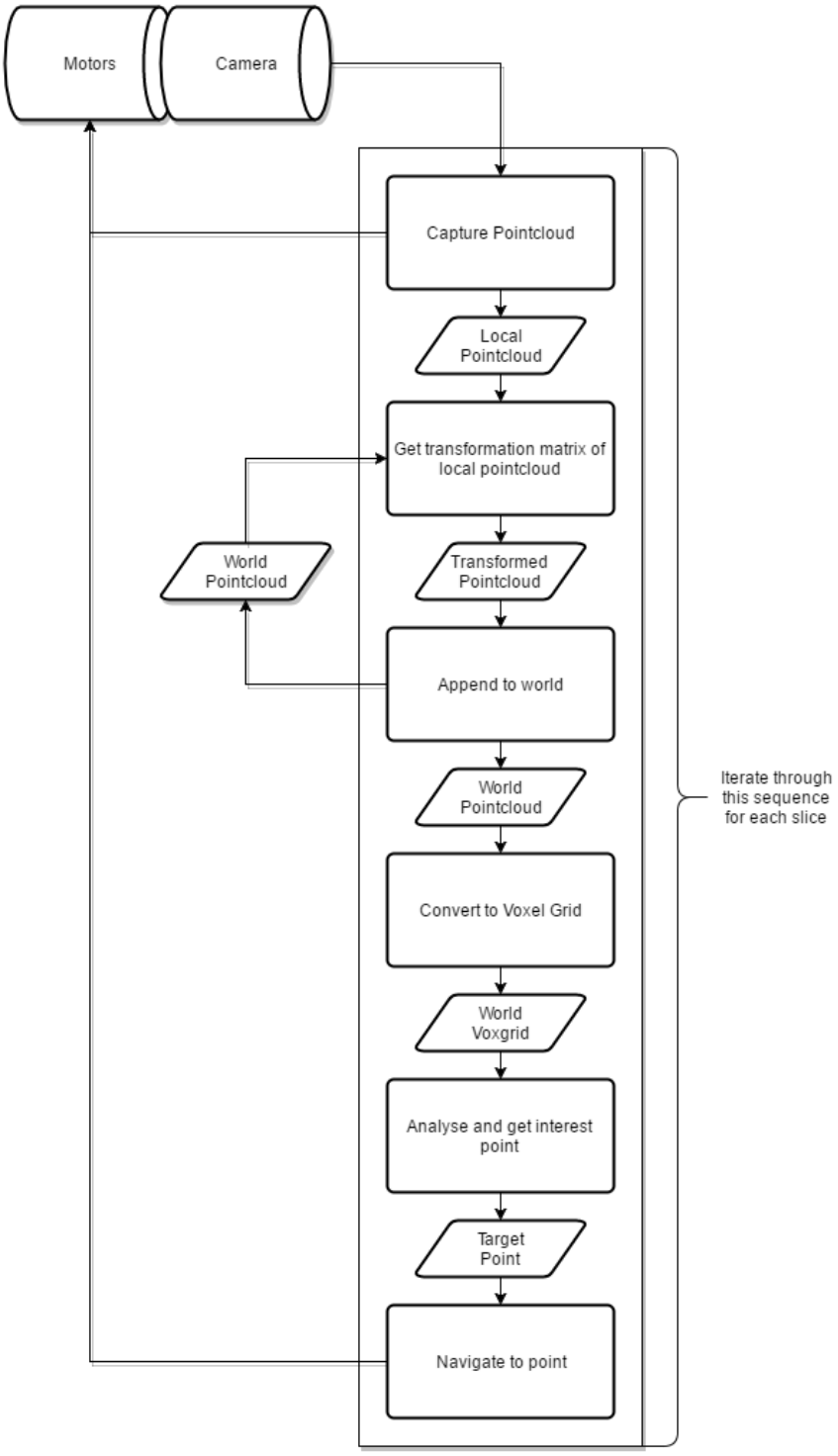


Figure 8: Flowchart of System overview

3.2 - Hardware Overview:

The robot is made of 3 main components. The scanner, wheels and processor. Figure 9 and 10 show the three main topological views and key components of the robot. Please note the power supplies pictured below are the secondary and tertiary supplies. The primary supply was in the form of a LIon USB power bank on the underside of the chassis, this powered the entire system. However, the LIon cells fulminated and therefore have been supplemented with additional supplies for testing and debugging.

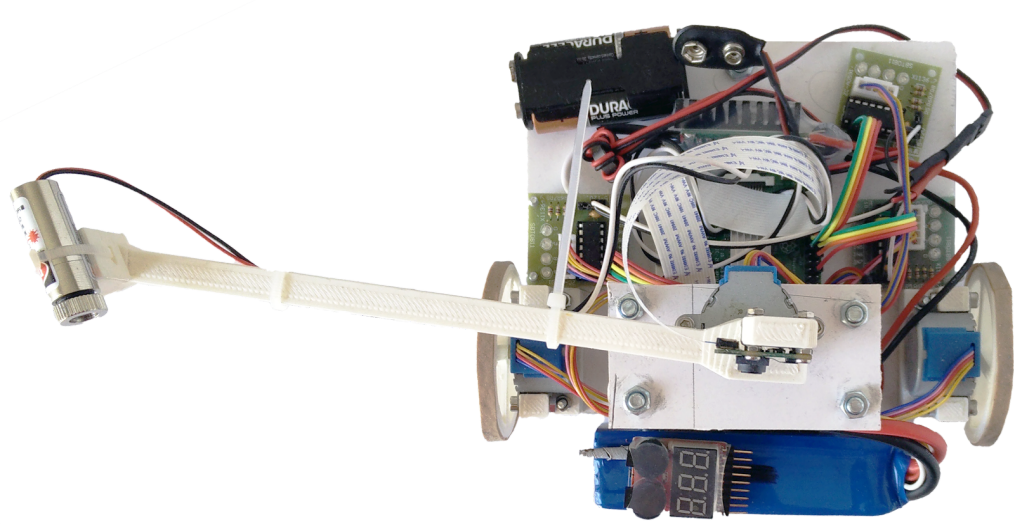


Figure 9a: Top view

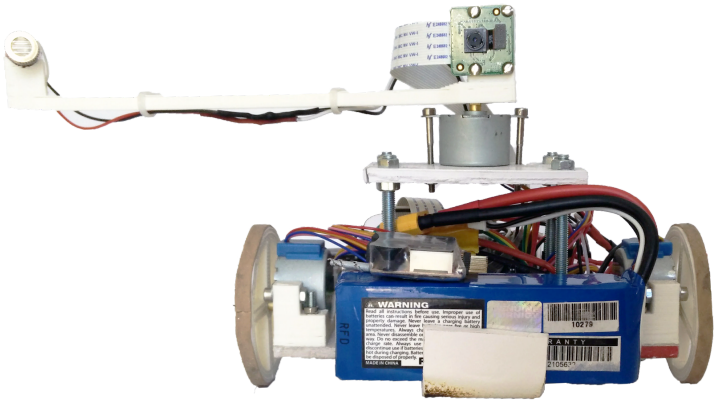


Figure 9b: Front view

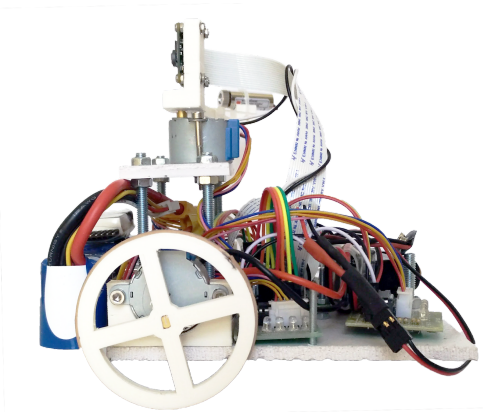


Figure 9c: Side view

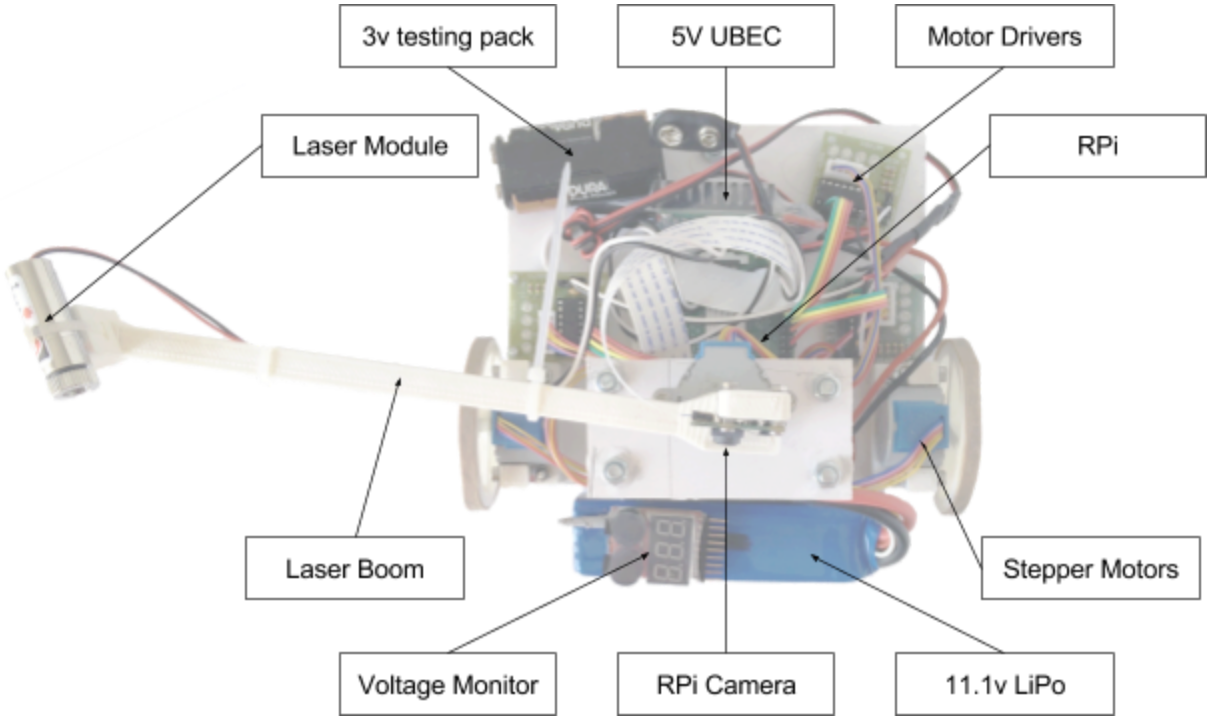


Figure 10: Top view with annotation

RPi - The Raspberry Pi forms the central processing unit for the robot.
RPi Camera - This is used to analyse the laser light emitted from the laser module and acts as the only input for the robot.
Laser Module - This emits the laser light which is picked up by the RPi camera.
Motor Drivers - These allow the low current GPIO signals from the Raspberry Pi to switch the high current 5v supply straight to the motors.
Stepper Motors - These are connected to the motor drivers and power the robot's behavior.
Laser Boom - This component attaches the turntable to the laser and camera module

11.1v LiPo - Secondary Lithium Polymer power supply.
Voltage Monitor - Ensures that the cells voltage in the LiPo do not drop to too low.
5V UBEC - This converts the 11.1v supply provided by the secondary LiPo to a 5v supply suitable for the Raspberry Pi and stepper motors.
3v Testing Pack - Tertiary debugging power supply for the laser module.

4 - Hardware Implementation

The entire hardware design is custom built for this project. It features an open-air design to allow rapid prototyping and development.

4.1 - Mechanical Parts

The aim of the chassis is to provide a rigid base for the components to be mounted to. High density foam board is an excellent choice for rapid prototyping as it can be milled and machined easily whilst still being rigid enough to support the robot's weight.

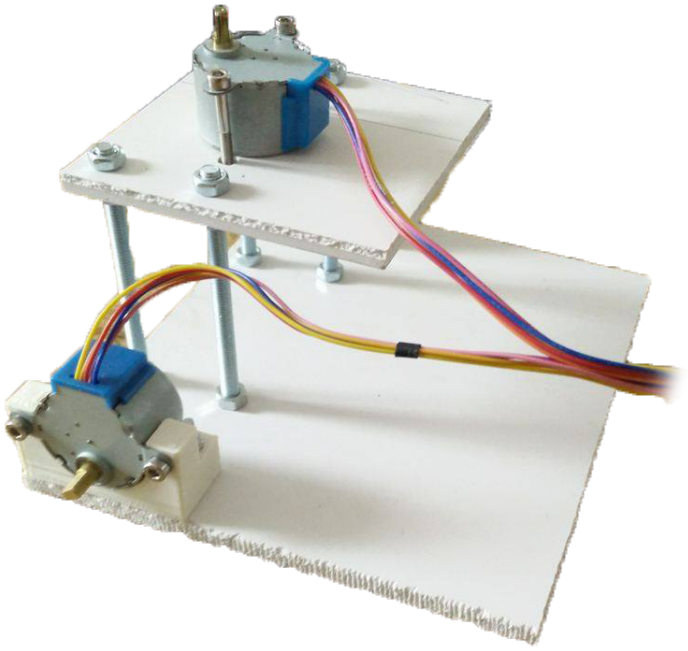


Figure 11: Corner view of chassis and motors

However 3D printing is a more suitable form of manufacture for more intricate parts such as wheels, motor mounts and boom arms. Fused deposition modeling printing is perfect for this application as it can print lightweight, rigid parts quickly which need very little finishing unlike other forms of printing.

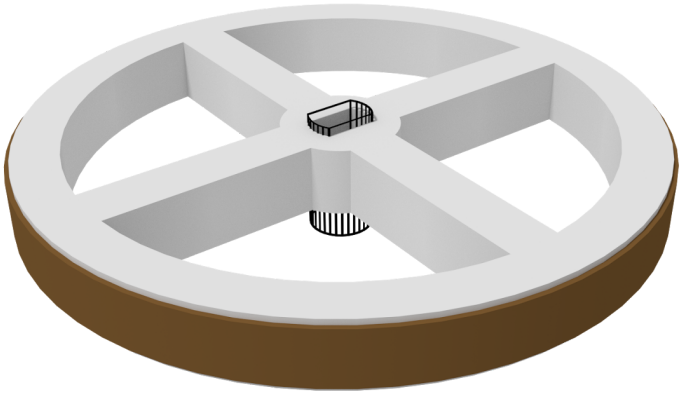


Figure 12: CAD model of wheel

4.1.1 - Boom Arm

The 3D printed boom arm was especially designed to ensure maximum rigidity could be achieved at minimal weight and could not have been achieved with classical manufacturing methods. The length of the arm and the angle of the laser has a direct correlation to the range of the scanner and the horizontal FOV of the camera. This will be explored further in the scanner chapter.

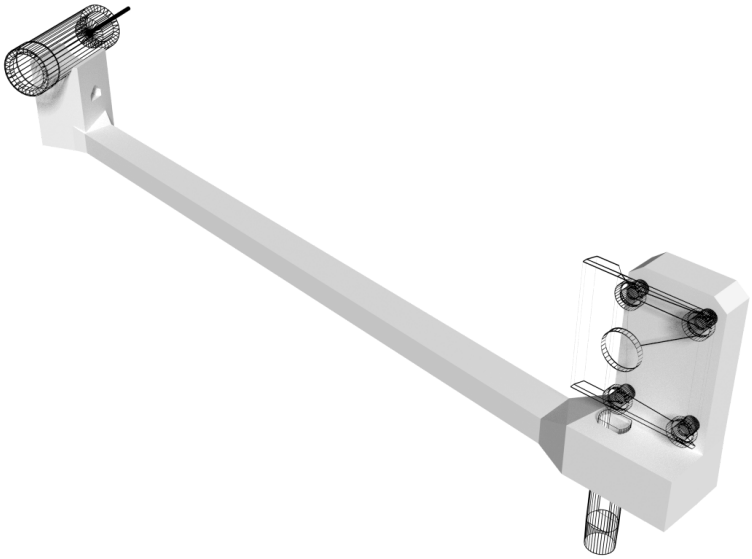


Figure 13: CAD model of boom arm

4.1.2 - Scanner Mount and Footprint

The footprint of the robot needed to be as small as possible to ensure it could navigate tight spaces as well as allowing the scanner to have a large field of view.

The dead zone circle is the area under the robot that is assumed to be safe when placed in a new environment. Mounting the scanner onto a raised platform increased the scannable area leaving a smaller dead zone.

To minimise the dead zone the immediate floor needs to be at the minimum range of the scanner. To calculate the optimal height of the scanner mount we run the following equations.

Known

$k = Closest\ scan\ distance$

$q_v = Camera\ vertical\ field\ of\ view$

Unknown

$h = Height\ of\ the\ scanners\ center$

Relationship

$$h = k(tan(\frac{q_u}{2}))$$

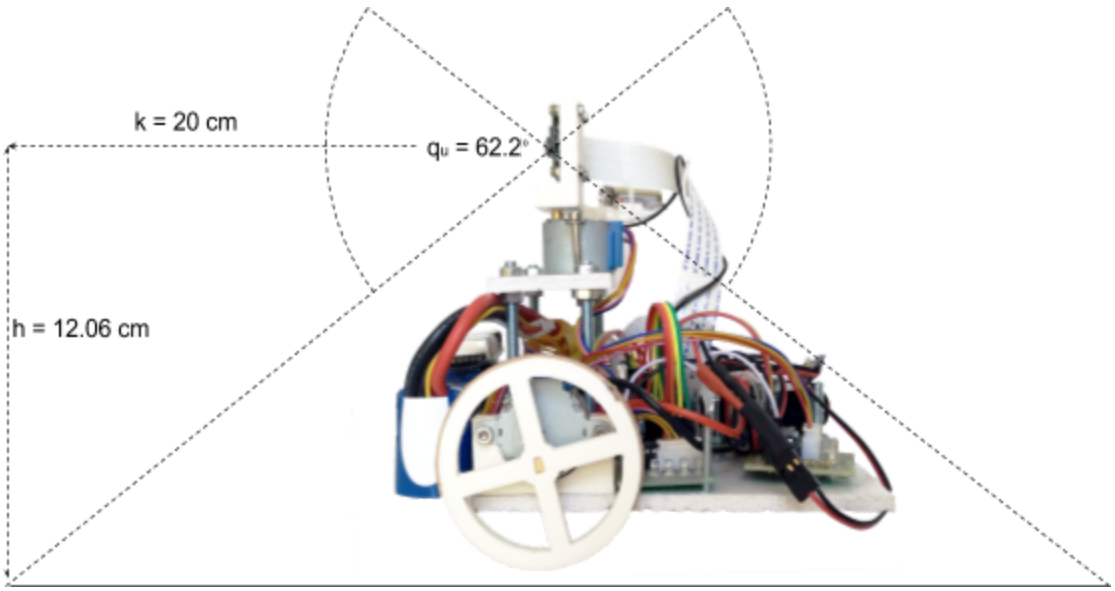


Figure 14: Diagram showing scan field of view

According to these equations and assuming the above boom is used, the optimal scanner height is 12cm as seen in figure 14. This was increased to 13 cm in the final build to accommodate for extra electronics and a slightly larger chassis; any lower than this and the immediate floor would be too close for the scanner to detect, therefore wasting scan field of view, any higher and the dead zone radius would increase.

4.1.3 - Wheels and Drivetrain

The movement mechanics are made using a simple built in gearing, 2 wheel and skid approach, reducing the number of components connected to the drivetrain. This increased the accuracy of the built in odometry as every part attached to the drivetrain adds mechanical play which reduces accuracy. This also simplified the odometry triangulation as the entire robot pivots around the center of the scanner. This is a very simple solution and is not suitable for rough environments however gives the highest motor odometry precision.

4.2 - Electronic Parts

Figure 15 shows a generic schematic of the entire electronics system. The single lines represent data and the double lines represent power.

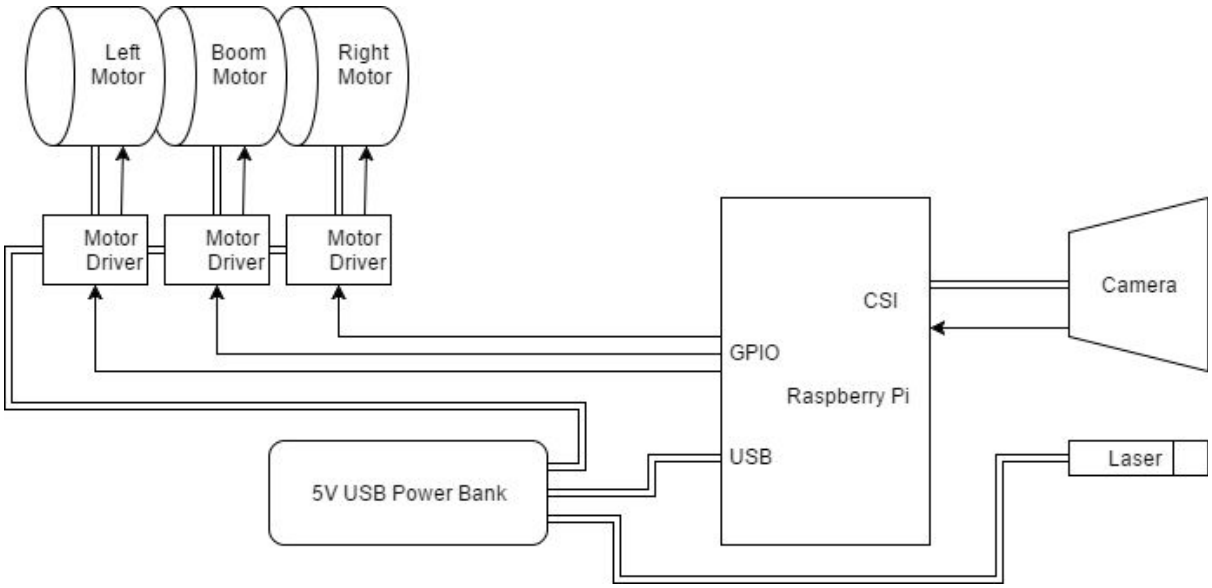


Figure 15: Schematic of electronics system

4.2.1 - Stepper Motors

Stepper motors require a digital signal input over multiple channels.[22] Each channel controls a coil inside the motor known as the phase. By iterating through a set of on-off steps you can rotate the motor to a high degree of accuracy depending on the number of channels and poles. A Darlington transistor array is required such as the ULN2803 however these are cheap and easy to implement using the Raspberry Pi's GPIO pins. It is also key to note that most stepper motors have significantly more poles per phase and usually have built in gearing. However the following examples assume a non-geared stepper motor with 4 channels and 4 poles.

Full stepping

Figures 16 and 17 show full stepping which charges each pole independently to rotate the motor.

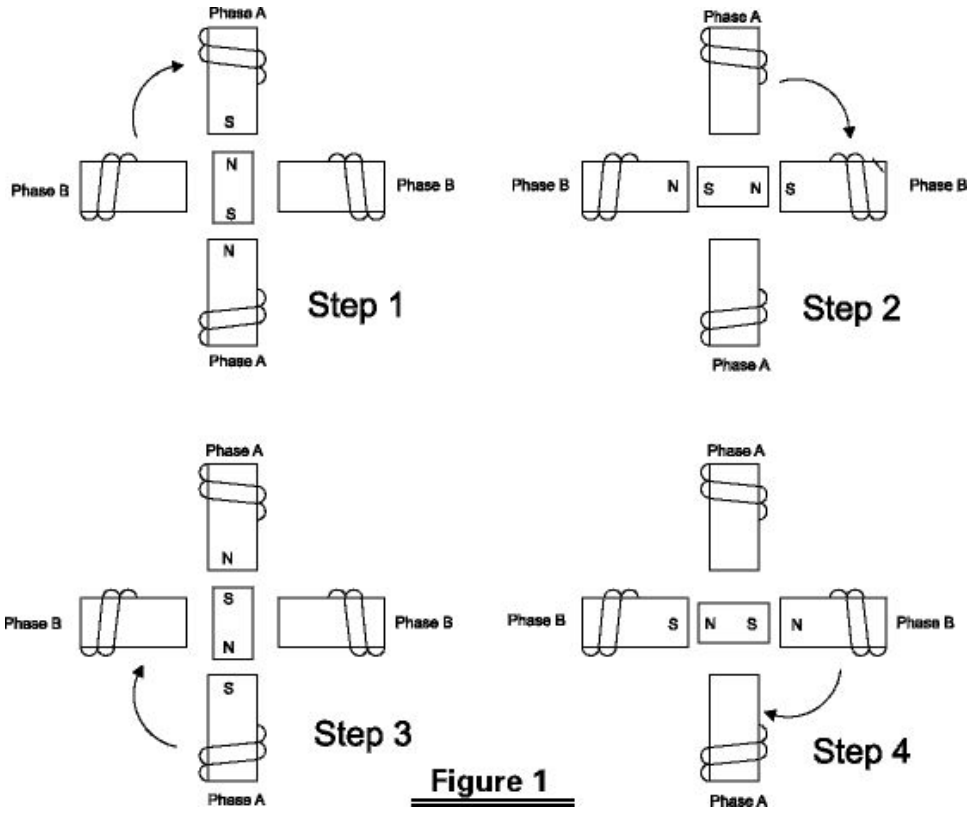
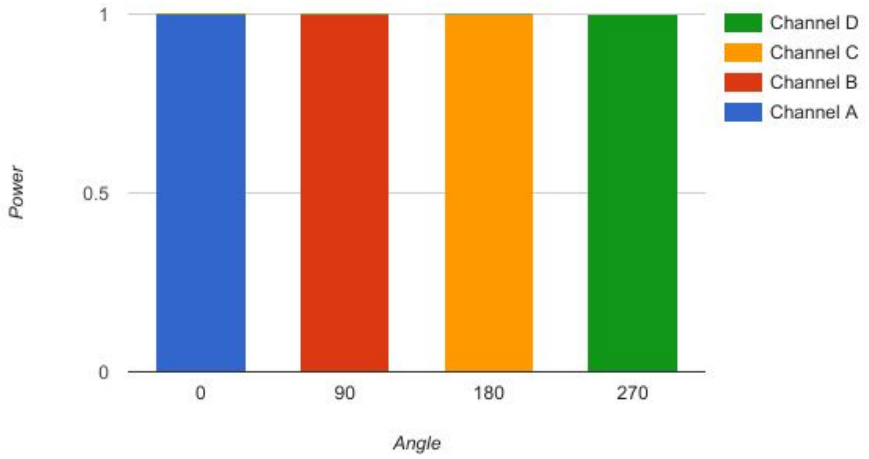


Figure 16: Diagram of stepper motor workings © Copyright pc-control.co.uk 2008[23]

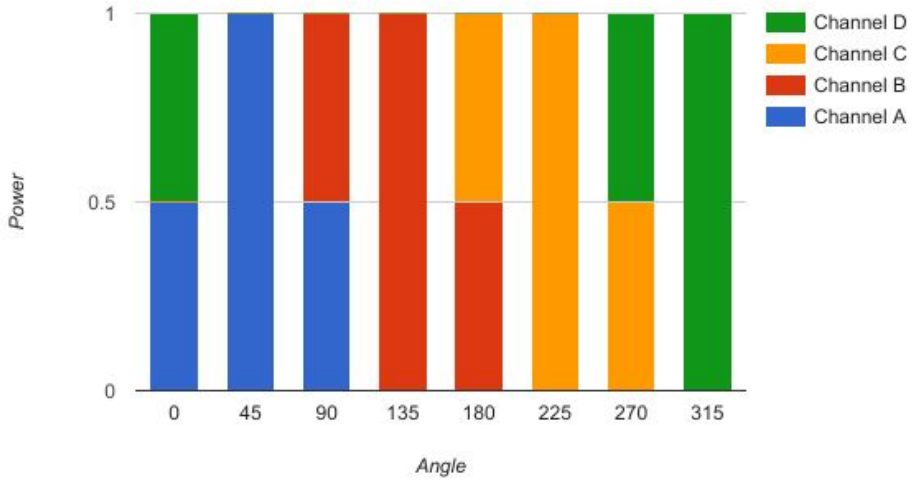



```
fullstep =
[
[1,0,0,0],
[0,1,0,0],
[0,0,1,0],
[0,0,0,1]]
```

Figure 17: Graph and code snippet showing full stepping

Half Stepping

Compared to full stepping, we can double the resolution by using half stepping where between each step we charge both poles of the adjacent coils to pull the core into a diagonal position. This provides us with the highest accuracy using a digital IO input and the stepping solution used in this project.




```
halfstep = [
[1,0,0,1],
[1,0,0,0],
[1,1,0,0],
[0,1,0,0],
[0,1,1,0],
[0,0,1,0],
[0,0,1,1],
[0,0,0,1]]
```

Figure 18: Graph and code snippet showing half stepping

Micro Stepping

To get an even higher resolution and if a semi-analogue output was available or a dedicated motor driver was used, a smoother and more accurate rotation could be achieved with micro stepping.

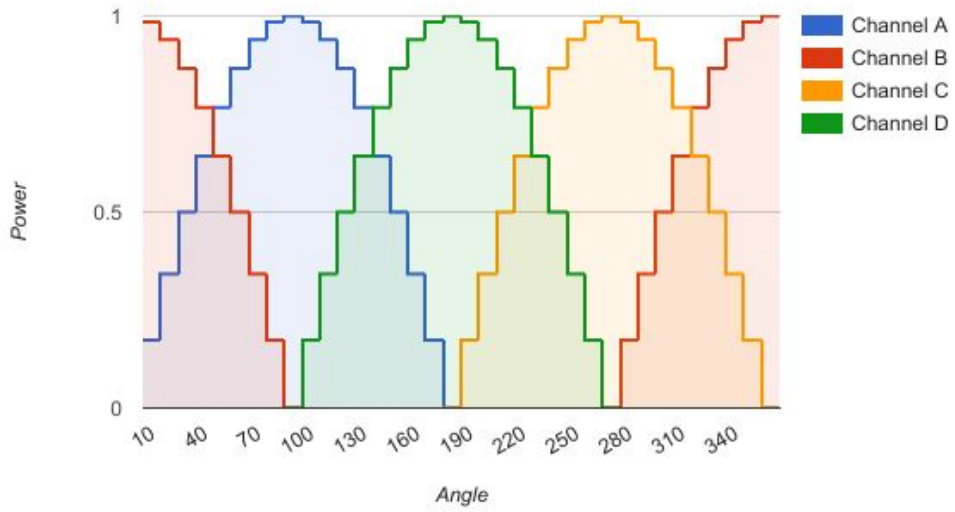


Figure 19: Graph showing stepper motor micro stepping

4.2.2 - Power Supply

Portable USB power packs are one of the most popular power supplies for Raspberry Pi based projects. This provides a clean 5v power supply to all electronic modules in the project. They are readily available and provide a self contained power system. Other power options include LiPo/LIon/NimH cells. The drawback to these systems is that a separate charging and switching circuit is required as most cells do not charge or provide 5v natively.

4.2.3 - Laser Module

Class 3R / 3A laser modules are readily available and inexpensive as they are commonly used for DIY levelling and pointing devices. However the main unique restriction of this component is its safety rating. Class 3R classification by the UK Government defines them as follows:
“The laser beams from these products exceed the maximum permissible exposure for accidental viewing and can potentially cause eye injuries, although the risk of injury is still low.”[24]
As this will be diffused using a line lens, this will reduce the risk of injury further and allow accidental viewing to be permissible.

5 - Software Implementation

UML

Figure 20 shows a UML diagram of all the custom written classes used in this software. The arrows represent a dependency. For instance the voxel grid class is dependent on the point cloud and voxel classes.

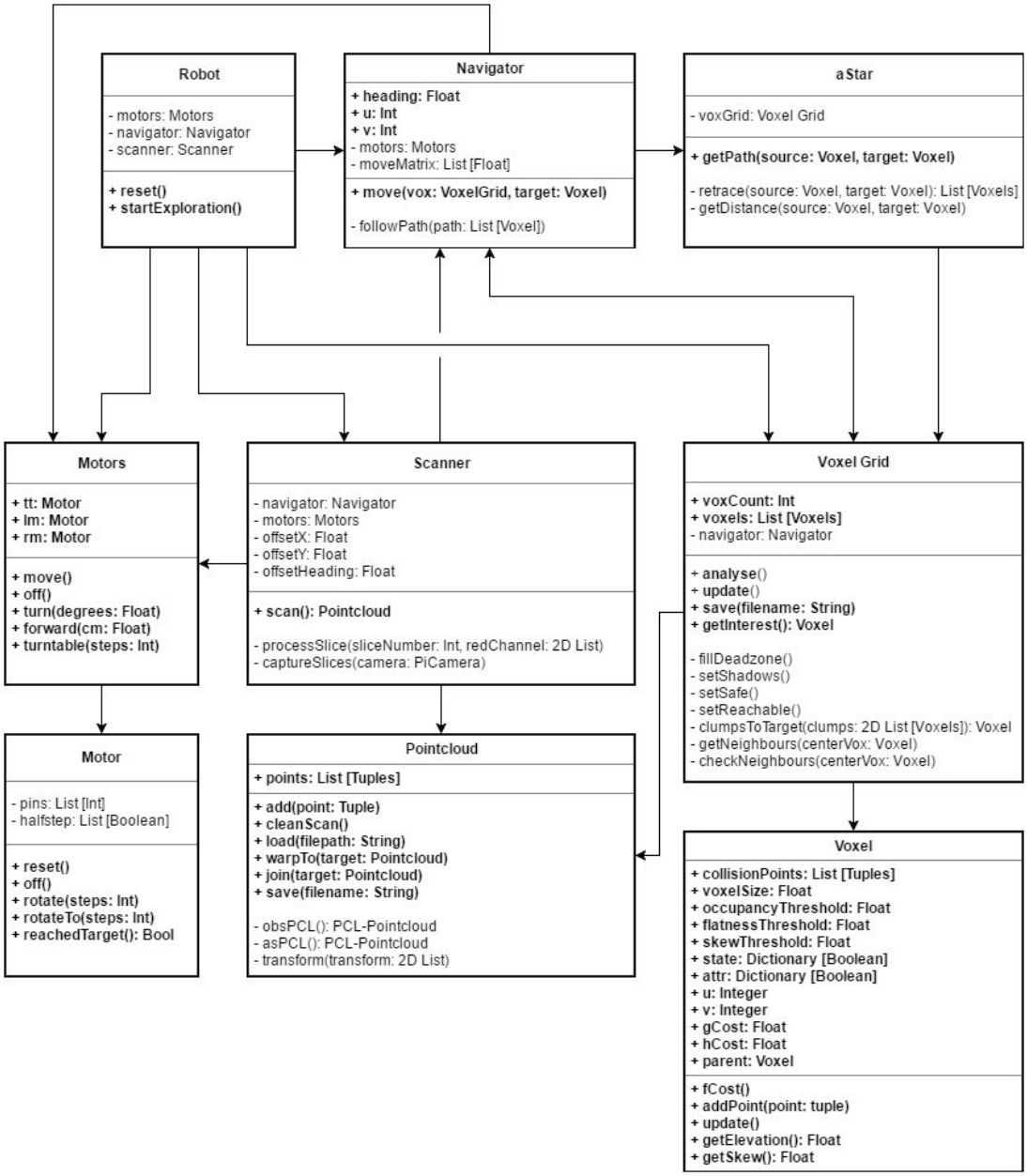


Figure 20: UML Representation of software

5.1 - Robot Module

The robot module holds the main control loop for the system. The robot navigates and explores with a set of commands which it iterates through until a stop condition is met. This forms the control loop. In this case this is when there are no more areas which the robot wants to explore. The only scan data that gets carried through each iteration is the world point cloud. This stores the robot's knowledge of its surroundings. A rough breakdown of the main control loop in Figure 21 is as follows:

- Scan the local environment
- Clean the scan to remove statistical outliers produced by noise
- Warp the local point cloud to the world pointcloud if the world pointcloud has data
- Add the warped local point cloud to the world environment.
- Convert the new world pointcloud to a voxel grid and analyse it
- Find an interest point in the voxel grid. If none exists exit out of the control loop.
- Otherwise navigate as close as possible to the interest point.

```
def startExploration(self):

        world = pointcloud()

        i = 0
        while True:

                local = self.scanner.scan()

                local.clean()

                world.warpTo(local)

                world.join(local)

                vox = voxGrid(world,self.navigator)

                vox.analyse()
                target = vox.getIntrest()

                world.save("worldSave_%i.obj"%i)

                if not target:
                        break

                self.navigator.move(vox,target)
                i += 1
```

Figure 21: Code Snippet from the main robot class

5.2 - Scanner Module

5.2.1 - Boom arm length and laser angle

The first challenge when developing the boom arm laser scanner is calculating the physical attributes of the boom arm. This is because the values for a, L and the camera's FOV q_u shown in Figure 22 directly affect the scan range of the robot. To calculate these values we need to input the maximum and minimum scan range (j, k) into the below equations.

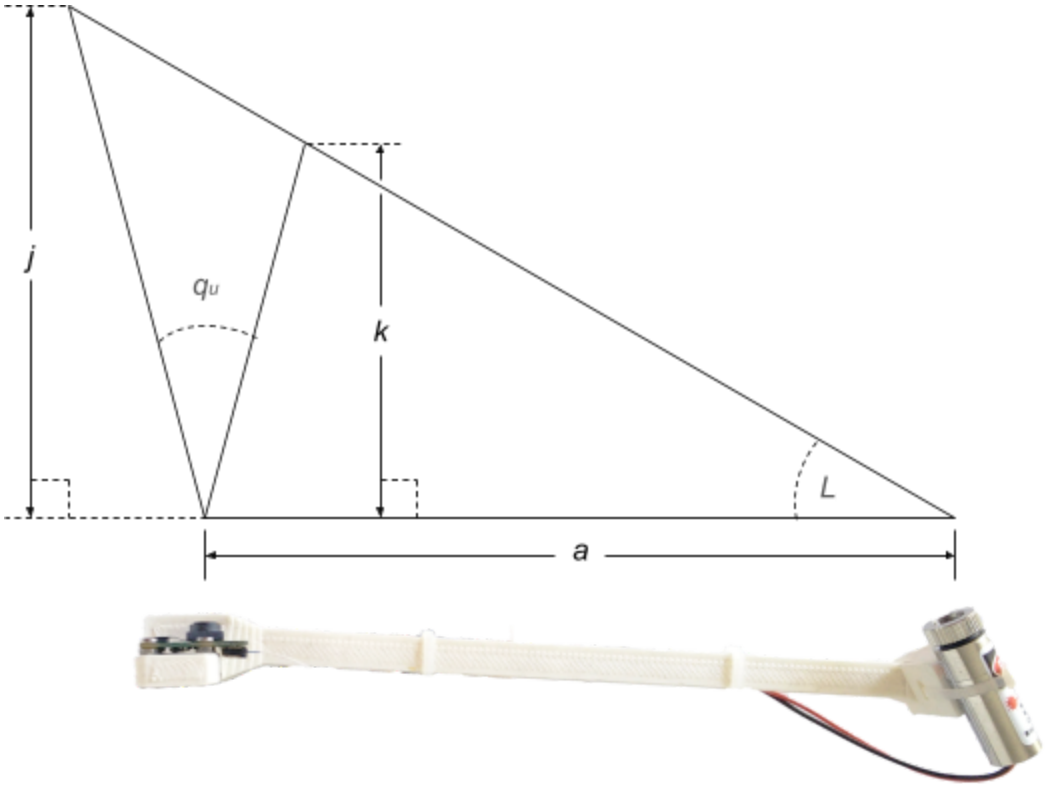


Figure 22: Diagram of Minimum/Maximum scan range

Known

j = *Furthest scan distance*

k = *Closest scan distance*

q_u = *Camera horizontal field of view*

Unknown

L = *Angle of laser*

a = *Length of boom arm*

Relationship

$$a = j(tan(90 - L) - cot(\frac{180 - q_u}{2}))$$

$$L = atan(\frac{(j-k)tan(\frac{180 - q_u}{2})}{j+k})$$

If we input the Raspberry Pi camera horizontal FOV q_u as 47° and the range (j, k) as 20 to 300 centimeter into these equations, the optimal and implemented boom design is to place the laser at 65° on a 17.5 centimeter boom arm.

Figure 23 shows a vertical board 20 centimeters away from the camera. As we can see the laser falls perfectly on the right most side of the image. This indicates that the above equations are correct.

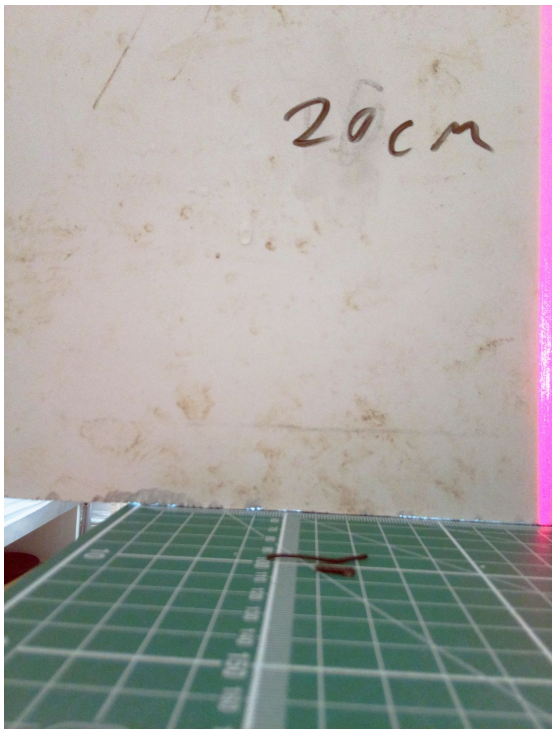


Figure 23: Image captured from the scanners camera

However these values must be conservative due to the inherent accuracy falloff. Figure 24 represents the loss in accuracy at longer distances. This is measured in centimeters of error a one pixel offset from the camera would cause. This assumes a Raspberry Pi camera with a horizontal resolution of 972 pixels is mounted to the above boom.

The two lines represent the accuracy of the x and y dimensions relative to the camera. The minimum and maximum scan range must be adjusted to take this accuracy falloff into account if a higher precision at longer ranges is needed.

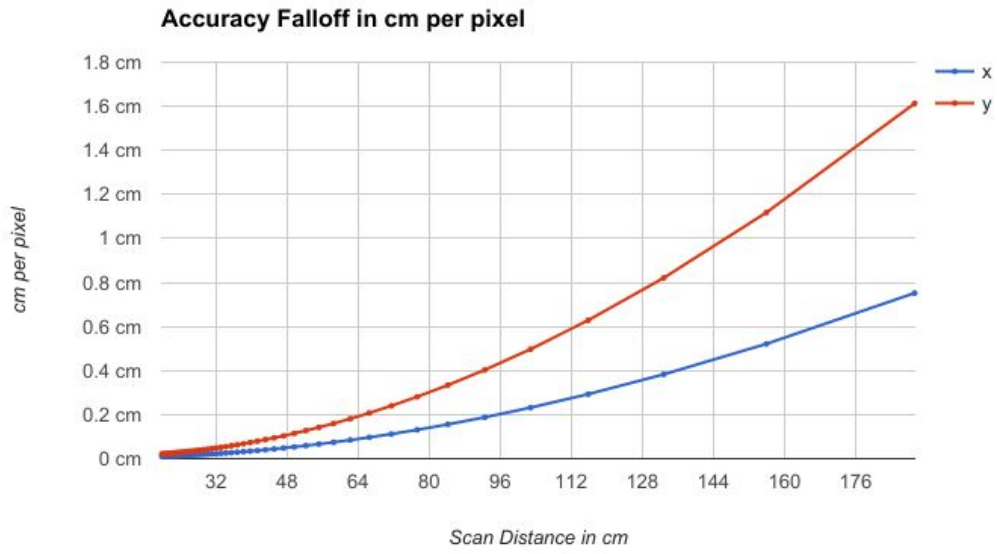


Figure 24: Graph representing accuracy falloff

5.2.2 - Coordinate systems

The two main coordinate systems used in this project are camera and world coordinates. Camera coordinates define a points position relative to the camera whereas the world coordinates define a points position relative to the world.

I shall be defining 3D points in the camera space as (x, y, z) and in the world space as (x', y', z'). Subscript u, v will also be used in the scanner module for referring to positions on the 2D camera plane. This should not be confused with the voxel grid positioning coordinates.

5.2.3 - Implementation of boom arm laser scanner

Figure 25 shows an overview on the process of converting the image captured by the scanner into a set of points.

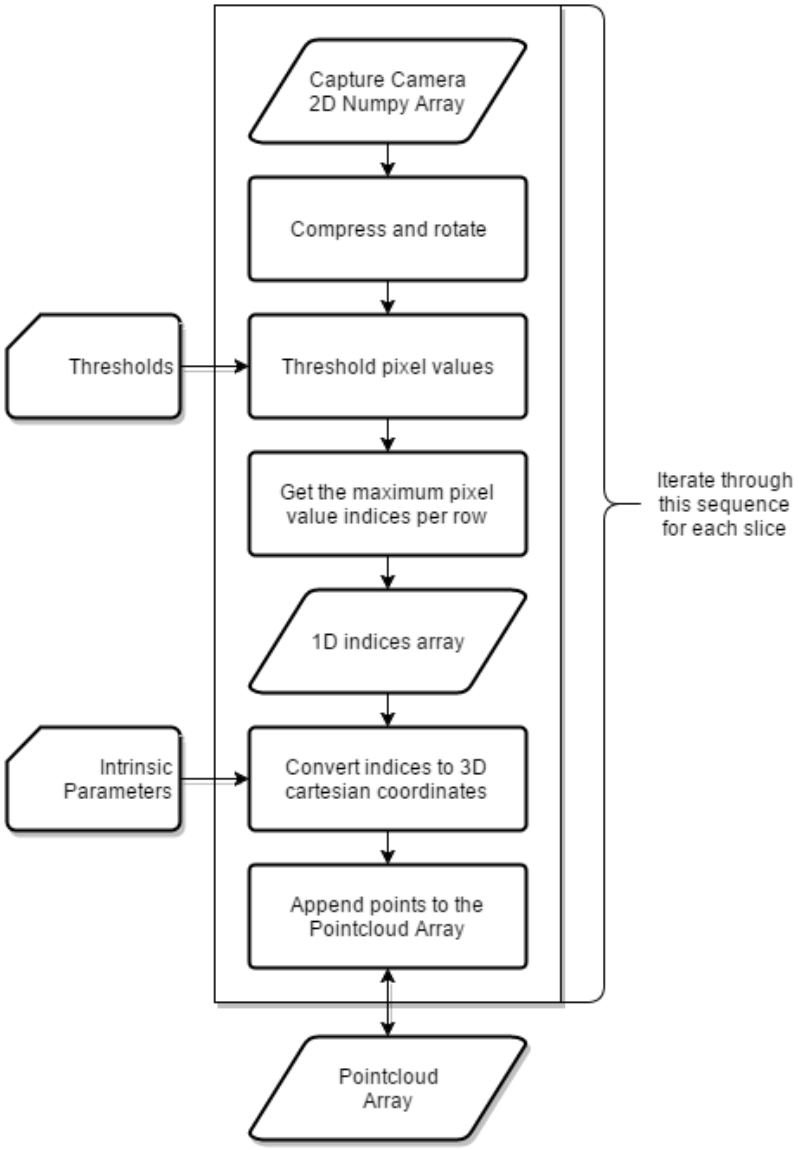


Figure 25: Flow chart of point cloud extraction from image data

The following equation is used to convert each image slice returned by the camera to points in world space using basic trigonometry. As each horizontal row of the image represents a single point this equation is applied every nth row depending on the point cloud density required.

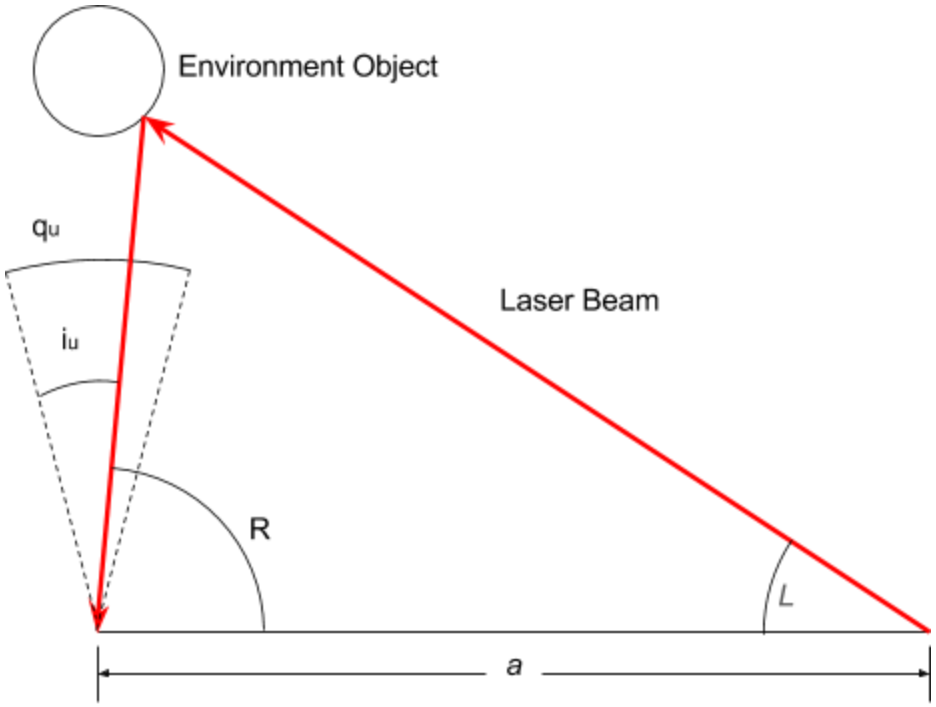


Figure 26: Trigonometry diagram for calculating R

To calculate the position of the environment object we first need to find R to ensure we have a solvable angle-side-angle triangle.

To find R we first need to convert i_u into degrees. When analysing the image this will be returned as a pixel index and therefore needs to be divided by the horizontal FOV in pixels p_u . This will return a value between 0 (far left) and 1 (far right). To change this to degrees we simply multiply it with the camera's horizontal FOV in degrees q_u. This forms the equation $\frac{i_u q_u}{p_u}$

We can define the maximum value of R as $\frac{\pi}{2} + \frac{q_u}{2}$ assuming that the camera is at a tangent to the laser. To find the value of R simply subtract i_u in degrees from the maximum response to form $\frac{\pi}{2} - \frac{i_u q_u}{p_u} + \frac{q_u}{2}$

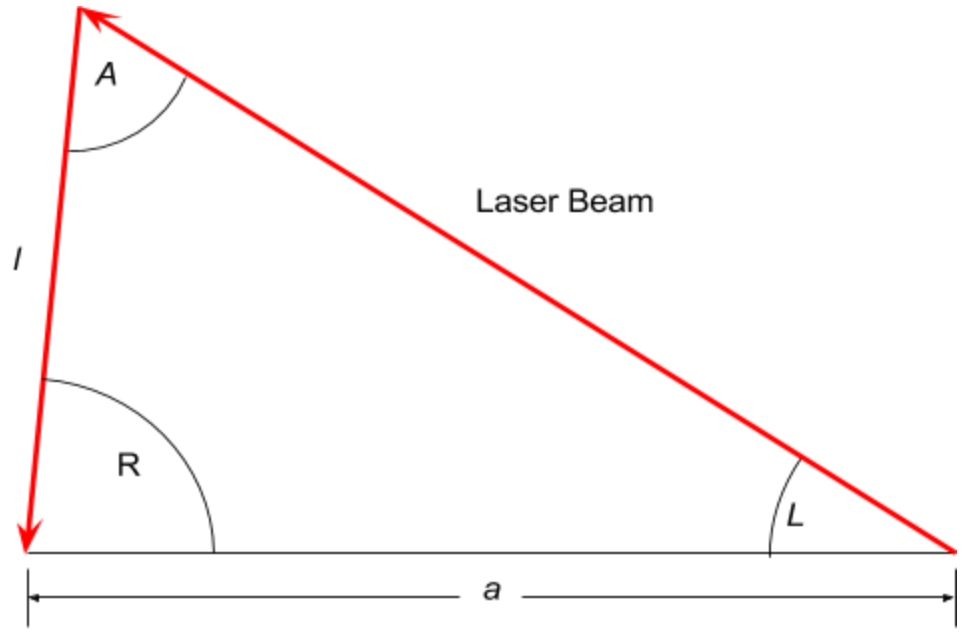


Figure 27: Trigonometry diagram for calculating l

To calculate the (x, y) position of the environment object we also need the distance l. This is calculated using the sine rule.

$\frac{a}{sin(A)} = \frac{b}{sin(B)}$

A can be calculated using the 180 triangle rule $A = \pi - (R + L)$

Therefore $\frac{l}{sin(L)} = \frac{a}{sin(A)}$ which can be simplified as $l = \frac{a(sin(L))}{sin(A)}$ and if we substitute in A

$l = \frac{a(sin(L))}{sin(\pi - R - L)}$

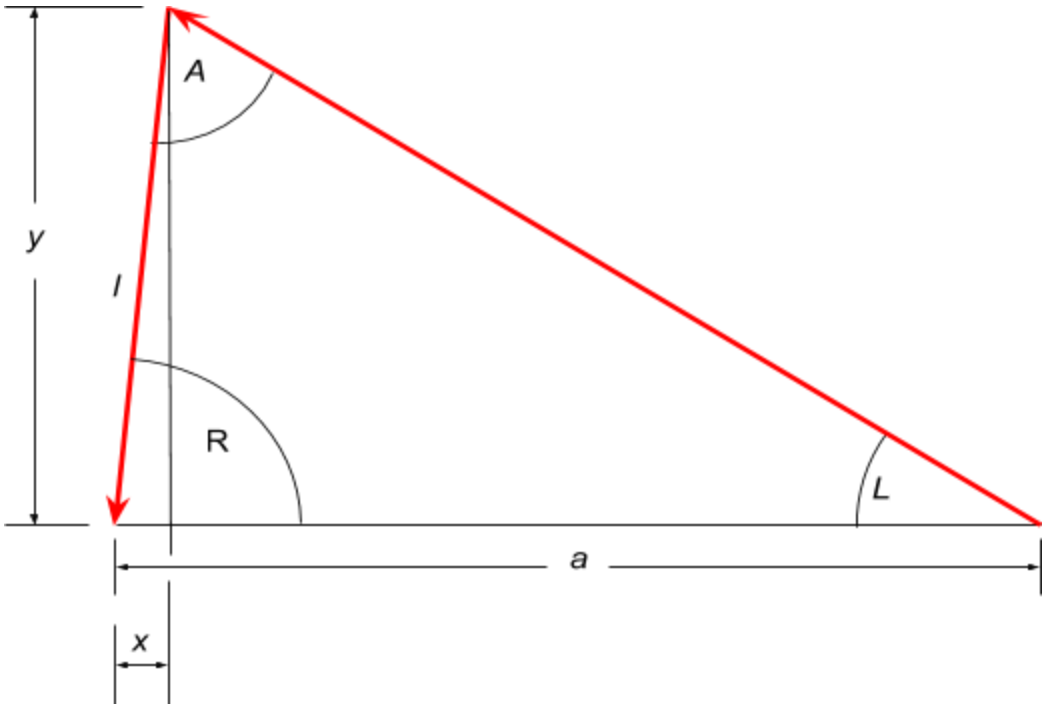


Figure 28: Trigonometry diagram for calculating (x,y)

Calculating (x,y) relies on the right angle triangle rule $sin(\theta) = \frac{o}{h}$ $cos(\theta) = \frac{a}{h}$ $tan(\theta) = \frac{o}{a}$
We can define $cos(R) = \frac{x}{l}$ and $sin(R) = \frac{y}{l}$ which can then be simplified to

$$x = l(cos(R))$$
$$y = l(sin(R))$$

To calculate z we first need to calculate the angle i_v in a similar way to i_u.

When analysing the image this will be returned as the vertical pixel index i_v and therefore needs to be divided by the vertical FOV in pixels q_v. This will return a value between 0 (top) and 1 (bottom). To change this to degrees we simply multiply it with the camera's vertical FOV in degrees q_v. This will return the angle of i_v and can be defined as $\frac{i_v q_v}{p_v}$

To calculate z we simply find the angle between the horizontal and the laser as $\frac{qv}{2} - \frac{i_v q_v}{p_v}$
Then use the $tan(\theta) = \frac{o}{a}$ rule to calculate $z = y.tan(\frac{q_v}{2} - \frac{i_v q_v}{p_v})$

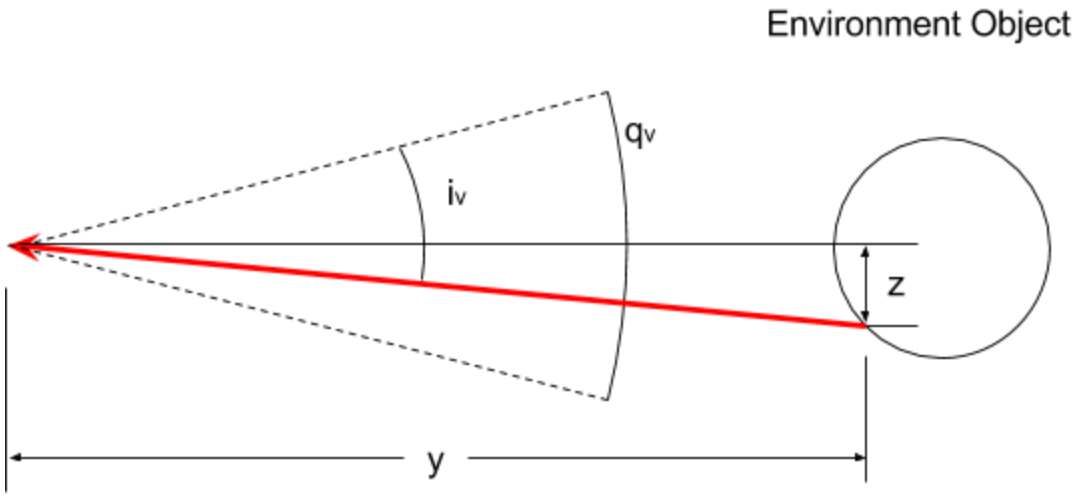


Figure 29: Diagram for calculating z

This gives us the (x, y, z) coordinate in camera space. To convert this to world space we can use a rotational matrix to include the turntable's rotation θ.

$$x' = x.cos(\theta) - y.sin(\theta)$$
$$z' = x.sin(\theta) - y.cos(\theta)$$

As we are rotating around the z axis, we only need to transform the x and y positions of the point.

Combining all of these equations we get the following calculation:

Known

i_u = *horizontal index of pixel being analysed*
i_v = *vertical index of pixel being analysed*
s_x = *robot's x position*
s_y = *robot's y position*
θ = *angle of turntable*
H = *robot's heading*

a = *boom length*
L = *laser angle*
p_u = *camera horizontal FOV in pixels*
p_v = *camera vertical FOV in pixels*
q_u = *camera horizontal FOV in degrees*
q_v = *camera vertical FOV in degrees*

Unknown

(x', y', z') = *point location world space*

Relationship

$$R = \frac{\pi}{2} - \frac{i_u q_u}{p_u} + \frac{q_u}{2}$$

$$l = \frac{a(sin(L))}{sin(\pi - R - L)}$$

$$x = -l(cos(R))$$

$$y = l(sin(R))$$

$$z = y.tan(\frac{q_v}{2} - \frac{i_v q_v}{p_v})$$

$$x' = x.cos(\theta + H) - y.sin(\theta + H) + s_x$$

$$y' = x.sin(\theta + H) - y.cos(\theta + H) + s_y$$

$$z' = z$$

Figure 30 shows a code snippet from the slice processor in the scanner module. This is an implementation of the above calculation.

```
def processSlice(self,i,red):

        redSubsampled = red[:,::ip.stepSize]
        maxIndexes = redSubsampled.argmax(axis=0)

        for subsampled in range(0,len(maxIndexes)):

                iu = maxIndexes[subsampled]

                iv = subsampled*ip.stepSize

                if redSubsampled[iu,subsampled] > ip.threshold:

                        theta = radians((i*360)/ip.numberOfSlices)+self.offsetHeading

                        R = pi/2 - (iu*ip.qu)/ip.pu + ip.qu/2
                        l = (ip.a*sin(ip.L))/sin(pi - (R+ip.L))

                        x = -l*cos(R)
                        y = l*sin(R)

                        z = -1*y*tan(ip.qv/2 - (iv*ip.qv)/ip.pv) # bit of a flip hack

                        xPrime = x*cos(theta) - y*sin(theta) + self.offsetX
                        yPrime = x*sin(theta) + y*cos(theta) + self.offsetY
                        zPrime = ip.zAdjuster(y,z)

                        point = (xPrime,yPrime,zPrime) #Saves full scan
                        #point = (xPrime,yPrime,z) #Saves un-adjusted z scan
                        #point = (x,y,z) #Saves camera space scan. Useful for calibration

                        self.pc.add(point)
```

Figure 30: Code snippet of the slice processor

5.2.4 - Camera resolution

The resolution of the camera is critical to the calculations and efficiency of the robot. The Raspberry Pi Camera has a native resolution of up to 2592x1944[25] however the resolution has a direct correlation with speed and sensitivity. If we define the scanner's real world accuracy to ~1mm, we can reduce the resolution drastically without affecting the scan precision. If we reduce the resolution to half (1296×972) we can increase the capture rate by a factor of 2.8. Even when rotating the camera into portrait to increase vertical FOV q_v the 972 levels of depth still gives us enough resolution.

We can also bin the pixels to improve the camera's sensitivity in low light conditions[26]. This process reduces the resolution by averaging pixels in 2 by 2 bins instead of point sampling which increases the sensitivity by a factor of 4. This has a very low processing overhead as binning is natively supported on the RPi's GPU.

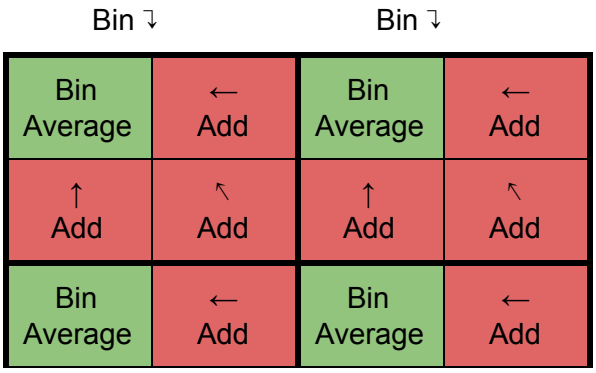


Figure 31: Pixel binning diagram

5.2.5 - Calibration and cleaning

Laser Twist

To ensure that the values of the intrinsic parameters are correct the scanner needs to be calibrated. One key calibration step is to ensure the laser is vertical. To do this we place the scanner in front of a flat wall and run a modified version of the scanner module. This returns the lasers horizontal difference in pixels between the vertical center and the quarter above. Figure 32 shows a capture from the scanner with an un-calibrated laser.

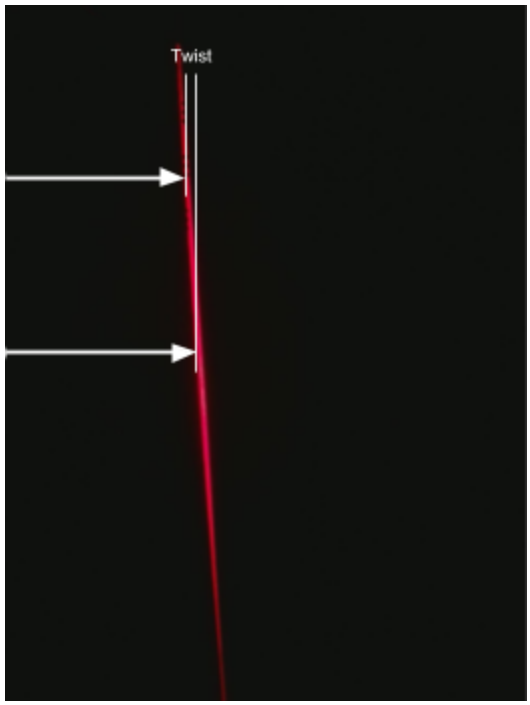


Figure 32: Representation of twist

```
def getTwist(self,img):

        r,g,b = img.split()
        redData = list(r.getdata())
        iv = int(ip.pv/2)
        rowData = redData[iv*ip.pu:iv*ip.pu+ip.pu]
        iu, value = max(enumerate(rowData), key=operator.itemgetter(1))
        iv2 = int(ip.pv/2-ip.pv/4)
        rowData = redData[iv2*ip.pu:iv2*ip.pu+ip.pu]
        iu2, value2 = max(enumerate(rowData), key=operator.itemgetter(1))
        if value2 > ip.threshold:
                print("laser twist: %f"%(iu-iu2))
```

Figure 33: Code snippet of modified scanner class

FOV Adjustment

Other calibration includes adjusting the camera FOV to ensure that measurements are accurate. Figure 34 shows a side view of points from a scan plotted in camera space. The black point cloud is before calibration and the yellow point cloud is after. We can see that the yellow scan data is much flatter and closer to the absolute truth. This shows the importance of good vertical FOV calibration as it can affect the floor's angle and therefore its traversability.

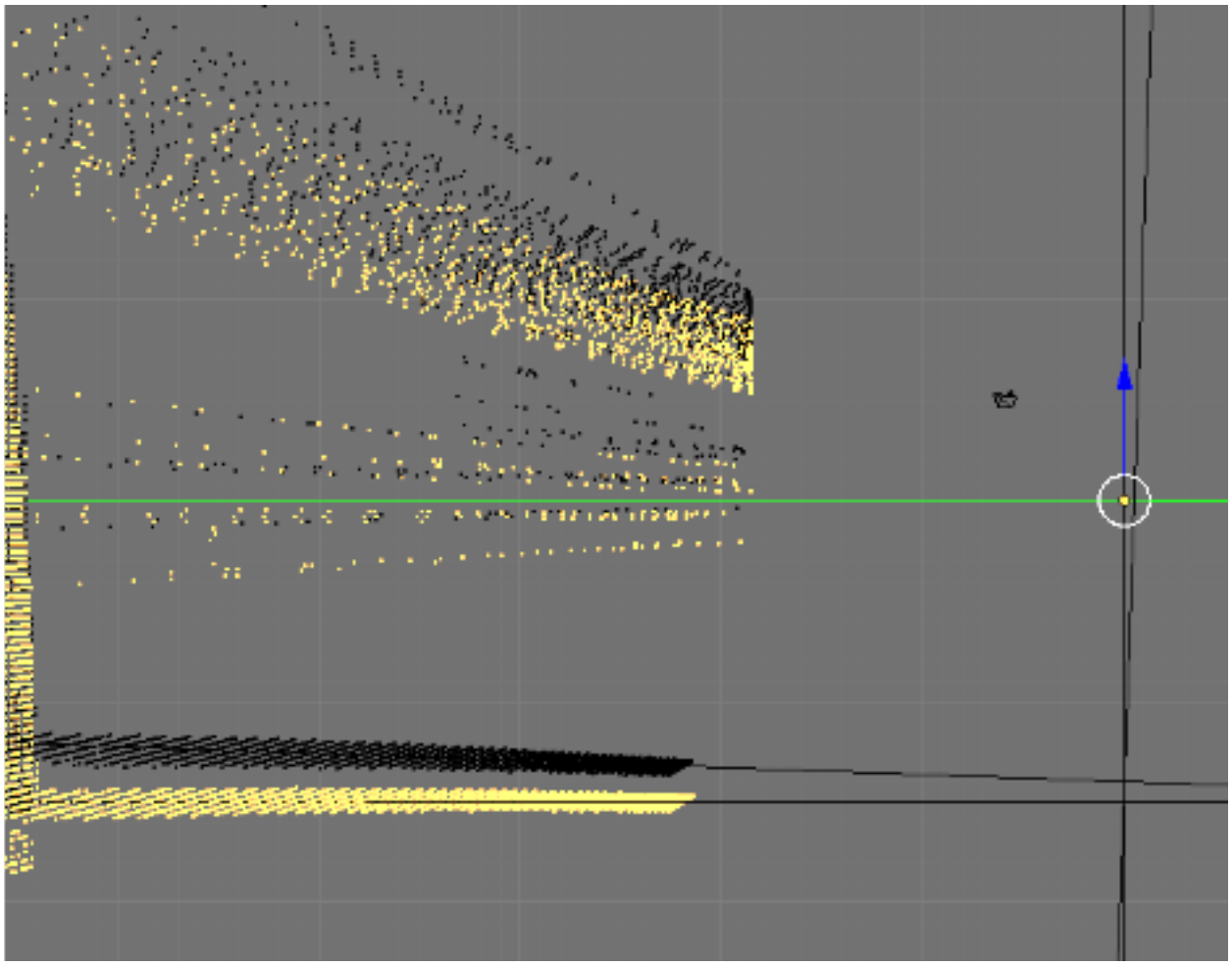


Figure 34: Pointcloud side view showing before/after FOV calibration

Figure 35a and 35b show the importance of horizontal FOV adjustment. The black lines here represent the absolute truth of the environment.

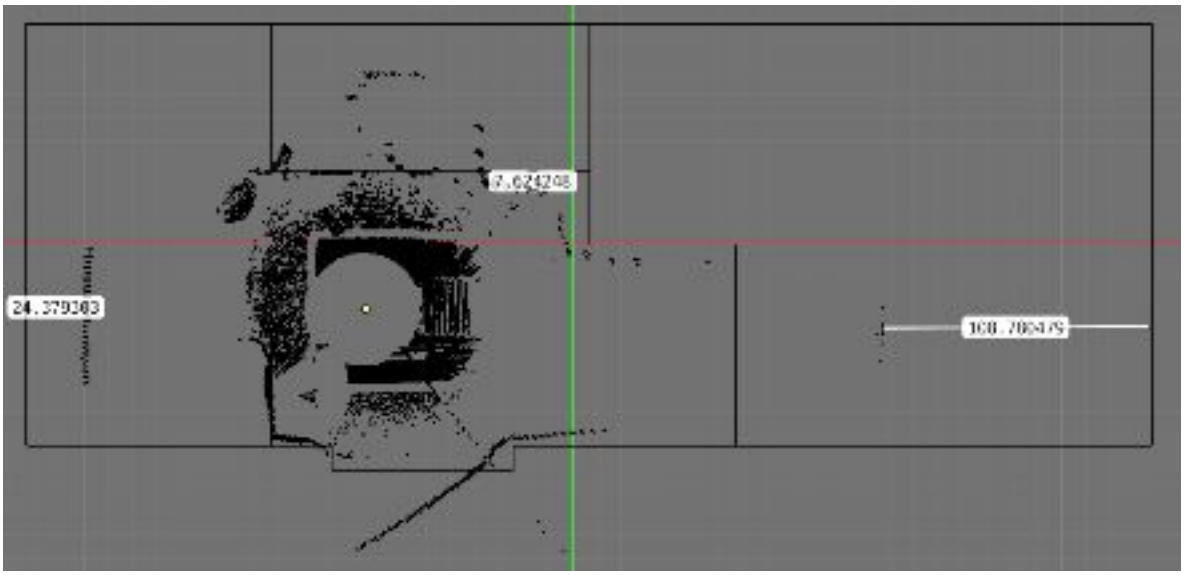


Figure 35a: Pointcloud showing horizontal FOV adjustment required in cm

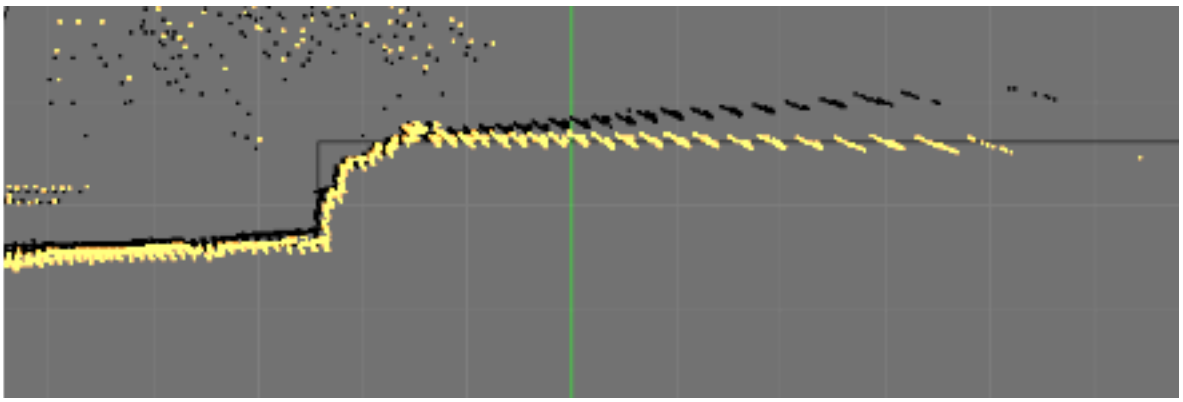


Figure 35b: Pointcloud top view showing before/after FOV calibration

Thresholding

As well as the intrinsic parameters such as the FOV, other variables need to be adjusted such as the mean_k value in the statistical outlier filter and the brightness threshold in the scanner. Figure 36 compares the vertex count of a scan versus the minimum brightness threshold used by the scanner.

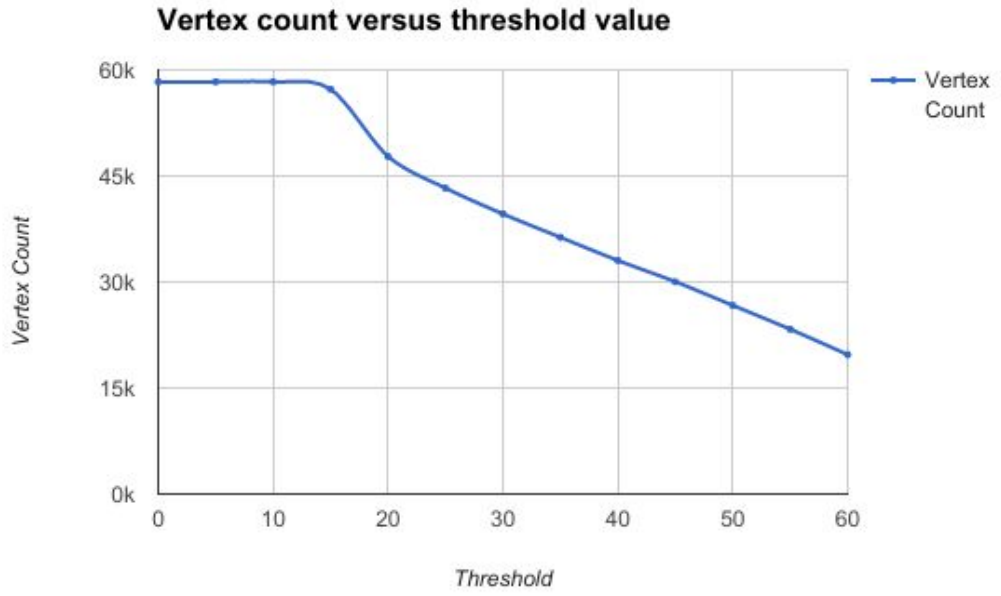


Figure 36: Graph of vertex count compared to brightness threshold

Floor snap

To assist in the voxel grid traversal algorithm we can also snap the floor response to remove noise. Figure 37 shows a side view of a scan that defines any point with a z value between two thresholds as floor and therefore snaps it to the average of the thresholds. This does however remove low lying obstacles such as thin wires or coins and therefore not ideal. This could be improved by creating an auto-calibrator which levels the floor automatically to adjust for minute changes in the hardware.

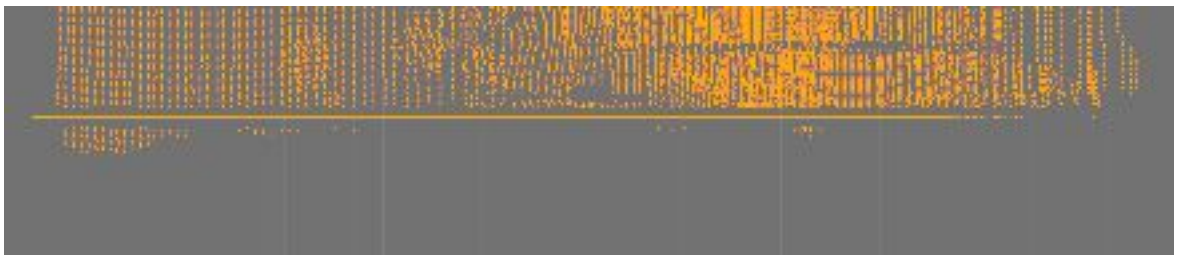


Figure 37: Pointcloud side view demonstrating floor snapper function

5.3 - Point Cloud Module

This module forms the data structure for the points generated by the scanner. It is instantiated for each point cloud the robot generates and also holds the world view point cloud. The main function this module is to provide efficient registration between point clouds. This contains modified Python Wrappers for the Point Cloud Library.

5.3.1 - Z axis lock

One key PCL function the point cloud module alters is that it locks all Z axis transformation when generating registration transformation matrices. It does so as it is assumed that the robot will only travel on a flat plane. Figure 38 shows an extreme example of bad registration with no Z axis lock.

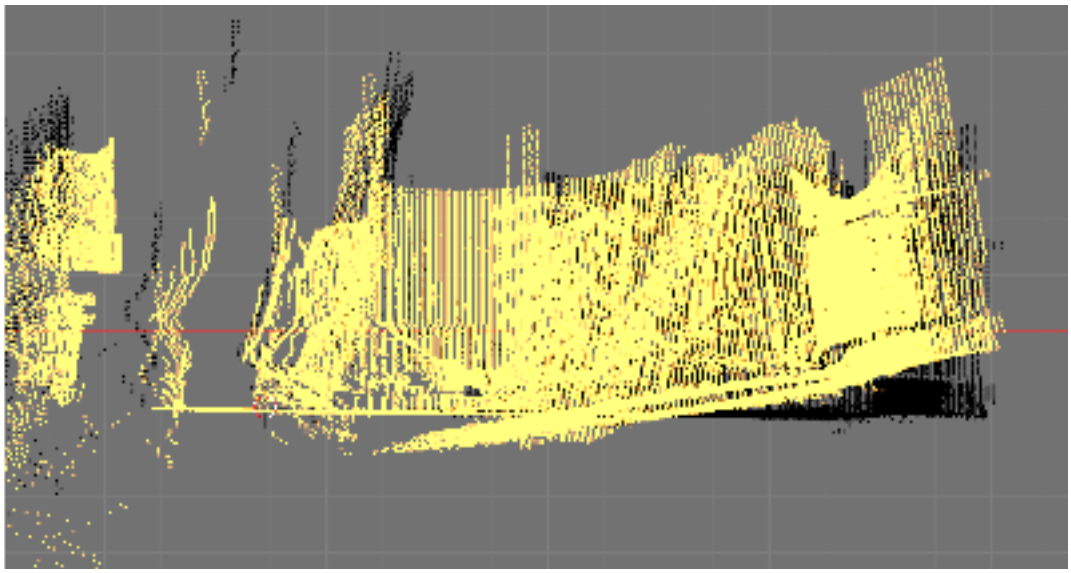


Figure 38: Extreme example of poor registration with no z axis lock

The Point Cloud Library ICP registration algorithm returns a homogenous transformation matrix similar to the one in Figure 39.

```
[[  7.07109809e-01   7.07109809e-01   1.53409928e-07   2.43779823e-07]
 [ -7.07109392e-01   7.07109272e-01   4.34304859e-08   2.82843518e+00]
 [  2.31105972e-07  -2.41568785e-07   1.00000346e+00  -9.56030476e-07]
 [  0.00000000e+00   0.00000000e+00   0.00000000e+00   1.00000000e+00]]
```

Figure 39: Screenshot of 4x4 homogenous transformation matrix

To remove all Z axis transformation we nullify the first, second and fourth values in the third row of the matrix. This ensures that each point's Z value is locked and cannot be transformed. This modified transformation matrix is then applied to the point cloud.

.707	.707	.00000015	.000000244
-.707	.707	.0000000434	2.83
~~.0000002.31~~	~~-.000000241~~	1.00	~~-.000000956~~
.000	.000	.000	1.00

Figure 40: Table of values in Figure 45 after z locking

Figure 41 shows a monkey head which has been rotated around the X, Y and Z axis. A disk of points at the base of the object has then been added to demonstrate the Z axis transformation lock.

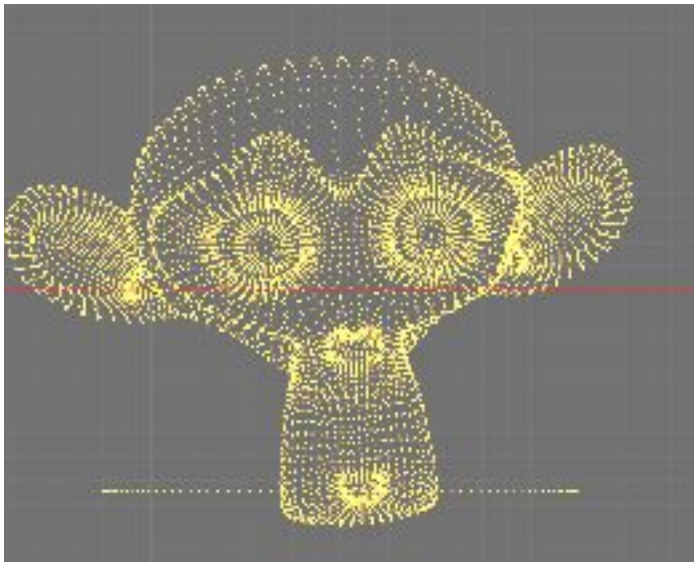


Figure 41: Point cloud of transformed monkey head

This has been run through the registration algorithm in the point cloud module to try and match it with the original un-transformed monkey head. As we can see in Figure 42 the X, Y and Z rotation has been corrected as much as possible without altering each points Z axis position. This is verified by the horizontal disk of points at the base of the monkey which have been unaffected by the transformation.

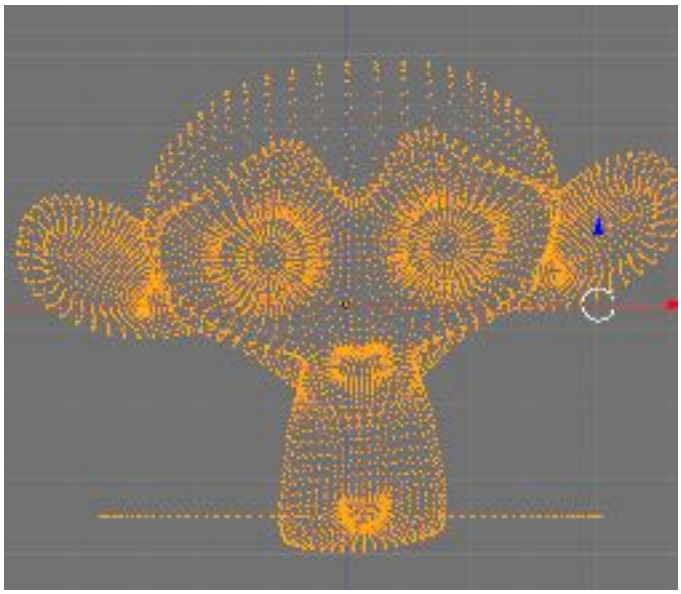


Figure 42: Point cloud after transformation correction

```
def transform(self,transf):
        print("Transforming pointcloud locking down the Z axis")
        newPoints = []

        transf[2] = [0,0,1,0]

        for point in self.points:
                homogenous = (point[0],point[1],point[2],1)
                newPoints.append(np.dot(transf,homogenous)[:-1])

        self.points = newPoints
```

Figure 43: Code snippet of Z locking function

5.3.2 - Registration convergence and fitness

However the point cloud only gets transformed if the registration converges and has a high fitness value. If it fails to do so it will transform the point cloud by the rough odometry returned by the navigation module only. This is to ensure that incorrect registrations are disregarded. In the future this could be fed back into registration with a higher iteration depth incurring a longer but more accurate registration step.

5.3.3 - ICP Non-commutative Complexity

The complexity of the ICP implementation by PCL is also non-commutative. Matching a small point cloud to a large point cloud is much faster than the inverse. Therefore another function of the point cloud module is to ensure that the most efficient registration is being made regardless of point cloud size. If the registration target is smaller than the source, it switches the target and source arguments of the ICP function and inverts the homogenous transformation matrix returned. This converts the target to source transformation to a source to target transformation.

```
if len(self.points) > len(target.points):
        converged,transf,e,fitness = pcl.registration.gicp(target.obsPCL(),self.obsPCL(),20)
        transf = np.linalg.inv(transf)
else:
        converged,transf,e,fitness = pcl.registration.gicp(self.obsPCL(),target.obsPCL(),20)

if converged and fitness < 1000:
        self.transform(transf)
else:
        print("Scan has not converged. will not warp to target")
```

Figure 44: Code snippet of registration optimisation

5.4 - Voxel and Voxel Grid Modules

The point clouds generated by the scanner have a high amount of data but little usable information about the world. To gain more information about the world we can reduce the resolution of the scan binning together points into a 2D voxel grid. We can reduce this as the robot only moves in 2 dimensions and therefore doesn't require vertical data. Figure 45 shows how the point data is binned.

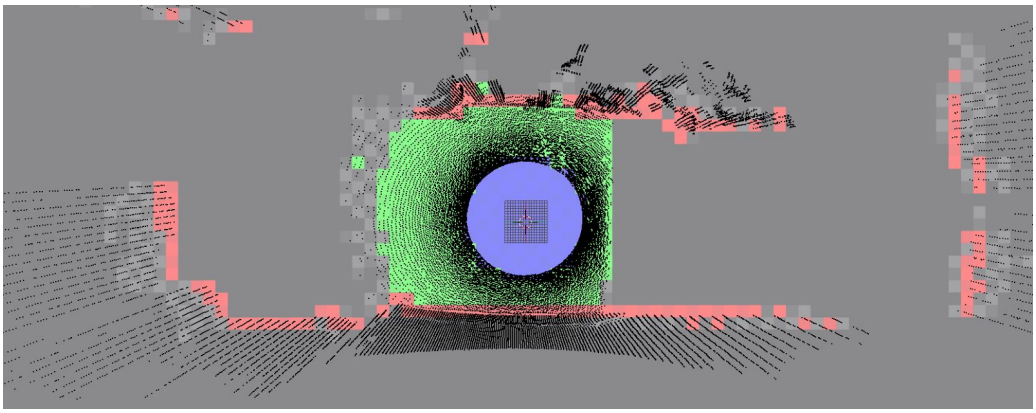


Figure 45: Point cloud with voxel map overlay with perspective FOV

Each voxel holds 2 main datasets, state and attribute. State only gets updated when the voxel has points added to it. It is an independent data set and only reliant on its own points. In comparison, the attribute data set holds information about the voxel which is reliant on external sources and therefore can only be updated by the voxel grid.

```
self.state = {"scanned":False,"occupied":False,"obstacle":False,"skewed":False}

self.attr = {"shadow":False,"explore":False,"safe":False,"reachable":False}
```

Figure 46: Code snippet from the voxel class

5.4.1 Voxel States

Figure 47 show some of the key voxel states.

Scanned
This states that the voxel contains any amount of points. This is updated as soon as a point is added.

Occupied
This states that the voxel contains enough points to make a reliable decision. The threshold is set as points per cm^2 to ensure that adjusting the voxel size does not affect the state.

Obstacle
This states that the maximum difference in point height is above the traversable threshold. This threshold is also set as change in height per cm.

Skewed
This states that the average position of all points is too far from the center of the voxel. This is to ensure that voxels with only partial data such as cliff edges are not set as safe or traversable.

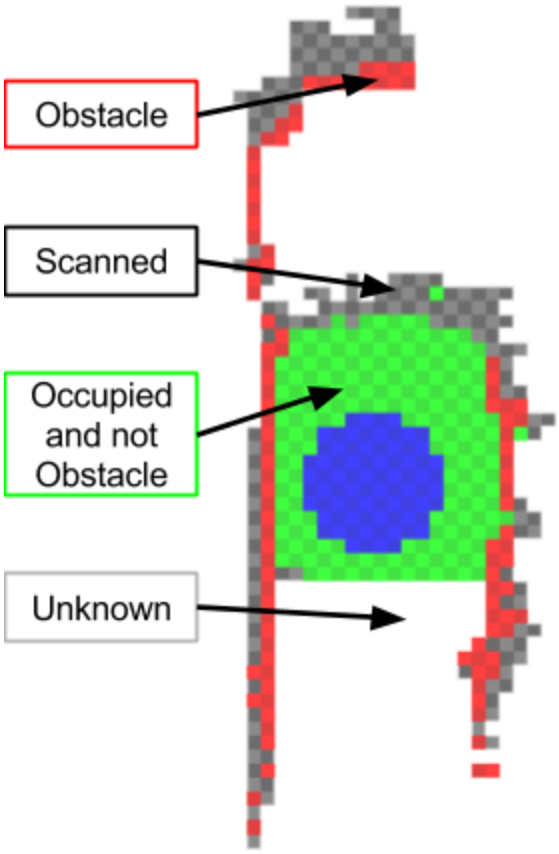


Figure 47: Voxel map showing voxel states

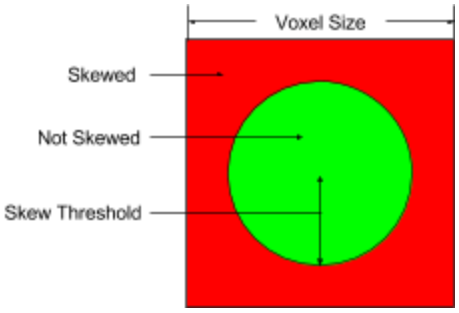


Figure 48: Diagram representing voxel skew threshold

5.4.2 Voxel Attributes

Figure 49 shows all the main voxel attributes. A key voxel clump is the safe but unreachable clump. This shows a flat area which the robot cannot explore as it cannot fit between the obstacles to the left and right.

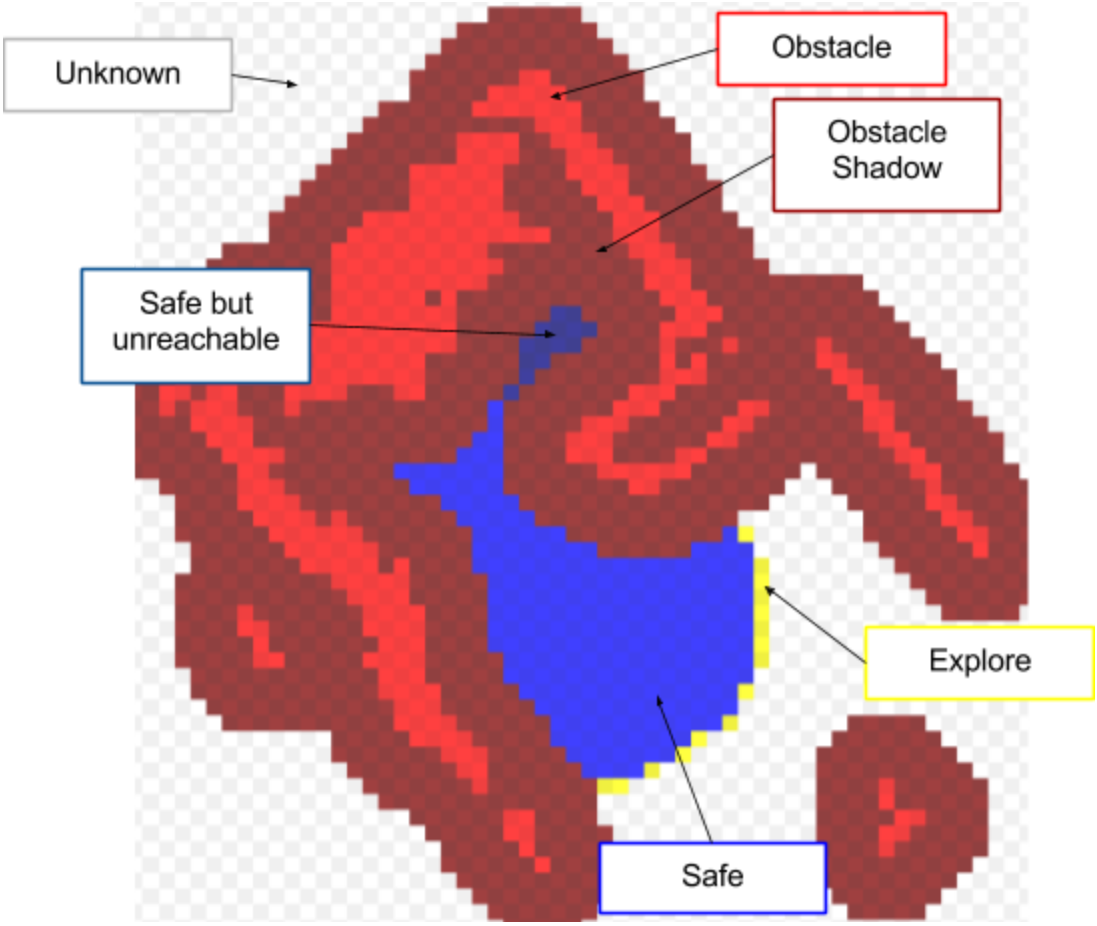


Figure 49: Voxel map showing voxel Attributes

Shadow
This states that the voxel is within a certain distance of an obstacle and therefore is not safe to traverse. This also covers obstacle voxels.

Safe
This states that the voxel is occupied, not skewed and not a shadow voxel and therefore safe to traverse. The only exception for this is the initial dead zone which forms under the robot which is automatically set to safe.

Explore
This states that the voxel has been scanned but hasn't got enough points to make a decision on the voxel's state. These voxels usually fall on the edges of scans where the laser hasn't got a high enough resolution or range to generate enough points. This infers that there is more data beyond these voxels and therefore helps form the target for the robot's next traversal. To ensure these targets are reachable, all exploration voxels need to be adjacent to a voxel which is either another exploration voxel or reachable.

Reachable
This states that the voxel is safe and reachable. It does this by taking the robot's position from the navigator and setting its corresponding voxel to reachable. Next it iterates through the entire voxelgrid and sets all safe neighbours of reachable voxels to reachable. It iterates through this process until no voxels have been updated. This is to ensure that exploration voxels adjacent to flat areas out of reach are not targeted.

5.4.3 Voxel Grid Target finder

One of the key functions of this class is to find the next exploration target. After the point cloud has been converted and the voxel grid has been analysed the get interest function is run. This finds the largest contiguous clump of exploration voxels. It does so by adding exploration voxels to a clump list recursively until all exploration voxels are in a clump list.
It then sorts the clump list by length and returns the largest clump. Once the clump has been found, the closest reachable voxel to the clumps centroid is set as the target for the next traversal. This is then passed to the route inspection algorithm along with the robot's voxel position which is converted to a list of voxels which forms the route the robot has to take to reach its target. If no exploration voxels exist and therefore no clumps are found the control loop stops and the exploration is complete.

```
def getIntrest(self):
	checkedVoxels = []
	clumps = []
	for voxel in self:
		self.neighbourhood = []
		if (voxel not in checkedVoxels) and voxel.attr["explore"]:
			reachableGroup = False
			for neighbour in self.getNeighbours(voxel):
				if neighbour.attr["reachable"]:
					reachableGroup = True
					break
			if reachableGroup:
				self.checkNeighbours(voxel)
				clumpSize = len(self.neighbourhood)
				clumps.append((self.neighbourhood,clumpSize))
				checkedVoxels += self.neighbourhood
	return self.clumpsToTarget(clumps)
```

Figure 50: Code snippet of get interest function from Voxel Grid class

Figure 51 shows how the voxel grid evolves as the robot moves from point to point. The last voxelgrid on the right has no reachable clumps and therefore has stopped exploration.

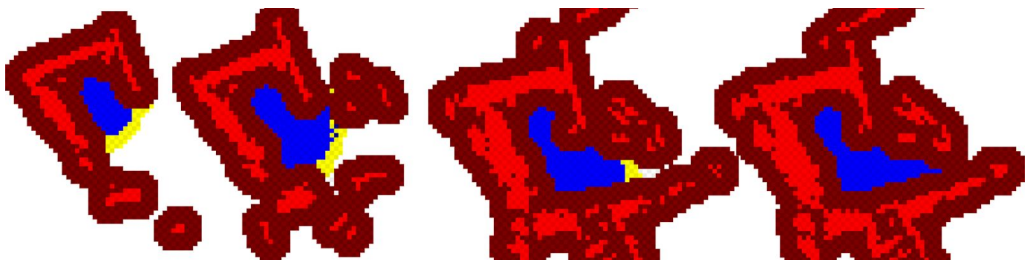


Figure 51: Voxel map sequence over time

5.5 - A* Route planner & Navigation Module

This navigator’s task is to keep track of the robot's voxel position and heading. It also coordinates the motor’s module to move the robot from voxel to voxel using the A* route inspection module.

The route planner takes the voxelgrid, starting position and target position and creates a safe path for the robot to traverse. This is done by implementing an A* route inspection algorithm. This provides safe and guaranteed optimal route creation. For details on the A* algorithm see Background Research.

```
def move(self,vox,target):
        print("Calculating move")

        routePlanner = aStar(vox)

        source = vox.voxels[self.u][self.v]

        path = routePlanner.getPath(source,target)

        self.followPath(path)

        self.motors.off()
```

Figure 52: Code snippet from Navigation class

5.6 - Motor Controller Modules

The motor modules control the motion of the robot using the estimated odometry of the hardware.
Below we are defining w as the wheel base width of the robot, m as the number of steps required for a full rotation and d as the wheel diameter.

5.6.1 - Straight forward motion

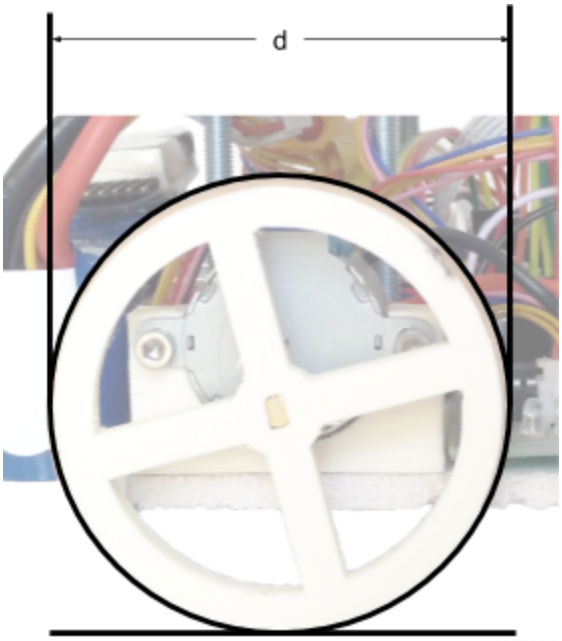


Figure 53: Diagram of wheel diameter

For a straight forward motion we first need to calculate the wheel circumference as πd. This will give us the distance travelled in one full rotation. Dividing our straight forward traversal distance t by the wheel circumference will give us the fractional number of rotations required. We define this as $t\frac{1}{\pi d}$.
Finally to get the number of steps we simply multiply the number of rotations required by the number of steps per rotation m. We can define this as $t\frac{m}{\pi d}$

Therefore the number of steps required for a straight forward traversal can be defined as:

Known

w = *wheel base width*

m = *number of steps per motor rotation*

d = *wheel diameter*

t = *straight forward distance*

Unknown

s = *steps required for straight forward motion*

Relationship

$$s = t\frac{m}{\pi d}$$

5.6.2 - Turning in place

For turning in place we calculate the distance each wheel has to counter-rotate to turn the robot $\theta°$ and plug it into the equation above.

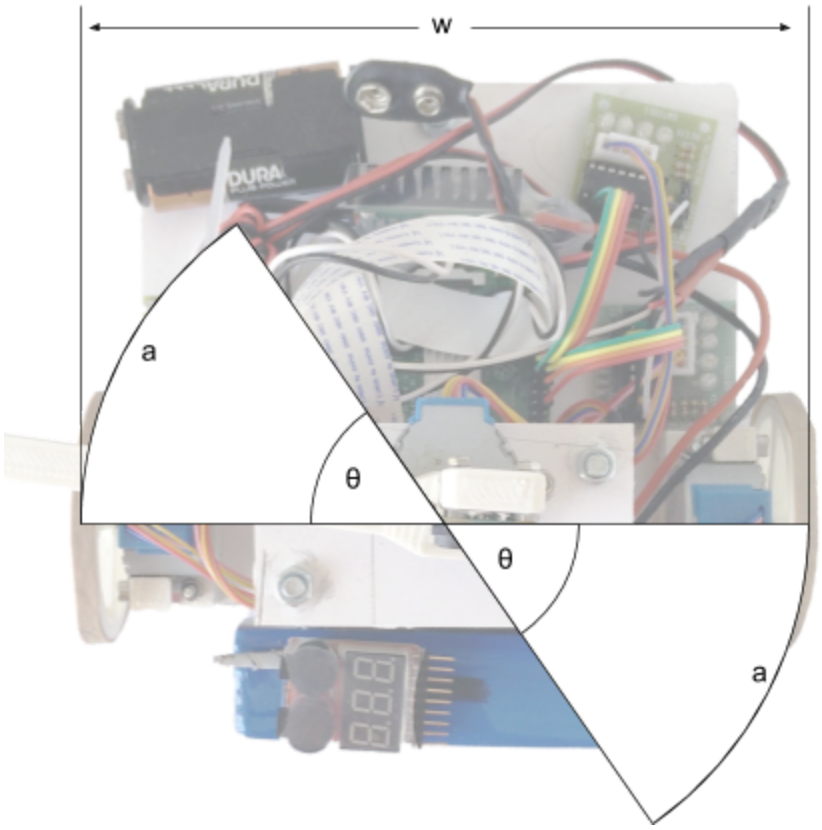


Figure 54: Diagram for turning in place trigonometry

To do so we need to measure arc length s which can be calculated as a fraction of a whole circle. The circumference of the whole circle is πw and the fraction of a full turn is $\frac{\theta}{360}$ Therefore the arc length $a = \pi w \frac{\theta}{360}$ and if we plug this into the equation for a straight traversal above it will get us the number of steps required to counter-rotate each motor. We can simplify this as: $\pi w \frac{\theta}{360} \frac{m}{\pi . d} \equiv \frac{\pi w \theta}{360} \frac{m}{\pi . d} \equiv \frac{\pi \theta w m}{360 \pi d} \equiv \frac{\theta w m}{360 d}$

Therefore we can define the number of counter-rotating steps as:

Known

$w = wheel\ base\ width$

$m = number\ of\ steps\ per\ motor\ rotation$

$d = wheel\ diameter$

$\theta = angle\ to\ turn\ in\ place$

Unknown

$s = steps\ required\ for\ turning\ in\ place$

Relationship

$$s = \frac{\theta . w . m}{360 . d}$$

5.7 - Networking

As we cannot guarantee that the robot's exploration area will provide a wifi connection we need to set up an ad-hoc connection. As the Raspberry Pi has a wireless chip and arial built in this is easily done and allows SSH access via a laptop within range of the robot.[27]

5.8 - Intrinsic Parameter

All parameters referred to in this chapter such as camera FOV, size, wheel diameter, GPIO ports, laser threshold and other variables are held in an intrinsic parameters file. This is to ensure that the system is adjustable while calibrating. It also allows the system to adapt to different hardware and different environments easily and efficiently.

```
from math import radians

#########
#Camera Parameters
#########

pu = 972  #camera horizontal FOV in pixels
pv = 1296 #camera vertical FOV in pixels


qv = radians(62.2) #camera vertical FOV in radians
qu = radians(47)   #camera horizontal FOV in radians

#########
#Scanner Hardware Parameters
#########

a = 17.5              #Boom Length in cm

L = radians(90-25)    #Laser Angle in radians

#########
#Odometry Hardware Parameters
#########
```

Figure 55: Code snippet from Intrinsic Parameters file.

6 - Test Results

6.1 - Module Tests

The following tests aim to verify the success of each module and therefore each part of the robot independently.

6.1.1 - Motor Odometry

Accuracy
The robot's odometry records the robot’s position in real world space. To calculate this, the robot needs to move around in real world coordinates. To calculate this we take aspects such as the wheel radius and wheelbase and calculate the number of steps or rotations the motors need to perform to travel that distance.
However, this cannot be 100% accurate as it relies on real world measurements which can slip and change.

Figure 56 shows a graph showing the overshoot of the robot when instructed to move along a straight line. As we can see there is a linear correlation which shows that the overshoot is related to the distance travelled. This indicates that the error in the wheel diameter. This is could be due to degradation of the rubber wheels and could easily be corrected by adding 0.322mm onto the wheel diameter variable. However even at the furthest traversal the odometry error is still below 0.482%. This is well within the recommended 2% linear drift set out by Søren Riisgaard and Morten Rufus Blas.[6]

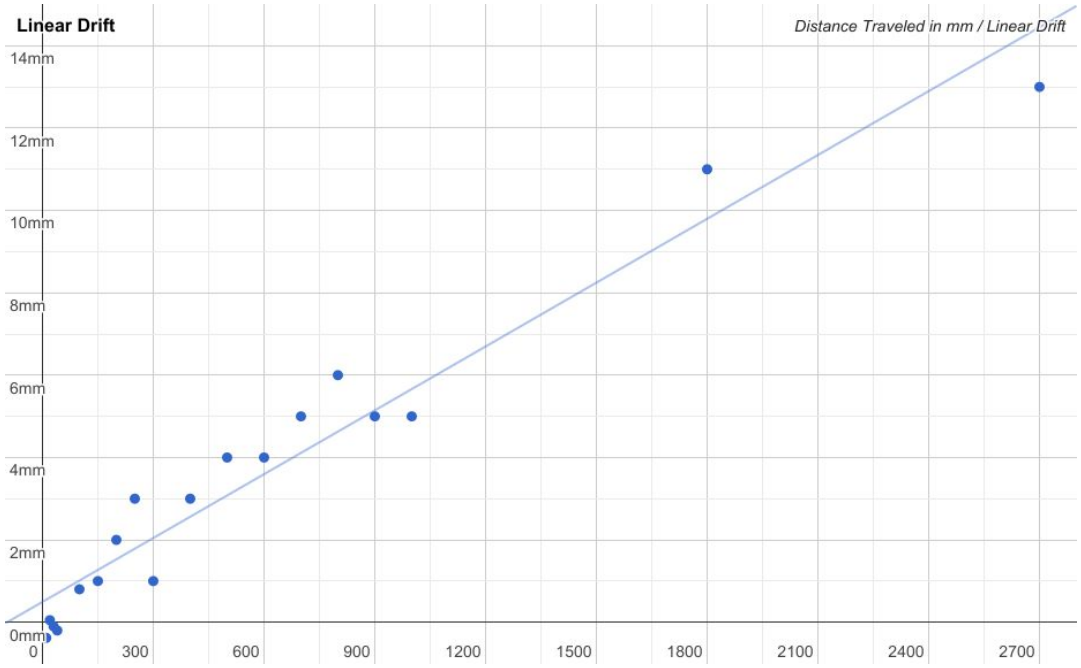


Figure 56: Graph of linear odometry drift

Figure 57 is a similar graph showing the drift as the robot turns on the spot. Again there seems to be some linear correlation showing an error in either the wheel diameter or wheelbase.However, at the maximum number of rotations the error is still below 0.723% of the total rotations. This is well within the recommended 4.4% rotational drift set out by Søren Riisgaard and Morten Rufus Blas.[6]

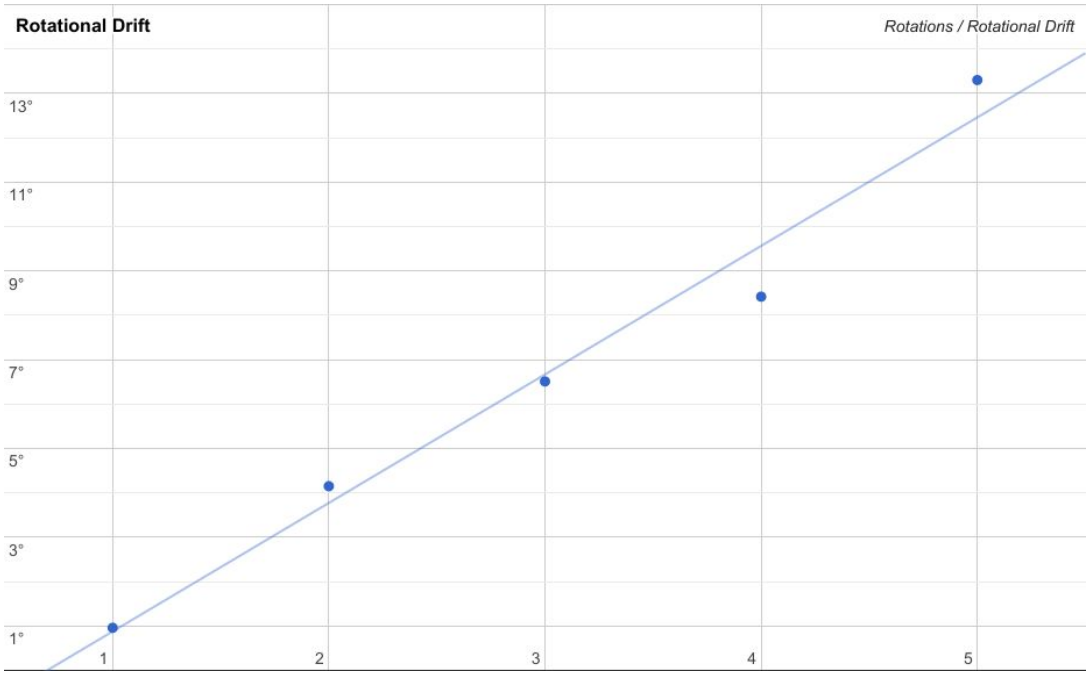


Figure 57: Graph of rotational odometry drift

Speed

The robots straight line movement speed is 4.75cm per second. This is limited by the motors used. This could be increased slightly however any faster than this and the stepper motor can skip causing a higher odometry drift. Another way to increase this speed would be using higher geared motors or larger wheels. However both of these would reduce the overall accuracy.

Adaptability

This module is highly adaptable as a change in chassis or wheel size can be quickly updated through the parameters file which stores all the robot's attributes

Obstacle traversal

Due to the rear skid the highest obstacle it can ride over is <2 mm as it is assumed this robot will be used on flat, clean surfaces. This could be increased if a bogey wheel is attached instead of the skid or rougher pneumatic tyres are used. However both of these solutions would reduce accuracy.

6.1.2 - Scanner

Accuracy

The accuracy of the scanner highly depends on the level of calibration, the environment and the robot's hardware. This could be improved with automatic calibration and leveling functions, however this would have taken longer to develop.

Figure 58 shows a comparison between the real world and the point cloud. The point cloud registers this distance as 82.95 cm which is well within 1 cm accuracy of the real value and therefore accurate enough for the robot to traverse safely.

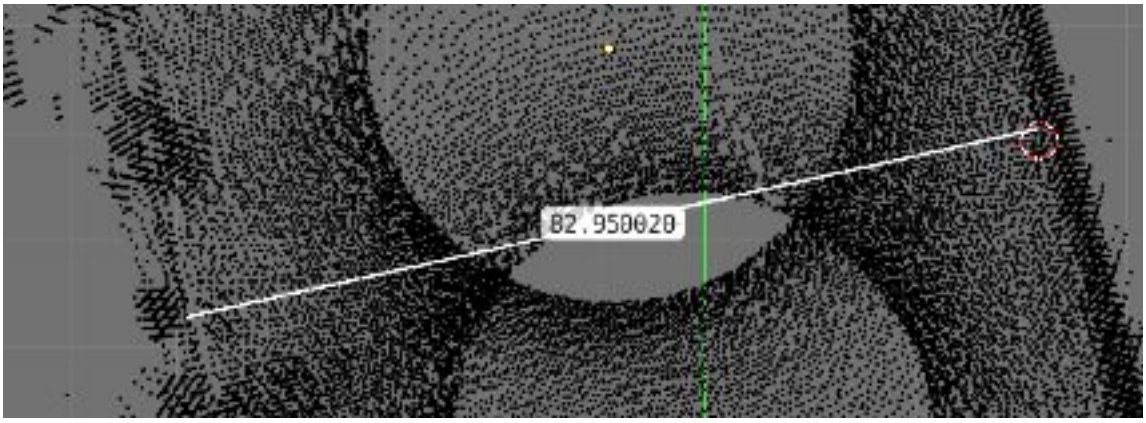


Figure 58a: Comparison showing scanner accuracy

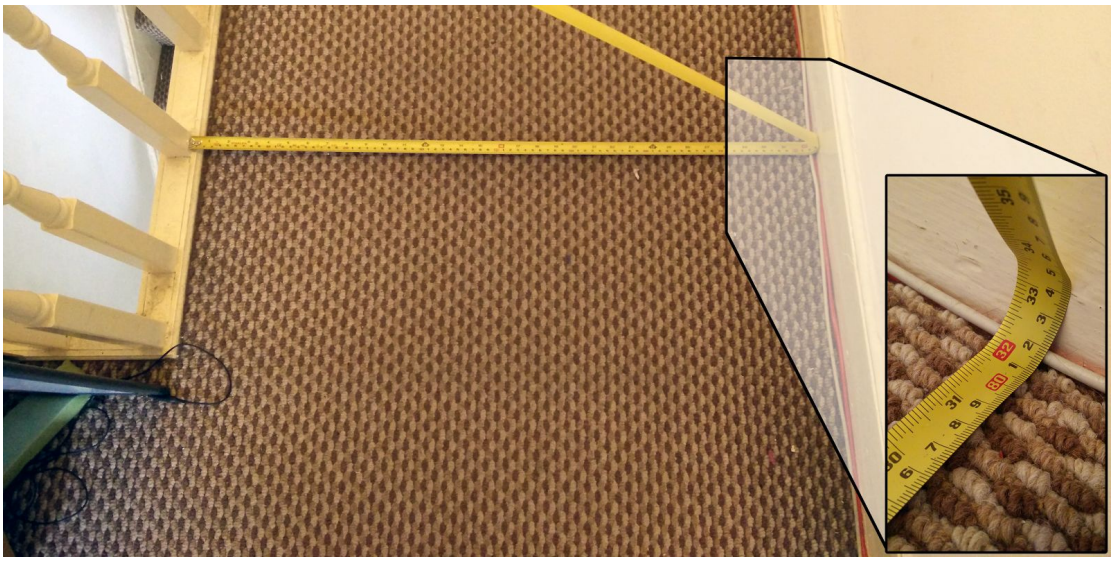


Figure 58b: Comparison showing scanner accuracy

Figure 59 shows a top down scan of the same environment compared to the absolute truth shown here as the black outlines. This shows not only the precision but also the accuracy falloff at long distances. The wall on the left is around 3 meters away from the robot's center and as you can see the point's distances have become more quantised. This scan also shows the board it was placed on for calibration.

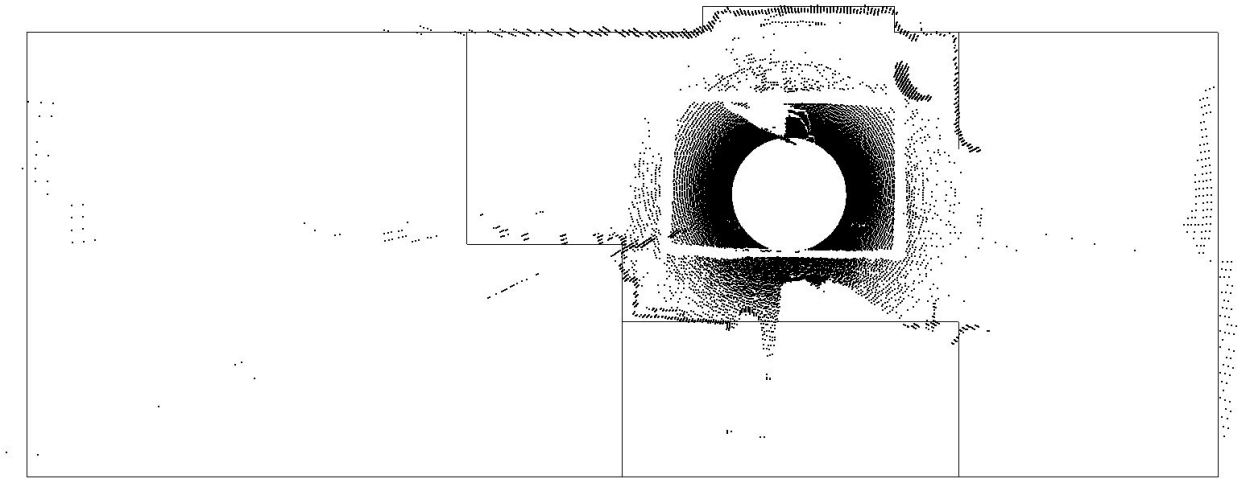


Figure 59a: Scan accuracy comparison with absolute truth (top view)

Figure 59b also shows a small amount of erroneous points to the right hand side. We can see some points registering below the floor level. This is due to the reflective vinyl surface however it does not affect the voxel state or registration algorithms. This will be explored later in the black floor environment test.

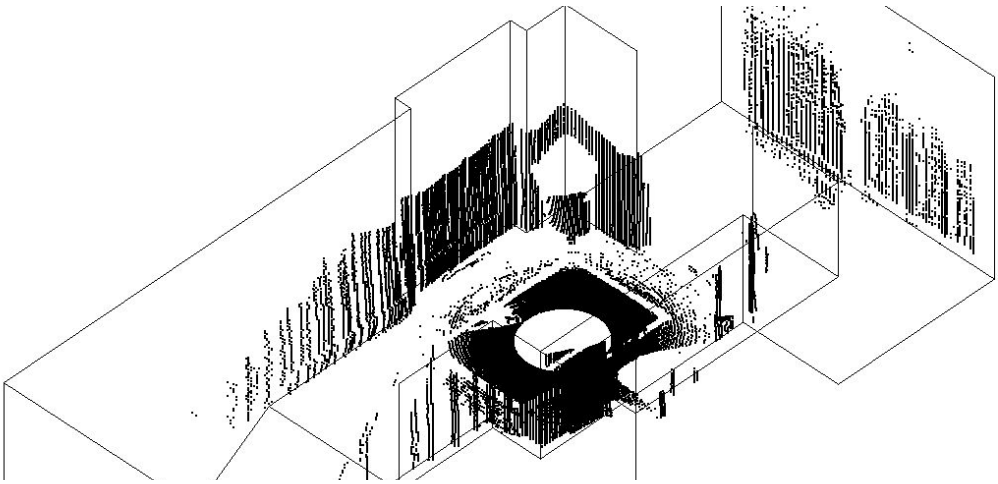


Figure 59b: Scan accuracy comparison with absolute truth (iso view)

The main issue whilst testing was random capture failures. An example of this can be seen below in Figure 60 where a clean environment produced an exceedingly noisy scan. This was unexplored because of its sporadic nature. This is speculated to be related to a hardware failure. The two main components which could cause this are the camera and the laser module. During testing the laser module had to be replaced four times as the brightness seemed to fall off over time. This was originally attributed to a power issue but after replacing the main power supply and isolating the laser module this was left unexplored.

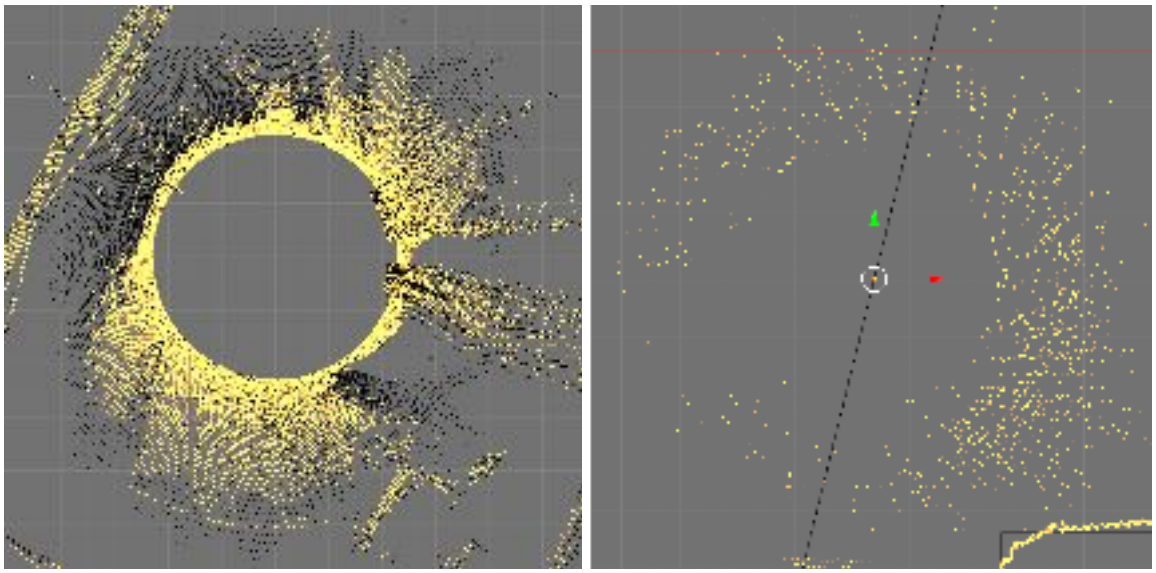


Figure 60: Point clouds of failed scans

Scans can also be visualised as a panoramic RGB depth map. However these images should only be used for visualisation as this data is quantised to 255 levels.

Colour	Calculated by	Represents
Green	The maximum brightness of the row	Certainty of the lasers position
Red	The index of the brightest pixel in the row	Reflected laser angle ≈depth
Blue	The cumulative pixel values of that row	Sharpness of laser response
Black	Any row where the maximum brightness falls below threshold	Unknown area

Figure 61a shows a floor level scan with some errors introduced by an external light source at the center of the scan. These form streaks as they form the brightest point for multiple slices whilst inside the camera's field of view.

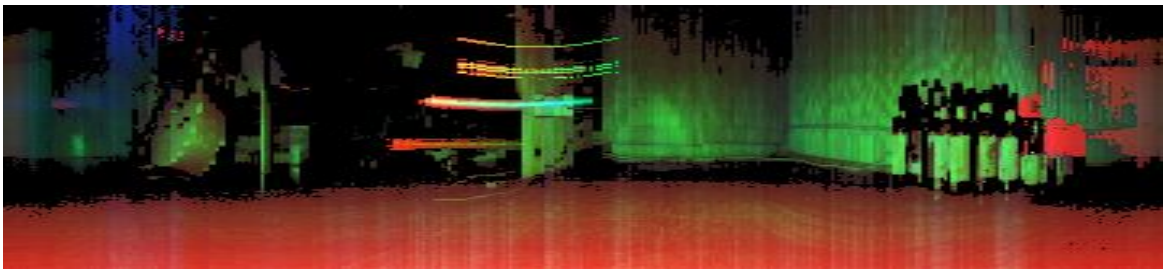


Figure 61a: RGB Depth map captured by scanner

Another notable point is the vertical streaks of blue seen at the bottom of the scan magnified in Figure 61b. This is where the scanners turntable jumped slightly whilst the capture was in progress therefore introducing a small amount of motion blur. This mechanical issue could be rectified with a torsion clock spring mounted to the arm to dampen motion.

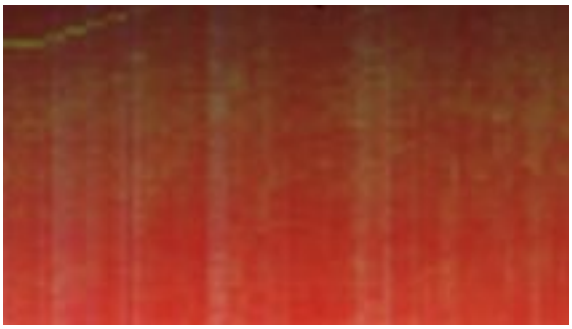


Figure 61b: Close up of Figure 61a

Figure 62 shows a desk level scan with similar artefacts as above.

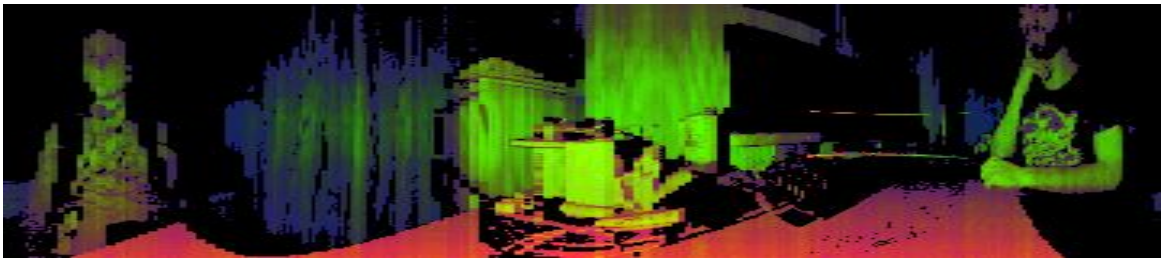


Figure 62: RGB Depth map captured by scanner

The blue streaks were caused by the turntables stepper motors. As they were not designed for degree level precision a small amount of play was introduced. At its worst this formed small amounts of motion blur and clumping of points in the scan which can be seen in Figure 63.

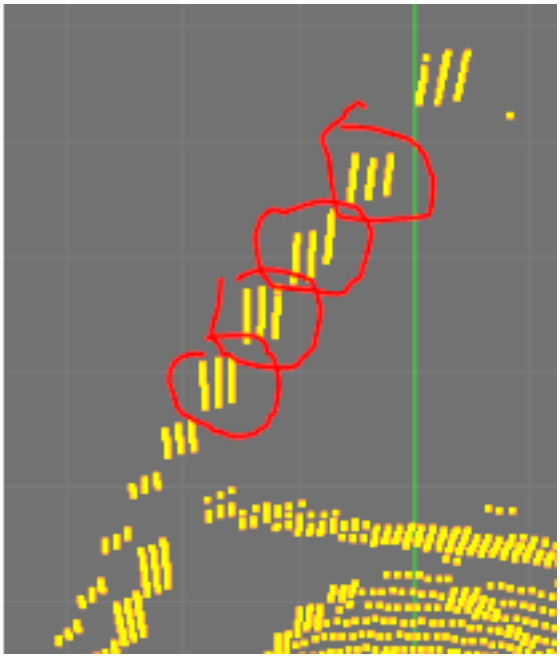


Figure 63: Point cloud clumping

Figure 64 shows a simplified representation of the inherent play marked in red in the motor's gear train. The commutative play equates to around 3 degrees of error in the boom's motion. The clumping comes from the boom's inertia generated by the motor. The first motion of the motor's gear (blue) pulls all the gears together giving the boom (yellow) a small amount of inertia.

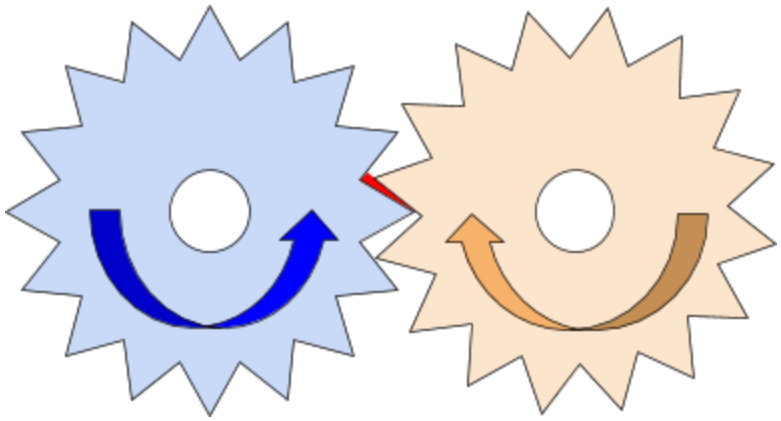


Figure 64a: Diagram Showing initial gear locking

As the image capture starts and the motor stops, the booms inertia continues to turn the cogs until they are all locked in the reverse direction.

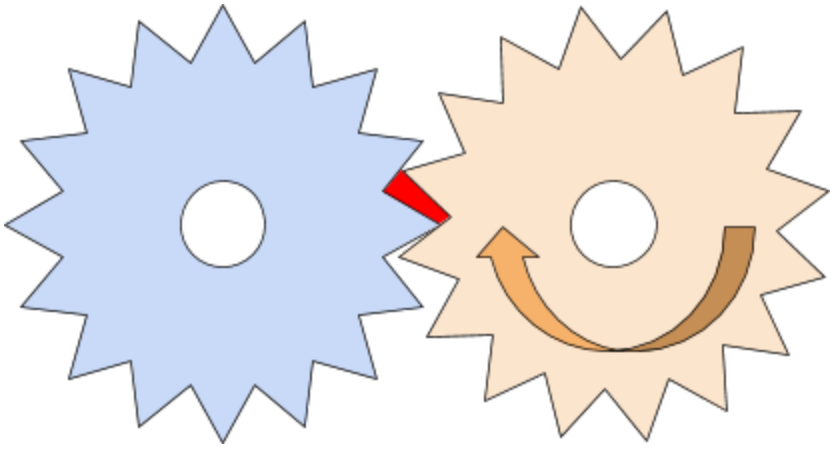


Figure 64b: Diagram Showing inertia

As the gears are no longer locked in the correct direction, the next 3 degrees are spent re-locking the cogs and therefore not moving the boom. As soon as they lock together again the boom will be pulled round repeating the process.
This issue could be rectified mechanically with higher quality motors or a torsion clock spring mounted to the arm to dampen motion. This could also be fixed in software by modeling this error so it can be removed or threading the system to allow for a more fluid motion. However twisting the CSI camera ribbon cable provided some resistance minimising this artefact.

Speed

The scanner speed is the bottleneck of the system. Therefore it has been made as efficient as possible. A full scan on average takes 43 seconds however this is highly dependent on the number of successful points registered. As an example, a timed scan has been broken down into the pie chart in Figure 65.

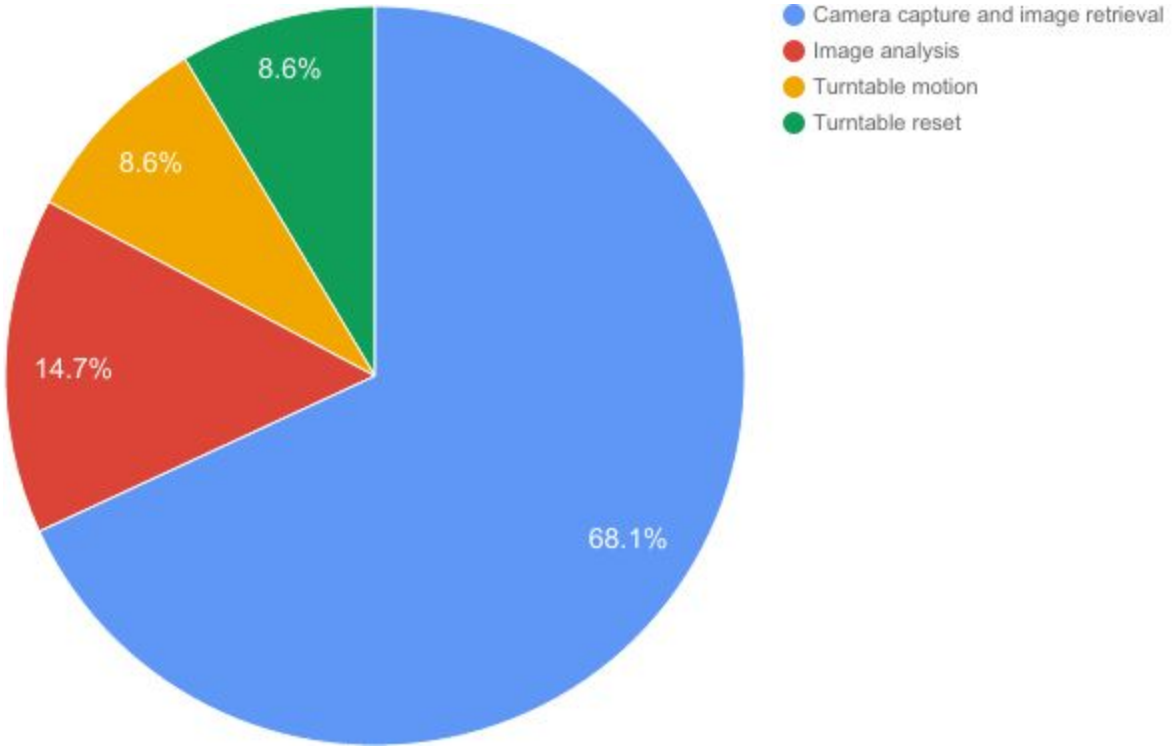


Figure 65: Chart of time spent scanning

As we can see the image capture takes the most time and could be improved further. One option to speed up capture and exploration time is to introduce threading. This could be used here to move the motor whilst analysing the previous image. This could reduce the scan time by a maximum of 31.9%. However for this project threading would have drastically increased development time.

Adaptability

The module is specially designed for the RPi camera so would be hard to adapt for a different camera. If a new camera was used the scan and capture slice functions would have to be changed to retrieve the image correctly. However the actual image processing function is highly adaptable as it take a generic 2D numpy array along with the camera parameters which are stored in the intrinsic parameters file.

Point density

The point density can be adjusted to ensure a critical balance between data gained and run time. This is because the time taken during voxelization, registration and image analysis directly correlates to the point density.

To reduce the point density, the images returned by the camera can be squashed vertically and the number of turntable motor steps per slice can be increased. During testing a squash factor of 8 and 12 turntable steps (≈1 degree) provided more than enough data to make an accurate model of the environment. These parameters provided up to 58,320 points however the average scan had around 40,000 points after filtering and thresholding.

The highest possible number of points would be the number of steps per 360 degree rotation of the motor multiplied for each pixel vertically. This would be 4104*1296 = 5,318,784 points before filtering

6.1.3 - Voxel Grid and Analyser

Accuracy

The main accuracy concern in this module is the target finder function. This finds the largest clump of exploration voxels in the voxel grid. Figure 66 shows two blue cross-hairs correctly identifying the closest voxel the robot can reach to scan the reachable unexplored voxel clumps coloured in pink.

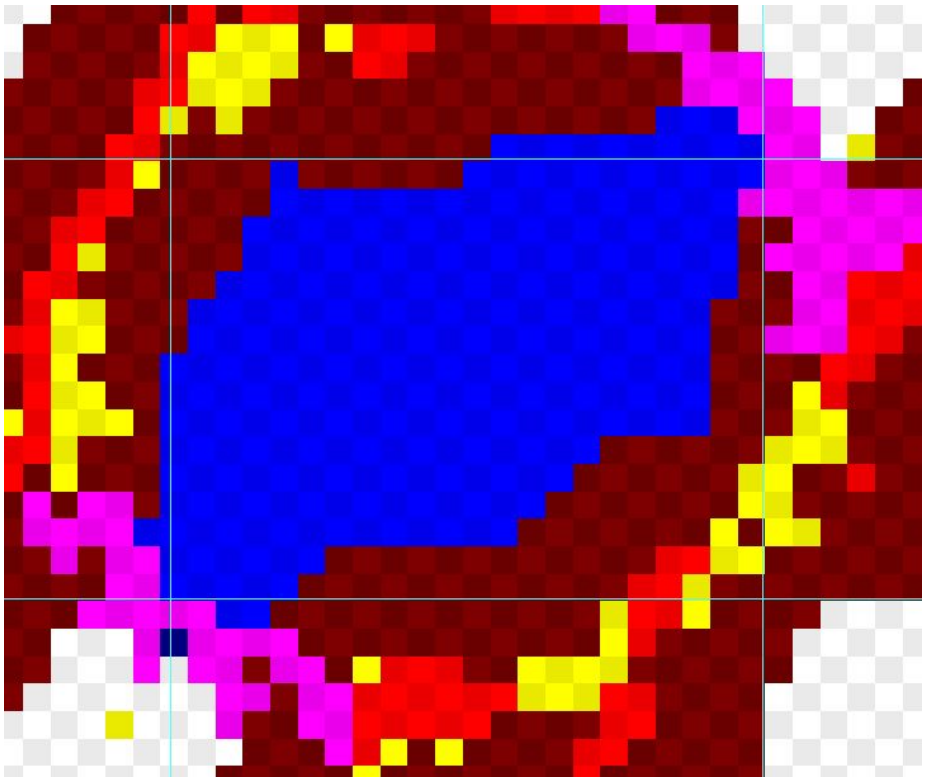


Figure 66: Voxel grid showing exploration clump centers

The second accuracy concern is setting each voxel to the correct state. Figure 67 shows the scan data being overlaid on the voxelgrid. This shows the voxelgrid correctly setting each voxels state in a clean test environment.

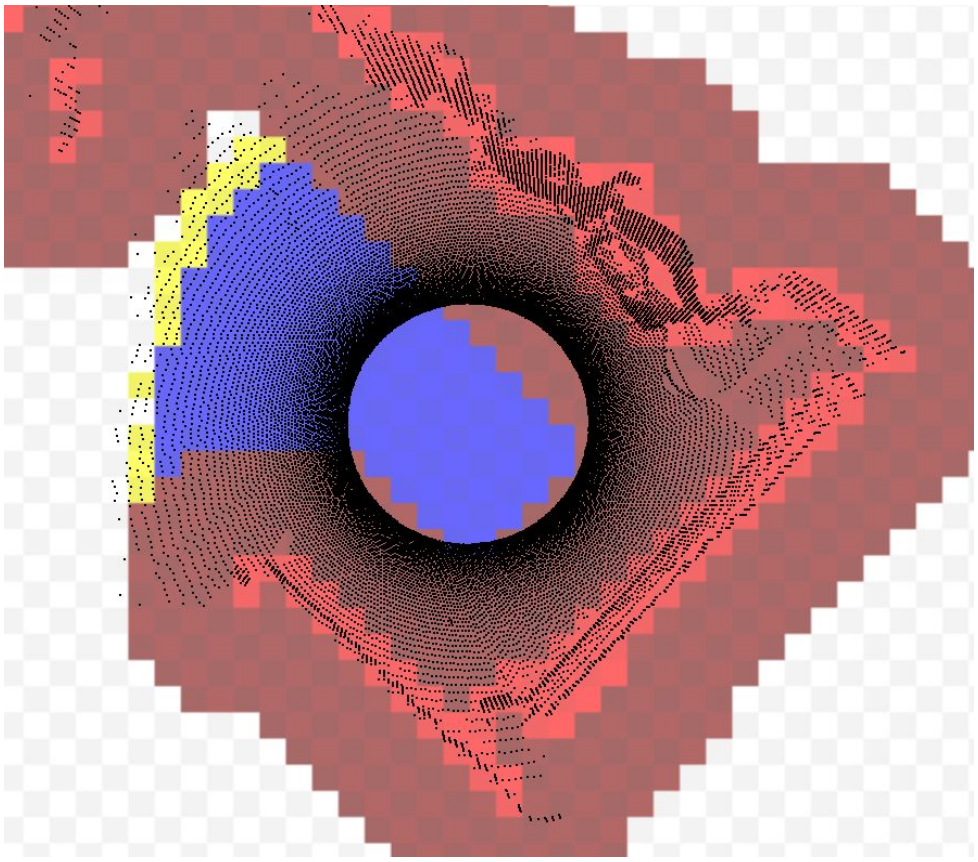


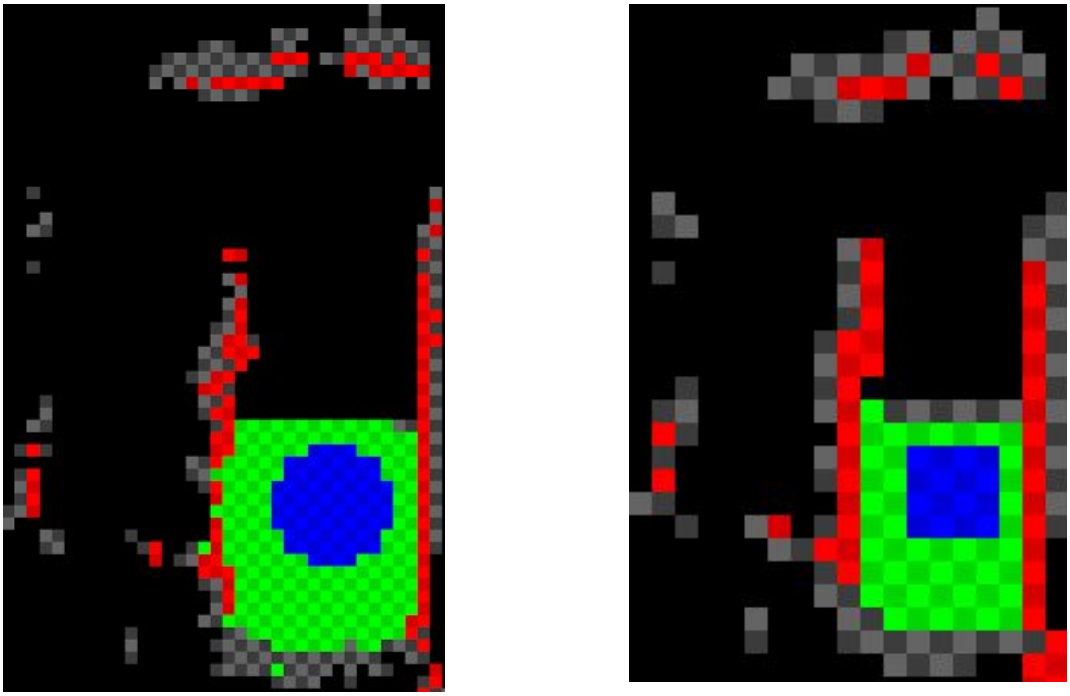
Figure 67: Voxelgrid showing exploration clump centers

Speed

The actual speed of the voxelgrid analyser is highly dependent on the scan area, voxel and point cloud size. Breaking a 40,000 point scan over 10^2 meters broken down into 5 cm voxels takes on average 1.6 seconds and analysis is complete in a further .5 seconds. Another function of the voxel grid is finding the target position for the next scan. This highly un-optimised recursive function is one of the slowest and needs further improvement, however it still only takes on average 2.9 seconds. Even after 7 compiled scans the exploration loop still falls within the 2 minute window. This could be decreased by using Numpy arrays, PCL voxelization or kd trees in place of Python's built in data structure for faster access.

Adaptability

Each voxel classification threshold is independent from other thresholds and voxel size. The makes the module highly adaptable to new environments with different thresholds and voxel sizes as shown in Figure 68.



Voxel size: 2.5 cm

Voxel size: 10 cm

Figure 68: Voxel grids with change in voxel size

6.1.4 - Route Inspection & Navigation

Accuracy and Speed

Figure 69 shows the A* algorithm generating a route through a real world voxel grid which is represented as an array of voxels which have been coloured pink.

Figure 69a: Voxel map of route planner path

Figure 69b: Voxel map of route planner path with obstacle

The voxel grid in Figure 69b has had a section removed. As unknown area is neither safe nor reachable, it is therefore avoided by the route planner. As seen here the route is sub-optimal and therefore hasn't been implemented correctly. This is probably due to an error in the voxel update code. However as it still provides a safe path from source to target it is therefore an admissible error.

To test this, we can run the following code in Figure 70 which generates the route in Figure 71. As we can see it is sub-optimal, however still manages to create a safe route through the environment.

```
import sys
import random

sys.path.append("modules")
from voxGrid import voxGrid
from aStar import aStar

grid = voxGrid(None,None)

for voxel in grid:
	voxel.attr["safe"] = True

for voxel in grid:
	if (30 < (voxel.u + voxel.v) < 40)
and (voxel.u > 10):
		voxel.attr["safe"] = False

routePlanner = aStar(grid)

source = grid.voxels[1][1]
target =
grid.voxels[grid.voxCount-2][grid.voxCount
-2]

path = routePlanner.getPath(source,target)

for voxel in path:

grid.voxels[voxel.u][voxel.v].attr["temp"]
= True

grid.save("test.png")
```

Figure 70: Code snippet of route planner test

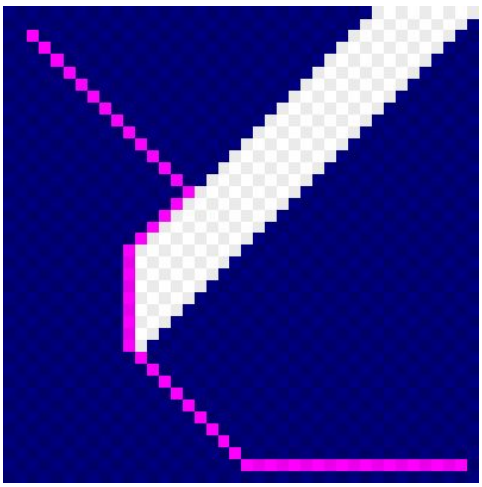


Figure 71: Route planner test output

Figure 72 shows a real world example of the robot traversing an environment autonomously and safely using the Navigation module. This is made up of 6 scans and 5 traversals.

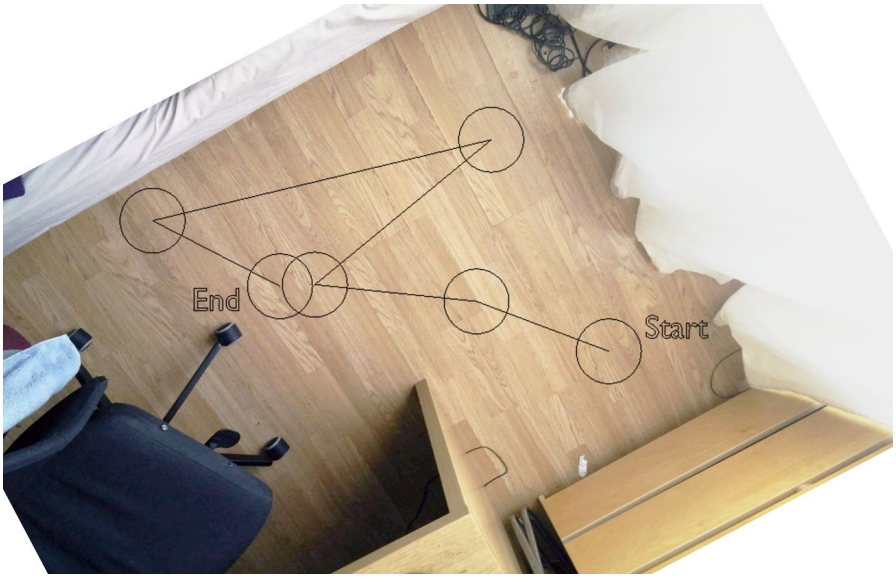


Figure 72: Real world navigation with path overlay

Figure 73 shows the point cloud formed by the above scans. Please note that the floor here has been removed to aid in clarity.

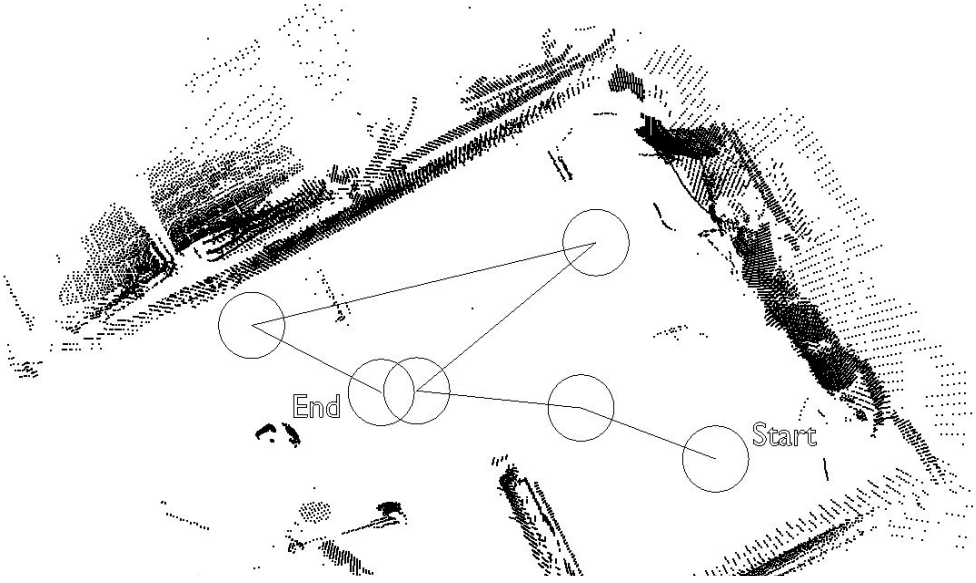


Figure 73: Pointcloud navigation with path overlay

Adaptability

This algorithm works with any size 2D voxelgrid with any states as long as the target and source voxels fall within this grid. The safe voxel traversal type is easily adjustable and could be added to the intrinsic parameters file.

6.2 - Real world testing

The following criteria are pulled from the project brief in the initial plan.

“The SLAM world model needs to be accurate enough to navigate around messy environments. For example a desk covered in books and papers.”
“The system should adapt to new environments automatically.”

As these attributes are highly dependent on the environment, they will be tested through a selection of real world explorations below. These examples should show that the system is accurate and dynamic enough to meet these criteria.

6.2.1 Bridge

This test included different surfaces, terrain and obstacles for the robot to overcome to build up a scan of the environment. This included low lying obstacles such as the roll of tape and overhangs that the robot would need to traverse under.
To successfully complete this task the voxel obstacle threshold needed to be increased to allow it to ride over the small bump to get onto the bridge. This was done quickly and efficiently through the intrinsic parameters file and shows how the system can be adapted on the fly.
One error this scan did highlight is that the voxel grid does not apply a shadow to empty space. This started to assign the edges of the bridge as exploration voxels even though they are on the edge of a cliff face. However the A* route inspection chose the most direct path straight over the bridge. This does need to be addressed in future work. The surfaces here are covered in masking tape to ensure that the laser gets identified correctly. Without this the dark shiny counter would be hard to accurately scan. This is tested in the black floor test.

Figure 74 shows the layout of the real world test.

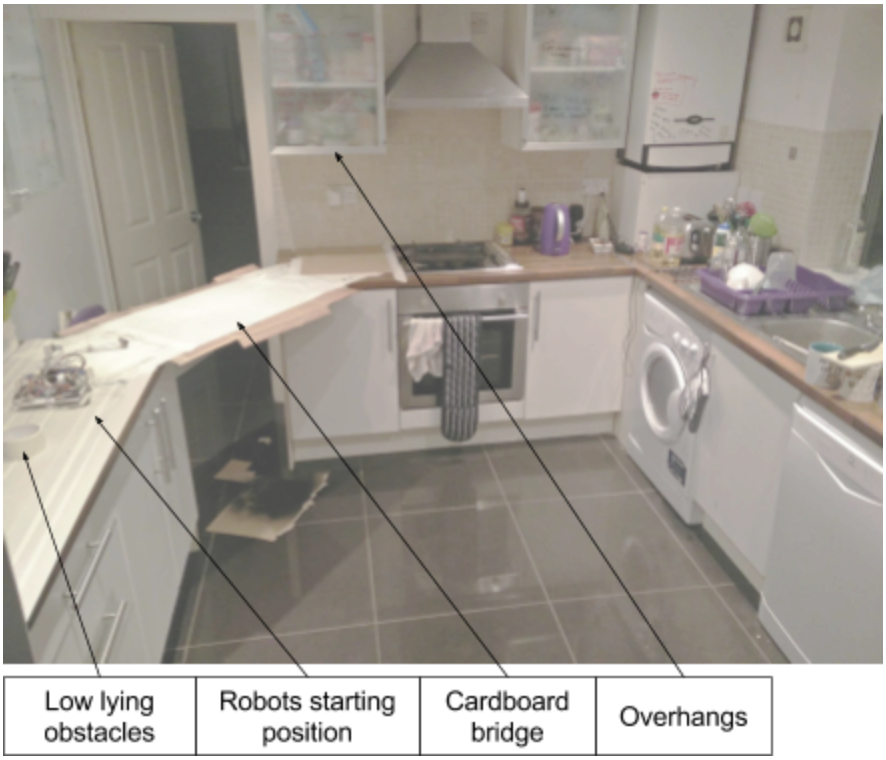


Figure 74: Photo of bridge test environment

Figure 75 shows the point cloud the robot returned after 4 consecutive scans. Regardless of the masking tape giving a good floor response, this had a below average number of points at 111,198 points over 4 scans. This is probably due to the sparse close areas the robot could see.

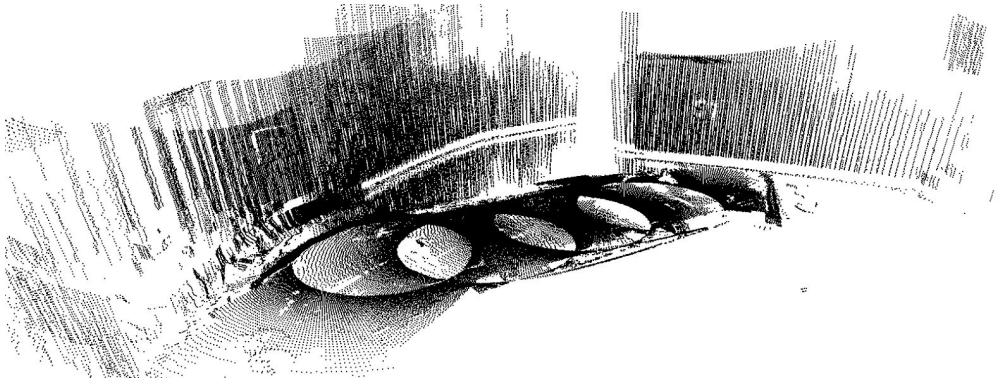


Figure 75: Point cloud returned by bridge test

Figure 76 shows the range of the scanner being able to pick up distant objects on the other side of the room such as door frames and walls. The furthest response being around 2.5 meters which is comfortably below the theoretical 3m range of the scanner.

Figure 76: Above view of point cloud returned by bridge test

Figure 77 shows the majority of the voxels have the correct state. However there are a few exploration voxels on the edge of the bridge which should have been set to unsafe as the robot cannot determine whether it can safely occupy the surrounding voxels.

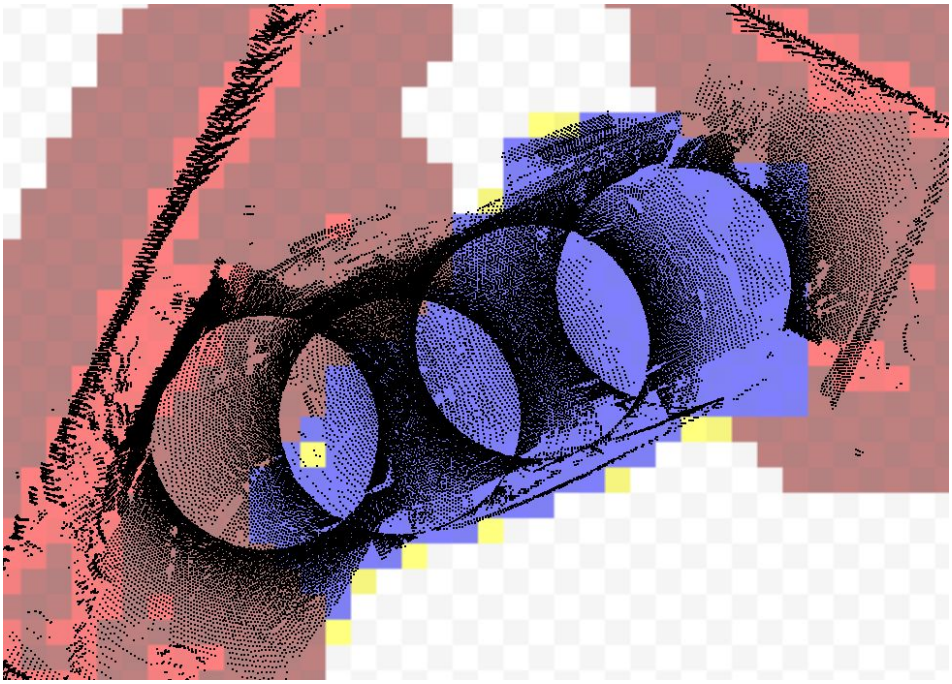


Figure 77: Voxel grid overlay of bridge test

Another interesting point about this scan is the level of detail the scanner can pick up. For instance the power plug on the wall which is clearly visible in Figure 78 due to the excellent registration between scans. Figure 78 is purposely blurred to make identification easier.

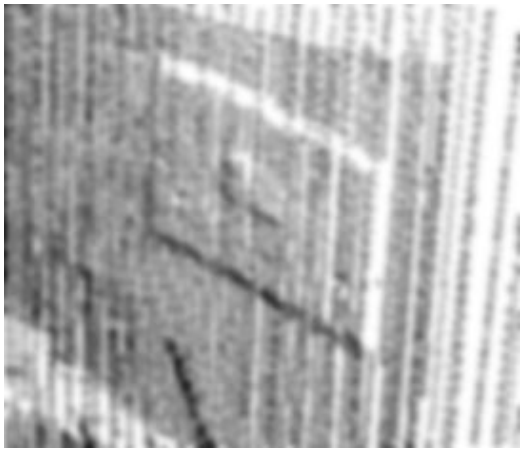


Figure 78: Point cloud details

6.2.2 Black Floor

This test was to demonstrate some of the environmental limitations of the system. In this test the robot was placed onto a dark shiny surface. It completed one scan and returned instantly as no floor was detected and therefore no explorable area was set.

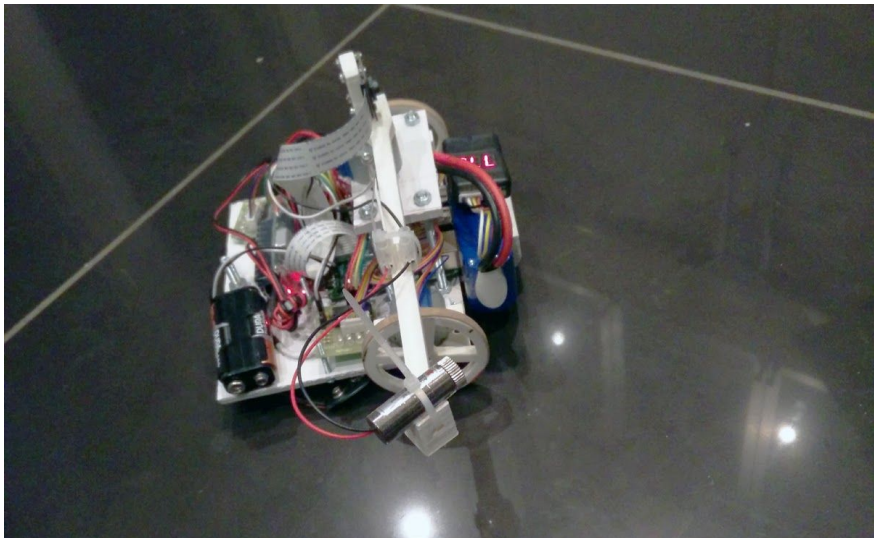


Figure 79: Photo of robot in black floor test environment

Figure 80 shows the sparse scan returned of the black floor test environment.

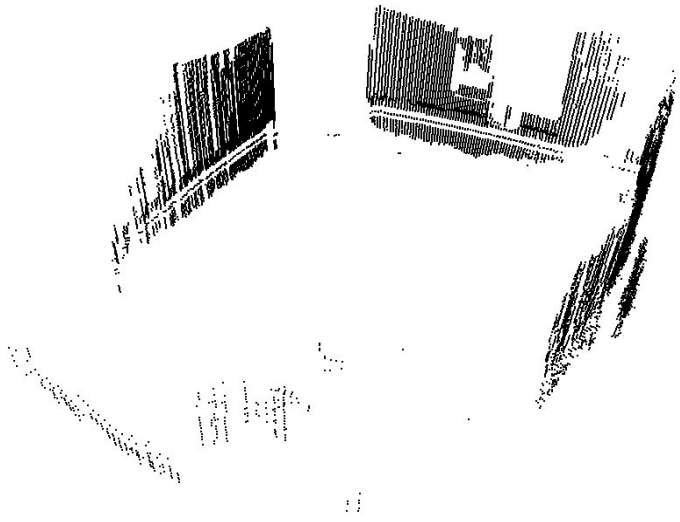


Figure 80: Point cloud of black floor test environment

An interesting point here is the points which get returned which are below the floor line. This is due to the laser bouncing off the floor and hitting the obstacle essentially acting as an optical and digital mirror. This effect has been visualised here in Figure 81. This occurs when the surface is more than 50% specular.

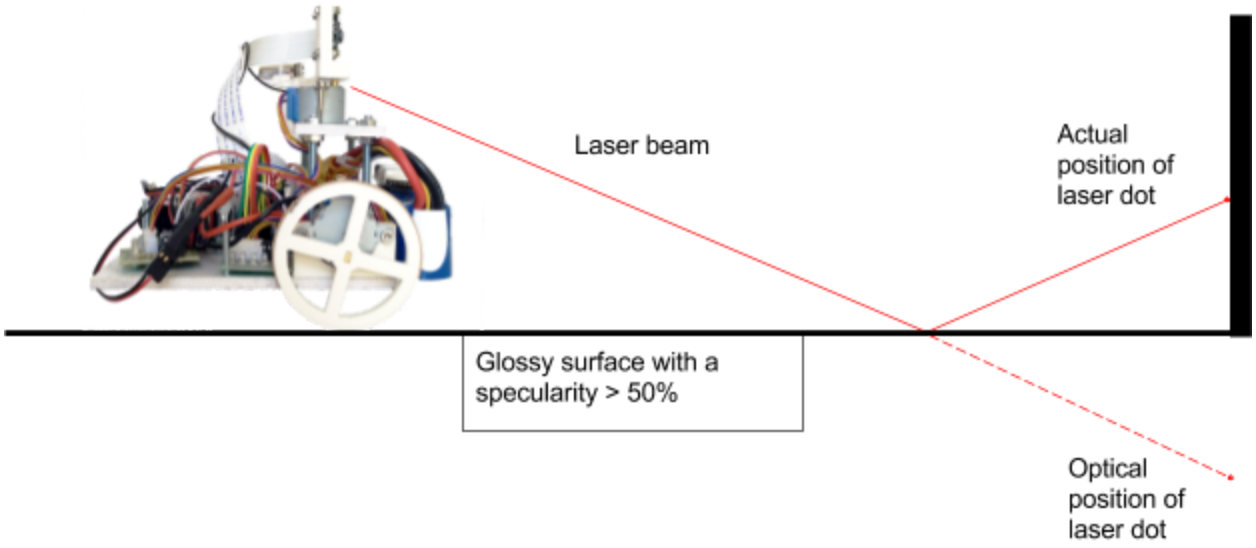


Figure 81: Diagram showing result of reflective flooring

6.2.3 Corridor

This test was to push the number of scans to the limit by giving the robot an open space. The robot completed 7 scans before stopping due to light pollution from one of the windows. Each scan was highly detailed and registered correctly.

Figure 82: Photo showing robot in corridor test environment

Figure 83 shows the scans returned by this test with and without the floor response for clarity. We can see the tight registration of objects such as the sloped roof, washbasket and box edges.

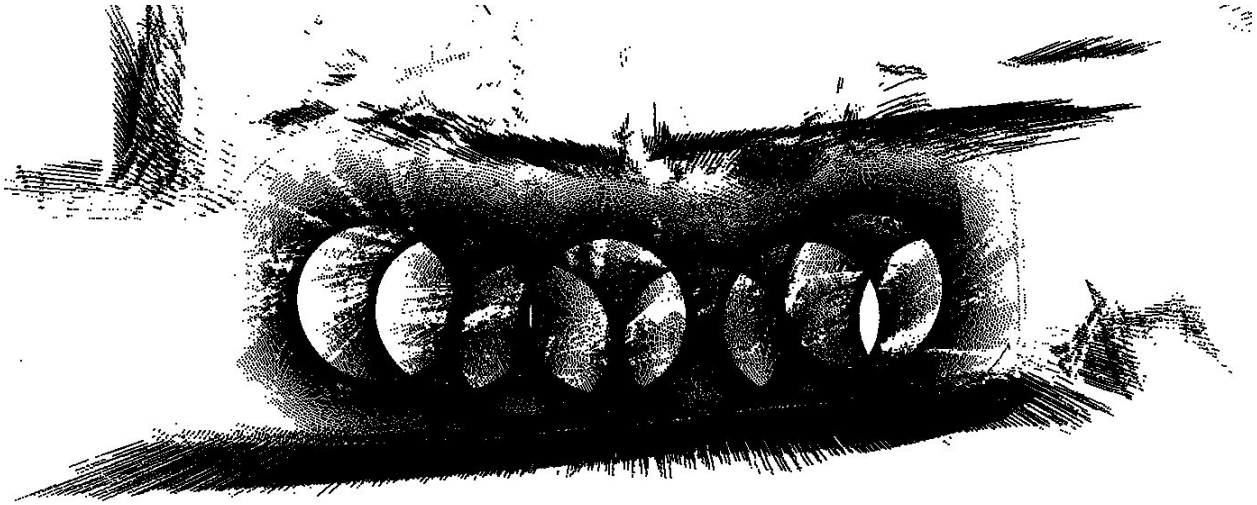


Figure 83a: Point cloud of corridor test environment

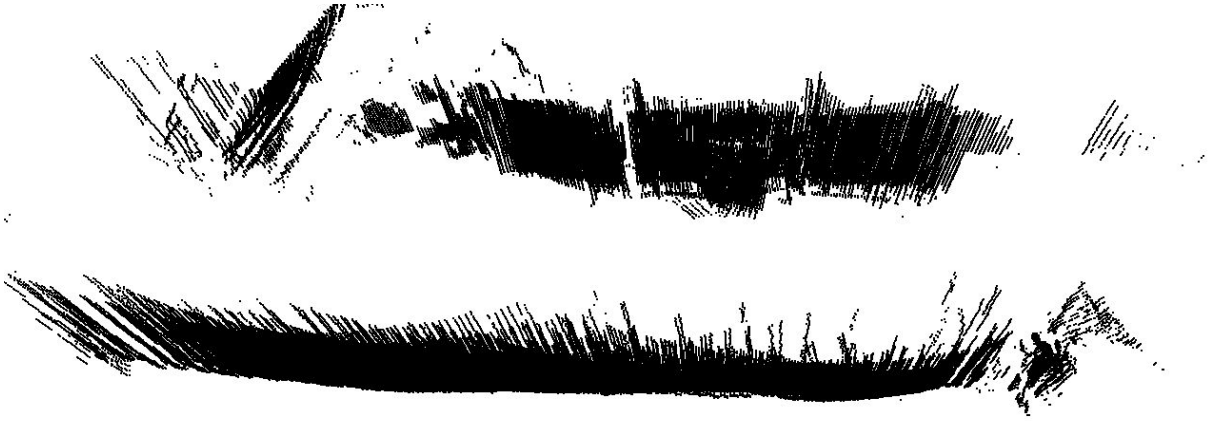


Figure 83b: Point cloud of corridor test environment with floor removed

6.2.4 Small obstacles

This test was to ensure that the robot would exit correctly when all of the reachable areas have been scanned. The obstacles have been placed apart as the robot won't try to fit through gaps smaller than its own radius. This is depicted by the obstacle shadow voxel state.

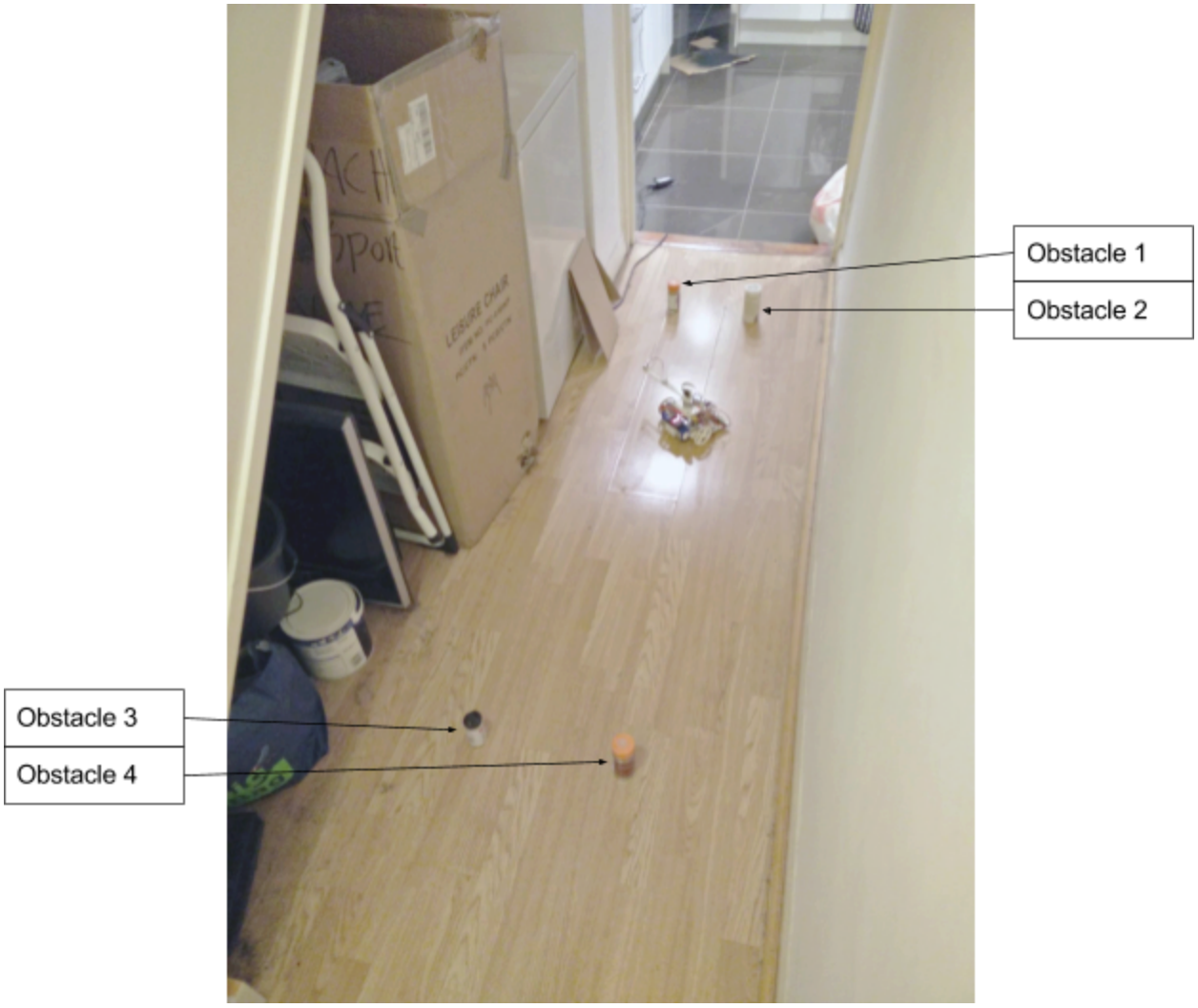


Figure 84: Photo of small obstacles test environment

Figure 85 shows the overlaid voxel grid on the point cloud data returned. The robot completed scanning the test area after 5 scans.

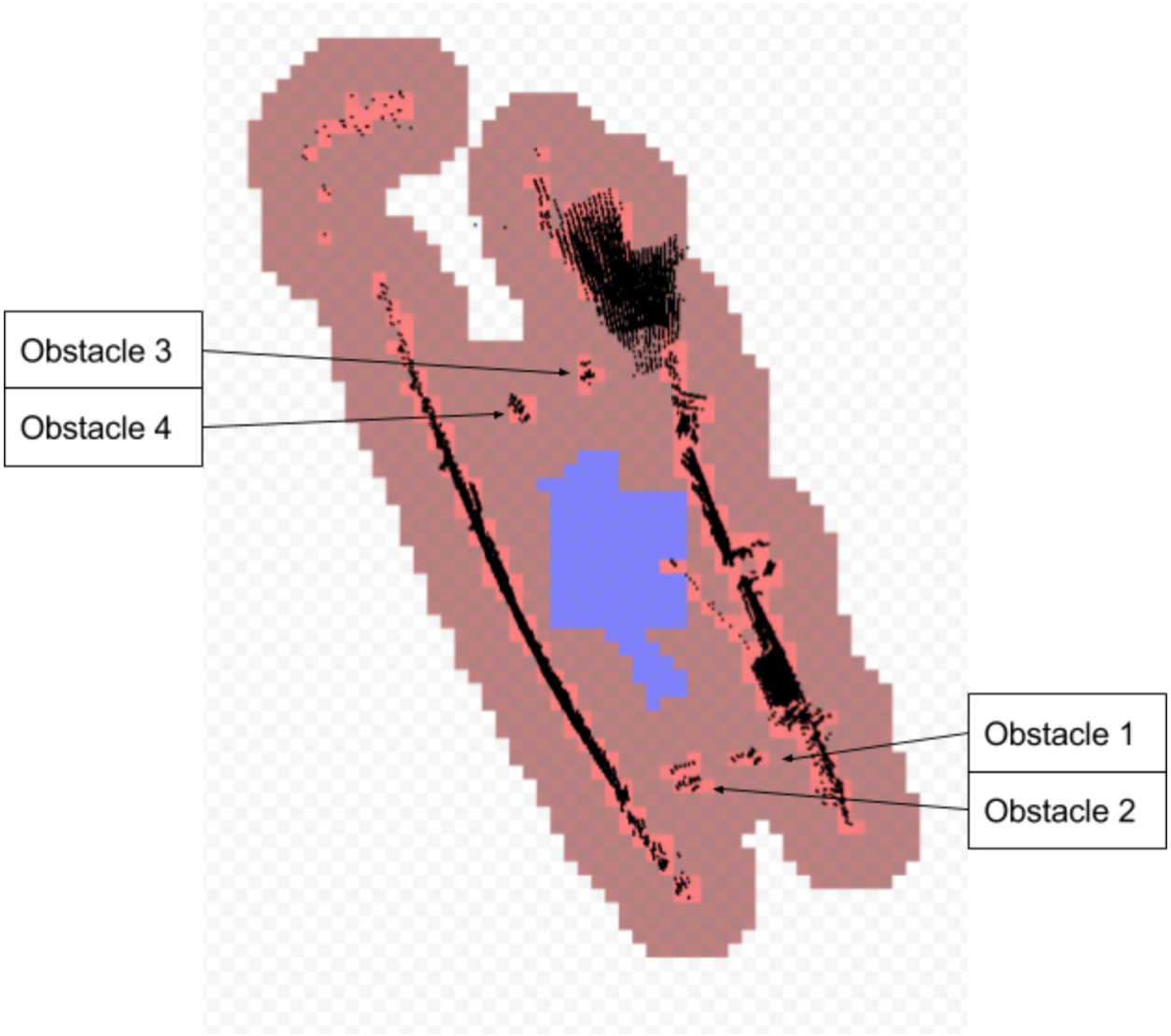


Figure 85: Voxel overlay of small obstacles test environment

An interesting note is the light pollution being emitted by the freezer. This can be seen on the right of the above diagram as a strip of red obstacle voxels. However this did not affect the final traversal. This could also be removed by decreasing the pointcloud cleaning threshold.

Figure 86 shows the evolution of the voxel grid as the robot tries to remove all yellow exploration voxels.

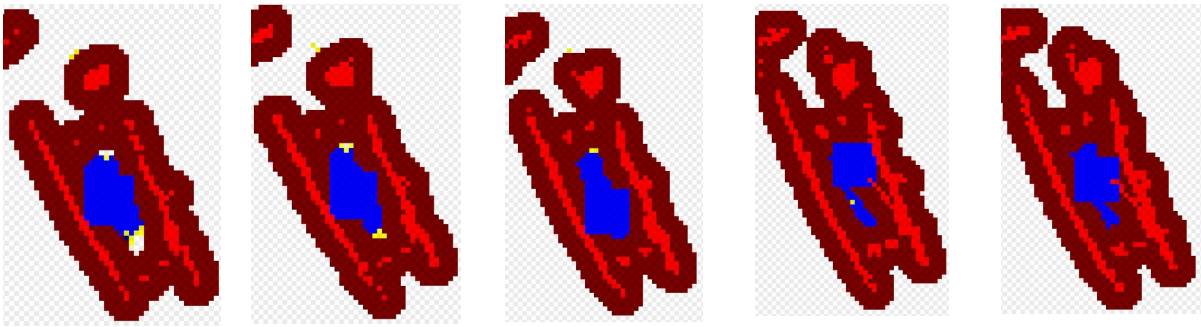


Figure 86: Voxel grid over time of small obstacles test environment

Figure 87 shows the floor response identifying the minimal number of scans needed to map the environment.

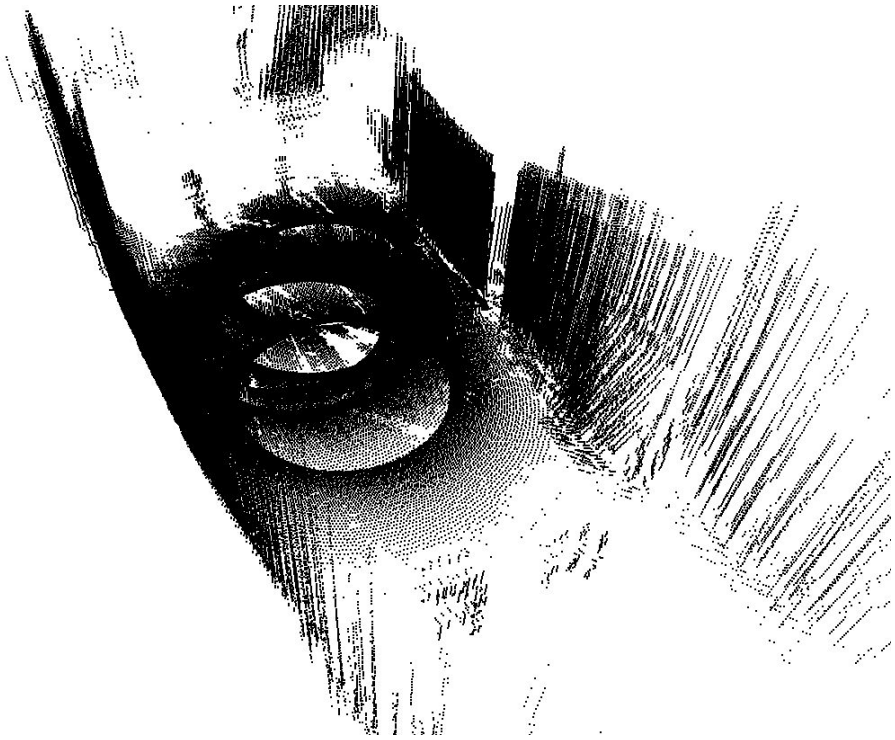


Figure 87: Point cloud of small obstacles test environment

6.2.5 Hallway

The main objective of this test was to test registration in a more sparse environment. One of the most interesting points in this scan is the mirror which can be see on the right of Figure 88.

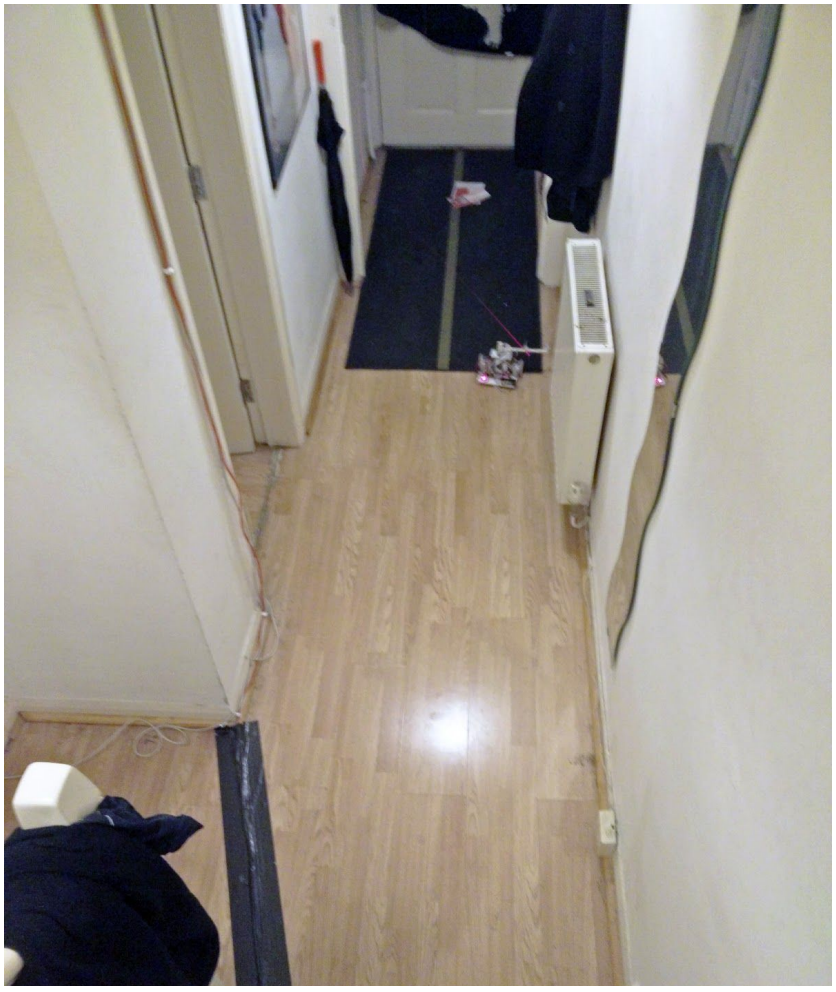


Figure 88: Photo of robot in the hallway test environment

As we can see here on the right hand side of the scan in Figure 89 and Figure 90 the mirror is rendered as a blank unknown area. This is because the light which was reflected traveled too far and was therefore too dim to be picked up. Paired with the black floor test this shows the dangers of reflective environments.

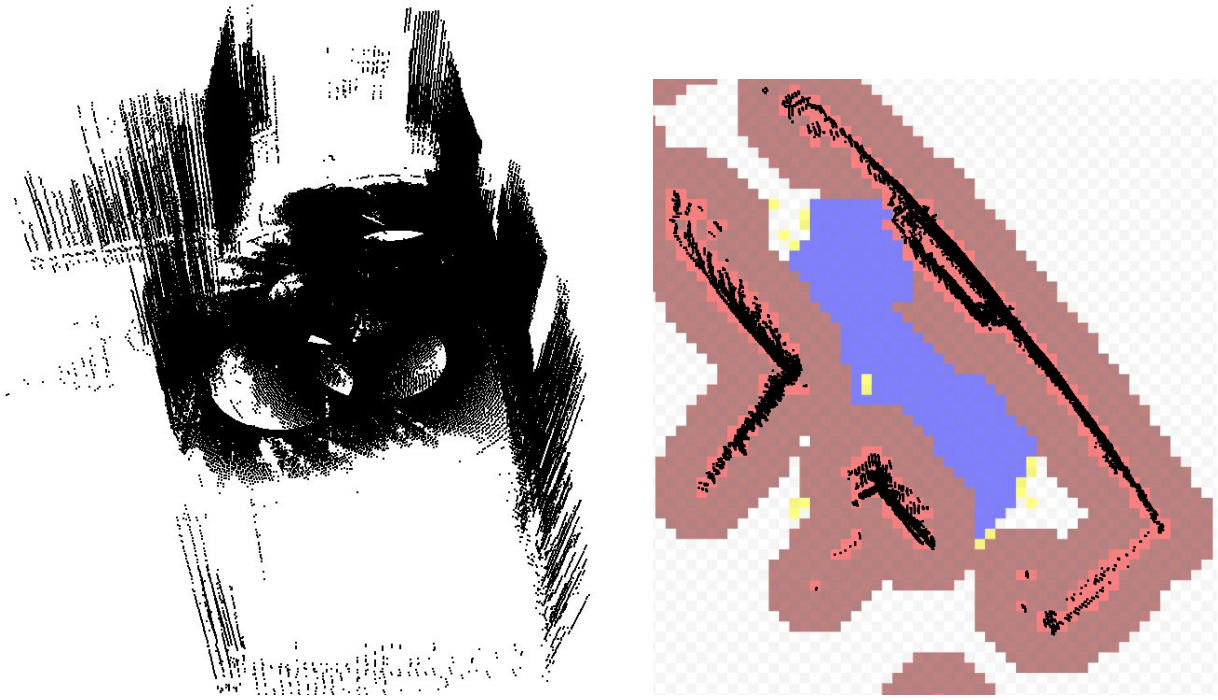


Figure 89 Pointcloud and voxel overlay of the corridor environment

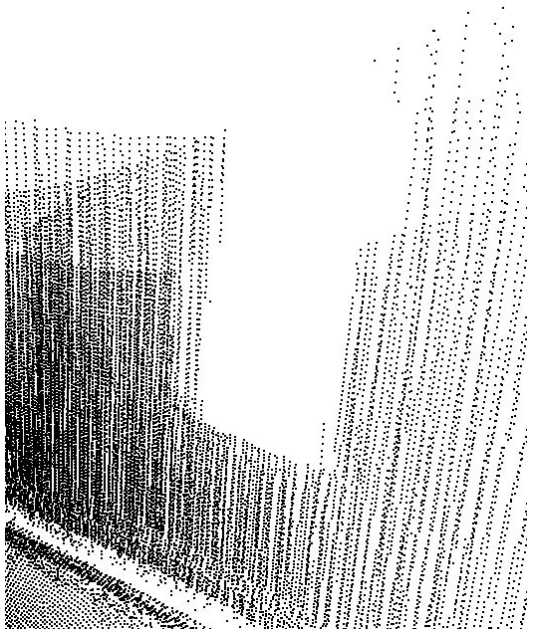


Figure 90: Point cloud close up of mirror

6.2.6 Small space

An interesting point in this test is that the robot has mapped the underside of the bed frame. This is obscured from a human's point of view and shows how the robot could be used in small hard to reach environments. This robot also quit exploration successfully as it had created a full map of the environment with no exploration voxels left which can be seen in Figure 93.

Figure 91: Photo of small space test environment

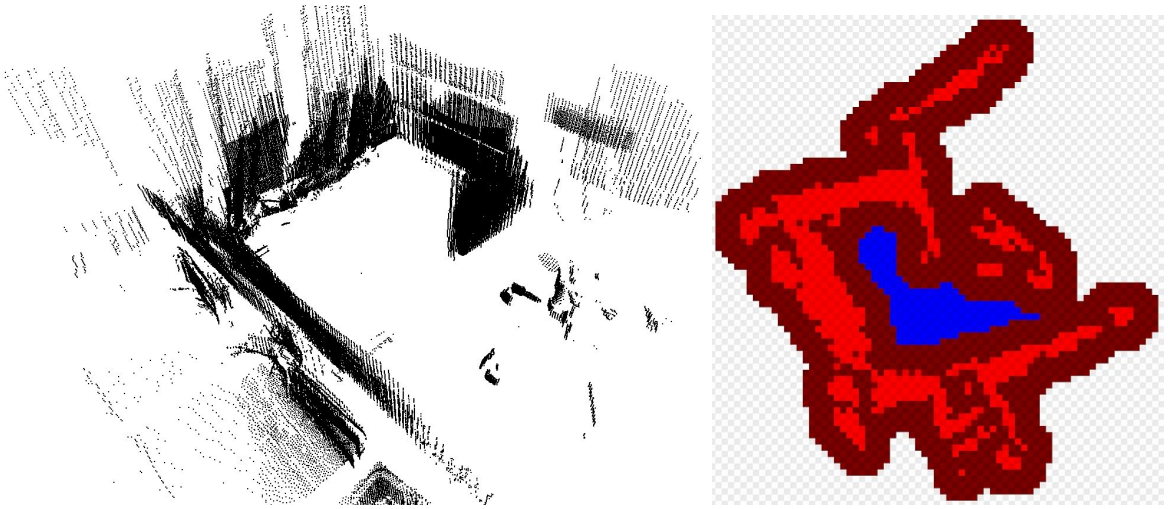


Figure 92/93: Point cloud and vox grid of small space test environment

6.2.7 Exterior

The system is not designed for outdoor use, however to show how the system can adapt it has been tested outside. Figure 94 shows the exterior environment test environment. It has been placed on a rough concrete path which increased the error in the built in odometry. Regardless of which, it still managed to take 4 scans before exiting due to light pollution from neighbouring windows and the moon.

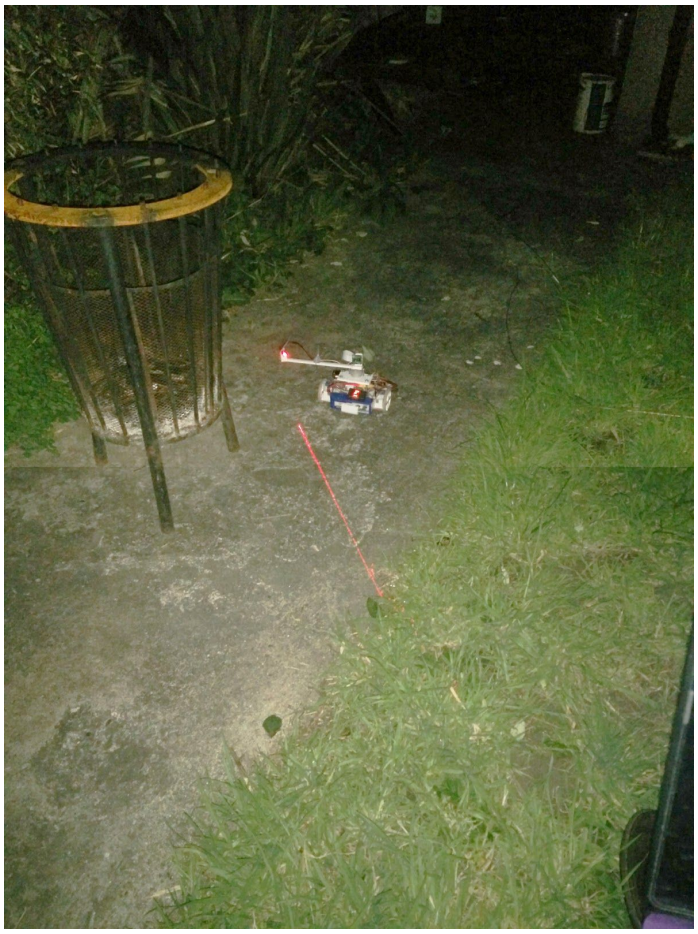


Figure 94: Photo of robot in exterior test environment

Figure 95 shows the effect of the rough terrain on the scanner. Regardless of the little amount of data it captured, it still managed to traverse the environment safely and pick out key details such as the leg of the fireplace.

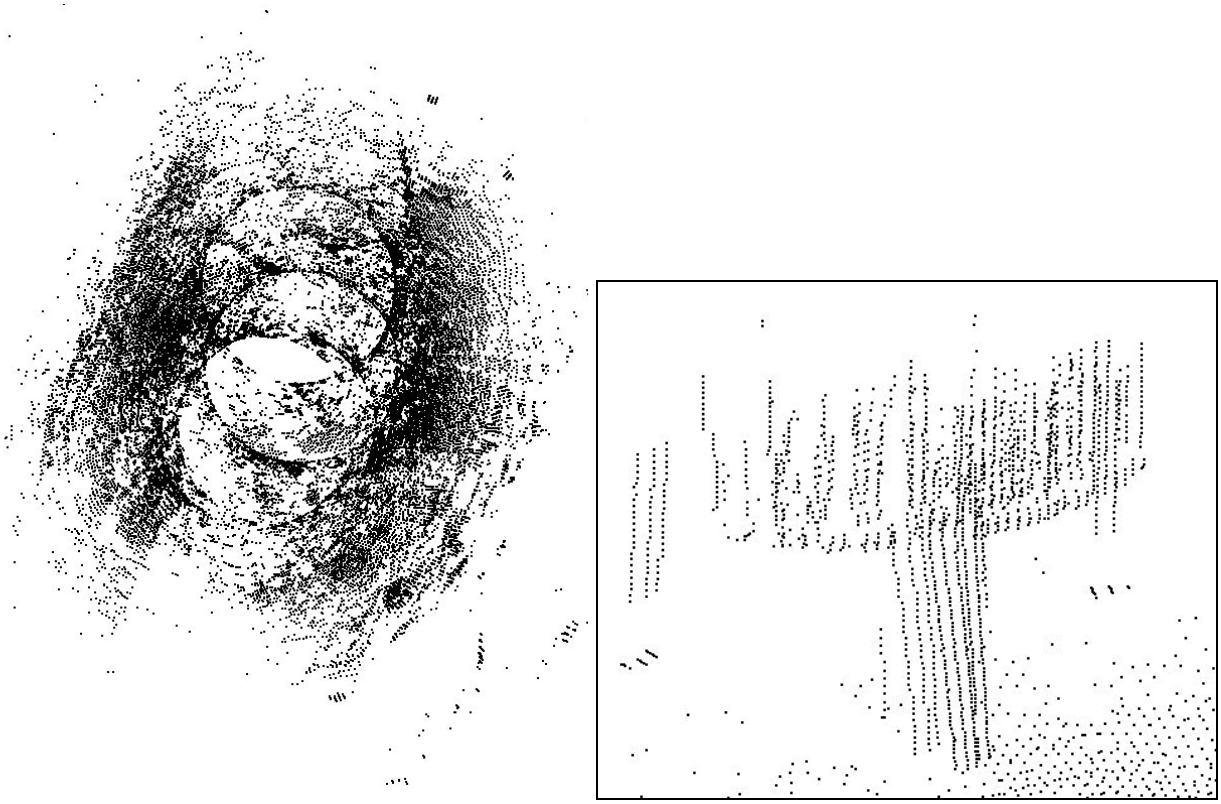


Figure 95: Point cloud and detail of exterior test environment

6.3 - System Speed

“The robotic platform should be able to scan an environment within a reasonable amount of time. For example, one scan and one 30cm route plan traversal should take no more than two minutes to complete.”

This is highly dependent on the environment however throughout testing, a single exploration loop takes anywhere between a minute and a minute and a half to complete.

As this is well inside the requirements, the scanner's resolution could be increased to get a higher density point cloud or the total scanning area increased. Figure 96 shows the timing breakdown of an average exploration loop.

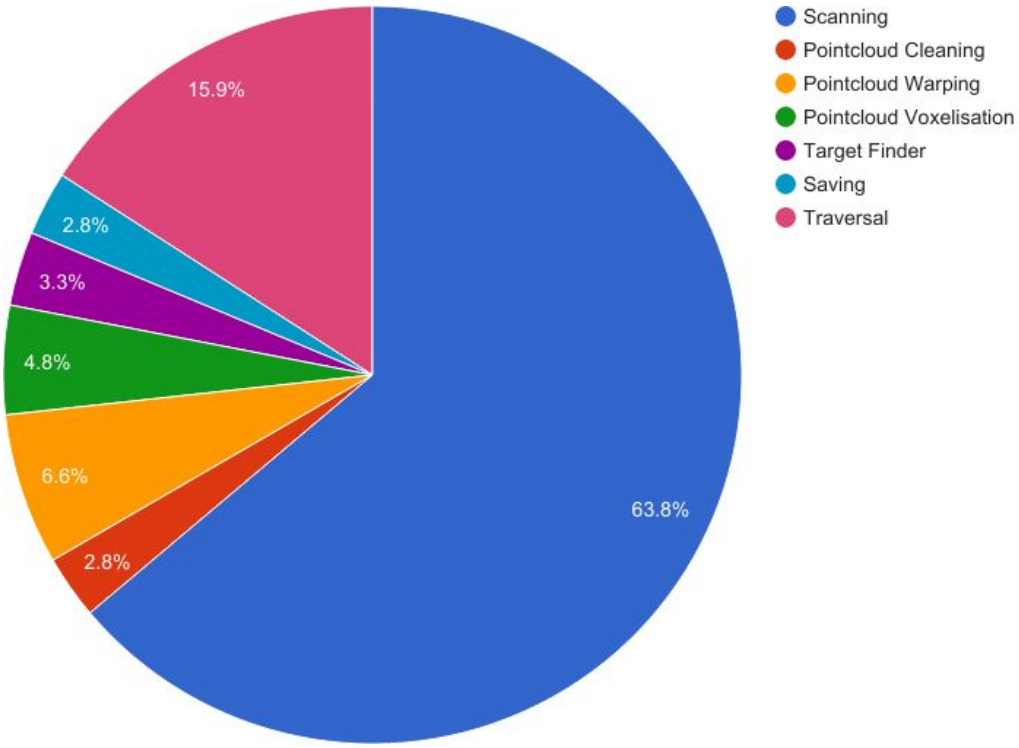


Figure 96: Pie Chart showing exploration speed breakdown

6.4 - System Price

“The entire system’s component value should be no more than £100”

Figure 97 shows the price of the project in total. Some items are estimated as their cost is unknown or unquantifiable. The total price falls below £100 and therefore satisfies the price constraint of the project.

Item	**Quantity**	**Price**
Raspberry Pi 3	1	£34.00
Laser module	1	£2.73
USB power bank*	1	~ £20.00
5-12v stepper motors	3	£2.50
Raspberry Pi Camera V2	1	£25.19
Printed parts, consumables, etc	N/A	~ £10.00
	Total	£99.42

Figure 97: Table breakdown of cost

7 - Limitations

These limitations were gathered during testing.

7.1 - Environmental factors

The main environmental limitations of the system are as follows:

The environment needs to have enough interest points less than 1.5m apart. This is to ensure correct registration between scans.

Surfaces should not be more than 50% transparent or specular. This is to ensure that the laser's first reflection is registered and not the transparent shadow ray or the reflected ray.

Surfaces should be light enough to stay within the identification threshold. The darker the environment the lower the laser identification threshold has to be set and the more noise that gets introduced.

Environment should not change whilst mapping is in progress. This would not put the robot at risk but it could either ruin registration or mark voxels as obstacles even after the obstacles were moved.

The environment should be relatively clean as to not introduce obstacles which would fall below thresholds and therefore get incorrectly identified. For instance rugs or threads less than 5mm in diameter may not be registered correctly.

The environment cannot have any ambient light brighter than the laser. This is crucially important as the current system measures red intensity only and therefore is susceptible to external light sources. This could be rectified with a colour based laser identification algorithm.

7.2 - Trade offs

Figure 98 shows the main three trade offs were price, run time and development time. The project aim was to optimise the price and the development time sacrificing the run time.

Figure 98: Diagram showing main trade offs

The main constraint for the run time versus the development time was the language choice. Python's objective is minimise development time. However to achieve this it sacrifices run time. The main bottlenecks are accessing Python's built in data structures. Because of this some were replaced with faster 3rd party structures such as Numpy. However if more time was available then more structures could be converted resulting in a faster run time. In an extreme case the entire system could be re-written in a lower level language such as C for unbeatable run time.

The hardware was mainly constrained by price. For instance smaller, more expensive motors would have provided a higher resolution and would have targeted some of the artefacts that the cheap stepper motors generated.

8 - Future development

8.1 Bug fixes

One of the main problems highlighted during testing was the inability to avoid cliffs. This could be rectified by giving any unknown area shadow. However this would make exploration voxels unreachable and therefore the target finder or the navigation module would need to be changed. As well as this there are slight errors in the A* algorithm that need to be addressed.

8.2 Upgrades

One of the main features this system needs is an automatic calibration tool. This is for calibrating the scanner automatically so that any minor hardware changes are accounted for. This could be done by placing the robot on a flat surface then comparing the real world result with a flat plane. The difference between the two could be used to drive the robot's parameters.

If the current software was kept, the first module that would need streamlining is the voxel grid analyser which iterates through the voxel grid multiple times. This could be converted to a single iterative operation which gets applied once to each voxel. The route planner also needs to be fixed so that the routes being found are not only safe but optimal too.

To make the system more resilient to external light sources, the laser identification step in the scanner class could take the laser’s wavelength into account.

To ensure that the hardware errors are minimal, the chassis could be 3D printed as one piece. This would remove play inherited from the temporary fittings used in the current construction. A more customisable chassis could also accommodate a castor wheel in place of the skid which would allow the robot to traverse over rougher environments.

The hardware could also be upgraded with more reliable batteries and lasers. This might incur a slight increase in cost but would also increase the reliability and reduce the weight of the system as a whole. The scanner system could be improved by adding a torsion spring to reduce turntable play and identifying the laser by wavelength to minimise external light source error.

8.3 New features

One key feature that might be useful in visualisation and analysis is a skinning tool which would convert the point cloud data into either shells or solid objects by reducing points and adding faces. This could drive content aware scanning through object recognition aiding speed and accuracy.

A drastic change if starting this project from scratch would be to write it in C. This would increase development time but decrease run time. This would also make the robot more diverse as it could handle larger datasets without incurring as much of a time penalty compared to Python.

Another interesting development would be to allow the robot to traverse non-flat environments. For instance, the system could be mounted on an all terrain platform and be used to explore exterior environments.This would require a new chassis, drivetrain and power supply as well as modification to the voxelization and scanner modules to correctly identify safe voxels.

9 - Critical Reflection

The main error I couldn't debug was the sporadic hardware failure. This caused more problems than expected when testing. This shows the importance of early debugging to ensure reliability. In the future I plan to run modular tests throughout development and store results for better reflection and debugging.

The research material I used throughout this project was found on an as-needed basis. In hindsight, the system could have been improved dramatically by gathering up more background material prior to the development stage. Striking a better balance between development and research would have ensured that lots of problems could have been avoided resulting in a more stable system.
This could also lead into using more packages earlier on in the project. This might have avoided reinventing the wheel at certain stages such as the voxelization module could have been substituted with the voxelization functionality inside PCL's Python wrappers.

During testing, I found that the file transfer and visualization was quite slow. This wouldn't be a make or break issue if it weren't for the large amounts of test data I had to process. Therefore in hindsight I would have made this system more efficient by implementing an auto-exporter to my desktop machine and therefore have a faster testing turnaround time.

During the development stage I found the tangible milestones set out in meetings and in the initial plan were invaluable for keeping track of progress. These helped me stay motivated and utilise my time efficiently. Figure 99 from my initial plan shows an excellent representation of how I could work on multiple problems simultaneously. This kept the project interesting and dynamic.

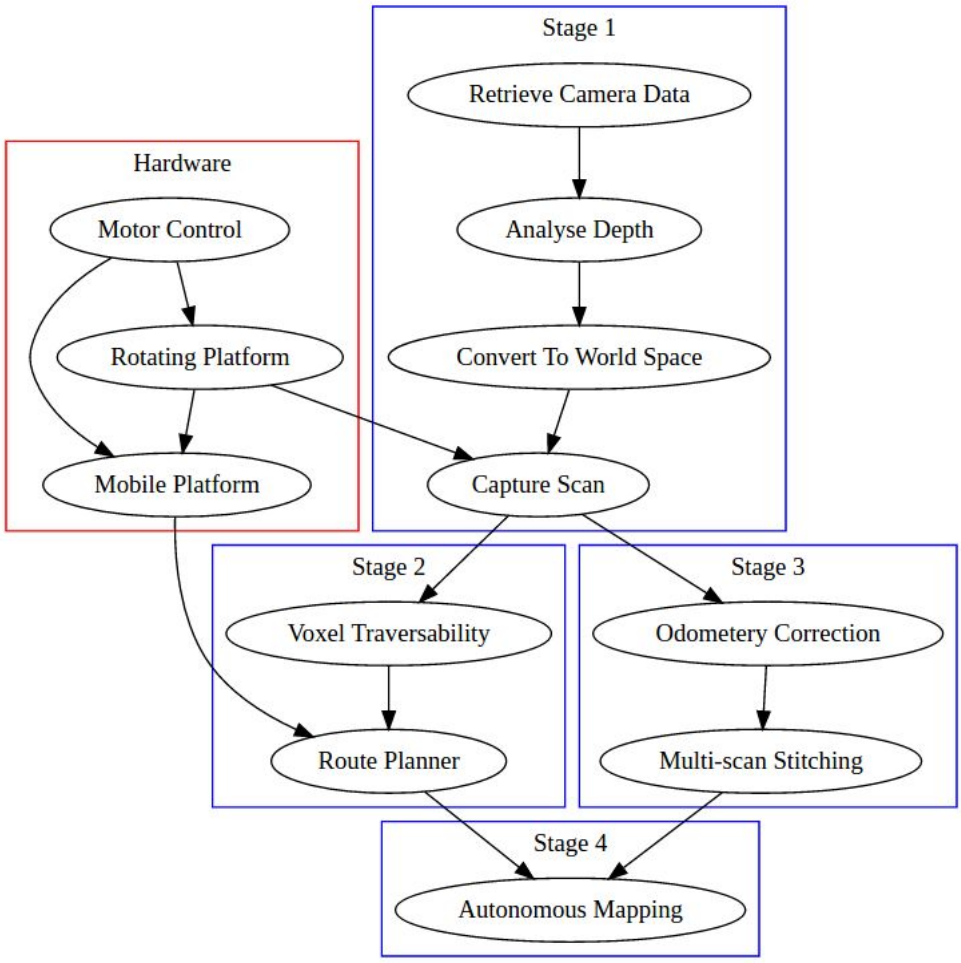


Figure 99: Work Plan Dependency Graph

10 - Conclusion

In conclusion, the project successfully completed the objectives set out in the initial plan. It exceeded expectations when it came to precision and adaptability. It also managed to retrieve a remarkable amount of information about the environment whilst still being affordable and easy to manufacture.

The solution outlined in this document is an original and unique approach to creating an small, affordable SLAM system, and this report provides enough background research and implementation notes to replicate this solution.

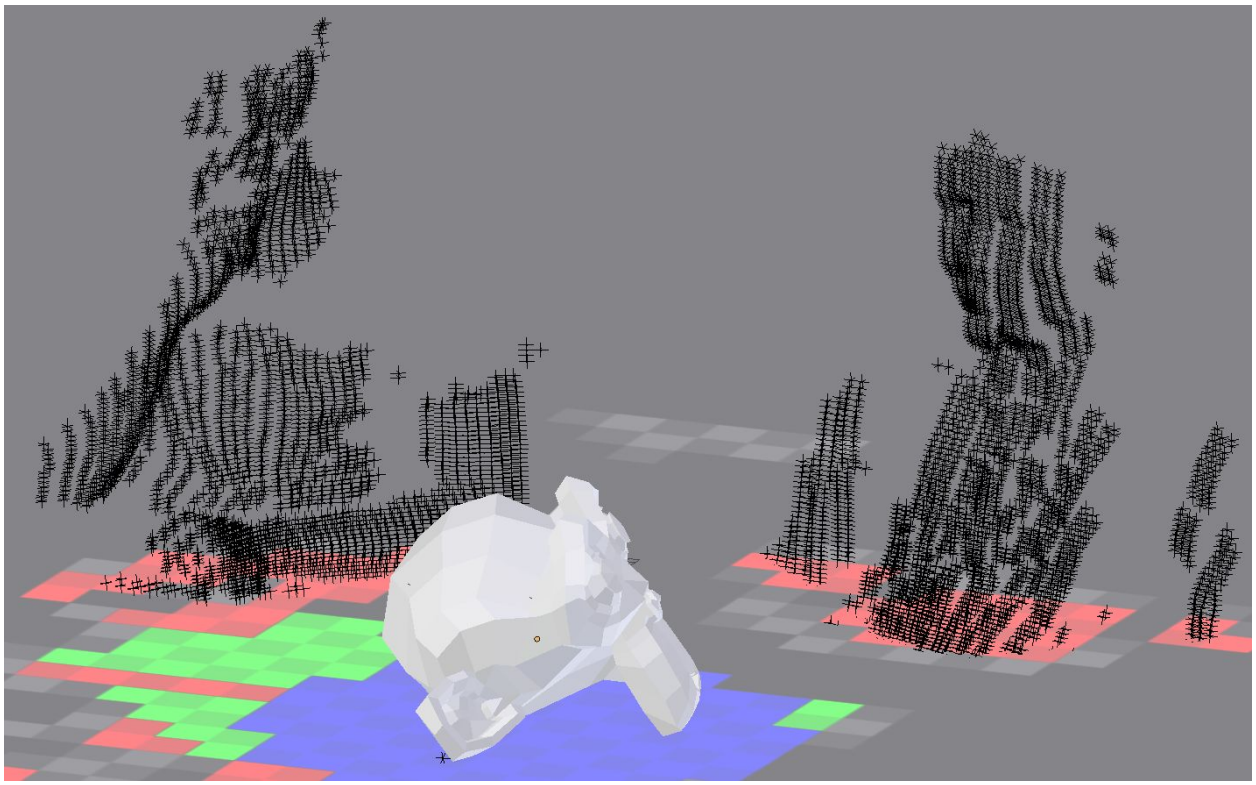


Figure 100: Test scan of desk environment

References

All references below are based on the IEEE referencing standard.

[1] S. Martin, "Laser Triangulation SLAM Initial Plan", Cardiff, 2017.

[2] R. Mayer, Scientific Canadian, 1st ed. Vancouver: Raincoast Books, 1999.

[3] "Kinect hardware", Developer.microsoft.com, 2017. [Online]. Available: https://developer.microsoft.com/en-us/windows/kinect/hardware. [Accessed: 23- Apr- 2017].

[4] "Sweep Product Specification", 2017. [Online]. Available: https://s3.amazonaws.com/scanse/SWEEP_DATA_SHEET.pdf. [Accessed: 01- May- 2017].

[5] P. E. Hart, N. J. Nilsson and B. Raphael, "A Formal Basis for the Heuristic Determination of Minimum Cost Paths," in *IEEE Transactions on Systems Science and Cybernetics*, vol. 4, no. 2, pp. 100-107, July 1968.

[6] S. Riisgaard and M. Rufus Blas, "SLAM for Dummies", 2017. [Online]. Available: https://ocw.mit.edu/courses/aeronautics-and-astronautics/16-412j-cognitive-robotics-spring-2005/projects/1aslam_blas_repo.pdf. [Accessed: 01- May- 2017].

[7] Smith, Self, Cheesman: Estimating uncertain spatial relationships in robotics

[8] S. Rusinkiewicz and M. Levoy, "Efficient variants of the ICP algorithm," Proceedings Third International Conference on 3-D Digital Imaging and Modeling, Quebec City, Que., 2001, pp. 145-152.

[9] R. Rusu and S. Cousins, "3D is here: Point Cloud Library (PCL)", IEEE International Conference on Robotics and Automation (ICRA), 2011.

[10] "Point Cloud Library (PCL): PCL API Documentation", Docs.pointclouds.org, 2017. [Online]. Available: http://docs.pointclouds.org/1.7.2/. [Accessed: 23- Apr- 2017].

[11] J. Vilches, "Interview with Raspberry's Founder Eben Upton", TechSpot, 2017. [Online]. Available: http://www.techspot.com/article/531-eben-upton-interview/. [Accessed: 25- Apr- 2017].

[12] "Raspberry Pi Documentation", Raspberrypi.org, 2017. [Online]. Available: https://www.raspberrypi.org/documentation/. [Accessed: 23- Apr- 2017].

[13] "Cython 0.21.2 : Python Package Index", Pypi.python.org, 2017. [Online]. Available: https://pypi.python.org/pypi/Cython/0.21.2. [Accessed: 23- Apr- 2017].

[14] "strawlab/python-pcl", GitHub, 2017. [Online]. Available: https://github.com/strawlab/python-pcl. [Accessed: 23- Apr- 2017].

[15] "Installing PCL on Raspberry Pi", Gist, 2017. [Online]. Available: https://gist.github.com/chatchavan/c758f1568d35bbf6dd75. [Accessed: 23- Apr- 2017].

[16] "Overview — NumPy v1.13.dev0 Manual", Docs.scipy.org, 2017. [Online]. Available: https://docs.scipy.org/doc/numpy-dev/. [Accessed: 23- Apr- 2017].

[17] "picamera — Picamera 1.13 Documentation", Picamera.readthedocs.io, 2017. [Online]. Available: http://picamera.readthedocs.io. [Accessed: 23- Apr- 2017].

[18] "Python Imaging Library (PIL)", Pythonware.com, 2017. [Online]. Available: http://www.pythonware.com/products/pil/. [Accessed: 23- Apr- 2017].

[19] "What is Arduino", Arduino.org, 2017. [Online]. Available: http://www.arduino.org/learning/getting-started/what-is-arduino. [Accessed: 23- Apr- 2017].

[20] "Intel® Galileo Board Documentation", Software.intel.com, 2017. [Online]. Available: https://software.intel.com/en-us/iot/hardware/galileo/documentation. [Accessed: 23- Apr- 2017].

[21] D. Hodges, "Fundamental theory and applications", IEEE Transactions on Circuits and Systems, vol. 46, no. 1, p. 102, 1999.

[22] S. Monk, "Overview | Adafruit's Raspberry Pi Lesson 10. Stepper Motors | Adafruit Learning System", Learn.adafruit.com, 2017. [Online]. Available: https://learn.adafruit.com/adafruits-raspberry-pi-lesson-10-stepper-motors/. [Accessed: 23- Apr- 2017].

[23] "Principles of operation of two phase stepper motors", Pc-control.co.uk, 2008. [Online]. Available: https://www.pc-control.co.uk/step-motor.htm. [Accessed: 25- Apr- 2017].

[24] "Laser radiation: safety advice - GOV.UK", Gov.uk, 2017. [Online]. Available: https://www.gov.uk/government/publications/laser-radiation-safety-advice/laser-radiation-safety-advice. [Accessed: 23- Apr- 2017].

[25] "Camera Module - Raspberry Pi Documentation", Raspberrypi.org, 2017. [Online]. Available: https://www.raspberrypi.org/documentation/hardware/camera/README.md. [Accessed: 01- May- 2017].

[26] "New camera mode released - Raspberry Pi", Raspberry Pi, 2017. [Online]. Available: https://www.raspberrypi.org/blog/new-camera-mode-released/. [Accessed: 23- Apr- 2017].

[27] "Ad Hoc setup in RPi 3", Raspberrypi.stackexchange.com, 2017. [Online]. Available: https://raspberrypi.stackexchange.com/a/49792. [Accessed: 23- Apr- 2017].

[28] J. A. Baltar, E. Delgado and A. Barreiro, "Mark-based vision for 3D vehicle tracking using least-squares and kalman filter," *Proceedings World Automation Congress, 2004.*, Seville, 2004, pp. 313-318.